Ответы <https://youtu.be/R8r8BX57RNM> на вопросы по ОСиСП

Ссылка на лекции Видни https://www.youtube.com/watch?v=QWv7ZpUkj8A&list=WL&index=1

Содержание

[1. Основы ОС UNIX, возможности 3](#_Toc107551544)

[2. Понятие системного вызова в ОС UNIX. Обработка ошибок 4](#_Toc107551545)

[3. Особенности современных операционных систем 5](#_Toc107551546)

[4. Файлы и файловая система в ОС UNIX. Права доступа 6](#_Toc107551547)

[5. Концепция слоистой ОС и системы на основе микроядра 8](#_Toc107551548)

[6. Владельцы файлов. Права доступа к файлам. Атрибуты файлов 10](#_Toc107551549)

[7. Функции микроядра 11](#_Toc107551550)

[8. Неименованные каналы ОС UNIX 13](#_Toc107551551)

[9. Понятие процесса, модели процессов 14](#_Toc107551552)

[10. Метаданные (атрибуты) файлов в ОС UNIX (?) 15](#_Toc107551553)

[11. Описание процесса, таблицы процесса (Реализация процесса Таненбаум) 16](#_Toc107551554)

[12. Процессы. Типы процессов 17](#_Toc107551555)

[13. Атрибуты процесса ОС UNIX 18](#_Toc107551556)

[14. Концепция потока, как составной части процесса 19](#_Toc107551557)

[15. Разработка программ в ОС UNIX. Обработка ошибок, переменные окружения 20](#_Toc107551558)

[16. Файлы, отображаемые в память (3.5.7) 23](#_Toc107551559)

[17. Процессы в ОС UNIX. Порождение процессов 24](#_Toc107551560)

[18. Сигналы в ОС UNIX. Их назначение и обработка 25](#_Toc107551561)

[19. Функции для работы с сигналами 28](#_Toc107551562)

[20. Файловая система Ехт2 29](#_Toc107551563)

[21. Файловая система в ОС UNIX EXT3 31](#_Toc107551564)

[22. Файловая система в ОС UNIX EXT4 32](#_Toc107551565)

[23. Процессы в ОС UNIX, системные вызовы wait, exit 34](#_Toc107551566)

[24. Взаимодействие процессов в ОС UNIX, очереди сообщений 36](#_Toc107551567)

[25. Взаимодействие процессов в ОС UNIX с применением семафоров 40](#_Toc107551568)

[26. Работа с файлами в ОС UNIX. Системные вызовы 42](#_Toc107551569)

[27. Взаимодействие процессов в ОС UNIX, разделяемая память 45](#_Toc107551570)

[28. Понятие потока в ОС UNIX. Создание потока, завершение потока 47](#_Toc107551571)

[29. Применение мьютексов при взаимодействии потоков в ОС UNIX 49](#_Toc107551572)

[30. Применение блокировок чтения-записи при взаимодействии потоков в ОС UNIX 51](#_Toc107551573)

[31. Взаимодействие процессов. Задача взаимного исключения 52](#_Toc107551574)

[32. Проблема тупиков. Алгоритм банкира 53](#_Toc107551575)

[33. Концепция виртуализации 56](#_Toc107551576)

[34. Подсистема управления памятью. Требования, предъявляемые к ней 57](#_Toc107551577)

[35. Виртуальная память. Задачи управления виртуальной памятью 58](#_Toc107551578)

[36. Задача замещения при управлении виртуальной памятью, часовой алгоритм 59](#_Toc107551579)

[37. Распределение памяти. Система двойников 61](#_Toc107551580)

[38. Copy on write 63](#_Toc107551581)

[39. Концепция Микроядерной ОС ***Перечитать!!!*** 64](#_Toc107551582)

[40. Типы операционных систем, их особенности 67](#_Toc107551583)

[41. Монолитные операционные системы, их особенности и недостатки 68](#_Toc107551584)

[42. Гибридные ОС 69](#_Toc107551585)

## 1. Основы ОС UNIX, возможности

Операционная система UNIX проектировалась как инструментальная система для разработки программного обеспечения. Своей уникальностью система обязана во многом тому обстоятельству, что она была, по сути, создана всего двумя разработчиками, причем создававшие ее люди делали систему для себя, и первое время ее использовали на мини-ЭВМ с очень скромными вычислительными ресурсами.

Отличительные черты:

* Система написана на языке высокого уровня (как следствие легкопереносима)
* Многозадачная, многопользовательская система
* Наличие стандартов
* Простой, но мощный пользовательский интерфейс
* Использование единой иерархической файловой системы
* Достаточно большое количество приложений
* Минимальное количество функций и простота
* Общность: одни и те же методы и механизмы должны были использоваться во многих случаях
  + обращения к файлам, устройствам ввода/вывода и буферам межпроцессных сообщений выполняются с помощью одних и тех же примитивов
  + одни и те же механизмы именования, присвоения альтернативных имен и защиты от несанкционированного доступа применяются к файлам с данными и директориями и устройствам
  + одни и те же механизмы работают в отношении программно и аппаратно инициируемых прерываний
* решение простых задач комбинированием небольших программ
* независимая от устройств файловая система

После запуска ОС в ней создаются несколько зарегистрированных пользователей:

* Root (суперпользователь, неограниченный права)
* Adm (суперпользователь, владеющий файловой системой)
* Bin (суперпользователь, владелец всех команд)
* Cron (пользователь-планировщик)
* Lp (отвечает за ввод-вывод на принтеры)

***Возможности:***

К основным функциям операционной системы UNIX можно отнести следующее:

1. Обработка прерываний.
2. Создание и уничтожение процессов.
3. Синхронизация процессов.
4. Организация взаимодействия между процессами.
5. Переключение процессов из одного состояния в другое.
6. Диспетчеризация.
7. Приостановка и активизация процессов.
8. Манипулирование блоками управления процессами.
9. Поддержка операции ввода-вывода.
10. Поддержка операции распределения и перераспределения памяти.
11. Поддержка работы файловых систем.
12. Поддержка механизма вызова-возврата по обращению к процедурам

## 2. Понятие системного вызова в ОС UNIX. Обработка ошибок

**Системные вызовы** преобразуют процесс, работающий в режиме пользователя, в защищенный процесс, работающий в режиме ядра. Это позволяет процессу вызывать защищенные процедуры ядра для выполнения системных функций.

Системные вызовы обеспечивают программный интерфейс для доступа к процедурам ядра. Они обеспечивают управление системными ресурсами, такими как память, пространство на дисках и периферийные устройства. Системные вызовы оформлены в виде библиотеки времени выполнения. Многие системные вызовы доступны через командный интерпретатор.

Системные вызовы обеспечивают:

* сопоставление действий пользователя с запросами драйверов устройств;
* управление процессами;
* реализацию операций ввода-вывода;
* управление файловой системой, файлами и каталогами;
* поддержку функций терминала.

**Обработка ошибок:**

Обычно в случае возникновения ошибки системные вызовы возвращают –1 и устанавливают значение переменной errno, указывающее возникновение ошибки. Библиотечные функции как правило значение errno не устанавливают, а код возврата различен для различных функций. Для уточнения возвращаемого значения библиотечной функции необходимо обратиться к электронному справочнику man(1). Переменная errno не обнуляется следующим нормально завершившимся системным вызовом, значит эту переменную следует анализировать сразу же после системного вызова, который завершился с ошибкой.

Имеются также две функции, помогающие сообщить причину ошибочной ситуации.

Функция **strerror** принимает в качестве аргумента номер ошибки и возвращает строку, содержащую сообщение о причине ошибочной ситуации.

Функция **perror** выводит в стандартный поток сообщений об ошибках информацию об ошибочной ситуации, основываясь на значении переменной errno. Строка s, передаваемая функции в качестве аргумента, предваряет сообщение об ошибке и может содержать дополнительную информацию, например, название функции, в которой произошла ошибка.

## 3. Особенности современных операционных систем

На современном этапе развития операционных систем на передний план вышли **средства обеспечения безопасности**. Это связано с возросшей ценностью информации, обрабатываемой компьютерами, а также с *повышенным уровнем угроз*, существующих при передаче данных по *сетям*, особенно по публичным, таким как Интернет. Многие операционные системы обладают сегодня развитыми средствами защиты информации, основанными на шифрации данных, аутентификации и авторизации.

Современным операционным системам присуща **кроссплатформенность**, т.е. способность работать на совершенно различных типах компьютеров.

В последние годы получила дальнейшее развитие долговременная тенденция **повышения удобства работы человека с компьютером**. *Эффективность работы человека становится основным фактором*, определяющим эффективность вычислительной системы в целом. Усилия человека не должны тратиться на настройку параметров вычислительного процесса, как это происходило в ОС предыдущих поколений.

Современная операционная система берет на себя выполнение задачи *выбора параметров операционной среды*, используя для этой цели различные адаптивные алгоритмы. Даже в процессе установки большинство ОС предлагают режим выбора параметров по умолчанию, который гарантирует пусть не оптимальное, но всегда приемлемое качество работы систем.

ОС предоставляют следующий перечень услуг:

* Разработка программ;
* Исполнение программ;
* Доступ к устройствам ввода-вывода;
* Контролируемый доступ к файлам;
* Системный доступ;
* Обнаружение ошибок и их обработка;
* Учет использования ресурсов;

## 4. Файлы и файловая система в ОС UNIX. Права доступа

**Файлы** в операционной системе Linux можно поделить на три основных типа:

* **Обычные файлы (-)**
  + могут содержать текст, исполняемые инструкции для программ, изображения или другую информацию. Примеры: текстовые файлы, исполняемые файлы, файлы изображений, файлы архивов, файлы библиотек программ
* **Каталог (d)**
  + Содержит таблицу индексных дескрипторов и названия файлов
  + **Теневой каталог**: нет разрешения на чтение, но если знать архитектуру каталога, можно управлять файлами в нем
* **Специальный файл устройства**
  + Блочные (b) обеспечивают буферизованный доступ к аппаратным компонентам;
  + Символьные не буферизованный доступ к аппаратным компонентам и ядру;
* **Файл FIFO или именованный канал(p)**
  + файлы, позволяющие настроить связь между двумя процессами перенаправив вывод одного процесса на вход другого;
  + неименованный канал не присутствует в файловой системе и недоступен во внешнем процессе, используется для межпроцессного взаимодействия внутри архитектуры процессов;
  + дочерний процесс – копия родительского.
* **Связь (link)(l)**
  + Файлы, которые указывают на другие файлы в системе по их имени;
  + Жесткая ссылка (адрес индексного дескриптора);
  + Мягкая ссылка (отдельный вид файла) является чем-то вроде ярлыка в windows.
* **Сокет (socket)(s)**
  + Файл, который может работать с протоколами TCP/UDP

**Файловая система**

* размещение и упорядочивание на носителе данных в виде файлов;
* определение максимально поддерживаемого объема данных на носителе информации;
* создание, чтение и удаление файлов;
* назначение и изменение атрибутов файлов;
* определение структуры файла;
* поиск файлов;
* организация каталогов для логической организации файлов;
* защита файлов при системном сбое;
* защита файлов от несанкционированного доступа и изменения их содержимого.

Задачи ФС:

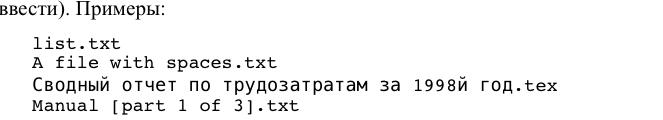
* присвоение имен файлам;
* программный интерфейс работы с файлами для приложений;
* отображение логической модели файловой системы на физическую организацию хранилища данных;
* поддержка устойчивости файловой системы к сбоям питания, ошибкам аппаратных и программных средств;
* содержание параметров файла, необходимых для правильного взаимодействия с другими объектами системы (ядро, приложения и пр.).

**Файловая система** в Unix организована в виде единого **дерева** каталогов от корневого каталога.

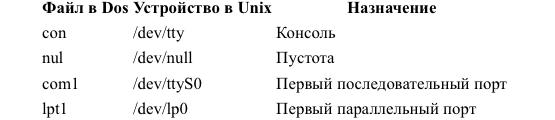
Каталоги разделяются косой чертой **"/"** (а не обратной, как в Dos), например, "/home/inpunix/ivanov/" – поддиректория "ivanov" в директории "inpunix" директории "home". Самая верхняя директория ("корневая") называется "/".

Правила именования:

1. Имя файла (и директории) может содержать практически любые символы, включая пробелы (и даже управляющие символы, типа перевода строки, если удастся их ввести). Примеры:



1. Символ "." не является особенным, и в имени может быть любое их количество, а может и не быть вовсе.
2. У программ не требуется какое-либо специальное расширение, типа .exe -- достаточно установить атрибут "x"
3. Заглавные и маленькие буквы различаются;
4. В отличие от Dos/Windows, нет зарезервированных имен, таких, как con, com1 и т.д. Вместо них есть соответствующие файлы устройств в директории /dev:



1. Длина имени может быть до 255 символов; полный путь (включая директории) – 1024 символа.

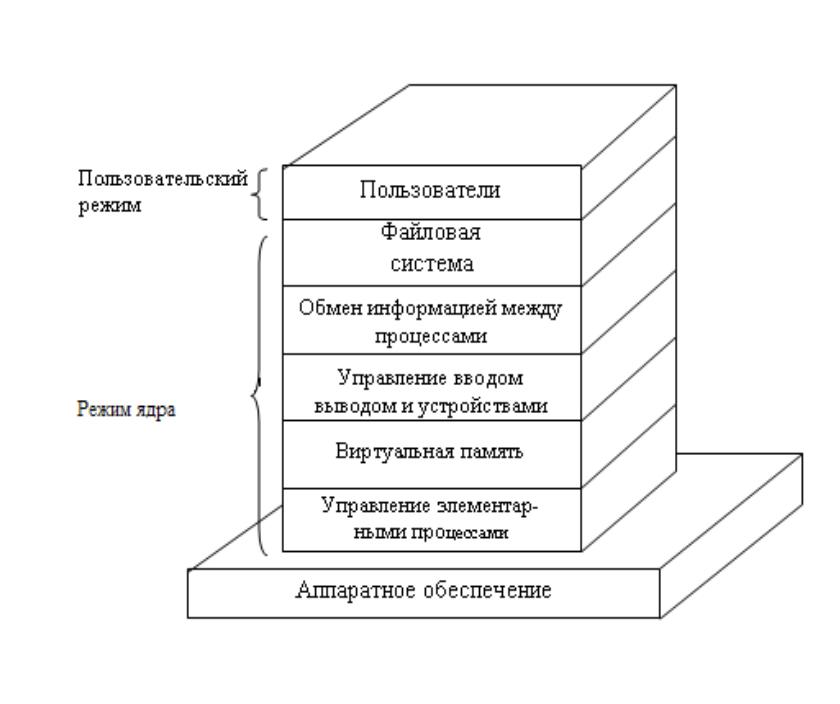
Хотя **единственный символ**, которого не может быть в имени, это "/" (поскольку это разделитель директорий), лучше не слишком увлекаться, и стараться не делать имен с пробелами, и не злоупотреблять различием в регистре букв -- это только внесет лишнюю путаницу. Также лучше не называть файлы по-русски.

В каждой директории есть поддиректория ".", **являющаяся ссылкой на саму директорию** и "..", **являющаяся ссылками директорию уровнем выше**.

Файлы, имена которых начинаются c точки (включая «.» и «..»), считаются "неинтересными", и в листинге директории обычно не показываются.

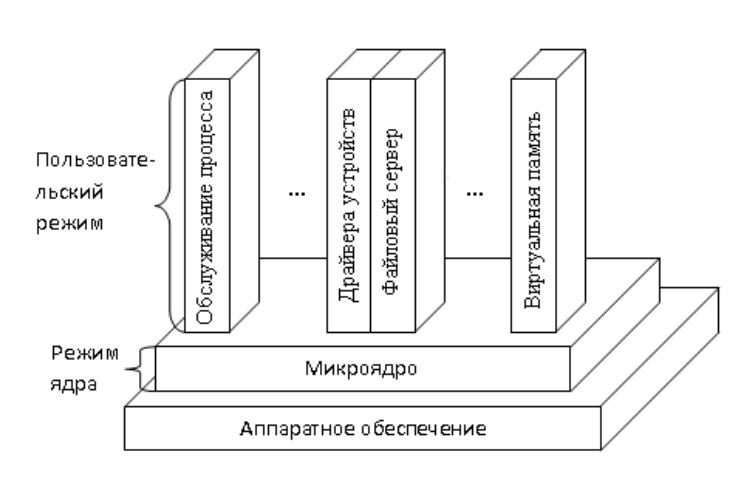
## 5. Концепция слоистой ОС и системы на основе микроядра

Слоистые ОС основаны на *иерархической организации функций*. Причем взаимодействие возможно только с функциями, находящимися *на соседних уровнях*.



Такой подход позволяет строить более *надёжные ОС*, позволяющие осуществлять модификацию, однако если потребуется ввести новые функции в слой, то необходимо вносить изменения и в соседние слои для возможности обращения к этим функциям. *Другая проблема* – это проблема безопасности, т. к. между слоями много точек обмена.

Основой концепции ОС на основе микроядра является *горизонтальное расположение уровней*, а приложения и услуги не являющиеся критическими, работают в пользовательском режиме. И хотя выбор того, что должно располагается в ядре, а что выносится за его пределы, зависит от архитектуры системы, общая тенденция такова, что многие службы, которые раньше размещались в ядре, теперь располагаются на уровне внешних подсистем, которые взаимодействуют с ядром и друг с другом, т. е. в *ядре должны располагаться только самые важные функции*. К таким подсистемам относятся драйвера устройств, файловые системы, менеджер виртуальной памяти, системы управления окнами, служба безопасности.



Процессы выполняются на пользовательском уровне, взаимодействуя между собой как *равноправные*, и обычно взаимодействие осуществляется с помощью *обмена сообщениями*, которые передаются через микроядро. Микроядро выступает в роли посредника: оно подтверждает правильность сообщений, передаёт их от одного компонента другому, предоставляет доступ аппаратному обеспечению. *Микроядро выполняет и защитные функции*: оно не пропускает сообщение, если такой обмен не разрешен.

