

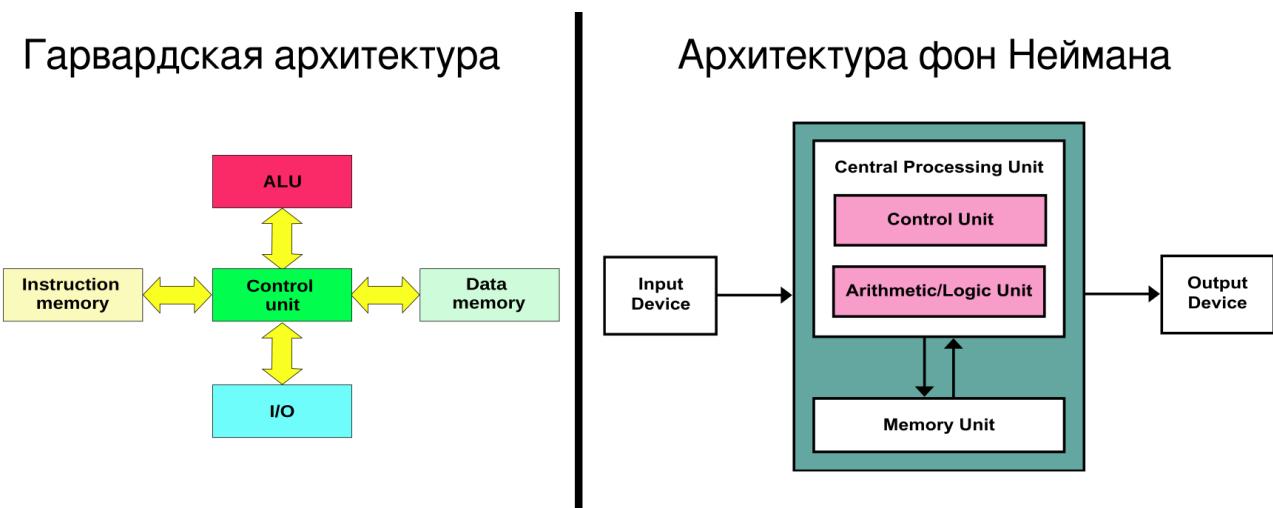
# Экзамен

Другие разобранные вопросы: [https://notesitmo.github.io/cse-notes/third-course/operating-systems/resources/all\\_raspisanno.pdf](https://notesitmo.github.io/cse-notes/third-course/operating-systems/resources/all_raspisanno.pdf)

## 1. Архитектура компьютерных систем. Архитектура Фон-Неймана и Гарвардская архитектура. Принципы архитектуры Фон-Неймана. Архитектуры NUMA и UMA.

---

Исторически в основном говорят о двух видах архитектур -- Гарвардская и Фон-Неймана.



В Гарвардской архитектуре центральным устройством является устройство управления, к которому подключены все остальные устройства и взаимодействуют через него. Память команд и данных физически разделена между собой, что дает нам возможность явно разделять команды и данные между собой. В противовес архитектуре фон-неймана, в которой появился центральный процессор, внутри которого находится управляющее устройство и АЛУ. Память команд и данных единая, что является одним из главных отличий архитектур между собой.

### Принципы архитектуры фон-неймана:

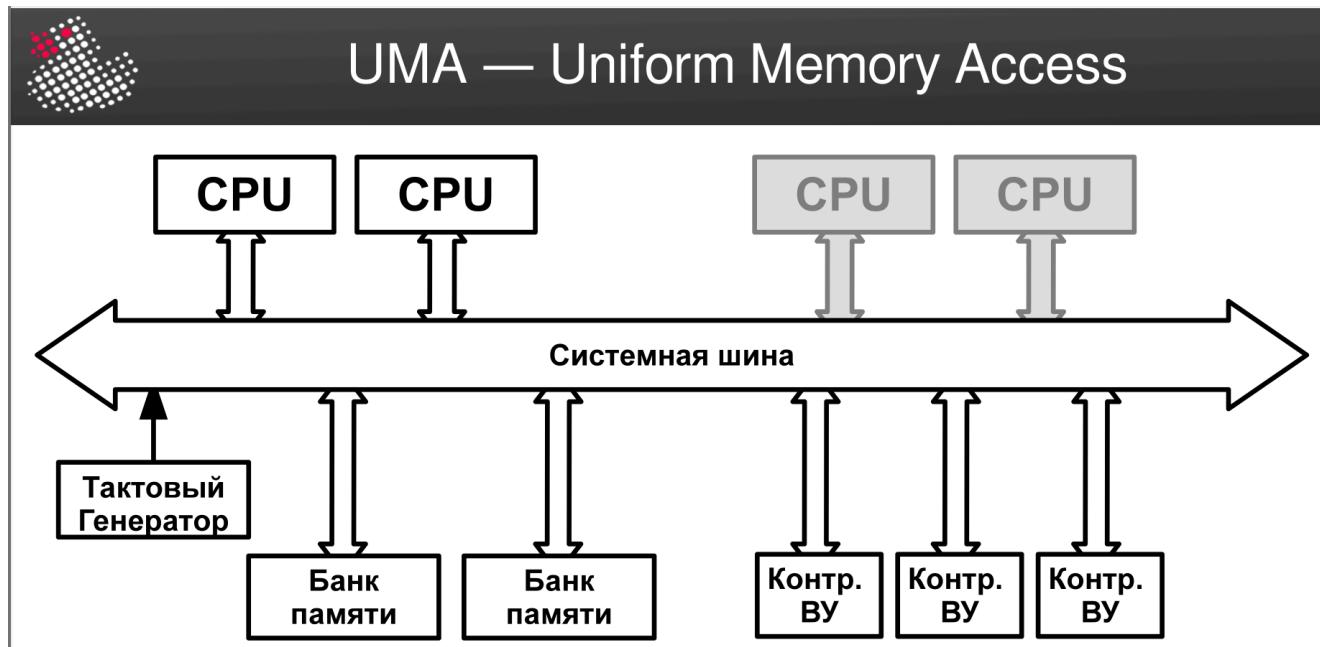
- однородность памяти
- адресность (память состоит из пронумерованных ячеек и процессору доступна любая ячейка)
- программное управление (вычисления представлены в виде программы, состоящего из последовательности команд)
- двоичное кодирование (вся информация кодируется 0 и 1)

## Архитектура UMA (Uniform Memory Access)

Архитектура многопроцессорных вычислительных систем с физической разделяемой памятью, где все процессоры имеют равные возможности по доступу к единому адресному пространству и доступ любого процессора к памяти производится единообразно и занимает одинаковое время.

Простая система.

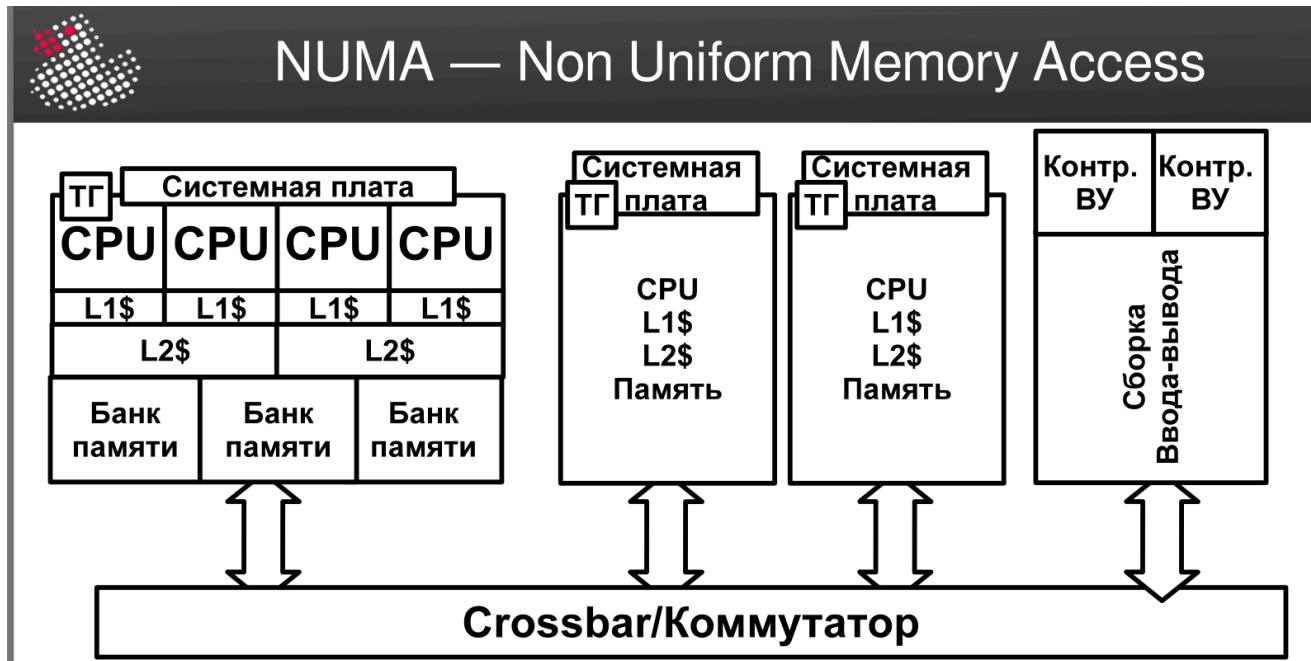
Минус такой архитектуры заключается в плохой масштабируемости. При двух процессорах особых проблем может не возникнуть, но при увеличении числа процессоров могут начаться конфликты и конкуренция за доступ к памяти. Системная шина становится узким местом.



## Архитектура NUMA (Nonuniform memory access)

Архитектурная модель многопроцессорных систем, при которой время доступа к участкам памяти зависит от их топологического расположения. То есть у нас есть набор системных плат, на которых есть несколько процессоров. На каждой плате есть своя локальная память (кэш + банки памяти) и ко всему этому есть доступ к удаленной памяти на другой системной плате через коммутатор (замена системной шине), который позволяет общаться всем платам и сборкам ввода-вывода между собой.

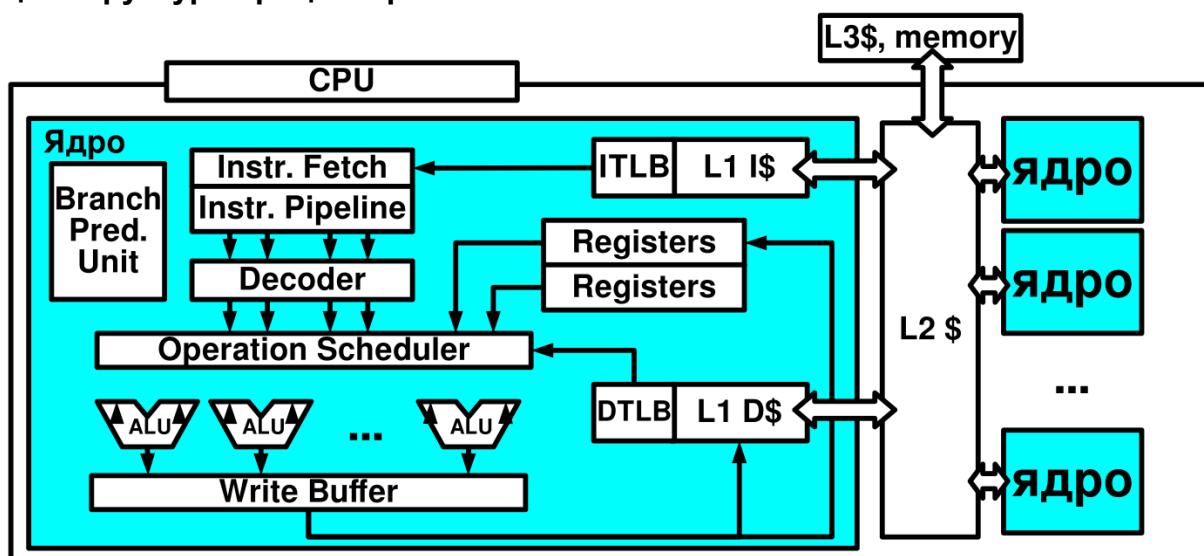
Таким образом, мы получаем масштабируемость системы.



- Адресное пространство общее для всего процессоров.
- Горячая замена системных плат.
- ОС должна уметь работать со всем этим

## 2. Общая организация процессора, памяти, организация вычислений

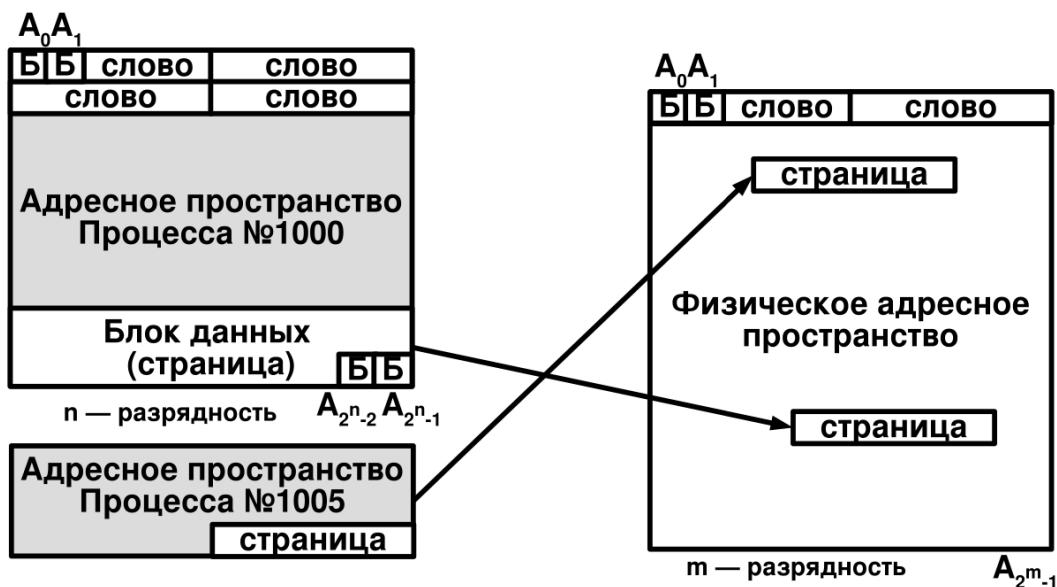
Общая структура процессора



- В процессе есть несколько ядер. Обычно ядра соединены с кэшем L2. В данном случае он общий для всех ядер, но так может быть не всегда
- Также можно получить доступ к кэшу L3

- Кэш первого уровня разделен на: кэш для данных и кэш для инструкций. Для каждого L1 кэша есть свой TLB (translation lookaside buffer). TLB -- это еще один кэш, который ускоряет трансляцию виртуальных адресов в физические. Исходя из принципа локальности большинство обращений к памяти будут сосредоточены в недавно использованных страницах и соответствующие записи уже будут находиться в TLB.
- Некоторые устройства, которые осуществляют выборку команд и декодирование и во многих процессорах также сделана конвейерная обработка инструкций.
- Планировщик задач и набор АЛУ (почему АЛУ так много? зачастую некоторые АЛУ используются для специальных операций например для выполнений действий над числами с плавающей точкой либо векторные операции)
- Несколько наборов регистров для ядер, что получило название в Intel как hyper-threading. В ядре может находиться несколько наборов регистров, которые могут хранить в себе состояние нескольких потоков, что для пользователя и ОС кажется будто у него увеличилось количество ядер
- Branch prediction unit -- это блок предсказания ветвлений в процессоре, который отвечает за минимизацию потерь производительности, возникающих при работе с условными и безусловными переходами (ветвлениями) в коде. Так как в процессоре используется конвейерная обработка, то ошибка предсказания ветвления может стоить очень много времени, чтобы очистить конвейер и загрузить новые инструкции с другой ветви перехода.
- Write buffer нужен, так как память медленная и чтобы не задерживать процессора данные из АЛУ будут записаны в буфер и позже уже слиты в основную память/кэш.

## Организация памяти



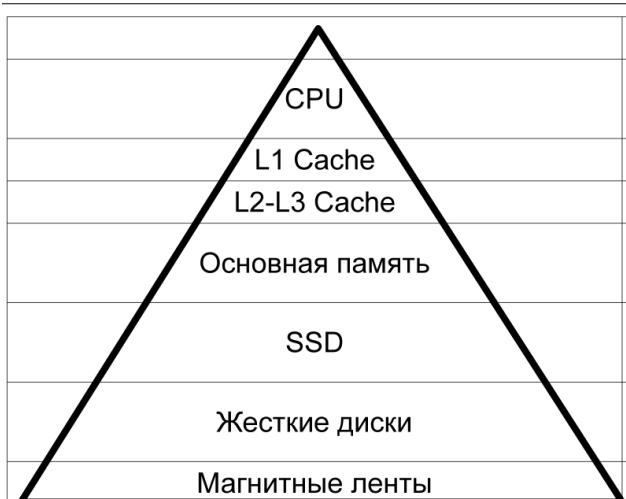
**Виртуальная память** -- это функциональная возможность, позволяющая программистам рассматривать память с логической точки зрения, не заботясь о наличии физической памяти достаточного объема.

**Виртуальная память** -- метод управления памятью компьютера, который позволяет

выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере путем перемещения частей программного между основной и вторичной памятью.

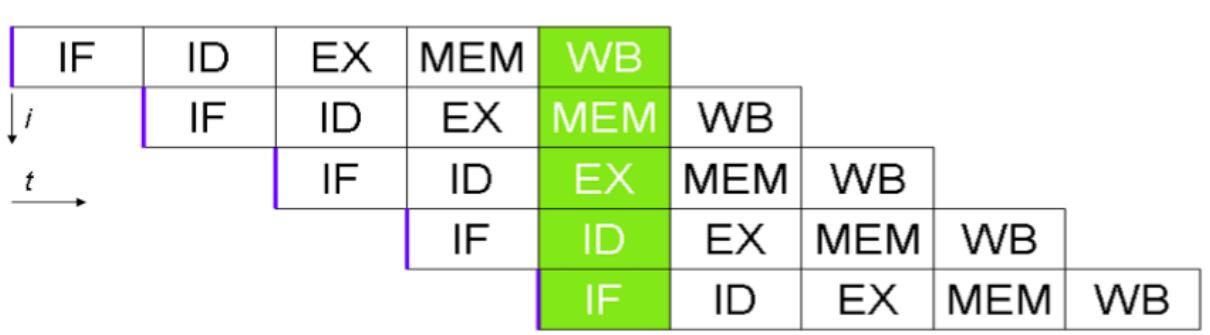
**Виртуальная память** -- схема расположения процессов в памяти, в которой:

- Вторичная память адресуется также как и основная
- Виртуальные адреса транслируются в адреса основной памяти
- Основная память может быть расширена на вторичную
- Размер памяти ограничен схемой адресации и размером вторичной памяти, но не фактическим количеством ячеек
- У каждого процесса свое адресное пространство
- Минимальная единица доступа к данным обычно 1 байт. Все данные объединены в страницы (зачастую по 4кб)
- Виртуальные адреса транслируются в физические с помощью MMU



	Объем	Тд	*	Тип	Управл.
CPU	100-1000 б.	<1нс	1с	Регистр	компилятор
L1 Cache	32-128Кб	1-4нс	2с	Ассоц.	аппаратура
L2-L3 Cache	0.5-32Мб	8-20нс	19с	Ассоц.	аппаратура
Основная память	0.5Гб-4Тб	60-200нс	50-300с	Адресная	программно
SSD	128Гб-1Тб/drive	25-250мкс	5д	Блочн.	программно
Жесткие диски	0.5Тб-4Тб/drive	5-20мс	4м	Блочн.	программно
Магнитные ленты	1-6Тб/к	1-240с	200л	Последов.	программно

### Организация вычислений



Ядро выполняет каждую команду последовательно. Цикл команды

- Выбора команд
- Декодирование
- Исполнение
- Чтение из памяти
- Запись

### Суть конвейерной обработки

Вместо выполнения инструкции последовательно, следующая инструкция может начать свой цикл до того момента пока предыдущая инструкция закончит свой

цикл исполнения. Таким образом, инструкции начинают проходить свои циклы исполнения параллельно. Однако, с этим возникают некоторые проблемы, например:

- скорость исполнения разных команд отличается, поэтому нужен какой-то механизм, который бы затормаживал или приостанавливал начало цикла новой команды.
- наличие условных переходов и необходимость в очистке и выборке команд другой ветви (тут нам помогает branch prediction unit)
- также нужен умный планировщик, который умел бы решать конфликты по данным, когда результат следующей операции зависит от предыдущей (которая может еще идти, а вторая операция может уже заканчиваться)

### **3. Организация прерываний, типы прерываний, контроллер прерываний**

---

**Прерывание** -- Приостановка процесса, такого как выполнение компьютерной программы, вызванное событием, внешним по отношению к этому процессу, и выполняемая таким образом, что процесс может быть возобновлен. Это может быть сигнал от программного или аппаратного обеспечения о наступлении какого-то высокоприоритетного события, на которой необходимо обратить внимание после чего возвращается в состояние до наступления сигнала прерывания.

- Выполняются в конце цикла команды
- Могут быть вызваны самой программой либо каким-то внешним устройством ввода-вывода
- Прерывания могут быть вложенными (с ограничением на вложенность) и также могут иметь приоритеты

#### **Основные этапы обработки прерываний**

1. Инициация прерывания -- устройство или программа отправляет сигнал прерывания через линию прерываний (Interrupt request line, IRQ)
2. Сохранение состояния процессора
  - Завершение текущей инструкции
  - Сохранение счетчика команд и регистр флагов, чтобы вернуться к исполнению программы после обработки прерывания
3. Определение источника прерывания -- процессор или контроллер прерывания, например PCI (Programmable interruption controller) определяют устройство, которое вызвало прерывание
4. Переход к обработчику прерывания (ISR)
  - Процессор передаёт управление функции обработки прерывания (Interrupt Service Routine, ISR).

- Адрес ISR обычно берётся из таблицы вектора прерываний (Interrupt Vector Table, IVT), где хранятся указатели на обработчики.

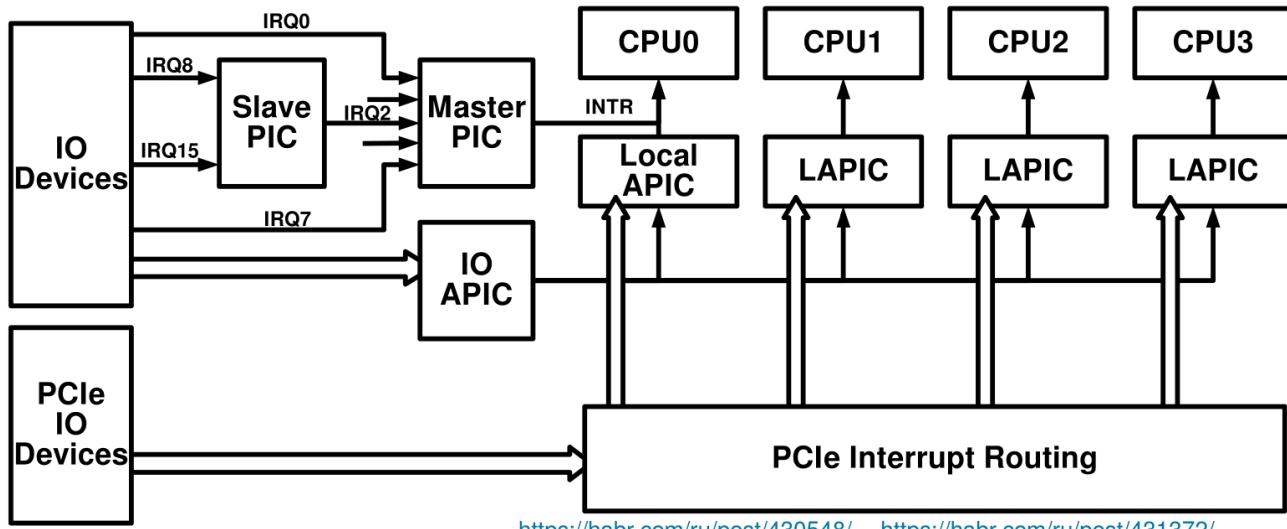
## 5. Обработка прерывания

- ISR выполняет код для обработки события, например, считывание данных с устройства, подтверждение прерывания и т.д.

## 6. Восстановление состояния процессора

- После завершения ISR процессор восстанавливает сохранённое состояние.
- Возвращается к выполнению программы с места, где она была прервана.

## Контроллер прерывания x86



Подробнее здесь: <https://habr.com/ru/articles/430548/>

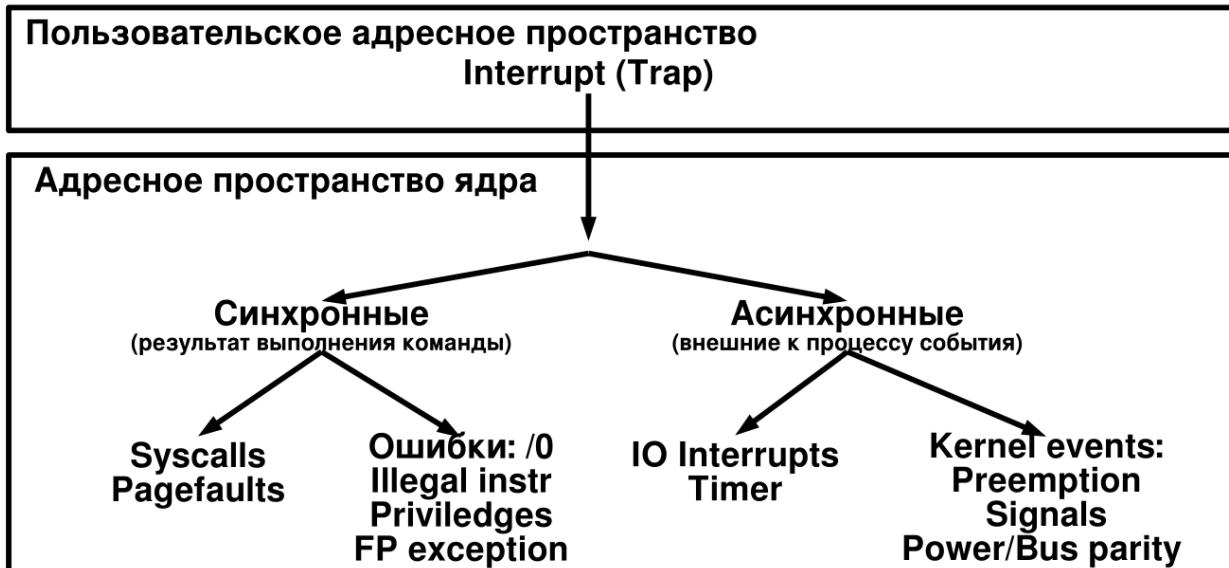
- **PIC** -- аппаратное устройство, используемое в компьютерах для управления запросами прерываний от различных периферийных устройств и передачи их процессору. Его основная задача — координировать прерывания и управлять их приоритетами, чтобы обеспечить корректную обработку событий.
- **Slave PIC** и **Master PIC** были сделаны для увеличения количества IRQ. Линиями 0-7 управлял Master PIC, линиями 8-15 Slave PIC и только Master PIC мог подать сигнал INTR
- Позже шина ISA сменилась на шину PCI и устройств стало намного больше чем IRQ, однако к IRQ стали разделяемы так что к ним можно подключить несколько устройств, поэтому несколько линий прерываний стали объединять в одну и подключать к IRQ.
- **APIC** (Advanced PIC) -- усовершенствованный аппаратный механизм для управления прерываниями в современных системах с поддержкой многопроцессорной архитектуры (SMP). **APIC** состоит из двух основных частей **IO APIC** и **Local APIC**.
- **IO APIC**
  - Контролирует внешние (аппаратные) прерывания от периферийных устройств. Маршрутизирует прерывания на соответствующие процессоры или ядра.

- Поддерживает большое количество IRQ. Маскирование и изменение приоритетов прерываний.
- Возможность использования MSI (Message Signaled Interrupts) вместо традиционных сигналов

- **Local APIC**

- Встроен в каждый процессор и отвечает за обработку прерываний, относящихся к конкретному ядру.
- Взаимодействует с IO APIC для маршрутизации внешних прерываний.

### Типы прерываний



**Синхронные** -- возникают в результате выполнения текущей инструкции процессором. Они заранее предсказуемые, так как происходят в заранее определенных точках программы и является результатом выполнения команды (нарушение каких-то условий при исполнении кода), например, деление на ноль, обращение к недопустимому адресу и так далее

**Асинхронные** -- классический тип прерываний, который исходит от периферийных устройств в непредсказуемое время. Нажмите клавиши мыши или клавиатуры, сигнал таймера, сигнал от сетевого адаптера о том, что пришел какой-то пакет и так далее. Формируется сигнал и запрос на прерывание, на которое ОС должна обратить внимание.

## 4. Типичные функции ОС. Интерфейсы ОС. Работа ОС как замена оператора ЭВМ.

---

**Операционная система** -- ПО, которое контролирует выполнение программ и обеспечивает такие службы, как распределение ресурсов, планирование, управление вводом-выводом и данными. Исполняет роль интерфейса между приложениями и

аппаратным обеспечением.

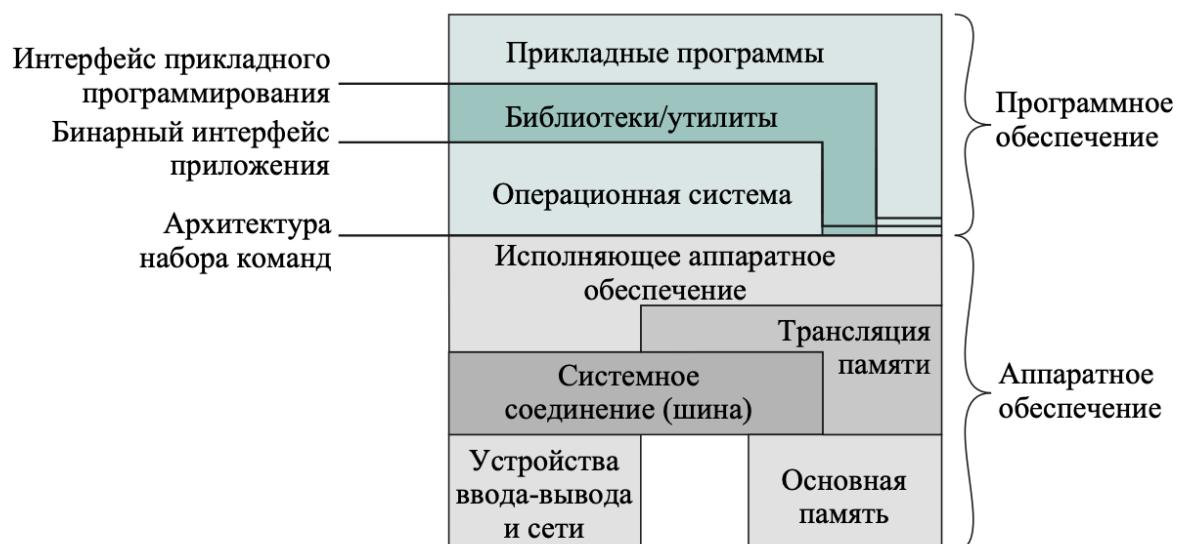
### Предназначение ОС:

- Удобство (ОС делает использование компьютера простым и удобным)
- Эффективность (ОС позволяет как можно эффективнее использовать ресурсы компьютерной системы)
- Возможность развития

### Типичные функции ОС:

- Разработка программ (содействие программистам при разработке программ, предоставление программистам разнообразные инструменты и службы, например редакторы или отладчики. Обычно эти службы реализованы в виде программ утилит, которые поддерживаются ОС, но не входят в состав ядра)
- Выполнение программ (ОС занимается рутинной работой вместо пользователя. Загрузка команд и данных в память, инициализация устройств ввода-вывода и подготовка других ресурсов)
- Доступ к устройствам ввода-вывода (ОС скрывает все детали реализации и дает пользователю единообразный интерфейс для работы с устройствами IO с помощью простых команд чтения и записи)
- Контролируемый доступ к файлам
- Системный доступ (Обеспечение защиты ресурсов и данных от несанкционированного использования, а также разрешение конфликтных ситуаций)
- Обнаружение ошибок и их обработка (как программных, так и аппаратных. В каждом из этих случаев ОС должна выполнить действия, которые сводят влияние ошибки на работу системы к минимуму)
- Учет использования ресурсов

### Интерфейсы ОС



**Рис. 2.1.** Структура программного и аппаратного обеспечения компьютера

- **ISA (Instruction set architecture)** Структура системы команд -- определяет набор команд машинного языка, которые может выполнять компьютер. Находится на границе между программным и аппаратным обеспечением. И прикладные

программы и утилиты имеют доступ к ISA + у ОС есть доступ к дополнительным машинным командам, которые относятся к управлению ресурсами системы.

- **Бинарный интерфейс (ABI (application binary interface))** -- определяет стандарт бинарной переносимости между программами. ABI определяет интерфейс системных вызовов ОС и аппаратных ресурсов и служб, доступных системе через пользовательскую ISA
- **Интерфейс прикладного программирования (API)** -- API обеспечивает программе доступ к аппаратным ресурсам и службам, доступным в системе через пользовательскую ISA с библиотечными вызовами на языке высокого уровня. Обычно любые системные вызовы выполняются через библиотеки. Применение API обеспечивает легкую переносимость прикладного программного обеспечения на другие системы, поддерживающие тот же API, путем перекомпиляции.

## Работа ОС как замена оператора ЭВМ

Раньше компьютеры управлялись с помощью пультов управления, состоящих из тумблеров, сигнальных ламп, некоторого устройства ввода вывода и принтера.

Программы загружались через устройство ввода данных (например, устройства ввода с перфоркарт). У оператора были следующие задачи:

- получить программу с данными от программиста
- подготовить программу к загрузке
- загрузить программу и компилятор
- запустить программу на вычисление
- распечатка результатов и передача ее программисту

Минусы такого подхода:

- расписание работы -- нужна была предварительная запись на машинное время. Обычно пользователь мог заказать время (условно на час) и закончить работу раньше, что приводило к простою машины
- время подготовки к работе -- очень долгий процесс запуска задания, которое нужно было загрузить в память, сохранить компилятор и позже скомпилированный код. Для каждого из этапов могли понадобиться магнитные ленты или колода перфокарт, а при возникновении ошибки весь процесс начинался заново

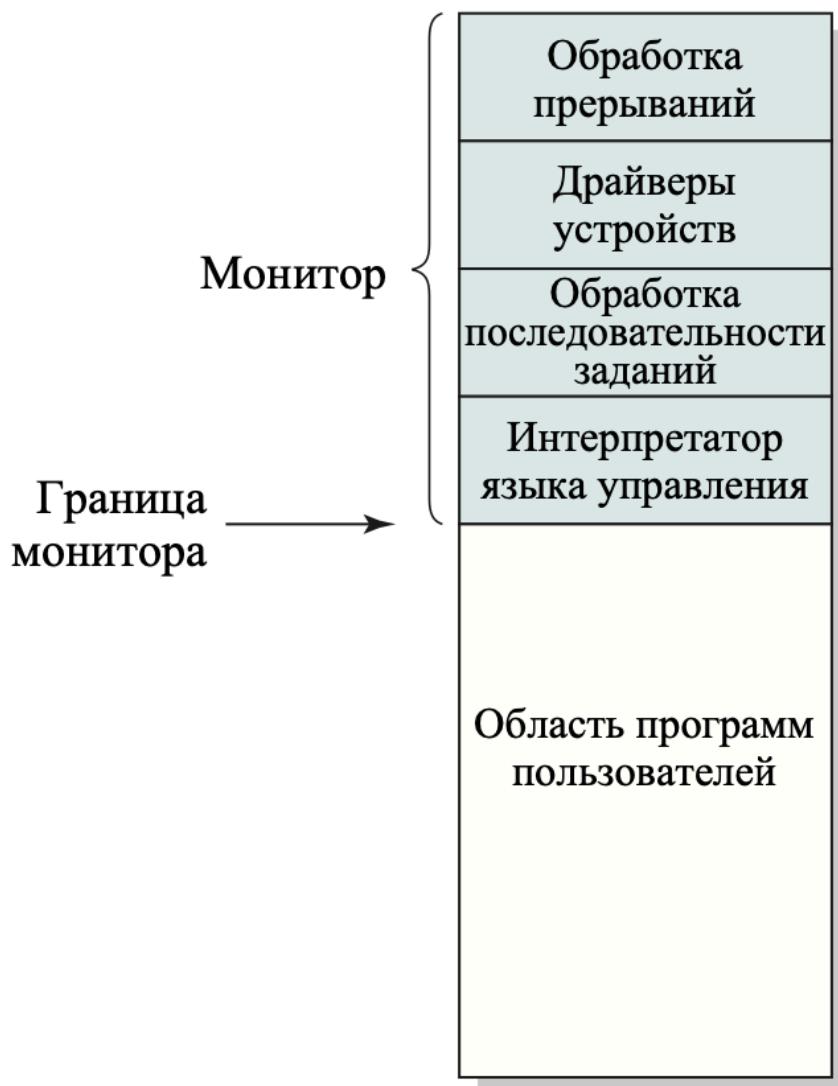
Такой подход назывался **последовательной обработкой**.

Позже для решения этих проблем и для замены рутинной работы оператора были придуманы пакетные системы и системные мониторы, которые ускоряли и упрощали работу с ВС.

## 5. Пакетная обработка. Системный монитор

---

Машины раньше были дорогие, поэтому необходимо использовать их как можно эффективнее. Простой и слишком долгая подготовка для запуска задания обходились слишком дорого. Поэтому придумали **пакетную обработку**.



Центральная идея, лежащая в основе простых пакетных систем, состоит в использование программы под названием **монитор**. Используя операционную систему такого типа, пользователь не имеет непосредственного доступа к машине. Вместо этого он передает свое задание на перфокартах или магнитной ленте оператору компьютера, который собирает разные задания в пакеты и помещает их в устройство ввода данных. Так они передаются монитору. Каждая программа составлена таким образом, что при завершении ее работы управление переходит к монитору, который автоматически загружает следующую программу.

#### **С точки зрения монитора работа происходит так:**

Монитор управляет последовательностью событий, обеспечивая выполнение заданий. Его резидентная часть всегда находится в памяти, а остальные функции загружаются при необходимости. Монитор считывает задания с устройства ввода, размещает их в памяти, передает управление программе пользователя, а после завершения возвращает управление и считывает следующее задание. Результаты направляются на устройство вывода.

### С точки зрения процессора

Процессор исполняет команды резидентного монитора. В другую область памяти считывается задание. После завершения чтения задания монитор отдает процессору команду перехода, по которой должно начаться исполнение задания. Исполнение будет идти до тех пор пока не закончится задания либо пока не возникнет ошибка и в любом из этих случаев следующей командой будет инструкция монитора.

Таким образом, системный монитор -- это просто обычная компьютерная программа чья работа основана на том, что процессор умеет выбирать команды из различных областей основной памяти, при этом происходит передача и возврат управления.

## 6. Анализ общесистемной эффективности, как предусловие многозадачности. Многозадачность, как способ повышения общесистемной эффективности. Системы разделения времени.

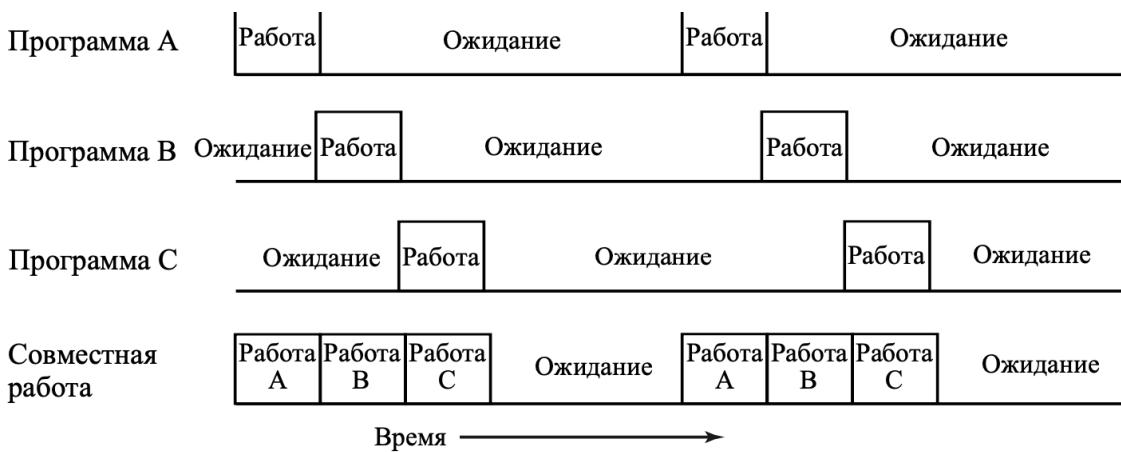
---

Процессору часто приходилось простоять даже при автоматическом выполнении заданий под управлением пакетной ОС. Проблема заключается в том, что устройства ввода-вывода очень медленные и работают гораздо медленнее, чем процессор. Пример программы, где для выполнения действий над одной записью из файла занимает 100 команд

Чтение одной записи из файла	$15 \mu s$
Выполнение 100 машинных команд	$1 \mu s$
Внесение одной записи в файл	$15 \mu s$
Всего	$31 \mu s$
Степень использования процессора	$= \frac{1}{31} = 0.032 = 3.2\%$

Таким образом, процессор будет простоявать 96% времени.

Однако, если предположить, что в памяти достаточно место для ОС и для нескольких программ пользователя, то теперь когда одно из заданий ждем окончание операции ввода-вывода другая программа может начать/продолжить исполняться таким образом мы уменьшаем процент простоя процессора. Более того, если в памяти достаточно места для нескольких программ, то они вообще могут исполняться параллельно переключаясь с одной задачи на другую. Такой режим известен как **многозадачность**



Многозадачные ОС намного сложнее однозадачных и требуют некоторой системы управления памятью, алгоритмов планирования и так далее.

### Системы разделения времени

Многозадачность значительно повышает эффективность работы ВС, однако хотелось так же обеспечить возможность взаимодействия пользователя(или даже нескольких) с компьютером. На сегодняшний день обеспечение интерактивных вычислений достигается путем использования отдельного ПК, однако в 60-е годы такой возможности не было.

Многозадачность позволяет процессору исполнять одновременно не только несколько заданий, но и также несколько интерактивных заданий. Такую организацию назвали **разделением времени**, потому что процессорное время разделяется между пользователями системы.

**Разделение времени (time sharing)** -- одновременное использование устройства несколькими пользователями.

В такой системе пользователи одновременно получают доступ к системе с помощью терминалов, а ОС чередует исполнение программ каждого пользователя через малые промежутки времени. Таким образом, если нужно обслужить n пользователей, то каждый получит  $1/n$  процессорного времени, не считая затраты ОС. С учетом, что реакция человека достаточно медленная, то с хорошо настроенной системой разделения времени время отклика компьютера будет сравнимо с реакцией пользователя.

**Квантование времени (time slicing)** -- режим работы, при котором двум или более процессам назначаются кванты времени на одном процессоре.

Именно с помощью квантования времени раньше реализовывались системы с разделением времени между пользователям. Через каждый квант времени генерировалось прерывание, при котором управление передавалось ОС и процессор мог перейти в распоряжение другому пользователю.

С приходом таких систем начали появляться проблемы такие как: защита одни программ от других, защита файловой системы пользователей друг от друга.

# 7. Процессы, проблемы современных процессов. Планирование выполнения процессов и управления ресурсов

Несколько определений **процесса**:

- выполняющаяся программа
- экземпляр программы, выполняющийся на компьютере
- объект, который можно идентифицировать и выполнять на процессоре
- единица активности, которую можно охарактеризовать единой цепочкой последовательных действий, текущим состоянием и связанным с ней набором системных ресурсов

## Структура процесса



- Исполняемая программа
- Набор потоков исполнения (внутри процесса мы можем создать набор потоков исполнения, что по сути является единицей потребления процессора )
- Связанные структуры ядра (при создании процесса внутри ОС ядро должно создать определенное количество структур, области памяти , которые содержат описания ресурсов необходимых для процесса )
- Адресное пространство (Код (read\_only), данные, стек, куча )
- Контекст исполнения или состояние процесса (вся необходимая информация, нужная ОС для управления процессом и процессору -- для его выполнения. Включает в себя различные регистры (счетчик команд, регистры данных), информация о приоритет процесса, о состоянии процесса и так далее). Контекст исполнения также используется при переключении контекста.
- Контекст безопасности (различные id)
- Ресурсы (файлы и пр.)
- Динамические библиотеки

## Проблемы современных процессов

- Защита памяти процессов
  - Недетерминированное поведение программы -- в условия совместного использования памяти и процессора программы могут влиять друг на друга, переписывая общие области памяти непредсказуемым образом. При этом результат работы программ может зависеть от порядка, в котором они были запущены
- Взаимные блокировки (deadlock, livelock, starvation)
  - deadlock -- тупиковая ситуация, возникающая, когда несколько процессов ожидают доступности ресурса, который не может стать доступным, потому что удерживается другим процессом, который находится в аналогичном состоянии ожидания
  - livelock -- состояние, при котором два или более процессов постоянно изменяют свое состояние в ответ на изменения в другом процессе (или процессах) без какой-либо полезной работы. Это похоже на взаимоблокировку тем, что при этом отсутствует какой-либо прогресс, но отличается тем, что ни один процесс не заблокирован и не ждет чего-либо
  - starvation -- Состояние, при котором процесс откладывается на неопределенное время, потому что предпочтение постоянно отдается другим процессам
- Проблемы синхронизации
  - Синхронизация потоков внутри процесса
  - Недостаточная надежность сигнального механизма (например, сигнал, что можно начать ввод-вывод)
- Взаимное исключение доступа к ресурсам
  - Один и тот же ресурс часто пытаются использовать несколько пользователей или программ одновременно, например, при редактировании файла. Без контроля это может привести к ошибкам. Для предотвращения требуется механизм взаимного исключения, разрешающий обновление файла только одной программе за раз. Проверить правильность такой реализации для всех сценариев крайне сложно.

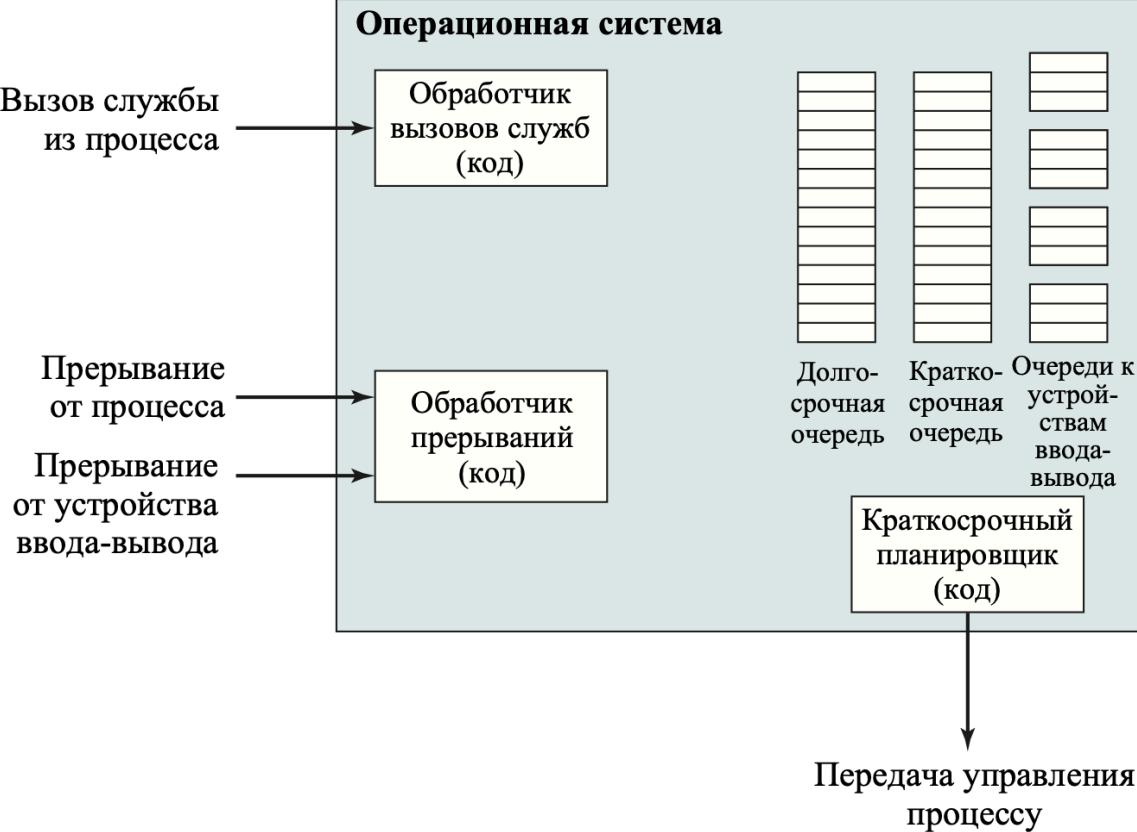
**Планирование выполнение процессов и управление ресурсами** (тут вообще кучу можно написать, но там дальше будут более точечные вопросы, поэтому тут наверное в общем надо написать)

При разработке стратегии распределения ресурсов необходимо принимать во внимание следующие факторы:

- равноправность (желательно, чтобы всем процессам, претендующим на какой-то ресурс предоставлялся равноправный доступ к нему )
- дифференциация отклика (Операционная система должна действовать динамически, в зависимости от обстоятельств)

- эффективность (Операционная система должна повышать пропускную способность системы, сводить к минимуму время ее отклика и, если она работает в системе разделения времени, обслуживать максимально возможное количество пользователей.)
- Планировщики процессов, дисков и прочее (Разные классы диспетчеризации (Time Sharing, interactives, real time, system, fair share (типа каждому пропорционально приоритету~количеству акций), fixed..))

[https://en.wikipedia.org/wiki/I/O\\_scheduling](https://en.wikipedia.org/wiki/I/O_scheduling)



Основные элементы ОС, участвующие в планировании процессов и управлении ресурсов в многозадачной среде.

- несколько списков процессов, которые ждут своей очереди на исполнение. **Краткосрочная очередь** -- процессы, находящиеся в основной памяти и готовые к исполнению. Выбор очередного процесса производится краткосрочным планировщиком или диспетчером. Общая стратегия заключается в том, чтобы каждому процессу в очереди давать к ней доступ. В **долгосрочной очереди** находится список новых процессов, ожидающих возможности использовать процессор. Операционная система добавляет их в систему, перенося из долгосрочной очереди в краткосрочную. В этот момент процессу необходимо выделить определенную часть основной памяти.

1. Долгосрочная очередь содержит новые процессы, ожидающие возможности использовать процессор.
2. Операционная система перемещает процессы из долгосрочной очереди в краткосрочную.

3. При перемещении процессу выделяется часть основной памяти.
4. ОС контролирует загрузку памяти и процессора, чтобы избежать перегрузки.
5. Для каждого устройства ввода-вывода создаётся своя очередь запросов.
6. ОС решает, какому процессу предоставить освободившееся устройство ввода-вывода.

## **8. Управление памятью, виртуальная память. Защита информации и безопасность ОС.**

---

### **Функции ОС, связанные с управлением памятью**

- Изоляция процессов
  - ОС должна следит за тем, чтобы ни один из независимых процессов не смог изменить содержимое памяти, отведенное другому процессу
- Управление выделением и освобождением памяти
  - Программы должны динамически размещаться в памяти в соответствии с требованиями. Распределением памяти должно быть прозрачно для программиста, таким образом программист будет избавлен от необходимости следить за ограничениями, связанные с конечностью памяти, а ОС будет повышать эффективность работы ВС.
- Поддержка модулей
  - Возможность определение модулей, динамические их создание, уничтожение и изменение
- Защита и контроль доступа
  - ОС должна следить каким образом различные пользователи могут осуществлять доступ к различным областям памяти. Одна программа может обратиться к пространству памяти другой программы, но только если это заложен в функциональность работы
- Долгосрочное хранение
  - Средства, позволяющие хранить информацию в течении долгого периода времени после выключения компьютера.
- Страницочный обмен
  - Paging, swaping

**Виртуальная память** -- это функциональная возможность, позволяющая программистам рассматривать память с логической точки зрения, не заботясь о наличии физической памяти достаточного объема. Место хранения, которое пользователь может рассматривать как адресуемую основную память, в которой виртуальные адреса преобразуются в реальные. Размер виртуальной памяти ограничен только схемой адресации ВС и объемом доступной вторичной памяти, а не фактическим количеством основной памяти.

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

Принципы работы виртуальной памяти были созданы для того, чтобы задания нескольких пользователей могли одновременно находиться в основной памяти и исполняться параллельно. К тому же из-за различий памяти, требуемой для разных процессов, стало сложность эффективно размещать их в памяти, поэтому была придумана страничная организация, в которой процесс разбивается на блоки фиксированного размера, которые и называются страницами.

Далее стало понятно, что не все страницы процесса могут находиться сразу в основной памяти. Достаточно лишь тех, который нужны в данный момент времени, а остальные могут лежать во вторичной памяти. При недостатке какой-то из страниц она может быть подгружена из вторичной памяти.

Программы обращаются к ячейкам памяти с использованием виртуальных адресов, транслируемых в ходе обращения в реальные адреса основной памяти. Если происходит обращение к виртуальному адресу, который не загружен в основную память, то один из блоков реальной памяти меняется местами с нужным блоком, который находится во вспомогательной памяти. Во время этого обмена процесс, который обратился к данному адресу, должен быть приостановлен. Задача разработки такого механизма преобразования адресов, который не требовал бы больших дополнительных ресурсов, и такой стратегии размещения данных в хранилище, которая сводила бы к минимуму перемещение данных между различными уровнями памяти, возлагается на разработчика операционной системы.

