**1. Понятие ОС**

Операционная система (ОС) - комплекс системных и управляющих программ, предназначенных для наиболее эффективного использования всех ресурсов вычислительной системы (ВС) (Вычислительная система - взаимосвязанная совокупность аппаратных средств вычислительной техники и программного обеспечения, предназначенная для обработки информации) и удобства работы с ней.

Назначение ОС - организация вычислительного процесса в вычислительной системе, рациональное распределение вычислительных ресурсов между отдельными решаемыми задачами; предоставление пользователям многочисленных сервисных средств, облегчающих процесс программирования и отладки задач. Операционная система исполняет роль своеобразного интерфейса ( Интерфейс - совокупность аппаратуры и программных средств, необходимых для подключения периферийных устройств к ПЭВМ) между пользователем и ВС, т.е. ОС предоставляет пользователю виртуальную ВС. Это означает, что ОС в значительной степени формирует у пользователя представление о возможностях ВС, удобстве работы с ней, ее пропускной способности. Различные ОС на одних и тех же технических средствах могут предоставить пользователю различные возможности для организации вычислительного процесса или автоматизированной обработки данных.  
 В программном обеспечении ВС операционная система занимает основное положение, поскольку осуществляет планирование и контроль всего вычислительного процесса. Любая из компонент программного обеспечения обязательно работает под управлением ОС.  
 В соответствии с условиями применения различают три режима ОС: пакетной обработки, разделения времени и реального времени. В режиме пакетной обработки ОС последовательно выполняет собранные в пакет задания. В этом режиме пользователь не имеет контакта с ЭВМ, получая лишь результаты вычислений. В режиме разделения времени ОС одновременно выполняет несколько задач, допуская обращение каждого пользователя к ЭВМ. В режиме реального времени ОС обеспечивает управление объектами в соответствии с принимаемыми входными сигналами. Время отклика ЭВМ с ОС реального времени на возмущающее воздействие должно быть минимальным.

**2. История развития операционных систем**

История развития операционных систем насчитывает уже много лет.

**Первое поколение** (1945–1955): электронные лампы.

Первый настоящий цифровой компьютер был чисто механической машиной без ОС.

**Второе поколение** (1955–1965): транзисторы и системы пакетной обработки. В основном программы для них составлялись на языке Фортран и ассемблере, а типичными ОС были FMS (Fortran Monitor System) и IBSYS.

**Третье поколение** (1965–1980): интегральные схемы и многозадачность

Появилась OS/36/, примерно на два или три порядка превышающая по объему FMS. Самым важным достижением явилась многозадачность (разбиение памяти на несколько частей). Система MULTICS не получила распространения, но оказала влияние на последующие ОС. Был разработан стандарт системы UNIX, названный POSIX, который в настоящее время поддерживается большинством версий UNIX.

**Четвертое поколение** (с 1980 года по наши дни): персональные компьютеры

Следующий период эволюции операционных систем связан с появлением БИС — больших интегральных схем. Появилась DOS (дисковая операционная система). Видоизмененная система была переименована в MS-DOS (MicroSoft Disk Operating System). CP/M, MS-DOS и другие операционные системы для первых микрокомпьютеров полностью основывались на командах, вводимых пользователем с клавиатуры. С 1985 по 1995 год Windows была просто графической оболочкой, работавшей поверх MS-DOS. После была выпущена самостоятельная версия Windows — Windows 95. В 1998 её модифицировали до Windows 98. Другой операционной системой Microsoft была Windows NT (New Technology). Она представляла собой полноценную 32-разрядную систему. В 2001 году была выпущена слегка обновленная версия Windows 2000, названная Windows XP. Эта версия выпускалась намного дольше, по существу заменяя все предыдущие версии Windows. После Windows 2000 серверная включала Windows Server 2003 и Windows 2008. Затем в январе 2007 года Microsoft выпустила окончательную версию преемника Windows XP под названием Vista. Microsoft надеялась, что она полностью заменит Windows XP, но этого так и не произошло. С появлением Windows 7, новой и менее требовательной к ресурсам операционной системы, многие решили вообще пропустить Vista. В 2012 году Microsoft выпустила ее преемника — Windows 8. Другим основным конкурентом в мире персональных компьютеров является ОС UNIX. Она имеет более сильные позиции на сетевых и промышленных серверах, также она находит все более широкое распространение и на настольных компьютерах, ноутбуках, планшетных компьютерах и смартфонах. Ее производные нашли широкое применение на мобильных устройствах, которые работают под управлением iOS 7 или Android.

**Пятое поколение** (с 1990 года по наши дни): мобильные компьютеры

В первое десятилетие после своего появления большинство смартфонов работало под управлением Symbian OS. Но долю рынка Symbian начали отбирать другие операционные системы, например RIM Blackberry OS и Apple iOS. Доля рынка популярность Symbian упала.

**3. Процессы**

Проце́сс — это идентифицируемая абстракция совокупности взаимосвязанных системных ресурсов на основе отдельного и независимого виртуального адресного пространства в контексте которой организуется выполнение потоков. Стандарт ISO 9000:2000 Definitions определяет процесс как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Процессы можно условно разбить на три категории:

- системные;

- фоновые;

- прикладные (пользовательские).

Системные процессы являются частью ядра ОС и всегда расположены в оперативной (основной) памяти. Выполняемые инструкции и данные этих процессов находятся в ядре системы, и поэтому они могут вызывать функции и обращаться к данным, недоступным для остальных процессов, например диспетчер страничного замещения, диспетчер памяти ядра, диспетчер буферного кэша и другие.

Фоновые процессы демоны – это неинтерактивные процессы, которые обычно запускаются при инициализации системы (после инициализации ядра) и обеспечивают работу различных подсистем. Например, системы терминального доступа, системы печати, системы сетевого доступа и другие. Эти процессы не связаны с пользовательскими сеансами работы и не могут непосредственно управляться пользователями. Прикладные процессы, как правило, порождаются в рамках пользовательского сеанса. Они могут выполняться как в интерактивном, так и в фоновом режимах.

Большинство процессоров поддерживают два режима работы: привилегированный, или режим ядра, и пользовательский, или режим задачи. Определенные команды выполняются только в привилегированном режиме, например команды управления памятью и вводом-выводом. Режим работы устанавливается в регистре слова состояния процессора (PSW) битом режима выполнения, который может быть изменен при наступлении некоторых событий. Например, если в результате прерывания управление пользовательским процессом переходит к процедуре ОС, данная процедура изменяет режим выполнения на привилегированный. Перед возвращением управления пользовательскому процессу режим выполнения изменяется обратно на пользовательский. Программы, выполняющиеся в режиме ядра, обладают полным контролем над процессором и имеют доступ ко всем ячейкам памяти.

**4. Модель процессов**

Можно рассматривать все функционирующее на компьютере программное обеспечение, включая операционную систему, в виде набора процессов. Каждый процесс можно описать набором параметров, включая текущие значения счетчика команд, регистров и переменных. С позиций данной абстрактной модели у каждого процесса есть собственный виртуальный центральный процессор. В действительности реальный процессор переключается с одного процесса на другой.

Для реализации модели процессов операционная система содержит системную таблицу процессов с одним элементом (дескриптором) для каждого процесса. Данный элемент называется блоком управления или дескриптором процесса. В дескрипторе процесса прямо или косвенно содержится информация, необходимая ядру системы.

Каждому процессу операционной системой выделяется виртуальное адресное пространство, представляющее собой набор виртуальных адресов, необходимых для выполнения процесса. Для прикладных программ эти адреса первоначально назначаются транслятором при создании сегментов кода и данных. В ходе выполнения процесс может увеличить размер назначенного виртуального адресного пространства, запросив у ОС создания дополнительных сегментов или увеличения существующих. Максимальный размер виртуального адресного пространства ограничивается разрядностью адреса данной архитектуры компьютера. Содержимое назначенного процессу виртуального адресного пространства представляет собой образ процесса.

Выполнение процесса может происходить в двух режимах: в режиме ядра или режиме задачи. В режиме задачи процесс выполняет инструкции прикладной программы, допустимые на непривилегированном уровне защиты процессора. При этом процессу недоступны системные структуры данных. Для получения услуг ядра процессу необходимо сделать системный вызов, после чего выполнение процесса переходит на привилегированный уровень. Таким образом, ядро системы защищает собственное адресное пространство от доступа прикладного процесса, который может нарушить целостность структур данных ядра.

**5. Создание процессов**

Самыми первыми создаются процессы в момент загрузки ОС.  
 **1) Загрузка системы**  
При инициализации системы создаются несколько исходных процессов  
В Unix – это процессы «демоны» Sched(pid0) init(pid1) – другие высокоуровневые (веб-сервер, емейл-сервер). Ядро – не процесс. Идентификаторы в Unix идут последовательно с приращением 1.  
В Windows NT ядро – это системный процесс System(Pid4), далее загружаются система управления подсистемами smss. Идентификаторы в Win идут с приращением 4, идентификатор 0 зарезервирован, системный процесс — 4.  
  **2) Текущий процесс порождает дочерний процесс.**  
Есть интересная особенность в построении ОС — так называемая иерархия процессов, исторически она существовала в ОС класса Unix. Там каждый процесс имеет строгое родство – есть родительский процесс и может быть дочерний процесс.  
Пример веб-сервер может порождать дочерний процесс для обработки нового запроса(это не есть хорошо, работает медленно);  
В Unix процесс init отслеживает авторизации пользователя для того, чтобы запустить оболочку(новые процессы).  
  **3) Пользователь создает новый процесс**  
Пользователь вызывает команду из текстовой оболочки или запускает новую программу из графической оболочки. Это создает новый процесс, родитель которого – оболочка ОС.  
**Этапы создания процесса**Чтобы создать процесс надо:

1) Присвоить уникальный идентификатор новому процессу  
2) Выделить ему место в памяти (для программы, данных и стека) – физически в памяти выделяются некоторые страницы (создается образ процесса на диске)  
3) Инициализировать РСВ (блок управления процессом)  
4) Добавить процесс в очередь «готовых» к выполнению.

**6. Завершение процессов**

После создания процесс начинает работать и выполняет свою задачу. Но ничто не длится вечно, даже процессы. Рано или поздно новые процессы будут завершены, обычно в силу следующих обстоятельств:

* обычного выхода (добровольно);
* выхода при возникновении ошибки (добровольно);
* возникновения фатальной ошибки (принудительно);
* уничтожения другим процессом (принудительно).

Большинство процессов завершаются по окончании своей работы. Когда компилятор откомпилирует заданную ему программу, он осуществляет системный вызов, сообщающий операционной системе о завершении своей работы. Этим вызовом в UNIX является exit, а в Windows — ExitProcess. Программы, работающие с экраном, также поддерживают добровольное завершение. Текстовые процессоры, интернет-браузеры и аналогичные программы всегда содержат значок или пункт меню, на котором пользователь может щелкнуть, чтобы приказать процессу удалить все временные файлы, которые им были открыты, и завершить свою работу.

Вторая причина завершения — обнаружение процессом фатальной ошибки. Например, если пользователь наберет команду cc foo.c

с целью компиляции программы foo.c, а файла с таким именем не будет, то произойдет простое объявление о наличии данного факта и выход из компилятора. Выхода из интерактивных, использующих экран процессов при задании им неверных параметров обычно не происходит. Вместо этого появляется диалоговое окно с просьбой о повторной попытке ввода параметров.

Третья причина завершения — ошибка, вызванная самим процессом, чаще всего связанная с ошибкой в программе. В качестве примеров можно привести неверную инструкцию, ссылку на несуществующий адрес памяти или деление на нуль. В некоторых системах (например, UNIX) процесс может сообщить операционной системе о своем намерении обработать конкретные ошибки самостоятельно, в таком случае, когда встречается одна из таких ошибок, процесс получает сигнал (прерывается), а не завершается.

Четвертая причина, из-за которой процесс может быть завершен, — это выполнение процессом системного вызова, приказывающего операционной системе завершить некоторые другие процессы. В UNIX этот вызов называется kill. Соответствующая функция Win32 называется TerminateProcess. В обоих случаях у процесса, вызывающего завершение, должны быть на это соответствующие полномочия. В некоторых системах при добровольном или принудительном завершении процесса тут же завершаются и все созданные им процессы. Но ни UNIX, ни Windows так не делают

**7. Иерархии процессов**

В некоторых системах, когда процесс порождает другой процесс, родительский и дочерний процессы продолжают оставаться определенным образом связанными друг с другом. Дочерний процесс может и сам создать какие-нибудь процессы, формируя иерархию процессов. Следует заметить, что, в отличие от растений и животных, использующих половую репродукцию, у процесса есть только один родитель (но нуль, один, два или более детей).

В UNIX процесс, все его дочерние процессы и более отдаленные потомки образуют группу процессов. Когда пользователь отправляет сигнал с клавиатуры, тот достигает всех участников этой группы процессов, связанных на тот момент времени с клавиатурой (обычно это все действующие процессы, которые были созданы в текущем окне). Каждый процесс по отдельности может захватить сигнал, игнорировать его или совершить действие по умолчанию, которое должно быть уничтожено сигналом.

В качестве другого примера, поясняющего ту ключевую роль, которую играет иерархия процессов, давайте рассмотрим, как UNIX инициализирует саму себя при запуске сразу же после начальной загрузки компьютера. В загрузочном образе присутствует специальный процесс, называемый init. В начале своей работы он считывает файл, сообщающий о количестве терминалов. Затем он разветвляется, порождая по одному процессу на каждый терминал. Эти процессы ждут, пока кто-нибудь не зарегистрируется в системе. Если регистрация проходит успешно, процесс регистрации порождает оболочку для приема команд. Эти команды могут породить другие процессы и т. д. Таким образом, все процессы во всей системе принадлежат единому дереву, в корне которого находится процесс init

В отличие от этого в Windows не существует понятия иерархии процессов, и все процессы являются равнозначными. Единственным намеком на иерархию процессов можно считать присвоение родительскому процессу, создающему новый процесс, специального маркера (называемого дескриптором), который может им использоваться для управления дочерним процессом. Но он может свободно передавать этот маркер какому-нибудь другому процессу, нарушая тем самым иерархию. А в UNIX процессы не могут лишаться наследственной связи со своими дочерними процессами.

**8. Состояния процессов**

Любой процесс в многозадачной ОС многократно испытывает переход из одного состояния в другое.

Основных состояний всего три.

* Работа (running) – в этом состоянии находится процесс, программу которого в данный момент выполняет процессор. Работающий процесс иногда удобно называть также текущим процессом.
* Готовность (ready) – состояние, их которого процесс может быть переведен в состояние работы, как только это сочтет нужным сделать ОС.
* Блокировка или, что то же самое, сон (sleeping, waiting) – состояние, в котором процесс не может продолжать выполнение, пока не произойдет некоторое внешнее по отношению к процессу событие.

Первые два состояния часто объединяют понятием активного состояния процесса.

Готовый к выполнению процесс не выполняется только потому, что есть другие не менее готовые процессы, по мнению системы более достойные занимать сейчас процессорное время. В каждый момент времени выбор одного из готовых процессов на роль работающего определяется логикой работы ОС.

В отличие от этого, спящий процесс – это всегда процесс, ожидающий некоторого конкретного события. Спящий процесс не сможет заработать, даже если процессор вдруг окажется свободным. Такой процесс, в соответствии со своей собственной логикой, ждет чего-то, что должно произойти.

* завершения начатой операции синхронного ввода/вывода;
* освобождения запрошенного у системы ресурса;
* истечения заданного интервала времени или достижения заданного момента времени;
* сигнала на продолжение действий от другого, взаимосвязанного процесса;
* сообщения от системы о необходимости выполнить определенные действия (например, перерисовать содержимое окна).

