**1.Структура вычислительной системы**

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

Любая вычислительная система состоит из того, что в англоязычных странах принято называть словом hardware, или техническое обеспечение: *процессор*, память, монитор, дисковые устройства и т.д., объединенные магистральным соединением, которое называется шиной. Во-вторых, вычислительная система состоит из программного обеспечения. Все программное обеспечение принято делить на две части: прикладное и системное. К прикладному программному обеспечению, как правило, относятся разнообразные банковские и прочие бизнес-программы, игры, текстовые процессоры и т. п. Под системным программным обеспечением обычно понимают программы, способствующие функционированию и разработке прикладных программ. Надо сказать, что деление на прикладное и системное программное обеспечение является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушенный в программировании, может считать Microsoft Word системной программой, а, с точки зрения программиста, это – приложение. Компилятор языка Си для обычного программиста – системная программа, а для системного – прикладная.

**2.Операционная система как виртуальная машина**

При разработке ***ОС***  широко применяется абстрагирование, которое является важным методом упрощения и позволяет сконцентрироваться на взаимодействии высокоуровневых компонентов системы, игнорируя детали их реализации. В этом смысле *ОС* представляет собой интерфейс между пользователем и компьютером. Архитектура большинства компьютеров на уровне машинных команд очень неудобна для использования прикладными программами. Например, работа с диском предполагает знание внутреннего устройства его электронного компонента – контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т. д. Ясно, что средний программист не в состоянии учитывать все особенности работы оборудования (в современной терминологии – заниматься разработкой драйверов устройств), а должен иметь простую высокоуровневую абстракцию, скажем, представляя информационное пространство диска, как набор файлов. Файл можно открывать для чтения или записи, использовать для получения или сброса информации, а потом закрывать. Это концептуально проще, чем заботиться о деталях перемещения головок дисков или организации работы мотора. Аналогичным образом, с помощью простых и ясных абстракций, скрываются от программиста все ненужные подробности организации *прерываний*, работы таймера, управления памятью и т. д. Более того, на современных вычислительных комплексах можно создать иллюзию неограниченного размера оперативной памяти и числа *процессоров*. Всем этим занимается *операционная система*. Таким образом, ***операционная система*** представляется пользователю *виртуальной машиной*, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

**3.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.

**4.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.

**5.Операционная система как постоянно функционирующее ядро**

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

**6.Основные понятия, концепции ОС. Системные вызовы**.

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

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

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

В большинстве *операционных систем**системный вызов* осуществляется командой программного *прерывания* (INT). Программное *прерывание* – это синхронное событие, которое может быть повторено при выполнении одного и того же программного кода.

**7.Основные понятия, концепции ОС. Прерывания**

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

**8.Основные понятия, концепции ОС. Исключительные ситуации**

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

**9.Основные понятия, концепции ОС. 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*.

**10.Архитектурные особенности ОС. Монолитное ядро**

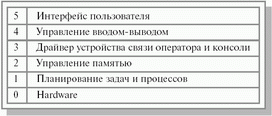
*Монолитное ядро* представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, ***монолитное ядро*** – это такая схема *операционной системы*, при которой все ее компоненты являются составными частями одной программы, используют 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* с *монолитным ядром* сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается *операционная система*. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция – это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность *операционной системы* в целом.

Даже в монолитных системах можно выделить некоторую структуру. Как в бетонной глыбе можно различить вкрапления щебенки, так и в *монолитном ядре* выделяются вкрапления сервисных процедур, соответствующих *системным вызовам*. Сервисные процедуры выполняются в привилегированном режиме, тогда как пользовательские программы – в непривилегированном. Для перехода с одного уровня привилегий на другой иногда может использоваться главная сервисная программа, определяющая, какой именно *системный вызов* был сделан, корректность входных данных для этого *вызова* и передающая управление соответствующей сервисной процедуре с переходом в привилегированный режим работы. Иногда выделяют также набор программных утилит, которые помогают выполнять сервисные процедуры.

