**1**

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

**Операционная система как виртуальная машина**

С помощью простых и ясных абстракций, скрываются от программиста все ненужные подробности организации *прерываний*, работы таймера, управления памятью и т. д. Более того, на современных вычислительных комплексах можно создать иллюзию неограниченного размера оперативной памяти и числа *процессоров*. Всем этим занимается *операционная система*. Таким образом, ***операционная система*** представляется пользователю *виртуальной машиной*, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

**Операционная система как менеджер ресурсов**

Следовательно, ***операционная система***, как *менеджер ресурсов*, осуществляет упорядоченное и контролируемое распределение *процессоров*, памяти и других ресурсов между различными программами.

**Операционная система как защитник пользователей и программ**

Если вычислительная система допускает совместную работу нескольких пользователей, то возникает проблема организации их безопасной деятельности. Всю эту деятельность осуществляет *операционная система* как организатор безопасной работы пользователей и их программ. С такой точки зрения ***операционная система*** представляется системой безопасности государства, на которую возложены полицейские и контрразведывательные функции.

**Операционная система как постоянно функционирующее ядро**

***Операционная система*** – это программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Казалось бы, это абсолютно правильное определение, но, как мы увидим дальше, во многих современных *операционных системах* постоянно работает на компьютере лишь часть *операционной системы*, которую принято называть ее ядром.

**2**

**Основные понятия, концепции ОС**

**Системные вызовы**

***Системные вызовы*** (*system calls*) – это интерфейс между *операционной системой* и пользовательской программой. Они создают, удаляют и используют различные объекты, главные из которых – процессы и файлы. Пользовательская программа запрашивает сервис у *операционной системы*, осуществляя *системный вызов*. Имеются библиотеки процедур, которые загружают *машинные регистры* определенными параметрами и осуществляют *прерывание*   *процессора*, после чего управление передается обработчику данного *вызова*, входящему в ядро *операционной системы*. Цель таких библиотек – сделать *системный вызов* похожим на обычный *вызов* подпрограммы.

Основное отличие состоит в том, что при *системном вызове* задача переходит в привилегированный режим или режим ядра (*kernel mode*). Поэтому *системные вызовы* иногда еще называют программными *прерываниями*, в отличие от аппаратных *прерываний*, которые чаще называют просто *прерываниями*.

В этом режиме работает код ядра *операционной системы*, причем исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро *операционной системы* имеет полный доступ к памяти пользовательской программы, и при *системном вызове* достаточно передать адреса одной или нескольких областей памяти с параметрами *вызова* и адреса одной или нескольких областей памяти для результатов *вызова*.

**Прерывания**

***Прерывание*** (*hardware interrupt*) – это событие, генерируемое внешним (по отношению к *процессору* ) устройством. Посредством аппаратных *прерываний* аппаратура либо информирует центральный *процессор* о том, что произошло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с диска в основную память). Важный тип аппаратных *прерываний* – *прерывания* таймера, которые генерируются периодически через фиксированный промежуток времени. *Прерывания* таймера используются *операционной системой* при планировании процессов. Каждый тип аппаратных *прерываний* имеет собственный номер, однозначно определяющий источник *прерывания*. Аппаратное *прерывание* – это асинхронное событие, то есть оно возникает вне зависимости от того, какой код исполняется *процессором* в данный момент. Обработка аппаратного *прерывания* не должна учитывать, какой процесс является текущим.

**Исключительные ситуации**

***Исключительная ситуация*** (exception) – событие, возникающее в результате попытки выполнения программой команды, которая по каким-то причинам не может быть выполнена до конца. Примерами таких команд могут быть попытки доступа к ресурсу при отсутствии достаточных привилегий или обращения к отсутствующей странице памяти. *Исключительные ситуации*, как и *системные вызовы*, являются синхронными событиями, возникающими в контексте текущей задачи. *Исключительные ситуации* можно разделить на исправимые и неисправимые. К исправимым относятся такие *исключительные ситуации*, как отсутствие нужной информации в оперативной памяти. После устранения причины исправимой *исключительной ситуации* программа может выполняться дальше. Возникновение в процессе работы *операционной системы*исправимых *исключительных ситуаций* считается нормальным явлением. Неисправимые *исключительные ситуации* чаще всего возникают в результате ошибок в программах (например, деление на ноль). Обычно в таких случаях *операционная система* реагирует завершением программы, вызвавшей *исключительную ситуацию*.

**Файлы**

Файлы предназначены для хранения информации на внешних носителях, то есть принято, что информация, записанная, например, на диске, должна находиться внутри файла. Обычно под файлом понимают именованную часть пространства на носителе информации.

Главная задача файловой системы (file system) – скрыть особенности ввода-вывода и дать программисту простую абстрактную модель файлов, независимых от устройств. Для чтения, создания, удаления, записи, открытия и закрытия файлов также имеется обширная категория *системных вызовов* (создание, удаление, открытие, закрытие, чтение и т.д.). Пользователям хорошо знакомы такие связанные с организацией файловой системы понятия, как каталог, текущий каталог, корневой каталог, путь. Для манипулирования этими объектами в *операционной системе* имеются *системные вызовы*.

**3**

**Архитектурные особенности ОС**

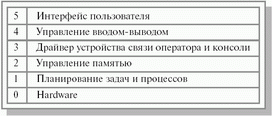
**Монолитное ядро**

По сути дела, *операционная система* – это обычная программа, поэтому было бы логично и организовать ее так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты *операционной системы* являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура *операционной системы* называется *монолитным ядром* (monolithic kernel). *Монолитное ядро* представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, ***монолитное ядро*** – это такая схема *операционной системы*, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной *операционной системы* ядро совпадает со всей системой.

Во многих *операционных системах* с *монолитным ядром* сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается *операционная система*. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция – это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность *операционной системы* в целом.

**Многоуровневые системы (Layered systems)**

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем – интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) и его студентами в 1968 г. Эта система имела следующие уровни:



Слоеные системы хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы, нужно лишь понимать, что они делают. Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев и что к какому слою относится. Слоеные системы менее эффективны, чем монолитные. Так, например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои от верхнего до нижнего.

**Виртуальные машины**

В начале лекции мы говорили о взгляде на *операционную систему* как на *виртуальную машину*, когда пользователю нет необходимости знать детали внутреннего устройства компьютера. Он работает с файлами, а не с магнитными головками и двигателем; он работает с огромной виртуальной, а не ограниченной реальной оперативной памятью; его мало волнует, единственный он на машине пользователь или нет. Рассмотрим несколько иной подход. Пусть *операционная система* реализует ***виртуальную машину*** для каждого пользователя, но не упрощая ему жизнь, а, наоборот, усложняя. Каждая такая *виртуальная машина* предстает перед пользователем как голое железо – копия всего hardware в вычислительной системе, включая *процессор*, привилегированные и непривилегированные команды, устройства ввода-вывода, *прерывания* и т.д. И он остается с этим железом один на один. При попытке обратиться к такому виртуальному железу на уровне привилегированных команд в действительности происходит *системный вызов* реальной *операционной системы*, которая и производит все необходимые действия. Такой подход позволяет каждому пользователю загрузить свою *операционную систему* на *виртуальную машину* и делать с ней все, что душа пожелает.

Недостатком таких *операционных систем* является снижение эффективности *виртуальных машин* по сравнению с реальным компьютером, и, как правило, они очень громоздки. Преимущество же заключается в использовании на одной вычислительной системе программ, написанных для разных *операционных систем*.

**Микроядерная архитектура**

Современная тенденция в разработке *операционных систем* состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом ***микроядерной архитектурой***(*microkernel* architecture) *операционной системы*, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования *процессора*, первичную обработку *прерываний*, операции ввода-вывода и базовое управление памятью.

**Смешанные системы**

Все рассмотренные подходы к построению *операционных систем* имеют свои достоинства и недостатки. В большинстве случаев современные *операционные системы* используют различные комбинации этих подходов. Так, например, ядро *операционной системы* Linux представляет собой монолитную систему с элементами *микроядерной архитектуры*. При компиляции ядра можно разрешить динамическую загрузку и выгрузку очень многих компонентов ядра – так называемых модулей. В момент загрузки модуля его код загружается на уровне системы и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции.

Другим примером смешанного подхода может служить возможность запуска *операционной системы* с *монолитным ядром* под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре *Mach*. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляется *монолитным ядром*. Данный подход сформировался в результате попыток использовать преимущества *микроядерной архитектуры*, сохраняя по возможности хорошо отлаженный код *монолитного ядра*.

Наиболее тесно элементы *микроядерной архитектуры* и элементы *монолитного ядра* переплетены в ядре Windows NT. Хотя Windows NT часто называют микроядерной *операционной системой*, это не совсем так. Микроядро NT слишком велико (более 1 Мбайт), чтобы носить приставку "микро". Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных *операционных системах*. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно *операционным системам* с *монолитным ядром*. По мнению специалистов Microsoft, причина проста: чисто микроядерный дизайн коммерчески невыгоден, поскольку неэффективен.

**4**

**Классификация ОС**

Существует несколько схем классификации *операционных систем*. Ниже приведена классификация по некоторым признакам с точки зрения пользователя.

**Реализация многозадачности**

По числу одновременно выполняемых задач *операционные системы* можно разделить на два класса:

* *многозадачные* (Unix, OS/2, Windows);
* однозадачные (например, MS-DOS).

***Многозадачная ОС***, решая проблемы распределения ресурсов и конкуренции, полностью реализует мультипрограммный режим в соответствии с требованиями раздела "Основные понятия, концепции *ОС* ".

*Многозадачный режим*, который воплощает в себе идею разделения времени, называется вытесняющим (*preemptive*). Каждой программе выделяется квант *процессорного* времени, по истечении которого управление передается другой программе. Говорят, что первая программа будет вытеснена. В вытесняющем режиме работают пользовательские программы большинства коммерческих *ОС*.

В некоторых *ОС* (Windows 3.11, например) пользовательская программа может монополизировать *процессор*, то есть работать в невытесняющем режиме. Как правило, в большинстве систем не подлежит вытеснению код собственно *ОС*. Ответственные программы, в частности задачи реального времени, также не вытесняются. Более подробно об этом рассказано в лекции, посвященной планированию работы *процессора*.

По приведенным примерам можно судить о приблизительности классификации. Так, в *ОС* MS-DOS можно организовать запуск дочерней задачи и наличие в памяти двух и более задач одновременно. Однако эта *ОС* традиционно считается однозадачной, главным образом из-за отсутствия защитных механизмов и коммуникационных возможностей.

**Поддержка многопользовательского режима**

По числу одновременно работающих пользователей *ОС* можно разделить на:

* однопользовательские (MS-DOS, Windows 3.x);
* *многопользовательские* (Windows NT, Unix).

Наиболее существенное отличие между этими *ОС* заключается в наличии у ***многопользовательских систем*** механизмов *защиты персональных данных* каждого пользователя.

**Многопроцессорная обработка**

Вплоть до недавнего времени вычислительные системы имели один центральный *процессор*. В результате требований к повышению производительности появились ***многопроцессорные системы***, состоящие из двух и более *процессоров* общего назначения, осуществляющих параллельное выполнение команд. Поддержка мультипроцессирования является важным свойством *ОС* и приводит к усложнению всех алгоритмов управления ресурсами. Многопроцессорная обработка реализована в таких *ОС*, как Linux, Solaris, Windows NT, и ряде других.

*Многопроцессорные ОС* разделяют на симметричные и асимметричные. В симметричных *ОС* на каждом *процессоре* функционирует одно и то же ядро, и задача может быть выполнена на любом *процессоре*, то есть обработка полностью *децентрализована*. При этом каждому из *процессоров* доступна вся память.

В асимметричных *ОС* *процессоры* неравноправны. Обычно существует главный *процессор* (master) и подчиненные (slave), загрузку и характер работы которых определяет главный *процессор*.

**Системы реального времени**

В разряд *многозадачных ОС*, наряду с *пакетными системами* и *системами разделения времени*, включаются также ***системы реального времени***, не упоминавшиеся до сих пор.

Они используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем они могут поступать, причем от нескольких источников одновременно.

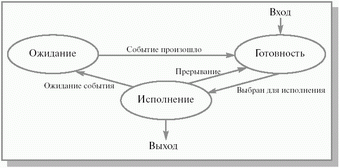
Столь жесткие ограничения сказываются на архитектуре *систем реального времени*, например, в них может отсутствовать виртуальная память, поддержка которой дает непредсказуемые задержки в выполнении программ. (См. также разделы, связанные с планированием процессов и реализацией виртуальной памяти.)

Приведенная классификация *ОС* не является исчерпывающей. Более подробно особенности применения современных *ОС* рассмотрены.

**5**

Понятие ***процесса*** характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы.

**6**



**7**

**Операции над процессами и связанные с ними понятия**

**Набор операций**

*Процесс* не может перейти из одного *состояния* в другое самостоятельно. Изменением *состояния процессов* занимается операционная система, совершая *операции* над ними. Количество таких *операций* в нашей модели пока совпадает с количеством стрелок на диаграмме *состояний*. Удобно объединить их в три пары:

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

В дальнейшем, когда мы будем говорить об алгоритмах планирования, в нашей модели появится еще одна *операция*, не имеющая парной: изменение приоритета *процесса*.

*Операции создания* и *завершения процесса* являются одноразовыми, так как применяются к *процессу* не более одного раза (некоторые системные *процессы* при работе вычислительной системы не завершаются никогда). Все остальные *операции*, связанные с изменением *состояния процессов*, будь то *запуск* или *блокировка*, как правило, являются многоразовыми. Рассмотрим подробнее, как операционная система выполняет *операции над процессами*.

**Process Control Block и контекст процесса**

Для того чтобы операционная система могла выполнять *операции над процессами*, каждый *процесс* представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного *процесса*:

* *состояние*, в котором находится *процесс* ;
* программный счетчик *процесса* или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
* содержимое регистров процессора;
* данные, необходимые для планирования использования процессора и управления памятью (приоритет *процесса*, размер и расположение адресного пространства и т. д.);
* учетные данные (идентификационный номер *процесса*, какой пользователь инициировал его работу, общее время использования процессора данным *процессом* и т. д.);
* сведения об устройствах ввода-вывода, связанных с *процессом* (например, какие устройства закреплены за *процессом*, таблицу открытых файлов).

Ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая *процесс*, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. Для нас это не имеет значения. Для нас важно лишь то, что для любого *процесса*, находящегося в вычислительной системе, вся информация, необходимая для совершения *операций* над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее *PCB* (*Process Control* Block) или *блоком управления процессом*. ***Блок управления процессом*** является моделью *процесса*для операционной системы. Любая *операция*, производимая операционной системой над *процессом*, вызывает определенные изменения в *PCB* . В рамках принятой модели *состояний процессов* содержимое *PCB* между *операциями* остается постоянным.

Информацию, для хранения которой предназначен *блок управления процессом*, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть *регистровым контекстом* *процесса*, а все остальное – *системным контекстом* *процесса*. Знания *регистрового* и *системного контекстов* *процесса* достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним *операции*. Однако этого недостаточно для того, чтобы полностью охарактеризовать *процесс*. Операционную систему не интересует, какими именно вычислениями занимается *процесс*, т. е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства *процесса*, возможно, наряду с *регистровым контекстом* определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве *процесса*, будем называть его *пользовательским контекстом*. Совокупность *регистрового*, *системного* и *пользовательского контекстов* *процесса* для краткости принято называть просто *контекстом процесса*. В любой момент времени *процесс* полностью характеризуется своим *контекстом*.

**Одноразовые операции**

Сложный жизненный путь *процесса* в компьютере начинается с его рождения. Любая операционная система, поддерживающая концепцию *процессов*, должна обладать средствами для их *создания*. В очень простых системах (например, в системах, спроектированных для работы только одного конкретного приложения) все *процессы* могут быть порождены на этапе старта системы. Более сложные операционные системы создают *процессы* динамически, по мере необходимости. Инициатором рождения нового *процесса* после старта операционной системы может выступить либо *процесс* пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый *процесс*. *Процесс*, инициировавший *создание* нового *процесса*, принято называть процессом-родителем (*parent process*), а вновь созданный *процесс* – процессом-ребенком (*child process*). Процессы-дети могут в свою очередь порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев *процессов* – генеалогический лес. Следует отметить, что все пользовательские *процессы* вместе с некоторыми *процессами* операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево.

При рождении *процесса* система заводит новый *PCB* с *состоянием процесса* рождение и начинает его заполнять. Новый *процесс* получает собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера *процесса* в операционной системе отводится ограниченное количество битов, для соблюдения уникальности номеров количество одновременно присутствующих в ней *процессов*должно быть ограничено. После *завершения* какого-либо *процесса* его освободившийся идентификационный номер может быть повторно использован для другого *процесса*.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый *процесс* может получить в свое распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы. Информация о выделенных ресурсах заносится в *PCB*.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по *регистровому* и *пользовательскому контекстам*, при этом должен существовать способ определения, кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла. Операционная система Unix разрешает порождение *процесса* только первым способом; для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребенок должен заменить свой *пользовательский контекст* с помощью специального системного вызова. Операционная система VAX/*VMS* допускает только второе решение. В Windows NT возможны оба варианта (в различных API).

Порождение нового *процесса* как дубликата процесса-родителя приводит к возможности существования программ (т. е. исполняемых файлов), для работы которых организуется более одного *процесса*. Возможность замены *пользовательского контекста* *процесса* по ходу его работы (т. е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же *процесса* может последовательно выполняться несколько различных программ.

После того как *процесс* наделен содержанием, в *PCB* дописывается оставшаяся информация, и *состояние* нового *процесса* изменяется на *готовность*. Осталось сказать несколько слов о том, как ведут себя процессы-родители после рождения процессов-детей. Процесс-родитель может продолжать свое выполнение одновременно с выполнением процесса-ребенка, а может ожидать *завершения* работы некоторых или всех своих "детей".

Мы не будем подробно останавливаться на причинах, которые могут привести к *завершению* жизненного цикла *процесса*. После того как *процесс* завершил свою работу, операционная система переводит его в *состояние* закончил исполнение и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в *блоке управления процессом*. При этом сам *PCB* не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что процесс-родитель после *завершения* процесса-ребенка может запросить операционную систему о причине "смерти" порожденного им *процесса* и/или статистическую информацию о его работе. Подобная информация сохраняется в *PCB*отработавшего *процесса* до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося *процесса*окончательно исчезают из системы. В операционной системе Unix *процессы*, находящиеся в *состоянии* закончил исполнение, принято называть процессами-зомби.

Следует заметить, что в ряде операционных систем (например, в VAX/*VMS*) гибель процесса-родителя приводит к *завершению* работы всех его "детей". В других операционных системах (например, в Unix) процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в *PCB* процессов-детей о породившем их *процессе* для того, чтобы генеалогический лес *процессов* оставался целостным. Рассмотрим следующий пример. Пусть *процесс* с номером 2515 был порожден *процессом* с номером 2001 и после *завершения* его работы остается в вычислительной системе неограниченно долго. Тогда не исключено, что номер 2001 будет использован операционной системой повторно для совсем другого *процесса*. Если не изменить информацию о процессе-родителе для *процесса* 2515, то генеалогический лес *процессов* окажется некорректным – *процесс* 2515 будет считать своим родителем новый *процесс* 2001, а *процесс* 2001 будет открещиваться от нежданного потомка. Как правило, "осиротевшие" *процессы* "усыновляются" одним из системных *процессов*, который порождается при старте операционной системы и функционирует все время, пока она работает.

#### Многоразовые операции

Одноразовые *операции* приводят к изменению количества *процессов*, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые *операции*, напротив, не приводят к изменению количества *процессов* в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

В этом разделе мы кратко опишем действия, которые производит операционная система при выполнении многоразовых *операций над процессами*. Более подробно эти действия будут рассмотрены далее в соответствующих лекциях.

***Запуск процесса***. Из числа *процессов*, находящихся в *состоянии* готовность, операционная система выбирает один *процесс* для последующего исполнения. Критерии и алгоритмы такого выбора будут подробно рассмотрены в лекции 3 – "Планирование *процессов* ". Для избранного *процесса* операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. То, как она это делает, будет в деталях описано в лекциях 8-10. Далее *состояние процесса* изменяется на *исполнение* , восстанавливаются значения регистров для данного *процесса* и управление передается команде, на которую указывает счетчик команд *процесса*. Все данные, необходимые для восстановления *контекста*, извлекаются из *PCB процесса*, над которым совершается *операция*.

***Приостановка процесса***. Работа *процесса*, находящегося в *состоянии* исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого *процесса*, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть *системного* и *регистрового контекстов* *процесса* в его *PCB*, переводит *процесс* в *состояние* готовность и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

***Блокирование процесса***. *Процесс* блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет *процесс* в очередь *процессов*, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть *контекста процесса* в его *PCB*, переводит *процесс* из *состояния* исполнение в *состояние* ожидание . Подробнее эта *операция* будет рассматриваться в лекции 13.

***Разблокирование процесса***. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый *процесс* в *состоянии* ожидание для данного события, и если находился, переводит его в *состояние* готовность, выполняя необходимые действия, связанные с наступлением события (инициализация *операции* ввода-вывода для очередного ожидающего *процесса* и т. п.). Эта *операция*, как и *операция**блокирования*, будет подробно описана в лекции 13.

#### Переключение контекста

До сих пор мы рассматривали *операции над процессами* изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек *операций*, выполняемых над различными *процессами*, и сопровождается переключением процессора с одного *процесса* на другой.

Давайте для примера упрощенно рассмотрим, как в реальности может протекать *операция**разблокирования процесса*, ожидающего ввода-вывода (см. [рис. 2.5](https://www.intuit.ru/studies/courses/2192/31/lecture/970?page=4#image.2.5)). При исполнении процессором некоторого *процесса* (на рисунке – *процесс* 1) возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании *операций* на устройстве. Над выполняющимся *процессом* производится *операция**приостановки*. Далее операционная система разблокирует *процесс*, инициировавший запрос на ввод-вывод (на рисунке – *процесс* 2) и осуществляет *запуск* приостановленного или нового *процесса*, выбранного при выполнении планирования (на рисунке был выбран *разблокированный процесс* ). Как мы видим, в результате обработки информации об окончании *операции* ввода-вывода возможна смена *процесса*, находящегося в *состоянии*исполнение.

Для корректного переключения процессора с одного *процесса* на другой необходимо сохранить *контекст* исполнявшегося *процесса* и восстановить *контекст процесса*, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности *процессов* называется *переключением контекста*. Время, затраченное на *переключение контекста*, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно колеблется в диапазоне от 1 до 1000 микросекунд. Существенно сократить накладные расходы в современных операционных системах позволяет расширенная модель *процессов*, включающая в себя понятие threads of execution (нити исполнения или просто нити). Подробнее о нитях исполнения мы будем говорить в лекции 4 – "Кооперация *процессов* и основные аспекты ее логической организации".

**8**

### Уровни планирования

В первой лекции, рассматривая эволюцию компьютерных систем, мы говорили о двух видах *планирования* в вычислительных системах: *планировании* заданий и *планировании* использования процессора. *Планирование* заданий появилось в *пакетных системах* после того, как для хранения сформированных *пакетов заданий* начали использоваться магнитные диски. Магнитные диски, являясь устройствами прямого доступа, позволяют загружать задания в *компьютер* в произвольном порядке, а не только в том, в котором они были записаны на *диск*. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, мы и назвали ***планированием*** заданий. *Планирование* использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии готовность могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит *процессор* в свое распоряжение, т. е. будет переведен в состояние *исполнение*, мы использовали это *словосочетание*. Теперь, познакомившись с концепцией процессов в вычислительных системах, оба вида *планирования* мы будем рассматривать как различные ***уровни планирования процессов***.

*Планирование* заданий используется в качестве ***долгосрочного планирования процессов***. Оно отвечает за порождение новых процессов в системе, определяя ее ***степень мультипрограммирования***, т. е. количество процессов, одновременно находящихся в ней. Если *степень мультипрограммирования* системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому *долгосрочное планирование* осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого *уровня планирования* – *долгосрочное*. В некоторых операционных системах *долгосрочное планирование* сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени *порождение процесса* происходит сразу после появления соответствующего запроса. Поддержание разумной *степени мультипрограммирования* осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20–30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена.

*Планирование* использования процессора применяется в качестве ***краткосрочного планирования процессов***. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому *краткосрочное планирование* осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, чем и обусловлено название этого *уровня планирования* – *краткосрочное*.

В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название swapping, что можно перевести на русский язык как "перекачка", хотя в специальной литературе оно употребляется без перевода – свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным *уровнем планирования процессов* – ***среднесрочным*** .

**9**

**Критерии планирования и требования к алгоритмам**

Для каждого *уровня планирования процессов* можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя *планирование*. К числу таких целей можно отнести следующие:

* Справедливость – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
* Эффективность – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
* Сокращение полного времени выполнения ( ***turnaround time*** ) – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
* Сокращение времени ожидания ( ***waiting time*** ) – сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.
* Сокращение времени отклика ( ***response time*** ) – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей *планирования* желательно также, чтобы алгоритмы обладали следующими свойствами.

* Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма *планирования* не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.
* Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.
* Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
* Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

**10**

Все *параметры* *планирования* можно разбить на две большие группы: статические *параметры* и динамические *параметры*. Статические*параметры* не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

К статическим *параметрам* вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические *параметры* системы описывают количество свободных ресурсов на данный момент.

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

· Каким пользователем запущен процесс или сформировано задание.

· Насколько важной является поставленная задача, т. е. каков *приоритет* ее выполнения.

· Сколько процессорного времени запрошено пользователем для решения задачи.

· Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.

· Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Алгоритмы *долгосрочного планирования* используют в своей работе статические и динамические *параметры* вычислительной системы и статические *параметры* процессов (динамические *параметры* процессов на этапе загрузки заданий еще не известны). Алгоритмы *краткосрочного* и *среднесрочного планирования* дополнительно учитывают и динамические характеристики процессов. Для *среднесрочного планирования* в качестве таких характеристик может использоваться следующая *информация*:

· сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;

· сколько оперативной памяти занимает процесс;

· сколько процессорного времени уже предоставлено процессу.

