МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М.В.ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

Курынин Р.В., Машечкин И.В., Терехин А.Н.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Конспект лекций

МОСКВА
2006

Содержание

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 (Россия) и на факультете компьютерных наук университета Ватерлоо (Онтарио, Канада).

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, досконального 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 (1).

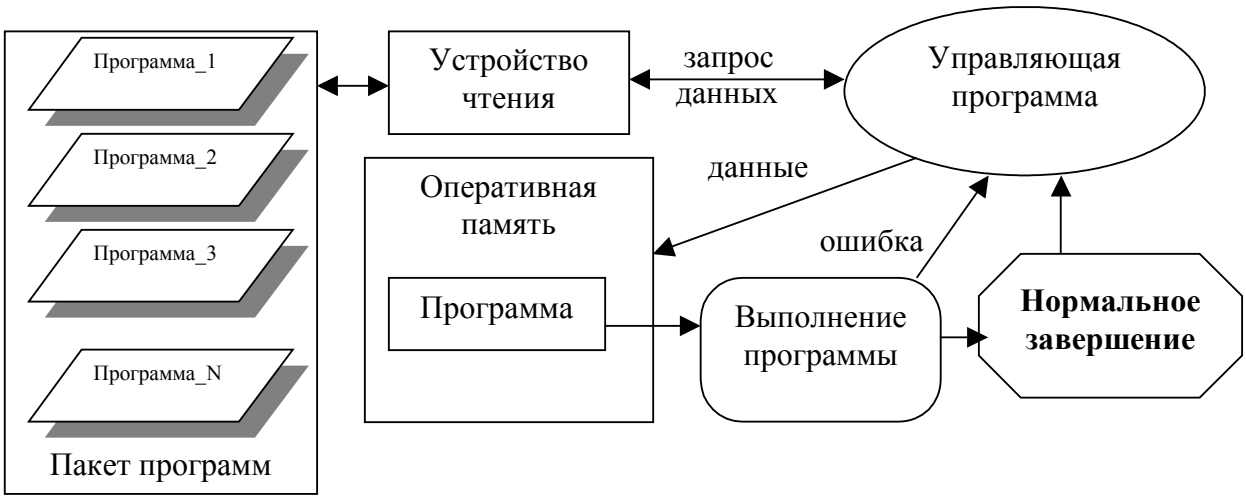


Рис. 1. Пакетная обработка заданий.

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* (1).

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**.

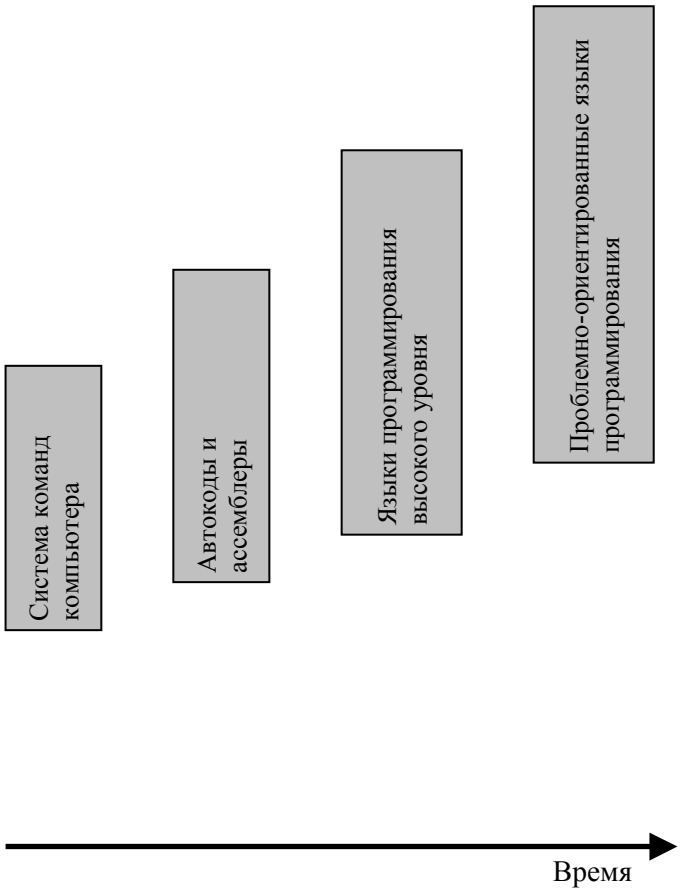


Рис. 2. Развитие языков и систем программирования.

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.

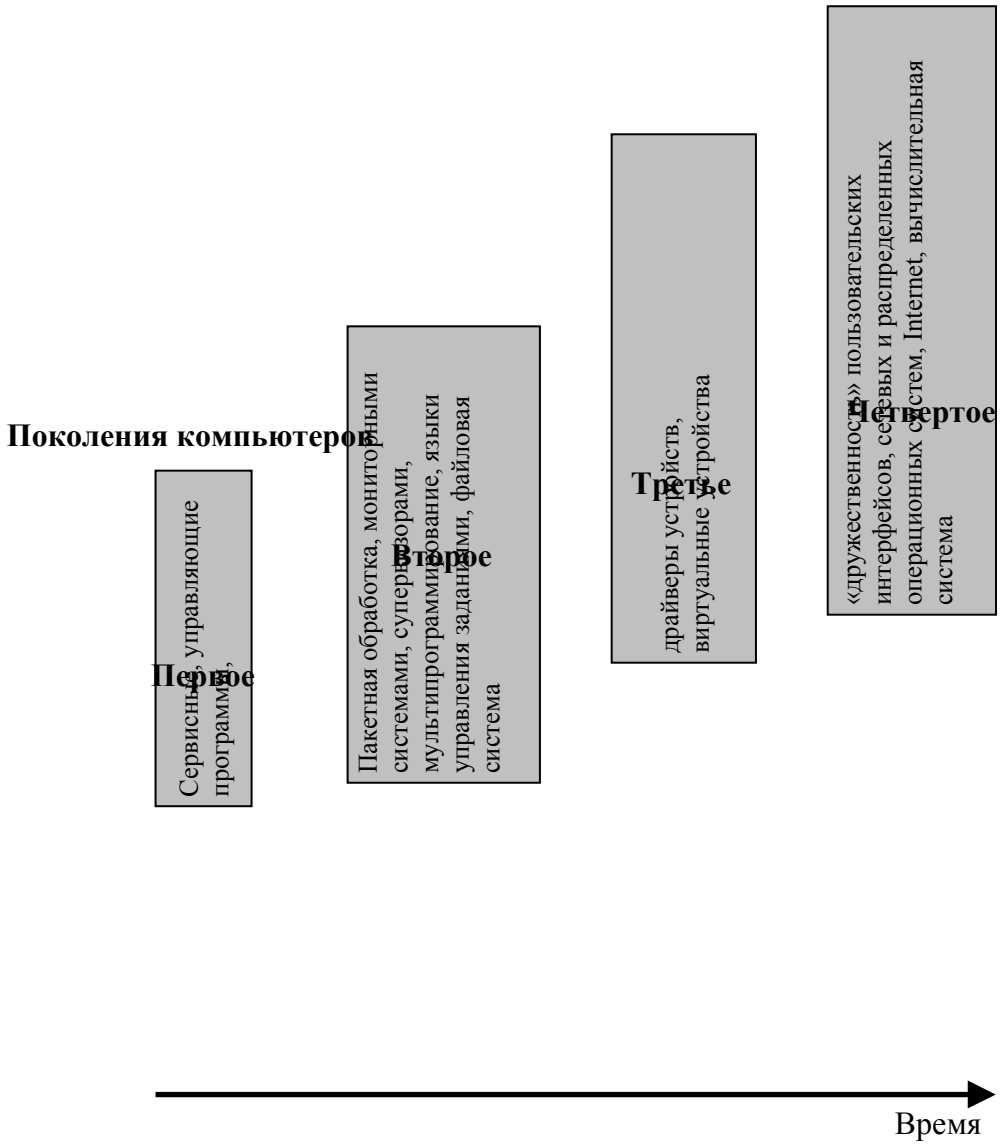


Рис. 3. Этапы эволюции.

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 (1).

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 (1.1.1). 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 (1.1.3).

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 рисунке 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.

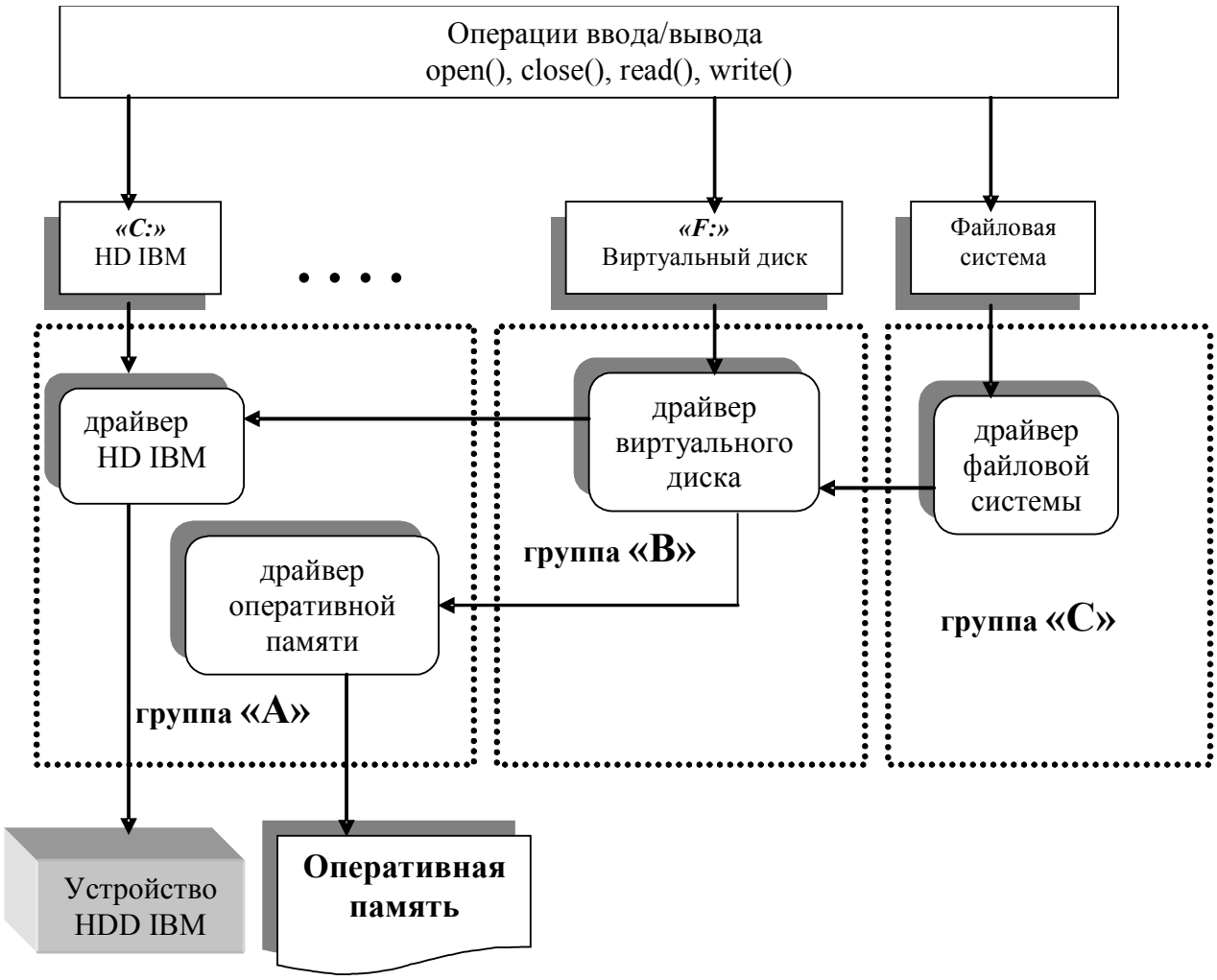


Рис. 6. Схема организации ввода-вывода в системе.

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 (1.1.5).

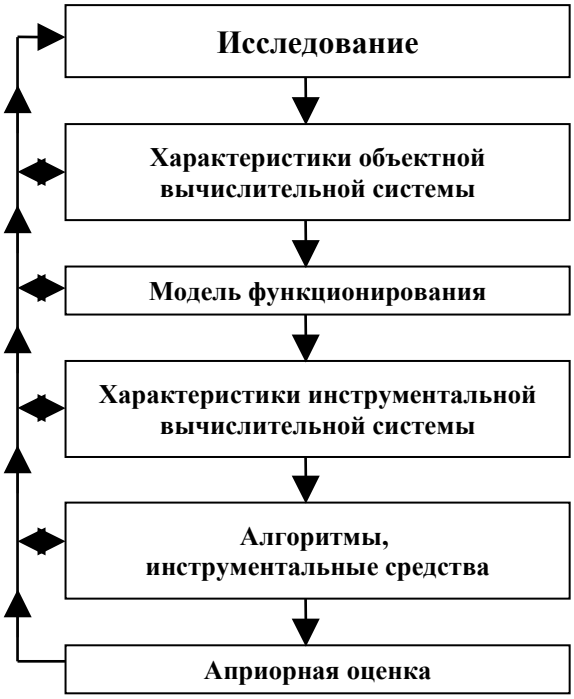


Рис. 7. Этапы проектирования.

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 (1.1.5). 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:

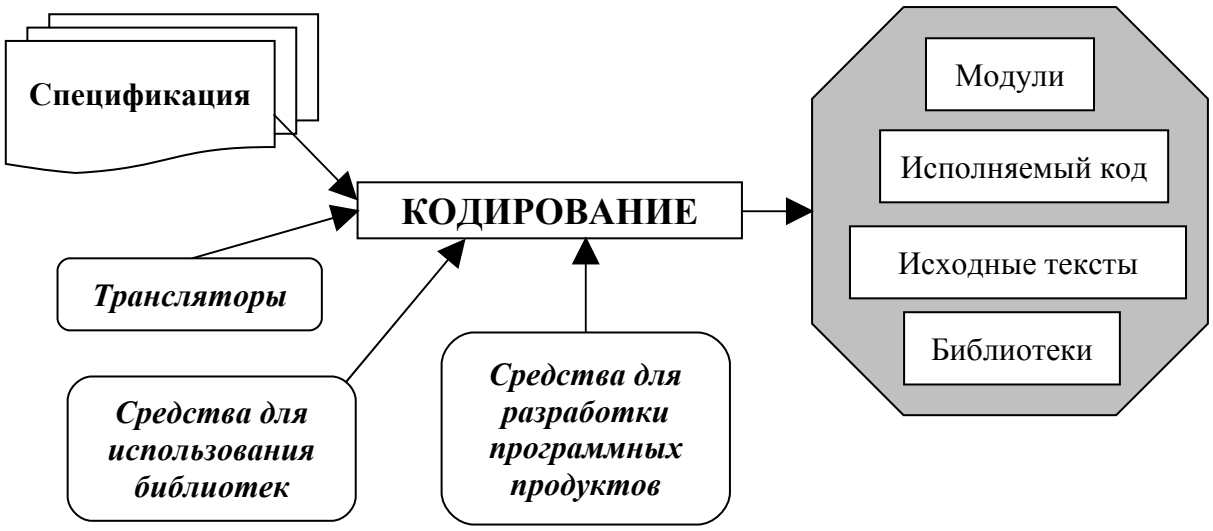


Рис. 8. Кодирование.

- 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 (1.1.5). 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**).

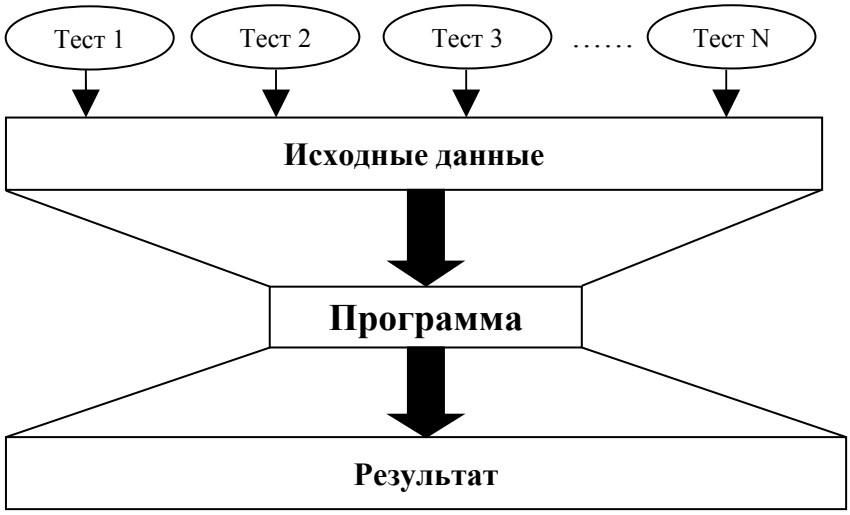


Рис. 9. Тестирование.

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** (1.1.5) — представляет разработку в виде строго линейной последовательности этапов, каждый из которых заканчивается фиксацией результата, и только после этого начинается последующий.

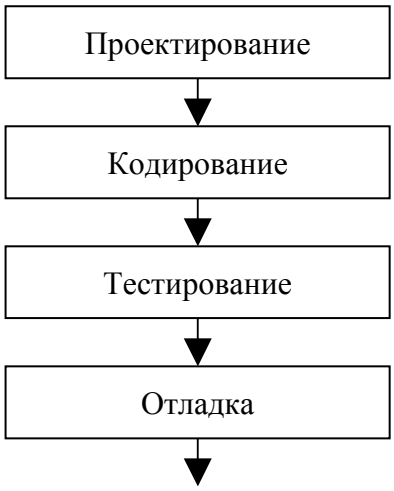


Рис. 10. Каскадная модель.

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** (1.1.5), которая в общем случае, предоставляет возможность осуществления анализа полученных на этапе результатов и возврат к любому предшествующему этапу.

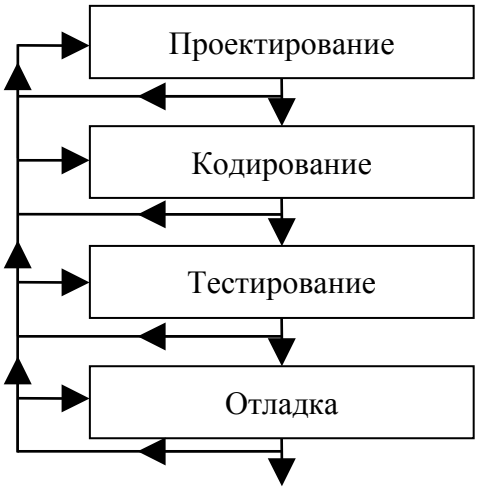


Рис. 11. Каскадная итерационная модель.

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 (1.1.5).

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.

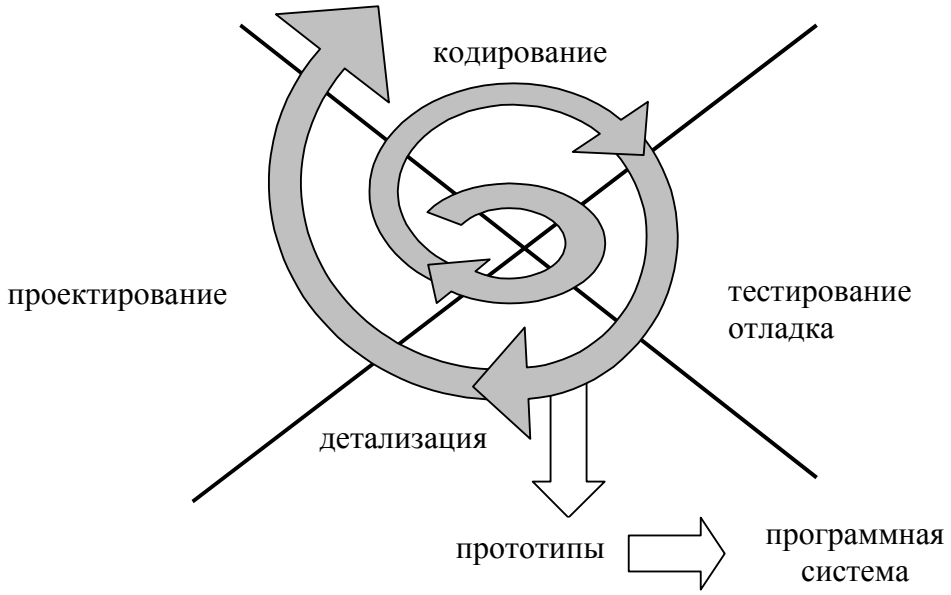


Рис. 12. Спиральная модель.

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 (1.1.6). 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.

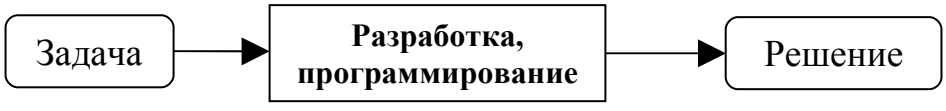


Рис. 13. Первый этап развития прикладных систем.

Второй этап — развитие систем программирования и появление средств создания и использования библиотек программ (1.1.6).

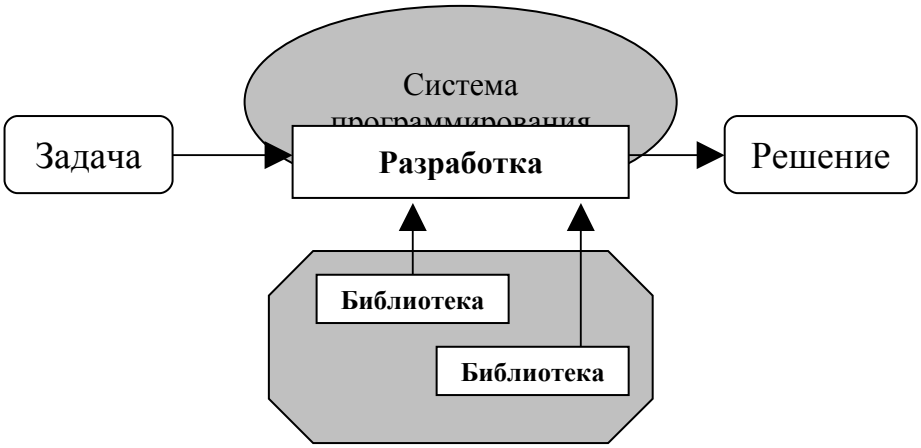


Рис. 14. Второй этап развития прикладных систем.

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 (1.1.6), 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;

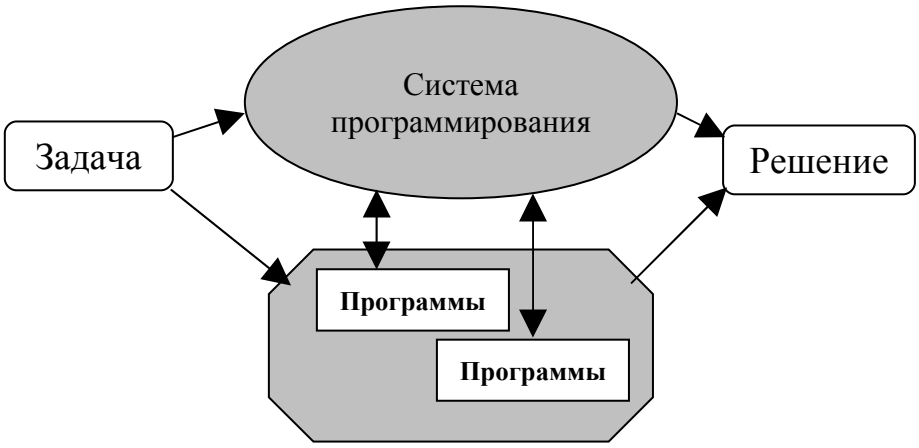


Рис. 15. Третий этап развития прикладных систем.

- 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 (1.1.6), предназначенный для автоматизации офисной деятельности или пакет MathCAD (1.1.6), предназначенный для решения задач, связанных с математическими и техническими расчетами.

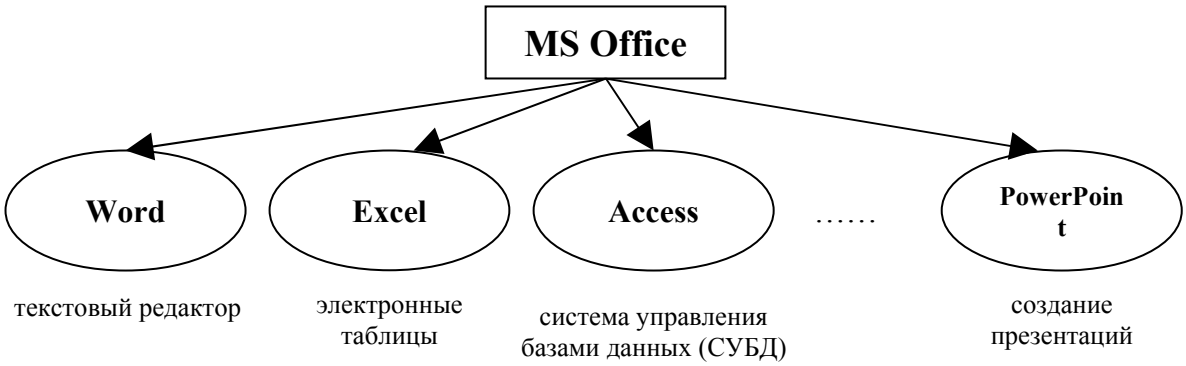


Рис. 16. Пакет программ Microsoft Office.

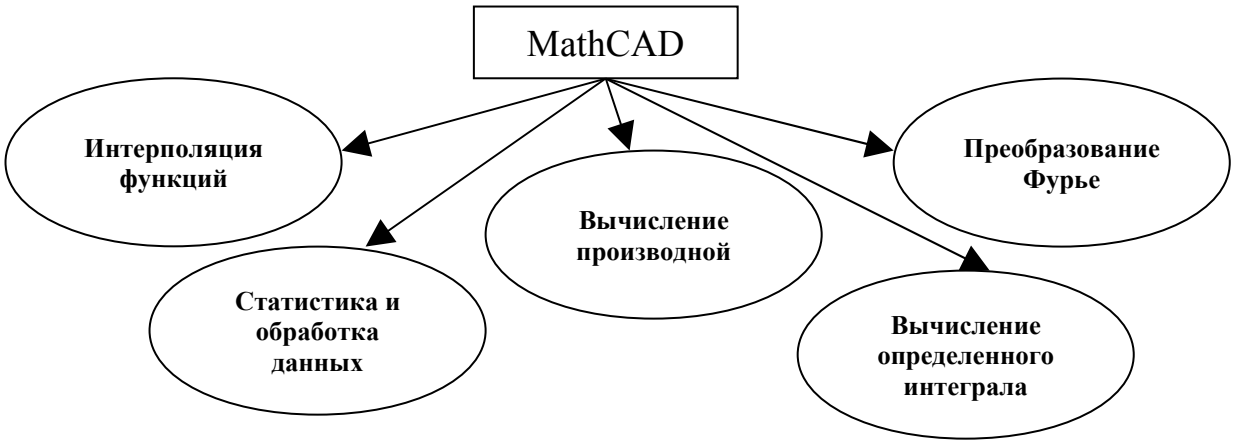


Рис. 17. Пакет MathCAD.

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 использующими 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 (1.1.6). 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.

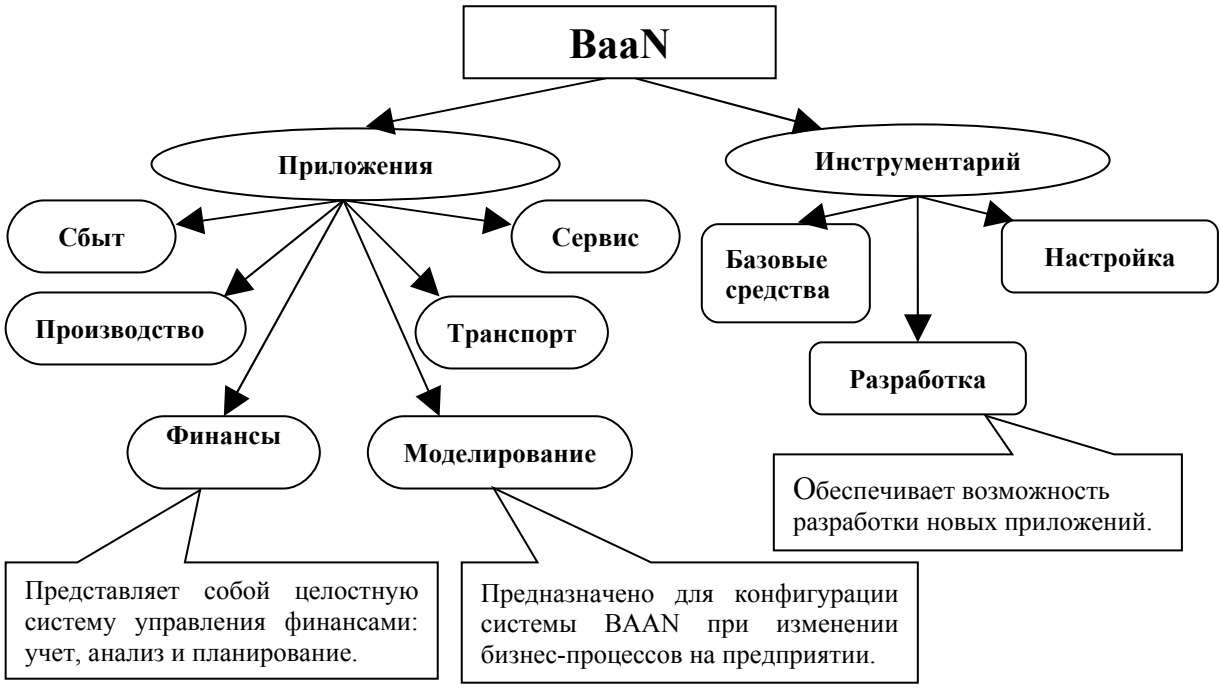


Рис. 18. Система BaaN.

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, литература

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 (1.1.7).

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. 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 прошлого века может вправе считаться сроком зарождения современной вычислительной техники. С этой датой связана публикация американского математика венгерского происхождения Джона фон Неймана (John Von Neumann) 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 должны были быть 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 или предложениях, полученных совместно Моучли, Эккертом и фон Нейманом, был подготовлен и опубликован за авторством только Джона фон Неймана. Распространение данного отчета в научной среде породило появление "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**: все поступающие и обрабатываемые компьютером данные кодируются при помощи двоичных сигналов.
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.
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. Это свойство определяет возможность программной генерации команд с последующим их выполнением.

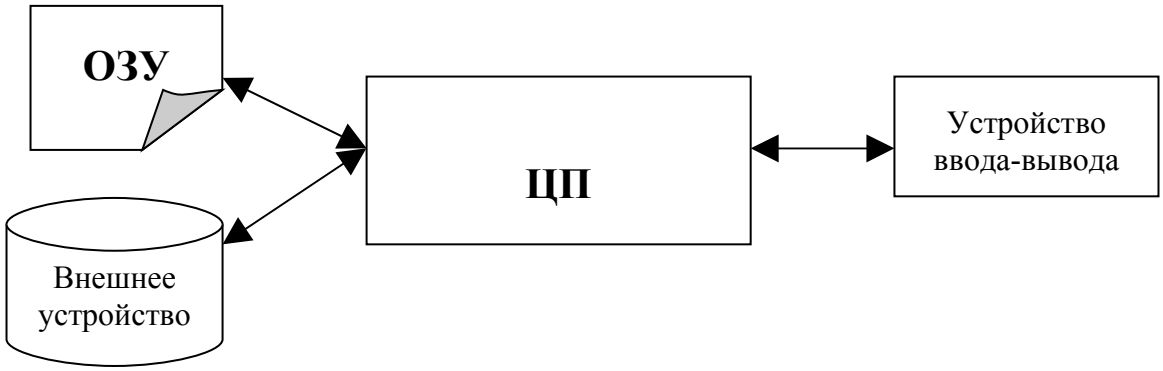


Рис. 20. Структура компьютера фон Неймана.

Рассмотрим упрощенную структуру компьютера фон Неймана (1.2.1):

- **Оперативное запоминающее устройство** (ОЗУ), или **основная память**, — устройство хранения данных, в котором находится исполняемая в настоящее время программа.
- **Внешние устройства —** программно управляемые устройства, входящие в состав компьютера, т.е. устройства, с которыми выполняемая программа может обмениваться 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 компьютера предусмотрены команды обмена с внешними устройствами.

Современные компьютеры по многим показателям не соответствуют модели фон Неймана. Ниже мы рассмотрим базовые структурные и функциональные особенности современных компьютеров (1.2.1), уделив особое внимание организации компьютера, как системы, объединяющей разнородные по назначению и производительности аппаратные компоненты, работающей под управлением операционной системы. Скорости обработки информации в процессоре, доступа к данным, размещенным в оперативной памяти, 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.

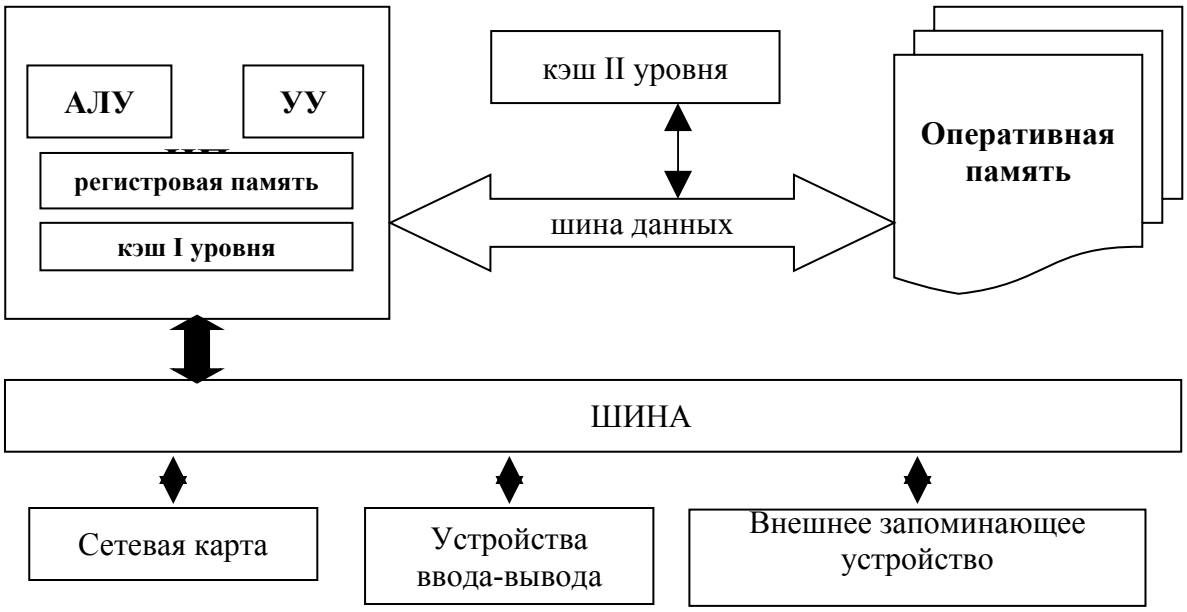


Рис. 21. Базовая архитектура современных компьютеров.

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.

1.2.2 Оперативное запоминающее устройство

Оперативное запоминающее устройство (RAM — Random-Access Memory) — это устройство хранения данных компьютера, в котором находится исполняемая в данный момент программа. ОЗУ еще называют основной памятью, или оперативной памятью. Команды программы, исполняемые компьютером, поступают в процессор исключительно из ОЗУ. Хранение программы, которая выполняется в настоящее время компьютером, является основным назначением оперативной памяти. Оперативная память состоит из **ячеек памяти**. Ячейка памяти — это устройство, в котором возможно хранение информации. Ячейка памяти может состоять из двух полей (1.2.2). Первое поле — поле машинного слова, второе — поле служебной информации (или ТЕГ). Рассмотрим назначение каждого из них.

Машинное слово	ТЕГ

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 используется термины «16-тиразрядный компьютер», или «32-хразрядный компьютер», или «64-хразрядный компьютер», это означает, что речь идет о компьютерах, оперативная память которых имеет машинные слова размером 16, 32 или 64 разряда соответственно.

Служебная информация — 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 машинном слове.

Использование в компьютере содержимого поля служебной информации может осуществляться в следующих целях.

- **Контроль целостности данных**. Содержимое поля используется для контрольного суммирования кода, размещенного в машинном слове. При каждой записи информации в машинное слово автоматически происходит контрольное суммирование и формирование содержимого поля служебной информации. 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 1.2.2 изображена ячейка памяти с 16-тиразрядным машинным словом и одноразрядным полем ТЕГа. Контрольный разряд дополняет код машинного слова до четности. Вариант А: содержимое машинного слова корректное (здесь следует отметить, что одноразрядное контрольное суммирование может "пропускать" потери пар единиц в коде машинного слова — вариант В), вариант Б — ошибка.
- **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. 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 размещенной в машинном слове информации.

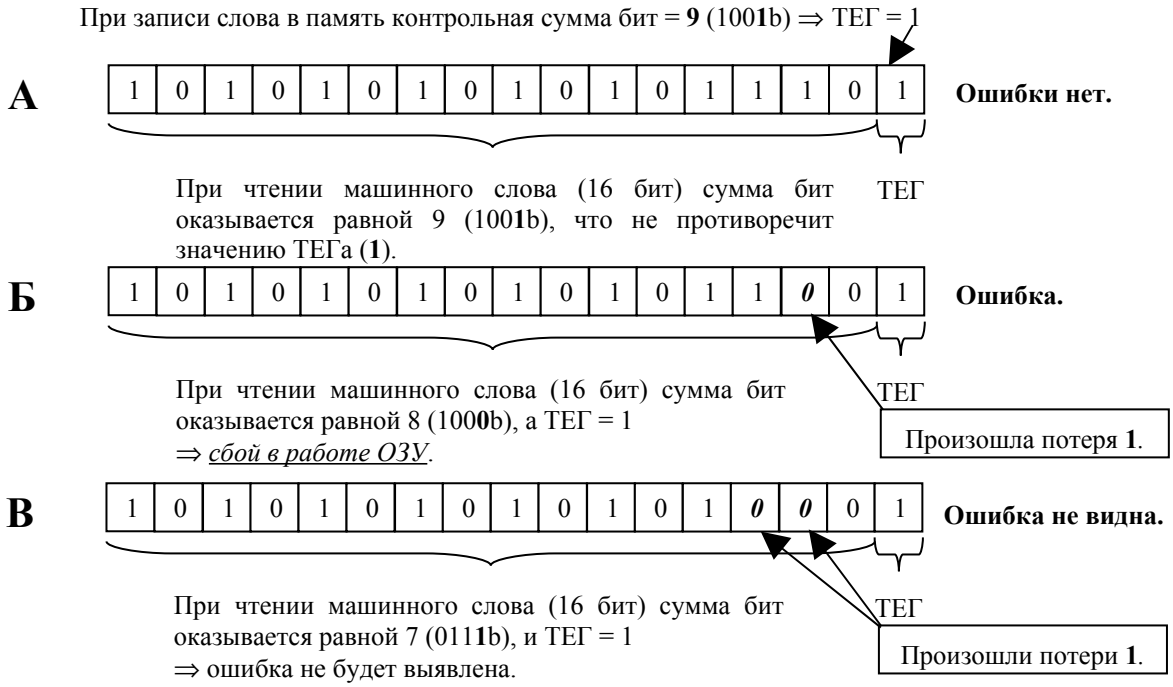


Рис. 23. Контроль четности.

- **Контроль доступа к машинным типам данных**. Развитием контроля за семантикой информации, размещенной в оперативной памяти, является появление средств контроля за использованием компьютерных типов 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 качеств 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 — $\mathbf{t_{access}}$) — это время между запросом на чтение слова из оперативной памяти и получением содержимого этого слова. Второй параметр — длительность цикла памяти (cycle time — $\mathbf{t_{cycle}}$) — это минимальное время между началом текущего и последующего обращения к памяти. Обычно, длительность цикла превосходит время доступа ($\mathbf{t_{cycle}}>\mathbf{t_{access}}$). Реальные соотношения между длительностью цикла и временем доступа зависят от конкретных технологий, применяемых для организации ОЗУ (в некоторых ОЗУ $\mathbf{t_{cycle}}/\mathbf{t_{access}}>2$). Последнее утверждение говорит о том, что возможна ситуация, при которой для чтения N слов из памяти потребуется времени больше, чем $\mathbf{N}\times\mathbf{t_{access}}$.

