Оглавление

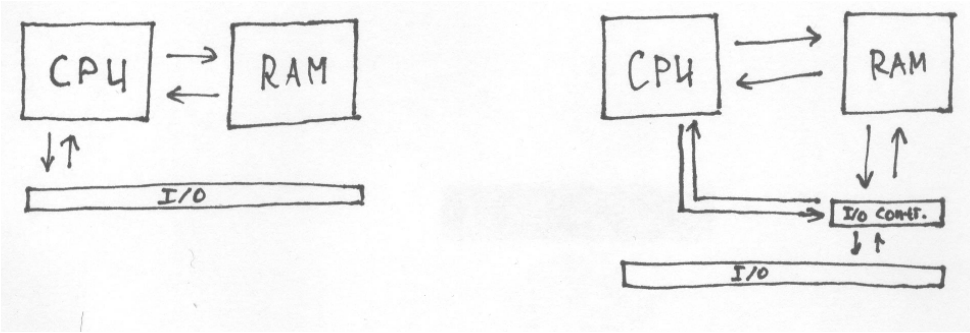
1. Функции и механизмы программ-диспетчеров, предшественников операционных систем

Идеологически ОС формировались в середине 70х. Сами же этапы становления ОС можно разделить на несколько этапов:

- Программы–диспетчеры (50е-60е года)
- Операционная система (конец 50х - 60е года)
- Сетевые / распределенные ОС (70е года)
- Появление мобильных / открытых / универсальных ОС
- Персональные ОС

Термин *ОС* на тот момент еще не существует. В программах-диспетчерах появилась половина того, что реализовано в современных ОС:

1. **Автоматизация загрузки и линковки кода** (работа с повторяющимся кодом). Решение — выделение библиотеки с подпрограммами.
2. **Оптимизация ввода-вывода.** Он изначально был организован через ЦП. Но потом появилась идея сделать специальный контроллер ввод-вывода, который бы организовал работу устройств ввода-вывода напрямую с RAM. Появляется термин «Прерывание» — прерывание процессора по вводу-выводу. При добавлении контроллера, с CPU подавался сигнал на загрузку данных в RAM контроллеру (передавая все необходимые для этого данные), после чего CPU, параллельно с контроллером выполнял свою работу. По завершению работы, контроллер вызывает "прерывание", которое позволило перейти процессору на адрес обработчика прерываний (в промежуток работы программы).



3. **Управление пакетной обработкой**, понятие «*пакета*» - совокупности программ. Появляются первые планировщики, которые вычисляют, какую программу нужно запустить в данный момент (расположение в очереди). Потом все объединили в один пакет "Главная программа"

2. Функции и механизмы мультипрограммных операционных систем

Мультипрограммность – это параллельное выполнение нескольких приложений, программист в этом случае не заботится о механизмах организации параллельной работы, эти функции выполняет ОС. Мультипрограммные ОС, кроме функций однопрограммных ОС, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

По числу одновременно работающих пользователей ОС делятся на:

- *однопользовательские* (MS-DOS, Windows 3.x, ранние версии OS/2)
- *многопользовательские* (UNIX, Windows NT)

Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

Задачами, которые были поставлены перед мультипрограммными ОС являются:

1. Обеспечение разделения процессорного времени. Появляется идея таймеров, по которым происходит прерывание. Появляется механизм для смены контекста, для корректной приостановки выполнения процесса с сохранением стека и регистров (при передаче управления контекст сохраняется, а при возобновлении управления – восстанавливается)
2. Задача виртуализации памяти — абстрагирование адресного пространства от физических адресов. Появился механизм, управляющий виртуальной памятью. Для каждого процесса предоставляется свое адресное пространство, в котором он может работать независимо от других процессов. Адрес памяти не связан с физическим адресом. При обращении к адресу программой, адрес пересчитывается и к нему дается доступ.
3. Системный вызов — механизм обращения к ОС с просьбой предоставления определённых ресурсов, доступных только ОС (например, выделение памяти)
4. Задача управления ОЗУ (выработка стратегии). Управление памятью связано с планировкой процессорного времени. Появляются стратегии управления памятью (что храним в RAM, а что в файле подкачки)
5. Задача разграничения ресурсов
6. Задача коммуникации программ. Появляются механизмы коммуникации (обмен с помощью ОС)
7. Задача синхронизации процессов. *Пример*: на принтер отправляются данные из 2х программ, необходимо управлять потоками так, что бы они не смешивались при переключении контекста.

3. Функции и механизмы сетевых и мобильных операционных систем

Развивается направление благодаря AT&T (из-за развития инфраструктуры компании и необходимости управления большим количеством оборудования). Появляется *распределенность выполнения* (чтобы машины не простаивали и не были перегружены). Появляются механизмы телекоммуникации, которые встраиваются в ОС.

Главными задачами являются разделение ресурсов сети (например, дисковые пространства) и администрирование сети. С помощью сетевых функций системный администратор определяет разделяемые ресурсы, задаёт пароли, определяет права доступа для каждого пользователя или группы пользователей. Отсюда деление:

- сетевые ОС для серверов;
- сетевые ОС для пользователей.

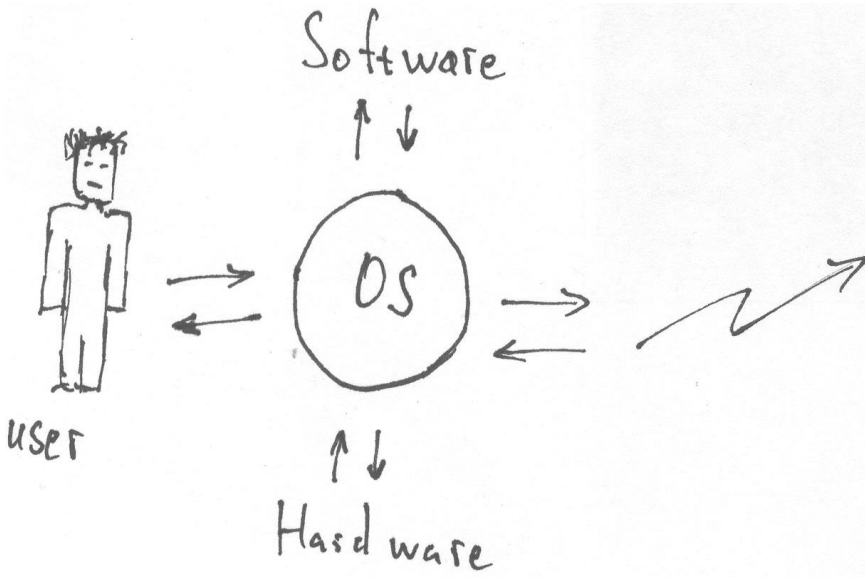
Существуют специальные сетевые ОС, которым приданы функции обычных систем (*Windows NT*) и обычные ОС (*Windows XP*), которым приданы сетевые функции. Сегодня практически все современные ОС имеют встроенные сетевые функции.

4. Задачи и механизмы организации интерфейса между пользовательскими приложениями и аппаратным обеспечением вычислительной системы

Зачем же нужна ОС? Ей основными задачи являются:

1. Обеспечение эффективности, надежности и безопасности выполнения пользовательских программ
2. Обеспечение эффективности, надежности и безопасности использования аппаратного обеспечения
3. Обеспечение эффективности, надежности и безопасности связи с другими системами (обеспечение доступа к телекоммуникационным ресурсам)
4. Обеспечение диалога с пользователями

Задачей ОС является максимальная абстракция пользователя



В связи с этим выделяют ряд характерных функций ОС:

- Обеспечение интерфейса между программным обеспечением, железом и пользователем
    - Поддержка разработки программного обеспечения (предоставление API ОС)
    - Обнаружение ошибок и их отладка
    - Управление по исполнению ПО
    - Обеспечение учета информационных ресурсов

- Обеспечение эффективности использования ресурсов — чем выше производительность, тем меньше надежность и наоборот
- Обеспечение надёжности эксплуатации аппаратных и программных средств.
    - Задача дублирования/резервирования
    - Предсказание сбоя
    - Проведение профилактических работ
- Обеспечение возможности развития ОС.

Обычно у ОС пытаются выделить функциональные подсистемы:

- *Подсистема управления процессором*. С точки зрения ОС, процесс - некая структура памяти (PCB / Дескриптор процесса / Process Controll Block), которая хранит данные о процессе. Существует таблица, которая показывает состояние процессов.
- Подсистема управления памятью - механизмы виртуализации памяти (обеспечение защиты процессов)
- Подсистема управления внешними устройствами
    - Механизм драйверов и автоопределения устройств Plug&Play
- Подсистема управления файлами
    - Механизм преобразования символьных имен в адреса хранения
    - Понятие каталогов и каталогизации файлов
- Подсистема пользовательского интерфейса

5. Методы организации эффективного использования ресурсов компьютера. Критерии эффективности. Управление ресурсами

Для эффективного использования ресурсов необходимо правильно распределять процессорное время между процессами (*планирование*). Для планирования нам необходимо, что бы каждый из процессов обладал каким либо критерием, по которому мы можем решить, который из процессов более важен.

Существует 2 типа критериев:

- *свертка* – каждый процесс оценивается по набору определенных параметров и критерием является сумма оценок.
  $\hat{k} = \alpha * k_1 + \beta * k_2 + \gamma * k_3$

- условный критерий – находим максимальное значение одного критерия, когда задано минимальное значение других.
  $\hat{k} = \max k_1$ при условии, что $k_2 > z_1$ и $k_3 > z_2$

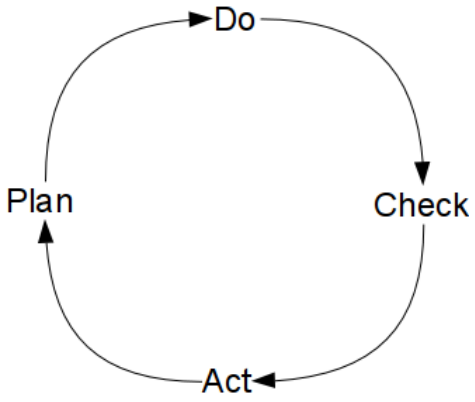
- PDCA (Plan-Do-Check-Act — планирование-действие-проверка-корректировка)

  - Планирование — установление целей и процессов, необходимых для достижения целей, планирование работ по достижению целей процесса и удовлетворения потребителя, планирование выделения и распределения необходимых ресурсов.

  - Выполнение — выполнение запланированных работ.

  - Проверка — сбор информации и контроль результата на основе ключевых показателей эффективности, получившегося в ходе выполнения процесса, выявление и анализ отклонений, установление причин отклонений.

  - Воздействие (управление, корректировка) — принятие мер по устранению причин отклонений от запланированного результата, изменения в планировании и распределении ресурсов.



6. Принципы управления процессами, памятью, файлами

Процесс — это совокупность набора исполняющихся команд, ассоциированных с ним ресурсов, и текущего момента выполнения, находящиеся под управлением ОС. Процесс может не находиться в исполнении в данный момент, однако ресурсы за ним могут быть закреплены.