A.1			
	A.0	A.2	
	A.5		
B.0	B.1	B.2	B.3
		A.7	
	A.9		
		A.8	
	B.5	B.6	

Основная память

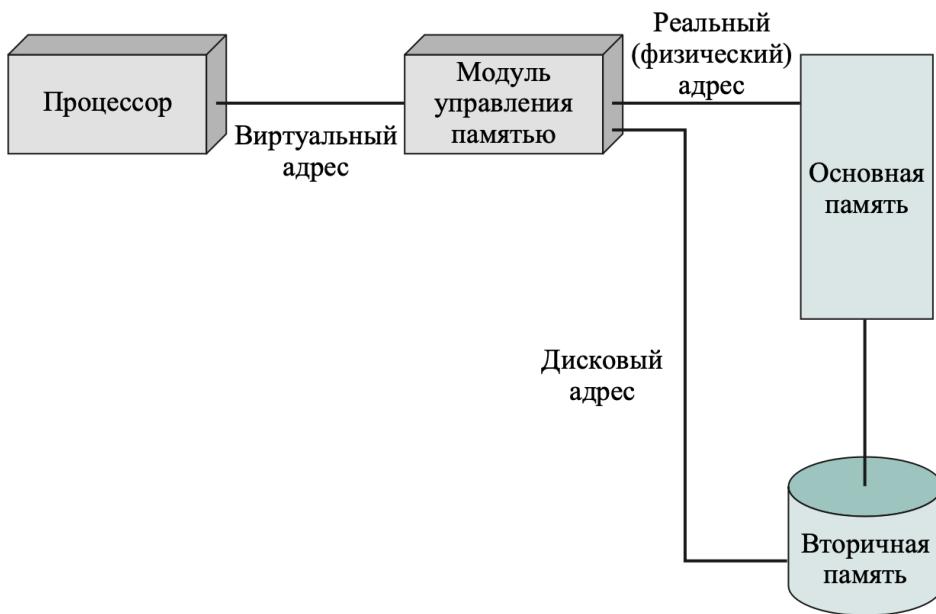
Основная память состоит из кадров (блоков, или фреймов) фиксированного размера, равного размеру страницы. Для работы программы некоторые (или все) ее страницы должны находиться в основной памяти



Диск

Вторичная память (диск) может хранить большое количество страниц фиксированного размера. Пользовательская программа состоит из некоторого количества страниц. Страницы всех программ и операционной системы хранятся на диске, как и файлы

### Схема адресации виртуальной памяти



- Виртуальная память управляет MMU (устройство, которое занимается переводом виртуальных адресов в физические) и TLB (кэш, чтобы ускорить трансляцию адресов)
- Также есть страницы, которые не могут быть выгружены из основной памяти

## Защита информации и безопасность ОС

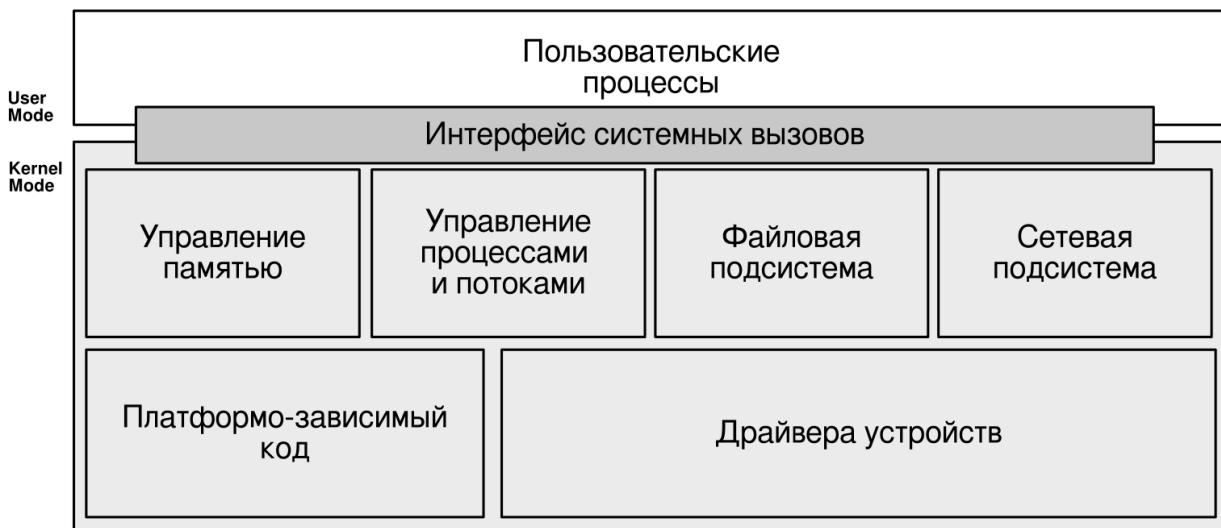
Большую часть задач по обеспечению безопасности и защиты информации можно условно разбить на 4 категории

- Доступ
  - Связан с защитой системы от постороннего вмешательства
- Конфиденциальность
  - Гарантирует невозможность чтения данных неавторизованным пользователям
- Целостность данных
  - Защита данных от неавторизованного изменения
- Аутентификация
  - Обеспечение надежной верификации пользователей и корректности сообщений или данных

## 9. Структура ядра ОС. Архитектура монолитного ядра, ядра с динамически загружаемыми модулями и микроядра

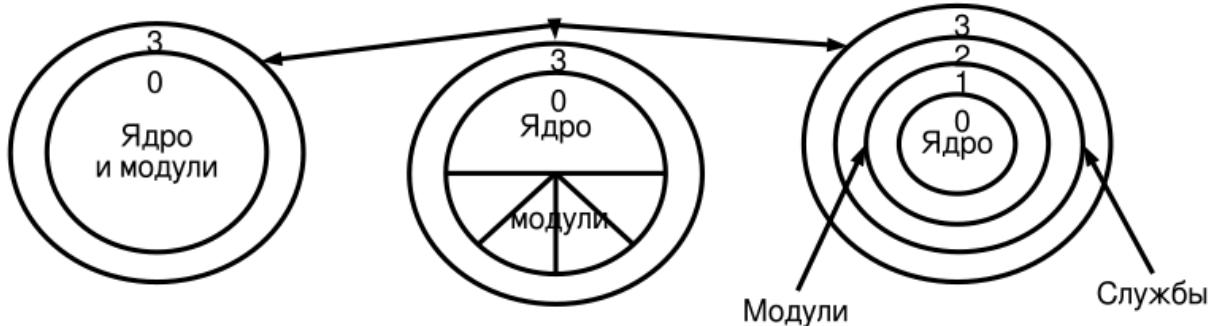
---

### Общая структура ядра



- Есть режим ядра (*kernel mode*) и режим пользователя (*User mode*)
- Между двумя режимами есть интерфейс системных вызовов, который является связующим звеном между ними и при помощи специальных соглашений данные передаются между пользователем и подсистемами ОС
- Пользовательские программы могут напрямую либо через библиотеки обращаться к сервисам ОС
- Все подсистемы работают на основании драйверов и платформо-зависимого кода (специальные написанные части ядра, которые зависят от архитектуры)  
В целом любая ОС состоит из таких подсистем как:

- Управление памятью
- Управление процессами и потоками
- Файловая подсистема
- Сетевая подсистема



### Монолитное ядро

Большое ядро, содержащее практически всю ОС, включая планирование, файловую систему, драйвера устройств управление памятью. Все функциональные компоненты ядра имеют доступ ко всем его внутренним структурам данным и процедурам. Как правило, монолитное ядро реализовано как единый процесс со всеми элементами, разделяющими одно и то же адресное пространство.

Является наследником мониторов. При изменении монолитного ядра приходится перекомпилировать все ядро и оно заново будет загружено в память и система начнет свою работу. Ядро и модули работают в своей области памяти и отгорожены от пользовательских программ. В настоящее время монолитное ядро в чистом виде не используется. В основном может использоваться внутри встроенных систем. В архитектуре Intel ядро работает на уровне 0, а пользовательский режим на уровне 3 (уровни 1, 2 не используются)

### Ядро с динамическими загружаемыми модулями

По архитектуре похоже на монолитное ядро, однако область памяти, где находится само ядро не является статической областью. В ней можно аллоцировать дополнительную память под код и новый модуль загрузить внутрь ядра.

При подключении нового устройства оно посылает свой ID в ОС. ОС пытается найти нужные драйвера и если находит то может подгрузить их в адресное пространство. Таким образом, мы подключаем новое устройство.

В монолитном ядре такого сделать нельзя (надо перекомпилировать все ядро)

### Микроядро

Небольшое привилегированное ядро операционной системы, обеспечивающее планирование процессов, управление памятью и службы связи и опирающееся на другие процессы для выполнения некоторых функций, традиционно связываемых с ядром операционной системы.

Пользовательские программы также работают на 3 уровне, однако уровни 1 и 2 используются в качестве сервисов (либо их называют серверов). Серверы можно

настраивать для требований конкретных приложений или среды. Фактически также выходит, что ядро может взаимодействовать с локальными и удаленными серверами по одной схеме.

На деле такая архитектура не используется так как он слишком медленная из-за частого переключения контекста.

## 10. Потоки исполнения, многопоточность, модели многопоточности

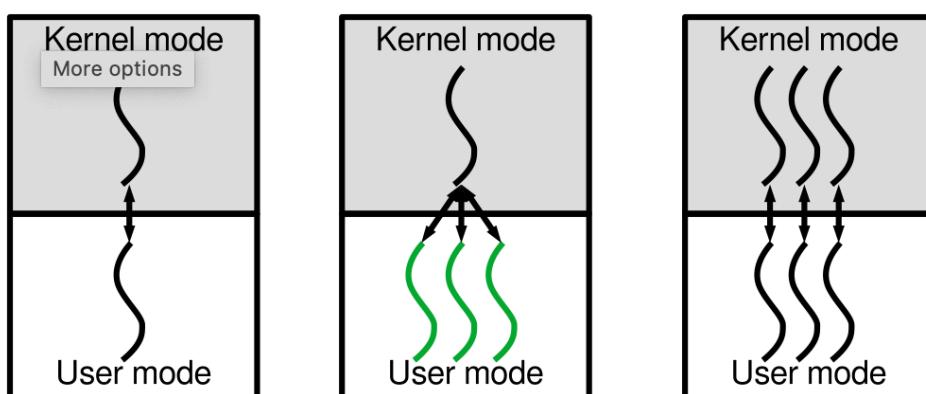
---

**Поток** -- диспетчерезуемая единица работы, включающая контекст процессора (в который входит счетчик команд и указатель стека) и собственную область памяти для стека (для возможности ветвлений). Поток исполняется параллельно и может быть прерван, так что процессор может заняться другим потоком. Процесс может состоять из нескольких потоков.

**Многопоточность** -- технология, при которой процесс, выполняющий приложение, разделяется на несколько одновременно выполняющихся потоков.

- Создание процессов очень дорогостоящая операция, поэтому были придуманы потоки исполнения.

### Модели многопоточности



- Один к одному
  - В каждом процессе находится не более одного потока исполнения
- Green threads
  - Легковесные потоки, которые полностью управляются пользовательской библиотекой или виртуальной машиной вместо ОС. Они эмулируют выполнение потоков на уровне пространства пользователя, а не пространства ядра.
  - Это было удобно для того, чтобы снизить переключение контекста. Мы получаем функциональность подобно обычным тредам, однако все равно нам доступно

- Многие ко многим
  - Несколько пользовательских потоков отображается на меньшее или равное количество ядер

**Posix Threads** -- это стандарт API для многопоточного программирования, определённый в спецификации POSIX (Portable Operating System Interface). Он предоставляет средства для создания и управления потоками на уровне ядра в Unix-подобных операционных системах, таких как Linux, macOS и другие.

## 11. Симметричная и ассиметричная многопроцессорная обработка

---

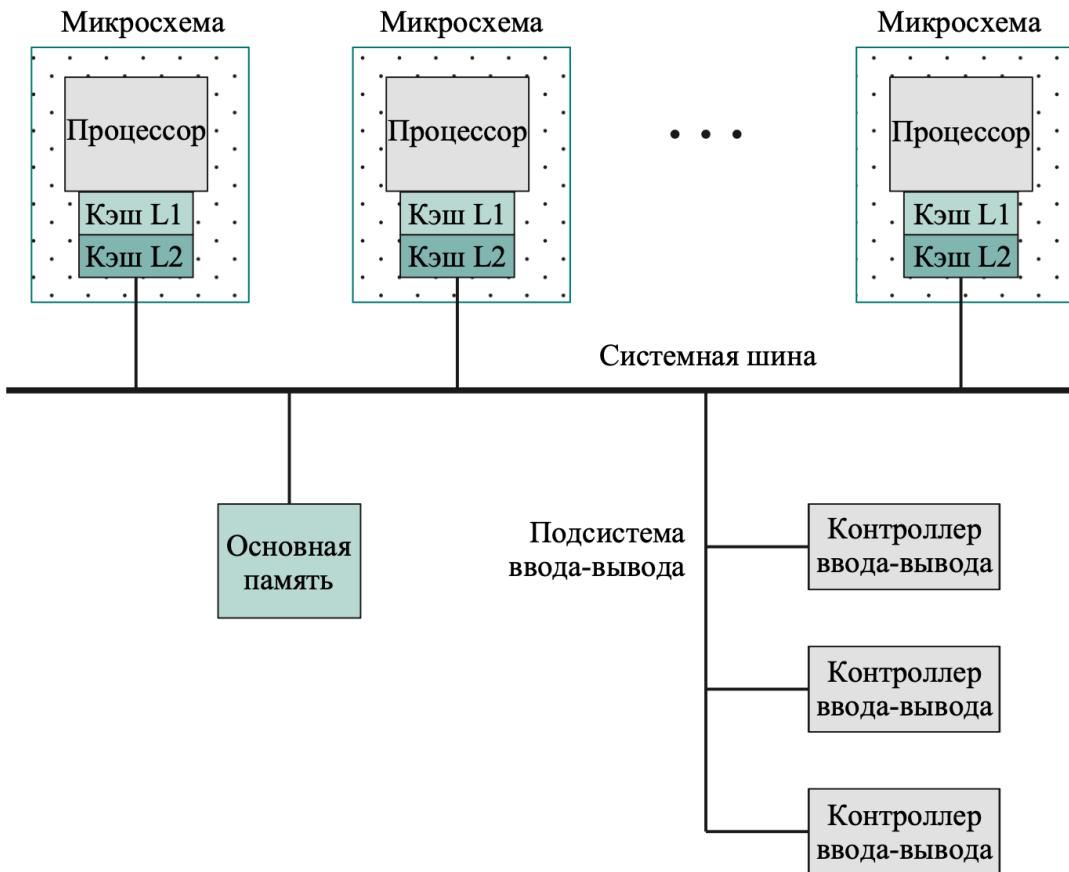
**SMP (symmetric multiprocessor)** можно определить как изолированную компьютерную системы обладающую следующими характеристиками:

- имеется не менее двух похожих процессоров со сравнимыми возможностями
- процессоры используют одну и ту же память и устройства вводы-вывода и соединены шиной или похожей схемой, так что время доступа к памяти оказывается примерно одинаковым
- все процессоры имеют общий доступ к устройствам ввода-вывода либо через одни и те же каналы, либо через различные каналы, которые обеспечивают одни и те же пути к устройству
- все процессоры могут исполнять одни и те же функции
- система управляется интегрированной операционной системой, которая обеспечивает взаимодействие между процессорами и их программами на уровне задач, заданий, файлов и элементов данных.

### Преимущества SMP

- производительность (возможность параллельного исполнения программ при наличии нескольких процессоров)
- надежность (отказ одного из процессоров не останавливает работу всей системы)
- инкрементный рост (повышение производительности путем добавление большего количества процессоров)
- масштабирование (замена и добавление и изменение характеристик системы)

### Общая организация SMP



У каждого процессора есть свой кэш. У каждого процессора есть доступ к основной памяти и устройствам ввода-вывожка через системную шину. Зачастую доступ к памяти организован так, что несколько процессоров могут одновременно получить доступ к памяти.

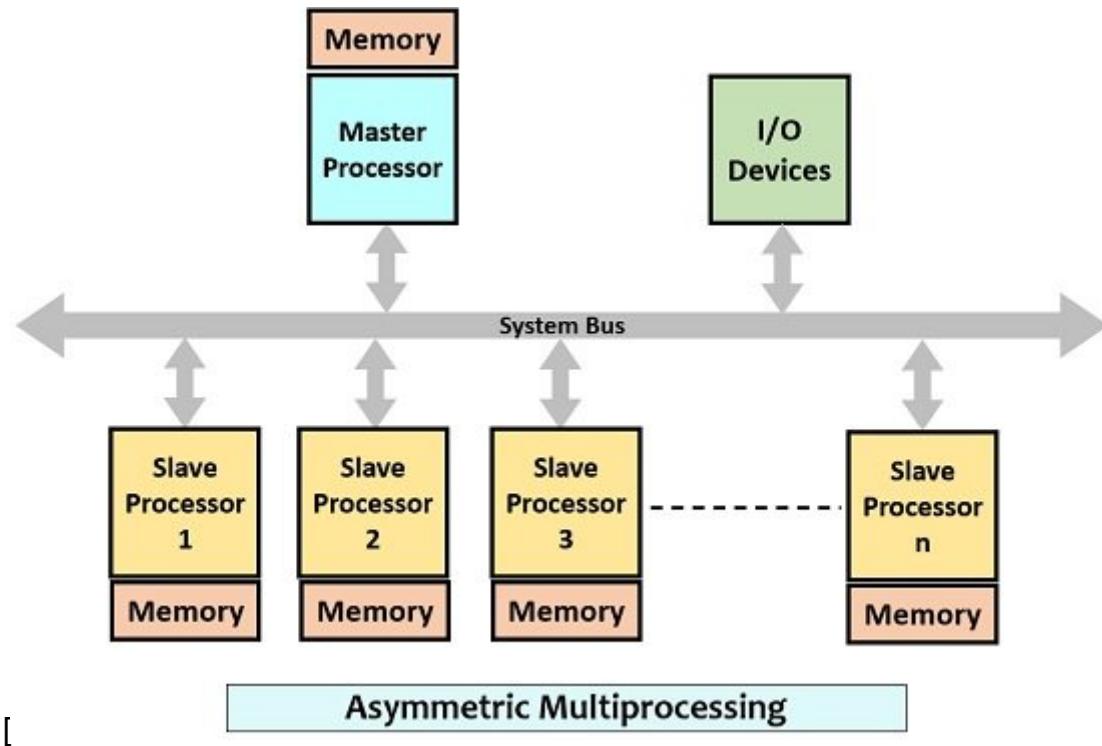
Появляется проблема согласованности кэшей

### **Ассиметрическая мультипроцессорность (ASMP)**

В системе с асимметрической многопроцессорностью не все процессоры играют одинаковую роль. Например, система может использовать (либо на аппаратном, либо на уровне операционной системы) только один процессор для выполнения кода операционной системы, или поручать только одному процессору выполнение операций ввода-вывода. В других ASMP-системах все процессоры могут выполнять код операционной системы и операции ввода-вывода, так что с этой стороны они ведут себя как симметричная многопроцессорная система, но определенная периферийная аппаратура может быть подсоединенена только к одному процессору, так что со стороны работы с этой аппаратурой система предстает асимметричной.

Также может быть Master-Slave отношения между процессорами, где один главный процессор назначает задачи подчиненным процессорам. (CPU-GPU)

### **Общая архитектура ASMP**



## 12. Виртуализация. Типы виртуализации

---

**Виртуализация** -- создание изолированной программной среды в рамках одного физического устройства.

ОС внутри которой работает другая ОС, называется **хост-системой**. А ОС, которая работает внутри называется **гостевой**.

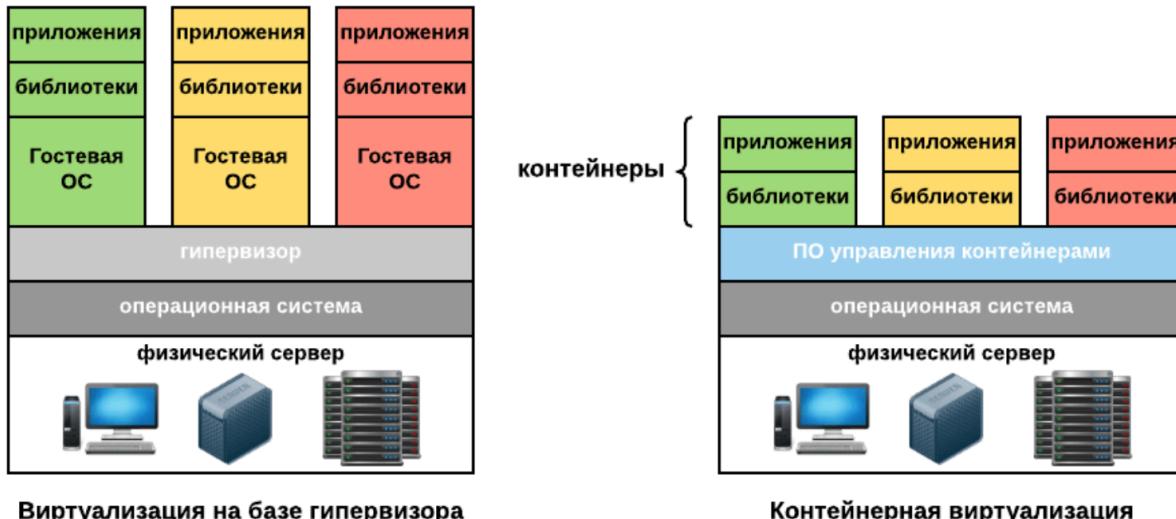
### Типы виртуализации

- **Виртуальная машина** -- экземпляр ОС (набор файлов )вместе с одним или несколькими приложениями, работающий в изолированном разделе компьютера. Позволяет одновременно запускать несколько ОС вместе, а также предотвращает взаимодействие приложений одно с другим. Такие ВМ называются **system VM**, потому что они предоставляют полную системную платформу и поддерживают выполнение полной ОС
- В другом случае можно сказать, что **виртуальная машина** -- интерпретатор, который находится под управлением другой программы и исполняет машинно-независимый код. К таким ВМ относится Java VM, Javascript в браузере либо Python. Такие ВМ называют **process VM (процессные ВМ)**, так как они создаются специально для запуска конкретного приложения и когда ВМ не нужна она не используется
- **Контейнеры приложений**
  - Контейнеризация -- метод виртуализации на уровне операционной системы, позволяющий запускать и управлять множеством изолированных

приложений (контейнеров) на одном хосте без необходимости виртуализировать каждую операционную систему.

- В отличие от традиционной виртуализации , где каждая виртуальная машина работает со своей собственной операционной системой, контейнеры делят одну и ту же операционную систему хоста, но остаются изолированными друг от друга. Это достигается благодаря использованию Namespace и Control group в Linux, которые обеспечивают изоляцию и управление ресурсами на уровне процессов.
  - Они легковесны, эффективно используют системные ресурсы и очень быстро запускать и останавливать
  - Самые популярные: Docker, Linux Containers, Solaris Containers
- **Аппаратная виртуализация** -- технология, предполагающая установку автономного гипервизора прямо на сервер без использования хостовой операционной системы (не всегда). Такая вычислительная среда отличается высокой производительностью и безопасностью. Используются гипервизоры 1 (bare-metal) и 2 (hosted) типов. VMWare, Virtual Box, Hyper-V
- **Облачные технологии**
    - Построены на базе аппаратной виртуализации
    - Возможен быстрый перенос данных с одной системы на другую в случае сбоя физической системы
    - Дополнительно включают provisioning и общий мониторинг

### Картинка для сравнения ВМ и контейнеризации



### Гипервизоры

Гипервизор -- это программное обеспечение, которое позволяет запускать и управлять виртуальными машинами (VM). Он создает, управляет и изолирует виртуальные машины, предоставляя им доступ к аппаратным ресурсам хостовой машины.

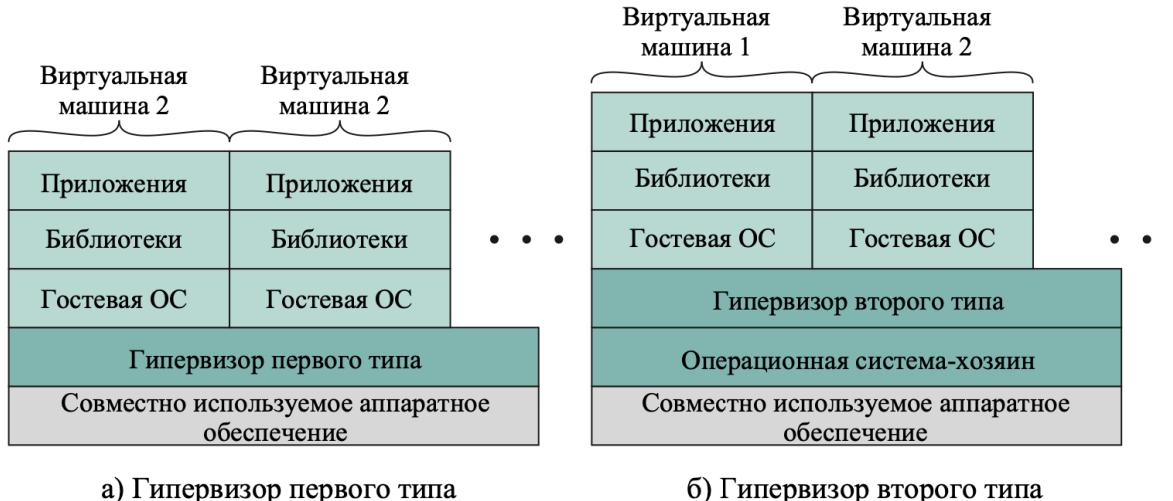
Гипервизор, действуя в качестве посредника для виртуальной машины, перехватывает и выполняет некоторые привилегированные команды, которые ее "родная"

операционная система будет выполнять на оборудовании своего узла. Это создает предпосылки к некоторому снижению производительности в процессе виртуализации, хотя со временем подобные издержки будут сведены к минимуму путем усовершенствования аппаратных и программных средств.

## Функции гипервизоров

- Управление выполнением ВМ
  - Включает планирование виртуальных машин, управление виртуальной памятью для изоляции, переключение контекста процессора, предотвращение конфликтов ресурсов и эмуляцию таймера и прерываний.
- Эмуляция устройств и управление ими
  - Эмуляция всех сетевых устройств и запоминающих устройств. Посредничество в доступе к физическим устройствам из разных ВМ
- Выполнение гипервизором привилегированных операций для гостевых виртуальных машин.
  - Гипервизор может от своего лица выполнять привилегированные операции
- Управление жизненным циклом ВМ
- Администрирование платформы и программного обеспечения гипервизора.

## Типы гипервизоров



- 1 тип (*bare-metal*)
  - Загружается как программный уровень непосредственно над физическим уровнем -- подобно тому как загружается ОС
  - Может непосредственно управлять физическими ресурсами узла
- 2 тип (*hosted*)
  - Используются ресурсы и функции ОС-хоста и действует он как программный модуль НАД ОС.
  - Как правило гипервизоры первого типа действуют лучше чем второго типа, так как первому типу не приходится соперничать за ресурсы с ОС хоста, что также ведет к большему количеству ресурсов, что дает нам возможность разместить больше ВМ

- Гипервизоры первого типа считаются также более безопасными так как первый тип делает запросы к ресурсам, которые обрабатываются вне данной ОС, и поэтому они не могут оказывать влияние на другие ВМ.
- Разработчики со вторым типом гипервизоров могут извлечь выгоду не только от ОС хоста, но и от виртуализированной ОС

## 13. Сбои и отказоустойчивость ОС. Причины появления отказов в ОС и способы борьбы с ними.

---

**Отказоустойчивость** -- способность системы или компонента продолжать нормальную работу, несмотря на наличие ошибок аппаратного или программного обеспечения.

**Отказоустойчивость предполагает:**

- **Наличие избыточности** аппаратуры (двойной или тройное резервирование). Обычно в коммерческих системах двойного резервирования бывает достаточно. К тому же для оптимизации работы рабочее и резервное устройство могут работать в параллель и при выходе одного из строя может наступить деградированный режим работы. Тройное резервирование применяется в военной и космической промышленности для обеспечения еще большей надежности. К тому же в таких системах применяется мажоритарный принцип резервирования, при котором каждое устройство отдает свой "голос" и используются данные, у которых больше всего голосов.
- **"Горячая замена" компонентов.** ОС должна поддерживать такую замену компонентов. Нужно для того, чтобы проводить замену компонентов без выключения системы.
- **Программная поддержка ОС** для горячей замены компонентов
- Организация **RAID** массивов

**Отказ** (и причины их появления) -- ошибочное состояние аппаратного или программного обеспечения в результате сбоя некоторого компонента, ошибки оператора, физической помехи от окружающей среды, ошибки проектирования, программирования или структуры данных

Отказы можно сгруппировать так

- **Постоянные** -- такой сбой после того как происходит будет присутствовать в системе до того момента пока его не устранит. Пример может служить поломка считывающей головки диска, ошибка ПО либо сгоревшая сетевая плата
- **Временные**

- Переходящие -- однократный сбой. Например, сбой при передаче в некотором бите из-за шума либо изменение бита памяти из-за внешнего излучения
- Периодические -- сбои, происходящие в разные, непредсказуемые моменты времени. Например, ошибки, вызванные неплотным соединением, приводящие к потери связи

## Методы резервирования

- **Пространственная избыточность** -- использование нескольких компонентном, которые одновременно выполняют одну и ту же функцию или настроены так, что один компонент доступен в качестве резервной копии
- **ВременнAя избыточность** -- повторение выполнение функции при обнаружении ошибки. Подход хорошо, что временных ошибок, но не подходит для постоянных. Пример -- ретрансляция блока данных
- **Информационная избыточность** -- репликация или копирование данных таким образом, чтобы ошибки в отдельных битах могли быть обнаружены и исправлены. Например -- схемы коррекции ошибок, которые используются в RAID массивах

## Методы повышения отказоустойчивости в ОС

- **Изоляция процессов.** Раньше изоляция процессов была сложной, так как они все работали в одном адресном пространстве. С появлением виртуальной появилась возможность изоляции процессов. Однако это не касается ядра ОС или драйверов, которые работают в одном адресном пространстве
- **Разрешение блокировок при параллелизме.** Полностью решить эту задачу нельзя, но ОС стараются не допускать появления различных дедлоков, лайвлоков и тд в работе.
- **Виртуализация.** Изоляция приложения или целой ОС в рамках хостовой ОС
- **Точки восстановления и откаты.** Могут создаваться какие-то резервные копии, которые сохраняются в системе и в случае появления какого-то сбоя система может откатиться на предыдущую сохраненную рабочую версию. Такой метод может использоваться как после временных так и постоянных сбоев.

## 14. Надежность. Среднее время восстановления.

### Коэффициент доступности и время простоя.

---

Три основные показателя качества функционирования системы:

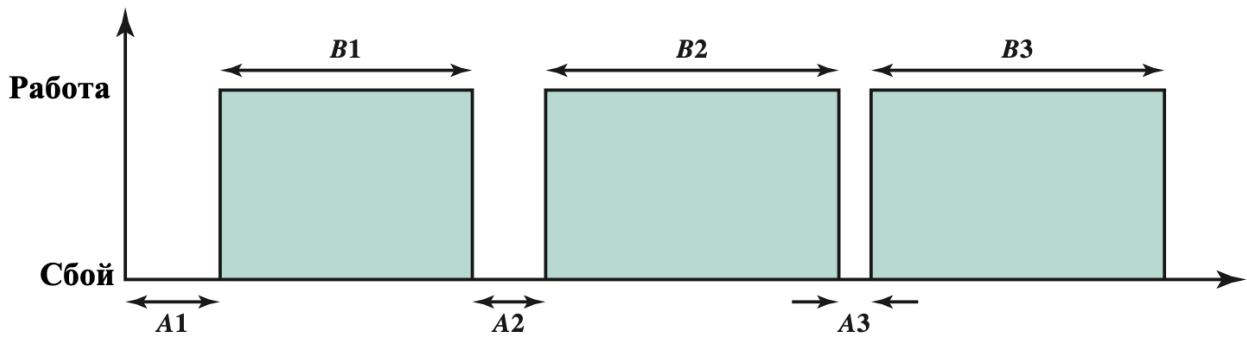
- надежность
- среднее время наработка на отказ (Mean Time to Failure)
- доступность

**Надежность =  $R(t)$**  -- вероятность бессбоиной работы до времени  $t$  при условии ее корректной работы в момент времени  $t = 0$ . Бессбоиная работа для ОС и компьютеров означает правильное выполнение набора программ и защиту данных от случайного изменения.

MTTF (среднее время наработка на отказ) определяется так:

$$MTTF = \int_0^x R(t)dt$$

**Среднее время восстановления (MTTR)** -- среднее время, необходимо для ремонта или замены неисправного элемента



$$MTTF = \frac{B1 + B2 + B3}{3} \quad MTTR = \frac{A1 + A2 + A3}{3}$$

**Доступность** системы определяется как доля времени, когда система доступна для обслуживания запросов пользователей. По сути является вероятностью того, что система при заданных условиях корректно функционирует в данный момент времени. Доступность может быть выражена такой формулой

$$\frac{MTTF}{MTTF + MTTR}$$

Время в течении которого система недоступна называется **простоем (downtime)**, относится к возможности обслуживания пользователей. Если ОС работает а веб-сервер нет, то это тоже считается простоем. Время в течении которого она доступна -- **временем безотказной работы (uptime)**.

## Классы доступности (условность)

Класс	Коэф. доступности	Время простоя в год
Непрерывная работа	1,0	0
Высокоотказоустойчивый	0,999999	32 секунды
Отказоустойчивый	0,99999	5 минут
Восстанавливаемый	0,9999	53 минуты
Высокодоступный	0,999	8,3 часа
Обычный	0,99-0,995	44-87 часов

## 15. Резервирование и отказоустойчивость

---

### Методы резервирования

- **Пространственная избыточность** -- использование нескольких компонентном, которые одновременно выполняют одну и ту же функцию или настроены так, что один компонент доступен в качестве резервной копии
- **ВременнAя избыточность** -- повторение выполнение функции при обнаружении ошибки. Подход хорошо, что временных ошибок, но не подходит для постоянных. Пример -- ретрансляция блока данных
- **Информационная избыточность** -- репликация или копирование данных таким образом, чтобы ошибки в отдельных битах могли быть обнаружены и исправлены. Например -- схемы коррекции ошибок, которые используются в RAID массивах

### Методы повышения отказоустойчивости в ОС

- **Изоляция процессов.** Раньше изоляция процессов была сложной, так как они все работали в одном адресном пространстве. С появлением виртуальной появилась возможность изоляции процессов. Однако это не касается ядра ОС или драйверов, которые работают в одном адресном пространстве
- **Разрешение блокировок при параллелизме.** Полностью решить эту задачу нельзя, но ОС стараются не допускать появления различных дедлоков, лайвлоков и тд в работе.
- **Виртуализация.** Изоляция приложения или целой ОС в рамках хостовой ОС
- **Точки восстановления и откаты.** Могут создаваться какие-то резервные копии, которые сохраняются в системе и в случае появления какого-то сбоя система может откатиться на предыдущую сохраненную рабочую версию. Такой метод может использоваться как после временных так и постоянных сбоев.

## 16. История развития ОС GNU/Linux. Single Unix Specification и POSIX.

- Зарождение первых ОС началось в **1961** году в MIT. Тогда появилась система CTSS (Compartiable Time-Sharing System) для IBM 709, а затем была перенесена на IBM 7094
- В **1967** году также от MIT была создана ITS (Incomaptiable timesharing system). Названа она была так в "шутку" и была предназначена для PDP-6 и в скором времени ее перенесли на PDP-10.
- В Манчестерском университете в **1962** году был создан один из самых мощных компьютеров на то время Atlas. Некоторые идеи заложенные в построение этой машины в каком-то виде используются и сейчас (супервизор Atlas и экстракоды Atlas )
- Прямой потомок ОС Unix -- Multics, которая была создана в **1965** году (MIT, GE, Bell Labs)
- В это же самое время IBM создает OS/360 в **1966**, которая до сих пор работает на мейнфреймах System/360. Мейнфраймы имеют специфичные ЯПы и очень дороги в обслуживании, однако они отказоустойчивые и надежные
- В 1968 голландцы создали ОС THE.

## UNIX

ОС была разработана компанией Bell Labs и запущена в эксплуатацию в 1970 году на PDP-7. Впоследствии появились различные версии этой ОС и главным событием стало перенос UNIX с PDP-7 на PDP-11, что послужило знаком, что ОС может использоваться массово. Версия 6 UNIX, появившаяся в 1976 году стала первой широко используемой за пределами Bell Labs. 7 версия (1978) стала прототипом большинства современных систем UNIX.

- Ключевые понятия: **вычислительный процесс и файл**. Ресурсы, которые ОС предоставляет, необходимо как-то представлять и лучшим способом можно назвать такую абстракцию как Файл, поэтому в UNIX все представлено в виде файлов и для программ было введено понятие вычислительного процесса
- Компонентная архитектура: принцип "одна программа - одна функция". Для программистов было удобно декомпозировать задачи, что привели к такой идеи. Сейчас у нас множество утилит, которые выполняют одну функцию и которые можно объединять в конвейеры или использовать в скриптах.
- Минимизация ядра. При разработке UNIX его создатели четко определили функциональность ядра для того чтобы предотвратить в дальнейшем усложнение ядра ОС. При этом все остальные функции были вынесены за пределы ядра, что привело к упрощенной поддержке ядра и разработка других функций стала проще из-за того, что она делалась на сторону пользователя.
- Независимость от аппаратной архитектуры и реализация на Си. ОС была переписана на СИ, что в то время было просто неслыханным (мало памяти как основной так и вторичной, компиляторы генерировали плохой код) так как считалось, что ОС -- сложная программа, для которой время работы было

- важным, поэтому она должна была быть написана на ассемблере. Однако это решение оказалось хорошим, так как это позволило сделать ОС отчасти переносимыми на разные архитектуры
- Унификация файлов

## Linux/GNU

Система Linux возникла как вариант UNIX, предназначенная для ПК с архитектурой IBM PC. Первая версия была написана Линусом Торвальдсом. В 1991 году первая версия была представлена в Интернете. С тех пор множество людей развивают Linux под общим руководством ее создателя. Именно благодаря тому, что эта система была бесплатной и ее код можно было беспрепятственно получить, она стала первой альтернативой для рабочих станций UNIX. **Сегодня Linux является полнофункциональной системой семейства UNIX (UNIX-like), способная работать почти на всех платформах.**

Залог успеха Linux в том, что эта система бесплатная, которая распространяется при поддержке фонда FSF (Free Software Foundation), целью которого создание надежного аппаратно-независимого ПО, которое было бы бесплатным и хорошего качества. Проект GNU этого фонда предоставляет инструменты для разработки ПО под общедоступной лицензией GPL (GNU Public License). Таким образом, система Linux в нынешнем виде является продуктом, который появился благодаря Линусу и затем многих других разработчиков в рамках проекта GNU (в 1992 Ричард Столлман присоединился к проекту Linux). (Linux по сути ядро, а GNU предоставляет множество утилит, которые делают Linux полноценной ОС)

## Single Unix Specification (SUS)

Развитие ОС нужно поддерживать в определенных рамках и если люди будут разрабатывать UNIX ОС, но в ней будут несовместимые бинарные либо прикладные интерфейсы, то UNIX ее назвать будет нельзя и более того они все будут несовместимы между собой, поэтому придумали **Single Unix Specification**, которая поддерживается The Open Group и Austin Group. Пройдя проверку на соответствие этой спецификации ОС можно назвать UNIX

### SUS включает в себя

- Основные определения (определения того, что может быть внутри ОС)
- Системные интерфейсы
- Командная оболочка и утилиты
- Пояснения
- X/Open Curses -- стандартизированная библиотека для разработки текстовых пользовательских интерфейсов в терминале. Библиотека предоставляет функционал для работы с текстовыми интерфейсами, абстрагируя особенности различных типов терминалов.
  - Предоставляет универсальный интерфейс для работы с терминалами, что делает приложения переносимыми

- Упрощает разработку программ с **многооконными интерфейсами**, обработкой ввода и выводом текста.

Сегодня UNIX можно назвать: AIX, HP-UX, Mac OS X, Solaris и тд

Также есть UNIX-like системы, которые не прошли сертификацию официально (они либо используют другое ядро либо другие интерфейсы), но очень похожи на UNIX: FreeBSD, OpenSolaris, NetBSD и другие

**POSIX** -- это набор стандартов для обеспечения совместимости программного обеспечения на уровне операционных систем. Эти стандарты определяют интерфейсы, которые операционная система должна предоставлять приложениям, чтобы они могли быть портируемыми между различными UNIX-подобными системами (и другими ОС).

POSIX охватывает несколько компонентов

- системные вызовы (read, write, fork, exec и тд)
- работу с потоками (posix threads, pthread)
- утилиты командной строки (grep, sed, awk)
- среда выполнения (стандартизация переменных окружения, работу с сигналами и управление процессами)
- файловая система

SUS включает в себя POSIX и чтобы ОС можно было назвать UNIX она должна соответствовать POSIX стандарту.

## 17. Понятие дистрибутива. Дистрибутивы Linux

---

**Дистрибутив** -- готовый к использованию набор ПО, включающий ядро ОС и дополнительные компоненты, которые формируют полноценную ОС

**Дистрибутив включает**

- **Ядро** (для Linux дистрибутивов всегда берется ядро от Линуса Торвальдса). Могут быть разных версий
- **Окружение**. Различные библиотеки, утилиты, который обеспечивают базовую функциональность ОС такие как работа с файлами, сетью, управление процессами и тд. Например, GNU
- **Менеджер пакетов и обновлений**. Программы для установки, обновления и удаления пакетов. Например: apt (Debian/Ubuntu), yum/dnf(Fedora), pacman(Arch Linux). Их много и они все похожи. Какой использовать -- личное предпочтение пользователя.

- **Графическая подсистема.** Работа с графическим интерфейсом (X11 или Wayland) и оконными менеджарми (GNOME, KDE, XFCE и др. )
- **Прикладные программы.** Набор приложений такие как: текстовые редакторы, браузеры, медиаплееры, офисные программы и тд
- **Поддержка.** То что можно купить и обратиться в случае наличия неисправности. Все проблемы поддержка решить не сможет, но может помочь понять известный ли этот баг или нет и могут перенаправить на нужное место в документации.

Дистрибутивы могут быть коммерческими или свободными.

Ubuntu/Canonical -- коммерческая, а Debian/Ubuntu -- свободная версия.

Red Hat Enterprise -- Fedora/Cent OS

SUSE -- Open SUSE

Также есть дистрибутивы, которые можно встретить в спец. решениях

- Oracle Enterprise -- взяли ядро Linux и переделали под себя
- Astra Linux -- используется в военной отрасли
- ArchLinux
- Gentoo

## 18. Архитектуры и основные подсистемы Linux. Linux Kernel map

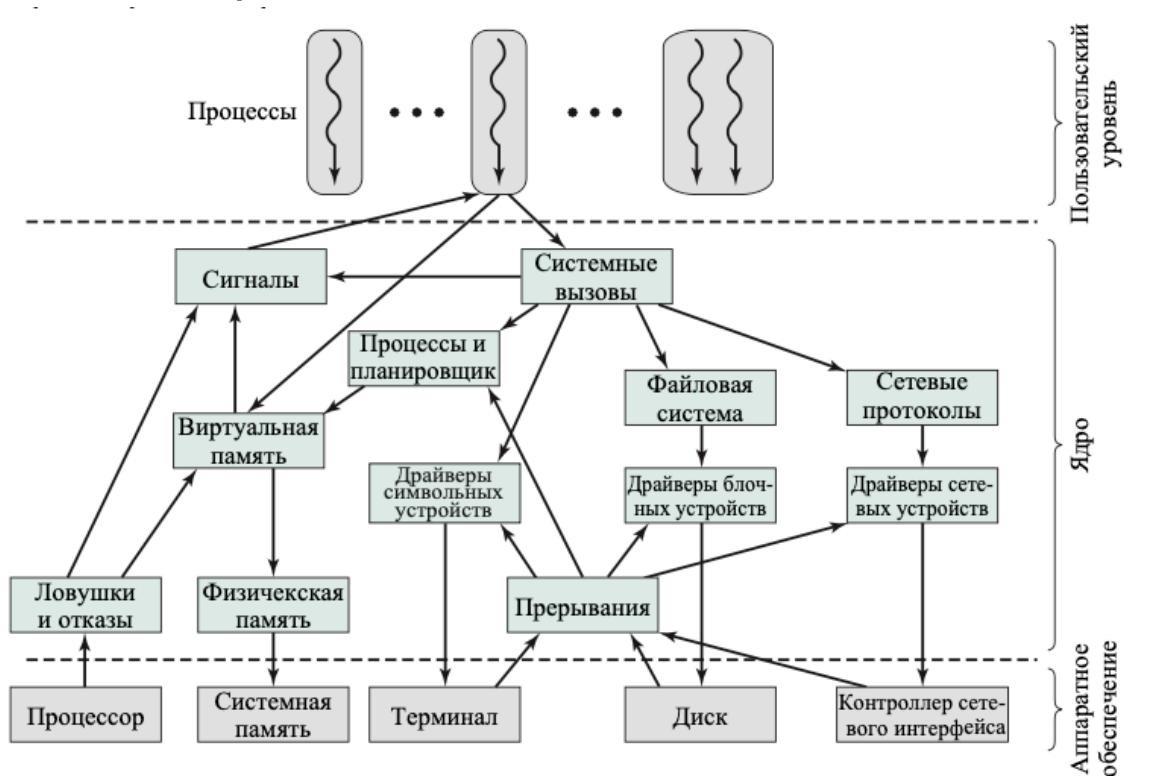
---

### Основные подсистемы Linux/Unix

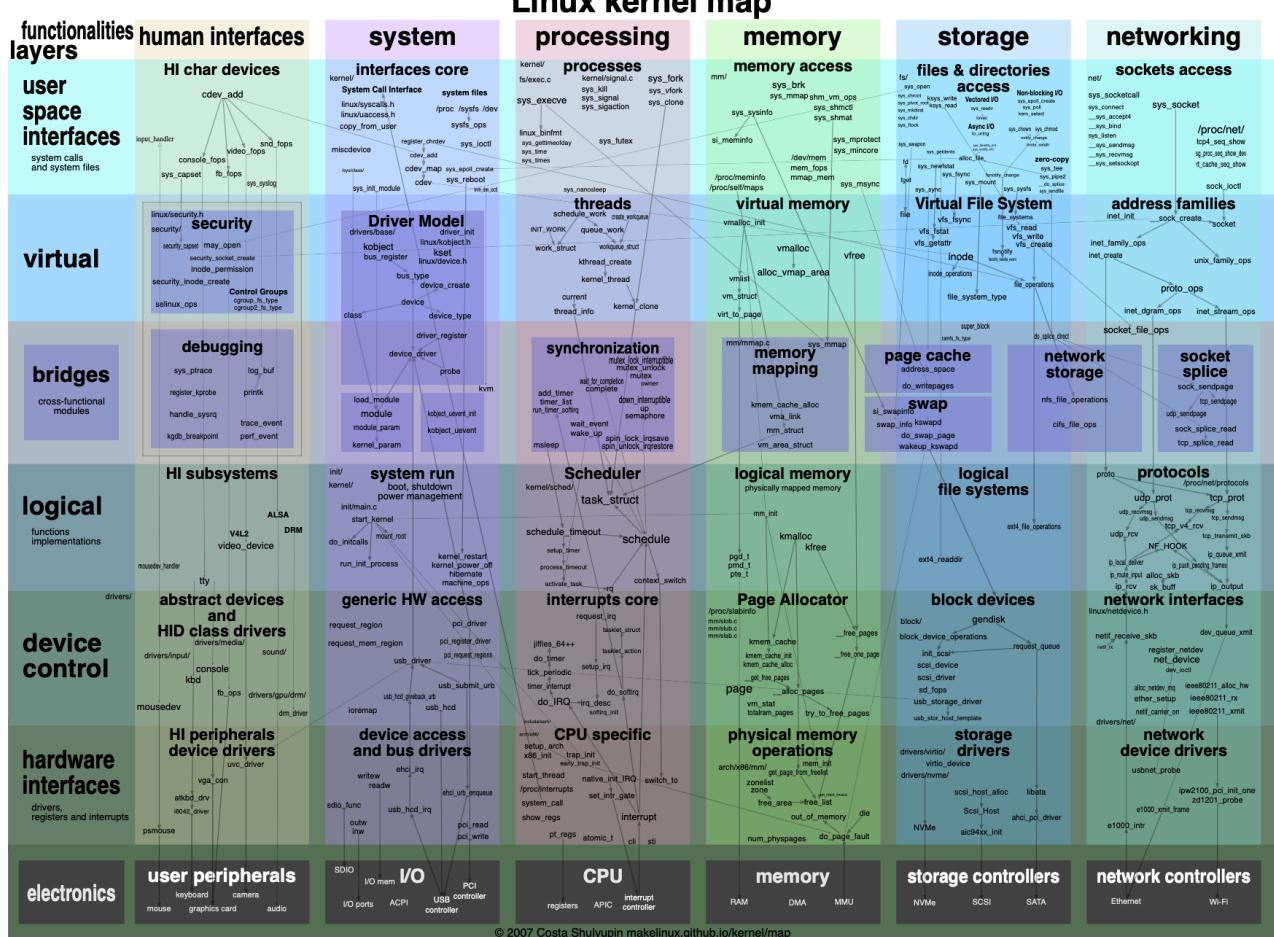
- **Процессы и планировщик** -- создание, управление и планирование процессов. Его задача создавать процессы, а потом ими управлять путем переключения добиваясь справедливого распределения ресурсов процессора между процессами
- **Виртуальная память** -- выделяет виртуальную память для процессов, создает странички, удаляет их, занимается свопингом и пейджингом
- **Физическая память** -- управляет пуллом кадров страниц и выделяет страницы для виртуальной памяти. Подсистема очень похожа на подсистему виртуальной памяти и они обе тесно связаны через MMU
- **Файловая система** -- предоставляет глобальное иерархическое пространство имен для файлов, каталогов и других объектов, связанных с файлами и функциями файловой системы. В Unix все является файлом и любая система быть то реальная или виртуальная пишется как драйвер для VFS.
- **Драйверы**
  - Символьных устройств -- управление устройствами, которые требуют от ядра отправки либо получения данных по одному байту, например -- терминал, принтер либо модем.

- Блочных устройств -- управление устройствами, которые читают и записывают данные блоками. К примеру различные виды вторичной памяти (диски, CD-ROM). Осуществляют буферизацию
- **Сетевые порты. TCP/IP.** Поддержка пользовательского интерфейса сокетов для набора протоколов. Сюда входит не только TCP/IP, но в целом набор каких-то протоколов. Занимает большую часть ядра
- **Драйверы сетевых устройств.** Управление картами сетевых интерфейсов и коммуникационными портами, которые подключаются к сетевым устройствам, такими как мосты или роутеры. С появлением виртуализации драйверы пришлось виртуализовать, что привело к их большому количеству
- **Ловушки и отказы.** Обработка генерируемых процессором прерываний, в том числе пуская систему в панику или подгружать доп страницы
- **Прерывания.** Обработка прерываний от периферийных устройств
- **Сигналы и IPC.** Межпроцессное взаимодействие. Наиболее частый сигнал -- ликвидация процесса. IPC включает в себя такие механизмы как: разделяемая память, средства синхронизации, обмен сообщениями, удаленные вызовы RPC, Семафоры

## Компоненты ядра Linux



# Linux Kernel Map



## **19. История развития Windows**

Впервые Microsoft использовала имя Windows в 1985 году для операционной среды, расширяющей возможности примитивной ОС MS-DOS, которая успешно использовалась на ранних ПК. Тогда была выпущена Windows 1.0. Имела 16-битную архитектуру и работала только в реальном режиме с ограничением на 640 Кб основной памяти.

В 1987 была выпущена Windows 2.0, которая все также оставалась надстройкой над MS-DOS (в ней улучшили графический интерфейс и apple подала на них в суд 😬 ).

Когда существовал DOS внутри процессоров не было многих средств, к которым мы привыкли. Например, не было виртуальной памяти и вся память представляла из себя единое адресное пространство, зачастую 640 Кб + сегментная адресация.

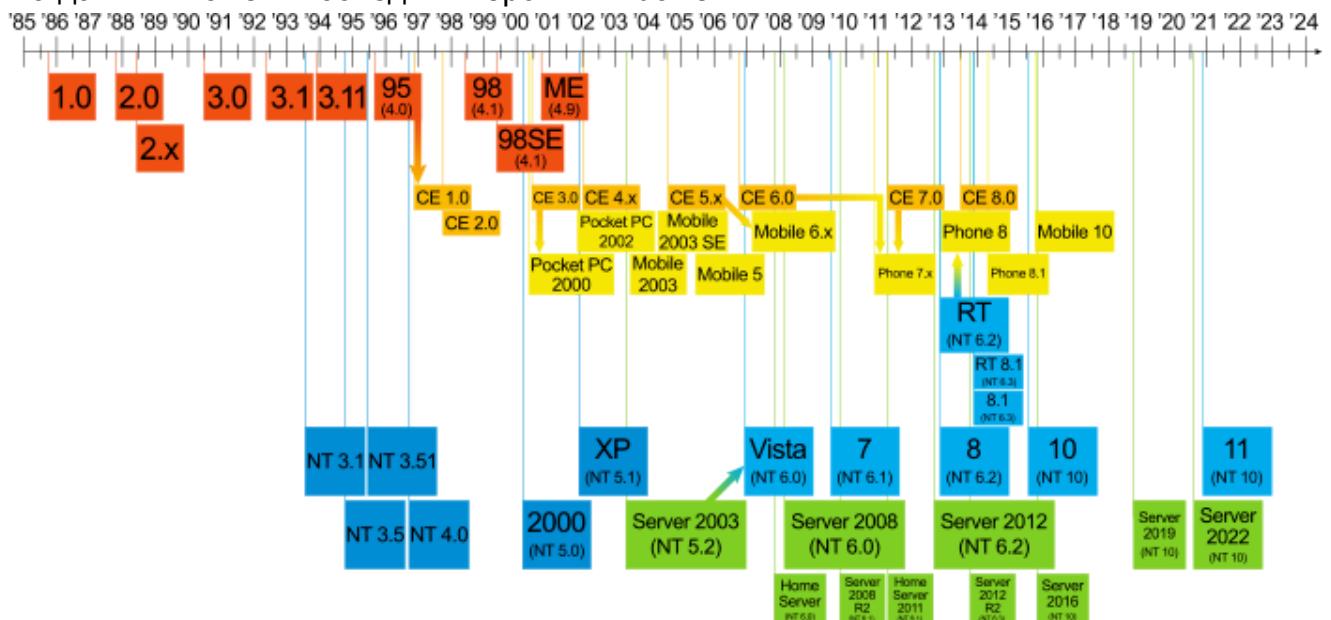
Более менее значимой версией стала Windows 3.x в 1990 году. В ней появилась поддержка виртуальной памяти и некоторые операции были переписаны с Си на ассемблер. Многозадачность была развита не так сильно, но эта версия стала зачатком той винды, которую мы знаем сегодня. За первые 6 месяцев было продано более двух миллионов копий

В 1995 году вышла Windows 95 (которая также была MS-DOS based)

- Встроенная поддержка многозадачности и драйверов plug-and-play
- 32-битная ОС, встроенная поддержка сетевых технологий и Интернета
- полноценная поддержка виртуальной памяти и драйверов от внешних устройств

Далее вышла Windows NT и чуть позже Windows 98, которые получали новые обновления и становились все стабильнее и стабильнее. Последняя версия, которая работала над MS-DOS стала Windows ME.