В любом из названных (и многих неназванных) случаев должно произойти некоторое событие, источник которого лежит вне данного процесса.

Любая многозадачная ОС предоставляет в распоряжение прикладных программ набор функций, переводящих вызвавший их процесс в состояние сна, в котором процесс не пытается использовать процессорное время. Такие системные функции называются блокирующими. К их числу относятся функции синхронного ввода/вывода, запроса ресурсов, приостановки до заданного времени, получения сообщений и многие другие.

Изображение выглядит как текст

Автоматически созданное описаниеПоскольку ОС берет на себя блокировку, «усыпление» процесса, она должна обеспечить и его разблокировку, «пробуждение». Чтобы это стало возможным, система должна для каждого спящего процесса помнить условия пробуждения процесса. Система отслеживает все события, способные разблокировать какой-либо процесс и, когда для одного или сразу нескольких процессов наступает ожидаемое событие, переводит эти события из состояния сна в состояние готовности.

**9. Реализация процессов**

Для реализации модели процессов операционная система ведет таблицу (состоящую из массива структур), называемую таблицей процессов, в которой каждая запись соответствует какому-нибудь процессу. Эти записи содержат важную информацию о состоянии процесса, включая счетчик команд, указатель стека, распределение памяти, состояние открытых им файлов, его учетную и планировочную информацию и все остальное, касающееся процесса, что должно быть сохранено, когда процесс переключается из состояния выполнения в состояние готовности или блокировки, чтобы позже он мог возобновить выполнение, как будто никогда не останавливался.

Существует область памяти (обычно это фиксированная область в нижних адресах), связанная с каждым классом устройств ввода-вывода, которая называется вектором прерывания. В ней содержится адрес процедуры, обслуживающей прерывание.

Все прерывания сначала сохраняют состояния регистров. Затем информация, помещенная в стек прерыванием, удаляется и указатель стека переустанавливается на временный стек, используемый обработчиком прерывания. Такие действия, как сохранение регистров и переустановка указателя стека, не могут быть выражены на языках высокого уровня, поэтому они выполняются небольшой подпрограммой на языке ассемблера.

Когда эта подпрограмма завершает свою работу, она вызывает процедуру, которая делает всю остальную работу для данного конкретного типа прерывания. Возможно, когда работа этой процедуры будет завершена, какой-нибудь процесс переходит в состояние готовности к работе и вызывается планировщик, чтобы определить, какой процесс будет выполняться следующим. После этого управление передается обратно коду, написанному на языке ассемблера, чтобы он загрузил для нового текущего процесса регистры и карту памяти и запустил выполнение этого процесса.

Процесс во время своего выполнения может быть прерван тысячи раз, но ключевая идея состоит в том, что после каждого прерывания прерванный процесс возвращается в точности к такому же состоянию, в котором он был до того, как случилось прерывание.

**10. Программные потоки**

В традиционных операционных системах у каждого процесса есть адресное пространство и один поток управления. Нередко возникают ситуации, в которых желательно иметь несколько потоков управления, квазипараллельно выполняющихся в одном адресном пространстве так, как будто они представляют собой отдельные процессы. Такие потоки управления обычно называются программными потоками, или легковесными процессами.

Программный поток имеет счетчик команд, указывающий на следующую исполняемую команду, регистры, содержащие рабочие переменные, а также стек, в котором хранится последовательность исполнения. Программные потоки дополняют модель процесса возможностью параллельного исполнения в рамках одного процессного окружения с высокой степенью независимости.

В качестве примера приложения, рассчитанного на многопоточность, рассмотрим веб-браузер. Веб-страницы зачастую содержат множество картинок небольшого размера. Для воспроизведения каждой картинки браузер должен установить отдельное соединение с сайтом, которому принадлежит страница, и послать запрос на ее получение. При поддержке браузером многопоточности можно загружать несколько картинок одновременно, что значительно ускоряет загрузку страницы, поскольку при небольшом размере большинства изображений установка соединений отнимает больше времени, чем передача данных.

Хотя программные потоки часто бывают полезными, они существенно усложняют программную модель и возникают некоторые проблемы. Например:

- программные потоки совместно используют большое количество структур данных. Что произойдет, если один программный поток закроет файл в то время, когда другой считывает из него данные?

- управление стеками. Во многих системах при переполнении стека процесса ядро автоматически увеличивает его. Если у процесса несколько программных потоков, стеков тоже должно быть несколько. Если ядро не знает о существовании этих стеков, оно не может их автоматически наращивать при переполнении. Ядро может даже не связать ошибки памяти с переполнением стеков.

Разумеется, эти проблемы преодолимы, но на их примере хорошо видно, что введение программных потоков в существующую систему без тщательной и продуманной реконструкции всей системы не имеет смысла.

**11. Взаимодействие между процессами. Гонки. Критические секции. Взаимное исключение с активным ожиданием. Запрет на прерывания. Переменные блокировки. Строгое чередование. Алгоритм Петерсона. Команда TSL**

***Взаимодействие между процессами***

Довольно часто процессам необходимо взаимодействовать с другими процессами. Например, в канале оболочки выходные данные одного процесса могут передаваться другому процессу, и так далее вниз по цепочке. Поэтому возникает необходимость во взаимодействии процессов.

В некоторых операционных системах совместно работающие процессы могут использовать какое-нибудь общее хранилище данных, доступное каждому из них по чтению и по записи. Это общее хранилище может размещаться в оперативной памяти или может быть представлено каким-нибудь общим файлом. Расположение общей памяти не меняет характера взаимодействия и возникающих при этом проблем.

Ситуация, когда два или более процесса считывают или записывают какие-нибудь общие данные, а окончательный результат зависит от того, какой процесс и когда именно выполняется, называется состязательной ситуацией или же гонкой.

Иногда процесс вынужден обращаться к общей памяти или файлам либо совершать какие-нибудь другие значимые действия, приводящие к состязаниям. Та часть программы, в которой используется доступ к общей памяти, называется критической областью или критической секцией. Если никакие два процесса не находились одновременно в своих критических областях, это позволило бы избежать состязаний.

Для решения проблемы состязаний есть несколько вариантов:

1. Взаимное исключение с активным ожиданием. В однопроцессорных системах простейшим решением подобной проблемы является запрещение всех прерываний каждым процессом сразу после входа в критическую область и их разрешение сразу же после выхода из критической области. Запрещение прерываний в большинстве своем является полезной технологией внутри самой операционной системы, но не подходит в качестве универсального механизма взаимных блокировок для пользовательских процессов.
2. Запрет на прерывания. Программное решение, в котором используется одна общая (блокирующая) переменная, исходное значение которой равно нулю. Когда процессу требуется войти в свою критическую область, сначала он проверяет значение блокирующей переменной. Если оно равно 0, процесс устанавливает его в 1 и входит в критическую область. Если значение уже равно 1, процесс просто ждет, пока оно не станет равно нулю. Таким образом, нулевое значение показывает, что ни один из процессов не находится в своей критической области, а единица — что какой-то процесс находится в своей критической области. Состязание возникнет в том случае, если второй процесс изменит значение блокирующей переменной сразу же после того, как первый процесс завершит повторную проверку ее значения.
3. Строгое чередование. Этот метод требует, чтобы два процесса попадали в критические секции строго по очереди. Ни один из них не сможет войти в критическую секцию два раза подряд. Хотя этот алгоритм и исключает условия гонок, его нельзя рассматривать всерьез, поскольку он нарушает третье правило успешной работы двух параллельных процессов с общими данными.
4. Алгоритм Петерсона. Перед тем, как начать исполнение [критической секции](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) кода, поток должен вызвать специальную процедуру со своим номером в качестве параметра. Она должна организовать ожидание потоком своей очереди входа в критическую секцию. После исполнения критической секции и выхода из неё поток вызывает другую процедуру, после чего уже другие потоки смогут войти в критическую область.
5. Команда TSL. Она считывает содержимое слова памяти LOCK в регистр RX, а затем сохраняет по адресу LOCK ненулевое значение. Операция чтения слова и сохранения в него значения гарантированно является неделимой — ни один другой процесс не может получить доступ к слову до окончания исполнения команды. Процессор, исполняющий команду TSL, блокирует для этой цели шину памяти.

**12. Примитивы взаимодействия между процессами. Семафоры. Мьютексы. Мониторы**

Вводится понятия двух примитивов.

**sleep**- системный запрос, в результате которого вызывающий процесс блокируется,пока его не запустит другой процесс.

**wakeup**- системный запрос, в результате которого блокированный процесс будет запущен.

Основное преимущество - отсутствие активного ожидания.

Проблема заключается в следующем, если спулер пуст, то wakeup срабатывает в пустую.

***Проблема переполненного буфера (проблема производителя и потребителя)***

Рассмотрим два процесса, которые совместно используют буфер ограниченного размера, один процесс пишет в буфер, другой считывает данные.

Чтобы первый процесс не писал, когда буфер полный, а второй не считывал, когда он пуст, вводится переменная *count* для подсчета количества элементов в буфере.

В этой ситуации оба процесса могут попасть в состояние ожидания, если пропадет сигнал активации.

Алгоритм такой ситуации:

1. Процесс **В**, считал count=0 (заблокироваться он еще не успел)
2. Планировщик передал управление процессу **А**
3. Процесс **А**, выполнил все вплоть до wakeup, пытаясь разблокировать процесс**В**(но он не заблокирован, wakeup срабатывает впустую)
4. Планировщик передал управление процессу **В**
5. И он заблокировался, и больше сигнала на разблокировку не получит
6. Процесс **А**в конце концов заполнит буфер и заблокируется, но сигнала на разблокировку не получит.

**Семафоры** - переменные для подсчета сигналов запуска, сохраненных на будущее.

Были предложены две операции **down** и **up** (аналоги sleep и wakeup).

Прежде чем заблокировать процесс down проверяет семафор, если он равен нулю, то он блокирует процесс, если нет, то процесс снова становится активным, и уменьшает семафор на единицу.

**Up** увеличит значение семафора на 1 или разблокирует процесс, находящийся в ожидании, **down**уменьшает значение семафора на 1 или блокирует процесс, если семафор = 0.

down и up выполняются как **элементарное действие**, т.е. процесс не может быть блокирован во время выполнения этих операций. Значит, у операционной системы должен быть запрет на все прерывания, и перевод процесса в режим ожидания.

***Решение проблемы переполненного буфера с помощью семафора***

Применим три семафора:

full - подсчет заполненных сегментов (в начале = 0)

empty - подсчет пустых сегментов (в начале = количеству сегментов)

mutex - для исключения одновременного доступа к буферу двух процессов.  (в начале = 1)

Мьютекс - упрощенная версия семафора, он управляет доступом к ресурсу. Показывает, блокирован или нет ресурс.

***Применение семафоров для устройств ввода/вывода***

Для устройств ввода/вывода семафор выставляется равный нулю. После запуска управляющего процесса выполняется down процесс блокируется. Когда нужно активизировать процесс управления, выполняется up.

**Монитор**— механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для обеспечения доступа к неразделяемым ресурсам.В простейшем случае монитор состоит из мьютекса и набора процедур, взаимодействующих с общим ресурсом.

**13. Передача сообщений. Разработка систем передачи сообщений. Решение проблемы производителя и потребителя путем передачи сообщений**

Передача сообщений. Межпроцессное взаимо­действие такого рода строится на двух примитивах: send и receive, которые ско­рее являются системными вызовами, нежели структурными компонентами язы­ка (что отличает их от мониторов и делает похожим на семафоры). Поэтому их легко можно инкапсулировать в библиотечные процедуры, например:

*send(destination. &message):*

*receive(source. &message):*

Первый запрос посылает сообщение заданному адресату, а второй получает сообщение от указанного источника (или от любого источника, если это не име­ет значения). Если сообщения нет, второй запрос блокируется до поступления сообщения либо немедленно возвращает код ошибки.

С системами передачи сообщений связано большое количество сложных проблем и конструктивных вопросов, которых не возникает в случае семафоров и мони­торов. Особенно много сложностей появляется в случае взаимодействия процес­сов, проистекающих на различных компьютерах, соединенных сетью. Так, сооб­щение может затеряться в сети. Чтобы избежать потери сообщений, отправитель и получатель договариваются, что при получении сообщения получатель посы­лает обратно подтверждение приема. Если отправитель не получает подтвержде­ние через некоторое время, он отсылает сообщение еще раз. Если сообщение получено, но подтверждение до отправи­теля не дошло, то отправитель направит сообщение еще раз, и до получателя оно дойдет дважды. Крайне важно, чтобы получатель мог отличить копию преды­дущего сообщения от нового. Обычно проблема решается с помощью занесения порядкового номера сообщения в тело самого сообщения. Если к получателю приходит письмо с номером, совпадающим с номером предыдущего письма, письмо классифицируется как копия и игнорируется. Решение проблемы успеш­ного обмена информации в условиях ненадежной передачи сообщений составля­ет основу изучения компьютерных сетей. Для систем обмена сообщениями также важен вопрос названий процессов. Необходимо однозначно определять процесс, указанный в запросе send или receive. Кроме того, встает вопрос аутентификации: каким образом клиент мо­жет определить, что он взаимодействует с настоящим файловым сервером, а не с самозванцем?

Помимо этого существуют конструктивные проблемы, существенные при расположении отправителя и получателя на одном компьютере. Одной из таких проблем является производительность. Копирование сообщений из одного про­цесса в другой происходит гораздо медленнее, чем операция на семафоре или вход в монитор. Было проведено множество исследований с целью увеличения эффективности передачи сообщений.

Проблему потерянных сигналов запуска можно ре­шить с помощью семафоров. Стандартным способом является реализация операций down и up в виде системных запросов, с запретом операционной системой всех прерываний на период проверки семафора, изменения его значения и возможного перевода процесса в состояние ожидания. Поскольку для выполнения всех этих действий требуется всего лишь несколько команд процессора, запрет прерываний не приносит никакого вреда. Если используются несколько процессоров, каждый сема­фор необходимо защитить переменной блокировки посредством команды tsl, чтобы гарантировать одновременное обращение к семафору только одного про­цессора. Необходимо понимать, что использование команды tsl принципиально отличается от активного ожидания, при котором производитель или потреби­тель ждут наполнения или опустошения буфера. Операция с семафором займет несколько микросекунд, тогда как активное ожидание может затянуться на су­щественно больший промежуток времени.

**14. Классические проблемы взаимодействия между процессами. Проблема обедающих философов. Проблема читателей и писателей**

В литературе по ОС описано несколько классических проблем межпроцессного взаимодействия. Например, проблемы обедающих философов, читателей и писателей, а также - спящего брадобрея.

Рассмотрим *проблему обедающих философов:* пять философов сидят за круглым столом, у каждого есть тарелка со спагетти, которые настолько скользкие, что каждому философу нужно две вилки, чтобы с ними управиться. Между каждыми двумя тарелками лежит одна вилка. Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений. Когда философ голоден, он пытается получить 2 вилки, левую и правую, в любом порядке. Вопрос состоит в следующем: можно ли написать алгоритм, который моделирует эти действия для каждого философа и никогда не застревает.

Для решения этой проблемы используется массив для отслеживания душевного состояния каждого философа - ест, размышляет или голодает (пытаясь получить вилки). Философ может начать есть только, если ни один из его соседей не ест. В программе используется массив семафоров, по одному на каждого философа, чтобы блокировать голодных философов, если их вилки заняты.

Проблема обедающих философов полезна для моделирования процессов, соревнующихся за монопольный доступ к ограниченному количеству ресурсов, например, к устройствам ввода-вывода.