Процесс для ОС существует тогда, когда структура, содержащую информацию о нем, присутствует у ОС.

Важно понимать, что понятие *процесса* не равносильно понятию *программа*. Одно и то же приложение может порождать множество процессов. И при системных вызовах, с точки зрения ОС процесс ваш, но в нем выполняется код ядра.

Что же касается хранения памяти, то её можно разделить на несколько уровней:

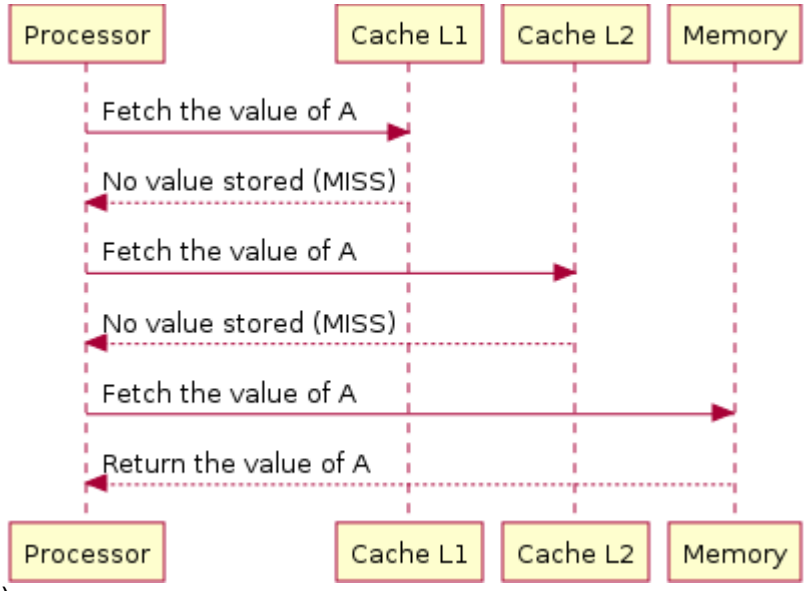
1. *регистры CPU* — ими оперирует непосредственно процессор, наиболее быстрый доступ (порядка 1 такта), но размером лишь в несколько сотен или, редко, тысяч байт.

2. *кэш процессора*, используемые микропроцессором компьютера для уменьшения среднего времени доступа к компьютерной памяти. Одной из проблем является фундаментальная проблема баланса между задержками кэша и интенсивностью попаданий (нахождений искомого фрагмента). Большие кэши имеют более высокий процент попаданий но, вместе с тем, и большую задержку. Чтобы ослабить противоречие между этими двумя параметрами, большинство компьютеров использует несколько уровней кэша, когда после маленьких и быстрых кэшей находятся более медленные большие кэши (в настоящий момент — суммарно до 3 уровней в иерархии кэшей).

    a. *Кэш процессора 1го уровня (L1)* — время доступа порядка нескольких тактов, размером в десятки килобайт

    b. *Кэш процессора 2го уровня (L2)* — большее время доступа (от 2 до 10 раз медленнее L1), около полумегабайта или более

c. *Кэш процессора 3го уровня (L3)* — время доступа около сотни тактов, размером в несколько мегабайт (в массовых процессорах используется недавно)



На рисунке представлена попытка процесса прочитать переменную A из кешей L1 и L2, и, поскольку в кэше их не оказалось, прочитать из RAM.

3. *ОЗУ системы* — время доступа от сотен до, возможно, тысячи тактов, но огромные размеры в несколько гигабайт, вплоть до сотен. Время доступа к ОЗУ может варьироваться для разных его частей в случае комплексов класса NUMA (с неоднородным доступом в память)
4. *HDD / SSD (дисковое хранилище)* — многие миллионы тактов, если данные не были закэшированы или забуферизованы заранее, размеры до нескольких терабайт. При кешировании зависит от реализации кэша.

Функциями ОС по управлению памяти являются:

- Мониторинг свободной и занятой памяти
- Первоначальное и динамическое выделение памяти процессу (динамическое выделение памяти производится по запросу из программы, выполненному системным вызовом)
- Настройка адресов программы на конкретную область физической памяти (виртуализация памяти)
- Защита памяти (невозможность одного процесса обратиться к адресному пространству, выделенному другому процессу)
- Дефрагментация памяти
- *Своппинг* - перемещение кода\данных процесса (или его части) на медленный носитель и обратно
    - Минусы: потеря производительности и невозможность возврата при отсутствии необходимого количества памяти

- Плюсы: отсутствие проблемы переадресации адресов

Файл — однородный объём информации, доступ к которому может быть осуществлен по однозначно определенному символьному имени.

Ко всем потокам ввода-вывода (которые могут быть связаны как с файлами, так и с папками, сокетами и FIFO) можно получить доступ через так называемые файловые дескрипторы. Файловый дескриптор — это неотрицательное целое число. Когда создается новый поток ввода-вывода, ядро возвращает процессу, создавшему поток ввода-вывода, его файловый дескриптор

7. Принципы разработки архитектуры современной операционной системы

Когда идет речь о каком то сложном ПО, то мы делим ПО на определенные блоки/модули/компоненты, каждый из которых выполняет набор задач и обладает определенными интерфейсами. Когда зарождались ОС, вопрос архитектуры особо не решался. Но это продолжалось до того, пока модулей не стало много. Так же отсутствие модулей делает её аппаратно-зависимой. Для того что бы парировать возникающие проблемы, стали выделять не просто компоненты, а стали создавать принципы, по которым нужно делать архитектуру ОС.

Какие принципы лежать в основе современной архитектуры ОС?

- *Модульная организация* (ОС можно представить совокупностью модулей)
- *Принцип совместимости* (способность ОС выполнять программы, написанные для других ОС или для более ранних версий данной ОС, а также для другой аппаратной платформы)
- *Функциональная избыточность* (закладывание в ОС больше функций и возможностей чем нужно на сегодняшний день)
- *Функциональная избирательность* (в ОС выделяется некоторая часть важных модулей, которая постоянно должна находится в ОП для более эффективной организации вычислительного процесса)

Помимо этого в ОС присутствуют утилиты, решающие интерфейсные и файловые задачи: рабочие столы, файл, менеджеры, кодеки и пр. Вопросы того что будет в ядре (из утилит) и легло в основу ОС.

8. Виды архитектур ядер операционных систем

Основными видами архитектур ядер ОС являются:

- Монолитное ядро (билет №9)
- Многослойная монолитная архитектура (билет №10)
- Микроядерная архитектура (билет №11)
- Наноядерная архитектура (ядро только обрабатывает прерывания)
- Экзоядро
- Гибридные ядра

9. Монолитная архитектура ядра операционной системы

Классическая и, на сегодняшний день, наиболее распространённая архитектура ядер операционных систем. Монолитные ядра предоставляют богатый набор абстракций оборудования

Весь код ОС состоит из множества функций, которые могут вызывать друг друга (каждая функция видит и может вызывать каждую)

Слои: *Main Program, Сервисные процедуры, Утилиты ядра (драйверы)*

Плюсы: весь код уходит только на полезную нагрузку, следовательно, высокая скорость работы

Минусы: сбой работы в одном из компонентов может нарушить работу всей системы, монолитные ядра зависят от архитектуры (для каждой новой архитектуры ядро приходится перекомпилировать), ядра очень часто большие и сложные в обслуживании , нет возможности динамического управления драйверами

10. Многослойная архитектура ядра операционной системы

Многослойный подход является универсальным и эффективным способом декомпозиции сложных систем любого типа. В соответствии с этим подходом система состоит из иерархии слоев, каждый слой обслуживает вышележащий, выполняя для него набор функций, которые образуют межслойный интерфейс. На основе функций нижележащего слоя вышележащий слой строит свои более сложные функции. Между слоями существуют строгие правила взаимодействия, а между модулями внутри слоя связи могут быть произвольными. Отдельный модуль может выполнить свою работу либо самостоятельно, либо обратиться к другому модулю своего слоя, либо обратиться за помощью к нижележащему слою через межслойный интерфейс.

Слои:

- Hardware
- Слой аппаратной поддержки ядра
  - аппаратная поддержка прерываний
  - средства поддержки привилегированного режима
  - поддержка виртуальной памяти и ее защита
  - смена контекстов
- HAL (драйверы) (абстракция от конкретных устройств)
- Базовые механизмы ядра
- Уровень управления (менеджеры ресурсов)
- Слой системных вызовов

Плюсы: минимизация издержек на переключение на режим ядра, высокая надежность.

Минусы: ресурсоемкость.

11. Микроядерная архитектура операционной системы.

Является альтернативой классическому многослойному способу построения операционных систем и состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизацией ядра. Взаимодействие между программами операционной системы обеспечивает специальный модуль ядра – микроядро. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью. Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

Микроядро содержит следующие слои:

- Hardware
- Слой аппаратной поддержки ядра
- HAL (драйверы)
- Базовые механизмы ядра

Плюсы: экономия памяти, гибкость.

Минусы:снижение надежности, большие накладные расходы на обращение к ядру.

12. Понятие процесса, потока, нити, задания.

Процесс - это совокупность набора исполняющихся команд, ассоциированных с ним ресурсов, и текущего момента выполнения, находящиеся под управлением ОС. (Каноническое определение).

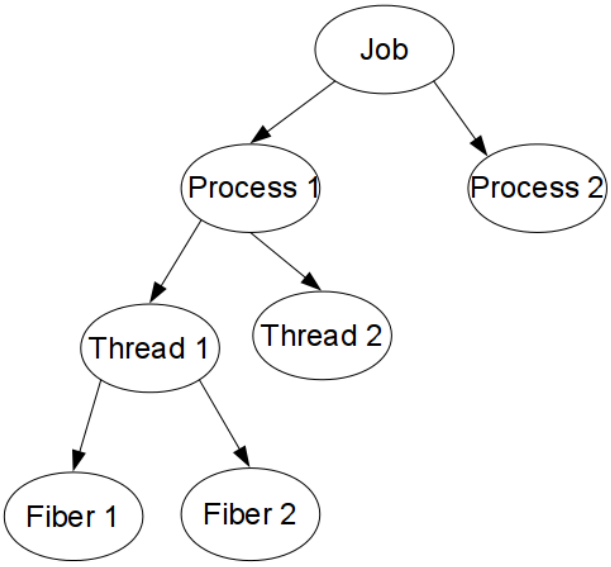
Важно:

1. ОС должна знать все о процессе;
2. процесс для ОС существует тогда, когда есть структура данных о нем у ОС;
3. у процесса могут быть ресурсы, однако процесс может не находится в выполнении.

- **Поток**(thread) - вид процесса, но в отличие от процесса они могут взамодействовать с другими потоками и иметь доступ к одним ресурсам (если потоки из одного процесса). Проблема: рассинхронизация потоков - ОС может отдавать предпочтение одному потоку, а другому не давать выполнится.