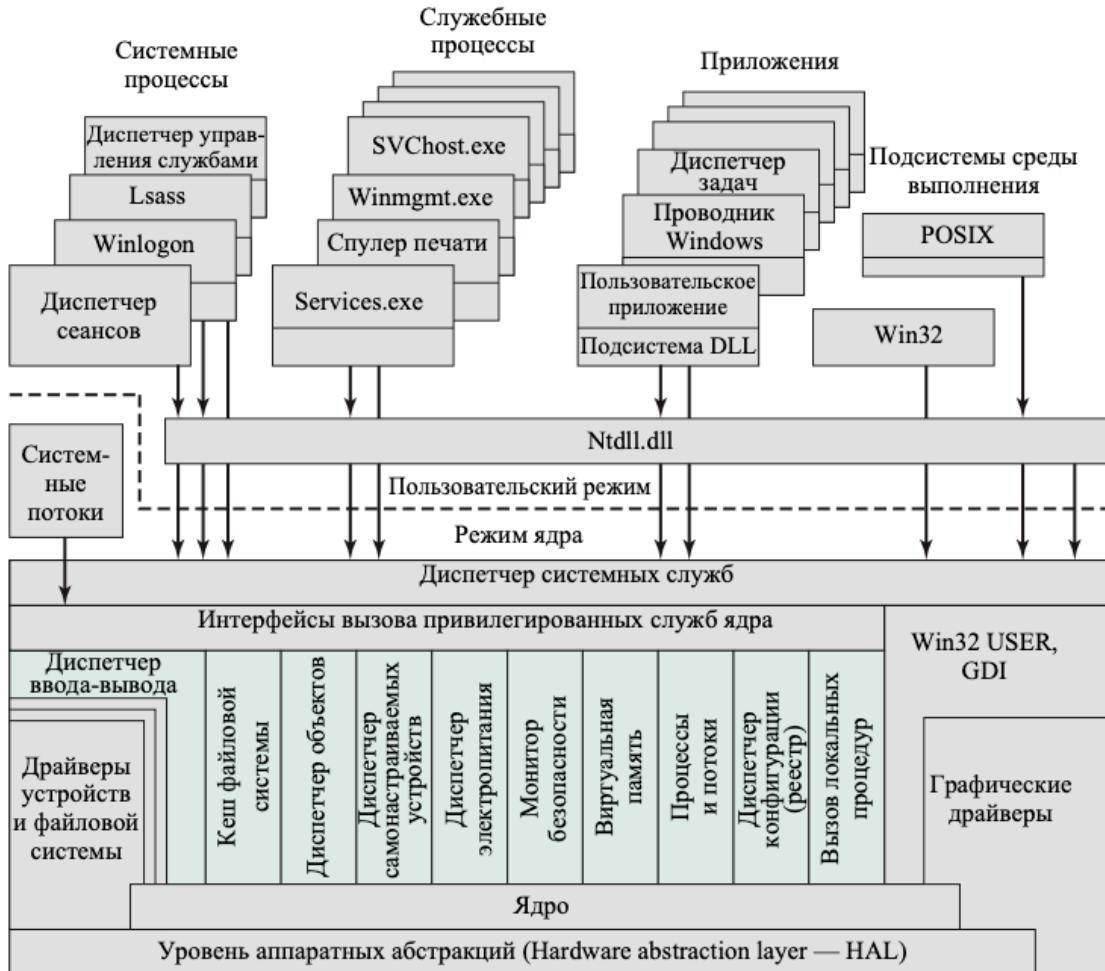
На данный момент последняя версия Windows 11



К тому же Windows делает несколько линеек ОС в том числе серверные, для мобильных устройств

## 20. Общая архитектура Windows. Windows API

С точки зрения архитектуры Windows похожа на Unix



Lsass — сервер проверки подлинности локальной системы безопасности

Выделенная область показывает исполнительную систему

POSIX — интерфейс переносимых операционных систем

GDI — интерфейс графического устройства

DLL — динамически подключаемая библиотека

Windows присущее четкое разделение на модули, где каждая функция системы управляется только одним компонентом (как в Unix). Остальные ее части и все приложения обращаются к этой функции через стандартные интерфейсы. Доступ к основным системным данным можно получить только через определенные функции и в принципе любой модуль заменить/удалить/обновить, не переписывая всю систему целиком.

## Пользовательский режим

- **Специальные процессы (системные процессы)** -- к таким процессам относятся служебные программы, которые не вошли в ОС Windows, например процесс входа в систему, система аутентификации и диспетчер сессий
- **Служебные процессы** -- очередь заданий печати, запись событий, пользовательские компоненты для взаимодействия с драйверами устройств, различные сетевые службы и тд. Службы используются как Microsoft так и сторонними разработчиками для расширения функциональности системы, так как это является единственным средством выполнения фоновой пользовательской активности

- **Подсистемы среды** -- это компоненты, которые предоставляют API (интерфейсы прикладного программирования) для запуска приложений, написанных для различных сред или платформ. Они обеспечивают взаимодействие между пользовательскими приложениями и ядром операционной системы. Основные компоненты это Win32 и POSIX

- **Пользовательские приложения** -- exe и динамические подключаемые библиотеки dll

### **Режим ядра**

Ntdll.dll -- системная библиотека системных вызовов.

Механизмы по работе с виртуальной памятью очень схожи с unix, что обусловлено аппаратурой.

- **Ядро** в котором происходит работа с процессами. Ядро управляет планированием потоков, переключений процессов, обработка исключений и прерываний, а также синхронизацией.
- **Уровень аппаратных абстракций (HAL)** -- выполняет отображение обобщенных команд и ответов аппаратуры на уникальные команды и ответы аппаратного обеспечения конкретной платформы. **HAL** **предназначен для скрытия различий в аппаратном обеспечении от основной части ядра операционной системы, таким образом, чтобы большая часть кода, работающая в режиме ядра, не нуждалась в изменении при её запуске на системах с различным аппаратным обеспечением.** Изолирует ОС от аппаратуры и позволяет единым образом обращаться к аппаратуре. HAL делает системную шину, контроллер ПДП, контроллер прерываний, системные таймеры и контроллер памяти разных компьютеров одинаковыми для ОС
- **Драйвера устройств** -- динамические библиотеки, расширяющие функциональность системы. Драйвера устройств аппаратуры, которые транслируют пользовательские вызовы функций ввода-вывода к конкретным устройствам, и программные компоненты реализации файловой системы, сетевых протоколов и других системных расширений, которые выполняются в режиме ядра.
- **Графические драйвера** -- работа с окнами, управление интерфейсом пользователя и вывод на экран. (стоит отметить, что в Unix вся часть работы с графикой вынесена в пользовательский режим, а в Windows исторически это теперь является часть ядра)

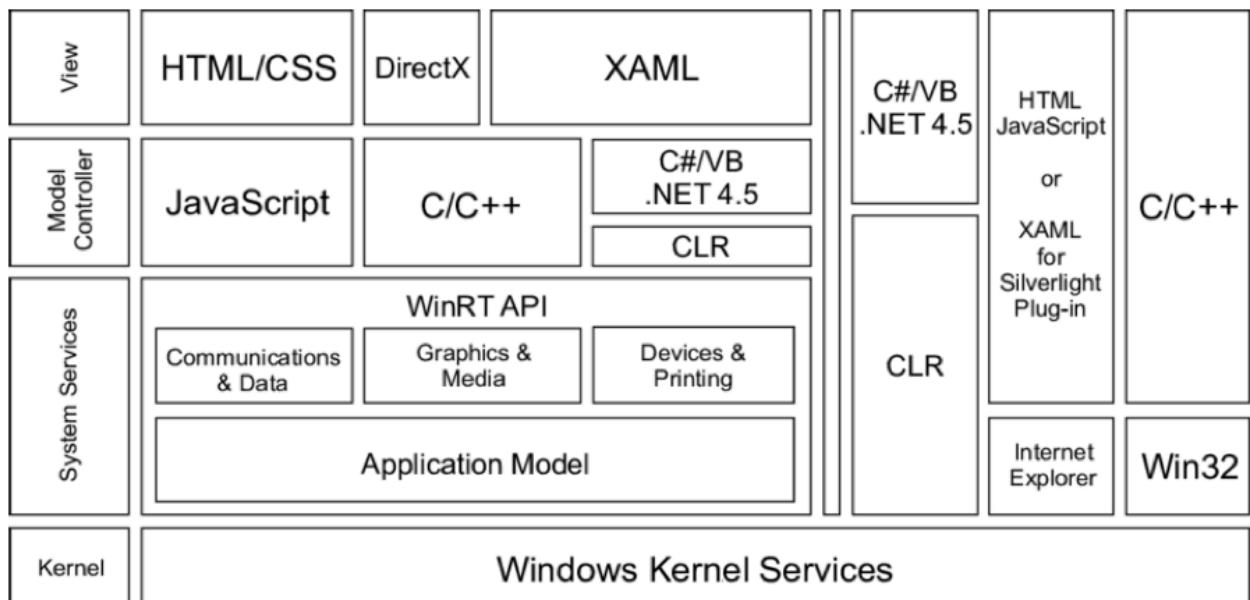
### **Исполнительная система внутри ядра (основные модули)**

- **Диспетчер ввода-вывода** -- каркас с помощью которого устройства ввода-вывода доступны для приложений. Отвечает за координацию работы драйверов устройств, выполняющих дальнейшую обработку. Реализует все API ввода-вывода Windows и следит за безопасностью и именованием устройства, сетевых протоколов и файловых систем.
- **Диспетчер кеша** -- повышает производительность храня файлового ввода-вывода путем хранения данных с диска, которые были недавно использованы.

## Отложенная запись на диск

- **Диспетчер объектов** -- создаёт, удаляет и управляет объектами, такими как процессы, потоки и объекты синхронизации. Он поддерживает правила работы с объектами, их именование, безопасность и создаёт дескрипторы с информацией о правах доступа и ссылками на объекты.
- **Plug and play** -- определяет нужные драйверы для устройства и загружает их
- **Диспетчер электропитания**
- **Монитор безопасности** -- Windows использует объектно-ориентированную модель для согласованного управления безопасностью. Авторизация доступа и аудит применяются ко всем защищённым объектам (файлы, процессы, устройства и т.д.) с помощью единых служебных программ.
- **Диспетчер виртуальной памяти**
- **Диспетчер потоков и процессов**
- **Диспетчер конфигурации** -- реализация и управления системным реестром, который является единым хранилищем настроек и параметров как для всей системы, так и для конкретного пользователя
- **Расширенный вызов локальных процедур** -- ALPC обеспечивает эффективный механизм межпроцессного взаимодействия для обмена информацией между локальными процессами. Похож на RPC

## Windows API (WinAPI, Win32API, WinRT)



У Windows есть общий подход к прикладным программным интерфейсам при помощи которого можно разрабатывать программы -- **WinRT**.

В WinRT есть представление и модели коннекторов, которые можно писать на JavaScript, C/C++ and C#. Также есть системные сервисы по работе с передачей данных, работы с графикой и устройствами ввода-вывода. К тому же есть поддержка обратной совместимости для Win32API (там также есть эти языки JavaScript, C/C++ and C# )

## 21. Сервисы, функции и важные компоненты Windows

## **Сервисы и функции**

Windows API не похоже на то, что есть в Unix, поэтому если нужно делать переносимые программы, то стоит писать в стиле POSIX (что также можно делать на винде).

### **Сервисы и функции:**

- **Windows API functions**

Управление процессами и потоками:

- CreateProcess — создание нового процесса.
- TerminateProcess — завершение процесса.

Работа с памятью:

- VirtualAlloc — выделение виртуальной памяти.
- VirtualFree — освобождение памяти.

Работа с файлами и каталогами:

- CreateFile — открытие или создание файла.
- ReadFile, WriteFile — чтение и запись данных в файл.
- DeleteFile — удаление файла.

Управление устройствами:

- DeviceIoControl — управление устройствами через драйверы.

- **System calls (Native system services)**

- NTCREATEUSERPROCESS -- прямой вызов ядра, тот самый системный вызов, который проходит через границу ядра

- **Kernel support functions**

- это набор низкоуровневых функций, используемых для управления ресурсами, синхронизации, обработки ввода/вывода и выполнения других критичных задач на уровне ядра операционной системы. Эти функции являются важным элементом для стабильной и безопасной работы ОС, а также предоставляют интерфейсы для взаимодействия с аппаратным обеспечением и выполнения других системных операций. В основном используются для разработки драйверов.
- ExAllocatePool2

- **Windows Services**

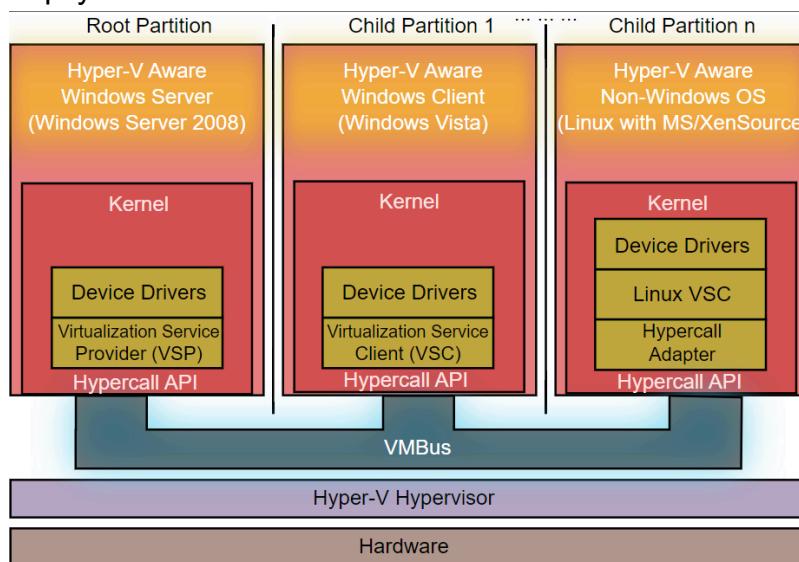
- Тип приложения, предназначенный для работы в фоновом режиме в операционной системе Windows. Службы запускаются и работают без взаимодействия с пользователем, обычно при старте системы или при запросе со стороны других программ. Они обеспечивают различные функциональные возможности, такие как поддержка сетевых сервисов, безопасность, мониторинг, выполнение периодических задач и другие.
- Управляются **Service Control Manager** -- это компонент операционной системы Windows, который отвечает за управление жизненным циклом всех служб (Windows Services) на компьютере. Он контролирует запуск, остановку,

настройку и взаимодействие служб с операционной системой и другими приложениями. SCM играет ключевую роль в организации работы службы, их мониторинге и управлении.

- **Dynamic Linked Libraries (DLL)** -- это файл, содержащий код и данные, которые могут быть использованы несколькими программами одновременно. DLL-файлы позволяют программам разделять функциональность, избегая дублирования кода и улучшая использование системных ресурсов. Они являются важной частью архитектуры операционных систем, таких как Windows. Как shared-object в linux

## Важные компоненты Windows

- **Гипервизор Hyper-V**. Позволяет запускать гостевые ОС нативно в винде (он платный). Зачастую люди просто используют VirtualBox. Гипервизор 1 типа Основан на разделении системы на два уровня -- **родительский раздел**, который управляет гипервизором и взаимодействует с оборудованием и содержит хостовую ОС и **дочерний раздел** -- сами виртуальные машины, которые запускаются на Hyper-V, где каждая ВМ имеет доступ только к тем ресурсам, которые предоставил родительский раздел.  
Основные компоненты архитектуры
  - Сам гипервизор
  - VMBus -- канал связи между ВМ
  - Virtual Device -- Эмуляция устройств для виртуальных машин (дисков, сетевых карт и т.д.).
  - Integration Service -- Специальные драйверы для улучшения производительности виртуальных машин.



- **Firmware** -- то программный код, который встроен в аппаратное устройство (например, материнскую плату, жёсткий диск, видеокарту, или периферийные устройства). Firmware служит для управления базовыми функциями оборудования, предоставления интерфейса между железом и операционной системой, а также обеспечения возможности его настройки и обновления и в целом загрузки кода при старте системы.

- **Terminal Servers**

- так как винда в целом подразумевалась как система для одного пользователя. Поэтому для обеспечения работы с серверами потребовалось написать специальные терминальные сервисы.
- это серверная технология, предоставляющая пользователям удалённый доступ к рабочему столу и приложениям, работающим на сервере. Она является частью службы **Remote Desktop Services (RDS)**, ранее известной как **Terminal Services**.
- Позволяет пользователям подключаться к серверу и получать изолированную сессию, что выглядит как отдельный рабочий стол.
- Используется Remote Desktop Protocol (тонкие клиенты), который работает поверх TCP/IP. Соответственно есть сильная зависимость от сети

- **Объекты и безопасность**

- **Объект** -- это абстракции, используемые операционной системой для управления ресурсами, такими как файлы, процессы, потоки, устройства и многое другое. Они реализуют общую модель управления доступом, позволяя приложениям взаимодействовать с этими ресурсами через унифицированный интерфейс.
- Хоть и ядро винды написано на Си, но его структура в значительной мере следует концепциям ООП. Этот подход позволяет совместно использовать ресурсы и данные различными процессами, а также защищать ресурсы от несанкционированного доступа.
- **Common Object Model** -- это независимая от платформы распределенная объектно-ориентированная система для создания двоичных компонентов программного обеспечения, которые могут взаимодействовать. СОМ это не ООП язык, а стандарт, который задает объектную модель и требования к программированию, которые позволяют СОМ-объекта взаимодействовать друг с другом. СОМ определяет основную природу СОМ-объекта. Как правило, программный объект состоит из набора данных и функций, которые управляют данными. СОМ-объект — это объект, в котором доступ к данным объекта достигается исключительно с помощью одного или нескольких наборов связанных функций. Эти наборы функций называются интерфейсами, а функции интерфейса называются методами. Кроме того, СОМ требует, чтобы единственный способ получить доступ к методам интерфейса — это указатель на интерфейс.
- В СОМ зачастую используется модель клиент-сервер и такое взаимодействие организовано так, что клиент и сервер могут находиться как в пределах одного процесса, компьютера или на разные машинах
- **Registry** -- Реестр Windows можно описать как центральное хранилище информации, где операционная система сохраняет многие настройки, параметры и конфигурации, необходимые для работы компьютера. В простых словах, это своего рода база данных, где содержатся все важные записи, относящиеся к

установленным программам, компонентам операционной системы, пользовательским настройкам и многому другому. Представлен в виде деревовидной структуры с ключ-значениями. Реестр может использоваться для устранения неполадок в системе, настройки системы или даже улучшения производительности системы.

- **Оснастки** -- представляют из себя программы/модули или инструменты, которые интегрируются в **Microsoft Management Console (MMC)**. Они предоставляют административный интерфейс для управления различными аспектами системы, такими как учетные записи пользователей, политики безопасности, службы, оборудование и сетевые параметры.
  - compmgmt.msc -- универсальная оснастка для управления дисками, просмотр журнала событий, управление локальными пользователями и группами, управления службами
  - gredit.msc -- настройка параметров системы таких как: автозапуск приложений, ограничения доступа к функциям, сетевые политики Некоторые оснастки могут использоваться утилитами для мониторинга и профилирования системы.

## 22. Процесс. Характеристики процесса в момент выполнения. Состояние процесса. Разделение ресурсов.

---

### Обобщение вычислений в ОС

- **Компьютерная система состоит из набора ресурсов.**
  - Ресурсы -- то, что потребляется для выполнение полезной работы
  - Процессоры, память, устройства ввода-вывода, таймеры и тд все представлено как ресурсы
- **Приложения решают практическую задачу**
  - Выполнение задачи процессором в очень обобщенном виде можно представить в виде такой схемы: Входные данные -> Обработка -> Выходные данные
- **ОС находится между приложением и оборудование**
  - **ОС обеспечивает функционально богатый, безопасный и единообразный интерфейс.** Пользователь не знает как физически заставить компьютер выполнять задачу, поэтому здесь помогает ОС, которая предоставляет приложениям набор интерфейсов. Таким образом, пользователь может получить необходимые ему ресурсы
  - **ОС предоставляет абстрактный ресурс и дает этот ресурс программам пользователя.** Эти ресурсы включают в себя основную память, сетевые интерфейсы, файловые системы и так далее. К тому же после того как ОС создала абстрактные ресурсы она также должна ими управлять.

## Процесс

- выполняемая программа
- экземпляр программы, выполняющейся на компьютере
- сущность, которая может быть назначена процессору и выполнена на нем
- единица активности, характеризуемая выполнением последовательности команд, текущим состоянием и связанным с ней множеством системных ресурсов.

## Характеристика процесса во время выполнения

- **Идентификатор.** Уникальный id, чтобы отличить процесс от других. Основным уникальным id можно считать PID. В старых ОС id раньше являлось именем процесса, так как невозможно было запустить несколько процессов одновременно.
- **Состояние.** Если процесс выполняется в настоящее время, то он находится в состоянии выполнения
- **Приоритет.** Представляет собой число, которое имеет значение только в момент сравнения приоритета данного процесса с приоритетом другого процесса. При этом если приоритет процесса А больше приоритета процесса В, то первый процесс должен выполниться раньше второго.
- **Программный счетчик.** Адрес очередной выполняемой команды программы
- **Указатели на области памяти.** Указатели на программный код и данные, связанные с этим процессом, а также любые блоки памяти, совместно используемые с другими процессами.
- **Контекст процесса.** Данные, которые находятся в регистрах процессора во время выполнения процесса. Также контекст процесса может означать исполнение процесса на уровне ядра или пользователя.
- **Статус ввода-вывода.** Внешние запросы ввода-вывода, устройства ввода-вывода, назначенные процессу, список файлов, используемых процессом и тд
- **Учетная информация.** Счетчики системных ресурсов, права доступа процесса и тд

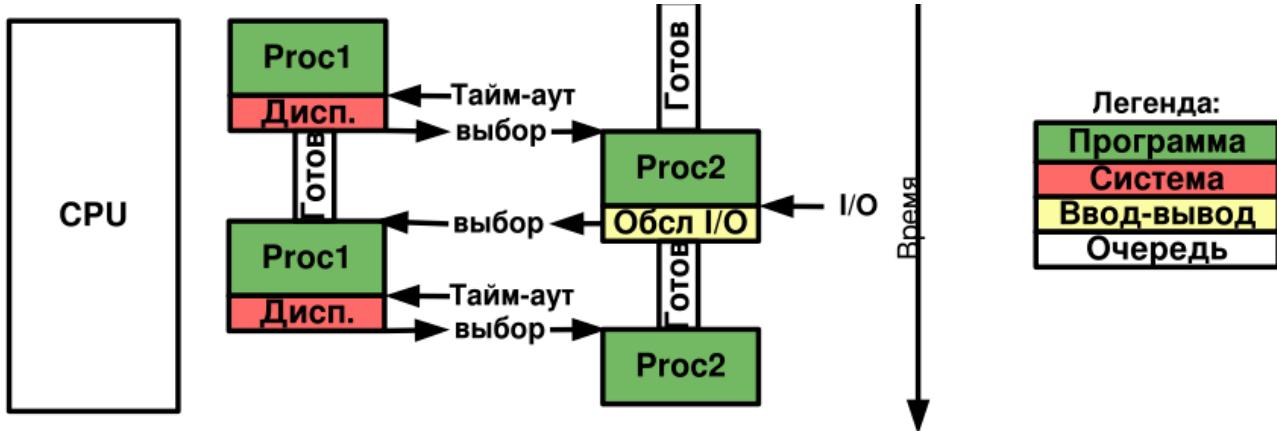
Вся информация характеристик процесса хранится в управляющем блоке процесса (PCB), который создается и управляет ОС. Важно то, что PCB содержит достаточную информацию, чтобы можно было прерывать процесс и позднее возобновить его исполнение. PCB является ключевым инструментом, который позволяет ОС поддерживать многопроцессорную работу системы

## Состояние процесса. Разделение ресурсов

С точки зрения процесса его работа состоит в выполнении определенного набора команд, последовательность которых задается значениями заносимые в счетчик команд.

Поведение процесса можно охарактеризовать, последовательно перечислив последовательность команд, которые были выполнены в ходе работы процесса. Такой перечень называется **следом (trace)**. Поведение процессора можно охарактеризовать, показав, как чередуются следы процессов.

**Рассмотрим простой пример**

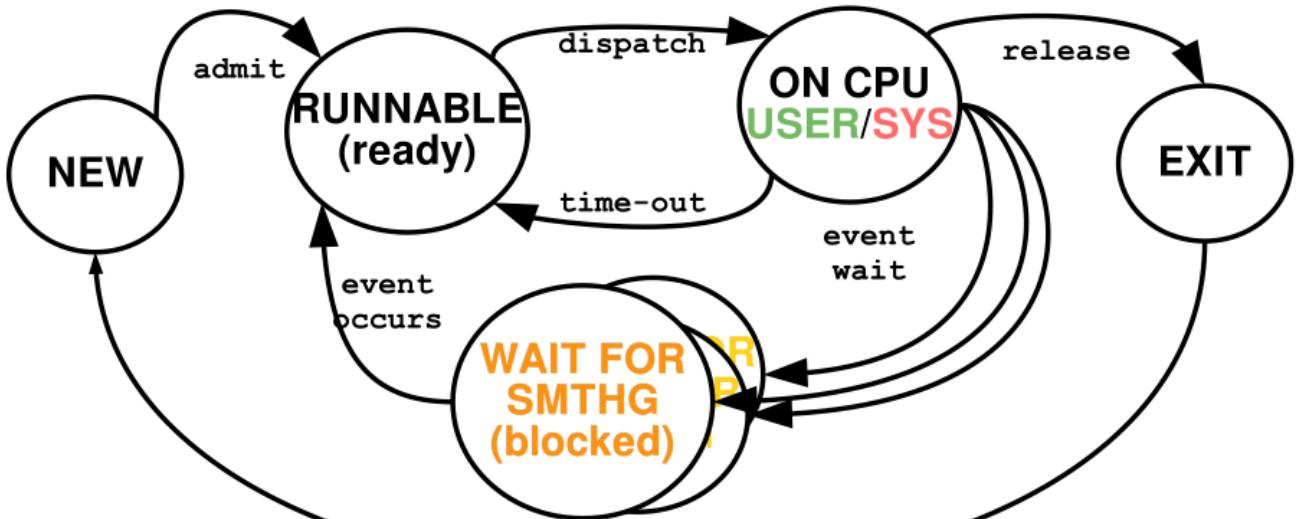


- Каждому процессу выделен какой-то квант времени, после которого запускается диспетчер, который принимает решения о необходимости переключения на другой процесс + для подсчета различных системных ресурсов. Если оба запущенных процесса имеют равный приоритет, то исходя из равноправного распределения ресурсов каждый из процессов должен исполняться примерно одинаковое количество времени на ядре.
- Сначала выполняется процесс 1. После истечения кванта времени происходит тайм-аут и запускается диспетчер
- Диспетчер смотрит и принимает решения о переключении на другой процесс. Происходит диспетчerezация и мы переключаемся на Процесс 2
- Начинает работать Процесс 2, но не доходит до истечения своего кванта времени и вместо этого появляется запрос на ввод-вывод. ОС видит, что происходит ввод-вывод и понимает, что смысла держать процесс, который тормозит систему, нет, поэтому опять диспетчerezуемся на Процесс 1 (при этом Процесс 2 блокируется и помещается в очередь ожидания)
- Процесс 1 готов, поэтому он начинает свою работу, израсходует свой квант времени и благодаря диспетчеру мы вернемся опять к Процессу 2 (если ввод-вывод завершился). Если ввод-вывода закончится раньше, чем Процесс 1 исчерпает свой квант, то ОС вытеснит Процесс 1 и вернется к Процесс 2 (вытесняющая многозаданность)

В Столлингсе глава 3 стр 150 (там более объемный пример, но мб более понятный)

## 23. Модель процесса с 5 состояниями, назначение состояний

Немного условная, но показательная картинка



## Операционные системы. Часть 2. Процессы и потоки



### New State

- Процесс создан, но еще не размещен в очереди процессов, готовых к исполнению (создан PCB, но, например, память не выделена или слишком много работающих процессов, поэтому ОС ждем пока их не станет меньше)
- **Причины создания процесса**
  - Вход в систему в интерактивном режиме (в систему с терминала входит новый пользователь)
  - Запуск скриптов пакетного задания (поступление управляющего потока пакетных заданий)
  - Запуск обработчика сервиса (ОС может создать процесс для выполнения некоторой функции, которая нужна для программы пользователя)
  - Порождение одного процесса другим

### Exit State

- Процесс не может продолжить свое исполнение дальше (структуры процесса все еще существуют)
- Выход происходит в два этапа
  - Процесс переходит в состояние завршающегося. ОС временно сохраняет таблицы и другую информацию, связанную с этим процессом, так что вспомогательные программы могут получить все необходимые сведения о завершающемся процессе.
  - После извлечения всей информации процесс больше не нужен и удаляется из системы (либо попадает на "кладбище" процессов откуда может быть возрожден дабы)
- **Причины попадания в состояние**
  - Обычное завершение (вызов exit)
  - Превышение лимитов на время выполнения
  - Недостаточный объем памяти

- Ошибки границ и защиты памяти (попытка получения данных из ячейки, на которую нет прав)
- Арифметическая ошибка
- Излишнее ожидание (опять же превышение лимита времени)
- Ошибка ввода-вывода
- Неправильная и привилегированная инструкция
- Команда оператора или ОС
- Завершение или запрос родительского процесса

## **Runnable State**

- Процесс обладает всеми ресурсами для выполнения, но нет возможности исполняться (все процессора заняты)
- **Причины нахождения в состоянии**
  - Низкий приоритет
  - Ожидание освобождений процессоров
  - Закончился квант времени

## **Running State**

- Команды приложения и ОС выполняются на процессоре
- **Процесс остается в этом состоянии если**
  - Не истек квант времени
  - Ожидание в спин-блокировке
  - В runnable состоянии нет процессов с более высоким приоритетом (вытесняющая многозадачность )
  - Обслуживанием высокоприоритетных прерываний
  - Нет блокирующих вызовов (ввод-вывод, ожидание блокировки)

## **Wait (blocked) State**

- Ожидание событий ОС, освобождение блокировки
- Есть несколько очередей для процессов в зависимости от причины блокировки или ожидания или также очереди могут создаваться по приоритетам
- **Состояние продолжается до тех пор пока:**
  - Освободиться блокировка
  - Придет сообщение от ОС о наступлении ожидаемого события (завершение ввода-вывода и прочее)
- Процесс не расходует ресурсы CPU
- Процесс может находиться в ожидании неопределенно долго (дедлок)
- Наиболее показательным является блокировки пейджинга, когда несколько процессов хотят сохранить часть памяти на диск или наоборот загрузить какую-то часть данных в память

- Если процесс пытаются убить, а он не убивается, то значит, что процесс находится именно на такой блокировке так как чтобы убить процесс ему нужно время на процессоре. При такой ситуации помогает перезагрузка ОС.

## 24. Paging и Swapping. Модель процессов с 7 состояниями

---

Причиной разработки схемы модели процесса с разными состояниями послужило более медленные операции ввода-вывода по сравнению с вычислениями. Конечно, при работе нескольких процессов (которые загружены полностью в память) при блокировке одного процесса может работать другой, но рано или поздно все процессы будут в заблокированном состоянии из-за медленной операции ввода-вывода.

Можно увеличить размер памяти, но это ведет к увеличению стоимости системы и программистам никогда не будет много памяти.

### Основной памяти всегда мало

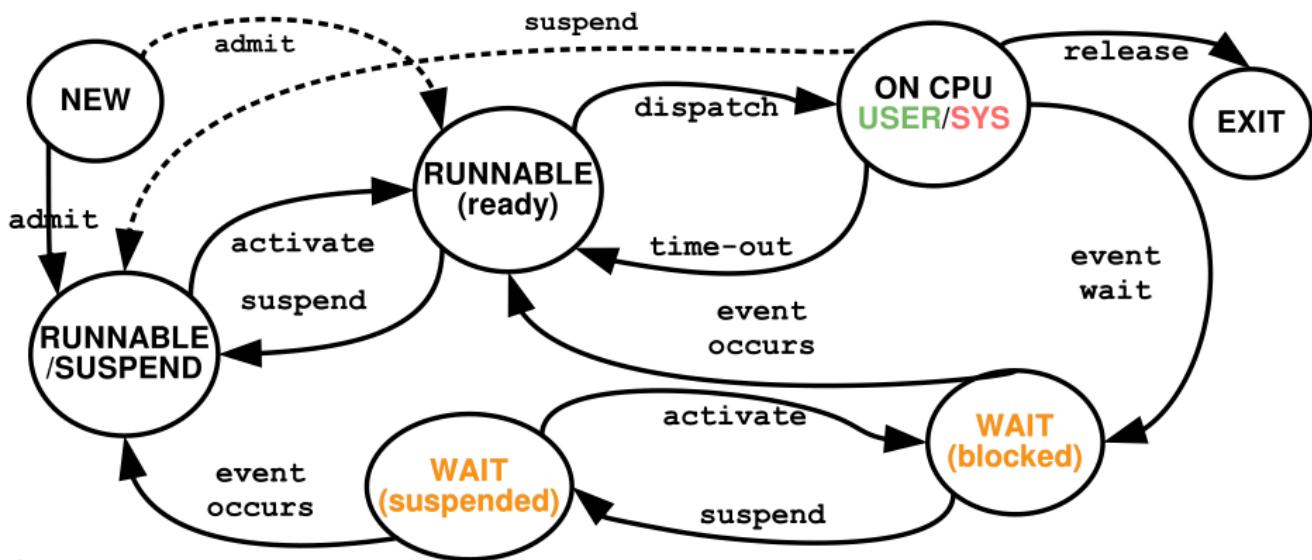
- Программисты используют максимально использовать ее
- Большое количество процессов

**Решение** -> поместить блокированный процесс (или процесс который не используется в принципе или долгое время )на диск и освободить основную память для других процессов. Для этого нужно организовать область подкачки процессов на диске.

**Swapping** -- выгрузка всего процесса, кроме критически важных для ядра структур управления (появился раньше + виртуальной памяти не было). Чистый свопинг не используется так как это долго с учетом, что нынешние программы могут занимать несколько Гб в памяти

**Paging** -- выгрузка (и загрузка) неиспользуемых страниц процесса на диск. После появления виртуальной памяти в основном сейчас используется пейджинг

Если в модель поведений процесса ввести свопинг, то придется ввести еще два состояния wait/suspended и blocked/suspended и в итоге мы получим модель с 7 состояниями



## Wait/Suspended State

- Процесс приостановлен и выгружен в область подкачки
- **Причины попадания**
  - Длительное ожидание событий ОС
  - Недостаток памяти. Нет смысла держать процесс, который не может исполниться
- По событию suspend процесс выгружается в область подкачки
- По событию activate процесс загружается в основную память
- Повышается нагрузка на дисковую подсистему

## Runnable/Suspended State

- Процесс готов к исполнению, но выгружен из памяти
- **Причины**
  - Был на Wait/Suspended и выгружен, но произошло событие, которое позволит процессу исполниться
  - Desperate Memory Conditions -- ситуация когда процессов настолько много, что самой ОС не хватает памяти, поэтому они все отправляются в своп
  - Команда пользователя (пользователь сам может приостановить процесс и при необходимости возобновить его)
  - Создание процесса в минимальном варианте без например создания сегментов памяти
  - Иногда также система сама может выгрузить процесс, готовый к исполнению, из-за приоритета и недостатка памяти

## Причины приостановки процессов

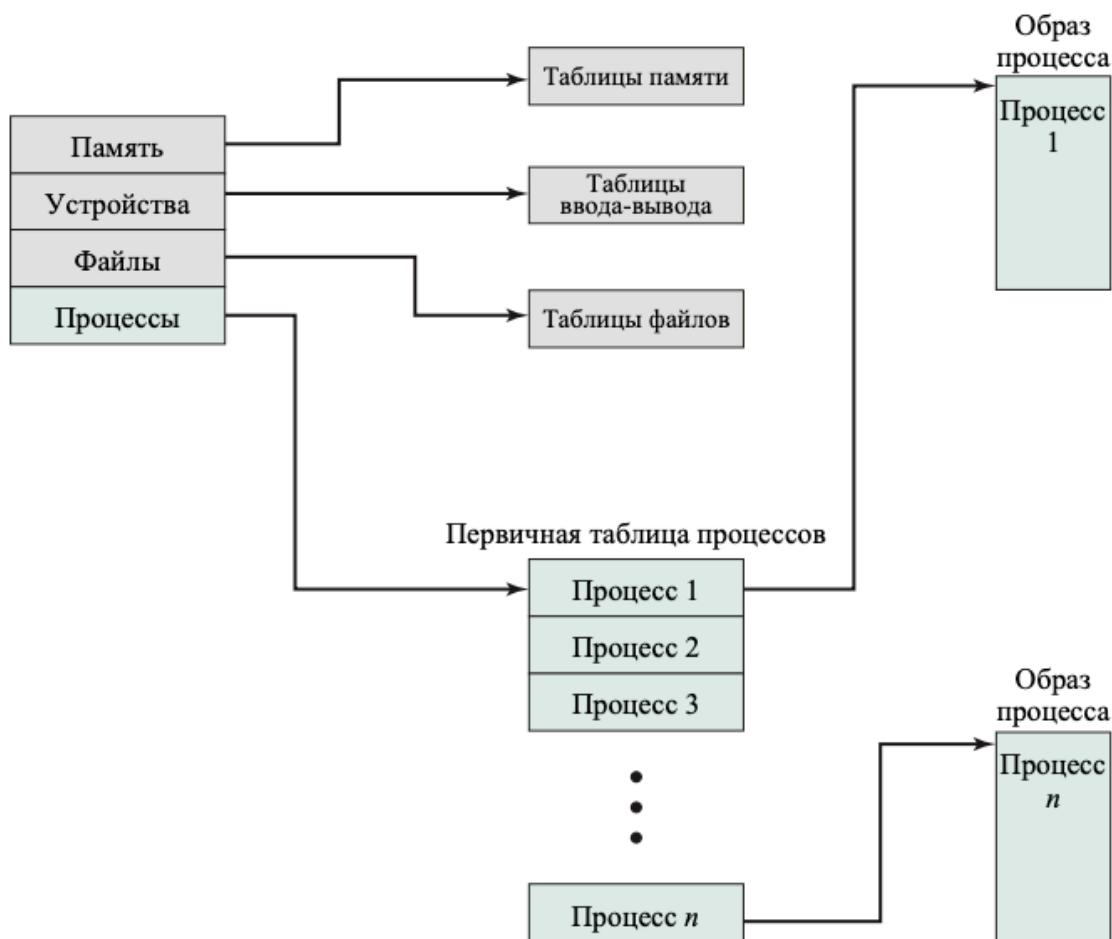
- **Swapping.** ОС нужно освободить память, чтобы загрузить готовый к исполнению процесс

- **Другие причины ОС.** ОС сама может приостановить фоновый или подозрительный процесс
- **Запрос интерактивного пользователя.** Пользователь может захотеть приостановить процесс, чтобы приступить к отладке программы или в связи с использованием некоторого ресурса
- **Временной режим выполнения.** Процесс может выполняться периодически и в интервалах, когда он не работает такой процесс может приостановиться
- **Запрос родительского процесса.** Родительский процесс может захотеть приостановить дочерние процессы для их проверки либо модификации, а также для координации их работы

## 25. Управляющие таблицы процесса. Образ процесса

---

Поскольку в задачи ОС входит управление процессами и ресурсами, она должна располагать информацией о текущем состоянии каждого процесса и ресурса. Самый универсальный способ -- это поддержка таблиц с информацией по каждому объекту управления. Общая таблица выглядит так и поддерживает информацию о памяти, вводе-выводе, файлов и процессов



### Таблицы памяти

Для отслеживания основной и вторичной памяти. Некоторая часть основной памяти резервируется для ОС, остальная же отведена на процессы. Процессы находящиеся

во вторичной памяти используют механизм виртуальной памяти либо свпоинга.

Таблицы памяти должны включать такую информацию как:

- объем основной памяти, отведенной процессу
- объем вторичной памяти для процесса
- все атрибуты защиты блоков основной или виртуальной памяти, как, например, указание, какой из процессов имеет доступ к той или иной совместно используемой области памяти;
- вся информация, необходимая для управления виртуальной памятью.

### **Таблицы ввода-вывода**

используются операционной системой для управления устройствами ввода-вывода и каналами компьютерной системы. В каждый момент времени устройство ввода-вывода может быть либо свободно, либо отдано в распоряжение какому-то определенному процессу. Если выполняется операция ввода-вывода, операционная система должна иметь информацию о ее состоянии и о том, какие адреса основной памяти задействованы в этой операции в качестве источника вывода или месте откуда передаются данные при вводе.

### **Таблицы файлов**

В этих таблицах находится информация о существующих файлах, их расположении во вторичной памяти, текущем состоянии и других атрибутах. Большая часть этой информации, если не вся, может поддерживаться системой управления файлами. В этом случае операционная система мало знает (или совсем ничего не знает) о файлах.

### **Таблицы процессов**

это структура данных, используемая операционной системой для хранения информации о каждом процессе, запущенном в системе. Она играет ключевую роль в управлении процессами, предоставляя ОС возможность отслеживать их состояние, ресурсы и взаимодействие.

Каждая запись в таблице процессов представляет из себя **образ процесса**

### **Типичные элементы образа процесса**

- **Данные пользователя.** Допускающая изменения часть пользовательского адресного пространства. Сюда могут входить данные программы, пользовательский стек и модифицируемый код
- **Пользовательская программа**
- **Системный стек.** С каждым процессом связан один или несколько системных стеков. Стек используется для хранения параметров, адресов возврата процедур и системных служб.
- **Управляющий блок процесса.** Данные необходимые для ОС для управления процессом

В простейшем случае образ процесса располагается в виде непрерывного блока памяти, расположенный во вторичной памяти.

Так как процессы могут также быть разбиты на страницы и следовательно также не весь процесс может находиться в основной памяти для его запуска, поэтому таблицы страниц должны обладать информацией о нахождении каждого образа процесса. Она может содержать в первичной таблице либо в самом образе процесса.



## 26. Управляющий блок процесса, состав PCB

Внутри образа процесса содержится **управляющий блок процесса**, который хранит информацию или набор атрибутов, которые используются ОС для управления этим процессом

Информация, которая хранится внутри PCB можно разбить на три основные категории:

- Информация по идентификации процесса
- Информация по состоянию процесса
- Информация, используемая при управлении процессом

Иногда под PCB понимают ту структуру, которая отражает процесс (`task_struct` в Linux). В других случаях под PCB могут иметь вид только ту часть, которая соответствует регистрам. Правильней всего понимать под PCB целиком ту структуру, которая управляет процессом

Можно также сказать, что состояние в ОС задается совокупностью PCB, так как почти каждый модуль модифицирует информацию, хранящуюся в этом блоке. Из-за этого возникает проблема при проектировании ОС. Дать доступ к PCB нетрудно, однако возникает вопрос как сохранить целостность этого блока и сдлеать так, чтобы какой-то модуль не повредил его. В качестве решения можно предложить использовать программу-обработчик, единственной задачей которой будет защита блока процесса и предоставление доступа к нему. Остается вопрос об эффективности такого решения.

### **Рассмотрим каждый пункт**

#### **1)Идентификаторы**

Числовые id, которые могут храниться в PCB

- id данного процесса
- id родительского процесса
- id пользователя (кто из пользователей отвечает за это задание)

Причем тут могут храниться не только id, но и другая идентификационная информация

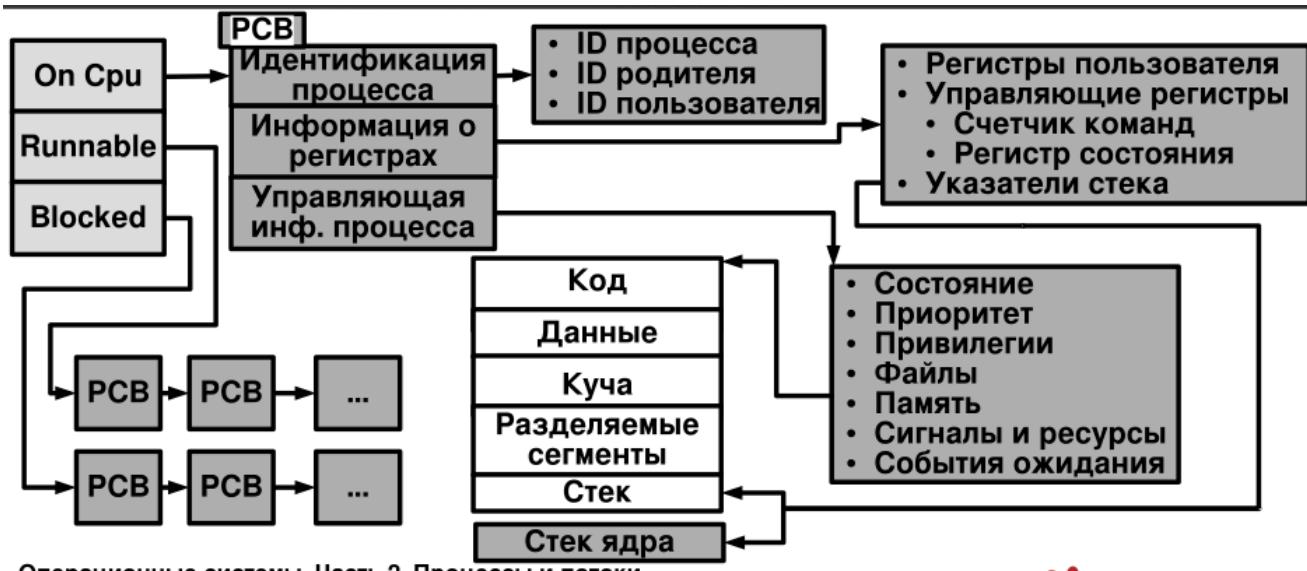
#### **2)Информация о состоянии процессора (информация о регистрах)**

- **Регистры доступные пользователю** -- те регистры, к которым можно обратиться посредством машинных команд, выполняющихся процессором. Обычно их от 8 до 32, однако в некоторых реализациях RISC встречается и свыше 100 регистров
- **Управляющие регистры и регистры состояния**
  - Счетчик команд
  - Коды условий -- отражают результат выполнения последней арифметической операции
  - Информация о состоянии -- сюда входят флаги разрешения прерываний и информация о режиме выполнения
- **Указатели стеков** -- указывает на его вершину. С каждым процессом связан один или несколько системных стеков, которые хранят параметры, адреса возвратов. В процессорах любого вида имеется регистр или набор регистров, известных под названием слово состояния программы (programm status word -- PSW), в которых содержится информация о состоянии и кодах условий.

#### **3)Управляющая информация процесса**

- **Информация по планированию и состоянию**
  - Состояние процесса
  - Приоритет -- может быть несколько полей для описания приоритета процесса (приоритет по умолчанию, текущий приоритет, максимальный приоритет)
  - Информация, связанная с планированием -- зависит от алгоритма планирования. К примеру может храниться время ожидания или время, в течении которого процесс выполнялся при последнем запуске

- Информация о событиях -- id события, наступление которого позволит продолжить выполнение процесса, находящегося в состоянии ожидания
- **Указатели на другие процессы для поддержки родственных связей**
- **Обмен информацией между процессами**
  - Различные флаги, сигналы и сообщения могут иметь отношения к обмену информацией между двумя процессами. Некоторая или вся информация может храниться внутри PCB
- **Привилегии процессов**
  - Могут выражаться в правах доступа к определенной области памяти либо к возможности выполнению определенных видов команд.
  - Возможность использования различных системных утилит и служб
- **Управление памятью**
  - Указатели на таблицы сегментов и/или страниц, в которых описывается распределение процесса в виртуальной памяти
- **Владение ресурсами и их использование**
  - Ресурсы, которыми владеет процесс (например, файл)
  - Может храниться история использования ресурсов, что может пригодиться при планировании



## 27. Функции ОС, связанные с процессами. Создание процессов, переключение процессов

### Функции ОС, связанные с процессами

#### Управление процессами

- Создание и завершение процессов
- Планирование и диспетчеризация процессов

- Переключение процессов
- Синхронизация и поддержка обмена информацией между процессами
- Организация PCB

### **Управление памятью**

- Выделение адресного пространства процессам

- Пейджинг и свопинг

- Управление страницами и сегментами

### **Управление вводом-выводом**

- Управление буферами

- Выделение процессам каналом и устройств ввода-вывода

### **Функции поддержки**

- Обработка прерываний

- Учет использования ресурсов

- Текущий контроль системы

## **Создание процесса**

### **1. Присвоить новому процессу уникальный id**

- В первичную таблицу процессов вносится запись

### **2. Выделить пространство для процесса**

- Включаются все элементы образа процесса. ОС должна знать сколько будет занимать пользовательское пространство
- Размер может задаваться по умолчанию либо исходя из запроса пользователя
- Если новый процесс создает другой процесс то родительский процесс может передать необходимые данные со своим запросом
- Необходимо установить соответствующие связи, если процесс будет использовать совместное адресное пространство
- Выделить место для PCB

### **3. Инициализация PCB**

- Ставятся все id (самого процесса, родительский и пользователя)
- Информация о состоянии процессора инициализируется нулями за исключением счетчика команд и указателя системного стека.
- Управляющая информация ставится на основе значений, установленных по умолчанию + берутся атрибуты из запроса. Например, приоритет может ставится минимальным, если другого не сказано. Изначально процесс может не владеть никакими ресурсами, если не был сделан явный запрос или если они не передались по наследству от родителя

### **4. Установить необходимые связи (поставить процесс в очередь ядра)**

### **5. Создать потоки ввода-вывода (stdin, stdout, stderr)**

### **6. Создать или расширить другие структуры данных**

- ОС может к примеру поддерживать для каждого процесса файл для учета ресурсов

## Переключение процесса

Процесс может работать в режиме ядро или в режиме пользователя

## Возможные причины, по которым управление может перейти к ОС

- **Прерывание**

- Внешнее по отношению к выполнению текущей команды
- Отклик на внешнее асинхронное событие

- **Ловушка**

- Связана с исполнением текущей команды
- Обработка ошибки или исключительная ситуация

- **Вызов супервизора**

- Явный запрос
- Вызов функций ОС

При обычно перывании управление сначала передается обработчику прерываний, который осуществляет подготовительные работы, а затем -- функции ОС, отвечающие за прерывания данного вида. Получаем следующие ситуации перехода из user в kernel-mode

- **Прерывание таймера**

- Процесс выполняется в течении кванта времени, поэтому ОС решает переключить его

- **Прерывание ввода-вывода**

- ОС определяет, что именно произошло и переводит все нужные процессы из блокированного состояния в состояние готовности
- Затем надо принять решение продолжить ли исполнять текущий процесс либо переключить на готовый процесс

- **Ошибка отсутствия блока в памяти (page fault)**

- При отсутствии блока памяти процесса в основной памяти его необходимо загрузить, поэтому ОС переводит процесс в блокированное состояние и может передать управление другому процессу.