Другой известной задачей является *проблема читателей и писателей,*моделирующая доступ к базе данных. Представьте себе базу данных бронирования билетов на самолет, к которой пытается получить доступ множество процессов. Можно разрешить одновременное считывание данных из базы, но если процесс записывает информацию в базу, доступ остальных процессов должен быть прекращен, даже доступ на чтение.

Эта задача решается таким образом, что первый читающий процесс выполняет операцию *down* на семафоре, чтобы получить доступ к базе данных. Последующие читатели просто увеличивают значение счетчика. По мере ухода читателей из базы значение счетчика уменьшается, и последний читающий процесс выполняет на семафоре операцию *up,* позволяя блокированному пишущему процессу получить доступ к базе.

Здесь нужно иметь ввиду один момент. Пишущий процесс не получит доступ к базе, пока в ней есть хоть один читающий процесс. Однако любой читающий процесс будет иметь доступ на чтение. В результате возможна ситуация, при которой пишущий процесс никогда не сможет попасть в базу. Чтобы этого избежать, нужно сделать так, чтобы, как только появляется ожидающий доступа пишущий процесс, все вновь поступающие читающие процесс становились в очередь за ним.

**15. Планирование. Поведение процесса. Когда требуется планирование. Категории алгоритмов планирования. Цели алгоритма планирования**

Когда компьютер работает в многозадачном режиме, на нем зачастую запускается сразу несколько процессов и потоков, претендующих на использование центрального процессора. Если доступен только один центральный процессор, необходимо выбрать, какой из этих процессов будет выполняться следующим. Та часть операционной системы, на которую возложен этот выбор, называется планировщиком, а алгоритм, который ею используется, называется алгоритмом планирования.

Ранний алгоритм планирования был довольно прост: требовалось всего лишь запустить следующее имеющиеся на ленте задание. С появлением многозадачных систем алгоритм планирования усложнился, поскольку в этом случае обычно фигурировали сразу несколько пользователей, ожидавших обслуживания.

С появлением персональных компьютеров ситуация изменилась в двух направлениях. Во-первых, основная часть времени отводилась лишь одному активному процессу. Во-вторых, с годами компьютеры стали работать настолько быстрее, что центральный процессор практически перестал быть дефицитным ресурсом.

Когда же дело касается сетевых служб, ситуация существенно изменяется. Здесь в конкурентную борьбу за процессорное время вступает уже несколько процессов, поэтому планирование снова приобретает значение.

Планировщик, кроме выбора «правильного» процесса, должен также заботиться об эффективной загрузке центрального процессора. Сначала должно произойти переключение из пользовательского режима в режим ядра. Затем должно быть сохранено состояние текущего процесса, включая сохранение его регистров в таблице процессов для их последующей повторной загрузки. Во многих системах должна быть также сохранена и карта памяти. После этого запускается алгоритм планирования для выбора следующего процесса. Затем в соответствии с картой памяти нового процесса должен быть перезагружен блок управления памятью. И наконец, новый процесс должен быть запущен. Вдобавок ко всему перечисленному переключение процессов обесценивает весь кэш памяти, заставляя его дважды динамически перегружаться из оперативной памяти. В итоге слишком частое переключение может поглотить существенную долю процессорного времени.

***Категории алгоритмов планирования***

В различных условиях окружающей среды требуются разные алгоритмы планирования. Это обусловлено тем, что различные сферы приложений предназначены для решения разных задач. При этом стоит различать три среды:

1) пакетную;

2) интерактивную;

3) реального времени;

В пакетных системах не бывает пользователей, терпеливо ожидающих за своими терминалами быстрого ответа на свой короткий вопрос. Поэтому для них зачастую приемлемы неприоритетные алгоритмы или приоритетные алгоритмы с длительными периодами для каждого процесса.

В среде с пользователями, работающими в интерактивном режиме, приобретает важность приоритетность, сдерживающая отдельный процесс от захвата центрального процесса, лишающего при этом доступа к службе всех других процессов.

В системах, ограниченных условиями реального времени, приоритетность иногда не требуется, поскольку процессы знают, что они могут запускаться только на непродолжительные периоды времени, и зачастую выполняют свою работу довольно быстро, а затем блокируются.

***Задачи алгоритма планирования***

Вот некоторые задачи алгоритма планирования, которых следует придерживаться при различных обстоятельствах:

- все системы (равнодоступность, принуждение к определенной политике, баланс);

- пакетные системы (производительность, оборотное время, использование центрального процессора);

- интерактивные системы (время отклика, пропорциональность);

- системы реального времени (соблюдение предельных сроков, предсказуемость).

**16. Планирование в системах пакетной обработки**

1. Первым пришел - первым обслужен

Наипростейший из всех алгоритмов планирования неприоритетный алгоритм. При использовании этого алгоритма центральный процессор выделяется процессам в порядке поступления их запросов. Когда в систему попадает первое задание, оно тут же запускается на выполнение и получает возможность выполняться как угодно долго. Оно не прерывается по причине слишком долгого выполнения. По мере поступления других заданий они помещаются в конец очереди. При блокировке выполняемого процесса следующим запускается первый процесс, стоящий в очереди.

2. Сначала самое короткое задание

Теперь рассмотрим другой неприоритетный алгоритм для пакетных систем, в котором предполагается, что сроки выполнения заданий известны заранее. Когда в ожидании запуска во входящей очереди находится несколько равнозначных по важности заданий, планировщик выбирает сначала самое короткое задание.

3. Приоритет наименьшему времени выполнения

Приоритетной версией алгоритма выполнения первым самого короткого задания является алгоритм первоочередного выполнения задания с наименьшим оставшимся временем выполнения. При использовании этого алгоритма планировщик всегда выбирает процесс с самым коротким оставшимся временем выполнения. Время выполнения заданий нужно знать заранее. При поступлении нового задания выполняется сравнение общего времени его выполнения с оставшимися сроками выполнения текущих процессов. Если для выполнения нового задания требуется меньше времени, чем для завершения текущего процесса, этот процесс приостанавливается и запускается новое задание.

**17. Планирование в интерактивных системах**

При планировании в интерактивных системах (ПК, сервера и т.д.) применяются следующие алгоритмы планирования:

Циклическое планирование

Каждому процессу назначается определенный интервал времени, называемый его **квантом**, в течение которого ему предоставляется возможность выполнения. Если процесс к завершению кванта времени все еще выполняется, то ресурс центрального процессора у него отбирается и передается другому процессу. Если процесс переходит в заблокированное состояние или завершает свою работу до истечения кванта времени, то переключение центрального процессора на другой процесс происходит именно в этот момент.

При истечении кванта времени процесс помещается в конец списка процессов и выполнение передается следующему процессу после предыдущего.

Приоритетное планирование

Если определенным лицам необходим больший приоритет выполнения задач, чем другим, то используется **приоритетное планирование**. Основная идея проста: каждому процессу присваивается значение приоритетности и запускается тот процесс, который находится в состоянии готовности и имеет наивысший приоритет.Чтобы предотвратить бесконечное выполнение высокоприоритетных процессов, планировщик должен понижать уровень приоритета текущего выполняемого процесса с каждым сигналом таймера или же каждому процессу может быть выделен максимальный квант допустимого времени выполнения.

Использование нескольких очередей

Использование нескольких очередей подразумевает использование классов приоритетов. Процессы, относящиеся к наивысшему классу, запускались на 1 квант времени, процессы следующего по нисходящей класса — на 2 кванта времени, процессы следующего класса — на 4 кванта времени и т. д. Как только процесс использовал все выделенные ему кванты времени, его класс понижался.

Выбор следующего самого короткого процесса

Если выполнение каждой команды рассматривать как отдельное «задание», то мы можем минимизировать общее время отклика, запустив первой выполнение самой короткой команды. Проблема состоит в определении того, какой из находящихся в состоянии готовности процессов является самым коротким.

Лотерейное планирование

Основная идея алгоритма состоит в раздаче лотерейных билетов на доступ к различным системным ресурсам, в том числе к процессорному времени. Когда планировщику нужно принимать решение, в случайном порядке выбирается билет, и ресурс отдается процессу, обладающему этим билетом. Более важным процессам, чтобы повысить их шансы на выигрыш, могут выдаваться дополнительные билеты.

Справедливое планирование

При циклическом планировании, если пользователь 1 запускает 9 процессов, а пользователь 2 запускает 1 процесс, то при циклическом планировании или при равных приоритетах пользователь 1 получит 90% процессорного времени, а пользователь 2 – только 10%.

Чтобы избежать подобной ситуации, некоторые системы перед планированием работы процесса берут в расчет, кто является его владельцем. В этой модели каждому пользователю распределяется некоторая доля процессорного времени и планировщик выбирает процессы, соблюдая это распределение. Таким образом, если каждому из двух пользователей было обещано по 50 % процессорного времени, то они его получат, независимо от количества имеющихся у них процессов.

**18. Планирование в системах реального времени**

Системы реального времени относятся к тому разряду систем, в которых время играет очень важную роль. Обычно одно или несколько физических устройств, не имеющих отношения к компьютеру, генерируют входные сигналы, а компьютер в определенный промежуток времени должен соответствующим образом на них реагировать. К примеру, компьютер в проигрывателе компакт-дисков получает биты от привода и должен превращать их в музыку за очень короткий промежуток времени. Если вычисления занимают слишком много времени, музыка приобретет довольно странное звучание. Другими системами реального времени являются система отслеживания параметров пациента в палате интенсивной терапии, автопилот воздушного судна, устройство управления промышленными роботами на автоматизированном предприятии. Во всех этих случаях получение верного результата, но с запозданием, зачастую так же неприемлемо, как и неполучение его вообще.

Системы реального времени обычно делятся на жесткие системы реального времени (системы жесткого реального времени), в которых соблюдение крайних сроков обязательно, и гибкие системы реального времени (системы мягкого реального времени), в которых нерегулярные несоблюдения крайних сроков нежелательны, но вполне допустимы. В обоих случаях режим реального времени достигается за счет разделения программы на несколько процессов, поведение каждого из которых вполне предсказуемо и заранее известно. Эти процессы являются, как правило, быстротечными и способными успешно завершить свою работу за секунду. При обнаружении внешнего события планировщик должен так спланировать работу процессов, чтобы были соблюдены все крайние сроки. События, на которые должна реагировать система реального времени, могут быть определены как периодические (происходящие регулярно) или апериодические (происходящие непредсказуемо). Возможно, системе придется реагировать на несколько периодических потоковых событий. В зависимости от времени, необходимого на обработку каждого события, с обработкой всех событий система может даже не справиться. Например, если происходит m периодических событий, событие i возникает с периодом Pi и для обработки каждого события требуется Ci секунд процессорного времени, то поступающая информация может быть обработана только в том случае, если . Система реального времени, отвечающая этому критерию, называется планируемой. Это означает, что такая система фактически может быть реализована. Процесс, не отвечающий этому тесту, не может быть планируемым, поскольку общее время центрального процессора, требуемое процессу, в совокупности больше того времени, которое этот центральный процессор может предоставить.

Алгоритмы планирования работы систем реального времени могут быть статическими или динамическими. Первый из них предусматривает принятие решений по планированию еще до запуска системы, а второй — их принятие в реальном времени, после того как начнется выполнение программы. Статическое планирование работает только при условии предварительного обладания достоверной информацией о выполняемой работе и о крайних сроках, которые нужно соблюсти. Алгоритмы динамического планирования подобных ограничений не имеют.

**19. Планирование программных потоков**

Поток — это «управляющая последовательность внутри процесса» и наименьшая единица выполнения программы.У всех процессов есть хотя бы один поток

Рассмотрим алгоритм планирования системы Linux. Начнем с того, что потоки в системе Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах.

В операционной системе Linux алгоритмом планирования различаются три класса потоков:

1. Потоки реального времени, обслуживаемые по алгоритму FIFO (First in First Out — первым прибыл, первым обслужен).

2. Потоки реального времени, обслуживаемые в порядке циклической очереди.

3. Потоки разделения времени.

Обслуживаемые по алгоритму FIFO потоки реального времени имеют наивысшие приоритеты и не могут вытесняться другими потоками, за исключением такого же потока реального времени FIFO с более высоким приоритетом, перешедшего в состояние готовности. Обслуживаемые в порядке циклической очереди потоки реального времени представляют собой то же самое, что и потоки реального времени FIFO, но с тем отличием, что они имеют квант времени и могут вытесняться по таймеру. Такой находящийся в состоянии готовности поток выполняется в течение кванта времени, после чего помещается в конец своей очереди.

Ни один из этих классов не является на самом деле классом реального времени. Здесь нельзя задать предельный срок выполнения задания и гарантировать его выполнение. Эти классы просто имеют более высокий приоритет, чем потоки стандартного класса разделения времени.

Обычные потоки составляют отдельный класс и планируются по особому алгоритму, поэтому с потоками реального времени они не конкурируют. Так же, как и обслуживаемым в порядке циклической очереди потокам реального времени, Linux выделяет обычным потокам время центрального процессора в соответствии с их требованиями и уровнями приоритета. В Linux время измеряется количеством тиков. Чтобы избежать нерационального расхода циклов центрального процессора на обслуживание прерываний от таймера, ядро может быть даже сконфигурировано в режиме «без тиков», что может пригодиться при запуске в системе только одного процесса или простое центрального процессора и необходимости перехода в энергосберегающий режим.

**20. Начальная загрузка**

Процесс загрузки Linux представляет собой действия, посредством которых приводятся в состояние готовности операционные системы на основе Linux. Этот процесс во многом схож с загрузкой BSD и других Unix-подобных систем, от которых он и происходит.

При загрузке компьютера происходит последовательная передача управления от системной прошивки компьютера (BIOS или UEFI) к загрузчику, а от него — к ядру. Затем ядро запускает планировщик и выполняет программу init, после чего ядро переходит в состояние бездействия до тех пор, пока не получит внешний вызов.

Основные этапы загрузки:

1. Системная прошивка компьютера выполняет первичную проверку и инициализацию аппаратного обеспечения.
2. В случае BIOS прошивка загружает в оперативную память и выполняет загрузочный код с одного из разделов заданного загрузочного устройства, который содержит фазу 1 загрузчика Linux. Фаза 1 загружает фазу 2 (значительный по размеру код загрузчика). Некоторые загрузчики могут использовать для этого промежуточный этап (под названием фаза 1,5), поскольку современные диски большого объёма могут некорректно считываться без дальнейшего кода. В случае UEFI запускается загрузчик загруженный со служебного раздела (EFS), который выбирается согласно настройкам приоритета загрузки определенного в энергонезависимой памяти компьютера. При этом возможна загрузка не только специализированного загрузчика, но можно загрузить и непосредственно ядро Linux (для этого ядро должно быть собрано с опцией EFI\_STUB).
3. Загрузчик зачастую предлагает пользователю меню с доступными вариантами загрузки. После выбора или после заданного тайм-аута загрузчик загружает ядро.
4. Загруженное ядро распаковывается в памяти, настраивает системные функции, такие как работа необходимого оборудования и управление страницами памяти, после чего делает вызов start\_kernel().
5. После этого start\_kernel() выполняет основную настройку системы (прерывания, остальные функции управления памятью, инициализацию устройств, драйверов и т. д.), а потом порождает процесс бездействия, диспетчер и отдельно от них — процесс init (выполняющийся в пользовательском пространстве).
6. Планировщик начинает более эффективно управлять системой, в то время как ядро переходит к бездействию.
7. Процесс init выполняет необходимые сценарии, которые настраивают все службы и структуры, не относящиеся к уровню ядра, в результате чего будет создано пользовательское окружение, и пользователю будет предоставлен экран входа в систему.

Когда происходит завершение работы, init вызывается для управляемого закрытия программ пользовательского уровня, тоже согласно сценариям. После этого init закрывается, а ядро производит своё собственное завершение работы.