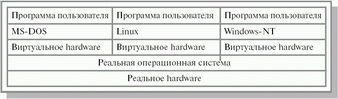
**11.Архитектурные особенности ОС. Многоуровневые системы (Layered systems)**

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. 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.

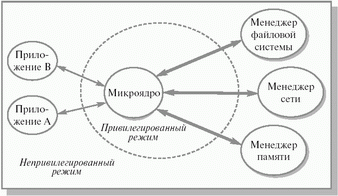
**12.Архитектурные особенности ОС. Виртуальные машины**

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. 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*.

**13.Архитектурные особенности ОС. Микроядерная архитектура**

***микроядерной архитектурой*** (*microkernel* architecture) *операционной системы*, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования *процессора*, первичную обработку *прерываний*, операции ввода-вывода и базовое управление памятью.

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

Основное достоинство *микроядерной архитектуры* – 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 непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

В то же время *микроядерная архитектура* *операционной системы* вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядерная *операционная система* по скорости не уступала *операционным системам* на базе *монолитного ядра*, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных *операционных систем* – необходимость очень аккуратного проектирования.

**14.Архитектурные особенности ОС. Смешанные системы**

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

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

**15.Классификация ОС. Реализация многозадачности. Поддержка многопользовательского режима. Многопроцессорная обработка. Системы реального времени**

#### Реализация многозадачности

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

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

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

В некоторых *ОС* (Windows 3.11, например) пользовательская программа может монополизировать *процессор*, то есть работать в невытесняющем режиме. Как правило, в большинстве систем не подлежит вытеснению код собственно *ОС*. Ответственные программы, в частности задачи реального времени, также не вытесняются.

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

#### Поддержка многопользовательского режима

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

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

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

#### Многопроцессорная обработка

В результате требований к повышению производительности появились ***многопроцессорные системы***, состоящие из двух и более *процессоров* общего назначения, осуществляющих параллельное выполнение команд. Многопроцессорная обработка реализована в Linux, , Windows NT

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

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

#### Системы реального времени

Они используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем они могут поступать, причем от нескольких источников одновременно. Столь жесткие ограничения сказываются на архитектуре *систем реального времени*, например, в них может отсутствовать виртуальная память, поддержка которой дает непредсказуемые задержки в выполнении программ.

**16.Понятие процесса**

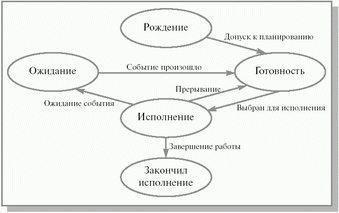
Понятие ***процесса*** характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между *процессами* и программами, обрабатываемыми вычислительными системами. Как будет показано далее, в некоторых операционных системах для работы определенных программ может организовываться более одного *процесса* или один и тот же *процесс* может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного *процесса* нельзя считать, что *процесс* представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. *Процесс* находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

**17.Состояния процесса**

Всякий новый *процесс*, появляющийся в системе, попадает в *состояние* готовность. *Операционная система*, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых *процессов* и переводит его в *состояние* исполнение. В *состоянии* исполнение происходит непосредственное выполнение программного кода *процесса*. Выйти из этого *состояния* *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.

Из *состояния* ожидание процесс попадает в *состояние* готовность после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.



Теперь для появления в вычислительной системе *процесс* должен пройти через *состояние* рождение. При рождении *процесс* получает в свое распоряжение *адресное пространство*, в которое загружается программный код *процесса*; ему выделяются *стек* и системные ресурсы; устанавливается начальное *значение* программного счетчика этого *процесса* и т. д. Родившийся *процесс* переводится в *состояние* готовность. При *завершении* своей деятельсти *процесс* из *состояния* исполнение попадает в *состояние* закончил исполнение. Модель *состояний процессов* для операционной системы *Windows* NT содержит 7 различных *состояний*, а для операционной системы Unix – 9