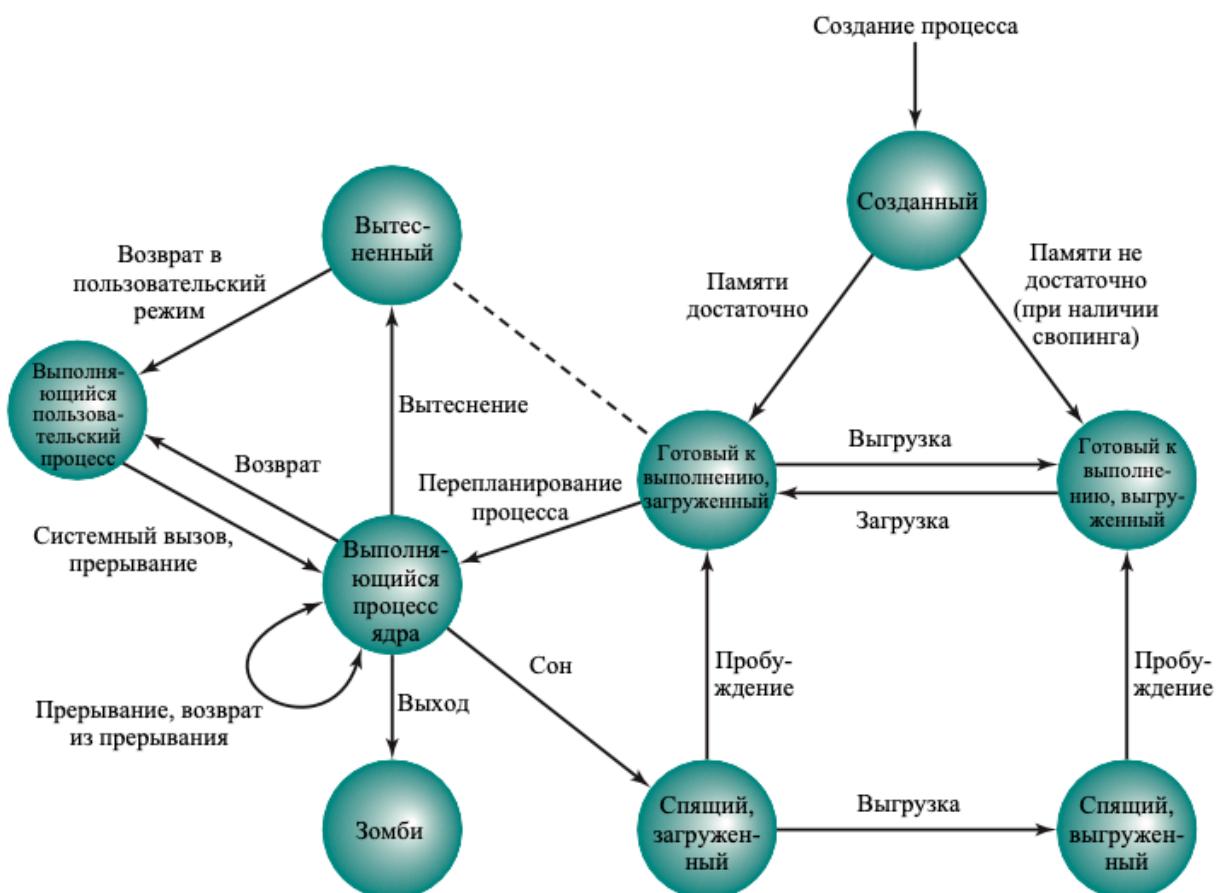
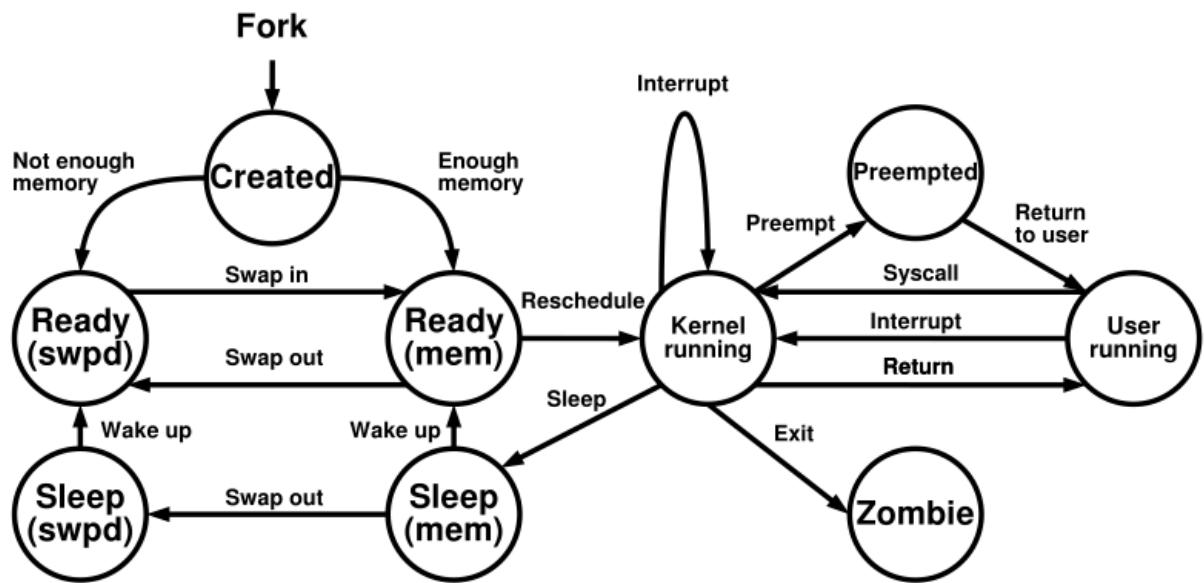
В случае **ловушки** все зависит от причины прерывания. Если ошибка фатальная, то процесс переходит в завершающееся состояние. Если нет, то ОС может попытаться восстановить процесс либо просто уведомить пользователя об ошибке.

Вызов **супервизора**, который исходит от выполняемой программы. Например, открытие файла, которое неизбежно приведет нас к выполнению кода на уровне ядра

При переключении режима мы должны сохранить PCB, в который входит информация о регистрах и о стеках. Если после прерывания не последует переключение процесса, то нам в целом достаточно сохранить информацию только о состоянии процессора.

## 28. Процессы в ОС Unix SVR4. Диаграмма состояний, основные структуры

Всего в Unix SVR4 есть 9 состояний



- User Running (выполняющийся в пользовательском режиме)

- **Kernel running** (выполняющийся в режиме ядра)

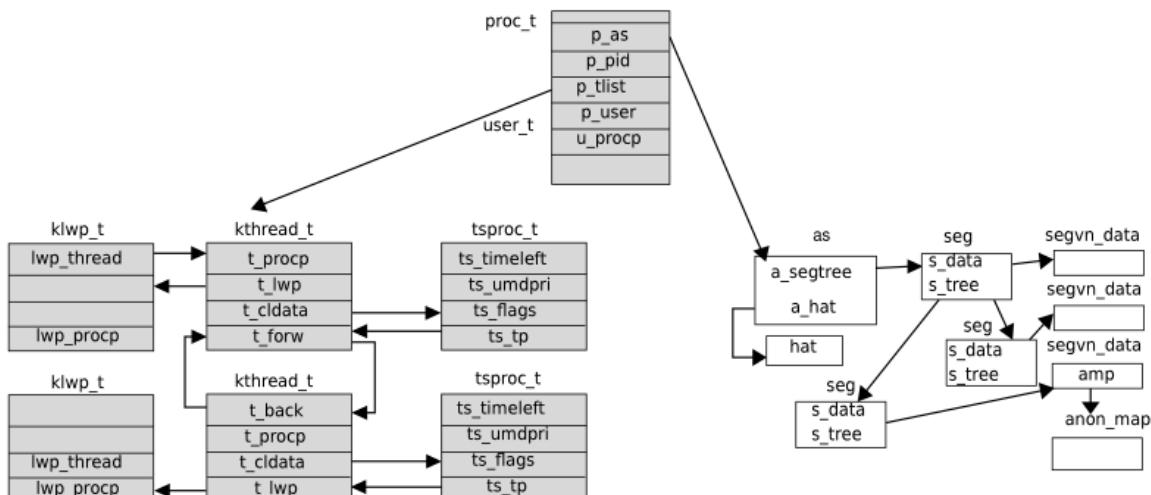
Причем user running и kernel running связаны прерываниями и ловушками, а также сисколами как отдельным видом прерывания. Вернуться в user mode можно возвратом из прерывания.

- **Ready\_mem** (готов к выполнению/загруженный в память) -- готов к выполнению как только ядро решит передать управление процессу
- **Sleep\_mem** (спящий/загруженный) -- не может выполняться, пока не произойдет некоторое событие; процесс загружен в память (блокированный)
- **Ready\_swpd** (готов/выгруженный) -- готов к исполнению, но должен произойти процесс свопинга прежде чем процесс сможет начать исполняться
- **Sleep\_swpd** (спящий/выгруженный) -- ожидает события; выгружен из памяти (блокированный)
- **Preempted** (вытесненный) -- в момент переключения режима ядра в пользовательский режим ядро решает передать управление другому процессу. **К тому же на схеме не указано, но ядро может выкинуть процесс в состояние готовности на этом этапе**
- **Created** (созданный) -- процесс только что создан. В зависимости от наличия свободной памяти может быть сразу загружен в основную память либо уйти в read\_swpd
- **Zombie** -- самого процесса больше не существует, но осталась записи о нем, чтобы ими мог воспользоваться родительский процесс

Состояние вытесненного процесса и готового/загруженного по сути одинаковы, однако приходят они туда по разным причинам

Кроме того в Unix есть два процесса, которых нет ни в каких других ОС. **Процесс 0** (**процесс свопинга**) -- специальный процесс, который создается при загрузке системы и загружается вместе с системой. Также процесс 0 порождает **процесс 1**, который является родительским по отношению ко всем остальным и при входе интерактивного пользователя именно этот процесс создает для пользователя новый процесс.

## Процессы в ОС Unix SVR4 и основные структуры



- В Solaris структура PCB состоит из двух частей: proc\_t и user\_t (могут занимать порядка 2Кб)
- p\_as содержит указатель память, которая содержит структуры, которые относятся к адресному пространству, конкретному сегменту и есть также структуры которые говорят нам именованное адресное пространство это или нет.
- Внутри процесса также есть такая абстракция как трэды, которые тоже представлены своими структурами данных. LWP (light-weight thread) по сути является отображение пользовательских потоков на потоки ядра и каждый LWP связан с kthread\_t на уровне ядра.
- hat (hardware address translation) --структура, отвечает за трансляцию виртуального адреса в физический.

Образ процесса можно разделить на 3 части:

### **Контекст пользователя**

- Текст процесса -- выполняемые машинные команды
- Данные процесса
- Пользовательский стек
- Совместно используемая память

### **Контекст регистров**

- Счетчик команд
- Регистр состояния процессора -- содержит состояние аппаратного обеспечения.  
Зависит от реализации
- Указатель стека -- указывает на вершину стека ядра (или user стека в зависимости от режима работы)
- Регистры общего назначения

### **Контекст системного уровня**

- Запись таблицы процессов -- определяет состояние процесса
- Пользовательская область -- информация по управлению процессом, нужна только в контексте данного процесса
- Таблица областей процесса -- Задает отображение виртуальных адресов в физические; содержит также поле полномочий, в котором указывается тип доступа, на который процесс имеет право: только для чтения, для чтения и записи или для чтения и выполнения
- Стек ядра

## Основные элементы записи таблицы процессов в Unix

<b>Состояние процесса</b>	Текущее состояние процесса
<b>Указатели</b>	Пользовательская область и область памяти процесса (текст, данные, стек)
<b>Размер процесса</b>	Дает возможность операционной системе определить, сколько памяти потребуется процессу
<b>Идентификаторы пользователя</b>	<b>Реальный идентификатор пользователя</b> (real user ID) указывает, кто из пользователей несет ответственность за выполняющийся процесс. <b>Фактический идентификатор пользователя</b> (effective user ID) может использоваться процессом для предоставления временных привилегий, связанных с определенной программой; на время выполнения этой программы в составе процесса последний использует фактический идентификатор пользователя
<b>Идентификаторы процесса</b>	Идентификатор данного и родительского процессов. Эти идентификаторы присваиваются процессу в состоянии создания
<b>Дескриптор событий</b>	Используется, когда процесс находится в спящем состоянии; с наступлением события процесс переходит в состояние готовности
<b>Приоритет</b>	Используется при планировании процессов
<b>Сигнал</b>	Перечисляет отправленные, но еще не обработанные сигналы
<b>Таймеры</b>	Включают время выполнения процесса, использование ресурсов ядром, а также пользовательские таймеры для отправки сигналов в определенное время
<b>P-связь</b>	Указатель на следующий элемент в очереди готовых к выполнению процессов (используется, когда процесс находится в состоянии готовности)
<b>Состояние памяти</b>	Указывает, находится ли образ процесса в основной памяти или выгружен из нее. Если процесс загружен в память, в этом поле также указывается, можно ли его выгрузить или он временно блокирован в основной памяти

## Пользовательская область

<b>Указатель таблицы процессов</b>	Указывает запись, соответствующую области пользователя
<b>Идентификаторы пользователя</b>	Реальный и фактический идентификаторы пользователя. Используются для определения пользовательских привилегий
<b>Таймеры</b>	Записывают время, затраченное на выполнение данного и дочерних процессов в пользовательском режиме и в режиме ядра
<b>Массив обработчиков сигналов</b>	Указывает, как будет реагировать процесс на каждый из пяти типов сигналов, заданных в системе (завершаться, игнорировать сигнал, выполнять заданную пользователем функцию)
<b>Управляющий терминал</b>	Указывает, с какого терминала был запущен процесс (если этот терминал существует)
<b>Поле ошибок</b>	Содержит записи об ошибках, произошедших во время системного вызова

<b>Возвращаемое значение</b>	Содержит результат выполнения системных вызовов
<b>Параметры ввода-вывода</b>	Задает объем передаваемых данных, адрес массива данных в пользовательском пространстве, а также смещения в файлах при вводе-выводе
<b>Файловые параметры</b>	Текущий и корневой каталоги описывают файловую систему процесса
<b>Таблица дескрипторов файлов пользователя</b>	Содержит записи об открытых файлах
<b>Границные поля</b>	Ограничивают размер процесса и размер файла, который он может записать
<b>Поля режимов доступа</b>	Установки режима доступа к создаваемым процессом файлам

## 29. Понятие потока выполнения, связь потока и процесса. Преимущества потоков.

---

Концепция процесса можно охарактеризовать двумя свойствами

- **Владение ресурсами.** Время от времени процесс может владеть ресурсами такими как: основная память, каналы и устройства ввода-вывода, файлы и тд. ОС выполняет защитные функции, предотвращая нежелательные взаимодействия процессов на почве владения ресурсов
- **Планирование и диспетчеризация.** Выполнение процесса может чередоваться с выполнением других процессов, поэтому процесс имеет такие параметры как состояние и приоритет диспетчеризации и представляет собой сущность по отношению к которой ОС выполняет планирование и диспетчеризацию.

Для различия двух характеристик выше, единицу диспетчеризации обычно называют **потоком** или **облегченным процессом**. А единицу владения ресурсами -- **процессом** или **заданием**

**Thread** -- единица выполнения программного кода

- Диспетчируемая единица работы, включающая контекст процессора (в который входит содержимое счетчика команд и указателя вершины стека), а также собственную область стека (для организации вызова подпрограмм). Команды потока выполняются последовательно; поток может быть прерван при переключении процессора на обработку другого потока.
- **Процесс** -- единица группировки общих ресурсов
- Набор из одного или нескольких потоков, а также связанных с этими потоками системных ресурсов (таких, как область памяти, в которую входят код и данные, открытые файлы, различные устройства). Эта концепция очень близка концепции выполняющейся программы. Разбивая приложение на несколько потоков,

программист получает все преимущества модульности приложения и возможность управления связанными с приложением временными событиями.

В многопоточной среде процесс определяется как структурная единица распределения ресурсов, а также структурная единица защиты. **С процессом связаны следующие элементы**

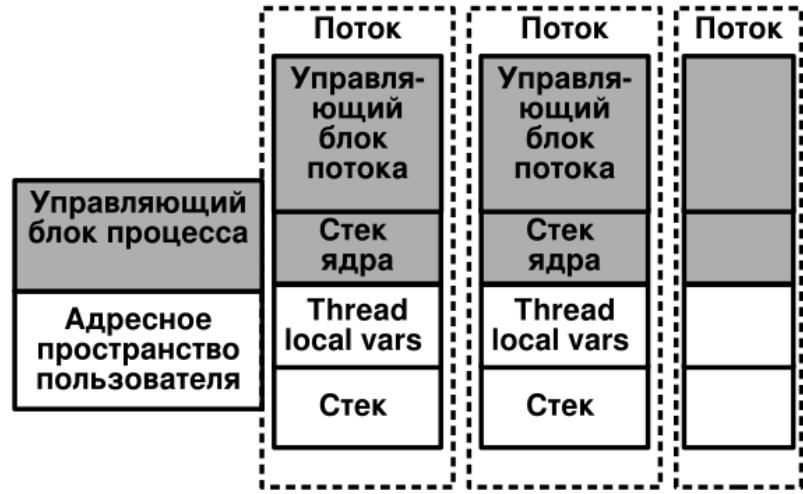
- Виртуальное адресное пространство, в котором живет образ процесса
  - Защищенный доступ к процессорам, другим процессам, файлам и ресурсам ввода-вывода
- В рамках процесса могут находиться один или несколько потоков, каждый из которых **обладает следующей характеристикой**
- Состояние выполнение потока
  - Сохраненный контекст (регистры и тд)
  - Стек выполнения (ядра и юзера)
  - Локальные переменные и статическая память для них. Они живут и умирают вместе с потоком
  - Доступ к памяти и ресурсам процесса, которому этот поток принадлежит. Доступ разделяется между всеми потоками процесса

#### Связь процесса и потока

#### Однопоточная модель



#### Многопоточная модель



В однопоточной модели процесса(где концепция потока и процесса одинаковы) в его представление входят управляющий блок этого процесса, пользовательское адресное пространство, а также стеки ядра и юзера, с помощью которых выполняются вызовы процедур и возвраты из них.

Пока процесс работает он управляет регистрами процессора. Когда выполнение процесса прерывается, регистры сохраняются в памяти. И вся информация про регистры находится в PCB.

В многопоточной среде с каждым процессом тоже связан один общий PCB и общее адресное пространство, к которому имеют доступ всем потоки, но теперь также у

каждого потока есть свой PCB, отдельный стек как ядра так и юзера и также пространство для локальных переменных.

Таким образом, все потоки процесса разделяют между собой состояние и ресурсы этого процесса, но структуры, которые отвечают за исполнение разные . Потоки находятся в одном и том же адресном пространстве и имеют доступ к одним и тем же данным. Если один поток изменяет в памяти какие-то данные, то другие потоки во время своего доступа к этим данным имеют возможность отследить эти изменения. Если один поток открывает файл с правом чтения, другие потоки данного процесса тоже могут читать из этого файла. Поэтому может возникнуть конкуренция за ресурсы с учетом того, что доступ к этим ресурсам осуществляется с большой скоростью

### Преимущества потоков

- Создание потока во много раз быстрее превышает создание нового процесса по скорости
- Потоки завершаются намного быстрее чем процесс (все равно стараюся держать пул потоков)
- Переключение между потоками в рамках одного процесса быстрее
- Обмен сообщений между потоками в рамках одного процесса быстрее так как они используют одну и ту же область памяти и одни и те же файлы, то участия ядра тут нет в отличии от процессов

### Потоковая конструкция также полезна в однопроцессорных системах

- **Работа в приоритетном и фоновом режимах.** Один поток может считывать ввод юзера, другой обновлять саму таблицу. Такая схема увеличивает воспринимаемую пользователем скорость работы системы и дает возможность ввода новых данных, даже когда предыдущие еще не были полностью обработаны
- **Асинхронная обработка.** Например создание потока, который производил бы резервное копирование раз в промежуток времени
- **Скорость выполнения.** Параллельная работа treadов позволяет увеличить скорость работы программы
- **Модульная структура программ.** Программы, осуществляющие разнообразные действия или выполняющие множество вводов из различных источников и выводов в разные места назначения, легче разрабатывать и реализовывать с помощью потоков.

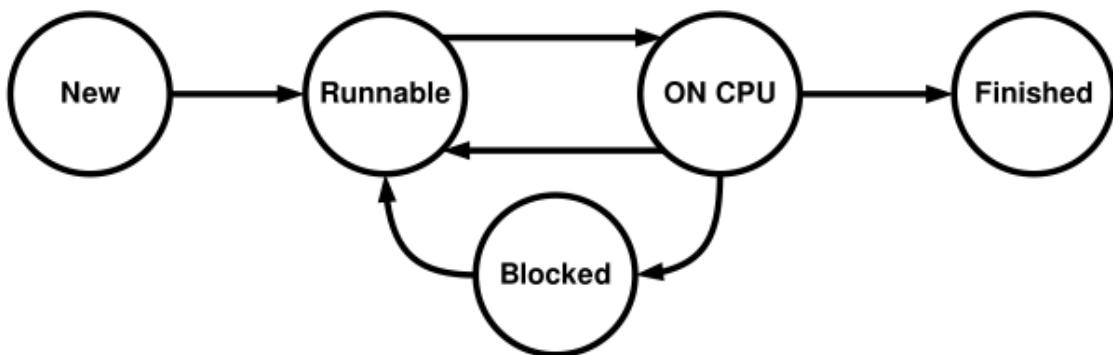
## 30. Состояние потоков. User Level Threads vs Kernel Level Threads

---

Состояния потоков похожи на состояния процессов за исключением отсутствия специфичных состояний связанных с пейджингом и свопингом

## Состояния

- **Порождение.** Обычно одновременно с процессом создается новый поток. Далее в рамках процесса один поток может породить другой. Новый поток создается со своим контекстом регистров и стековым пространством, после чего помещается в очередь готовых потоков
- **Runnable (готов).** Поток готов к исполнению и ждет своей очереди на CPU
- **On CPU.** Исполнение на CPU. Может быть его покинуть полностью реализовав свой квант времени либо если заблокирован операцией ввода-вывода
- **Блокированное.** Если потоку нужно подождать наступления некоторого события, он блокируется (при этом сохраняются содержимое его пользовательских регистров, счетчика команд, а также указатели стеков). После этого процессор может перейти к выполнению другого готового потока.
- **Завершение.** Удаление контекста регистров и стека



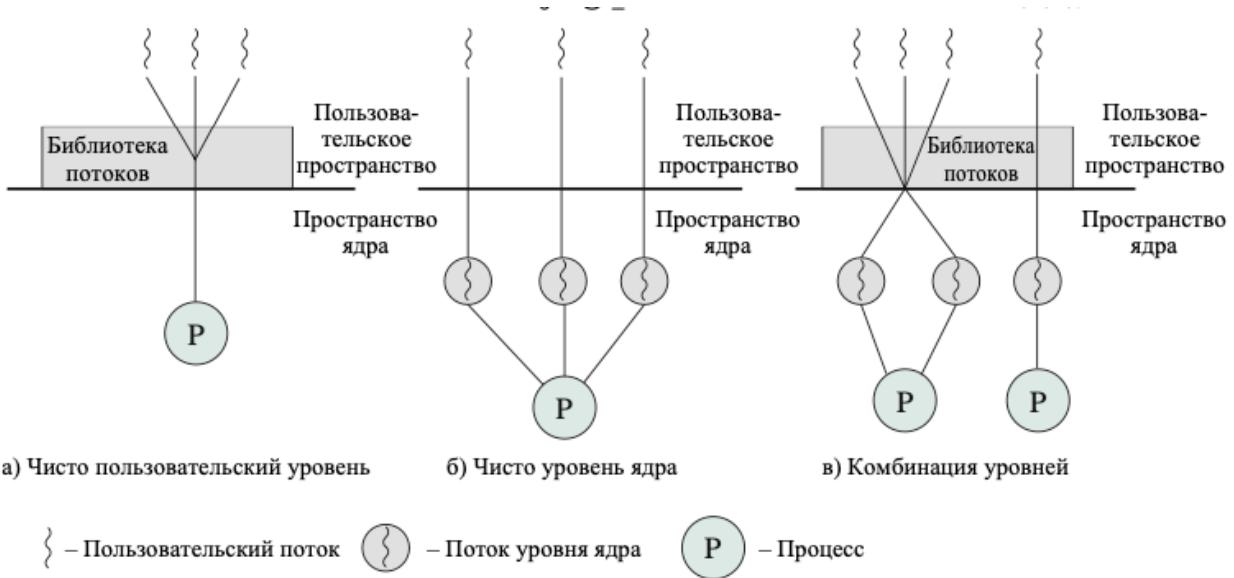
Важно, что адресное пространство потоков одно, значит необходима синхронизация по общим данным

Также блокировка потока не должна приводить к блокировке всего процесса.

## Типы потоков

Выделяются два общих вида потоков

- **ULT (user-level threads либо green threads)** -- потоки на уровне юзера, реализуются библиотеками
- **KLT (kernel level threads или lightweight processes)** -- потоки на уровне ядра и реализуются ядром.



**В ULT (первая картинка)** все действия по управлению потоками выполняются самим приложением, ядро по сути и не знает о существовании потоков. Чтобы приложение было многопоточным, его следует создавать с применением **специальной библиотеки**, представляющей собой пакет программ для работы с потоками на уровне ядра. Такая библиотека для работы с потоками содержит код, с помощью которого можно создавать и удалять потоки, производить обмен сообщениями и данными между потоками, планировать их выполнение, а также сохранять и восстанавливать их контекст. По сути библиотека выполняет действия ядра только на уровне юзера.

#### Некоторые преимущества ULT

- Переключение потоков не предусматривает переход в режим ядра
- Алгоритм планирования может выбирать с учетом специфики определенного приложения и это не повлияет на алгоритм планирования внутри ядра
- Поддержка любой ОС. Изменения в ядро вносить не надо, а библиотека это просто набор утилит, работающих на уровне приложения

#### Недостатки ULT

- Многие системные вызовы внутри ОС блокирующие, что приводит к блокировке всех потоков процесса
- За одним процессом закрепляется один процессор, поэтому мы не сможем выполнять несколько потоков одновременно, однако тем не менее такой подход может давать свои выигрыши

Эти две проблемы разрешимы. Например, их можно преодолеть, если писать приложение не в виде нескольких потоков, а в виде нескольких процессов. Однако при таком подходе основные преимущества потоков сводятся на нет: каждое переключение становится не переключением потоков, а переключением процессов, что приводит к значительно большим накладным затратам.

Другим методом преодоления проблемы блокирования является преобразование блокирующего системного вызова в неблокирующий, известный как *jacketing*. Например, вместо непосредственного вызова системной процедуры ввода-вывода поток вызывает подпрограмму-оболочку, которая производит ввод-вывод на уровне приложения. В этой программе содержится код, который проверяет, занято ли устройство ввода-вывода. Если оно занято, поток передает управление другому потоку (что происходит с помощью библиотеки потоков). Когда наш поток вновь получает управление, он повторно осуществляет проверку занятости устройства ввода-вывода.

### Чаще всего сейчас встречаются KLT

Вторая картинка.

В области приложений нет кода по управлению потоками.

Каждый поток юзера отображается на поток ядра и планирование выполняется ядром на основе потоков.

- Ядро может осуществлять планирование работы нескольких потоков одного и того же процесса на нескольких процессах
- При блокировке одного потока ядро может выбрать на исполнение другой поток. Однако для переключение между потоками мы должны переключаться между режимами и опускаться в ядро, что дает свои расходы по времени.

### Комбинированный подход

В комбинированных системах создание потоков выполняется полностью в пользовательском пространстве, там же, где и код планирования и синхронизации потоков в приложениях. **Несколько потоков на пользовательском уровне, входящих в состав приложения, отображаются в такое же или меньшее число потоков на уровне ядра.** Программист может изменять число потоков на уровне ядра, подбирая его таким, чтобы оно позволило достичь наилучших результатов.

При комбинированном подходе несколько потоков одного и того же приложения могут выполняться одновременно на нескольких процессорах, а блокирующие системные вызовы не приводят к блокировке всего процесса. При надлежащей реализации такой подход будет сочетать в себе преимущества подходов, в которых применяются только потоки на пользовательском уровне или только потоки на уровне ядра, сводя недостатки каждого из этих подходов к минимуму.

## 31. Многопроцессорность и многопоточность. Закон Амдала

---

Потенциальные преимущества многоядерных систем в смысле производительности зависят от способности эффективно использовать параллельные ресурсы, доступные

приложению. И можно задаться вопросом насколько можно ускорить программу на N процессорах с использованием потоков ?

Закон Амдала был предложен Джином Амдалом в 1967 году и касается потенциального ускорения многопроцессорной программы по сравнению с однопроцессорной.

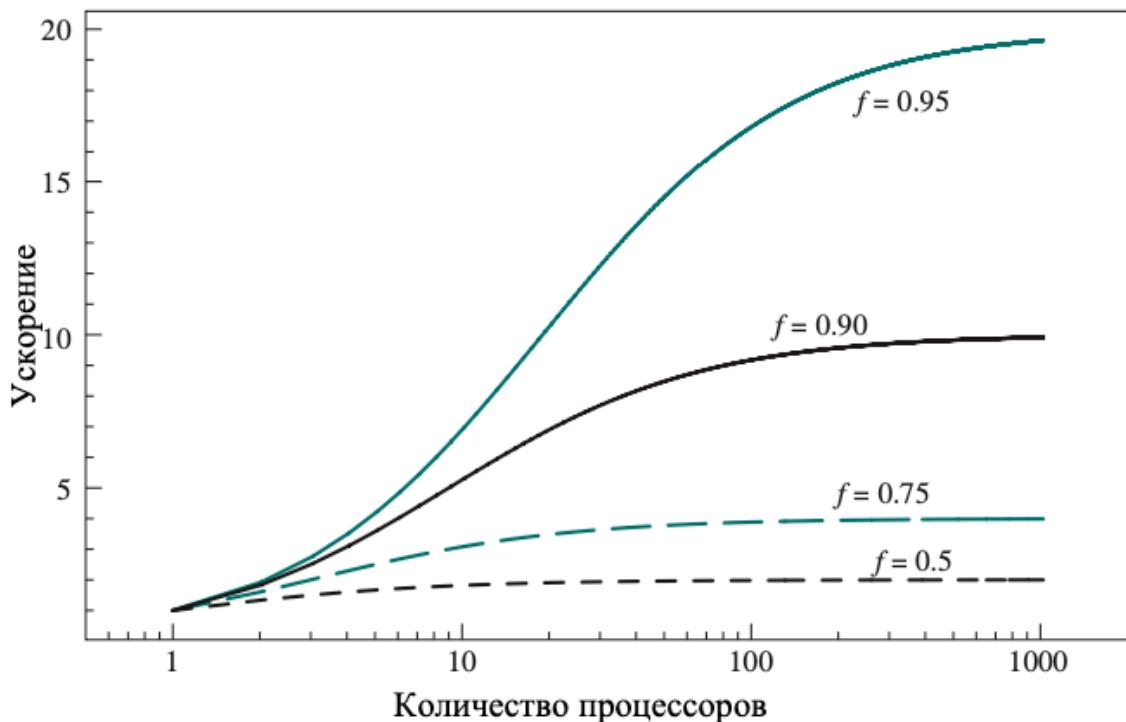
$$\text{Ускорение} = \frac{\text{Время - выполнения - на - одном - процессоре}}{\text{Время - выполнения - на - } N - \text{процессорах}} = \frac{T \cdot (1 - f) + T \cdot f}{T \cdot (1 - f) + \frac{T \cdot f}{N}} = \frac{1}{(1 - f) + \frac{f}{N}}$$

$f$  - часть кода, которая может быть бесконечно распараллелена, тогда  $(1 - f)$  - последовательная часть

$T$  - общее время выполнения программы

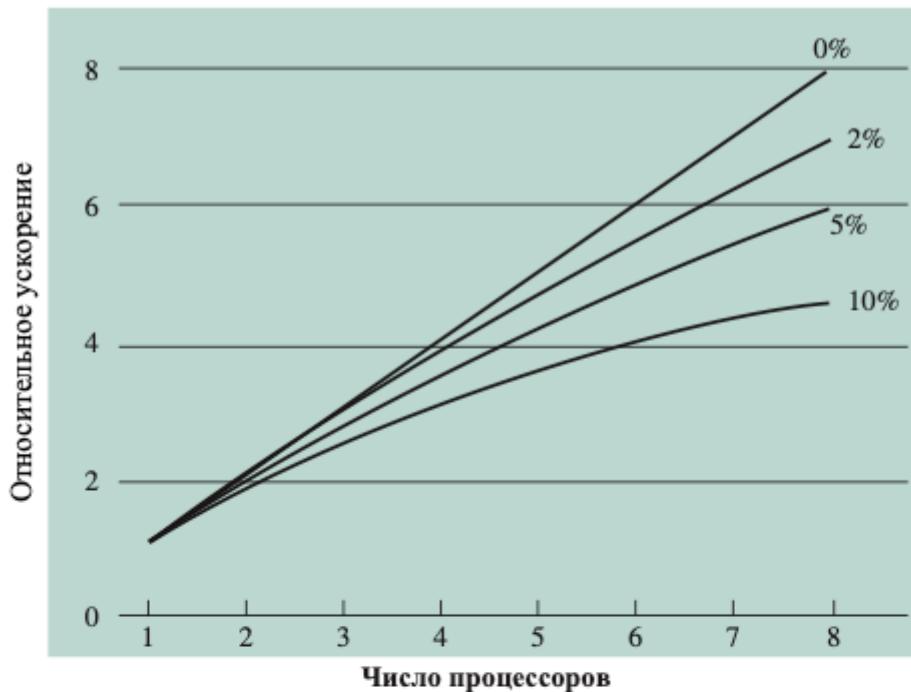
Таким образом видно, что

- Когда  $f$  мало, то использование параллельных процессоров малоэффективно
- Когда  $N \rightarrow \infty$ , то ускорение будет ограничен  $\frac{1}{1-f}$ , что приводит к уменьшению эффективности в пересчете на один процессор при использовании большого количества процессоров.

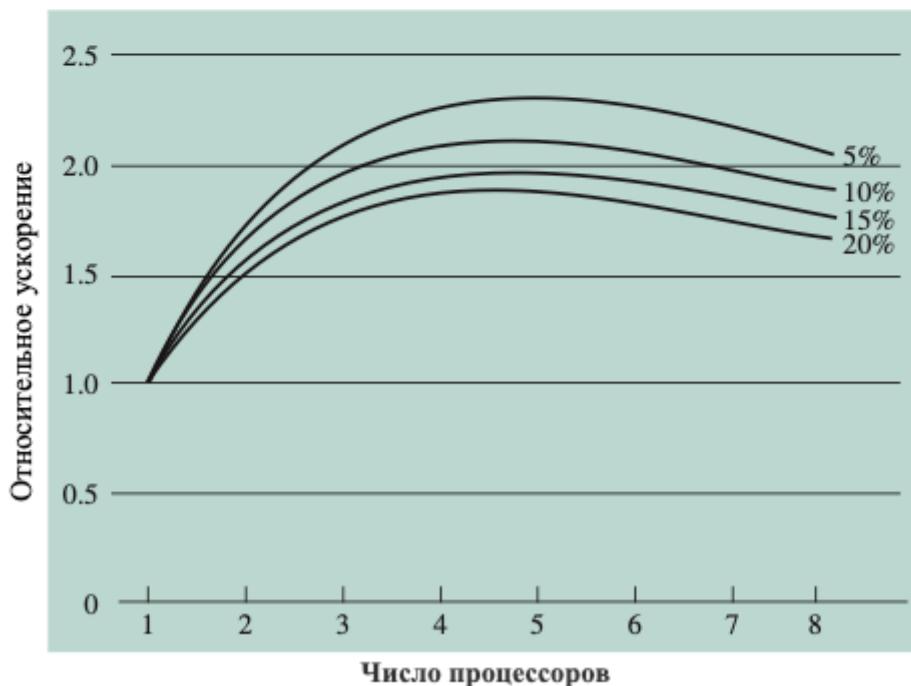


Видно, что даже серьезно увеличивая  $f$  мы никогда не сможем добиться того, что производительность увеличится с ростом количества процессоров. Так как также не стоит забывать о наличии синхронных участков кода, которые не могут быть распараллелены (при переключении потоков и тд)

При количестве процессоров 1000 и  $f = 0.95$  у нас увеличение скорости всего в 20 раз. Отсюда пошли неблокирующие алгоритмы и их популярность.



а) Ускорение для различного количества последовательного кода



б) Ускорение с учетом накладных расходов

## 32. Механизм параллельных вычислений, функции ОС

### Механизм параллельных вычислений

- Однопроцессорные системы -- процессы чередуются
- Процессы выполняются в отдельные кванты времени. Процесс может работать либо ожидать/находиться в заблокированном состоянии

- В каждый конкретный момент времени на процессоре может работать один процесс



- **Многопроцессорные** -- чередуются и перекрываются. Здесь появляются проблемы с конкуренцией за общие ресурсы и голодание
- Для поддержки нескольких процессов система должна быть многопроцессорной
- Здесь можно заметить как процессы борются за ресурсы и некоторых процессы находятся в состоянии голодания
- Процесс 3 занимает все время процессора 1. Это может происходить из-за того, что переродвинуть процесс на другой процессор операция затратная и при том, что процесс 3 делает каку-то активную



### Требуемые функции ОС

- **Отслеживание ресурсов процесса/потока.** Отслеживание ресурсов и при возникновении проблем решать их
- **Распределение и освобождение ресурсов для каждого активного процесса/потока**
- **Защита ресурсов процесса/потока от непреднамеренного воздействия других процессов/потоков.** У потоков одно адресное пространство, что нужно ОС минимизировать воздействие
- **Независимость результата процесса/потока от скорости его выполнения и скорости других процессов/потоков.** Диспетчеризация должна быть равноправной.

## 33. Проблемы параллельного выполнения:

**Взаимоисключения, взаимоблокировки, голодания.**

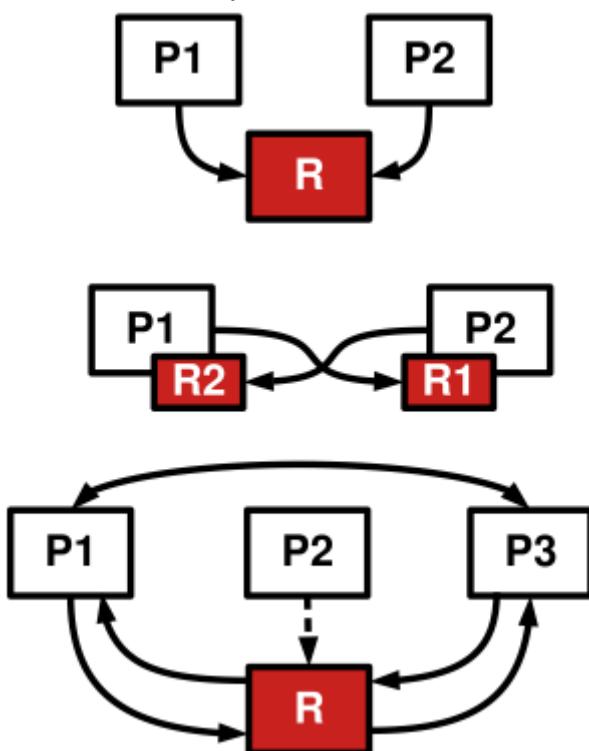
**Требования к взаимным исключениям. Уровни взаимодействия процессов и потоков**

### Проблемы параллельного выполнения

- **Взаимоисключения (mutual exclusion)** -- требование, чтобы, когда один процесс находится в критическом участке, который получает доступ к общим ресурсам,

никакой другой процесс не мог находиться в критическом участке, который обращается к любому из этих общих ресурсов.

- **Взаимоблокировки (deadlock)** -- ситуация, когда два и более процессов не в состоянии работать, поскольку каждый из процессов ожидает выполнения некоторого действия другим процессом
- **livelock** -- состояние, при котором два или более процессов постоянно изменяют свое состояние в ответ на изменения в другом процессе (или процессах) без какой-либо полезной работы. Это похоже на взаимоблокировку тем, что при этом отсутствует какой-либо прогресс, но отличается тем, что ни один процесс не заблокирован и не ждет чего-либо
- **Голодание (starvation)** -- Ситуация, когда запуск процесса пропускается планировщиком бесконечное количество раз, хотя процесс готов к работе, он никогда не выберется



### Требования к взаимным исключениям

- Взаимоисключения должны осуществляться в принудительном порядке. В любой момент времени из всех процессов, имеющих критический участок одного и того же ресурса или общего объекта, в этом участке может находиться всего один процесс
- Процесс, завершающий работу в некритическом участке, не должен влиять на другие процессы. Процесс без блокировки не может менять данные процесса с блокировкой
- Не должна возникать ситуация взаимоблокировок или голодания
- Когда в критическом участке нет ни одного процесса, то любой процесс, запросивший доступ к нему должен немедленно его получить
- Нет предположений о количестве процессов или их относительных скоростях работы. При этом NUMA архитектура может включать процессы с разной

частотой.

- Процесс остается в критическом участке в течении ограниченного количества времени

## Аппаратная поддержка взаимных исключений

- Если система однопроцессорная, то поможет запрет прерываний

```
while (true)
{
    /* Запрет прерываний */;
    /* Критический участок */;
    /* Разрешение прерываний */;
    /* Остальной код */;
}
```

Поскольку критический раздел не может быть прерван, то это гарантирует взаимоисключение. Однако, эффективность работы может снизиться так как мы не сможем чередовать программы и также мы можем потерять важное прерывание.

- Многопроцессорные системы

Так как обращение к ячейки памяти атомарно, то проектировщики процессоров смогли воспользоваться этим и придумать машинные команды, которые также атомарны и помогают обеспечить взаимоисключение

На основе таких атомарных операциях создан один из самых простых способов обеспечения взаимоисключения -- **спин лок**

**Спинлок** (spinlock) — это механизм синхронизации, который используется для защиты критической секции кода в многопоточных программах. Основной принцип спинлока заключается в том, что поток **постоянно проверяет доступность ресурса в цикле (“спинит”), пока не получит доступ к нему.**

**Ключевые особенности спинлока:**

### 1. Без блокировки:

- Поток, ожидающий доступа к ресурсу, не засыпает, а активно проверяет доступность ресурса в цикле.
- Это полезно для кратковременных операций, где накладные расходы на блокировку/разблокировку потока (например, через mutex) слишком велики.

### 2. Минимальные задержки:

- Спинлок эффективен, если ресурс освобождается быстро.

### 3. Высокая нагрузка на процессор:

- Поток активно потребляет процессорное время, что может быть неэффективно для долгих ожиданий.

## Взаимодействие потоков/процессов

### Процессы не осведомлены друг о друге

- Конкуренция за ресурсы

- Один процесс не зависит от действий другого
- Возможно влияние на время работы одного процесса на другой
- **Проблемы:** взаимоисключения, взаимоблокировки, голодания

### Процессы косвенно осведомлены один о другом

- Сотрудничество с использованием общих ресурсов -- к примеру есть один файл или одна область памяти к которой обращаются процессы
- Результат работы одного процесса может зависеть от информации, полученной от других процессов
- Возможно влияние одного процесса на время другого
- **Проблемы:** взаимоисключения, взаимоблокировки, голодания, связь данных (как раз возникает из-за того, что несколько процессов могут при одновременном доступе испортить данные)

### Непосредственная осведомленность друг о друге

- Сотрудничество с использованием связи. Связь обеспечивает возможность синхронизации или координации действий процессов
- Результат работы одного процесса может зависеть от информации, полученной от других процессов
- Возможно влияние одного процесса на время другого
- **Проблемы:** взаимоисключения, голодания

## 34. Примитивы синхронизации ОС. Предназначение примитивов синхронизации

---

**Примитивы синхронизации** -- это инструменты, предоставляемые операционной системой (или библиотеками), для упорядочивания доступа к общим ресурсам (например, памяти, файлам, устройствам) в многозадачных системах.

Каждый вид примитива синхронизации создавался под конкретную задачу. Реализация примитивов синхронизации зависит от системы, в целом они похожи, но могут быть небольшие детали, которые отличаются. Именно поэтому решения для кода ядра не так легко портируются с одной ОС на другую в отличии от пользовательских программ, где есть стандарты тот же самый POSIX.

**Семафор** -- захват и освобождение множественного ресурса. Целочисленное значение, используемое для передачи сигналов между процессорами. Над семафором можно применить три операции: инициализация, декрмент и инкремент значения. Операция уменьшения может привести к блокировке процесса, а увеличения -- к разблокировке

**Бинарный семафор** -- используется для захвата или освобождения одного ресурса (может принимать значения 0 или 1)

**Мьютекс** -- блокировка и освобождение ресурса единственным процессом/потоком. Процесс, блокирующий мьютекс, должен сам разблокировать его.

**Условные переменные** -- Блокировка до выполнения какого-либо условия.

**Блокировки чтения/записи (rw-lock)** -- основная задача сделать так, чтобы много потоков смогли читать одновременно, но только один поток смог писать что-то. Используется двойная блокировка для чтения и записи

**Спин-локи** -- на уровне ядра эквивалентны одной из разновидности мьютексов. Основной принцип спинлока заключается в том, что поток **постоянно проверяет доступность ресурса в цикле**. (см 33 вопрос)

**Мониторы** -- конструкции япов, которые скрывают низкоуровневые примитивы синхронизации. Это высокоуровневый примитив синхронизации, который представляет собой абстракцию для управления доступом к общим ресурсам в многозадачной среде. Монитор инкапсулирует данные и методы, обеспечивая автоматическое управление синхронизацией. Он гарантирует, что только один поток может выполнять код внутри монитора в данный момент времени, что помогает избежать гонок данных. Может иметь очередь процессов, ожидающих доступа к монитору.

**Флаги событий** -- связывание условий продолжения выполнения с одним или несколькими флагами (битами блокирующей переменной)

**Почтовые ящики** -- средство обмена информацией между двумя процессами, которое может быть использовано для синхронизации

## 35. Примитивы синхронизации ОС. Семафоры и мьютессы

---

### Семафор (Дейкстра)

Обычно семафоры называют counting semaphore так как ресурсов может быть несколько.

Дейкстра работал над вопросами параллельных вычислений и рассматривал разработку ОС как построение множества сотрудничающих последовательных процессов и создание эффективных и надежных механизмов такого сотрудничества.

Фундаментальный принцип заключается в том, что два и более процессов могут сотрудничать посредством простых сигналов. Для сигнализации используются переменные -- семафоры. Для передачи сигнала через семафор `s` используется примитив `semSignal(s)`, а для получения -- `semWait(s)`

## Info

Либо в оригинале вместо `wait` используется `P` (`proberen`) от слова проверка и вместо `signal` -- `V` (`verhogen`) от слова увеличение

- Семафор может быть инициализирован целочисленным значением
- Операция `semWait` уменьшает значение семафора. Если это значение становится меньше нуля, то процесс, который выполнил эту операцию блокируется
- Операция `semSignal` увеличивает значение семафора. Если это значение меньше или равно нулю, заблокированный операцией `semWait` процесс (если такой есть) деблокируется

Других операций по получению значения семафора или его изменения нет

Для хранения процессов, ожидающих блокировку, используется очередь:

- **Сильный семафор** -- используется политика FIFO для выбора процесса, который зайдет в критическую секцию
- **Слабый семафор** -- порядок извлечения процессов не определен
- **Приоритетная выборка процессов**

### Описание работы семафора

В начале работы семафор имеет значение нуль или некоторое положительное значение. Если значение положительное, то оно равно количеству процессов, которые могут вызвать операцию получения сигнала и немедленно продолжить выполнение. Если же значение равно нулю (полученное либо при инициализации, либо потому, что количество процессов, равное первоначальному значению семафора, вызвано операцию ожидания), очередной ожидающий процесс блокируется, а значение семафора становится отрицательным. Каждое последующее ожидание уменьшает значение семафора, так что оно имеет отрицательное значение, по модулю равное числу процессов, ожидающих разблокирования. Когда значение семафора отрицательное, каждый сигнал разблокирует один из ожидающих процессов.

### Пример определения примитивов семафора

```

struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* Процесс помещается в s.queue */;
        /* Блокировка данного процесса */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count<= 0) {
        /* Удаление процесса P из s.queue */;
        /* Перемещение процесса P в список готовности */;
    }
}

```

На программном уровне реализацию семафора можно сделать с помощью compare and swap алгоритма для многопроцессорной системы и для однопроцессорной с помощью разрешения и запрета прерываний

<pre> semWait(s) {     <b>while</b> (compare_and_swap(s.flag, 0 , 1)            == 1)         /* Ничего не делать */;     s.count--;     <b>if</b> (s.count &lt; 0)     {         /* Поместить данный            процесс в s.queue*/;         /* Блокировать этот процесс            (следует также установить            s.flag равным 0) */;     }     s.flag = 0; }  semSignal(s) {     <b>while</b> (compare_and_swap(s.flag, 0 , 1)            == 1)         /* Ничего не делать */;     s.count++;     <b>if</b> (s.count&lt;= 0)     {         /* Убрать процесс            P из s.queue */;         /* Поместить процесс P в список            готовых к выполнению            процессов*/;     }     s.flag = 0; } </pre>	<pre> semWait(s) {     inhibit interrupts;     s.count--;     <b>if</b> (s.count &lt; 0)     {         /* Поместить данный            процесс в s.queue*/;         /* Блокировать этот процесс            и разрешить прерывания */;     }     <b>else</b>         allow interrupts; }  semSignal(s) {     inhibit interrupts;     s.count++;     <b>if</b> (s.count&lt;= 0)     {         /* Убрать процесс            P из s.queue */;         /* Поместить процесс P в список            готовых к выполнению            процессов*/;     }     allow interrupts; } </pre>
--	---

а) Команда сравнения и присваивания

б) Прерывания

**Рис. 5.17.** Две возможные реализации семафоров

## Бинарный семафор

Ограниченнная версия семафора, где значение семафора может быть либо 0 либо

1. То есть мы допускаем только один процесс к ресурсу

- Может быть инициализирован значениями 0 или 1
  - `semWaitB` проверяет значение семафора. Если == 0, то процесс блокируется. Если == 1, то процесс продолжает выполняться, получая блокировку, и значение семафора меняется на 0
  - `semSignalB` проверяет есть ли процесс блокированный этим семафором (равен ли семафор нулю). Если есть, то блокированный процесс деблокируется. Если заблокированных процессов нет, значение семафора устанавливается 1
- Бинарный семафор тесно связан с концепций взаимоисключения так как он также обеспечивает доступ только одного процесса к ресурсу

```
struct binary_semaphore {  
    enum {zero, one} value;  
    queueType queue;  
};  
void semWaitB(binary_semaphore s)  
{  
    if (s.value == one)  
        s.value = zero;  
    else {  
        /* Процесс помещается в s.queue */;  
        /* Блокировка данного процесса */;  
    }  
}  
void semSignalB(semaphore s)  
{  
    if (s.queue is empty())  
        s.value = one;  
    else {  
        /* Удаление процесса P из s.queue */;  
        /* Перемещение процесса P в список готовности */;  
    }  
}
```

## Мьютексы (mutual exclusion)

**Мьютекс** -- это механизм синхронизации в операционных системах, который используется для предотвращения одновременного доступа нескольких потоков или процессов к общему ресурсу, таким как файл, переменная или память. Мьютекс гарантирует, что в любой момент только **один поток** может владеть ресурсом.

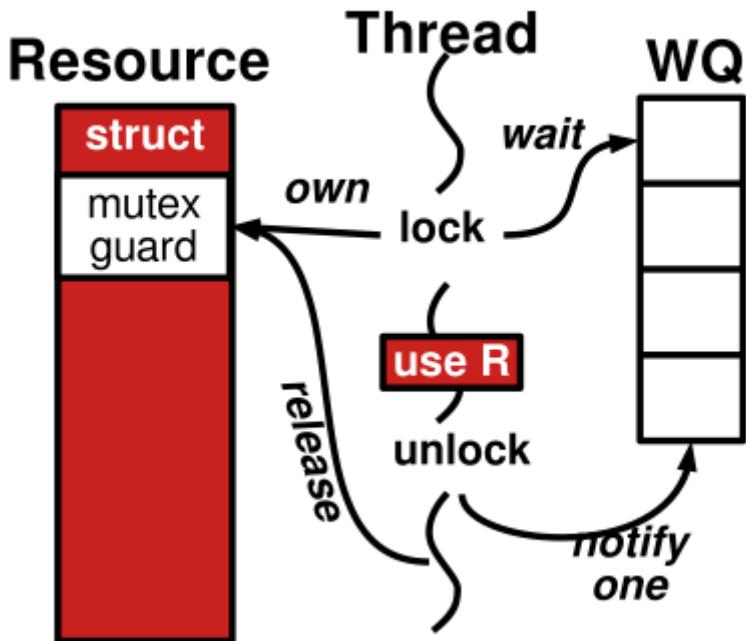
Имеется очень много реализаций: блокирующие, спин, адаптивные, фьютексы и так далее

## Принцип работы

Мьютекс использует программный флаг. Разработчики стараются укладывать мьютексы в одно целое число (оно может быть большим, но зато одно). При захвате данных мьютекс устанавливается в состояние блокировки (обычно 0), которое предотвращает остальные попытки использования. Ключевое отличие мьютекс от бинарного

семафора заключается в том, что процесс блокирующий мьютекс (устанавливающий его значение в 0) должен быть тем же, что и разблокирующий его. То есть один процесс сам блокирует и деблокирует, а в бинарном семфоре один процесс блокирует, но разблокировать должен другой. В мьютексе семантикой установлено, что если unlock выполняется не тем же процессов, что произвел lock, то может показаться ошибка

Может быть блокриующим (как на картинке, где мы помещаем процесс в очередь, если мьютекс занят) либо неблокриующим, когда мы в цикле постоянно смотрим свободен ли мьютекс (такой подход хороший, если мы знаем, что блокировка будет короткой)



**Захват мьютекса должен быть коротким**

**Проблема priority inversion**

- Проблема, которая возникает в многозадачных операционных системах, когда высокоприоритетный поток (или процесс) оказывается заблокированным из-за того, что низкоприоритетный поток удерживает ресурс, необходимый для выполнения высокоприоритетного потока.
- При этом среднеприоритетный поток, который не зависит от ресурса, может продолжать выполнение, что приводит к нежелательной задержке выполнения высокоприоритетного потока.

#### • Решение проблемы

##### Наследование приоритета

- Поток с низким приоритетом, удерживающий ресурс, временно получает приоритет ожидающего потока с более высоким приоритетом.
- После освобождения ресурса поток возвращается к своему изначальному приоритету.

## 36. Примитивы синхронизации ОС. Условные переменные и rw-locks

---

### Условные переменные

#### Задача потребителя/производителя

Имеется один или несколько производителей, генерирующих данные некоторого типа (записи, символы и т.п.) и помещающих их в буфер, а также единственный потребитель, который извлекает помещенные в буфер элементы по одному. Требуется защитить систему от перекрытия операций с буфером, т.е. обеспечить, чтобы одновременно получить доступ к буферу мог только один процесс (производитель или потребитель). Проблема заключается в том, чтобы гарантировать, что производитель не будет пытаться добавить данные в буфер, если он заполнен, и что потребитель не будет пытаться удалить данные из пустого буфера.

Условные переменные используются для того, чтобы один поток смог ожидать наступления какого-либо события от другого потока.

Общая семантика сводится к трем методам:

- `wait` -- Поток освобождает связанный мьютекс и переходит в состояние ожидания (спит), пока другой поток не уведомит его.
- `signal` -- Будит один из потоков, который ожидал на данной условной переменной.
- `broadcast` -- Будит все потоки, ожидающие на условной переменной.

Перед вызовом `wait` поток должен захватить мьютекс, чтобы избежать гонок. Когда поток ожидает по какой-то переменной мьютекс может быть снят, а при пробуждении опять захвачен.