**21. Обработка прерываний**

Бывают случаи, когда вполне можно обойтись и программным вводом-выводом, однако все же для большинства операций ввода-вывода прерывания являются не чем иным, как суровой необходимостью. Они должны быть спрятаны как можно глубже в недрах операционной системы, чтобы о них было известно, как можно меньшей части этой системы. Лучший способ их спрятать - это заблокировать тот драйвер, который начал операцию ввода-вывода, до тех пор, пока не будет завершен ввод-вывод и не будет получено прерывание. Драйвер может заблокировать себя сам, выполнив, к примеру, процедуру down на семафоре, или процедуру wait на переменные состояния, или процедуру receive на сообщении, или еще что-либо подобное. Когда происходит прерывание, то все необходимое для его обработки делает процедура — обработчик прерывания. Затем она может разблокировать ожидавший данного прерывания драйвер. В ряде случаев она просто выполняет процедуру up на семафоре. В других случаях вызывает процедуру signal для переменной условия в мониторе. Во всех остальных случаях она посылает заблокированному драйверу сообщение. В любом случае заключительным действием прерывания будет предоставление возможности возобновления работы ранее заблокированному драйверу.  
Обработка прерывания не ограничивается только его перехватом, вызовом процедуры up для некого семафора, а затем выполнением команды IRET для возвращения из прерывания к предыдущему процессу. На операционную систему возлагается куда более существенный объем работы. Следует заметить, что подробности этой работы сильно зависят от конкретной операционной системы, поэтому некоторые из перечисленных далее шагов конкретной машине могут и не понадобиться, в отличие от шагов, не попавших в этот перечень.  
1. Сохранить все регистры (включая PSW), которые еще не были сохранены аппаратурой, вызвавшей прерывание.  
2. Установить контекст для процедуры обработки прерывания. Здесь может быть  
задействована установка TLB, MMU и таблицы страниц.  
3. Установить стек для процедуры обработки прерывания.  
4. Послать подтверждение контроллеру прерываний. В отсутствие централизованного контроллера прерываний разрешить прерывания.  
5. Скопировать регистры из того места, где они были сохранены (возможно, из какого-нибудь стека), в таблицу процессов.  
6. Запустить процедуру обработки прерывания, которая извлечет информацию из  
регистров контроллера устройства, вызвавшего прерывание.  
7. Выбрать следующий запускаемый процесс. Если прерывание привело к готовности  
какого-то ранее заблокированного процесса, имеющего высокий уровень приоритета, то теперь может быть выбран запуск именно этого процесса.  
8. Установить контекст MMU для следующего запускаемого процесса. Могут потребоваться и некоторые установки TLB.  
9. Загрузить регистры нового процесса, включая его PSW.  
10. Запустить выполнение нового процесса.

**22. Файлы. Именование файлов. Структура файла. Типы файлов. Доступ к файлам. Атрибуты файлов**

Файл является механизмом абстрагирования. Он предоставляет способ сохранения информации на диске и последующего ее считывания.

Когда процесс создает файл, он присваивает ему имя. Когда процесс завершается, файл продолжает существовать, и к нему по этому имени могут обращаться другие процессы. Некоторые файловые системы различают буквы верхнего и нижнего регистров, а некоторые не делают таких различий. Многие операционные системы поддерживают имена файлов, состоящие из двух частей, разделенных точкой.

***Структура файла***

Файлы могут быть структурированы несколькими различными способами. Чаще файл представляет собой бессистемную последовательность байтов. В сущности, операционной системе все равно, что содержится в этом файле, — она видит только байты.

В другой модели файл представляет собой последовательность записей фиксированной длины, каждая из которых имеет собственную внутреннюю структуру.

При третьей разновидности организации файл состоит из дерева записей, необязательно одинаковой длины, каждая из которых в конкретной позиции содержит ключевое поле. Дерево сортируется по ключевому полю, позволяя выполнять ускоренный поиск по конкретному ключу.

***Типы файлов***

Многие операционные системы поддерживают несколько типов файлов.

Обычными считаются файлы, содержащие информацию пользователя. Каталоги — это системные файлы, предназначенные для поддержки структуры файловой системы. Символьные специальные файлы имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, к которым относятся терминалы, принтеры и сети. Блочные специальные файлы используются для моделирования дисков.

**Доступ к файлам**

В самых первых операционных системах предоставлялся только один тип доступа к файлам — последовательный. В этих системах процесс мог читать все байты или записи файла только по порядку, с самого начала, но не мог перепрыгнуть и считать их вне порядка их следования.

Когда для хранения файлов стали использоваться диски, появилась возможность считывать байты или записи файла вне порядка их размещения или получать доступ к записям по ключу, а не по позиции. Файлы, в которых байты или записи могли быть считаны в любом порядке, стали называть файлами произвольного доступа.

***Атрибуты файлов***

У каждого файла есть свои имя и данные. Вдобавок к этому все операционные системы связывают с каждым файлом и другую информацию, к примеру дату и время последней модификации файла и его размер. В некоторых системах для доступа к файлу пользователь должен ввести пароль, в этом случае пароль может быть одним из атрибутов файла.

Атрибуты флаги представляют собой биты или небольшие поля, с помощью которых происходит управление некоторыми конкретными свойствами или разрешение их применения. Поля длины записи, позиции ключа и длины ключа имеются только у тех файлов, записи которых можно искать по ключу. Они предоставляют информацию, необходимую для поиска ключей. Различные показатели времени позволяют отслеживать время создания файла, последнего доступа к этому файлу, его последнего изменения.

**23. Операции с файлами. Каталоги. Пути. Каталоги в UNIX**

**Операции с файлами**

Файлы предназначены для хранения информации с возможностью ее последующего извлечения. Разные системы предоставляют различные операции, позволяющие сохранять и извлекать информацию. Например:

* *Create* (Создать). Создает файл без данных. Цель вызова состоит в объявлении о появлении нового файла и установке ряда атрибутов.
* *Delete* (Удалить). Позволяет удалить файл, чтобы освободить дисковое пространство.
* *Open* (Открыть).— дать возможность системе извлечь и поместить в оперативную память атрибуты и перечень адресов на диске, чтобы ускорить доступ к ним при последующих вызовах.

***Каталоги***

Обычно в файловой системе для упорядочения файлов имеются каталоги или папки, которые сами по себе являются файлами. Самая простая форма системы каталогов состоит из одного каталога, содержащего все файлы.

Для организации хорошей файловой системы нужна иерархия (то есть дерево каталогов). Такой подход позволяет иметь столько каталогов, сколько необходимо для группировки файлов естественным образом.

***Пути***

Каждый файл и справочник системы UNIX идентифицируется уникальным именем пути. Имя пути показывает местоположение файла или справочника и обеспечивает направление поиска его. Существует 2 типа имени пути: полное и родственное.

Полное имя пути (иногда называемое абсолютным именем пути) дает направление, которое начинается в справочнике root и показывает путь далее по уникальной последовательности справочников к конкретному справочнику или файлу. Вы можете использовать полное имя пути для поиска любого файла или справочника в системе UNIX.

Родственное имя пути дает направления, которые начинаются в вашем текущем рабочем справочнике и ведут вас вверх или вниз через серию справочников к конкретному файлу или справочнику. Двигаясь вниз из текущего справочника, вы можете получить доступ к своему файлу или справочнику. Двигаясь вверх из текущего каталога, вы пройдете через родительские справочники к родителю всех системных справочников, т.е. к root.

***Каталоги в UNIX***

В корневом каталоге находятся следующие подкаталоги:

"Базовые" каталоги, которые для правильного старта системы должны быть доступны в момент загрузки ядра:

*/etc* – конфигурационные файлы;

*/bin* – минимальный набор утилит пользователя, необходимых для запуска системы;

*/sbin* – минимальный набор утилит администратора, необходимых для запуска системы;

И т.д.

Виртуальные файловые системы в каталогах /proc и /sys предоставляют доступ к переменным ядра ОС, оформленным в виде виртуальных файлов.

Имя каталога /usr произошло от слова User.

В каталоге /usr находятся следующие подкаталоги:

*/usr/bin, /usr/sbin, /usr/lib* – по аналогии с каталогами в корне;

*/usr/include* – заголовочные файлы для компиляторов языков C и C++ ;

И др.

Каталог /var предназначен для хранения больших объемов данных. Иногда его располагают на отдельном большом разделе диска. В /var можно отметить следующие подкаталоги:

*/var/log* – лог-файлы;

*/var/mail* – почтовые ящики пользователей;

*/var/tmp* – общедоступный каталог для временных файлов.

**24. Реализация файлов. Неразрывные файлы. Связанные списки. Индексные узлы**

Файловые системы хранятся на дисках. Большинство дисков может быть разбито на один или несколько разделов, на каждом из которых будет независимая файловая система. Сектор 0 на диске называется главной загрузочной записью (MBR) и используется для загрузки компьютера. В конце MBR содержится таблица разделов. Из этой таблицы берутся начальные и конечные адреса каждого раздела. Один из разделов в этой таблице помечается как активный. При загрузке компьютера считывает и выполняет MBR. Первое, что делает программа MBR, — находит расположение активного раздела, считывает его первый блок, который называется загрузочным, и выполняет его.  
Непрерывное размещение  
У непрерывного распределения дискового пространства есть два существенных преимущества. Во-первых, его просто реализовать, поскольку отслеживание местонахождения принадлежащих файлу блоков сводится всего лишь к запоминанию двух чисел: дискового адреса первого блока и количества блоков в файле. Во-вторых, у него превосходная производительность считывания, поскольку весь файл может быть считан с диска за одну операцию. Для нее потребуется только одна операция позиционирования. Непрерывное размещение характеризуется простотой реализации и высокой производительностью. К сожалению, у непрерывного размещения есть также очень серьезный недостаток: со временем диск становится фрагментированным. Со временем диск заполнится и понадобится либо его уплотнить, что является слишком затратной операцией, либо повторно использовать последовательности свободных блоков между файлами. При создании нового файла необходимо знать его окончательный размер, чтобы выбрать подходящий для размещения участок.  
Размещение с использованием связанного списка  
Второй метод хранения файлов заключается в представлении каждого файла в виде связанного списка дисковых блоков. Первое слово каждого блока используется в качестве указателя на следующий блок, а вся остальная часть блока предназначается для хранения данных. В отличие от непрерывного размещения, в этом методе может быть использован каждый дисковый блок. При этом потери дискового пространства на фрагментацию отсутствуют. Кроме того, достаточно, чтобы в записи каталога хранился только дисковый адрес первого блока. Всю остальную информацию можно найти, начиная с этого блока.  
Индексные узлы  
Последним из рассматриваемых методов отслеживания принадлежности конкретного блока конкретному файлу является связь с каждым файлом структуры данных, называемой i-узлом, содержащей атрибуты файла и дисковые адреса его блоков. При использовании i-узла появляется возможность найти все блоки файла. Большим преимуществом этой схемы перед связанными списками, использующими таблицу в памяти, является то, что i-узел должен быть в памяти только в том случае, когда открыт соответствующий файл. Заранее нужно будет зарезервировать только этот объем памяти. Обычно этот массив значительно меньше того пространства, которое занимает таблица расположения файлов, рассмотренная в предыдущем разделе. Причина проста. Таблица, предназначенная для хранения списка всех дисковых блоков, пропорциональна размеру самого диска.

**25. Организация дискового пространства. Размер блока. Учет свободных блоков. Надежность файловой системы. Непротиворечивость файловой системы. Резервные копии**

Для хранения файла размером n байт возможно использование двух стратегий: выделение на диске n последовательных байтов или разбиение файла на несколько непрерывных блоков. При использовании первой стратегии возникает дилемма: по мере увеличение размера файла потребуется его перемещение на новое место на диске. По этой причине почти все файловые системы разбивают файлы на блоки фиксированного размера, которые не нуждаются в смежном расположении.

В вопросе отслеживания свободных блоков памяти также существует два подхода. Первый состоит в использовании связанного списка дисковых блоков. Второй подход управления свободным дисковым пространством использует битовый массив. Для диска, имеющего *n* блоков, требуется битовый массив, состоящий из *n* битов. Свободные блоки представлены в массиве в виде единиц, а распределенные — в виде нулей (или наоборот).

Также существует проблема, что у жестких дисков часто бывают врожденные дефектные блоки. Решение сводится к тому, что файловая система аккуратно составляет файл, содержащий в себе все сбойные блоки. Благодаря такому подходу блоки исключаются из списка свободных и никогда не задействуются для хранения данных. Если не делать к данному файлу обращений, никаких проблем (за исключением особых случаев) не возникнет.

Еще одним аспектом проблемы надежности является непротиворечивость файловой системы. Файловые системы обычно читают блоки данных, модифицируют их и записывают обратно. Если в системе произойдет сбой прежде, чем все модифицированные блоки будут записаны на диск, файловая система может оказаться в противоречивом состоянии. Обычно контролируется непротиворечивость объектов двух типов: блоков и файлов. При проверке непротиворечивости блоков программа создает две таблицы, каждая из которых содержит счетчик для каждого блока, изначально установленный в 0. Счетчики в первой таблице фиксируют, в каком количестве каждый блок присутствует в файле. Счетчики во второй таблице показывает, сколько раз каждый блок учитывается в списке свободных блоков. Затем программа считывает все индексные узлы. Начиная с индексного узла, можно построить список всех номеров блоков, занятых соответствующим файлом. При считывании каждого номера блока счетчик этого блока увеличивается на единицу. Затем программа анализирует список или битовую карту свободных блоков, чтобы обнаружить все неиспользуемые блоки. Каждый раз, встречая номер блока в списке свободных блоков, программа инкрементирует соответствующий счетчик во второй таблице. Если файловая система непротиворечива, каждый блок будет встречаться только один раз либо в первой, либо во второй таблице. При проверке непротиворечивости файловой системы кроме блоков программа проверяет также и систему каталогов.

Резервные копии нужны для двух случаев: восстановление после какой-либо аварии и восстановление после ошибок пользователей. Для резервного копирования диска можно воспользоваться одной из двух стратегий: физической или логической архивацией. Физическая архивацияведется с нулевого блока диска, при этом все дисковые блоки записываются на лету в порядке их следования и, когда скопирован последний блок, запись останавливается. Логическая архивацияначинается в одном или нескольких указанных каталогах и рекурсивно архивирует все найденные там файлы и каталоги, в которых произошли изменения со времени какой-нибудь заданной исходной даты.