**18.Операции над процессами и связанные с ними понятия. Набор операций**

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

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

**19.Операции над процессами и связанные с ними понятия. Process Control Block и контекст процесса**

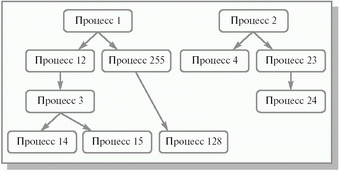
Для того чтобы операционная система могла выполнять *операции над процессами*, каждый *процесс* представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного *процесса*:

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

Для нас важно лишь то, что для любого *процесса*, находящегося в вычислительной системе, вся информация, необходимая для совершения *операций* над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее *PCB* (*Process Control* Block) или *блоком управления процессом*. ***Блок управления процессом*** является моделью *процесса* для операционной системы. Любая *операция*, производимая операционной системой над *процессом*, вызывает определенные изменения в *PCB* .В рамках принятой модели *состояний процессов* содержимое *PCB* между *операциями* остается постоянным.

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

**20.Операции над процессами и связанные с ними понятия. Одноразовые операции**

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

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

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

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

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

После того как *процесс* наделен содержанием, в *PCB* дописывается оставшаяся информация, и *состояние* нового *процесса* изменяется на *готовность*. В это время процесс-родитель может продолжать свое выполнение одновременно с выполнением процесса-ребенка, а может ожидать *завершения* работы некоторых или всех своих "детей".

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

**21.Операции над процессами и связанные с ними понятия. Многоразовые операции**

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

***Запуск процесса***. Из числа *процессов*, находящихся в *состоянии* готовность, операционная система выбирает один *процесс* для последующего исполнения. Для избранного *процесса* операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. Далее *состояние процесса* изменяется на *исполнение*, восстанавливаются значения регистров для данного *процесса* и управление передается команде, на которую указывает счетчик команд *процесса*. Все данные, необходимые для восстановления *контекста*, извлекаются из *PCB процесса*, над которым совершается *операция*.

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

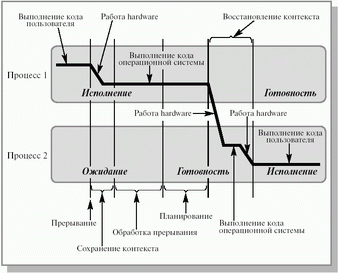
***Блокирование процесса***. *Процесс* блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов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

***Разблокирование процесса***. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый *процесс* в *состоянии* ожидание для данного события, и если находился, переводит его в *состояние* готовность, выполняя необходимые действия, связанные с наступлением события

**22.Операции над процессами и связанные с ними понятия. Переключение контекста**

Деятельность мультипрограммной операционной системы состоит из цепочек *операций*, выполняемых над различными *процессами*, и сопровождается переключением процессора с одного *процесса* на другой.

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

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

**23.Уровни планирования**

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

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

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

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

**24.Критерии планирования и требования к алгоритмам**

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

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

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

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

**25.Параметры планирования**

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

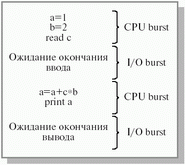
К статическим *параметрам* вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические *параметры* системы описывают количество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



**Рис. 3.1.**Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода

Для *краткосрочного планирования* нам понадобится ввести еще два динамических *параметра*. *Деятельность* любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название ***CPU burst***, а промежуток времени непрерывного ожидания ввода-вывода – ***I/O burst*** .

**26.Вытесняющее и не вытесняющее планирование**

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

1. Когда процесс переводится из состояния исполнение в состояние закончил исполнение.
2. Когда процесс переводится из состояния исполнение в состояние ожидание.
3. Когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера).
4. Когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие).

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* *не вытесняющее (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.

