1) Что такое операционная система

**Операционная система**-это резидентная в памяти компьютера программа, которая выполняет 2 мало связанные функции

* обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины
* повышение эффективности использования компьютера путем рационального управления его ресурсами

Структура вычислительной системы

1)Техническое обеспечение: процессор, память, монитор, дисковые устройства и т.д., объединенные магистральным соединением, которое называется шиной.

2)Программное обеспечение. Все программное обеспечение принято делить на две части: прикладное и системное. Это деление является условным.



* Операционная система как виртуальная машина

Архитектура компьютера очень сложна и неудобна для использования прикладных программ. Например, вместо того чтобы разбираться как работает жесткий диск и учитывать куда нужно переставить головку считывателя, программисту предоставляется высокоуровневая абстракция, которая позволяет ему без усилий считывать и записывать данные на диск, не заставляя программиста знать внутреннее устройство. Таким образом работают абстракции объединенные в одну **операционную систему.** С которой работать проще чем напрямую с оборудованием.

* Операционная система как менеджер ресурсов

Операционная система предназначена для управления всеми частями весьма сложной архитектуры компьютера. Представим, к примеру, что произойдет, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер. Мы получили бы мешанину строчек и страниц, выведенных различными программами. Операционная система предотвращает такого рода хаос за счет буферизации информации, предназначенной для печати, на диске и организации очереди на печать. Для многопользовательских компьютеров необходимость управления ресурсами и их защиты еще более очевидна. 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. Необходимо обеспечить сохранность информации на диске, чтобы никто не мог удалить или повредить чужие файлы. Нельзя разрешить программам одних пользователей произвольно вмешиваться в работу программ других пользователей. Нужно пресекать попытки несанкционированного использования вычислительной системы. 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 постоянно взаимодействующая со всеми прикладными программами(резидентные программы)
* Во многих операционных системах постоянно работает на компьютере лишь часть операционной системы , которую принято называть ее ядром

( взято из лекций федорова и в интернете тоже самое)

2)Краткая история эволюции вычислительных систем

1-ый период ( 1945-1955 года)

* Ламповые машины
* Нет разделения персонала
* Ввод программы перфокартами
* Одновременное выполнение только одной операции
* Появление прообразов первых компиляторов
* Нет операционных систем

-----------------------------------------------------------------------------------

*Научно-исследовательская работа в области вычислительной техники*

*-----------------------------------------------------------------------------------*

* Первый период (1945–1955 гг.). Ламповые машины. Операционных систем нет
  + В середине 40-х были созданы первые ламповые вычислительные устройства и появился принцип программы, хранящейся в памяти машины (John Von Neumann, июнь 1945 г.). Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. Программа загружалась в память машины в лучшем случае с колоды перфокарт, а обычно с помощью панели переключателей.
  + Вычислительная система выполняла одновременно только одну операцию (ввод-вывод или собственно вычисления). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951–1952 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Nat Rochester разрабатывает Ассемблер для IBM-701.
  + Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно. Такой режим работы называется последовательной обработкой данных. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

2-й период ( 1955- начало 60-х )

* Транзисторные машины
* Происходит разделение персонала
* Бурное развитие алгоритмических языков
* Ввод задания колодой перфокарт
* Вывод результатов на печать
* Пакеты заданий и системы пакетной разработки

--------------------------------------------------------------------------------------------

*Начало использования ЭВМ в научных и коммерческих целях*

--------------------------------------------------------------------------------------------

* Второй период (1955 г.–начало 60-х). Компьютеры на основе транзисторов. Пакетные операционные системы
  + С середины 50-х годов появились полупроводниковые элементы. Применение транзисторов вместо часто перегоравших электронных ламп привело к повышению надежности компьютеров. Одновременно наблюдается бурное развитие алгоритмических языков (LISP, COBOL, ALGOL-60, PL-1 и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования.
  + Изменяется сам процесс прогона программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт и указывает необходимые ресурсы. Такая колода получает название задания. Оператор загружает задание в память машины и запускает его на исполнение. Полученные выходные данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно продолжительное) время.
  + Смена запрошенных ресурсов вызывает приостановку выполнения программ, в результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая пакет заданий.
  + Появляются первые системы пакетной обработки, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине.

3-ий период ( начало 60х - 1980 года)

* Машины на интегральных схемах
* Использование спулинга
* Планирование заданий
* Мультипрограммные пакетные системы
* Третий период (начало 60-х – 1980 г.). Компьютеры на основе интегральных микросхем. Первые многозадачные ОС
  + В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Растет сложность и количество задач, решаемых компьютерами. Повышается производительность процессоров.
  + Повышению эффективности использования процессорного времени мешает низкая скорость работы механических устройств ввода-вывода . Вместо непосредственного чтения пакета заданий с перфокарт в память начинают использовать его предварительную запись, сначала на магнитную ленту, а затем и на диск.
  + Когда в процессе выполнения задания требуется ввод данных, они читаются с диска. Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а печатается только после завершения задания. Вначале действительные операции ввода-вывода осуществлялись в режиме off-line, то есть с использованием других, более простых, отдельно стоящих компьютеров.
  + Магнитные ленты были устройствами последовательного доступа, то есть информация считывалась с них в том порядке, в каком была записана. Появление магнитного диска, для которого не важен порядок чтения информации, то есть устройства прямого доступа, привело к дальнейшему развитию вычислительных систем. При обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их ввода. При обработке пакета заданий на магнитном диске появилась возможность выбора очередного выполняемого задания. Пакетные системы начинают заниматься планированием заданий: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т.д. на счет выбирается то или иное задание

##### 4-ый период (с 1980 г. по настоящее время)

##### Четвертый период (с 1980 г. по настоящее время). Персональные компьютеры. Классические, сетевые и распределенные системы

-появление компьютеров доступных каждому человеку

-появление больших интегральных схем

-разработка “дружественного “ по

-развитие сети компьютеров

* появление больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и снижение стоимости микросхем. Компьютер по цене и простоте эксплуатации стал доступен отдельному человеку, а не отделу предприятия или университета. Наступила эра персональных компьютеров. Первоначально персональные компьютеры предназначались для использования одним пользователем в однопрограммном режиме, что повлекло за собой деградацию архитектуры этих ЭВМ и их операционных систем (в частности, пропала необходимость защиты файлов и памяти, планирования заданий и т. п.).
* Компьютеры стали использоваться не только специалистами, что потребовало разработки "дружественного" программного обеспечения.
* Однако рост сложности и разнообразия задач, решаемых на персональных компьютерах, необходимость повышения надежности их работы привели к возрождению практически всех черт, характерных для архитектуры больших вычислительных систем.
* В середине 80-х стали бурно развиваться сети компьютеров, в том числе персональных, работающих под управлением сетевых или распределенных операционных систем.
* В сетевых операционных системах пользователи могут получить доступ к ресурсам другого сетевого компьютера, только они должны знать об их наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения 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.
* В дальнейшем автономные операционные системы мы будем называть классическими операционными системами.

3)Основные понятия, концепции ОС

* Системные вызовы

**Системные вызовы** – 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 предается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек – сделать системный вызов похожим на обычный вызов подпрограммы.

* Прерывания

**Прерывания** – 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 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 неисправимых исключительных ситуаций 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.

* Процессы, нити

<http://uinc.ru/articles/34/>

??

4) Архитектурные особенности ОС

* Монолитное ядро

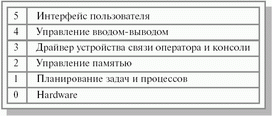
*Монолитное ядро* представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом,***монолитное ядро***– это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют 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 монолитным ядром является 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.

*Монолитное ядро*– старейший способ организации операционных *систем*. Примером систем с монолитным *ядром является* большинство Unix-систем.

* 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 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

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.

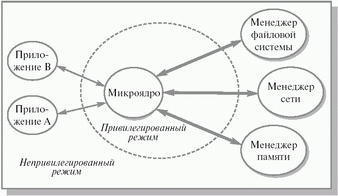


**Недостатком** таких операционных систем является снижение эффективности виртуальных машин по сравнению с реальным компьютером, и, как правило, они очень громоздки.

**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** 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.



Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

**Достоинства:** высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства.Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне не привилегированной программы менее опасна, чем отказ на уровне режима ядра.

* Смешанные системы

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. 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.

+

Таким образом, Windows NT можно с полным правом назвать гибридной операционной системой.

5) Классификация ОС

* Реализация многозадачности
* Организация рационального использования ресурсов компьютера – одна из важнейших функций ОС.
* **Многозадачность (мультипрограммирование) –** способ организации вычислительного процесса, при котором на одном (или нескольких) процессоре (-ах) выполняется несколько задач. Данные задачи совместно используют не только ресурсы процессора, но также оперативную и внешнюю память, устройства ввода-вывода и т.д.
* Организациявычислительного процесса характеризуется несколькими критериями эффективности:
  + *Пропускная способность* – количество выполняемых задач в единицу времени;
  + *Удобство работы пользователя*
  + *Реактивность системы* – способность системы выдерживать промежутки времени между запуском программы и получением результата.
* 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 многопользовательских ***систем механизмов*** защиты персональных данных каждого пользователя.

* Многопроцессорная обработка
* **Мультипроцессорная обработка** – способ организации вычислительного процесса в системах с несколькими процессорами (разделение процедуры обработки на разных процессорах).
* При мультипроцессорной обработке усложняется задача управления ресурсами – планирование процессов для нескольких процессоров, использующих общую оперативную и внешнюю память, другие устройства. Необходимо планировать средства блокировки при доступе к разделяемым информационным структурам ядра.
* Современные серверные ОС включают поддержку мультипроцессорной обработки – Sun Solaris, IBM OS/2, Novell NetWare, Microsoft Windows.
* Системы реального времени
* Системы реального времени представляют собой вариант систем с разделением времени выполнения в случае, если задача обработки информации должна быть выполнена в течение некоторого предельно допустимого времени.
* Мультипрограммная смесь в системах реального времени – фиксированный набор заранее разработанных программ. Выбор задачи осуществляется по прерываниям или в соответствии с расписанием плановых работ.
* Способность ОС к быстрому ответу зависит от скорости переключения с одной задачи на другую, в частности, от скорости обработки прерываний.
* В системах реального времени при планировании загрузки устройств устанавливается некоторый запас вычислительной мощности системы, для того, чтобы, в случае пиковой нагрузки, система могла осуществить прерывание работы приложения.
* Примера таких систем – управление техническими объектами или технологическими процессами. В таких системах критерий эффективности – способность выдержать заданные интервалы времени.

6) Понятие процесса

* Основная задача ОС – распределение ресурсов между процессами и потоками.
  + **Процесс**, связан с программным кодом исполняемого модуля и рассматривается ОС как заявка на потребление всех видов ресурсов, кроме процессорного времени. Для изоляции процессов друг от друга ОС обеспечивает каждый процесс отдельным виртуальным адресным пространством. Один процесс не может получить прямого доступа к командам и данным другого процесса.
  + **Поток** представляет собой процесса, способ распараллеливания вычислений. Поток – последовательность команд, выполняемых процессором.
* ОС распределяет процессорное время между потоками. Процессу назначается адресное пространство и набор ресурсов, которые используются всеми его потомками.
* **Процесс** характеризуется 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.
* Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра, как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

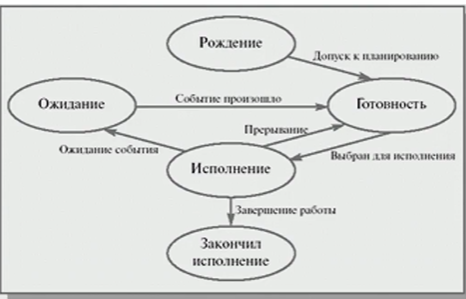
7) Состояния процесса

* Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой.
* Пока один процесс выполняется, остальные ждут своей очереди.
* Каждый процесс может находиться как минимум в двух состояниях: процесс исполняется и процесс не исполняется.