*Достоинства концепции микроядра*:

1. **Единообразные интерфейсы**. Используются для запросов, генерируемых процессами. Процессам не нужно различать приложения, выполняемые на уровне ядра и на пользовательском уровне, т.к. доступ ко всем службам осуществляется только с помощью передачи сообщений.
2. **Расширяемость**. Имеется возможность добавлять в ОС новые услуги или сервисы, а также обеспечивать множественную реализацию сервисов в одной и той же функциональной области. Например, можно организовывать несколько различных способов хранения файлов на дисках. Добавление новых служб требует модификации лишь некоторых других служб и не требует изменений микроядра.
3. **Гибкость**. Допускается также удаление некоторых служб для получения компактной ОС. Если пользователю не требуются некоторые подсистемы он может скомпоновать ОС под свои нужды.
4. **Переносимость**. Программный код, взаимодействующий с аппаратными средствами, или большая его часть в микроядре, что уменьшает объем работ переноса на новые аппаратные средства.
5. **Надежность**. Чем больший код имеет программа, тем труднее ее протестировать. Однако небольшое ядро ОС можно тщательно проверить. А небольшое число интерфейсов прикладного программирования позволяет реализовать подсистемы, работающие вне ядра, с достаточно качественным программным кодом. Имея стандартизованный набор интерфейсов, разработчик отдельного приложения не может повлиять на другие системные компоненты.
6. Микроядро способствует поддержке **распределенных ОС**. Сообщение, которое передается от обслуживаемых сервисов к обслуживающим должно содержать идентификатор запрашиваемой услуги. Если система такова, что все процессы и сервисы обладают в ней уникальным идентификатором, то на уровне микроядра образуется единый образ системы. Процесс может отправлять сообщение, не зная, на какой именно машине выполняется приложение, к которому он обращается.
7. Подход на основе микроядра **хорошо функционирует среди объектно-ориентированных ОС**. Такой подход способствует более строгой разработке ядра и модульных расширений ОС. Перспективным считается подход, в котором сочетается архитектура с микроядром, принципы объектно-ориентрованных систем, которые реализуются с использованием компонентов. Компоненты – объекты с четко заданными интерфейсами, которые могут объединяться, образуя программу по принципу блоков.

*Недостатки*:

1. **Низкая производительность**: создание сообщения и отправка его через микроядро с последующим получением и декодированием ответа занимает больше времени, чем непосредственно вызов сервиса.
2. **Многое в обеспечении производительности зависит от функциональных возможностей ядра**: избирательное увеличений функциональности ядра приводит к уменьшению количества переключений между режимами пользовательским и ядра.

## 6. Владельцы файлов. Права доступа к файлам. Атрибуты файлов

**Владельцы файлов:**

1. Класс владельца-пользователя файла.
2. Класс членов группы, являющейся владельцем файлов.
3. Класс остальных пользователей, кроме суперпользователя.

**Владелец -** набор прав для владельца файла, пользователя, который его создал или сейчас установлен его владельцем. Обычно владелец имеет все права, чтение, запись и выполнение.

**Группа** - любая группа пользователей, существующая в системе и привязанная к файлу. Но это может быть только одна группа и обычно это группа владельца, хотя для файла можно назначить и другую группу.

**Остальные** - все пользователи, кроме владельца и пользователей, входящих в группу файла.

**Root** - имеет доступ ко всему.

**Права доступа к файлу:**

1. Чтение (r): разрешает получать содержимое файла, но на запись нет. Для каталога позволяет получить список файлов и каталогов, расположенных в нем
2. Запись (w): разрешает записывать новые данные в файл или изменять существующие, а также позволяет создавать и изменять файлы и каталоги
3. Выполнение (x): флаг для идентификации системой исполняемого файла

**Дополнительные атрибуты доступа:**

* t - сохранить образ выполняемого файла в оперативной памяти после завершения (применялся раньше, когда доступ к памяти был долги и трудоемким);
* s - установить пользовательский идентификатор процесса при выполнении (т.е. выполнять под именем определенного пользователя всегда);
* s - установить групповой идентификатор процесса при выполнении;
* 1 - установить обязательное блокирование файлов при выполнении

**Специальные права доступа:**

Для того, чтобы позволить обычным пользователям выполнять программы от имени суперпользователя без знания его пароля была придумана такая вещь, как SUID и SGID биты.

1. SUID
2. SGID
3. Sticky-bit

**SUID** - если этот бит установлен, то при выполнении программы, id пользователя, от которого она запущена заменяется на id владельца файла. Фактически, это позволяет обычным пользователям запускать программы от имени *суперпользователя.*

**SGID** - этот флаг работает аналогичным образом, только разница в том, что пользователь считается членом группы, с которой связан файл, а не групп, к которым он действительно принадлежит. Если SGID флаг установлен на каталог, все файлы, созданные в нем, будут связаны с группой каталога, а не пользователя. Такое поведение используется для организации общих папок.

**Sticky-bit** - этот бит тоже используется для создания общих папок. Если он установлен, то пользователи могут только создавать, читать и выполнять файлы, но не могут удалять файлы, принадлежащие другим пользователям.

**Просмотр прав файла** осуществляется с помощью команды ls -l

## 7. Функции микроядра

Принципы микроядра:

* Исключительные права на привилегированный режим
* Принцип разделения политики и механизмов (механизмы не должны диктовать или ограничивать политики, в соответствии с которыми принимаются решения о том, какие операции разрешить и какие ресурсы выделять)
* Принцип минимализма
* Микроядро как драйвер процессора

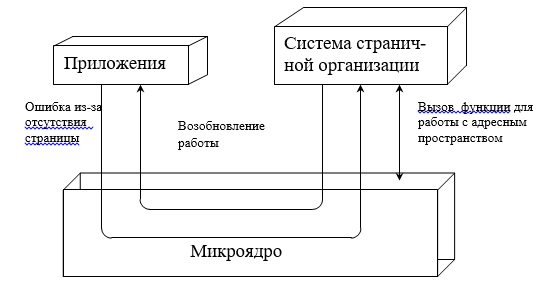
Функции микроядра:

* Механизмы управления памятью, такие как адресные пространства, должны быть включены в микроядро. Они также содержат функции защиты памяти.
* Механизмы планирования процессора должны содержать планировщики процессов и потоков;
* Межпроцессное взаимодействие, управление серверами.
* Управление вводом-выводом и прерываниями.

*Низкоуровневое управление памятью*

Микроядро должно отвечать лишь за отображение виртуальной страницы на физическую страницу.

Блок управления памятью, включая систему защиты адресного пространства одного процесса от другого, а также алгоритм замены страниц и другие логические схемы страничной организации памяти можно реализовать вне ядра.



При обращении приложения к странице, которая отсутствует в основной памяти, возникает прерывание из-за отсутствия страницы и управление перехватывается ядром. Ядро отправляет системе страничной организации памяти сообщение, в котором указывается запрашиваемая страница. Система страничной организации памяти может принять решение о загрузке данной страницы в оперативную память и выделения для этого физической страницы. Система страничной организации и ядро взаимодействуют между собой, чтобы логические операции, выполняемые системой страничной организации памяти, отображались в физическую память. Как только нужная страница станет доступна, то система страничной организации отправляет через микроядро сообщение приложению о том, что приложение может продолжить работу.

Модуль виртуальной памяти принимает решение, когда загружать страницу в память, и какую из страниц, находящихся в памяти, следует заменить.

Операции поддержки внешних систем страничной организации памяти и управления внешней памятью оставшиеся в ядре:

* **Предоставление**: процесс, который владеет адресным пространством, может предоставлять некоторые свои страницы другому процессу. Ядро удаляет эти страницы из адресного пространства первого процесса и передает их второму процессу.
* **Отображение**: процесс может отображать любые свои страницы в адресное пространство другого процесса. После чего оба процесса будут иметь доступ к этим страницам, то есть создаётся общая область памяти. Ядро не меняет информации о принадлежности страниц первому процессу, но выполняет отображение, предоставляя другому процессу доступ к этим страницам.
* **Восстановление**: процесс может восстановить любые страницы, предоставленные другим процессам или отображенные в их адресное пространство.

*Взаимодействие между процессами*

Основной формой взаимодействия между процессами (потоками) являются *сообщения*. Сообщения включают в себя заголовок, в котором указаны идентификаторы процесса отправителя и процесса получателя, а также указатель на блок данных и некоторые управляющие сведения о процессе. Если адресное пространство в процессах не перекрывается, то передача сообщения от одного процесса другому — это копирование одной области памяти в другую.

Взаимодействие между процессами основывается на относящихся к этим процессам портам. *Порт* — это очередь сообщений, предназначенная для определенного процесса. С портом связан список возможностей, в которых указано с какими процессами данный процесс может обмениваться информацией. Процесс может разрешить доступ к себе, отправив в ядро сообщение, в котором указана новая возможность порта.

*Управление вводом-выводом и прерываниями*

В микроядре имеется возможность обрабатывать аппаратные прерывания подобно сообщениям и включать в адресное пространство порта ввода-вывода. Микроядро распознает прерывания, но само не обрабатывает их. Оно генерирует сообщение процессу, работающему на пользовательском уровне и связанному с данным прерыванием, т. е. ядро преобразует прерывание в сообщение, но само в обработке аппаратно-зависимых прерываний не участвует.

В некоторых системах предлагается рассматривать аппаратное обеспечение как набор потоков, которые обладают своими идентификаторами и отправляют сообщение, состоящее из идентификатора данного потока соответствующим потокам в пользовательских программах. Поток-получатель выясняет, является ли полученное сообщение прерыванием.

## 8. Неименованные каналы ОС UNIX

Неименованный канал создается вызовом **pipe**, который заносит в массив int[2] два дескриптора открытых файлов. fd[0] — открыт на чтение, fd[1] — на запись. Канал уничтожается, когда будут закрыты все файловые дескрипторы, ссылающиеся на него.

В рамках одного процесса **pipe** смысла не имеет, передать информацию о нем в произвольный процесс нельзя (имени нет, а номера файловых дескрипторов в каждом процессе свои). Единственный способ использовать **pipe** — унаследовать дескрипторы при вызове **fork**. После вызова **fork** канал окажется открытым на чтение и запись в родительском и дочернем процессе, т.е. теперь на него будут ссылаться 4 дескриптора. Теперь надо определиться с направлением передачи данных — если надо передавать данные от родителя к потомку, то родитель закрывает дескриптор на чтение, а потомок — дескриптор на запись.

Оставлять оба дескриптора незакрытыми плохо по двум причинам:

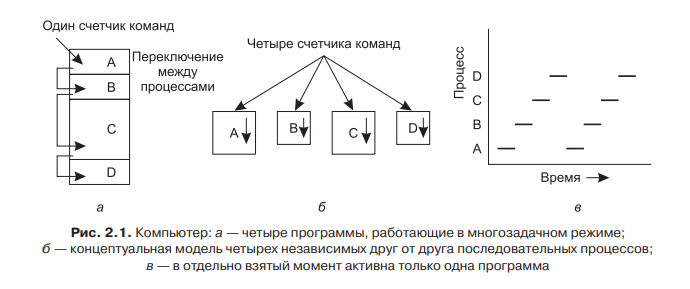
* Родитель после записи не может узнать считал ли дочерний процесс данные, а если считал, то сколько. Соответственно, если родитель попробует читать из **pipe**, то, вполне вероятно, он считает часть собственных данных, которые станут недоступными для потомка.
* Если один из процессов завершился или закрыл свои дескрипторы, то второй этого не заметит, так как **pipe** на его стороне по-прежнему открыт на чтение и на запись.

## 9. Понятие процесса, модели процессов

**Процесс** — это экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных. Концептуально у каждого процесса есть свой, виртуальный, центральный процессор. Постоянное переключение между процессами, выполняемое ЦП, называется **мультипрограммированием**, или **многозадачным режимом работы**.

На рис. 2.1, а показан компьютер, работающий в многозадачном режиме и имеющий в памяти четыре программы. На рис. 2.1, б показаны четыре процесса, каждый из которых имеет собственный алгоритм управления (то есть собственный логический счетчик команд) и работает независимо от всех остальных. Понятно, что на самом деле имеется только один физический счетчик команд, поэтому при запуске каждого процесса его логический счетчик команд загружается в реальный счетчик. Когда работа с процессом будет на некоторое время прекращена, значение физического счетчика команд сохраняется в логическом счетчике команд, размещаемом процессом в памяти. На рис. 2.1, в показано, что за довольно длительный период наблюдения продвинулись вперед все процессы, но в каждый отдельно взятый момент времени реально работает только один процесс.

Поскольку центральный процессор переключается между процессами, скорость, с которой процесс выполняет свои вычисления, не будет одинаковой и, скорее всего, не сможет быть вновь показана, если тот же процесс будет запущен еще раз. Поэтому процессы не должны программироваться с использованием каких-либо жестко заданных предположений относительно времени их выполнения.



Традиционная модель процесса, имеющаяся в Linux, — это разветвление (с помощью команды fork) для создания нового процесса, за которым следует команда exec для инициализации этого процесса кодом, предназначенным для выполнения, с последующим запуском его на выполнение. За управление этим выполнением отвечает оболочка, разветвляя и выполняя процессы, необходимые для работы команд оболочки. Когда происходит выход из этих команд, Linux удаляет процесс.

## 10. Метаданные (атрибуты) файлов в ОС UNIX (?)

Определение *метаданных* является "данными о других данных." С файловой системой данные содержатся в его файлах и каталогах, и метаданные отслеживают информацию о каждом из этих объектов: действительно ли это - регулярный файл, каталог, или ссылка? Каков ее размер, дата создания, последняя измененная дата, владелец файла, групповой владелец, и права доступа?

Метаданные файловой системы обычно упоминаются как его *атрибуты файла*.

**Атрибуты файла**:

Технически это набор из девяти основных битов. Определяющих какие из *пользователей* обладают правами на чтение, запись. А также запуск файлов для выполнения. Данный *набор* формирует код, называемый режимом доступа к файлу/каталогу. Первые три бита определяют права доступа для владельца. Следующие — для группы пользователей, к которой относится файл. Последние три бита — права доступа для всех остальных пользователей в системе. На практике это выглядит так:



(Все что выводит ls, это атрибуты, то есть права, владелец, группа, размер, дата, время)

Этот вывод говорит о том, что файл .*mysql\_history* доступен для чтения и записи только его владельцу — пользователю john. О чём свидетельствуют первые три символа (rw-) в режиме доступа. Символ «-», стоящий перед ними — это признак, что это файл, а не каталог, блочное устройство и т. д.

Существует также ещё *четыре дополнительных бита*. Которые определяют *тип самого файла* и задаются непосредственно при создании файла. С помощью команды *chmod* можно менять основные (и некоторые дополнительные) биты режима доступа. Сделать это может только владелец файла или суперпользователь. Просматривать атрибуты (в том числе и режимы доступа) позволяет команда *ls*.

## 11. Описание процесса, таблицы процесса (Реализация процесса Таненбаум)

Две концепции: владение ресурсами и выполнение процесса.

*Владение ресурсами*

Процесс включает **виртуальное адресное пространство**, в котором содержится образ процесса, и на протяжении каких-то интервалов времени может владеть такими ресурсами, как основная память, каналы и устройства ввода/вывода, файлы, или на некоторое время получать контроль над ними. ОС выполняет защитные функции, предотвращая нежелательное взаимодействие процессов в сфере распределения ресурсов.

*Планирование выполнения*

Выполнение процесса осуществляется путём выполнения кода одной или нескольких программ, причём выполнение процессов может чередоваться друг с другом.

Процесс имеет такие параметры, как:

* состояние процесса;
* текущий приоритет, в соответствии с которым происходит планирование выполнения процесса.

Для реализации модели процесса ОС содержит таблицу процессов (массив структур) с одним элементом (записью) для каждого процесса. Элемент таблицы содержит информацию о состоянии процесса, а также иную информацию, которую необходимо сохранять при переключении в состояние готовности или блокировки для последующего запуска.

Наиболее важные поля элемента таблицы процессов:

|  |  |  |
| --- | --- | --- |
| **Управление процессом** | **Управление памятью** | **Управление файлами** |
| Регистры | Указатель на информацию о **текстовом сегменте** | Корневой каталог |
| Счётчик команд | Указатель на информацию о **сегменте данных** | Рабочий каталог |
| Слово состояния программы | Указатель на информацию о **сегменте стека** | Дескрипторы файлов |
| Указатель стека |  | Идентификатор пользователя |
| Состояние процесса |  | Идентификатор группы |
| **Приоритет** |  |  |
| Параметры планирования |  |  |
| **Идентификатор процесса** |  |  |
| **Родительский процесс** |  |  |
| **Группа процесса** |  |  |
| Сигналы |  |  |
| Время запуска процесса |  |  |
| Использованное время процесса | |  |
| Время процесса, использованное дочерними процессами | |  |
| Время следующего аварийного сигнала | |  |

## 12. Процессы. Типы процессов

Процессы в UNIX можно разделить на следующие типы:

* **Системные** (vmdaemon, pagezero, bufdaemon, syncer)
  + являются частью ядра и всегда расположены в оперативной памяти
  + не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы
  + выполняемые инструкции и данные этих процессов находятся в ядре системы, таким образом, они могут вызывать функции и обращаться к данным, недоступным для остальных процессов
* **Демоны/сервисы** (usbd, httpd, sshd)
  + запускаются при инициализации системы, но после инициализации ядра и обеспечивают работу различных подсистем UNIX: системы терминального доступа, системы печати, сетевых служб и т.д.
  + не связаны ни с одним пользователем. Большую часть времени демоны ожидают, пока тот или иной процесс запросит определенную услугу.
* **Интерактивные/прикладные процессы** (ls, sh, fsck …)
  + относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы.
* **Процесс init**
  + начальный командный интерпретатор, который обеспечивает выполнение команд пользователя в системе UNIX.

## 13. Атрибуты процесса ОС UNIX

Атрибуты позволяют ОС эффективно управлять работой процесса. Просмотр атрибутов процесса: ps -ef.

Идентификатор процесса

Process ID (PID) — каждый процесс имеет уникальный идентификатор, позволяющий ядру системы различать процессы. При создании нового процесса, ядро присваивает ему следующий свободный идентификатор. Присвоение PID — по возрастающей, то есть PID нового процесса больше, чем PID процесса, созданного перед ним. Если PID достиг максимального значения, следующий процесс получит минимальный свободный и цикл повторяется. Когда процесс завершает работу — ядро освобождает занятый им PID.

Родительский процесс

Parent Process ID (PPID) — PID процесса, породившего данный.

Приоритет процесса

Приоритет процесса (nice number) — относительный приоритет процесса, учитываемый планировщиком при определении очередности запуска. Чем меньше число, тем больше приоритет (чем более “приятный” процесс, тем меньше он загружает CPU). Фактическое распределение ресурсов — приоритет выполнения: динамически изменяется ядром во время выполнения. **Относительный** — постоянен, но может изменяться администратором или пользователем с помощью nice.

Терминальная линия

Терминальная линия (TTY) — терминал или псевдотерминал, ассоциированный с процессом (демоны не имеют ассоциированного терминала).

Идентификаторы пользователей

Реальный (RID) и эффективный (EUID) идентификаторы пользователя. RID — идентификатор пользователя, запустившего этот процесс. EUID служит для определения прав доступа процесса к системным ресурсам.

Идентификаторы групп

Реальный (RGID) и эффективный (EGID) идентификаторы группы. RGID — GID первичной группы пользователя, запустившего процесс. EGID служит для определения прав доступа пользователя по классу доступа группы.

## 14. Концепция потока, как составной части процесса

Применение потоков:

1. Во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в **квазипараллельном** режиме. Важную роль имеет для приложений, где не подходит использование процессов в виду из раздельного адресного пространства.
2. **Легкость (быстрота) создания** потоков по сравнению с тяжеловесными процессами.
3. **Производительность**: когда потоки работают в рамках одного центрального процессора, они не приносят никакого роста производительности, но, когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.
4. Весьма **полезны для систем с несколькими ЦП**, где есть реальная возможность параллельных вычислений.