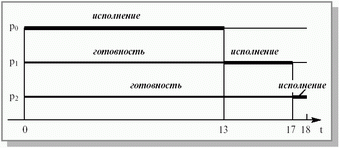
**27.Алгоритмы планирования. First-Come, First-Served (FCFS)**

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

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| **Процесс** | p0 | p1 | p2 |
| **Продолжительность очередного *CPU burst*** | 13 | 4 | 1 |

Преимуществом *алгоритма FCFS* является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных *CPU burst* . Эти времена приведены в [таблице 3.1.](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=3#table.3.1) в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка *CPU burst* , что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

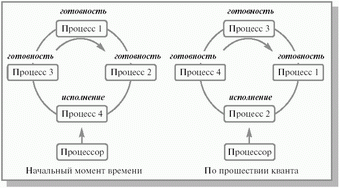
Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на [рисунке 3.2](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=3#image.3.2). Первым для выполнения выбирается процесс p0, который получает процессор на все время своего *CPU burst* , т. е. на 13 единиц времени. После его окончания в состояние исполнение переводится процесс p1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 – 13 единиц, для процесса p2 – 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0составляет 13 единиц времени, для процесса p1 – 13 + 4 = 17 единиц, для процесса p2 – 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.



**Рис. 3.2.**Выполнение процессов при порядке p0,p1,p2

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

**28.Алгоритмы планирования. Round Robin (RR)**

Модификацией *алгоритма FCFS* является алгоритм, получивший название *Round Robin (Колесо обозрения)*  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 (см. [рис. 3.4.](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=3#image.3.4)). 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.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной *кванта времени* равной 4. Выполнение этих процессов иллюстрируется [таблицей 3.2](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=3#table.3.2). Обозначение "И" используется в ней для процесса, находящегося в состоянии исполнение, обозначение "Г" – для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.2. | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | И | И | И | Г | Г | Г | Г | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | Г | Г | Г | И | И | И | И |  |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | И |  |  |  |  |  |  |  |  |  |

На производительность *алгоритма RR* сильно влияет величина *кванта времени*. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины *кванта времени*, равной 1 (см. [табл. 3.3.](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=3#table.3.3)). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.3. | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | Г | Г | И | Г | И | Г | И | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | И | Г | Г | И | Г | И | Г | И |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

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

**29.Алгоритмы планирования. Shortest-Job-First (SJF)**

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 *невытесняющим*.

Рассмотрим пример работы *невытесняющего алгоритма SJF*. Пусть в состоянии готовность находятся четыре процесса, p0, p1, p2 и p3, для которых известны времена их очередных *CPU burst*. Эти времена приведены в [таблице 3.4](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=4#table.3.4). Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка *CPU burst*, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 3.4. | | | | |
| **Процесс** | p0 | p1 | p2 | p3 |
| **Продолжительность очередного *CPU burst*** | 5 | 3 | 7 | 1 |

*Не вытесняющий* *алгоритм*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.5. | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **p0** | Г | Г | Г | Г | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) *алгоритм SJF* является оптимальным с точки зрения минимизации среднего времени ожидания среди класса *невытесняющих* алгоритмов.

Для рассмотрения примера *вытесняющего SJF* *планирования* мы возьмем ряд процессов p0, p1, p2 и p3 с различными временами *CPU burst* и различными моментами их появления в очереди процессов, готовых к исполнению (см. [табл. 3.6.](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=4#table.3.6)).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.6. | | | | | | | | | | | | | | | | | | | | | | | |
| **Процесс** | | | **Время появления в очереди очередного *CPU burst*** | | | | | | | | | | | | | | | | | | | | **Продолжительность** |
| **p0** | | | 0 | | | | | | | | | | | | | | | | | | | | 6 |
| **p1** | | | 2 | | | | | | | | | | | | | | | | | | | | 2 |
| **p2** | | | 6 | | | | | | | | | | | | | | | | | | | | 7 |
| **p3** | | | 0 | | | | | | | | | | | | | | | | | | | | 5 |
| Таблица 3.7. | | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | | Г | Г | Г | Г | Г | И | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** |  |  | | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** |  |  | |  |  |  |  | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И | И | | Г | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |

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

**30.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*. К недостаткам этого алгоритма можно отнести невозможность 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.