**26. Производительность файловой системы. Кэширование. Упреждающее чтение блоков. Уменьшение количества перемещений головки диска**

Доступ к диску осуществляется намного медленнее, чем доступ к оперативной памяти. Если нужно считать только одно слово, то доступ к памяти примерно в миллион раз быстрее, чем доступ к диску. В результате такой разницы во времени доступа во многих файловых системах применяются различные усовершенствования, предназначенные для повышения производительности.  
Наиболее распространенным методом сокращения количества обращений к диску является блочное кэширование или буферное кэширование. Кэш представляет собой коллекцию блоков, логически принадлежащих диску, но хранящихся в памяти с целью повышения производительности. Для управления кэшем могут применяться различные алгоритмы, но наиболее распространенный из них предусматривает проверку всех запросов для определения того, имеются ли нужные блоки в кэше. Последующий запрос к тому же самому блоку может быть удовлетворен непосредственно из кэша. Теперь возникает проблема с записыванием изменённых блоков кэша обратно на диск. В UNIX используется системный вызов sync, вынуждающий немедленно записать все измененные блоки на диск. При запуске системы в фоновом режиме начинает действовать программа, которая обычно называется update. Она работает в бесконечном цикле и осуществляет вызовы sync. В Windows практикуется другой подход, состоящий в том, что каждый модифицированный блок записывается на диск сразу же. Кэш, в котором все модифицированные блоки немедленно записываются на диск, называется сквозным кэшем, или кэшем со сквозной записью. При использовании сквозного кэша количество обращений ввода-вывода к диску больше, чем при применении обычного кэша. В большинстве программ применяется внутренняя буферизация, поэтому обычно эти программы обращаются к системному вызову write не с одним символом, а с целыми строками или большими единицами данных.  
Второй метод повышения производительности файловой системы — помещение блоков в кэш до того, как они оказываются нужными. Это увеличивает процент кэш-попаданий. В частности, многие файлы считываются последовательно. Когда файловой системе поступает запрос на блок k файла, она удовлетворяет его, однако затем скрытно проверяет в кэше наличие блока k + 1. Если блок отсутствует, он считывается в кэш в надежде на то, что, когда он понадобится, его присутствие будет обеспечено.  
Кэширование и упреждающее чтение не являются единственными способами повышения производительности системы. Другой важный метод состоит в уменьшении затрат времени на перемещение блока головок. Когда записывается выходной файл, файловая система должна зарезервировать место для чтения таких блоков за одну операцию. Даже при использовании списка свободных блоков может быть выполнена определенная кластеризация данных. Если сектор состоит из 512 байт, система может использовать блоки размером в 1 Кбайт (два сектора), но выделять пространство на диске в единицах по два блока (четыре сектора). Это не то же самое, что использовать 2-килобайтные дисковые блоки, так как кэш рассчитан на килобайтные блоки, и дисковые операции чтения и записи будут по-прежнему требовать килобайтных блоков. Однако при последовательном чтении файла количество операций поиска цилиндра уменьшится вдвое, что значительно повысит производительность.

**27. Файловые системы с журнальной структурой**

В основу была положена идея структурировать весь диск в виде очень большого журнала. Периодически, когда в этом возникает особая надобность, все ожидающие осуществления записи, находящиеся в буфере памяти, собираются в один непрерывного сегмент и в таком виде записываются на диск в конец журнала. Таким образом, отдельный сегмент может вперемешку содержать i-узлы, блоки каталога и блоки данных. В начале каждого сегмента находится сводная информация, в которой сообщается, что может быть найдено в этом сегменте. Если средний размер сегмента сможет быть доведен примерно до 1 Мбайт, то будет использоваться практически вся пропускная способность диска. В этой конструкции по-прежнему используются i-узлы той же структуры, что и в UNIX, но теперь они не размещаются в фиксированной позиции на диске, а разбросаны по всему журналу. Тем не менее, когда определено местоположение i-узла, местоположение блоков определяется обычным образом. Для поиска i-узлов ведется массив i-узлов, проиндексированный по i-номерам. Элемент i в таком массиве указывает на i-узел на диске. Массив хранится на диске, но также подвергается кэшированию, поэтому наиболее интенсивно использующиеся фрагменты большую часть времени будут находиться в памяти.  
Управление использованием блоков на диске в этой системе имеет необычный характер, поскольку, когда файловый блок опять записывается на диск в новый сегмент, должен быть найден i-узел файла, после чего он должен быть обновлен и помещен в память для записи в следующий сегмент. Затем должен быть обновлен массив i-узлов, чтобы в нем присутствовал указатель на новую копию.  
Основной принцип заключается в журналировании всех намерений файловой системы перед их осуществлением. Поэтому, если система терпит аварию еще до того, как у нее появляется возможность выполнить запланированные действия, после перезагрузки она может посмотреть в журнал, определить, что она собиралась сделать на момент аварии, и завершить свою работу. Файловые системы, которые называются журналируемыми файловыми системами, нашли свое применение.  
При журналировании все операции должны быть идемпотентными, что означает возможность их повторения необходимое число раз без нанесения какого-либо вреда. Такие операции, как «обновить битовый массив, пометив i-узел k или блок n свободными», могут повторяться до тех пор, пока все не завершится должным и вполне безопасным образом. В то же время добавление только что освободившихся блоков из i-узла k к концу перечня свободных блоков не является идемпонентным действием, поскольку они уже могут присутствовать в этом перечне. Более ресурсоемкая операция «просмотреть перечень свободных блоков и добавить к нему блок n, если он в нем отсутствовал» является идемпонентной.

**28. Устройства ввода-вывода. Контроллеры устройств. Ввод-вывод с отображением на память. Прерывания. Прямой доступ к памяти**

***Устройства ввода-вывода***

Устройства ввода-вывода можно условно разделить на две категории: блочные устройства и символьные устройства. К блочным относятся такие устройства, которые хранят информацию в блоках фиксированной длины, у каждого из которых есть собственный адрес. Вся передача данных ведется пакетами из одного или нескольких целых (последовательных) блоков. Важным свойством блочного устройства является то, что оно способно читать или записывать каждый блок независимо от всех других блоков.

Другой тип устройств ввода-вывода — символьные устройства. Они выдают или воспринимают поток символов, не относящийся ни к какой блочной структуре. Они не являются адресуемыми и не имеют никакой операции позиционирования. В качестве символьных устройств могут рассматриваться принтеры, сетевые интерфейсы, мыши и множество других устройств, не похожих на дисковые устройства.

***Контроллеры устройств***

Электронный компонент называется контроллером устройства, или адаптером. На персональных компьютерах он часто присутствует в виде микросхемы на системной плате или печатной платы, вставляемой в слот расширения. Многие контроллеры способны управлять двумя, четырьмя или даже восемью одинаковыми устройствами. Задача контроллера состоит в преобразовании последовательного потока битов в блок байтов и коррекции ошибок в случае необходимости.

У каждого контроллера для связи с центральным процессором имеется несколько регистров. Путем записи в эти регистры операционная система может давать устройству команды на предоставление данных, принятие данных, включение, выключение или выполнение каких-нибудь других действий. В дополнение к регистрам управления у многих устройств имеется буфер данных, из которого операционная система может считывать данные и в который она может их записывать.

Центральный процессор может запрашивать данные у контроллера ввода-вывода побайтно, но при этом будет нерационально расходоваться его рабочее время, поэтому чаще всего используется другая схема, которая называется прямым доступом к памяти (DMA). Операционная система может использовать DMA только при наличии аппаратного DMA-контроллера, присутствующего у большинства систем. Для упорядочения обмена данными с несколькими устройствами, проводимого нередко в параллельном режиме, доступен только один DMA-контроллер.

***Прерывания***

Когда устройство ввода-вывода завершает порученную ему работу, оно инициирует прерывание. Это делается путем выставления сигнала на специально выделенной линии шины. Если не было никаких других отложенных прерываний, контроллер прерываний немедленно обрабатывает инициированное прерывание. Но если он находится в процессе обработки другого прерывания или какие-нибудь другое устройство в то же самое время выставило на шине запрос на прерывание более высокого приоритетного уровня, то устройство на некоторое время просто игнорируется. В таком случае оно продолжает выставлять на шину сигнал на прерывание до тех пор, пока оно не будет обслужено центральным процессором.

**29. Программное обеспечение ввода-вывода. Обработчики прерываний. Драйверы устройств. Буферизация. Сообщения об ошибках. Захват и освобождение выделенных устройств**

***Программный ввод-вывод***

Есть три фундаментально различных способа осуществления операций ввода-вывода. Проще всего организовать ввод-вывод, возложив всю работу на центральный процессор. Этот способ называется программным вводом-выводом. Основное проявление программного ввода-вывода состоит в том, что после вывода символа центральный процессор постоянно опрашивает устройство на готовность приема следующего символа.

***Обработчики прерываний***

Когда происходит прерывание, то все необходимое для его обработки делает процедура — обработчик прерывания. Затем она может разблокировать ожидавший данного прерывания драйвер. Она вызывает процедуру signal для переменной условия в мониторе или посылает заблокированному драйверу сообщение. В любом случае заключительным действием прерывания будет предоставление возможности возобновления работы ранее заблокированному драйверу.

***Драйверы устройств***

Программа для управления каждым подключенным к компьютеру устройством ввода-вывода, учитывающая его особенности, называется драйвером устройства. Каждый драйвер устройства обычно управляет одним типом устройства или как максимум одним классом родственных устройств.

Самые распространенные категории — это драйверы блочных устройств, к ним относятся драйверы дисков, содержащих множество блоков данных, к которым можно обращаться независимо от всех остальных блоков, и драйверы символьных устройств, к которым относятся драйверы клавиатур и принтеров — устройств, которые генерируют или воспринимают поток символов.

***Буферизация***

Существует несколько распространенных форм буферизации: двойная буферизация и использование кольцевого буфера. В первом случае два буфера работают по очереди: пока один из них копируется в пространство пользователя, другой аккумулирует новые поступления. Во втором случае буфер состоит из области памяти и двух указателей, один из которых указывает на следующее свободное слово, в которое можно поместить новые данные, а другой — на первое слово тех данных в буфере, которые еще не были из него выведены. Оба указателя ходят по кругу, переходя обратно к нижним адресам буфера, как только достигнут его верхних адресов.

***Сообщения об ошибках***

При возникновении ошибок операционная система должна их обработать наилучшим образом. Есть ошибки, которые возникают в том случае, если процесс запрашивает что-нибудь невозможное. Другие ошибки возникают при предоставлении неверного адреса буфера или других параметров, или указании неверного устройства. К другому классу относятся фактические ошибки ввода-вывода. При таких обстоятельствах решение о дальнейших действиях возлагается на драйвер.

***Распределение и высвобождение выделенных устройств***

Некоторые устройства в любой момент времени могут использоваться только одним процессом. Операционная система должна проверять запросы на использование и принимать их или отвергать в зависимости от доступности запрашиваемого устройства. Простой способ обработки этих запросов заключается в требовании к процессам непосредственно открывать специальные файлы для этих устройств с помощью системных вызовов open. Если устройство недоступно, то системный вызов open потерпит неудачу. Освобождение выделенного устройства происходит после его закрытия с помощью системного вызова close.

**30. Программное обеспечение ввода-вывода пользовательского пространства. Взаимная блокировка. Ресурсы**

Хотя большая часть программного обеспечения ввода/вывода пространства пользователя находится в операционной системе, небольшие его порции состоят из библиотек, присоединенных к программам пользователя, или даже целых программ, работающих вне ядра. Системные вызовы, включая системные вызовы ввода/вывода, обычно собраны из библиотечных процедур. Набор всех этих библиотечных процедур, несомненно, является частью системы ввода/вывода.  
Стандартная библиотека ввода/вывода содержит большое количество процедур, включающих операции ввода/вывода и работающих как часть программы пользователя.  
Не все программное обеспечение ввода/вывода пространства пользователя состоит из библиотечных процедур. Другая важная категория — это система спулинга. Спулинг представляет собой способ работы с выделенными устройствами в многозадачной системе.

***Взаимоблокировка***  
В компьютерных системах существует большое количество ресурсов, каждый из которых в конкретный момент времени может использоваться только одним процессом. Поэтому все операционные системы обладают способностью предоставлять процессу эксклюзивный доступ (по крайней мере, временный) к определенным ресурсам.  
Часто прикладной процесс нуждается в исключительном доступе не к одному, а к нескольким ресурсам. В этот момент процессы заблокированы и будут вечно оставаться в подвешенном состоянии. Такая ситуация называется тупиком или взаимоблокировкой.  
Взаимоблокировки появляются при работе как с аппаратными, так и с программными ресурсами.  
***Ресурсы***Система может зайти в тупик, когда процессам предоставляются исключительные права доступа к устройствам, файлам и т. д. Объекты предоставления доступа называются ресурсами. Ресурсом может быть аппаратное устройство или часть информации. Кроме того, в системе может оказаться несколько идентичных экземпляров какого-либо ресурса. Если в системе есть несколько экземпляров ресурса, то в ответ на обращение к нему может предоставляться любая из доступных копий. Короче говоря, ресурс — это все то, что вправе использоваться только одним процессом в любой момент времени.  
Ресурсы бывают двух типов: выгружаемые и невыгружаемые. Выгружаемый ресурс позволяет безболезненно забирать у владеющего им процесса.. Невыгружаемый ресурс, в противоположность выгружаемому, — это такой ресурс, который нельзя забрать от текущего владельца, не уничтожив результаты вычислений.

Взаимоблокировки касаются невыгружаемых ресурсов. Если ресурс недоступен, запрашивающий его процесс вынужден ждать. В некоторых операционных системах при неудачном обращении к ресурсу процесс автоматически блокируется и возобновляется только после того, как ресурс становится доступным. В других системах запрос ресурса, получивший отказ, возвращает код ошибки, тогда вызывающий процесс может подождать немного и повторить попытку.

**31. Механизм взаимной блокировки. Условия взаимной блокировки. Моделирование взаимных блокировок**

Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.Так как все процессы находятся в состоянии ожидания, ни один из них не будет причиной какого-либо события, которое могло бы активизировать любой другой процесс в группе, и все процессы продолжают ждать до бесконечности. В этой модели мы предполагаем, что процессы имеют только один поток управления и что нет прерываний, способных активизировать заблокированный процесс. Условие отсутствия прерываний необходимо, чтобы предотвратить ситуацию, когда тот или иной заблокированный процесс активизируется, скажем, по сигналу тревоги и затем приводит к событию, которое освободит другие процессы в группе.  
В большинстве случаев событием, которого ждет каждый процесс, является возврат какого-либо ресурса, в данный момент занятого другим процессом группы. Ни один из процессов не может работать, ни один из них не может освободить какой-либо ресурс и ни один из них не может возобновиться.  
Условия взаимоблокировки  
Коффман и другие исследователи доказали, что для возникновения ситуации взаимоблокировки должны выполняться четыре условия.  
1. Условие взаимного исключения.  
2. Условие удержания и ожидания.  
3. Условие отсутствия принудительной выгрузки ресурса.  
4. Условие циклического ожидания.  
Для того чтобы произошла взаимоблокировка, должны выполниться все эти четыре условия. Если хоть одно из них отсутствует, тупиковая ситуация невозможна.  
Моделирование взаимоблокировок  
Холт показал, как смоделировать четыре условия возникновения тупиков, используя направленные графы. Графы имеют два вида узлов: процессы, и ресурсы. Ребро, направленное от узла ресурса к узлу процесса, означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется этим процессом.  
Ребро, направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу.

**32. Администрирование доступа к ресурсам. Алгоритм страуса. Обнаружение и устранение взаимных блокировок. Предотвращение взаимных блокировок. Избежание взаимных блокировок. Алгоритм банкира**

***Страусовый алгоритм***

Это стратегия игнорирования потенциальных проблем на том основании, что они могут быть чрезвычайно редкими. Например, ОС Unix, имеющая в ядре ряд массивов фиксированной размерности, потенциально страдает от тупиков, даже если они не обнаружены.  Подход Unix состоит в том, чтобы игнорировать данную проблему в предположении, что большинство пользователей предпочтут случайный тупик нелепым правилам заставляющих их иметь один процесс, один открытый файл и т.п. Таким образом, мы сталкиваемся с нежелательным выбором между строгостью и удобством.  Трудно найти общее, устраивающее всех решение.  
***Обнаружение и устранение взаимоблокировок***

При использовании этого метода **система не пытается предотвратить попадание в тупиковые ситуации и не игнорирует их**. Вместо этого она **позволяет взаимоблокировке произойти**, старается определить, когда это случилось, и затем совершает некие действия к возврату си­стемы к состоянию, имевшему место до того, как система попала в тупик.