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 (1.2.2, 1.2.2).

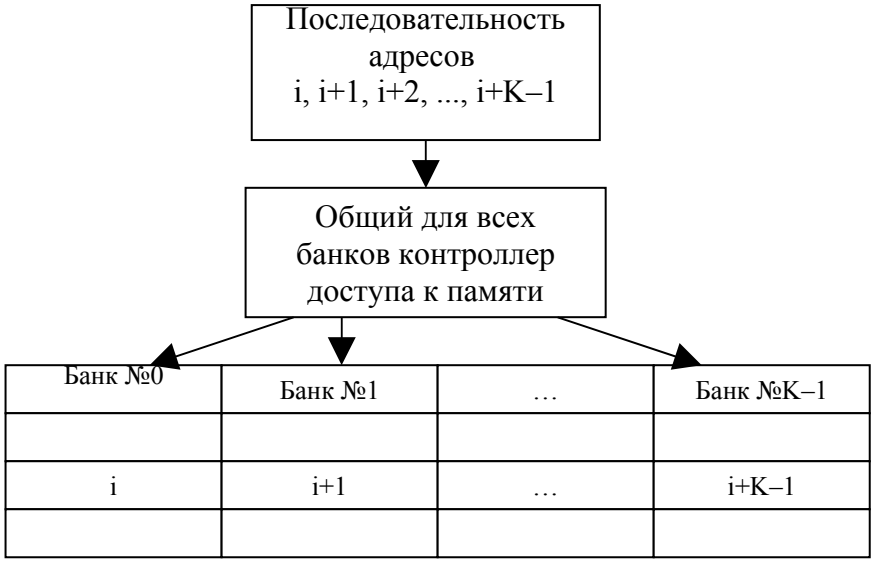


Рис. 24. ОЗУ без расслоения памяти — один контроллер на все банки.

Все ОЗУ состоит из **K** банков, каждый из которых может работать независимо. Ячейки памяти распределены между банками таким образом, что у любой ячейки ее соседи размещаются в соседних блоках. 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. Используя организацию параллельной работы банков, в идеальном случае, можно повысить производительность работы ОЗУ в **K** раз. Для этих целей необходимо использовать более сложную архитектуру системы управления памятью.

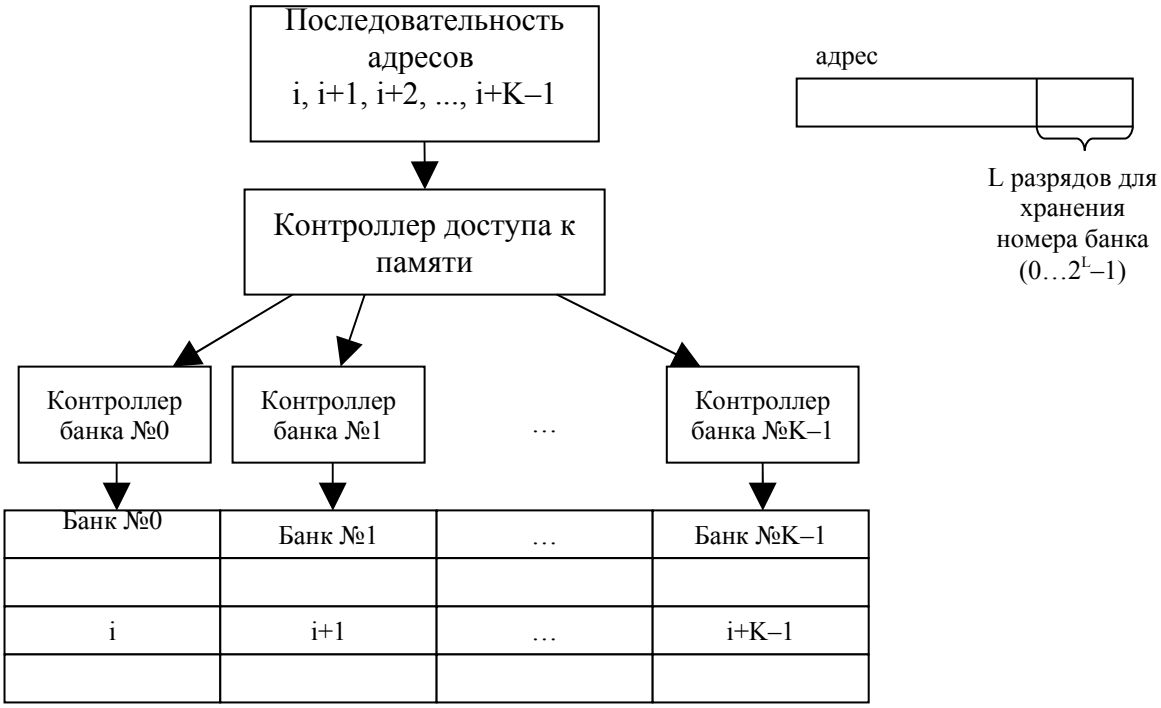


Рис. 25. ОЗУ с расслоением памяти — каждый банк обслуживает отдельный контроллер.

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

1.2.3 Центральный процессор

Процессор, или **центральный процессор** (**ЦП**), компьютера обеспечивает последовательное выполнение машинных команд, составляющих программу, размещенную в оперативной памяти. Термин «центральный процессор» соответствует ситуации сегодняшнего дня, когда современный компьютер имеет в своем составе значительное количество специализированных управляющих компьютеров. Подобные компьютеры могут осуществлять управление контроллерами устройств, быть встроены в сами устройства, выполнять специализированные операции над данными программы.

Рассмотрим основные компоненты обобщенной структурной организации центрального процессора (1.2.3).

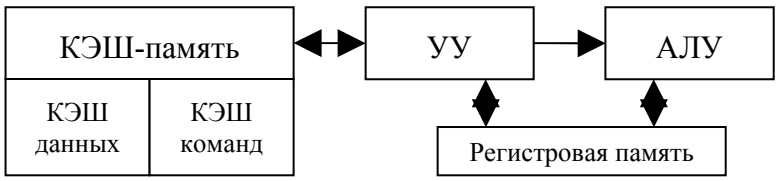


Рис. 26. Структура организации центрального процессора.

1.2.3.1 Регистровая память

Регистровый файл (register file), или ***регистровая память***, — совокупность устройств памяти процессора — т.н. регистров, предназначенных для временного хранения управляющей информации, операндов и/или результатов выполняемых команд. Регистровый файл 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. 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. 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 схему выполнения программы (1.2.3.2) в модельном компьютере.

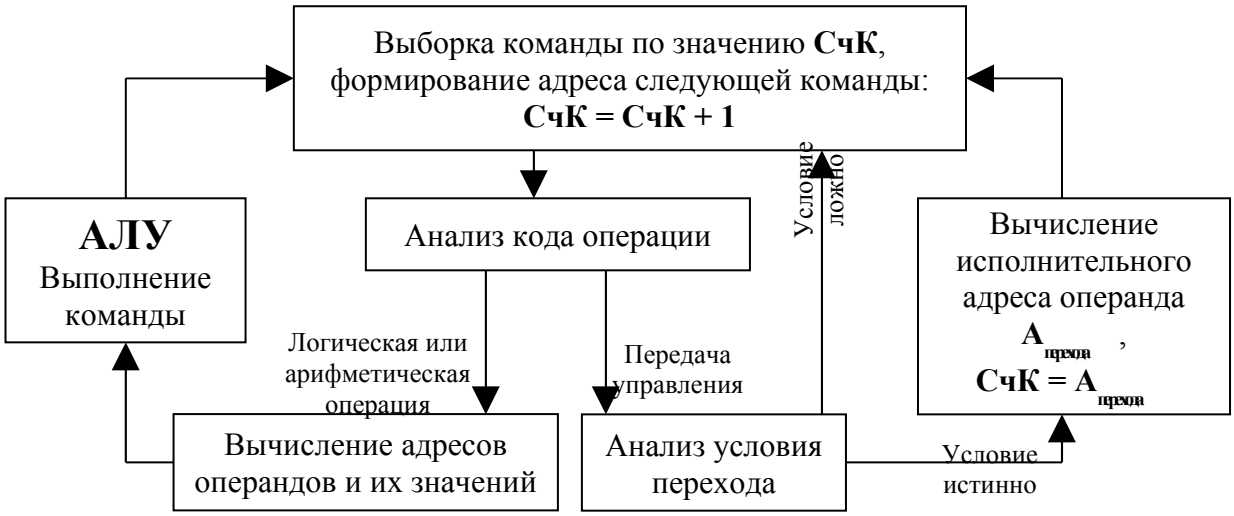


Рис. 27. Схема выполнения программы.

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, адреса соседних машинных слов отличаются на единицу. Итак, рассмотрим последовательность действий в устройстве управления процессора при выполнении программы.

1. По содержимому счетчика команд **СчК** 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.1.2.3.2.
   - 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.1.2.3.2. Иначе, вычисляется исполнительный адрес операнда $\mathbf{A_{перехода}}$, затем: $\mathbf{СчК = A_{перехода}}$, переход на п.1.2.3.2.
   - 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.1.2.3.2.

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. 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.

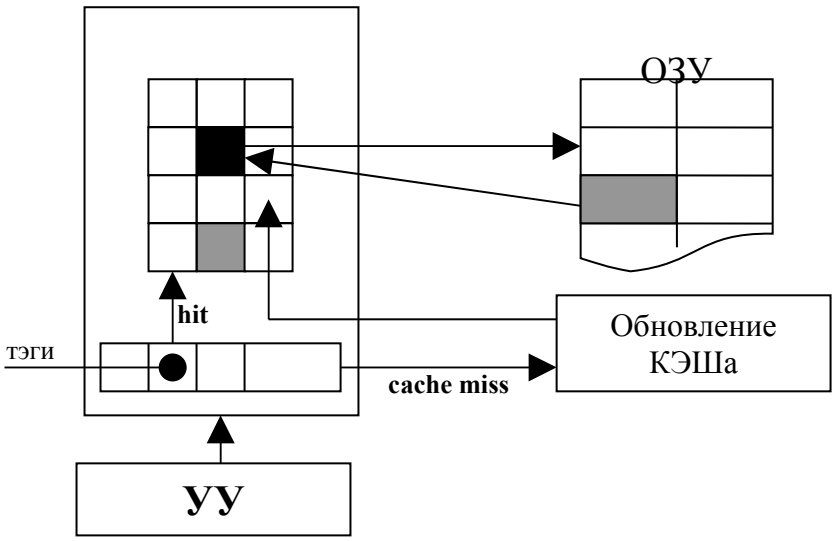


Рис. 28. Общая схема работы КЭШа.

Общая организация КЭШа следующая (1.2.3.3).

1. Условно, вся память разделяется на блоки одинакового размера. Обмен данными между КЭШем и оперативной памятью осуществляется блоками (размер блока может соответствовать машинному слову или группе машинных слов). 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. Каждому блоку КЭШа ставится в соответствие адресный тег, по содержимому которого возможно однозначно адресовать содержимое блока. 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). Если искомых данных нет в КЭШе, то фиксируется **промах** (cache miss).
3. При возникновении промаха происходит 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, 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. 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 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.

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.

Организация и использование КЭШ-памяти в процессоре развивает рабочий цикл модельного компьютера, рассмотренный выше: при выборке очередных команд, получении операндов команд и записи результатов выполнения команд в ОЗУ добавляются схемы организации использования КЭШ-памяти.

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 ошибочной ситуации, которая может 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, 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 событий, возникающих в вычислительной системе, в современных компьютерах предусмотрен аппарат прерываний.

Прерыванием называется событие в компьютере, при возникновении которого в процессоре происходит предопределенная последовательность действий. 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 всех прерываний, возникающих при функционировании вычислительной системы. Традиционно прерывания разделяются на две группы: **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 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 (1.2.3.4). 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. 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. **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. 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. Для перехода на программный этап обработки прерываний необходимо решить вопрос, как аппаратура передаст операционной системе информацию о том, какое прерывание произошло. Существует несколько моделей аппаратного решения этого вопроса.
   - Первая модель — использование специального регистра прерываний, каждый разряд которого соответствует конкретному прерыванию, 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 (1.2.3.4). Она предполагает, что имеется **главный регистр прерывания** и **периферийные**. В главном регистре прерывания выделяются разряды, индицирующие не только появление конкретных прерываний, но и разряды, индицирующие появление прерываний в периферийных регистрах. В данной модели управление передается в операционную систему на адрес входа в программу.

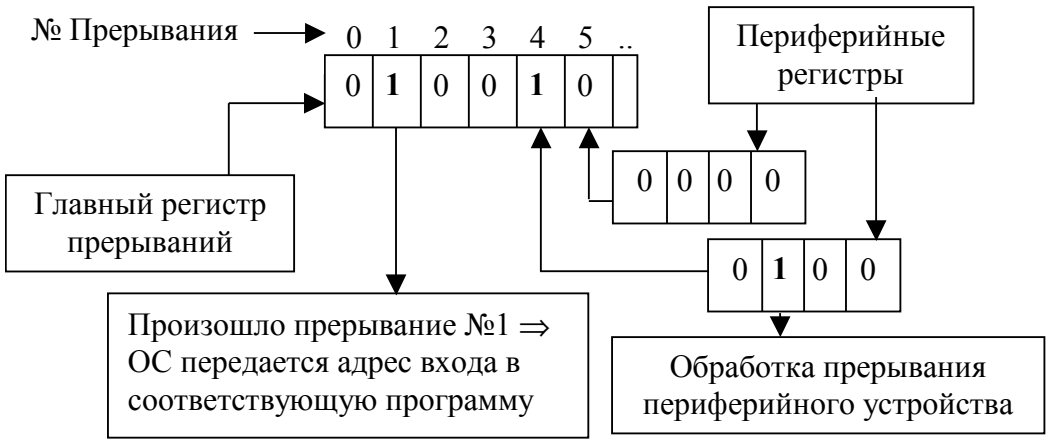


Рис. 30. Использование иерархической модели регистров прерывания.

   - Вторая модель — использование регистра слова состояния процессора. В этом случае в данном регистре резервируется часть разрядов, в которых отображается номер возникшего прерывания. Управление также передается на фиксированный адрес входа в программу обработки прерываний.
   - Третья 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 (31). При возникновении прерывания после сохранения регистров осуществляется передача прерывания по адресу, соответствующему номеру прерывания.

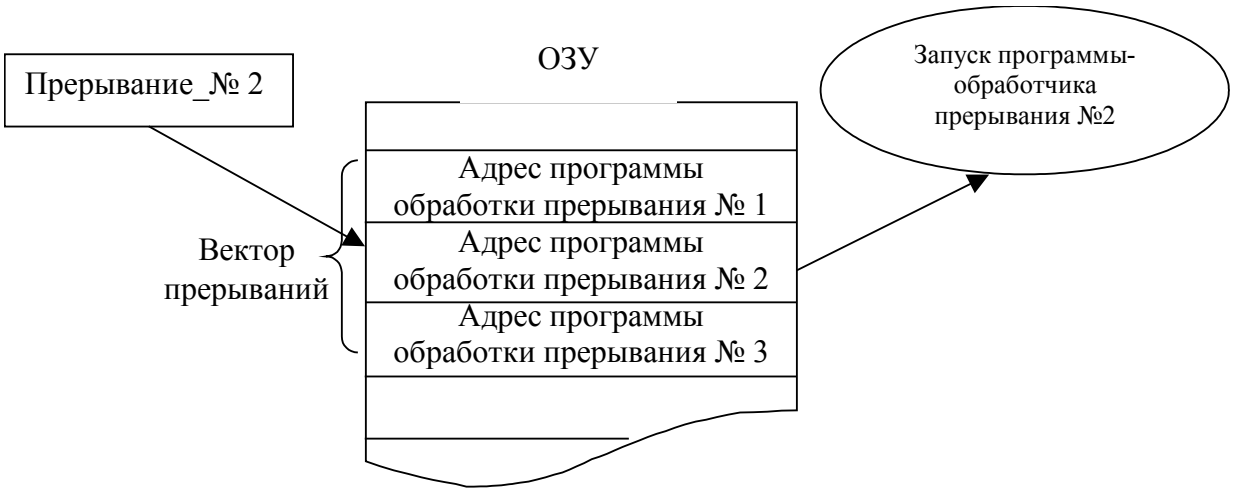


Рис. 31. Использование вектора прерываний.

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 (результат аппаратного сохранения 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 (1.2.3.4).

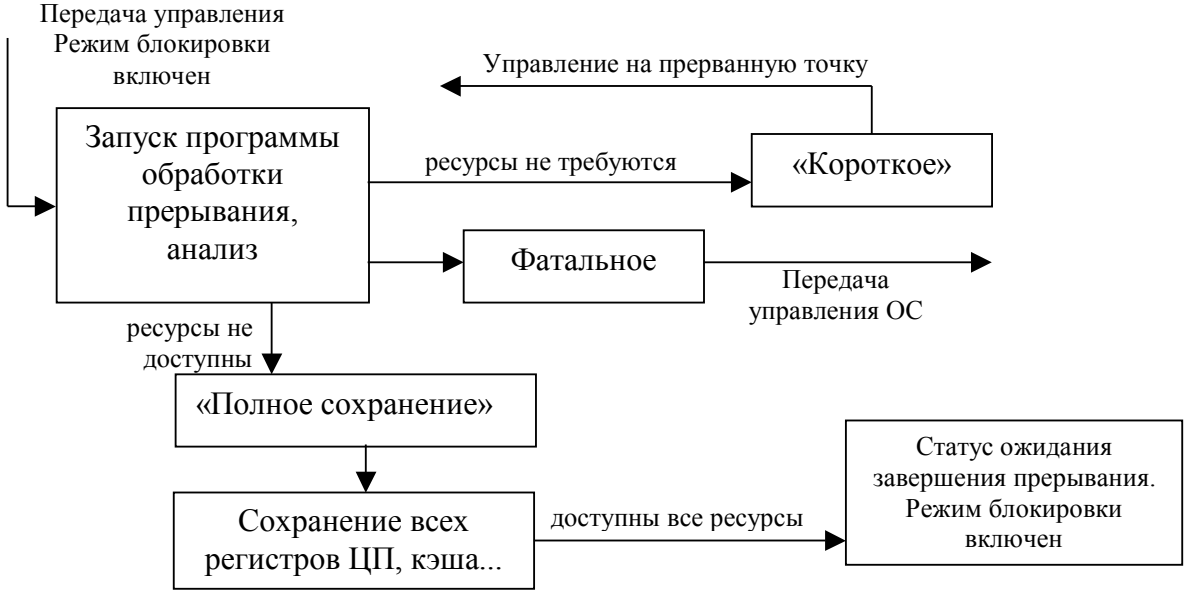


Рис. 32. Этап программной обработки прерываний.

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 (п.1.2.3.4).
   - Если прерывание является «фатальным» для программы, т.е. после этого прерывания продолжить выполнение программы невозможно (например, в программе произошло 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 сохранение»: осуществляется полное сохранение 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. Если 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, то спектр внешних устройств современных компьютеров несоизмеримо шире, что соответствует разнообразию задач, решаемых средствами современных вычислительных систем (1.2.4).

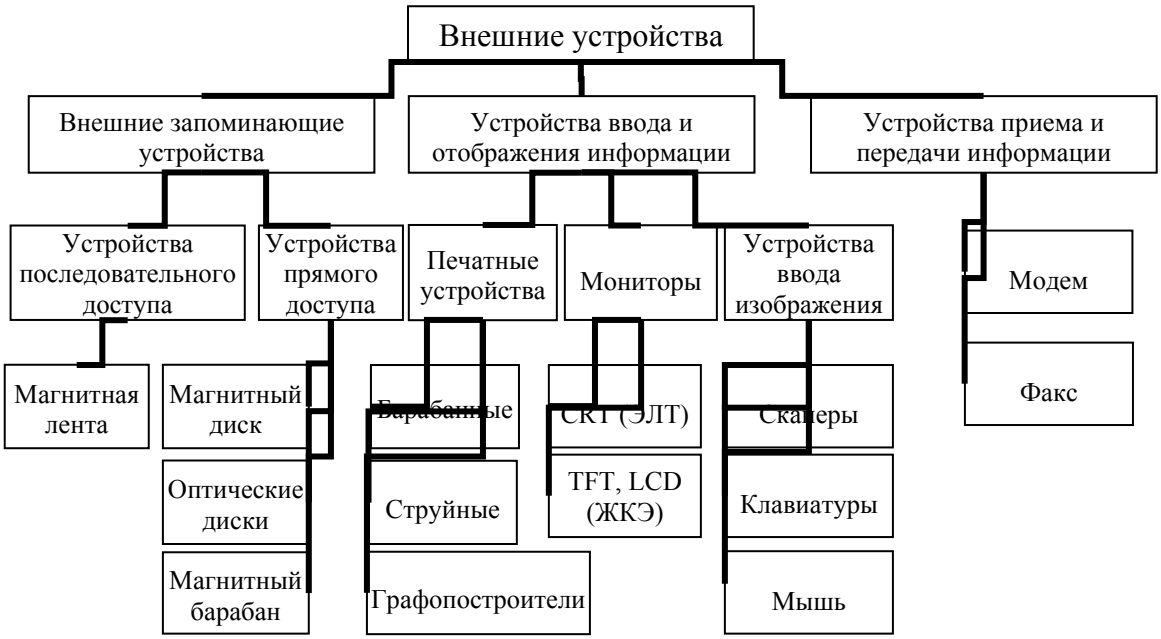


Рис. 33. Внешние устройства.

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

1.2.4.1 Внешние запоминающие устройства

Внешние запоминающие устройства (ВЗУ) предназначены для организации хранения данных и программ. Обычно операции чтения или записи 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 (1.2.4.1).

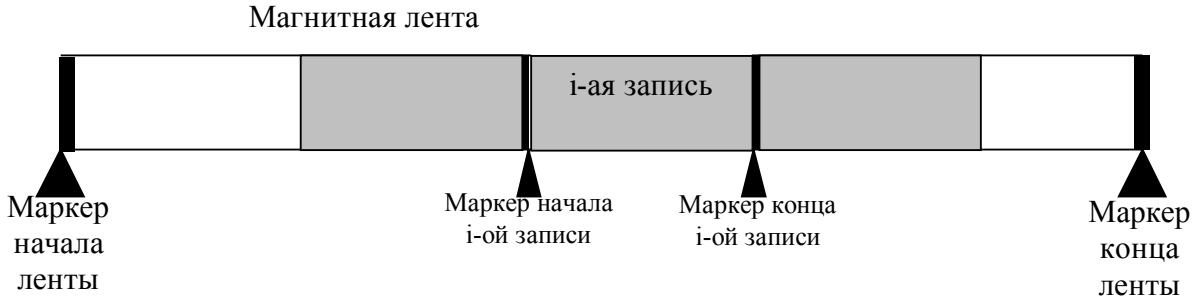


Рис. 34. Магнитная лента.

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 информации. Примером устройств прямого доступа могут служить **магнитные диски**, или **дисковые устройства**.

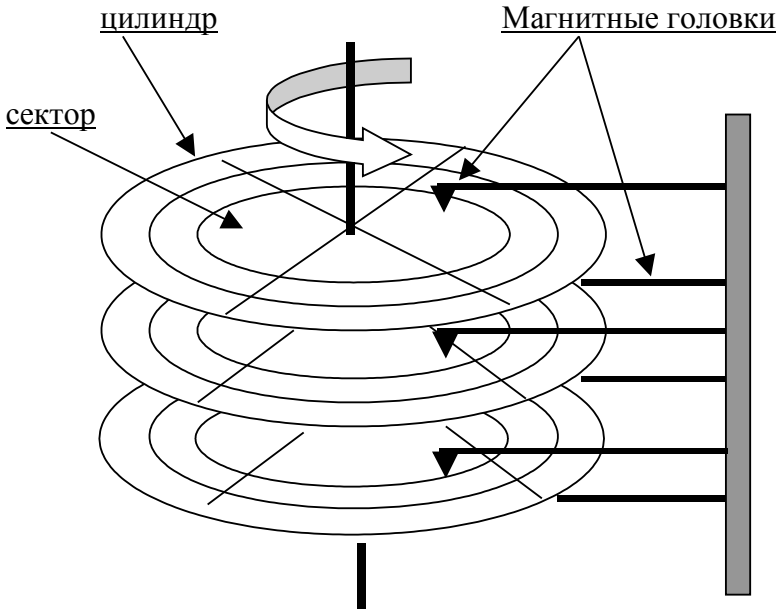


Рис. 35. Принцип устройства магнитного диска.

Магнитные диски являются самыми распространенными устройствами внешней памяти современных компьютеров. Рассмотрим принципиальную схему организации магнитного диска (1.2.4.1). 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** (1.2.4.1). 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, соответствующей 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.

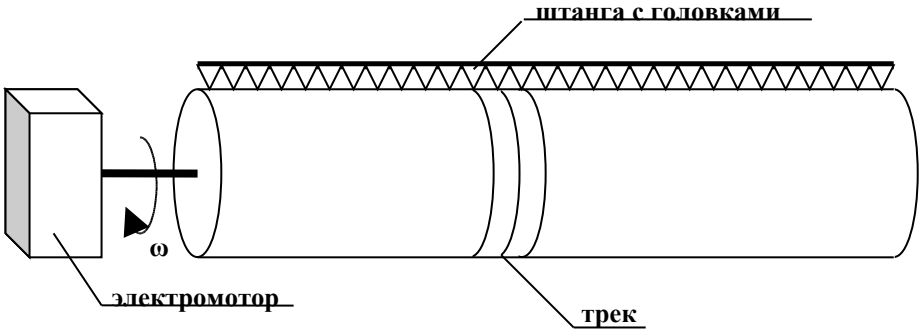


Рис. 36. Принцип устройства магнитного барабана.

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**). 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 1.2.4.1 либо как «плюс-минус», либо как «минус-плюс»).

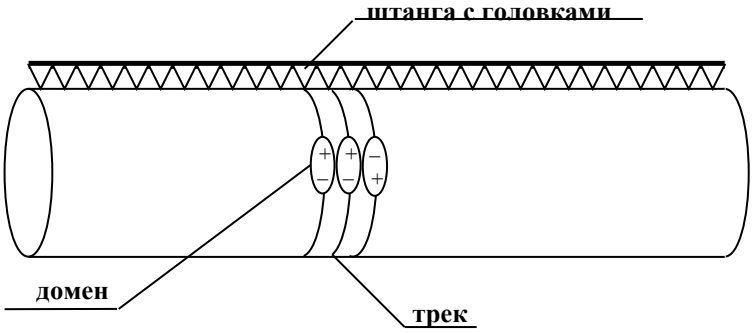


Рис. 37. Принцип устройства памяти на магнитных доменах.

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 завершения обмена (1.2.4.2). Тем самым в системе возникали задержки, которые снижали эффективность функционирования ВС.

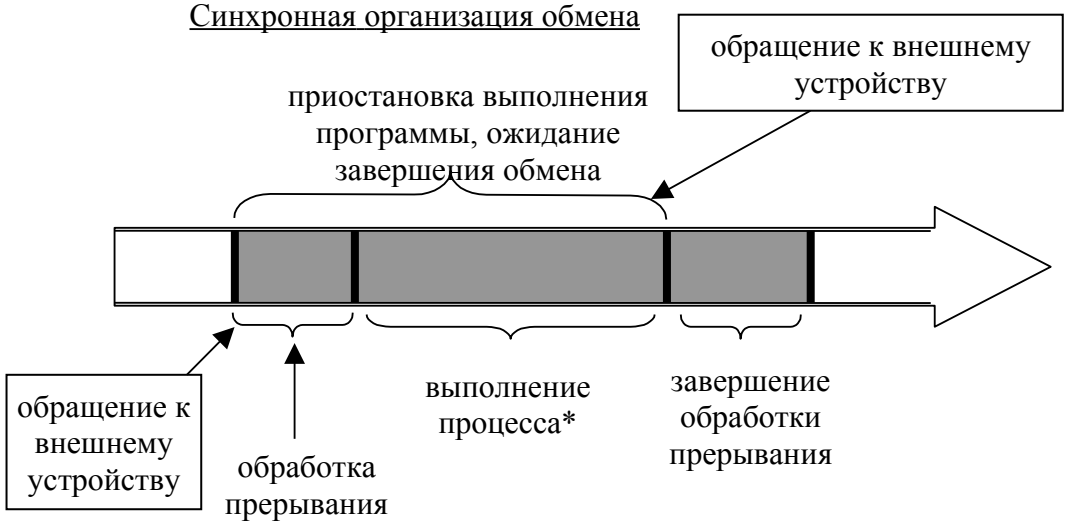


Рис. 38. Синхронная и асинхронная работа с ВУ.

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 управляющая информация, 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 данным ВЗУ и оперативной памятью. Рассмотрим теперь различные модели организации управления ВЗУ.

Простейшей моделью является **непосредственное управление процессором** внешними устройствами (1.2.4.3). 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 через 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.

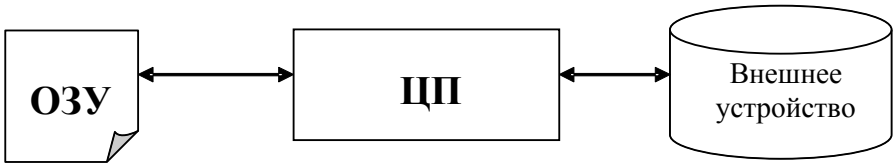


Рис. 39. Непосредственное управление центральным процессором внешнего устройства.

Следующая модель предлагает **синхронное управление** внешними устройствами с использованием контроллеров внешних устройств (1.2.4.3). 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.

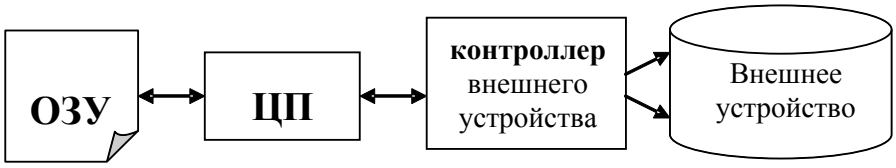


Рис. 40. Синхронное/асинхронное управление внешними устройствами с использованием контроллеров внешних устройств.

Вслед за предыдущим типом устройств появились устройства, позволяющие осуществлять **асинхронное управление** с использованием контроллеров ВЗУ (1.2.4.3). В этом случае центральный процессор подает команду на обмен и не дожидается, когда эту команду отработают

контроллер и устройство, т.е. он 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, 1.2.4.3). 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.

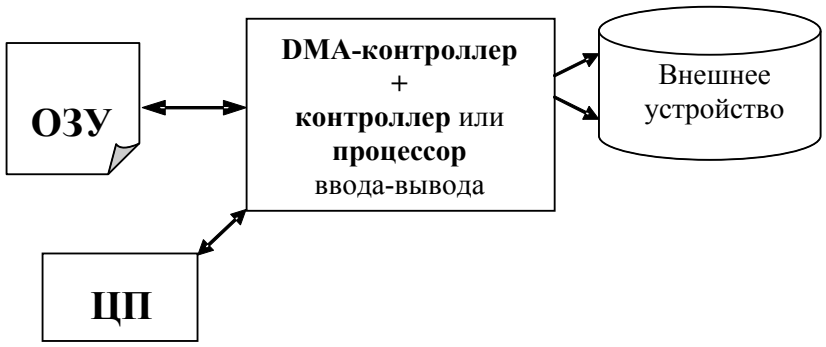


Рис. 41. Использование контроллера прямого доступа к памяти (DMA) или процессора (канала) ввода-вывода при обмене.

И, наконец, последняя модель основана на использовании **процессора** или **канала ввода-вывода** (1.2.4.3). 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, 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 (1.2.5).

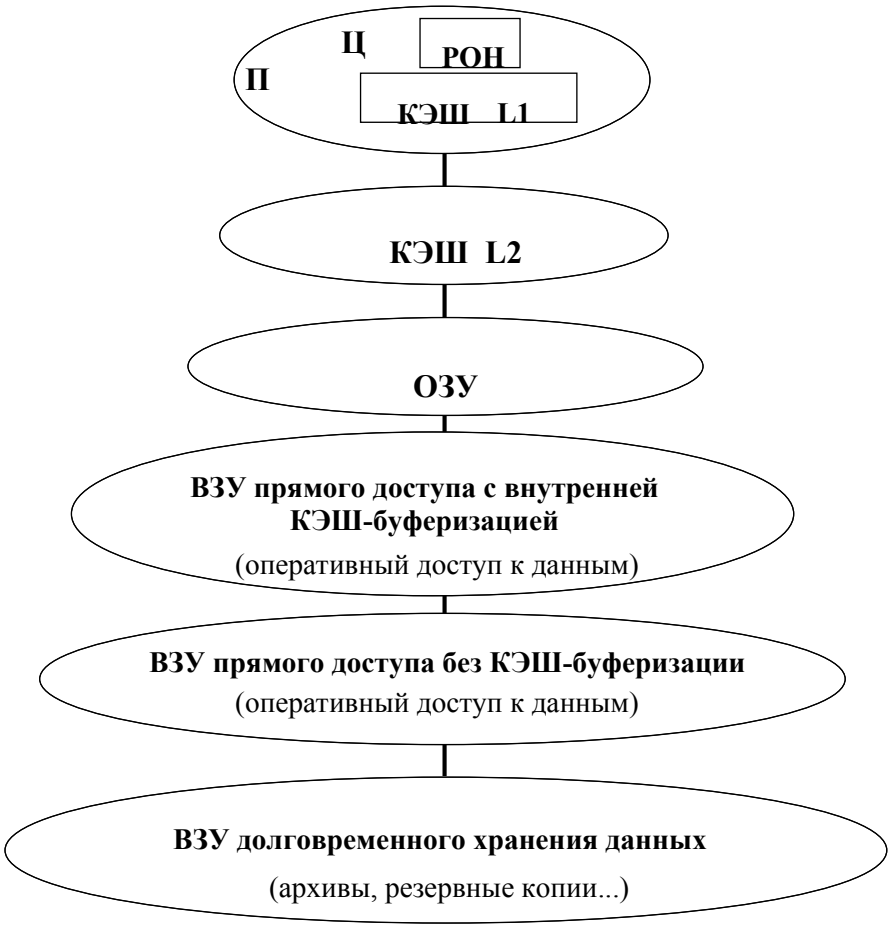


Рис. 42. Иерархия памяти.

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 являет то, что в ней располагается исполняемая в данный момент центральным процессором 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 целью 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, в частности, операционной системы и систем программирования.

1.2.6.1 Требования к аппаратуре для поддержки мультипрограммного режима

Выше уже речь уже шла о **мультипрограммном режиме**, когда в обработке могут находиться две и более программы пользователей, и 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 (1.2.6.1). Пускай 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 других программ, пока *Программа 1* ожидает завершения своего обмена. На рисунке 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.

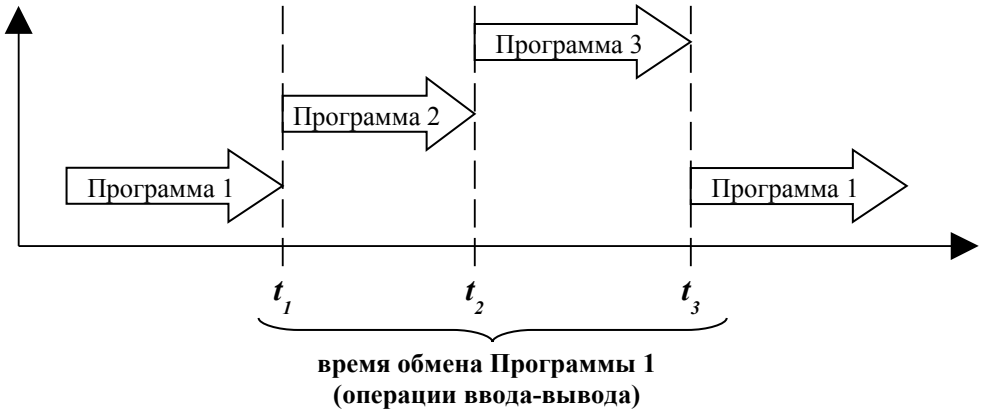


Рис. 43. Мультипрограммный режим.

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.

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 плоха ситуация, когда другая программа может что-то записать в чужое адресное пространство. Соответственно, для корректного мультипрограммирования система должна обеспечивать эксклюзивное владение программ 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***. 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 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 конечных физических устройств, 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 механизм защиты памяти. Значения указанных регистров границ устанавливаются посредством специальных машинных команд. 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 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 исполняет абсолютно все команды,

представленные в программе. Если же программа исполняется в пользовательском режиме, то ей доступны для исполнения лишь некоторое подмножество машинных команд (если же при обработке такой программы встретится недопустимая команда, то в системе возникнет прерывание по запрещенной команде).

Тогда возникает вопрос, что должна делать программа, обрабатываемая в пользовательском режиме, для печати, например, своих данных. 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, положим, передавать необходимые данные, которые могут свидетельствовать о желании данной программы распечатать какую-то 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, еще одна серьезная проблема, которая может возникнуть при организации мультипрограммного режима, связана с тем, что в выполняемой в текущий момент программе встретилась семантическая ошибка — 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 пользователя. 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 будет периодически прерываться, управление периодически будет попадать операционной системе, что даст возможность поставить на счет другую программу либо снять со счета (например, по команде пользователя) эту зависшую программу.

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.

Вложенные обращения к подпрограммам (1.2.6.2). 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) для выполнения 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).

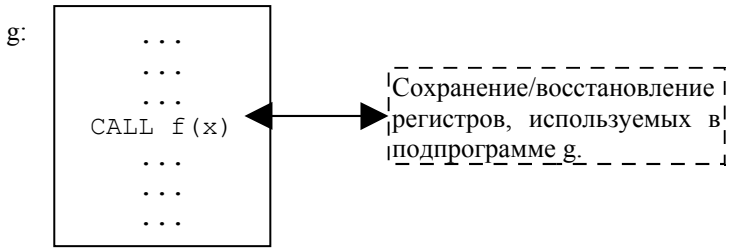


Рис. 44. Вложенные обращения к подпрограммам.

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 (1.2.6.2). Рассмотрим процесс получения исполняемого кода программы. 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. Исторически первые исполняемые модули настраивались на те адреса оперативной памяти, в рамках которых он должен был 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.

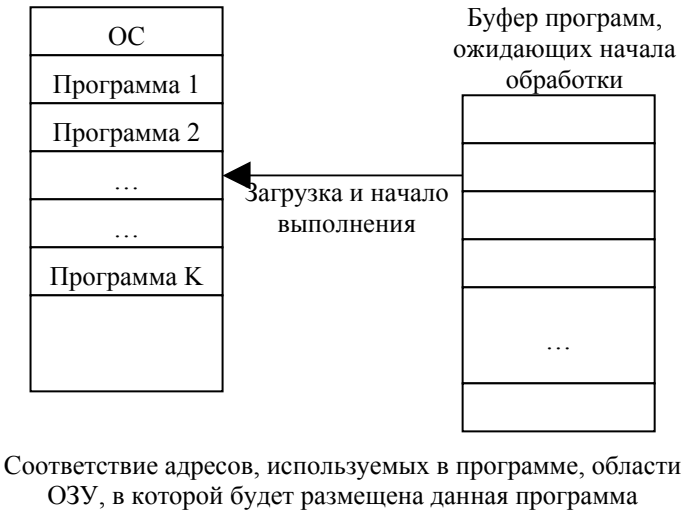


Рис. 45. Перемещаемость программы по ОЗУ.

Фрагментация памяти. Положим, что предыдущая проблема, связанная с перемещаемостью кода, решена в нашей системе: любой исполняемый модуль может быть загружен в произвольное место ОЗУ для дальнейшего выполнения. Но в этом случае возникает иная проблема.

