

## БИЛЕТ 1 Поколения компьютеров

**Первое поколение компьютеров:** середина 40-х — начало 50-х годов XX века. Компьютеры этого поколения строились на электронно-вакуумных лампах (**сильно грелись**). В 1946 г. в Пенсильванском университете США была разработана вычислительная машина ENIAC, которая считается одной из первых электронных вычислительных машин (ЭВМ). Для баллистики и энергетики. Это разработка Пенсильванского университета

Проблемы:

- Очень часто выходили из строя вакуумные лампы

Компьютер состоял из процессора, оперативного запоминающего устройства и достаточно примитивных внешних устройств: устройства вывода (вывод цифровой информации на бумажную ленту), внешних запоминающих устройств (ВЗУ) — аппаратных средств хранения готовых к исполнению программы и данных (магнитные ленты), и устройства ввода, позволявшего вводить в оперативную память компьютера предварительно подготовленные на специальных носителях (перфокартах, перфоленте и пр.) программы и данные.

- однопользовательский, персональный режим.
- Программирование в машинных кодах
- Появился класс программ, обеспечивающих определенные сервисные функции программирования — это ассемблеры (эти программы уже были в компьютере ROM)

**Компьютеры второго поколения:** конец 50-х годов — вторая половина 60-х годов XX века. Использование полупроводниковых приборов — диодов и транзисторов. (БЭСМ 6)

- пакетная обработка заданий (в компьютере работала специальная управляющая программа, в функции которой входила последовательная загрузка в оперативную память и запуск на выполнение программ из заранее подготовленного пакета программ)
- компьютеры с **аппаратной поддержкой режима мультипрограммирования**
- появились языки управления заданиями.
- Появились виртуальные ресурсы ( - объекты, который видится как устройство которое реализованно как программа ПРИМЕР: файловая система (ее нет аппаратно))
- Появились первые **прообразы современных файловых систем** — систем, позволяющих систематизировать и упростить способы хранения и доступа пользователей к данным, размещенным на внешних запоминающих устройствах.
- Появились первые операционные системы

**Компьютеры третьего поколения:** конец 60-х — начало 70-х годов XX века. Использование в качестве элементной базы интегральных схем (они выполняли уже более сложные функции по типу сумматора).

- начало **аппаратной унификации их узлов и устройств**, позволившей стимулировать создание семейств компьютеров, аппаратная комплектация которых могла достаточно просто варьироваться владельцем компьютера (раньше для каждого компа было все свое — разъемы, платы и все мелкие детали. А теперь стало все взаимозаменямо)
- Третье поколение компьютеров строилось на **модульном принципе**
- **Появление семейств компьютеров (различная производительность + программная преемственность снизу - вверх).** Помогает предприятиям выбирать компьютер по целям и задачам. Популярное семейство - IBM
- В операционных системах появились специальные программы управления устройствами — **драйверы устройств**, которые имели стандартные интерфейсы, позволявшие при аппаратной модификации

компьютера достаточно просто обеспечивать программный доступ к новым или модифицированным устройствам.

- Унификация языков программирования (появления стандартов для языков программирования).
- Появились **виртуальные устройства**, драйверы которых предоставляли пользователю набор единых правил работы с группой внешних устройств, что позволило создавать программы, не зависящие от типов используемых внешних устройств.

Появление UNIX и Си (unix написан на си). Си по производительности сравним с Ассемблером по производительности. От ассемблера начали отказываться. Си – машинно-независимый язык

**Компьютеры четвертого поколения**, в первую очередь, ассоциируются с персональными компьютерами. 70-е по наше время

Их элементная база – большие (реализовывала устройство) и сверхбольшие интегральные схемы (процессоры Intel). Устройства – законченные функциональные узлы компьютера. Микропроцессор – реализация функционального узла компьютера. Появились новые функции в операционной системе: проблемы безопасности хранения и передачи данных. Массово формируются многопроцессорные системы. Параллельные системы становятся массовыми.

- «дружественность» пользовательских интерфейсов
- сетевые технологии
- безопасность хранения и передачи данных

## Билет №2 Структура Вычислительной системы. Ресурсы ВС – физические ресурсы, виртуальные ресурсы. Уровень операционной системы.

**Вычислительная система** (ВС) (не компьютер) – совокупность аппаратных и программных средств, функционирующих в единой системе и предназначенных для решения задач определенного класса.

Пример – банкомат, касса

Прикладные системы (верхний)
Системы программирования
Управление логическими/ виртуальными устройствами
Управление физическими устройствами (первый программный уровень)
Аппаратные средства (нижний)

### Аппаратный уровень

Вычислительной системы определяется набором аппаратных компонентов и их характеристики, используемых вышестоящими уровнями иерархии и оказывающих влияние на эти уровни.

#### Система команд + физ. ресурсы

Физические ресурсы: процессор, оперативная память, внешнее устройство

### Характеристики физ. ресурсов:

- **Правила программного использования**, которые определяют возможность корректного использования в программе. ( обращение к устройству с помощью команд) (**ключевое**)
- Производительность или емкость: тактовая частота, длина обрабатываемого машинного слова.
- Степень занятости или используемости данного физического ресурса.  
Нет единого правила формирования этих характеристик. Мы можем определить, какие компоненты соответствуют данному физическому ресурсу.

**Средства программирования, доступные на аппаратном уровне:**

- Система команд компьютера.
- Аппаратные интерфейсы программного взаимодействия с физическими ресурсами.

## Управление физическими ресурсами

Назначение – систематизация и стандартизация правил программного использования физических ресурсов, т.е совокупность драйверов. **Программный уровень**

**Драйвер физического устройства** – программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством. Но и скрывает от пользователя некоторые детали.

**Драйвер физического устройства решал задачи:**

1. Скрытие от пользователя некоторых нюансов.
2. Предоставление упрощенного интерфейса для упрощенного доступа к данному физическому ресурсу. Таким образом, на уровне управления физическими ресурсами (устройствами) вычислительной системы пользователю доступна система команд компьютера, а также интерфейсы драйверов физических устройств компьютера. (жесткий диск – считывание данных. Возникает ошибка. А в драйвере есть сценарий, который пытается решить эту проблему)

Пример – магнитная лента. Произвольное кол-во записей на ней.

## **Управление логическими/виртуальными ресурсами.** (создано для упрощения)

В основу этого уровня легло обобщение особенностей физических устройств одного вида и создание драйверов, имеющих единые интерфейсы, посредством которых осуществляется доступ к различным физическим устройствам одного типа.

**Логическое/виртуальное устройство (ресурс)** – устройство, некоторые или все эксплуатационные характеристики которого реализованы программным образом.

– унификация доступа к устройствам одного типа

**Драйвер логического/виртуального ресурса** – программа, обеспечивающая существование и использование соответствующего ресурса. Он определяет к какому физическому устройству идет запрос

Этот уровень ориентирован на пользователя.

Драйверы можно разделить на 3 группы: 1) драйверы физических устройств 2) драйверы виртуальных устройств определенного типа 3) драйверы виртуальных устройств

**Ресурсы вычислительной системы** – совокупность всех физических и виртуальных ресурсов. Одной из характеристик ресурсов является их конечность, следовательно, возникает конкуренция за обладание ресурсом между его программными потребителями.

Средства программирования, доступные на уровнях управления ресурсами ВС:

- система команд компьютера
- программные интерфейсы драйверов устройств (как физических, так и виртуальных)

**Операционная система** – это комплекс программ, обеспечивающий управление ресурсами вычислительной системы. Пользователю же доступна система команд. Разветвленная иерархия виртуальных и физических устройств.

## Билет №3 Структура вычислительной системы. Ресурсы ВС – физические, виртуальные. Уровень систем программирования.

**Система программирования** – это комплекс программ, обеспечивающий поддержание жизненного цикла программы в вычислительной системе.

Жизненный цикл программы в вычислительной системе состоит из четырех основных этапов:

- **Проектирование программного продукта.** Состоит из нескольких взаимосвязанных между собой действий: исследование, характеристика объектов вычислительной системы (какая платформа), математическая модель функционирования (какие алгоритмы), характеристика инструментальной вычислительной

системы (какими средствами мы запрограммируем, выбор библиотек), алгоритмы и инструментальные средства, априорная оценка.

- **Кодирование** (программная реализация). Построение кода на основании спецификаций при использовании языков программирования и трансляторов. Системы поддержки версий – фиксируют реализацию продукта в данный момент времени.
- **Тестирование и отладка** – это проверка программы на тестовых нагрузках. Принимается решение о формировании минимального набора тестов, более полно проверяющих программу.
- **Ввод программной системы в эксплуатацию (внедрение) и сопровождение.**

**Отладка** – процесс поиска, локализации и исправления зафиксированных при тестировании ошибок.

Последний этап предъявляет программному продукту целый ряд специфических требований.