У потока есть **счетчик команд** (отслеживает, какую очередную инструкцию нужно выполнять), **регистры** (которые содержат текущие рабочие переменные), **стек с протоколом выполнения** (содержит по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры). Хотя поток может быть выполнен в рамках какого-нибудь процесса, сам поток и его процесс являются разными понятиями и должны рассматриваться по отдельности. Процессы используются для группировки ресурсов в единое образование, а потоки являются «сущностью», распределяемой для выполнения на центральном процессоре.

Потоки добавляют к модели процесса возможность реализации нескольких в значительной степени независимых друг от друга выполняемых задач в единой среде процесса. Наличие нескольких потоков, выполняемых параллельно в рамках одного процесса, является аналогией наличия нескольких процессов, выполняемых параллельно на одном компьютере. В первом случае потоки используют единое адресное пространство и другие ресурсы. А в последнем случае процессы используют общую физическую память, диски, принтеры и другие ресурсы. Поскольку потоки обладают некоторыми свойствами процессов, их иногда называют **облегченными процессами**. Термин «многопоточный режим» также используется для описания ситуации, при которой допускается работа нескольких потоков в одном и том же процессе. Некоторые центральные процессоры обладают непосредственной аппаратной поддержкой многопоточного режима и проводят переключение потоков за наносекунды.

Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости. В отличие от различных процессов, которые могут принадлежать различным пользователям и которые могут враждовать друг с другом, один процесс всегда принадлежит одному и тому же пользователю.

|  |  |
| --- | --- |
| **Элементы, присущие каждому процессу** | **Элементы, присущие каждому потоку** |
| Адресное пространство | Счетчик команд |
| Глобальные переменные | Регистры |
| Открытые файлы | Стек |
| Дочерние процессы | Состояние |
| Необработанные дочерние сигналы |  |
| Сигналы и обработчики сигналов |  |
| Учетная информация |  |

## 15. Разработка программ в ОС UNIX. Обработка ошибок, переменные окружения

Выполнение программы начинается с создания в памяти её образа и связывания с процессом структур ядра ОС, инициализации и передачи управления инструкциям программы. Завершение программы приводит к освобождению памяти и соответствующих структур ядра. **Образ программы** в памяти содержит сегменты инструкций и данных, созданных компилятором, а также стек, которым пользуется программа в ходе выполнения.

**Окружение** — это область, которую оболочка создает каждый раз при запуске сеанса, содержащего переменные, определяющие системные свойства. Например, это может быть часовой пояс в системе, пути к определенным файлам, приложения по умолчанию, локали и многое другое. Переменные окружения также могут использоваться в программах оболочки или в подоболочках для выполнения различных операций.

Переменные имеют следующий формат:

KEY=value

KEY="Some other value"

KEY=value1:value2

Должны соблюдаться следующие правила:

* Переменные окружения должны быть написаны большими буквами.
* Несколько значений переменных разделяются «:»
* Вокруг символа «=» нет пробела

Команды: **env** или **printenv** для отображения переменных окружения.

**Окружение** – набор переменных и их значений, с помощью которых программе можно передавать данные общего назначения.

Описание головной функции программы может выглядеть так:

main (int argc, char \*argv[], char \*envp[]);

или

extern char \*\*environ; // глобальная переменная, указывающая на строки с

// переменными окружения.

main (int argc, char \*argv[]);

argc равен количеству аргументов;

argv - массив указателей на аргументы;

envp - массив указателей на цепочки символов, образующие окружение.

Принято соглашение, по которому значение argc не меньше 1, а первый элемент массива argv указывает на цепочку символов, содержащую имя нового выполняемого файла.

Перед началом выполнения любой программы во внешнюю переменную environ помещается адрес массива указателей на цепочки символов, образующие окружение процесса. С помощью этой переменной (как и с помощью аргумента envp функции main) в новом процессе всегда можно получить доступ к окружению.

Каждая переменная содержит строку вида:

HOME=/home/student/851003.

Основные переменные окружения:

|  |  |
| --- | --- |
| **Название** | **Значение** |
| HOME | имя каталога, который становится текущим после входа пользователя в систему |
| PATH | последовательность полных файловых путей, разделенных двоеточием, которые используются некоторыми программами для поиска файлов, заданных относительным именем |
| TERM | тип терминала |
| TZ | информация о часовом поясе |
| LOGNAME | регистрационное имя пользователя |
| LC\_NUMERIC | значение категории локализации, определяющей правила национального представления чисел с плавающей точкой |
| LC\_TIME | значение категории локализации, определяющей правила национального представления времени и даты |

Для получения и установки значений переменной окружения применяются две функции: ***getenv*** возвращает значение переменной окружения, имеющей имя name; ***putenv*** помещает переменную и ее значение в окружение программы. Переменные окружения позволяют передать программе некоторую информацию, однако большую часть информации программа получает от пользователя в ходе ее выполнения. Вводимое новое значение переменной окружения будет действительно только для данного процесса и порожденных им процессов. После завершения текущего процесса измененная переменная окружения принимает исходное значение.

Можно модифицировать переменные окружения:

#include<stdlib.h>

Int setenv(const char \* name, const char \* value, int ov );

Int unsetenv(const char \* name);

**Setenv** – добавляет новую переменную окружения или заменяет уже существующую переменную окружения заданную первым аргументом, присваивает ей значение, заданное вторым аргументом, третий аргумент – это флаг, показывающий нужно ли перезаписывать переменную, если она уже существует (если 0, то не перезаписывается, в противном случае изменяется). При успешном завершении возвращает 0 и -1 при ошибке.

**Unsetenv** – удаляет переменную, заданную в начале аргумента из окружения. При успешном завершении возвращает 0 и -1 при ошибке.

**Обработка ошибок**

Обычно ошибки делят на следующие типы:

1) ошибки входных данных;

2) непредвиденные ошибки;

3) скрытые ошибки;

4) унаследованные ошибки.

perror, errno, sys\_errlist, sys\_nerr - системные сообщения об ошибках

Обычно в случае возникновения ошибки системные вызовы возвращают -1 и устанавливают значение переменной errno, указывающее возникновение ошибки. Библиотечные функции, как правило, значение errno не устанавливают, а код возврата различен для различных функций. Переменная errno не обнуляется следующим нормально завершившимся системным вызовом, следовательно, эту переменную следует анализировать сразу же после системного вызова, который завершился с ошибкой. Имеются также две функции, помогающие сообщить причину ошибочной ситуации.

Функция **strerror** принимает в качестве аргумента номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации.

#include <errno.h>

Void perror (const char \*s);

extern int errno;

extern char \*sys\_errlist [];

extern int sys\_nerr;

Функция **perror** формирует в стандартном протоколе ошибок сообщение, которое описывает последнюю ошибку, случившуюся во время выполнения системного вызова или библиотечной функции. Код ошибки извлекается из внешней переменной errno, которой он присваивается при возникновении ошибки, и которая не изменяется вызовами, не приведшими к ошибке. Строка s, передаваемая функции в качестве аргумента, предваряет сообщение об ошибке и может содержать дополнительную информацию, например, название функции, в которой произошла ошибка. Чтобы упростить нестандартное форматирование сообщений, предоставляется массив текстов сообщений **sys**\_**errlist**; для получения текста сообщения без символа перевода строки можно использовать errno в качестве индекса в этом массиве. **Sys**\_**nerr** - число сообщений в массиве; это значение следует проверять, поскольку новые коды ошибок могут быть добавлены в систему без соответствующей коррекции сообщений.

|  |  |
| --- | --- |
| **Errno** | **Описание** |
| ENOENT | Не найден файл или каталог |
| ESRCH | Нет такого процесса |
| EIO | Ошибка ввода/вывода: может иногда иметь место во время вызова, который следует за вызовом, к которому на самом деле относится эта ошибка |
| ENXIO | Неверное устройство или адрес: Ввод/вывод в специальный файл ссылается на устройство, которого не существует |
| E2BIG | Длинный список аргументов: При обращении к системному вызову из семейства exec задан список аргументов, содержащий более 5120 байт |
| ENOEXEC | Неверный формат выполняемого файла: Сделан запрос на выполнение файла, который, хотя и имеет соответствующие права, не начинается с допустимого магического числа |
| EBADF | Неверный номер файла: Или дескриптор файла указывает на неоткрытый файл, или запрос на чтение (соответственно запись) сделан для файла, открытого только для записи (соответственно чтения) |
| ECHILD | Нет порожденных процессов: Системный вызов wait был выполнен процессом, не имеющим существующих порожденных процессов или процессов, которых он не дождался |
| EAGAIN | Слишком много процессов: Системный вызов fork завершился аварийно, так как системная таблица процессов полна или пользователю не разрешено более создавать новые процессы |
| ENOMEM | Исчерпано пространство: Ошибка может также иметь место, если набор секций команд данных и стека требует слишком много сегментных регистров или, если при вызове fork недостаточно места в области подкачки |
| EACCESS | Нарушение прав доступа |
| EFAULT | Недопустимый адрес: Произошло прерывание при попытке использования аргументов системного вызова |
| EBUSY | Какой-то элемент занят |

## 16. Файлы, отображаемые в память (3.5.7)

Идея состоит в том, что процесс может выдать системный вызов для отображения файла на какую-то часть виртуального адресного пространства процесса. В большинстве реализаций на момент отображения в память еще не введены никакие страницы, но поскольку мы имеем дело со страницами, они требуют постраничной организации с использованием дискового файла в качестве резервного хранилища. Когда процесс выходит из рабочего состояния или явным образом демонтирует отображение файла, все измененные страницы записываются обратно в файл на диске.

Отображаемые файлы предоставляют альтернативную модель для ввода-вывода. Вместо осуществления операций чтения и записи к файлу можно обращаться как к большому символьному массиву, находящемуся в памяти. В некоторых ситуациях программисты находят эту модель более удобной.

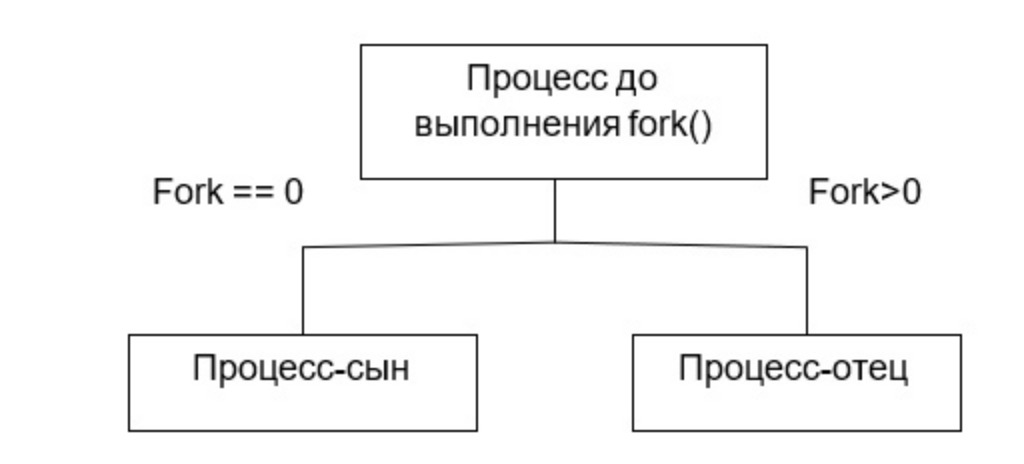
Если два или более процесса одновременно отображаются на один и тот же файл, они могут связываться посредством совместно используемой памяти. Запись, произведенная одним процессом в общую память, становится тут же видна, если другой процесс считывает данные из части своего виртуального адресного пространства, отображенного на файл. Таким образом, данный механизм предоставляет канал между двумя процессами, обладающий высокой пропускной способностью, и он довольно часто используется именно в этом качестве (вплоть до отображения рабочего файла). При доступности отображаемых на память файлов совместно используемые библиотеки могут воспользоваться этим механизмом.

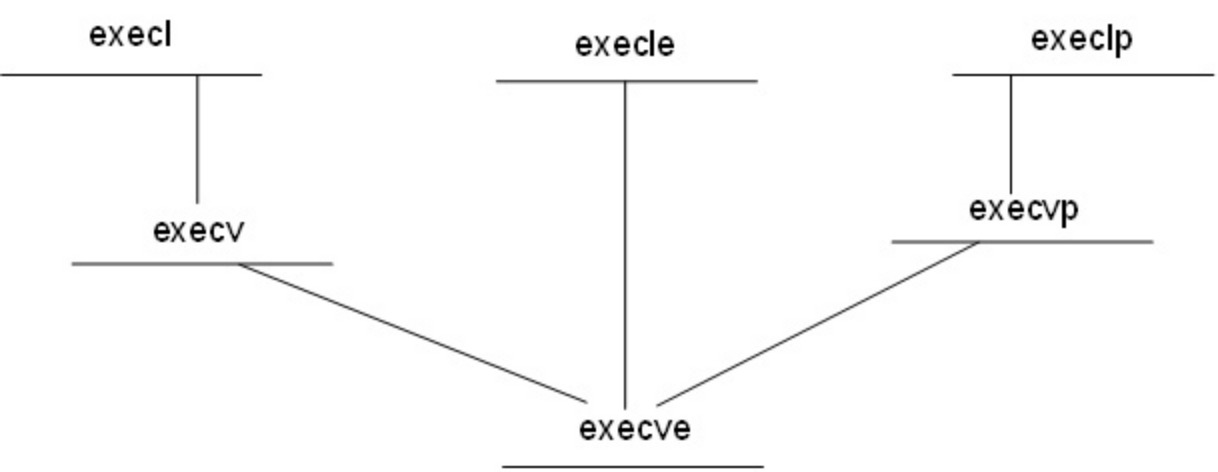
## 17. Процессы в ОС UNIX. Порождение процессов

Процессу, в отличие от fork, создающего порожденный процесс, который выполняется независимо от родительского, exec изменяет содержание выполняющегося процесса для выполнения другой программы. Вызов функции exec может быть неудачным. В случае ошибки возвращается значение –1 в вызывающий образ, а также обнаруживается ряд ошибок. Буква ‘p’ в вызовах exec указывает, что если значение аргумента file не начинается с ’/’, то есть корневого каталога, то функции при поиске надлежащего выполнению файла будут использовать переменную среды PATH. Для всех остальных функций exec фактическим значением их первого аргумента должно быть путевое имя файла.

После набора команды оболочка создает дочерний процесс, который должен выполнить команду пользователя. Он делает это, используя системный вызов execve, который полностью заменяет образ памяти процесса файлом, указанным в первом параметре.

В наиболее общем случае команда execve имеет три параметра: имя выполняемого файла, указатель на массив аргументов и указатель на массив переменных окружения. Эти параметры мы рассмотрим в дальнейшем. Различные библиотечные подпрограммы, включая execl, execv, execle и execve, предусматривают возможность пропуска параметров или указания их различными способами.





## 18. Сигналы в ОС UNIX. Их назначение и обработка

Сигналы:

* установление и обозначение сигналов в форме целочисленных значений;
* маркер в строке таблицы процессов для прибывших сигналов;
* таблица с адресами функций, которые определяют реакцию на прибывающие сигналы.

Реакция на сигналы:

* вызов собственной функции обработки;
* игнорирование сигнала
* использование предварительно установленной функции обработки по умолчанию.

Программные прерывания, предоставляющие механизм для обработки асинхронных событий. Такие события могут инициироваться пользователем (нажатие клавиш ctrl+c) или возникать вследствие действий в программе или ядре.

Отличительная особенность сигналов состоит в том, что они появляются асинхронно, и обработка их производится тоже асинхронно. Могут быть написаны функции обработки сигналов, эти функции регистрируются процессом в ядре операционной системы, а операционная система инициирует выполнение этих функций при получении соответствующих сигналов в соответствии с установленными правилами обработки.

Сигналы имеют определенный жизненный цикл:

1. состояние, связанное с генерированием (отправлением, поднятием) сигнала:

* generate //
* send // смысл: сигнал появился
* raise //

1. состояние хранения (store) – ядро операционной системы хранит его до тех пор, пока не появляется условие для доставки сигнала. И после того, как соответствующие условия создаются, ядро инициирует (обрабатывает) поступивший сигнал.

Ядро может выполнить 3 действия, связанные с обработкой сигнала:

1. Игнорирование сигнала. Сигнал направлен процессу, но действий нет. НО есть ряд сигналов, которые нельзя игнорировать:

* SIGKILL //связаны с остановкой или «убийством» процесса
* SIGSTOP

1. Выполнить действие по умолчанию.

Для всех сигналов предусмотрены действия, которые выполняются по умолчанию. Это или завершение процесса, или может быть действие игнорировать сигнал, захватить и обработать сигнал. Процесс прерывается, и запускается функция обработчик

1. Установление собственного обработчика

Написать функцию обработчик и зарегистрировать её. Но есть некоторые сигналы, которые нельзя описать самостоятельно. Ядро приостанавливает текущее выполнение процесса, и переходит к выполнению ранее зарегистрированной функции. После завершения выполнения функции ядро продолжает обрабатывать остановленный исходный процесс. Все сигналы имеют символическое имя. Все символические имена описаны в <signal.h> и начинаются с префикса SIG. Сигналы имеют номера (целочисленные). Общее число сигналов системы порядка 30.

Сигналы, требуемые POSIX (в большинстве систем linux есть дополнительные, но программы могут оказаться непереносимыми на другие версии и linux и UNIX):

|  |  |
| --- | --- |
| **Сигнал** | **Причина** |
| SIGABRT | Посылается, чтобы прервать процесс и создать дамп памяти |
| SIGALRM | Истекло время будильника |
| SIGFPE | Произошла ошибка при выполнении операции с плавающей точкой (например,  деление на 0) |
| SIGHUP | На телефонной линии, использовавшейся процессом, была повешена трубка.  Ядро отправляет лидеру сеанса, когда терминал сеанса отключается. Также этот сигнал отправляется всем процессам приоритетной группы, когда лидер сеанса завершается (пользователь выходит из системы) |
| SIGILL | Попытка выполнить неразрешенную инструкцию |
| SIGQUIT | Пользователь нажал клавишу, требующую выполнения дампа памяти |
| SIGKILL | Посылается, чтобы уничтожить процесс (не может игнорироваться или перехватываться) |
| SIGPIPE | Процесс пишет в канал, из которого никто не читает |
| SIGSEGV | Процесс обратился к неверному адресу памяти |
| SIGTERM | Вежливая просьба к процессу завершить свою работу |
| SIGUSR1 | Может быть определен приложением |
| SIGUSR2 | Может быть определен приложением |

**Системный вызов установки *обработчика* на сигнал**

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signo, sighandler\_t handler)

Позволяет зарегистрировать обработчик для сигнала, указанного в качестве первого аргумента. Второй аргумент - адрес функции обработчика. В качестве второго аргумента могут использоваться также:

* SIG\_DFL//восстановить поведение по умолчанию для сигнала signo.
* SIG\_IGN//игнорировать сигнал, указанный параметром signo

Функция signal() возвращает предыдущее поведение сигнала, которое может принимать вид указателя на обработчик сигнала, SIG\_DFL или SIG\_IGN. При ошибке возвращается значение SIG\_ERP. Переменная errno не устанавливается.