#### Для решения задачи producer/consumer

- Producer захватывает лок и если место в буфере нет, то ждет по переменной `spaceAvailable`
- Как только от Consumer мы получаем сигнал, что место есть, то мы пишем по индексу `putIndex` букву и подаем сигнал Consumer, что появились данные + отпускаем лок
- Consumer в свою очередь захватывает лок и ждем пока не придет сигнал о наличии данных
- Если данные были или пришел сигнал, то мы читаем значение из буфера по индексу `getIndex` и подаем сигнал, что у нас есть место в буфере. Отпускаем лок
- While при ожидании места в буфере или данных нужен из-за возможного внезапного отпускания блокировки по условной переменной, поэтому это еще один

## уровень защиты

```
void produce(char c) {
    lock_acquire(&lock);
    while (count == SIZE) {
        cond_wait(&spaceAvailable, &lock);
    }
    count++;
    buffer[putIndex] = c;
    putIndex++;
    if (putIndex == SIZE) {
        putIndex = 0;
    }
    cond_signal(&dataAvailable, &lock);
    lock_release(&lock);
}
```

```
char consume() {
    char c;
    lock_acquire(&lock);
    while (count == 0) {
        cond_wait(&dataAvailable, &lock);
    }
    count--;
    c = buffer[getIndex];
    getIndex++;
    if (getIndex == SIZE) getIndex = 0;
    cond_signal(&spaceAvailable, &lock);
    lock_release(&lock);
    return c;
}
```

## rwlocks

Имеются данные, совместно используемые рядом процессов. Данные могут находиться в файле, в блоке основной памяти или даже в регистрах процессора. Имеется несколько процессов, которые только читают эти данные (читатели), и несколько других, которые только записывают данные (писатели). При этом должны удовлетворяться следующие условия:

- Любое число читателей могут одновременно читать файл
- Записывать информацию в файл в конкретный момент времени может только один писатель
- Когда писатель записывает, ни один читатель не может читать файл  
Таким образом, читатели представляют собой процессы, которые не обязаны быть взаимоисключающими, в то время как писатели являются процессами, которые должны исключить выполнение любых других процессов, как читателей, так и писателей.
- Когда читатели читают, они захватывают readlock, количество одновременных читателей содержится в rwlock
- Когда писатель требует записи -- он устанавливает требование записи и ожидает rwlock
- rwlock ждет освобождения readlock
- Оповащает писателей
  - Один захватывает writelock (читатели не могут читать )
- Оповещает читателей
  - Захватывает readlock ( писатели должны требовать )

Также есть проблема "громящего стада," когда у нас в очереди скапливается очень много потоков и при освобождении лока они все начнут конкурировать между собой на захват этого лока.

Также есть реализации когда мы отдаем приоритет писателям, то есть читатели не смогут зайти в ресурс, если у нас есть хоть какие-то писатели.

## 37. Примитивы синхронизации ОС. Мониторы, флаги событий, передача сообщений

## **Мониторы**

**Мониторы** -- конструкции япов, которые скрывают низкоуровневые примитивы синхронизации. Это высокоуровневый примитив синхронизации, который представляет собой абстракцию для управления доступом к общим ресурсам в многозадачной среде. Монитор инкапсулирует данные и методы, обеспечивая автоматическое управление синхронизацией. Он гарантирует, что только один поток может выполнять код внутри монитора в данный момент времени, что помогает избежать гонок данных. Может иметь очередь процессов, ожидающих доступа к монитору.

- Набор процедур, выполняющих операции над общими данными
- Каждая процедура подразумевает захват блокировки общих данных
- Локальные переменные используются только внутри монитор
- Для ожидания и оповещения используются условные переменные
- Parallel Pascal, Java
- Реализуются высокоуровнево, программисту не требуется возиться с захватом/освобождением или ожиданием/нотификацией

### **Основные характеристики монитора**

- Локальные переменные монитора доступны только его процедурам. Внешние процедуры к этим переменным доступа получить не могут
- Процесс входит в монитор путем вызова его процедур
- В любой момент времени в мониторе может выполняться только один процесс, остальные вынуждены ждать.

## **Event flags (флаги событий)**

Набор (битов) флагов в ожидании событий

Флаги событий позволяют одному потоку или процессу ожидать, пока другой поток или процесс установит флаг, чтобы продолжить выполнение. Этот механизм часто используется для организации взаимодействия между потоками в реальном времени.

### **Операции**

- Установка флага
- Сброс флага
- Ожидание флага
- Ожидание всех флагов

Реализации: VMS, Python

## **Message passing (передача сообщений)**

При взаимодействии процессов между собой должны удовлетворяться два фундаментальных требования: синхронизации и коммуникации. Процессы должны быть синхронизированы, с тем чтобы обеспечить выполнение взаимных исключений; сотрудничающие процессы должны иметь возможность обмениваться информацией. Одним из подходов к обеспечению обеих указанных функций является передача сообщений.

Важным достоинством передачи сообщений является ее пригодность для реализации как в одно и многопроцессорных системах с общей памятью, так и в распределенных системах.

- Определено две операции send(получатель, сообщение) и receive(отправитель, сообщение)

### **Характеристика системы передач сообщений**

## **Синхронизация**

**Отправление** может быть

- Блокирующими (мы ждем пока сообщение будет отправлен и доставлено)
- Неблокирующими (отправляем сообщение и идем дальше)

### **Получение**

- Блокирующее (мы ждем пока не получим сообщение)
- Неблокирующее (если сообщения нет, то мы продолжаем работу и не пытаемся получить какое-то сообщение)
- Проверка доставки (позволить процессу, который ждет сообщения, проверить нет ли сообщения, которое должно быть отправлено ему)

### **Самые распространенные комбинации**

- **Оба блокирующие.** Блокируются до тех пор пока сообщение не будет доставлено по назначению. Такое еще называют **рандеву** и обеспечивает тесную синхронизацию
- **Неблок. отправление, блок. получение.** Самая популярная реализация и позволяет отправителю с максимальной скоростью отправить все сообщения, а получатель будет ожидать сообщения
- **Оба неблокирующие**

## **Адресация**

**Для обоих процессов есть**

- **Прямая адресация.** Для **отправления** мы можем явно указать id процесса-получателя. Для **получается** есть два пути. Первый заключается в явном указании id процесса-отправителя, такой подход эффективен, но иногда сложно узнать от кого должно прийти сообщение. Второй путь заключается в неявной адресации, параметр отправитель у процесса-получателя возвращается после получения сообщения.
- **Косвенная адресация.** Предполагает, что процессы не напрямую обмениваются сообщениями а через посредника, в который приходят сообщения (mailbox). Таким образом, для связи между двумя процессами один из них посыпает сообщение в соответствующий почтовый ящик, из которого его заберет второй процесс. При такой схеме работы отношения между отправителями могут быть любыми -- 1:1,

1:N, M:1, M:N

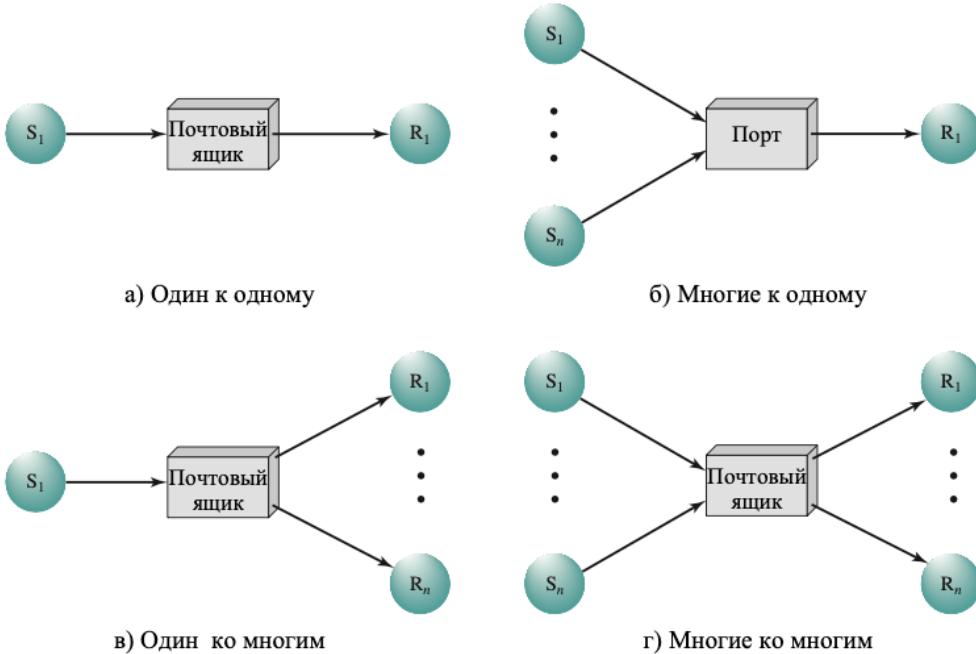


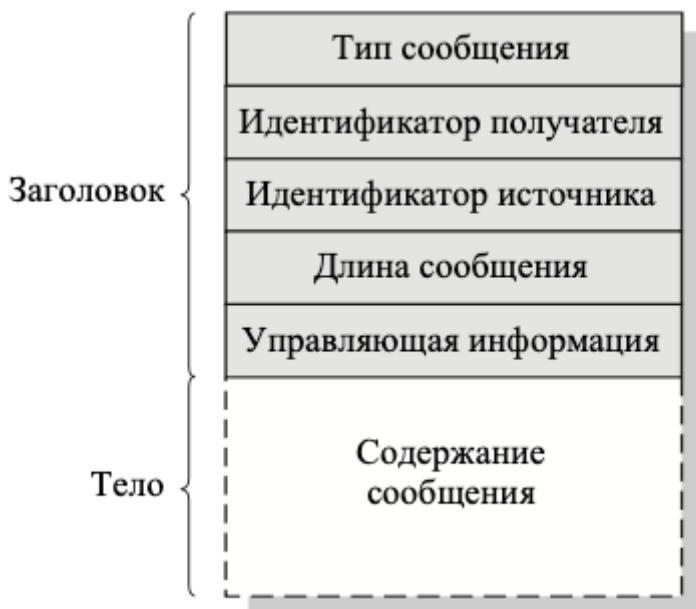
Рис. 5.21. Косвенная связь между процессами

- **Статическая связь между процессами.** Например, связи 1:1 и M:1.
- **Динамическая связь.** С помощью примитивов connect и disconnect можно устанавливать связь между процессами
- **Владение.** В случае порта он как правило принадлежит процессу-получателю (и умирает вместе с ним)

## Формат сообщения

Формат сообщения зависит от цели и от того где работает система.

- Маленькое сообщение фиксированной длины
- Файл или ссылка на файл
- **Сообщение переменной длины**



## Принцип работы очереди

FIFO

Приоритетная

## 38. Примитивы синхронизации ОС. Неблокирующие примитивы синхронизации и неблокирующие структуры данных

---

Неблокирующие примитивы и структуры данных используются для работы с параллельными процессами или потоками без необходимости применения блокировок (например, мьютексов). Их цель — обеспечить эффективный доступ к общим данным, минимизируя накладные расходы и избегая проблем с блокировками

**Есть три уровня неблокирующего доступа**

- **lock-free**

- Гарантируется, что хотя бы один поток сможет завершиться за конечное время
- Если несколько потоков конкурируют за ресурс, все они могут продолжать попытки, но ни один не будет блокирован.
- Пример: очередь, в которой добавление и удаление элементов выполняется с помощью CAS (compare and swap) в цикле.
- Некоторые потоки могут начать голодать
- Исключает взаимные блокировки
- К примеру есть счетчик, который инкрементируется с помощью CAS. То есть ни один поток не блокирует эту секцию для инкремента, но только один поток сможет инкрементировать счетчик, остальные будут заново пытаться

- **wait-free**

- Каждый поток может завершить свою операцию за фиксированное количество шагов, независимо от действий других потоков.
- Полностью отсутствует голодание
- Самый строгий уровень
- Сложно в реализации
- все wait-free алгоритмы также lock-free

- **obstruction-free**

- Гарантирует успешное завершение операции потока, если он работает в **отсутствие конкуренции** (то есть другие потоки не вмешиваются в его выполнение).
- Операция может быть прервана другим потоком, но если потоку удастся продолжить работу без помех, она завершится.

- Нет гарантий, что поток завершит свою операцию, если конкуренция существует.
- все lock-free алгоритмы также obstruction-free

Неблокирующие структуры -- списки, деревья, очереди ... over 9000

## 39. Управление памятью, основные определения и требования к организации

---

В многозадачных системах пользовательская часть памяти должны быть распределена для размещения нескольких процессов. Эта задача распределения выполняется ОС динамически и называется **управление памятью**

### Определения

**Основная память (main memory)** -- область, где процессор может исполнять программы

**Вторичная память** -- область памяти, где программы и данные могут храниться, в том числе и во время исполнения

**Кадр (frame)** -- блок фиксированного размера основной памяти

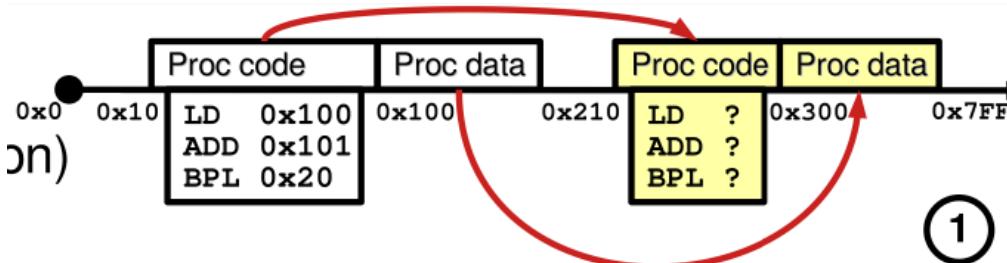
**Страница (page)** -- блок данных фиксированной длины, находящийся во вторичной памяти (такой как диск). Страница может быть временно скопирована в кадр основной памяти

**Сегмент** -- блок данных переменной длины, находящийся во вторичной памяти. В доступную область основной памяти может быть скопирован сегмент полностью (сегментация). Возможно также разделение сегмента на страницы и копирование страниц в кадры основной памяти по отдельности

Сейчас применяется сегментно-страничная организация памяти. При этом страница -- характеристика аппаратного обеспечения, а сегмент -- более программная характеристика, которая определяет права рабочей копии процесса, а также какие-либо дополнительные характеристики.

### Требования к организации

#### Перемещение (relocation)



Заранее мы не знаем как программы будут резидентно находиться в основной памяти. Кроме того мы должны максимизировать загрузку процессора, а для этого может потребоваться возможность загрузки и выгрузки активных процессов из основной

памяти. Условие загрузки процесса на тот же участок основной памяти было бы сильным ограничением. Поэтому необходимо, чтобы программа была **переносимой**.

Очевидно, что операционной системе необходимо знать местоположение управляющей информации процесса и стека выполнения, а также точки входа для начала выполнения процесса. Поскольку управлением памятью занимается операционная система и она же размещает процесс в основной памяти, соответствующие адреса она получает ав томатически. Однако программе также нужно обращаться к памяти в самой программе, поэтому указание абсолютных адресов может сломать всю программу после ее переноса. Поэтому к примеру можно указать относительные адреса начиная с адреса счетчика команд

### Защита

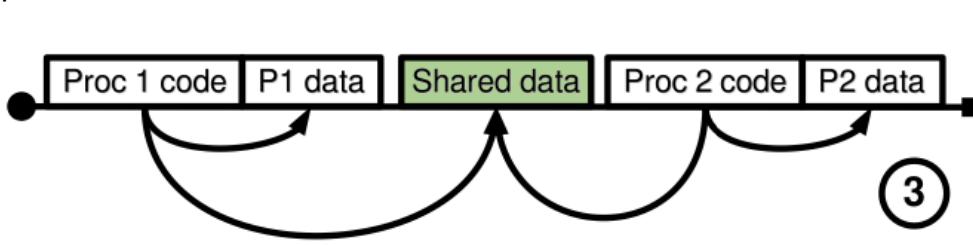
Код одних процессов не должен иметь возможность без разрешения обращаться к памяти другого процесса для чтения или записи. Расположение программы заранее неизвестно и к тому же в некоторых языках адрес может вычисляться динамически во время исполнения.

Требование защиты должно быть обеспечено на уровне аппаратуры, потому что ОС не в состоянии предвидеть все возможные обращения к памяти. Даже если бы ОС могла это сделать, то ей бы пришлось сканировать все программы, находящиеся в основной памяти, что было бы слишком расстачительно касаемо ресурсов процессора

### Совместное использование (time sharing)

Механизм защиты памяти должен обеспечивать возможность нескольким процессам обращаться к одной и той же области памяти. Например, для совместного использования одного кода или данных, система управления памятью должна предоставлять контролируемый доступ к разделяемым областям памяти, не ослабляя при этом защиту.

Нужно для снижения количества библиотек и уменьшить статическое связывание процессов



### Логическая организация

Нам необходимо, чтобы ОС в явном виде могла уметь размещать программы в памяти, брать информацию из вторичной памяти, загружать ее в основную, сбрасывать обратно и так далее. И здесь нам нужен какой-то логический подход, который позволял бы это делать.

К тому же в основном программы представляют из себя набор модулей, которые могут быть неизменными или меняться со временем. Если ОС и аппаратура смогут эффективно работать с такими программами, то:

- Модули могут быть скомпилированы и созданы отдельно друг от друга при этом все ссылки на модули сохраняются
- Разные модули смогут получать разные права
- Возможно применение механизма, обеспечивающего совместное использование модулей разными процессами.

Самый лучший инструмент для реализации такого требования -- сегментация

### Физическая организация

Есть основная -- быстрая, дорогая и не долговременная память и есть вторичная -- медленная, дешевая и долговременная память.

Вторичная память может служить для хранения программ и данных, а основная -- для программ и данных, которые нужны в текущий момент.

В такой двухуровневой системе основной заботой системы становится организация потоков информации между уровнями. Ответственность можно было бы положить на программиста, но это неэффективно потому что:

- Основной памяти может быть недостаточно для программы и данных. В этом случае программисты прибегают к практике известной **как структуры с перекрытием -- оверлеи**, когда программа и данные организованы таким образом, что различные модули могут быть назначены одной и той же области памяти. Основная программа ответственна за перегрузку модулей при необходимости.
- Во многозадачной среде программист при разработке программы не знает, какой объем памяти будет доступен программе и где эта память будет располагаться.

## 40. Фиксированное и динамическое размещение программ в памяти

---

### Фиксированное размещение

Основная память разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера.

Возможно два варианта фиксированного распределения

- **Разделы одинакового размера.**

В этом случае любой процесс, размер которого не прерывает размер раздела, может быть загружен в любой доступный фрейм. Если все разделы заняты то ОС может выгрузить процесс во вторичную память и загрузить другой процесс в основную память.

При использовании такого подхода есть **две трудности**

- Программа может быть слишком велика для размещения в разделе. В это случае необходимо использовать **оверлеи**, чтобы в любой момент времени программе

необходим был один раздел памяти. И программа сама должна загружать недостающие модули программы в память

- **Неэффективное использование памяти.** Есть программы, которые будут меньше размера раздела, однако программа все равно будет занимать весь раздел. Такой феномен называется **внутренней фрагментацией**

**Общий минус фиксированного размещения заключается** в том, что мы ограничиваем количество процессов, которые одновременно могут выполняться.

- **Разделы разного размера**

В таком случае можно будет обойтись без оверлеев и уменьшить внутреннюю фрагментацию

- Поскольку размеры разделов устанавливаются заранее, в момент генерации системы, небольшие процессы приводят к неэффективному использованию памяти. В средах, в которых заранее известны потребности в памяти всех задач, применение описанной схемы может быть оправдано, но в большинстве случаев эффективность этой технологии крайне низка.



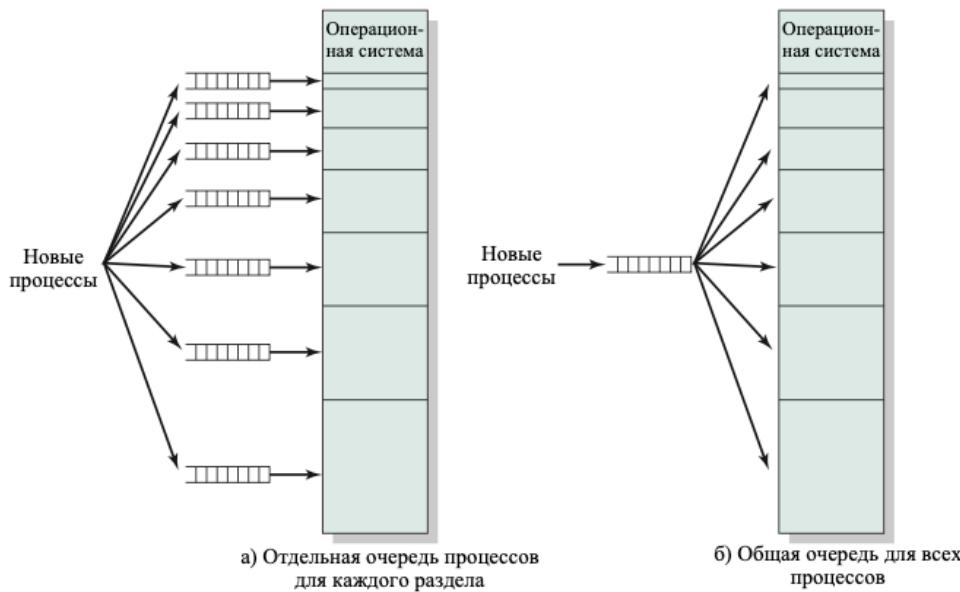
а) Разделы одинакового размера



б) Разделы разных размеров

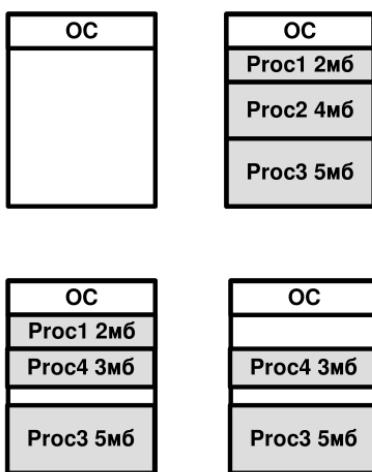
Есть несколько алгоритмов размещения процессов в памяти именно для разделов разного размера (для разделов одинакового размера задача тривиальная так как не важно куда зашружать процесс)

- **У каждого раздела есть своя очередь.** Тогда каждый процесс будет размещаться в таком разделе, чтобы он полностью вместил этот процесс. Плюс в том, что процессы могут быть разделены между разделами так, чтобы минимизировать внутреннюю фрагментацию. Минус в том, что некоторые разделы могут пустовать и мы будем неэффективно использовать память
- **Одна очередь для всех процессов.** В момент когда требуется загрузить процесс в основную память для этого выбирается наименьший раздел. Если все разделы заняты то стоит выгрузить какой-то процесс и предпочтение стоит отдать разделу, который сможет вместить новый процесс целиком



### Динамические размещение

Разделы создаются динамически. Каждый процесс загружается в раздел строго необходимого размера.



Такой подход хорошо начинает работу, но плохо продолжает, что в конечном счете приводит к наличию мелких дыр в памяти. Со временем память становится все более и более фрагментированной и снижается эффективность ее использования. Это явление называется внешняя фрагментация.

Решение такой проблемы может быть процесс **уплотнения памяти**, при котором процессы перемещаются таким образом, чтобы они занимали смежные области памяти. Однако при этом расходуется время и к тому же надо обеспечить переносимость программ

Подходит для загрузки драйверов так как при запуске системы они сразу могли размещаться в памяти и не требовали перемещения. Однако если требовалось поменять драйвер то чтобы это сделать надо перезагрузить систему, чтобы не мучаться с уплотнением памяти.

## Есть три алгоритма размещения

- **Метод наилучшего подходящего.** Выбирается блок размер максимально близкого к размеру процесса
- **Первого подходящего.** Выбирает первый достаточный по размеру блок для вмешения процесса. Просмотр блоков начинается с нача
- **Следующего подходящего.** Похож на первого подходящего, однако просмотр блоков начинается с места последнего выделения.

Наилучшие результаты дает метод первого подходящего, метод след подхоядщего дает результаты чуть хуже так как конец памяти начинает сильно фрагментироваться, а метод наилучшего подхоядщего самый плохой так как происходит сильная фрагментация памяти везде.

Также есть система двойников для улучшения динамического размещения процессов и снижения фрагментации памяти. Эта система строится так

- Общий объем памяти при запросе на выделение делится на блоки равные степеням двойки (1МБ -> 2 блока по 512Кб -> 2 блока по 256КБ и тд)
  - Каждый блок имеет бадди или соседа, с которым он может объединиться при условии что оба блока свободны и одинакового размера
  - При запросе памяти происходит деление блоков до того момента пока не найдем блок минимального размера, который смог бы сместить наш процесс.
  - При освобождении происходит проверка на бадди и при этом этот процесс может идти рекурсивно пока возможно объединение
- Систему двойников можно представить в виде бинарного дерева, что упростит работу с таким алгоритмом

## Пример работы

Исходный блок 1 Мбайт	1М				
Запрос 100 Кбайт	A = 128K	128K	256K		512K
Запрос 240 Кбайт	A = 128K	128K	B = 256K		512K
Запрос 64 Кбайт	A = 128K	C = 64K	64K	B = 256K	512K
Запрос 256 Кбайт	A = 128K	C = 64K	64K	B = 256K	D = 256K
Освобождение B	A = 128K	C = 64K	64K	256K	D = 256K
Освобождение A	128K	C = 64K	64K	256K	D = 256K
Запрос 75 Кбайт	E = 128K	C = 64K	64K	256K	D = 256K
Освобождение C	E = 128K	128K		256K	D = 256K
Освобождение E			512K		D = 256K
Освобождение D				1M	

## 41. Модели аппаратного перемещения программ

Перемещать программы посредством ОС долго, поэтому необходимо поместить в аппаратуру какой-то элемент, который помогал бы ОС средствами аппаратуры.

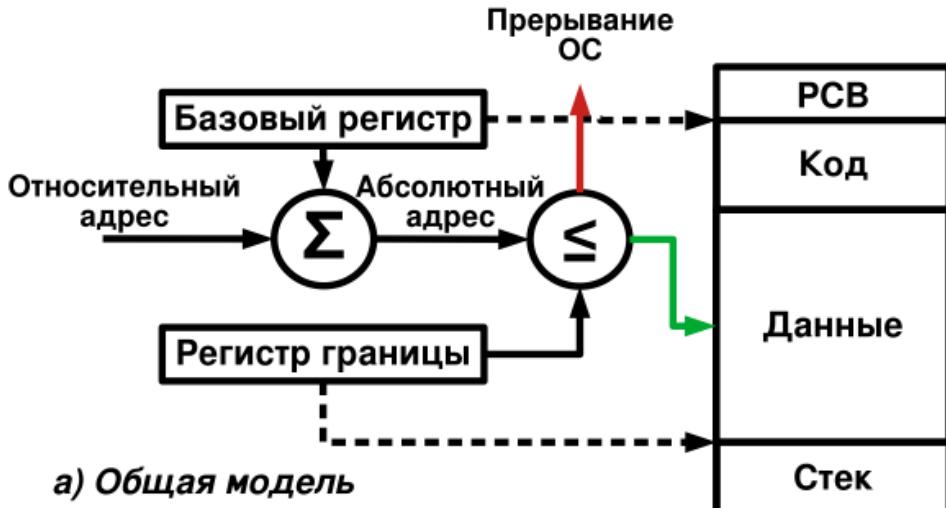
**Следует различать типы адресов**

- Логический адрес -- ссылка на ячейку памяти, не зависящую от текущего расположения данных в памяти, перед тем как получить доступ к такой ячейки необходимо провести процесс трансляции
- Относительный адрес -- частный случай логического, когда адрес определяется положением относительно некоторой известной точки
- Физический адрес -- действительное расположение нужной ячейки памяти

Расположение программы можно задать при помощи специальных регистров

- Базовый регистр -- в нем находится начало программы

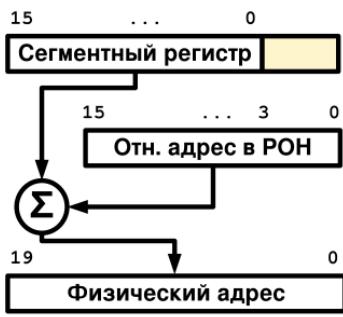
- Регистр границы -- адрес последней ячейки памяти программы



У нас есть относительный адрес, который складывается с базовым регистром. Так мы получаем абсолютный адрес. Также нам надо произвести сравнение не вышли ли мы за пределы регистра границ. Если нет то все окей, если да то генерируется прерывание. Таким образом выше базового регистра мы не сможем никуда вылезти, а ниже регистра границ тоже не можем спасибо компаратору.

### Адресация в процессоре 8086

Процессор 8086 задумывался как 16-разрядный, поэтому по сути он мог адресовать только 64Кб памяти, что очень мало. Поэтому была предложена следующая адресация, которая позволяла адресовать существенно больший объем памяти.



*б) Модель 8086*

Пользователю был доступен сегментный 16-разрядный регистр. Туда можно загрузить какие-то данные и эти данные определят места, где начинаются сегменты. Также был регистр для смещения (так как он был также 16-разрядным то мы могли указать смещение внутри 64Кб сегмента). Сдвинутый на 4 разряда влево сегментный адрес прибавлялся к относительному регистру и получался 20-разрядный физический адрес. В такое архитектуре было 4 сегмента (сегмент кода, данных, стека и доп. сегмент) по 64Кб, получается удалось расширить адресное пространство программы с 64Кб до 1Мб.

## 42. Простой страничный поход и простая сегментная организация.

## Страницочный подход

Предположим, что основная память разделена на одинаковые блоки относительно небольшого размера (для удобства берут размер кратен степени 2). Тогда блоки процесса известные как **страницы** могут быть связаны со свободными блоками памяти, известные как **кадры**. **Каждый** кадр содержит одну страницу данных.

При страницочном подходе внешняя фрагментация памяти в целом отсутствует, а внутренняя фрагментация будет только в последнем блоке процесса.

При такой организации мы не обязаны располагать все страницы в основной памяти последовательно благодаря логическим адресам, однако для этого мы должны поддерживать **таблицу страниц**.

Номер кадра	Основная память
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

а) 15 доступных кадров

Номер кадра	Основная память
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

б) Загрузка процесса А

Номер кадра	Основная память
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	
8	
9	
10	
11	
12	
13	
14	

в) Загрузка процесса В

Номер кадра	Основная память
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

г) Загрузка процесса С

Номер кадра	Основная память
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

д) Выгрузка процесса В

Номер кадра	Основная память
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

е) Загрузка процесса D

**Таблица страниц** -- структура, которая указывает расположение кадров каждой страницы процесса.

Теперь процессору достаточно знать где находится таблица страниц для текущего процесса и сам логический адрес должен состоять из номера страницы и смещения внутри страницы, что позже используется процессором для трансляции этого в физический адрес. Таблица страниц по одной записи для каждой страницы, каждая запись может быть проиндексирована и равна номеру страницы и каждая запись содержит номер фрейма в основной памяти. К тому же ОС должна держать список свободных фреймов.

0	0
1	1
2	2
3	3

Таблица страниц процесса А

0	—
1	—
2	—

Таблица страниц процесса В

0	7
1	8
2	9
3	10

Таблица страниц процесса С

0	4
1	5
2	6
3	11
4	12

Таблица страниц процесса D

13
14

Список свободных кадров

Использование страниц размером кратного 2 дает нам простую аппаратную реализацию трансляции адресов. Пусть адрес состоит из  $n+m$  бит, где  $n$  -- номер страницы, а  $m$ --смещение

- Выделяем номер страницы, то есть берем  $n$  первым бит
- Ищем в таблице страниц по номеру страниц номер кадра  $k$
- Начальный адрес кадра представляет из себя  $k * 2^m$  то есть номер кадра умноженный на размер кадра а нужный адрес равен это число + смещение, которое находится в  $m$  последних бит логического адреса.

## ВЫВОД

Итак, в случае простой страничной организации основная память разделяется на множество небольших кадров одинакового размера. Каждый процесс разделяется на страницы того же размера, что и кадры; малые процессы требуют меньшего количества кадров, большие - большего. При загрузке процесса в память все его страницы загружаются в свободные кадры, и информация о размещении страниц заносится в соответствующую таблицу. Такой подход позволяет избежать множества присущих распределению памяти проблем.

## Сегментная организация

Альтернативный подход -- сегментация. На сегменты не накладывается условие равенства размеров.

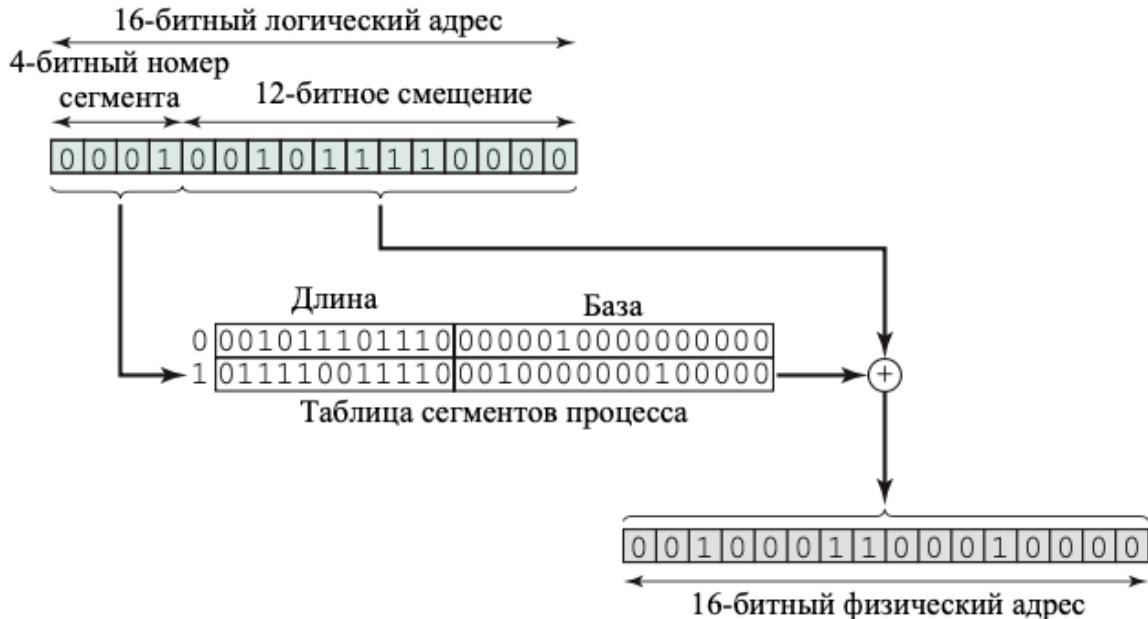
Как и при страничной организации адрес состоит из двух частей -- номер сегмента и смещение.

Устраняется внутренняя фрагментация, однако есть внешняя фрагментация, однако ее степень снижается ввиду разделения процесса на ряд небольших частей.

Главная забота при работе с сегментами становится то чтобы размер сегмента не превысил максимальный. При работе с сегментами у нас отсутствует простая связь между логическими и физическими адресами и для трансляции необходимо сделать следующее

- Выделить номер сегмента из  $n$  первым бит логического адреса
- Найти физический адрес начала сегмента из таблицы сегментов по номеру сегмента
- Сравнить смещение, представляющее собой  $m$  правых бит из логического адреса, с длиной сегмента. Если смещение больше длины, то адрес неверный

- Требуемый физический адрес = физический адрес начала сегмента + смещение



б) Сегментация

## ВЫВОД

Итак, в случае простой сегментации процесс разделяется на ряд сегментов, размер которых может быть разным. При загрузке процесса все его сегменты размещаются в свободных областях памяти и соответствующая информация вносится в таблицу сегментов.

## 43. Виртуальная память основные определения и принципы организации аппаратуры и управляемых программ.

**Реальный (физический) адрес** -- адрес ячейки в основной памяти

**Виртуальный адрес** -- логический адрес внутри процесса. Адрес, присвоенный местоположению в виртуальной памяти, который позволяет обращаться к данному местоположению так, как если бы это была часть основной памяти

**Адресное пространство** -- диапазон адресов, доступный процессу

**Виртуальное адресное пространство** -- область для одного процесса с виртуальными адресами

**Виртуальная память** -- схема расположения процессов в памяти, в которой:

- Вторичная память адресуется также как и основная
- Виртуальные адреса транслируются в адреса основной памяти
- Основная память может быть расширена на вторичную
- Размер памяти ограничен схемой адресации и размером вторичной памяти, но не фактическим количеством ячеек основной памяти

## **Принцип организации аппаратуры и управляющих программ**

Организация виртуальной памяти -- совокупность программных и аппаратных средств.

Современные схемы реализации -- это совокупность страничной и сегментной адресации.

- Все ссылки на память в рамках процесса представляют собой виртуальные адреса, которые динамически транслируются в физические. Таким образом, процесс может быть выгружен при необходимости и загружен в ту же или другую область основной памяти
- Процесс разбивается на страницы или сегменты, которые не обязаны находиться рядом друг с другом

**Две эти характеристики приводят к тому, что наличие всех страниц или сегментов в основной памяти одновременно не является обязательным условием**

Если фрагмент, который находится в основной памяти содержит следующую инструкцию то как минимум программа какое-то время может выполняться без наличия всех ее частей в основной памяти.

Часть процесса, которая находится в некоторый момент времени в основной памяти называется **резидентной частью** этого процесса

### **Следствия применения такой стратегии**

- **В основной памяти может поддерживаться большее количество процессов.** Поскольку мы загружаем туда всего несколько нужных блоков для каждого процесса, то таким образом мы можем поместить в основную память больше процессов
- **Процесс может быть больше чем основная память.** Раньше необходимо было заботиться о том, чтобы программа поместились в основную память либо применять различные стратегии оверлея. Сейчас же эта функция перешла к аппаратуре и ОС и в распоряжении программиста по сути есть большой объем памяти, который он может использовать. ОС при необходимости загружает/выгружает нужные блоки.

**Real Memory** --та память, непосредственно в которой исполняется процесс

**Virtual Memory** -- та память, которую процесс может занять (к примере целый диск)

К тому же стоит заметить, что ОС следует очень разумно обмениваться блоками данными между основной и вторичной памяти. Если ОС будет постоянно выгружать и загружать блок, который постоянно нужен (потому что к примеру основная память вся заполнена, но необходимо место для нового процесса), то получится, что компьютер почти все время будет заниматься только обменом данных а не полезной работой.

Поэтому придумали некоторые методы, которые базируются на принципе локальности, который гласит, что обращения к коду и данными в процессе имеют тенденцию к кластеризации.

## 44. Виртуальный страничный обмен. Двухуровневая организация MMU и TLB 80386

---

Как и в случае простой страничной организации памяти процесс делится на набор страниц одинакового размера, равного размеру кадра основной памяти, однако для выполнения не все страницы необходимо загружать в основную память.

### Виртуальный адрес



В современных системах принято, что страница занимает 4Кб, поэтому смещение обычно занимает 12 разрядов

### Запись таблицы страниц



P -- бит присутствия в основной памяти

M -- бит модификации. Показывает было ли изменено содержание страницы со временем последней загрузки ее в основную память. Если модификаций не было, то при выгрузке страницы нам не потребуется записывать страницу обратно на диск так как смысла нет

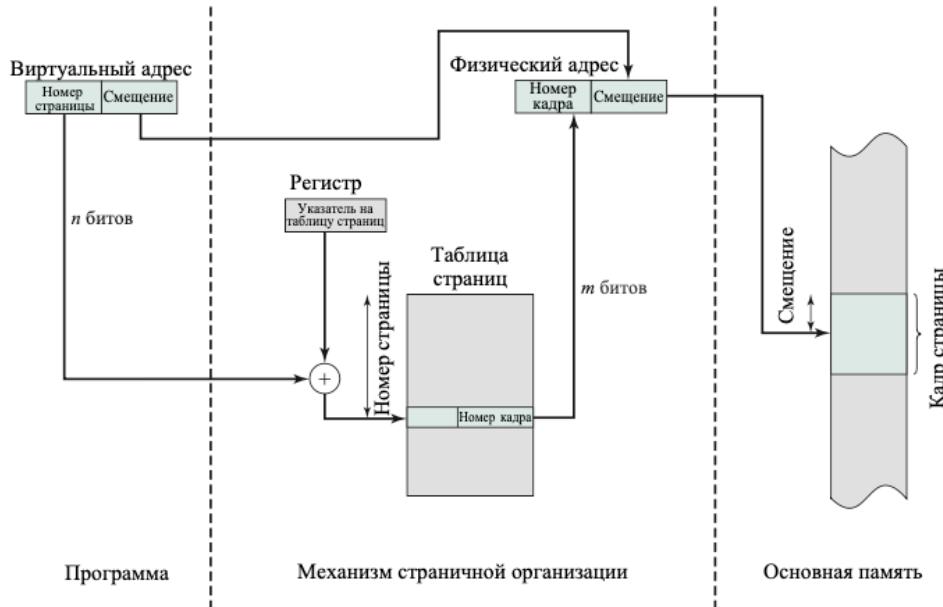
### Структуры таблицы страниц

#### Одноуровневая

Поскольку таблица страниц имеет переменную длину в зависимости от размера процесса, то хранить ее в регистрах не получится, поэтому она должна храниться в основной памяти.

В одноуровневой структуре у нас есть одна таблица, которая хранит все страницы процесса

## Пример аппаратной реализации



## Двух- и Многоуровневая

К примеру в архитектуре VAX каждый процесс может иметь до  $2^{31} = 2\text{Гб}$  виртуальной памяти. С размером страниц 512 байт нам понадобится  $2^{22}$  записей в таблице страниц для каждого процесса что неприемлемо держать такие таблицы в основной памяти. На сегодняшний день большинство схем виртуальной памяти используют многоуровневые страницы, где сама таблица страниц храниться в виртуальной памяти становясь ее объектами.

Для двухуровневой организации мы можем использовать корневую таблицу, каждая запись которой будет ссылаться на другие таблицы страниц и уже с помощью них мы сможем находить нужные адреса

Так, мы можем в основной памяти держать корневую таблицу (размер которой небольшой) и несколько страниц второго уровня, заменяя их из вторичной памяти при необходимости

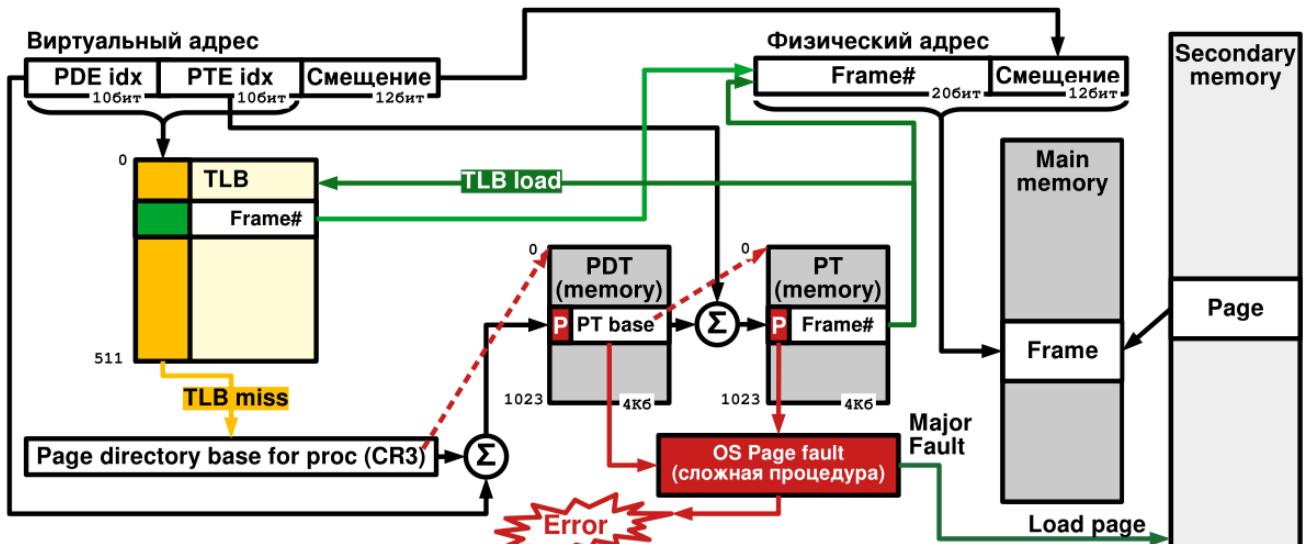
Но конечно все это дает свои накладные расходы по количеству обращений к памяти, чтобы получить нужный адрес.

## Инвертированная таблица

Альтернативой иерархической таблице страниц является инвертированная таблица страниц. Она также используется для трансляции виртуальных адресов в физические. Инвертированная таблица страниц отличается тем, что вместо хранения записи для каждой виртуальной страницы каждого процесса, она хранит записи для всех физических страниц памяти.

С помощью чепоек мы можем ходить по таблице пока не найдем нужный адрес и к тому же такая таблица занимает фиксированный размер

## Двухуровневая организация MMU и TLB 80386



**PDE (page directory entry)** -- указывает на смещение в таблице PDT, которая является вспомогательной для нахождения адреса из основной таблице страниц

**PTE (page table entry)** -- смещение в таблице страниц. Складывает с найденным РТ base из PDT для поиска фрейма из основной памяти.

- Все страницы по 4Кб

**Page directory table for proc (CR3)** -- регистр, который имеет уникальное значение для каждого процесса. Этот регистр указывает на начало таблицы PDT. У каждого процесса этот регистр хранится в контексте процесса. Когда процесс переключается этот регистр также перезаписывается другим значением

**TLB (translation lookaside buffer)**--просто кэш, построенный на базе ассоциативной памяти, который помогает ускорить поиск фреймов основываясь на виртуальном адресе (совокупность PDE index и PTE idx, которые являются тегом для кэша). Получив виртуальный адрес сначала мы смотрим нет ли в TLB уже нужной записи, если есть, то сразу пишем номер фрейма в физический адрес и копируем смещение. Если нет то у нас происходит кэш мисс и мы идем искать адрес из таблицы страниц

### TLB -- часть процессора как часть MMU

Так как таблицы страниц находятся также в основной памяти, то их также может не быть. Некоторые ОС делают их невыгружаемыми.

Либо если при трансляции оказалось, что нужной страницы нет в памяти, то срабатывает **page fault**, что является сложной процедурой и в разных системах ее обрабатывают по-разному. Основной путь решения такой

- Если мы знаем, что страница есть во вторичной памяти, то мы можем оформить запрос ко вторичной памяти, загрузки фрейм в основную память и продолжить трансляцию.

После решения проблемы page fault и в целом после завершения трансляции адреса у нас автоматически обновляется TLB записывая туда новую запись

## Типы page fault

- **Minor fault** -- когда у нас есть страничка в основной памяти, но нет мэппинга для нее в таблице страниц
- **Major fault** -- когда у нас есть страница во вторичной памяти и мы должны загрузить ее в основную

## 45. Инвертированная таблица страниц

Альтернативой иерархической таблице страниц является инвертированная таблица страниц. Она также используется для трансляции виртуальных адресов в физические. Инвертированная таблица страниц отличается тем, что вместо хранения записи для каждой виртуальной страницы каждого процесса, она хранит записи для всех физических страниц памяти.

Часть виртуального адреса, представляющая из себя номер страницы попадает на хэш-функцию, на выходе имеет  $m$ -битный номер, который используется как индекс в самой таблице. Каждому кадру соответствует одна запись в таблице. Таким образом, нам нужен фиксированный размер памяти для таблицы вне зависимости от количества процессов и поддерживаемых страниц. Поскольку на одну и ту же запись могут отображаться несколько виртуальных адресов то используется технология цепочек для обработки переполнения.

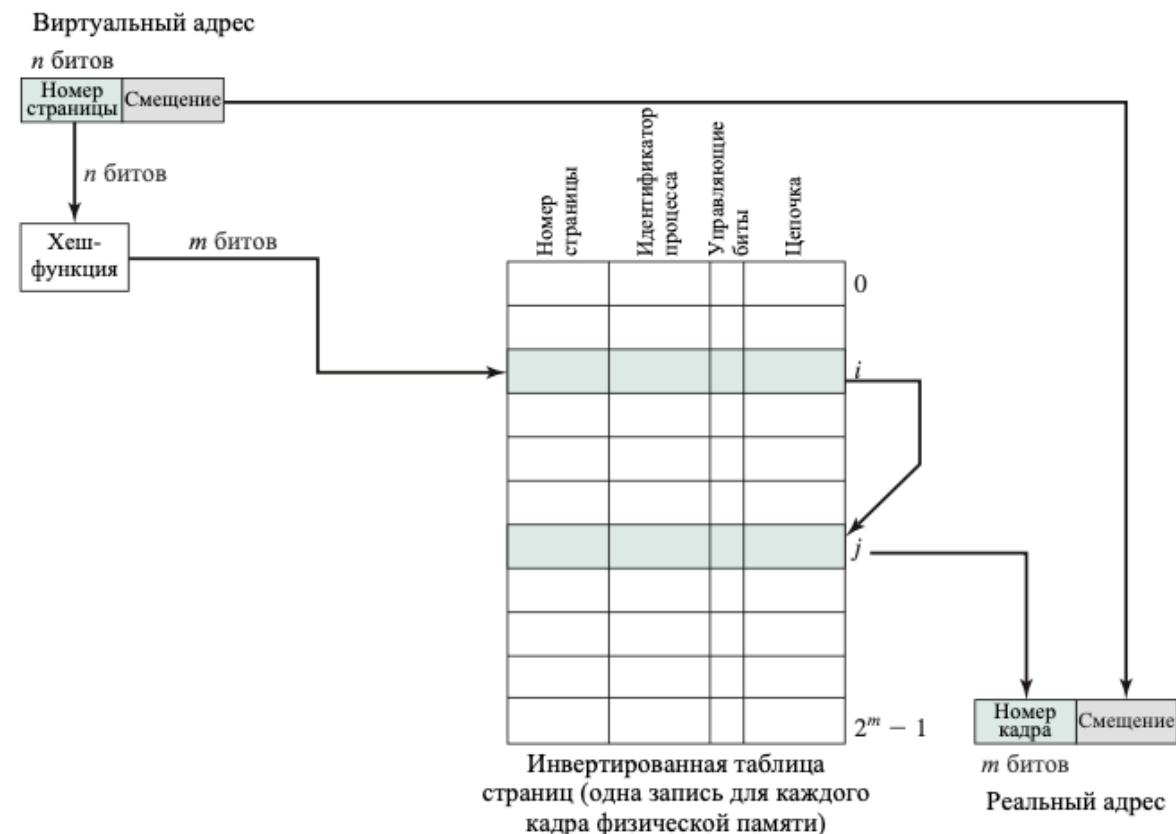


Рис. 8.5. Структура инвертированной таблицы страниц

Каждая запись в таблице включает:

- **Номер страницы.** Часть виртуального адреса, относящегося к номеру страницы
- **ID процесса.** Процесс, который владеет этой страницей. Комбинация PID и номера страницы идентифицирует страницы в виртуальном адресном пространстве
- **Управляющие биты.** Флаги, к примеры, корректности, модификации и иные, а также информацию о защите и блокировках
- **Указатель цепочки.** Это поле нулевое, если для этой записи нет других связанных цепочкой записи. В противном случае это поле содержит значение индекса (0 до  $2^m - 1$ ) следующей записи в таблице.

## 46. Сегментно-страничная виртуальная память

---

Сегментная виртуальная память основывается на разделении процесса на ряд сегментов, которые могут иметь разные (динамические размеры), что дает следующие преимущества по сравнению с несегментированным адресным пространством

- **Упрощение обработки растущих структур данных.** Если неизвестен размер структуры то ей можно назначить свой сегмент, который при необходимости будет увеличиваться. Если места для сегмента в памяти недостаточно, то он может быть перенесен во вторичную память
- **Упрощение совместно используемых данными.** Их можно вынести в отдельный сегмент, к которому будут обращаться все процессы
- **Улучшение защиты.** На каждый сегмент, который точно определяет множество данных, можно поставить права доступа

### Виртуальный адрес и запись в таблице страниц выглядят так

Длина в записи таблице страниц нам нужен, чтобы случайно не выйти за пределы сегмента и удостовериться, что адрес действительно верный.

Виртуальный адрес

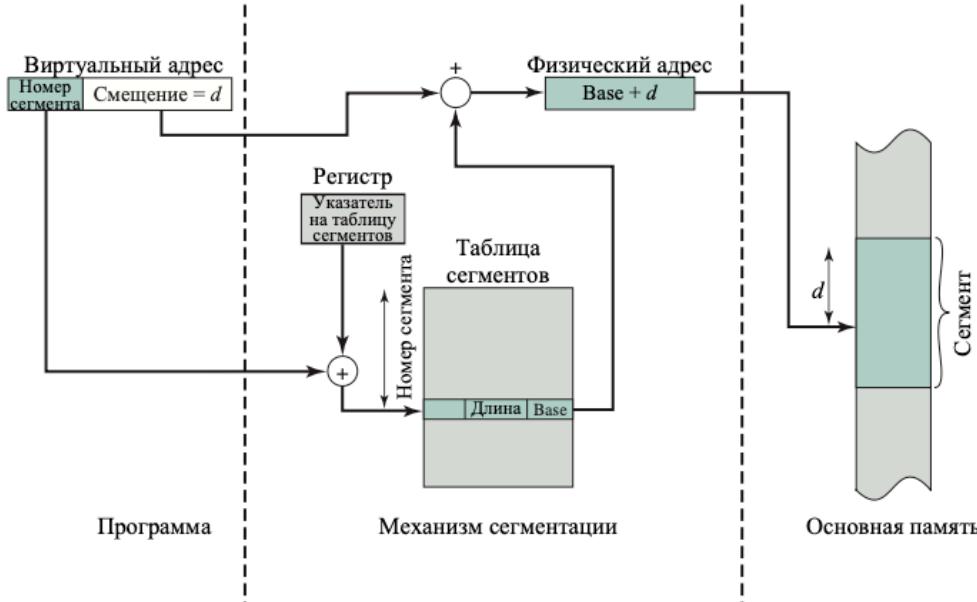
Номер сегмента	Смещение
----------------	----------

Запись таблицы сегментов

P M	Прочие управляющие биты	Длина	Начальный адрес сегмента
--------	-------------------------	-------	--------------------------

б) Сегментация

## Процесс трансляции адресов аналогичен со страницной виртуальной памятью



### Сегментно-страничный подход

Наиболее часто используемый подход в реальных системах.

В комбинированной схеме адресное пространство делится на ряд сегментов по усмотрению программиста. Каждый сегмент в свою очередь разбивается на ряд страниц одинакового размера. Если сегмент меньше страницы, то сегмент занимает всю страницу.