Этапы жизненного цикла программы могут комбинироваться. Среди современных технологий разработки программного обеспечения можно выделить **каскадную модель, каскадную итерационную модель и спиральную модель**

**Система программирования** – это комплекс программ, обеспечивающий технологию автоматизации проектирования, кодирования, тестирования, отладки и сопровождения программного обеспечения. С 90-х годов 20 века по настоящее время – появляются промышленные средства автоматизации проектирования программного обеспечения, case-средства, унифицированного языка моделирования UML. Системы программирования – интегрированные системы.

**Средства программирования, доступные на уровне системы программирования** – программные средства и компоненты СП, обеспечивающие поддержание жизненного цикла программы

## Билет №4 Структура Вычислительной системы. Ресурсы ВС- физические и виртуальные. Уровень прикладных систем.

Ради него строится ВС

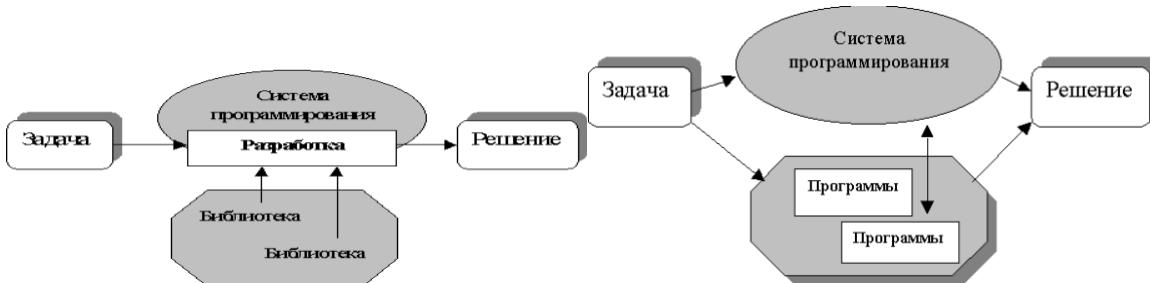
**Прикладная система** – программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области.

Прикладная система является прагматической основой всей вычислительной системы, так как, в конечном счете, именно для решения конкретных прикладных задач создавались все уровни вычислительной системы, которые мы рассмотрели к настоящему времени.

### Первый этап развития прикладных систем

Задача → Разработка, программирование → Решение

**Второй этап** Развитие систем программирования и появление средств создания и использования библиотек программ



**Третий этап** характеризуется появлением **пакетов прикладных программ**, имеющих развитые и стандартизованные интерфейсы, возможность совместного использования различных пакетов.

### Основные тенденции в развитии современных прикладных систем

**Современные прикладные системы характеризуются:**

- Стандартизация моделей автоматизируемых бизнес-процессов и построение в соответствии с данными моделями прикладных систем управления. В результате детального анализа и структуризации процессов, происходящих на различных уровнях управления предприятиями, взаимодействия предприятий друг с другом или взаимодействия предприятия с потребителями были стандартизованы разнообразные модели бизнес-процессов
  - **B2B** (business to business)
  - **B2C** (business to customer)
  - **ERP** (Enterprise Resource Planning)
  - **CRM** (Customer Relationship Management)
- **Открытость системы**: потребителю системы открыты прикладные интерфейсы, обеспечивающие основную функциональность системы, а также стандарты организации внутренних данных. **API - Application Programming Interface**
- **Использование современных технологий и моделей организации системы**: Internet/Intranet-технологии, средства и методы объектно-ориентированного программирования (ООП)

#### Категории пользователей

1. **Оператор или прикладной пользователь**, оперируя средствами пользовательского интерфейса и функциональными возможностями системы, решает конкретные прикладные задачи.
2. **Системный программист** — пользователь компонентов прикладной системы, обеспечивающий возможности интеграции данной системы в конкретной вычислительной системе, возможности настройки в соответствии с конкретными особенностями эксплуатации системы на конкретном предприятии
3. **Системный администратор** обеспечивает выполнение текущих работ по поддержке функционирования программной системы в конкретных условиях:

## Билет №5 Структура вычислительной системы. Понятие виртуальной машины.

Понятие виртуальной машины неотрывно связано с понятием виртуальных и физических ресурсов . Мы можем сделать некий срез уровня любой вычислительной системы, основываясь на иерархии и классификации по уровням. Например, мы можем рассматривать только аппаратный уровень, или только уровень операционной системы. На каждом из этих уровней мы встретимся с понятием «виртуальной машины». Дело в том, что мы никогда не можем работать просто с «компьютером». Каждый раз нам приходится использовать некую программную прослойку между нами и машиной, будь то ассемблер или Windows 95. Совокупность программных средств, обеспечивающих в любой момент времени нашу связь с компьютером, мы и назовем виртуальной машиной. Хочется подчеркнуть, что виртуальная машина всегда разная. Например, если мы работаем с DOS, то наша виртуальная машина обладает следующими характеристиками: во-первых, она имеет систему команд ДОС, то есть в то время, как физически для нашего компьютера определена система команд низкого уровня, наша виртуальная машина обладает системой команд, которые включают в себя команды «dir» или «cd». Виртуальная машина DOS способна выполнять только одну задачу в один момент времени, она не предназначена для мультипрограммирования, хотя на деле мы можем работать за многопроцессорной рабочей станцией. С другой стороны, виртуальная машина Windows имеет больший объем оперативной памяти (речь идет о подкачке), по сравнению с компьютером, на котором она установлена. То есть можно сказать, что виртуальная машина никак или практически никак не связана с физической, за исключением, конечно же, того, что виртуальная машина в любом случае вынуждена использовать физическую. Рассмотрим виртуальные машины по уровням.

**Начнем с уровня физических ресурсов.** Пусть у нас есть жесткий диск и драйвер этого диска. В этом случае драйвер представляет собой виртуальную машину, ведь если подумать, драйвер никак не связан с диском, драйвер можно скопировать на дискету и унести от диска. Но драйвер, с другой стороны, и есть для пользователя диск, поскольку именно драйвер – это то, что позволяет использовать диск. Таким образом, можно сказать, что без драйвера диск - не диск. Виртуальная машина здесь – это программа, представляющая собой лишь одну часть аппаратного обеспечения.

**Уровень логических ресурсов.** Пусть жесткий диск поделен на два логических раздела. В этом случае, интерфейс каждого из разделов – отдельная виртуальная машина. Каждый из логических дисков имеет меньше памяти, чем весь диск в целом, то есть представленные виртуальные машины обладают меньшим количеством ресурсов по сравнению с физическими характеристиками данного компьютера.

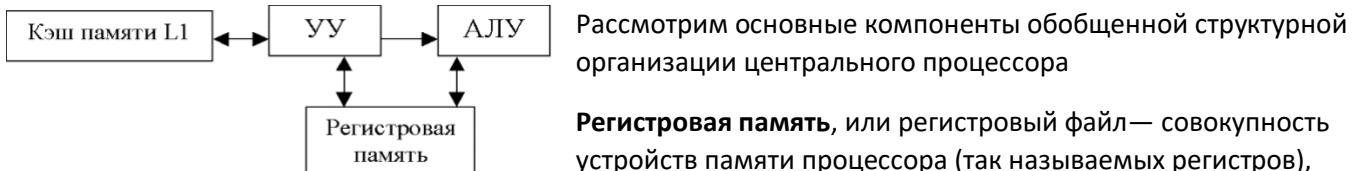
**Уровень систем программирования.** СП дают возможность создать виртуальную машину, имеющую определенный набор команд. Например, компилятор gcc позволяет эмулировать компьютер, чья система команд определена стандартом ANSI C.

**Уровень прикладных систем.** Рассмотреть базу данных и повторить то же самое

# Билет №5 Основы архитектуры компьютера. Основные компоненты и характеристики. Структура и функционирование ЦП.

НЕ все

**Процессор, или центральный процессор (ЦП),** компьютера обеспечивает выполнение машинных команд, составляющих программу, размещенную в оперативной памяти.



предназначенных для временного хранения управляющей информации, operandов и/или результатов выполняемых команд. Регистровая память обычно включает в себя регистры общего и специальные регистры.

**Регистры общего назначения (РОН)** Используются в машинных командах для организации индексирования и определения исполнительных адресов operandов, а также для хранения значений наиболее часто используемых operandов, в этом случае сокращается число реальных обращений в ОЗУ и повышается системная производительность ЭВМ.

**Специальные регистры** предназначены для координации информационного взаимодействия основных компонентов процессора. В их состав могут входить специальные регистры, обеспечивающие управление устройствами компьютера, регистры, содержимое которых используется для представления информации об актуальном состоянии выполняемой процессором программы и т.д. К наиболее распространенным специальным регистрам относятся: счетчик команд, указатель стека, слово состояния.