*int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);*

Системный вызов **sigaction** используется для изменения действий процесса при получении соответствующего сигнала. Параметр signum задает номер сигнала и может быть равен любому номеру, кроме SIGKILL и SIGSTOP. Если параметр act не равен нулю, то новое действие, связанное с сигналом signum, устанавливается соответственно act. Если oldact не равен нулю, то предыдущее действие записывается в oldact. Структура sigaction имеет следующий формат:

*Struct sigaction {*

*void (\*sa\_handler)(int);*

*void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);*

*sigset\_t sa\_mask;*

*int sa\_flags;*

*}*

sa\_handler задает тип действий процесса, связанный с сигналом signum, и может быть равен: SIG\_DFL для выполнения стандартных действий, SIG\_IGN для игнорирования сигнала,- или быть указателем на функцию обработки сигнала.

sa\_mask задает маску сигналов, которые должны блокироваться при обработке сигнала. Также будет блокироваться и сигнал, вызвавший запуск функции, если только не были использованы флаги SA\_NODEFER или SA\_NOMASK.

sa\_flags содержит набор флагов, которые могут влиять на поведение процесса при обработке сигнала. Так, SA\_SIGINFO - Обработчик сигнала требует 3-х аргументов, а не одного. В этом случае надо использовать параметр sa\_sigaction вместо sa\_handler.

Функция ожидания:

*#include <unistd.h>*

*int pause (void);*

После вызова функции pause вызывающий процесс (или подзадача) приостанавливается до тех пор, пока не получит сигнал. Данный сигнал либо остановит процесс, либо заставит его вызвать функцию обработки этого сигнала. **Сигнал должен отличаться от тех сигналов, которые игнорируются вызывающим процессом**. Если сигнал привел к завершению выполнения вызывающего процесса, то возврата из pause не будет.

Когда порождается новый процесс, для него устанавливается поведение всех сигналов по умолчанию, если только родительский процесс не игнорирует какие-то сигналы. Таким образом, новый процесс выполняет действие по умолчанию для всех сигналов, захватываемых предком, а все прочие сигналы не вызывают действий

Свойство игнорирования сигналов порожденным процессом, заимствованное у родительского процесса, можно использовать для создания фоновых процессов.

## 19. Функции для работы с сигналами

В системах Linux концепция сигналов была расширена. Это вызвано следующими недостатками старого ANSI-C подхода:

* Не был возможен опрос актуального статуса сигналов;
* Возникают конфликты между 2 одинаковыми сигналами: один обработчик сигнала должен игнорировать его, а другой должен выполнять определенную функцию.

В новой концепции сигналов вначале вводится переменная примитивного типа данных sigset\_t signal\_set.

**Множество сигналов инициализируется функцией sigemptyset**().

int sigemptyset(sigset\_t \*sig\_m);

**Добавление новых сигналов для конкретного процесса**:

int sigaddset(sigset\_t \*sig\_m, int signr);

signr являются номерами сигналов, которые добавляются во множество. Также можно использовать символическое имя, например:

sigaddset(&signal\_set, SIGINT);

**Удаление signr из множества сигналов sig\_m**:

int sigdelset(setsig\_t \*sig\_m, int signr);

**Проверка на наличие сигнала в множестве**:

int sigismember(sigset\_t sig\_m,int signr);

Если сигнал присутствует в множестве, функция возвращает 1, иначе 0.

**Сохранение или изменение маски сигналов**:

int sigprocmask(int mode, const sigset\_t \*sig\_m, sigset\_t \*alt\_sig\_m);

Существует три режима использования этой функции:

* sigprocmask (mode, NULL, alt\_sig\_m);

При этом в текущем процессе множество сигналов записывается по адресу alt\_sig\_m. mode в этом случае не оказывает эффекта.

* sigprocmask (mode, sig\_m, NULL);

Маска сигналов изменяется на новую маску sig\_m.

* sigprocmask (mode, sig\_m, alt\_sig\_m);

Сначала актуальная маска сигналов записывается в alt\_sig\_m, то есть сохраняется. Затем устанавливается множество сигналов sig\_m.

Предопределённые константы:

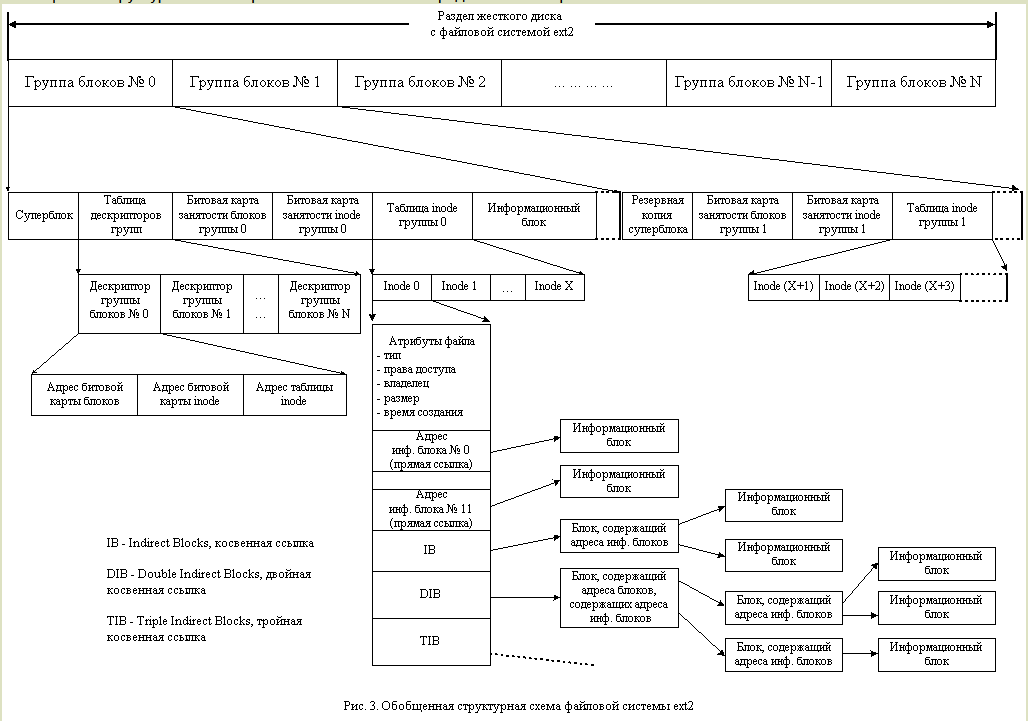
* SIG\_BLOCK: ко множеству сигналов добавляются все установленные в sig\_m сигналы.
* SIG\_UNBLOCK: Сигналы, установленные в sig\_m, удаляются.
* SIG\_SETMASK: устанавливается новая маска сигналов с сигналами, указанными в sig\_m.

**Изменение маски или запрет всех сигналов во время выполнения определенной части кода**:

int sigsuspend(const sigset\_t \*sig\_m);

С помощью этой функции можно блокировать процесс до тех пор, пока не придет нужный сигнал.

## 20. Файловая система Ехт2



Блок 0 не используется системой Linux и содержит код загрузки компьютера. Следом за блоком 0 дисковый раздел разделен на группы блоков.

Первый блок — это **суперблок** (superblock), в котором хранится информация о компоновке файловой системы, включая количество inode, количество дисковых блоков, *начало списка свободных дисковых блоков(?).*

Затем следует **дескриптор группы**, содержащий информацию о расположении битовых массивов, количестве свободных блоков и inode в группе, а также количестве каталогов в группе. Файловая система ext2 пытается распределить каталоги равномерно по всему диску.

В двух **битовых массивах** ведется учет свободных блоков и свободных inode. Размер каждого битового массива равен одному блоку.

Затем располагаются сами **inode**. Они нумеруются от 1 до некоторого максимума. Размер каждого inode — 128 байт, и описывает он ровно **один файл**. inode содержит учетную информацию, а также достаточное количество информации для определения местоположения всех дисковых блоков, которые содержат данные файла.

Следом за inode-ами идут **блоки данных**. Здесь хранятся все файлы и каталоги. Если файл или каталог состоит более чем из одного блока, то эти блоки не обязаны быть непрерывными на диске. В действительности блоки большого файла, скорее всего, будут разбросаны по всему диску.

Ext2 пытается расположить обычные файлы в той же самой группе блоков, что и родительский каталог, а файлы данных — в том же блоке, что и inode исходного файла (при условии, что там имеется достаточно места). Битовые массивы используются для того, чтобы принимать быстрые решения относительно выделения места для новых данных файловой системы. Когда выделяются новые блоки файлов, то ext2 также делает упреждающее выделение нескольких (восьми) дополнительных блоков для этого же файла (чтобы минимизировать фрагментацию файла из-за будущих операций записи).

Система поддерживает кэш каталогов, к которым недавно производился доступ. Поиск в кэше производится по имени файла, и если оно найдено, то дорогой линейный поиск уже не нужен.

Если файл имеется в наличии, то система извлекает номер inode, inode помещается в таблицу inode — структуру данных ядра, которая содержит все inode для открытых в данный момент файлов и каталогов. Формат элементов inode должен содержать все поля, которые возвращает системный вызов stat.

Таблица описания открытых файлов — объединяющая структура между таблицей дескрипторов файлов и таблицей inode, чтобы хранить в ней указатель в файле (а также бит чтения/записи), чтобы позволить родительскому и дочернему процессам совместно использовать один указатель в файле, но для посторонних процессов выделять персональные указатели.

## 21. Файловая система в ОС UNIX EXT3

Файловая система **Ext3** являет собой доработанную версию **Ext2**. Информация все также записываются в блоки данных одинаковой длины, а максимальный размер файла остался на прежнем уровне – **2ТБ**.

Однако добавилось **журналирование** – технология, которая позволяет значительно снизить риск потери информации.

Суть заключается в следующем: файловая система отмечает все планируемые изменения в структуре данных. В случае потери питания система мгновенно считает информацию с «журнала» и **файлы не будут утеряны**.

**Ext3** поддерживает три уровня журналирования:

1. **Journal** – файловая система записывает и метаданные, и пользовательские данные в журнал, тем самым практически гарантируя **успешное восстановление файловой системы** в случае потери питания. В то же время этот метод наиболее сильно снижает производительность файловой системы. При хранении журнала на отдельном жестком диске или разделе гарантирует целостность данных.
2. **Writeback** – записываются только метаданные в журнал, т.е. изменения в файловой системе. Синхронизация метаданных и данных в файловой системе отсутствует. Этот уровень журналирования является самым быстрым, но не может гарантировать целостность данных.
3. **Ordered** (**упорядоченное журналирование**) – процесс записи выполняется в определенном порядке: сначала записываются метаданные в журнал, далее пользовательские данные записываются в файловую систему, после этого, метаданные подключаются к файлам на диске. В случае отключения питания «исчезнут» только те файлы, которые записывались на диск в этот момент. Структура файловой системы не повреждается.

## 22. Файловая система в ОС UNIX EXT4

**Ext4** была представлена в 2008 году и является файловой системой Linux по умолчанию с 2010 года. Она использует базовые принципы **Ext3**, однако скорость работы была значительно увеличена (она даже выше чем в **Ext2**), увеличен максимальный размер файла до 16 ТБ, добавлена поддержка накопителей объемом до 1 Экзабайта (1 048 576 Терабайт) и многое другое.

**Особенности Ext4:**

* Совместимость с **Ext2** и **Ext3**
* Больший размер файлов и файловой системы
* Масштабируемость подкаталогов – возможность создаваать неограниченное кол-во подкаталогов
  + Все дело в том, что **Ext4** использует структуру данных HTree, которая является версией B-дерева, адаптированной специально для **Ext4**
* Экстенты: один адрес на первый блок последовательности блоков данных.
  + Дело в том, что в старых версиях отображение блоков данных файла реализовано старым способом — то есть, отображаются все блоки, относящиеся к конкретному файлу. Внедрение экстентов позволяет выводить большое количество последовательных блоков информации при помощи одного дескриптора. Такой подход увеличивает производительность файловой системы в несколько раз. Система сохраняет только адрес первого и последнего блока данных, которые соответствуют большому файлу
* Многоблочное распределение: записывает всю экстенту, а не блоки по очереди
* Отложенное распределение
  + выделяет блоки памяти непосредственно перед записью файлов на диск. Такой подход позволяет снизить нагрузку на кеш-память и -> увеличить производительность
* Быстрый fsck: функция, позволяющая управлять файловой системой
* Режим без журналирования: ускорение файловой системы
* Онлайн-дефрагментация: перемещение файлов для создания экстент
* Устойчивое прераспределение: заранее выделяется место на жестком диске
* Механизм «шлагбаумов» по умолчанию включен: перед записью в файл заносится запись в журнал для гарантирования целостности системы
* Наносекунлная временная отметка (в **Ext3** секундная)
* Внедрение контрольных сумм журналов. Этот метод постоянно проверяет блоки данных на наличие повреждений. Это сокращает время журналирования -> повышается уровень производительности и безопасности.
* Быстрая проверка файловой системы. Реализована она следующим образом: в Ext4 таблицы inodes и свободные группы блоков помечены. Благодаря этому во время проверки файловой системы проверяются только блоки с данными
* команда **e4defrag**, которая позволяет дефрагментировать как весь диск, так и отдельные файлы. Снижение уровня фрагментации не только делает систему быстрее, но и снижает нагрузку на процессор и т.д.

Преимущества и недостатки файловой системы **Ext4**

Преимущества:

1. Журналирование (в случае сбоев без проблем можно восстановить структуру файловой системы);
2. Поддержка шифрования;
3. Высокая стабильность — файловая система **Ext4** может сама контролировать свое состояние;
4. Поддержка по умолчанию во многих дистрибутивах (-> весь инструментарий для работы с **Ext4** будет из коробки);
5. Активная разработка;
6. Низкий уровень фрагментации (-> очень быстрая файловая система);
7. Большое количество лимитов — во время создания файловой системы создается некоторое количество inodes, которое необходимо для записи файлов. В некоторых файловых системах количество inodes может оказаться слишком малым, и пользователь не может записать новый файл даже при наличии свободного места на диске. К счастью, **Ext4** лишена этой проблемы и количества inodes хватает как для обычных пользователей, так и для серверных систем.

Недостатки:

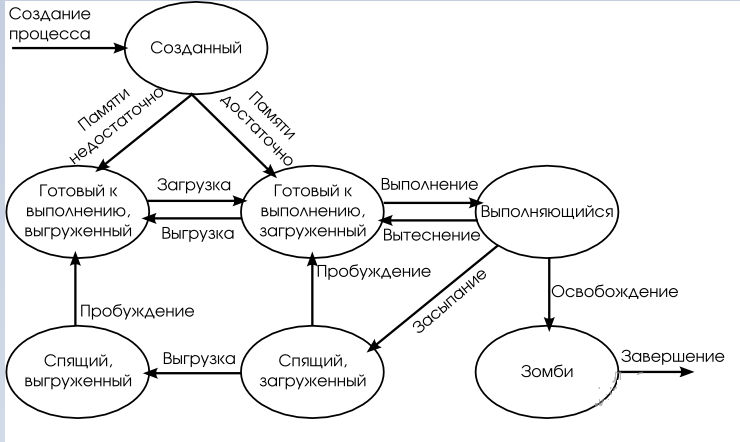
1. Отсутствие поддержки функций файловых систем следующего поколения (в **Ext4** отсутствует поддержка управления томами, дедупликация данных и т.д);
2. Отсутствие проверки контрольных сумм для данных (-> невозможно обнаружить повреждения данных из-за аппаратных сбоев оборудования);
3. Плохая масштабируемость — несмотря на то, что максимальный размер раздела составляет 1 Экзабайт, в реальности при создании разделов размером более 100 Терабайт очень сильно падает производительность;
4. Отсутствие поддержки прозрачного сжатия и прозрачного шифрования;
5. Inodes занимают до 10% объема раздела — если во время создания файловой системы будет выделено меньшее количество inodes – есть вероятность, что они закончатся и тогда пользователь не сможет записать новые файлы даже при наличии свободного места на диске.

## 23. Процессы в ОС UNIX, системные вызовы wait, exit

Одновременно может быть запущена более чем одна программа. Каждая программа, работающая в некоторый момент времени, называется **процессом**.

У каждого процесса есть уникальный номер, называемый PID (process ID) и, как и у файлов, у каждого процесса есть владелец и группа. Информация о владельце и группе процесса используется для определения того, какие файлы и устройства могут быть открыты процессом с учетом прав на файлы, о которых говорилось ранее. Также у большинства процессов есть родительский процесс. Например, при запуске команд из оболочки, оболочка является процессом, и любая запущенная команда также является процессом. Для каждого запущенного таким путем процесса оболочка будет являться родительским процессом. Исключением из этого правила является специальный процесс, называемый init (всегда первый процесс, его PID всегда равен 1, запускается автоматически ядром во время загрузки).

Жизненный цикл процесса



Для запуска программ в UNIX используются два системных вызова: **fork()** и **exec()**.

**fork()** — системный вызов, создающий новый (дочерний) процесс, идентичный выполняющему этот вызов. После вызова **fork()** алгоритм обычно разветвляется (родительский процесс получает от **fork()** значение PID дочернего процесса, а дочерний получает нуль).

После **fork()** дочерний процесс чаще всего выполняет системный вызов **exec()**, загружающий в пространство процесса новую программу. Так, первый (нулевой) процесс UNIX (ядро системы) создаёт свою копию, чтобы запустить init (процесс с PID = 1), который в свою очередь создаёт дочерние процессы для запуска инициализации системы и терминалов.

При завершении работы программа (дочерний процесс) выполняет системный вызов **exit()1**, при этом родительский процесс с помощью системного вызова **wait()2** (или **waitpid()3**) должен очистить таблицы планировщика процессов от информации о своем дочернем процессе. Если этого не происходит и родительский процесс завершается, не вызвав **wait()** для всех своих дочерних процессов, в системе возникают зомби (zombie), представляющие собой записи в таблицах планировщика процессов. Очистить операционную систему от зомби можно только с помощью перезагрузки.

Команды мониторинга процессов

Две команды очень полезны для просмотра работающих в системе процессов:

* ps (process status): используется для получения списка запущенных процессов и может показать их PID, сколько памяти они используют, команду, которой они были запущены и т.д.
* top (table of processes): показывает запущенные процессы и обновляет экран каждые несколько секунд, что позволяет наблюдать за работой компьютера в реальном времени.

1(**exit()**) — "немедленно" завершает работу программы. Все дескрипторы файлов, принадлежащие процессу, закрываются, все его дочерние процессы начинают управляться процессом 1 (init), а родительскому процессу посылается сигнал SIGCHLD.

