Федеральное агентство связи
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ
им. проф. М.А. БОНЧ-БРУЕВИЧА»

А. В. Красов
А. Ю. Цветков

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЗАЩИЩЕННЫХ ИНФОКОММУНИКАЦИОННЫХ СИСТЕМ

УЧЕБНОЕ ПОСОБИЕ

СПб ГУТ)))

Санкт-Петербург
2013

УДК
ББК
К

Рецензент
С.Е. Душин – доктор технических наук, профессор
(СПбГЭТУ «ЛЭТИ»)

Рекомендовано к печати
редакционно-издательским советом университета

К?? Системное программное обеспечение защищенных инфокоммуникационных систем: учебное пособие / *А.В. Красов, А.Ю. Цветков.* – СПб. : Издательство СПбГУТ, 2013. – 78 с.

Содержит учебные материалы по дисциплине «Системное программное обеспечение защищенных инфокоммуникационных систем», в которой студенты знакомятся с основами разработки защищенных приложений на языке C++ под *nix операционные системы.

Предназначены для студентов, обучающихся по направлениям подготовки бакалавров ФГОС ВПО III 210700 «Инфокоммуникационые технологии и системы связи», 100100 «Сервис», 090900 «Информационная безопасность».

УДК
ББК

Оглавление

Оглавление 3
Введение 4
1. Введение в операционные системы 6
    1.1 Классификация операционных систем 7
    1.2 Введение в Unix системы 9
    1.3 Управление процессами 13
    1.4 Управление памятью 19
2. Системное программирование в UNIX 34
    2.1 Работа с файловой структурой UNIX 34
    2.2 Процессы в операционной системе UNIX 46
    2.3 Сигналы в операционной системе UNIX 52
    2.4 Взаимодействие процессов 57
    2.5 Семафоры 62
    2.6 Интерфейс транспортного уровня 67
Заключение 75
Литература 76

Введение

Работа пользователя с компьютером, это в первую очередь работа с операционной системой. Данное взаимодействие показано на рис.1.

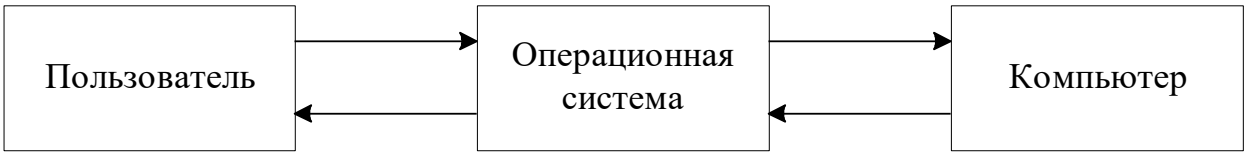


Рис. 1. Взаимодействие пользователя с компьютером

Пользователь не выполняет все операции по управлению аппаратными ресурсами компьютера непосредственно сам, за него управлением аппаратурой компьютера занимается операционная система. Таким образом, операционную систему можно рассматривать программу виртуальной машины, с которой непосредственно и работает пользователь. Такая организация избавляет большинство пользователей от необходимости знать уникальные особенности работы конкретной аппаратуры компьютера: портов ввода/вывода, работы с памятью и т.п. и реализовывать работу с ними на языках низкого уровня.

Операционная система скрывает от программиста все аппаратные особенности и предоставляет возможность простого и удобного доступа к ресурсам реального компьютера, от эффективности организации которого напрямую зависит эффективность работы пользователя на компьютере в целом.

Современные условия развития информационных технологий привело к тому, что одной из важнейших характеристик операционных систем стала необходимость обеспечения информационной безопасности. Другой особенностью современных операционных систем стала тесная интеграция с различными сетевыми компонентами. Появление сетевых приложений к операционным системам часто приводят к появлению изъянов в защите и нарушению целостности операционных систем.

Решение проблемы обеспечения информационной безопасности основывается на двух основных тенденциях:

- Создание защищенной операционной системы с нуля. В этом случае функции защиты информации непосредственно закладываются в ядро операционной системы, как единое целое с функциями самой операционной системы. Однако данный подход очень трудоемок и может оказаться не «по карману» не только компаниям, но даже и многим государствам.
- Встраивание функций защиты информации в уже существующую операционную систему. Решение данной задачи требует открытости архитектуры операционной системы для разработчика средств защиты. В наибольшей степени такие возможности предоставляет операционная

система UNIX, на базе которой сделано большинство современных защищенных операционных систем.

Исходя из выше изложенного, в данном пособии рассматривается архитектура построения операционных систем и вопросы разработки системных приложений под операционную систему UNIX.

1. Введение в операционные системы

Операционная система выполняет две функции:

- Предоставляет пользователю виртуальную машину, обеспечивающую стандартизованный интерфейс к аппаратным ресурсам реального компьютера.
- Обеспечивает управление ресурсами компьютера.

Управление ресурсами предполагает распределение ресурсов между задачами и пользователями и отслеживание состояния ресурса.

Потребность в данных функции возникло не сразу, в первых поколениях ЭВМ функции операционных систем выполнял машинный язык, на котором осуществлялась решение задач. Фактически данный машинный язык и его библиотеки можно рассматривать в качестве прообраза операционных систем. Расширение парка ЭВМ потребовало выделения набора функций общих для различных пользователей, необходимости переключения различных задач без выключения ЭВМ. Выделение этих функций в отдельную задачу и привело к появлению операционных систем первого поколения.

Эволюцию операционных систем можно представить следующим образом:

Первый период (1945 -1955)

Программирование осуществлялось на машинном языке, который выполнял функции прообраза операционной системы. Все задачи организации вычислительного процесса решались вручную каждым программистом.

Второй период (1955 - 1965)

В эти годы появились первые алгоритмические языки. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программ за другой. Был разработан язык управления пакетом заданий (колоды перфокарт).

Третий период (1965 - 1980)

Появление программно–совместимых машин. Программная совместимость требовала и совместимости (универсальности) операционных систем, работающих на различных ЭВМ. Первым универсальным операционным системам была характерна низкая эффективность. Появление таких понятий как: Мультипрограммирование, спулинг, разделения времени.

Четвертый период (1980 - настоящее время)

Современное поколение операционных систем. С середины 80-х стали бурно развиваться сетевые или распределенные операционные системы, в качестве примера которых можно привести UNIX, Novel, WindowsNT.

1.1 Классификация операционных систем

Операционные системы могут различаться особенностями реализации алгоритмов управления ресурсами компьютера, особенностями методов проектирования, архитектурными особенностями аппаратных платформ, областью применения.

Поддержка многозадачности

Операционные системы делятся на: однозадачные (например, MS-DOS, MSX) и многозадачные (OS/2, UNIX, Windows 95 и выше).

Однозадачные операционные системы предоставляют ресурсы всей виртуальной машины только одному пользователю.

Многозадачные операционные системы осуществляют разделение ресурсов реального компьютера между несколькими задачами и управление разделяемыми между задачами ресурсами.

Поддержка многопользовательского режима

В зависимости от особенностей работы пользователей операционные системы делят на однопользовательские (MS-DOS, Windows 3.x, ранние версии OS/2) и многопользовательские (UNIX, Windows NT).

Однопользовательские операционные системы – все выполняемые задачи принадлежат одному логическому пользователю, даже если это на самом деле различные люди. В этом случае не возможно разделять процессы различных пользователей и обеспечить их защиту.

Многопользовательские операционные системы – разделение пользователей позволяет организовать разделение и защиту от несанкционированного доступа процессов и ресурсов различных пользователей.

Реализация многозадачности

В зависимости от реализации многозадачности разделяют не вытесняющую (NetWare, Windows 3.x) и вытесняющую многозадачность (Windows NT, OS/2, UNIX).

В операционных системах с вытесняющей многозадачностью переключение процессов осуществляется только после завершения обработки текущего процесса.

При вытесняющей многозадачности инициатором переключения процессов выступает операционная система.

Многозадачные операционные системы разделяют:

- системы пакетной обработки (операционная система ЕС ЭВМ);
- системы разделения времени (UNIX, VMS);
- системы реального времени (QNX, RT/11).

Поддержка многонитевости – распараллеливания вычислений в рамках одной задачи.

Многопроцессорная обработка (*мультипроцессирование)*

При многопроцессорной обработке все системные и пользовательские процессы или их составные части распределяются между группой

процессоров. Многопроцессорная обработка позволяет существенно повысить скорость выполнения процессов. К многопроцессорным операционным системам относятся Solaris 2.x, Open Server 3.x , OS/2, Windows NT и Novell NetWare 4.1.

Поддержка различных аппаратных платформ

В настоящее время созданы операционные системы для различных аппаратных платформ, таких как: персональные компьютеры, мини-компьютеры, мейнфреймы, кластеры и сети ЭВМ. Учет специфики аппаратных средств привило к появлению различных операционных систем.

Мобильные операционные системы

В зависимости от наличия возможности переносить операционную систему с одного компьютера на другой выделяют класс мобильных операционных систем, например Novell NetWare, UNIX.

Рассмотренные выше особенности функционирования тесто связанны с особенностями реализации операционных систем. К основным концепциям построения операционных систем относятся:

Принципы построения ядра системы – монолитное ядро или микроядерный подход.

При монолитном ядре операционная система является одной программой, выполняющей большинство системных функций.

При микроядерной архитектуре микроядро выполняет только минимально–достаточный набор наиболее существенных функций операционной системы по управлению аппаратурой компьютера на самом низком уровне. Все остальные функции, условно называемые высокого уровня выполняют внешние компоненты (системные программы) операционной системы.

Микроядерная архитектура позволяет организовать более гибкую и надежную операционную систему за счет некоторого снижения быстродействия.

Объектно–ориентированная система

Технология объектно-ориентированного программирования несет огромные преимущества при разработке программного обеспечения, в том числе и системного, по сравнению с другими стилями программирования. Фундаментальные механизмы объектно–ориентированного программирования – наследование, инкапсуляция, полиморфизм позволяют организовать эффективные программные структуры с защитой данных отдельных классов от несанкционированного доступа, конструирования и уничтожение объектов.

Множественность прикладных сред

Концепция множественных прикладных сред реализуется на основе микроядра, над которым создаются виртуальные среды для работы пользовательских приложений различных операционных систем.

Распределенная организация операционных систем

При работе в компьютерной сети предоставляется возможность распределить некоторые, в первую очередь ресурсоемкие, функции операционной системы между компьютерами сети. Реализация такого разделения требует создание единой сетевой службы управления разделяемыми ресурсами.

1.2 Введение в Unix системы

Операционная система GNU/Linux родилась в 1991 году. Но её рождение произошло не на пустом месте и имеет свою предысторию.

В 70-х годах XX века такого понятия, как операционная система практически ещё не существовало. Каждый производимый компьютер поставлялся просто как набор оборудования, управляемый системой команд его процессора. Пользователю (в то время это были крупные организации, так как компьютеры были достаточно дороги) предоставлялось самому выбирать способ создания программного обеспечения для работы с таким компьютером. В такой ситуации переход на более новый и мощный компьютер вынуждал переписывать созданное программное обеспечение практически с нуля.

Чтобы как-то изменить данную ситуацию, специалистом Bell Labs Денисом Ритчи был создан язык программирования C для того, чтобы можно было использовать одну и ту же программу на компьютерах с разной системой команд процессора (при условии? что для данного процессора существует компилятор языка C). После создания компилятора C при участии Кена Томсона была создана операционная система UNICS, перед которой ставились задачи обеспечения многопользовательности и многозадачности. Впоследствии название системы эволюционировало в UNIX, а её исходный код был распространён среди других организаций. Это привело к возникновению нескольких ветвей операционной системы UNIX. Каждая организация развивала свой вариант системы (например, BSD – разработанная в институте Беркли, AIX – от фирмы IBM, HP-UX, SunOS и другие).

В 80-х годах XX века получили развитие персональные компьютеры, которые были доступны обычному пользователю. И к концу 80-х годов сложилась ситуация, что компьютеры можно было условно разделить на две группы: 1) большие мощные промышленные системы, управляемые многозадачными системами и обеспечивающие одновременную работу многих пользователей; 2) персональные компьютеры, управляемые

системами семейства DOS и Windows, которые не могли в то время обеспечить многозадачность и многопользовательность.

Функциональные характеристики

К основным функциям ядра ОС UNIX принято относить следующие:

1) Инициализация системы - функция запуска и раскрутки. Ядро системы обеспечивает средство раскрутки (bootstrap), которое обеспечивает загрузку полного ядра в память компьютера и запускает ядро.
1) Управление процессами и нитями - функция создания, завершения и отслеживания существующих процессов и нитей ("процессов", выполняемых на общей виртуальной памяти). Поскольку ОС UNIX является мультипроцессной операционной системой, ядро обеспечивает разделение между запущенными процессами времени процессора (или процессоров в мультипроцессорных системах) и других ресурсов компьютера для создания внешнего ощущения того, что процессы реально выполняются в параллель.
2) Управление памятью - функция отображения практически неограниченной виртуальной памяти процессов в физическую оперативную память компьютера, которая имеет ограниченные размеры. Соответствующий компонент ядра обеспечивает разделяемое использование одних и тех же областей оперативной памяти несколькими процессами с использованием внешней памяти.
3) Управление файлами - функция, реализующая абстракцию файловой системы, - иерархии каталогов и файлов. Файловые системы ОС UNIX поддерживают несколько типов файлов. Некоторые файлы могут содержать данные в формате ASCII, другие будут соответствовать внешним устройствам. В файловой системе хранятся объектные файлы, выполняемые файлы и т.д. Файлы обычно хранятся на устройствах внешней памяти; доступ к ним обеспечивается средствами ядра. В мире UNIX существует несколько типов организации файловых систем. Современные варианты ОС UNIX одновременно поддерживают большинство типов файловых систем.
4) Коммуникационные средства - функция, обеспечивающая возможности обмена данными между процессами, выполняющимися внутри одного компьютера (IPC - Inter-Process Communications), между процессами, выполняющимися в разных узлах локальной или глобальной сети передачи данных, а также между процессами и драйверами внешних устройств.
5) Программный интерфейс - функция, обеспечивающая доступ к возможностям ядра со стороны пользовательских процессов на основе механизма системных вызовов, оформленных в виде библиотеки функций.

Особенности архитектуры ОС Unix

Архитектура ОС UNIX – многоуровневая. На нижнем уровне, непосредственно над оборудованием, работает ядро операционной системы. Функции ядра доступны через интерфейс системных вызовов, образующих второй уровень. На следующем уровне работают командные интерпретаторы, команды и утилиты системного администрирования, коммуникационные драйверы и протоколы, - все то, что обычно относят к системному программному обеспечению. Наконец, внешний уровень образуют прикладные программы пользователя, сетевые и другие коммуникационные службы, СУБД и утилиты.

Способы управления процессами и ресурсами

Файлы и процессы, являются центральными понятиями модели операционной системы UNIX.

Обращения к операционной системе выглядят так же, как обычные вызовы функций в программах на языке Си, и библиотеки устанавливают соответствие между этими вызовами функций и элементарными системными операциями. При этом программы на ассемблере могут обращаться к операционной системе непосредственно, без использования библиотеки системных вызовов. Программы часто обращаются к другим библиотекам, таким как библиотека стандартных подпрограмм ввода-вывода, достигая тем самым более полного использования системных услуг. Для этого во время компиляции библиотеки связываются с программами и частично включаются в программу пользователя. Совокупность обращений к операционной системе разделена на те обращения, которые взаимодействуют с подсистемой управления файлами, и те, которые взаимодействуют с подсистемой управления процессами. Файловая подсистема управляет файлами, размещает записи файлов, управляет свободным пространством, доступом к файлам и поиском данных для пользователей. Процессы взаимодействуют с подсистемой управления файлами, используя при этом совокупность специальных обращений к операционной системе, таких как open (для того, чтобы открыть файл на чтение или запись),close, read, write, stat (запросить атрибуты файла), chown (изменить запись с информацией о владельце файла) и chmod (изменить права доступа к файлу).

Подсистема управления файлами обращается к данным, которые хранятся в файле, используя буферный механизм, управляющий потоком данных между ядром и устройствами внешней памяти. Буферный механизм, взаимодействуя с драйверами устройств ввода-вывода блоками, инициирует передачу данных к ядру и обратно. Драйверы устройств являются такими модулями в составе ядра, которые управляют работой периферийных устройств. Устройства ввода-вывода блоками относятся программы пользователя к типу запоминающих устройств с произвольной выборкой; их драйверы построены таким образом, что все остальные

компоненты системы воспринимают эти устройства как запоминающие устройства с произвольной выборкой. Например, драйвер запоминающего устройства на магнитной ленте позволяет ядру системы воспринимать это устройство как запоминающее устройство с произвольной выборкой. Подсистема управления файлами также непосредственно взаимодействует с драйверами устройств "неструктурированного" ввода-вывода, без вмешательства буферного механизма. К устройствам неструктурированного ввода-вывода, иногда именуемым устройствами посимвольного ввода-вывода (текстовыми), относятся устройства, отличные от устройств ввода-вывода блоками.

Подсистема управления процессами отвечает за синхронизацию процессов, взаимодействие процессов, распределение памяти и планирование выполнения процессов. Подсистема управления файлами и подсистема управления процессами взаимодействуют между собой, когда файл загружается в память на выполнение: подсистема управления процессами читает в память исполняемые файлы перед тем, как их выполнить.

Примерами обращений к операционной системе, используемых при управлении процессами, могут служить fork (создание нового процесса), exec (наложение образа программы на выполняемый процесс), exit (завершение выполнения процесса), wait (синхронизация продолжения выполнения основного процесса с моментом выхода из порожденного процесса), brk (управление размером памяти, выделенной процессу) и signal (управление реакцией процесса на возникновение экстраординарных событий.

Модуль распределения памяти контролирует выделение памяти процессам. Если в какой-то момент система испытывает недостаток в физической памяти для запуска всех процессов, ядро пересылает процессы между основной и внешней памятью с тем, чтобы все процессы имели возможность выполняться. Существует два способа управления распределением памяти: выгрузка (подкачка) и замещение страниц. Программу подкачки иногда называют планировщиком, т.к. она "планирует" выделение памяти процессам и оказывает влияние на работу планировщика центрального процессора. «Планировщик» планирует очередность выполнения процессов до тех пор, пока они добровольно не освободят центральный процессор, дождавшись выделения какого-либо ресурса, или пока ядро системы не выгрузит их после того, как их время выполнения превысит заранее определенный квант времени. Планировщик выбирает на выполнение готовый к запуску процесс с наивысшим приоритетом; выполнение предыдущего процесса (приостановленного) будет продолжено тогда, когда его приоритет будет наивысшим среди приоритетов всех готовых к запуску процессов. Существует несколько

форм взаимодействия процессов между собой, от асинхронного обмена сигналами о событиях до синхронного обмена сообщениями.

Наконец, аппаратный контроль отвечает за обработку прерываний и за связь с машиной. Такие устройства, как диски и терминалы, могут прерывать работу центрального процессора во время выполнения процесса. При этом ядро системы после обработки прерывания может возобновить выполнение прерванного процесса. Прерывания обрабатываются не самими процессами, а специальными функциями ядра системы, перечисленными в контексте выполняемого процесса.

1.3 Управление процессами

Основная функция операционной системы – управление ресурсами компьютера. Заявка на потребление системных ресурсов называется процессом (задачей). Задача управления (планирования) процессов это распределение времени процессора между множеством процессов и организация взаимодействия процессов между собой, отслеживание ресурсов находящихся в совместном использовании.

Состояние процессов

В многозадачной системе процесс может находиться в одном из состояний:

выполнение – активное состояние процесса;

ожидание – пассивное состояние процесса;

готовность – пассивное состояние процесса (полностью готового, по ресурсам для выполнения) вызванное тем, что процессор занят обслуживанием другого процесса.