**Устройство управления** - координирует выполнение команд программы процессором.

**Арифметико-логическое устройство** - обеспечивает выполнение команд, предусматривающих арифметическую или логическую обработку operandов.

**Рабочий цикл процессора** – последовательность действий, происходящая в процессоре во время выполнения программы.



Решение, которое на сегодняшний день является наиболее эффективным, основывается на аппаратных средствах, позволяющих при выполнении программы автоматически минимизировать количество реальных обращений в оперативную память за operandами и командами программы за счет **кэширования памяти** — размещения части данных в более высокоскоростном запоминающем устройстве. Таким

средством является **КЭШ-память** (cache memory) — высокоскоростное устройство хранения данных, используемое для буферизации работы процессора с оперативной памятью. В общем случае, кэш представляет собою аппаратную «емкость», в которой аккумулируются наиболее часто используемые данные из оперативной памяти

# Билет №6 Основы архитектуры компьютера. Оперативное запоминающее устройство. Расслоение памяти.

**Оперативное запоминающее устройство** – это устройство для хранения данных, в котором находится исполняемая программа. ОЗУ еще называют основной памятью, или оперативной памятью. Команды программы, исполняемые компьютером, поступают в процессор исключительно из ОЗУ.

Оперативная память состоит из **ячеек памяти**. **Ячейка памяти** – это устройство, в котором размещается информация. Ячейка памяти может состоять из двух полей (Рис. 22). Первое поле – поле машинного слова, второе – поле служебной информации (или ТЕГ).

**Машинное слово** – поле программно изменяемой информации. В машинном слове могут располагаться машинные команды (или части машинных команд) или данные, с которыми может оперировать программа.

**Поле служебной информации** – ТЕГ – поле ячейки памяти, в котором схемами контроля процессора и ОЗУ автоматически размещается информация, необходимая для осуществления контроля за целостностью и корректностью использования данных, размещаемых в соответствующем машинном слове.

Использование поля служебной информации (ТЭГа) может осуществляться в следующих целях.

- разряды контроля четности машинного слова (при записи машинного слова подсчет числа единиц в коде машинного слова и дополнение до четного или нечетного в контрольном разряде), при чтении контролль соотвествия;
- разряды контроля данные-команда (обеспечение блокировки передачи управления на область данных программы или несанкционированной записи в область команд);
- машинный тип данных – осуществление контроля за соответствием машинной команды и типа ее операндов;

В ОЗУ все ячейки памяти имеют уникальные имена, имя – **адрес ячейки памяти**. Обычно адрес – это порядковый номер ячейки памяти. Доступ к содержимому машинного слова осуществляется посредством использования адреса. Обычно скорость доступа к данным ОЗУ существенно ниже скорости обработки информации в ЦП.

Производительность оперативной памяти – скорость доступа процессора к данным, размещенным в ОЗУ:

- время доступа** – время между запросом на чтение слова из оперативной памяти и получением содержимого этого слова. ( $t_{access}$ )
- длительность цикла памяти** – минимальное время между началом текущего и последующего обращения к памяти. (время доступа + время на регенерацию памяти, тк при считывании из памяти этот элемент удаляется и приходится его заново записывать) ( $t_{cycle}$ )

Необходимо, чтобы итоговая скорость выполнения команды процессором как можно меньше зависела от скорости доступа к коду команды и к используемым в ней операндам из памяти. Это составляет проблему, которая системным образом решается на уровне архитектуры ЭВМ.

**РЕШЕНИЕ -> Расслоение ОЗУ** – один из аппаратных путей решения проблемы дисбаланса в скорости доступа к данным, размещенным в ОЗУ и производительностью ЦП.



Суть расслоения ОЗУ состоит в следующем. Все ОЗУ состоит из  $k$  блоков, каждый из которых может работать независимо. Ячейки памяти распределены между блоками таким образом, что у любой ячейки ее соседи размещаются в соседних блоках.

Возможны две модели организации доступа к памяти

1. **с централизованным контроллером доступа к памяти** (Рис. 24 – один контроллер управляет всеми банками; в этом случае нет проблемы цикла памяти, т.к. соседние ячейки памяти находятся в разных банках; но нет эффекта при параллелизме). После одного обращения ждем  $t_{access}$
2. **с контроллерами для каждого из банков – запросы в ОЗУ могут происходить параллельно если идет обращение в разные банки.** То есть цепочка  $I, i+1 \dots I-K+1$  обрабатывается за  $t_{access}$

Тип данных – определенный формат данных, с конкретным набором операций, известных для этого формата. Наличие и формат тега зависит от реализации. Доступ к содержимому машинного слова может быть прямым или косвенным.

## Билет №7 Основы архитектуры компьютера. Основные компоненты и характеристики. Кэширование ОЗУ

Ключевой проблемой функционирования компьютеров является проблема несоответствия производительности центрального процессора и скорости доступа к информации, размещенной в оперативной памяти. Решение, которое на сегодняшний день является наиболее эффективным, основывается на аппаратных средствах, позволяющих при выполнении программы автоматически минимизировать количество реальных обращений в оперативную память за операндами и командами программы за счет **кэширования памяти** — размещения части данных в более высокоскоростном запоминающем устройстве. Таким средством является **КЭШ-память** — высокоскоростное устройство хранения данных, используемое для буферизации работы процессора с оперативной памятью. В общем случае, кэш представляет собою аппаратную «емкость», в которой аккумулируются наиболее часто используемые данные из оперативной памяти.

Скорость доступа к информации, размещённой в КЭШе, соизмерима со скоростью обработки информации в ЦП. Обмен данными при выполнении программы (чтение команд, чтение значений операндов, запись результатов) происходит не с ячейками оперативной памяти, а с содержимым КЭШа. При необходимости из КЭШа «выталкивается» часть данных в ОЗУ или загружаются из ОЗУ новые данные. Варьируя размеры КЭШа, можно существенно минимизировать частоту реальных обращений к оперативной памяти.

КЭШе между командами программы и обрабатываемыми данными современные компьютеры имеют два независимых КЭШа: **КЭШ данных** и **КЭШ команд**, каждый из которых работает со своим потоком информации — потоком команд и потоком операндов.

Общая схема работы КЭШа следующая .

1. Условно, вся память разделяется на блоки одинакового размера. Обмен данными между КЭШем и оперативной памятью осуществляется блоками фиксированного объёма.
2. Каждый блок имеет спецификатор доступа – тэг, в котором находится служебная информация, характеризующая данный блок. Когда процессору нужно обратиться за командой или за данными в ОП, сначала происходит обращение к КЭШу. По содержимому адресного тэга можно однозначно адресовать содержимое блока. Анализ тэгов блоков КЭШа производится аппаратно. Таким образом, после вычисления исполнительного адреса операнда или команды устройство управления может определить, находится ли соответствующая информация в одном из блоков КЭШ-памяти или нет. Факт нахождения искомых данных в КЭШе называется попаданием – в этом случае данные берутся из КЭШа, и обращение в ОП не осуществляется. Если искомых данных нет в КЭШе, то фиксируется промах
3. При возникновении промаха происходит вытеснение – обновление содержимого КЭШа. Для этого выбирается блок-претендент на вытеснение, т.е. блок, содержимое которого будет заменено. Стратегия этого выбора зависит от конкретной организации процессора.
4. Отдельно следует обратить внимание на организацию вытеснения блока в КЭШе данных, т.к. содержимое блоков КЭШа может не соответствовать содержимому памяти (это возникает при обработке команд записи данных в память). В этом случае также возможно использование нескольких стратегий вытеснения. Первая — **сквозное кэширование**: при выполнении команды записи данных обновление происходит как в КЭШе, так и в оперативной памяти. При записи все из кэша копируется в ОЗУ, то что было в кэше забываем. Другой

стратегией является **кэширование с обратной связью** суть которой заключается в использовании специального тега модификации. Если в Кэше находится блок, который был модифицирован, то этот блок сгружается в ОЗУ по адресу который находится в теге и теперь этот блок свободен для новой информации

## ВАЖНО

Кэш дает нам возможность наиболее эффективно использовать расслоения памяти (так как мы часто работаем с последовательностью адресов из ОЗУ в кэш памяти) Читаем порциями !!!

## **!ПЛЮСЫ КЭША!**

- Работа с расслоением
- Сокращения числа обращений в ОЗУ

## Билет №8 Основы архитектуры компьютера. Аппарат прерываний.

### Последовательность действий в вычислительной системе при обработке прерываний

**Аппарат прерываний ЭВМ** - возможность аппаратуры ЭВМ стандартным образом обрабатывать возникающие в вычислительной системе события. Данные события будем называть прерываниями.