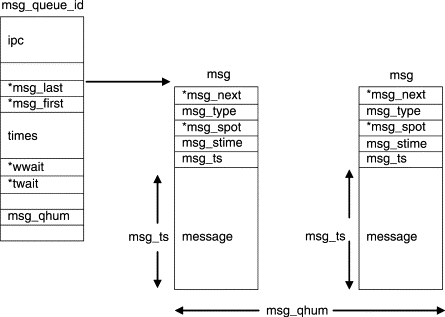
2(**wait()**) — блокирует вызывающий процесс до тех пор, пока один из его дочерних процессов не завершится (или не произойдет ошибка)

3(**waitpid()**) — приостанавливает выполнение вызвавшего процесса до тех пор, пока не изменится состояние потомка, заданного аргументом **pid**. По умолчанию **waitpid()** ожидает только прекращения работы потомка, но это можно изменить через аргумент options.

Значением **pid** может быть:

* < -1
  + означает, что нужно ждать любого потомка, чей идентификатор группы процессов равен абсолютному значению **pid**.
* -1
  + означает, что нужно ждать любого потомка.
* 0
* означает, что нужно ждать любого потомка, чей идентификатор группы процессов равен таковому у вызывающего процесса.
* > 0
  + означает, что нужно ждать любого потомка, чей идентификатор процесса равен **pid**.

## 24. Взаимодействие процессов в ОС UNIX, очереди сообщений



В UNIX процессы выполняются в собственном адресном пространстве и по существу изолированы друг от друга. Тем самым сведены к минимуму возможности влияния процессов друг на друга, что является необходимым в многозадачных операционных системах. Однако от одиночного изолированного процесса мало пользы. Сама концепция UNIX заключается в модульности, т.е. основана на взаимодействии между отдельными процессами.

Для реализации взаимодействия требуется:

* Обеспечить средства взаимодействия между процессами и ключ одновременно;
* Исключить нежелательное влияние одного процесса на другой.

Взаимодействие между процессами необходимо для решения следующих задач:

* Передача данных
  + Один процесс передает данные другому процессу, при этом их объем может варьироваться от десятков байтов до нескольких мегабайтов.
* Совместное использование данных
  + Вместо копирования информации от одного процесса к другому, процессы могут совместно использовать одну копию данных, причем изменения, сделанные одним процессом, будут сразу же заметны для другого. Количество взаимодействующих процессов может быть больше двух. При совместном использовании ресурсов процессам может понадобиться некоторый протокол взаимодействия для сохранения целостности данных и исключения конфликтов при доступе к ним.
* Извещения
  + Процесс может известить другой процесс или группу процессов о наступлении некоторого события. Это может понадобиться, например, для синхронизации выполнения нескольких процессов.

Очевидно, что решать данную задачу средствами самих процессов неэффективно, а в рамках многозадачной системы — опасно и потому невозможно. Таким образом, сама операционная система должна обеспечить механизмы межпроцессного взаимодействия (Inter-Process Communication, IPC).

К средствам межпроцессного взаимодействия, присутствующим во всех версиях UNIX, можно отнести:

* сигналы
* каналы
* FIFO (именованные каналы)
* сообщения (очереди сообщений)
* семафоры
* разделяемую память
* сокеты

Последние три типа IPC обычно обобщенно называют *System V IPC*.

Во многих версиях UNIX есть еще одно средство IPC — сокеты, впервые предложенные в BSD UNIX.

*Очереди сообщений*

Для обеспечения возможности обмена сообщениями между процессами этот механизм поддерживается следующими системными вызовами:

* **msgget** для образования новой очереди сообщений или получения дескриптора существующей очереди;
* **msgsnd** для посылки сообщения (вернее, для его постановки в указанную очередь сообщений);
* **msgrcv** для приема сообщения (вернее, для выборки сообщения из очереди сообщений);
* **msgctl** для выполнения ряда управляющих действий.

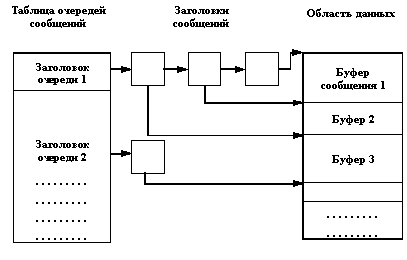
Системный вызов **msgget** обладает стандартным для семейства "get" системных вызовов синтаксисом:

*msgqid = msgget(key, flag);*

Ядро хранит сообщения в виде связного списка (очереди), а дескриптор очереди сообщений является индексом в массиве заголовков очередей сообщений. В дополнение к информации, общей для всех механизмов IPC в UNIX System V, в заголовке очереди хранятся также:

* указатели на первое и последнее сообщение в данной очереди;
* число сообщений и общее количество байтов данных во всех них вместе взятых;
* идентификаторы процессов, которые последними послали или приняли сообщение через данную очередь;
* временные метки последних выполненных операций **msgsnd**, **msgrsv** и **msgctl**.

Как обычно, при выполнении системного вызова **msgget** ядро ОС UNIX либо создает новую очередь сообщений, помещая ее заголовок в таблицу очередей сообщений и возвращая пользователю дескриптор вновь созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий дескриптор очереди.



*Рис. Структуры данных, используемые для организации очередей сообщений*

Для посылки сообщения используется системный вызов **msgsnd**:

*msgsnd(msgqid, msg, count, flag);*

* **msg** — это указатель на структуру, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив - собственно сообщение;
* **count** задает размер сообщения в байтах;
* **flag** определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти.

Для того чтобы ядро успешно поставило указанное сообщение в указанную очередь сообщений, должны быть выполнены следующие условия:

* обращающийся процесс должен иметь соответствующие права по записи в данную очередь сообщений;
* длина сообщения не должна превосходить установленный в системе верхний предел;
* общая длина сообщений (включая вновь посылаемое) не должна превосходить установленный предел;
* указанный в сообщении тип сообщения должен быть положительным целым числом.

Для приема сообщения используется системный вызов **msgrcv**:

*count = msgrcv(id, msg, maxcount, type, flag);*

* **msg** — это указатель на структуру данных в адресном пространстве пользователя, предназначенную для размещения принятого сообщения;
* **maxcount** задает размер области данных (массива байтов) в структуре **msg**;
* значение **type** специфицирует тип сообщения, которое желательно принять (если это значение является положительным целым числом, ядро выбирает из очереди сообщений первое сообщение с таким же типом, если же значение параметра **type** есть отрицательное целое число, то ядро выбирает из очереди первое сообщение, значение типа которого меньше или равно абсолютному значению параметра **type**);
* значение параметра **flag** указывает ядру, что следует предпринять, если в указанной очереди сообщений отсутствует сообщение с указанным типом.

Возвращаемое значение системного вызова задает реальное число байтов, переданных пользователю.

Выполнение системного вызова, как обычно, начинается с проверки правомочности доступа обращающегося процесса к указанной очереди. Далее, если значением параметра **type** является нуль, ядро выбирает первое сообщение из указанной очереди сообщений и копирует его в заданную пользовательскую структуру данных. После этого корректируется информация, содержащаяся в заголовке очереди (число сообщений, суммарный размер и т.д.). Если какие-либо процессы были отложены по причине переполнения очереди сообщений, то все они активизируются. В случае, если значение параметра **maxcount** оказывается меньше реального размера сообщения, ядро не удаляет сообщение из очереди и возвращает код ошибки. Однако, если задан флаг MSG\_NOERROR, то выборка сообщения производится, и в буфер пользователя переписываются первые **maxcount** байтов сообщения.

Во всех случаях, если в указанной очереди отсутствуют сообщения, соответствующие спецификации параметра **type**, ядро откладывает (усыпляет) обратившийся процесс до появления в очереди требуемого сообщения. Однако, если в параметре **flag** задано значение флага IPC\_NOWAIT, то процесс немедленно оповещается об отсутствии сообщения в очереди путем возврата кода ошибки.

Для опроса состояния описателя очереди сообщений, изменения его состояния (например, изменения прав доступа к очереди) и для уничтожения указанной очереди сообщений (детали мы опускаем) используется системный вызов **msgctl**:

*msgctl(id, cmd, mstatbuf);*

## 25. Взаимодействие процессов в ОС UNIX с применением семафоров

Семафор в ОС UNIX состоит из следующих элементов:

* значение семафора;
* идентификатор процесса, который хронологически последним работал с семафором;
* число процессов, ожидающих увеличения значения семафора;
* число процессов, ожидающих нулевого значения семафора.

Для работы с семафорами поддерживаются три системных вызова:

* **semget** для создания и получения доступа к набору семафоров;
* **semop** для манипулирования значениями семафоров (это именно тот системный вызов, который позволяет процессам синхронизироваться на основе использования семафоров);
* **semctl** для выполнения разнообразных управляющих операций над набором семафоров.

Системный вызов **semget** имеет следующий синтаксис:

*id = semget(key, count, flag);*

где прямые параметры **key** и **flag** и возвращаемое значение системного вызова имеют тот же смысл, что для других системных вызовов семейства "get", а параметр **count** задает число семафоров в наборе семафоров, обладающих одним и тем же ключом. После этого индивидуальный семафор идентифицируется дескриптором набора семафоров и номером семафора в этом наборе. Если к моменту выполнения системного вызова **semget** набор семафоров с указанным ключом уже существует, то обращающийся процесс получит соответствующий дескриптор, но так и не узнает о реальном числе семафоров в группе (хотя позже это все-таки можно узнать с помощью системного вызова **semctl**).

Основным системным вызовом для манипулирования семафором является **semop**:

*oldval = semop(id, oplist, count);*

* **id** — это ранее полученный дескриптор группы семафоров;
* **oplist** — массив описателей операций над семафорами группы;
* **count** — размер этого массива.

Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива **oplist** имеет следующую структуру:

* номер семафора в указанном наборе семафоров;
* операция;
* флаги.

Если проверка прав доступа проходит нормально, и указанные в массиве **oplist** номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов выполняется следующим образом. Для каждого элемента массива **oplist** значение соответствующего семафора изменяется в соответствии со значением поля "операция".

* Если значение поля операции > 0, то значение семафора увеличивается на единицу, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются).
* Если значение поля операции и значение семафора = 0, выбирается следующий элемент массива **oplist**. Если же значение семафора != 0, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания (усыпляется).
* Наконец, если значение поля операции < 0, и его абсолютное значение <= значению семафора, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует (пробуждает) все процессы, ожидающие нулевого значения этого семафора. Если же значение семафора < абсолютной величины поля операции, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и откладывает (усыпляет) текущий процесс до наступления этого события.

Основным поводом для введения массовых операций над семафорами было стремление дать программистам возможность избегать тупиковых ситуаций в связи с семафорной синхронизацией. Это обеспечивается тем, что системный вызов **semop**, каким бы длинным он не был (по причине потенциально неограниченной длины массива **oplist**) выполняется как атомарная операция, т.е. во время выполнения **semop** ни один другой процесс не может изменить значение какого-либо семафора.

Наконец, среди флагов-параметров системного вызова **semop** может содержаться флаг с символическим именем IPC\_NOWAIT, наличие которого заставляет ядро ОС UNIX не блокировать текущий процесс, а лишь сообщать в ответных параметрах о возникновении ситуации, приведшей бы к блокированию процесса при отсутствии флага IPC\_NOWAIT.

Системный вызов **semctl** имеет формат

*semctl(id, number, cmd, arg);*

* **id** — дескриптор группы семафоров;
* **number** — номер семафора в группе;
* **cmd** — код операции;
* **arg** — указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции.

В частности, с помощью **semctl** можно уничтожить индивидуальный семафор в указанной группе.

Преимущества семафоров:

1. Можно нарастить владельцем
2. Семафор – объект ядра, независимый от процессов и т.д.
3. Не позволяет входить в критические секции

Недостатки:

1. Инверсия приоритетов
2. Сложность в написании, что ведет к ошибкам при программировании
3. Могут возникать дедлоки

## 26. Работа с файлами в ОС UNIX. Системные вызовы

Все файлы, с которыми могут манипулировать пользователи, располагаются в файловой системе, представляющей собой дерево, промежуточные вершины которого соответствуют каталогам, и листья - файлам и пустым каталогам. Реально на каждом логическом диске (разделе физического дискового пакета) располагается отдельная иерархия каталогов и файлов. Для получения общего дерева в динамике используется "монтирование" отдельных иерархий к фиксированной корневой файловой системе.

Каждый каталог и файл файловой системы имеет уникальное полное имя (в ОС UNIX это имя принято называть **full pathname** — имя, задающее полный путь, поскольку оно действительно задает полный путь от корня файловой системы через цепочку каталогов к соответствующему каталогу или файлу). Каталог, являющийся корнем файловой системы (корневой каталог), в любой файловой системе имеет предопределенное имя "/". **Полное имя файла**, например, */bin/sh* означает, что в корневом каталоге должно содержаться имя каталога bin, а в каталоге bin должно содержаться имя файла sh. Коротким или **относительным именем файла** (relative pathname) называется имя (возможно, составное), задающее путь к файлу от текущего рабочего каталога.

В каждом каталоге содержатся два специальных имени, имя ".", именующее сам этот каталог, и имя "..", именующее "родительский" каталог данного каталога, т.е. каталог, непосредственно предшествующий данному в иерархии каталогов.

*Системные вызовы*

Основные системные функции, обеспечивающие обращение к уже существующим файлам, — это функции **open**, **read**, **write**, **create**, **lseek**, **close** и **unlink**.

Для того, чтобы указать ядру файл, к которому следует сделать обращение, необходимо этот файл либо открыть (**open**), либо создать (**сгеаte**).

**Открытие** — это процесс присвоения по имени файла некоторого значения некоторой переменной, хранящейся в области памяти процесса и называемой **дескриптором файла**. Если файл существует -> существует и дескриптор, и ссылка по имени успешно разрешена.

Синтаксис вызова функции **open**:

*fd = open(pathname, flags, modes);*

* **pathname** — имя файла;
* **flags** указывает режим открытия (например, для чтения или записи);
* **modes** содержит права доступа к файлу в случае, если файл создается.

Системная функция **open** возвращает целое число (\*), именуемое пользовательским дескриптором файла.

Другие операции над файлами, такие как чтение, запись, позиционирование головок чтения/записи, воспроизведение дескриптора файла, установка параметров ввода/вывода, определение статуса файла и закрытие файла, используют значение дескриптора файла, возвращаемое системной функцией **open**.

***Create***

Системный вызов **create** создает новый файл или перезаписывает уже существующий. Если файла не существовало, ему присваивается код защиты, указанный в параметре **mode.** Если файл существовал, его код защиты и имя владельца сохраняются, но длина обнуляется. Одновременно файл открывается на запись и пользователю возвращается номер дескриптора. Формат вызова **create** следующий:

*create (name, mode);*

***Read***

Функция **read** служит для обмена между памятью процесса и внешним объектом, когда его дескриптор уже получен.

Синтаксис вызова системной функции **read**:

*number = read(fd, buffer, count);*

* **fd** — дескриптор файла, возвращаемый функцией **open**;
* **buffer** — адрес структуры данных в пользовательском процессе, где будут размещаться считанные данные в случае, если успешно завершится выполнение функции read;
* **count** — количество байт, которые пользователю нужно прочитать;
* **number** — количество фактически прочитанных байт.

После того, как ядро установит значения параметров ввода/вывода в адресном пространстве процесса, оно обращается к индексу, используя указатель из таблицы файлов, и блокирует его прежде, чем начать чтение из файла.

Затем начинается цикл, выполняющийся до тех пор, пока операция чтения не будет произведена до конца. Ядро преобразовывает смещение в байтах внутри файла в номер блока и вычисляет смещение внутри блока до места, откуда следует начать ввод/вывод, а также количество байт, которые будут прочитаны из блока.

После считывания блока в буфер ядро копирует данные из блока по адресу, установленному в пользовательском процессе. Если запрос пользователя не удовлетворен, ядро повторяет весь цикл, преобразовывая смещение в байтах внутри файла в номер блока, считывая блок с диска в системный буфер, копируя данные из буфера в пользовательский процесс, освобождая буфер и корректируя значения параметров ввода/вывода в адресном пространстве процесса. Цикл завершается либо, когда ядро выполнит запрос пользователя полностью, либо, когда в файле кончатся данные, либо, когда ядро обнаружит ошибку при чтении данных с диска или при копировании данных в пространство пользователя. Ядро корректирует значение смещения в таблице файлов в соответствии с количеством фактически прочитанных байт.

***Write***

Синтаксис вызова системной функции **write**:

*number = write(fd, buffer, count);*

где переменные **fd, buffer, count** и **number** имеют тот же смысл, что и для вызова системной функции **read.** Алгоритм записи в обычный файл похож на алгоритм чтения из обычного файла, за исключением того, что дескриптор должен быть предварительно создан или файл открыт на запись. Практически система не ограничивает длину передаваемой информации, однако при записи на диски и ленты лучше всего работать блоками, поскольку эти операции производятся с большей эффективностью. При обработке символьных данных, когда возникает необходимость принимать и передавать данные из файла в файл по одному символу, можно использовать системные функции **getchar** и **putchar.**

***Lseek***

Системный вызов **Lseek** обеспечивает продвижение указателя к нужному месту в файле для осуществления операций ввода/вывода.

Синтаксис вызова системной функции:

*position = lseek(fd, offset, ptr);*

* **fd** — дескриптор файла, идентифицирующий файл;
* **offset** — смещение в байтах;
* **ptr** указывает, является ли значение **offset** смещением от начала файла — **0**, смещением от текущей позиции ввода/вывода — **1** или смещением от, конца файла — **2**.

Возвращаемое значение, **position**, является смещением в байтах до места, где будет начинаться следующая операция чтения или записи. Для того чтобы выполнить функцию **Lseek**, ядро просто выбирает значение смещения из таблицы файлов; в последующих вызовах функций **read** и **write** смещение из таблицы файлов используется в качестве начального смещения.

***Close***

Формат системного вызова **close** следующий:

*close (fd);*

где **fd** — дескриптор закрываемого файла.

Закрытие файлов, открытых процессом, происходит автоматически при завершении процесса, но, т.к. существует все-таки ограничение на количество открытых одновременно одним процессом файлов, то приходится прибегать к их закрытию. Например, при замене файлов стандартного ввода и вывода используется функция **close**. Код возврата при успешном завершении операции - 0, а при неуспешном - -1.

***Unlink***

Удаление файла производится при помощи системного вызова **unlink**.

Формат системного вызова **unlink** следующий:

*unlink (name);*

где **name** — указатель на строку символов, заканчивающуюся обязательным нулем, где содержится имя удаляемого файла. При этом ссылка на файл из текущего каталога удаляется и, если файл не упоминается в другом каталоге, содержимое его также удаляется.

## 27. Взаимодействие процессов в ОС UNIX, разделяемая память

Для работы с разделяемой памятью используются четыре системных вызова:

* **shmget** создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
* **shmat** подключает сегмент с указанным дескриптором к виртуальной памяти обращающегося процесса;
* **shmdt** отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
* **shmctl** служит для управления разнообразными параметрами, связанными с существующим сегментом.