- Была предложена система с «**волокнами**», о которых ОС ничего не знает, которые управляются потоком. Волокна(fiber, нить) — подмножество потока. Плюс такой системы — простая реализация синхронизации. Минус — накладные расходы на внутренне планирование вместо выполнения самого кода.

- **Задание** (Job, C-group) — объединение процессов, для которых есть общие ограничения ресурсов (процессорное время, память).



13. Функции подсистемы управления процессами.

1. Создание процессов и потоков (создать определенную структуру данных, связанную с наследованием). Любой процесс порождается другим процессом. Существуют категории данных: PID, PPID, UID (какому пользователю принадлежит этот процесс); Status word (состояние процесса: выполняется, в ожидании, в exception'е, в состоянии завершения, завершился), информация для управления: сколько живет, какие ресурсы ассоциированы с ним, флаги синхронизации процессов

2. Обеспечение процессорных потоков ресурсами (обеспечение ресурсами на старте, а потом в процессе выполнения)

3. Изоляция процессов (безопасность)

4. Планирование выполнения процессов и потоков

5. Диспетчеризация процессов и потоков (Передача управления от одного процесса другому : сохранение текущего регистрового контекста процесса, загрузка регистрового контекста следующего контекста, смена состояний процессора)

6. Организация межпроцессорного взаимодействия (обмен между процессами данными\управлением, например посылка сигналов)

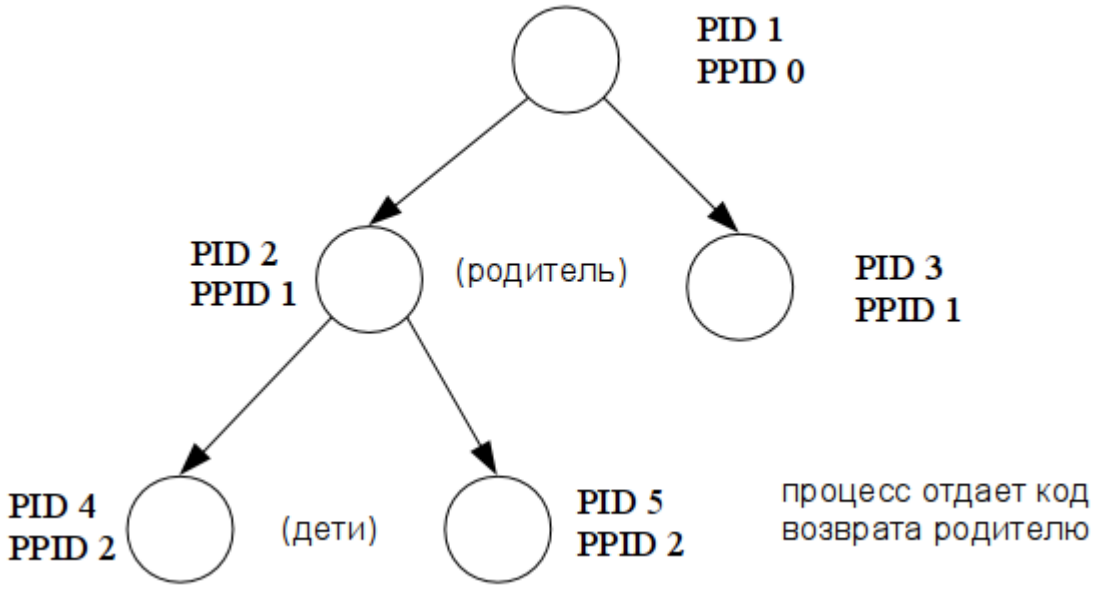
7. Синхронизация процессов и потоков (обеспечение согласованного продвижения процессов вдоль временной шкалы)

8. Завершение и уничтожение процессов и потоков (только ОС имеет доступ к ресурсам, она их и освобождает)

14. Методы создания процессов.

Существуют две модели порождения процессов:

1. Древовидная (каждый процесс знает родителя, родитель отвечает за своих потомков) (UNIX)

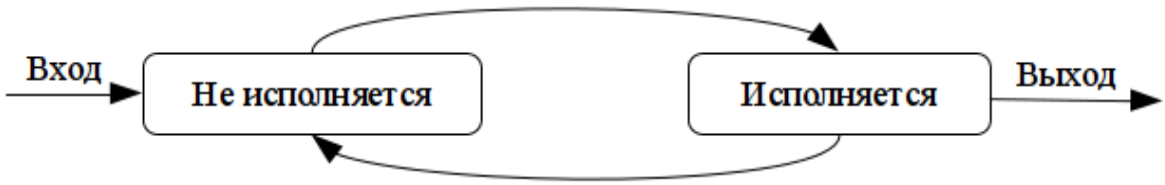


Если родитель завершится раньше дочернего процесса (осиротевший процесс), то его необходимо усыновить (усыновление дедушкой, общественное усыновление, если используется C groups, то усыновляет диспетчер группы). Если в момент усыновления появляется сбой, то получается зомби-процесс, который, с точки зрения ОС, является процессом с временем выполнения и ресурсами, но который не может вернуть код возврата.
Родительский процесс порождает дочерний как собственный клон, у которого он меняет определенные права, а только потом присваивает PID, после чего он становится полноценным процессом.

2. Windows - у любого процесса родителем является единый диспетчер (нет наследования). Диспетчер сам возвращает коды выполнения, которые приходят к нему, получателю. Несет ответственность за все процессы.
При создании процесса в Windows создается чистая структура, которой присваиваются необходимые данные.

15. Модель жизненного цикла процесса.

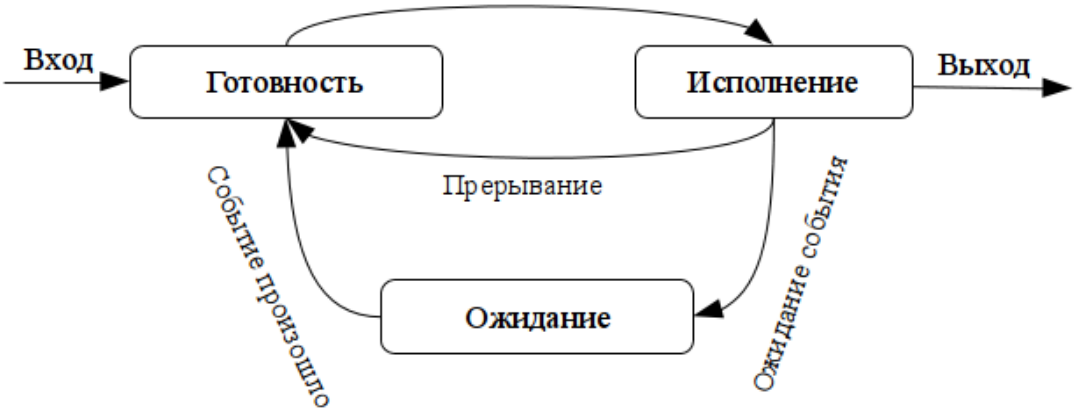
1. Двухуровневая модель



'+' - простое планирование (очередь с приоритетами)

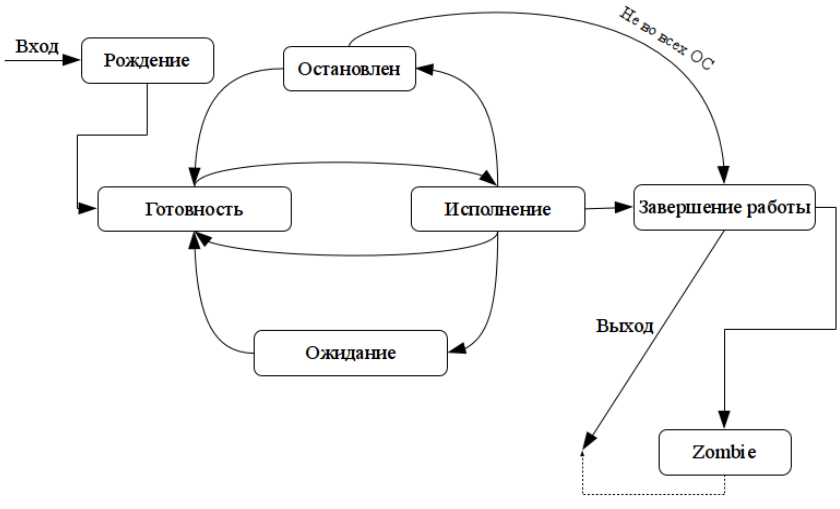
'-' - неизвестно, почему процесс не исполняется

2.Трехуровневая модель



Это Необходимый минимум.
В Windows при переходе из ожидания в готовность увеличивается приоритет.

3. Все ОС, как минимум, имеют **5 уровней**
4-уровневая модель = 3-уровневая + рождение
5-уровневая = 4-уровневая + завершение работы
6-уровневая (Unix) = 5 + Zombie
7-уровневая = 6 + Остановлен (в Windows — подождем, вдруг отомрет)



16. Виды планирования и их место в жизненном цикле процесса.

Виды планирования:

1. **Краткосрочное** (между готовностью и исполнением - определение какому процессу сейчас дать выполняться)

2. **Долгосрочное** (между рождением и готовностью - если мы даем процессу родиться, то состояние операционной системой изменится)

3. **Среднесрочное** (ожидание разделяется на ожидание в оперативной памяти и вне ОП; готовность — на готовность в ОП и на готовность вне ОП)

4. **Планирование отдельных очередей ввода/вывода** (несколько процессов одновременно обращаются к жесткому диску? В очередь их!)

17. Критерии эффективности и свойства методов планирования процессов, параметры планирования процессов.

1. Критерии планирования

    1. Критерий справедливости — каждому процессу гарантируется либо равный доступ к ресурсу, либо доступ пропорциональный какому-либо параметру
    2. Эффективность использования ресурсов
    3. Сокращение полного времени выполнения (времени между рождением процесса и его завершением; обычно вводят среднее время по всем процессам)
    4. Сокращение времени ожидания
    5. Сокращение времени отклика (Работает только для интерактивных процессов и в основном в системах реального времени. Реакция процесса на обращения должна быть минимальна по времени)

Вообще, эти критерии друг другу противоречат.

2. Свойства методов планирования

    1. Предсказуемость — используя один и тот же алгоритм при одних и тех же условиях мы получим один и тот же результат.
    2. Масштабируемость на максимально большое число входных данных (параметров)
    3. Минимальные накладные расходы

3. Параметры планирования

	Параметры вычислительной системы	Параметры отдельного процесса
Статические	Не изменяются в рамках существующего сеанса ОС (до перезагрузки, завершения работы). Аппаратные характеристики, память	Исходный приоритет, права доступа
Динамические	Свободные ресурсы и текущее их состояние	CPU-burst — время непрерывного использования процессора I/O-burst — время непрерывного нахождения в ожидании

18. Дисциплины обслуживания без внешнего управления приоритетами (FCFS, RR, SJF), гарантированное планирование.

- Дисциплина обслуживания — правило, по которому в некоторый момент времени выбирается процесс.