**11**

Вытесняющее и невытесняющее планирование

Процесс *планирования* осуществляется частью операционной системы, называемой планировщиком. *Планировщик* может принимать решения о выборе для исполнения нового процесса из числа находящихся в состоянии готовность в следующих четырех случаях.

1. Когда процесс переводится из состояния исполнение в состояние закончил исполнение.

2. Когда процесс переводится из состояния исполнение в состояние ожидание.

3. Когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера).

4. Когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие). Подробно процедура такого перевода рассматривалась в лекции 2 (раздел "Переключение контекста"), где мы показали, почему при этом возникает возможность смены процесса, находящегося в состоянии исполнение.

Если в операционной системе планирование осуществляется только в вынужденных ситуациях, говорят, что имеет *место* *невытесняющее (nonpreemptive) планирование*. Если *планировщик* принимает и вынужденные, и невынужденные решения, говорят о *вытесняющем (preemptive) планировании*. Термин *"вытесняющее планирование"* возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния *исполнение* другим процессом.

*Невытесняющее планирование* используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме *планирования* процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод *планирования* относительно просто реализуем и достаточно эффективен, так как позволяет выделить большую часть процессорного времени для работы самих процессов и до минимума сократить *затраты* на переключение контекста. Однако при *невытесняющем планировании* возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

*Вытесняющее планирование* обычно используется в системах разделения времени. В этом режиме *планирования* процесс может быть приостановлен в любой момент исполнения. Операционная система устанавливает специальный таймер для генерации *сигнала прерывания*по истечении некоторого интервала времени – *кванта*. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемое время отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают "зависание" компьютерной системы из-за зацикливания какой-либо программы

**12**

Простейшим алгоритмом *планирования* является алгоритм, который принято обозначать аббревиатурой *FCFS* по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его *PCB* помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его *PCB*. Очередь подобного типа имеет в программировании специальное наименование – FIFO1, сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет *невытесняющее планирование*. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего *CPU burst* . После этого для выполнения выбирается новый процесс из начала очереди.

Преимуществом *алгоритма FCFS* является легкость его реализации, но в то же время он имеет и много недостатков.

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным *CPU burst* , то короткие процессы, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала выполнения. Поэтому *алгоритм FCFS* практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в *интерактивных процессах*.

Алгоритмы планирования. Round Robin (RR)

Модификацией *алгоритма FCFS* является алгоритм, получивший название *Round Robin* (*Round Robin* – это вид детской карусели в США) или сокращенно *RR*. По сути дела, это тот же самый алгоритм, только реализованный в режиме *вытесняющего планирования*. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный *квант времени*, обычно 10 – 100 миллисекунд. Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного *кванта времени*. При выполнении процесса возможны два варианта.

· Время непрерывного использования процессора, необходимое процессу (остаток текущего *CPU burst* ), меньше или равно продолжительности *кванта времени*. Тогда процесс по своей воле освобождает процессор до истечения *кванта времени*, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет *кванта* заново.

· Продолжительность остатка текущего *CPU burst* процесса больше, чем *квант времени*. Тогда по истечении этого *кванта* процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

При очень больших величинах *кванта времени*, когда каждый процесс успевает завершить свой *CPU burst* до возникновения прерывания по времени, *алгоритм RR* вырождается в *алгоритм FCFS*. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения *контекста процессов*. В реальных условиях при слишком малой величине *кванта времени* и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

**13**

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов *планирования*. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность. Этим очередям приписываются фиксированные *приоритеты*. Например, *приоритет* очереди системных процессов устанавливается выше, чем *приоритет* очередей пользовательских процессов. А *приоритет* очереди процессов, запущенных студентами, ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для *планирования* могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться *алгоритм FCFS*, а для интерактивных процессов – *алгоритм RR*. Подобный подход, получивший название *многоуровневых очередей*, повышает гибкость *планирования*: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

**14**

Дальнейшим развитием алгоритма *многоуровневых очередей* является добавление к нему механизма обратной связи. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из одной очереди в другую в зависимости от своего поведения.

Для простоты рассмотрим ситуацию, когда процессы в состоянии готовность организованы в 4 очереди, как на [рисунке 3.6](https://www.intuit.ru/studies/courses/2192/31/lecture/972?page=6#image.3.6). *Планирование процессов* между очередями осуществляется на основе *вытесняющего* приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее *приоритет*. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс. Процессы в очереди 2 не будут выбраны для выполнения, пока есть хоть один процесс в очередях 0 и 1. И наконец, процесс в очереди 3 может получить процессор в свое распоряжение только тогда, когда очереди 0, 1 и 2 пусты. Если при работе процесса появляется другой процесс в какой-либо более *приоритетной* очереди, исполняющийся процесс вытесняется новым. *Планирование процессов* внутри очередей0 – 2 осуществляется с использованием *алгоритма RR*, *планирование* процессов в очереди 3 основывается на *алгоритме FCFS*.

Родившийся процесс поступает в очередь 0. При выборе на исполнение он получает в свое распоряжение *квант времени* размером 8единиц. Если продолжительность его *CPU burst* меньше этого *кванта времени*, процесс остается в очереди 0. В противном случае он переходит в очередь 1. Для процессов из очереди 1 *квант времени* имеет величину 16. Если процесс не укладывается в это время, он переходит в очередь 2. Если укладывается – остается в очереди 1. В очереди 2 величина *кванта времени* составляет 32 единицы. Если для непрерывной работы процесса и этого мало, процесс поступает в очередь 3, для которой квантование времени не применяется и, при отсутствии готовых процессов в других очередях, может исполняться до окончания своего *CPU burst*. Чем больше значение продолжительности *CPU burst*, тем в менее *приоритетную* очередь попадает процесс, но тем на большее процессорное время он может рассчитывать. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора, окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, – в низкоприоритетных.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры процессы из очередей 1, 2 и 3 могут помещаться в очередь 0, после завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1, а после завершения ожидания всех других событий – из очереди 3 в очередь 2. Перемещение процессов из очередей с низкими *приоритетами* в очереди с высокими *приоритетами* позволяет более полно учитывать изменение поведения процессов с течением времени.

*Многоуровневые очереди с обратной связью* представляют собой наиболее общий подход к *планированию процессов* из числа подходов, рассмотренных нами. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибкостью. Понятно, что существует много других разновидностей такого способа *планирования*, помимо варианта, приведенного выше. Для полного описания их конкретного воплощения необходимо указать:

· Количество очередей для процессов, находящихся в состоянии готовность.

· Алгоритм *планирования*, действующий между очередями.

· Алгоритмы *планирования*, действующие внутри очередей.

· Правила помещения родившегося процесса в одну из очередей.

· Правила перевода процессов из одной очереди в другую.

**15**

*Алгоритмы SJF* и *гарантированного планирования* представляют собой частные случаи *приоритетного планирования*. При *приоритетном планировании* каждому процессу присваивается определенное числовое значение – *приоритет*, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми *приоритетами* планируются в порядке *FCFS*. Для *алгоритма SJF* в качестве такого *приоритета выступает* оценка продолжительности следующего *CPU burst*. Чем меньше значение этой оценки, тем более высокий *приоритет* имеет процесс. Для алгоритма *гарантированного планирования приоритетом* служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса *приоритет* .

Алгоритмы назначения *приоритетов* процессов могут опираться как на *внутренние параметры*, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей *I/O burst* к *CPU burst* и т. д.*Алгоритмы SJF* и гарантированного планирования используют *внутренние параметры*. В качестве *внешних параметров* могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы. Высокий внешний *приоритет* может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

*Планирование* с использованием *приоритетов* может быть как *вытесняющим*, так и *невытесняющим*. При *вытесняющем планировании* процесс с более высоким *приоритетом*, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким *приоритетом*. В случае *невытесняющего планирования* он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов *приоритетного* *планирования*.

В рассмотренном выше примере *приоритеты* процессов с течением времени не изменялись. Такие *приоритеты* принято называть статическими. Механизмы статической *приоритетности* легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее *приоритетного* процесса. Однако статические *приоритеты* не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические *приоритеты процессов*, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического *приоритета*, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического *приоритета процесса* является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение *приоритета процессов* проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного *кванта времени* или по завершении процесса. Примерами алгоритмов с динамическими *приоритетами* являются *алгоритм SJF* и алгоритм *гарантированного планирования*. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

Главная проблема *приоритетного планирования* заключается в том, что при ненадлежащем выборе механизма назначения и изменения *приоритетов* низкоприоритетные процессы могут не запускаться неопределенно долгое время. Обычно случается одно из двух. Или они все же дожидаются своей очереди на исполнение (в девять часов утра в воскресенье, когда все приличные программисты ложатся спать). Или вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения *приоритета процесса*, находящегося в состоянии готовность. Пусть изначально процессам присваиваются *приоритеты* от 128 до 255. Каждый раз по истечении определенного промежутка времени значения*приоритетов* готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии исполнение, присваивается первоначальное значение *приоритета*. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

**16**

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их *кооперации*?

· Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.

· Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.

· Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные *процессы*, взаимодействующие путем передачи *сообщений* через микроядро.

· Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. "Общение" процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если *деятельность* процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в "общении" не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть кооперативными или *взаимодействующими процессами* , в отличие от *независимых процессов*, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить *значение*, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

**17**

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. *По* объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

· *Сигнальные*. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?

· *Канальные*. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.

· *Разделяемая память*. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием *разделяемой памяти* занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование *разделяемой памяти* для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как *сигнальным* и *канальным средствам коммуникации* для этого необходимы специальные системные вызовы. *Разделяемая память* представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

**18**

##### Буферизация

Может ли линия связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект? Каков объем этой информации? Иными словами, речь идет о том, обладает ли канал связи буфером и каков объем этого буфера. Здесь можно выделить три принципиальных варианта.

1. Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами (в реальности этот случай никогда не реализуется).
2. Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места недостаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.
3. Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании канального средства связи с непрямой адресацией под емкостью буфера обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

##### Поток ввода/вывода и сообщения

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. При передаче данных с помощью потоковой модели операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т. е. одним куском или порциями по 20 байт, пришли они от одного процесса или от разных. Данные представляют собой простой поток байтов, без какой-либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить pipe и FIFO, описанные ниже.

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через pipe (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у нас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут "сливать" информацию, а из другого конца принимать полученный поток. Такой способ реализует потоковую модель ввода/вывода. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключительно со своими наследниками – процессами-детьми и их потомками. Поэтому использовать pipeдля связи между собой могут только родственные процессы, имеющие общего предка, создавшего данный канал связи.

Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть FIFO или именованный pipe. Именованный pipe может использоваться для организации связи между любыми процессами в системе.

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные сообщения, вводя между данными, по крайней мере, границы сообщений. Примером границ сообщений являются точки между предложениями в сплошном тексте или границы абзаца. Кроме того, к передаваемой информации могут быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено. Примером указания отправителя могут служить подписи под эпиграфами в лекции. Все сообщения могут иметь одинаковый фиксированный размер или могут быть переменной длины. В вычислительных системах используются разнообразные средства связи для передачи сообщений: очереди сообщений, sockets (гнезда) и т. д. Часть из них мы рассмотрим подробнее в дальнейшем, в частности очереди сообщений будут рассмотрены в лекции 6, а гнезда (иногда их еще называют по транслитерации английского названия – сокеты) в лекции 14.

И потоковые линии связи, и каналы сообщений всегда имеют буфер конечной длины. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для сообщений, мы будем измерять ее в сообщениях.

**19**

Мы будем называть способ коммуникации надежным, если при обмене данными выполняются четыре условия.

1. Не происходит потери информации.

2. Не происходит повреждения информации.

3. Не появляется лишней информации.

4. Не нарушается порядок данных в процессе обмена.

Очевидно, что передача данных через *разделяемую память* является надежным способом связи. То, что мы сохранили в *разделяемой памяти*, будет считано другими процессами в первозданном виде, если, конечно, не произойдет сбоя в питании компьютера. Для других средств коммуникации, как видно из приведенных выше примеров, это не всегда верно.

Каким образом в вычислительных системах пытаются бороться с ненадежностью коммуникаций? Давайте рассмотрим возможные варианты на примере обмена данными через линию связи с помощью *сообщений*. Для обнаружения повреждения информации будем снабжать каждое передаваемое *сообщение* некоторой контрольной суммой, вычисленной по посланной информации. При приеме *сообщения* контрольную сумму будем вычислять заново и проверять ее соответствие пришедшему значению. Если данные не повреждены (контрольные суммы совпадают), то подтвердим правильность их получения. Если данные повреждены (контрольные суммы не совпадают), то сделаем вид, что *сообщение* к нам не поступило. Вместо контрольной суммы можно использовать специальное кодирование передаваемых данных с помощью кодов, исправляющих ошибки. Такое кодирование позволяет при числе искажений информации, не превышающем некоторого значения, восстановить первоначальные неискаженные данные. Если по прошествии некоторого интервала времени подтверждение правильности полученной информации не придет на передающий конец линии связи, будем считать информацию утерянной и пошлем ее повторно. Для того чтобы избежать двойного получения одной и той же информации, на приемном конце линии связи должен осуществляться соответствующий контроль. Для гарантии правильного порядка получения *сообщений* будем их нумеровать. При приеме *сообщения* с номером, не соответствующим ожидаемому, поступаем с ним как с утерянным и ждем *сообщения* с правильным номером.

Подобные действия могут быть возложены:

· на операционную систему;

· на процессы, обменивающиеся данными;

· совместно на систему и процессы, разделяя их ответственность. Операционная система может обнаруживать ошибки при передаче данных и извещать об этом *взаимодействующие процессы* для принятия ими решения о дальнейшем поведении.

**20**

Для того чтобы реализовать нашу идею, введем новую абстракцию внутри понятия "процесс" – *нить исполнения* или просто *нить* (в англоязычной литературе используется термин *thread* ). *Нити процесса* разделяют его программный код, *глобальные переменные* и системные ресурсы, но каждая *нить* имеет собственный программный *счетчик*, свое содержимое регистров и свой *стек*. Теперь процесс представляется как совокупность взаимодействующих *нитей* и выделенных ему ресурсов. Процесс, содержащий всего одну *нить исполнения*, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин "традиционный процесс". Иногда *нити* называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. *Нити*, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из одного состояния в другое. Состояния *нитей* аналогичны состояниям традиционных процессов. Из состояния рождение процесс приходит содержащим всего одну *нить исполнения*. Другие *нити процесса* будут являться потомками этой нити-прародительницы. Мы можем считать, что процесс находится в состоянии готовность, если хотя бы одна из его *нитей* находится в состоянии готовность и ни одна из *нитей* не находится в состоянии исполнение. Мы можем считать, что процесс находится в состоянии исполнение, если одна из его *нитей* находится в состоянии исполнение. Процесс будет находиться в состоянии ожидание, если все его *нити* находятся в состоянии ожидание. Наконец, процесс находится в состоянии закончил исполнение, если все его *нити* находятся в состоянии закончила исполнение. Пока одна *нить процесса* заблокирована, другая *нить* того же процесса может выполняться. *Нити* разделяют *процессор* так же, как это делали традиционные процессы, в соответствии с рассмотренными алгоритмами планирования.

Поскольку *нити* одного процесса разделяют существенно больше ресурсов, чем различные процессы, то *операции* создания новой *нити* и переключения контекста между *нитями* одного процесса занимают значительно меньше времени, чем аналогичные *операции* для процессов в целом. Предложенная нами схема *совмещения работы* в терминах *нитей* одного процесса получает право на существование.

**21**

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых " активностях ". Под активностями мы будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые, или атомарные, операции. Например, активность "приготовление бутерброда" можно разбить на следующие атомарные операции:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать ломтик хлеба маслом.
4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

P: a b c  
Q: d e f

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving.

*Атомарные операции* *активностей* могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри*активностей*. Так как псевдопараллельное выполнение двух *активностей* приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две *активности* P и Q, состоящие из двух *атомарных операций* каждая:

P: x=2 Q: x=3

y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для *активностей* общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор *активностей* (например, программ) *детерминирован*, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он *недетерминирован* . Выше приведен пример*недетерминированного набора программ*. Понятно, что *детерминированный набор* *активностей* можно безбоязненно выполнять в режиме разделения времени. Для *недетерминированного набора* такое *исполнение* нежелательно.

*Условия Бернстайна* информативны, но слишком жестки. *По* сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы *детерминированный набор* образовывали *активности*, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований *атомарных операций*, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный *доступ* программ к некоторым данным.

Про *недетерминированный набор программ* (и *активностей* вообще) говорят, что он имеет *race condition* ( *состояние гонки,* состояние состязания). В приведенном выше примере процессы состязаются за *вычисление* значений переменных x и y.

Задачу *упорядоченного доступа* к *разделяемым данным* (устранение *race condition*) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется *взаимоисключением (mutual exclusion)* . Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними *взаимоисключениями* уже не обойтись, нужна *взаимосинхронизация* поведения программ.

**22**

Важным понятием при изучении способов синхронизации процессов является понятие *критической секции* (*critical section*) программы. *Критическая секция* – это часть программы, исполнение которой может привести к возникновению *race condition* для определенного набора программ. Чтобы исключить эффект гонок *по* отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей *критической секции*, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию *взаимоисключения* для *критических секций* программ. Реализация *взаимоисключения* для *критических секций* программ с практической точки зрения означает, что *по* отношению к другим процессам, участвующим во взаимодействии, *критическая секция* начинает выполняться как *атомарная операция*.

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем *критическом участке*, другие процессы не могли войти в свои *критические участки*. Мы видим, что *критический участок* должен сопровождаться прологом ( entry section ) – "закрыть дверь изнутри на засов" – и эпилогом ( exit section ) – "отодвинуть засов", которые не имеют отношения к *активности* одиночного процесса. Во *время выполнения* пролога процесс должен, в частности, получить разрешение на вход в *критический участок*, а во *время выполнения* эпилога – сообщить другим процессам, что он покинул *критическую секцию*.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

while (some condition) {

entry section

critical section

exit section

remainder section

}

Здесь под *remainder* section понимаются все *атомарные операции*, не входящие в *критическую секцию*.

**23**

Организация *взаимоисключения* для *критических участков*, конечно, позволит избежать возникновения *race condition*, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих *критические участки*, если они могут проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд *взаимоисключения*. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store,test ) являются *атомарными операциями*.

2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.

3. Если процесс Pi исполняется в своем *критическом участке*, то не существует никаких других процессов, которые исполняются в соответствующих *критических секциях*. Это условие получило название условия *взаимоисключения (mutual exclusion)*.

4. Процессы, которые находятся вне своих *критических участков* и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные *критические участки*. Если нет процессов в *критических секциях* и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в *remainder* section, должны принимать решение о том, какой процесс войдет в свою *критическую секцию*. Такое решение не должно приниматься бесконечно долго. Это условие получило название *условия прогресса (progress)* .

5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой *критический участок*. От того момента, когда процесс запросил разрешение на вход в *критическую секцию*, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои *критические участки* лишь ограниченное число раз. Это условие получило название *условия ограниченного ожидания (bound waiting)* .

**24**

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

while (some condition) {

запретить все прерывания

critical section

разрешить все прерывания

remainder section

}

Поскольку выход процесса из состояния исполнение без его завершения осуществляется по прерыванию, внутри *критической секции* никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к *критическим секциям* внутри самой операционной системы, например при обновлении содержимого *PCB*.

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в *критическую секцию* только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из *критической секции* процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " *Критическая секция* ").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменная является разделяемой \*/

while (some condition) {

while(lock); lock = 1;

critical section

lock = 0;

remainder section

}

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию *взаимоисключения*, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессуP1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в *критический участок*. Мы получаем два процесса, одновременно выполняющих свои *критические секции*.

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для *критического участка*, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

shared int turn = 0;

while (some condition) {

while(turn != i);

critical section

turn = 1-i;

remainder section

}

Очевидно, что *взаимоисключение* гарантируется, процессы входят в *критическую секцию* строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет *условию прогресса*. Например, если значение turn равно 1, и процесс P0 готов войти в*критический участок*, он не может сделать этого, даже если процесс P1 находится в remainder section.

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в*критический участок*

shared int ready[2] = {0, 0};

Когда i-й процесс готов войти в *критическую секцию*, он присваивает элементу массива ready[i] значение равное 1. После выхода из*критической секции* он, естественно, сбрасывает это значение в 0. Процесс не входит в *критическую секцию*, если другой процесс уже готов к входу в *критическую секцию* или находится в ней.

while (some condition) {

ready[i] = 1;

while(ready[1-i]);

critical section

ready[i] = 0;

remainder section

}

Полученный алгоритм обеспечивает *взаимоисключение*, позволяет процессу, готовому к входу в *критический участок*, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает *условие прогресса*. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в *критическую секцию*. Возникает ситуация, которую принято называть тупиковой (*deadlock*).

**25**

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

shared int ready[2] = {0, 0};

shared int turn;

while (some condition) {

ready[i] = 1;

turn =1-i;

while(ready[1-i] && turn == 1-i);

critical section

ready[i] = 0;

remainder section

}

При исполнении пролога *критической секции* процесс Pi заявляет о своей готовности выполнить *критический участок* и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в *критическом участке* продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия *взаимоисключения* методом от противного. Пусть оба процесса одновременно оказались внутри своих*критических секций*. Заметим, что процесс Pi может войти в *критическую секцию*, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои *критические секции* одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в *критические секции* из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в*критический участок*, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания *критического участка* процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой*критический участок*. Мы пришли к противоречию. Следовательно, имеет место *взаимоисключение*.

Докажем выполнение *условия прогресса*. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою*критическую секцию* только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению*критического участка*, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению *критического участка*, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение *критической секции*. Если процесс P1 завершил выполнение *критического участка*, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, *условие прогресса* выполняется.

Отсюда же вытекает выполнение *условия ограниченного ожидания*. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего *критического участка* после не более чем одного прохода по*критической секции* процесса P1.

**26**

*Алгоритм Петерсона* дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название *алгоритм булочной*, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за не атомарности операции вычисления следующего номера *алгоритм булочной* не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

shared enum {false, true} choosing[n];

shared int number[n];

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

(a,b) < (c,d), если a < c

или если a == c и b < d

max(a0, a1, ...., an) – это число k такое, что

k >= ai для всех i = 0, ...,n

Структура процесса Pi для *алгоритма булочной* приведена ниже

while (some condition) {

choosing[i] = true;

number[i] = max(number[0], ...,

number[n-1]) + 1;

choosing[i] = false;

for(j = 0; j < n; j++){

while(choosing[j]);

while(number[j] != 0 && (number[j],j) <

(number[i],i));

}

critical section

number[i] = 0;

remainder section

}

**27**

Наличие аппаратной поддержки *взаимоисключений* позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации*взаимоисключений*, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как *атомарные операции*. Давайте обсудим, как концепции таких команд могут использоваться для реализации *взаимоисключений*.

#### Команда Test-and-Set (проверить и присвоить 1)

О выполнении команды *Test-and-Set* , осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

int Test\_and\_Set (int \*target){

int tmp = \*target;

\*target = 1;

return tmp;

}

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал*взаимоисключения*

shared int lock = 0;

while (some condition) {

while(Test\_and\_Set(&lock));

critical section

lock = 0;

remainder section

}

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет *условию ограниченного ожидания* для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

#### Команда Swap (обменять значения)

Выполнение команды *Swap* , обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

void Swap (int \*a, int \*b){

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

Применяя атомарную команду *Swap* , мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key, локальную для каждого процесса:

shared int lock = 0;

int key;

while (some condition) {

key = 1;

do Swap(&lock,&key);

while (key);

critical section

lock = 0;

remainder section

}

**28**

### Семафоры

Одним из первых *механизмов*, предложенных для синхронизации поведения процессов, стали *семафоры*, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

#### Концепция семафоров

*Семафор* представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

P(S): пока S == 0 процесс блокируется;

S = S – 1;

V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции P над *семафором* S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего изS вычитается 1. При выполнении операции V над *семафором* S к его значению просто прибавляется 1. В момент создания *семафор*может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

**29**

#### Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача *producer-consumer* (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом.

Producer: while(1) {

produce\_item;

put\_item;

}

Consumer: while(1) {

get\_item;

consume\_item;

}

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения. Как можно реализовать эти условия с помощью *семафоров*? Возьмем три *семафора*: empty, full и *mutex*. *Семафор* full будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. *Семафор* empty будем использовать для организации ожидания производителя при заполненном буфере, а*семафор* *mutex* – для организации взаимоисключения на критических участках, которыми являются действия put\_item и get\_item(операции "положить информацию" и "взять информацию" не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на C-подобном языке выглядит так:

Semaphore mutex = 1;

Semaphore empty = N; /\* где N – емкость буфера\*/

Semaphore full = 0;

Producer:

while(1) {

produce\_item;

P(empty);

P(mutex);

put\_item;

V(mutex);

V(full);

}

Consumer:

while(1) {

P(full);

P(mutex);

get\_item;

V(mutex);

V(empty);

consume\_item;

}

**30**

***Мониторы*** представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. *Монитор* обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих *монитору*. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри *монитора*, и свои параметры. На абстрактном уровне можно описать структуру*монитора* следующим образом:

monitor monitor\_name {

описание внутренних переменных ;

void m1(...){...

}

void m2(...){...

}

...

void mn(...){...