После того, как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

Синтаксис системного вызова **shmget** выглядит следующим образом:

*shmid = shmget(key, size, flag);*

Параметр **size** определяет желаемый размер сегмента в байтах. Далее работа происходит по общим правилам. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является дескриптор существующего сегмента (и обратившийся процесс так и не узнает реального размера сегмента, хотя впоследствии его все-таки можно узнать с помощью системного вызова **shmctl**). В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Создание сегмента не означает немедленного выделения под него основной памяти. Это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Аналогично, при выполнении последнего системного вызова отключения сегмента от виртуальной памяти соответствующая основная память освобождается.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову **shmat**:

*virtaddr = shmat(id, addr, flags);*

Здесь **id** — это ранее полученный дескриптор сегмента, а **addr** — желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является реальный виртуальный адрес начала сегмента (его значение не обязательно совпадает со значением прямого параметра **addr**). Если значением **addr** является нуль, ядро выбирает наиболее удобный виртуальный адрес начала сегмента. Кроме того, ядро старается обеспечить (но не гарантирует) выбор такого стартового виртуального адреса сегмента, который обеспечивал бы отсутствие перекрывающихся виртуальных адресов данного разделяемого сегмента, сегмента данных и сегмента стека процесса (два последних сегмента могут расширяться).

Для отключения сегмента от виртуальной памяти используется системный вызов **shmdt**:

*shmdt(addr);*

где **addr** — это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова **shmat**. Естественно, система гарантирует (на основе использования таблицы сегментов процесса), что указанный виртуальный адрес действительно является адресом начала (разделяемого) сегмента в виртуальной памяти данного процесса.

Системный вызов **shmctl**:

*shmctl(id, cmd, shsstatbuf);*

содержит прямой параметр **cmd**, идентифицирующий требуемое конкретное действие, и предназначен для выполнения различных функций.

Уничтожение сегмента производится следующим образом. Если к моменту выполнения системного вызова ни один процесс не подключил сегмент к своей виртуальной памяти, то основная память, занимаемая сегментом, освобождается, а соответствующий элемент таблицы разделяемых сегментов объявляется свободным. В противном случае в элементе таблицы сегментов выставляется флаг, запрещающий выполнение системного вызова **shmget** по отношению к этому сегменту, но процессам, успевшим получить дескриптор сегмента, по-прежнему разрешается подключать сегмент к своей виртуальной памяти. При выполнении последнего системного вызова отключения сегмента от виртуальной памяти операция уничтожения сегмента завершается.

## 28. Понятие потока в ОС UNIX. Создание потока, завершение потока

Каждому процессу соответствует адресное пространство и одиночный поток исполняемых команд. В многопользовательских системах, при каждом обращении к одному и тому же сервису, приходится создавать новый процесс для обслуживания клиента. Это менее выгодно, чем создать квазипараллельный поток внутри этого процесса с одним адресным пространством.

С каждым потоком связывается:

* Счетчик выполнения команд
* Регистры для текущих переменных
* Стек
* Состояние

Потоки делят между собой элементы своего процесса:

* Адресное пространство
* Глобальные переменные
* Открытые файлы
* Таймеры
* Семафоры
* Статистическую информацию.

*Преимущества использования потоков*

1. Упрощение программы в некоторых случаях, за счет использования общего адресного пространства.
2. Быстрота создания потока, по сравнению с процессом, примерно в 100 раз.
3. Повышение производительности самой программы, т.к. есть возможность одновременно выполнять вычисления на процессоре и операцию ввода/вывода. Пример: текстовый редактор с тремя потоками может одновременно взаимодействовать с пользователем, форматировать текст и записывать на диск резервную копию.

|  |  |
| --- | --- |
| **Вызовы, связанные с потоком** | **Описание** |
| pthread\_create | Создание нового потока |
| pthread\_exit | Завершение работы вызвавшего потока |
| pthread\_join | Ожидание выхода из указанного потока |
| pthread\_yield | Освобождение ЦП, позволяющее выполняться другому потоку |
| pthread\_attr\_init | Создание и инициализация структуры атрибутов потока |
| pthread\_attr\_destroy | Удаление структуры атрибутов потока |
| Pthread\_detach | Превращает поток из присоядинямого в отсоединенный (т.е. его не ловит pthread\_join) |
| Pthread\_cancel | Вежливо просит поток завершиться |

Все потоки Pthreads имеют определенные свойства. У каждого потока есть свои идентификатор, набор регистров (включая счетчик команд) и набор атрибутов, которые сохраняются в определенной структуре. Атрибуты включают размер стека, параметры планирования и другие элементы, необходимые при использовании потока.

*Создание потока*

Функция **pthread\_create()**позволяет добавить новый поток управления к текущему процессу. Прототип функции:

*int pthread\_create(pthread\_t \*tid, const pthread\_attr\_t \*tattr, void\*(\*start\_routine)(void \*), void \*arg);*

Когда атрибуты объекта не определены, они равны NULL, и поток, создаваемый по умолчанию, имеет следующие признаки:

* Неограниченность;
* Неотделимость от процесса;
* Стек с размером по умолчанию;
* Приоритет родителя.

Существует возможность также создать объект атрибутов потока с помощью функции **pthread\_attr\_init(),** а затем использовать этот объект для создания самого потока.

Функция **pthread\_create()**вызывается с атрибутом **attr**, определяющим необходимое поведение; ***start\_routine***— это функция, с которой новый поток начинает свое выполнение. Когда ***start\_routine***завершается, поток завершается со статусом выхода, установленным в значение, возвращенное ***start\_routine****.*

Если вызов **pthread\_create()**успешно завершен, идентификатор созданного потока сохраняется по адресу *tid*.

Создание потока с использованием аргумента атрибутов NULL оказывает тот же эффект, что и использование атрибута по умолчанию: оба создают одинаковый поток. При инициализации *tattr* он обретает поведение по умолчанию; **pthread\_create()**возвращает 0 при успешном завершении. Любое другое значение указывает, что произошла ошибка.

*Ожидание завершения потока*

Функция **pthread\_join()** используется для ожидания завершения потока:

*int pthread\_join(thread\_t tid, void \*\*status);*

Функция **pthread\_join()**блокирует вызывающий поток, пока указанный поток не завершится. Указанный поток должен принадлежать текущему процессу и не должен быть отделен. Если **status !=** NULL, он указывает на переменную, которая принимает значение статуса выхода завершенного потока при успешном завершении **pthread\_join()**. Несколько потоков не могут ждать завершения одного и того же потока. Если они пытаются выполнить это, один поток завершается успешно, а все остальные — с ошибкой ESRCH. После завершения **pthread\_join()**, любое пространство стека, связанное с потоком, может быть использовано приложением.

## 29. Применение мьютексов при взаимодействии потоков в ОС UNIX

**Мьютексы** справляются лишь с управлением взаимным исключением доступа к общим ресурсам или фрагментам кода. Простота и эффективность реализации мьютексов делает их особенно полезными для совокупности потоков, целиком реализованных в пользовательском пространстве.

**Мьютекс** — это совместно используемая переменная, которая может находиться в одном из двух состояний: **заблокированном** или **незаблокированном**. Следовательно, для их представления нужен только один бит, но на практике зачастую используется целое число, при этом нуль означает незаблокированное, а все остальные значения — заблокированное состояние. Для работы с мьютексами используются две процедуры. Когда потоку (или процессу) необходим доступ к критической области (критическая область кода обычно содержит операции считывания или изменения данных, используемых потоками совместно), он вызывает процедуру ***mutex\_lock***. Если мьютекс находится в незаблокированном состоянии (означающем доступность входа в критическую область), вызов проходит удачно и вызывающий поток может свободно войти в критическую область.

В то же время, если мьютекс уже заблокирован, вызывающий поток блокируется до тех пор, пока поток, находящийся в критической области, не завершит свою работу и не вызовет процедуру ***mutex\_unlock***. Если на мьютексе заблокировано несколько потоков, то произвольно выбирается один из них, которому разрешается воспользоваться заблокированностью других потоков.

При работе с потоками (в пользовательском пространстве) складывается несколько иная ситуация, связанная с отсутствием таймера, останавливающего работу слишком долго выполняющегося процесса. Следовательно, поток, пытающийся воспользоваться блокировкой, находясь в состоянии активного ожидания, войдет в бесконечный цикл и никогда не завладеет блокировкой, поскольку он никогда не позволит никакому другому потоку возобновить выполнение и снять блокировку.

Пока речь идет о совокупности потоков, реализованных в пользовательском пространстве, проблем с совместным доступом нескольких потоков к одному и тому же мьютексу не возникает, поскольку потоки выполняются в общем адресном пространстве. Но в большинстве предыдущих решений, например, в алгоритме Петерсона (если интересно, в Таненбауме есть) и семафорах, было невысказанное предположение, что несколько процессов имеют доступ по крайней мере к какому-то объему общей памяти, возможно, всего лишь к одному слову памяти, но все же имеют. Если у процессов разобщенные адресные пространства, о чем мы неизменно упоминали, то как они будут совместно использовать переменную turn в алгоритме Петерсона, или семафоры, или общий буфер?

На этот вопрос есть два ответа. Во-первых, некоторые общие структуры данных, например, семафоры, могут храниться в ядре и быть доступны только через системные вызовы. Такой подход позволяет устранить проблему. Во-вторых, большинство современных операционных систем (включая UNIX и Windows) предлагают процессам способ, позволяющий использовать некоторую часть их адресного пространства совместно с другими процессами. В этом случае допускается совместное использование буферов и других структур данных. В худшем случае, когда нет доступа ни к чему другому, можно воспользоваться общим файлом.

Если два или более процесса совместно используют все свои адресные пространства или их большие части, различие между процессами и потоками немного размывается, но все равно присутствует. Два процесса, использующие общее адресное пространство, все равно имеют различные открытые файлы, аварийные таймеры и другие присущие процессам отличительные свойства, а вот для потоков в рамках одного процесса эти свойства являются общими. И никуда не деться от того обстоятельства, что несколько процессов, использующих общее адресное пространство, никогда не будут столь же эффективными, как потоки, реализованные на пользовательском уровне, поскольку к управлению процессами неизменно привлекается ядро.

//Взято из Таненбаума по Современным ОС (стр. 163)

Преимущества мьютексов:

1. Просто блокировка ресурса
2. В одной секции критического кода только 1 процесс

Недостатки:

1. Если спит процесс, захвативший ресурс, остальные все равно ждут
2. Мьютекс не может быть освобожден из другого процесса
3. Нормальная реализация приводит к потере производительности процессора

## 30. Применение блокировок чтения-записи при взаимодействии потоков в ОС UNIX

Проблема чтения-записи относится к ситуации, в которой ресурс допускает два типа доступа (чтение и запись). Один тип доступа должен быть предоставлен эксклюзивно (например, запись), но другой тип может быть общим (например, чтение). К примеру, любое количество процессов может без труда читать из одного и того же файла, но только один процесс должен изменять файл в это время.

Две распространенные стратегии обработки синхронизации чтения-записи называются **строгой синхронизацией чтения** **и строгой синхронизацией записи**. Строгая синхронизация чтения всегда отдает предпочтение читателям, предоставляя доступ читателям до тех пор, пока писатель в данный момент не пишет. Строгая синхронизация записи всегда отдает предпочтение писателям, задерживая чтение до завершения всех ожидающих или активных модулей записи. Система бронирования авиабилетов будет использовать сильное предпочтение писателей, поскольку читателям нужна самая последняя информация. С другой стороны, библиотечная справочная база данных может отдавать предпочтение читателям.

Стандарт POSIX предоставляет блокировки чтения-записи, которые позволяют нескольким читателям применять блокировку, при условии, что писатель не поддерживает блокировку. POSIX утверждает, что от реализации зависит, разрешать ли читателю применять блокировку, если писатели не имеют возможности блокировки.

Применительно к блокировкам чтение-запись предоставляются следующие группы функций:

**инициализация и разрушение блокировок**:

pthread\_rwlock\_init(), pthread\_rwlock\_destroy()

**установка блокировки на чтение**:

pthread\_rwlock\_rdlock(), pthread\_rwlock\_tryrdlock(), pthread\_rwlock\_timedrdlock()

**установка блокировки на запись**:

pthread\_rwlock\_wrlock(), pthread\_rwlock\_trywrlock(), pthread\_rwlock\_timedwrlock()

**снятие блокировки чтение-запись:**

pthread\_rwlock\_unlock()

**инициализация и разрушение атрибутных объектов блокировок:**

pthread\_rwlockattr\_init(), pthread\_rwlockattr\_destroy()

**получение и установка атрибутов блокировок в атрибутных объектах**:

pthread\_rwlockattr\_getpshared(), pthread\_rwlockattr\_setpshared()

Как правило, чтобы воспрепятствовать зависанию писателей, читателей в подобной ситуации «тормозят», иначе они так и будут подхватывать блокировку друг у друга. Для политик планирования реального времени преимущество получает более приоритетный поток.

Правила:

* Параллельно одним или несколькими потоками управления может быть установлено несколько блокировок на чтение. Сколько раз устанавливали такую блокировку, столько же раз ее необходимо снять
* Установка блокировки писателем возможна только после последнего снятия
* Максимально допустимое число одновременно установленных блокировок на чтение зависит от реализации
* Когда снимают блокировку, на которую претендуют и читатели, и писатели, как правило, преимущество получают писатели, но, строго говоря, решение зависит от реализации
* Снимать блокировку может только тот поток управления, который ее устанавливал; в противном случае поведение не определено

## 31. Взаимодействие процессов. Задача взаимного исключения

IPC:

* Сигналы
* полудуплексные каналы UNIX
* FIFO (именованные каналы)
* Очереди сообщений в стиле SYSV
* Множества (наборы) семафоров в стиле SYSV
* Разделяемые сегменты памяти в стиле SYSV
* Сетевые сокеты (в стиле Berkeley)
* Полнодуплексные каналы

Традиционные и широко используемые механизмы синхронизации, которые относятся к группе методов, *предотвращающих* конфликты, предполагают взаимное исключение по доступу к общим данным для одновременно выполняемых потоков, а различные реализации моделей параллельного программирования предоставляют необходимые средства для решения данной задачи.

Задача взаимного исключения в классическом виде впервые была сформулирована и решена (приведено решение математика Деккера) в работе голландского математика Дейкстры. Формальная постановка задачи выглядит следующим образом.

Предположим, что в ВС более двух процессов, выполняющих некоторую последовательность программных инструкций в бесконечном цикле. Выполняемые инструкции разделяются на четыре непрерывных блока кода: код, не требующий синхронизации, критическая секция (КС), вход в КС, выход из КС. Каждый процесс начинает свое выполнение с блока кода, не требующего синхронизации. В некоторый момент времени процессу может потребоваться выполнить инструкции из блока критической секции. Перед этим ему необходимо войти в КС, выполнив инструкции блока входа в КС, тем самым обеспечив себе эксклюзивное право выполнения инструкций КС. По завершении КС процесс выполняет инструкции выхода из нее, тем самым оповещая другие процессы о возможности получения доступа к КС. После выхода из КС процесс продолжает выполнять код, не требующий синхронизации.

Задача взаимного исключения состоит в разработке кода для блоков входа в КС и выхода из КС таким образом, чтобы были удовлетворены два основных требования:

*1. взаимное исключение* (англ. mutual exclusion) — свойство безопасности, которое гарантирует, что никакие два процесса не могут одновременно находиться внутри КС;

2. *отсутствие мертвой блокировки* или *свобода от взаимоблокировок* (англ. deadlock freedom) — свойство живости, подразумевающее, что если один процесс пытается войти в КС, то некоторый процесс, не обязательно тот же самый, рано или поздно войдет в свою КС.

Условие отсутствия мертвой блокировки предоставляет *гарантию прогресса* (живость) всей системы в целом, однако оно допускает ситуацию “голодания” отдельных процессов, которые периодически и безуспешно пытаются войти в КС. Более строгое условие гарантии прогресса — условие отсутствия “голодания” процессов (англ. starvation freedom) — если один процесс пытается войти в КС, то этот процесс рано или поздно обязательно войдет в свою КС. Для перечисленных условий решение задачи взаимного исключения может гарантировать определенную справедливость порядка выполнения КС процессами. Примером справедливого условия может служить условие “первый зашел-первым обслужен” или FIFO-справедливость (англ. first-in-first-out) — ни один процесс не может войти в КС до процесса, уже ожидающего своей очереди на вход в КС.

## 32. Проблема тупиков. Алгоритм банкира

**Тупики**

В компьютерных системах множество ресурсов, которые одновременно могут использоваться только одним процессом. Если два процесса будут использовать один и тот же элемент таблицы файловой системы, то эта система непременно будет повреждена. Поэтому все операционные системы способны временно предоставлять процессу исключительные права доступа к конкретным ресурсам.

Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае, когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется ***тупиком*** (***deadlock***). Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. Системная тупиковая ситуация, или «зависание системы», является следствием того, что один или более процессов находятся в состоянии тупика. Иногда подобные ситуации называют ***взаимоблокировками***.

Взаимоблокировка в группе процессов возникает в том случае, если каждый процесс из этой группы ожидает события, наступление которого зависит исключительно от другого процесса из этой же группы.

В большинстве случаев событием, наступления которого ожидает каждый процесс, является высвобождение какого-либо ресурса, которым на данный момент владеет другой участник группы. Этот вид взаимоблокировки называется **ресурсной взаимоблокировкой**.

Коффман (Coffman et al., 1971) показал, что для возникновения ресурсных взаимоблокировок должны выполняться четыре условия:

1. **Условие взаимного исключения**. Каждый ресурс либо выделен в данный момент только одному процессу, либо доступен.

2. **Условие удержания и ожидания**. Процессы, удерживающие в данный момент ранее выделенные им ресурсы, могут запрашивать новые ресурсы.

3. **Условие невыгружаемости**. Ранее выделенные ресурсы не могут быть принудительно отобраны у процесса. Они должны быть явным образом высвобождены тем процессом, который их удерживает.

4. **Условие циклического ожидания**. Должна существовать кольцевая последовательность из двух и более процессов, каждый из которых ожидает высвобождения ресурса, удерживаемого следующим членом последовательности.

***Соответственно, чтобы избежать взаимоблокировок необходимо гарантировать невыполнение хотя бы одного из этих условий****.*

Основная часть взаимоблокировок связана с ресурсами, к которым некоторым процессам были предоставлены исключительные права доступа. Чтобы придать рассмотрению взаимоблокировок как можно более универсальный характер, мы будем называть объекты, к которым предоставляется доступ, **ресурсами**. Ресурсы - аппаратные устройства (например, привод Blu-ray-дисков) или какая-то часть информации (например, запись базы данных).

Ресурсы бывают двух видов: **выгружаемые** и **невыгружаемые**.

К выгружаемым относятся такие ресурсы, которые могут быть безболезненно отобраны у процесса, который ими обладает. Невыгружаемый ресурс нельзя отобрать у его текущего владельца, не вызвав потенциально сбоя в вычислениях (если у процесса, который уже приступил к записи на Blu-ray-диск, внезапно отобрать пишущий привод и отдать его другому процессу, это приведет к порче Blu-ray-диска).