- FCFS — fisrt come, first served — самый простой вариант, обычная очередь, нересурсоемкий

		CPU-burst	Пессимистичный сценарий		
I	p_0	13	uuuuuuuuuuuuu		
II	p_2	4		uuuu	
III	p_3	1			u

t_{full} = 18 (полное время выполнения)

$\hat{t}_s$ = (0 + 13 + 17) / 3 = 10 (среднее время ожидания)

$\hat{t}_f$ = (13 + 17 + 18) / 3 = 16 (среднее полное время выполнения)

		CPU-burst	Оптимистичный сценарий		
I	p_0	1	u		
II	p_2	4		uuuu	
III	p_3	13			uuuuuuuuuuuuu

t_{full} = 18
$\hat{t}_s$ = (0 + 1 + 5) / 3 = 2
$\hat{t}_f$ = (1 + 5 + 18) / 3 = 8

- RR (Round Robin (карусель такая)) — вытесняющий аналог FCFS
Дается фиксированный квант времени (k), потом прерывание

		CPU-burst	k = 4			
I	p_0	13	uuuu			uuuuuuuuu
II	p_2	4		uuuu		
III	p_3	1			u	

t_{full} = 18
$\hat{t}_s$ = (5 + 4 + 8) / 3 = 5,(6)
$\hat{t}_f$ = (18 + 8 + 9) / 3 = 11,(6)

		CPU-burst	k = 1																	
I	p_0	13	u			u		u		u		u	u	u	u	u	u	u	u	u
II	p_2	4		u			u		u		u									
III	p_3	1			u															

t_{full} = 18
$\hat{t}_s$ = (5 + 5 + 2) / 3 = 4
$\hat{t}_f$ = (18 + 9 + 3) / 3 = 10

Чем меньше квант времени, тем оптимистичней сценарий, но тем больше накладные расходы на переключение между процессами.
В среднем, этот алгоритм лучше, чем FCFS. (пессимистичный сценарий лучше, оптимистичный не хуже)

- SJF — shortest job first (с внутренним вычислением приоритетов)
  - Невытесняющий SJF — выбирается процесс с меньшим CPU-burst

	t рождения	CPU-burst				
p_0	0	6			uuuuuu	
p_2	2	2		uu		
p_3	6	7				uuuuuuu
p_4	0	5	uuuuu			

  - Вытесняющий SJF

	t рождения	CPU-burst	k = 2				
p_0	0	6				uuuuuu	
p_2	2	2		uu			
p_3	6	7					uuuuuuu
p_4	0	5	uu		uuu		

- Гарантированное планирование

  Пусть, у нас есть N пользователей.
  $T_i : i \in [1 .. N]$ — время нахождения i-го пользователя в системе
  $\tau_i : i \in [1 .. N]$ — время исполнения программ i-го пользователя

  Время будет распределяться справедливо, если

  $$\tau_i = \frac{T_i}{N}$$

  $$R = \frac{\tau_i * N}{T_i}$$ — коэффициент справедливости.

  Систему с таким планированием просто обдурить.
  Логинишься, развлекаешься пару часов, запускаешь свою программу, тебе дают много процессорного времени.

19. Приоритетное планирование с внешним управлением приоритетами, многоуровневые очереди.

- Невытесняющее планирование, использующее модель приоритезации.

	t рождения	CPU-burst	приоритет				
p_0	0	6	4				uuuuuu
p_2	2	2	3		uu		
p_3	6	7	2			uuuuuuu	
p_4	0	5	1	uuuuu			

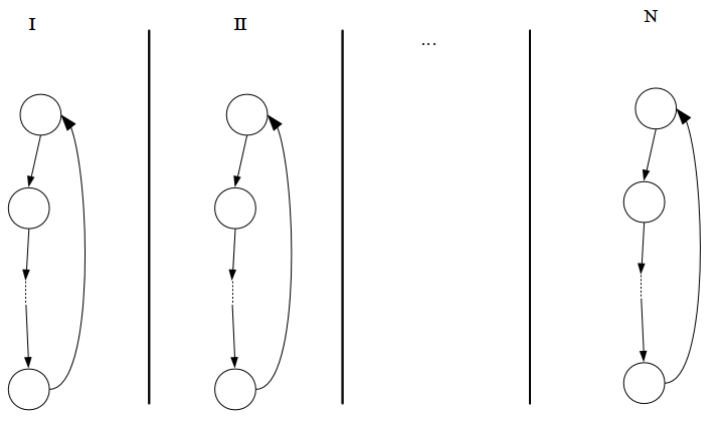
- Вытесняющее планирование, использующее модель приоритезации.

Вытеснение происходит и по прошествию кванта времени, и при появлении процесса с более высоким приоритетом
'–' — низкоприоритетный процесс может вообще никогда не запуститься.

	t рождения	CPU-burst	приоритет					
p_0	0	6	4					uuuuuu
p_2	2	2	3		u		u	
p_3	6	7	2			uuuuuuu		
p_4	0	5	1	uuuuu				

- Вертикальный лифт — если какой-либо процесс не может выполниться из-за своего низкого приоритета, то тогда ему повышают приоритет на 1. Так происходит до тех пор, пока процесс-таки не отработает свой квант. После этого его приоритет приобретает первоначальное значение. Медленно, но верно такой процесс будет выполняться.
- Многоуровневые очереди

  Внутри очереди процессы равноправны (RR)



Если в первой очереди процессы завершились, начинается обработка 2-ой.

- Многоуровневая очередь с обратной связью (Multi-level feedback queue)

Представим, что каждой очереди мы сопоставляем определенный квант времени. Изначально система ставит родившиеся процессы в самую приоритетную очередь. Там процесс выполняется один квант. Если этот процесс является хорошим (ему хватило времени выполниться), то мы оставляем его в этой очереди. Если он выполняется дольше этого времени(является плохим), то его перемещают в очередь с более низким приоритетом. И так далее, пока он не уйдет в последнюю очередь.

Если процесс попал в очередь 3 (например, там дается 32 мс) и 3 раза подряд уложился в 16 мс, то его переместят во 2-ю очередь (16 мс).

20. Организация планирования процессов в Microsoft Windows Vista и GNU/Linux

Windows Vista поддерживает многоуровневые очереди. У него 32 очереди. Очередь с максимальным приоритетом — 31-ая, с минимальным — нулевая. Очереди разделены на две группы: [16..31] (системные процессы), [0..15] (пользовательские процессы). [16..31] — фиксированные очереди, устанавливаемые при рождении, внутри которых нельзя перемещаться.

Для пользовательских процессов используется [0..15]. Первоначальная очередь (очередь куда помещается процесс при старте) определяется приоритетом пользователя. Пользователь же может менять приоритет этих процессов (в пределах [0..15]). Вылезти дальше 15 очереди пользовательский процесс не сможет.
Также система руководствуется логикой:

- Если процесс ожидал I/O:
  - с диска, то его очередь +=1
  - с USB, очередь +=2
  - с клавиатуры — +=4
  - с звуковой карты — +=6
- Поощряются процессы, вышедшие из взаимной блокировки (тот процесс, который сам покинул мьютекс) +=2
- Активное окно — окно на которое перенаправлен I/O:клавиатура, +=1

В GNU/Linux смешаны в кучу все возможные планирования. В нем есть 3 группы процессов:

1. FIFO процессы
2. RR процессы
3. Другие процессы (Other)

- FIFO

  Процессы, для которых не работают вытеснения. (не работают все прерывания). Обычно это используется для системных процессов. Для некоторых системных процессов мы должны сделать, чтобы их не могли прервать (например, если это наш планировщик). Если же есть ошибка в процессе FIFO, и он зациклился, то возможностей его прервать нет (=> крах системы).
- RR

  Равноправные системные процессы. Выполняются, если в FIFO никого нет
- Other

  -20 — max; 20 — min

  По умолчанию пользовательские процессы >= 0. Уменьшить приоритет своих процессов может любой, а как-либо повысить может только root (суперпользователь (администратор системы)).

  Работают по принципу комбинированного приоритета.

Процесс-демон (процесс, оторванный от консоли) получает -5 к приоритету

У каждого процесса есть множитель, который показывает, насколько "хорош" процесс, и умножает на него приоритет

Фактический приоритет процесса = внешний (установлен пользователем) × внутренний (установлен ОС)

21. Проблемы взаимодействующих процессов.

- Взаимоисключение — существует неразделяемый ресурс. Необходимо обеспечить невозможность одновременного доступа двумя разными процессами к нему.

Неразделяемый ресурс — ресурс, который может быть использован только одним процессом в одно время.

- Тупики (взаимоблокировки) (24 билет)

	R_1	R_2
P_0	t_1	t_2
P_1	t_2	t_1

- 'Голодание' процессов — ситуация, когда для одного процесса долго не предоставляется ресурс, предоставляющийся другим.

22. Алгоритмы реализации взаимоисключений.

Допустим, что существует неразделяемый ресурс. Необходимо обеспечить невозможность одновременного доступа нескольких процессов к нему.

Критическая секция — фрагмент кода, в котором происходит обращение к неразделяемому ресурсу.

Race condition (соревнование) — несколько процессов гарантированно столкнутся у критической секции.
Пролог критического входа — критическая секция — эпилог.

Пролог. В нем сообщается, что процесс вступает в критическую секцию и проверяет, чтобы в критической секции не было других процессов.

Эпилог. В нем сообщается о том, что процесс покинул критическую секцию.

Алгоритмы:

1. Запрещение прерываний.

   - Поскольку центральный процессор переключается с одного процесса на другой в результате таймерных или какие-нибудь других прерываний, то при выключенных прерываниях он не сможет переключиться на другой процесс. Поскольку процесс запретил прерывания, он может исследовать и обновлять общую память, не опасаясь вмешательства со стороны любого другого процесса.

   - Неразумно давать пользовательским процессам полномочия выключать все прерывания. Если один из них выключит и не включит прерывания, то система упадет. Если иметь дело с многопроцессорной системой, то запрет прерываний коснется только того процессора, на котором выполняется запретительная инструкция. Все остальные процессоры будут работать в штатном режиме и смогут обращаться к общей памяти.

   - Может использоваться ядром ОС для выполнения нескольких инструкций, например, при обновлении переменных или списков.

2. Использование блокирующих переменных.

   - Shared int lock = 0;

```
Process()
  while (lock == 1);                    \
  lock = 1;                        /   пролог
  { critical section }

  lock = 0;                  > эпилог
```

- При возникновении состязательной ситуации возможен случай, при котором два процесса могут одновременно оказаться в критической области.
  Предположим, что один процесс считывает значение блокирующей переменной и видит, что оно равно нулю. Перед тем, как он сможет установить значение в единицу, планировщик запускает другой процесс, который устанавливает значение в единиуц. Когда возобновляется выполнение первого процесса, он также установит значение блокирующей переменной в единицу, и два процесса одновременно окажутся в своих критических областях.