Граф состояний процесса показан на рис. 1.1.

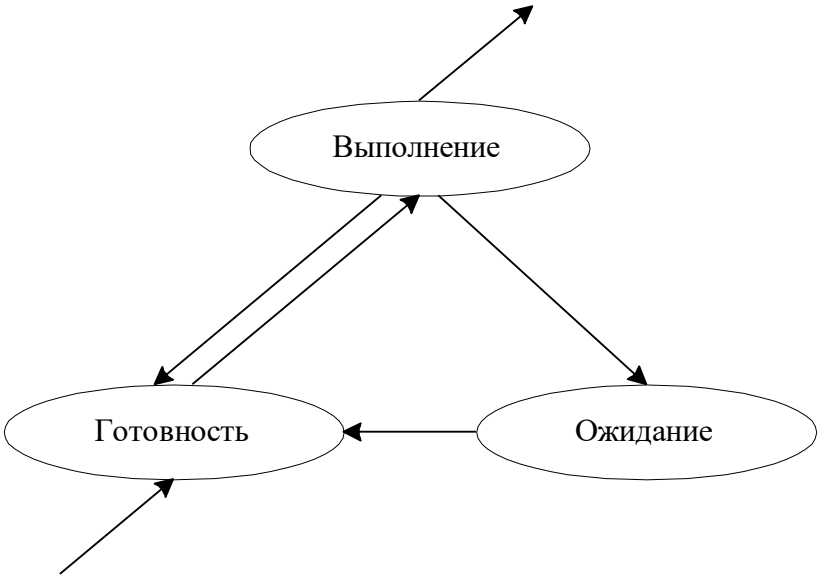


Рис. 1.1. Граф состояний процесса

Жизненный цикл процесса начинается с состояния «готовность», когда процесс готов к выполнению и ждет своей очереди. Из состояния «готовность» процесс переходит в состояние «выполнения». В однозадачной операционной системе в состоянии «выполнения» может находиться только один процесс. Он занимает процессор либо до завершения своего выполнения, либо процесс может быть вытеснен по инициативе процессора или в случае если ему не хватает ресурсов, в состояния «ожидание» или «готовность». В этих состояниях могут находиться одновременно сразу несколько процессов.

Контекст и дескриптор процесса

Контекстом процесса – состояние операционной среды отображается состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода–вывода, кодами ошибок выполняемых данным процессом системных вызовов и т.д.

Дескриптором процесса – идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другая информация.

Дескриптор процесса по сравнению с контекстом содержит более оперативную информацию, которая должна быть легко доступна подсистеме планирования процессов. Контекст процесса содержит менее актуальную информацию и используется операционной системой только после того, как принято решение о возобновлении прерванного процесса.

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки.

Для создания процесса необходимо:

1. создать информационные структуры, описывающие данный процесс, то есть его дескриптор и контекст;
2. включить дескриптор нового процесса в очередь готовых процессов;
3. загрузить кодовый сегмент процесса в оперативную память или в область свопинга.

Алгоритмы планирования процессов

Планирование процессов включает в себя решение следующих задач:

1. определение момента времени для смены выполняемого процесса;
2. выбор процесса на выполнение из очереди готовых процессов;
3. переключение контекстов "старого" и "нового" процессов.

Первые две задачи решаются программными средствами, а последняя, в значительной степени аппаратно.

Существует множество различных алгоритмов планирования процессов, в первую очередь – алгоритмы, основанные на квантовании, и алгоритмы, основанные на приоритетах. Под приоритетом понимается

число, которое ставится в соответствие важности данного процесса для системы.

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

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

В системах с абсолютными приоритетами, кроме того, смена процесса возможна, в случае если в очереди процессов появился процесс с большим значением приоритета, чем у выполняемого в данный момент времени.

Многие системы используют для алгоритмов планирования оба этих принципа: для работы процесса выделяется определенное количество квантов времени, а выборка следующего процесса из очереди происходит с учетом приоритетов.

Различают два основных типа алгоритмов планирования процессов – вытесняющие и не вытесняющие.

Невытесняющая многозадачность – предусматривает, что активный процесс выполняется до тех пор, пока он сам не отдаст управление планировщику операционной системы. Использование алгоритма невытесняющей многозадачности в операционной системе Novell NetWare, позволило добиться высокой скорости выполнения файловых операций, что очень важно для файл-сервера.

Вытесняющая многозадачность – предполагает, что решение о переключение выполнения процесса принимается планировщиком операционной системы. Планирование процессов на основе вытесняющей многозадачности используется в большинстве современных операционных систем, таких как UNIX, Windows NT, OS/2, VAX/VMS.

Средства синхронизации процессов

При взаимодействии процессов часто возникает проблема, когда выполнение одного из процессов должно быть приостановлено до тех пор, пока другой процесс не подготовит ему необходимые данные.

На рис. 1.2 показан граф вычислительного процесса. Выполнение процесса А, должно быть приостановлено, до тех пор, пока процесс Б не подготовит необходимые данные для выполнения блока «c».

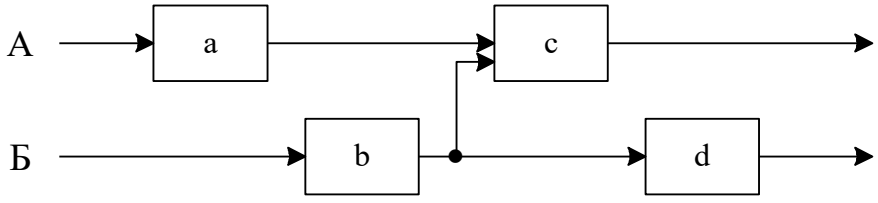


Рис. 1.2. Граф вычислительного процесса

Решение данной проблемы возможно при организации синхронизации процессов.

Критические секции

Критическая секция – часть программы, в которой осуществляется доступ к совместным ресурсам. Для нормальной работы программы необходимо обеспечить работу с указанным ресурсом во время критической сессии только одного процесса. Это может быть организованно следующим образом:

- Запрещение процессу во время нахождения в критической секции все прерывания. Данный способ может привести к сбою в работе всей системы по этому на практике, как правило, не применяется.
- Использование блокирующих переменных. Пример использования блокирующих переменных показан на блок–схеме (рис. 1.3). Данные операторы должны быть вставлены в каждый процесс, работающий с ресурсом D. Признаком что ресурс занят, является значение переменной flagD=1. Главный недостаток использования метода блокирующих переменных заключается в наличии холостого цикла ожидания освобождения ресурса. Во время данного холостого цикла процесс бесполезно тратит ресурсы процессора.
- Использование системных функций wait() и post(). Избежать бесполезных потерь времени и работе холостого цикла возможно при использовании функций работы с событиями. Функция wait(id) переводит процесс в режим ожидания до освобождения ресурса с идентификатором id. Функция post(id) сообщает операционной системе, что ресурс id освободился и можно выбрать из очереди процессов находящихся в ожидании данного ресурса следующий процесс. Блок–схема, показывающая работу с критической секцией на основе системных событий показана на рис. 1.4.

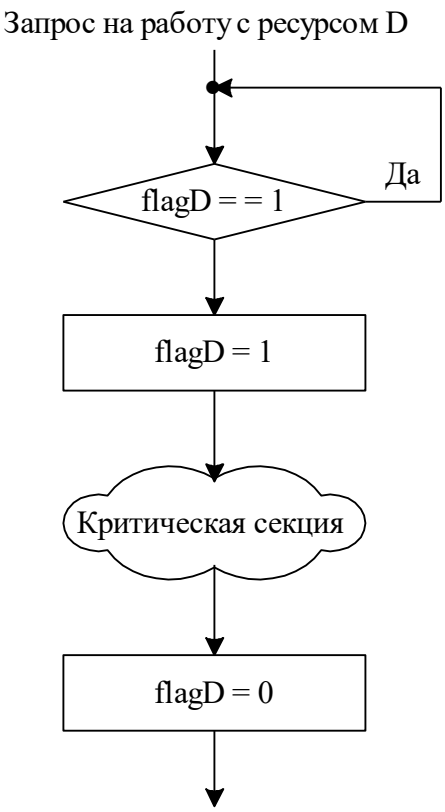


Рис. 1.3. Фрагмент блок–схемы, использующие блокирующие переменные

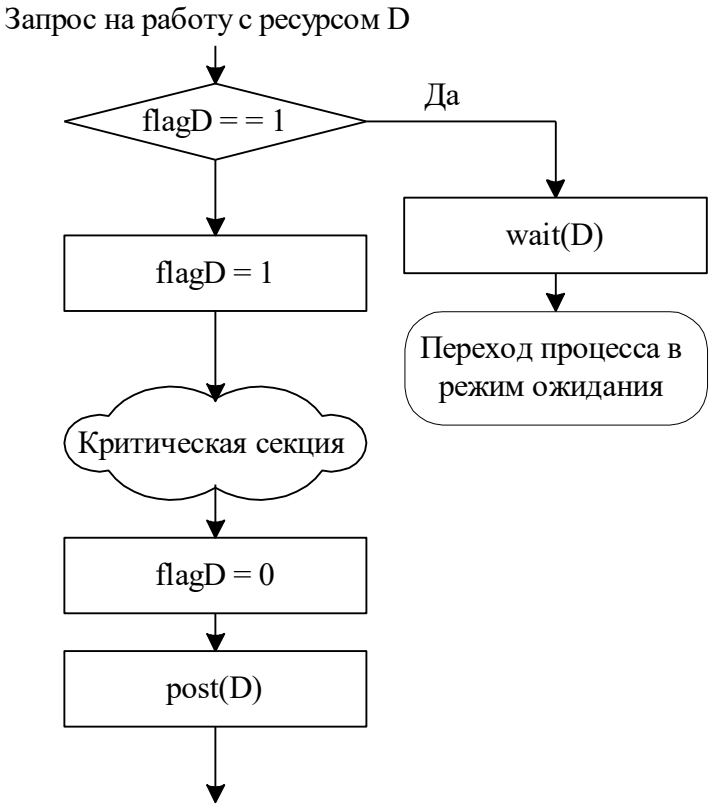


Рис.1.4. Фрагмент блок–схемы, использующий системные события для работы с критической секцией

Тупики

Использование общих ресурсов несколькими процессами может привести к их взаимной блокировке. Например, обмен данными через общий файл, представленный на рис. 1.5.

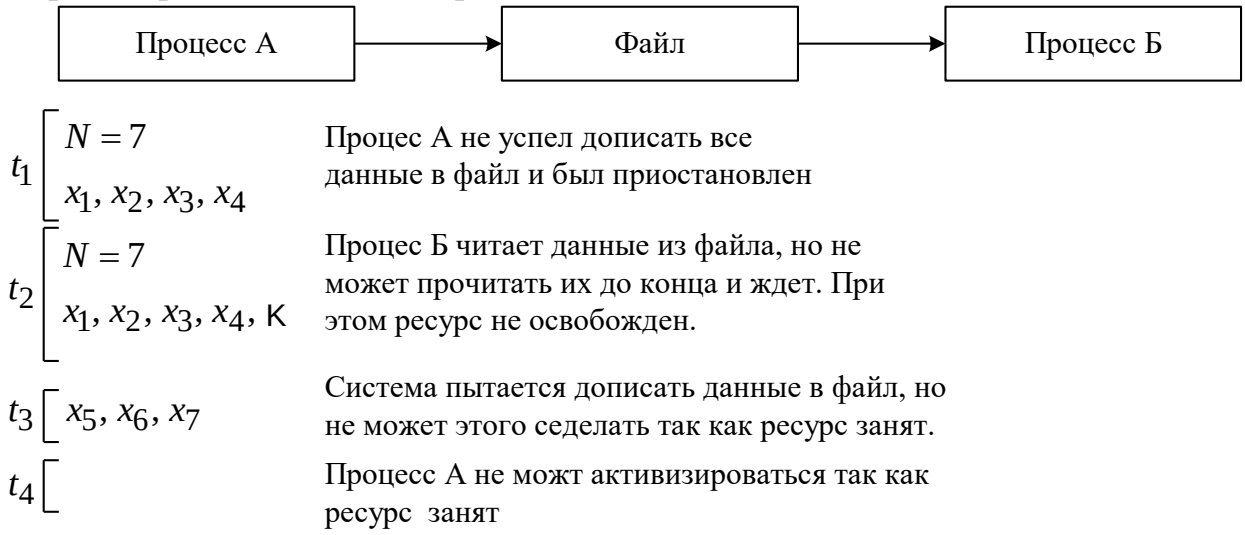


Рис. 1.5. Взаимная блокировка процессов

Процесс А записывает в файл 7 чисел с указанием их количества, но в файл записывается только первые 4 из них, например, остальные еще находятся в буфере обмена с файлом. В это время получает управление процесс Б, который открывает общий файл и читает данные. Не получив ожидаемое количество чисел процесс Б ждет, продолжая оставлять ресурс занятым. Так как файл остается открытым система не может дописать последние числа. Данная ситуация приводит к тупику, так как процесс А не может записывать новые данные пока процесс Б не освободил файл. Процесс Б не может освободить файл, пока он не прочитает ожидаемого количества чисел.

Решение проблемы возникновения тупиков может быть следующим:

• Предотвращение тупиков. Данный подход предусматривает разрешение ситуаций с возможностью возникновения тупиков на уровне исходного текста.

• Распознавание тупиков. Определение взаимных блокировок может быть построено на основании таблиц запросов к ресурсам.

• Восстановление системы после тупиков. Возникший тупик может быть разрешен за счет снятия или отката к предыдущим этапам выполнения отдельных процессов.

Нити

Нить подобна процессу, каждая нить выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Нити могут находиться в одном из следующих состояний: выполнение, ожидание и готовность.

Нити одного процесса имеют одно и то же адресное пространство и общие глобальные переменные. Наличие полного доступа ко всем виртуальным адресам остальных нитей не позволяет организовать защиту данных отдельных нитей процесса.

Нити имеют собственные:

- программный счетчик;
- стек;
- регистры;
- нити-потомки;
- состояние.

Нити разделяют:

- адресное пространство;
- глобальные переменные;
- открытые файлы;
- таймеры;
- семафоры;
- статистическую информацию.

1.4 Управление памятью

Память является важнейшим ресурсом, требующим тщательного управления со стороны операционной системы. Традиционно в самых младших адресах располагается сама операционная система.

Важнейшими функциями операционной системы по управлению памятью является

- отслеживание свободной и занятой памяти;
- защита занятой памяти от проникновения в нее других процессов;
- выделения памяти процессам;
- освобождение памяти при завершении процессов и очистка освободившейся памяти от кода и данных принадлежавших завершенному процессу;
- вытеснение процессов из оперативной памяти в свопинг файл на диске, возвращение процессов из свопинга;
- отслеживание перемещения сегментов памяти и соответствия логических и физических адресов переменных.

Работу с адресами в программах можно разделить на три уровня:

- символические имена – имена переменных и метки в программе;
- виртуальные адреса – адреса вырабатываемые компилятором для переменных и меток программы;
- физические адреса – реальные адреса размещения операторов и переменных в памяти компьютера в процессе выполнения.

Иллюстрация работы с адресами приведена на рис. 1.6.

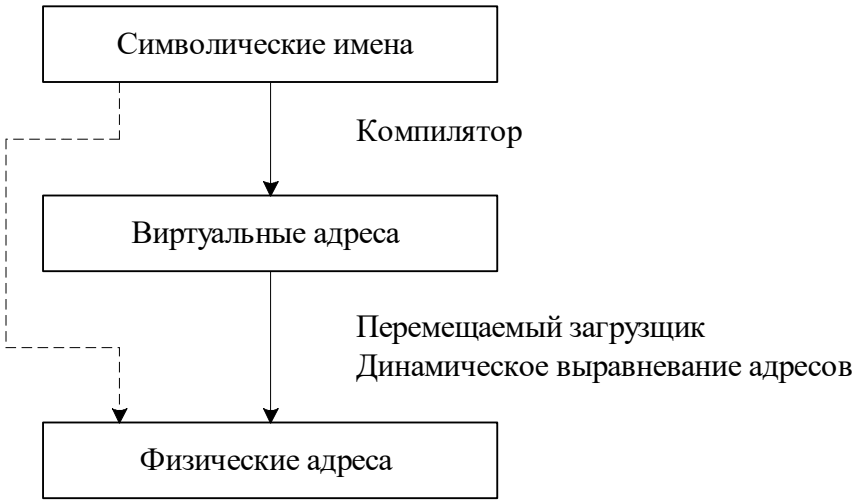


Рис. 1.6. Типы адресов

Преобразование виртуальных адресов в физические может выполняться перемещаемым загрузчиком операционной системы в процессе запуска или перемещения процесса в памяти (например, выгрузка не активного процесса в файл свопинга на диск). Кроме того, существует система динамического выравнивания адреса, реализованная аппаратно в процессоре при преобразовании из линейного адреса в физический адрес.

Пунктирная линия на рис. 7 показывает режим работы с конкретными физическими адресами в программе. Например, два процесса обмениваются данными, расположенными в заранее определенной области физической памяти.

Методы распределения памяти

Методы распределения памяти можно разделить на методы использующие обмен с использованием дискового пространства и методы, не использующие дисковое пространство. Классификация методов распределения памяти приведена на рис. 1.7.

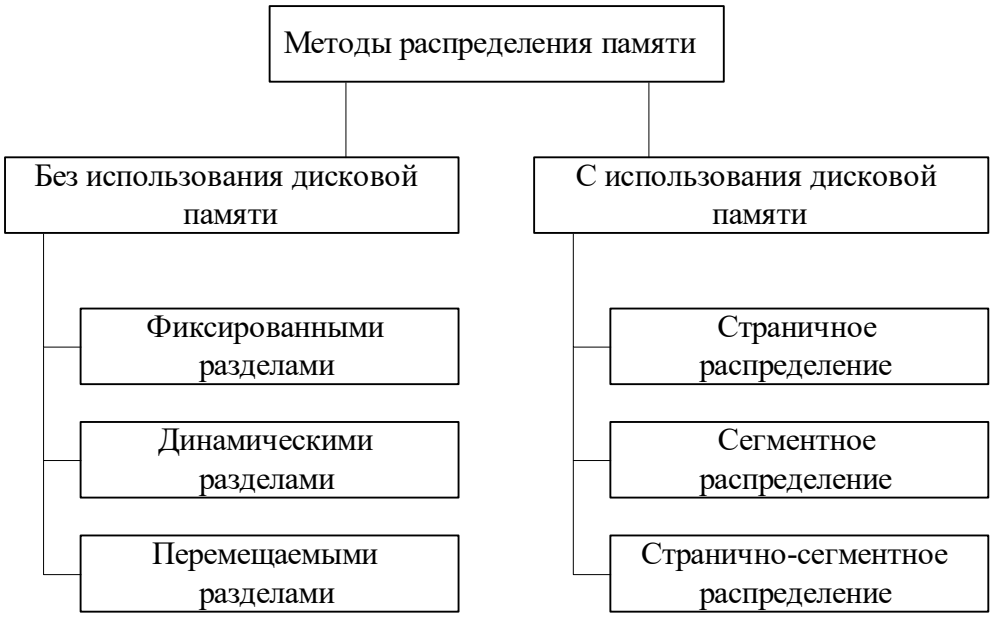


Рис. 1.7. Классификация методов распределения памяти

Методы распределения памяти без использования дисковой памяти

Фиксированные разделы

Распределение памяти с фиксированными разделами показано на рис. 1.8.