**Прерывание** — событие в компьютере, при возникновении которого в процессоре происходит предопределенная последовательность действий. *Некоторые события комп обрабатывает автоматически*. Он предполагает уже наличие некоторых программ в ОЗУ -> это часть ОС

Прерывания сделаны для того, чтобы компьютер не весь полностью заканчивал свою работу аварийно (синий экран)

#### Типы прерываний:

- **Внутренние** — инициируются схемами контроля работы процессора (деление на ноль, целостность информации в озу (бит четности))
- **Внешние** — события, возникающие в компьютере в результате взаимодействия центрального процессора с внешними устройствами

Обработка прерывания предполагает две стадии: **аппаратную**, которая включает предопределённую реакцию процессора на возникновение прерывания, и **программную**, которая предполагает выполнение специальной программы обработки прерывания, являющейся частью операционной системы.



Сначала рассмотрим этап аппаратной обработки прерывания.

1. Завершается выполнение текущей команды (за исключением случаев, когда прерывание возникает по причине некорректного выполнения команды).
2. Обработка прерывания предполагает остановку выполнения текущей программы, запуск специальной программы обработки прерывания, а затем, возможно, продолжение выполнения прерванной программы (с прерванного места). Для обеспечения этой возможности необходимо зафиксировать актуальное состояние компьютера в момент прихода прерывания. Поэтому аппаратный этап обработки прерываний регламентирует перечень регистров, которые автоматически будут сохранены процессором. Поэтому включается режим блокировки прерываний. При этом режиме в системе запрещается инициализация новых прерываний: возникающие в это время прерывания могут либо игнорироваться, либо откладываться (зависит от конкретной аппаратуры компьютера и типа прерывания). Блокировка нужна для того, чтобы не было еще других прерываний. Тк буфер один, то все сохраненные данные потеряются со следующим параллельным прерыванием. Поэтому мы блокируем прерывания.
3. Аппаратное копирование содержимого сохраняемых регистров («малое упрытывание»). Включенный режим блокировки прерывания гарантирует сохранность этих данных до момента завершения предварительной обработки прерывания и выключения блокировки прерываний.
4. Переход на программный этап обработки прерываний. Аппаратно передаём управление на некоторую фиксированную точку в ОЗУ, в которой предполагается наличие программы обработки прерываний операционной системы. Для перехода на программный этап обработки прерываний необходимо решить вопрос, как аппаратура передаст операционной системе информацию о том, прерывание какого типа произошло. Существует несколько моделей аппаратного решения этого вопроса.



- Первая модель — использование специального регистра прерываний.
  - Вторая модель — использование вектора прерываний. Предполагается, что по количеству возможных прерываний в ОЗУ выделена группа машинных слов — вектор прерываний.
  - Третья модель — использование регистра слова состояния процессора. В этом случае в данном регистре резервируется часть разрядов — поле, в которое передаётся номер возникшего прерывания.
1. Анализ и предварительная обработка прерывания. Происходит идентификация типа прерывания, определяются причины.
- Если прерывание «короткое», т.е. обработка не требует дополнительных ресурсов ЦП и времени, то прерывание обрабатывается, выключается режим блокировки прерываний, восстанавливается состояние процессора, соответствующее точке прерывания исходной программы, и передается управление на прерванную точку. (прерывание по таймеру)
  - Если прерывание является «фатальным» для программы, т.е. после этого прерывания продолжить выполнение программы невозможно (например, в программе произошло деление на ноль или обращение к несуществующему в ОЗУ адресу), то выключается режим блокировки прерываний, и управление передается в ту часть ОС, которая прекратит выполнение прерванной программы.
2. «Полное упрытывание». Если прерывание не короткое и не фатальное, то для обработки такого прерывания потребуются ресурсы. Поэтому осуществляется полное сохранение контекста (т.е. всех регистров ЦП, использовавшихся прерванной программой) в специальную программную таблицу. В данную таблицу копируется содержимое регистровой или КЭШ-памяти, содержащей сохраненные значения ресурсов ЦП, а также копируются все оставшиеся регистры ЦП, используемые программно, но не сохраненные аппаратно. После данного шага программе обработки прерываний становятся доступны все ресурсы ЦП, а прерванная программа получает статус ожидания завершения обработки прерывания. В общем случае, программ, ожидающих завершения обработки прерывания, может быть произвольное количество.
3. До данного момента времени все действия происходили в режиме блокировки прерываний. После полного сохранения регистров происходит снятие режима блокировки прерываний, то есть включается стандартный режим работы процессора, при котором возможно появление прерываний.
4. Операционная система завершает обработку прерывания.

# *Билет №9 Основы архитектуры компьютера. Внешние устройства.*

## *Организация управления и потоков данных при обмене с внешними устройствами*

### Обмен данными:

- записями фиксированного размера – **блоками (флешка, жесткий диск)**
- записями произвольного размера (магнитная лента)

### Доступ к данным:

- операции чтения и записи (жесткий диск, CDRW).
- только операции чтения (CDROM, DVDRAM, ...).

**Устройства последовательного доступа** — это устройства, при доступе к содержимому произвольной записи которых «просматриваются» все записи, предшествующие искомой (Магнитная лента)

**Устройство прямого доступа** обеспечивает выполнение операций чтения/записи без считывания дополнительной (предыдущей) информации (Магнитные диски, Магнитный барабан, Магнитно - электронные ВЗУ прямого доступа)

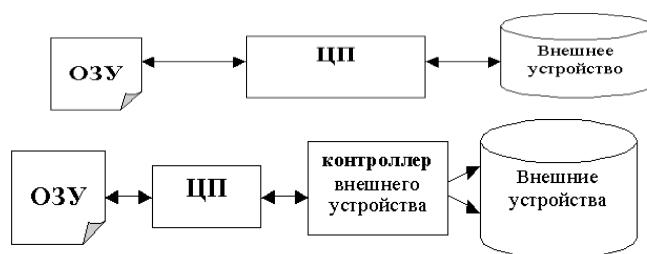
Внешние устройства во многом определяют эксплуатационные характеристики, как компьютера, так и вычислительной системы в целом.

Модель синхронизации, поддерживаемая аппаратурой компьютера при взаимодействии центрального процессора с внешними устройствами.

**Синхронная работа с ВУ.** При синхронной организации обмена в момент обращения к внешнему устройству программа будет приостановлена до момента завершения обмена. Тем самым при использовании такой модели в системе возникали задержки, которые снижали эффективность функционирования ВС.

**Асинхронная работа с ВУ.** При асинхронной организации работы внешних устройств последовательность событий, происходящих в системе, следующая: **(требование – Аппарат прерывания)**

1. Для простоты изложения будем считать, что в системе прерываний компьютера имеется специальное внутреннее прерывание «обращение к системе», которое инициируется выполнением программой специальной команды. Программа инициирует прерывание «обращение к системе» и передает заказ на выполнение обмена, параметры заказа могут быть переданы через специальные регистры, стек и т.п. В операционной системе происходит обработка прерывания, при этом конкретному драйверу устройства передается заказ на выполнение обмена.
2. После завершения обработки «обращения к системе» программа может продолжить свое выполнение, или может быть запущено выполнение другой программы.
3. По завершении выполнения обмена происходит прерывание, после обработки которого программа, выполнявшая обмен, может продолжить свое выполнение. Асинхронная схема обработки обращений к ВУ позволяет сглаживать дисбаланс в скорости выполнения машинных команд и скоростью доступа к ВУ.