Алгоритм банкира

Алгоритм планирования, позволяющий избегать взаимных блокировок. Алгоритм моделирует банкира в маленьком городке, имеющего дело с группой клиентов, которым он выдал ряд кредитов. У банкира не обязательно имеется на руках достаточно средств, чтобы выдать каждому клиенту полную сумму кредита. Банкир знает, что не всем клиентам понадобится вся сумма немедленно, поэтому он зарезервировал только часть единиц, а не все, которые требуются клиентам. Он рассчитывает на то, что каждый из клиентов будет способен погасить кредит вскоре после его получения. В этом случае банкир сможет обслужить все запросы (чтобы провести аналогию с компьютерной системой, считаем, что клиенты — это процессы, единицами, скажем, являются накопители на магнитной ленте, а банкир — это операционная система). В алгоритме банкира каждый запрос по мере поступления изучается на предмет того, приведет ли его удовлетворение к безопасному состоянию. Если да, процесс получает ресурс, иначе запрос откладывается на более позднее время. Чтобы понять, является ли состояние безопасным, банкир оценивает, достаточно ли ресурсов для завершения работы какого-либо клиента. Если да, эти ссуды считаются погашенными, после чего проверяется следующий ближайший к верхнему пределу займа клиент и т. д. Если, в конце концов, все ссуды могут быть погашены, состояние является безопасным, и исходный запрос можно удовлетворить.

**33. Реальные диски. Аппаратное обеспечение диска**

Существует множество различных типов дисков. Наиболее распространенными из них являются магнитные жесткие диски. Их характеризует сравнительно высокая скорость чтения-записи данных, что позволяет им быть вполне приемлемой второстепенной памятью (для реализации страничной организации памяти, файловой системы и т. д.). Массивы, составленные из таких дисков, иногда используются для организации высоконадежного хранилища данных. Для распространения программ, данных и фильмов не менее большое значение имеют разнообразные оптические диски (DVD и Blu-rayдиски). И наконец, растущей популярностью благодаря высокой скорости работы и отсутствию механических частей пользуются твердотельные диски. В следующих разделах в качестве примера оборудования будут рассмотрены магнитные диски, а затем дано описание программного обеспечения для дисковых устройств в целом.

Основные понятия:

* **Головка (Head)** - электромагнит, скользящий над поверхностью диска, для каждой поверхности используется своя головка. Нумерация начинается с 0.
* **Дорожка (Track)** - концентрическая окружность, которое может прочитать головка в одной позиции. Нумерация дорожек начинается с внешней (первая имеет номер - 0).
* **Цилиндр (Cylinder)** - совокупность всех дорожек с одинаковым номером на всех дисках, т.к. дисков может быть много и на каждом диске запись может быть с двух сторон.
* **Маркер** - от него начинается нумерация дорожек, есть на каждом диске.
* **Сектор** - на сектора разбивается каждая дорожка, сектор содержит минимальный блок информации. Нумерация секторов начинается от маркера.
* **Геометрия жесткого диска**- набор параметров диска, количество головок, количество цилиндров и количество секторов.
* У современных жестких дисков **контроллер**встроен в само устройство, и берет на себя большую часть работы, которую не видит ОС.

Например, скрывают физическую геометрию диска, предоставляя виртуальную геометрии.

Физическая и виртуальная геометрия диска

На внешних дорожках число секторов делают больше, а на внутренних меньше. На реальных дисках таких зон может быть несколько десятков.

**34. RAID. Программное обеспечение жестких дисков. Обработка ошибок. Кеширование**

Один из классов устройств ввода-вывода RAID (Redundant Array of Inexpensive Disks) — избыточный массив недорогих дисков. Все RAID-системы обладают свойством распределения данных по дисковым устройствам, позволявшим проводить параллельные операции.

**RAID-система имеет несколько уровней**

Лучше всего RAID-система уровня 0 справляется с большими запросами, и чем больше запрос, тем лучше.

**RAID-система уровня 1.** В нем все диски продублированы, то есть на четыре первичных диска приходится четыре дублирующих. При записи каждый блок записывается дважды. При чтении может быть использована любая из копий, что позволяет распределить нагрузку на большее количество приводов.

В отличие от RAID-систем уровней 0 и 1, работающих с блоками, состоящими из секторов, **RAID-система уровня 2** работает на основе слов и даже байтов.

**RAID-система уровня 3** является упрощенной версией RAID уровня 2. Здесь для каждого слова данных вычисляется один бит четности, который записывается на отдельный диск четности.

**RAID-система уровня 4** похожа на RAID-систему уровня 0 с блоком четности для каждой группы блоков, который записывается на отдельном диске.

Вследствие большой нагрузки на диск четности он может стать узким местом, которое в **RAID-системе уровня 5** устраняется за счет распределения битов четности равномерно по кругу на все дисковые приводы. Но в случае повреждения диска реконструировать его содержимое будет непросто.

**RAID-система уровня 6** похожа на систему уровня 5, за исключением использования дополнительного блока четности. Иными словами, данные распределяются по дискам с двумя блоками четности вместо одного.

***Обработка ошибок***

Производственные дефекты проявляются в сбойных секторах, из которых неверно считывается только что записанная информация. Если дефект весьма незначителен и составляет, скажем, всего несколько бит, то сбойный сектор может использоваться при условии постоянной корректировки ошибок с помощью ECC-кода. При более существенных размерах скрыть дефект уже невозможно. Существуют **два основных подхода** к решению проблемы сбойных блоков: **обработка таких блоков в контроллере** или их **обработка средствами операционной системы**. При первом подходе еще до выпуска диска с предприятия он проходит тестирование, и на него записывается список сбойных секторов. Другой способ заключается в сдвиге всех секторов на одну позицию.

***Кеширование***

В современных жестких дисках задержки позиционирования и подхода сектора настолько влияют на производительность, что чтение по одному или по два сектора считается крайне неэффективной операцией. Поэтому многие контроллеры дисков всегда читают и помещают в кэш сразу несколько секторов, даже если запрос пришел только на один из них.

**35. Терминалы. Аппаратное обеспечение терминала. Программное обеспечение терминала**

В качестве средства взаимодействия человека с компьютером используются клавиатура и монитор. На больших универсальных компьютерах зачастую работает множество удаленных пользователей, у каждого из которых в качестве отдельного модуля имеется устройство, состоящее из клавиатуры и подключенного дисплея. Исторически такие устройства называются терминалами.

***Клавиатуры***

На персональных компьютерах клавиатура содержит микропроцессор, который обычно через специализированный последовательный порт обменивается данными с микросхемой контроллера, расположенной на системной плате (хотя сейчас все чаще клавиатура подключается к порту USB). Одно прерывание генерируется при нажатии клавиши, а второе — сразу же, как только она будет отпущена. По каждому из этих клавиатурных прерываний драйвер клавиатуры извлекает информацию о том, что именно произошло, пользуясь при этом портом ввода-вывода, связанным с клавиатурой. Все остальное происходит благодаря программному обеспечению и практически не зависит от аппаратуры.

***Мышь***

Популярной разновидностью терминалов является оптическая мышь, оборудованная в нижней части одним или несколькими светоизлучающими диодами и фотодетекторами. Современные оптические мыши имеют встроенную микросхему обработки изображения и производят в низком разрешении непрерывную съемку находящейся под ними поверхности, изучая изменение изображений от снимка к снимку. Как только мышь будет перемещена в любом направлении на определенное минимальное расстояние или ее кнопка будет нажата или освобождена, компьютеру будет отослано сообщение. У мышей могут быть одна, две или три кнопки. У некоторых мышей имеется колесо, которое может отправлять компьютеру дополнительные данные. Мышь показывает только изменение позиции, а не саму по себе абсолютную позицию.

***Программное обеспечение вывода информации***

***Текстовые окна***

Вывод проще ввода, когда он представляет собой последовательный вывод символов одного и того же шрифта, размера и цвета. Большей частью программы отправляют символы в текущее окно, где они и отображаются. Обычно за один системный вызов выводится блок символов, к примеру целая строка. Экранные редакторы и многие другие сложные программы должны уметь обновлять экран сложными способами, например удалять одну строку в середине экрана.

***Графические пользовательские интерфейсы***

В GUI есть четыре наиболее важных элемента, обозначаемых символами WIMP. Эти буквы означают соответственно: окна — Windows, значки — Icons, меню — Menus и указывающие устройства — Pointing device. Окна представляют собой прямоугольные области экрана, используемые для запуска программ. Значки являются небольшими обозначениями, на которых можно щелкать мышью с целью выполнения каких-либо действий. Меню представляют собой перечни действий, любое из которых может быть выбрано. И наконец, указывающие устройства — это мыши, трекболы или другая аппаратура, используемая для перемещения указателя по экрану с целью выбора элементов.

**36. Оболочка**

Оболочка операционной системы — интерпретатор команд операционной системы, обеспечивающий интерфейс для взаимодействия пользователя с функциями системы.  
Оболочка - это самый внешний уровень операционной системы. Оболочки содержат в себе язык программирования для управления процессами и файлами, а также запуска и управления другими программами. Оболочка управляет взаимодействием между вами и операционной системой, показывая вам приглашения для ввода, интерпретируя ввод для операционной системы, а затем обрабатывая результирующий вывод операционной системы.  
Оболочки предоставляют вам способ взаимодействия с операционной системой. Это взаимодействие проводится или интерактивно (ввод с клавиатуры обрабатывается немедленно), или в качестве сценария оболочки. Сценарий оболочки - это последовательность команд оболочки и операционной системы, которая хранится в файле.  
Когда вы входите в систему, она находит имя программы оболочки для выполнения. После запуска оболочка показывает приглашение командной строки. Этим приглашением обычно является $. При вводе команды и нажатии клавиши Enter оболочка вычисляет команду и пытается ее выполнить. В зависимости от инструкций команды оболочка записывает вывод команды на экран или перенаправляет вывод. Затем она снова показывает приглашение командной строки и ожидает ввода другой команды.  
Оболочки операционных систем обеспечивают:  
создание, переименование, копирование, пересылку, удаление и быстрый поиск файла в текущем каталоге диска или на всех дисках компьютера;  
просмотр, создание и сравнение каталогов;  
просмотр, создание и редактирование текстовых файлов;  
архивацию, обновление и разархивацию архивных файлов, и просмотр архивов;  
синхронизацию каталогов, расщепление и слияние файлов;  
поддержку связи двух компьютеров через последовательный или параллельный порты;  
форматирование и копирование дискет, смену метки дискеты и метки тома для жестких дисков, а также чистку дисков от ненужных файлов;  
запуск программ.

**37. Управление памятью. Базовые механизмы управления памятью. Однозадачная система без подкачки и замещения страниц. Многозадачная система с фиксированными разделами**

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса. Функциями ОС по управлению памятью являются: отслеживание свободной и занятой памяти, выделение памяти процессам и освобождение памяти при завершении процессов, вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место, а также настройка адресов программы на конкретную область физической памяти.  
Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную и вторичную память.  
Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес. Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.  
Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.  
Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:  
отображение адресного пространства процесса на конкретные области физической памяти;  
распределение памяти между конкурирующими процессами;  
контроль доступа к адресным пространствам процессов;  
выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;  
учет свободной и занятой памяти.

**38. Переадресация и защита**

Многозадачность вносит две существенные проблемы, требующие решения, — это переадресация для перемещения программы в памяти и защита. Когда программа компонуется, компоновщик должен знать, с какого адреса будет начинаться программа в памяти.  
Например, предположим, что первая команда представляет собой вызов процедуры с абсолютным адресом 100 внутри двоичного файла, создаваемого компоновщиком. Если эта программа загрузится в раздел 1, команда обратится к абсолютному адресу 100, принадлежащему операционной системе. А нужно вызвать процедуру по адресу 100 К + 100. Если же программа загрузится в раздел 2, команду нужно переадресовать по адресу 200 К + 100 и т. д. Эта проблема известна как проблема переадресации.  
Одним из возможных решений является модификация команд во время загрузки программы в память.  
В программе, загружаемой в раздел 1, к каждому адресу прибавляется значение 100 К, в программе, которая попадает в раздел 2, к адресам добавляется значение 200 К и т. д. Чтобы выполнить подобную переадресацию во время загрузки, компоновщик должен включить в двоичную программу список или битовую карту с информацией о том, какие слова в программе являются адресами, а какие — кодами машинных команд. Так работает операционная система OS/MFT.  
Переадресация во время загрузки не решает проблемы защиты. Вредоносные программы всегда могут организовать какую-нибудь новую команду перехода и с ее помощью проникнуть в память. Поскольку в такой системе используется абсолютная адресация памяти, а не смещение относительно того или иного регистра, не существует способа, который позволил бы запретить программе обращаться к любому слову в памяти для его чтения или записи.  
В многопользовательских системах крайне нежелательно разрешать процессам доступ к области памяти, принадлежащей другим пользователям.  
Для защиты компьютера IBM 360 разработчики приняли следующее решение: они разделили память на блоки по 2 Кбайт и назначили каждому блоку 4-разрядный код защиты. Этот 4-разрядный ключ содержал слово состояния программы (Program Status Word, PSW). Аппаратура IBM 360 перехватывала все попытки работающих процессов обратиться к любой части памяти, код защиты которой отличался от содержимого слова состояния программы.  
Альтернативное решение сразу обеих проблем (защиты и переадресации) заключается в оснащении машины двумя специальными аппаратными регистрами, называемыми базовым и ограничительным. При планировании процесса в базовый регистр загружается адрес начала раздела памяти, а в ограничивающий регистр — длина раздела. К каждому автоматически формируемому адресу перед его передачей в память прибавляется содержимое базового регистра. Таким образом, если базовый регистр содержит величину 100 К, команда CALL 100 будет превращена в команду CALL 100К + 100 без изменения самой команды. Базовый и ограничительный регистры защищаются аппаратно, чтобы не допустить их изменений пользовательскими программами.  
Неудобство, присущее этой схеме, — необходимость выполнять операции сложения и сравнения при каждом обращении к памяти. Операция сравнения может быть выполнена быстро, но сложение относительно нее — медленная операция, что обусловлено временем распространения сигнала.

**39. Подкачка. Алгоритмы замещения страниц. Оптимальный алгоритм замещения страниц. Алгоритм NRU. Алгоритм FIFO. Алгоритм второго шанса. Алгоритм часов. Алгоритм LRU.**

**Подкачка**

Подкачка — один из механизмов виртуальной памяти, при котором отдельные фрагменты памяти перемещаются из ОЗУ во вторичное хранилище, освобождая ОЗУ для загрузки других активных фрагментов памяти. Такими фрагментами в современных ЭВМ являются страницы памяти.

***Алгоритмы замещения страниц***

Основная цель алгоритма замены страницы заключается в том, чтобы сделать частоту замены страницы (также можно сказать, чтобы быть самым низким уровнем кармана).

***Оптимальный алгоритм замещения страниц***

Выбранная страница никогда не будет использоваться позже, или она больше не доступа к доступами до доступа, обычно обеспечивая самую низкую отсутствующую страницу. Однако, поскольку люди не могут предсказать, сколько времени больше не доступна, алгоритм является теоретическим алгоритмом. Оптимальный алгоритм замены может использоваться для оценки других алгоритмов.

***Алгоритм NRU***

Алгоритм NRU (Not Recently Used) удаляет страницу с помощью случайного поиска в непустом классе с наименьшим номером. Привлекательность алгоритма NRU заключается в том, что он легок для понимания, умеренно сложен в реализации и дает производительность, которая может вполне оказаться достаточной.

***Алгоритм FIFO***

Выбор страницы - первая страница страницы. Алгоритм прост, но он также будет заменен страницами, которые часто получают доступ, тем самым поднимая скорость стирки.

Пример: все еще используйте вышеприведенный пример, используйте алгоритм FIFO для замены страницы, процесс следующий следующим образом:

Замена страницы осуществляется более чем лучшим алгоритмом размещения.