Пускай наша система работает в мультипрограммном режиме. И в начале работы были запущены на исполнение *Программа 1*, *Программа 2* и т.д., вплоть до некоторого номера *K*. 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*** (1.2.6.2). В некоторый момент может оказаться, что в ОЗУ находится несколько процессов, между которыми имеются фрагменты свободной памяти, каждый из которых не достаточен для того, чтобы загрузить какую-либо готовую к исполнению программу. Но количество подобных фрагментов может быть настолько большим, что суммарно свободное пространство ОЗУ позволил 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.

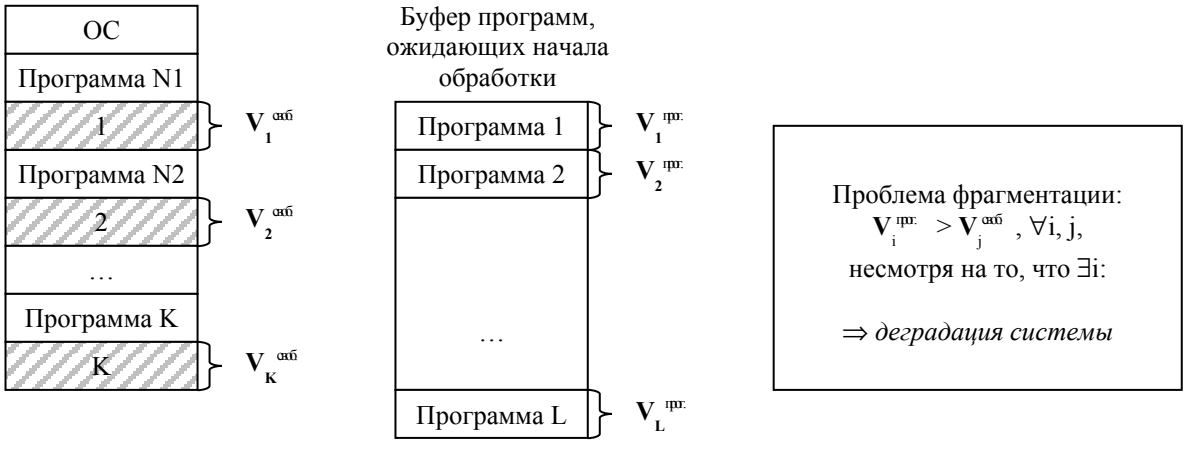


Рис. 46. Фрагментация памяти.

После того, как мы указали основные проблемы, возникающие при исполнении программ, рассмотрим, как эти проблемы могут решаться.

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***). Суть этого решения заключается в следующем (1.2.6.3). В процессоре имеется некоторое количество K физических регистров, предназначенных для использования в пользовательских программах. Эти регистры пронумерованы от 0 до K–1. Также имеется понятие регистрового окна — это набор регистров, по количеству меньший K, который в каждый момент времени доступен для программы пользователя. Соответственно, эти 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.

Итак, имеющиеся K физических регистров разбиты 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. Считается, что достигается эффект оптимизации при четырех окнах, что 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 1.2.6.3 приведены схемы работы с регистровыми окнами.

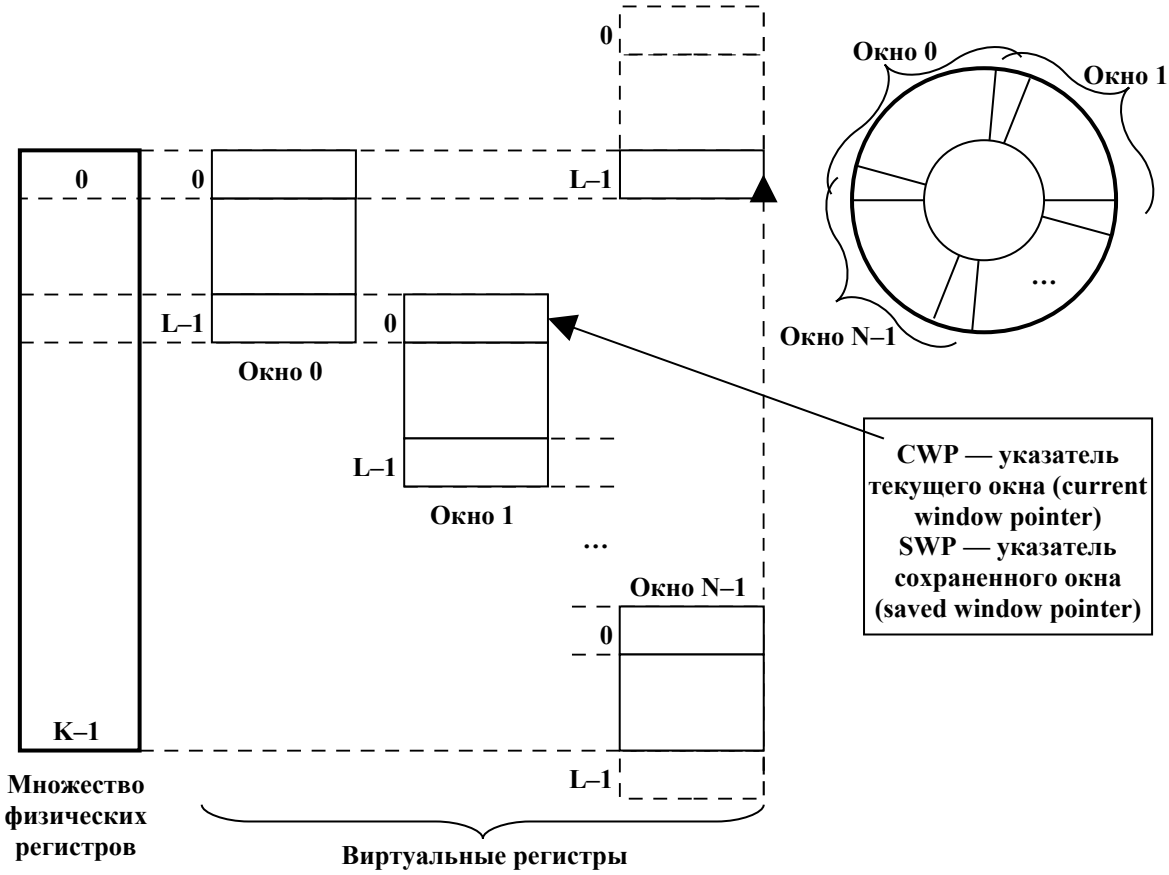


Рис. 47. Регистровые окна.

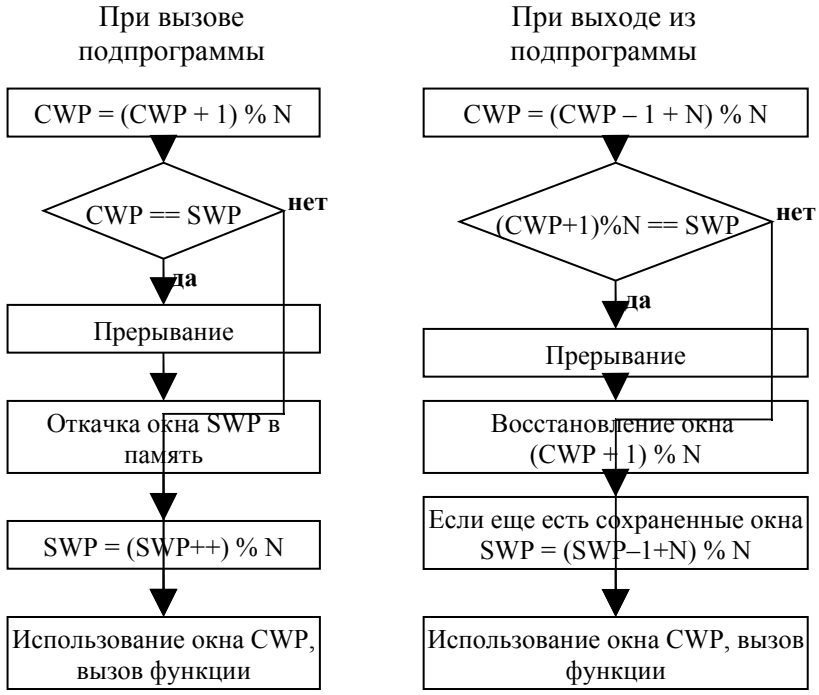


Рис. 48. Регистровые окна. Вход и выход из подпрограммы.

Модель организации регистровой памяти в Intel Itanium. 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 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 32-ого регистра 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.

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. Системный стек может применяться для оптимизации работ, связанных со сменой контекстов программ. В частности, этот механизм может использоваться при обработке прерывания: если в системе возникает прерывание, процессор просто скидывает в стек содержимое необходимых регистров. Если же возникнет второе прерывание, то процессор поверх предыдущих данных скинет в стек новое содержимое регистров, чтобы обработать вновь пришедшее прерывание.

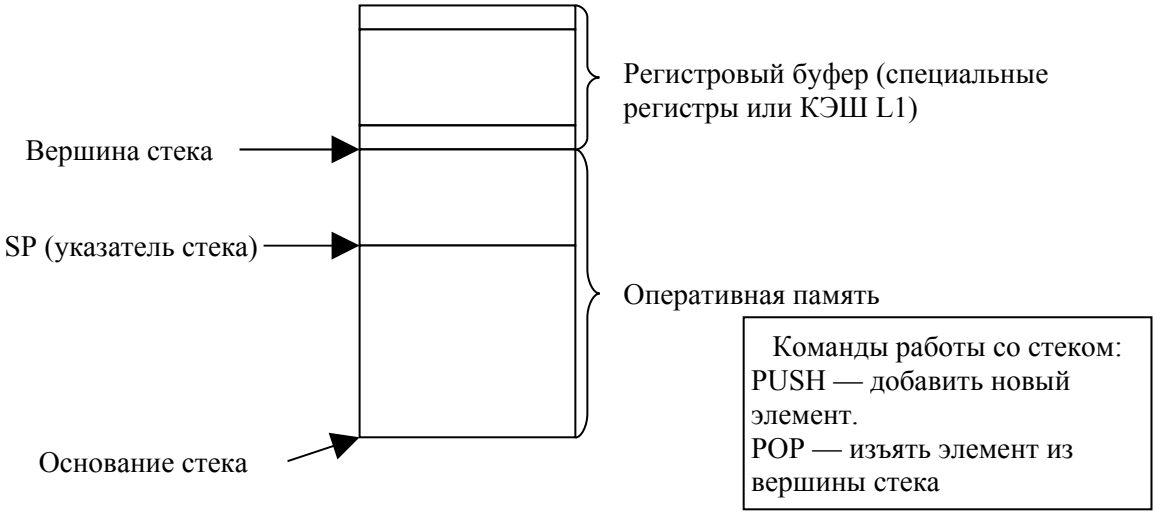


Рис. 49. Системный стек.

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 (1.2.6.4). 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).

1.2.6.5 Виртуальная память

Следующий аппарат компьютера, который также сильно связан с поддержкой программного обеспечения, — это аппарат ***виртуальной памяти***. 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, где реально будут размещены, к каким физическим адресам привязаны. Виртуальные адреса существуют «вне машины». Соответственно, стоит проблема привязки

виртуального адресного пространства физической памяти. И эта проблема решается за счет аппарата виртуальной памяти.

Итак, ***аппарат виртуальной памяти*** — это аппаратные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе, адресам физической памяти, в которой размещена программа при выполнении. И реализацией одной из моделей аппарата виртуальной памяти является аппарат ***базирования адресов***.

Механизм базирования адресов основан на двоякой интерпретации получаемых в ходе выполнения программы исполнительных адресов ($А_{исп.}^{прог.}$). С одной стороны, его можно интерпретировать как ***абсолютный исполнительный адрес***, когда физический адрес в некотором смысле соответствует исполнительному адресу программы ($А_{исп.}^{физ.} = А_{исп.}^{прог.}$). 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. Иными словами, имеется оперативная память с ячейками с номерами от 0 до некоторого А–1, и, начиная с некоторого адреса K, расположена программа. Тогда адрес $А_{исп.}^{прог.}$ внутри программы можно трактовать, как отступ от физической ячейки с адресом K на величину $А_{исп.}^{прог.}$. 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 $А_{исп.}^{физ.} = А_{исп.}^{прог.} + <R_{базы}>$.

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.

Развитием аппарата виртуальной памяти является аппарат ***страничной организации памяти***. Ниже мы рассмотрим модельный сильно упрощенный пример страничной памяти. Данная модель представляет все адресное пространство оперативной памяти в виде последовательности страниц. ***Страница*** — это область адресного пространства фиксированного размера: обычно размер страницы кратен степени двойки, будем считать, что размер страницы 2^k. Тогда все адресное пространство представимо в виде последовательности страниц (нулевая, первая и т.д.). Сказанное означает, что структура адреса представима в виде двух полей (1.2.6.5): правые k разрядов представляют адрес внутри страницы, а оставшиеся разряды отвечают за номер страницы. Тогда количество страниц в системе ограничено разрядностью поля «Номер страницы».

Рис. 50. Страничная организация памяти.

В центральном процессоре имеется аппаратная таблица, называемая ***таблицей страниц***, предназначенная для следующих целей. Количество строк в этой таблице определяется максимальным числом виртуальных страниц, ограниченное схемами работы процессора и максимальной адресной разрядностью процессора. Каждой виртуальной странице 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 на номер физической страницы посредством обращения к таблице страниц (1.2.6.5). 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.

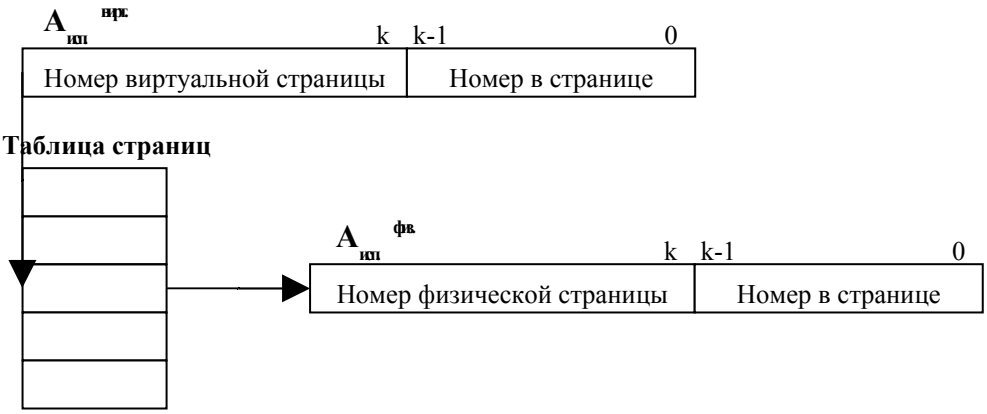


Рис. 51. Страничная организация памяти. Преобразование виртуального адреса в физический.

Типовая схема преобразования адресов достаточно проста (1.2.6.5). Пускай в таблице страниц имеется N строк. Это означает, что в компьютере дозволено использовать N страниц. Содержимое каждой i-ой строки таблицы — α_i, оно определяется операционной системой в момент запуска процесса. Пускай в нашем модельном примере если $\alpha_i >= 0$, то это номер физической страницы, которая соответствует i-ой виртуальной странице. 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. 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.

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 построено на использовании регистров. Также данный подход решает проблему фрагментации, поскольку все программы оперируют в терминах страниц (каждая из которых имеет фиксированный размер). 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: ее нулевая виртуальная страница 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. Реально в ОЗУ может находиться лишь незначительное число страниц, в которых расположены команды и требуемые для текущих вычислений операндов, а все оставшиеся страницы могут находиться на внешней памяти — в областях подкачки. Как следствие 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.

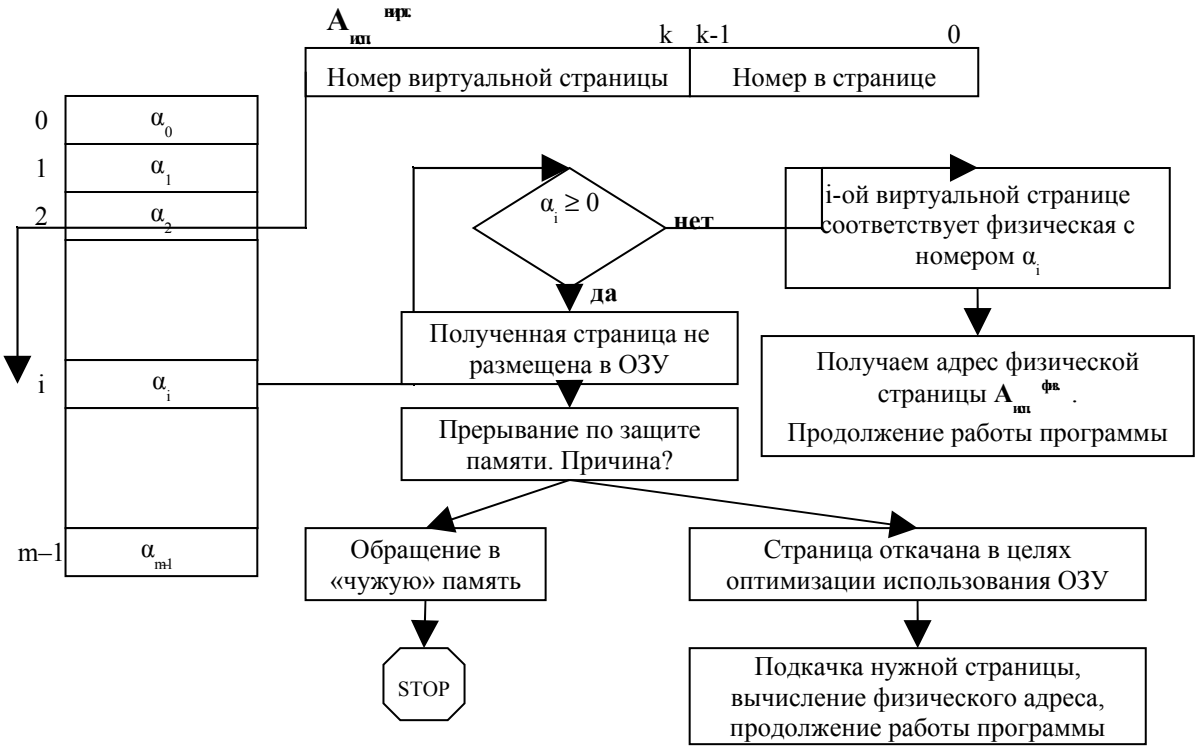


Рис. 52. Страничная организация памяти. Схема преобразования адресов.

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 (т.е., решив вопрос с т.н. ***внешней фрагментацией***, в указанном случае не используется память, размером со страницу минус один байт). К тому же описанная выше модель является вырожденной: если таблица страниц целиком располагается на регистровой памяти, то в силу дороговизны последней размеры подобной таблицы будут слишком малы (а следовательно, будет невелико количество физических страниц). Реальные современные системы имеют более сложную логическую организацию, и речь о ней пойдет ниже.

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, 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. Размеры 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 — это **классификация по Флинну** (M.Flynn), основанная на оценке некоторых характеристик потоков информации в машине.

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.

В классификации по Флинну выделяют следующие четыре архитектуры:

- **ОКОД** (одиночный поток команд, одиночный поток данных, или **SISD** — single instruction, single data stream) — это традиционная однопроцессорная система (близкая машине фон Неймана).
- **ОКМД** (одиночный поток команд, множественный поток данных, или **SIMD** — single instruction, multiple data stream) — например, векторные компьютеры, способные оперировать векторами данных. Обычно для этих целей в данных машинах существуют векторные регистры, а также обычно имеются векторные операции, предполагающие векторную обработку.
- **МКОД** (множественный поток команд, одиночный поток данных, или **MISD** — multiple instruction, single data stream) — данный 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, 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). 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** (1.2.7). 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.

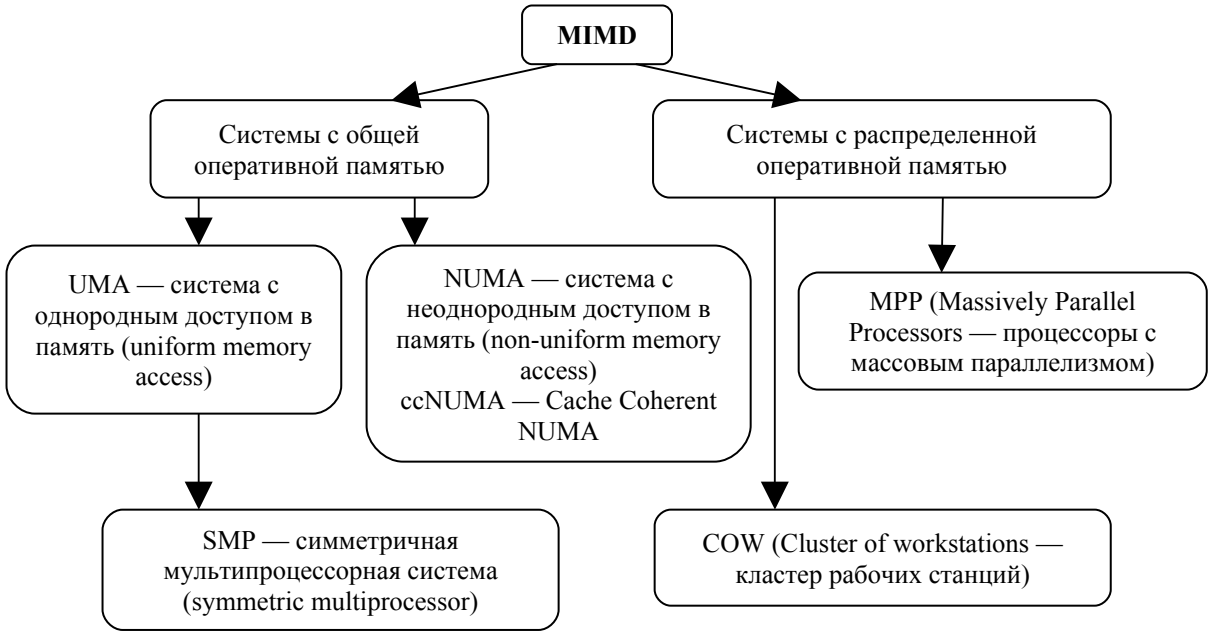


Рис. 53. Классификация МКМД.

Рассмотрение систем с общей оперативной памятью начнем с UMA. **UMA** (uniform memory access) — система с однородным доступом в память. В данной модели произвольный процессорный элемент имеет доступ к произвольной точке оперативной памяти (доступ с одинаковым временем). Развитием архитектуры UMA стала модель **SMP** (symmetric multiprocessor — симметричная мультипроцессорная система). В этой модели (1.2.7) 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 шина в ней является «узким горлом», поэтому данная модель накладывает существенные ограничения на количество подключаемых процессоров (обычно 2, 4, 8, вплоть до 32). Во-вторых, возникают дополнительные проблемы с КЭШ 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 приведены в следующей таблице:

	Действия локального КЭШа (в том ЦП, где выполняется операцию)	Действия «внешнего КЭШа» (на других процессорах)
R– (промах при чтении)	M → C (идет чтение из ОП в кэш)	ничего
R+ (попадание при чтении)	USE C (использование кэш)	ничего (т.к. ничего не увидит)
W– (промах по записи)	→ M (обновление памяти, кэш не обновляется)	ничего
W+ (попадание при записи)	→ C → M (обновление и КЭШа, и памяти)	соответствующая запись из КЭШа будет удалена

Замечание. При промахе по записи производится только обновление памяти, т.к. реализуется стратегия, ориентированная на преимущественное чтение.

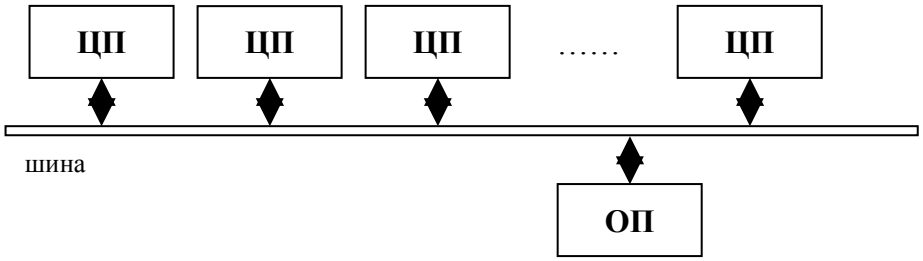


Рис. 54. SMP-система.

Иной подход к реализации систем с общей оперативной памятью предлагает архитектура NUMA (non-uniform memory access — система с неоднородным доступом в память). Для данных систем (1.2.7) характерны следующие свойства:

- общее адресное пространство;
- характеристики доступа процессора к области оперативной памяти зависит от того, к каким областям идет обращение.

Модификацией модели NUMA является модель ccNUMA (Cache coherent NUMA) — это NUMA-система с когерентными КЭШами. Данные системы позволяют подключать несколько сотен процессоров, но остаются ограничения, связанные с использованием системной шины, а также возникают ограничения, связанные с сс-архитектурой: появляются системные потоки служебной информации, что ведет к дополнительным накладным расходам.

Теперь рассмотрим системы с распределенной оперативной памятью. 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 классов).

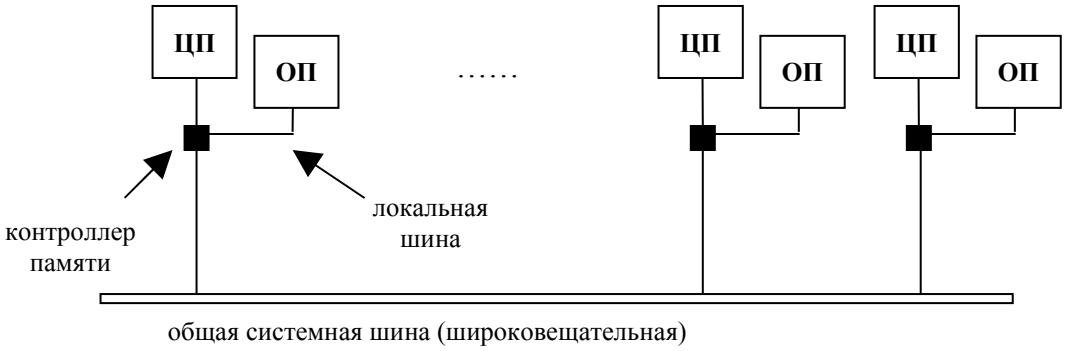


Рис. 55. NUMA-система.

MPP обычно являются дорогостоящими специализированными многопроцессорными системами, поэтому не находят массового применения. Системы данного класса имеют разнообразные формы архитектур: это могут быть макроконвейерные архитектуры, кубы и гиперкубы.

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 с другими вычислительными узлами используют специализированные компьютерные сети.

Кластеры могут создавать для достижения следующих основных целей:

– построение кластера как высокопроизводительной вычислительной системы, т.е. вычислительного кластера (критерием эффективности выступает скорость обработки информации);
– построение кластера, обеспечивающего надежность. Данный тип кластеров строится для решения конкретной прикладной задачи (например, сервер базы данных авиабилетов), при этом выход из строя некоторых узлов не означает отказ системы: система продолжается функционировать пускай и со сниженной производительностью.

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. 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)

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).

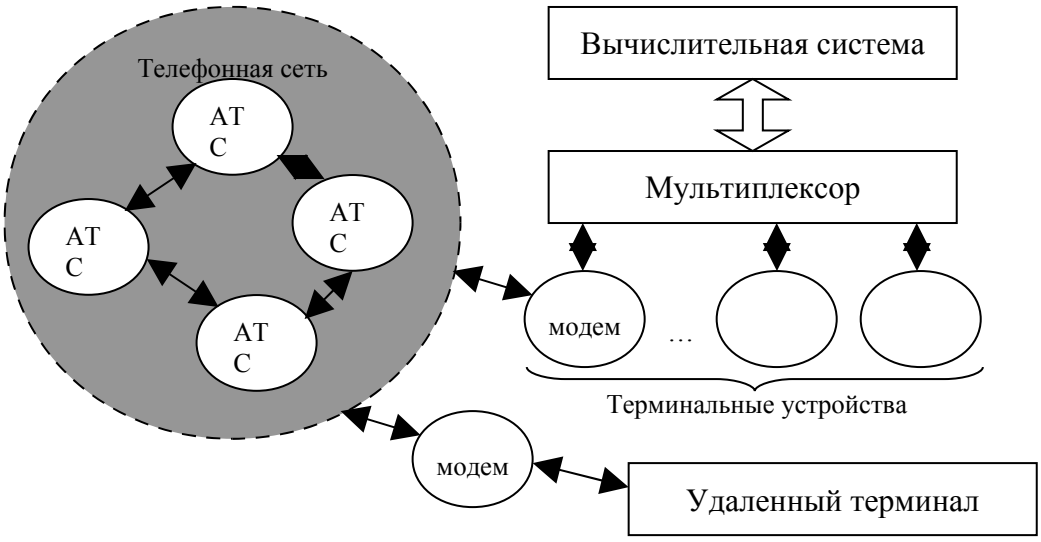


Рис. 56. Терминальные комплексы.

Терминальные комплексы предполагают в своем составе следующие компоненты:
– основная вычислительная система (к которой делается 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).

Терминальные комплексы можно разделить на **локальные** и **удаленные**. Локальные терминальные ТК имеют либо непосредственное подключение к ВС по локальному каналу связи, либо подключение через мультиплексор (устройство, позволяющее свести N каналов в один). **Удаленные** ТК 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.

Линии связи/каналы. Существуют три критерия, по которым можно делить каналы. Во-первых, все каналы можно поделить на **коммутируемые** и **выделенные**.

Коммутируемый канал — это канал, выделяемый на весь сеанс работ терминального устройства. Примером здесь 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 связи).

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

– ***канал точка-точка*** — общение двух устройств;
– ***многоточечный канал*** — общение многих устройств (например, при мультиплексировании).

И, наконец, каналы можно делить по направлению движения информации в канале:

– ***симплексный*** — канал работает в одном направлении (например, репродуктор громкой связи на вокзале или в организациях);
– ***дуплексный*** — двунаправленный канал (например, телефон);
– ***полудуплексный*** — канал работает 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 следующим шагом стала замена терминальных устройств компьютерами.

Компьютерная сеть — это объединение компьютеров (или вычислительных систем), взаимодействующих через коммуникационную среду (1.2.9).

Коммуникационная среда — каналы и средства передачи данных.

Можно выделить следующие свойства компьютерных сетей. Во-первых, сеть может состоять из значительного числа связанных между собой автономных компьютеров. 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.

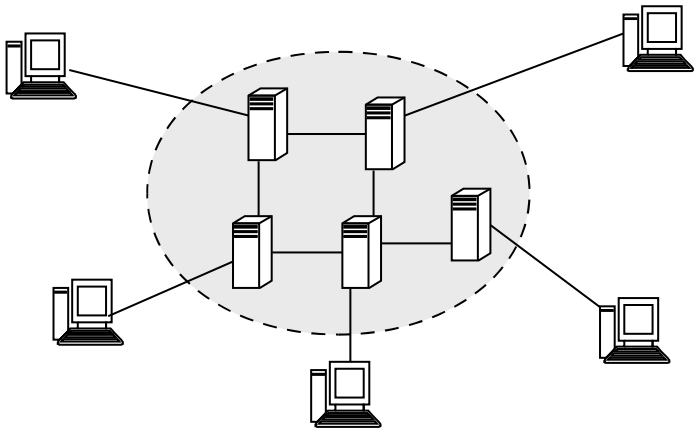


Рис. 57. Компьютерные сети.

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

Следующее свойство — расширяемость. Компьютерная сеть должна обеспечивать возможность развития по протяженности, по расширению пропускной способности канала, по составу и производительности компонентов сети.

И, наконец, возможность применения симметричных интерфейсов обмена информацией между компьютерами сети, позволяющих произвольным способом распределять функции сети.

Будем говорить, что логически компьютеры, составляющие сеть, состоят из **абонентских машин** (или основных компьютеров — хостов) и **коммуникационных** (или вспомогательных) компьютеров (шлюзы, маршрутизаторы и пр.). Последние выполняют фиксированные функции по обеспечению функционирования сети. Это деление логическое. На практике может оказаться, что одна машина выполняет роль как абонентской машины, так и коммуникационной.

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.

В **сети коммутации каналов** 2 абонента сети взаимодействуют при помощи сеанса связи, любой из которых является обменом сообщениями. Под сообщением будем понимать логически целостный набор данных произвольного размера. Коммутация каналов происходит на весь сеанс связи.

К достоинствам данных сетей можно отнести следующие свойства:

- канал находится всегда в состоянии готовности;
- требования к коммуникационному оборудованию минимальны: по сути, нет требований к обеспечению буферизации;
- обмены происходят порциями данных произвольной длины — ***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 коммутационного канала: в отдельно взятом сеансе может быть низкая интенсивность обменов сообщениями;
- при отказах и сбоях повторение переданной информации является сложной задачей.

Сети коммутации сообщений — это сети, которые оперируют термином «передача сообщения», а не «сеанс связи», т.е. один абонент другому отправляет сообщение. Выделение канала для передачи каждого сообщения происходит поэтапно 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 неопределенный 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;
- обеспечение буферизации требует дорогостоящего коммутационного оборудования;
- 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, что в основе лежит сеть, использующая ненадежные средства связи. Функционирования данной сети состоит в следующем: любое сообщение дробится на блоки фиксированной длины, которые называются ***пакетами***. 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.

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 то, что по сети перемещается накладная информация, которая прибавляется к каждому пакету при разбиении сообщения. 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

Теперь 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.

Аналогичная картина и в вычислительной технике. Была введена стандартизация на компьютерные комплектующие, а также на программные интерфейсы. Стандартизация интерфейсов дает возможность взаимозаменяемости компонентов и их работоспособности.

Изначально компьютерные сети развивались на основе терминальных комплексов, которые строились по внутрикорпоративным правилам. Это означает, что коммуникационное программное обеспечение, аппаратура сопряжения и пр. были «своими», и при необходимости модернизации этой сети возникали сложные физические и технологические проблемы.

Развитие сетей определило массовость их использования. Возникла необходимость создания сетей, которые могли бы достаточно прочно расширяться без привлечения существенных переделок, модернизироваться, в которых могли бы меняться ПО, добавляться новые службы. 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). Эта модель ISO/OSI рассматривает сеть и взаимодействие компьютеров в сети в виде семи функциональных уровней. Стоит отметить, что данная модель является рекомендацией, а не стандартом: ISO выделила их на основе анализа исторического развития компьютерных сетей (1.2.10).

Рис. 58. Модель организации взаимодействия в сети ISO/OSI.

Сначала более детально рассмотрим назначение каждого уровня, а затем сформулируем основные понятия.

Физический уровень. На этом уровне происходит непосредственно передача неструктурированной двоичной информации. Для передачи используется конкретная физическая среда (кабель, радиоволны и т.п.). На данном уровне обеспечивается 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, обеспечение 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.

Сетевой уровень. На этом уровне 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, а также программное взаимодействие (а не взаимодействие устройств). Тут же принимается решение о выборе типа услуг (транспортировка данных с установлением виртуального канала или же без оного). В случае установления виртуального канала осуществляется контроль за доставкой и отсутствием ошибок. Если же виртуальный канал не устанавливается, то уровень не несет ответственности за доставку.

Сеансовый уровень. Этот уровень обеспечивает управление сеансами связи. На этом уровне решаются задачи подтверждения полномочий, т.е. осуществляется работа со всякого рода ролями и пр., а также решаются задачи организации меток, или контрольных точек, по сеансу, которые позволяют в случае возникновения сбоя повторять передачу не с начала, а с последней установленной контрольной точки.

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

Уровень прикладных программ. На этом уровне формализуются правила по взаимодействию с прикладными системами.

Теперь на основе рассмотренных уровней можно дать определения основных понятий (1.2.10).

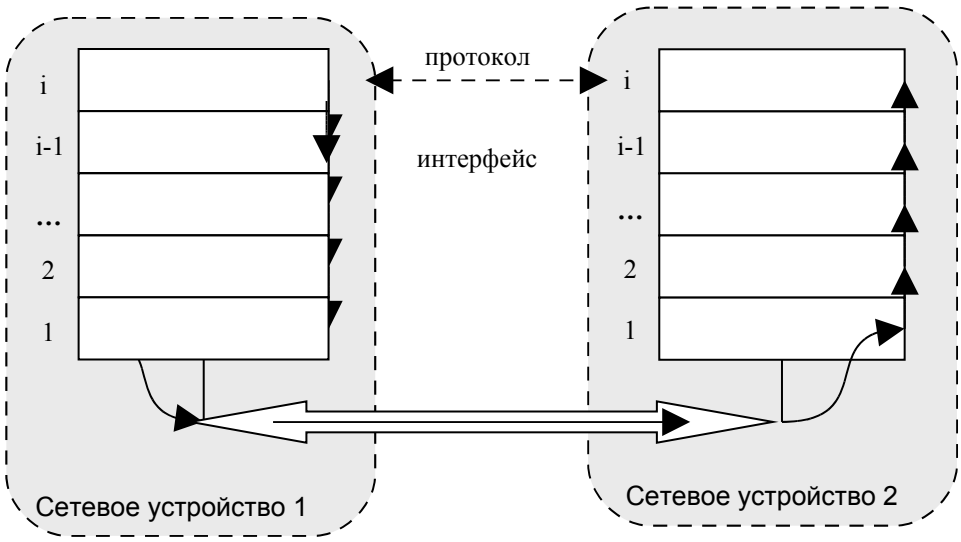


Рис. 59. Логическое взаимодействие сетевых устройств по i-ому протоколу.

Протокол — формальное описание сообщений и правил, по которым сетевые устройства (вычислительные системы) осуществляют обмен информацией. Таким образом, протокол обеспечивает взаимодействие в сети между различными машинами на одном уровне. Любой из уровней может содержать произвольное число протоколов, но общаться могут лишь протоколы одного уровня. Также 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 протоколов, реализованных в системе. Стек может быть произвольной глубины, т.е. в нем, возможно, не будут представлены протоколы некоторых уровней модели ISO/OSI.

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

Рассмотрим еще одну модель организации сетевого взаимодействия — семейство протоколов TCP/IP (1.2.11). 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 министерства обороны США в агентстве перспективных разработок МО США DARPA. Это агентство разработало сеть ARPA-net, которая в своем развитии легла в основу современной сети Internet (поскольку это семейство протоколов было интегрировано в ОС BSD Unix).

Рис. 60. Семейство протоколов TCP/IP.

Попытаемся сопоставить модели TCP/IP и ISO/OSI.

Уровень доступа к сети. Этот уровень соответствует физическому и канальному уровням модели ISO/OSI. На нем 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.

Транспортный уровень. Он покрывает сеансовый и транспортный уровни модели ISO/OSI. На этом уровне имеется возможность использования протоколов, которые устанавливают виртуальное соединение или не устанавливают его.

Уровень прикладных программ. Он разрешает проблемы уровня представления и уровня прикладных программ модели ISO/OSI.

Эти уровни модели TCP/IP являются пакетными: на каждом уровне система оперирует порциями данных, обладающими характеристиками соответствующего уровня (1.2.11). Имея содержательную информацию на прикладном уровне, двигаясь от верхнего уровня модели к нижнему, эта информация при необходимости дробится на пакеты фиксированного размера, и к каждому из них добавляется заголовочная информация.

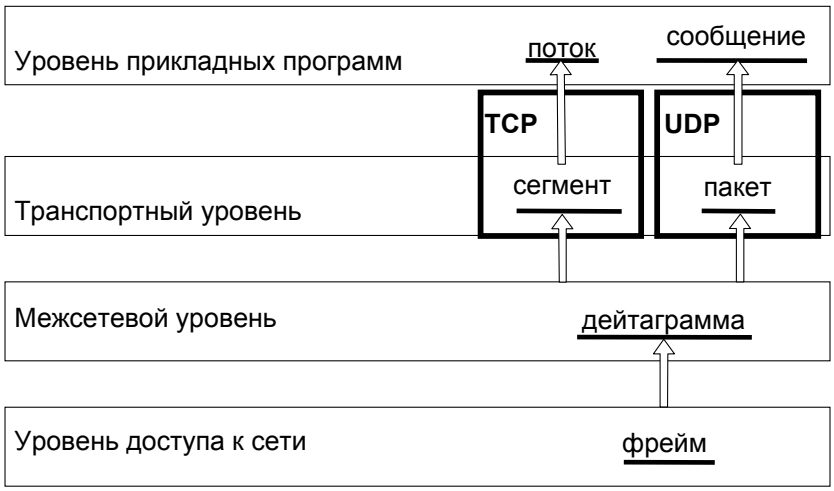


Рис. 61. Взаимодействие между уровнями протоколов TCP/IP.