3. Алгоритм строгого чередования

- Shared int turn = 0;

```
Process()
  while (turn !=  i);
  { critical section }

  turn =  1 - i;
```

- Алгоритм строгого чередования можно использовать в той ситуации, если два процесса будут входить в свои критические области, строго по очереди.
- Возможна ситуация, при которой один процесс будет заблокирован другим процессом, который находится вне критической области. Допустим, первый процесс освободил критическую область, дав возможность другому процессу в нее войти. Но другой процесс уже больше не входит в эту критическую область. => первый процесс оказывается заблокирован. При возможности возникновения такой ситуации, использование этого алгоритма недопустимо.

4. Массив флагов (флаги готовностей)

- Shared int ready[2] = {0,0};

```
Process()
 ready[i] = 1;
  while (ready[1-i] == 1);
  { critical section }

 ready[i] =  0;
```

- Возможна ситуация, в которой оба процесса окажутся в вечном ожидании входа в свои критические области. Допустим, что первый процесс подошел к критической области, установил флаг в 1, а затем планировщик передал управление второму процессу, который также устанавливает свой флаг в единицу. Далее оба процесса не могут войти в критическую секцию, бесконечно ожидая установления флага процесса-конкурента в ноль.

1. Алгоритм Петтерсона (алгоритм взаимной вежливости)

- Shared int ready[2] = {0,0};

Shared int turn;

```
Process()
  ready[i] = 1;
  turn = 1 -  i;
   while (ready[1-i] == 1 && turn == 1 -  i);
   { critical section }

  ready[i] =  0;
```

- Допустим, ни один процесс не находится в критической секции. Затем 0-й процесс устанавливает ready[0] = 1; и turn = 1;. Поскольку ready[1] = 1 0-й, процесс условие в while ложно, и процесс уходит в критическую область. Затем, если 1-й процесс захочет войти в критическую область, то он будет ожидать до тех пор, пока ready[1 - 1] не станет нулем, то есть 0-й процесс не выйдет из критической секции.

- Допустим, оба процесса практически одновременно подошли к критической секции. Они оба будут сохранять номер процесса-конкурента в переменной turn. В расчет будет браться последнее сохранение, поскольку первое будет затерто новым. Допустим, 1-й процесс сохранил 0 в turn последним. Когда оба процесса доберутся до оператора while, 0-й процесс не выполнит его ни одного раза и войдет в свою критическую область. 1-й процесс войдет в цикл и не будет входить в свою критическую область до тех пор, пока процесс 0 не выйдет из своей критической области.

5. Test and Set Lock (проверь и установи блокировку)

- Shared int lock = 0;

```
Process()
   while (TestAndSet(&lock));
   { critical section }

   lock = 0;
```

- TSL считывает содержимое слова памяти lock в регистр, а по адресу памяти, отведенному для lock, записывает ненулевое значение. При этом гарантируется неделимость операций чтения слова и сохранения в нем нового значения — никакой другой процесс не может получить доступ к слову в памяти, пока команда не завершит свою работу. Центральный процессор, выполняющий команду TSL, блокирует шину памяти, запрещая другим центральным процессорам доступ к памяти до тех пор, пока не будет выполнена эта команда

- TSL — атомарная операция.

6. Семафоры Дейкстра

- Дейктра предложил использовать целочисленную переменную для подсчета количества активизация, отложенных на будущее. Он предложил учредить новый тип переменной — семафор. Значение семафора может быть равно 0, что будет свидетельствовать об отсутствии сохраненных активизаций или иметь какое-нибудь

положительное значение, если ожидается не менее одной активизации.

- Также Дейкстра предложил использовать две операции: P и V.

- P(S) (Proberen (пытаться), down) — атомарная операция, проверяющая значение семафора, при ненулевом значении уменьшает его на единицу, а при нулевом приостанавливает выполнение процесса, не завершая в этот раз операцию P.

- V(S) (Verhogen (поднимать), up) — операция, увеличивающая значение, адресуемое семафором, на 1. Если с этим семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции P, то система выбирает один из них и позволяет ему завершить его операцию P. Операция увеличения значения семафора на 1 и активизации одного из процессов также является атомарной.

- Мьютекс — семафор принимающий значения 0 и 1

7. Мониторы Хоара

- Монитор представляет собой коллекцию переменных и структур данных, сгруппированных вместе в специальную разновидность модуля или пакета процедур. Процессы могут вызывать любые необходимые им процедуры, имеющиеся в мониторе, но они не могут получить непосредственный доступ к внутренним структурам данных монитора из процедур, объявленных за пределами монитора.

- В любой момент времени в мониторе может быть активен только один процесс. Мониторы являются конструкцией языка программирования, поэтому компилятор осведомлен об их особенностях и способен обрабатывать вызовы процедур монитора не так, как вызовы всех остальных процедур. Обычно при вызове процессом процедуры монитора первые несколько команд процедуры осуществляют проверку на текущее присутствие активности других процессов внутри монитора. Если какой-нибудь процесс будет активен, вызывающий процесс будет приостановлен до тех пор, пока другой процесс не освободит монитор. Если монитор никаким другим процессом не используется, вызывающий процесс может в него войти.

23. Семафоры Дейкстра. Решение проблемы «производитель-потребитель» с помощью семафоров.

1. См. 22-ой билет, 7-ой пункт.

2. Задача производителя и потребителя (задача ограниченного буфера)
Пусть два процесса используют общий буфер фиксированного размера. Один из них, производитель, помещает информацию в буфер, а другой, потребитель, извлекает ее оттуда.
Проблемы возникают в тот момент, когда производителю требуется поместить новую запись в уже заполненный буфер. Решение заключается в блокировании производителя до тех пор, пока потребитель не извлечет как минимум одну запись. Также, если потребителю нужно извлечь запись из буфера и он видит, что буфер пуст, он блокируется до тех пор, пока производитель не поместит что-нибудь в буфер и не активизирует этого потребителя.

3. Семафор mutex используется для организации взаимного исключения. Семафоры full и empty используются для синхронизации. Они гарантируют, что производитель приостановит свою работу при заполненном буфере, а потребитель приостановит свою работу, если этот буфер опустеет.

4.

```
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
producer()
   while (TRUE)
       item = produce_item();
       P(&empty);
       P(&mutex);
       put_item(item);
       V(&mutex);
       V(&full);
```

```
consumer()
  while (TRUE)
       P(&full);
       P(&mutex);
       item = get_item();
       V(&mutex);
       V(&empty);
       consume_item(item);
```

24. Тупики. Условия возникновения и направления борьбы с тупиками.

1. Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вызвать только другой процесс данного множества. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут бесконечно спать вместе. Тупики могут иметь место как на аппаратных, так и на программных ресурсах.
2. Условия возникновения тупиков
    1. Условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс.
    2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.
    3. Условие неперераспределяемости (No preemtion). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
    4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

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

3. Основные направления борьбы с тупиками
    1. Игнорирование проблемы в целом
    Простейший подход – не замечать проблему тупиков. Для того чтобы принять такое решение, необходимо оценить вероятность возникновения взаимоблокировки и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Проектировщики обычно не желают жертвовать производительностью системы или удобством пользователей для внедрения сложных и дорогостоящих средств борьбы с тупиками.
    2. Предотвращение тупиков
    Цель предотвращения тупиков – обеспечить условия, исключающие возможность возникновения тупиковых ситуаций. Большинство методов связано с предотвращением одного из условий возникновения взаимоблокировки.
    Методы:
        1. Нарушение условия взаимоисключения
        2. Нарушение условия ожидания других ресурсов (отдать все ресурсы, которые есть, если обращаются к заблокированному ресурсу)
        3. Нарушить условие неперераспределяемости (разрешить отбирать ресурсы)
        4. Нарушение условия кругового ожидания
        5. Тщательное распределение ресурсов.
    3. Обнаружение тупиков
    Обнаружение взаимоблокировки сводится к фиксации тупиковой ситуации

и выявлению вовлеченных в нее процессов. Для этого производится проверка наличия циклического ожидания в случаях, когда выполнены первые три условия возникновения тупика.

4. Восстановление после тупиков
Обнаружив тупик, можно вывести из него систему, нарушив одно из условий существования тупика. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы. Сложность восстановления обусловлена рядом факторов.

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

25. Принципы управления памятью вычислительной системы. Виртуальная память и преобразование адресов.

1. Абстрагирование памяти.

Адресное пространство — это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется свое собственное адресное пространство, независимое от того адресного пространства, которое принадлежит другим процессам.
Понятие адресного пространства создает своеобразную абстрактную память, в которой существуют программы.

2. Функции ОС по управлению памятью:

    1. Мониторинг свободной и занятой памяти

    2. Первоначальное и динамическое выделение памяти процессам, освобождение памяти

    3. Настройка адресов программы на конкретную область физической памяти

    4. Полное или частичное вытеснение данных проессов во внешнюю память и возвращение их обратно

    5. Защита памяти

    6. Дефрагментация памяти

3. Использование внешних накопителей

    1. swapping — выгрузка и загрузка процесса целиком на внешний накопитель и обратно.

        - \+ Не нужно решать проблему замены адресов?
        - \- Низкая производительность
        - \- Не факт, что процесс целиком влезет в RAM
        - \+ отсутствует проблема переадресации адресов (вычисление реальных адресов при загрузке в оперативную память, а не вычисление адреса при каждом обращении, используя системные вызовы)

    2. Виртуальная память — часть процесса находится в RAM, а часть на внешнем накопителе

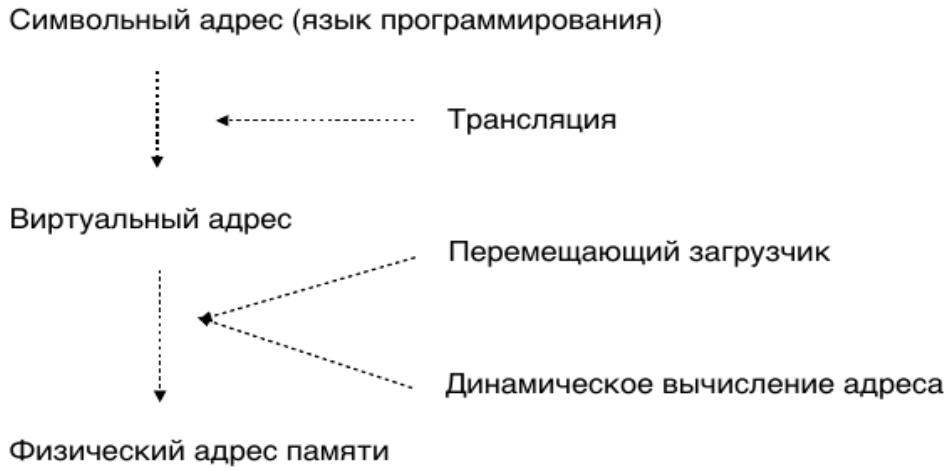
    3. Способы организации

        1. Использование файла подкачки

            - \+ Неограниченность файла
            - \- Если диск забит, то все плохо
            - \- Производительность

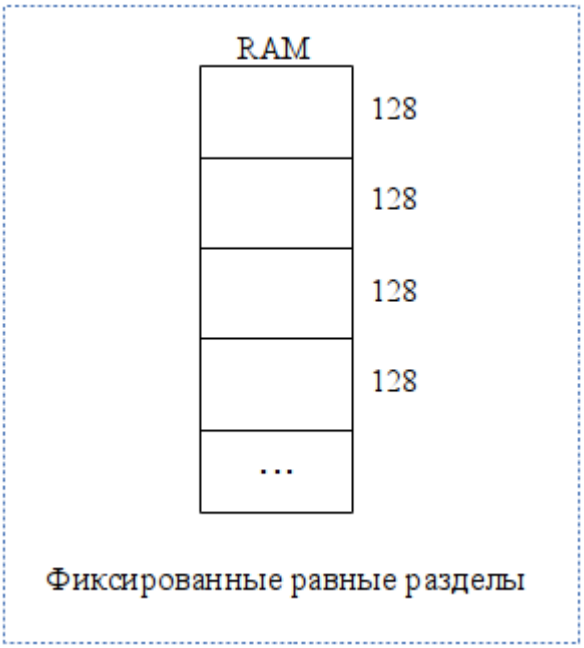
2. Использование раздела подкачки
    - - Размер ограничен
    - + Производительность

4. Преобразование адресов.
    - Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах .
        1. Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код.
        2. Этап загрузки (Load time). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.
        3. Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом.