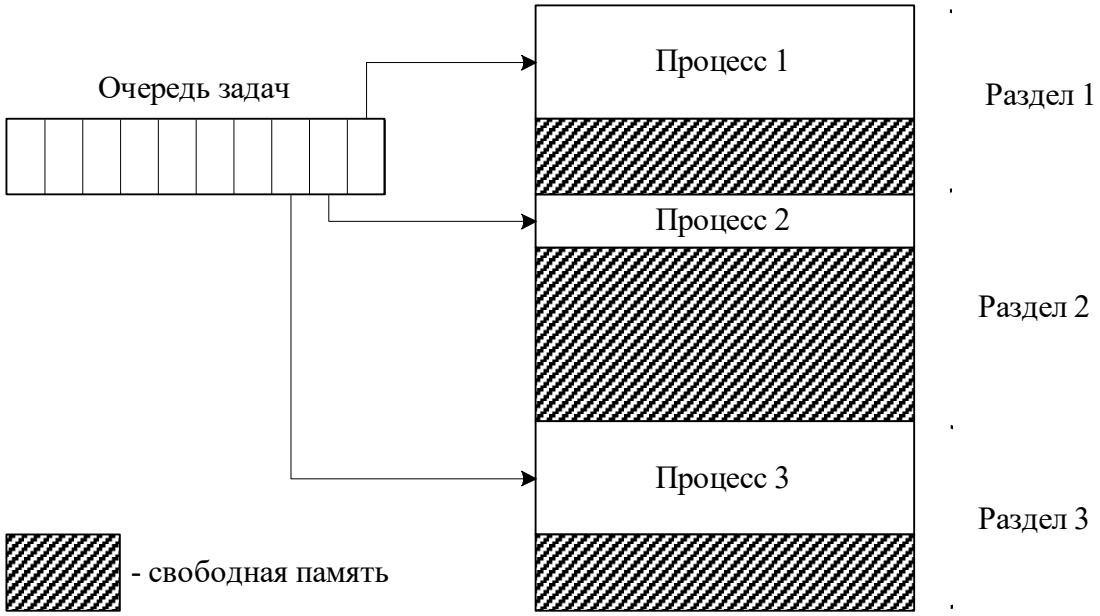


Рис. 1.8. Распределение памяти с фиксированными разделами

Распределение памяти с фиксированными разделами – самое простое в реализации. Оперативная память делится на разделы, например это, может быть выполнено программистом. Процессы из очереди задач загружаются в раздел соответствующего размера. Однако при таком способе

распределения памяти, как это видно на рис. 1.8, большое количество оперативной памяти остается не использованной.

Динамические разделы

Распределение памяти с динамическими разделами представлено на рис. 1.9.

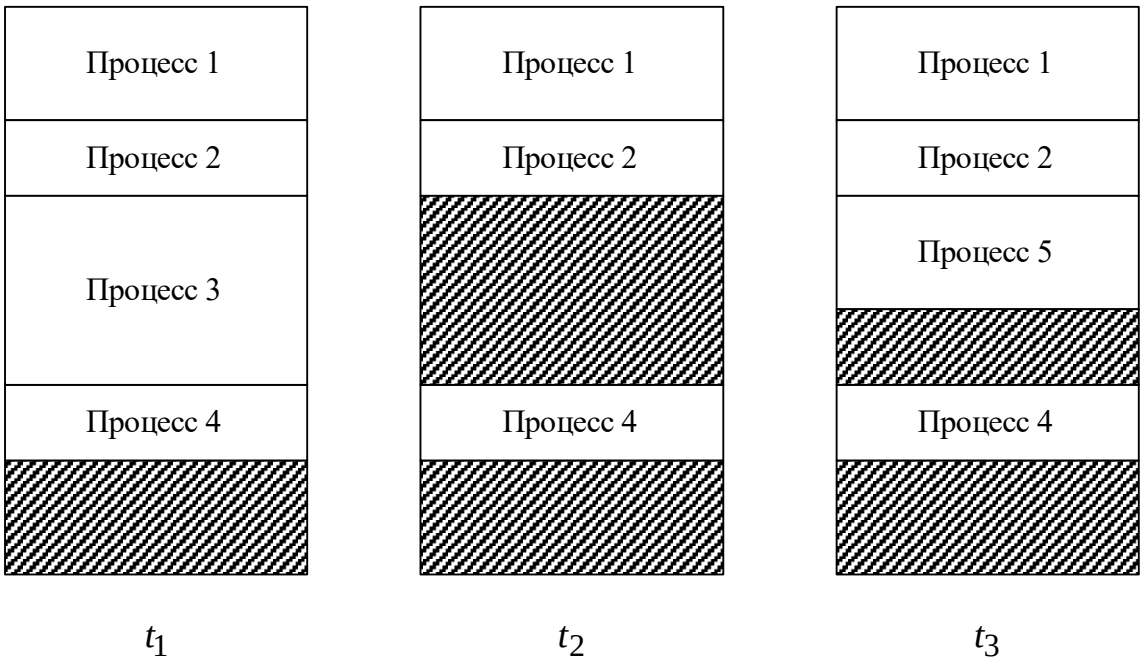


Рис. 1.9. Распределение памяти с динамическими разделами

Первоначально (момент времени t_1) процессы последовательно занимают память. В момент времени t_2 процесс 3 завершил свою работу, занятая им ранее память остается свободной. При запуске следующего приложения (процесс 5, в момент времени t_3) операционная система просматривает свободные разделы памяти и занимает под процесс первый найденный свободный раздел достаточного размера.

Длительная работа операционной системы с динамическими разделами приводит к фрагментации памяти – наличия большого количества несмежных свободных участков памяти, что может привести к невозможности запустить процесс, требующий размера большего, чем самый большой свободный участок.

Перемещаемые разделы

При распределении памяти на основе перемещаемых разделов операционная система периодически проводит сбор свободных разделов образовавшихся при фрагментации памяти. Иллюстрация данного способа приведена на рис. 1.10.

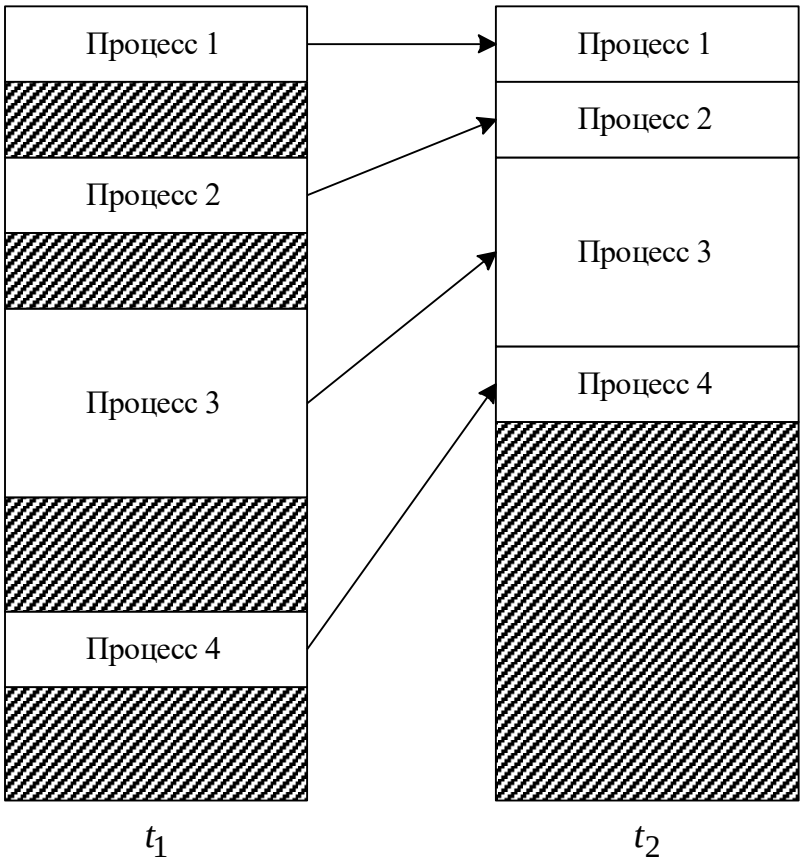


Рис. 1.10. Иллюстрация распределения памяти с перемещаемыми разделами

В момент времени t_1 система начинает сбор свободных разделов памяти для получения одного непрерывного раздела.

Данный способ работы с памятью позволяет предоставить максимальные ресурсы памяти для запуска нового процесса. Но при этом требует пересчета виртуальных адресов в физические адреса практически для всех процессов при каждой процедуре сбора свободных разделов.

Методы распределения памяти с использованием дисковой памяти

Довольно часто программисту приходится разрабатывать программы, размер которых превосходит размер оперативной памяти. Разработка их в этом случае производится с использованием виртуальной памяти.

Виртуальная память позволяет:

организовать размещение части данных и кода процессов на дисковой (или другой) памяти;

организовать автоматическую подгруздку необходимых данных и процедур в оперативную память;

автоматически организовать преобразования виртуальных адресов в физические.

Организация виртуальной памяти с использованием дискового пространства организуется в виде: страничного, сегментного и странично–

сегментного распределения памяти. Наиболее распространенных способов организации виртуальной памяти с выгрузкой на диск это работа с файлом свопинга.

Страничное распределение

Иллюстрация страничного распределения памяти приведена на рис. 1.11.

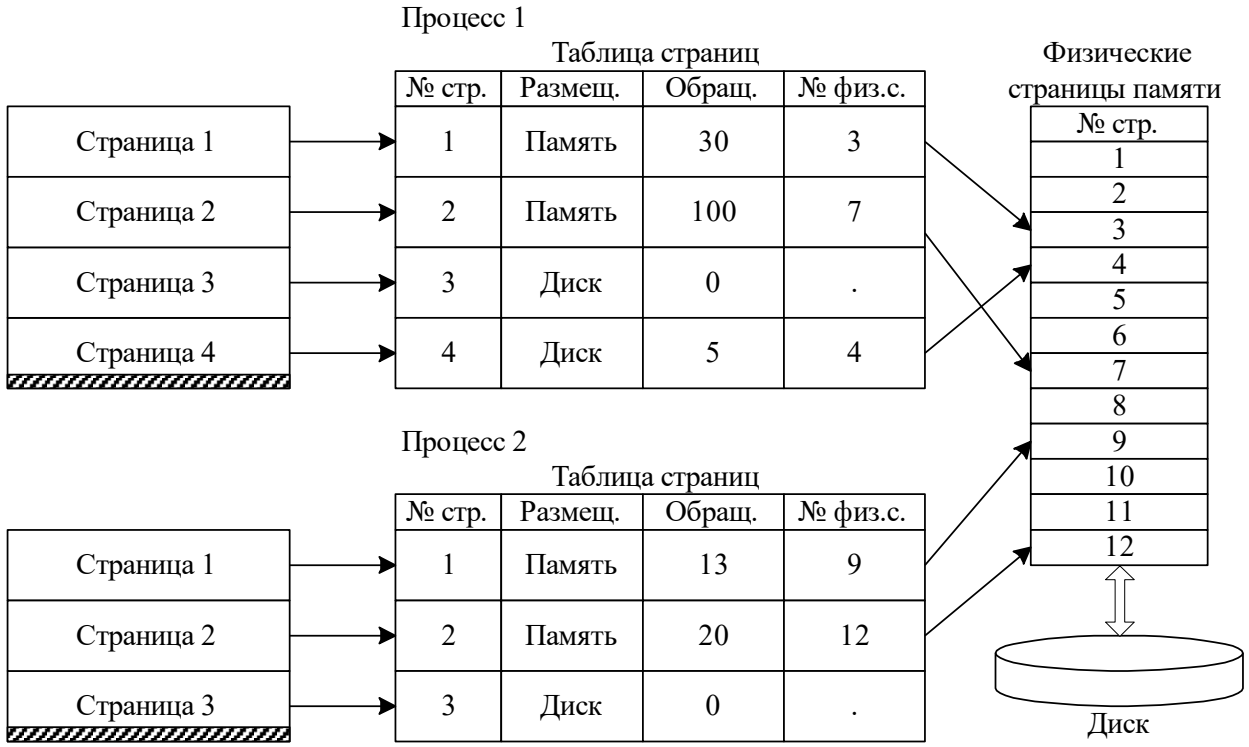


Рис. 1.11. Страничное распределение памяти

Каждый процесс, представленный на рис. 1.11, состоит из нескольких виртуальных страниц памяти. Для связи с физическими страницами для каждого из процессов строится таблица страниц, в которой размещается следующая информация: номер виртуальной страницы, способ размещения (память/диск), счетчик кол-ва обращений, номер физической страницы. Физические страницы памяти могут располагаться в произвольном порядке. Операционная система выполняет функции загрузки требуемых процессам страниц памяти с диска и, при необходимости, а так же недостатке свободной памяти, выгрузке редко используемых страниц на диск.

Размер страницы во всех операционных системах выбирается кратным 2^n, чаще всего используются размеры 512 или 1024.

Механизм формирования физического адреса при страничном распределении памяти представлен на рис. 1.12.

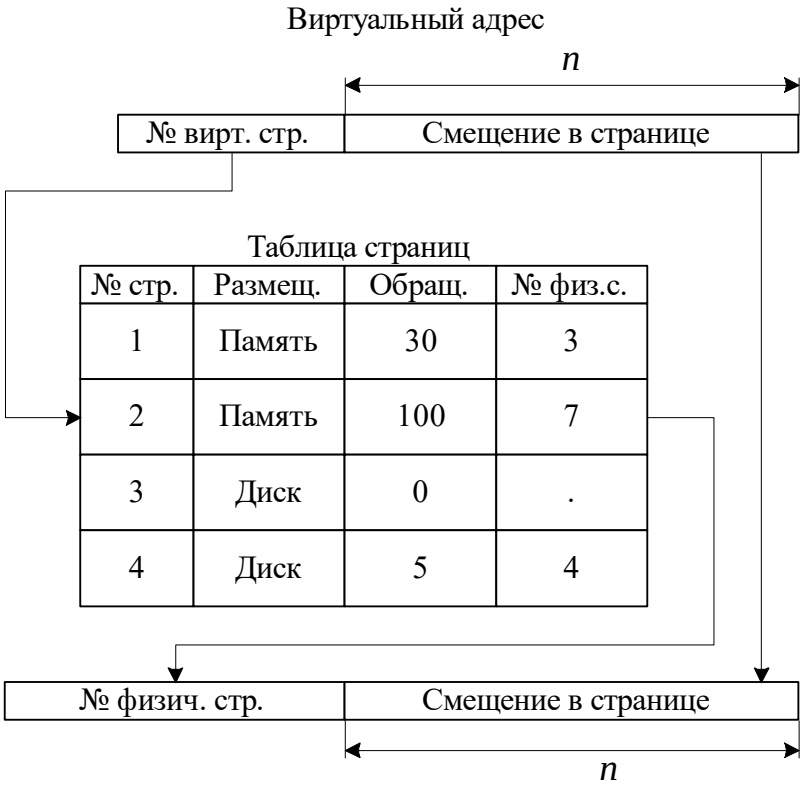


Рис. 1.12. Механизм формирования адреса

Виртуальный адрес, состоящий из номера виртуальной станицы и смещения в ней, преобразуется следующим образом: младшие *n* разрядов адреса, соответствующие смещению в странице передаются в младшие разряды физического адреса; номер физической страницы извлекается из таблицы страниц процесса по номеру виртуальной страницы, причем размеры адресов могут не совпадать.

Сегментное разделение

Главный недостаток страничного распределения памяти это одинаковый размер страниц и не возможность организации доступа нескольких процессов к одному модулю. Эти недостатки устранены при сегментном способе распределения памяти, представленном на рис. 1.13. Кроме того, сегментное разделение памяти предусматривает установление различных атрибутов для сегментов – доступ только по чтению, по записи и другие.

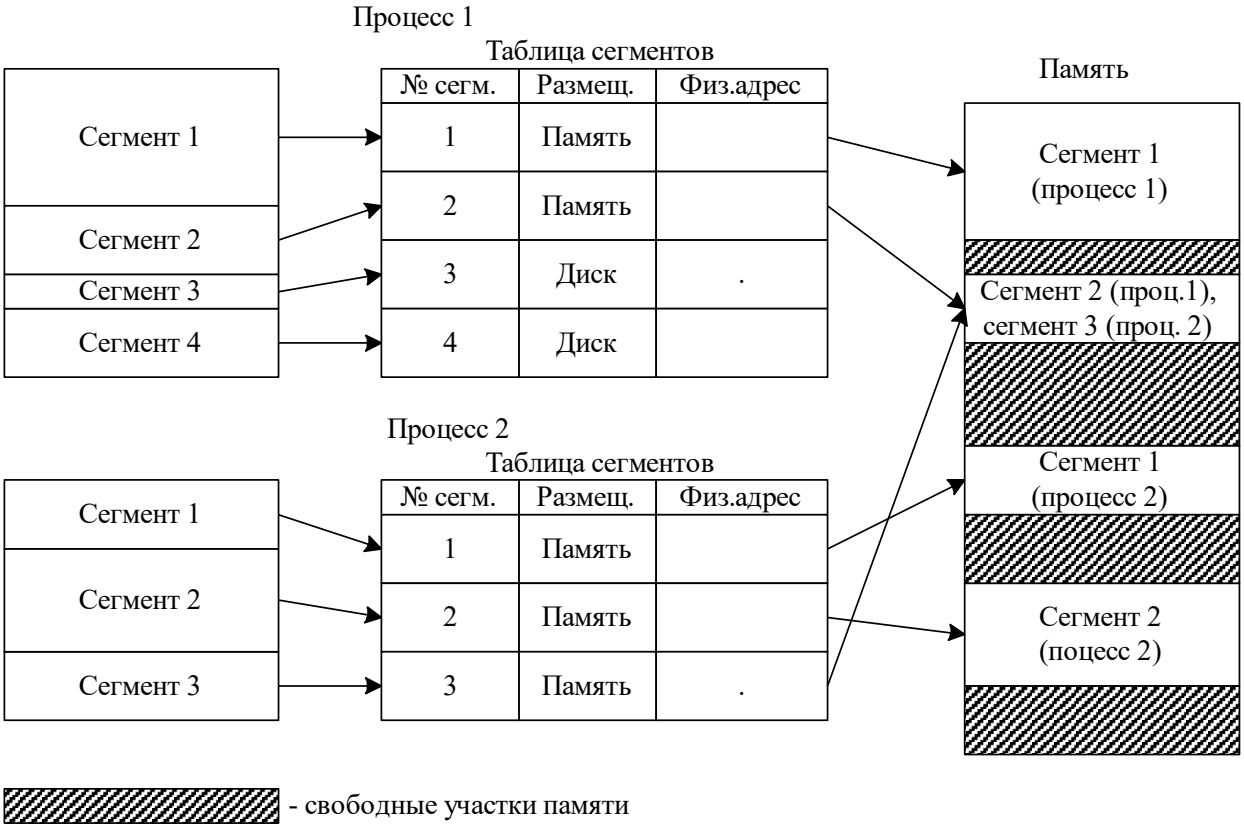


Рис. 1.13. Сегментное распределение памяти

При запуске процесса используемая часть сегментов помещается в оперативную память. Для каждого из сегментов, операционная система выбирает подходящий по размеру свободный участок памяти. Размер сегмента определяется программистом при написании программы (по умолчанию устанавливается компилятором). При этом возможен одновременный доступ нескольких процессов к одному сегменту, например к общей функции или сегменту данных.

Работа с сегментной организацией памяти – формирование адреса выгрузка/загрузка с диска аналогична используемой при страничной организации памяти, однако кроме этого проверяются атрибут доступа к сегменту.

Странично–сегментное распределение памяти

Странично–сегментное распределение памяти предусматривает сочетание изложенных выше подходов. Каждый определенный программистом сегмент, в этом случае состоит из группы страниц. Механизм формирования адреса в сранично–сегментном способе распределения памяти представлен на рис. 1.14.