Модель процессов

* Всякий новый процесс, появляющийся в системе, попадает в состояние готовности. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние исполнение.
* В состоянии исполнения происходит непосредственное выполнение программного кода процесса. Выйти из состояния выполнения процесс может по трем причинам:
  + операционная система прекращает его деятельность;
  + он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ожидание;
  + в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние готовность.
* Модель должна описывать поведение процессов не только во время их существования, но и при появлении процесса в системе и его исчезновении. Для полноты картины вводится еще два состояния процессов: рождение и закончил исполнение



Состояния процессов

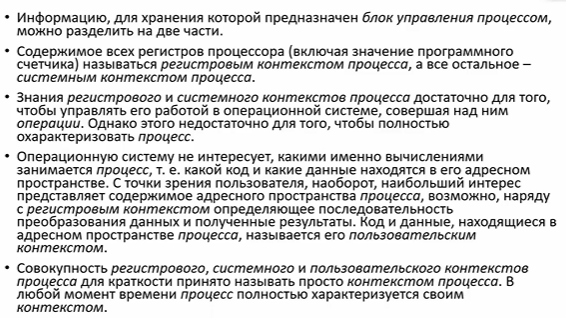
* При рождении процесс получает в свое распоряжение **адресное пространство**, в которое загружается программный код процесса;
* Процессу выделяются **стек** и **системные ресурсы**; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние готовность. При завершении своей деятельности процесс из состояния исполнение попадает в состояние закончил исполнение.
* В операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое.
* Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9.

8) Операции над процессами и связанные с ними понятия

* Набор операций
* Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается ОС, совершая операции над ними.
* Операции можно объединить 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).
* Операции создания и завершения процесса являются **одноразовыми**.
* Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются **многоразовыми**.

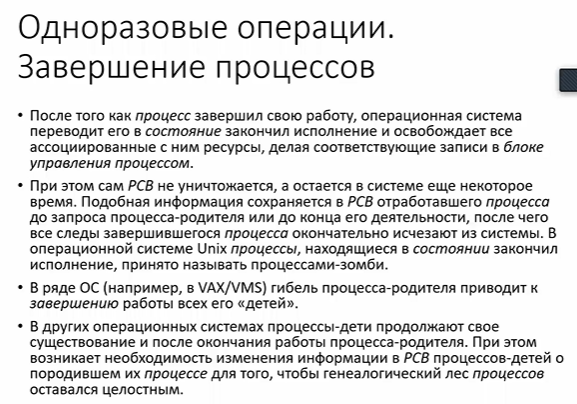
Управление процессами

* 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).
* Process Control Block и контекст процесса

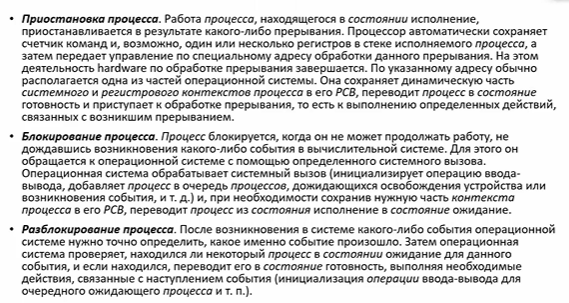


* Одноразовые операции





* Многоразовые операции



* Переключение контекста
* Мультипрограммная операционная система выполняет цепочку *операций*, выполняемых над различными *процессами*, и переключая процессор с одного *процесса* на другой.
* *Операция разблокирования процесса*, ожидающего ввода-вывода может быть представлена следующим образом.
* При исполнении процессором некоторого *процесса* 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 и осуществляет *запуск* приостановленного или нового *процесса*, выбранного при выполнении планирования
* Для корректного переключения процессора с одного *процесса* на другой ОС должна сохранить *контекст* исполняющегося *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).

9) Уровни планирования

* Для реализации многозадачности ОС выполняет планирование и диспетчеризацию потоков.
* **Планирование** – определение момента времени для смены текущего потока, а также выбор нового потока для выполнения.
* **Диспетчеризация** – реализация найденного в ходе планирования решения, т.е. организация переключения с одного потока на другой.
* Планирование может быть *динамическим –* решения принимаются системой на основе анализа текущей ситуации (характерен для универсальных ОС) или *статическим* – если потоки запускаются на основе заранее разработанного расписания (характерен для ОС реального времени).

10) Критерии планирования процессов и требования к алгоритмам

**Цели:**

1.**Справедливость**: гарантировать каждому заданию или процессу некоторую часть времени использования процессора в компьютерной системе , не допуская ситуаций, когда процесс одного пользователя постоянно занимает процессор , а процесс другого пользователя фактически не выполняется

2. **Эффективность**: постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению (в реальных ВС загрузка процессора колеблется от 40 до 90 %)

3. **Сокращение полного времени выполнения** (*turnaround time*): обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением

4. **Сокращение времени ожидания** (*waiting time*): минимизировать время, которое проводят процессы в состоянии ***готовность*** и задания в очереди для загрузки

5. **Сокращение времени отклика** (*response time*): минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя

Желательные свойства

1. Предсказуемость: одно и то же задание должно выполняться приблизительно за одно и то же время
2. Минимальные накладные расходы
3. Равномерная загрузка ресурсов ВС с предпочтением процессов, которые будут занимать мало используемые ресурсы
4. Масштабируемость: алгоритмы должны сохранять работоспособность при увеличении нагрузки

Вывод :

* Многие приведенные цели и свойства являются противоречивыми
* Улучшение работы алгоритма с точки зрения одного критерия ухудшает ее с точки зрения другого критерия
* Приспособление алгоритма под один класс задач, приводит к дискриминации задач другого класса

11) Параметры планирования процессов

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры.

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.

В краткосрочном планировании деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода.

12) Вытесняющее и невытесняющее планирование процессов

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).

В случаях 1 и 2 процесс, находившийся в состоянии исполнение, не может дальше исполняться, и операционная система вынуждена осуществлять планирование выбирая новый процесс для выполнения. В случаях 3 и 4 планирование может как проводиться, так и не проводиться, планировщик не вынужден обязательно принимать решение о выборе процесса для выполнения, процесс, находившийся в состоянии исполнение может просто продолжить свою работу. 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.

13) Алгоритмы планирования

* First-Come, First-Served (FCFS)

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой **FCFS** по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). 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 – FIFO, 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.

(где-то здесь мы рассматриваем пример и делаем вывод)

**Вывод**: среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. алгоритм FCFS практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в интерактивных процессах.

* 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.

**Вывод**: При очень больших величинах кванта времени, алгоритм RR вырождается в алгоритм FCFS.

* Shortest-Job-First (SJF)

Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность FCFS и RR алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название "кратчайшая работа первой" или Shortest Job First ( **SJF** ).

SJF-алгоритм краткосрочного планирования может быть как вытесняющим, так и не вытесняющим. При невытесняющем SJF - планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF - планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

**Вывод**: Основную сложность при реализации алгоритма SJF представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов.

14) Алгоритмы планирования

* Гарантированное планирование
* При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~1/N часть процессорного времени.
* Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: Ti – время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и \tau_i – суммарное процессорное время уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение Ti/N процессорного времени. Если

\tau_i << T_i/N то i -й пользователь несправедливо обделен процессорным временем. Если же

\tau_i >> T_i/N то система явно благоволит к пользователю с номером i.

* Вычислим для процессов каждого пользователя значение коэффициента справедливости

\tau_i N/T_i и будем предоставлять очередной квант времени готовому процессу с наименьшей величиной этого отношения.

* Предложенный алгоритм называют алгоритмом гарантированного планирования.

**Недостатки:**

-невозможно предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

#### 

* Приоритетное планирование
* 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 не вытесняющим.
* 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.

**Вывод :**Приоритеты , которые с течением времени не изменялись, 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. 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. Или вычислительную систему приходится выключать, и они теряются . Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии готовность.

* Многоуровневые очереди (Multilevel Queue)
* 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.
* Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)

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.

* Планирование процессов между очередями осуществляется на основе вытесняющего приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее приоритет. Если при работе процесса появляется другой процесс в какой-либо более приоритетной очереди, исполняющийся процесс вытесняется новым.
* 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. 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.

Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

15) Взаимодействующие процессы

Для достижения поставленной цели различные процессы 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, например, 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.

16) Категории средств обмена информацией

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.
* **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.

17) Логическая организация механизма передачи информации

* Установление связи

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: прямую и непрямую. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется **симметричной прямой адресацией** . **Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные**. Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется **асимметричной прямой адресацией** .

При **непрямой адресации** данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Единственное, что нужно для использования средства связи, – это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании непрямой адресации инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

* Информационная валентность процессов и средств связи

Понятно, что при прямой адресации только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При непрямой адресации может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь **симплексной** , двунаправленную связь с поочередной передачей информации в разных направлениях – **полудуплексной** , а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – **дуплексной** . Прямая и непрямая адресация не имеет непосредственного отношения к направленности связи.

**Валентность средств связи**- определяет число процессов , которые могут быть одновременно ассоциированными с конкретным средством связи

**Информационная валентность процессов**- определяет число средств связи одновременно ассоциированными двумя процессами

* Особенности передачи информации с помощью линий связи
* передача информации между процессами посредством линий связи является достаточно безопасной по сравнению с использованием разделяемой памяти и более информативной по сравнению с сигнальными средствами коммуникации.
* разделяемая память не может быть использована для связи процессов, функционирующих на различных вычислительных системах. Возможно, именно поэтому каналы связи из средств коммуникации процессов получили наибольшее распространение. Коснемся некоторых вопросов, связанных с логической реализацией канальных средств коммуникации.
* Буферизация, Поток ввода/вывода и сообщения

**Буферизация**

1. Буфер нулевой емкости 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 ). 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 pipe для 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.

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.

* Надежность средств связи, Завершение связи

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.

#### **Как завершается связь?**

#### Здесь нужно выделить два аспекта: требуются ли от процесса какие-либо специальные действия по прекращению использования средства коммуникации и влияет ли такое прекращение на поведение других процессов. Для способов связи, которые не подразумевали никаких инициализирующих действий, обычно ничего специального для окончания взаимодействия предпринимать не надо. Если же установление связи требовало некоторой инициализации, то, как правило, при ее завершении бывает необходимо выполнить ряд операций, например сообщить операционной системе об освобождении выделенного связного ресурса.

#### Если кооперативные процессы прекращают взаимодействие согласованно, то такое прекращение не влияет на их дальнейшее поведение. Иная картина наблюдается при несогласованном окончании связи одним из процессов. Если какой-либо из взаимодействующих процессов, не завершивших общение, находится в этот момент в состоянии ожидания получения данных либо попадает в такое состояние позже, то операционная система обязана предпринять некоторые действия для того, чтобы исключить вечное блокирование этого процесса. Обычно это либо прекращение работы ожидающего процесса, либо его извещение о том, что связи больше нет (например, с помощью передачи заранее определенного сигнала).

#### 

18) Нити исполнения. нужно короче сделать

любой отдельно взятый процесс в мультипрограммной системе никогда не может быть выполнен быстрее, чем при работе в однопрограммном режиме на том же вычислительном комплексе. Тем не менее, если алгоритм решения задачи обладает определенным внутренним параллелизмом, мы могли бы ускорить его работу, организовав взаимодействие нескольких процессов. Рассмотрим следующий пример. Пусть у нас есть следующая программа на псевдоязыке программирования:

Ввести массив a

Ввести массив b

Ввести массив c

a = a + b

c = a + c

Вывести массив c

При выполнении такой программы в рамках одного процесса этот процесс четырежды будет блокироваться, ожидая окончания операций ввода-вывода. Но наш алгоритм обладает внутренним параллелизмом. Вычисление суммы массивов a + b можно было бы выполнять параллельно с ожиданием окончания операции ввода массива c.

Ввести массив a

Ожидание окончания операции ввода

Ввести массив b

Ожидание окончания операции ввода

Ввести массив с

Ожидание окончания операции ввода a = a + b

c = a + c

Вывести массив с

Ожидание окончания операции вывода

Такое совмещение операций по времени можно было бы реализовать, используя два взаимодействующих процесса. Для простоты будем полагать, что средством коммуникации между ними служит разделяемая память. Тогда наши процессы могут выглядеть следующим образом.