С точки зрения программиста, логический адрес -- номер сегмента и смещение в нем. С позиции ОС логический адрес -- номер страницы внутри сегмента и смещение в нем.

**Причем страницная часть реализуется аппаратурой.** Можно сказать, что сегменты логически делят наш процесс, а внутри сегментов -- страницы уже физически отображаются на основную память.

#### Виртуальный адрес

Номер сегмента	Номер страницы	Смещение
----------------	----------------	----------

#### Запись таблицы сегментов

Управляющие биты	Длина	Начальный адрес сегмента
------------------	-------	--------------------------

#### Запись таблицы страниц

P	M	Other control bits	Номер кадра
---	---	--------------------	-------------

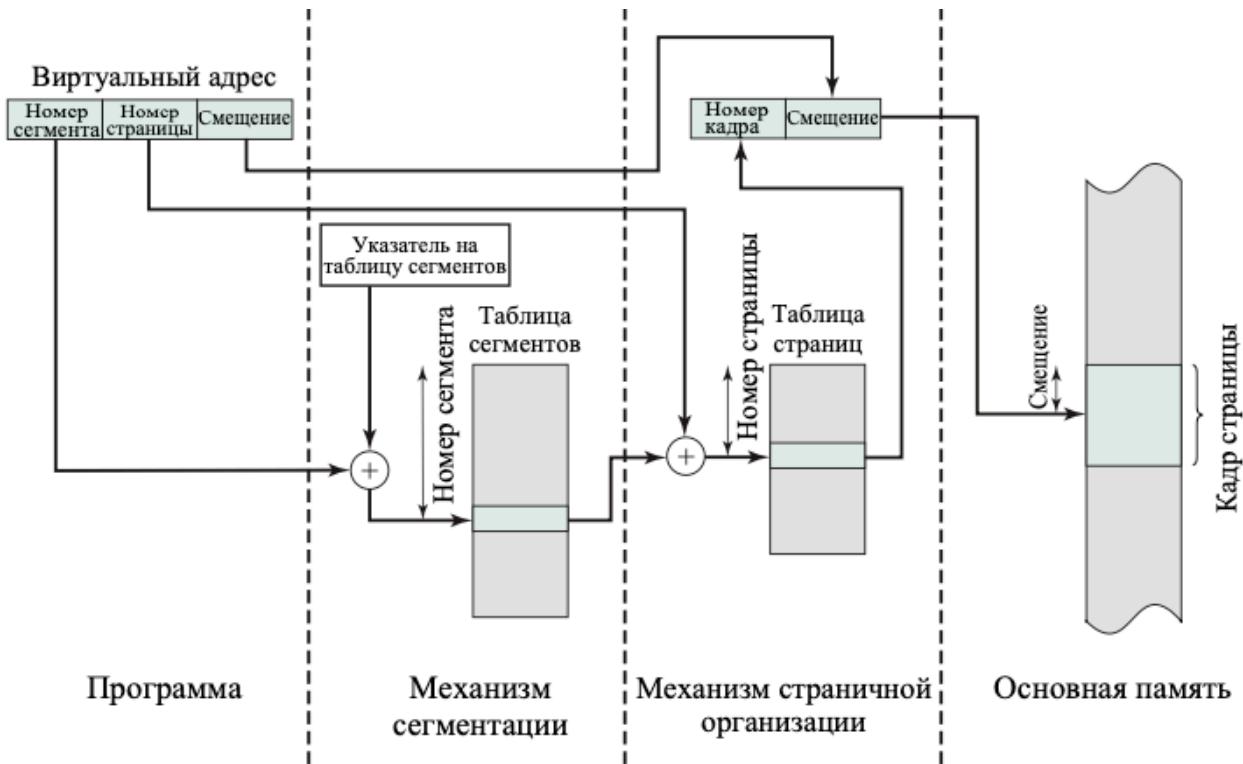
P — бит присутствия  
M — бит модификации

в) Комбинация страницной организации и сегментации

С каждым процессом связана одна таблица сегментов и несколько (по одной на сегмент) таблиц страниц.

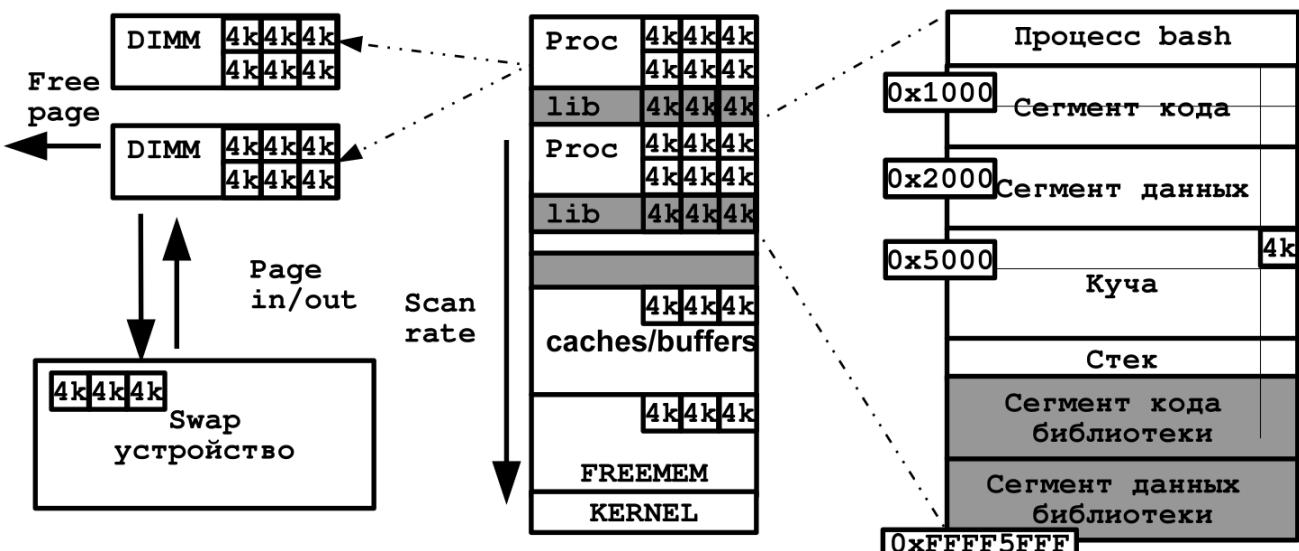
Получив виртуальный адрес, процессор использует его часть, представляющую номер

сегмента, в качестве индекса в таблице сегментов для поиска таблицы страниц данного сегмента. После этого часть адреса, представляющая собой номер страницы, используется для поиска соответствующего кадра основной памяти в таблице страниц; затем часть адреса, представляющая смещение, используется для получения искомого физического адреса путем добавления к начальному адресу кадра.



Ядро обычно занимает какую-то часть основной памяти (Kernel), также в основной памяти может храниться список свободных страниц и при запросе ОС может выдать их (FREEMEM)

Все страницы лежат в виде такого списка и при недостатке памяти ОС начинает сканировать их и искать то, что можно выгрузить из памяти.



## 47. Влияние размера страницы виртуальной памяти на ОС. Стратегии ОС по работе с виртуальной памятью.

---

При разработке ОС надо определиться с размером страницы. **Важно учитывать:**

- Размер таблицы страниц
- Внутренняя фрагментация
- Количество page-fault при трансляции адреса
- Скорость взаимодействия со вторичной памятью (и размер блока)
- Локальность данных (в многопоточных приложения ниже)
- Количество промахов TLB, размер TLB, размера кэшов

С одной стороны страницы должны быть маленькие, чтобы уменьшить внутреннюю фрагментацию, но с другой стороны чем меньше страницы, тем их больше, что приводит к перегрузке TLB, возникает большое количество прерываний и page fault и сами таблицы страниц становятся больше.

Результативность поиска TLB определенного размера с ростом размера процессов и уменьшением локальности снижается.

Много что зависит также от возможностей не только ОС, но и самого процесса в частности MMU.

Вторичная память также имеет значение. Раньше, когда вторичная память была простым HDD диском, то явно выгоднее было бы прочитать все секторы на дорожке, нежели по блокам (которые занимали по 512Кб). Так ОС читала набор блоков сразу, чтобы ускорить процесс загрузки страниц.

Современные процессора могут работать с разными размерами страниц (large, huge pages)

- Intel: 4Mb, 2Mb, 1Gb
- Sparc64: 8Kb, 64Kb, 512Kb, 4Mb, 32Mb, 256Mb

Совсем большие страницы стараются использовать тогда, когда их не придется класть в своп, так как скорость записи на диск будет медленная.

## Стратегии ОС по работе с виртуальной памятью

### Стратегии выборки

Определяет когда страница должна быть передана в основную память.

- **По требованию.** Страница передается только тогда, когда выполняется обращение к ячейке памяти, расположенной на этой странице. Когда только запускается система должно быть много прерываний, но далее срабатывает принцип локальности, что снижает количество page fault к минимуму
- **Предварительная выборка.** Загружается не только страницы вызвавшая прерывание. Предварительная выборка использует характеристики устройств вторичной памяти. Если страницы процесса находятся последовательно, то явно гораздо эффективней загрузить их все в память ( read ahead). Если такого не

происходит, то конечно никакого выигрыша нет. Может применяться при первом запуске либо при каждом прерывании

### **Стратегии размещения**

В чистой сегментации имеет место рассматривать стратегии такие как: наилучший подходящий, первый подходящий или следующий подходящий.

Однако в системах со страничной или сегментно-страничной стратегия размещения не так важна, потому что аппаратная трансляция адреса и аппаратное обращение к памяти одинаково результативны при любых сочетаниях "страница-кадр".

В NUMA процессорах где скорость обращения памяти зависит от ее расположение стратегия размещения может иметь серьезный эффект на скорость работы системы. Суммарная производительность системы зависит от того, насколько близко к процессору будут находиться данные.

### **Стратегия отчистки**

Противоположность стратегии выборки. Определение когда измененная страница должна быть записана во вторичную память .

- **По требованию.** Записывается во вторичную память только тогда, когда она была выбрана для замещения
- **Предварительная очистка\*** . Изменённые страницы (т.е. страницы, содержимое которых было модифицировано в оперативной памяти) записываются во вторичную память заранее, до того момента, когда они станут кандидатами на замещение.

Каждая стратегия имеет недостатки:

- При предварительной очистке страницы могут записываться зря, если они будут изменены до замещения, перегружая вторичную память.
- При очистке по требованию приходится одновременно записывать одну страницу и читать другую, снижая производительность.

Улучшенный подход использует буферизацию. Замещаемые страницы делятся на модифицированные и немодифицированные.

- Модифицированные страницы периодически записываются пакетами и переходят в список немодифицированных.
- Немодифицированные страницы либо удаляются при замещении, либо остаются, если к ним снова обращаются.

### **Управление многозаданостью**

Если одновременно резидентны несколько процесс, то часто будет возникать ситуация, когда все процессы будут заблокированы и ОС надо тратить время на свопинг. С другой стороны, если в основной памяти будет много процессов, то в среднем размер резидентного множества будет мал, что приведет к частой генерации ошибок обращения и снижению пропускной способности системы.

При возрастании уровня многозадачности от малых значений эффективность использования процессора возрастает в связи с уменьшением вероятности одновременного блокирования всех процессов. Однако через некоторое время достигается состояние, когда средний размер резидентного множества становится неадекватным. Это приводит к существенному росту количества прерываний обращений к странице и как следствие к снижению эффективности использования процессора.

наверное необязательно, но пусть будет если будет время и силы выучить

Есть ряд решений:

- **Критерий 50%.** Поддержка степени использования устройства хранения страниц на 50 процентов
- **Критерий L=S.** Уровень многозаданочти натсрывается таким образом, что среднее время между прерываниями процессов равнялось среднему временем, требующемуся для обработки прерывания.

### **Приостановка процессов**

При необходимости снижения степени многозадачности один или несколько резидентных в настоящее время процессов должны быть приостановлены (выгружены во вторичную память). В [36] перечислены шесть возможностей.

- **Процесс с наименьшим приоритетом.** Так реализована стратегия планировщика, не имеющая отношения к вопросам производительности.
- **Процесс, вызывающий прерывания.** Данный выбор основан на большой вероятности того, что у процесса, генерирующего прерывания из-за отсутствия страницы, рабочее множество не резидентно, и суммарная производительность системы не пострадает при приостановке данного процесса. Кроме того, при таком выборе блокируется процесс, который и так практически все время находится в заблокированном состоянии, так что его приостановка приводит к снижению накладных расходов, связанных с замещением страниц и операциями ввода-вывода.
- **Последний активированный процесс.** Маловероятно, что у этого процесса рабочее множество резидентно.
- **Процесс с минимальным резидентным множеством.** Этот выбор минимизирует будущие затраты на загрузку данного процесса. К сожалению, таковыми являются процессы с высокой степенью локальности.
- **Наибольший процесс.** При этом выборе мы освобождаем большое количество кадров перегруженной процессами памяти, снижая тем самым количество процессов, которые должны быть деактивированы.
- **Процесс с максимальным остаточным окном выполнения.** В большинстве схем планирования процесс может выполняться только определенное количество квантов времени до прерывания и перемещения его в конец очереди активных процессов. Данный выбор приближается к стратегии планирования, предоставляющей преимущество процессам с наименьшим временем работы.

### **Стратегии замещения**

Вопросы стратегии замещения представляют собой, пожалуй, наиболее полноизученный аспект управления памятью. Когда все кадры основной памяти заняты и требуется разместить новую страницу в процессе обработки прерывания из-за отсутствия

страницы, стратегия замещения определяет, какая из находящихся в настоящее время в основной памяти страниц должна быть выгружена, чтобы освободить кадр для загружаемой страницы. Все стратегии направлены на то, чтобы выгрузить страницу, обращений к которой в ближайшем будущем не последует. В соответствии с принципом локальности часто наблюдается сильная корреляция между множеством страниц, к которым в последнее время были обращения, и множеством страниц, к которым будут обращения в ближайшее время. Таким образом, большинство стратегий пытаются определить будущее поведение программы на основе ее прошлого поведения. При рассмотрении разных стратегий следует учитывать, что чем более совершенный и интеллектуальный алгоритм использует стратегия, тем выше будут накладные расходы при его реализации.

### **Управление резидентной частью**

## **48. Стратегии замещения страниц ОС. Часовой Алгоритм.**

### **Управление резидентной частью процесса**

---

### **Стратегии замещения страниц**

Выбор страниц в основной памяти для их замещения загружаемыми страницами из вторичной памяти. **Включает ряд вопросов**

- Какое кол-во кадров должно быть выделено каждому процессу
- Стоит ли нам рассматривать страницы одного процесса либо все множество страниц
- Какие именно страницы следует выбрать -- стратегия замещения определяет именно это

Все стратегии направлены на то, чтобы выгрузить те страницы к которым не будет обращений в ближайшее время. К тому же большинство стратегий пытаются предсказать поведение программы на основе ее прошлого поведения.

### **Ограничение**

- Некоторые кадры основной памяти могут быть заблокированы. Блокировка означает невозможность замещения страницы в данный момент. Большинство ядер и управляющих структур находятся в таком состоянии. Блокировка осуществляется путем установки соответственного бита у каждого кадра.
- Если кадр используется несколькими процессами, то его выгрузка может повлиять на работу всех этих процессов

### **Основные алгоритмы**

- **Оптимальный алгоритм.** Выбор для замещения страницы, обращение к которой будет через наибольший промежуток времени по сравнению с остальными. Такой алгоритм в целом невозможен так как мы не можем предсказать все будущие события

Обращения к страницам	2	3	2	1	5	2	4	5	3	2	5	2
Оптимальный алгоритм												
F					F				F			F

- **LRU (дольше всех неиспользованная).** Замещает ту страницу, к которой обращений не было больше всего чем к другим. Недалека от оптимальной. Сложна в реализации как программно, так и аппаратно. Зачастую используются гибридные аналоги

Обращения к страницам	2	3	2	1	5	2	4	5	3	2	5	2
Алгоритм "дольше всех неиспользовавшийся"												
F					F				F			F

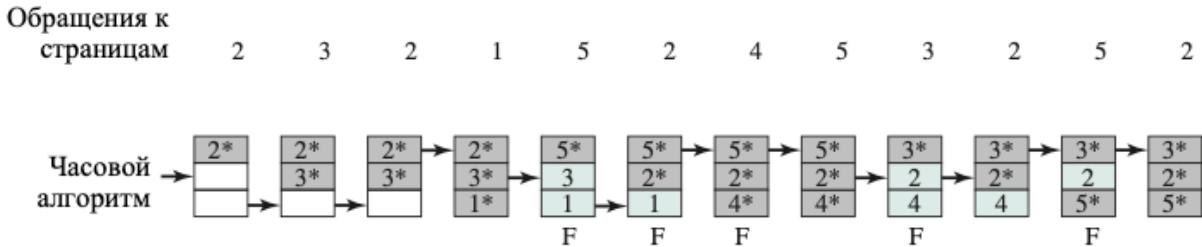
- **FIFO.** Рассматривает кадры страниц процесса как циклический буфер с циклическим же удалением страниц. Все что требуется -- указатель циклически проходящий по кадрам. Замещается та страница, находящаяся в памяти дольше всех. Однако далеко не всегда эта страница редко используется

Обращения к страницам	2	3	2	1	5	2	4	5	3	2	5	2
Алгоритм "первым вошел — первым вышел"												
F					F				F			F

- **Часовой алгоритм** 💀 💀

Часовая стратегия. В простейшем виде с каждым кадром связывается один дополнительный бит, известный как бит использования (**fun fact:** отсутствие битов сделаем часовой алгоритм простым FIFO). Когда страница впервые загружается, то бит устанавливается в единицу. При последующих обращениях к странице, вызвавших прерывание из-за отсутствия страницы, этот бит также ставится 1. Множество кадров-кандидатов на замещение рассматривается как циклический буфер, с которым связан указатель. При замещении страницы указатель переходит на следующий кадр. Когда наступает время замещения страницы, операционная система сканирует буфер для поиска кадра, бит использования которого равен 0. Всякий раз, когда в процессе поиска встречается кадр с битом использования, равным 1, он сбрасывается в 0. Первый же встреченныи кадр с

нулевым битом использования выбирается для замещения. Если все кадры имеют бит использования, равный 1 , указатель совершает полный круг и возвращается к начальному положению, заменяя страницу в этом кадре.



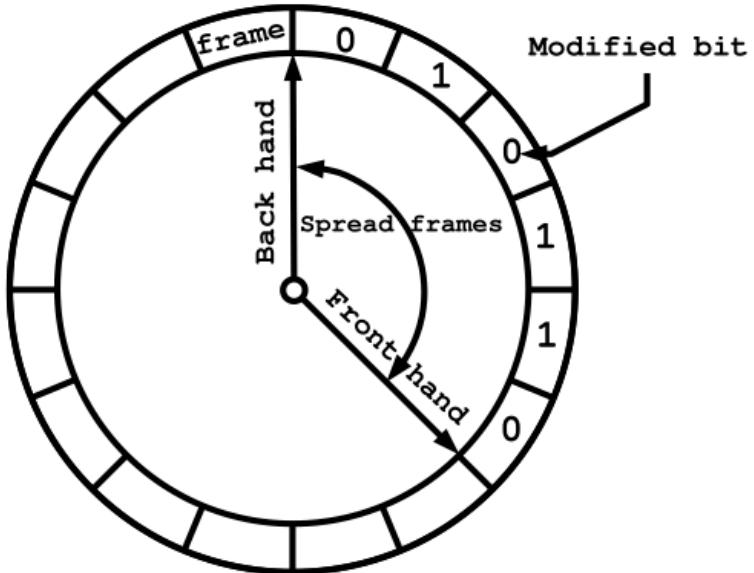
Повысить эффективность часового алгоритма можно путем добавления бита модификации помимо бита использования. Итак, часовой алгоритм циклически проходит по всем страницам буфера в поисках страницы, которая не была модифицирована со времени загрузки и давно не использовалась. Такая страница - хороший кандидат на замещение, особенно с учетом того, что ее не надо записывать на диск. Если при первом проходе кандидатов на замещение не нашлось, алгоритм снова проверяет буфер, теперь уже в поисках модифицированной, давно не использовавшейся страницы. Хотя такая страница и должна быть записана перед замещением, в соответствии с принципом локальности она вряд ли понадобится в ближайшем будущем. Если и этот проход окажется неудачным, все страницы помечают ся как давно не использованные, и выполняется третий проход.

## ИЗ ЛЕКЦИИ

Аналогично есть бит изменения. Циклический буфер, но два указателя: Front Hand и BackHand, которые врачаются одновременно по всему буферу

- Первая стрелка сбрасывает бит модификации
- Модификация фрейма устанавливает бит модификации
- Задняя стрелка проверяет установился ли бит модификации для страница за то время за которое задняя стрелка дойдет до положения передней стрелки
- Если бит установлен -- значит страница использовалась и плохой кандидат. Если нет -- то в зависимости от загрузки системой пейджингом можно положить эту страницу в буфер страниц, которые требуют вытеснения во вторичную память.
- Расстояние между стрелками может динамически меняться в зависимости от нагрузки системы. Чем меньше расстояние тем больше и быстрее страниц будет выгружаться

В сами страницы мы не заглядываем, а работаем с управляющей информацией, которая может находиться в PTE.



В современных системах количество фреймов насчитывае миллионы и поэтому сейчас пытаются отойти от часовых алгоритмов на более современные и быстрые.

### **Управление резидентной частью процессов**

При загрузке и работе процесса нет смысла держать все его страницы. Тогда вопрос сколько страниц стоит загружать для каждого процесса ? Здесь играет ряд факторов

- Чем меньше памяти выделяется процессу, тем более процессов может одновременно находиться в памяти. Увеличивает шансы того, что ОС найдет хотя бы один процесс готовый к выполнению
- При небольшом кол-ве страниц процесса повышается количество пейдж фолтов, несмотря на принцип локальности
- После N страниц дополнительное выделение памяти не приводить к значительному снижению page faults

### **Два типа стратегий**

#### **Фиксированный**

- Количество кадров процесса фиксированно. Размер указывает при создании процесса
- Страница для замещения выбирается среди выделенных процессу кадров.
- Большие процессы будут испытывать недостаток памяти, а маленькие процессы будут неэкономно расходовать ресурсы системы.

#### **Переменное распределени**

- Количество кадров выделенных процессу может меняться (в идеале процессу, который страдает от частых прерываний и наоборот процессу у которого мало прерываний некоторые кадры могут быть отобраны)
- Страница для замещения также выбирается среди выделенных процессу кадров

Переменное распределение более мощное, но сложное в реализации и дает свои накладные расходы по отслеживанию поведения процесса

### Область видимости замещений

- **Локальная.** Страницы выбираются только среди резидентных страниц того процесса, который стал причиной прерываний
- **Глобальная.** Рассматриваются все незаблокированные страницы в основной памяти, независимо от принадлежности прерывания

При фиксированном распределении глобальное замещение невозможно так как нам тогда необходимо будет заменить страницу того же процесса на другую страницу того же процесса, а переменное распределение может поддерживать обе области видимости

В современном мире используется динамическое распределение с глобальной видимостью.

## 49. Виды планирования процессов. Критерии краткосрочного планирования. Приоритеты.



В явном виде можно выделить несколько временных видов планирования.

1. Краткосрочное планирование, позволяющее, например, переключать трэды (или процессы) из одного состояния в другое. То есть решение о том, какой из доступных процессов будет выполняться процессором.
2. Среднесрочное планирование, которое обычно обеспечено областью подкачки (то бишь это swapping или paging); и надо спланировать - какие страницы положить в область подкачки, а какие оттуда брать

3. Долгосрочное планирование - обычно связанное с процессами. Долгосрочное планирование работает с пакетными заданиями, то есть с заданиями, которые представляют собой последовательность запуска различных программ, для выполнения которых операционная система должна уметь планировать эти процессы. Проще говоря - это решение о добавлении процесса в пул выполняемых процессов
4. Также еще есть планирование ввода-вывода - решение о том, какой из запросов процессов на операции ввода-вывода будет обработан доступным устройством ввода-вывода.

Пример долгосрочного планирования -- запуск си или c++ компилятора.

Запускается огромное количество служебных утилит, которые компилируют исходный код. Так запускается большо задание, которое состоит из явно не из одного процесса, то ОС должна понимать, как лучше такие пакетные задания планировать в долгосрочной перспективе.



## Критерии краткосрочного планирования

- Пользовательские, связанные с производительностью
  - Turnaround time — время оборота
  - Responce time — время отклика
  - Deadline — предельное время
- Пользовательские, иные
  - Predictability — предсказуемость (независимость от остальной загрузки системы)
- Системные, связанные с производительностью
  - Throughput — пропускная способность
  - Processor Utilization — загруженность процессора
- Системные, иные
  - Fairness — справедливость
  - Enforcing priorities — принудительная приоритезация
  - Balancing resources - сбалансированная загрузка

Критерии краткосрочного планирования:

- Пользовательские, связанные с производительностью.
  1. Turnaround time -- время оборота. Интервал времени между передачей процесса для выполнения и его завершением. То бишь включает время выполнения и время на ожидание ресурсов.
  2. Responce time -- время отклика. В интерактивных процессах это время, истекшее от начала запроса до получения ответа.
  3. Deadline -- предельное время или предельный срок. При указании предельного срока завершения процесса, планирование должно подчинить ему все прочие цели максимизации количества процессов, завершающихся в срок.
- Пользовательские, иные

1. Predictability -- предсказуемость. То есть данное задание должно выполняться примерно за одно и то же количество времени и с одной и той же стоимостью, независимо от нагрузки системы.
- Системные, связанные с производительностью
  1. Throughput -- пропускная способность. Стратегия планирования должна пытаться максимизировать количества процессов, завершающихся за единицу времени, что является мерой количества выполненной системой работы.
  2. Processor Utilization -- загруженность процессора.
- Системные, иные
  1. Fairness -- справедливость. При отсутствии дополнительных указаний от пользователя или системы все процессы должны рассматриваться как равнозначные и ни один процесс не должен подвергнуться голоданию.
  2. Enforcing priorities -- принудительная приоритизация. Если процессам назначены приоритеты, то стратегия планирования должна отдавать предпочтения процессам с более высоким приоритетом.
  3. Balancing resources -- Предпочтение должно быть отдано процессу, которые недостаточно использует важные ресурсы.

## 50. Использование приоритетов



- Процессам присваивается цифровой приоритет
  - В разных ОС - разные схемы назначения приоритетов
- Из очередей выбирается процесс с наивысшим приоритетом
- Если приоритеты процессов совпадают, то используется дополнительная стратегия
- Процессы с низким приоритетом могут голодать
- Приоритеты могут динамически изменяться

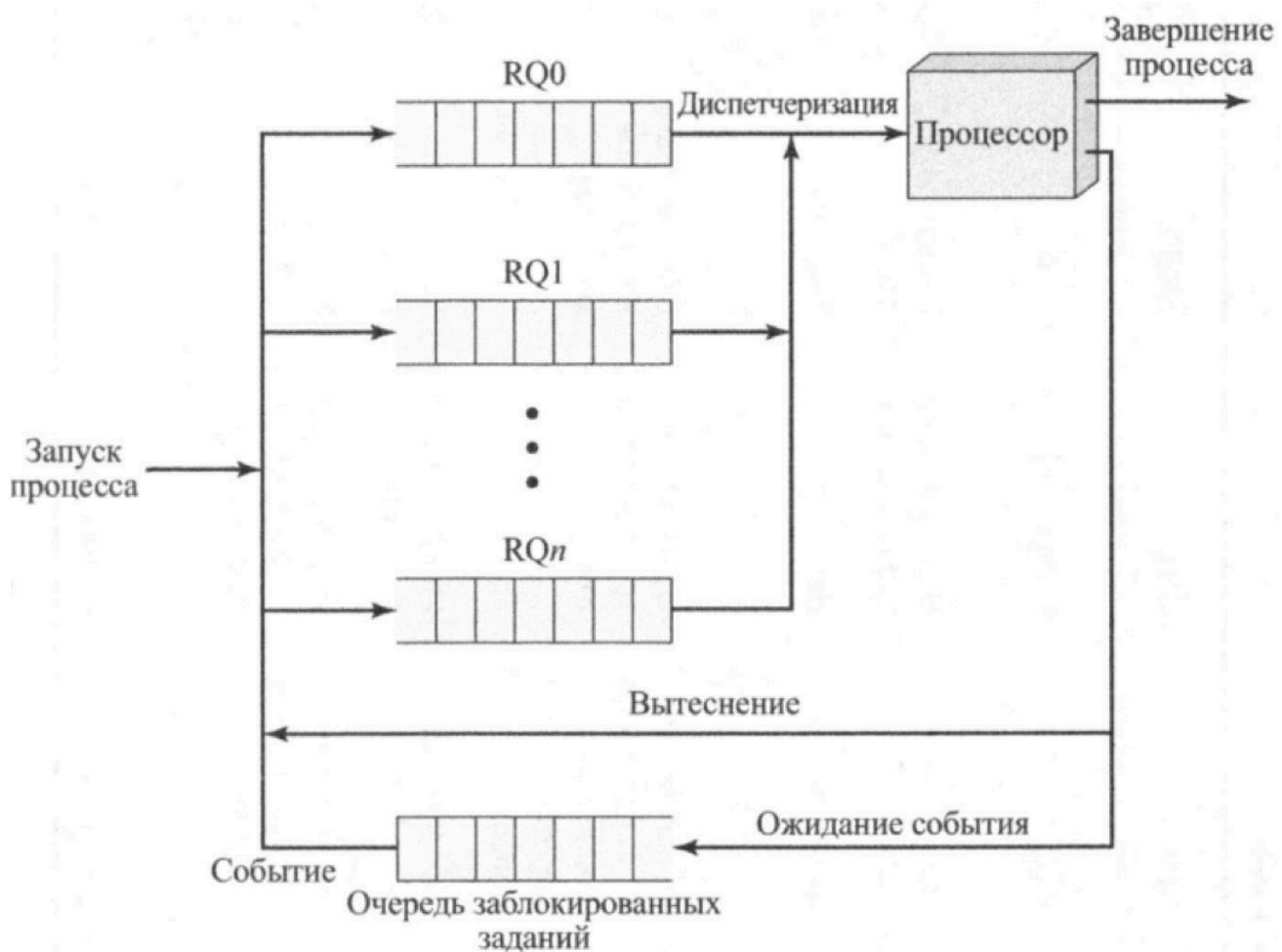
Для того, чтобы выделить процессы и отделить их один от другого во всех современных операционных системах используется приоритеты.

Приоритет -- это просто цифровая величина.

Как же работает использование приоритетов:

1. Процессы ставятся в очереди и выбираются из этих очередей. По умолчанию выбирается процесс с наивысшим приоритетом (RQ0)

2. Если в очереди имеется один или несколько процессов, процесс для работы выбирается с использованием некой стратегии планирования.
3. Если очередь RQ0 пуста, рассматривается очередь RQ1 и т.д.



**Рис. 9.4. Планирование с учетом приоритета процессов**

Одна из основных проблем в такой чисто приоритетной схеме планирования состоит в том, что процессы с низким приоритетом могут оказаться в состоянии голодания. Это будет происходить при постоянном поступлении новых готовых к выполнению процессов с высоким приоритетом. Если такое поведение нежелательно, приоритет процесса может изменяться с его "возрастом" или историей выполнения.

Помимо этого, существует наследование приоритетов, которое позволяет избежать проблему, называемую Priority Inversion (в некоторых случаях задача с высоким приоритетом может быть косвенно вытеснена более низкой приоритетной задачей, эффективно инвертирующей относительные приоритеты двух задач; в целом это нарушает модель приоритета).

## 51. Стратегии планирования FCFS, RR, SPN, SRT, HRRN, Feedback

Таблица 9.3. ХАРАКТЕРИСТИКИ РАЗЛИЧНЫХ СТРАТЕГИЙ ПЛАНИРОВАНИЯ

	FSCF	Круговая	SPN	SRT	HRRN	Со снижением приоритета
<b>Функция выбора</b>	$\max[w]$	const	$\min[s]$	$\min[s - e]$	$\max\left(\frac{w+s}{s}\right)$	См. текст
<b>Режим решения</b>	Невытесняющий	Вытесняющий (по времени)	Невытесняющий	Вытесняющий (по решению)	Невытесняющий	Вытесняющий (по времени)
<b>Пропускная способность</b>	Не важна	Может быть низкой при малом кванте времени	Высокая	Высокая	Высокая	Не важна
<b>Время отклика</b>	Может быть большим, в особенности при больших отклонениях во времени выполнения процесса	Обеспечивает хорошее время отклика для коротких процессов	Обеспечивает хорошее время отклика для коротких процессов	Обеспечивает хорошее время отклика	Обеспечивает хорошее время отклика	Не важна
<b>Накладные расходы</b>	Минимальны	Минимальны	Могут быть высокими	Могут быть высокими	Могут быть высокими	Могут быть высокими
<b>Влияние на процессы</b>	Плохо оказывается на коротких процессах и процессах с интенсивным вводом-выводом	Беспристрастна	Плохо оказывается на длинных процессах	Плохо оказывается на длинных процессах	Хороший баланс	Может привести к предпочтению процессов с интенсивным вводом-выводом
<b>Голодание</b>	Отсутствует	Отсутствует	Возможно	Возможно	Отсутствует	Возможно

Основные понятия:

- $w$  -- время, затраченное к этому моменту системой (время ожидания процесса в очереди);
- $e$  -- время, затраченное к этому моменту на выполнение (общее время выполнения);
- $s$  -- общее время обслуживания, требующееся процессу, включая  $e$  (обычно эта величина оценивается или задается пользователем).

Режимы принятия решения подразделяются на две основные категории.

- **Невытесняющие.** В этом случае находящийся в состоянии выполнения процесс продолжает выполнение до тех пор, а) пока он не завершится или б) пока не окажется в заблокированном состоянии ожидания завершения операции ввода-вывода или запроса некоторого системного сервиса.
- **Вытесняющие.** Выполняющийся в настоящий момент процесс может быть прерван и переведен операционной системой в состояние готовности к выполнению. Решение о вытеснении может приниматься при запуске нового процесса по прерыванию, которое переводит заблокированный процесс в состояние готовности к выполнению, или периодически - на основе прерываний таймера.

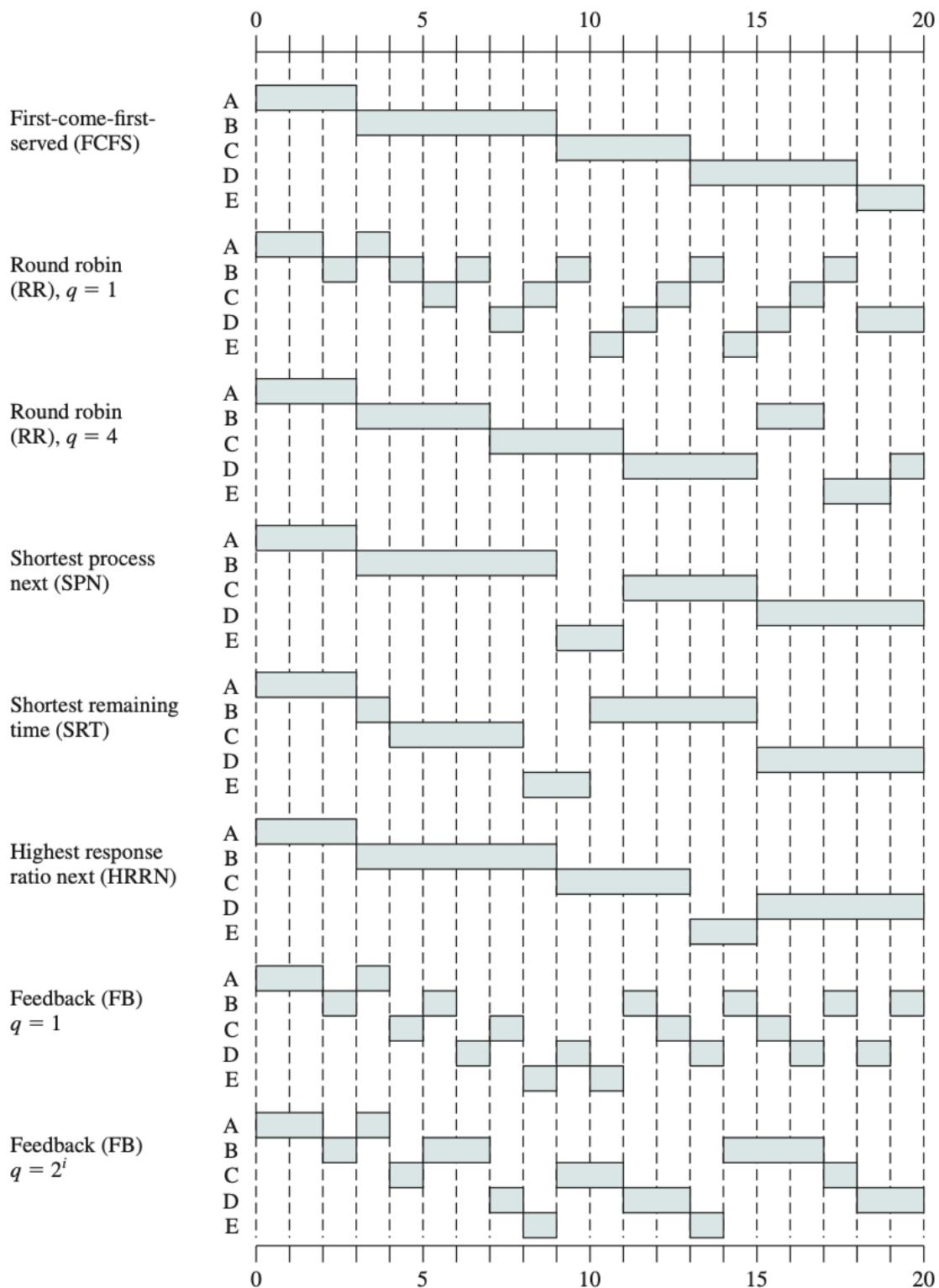
**Время оборота T -- turnaround time -- TAT**, это полное время, затраченное процессом в системе (время ожидания и обслуживания).

**Нормализованное время оборота** -- это величина, которая определяется как отношение времени оборота ко времени обслуживания и указывает относительную

задержку, испытываемую процессом. Обычно, чем больше время выполнения процесса, тем больше абсолютная величина задержки, которая может быть приемлемой. Минимальное значение этого отношения - 1,0. Возрастание значения отношения соответствует снижению уровня обслуживания.

**Таблица 9.4. Пример планирования процессов**

Процесс	Время запуска	Время обслуживания
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



**Рис. 9.5. Сравнение стратегий планирования**

### Стратегия планирования:

1. **FCFS** -- это алгоритм, в котором задачи обрабатываются в порядке их поступления в очередь. Как только задача занимает процессор, она выполняется до завершения, независимо от времени её выполнения. Например, если задачи A, B и C поступили в очередь в порядке A → B → C, то они будут выполняться в этом же порядке, независимо от их продолжительности.
2. **RR (Round Robin)** -- стратегия планирования, основанная на FCFS, но с добавлением кванта времени. Таймер генерирует прерывания через

определенные интервалы времени. При каждом прерывании выполняющийся в настоящий момент процесс помещается в очередь готовых к выполнению процессов, и начинает выполняться очередной процесс, выбранный в соответствии со стратегией FCFS. Эта методика известна также как квантование времени (time slicing), поскольку перед тем как оказаться вытесненным, каждый процесс получает квант времени для выполнения. В данной методике стоит правильно подбирать данный квант времени. Эффективность зависит от размера кванта времени. Слишком маленький квант увеличивает расходы на переключение контекста, а слишком большой — сводит преимущества к FCFS.

3. **SPN -- shortest process next** -- короткие процессы вперед. Для выполнения берется процесс с наименьшим ожидаемым временем выполнения. Основная трудность в применении стратегии SPN состоит в том, что для ее осуществления необходима по меньшей мере оценка времени выполнения, требующегося каждому процессу. При выполнении пакетных заданий может понадобиться оценка этого значения программистом и его предоставление операционной системе. Если оценка программиста существенно ниже реального времени выполнения, система может прекратить выполнение задания.
4. **SRT -- Shortest remaining time** -- наименьшее время до завершения. Стратегия наименьшего остающегося времени (shortest remaining time - SRT) представляет собой вытесняющую версию стратегии SPN. В этом случае планировщик выбирает процесс с наименьшим ожидаемым временем до окончания процесса. При присоединении нового процесса к очереди готовых к выполнению процессов может оказаться, что его оставшееся время в действительности меньше, чем оставшееся время выполняемого в настоящий момент процесса. Планировщик, соответственно, может применить вытеснение при готовности нового процесса. Как и при использовании стратегии SPN, планировщик для корректной работы функции выбора должен оценивать время выполнения процесса; в этом случае также имеется риск голодания длинных процессов. В случае использования стратегии SRT таких больших перекосов в пользу длинных процессов, как при использовании стратегии FCFS, нет; в отличие от стратегии кругового планирования здесь не генерируются дополнительные прерывания, что снижает накладные расходы. С другой стороны, в этом случае необходимость фиксировать и записывать время выполнения процессов приводит к увеличению накладных расходов. В связи с тем что короткие задания немедленно получают преимущество перед выполняющимися длинными заданиями, стратегия SRT существенно выигрывает у стратегии SPN во времени оборота.
5. **HRRN -- Highest Response Ratio Next** -- наивысшее отношение отклика. Для того, чтобы понять как работает данная стратегия введем некоторые понятия. Так как мы не можем знать в общем случае время обслуживания заранее, но можем оценить его либо на основе предыдущих выполнений, либо на основе информации, пользователем или задаваемой при настройке. Рассмотрим соотношение:

$$R = \frac{w + s}{s}$$

где

$R$  -- отношение отклика;

$w$  -- время, затраченное процессом на ожидание;

$s$  -- ожидаемое время обслуживания.

Если процесс будет немедленно диспетчеризован, его значение  $R$  будет равно нормализованному времени оборота. Заметим, что минимальное значение (равное 1)  $R$  при нимает при входе процесса в систему. Таким образом, правило стратегии планирования наивысшего отношения отклика (highest response ratio next - HRRN) можно сформулировать так: при завершении или блокировании текущего процесса для выполнения из очереди готовых процессов выбирается тот, который имеет наибольшее значение  $R$ . Такой подход довольно привлекателен, поскольку учитывает возраст процесса. Короткие процессы получают преимущество перед продолжительным (в силу меньшего знаменателя, увеличивающего отношение), однако и увеличение возраста процесса приводит к тому же результату, так что в конечном счете длинные процессы смогут конкурировать с короткими. Как и в случае использования стратегий SRT и SPN, в описанной стратегии требуется оценка времени обслуживания для определения максимального значения  $R$ .

6. **Feedback** -- снижение приоритета в зависимости от длительности времени исполнения. Если у нас нет никаких указаний об относительной продолжительности процессов, то мы не можем использовать ни одну из рассмотренных стратегий, SPN, SRT или HRRN. Еще один путь предоставления преимущества коротким процессам состоит в применении штрафных санкций к долго выполняющимся процессам. Другими словами, раз уж мы не можем работать с оставшимся временем выполнения, мы будем работать с затраченным до настоящего момента временем. Вот как этого можно достичь. Выполняется вытесняющее (по квантам времени) планирование с использованием динамического механизма. Процесс при входе в систему помещается в очередь RQ0 (см. рис. 9.4). После первого вытеснения и возвращения в состояние готовности процесс помещается в очередь RQ1. В дальнейшем при каждом вытеснении этот процесс вносится в очередь со всем меньшим и меньшим приоритетом. Соответственно, быстро выполняющиеся короткие процессы не могут далеко зайти в иерархии приоритетов, в то время как длинные процессы постепенно теряют свой приоритет. Таким образом, новые короткие процессы получают преимущество в выполнении перед старыми длинными процессами. В рамках каждой очереди для выбора процесса используется стратегия FCFS. По достижении очереди с наиболее низким приоритетом процесс уже не покидает ее, всякий раз после вытеснения попадая в нее вновь (таким образом, эта очередь,

по сути, обрабатывается с использованием кругового планирования).

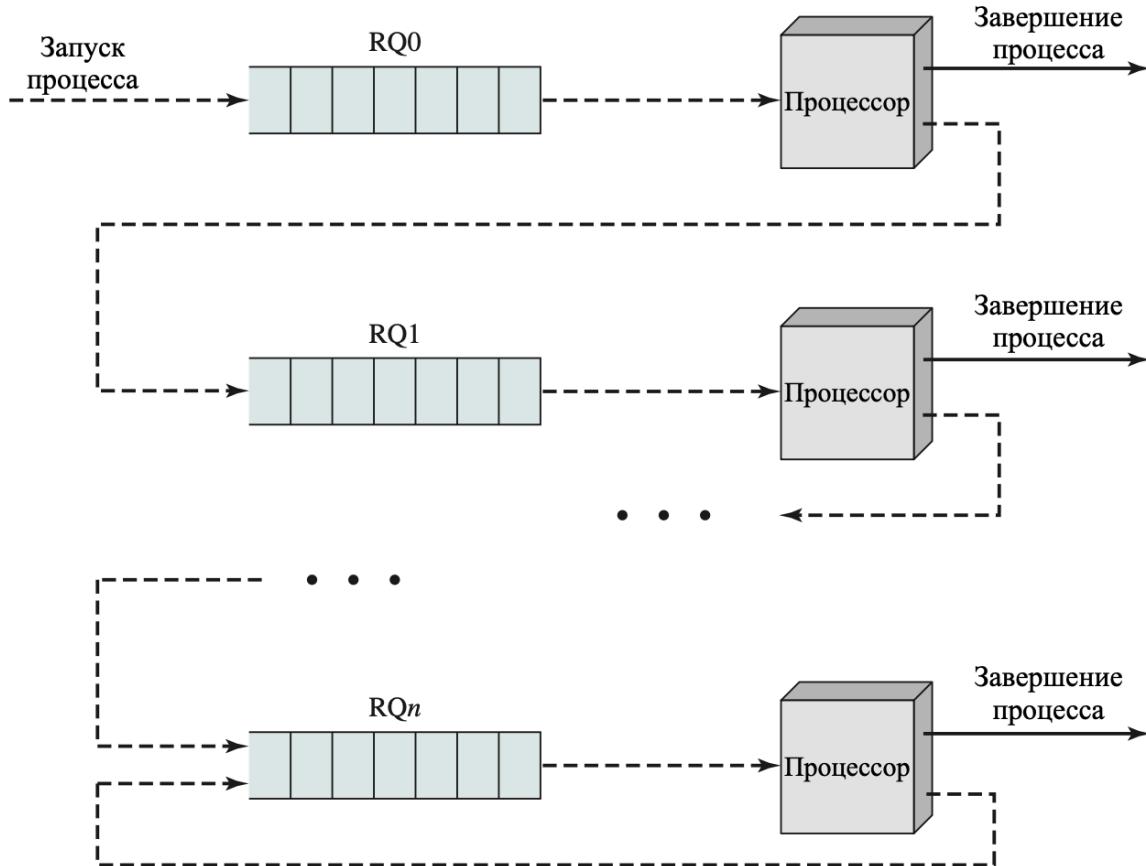


Рис. 9.10. Планирование со снижением приоритета

## 52. Feedback планировщик и классы планирования ОС UNIX SVR4.

Feedback on SVR4 (Solaris)  
TimeSharing class

globpri	quantum	tqexp	slpreat	maxwait	lwait
59	2	49	59	32000000	59
58	4	48	58	0	59
57	4	47	58	0	59
...					
40	4	30	55	0	55
...					
30	8	20	53	0	53
29	12	19	52	0	52
...					
21	12	11	52	0	52
20	12	10	52	0	52
19	16	9	51	0	51
...					
12	16	2	51	0	51
11	16	1	51	0	51
10	16	0	51	0	51
9	20	0	50	0	50

Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

globpri - глобальный приоритет,  
quantum - выделяемый квант времени (мс),  
tqexp - приоритет, если квант полностью выбран CPU,  
slpreat - приоритет, если ожидаем ресурса  
maxwait — граничное время длинного ожидания (мс)  
lwait - приоритет, в случае длинного ожидания ресурса

Feedback-стратегия классическим образом реализована в Solaris и System V Release 4.0.

В том и в другом случае вся механика примерно одинакова). Работает это так. Каждый процессор внутри Solaris имеет свои собственные очереди ожидания. На схеме их показано 59 (но на самом деле их больше, т. к. для каждого приоритета существует своя очередь).

И внутри каждой очереди используется принцип FIFO. Это означает, что процессы с одинаковым приоритетом выполняются последовательно друг за другом.

Данный пример рассматривает классический вариант однопроцессорного планирования, хотя стратегия Feedback применима и для многопроцессорного варианта.

После того, как процесс выполнился на процессоре, он "уходит" в очередь ожидания или на блокировки.

Вся стратегия диспетчеризации представлена в виде большой таблицы, кусочек из которой приведен на этом слайде.

- Первый столбец таблицы: **globpri** -- глобальный приоритет. Минимальное значение -- 0, максимальное значение 169 (с 160-169 используются приоритеты для высокоприоритетных прерываний которые надо обработать)
- Второй столбец таблицы: **quantum** - то есть каждому приоритету таблицы присвоен свой квант времени (мс)
- Третий столбец таблицы: **tqexp** -- приоритет, который будет присвоен процессу, если полностью исчерпать квант времени.
- Четвертый столбец таблицы: **slpreat** -- приоритет, когда процесс в момент диспетчеризации находится в ожидании ресурса.
- Пятый столбец таблицы: **maxwait** -- граничное время ожидания ресурса (мс) (в современных версиях почти не используется)
- Шестой столбец таблицы: **lwait** -- приоритет, который будет присвоен процессу, если он превысит значение maxwait

Классы планирования в SVR4:

Класс приоритета	Глобальное значение	Последовательность планирования
Реальное время	159 • • • • 100	Первые
Ядро	99 • • 60	
Разделение времени	59 • • • 0	Последние

**Рис. 10.12.** Очереди диспетчера SVR4

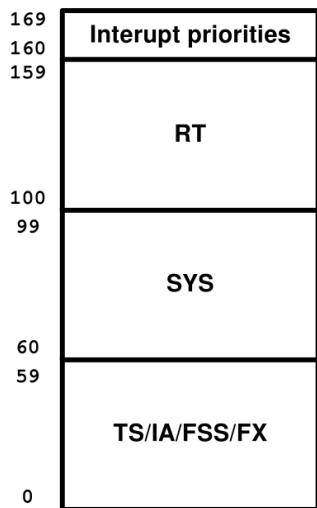
Вот краткое описание каждого класса приоритетов.

- **Реального времени (159-100).** Процессы этих уровней приоритета гарантированно выбираются для выполнения прежде любых процессов ядра и процессов с раз делением времени. Кроме того, процессы реального времени могут использовать точки вытеснения для прерывания выполнения процессов ядра и пользовательских процессов.
- **Ядра (99-60).** Процессы с этими уровнями приоритета гарантированно выбираются для выполнения прежде всех процессов с разделением времени, но уступают процессам реального времени.
- **Разделения времени (59-0).** Процессы с низшим приоритетом, принадлежащие пользовательским приложениям (кроме приложений реального времени).

Ответ на данный вопрос из лекции выглядел примерно так:



## Классы планирования SVR4



- TimeSharing — разделение времени (Feedback)
- InterActive — интерактивный (boost к активному приложению)
- Fixed, System, RealTime — классы с фиксированными приоритетами
- Fair Share Scheduler — справедливый планировщик
- Отдельные приоритеты для Interrupt threads
- Учет афинити для NUMA

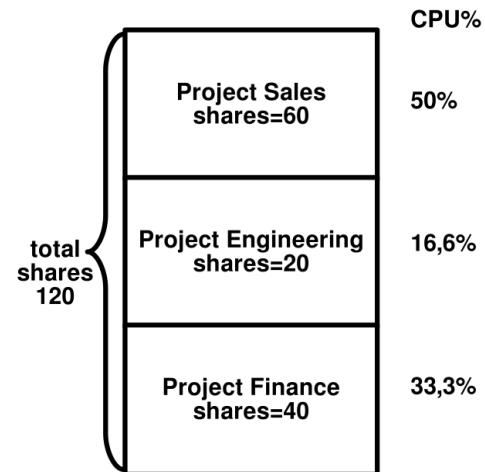
- interActive (разделения времени) -- так называемый интерактивный класс. Он достаточно простой и связан с графической подсистемой. И активному процессу (например, тому, куда мы постоянно вводим свои команды, или графическому фрейму) с его помощью удается повысить параметр boost, а с ним и свой приоритет. В этом случае к приоритету добавляется 10 (или около того). Соответственно, приоритет еще и «обрезается под рамки», чтобы не превышать 59. Если же приложение неактивно, то дополнительных уровней приоритета оно не получает, а значит, оно имеет обычный приоритет.
- Fixed (разделения времени), System (ядро) и RealTime (реального времени) -- это в сущности, одинаковые классы. В них лишь фиксируется приоритет для разных задач в зависимости от их важности. При этом программисты назначают им приоритеты. И, соответственно в фиксированном приоритете приоритет не меняется внутри класса. Каждому приоритету соответствуют свои кванты времени. На пользовательском уровне такие процессы тоже есть. Но они в явном виде запускаются администратором. И их всего несколько: NFS Demon, который работает с фиксированным приоритетом, и еще несколько программ. В случае же уровня ядра, там фиксированный приоритет активно работает на уровне ядра. в RealTime тоже широко используются фиксированные приоритеты.
- Fair Shair Scheduler (пользовательский уровень) -- справедливый планировщик.
- Также есть отдельные приоритеты для Interrupt threads.

### 53. Справедливое планирование.



## Справедливое планирование SVR4

- Выделяет время процессора на основании заданных user shares
- Используется как замена TS/IA классов
- Учитывает все процессоры в системе



Операционные системы. Часть 3. Память и планирование

...

Справедливое планирование -- концепция планирования, которая позволяет нам выделять время в процессоре пакетами (bundle).

Посмотрим на картинку в правой части. Предположим у некой организации есть большая вычислительная машина, на которой работают люди из различных отделов. При этом есть системные возможности, которые могут указать, какое значение виртуальных токенов (т.е. единиц процессора) могут получить пользователи (shares). Причем -- это просто число. Но значение имеет не само число, а сумма всех таких числе в системе.