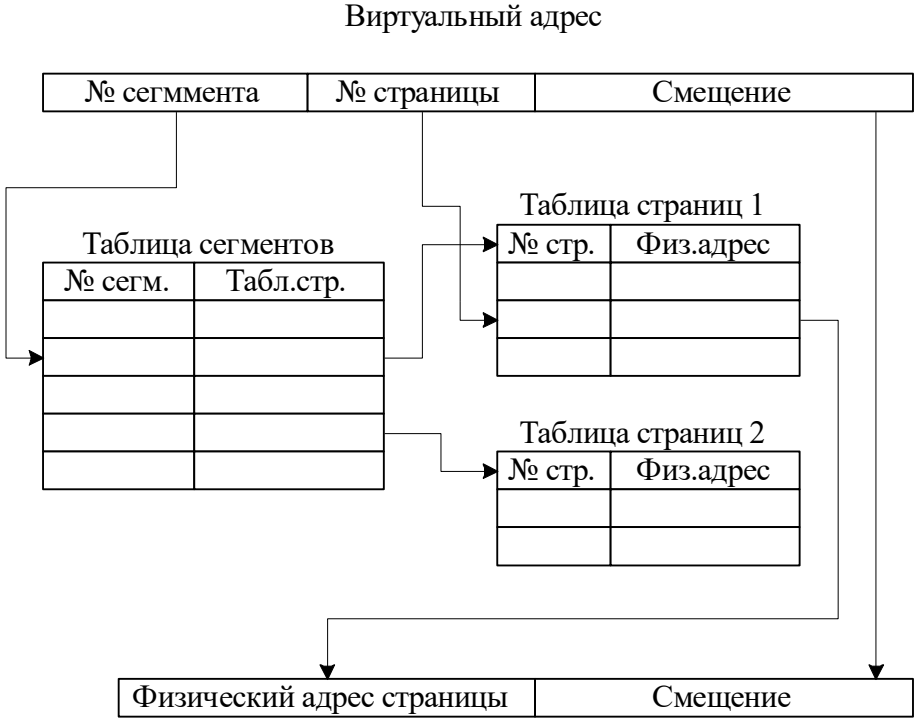


Рис. 1.14. Странично–сегментное распределение памяти

При странично–сегментное распределение памяти операционная система по номеру сегмента находит таблицу страниц этого сегмента. Номер страницы в виртуальном адресе позволяет сформировать физический адрес начала страницы в памяти компьютера. Смещение из виртуального адреса передается в младшие разряды физического адреса без изменений.

Иерархия запоминающих устройств

Иерархия запоминающих устройств приведена на рис. 1.15.

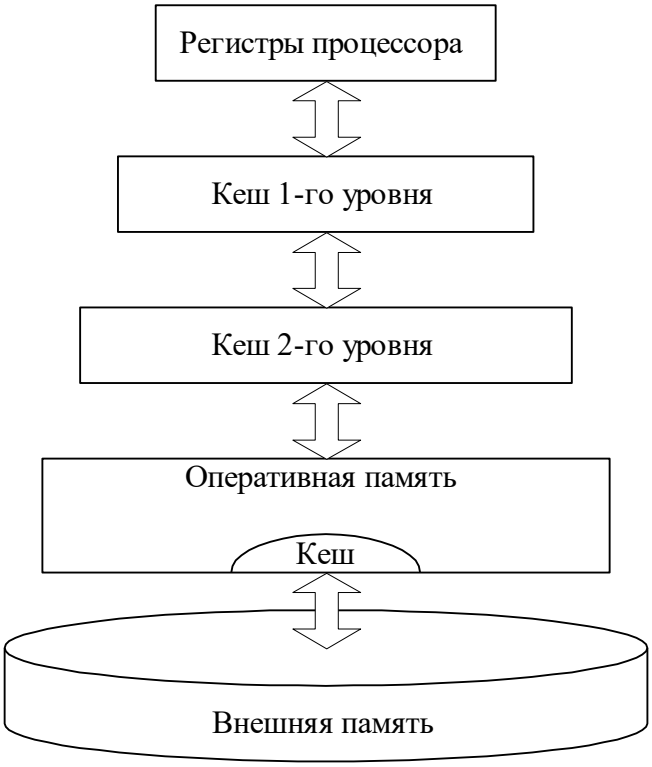


Рис. 1.15. Иерархия запоминающих устройств

Самым быстродействующим устройством памяти компьютера являются регистры процессора. В 16 пользовательских и 16 системных регистрах процессор выполняет все действия, которые реализуются на компьютере. Кеш память 1-го уровня располагается в процессоре и работает на полной частоте ядра процессора. Наличие кэш памяти – сверх быстрой оперативной памяти, позволяет процессору перегрузить данные в кэш и обработать их на высокой скорости. В современных процессорах разделяют внутренние КЭШи для данных, инструкций и команд. В виду высокой стоимости сверх быстрой оперативной памяти, кэш 1-го уровня стараются делать не большим, и на современных компьютерах используется кэш 2-го уровня, работающая на половинной частоте ядра. Кеш 2-го уровня медленнее, чем кэш 1-го уровня, но значительно превосходит по быстродействию основную память.

Все данные и выполняемые программы размещаются в оперативной памяти. Отдельный участок оперативной памяти выделяется для кэширования внешней памяти – например жестких дисков. Кэш память физических устройств позволяет организовать буфер объемна с устройством. Организация кэш памяти представляется на рис. 1.16.

Физический адрес в памяти	Данные	Служебная информация	
		Бит модификации	Бит обращения

Рис. 1.16. Организация кэш памяти

Данные, часто используемые процессором, перегружаются в кэш, при этом запоминается физический адрес их размещения в основной памяти. Если процессор обращается к ячейке, значение которой перегружено в кэш, то на обращение процессора следует более быстрый ответ от кэша, аннулирующий запрос к основной памяти. При обращении к кэшированным данным устанавливаются биты модификации и обращения. Периодически процессор проверяет содержимое кэш памяти и выгружает те ячейки, к которым не было обращения длительное время или, в случае дефицита ресурсов, обращение производится реже.

Средства аппаратной поддержки управления памятью в процессорах Intel 80386 и выше

Процессоры i386 и выше имеет два режима работы – реальный и защищенный. В реальном режиме процессор работает так же как 8086. В защищенном режиме процессор может использовать 32-х разрядный режим доступа к памяти, в том числе механизмы поддержки виртуальной памяти и механизмы переключения задач. В защищенном режиме процессор для каждой задачи процессор может эмулировать виртуальные процессоры, имеющие общую память со страничной организацией.

Средства поддержки сегментации памяти

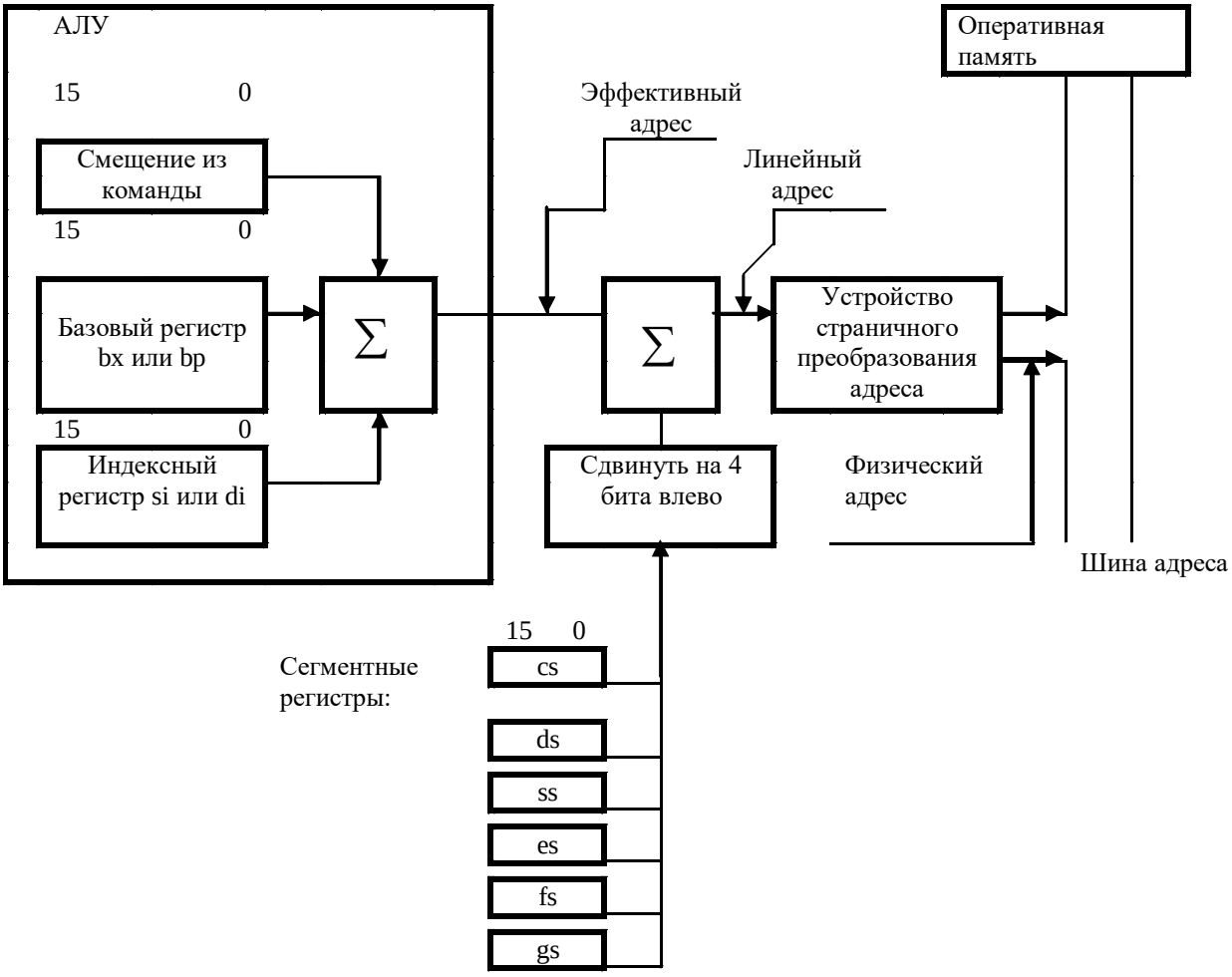


Рис. 1.17. Аппаратная поддержка сегментной организации памяти в процессоре

Адресное пространство процессора i386 с 32-х разрядной шиной адреса составляет 4 Гбайт. Архитектура процессора Intel приведена на рис. 1.17. Виртуальный адрес, состоит из номера сегмента и смещение внутри сегмента. В зависимости от способа адресации используемого в команде смещение может получаться непосредственно из команды, или с помощью косвенной адресации. Номер сегмента выбирается из одного из сегментных регистров процессора: CS, SS, DS, ES, FS или GS.

Средства сегментной организации памяти образуют верхний уровень средств управления виртуальной памятью процессора, а средства страничной организации – нижний уровень. Структура формирования адреса при сегментной организации памяти приведена на рис. 1.18.

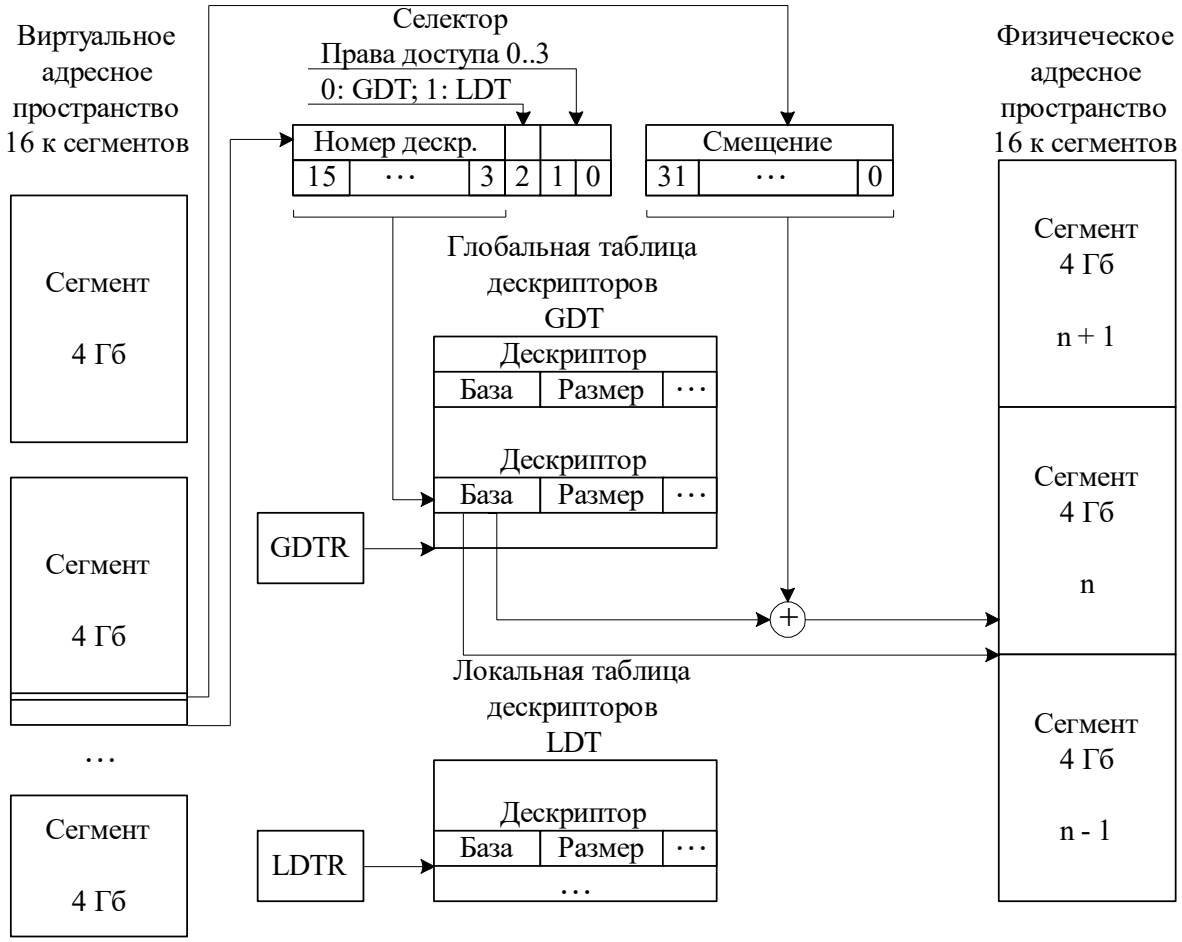


Рис. 1.18. Структурная схема поддержки сегментной организации памяти

Смещение (32 бита) позволяет адресовать $2^{32} = 4\,\text{Гб}$ памяти. Количество сегментов определяется размером поля селектора $2^{13} = 8192$ сегментов. Служебные поля селектора определяют режим доступа (2 бита) и признак таблицы дескрипторов: 0 – GDT, 1 – LDT. Глобальная таблица – GDT, используется операционной системой и для организации межзадачного взаимодействия. Локальная таблица дескрипторов – LDT создается для каждой из задач. База из таблицы дескрипторов указывает на начало сегмента физической памяти. Сумма базы и смещения указывает на адрес операнда в физической памяти. Адреса таблиц хранятся в регистрах GDTR и LDTR.

Странично–сегментный механизм

Работа странично–сегментного механизма в целом повторяет работу страничного режима. Физическое и виртуальное адресные пространства разбиты на страницы размером 4 К.

Линейный виртуальный адрес содержит в своих старших разрядах номер виртуальной страницы, а в младших 12 разрядах смещение внутри страницы. Схема формирования адреса приведена на рис. 1.19.

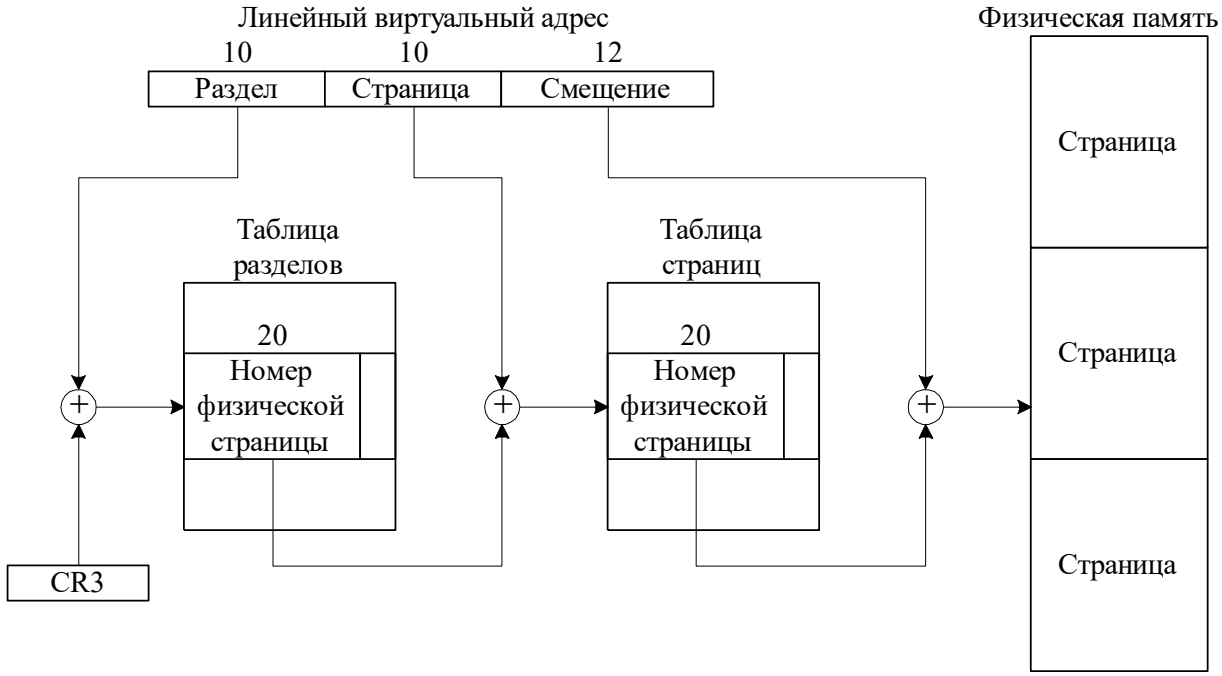


Рис. 1.19. Формирование адреса при странично–сегментном механизме управления памяти

Реализация странично–сегментной организации виртуальной памяти в процессорах Intel позволяет обеспечить защиту переменные и операторы различных процессов за счет:

- изоляции адресных пространств процессов в физической памяти путем назначения им различных физических страниц или сегментов;
- защиту сегментов от несанкционированного доступа с помощью многоуровневой системы привилегий.

Вызов программ

Вызов подпрограмм осуществляется с помощью команд JMP и CALL. Схема вызова подпрограммы приведена на рис. 1.20.