Процесс 1 Процесс 2

Ввести массив a Ожидание ввода

Ожидание окончания массивов a и b

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания a = a + b

операции ввода

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Казалось бы, мы предложили конкретный способ ускорения решения задачи. Однако в действительности дело обстоит не так просто. Второй процесс должен быть создан, оба процесса должны сообщить операционной системе, что им необходима память, которую они могли бы разделить с другим процессом, и, наконец, нельзя забывать о переключении контекста. Поэтому реальное поведение процессов будет выглядеть примерно так.

Процесс 1 Процесс 2

Создать процесс 2

Переключение контекста

Выделение общей

памяти

Ожидание ввода

a и b

Переключение контекста

Выделение общей памяти

Ввести массив a

Ожидание окончания

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания

операции ввода

Переключение контекста

a = a + b

Переключение контекста

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Очевидно, что мы можем не только не выиграть во времени при решении задачи, но даже и проиграть, так как временные потери на создание процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, полученный за счет совмещения операций.

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.

Нить 1 Нить 2

Создать нить 2

Переключение контекста нитей

Ожидание ввода a и b

Переключение контекста нитей

Ввести массив a

Ожидание окончания

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания

операции ввода

Переключение контекста нитей

a = a + b

Переключение контекста нитей

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Различают операционные системы, поддерживающие нити на уровне ядра и на уровне библиотек. Все сказанное выше справедливо для операционных систем, поддерживающих нити на уровне ядра. В них планирование использования процессора происходит в терминах нитей, а управление памятью и другими системными ресурсами остается в терминах процессов. В операционных системах, поддерживающих нити на уровне библиотек пользователей, и планирование процессора, и управление системными ресурсами осуществляются в терминах процессов. Распределение использования процессора по нитям в рамках выделенного процессу временного интервала осуществляется средствами библиотеки. В подобных системах блокирование одной нити приводит к блокированию всего процесса, ибо ядро операционной системы не имеет представления о существовании нитей. По сути дела, в таких вычислительных системах просто имитируется наличие нитей исполнения.

Далее в этой части курса для простоты изложения мы будем использовать термин "процесс", хотя все сказанное будет относиться и к нитям исполнения.

19) Interleaving, race condition и взаимоисключения. нужно сделать короче

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. Возможные варианты чередования:

а b c d e f

a b d c e f

a b d e c f

a b d e f c

a d b c e f

......

d e f a b c

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 псевдо параллельного 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 псевдо параллельного 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 псевдо параллельном 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.

Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) - суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) - суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x=u+v

y=x\*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

* пересечение W(P) и W(Q) пусто,
* пересечение W(P) с R(Q) пусто,
* пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

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.

20) Критическая секция

**Критическая секция** – это часть программы, выполнение 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.

так, 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.

21) Программные алгоритмы организации взаимодействия процессов

* Требования, предъявляемые к алгоритмам

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)** .

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции.

* Запрет прерываний, Переменная-замок

**Запрет прерываний**

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.

#### **Переменная-замок**

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 процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

* Строгое чередование, Флаги готовности

#### **Строгое чередование**

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**.

#### **Флаги готовности**

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 процессу 1, 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).

22) Программные алгоритмы организации взаимодействия процессов

* Алгоритм Петерсона, Алгоритм булочной (Bakery algorithm)

#### **Алгоритм Петерсона**

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.

3)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.

4)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.

5)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.

#### **Алгоритм булочной (Bakery algorithm)**

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 неатомарности 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

}

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

* Аппаратная поддержка взаимоисключений

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

}

23) Семафоры

**Семафоры**- это объекты синхронизации , которые могут находиться в двух состояниях- открыт и закрыт

* Бинарные семафоры только открывают или закрывают вход в критическую секцию. По своему поведению похожи на mutex
* Считающие семафоры позволяют определить , сколько процессов могут одновременно войти в свою критическую секцию

-Если счетчик =0, то вход в критическую секцию невозможен

-Если счетчик > 0, то процесс может войти в свою критическую секцию

* В отличие от mutex ,семафоры реализуют активное ожидание- процесс на семафоре не блокируется, а продолжает проверять, не открылся ли семафор
* 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 из S 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.

* 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 С-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;

}

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что семафоры использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и взаимосинхронизации скорости работы процессов.

24) Мониторы

Решение задачи производитель-потребитель (producer-consumer) с помощью семафоров требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера.

Допустим, что в рассмотренном примере мы случайно поменяли местами операции (P), сначала выполнив для семафора mutex, а уже затем для семафоров full и empty. Допустим, что потребитель, войдя в свой критический участок (mutex сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию. (deadlock)

В сложных программах мы не можем с точностью провести анализ на бумаге. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от interleaving(чередования)(race condition) атомарных операций, и ошибки могут быть трудно воспроизводимы. Для того чтобы облегчить работу программистов, (в 1974 году Хором (Hoare)) был предложен механизм, получивший название мониторов. Рассмотрим конструкцию.

**Мониторы** представляют собой тип данных, который может быть 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;

}

}

Есть какой-то монитор и у него есть какие-то методы

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

Также существует блок реализации который запускается единожды в момент создания монитора или первым выполнением его метода.

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, то использование мониторов существенно упрощается, а вероятность ошибок становится меньше.

(Но одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов.)

Но мы не можем рассчитывать только на взаимоисключения

Нужны еще и средства организации очередности процессов, как семафоры 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. Тогда предложили механизм: разбудивший процесс покидает монитор немедленно после исполнения операции signal.

(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)) **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**.

----------------------------------------------------------------------------------------------------------

(Давайте применим концепцию мониторов к решению задачи производитель-потребитель.

monitor ProducerConsumer {

condition full, empty;

int count;

void put() {

if(count == N) full.wait;

put\_item;

count += 1;

if(count == 1) empty.signal;

}

void get() {

if (count == 0) empty.wait;

get\_item();

count -= 1;

if(count == N-1) full.signal;

}

{

count = 0;

}

}

Producer:

while(1) {

produce\_item;

ProducerConsumer.put();

}

Consumer:

while(1) {

ProducerConsumer.get();

consume\_item;

}

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация мониторов требует разработки специальных языков программирования и компиляторов для них. Мониторы встречаются в таких языках, как параллельный Евклид, параллельный Паскаль, Java и т. д. Эмуляция мониторов с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция семафоров. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.)

25) Сообщения

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае прямой адресации мы будем обозначать их так:

| send(P, message) – послать сообщение message процессу P ; |
| --- |
| receive(Q, message) – получить сообщение message от процесса Q. |

В случае непрямой адресации мы будем обозначать их так:

| send(A, message) – послать сообщение message в почтовый ящик A ; |
| --- |
| receive(A, message) – получить сообщение message из почтового ящика A. |

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

26) Эквивалентность семафоров, мониторов и сообщений

* Реализация мониторов и передачи сообщений с помощью семафоров

Рассмотрим сначала, как реализовать мониторы с помощью семафоров. Для этого нам нужно уметь реализовывать взаимоисключения при входе в монитор и условные переменные. Возьмем семафор mutex с начальным значением 1 для реализации взаимоисключения при входе в монитор и по одному семафору ci для каждой условной переменной. Кроме того, для каждой условной переменной заведем счетчик fi для индикации наличия ожидающих процессов. Когда процесс входит в монитор, компилятор будет генерировать вызов функции monitor\_enter, которая выполняет операцию P над семафором mutex для данного монитора. При нормальном выходе из монитора (то есть при выходе без вызова операции signal для условной переменной ) компилятор будет генерировать вызов функции monitor\_exit, которая выполняет операцию V над этим семафором.

Для выполнения операции wait над условной переменной компилятор будет генерировать вызов функции wait, которая выполняет операцию V для семафора mutex, разрешая другим процессам входить в монитор, и выполняет операцию P над соответствующим семафором ci, блокируя вызвавший процесс. Для выполнения операции signal над условной переменной компилятор будет генерировать вызов функции signal\_exit, которая выполняет операцию V над ассоциированным семафором ci (если есть процессы, ожидающие соответствующего события), и выход из монитора, минуя функцию monitor\_exit.

Semaphore mutex = 1;

void monitor\_enter(){

P(mutex);

}

void monitor\_exit(){

V(mutex);

}

Semaphore ci = 0;

int fi = 0;

void wait(i){

fi=fi + 1;

V(mutex);

P(ci);

fi=fi - 1;

}

void signal\_exit(i){

if (fi)V(ci);

else V(mutex);

}

Заметим, что при выполнении функции signal\_exit, если кто-либо ожидал этого события, процесс покидает монитор без увеличения значения семафора mutex, не разрешая тем самым всем процессам, кроме разбуженного, войти в монитор. Это увеличение совершит разбуженный процесс, когда покинет монитор обычным способом или когда выполнит новую операцию wait над какой-либо условной переменной.

Рассмотрим теперь, как реализовать передачу сообщений, используя семафоры. Для простоты опишем реализацию только одной очереди сообщений. Выделим в разделяемой памяти достаточно большую область под хранение сообщений, там же будем записывать, сколько пустых и заполненных ячеек находится в буфере, хранить ссылки на списки процессов, ожидающих чтения и памяти. Взаимоисключение при работе с разделяемой памятью будем обеспечивать семафором mutex. Также заведем по одному семафору ci на взаимодействующий процесс, для того чтобы обеспечивать блокирование процесса при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, желающего получить сообщение.

Процесс-получатель с номером i прежде всего выполняет операцию P(mutex), получая в монопольное владение разделяемую память. После чего он проверяет, есть ли в буфере сообщения. Если нет, то он заносит себя в список процессов, ожидающих сообщения, выполняет V(mutex) и P(ci). Если сообщение в буфере есть, то он читает его, изменяет счетчики буфера и проверяет, есть ли процессы в списке процессов, жаждущих записи. Если таких процессов нет, то выполняется V(mutex), и процесс-получатель выходит из критической области. Если такой процесс есть (с номером j ), то он удаляется из этого списка, выполняется V для его семафора cj, и мы выходим из критического района. Проснувшийся процесс начинает выполняться в критическом районе, так как mutex у нас имеет значение 0 и никто более не может попасть в критический район. При выходе из критического района именно разбуженный процесс произведет вызов V(mutex).

Как строится работа процесса-отправителя с номером i? Процесс, посылающий сообщение, тоже ждет, пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию P(mutex). Далее он проверяет, есть ли место в буфере, и если да, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, ожидающие сообщения. Если нет, выполняет V(mutex) и выходит из критической области, если есть, "будит" один из них (с номером j ), вызывая V(cj), с одновременным удалением этого процесса из списка процессов, ожидающих сообщений, и выходит из критического региона без вызова V(mutex), предоставляя тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель вносит себя в очередь процессов, ожидающих возможности записи, и вызывает V(mutex) и P(ci).

* Реализация семафоров и передачи сообщений с помощью мониторов

Нам достаточно показать, что с помощью мониторов можно реализовать семафоры, так как получать из семафоров сообщения мы уже умеем.

Самый простой способ такой реализации выглядит следующим образом. Заведем внутри монитора переменную-счетчик, связанный с эмулируемым семафором список блокируемых процессов и по одной условной переменной на каждый процесс. При выполнении операции P над семафором вызывающий процесс проверяет значение счетчика. Если оно больше нуля, уменьшает его на 1 и выходит из монитора. Если оно равно 0, процесс добавляет себя в очередь процессов, ожидающих события, и выполняет операцию wait над своей условной переменной. При выполнении операции V над семафором процесс увеличивает значение счетчика, проверяет, есть ли процессы, ожидающие этого события, и если есть, удаляет один из них из списка и выполняет операцию signal для условной переменной, соответствующей процессу.

* Реализация семафоров и мониторов с помощью очередей сообщений

Покажем, наконец, как реализовать семафоры с помощью очередей сообщений. Для этого воспользуемся более хитрой конструкцией, введя новый синхронизирующий процесс. Этот процесс имеет счётчик и очередь для процессов, ожидающих включения семафора. Для того чтобы выполнить операции P и V, процессы посылают синхронизирующему процессу сообщения, в которых указывают свои потребности, после чего ожидают получения подтверждения от синхронизирующего процесса.