**31.Алгоритмы планирования. 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.

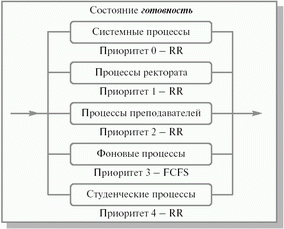
Пусть в очередь процессов, находящихся в состоянии готовность, поступают те же процессы, что и в примере для *вытесняющего алгоритма SJF*, только им дополнительно еще присвоены *приоритеты* (см. [табл. 3.8.](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=5#table.3.8)). В вычислительных системах не существует определенного соглашения, какое значение *приоритета* – 1 или 4 считать более *приоритетным*. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему *приоритету*, т. е. наиболее *приоритетным* в нашем примере является процесс p3, а наименее *приоритетным* – процесс p0.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 3.8. | | | |
| **Процесс** | **Время появления в очереди** | **Продолжительность очередного *CPU burst*** | ***Приоритет*** |
| **p0** | 0 | 6 | 4 |
| **p1** | 2 | 2 | 3 |
| **p2** | 6 | 7 | 2 |
| **p3** | 0 | 5 | 1 |

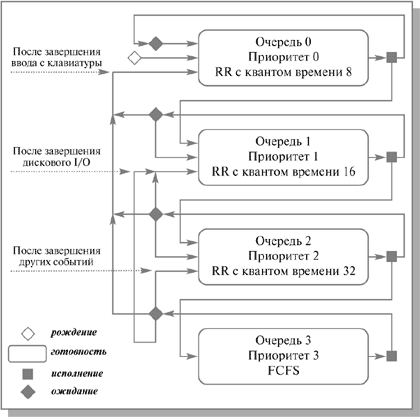
Иным будет предоставление процессора процессам в случае *вытесняющего приоритетного планирования* (см. [табл. 3.10.](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=5#table.3.10)). Первым, как и в предыдущем случае, начнет исполняться процесс p3, а по его окончании – процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше, будет исполняться процесс p0.

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

**32.Алгоритмы планирования. Многоуровневые очереди (Multilevel Queue)**

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов *планирования*. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность (см. [рис. 3.5](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=6#image.3.5)). 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. 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.

**33.Алгоритмы планирования. 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. Для простоты рассмотрим ситуацию, когда процессы в состоянии готовность организованы в 4 очереди, как на [рисунке 3.6](http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=6#image.3.6). *Планирование процессов* между очередями осуществляется на основе *вытесняющего* приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее *приоритет*. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс и т.д. Если при работе процесса появляется другой процесс в какой-либо более *приоритетной* очереди, исполняющийся процесс вытесняется новым. Родившийся процесс поступает в очередь 0. Чем больше значение продолжительности *CPU burst*, тем в менее *приоритетную* очередь попадает процесс, но тем на большее процессорное время он может рассчитывать. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора, окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, – в низкоприоритетных.

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

*Многоуровневые очереди с обратной связью* представляют собой наиболее общий подход к *планированию процессов* из числа подходов, рассмотренных нами. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибкостью. 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.

**34.Взаимодействующие процессы**

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой. Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их *кооперации*?

* *Повышение скорости работы*. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи.
* *Совместное использование данных*. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
* *Модульная конструкция какой-либо системы*. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные *процессы*, взаимодействующие путем передачи *сообщений* через микроядро.
* Наконец, это может быть необходимо просто *для удобства работы пользователя*, желающего, например, редактировать и отлаживать программу одновременно.

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.

**35.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.

**36.Логическая организация механизма передачи информации. Как устанавливается связь?**

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.

**39.Логическая организация механизма передачи информации. Надежность средств 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, 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.

**40.Нити исполнения**

В свое время внедрение идеи мультипрограммирования позволило повысить пропускную способность компьютерных систем, т. е. уменьшить среднее *время ожидания* результатов работы процессов. Но 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*.

**41.Interleaving, race condition и взаимоисключения**

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 d b c e f

d e f a b c и т.д.

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

P: x=2 Q: x=3

y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для *активностей* общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор *активностей* (например, программ) ***детерминирован***, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он ***недетерминирован***.