**Алгоритм второго шанса.**

Алгоритм FIFO может заменить часто используемую страницу, чтобы избежать этой проблемы, сделать простую модификацию для этого алгоритма:

Установите Bit it Page, когда страница доступна (чтение или запись). Когда вам нужно заменить, установите флажок R старейшую страницу. Если BIT R равен 0, то эта страница старая и не используется, может быть заменена немедленно; если это 1, то бит r очищается 0, поставьте страницу в конце списка, измените время его загрузки. Так же, как просто поставить его, затем продолжать искать из главы связанного списка.

***Алгоритм часов***

Алгоритм второй возможности должен переместить страницу в связанном списке для снижения эффективности. Часовой алгоритм использует список связанного кольца для связи страницы и использовать указатель на самую старую страницу.

***Алгоритм LRU***

Хотя страница, которая будет использоваться в будущем, не может знать, вы можете знать случай прошлого, используя страницу в прошлом. LRU будет заменена на ранее неиспользованной странице, которая не используется в прошлом некоторое время и не будет доступна в ближайшем будущем.

Поддерживайте связанный список всех страниц в памяти. Когда доступна страница, переместите эту страницу в заголовок списка. Это гарантирует, что страница длится в последний раз.

**40. Локальная и глобальная политики распределения памяти. Размер страницы. Интерфейс виртуальной памяти. Сегментация.**

**Локальная и глобальная политики распределения памяти**

Локальные алгоритмы соответствуют размещению каждого процесса в фиксированной области памяти. Глобальные алгоритмы динамически распределяют страничные блоки между выполняющимися процессами. Таким образом, количество страничных блоков, предоставленных каждому процессу, изменяется со временем. В целом глобальные алгоритмы работают лучше, особенно если размер рабочего набора может изменяться за время жизни процесса. Если используется локальный алгоритм и рабочий набор увеличивается в размере, мы получим пробуксовку. Когда рабочий набор уменьшается, в случае локального алгоритма часть памяти тратится впустую. Если же задействован глобальный алгоритм, система должна непрерывно решать, сколько страничных блоков предоставить каждому процессу.  
  
Зачастую размер страницы является параметром, выбираемым операционной системой. Определение оптимального размера страницы требует учета нескольких взаимосвязанных факторов. Прежде всего, есть два довода в пользу маленького размера страниц. Случайно выбранный текст, данные или сегмент стека не заполняют страницы целиком. В среднем половина последней страницы оказывается пустой, и это дополнительное пространство пропадает. Такие потери называют внутренней фрагментацией. Большой размер страницы скорее, чем маленький, станет причиной того, что в памяти окажется неиспользуемая часть страницы.  
В некоторых системах программистам доступен контроль над отображением памяти, и они могут воспользоваться им, чтобы обогатить поведение программы. Одним из поводов предоставления программистам контроля над отображением памяти является разрешение одному или нескольким процессам совместно использовать одну и ту же память. Если программисты могут присваивать имена областям памяти, то появляется возможность одному процессу предоставить другому процессу имя области памяти, чтобы этот процесс мог также отображаться на нее. Когда два (или несколько) процесса совместно используют одни и те же страницы, появляется возможность использования общего высокоскоростного канала: один процесс ведет запись в общую память, а другой процесс считывает из нее данные.  
Сегментация - это деление памяти на сегменты. Это механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния. С точки зрения разработчиков программного обеспечения, сегментация дает удобный способ совместного использования информации несколькими процессами. Конкретный сегмент может использоваться совместно с другими без нарушения требований его защиты. Сегментация также предполагает естественное разделение программных строк и данных и отделение модуля от модуля.

**41. Управление памятью с помощью битовых карт. Управление памятью с помощью связанных списков**

Если память выделяется динамически, этим процессом должна управлять операционная система. Существует два подхода к учету использования памяти: битовые карты и списки свободных областей. При работе с битовой картой память разделяется на блоки размером от нескольких слов до нескольких килобайтов. В битовой карте каждому свободному блоку  
соответствует один нулевой бит, а каждому занятому блоку — бит, установленный в 1 (или наоборот). Размер минимального выделяемого блока — весьма важный параметр системы. Чем он меньше, тем больше битовая карта. Если же отдать предпочтение большим блокам, битовая карта станет меньше, но при этом может оказаться неиспользуемой существенная часть последнего блока каждого процесса. Битовая карта предоставляет простой механизм отслеживания слов в памяти фиксированного объема, поскольку ее размер зависит только от объема памяти и размера минимального блока. Для этой схемы характерна проблема — при решении переместить процесс из к блоков в память менеджер памяти должен найти в битовой карте последовательность из к смежных нулевых битов. Поиск последовательности заданной длины в битовой карте является медленной операцией (так как искомая последовательность битов может пересекать границы слов в битовом массиве). Это — главный аргумент противников битовых карт.  
Другой подход к отслеживанию состояния памяти предоставляют связанные списки занятых и свободных фрагментов памяти, где фрагментом является либо процесс, либо участок между двумя процессами. Каждая запись в списке указывает, является ли область памяти свободной или занятой процессом, а также содержит адрес, с которого начинается эта область, ее длину и указатель на следующую запись. Так как ячейка таблицы процессов для завершившегося процесса обычно непосредственно указывает на запись в списке для этого процесса, возможно, удобнее иметь список с двумя связями, чем с одной. Такая двунаправленная структура упрощает поиск предыдущей записи и оценку возможности конкатенации.

**42. Виртуальная память. Замещение страниц. Таблицы страниц. Буферы быстрого преобразования адресов. Инвертированные таблицы страниц**

***Виртуальная память***

В основе виртуальной памяти лежит идея, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые страницами. Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету, иначе операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду.

При использовании виртуальной памяти вместо отдельного перемещения только сегмента текста или данных программы на физическую память в сравнительно небольших блоках может быть отображено все адресное пространство.

***Замещение страниц***

Замещение страниц лучше всего работает при наличии достаточного количества свободных страничных блоков, которые могут потребоваться при возникновении ошибки отсутствия страницы. Над поставкой свободных страничных блоков работает фоновый процесс, называемый страничным демоном, который периодически проверяет состояния памяти. Если свободно слишком мало страничных блоков, страничный демон начинает подбор страниц для выгрузки путём замещения некоторых страниц.

***Таблицы страниц***

При простой реализации отображение виртуальных адресов на физические может быть сведено к следующему: виртуальный адрес делится на номер виртуальной страницы и смещение.

Номер виртуальной страницы используется в качестве индекса внутри таблицы страниц, нужный для поиска записи для этой страницы. Из записи в таблице берется номер страничного блока. Номер страничного блока присоединяется к старшим битам смещения, заменяя собой номер виртуальной страницы, чтобы сформировать физический адрес, который может быть послан к памяти.

Таблица страниц нужна для отображения виртуальных страниц на страничные блоки.

***Буферы быстрого преобразования адресов***

Буфер быстрого преобразования адресов нужен для отображения виртуальных адресов на физические без просмотра таблицы страниц. Зачастую это устройство находится внутри диспетчера памяти и состоит из небольшого количества записей. Эти поля имеют точное соответствие с полями в таблице страниц, за исключением номера виртуальной страницы, который в таблице страниц не нужен. Еще один бит указывает на то используется страница или нет.

***Инвертированные таблицы страниц***

Альтернатива постоянно растущим уровням иерархии страничной адресации называется инвертированными таблицами страниц. В данной конструкции имеется одна запись для каждого страничного блока в реальной памяти, а не одна запись на каждую страницу в виртуальном адресном пространстве. В каждой записи отслеживается, что именно находится в страничном.

Инвертированные таблицы страниц нашли широкое применение на 64-разрядных машинах, так как даже при очень больших размерах страниц количество записей в обычных таблицах страниц будет слишком большим.

**43. Системные вызовы**

Системные вызовы (system calls) — это интерфейс между операционн ной 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.

**44. Системные вызовы для управления процессами. Системные вызовы fork, exit и wait. Системный вызов exec**

***Системные вызовы для управления процессами***

Первая группа вызовов предназначена для управления процессами. Начнем рассмотрение системного вызова fork (разветвление). Вызов fork является единственным существующим в POSIX способом создания нового процесса.  После выполнения вызова fork исходный процесс и его копия (родительский и дочерний процессы) выполняются независимо друг от друга.

В большинстве случаев после вызова fork дочернему процессу необходимо выполнить программный код, отличный от родительского.

Вызов fork используется также и в виде оболочки. После набора команды оболочка создает дочерний процесс, который должен выполнить команду пользователя.

***Системные вызовы fork, exit и wait***

Когда создаются или уничтожаются процессы, необходимо выделять и освобождать память. Кроме того, нужно обновлять таблицу процессов, в том числе и те ее части, которые поддерживаются ядром и файловой системой. Эту деятельность координирует менеджер процессов. За создание процесса отвечает вызов fork, который описывался выше, выполняющийся за несколько шагов.

Если таблица еще не заполнена, делается попытка выделить память под данные и стек дочернего процесса. Для процессов с разделенными пространствами кода и данных запрашивается только память, достаточная для размещения стека и данных. Если этот шаг пройден успешно, fork гарантированно выполняется. Затем выделенная область памяти заполняется, в таблице процессов находится и заполняется ячейка нового процесса, для него выбирается PID и другие части системы информируются о создании нового процесса.

Процесс полностью завершается только по факту наступления двух следующих событий:

♦ Процесс закончил выполняться (самостоятельно или по сигналу).

♦ Родительский процесс, чтобы выяснить, чем все закончилось, выполнил системный вызов wait.

Процесс, прекративший выполняться или завершенный по сигналу, при условии, что его родитель еще не выполнил вызов wait, попадает в своего рода состояние зомби. Он исключается из планирования, сигнальный таймер у него отключается (если он был включен), но процесс остается в таблице процессов. Память процесса при этом освобождается. В состоянии зомби процесс находится временно, и оно редко длится долго. Когда родительский процесс наконец выполняет вызов wait, занятая ячейка в таблице процессов освобождается, и файловая система и ядро уведомляются об этом.

***Системный вызов exec***

В операционной системе UNIX есть системный вызов exec для замены виртуального адресного пространства процесса.

Наиболее общий вариант его использования выглядит следующим образом:

*exec(name, argp, envp);*

Данный системный вызов загружает исполняемый файл name и передает ему аргументы, на которые указывает argp, и список переменных окружения, на который указывает envp.

**45. Системные вызовы для управления сигналами. Таймеры пользовательского пространства**

***Системные вызовы для управления сигналами***

Сигналы могут использоваться для того, чтобы перехватывать некоторые аппаратные исключения, например недопустимые инструкции или переполнение при операциях с плавающей точкой. Задержки также реализуются через сигналы.

Если процесс никак не анонсировал свое желание отвечать на сигналы, то, получив сигнал, он просто принудительно завершается. Чтобы избежать подобной судьбы процесс может использовать системный вызов sigaction, который позволяет указать новый адрес процедуры для обработки сигнала и узнать адрес прежней процедуры.

После того как такой вызов сделан, при получении сигнала соответствующего типа состояние процесса будет сохранено в стеке и вызван обработчик сигнала.

В MINIX 3 сигналы можно блокировать. Блокированный сигнал задерживается до тех пор, пока не будет разблокирован. Вызов sigprocmask позволяет процессу задать набор блокируемых сигналов, передавая ядру битовую карту.

***Таймеры пользовательского пространства***

В традиционных операционных системах работа с оповещениями полностью выполняется ядром или драйвером часов в пространстве ядра. В операционной системе MINIX 3 ответственность за оповещения пользовательских процессов возлагается на менеджер процессов. Это снижает нагрузку на ядро и упрощает код, выполняемый в пространстве ядра. Если заданному объему кода соответствует определенное число неизбежных ошибок, то с уменьшением ядра число ошибок также сокращается. Если количество ошибок останется неизменным, их последствия окажутся менее серьезными, если они будут происходить в пользовательском пространстве, а не в ядре.

В MINIX 3 в пространстве ядра поддерживаются лишь таймеры системных процессов. Менеджер процессов управляет другой очередью таймеров, которые запущены пользовательскими процессами, запросившими оповещения. Менеджер процессов запрашивает оповещение от часов только для таймера, находящегося в начале своей очереди. Когда после прерывания от часов определяется истечение таймера, менеджеру процессов передается уведомление. Менеджер процессов проверяет собственную очередь таймеров, посылает сигнал пользовательскому процессу и подает новый запрос, если его очередь не пуста.

Существует вероятность истечения нескольких таймеров в одном такте часов. Менеджер процессов берет задачу обработки сразу несколько истекших таймеров на себя, освобождая ядро от поиска и очистки собственной очереди, а также от генерации множества уведомлений.

Оповещения могут быть отменены. Нагрузка на код ядра снижается, если отмена выполняется в очереди менеджера процессов, находящегося за пределами ядра. Очередь ядра требует внимания лишь при истечении таймера, находящегося в ее начале, а также при изменении процесса, находящегося в начале очереди менеджера процессов.

**46. Системные вызовы для управления файлами, каталогами, правами доступа**

Чтобы прочитать данные из **файла** или записать их в файл, сначала его необходимо открыть. Для данного вызова необходимо указать имя открываемого файла (с указанием абсолютного пути либо пути относительно рабочего каталога) и код O\_RDONLY, O\_WRONLY или O\_RDWR, означающий, что файл открывается для чтения, записи или для чтения и записи соответственно. Для создания нового файла используется параметр O\_CREAT. Чтобы закрыть файл используется системный вызовов close. Системный вызов lseek может изменить значение указателя. Вызов lseek имеет три параметра: первый — это дескриптор файла, второй — позиция в файле, третий — указание, относительно чего задана позиция. Вызов lseek возвращает абсолютную позицию в файле (в байтах) после изменения указателя.

Для каждого файла UNIX хранит следующие данные: код режима файла, размер, время последнего изменения и другую информацию. Программы могут запрашивать эту информацию посредством системного вызова stat. Его первый параметр определяет файл, информацию о котором необходимо получить, второй является указателем на структуру, в которую она должна быть помещена. Для открытого файла системный вызов fstat.

Вызовы mkdir и rmdir соответственно создают и удаляют пустые **каталоги**. Вызов link позволяет одному и тому же файлу появляться под двумя или более именами. Системный вызов mount позволяет объединять в одну две файловые системы.  
При помощи системного вызова mount файловая система USB-диска может быть подключена к корневой файловой системе.

В языке C типичный оператор, выполняющий подключение («монтирование») файловой системы, выглядит так:  
mount("/dev/sdb0", "/mnt", 0);

Системный вызов chmod позволяет изменять биты прав доступа к файлу. Например, чтобы сделать файл доступным для чтения для всех, а для владельца — доступным и для чтения и для записи, необходимо выполнить следующий системный вызов:  
chmod("file", 0644);

**47. Структура ОС. Монолитные системы. Многоуровневые системы. Виртуальные машины**

***Монолитные системы***

Вся ОС работает как единая программа в режиме ядра. ОС написана в виде набора процедур, связанных вместе в одну большую исполняемую программу. Каждая процедура может свободно вызвать любую другую, если та выполняет какое-нибудь полезное действие, в котором нуждается первая процедура. Отказ в любой из этих процедур приведет к аварии всей ОС.

***Многоуровневые системы***

Изображение выглядит как стол

Автоматически созданное описаниеОбобщением подхода к монолитной системе является организация ОС в виде иерархии уровней, каждый из которых является надстройкой над нижележащим уровнем. Первой системой была система THE. У системы было шесть уровней.

Дальнейшее обобщение многоуровневой концепции было сделано в системе MULTICS. MULTICS использовались серии концентрических колец, где внутренние кольца обладали более высокими привилегиями по отношению к внешним. Когда процедуре из внешнего кольца требовалось вызвать процедуру внутреннего кольца, ей нужно было создать эквивалент системного вызова, то есть выполнить инструкцию TRAP.