После получения сообщения синхронизирующий процесс проверяет значение счетчика, чтобы выяснить, можно ли совершить требуемую операцию. Операция V всегда может быть выполнена, в то время как операция P может потребовать блокирования процесса. Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в очередь блокированных процессов, и подтверждение не посылается. Позднее, когда какой-либо из других процессов выполнит операцию V, один из блокированных процессов удаляется из очереди ожидания и получает соответствующее подтверждение.

Поскольку мы показали ранее, как из семафоров построить мониторы, мы доказали эквивалентность мониторов, семафоров и сообщений.

27) Условия возникновения тупиков

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.

**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. Например, в системах управления базами данных записи могут быть локализованы процессами, чтобы избежать состояния гонок . 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. Некоторые 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.

### **Условия возникновения тупиков**

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 preemption). 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.

28) Основные направления борьбы с тупиками

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 цикла ожидания ресурсов.

Итак, основные направления борьбы с тупиками:

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

29) Игнорирование проблемы тупиков

Простейший подход – не замечать проблему тупиков. Для того чтобы принять такое решение, необходимо оценить вероятность возникновения взаимоблокировки и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Проектировщики обычно не желают жертвовать производительностью системы или удобством пользователей для внедрения сложных и дорогостоящих средств борьбы с тупиками.

Любая ОС, имеющая в ядре ряд массивов фиксированной размерности, потенциально страдает от тупиков, даже если они не обнаружены. Таблица открытых файлов, таблица процессов, фактически каждая таблица являются ограниченными ресурсами. Заполнение всех записей таблицы процессов может привести к тому, что очередной запрос на создание процесса может быть отклонен. При неблагоприятном стечении обстоятельств несколько процессов могут выдать такой запрос одновременно и оказаться в тупике. Следует ли отказываться от вызова CreateProcess, чтобы решить эту проблему?

Подход большинства популярных ОС (Unix, Windows и др.) состоит в том, чтобы игнорировать данную проблему в предположении, что маловероятный случайный тупик предпочтительнее, чем нелепые правила, заставляющие пользователей ограничивать число процессов, открытых файлов и т. п. Сталкиваясь с нежелательным выбором между строгостью и удобством, трудно найти решение, которое устраивало бы всех.

30) Способы предотвращения тупиков

Цель предотвращения тупиков – обеспечить условия, исключающие возможность возникновения тупиковых ситуаций. Большинство методов связано с предотвращением одного из условий возникновения взаимоблокировки.

Система, предоставляя ресурс в распоряжение процесса, должна принять решение, безопасно это или нет. Возникает вопрос: есть ли такой алгоритм, который помогает всегда избегать тупиков и делать правильный выбор. Ответ – да, мы можем избегать тупиков, но только если определенная информация известна заранее.

* Способы предотвращения тупиков путем тщательного распределения ресурсов. Алгоритм банкира

Можно избежать взаимоблокировки, если распределять ресурсы, придерживаясь определенных правил. Среди такого рода алгоритмов наиболее известен **алгоритм банкира**, предложенный Дейкстрой, который базируется на так называемых **безопасных** или **надежных** состояниях (safe state). **Безопасное состояние** – это такое состояние, для которого имеется по крайней мере одна последовательность событий, которая не приведет к взаимоблокировке . Модель алгоритма основана на действиях банкира, который, имея в наличии капитал, выдает кредиты.

Суть алгоритма состоит в следующем.

* Предположим, что у системы в наличии n устройств, например лент.
* ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает n.
* Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени.
* Текущее состояние системы называется **надежным**, если ОС может обеспечить всем процессам их выполнение в течение конечного времени.
* В соответствии с алгоритмом банкира выделение устройств возможно, только если состояние системы остается надежным.

Термин **ненадежное** состояние не предполагает, что обязательно возникнут тупики. Он лишь говорит о том, что в случае неблагоприятной последовательности событий система может зайти в тупик.

Данный алгоритм обладает тем достоинством, что при его использовании нет необходимости в перераспределении ресурсов и откате процессов назад. Однако использование этого метода требует выполнения ряда условий.

* Число пользователей и число ресурсов фиксировано.
* Число работающих пользователей должно оставаться постоянным.
* Алгоритм требует, чтобы клиенты гарантированно возвращали ресурсы.
* Должны быть заранее указаны максимальные требования процессов к ресурсам. Чаще всего данная информация отсутствует.

Наличие таких жестких и зачастую неприемлемых требований может склонить разработчиков к выбору других решений проблемы взаимоблокировки.

* Предотвращение тупиков за счет нарушения условий возникновения тупиков

В отсутствие информации о будущих запросах единственный способ избежать взаимоблокировки – добиться невыполнения хотя бы одного из условий раздела "Условия возникновения тупиков ".

* Нарушение условия взаимоисключения, Нарушение условия ожидания дополнительных ресурсов

##### **Нарушение условия взаимоисключения**

В общем случае избежать взаимоисключений невозможно. Доступ к некоторым ресурсам должен быть исключительным. Тем не менее некоторые устройства удается обобществить. В качестве примера рассмотрим принтер. Известно, что пытаться осуществлять вывод на принтер могут несколько процессов. Во избежание хаоса организуют промежуточное формирование всех выходных данных процесса на диске, то есть разделяемом устройстве. Лишь один системный процесс, называемый сервисом или демоном принтера, отвечающий за вывод документов на печать по мере освобождения принтера, реально с ним взаимодействует. Эта схема называется спулингом (spooling). Таким образом, принтер становится разделяемым устройством, и тупик для него устранен.

К сожалению, не для всех устройств и не для всех данных можно организовать спулинг. Неприятным побочным следствием такой модели может быть потенциальная тупиковая ситуация из-за конкуренции за дисковое пространство для буфера спулинга. Тем не менее в той или иной форме эта идея применяется часто.

##### **Нарушение условия ожидания дополнительных ресурсов**

Условия ожидания ресурсов можно избежать, потребовав выполнения стратегии двухфазного захвата.

* В первой фазе процесс должен запрашивать все необходимые ему ресурсы сразу. До тех пор пока они не предоставлены, процесс не может продолжать выполнение.
* Если в первой фазе некоторые ресурсы, которые были нужны данному процессу, уже заняты другими процессами, он освобождает все ресурсы, которые были ему выделены, и пытается повторить первую фазу.

В известном смысле этот подход напоминает требование захвата всех ресурсов заранее. Естественно, что только специально организованные программы могут быть приостановлены в течение первой фазы и рестартованы впоследствии.

Таким образом, один из способов – заставить все процессы затребовать нужные им ресурсы перед выполнением ("все или ничего"). Если система в состоянии выделить процессу все необходимое, он может работать до завершения. Если хотя бы один из ресурсов занят, процесс будет ждать.

Другим примером может служить механизм двухфазной локализации записей в СУБД. Однако в целом подобный подход приводит к неэффективному использованию компьютера. Как уже отмечалось, перечень будущих запросов к ресурсам редко удается спрогнозировать. Если такая информация есть, то можно воспользоваться алгоритмом банкира. Заметим также, что описываемый подход противоречит парадигме модульности в программировании, поскольку приложение должно знать о предполагаемых запросах к ресурсам во всех модулях.

* Нарушение принципа отсутствия перераспределения. Нарушение условия кругового ожидания

##### **Нарушение принципа отсутствия перераспределения**

Если бы можно было отбирать ресурсы у удерживающих их процессов до завершения этих процессов, то удалось бы добиться невыполнения третьего условия возникновения тупиков. Перечислим минусы данного подхода.

Во-первых, отбирать у процессов можно только те ресурсы, состояние которых легко сохранить, а позже восстановить, например состояние процессора. Во-вторых, если процесс в течение некоторого времени использует определенные ресурсы, а затем освобождает эти ресурсы, он может потерять результаты работы, проделанной до настоящего момента. Наконец, следствием данной схемы может быть дискриминация отдельных процессов, у которых постоянно отбирают ресурсы.

Весь вопрос в цене подобного решения, которая может быть слишком высокой, если необходимость отбирать ресурсы возникает часто.

##### **Нарушение условия кругового ожидания**

Трудно предложить разумную стратегию, чтобы избежать последнего условия из раздела "Условия возникновения тупиков " – циклического ожидания.

Способы:

***– упорядочить ресурсы.*** Например, можно присвоить всем ресурсам уникальные номера и потребовать, чтобы процессы запрашивали ресурсы в порядке их возрастания. Тогда круговое ожидание возникнуть не может. После последнего запроса и освобождения всех ресурсов можно разрешить процессу опять осуществить первый запрос. Очевидно, что практически невозможно найти порядок, который удовлетворит всех.

Пример упорядочивания ресурсов – создание иерархии спин-блокировок в Windows 2000. Спин-блокировка – простейший способ синхронизации (вопросы синхронизации процессов рассмотрены в соответствующей лекции). Спин-блокировка может быть захвачена и освобождена процессом. Классическая тупиковая ситуация возникает, когда процесс P1 захватывает спин-блокировку S1 и претендует на спин-блокировку S2, а процесс P2, захватывает спин-блокировку S2 и хочет дополнительно захватить спин-блокировку S1. Чтобы этого избежать, все спин-блокировки помещаются в упорядоченный список. Захват может осуществляться только в порядке, указанном в списке.

– ***действовать в соответствии с правилом, согласно которому каждый процесс может иметь только один ресурс в каждый момент времени***. Если нужен второй ресурс – освободи первый. Очевидно, что для многих процессов это неприемлемо.

Таким образом, технология предотвращения циклического ожидания, как правило, неэффективна и может без необходимости закрывать доступ к ресурсам.

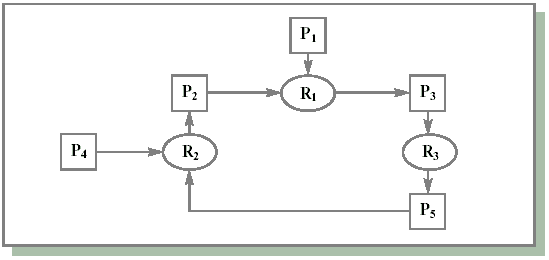
31) Обнаружение тупиков

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.

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.

Существуют и другие способы обнаружения тупиков, применимые также в ситуациях, когда имеется несколько ресурсов каждого типа(редукция графа распределения ресурсов, матричный алгоритм)

32) Восстановление после тупиков

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.

33) Физическая организация памяти компьютера

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.

Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.

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.

* Локальность

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как **принцип локальности** или локализации обращений.

(Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. )В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования иерархической схемы. (Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при переработке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).)

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим адресным пространством

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в основной и внешней памяти компьютера. В некоторых схемах потоки между оперативной и внешней памятью регулируются программистом (см. например, далее оверлейные структуры ), однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

.

34) Логическая память

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 пространстве дублирующую 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.

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.

35) Связывание адресов

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**

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:

* 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.
* 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

36) Функции системы управления памятью

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

* отображение адресного пространства процесса на конкретные области физической памяти;
* распределение памяти между конкурирующими процессами;
* контроль доступа к адресным пространствам процессов;
* выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
* учет свободной и занятой памяти.

37) Схемы управления памятью

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.

38) 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.

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

Недостатки:

– число одновременно выполняемых процессов ограничено числом разделов.

– потеря части памяти, выделенной процессу, но не используемой им(фрагментация). 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.

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. :

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 . Смежные свободные участки могут быть объединены.

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!

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.

39) Страничная память

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 кадр содержит 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.

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.

40) Сегментная и сегментно-страничная организация памяти

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...)**.

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 программа обращается 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.

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 сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

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.

41) Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление виртуальной памяти (virtual memory). Впервые она была реализована в 1959 г. на компьютере "Атлас", разработанном в Манчестерском университете.

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 сегментно-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.

Сегментная организация в чистом виде встречается редко.

42) Архитектурные средства поддержки виртуальной памяти

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 пространство существенно 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.

43) 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 , 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.