Остановимся на каждом из уровней модели TCP/IP более подробно.

На **уровне доступа к сети** протоколы обеспечивают систему средствами для передачи данных другим устройствам в сети. В качестве примера можно привести протокол **Ethernet**, являющегося разработкой исследовательского центра компании Xerox (1976 г.), который основывается на единой шине (это широковещательная сеть). Для сетевых устройств обеспечивается множественный доступ с контролем несущей и обнаружением конфликтов (Carrier Sense Multiple Access with Collision Detection — CSMA/CD). Термины ***широковещательный*** и ***множественный доступ*** означают, что любой пакет, «выкинутый» в сеть, виден всем абонентам этой сети. Каждый абонент «слушает» сеть, и тот, кому предназначен пакет, забирает его. ***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» 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 линия занята. Также при интенсивной работе возрастает частота конфликтов, что ведет к снижению производительности системы.

В качестве физической среды передачи данных используются самые разные источники: это может быть «толстый» Ethernet, «тонкий» Ethernet, витая пара, оптоволокно, радиосигнал.

Межсетевой уровень. ***Протокол IP*** — это один из основных протоколов. 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.

Основная функция этого протокола — поддержание системы адресации, позволяющей объединять различные (или гетерогенные) сети в единое целое (т.е. это межсетевая адресация — internet-адресация), а также поддержание маршрутизации. ***IP-адрес*** — это 32-разрядное число, которое кодирует информацию о конкретной сети и компьютере внутри этой сети. Имеются три категории содержательных IP-адресов сетей (1.2.11).

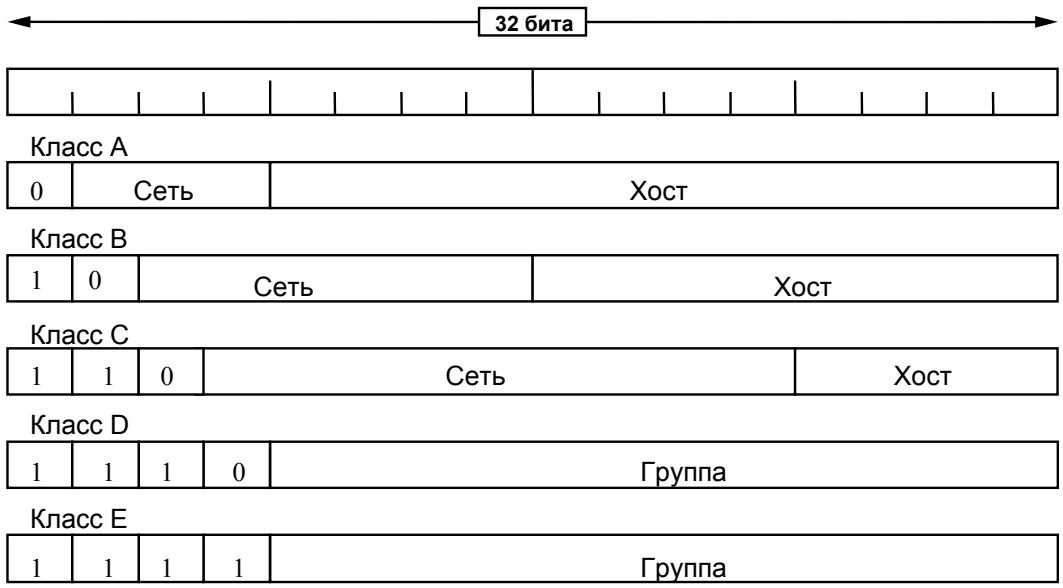


Рис. 62. Система адресации протокола IP.

Формат **класса A** позволяет задавать адреса до 126 сетей с 16 млн. хостов в каждой, **класса B** — до 16382 сетей с 64 Кбайт хостами, и, наконец, **класса C** — 2 млн. сетей с 254 хостами в каждой. 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.

Как отмечалось выше, каждый из уровней взаимодействует с соседними уровнями в соответствии с теми или иным протоколами порциями данных, имеющими специфичными для каждого уровня названия. Так, для межсетевого уровня пакет называется дейтограммой.

Протокол IP подразумевает использование некоторых специализированных компьютеров. Это компьютеры, предназначенные для организации физического объединения различных сетей, и они называются **шлюзами**. В общем случае шлюз имеет два и более сетевых адаптера, на которых функционирует соответствующее число (два или более) стеков протоколов.

Перед межсетевым уровнем также стоит задача **маршрутизации** — определить по имеющему IP-адресу получателя 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 совмещать компьютерам несколько функций: одна и та же машина может быть одновременно и шлюзом, и маршрутизатором, и хостом, причем работающий за ним пользователь может не догадываться об организации локальной сети, в которой он работает.

Рассмотрим **пример** (1.2.11). Пускай 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.

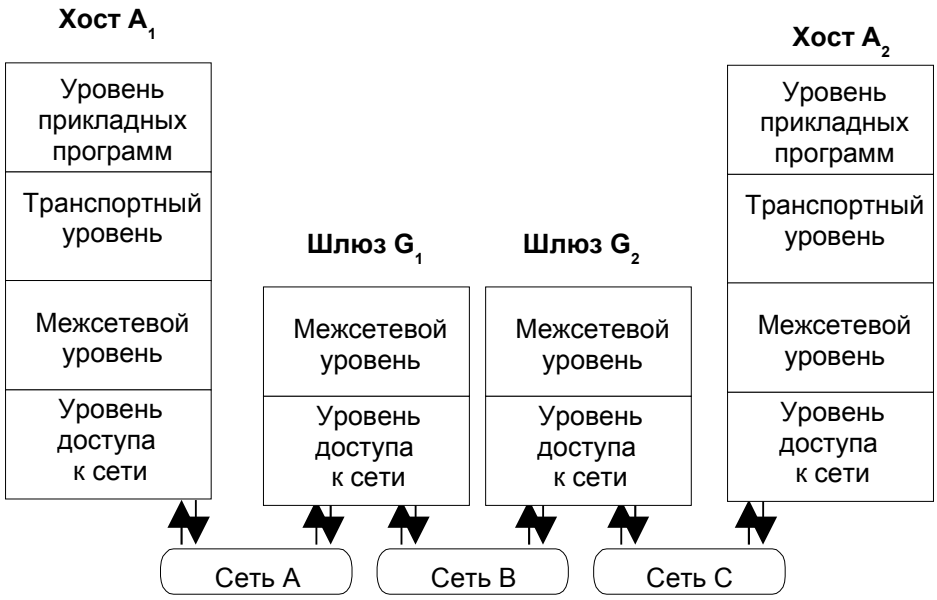


Рис. 63. Маршрутизация дейтаграмм.

Транспортный уровень. 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 ARPA-NET, ставило перед собой задачу разработку 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 дисбаланс различных сетей с точки зрения надежности каналов актуальна и по сей день, а разработанные протоколы решали эту проблему.

Среди протоколов транспортного уровня необходимо отметить протоколы TCP и UDP. 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 сильно падать, особенно в линиях связи с плохими техническими характеристиками. Итак, этот протокол подразумевает, что для каждого полученного пакета адресат обязан отправить подтверждение о доставке. К этому необходимо доставить, 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). Данный протокол подразумевает отправку пакетов по сети без гарантии их доставки (он выбрасывает пакет и сразу же «забывает» о нем).

Уровень прикладных программ. На этом уровне находятся протоколы, часть которых опираются на протокол TCP, а часть — на UDP.

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; протокол 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. 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 должны контролироваться, с одной стороны, на прикладном уровне, а с другой стороны, они 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 доменные имена и обратно.

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 ее определение детально.

Начнем с того, что операционная система обеспечивает контроль за **существованием** ресурсов. 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).

И, наконец, **распределение**: под этим будем понимать 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.

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 в один и тот же момент времени могут принадлежать двум и более процессам, и их 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 модели организации этого выделения. Первый способ — это предварительная декларация ресурсов. В этом случае до ввода программы в систему и формирования для нее процесса описывается 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 предварительны декларированы. Если в системе нет заказанного ресурса, то она, скорее всего, не станет запускать процесс, который запросил этот ресурс.

Вторая модель — это динамическое дополнение 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 смешанные 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.

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

И, наконец, это **предсказуемость** системы, являющееся 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, а также подразумевать устойчивость системы к возникновению подобных обстоятельств.

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 (1.3.1).

Рис. 64. Структурная организация ОС.

Ядро (***kernel***) ОС — это часть ОС, в которой реализована функциональность ОС; ядро работает в режиме супервизора, т.е. в привилегированном режиме, и резидентно (постоянно) размещается в оперативной памяти. Итак, по определению ядро обеспечивает реализацию некоторого набора функций операционной системы. Это может быть очень большой набор функций, а может 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.

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 функций, поэтому модификация функционального набора достаточно затруднительна. 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 1.3.1 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.

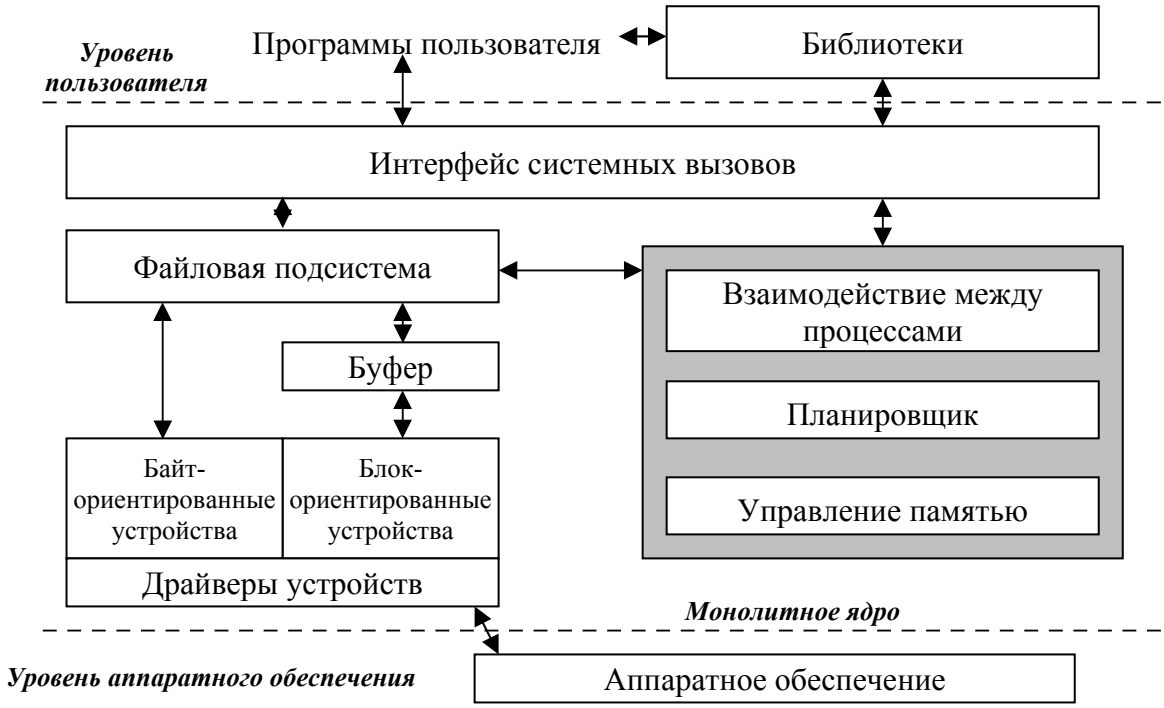


Рис. 65. Структура ОС с монолитным ядром.

Что касается достоинств данного подхода, то можно утверждать, что для конкретного состава функциональности и логики ядра это будет наиболее эффективное решение. 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** (1.3.1). Функционирование операционных систем подобного типа основывается на использовании т.н. ***микроядра***. В этом случае выделяется минимальный набор функций, которые включаются в ядро. Все оставшиеся функции представляются в виде драйверов, которые подключаются к ядру посредством некоторого стандартного интерфейса.

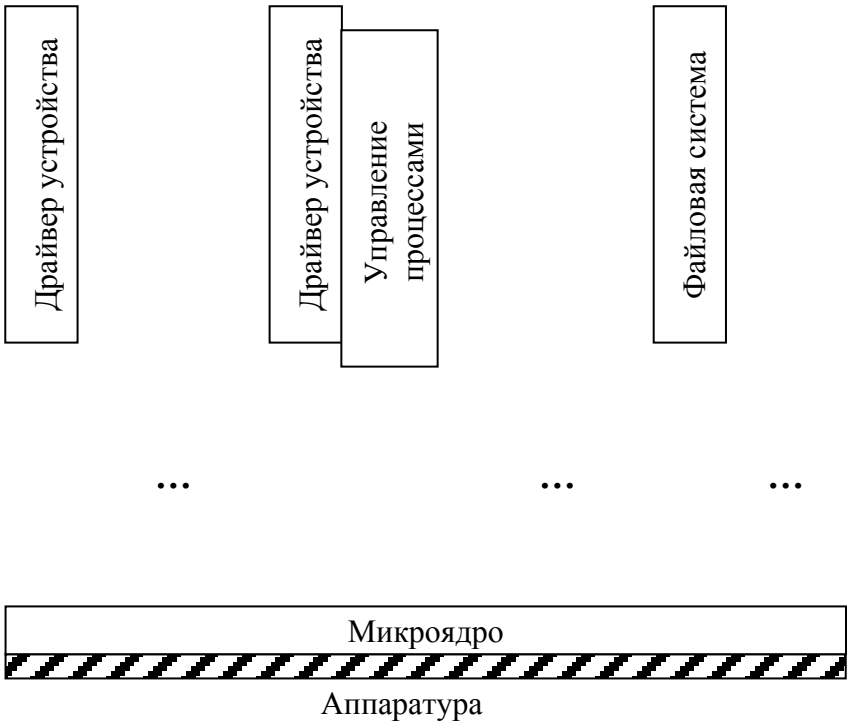


Рис. 66. Структура ОС с микроядерной архитектурой.

Такая архитектура получается хорошо расширяемой, она почти не имеет никаких ограничений по количеству подключаемых драйверов и их функциональное наполнение, 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 следуют из достоинств 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 (пускай, сначала это будут виртуальный диск). После чего файловая система обращается к микроядру, которое, в свою очередь, находит драйвер виртуального диска, передает 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 запросов, следующих от драйвера через микроядро к другому драйверу, благодаря чему эффективность системы снижается.

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

1.3.2 Логические функции ОС

Рассматривая ОС, ее функциональность 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*.

На уровне ***управления процессами*** решаются проблемы формирования процессов, поддержание жизненного цикла процесса, организация взаимодействия процессов, т.е. организация взаимодействия процесса с системой в целом и с другими процессами в частности.

Блок ***управления оперативной памятью*** реализует программную поддержку той или иной стратегии организации памяти. При необходимости на этом уровне реализуется поддержка аппарата виртуальной памяти, решается задача выделения и изъятия памяти у процесса.

Функции ***планирования*** можно понимать с разных точек зрения. Можно понимать планирование в узком смысле слова, т.е. планирование центрального процессора (т.е. планирование доступа процессов к центральному процессору). На самом деле, функций планирования большое множество, поскольку применять планирование приходится при организации многих механизмов операционной системы. Так, упоминавшаяся только что задача изъятия памяти у процессов является задачей планирования, поскольку ставится вопрос, по какому принципу будет происходить это изъятие. Взаимодействие с внешними устройствами тоже не может обойтись без решения задач планирования: так или иначе, поток заказов на обмен, поступающих в системе, может превосходить пропускную способность устройства, образуется конкуренция по доступу к устройству — выстраивается очередь заказов на обмен. Соответственно, ставится вопрос, как организовать обработку этой очереди. Возможны различные стратегии: FIFO, LIFO и пр. — и для каждой из них будет свой результат. Итак, сфера применения решения задач планирования достаточно широка, просто в одних случаях планирование рассматривают в рамках какой-либо функциональности, а в других случаях — отдельно.

Блок ***управления данными и файловой системой*** также является достаточно важным, поскольку ни один процесс не сможет без него функционировать. На этом уровне применяются множество различных стратегий, организаций и пр., о чем речь пойдет позже. Блок ***управления внешними устройствами***, подобно блоку управления оперативной памятью, зачастую оказывается скрытым для пользователей системы, в некоторых случаях он интегрирован в файловую систему, как это сделано в ОС Unix. На этом уровне также решаются множество специфических задач: задача кэширования обменов, задача повышения надежности обменов и пр.

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 минимизация потерь работы центрального процессора. 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, требующие определенного объема времени работы процессора. Как следует из названия, эти системы оперируют термином ***пакет программ***.

Пакет программ — это некоторая совокупность программ, которые необходимо обработать системе. 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 осуществить обмен, 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. 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 ОС 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 время отклика системы на запрос пользователя (положим, если пользователь набирает текст в текстовом редакторе, т.е. будет важно, что 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.

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 фиксируется некоторый набор 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. Выделяют различные группы систем реального времени (жесткого времени, 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. **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, почтовая приложение может быть распределенным: есть функции перемещения, есть сервер-получатель, есть клиентская часть, обеспечивающая интерфейс работы пользователя с указанным сервером.

Рис. 67. Структура сетевой ОС.

Вторую категорию составляют ***распределенные*** ОС. **Распределенной операционной** системой считается система, функционирующая на многопроцессорном или многомашинном комплексе, при этом на каждой машине 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.

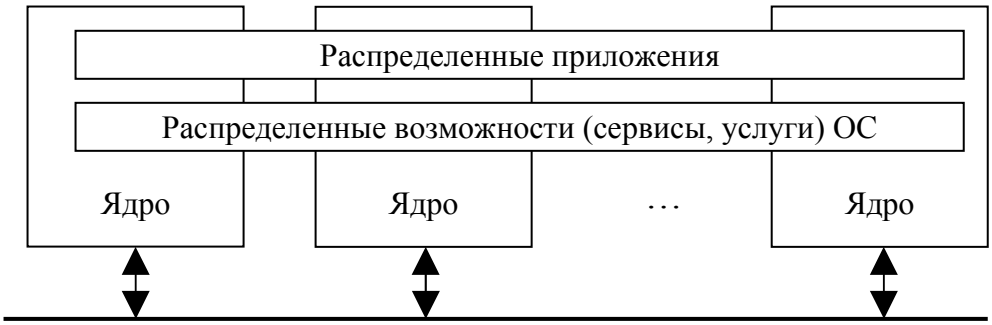


Рис. 68. Структура распределенной ОС.

2 Управление процессами

2.1 Основные концепции

Выше уже встречалось понятие ***процесса*** и некоторые его определения. Итак, 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 доступ к нему могут иметь два и более процесса.

Также ранее отмечалось, что одной из функций логического блока управления процессами ОС является функция поддержания ***жизненного цикла*** процесса. ***Жизненный цикл*** процесса — это те этапы, через которые может проходить процесс в ходе своей обработки и исполнении 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,
- исполнение 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 — важный этап, связанный с возвратом процессом принадлежащих ему ресурсов.

2.1.1 Модели операционных систем

Ниже будем рассматривать некоторую **модельную операционную систему**. Будем считать, что этапы жизненного цикла процесса разделены на два блока. Первый блок — это размещение процесса, или программы, в ***буфере ввода процессов*** (***БВП***). В этом буфере размещаются процессы от момента их формирования, или ввода в систему, до начала обработки его центральным процессором. Второй блок 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 блока, т.к. эта модель отражает наиболее общую картину. Процесс после его формирования не обязательно сразу попадает на процессор, а многие информационные системные структуры образуются только тогда, когда процесс начинает обрабатываться, соответственно, поэтому можно провести разделение по структурной организации. Размеры буферов в различных системах могут варьироваться.

Теперь рассмотрим модели операционных систем того или иного класса систем, и начнем мы рассмотрение модели **пакетной однопроцессной системы** (2.1.1). В подобной системе жизненный цикл процесса состоит всего из трех этапов. Первый этап — 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. 69. 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 — модель **пакетной мультипроцессной системы** (2.1.1). 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 либо процесс 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.

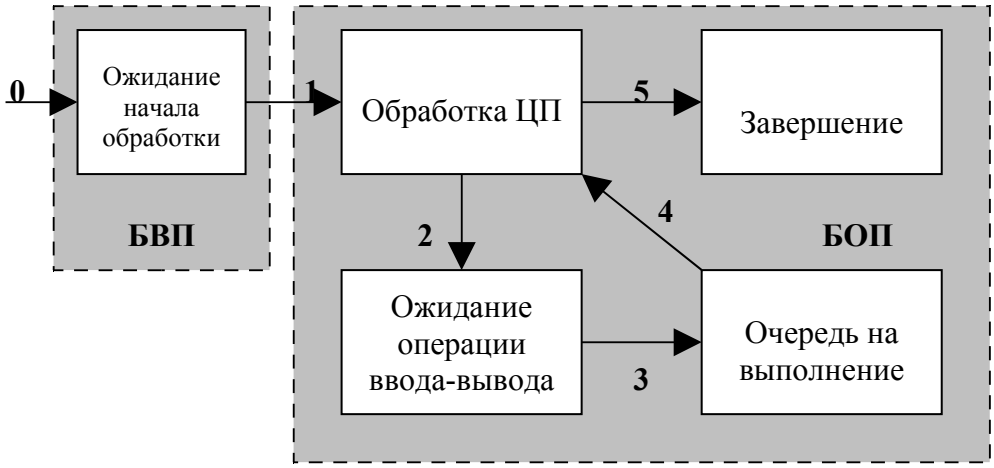


Рис. 70. Модель пакетной мультипроцессной системы. 0 — поступление процесса в очередь на начало обработки ЦП (процесс попадает в БВП). 1 — начало обработки процесса на ЦП (из БВП в БОП). 2 — процесс прекращает обработку ЦП по причине ожидания операции ввода-вывода, поступает в очередь завершения операции 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 системы небольшие изменения, можно получить модель **операционной системы с разделением времени** (2.1.1). Структурно достаточно добавить возможность перехода из состояния обработки центральным процессором в очередь готовых на выполнение процессов. Т.е. система имеет возможность прервать выполнение текущего процесса и поместить его 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 (2.1.1), 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.

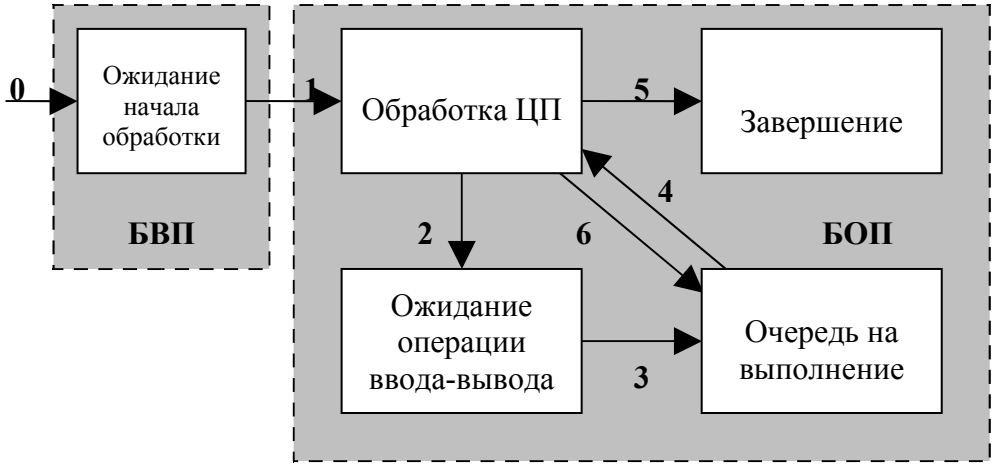


Рис. 71. Модель ОС с разделением времени. 6 — процесс прекращает обработку ЦП, но в любой момент может быть продолжен (истек квант времени ЦП, выделенный процессу). Поступает в очередь процессов, ожидающих продолжения выполнения центральным процессором (БОП).

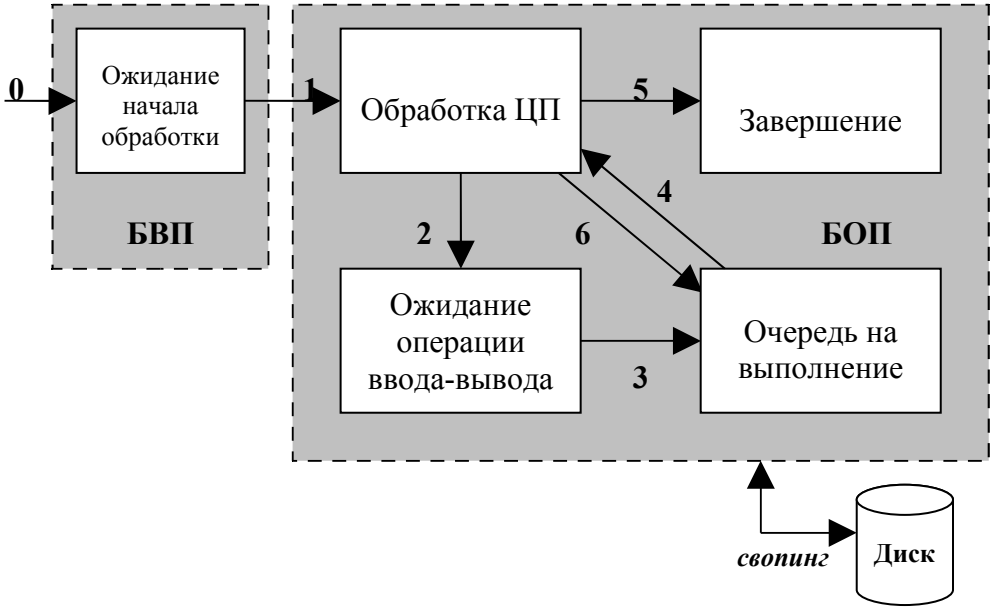


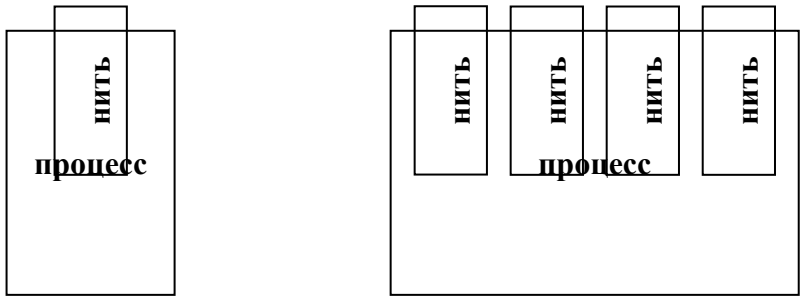
Рис. 72. Модель ОС с разделением времени (модификация). Заблокированный процесс может быть откачан (свопирован) на внешний носитель, а на освободившееся место может быть подкачен процесс с внешнего носителя, который был откачен ранее, либо взят новый.

2.1.2 Типы процессов

Рассматривая процесс в той или иной операционной системе, можно обнаружить, что встречается деление процессов на две категории: т.н. ***полновесные*** процессы и ***легковесные*** процессы, или ***нити***.

Полновесные процессы (иногда их называют просто ***процессы***) — это те процессы, машинный код которых обладает эксклюзивными правами на владение оперативной памятью (т.е. это традиционная однопроцессная программа).

Альтернативой являются т.н. ***легковесные процессы***, известные также как ***нити***, — это процессы, которые могут работать совместно с другими процессами на общем пространстве оперативной памяти. Обычно легковесные процессы реализуются внутри полновесного процесса.



а. б.

Рис. 73. Типы процессов: однонитевая (а) и многонитевая (б) организации.

Тогда традиционную однопроцессную программу, которую мы отнесли к полновесным процессам, можно теперь переопределить как однонитевой процесс, т.е. этому процессу эксклюзивно выделена память, и внутри существует один набор команд, который владеет и работает в этой защищенной области памяти. Многонитевая организация подразумевает выделение процессу защищенной области памяти, но внутри эта область доступна двум и более нитям.

Организуя многонитевые процессы, обычно преследуются следующие цели. Во-первых, это снижение накладных расходов. Как отмечалось выше, смена контекста полновесных процессов является трудоемкой операцией. В то же время, смена контекста нитей в рамках одного процесса является более простой задачей, поскольку не требуется полного переконфигурирования системы.

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 расширить. ***Процесс*** — это совокупность исполняемого кода с собственным адресным пространством, представляющее собой множество виртуальных адресов, которые может использовать процесс, и назначенными ему ресурсами системы, и которая содержит хотя бы одну 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

Говоря о различных механизмах, происходящих в системе, часто затрагивался термин ***контекст процесса***. Под ***контекстом процесса*** мы будем понимать совокупность данных, характеризующих актуальное состояние процесса. Обычно контекст процесса состоит из нескольких компонент:

- **пользовательская составляющая** — это совокупность машинных команд и данных, которые характеризуют выполнение данного процесса;
- **системная составляющая**, которая содержит в себе информацию об именовании, правах процесса, т.е. различного рода учетная системная информация, а также содержит информацию о состоянии процесса в точке останова (содержимое регистров, настройки процесса и пр.). Соответственно, о последнем имеет смысл говорить лишь тогда, когда процесс откачан. Во время исполнения процесса обычно говорят об **аппаратной составляющей** контекста (т.е это актуальное состояние регистров, актуальные настройки процесса и пр.). Таким образом, когда

процесс обрабатывается на процессоре, то актуальна аппаратная составляющая, когда процесс отложен — актуальна системная составляющая.

2.2 Реализация 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, рассмотрим ее теперь более детально.

С точки зрения понимания термина ***процесса*** в ОС Unix данное понятие можно определить двояко. В первом случае ***процесс*** можно определить как объект, зарегистрированный в таблице процессов. Таблица процессов является одной из специальных системных таблиц, которая, очевидно, 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, размер которой является параметром настройки ОС, и, соответственно, количество процессов в системе является системным ресурсом. Таблица устроена позиционным образом, а это означает, что 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. Соответственно, каждая запись таблицы (2.2.1) имеет ссылку на контекст процесса, который структурно состоит из пользовательской, системной и аппаратной составляющих.

Пользовательская составляющая — это тело процесса. В нее входит сегмент кода, который содержит машинные команды и неизменяемые константы. Сегмент кода — это обычно не изменяемая программным способом часть тела процесса. Также в пользовательскую составляющую входит сегмент данных, в который входит 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 в функциях, реализуются автоматические и регистровые переменные, а также в этой области организуется динамическая память (т.н. куча).

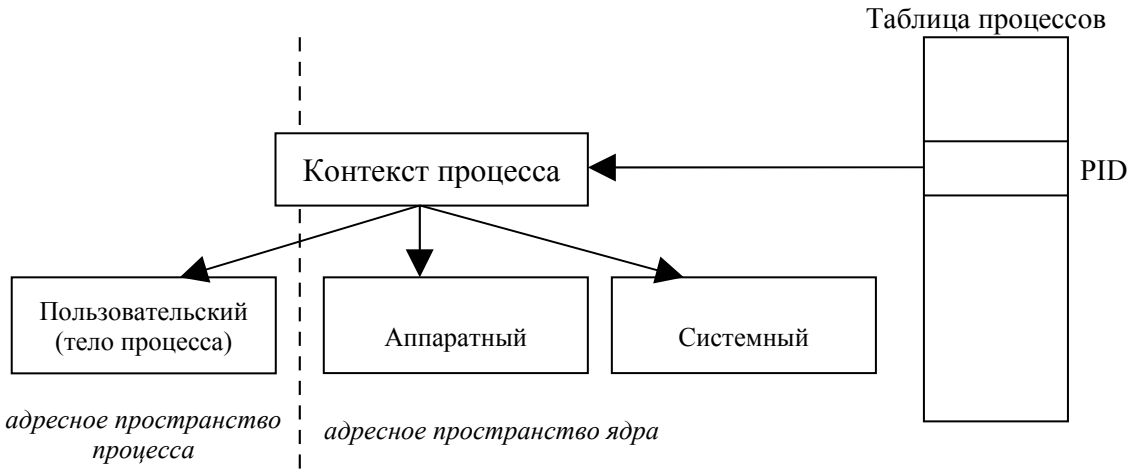


Рис. 74. Таблица процессов в ОС Unix.

В ОС Unix реализован такая возможность, как **разделение сегмента кода** (2.2.1). Допустим, в системе работает несколько (пускай, 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 оказывается своим. 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.

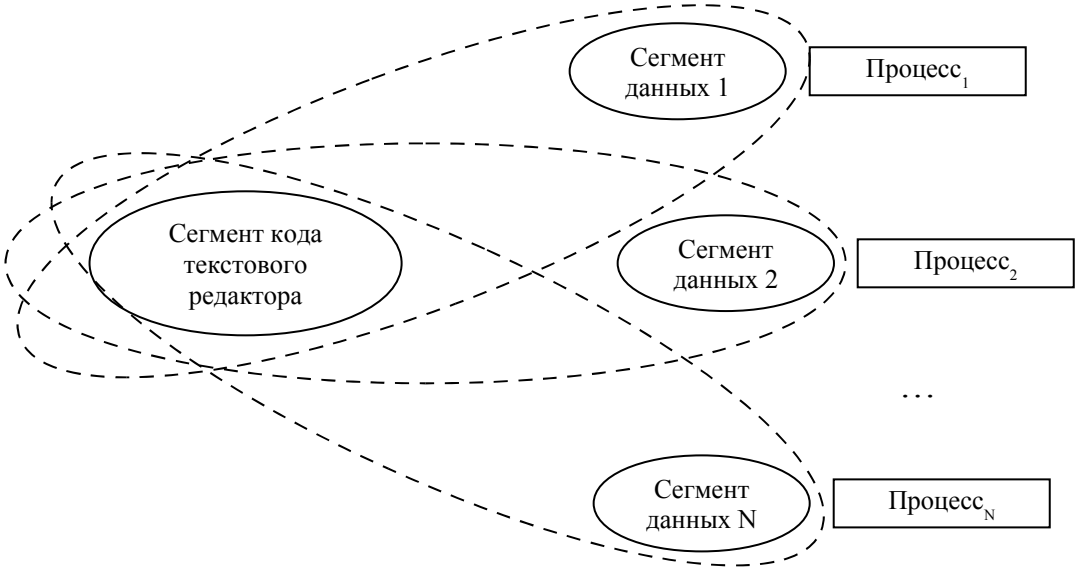


Рис. 75. Разделение сегмента кода.

Аппаратная составляющая включает в себя все регистры, аппаратные таблицы процессора и пр., характеризующие актуальное состояние процесса в момент его выполнения на процессоре.

Системная составляющая содержит системную информацию об идентификации процесса (т.е. идентификация пользователя, сформировавшего процесс), системная информация об открытых и используемых в процессе файлах и пр., а также сохраненные значения аппаратной составляющей. Итак, системная составляющая хранит множество параметров, рассмотрим некоторые из них.

Одними из таких параметров являются реальный и эффективный идентификаторы пользователя-владельца. В ОС Unix формирование 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).

Итак, следуя второй трактовке, ***процессом*** называется объект, порожденный системным вызовом *fork()*. 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

Рассмотрим теперь, 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 процесс получает уникальный идентификатор, а также в системе создается контекст для дочернего процесса.

Новый процесс наследует почти все атрибуты исходного родительского процесса, за исключением идентификационной информации (т.е. у дочернего процесса, в частности, свой идентификатор PID и иной идентификатор родительского процесса). Среди прочего дочерний процесс наследует открытые отцовским процессом файлы. На это свойство в ОС Unix опираются многие механизмы. Хотя необходимо отметить, что наследуются необязательно все открытые файлы: если некоторый файл открывался в специальном режиме, то при формировании дочернего процесса этот файл для него будет автоматически закрыт.

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

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

```
pid_t fork(void);
```

Соответственно, при успешном завершении сыновнему процессу возвращается значение 0, а родительскому процессу — идентификатор порожденного процесса; в случае ошибки возвращается -1, а в переменной *errno* будет храниться код ошибки. Поскольку дочерний процесс является копией отцовского процесса, то возникает проблема, как отличить, какой из процессов в данный момент обрабатывается. Анализируя результат, возвращаемый системным вызовом *fork()*, можно определить, что текущий процесс является предком или потомком.

Рассмотрим **пример** (2.2.2). Пускай 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 дочерний 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 дочерний 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.

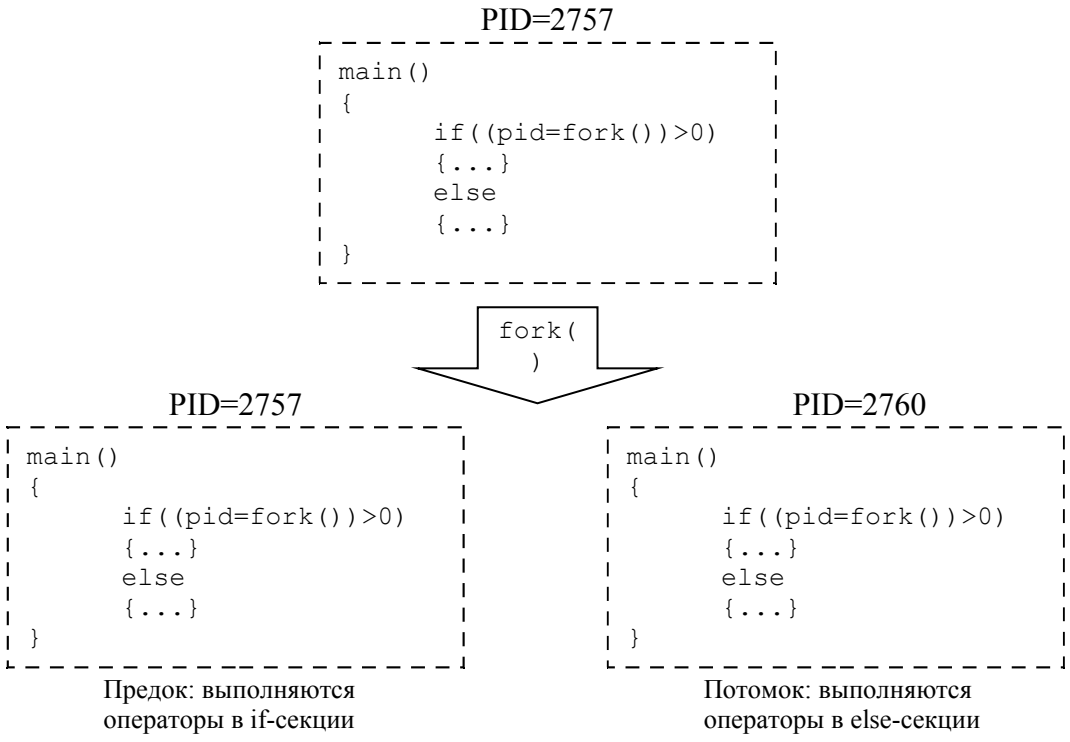


Рис. 76. Пример использования системного вызова *fork()*.

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.

```
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());

}
```

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 исполняемого файла. Параметры arg0, …, argn являются аргументами программы, передаваемые 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.

Итак, 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** (2.2.2). 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()).

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.

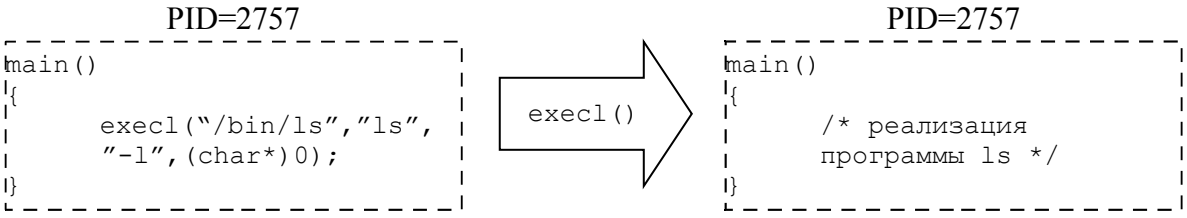


Рис. 77. Пример использования системного вызова *execl()*.