**1. Непосредственное управление** внешними устройствами центральным процессором

**2. Синхронное управление** внешними устройствами с использованием контроллеров внешних устройств.

**3. Асинхронное управление**

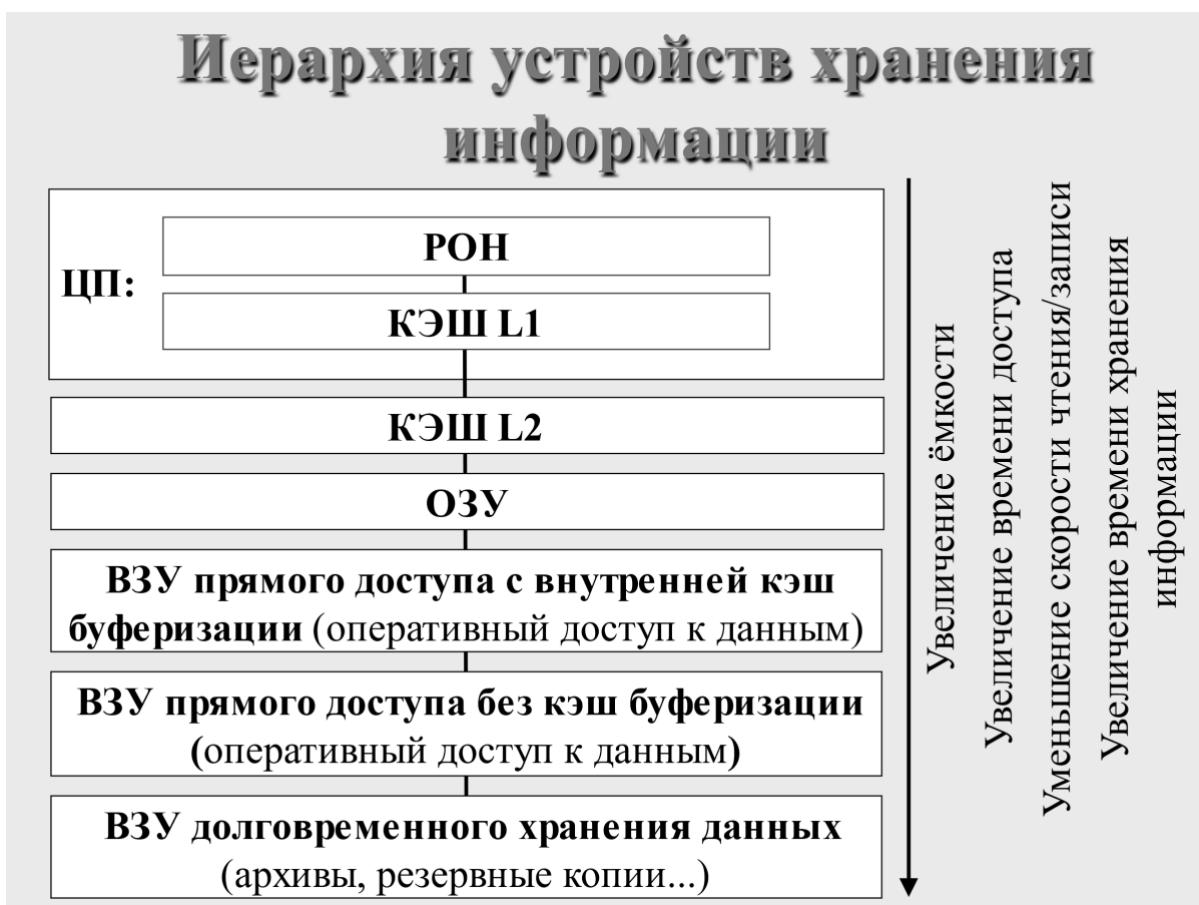
внешними устройствами с использованием контроллеров внешних устройств. **4. Использование контроллера прямого доступа** к памяти (DMA) при обмене. Управление внешними устройствами с использованием процессора или канала ввода/вывода.



## БИЛЕТ 10 Иерархия памяти

1. В центральном процессоре наиболее быстрые и наиболее дорогостоящие – регистры общего назначения и кэш-буфер. (самое ценное)
2. Оперативное запоминающее устройство: кэш-устройства (вне центрального процессора – между оперативной памятью и центральным процессором). КЭШ второго уровня – отдельное устройство, находится между ЦП и ОЗУ, то есть отдельное устройство. Работает медленнее чем КЭШ 1 но быстрее чем ОЗУ
3. Внешние устройства – для организации оперативного доступа к данным (есть кэш буфер – для сокращения обращения к ВЗУ). (дисковое устройство. Считал – записал. У таких устройств есть буфер, который выполняет роль кэша)
4. Устройства прямого доступа без кэш-буферизации.
5. Устройства для долговременного массового хранения данных.

Скорость доступа к ВЗУ очень мала



## **БИЛЕТ 11 Мультипрограммный режим**

Итак, выше мы выяснили, что, несмотря на возможность асинхронной работы с ВУ, имеют место периоды ожидания программой завершения обмена. Если система обрабатывает единственную программу, то в это время ЦП не производит никакой полезной работы, то есть простаивает (на самом деле термин простой достаточно условный, так как при этом работает операционная система).

Решением проблемы простого ЦП в этом случае является использование ВС в **мультипрограммном режиме**, в режиме при котором возможна организация переключения выполнения с одной программы на другую

1. **Аппарат защиты памяти.** Аппаратная возможность ассоциирования некоторых областей ОЗУ с одним из выполняющихся процессов/программ. Настройка аппарата защиты памяти происходит аппаратно, то есть назначение программе/процессу области памяти происходит программно (т.е., в общем случае операционная система устанавливает соответствующую информацию в специальных регистрах), а контроль за доступом – автоматически. При этом при попытке другим процессом/программой обратиться к этим областям ОЗУ происходит прерывание “Защита памяти” В спец регистрах лежат начало и конец сегмента памяти, который доступен процессу, а дальше процесс работает с этой памятью автоматически
2. Наличие специального режима **операционной системы (привилегированный режимом или режим супервизора)** ЦП. Суть заключается в следующем: все множество машинных команд разбивается на 2 группы. Первая группа – команды, которые могут исполняться всегда (пользовательские команды). Вторая группа – команды, которые могут исполняться только в том случае, если ЦП работает в режиме ОС. Если ЦП работает в режиме пользователя, то попытка выполнения специализированной команды вызовет прерывание – “Запрещенная команда”. Какова необходимость наличия такого режима выполнения команд? Простой пример – управление аппаратом защиты памяти. Для корректного функционирования этого аппарата необходимо обеспечить централизованный доступ к командам настройки аппарата защиты памяти. То есть эта возможность должна быть доступна не всем программам.
3. Необходимо наличие аппарата прерываний. Как минимум в машине должно быть прерывание по таймеру, что позволит избежать “зависания” всей системы при зацикливании одной из программ.

## **БИЛЕТ 12 Организация регистровой памяти (регистровые окна, стек)**

Одно из более или менее новых решений, предназначенное для минимизации накладных расходов, связанных с обращениями к подпрограммам, основано на использовании в современных процессорах т.н. регистровых окон (register windows). Это решение нацелено на решение проблем сохранения/восстановления регистров. В процессоре имеется некоторое количество K физических регистров общего назначения, предназначенных для использования в пользовательских программах. Эти регистры пронумерованы от 0 до K-1. В каждый момент времени программе доступно т.н. регистровое окно, состоящее из L регистров. Соответственно, все K физических регистров разделяются на регистровые окна некоторым способом. Т.е. окна организованы таким способом, что последний регистр предыдущего окна отображается на тот же физический регистр, что и нулевой регистр следующего окна.

При обращении из текущей программы в другую программу автоматически происходит смена окна, т.е. текущей программе становится доступно другое окно, состоящее тоже из L регистров. Первая группа регистров – регистры формальных параметров (включая адреса возврата), вторая – регистры локальных параметров, третья – регистры фактических параметров. Каждый из регистров окна отображается на один из регистров базового регистрационного файла. Далее, следующее регистровое окно, которое получит программа при обращении к подпрограмме, имеет пересечение с текущим окном, через начало и конец регистрационного окна. Это означает, что если в текущей программе мы загрузили значения на регистры, через которые будем передавать фактические параметры, то при обращении к подпрограмме она в своём новом окне получит фактические параметры через соответствующие формальные параметры. При этом локальные регистры текущей подпрограммы сохранять не нужно.

Имеются два управляющих регистра: указатель текущего окна (CWP) и указатель сохранённого окна (SWP). Суть заключается в том, что количество регистрационных окон ограничено. И если глубина вложенности больше, чем количество регистрационных окон, то возникает проблема. В этом случае какое-то окно необходимо сохранить в ОП или стеке, чтобы потом его использовать. При этом эффективность, естественно, начнёт падать. Считается, что наиболее оптимальный эффект оптимизации достигается при четырех окнах.

### **Стек**

Будем рассматривать системы, в которых имеется аппаратная поддержка стека. Это означает, что имеется регистр, который ссылается на вершину стека, и есть некоторый механизм, который поддерживает работу со стеком. Использование системного стека может частично решать проблему минимизации накладных расходов при смене обрабатываемой программы. В частности, этот механизм может использоваться при обработке прерывания: если в системе возникает прерывание, процессор просто сохраняет в стеке содержимое необходимых регистров («малое упаковывание»). Если же возникнет второе прерывание, то процессор поверх предыдущих данных складывает в стек новое содержимое регистров, чтобы обработать вновь пришедшее прерывание.

## **БИЛЕТ 13 Виртуальная оперативная память**

Проблемы перемещаемости программы по ОЗУ и фрагментации памяти связаны с необходимостью наличия т.н. аппарата **виртуальной памяти**, т.е. аппаратного средства процессора, которое обеспечивает преобразование (установление соответствия) логических адресов, используемых внутри программы, в те адреса физической оперативной памяти, в которой размещается программа во время выполнения.