}

{

блок инициализации

внутренних переменных;

}

}

Здесь функции m1,..., mn представляют собой функции-методы *монитора*, а блок инициализации внутренних переменных содержит*операции*, которые выполняются один и только один раз: при создании *монитора* или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии **готовность** или **исполнение**, внутри данного *монитора*. Поскольку *мониторы* представляют собой особые конструкции языка программирования, *компилятор* может отличить *вызов функции*, принадлежащей *монитору*, от вызовов других функций и обработать его специальным образом, добавив к нему *пролог* и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на *компилятор*, а не на программиста, работа программиста при использовании *мониторов* существенно упрощается, а *вероятность* возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно *семафорам* full и empty в предыдущем примере. Для этого в *мониторах* было введено понятие условных переменных (condition variables)1, над которыми можно совершать две *операции* wait и signal, отчасти похожие на *операции* P и V над *семафорами*.

Если *функция* *монитора* не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо *условной переменной*. При этом процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в *монитор*.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой *условной переменной*. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались *операции* signal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри *монитора*? Хор предложил, чтобы пробужденный процесс подавлял *исполнение* разбудившего процесса, пока он сам не покинет *монитор*. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает *монитор* немедленно после исполнения *операции* signal. Мы будем придерживаться подхода Хансена.

**Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция** signal **всегда должна выполняться после операции** wait. **Если операция** signal **выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции** wait **всегда будет приводить к блокированию процесса**.

**31**

В предыдущих лекциях мы рассматривали способы синхронизации процессов, которые позволяют процессам успешно кооперироваться. Однако в некоторых случаях могут возникнуть непредвиденные затруднения. Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется тупиком (deadlock) . Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. Системная тупиковая ситуация, или "зависание системы", является следствием того, что один или более процессов находятся в состоянии тупика. Иногда подобные ситуации называют взаимоблокировками . В общем случае проблема тупиков эффективного решения не имеет.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот.

Определение. Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вызвать только другой процесс данного множества. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут спать вместе.

Выше приведен пример взаимоблокировки, возникающей при работе с так называемыми выделенными устройствами. Тупики, однако, могут иметь место и в других ситуациях. Hапример, в системах управления базами данных записи могут быть локализованы процессами, чтобы избежать состояния гонок (см. лекцию 5 "Алгоритмы синхронизации"). В этом случае может получиться так, что один из процессов заблокировал записи, необходимые другому процессу, и наоборот. Таким образом, тупики могут иметь место как на аппаратных, так и на программных ресурсах.

Тупики также могут быть вызваны ошибками программирования. Например, процесс может напрасно ждать открытия семафора, потому что в некорректно написанном приложении эту операцию забыли предусмотреть. Другой причиной бесконечного ожидания может быть дискриминационная политика по отношению к некоторым процессам. Однако чаще всего событие, которого ждет процесс в тупиковой ситуации, – освобождение ресурса, поэтому в дальнейшем будут рассмотрены методы борьбы с тупиками ресурсного типа.

Ресурсами могут быть как устройства, так и данные. Hекоторые ресурсы допускают разделение между процессами, то есть являются разделяемыми ресурсами. Например, память, процессор, диски коллективно используются процессами. Другие не допускают разделения, то есть являются выделенными, например лентопротяжное устройство. К взаимоблокировке может привести использование как выделенных, так и разделяемых ресурсов. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как запись предполагает исключительный доступ к данным на диске. Можно считать, что часть диска, куда происходит запись, выделена конкретному процессу. Поэтому в дальнейшем мы будем исходить из предположения, что тупики связаны с выделенными ресурсами , то есть тупики возникают, когда процессу предоставляется эксклюзивный доступ к устройствам, файлам и другим ресурсам.

Традиционная последовательность событий при работе с ресурсом состоит из запроса, использования и освобождения ресурса. Тип запроса зависит от природы ресурса и от ОС. Запрос может быть явным, например специальный вызов request, или неявным – open для открытия файла. Обычно, если ресурс занят и запрос отклонен, запрашивающий процесс переходит в состояние ожидания.

Далее в данной лекции будут рассматриваться вопросы обнаружения, предотвращения, обхода тупиков и восстановления после тупиков. Как правило, борьба с тупиками – очень дорогостоящее мероприятие. Тем не менее для ряда систем, например для систем реального времени, иного выхода нет.

**32**

Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 г.

1. Условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс.
2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.
3. Условие неперераспределяемости (No preemtion). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Для образования тупика необходимым и достаточным является выполнение всех четырех условий.

Обычно тупик моделируется циклом в графе, состоящем из узлов двух видов: прямоугольников – процессов и эллипсов – ресурсов, наподобие того, что изображен на [рис. 7.1](https://www.intuit.ru/studies/courses/2192/31/lecture/980?page=1#image.7.1). Стрелки, направленные от ресурса к процессу, показывают, что ресурс выделен данному процессу. Стрелки, направленные от процесса к ресурсу, означают, что процесс запрашивает данный ресурс.

**33**

Проблема тупиков инициировала много интересных исследований в области информатики. Очевидно, что условие циклического ожидания отличается от остальных. Первые три условия формируют правила, существующие в системе, тогда как четвертое условие описывает ситуацию, которая может сложиться при определенной неблагоприятной последовательности событий. Поэтому методы предотвращения взаимоблокировок ориентированы главным образом на нарушение первых трех условий путем введения ряда ограничений на поведение процессов и способы распределения ресурсов. Методы обнаружения и устранения менее консервативны и сводятся к поиску и разрыву циклаожидания ресурсов.

Итак, основные направления борьбы с тупиками:

* Игнорирование проблемы в целом
* Предотвращение тупиков
* Обнаружение тупиков
* Восстановление после тупиков

**34**

Обнаружение взаимоблокировки сводится к фиксации тупиковой ситуации и выявлению вовлеченных в нее процессов. Для этого производится проверка наличия циклического ожидания в случаях, когда выполнены первые три условия возникновения тупика. Методы обнаружения активно используют графы распределения ресурсов.

Рассмотрим модельную ситуацию.

* Процесс P1 ожидает ресурс R1.
* Процесс P2 удерживает ресурс R2 и ожидает ресурс R1.
* Процесс P3 удерживает ресурс R1 и ожидает ресурс R3.
* Процесс P4 ожидает ресурс R2.
* Процесс P5 удерживает ресурс R3 и ожидает ресурс R2.

Вопрос состоит в том, является ли данная ситуация тупиковой, и если да, то какие процессы в ней участвуют. Для ответа на этот вопрос можно сконструировать граф ресурсов, как показано на [рис. 7.3](https://www.intuit.ru/studies/courses/2192/31/lecture/980?page=3#image.7.3). Из рисунка видно, что имеется цикл, моделирующий условие кругового ожидания, и что процессы P2,P3,P5, а может быть, и другие находятся в тупиковой ситуации.

Визуально легко обнаружить наличие тупика, но нужны также формальные алгоритмы, реализуемые на компьютере.

Один из таких алгоритмов описан в [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)], там же можно найти ссылки на другие алгоритмы.

Существуют и другие способы обнаружения тупиков, применимые также в ситуациях, когда имеется несколько ресурсов каждого типа. Так в [[Дейтел, 1987](https://www.intuit.ru/studies/courses/2192/31/literature#literature.16)] описан способ, называемый редукцией графа распределения ресурсов, а в [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)] – матричный алгоритм.

**35**

Обнаружив тупик, можно вывести из него систему, нарушив одно из условий существования тупика. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы.

Сложность восстановления обусловлена рядом факторов.

* В большинстве систем нет достаточно эффективных средств, чтобы приостановить процесс, вывести его из системы и возобновить впоследствии с того места, где он был остановлен.
* Если даже такие средства есть, то их использование требует затрат и внимания оператора.
* Восстановление после тупика может потребовать значительных усилий.

Самый простой и наиболее распространенный способ устранить тупик – завершить выполнение одного или более процессов, чтобы впоследствии использовать его ресурсы. Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов. После каждой ликвидации должен запускаться алгоритм обнаружения тупика.

По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу (такие процессы называются идемпотентными). Примером такого процесса может служить компиляция. С другой стороны, процесс, который изменяет содержимое базы данных, не всегда может быть корректно запущен повторно.

В некоторых случаях можно временно забрать ресурс у текущего владельца и передать его другому процессу. Возможность забрать ресурс у процесса, дать его другому процессу и затем без ущерба вернуть назад сильно зависит от природы ресурса. Подобное восстановление часто затруднительно, если не невозможно.

В ряде систем реализованы средства отката и перезапуска или рестарта с контрольной точки (сохранение состояния системы в какой-то момент времени). Если проектировщики системы знают, что тупик вероятен, они могут периодически организовывать для процессов контрольные точки. Иногда это приходится делать разработчикам прикладных программ.

Когда тупик обнаружен, видно, какие ресурсы вовлечены в цикл кругового ожидания. Чтобы осуществить восстановление, процесс, который владеет таким ресурсом, должен быть отброшен к моменту времени, предшествующему его запросу на этот ресурс

**36**

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую ) и вторичную (внешнюю) память.

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

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями, как показано на [рис. 8.1](https://www.intuit.ru/studies/courses/2192/31/lecture/982?page=1#image.8.1). Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.

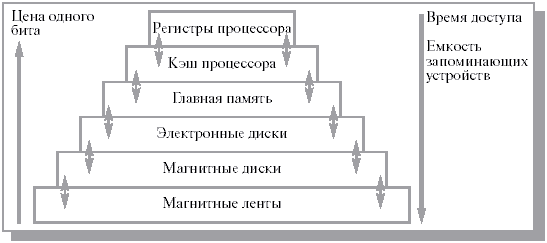


Рис. 8.1. Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

**37**

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. Сегмент – область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем адресном пространстведублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

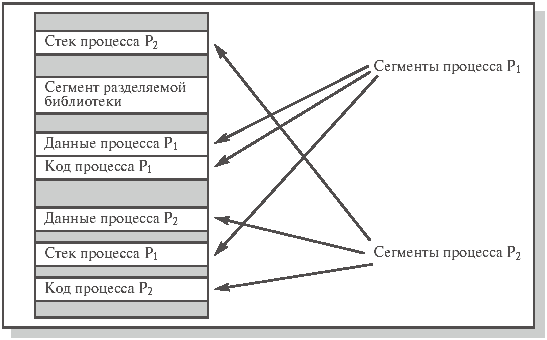


Рис. 8.2. Расположение сегментов процессов в памяти компьютера

Некоторые сегменты, описывающие адресное пространство процесса, показаны на [рис. 8.2](https://www.intuit.ru/studies/courses/2192/31/lecture/982?page=1#image.8.2). Более подробная информация о типах сегментовимеется в лекции 10.

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

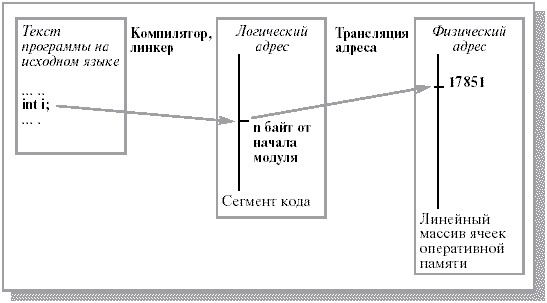
Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как n байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) адресным пространством.

**38**

Итак логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 232) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют трансляцией (привязкой) адреса или связыванием адресов (см. [рис. 8.3](https://www.intuit.ru/studies/courses/2192/31/lecture/982?page=1#image.8.3)).

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах [[Silberschatz, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.8)].

* Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.
* Этап загрузки (Load time). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.
* Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм (см. лекцию 9).

****

**39**

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

##### Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов(см. [рис. 8.4](https://www.intuit.ru/studies/courses/2192/31/lecture/982?page=2#image.8.4)).

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

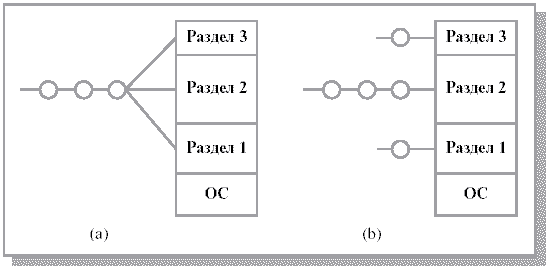


Рис. 8.4. Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

##### Один процесс в памяти

Частный случай схемы с фиксированными разделами – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

##### Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел "Виртуальная память").

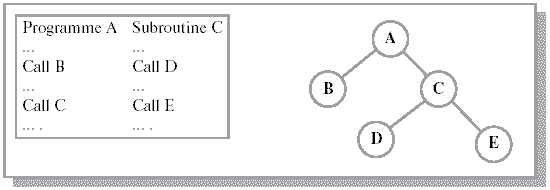


Рис. 8.5. Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Для примера, приведенного на [рис. 8.5](https://www.intuit.ru/studies/courses/2192/31/lecture/982?page=2#image.8.5), текст этого файла может выглядеть так:

A-(B,C)  
C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

##### Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

##### Схема с переменными разделами

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера ([рис. 8.6](https://www.intuit.ru/studies/courses/2192/31/lecture/982?page=2#image.8.6)). Смежные свободные участки могут быть объединены.

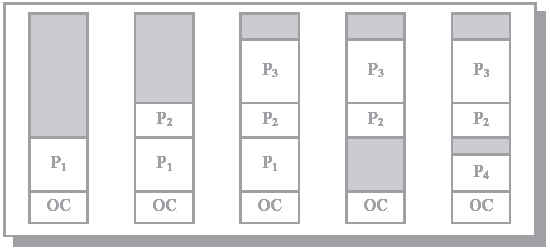


Рис. 8.6. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии.

* Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел.
* Стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.
* Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

**40**

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы(page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадрсодержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на [рис. 8.7](https://www.intuit.ru/studies/courses/2192/31/lecture/982?page=3#image.8.7). Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

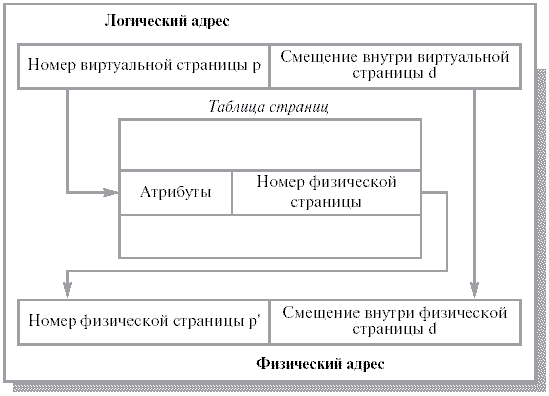


Рис. 8.7. Связь логического и физического адресов при страничной организации памяти

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

**41**

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 232 байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программаобращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

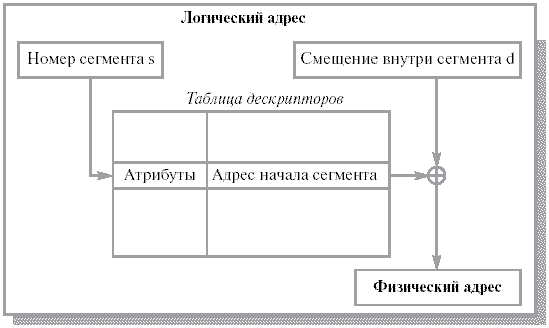


Рис. 8.8. Преобразование логического адреса при сегментной организации памяти

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегментаи смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

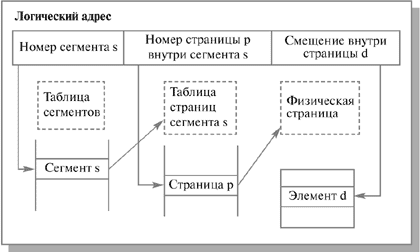


Рис. 8.9. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Сегментно-страничная и сегментная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

**42**

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И, наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц .

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.

* Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.
* Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.
* Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы "видимости" практически неограниченной (характерный размер для 32-разрядных архитектур 232 = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный компьютер PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. Операционная система этого компьютера тем не менее поддерживала виртуальную память, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин " виртуальная память " означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (page fault).

Любая из трех ранее рассмотренных схем управления памятью – страничной, сегментной и сегментно-страничной – пригодна для организации виртуальной памяти. Чаще всего используется cегментно-страничная модель, которая является синтезом страничной модели и идеи сегментации. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти.

**43**

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью. Компоненты аппаратно-независимой подсистемы будут рассмотрены в следующей лекции.

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространствосущественно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС – сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае виртуальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

#### Страничная виртуальная память

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид (см. раздел "Структура таблицы страниц "), структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение(page fault) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающую его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

#### Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например, может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента.

На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

**44**

Организация таблицы страниц – один из ключевых элементов отображения адресов в страничной и сегментно-страничной схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 – read/write, 1– read only...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; бит ссылки, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 264 байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти (см. следующий раздел).

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем 232/212=220, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой многоуровневой таблицы страниц. Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 220 строк, разбивается на 210 таблиц второго уровня по 210 строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 210 строк. 32-разрядный адрес делится на 10-разрядное поле p1, 10-разрядное поле p2 и 12-разрядное смещение d. Поле p1 указывает на нужную строку в таблице первого уровня, поле p2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. [рис. 9.1](https://www.intuit.ru/studies/courses/2192/31/lecture/984?page=2#image.9.1)).

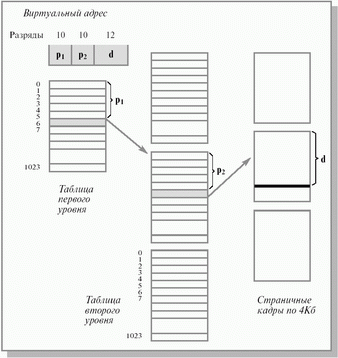


Рис. 9.1. Пример двухуровневой таблицы страниц

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт x 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих таблицах много меньше 220. Такой подход естественным образом обобщается на три и более уровней таблицы.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

**45**

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть таблицы страниц.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется ассоциативной памятью, иногда также употребляют термин буфер поиска трансляции (translation lookaside buffer – TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.

В начале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин "процент попаданий в кэш". Таким образом, hit ratio – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память – 20 нс. С 90% hit ratio среднее время определения адреса – 0,9x20+0,1x100 = 28 нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс "не видел" в ассоциативной памяти информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая ( ассоциативная память + таблица страниц ) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности, о чем говорилось во введении к предыдущей лекции.

**46**

Из материала предыдущей лекции следует, что отображение виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к виртуальной странице возникает исключительная ситуация "страничное нарушение" ( page fault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение". В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение.

Нас будет интересовать конкретный вариант страничного нарушения - обращение к отсутствующей странице, поскольку именно его обработка во многом определяет производительность страничной системы. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты страничных нарушений, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

* обслуживания исключительной ситуации ( page fault );
* чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);
* возобновления выполнения процесса, вызвавшего данный page fault.

Для решения первой и третьей задач ОС выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности page fault 5x10-7 оказывается достаточно, чтобы снизить производительность страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты page faults является одной из ключевых задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

**47**

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки - по запросу и с упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (placement policy) - в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто - в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (replacement policy) - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений . Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

**48**

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободный страничный кадр. В этом случае операционная система в соответствии с заложенными в нее критериями должна:

* найти некоторую занятую страницу основной памяти;
* переместить в случае надобности ее содержимое во внешнюю память;
* переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
* должным образом модифицировать необходимый элемент соответствующей таблицы страниц;
* продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы в таблице страниц). Бит модификации устанавливается компьютером, если хотя бы один байт был записан на страницу. При выборе кандидата на замещение проверяется бит модификации. Если битне установлен, нет необходимости переписывать данную страницу на диск, ее копия на диске уже имеется. Подобный метод также применяется к read-only-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее приложениеможет подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот список страниц иногда называют резидентным множеством процесса. В одном из следующих разделов рассмотрен вариант алгоритма подкачки, основанный на приведении резидентного множества в соответствие так называемому рабочему набору процесса.

Эффективность алгоритма обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих page faults. Эта последовательность называется строкой обращений (reference string). Мы можем генерировать строку обращений искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:

* для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка;
* несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Как уже говорилось, большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. Флаг ссылки (reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше флаг изменения (modify бит) устанавливается, если производится запись в эту страницу. Операционная система периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

Рассмотрим ряд алгоритмов замещения страниц.

#### Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но page fault происходит немедленно.

##### Аномалия Билэди (Belady)

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место page faults. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название "аномалии Билэди"или "аномалии FIFO ".

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO.

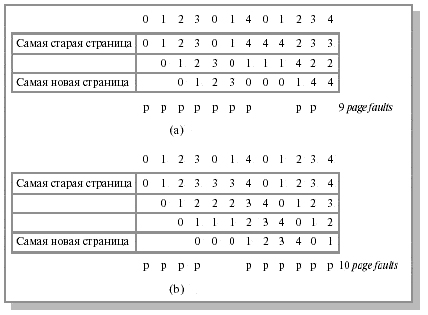


Рис. 10.1. Аномалия Билэди: (a) - FIFO с тремя страничными кадрами; (b) - FIFO с четырьмя страничными кадрами

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

#### Оптимальный алгоритм (OPT)

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту page faults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов - алгоритм SJF).

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

#### Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое - хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего, имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется least recently used алгоритм ( LRU ). Работа алгоритма проиллюстрирована на рис. [рис. 10.2](https://www.intuit.ru/studies/courses/2192/31/lecture/986?page=2#image.10.2). Сравнивая рис. 10.1 b и 10.2, можно увидеть, что использование LRU алгоритма позволяет сократить количество страничных нарушений.

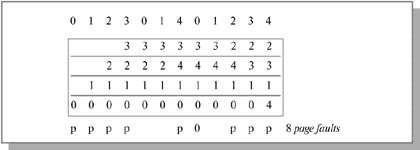


Рис. 10.2. Пример работы алгоритма LRU

LRU - хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

В [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)] рассмотрен вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют аномалии Билэди и называются стековыми (stack) алгоритмами.

#### Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, - алгоритм NFU(Not Frequently Used).

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему "забывать". Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

#### Другие алгоритмы

Для полноты картины можно упомянуть еще несколько алгоритмов.

Например, алгоритм Second-Chance - модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

В компьютере Macintosh использован алгоритм NRU (Not Recently-Used), где страница-"жертва" выбирается на основе анализа битов модификации и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/VMS и Mach.

**49**

Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобождением всех кадров? Что следует понимать под достаточным количеством кадров?

#### Трешинг (Thrashing)

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует page faults. Высокая частота страничных нарушений называется трешинг (thrashing, иногда употребляется русский термин "пробуксовка", см. [рис. 10.3](https://www.intuit.ru/studies/courses/2192/31/lecture/986?page=3#image.10.3)). Процесс находится в состоянии трешинга, если при его работе больше времени уходит на подкачку страниц, нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.

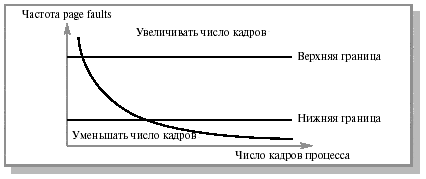


Рис. 10.3. Частота page faults в зависимости от количества кадров, выделенных процессу

Часто результатом трешинга является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгоритме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением степени мультипрограммирования, что приводит к еще большему трешингу и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа трешинга возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие трешинга, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Итак, трешинг - это высокая частота страничных нарушений. Hеобходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту page faults, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой page faults.

Для предотвращения трешинга требуется выделять процессу столько кадров, сколько ему нужно. Hо как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс реально использует. Для решения этой задачи Деннинг использовал модель рабочего множества, которая основана на применении принципа локальности.

#### Модель рабочего множества

Рассмотрим поведение реальных процессов.

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает page fault, требующий подкачки порции кода. Следующий page fault происходит при локализации глобальных переменных и еще один - при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, page faults возникают редко.

Таким образом, существует набор страниц (P1, P2, ... Pn), активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества page faults. Этот набор страниц называется рабочим множеством W(t,T) ( working set ) процесса. Число страниц в рабочем множестве определяется параметром Т, является неубывающей функцией T и относительно невелико. Иногда T называют размером окна рабочего множества, через которое ведется наблюдение за процессом (см. [рис. 10.4](https://www.intuit.ru/studies/courses/2192/31/lecture/986?page=3#image.10.4)).

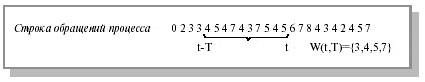


Рис. 10.4. Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Hапример, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными программы. Если процессу выделять меньше кадров, чем ему требуется для поддержки рабочего множества, он будет находиться в состоянии трешинга.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени (t-T, t) программа обращалась к страницам W(t,T), то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (t, t+T). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества - его размер. ОС должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования ti (для страницы с номером i ), то принадлежность i-й страницы к рабочему набору, определяемому параметром T в момент времени t будет выражаться неравенством: t-T < ti < t. Алгоритм выталкивания страниц WSClock, использующий информацию о рабочем наборе процесса, описан в [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)].

Другой способ реализации данного подхода может быть основан на отслеживании количества страничных нарушений, вызываемых процессом. Если процесс часто генерирует page faults и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо стратегией замещения страниц. Несмотря на то что система при этом может пробуксовывать в моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

**50**

Подсистема виртуальной памяти работает производительно при наличии резерва свободных страничных кадров. Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия трешинга, реализованы в составе фоновых процессов (их часто называют демонами или сервисами), которые периодически "просыпаются" и инспектируют состояние памяти. Если свободных кадров оказывается мало, они могут сменить стратегию замещения. Их задача - поддерживать систему в состоянии наилучшей производительности.

Примером такого рода процесса может быть фоновый процесс - сборщик страниц, реализующий облегченный вариант алгоритма откачки, основанный на использовании рабочего набора и применяемый во многих клонах ОС Unix (см., например,[[Bach, 1986](https://www.intuit.ru/studies/courses/2192/31/literature#literature.1)]). Данный демонпроизводит откачку страниц, не входящих в рабочие наборы процессов. Он начинает активно работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога, и пытается выталкивать страницы в соответствии с собственной стратегией.

Но если возникает требование страницы в условиях, когда список свободных страниц пуст, то начинает работать механизм свопинга, поскольку простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing, и разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится в очередь на выгрузку в расчете на то, что после завершения выгрузки хотя бы одного из процессов свободной памяти уже может быть достаточно.