***Виртуальные машины***

Основа системы, известная как монитор виртуальных машин, запускается непосредственно на обычном оборудовании и обеспечивает многозадачность, предоставляя верхнему уровню не одну, а несколько виртуальных машин. Но эти виртуальные машины не являются машинами с расширенной архитектурой. Они не поддерживают файлы и другие полезные свойства. Они являются точной копией исходной аппаратуры, включающей режим ядра и пользователя и все остальное что есть у настоящей машины.

Сегодня в число популярных гипервизоров кроме VMware иXen входят KVM (для ядра Linux), VirtualBox (от Oracle) и Hyper-V (от Microsoft).

*Виртуальная машина Java*

Виртуальные машины используются для запуска программ на языке Java. Компилятор Java создает код для JVM(Java Virtual Machine), который затем обычно выполняется программным интерпретатором JVM. Преимущество в том, что код для JVM может доставляться через Интернет на любой компьютер, имеющий JVM-интерпретатор, и запускаться на этом компьютере. Другим преимуществом использования JVM является то, что при полученные JVM-программы могут быть выполнены в защищенной среде, не имея возможности похитить данные или нанести любой другой вред.

**48. Безопасность. Безопасное окружение. Угрозы. Общие виды атак на систему безопасности. Принципы разработки механизмов безопасности**

У большинства компаний есть ценная информация, требующая надежной защиты. Охрана этой информации от несанкционированного использования становится основной задачей всех ОС. Компания VMware является одной из первых, кто задумался о создании **защищенных пользовательских окружений**. Она создала продукт VMware ACE 1.0. в котором можно было создавать защищенные политиками безопасности виртуальные машины и использовать их как рабочие среды для сотрудников компании.

**Угрозы.** Во многих работах, посвященных безопасности, безопасность информационных систем разбита на три компонента: конфиденциальность, целостность и доступность. Вместе все три компонента часто называют CIA. Конфиденциальность направлена на сохранение секретности данных. “Целостность” означает, что пользователи, не обладающие необходимыми правами, не должны иметь возможности изменять какие-либо данные без разрешения их владельцев. “Доступность” означает, что никто не может нарушить работу системы и вывести ее из строя. Многие из **средств взлома** были разработаны белыми хакерами. Изначально эти инструментальные средства служили удобными инструментами тестирования безопасности компьютерных систем. Например средство nmap, помогает взломщикам посредством сканирования портов выявлять сетевые услуги, предлагаемые компьютерной системой. Одной из простейших технологий сканирования, предлагаемых nmap, является попытка TCP-подключений к каждому возможному номеру порта на компьютерной системе. Успешное подключение свидетельствует о наличии на системе службы, прослушивающей этот порт. Еще один набор инструментов, имеющий общее название dsniff, предлагает различные способы отслеживания сетевого трафика и перенаправления сетевых пакетов. LOIC — это средство запуска атак, приводящих к отказу от обслуживания. Основная часть преступных действий в Интернете построена на инфраструктурах, известных как ботнеты, или бот-сети которые состоят из тысяч зараженных компьютеров. Еще одним аспектом безопасности, имеющим отношение к конфиденциальности, является закрытость : защита отдельных пользователей от злоупотреблений, связанных с их личной информацией.

**49. Аутентификация пользователей. Пароли. Механизмы защиты. Домены защиты. Списки управления доступом. Мандаты. Секретные каналы**

***Аутентификация***

Каждая надежная компьютерная система должна требовать от всех пользователей во время входа проходить аутентификацию.

Способы аутентификации пользователей основываются на трех основных принципах:

* на чем-нибудь, что известно пользователю;
* на чем-нибудь, что есть у пользователя;
* на чем-нибудь, что он собой представляет.

Наиболее широкое применение нашла форма аутентификации, требующая от пользователя ввода регистрационного имени и пароля.

***Пароли* Пароль** - условное слово или произвольный набор знаков, состоящий из букв, цифр и других символов, и предназначенный для подтверждения личности или полномочий. Если допустимо использование только цифр, то такую комбинацию иногда называют ПИН-кодом. Пароли часто используются для защиты информации от несанкционированного доступа.

* Пароль не должен быть слишком коротким.
* Пароль не должен быть словарным словом или простым их сочетанием.
* Пароль не должен состоять только из общедоступной информации о пользователе.
* В качестве популярных рекомендаций к составлению пароля можно назвать использование сочетания слов с цифрами и специальными символами , использование малораспространённых или несуществующих слов, соблюдение минимальной длины.

***Механизмы защиты***

В некоторых устаревших операционных системах пароли хранились на диске в незашифрованном виде, но защищались с помощью обычных системных механизмов защиты. В UNIX используется более удачное решение. Программа входа в систему просит пользователя ввести его имя и пароль. Пароль тут же «шифруется» за счет использования его в качестве ключа для зашифровки фиксированного блока данных. Затем программа входа в систему читает файл паролей, представляющий собой простой набор ASCII-строк, по одной для каждого пользователя, пока не найдет строку, в которой содержится регистрационное имя пользователя. Если зашифрованный пароль, содержащийся в этой строке, соответствует только что вычисленному зашифрованному паролю, вход в систему разрешается, а если не соответствует, то отклоняется.

***Домены защиты***

Домен представляет собой множество пар (объект, права доступа). Каждая пара определяет объект и некоторое подмножество операций, которые могут быть выполнены в отношении этого объекта. Одним из фундаментальных понятий является принцип минимальных полномочий.

***Списки управления доступом***

Списки управления доступом - это защищенные информационные ресурсы, в которых указано, кому и к каким ресурсам разрешен доступ. Операционная система обеспечивает раздельную защиту для разных режимов доступа. Владелец информации может предоставить другим пользователям права на запись или чтение для своего ресурса. Пользователь, которому предоставлены права доступа к ресурсу, может передавать эти права другим пользователям.

**50. Операции с отдельными файлами. Создание, открытие и закрытие файлов. Чтение файла. Запись. Каналы ввода-вывода**

**Операции с файлами**

Файлы предназначены для хранения информации с возможностью ее последующего извлечения. Наиболее распространенные системные вызовы, относящиеся к работе с файлами:

* + *Create* (Создать). Создает файл без данных.
  + *Delete* (Удалить). Когда файл больше не нужен, его нужно удалить, чтобы освободить дисковое пространство.
  + *Open* (Открыть). Перед использованием файла процесс должен его открыть.
  + *Close* (Закрыть). После завершения всех обращений к файлу потребность в его атрибутах и адресах на диске уже отпадает, поэтому файл должен быть закрыт, чтобы освободить место во внутренней таблице.
  + *Read* (Произвести чтение). Считывание данных из файла.
  + *Write* (Произвести запись). Запись данных в файл, как правило, с текущей позиции.

**Каналы ввода-вывода**

**Каналы ввода-вывода** и интерфейсы обеспечивают взаимодействие центральных устройств вычислительной машины и периферийных устройств.

Каналы ввода-вывода — самостоятельные в логическом отношении устройства, которые работают под управлением собственных программ, находящихся в памяти.

КВВ и интерфейсы выполняют следующие функции:

1. Позволяют иметь машины с переменным составом периферийных устройств.
2. Обеспечивают параллельную работу периферийных устройств как между собой, так и по отношению к процессору.
3. Обеспечивают автоматическое распознавание и реакцию процессора на различные ситуации, возникающие в периферийных устройствах.

Существует три вида каналов ввода-вывода:

* Мультиплексный канал. Сам канал быстродействующий, но обслуживает медленное периферийное устройство.
* Селекторный канал. Канал быстродействующий и обслуживает быстрые устройства.
* Блок-мультиплексорный канал. Подключившись к одному устройству, передаёт часть информации. После этого подключается к другому устройству.

Устройство “мультиплексный канал” предназначено для “параллельного” выполнения обменов с подключенными к нему несколькими внешними устройствами. Выполнение таких обменов производится группами данных, на которые автоматически разбивается массив данных, передаваемый в каждом обмене. Обмен блоками данных по мультиплексному каналу производится между оперативной памятью и выделенным для внешнего устройства буфером в памяти его контроллера.

К селекторным каналам через специальную аппаратуру непосредственного управления устройствами (“контроллеры” – control unit) подключаются устройства, обмен данными с которыми производится в режиме монопольного захвата канала. Эффективное использование селекторного канала возможно для обмена через него с так называемыми “быстрыми” внешними устройствами. Основными такими устройствами являются устройства внешней памяти на магнитных дисках.

**51. Монтирование файловых систем. Создание и уничтожение ссылок**

**Монтирование файловых систем**

Функция *mount* (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а функция *umount* (демонтировать) выключает файловую систему из иерархии*.*

Процедура ***монтирования*** состоит в следующем. Пользователь сообщает ОС имя устройства и место в файловой структуре (имя пустого каталога), куда нужно присоединить файловую систему (точка монтирования).

Затем ОС должна убедиться, что устройство содержит действительную файловую систему ожидаемого формата с суперблоком, списком индексов и корневым индексом.

Некоторые ОС осуществляют монтирование автоматически, как только встретят диск в первый раз (жесткие диски на этапе загрузки, гибкие - когда они вставлены в дисковод), ОС ищет файловую систему на устройстве. Если файловаясистема на устройстве имеется, она монтируется на корневом уровне, при этом к цепочке имен абсолютного имени файла (pathname) добавляется буква раздела. Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. Занесение информации в таблицу монтирования производится немедленно, поскольку может возникнуть конфликт между двумя процессами.

**Создание и уничтожение ссылок**

В Linux существует два типа ссылок на файлы. Это символические и жесткие ссылки Linux.

**Символические ссылки**

Символические ссылки более всего похожи на обычные ярлыки. Они содержат адрес нужного файла в вашей файловой системе. Когда вы пытаетесь открыть такую ссылку, то открывается целевой файл или папка. Главное ее отличие от жестких ссылок в том, что при удалении целевого файла ссылка останется, но она будет указывать в никуда, поскольку файла на самом деле больше нет.

Основные особенности символических ссылок:

* Могут ссылаться на файлы и каталоги;
* После удаления, перемещения или переименования файла становятся недействительными;
* Права доступа и номер inode отличаются от исходного файла;
* При изменении прав доступа для исходного файла, права на ссылку останутся неизменными;
* Можно ссылаться на другие разделы диска;
* Содержат только имя файла, а не его содержимое.

**Жесткие *ссылки***

Этот тип ссылок реализован на более низком уровне файловой системы. Файл размещен только в определенном месте жесткого диска. Но на это место могут ссылаться несколько ссылок из файловой системы. Каждая из ссылок - это отдельный файл, но ведут они к одному участку жесткого диска. Файл можно перемещать между каталогами, и все ссылки останутся рабочими, поскольку для них неважно имя. Оособенности:

* Работают только в пределах одной файловой системы;
* Нельзя ссылаться на каталоги;
* Имеют ту же информацию и набор разрешений что и у исходного файла;
* Разрешения на ссылку изменяться при изменении разрешений файла;
* Можно перемещать и переименовывать и даже удалять файл без вреда ссылке.

Для создания символических ссылок существует утилита ln. Ее синтаксис очень прост:

*$ ln опции файл\_источник файл\_ссылки*

А что касается удаления, если вы удалите ссылку, то исходный файл останется на месте.

1. Понятие операционной системы
2. История развития операционных систем
3. Процессы
4. Модель процессов
5. Создание процессов
6. Завершение процессов
7. Иерархии процессов
8. Состояния процессов
9. Реализация процессов
10. Программные потоки
11. Взаимодействие между процессами. Гонки. Критические секции. Взаимное исключение с активным ожиданием. Запрет на прерывания. Переменные блокировки. Строгое чередование. Алгоритм Петерсона. Команда TSL.
12. Примитивы взаимодействия между процессами. Семафоры. Мьютексы. Мониторы.
13. Передача сообщений. Разработка систем передачи сообщений. Решение проблемы производителя и потребителя путем передачи сообщений.
14. Классические проблемы взаимодействия между процессами. Проблема обедающих философов. Проблема читателей и писателей.
15. Планирование. Поведение процесса. Когда требуется планирование. Категории алгоритмов планирования. Цели алгоритма планирования.
16. Планирование в системах пакетной обработки.
17. Планирование в интерактивных системах.
18. Планирование в системах реального времени.
19. Планирование программных потоков.
20. Начальная загрузка.
21. Обработка прерываний.
22. Файлы. Именование файлов. Структура файла. Типы файлов. Доступ к файлам. Атрибуты файлов.
23. Операции с файлами. Каталоги. Пути. Каталоги в UNIX.
24. Реализация файлов. Неразрывные файлы. Связанные списки. Индексные узлы.
25. Организация дискового пространства. Размер блока. Учет свободных блоков. Надежность файловой системы. Непротиворечивость файловой системы. Резервные копии.
26. Производительность файловой системы. Кэширование. Упреждающее чтение блоков. Уменьшение количества перемещений головки диска.
27. Файловые системы с журнальной структурой.
28. Ввод-вывод. Аппаратное обеспечение ввода-вывода. Устройства ввода-вывода. Контроллеры устройств. Ввод-вывод с отображением на память. Прерывания. Прямой доступ к памяти.
29. Программное обеспечение ввода-вывода. Обработчики прерываний. Драйверы устройств. Буферизация. Сообщения об ошибках. Захват и освобождение выделенных устройств.
30. Программное обеспечение ввода-вывода пользовательского пространства. Взаимная блокировка. Ресурсы.
31. Механизм взаимной блокировки. Условия взаимной блокировки. Моделирование взаимных блокировок.
32. Администрирование доступа к ресурсам. Алгоритм страуса. Обнаружение и устранение взаимных блокировок. Предотвращение взаимных блокировок. Избежание взаимных блокировок. Алгоритм банкира.
33. Реальные диски. Аппаратное обеспечение диска.
34. RAID. Программное обеспечение жестких дисков. Обработка ошибок. Кеширование.
35. Терминалы. Аппаратное обеспечение терминала. Программное обеспечение терминала.
36. Оболочка
37. Управление памятью. Базовые механизмы управления памятью. Однозадачная система без подкачки и замещения страниц. Многозадачная система с фиксированными разделами.
38. Переадресация и защита.
39. Подкачка. Алгоритмы замещения страниц. Оптимальный алгоритм замещения страниц. Алгоритм NRU. Алгоритм FIFO. Алгоритм второго шанса. Алгоритм часов. Алгоритм LRU.
40. Локальная и глобальная политики распределения памяти. Размер страницы. Интерфейс виртуальной памяти. Сегментация.
41. Управление памятью с помощью битовых карт. Управление памятью с помощью связанных списков.
42. Виртуальная память. Замещение страниц. Таблицы страниц. Буферы быстрого преобразования адресов. Инвертированные таблицы страниц.
43. Системные вызовы
44. Системные вызовы для управления процессами. Системные вызовы fork, exit и wait. Системный вызов exec.
45. Системные вызовы для управления сигналами. Таймеры пользовательского пространства.
46. Системные вызовы для управления файлами, каталогами, правами доступа
47. Структура операционной системы. Монолитные системы. Многоуровневые системы. Виртуальные машины
48. Безопасность. Безопасное окружение. Угрозы. Общие виды атак на систему безопасности. Принципы разработки механизмов безопасности
49. Аутентификация пользователей. Пароли. Механизмы защиты. Домены защиты. Списки управления доступом. Мандаты. Секретные каналы.
50. Операции с отдельными файлами. Создание, открытие и закрытие файлов. Чтение файла. Запись. Каналы ввода-вывода.
51. Монтирование файловых систем. Создание и уничтожение ссылок.