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.

**42.Критическая секция**

Важным понятием при изучении способов синхронизации процессов является понятие *критической секции* (*critical section*) программы. ***Критическая секция*** – это часть программы, исполнение которой может привести к возникновению *race condition* для определенного набора программ. Чтобы исключить эффект гонок *по* отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей *критической секции*, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию *взаимоисключения* для *критических секций* программ. Реализация *взаимоисключения* для *критических секций* программ с практической точки зрения означает, что *по* отношению к другим процессам, участвующим во взаимодействии, *критическая секция* начинает выполняться как *атомарная операция*. Давайте рассмотрим следующий пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов (*таблица* 5.1):

Здесь *критический участок* для каждого процесса – от *операции* "Обнаруживает, что хлеба нет" до *операции* "Возвращается в комнату" включительно. В результате отсутствия *взаимоисключения* мы из ситуации "Нет хлеба" попадаем в ситуацию "Слишком много хлеба". Если бы этот *критический участок* выполнялся как *атомарная операция* – "Достает два батона хлеба", то проблема образования излишков была бы снята.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.1. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Обнаруживает, что хлеба нет |  |  |
| 17-09 | Уходит в магазин |  |  |
| 17-11 |  | Приходит в комнату |  |
| 17-13 |  | Обнаруживает, что хлеба нет |  |
| 17-15 |  | Уходит в магазин |  |
| 17-17 |  |  | Приходит в комнату |
| 17-19 |  |  | Обнаруживает, что хлеба нет |
| 17-21 |  |  | Уходит в магазин |
| 17-23 | Приходит в магазин |  |  |
| 17-25 | Покупает 2 батона на всех |  |  |
| 17-27 | Уходит из магазина |  |  |
| 17-29 |  | Приходит в магазин |  |
| 17-31 |  | Покупает 2 батона на всех |  |
| 17-33 |  | Уходит из магазина |  |
| 17-35 |  |  | Приходит в магазин |
| 17-37 |  |  | Покупает 2 батона на всех |
| 17-39 |  |  | Уходит из магазина |
| 17-41 | Возвращается в комнату |  |  |
| 17-43 |  |  |  |
| 17-45 |  |  |  |
| 17-47 |  | Возвращается в комнату |  |
| 17-49 |  |  |  |
| 17-51 |  |  |  |
| 17-53 |  |  | Возвращается в комнату |

Сделать процесс добывания хлеба *атомарной операцией* можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а *по* окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (*таблица* 5.2).

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.2. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Достает два батона хлеба |  |  |
| 17-43 |  | Приходит в комнату |  |
| 17-47 |  |  | Приходит в комнату |

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

while (some condition) {

entry section

critical section

exit section

remainder section

}

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

**43.Программные алгоритмы организации взаимодействия процессов. Требования, предъявляемые к алгоритмам**

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

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд *взаимоисключения*. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются *атомарными операциями*.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем *критическом участке*, то не существует никаких других процессов, которые исполняются в соответствующих *критических секциях*. Это условие получило название условия *взаимоисключения (mutual exclusion)*.
4. Процессы, которые находятся вне своих *критических участков* и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные *критические участки*. Если нет процессов в *критических секциях* и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в *remainder* section, должны принимать решение о том, какой процесс войдет в свою *критическую секцию*. Такое решение не должно приниматься бесконечно долго. Это условие получило название ***условия прогресса (progress)*** .
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой *критический участок*. От того момента, когда процесс запросил разрешение на вход в *критическую секцию*, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои *критические участки* лишь ограниченное число раз. Это условие получило название ***условия ограниченного ожидания (bound waiting)*** .

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

**44.Программные алгоритмы организации взаимодействия процессов. 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*.

**45.Программные алгоритмы организации взаимодействия процессов. 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*).