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()*, но в ОС Unix обычно применяется связка вызовов *fork-exec*.

Для иллюстрации сказанного рассмотрим еще один пример. 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 дочернем процессе (PID = 2760) управление передается на первую инструкцию then-блока, где происходит обращение к системному вызову *execl()*. После чего тело дочернего процесса меняется на тело команды **ls**.

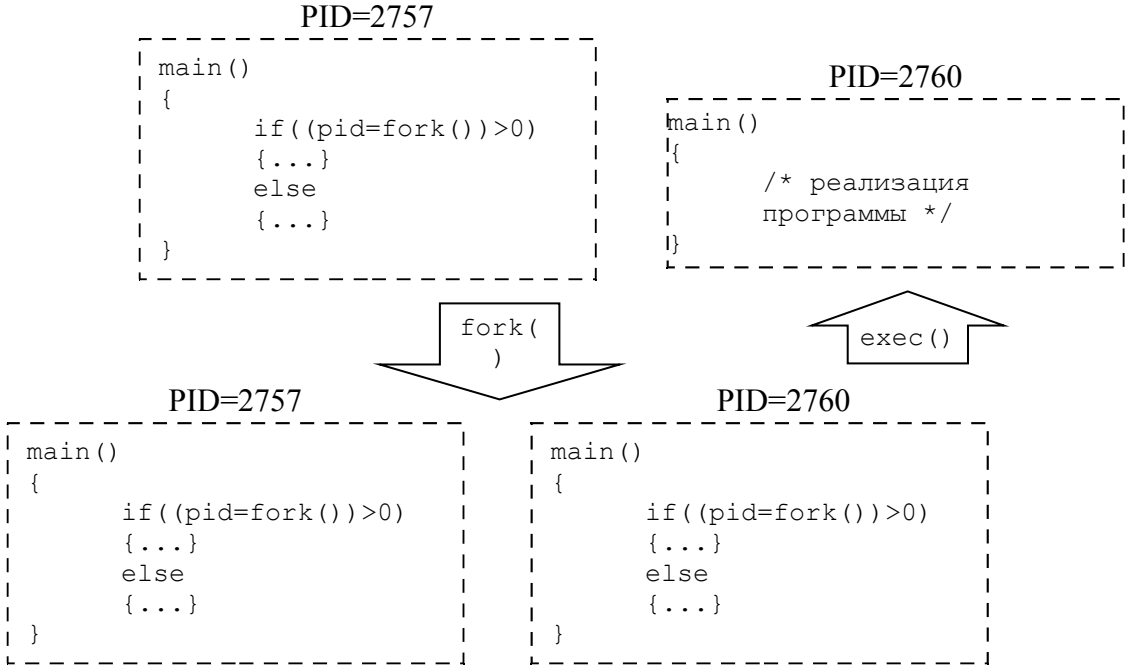


Рис. 78. Пример использования схемы *fork-exec*.

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. Прежде всего, речь пойдет о завершении процесса. Вообще говоря, процесс может завершиться по одной из двух причин. Первая причина связана с возникновением в процессе сигнала. Сигнал можно считать программным аналогом прерывания, и речь о них пойдет ниже при обсуждении вопросов взаимодействия процессов. Сигнал может быть связан с тем, что в процессе произошло деление на ноль, или сигнал может прийти от другого процесса с указанием незамедлительного завершения. 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(). В последнем случае компилятор заменит действие оператора return обращением к системному вызову *_exit()*.

```
#include <unistd.h>


void _exit(int status);
```

Системный вызов *exit()* не возвращает никакого значения, поскольку он всегда прорабатывает. А через его единственный параметр status возвращается т.н. программный код завершения. Это значение передается операционной системе как код завершения программы. В принципе значение этой переменной может быть произвольным, но в системе есть договоренность, что возврат нулевого значения сигнализирует об успешном завершении процесса, остальные значения трактуются как ошибочное завершение (в частности, процесс может возвращать некий код ошибки).

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. Во-первых, корректно освобождаются ресурсы (закрываются файлы, освобождаются оперативная память и пр.). Во-вторых, поскольку ОС Unix является «семейственной» системой (у каждого процесса может быть целая иерархия потомков), стоит проблема, какой процесс считать отцовским после завершения данного родительского процесса, и в ОС Unix принято решение, что все сыновние процессы усыновляются процессом с номером 1. И, наконец, процессу-предку от данного завершаемого процесса передается сигнал SIGCHLD, но в большинстве случаев его игнорируют.

Симметричную картину иллюстрирует системный вызов *wait()*, который обеспечивает в процессе получение информации о факте завершения одного из его потомков.

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

#include <sys/wait.h>


pid_t wait(int *status);
```

Обычно при обращении к системному вызову *wait()* возможны следующие варианты. Во-первых, если до обращения к этому системному вызову какие-то сыновние процессы уже завершились, то процесс получит информацию об одном из этих процессов. Если же у процесса нет дочерних процессов, то, обращаясь к системному вызову *wait()*, процесс сразу получит соответствующий код ответа. В-третьих, если у процесса имеются дочерние процессы, но ни один из них не завершился, то при обращении к указанному системному вызову данный отцовский процесс будет блокирован до завершения одного из своих сыновних процессов.

По факту завершения одного из процессов родительский процесс при обращении к системному вызову *wait()* получает следующую информацию. В случае успешного завершения возвращается идентификатор PID завершившегося процесса, или же -1 — в случае ошибки или прерывания. А через параметр status передается указатель на целочисленную переменную, в которой система возвращает процессу причины завершения сыновнего процесса. Данный параметр содержит в старшем байте код завершения процесса-потомка (***пользовательский код завершения процесса***), передаваемый в качестве параметра системному вызову *exit()*, а в младшем байте — индикатор причины завершения процесса-потомка, устанавливаемый ядром ОС Unix

(***системный код завершения процесса***). Системный код завершения хранит номер сигнала, приход которого в сыновний процесс вызвал его завершение.

Необходимо сделать замечание, касающееся системного вызова *wait()*. Данный системный вызов не всегда отрабатывает на завершении дочернего 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 удаляется запись о нем из таблицы процессов.

Рассмотрим **пример использования системного вызова *wait()***. В данном случае 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();

     }
```

```
}
```

Пусть существуют три исполняемых файла print1, print2, print3, каждый из которых только печатает текст first, second, third соответственно, а код вышеприведенного 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();

     }

     if ((fork()) == 0) /*второй процесс-потомок*/

     {

          execl("/bin/echo","echo","this is","string 2",0);

          exit();

     }

     /*процесс-предок*/

     printf("process-father is waiting for children\n");
```

```
        while(wait() != -1);

        printf("all children terminated\n");

        exit();

}
```

2.2.3 Жизненный цикл процесса. Состояния процесса

Рассмотрим обобщенную и несколько упрощенную схему жизненного цикла процессов в ОС Unix (2.2.3). Можно выделить целую совокупность состояний, в которых может находиться процесс.

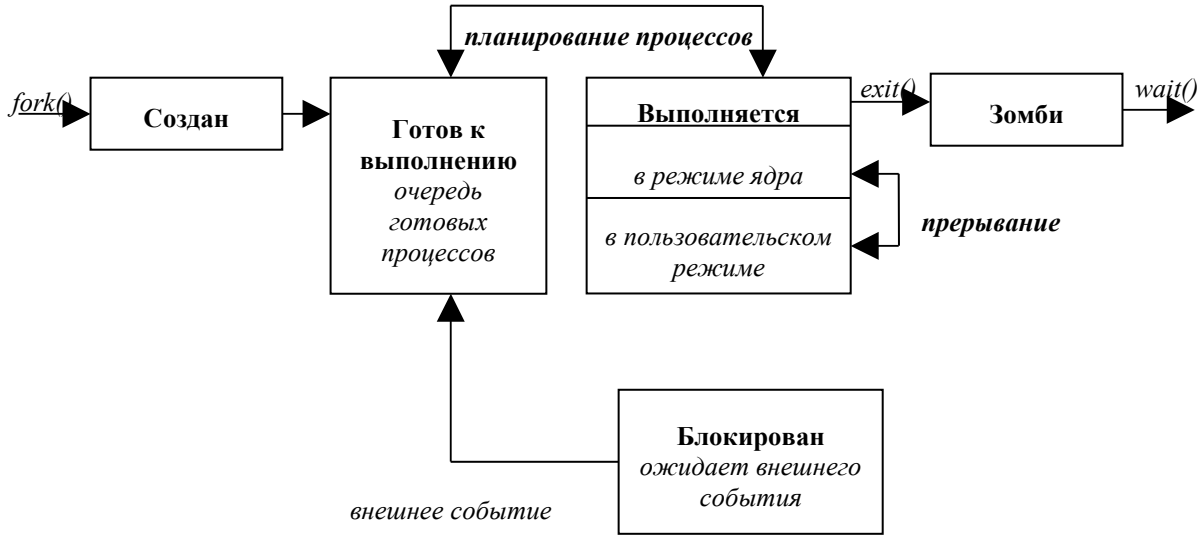


Рис. 79. Жизненный цикл процессов.

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), который попадает в БВП (о котором речь шла выше). В ОС Unix БВП состоит из одного процесса, который после создания попадает в БОП. Итак, после создания процесса он получает статус, что он готов к выполнению. Из этого состояния по решению планировщика он может переходить в состояние выполнения, а затем обратно в состояние готовности к выполнению в случае исчерпания кванта времени обработки на процессоре. В состоянии выполнения процесс может работать как в пользовательском режиме, так и в режиме ядра, или супервизора. В режиме ядра процесс работает при обращении к системному вызову, когда ядро выполняет некоторые действия для конкретного процесса.

Затем из режима выполнения процесс может перейти в состояние блокировки. В этом состоянии процесс находится, ожидая завершения некоторого действия: например, ожидая завершения внешнего взаимодействия. Соответственно, по возникновению 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, 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.

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 процесс в ОС Unix создается по приведенной схеме, но существуют два процесса с номерами 0 и 1, которые являются исключениями из данного правила.

Рассмотрим детально, как формируются данные процессы, но для этого необходимо разобраться, что происходит в системе при включении компьютера. Практически во всех компьютерах имеется область памяти, способная постоянно хранить информацию — т.н. **постоянное запоминающее устройство** (**ПЗУ**). В этой области памяти находится т.н. **аппаратный загрузчик компьютера**. Данный загрузчик в общем случае имеет информацию о перечне и приоритетах системных устройств компьютера, которые априори могут содержать операционную систему. Приоритет определяет тот порядок, в котором аппаратный загрузчик осуществляет перебор устройств по списку в поисках программного загрузчика операционных систем. 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 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 указанный 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.

Первым делом происходит инициализация системы, которая включает в себя установку начальных параметров в аппаратных интерфейсах: установку системных часов, установка диспетчера оперативной памяти, установка средств защиты оперативной памяти. Затем, исходя из параметров настройки операционной системы, осуществляется формирование системных структур данных (в частности, создается таблица процессов). После этого ядро создает нулевой процесс. 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. Этот нулевой процесс в общем случае соответствует ядру (это процесс ядра), но этот процесс имеет особенность: он не имеет сегмента кода. Это означает, что нулевая запись таблицы процессов ссылается на контекст, в котором отсутствует ссылка на сегмент кода процесса. Нулевой процесс существует на всем протяжении функционирования ОС, причем он иллюстрирует нештатное формирование процесса в системе.

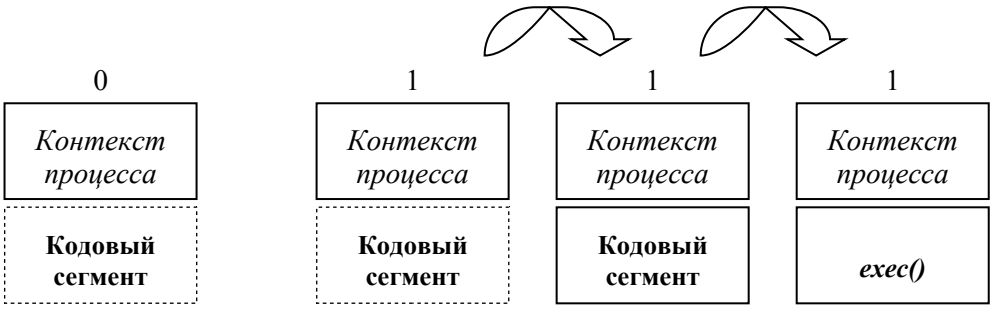


Рис. 80. Формирование нулевого и первого процессов.

Следующим этапом ядро начинает формирование первого процесса, который также создается нестандартным образом, при этом выполняются следующие действия. Ядро осуществляет копирование нулевой записи в первую. После чего для первой записи выделяется пространство оперативной памяти и создается тело процесса. В тело процесса записывается код системного вызова *exec()*, после этого происходит внутри первого процесса обращение к этому системному вызову с параметром /etc/init. Таким образом, можно отметить, что сам первый процесс формируется нестандартным путем, но его тело его в конце уже формируется «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 соответствующую стратегию организации работы системы: либо это однопользовательская система, либо многопользовательская. Эта стратегия определяется параметрами, которые возникают на стадии загрузки ядра и инициализации системы. Соответственно, система опознает один из подключенных терминалов как системную консоль. Если система однопользовательская, то происходит подключение интерпретатора команд к системной консоли. 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**. Процесс **getty** — это процесс, который обеспечивает работу конкретного терминала. Заметим, что процесс **init** создает процесс **getty** уже стандартным способом, и после вообще все процессы создаются лишь по схеме *fork-exec*.

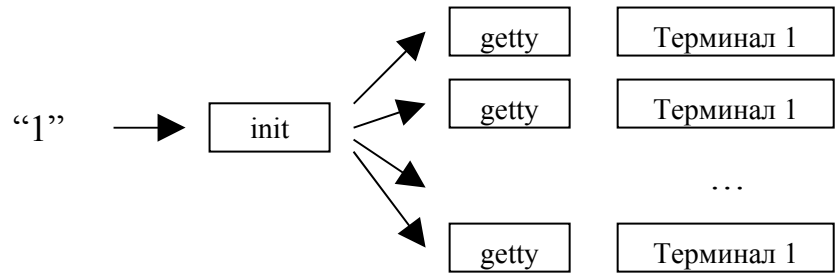


Рис. 81. Инициализация системы.

После старта процесс **getty** печатает на экране приглашение ввести логин (2.2.4). После того, как пользователь вводит логин, процесс **getty** загружает на свое место программу **login**. Соответственно, программа **login** запрашивает ввода пароля, который после ввода и проверяет. В первых версиях ОС Unix все пароли хранились в зашифрованном виде в файле passwd. Если введенный пароль оказывается верным, программа **login** загружает параметры работы конкретного пользователя, загружает интерпретатор команд (**shell**), и пользователь может начинать работать в системе. Заметим, что тип загружаемого интерпретатора команд также задается среди параметров работы данного пользователя. А, вообще говоря, в настройках вместо интерпретатора команд

может присутствовать любой исполняемый файл, например, это может быть менеджер по обслуживанию СУБД, функционирующей в системе.

Сеанс работы пользователя с системой представляется в виде файла, с которым происходят операции чтения и записи. Соответственно, работа с системой заканчивает закрытием файла — подачей символа EOF (end of file), этот код нажатия комбинации клавиш Ctrl+D на клавиатуре. После передачи этого символа интерпретатор завершается. Как только оказывается, с терминалом не связан ни один процесс, процесс **init** запускает новый процесс **getty**, который ассоциируется с этим терминалом, который, в свою очередь, снова печатает на экране приглашение ввести логин.

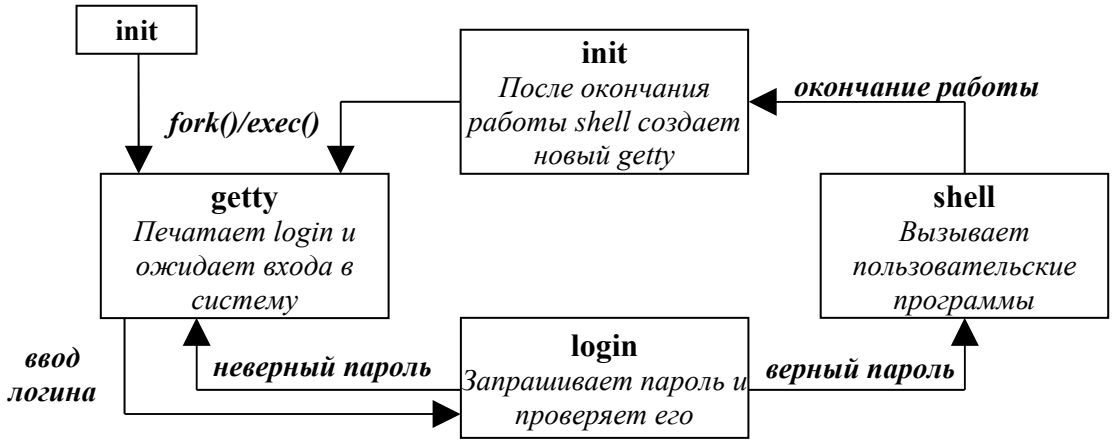


Рис. 82. Схема работы пользователя с ОС Unix.

2.3 Планирование

2.4 Взаимодействие процессов

2.4.1 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 время их выполнения хотя бы частично перекрываются. Т.е. можно сказать, что все процессы, находящиеся в буфере обрабатываемых процессов, являются параллельными, т.к. в той или иной степени их выполнения во времени пересекаются, они существуют одновременно. 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 множество независимых ресурсов, т.е. те ресурсы, которые принадлежат независимым процессам, в пересечении дают пустое множество. Альтернативой независимым процессам являются взаимодействующие процессы — те процессы, пересечение множеств ресурсов которых непустое. При этом одни процессы могут оказывать влияние на другие процессы, участвующие в этом взаимодействии.

Совместное использование ресурсов двумя и более процессами, когда каждый из них некоторое время владеет этими ресурсами, называется ***разделением ресурсов*** (как аппаратных, так и программных, или виртуальных). Разделяемый ресурс, использование которого организовано таким образом, что он может быть доступен в каждый момент времени только одному из взаимодействующих процессов, называется ***критическим ресурсом***. Соответственно, часть

программы, в рамках которой осуществляется работа с критическим ресурсом, называется ***критической секцией***.

При организации корректного взаимодействия процессов очень важно требование, декларирующее, что результат работы взаимодействующих процессов не должен зависеть от порядка переключения выполнения между этими процессами, т.е. от соотношения скорости выполнения данного процесса со скоростями выполнения других процессов.

Рассмотрим **пример** (2.4.1). 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 ввел значение Y этой же переменной **in**. Далее оба процесса читают эту переменную, и в обоих случаях процессы прочтут значение Y. Возможно, что процессы могли совершить эти действия в ином порядке (поскольку по-другому могли быть обработаны на процессоре), и результат был бы отличным от этого. Соответственно, подобная ситуация, когда процессы конкурируют за разделяемый ресурс, называются ***гонкой процессов*** (***race conditions***).

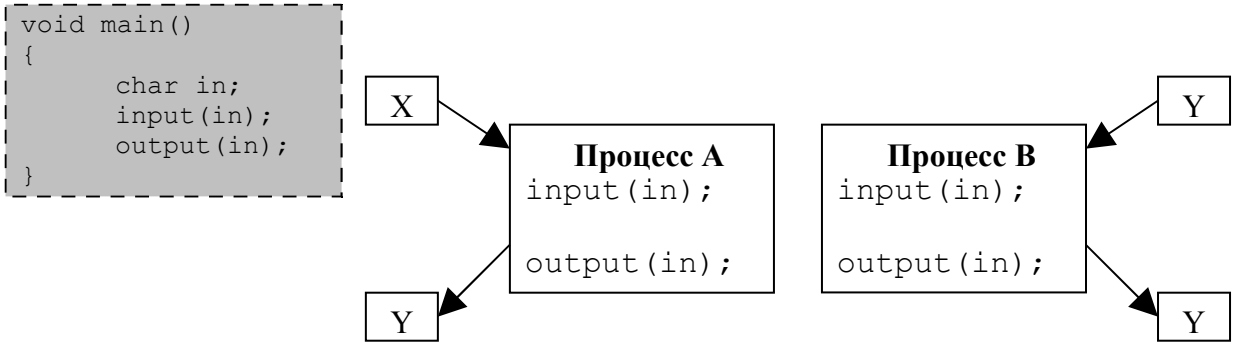


Рис. 83. Гонка процессов.

Для минимизации проблем, возникающих при гонках, используется ***взаимное исключение*** — такой способ работы с разделяемым ресурсом, при котором в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ. Для организации модели взаимного исключения используются различные модели синхронизации. Прежде, чем рассматривать их, необходимо отметить те проблемы, которые могут возникать при организации взаимного исключения — это ***тупики*** и ***блокировки***.

Блокировка — это ситуация, когда доступ к разделяемому ресурсу одного из взаимодействующих процессов не обеспечивается за счет активности более приоритетных процессов. Отметим следующее. Рассмотрим некоторую модель доступа к разделяемому ресурсу, построенную на приоритетах, когда более приоритетный запрос на обращение к ресурсу будет обработан быстрее, чем менее приоритетный. И пусть в этой модели работают два процесса, у которого приоритеты доступа к разделяемому 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 «клинчевая», когда из-за некорректной организации доступа к разделяемым ресурсам происходит взаимоблокировка. Рассмотрим **пример тупиковой ситуации** (2.4.1).

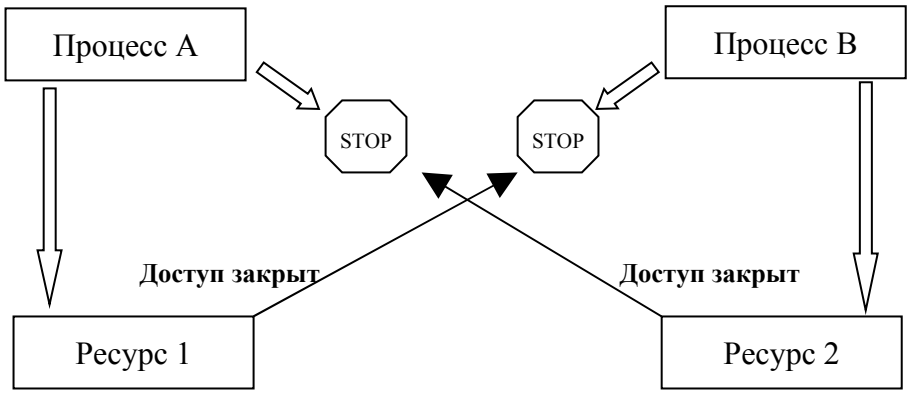


Рис. 84. Пример тупиковой ситуации (deadlock).

Предположим, что есть два процесса A и B, а также пара критических ресурсов. Пускай 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. Пусть также в это время процесс B войдет в критическую секцию ресурса 2. И этот ресурс также будет блокирован для доступа другим процессам. Пускай процесс A, не выходя из критической секции ресурса 1, пытается захватить ресурс 2. Но последний уже захвачен процессом B, и процесс A блокируется. Аналогично, пускай процесс B, не освобождая ресурс 2, пытается захватить ресурс 1 и также блокируется. Это пример простейшего тупика. Из него процессы никогда не смогут выйти. Соответственно, решением в данном случае может быть перезапуск системы или уничтожение обоих или одного из процессов.

2.4.2 Способы организации взаимного исключения

В этом разделе речь пойдет о способах, позволяющих обеспечить работу с критическими ресурсами, т.е. тот способ работы с разделяемым ресурсом, при котором в любой момент времени с ним может работать не более одного процесса, остальные процессы будут заблокированы. В настоящий момент известно множество механизмов, среди которых мы рассмотрим **семафоры Дейкстры**, **мониторы Хоара** и **аппарат передачи сообщений**.

Семафоры Дейкстры — это формальная модель организации доступа, предложенная голландским ученым Дейкстрой, которая основывается на следующей концепции. Имеется специальный тип данных — ***семафор***. Переменная типа ***семафор*** может иметь целочисленные значения. Над этими переменными определены следующие операции: **down(**S**)** (или **P(**S**)**) и **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, один из них разблокировывается и завершает выполнение операции **down**, т.е. вновь уменьшает значение семафора. Выбор процесса никак не оговаривается.

При этом 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 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 обеспечивает взаимное исключение.

В приведенном ниже (2.4.2) **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. 85. Пример двоичного семафора.

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

Мониторы Хоара — модель синхронизации, в которой, в частности, предпринята попытка обойти требование аппаратной поддержки атомарности упомянутых выше операций. 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 некие процедуры и функции, а также данные, с которыми работают эти процедуры и функции. При этом данный модуль обладает следующими свойствами:

- данные монитора доступны только через процедуры и функции этого монитора;
- считается, что процесс занимает (или входит) монитор тогда, когда он начинает использовать одну из процедур или функций монитора;
- в любой момент времени внутри монитора может находиться не более одного процесса, остальные процессы в зависимости от используемой стратегии поведения либо получает отказ, либо блокируется, становясь в очередь.

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 средствами организации работы в основном в однопроцессорных системах либо многопроцессорных системах с общей памятью. В многопроцессорных системах с

распределенной памятью эти средства не очень подходят. Для них в настоящий момент часто используется механизм ***передачи сообщений***.

Механизм ***передачи сообщений*** основан на двух функциональных примитивах: **send** (отправить сообщение) и **receive** (принять сообщение). Данные операции можно разделить по трем критериям: синхронизация, адресация и длина сообщения.

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

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

Блокирующий receive: процесс-получатель будет блокирован 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. Отметим одно важное свойство аппарата сообщений, заключающееся в том, что в нем явно совмещены средства передачи информации и синхронизации, таким образом, механизм передачи сообщений можно использовать для достижения двух целей.

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

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

Иллюстрацией данной модели может выступать модель MPI — интерфейсы передачи сообщений, на основе которых строятся почти все кластерные системы, т.е. системы с распределенной ОП, но точно также MPI может работать в системах с общей памятью.

2.4.3 Классические задачи синхронизации процессов

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

Обедающие философы (2.4.3). Пускай существует круглый стол, 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. Любой философ может взять обе вилки, покушать, затем положить вилки на стол, после этого вилки может взять его сосед и повторить эти действия. Если мы организуем работу таким образом, что любой философ, желающий поесть, берет сначала левую вилку, затем правую, после чего начинает кушать, то в какой-то момент может возникнуть ситуация тупика (когда каждый возьмет по одной левой вилке, а правая будет захвачена соседом).

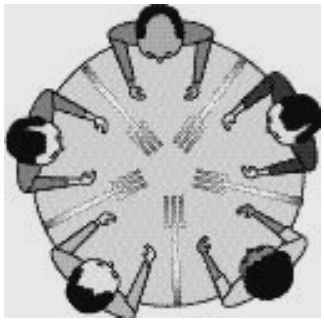


Рис. 86. Обещающие философы.

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

Рассмотрим элементарное решение данной задачи.

```
#define N 5

void Philosopher(int i)

{

     while(TRUE)

     {

          Think();

          /* взятие левой вилки */

          TakeFork(i);

          /* взятие правой вилки */

          TakeFork((i + 1) % N);

          Eat();

          /* освобождение левой вилки */

          PutFork(i);

          /* освобождение правой вилки */

          PutFork((i + 1) % N);

     }

}
```

Как было показано выше, в данном случае возможно появление ситуации, когда произойдет взаимоблокировка философов. Рассмотрим иное решение.

```
/* количество философов */

#define N 5
```

```
/* Номера левого и правого */
#define LEFT (i-1)%N
#define RIGHT (i+1)%N
/* состояния философов:
«думает»,
«желает поесть»,
«кушает»
*/
#define THINKING 0
#define HUNGRY 1
#define EATING 2

/* переопределяем тип СЕМАФОР */
typedef int semaphore;
/*
массив состояний каждого из философов, инициализированный нулями
*/
int state[N];
/* семафор для доступа в критическую секцию */
semaphore mutex = 1;
/*
массив семафоров по одному на каждого из философов,
инициализированный нулями
*/
semaphore s[N];

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

```
{
while(TRUE)
{
     Think();
     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);
}

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 «голоден»). 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 (подымаем 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 этой точке заблокируется до тех пор, пока оба соседа не освободят вилки.

Внутри функции освобождения вилок PutForks первым делом 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 их 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.

Задача «читателей и писателей». 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.

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

```
/* переопределение типа семафор */

typedef int semaphore;

/* семафор для доступа в критическую секцию */

semaphore mutex = 1;

/* семафор для доступа к хранилищу данных */

semaphore db = 1;

/* количество читателей внутри хранилища */

int rc = 0;


/* процесс-читатель */

void Reader(void)

{

     while(true)

     {

          down(&mutex);

          rc = rc + 1;

          if(rc == 1)

               down(&db);
```

```
            up(&mutex);
            ReadDataBase();
            down(&mutex);
            rc = rc - 1;
            if(rc == 0)
                  up(&db);
            up(&mutex);
            UseDataRead();
      }
}

/* процесс-писатель */
void Writer(void)
{
      while(TRUE)
      {
            ThinkUpData();
            down(&db);
            WriteDataBase();
            up(&db);
      }
}
```

В приведенном решении процесс-читатель в каждом цикле своей работы входит в критическую секцию (за счет опускания семафора mutex), увеличивает счетчик читателей, находящихся в хранилище, на 1. Затем проверяет, что если он является первым читателем (т.е. в данный момент он единственный клиент в хранилище), то опускает семафор db, тем самым, препятствуя писателем войти в систему, если они того пожелают. 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 это блокировка происходит внутри критической секции, поэтому остальные читатели будут блокироваться на опускании семафора mutex.) После этого происходит выход из критической секции (подымаем 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 этот читатель является последним клиентом в библиотеке, то происходит поднятие 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, читатель это или писатель) не покинет его. После этого он производит корректировку данных в хранилище и покидает его, поднимая семафор db.

Заметим, что в данном решении если хотя бы один читатель находится внутри системы, то любой следующий читатель беспрепятственно в нее попадет, писатель же будет ожидать, когда все посетители покинут хранилище, т.е. реализована стратегия приоритетности читателя перед писателем.

Данная задача иллюстрирует модель доступа к общему ресурсу процессов, имеющих разные приоритеты.

Задача о «спящем парикмахере». Представим себе парикмахерскую, в которой имеется единственно рабочее кресло и единственный цирюльник. В парикмахерской есть комната ожидания, в которой стоят N кресел, на которых могут сидеть клиенты, ожидающие своей очереди. Если свободных кресел нет, то вновь приходящие клиенты сразу же покидают заведение. Когда посетителей нет, парикмахер может сидеть в своем кресле и дремать.

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

Рассмотрим реализацию данной модели.

```
/* количество стульев в комнате ожидания */

#define CHAIRS 5

/* переопределение типа СЕМАФОР */

typedef int semaphore;

/* наличие посетителей, ожидающих парикмахера */

semaphore customers = 0;

/*

количество парикмахеров, ожидающих посетителей (0 или 1)

*/

semaphore barbers = 0;

/* семафор для доступа в критическую секцию */

semaphore mutex = 1;

/* количество ожидающих посетителей */

int waiting = 0;


/* Брадобрей */
```

```
void Barber(void)
{
     while(TRUE)
     {
          down(&customers);
          down(&mutex);
          waiting = waiting - 1;
          up(&barbers);
          up(&mutex);
          CutHair();
     }
}

/* Посетитель */
void Customer(void)
{
     down(&mutex);
     if(waiting < CHAIRS)
     {
          waiting = waiting + 1;
          up(&customers);
          up(&mutex);
          down(&barbers);
          GetHaircut();
     }
     else
     {
          up(&mutex);
```

```
        }

}
```

Процесс-парикмахер первым делом 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 первым делом 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 он 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 Реализация межпроцессного 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 (одной ОС)**. Первым делом встает 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 внутри одной ОС можно выделить две основных группы решений данной задачи (3.1).

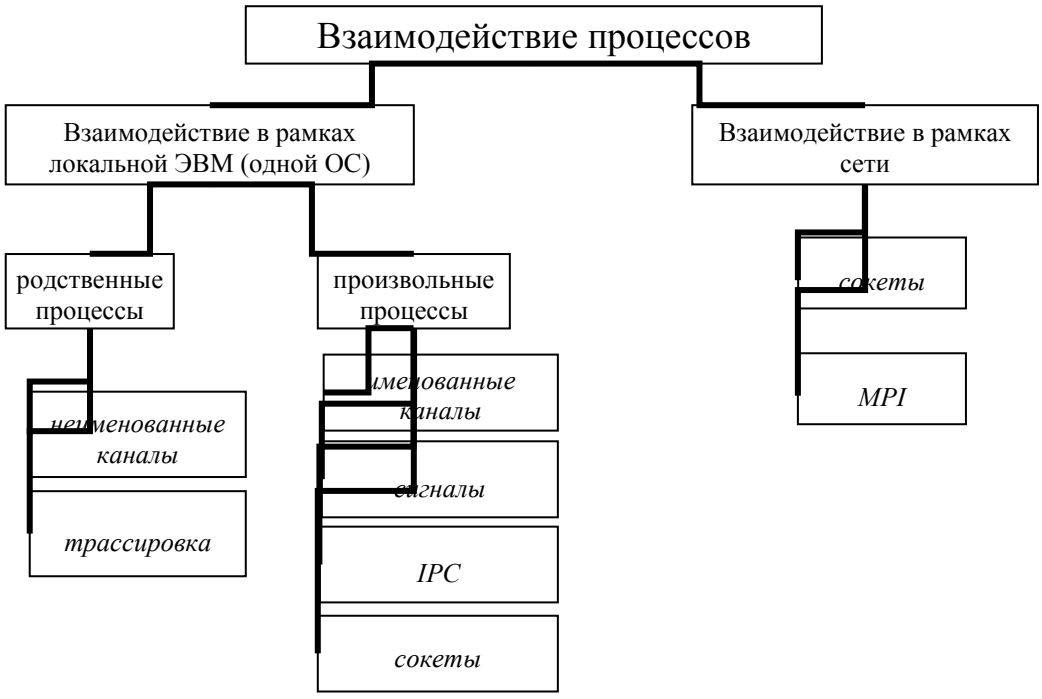


Рис. 87. Способы организации взаимодействия процессов.

Первая группа способов основана на **взаимодействии родственных процессов**. При взаимодействии родственных процессов, т.е. процессов, связанных некоторой иерархией родства, ОС обычно предоставляет возможность наследования некоторых характеристик родительских процессов дочерними процессами. 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 — это 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), предоставляющая 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.

Пускай у нас есть две машины, имеющие сетевые имена A и B, на которых работают соответственно процессы P_1 и P_2. Тогда, чтобы именовать процесс в сети, достаточно использовать связку «сетевой имя машины + имя процесса внутри этой машины». 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 2.4.2). 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 некотором наступившем в системе событии. 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 или ОС. Для иллюстрации приведем следующий пример. Пускай 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. 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».

При получении процессом сигнала возможны 3 типа реакции на него. 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).

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

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

#include <signal.h>


int kill(pid_t pid, int sig);
```

В данной функции первый параметр (pid) — идентификатор процесса, которому необходимо послать сигнал, а второй параметр (sig) — номер передаваемого сигнала. Если первый параметр отличен от нуля, то он трактуется как идентификатор процесса-адресата; если же он нулевой, то сигнал посылается всем процессам данной группы. При удачном выполнении 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 сигнал, реакцию на приход которого надо изменить. Второй аргумент disp определяет новую реакцию на приход указанного сигнала. Итак, disp — это либо определенная пользователем функция-обработчик сигнала, либо одна из констант: SIG_DFL

(обработка сигнала по умолчанию) или SIG_IGN (игнорирование сигнала). В случае успешного завершения системного вызова *signal()* возвращается значение предыдущего режима обработки данного сигнала (т.е. либо указатель на функцию-обработчик, либо одну из указанных констант).

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. В одном случае системный вызов прерывается с отрицательным кодом возврата, а в переменную *errno* заносится код ошибки. Либо системный вызов «дорабатывает» до конца. Мы будем придерживаться первой стратегии (прерывание системного вызова).

Рассмотрим ряд примеров.

Пример. Перехват и обработка сигнала. В данной программе 4 раза можно нажать CTRL+C (послать сигнал SIGINT), и ничего не произойдет. На 5-ый раз процесс обработает сигнал обработчиком по умолчанию и поэтому завершится.

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

#include <signal.h>

#include <stdio.h>


int count = 1;


/* обработчик сигнала */

void SigHndlr(int s)

{

     printf("\nI got SIGINT %d time(s)\n", count++);

if(count == 5)

{

     /* установка обработчика по умолчанию */

     signal(SIGINT, SIG_DFL);

}

}
```

```
/* тело программы */

int main(int argc, char **argv)

{

     /* установка собственного обработчика */

     signal(SIGINT, SigHndlr);

     while(1);

     return 0;

}
```

Пример. Удаление временного файла при завершении программы. 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)

{

     /* удаление временного файла */

     unlink(tempfile);

     /* завершение работы */

exit(0);

}




int main(int argc, char **argv)
```

```
{
     signal(SIGINT, SigHndlr);
     ...
     /* открытие временного файла */
     creat(tempfile, 0666);
     ...
     /* удаление временного файла */
     unlink(tempfile);
     return 0;
}
```

Пример. Программа «будильник». При запуске программа просит ввести имя и ожидает ввод этого имени. А дальше в цикле будут происходить следующие действия. Если по прошествии некоторого времени пользователь так и не ввел имени, то программа повторяет свою просьбу.

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

void Alrm(int s)
{
     printf("\n жду имя \n");
     alarm(5);
}

int main(int argc, char **argv)
{
     char s[80];
     signal(SIGALRM, Alrm);
     alarm(5);
     printf("Введите имя\n");
```

```
    for(;;)
    {
        printf("имя:");
        if(gets(s) != NULL) break;
    }
    printf("OK!\n");
    return 0;
}
```

В данном примере происходит установка обработчика сигнала SIGALRM. 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. Дальше 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. Затем управление возвращается в функцию main в бесконечный цикл. Далее последовательность действий повторяется.

Пример. Двухпроцессный вариант программы «будильник». Данный пример будет повторять предыдущий, но теперь функции ввода строки и напоминания будут разнесены по разным процессам.

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

Void Alrm(int s)
{
    printf("\nБыстрее!!!\n");
}

int main(int argc, char **argv)
{
```

```
    char s[80];
    int pid;

    signal(SIGALRM, Alrm);
    if(pid = fork())
    {
        /* ОТЦОВСКИЙ ПРОЦЕСС */
        for(;;)
        {
            sleep(5);
            kill(pid, SIGALRM);
        }
    }
    else
    {
        /* СЫНОВИЙ ПРОЦЕСС */
        printf("Введите имя\n");
        for(;;)
        {
            printf("имя: ");
            if(gets(s) != NULL) break;
        }
        printf("OK!\n");
        /* уничтожение отцовского процесса */
        kill(getppid, SIGKILL);
    }
    return 0;
}
```

В этом примере происходит установка обработчика сигнала SIGALRM. Затем происходит обращение к системному вызову *fork()*, который породит дочерний процесс. Далее отцовский процесс в бесконечном цикле производит одну и ту же последовательность действий. Засыпает на 5 единиц времени (посредством системного вызова *sleep()*), затем шлет сигнал SIGALRM своему сыну с помощью системного вызова *kill()*. Первым параметром данному системному вызову передается идентификатор дочернего процесса (PID), который был получен после вызова *fork()*.

Дочерний 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 дочернего 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 Неименованные каналы

Неименованный канал (или ***программный канал***) представляется в виде области памяти на внешнем запоминающем устройстве, управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы, т.е. это 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.

Организация данных в канале использует стратегию FIFO, т.е. информация, первой записанная в канал, будет и первой прочитанной из канала. Это означает, что для данных файловых дескрипторов недопустимы работы по перемещению файлового указателя. В отличие от файлов канал не имеет имени. Кроме того, в отличие от файлов неименованный канал существует в системе, пока существуют процессы, его использующие. Предельный размер канала, который может быть выделен процессам, декларируется параметрами настройки операционной системы.

Для создания неименованного канала используется системный вызов *pipe()*.

```
#include <unistd.h>


int pipe(int *fd);
```

Аргументом данного системного вызова является 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.