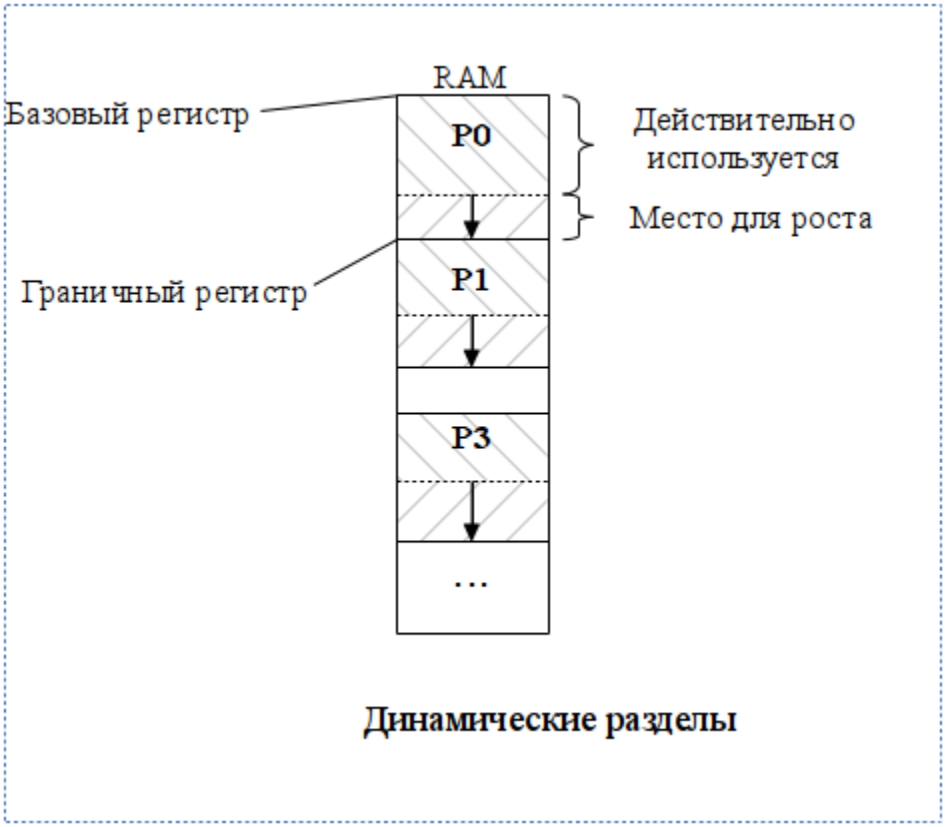
26. Методы распределения оперативной памяти без использования внешней памяти.

1. Фиксированные разделы
    1. Равные
        1. Оверлейное программирование (необходимо использовать, если программа не влезает в блок (ответственность на программисте))
        2. Конечный набор процессов
        3. Нерациональное использование памяти
2. Неравные
    1. Не всегда известно количество необходимой памяти
2. Динамические разделы
3. Перемещаемые разделы
    1. Для каждого процесса хранится 2 адреса: базовый и граничный
    2. Когда перемещаем?
        1. Необходимость в месте для процесса (нужно один раз подождать)
        2. Поддержка фрагментации на приемлемом уровне (система как бы подвисает) (раз в период времени выбирает процесс для перемещения и перемещает)

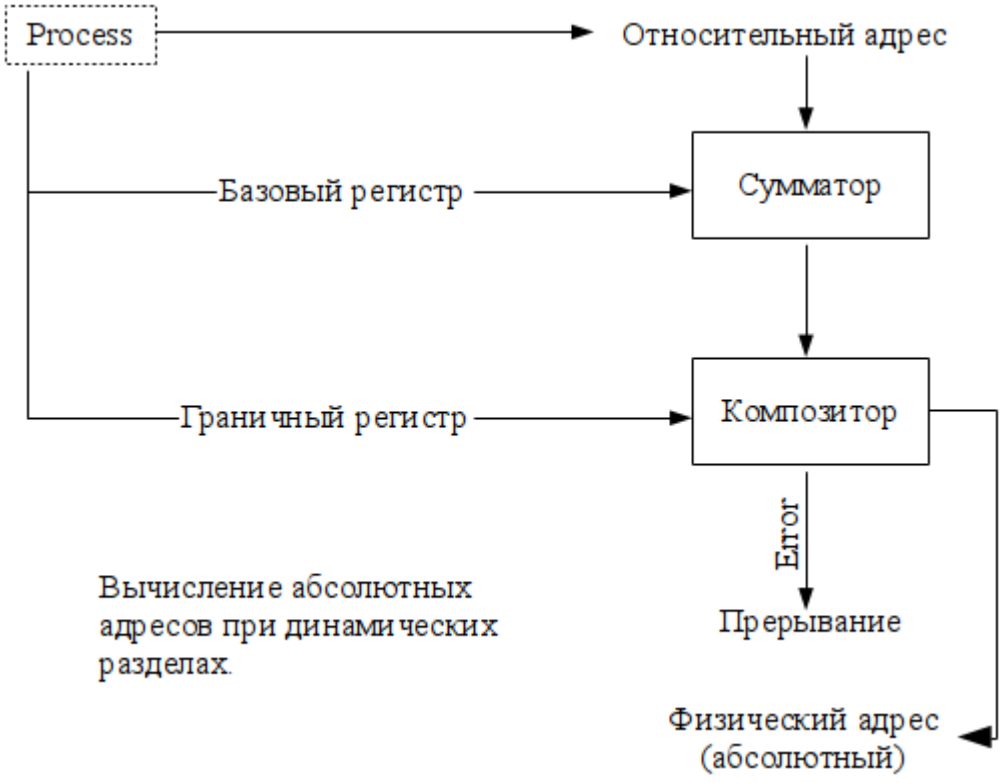


Фиксированные равные разделы

Фиксированные неравные разделы



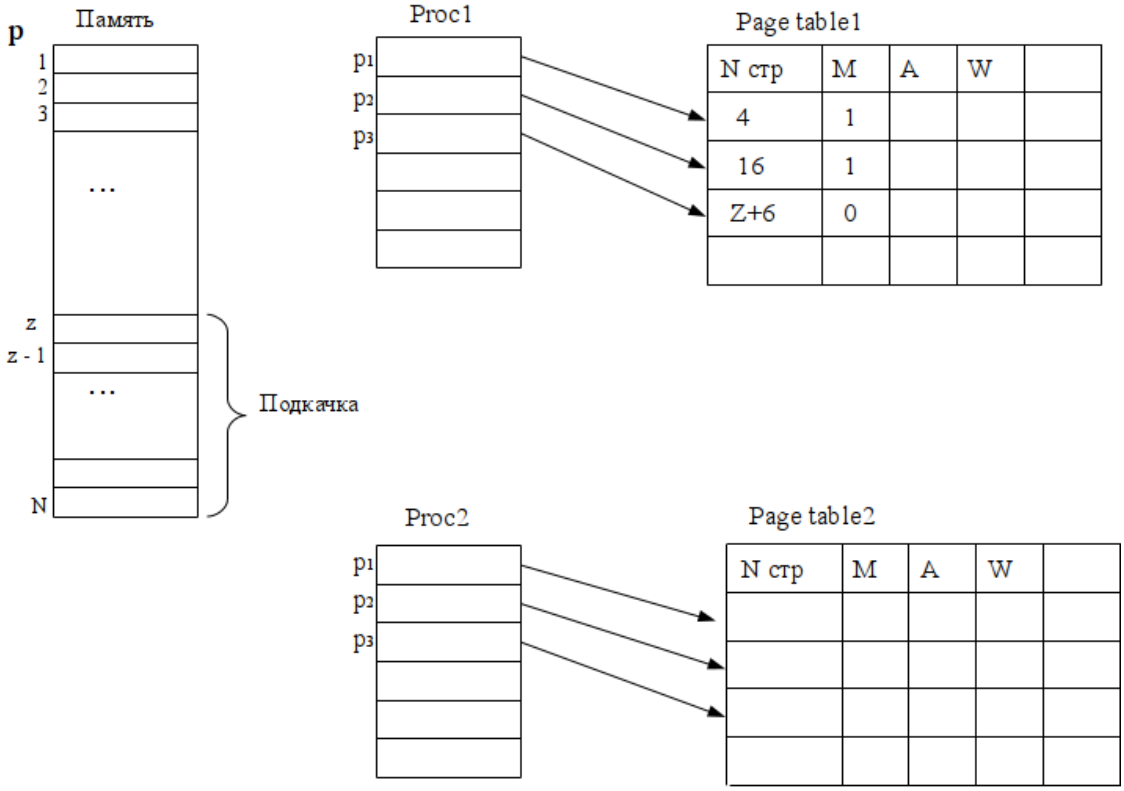
Динамические разделы



Вычисление абсолютных адресов при динамических разделах.

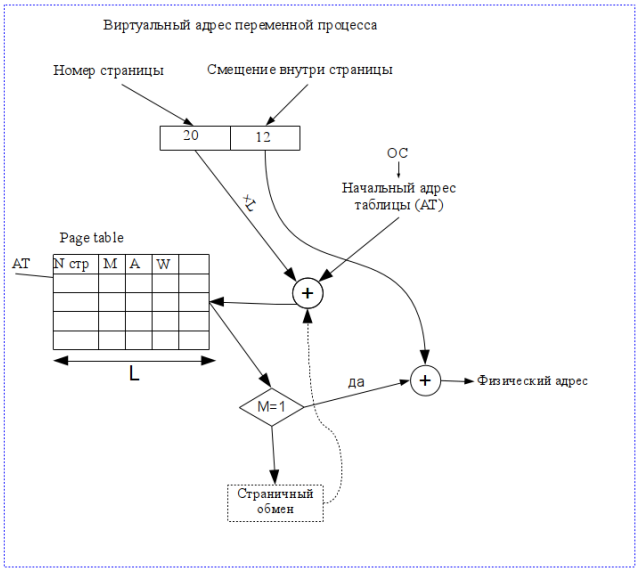
27. Страничная организация виртуальной памяти.