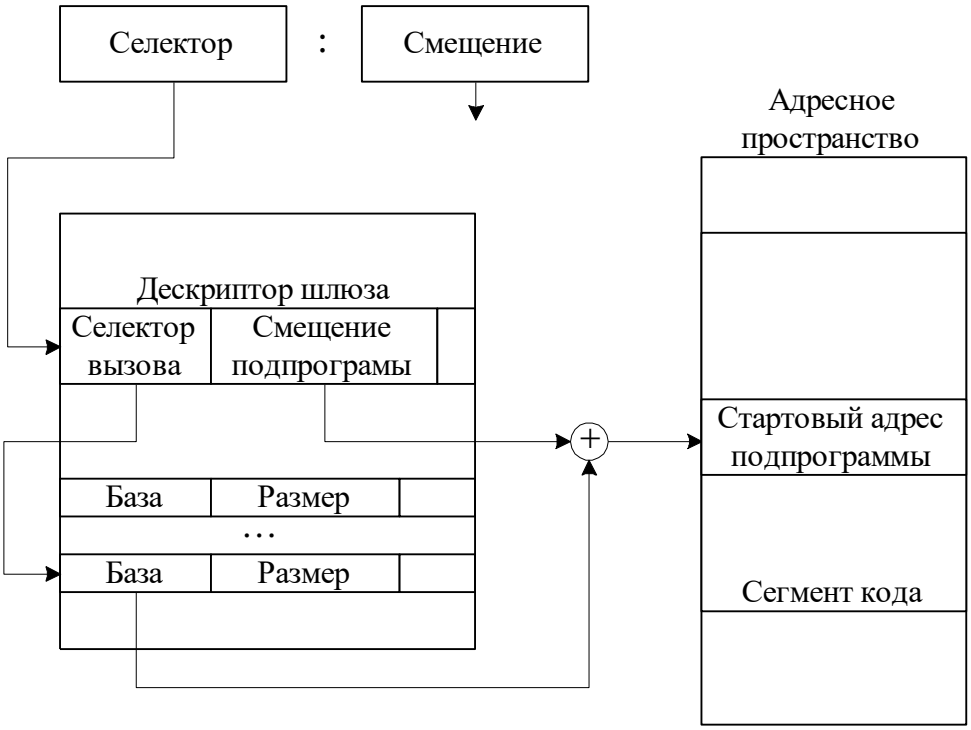


Рис. 1.20. Вызов подпрограммы

Простейший вызов может быть организован с помощью команд JMP или CALL указанием полного адреса (селектор : смещение) вызываемой процедуры. Однако данный способ позволяет вызывать из программы только пользовательские подпрограммы, имеющие 3-й уровень приоритета. Вызов системных функций, имеющих 0-й уровень приоритета, таким способом будет невозможен.

Для решения этой проблемы в процессоре есть другой способ вызова подпрограмм основанный на заранее определенном наборе точек входа в привилегированные кодовые сегменты. Эти точки входа описываются с помощью специальных дескрипторов – шлюзов вызова подпрограмм.

Данный способ вызова показан на рис. 21. Значение селектора определяет дескриптор шлюза в таблице. Из полей дескриптора определяется смещение и ссылка на значение базы, их сумма определяет стартовый адрес подпрограммы.

При определении адреса входа в вызываемом сегменте смещение из поля команды CALL не используется, а используется смещение из дескриптора шлюза, что не дает возможности задаче самой определять точку входа в защищенный кодовый сегмент.

При вызове процедур, с различными уровнями привилегий, возникает проблема передачи параметров между различными стеками, так как для надежной защиты задачи различного уровня привилегий имеют различные сегменты стеков. Селекторы этих сегментов хранятся в контексте задачи – сегменте TSS (Task State Segment).

2. Системное программирование в UNIX

2.1 Работа с файловой структурой UNIX

Операционная система UNIX имеет набор API функций, построенных на основе системных вызовов функций ядра операционной системы. Большинство API функций работает в режиме ядра – защищенном режиме, позволяющему процессу получать доступ к данным ядра операционной системы. После завершения выполнения системного вызова API функции, процесс возвращается в режим пользователя. Переключения контекстов во время вызова системных API функций требует дополнительного времени, по сравнению с вызовом функций библиотеки, но предоставляет значительно больше возможностей для выполнения системных операций.

Большинство функций используют стандарт POSIX.1 и POSIX.1b, в целом аналогичных API функциям UNIX.

Функции API возвращают значение –1 в случае не успешного выполнения операции. Конкретное значение ошибки содержится в глобальной переменной errno описанной в файле errno.h. Тестовую строку, содержащую системное сообщение об ошибке, возвращает функция strerror(). Наиболее часто встречающиеся ошибки приведены в табл. 2.1.

Табл. 2.1 Значение кодов ошибок

errno	Описание ошибки
EACCESS	Процесс не имеет права на выполнение операции.
EPERM	Вызывающий процесс не имеет прав привилегированного пользователя.
ENOENT	Используется недопустимое имя файла.
BADF	Используется недопустимый дескриптор файла.
EINTR	Вызов функции прерван внешним сигналом.
EAGAIN	Один из используемых в процессе выполнения API временно не доступен (занят).
ENOMEM	Ошибка динамического выделения памяти.
EIO	Ошибка ввода/вывода.
EPIPE	Попытка записи в канал, для которого не существует читающего процесса.
EFAULT	Ошибка адреса одного из аргументов.
ENOEXEC	Ошибка запуска процесса.
ECHILD	Процесс не имеет дочерних процессов, завершения которых он мог бы ждать.

Работа с файлами в операционной системе UNIX

Для преставления информации о файле в UNIX используется структура stst описанная в файле sys/stat.h. Структура stst инициализируется и заполняется функцией stat() или fstat().

Структура stat описана следующим образом:

```
    struct stat
    {
      dev_t st_dev ; /* идентификатор файловой системы  */
      ino_t st_ino ;   /* номер индексного дескриптора файла
   */
      mode_t   st_mode ;  /* тип файла и флаги доступа  */
      nlink_t  st_nlink ; /* счетчик жестких ссылок  */
      uid_t st_uid ;   /* UDI владельца файла     */
      gid_t st_gid ;   /* GID владельца файла     */
      dev_t st_rdev ;  /* номера устройств     */
      off_t st_size ;  /* размер файла   */
      time_t   st_atime ; /* время последнего доступа   */
      time_t   st_mtime ; /* время последней модификации   */
      time_t   st_ctime ; /*   время   последнего   изменения
статуса     */
    }
```

Максимальное значение одновременно открытых файлов определяется значением OPEN_MAX (_POSIX_OPEN_MAX), определенных в limits.h.

Порядок действий при открытии файла выглядит следующим образом:

1. Поиск в таблице дескрипторов первой свободной позиции. Если свободная позиция не обнаружена, то возвращается значение –1. Иначе, в качестве дескриптора файла, возвращается номер задействованной позиции в таблице.

2. Поиск незадействованной позиции в таблице файлов. Если свободная позиция не обнаружена, то возвращается значение –1. Если в таблице файлов найдена свободная позиция, то выполняются следующие действия:

- в выделенной записи таблицы дескрипторов делается ссылка на найденную запись в таблице файлов, аналогичная ссылка на номер в таблице файлов делается на номер в таблице дескрипторов;
- в записи таблицы файлов формируется указатель текущей позиции в файле (смещение от начала файла);
- в записи таблицы файлов записывается режим, в котором был открыт файл;
- значение счетчика количества дескрипторов связанных с файлом в таблице файлов устанавливается в 1;
- значение счетчика ссылок индексного дескриптора (указывает, сколько записей таблицы файлов указывают на данный дескриптор) увеличивается на 1.

При закрытии файла выполняются следующие действия:

1. отметить запись в таблице дескрипторов как свободную;

2. уменьшить значение счетчика таблицы файлов на 1, если значение не равно 0, то переход на пункт 6;

3. отметить запись в таблице файлов как свободную;

4. уменьшить значение счетчика в таблице индексных дескрипторов на 1, если значение не равно 0, то переход на пп 6;

5. если значение счетчика жестких ссылок равно 0, то позиция в таблице помечается свободной и освобождается вся физическая память, выделенная для хранения файла;

6. ядро передает управление процессу и возвращает код успешного выполнения.

Работа с файлами реализуются через потоки или через дескриптор файла. Работа через указатели потоков более эффективна при больших объемах последовательных операций чтения/записи в файл. Это является следствием того, что библиотечные функции Си, предназначенные для работы с файлами через потоки ввода/вывода, используют буферизацию.

Работа файлами через дескриптор файла является более эффективным для приложений осуществляющих произвольный доступ к данным из файла.

Однако, на практике для эти два способа оказываются сильно связанны между собой. Функция fopen() для фактического открытия файла обращается к функции open(), работающей через дескрипторы.

Ниже представлены наиболее часто использующиеся функции, описанные в модуле stdio.h:

FILE * fopen(char *NameFile, char *open_mode)

Функция fopen() открывает файл указанной в строке NameFile. Режим открытия файла определяется строкой open_mode. Допустимы следующие режимы открытия файлов:

Таблица 2.2. Режимы открытия файлов

“r”	для чтения	“r+”	для чтения в монопольном режиме
“a”	для добавления в конец файла	“a+”	для добавления в монопольном режиме
“w”	для записи	“w+”	для записи в монопольном режиме

В случае если функции fopen() не удалось открыть файл, то она возвращает значение NULL.

Пример использования функции fopen():

```
FILE *FP ;
FP = fopen(“test.txt”,”r”) ;
if(FP = = NULL)
  {
   printf(“\n Ошибка открытия файла”) ;
   . . . /* Обработка ошибки открытия файла */
  }
```

int fileno(FILE *FP)

Функция fileno() определяет дескриптор файла по указателю на поток. Запись FILE содержит дескриптор открытого файла, которая может быть

получена и использована для выполнения операций чтения/записи в файл через дескриптор.

Например, для рассмотренного выше примера возможно выполнение следующих действий:

```
FILE *FP ;
int handle, A[10] ;
FP = fopen("test.txt","w") ;
if(FP = = NULL)
  {   printf("\n Ошибка открытия файла") ;
       . . . /* Обработка ошибки открытия файла */
  }
handle = fileno(FP) ;
write(handle, A, sizeof(A)) ;
```

FILE * fdopen(int handle, char *open_mode)

Функция fdopen() создает указатель на поток ввода/вывода для файла по номеру дескриптора. В целом функция аналогична функции fopen(), однако файл повторно не открывается, а только создается указатель в таблице потоков для уже существующего значения дескриптора. После выполнения функции программист может работать с файлами одновременно и через дескриптор и через указатель потока.

Например:

```
int handle ;
FILE *FP ;
float r1, r2 ;
handle = open("test.dat", O_BINARY) ;
FP = fdopen(handle, "r") ;
read(handle, &r1, sizeof(float) ;
fscanf(FP,"%f",&r2) ;
```

Работа структурой каталогов

С точки зрения реализации файловой структуры, каталог – это служебный файл с информацией о файлах и подкаталогах. Каждая запись описывается структурой struct direct.

```
struct direct
{
       unsigned short int d_fileno;
        char d_name[14];
}
```

Для работы со структурой каталогов в Си предусмотрен набор функций, в целом реализуемых через системные вызовы open(), read(), write(), lseek(), close(). Функции работы с каталогами определены в direct.h (sys/dir.h в BSD UNIX).

int mkdir(char *name, mode_t mode)

Функция mkdir() создает каталог с указанным именем и атрибутами, которые как и создаваемые файлы маскируются значением umask. В случае не успешного завершения вызова функция возвращает значение –1.

```
Char pathname[] = "/home/user/new_dir";

If(mkdir(pathname, mode) != 0)
{
printf("Error opening directory %s", pathname);
}
```

DIR *opendir(char *name_dir)

Функция opendir() возвращает указатель на поток связанный со служебным файлом каталога. В случае если функции opendir() не сумеет открыть каталог, то возвращается значение NULL.

dirent * readdir(DIR *dir_id)

Функция readdir() выдает указатель на следующий активный элемент каталога. Указатели на неактивные элементы каталога не выдаются. При достижении конца каталога или при выявлении некорректной позиции в каталоге возвращается пустой указатель.

Функция telldir() выдает текущую позицию в указанном потоке каталога.

Функция seekdir() устанавливает позицию для последующей операции readdir() над потоком каталога. Данная позиция совпадает с той, которая была получена в результате выполнения операции telldir(), вычислившей значение loc. Значения, которые возвращает telldir(), корректны только в том случае, если каталог не сжимался и не расширялся. Такая проблема не возникает в случае версии 5, но может возникнуть для некоторых других типов файловых систем.

void rewinddir(DIR *dir_id)

Макрос rewinddir() переустанавливает в начало позицию в указанном потоке каталога. В связи с тем, что rewinddir() реализована в виде макроса, к ней нельзя применить операцию вычисления адреса функции.

int closedir(DIR *dir_id)

Функция closedir() закрывает указанный поток каталога и освобождает структуру DIR.

```
int rmdir(char *name)
```

Функция rmdir() удаляет каталог. **Внимание!** Удаляемый каталог должен быть пустым.

Например, рассмотрим фрагмент программы для поиска в каталоге элемента name:

```
dirp = opendir (".");
while ((dp = readdir (dirp)) != NULL)
if (strcmp (dp->d_name, name) == 0)
  {
      closedir (dirp);
      return FOUND;
   }
 closedir (dirp);
 return NOT_FOUND;
```

Работа с жесткими ссылками

Жесткая ссылка – это путевое имя файла в операционной системе UNIX. Обычно файлы имеют только одну жесткую ссылку – полное имя файла. Новую жесткую ссылку можно создать с помощью команды ln, например:

ln /stud/pim.doc /users/test.dat.

После выполнения этой команды, пользователи могут работать с этим файлом и как с /stud/pim.doc, и как /users/test.dat.

Кроме жестких ссылок, команда ln позволяет создавать и символические ссылки. Разница между жесткими и символическими ссылками представлена в табл. 2.3.

Таблица 2.3. Жесткие и символические ссылки

Жесткая ссылка	Символическая ссылка
Не создает нового индексного дескриптора.	Создает новый индексный дескриптор.
Не позволяет привилегированному пользователю ссылаться на каталоги	Может ссылаться на каталоги.
Не может ссылаться на файлы в других файловых системах	Может ссылаться на файлы в других файловых системах.
Увеличивает значение счетчика жестких ссылок соответствующего индексного дескриптора.	Не изменяет значение счетчика жестких ссылок соответствующего индексного дескриптора.

Функции работы с файлами

int open(char *file_name, int access_mode, mode_t mode_file)

Функция open() открывает файл для работы через дескриптор файла. Прототипы функции описаны в sys/types.h и fcntl.h.

Значения аргумента access_mode приведены в табл. 2.4.

Таблица 2.4 Флаги режима открытия файла

Флаг режима доступа	Описание
O_RDONLY	Открывает файл только для чтения.
O_WRONLY	Открывает файл только для записи.
O_RDWR	Открывает файл и для чтения и для записи.
O_APPEND	Открывает файл для добавления.
O_CREAT	Режим создания файла. Если файл уже существует, то флаг не влияет.
O_EXCL	Используется совместно с флагом O_CREAT для завершения работы функции open() в случае если создаваемый файл уже существует.
O_TRUNC	Открыть файл с усечением.
O_NOBLOCK	Запрещает режим блокировки операций чтения и записи в файл. **Внимание!** Флаг применим только к файлам устройств и FIFO файлам.
O_NOCTTY	Запрещает использовать файл терминального устройства в качестве управляющего терминала вызывающего процесса. **Внимание!** Флаг применим только к файлам терминальных устройств.

Аргумент mode_file функции open() используется только при флаге O_CREAT и задает атрибуты создаваемого файла.

В случае не успешного завершения функция open() возвращает значение –1. Значение дескриптора успешно открытого файла находится в пределах от 0 до OPEN_MAX–1.

Например:

```
int handle ;
handle = open("test.dat", O_BINARY) ;
if( handle == -1)
  {
   printf("\n Ошибка открытия файла") ;
   . . .
   }
```

Атрибуты созываемого файла автоматически маскируются значением umask вызывающего процесса. Значение umask указывает задает те атрибуты файла, которые необходимо исключить. Значение umask передается процессам наследникам, и может быть модифицировано функцией umask().

mode_t umask(mode_t new_mask)

Функции umask() передается новое значение маски, при этом она возвращает старое значение маски.

Например:

```
mode_t  old_mask ;
old_mask = umask( S_IXGRP | S_IWOTH | S_IXOTH ) ;
```

При создании файла функций open() файл будет создан с атрибутами определенными выражением: mode_file & ~umask.

size_t read(int handle, void *buf, size_t size)

Функция read() читает данные из файла, заданного дескриптором handle. Указанное значением size количество байт читается в переменную по адресу buf. Функция read() возвращает количество прочитанных байт.

Например:

```
int A[20], N ;
size_t size_read ;
. . .
read(handle, &N, sizeof(N)) ;
size_read = read(handle, A, sizeof(A)) ;
if( size_read < sizeof(A))
   {
      printf("\n Прочитано меньше байт чем ожидалось ") ;
      . . .
   }
```

В связи тем что size_t определен как int, максимальный размер блока, читаемого из файла не может превышать максимального значения типа int.

Одной из возможных причин, почему функция read() прочитала не полный объем данных, может быть прерывания при поступлении сигнала. В том случае, если выполнение функции read() было прервано сигналом, операционная система (кроме BSD) не перезапустит его автоматически.

size_t write(int handle, void *buf, size_t size)

Функция write() записывает указанное количество байт (size) из переменной расположенной по адресу buf в файл открытый через дескриптор handle. Функция write() возвращает количество успешно записанных байт.

Например:

```
int A[20], N, size_write ;
. . .
write(handle, &N, sizeof(N)) ;
size_write = write(handle, A, sizeof(A)) ;
```

int fcntl(int handle, int cmd, …)

Функция fcntl() позволяет получать и устанавливать флаги управления доступом для файла, заданного дескриптором handle. Например, если файл открыт для чтения, а пользователю необходимо внести в него изменения, или если необходимо организовать работу с файлом через различные дескрипторы.

Значение cmd может быть одним из приведенных в табл. 2.5.

Табл. 2.5. Аргументы функции fcntl()

Значение cmd	Описание
F_GETFL	Возвращает режим открытия файла.
F_SETFL	Изменяет режим открытия файла на указанный.
F_GETFD	Возвращает значение флага close-on-exec.
F_SETFD	Устанавливает или сбрасывает, в зависимости от третьего аргумента (1 или 0) флаг close-on-exec.
F_DUPFD	Создает дубликат дескриптора файла.

Например:

```
int handle, A[20] ;
handle = open("test.dat", O_BINARY) ;
read(handle, A, sizeof(A)) ;
fcntl(handle, F_SETFL, O_RDWR | O_BINARY) ;
write(handle, A, sizeof(A)) ;
```

off_t lseek(int handle, off_t pos, int base)

Функция lseek() позволяет позиционироваться по файлу. Указатель для файла handle перемещается в позицию pos отсчитываемую от значения base. Допустимые значения base приведены в табл. 2.6.

Табл. 2.6. Аргументы функции lseek()

Значение base	Описание
SEEK_SET	Начало файла.
SEEK_CUR	Текущая позиция в файле.
SEEK_END	Конец файла.

Функция lseek() возвращает значение позиции, в которую был перемещен указатель файла.

int link(char *name, char *new_name)

Функция link() создает новую ссылку на существующий файл. Операционная система UNIX не позволяет создавать жесткие ссылки на файлы, находящиеся в других файловых системах. Для создания жесткой ссылки на каталог необходимо иметь права привилегированного пользователя. Если операция выпялилась не успешно, то функция возвращает значение –1.

```
if(link("/path/to/file", "/path/to/link") != 0 )
{
  printf("Error creating link");
}
```

int unlink(char *name)

Функция unlink() удаляет жесткую ссылку на файл. При этом если ссылка являлась последней (счетчик жестких ссылок стал равным 0), то удаляется и сам файл. Если вызов функции закончился не успешно, то возвращается значение –1.