Неименованные каналы в общем случае предназначены для 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 нижеприведенном примере производится копирование текстовой строки с использованием канала. Этот пример является «надуманным»: он иллюстрирует случай использования канала в рамках одного процесса.

```
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 pipe[1] хранит открытый файловый дескриптор, через который можно писать в канал, а pipe[0] — файловый дескриптор, через который можно писать 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.

Можно отметить следующие **особенности организации чтения** данных из канала. 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 прочесть порцию данных большую, чем та, которая находится в канале, и при этом существуют открытые дескрипторы записи в данный канал, то процесс считывает имеющуюся порцию данных и блокируется до появления недостающих данных. Заметим, что 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, заблокированный на чтение, будет разблокирован, и ему будет передан код конца файла. Соответственно, если заблокированы два и более процесса на чтение, то порядок разблокировки определяется конкретной реализацией. Отметим, что в системе имеется системный вызов *fcntl()*, посредством которого можно установить режим чтения из канала без блокировки.

Теперь рассмотрим **особенности организации записи** в канал. Если процесс пытается записать в канал порцию данных, превосходящую доступное в канале свободное пространство, то часть этой порции данных, равная размеру свободного пространства канала, помещается в канал, и процесс блокируется до появления в канале необходимого свободного пространства. Можно избежать блокировки, используя системный вызов *fcntl()*.

Если процесс пытается записать информацию в канал, с которым в данный момент не связан ни один открытый дескриптор чтения, то процесс получает сигнал SIGPIPE. Таким образом система уведомляет процесс, что произвести операцию записи в канал в настоящий момент нельзя, поскольку нет читающей стороны (а в случае неименованных каналов восстановить ее невозможно).

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 и читает информацию в канал. Но традиционной схемой организации работы с каналом является однонаправленная организация, когда канал связывает два, в большинстве случаев, или несколько взаимодействующих процесса, каждый из которых может либо читать, либо писать в канал.

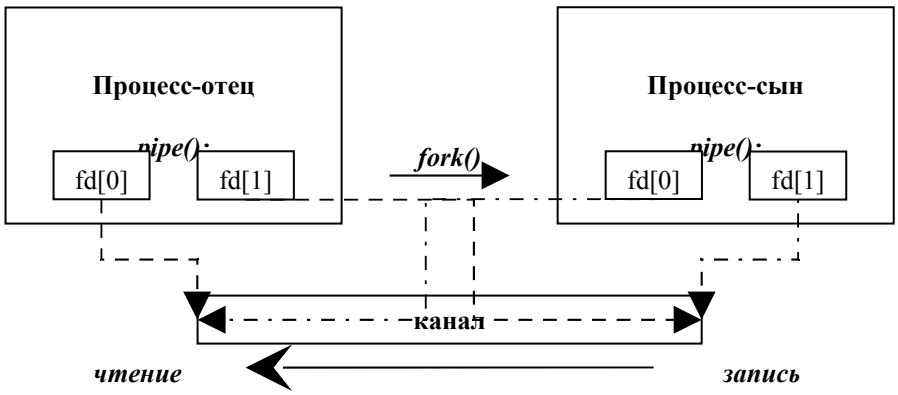


Рис. 88. Схема взаимодействия процессов с использованием неименованного канала.

Пример. Схема организации взаимодействия процессов с использованием канала (3.1.2)**.** Схема всегда такова: некоторый родительский процесс внутри себя порождает канал, после этого идут обращения к системным вызовам *fork()* — создается дерево процессов, но за счет того, что при порождении процесса открытые файловые дескрипторы наследуются, дочерний 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 дочерним процессами, причем процесс-отец будет писать в канал, а процесс-сын — читать из него.

```
int main(int argc, char **argv)

{

     int fd[2];


     pipe(fd);

     if(fork())

     {

          close(fd[0]);
```

```
            write(fd[1],...);
            ...
            close(fd[1]);
            ...
      }
      else
      {
            close(fd[1]);
            while(read(fd[0], ...))
            {
                  ...
            }
            ...
      }
}
```

В рассмотренном примере после создания канала посредством системного вызова *pipe()* и порождения дочернего 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 записи в канал в отцовском процессе можно не делать, т.к. при завершении процесса все открытые файловые дескрипторы будут автоматически закрыты. Но в дочернем процессе закрытие дескриптора записи в канал обязателен: в противном случае, поскольку дочерний 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 дочернего 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. В системе можно 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 предлагаемом примере реализуется конвейер команд **print|wc**, в котором команда **print** осуществляет печать некоторого текста, а команда **wc** выводит некоторые статистические характеристики входного потока (количество байт, строк и т.п.).

```
int main(int argc, char **argv)
{
     int fd[2];
     pipe(fd); /* организовали канал */
     if(fork())
     {
          /* ПРОЦЕСС-РОДИТЕЛЬ */
          /* 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 print */
          execlp("print","print",0);
     }

     /* ПРОЦЕСС-ПОТОМОК */
     /*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 открывается канал, затем порождается дочерний 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 print, у которой выходная информация будет писаться в канал.

В дочернем 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). В данном примере рассматривается корректную организацию двунаправленной работы, когда каждый из взаимодействующих процессов могут и читать, и писать из канала.

Итак, пускай есть два процесса, которые через канал будут перекидывать «мячик»-счетчик, подсчитывающий количество своих бросков в канал, некоторое предопределенное число раз. Извещение процесса о получении управления (когда он может взять «мячик» из канала, увеличить его на 1 и снова бросить в канал) будет происходить на основе механизма 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 */
     signal(SIGUSR1, SigHndlr);

     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);
     }
     else
          kill(target_pid, SIGUSR1);
}
```

```
int main(int argc, char **argv)
{
     /* организация канала */
     pipe(fd);
     /* установка обработчика сигнала для обоих процессов*/
     signal(SIGUSR1, SigHndlr);

     cnt = 0;

     if(target_pid = fork())
     {
          /* Предку остается только ждать завершения
          потомка */
          wait(&status);
          printf("Parent is going to be terminated\n");
          close(fd[1]);
          close(fd[0]);
          return 0;
     }
     else
     {
          /* процесс-потомок узнает PID родителя */
          target_pid = getppid();
          /* потомок начинает пинг-понг */
          write(fd[1], &cnt, sizeof(int));
          kill(target_pid, SIGUSR1);
          for(;;); /* бесконечный цикл */
```

```
    }
}
```

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. Посмотрим на функцию обработчика сигнала SIGUSR. В ней проверяется, не превзошло ли значение cnt величины MAX_CNT. В этом случае 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 дочерний 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 дочернего процесса, который и инициализирует завершение. Он печатает сообщение о своем завершении, закрывает дескрипторы, ассоциированные с каналом, и завершается посредством системного вызова *exit()*. Если же указанное условие ложно, то в данный момент управление находится у отцовского процесса, который сразу же передает его дочернему 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 дочернего процесса. После этого отцовский процесс начинает ожидать завершения дочернего 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 дочернем 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.

3.1.3 Именованные каналы

Файловая система ОС Unix поддерживает некоторую совокупность файлов различных типов. Файловая система рассматривает каталоги как файлы специального типа *каталог*, обычные файлы, с которым мы имеем дело в нашей повседневной жизни, — как *регулярные* файлы, устройства, с которыми работает система, — как специальные *файлы устройств*. Точно так же файловая система ОС Unix поддерживает специальные файлы, которые называются ***FIFO-файлами*** (или ***именованными каналами***). Файлы этого типа очень схожи с обыкновенными файлами (в них можно писать и из них можно читать информацию), за исключением того факта,

что они организованы 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. Эти файлы хранятся на внешних запоминающих устройствах, поэтому возможно открыть этот файл, записать в него информацию, а через любой промежуток времени (в течение которого допустимы перезагрузки системы) прочитать записанную информацию.

Для создания файлов FIFO в различных реализациях используются разные системные вызовы, одним из которых может являться *mkfifo()*[1].

```
int mkfifo(char *pathname, mode_t mode);
```

Первым параметром является имя создаваемого файла, а второй параметр отвечает за флаги владения, права доступа и т.п.

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 неименованным каналам.

Пример. «Клиент-сервер». В нижеприведенном примере 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.

```
/* процесс-сервер */

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/file.h>


int main(int argc, char **argv)

{

     int fd;

     int pid;
```

[1] 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).

```
    mkfifo("fifo", FILE_MODE | 0666);
    fd = open("fifo", O_RDONLY | O_NONBLOCK);
    while(read(fd, &pid, sizeof(int)) == -1);
    printf("Server %d got message from %d !\n", getpid(), pid);
    close(fd);
    unlink("fifo");
}

/* процесс-клиент */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>

int main(int argc, char **argv)
{
    int fd;

    int pid = getpid();
    fd = open("fifo", O_RDWR);
    write(fd, &pid, sizeof(int));
    close(fd);
}
```

В рассмотренном примере процесс-сервер создает FIFO-файл с некоторыми опциями и правами доступа, разрешающими всем писать и читать из этого файла, затем подключается к созданному FIFO-файлу в режиме только чтения без блокировок. Затем сервер ожидает появления информации в канале. Получив ее, он печатает соответствующее сообщение, закрывает дескриптор, ассоциированный с данным FIFO-файлом, и уничтожает сам файл.

Клиентский процесс открывает FIFO-файл в режиме чтения-записи, пишет в канал свой идентификатор процесса, а затем закрывает файловый дескриптор, ассоциированный с данным FIFO-файлом.

3.1.4 Модель межпроцессного взаимодействия «главный–подчиненный»

Достаточно часто при организации многопроцессной работы необходимо наличие возможности, когда один процесс является главным по отношению к другому процессу. В частности, это необходимо для организации средств отладки, когда есть процесс-отладчик и отлаживаемый процесс. Для механизма отладки оказалось бы полезным, 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 обеспечения 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**.

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, 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); и т.д.

Ниже приводится список возможных значений параметра cmd.

- cmd = **PTRACE_TRACEME** — сыновний процесс вызывает в самом начале своей работы ptrace с таким кодом операции, позволяя тем самым трассировать себя.
- cmd = **PTRACE_PEEKDATA** — чтение слова из адресного пространства отлаживаемого процесса.
- cmd = **PTRACE_PEEKUSER** — чтение слова из контекста процесса (из пользовательской составляющей, содержащейся в <sys/user.h>).

- cmd = **PTRACE_POKEDATA** — запись данных в адресное пространство процесса-потомка.
- cmd = **PTRACE_POKEUSER** — запись данных в контекст трассируемого процесса.
- cmd = **PTRACE_GETREGS**, **PTRACE_GETFREGS** — чтение регистров общего назначения (в т.ч. с плавающей точкой) трассируемого процесса.
- cmd = **PTRACE_SETREGS**, **PTRACE_SETFREGS** — запись в регистры общего назначения (в т.ч. с плавающей точкой) трассируемого процесса.
- cmd = **PTRACE_CONT** — возобновление выполнения трассируемого процесса.
- cmd = **PTRACE_SYSCALL**, **PTRACE_SINGLESTEP** — возобновляется выполнение трассируемой программы, но снова останавливается после выполнения одной инструкции.
- cmd = **PTRACE_KILL** — завершение выполнения трассируемого процесса.

Рассмотрим типовую схему организации трассировки. Будем рассматривать взаимодействие родительского процесса-отладчика с подчиненным дочерним процессом (3.1.4).

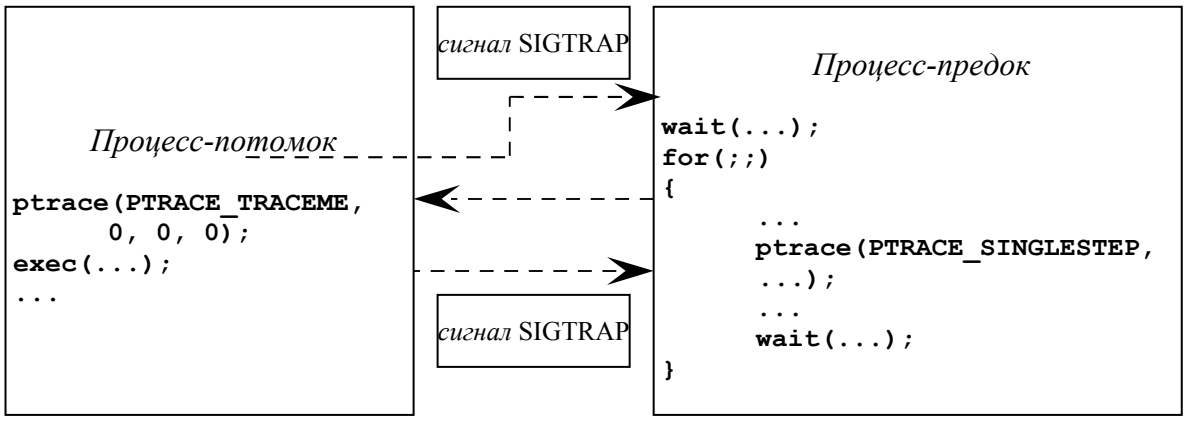


Рис. 89. Общая схема трассировки процессов.

Отцовский процесс формирует дочерний процесс и ожидает его завершения посредством обращения к системному вызову *wait()*. Дочерний 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 пошлет сигнал SIGTRAP родительскому процессу. 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 языка и операторами языка.

Механизм организации контрольной точки в адресно-кодовом отладчике достаточно простой. Пускай нам необходимо по некоторому адресу A установить контрольную точку, т.е. 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 отлаживаемого процесса, взятый по данному адресу. Для установки контрольной точки по адресу A 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 A (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 A в сегмент кода записывается команда, которая вызывает прерывание и,

соответственно, приход предопределенного события. 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 A. Происходит деление на ноль. Соответственно, происходит прерывание, система передает сигнал. И отладчик через системный вызов *wait()* получает код возврата и «понимает», что в дочернем процессе случилось деление на ноль. Отладчик посредством системного вызова *ptrace()* читает адрес остановки в контексте дочернего процесса. Далее анализируется причина остановки. Если причина остановки явилось деление на ноль, то возможны две ситуации: это действительно деление на ноль, как ошибка, либо 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 производится следующие действия. По адресу A записывается оригинальное машинное слово. После этого системным вызовом *ptrace()* включаем шаговый режим. И выполняем одну команду (которую только что записали по адресу A). 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 основании информации об этой точке программы можно, обратившись к содержимому базы данных, определить то пространство имен, доступных из этой точки. Если интересующая нас переменная v оказалась доступна, то продолжается работа: 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.

Пускай переменная v оказалась автоматической переменной или формальным параметром. Переменные этих типов обычно реализуются в вершине 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 на стек, после этого к содержимому регистра 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.

Для 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.

3.2 Система межпроцессного взаимодействия IPC (Inter-Process Communication)

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 трех типов.

- ***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).
- ***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 в ОС). Семафоры IPC предназначены, в первую очередь, для использования в качестве средств организации синхронизации.
- ***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.

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, предположим, общая память 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 (в частности, он 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 его атрибуты. Это означает, что если один процесс для генерации ключа ссылается на некоторый файл, создает ресурс, потом этот файл уничтожается, 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: право 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 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 в начале выше, а также некоторые флаги. Соответственно, эти функции в зависимости от флагов позволяют создавать или подключаться к существующему ресурсу IPC, ассоциированному с указываемым ключом.

```
<ResourceName>get(key, ..., flags);
```

Параметр флаги (flags) является один из важных параметров. Он может быть комбинацией различных флагов. Основные из них приведены ниже.

- **IPC_PRIVATE** — данный флаг определяет создание IPC-ресурса, 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 дочерним 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 к нему позволяют к нему обратиться, то процесс получит дескриптор ресурса и продолжит работу. Иначе функция *get* вернет -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, но главное отличие заключается в том, что сообщения в очереди сообщений IPC типизированы. Каждое сообщение помимо содержательной своей части имеет атрибут ***тип сообщения***. Тогда очередь сообщений можно рассматривать с двух позиций: во-первых, как сквозную очередь (когда тип сообщения не важен, они все находятся в единой очереди), а, во-вторых, как суперпозицию очередей однотипных сообщений (3.2.1). При этом способ интерпретации допускает одновременно различные типы интерпретации. Непосредственный выбор интерпретации определяется в момент считывания сообщения из очереди.

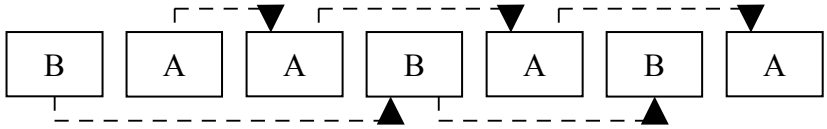


Рис. 90. Очередь сообщений IPC.

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.

Существует блок функций использования очереди: в частности, функции отправки и приема сообщения. Сначала рассмотрим функцию отправки сообщений.

```
#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 данной функции — 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 необходимо указывать структуру, содержащую следующие поля:

- **long msgtype** — тип сообщения (только положительное длинное целое);
- **char msgtext[]** — данные (тело сообщения).

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, как уже говорилось, из подочереди с заданным типом (или из общей очереди, если тип не задан) будет выбрано самое старое сообщение.

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 из 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. 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.

Пример. Использование очереди сообщений. В приведенном ниже примере участвуют три процесса: основной процесс и процессы A и B. Основной процесс считывает из стандартного 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 А*/
               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 А*/
               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);[2]
                    /* 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>
```

[2] 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.

```
#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);[3]
          printf("%s", Message.Data);
          if(Message.Data[0] == 'q' || Message.Data[0] == 'Q')
               break;
     }
     return 0;
```

[3] 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, 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.

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. 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.

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. Чаще всего, однако, в качестве значения этого аргумента передается 0, что 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), 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.

Возможные значения аргумента cmd:

- **IPC_STAT** — скопировать структуру, описывающую управляющие параметры области памяти;
- **IPC_SET** — заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре buf;
- **IPC_RMID** — удалить ресурс;
- **SHM_LOCK**, **SHM_UNLOCK** — блокировать или разблокировать область памяти. Это единственные средства синхронизации в данном ресурсе, их реализация должна поддерживаться аппаратурой.

Пример. Работа с общей памятью в рамках одного процесса. В данном примере процесс создает ресурс разделяемая память, размером в 100 байт (и 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.

```
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 используются для организации синхронизации взаимодействующих процессов. Рассмотрение функций для работы с семафорами мы начнем традиционно с функции создания/доступа к данному ресурсу — функции *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, второй — количество семафоров (длина массива семафоров), и третий параметр — флаги. Через флаги можно определить права доступа и те операции, которые должны выполняться (открытие семафора, проверка, и т.д.). Функция *semget()* возвращает целочисленный идентификатор созданного разделяемого ресурса, либо -1 в случае ошибки. Необходимо отметить, что если процесс подключается к существующему ресурсу, то возможно появление коллизий, связанных с неверным указанием длины массива семафоров.

Основная функция для работы с семафорами — функция *semop()*.

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

#include <sys/ipc.h>

#include <sys/sem.h>


int semop(int semid, struct sembuf *semop, size_t nops);
```

Первый аргумент — идентификатор ресурса, второй аргумент является указателем на начало массива структур, определяющих операции, которые необходимо произвести над семафорами. Третий параметр — количество структур в указанном массиве. Каждый элемент данного массива — это структура определенного вида, предназначенная для выполнения операции над соответствующим семафором в массиве семафоров. Ниже приводится указанная структура.

```
struct sembuf

{

     short sem_num; /* номер семафора в массиве */

     short sem_op;  /* код производимой операции */

     short sem_flg; /* флаги операции */

}
```

Поле операции интерпретируется следующим образом. Пускай значение семафора с номером **num** равно **val**. Тогда порядок работы с семафором можно записать в виде следующей схемы.

```
Если sem_op = 0 то

          если val ≠ 0 то

                    пока (val ≠ 0) [процесс блокирован]

[возврат из вызова]
```

```
Если sem_op ≠ 0 то
        если val + sem_op < 0 то
                пока (val + sem_op < 0) [процесс блокирован]
        val = val + sem_op
```

Понимать данную последовательность действий надо так. Если код операции равен нулю, а значение данного семафора не равно нулю, то процесс будет блокирован до тех пор, пока значение семафора не обнулится. Если же и код операции нулевой, и значение семафора нулевое, то никаких блокировок не произойдет, и операция завершится. Если код операции отличен от нуля (т.е. процесс желает скорректировать значение семафора), то в этом случае делается следующая проверка. Если сумма текущего значения семафора и кода операции отрицательная, то процесс будет блокирован. Как только он разблокируется, происходит коррекция. Замети, что если указанная сумма значения семафора и кода операции неотрицательная, то коррекция происходит без блокировок.

Для управления данным типом разделяемых ресурсов используется системный вызов *semctl()*.

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

#include <sys/ipc.h>

#include <sys/sem.h>


int semctl(int semid, int num, int cmd, union semun arg);
```

Параметрами данной функции являются, соответственно, дескриптор массива семафоров, индекс семафора в массиве, команда и управляющие параметры. Среди множества команд, которые можно выполнять с помощью данной функции, можно особо отметить две: команду удаления ресурса (IPC_RMID) и команду инициализации и модификации значения семафора (IPC_SET). Используя последнюю команду, можно использовать массив семафоров уже не как средство синхронизации, а как средство передачи информации между взаимодействующими процессами (что само по себе является, как минимум, неэффективным, поскольку семафоры создавались именно как средства синхронизации).

Данная функция возвращает значение, соответствующее выполнявшейся операции (по умолчанию 0), или -1 в случае неудачи. Ниже приводится определение типа последнего параметра.

```
<sys/sem.h>


union semun

{

     int val;                    /* значение одного семафора */
```

```
    struct semid_ds *buf;    /* параметры массива семафоров в
                             целом (количество, права доступа,
                             статистика доступа)*/
    ushort *array;           /* массив значений семафоров */
}
```

Пример. Использование разделяемой памяти и семафоров. 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. 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).

```
/* 1-ый 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');
    /* создаем один семафор с определенными правами доступа */
```

```
semid = semget(key, 1, 0666 | IPC_CREAT | IPC_EXCL);
/*создаем разделяемую память на 256 элементов */
shmid = shmget(key, NMAX, 0666 | IPC_CREAT | IPC_EXCL);
/* подключаемся к разделу памяти, в shmaddr – 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;
/* запуск цикла */
do
{
     printf("Введите строку:");
     if(fgets(str, NMAX, stdin) == NULL)
          /* пишем признак завершения – строку "Q" */
          strcpy(str, "Q");
     /* в текущий момент семафор открыт для этого процесса*/
     /* копируем строку в разд. память */
     strcpy(shmaddr, str);
     /* 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 */
    /* уничтожаем разделяемую память */
    shmctl(shmid, IPC_RMID, NULL);
    /* уничтожаем семафор */
    semctl(semid, 0, IPC_RMID, (int) 0);
    return 0;
}

/* 2-ой 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 тот же самый ключ */
    key = ftok("/usr/ter/exmpl", 'S');
```

```
semid = semget(key, 1, 0666);
shmid = shmget(key, NMAX, 0666);
/* аналогично предыдущему процессу –
инициализация ресурсов */
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 неотрицательным, т.е. пока значение
     семафора не станет, как минимум, 2 (2 - 2 >= 0) */
     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 */
     /* копируем строку из разд. памяти */
     strcpy(str, shmaddr);
     if(str[0] == 'Q')
          /* завершение работы - освобождаем разд. память*/
          shmdt(shmaddr);
     /* после работы – обнулим семафор */
     sops.sem_op = -1;
```

```
            semop(semid, &sops, 1);

            printf("Read from shared memory: %s\n", str);

        } while (str[0] != 'Q');

        return 0;

}
```

3.3 Сокеты — унифицированный интерфейс программирования распределенных систем

Средства межпроцессного взаимодействия ОС UNIX, представленные в системе IPC, решают проблему взаимодействия двух 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. При этом механизм должен быть унифицированным: он должен в определенной степени позволять абстрагироваться от расположения процессов и давать 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 Берклиевском UNIX — BSD, начиная с версии 4.2, и названный ***сокетами*** (***sockets***). Ниже подробно рассматривается этот механизм.

Механизм сокетов обеспечивает два типа соединений. Во-первых, это соединение, обеспечивающее установление виртуального канала (т.е. обеспечиваются соответствующие свойства, в частности, гарантируется порядок передачи сообщения), его прямым аналогом является протокол TCP. Во-вторых, это дейтаграммное соединение (соответственно, без обеспечения порядка передачи и т.п.), аналогом которого является протокол UDP.

Именование сокетов для организации работы с ними определяется т.н. ***коммуникационным доменом***. Аппарат сокетов в общем случае поддерживает целый спектр коммуникационных доменов, среди которых нас будут интересовать два из них: домен AF_UNIX (семейство имен в ОС Unix) и AF_INET (семейство имен для сети Internet, определяемое стеком протоколов TCP/IP).

Для создания сокета используется системный вызов *socket()*.

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

#include <sys/socket.h>


int socket(int domain, int type, int protocol);
```

Первый параметр данного системного вызова определяет код коммуникационного домена. Коммуникационный домен, в свою очередь, определяет структуру именования, которая может

быть использована для сокета. Как говорилось выше, на сегодняшний день существует целый ряд доменов, мы же остановимся на доменах, обозначаемых константами AF_UNIX и AF_INET.

Второй параметр отвечает за тип сокета: либо SOCK_STREAM (тип виртуальный канал), либо SOCK_DGRAM (дейтаграммный тип сокета).

Последний параметр вызова — протокол. Выбор значения данного параметра зависит от многих факторов — и в первую очередь, от выбора коммуникационного домена и от выбора типа сокета. Если указывается нулевое значение этого параметра, то система автоматически выберет протокол, учитывая значения первых аргументов вызова. А можно указать константу, связанную с именем конкретного протокола: например, IPPROTO_TCP для протокола TCP (домена AF_INET) или IPPROTO_UDP для протокола UDP (домена AF_INET). Но в последнем случае необходимо учесть, что могут возникать ошибочные ситуации. Например, если явно выбран домен AF_INET, тип сокета виртуальный канал и протокол UDP, то возникнет ошибка. Однако, если домен будет тем же, тип сокета дейтаграммный и протокол TCP, то ошибки не будет: просто дейтаграммное соединение будет реализовано на выбранном протоколе.

В случае успешного завершения системный вызов *socket()* возвращает открытый файловый дескриптор, ассоциированный с созданным сокетом. Как отмечалось выше, сокеты представляют собой особый вид файлов в файловой системе ОС Unix. Но данный дескриптор является локальным атрибутом: это лишь номер строки в таблице открытых файлов текущего процесса, в которой появилась информация об этом открытом файле. И им нельзя воспользоваться другим процессам, чтобы организовать взаимодействие с текущим процессом посредством данного сокета. Необходимо связать с этим сокетом некоторое имя, доступное другим процессам, посредством которого они смогут осуществлять взаимодействие. Для организации именования используется системный вызов *bind()*.

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

#include <sys/socket.h>


int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

Посредством данного системного вызова возможно связывание файлового дескриптора (первого параметра), ассоциированного с открытым сокетом, с некоторой структурой, в которой будет размещаться имя (или адрес) данного сокета. Для разных коммуникационных доменов структура данного имени различна. Например, для домена AF_UNIX она выглядит следующим образом:

```
#include <sys/un.h>


struct sockaddr_un

{

     short sun_family; /* == AF_UNIX */

     char sun_path[108];

};
```

Первое поле в данной структуре — это код коммуникационного домена, а второе поле — это полное имя. Для домена AF_INET структура выглядит несколько иначе:

```
#include <netinet/in.h>


struct sockaddr_in

{

     short sin_family;        /* == AF_INET */

     u_short sin_port;        /* номер порта */

     struct in_addr sin_addr; /* IP-адрес хоста */

     char sin_zero[8];        /* не используется */

};
```

В данной структуре присутствует различная информация, в частности, IP-адрес, номер порта и т.п.

И, наконец, последний аргумент addrlen рассматриваемого системного вызова характеризует размер структуры второго аргумента (т.е. размер структуры sockaddr).

В случае успешного завершения данный вызов возвращает значение 0, иначе — -1.

Механизм сокетов включает в свой состав достаточно разнообразные средства, позволяющие организовывать взаимодействие различной топологии. В частности, имеется возможность организации взаимодействия с т.н. ***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 (3.3). Заметим, что тип сокета в данном случае не важен, т.е. можно использовать сокеты, являющиеся виртуальными каналами, так и дейтаграммными.

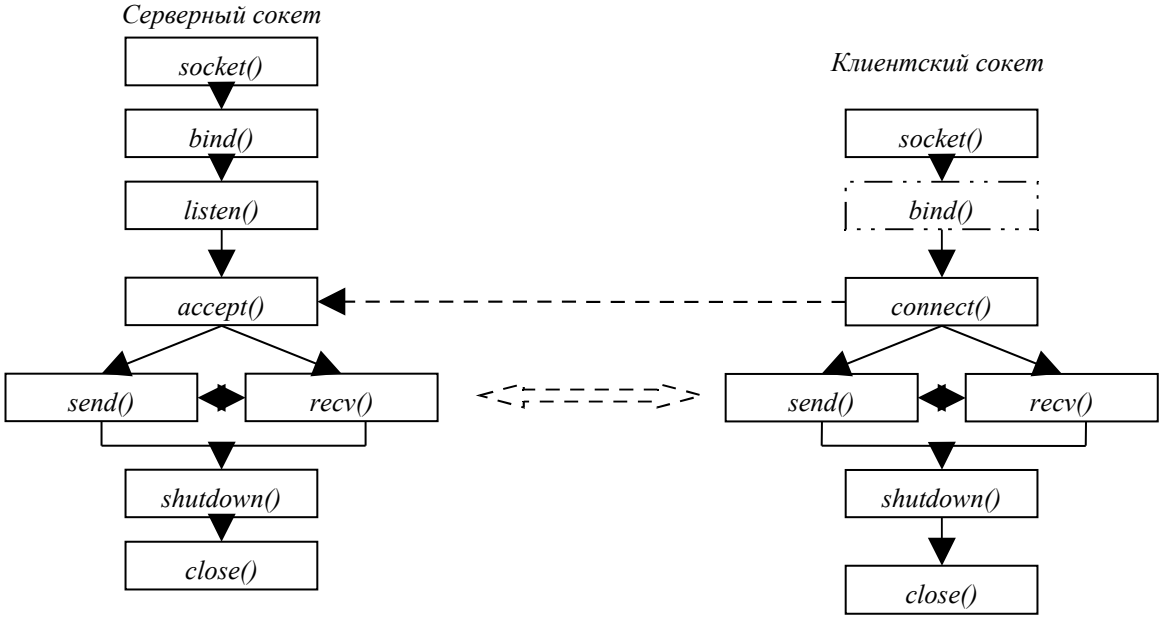


Рис. 91. Схема работы с сокетами с предварительным установлением соединения.

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 т.н. ***режим прослушивания*** (посредством системного вызова *listen()*): это означает, что данный процесс может принимать запросы на соединение с ним от клиентских процессов. При этом, в вызове *listen()* оговаривается очередь запросов на соединение, которая может формироваться к данному процессу серверной части.

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 три 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 будет ожидать обслуживания.

И, наконец, в-третьих, может оказаться, что указанная очередь переполнена. В этом случае клиент получает отказ с соответствующим уведомлением, что сервер в данный момент занят.

Итак, данная схема организована таким образом, что к одному серверному узлу может иметь 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 дочерний процесс для каждого вызова *accept()*, и тогда все действия по работе с данным клиентом ложатся на этот дочерний 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. 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).

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()*.

Далее будут более подробно рассмотрены упомянутые системные вызовы для работы с сокетами.

Для посылки запроса на соединение используется системный вызов *connect()*.

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

#include <sys/socket.h>


int connect(int sockfd, struct sockaddr *serv_addr,

                                                int addrlen);
```

Первый параметр функции — дескриптор «своего» сокета, через который будет посылаться запрос на соединение. Второй параметр — указатель на структуру, содержащую адрес сокета, с которым производится соединение, в формате, который обсуждался выше. Третий параметр — длина структуры, передающейся вызову во втором аргументе.

В случае успешного завершения вызов возвращает значение 0, иначе возвращается -1, а в переменную *errno* заносится код ошибки.

Для перехода серверного процесса в режим прослушивания сокета используется системный вызов *listen()*.

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

#include <sys/socket.h>


int listen(int sockfd, int backlog);
```

Параметры вызова — дескриптор сокета и максимальный размер очереди необработанных запросов на соединение. В случае успешного завершения вызов возвращает значение 0, иначе возвращается -1, а в переменную *errno* заносится код ошибки.

Для подтверждения соединения используется системный вызов *accept()*. Этот вызов ожидает появление запроса на соединение (в очереди необработанных запросов), при появлении последнего формируется новый сокет, и его дескриптор возвращается в качестве значения функции.

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

#include <sys/socket.h>


int accept (int sockfd, struct sockaddr *addr,

                                          int *addrlen);
```

Первый параметр данной функции — дескриптор «главного» сокета. Второй параметр — указатель на структуру, в которой возвращается адрес клиентского сокета, с которым установлено

соединение (если адрес клиента не интересует, передается NULL). В последнем параметре возвращается реальная длина упомянутой структуры.

Для модели с предварительным установлением соединения можно использовать системные вызовы чтения и записи в файл — соответственно, *read()* и *write()* (в качестве параметра этим функциям передается дескриптор сокета), а также системные вызовы *send()* (посылка сообщения) и *recv()* (прием сообщения).

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

#include <sys/socket.h>


int send(int sockfd, const void *msg, int msglen,

                                        unsigned int flags);


int recv(int sockfd, void *buf, int buflen, unsigned int flags);
```

Параметры: sockfd — дескриптор сокета, через который передаются данные; msg — указатель на начало сообщения; msglen — длина посылаемого сообщения; buf — указатель на буфер для приема сообщения; buflen — первоначальный размер буфера; и, наконец, параметр flags может содержать комбинацию специальных опций. Например:

- **MSG_OOB** — флаг сообщает ОС, что процесс хочет осуществить прием/передачу экстренных сообщений;
- **MSG_PEEK** — при вызове *recv()* процесс может прочитать порцию данных, не удаляя ее из сокета. Последующий вызов *recv()* вновь вернет те же самые данные.

Для организации приема и передачи сообщений в модели ***без предварительного установления соединения*** (3.3) используется пара системных вызовов *sendto()* и *recvfrom()*.

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

#include <sys/socket.h>


int sendto(int sockfd, const void *msg,

                int msglen, unsigned int flags,

                const struct sockaddr *to, int tolen);


int recvfrom(int sockfd, void *buf,

                int buflen, unsigned int flags,

                struct sockaddr *from, int *fromlen);
```

Первые четыре параметра каждого из вызовов имеют ту же семантику, что и параметры вызовов *send()* и *recv()* соответственно. Остальные параметры имеют следующий смысл: to —

указатель на структуру, содержащую адрес получателя; tolen — размер структуры to; from — указатель на структуру с адресом отправителя; fromlen — размер структуры from.

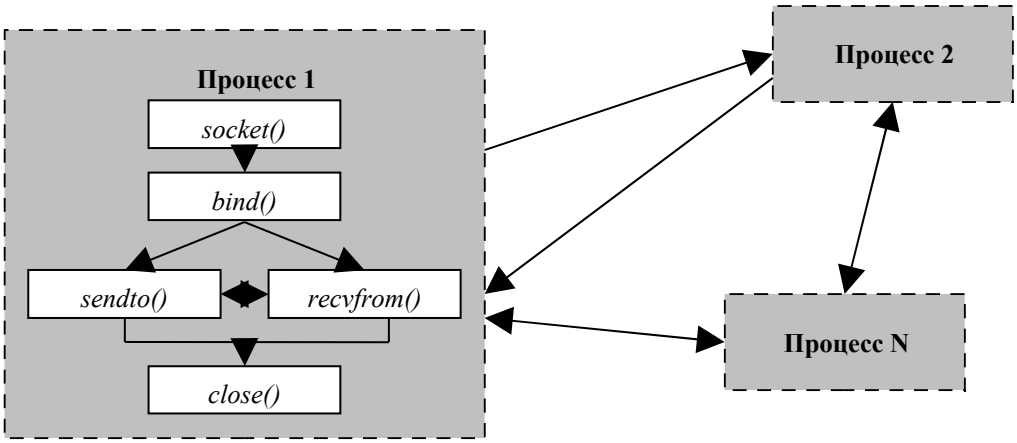


Рис. 92. Схема работы с сокетами без предварительного установления соединения.

Для завершения работы с сокетом используется системный вызов *shutdown()*.