В ОС Windows 2000 аналогичную роль играет менеджер балансного набора (Working set manager), который вызывается раз в секунду или тогда, когда размер свободной памяти опускается ниже определенного предела, и отвечает за суммарную политику управления памятью и поддержку рабочих множеств.

**51**

#### Имена файлов

Файлы представляют собой абстрактные объекты. Их задача - хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает файл, он дает ему имя. После завершения процесса файл продолжает существовать и через свое имя может быть доступен другим процессам.

Правила именования файлов зависят от ОС. Многие ОС поддерживают имена из двух частей (имя+расширение), например progr.c ( файл, содержащий текст программы на языке Си) или autoexec.bat ( файл, содержащий команды интерпретатора командного языка). Тип расширения файла позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно ОС накладывают некоторые ограничения, как на используемые в имени символы, так и на длину имени файла. В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов).

#### Типы файлов

Важный аспект организации файловой системы и ОС - следует ли поддерживать и распознавать типы файлов. Если да, то это может помочь правильному функционированию ОС, например не допустить вывода на принтер бинарного файла.

Основные типы файлов: регулярные (обычные) файлы и директории (справочники, каталоги ). Обычные файлы содержат пользовательскую информацию. Директории - системные файлы, поддерживающие структуру файловой системы. В каталоге содержится перечень входящих в него файлов и устанавливается соответствие между файлами и их характеристиками ( атрибутами ). Мы будем рассматривать директории ниже.

Напомним, что хотя внутри подсистемы управления файлами обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла в виде линейной последовательности байтов. Такое представление позволяет использовать абстракцию файла при работе с внешними устройствами, при организации межпроцессных взаимодействий и т. д. Так, например, клавиатура обычно рассматривается как текстовый файл, из которого компьютер получает данные в символьном формате. Поэтому иногда к файлам приписывают другие объекты ОС, например специальные символьные файлы и специальные блочные файлы, именованные каналы и сокеты, имеющие файловый интерфейс. Эти объекты рассматриваются в других разделах данного курса.

Далее речь пойдет главным образом об обычных файлах.

Обычные (или регулярные) файлы реально представляют собой набор блоков (возможно, пустой) на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную (бинарную) информацию.

Текстовые файлы содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой тип файлов - нетекстовые, или бинарные, файлы. Обычно они имеют некоторую внутреннюю структуру. Например, исполняемый файл в ОС Unix имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. ОС выполняет файл, только если он имеет нужный формат. Другим примером бинарного файла может быть архивный файл. Типизация файлов не слишком строгая.

Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями .c, .pas, .txt - ASCII-файлы, файлы с расширениями .exe - выполнимые, файлы с расширениями .obj, .zip - бинарные и т. д.

#### Атрибуты файлов

Кроме имени ОС часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т. д. Эти другие характеристики файлов называются атрибутами . Список атрибутов в разных ОС может варьироваться. Обычно он содержит следующие элементы: основную информацию (имя, тип файла ), адресную информацию (устройство, начальный адрес, размер), информацию об управлении доступом (владелец, допустимые операции) и информацию об использовании (даты создания, последнего чтения, модификации и др.).

Список атрибутов обычно хранится в структуре директорий (см. следующую лекцию) или других структурах, обеспечивающих доступ к данным файла.

**52**

Программист воспринимает файл в виде набора однородных записей. Запись - это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются - для ввода. Вопросы распределения блоков внешней памяти между файлами рассматриваются в следующей лекции.

ОС поддерживают несколько вариантов структуризации файлов.

#### Последовательный файл

Простейший вариант - так называемый последовательный файл. То есть файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой неструктурированную последовательность байтов.

Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Такой способ доступа называется последовательным (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewind).

#### Файл прямого доступа

В реальной практике файлы хранятся на устройствах прямого (random) доступа, например на дисках, поэтому содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри файла.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих файлу. О связи относительного номера блока с абсолютным его номером на диске рассказывается в следующей лекции.

Естественно, что в этом случае для доступа к середине файла просмотр всего файла с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek. Файл, байты которого могут быть считаны в произвольном порядке, называется файлом прямого доступа .

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, - наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. В большинстве языков высокого уровня предусмотрены операторы посимвольной пересылки данных в файл или из него.

Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать файлы. В частности, многие СУБД хранят свои базы данных в обычных файлах.

#### Другие формы организации файлов

Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних ОС, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании - хранение файла в виде последовательности записей фиксированной длины, каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком. Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В ОС CP/M файлы были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

Другой способ представления файлов - последовательность записей переменной длины, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (см. [рис. 11.1](https://www.intuit.ru/studies/courses/2192/31/lecture/988?page=1#image.11.1)). Базисная операция в данном случае - считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям последовательного файла называется индексно-последовательным.

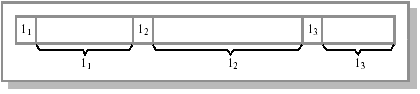


Рис. 11.1. Файл как последовательность записей переменной длины

В некоторых системах ускорение доступа к файлу обеспечивается конструированием индекса файла. Индекс обычно хранится на том же устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются индексированными, а метод доступа к ним - доступ с использованием индекса.

Предположим, у нас имеется большой несортированный файл, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента. [Рис. 11.2](https://www.intuit.ru/studies/courses/2192/31/lecture/988?page=1#image.11.2)иллюстрирует решение данной проблемы - организацию метода доступа к файлу с использованием индекса.

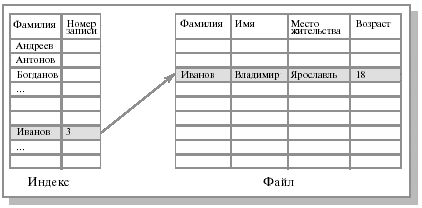


Рис. 11.2. Пример организации индекса для последовательного файла

Следует отметить, что почти всегда главным фактором увеличения скорости доступа является избыточность данных.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде ОС (Unix и некоторых других, см. следующую лекцию), может служить другим примером организации индекса.

В этом случае ОС использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Для больших файлов индекс может быть слишком велик. В этом случае создают индекс для индексного файла (блоки промежуточного уровня или блоки косвенной адресации).

**53**

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти. Рассмотрим в качестве примера основные файловые операции ОС Unix [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)].

* Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.
* Удаление файла и освобождение занимаемого им дискового пространства.
* Открытие файла. Перед использованием файла процесс должен его открыть. Цель данного системного вызова - разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файладля быстрого доступа к его данным. Открытие файла является процедурой создания дескриптора или управляющего блока файла. Дескриптор (описатель) файла хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках программирования, под дескриптором понимается альтернативное имя файла или указатель на описание файла в таблице открытых файлов, используемый при последующей работе с файлом . Например, на языке Cи операция открытия файлаfd=open(pathname,flags,modes); возвращает дескриптор fd, который может быть задействован при выполнении операций чтения ( read(fd,buffer,count); ) или записи.
* Закрытие файла. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.
* Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать текущую позицию.
* Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.
* Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции, например переименование файла, получение атрибутов файла и т. д.

Существует два способа выполнить последовательность действий над файлами [[Олифер, 2001](https://www.intuit.ru/studies/courses/2192/31/literature#literature.23)].

В первом случае для каждой операции выполняются как универсальные, так и уникальные действия (схема stateless). Например, последовательность операций может быть такой: open, read1, close, ... open, read2, close, ... open, read3, close.

Альтернативный способ - это когда универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия. В этом случае последовательность вышеприведенных операций будет выглядеть так: open, read1, ... read2, ... read3, close.

Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой операции становятся независимыми от результатов предыдущей операции; поэтому он иногда применяется в распределенных файловых системах (например, Sun NFS).

**54**

Количество файлов на компьютере может быть большим. Отдельные системы хранят тысячи файлов, занимающие сотни гигабайт дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой - каталогов (или директорий ).

Каждый каталог содержит список каталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории (см., например, [рис.11.3](https://www.intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.3)).

Число директорий зависит от системы. В ранних ОС имелась только одна корневая директория, затем появились директории для пользователей (по одной директории на пользователя). В современных ОС используется произвольная структура дерева директорий.

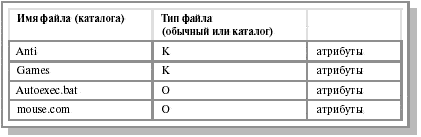


Рис. 11.3. Директории

Таким образом, файлы на диске образуют иерархическую древовидную структуру (см. [рис. 11.4](https://www.intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.4)).

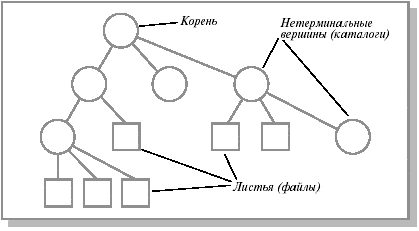


Рис. 11.4. Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на [рис. 11.4](https://www.intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.4), наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершинойили листом (в данном случае является файлом ). Нелистовые вершины - справочники или каталоги содержат списки листовых и нелистовых вершин. Путь от корня к файлу однозначно определяет файл.

Подобные древовидные структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень - вершина, не имеющая входящих ребер. Как мы увидим в следующей лекции, связывание файлов, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах, могут совпадать. Для того чтобы однозначно определить файл по его имени (избежать коллизии имен), принято именовать файл так называемым абсолютным или полным именем (pathname), состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл. То есть полное имя включает цепочку имен - путь к файлу, например /usr/games/doom. Такие имена уникальны. Компоненты пути разделяют различными символами: "/" (слэш) в Unix или обратными слэшем в MS-DOS (в Multics - ">"). Таким образом, использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени - относительный путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом. Тогда на файлы в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочем каталоге. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам. Рассмотрим для примера путь/usr/linux/progr.c. Алгоритм одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая директория. Затем находится компонент пути usr, т. е. в корневой директории ищется файл /usr. Исследуя этот файл, система понимает, что данный файл является каталогом, и блоки его данных рассматривает как список файлов и ищет следующий компонент linux в нем. Из строки для linux находится файл, соответствующий компоненту usr/linux/. Затем находится компонент progr.c, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла.

Отклонение от типовой обработки компонентов pathname может возникнуть в том случае, когда этот компонент является не обычным каталогом с соответствующим ему индексным узлом и списком файлов, а служит точкой связывания (принято говорить "точкой монтирования") двух файловых архивов. Этот случай рассмотрен в следующей лекции.

Многие прикладные программы работают с файлами, находящимися в текущей директории, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать каталоги, содержащие различные программные пакеты. Для реализации этой возможности в большинстве ОС, поддерживающих иерархическую структуру директорий, используется обозначение " ." - для текущей директории и " .." - для родительской.

#### Разделы диска. Организация доступа к архиву файлов.

Задание пути к файлу в файловых системах некоторых ОС отличается тем, с чего начинается эта цепочка имен.

В современных ОС принято разбивать диски на логические диски (это низкоуровневая операция), иногда называемые разделами(partitions). Бывает, что, наоборот, объединяют несколько физических дисков в один логический диск (например, это можно сделать в ОС Windows NT). Поэтому в дальнейшем изложении мы будем игнорировать проблему физического выделения пространства для файлов и считать, что каждый раздел представляет собой отдельный (виртуальный) диск. Диск содержит иерархическую древовидную структуру, состоящую из набора файлов, каждый из которых является хранилищем данных пользователя, и каталогов или директорий (то есть файлов, которые содержат перечень других файлов, входящих в состав каталога ), необходимых для хранения информации о файлах системы.

В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался на одном диске (разделе диска). В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск (буквы диска). Например, c:\util\nu\ndd.exe. Такой способ именования используется в файловых системах DEC и Microsoft.

В других системах (Multics) вся совокупность файлов и каталогов представляет собой единое дерево. Сама система, выполняя поиск файлов по имени, начиная с корня, требовала установки необходимых дисков.

В ОС Unix предполагается наличие нескольких архивов файлов, каждый на своем разделе, один из которых считается корневым. После запуска системы можно "смонтировать" корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему.

Технически это осуществляется с помощью создания в корневой файловой системе специальных пустых каталогов (см. также следующую лекцию). Специальный системный вызов mount ОС Unix позволяет подключить к одному из этих пустых каталогов корневой каталог указанного архива файлов. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Задачей ОС является беспрепятственный проход точки монтирования при получении доступа к файлу по цепочке имен. Если учесть, что обычно монтирование файловой системы производится при загрузке системы, пользователи ОС Unix обычно и не задумываются о происхождении общей файловой системы.

**55**

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории - это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС. Напомним, что операции над каталогами являются прерогативой ОС, то есть пользователь не может, например, выполнить запись в каталог начиная с текущей позиции. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с каталогами [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)].

* Создание директории. Вновь созданная директория включает записи с именами ' .' и ' ..', однако считается пустой.
* Удаление директории. Удалена может быть только пустая директория.
* Открытие директории для последующего чтения. Hапример, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.
* Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.
* Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может использоваться системный вызов Read, но в этом случае от программиста потребуется знание внутренней структуры директории.
* Получение списка файлов в каталоге.
* Переименование. Имена директорий можно менять, как и имена файлов.
* Создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент.
* Удаление файла. Удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление файлов предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например связанных с защитой информации.

**56**

Общие проблемы безопасности ОС рассмотрены в лекциях 15-16. Информация в компьютерной системе должна быть защищена как от физического разрушения (reliability), так и от несанкционированного доступа (protection).

Здесь мы коснемся отдельных аспектов защиты, связанных с контролем доступа к файлам.

#### Контроль доступа к файлам

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

#### Списки прав доступа

Hаиболее общий подход к защите файлов от несанкционированного использования - сделать доступ зависящим от идентификатора пользователя, то есть связать с каждым файлом или директорией список прав доступа (access control list), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа - список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список. У такой техники есть два нежелательных следствия.

* Конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы.
* Запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

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

* Владелец (Owner).
* Группа (Group). Hабор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему.
* Остальные (Univers).

Это позволяет реализовать конденсированную версию списка прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в Unix операции чтения, записи и исполнения контролируются при помощи 9 бит (rwxrwxrwx).

**57**

Система хранения данных на дисках может быть структурирована следующим образом (см. [рис. 12.1](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=1#image.12.1)).

Нижний уровень - оборудование. Это в первую очередь магнитные диски с подвижными головками - основные устройства внешней памяти, представляющие собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов) таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно непосредственно получить доступ к любому блоку (организовать прямой доступ к файлам).

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая системой ввода-вывода (см. лекцию 13). Система ввода-вывода предоставляет в распоряжение более высокоуровневого компонента ОС - файловой системы - используемое дисковое пространство в виде непрерывной последовательности блоков фиксированного размера. Система ввода-вывода имеет дело с физическими блоками диска, которые характеризуются адресом, например диск 2, цилиндр 75, сектор 11. Файловая система имеет дело с логическими блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В структуре системы управления файлами можно выделить базисную подсистему, которая отвечает за выделение дискового пространства конкретным файлам, и более высокоуровневую логическую подсистему, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации, исходя из символического имени файла. Она также ответственна за авторизацию доступа к файлам (см. лекции 11 и 16).

Стандартный запрос на открытие (open) или создание (create) файла поступает от прикладной программы к логической подсистеме. Логическая подсистема, используя структуру директорий, проверяет права доступа и вызывает базовую подсистему для получения доступа к блокам файла. После этого файл считается открытым, он содержится в таблице открытых файлов, и прикладная программа получает в свое распоряжение дескриптор (или handle в системах Microsoft) этого файла. Дескриптор файла является ссылкой на файл в таблице открытых файлов и используется в запросах прикладной программы на чтение-запись из этого файла. Запись в таблице открытых файлов указывает через систему выделения блоков диска на блоки данного файла. Если к моменту открытия файл уже используется другим процессом, то есть содержится в таблице открытых файлов, то после проверки прав доступа к файлу может быть организован совместный доступ. При этом новому процессу также возвращается дескриптор - ссылка на файл в таблице открытых файлов. Далее в тексте подробно проанализирована работа наиболее важных системных вызовов.

**58**

#### Методы выделения дискового пространства

Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска. В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

##### Выделение непрерывной последовательностью блоков

Простейший способ - хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию.

Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Этот способ распространен мало, и вот почему. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения блока нужного размера из списка свободных блоков. Типовыми решениями этой задачи являются стратегии первого подходящего, наиболее подходящего и наименее подходящего (сравните с проблемой выделения памяти в методе с динамическим распределением). Как и в случае выделения нужного объема оперативной памяти в схеме с динамическими разделами (см. лекцию 8), метод страдает от внешней фрагментации, в большей или меньшей степени, в зависимости от размера диска и среднего размера файла.

Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать, особенно в тех случаях, когда размер файла меняется. Если места не хватило, то пользовательская программа может быть приостановлена с учетом выделения дополнительного места для файла при последующем рестарте. Некоторые ОС используют модифицированный вариант непрерывного выделения - основные блоки файла + резервные блоки. Однако с выделением блоков из резерва возникают те же проблемы, так как приходится решать задачу выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или "сборка мусора", цель которой состоит в объединении свободных участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для стационарных файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

##### Связный список

Внешняя фрагментация - основная проблема рассмотренного выше метода - может быть устранена за счет представления файла в виде связного списка блоков диска. Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла - EOF). Каждый блок содержит указатель на следующий блок (см. [рис. 12.2](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=2#image.12.2)).

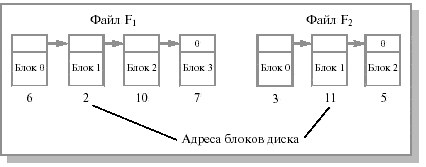


Рис. 12.2. Хранение файла в виде связного списка дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.

Связное выделение имеет, однако, несколько существенных недостатков.

Во-первых, при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i-1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.

Во-вторых, данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.

Наконец, для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

##### Таблица отображения файлов

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов ( FAT - file allocation table ) (см. [рис. 12.3](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=2#image.12.3)). Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.)

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

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.

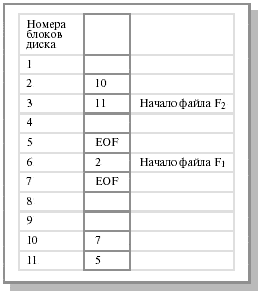


Рис. 12.3. Метод связного списка с использованием таблицы в оперативной памяти

##### Индексные узлы

Наиболее распространенный метод выделения файлу блоков диска - связать с каждым файлом небольшую таблицу, называемую индексным узлом ( i-node ), которая перечисляет атрибуты и дисковые адреса блоков файла (см. [рис 12.4](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=2#image.12.4)). Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

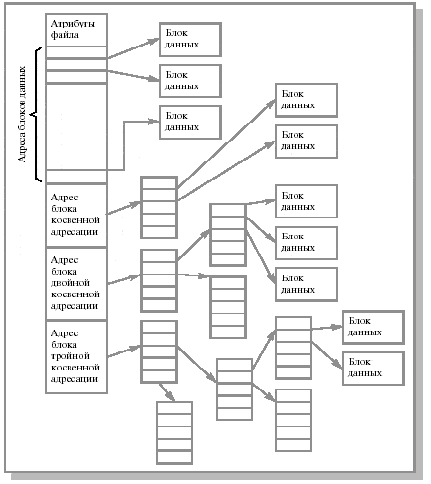
****

Рис. 12.4. Структура индексного узла

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

**59**

##### Учет при помощи организации битового вектора

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

Главное преимущество этого подхода состоит в том, что он относительно прост и эффективен при нахождении первого свободного блока или n последовательных блоков на диске. Многие компьютеры имеют инструкции манипулирования битами, которые могут использоваться для этой цели. Hапример, компьютеры семейств Intel и Motorola имеют инструкции, при помощи которых можно легко локализовать первый единичный бит в слове.

Описываемый метод учета свободных блоков используется в Apple Macintosh.

Несмотря на то что размер описанного битового вектора наименьший из всех возможных структур, даже такой вектор может оказаться большого размера. Поэтому данный метод эффективен, только если битовый вектор помещается в памяти целиком, что возможно лишь для относительно небольших дисков. Например, диск размером 4 Гбайт с блоками по 4 Кбайт нуждается в таблице размером 128 Кбайт для управления свободными блоками. Иногда, если битовый вектор становится слишком большим, для ускорения поиска в нем его разбивают на регионы и организуют резюмирующие структуры данных, содержащие сведения о количестве свободных блоков для каждого региона.

##### Учет при помощи организации связного списка

Другой подход - связать в список все свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте диска, попутно кэшируя в памяти эту информацию.

Подобная схема не всегда эффективна. Для трассирования списка нужно выполнить много обращений к диску. Однако, к счастью, нам необходим, как правило, только первый свободный блок.

Иногда прибегают к модификации подхода связного списка, организуя хранение адресов n свободных блоков в первом свободном блоке. Первые n-1 этих блоков действительно используются. Последний блок содержит адреса других n блоков и т. д.

Существуют и другие методы, например, свободное пространство можно рассматривать как файл и вести для него соответствующий индексный узел.

#### Размер блока

Размер логического блока играет важную роль. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, таким образом, файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации (каждый файл занимает целое число блоков, и в среднем половина последнего блока пропадает) снижается процент полезного дискового пространства.

Для систем со страничной организацией памяти характерна сходная проблема с размером страницы.

Проведенные исследования показали, что большинство файлов имеют небольшой размер. Например, в Unix приблизительно 85% файлов имеют размер менее 8 Кбайт и 48% - менее 1 Кбайта.

Можно также учесть, что в системах с виртуальной памятью желательно, чтобы единицей пересылки диск-память была страница (наиболее распространенный размер страниц памяти - 4 Кбайта). Отсюда обычный компромиссный выбор блока размером 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт.

#### Структура файловой системы на диске

Рассмотрение методов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структура служебных данных типовой файловой системы, например Unix, на одном из разделов диска, таким образом, может состоять из четырех основных частей (см. [рис. 12.5](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=3#image.12.5)).

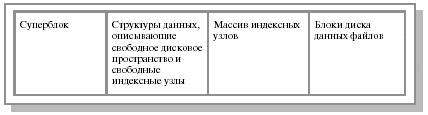


Рис. 12.5. Примерная структура файловой системы на диске

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

* тип файловой системы;
* размер файловой системы в блоках;
* размер массива индексных узлов ;
* размер логического блока.

Описанные структуры данных создаются на диске в результате его форматирования (например, утилитами format, makefs и др.). Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

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

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

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

**60**

Так же как файл должен быть открыт перед использованием, и файловая система, хранящаяся на разделе диска, должна быть смонтирована, чтобы стать доступной процессам системы.

Функция mount (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а функция umount (демонтировать) выключает файловую систему из иерархии. Функция mount, таким образом, дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков.

Процедура монтирования состоит в следующем. Пользователь (в Unix это суперпользователь) сообщает ОС имя устройства и место в файловой структуре (имя пустого каталога), куда нужно присоединить файловую систему (точка монтирования ) (см. [рис. 12.9](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=5#image.12.9) и [рис. 12.10](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=5#image.12.10)). Hапример, в ОС Unix библиотечный вызов mount имеет вид:

mount(special pathname,directory pathname,options);

где special pathname - имя специального файла устройства (в общем случае имя раздела), соответствующего дисковому разделу с монтируемой файловой системой, directory pathname - каталог в существующей иерархии, где будет монтироваться файловая система(другими словами, точка или место монтирования ), а options указывает, следует ли монтировать файловую систему "только для чтения" (при этом не будут выполняться такие функции, как write и create, которые производят запись в файловую систему). Затем ОС должна убедиться, что устройство содержит действительную файловую систему ожидаемого формата с суперблоком, списком индексов и корневым индексом.

Некоторые ОС осуществляют монтирование автоматически, как только встретят диск в первый раз (жесткие диски на этапе загрузки, гибкие - когда они вставлены в дисковод), ОС ищет файловую систему на устройстве. Если файловая система на устройстве имеется, она монтируется на корневом уровне, при этом к цепочке имен абсолютного имени файла (pathname) добавляется буква раздела.

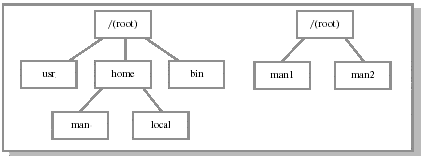


Рис. 12.9. Две файловые системы до монтирования

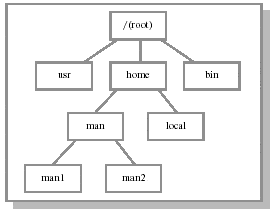


Рис. 12.10. Общая файловая система после монтирования

Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информацияо вновь смонтированном устройстве, о его суперблоке и корневом каталоге, а также сведения о точке монтирования. Для устранения потенциально опасных побочных эффектов число линков (см. следующий раздел) к каталогу - точке монтирования - должно быть равно 1. Занесение информации в таблицу монтирования производится немедленно, поскольку может возникнуть конфликт между двумя процессами. Например, если монтирующий процесс приостановлен для открытия устройства или считывания суперблока файловой системы, а тем временем другой процесс может попытаться смонтировать файловую систему.

Наличие в логической структуре файлового архива точек монтирования требует аккуратной реализации алгоритмов, осуществляющих навигацию по каталогам. Точку монтирования можно пересечь двумя способами: из файловой системы, где производится монтирование, в файловую систему, которая монтируется (в направлении от глобального корня к листу), и в обратном направлении. Алгоритмы поиска файлов должны предусматривать ситуации, в которых очередной компонент пути к файлу является точкой монтирования, когда вместо анализа индексного узла очередной директории приходится осуществлять обработку суперблока монтированной системы.

**61**

Иерархическая организация, положенная в основу древовидной структуры файловой системы современных ОС, не предусматривает выражения отношений, в которых потомки связываются более чем с одним предком. Такая негибкость частично устраняется возможностью реализации связывания файлов или организации линков ( link ).

Ядро позволяет пользователю связывать каталоги, упрощая написание программ, требующих пересечения дерева файловой системы (см. [рис. 12.11](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=5#image.12.11)). Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл). Например, выполняемый файлтрадиционного текстового редактора ОС Unix vi обычно может вызываться под именами ex, edit, vi, view и vedit файловой системы. Соединение между директорией и разделяемым файлом называется "связью" или "ссылкой" ( link ). Дерево файловой системы превращается в циклический граф.