Функция unlink() может работать с каталогами, только в случае если процесс обладает правами привилегированного пользователя.

```
If(unlink("/file/to/delete") != 0)
{
  printf("Error");
}
```

int rename(char *old_name, char *new_name)

Функция rename() переименовывает файл. При этом требуется, что бы файлы находились в одной файловой системе. В случае не успешного выполнения функция возвращает значение –1.

```
If(rename("/path/to/file/oldname","/path/to/file/newname")!= 0)
{
   printf("Error");
}
```

int stat(char *name, struct stat *atrib)

int fstat(int handle, struct stat *atrib)

Функции stat() и fstat() позволяют получить атрибуты файла. Для функции stat() файл передается именем, а для функции fstat() – дескриптором файла. В случае если у процесса не доступа к файлу, то функции возвращают значение –1. В случае успешного выполнения функции заполняют поля структуры struct stat, которая описана следующим образом:

```
struct stat
{
  dev_ts   t_dev ;  /* идентификатор файловой системы   */
  ino_t st_ino ; /* номер индексного дескриптора файла  */
  mode_t   st_mode ;   /* тип файла и флаги доступа  */
  nlink_t  st_nlink ;  /* счетчик жестких ссылок  */
  uid_t st_uid ;    /* идентификатор владельца файла */
  gid_t st_gid ;    /* идентификатор группы    */
  dev_t st_rdev ;   /* номер устройства  */
  off_t st_size ;   /* размер файла в байтах   */
  time_t   st_atime ;  /* время последнего доступа   */
  time_t   st_mtime ;  /* время последней модификации   */
  time _t  st_ctime ;  /*   время   последнего   изменения
статуса  */
}
```

```
stat attrib;
if(stat("filename", &attrib) != 0)
{
  printf("Error");
}
printf("Filesize %d bytes", attrib.st_size);
```

int lstat(char *name, struct stat *atrib)

Функция полностью аналогична функции stat() но работает не только с фалами, но и с символическими ссылками (выдает данные не по файлу, на который ссылается ссылка, по самой ссылки).

int access(char *name, int flag)

Функция access() проверяет наличие файла и прав доступа на этот файл у пользователя. Значение параметра flag определяет тип запроса к файлу. В случае положительного ответа, функция возвращает значение 0, иначе значение –1. Возможные значения параметра flag приведены в табл. 2.7.

Табл. 2.7. Запросы к атрибутам файла

Значение flag	Запрос к файлу
F_OK	Проверить существует ли файл.
R_OK	Есть ли у вызывающего процесса право на чтение.
W_OK	Есть ли у вызывающего процесса право на запись.
X_OK	Есть ли у вызывающего процесса право на выполнение.

Значение flag может быть получено путем побитового сложения отдельных атрибутов, например: flag = F_OK | R_OK.

```
If(access("filename", X_OK) != 0)
{
  printf("You cannot execute this file");
}
```

int close(int handle)

Функция close() закрывает файл, заданный дескриптором handle. В случае не удачного завершения вызова, возвращается значение –1, а переменная errno устанавливается в код ошибки. По умолчанию ядро операционной системы UNIX при завершении процесса закрывает все открытые им файлы.

```
If(close(fp) != 0)
{
  printf("Error closing file");
}
```

int fcntl(int handle, int cmd_flag, struct flock *f_flock)

Функция fcntl() позволяет блокировать файл, заданный дескриптором handle. Операционная система UNIX позволяет организовывать доступ к файлу со стороны многих процессов. При этом может иметь место попытки работы с одним и тем же участком фала. Решение этой ситуации возможно за счет блокировки файлов, или областей файлов.

Конкретный режим блокировки задается значением cmd_flag, приведенные в табл. 2.8.

Табл. 2.8. Режимы блокировки файлов

Значение cmd_flag	Назначение
F_SETLK	Блокировать файл.
F_SETLKW	Блокирует файл и вызывающий процесс на время пока действует блокировка на файл.
F_GETLK	Дает запрос, о том какой процесс заблокировал файл.

Указание, какую область файла необходимо блокировать, заносится в поля структуры struct flock, описанной следующим образом:

```
struct flock
   {
   short    l_type ;  /* тип блокировки   */    short
   l_whence ;   /* адрес следующего поля  */ off_t    l_start  ;
    /* адрес начала блокировки   */
   off_t    l_len ;   /* размер заблокированной области */
   pid_t    l_pid ;   /*  идентификатор  блокирующего  процесса
   */
   }
```

Тип блокировки задается значением поля l_type, значения которого приведены в табл. 2.9.

Табл.2.9.Значения поля l_type

Значение l_type	Действие
F_RDLCK	Установить блокировку по чтению.
F_WRLCK	Установить блокировку по записи.
F_UNLCK	Снять блокировку указанной области.

Если установить значение поля l_len равном 0, то будет заблокирована вся область до конца файла.

Внимание! Все установленные процессом блокировки снимаются при его завершении.

```
Int owner_id = fcntl(fp, F_GETOWN);
If(owner_id == -1)
{
  printf("No owner?");
}
```

2.2 Процессы в операционной системе UNIX

Процесс – это программа, находящаяся в стадии выполнения. Процесс в ОС UNIX создается с помощью системного вызова fork(). Создаваемый процесс получает копи сегментов стека, данных, кода и таблицу дескрипторов родительского процесса.

Родительский процесс может приостановить свое выполнение с помощью функций wait() и waitpid() до завершения процесса наследника. В случае независимого продолжения процесса родителя и наследника родительский процесс может получать информацию о состоянии процесса наследника с помощью функций signal() и sigaction().

Завершение выполнения процесса выполняется с помощью функции _exit(). Традиционно считается, что нулевой код завершения процесса является успешным.

Вызов exec() позволяет запустить другую программу, вместо вызывающего процесса. В результате данного вызова идентификатор процесса и другая служебная информация, включая дескрипторы открытых каталогов и файлов (кроме тех у которых установлен флаг close-on-exec – закрыть по завершении процесса).

pid_t fork(void)

Функция fork() создает дочерний процесс. В случае успешного выполнения возвращается идентификатор созданного процесса. В случае не успешного выполнения возвращается значение –1 и устанавливается значение переменной errno. Текстовый вариант сообщения об ошибке может быть получено с помощью функции prror().

```
Pid_t pid_id = fork();
If(pid_id == -1)
{
  printf("Error creating process");
}
```

void _exit(int exit_code)

Функция _exit() завершает выполнение процесса, при этом освобождаются все сегменты процесса и закрываются все дескрипторы открытых файлов.

Соответствующая завершаемому процессу запись в таблице процессов сохраняется для предоставлении возможности родительскому процессу получить код завершения процесса.

В качестве кода завершения процесса используются значения от 0 до 255.

pid_t wait(int *status_p)
pid_t waitpid(pit_t child_pid, int *staus_p, int option)

Функции wait() и waitpid() приостанавливают выполнение родительского процесса до завершения вызванного дочернего процесса. Данные вызовы позволяют получить код завершения процесса и освободить запись о процессе в таблице процессов.

Функция wait() приостанавливает выполнение родительского процесса до получения сигнала или завершения одного из дочерних процессов. В случае завершения дочернего процесса функция wait() возвращает код завершения процесса. Если в момент вызова функции wait() родительский процесс не имеет ни одного созданного им дочернего процесса, то возвращается значение –1.

Функция waitpid() позволяет получить более полную информацию о завершаемом процесса. В зависимости от значения аргумента child_pid, можно уточнить, завершения какого из дочерних процессов ждет родительский процесс. Возможные значения child_pid приведены в табл. 2.10.

Табл. 2.10. Аргументы функции waitpid()

Значение child_pid	Завершения какого дочернего процесса ждет родительский процесс
ID дочернего процесса	Завершения дочернего процесса с идентификатором равным указанному значению.
–1	Завершения любого дочернего процесса.
0	Завершения любого дочернего процесса, принадлежащего к одной группе с родительским процессом.
Отрицательное значение (кроме –1)	Завершения любого дочернего процесса, идентификатор группы которого совпадает с абсолютным значением child_pid.

Значение аргумента opion позволяет задать режимы управления заданиями и блокировки. В переменную status_p, передаваемую через указатель записывается код завершения процесса устанавливаемый в функции _exit() дочернего процесса. В случае если код завершения контролировать не требуется, то передается значение NULL.

int execl(char *name, char *arg,)
int execlp(char *name, char *arg)
int execle(char *name, char *arg, char **env)
int execv(char *name, char **arg)
int execvp(char *name, char **arg)
int execve(char *name, char **arg, char **env)

Функции exec() позволяют запустить на выполнение новый процесс. При этом новый процесс заменяет в памяти сегменты кода, данных и стека

вызывающего процесса, но не создает новую запись в таблице процессов. Функции имеют незначительные отличия по особенностям передачи парламентов. Аргумент *name соответствует имени вызываемого файла. Если имя файла не начинается с символа ‘\’, то некоторые из приведенных функций позволяют использовать значение переменной окружения среды PATH. Аргументы запуска программы arg позволяют задать второй аргументы функции main(int argc, char **arg [, char **evn]) вызываемой программы. В зависимости от конкретной функции аргумент передается либо в виде одной строки символов(отдельные аргументы разделяются пробелом, как в командной строке), либо в виде массива строк, заканчивающихся значением NULL.

Аргумент **env позволяет передать параметры окружения среды UNIX в вызываемый процесс.

Использование аргументов функции main() позволяет передать в вызываемый процесс параметры из родительского процесса.

int system(char *cmd)

Функция system() вызывает команду, или запускает файл, указанный в аргументе cmd. В случае если необходимо вызвать несколько команд за одно обращение к функции system(), они пишутся через точку с запятой.

Функция system() вызывает fork() для создания процесса. Порожденный процесс в свою очередь вызывает execlp() и передает аргументы процессу.

Функция system() ожидает получение информации от waitpid() сигнала о том что порожденный процесс завершился вызовом _exit(). В случае неуспешного завершения порожденного процесса функция system() возвращает значение –1.

int pipe(int fifo[2])

Функция pipe() создает коммуникационный канал между двумя взаимосвязанными процессами. При этом создается служебный файл канала, который автоматически удаляется при закрытии файловых дескрипторов использующихся для работы с каналом.

В большинстве версий UNIX канал может быть только однонаправленным. Дескриптор fifo[0] используется для записи данных в канал, а дескриптор fifo[1] для чтения данных из канала связи. Использование двух направленных каналов связи, допустимое в отдельных версиях UNIX, обычно не рекомендуется, так как приводит к непереносимости программы. Так же не рекомендуется работать через один канал связи более чем с двумя процессами, так как это может привести к неоднозначным результатам.

В отношении канала связи не допустимо использование функции lseek(). Выпорка данных производится только последовательно. Физически данные

удаляются из канала сразу после прочтения. Максимальный размер буфера, выделяемого для канала связи определяется значением PIPE_BUF.

Для передачи значений fifo[] процессу наследнику, или нескольким наследникам обычно функция pipe() вызывается до создания процессов. В этом случае, процесс наследник, созданный с помощью функции fork(), имеет копию дескрипторов родительского процесса.

При передаче данных по каналу связи, принимающая сторона может проверить отсутствие данных в канале связи по значению, возвращаемому функцией read() – функция вернула меньшее число байт, чем Вы заказали прочитать.

Если принимающая сторона, закрыла канал связи, то передающему процессу будет направлено сообщение о том, что канал разорван – SIGPIPE.

В случае успешного завершения вызова функция pipe() возвращает значение 0, иначе, значение –1 и устанавливает значение переменой errno в код ошибки.

pid_t getpid(void)
pid_t getppid(void)
pid_t getpgrp(void)
pid_t getuid(void)
pid_t geteuid(void)
pid_t getgid(void)
pid_t getegid(void)

Приведенный выше набор функций, позволяет получить атрибуты процесса.

Назначение функций приведено в табл. 2.11

Табл. 2.11. Функции получения атрибутов процессов

Функция	Назначение
pid_t getpid(void)	Получить идентификатор текущего процесса.
pid_t getppid(void)	Получить идентификатор родительского процесса.
pid_t getpgrp(void)	Получить идентификатор группы вызывающего процесса.
pid_t getuid(void)	Получить реальный идентификатор владельца вызывающего процесса.
pid_t geteuid(void)	Получить реальный идентификатор группы вызывающего процесса.
pid_t getgid(void)	Значение атрибута eUID вызывающего процесса (используются для определения прав на доступ к файлам).
pid_t getegid(void)	Значение атрибута eGID вызывающего процесса (используются для определения прав на доступ к файлам).

Например, рассмотрим программу вывода информации о процессе:

```
#include <stdio.h>
#include <sys/types.h>
#include <uinstd.h>
void main(void)
{
  printf("\n идентификатор текущего процесса: %d",(int) getpid() ) ;
  printf("\n идентификатор родительского процесса: %d",(int) getppid() ) ;
  printf("\n идентификатор группы вызыв. процесса: %d",(int) getpgrp() ) ;
  printf("\n реальный идентификатор владельца: %d",(int) getuid() ) ;
  printf("\n реальный идентификатор группы: %d",(int) getgid() ) ;
  printf("\n значение eUID: %d",(int) geteuid() ) ;
  printf("\n значение eGID: %d",(int) getegid() ) ;
}
```

int setsid(void)
int setpgid(pid_t pid, pid_t pgid)
int setuid(uid_t uid)
int seteuid(uid_t uid)
int setgid(gid_t gid)
int setegid(gid_t gid)

Приведенный выше набор функций позволяет устанавливать отдельные атрибуты процессов. Некоторые атрибуты процессов изменять не допускается, например: идентификаторы самого процесса и родительского процесса. Назначение функций приведено в табл. 2.12.

Табл. 2.12. Функции установки атрибутов процессов

Функция	Назначение
int setsid(void)	Отсоединение процесса. В этом случае процесс, работает не зависимо от родительского процесса.
int setpgid(pid_t pid, pid_t pgid)	Делает процесс лидером новой группы процессов.
int setuid(uid_t uid)	Измерить реальный и эффективный UID на указанный.
int seteuid(uid_t uid)	Изменить эффективный идентификатор владельца процесса.
int setgid(gid_t gid)	Изменяет значение eUID вызывающего процесса.
int setegid(gid_t gid)	Изменяет эффективный идентификатор группы вызывающего процесса.

В случае успешного завершения вызова функция pipe() возвращает значение 0, иначе, значение –1 и устанавливает значение переменой errno в код ошибки.

```
    /* Пример коммуникации процессов при помощи программных
каналов
     * (трубы, pipes).
     *      Данная программа превращается в две программы,
     *      соединенные трубами в таком порядке:
     *
     *         stdout                  stdin
     *        /------------ PIP1 -----------> cmd2
     *      cmd1 <----------PIP2---------------/
     *         stdin                   stdout
     */
    /* файл LOOP_strt.c */
    #include <stdio.h>
    #define eq(s1,s2) ( strcmp(s1,s2) == 0 ) /* истина, если
строки равны */
    #define SEP          "---"                  /* разделитель
команд при наборе */
    main( c, v ) char **v;
    {
      char **p, **q;
      int pid;
      int PIP1[2];    /* труба cmd1-->cmd2 */
      int PIP2[2];    /* труба cmd2-->cmd1 */
      if( c==1 ){
         printf( "Call: strt cmd1... %s cmd2...\n", SEP );
         exit(1);
      }
            /* разбор аргументов */
      v++;
      /* в p - аргументы первой команды */
      p = v;
      while( *v && !eq( *v, SEP ))
         v++;
      *v = NULL;
      v++;
      /* в q - аргументы второй команды */
      q = v;
      pipe( PIP1 );   /* создаем две трубы */
      pipe( PIP2 );   /* PIP[0] - открыт на чтение, PIP[1] - на
запись */
      if( pid = fork()){      /* развилка: порождаем процесс */
      /* ПОРОЖДЕННЫЙ ПРОЦЕСС */
         fprintf( stderr, "сын=%s pid=%d\n", p[0], getpid());
         /* перенаправляем stdout нового процесса в PIP1 */
         dup2( PIP1[1], 1 );
         close( PIP1[1] );
         /* канал чтения мы не будем использовать */
            close( PIP1[0] );
         /* перенаправляем stdin из PIP2 */
         dup2( PIP2[0], 0 );
```

```
    close( PIP2[0] );
    /* канал записи мы не будем использовать */
       close( PIP2[1] );
    /* начинаем выполнять программу, содержащуюся в
     * файле p[0] с аргументами p (т.е. cmd1)
     */
    execvp( p[0], p );
    /* возврата из сисвызова exec не бывает */
}else{
/* ПРОЦЕСС-РОДИТЕЛЬ */
    fprintf( stderr, "отец=%s pid=%d\n", q[0], getpid());
    /* перенаправляем stdout в PIP2 */
    dup2( PIP2[1], 1 );
    close( PIP2[1] ); close( PIP2[0] );
    /* перенаправляем stdin из PIP1 */
    dup2( PIP1[0], 0 );
    close( PIP1[0] ); close( PIP1[1] );
    /* запускаем cmd2 */
    execvp( q[0], q );
}
```

2.3 Сигналы в операционной системе UNIX

Сигналы инициализируются событиями и посылаются для уведомления о произошедшем событии в процесс. Источникам сигналов могут выступать события в аппаратуре компьютера или в операционной системе. Так же возможен обмен сигналами между процессами. Сигналы в UNIX являются программным аналогом системы прерываний компьютера.

Список сигналов приведен в табл. 2.13. Полный список сигналов используемых в операционной системе UNIX содержится в файле signal.h.

Табл. 2.13. Сигналы в операционной системе UNIX

Сигнал	Событие	Файл core
SIGALRM	Наступление тайм–аута таймера сигналов. Может генерироваться функцией alarm().	Нет
SIGABRT	Аварийное завершение процесса. Генерируется функцией abort().	Да
SIGFPE	Недопустимая математическая операция.	Да
SIGHUP	Разрыв связи с управляющим терминалом.	Нет
SIGILL	Недопустимая машинная команда.	Да
SIGINT	Прерывание процесса, обычно генерируется нажатием Ctrl+C.	Да
SIGKILL	Уничтожение процесса. Может генерироваться с помощью команды kill.	Да
SIGPIPE	Недопустимая запись в канал связи.	Да
SIGQUIT	Выход из процесса. Генерируется при нажатии	Да

	клавиш Ctrl+\	
SIGEGV	Ошибка сегментации.	Да
SIGTERM	Завершение процесса. Может генерироваться с помощью команды kill.	Да
SIGUSR1	Определяется пользователем.	Нет
SIGUSR2	Определяется пользователем.	Нет
SIGCHID	Посылается в родительский процесс при завершении дочернего процесса.	Нет
SIGCONT	Возобновление остановленного процесса.	Нет
SIGSTOP	Остановка процесса.	Нет
SIGTTIN	Остановка фонового процесса в случае запроса данных от управляющего терминала.	Нет
SIGSTP	Остановка процесса нажатием клавиш Ctrl+Z.	Нет
SIGTTOU	Остановка фонового процесса в случае попытки вывести данные на свой управляющий терминал.	Нет

Пользователь может реализовать собственную обработку вызванного сигнала. Многие сигналы, вызывающие завершение процесса формируют файл core, содержащий образ завершенного процесса. Данный образ позволяет проанализировать наличие смысловой ошибки в программе.

Если в процесс посылается много экземпляров сигнала, то пользовательский обработчик обслуживает только первый из них, а остальные сигналы обрабатывает стандартный обработчик. При этом нет гарантии что процесс перехватит все сигналы, кроме того возможно ситуация когда процесс получит повторный экземпляр сигнала, до завершения обработки первого экземпляра. При этом возникает вариант соревнования обработки различных экземпляров сигнала. Это является одним из главных недостатков операционной системы UNIX.

Функции для работы с сигналами

Функция signal()

Прототип функции signal(), описанной в signal.h, выглядит следующим образом:

void *signal (int signal_num, void (*handler)(int))

Значение signal_num – идентификатор сигнала, возможные значения которых приведены в табл. 2.13. Пользовательский обработчик сигнала указывается через указатель. Аргументом функции является целое число.

Например:

```
#include <signal.h>
#include<stdio.h>
/* Функция обработчик сигнала */
void sign_term(int sig_num)
  {
     . . .
     printf("\n получен сигнал SIGTERM") ;
  }
```

```
void main(void)
  {
      . . .
signal(SIGTERM, sign_term) ;
      . . .
  }
```

В файле signal.h описаны макросы указывающие необходимость игнорировать сигнал или выполнить действие по умолчанию.