**46.Программные алгоритмы организации взаимодействия процессов. 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.

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

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

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

**47.Программные алгоритмы организации взаимодействия процессов. 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

}

**48.Программные алгоритмы организации взаимодействия процессов. 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

}

#### 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

}

**49.Семафоры. 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 Разблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

**50.Семафоры. Решение проблемы producer-consumer с помощью семафоров**

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

Producer: while (1) {

produce\_item;

put\_item;

}

Consumer: while (1) {

get\_item;

consume\_item;

}

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

Semaphore mutex = 1;

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

Semaphore full = 0;

Producer:

while (1) {

produce\_item;

P(empty);

P(mutex);

put\_item;

V(mutex);

V(full);

}

Consumer:

while (1) {

P(full);

P(mutex);

get\_item;

V(mutex);

V(empty);

consume\_item;

}

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.

**51.Мониторы**

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

monitor monitor\_name {

описание внутренних переменных;

void m1(...) {...

}

void m2(...){...

}

...

void mn(...) {...

}

{

блок инициализации

внутренних переменных;

}

}

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

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

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

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

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

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

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.

**52.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*.

**53 .Файлы с точки зрения пользователя. Файловая система.**

**Файловая система** - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во *внешней памяти*, и обеспечить пользователю удобный *интерфейс* при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском - прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. *Операции* записи или чтения *файла* концептуально проще, чем низкоуровневые *операции* работы с устройствами.

Основная идея использования *внешней памяти* состоит в следующем. ОС делит память на блоки фиксированного размера, например, 4096 байт. *Файл*, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранится в виде последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. В некоторых ОС (MS-DOS) адреса блоков, содержащих данные *файла*, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. В других ОС (Unix) адреса блоков данных *файла* хранятся в отдельном блоке *внешней памяти* (так называемом индексе или индексном узле). Этот прием, называемый ***индексацией***, является наиболее распространенным для приложений, требующих произвольного доступа к записям *файлов*. *Индекс* *файла* состоит из списка элементов, каждый из которых содержит номер блока в *файле* и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой **текущей** позиции, которая характеризуется смещением от начала *файла*. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. *Адрес* же нужного блока диска можно затем извлечь из индекса *файла*. Базовой операцией, выполняемой по отношению к *файлу*, является чтение блока с диска и перенос его в *буфер*, находящийся в основной памяти.

*Файловая система* позволяет при помощи системы справочников (каталогов, *директорий*) связать уникальное имя *файла* с блоками вторичной памяти, содержащими данные *файла*. Иерархическая структура *каталогов*, используемая для управления *файлами*, может служить другим примером индексной структуры. В этом случае *каталоги* или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся *файловая система* компьютера представляет собой большой индексированный *файл*. Помимо собственно *файлов* и структур данных, используемых для управления *файлами* (каталоги, дескрипторы *файлов*, различные таблицы распределения *внешней памяти*), понятие "*файловая система*" включает *программные средства*, реализующие различные *операции над файлами*.

Перечислим **основные функции** файловой системы.

1. Идентификация *файлов*. Связывание имени *файла* с выделенным ему пространством *внешней памяти*.
2. Распределение *внешней памяти* между *файлами*. Для работы с конкретным *файлом* пользователю не требуется иметь информацию о местоположении этого *файла* на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
3. Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.
4. Обеспечение защиты от несанкционированного доступа.
5. Обеспечение совместного *доступа к файлам*, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
6. Обеспечение высокой производительности.

Иногда говорят, что ***файл*** - это поименованный набор связанной информации, записанной во вторичную память. Для большинства пользователей *файловая система* - наиболее видимая часть ОС. Она предоставляет механизм для онлайнового хранения и доступа как к данным, так и к программам для всех пользователей системы. С точки зрения пользователя, ***файл*** - единица *внешней памяти*, то есть данные, записанные на диск, должны быть в составе какого-нибудь *файла*.

Важный аспект организации файловой системы - учет стоимости операций взаимодействия с вторичной памятью. Процесс считывания блока диска состоит из позиционирования считывающей головки над дорожкой, содержащей требуемый блок, ожидания, пока требуемый блок сделает оборот и окажется под головкой, и собственно считывания блока. Для этого требуется значительное время (десятки миллисекунд). В современных компьютерах обращение к диску осуществляется примерно в 100 000 раз медленнее, чем обращение к оперативной памяти. Таким образом, критерием вычислительной сложности алгоритмов, работающих с *внешней памятью*, является количество обращений к диску.