Это удобно, но создает ряд дополнительных проблем.

Простейший способ реализовать связывание файла - просто дублировать информацию о нем в обеих директориях. При этом, однако, может возникнуть проблема совместимости в случае, если владельцы этих директорий попытаются независимо друг от друга изменить содержимое файла. Например, в ОС CP/M запись в директории о файле непосредственно содержит адреса дисковых блоков. Поэтому копии тех же дисковых адресов должны быть сделаны и в другой директории, куда файл линкуется. Если один из пользователей что-то добавляет к файлу, новые блоки будут перечислены только у него в директории и не будут "видны" другому пользователю.

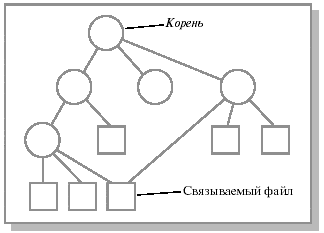


Рис. 12.11. Структура файловой системы с возможностью связывания файла с новым именем

Проблема такого рода может быть решена двумя способами. Первый из них - так называемая жесткая связь (hard link). Если блоки данных файла перечислены не в директории, а в небольшой структуре данных (например, в индексном узле ), связанной собственно с файлом, то второй пользователь может связаться непосредственно с этой, уже существующей структурой.

Альтернативное решение - создание нового файла, который содержит путь к связываемому файлу. Такой подход называется символическойлинковкой (soft или symbolic link). При этом в соответствующем каталоге создается элемент, в котором имени связи сопоставляется некоторое имя файла (этот файл даже не обязан существовать к моменту создания символической связи ). Для символической связи может создаваться отдельный индексный узел и даже заводиться отдельный блок данных для хранения потенциально длинного имени файла.

Каждый из этих методов имеет свои минусы. В случае жесткой связи возникает необходимость поддержки счетчика ссылок на файл для корректной реализации операции удаления файла. Например, в Unix такой счетчик является одним из атрибутов, хранящихся в индексном узле. Удаление файла одним из пользователей уменьшает количество ссылок на файл на 1. Реальное удаление файла происходит, когда число ссылок на файл становится равным 0.

В случае символической линковки такая проблема не возникает, так как только реальный владелец имеет ссылку на индексный узел файла. Если собственник удаляет файл, то он разрушается, и попытки других пользователей работать с ним закончатся провалом. Удаление символического линка на файл никак не влияет. Проблема организации символической связи - потенциальное снижение скорости доступа к файлу. Файл символического линка хранит путь к файлу, содержащий список вложенных директорий, для прохождения по которому необходимо осуществить несколько обращений к диску.

Символический линк имеет то преимущество, что он может использоваться для организации удобного доступа к файлам удаленных компьютеров, если, например, добавить к пути сетевой адрес удаленной машины.

Циклический граф - структура более гибкая, нежели простое дерево, но работа с ней требует большой аккуратности. Поскольку теперь к файлу существует несколько путей, программа поиска файла может найти его на диске несколько раз. Простейшее практическое решение данной проблемы - ограничить число директорий при поиске. Полное устранение циклов при поиске - довольно трудоемкая процедура, выполняемая специальными утилитами и связанная с многократной трассировкой директорий файловой системы.

**62**

#### Целостность файловой системы

Важный аспект надежной работы файловой системы - контроль ее целостности. В результате файловых операций блоки диска могут считываться в память, модифицироваться и затем записываться на диск. Причем многие файловые операции затрагивают сразу несколько объектов файловой системы. Например, копирование файла предполагает выделение ему блоков диска, формирование индексного узла, изменение содержимого каталога и т. д. В течение короткого периода времени между этими шагами информация в файловой системе оказывается несогласованной.

И если вследствие непредсказуемой остановки системы на диске будут сохранены изменения только для части этих объектов (нарушена атомарность файловой операции), файловая система на диске может быть оставлена в несовместимом состоянии. В результате могут возникнуть нарушения логики работы с данными, например появиться "потерянные" блоки диска, которые не принадлежат ни одному файлу и в то же время помечены как занятые, или, наоборот, блоки, помеченные как свободные, но в то же время занятые (на них есть ссылка в индексном узле ) или другие нарушения.

В современных ОС предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и затем полностью или частично восстановить ее целостность.

##### Порядок выполнения операций

Очевидно, что для правильного функционирования файловой системы значимость отдельных данных неравноценна. Искажение содержимого пользовательских файлов не приводит к серьезным (с точки зрения целостности файловой системы) последствиям, тогда как несоответствия в файлах, содержащих управляющую информацию (директории, индексные узлы, суперблок и т. п.), могут быть катастрофическими. Поэтому должен быть тщательно продуман порядок выполнения операций со структурами данных файловой системы.

Рассмотрим пример создания жесткой связи для файла [[Робачевский, 1999](https://www.intuit.ru/studies/courses/2192/31/literature#literature.33)]. Для этого файловой системе необходимо выполнить следующие операции:

* создать новую запись в каталоге, указывающую на индексный узел файла;
* увеличить счетчик связей в индексном узле.

Если аварийный останов произошел между 1-й и 2-й операциями, то в каталогах файловой системы будут существовать два имени файла, адресующих индексный узел со значением счетчика связей, равному 1. Если теперь будет удалено одно из имен, это приведет к удалению файла как такового. Если же порядок операций изменен и, как прежде, останов произошел между первой и второй операциями, файл будет иметь несуществующую жесткую связь, но существующая запись в каталоге будет правильной. Хотя это тоже является ошибкой, но ее последствия менее серьезны, чем в предыдущем случае.

##### Журнализация

Другим средством поддержки целостности является заимствованный из систем управления базами данных прием, называемый журнализация(иногда употребляется термин "журналирование" ). Последовательность действий с объектами во время файловой операции протоколируется, и если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала операции. Подобная избыточность может стоить дорого, но она оправдана, так как в случае отказа позволяет реконструировать потерянные данные.

Для отката необходимо, чтобы для каждой протоколируемой в журнале операции существовала обратная. Например, для каталогов и реляционных СУБД это именно так. По этой причине, в отличие от СУБД, в файловых системах протоколируются не все изменения, а лишь изменения метаданных ( индексных узлов, записей в каталогах и др.). Изменения в данных пользователя в протокол не заносятся. Кроме того, если протоколировать изменения пользовательских данных, то этим будет нанесен серьезный ущерб производительности системы, поскольку кэширование потеряет смысл.

Журнализация реализована в NTFS, Ext3FS, ReiserFS и других системах. Чтобы подчеркнуть сложность задачи, нужно отметить, что существуют не вполне очевидные проблемы, связанные с процедурой отката. Например, отмена одних изменений может затрагивать данные, уже использованные другими файловыми операциями. Это означает, что такие операции также должны быть отменены. Данная проблема получила название каскадного отката транзакций [Брукшир, 2001]

##### Проверка целостности файловой системы при помощи утилит

Если же нарушение все же произошло, то для устранения проблемы несовместимости можно прибегнуть к утилитам (fsck, chkdsk, scandisk и др.), которые проверяют целостность файловой системы. Они могут запускаться после загрузки или после сбоя и осуществляют многократное сканирование разнообразных структур данных файловой системы в поисках противоречий.

Возможны также эвристические проверки. Hапример, нахождение индексного узла, номер которого превышает их число на диске или поиск в пользовательских директориях файлов, принадлежащих суперпользователю.

К сожалению, приходится констатировать, что не существует никаких средств, гарантирующих абсолютную сохранность информации в файлах, и в тех ситуациях, когда целостность информации нужно гарантировать с высокой степенью надежности, прибегают к дорогостоящим процедурам дублирования.

#### Управление "плохими" блоками

Наличие дефектных блоков на диске - обычное дело. Внутри блока наряду с данными хранится контрольная сумма данных. Под "плохими" блоками обычно понимают блоки диска, для которых вычисленная контрольная сумма считываемых данных не совпадает с хранимой контрольной суммой. Дефектные блоки обычно появляются в процессе эксплуатации. Иногда они уже имеются при поставке вместе со списком, так как очень затруднительно для поставщиков сделать диск полностью свободным от дефектов. Рассмотрим два решения проблемы дефектных блоков - одно на уровне аппаратуры, другое на уровне ядра ОС.

Первый способ - хранить список плохих блоков в контроллере диска. Когда контроллер инициализируется, он читает плохие блоки и замещает дефектный блок резервным, помечая отображение в списке плохих блоков. Все реальные запросы будут идти к резервному блоку. Следует иметь в виду, что при этом механизм подъемника (наиболее распространенный механизм обработки запросов к блокам диска) будет работать неэффективно. Дело в том, что существует стратегия очередности обработки запросов к диску (подробнее см. лекцию "ввод-вывод"). Стратегия диктует направление движения считывающей головки диска к нужному цилиндру. Обычно резервные блоки размещаются на внешних цилиндрах. Если плохой блок расположен на внутреннем цилиндре и контроллер осуществляет подстановку прозрачным образом, то кажущееся движение головки будет осуществляться к внутреннему цилиндру, а фактическое - к внешнему. Это является нарушением стратегии и, следовательно, минусом данной схемы.

Решение на уровне ОС может быть следующим. Прежде всего, необходимо тщательно сконструировать файл, содержащий дефектные блоки. Тогда они изымаются из списка свободных блоков. Затем нужно каким-то образом скрыть этот файл от прикладных программ.

**63**

Поскольку обращение к диску - операция относительно медленная, минимизация количества таких обращений - ключевая задача всех алгоритмов, работающих с внешней памятью. Наиболее типичная техника повышения скорости работы с диском - кэширование.

#### Кэширование

Кэш диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска (см. [рис. 12.12](https://www.intuit.ru/studies/courses/2192/31/lecture/990?page=8#image.12.12)). Если имеется запрос на чтение/запись блока диска, то сначала производится проверка на предмет наличия этого блока в кэше. Если блок в кэше имеется, то запрос удовлетворяется из кэша, в противном случае запрошенный блок считывается в кэш с диска. Сокращение количества дисковых операций оказывается возможным вследствие присущего ОС свойства локальности (о свойстве локальности много говорилось в лекциях, посвященных описанию работы системы управления памятью).

Аккуратная реализация кэширования требует решения нескольких проблем.

Во-первых, емкость буфера кэша ограничена. Когда блок должен быть загружен в заполненный буфер кэша, возникает проблема замещения блоков, то есть отдельные блоки должны быть удалены из него. Здесь работают те же стратегии и те же FIFO, Second Chance и LRU-алгоритмы замещения, что и при выталкивании страниц памяти.

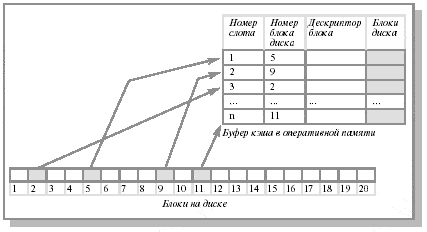


Рис. 12.12. Структура блочного кэша

Замещение блоков должно осуществляться с учетом их важности для файловой системы. Блоки должны быть разделены на категории, например: блоки индексных узлов, блоки косвенной адресации, блоки директорий, заполненные блоки данных и т. д., и в зависимости от принадлежности блока к той или иной категории можно применять к ним разную стратегию замещения.

Во-вторых, поскольку кэширование использует механизм отложенной записи, при котором модификация буфера не вызывает немедленной записи на диск, серьезной проблемой является "старение" информации в дисковых блоках, образы которых находятся в буферном кэше. Несвоевременная синхронизация буфера кэша и диска может привести к очень нежелательным последствиям в случае отказов оборудования или программного обеспечения. Поэтому стратегия и порядок отображения информации из кэша на диск должна быть тщательно продумана.

Так, блоки, существенные для совместимости файловой системы (блоки индексных узлов, блоки косвенной адресации, блоки директорий), должны быть переписаны на диск немедленно, независимо от того, в какой части LRU-цепочки они находятся. Hеобходимо тщательно выбрать порядок такого переписывания.

В Unix имеется для этого вызов SYNC, который заставляет все модифицированные блоки записываться на диск немедленно. Для синхронизации содержимого кэша и диска периодически запускается фоновый процесс-демон. Кроме того, можно организовать синхронный режим работы с отдельными файлами, задаваемый при открытии файла, когда все изменения в файле немедленно сохраняются на диске.

Наконец, проблема конкуренции процессов на доступ к блокам кэша решается ведением списков блоков, пребывающих в различных состояниях, и отметкой о состоянии блока в его дескрипторе. Например, блок может быть заблокирован, участвовать в операции ввода-вывода, а также иметь список процессов, ожидающих освобождения данного блока.

#### Оптимальное размещение информации на диске

Кэширование - не единственный способ увеличения производительности системы. Другая важная техника - сокращение количества движений считывающей головки диска за счет разумной стратегии размещения информации. Например, массив индексных узлов в Unix стараются разместить на средних дорожках. Также имеет смысл размещать индексные узлы поблизости от блоков данных, на которые они ссылаются и т. д.

Кроме того, рекомендуется периодически осуществлять дефрагментацию диска (сборку мусора), поскольку в популярных методиках выделения дисковых блоков (за исключением, может быть, FAT ) принцип локальности не работает, и последовательная обработка файла требует обращения к различным участкам диска.

**64**

В простейшем случае процессор, память и многочисленные внешние устройства связаны большим количеством электрических соединений – линий, которые в совокупности принято называть локальной магистралью компьютера. Внутри локальной магистрали линии, служащие для передачи сходных сигналов и предназначенные для выполнения сходных функций, принято группировать в шины. При этом понятие шины включает в себя не только набор проводников, но и набор жестко заданных протоколов, определяющий перечень сообщений, который может быть передан с помощью электрических сигналов по этим проводникам. В современных компьютерах выделяют как минимум три шины:

* шину данных, состоящую из линий данных и служащую для передачи информации между процессором и памятью, процессором и устройствами ввода-вывода, памятью и внешними устройствами;
* адресную шину, состоящую из линий адреса и служащую для задания адреса ячейки памяти или указания устройства ввода-вывода, участвующих в обмене информацией;
* шину управления, состоящую из линий управления локальной магистралью и линий ее состояния, определяющих поведение локальной магистрали . В некоторых архитектурных решениях линии состояния выносятся из этой шины в отдельную шину состояния.

Количество линий, входящих в состав шины, принято называть разрядностью ( шириной ) этой шины. Ширина адресной шины, например, определяет максимальный размер оперативной памяти, которая может быть установлена в вычислительной системе. Ширина шины данныхопределяет максимальный объем информации, которая за один раз может быть получена или передана по этой шине.

Операции обмена информацией осуществляются при одновременном участии всех шин. Рассмотрим, к примеру, действия, которые должны быть выполнены для передачи информации из процессора в память. В простейшем случае необходимо выполнить три действия.

1. На адресной шине процессор должен выставить сигналы, соответствующие адресу ячейки памяти, в которую будет осуществляться передача информации.
2. На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть записана в память.
3. После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с памятью, что приведет к занесению необходимой информации по нужному адресу.

Естественно, что приведенные выше действия являются необходимыми, но недостаточными при рассмотрении работы конкретных процессоров и микросхем памяти. Конкретные архитектурные решения могут требовать дополнительных действий: например, выставления на шину управления сигналов частичного использования шины данных (для передачи меньшего количества информации, чем позволяет ширина этой шины); выставления сигнала готовности магистрали после завершения записи в память, разрешающего приступить к новой операции, и т. д. Однако общие принципы выполнения операции записи в память остаются неизменными.

В то время как память легко можно представить себе в виде последовательности пронумерованных адресами ячеек, локализованных внутри одной микросхемы или набора микросхем, к устройствам ввода-вывода подобный подход неприменим. Внешние устройства разнесены пространственно и могут подключаться к локальной магистрали в одной точке или множестве точек, получивших название портов ввода-вывода . Тем не менее, точно так же, как ячейки памяти взаимно однозначно отображались в адресное пространство памяти, порты ввода-вывода можно взаимно однозначно отобразить в другое адресное пространство – адресное пространство ввода-вывода. При этом каждый порт ввода-вывода получает свой номер или адрес в этом пространстве. В некоторых случаях, когда адресное пространство памяти (размер которого определяется шириной адресной шины ) задействовано не полностью (остались адреса, которым не соответствуют физические ячейки памяти) и протоколы работы с внешним устройством совместимы с протоколами работы с памятью, часть портов ввода -вывода может быть отображена непосредственно в адресное пространство памяти (так, например, поступают с видеопамятью дисплеев), правда, тогда эти порты уже не принято называть портами. Надо отметить, что при отображении портов в адресное пространство памяти для организации доступа к ним в полной мере могут быть задействованы существующие механизмы защиты памяти без организации специальных защитных устройств.

В ситуации прямого отображения портов ввода-вывода в адресное пространство памяти действия, необходимые для записи информации и управляющих команд в эти порты или для чтения данных из них и их состояний, ничем не отличаются от действий, производимых для передачи информации между оперативной памятью и процессором, и для их выполнения применяются те же самые команды. Если же порт отображен в адресное пространство ввода-вывода, то процесс обмена информацией инициируется специальными командами ввода-вывода и включает в себя несколько другие действия. Например, для передачи данных в порт необходимо выполнить следующее.

* На адресной шине процессор должен выставить сигналы, соответствующие адресу порта, в который будет осуществляться передача информации, в адресном пространстве ввода-вывода.
* На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть передана в порт.
* После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с устройствами ввода-вывода (переключение адресных пространств!), что приведет к передаче необходимой информации в нужный порт.

Существенное отличие памяти от устройств ввода-вывода заключается в том, что занесение информации в память является окончанием операции записи, в то время как занесение информации в порт зачастую представляет собой инициализацию реального совершения операции ввода-вывода. Что именно должны делать устройства, приняв информацию через свой порт, и каким именно образом они должны поставлять информацию для чтения из порта, определяется электронными схемами устройств, получившими название контроллеров . Контроллер может непосредственно управлять отдельным устройством (например, контроллер диска), а может управлять несколькими устройствами, связываясь с их контроллерами посредством специальных шин ввода-вывода (шина IDE, шина SCSI и т. д.).

Современные вычислительные системы могут иметь разнообразную архитектуру, множество шин и магистралей, мосты для перехода информации от одной шины к другой и т. п. Для нас сейчас важными являются только следующие моменты.

* Устройства ввода-вывода подключаются к системе через порты.
* Могут существовать два адресных пространства: пространство памяти и пространство ввода-вывода.
* Порты, как правило, отображаются в адресное пространство ввода-вывода и иногда – непосредственно в адресное пространство памяти.
* Использование того или иного адресного пространства определяется типом команды, выполняемой процессором, или типом ее операндов.
* Физическим управлением устройством ввода-вывода, передачей информации через порт и выставлением некоторых сигналов на магистрали занимается контроллер устройства.

Именно единообразие подключения внешних устройств к вычислительной системе является одной из составляющих идеологии, позволяющих добавлять новые устройства без перепроектирования всей системы.

**65**

Контроллеры устройств ввода-вывода весьма различны как по своему внутреннему строению, так и по исполнению (от одной микросхемы до специализированной вычислительной системы со своим процессором, памятью и т. д.), поскольку им приходится управлять совершенно разными приборами. Не вдаваясь в детали этих различий, мы выделим некоторые общие черты контроллеров, необходимые им для взаимодействия с вычислительной системой. Обычно каждый контроллер имеет по крайней мере четыре внутренних регистра, называемых регистрами состояния , управления, входных данных и выходных данных. Для доступа к содержимому этих регистров вычислительная система может использовать один или несколько портов, что для нас не существенно. Для простоты изложения будем считать, что каждому регистру соответствует свой порт.

Регистр состояния содержит биты, значение которых определяется состоянием устройства ввода-вывода и которые доступны только для чтения вычислительной системой. Эти биты индицируют завершение выполнения текущей команды на устройстве ( бит занятости ), наличие очередного данного в регистре выходных данных ( бит готовности данных ), возникновение ошибки при выполнении команды ( бит ошибки ) и т. д.

Регистр управления получает данные, которые записываются вычислительной системой для инициализации устройства ввода-вывода или выполнения очередной команды, а также изменения режима работы устройства. Часть битов в этом регистре может быть отведена под код выполняемой команды, часть битов будет кодировать режим работы устройства, бит готовности команды свидетельствует о том, что можно приступить к ее выполнению.

Регистр выходных данных служит для помещения в него данных для чтения вычислительной системой, а регистр входных данныхпредназначен для помещения в него информации, которая должна быть выведена на устройство. Обычно емкость этих регистров не превышает ширину линии данных (а чаще всего меньше ее), хотя некоторые контроллеры могут использовать в качестве регистров очередь FIFO для буферизации поступающей информации.

Разумеется, набор регистров и составляющих их битов приблизителен, он призван послужить нам моделью для описания процесса передачи информации от вычислительной системы к внешнему устройству и обратно, но в том или ином виде он обычно присутствует во всех контроллерах устройств.

**66**

Использование механизма прерываний позволяет разумно загружать процессор в то время, когда устройство ввода-вывода занимается своей работой. Однако запись или чтение большого количества информации из адресного пространства ввода-вывода (например, с диска) приводят к большому количеству операций ввода-вывода, которые должен выполнять процессор. Для освобождения процессора от операций последовательного вывода данных из оперативной памяти или последовательного ввода в нее был предложен механизм прямого доступавнешних устройств к памяти – ПДП или Direct Memory Access – DMA. Давайте кратко рассмотрим, как работает этот механизм.

Для того чтобы какое-либо устройство, кроме процессора, могло записать информацию в память или прочитать ее из памяти, необходимо чтобы это устройство могло забрать у процессора управление локальной магистралью для выставления соответствующих сигналов на шины адреса, данных и управления. Для централизации эти обязанности обычно возлагаются не на каждое устройство в отдельности, а на специальный контроллер – контроллер прямого доступа к памяти. Контроллер прямого доступа к памяти имеет несколько спаренных линий – каналов DMA, которые могут подключаться к различным устройствам. Перед началом использования прямого доступа к памяти этот контроллер необходимо запрограммировать, записав в его порты информацию о том, какой канал или каналы предполагается задействовать, какие операции они будут совершать, какой адрес памяти является начальным для передачи информации и какое количество информации должно быть передано. Получив по одной из линий – каналов DMA, сигнал запроса на передачу данных от внешнего устройства, контроллер по шине управления сообщает процессору о желании взять на себя управление локальной магистралью. Процессор, возможно, через некоторое время, необходимое для завершения его действий с магистралью, передает управление ею контроллеру DMA, известив его специальным сигналом. Контроллер DMA выставляет на адресную шину адрес памяти для передачи очередной порции информации и по второй линии канала прямого доступа к памяти сообщает устройству о готовности магистрали к передаче данных. После этого, используя шину данных и шину управления, контроллер DMA, устройство ввода-вывода и память осуществляют процесс обмена информацией. Затем контроллер прямого доступа к памятиизвещает процессор о своем отказе от управления магистралью, и тот берет руководящие функции на себя. При передаче большого количества данных весь процесс повторяется циклически.

При прямом доступе к памяти процессор и контроллер DMA по очереди управляют локальной магистралью. Это, конечно, несколько снижает производительность процессора, так как при выполнении некоторых команд или при чтении очередной порции команд во внутренний кэш он должен поджидать освобождения магистрали, но в целом производительность вычислительной системы существенно возрастает.

При подключении к системе нового устройства, которое умеет использовать прямой доступ к памяти, обычно необходимо программно или аппаратно задать номер канала DMA, к которому будет приписано устройство. В отличие от прерываний, где один номер прерывания мог соответствовать нескольким устройствам, каналы DMA всегда находятся в монопольном владении устройств.

**67**

Рассмотренные в предыдущем разделе физические механизмы взаимодействия устройств ввода-вывода с вычислительной системой позволяют понять, почему разнообразные внешние устройства легко могут быть добавлены в существующие компьютеры. Все, что необходимо сделать пользователю при подключении нового устройства, – это отобразить порты устройства в соответствующее адресное пространство, определить, какой номер будет соответствовать прерыванию, генерируемому устройством, и, если нужно, закрепить за устройством некоторый канал DMA. Для нормального функционирования hardware этого будет достаточно. Однако мы до сих пор ничего не сказали о том, как должна быть построена подсистема управления вводом-выводом в операционной системе для легкого и безболезненного добавления новых устройств и какие функции вообще обычно на нее возлагаются.

##### Структура системы ввода-вывода

Если поручить неподготовленному пользователю сконструировать систему ввода-вывода, способную работать со всем множеством внешних устройств, то, скорее всего, он окажется в ситуации, в которой находились биологи и зоологи до появления трудов Линнея [[Linnaes, 1789](https://www.intuit.ru/studies/courses/2192/31/literature#literature.6)]. Все устройства разные, отличаются по выполняемым функциям и своим характеристикам, и кажется, что принципиально невозможно создать систему, которая без больших постоянных переделок позволяла бы охватывать все многообразие видов. Вот перечень лишь нескольких направлений (далеко не полный), по которым различаются устройства.

* Скорость обмена информацией может варьироваться в диапазоне от нескольких байтов в секунду (клавиатура) до нескольких гигабайтов в секунду (сетевые карты).
* Одни устройства могут использоваться несколькими процессами параллельно (являются разделяемыми), в то время как другие требуют монопольного захвата процессом.
* Устройства могут запоминать выведенную информацию для ее последующего ввода или не обладать этой функцией. Устройства, запоминающие информацию, в свою очередь, могут дифференцироваться по формам доступа к сохраненной информации: обеспечивать к ней последовательный доступ в жестко заданном порядке или уметь находить и передавать только необходимую порцию данных.
* Часть устройств умеет передавать данные только по одному байту последовательно ( символьные устройства ), а часть устройств умеет передавать блок байтов как единое целое ( блочные устройства ).
* Существуют устройства, предназначенные только для ввода информации, устройства, предназначенные только для вывода информации, и устройства, которые могут выполнять и ввод, и вывод.