**Алгоритм Банкира (для одного типа ресурса)**

Модель алгоритма основана на примере банкира маленького городка, имеющего дело с группой клиентов, которым он выдал ряд кредитов. Алгоритм проверяет, ведет ли выполнение каждого запроса к небезопасному состоянию. Если да, то запрос отклоняется. Если удовлетворение запроса к ресурсу приводит к безопасному состоянию, ресурс предоставляется процессу.

Алгоритм банкира рассматривает каждый запрос по мере поступления и проверяет, приведет ли его удовлетворение к безопасному состоянию. Если да, то запрос удовлетворяется, в противном случае запрос откладывается до лучших времен.



*ПРИМЕР С БАНКОМ*

На рис. 6.9, а показаны четыре клиента: A, B, C и D, каждый из которых получил определенное количество единиц кредита. Банкир знает, что не всем клиентам тотчас же понадобится максимальная сумма их кредита, поэтому для обслуживания их потребностей он зарезервировал только 10 единиц, а не все 22, которые нужны клиентам.

Клиенты занимаются своими делами, время от времени запрашивая ресурсы. В какой-то определенный момент возникает ситуация, показанная на рис. 6.9, б. Это состояние не представляет опасности, поскольку при оставшихся двух единицах банкир может отложить выполнение любых запросов, за исключением запроса клиента C, позволяя C завершить свои дела и высвободить все четыре своих ресурса.

Рассмотрим, что получится, если запрос от B одной дополнительной единицы будет удовлетворен в ситуации, показанной на рис. 6.9, б. Мы получим небезопасную ситуацию, показанную на рис. 6.9, в. Если все клиенты внезапно запросят максимальные ссуды, банкир не сможет удовлетворить никого из них, и мы получим взаимоблокировку. Небезопасное состояние необязательно приводит к взаимоблокировке, поскольку клиенту может и не понадобиться максимальная сумма кредита, но банкир не может рассчитывать на это.

**Алгоритм Банкира (для нескольких типов ресурсов)**

Алгоритм банкира может быть распространен на работу с несколькими ресурсами. На рис. 6.10 показано, как он работает. Здесь изображены две матрицы. Левая матрица показывает, сколько экземпляров каждого ресурса в данный момент выделено каждому из пяти процессов. Правая показывает, сколько экземпляров ресурсов все еще необходимо каждому процессу для завершения его работы. На рис. 6.5 эти матрицы назывались C и R.

Судя по значению вектора E, в системе имеется шесть накопителей на магнитной ленте, три плоттера, четыре принтера и два привода Blu-ray-дисков. Из них заняты в данный момент пять накопителей, три плоттера, два принтера и два привода Blu-ray-дисков. Этот факт можно установить путем сложения значений четырех столбцов, соответствующих ресурсам, в левой матрице. Вектор доступных ресурсов — это разница между количеством присутствующих в системе ресурсов и количеством ресурсов, используемых в настоящее время.

**Алгоритм проверки состояния на безопасность.**

1. Ищем в матрице R строку, соответствующую процессу, чьи неудовлетворенные потребности в ресурсах меньше или равны вектору A. Если такой строки не существует, то система в конце концов войдет в состояние взаимоблокировки, поскольку ни один процесс не сможет доработать до успешного завершения (предполагается, что процессы удерживают все ресурсы, пока не завершат свою работу).

2. Допускаем, что процесс, чья строка была выбрана, запрашивает все необходимые ему ресурсы (возможность чего гарантируется) и завершает свою работу. Отмечаем этот процесс как завершенный и прибавляем все его ресурсы к вектору A.

3. Повторяем шаги 1 и 2 до тех пор, пока либо все процессы будут помечены как завершенные (в этом случае исходное состояние было безопасным), либо не останется процессов, чьи запросы могут быть удовлетворены (в этом случае система не была в безопасном состоянии).

## 33. Концепция виртуализации

Основная идея заключается в том, что монитор виртуальных машин (Virtual Machine Monitor (VMM)) создает иллюзию присутствия нескольких (виртуальных) машин на одном и том же физическом оборудовании. VMM известен также как **гипервизор**.

Виртуальные машины работают так же, как и реальные. В частности, у них должна быть возможность начальной загрузки, как на реальных машинах, и установки на них произвольных операционных систем, точно так же, как это может быть сделано на реальном оборудовании. Предоставление этой иллюзии с обеспечением достаточной эффективности является задачей гипервизора. Гипервизоры должны хорошо проявлять себя по трем направлениям:

1. **Безопасность** — у гипервизора должно быть полное управление виртуализированными ресурсами.
2. **Эквивалентность** — поведение программы на виртуальной машине должно быть идентичным поведению этой же программы, запущенной на реальном оборудовании.
3. **Эффективность** — основная часть кода в виртуальной машине должна выполняться без вмешательства гипервизора.

Безопасный способ выполнения инструкций заключается в поочередном рассмотрении каждой инструкции в интерпретаторе и в выполнении именно того, что нужно для данной инструкции. Некоторые инструкции могут быть выполнены напрямую, но их немного. Например, интерпретатор может быть способен выполнить инструкцию INC (инкремент) просто как есть, но инструкции, которые небезопасно выполнять напрямую, должны быть эмулированы интерпретатором. Например, нельзя разрешать гостевой операционной системе блокировать прерывания для всей машины или модифицировать отображения страниц в таблицах. Нужно применить прием, заставляющий операционную систему, посаженную поверх **гипервизора**, полагать, что она заблокировала прерывания или изменила отображение страниц на машине. Интерпретатор может быть безопасным и при тщательной реализации, возможно, даже высококачественным, но производительность его может оказаться не на высоте. Для того, чтобы соответствовать также критериям производительности, мониторы виртуальных машин (VMM) стараются выполнить основную часть кода непосредственным образом.

## 34. Подсистема управления памятью. Требования, предъявляемые к ней

Память является для процесса таким же важным ресурсом, как и процессор. Процесс может стать активным только в том случае, если хотя бы часть его кодов и данных находится в ОП. Управление памятью включает

* распределение имеющейся физической памяти между всеми существующими в системе процессами;
* загрузку кодов и данных процессов в отведенные им области памяти;
* настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области;
* защиту областей памяти каждого процесса.
* Кэширование памяти

Основные функции ОС по управлению памятью:

* Отслеживание свободной и занятой памяти
* Выделение памяти процессу и освобождение по его завершении
* Вытеснение процессов из ОП во внешнюю память, если основной памяти недостаточно для размещения их обратно в ОП
* Настройка адресов программы на конкретную область физической памяти

Один из наиболее распространенных способов управления памятью является механизм виртуальной памяти.

## 35. Виртуальная память. Задачи управления виртуальной памятью

**Виртуальная память** - метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путем автоматического перемещения частей программы между основной памятью и вторичным хранилищем (жёстким диском).

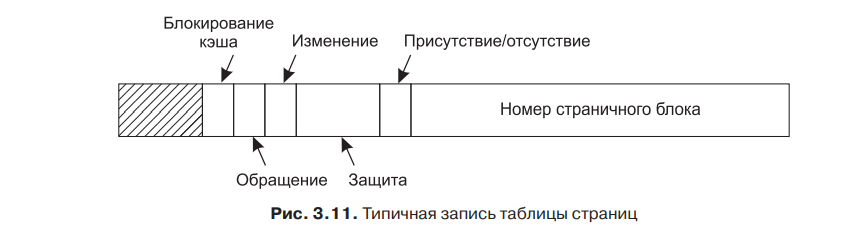
В основе виртуальной памяти лежит идея, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые **страницами**. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы одновременное присутствие в памяти всех страниц необязательно. Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету. Когда программа ссылается на часть своего адресного пространства, которое не находится в физической памяти, операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду.

При управлении виртуальной памятью решаются задачи:

1. **размещение** - задача, которая заключается в выборе в адресном пространстве оперативной памяти сегментов или страниц, на которые будут отображаться сегменты или страницы виртуального адресного пространства. При решении этой задачи стараются выбрать максимально простой алгоритм распределения памяти. Алгоритм строится таким образом, чтобы уменьшить частоту его использования.
2. **перемещение** - задача, которая заключается в выборе информации, которая хранится во внешней виртуальной памяти, для переноса в оперативную память.

Используется 2 основных варианта выборки информации во внешней памяти:

1. по требованию (страница передается в ОП только при обращении к ячейке памяти, расположенной на этой странице);
2. предварительно (предварительная загрузка нескольких “наиболее вероятных” страниц в ОП).
3. **преобразование** - задача, которая заключается в нахождении абсолютного физического адреса основной памяти по его виртуальному адресу.
4. **замещение** - задача, которая заключается в выборе страницы из оперативной памяти, которую следует переместить во внешнюю память.



## 36. Задача замещения при управлении виртуальной памятью, часовой алгоритм

Целью этой задачи является выбор среди пространства ОП той страницы, которую следует переместить во внешнюю память. Задача решается, когда обнаруживается отсутствие свободной страницы в ОП, а вместе с тем следует запрос на размещение новых страниц. Первый, наиболее простой подход, заключается в остановке процесса, потребовавшего страницу ОП в ситуации, когда вся память распределена. В этом случае управление передается другому процессу, у которого нет требований на дополнительную память, а в отношении приостановленного процесса либо не принимается никаких действий, либо все его страницы выгружаются во внешнюю память. Недостаток этого подхода- дискриминации подвергается тот процесс, в ходе выполнения которого возникла потребность замещения.

Идеальная стратегия замещения: **должна быть замещена та страница, к которой дольше всего не будет обращений в будущем.**

Существуют следующие стратегии:

- для замещения выбирается страница случайным образом;

- выбирается страница, которая дольше всего была в ОП;

- FIFO (удаляется та страница, которая раньше всех была распределена какому-либо процессу). Фактически, это реализация предыдущей стратегии, но она может быть реализована для различных процессов;

- алгоритм удаления дольше всех неиспользовавшейся страницы. Если долго обращения к странице не было, следовательно, в будущем тоже не предвидится.

В простейшем случае, с каждой страницей для этой стратегии связывается бит использования. Этот бит установлен в 1 при обращении к странице, а способ сброса бита в 0 и определяет способы реализации данной стратегии.

В настоящее время для решения задачи замещения находят применение варианты «часового» алгоритма.



Имеется циклический буфер размерности n, в каждом элементе хранится номер страницы и бит использования. Бит использования устанавливается в 1, когда к странице произведено обращение, этот бит также устанавливается при первой загрузке страницы в ОП. Указатель буфера указывает на последнюю замещенную страницу. Когда возникает потребность решить задачу замещения, указатель перемещается на следующий элемент буфера. Если бит использования установлен в 0, то производится замещение соответствующей страницы. Если же окажется, что бит использования равен 1, то страница не замещается, бит использования устанавливается в 0, а указатель перемещается на следующий элемент буфера. Перемещение указателя осуществляется до тех пор, пока не будет обнаружена страница с 0 (нулевым) битом использования. Повысить эффективность часового алгоритма можно путем увеличения количества используемых при его работе битов. Практически во всех системах страничной организации со страницей связывается бит модификации. Этот бит указывает, что страница не может быть замещена до тех пор, пока её содержимое не будет переписано во внешнюю память. Соответственно может быть 4 комбинации битов использования и битов модификации:

*n m*

*0 0* *n=0-давно использован*

*1 0* *1-недавно использован*

*0 1* *m=0- не модифицирован*

*1 1* *1- модифицирован*

Часовой алгоритм выглядит следующим образом:

1. Сканируем буфер, начиная с текущего положения. В процессе сканирования бит использования не изменяется. Первая страница с состоянием битов (0,0) замещается.
2. Если такой страницы нет, то ищем страницу с параметрами (0,1). Если такая страница найдена, она замещается. В процессе выполнения данного шага у всех просмотренных страниц сбрасывается бит использования.
3. Если выполнение шага 2 не дало результата, значит, у всех страниц будет сброшен бит использования, указатель буфера вернется в исходное положение, затем повторяем шаг 1 и, при необходимости, 2.

Явление **пробуксовки** наблюдается тогда, когда ОП имеет небольшие размеры, а программы велики по размеру. В этом случае может возникнуть ситуация частого замещения страниц, и большая часть процессорного времени тратится на выполнение служебных функций. Следовательно, резко замедляется выполнение пользовательских программ.

## 37. Распределение памяти. Система двойников

Основной функцией ОС по управлению памятью является размещение программы в основной памяти для её выполнения процессором. В современных ОС решение этой задачи предполагает использование *виртуальной памяти*.

Известно несколько *способов распределения памяти*:

1. Фиксированное распределение
2. Динамическое распределение
3. Простая страничная организация
4. Простая сегментация
5. Страничная организация виртуальной памяти
6. Сегментация виртуальной памяти

**Фиксированное распределение.** Оперативная память (ОП) разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера.

Плюсы:

1. Простота реализации
2. Малые системные затраты

Минусы:

1. неэффективное использование памяти из-за внутренней фрагментации и фиксированного максимального количества процессов.

**Динамическое распределение** - Разделы создаются динамически, каждый процесс загружается в раздел необходимого размера.

Плюсы:

1. Отсутствие фрагментации
2. Более эффективное использование ОП

Минусы:

1. Затраты процессора на противодействие внешней фрагментации

**Простая страничная организация.** ОП разделена на ряд кадров равного размера. Каждый процесс распределен на некоторое количество страниц равного размера, такой же длины, что и кадры памяти. Процесс загружается путем загрузки всех его страниц.

Плюсы - отсутствие внешней фрагментации.

Минусы - небольшая внутренняя фрагментация.

**Простая сегментация.** Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в динамические, не обязательно смежные, разделы.

Плюсы - отсутствие внутренней фрагментации.

Минусы – проблемы с внешней фрагментацией.

**Страничная организация виртуальной памяти**. Подобна простой страничной организации, но не требуется загружать все страницы процесса. Необходимые нерезидентные страницы автоматически подгружаются в память.

Плюсы:

1. Отсутствие внешней фрагментации
2. Более высокая степень многозадачности
3. Большое виртуальной адресное пространство

Минусы:

1. Значительные затраты на управление виртуальной памятью.

**Сегментация виртуальной памяти.** Подобна простой сегментации, но не требуется загружать все сегменты процесса. Необходимые нерезидентные сегменты автоматически подгружаются в память.

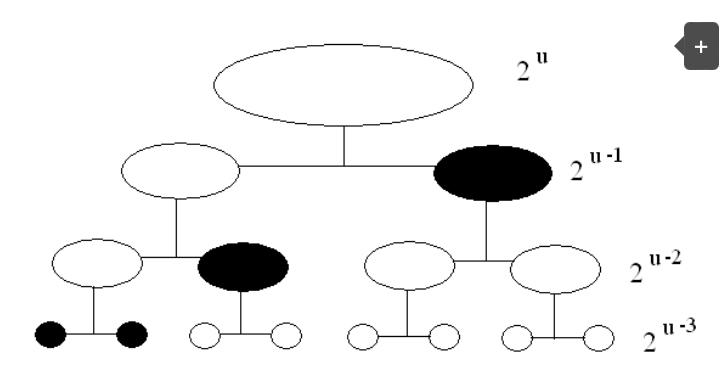
Плюсы:

1. Отсутствие внутренней фрагментации.
2. Более высокая степень многозадачности.
3. Больше виртуальной адресное пространство.
4. Подножка защиты и совместного использования.

Минусы - затраты на управление сложной виртуальной памятью.

**Система двойников:**

Фиксированное распределение памяти ограничивает количество активных процессов и неэффективно использует память при несоответствии между размерами разделов и процессами. Динамическое распределение реализуется более сложно и включает затраты на уплотнение памяти. Решением в этом плане является *система двойников*. В ней память распределяется блоками размером , l < k < u , где -минимальный размер выделяемого блока, а –максимальный размер (вся доступная распределенная память). Вначале все доступное для распределения адресное пространство рассматривается как единый блок размером . Если запрашивается блок размером s, таким что , то выделяется весь блок памяти, в противном случае, блок будет разделен на два одинаковых подблока (двойника), размерами . Если , то выделяется блок, иначе повторяется. Процесс деления продолжается до тех пор, пока не будет сгенерирован наименьший блок, размер которого не меньше . Система двойников всегда ведет список доступных блоков для каждого размера , где . Блок может быть удален из списка путем разделения его пополам и внесения двух свободных блоков в список . Когда пара свободных блоков в списке оказывается освобожденной, они удаляются из этого списка и объединяются в единый блок в списке .



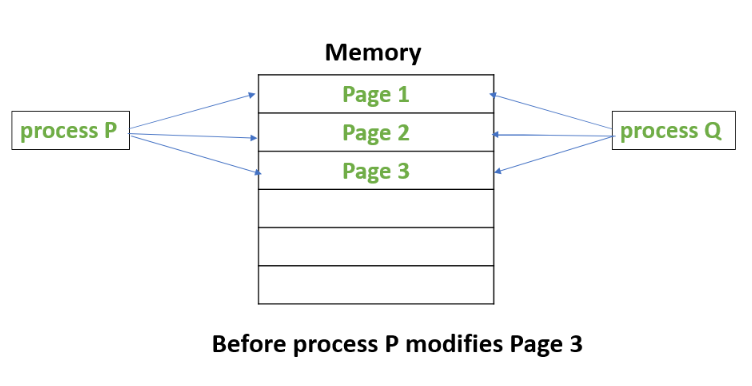
## 38. Copy on write

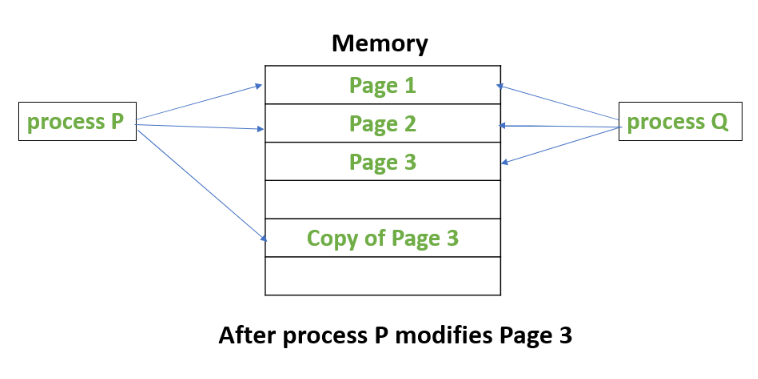
Механизм копирования при записи (Copy-On-Write) используется для оптимизации многих процессов, происходящих в операционной системе, таких как, например, работа с оперативной памятью или файлами на диске.

Идея подхода copy-on-write заключается в том, что при чтении области данных используется общая копия, в случае изменения данных — создается новая копия.

Например, при работе UNIX-функции fork() вместо реального копирования, под которое надо выделять память, ядро меняет дескрипторы страниц памяти родительского процесса, запрещая какую-либо запись в страницы данных. Затем создается дочерний процесс, которому копируются дескрипторы страниц памяти родительского процесса. При этом ядро помечает эти страницы как совместно используемые.