Неформально **виртуальное адресное пространство** можно определить как то адресное пространство, которое используется внутри программ (написанных, например, на языках программирования высокого уровня В исполняемом модуле используется т.н. программная (логическая, виртуальная) адресация. Виртуальные адреса существуют «вне машины». Соответственно, стоит проблема установления соответствия между программной адресацией и физической памятью. И эта проблема решается за счет аппарата виртуальной памяти.

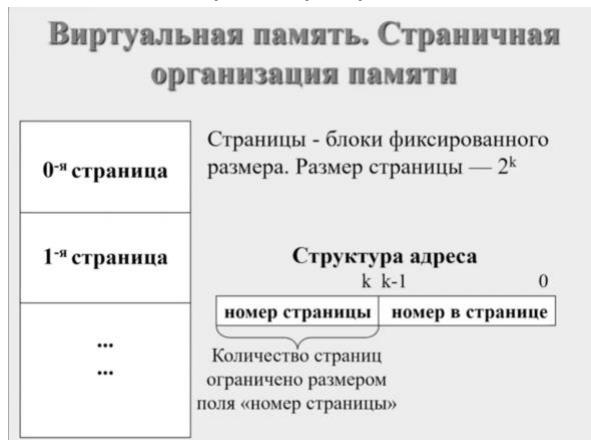
Реализацией одной из моделей аппарата виртуальной памяти является **аппарат базирования адресов**. Механизм базирования адресов основан на двойкой интерпретации получаемых в ходе выполнения программы исполнительных адресов (Аисп. прог.). С одной стороны, его можно интерпретировать как **абсолютный исполнительный адрес**, когда физический адрес в некотором смысле соответствует исполнительному адресу программы (Аисп. физ.= Аисп. прог.). С другой стороны, исполнительный адрес программы можно проинтерпретировать как **относительный адрес**, т.е. адрес, зависящий от места дислокации программы в ОЗУ (адрес относительно точки загрузки программы). Иными словами, имеется оперативная память с ячейками с номерами от 0 до некоторого L-1, и, начиная с некоторого адреса K, расположена программа. Тогда адрес Аисп. прог. внутри программы можно трактовать, как отступ от физической ячейки с адресом K на величину Аисп. прог.. Для реализации модели базирования используется специальный регистр базы, в который в момент загрузки процесса в оперативную память операционная система записывает начальный адрес загрузки (т.е. K). Тогда реальный физический адрес получается, исходя из формулы Аисп. физ.= Аисп. прог.+Rбазы. Это считает сам процессор. Регистр базы находится в процессоре

Но при этом необходимо помнить, что программа представляется в виде непрерывной области виртуальной памяти, которая загружается в непрерывный фрагмент физической памяти.

Развитием аппарата виртуальной памяти является **аппарат страничной организации памяти**.

Страницчная организация памяти предполагает разделение всего пространства ОЗУ на блоки одинакового размера – страницы. Обычно размер страницы равен  $2^k$ .

## БИЛЕТ 14 Пример организации страничной виртуальной памяти



Развитием аппарата виртуальной памяти является **аппарат страницной организации памяти**. Данная модель представляет все адресное пространство оперативной памяти в виде последовательности блоков фиксированного размера, называемых страницами. **Страница** — это область адресного пространства фиксированного размера: обычно размер страницы кратен степени двойки — будем считать, что размер страницы  $2^k$ . Тогда структура адреса представима в виде двух полей: правые  $k$  разрядов представляют адрес внутри страницы, а оставшиеся разряды отвечают за номер страницы. Тогда количество виртуальных страниц в системе ограничено разрядностью поля «Номер виртуальной страницы» в адресе.

Итак, **виртуальное адресное пространство** — это множество виртуальных страниц, доступных для использования в программе. **Физическое адресное пространство** — это оперативная память, подключенная к данному компьютеру. Физическая память может иметь произвольный размер по отношению к размеру виртуальной памяти (число физических страниц может быть меньше, больше или равно числу виртуальных страниц).

И виртуальная и физическая память разделены на страницы. Номер в страницы одинаковый.

В центральном процессоре имеется аппаратная (регистровая) таблица, называемая **таблицей страниц**, предназначенная для организации отображения между виртуальными и физическими адресами при страницной организации памяти. Количество строк в этой таблице определяется максимальным числом виртуальных страниц, которое зависит от схем работы процессора и максимальной адресной разрядности процессора. Каждой виртуальной странице исполняемой программы ставится в соответствие строка таблицы страниц с тем же номером. Внутри каждой записи таблицы страниц находится номер физической страницы, в которой размещается соответствующая виртуальная страница программы. Соответственно, аппарат виртуальной страницной памяти позволяет **автоматически** (т.е. аппаратно) преобразовывать номер виртуальной страницы в номер физической страницы посредством обращения к таблице страниц.

# Виртуальная память. Страницчная организация памяти

Модельный пример организации страницочной виртуальной памяти



## Билет 15. Многомашинные, многопроцессорные ассоциации.

### Классификация. Примеры.

В зависимости от степени интегрированности машин в рамках одного комплекса различают многопроцессорные ассоциации, где степень связности машин довольно велика, и многомашинные ассоциации, в которых наблюдаются слабые связи между машинами (в некоторых случаях говорят о сетях ЭВМ).

Одна из наиболее простых классических классификаций многопроцессорных систем — это **классификация по Флинну**, основанная на анализе некоторых характеристик потоков информации в машине. Основная концепция этой классификации — перебор всевозможных характеристик потока команд (инструкций) и потока данных. Обработка каждого из этих потоков может быть одиночная либо множественная.

В контексте машины можно выделить два потока информации: **поток управления** (для передачи управляющих воздействий на конкретное устройство) и **поток данных** (циркулирующий между оперативной памятью и внешними устройствами).

Флинн предлагает рассматривать компьютер с позиции 2 потоков: - поток команд: выбор одной или группы команд - поток данных, operandов: с выполнением каждой команды выбирается либо единичная, либо множественная порция данных. В результате получаем четыре класса архитектур:

— **ОКОД** (одиночный поток команд, одиночный поток данных, или SISD) — это традиционные компьютеры (близкие машине фон Неймана) с единственным ЦП. Они имеют одно устройство управления, которое последовательно выбирает команды, и каждая команда обрабатывает единичную порцию данных.

— **ОКМД** (одиночный поток команд, множественный поток данных, или SIMD) — например, векторные компьютеры, способные оперировать векторами данных, матричная обработка данных. Обычно для этих целей в данных машинах существуют векторные регистры, а также обычно имеются векторные операции, предполагающие векторную обработку. В этой архитектуре имеется одно УУ, которое последовательно выбирает команды, а обработка данных ведётся агрегировано. Заметим, что принадлежность конкретной системы к конкретному классу относительно условна.

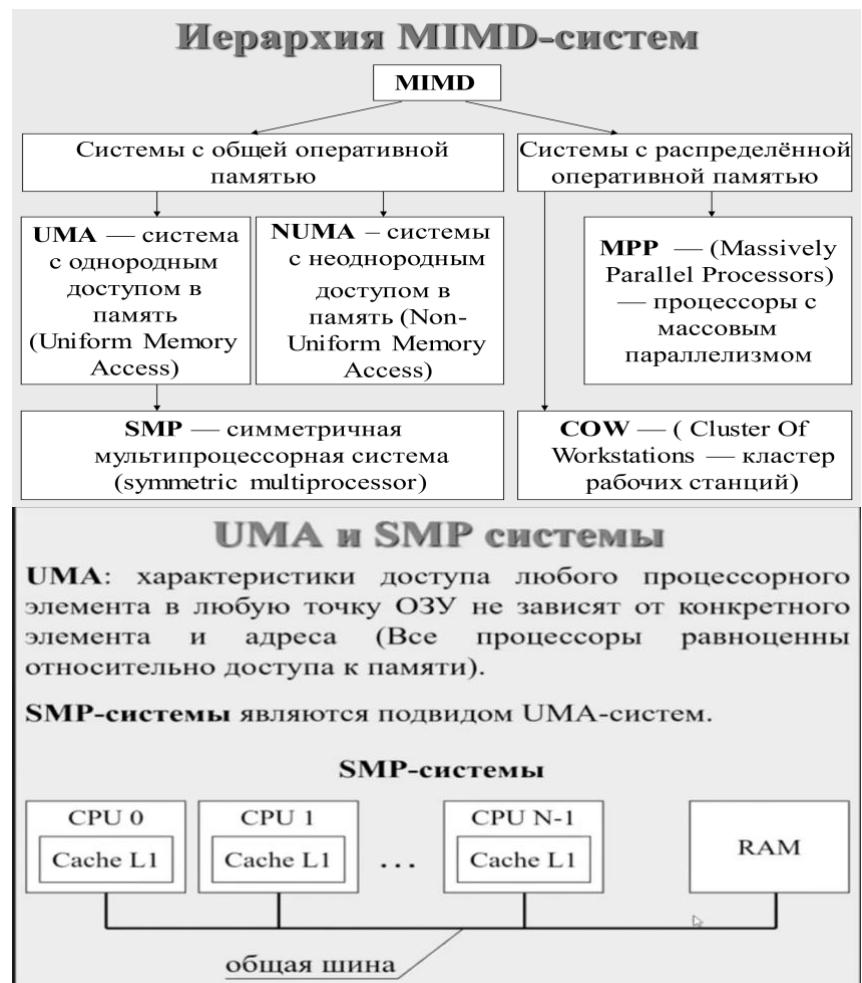
– **МКОД** (множественный поток команд, одиночный поток данных, или MISD) — имеется смесь команд, которая оперирует над одними и теми же данными. Этот класс архитектур является спорным. Существуют различные точки зрения о существовании каких-либо систем данного класса, и если таковые имеются, то какие именно. В некотором смысле сюда можно отнести специализированные системы обработки видео- и аудиоинформации (DSPпроцессоры), а также конвейерные системы.