**54 .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).

**55.Общие сведения о файлах. Типы файлов.**

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.

**56.Общие сведения о файлах. 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*.

**57.Организация файлов и доступ к ним. Последовательный файл**

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. 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*).

**58.Организация файлов и доступ к ним. 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*. 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.

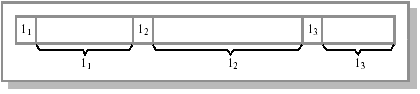
Подобную логическую структуру имеют *файлы* во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого *файла*. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки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

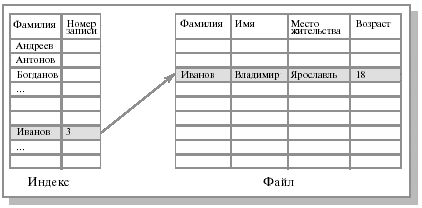
**59.Организация файлов и доступ к ним. Другие формы организации файлов**

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](http://www.intuit.ru/studies/courses/2192/31/lecture/988?page=1#image.11.1)). Базисная операция в данном случае - считать запись с каким-либо значением ключа. Записи могут располагаться в *файле* последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям *последовательного файла* называется **индексно-последовательным**.

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](http://www.intuit.ru/studies/courses/2192/31/lecture/988?page=1#image.11.2)иллюстрирует решение данной проблемы - организацию метода *доступа к файлу* с использованием индекса. 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).

**60.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](http://www.intuit.ru/studies/courses/2192/31/literature#literature.30)].

* Создание *файла*, не содержащего данных. Смысл данного вызова - объявить, что *файл* существует, и присвоить ему ряд *атрибутов*. При этом выделяется место для *файла* на диске и вносится запись в *каталог*.
* Удаление *файла* и освобождение занимаемого им дискового пространства.
* Открытие *файла*. Перед использованием *файла* процесс должен его открыть. Цель данного системного вызова - разрешить системе проанализировать *атрибуты файла* и проверить права доступа к нему, а также считать в оперативную память список адресов блоков *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 *файло*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](http://www.intuit.ru/studies/courses/2192/31/literature#literature.23)].

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

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

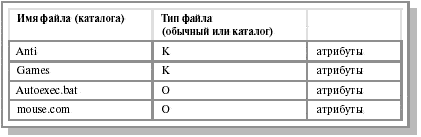
Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой *операции* становятся независимыми от результатов предыдущей *операции*; поэтому он иногда применяется в распределенных файловых системах (например, *Sun* *NFS*).

**61.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*).

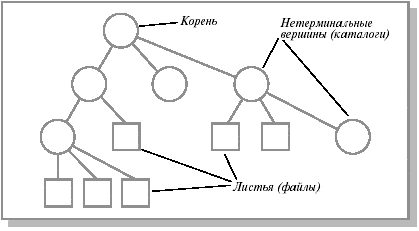
Каждый *каталог* содержит *список* *каталогов* и/или *файлов*, содержащихся в данном *каталоге*. *Каталоги* имеют один и тот же внутренний формат, где каждому *файлу* соответствует одна *запись* в *файле* *директории* (см., например, [рис.11.3](http://www.intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.3)).

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



**Рис. 11.3.**Директории

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



**Рис. 11.4.**Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на [рис. 11.4](http://www.intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.4), наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется *терминальной 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*.

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.

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.

**62.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](http://www.intuit.ru/studies/courses/2192/31/literature#literature.30)].

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

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

**63.Защита файлов**

*Информация* в компьютерной системе должна быть защищена как от физического **разрушения** (*reliability*), так и от несанкционированного **доступа** (*protection*).

#### 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.

**64.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 следствия.

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

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

* **Владелец** (Owner).
* **Группа** (Group). Набор 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).