В области технического обеспечения удалось выделить несколько основных принципов взаимодействия внешних устройств с вычислительной системой, т. е. создать единый интерфейс для их подключения, возложив все специфические действия на контроллеры самих устройств. Тем самым конструкторы вычислительных систем переложили все хлопоты, связанные с подключением внешней аппаратуры, на разработчиков самой аппаратуры, заставляя их придерживаться определенного стандарта.

Похожий подход оказался продуктивным и в области программного подключения устройств ввода-вывода. Подобно тому как Линнею удалось заложить основы систематизации знаний о растительном и животном мире, разделив все живое в природе на относительно небольшое число классов и отрядов, мы можем разделить устройства на относительно небольшое число типов, отличающихся по набору операций, которые могут быть ими выполнены, считая все остальные различия несущественными. Мы можем затем специфицировать интерфейсы между ядром операционной системы, осуществляющим некоторую общую политику ввода-вывода, и программными частями, непосредственно управляющими устройствами, для каждого из таких типов. Более того, разработчики операционных систем получают возможность освободиться от написания и тестирования этих специфических программных частей, получивших название драйверов, передав эту деятельность производителям самих внешних устройств. Фактически мы приходим к использованию принципа уровневого или слоеного построения системы управления вводом-выводом для операционной системы (см. [рис. 13.1](https://www.intuit.ru/studies/courses/2192/31/lecture/992?page=3#image.13.1)).

Два нижних уровня этой слоеной системы составляет hardware: сами устройства, непосредственно выполняющие операции, и их контроллеры, служащие для организации совместной работы устройств и остальной вычислительной системы. Следующий уровень составляют драйверы устройств ввода-вывода, скрывающие от разработчиков операционных систем особенности функционирования конкретных приборов и обеспечивающие четко определенный интерфейс между hardware и вышележащим уровнем – уровнем базовой подсистемы ввода-вывода, которая, в свою очередь, предоставляет механизм взаимодействия между драйверами и программной частью вычислительной системы в целом.

**68**

Как и система видов Линнея, система типов устройств является далеко не полной и не строго выдержанной. Устройства обычно принято разделять по преобладающему типу интерфейса на следующие виды:

* символьные (клавиатура, модем, терминал и т. п.);
* блочные (магнитные и оптические диски и ленты, и т. д.);
* сетевые (сетевые карты);
* все остальные (таймеры, графические дисплеи, телевизионные устройства, видеокамеры и т. п.);

Такое деление является весьма условным. В одних операционных системах сетевые устройства могут не выделяться в отдельную группу, в некоторых других – отдельные группы составляют звуковые устройства и видеоустройства и т. д. Некоторые группы в свою очередь могут разбиваться на подгруппы: подгруппа жестких дисков, подгруппа мышек, подгруппа принтеров. Нас такие детали не интересуют. Мы не ставим перед собой цель осуществить систематизацию всех возможных устройств, которые могут быть подключены к вычислительной системе. Единственное, для чего нам понадобится эта классификация, так это для иллюстрации того положения, что устройства могут быть разделены на группы по выполняемым ими функциям, и для понимания функций драйверов, и интерфейса между ними и базовой подсистемой ввода-вывода.

Для этого мы рассмотрим только две группы устройств: символьные и блочные. Как уже упоминалось в предыдущем разделе, символьные устройства – это устройства, которые умеют передавать данные только последовательно, байт за байтом, а блочные устройства – это устройства, которые могут передавать блок байтов как единое целое.

К символьным устройствам обычно относятся устройства ввода информации, которые спонтанно генерируют входные данные: клавиатура, мышь, модем, джойстик. К ним же относятся и устройства вывода информации, для которых характерно представление данных в виде линейного потока: принтеры, звуковые карты и т. д. По своей природе символьные устройства обычно умеют совершать две общие операции: ввести символ (байт) и вывести символ (байт) – get и put.

Для блочных устройств, таких как магнитные и оптические диски, ленты и т. п. естественными являются операции чтения и записи блока информации – read и write, а также, для устройств прямого доступа, операция поиска требуемого блока информации – seek.

Драйверы символьных и блочных устройств должны предоставлять базовой подсистеме ввода-вывода функции для осуществления описанных общих операций. Помимо общих операций, некоторые устройства могут выполнять операции специфические, свойственные только им – например, звуковые карты умеют увеличивать или уменьшать среднюю громкость звучания, дисплеи умеют изменять свою разрешающую способность. Для выполнения таких специфических действий в интерфейс между драйвером и базовой подсистемой ввода-вывода обычно входит еще одна функция, позволяющая непосредственно передавать драйверу устройства произвольную команду с произвольными параметрами, что позволяет задействовать любую возможность драйвера без изменения интерфейса. В операционной системе Unix такая функция получила название ioctl (от input-output control).

Помимо функций read, write, seek (для блочных устройств ), get, put (для символьных устройств ) и ioctl, в состав интерфейса обычно включают еще следующие функции.

* Функцию инициализации или повторной инициализации работы драйвера и устройства – open.
* Функцию временного завершения работы с устройством (может, например, вызывать отключение устройства) – close.
* Функцию опроса состояния устройства (если по каким-либо причинам работа с устройством производится методом опроса его состояния, например, в операционных системах Windows NT и Windows 9x так построена работа с принтерами через параллельный порт) – poll.
* Функцию остановки драйвера, которая вызывается при остановке операционной системы или выгрузке драйвера из памяти, halt.

Существует еще ряд действий, выполнение которых может быть возложено на драйвер, но поскольку, как правило, это действия базовой подсистемы ввода-вывода, мы поговорим о них в следующем разделе. Приведенные выше названия функций, конечно, являются условными и могут меняться от одной операционной системы к другой, но действия, выполняемые драйверами, характерны для большинства операционных систем, и соответствующие функции присутствуют в интерфейсах к ним.

**69**

Базовая подсистема ввода-вывода служит посредником между процессами вычислительной системы и набором драйверов. Системные вызовы для выполнения операций ввода-вывода трансформируются ею в вызовы функций необходимого драйвера устройства. Однако обязанности базовой подсистемы не сводятся к выполнению только действий трансляции общего системного вызова в обращение к частной функции драйвера. Базовая подсистема предоставляет вычислительной системе такие услуги, как поддержка блокирующихся, неблокирующихся и асинхронных системных вызовов, буферизация и кэширование входных и выходных данных, осуществление spooling'a и монопольного захвата внешних устройств, обработка ошибок и прерываний, возникающих при операциях ввода-вывода, планирование последовательности запросов на выполнение этих операций. Давайте остановимся на этих услугах подробнее.

##### Блокирующиеся, неблокирующиеся и асинхронные системные вызовы

Все системные вызовы, связанные с осуществлением операций ввода-вывода, можно разбить на три группы по способам реализации взаимодействия процесса и устройства ввода-вывода.

* К первой, наиболее привычной для большинства программистов группе относятся блокирующиеся системные вызовы. Как следует из самого названия, применение такого вызова приводит к блокировке инициировавшего его процесса, т. е. процесс переводится операционной системой из состояния исполнение в состояние ожидание. Завершив выполнение всех операций ввода-вывода, предписанных системным вызовом, операционная система переводит процесс из состояния ожидание в состояние готовность. После того как процесс будет снова выбран для исполнения, в нем произойдет окончательный возврат из системного вызова. Типичным для применения такого системного вызова является случай, когда процессу необходимо получить от устройства строго определенное количество данных, без которых он не может выполнять работу далее.
* Ко второй группе относятся неблокирующиеся системные вызовы. Их название не совсем точно отражает суть дела. В простейшем случае процесс, применивший неблокирующийся вызов, не переводится в состояние ожидание вообще. Системный вызов возвращается немедленно, выполнив предписанные ему операции ввода-вывода полностью, частично или не выполнив совсем, в зависимости от текущей ситуации (состояния устройства, наличия данных и т. д.). В более сложных ситуациях процесс может блокироваться, но условием его разблокирования является завершение всех необходимых операций или окончание некоторого промежутка времени. Типичным случаем применения неблокирующегося системного вызова может являться периодическая проверка на поступление информации с клавиатуры при выполнении трудоемких расчетов.
* К третьей группе относятся асинхронные системные вызовы. Процесс, использовавший асинхронный системный вызов, никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода-вывода и немедленно возвращается, после чего процесс продолжает свою регулярную деятельность. Об окончании завершения операции ввода-вывода операционная система впоследствии информирует процесс изменением значений некоторых переменных, передачей ему сигнала или сообщения или каким-либо иным способом. Необходимо четко понимать разницу между неблокирующимися и асинхронными вызовами. Неблокирующийся системный вызов для выполнения операции read вернется немедленно, но может прочитать запрошенное количество байтов, меньшее количество или вообще ничего. Асинхронный системный вызов для этой операции также вернется немедленно, но требуемое количество байтов рано или поздно будет прочитано в полном объеме.

**70**

##### Строение жесткого диска и параметры планирования

Современный жесткий магнитный диск представляет собой набор круглых пластин, находящихся на одной оси и покрытых с одной или двух сторон специальным магнитным слоем (см. [рис. 13.2](https://www.intuit.ru/studies/courses/2192/31/lecture/992?page=6#image.13.2)). Около каждой рабочей поверхности каждой пластины расположены магнитные головки для чтения и записи информации. Эти головки присоединены к специальному рычагу, который может перемещать весь блок головок над поверхностями пластин как единое целое. Поверхности пластин разделены на концентрические кольца, внутри которых, собственно, и может храниться информация. Набор концентрических колец на всех пластинах для одного положения головок (т. е. все кольца, равноудаленные от оси) образует цилиндр. Каждое кольцо внутри цилиндра получило название дорожки (по одной или две дорожки на каждую пластину). Все дорожки делятся на равное число секторов. Количество дорожек, цилиндров и секторов может варьироваться от одного жесткого диска к другому в достаточно широких пределах. Как правило, сектор является минимальным объемом информации, которая может быть прочитана с диска за один раз.

При работе диска набор пластин вращается вокруг своей оси с высокой скоростью, подставляя по очереди под головки соответствующих дорожек все их сектора. Номер сектора, номер дорожки и номер цилиндра однозначно определяют положение данных на жестком диске и, наряду с типом совершаемой операции – чтение или запись, полностью характеризуют часть запроса, связанную с устройством, при обмене информацией в объеме одного сектора.

При планировании использования жесткого диска естественным параметром планирования является время, которое потребуется для выполнения очередного запроса. Время, необходимое для чтения или записи определенного сектора на определенной дорожке определенного цилиндра, можно разделить на две составляющие: время обмена информацией между магнитной головкой и компьютером, которое обычно не зависит от положения данных и определяется скоростью их передачи (transfer speed), и время, необходимое для позиционирования головки над заданным сектором, – время позиционирования (positioning time). Время позиционирования, в свою очередь, состоит из времени, необходимого для перемещения головок на нужный цилиндр, – времени поиска (seek time) и времени, которое требуется для того, чтобы нужный сектор довернулся под головку, – задержки на вращение (rotational latency). Времена поиска пропорциональны разнице между номерами цилиндров предыдущего и планируемого запросов, и их легко сравнивать. Задержка на вращение определяется довольно сложными соотношениями между номерами цилиндров и секторов предыдущего и планируемого запросов и скоростями вращения диска и перемещения головок. Без знания соотношения этих скоростей сравнение становится невозможным. Поэтому естественно, что набор параметров планирования сокращается до времени поиска различных запросов, определяемого текущим положением головки и номерами требуемых цилиндров, а разницей в задержках на вращение пренебрегают.

##### Алгоритм First Come First Served (FCFS)

Простейшим алгоритмом, к которому мы уже должны были привыкнуть, является алгоритм First Come First Served (FCFS) – первым пришел, первым обслужен. Все запросы организуются в очередь FIFO и обслуживаются в порядке поступления. Алгоритм прост в реализации, но может приводить к достаточно длительному общему времени обслуживания запросов. Рассмотрим пример. Пусть у нас на диске из 100 цилиндров (от 0 до 99) есть следующая очередь запросов: 23, 67, 55, 14, 31, 7, 84, 10 и головки в начальный момент находятся на 63-м цилиндре. Тогда положение головок будет меняться следующим образом:

63->23->67->55->14->31->7->84->10

и всего головки переместятся на 329 цилиндров. Неэффективность алгоритма хорошо иллюстрируется двумя последними перемещениями с 7 цилиндра через весь диск на 84 цилиндр и затем опять через весь диск на цилиндр 10. Простая замена порядка двух последних перемещений (7 \to 10 \to 84) позволила бы существенно сократить общее время обслуживания запросов. Поэтому давайте перейдем к рассмотрению другого алгоритма.

##### Алгоритм Short Seek Time First (SSTF)

Как мы убедились, достаточно разумным является первоочередное обслуживание запросов, данные для которых лежат рядом с текущей позицией головок, а уж затем далеко отстоящих. Алгоритм Short Seek Time First (SSTF) – короткое время поиска первым – как раз и исходит из этой позиции. Для очередного обслуживания будем выбирать запрос, данные для которого лежат наиболее близко к текущему положению магнитных головок. Естественно, что при наличии равноудаленных запросов решение о выборе между ними может приниматься исходя из различных соображений, например по алгоритму FCFS. Для предыдущего примера алгоритм даст такую последовательность положений головок:

63->67->55->31->23->14->10->7->84

и всего головки переместятся на 141 цилиндр. Заметим, что наш алгоритм похож на алгоритм SJF планирования процессов, если за аналог оценки времени очередного CPU burst процесса выбирать расстояние между текущим положением головки и положением, необходимым для удовлетворения запроса. И точно так же, как алгоритм SJF, он может приводить к длительному откладыванию выполнения какого-либо запроса. Необходимо вспомнить, что запросы в очереди могут появляться в любой момент времени. Если у нас все запросы, кроме одного, постоянно группируются в области с большими номерами цилиндров, то этот один запрос может находиться в очереди неопределенно долго.

Точный алгоритм SJF являлся оптимальным для заданного набора процессов с заданными временами CPU burst. Очевидно, что алгоритм SSTFне является оптимальным. Если мы перенесем обслуживание запроса 67-го цилиндра в промежуток между запросами 7-го и 84-го цилиндров, мы уменьшим общее время обслуживания. Это наблюдение приводит нас к идее целого семейства других алгоритмов – алгоритмов сканирования.

##### Алгоритмы сканирования (SCAN, C-SCAN, LOOK, C-LOOK)

В простейшем из алгоритмов сканирования – SCAN – головки постоянно перемещаются от одного края диска до другого, по ходу дела обслуживая все встречающиеся запросы. По достижении другого края направление движения меняется, и все повторяется снова. Пусть в предыдущем примере в начальный момент времени головки двигаются в направлении уменьшения номеров цилиндров. Тогда мы и получим порядок обслуживания запросов, подсмотренный в конце предыдущего раздела. Последовательность перемещения головок выглядит следующим образом:

63->55->31->23->14->10->7->0->67->84

и всего головки переместятся на 147 цилиндров.

Если мы знаем, что обслужили последний попутный запрос в направлении движения головок, то мы можем не доходить до края диска, а сразу изменить направление движения на обратное:

63->55->31->23->14->10->7->67->84

и всего головки переместятся на 133 цилиндра. Полученная модификация алгоритма SCAN получила название LOOK.

Допустим, что к моменту изменения направления движения головки в алгоритме SCAN, т. е. когда головка достигла одного из краев диска, у этого края накопилось большое количество новых запросов, на обслуживание которых будет потрачено достаточно много времени (не забываем, что надо не только перемещать головку, но еще и передавать прочитанные данные!). Тогда запросы, относящиеся к другому краю диска и поступившие раньше, будут ждать обслуживания несправедливо долго. Для сокращения времени ожидания запросов применяется другая модификация алгоритма SCAN – циклическое сканирование. Когда головка достигает одного из краев диска, она без чтения попутных запросов (иногда существенно быстрее, чем при выполнении обычного поиска цилиндра) перемещается на другой край, откуда вновь начинает движение в прежнем направлении. Для этого алгоритма, получившего название C-SCAN, последовательность перемещений будет выглядеть так:

63->55->31->23->14->10->7->0->99->84->67

По аналогии с алгоритмом LOOK для алгоритма SCAN можно предложить и алгоритм C-LOOK для алгоритма C-SCAN:

63->55->31->23->14->10->7->84->67

Существуют и другие разновидности алгоритмов сканирования, и совсем другие алгоритмы, но мы на этом закончим, ибо было сказано: "И еще раз говорю: никто не обнимет необъятного".

**71**

В первой лекции мы говорили, что существует два основных подхода к организации операционных систем для вычислительных комплексов, связанных в сеть, – это сетевые и распределенные операционные системы. Необходимо отметить, что терминология в этой области еще не устоялась. В одних работах все операционные системы, обеспечивающие функционирование компьютеров в сети, называются распределенными, а в других, наоборот, сетевыми. Мы придерживаемся той точки зрения, что сетевые и распределенные системы являются принципиально различными.

В сетевых операционных системах для того, чтобы задействовать ресурсы другого сетевого компьютера, пользователи должны знать о его наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных сетевых средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения существенно не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся, на локальной или удаленной машине, и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

Изучение строения распределенных операционных систем не входит в задачи нашего курса. Этому вопросу посвящены другие учебные курсы – Advanced operating systems, как называют их в англоязычных странах, или "Современные операционные системы", как принято называть их в России.

В этой лекции мы затронем вопросы, связанные с сетевыми операционными системами, а именно – какие изменения необходимо внести в классическую операционную систему для объединения компьютеров в сеть.

**72**

Все перечисленные выше цели объединения компьютеров в вычислительные сети не могут быть достигнуты без организации взаимодействия процессов на различных вычислительных системах. Будь то доступ к разделяемым ресурсам или общение пользователей через сеть – в основе всего этого лежит взаимодействие удаленных процессов, т. е. процессов, которые находятся под управлением физически разных операционных систем. Поэтому мы в своей работе сосредоточимся именно на вопросах кооперации таких процессов, в первую очередь выделив ее отличия от кооперации процессов в одной автономной вычислительной системе (кооперации локальных процессов ), о которой мы говорили в лекциях 4, 5 и 6.

1. Изучая взаимодействие локальных процессов, мы разделили средства обмена информацией по объему передаваемых между ними данных и возможности влияния на поведение другого процесса на три категории: сигнальные, канальные и разделяемая память. На самом деле во всей этой систематизации присутствовала некоторая доля лукавства. Мы фактически классифицировали средства связи по виду интерфейса обращения к ним, в то время как реальной физической основой для всех средств связи в том или ином виде являлось разделение памяти. Семафоры представляют собой просто целочисленные переменные, лежащие в разделяемой памяти, к которым посредством системных вызовов, определяющих состав и содержание допустимых операций над ними, могут обращаться различные процессы. Очереди сообщений и pip'ы базируются на буферах ядра операционной системы, которые опять-таки с помощью системных вызовов доступны различным процессам. Иного способа реально передать информацию от процесса к процессу в автономной вычислительной системе просто не существует. Взаимодействие удаленных процессов принципиально отличается от ранее рассмотренных случаев. Общей памяти у различных компьютеров физически нет. Удаленные процессы могут обмениваться информацией, только передавая друг другу пакеты данных определенного формата (в виде последовательностей электрических или электромагнитных сигналов, включая световые) через некоторый физический канал связи или несколько таких каналов, соединяющих компьютеры. Поэтому в основе всех средств взаимодействия удаленных процессов лежит передача структурированных пакетов информации или сообщений.
2. При взаимодействии локальных процессов и процесс–отправитель информации, и процесс-получатель функционируют под управлением одной и той же операционной системы. Эта же операционная система поддерживает и функционирование промежуточных накопителей данных при использовании непрямой адресации. Для организации взаимодействия процессы пользуются одними и теми же системными вызовами, присущими данной операционной системе, с одинаковыми интерфейсами. Более того, в автономной операционной системе передача информации от одного процесса к другому, независимо от используемого способа адресации, как правило (за исключением микроядерных операционных систем), происходит напрямую – без участия других процессов-посредников. Но даже и при наличии процессов-посредников все участники передачи информации находятся под управлением одной и той же операционной системы. При организации сети, конечно, можно обеспечить прямую связь между всеми вычислительными комплексами, соединив каждый из них со всеми остальными посредством прямых физических линий связи или подключив все комплексы к общей шине (по примеру шин данных и адреса в компьютере). Однако такая сетевая топология не всегда возможна по ряду физических и финансовых причин. Поэтому во многих случаях информация между удаленными процессами в сети передается не напрямую, а через ряд процессов-посредников, "обитающих" на вычислительных комплексах, не являющихся компьютерами отправителя и получателя и работающих под управлением собственных операционных систем. Однако и при отсутствии процессов-посредников удаленные процесс-отправитель и процесс-получатель функционируют под управлением различных операционных систем, часто имеющих принципиально разное строение.
3. Вопросы надежности средств связи и способы ее реализации, рассмотренные нами в лекции 4, носили для случая локальных процессовскорее теоретический характер. Мы выяснили, что физической основой "общения" процессов на автономной вычислительной машине является разделяемая память. Поэтому для локальных процессов надежность передачи информации определяется надежностью ее передачи по шине данных и хранения в памяти машины, а также корректностью работы операционной системы. Для хороших вычислительных комплексов и операционных систем мы могли забыть про возможную ненадежность средств связи. Для удаленных процессов вопросы, связанные с надежностью передачи данных, становятся куда более значимыми. Протяженные сетевые линии связи подвержены разнообразным физическим воздействиям, приводящим к искажению передаваемых по ним физических сигналов (помехи в эфире) или к полному отказу линий (мыши съели кабель). Даже при отсутствии внешних помех передаваемый сигнал затухает по мере удаления от точки отправления, приближаясь по интенсивности к внутренним шумам линий связи. Промежуточные вычислительные комплексы сети, участвующие в доставке информации, не застрахованы от повреждений или внезапной перезагрузки операционной системы. Поэтому вычислительные сети должны организовываться исходя из предпосылок ненадежности доставки физических пакетов информации.
4. При организации взаимодействия локальных процессов каждый процесс (в случае прямой адресации) и каждый промежуточный объект для накопления данных (в случае непрямой адресации) должны были иметь уникальные идентификаторы – адреса – в рамках одной операционной системы. При организации взаимодействия удаленных процессов участники этого взаимодействия должны иметь уникальные адреса уже в рамках всей сети.
5. Физическая линия связи, соединяющая несколько вычислительных комплексов, является разделяемым ресурсом для всех процессов комплексов, которые хотят ее использовать. Если два процесса попытаются одновременно передать пакеты информации по одной и той же линии, то в результате интерференции физических сигналов, представляющих эти пакеты, произойдет взаимное искажение передаваемых данных. Для того чтобы избежать возникновения такой ситуации (race condition!) и обеспечить эффективную совместную работу вычислительных систем, должны выполняться условия взаимоисключения, прогресса и ограниченного ожидания при использовании общей линии связи, но уже не на уровне отдельных процессов операционных систем, а на уровне различных вычислительных комплексов в целом.

**73**

Для описания происходящего в автономной операционной системе в лекции 2 было введено основополагающее понятие "процесс", на котором, по сути дела, базируется весь наш курс. Для того чтобы описать взаимодействие удаленных процессов и понять, какие функции и как должны выполнять дополнительные части сетевых операционных систем, отвечающих за такое взаимодействие, нам понадобится не менее фундаментальное понятие – протокол.

"Общение" локальных процессов напоминает общение людей, проживающих в одном городе. Если взаимодействующие процессы находятся под управлением различных операционных систем, то эта ситуация подобна общению людей, проживающих в разных городах и, возможно, в разных странах.

Каким образом два человека, находящиеся в разных городах, а тем более странах, могут обмениваться информацией? Для этого им обычно приходится прибегать к услугам соответствующих служб связи. При этом между службами связи различных городов (государств) должны быть заключены определенные соглашения, позволяющие корректно организовывать такой обмен. Если речь идет, например, о почтовых отправлениях, то в первую очередь необходимо договориться о том, что может представлять собой почтовое отправление, какой вид оно может иметь. Некоторые племена индейцев для передачи информации пользовались узелковыми письмами – поясами, к которым крепились веревочки с различным числом и формой узелков. Если бы такое письмо попало в современный почтовый ящик, то, пожалуй, ни одно отделение связи не догадалось бы, что это – письмо, и пояс был бы выброшен как ненужный хлам. Помимо формы представления информации необходима договоренность о том, какой служебной информацией должно снабжаться почтовое отправление (адрес получателя, срочность доставки, способ пересылки: поездом, авиацией, с помощью курьера и т. п.) и в каком формате она должна быть представлена. Адреса, например, в России и в США принято записывать по-разному. В России мы начинаем адрес со страны, далее указывается город, улица и квартира. В США все наоборот: сначала указывается квартира, затем улица и т. д. Конечно, даже при неправильной записи адреса письмо, скорее всего, дойдет до получателя, но можно себе представить растерянность почтальона, пытающегося разгадать, что это за страна или город – "кв.162"? Как видим, доставка почтового отправления из одного города (страны) в другой требует целого ряда соглашений между почтовыми ведомствами этих городов (стран).

Аналогичная ситуация возникает и при общении удаленных процессов, работающих под управлением разных операционных систем. Здесь процессы играют роль людей, проживающих в разных городах, а сетевые части операционных систем – роль соответствующих служб связи. Для того чтобы удаленные процессы могли обмениваться данными, необходимо, чтобы сетевые части операционных систем руководствовались определенными соглашениями, или, как принято говорить, поддерживали определенные протоколы. Термин "протокол" уже встречался нам в лекции 13, посвященной организации ввода-вывода в операционных системах, при обсуждении понятия "шина". Мы говорили, что понятие шины подразумевает не только набор проводников, но и набор жестко заданных протоколов, определяющий перечень сообщений, который может быть передан с помощью электрических сигналов по этим проводникам, т. е. в "протокол" мы вкладывали практически тот же смысл. В следующем разделе мы попытаемся дать более формализованное определение этого термина.