– **МКМД** (множественный поток команд, множественный поток данных, или MIMD) — это системы, которые содержат не менее двух устройств управления (это может быть один сложный процессор с множеством устройств управления). Множество процессоров одновременно выполняют различные последовательности команд над своими данными. Это наиболее распространённая категория архитектур. На сегодняшний день данная категория во многом определяет свойства и характеристики многопроцессорных и параллельных вычислительных систем

Среди систем МКМД можно выделить два подкласса: **системы с общей оперативной памятью** и **системы с распределенной памятью**. Для систем первого типа характерно то, что любой процессор имеет непосредственный доступ к любой ячейке этой общей оперативной памяти.

**UMA** — к любой ячейке я могу добраться за одинаковое время

**SMP** — обмен между двумя устройствами блокирует общую шину. Кол-во ЦП определяется пропускной способностью шины.



**SMP-системы**

**Синхронизация кэша.**

**Поведение кэш-памяти с отслеживанием при чтении/записи**

Операции	Локальный кэш	Кэш других процессоров
Промах при чтении	Запись из памяти в кэш	Ничего
Попадание при чтении	Использование кэша	Ничего (операция «не видна»)
Промах при записи	Запись в память	Соответствующая запись в кэше удаляется
Попадание при записи	Запись в память и кэш	Соответствующая запись в кэше удаляется

NUMA: локальная память – память с которой наиболее скоростная работа. Контроллер определяет, находится ли информация в локальной памяти.

## Иерархия MIMD-систем

**Системы с распределённой оперативной памятью**  
представляются как объединение компьютерных узлов, каждый из которых состоит из процессора и ОЗУ, непосредственный доступ к которой имеет только «свой» процессорный элемент. Класс наиболее перспективных систем.

### Виды систем с распределённой оперативной памятью

- MPP — Massively Parallel Processors
- COW — Cluster Of Workstations

## MPP-системы

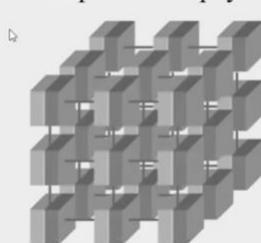
**MPP** — Специализированные дорогостоящие ВС. Эти компьютеры могут выстраиваться, процессорные элементы могут объединяться в различные топологии: макроконвейер, n-мерный гиперкуб и др.

### Примеры топологий MPP-систем

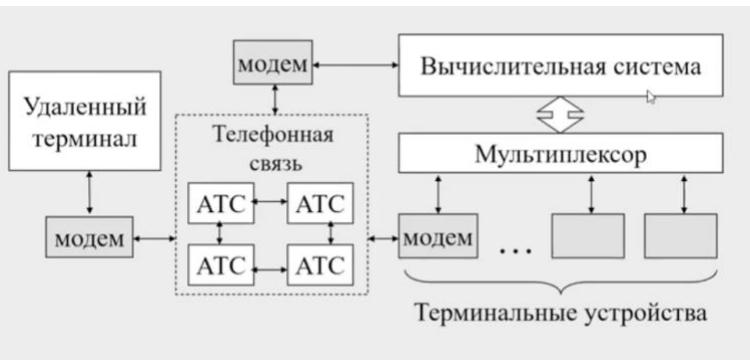
Макроконвейер:



3-мерный гиперкуб:



## БИЛЕТ 16. Терминальные комплексы. Компьютерные сети.



Современные многопроцессорные системы строятся как специализированные компьютерные сети. Основой современных компьютерных сетей было появление терминальных комплексов. **Терминальный комплекс** — это многомашинная ассоциация, предназначенная для организации массового доступа удаленных и локальных пользователей к ресурсам некоторой вычислительной системы. Суть ТК заключается в следующем.

Пусть имеется некая вычислительная система. Ставится задача организовать массовый доступ к ее ресурсам. Под ресурсами в данном случае могут пониматься, например, высокая производительность рассматриваемой системы или же информационный ресурс

Терминальные комплексы предполагают наличие в своем составе следующих компонентов: – основная вычислительная система (к которой обеспечивается доступ); – локальные мультиплексоры; – локальные терминалы; – модемы; – удаленные терминалы; – удаленные мультиплексоры.

Терминал – устройство, через которое можно получить доступ к информации

Терминальные устройства можно разделить на локальные и удаленные. **Локальные** терминальные устройства имеют либо непосредственное подключение к ВС по локальному каналу связи, либо подключение через мультиплексор (к одному порту можно подключить два и более терминальных устройств ПРИМЕР – юзб хаб). **Удаленные** терминальные устройства подключаются к ВС через коммуникационную среду. Нужны каналы связи для этого

Модем - переводило цифровой в аналоговый (телефон)

Линии связи/каналы. Существуют три критерия, по которым можно классифицировать каналы. Во-первых, с точки зрения организации канала все каналы можно поделить на **коммутируемые и выделенные**.

- **Коммутируемый канал** — это линия, выделяемая на весь сеанс работы терминального устройства. Примером коммутируемого канала может служить телефонный разговор. (**недостаток – помехи, плохая связь. Если перезвонить то может выделится другой канал и связь будет лучше**)
- **Выделенный канал** обеспечивает связь терминального устройства с ВС на постоянной основе. (Это зло. )

Во-вторых, все каналы можно поделить по количеству участников общения: –

- **канал точка-точка** — одно устройство общается с одним устройством (подключение к удалённому терминалу без мультиплексирования);
- **многоточечный канал** — общение многих устройств (например, при мультиплексировании): подключение терминала осуществляется через локальный мультиплексор.

И, наконец, каналы можно делить с точки зрения направления движения информации в канале:

- **симплексный (по одному направлению. Пример – человек с мегафоном)**
- **дуплексный** (двунаправленный) (Разговор по телефону)
- **полудуплексный** (рация – ПРИЕМ и ПЕРЕДАЧА )

Развитие терминальных комплексов положило основу развития компьютерных сетей. И первым шагом к сетям стала замена терминальных устройств компьютерами.

**Компьютерная сеть** — это объединение компьютеров (или вычислительных систем), взаимодействующих через коммуникационную среду. **Коммуникационная среда** — каналы и средства передачи данных.

1. Большое число связанных узлов, обеспечивающих решение определённых задач
2. Возможность распределения обработки информации
3. Расширяемость сети (сеть должна обеспечивать развитие сети по протяжённости, по расширению пропускной способности каналов, по составу и производительности узлов, входящих в состав сети)
4. Применение симметричных интерфейсов обмена информации внутри сети (возможность произвольного распределения функций внутри сети)

**Абонентские** или основные компьютеры — **хосты** и **Коммуникационные** или вспомогательные компьютеры (шлюзы, маршрутизаторы, ...) С точки зрения взаимодействия компьютеров в сети, традиционно для функционирования компьютерных сетей используются 3 модели организации каналов — это

**сети коммутации каналов** – сеть обеспечивает канал на весь сеанс связи

**сети коммутации сообщений** – в виде последовательности сообщений (не требуется канал. Сообщение идет в узлы и там уже сообщения будут передвигаться по узлам)

**сети коммутации пакетов** – любое сообщение разбивается на пакеты определенного размера (в пакете Заголовок и Данные). Пакеты могут приходить в разном порядке.

Сразу отметим, что большинство современных сетей являются комбинациями этих основных моделей сетей.

**Сообщение** — логически целостная порция данных, имеющая произвольный размер.

Взаимодействие абонентов осуществляется сеансами связи. **Сеанс связи** состоит из обмена сообщениями между абонентами. Начало/завершение сеанса связи. **Сеть коммутации каналов** обеспечивает выделение коммуникаций абонентам на весь сеанс связи. Взаимодействие представляется в виде последовательности обменов сообщениями. Каждое сообщение разбивается на блоки фиксированного размера - **пакеты**.