44) 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 не перекрывающиеся 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.

45) Структура таблицы страниц

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 .

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.

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.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры.

46) Ассоциативная память

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.

.

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.

47) Инвертированная таблица страниц

Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляют собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение **инвертированной таблицы страниц**

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной таблицы для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов.

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

48) Размер страницы

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и следовало ожидать, нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы – это степень двойки от 29 до 214 байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

49) Исключительные ситуации при работе с памятью

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.

50) Стратегии управления страничной памятью

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.

51) Алгоритмы замещения страниц

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 бит не 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 приложение может 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.

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 ). использование LRU алгоритма позволяет сократить количество страничных нарушений.

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://intuit.ru/studies/professional_retraining/941/courses/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.

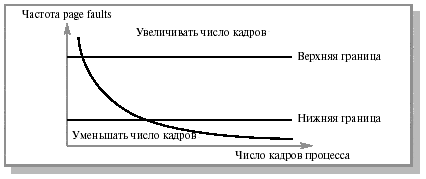
Например, алгоритм Second-Chance - модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, 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.

52) Управление количеством страниц, выделенных процессу. Модель рабочего множества

* 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://intuit.ru/studies/professional_retraining/941/courses/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. Необходимо 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

Существует набор страниц , активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода производительно работать, избегая большого количества page faults. Этот набор страниц называется **рабочим множеством** процесса.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Например, 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. 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.

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.

53) Страничные демоны

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://intuit.ru/studies/professional_retraining/941/courses/31/literature#literature.1)]). Данный демон производит 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, потребовавший 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.

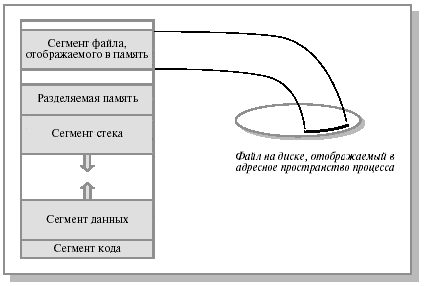
54) Программная поддержка сегментной модели памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти, - ведение таблиц страниц, трансляция адреса, обработка страничных ошибок, управление ассоциативной памятью и др. - тесно связана со структурами данных, обеспечивающими удобное представление адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной ОС.

Чаще всего виртуальная память процесса ОС разбивается на сегменты пяти типов: кода программы, данных, стека, разделяемый и сегмент файлов, отображаемых в память (см. [рис. 10.5](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/986?page=4#image.10.5)).

**Сегмент программного кода** содержит только команды. Сегмент программного кода не модифицируется в ходе выполнения процесса, обычно страницы данного сегмента имеют атрибут read-only. Следствием этого является возможность использования одного экземпляра кода для разных процессов.

**Сегмент данных**, содержащий переменные программы и сегмент стека, содержащий автоматические переменные, могут динамически менять свой размер (обычно данные в сторону увеличения адресов, а стек - в сторону уменьшения) и содержимое, должны быть доступны по чтению и записи и являются приватными сегментами процесса.



**Рис. 10.5.** Образ процесса в памяти

С целью обобществления памяти между несколькими процессами создаются **разделяемые сегменты**, допускающие доступ по чтению и записи. Вариантом разделяемого сегмента может быть **сегмент файла, отображаемого в память**. Специфика таких сегментов состоит в том, что из них откачка осуществляется не в системную область выгрузки, а непосредственно в отображаемый файл. Реализация разделяемых сегментов основана на том, что логические страницы различных процессов связываются с одними и теми же страничными кадрами.

Сегменты представляют собой непрерывные области (в Linux они так и называются - области) в виртуальном адресном пространстве процесса, выровненные по границам страниц. Каждая область состоит из набора страниц с одним и тем же режимом защиты. Между областями в виртуальном пространстве могут быть свободные участки. Естественно, что подобные объекты описаны соответствующими структурами (см., например, структуры mm\_struct и vm\_area\_struct в Linux).

Часть работы по организации сегментов может происходить с участием программиста. Особенно это заметно при низкоуровневом программировании. В частности, отдельные области памяти могут быть поименованы и использоваться для обмена данными между процессами. Два процесса могут общаться через разделяемую область памяти при условии, что им известно ее имя (пароль). Обычно это делается при помощи специальных вызовов (например, map и unmap), входящих в состав интерфейса виртуальной памяти.

Загрузка исполняемого файла (системный вызов exec) осуществляется обычно через отображение (mapping) его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. Например, сегмент кода является сегментом отображаемого в память файла, содержащего исполняемую программу. При попытке выполнить первую же инструкцию система обнаруживает, что нужной части кода в памяти нет, генерирует page fault и подкачивает эту часть кода с диска. Далее процедура повторяется до тех пор, пока вся программа не окажется в оперативной памяти.

Как уже говорилось, размер сегмента данных динамически меняется. Рассмотрим, как организована поддержка сегментов данных в Unix. Пользователь, запрашивая (библиотечные вызовы malloc, new ) или освобождая ( free, delete ) память для динамических данных, фактически изменяет границу выделенной процессу памяти через системный вызов brk (от слова break), который модифицирует значение переменной brk из структуры данных процесса. В результате происходит выделение физической памяти, граница brk смещается в сторону увеличения виртуальных адресов, а соответствующие строки таблиц страниц получают осмысленные значения. При помощи того же вызова brk пользователь может уменьшить размер сегмента данных. На практике освобожденная пользователем виртуальная память (библиотечные вызовы free, delete ) системе не возвращается. На это есть две причины. Во-первых, для уменьшения размеров сегмента данных необходимо организовать его уплотнение или "сборку мусора". А во-вторых, незанятые внутри сегмента данных области естественным образом будут вытолкнуты из оперативной памяти вследствие того, что к ним не будет обращений. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек.

Более подробно информация об адресных пространствах процессов в Unix изложена в [[Кузнецов](https://intuit.ru/studies/professional_retraining/941/courses/31/literature#literature.36)], [[Bach, 1986](https://intuit.ru/studies/professional_retraining/941/courses/31/literature#literature.1)].

* Отдельные аспекты функционирования менеджера памяти

Корректная работа менеджера памяти помимо принципиальных вопросов, связанных с выбором абстрактной модели виртуальной памяти и ее аппаратной поддержкой, обеспечивается также множеством нюансов и мелких деталей. В качестве примера такого рода компонента рассмотрим более подробно **локализацию страниц в памяти**, которая применяется в тех случаях, когда поддержка страничной системы приводит к необходимости разрешить определенным страницам, хранящим буферы ввода-вывода, другие важные данные и код, быть блокированными в памяти.

Рассмотрим случай, когда система виртуальной памяти может вступить в конфликт с подсистемой ввода-вывода. Например, процесс может запросить ввод в буфер и ожидать его завершения. Управление передастся другому процессу, который может вызвать page fault и, с отличной от нуля вероятностью, спровоцировать выгрузку той страницы, куда должен быть осуществлен ввод первым процессом. Подобные ситуации нуждаются в дополнительном контроле, особенно если ввод-вывод реализован с использованием механизма прямого доступа к памяти (DMA). Одно из решений данной проблемы - вводить данные в не вытесняемый буфер в пространстве ядра, а затем копировать их в пользовательское пространство.

Второе решение - локализовать страницы в памяти, используя специальный бит локализации, входящий в состав атрибутов страницы. Локализованная страница замещению не подлежит. Бит локализации сбрасывается после завершения операции ввода-вывода.

Другое использование бита локализации может иметь место и при нормальном замещении страниц. Рассмотрим следующую цепь событий. Низкоприоритетный процесс после длительного ожидания получил в свое распоряжение процессор и подкачал с диска нужную ему страницу. Если он сразу после этого будет вытеснен высоко приоритетным процессом, последний может легко заместить вновь подкачанную страницу низкоприоритетного, так как на нее не было ссылок. Имеет смысл вновь загруженные страницы помечать битом локализации до первой ссылки, иначе низкоприоритетный процесс так и не начнет работать.

Использование бита локализации может быть опасным, если забыть его отключить. Если такая ситуация имеет место, страница становится неиспользуемой. SunOS разрешает использование данного бита в качестве подсказки, которую можно игнорировать, когда пул свободных кадров становится слишком маленьким.

Другим важным применением локализации является ее использование в системах мягкого **реального времени**. Рассмотрим процесс или нить реального времени. Вообще говоря, виртуальная память - антитеза вычислений реального времени, так как дает непредсказуемые задержки при подкачке страниц. Поэтому системы реального времени почти не используют виртуальную память. ОС Solaris поддерживает как реальное время, так и разделение времени. Для решения проблемы page faults, Solaris разрешает процессам сообщать системе, какие страницы важны для процесса, и локализовать их в памяти. В результате возможно выполнение процесса, реализующего задачу реального времени, содержащего локализованные страницы, где временные задержки страничной системы будут минимизированы.

Помимо системы локализации страниц, есть и другие интересные проблемы, возникающие в процессе управления памятью. Так, например, бывает непросто осуществить повторное выполнение инструкции, вызвавшей page fault. Представляют интерес и алгоритмы отложенного выделения памяти (копирование при записи и др.). Ограниченный объем данного курса не позволяет рассмотреть их более подробно.

55) Общие сведения о файлах

* 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.

56) Организация файлов и доступ к ним

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.

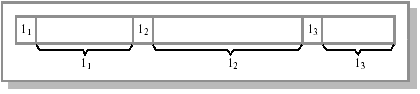
Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки 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.

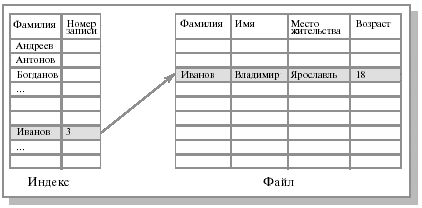
Другой способ представления файлов - **последовательность записей переменной длины**, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (см. [рис. 11.1](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/988?page=1#image.11.1)). Базисная операция в данном 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://intuit.ru/studies/professional_retraining/941/courses/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).

57) Операции над файлами

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://intuit.ru/studies/professional_retraining/941/courses/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 файла для 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 файла fd=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://intuit.ru/studies/professional_retraining/941/courses/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).

58) Директории. Логическая структура файлового архива

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 .

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.

Таким образом, файлы на диске образуют иерархическую древовидную структуру

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на [рис. 11.4](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/988?page=2#image.11.4), наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или 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.

* Операции над директориями

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории - это файлы, логика работы с ними 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://intuit.ru/studies/professional_retraining/941/courses/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. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.
* Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.
* Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. 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.

59) Защита файлов

* 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

Наиболее 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 последствия.

* 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).

### 

60) Общая структура файловой системы

Система хранения данных на дисках может быть структурирована следующим образом

**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**. 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.

* Управление внешней памятью

Прежде чем описывать структуру данных файловой системы на диске, необходимо рассмотреть алгоритмы выделения дискового пространства и способы учета свободной и занятой дисковой памяти. Эти задачи связаны между собой.

#### **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

61) Методы выделения дискового пространства

* 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.

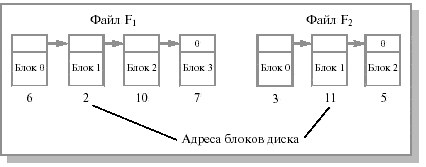
Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать, особенно в тех случаях, когда размер файла меняется. Если места не хватило, то пользовательская 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://intuit.ru/studies/professional_retraining/941/courses/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://intuit.ru/studies/professional_retraining/941/courses/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.

* 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://intuit.ru/studies/professional_retraining/941/courses/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.

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.

62) Управление свободным и занятым дисковым пространством

* Учет при помощи организации битового вектора

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

Главное преимущество этого подхода 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. Например, 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.

63) 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.

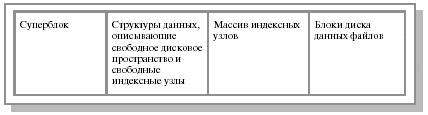
Проведенные исследования показали, что большинство файлов имеют небольшой размер. Например, в Unix приблизительно 85% файлов имеют размер менее 8 Кбайт и 48% - менее 1 Кбайта.