Необходимо отметить, что и локальные процессы при общении также должны руководствоваться определенными соглашениями или поддерживать определенные протоколы. Только в автономных операционных системах они несколько завуалированы. В роли таких протоколоввыступают специальная последовательность системных вызовов при организации взаимодействия процессов и соглашения о параметрах системных вызовов.

Различные способы решения проблем 3–5, поднятых в предыдущем разделе, по существу, представляют собой различные соглашения, которых должны придерживаться сетевые части операционных систем, т. е. различные сетевые протоколы. Именно наличие сетевых протоколовпозволяет организовать взаимодействие удаленных процессов.

При рассмотрении перечисленных выше проблем необходимо учитывать, с какими сетями мы имеем дело.

В литературе принято говорить о локальных вычислительных сетях (LAN – Local Area Network) и глобальных вычислительных сетях (WAN – Wide Area Network). Строгого определения этим понятиям обычно не дается, а принадлежность сети к тому или иному типу часто определяется взаимным расположением вычислительных комплексов, объединенных в сеть. Так, например, в большинстве работ к локальным сетям относят сети, состоящие из компьютеров одной организации, размещенные в пределах одного или нескольких зданий, а к глобальным сетям – сети, охватывающие все компьютеры в нескольких городах и более. Зачастую вводится дополнительный термин для описания сетей промежуточного масштаба – муниципальных или городских вычислительных сетей (MAN – Metropolitan Area Network) – сетей, объединяющих компьютеры различных организаций в пределах одного города или одного городского района. Таким образом, упрощенно можно рассматривать глобальные сети как сети, состоящие из локальных и муниципальных сетей. А муниципальные сети, в свою очередь, могут состоять из нескольких локальных сетей. На самом деле деление сетей на локальные, глобальные и муниципальные обычно связано не столько с местоположением и принадлежностью вычислительных систем, соединенных сетью, сколько с различными подходами, применяемыми для решения поставленных вопросов в рамках той или иной сети, – с различными используемыми протоколами.

**74**

Даже беглого взгляда на перечень проблем, связанных с логической организацией взаимодействия удаленных процессов, достаточно, чтобы понять, что построение сетевых средств связи – задача более сложная, чем реализация локальных средств связи. Поэтому обычно задачу создания таких средств решают по частям, применяя уже неоднократно упоминавшийся нами "слоеный", или многоуровневый, подход.

Как уже отмечалось при обсуждении "слоеного" строения операционных систем на первой лекции, при таком подходе уровень N системы предоставляет сервисы уровню N+1, пользуясь в свою очередь только сервисами уровня N-1. Следовательно, каждый уровень может взаимодействовать непосредственно только со своими соседями, руководствуясь четко закрепленными соглашениями – вертикальными протоколами, которые принято называть интерфейсами.

Самым нижним уровнем в слоеных сетевых вычислительных системах является уровень, на котором реализуется реальная физическая связьмежду двумя узлами сети. Из предыдущего раздела следует, что для обеспечения обмена физическими сигналами между двумя различными вычислительными системами необходимо, чтобы эти системы поддерживали определенный протокол физического взаимодействия – горизонтальный протокол.

На самом верхнем уровне находятся пользовательские процессы, которые инициируют обмен данными. Количество и функции промежуточных уровней варьируются от одной системы к другой. Вернемся к нашей аналогии с пересылкой почтовых отправлений между людьми, проживающими в разных городах, правда, с порядком их пересылки несколько отличным от привычного житейского порядка. Рассмотрим в качестве пользовательских процессов руководителей различных организаций, желающих обменяться письмами. Руководитель (пользовательский процесс) готовит текст письма (данные) для пересылки в другую организацию. Затем он отдает его своему секретарю (совершает системный вызов – обращение к нижележащему уровню), сообщая, кому и куда должно быть направлено письмо. Секретарь снимает с него копию и выясняет адрес организации. Далее идет обращение к нижестоящему уровню, допустим, письмо направляется в канцелярию. Здесь оно регистрируется (ему присваивается порядковый номер), один экземпляр запечатывается в конверт, на котором указывается, кому и куда адресовано письмо, впечатывается адрес отправителя. Копия остается в канцелярии, а конверт отправляется на почту (переход на следующий уровень). На почте наклеиваются марки и делаются другие служебные пометки, определяется способ доставки корреспонденции и т. д. Наконец, поездом, самолетом или курьером ( физический уровень!) письмо следует в другой город, в котором все уровни проходятся в обратном порядке. Пользовательский уровень (руководитель) после подготовки исходных данных и инициации процесса взаимодействия далее судьбой почтового отправления не занимается. Все остальное (включая, быть может, проверку его доставки и посылку копии в случае утери) выполняют нижележащие уровни.

Заметим, что на каждом уровне взаимодействия в городе отправителя исходные данные (текст письма) обрастают дополнительной служебной информацией. Соответствующие уровни почтовой службы в городе получателя должны уметь понимать эту служебную информацию. Таким образом, для одинаковых уровней в различных городах необходимо следование специальным соглашениям – поддержка определенных горизонтальных протоколов.

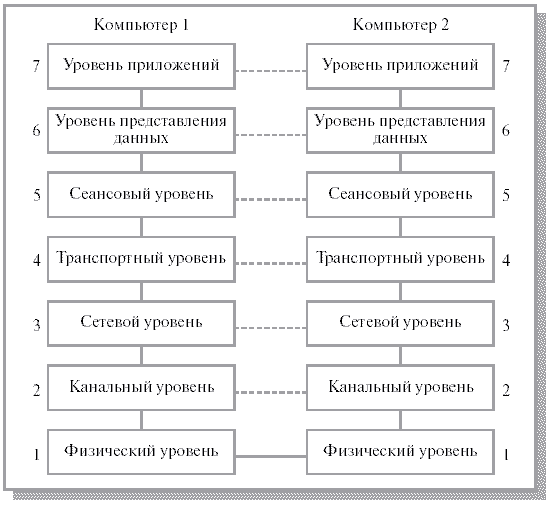
Точно так же в сетевых вычислительных системах все их одинаковые уровни, лежащие выше физического, виртуально обмениваются данными посредством горизонтальных протоколов. Наличие такой виртуальной связи означает, что уровень N компьютера 2 должен получить ту же самую информацию, которая была отправлена уровнем N компьютера 1. Хотя в реальности эта информация должна была сначала дойти сверху вниз до уровня 1 компьютера 1, затем передана уровню 1 компьютера 2 и только после этого доставлена снизу вверх уровню N этого компьютера.

Формальный перечень правил, определяющих последовательность и формат сообщений, которыми обмениваются сетевые компоненты различных вычислительных систем, лежащие на одном уровне, мы и будем называть **сетевым протоколом** .

Всю совокупность вертикальных и горизонтальных протоколов (интерфейсов и сетевых протоколов ) в сетевых системах, построенных по"слоеному" принципу, достаточную для организации взаимодействия удаленных процессов, принято называть **семейством** протоколов или **стеком** протоколов. Сети, построенные на основе разных стеков протоколов, могут быть объединены между собой с использованием вычислительных устройств, осуществляющих трансляцию из одного стека протоколов в другой, причем на различных уровнях слоеной модели

Эталоном многоуровневой схемы построения сетевых средств связи считается семиуровневая модель открытого взаимодействия систем (Open System Interconnection – OSI), предложенная Международной организацией Стандартов (International Standard Organization – ISO) и получившая сокращенное наименование **OSI/ISO** (см. [рис. 14.1](https://www.intuit.ru/studies/courses/2192/31/lecture/994?page=4#image.14.1)).

Давайте очень кратко опишем, какие функции выполняют различные уровни модели OSI/ISO [[Олифер, 2001](https://www.intuit.ru/studies/courses/2192/31/literature#literature.23)]:

* Уровень 1 – **физический**. Этот уровень связан с работой hardware. На нем определяются физические аспекты передачи информации по линиям связи, такие как: напряжения, частоты, природа передающей среды, способ передачи двоичной информации по физическому носителю, вплоть до размеров и формы используемых разъемов. В компьютерах за поддержку физического уровня обычно отвечает сетевой адаптер.
* Уровень 2 – **канальный**. Этот уровень отвечает за передачу данных по физическому уровню без искажений между непосредственно связанными узлами сети. На нем формируются физические пакеты данных для реальной доставки по физическому уровню . Протоколы канального уровня реализуются совместно сетевыми адаптерами и их драйверами (понятие драйвера рассматривалось в лекции 13).
* Уровень 3 – **сетевой**. Сетевой уровень несет ответственность за доставку информации от узла-отправителя к узлу-получателю. На этом уровне частично решаются вопросы адресации, осуществляется выбор маршрутов следования пакетов данных, решаются вопросы стыковки сетей, а также управление скоростью передачи информации для предотвращения перегрузок в сети.
* Уровень 4 – **транспортный**. Регламентирует передачу данных между удаленными процессами. Обеспечивает доставку информации вышележащим уровням с необходимой степенью надежности, компенсируя, быть может, ненадежность нижележащих уровней, связанную с искажением и потерей данных или доставкой пакетов в неправильном порядке. Наряду с сетевым уровнем может управлять скоростью передачи данных и частично решать проблемы адресации.
* 
* **Рис. 14.1.** Семиуровневая эталонная модель OSI/ISO
* Уровень 5 – **сеансовый**. Координирует взаимодействие связывающихся процессов. Основная задача – предоставление средств синхронизации взаимодействующих процессов. Такие средства синхронизации позволяют создавать контрольные точки при передаче больших объемов информации. В случае сбоя в работе сети передачу данных можно возобновить с последней контрольной точки, а не начинать заново.
* Уровень 6 – **уровень представления данных**. Отвечает за форму представления данных, перекодирует текстовую и графическую информацию из одного формата в другой, обеспечивает ее сжатие и распаковку, шифрование и декодирование.
* Уровень 7 – **прикладной**. Служит для организации интерфейса между пользователем и сетью. На этом уровне реализуются такие сервисы, как удаленная передача данных, удаленный терминальный доступ, почтовая служба и работа во Всемирной паутине (Web-браузеры).

**75**

Любой пакет информации, передаваемый по сети, должен быть снабжен адресом получателя. Если взаимодействие подразумевает двустороннее общение, то в пакет следует также включить и адрес отправителя. В лекции 4 мы описали один из протоколов организации надежной связи с использованием контрольных сумм, нумерации пакетов и подтверждения получения неискаженного пакета в правильном порядке. Для отправки подтверждений обратный адрес также следует включать в пересылаемый пакет. Таким образом, практически каждый сетевой пакет информации должен быть снабжен адресом получателя и адресом отправителя. Как могут выглядеть такие адреса?

Несколько раньше, обсуждая отличия взаимодействия удаленных процессов от взаимодействия локальных процессов, мы говорили, что удаленные адресаты должны обладать уникальными адресами уже не в пределах одного компьютера, а в рамках всей сети. Существует два подхода к наделению объектов такими сетевыми адресами: одноуровневый и двухуровневый.

#### Одноуровневые адреса

В небольших компьютерных сетях можно построить одноуровневую систему адресации. При таком подходе каждый процесс, желающий стать участником удаленного взаимодействия (при прямой адресации), и каждый объект, для такого взаимодействия предназначенный (при непрямой адресации), получают по мере необходимости собственные адреса (символьные или числовые), а сами вычислительные комплексы, объединенные в сеть, никаких самостоятельных адресов не имеют. Подобный метод требует довольно сложного протокола обеспечения уникальности адресов. Вычислительный комплекс, на котором запускается взаимодействующий процесс, должен запросить все компьютеры сети о возможности присвоения процессу некоторого адреса. Только после получения от них согласия процессу может быть назначен адрес. Поскольку процесс, посылающий данные другому процессу, не может знать, на каком компоненте сети находится процесс-адресат, передаваемая информация должна быть направлена всем компонентам сети (так называемое **широковещательное сообщение – broadcast message** ), проанализирована ими и либо отброшена (если процесса-адресата на данном компьютере нет), либо доставлена по назначению. Так как все данные постоянно передаются от одного комплекса ко всем остальным, такую одноуровневую схему обычно применяют только в локальных сетях с прямой физической связью всех компьютеров между собой (например, в сети NetBIOS на базе Ethernet), но она является существенно менее эффективной, чем двухуровневая схема адресации.

#### Двухуровневые адреса

При двухуровневой адресации полный сетевой адрес процесса или промежуточного объекта для хранения данных складывается из двух частей – адреса вычислительного комплекса, на котором находится процесс или объект в сети ( удаленного адреса ), и адреса самого процесса или объекта на этом вычислительном комплексе ( локального адреса ). Уникальность полного адреса будет обеспечиваться уникальностью удаленного адреса для каждого компьютера в сети и уникальностью локальных адресов объектов на компьютере. Давайте подробнее рассмотрим проблемы, возникающие для каждого из компонентов полного адреса.

#### Удаленная адресация и разрешение адресов

Инициатором связи процессов друг с другом всегда является человек, будь то программист или обычный пользователь. Как мы неоднократно отмечали в лекциях, человеку свойственно думать словами, он легче воспринимает символьную информацию. Поэтому очевидно, что каждая машина в сети получает символьное, часто даже содержательное имя. Компьютер не разбирается в смысловом содержании символов, ему проще оперировать числами, желательно одного и того же формата, которые помещаются, например, в 4 байта или в 16 байт. Поэтому каждый компьютер в сети для удобства работы вычислительных систем получает числовой адрес. Возникает проблема отображения пространства символьных имен (или адресов) вычислительных комплексов в пространство их числовых адресов. Эта проблема получила наименование **проблемы разрешения адресов**.

С подобными задачами мы уже сталкивались, обсуждая организацию памяти в вычислительных системах (отображение имен переменных в их адреса в процессе компиляции и редактирования связей) и организацию файловых систем (отображение имен файлов в их расположении на диске). Посмотрим, как она может быть решена в сетевом варианте.

Первый способ решения заключается в том, что на каждом сетевом компьютере создается файл, содержащий имена всех машин, доступных по сети, и их числовые эквиваленты. Обращаясь к этому файлу, операционная система легко может перевести символьный удаленный адрес в числовую форму. Такой подход использовался на заре эпохи глобальных сетей и применяется в изолированных локальных сетях в настоящее время. Действительно, легко поддерживать файл соответствий в корректном виде, внося в него необходимые изменения, когда общее число сетевых машин не превышает нескольких десятков. Как правило, изменения вносятся на некотором выделенном административном вычислительном комплексе, откуда затем обновленный файл рассылается по всем компонентам сети.

В современной сетевой паутине этот подход является неприемлемым. Дело даже не в размерах подобного файла, а в частоте требуемых обновлений и в огромном количестве рассылок, что может полностью подорвать производительность сети. Проблема состоит в том, что добавление или удаление компонента сети требует внесения изменений в файлы на всех сетевых машинах. Второй метод разрешения адресов заключается в частичном распределении информации о соответствии символьных и числовых адресов по многим комплексам сети, так что каждый из этих комплексов содержит лишь часть полных данных. Он же определяет и правила построения символических имен компьютеров.

Один из таких способов, используемый в Internet, получил английское наименование Domain Name Service или сокращенно DNS. Эта аббревиатура широко используется и в русскоязычной литературе. Давайте рассмотрим данный метод подробнее.

Организуем логически все компьютеры сети в некоторую древовидную структуру, напоминающую структуру директорий файловых систем, в которых отсутствует возможность организации жестких и мягких связей и нет пустых директорий. Будем рассматривать все компьютеры, входящие во Всемирную сеть, как область самого низкого ранга (аналог корневой директории в файловой системе) – ранга 0. Разобьем все множество компьютеров области на какое-то количество подобластей (domains). При этом некоторые подобласти будут состоять из одного компьютера (аналоги регулярных файлов в файловых системах), а некоторые – более чем из одного компьютера (аналоги директорий в файловых системах). Каждую подобласть будем рассматривать как область более высокого ранга. Присвоим подобластям собственные имена таким образом, чтобы в рамках разбиваемой области все они были уникальны. Повторим такое разбиение рекурсивно для каждой области более высокого ранга, которая состоит более чем из одного компьютера, несколько раз, пока при последнем разбиении в каждой подобласти не окажется ровно по одному компьютеру. Глубина рекурсии для различных областей одного ранга может быть разной, но обычно в целом ограничиваются 3 – 5 разбиениями, начиная от ранга 0.

В результате мы получим дерево, неименованной вершиной которого является область, объединяющая все компьютеры, входящие во Всемирную сеть, именованными терминальными узлами – отдельные компьютеры (точнее – подобласти, состоящие из отдельных компьютеров), а именованными нетерминальными узлами – области различных рангов. Используем полученную структуру для построения имен компьютеров, подобно тому как мы поступали при построении полных имен файлов в структуре директорий файловой системы. Только теперь, двигаясь от корневой вершины к терминальному узлу – отдельному компьютеру, будем вести запись имен подобластей справа налево и отделять имена друг от друга с помощью символа ".".

Допустим, некоторая подобласть, состоящая из одного компьютера, получила имя serv, она входит в подобласть, объединяющую все компьютеры некоторой лаборатории, с именем crec. Та, в свою очередь, входит в подобласть всех компьютеров Московского физико-технического института с именем mipt, которая включается в область ранга 1 всех компьютеров России с именем ru. Тогда имя рассматриваемого компьютера во Всемирной сети будет serv.crec.mipt.ru. Аналогичным образом можно именовать и подобласти, состоящие более чем из одного компьютера.

В каждой полученной именованной области, состоящей более чем из одного узла, выберем один из компьютеров и назначим его ответственным за эту область – сервером DNS. Сервер DNS знает числовые адреса серверов DNS для подобластей, входящих в его зону ответственности, или числовые адреса отдельных компьютеров, если такая подобласть включает в себя только один компьютер. Кроме того, он также знает числовой адрес сервера DNS, в зону ответственности которого входит рассматриваемая область (если это не область ранга 1), или числовые адреса всех серверов DNS ранга 1 (в противном случае). Отдельные компьютеры всегда знают числовые адреса серверов DNS, которые непосредственно за них отвечают.

Рассмотрим теперь, как процесс на компьютере serv.crec.mipt.ru может узнать числовой адрес компьютера ssp.brown.edu. Для этого он обращается к своему DNS -серверу, отвечающему за область crec.mipt.ru, и передает ему нужный адрес в символьном виде. Если этот DNS-сервер не может сразу представить необходимый числовой адрес, он передает запрос DNS -серверу, отвечающему за область mipt.ru. Если и тот не в силах самостоятельно справиться с проблемой, он перенаправляет запрос серверу DNS, отвечающему за область 1-го ранга ru. Этот сервер может обратиться к серверу DNS, обслуживающему область 1-го ранга edu, который, наконец, затребует информацию от сервера DNSобласти brown.edu, где должен быть нужный числовой адрес.

В действительности, каждый сервер DNS имеет достаточно большой кэш, содержащий адреса серверов DNS для всех последних запросов. Поэтому реальная схема обычно существенно проще, из приведенной цепочки общения DNS -серверов выпадают многие звенья за счет обращения напрямую.

Рассмотренный способ разрешения адресов позволяет легко добавлять компьютеры в сеть и исключать их из сети, так как для этого необходимо внести изменения только на DNS -сервере соответствующей области.

Если DNS -сервер, отвечающий за какую-либо область, выйдет из строя, то может оказаться невозможным разрешение адресов для всех компьютеров этой области. Поэтому обычно назначается не один сервер DNS, а два – основной и запасной. В случае выхода из строя основного сервера его функции немедленно начинает выполнять запасной.

В реальных сетевых вычислительных системах обычно используется комбинация рассмотренных подходов. Для компьютеров, с которыми чаще всего приходится устанавливать связь, в специальном файле хранится таблица соответствий символьных и числовых адресов. Все остальные адреса разрешаются с использованием служб, аналогичных службе DNS. Способ построения удаленных адресов и методы разрешения адресов обычно определяются протоколами сетевого уровня эталонной модели.

Мы разобрались с проблемой удаленных адресов и знаем, как получить числовой удаленный адрес нужного нам компьютера. Давайте рассмотрим теперь проблему адресов локальных: как нам задать адрес процесса или объекта для хранения данных на удаленном компьютере, который в конечном итоге и должен получить переданную информацию.

#### Локальная адресация. Понятие порта

Во второй лекции мы говорили, что каждый процесс, существующий в данный момент в вычислительной системе, уже имеет собственный уникальный номер – PID. Но этот номер неудобно использовать в качестве локального адреса процесса при организации удаленной связи. Номер, который получает процесс при рождении, определяется моментом его запуска, предысторией работы вычислительного комплекса и является в значительной степени случайным числом, изменяющимся от запуска к запуску. Представьте себе, что адресат, с которым вы часто переписываетесь, постоянно переезжает с место на место, меняя адреса, так что, посылая очередное письмо, вы не можете с уверенностью сказать, где он сейчас проживает, и поймете все неудобство использования идентификатора процесса в качестве его локального адреса. Все сказанное выше справедливо и для идентификаторов промежуточных объектов, использующихся при локальном взаимодействии процессов в схемах с непрямой адресацией.

Для локальной адресации процессов и промежуточных объектов при удаленной связи обычно организуется новое специальное адресное пространство, например представляющее собой ограниченный набор положительных целочисленных значений или множество символических имен, аналогичных полным именам файлов в файловых системах. Каждый процесс, желающий принять участие в сетевом взаимодействии, после рождения закрепляет за собой один или несколько адресов в этом адресном пространстве. Каждому промежуточному объекту при его создании присваивается свой адрес из этого адресного пространства. При этом удаленные пользователи могут заранее договориться о том, какие именно адреса будут зарезервированы для данного процесса, независимо от времени его старта, или для данного объекта, независимо от момента его создания. Подобные адреса получили название портов, по аналогии с портами ввода-вывода.

Необходимо отметить, что в системе может существовать несколько таких адресных пространств для различных способов связи. При получении данных от удаленного процесса операционная система смотрит, на какой порт и для какого способа связи они были отправлены, определяет процесс, который заявил этот порт в качестве своего адреса, или объект, которому присвоен данный адрес, и доставляет полученную информацию адресату. Виды адресного пространства портов (т. е. способы построения локальных адресов ) определяются, как правило, протоколами транспортного уровня эталонной модели.

#### Полные адреса. Понятие сокета (socket)

Таким образом, полный адрес удаленного процесса или промежуточного объекта для конкретного способа связи с точки зрения операционных систем определяется парой адресов: <числовой адрес компьютера в сети, порт>. Подобная пара получила наименование **socket** (в переводе – "гнездо" или, как стали писать в последнее время, сокет ), а сам способ их использования – **организация связи с помощью** сокетов . В случае непрямой адресации с использованием промежуточных объектов сами эти объекты также принято называть сокетами. Поскольку разные протоколы транспортного уровня требуют разных адресных пространств портов, то для каждой пары надо указывать, какой транспортный протокол она использует, – говорят о разных типах сокетов.

В современных сетевых системах числовой адрес обычно получает не сам вычислительный комплекс, а его сетевой адаптер, с помощью которого комплекс подключается к линии связи. При наличии нескольких сетевых адаптеров для разных линий связи один и тот же вычислительный комплекс может иметь несколько числовых адресов. В таких системах полные адреса удаленного адресата (процесса или промежуточного объекта) задаются парами <числовой адрес сетевого адаптера, порт> и требуют доставки информации через указанный сетевой адаптер.

**76**

При наличии прямой линии связи между двумя компьютерами обычно не возникает вопросов о том, каким именно путем должна быть доставлена информация. Но, как уже упоминалось, одно из отличий взаимодействия удаленных процессов от взаимодействия процессов локальных состоит в использовании в большинстве случаев процессов-посредников, расположенных на вычислительных комплексах, не являющихся комплексами отправителя и получателя. В сложных топологических схемах организации сетей информация между двумя компьютерами может передаваться по различным путям. Возникает вопрос: как организовать работу операционных систем на комплексах -участниках связи (это могут быть конечные или промежуточные комплексы) для определения маршрута передачи данных? По какой из нескольких линий связи (или через какой сетевой адаптер) нужно отправить пакет информации? Какие протоколы маршрутизации возможны? Существует два принципиально разных подхода к решению этой проблемы: маршрутизация от источника передачи данных и одношаговая маршрутизация.

* Маршрутизация от источника передачи данных. При маршрутизации от источника данных полный маршрут передачи пакета по сети формируется на компьютере-отправителе в виде последовательности числовых адресов сетевых адаптеров, через которые должен пройти пакет, чтобы добраться до компьютера-получателя, и целиком включается в состав этого пакета. В этом случае промежуточные компоненты сети при определении дальнейшего направления движения пакета не принимают самостоятельно никаких решений, а следуют указаниям, содержащимся в пакете.
* Одношаговая маршрутизация. При одношаговой маршрутизации каждый компонент сети, принимающий участие в передаче информации, самостоятельно определяет, какому следующему компоненту, находящемуся в зоне прямого доступа, она должна быть отправлена. Решение принимается на основании анализа содержащегося в пакете адреса получателя. Полный маршрут передачи данных складывается из одношаговых решений, принятых компонентами сети.

Маршрутизация от источника передачи данных легко реализуется на промежуточных компонентах сети, но требует полного знания маршрутов на конечных компонентах. Она достаточно редко используется в современных сетевых системах, и далее мы ее рассматривать не будем.

Для работы алгоритмов одношаговой маршрутизации, которые являются основой соответствующих протоколов, на каждом компоненте сети, имеющем возможность передавать информацию более чем одному компоненту, обычно строится специальная таблица маршрутов (см. [рис. 14.3](https://www.intuit.ru/studies/courses/2192/31/lecture/994?page=6#image.14.3)). В простейшем случае каждая запись такой таблицы содержит: адрес вычислительного комплекса получателя; адрес компонента сети, напрямую подсоединенного к данному, которому следует отправить пакет, предназначенный для этого получателя; указание о том, по какой линии связи (через какой сетевой адаптер) должен быть отправлен пакет. Поскольку получателей в сети существует огромное количество, для сокращения числа записей в таблице маршрутизации обычно прибегают к двум специальным приемам.