Память разбивается на блоки (страницы). Размер блока — 1КиБ|2КиБ|4КиБ. Может достигать 128КиБ.



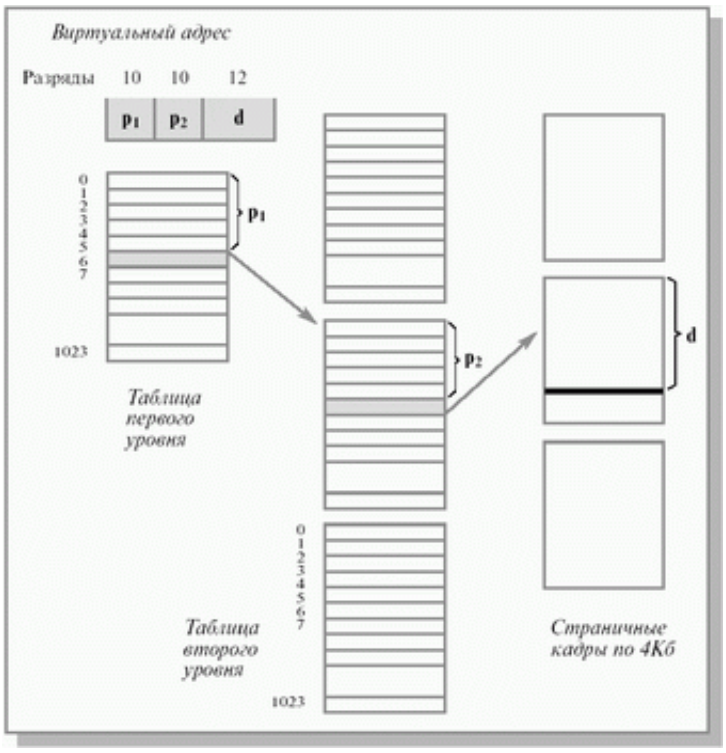
Столбцы таблицы

1. N ссылается на физическое пространство
2. M- в физической памяти / в подкачке?
3. A — обращались ли к фрагменту памяти после последней операции страничного обмена?
4. W — происходило ли изменение данных на этой странице?

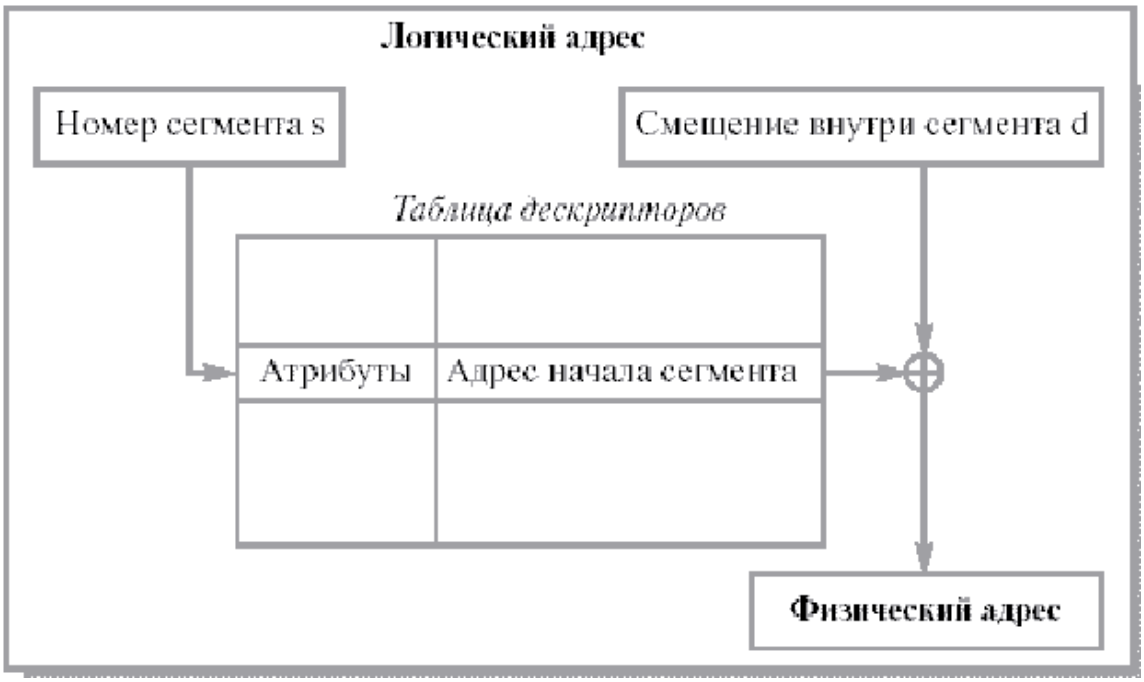


Это происходит при каждом обращении к памяти => необходима оптимизация.
Используется кэширование (аппаратное решение (L3-кэш))

Используется иерархия страниц (разбиение таблицы страниц на таблицы размером в одну страницу, над которыми есть таблица первого уровня.



28. Сегментно-страничная организация виртуальной памяти.

1. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).
2. Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 232 байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.
Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.
3. 

4. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

5. 

6. (К 27-му тоже подойдет) Основные вопросы при страничном и сегментно-страничном распределении

1. Когда передавать страницу в основную память?
    - При загрузке процесса
    - При обращении к странице
2. Где размещать страницу в физическом пространстве?
3. Какую страницу выбрать для замещения (и выгрузить)?
4. Когда давать родиться процессу?

29. 30. Методы выделения дискового пространства и записи последовательности блоков данных: непрерывная последовательность блоков, связный список, таблица размещения файлов, индексные дескрипторы

Файл - некий объем однородной информации, к которой можно получить доступ по определенному символьному имени.

Так же **файл** - объем однородной информации, хранящийся на внешнем устройстве и имеющий дескриптор доступа.

Если мы говорим про ФС, то обычно мы рассуждаем следующим образом:

У нас есть Physical drive. У него есть контроллер. Для контроллера память представляется линией блока. Размер блока 1-4Кб.

Внутри PD существует набор partition'ов (разделов)

Раздел - некое изолированное сборище блоков (в общем случае идущих подряд). На отдельном разделе могут быть созданы одна или несколько ФС, уже в которых расположены файлы.

Задача ОС – знать, где какой блок находится (при обращении).

Если у нас есть еще один диск, то можно создать единую ФС, расположенную на нескольких физических устройствах.

Рассмотрим файловые системы:

1. **Непрерывная**. Файл - непрерывная последовательность блоков. Каталог - некая таблица, в которой есть символьное имя, номер стартового блока, размер и атрибут. Номер стартового блока ссылается на конкретный блок.
+ минимальные накладные расходы
- большие проблемы с фрагментацией
- только последовательная запись
! используется на оптических дисках
! в Audio CD\DVD каждая дорожка - отдельный трек
2. **Связанные диски**. В конце блока данных файла содержится ссылка на номер следующего блока.
+ маленькие накладные расходы
- потеря блока посередине ведет потере ко всей оставшейся части файла
- проблема арифметики при расчете размера
- очень высокая скорость нелинейного доступа
- затруднена скорость работы с большими файлами

3. **Таблица размещения файла** (File location table / FAT). Структуру каталога не меняем, а в памяти храним матрицу 2xN. В этой матрице верхняя строка содержит номера следующих ячеек или EOF. FAT32 - 32бит под хранение номеров. Если ФС больше позволяемого адресовать пространства, то она содержит несколько таблиц, которые делят всю ФС
- жесткое ограничение на размер файла

- большие накладные расходы на оперативную память
- Если грохнуть таблицу адресации то будет большая проблема
! флешку с FAT32 не надо выдергивать без безопасного извлечения, т.к. в момент записи происходит работа с FAT'ом. ОС осуществляет постоянную работу с устройством, поэтому вам может так повезти, что накроется системная область жесткого диска

4. **ext2**. С каждым файлом ассоциируем структуру его размещения. Резервируем участок для дескриптора ФС. Далее резервируем N блоков для дескриптора файлов. Далее резервируем участок битовой карты. И дальше участок данных. В каталоге содержится список файлов с и номерами inode'ов

! В UNIX при изменении файла он не меняется в каталоге
! i-node имеет фиксированный размер => быстрый линейный доступ. В нем содержится большое количество атрибут. В последних 16 словах расположена таблица размещений, говорящих о том, в каких блоках расположен кусок файла. Если 16 блоков данных не хватает, то последним номером из слов будет храниться адрес специального блока в котором хранится следующий набор следующий ссылок (рекурсия)
+ При выдергивании флешки может пострадать только файл, с которым сейчас велась работа
! Стали использовать битовую карту, которая содержит 0/1 - статус занятости/свободности текущего блока. Для нее хранится хеш, который с ней сверяется при каждом обращении

5. **ext3**. По сути ext2 + журналирование. Мы составляем план работы, сохраняем его. При выполнении конкретного пункта вычеркиваем его. При запуске ФС смотрим, не пустая ли очередь. Если не пустая - выполняем план из журнала

- потеря памяти и скорости
+ отказоустойчивость
! правильно хранить журнал на отдельном диске
На самом деле работает 3 модели журналов:
* *WriteBack* - параллельно ведем журнал и выполняет эти действия. В Журнал сами данные не пишутся, а пишутся сами изменения. Тогда журнал получается маленьким
* *Order* - запись реальных данных будет после формирования журнала, но данные в журнал не пишутся
* *Journal* - журналируем абсолютно все (даже данные)

6. **ext4.** Изменены размеры блоков => количество адресов. Размер ФС заложен до 2Эксобайтов

! искусственная дефрагментация - данные пытаются писаться подряд, но в ext 4 резервируют данные после файлов для возможности дописывания. Но мы рассчитываем, что диск не будет забиваться под завязку (примерно 2/3 диска)
! в директории стали хранить хеш файла и использовать b-дерево для поиска
+ битовую карту не страшно потерять

31-36. Все про виртуализацию и синхронизацию

Виртуализация - в общем случае виртуализация абстрагирование вычислительных процессов от реального железа. Более сложная ситуация – абстрагирование ОС

Когда мы говорим про облачные технологии то подразумевается, что внутри - все знают (где что лежит), но из вне это эта информация не доступна.