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.

64) 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, может состоять из четырех основных частей (см. [рис. 12.5](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=3#image.12.5)).

Рассмотрим структуру служебных типовых данных в дисковой файловой системе. Например Unix, на одном из разделов диска, может состоять из четырех основных частей



**Рис. 12.5.** Примерная структура файловой системы на диске

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

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

2)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 суперблок непрерывно подвергался модификации, что снижало надежность файловой системы в целом. Выделение структур данных, описывающих дисковое пространство, в отдельную часть является более правильным решением.

3)Массив индексных узлов (i-list) содержит список индексов, соответствующих файлам данной системы. Размер массива индексных узлов определяется администратором при установке системы. От этого индекса зависит максимальное число файлов, которые могут быть созданы в файловой системе.

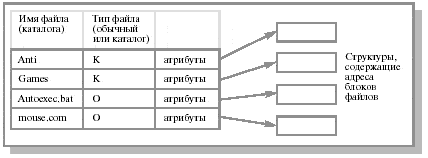
4)В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов.

Запоминание этой информации является модификацией массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

65) Реализация директорий

Как уже говорилось, директория или каталог - это файл, имеющий вид таблицы и хранящий список входящих в него файлов или каталогов. Основная задача файлов-директорий - поддержка иерархической древовидной структуры файловой системы. Запись в директории имеет определенный для данной ОС формат, зачастую неизвестный пользователю, поэтому блоки данных файла-директории заполняются не через операции записи, а при помощи специальных системных вызовов (например, создание файла).

Для доступа к файлу ОС использует путь (pathname), сообщенный пользователем. Запись в директории связывает имя файла или имя поддиректории с блоками данных на диске (см. [рис. 12.6](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=4#image.12.6)). В зависимости от способа выделения файлу блоков диска (см. раздел "Методы выделения дискового пространства") эта ссылка может быть номером первого блока или номером индексного узла. В любом случае обеспечивается связь символьного имени файла с данными на диске.



**Рис. 12.6.** Реализация директорий

Когда система открывает файл, она ищет его имя в директории. Затем из записи в директории или из структуры, на которую запись в директории указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в главной памяти. Все последующие ссылки на данный файл используют эту информацию. Атрибуты файла можно хранить непосредственно в записи в директории, как показано на [рис. 12.6](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=4#image.12.6). Однако для организации совместного доступа к файлам удобнее хранить атрибуты в индексном узле, как это делается в Unix.

* Примеры реализации директорий в некоторых ОС

##### **Директории в ОС MS-DOS**

В ОС MS-DOS типовая запись в директории имеет вид, показанный на [рис. 12.7](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=4#image.12.7).



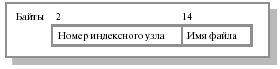
**Рис. 12.7.** Вариант записи в директории MS-DOS

В ОС MS-DOS, как и в большинстве современных ОС, директории могут содержать поддиректории (специфицируемые битом атрибута), что позволяет конструировать произвольное дерево директорий файловой системы.

Номер первого блока используется в качестве индекса в таблице FAT. Далее по цепочке в этой таблице могут быть найдены остальные блоки.

##### **Директории в ОС Unix**

Структура директории проста. Каждая запись содержит имя файла и номер его индексного узла (см. [рис. 12.8](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=4#image.12.8)). Вся остальная информация о файле (тип, размер, время модификации, владелец и т. д. и номера дисковых блоков) находится в индексном узле.



**Рис. 12.8.** Вариант записи в директории Unix

В более поздних версиях Unix форма записи претерпела ряд изменений, например имя файла описывается структурой. Однако суть осталась прежней.

66) Поиск в директории

* Линейный поиск

Существует несколько стратегий просмотра списка символьных имен. Простейшей из них является линейный поиск. Директория просматривается с самого начала, пока не встретится нужное имя файла. Хотя это наименее эффективный способ поиска, оказывается, что в большинстве случаев он работает с приемлемой производительностью. Например, авторы Unix утверждали, что линейного поиска вполне достаточно. По-видимому, это связано с тем, что на фоне относительно медленного доступа к диску некоторые задержки, возникающие в процессе сканирования списка, несущественны.

Метод прост, но требует временных затрат. Для создания нового файла вначале нужно проверить директорию на наличие такого же имени. Затем имя нового файла вставляется в конец директории (если, разумеется, файл с таким же именем в директории не существует, в противном случае нужно информировать пользователя). Для удаления файла нужно также выполнить поиск его имени в списке и пометить запись как неиспользуемую.

Реальный недостаток данного метода - последовательный поиск файла. Информация о структуре директории используется часто, и неэффективный способ поиска будет заметен пользователям. Можно свести поиск к бинарному, если отсортировать список файлов. Однако это усложнит создание и удаление файлов, так как требуется перемещение большого объема информации.

* Хеш-таблица

Хеширование (см. например, [[Ахо, 2001](https://intuit.ru/studies/professional_retraining/941/courses/31/literature#literature.10)]) - другой способ, который может использоваться для размещения и последующего поиска имени файла в директории. В данном методе имена файлов также хранятся в каталоге в виде линейного списка, но дополнительно используется хеш-таблица. Хеш-таблица, точнее построенная на ее основе хеш-функция, позволяет по имени файла получить указатель на имя файла в списке. Таким образом, можно существенно уменьшить время поиска.

В результате хеширования могут возникать коллизии, то есть ситуации, когда функция хеширования, примененная к разным именам файлов, дает один и тот же результат. Обычно имена таких файлов объединяют в связные списки, предполагая в дальнейшем осуществление в них последовательного поиска нужного имени файла. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число коллизий. Однако всегда есть вероятность неблагоприятного исхода, когда непропорционально большому числу имен файлов функция хеширования ставит в соответствие один и тот же результат. В таком случае преимущество использования этой схемы по сравнению с последовательным поиском практически утрачивается.

* Другие методы поиска

Помимо описанных методов поиска имени файла, в директории существуют и другие. В качестве примера можно привести организацию поиска в каталогах файловой системы NTFS при помощи так называемого B-дерева, которое стало стандартным способом организации индексов в системах баз данных (см. [[Ахо, 2001](https://intuit.ru/studies/professional_retraining/941/courses/31/literature#literature.10)]).

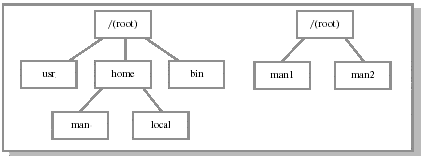
67)Монтирование файловых систем, Связывание файлов

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

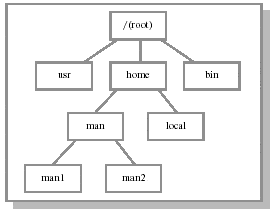
Функция mount (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем(дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков).

Функция umount (демонтировать) выключает файловую систему из иерархии.

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://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=5#image.12.9) и [рис. 12.10](https://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=5#image.12.10)).



**Рис. 12.9.** Две файловые системы до монтирования



**Рис. 12.10.** Общая файловая система после монтирования

Например, 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 ),

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 супер блоком, 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.

Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информация о 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.

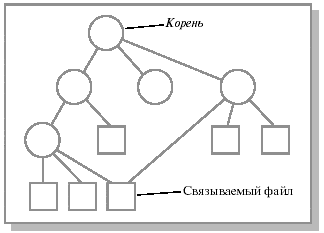
### **Связывание файлов**

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://intuit.ru/studies/professional_retraining/941/courses/31/lecture/990?page=5#image.12.11)). Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл). Например, выполняемый файл традиционного 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 **символической** линковкой (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.

68) Кооперация процессов при работе с файлами, Примеры разрешения коллизий и тупиковых ситуаций

### **Кооперация процессов при работе с файлами**

Когда различные пользователи работают вместе над проектом, они часто нуждаются в разделении файлов.

Разделяемый файл - разделяемый ресурс. Как и в случае любого совместно используемого ресурса, процессы должны синхронизировать доступ к совместно используемым файлам, каталогам, чтобы избежать тупиковых ситуаций, дискриминации отдельных процессов и снижения производительности системы.

Например, если несколько пользователей одновременно редактируют какой-либо файл и не принято специальных мер, то результат будет непредсказуем и зависит от того, в каком порядке осуществлялись записи в файл. Между двумя операциями read одного процесса другой процесс может модифицировать данные, что для многих приложений неприемлемо. Простейшее решение данной проблемы - предоставить возможность одному из процессов захватить файл, то есть блокировать доступ к разделяемому файлу других процессов на все время, пока файл остается открытым для данного процесса. Однако это было бы недостаточно гибко и не соответствовало бы характеру поставленной задачи.

Рассмотрим вначале **грубый** подход, то есть временный захват пользовательским процессом файла или записи (части файла между указанными позициями).

Системный вызов, позволяющий установить и проверить блокировки на файл, является неотъемлемым атрибутом современных многопользовательских ОС. В принципе, было бы логично связать синхронизацию доступа к файлу как к единому целому с системным вызовом open (т. е., например, открытие файла в режиме записи или обновления могло бы означать его монопольную блокировку соответствующим процессом, а открытие в режиме чтения - совместную блокировку). Так поступают во многих операционных системах (начиная с ОС Multics). В ОС Unix это не так, что имеет исторические причины.

В первой версии системы Unix, разработанной Томпсоном и Ричи, механизм захвата файла отсутствовал. Применялся очень простой подход к обеспечению параллельного (от нескольких процессов) доступа к файлам: система позволяла любому числу процессов одновременно открывать один и тот же файл в любом режиме (чтения, записи или обновления) и не предпринимала никаких синхронизационных действий. Вся ответственность за корректность совместной обработки файла ложилась на использующие его процессы, и система даже не предоставляла каких-либо особых средств для синхронизации доступа процессов к файлу. Однако впоследствии для того, чтобы повысить привлекательность системы для коммерческих пользователей, работающих с базами данных, в версию V системы были включены механизмы захвата файла и записи, базирующиеся на системном вызове fcntl.

Допускается два варианта синхронизации: с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено, и без ожидания, когда процесс немедленно оповещается об удовлетворении требования блокировки или о невозможности ее удовлетворения в данный момент.

Установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами-потомками этого процесса. Более того, даже если некоторый процесс пользуется синхронизационными возможностями системного вызова fcntl, другие процессы по-прежнему могут работать с тем файлом без всякой синхронизации. Другими словами, это дело группы процессов, совместно использующих файл, - договориться о способе синхронизации параллельного доступа.

Более тонкий подход заключается в прозрачной для пользователя блокировке отдельных структур ядра, отвечающих за работу с файлами части пользовательских данных. Например, в ОС Unix во время системного вызова, осуществляющего ту или иную операцию с файлом, как правило, происходит блокирование индексного узла, содержащего адреса блоков данных файла. Может показаться, что организация блокировок или запрета более чем одному процессу работать с файлом во время выполнения системного вызова является излишней, так как в подавляющем большинстве случаев выполнение системных вызовов и так не прерывается, то есть ядро работает в условиях невытесняющей многозадачности. Однако в данном случае это не совсем так. Операции чтения и записи занимают продолжительное время и лишь инициируются центральным процессором, а осуществляются по независимым каналам, поэтому установка блокировок на время системного вызова является необходимой гарантией атомарности операций чтения и записи. На практике оказывается достаточным заблокировать один из буферов кэша диска, в заголовке которого ведется список процессов, ожидающих освобождения данного буфера. Таким образом, в соответствии с семантикой Unix изменения, сделанные одним пользователем, немедленно становятся "видны" другому пользователю, который держит данный файл открытым одновременно с первым.

#### **Примеры разрешения коллизий и тупиковых ситуаций**

Логика работы системы в сложных ситуациях может проиллюстрировать особенности организации мультидоступа.