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

#include <sys/socket.h>


int shutdown (int sockfd, int mode);
```

Первый параметр — дескриптор сокета, второй — режим закрытия соединения:

- **0** — сокет закрывается для чтения;
- **1** — сокет закрывается для записи;
- **2** — сокет закрывается и для чтения, и для записи.

Для закрытия сокета используется системный вызов *close()*, в котором в качестве параметра передается дескриптор сокета.

4 Файловые системы

4.1 Основные концепции

Под ***файловой системой*** (ФС) мы 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.

Файловая система является с точки зрения пользователя первым виртуальным ресурсом (который появился в операционных системах), достаточно понятным и достаточно просто используемым во время его работы за машиной. 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 необходима 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 должен был учитывать пользователь 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 ним 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 словами 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 от несанкционированного доступа к информации. 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 на определенные операции: 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 который 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 какой-то 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.

Во-первых, это **начало работы с файлом** (или **открытие файла**). Большинство систем для процессов, желающих работать с файлом, предполагают наличие 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***, в котором отражается актуальное состояние 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 системы 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 корректное завершение работы с файлом: в частности, по необходимости освобождаются системные ресурсы, в т.ч. 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** — открытые/создание файла;
- **close** — закрытие;
- **read**/**write** — читать/писать (относительно указателя чтения/записи соответственно);
- **delete** — удалить файл 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, возможны 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 файла.

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 (4.1.3) присутствует единственный каталог, в котором перечислены всевозможные имена файлов, находящихся в данной системе. Этот каталог устроен простым способом: о каждом файле 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.

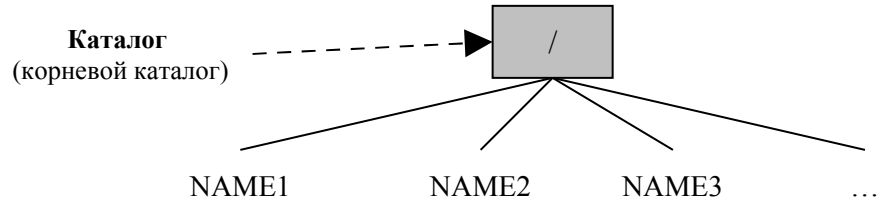


Рис. 93. Модель одноуровневой файловой системы.

Следующей моделью является **двухуровневая файловая система** (4.1.3). 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.

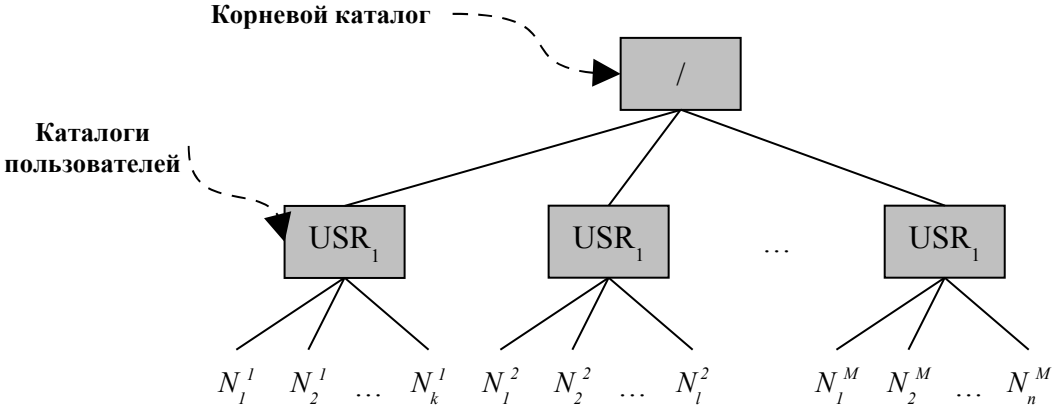


Рис. 94. Модель двухуровневой файловой системы.

И, наконец, последняя модель, которую мы рассмотрим, — **иерархическая файловая система** (4.1.3). Современные многопользовательские файловые системы основываются на использовании иерархических структур данных, в частности, на использовании деревьев.

Вся информация в 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.

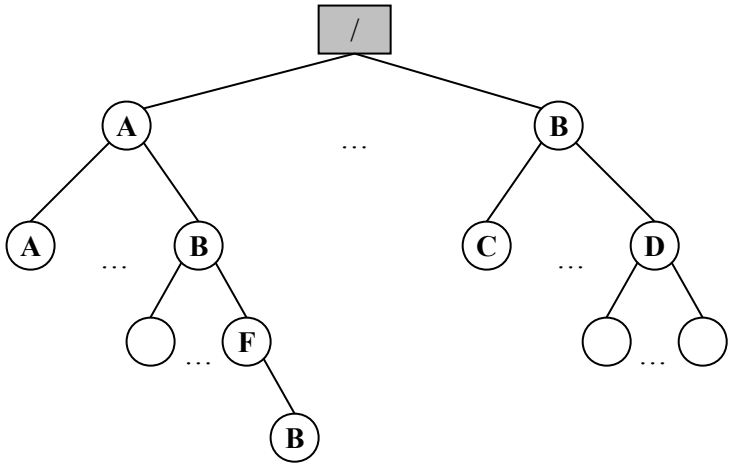


Рис. 95. Модель иерархической файловой системы.

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, находящего в текущем каталоге, 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/F/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 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, поскольку в дисководе присутствует дискета, то считается, что дисковод готов к работе. Тогда происходит попытка загрузки операционной системы с дисковода, и если это будет неуспешно, то будет выведено сообщение об ошибке загрузки системы. Если же дискеты во флопе-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. 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.

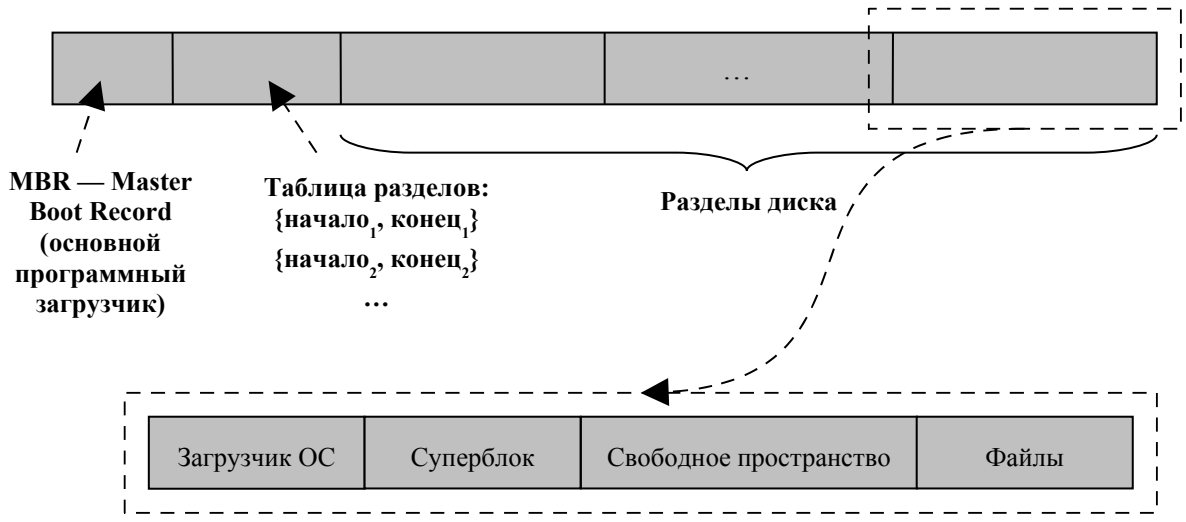


Рис. 96. Структура «системного» диска.

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 лежит 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** (4.1.5). 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.

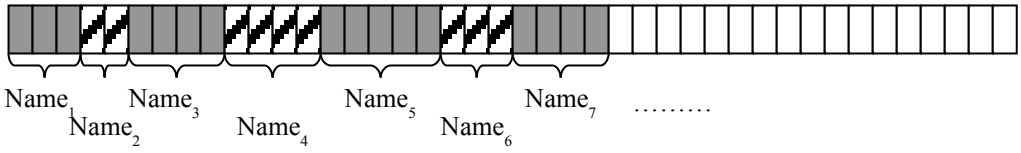


Рис. 97. Модель непрерывных файлов.

Следующей моделью является **модель файлов, имеющих организацию связанного списка** (4.1.5). 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. 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.

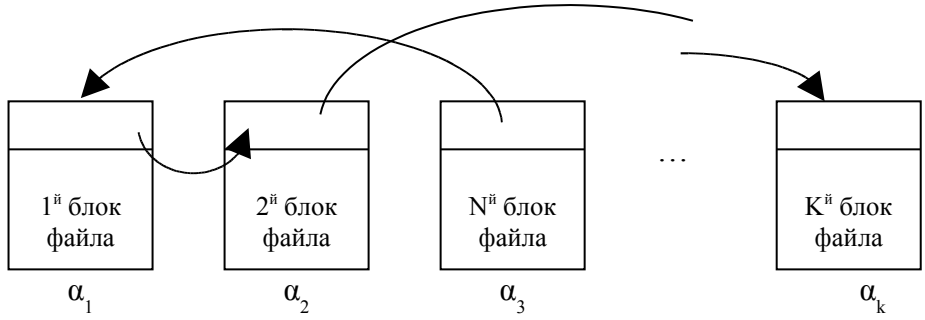


Рис. 98. Модель файлов, имеющих организацию связанного списка.

Другой подход иллюстрирует модель, основанная на использовании т.н. **таблицы размещения файлов** (***File Allocation Table — FAT***, 4.1.5). 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 имела 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-тигигабайтного 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.

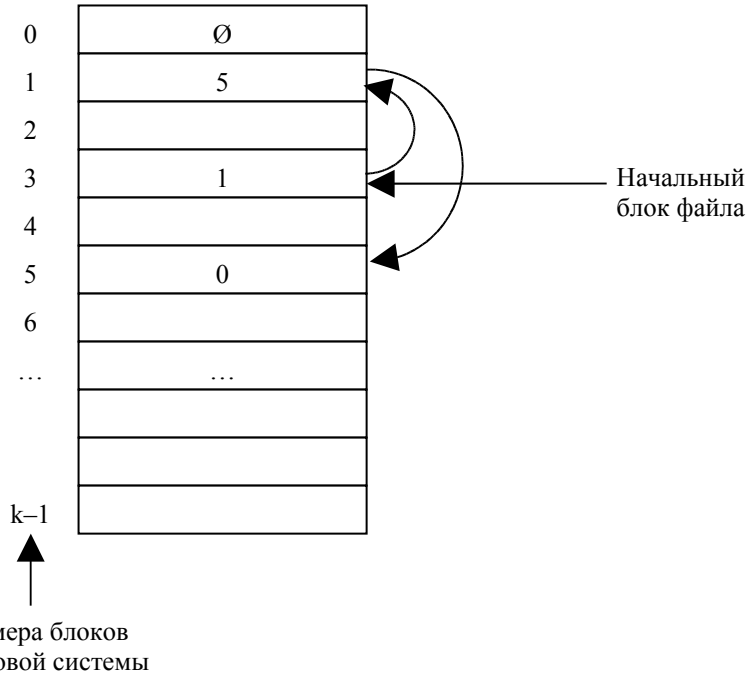


Рис. 99. Таблица размещения файлов (FAT).

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 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. 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 находятся в оперативной доступности, зато 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.

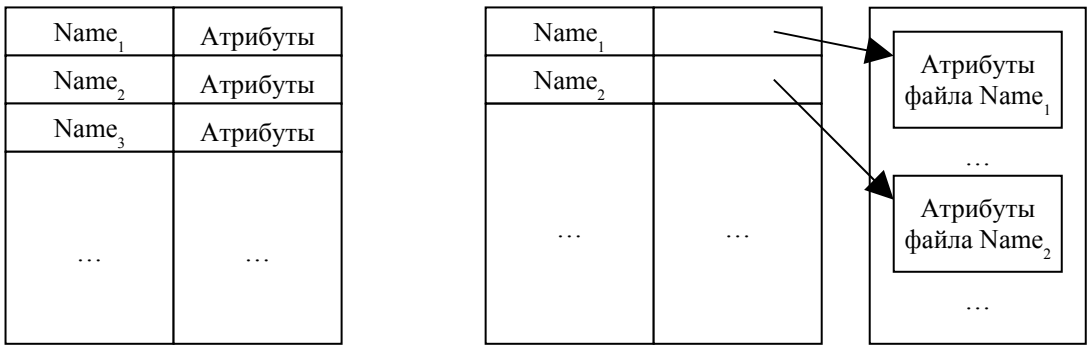


Рис. 100. Модели организации каталогов.

Одной из проблем, встречающейся в 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 его 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. 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, существуют 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 какое бы из имен, ассоциированных с данным содержимым, не взяли, мы посредством этого имени можем выполнить все операции, и они будут выполнены одинаково. Такая модель реализована в некоторых файловых системах посредством установления т.н. ***жесткой связи*** (4.1.7). 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.

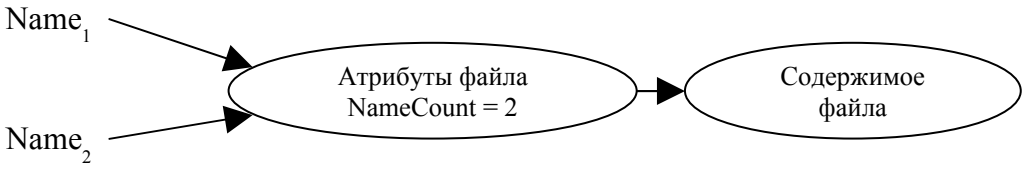


Рис. 101. Пример жесткой связи.

Следующей модель организации — это «мягкая» ссылка, или ***символическая связь*** (хотя здесь лучше бы подошло название *символьной* связи, 4.1.7). Данное именование является несимметричным, суть которого заключается в следующем. Пускай 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 файла-ссылки), 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.

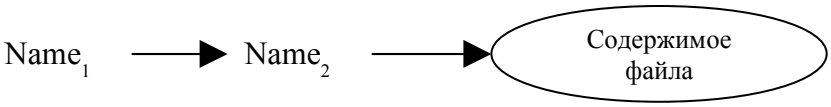


Рис. 102. Пример символической связи.

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, неэффективность использования пространства ВЗУ (если, блок, к примеру, имеет размер 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 них 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 размер списка может достигать больших размеров: если размер блока 1 Кбайт, т.е. его можно представить в виде 256 четырехбайтных слов, то такой блок может содержать в себе 255 номеров свободных блоков и одну ссылку на следующий блок со списком, тогда для жесткого диска, емкостью 16 Гбайт, потребуется 16794 блока. 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. Заметим, что операция пересчета достаточно трудоемка, к тому же эта модель требует выделение под массив стационарного ресурса: так, для 16-тигигабайтного жесткого диска потребуется 2048 блоков для хранения битового массива.

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.

В общем случае модель квотирования может иметь два типа лимитов: ***жесткий*** и ***гибкий***. Для каждого пользователя при регистрации его в системе для него определяются два типа квот. ***Жесткий лимит*** — это количество имен в каталогах или блоков файловой системы, которое он 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***. 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 равно нулю, то вход пользователя в систему блокируется, иначе 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.

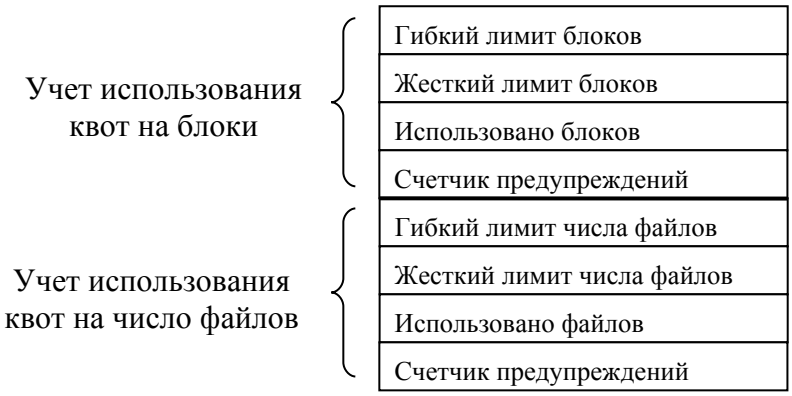


Рис. 103. Квотирование пространства файловой системы.

Рассмотренная модель имеет большую эффективность при использовании именно пары этих параметров. Если в системе реализовано 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 исполнительные 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 определенных категорий пользователей (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 данный 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 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 (4.1.11).

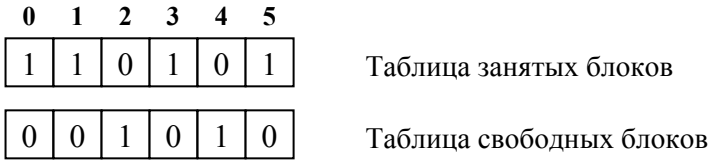


Рис. 104. Проверка целостности файловой системы. Непротиворечивость файловой системы соблюдена.

Если же находится блок, о котором нет информации ни в таблице свободных, ни в таблице занятых блоков (т.е. и в соответствующих ячейка стоят нули), то считается, что этот блок потерян из списка свободных блоков (4.1.11). 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.

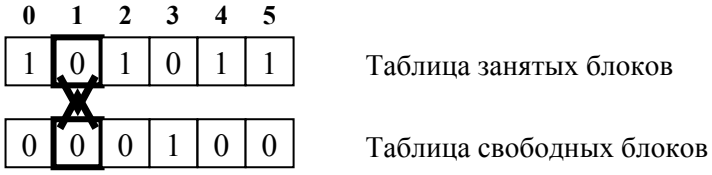


Рис. 105. Проверка целостности файловой системы. Зафиксирована пропажа блока.

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 (4.1.11).

Рис. 106. 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 (4.1.11). 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. Пускай 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 переопределение списка свободных блоков и, наконец, обратно переименовывает эти копии с фиксацией факта их возможной некорректности.

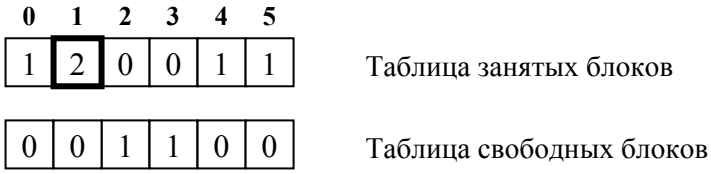
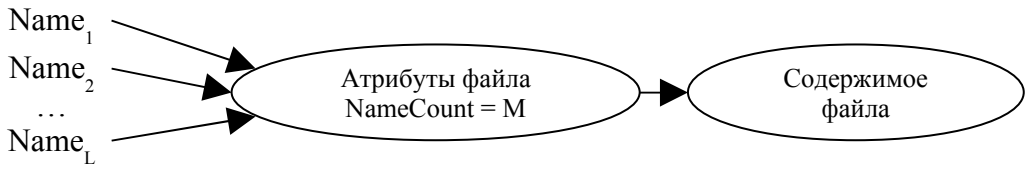


Рис. 107. Проверка целостности файловой системы. Зафиксировано дублирование занятого блока.

И, наконец, для проверки корректности файловой системы может выполняться проверка соответствия числа реального количества жестких связей тому значению, которое хранится среди атрибутов файла (4.1.11). 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. 108. 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 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 которой 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 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 который 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 его 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. 3.1.3);
- ***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, без этого владельца, и, наконец, права всех оставшихся пользователей системы (без указанной группы владельца). Соответственно, для каждой из категорий определяются вышеперечисленные права доступа.

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 (4.2.2), т.е. у нее есть корневой каталог /, из которого за счет каталогов разных уровней вложенности «вырастает» целое дерево **имен** файлов.

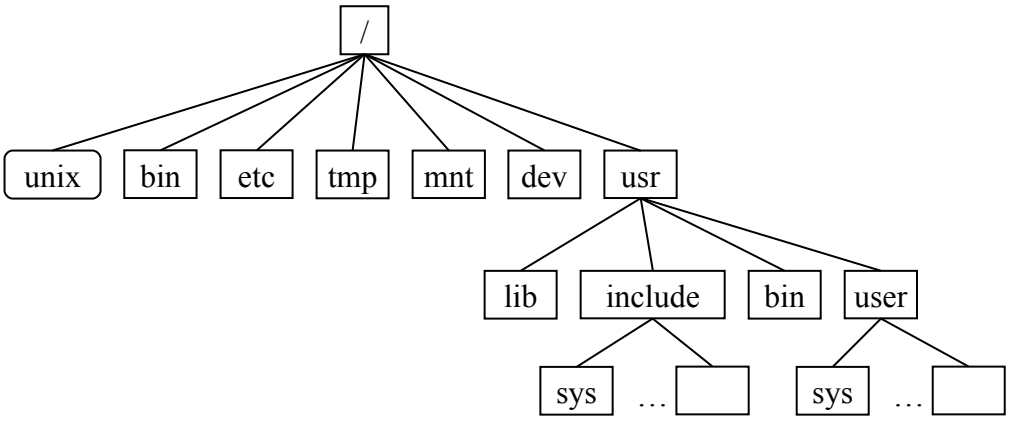


Рис. 109. Логическая структура каталогов.

Система предполагает, что в корневом каталоге всегда расположен некоторый файл, в котором размещается код ядра операционной системы. Сразу оговоримся, что мы рассматриваем некоторую модельную системы: 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 пользователей.

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 один 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**.

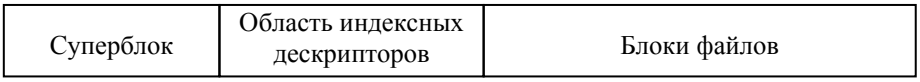


Рис. 110. Структура файловой системы версии System V.

Данная файловая система имеет следующую структуру (4.2.3). Файловая система занимает часть того раздела, в котором она находится (назовем его ***системным разделом***, чтобы отличать его от разделов с другими файловыми системами, имеющими схожую организацию, и 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***. 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 индексный 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 — наличие 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 (4.2.3.1), который размещается в нескольких блоках. Первый блок располагается в суперблоке (а значит, в оперативной памяти). Каждый блок хранит номера свободных блоков, а также номер следующего блока данного массива.

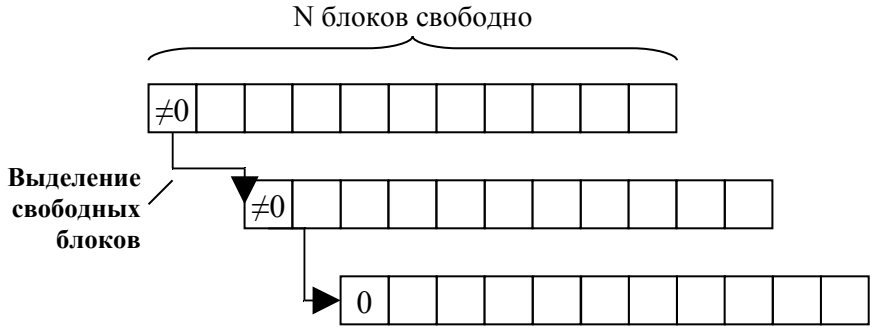


Рис. 111. Работа с массивами номеров свободных блоков.

Работа с массивом номеров свободных блоков достаточно проста. При запросе на получение свободного блока происходит поиск в первом блоке массива ячейки с содержательной (ненулевой) информацией, обнуление найденной ячейки, а блок с найденным номером выдается в ответ на запрос. Если же происходит обнуление последней ячейки блока, ссылающейся на следующий блок массива, то предварительно содержимое этого блока загружается в суперблок и используется уже как первый блок этого массива. Если же какой-то блок освобождается, то выполняются противоположные действия в обратном порядке.

На первый взгляд может показаться, что хранение в блоках массива свободных блоков уменьшают 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 — это массив 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 дескрипторах, 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*** (4.2.3.3) 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.

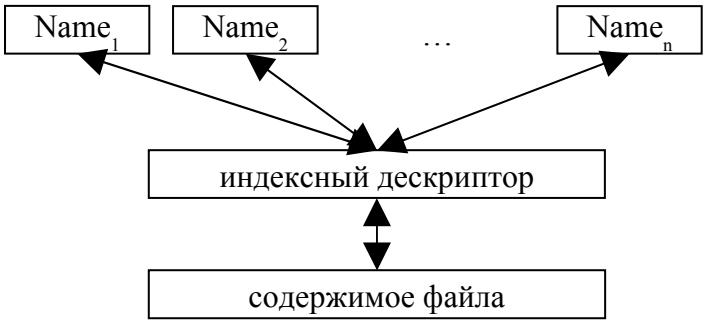


Рис. 112. Индексные дескрипторы.

Для каждого индексного дескриптора существует, по меньшей мере, одно имя, зарегистрированное в каталогах файловой системы. И еще раз повторимся, сказав, что, говоря о древовидности файловой системы, то понимают древовидности не с точки зрения размещения файла, а с точки зрения размещения имен файлов.

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 блоков файла.

Организация блоков файла — еще одна удачная особенность файловой системы ОС Unix. Структура организации блоков файла выглядит следующим образом. Массив 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 блоков файла, тринадцатый элемент используется уже для трехуровневой косвенной адресации блоков.

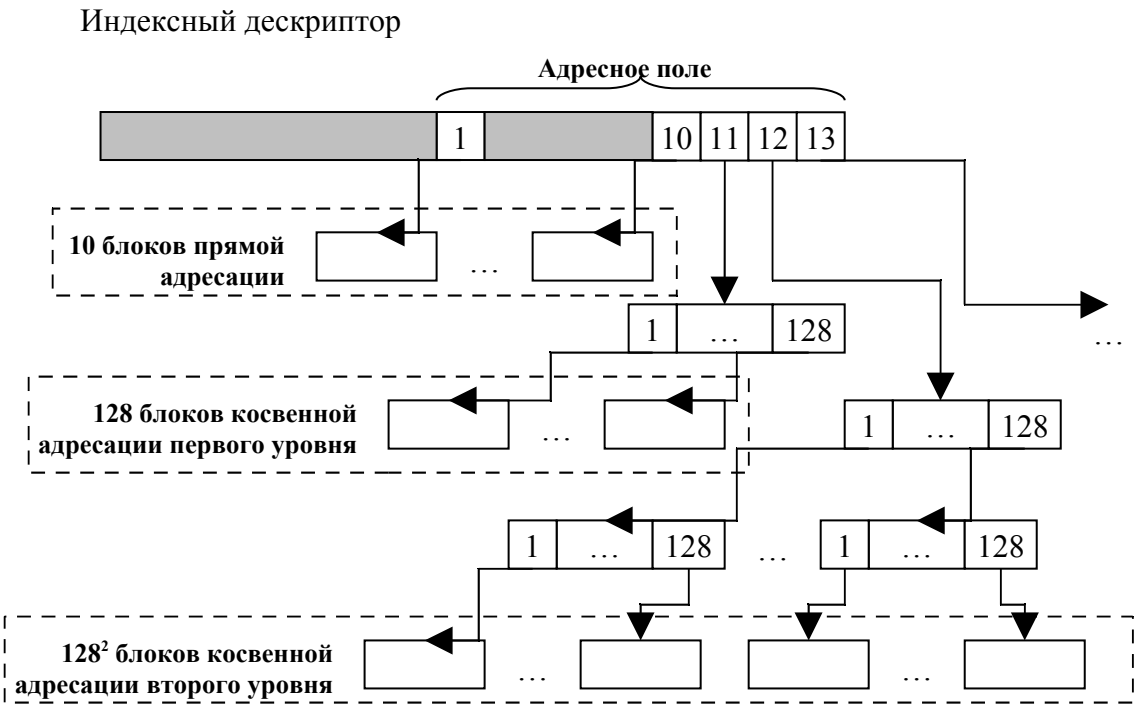


Рис. 113. Адресация блоков файла.

Рассмотрим **пример** системы (4.2.3.3), в которой размер блока равен 512 байтам (т.е. 128 четырехбайтовых чисел). 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, состоящем в нашем случае из 128 номеров блоков файла, которые следуют за первыми десятью блоками. Иным 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 (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, 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 (4.2.3.4) имеют следующую структурную организацию. Каждая запись в ней нем имеет фиксированный размер: длина записи довольно сильно варьировалась, мы будем считать, что длина записи 16 байт. Первые два байта хранят номер индексного дескриптора файла,

а оставшиеся 14 байтов — это 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.

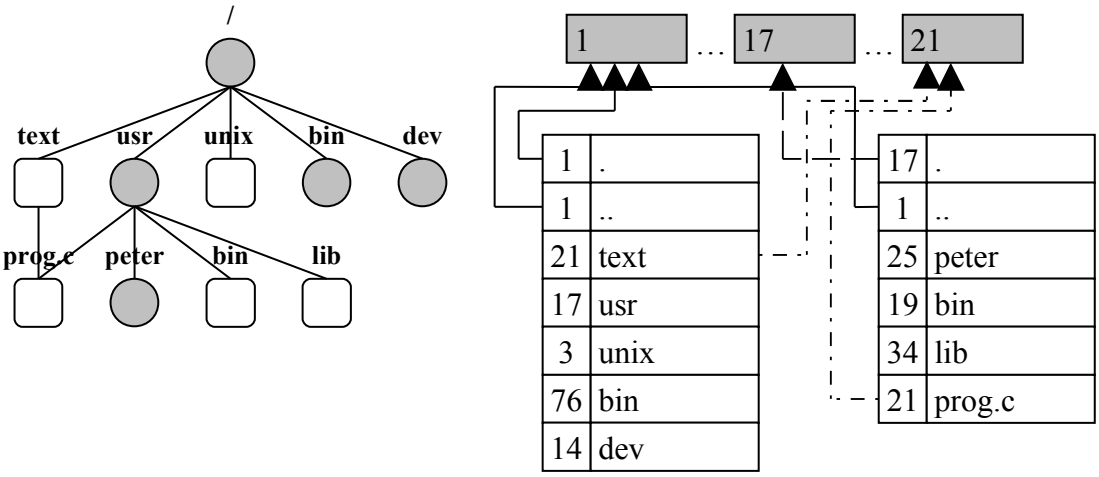


Рис. 114. Файл-каталог.

Отвлекаясь от файловой системы версии System V, отметим, что многие более развитые файловые системы ОС Unix поддерживают средства **установления связей** (4.2.3.4) 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, то первым делом 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 файла специального типа — типа ссылки. 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 (права файла ссылки будут отличаться от прав файла, на который он ссылается).

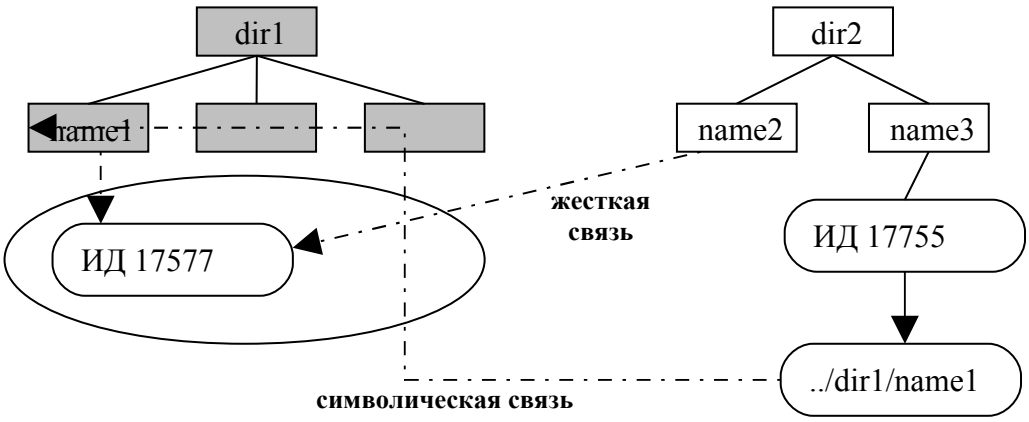


Рис. 115. Установление связей.

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 следует из 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 достаточно небольшие на сегодняшний день: возникают ситуации, когда необходимо создать имена с относительно длинными именами).

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 (4.2.4). 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.

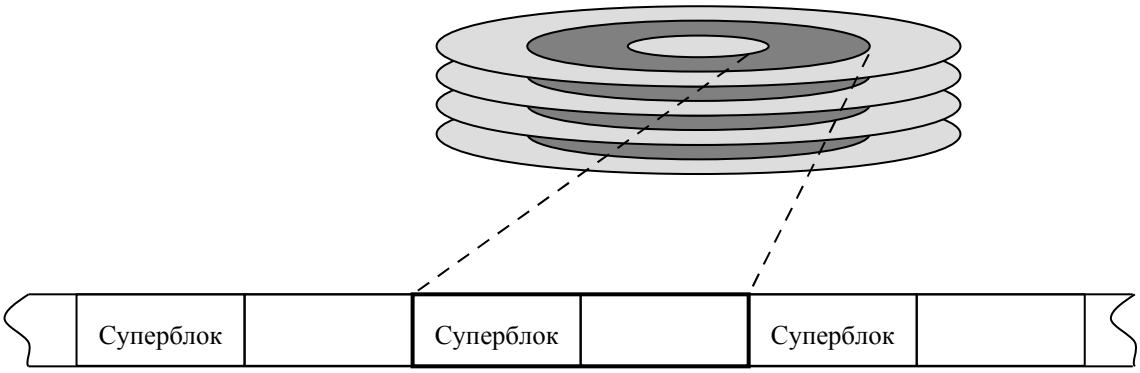


Рис. 116. Структура файловой системы версии FFS BSD.

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 фрагментации файла по разделу: файл либо целиком размещается в одном кластере, либо он 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** (4.2.4.1). Представим следующую ситуацию: пускай 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 протяженным промежутком 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.

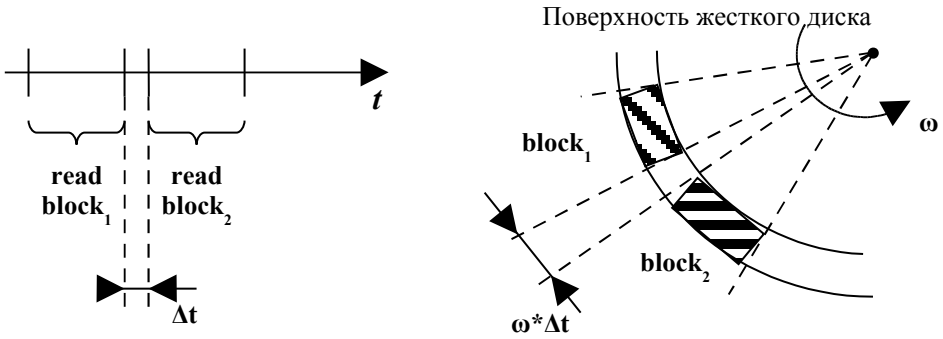


Рис. 117. Стратегия размещения последовательных блоков файлов.

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 крупные блоки, и малоразмерные имеют свои плюсы и минусы, 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 (4.2.4.2). Начиная с первого и заканчивая предпоследним, эти блоки целиком заполнены содержимым данного файла. Соответственно, номера этих блоков хранятся среди атрибутов файла. Последний блок выделен отдельно: помимо его номера в атрибутах файла хранятся и номера занятых в нем фрагментов, принадлежащих данному файлу. Информация о блоках и фрагментах могла быть представлена разными способами: например, двоичная маска, или же номер 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. 118. 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. Пускай в системе хранятся файлы petya.txt и vasya.txt (4.2.4.3), для которых в соответствующих индексных дескрипторах хранится информация об их размерах и номеров блоков, принадлежащих файлам, в виде стартовых фрагментов. Соответственно, файл petya.txt расположен в нулевом блоке (стартовый фрагмент № 00), первом (стартовый фрагмент № 04) и второго блока (начинающегося с 08 фрагмента). 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 (стартовый фрагмент № 10), при этом во втором блоке файлу принадлежит только 10 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.

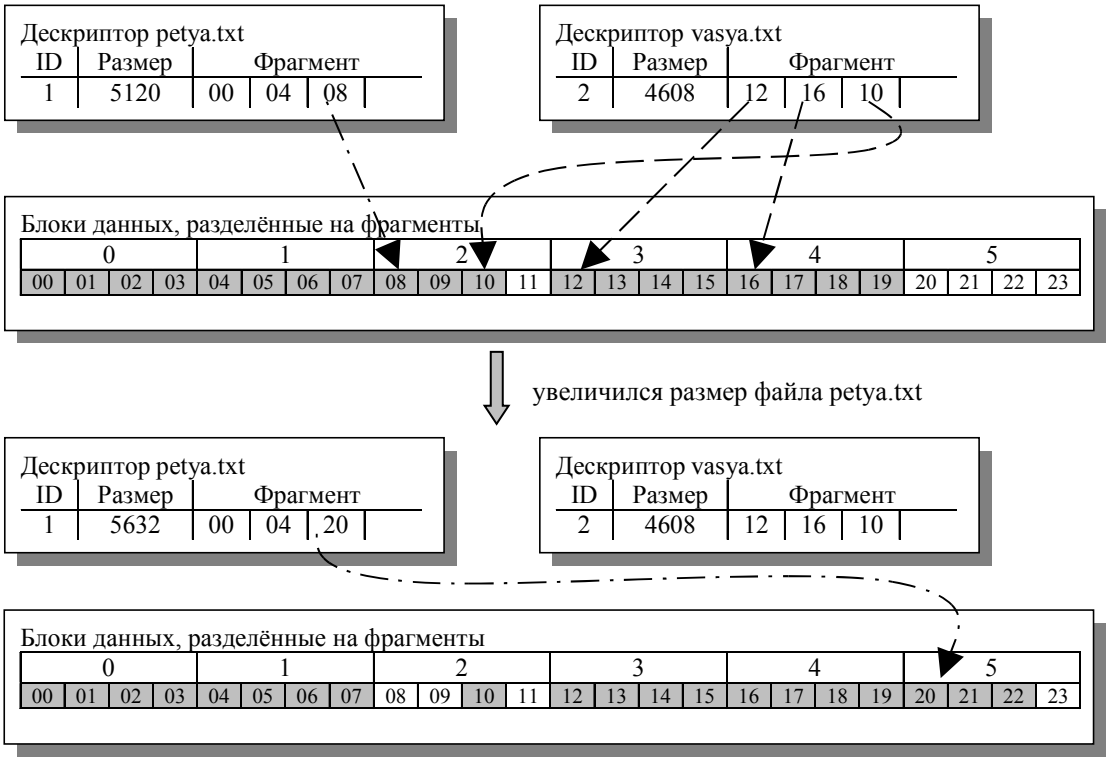


Рис. 119. Выделение пространства для файла.

Если, например, размер файла petya.txt увеличивается на столько, что конец файла не помещается в 08 и 09 фрагментах, то система начинает поиск блока с тремя **подряд идущими** свободными фрагментами. (Соответственно, если размер файл 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 (4.2.4.4). Каталог состоит из записей переменной длины, состоящих из блоков, размером в 4 байта. Начальная запись содержит номер индексного дескриптора, размер записи (т.е. ссылка на последний элемент записи) и длина 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 счет той же компрессии, а за счет тривиального «склеивания» с предыдущей записью.

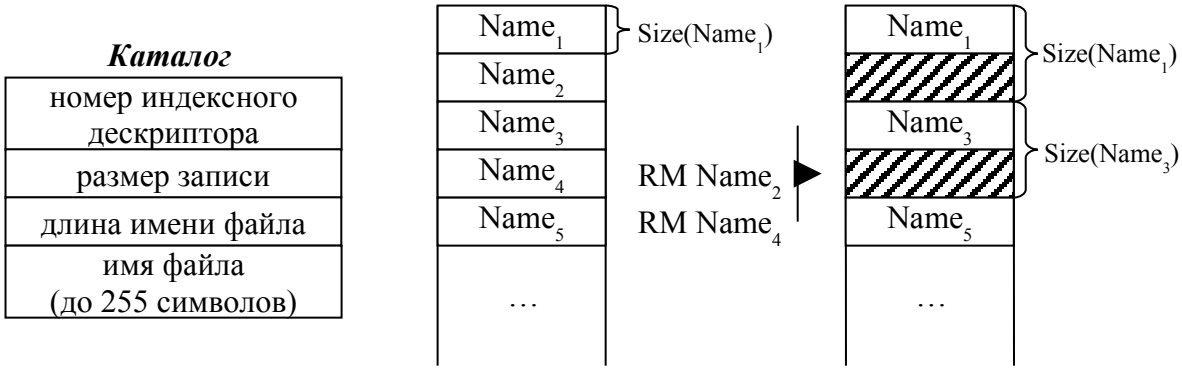


Рис. 120. Структура каталога FFS BSD.

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 будет либо 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 все равно разрешены чтение и запись информации из блокированной области. Для обеспечения корректной работы с файлом необходимо средство определения установки блокировки на той или иной области, для этого опять-таки используется системный вызов *fcntl()*. Области с рекомендательными блокировками могут пересекаться.

5 Управление оперативной памятью

Будем говорить о функциях управления оперативной памятью в контексте решения следующих основных задач. Во-первых, это осуществление контроля использования ресурса, т.е. одной из функций оперативной памяти является учет состояния каждой доступной в системе единицы: знание о том, свободна она или распределена.

Второй задачей является выбор стратегии распределения памяти. Иными словами, решается задача, какому процессу, в течение которого 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, какую 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 сегментно-страничное распределение), а также методы управления ею. При этом при обсуждении каждой стратегии будем обращать внимание на основные концепции очередной стратегии, на те аппаратные средства, необходимые для поддержания данной модели, на типовые алгоритмы, а также постараемся обсудить основные достоинства и недостатки.