#define SIG_DFL void (*) (int) 0

#define SIG_IGN void (*) (int) 1

Макрос SIG_DFL показывает, что необходимо использовать стандартный системный обработчик.

Макрос SIG_IGN указывает сигнал, который необходимо проигнорировать.

Пример использования данных макросов приведен ниже:

signal(SIGINT, SIG_IGN) ;

signal(SIGSEGV, SIG_DFL) ;

Функция signal() возвращает адрес старого обработчика событий. Его значение используется для возвращения системного обработчика события. Например:

```
void *old_handler ;
/*  Устанавливаем режим игнорировать сигнал прерывания процесса по Cntl+C */
old_handler = signal(SIGINT, SIG_IGN) ;
. . .
/* Восстанавливаем стандартный обработчик сигнала */
signal(SIGINT, old_handler) ;
```

Для UNIX System V.3, V4 используется функция sigset() полностью идентичная функции signal(). В последних версиях для задания пользовательских версий появилась функция sigaction().

Сигнальная маска

В операционной системе UNIX можно задать сигнальную маску, блокирующие указанные сигналы для передачи в процесс.

int sigprocmask(int cmd, sigset_t *new_mask, sigset_t *old_mask)

Режим работы функции sigprocmask() задается с помощью аргумента cmd, возможные значения приведены в табл. 2.14.

Табл. 2.14. Аргументы функции sigprocmask()

Аргумент cmd	Выполняемые действия
SIG_SETMASK	Установить новое значение сигнальной маски.
SIG_BLOCK	Добавить сигналы к уже установленной сигнальной маски.
SIG_UNBLOCK	Удалить из сигнальной маски указанные сигналы.

Новое значение сигнальной маски (или изменения в сигнальной маске) содержатся в переменной new_mask, передаваемой в функцию в виде указателя. Старое значение маски сигналов, сохраняется в переменной old_mask, передаваемой так же через указатели.

Структура sigset_t определена в файле signal.h следующим образом:

```
typedef unsigned long sigset_t;
```

В операционной системе UNIX есть целый набор функций для работы переменными масок сигналов.

int sigmptyset(segset_t *new_mask)

Функция sigmptyset() сбрасывает все сигналы, указанные в маске new_mask.

int sigaddset(segset_t *new_mask, int signal_num)

Функция sigaddset() устанавливает в маске new_mask сигнал с номером signal_num.

int sigaddset(segset_t *new_mask, int signal_num)

Функция sigdelset() очищает в маске new_mask сигнал с номером signal_num.

int sigafillset(segset_t *new_mask)

Функция устанавливает все флаги сигналов в указанной маске.

В случае не успешного выполнения все перечисленные выше функции возвращают значение –1, а в случае успешного выполнения – значение 0.

int sigimember(segset_t *new_mask, int sign_num)

Функция возвращает значение 1 если в маске new_mask установлен сигнал с номером sign_num. Если флаг указанного сигнала не установлен, то возвращается значение 0.

int sigpending(sigset_t *mask)

Функция sigpending() определяет наличие сигналов, ожидающих данный процесс. Флаги сигналов записываются в переменную mask. В случае не успешного выполнения функция возвращает значение –1.

Функция посылки сигнала в процесс kill()

int kill(pid_t pid, int signal_num)

Функция kill() посылает сигнал с номером, заданном в переменной sinal_num, в указанный аргументом pid процесс. Возможные значения аргументов приведены в табл. 2.15.

Табл. 2.15. Аргументы функции kill()

Значения pid	Кому передается сигнал
Положительное число	Процессу с указанным значением идентификатора.
0	Сигнал посылается всем процессам, чей GID совпадает с идентификатором группы вызывающего процесса.
–1	Сигнал посылается во все процессы, реальный владельца которых совпадает с эффективным идентификатором вызывающего процесса. В случае если владелец взывающего процесса является привилегированным пользователем, то сигнал посылается во все процессы (кроме ядра системы – значения идентификаторов 0 и 1).
Отрицательное число	Сигнал посылается во все процессы, чей GID совпадает с абсолютным значением аргумента pid.

Функция alarm()

unsigned int alarm(unsigned int time_interval)

Функция alarm() посылает процессу сигнал SIGALRM через указанное число секунд. При вызове, функция возвращает число секунд, оставшиеся до посылки сигнала. Если при вызове функции alarm() установить значение time_interval равно 0, то таймер отключается.

Функция alarm() позволяет приостановить выполнения процесса на указанное время. Однако с помощью нее можно устанавливать время только в секундах, а так же можно работать только с одним таймером. Если этого недостаточно, то можно воспользоваться функциями settimet() и gettimet() позволяющими задавать время в милисекундах и работать сразу с несколькими (до 3-х) таймерами.

int settimet(int which, struct itimerval *val, struct itimerval *old)

int gettimet(int which, struct itimerval *old)

Тип таймера определяется значением аргумента which, возможные значения приведены в табл. 2.16.

Табл. 2.16. Типы таймеров

Значение which	Тип таймера
ITIMER_REAL	Таймер реального времени. По истечении установленного времени посылается сигнал SIGALARM.
ITIIMER_VIRTUAL	Таймер времени, затраченного процессом на задачу пользователя. По истечении времени посылается сигнал SIGVTALARM.
ITIMER_PROF	Таймер общего времени, затраченного процессом на задачу пользователя и на системные задачи. По истечении времени посылается сигнал SIGPROF.

Структура itimerval определена следующим образом:

```
struct itimerval
{
  struct   timeval  it_interval ;  /* интервал таймера  */
  struct   timeval  it_value ;  /* текущее значение таймера
*/
 }
```

Структура timeval описана следующим образом:

```
struct timeval
{
  time_t      tv_sec;
  suseconds_t tv_usec;
 }
```

2.4 Взаимодействие процессов

Функции работы с сообщениями

Операционная система UNIX позволяет организовать взаимодействие нескольких процессов. Организация взаимодействия процессов для различных версий операционной системы UNIX различно.

В адресном пространстве ядра операционной системы создается таблица очередей сообщений. Работа с сообщениями по структуре аналогична работе с файлами.

Функции работы с сообщениями описаны в модуле ipc.h, msg.h и types.h.

Функция msgget()

int msgget(key_t key, int flag)

Функция msgget() открывает очередь сообщения, идентификатор которой задан значением key. Если значение key равно IPC_PRIVATE, то создается очередь сообщений для эксклюзивного использования вызывающего процесса.

Значение аргумента flag указывает на требуемое от функции msgget() действие. Возможные значения аргумента flag приведены в табл. 2.17.

Табл. 2.17. Значения аргумента функции msgget()

Аргумент flag	Действие выполняемое функцией msgget()
0	Возвращает значение дескриптора очереди сообщений с номером key.
IPC_CREAT	Создает новую очередь с указанным идентификатором key. При этом необходимо указывать права доступа, например IPC_CREAT | 0644.
IPC_EXCL	Используется совместно с флагом IPC_CREAT, если пользователю необходимо обязательно создать новую очередь сообщений. При наличии этого флага, если очередь уже существует, то функция возвращает значение –1.

В случае не успешного выполнения функция msgget() возвращает значение –1 и устанавливает соответствующее значение переменой ошибки errno. Возможные сообщения об ошибке при работе с функцией msgget() приведены в табл. 2.18.

Табл. 2.18. Значения ошибок при вызове функции msgget()

Значение errno	Причина возникновения ошибки
ENOSPC	Превышен лимит системы – MSGMNI
ENOENT	Нет очереди с указанном номером key и нет указаний на создание новой очереди.
EEXIST	Установлены флаги IPC_EXCL и IPC_CREAT, а очередь сообщений с номером key уже существует.
EACCESS	У вызывающего процесса нет прав на доступ к данной очереди сообщений.

Функция msgsnd()
int msgsnd(int msgfd, char *msg, int len, int flag)

Сообщение, записанное в поле структуры msg, длинной len байт посылается в очередь сообщений с дескриптором msgfd.

Формат сообщения задается в виде структуры:

```
struct
{
  long  mtype    ;  /* тип сообщения  */
  char  mtext[MSGMAX]  ;  /* буфер для текста сообщения */
}
```

Значение аргумента flag указывает тип вызова – блокирующий или не блокирующий. Рекомендуется использовать деблокирующий вызов функции, для этого необходимо установить значение flag = 0.

В случае не успешного выполнения операции функция msgsnd() возвращает значение –1.

Функция msgrcv()
int msgsnd(int msgfd, void *msg, int len, int mtype, int flag)

Функция msgsnd() принимает сообщение типа mtype из очереди сообщений с дескриптором msgfd. Сообщение записывается в структуру msg, аналогичную описанной выше.

```
struct msg_msg
 {
  struct list_head m_list;
  long  m_type;
  int m_ts;           /* message text size */
  struct msg_msgseg* next;
  void *security;
};
```

Возможные значения типа сообщений mtype приведены в табл. 2.19.

Табл. 2.19. Типы сообщений

Значение mtype	Тип сообщения
0	Самое старое сообщение любого типа из указанной очереди.
Положительное число	Принять самое старое сообщение указанного типа.
Отрицательное число	Принять самое старое сообщение из очереди с минимальным номером, меньшим или равным абсолютному значению аргумента mtype.

Максимальный размер принимаемого сообщения определяется значением аргумента len. Если принимаемое из очереди сообщение, имеет больший размер, то возвращается код не успешного завершения.

Аргумент flag указывает на режим блокировки вызова. Рекомендуется разрешать блокировать процесс, для этого необходимо устанавливать значение flag в 0. Для запрещения блокирования сообщения необходимо указать флаг IPC_NOWAIT.

В случае успешного завершения функция msgsnd() возвращает количество прочитанных из очереди сообщений байт, в случае не успешного завершения возвращается значение –1.

Функция msgctl()

int msgctl(int msgfd, int cmd, struct msqid_ds *mbuf)

Функция позволяет получать и изменять параметры очереди сообщений, заданной дескриптором msgfd. Аргумент cmd указывает на конкретное действие. Возможные варианты приведены в табл. 2.20.

Табл. 2.20. Аргументы функции msgctl()

Значение cmd	Действие
IPC_STAT	Записать параметры очереди сообщений в структуру mbuf.
IPC_SET	Изменить параметры очереди на указанные в структуре mbuf. Для выполнения данной операции пользователь должен иметь права привилегированного пользователя, быть создателем или владельцем очереди.
IPC_RMID	Удалить очередь из системы. Для выполнения данной операции пользователь должен иметь права привилегированного пользователя, быть создателем или владельцем очереди.

Структура struct msqid_ds описана следующим образом:

```
struct msqid_ds
{
     struct ipc_perm msg_perm;
   struct msg *msg_first;     /* первое сообщение в очереди*/
   struct msg *msg_last;      /* последнее сообщение в очереди
*/
   __kernel_time_t msg_stime; /* последнее msgsnd время */
   __kernel_time_t msg_rtime; /* последнее msgrcv время */
```

```
   __kernel_time_t msg_ctime; /* последнее change время*/
   unsigned long  msg_lcbytes;  /*  Reuse  junk  fields  for  32
bit */
   unsigned long  msg_lqbytes;  /* ditto */
   unsigned short msg_cbytes; /* кол-во байт в сообщении */
   unsigned short msg_qnum;  /* кол-во сообщений в очереди */
   unsigned short msg_qbytes; /*  max  number  of  bytes  on  queue
*/
   __kernel_ipc_pid_t msg_lspid; /* pid последнего msgsnd */
   __kernel_ipc_pid_t msg_lrpid; /* последний pid */

}
```

В случае успешного выполнения функция возвращает значение 0, иначе возвращается значение –1.

Пример приложения клиент/сервер

```
/*Функция "эхо", использующая файлы сообщений       */

 /*файл mes.h**************************/
 #include "commun.h"
 #include <sys/ipc.h>
 #include <sys/msg.h>
 #define MKEY1 8001       /*ключ 1: клиент пишет, сервер
                                читает */
 #define MKEY2 8002       /*ключ 2: клиент читает, сер-
                                вер пишет */
 #define PERM 0600        /*разрешение на доступ */
 #define NORMAL 1         /*обычное сообщение */
 #define FIN 2            /*завершающее сообщение */
 /*файл client.c ***************************/
 #include "mes.h"

 clientipc()
 {
   int rid, wid;   /*идентификаторы файлов сообщений */
 /*получение идентификаторов, связанных с ключами */
   wid = msgget((key_t) MKEY1, 0);
   rid = msgget((key_t) MKEY2, 0);
 /*обращение к циклу чтения-записи   */
   client(rid, wid);
   exit(0);
 }

 /*функция приема-передачи   */
 client(rid,  wid)
 int rid;      /*идентификатор файла чтения */
 int wid;      /*идентификатор файла записи */
 {
   struct msgbuf {
```

```
    long mtype;
    char mtext[TAILLEMAXI] } buf;
/*цикл приема-передачи буферов        */
  for (i=0; i<nbuf; i++)  {
    buf.mtype = NORMAL;
    retour = msgsnd(wid, &buf, lbuf, 0);

    retour = msgrcv(rid, &buf, lbuf, 0, 0);
  }
/*посылка nbgf FIN для остановки сервера */
  buf.mtype = FIN;
  retour = msgsnd(wid, &buf, 0, 0);
}

/*файл server.c         ****************************/
#include "mes.h"

serveuripc()
{
  int rid, wid;  /*идентификаторы файлов сообщений */

/*создание файлов сообщений    */
  rid = msgget((key_t) MKEY1, PERM|IPC_CREAT);
  wid = msgget((key_t) MKEY2, PERM|IPC_CREAT);
/*обращение к циклу чтения-записи*/
  serveur(rid, wid);
/*удаление файлов сообщений, поскольку при завершении
  работы процесса они не уничтожаются */
  msgctl(rid, IPC_RMID, 0);
  msgctl(wid, IPC_RMID, 0);
}

/*функция приема-передачи */
serveur(rid,  wid)
int rid;       /*идентификатор файла чтения*/
int wid;       /*идентификатор файла записи*/
{
/*обработка, симметричная по отношению к клиенту */
   .....................................
/*остановка, если оказалось, что тип=FIN */
     if (buf.mtype == FIN) return;
}
```

2.5 Семафоры
Семафоры в операционной системе UNIX

Семафоры в операционной системе UNIX позволяют синхронизировать множество процессов. Семафоры объединяются в группы и хранятся в адресном пространстве ядра операционной системы, в том числе и после завершения создавшего их процесса. В случае удаления семафоров, ядро активизирует все процессы, заблокированы с использованием удаленных семафоров. Функции работы с семафорами и системные переменные определены в sem.h. Имена системных переменных, ограничивающих работу с семафорами приведены в табл. 2.21.

Табл. 2.21. Системные ограничения на работу с семафорами

Имя переменной	Назначение
SEMMNI	Максимальное число наборов семафоров
SEMMNS	Максимальное число семафоров во всех наборах.
SEMMSL	Максимальное число семафоров в одном наборе.
SEMOPM	Максимальное число семафоров в наборе, над котором можно выполнять одновременно действия.

Информация о семафоре хранится в структуре ipc_perm, которая описана следующим образом:

```
struct ipc_perm
{
  __kernel_key_t key;
  __kernel_uid_t uid;
  __kernel_gid_t gid;
  __kernel_uid_t cuid;
  __kernel_gid_t cgid;
  __kernel_mode_t  mode;
  unsigned short seq;
};
```

Функции работы с семафорами

int semget(key_t key, int num_sem, int flag)

Функция semget() открывает или создает набор семафоров с идентификатором key. Функция возвращает дескриптор для работы с набором семафоров.

Возможные аргумента falg представлены в табл. 2.22.

Табл. 2.22. Аргументы semget()

Значение flag	Действия
0	Вернуть значение дескриптора указанного набора семафоров.
IPC_CREAT	Создать новый набор семафоров. Число создаваемых семафоров передается в аргументе num_sem. Значение флага IPC_CREAT побитово складывается с необходимыми правами доступа к набору.
IPC_EXCL	Проверка создания новой очереди сообщений. Если создаваемая очередь уже существует, то возвращает значение –1. Используется совместно с флагом IPC_CREAT.

В случае неуспешного выполнения функция возвращает значение –1.

int semop(int semfd, struct sembuf *ptr, int len)

Функция semop() позволяет менять значение семафора в наборе с указанным дескриптором semfd.

Структура sembuf, содержащая информацию о семафоре и об операции над ним, описана следующим образом:

```
struct sembuf
{
  short    sem_num ;   /* индекс семафора   */
  short    sem_op ;    /* операция над семафором  */
  short    sem_flg ;   /* флаг операции над семафором   */
}
```

Количество структур, на которые указывает указатель prt, передается в аргументе len.

Операция над семафором определяется значением поля sem_op. Возможные значения приведены в табл. 2.23.

Табл. 2.23. Операции над семафорами

Значение поля sem_op	Выполняемая операция
Положительное число X	Увеличить значение указанного семафора на указанное число.
Отрицательное число –X	Уменьшить значение семафора на указанную величину.
0	Проверить равенство значения семафора на 0.

В случае успешного выполнения операции функция semop() возвращает значение 0, а в случае не успешного выполнения возвращается значение –1.

int semctl(int semfd, int num, int cmd, union seun arg)

Функция semctl() позволяет получать и изменять параметры набора семафоров с указанным значением дескриптора semfd и номера семафора num. Возможные операции над семафорами, задаваемые значением аргумента cmd, приведены в табл. 5.4.

Табл. 2.24. Операции над семафорами

Значение cmd	Операция над семафорами
IPC_STAT	Копировать управляющие параметры в буфер arg.
IPC_SET	Установить новые параметры семафоров заданные в буфере arg.
IPC_RMID	Удалить семафор из системы.
GETALL	Скопировать все значения семафоров в массив.
SETALL	Установить значения семафоров на указанные в массиве.
GETVAL	Возвратить значение семафора с номером num.
SETVAL	Установить значение семафора с номером num на указанное в поле val.
GETPID	Возвратить идентификатор процесса, который последним выполнял операцию с семафором с номером num.
GETNCNT	Возразить количество заблокированных процессов, ожидающих увеличения семафора с номером num.
GETZCNT	Возразить количество заблокированных процессов, ожидающих обращение семафора с номером num в ноль.

Для выполнения операций, связанных с изменением значений параметров семафоров процесс должен иметь права привилегированного пользователя, быть создателем или владетелем семафора.

Объединение semun описана следующим образом:

```
  uion semun
  {
    int   val ;    /* значение семафора    */
    struct semid_ds   *buf ;   /*    управляющие    параметры
семафоров   */
    ushort   *array ;    /* массив значений семафоров  */
  }
```

В случае не удачного завершения функция setctl() возвращает значение –1.