Если продавцам будет дано 60 shares, инженерам 20 shares, финансистом 40 shares, это значит будет использоваться общее количество shares -- 120. Потом определим долю времени, которое будет выдаваться на каждое направление из этих "projects". Соответственно 50%, 16,6%, 33,3%.

Если заданий у группы "Engineering" в какой-то момент нет, то все остальные группы в системе "тратят" неиспользованное инженерами объем времени, справедливо деля его между собой в соответствие от начального распределения нагрузки. А как только от "инженеров" придут задачи, то они сразу же отберут свою долю от общей процессорной мощности, включая все процессоры.

В Solaris это используется в виде замены TimeSharing и InterActive классов. Поэтому, если происходит запуск этого планировщика, то он заменяет TimeSharing и InterActive классы.

Следует отметить, что в Solaris этот планировщик использовал "плавающую точку" внутри ядра, чего обычно стараются избегать. Поэтому, на некоторых моделях процессора, его не рекомендовалось использовать.

## 54. Планирование в многопроцессорных системах. Типы многопроцессорных систем с точки зрения организации планирования. Гранулярность и проектирование

# планировщиков процессов и потоков для многопроцессорных систем.



## Типы многопроцессорных систем

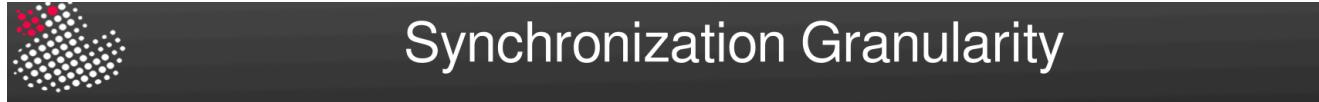
- Слабосвязанные (распределенные, кластеры)
  - Набор систем со своей основной памятью и подсистемой ввода-вывода
  - Распределение заданий между системами
  - Распределение общих данных и результатов
- Функционально-специализированные
  - Ведущий процессор выполняет координацию
  - Ведомые процессоры выполняют вычисления
- Сильносвязанные процессоры
  - Имеют общую память и систему кэшей
  - Управляются одной ОС

Типы многопроцессорных систем:

- Слабосвязанные -- это так называемые распределенные системы или "гриды" (grid), кластеры и так далее. В слабосвязанных системах приложение можно запустить на "гриде". При этом если оно запустится на нескольких разделенных узлах, то займет выделенное количество процессоров и будет при этом использовать общие данные.
  - Их особенностью является то, что они:
    1. Являются совокупностью отдельных систем со своей основной памятью и структурами ввода-вывода;
    2. Внутри этого кластера устройств должно быть распределение заданий между системами (либо автоматическое, либо ручное, для этого и нужен планировщик)
    3. Должно быть обеспечено распределение общих данных и результатов (то есть должно быть решено, где конкретно хранятся общие данные и результаты)
- Функционально-специализированные -- это системы, которые предназначены для выполнения какой-то конкретной функции. Например, это GPU -- для отображения графической информации или какого-нибудь другого специализированного процесса ввода-вывода.
  - Их особенностью является то, что они:
    1. Имеют специальный ведущий процессор, который осуществляет координацию и каким-то образом раздает задания;
    2. Имеют ведомые процессы, которые выполняют вычисления

- Сильносвязанные процессоры -- такие системы, которые обычно имеют общую память и сильную скорость передачу данных по сравнению с слабосвязанной
  - Их особенностью является то, что они:
    - Имеют общую память и систему кэшей -- например, архитектура NUMA
    - Управляются всегда в рамках одной ОС в отличие от двух предыдущих, которые могут управляться разными системами.

**Гранулярность:**



- Гранулярность синхронизации — частота синхронизации между процессами в системе
- Fine (тонкая) — параллельные вычисления на уровне отдельных машинных команд
  - Менее 20 команд
- Medium (средняя) — на уровне одного приложения
  - 20-200 команд
- Coarse (грубая) — на уровне взаимодействующих процессов
  - 200-2000 команд
- Very Coarse (очень грубая) — на уровне распределенных систем
  - 2000-1 000 000
- Independent (независимая) — нет синхронизации, независимые процессы

В целом думаю слайд хорошо описывает понятие гранулярности. Добавлю лишь то, что с помощью него достаточно удобно оценивать степень распараллеливания процесса и его диспетчеризации.

- Fine (тонкая) -- в реальной жизни тонкая гранулярность требуется лишь в специфических приложениях, таких как игры, когда создается многопоточная программа, которая обрабатывает один массив данных, и при этом треду назначается обрабатывать конкретную строку и столбец.
- Medium (средняя) -- характерна для обычных приложений.
- Coarse (грубая) -- уровень гранулярности применяется, когда у нас есть разные процессы, которые должны между собой взаимодействовать (например через разделяемую память (*shared memory*)).
- Very coarse (очень грубая) -- применяется для распределенных систем, когда у нас один кластер посыпает сообщения другому кластеру, поэтому синхронизация нужна редко.
- Independent (независимая) -- характерна для независимых систем, в которых синхронизация отсутствует. Но в этих независимых системах может присутствовать синхронизация на уровне операционной системы.

**Проектирование планировщиков процессов и потоков для многопроцессорных систем**



# Вопросы проектирования планировщиков в случае (сильно) многопроцессорных систем

- Назначение процессов процессорам
  - Статическое — выделяем процессор для процесса
  - Динамическое — общая очередь для всех процессов
  - Динамическая балансировка нагрузки
- Использование многозадачности на отдельных процессорах
  - Нужна ли для статического назначения многозадачность?
- Диспетчеризация процесса
  - Так ли плох будет FCFS? Влияние выбора алгоритма диспетчеризации снижается.

Когда система содержит большое количество процессоров, то появляются при проектировании планировщиков совершенно другие вопросы по сравнению с однопроцессорными системами. Происходит это все потому, что количество процессоров и задач, которые поступают на такую многопроцессорную систему, могут существенно изменить подходы, использующиеся внутри планировщика.

В итоге мы приходим к следующим заданиям:

- Назначение процессов процессорам. В случае с однопроцессорными системами нам все время приходилось думать как делить процессы между одним процессором, но в случае с многопроцессорными системами делить особо ничего и не нужно. Это не означает, что ресурсы совсем разделять не нужно, а просто имеется возможность выделение процессов процессорам для решения конкретной задачи. Поэтому мы приходим к следующему распределению процессов:
  1. Статическое -- выделяем процессор для одного процесса целиком. Это имеет ряд своих преимуществ, так как снижается количество переключений контекста и не будет остыивание кэшей, что сильно повышает производительность.
  2. Динамическое -- существует общая очередь для всех процессов, далее процессы распределяются по процессорам. У этого подхода есть минусы. Например, это замедляет систему, если процессы связаны друг с другом.
  3. Динамическая балансировка нагрузки -- системы, которые позволяют во время работы на многопроцессорной системе осуществлять переносы процессов с одного процессора на другие процессорные "ноды".
- Использование многозадачности на отдельных процессорах.
  - Соответственно мы приходим к вопросу, а нужно ли для статического назначения многозадачность? Если раньше нам приходилось вынужденно

использовать многозадачность, то сейчас мы можем использовать процессоры как ресурсы для отдельных задач (или группа процессоров для группы задач)

- Диспетчеризация процесса.
  - Если раньше нужно было думать, какой алгоритм планирования процессов использовать, то сейчас уже нет большой разницы условно между FCFS (first come first serve ) и RR (round robin) так как процессоров много



## Подходы к планированию потоков

- Load Sharing — глобальная очередь потоков
  - Равномерное распределение нагрузки
  - Нет централизованного планировщика
  - Минусы — блокировки центральной очереди, промахи кэшей, неэффективность при тонкой средней гранулярности
- Gang scheduling — связанные потоки распределяются на связанные процессы по одному на процессор
  - Снижение накладных расходов на планирование при тонкой и средней гранулярности
- Dedicated processor assignment — назначается пул процессоров равный количеству потоков
  - В экстремальном случае увеличивает простой процессора
  - Полное устранение переключений повышает скорость работы
- Динамическое планирование
  - В отсутствии свободных CPU ресурсы изымаются из процесса, использующего несколько CPU

Операционные системы. Часть 3. Память и планирование



Существует несколько подходов к созданию планировщиков:

Load Sharing -- глобальная очередь потоков.

- Плюсы:
  1. Равномерно распределяет нагрузку между всеми процессорами
  2. В современных ядрах он реализован без специального модуля планировщика (то есть процессы и потоки диспетчериизуют сами себя)
- Минусы:
  1. Блокировки центральной очереди. Так как процессоров и процессов будет много, то блокировка будет заниматься для этого достаточно часто
  2. Промахи кэшей. Если придется диспетчerezировать процессы в глобальной очереди, то неизбежны частые промахи кэшей.
  3. Совсем неэффективны в системах при тонкой и средней гранулярности.
- Gang scheduling -- связанные потоки распределяются на связанные процессы по одному на процессор.
- Плюсы:
  - При использовании данного подхода идет сильное снижение "накладных расходов" при тонкой и средней гранулярности. Это происходит, потому что процессоры работают на одних кэшах, так как они не передвигаются в

рамках одной группы процессоров, связанных с одной нодой NUMA, поэтому и будут использовать один кэш. Лишние обращения к памяти уменьшаются.

Dedicated processor assignment -- назначается пул процессоров равный количеству потоков.

- Минусы:
  - Увеличиваются простои процессора, так как потоки могут быть в заблокированном состоянии, в эти моменты процессоры будут простаивать.
- Плюсы:
  - Полное устранение переключений, имеющее место при этом подходе, сильно повышает производительность, потому что каждый процесс будет выполняться за минимальное время работы системы (как в случае однопроцессорной системы)

Динамическое планирование. Суть его заключается в том, что если планировщик уже распределил процессоры между процессами, то в случае поступления новой задачи и отсутствия свободных ресурсов (когда процессоры закончились), свободные новые ресурсы могут изыматься от процессов, которые используют несколько CPU (например если процесс использует 3 CPU, то один процессор у него забирают и отдают вновь появившемуся процессу).

## 55. ОС реального времени и планировщики. Deadline-планирование.



- Hard & Soft realtime, Периодичность работы
- Основные Требования
  - Determinism — выполнение операций в предопределенный интервал времени. Мера: время от прерывания до начала его обработки.
  - Responsiveness — сколько времени требуется для ответа?
  - User control — управление планированием со стороны пользователя
  - Reliability — повышенные требования по сравнению с «обычными» ОС
  - Fail-soft operation — мягкая реакция на ошибки (не kernel dump)
- Планировщик: строгое использование приоритетов с вытеснением и ограниченные, минимальные задержки

Существует 2 типа систем реального времени -- Hard и Soft realtime.

Hard RealTime -- это системы с достаточно строгими требованиями. Если какая-то расчетная задача не будет выполнена в реальное "физическое время", то могут произойти неисправимые последствия (например, падение ракеты и т.д.).

Soft RealTime -- системы с уже не такими строгими требованиями, как Hard. Если же процесс выйдет за рамки, заданного ему ограничения, то ничего страшного не случится, хотя их стараются все равно придерживаться.

Важной характеристикой систем реального времени является преодичность работы. Понятие преодичности означает, что все реальное время разделено на отрезки, по окончании которых должны проводиться проверки чего-либо и приниматься какие-то решения.

Основные требования, которые определяются к системам реального времени:

- Determinism -- четко определяющий конкретное время того, когда произошло прерывание, которое сообщает о событии до его обработки. Иногда еще и фиксируют конечный момент времени, не позже которого должно произойти прерывание;
- Responsiveness -- реактивность ответа. Время, которое необходимо от получения сигнала, до выработки ответа;  
**Детерминизм и чувствительность образуют в целом время отклика системы.**
- User Control -- предъявляет требования к операционным системам о том, что пользователь должен достаточно тонко задавать характеристики процессов, чтобы планировщик мог понять, каким образом их лучше спланировать. При этом следует иметь в виду, что планировщики в системах реального времени поддерживают большое количество параметров, которые пользователь может изменить;
- Reliability -- надежность. Это отдельные специальные требования, учитывающие то, что любая система реального времени работает с реальными объектами. И поэтому она должна быть более надежна, чем обычная ОС.
- Fail soft operation -- мягкая реакция на ошибки. Означает, что если происходит ошибка, то операционная система не должна "впадать в панику".

Планировщик: строгое использование приоритетов с вытеснением и ограниченные, минимальные задержки. Можно добавить, что задержки в ОС реального времени обычно измеряются в микросекундах (а не в миллисекундах).



## Deadline планирование

- Важены своевременные завершение или начало выполнения задания
  - Конфликты, сбои и временные недостатки ресурсов не должны влиять
- Задания могут включать дополнительную информацию:
  - Ready time — время готовности задания к выполнению
  - Starting deadline — предельное время начала выполнения
  - Completion deadline — предельное время полного завершения задания
  - Processing time — время, необходимое для полного выполнения задания
  - Resource requirements — список ресурсов (не процессор) для задания
  - Priority — мера важности задания
  - Subtask structure — обязательные и необязательные задачи
- Rate Monotonic Scheduling — см. Столлинг гл. 10.2

Deadline планирование или планирование с предельными сроками. В ОС реального времени не столь важно время выполнения задания, как его своевременное завершение (или начало) -- не слишком рано и не слишком поздно, несмотря на любые требования к ресурсам и могущие возникать конфликты, перегрузку процессора или аппаратные либо программные сбои. То есть возникают вопросы быстрой обработки прерываний и диспетчеризации заданий.

Задания могут включать дополненную информацию:

- **Время готовности.** Время, когда задание становится доступным для выполнения. В повторяющемся или периодическом задании время готовности представляет собой последовательность заранее известных времен. В непериодическом задании это время может быть известно заранее, но может быть и так, что операционная система узнает его только в тот момент, когда задание станет действительно готовым к выполнению.
- **Предельное время начала выполнения.** Время, когда должно начаться выполнение задания.
- **Предельное время завершения выполнения.** Время, когда задание должно быть полностью завершено. Обычно задания реального времени имеют ограничение — либо по предельному времени начала выполнения, либо по предельному времени завершения выполнения, но не оба ограничения одновременно.
- **Время выполнения.** Время, необходимое заданию для полного выполнения. В некоторых случаях это время известно, а в некоторых система сама оценивает взвешенное среднее значение. В других системах планирования эта величина не используется.
- **Требования к ресурсам.** Множество ресурсов (отличных от процессора), требующихся заданию при его выполнении.
- **Приоритет.** Мера относительной важности задания. Жесткие задания реального времени имеют "абсолютный" приоритет, приводя к сбою системы при нарушении временных ограничений этих заданий. Если система продолжает работу несмотря ни на что, то как жесткие, так и мягкие задания получают относительные приоритеты, использующиеся в качестве указаний планировщику.
- **Структура подзадач.** Задача может быть разбита на обязательные и необязательные подзадачи. Жесткие предельные сроки при таком разделении имеют только обязательные подзадачи.

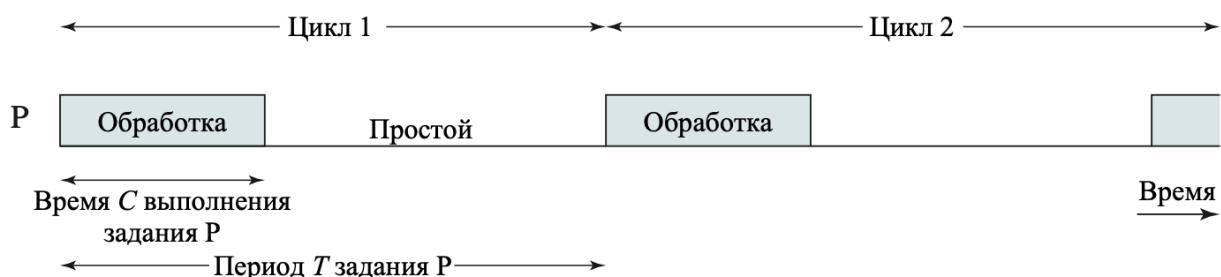
**Rate Monotonic Scheduling** (частотно-монотонное планирование):

RMS -- это метод разрешения конфликтов многозадачного планирования для периодических задач. **Схема RMS назначает приоритеты задачам на основе их периодов. Чем меньше период -- тем выше приоритет. Когда для выполнения готово более одного задания, первым обслуживается задание с кратчайшим периодом.** Если изобразить приоритет заданий как функцию их частоты, результатом оказывается монотонно растущая функция - откуда и название "частотно-монотонное планирование". Период задания Т представляет собой интервал времени между поступлениями двух последовательных экземпляров заданий одного типа. Частота заданий (измеряемая в герцах (Гц)) пред ставляет собой величину, обратную периоду (в секундах).

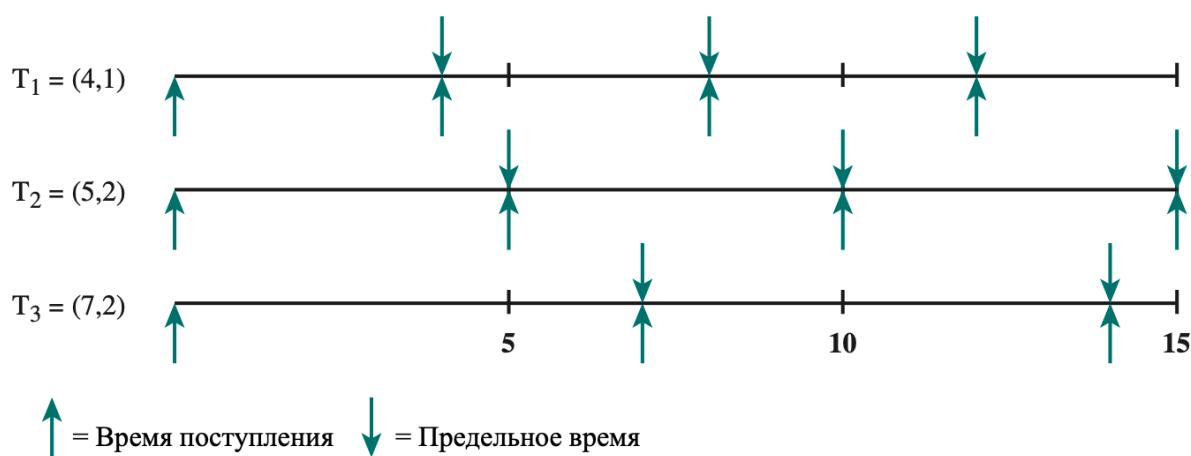
### Условия применимости RMS:

- Все задачи **периодические** и известны заранее.
- Время выполнения (execution time) каждой задачи меньше или равно её периоду.
- Задачи независимы (не блокируют друг друга и не синхронизируются).
- Задачи выполняются на одном процессоре.
- Никакого переключения контекста, вызванного внешними прерываниями.

На рис. 10.8 показан простой пример RMS. Экземпляры заданий пронумерованы последовательно. Как можно видеть, для задания 3 второй экземпляр не выполняется, поскольку предельное время просрочено. Третий экземпляр испытывает вытеснение, но все равно остается в состоянии завершиться до предельного срока завершения.



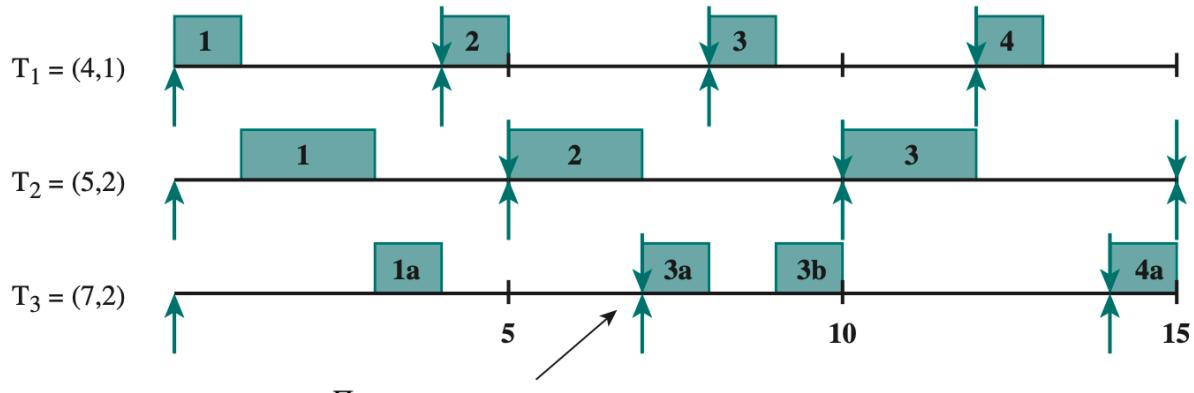
**Рис. 10.7. Временная диаграмма периодического задания**



(а) Времена поступления и предельные времена завершения задания  $T_i = (P_i, C_i)$ ;  
 $P_i$  = период,  $C_i$  = время обработки

↑ = Время поступления      ↓ = Предельное время

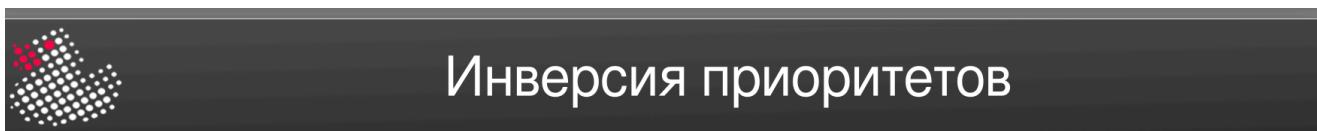
(а) Времена поступления и предельные времена завершения задания  $T_i = (P_i, C_i)$ ;  $P_i$  = период,  $C_i$  = время обработки



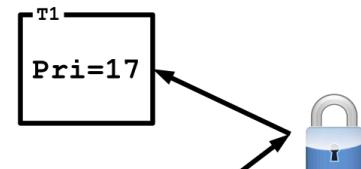
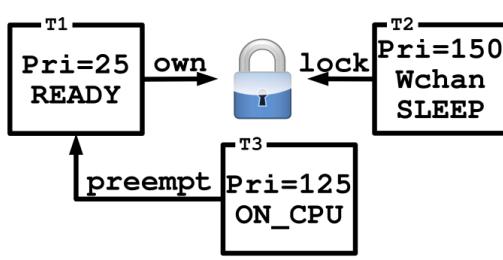
(б) Результаты планирования

Рис. 10.8. Пример работы RMS

## 56. Проблема инверсии приоритетов, типы инверсии и способы решения в планировщике.



- Решается при помощи наследования приоритетов



**Инверсия приоритета** представляет собой явление, которое может произойти в любой схеме планирования на основе приоритетов с вытеснением, но особенно актуально в контексте планирования реального времени. **Инверсия приоритета** (priority inversion) проходит, когда система заставляет задание с высоким приоритетом ожидать задание с низким приоритетом. Простой пример инверсии приоритета возникает, когда задание с низким приоритетом блокирует ресурс (например, устройство или бинарный семафор), и задание с более высоким

приоритетом также пытается заблокировать этот же ресурс. Высокоприоритетное задание оказывается в заблокированном состоянии до тех пор, пока ресурс не станет доступным. Если низкоприоритетное задание вскоре завершает работу с ресурсом и освобождает его, высокоприоритетные задачи могут быстро возобновить работу, и вполне возможно, что никакие ограничения реального времени не будут нарушены.

Есть несколько типов инверсий приоритетов:

- **Bounded priority inversion** (ограниченная инверсия приоритетов) -- это не самый плохой, но все же неприятный исход при планировании с вытеснением. Легче всего пояснить её на рисунке презентации. Представим, что у нас есть 2 процесса T1 и T2. Приоритет второго процесса значительно выше, но может случиться так, что процесс T1 заблокирует общий ресурс, хоть и на короткое время, то при вытеснении его процессом T2 он не сможет захватить данный ресурс, так как сейчас он заблокирован процессом T1. Поэтому процессу T2 придется ждать пока процесс T1 не освободит данный ресурс.
- **Unbounded priority inversion** (неограниченная инверсия приоритетов) -- это уже более серьезная ситуация, в которой продолжительность инверсии приоритетов зависит не только от времени, необходимого для обработки совместно используемого ресурса, но и от непредсказуемых действий других, не связанных задач. То есть в примере на слайде мы видим, что появился процесс T3, чей приоритет выше, чем у T1 и ниже, чем у T2, при этом ему не нужен доступ к общему ресурсу. В этом случае процесс T1 будет вытесняться процессом T3 до тех пор, пока T3 сам не уйдет в блокировку. В итоге процесс T1 будет находиться в состоянии готовности и ожидать момента, когда он сможет продолжить выполняться, тем самым блокируя процесс T2 на неограниченный срок.  
Есть классный пример в столлингсе, начиная со страницы 569, но в целом суть ясна).

Способы решения проблемы в планировщике:

1. **Наследование приоритетов.** Основная идея наследования приоритетов (*priority inheritance*) заключается в том, что задача с более низким приоритетом наследует приоритет любой высокоприоритетной задачи, ожидающей совместно используемый ресурс. Это изменение приоритета происходит, как только более высокоприоритетное задание блокируется в ожидании ресурса; оно должно заканчиваться, когда ресурс освобождается задачей с более низким приоритетом. То есть для нашего примера, процесс T1 временно получит приоритет не меньше, чем процесс T2.
2. **Потолок приоритетов.** В подходе с потолком приоритета с каждым ресурсом связан приоритет. Приоритет, назначенный ресурсу, на один уровень выше приоритета его наиболее приоритетного пользователя. Затем планировщик динамически назначает этот приоритет любой задаче, которая обращается к

ресурсу. Когда задание заканчивает работу с ресурсом, его приоритет возвращается в нормальное состояние.

3. Возможно также решение с помощью примитивов синхронизации: мьютексов, семафоров, read-write lock-ов и так далее, потому что может быть случай с цепочкой инверсией приоритетов, которые также нужно решать.

## **57. Ввод-вывод. Современные устройства и скорости обмена, развитие способов ввода- вывода, логическая структура ввода-вывода.**

---

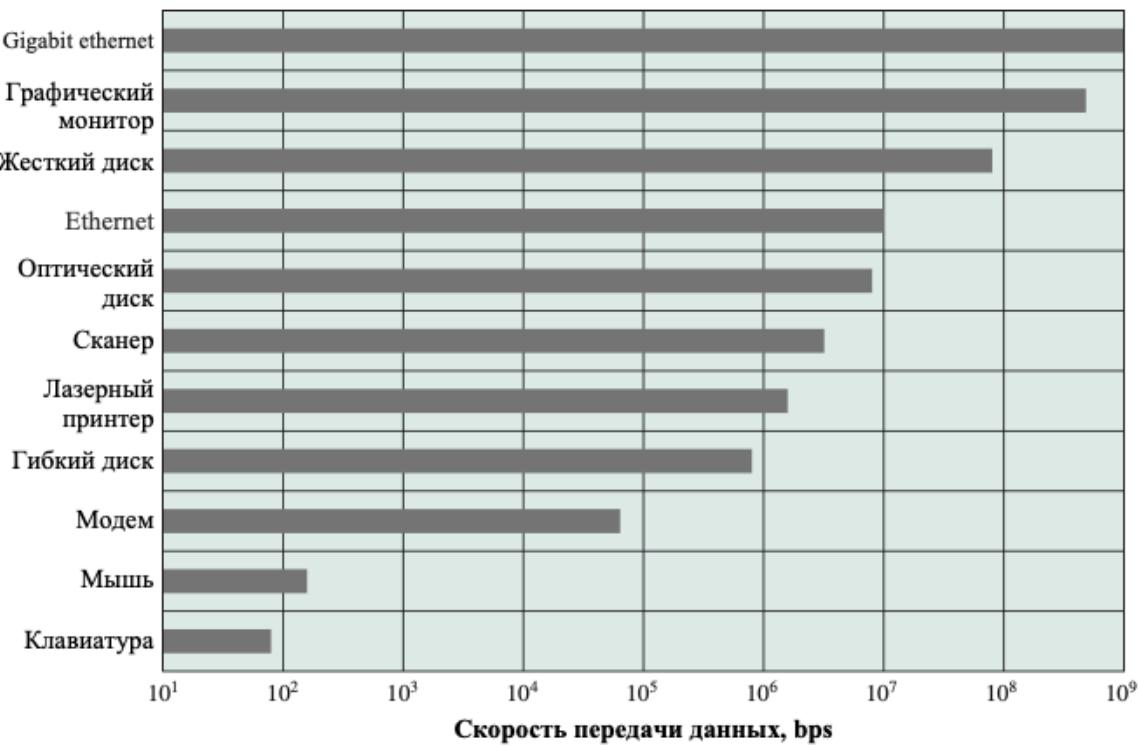
**Устройства делятся на**

- **Пользовательские.** Внешние устройства такие как мышь, клавиатура, дисплей
- **Внутренние.** Связь с электронным оборудованием. Флешка, контроллер какой-то, датчики и тд. Подключаются драйверами.
- **Коммуникация.** Для связи с удаленными устройствами. Модем, сетевые драйверы

Задача ОС -- обеспечивать нормальную работу как с быстрыми устройствами так и с медленными не вызывая томожения системы.

**Характеристики устройств**

- **Скорость передачи данных.** Как внутри одного класса устройств, так и между разными классами
- **Применение**
- **Сложность управления.** Для принтера и диска требуются разные контроллеры и существенно разные по сложности.
- **Единица передачи данных.** Как поток байтов так и блоками
- **Представление данных.** Разные схемы кодирования, кодировка символов и контроллер четности
- **Условия ошибок.** Причины возникновения, сообщения об ошибке и ее решение сильно отличаются переходом от устройства к устройству.



Все это нам говорит о том, что нельзя построить единый подход к работе с разными устройствами.

## Развитие способов ввода-вывода

- **Программируемый ввод-вывод.** Процессор посылает необходимые команды контроллеру, после этого процесс находится в состоянии ожидания завершения ввода-вывода. Или позже появились контроллеры, задача которых была сопряжение быстрой вычислительной шины с медленной шиной ввода-вывода. Сами контроллеры пытались унифицировать.
- **Ввод-вывод с прерываниями.** В контроллер добавили перерывания, исключая спин-лупы и ожидания. Таким образом нам не надо будет ждать постоянно готовность данных от внешнего устройства и мы можем прерывать процесс только при необходимости.
- **Прямой доступ к памяти**
  - В контроллере добавляются регистры и счетчики для обеспечения переноса области буфера в контроллере в область памяти. То есть сам контроллер занимался вводом-выводом без участия процессора. Регистры и счетчики нужны для того, чтобы понять в какую область памяти мапить буфер контроллера
  - Контроллер превращается в отдельный вычислительный модуль с процессором и системой команд
  - В целом такие контроллеры могут называться процессорами ввода-вывода
  - Модуль ввода-вывода может обладать также своей памятью и уже будет являться отдельным компьютером. При такой конфигурации почти все устройства внешние могут управляться с минимальным участием процессора

## **Логическая структура ввода-вывода**

При проектировании системы ввода-вывода стоит учитывать два пункта:

универсальность и эффективность

Полной универсальности достичь невозможно, но можно как-то объединять и унифицировать процесс ввода-вывода для разных, но похожих устройств

**Есть несколько структур в зависимости от устройства**

**Локальное устройство, связь посредством потока байтов или записей**

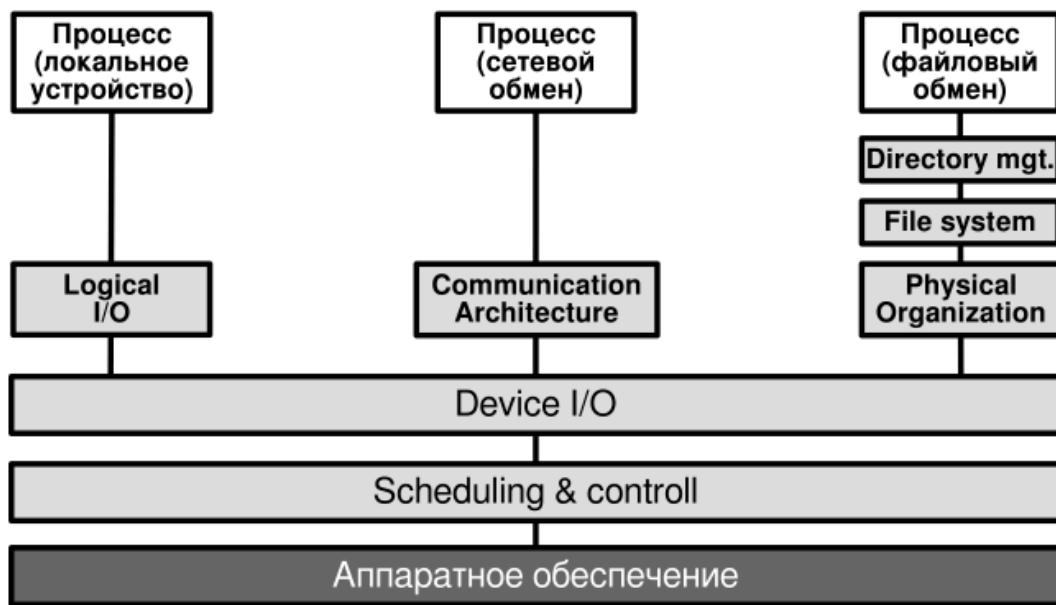
- **Логический ВВ** -- модуль, который общается с устройством как с логическим ресурсом и не обращает внимания на фактические детали реализации. Работает посредником между процессом и устройством позволяя процессам работать с устройством посредством ID устройства и простых команд записи, чтения и тд
- **Устройство ВВ** -- Запрошенные операции и данные конвертируются в инструкции ВВ, команд управления каналами и команд контроллера. Может применяться буферизация
- **Планирование и контроль** -- реальная организация очередей и планирование ВВ. Работа с прерываниями и непосредственно с аппаратным обеспечением

**Сетевой обмен**

- Принципиальное отличие от прошлого это замена логического ВВ на коммуникационную архитектуру, которая может состоять из многих уровней. К примеру TCP/IP
- Очень сложная система для поддержки многих протоколов
- При обновлении новых более быстрых компонентов эта часть постоянно переписывается для поддержки скорости обработки

**Файловый обмен**

- **Управление каталогами** -- Преобразование символьных имен файлов в ID, указывающие на файл как косвенно с помощью индексной таблицы так и напрямую. Эта часть связана с добавление каталога файлов, удалением и реорганизацией
- **Файловая система** -- работа с логической структурой файлов и операциями, указываемые пользователями такими как открытие, закрытие чтение запись + управление правами доступа
- **Физическая организация** -- логические ссылки на файлы должны быть преобразованы в физические адреса конкретных запоминающих устройств.



## 58. Буферизация ввода вывода. Ввод-вывод в UNIX SVR4

Существуют два типа устройств ВВ

- **Блочно-ориентированные.** Сохраняют и передают информацию блоками фиксированного размера. К примеру -- диски и USB
- **Поточно-ориентированные.** Передают данные в виде неструктурированных потоков байтов, без блочной структуры. К примеру -- терминале, принтеры, мышь и тд.

### Виды буферизации

#### Info

I - время ВВ

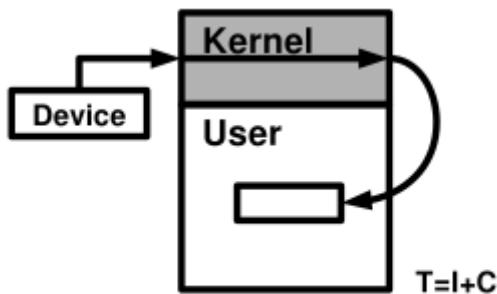
C - время обработки

M - время пересылки

Для блочных устройств нам надо решить стоит ли использовать буферизацию

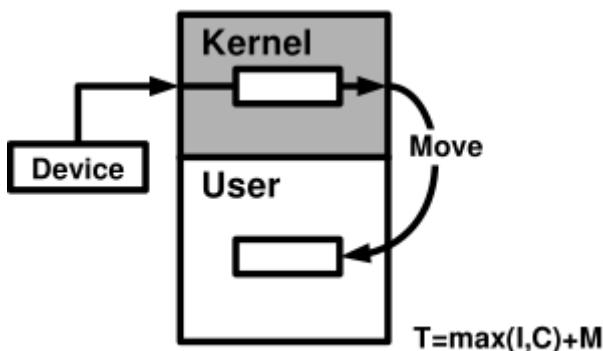
- **Без буферизации.** Программа пользователя запросила у ядра какой-то кусочек файла. Ядро превратило кусок файла в блок с каким-то номером на жестком диске. Ядро запросило эту информацию у драйвера устройства и далее по стрелке при помощи DMA в адресное пространство юзера записался этот блок.
- Такой метод называется Direct IO и он максимально быстро отдает данные приложению.

- Может возникнуть ситуация в многозадачной системе, когда страница, которая запросила блок пойдет в swap и окажется, что данные с диску некуда просто загружать. Решение проблемы можно сделать так: заблокировать страницу, чтобы ОС не имела права выгружать ее в swap пока не придет ответ от устройства. Однако когда таких процессов много, то в какой-то момент нам нечего будет класть в swap и система начнет стоять.
- Такой метод распространен в БД



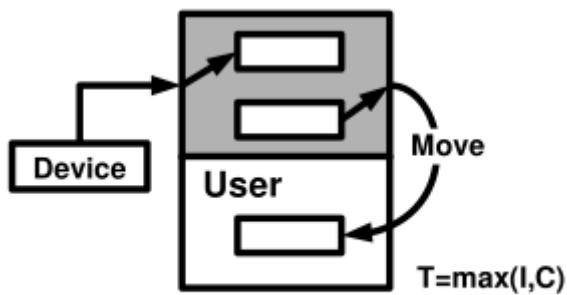
а) без буферизации

- **Одиночная буферизация.** В момент когда пользовательский процесс выполняет запрос ВВ, ОС назначает ему буфер в системной части основной памяти.
- Процесс выглядит так: пересылка данных в системный буфер. Когда заврешается эта операция, то происходит пересылка данных в пользовательскую часть памяти и мы сразу можем во-первых продолжать исполнение процесса, который вызвал ВВ и во-вторых сразу может начаться чтение следующего блока данных для другого процесса или этого же.
- Процесс юзера может обрабатывать один блок данных в то время как считывается другой блок и при этом выгрузка процесса может происходить спокойно без препятствий
- Для символьных устройств такой вид буферизации также подходит



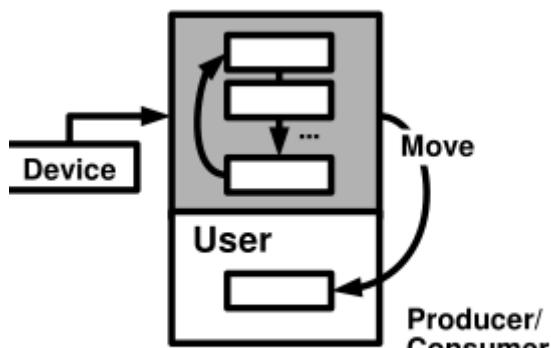
б) Одиночная буферизация

- **Двойная буферизация.** Теперь процесс выполняет передачу данных в один буфер (или считывание из него), в то время как операционная система освобождает (или заполняет) другой.
- Скорачиваем время работы (не понял почему так), но при этом растут требования к размеру системной памяти



в) Двойная буферизация

- **Циклический буфер.** При использовании более двух буферов схема именуется циклической буферизации. В ней каждый индивидуальный буфер представляет из себя модуль циклического буфера. Такая буферизация описывается схемой Produce/Consumer



г) Кольцевая буферизация

## Ввод-вывод в SVR4

В Unix каждое устройство ВВ представляется в виде файла и чтение/запись происходит также как и в любой другой пользовательский файл.

### Буферный кэш

Для управления буферным кэшем используется три списка

- Список свободных слотов
- **Список устройств.** Список всех устройств и их буфера
- **Очередь драйвера ВВ.** Список буферов, участвующих в операциях ВВ конкретного устройства

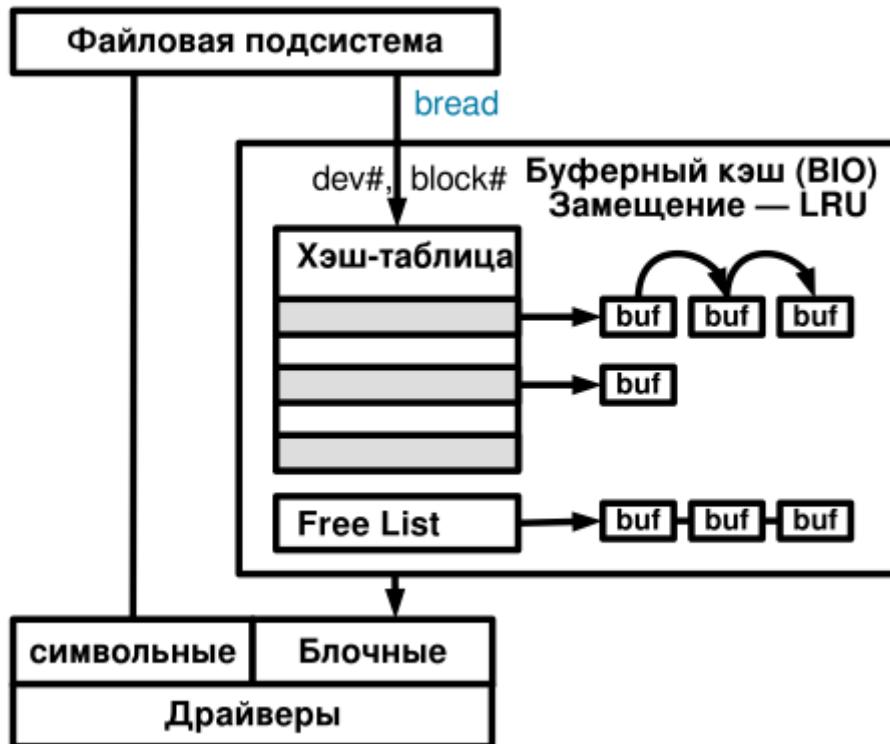
Буфер однажды назначенный устройству всегда ему принадлежит даже если находится в списке свободных до тех пор пока реально не будет нужен какому-то другому устройству.

Используется хэш таблица с ключем -- (номер устройства, номер блока)

**Для замещения используется LRU**

Для символьных устройств информация либо записывается в очередь символов устройством ВВ и считывается процессов, либо записывается процессом и считывается устройством. Используется producer/consumer для реализации такой схемы. Прочитанные символы сразу удаляются из буфера.

Небуферизированный ВВ использует DMA. Процесс выполняющий небуф. ВВ блокируется и не может быть выгружен. Так, уменьшается общая производительность системы. Кроме того устройство ВВ и процесс оказываются связанными до момента конца ВВ, делая его недоступным для других процессов



## 59. Диски и дисковое планирование

---

Есть два вида дисков: Hard Disk и SSD

В жестких дисках есть несколько блинтов или поверхностей, естьчитывающая головка и дорожки, которые разбиты на сектора. Сейчас используется SATA интерфейс и другие

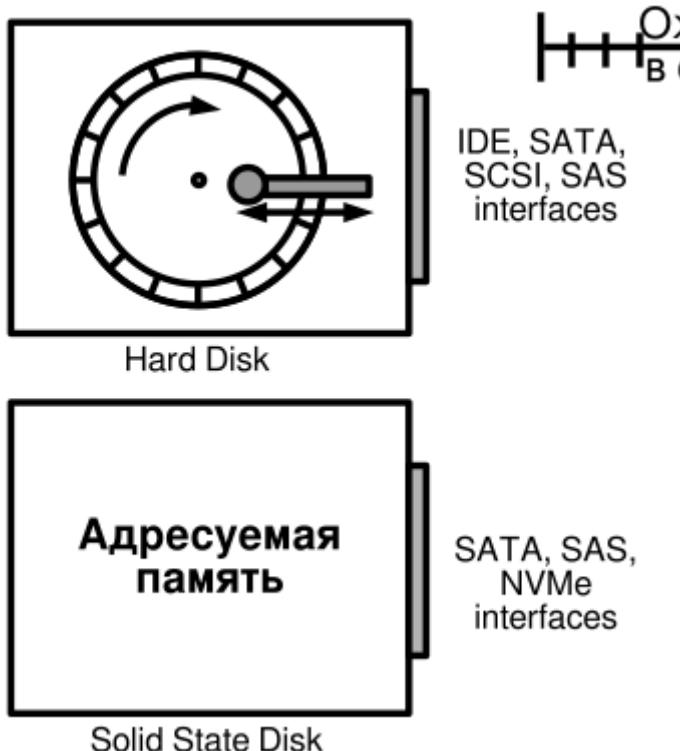
В SSD у нас есть простая адресуемая память без движущихся частей.

Жесткие диски хорошо справляются с последовательным чтением. Для случайного обращения к данным лучше использовать SSD.

### Для жестких дисков

- Время поиска -- время, которое затрачивается движущейся головкой на ее позиционирование на нужную дорожку
- Время задержки из-за вращения -- время которое тратиться, чтобы передвинуться на начало искомого сектора внутри дорожки.
- Сумма времени поиска и задержки вращения составляет время доступа  
Также есть такое понятие как позиционное считывание (rotational postitional sensing -- RPS) которая работает так: При выполнении команды поиска происходит освобождение канала для обработки других операций ввода-вывода.

После выполнения поиска устройство определяет момент, когда данные окажутся под головкой. Как только этот сектор подходит к головке, устройство пытается восстановить связь с узлом. Если либо контроллер, либо канал занят другой операцией ввода-вывода, то попытка восстановления связи оказывается неудачной и диск совершает полный оборот перед повторной попыткой.



### Время передачи данных

Время нахождения в очереди + время освобождения канала + время поиска + время задержки вращения + передача данных

У SSD будет всего лишь: время в очереди + время освобождения канала + еще какие-то другие параметры



$$T_d = T_o + T_k + T_n + \frac{1}{2r} + \frac{b}{rN}$$

- Тд — время доступа
- То — время ожидания в очереди ОС
- Тк — время освобождения канала
- Тп — время поиска
- r — скорость вращения RPM
- N — количество байт на дорожке
- b — количество байт в запросе

ATA, SAS,  
NVMe  
interfaces

## **Дисковое планирование**

### **FIFO**

- Справедливая стратегия
- Выгоден только для небольшого кол-ва процессов и запросах близких по секторам друг к другу, на большом превращается в случайный доступ

### **PRI (использование приоритетов)**

- Выгоден для самой ОС
- Такая система позволяет выполнить большое кол-во коротких и быстрых заданий, что хорошо для ОС, так как таким заданиям присваивается больший приоритет обычно.
- У больших заданий наблюдается слишком длительное ожидание выполнение дисковых операций.
- Не подходит для работы с БД

### **LIFO (last in-first out)**

- Задание размещеннное в очереди первое будет выполнены первым
- В системах обработки транзакций при предоставлении устройства для последнего пользователя должно выполняться лишь небольшое перемещение указателя последовательного файла. Использование преимуществ локальности позволяет повысить пропускную способность и уменьшить длину очереди.
- Если нагрузка на диск велика, то есть шанс голодания процесса

### **Учитывание текущего состояния диска (например дорожки)**

### **SSTF (Shortest Service Time First) -- наименьшее время обслуживания**

- Выбор такого дискового запроса на ВВ, который требует наименьшего перемещения головки из текущей позиции.
- Минимизация время поиска
- Дает лучший показатель чем FIFO

### **SCAN (elevator algorithm)**

- В ходе рассмотрения алгоритмов выше некоторые из них могут оставить какой-то запрос невыполненным из-за того, что в очереди могут появляться новые более выгодные запросы
- Избежать это можно посредством SCAN алгоритма или его называют алгоритмом лифта
- Премещение головки происходит только в одном направлении, удовлетворяя те запросы, которые соответствуют выбранному направлению. После достижения последней дорожки направления направление меняется на противоположное
- SCAN и SSTF дают похожие результаты.

- Нетрудно увидеть, что стратегия SCAN оказывает предпочтение тем заданиям, запросы которых относятся к дорожкам, находящимся ближе всего к центру либо наиболее удаленным от него, а также отдает предпочтение запросам, поступившим последними.

### C-SCAN (циклическое сканирование)

- Когда обнаруживается последняя дорожка то мы просто возвращаемся в начало откуда начали, а не идем в обратном направлении
- Максимальная задержка уменьшается, вызывая новыми запросами

### N-STEP-CSCAN

- При использовании SSTF, SCAN, C-SCAN может быть такое, что один или несколько процессов с высокой частотой обращений к одной дорожке монополизируют устройство за счет многочисленных повторений запросов к одной и той же дорожке
- Данная стратегия делит очереди запросов на подочереди длиной N. Каждая подочередь обрабатывается за один прием стратегией SCAN. В ходе обработки очереди могут добавиться еще запросы, если в конце текущей обработки запросов меньше чем N то они обрабатываются на следующем проходе.
- При больших значениях N производительность достигает производительности SCAN

### FSCAN

- Стратегия, использующая две подочереди. С началом сканирования все запросы находятся в одной из очередей; другая при этом остается пустой. Во время сканирования первой очереди все новые запросы попадают во вторую очередь. Таким образом, обслуживание новых очередей откладывается, пока не будут обработаны все старые запросы.

## 60. Концепция RAID

---

Раньше предложили в коробку вставить большое кол-во дисков и назвали это JBOD (just bunch of disk) и какая-то шина, которая позволяла подключать определенное кол-во дисков. Тогда для сервера мы могли подключать в HBA (hot bus adapters) несколько таких коробок.