Рассмотрим в качестве примера **образование потенциального тупика** при создании связи ( link ), когда разрешен совместный доступ к файлу

Два процесса, выполняющие одновременно следующие функции:

процесс A: link("a/b/c/d","e/f/g");

процесс B: link("e/f","a/b/c/d/ee");

могут зайти в тупик. Предположим, что процесс A обнаружил индекс файла "a/b/c/d" в тот самый момент, когда процесс B обнаружил индекс файла "e/f". Фраза "в тот же самый момент" означает, что системой достигнуто состояние, при котором каждый процесс получил искомый индекс. Когда же теперь процесс A попытается получить индекс файла "e/f", он приостановит свое выполнение до тех пор, пока индекс файла "f" не освободится. В то же время процесс B пытается получить индекс каталога "a/b/c/d" и приостанавливается в ожидании освобождения индекса файла "d". Процесс A будет удерживать заблокированным индекс, нужный процессу B, а процесс B, в свою очередь, будет удерживать заблокированным индекс, необходимый процессу A.

Для предотвращения этого классического примера взаимной блокировки в файловой системе принято, чтобы ядро освобождало индекс исходного файла после увеличения значения счетчика связей. Тогда, поскольку первый из ресурсов (индекс) свободен при обращении к следующему ресурсу, взаимной блокировки не происходит.

Поводов для нежелательной **конкуренции** между процессами много, особенно **при удалении** имен каталогов. Предположим, что один процесс пытается найти данные файла по его полному символическому имени, последовательно проходя компонент за компонентом, а другой процесс удаляет каталог, имя которого входит в путь поиска. Допустим, процесс A делает разбор имени "a/b/c/d" и приостанавливается во время получения индексного узла для файла "c". Он может приостановиться при попытке заблокировать индексный узел или при попытке обратиться к дисковому блоку, где этот индексный узел хранится. Если процессу B нужно удалить связь для каталога с именем "c", он может приостановиться по той же самой причине, что и процесс A. Пусть ядро впоследствии решит возобновить процесс B раньше процесса A. Прежде чем процесс A продолжит свое выполнение, процесс B завершится, удалив связь каталога "c" и его содержимое по этой связи. Позднее процесс A попытается обратиться к несуществующему индексному узлу, который уже был удален. Алгоритм поиска файла, проверяющий в первую очередь неравенство значения счетчика связей нулю, должен сообщить об ошибке.

Можно привести и другие примеры, которые демонстрируют необходимость тщательного проектирования файловой системы для ее последующей надежной работы.

69) Надежность файловой системы, Целостность файловой системы

### **Надежность файловой системы**

Жизнь полна неприятных неожиданностей, а разрушение файловой системы зачастую более опасно, чем разрушение компьютера. Поэтому файловые системы должны разрабатываться с учетом подобной возможности. Помимо очевидных решений, например своевременное дублирование информации (backup), файловые системы современных ОС содержат специальные средства для поддержки собственной совместимости.

#### **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

70) Порядок выполнения операций, Журнализация

##### **Порядок выполнения операций**

Очевидно, что для правильного функционирования файловой системы значимость отдельных данных не равноценна. 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://intuit.ru/studies/professional_retraining/941/courses/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]

71)Проверка целостности файловой системы при помощи утилит Управление "плохими" блоками

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

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. Например, 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.

72) Производительность файловой системы

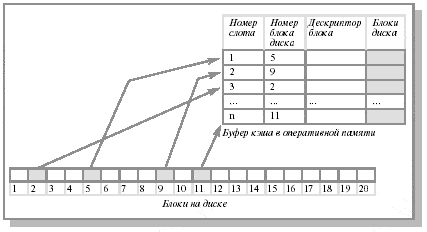
Поскольку обращение к диску - операция относительно медленная, минимизация количества таких обращений - ключевая задача всех алгоритмов, работающих с внешней памятью. Наиболее типичная техника повышения скорости работы с диском - кэширование.

* Кэширование

Кэш диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска (см. [рис. 12.12](https://intuit.ru/studies/professional_retraining/941/courses/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. Необходимо 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.

73) 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.

74) Реализация некоторых операций над файлами

* Системные вызовы, работающие с символическим именем файла
* Системные вызовы, связывающие pathname с дескриптором файла

Это функции создания и открытия файла. Например, в ОС Unix

fd = create(pathname,modes);

fd = open(pathname,flags,modes);

Другие операции над файлами, такие как чтение, запись, позиционирование головок чтения-записи, воспроизведение дескриптора файла, установка параметров ввода-вывода, определение статуса файла и закрытие файла, используют значение полученного дескриптора файла.

Рассмотрим работу системного вызова open.

Логическая файловая подсистема просматривает файловую систему в поисках файла по его имени. Она проверяет права на открытие файла и выделяет открываемому файлу запись в таблице файлов. Запись таблицы файлов содержит указатель на индексный узел открытого файла. Ядро выделяет запись в личной (закрытой) таблице в адресном пространстве процесса, выделенном процессу (таблица эта называется таблицей пользовательских дескрипторов открытых файлов) и запоминает указатель на данную запись. В роли указателя выступает дескриптор файла, возвращаемый пользователю. Запись в таблице пользовательских файлов указывает на запись в глобальной таблице файлов.

Первые три пользовательских дескриптора (0, 1 и 2) именуются дескрипторами файлов стандартного ввода, стандартного вывода и стандартного файла ошибок. Процессы в системе Unix по договоренности используют дескриптор файла стандартного ввода при чтении вводимой информации, дескриптор файла стандартного вывода при записи выводимой информации и дескриптор стандартного файла ошибок для записи сообщений об ошибках.

75) Реализация некоторых операций над файлами

* Связывание файла

Системная функция link связывает файл с новым именем в структуре каталогов файловой системы, создавая для существующего индекса новую запись в каталоге. Синтаксис вызова функции link:

link(source file name, target file name);

где source file name - существующее имя файла, а target file name - новое (дополнительное) имя, присваиваемое файлу после выполнения функции link.

Сначала ОС определяет местонахождение индекса исходного файла и увеличивает значение счетчика связей в индексном узле. Затем ядро ищет файл с новым именем; если он существует, функция link завершается неудачно, и ядро восстанавливает прежнее значение счетчика связей, измененное ранее. В противном случае ядро находит в родительском каталоге свободную запись для файла с новым именем, записывает в нее новое имя и номер индекса исходного файла.

* Удаление файла

В Unix системная функция unlink удаляет из каталога точку входа для файла. Синтаксис вызова функции unlink:

unlink(pathname);

Если удаляемое имя является последней связью файла с каким-либо каталогом, ядро в итоге освобождает все информационные блоки файла. Однако если у файла было несколько связей, он остается все еще доступным под другими именами.

Для того чтобы забрать дисковые блоки, ядро в цикле просматривает таблицу содержимого индексного узла, освобождая все блоки прямой адресации немедленно. Что касается блоков косвенной адресации, то ядро освобождает все блоки, появляющиеся на различных уровнях косвенности, рекурсивно, причем в первую очередь освобождаются блоки с меньшим уровнем.

* Системные вызовы, работающие с файловым дескриптором

Открытый файл может использоваться для чтения и записи последовательностей байтов. Для этого поддерживаются два системных вызова read и write, работающие с файловым дескриптором (или handle в терминологии Microsoft), полученным при ранее выполненных системных вызовах open или creat.

76) Функции ввода-вывода из файла

Системный вызов read выполняет чтение обычного файла

number = read(fd,buffer,count);

где fd - дескриптор файла, возвращаемый функцией open, buffer - адрес структуры данных в пользовательском процессе, где будут размещаться считанные данные в случае успешного завершения выполнения функции read, count - количество байтов, которые пользователю нужно прочитать, number - количество фактически прочитанных байтов.

Синтаксис вызова системной функции write (писать):

number = write(fd,buffer,count);

где переменные fd, buffer, count и number имеют тот же смысл, что и для вызова системной функции read. Алгоритм записи в обычный файл похож на алгоритм чтения из обычного файла. Однако если в файле отсутствует блок, соответствующий смещению в байтах до места, куда должна производиться запись, ядро выделяет блок и присваивает ему номер в соответствии с точным указанием места в таблице содержимого индексного узла.

Обычное использование системных функций read и write обеспечивает последовательный доступ к файлу, однако процессы могут использовать вызов системной функции lseek для указания места в файле, где будет производиться ввод-вывод, и осуществления произвольного доступа к файлу. Синтаксис вызова системной функции:

position = lseek(fd,offset,reference);

где fd - дескриптор файла, идентифицирующий файл, offset - смещение в байтах, а reference указывает, является ли значение offset смещением от начала файла, смещением от текущей позиции ввода-вывода или смещением от конца файла. Возвращаемое значение, position, является смещением в байтах до места, где будет начинаться следующая операция чтения или записи.

77) Современные архитектуры файловых систем

Современные ОС предоставляют пользователю возможность работать сразу с несколькими файловыми системами (Linux работает с Ext2fs, FAT и др.). Файловая система в традиционном понимании становится частью более общей многоуровневой структуры.

На верхнем уровне располагается так называемый диспетчер файловых систем (например, в Windows 95 этот компонент называется installable filesystem manager). Он связывает запросы прикладной программы с конкретной файловой системой.

Каждая файловая система (иногда говорят - драйвер файловой системы) на этапе инициализации регистрируется у диспетчера, сообщая ему точки входа, для последующих обращений к данной файловой системе.

Та же идея поддержки нескольких файловых систем в рамках одной ОС может быть реализована по-другому, например исходя из концепции виртуальной файловой системы. Виртуальная файловая система (vfs) представляет собой независимый от реализации уровень и опирается на реальные файловые системы (s5fs, ufs, FAT, NFS, FFS. Ext2fs ѕ). При этом возникают структуры данных виртуальной файловой системы типа виртуальных индексных узлов vnode, которые обобщают индексные узлы конкретных систем.

78)Общие сведения об архитектуре компьютера

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 данных определяет 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.

* Структура контроллера устройства

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 данных** предназначен 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.

* Опрос устройств и прерывания. Исключительные ситуации и системные вызовы

Рассмотрим пример -команда записи, теперь уже записи или вывода данных на внешнее устройство. В нашей модели для вывода информации, помещающейся в регистр входных данных, без проверки успешности вывода процессор и контроллер должны связываться следующим образом.

1. Процессор в цикле читает информацию из порта регистра состояний и проверяет значение **бита занятости**. Если **бит занятости** установлен, то это означает, что устройство еще не завершило предыдущую операцию, и процессор уходит на новую итерацию цикла. Если **бит занятости** сброшен, то устройство готово к выполнению новой операции, и процессор переходит на следующий шаг.
2. Процессор записывает код команды вывода в порт регистра управления.
3. Процессор записывает данные в порт регистра входных данных.
4. Процессор устанавливает **бит готовности команды**. В следующих шагах процессор не задействован.
5. Когда контроллер замечает, что **бит готовности команды** установлен, он устанавливает **бит занятости**.
6. Контроллер анализирует код команды в регистре управления и обнаруживает, что это команда вывода. Он берет данные из регистра входных данных и инициирует выполнение команды.
7. После завершения операции контроллер обнуляет **бит готовности команды**.
8. При успешном завершении операции контроллер обнуляет **бит ошибки** в регистре состояния, при неудачном завершении команды – устанавливает его.
9. Контроллер сбрасывает **бит занятости**.

При необходимости вывода новой порции информации все эти шаги повторяются. Если процессор интересует, корректно или некорректно была выведена информация, то после шага 4 он должен в цикле считывать информацию из порта регистра состояний до тех пор, пока не будет сброшен бит занятости устройства, после чего проанализировать состояние бита ошибки.

Как видим, на первом шаге (и, возможно, после шага 4) процессор ожидает освобождения устройства, непрерывно опрашивая значение бита занятости. Такой способ взаимодействия процессора и контроллера получил название polling или, в русском переводе, **способа опроса устройств**.

Для того чтобы процессор не дожидался состояния готовности устройства ввода-вывода в цикле, а мог выполнять в это время другую работу, необходимо, чтобы устройство само умело сигнализировать процессору о своей готовности. Технический механизм, который позволяет внешним устройствам оповещать процессор о завершении команды вывода или команды ввода, получил название механизма **прерываний** .