Попытка записи в отображённые страницы (неважно, со стороны родительского или дочернего процесса) вызывает исключение (exception), которое передаёт управление в ядро. Ядро видит, что это обращение было законным, и создаёт копию изменяемой страницы. Таким образом удаётся снизить количество потребляемой программами физической памяти. Механизм COW достаточно сложен в реализации, особенно в многоядерных системах, а ошибки в нём могут приводить к уязвимостям, например, Уязвимость Dirty COW.





## 39. Концепция Микроядерной ОС ***Перечитать!!!***

Микроядерная архитектура является альтернативой структурной организации ОС, в соответствии с которой все основные функции ОС, составляющие многослойное ядро, выполняются в привилегированном режиме. При этом некоторые вспомогательные функции ОС оформляются в виде приложений и выполняются в пользовательском режиме наряду с обычными пользовательскими программами, становясь системными утилитами или обрабатывающими программами. Каждое приложение пользовательского режима работает в собственном адресном пространстве и защищено тем самым от какого-либо вмешательства других приложений. Код ядра, выполняемый в привилегированном режиме, имеет доступ к областям памяти всех приложений, но сам полностью от них защищен. Приложения обращаются к ядру с запросами на выполнение системных функций.

Суть микроядерной архитектуры состоит в следующем. В привилегированном режиме остается работать только очень небольшая часть ОС, называемая *микроядром*. Микроядро защищено от остальных частей ОС и приложений. В состав микроядра обычно входят машинно-зависимые модули, а также модули, выполняющие базовые (но не все) функции ядра по управлению процессами, обработке прерываний, управлению виртуальной памятью, пересылке сообщений и управлению устройствами ввода-вывода, связанные с загрузкой или чтением регистров устройств. Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции операционной системы трудно, если не невозможно, выполнить в пространстве пользователя.

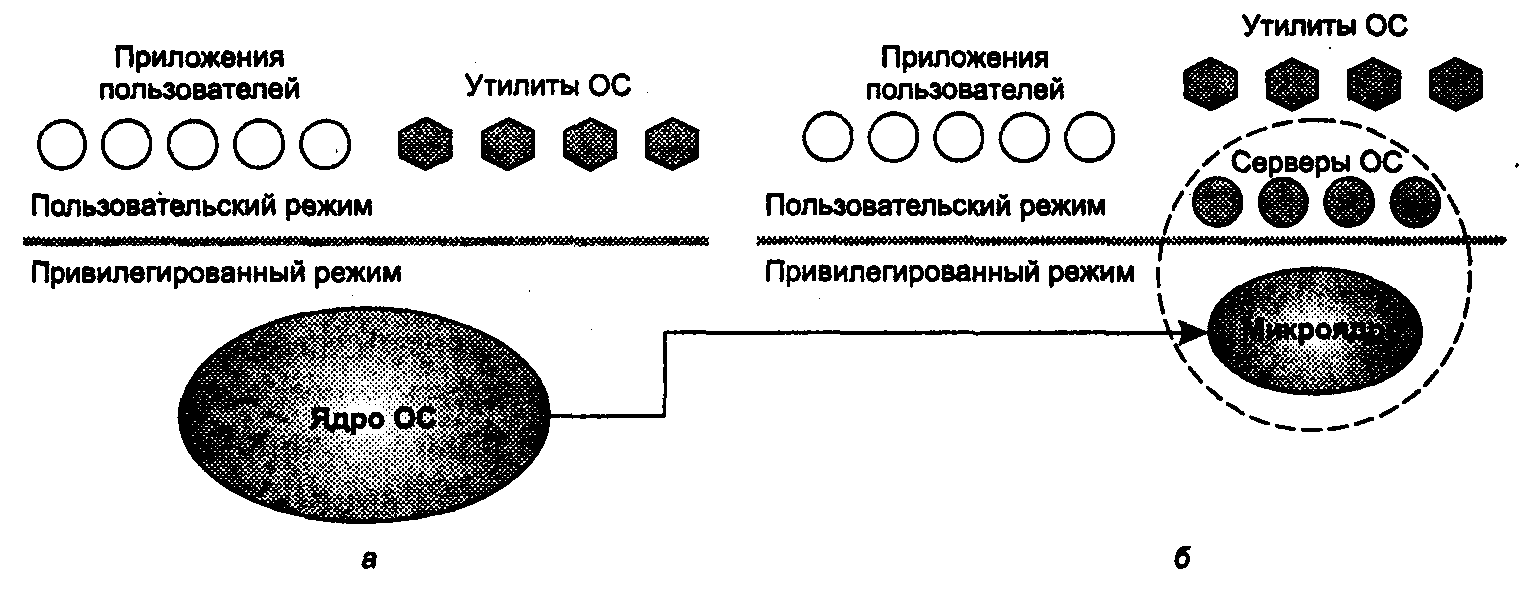


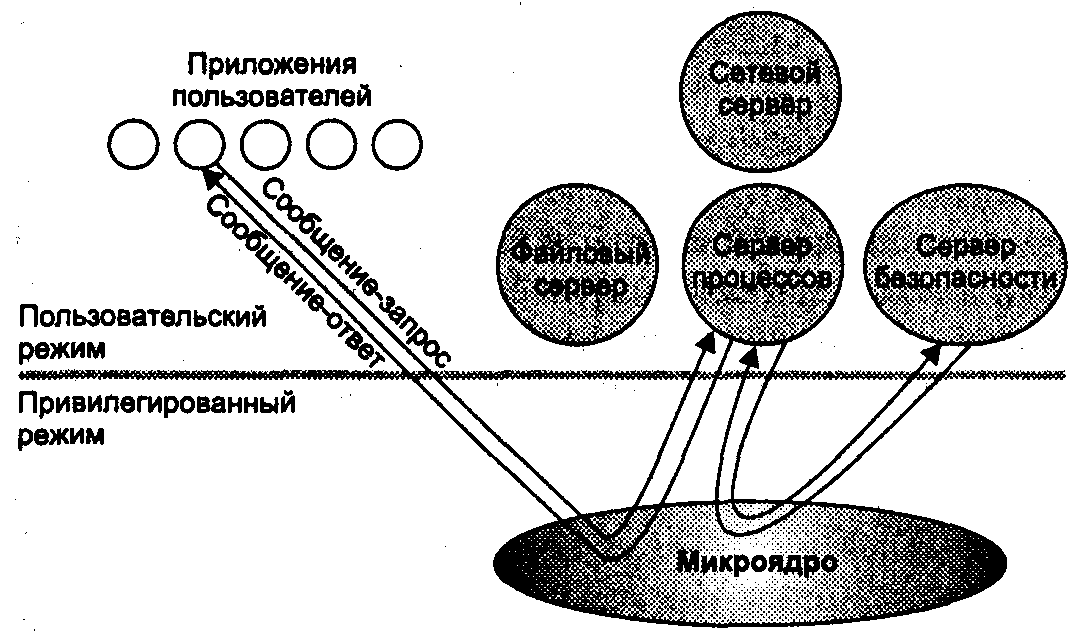
Рисунок - Перенос основного объема функций ядра в пользовательское пространство

Все остальные более *высокоуровневые функции ядра* оформляются в виде приложений, работающих в пользовательском режиме. Однозначного решения о том, какие из системных функций нужно оставить в привилегированном режиме, а какие перенести в пользовательский, не существует. В общем случае многие менеджеры ресурсов, являющиеся неотъемлемыми частями обычного ядра - файловая система, подсистемы управления виртуальной памятью и процессами, менеджер безопасности и т. п., - становятся *«периферийными»* модулями, работающими в пользовательском режиме.

Работающие в пользовательском режиме менеджеры ресурсов имеют принципиальные отличия от традиционных утилит и обрабатывающих программ операционной системы, хотя при микроядерной архитектуре все эти программные компоненты также оформлены в виде приложений.

Утилиты и обрабатывающие программы вызываются в основном пользователями. Ситуации, когда одному приложению требуется выполнение функции (процедуры) другого приложения, возникают крайне редко. Поэтому в операционных системах с классической архитектурой отсутствует механизм, с помощью которого одно приложение могло бы вызвать функции другого.

Совсем другая ситуация возникает, когда в форме приложения оформляется часть операционной системы. По определению, основным назначением такого приложения является обслуживание запросов других приложений, например, создание процесса, выделение памяти, проверка прав доступа к ресурсу и т. д. Именно поэтому менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС, то есть модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей ОС. Очевидно, что для реализации микроядерной архитектуры необходимым условием является наличие в операционной системе удобного и эффективного способа вызова процедур одного процесса из другого. Поддержка такого механизма и является одной из главных задач микроядра.

Схематично механизм обращения к функциям ОС, оформленным в виде серверов, выглядит следующим образом. Клиент, которым может быть либо прикладная программа, либо другой компонент ОС, запрашивает выполне­ние некоторой функции у соответствующего сервера, посылая ему сообщение. Непосредственная передача сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга. Микроядро, выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из этих приложений и поэтому может работать в качестве посредника. Микроядро сначала передает сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения. Таким образом, работа микроядерной операционной системы соответствует известной модели клиент-сервер, в которой роль транспортных средств выполняет микроядро.

Операционные системы, основанные на концепции микроядра, в высокой степени удовлетворяют большинству требований, предъявляемых к современным ОС, обладая переносимостью, расширяемостью, надежностью и создавая хорошие предпосылки для поддержки распределенных приложений. За эти достоинства приходится платить снижением производительности, и это является основным недостатком микроядерной архитектуры.

(преза)

Преимущества микроядра:

* Архитектура микроядра небольшая и изолированная, поэтому она может функционировать лучше
* Микроядра безопасны, потому что включены только те компоненты, которые в противном случае нарушают работу систему
* Расширение системы более доступно, поэтому его можно добавить в системное приложение, не нарушая работу ядра
* Микроядра являются модульными, и различные модули можно заменить, загружать, модифицировать, даже не касаясь ядра.
* Меньше сбоев системы по сравнению с монолитными
* Интерфейс микроядра помогает обеспечить более модульную структуру системы
* Не перекомпилируя, добавление новых функций
* Неисправность сервера также изолирована, как и любая другая пользовательская программа
* Система микроядра является гибкой, поэтому разные стратегии и API, реализованные на разных серверах, могут сосуществовать в системе.
* Повышенная безопасность и стабильность приведут к уменьшению количества кода, работающего в режиме ядра.

Недостатки микроядра:

* Предоставление услуг в системе микроядра является дорогостоящим по сравнению с обычной монолитной системой
* Переключение контекста или вызов функции необходимы, когда драйверы реализованы как процедуры или процессы, соответственно
* Производительность системы микроядра может быть безразличной и может привести к некоторым проблем

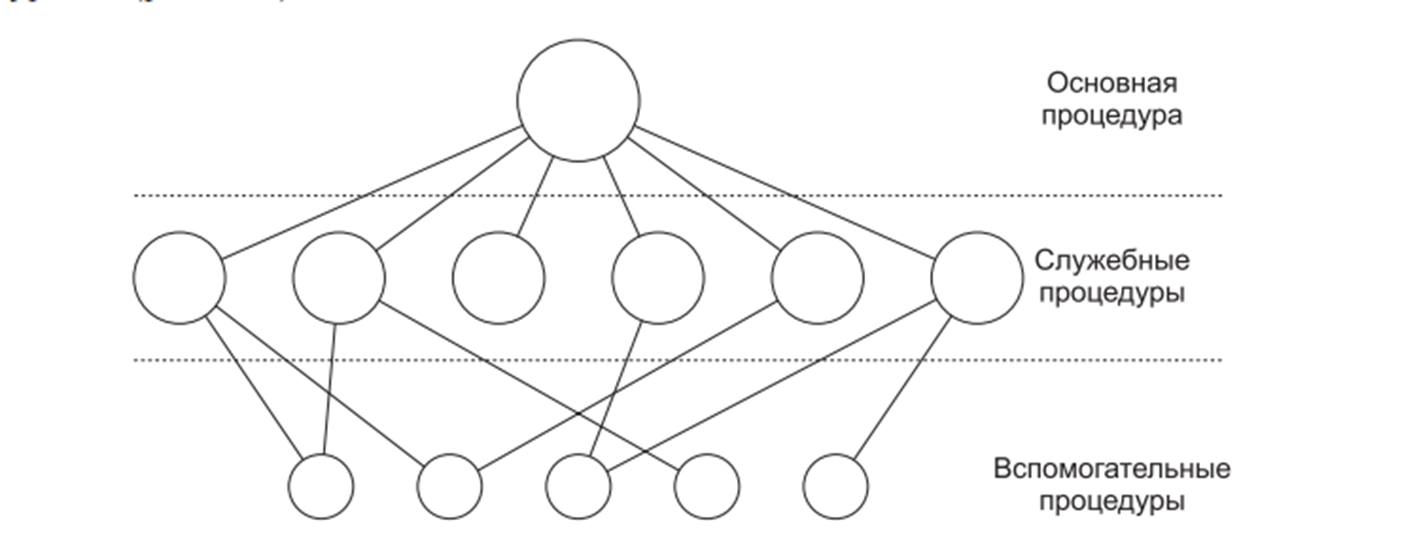
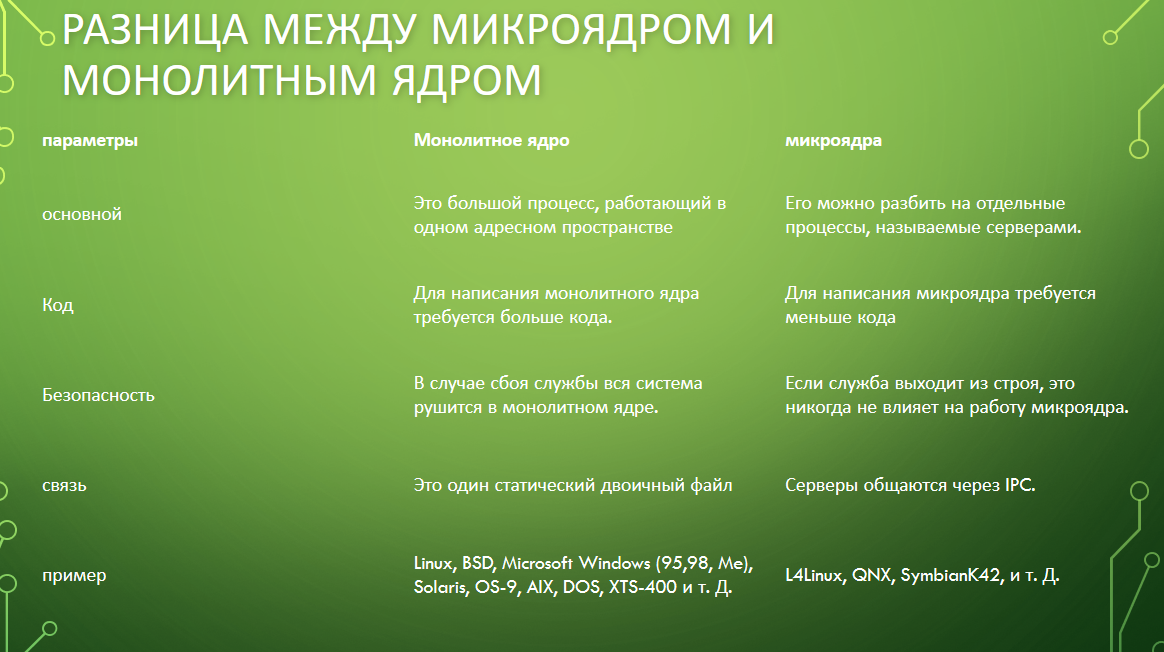
## 40. Типы операционных систем, их особенности

Существует несколько классификаций ОС по критериям:

1. Способ организации вычислений:
   1. Системы пакетной обработки: основной задачей является организация наибольшего количества задач за единицу времени. Определенные процессы объединяются в пакет, который затем обрабатывает ОС.
   2. Системы разделения времени (?): обеспечение возможности единовременного взаимодействия с устройством сразу нескольким пользователям. В порядке очереди каждый пользователь получает определённый промежуток процессорного времени.
   3. Системы реального времени: организация работы каждой задачи за определённый промежуток времени, присущий каждой задаче.
2. Тип ядра:
   1. Монолитные
   2. Микроядерные
      1. Первого поколения (March)
      2. Второго поколения (L4)
   3. Гибридные
   4. Экзоядерные
      1. Rump Kernel
      2. Unikernel
   5. Многоядерные
3. Количества единовременно решаемых задач:
   1. Однозадачные
   2. Многозадачные
4. Количество пользователей:
   1. Однопользовательские
   2. Многопользовательские
5. Количество поддерживаемых процессоров:
   1. Однопроцессорные
   2. Многопроцессорные
6. Возможность работы в сети:
   1. Локальные – автономные ОС, которым не позволено работать с компьютерными сетями
   2. Сетевые – ОС, поддерживающие КС.
7. Роль в сетевом взаимодействии:
   1. Серверные – ОС, предоставляющие доступ к ресурсам сети и осуществляющие управление сетевой инфраструктурой
   2. Клиентские – ОС, которые имеют возможность получения доступа к ресурсам сети
8. Тип лицензии:
   1. Открытые – с открытым исходным кодом
   2. Проприетарные – ОС, связанные с определенным правообладателем, и, как правило, имеющие закрытый исходный код
9. Сфера использования:
   1. ОС мейнфреймов
   2. ОС серверов
   3. ОС ПК
   4. ОС мобильных устройств
   5. Встроенные ОС
   6. ОС маршрутизаторов

## 41. Монолитные операционные системы, их особенности и недостатки

Здесь вся операционная система работает как единая программа в режиме ядра. Операционная система написана в виде набора процедур, связанных вместе в одну большую исполняемую программу



Монолитные ОС являются прямой противоположностью микроядерным ОС. При этом можно согласиться с тем, как трактуется архитектура монолитных ОС в работе. В монолитной ОС, несмотря на её возможную сильную структуризацию, очень трудно удалить один из уровней многоуровневой модульной структуры. Добавление новых функций и изменение существующих для монолитных ОС требует очень хорошего знания всей архитектуры ОС и чрезвычайно больших усилий.

При поддержке монолитных ОС возникает ряд проблем, связанных с тем, что все функции макроядра работают в едином адресном пространстве. Во-первых, это опасность возникновения конфликта между различными частями ядра; во-вторых – сложность подключения к ядру новых драйверов.

## 42. Гибридные ОС

Сочетают элементы микроядерной и монолитной архитектуры. Значительная часть функций, традиционно выполняемых ядром ОС, там и реализовано, но некоторые реализованы в виде отдельных программ, исполняемых в собственных адресных пространствах, что соответствует идеям микроядерной архитектуры. Например, вынести из ядра драйверы устройств, оставив остальные компоненты.

К гибрилным относят MacOS X, Windows NT, DragonFly BSD, поскольку значительная часть системных служб и драйверов данных систем реализуется в виде процессов пользовательского режима. Большинство гибридных ядер поддерживает архитектуру подгружаемых модулей ядра: подгружаемые компоненты гибридных ядер располагаются в вытесняемой памяти и взаимодействуют друг с другом передачей сообщений. В отличие от микроядер, как правило, большинство компонентов ядра работают в одном адресном пространстве.