5.1 Одиночное непрерывное распределение

Данная модель распределения оперативной памяти (5.1) 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. Тогда если процессор в режиме пользователя пытается обратиться в область операционной системы, возникает прерывание. Алгоритмы, используемые при таком распределении, достаточно просты, и мы не будем их здесь обсуждать.

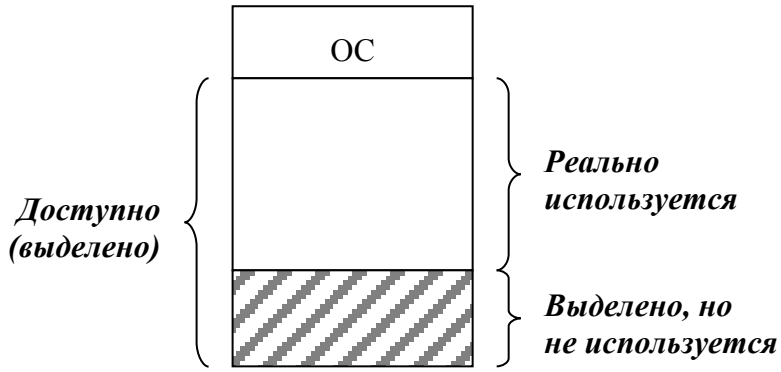


Рис. 121. Одиночное непрерывное распределение.

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 процесс обращается к памяти 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 (5.2). 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.

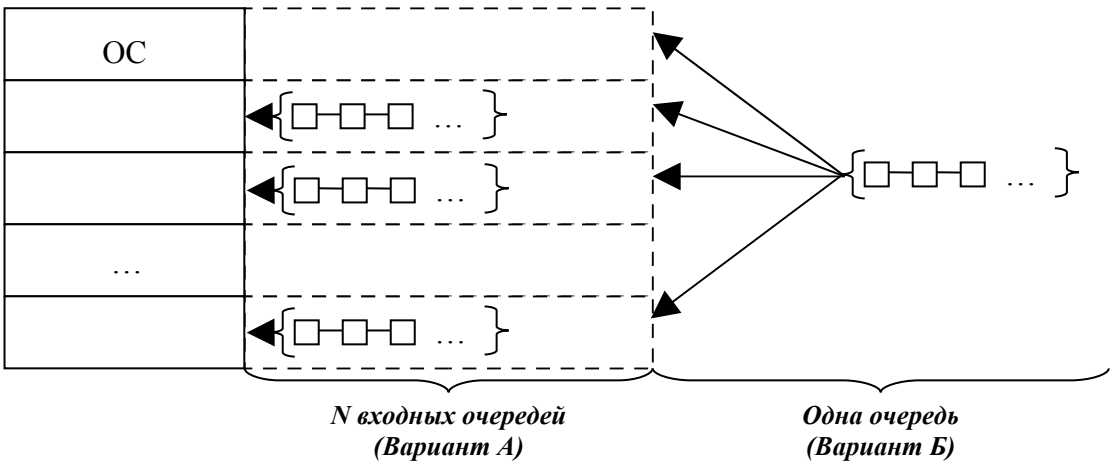


Рис. 122. Распределение неперемещаемыми разделами.

Альтернативной аппаратной реализацией может служить механизм ключей защиты (PSW — process[or] status word), которые могут находиться в слове состояния процесса и в слове состояния процессора. Данное решение подразумевает, что каждому разделу ОЗУ ставится в соответствие некоторый ключ защиты. Если аппаратура поддерживает, то в процессоре имеется слово состояния, в котором может находиться ключ защиты доступного в данный момент раздела. Соответственно, у процесса также есть некоторый ключ защиты, который тоже хранится в некотором регистре. Если при обращении к памяти эти ключи защиты совпадают, то доступ считается разрешенным, иначе возникает прерывание по защите памяти.

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 каким-то 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 этого 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 (возникающей, 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, основанного на следующем. Для каждого процесса имеется счетчик дискриминации. Под ***дискриминацией*** будем понимать ситуацию, когда в системе освободился раздел, достаточный для загрузки данного раздела, но система планирования ОС его пропустила. Соответственно, при каждой дискриминации из счетчика дискриминации процесса вычитается единица. Тогда при просмотре очереди планировщик первым делом проверяет

значение этого счетчика: если его значение — ноль 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 внутреннюю фрагментацию в разделах, поскольку зачастую процесс, загруженный в раздел, оказывается меньшего размера, чем сам 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 (5.3) 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.

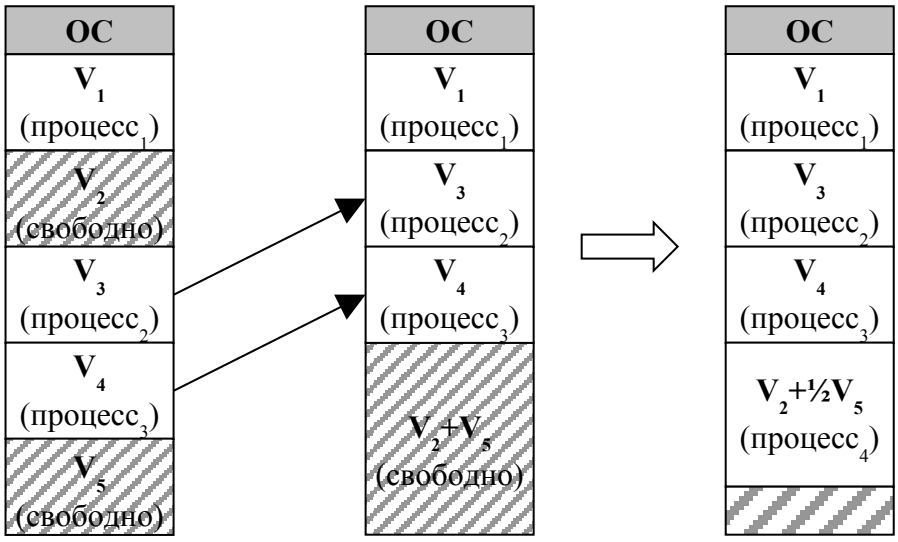


Рис. 123. Распределение перемещаемыми разделами.

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 крупный процесс, но каждый из этих участков меньше 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 началу оперативной памяти, тогда в конце ОЗУ окажется вся свободная память. Таким образом, стратегии могут быть разными.

Что касается аппаратной поддержки, то здесь она аналогична предыдущей модели: требуются аппаратные средства защиты памяти (регистры границ или же ключи защиты) и аппаратные средства, позволяющая 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.

К основному достоинству данной модели распределения памяти необходимо отнести ликвидацию 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 частота компрессии будет достаточно частой, а продолжительность компрессии с точки зрения пользователя 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 (5.4), которые называются ***страницами***. Есть ***виртуальное адресное пространство*** — это то пространство, в котором оперирует программа, и ***физическое адресное пространство*** — это то пространство, которое есть в наличии у компьютера. Соответственно, при страничном распределении памяти существуют программно-аппаратные средства, позволяющие устанавливать соответствие между виртуальными и физическими страницами. Механизм преобразования виртуального адреса в физический обсуждался выше, 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.

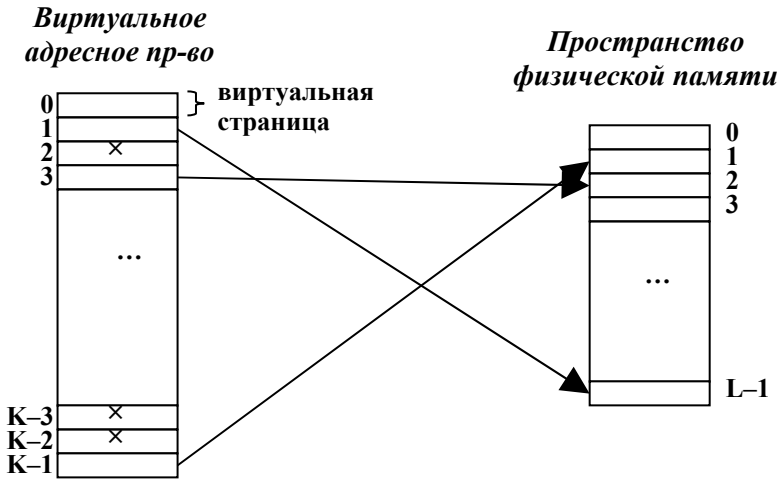


Рис. 124. Страничное распределение.

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 на 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 размещения 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 (5.4) 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.

Рис. 125. 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, 5.4). Данный метод подразумевает наличие аппаратной таблицы относительно небольшого размера (порядка 8 – 128 записей). Данная таблицы 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.

Теперь, имея виртуальный адрес, состоящий 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.

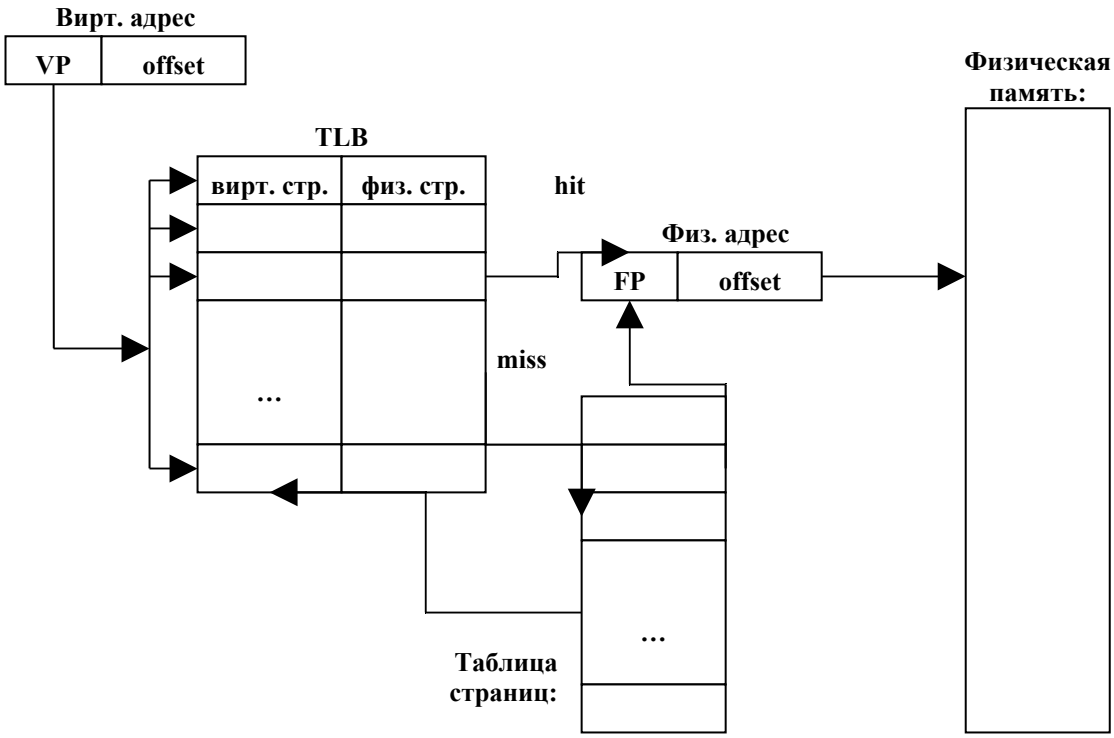


Рис. 126. TLB-таблица (Translation Look-aside Buffer).

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. это все хранит 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** (5.4). В этом случае информация о странице представляется не в виде одного номера страницы, а в виде совокупности номеров, используя которые посредством обращения к соответствующим таблицам, участвующим в иерархии (это может быть 2-х-, 3-х- или даже 4-хуровневая иерархия), можно получить номер соответствующей физической страницы.

Пускай имеется 32-разрядный виртуальный адрес, который в свете рассмотренной ранее модели может, например, содержать 20-разрядный номер виртуальной страницы и 12-разрядного значения 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 групп страниц найти необходимую ячейку, содержащую начальный адрес таблицы второго уровня, затем по этому адресу и по значению смещения в виртуальном адресе находится нужная запись в таблице страниц второго уровня, которая уже и содержит номер соответствующей физической страницы.

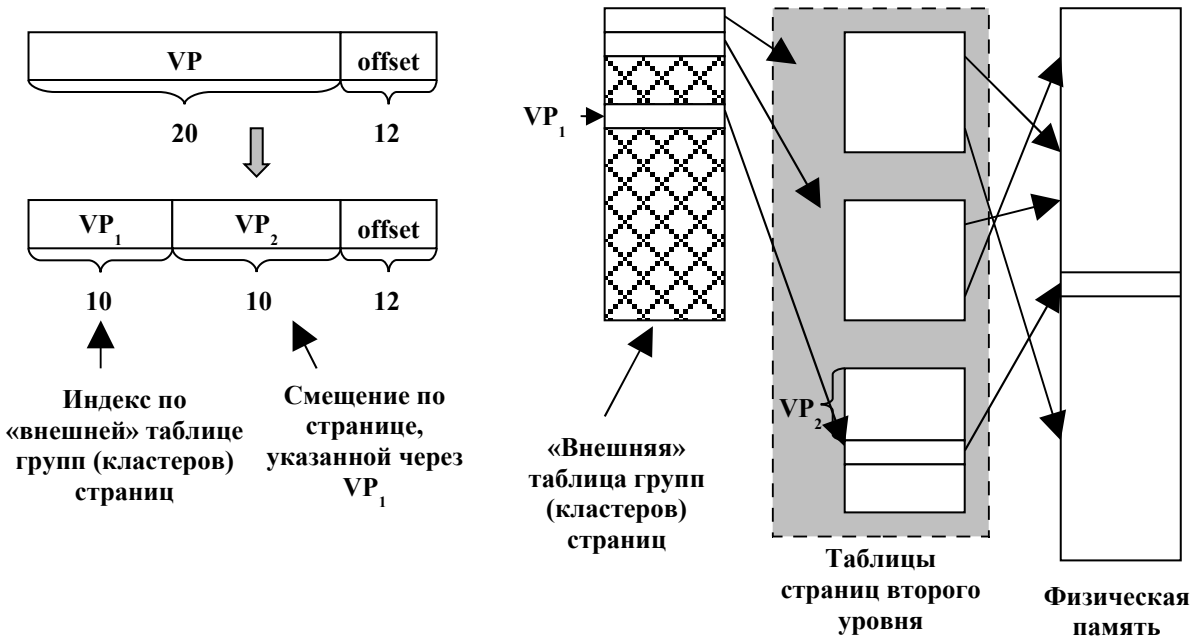


Рис. 127. Иерархическая организация таблицы страниц.

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 больше 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, которое основано на использовании ***хеширования*** (использования т.н. ***хеш-таблиц***), базирующееся, в свою очередь, на использовании ***хеш-функции***, или функции расстановки (5.4). Эти функции используются в следующей задаче: пускай 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.

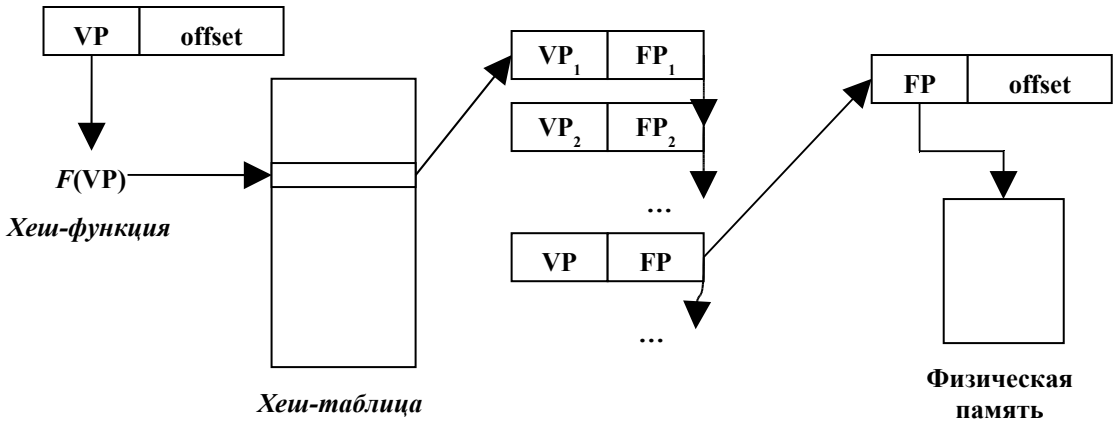


Рис. 128. Использование хеш-таблиц.

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 коллизии, 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*** (5.4). Главной сложностью данного решения является требование к процессору на аппаратном уровне работать с идентификаторами процессов (их PID). Примерами таких процессоров могут служить процессоры из линеек SPARC и PowerPC.

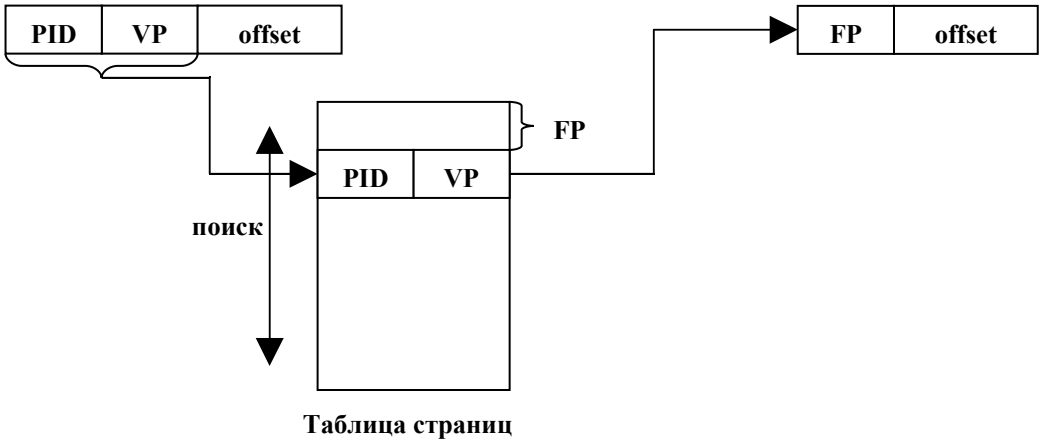


Рис. 129. Инвертированные таблицы страниц.

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 страницы (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 странице, 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 контекстам 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 возможно 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, она поступает следующим образом. Все страницы, принадлежащие данному процессу, делятся на 4 категории в зависимости от значения 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 тогда 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 переопределяется время загрузки данной страницы 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 (5.4). 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.

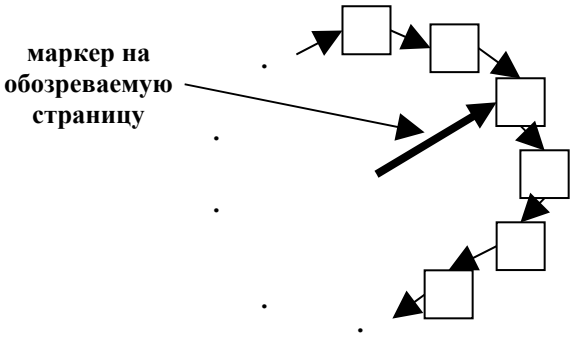


Рис. 130. Замещение страниц. Алгоритм «Часы».

Следующая группа алгоритмов позволяют учитывать более адекватно старение и использование страниц и, соответственно, 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.

Пускай 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, какая-то страница в некоторый период времени интенсивно использовалась, и 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, основанного на том, что каждый раз по таймеру значение счетчика сдвигается на 1 разряд влево, после чего последний (правый) разряд устанавливается в значение R-признака.

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 плохо 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. Итак, вот основные недостатки страничного распределения памяти.

Избавиться от указанных недостатков на концептуальном уровне призвана модель сегментного распределения памяти (5.5). 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.

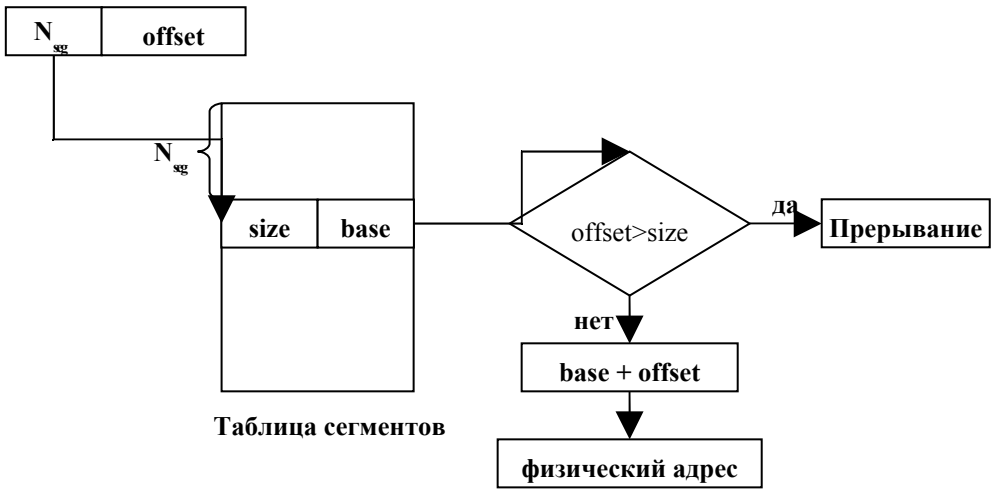


Рис. 131. Сегментное распределение.

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 чего 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 в той модели 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 упрощенную модель этой реализации (5.6). Виртуальный адрес в этой модели представляется в виде ***селектора*** (информации о сегменте) и смещения в сегменте.

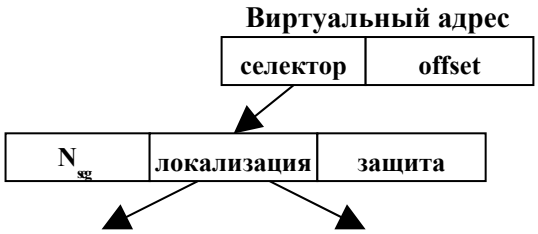


Рис. 132. Сегментно-страничное распределение. Упрощенная модель Intel.

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 (5.6). 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 отводится 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.

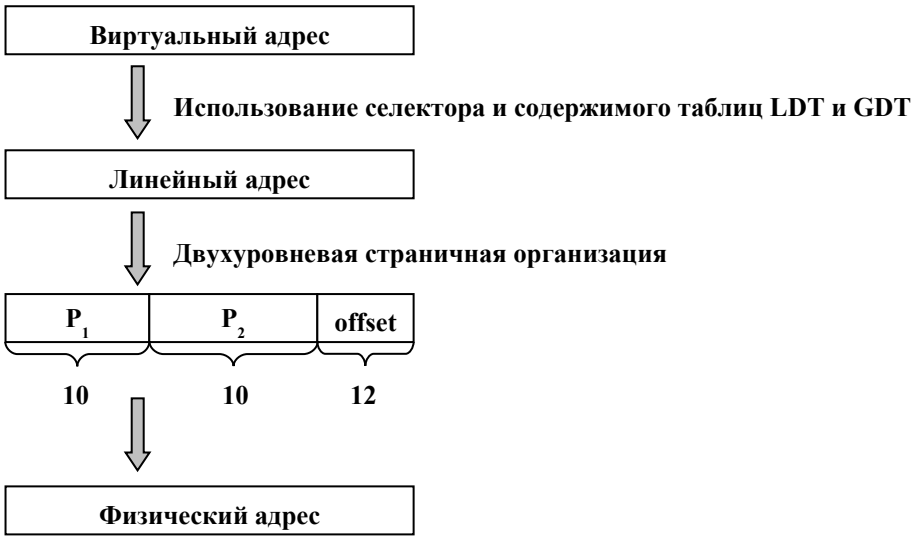


Рис. 133. Схема преобразования адресов.

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 (6.1.1.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).

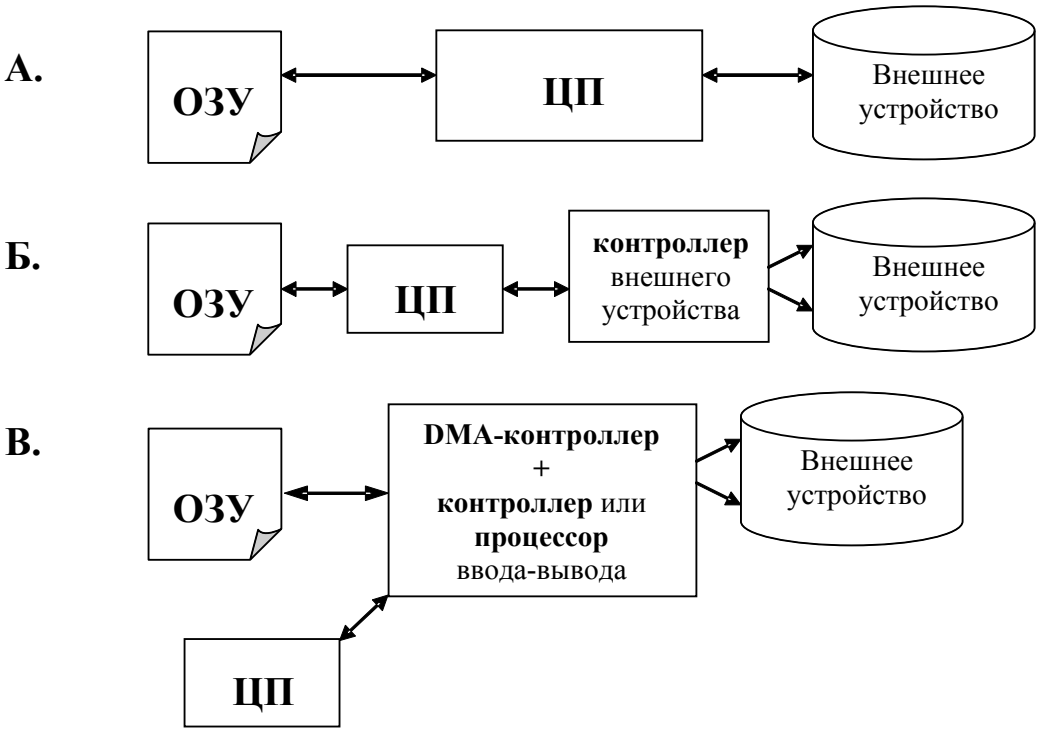


Рис. 134. Модели управления внешними устройствами: непосредственное (А), синхронное/асинхронное (Б), с использованием контроллера прямого доступа или процессора (канала) ввода-вывода.

Вторая модель, появившаяся с развитием вычислительной техники, связана с появлением **специализированных контроллеров** устройств, которые концептуально располагались между центральным процессором и соответствующими внешними устройствами (6.1.1.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** модели, 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 свои какие-то 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, 6.1.1.B). Процессор генерировал последовательность управляющих команд, указывая координаты в оперативной памяти, куда надо положить или откуда 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 **специализированным процессором** (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 макрокомандами, являющимися функционально-емкими. 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 (6.1.2). В основании лежит **аппаратура**, а далее следуют программные уровни, начиная с уровня **программ обработки прерываний**, затем — **драйверы физических устройств**, а на вершине иерархии лежит уровень **драйверов логических устройств**, причем каждый уровень строится на основании нижележащего уровня.

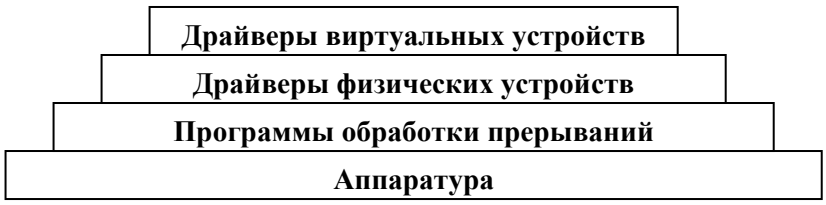


Рис. 135. Иерархия архитектуры программного управления внешними устройствами.

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, пользователь, пожелавший 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. Заведомо известно, что любое внешнее устройство работает медленнее центральной части компьютера, и, соответственно, стоит проблема сглаживания разброса производительностей различных компонент системы. Причем, если речь идет об устройствах, не являющихся устройствами оперативного доступа, т.е. не является устройством, к которому 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 реализации буфера, а если дело касается магнитного диска, рассчитанного для использования 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 (т.е. происходит обращение и считывание соответствующей дорожки). Пускай имеется очередь запросов к следующим дорожкам: 4, 40, 11, 35, 7, 14. 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 (6.1.3) 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	

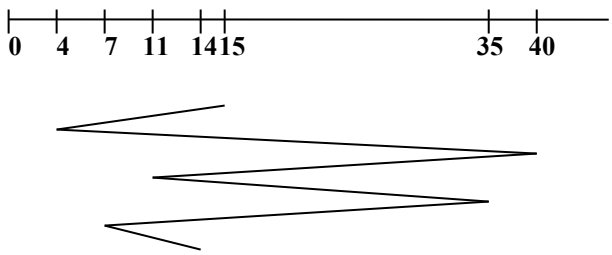


Рис. 136. Планирование дисковых обменов. Модель FIFO.

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 (6.1.3).

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. 137. 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 номер 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 (6.1.3).

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. 138. 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 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 (6.1.3).

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. 139. 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 (6.1.3).

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. 140. 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 (способ разделения может быть произвольным, в частности, по локализации запросов, по времени поступления и т.д.) делится на N подочередей, затем по какой-либо стратегии выбирается очередь, которая будет обрабатываться (например, по порядку их 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 богатому 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, порции данных фиксированного размера, называемые ***полосами***, которым осуществляется обмен в данных системах. Размер полосы зависит от конкретного устройства (при обсуждении файловых систем была упомянута иерархия различных блоков, так 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 (6.1.4). 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.

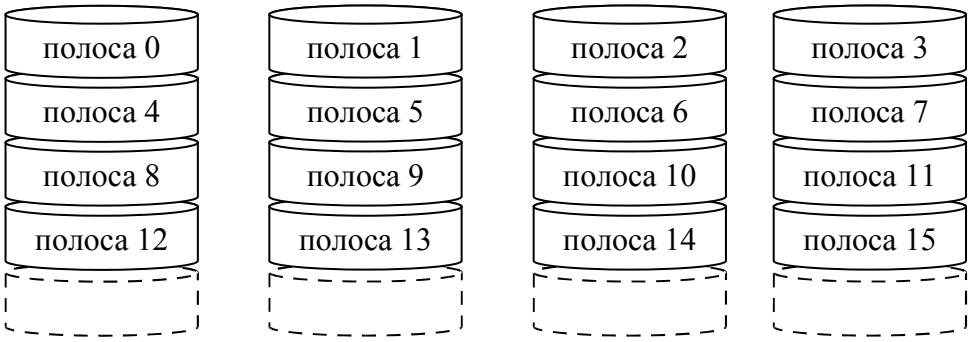


Рис. 141. RAID 0.

RAID 1, или система зеркалирования (6.1.4). 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.

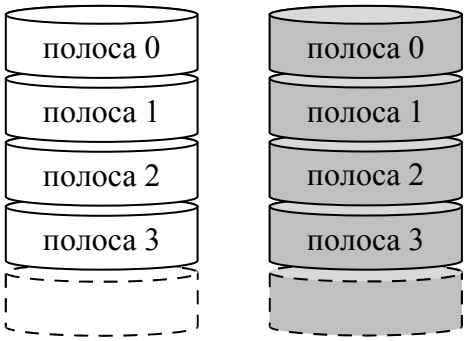


Рис. 142. RAID 1.

Данная модель является достаточно дорогой, причем дороговизна заключается не в непосредственной стоимости дисков (если предположить, что средняя цена диска 100 USD, то организация на покупку 10 основных дисков и 10 дисков-дублеров 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 электропитанием 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**, 6.1.4, и **RAID 3**, 6.1.4) — 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, позволяющие 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 должно быть 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.

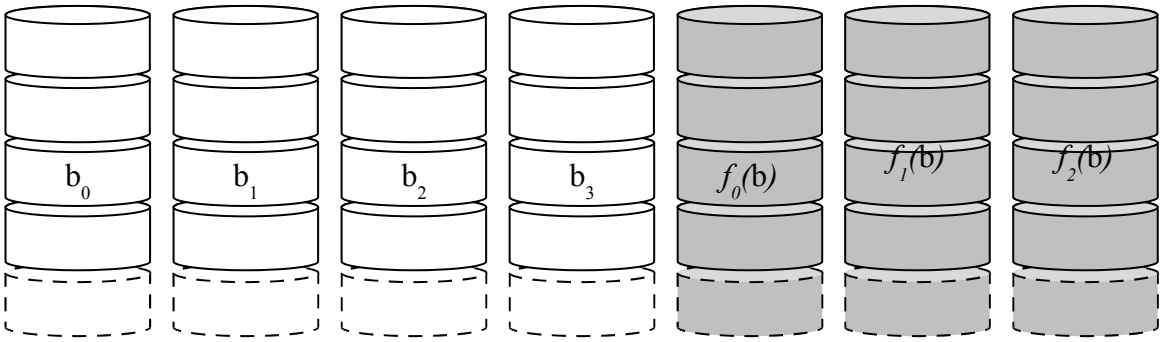
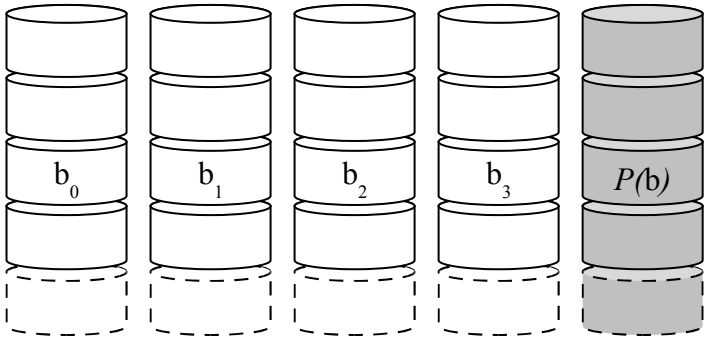


Рис. 143. RAID 2. Избыточность с кодами Хэмминга (Hamming, исправляет одинарные и выявляет двойные ошибки).

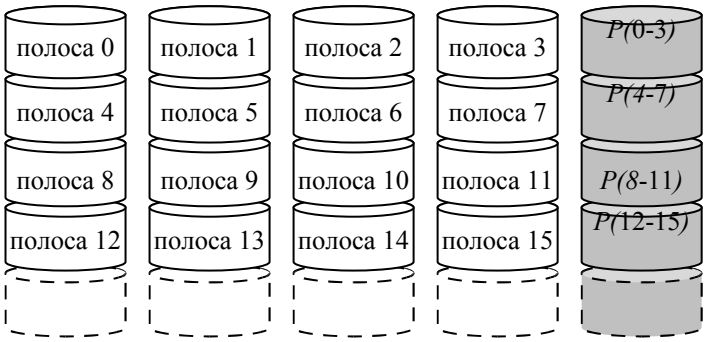


В данном случае имеется 4 диска данных и 1 диск четности. Тогда для диска четности:
$X_4(i)=X_0(i)xorX_1(i)xorX_2(i)xorX_3(i)$
Если произойдет потеря данных на первом диске, то для восстановления достаточно воспользоваться формулой:
$X_1(i)=X_0(i)xorX_2(i)xorX_3(i)xorX_4(i)$

Рис. 144. RAID 3. Четность с чередующимися битами.

RAID 4 является упрощением RAID 3 (6.1.4). 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** (6.1.4) и **RAID 6** (6.1.4) спроектированы так, чтобы повысить надежность системы по сравнению с RAID 3 и 4 уровней. Опасно оказывается хранить важную информацию (в данном случае полосы четности) на одном носителе, т.к. при каждой записи происходит всегда обращение 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).



В данном случае имеется 4 диска данных и 1 диск четности. Тогда для диска четности:
$X_4(i)=X_0(i)xorX_1(i)xorX_2(i)xorX_3(i)$

После обновления полосы на первом диске:
$X_{4new}(i)=X_4(i)xorX_1(i)xorX_{1new}(i)$

Рис. 145. RAID 4.

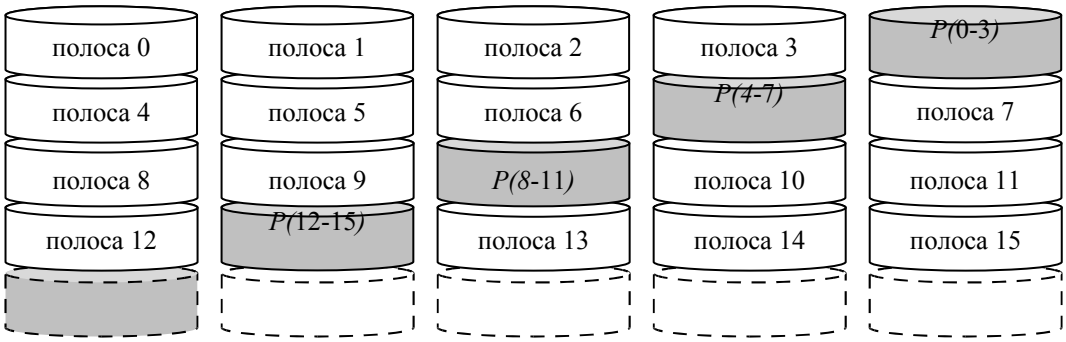


Рис. 146. RAID 5. Распределенная четность (циклическое распределение четности).

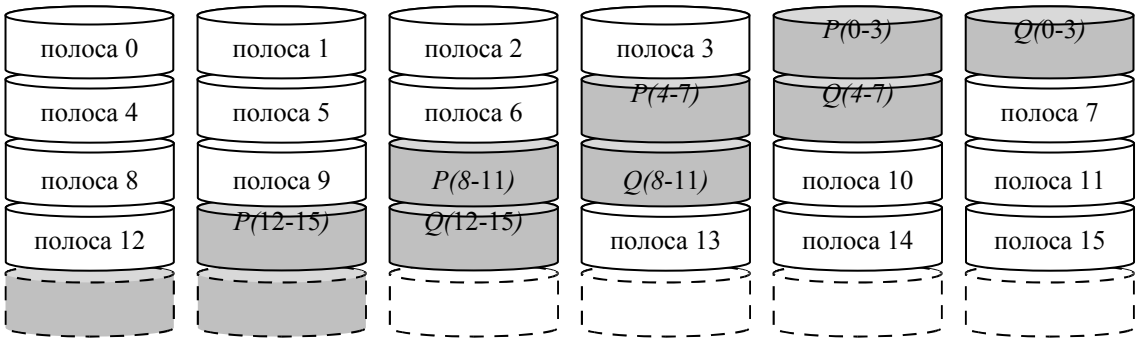


Рис. 147. RAID 6. Двойная избыточность (циклическое распределение четности с использованием двух схем контроля; требуется N+2 диска).

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, а зачастую кратен 512 байтам. 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 устройства, с которыми

обмен не осуществим. Примером такого устройства может служить таймер: реально обмены с таймером не происходят, он используется для генерации в системе через определенные промежутки времени прерываний, но относится таймер 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 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 драйвера устройства);
- связывание выбранного для именования устройства имени 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 соответствующей типу файла устройства таблице драйверов. А младший номер (minor number) — это некоторая дополнительная информация, передаваемая драйверу при обращении. За счет этого реализуется механизм, когда один драйвер может управлять несколькими сходными устройствами.

6.2.2 Системные таблицы драйверов устройств

Для регистрации драйверов в системе используются две системные таблицы: таблицы блок-ориентированных устройств — **bdevsw**, и таблица байт-ориентированных устройств — **cdevsw**. Соответственно, старший номер хранит ссылку на драйвер, хранящийся 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 реализовано 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 системы она 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 в этом случае 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 предполагали, как и большинство систем, «жесткие» 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 (когда есть готовые объектные модули) ядра. 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 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 разные, соответственно, команды, при этом обычно присутствуют как команды подключения драйверов, так и симметричные команды удаления драйверов.

6.2.5 Организация обмена данными с файлами

В этом разделе мы рассмотрим механизм организации обмена данными с файлами, после чего станет понятным, что происходит в системе, когда один и тот же файл открывается в системе одновременно несколькими процессами, а 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. ***файловый дескриптор*** — 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 открытых в данный момент в системе файлов. 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** (6.2.5). Пускай 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.

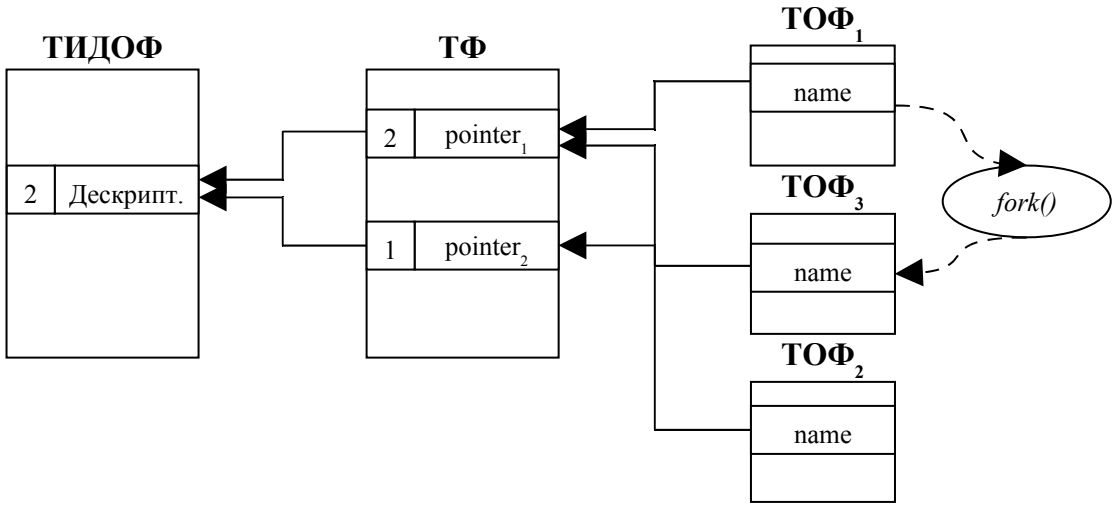


Рис. 148. Организация обмена данными с файлами.

Пускай 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 дочерний 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.

6.2.6 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. Все действия тут те же самые (вплоть до отдельных нюансов). Цель кэширования — минимизация обменов с внешними устройствами.

Для буферизации используют пул буферов, размером в один блок каждый. Вкратце рассмотрим алгоритм, состоящий из пяти действий.

1. Поиск заданного блока в буферном пуле. Если удачно, то переход на п.6.2.6
2. Поиск буфера в буферном пуле для чтения и размещения заданного блока.
3. Чтение блока в найденный буфер.
4. Изменение счетчика времени во всех буферах.

5. Содержимое данного буфера передается в качестве результата.

Итак, повторимся: 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, располагаемых 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.

6.2.7 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 файловой системы, 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 восстановима.