Для запрещения прерываний, а точнее, для невосприимчивости процессора к внешним прерываниям обычно существуют специальные команды, которые могут маскировать (запрещать) все или некоторые из прерываний устройств ввода-вывода. В то же время определенные кризисные ситуации в вычислительной системе (например, неустранимый сбой в работе оперативной памяти) должны требовать ее немедленной реакции. Такие ситуации вызывают прерывания, которые невозможно замаскировать или запретить и которые поступают в процессор по специальной линии шины прерываний, называемой линией немаскируемых прерываний (NMI – Non-Maskable Interrupt).

Не все внешние устройства являются одинаково важными с точки зрения вычислительной системы. Соответственно, некоторые прерывания являются более существенными, чем другие. Контроллер прерываний обычно позволяет устанавливать приоритеты для прерываний от внешних устройств. При почти одновременном возникновении прерываний от нескольких устройств (во время выполнения одной и той же команды процессора) процессору сообщается номер наиболее приоритетного прерывания для его обслуживания в первую очередь. Менее приоритетное прерывание при этом не пропадает, о нем процессору будет доложено после обработки более приоритетного прерывания. Более того, при обработке возникшего прерывания процессор может получить сообщение о возникновении прерывания с более высоким приоритетом и переключиться на его обработку.

Механизм обработки прерываний, по которому процессор прекращает выполнение команд в обычном режиме и, частично сохранив свое состояние, отвлекается на выполнение других действий, оказался настолько удобен, что зачастую разработчики процессоров используют его и для других целей. Похожим образом процессор обрабатывает **исключительные ситуации** и программные прерывания.

**Исключительные ситуации** возникают во время выполнения процессором команды. К их числу относятся ситуации переполнения, деления на ноль, обращения к отсутствующей странице памяти и т. д.

**Программные прерывания** возникают после выполнения специальных команд, как правило, для выполнения привилегированных действий внутри системных вызовов. Программные прерывания имеют следующие свойства.

* Программное прерывание происходит в результате выполнения специальной команды.
* Процессор при выполнении программного прерывания сохраняет свое состояние перед выполнением **следующей** команды.
* Программные прерывания, естественно, возникают синхронно с работой процессора и **абсолютно предсказуемы** программистом.
* Прямой доступ к памяти (Direct Memory Access – DMA)

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 доступа внешних 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 памяти извещает 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.

* Логические принципы организации ввода-вывода

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://intuit.ru/studies/professional_retraining/941/courses/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://intuit.ru/studies/professional_retraining/941/courses/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.

79) Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами

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.

* Функции базовой подсистемы ввода-вывода

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. Не Блокирующийся 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.
* Буферизация и кэширование, Spooling и захват устройств

##### **Буферизация и кэширование**

Под **буфером** обычно понимается некоторая область памяти для запоминания информации при обмене данных между двумя устройствами, двумя процессами или процессом и устройством. Обмен информацией между двумя процессами относится к области кооперации процессов, и мы подробно рассмотрели его организацию в соответствующей лекции. Здесь нас будет интересовать использование буферов в том случае, когда одним из участников обмена является внешнее устройство. Существует три причины, приводящие к использованию буферов в базовой подсистеме ввода-вывода.

* Первая причина буферизации – это разные скорости приема и передачи информации, которыми обладают участники обмена. Рассмотрим, например, случай передачи потока данных от клавиатуры к модему. Скорость, с которой поставляет информацию клавиатура, определяется скоростью набора текста человеком и обычно существенно меньше скорости передачи данных модемом. Для того чтобы не занимать модем на все время набора текста, делая его недоступным для других процессов и устройств, целесообразно накапливать введенную информацию в буфере или нескольких буферах достаточного размера и отсылать ее через модем после заполнения буферов.
* Вторая причина буферизации – это разные объемы данных, которые могут быть приняты или получены участниками обмена единовременно. Возьмем другой пример. Пусть информация поставляется модемом и записывается на жесткий диск. Помимо обладания разными скоростями совершения операций, модем и жесткий диск представляют собой устройства разного типа. Модем является символьным устройством и выдает данные байт за байтом, в то время как диск является блочным устройством и для проведения операции записи для него требуется накопить необходимый блок данных в буфере. Здесь также можно применять более одного буфера. После заполнения первого буфера модем начинает заполнять второй, одновременно с записью первого на жесткий диск. Поскольку скорость работы жесткого диска в тысячи раз больше, чем скорость работы модема, к моменту заполнения второго буфера операция записи первого будет завершена, и модем снова сможет заполнять первый буфер одновременно с записью второго на диск.
* Третья причина буферизации связана с необходимостью копирования информации из приложений, осуществляющих ввод-вывод, в буфер ядра операционной системы и обратно. Допустим, что некоторый пользовательский процесс пожелал вывести информацию из своего адресного пространства на внешнее устройство. Для этого он должен выполнить системный вызов с обобщенным названием write, передав в качестве параметров адрес области памяти, где расположены данные, и их объем. Если внешнее устройство временно занято, то возможна ситуация, когда к моменту его освобождения содержимое нужной области окажется испорченным (например, при использовании асинхронной формы системного вызова). Чтобы избежать возникновения подобных ситуаций, проще всего в начале работы системного вызова скопировать необходимые данные в буфер ядра операционной системы, постоянно находящийся в оперативной памяти, и выводить их на устройство из этого буфера.

Под словом **кэш** (cache – "тайник, запас"), этимологию которого мы не будем здесь рассматривать, обычно понимают область быстрой памяти, содержащую копию данных, расположенных где-либо в более медленной памяти, предназначенную для ускорения работы вычислительной системы. Мы с вами сталкивались с этим понятием при рассмотрении иерархии памяти. В базовой подсистеме ввода-вывода не следует смешивать два понятия, буферизацию и кэширование, хотя зачастую для выполнения этих функций отводится одна и та же область памяти. Буфер часто содержит единственный набор данных, существующий в системе, в то время как кэш по определению содержит копию данных, существующих где-нибудь еще. Например, буфер, используемый базовой подсистемой для копирования данных из пользовательского пространства процесса при выводе на диск, может в свою очередь применяться как кэш для этих данных, если операции модификации и повторного чтения данного блока выполняются достаточно часто.

Функции буферизации и кэширования не обязательно должны быть локализованы в базовой подсистеме ввода-вывода. Они могут быть частично реализованы в драйверах и даже в контроллерах устройств, скрытно по отношению к базовой подсистеме.

##### **Spooling и захват устройств**

О понятии **spooling** мы говорили в первой лекции нашего курса, как о механизме, впервые позволившем совместить реальные операции ввода-вывода одного задания с выполнением другого задания. Теперь мы можем определить это понятие более точно. Под словом spool мы подразумеваем буфер, содержащий входные или выходные данные для устройства, на котором следует избегать чередования его использования (возникновения interleaving – см. раздел "Interleaving, race condition и взаимоисключения" лекции 5) различными процессами. Правда, в современных вычислительных системах spool для ввода данных практически не используется, а в основном предназначен для накопления выходной информации.

Рассмотрим в качестве внешнего устройства принтер. Хотя принтер не может печатать информацию, поступающую одновременно от нескольких процессов, может оказаться желательным разрешить процессам совершать вывод на принтер параллельно. Для этого операционная система вместо передачи информации напрямую на принтер накапливает выводимые данные в буферах на диске, организованных в виде отдельного spool-файла для каждого процесса. После завершения некоторого процесса соответствующий ему spool-файл ставится в очередь для реальной печати. Механизм, обеспечивающий подобные действия, и получил название spooling.

В некоторых операционных системах вместо использования spooling для устранения race condition применяется механизм монопольного захвата устройств процессами. Если устройство свободно, то один из процессов может получить его в монопольное распоряжение. При этом все другие процессы при попытке осуществления операций над этим устройством будут либо блокированы (переведены в состояние **ожидание** ), либо получат информацию о невозможности выполнения операции до тех пор, пока процесс, захвативший устройство, не завершится или явно не сообщит операционной системе о своем отказе от его использования.

Обеспечение spooling и механизма захвата устройств является прерогативой базовой подсистемы ввода-вывода.

* Обработка прерываний и ошибок

Если при работе с внешним устройством вычислительная система не пользуется методом опроса его состояния, а задействует механизм прерываний, то при возникновении прерывания, как мы уже говорили раньше, процессор, частично сохранив свое состояние, передает управление специальной программе обработки прерывания. Мы уже рассматривали действия операционной системы над процессами, происходящими при возникновении прерывания, в разделе "Переключение контекста" лекции 2, где после возникновения прерывания осуществлялись следующие действия: сохранение контекста, обработка прерывания, планирование использования процессора, восстановление контекста. Тогда мы обращали больше внимания на действия, связанные с сохранением и восстановлением контекста и планированием использования процессора. Теперь давайте подробнее остановимся на том, что скрывается за словами "обработка прерывания ".

Одна и та же процедура обработки прерывания может применяться для нескольких устройств ввода-вывода (например, если эти устройства используют одну линию прерываний, идущую от них к контроллеру прерываний ), поэтому первое действие собственно программы обработки состоит в определении того, какое именно устройство выдало прерывание. Зная устройство, мы можем выявить процесс, который инициировал выполнение соответствующей операции. Поскольку прерывание возникает как при удачном, так и при неудачном ее выполнении, следующее, что мы должны сделать, – это определить успешность завершения операции, проверив значение **бита ошибки** в регистре состояния устройства. В некоторых случаях операционная система может предпринять определенные действия, направленные на компенсацию возникшей ошибки. Например, в случае возникновения ошибки чтения с гибкого диска можно попробовать несколько раз повторить выполнение команды. Если компенсация ошибки невозможна, то операционная система впоследствии известит об этом процесс, запросивший выполнение операции, (например, специальным кодом возврата из системного вызова). Если этот процесс был заблокирован до выполнения завершившейся операции, то операционная система переводит его в состояние **готовность**. При наличии других неудовлетворенных запросов к освободившемуся устройству операционная система может инициировать выполнение следующего запроса, одновременно известив устройство, что прерывание обработано. На этом, собственно, обработка прерывания заканчивается, и система может приступать к планированию использования процессора.

Действия по обработке прерывания и компенсации возникающих ошибок могут быть частично переложены на плечи соответствующего драйвера. Для этого в состав интерфейса между драйвером и базовой подсистемой ввода-вывода добавляют еще одну функцию – функцию обработки прерывания intr.

* Планирование запросов

При использовании неблокирующегося системного вызова может оказаться, что нужное устройство уже занято выполнением некоторых операций. В этом случае неблокирующийся вызов может немедленно вернуться, не выполнив запрошенных команд. При организации запроса на совершение операций ввода-вывода с помощью блокирующегося или асинхронного вызова занятость устройства приводит к необходимости постановки запроса в очередь к данному устройству. В результате с каждым устройством оказывается связан список неудовлетворенных запросов процессов, находящихся в состоянии **ожидания**, и запросов, выполняющихся в асинхронном режиме. Состояние **ожидание** расщепляется на набор очередей процессов, ожидающих различных устройств ввода-вывода (или ожидающих изменения состояний различных объектов – семафоров, очередей сообщений, условных переменных в мониторах и т. д. – см. лекцию 6).

После завершения выполнения текущего запроса операционная система (по ходу обработки возникшего прерывания ) должна решить, какой из запросов в списке должен быть удовлетворен следующим, и инициировать его исполнение. Точно так же, как для выбора очередного процесса на исполнение из списка готовых нам приходилось осуществлять краткосрочное планирование процессов, здесь нам необходимо осуществлять планирование применения устройств, пользуясь каким-либо алгоритмом этого планирования. Критерии и цели такого планирования мало отличаются от критериев и целей планирования процессов.

Задача планирования использования устройства обычно возлагается на базовую подсистему ввода-вывода, однако для некоторых устройств лучшие алгоритмы планирования могут быть тесно связаны с деталями их внутреннего функционирования. В таких случаях операция планирования переносится внутрь драйвера соответствующего устройства, так как эти детали скрыты от базовой подсистемы. Для этого в интерфейс драйвера добавляется еще одна специальная функция, которая осуществляет выбор очередного запроса, – функция strategy.