Во-первых, числовые адреса топологически близко расположенных комплексов (например, комплексов, принадлежащих одной локальной вычислительной сети ) стараются выбирать из последовательного диапазона адресов. В этом случае запись в таблице маршрутизации может содержать не адрес конкретного получателя, а диапазон адресов для некоторой сети (номер сети).

Во-вторых, если для очень многих получателей в качестве очередного узла маршрута используется один и тот же компонент сети, а остальные маршруты выбираются для ограниченного числа получателей, то в таблицу явно заносятся только записи для этого небольшого количества получателей, а для маршрута, ведущего к большей части всей сети, делается одна запись – маршрутизация по умолчанию (default). Пример простой таблицы маршрутизации для некоторого комплекса некой абстрактной сети приведен ниже:

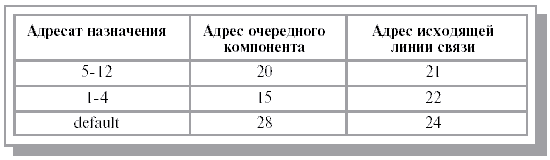


Рис. 14.3. Простая таблица маршрутизации

По способам формирования и использования таблиц маршрутизации алгоритмы одношаговой маршрутизации можно разделить на три класса:

* алгоритмы фиксированной маршрутизации ;
* алгоритмы простой маршрутизации ;
* алгоритмы динамической маршрутизации.

При фиксированной маршрутизации таблица, как правило, создается в процессе загрузки операционной системы. Все записи в ней являются статическими. Линия связи, которая будет использоваться для доставки информации от данного узла к некоторому узлу A в сети, выбирается раз и навсегда. Обычно линии выбирают так, чтобы минимизировать полное время доставки данных. Преимуществом этой стратегии является простота реализации. Основной же недостаток заключается в том, что при отказе выбранной линии связи данные не будут доставлены, даже если существует другой физический путь для их передачи.

В алгоритмах простой маршрутизации таблица либо не используется совсем, либо строится на основе анализа адресов отправителей приходящих пакетов информации. Различают несколько видов простой маршрутизации – случайную, лавинную и маршрутизацию по прецедентам. При случайной маршрутизации прибывший пакет отсылается в первом попавшемся направлении, кроме исходного. При лавинной маршрутизации один и тот же пакет рассылается по всем направлениям, кроме исходного. Случайная и лавинная маршрутизации, естественно, не используют таблиц маршрутов. При маршрутизации по прецедентам таблица маршрутизации строится по предыдущему опыту, исходя из анализа адресов отправителей приходящих пакетов. Если прибывший пакет адресован компоненту сети, от которого когда-либо приходили данные, то соответствующая запись об этом содержится в таблице маршрутов, и для дальнейшей передачи пакета выбирается линия связи, указанная в таблице. Если такой записи нет, то пакет может быть отослан случайным или лавинным способом. Алгоритмы простой маршрутизации действительно просты в реализации, но отнюдь не гарантируют доставку пакета указанному адресату за приемлемое время и по рациональному маршруту без перегрузки сети.

Наиболее гибкими являются алгоритмы динамической или адаптивной маршрутизации, которые умеют обновлять содержимое таблиц маршрутов на основе обработки специальных сообщений, приходящих от других компонентов сети, занимающихся маршрутизацией, удовлетворяющих определенному протоколу. Такие алгоритмы принято делить на два подкласса: алгоритмы дистанционно-векторные и алгоритмы состояния связей.

При дистанционно-векторной маршрутизации компоненты операционных систем на соседних вычислительных комплексах сети, занимающиеся выбором маршрута (их принято называть маршрутизатор или router ), периодически обмениваются векторами, которые представляют собой информацию о расстояниях от данного компонента до всех известных ему адресатов в сети. Под расстоянием обычно понимается количество переходов между компонентами сети ( hops ), которые необходимо сделать, чтобы достичь адресата, хотя возможно существование и других метрик, включающих скорость и/или стоимость передачи пакета по линии связи. Каждый такой вектор формируется на основании таблицы маршрутов. Пришедшие от других комплексов векторы модернизируются с учетом расстояния, которое они прошли при последней передаче. Затем в таблицу маршрутизации вносятся изменения, так чтобы в ней содержались только маршруты с кратчайшими расстояниями. При достаточно длительной работе каждый маршрутизатор будет иметь таблицу маршрутизации с оптимальными маршрутами ко всем потенциальным адресатам.

Векторно-дистанционные протоколы обеспечивают достаточно разумную маршрутизацию пакетов, но не способны предотвратить возможность возникновения маршрутных петель при сбоях в работе сети. Поэтому векторно-дистанционная маршрутизация может быть эффективна только в относительно небольших сетях. Для больших сетей применяются алгоритмы состояния связей, на каждом маршрутизаторе строящие графы сети, в качестве узлов которого выступают ее компоненты, а в качестве ребер, обладающих стоимостью, существующие между ними линии связи. Маршрутизаторы периодически обмениваются графами и вносят в них изменения. Выбор маршрута связан с поиском оптимального постоимости пути по такому графу.

**77**

Знание возможных угроз, а также уязвимых мест защиты, которые эти угрозы обычно эксплуатируют, необходимо для того, чтобы выбирать наиболее экономичные средства обеспечения безопасности.

Считается, что безопасная система должна обладать свойствами конфиденциальности, доступности и целостности. Любое потенциальное действие, которое направлено на нарушение конфиденциальности, целостности и доступности информации, называется угрозой . Реализованная угроза называется атакой .

Конфиденциальная ( confidentiality ) система обеспечивает уверенность в том, что секретные данные будут доступны только тем пользователям, которым этот доступ разрешен (такие пользователи называются авторизованными). Под доступностью ( availability ) понимают гарантию того, что авторизованным пользователям всегда будет доступна информация, которая им необходима. И наконец, целостность ( integrity ) системы подразумевает, что неавторизованные пользователи не могут каким-либо образом модифицировать данные.

Защита информации ориентирована на борьбу с так называемыми умышленными угрозами, то есть с теми, которые, в отличие от случайных угроз (ошибок пользователя, сбоев оборудования и др.), преследуют цель нанести ущерб пользователям ОС.

Умышленные угрозы подразделяются на активные и пассивные. Пассивная угроза – несанкционированный доступ к информации без изменения состояния системы, активная – несанкционированное изменение системы. Пассивные атаки труднее выявить, так как они не влекут за собой никаких изменений данных. Защита против пассивных атак базируется на средствах их предотвращения.

Можно выделить несколько типов угроз. Наиболее распространенная угроза – попытка проникновения в систему под видом легального пользователя, например попытки угадывания и подбора паролей. Более сложный вариант – внедрение в систему программы, которая выводит на экран слово login. Многие легальные пользователи при этом начинают пытаться входить в систему, и их попытки могут протоколироваться. Такие безобидные с виду программы, выполняющие нежелательные функции, называются "троянскими конями". Иногда удается торпедировать работу программы проверки пароля путем многократного нажатия клавиш del, break, cancel и т. д. Для защиты от подобных атак ОС запускает процесс, называемый аутентификацией пользователя (см. лекцию 16, раздел "Идентификация и аутентификация").

Угрозы другого рода связаны с нежелательными действиями легальных пользователей, которые могут, например, предпринимать попытки чтения страниц памяти, дисков и лент, которые сохранили информацию, связанную с предыдущим использованием. Защита в таких случаях базируется на надежной системе авторизации (см. лекцию 16, раздел "Авторизация. Разграничение доступа к объектам ОС"). В эту категорию также попадают атаки типа отказ в обслуживании, когда сервер затоплен мощным потоком запросов и становится фактически недоступным для отдельных авторизованных пользователей.

Наконец, функционирование системы может быть нарушено с помощью программ-вирусов или программ-"червей", которые специально предназначены для того, чтобы причинить вред или недолжным образом использовать ресурсы компьютера. Общее название угроз такого рода – вредоносные программы (malicious software). Обычно они распространяются сами по себе, переходя на другие компьютеры через зараженные файлы, дискеты или по электронной почте. Наиболее эффективный способ борьбы с подобными программами – соблюдение правил "компьютерной гигиены". Многопользовательские компьютеры меньше страдают от вирусов по сравнению с персональными, поскольку там имеются системные средства защиты.

**78**

Многие службы информационной безопасности, такие как контроль входа в систему, разграничение доступа к ресурсам, обеспечение безопасного хранения данных и ряд других, опираются на использование криптографических алгоритмов. Имеется обширная литература поэтому актуальному для безопасности информационных систем вопросу.

**Шифрование** – процесс преобразования сообщения из открытого текста (plaintext) в шифротекст (ciphertext) таким образом, чтобы:

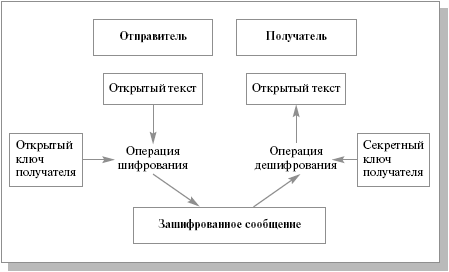
* его могли прочитать только те стороны, для которых оно предназначено;
* проверить подлинность отправителя (аутентификация);
* гарантировать, что отправитель действительно послал данное сообщение.

В алгоритмах шифрования предусматривается наличие ключа. Ключ – это некий параметр, не зависящий от открытого текста. Результат применения алгоритма шифрования зависит от используемого ключа. В криптографии принято правило Кирхгофа: "Стойкость шифра должна определяться только секретностью ключа". Правило Кирхгофа подразумевает, что алгоритмы шифрования должны быть открыты.

В методе шифрования с **секретным** или симметричным ключом имеется один ключ, который используется как для шифрования, так и для расшифровки сообщения. Такой ключ нужно хранить в секрете. Это затрудняет использование системы шифрования, поскольку ключи должны регулярно меняться, для чего требуется их секретное распространение. Наиболее популярные алгоритмы шифрования с секретным ключом: DES, TripleDES, ГОСТ и ряд других.

Часто используется шифрование с помощью односторонней функции, называемой также хеш- или дайджест-функцией. Применение этой функции к шифруемым данным позволяет сформировать небольшой дайджест из нескольких байтов, по которому невозможно восстановить исходный текст. Получатель сообщения может проверить целостность данных, сравнивая полученный вместе с сообщением дайджест с вычисленным вновь при помощи той же односторонней функции. Эта техника активно используется для контроля входа в систему. Например, пароли пользователей хранятся на диске в зашифрованном односторонней функцией виде. Наиболее популярные хеш-функции: MD4, MD5 и др.

В системах шифрования с **открытым** или **асимметричным ключом** (public/ assymmetric key) используется два ключа (см. рис. 15.1). Один из ключей, называемый открытым, несекретным, используется для шифрования сообщений, которые могут быть расшифрованы только с помощью секретного ключа, имеющегося у получателя, для которого предназначено сообщение. Иногда поступают по-другому. Для шифрования сообщения используется секретный ключ, и если сообщение можно расшифровать с помощью открытого ключа, подлинность отправителя будет гарантирована (система электронной подписи). Этот принцип изобретен Уитфилдом Диффи (Whitfield Diffie) и Мартином Хеллманом (Martin Hellman) в 1976 г.



**Рис. 15.1.** Шифрование открытым ключом

Использование открытых ключей снимает проблему обмена и хранения ключей, свойственную системам с симметричными ключами. Открытые ключи могут храниться публично, и каждый может послать зашифрованное открытым ключом сообщение владельцу ключа. Однако расшифровать это сообщение может только владелец открытого ключа при помощи своего секретного ключа, и никто другой. Несмотря на очевидные удобства, связанные с хранением и распространением ключей, асимметричные алгоритмы гораздо менее эффективны, чем симметричные, поэтому во многих криптографических системах используются оба метода.

**79**

Идея, положенная в основу метода, состоит в том, чтобы найти такую функцию y=\Phi(x), для которой получение обратной функции x=f^{-1}(y) было бы в общем случае очень сложной задачей (NP-полной задачей). Например, получить произведение двух чисел n=pxq просто, а разложить n на множители, если p и q достаточно большие простые числа, – NP-полная задача с вычислительной сложностью ~ n^{10}. Однако если знать некую секретную информацию, то найти обратную функцию x=f^{-1}(y) существенно проще. Такие функции также называют односторонними функциями с лазейкой или потайным ходом.

Применяемые в RSA прямая и обратная функции просты. Они базируются на применении теоремы Эйлера из теории чисел.

Прежде чем сформулировать теорему Эйлера, необходимо определить важную функцию \Phi(n) из теории чисел, называемую функцией Эйлера. Это число взаимно простых (взаимно простыми называются целые числа, не имеющие общих делителей) с n целых чисел, меньших n. Например, \Phi(7)=6. Очевидно, что, если p и q – простые числа и p \ne q, то \Phi(p)=p-1, и \Phi(pq)=(p-1)(q-1).

#### Теорема Эйлера

Теорема Эйлера утверждает, что для любых взаимно простых чисел x и n
(x < n)

x^{\Phi(n)} \mod n = 1

или в более общем виде

x^{k\Phi(n)+1} \mod n = 1

Сформулируем еще один важный результат. Для любого m>0 и 0<e<m, где e и m взаимно просты, найдется единственное 0<d<m, такое, что

de \mod m = 1.

Здесь d легко можно найти по обобщенному алгоритму Евклида (см., например, Д. Кнут. Искусство программирования на ЭВМ, т.2, 4.5.2). Известно, что вычислительная сложность алгоритма Евклида ~ ln n.

Подставляя \Phi(n) вместо m, получим de \mod \Phi(n)=1

или

de = k \Phi(n)+1

Тогда прямой функцией будет

\Phi(x) = x^e \mod n

где x – положительное целое, x < n=pq, p и q – целые простые числа и, следовательно,

\Phi (n)=(p-1)(q-1)

где e – положительное целое и e < \Phi(n). Здесь e и n открыты. Однако p и q неизвестны (чтобы их найти, нужно выполнить разбиение n на множители), следовательно, неизвестна и \Phi(n), а именно они и составляют потайной ход.

Вычислим обратную функцию

\Phi^{-1}(y) = y^d \mod n = x^{ed} \mod n = x^{k \Phi (n)+1} \mod n = x

Последнее преобразование справедливо, поскольку x < n и x и n взаимно просты.

При практическом использовании алгоритма RSA вначале необходимо выполнить генерацию ключей. Для этого нужно:

1. Выбрать два очень больших простых числа p и q ;
2. Вычислить произведение n=pq ;
3. Выбрать большое случайное число d, не имеющее общих сомножителей с числом (p-1)(q-1) ;
4. Определить число e, чтобы выполнялось
5. (ed) \mod ((p-1)(q-1))=1.

Тогда открытым ключом будут числа e и n, а секретным ключом – числа d и n.

Теперь, чтобы зашифровать данные по известному ключу {e,n}, необходимо сделать следующее.

* Разбить шифруемый текст на блоки, где i-й блок представить в виде числа M, величина которого меньше, чем n. Это можно сделать различными способами, например используя вместо букв их номера в алфавите.
* Зашифровать текст, рассматриваемый как последовательность чисел m(i), по формуле c(i)=(m(i)^{e}) \mod n.

Чтобы расшифровать эти данные, используя секретный ключ {d,n}, необходимо вычислить: m(i) = (c(i)^{d}) \mod n. В результате будет получено множество чисел m(i), которые представляют собой часть исходного текста.

**80**

Для начала рассмотрим проблему контроля доступа в систему. Наиболее распространенным способом контроля доступа является процедура регистрации. Обычно каждый пользователь в системе имеет уникальный идентификатор. Идентификаторы пользователей применяются с той же целью, что и идентификаторы любых других объектов, файлов, процессов. Идентификация заключается в сообщении пользователем своего идентификатора. Для того чтобы установить, что пользователь именно тот, за кого себя выдает, то есть что именно ему принадлежит введенный идентификатор, в информационных системах предусмотрена процедура аутентификации (authentication, опознавание, в переводе с латинского означает "установление подлинности"), задача которой - предотвращение доступа к системе нежелательных лиц.

Обычно аутентификация базируется на одном или более из трех пунктов:

* то, чем пользователь владеет (ключ или магнитная карта);
* то, что пользователь знает (пароль);
* атрибуты пользователя (отпечатки пальцев, подпись, голос).

#### Пароли, уязвимость паролей

Наиболее простой подход к аутентификации - применение пользовательского пароля.

Когда пользователь идентифицирует себя при помощи уникального идентификатора или имени, у него запрашивается пароль. Если пароль, сообщенный пользователем, совпадает с паролем, хранящимся в системе, система предполагает, что пользователь легитимен. Пароли часто используются для защиты объектов в компьютерной системе в отсутствие более сложных схем защиты.

Недостатки паролей связаны с тем, что трудно сохранить баланс между удобством пароля для пользователя и его надежностью. Пароли могут быть угаданы, случайно показаны или нелегально переданы авторизованным пользователем неавторизованному.

Есть два общих способа угадать пароль. Один связан со сбором информации о пользователе. Люди обычно используют в качестве паролей очевидную информацию (скажем, имена животных или номерные знаки автомобилей). Для иллюстрации важности разумной политики назначения идентификаторов и паролей можно привести данные исследований, проведенных в AT&T, показывающие, что из 500 попыток несанкционированного доступа около 300 составляют попытки угадывания паролей или беспарольного входа по пользовательским именам guest, demo и т. д.

Другой способ - попытаться перебрать все наиболее вероятные комбинации букв, чисел и знаков пунктуации (атака по словарю). Например, четыре десятичные цифры дают только 10 000 вариантов, более длинные пароли, введенные с учетом регистра символов и пунктуации, не столь уязвимы, но тем не менее таким способом удается разгадать до 25% паролей. Чтобы заставить пользователя выбрать трудноугадываемый пароль, во многих системах внедрена реактивная проверка паролей, которая при помощи собственной программы-взломщика паролей может оценить качество пароля, введенного пользователем.

Несмотря на все это, пароли распространены, поскольку они удобны и легко реализуемы.

##### Шифрование пароля

Для хранения секретного списка паролей на диске во многих ОС используется криптография. Система задействует одностороннюю функцию, которую просто вычислить, но для которой чрезвычайно трудно (разработчики надеются, что невозможно) подобрать обратную функцию.

Например, в ряде версий Unix в качестве односторонней функции используется модифицированный вариант алгоритма DES. Введенный пароль длиной до 8 знаков преобразуется в 56-битовое значение, которое служит входным параметром для процедуры crypt(), основанной на этом алгоритме. Результат шифрования зависит не только от введенного пароля, но и от случайной последовательности битов, называемой привязкой (переменная salt). Это сделано для того, чтобы решить проблему совпадающих паролей. Очевидно, что саму привязку после шифрования необходимо сохранять, иначе процесс не удастся повторить. Модифицированный алгоритм DES выполняется, имея входное значение в виде 64-битового блока нулей, с использованием пароля в качестве ключа, а на каждой следующей итерации входным параметром служит результат предыдущей итерации. Всего процедура повторяется 25 раз. Полученное 64-битовое значение преобразуется в 11 символов и хранится рядом с открытой переменной salt.

В ОС Windows NT преобразование исходного пароля также осуществляется многократным применением алгоритма DES и алгоритма MD4.

Хранятся только кодированные пароли. В процессе аутентификации представленный пользователем пароль кодируется и сравнивается с хранящимися на диске. Таким образом, файл паролей нет необходимости держать в секрете.

При удаленном доступе к ОС нежелательна передача пароля по сети в открытом виде. Одним из типовых решений является использование криптографических протоколов. В качестве примера можно рассмотреть протокол опознавания с подтверждением установления связи путем вызова - CHAP (Challenge Handshake Authentication Protocol).

Опознавание достигается за счет проверки того, что у пользователя, осуществляющего доступ к серверу, имеется секретный пароль, который уже известен серверу.

Пользователь инициирует диалог, передавая серверу свой идентификатор. В ответ сервер посылает пользователю запрос (вызов), состоящий из идентифицирующего кода, случайного числа и имени узла сервера или имени пользователя. При этом пользовательское оборудование в результате запроса пароля пользователя отвечает следующим ответом, зашифрованным с помощью алгоритма одностороннего хеширования, наиболее распространенным видом которого является MD5. После получения ответа сервер при помощи той же функции с теми же аргументами шифрует собственную версию пароля пользователя. В случае совпадения результатов вход в систему разрешается. Существенно, что незашифрованный пароль при этом по каналу связи не посылается.

**81**

После успешной регистрации система должна осуществлять авторизацию (authorization) - предоставление субъекту прав на доступ к объекту. Средства авторизации контролируют доступ легальных пользователей к ресурсам системы, предоставляя каждому из них именно те права, которые были определены администратором, а также осуществляют контроль возможности выполнения пользователем различных системных функций. Система контроля базируется на общей модели, называемой матрицей доступа. Рассмотрим ее более подробно.

Как уже говорилось в предыдущей лекции, компьютерная система может быть смоделирована как набор субъектов (процессы, пользователи) и объектов. Под объектами мы понимаем как ресурсы оборудования (процессор, сегменты памяти, принтер, диски и ленты), так и программные ресурсы (файлы, программы, семафоры), то есть все то, доступ к чему контролируется. Каждый объект имеет уникальное имя, отличающее его от других объектов в системе, и каждый из них может быть доступен через хорошо определенные и значимые операции.

Операции зависят от объектов. Hапример, процессор может только выполнять команды, сегменты памяти могут быть записаны и прочитаны, считыватель магнитных карт может только читать, а файлы данных могут быть записаны, прочитаны, переименованы и т. д.

Желательно добиться того, чтобы процесс осуществлял авторизованный доступ только к тем ресурсам, которые ему нужны для выполнения его задачи. Это требование минимума привилегий, уже упомянутое в предыдущей лекции, полезно с точки зрения ограничения количества повреждений, которые процесс может нанести системе. Hапример, когда процесс P вызывает процедуру А, ей должен быть разрешен доступтолько к переменным и формальным параметрам, переданным ей, она не должна иметь возможность влиять на другие переменные процесса. Аналогично компилятор не должен оказывать влияния на произвольные файлы, а только на их хорошо определенное подмножество (исходные файлы, листинги и др.), имеющее отношение к компиляции. С другой стороны, компилятор может иметь личные файлы, используемые для оптимизационных целей, к которым процесс Р не имеет доступа.

Различают дискреционный (избирательный) способ управления доступом и полномочный (мандатный).

При дискреционном доступе, подробно рассмотренном ниже, определенные операции над конкретным ресурсом запрещаются или разрешаются субъектам или группам субъектов. С концептуальной точки зрения текущее состояние прав доступа при дискреционном управлении описывается матрицей, в строках которой перечислены субъекты, в столбцах - объекты, а в ячейках - операции, которые субъект может выполнить над объектом.

Полномочный подход заключается в том, что все объекты могут иметь уровни секретности, а все субъекты делятся на группы, образующие иерархию в соответствии с уровнем допуска к информации. Иногда это называют моделью многоуровневой безопасности, которая должна обеспечивать выполнение следующих правил.

* Простое свойство секретности. Субъект может читать информацию только из объекта, уровень секретности которого не выше уровня секретности субъекта. Генерал читает документы лейтенанта, но не наоборот.
* \*-свойство. Субъект может записывать информацию в объекты только своего уровня или более высоких уровней секретности. Генерал не может случайно разгласить нижним чинам секретную информацию.

Некоторые авторы утверждают [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)], что последнее требование называют \*-свойством, потому что в оригинальном докладе не смогли придумать для него подходящего названия. В итоге во все последующие документы и монографии оно вошло как \*-свойство.

Отметим, что данная модель разработана для хранения секретов, но не гарантирует целостности данных. Например, здесь лейтенант имеет право писать в файлы генерала. Более подробно о реализации подобных формальных моделей рассказано в [[Столлингс, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.29)], [[Таненбаум, 2002](https://www.intuit.ru/studies/courses/2192/31/literature#literature.30)].

Большинство операционных систем реализуют именно дискреционное управление доступом. Главное его достоинство - гибкость, основные недостатки - рассредоточенность управления и сложность централизованного контроля.

#### Домены безопасности

Чтобы рассмотреть схему дискреционного доступа более детально, введем концепцию домена безопасности (protection domain). Каждый домен определяет набор объектов и типов операций, которые могут производиться над каждым объектом. Возможность выполнять операции над объектом есть права доступа, каждое из которых есть упорядоченная пара <object-name, rights-set>. Домен, таким образом, есть набор прав доступа. Hапример, если домен D имеет права доступа <file F, {read, write}>, это означает, что процесс, выполняемый в домене D, может читать или писать в файл F, но не может выполнять других операций над этим объектом. Пример доменов можно увидеть на [рис.16.1](https://www.intuit.ru/studies/courses/2192/31/lecture/998?page=2#image.16.1).

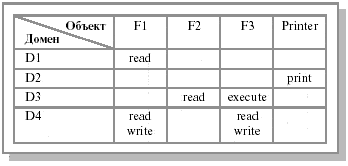


Рис. 16.1. Специфицирование прав доступа к ресурсам

Связь конкретных субъектов, функционирующих в операционных системах, может быть организована следующим образом.

* Каждый пользователь может быть доменом. В этом случае набор объектов, к которым может быть организован доступ, зависит от идентификации пользователя.
* Каждый процесс может быть доменом. В этом случае набор доступных объектов определяется идентификацией процесса.
* Каждая процедура может быть доменом. В этом случае набор доступных объектов соответствует локальным переменным, определенным внутри процедуры. Заметим, что когда процедура выполнена, происходит смена домена.

Рассмотрим стандартную двухрежимную модель выполнения ОС. Когда процесс выполняется в режиме системы (kernel mode), он может выполнять привилегированные инструкции и иметь полный контроль над компьютерной системой. С другой стороны, если процесс выполняется в пользовательском режиме, он может вызывать только непривилегированные инструкции. Следовательно, он может выполняться только внутри предопределенного пространства памяти. Наличие этих двух режимов позволяет защитить ОС (kernel domain) от пользовательских процессов (выполняющихся в user domain). В мультипрограммных системах двух доменов недостаточно, так как появляется необходимость защиты пользователей друг от друга. Поэтому требуется более тщательно разработанная схема.