Скорость ВВ конечно повышалась, но надежность падала. Отказы появляются все чаще и чаще. Поэтому придумали RAID, которые объединили в логический том и предусмотрели возможность восстановления данных (не везде)

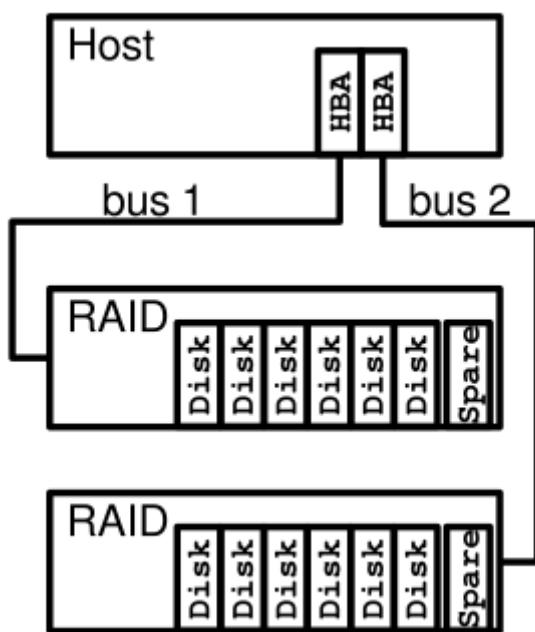
**RAID (Redundant Array of Independent Disks)** -- избыточный массив независимых дисков. Набор физических дисков, которые рассматриваются ОС как единый логический диск

RAID-схема состоит из 7 уровней, каждый из которых определяет какую-то архитектуру  
**Характеристики**

- Данные распределены по физическим дискам массива
- Избыточная емкость может использоваться для хранения контрольной информации, гарантирующая восстановление данных в случае отказа одного из дисков.
- Один блок может находиться на нескольких дисках
- Один запрос может исполняться параллельно
- Необходимо учитывать производительность адаптеров и шин подключения к адаптерам, а также шин I/O

**Из семи уровней обычно используются: 0,1,5,6**

При выходе из строя одного диска начинает работать spare диск



## 61. RAID-0, 1, 10, 0+1

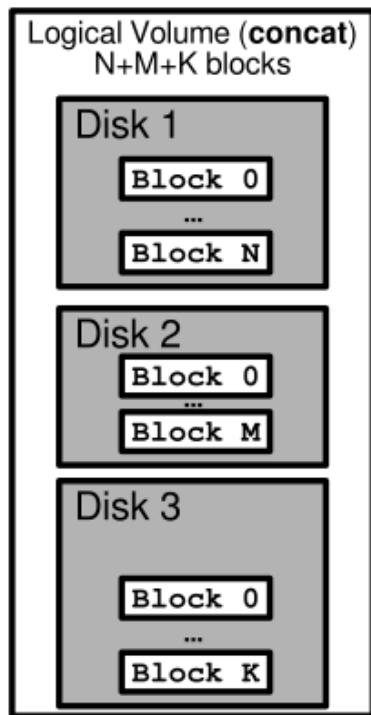
---

### RAID-0

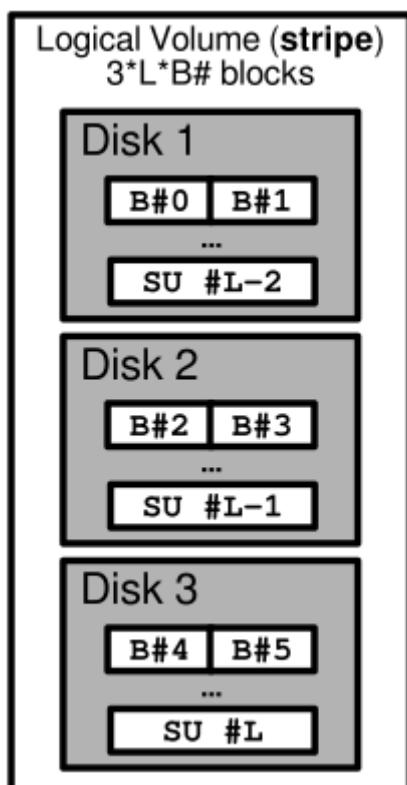
рейд-0 не использует избыточности и по сути не является настоящих рейд-уровнем, но его используют там где доминируют вопросы производительности и емкости, а снижение стоимости более важно, чем надежность

**В целом рейд-0 существует в двух вариантах**

- **Concatenation.** Берем диски, которые могут быть также разного размера. Ставим специальное ПО под названием **Volume Management** и с помощью пары команд можно объединить несколько подключенных дисков в один. Происходит конкатенация дисков и мы получаем одно общее адресное пространство из нескольких дисков



- **Striping.** Предложили объединить блоки в Stripe Units размером Stripe Size (16-128Kb)
- SU зависит от конфигурации.
- Все стрип юниты располагаются циклически то есть 1 юнит на первом диске, второй юнит на втором и тд



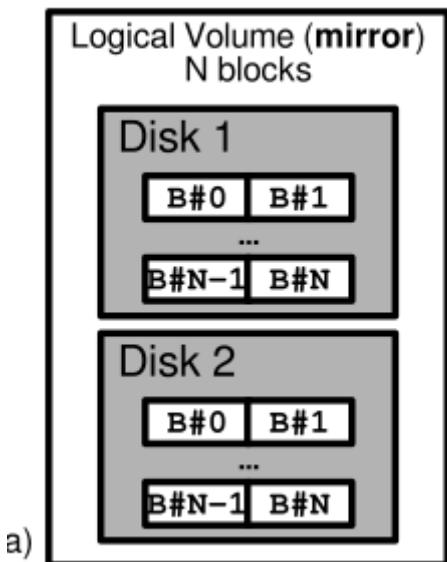
**Характеристик**

- Низкая надежность
- Высокая пропускная способность
- Высокая скорость обработки запросов чтения и записи

## RAID-1

**Зеркалирование.** Добавляем избыточность путем просто копирования данных.

Зеркальный диск должен обладать такой же конфигурацией, что и основной



a)

## Характеристики

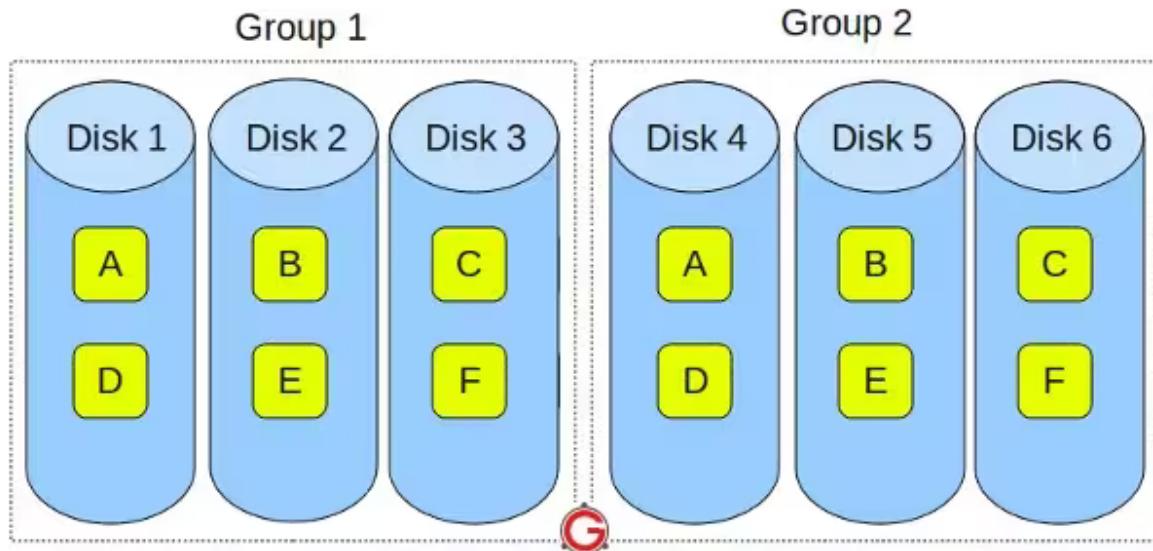
- Запрос на чтение может быть обслужен любым из двух дисков, содержащий необходимые данные. Для обслуживания выбирается тот диск, у которого минимальное время поиска и минимальная задержка вращения
- Для запроса на запись нам надо обновить обе полосы, что может быть сделано в параллельном режиме, поэтому скорость записи определяется более медленным диском.
- Простота восстановления в случае сбоя

## RAID 0 + 1

В такой конфигурации данные сначала страйпируются на блоки, а затем подключается зеркало, которое также содержит эти страйпы

(если я правильно понял, то это просто RAID 0 только с зеркалированием)

- Минимальная конфигурация требует 4 диска (тк для RAID-0 со страйпами нам нужно 2 диска, чтобы циклически так сказать размещать страйпы + 2 диска для зеркала)
- Быстрее скорость записи
- Скорость чтения очень быстрая
- Менее надежная конфигурация
- В данном случае если один диск выйдет из строя то весь массив в целом ломается

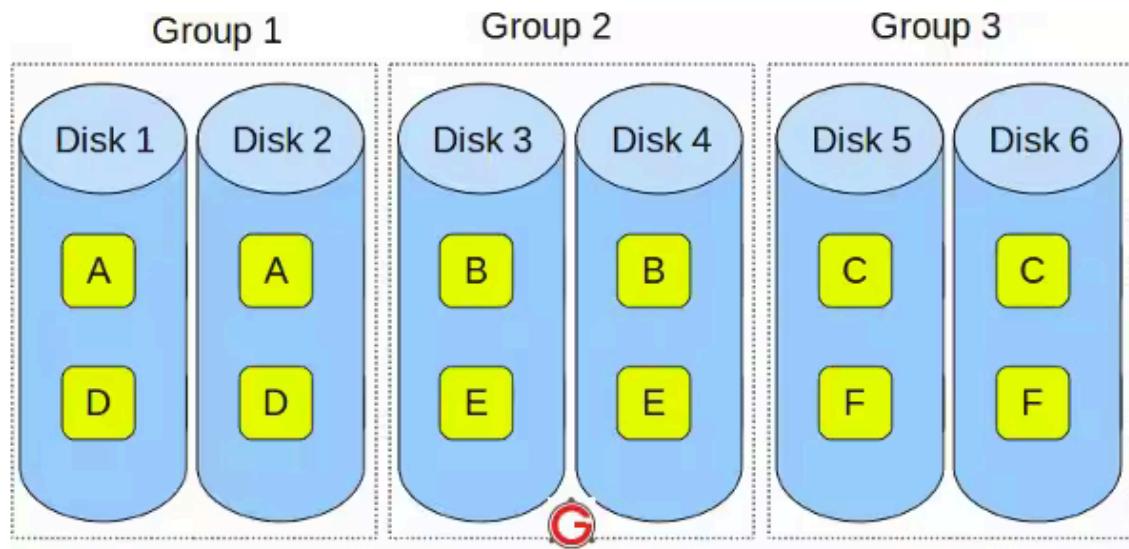


RAID 01 – Blocks Striped. ( and Blocks Mirrored)

### RAID 1 + 0

Обладает повышенной скоростью восстановления и более надежный. В данном случае мы сначала делаем зеркалирование а потом разбиваем на страйпы

- Если один диск выйдет из строя, то страйпы все равно продолжают работать
- Также нужно минимум 4 диска для формирования массива



RAID 10 – Blocks Mirrored. ( and Blocks Striped)

## 62. RAID-4, 5, 6. Аппаратные дисковые массивы

В RAID-4,5,6 используется технология независимого доступа, то есть каждый диск может работать независимо от другого так что отдельные запросы ВВ могут выполняться параллельно.

Во всех этих схемах также идет расщепление данных на полосы причем достаточно большие.

## RAID-4

По соответствующим полосам на каждом диске данных вычисляется полоса четности, хранящаяся на доп диске

При каждой записи мы должны обновить не только блоки данных, но и пересчитать биты четности. Таким образом нам нужно два чтения и две записи, чтобы сначала прочитать данные + старые биты четности и записать данные + новые биты четности.

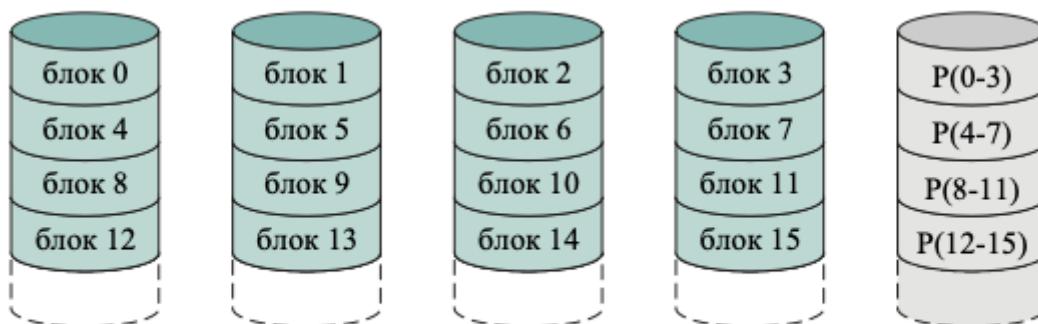
Избыточность: Disks +1

Изначально для каждого  $i$ -го бита выполняется следующее соотношение:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \quad (11.1)$$

После обновления (измененные биты отмечены штрихом) получаем:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$



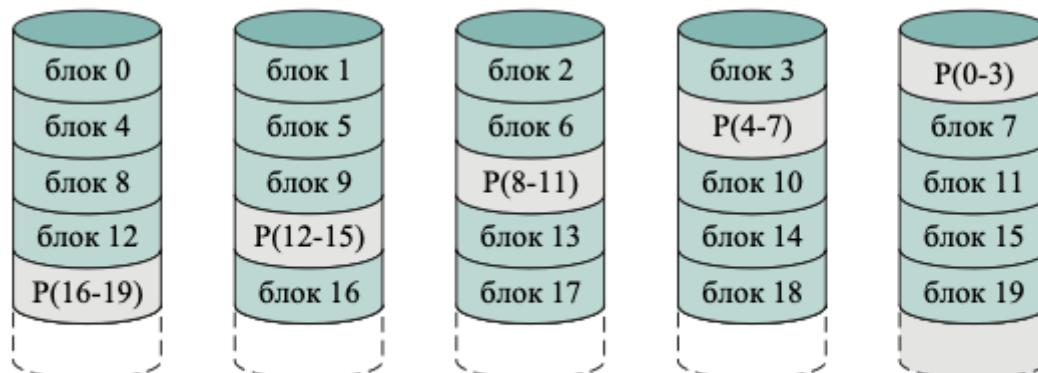
д) RAID 4 (четность с чередующимися блоками)

## RAID-5

Организация подобна RAID-4, но полосы четности распределяются по всеми дискам

Потеря одного любого диска не приводит к потери всех данных, то есть высокая надежность

Избыточность -- Disks + 1

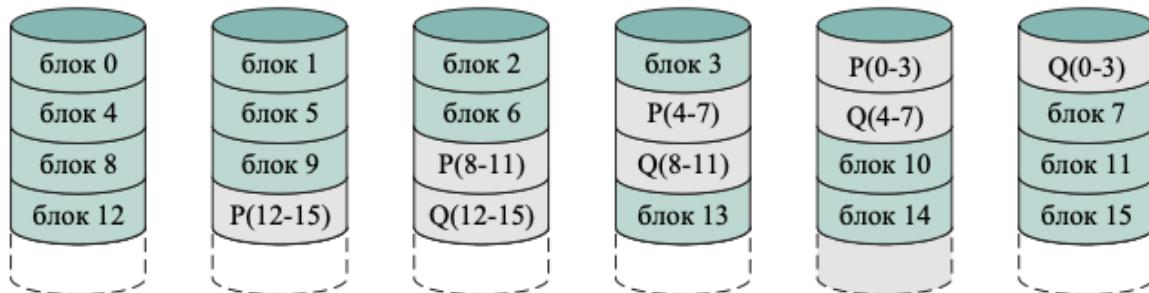


е) RAID 5 (распределенная четность с чередующимися блоками)

## RAID-6

В этой схеме выполняются два различных расчета четности, результаты которых хранятся в разных блоках на разных дисках. Поэтому массивы RAID 6 с объемом пользовательских данных, требующих N дисков, состоят из **N+2** дисков.

Преимущество RAID 6 состоит в том, что эта схема обеспечивает чрезвычайно высокую надежность хранения данных. Потери данных возможны лишь при одновременном выходе из строя трех дисков массива. С другой стороны, у RAID 6 высокие накладные расходы при операциях записи, поскольку каждая запись затрагивает два блока четности.



ж) RAID 6 (двойная распределенная четность с чередующимися блоками)

## Аппаратные дисковые массивы

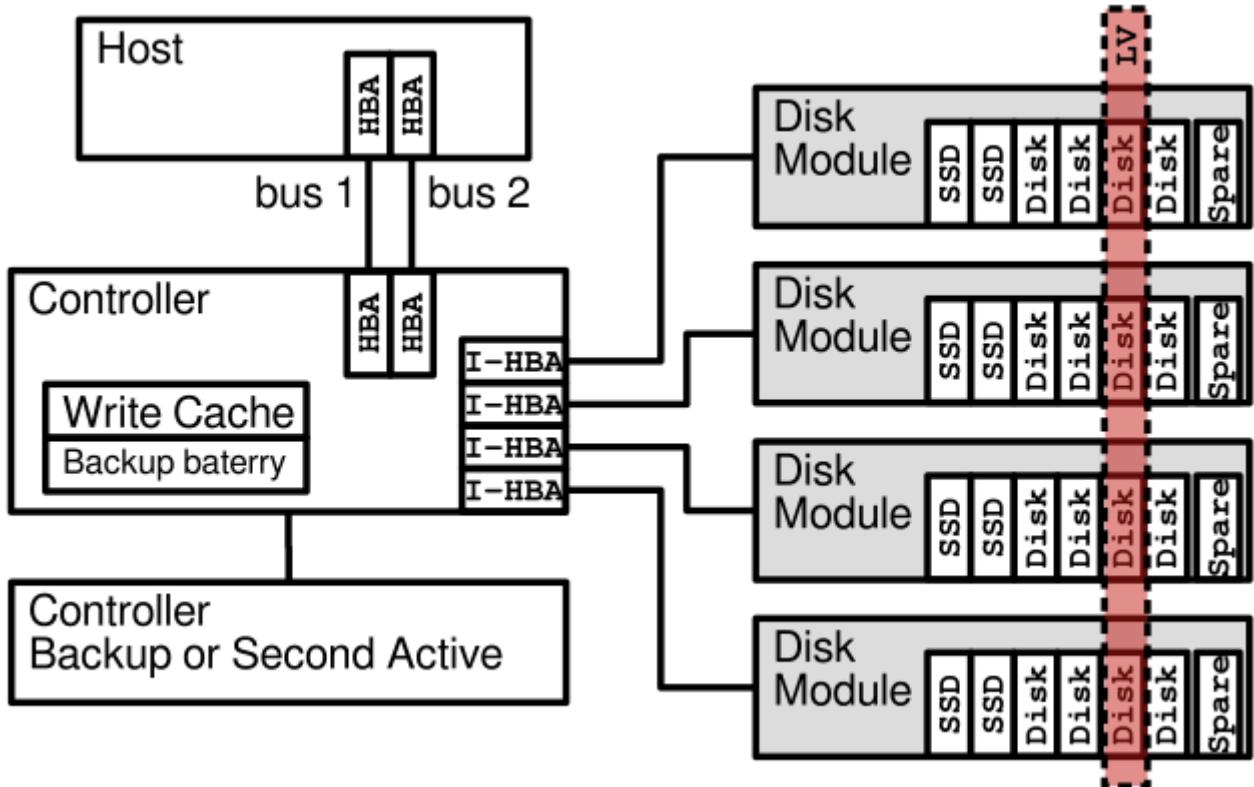
Используются в облачных хранилищах и имеют очень сложную архитектуру

### Состоит из

- Быстро считают четность
- Быстрый ВВ
- Внутри есть несколько НВА, к которой подключаются дисковые полки
- Контроллеры могут дублироваться для надежности, который также подключен к дисковым полкам
- Есть специальные буфера, которые используются как промежуточное хранилище при очень быстром ВВ и потом внутренняя часть контроллера разносит данные по томам.
- И есть доп батарея в случае отключения электричества

С помощью ПО мы видим такой контроллер как простой диск

Такие массивы отказоустойчивые, быстрые, позволяют хранить большой объем данных, но очень дорогие



На всякий путь слайд будет

## RAID 2 и RAID 3

- Диски синхронизированы между собой по позициям головок
- Может использоваться очень маленький stripe uint (байт или слово)
- На практике используются редко
- Запись на все диски одновременно

Logical Volume (Hamming ECC) RAID 2						
Disk 1	Disk 2	Disk 3	Disk 4	Disk 5	Disk 6	Disk 7
A0 B0 ...	A1 B1 ...	A2 B2 ...	A3 B3 ...	H0A H0B ...	H1A H1B ...	H2A H2B ...

Logical Volume (Parity Check) RAID 3				
Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
A0 More options ...	A1 B1 ...	A2 B2 ...	A3 B3 ...	PA PB ...

- RAID 2:
  - Необходимо большое количество дисков
  - Коды Хемминга — коррекция одиночных ошибок, обнаружение двойных.
  - Минимум 7 дисков
- RAID3:
  - Подсчет четности
  - Исправление одиночной ошибки
  - Избыточность Disk+1

**Операционные системы. Часть 3. Память и планирование**  
 All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

**tuneit** 14

### 63. Файловый ввод-вывод, основные определения. Задачи ОС по управлению файлами. Совместное использование файлов

Коллекции данных, именуемые файлами, со следующими желательными свойствами

- **Долгосрочное существование.** Хранятся и не исчезают при выходе пользователя из системы
- **Совместное использование процессами.** Файлы имеют имена и связанные с ними права доступа, которые обеспечивают управляемый совместный доступ
- **Структура.** Файл может иметь внутреннюю структуру, удобную для конкретных приложений. И сами файлы могут организовываться в иерархическую или более сложную структуру

### **Основные файловые операции**

- Создание
- Удаление
- Открытие
- Закрытие
- Чтение
- Запись

**Поле** -- основной элемент данных. Содержит единственное значение, такое как имя служащего или дата или значение полученное от датчика какого-то. Характеризуется длиной и типом данных. В зависимости от структуры данных поля могут быть фиксированной длины либо переменной. В последнем случае обычно поле состоит тогда из трех подполей: действительное значение, имя поля, длина поля.

**Запись** -- набор связанных полей, которые могут быть обработаны программой как единое целое. Записи могут быть фиксированной длины или переменной (тогда у нас переменная длина полей либо переменное кол-во полей в записи)

**Файл** -- набор однородных записей. Рассматривается как единое целое и обращение идет по имени. В некоторых сложных системах уровень доступа осуществляется на уровне записи или даже поле.

**База данных** -- файл со сложной взаимозависимая структура со ссылками на другие поля.

Операции, проводимые с файлом, могут влиять на его структуру

Файл может не иметь структуру и представлять собой поток символов к примеру простой текстовый файл.

## **Подсистема управления файлами и задачи ОС**

### **Цели системы управления файлами**

- **Возможность хранить файлы и выполнять пользователю над ними операции**
  - Создание, удаление, чтение и запись
  - Иметь управляемый доступ к файлами других пользователей
  - Управлять доступом к своим файлам со стороны других пользователей (для этого и предыдущего пункта есть базовые права rwx для трех групп + также

добавили Access Control List в Unix с командой `setfacl` которая позволяет назначит права для разных категорий пользователя не опираясь на группы)

- Перемещение данных между файлами
- Выполнять резервное копирвоание и восстановление файлов. Обычно ложится не на саму ФС, а на другие приложения
- Иметь доступ к файлам по именам
- **Гарантия корректности данных.** Уверенность, что содержание файлов не меняется со временем
- **Обеспечение приемлемой производительности**
- **Поддержка различных типов устройств хранения.** Набор устройств контролируется драйверами, которые организуют общение между устройством и компьютером
- **Минимизация или исключения потерь и повреждения данных.** Создание безопасных операций, которые бы не портили данные.
- **Обеспечение базового набора функций ввода-вывода**
- **Обеспечение совместного использования файлов.**

## Совместное использование

Включает в себя разделение прав и способы блокировки файла

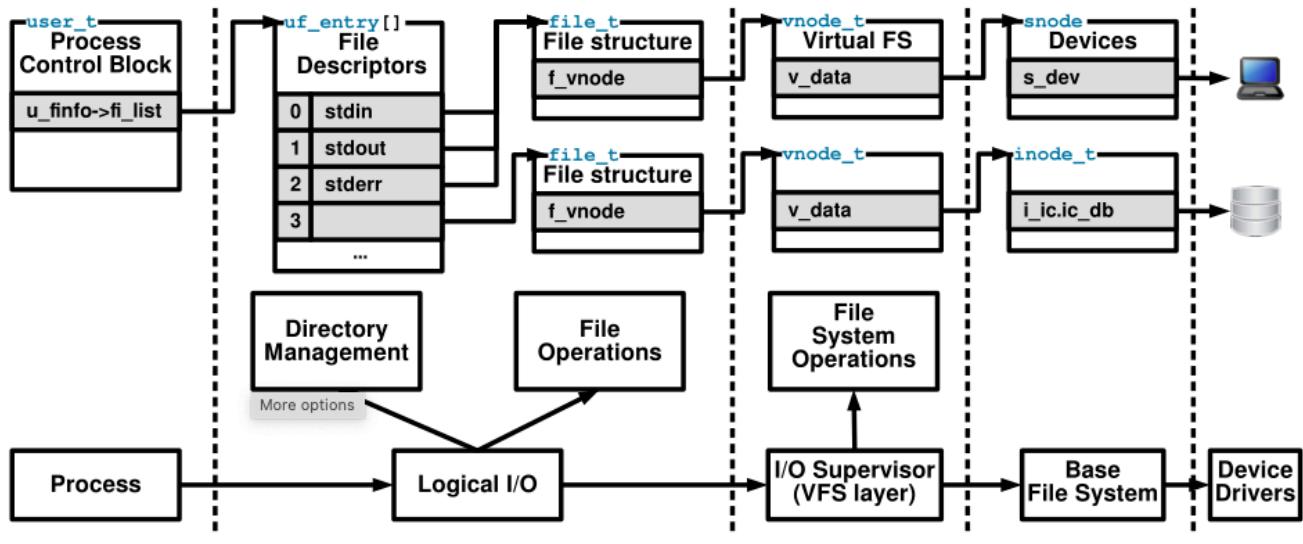
Здесь необходимо понять, что нужно делать при совместном использовании файла. Использовать блокировки, но что нам делать при блокировке? Блокировать ли файл целиком или только ту запись, которая меняется и как сообщить другим процессам или потокам про это. Кроме блокировок также есть файлы, которые используются совместно процессами (по типу dll и shared objects). Задача читателей/писателей (рассматривалась при rwlocks)

**Владелец** -- пользователь, имеющий все права на файл и распределяющий эти права

- **Отсутствие прав** -- другие юзеры даже не знаю о существовании этого файла. Нужно обеспечить запрет на чтение каталога где находится файл
- **Знание** -- юзер может обнаружить файл и установить его владельца
- **Выполнение** -- юзер может загрузить и выполнить программу, однако копирование запрещено
- **Чтение** -- чтение файла для любой цели, включая копирование и выполнение
- **Добавление** -- добавление данных в файл в его конец, но изменение или удаление содержимого файла нельзя. Используется для накопления данных из источников
- **Обновление** -- может выполнять изменение, удаление или добавление данных в файле. Начальная запись в файл, полная или чатсичная перезапись и удаление
- **Изменение защиты** -- изменение прав доступа, обычно только владелец может это делать, но в некоторых системах такое право можно расширить на других

- Удаление
- Классы пользователей
- Конкретный пользователь
- Группы пользователей (наличие Access Control List)
- Все

## 64. Управление файлами в UNIX SVR4



Картина представляет собой то, как общие задачи по управления файлами ложатся на структуры, которые есть в Юниксе

- У нас есть какой-то процесс, который работает с файлами и вызывает вызовы самого ядра
- В ядре есть логический ВВ. Обычно относится к логическому ВВ то, что близко для пользователя например, перейти в директорию, открыть файл
- В логическом ВВ есть **Directory Management**, модуль который занимается организацией файлов, хранение данных в каталогах и так далее и также есть модуль **File Operations**, который как раз занимается организацией операций по работе с файлами.
- Дальше идет уровень виртуальной ФС или уровень супервизора. Все файлы должны приходить к внутреннему интерфейсу -- VFS. Все файловые системы будут наследовать действия, которые могут выполняться вне зависимости от вида ФС. Каждая ФС представляется виртуальным узлом и в них есть ссылки на конкретные файловые системы, которые должны использовать. Этот уровень взаимодействует тесно с базовой ФС, который уже зависит от драйверов устройств и физического расположения блоков на диске.

Что касается структур

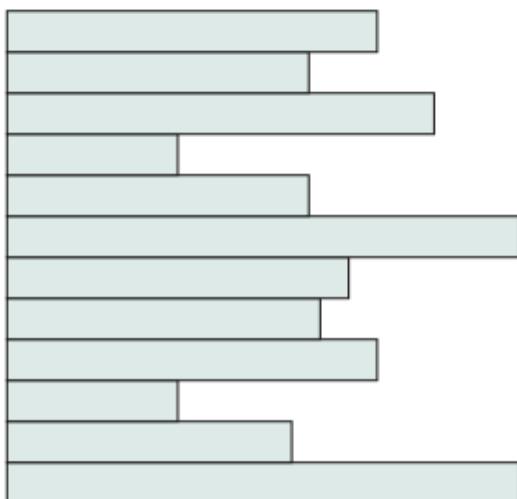
- В PCB в пользовательской части есть список файлов, которые были открыты процессов
- Сам список -- просто массив uf\_entry[] состоящий из файловых дескрипторов
- В зависимости от того на какой ФС мы откроем файл структура VFS не поменяются, а структура ближе к драйверам уже подмениться.
- Каждая запись в uf\_entry[] указывает на структуру типа file\_t которая дальше указывает на виртуальную vnode\_t внутри VFS в виде абстрактного представления файла без указания типа файла.
- Далее сами vnode\_t связаны уже с конкретными реализациями. К примеру snode связаны с потоками stdin, stdout, stderr которые связаны с конкретным терминалом

## 65. Каталоги файлов. Элементы каталога, операции ОС

---

### Сначала надо узнать о внутренней организации файлов

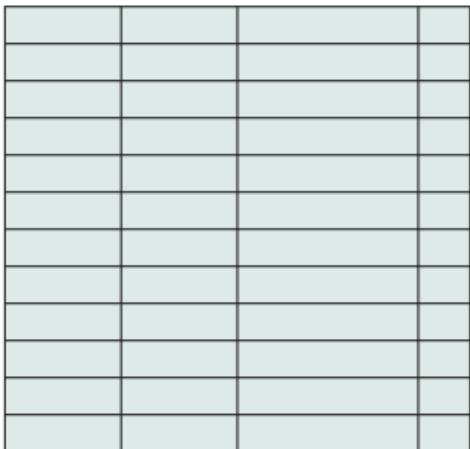
- **Смешанный файл.** Данные накапливаются в порядке своего поступления. Каждая запись состоит из одного пакета данных. Каждая запись может иметь различные поля и располагаться они также могут в разных порядках, поэтому каждая запись должна описывать сама себя. Длина записи также должно быть указано как-то. Поскольку как таковой структуры тут нет, то поиск записей делается только полным перебором



Записи переменной длины  
Переменный набор полей  
Хронологический порядок

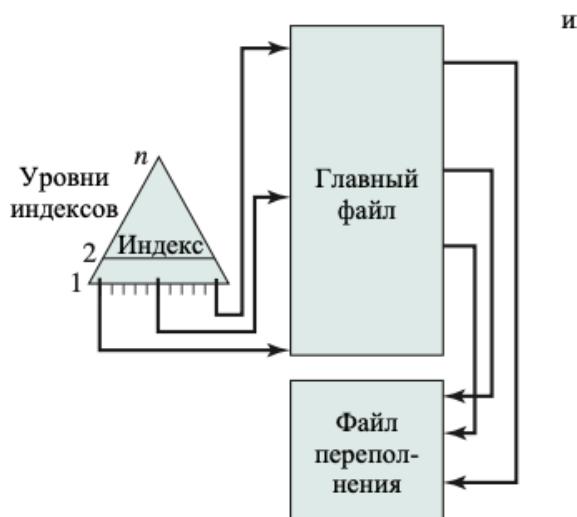
- **Последовательный файл.** это структура, в которой записи имеют фиксированный формат, одинаковую длину и поля фиксированной длины, организованные в определённом порядке. Одно из полей, обычно первое, называется **ключевым** и уникально идентифицирует запись. Записи упорядочены по ключу: в алфавитном или числовом порядке. Неэффективен для интерактивных приложений, так как поиск записи требует последовательного перебора. Внесение новых записей может потребовать создания и слияния с

журнальным файлом.



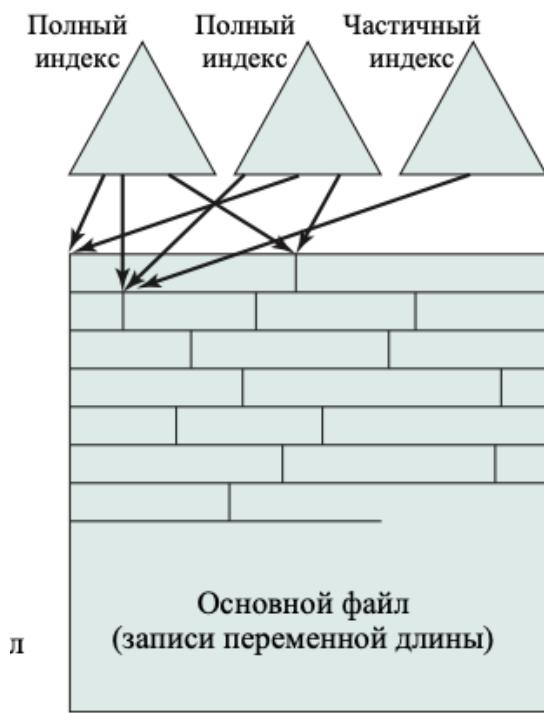
Записи постоянной длины  
Фиксированный набор нолей в фиксированном  
порядке  
Последовательный порядок, основанный на  
ключевом поле

- **Индексно-последовательный файл.** сохраняет последовательность записей по ключевому полю, добавляя **индекс** для быстрого поиска и **файл переполнения** для вставки новых записей. **Индекс** содержит ключи и указатели на записи в основном файле. Поиск начинается в индексе, а затем продолжается в основном файле, начиная с указанной позиции. Это значительно сокращает время поиска. **Файл переполнения** используется для добавления новых записей. Указатели связывают записи основного файла с новыми записями. Периодически выполняется **слияние** основного файла и файла переполнения для оптимизации структуры.



в) Индексно-последовательный файл

- **Индексированный файл.** Индексно-последовательный файл ограничен работой с ключевым полем, что делает поиск по другим характеристикам невозможным. Для гибкости используют **обобщенные индексированные файлы** с несколькими индексами для разных полей. **Полный индекс** содержит элементы для всех записей. **Частичный индекс** охватывает только записи с интересующим полем. Размещение записей не ограничено, пока запись ссылается на индекс.



г) Индексированный файл

### **Каталог файлов**

#### **Основная информация**

- **Имя файла** -- должно быть уникальным внутри каталога
- **Тип файла** -- текстовые, бинарный, загружаемый модуль и тд
- **Организация файла**

#### **Адресная информация**

- **Том(носитель)** -- устройство, на котором хранится файл
- **Начальный адрес** -- номер первого блока файла
- **Занимаемый и зарезервированный размеры** -- текущий и максимальный размеры

#### **Информация об управлении доступом**

- **Владелец** -- кто создал или кому передали доступ владения файлом
- **Информация о доступе** -- может включать имя и пароль каждого авторизованного пользователя
- **Допустимые действия** -- управление чтением, записью, пересылкой по сети

#### **Информация об использовании**

- **Дата создания**
- **Создаеть**
- **Дата последенго чтения**
- **Последний читатель**
- **Дата последнегого изменения**

- **Последний редактор файла**
- **Дата последнего резервного сохранения**
- **Текущее использование** -- текущие действия с файлом, о процессе или процессах, которые открыли файл. Заблокирован ли файл кем-то. Обновлен ли файл в основной памяти, но еще не на диске

В Юниксе каталог хранит обычно минимум информации и вся основная информация харниться уже в самой записи файла.

### **Каталог и операции ОС**

В простейшем виде структура каталога это простой список записей, по одной для каждого файла. Эту структур можно представить в виде простого последовательного файла, в котором ключевым полем служит имя файла. Сейчас такая технология не подходит

### **Основные операции**

- **Поиск.** При обращении пользователя или приложения к файлу требуется выполнение поиска записи об этом файле в каталоге
- **Создание файла.** При создании файла надо добавить элемент в каталог
- **Удаление файла.** Удаление элемента из каталога
- **Список файлов в каталоге.** Запрос на просмотр всего содержимого каталога. Обычно возвращается список всех файлов, принадлежащих пользователю, а также некоторые атрибуты файла (тип, информация о доступе, информация об использовании и тд)
- **Обновление каталога.** Изменение хотя бы одного атрибута файла требует внесений изменений в соответствующий элемент каталога.

Для обеспечения поддержки всех этих операций простой список не подходит. Поэтому придумали деревовидную структуру, где есть корневой каталог, внутри которого размещены каталоги пользователей. Каждый из каталогов пользователей может хранить внутри еще набор подкаталогов. Таким образом, пользователи могут создавать файлы с одинаковыми именами и создавать нужную им организацию файлов.

Каталог может быть полностью или частично загружен в основную память.

## **66. Размещение записей и файлов в блоках данных.**

### **Сложность и типы организации размещения.**



# Записи и блоки

Блок 1			
Запись 1	Запись 2	Запись 3	free
Блок 2			

Запись 4	Запись 5	Запись 6	free
----------	----------	----------	------

а) Фиксированное группирование

Блок 1			
Запись 1	Запись 2	Зап.3	ptr
Блок 2			

Запись 3	Запись 4	Запись 5	ptr
----------	----------	----------	-----

б) Группирование переменной длины со сцеплением  
Блок 1

Запись 1	Запись 2	Запись 3	free
Блок 2			

Запись 4	Запись 5	Запись 6	free
----------	----------	----------	------

в) Группирование переменной длины без сцепления

- Запись — логическая единица доступа к структурированному файлу
  - ОС осуществляет ввод-вывод блоками
- Какой должен быть размер блок относительно размера записи?
  - Блок больше → Больше записей передано за одну операцию ввода-вывода
  - При случайному доступе — передаем записи, которые не используются
  - Большим блокам нужны большие буферы ОС

Для начала нам нужно определиться, как сгруппировать записи внутри файла так, чтобы ОС работала эффективно с блоками.

Есть три способа организации записи внутри файла для того, чтобы ОС было удобно работать с файлами: (**ОС осуществляет ввод-вывод блоками!**)

1. Фиксированное группирование -- такое группирование, при котором берут записи одинаковой длины и заполняют ими блок до какого-то предела, оставляя неиспользованным небольшое количество свободного места. Соответственно, при этом хорошо то, что при таком подходе гарантируется, что любое чтение записи из процесса всегда прочитает одни и те же данные (т. е. при этом не нужно читать несколько блоков, чтобы прочитать одну единственную запись).
2. Группирование переменной длины со сцеплением -- группирование записей, которые могут быть разной длины, при этом запись может полностью не вместиться в блок и придется перенести часть информации в другой блок. В конце такого блока есть указатель на начало записи следующего блока. Но есть минус у такого подхода в том, что при случайному доступе (в случае если часть записи будет храниться на другом блоке) нам придется прочитать еще и другой блок, то есть производительность будет падать. При последовательном же доступе никаких "накладок" у нас не будет.
3. Группирование переменной длины без сцепления -- это группирование записей, разной длины, но при этом записи могут быть также размещены, как и в 1 методе, но необходимо сделать записи такими, чтобы они все помещались внутрь одного блока (не было разрывов кусков записей на два блока). Единственный существующий здесь нюанс - это тот пустой кусочек, который может остаться в конце блока. Ведь он может быть достаточно большим, так как на границу раздела может попасть очень большая запись. При этом появляется внутренняя фрагментация данных.

Зададимся вопросом: какой размер блока выбрать относительно размера записи?

- Если блок будет большим, то логично, что больше записей можно будет передать за 1 операцию ввода-вывода. Это значительно облегчает жизнь ОС, так как количество операций ввода/вывода уменьшается.
- Но есть у больших блоков нюанс -- случайный доступ, так как нам придется передавать записи, которые не используются, а если блок будет большим, то и кол-во ненужных записей увеличивается.
- Для больших страниц нужен большой буфер  
Мы как программисты можем задавать размер записей, но ОС работает на уровне блоков. Так что эта задача программиста сделать так, чтобы ожидания ОС совпали с нашими ожиданиями касаемо эффективности работы ВВ  
Обычно размер блока в ОС равен размеру страницы или же кратен размеру страниц (4 страницы, 5, 6 и т.д.)



- Как на физической структуре диска, состоящего из блоков разместить файлы?
  - Предварительное выделение пространства для всего файла?
  - Размер порции файла?
  - Фиксированный или переменный размер порции?
- Решения влияют на характеристики:
  - Внутреннюю и внешнюю фрагментацию
  - Частота выделения (allocation) блоков для всей файловой системы
  - Время выделения блоков
  - Размер служебной информации о размещении файлов

Представим себе диск. Диск реализован в виде блоков, которые расположены на дорожке. У нас поставлена задача -- организовать файловую систему.

Можно задаться вопросом. Как на физической структуре диска, состоящего из блоков разместить файлы?

- Можно сразу предварительно выделить пространство для всего файла. Это делается достаточно быстро путем одной записи (записи стоит рассматривать как логический элемент на уровне пользователя). У этого подхода есть минус -- файлу будет тяжело расти за эти пределы.
- Далее мы задаемся вопросом, а какие порции будут у файла, то есть какими порциями мы будем читать и записывать данные.
- Отвечая на предыдущий вопрос есть два ответа. Либо порции будут фиксированного размера, либо переменного.

Далее следует иметь в виду, что стратегия размещения файлов на физической структуре диска влияет на массу характеристик.

- Внутреннюю и внешнюю фрагментацию
- Частоту выделения (allocation) блоков для всей файловой системы
- Время выделения блоков
- Размер служебной информации, который может хранить эта файловая система
- На общий объем информации, который может хранить эта файловая система и т.д.

## 67. Непрерывное размещение файлов (на примере ОС RT-11) .



- Особенности

- Предварительное выделение пространства для всего файла.
- Один уровень каталога
- Размер блока 512 байт

- Характеристики:

- Внутренняя фрагментация мала
- Внешняя может быть большой — необходимо обязательное сжатие ФС
- Выделение блоков для файла производится однократно
- Невысокое время выделения блоков — если ФС пуста, то минимальное, если заполнена, то возможны ошибки размещения
- Минимальный размер служебной информации о размещении файлов

Рассмотрим вариант решения проблемы создания файловой системы, называемой непрерывное последовательное размещение файлов. Подобное решение реализовано в ОС RT-11.

Особенности этого решения:

- предварительное выделение пространства для всего файла, при этом, когда происходит запись файла, указывается сколько пространства он будет занимать и одновременно происходит выделение для него места на диске;
- в рассматриваемой системе существует один уровень каталога; причем подкаталоги создавать нельзя;
- размер используемого блока - 512 байт.

В начале диска используется бутовая запись (MBR -- Master Boot Record ). Содержит сведения о структуре жесткого диска и код для запуска операционной системы.

Далее последовательно расположены файлы, если размер файла до конца блока не доходил, то там остается свободное место.

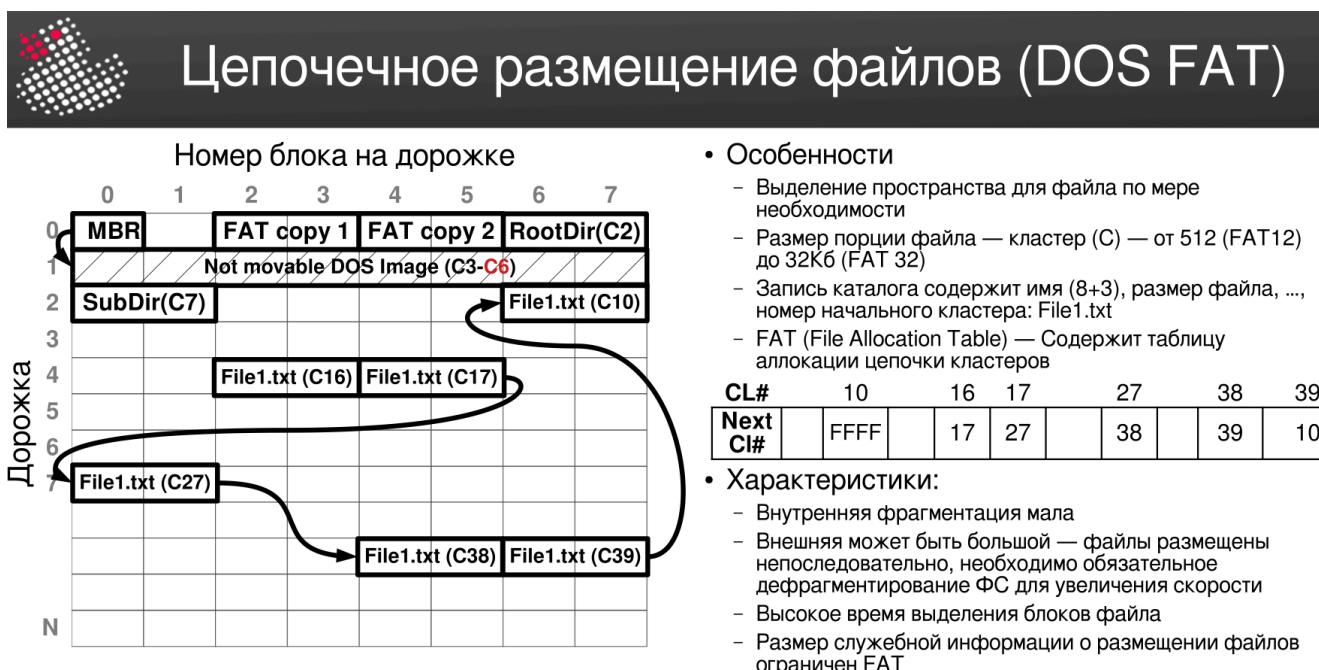
Видно, что в данном случае внутренняя фрагментация не очень большая, но проблема будет во внешней фрагментации. Когда пользователь начнет создавать и удалять файлы, то на диске будут оставаться "зазоры", получившиеся на местах удаленных файлов.

Таким образом появляется необходимость сжатия файловой системы для того, чтобы исключить свободное пространство. Для такого сжатия необходимо вначале запустить служебную утилиту для «сжатия» диска. Эта утилита переместит все файлы «в начало диска». И после этого откроется пользовательское последовательное пространство, в которое опять можно осуществлять запись.

Стоит также отметить еще ряд важных характеристик данной реализации файловой системы.

- Может быть разное время выделение данных блоков -- т. е. когда файловая система пуста, то блоки файлу будут выделены быстро. А вслучае, если система уже заполнена некоторым количеством файлов, то там нужно будет поискать место, необходимое для файла.
- Служебный объем информации, хранящийся на диске -- минимальный. То есть служебной информацией у нас является MBR и каталог, а все остальное место диска используется для хранения файлов.
- Также из преимуществ можно выделить то, что файлы располагаются последовательно. То есть если необходимо исполнить программу, то система просто берет и быстро читает программу целиком в файл. В этом случае скорость работы с файлами (чтение и запись файла) при такой реализации файловой системы является максимальной из всех возможных.

## 68. Цепочечное размещение файлов (на примере DOS FAT) .



Рассмотрим еще один метод реализации файловых систем, называемый цепочечный метод.

Исторически такая реализация появилась в файловой системе DOS. Эта файловая система до сих пор используется на флэшках.

Называется она FAT, хотя FAT, по сути дела, является лишь частью файловой системы.  
FAT - File Allocation Table.

Особенностями этой файловой системы заключались в следующем:

- файл разбивался на кластеры одинакового размера. (т. е. в этой реализации не было необходимости выделять пространство диска для всего файла), его можно было выделить из свободных кластеров по мере надобности программы.
- Размер этого кластера был от 512 байт в первых версиях (FAT 12), до 32 Кб в более поздней версии FAT 32.
- Запись каталога содержит имя (8 + 3). 8 символов имени плюс 3 символа расширения
- File Allocation Table содержит цепочки кластеров.

CL#	10	16	17	27	38	39
Next Cl#		FFFF	17	27	38	39
			10			

Рассмотрим приведенную на слайде таблицу. Представим, что в таблице находятся номера кластеров от 2го максимального номера кластера. А содержимое ячеек этой таблицы в битах указывает на следующий кластер для данного кластера. Например, необходимо разместить на диске файл File1. При этом известно, что начальный кластер этого файла называется «кластер 16» и находится в блоке 2 дорожки 4. По приведенной FAT-таблице определяем, что в 16 кластере имеется указание на следующий кластер 17. В 17 ячейке существует ссылка на 27 кластер; следующая ссылка будет на 38 кластер, потом на 39. В 39 кластере находится переключение на начало диска в виде кластера 10. Далее идет признак того, что цепочка кластеров закончилась.

Кроме пользовательских данных, диск в этой реализации файловой системы содержит MBR и две копии FAT, необходимые для того, чтобы не потерять информации о размещении файлов.

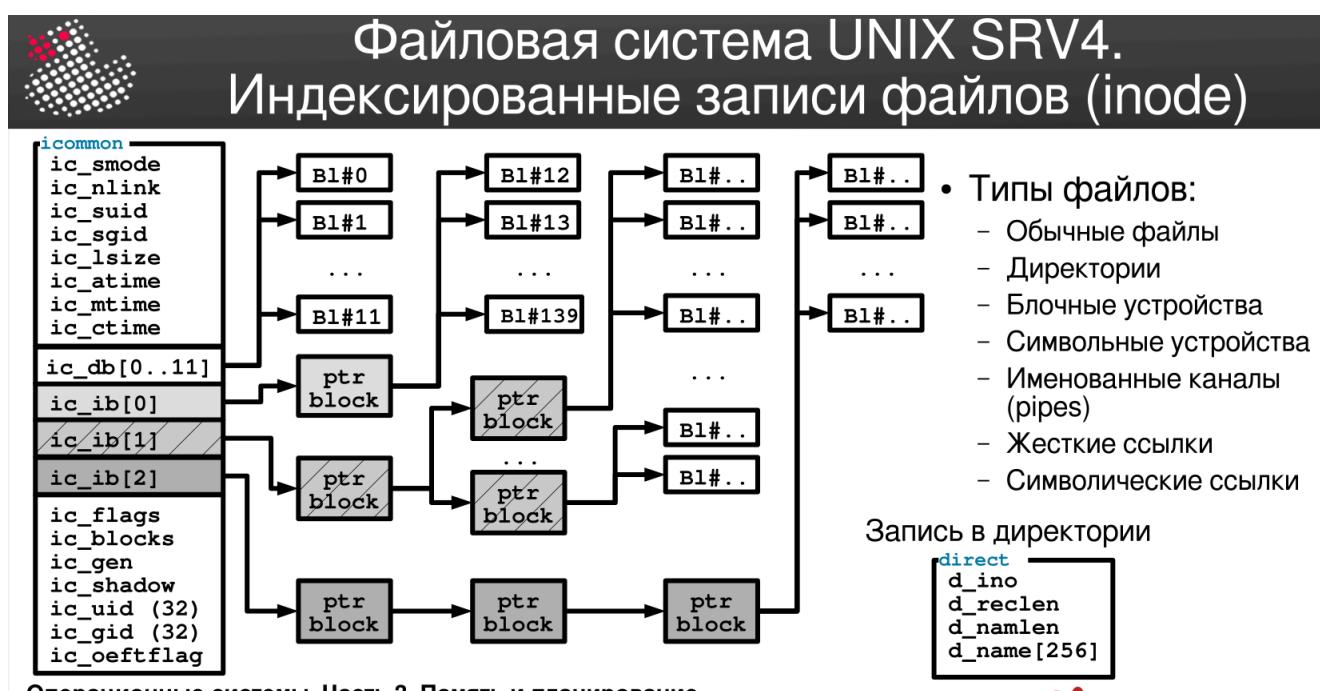
Помимо этого, в рассматриваемой реализации файловой системы есть корневая директория RootDir(C2). В корневой директории содержатся ссылки на другие директории и файлы.

Характеристики данной реализации:

- Внутренняя фрагментация мала. Так как файл состоит из цепочек, то мы можем сканировать диск и найти свободные блоки на нем и так далее пока полностью не разместим файл на диске.
- Внешняя фрагментация может быть большой. Так как файлы размещены непоследовательно, то необходимо обязательное дефрагментирование ФС. Дефragmentация размещает файлы последовательно и сжимает, чтобы не было лишних пространств между файлами.

- Высокое время выделения блока файлов (возможно из-за сканирования в поиске нужных мест для размещения файла)
- Размер служебной информации о размещении файлов ограничен FAT. То есть сколько у нас может быть записей внутри FAT, столько блоков можно будет разместить на диске (кластеров), соответственно такой будет объем нашей ФС. Поэтому разработчики DOS, Windows и подобных операционных систем, использующих FAT, начали эту файловую систему развивать. У простейшей версии FAT надо было увеличить размер записи. Вначале сделали FAT 16 (то есть  $2^{16} - 1$  записей), потом еще увеличили до FAT 32. Потом сделали достаточно большие блоки внутри FAT 32. При этом получилось, что, если вам нужно записать файл 10Кб, то он все равно будет занимать 32Кб на диске, что в свою очередь говорит о внутренней фрагментации, поэтому FAT не рекомендован для хранения большого количества мелких файлов.

## 69. Индексированное размещение (на примере файловой системы UNIX UFS) .



На слайде вы можете увидеть индексированные записи внутри файловой системы (UNIX SRV4).

В записи о файле есть:

- `ic_smode` -- права или режимы доступа к файлу
- `ic_nlink` -- количество жестких ссылок на данный файл
- `ic_suid` -- пользователь
- `ic_sgid` -- группа пользователя
- `ic_size` -- размер файла
- `ic_atime` -- access time, время доступа к файлу
- `ic_mtime` -- время модификации файла

- `ic_ctime` -- creation time, время создания файла
- 

- `ic_flags` -- флаги
- `ic_blocks` -- количество блоков на диске, которые занимает файл
- `ic_gen`
- `ic_shadow` -- атрибут, который позволяет накладывать тень и под ней скрываться
- `ic_uid (32)`
- `ic_gid (32)`
- `ic_oeflflag`

Но самое интересное посмотреть на организацию хранения файлов:

Для этого предназначены два массива.

Первый массив называется `ic_db[0..11]` (`db` - direct block) , т.е. когда мы прочитали запись о файле - мы сразу имеем 12 прямых номеров блоков, которые представляют собой начало нашего файла. Т. е. прочитали запись о файле и сразу знаем где начать искать содержимое нашего файла.

Следующий массив - это массив `indirect blocks`, который состоит из трех записей.

`Ic_ib[0]` - косвенные блоки с одним указателем (блоки с одинарной косвенностью), т.е. они указывают на блок указателей в диске (`ptr block`), а блок в диске содержит в себе указатели `b#12 b#13 ...b#139` - указатели на дальнейшие блоки. Т. к. блок на диске может быть минимум 512 байт, то в этот массив могут войти 128 указателей (128 получается делением 512 на 4 байта; 4 байта - это 32-разрядный размер слова, в котором хранится указатель на блок). (В `ic_db[0..11]` хранятся тоже 32-разрядные номера блоков). `Ic_ib[0]` - это блок с одинарной косвенностью.

Если объема памяти не хватает, то используются блоки с двойной косвенностью.

`Ic_ib[1]` - (блоки с двойной косвенностью) здесь тоже есть указание на блок указателей (`ptr block`), который в свою очередь указывает на блок указателей (`ptr block`), каждый из которых указывает на блок диска `b1#...`

Т.е. посмотрели в одном наборе указателей (`ptr block`), нашли следующий блок, и т.к. известно, что это двойная косвенность, то мы понимаем, что в этом блоке не данные, а указатели (`ptr block`), и уже по ним определяем следующий номер необходимого диска. Случайный доступ достаточно прост. Нужно лишь очень быстро посчитать по смещению относительно начала файла номер блока, который нам необходим.

`Ic_ib[2]` - блок указателей с тройной косвенностью ; в нем надо будет последовательно пройти через три блока указателей (`ptr block`) и только потом получить необходимые блоки дисков.

Все перечисленное называется индексированные записи файлов. То есть в системе всегда существует запись, которая индексом указывает на блок, который файл может использовать.

Всего в UNIX есть несколько типов файлов:

- Обычные файлы;
- Директории;
- Блочные устройства;
- Символьные устройства
- Именованные каналы (pipes)
- Жесткие ссылки (это тоже обычный файл, жесткие ссылки и обычные файлы -- это один тип файлов, просто в описании увеличивается параметр nlink, когда создается жесткая ссылка)
- Символические ссылки

В UNIX существует запись в директории direct.

- С директорией, которая хранится на диске все просто: в ней существует указатель на inode: d\_ino (т.е. на номер записи в этой файловой системе).
- Потом идет размер записи: d\_reclen.
- Потом - длина имени: d\_namelen.
- Потом - в байтах идет само имя d\_name(256), которое заканчивается стоп-символом. Соответственно, этой длиной вы указываете сколько символов будет в имени.

После того, как структура закончилась, в директории идет следующая аналогичная запись.

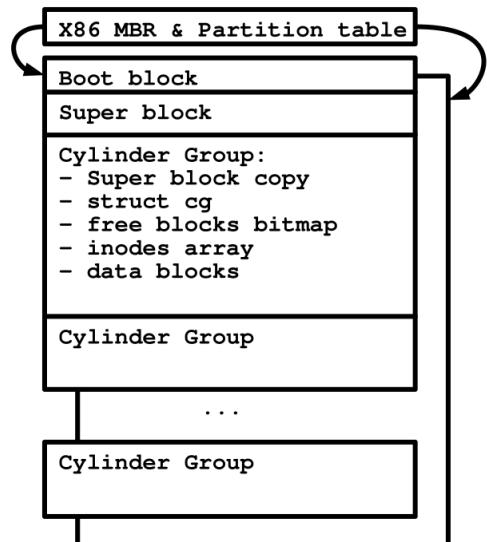
Директории - это файлы, в которых содержатся ссылки на inodes.

Inodes находятся на диске в специальных местах (icommon). И они в свою очередь обеспечивают адресацию блоков внутри нашего файла и хранения некоей служебной информации.



- Файловая система содержит:

- Загрузчик ОС
- Суперблок — геометрия и служебные параметры файловой системы
- Цилиндровые группы — информация о свободных блоках, массив записей о файлах (inode), блоки данных — равномерно распределены по всему дисковому пространству



Возможно слайд данный и не нужен для этого вопроса, но пускай будет.