Пример взаимодействия процессов с использованием семафоров

```
/*Функция "эхо", использующая семафоры */

 /*файл mem.h              ****************************/
 #include "commun.h"
 #include <sys/ipc.h>
 #include <sys/sem.h>
 #include <sys/shm.h>
 #define MEMKEY 7899      /*ключ, связанный с разделяемой памя-
                            тью */
 #define SEMKEY1 9001     /*ключ, связанный с семафором 1 */
 #define SEMKEY2 9002     /*ключ, связанный с семафором 2 */
 #define PERM 0600        /*разрешение на доступ */

 /*файл client.c           ****************************/
 #include "mem.h"
```

```
clientipc()
{
  int memid;       /*идентификатор разделяемой памяти */
  int semclient, semserveur; /*идентификаторы семафоров */

/*получение идентификаторов, связанных с ключами для
  разделяемой памяти*/
  memid = shmget((key_t) MEMKEY, lbuf, 0);
/*получение идентификаторов, связанных с ключами для
  семафоров  */
  semserveur = semget((key_t) SEMKEY1, 1, 0);
  semclient = semget((key_t) SEMKEY2, 1, 0);
/*обращение к циклу чтения-записи */
  client(memid, semclient, semserveur);
}

/*функция приема-передачи    */
client(memid,  semclient,  semserveur)
int memid;                /*идентификатор разделяемой памяти*/
int semclient;            /*идентификатор семафора */
int semserveur;           /*идентификатор семафора */
{
  char *pbuf;     /*указатель на начало разделяемой памяти */

/*определение адреса разделяемой памяти    */
  pbuf = (char *) shmat(memid, 0, 0);
/*цикл приема-передачи буферов        */
  for (i=0; i<nbuf; i++) {
/*ожидание на семафоре клиента (освобождаемого сервером,
  разрешающим клиенту писать) */
       P(semclient);
/*освобождение семафора сервера (разрешение серверу читать*/
       V(semserveur);
/*при приеме сообщений клиент и сервер меняются ролями */
       P(semclient);
       V(semserveur);
   }
/*для указания серверу на то, что он должен остановиться,
  в первый байт буфера заносится 0 */
   P(semclient);
   *pbuf = 0;
   V(semserveur);
}

/*файл serveur.c         ***************************/
#include "mem.h"

serveuripc()
{
  int memid;                    /*идентификатор памяти */
  int semclient, semserveur;    /*идентификатор семафоров */
```

```
  union semun {
     int val;
     struct semid_ds *buf;
     ushort *array;
  } semctl_arg;     /* структура управления семафором */

/*создание идентификаторов, связанных с ключом для
  разделяемой памяти */
  memid = shmget((key_t) MEMKEY, TAILLEMAXI, PERM|IPC_CREAT);
/*создание идентификаторов, связанных с ключами для
  семафоров       */
  semserveur = semget((key_t) SEMKEY1, 1, PERM|IPC_CREAT);
  semclient = semget((key_t) SEMKEY2, 1, PERM|IPC_CREAT);
/*инициализация семафоров       */
  semctl_arg.val = 0;
  semctl(semserveur, 0, SETVAL, semctl_arg);
  semctl(semclient, 0, SETVAL, semctl_arg);
/*обращение к циклу чтения-записи    */
  serveur(memid, semclient, semserveur);
/*отказ от разделяемой памяти и семафоров        */
  shmctl(memid, IPC_RMID, 0);
  semctl(semserveur, 1, IPC_RMID, 0);
  semctl(semclient, 1, IPC_RMID, 0);
}

/*функция приема-передачи           */
serveur(memid,  semclient,  semserveur)
int memid;          /*идентификатор разделяемой памяти */
int semclient;      /*идентификатор семафора */
int semserveur;     /*идентификатор семафора */
{
/*обработка, симметричная по отношению к клиенту */
   ..........................................
/*выход, если установлен флаг окончания */
    if (*pbuf == 0)  {
       return;
    }
}

/*файл sem.c*/
#include "mem.h"

/*функция, реализующая операции над семафорами */
static void semcall(sid,  op)
int sid;            /*идентификатор    */
int op;             /*операция    */
{
  struct sembuf sb;
  sb.sem_num=0;
  sb.sem_op=op;
  sb.sem_flg=0;
```

```
  semop(sid, &sb, 1);
}

/*установка семафора   */
void P(sid)
int sid;           /*идентификатор     */
{
  semcall(sid, -1);
}

/*сброс (освобождение) семафора */
void V(sid)
int sid;           /*идентификатор     */
{
  semcall(sid, 1);
}
```

2.6 Интерфейс транспортного уровня

Для взаимодействия процессов в рамках компьютерной сети в операционной системе UNIX используются гнезда. Гнезда предоставляют возможность работать с протоколами TCP, UDP и обращаться непосредственно по IP адресу компьютера.

Протокол TCP используется для создания виртуального канала, а UPD – дейтограммам. Виртуальный канал предусматривает установление соединения между гнездами и последовательную передачу данных, что обеспечивает высокую надежность передачи данных. Соединение типа дейтограмма работают быстрее чем виртуальный канал, но при этом не выполняется условие последовательного передачи данных, что снижает надежность передачи данных. Дейтограммное соединение используется в приложениях, где важно быстродействие и физический канал связи обеспечивает высокую надежность передачи данных.

Функции для работы каналами связи

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

Функция socket() для указанного пользователем домена гнездо заданного типа с указанным протоколом.

Аргумент domain и type определяет тип домена и гнезда. Возможные значении аргументов приведены в табл. 2.25.

Табл. 2.25. Аргументы функции socket()

Значение аргумента **domain**	Использование
AF_UNIX	Домены системы UNIX
AF_INET	Internet домены
Значение аргумента **type**	Использование
SOCK_STREAM	Передача сообщений в виде упорядоченного двунаправленного потока батов с предварительным установлением соединения.
SOCK_DGRAM	Взаимодействие с помощью дейтаграмм. Сообщения передаются без установления предварительного соединения.
SOCK_SEQPACKET	Передача сообщений в виде упорядоченного двунаправленного потока батов фиксированной максимальной длины с предварительным установлением соединения.

Значение аргумента protocol позволяет выбрать используемый при соединении протокол. Если значение protocol = 0, то протокол выбирается операционной системой.

В случае успешного выполнения функция socket() возвращает значение дескриптора гнезда. В случае неуспешного выполнения возвращается значение –1.

int bind(int sid, struct sockadr *name, int len)

Функция bind() присваивает символическое имя содержащиеся в структуре name, с указанным размером len, гнезду с дескриптором sid.

Для операционной системы UNIX имя гнезда задается аналогично имени файла. Структура sockadr, для домена UNIX, описана следующим образом:

```
struct sockaddr
{
short    sun_family ;   /* тип операционной системы – AF_UNIX
   */
char  sun_path[] ;   /* символическое имя    */
}
```

В случае домена Internet имя должно состоять из имени компьютера и номера порта. Аргумент name в этом случае имеет следующий формат:

```
    struct sockaddr_in
    {
    short   sin_family ;   /* тип операционной системы –
AF_INET  */
    u_short    sin_port ;  /* номер порта    */
    struct in_addr   sin_addr ;  /* имя хоста удаленного
компьютера  */
    }
```

В случае успешного завершения функция bind() возвращает значение 0, а в случае не успешного выполнения, значение –1.

int listen(int sid, int size)

Функция listen() создает гнездо со стороны серверного процесса для соединений типа виртуальный канал. Функции передаются – дескриптор гнезда – sid, и size – максимальное число запросов на установленное соединение (size ≤ 5).

В случае успешного завершения функция listen() возвращает значение 0, а в случае не успешного выполнения, значение –1.

int connect(int sid, struct sockaddr *name, int len)

Функция connect() вызывается клиентским гнездом для установления связи с серверным гнездом. Дескриптор гнезда задается аргументом sid. Имя серверного гнезда задается аргументом name, имеющим размер len.

В случае успешного завершения функция connect() возвращает значение 0, а в случае не успешного выполнения, значение –1.

int accept(int sid, struct sockaddr *name, int len)

Функция accept() устанавливает соединение серверного гнезда с подключающимся клиентским гнездом. Дескриптор гнезда с символическим именем, записанным в структуре name, размера len, передается в аргументе sid.

В случае успешного завершения функция возвращает дескриптор нового гнезда, предназначенного для эксклюзивного обмена сообщениями между сервером и клиентом. В случае не успешного выполнения функция возвращает значение –1.

int send(int sid, char *buf, int len, int flag)

Функция send() пересылает сообщение, содержащееся в буфере buf, длинной len байт в гнездо заданное дескриптором sid. Значение аргумента falg для обычных сообщений устанавливается в значение 0. Для привилегированных сообщений указывается MSG_OOB.

В случае успешного завершения функция возвращает число байт, записанных в канал связи. В случае не успешного завершения функция возвращает –1.

int recv(int sid, char *buf, int len, int flag)

Функция recv() принимает сообщение из гнезда с указанным дескриптором sid. Сообщение записывается в буфер buf, максимальная длина принимаемого сообщения задается значением len. Аргумент flag определяет тип принимаемого сообщения. Если установлено значение MSG_OOB, то принимаются только привилегированные сообщения. Если flag = 0, то принимаются все сообщения. Если значение flag = MSG_PEEK то прочитанное сообщение остается в очереди.

В случае успешного завершения функция возвращает число байт, прочитанных из канала связи. В случае не успешного завершения функция возвращает –1.

int shutdown(int sid, int mode)

Функция shutdown() завершает работу канала связи, с указанным дескриптором sid. Возможные значения аргумента mode приведены в табл. 2.26.

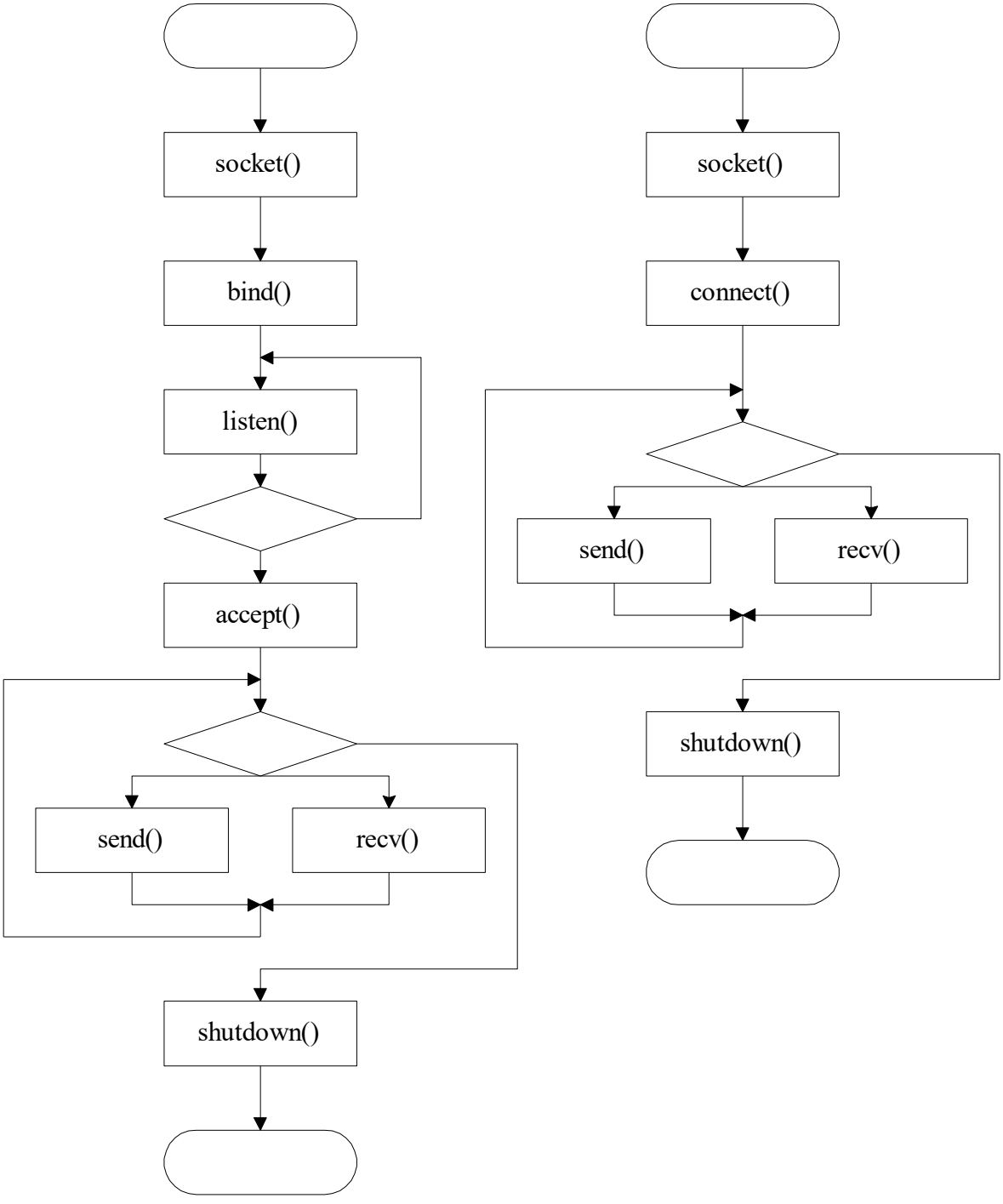
Табл. 2.26. Режимы закрытия канала связи

Аргумент mode	Режим закрытия канала связи
0	Закрыть гнездо для чтения.
1	Закрыть гнездо для записи.
2	Полностью закрыть гнездо для чтения и для записи.

В случае успешного завершения функция shutdown() возвращает значение 0, а в случае не успешного выполнения, значение –1.

Работа с виртуальным каналом связи

Работа с виртуальным каналом связи представлена на рис. 2.1.



а) б)

Рис. 2.1. Работа с виртуальным каналом связи
а) – серверная часть, б) – клиентская часть

Работа с дейтаграммный канал связи

Работа с дейтаграммным каналом связи представлена на рис. 2.2.

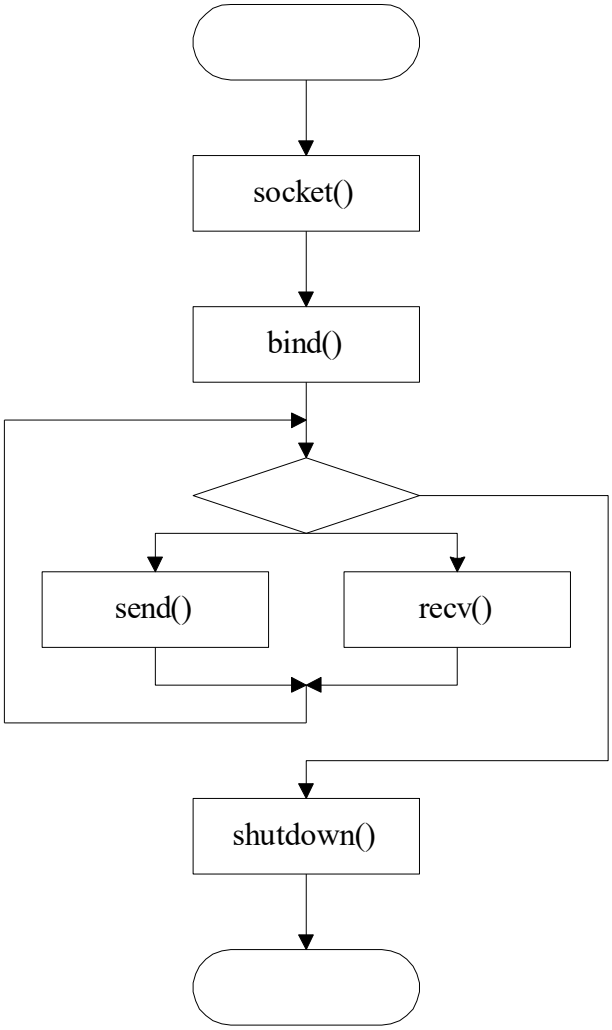


Рис. 2.2. Алгоритм работы процессов через дейтаграмные гнезда

Пример реализации клиент–серверного приложения

Server.cpp

```
#include <errno.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main( int argc, char **argv )
{
  struct sockaddr_in local;
  SOCKET s;
  SOCKET s1;
  int rc;
  char buf[ 1024 ];
```

```
  s = socket( PF_INET, SOCK_STREAM, 0 );

  bzero( &local, sizeof( local ) );
  local.sin_family = AF_INET;
  local.sin_port = htons( 9000 );
  local.sin_addr.s_addr = htonl( INADDR_ANY );
  if ( bind( s, ( struct sockaddr * )&local,
     sizeof( local ) ) < 0 )
     error( 1, errno, "Error" );
  if ( listen( s, NLISTEN ) < 0 )
     error( 1, errno, "listen failed" );
  s1 = accept( s, NULL, NULL );

  for ( ;; )
  {
     rc = recv( s1, buf, sizeof( buf ), 0 );
     if ( rc < 0 )
        error( 1, errno, "recv failed" );
     if ( rc == 0 )
        error( 1, 0, "Client disconnected\n" );
     rc = send( s1, buf, rc, 0 );
     if ( rc < 0 )
        error( 1, errno, "send failed" );
  }
}
```

Client.cpp

```
#include <errno.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main( int argc, char **argv )
{
  struct sockaddr_in peer;
  SOCKET s;
char lin[1024];
int rc;

  bzero( sap, sizeof( *peer ) );
  peer ->sin_family = AF_INET;
peer ->sin_addr.s_addr = htonl( INADDR_ANY );

  peer->sin_port = htons( 9000);

  s = socket( AF_INET, SOCK_STREAM, 0 );

  if ( connect( s, ( struct sockaddr * )&peer,
      sizeof( peer ) ) )
```

```
     error( 1, errno, "connect failed" );

rc = recv( s, lin, sizeof( lin ) - 1, 0 );
if(rc > 0)
printf("%s", lin);
  else
     printf("Error");

  exit( 0 );
}
```

Заключение

Современный уровень развития системного программного обеспечения характеризуются значительным повышением роли компьютерной техники в нашей жизни. Работая с компьютером, в первую очередь мы работаем с виртуальной машиной, посредством которой операционная система предоставляет доступ к аппаратным возможностям компьютера. С каждым годом эта виртуальная машина становится все сложнее, и все больше начинает проникать во все стадии человеческой жизни. Теоретические модели компьютерных систем все больше отрываются от реального современного уровня информационных технологий. Появление виртуальных машин и языков интерпретаторов делает данные не отличимыми от исполнимых модулей.

В условиях сильной зависимости от информационных технологий, во многом уровень нашей информационной безопасности определяется ядром операционной системы. Задача обеспечения информационной безопасности должна строится не только на анализе уже состоявшихся атаках, но в первую очередь на понимании архитектуры, уязвимых местах компьютерных систем и комплексов. Представление об основных принципах и архитектуре организации операционных систем, в первую очередь об операционной системе UNIX, широко использующейся не только в компьютерных системах но в качестве специализированной операционной системы для различного телекоммуникационного оборудования, возможность решать, ряд системных задач под эту операционную систему являются крайне важным для специалиста в области защиты информации.

Литература

1. Дейтел Г. Введение в операционные системы. В 2-х т. М.: Мир 1987.
2. Дунаев С. UNIX сервер. В 2-х т. М. Диалог–МИФИ. 1990.
3. Зегжда Д.П., Ивашко А.М. Как построить защищенную информационную систему. СПб.: НПО «Мир и семья – 95», 1897.
4. Красов А.В. Программирование на Си. Ч1. Основные конструкции языка. СПб. 2001. (или более раннее издание)
5. Красов А.В. Применение системного программирования в прикладных задачах. СПб. 2001.
6. Лоренс Б. Novell NetWare 4.1. СПб.: BHV, 1996.
7. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. СПб.: Питер, 2001 .
8. Чан Т. Системное программирование на Си++ для UNIX. Киев. BHV 1997.
9. Защита информации в ОС Unix: учебное пособие. / А.В. Красов, Д.И. Кириллов, А.Ю. Цветков, И.А. Федянин. – СПб. : Издательство СПбГУТ, 2012.
10. Разработка защищенных приложений на Java: учебное пособие / А.В. Красов, А.С. Верещагин. – СПб. : Издательство СПбГУТ, 2012.

Андрей Владимирович Красов
Александр Юрьевич Цветков

РАЗРАБОТКА ЗАЩИЩЕННЫХ ПРИЛОЖЕНИЯ

УЧЕБНОЕ ПОСОБИЕ

Редактор ***Л.А. Медведева***

План 2013 г., п. ??
Подписано к печати ??
Объем ?? усл. печ. л. Тираж 140 экз. Заказ ??
Издательство СПбГУТ. 191186 СПб., наб. р. Мойки, 61
Отпечатано в СПбГУТ

А. В. Красов
А. Ю. Цветков

РАЗРАБОТКА ЗАЩИЩЕННЫХ ПРИЛОЖЕНИЯ

УЧЕБНОЕ ПОСОБИЕ

Санкт-Петербург
2013