Потребности / области применения:

- Поддержка устаревших приложений и ОС.
- Повышение отказоустойчивости и надежности за счет управления аппаратными отказами
- Консолидация серверов и повышение управляемости
    - Например, VS на котором запущен коммутатор
    - Если у меня есть набор однотипных серверов (blad’ов/лезвий), то я могу сделать миграцию. И в таком случае я могу прозрачно (незаметно для внешнего мира) перенести сервер на другой blade.

А какая она бывает?

- По принципу действия
    - Full hardware emulation (VBox)
        - -производительность
    - Full virtualization (гипервизор)
    - Виртуализация операционных сред (Virtuoza, OpenVZ на RedHat, CentOS)
        - +производительность
        - -мобильность
    - Комбинация
- По типу виртуализируемой среды
    - Виртуализация представлений (наша задача не изолировать реально процессы друг от друга, а сделать только что бы пользователи не мешали друг другу)
    - Виртуализация рабочих мест (виртуализируем рабочее место)
    - Виртуализация серверов
    - Виртуализация приложений

Гипервизор (Hypervisor) (или Монитор виртуальных машин) —программа или аппаратная схема, обеспечивающая или позволяющая одновременное, параллельное выполнение нескольких или даже многих операционных систем на одном и том же хост-компьютере. Гипервизор также обеспечивает изоляцию операционных систем

друг от друга, защиту и безопасность, разделение ресурсов между различными запущенными ОС и управление ресурсами. Гипервизор строится на ядре какой либо классической системы.

Все, как обычно, упирается в проблему виртуализации: либо сильная виртуализация и потеря производительности, либо облегченная виртуализация, с меньшей изоляцией, но с большей скоростью.

Red Hat Enterprise Virtualization – один из продуктов

1. **Живая миграция (live migration)** – если на одном из аппаратных узлов работает машина, то не останавливая её и подключения к ней, позволяет перенести её на другой аппаратный узел

2. **Высокая доступность** – в случае отказа любой виртуальной машины, в автоматическом режиме происходит воссоздание машины (не убиваемые виртуальные сервера).

3. **Управление питанием** – в каждый момент времени мы можем рассчитать необходимое количество аппаратных узлов, которые должны быть подключены и распределить виртуальные машины между ними. (При этом время отклика не повышает предельную величину). При этом алгоритм смотрит, какая была нагрузка раньше, и распределяет между ними нагрузку для равномерной амортизации.

4. Возможность нескольких виртуальных машин использовать разделяемую память (поддерживается не во всех версиях)

Типичная ситуация:

Сервера обработки образуют **кластеры** (множество лезвий в стойке, которые могут работать под RHEV – Red Hat Enterprise Virtualization), которые связаны с **Сервером хранения данных** (который выглядит и для пользователя работает как внешний диск, но является сложной техникой со своей ОС). Внешние подключения идут через сервер Windows, который, используя БД MSSQL, координирует всю систему целиком.

Основная проблема: производительность, т.к каждая ос принимает самостоятельное решение о ресурсах. Виртуальная машина не знает сколько у нее ресурсов на самом деле, поэтому все алгоритмы планирования оказываются неэффективными.

Когда мы столкнемся с вычислением сложной задачи, неэффективность планирования внутри вирт машин окажется большой, поэтому тут круто распределенную ос делать

Распределенная ОС

Сетевая ОС - у нас существует N копий ОС, но они могут перебрасывать задания от одной к другой (с файлами и ресурсами). Такие системы могут обмениваться процессами. Все ОС независимы и у каждой свое планирование. Касаться их не будем, т.к. ничего супер интересного в них нет

Распределенная ОС одна. У нее множество ядер, а планировщик и сервер памяти общий (не процесс передается с узла на узел, а, например, конкретная страница памяти обрабатывается на конкретном узле)

Мультипроцессорная система - подкласс многопроцессорных компьютерных систем, где есть несколько процессоров и одно адресное пространство, видимое для всех процессоров. Мы не будем касаться их архитектуры, т.к. они сложны и определяются конкретным производителем.

Давайте пообщаемся об *распределенных ОС!*

Задачи

- Пользовательское ПО не знает, где расположены ресурсы
- Прозрачность миграции (перенос с узла на узел, незаметно для приложения)
- Прозрачность размножения - размножать один и тот же процесс на несколько физических машин, а потом сложить результат (опять же с прозрачностью)
- Прозрачность конкуренции - процесс не должен знать, что ему нужно бороться
- Прозрачность параллелизма - от пользовательской программы не требуется дополнительного кода для управления потоками

Получается, что мы должны абстрагировать ПО, с другой стороны мы должны знать что где находится

Проблемы

- Распределенность вычислений (определить кто где должен выполняться)
- Распределенность памяти - любой процессор взаимодействует с RAM, поэтому мы должны с процессом перемещать и RAM
- Распределенность файловой системы

Для решения задачи распределения памяти мы должны понимать, где кто лежит, снизить накладные расходы на перемещение, синхронизировать совместное изменение данных (как узнать кто изменил последним и обновить это)

Алгоритмы

Использование **одного центрального сервера памяти** (в таблице расположения страниц указывать, на каком хосте располагается ОС). Но этот сервер памяти может оказаться узким местом (от его производительности зависит общая производительность). Плюс такой подход эффективен, когда весь процесс с файлами находится на одном узле

Миграционный алгоритм, говорящий о том, что в каждый момент времени есть у каждой машины свой набор актуальных страниц. Когда некоторый процесс дошел до какого-то адреса, то узнав в главной таблице где лежит таблица, то он её мигрирует себе. Может возникать **трешинг**, когда у нас есть часто используемая страница, то она последовательно перемещается от хоста к хосту и это тормозит все ос. Как вариант **keep aliving** – страница в течении тайм аута удерживается на конкретном узле.

Но оба предыдущих алгоритмов полагают, что существует только одна копия процесса. Возникает задача, что если у нас есть система, которая мало пишет, но много читает, тогда для производительности было бы здорово поместить копию на каждой машине. Это **алгоритм размножения для чтения**. Тогда по необходимости *сервер памяти* может делать копию страниц. Но возникает проблема, когда один из хостов изменяет память, а другие об этом не знают. Как вариант, использовать **барьер**, сообщение, которое сообщает всем узлам, что нужно остановиться и изменить область памяти. (При этом останавливаются **все** хосты). Поэтому получается, что архитекторы каждый раз берут и считают, с какого уровня такой алгоритм будет приносить пользу

Алгоритм **полного размещения**, когда все доступны всем на чтение и на запись. Например, когда в СУБД разрешено только добавление записи. Когда такая ситуация, нам не очень важно в каком порядке внесены изменения, важно только знать, кто раньше пришел. Для этого делаем широковещательную рассылку на добавление записи на всех хостах.

Проблема распределения файлов

- Распределение файлового сервера
  - Миграция файла целиком (копирование или перенос, в зависимости от алгоритма)
  - А давайте определенный хост будет **интерфейсом управления определенного файла**. Тогда происходит перенаправление не файлов, а системных вызовов.
- Вопрос каталогов - что такое имя, если все может быть расположена на разных серверах
  - Единое пространство имен, однако обращение идет через единый узел.
  - Но можно сделать что бы в начале пути указывалось адрес хоста, на котором расположен файл

Что у нас тогда получается? Процессы хотят отправлять что то друг другу и получать обратную информацию. Для этого существует набор команд

- **Барьер** - все кроме отправителя должны остановить свою работу
- **Broadcast** – один из набора процессов отправляет всем свои данные
- **Scatter** - если у одного из процессов есть набор команд, то он раскидывает их между процессами
- **Gather** - то же самое, но наоборот (соединяет процессы всех в одни)
- **All Gather** - собирает из всех ко всем
- **All to All** – все процессы ко всем

Синхронизация времени в распределенных системах.

У нас в двух системах выполняется по одному процессу, но которые должны взаимодействовать между собой. Сложность в том, что время обмена может быть большим а время выполнения малое, поэтому возникает вопрос кто отправил сообщение первый?

Идеи решения:

Отдельный сервер, который шлет всем одинаковые сигналы

Математик *Лэмпорт* в 78 году предложил способ синхронизации: каждый процесс ведет независимый отсчет времени. В некоторый такт отправляется сигнал с информацией о текущей итерации. Если у нас процесс, который послал номер своего процесса, имеет номер меньше чем у второго, то у второго номер просто инкрементируется. Если же второй отстает, и его время меньше, то считается что событие пришло в то время, которое сейчас у первого.
Но возникает ситуация, когда два оправляют одновременно: в этом случае считаем, что один из них имеет больший приоритет (не численная нумерация шагов)

Но нам еще и нужен **координатор**. Но возникает ситуация, что если координатор упадет, то система перестанет работать. Давайте тогда предположим, что каждый процесс по запросу может стать координатором. Но если у меня **N** процессов, то как узнать, кто координатор? Тут работает два способа

Алгоритм задиры: любой процесс, который не может найти координатора, посылает по цепочке запрос выбора. Каждый последующий посылает следующему запрос и так далее пока не будет конца. Так же, если координатор воскрес, то он посылает всем информацию, что он стал координатором

Минус: каждый процесс должен знать соседей

37-38. Идентификация и аутентификация пользователей. Авторизация, пользователей, способы разграничения прав доступа.

Login (logical input). Логический вход сопоставляется физическому лицу.

Каждое физическое лицо должно быть идентифицировано – значит, у него должен быть свой идентификатор.

Идентификация - присвоение субъекту некоторого уникального идентификатора.

Типы идентификаторов:

- То, что знает только этот пользователь: пароль — самый небезопасный вариант. Ломается брутфорсом — перебором. Можно перебирать по словарю, и скорее всего это будет эффективно. Можно использовать информацию о пользователе.
- То, чем пользователь владеет: номер телефона, смарт-карта. Можно украсть.
- Атрибут субъекта: сетчатка, отпечаток пальца. Тоже можно попробовать украсть.

Аутентификация - процесс проверки идентификатора

Авторизация - решение вопроса, дать ли пользователю права для совершения действия

Двухфакторная авторизация – авторизация, используя два типа идентификатора.

Тогда получается что у нас есть матрица авторизации [объекты;пользователи] в которой в пересечении стоят права.

Существует два способа распределения прав:

- **Мандатный доступ**: у каждого объекта и у каждого субъекта есть уровень доступа, от их соотношения зависит, есть или нет доступ
- **Дискреционный доступ**: у каждого пользователя просто есть набор прав

Но существует проблема – в матрице очень много пустых значений (*разряженная матрица*). Тогда каждому объекту можно ввести свой лист доступа - ACL.

В unix присутствует 3 группы прав (владелец, группа, другие). Можно подключить модуль selinux и включить ACL.

Так же в unix пошли еще дальше и объединили дискреционный и мандатный доступ: сначала файл проверяется, по группам/ACL , а потом, например, в файлах проверяются мандаты.

Немного о безопасности данных

При хранении данных в любой памяти данные не стираются, они перезаписываются. Даже если вы отформатируете (сотрете оглавление диска) и потом забьете другими файлами, то останется явление остаточной намагниченности, которое уйдет только после примерно шести перезаписываний.