Структура пакета (заголовок, данные)

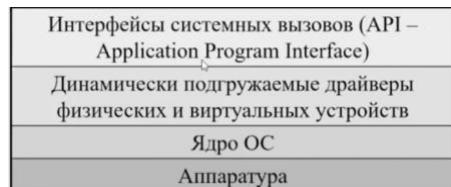
## 17 Операционные системы. Основные компоненты и логические функции. Базовые понятия: ядро, процесс, ресурс, системные вызовы. Структурная организация ОС.

**Операционная система** — комплекс программ, обеспечивающий контроль за существованием, распределением и использованием ресурсов ВС.

**Процесс** — совокупность машинных команд и данных, обрабатываются в рамках ВС и обладающая правами на владение некоторым набором ресурсов.

Любая ОС должна удовлетворять следующим свойствам: надежность, защита, эффективность, предсказуемость

Аппаратура не включена в ОС.



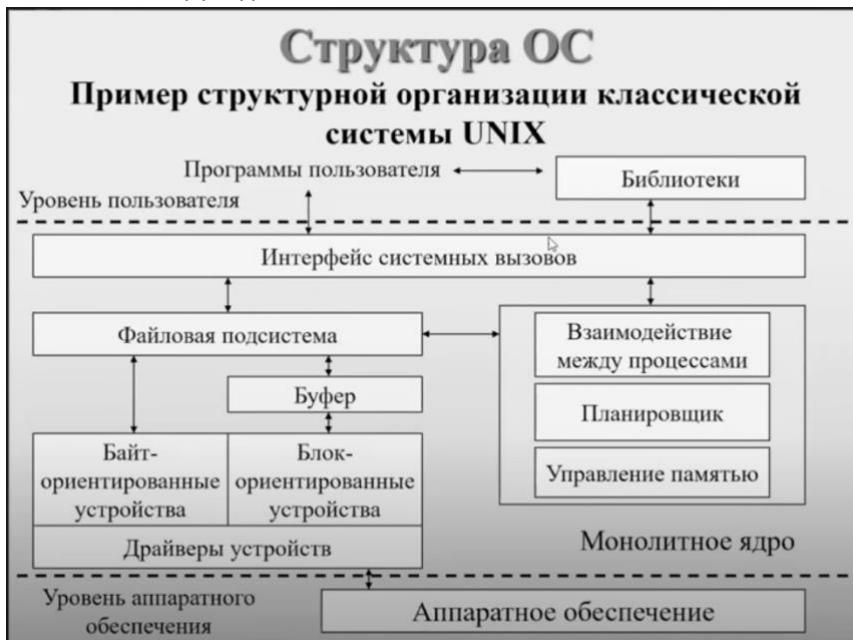
### Типовая структура ОС.

**Ядро** – резидентная часть (это часть ОС которая в оперативной памяти все время) ОС, работающая в режиме супервизора. В ядре размещаются программы обработки прерываний и драйверы наиболее «ответственных» устройств. Это могут быть и физические, и виртуальные устройства. Например, в ядре могут располагаться драйверы файловой системы, ОЗУ. Обычно ядро работает в режиме **физической адресации (не используется виртуальная память)**. Состоит из некоторых программ базового функционала и машинных команд.

Следующие уровни структуры – **динамически подгружаемые драйверы физических и виртуальных устройств**. Это драйверы, добавление которых в систему возможно «на ходу» без перекомпоновки программ ОС. Они могут являться резидентными и нерезидентными, а также могут работать как в режиме супервизора, так и в пользовательском режиме. Резидентные драйверы подгружаются в систему в процессе ее загрузки и находятся в ней до завершения ее работы. Примером резидентного драйвера может быть драйвер физического диска. Нерезидентные драйверы вызываются операционной системой на сеанс работы с соответствующими устройствами (например, драйвер флэш-памяти). **Они подгружается в ОЗУ для надобности**

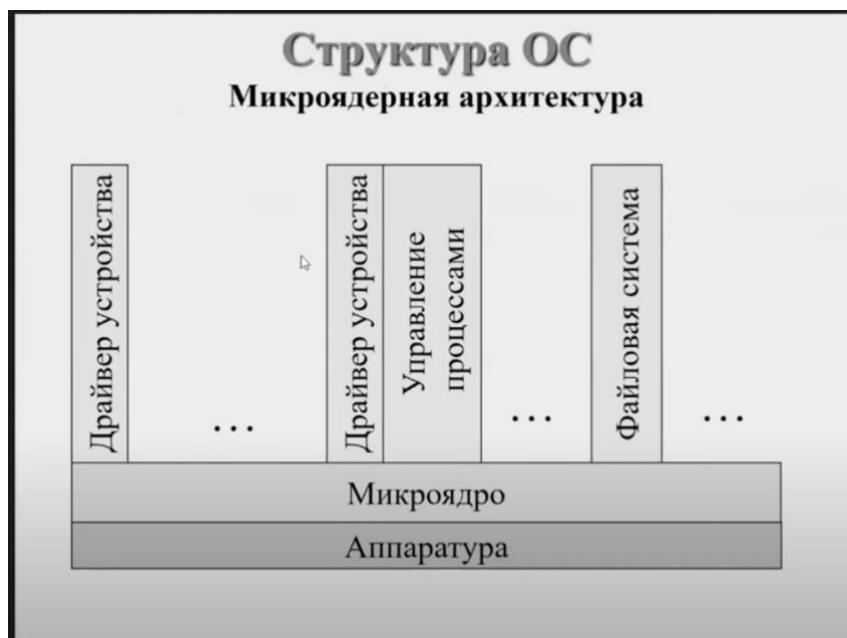
И, наконец, некоторой логической вершиной рассматриваемой структуры ОС будут являться **интерфейсы системных вызовов** (API — Application Program Interface). Под системным вызовом будем понимать средство обращения процесса к ядру операционной системы за выполнением той или иной функции (возможности, услуги, сервиса).

Итак, существует несколько подходов к структурной организации операционных систем. Один из них можно назвать **классическим**: он использовался в первых операционных системах и используется до сих пор — это подход, основанный на использовании **монолитного ядра**. В этом случае ядро ОС представляет собою единую монолитную программу, в которой отсутствует явная структуризация, хотя, конечно, в ней есть логическая структуризация.



Монолитное ядро это и есть ядро. Это отдельная программа.

Альтернативу данному подходу предлагают **многослойные операционные системы**. В этом случае все уровни разделяются на некоторые функциональные слои. Здесь можно провести аналогию с моделью сетевых протоколов. Между слоями имеются фиксированные интерфейсы. Управление происходит посредством взаимодействия соседних слоев



Третий подход предлагает использовать **микроядерную архитектуру**. Функционирование операционных систем подобного типа основывается на использовании т.н. микроядра. В этом случае выделяется минимальный набор функций (например, первичная обработка прерываний и некоторые функции

управления процессами, управление ОЗУ), которые включаются в ядро. Микроядро обработка переходов от драйвера к драйверу. Вся остальная функциональность представляется в виде драйверов, которые подключаются к ядру посредством некоторого стандартного интерфейса. Это такой конструктор. Удобный для изменения и модификаций. **Недостаток:** накладные расходы по перемещению от драйвера к драйверу.

Накладные расходы - действия системы которые не связаны с обработкой содержательного запроса.

Можно выделить следующие основные логические функции ОС:

1. управление процессами; (обработка событий в процессе)
2. управление ОП; (стратегии выделение памяти)
3. планирование; (распределение времени работы ЦП. Какому процессу надо выделить время работы процесса)
4. управление устройствами и ФС. (определяет стратегию того что может подключаться к компю). В UNIX устройство это файл. У нас унифицированный набор устройств для работы с файлами.
5. Сетевое взаимодействие (нужен доступ за обновлениями). В них глубоко интегрированно сетевое взаимодействие
6. Безопасность.

## 18. Операционные системы. Пакетная ОС, ОС разделения времени, ОС реального времени.

### ТИПЫ ОС:

- **Пакетная**
- **Системы разделения времени**
- **Системы реального времени**

### Пакетная ОС

**Пакет программ** – совокупность программ, для выполнения каждого из которых требуется некоторое время работы процессора. Этот тип был на первых компьютерах. Пакет программ – стопка перфокарт. Критерий эффективности – минимизация накладных расходов.

#### **Стратегия переключения с одного процесса на другой, если**

- а) выполняемый процесс завершен
- б) возникло прерывание по обмену в выполняемом программе
- в) зафиксировался факт зацикливания.

Они нужны для обработки крупных вычислений.

Минусы:

- Максимально нагружен процессор (из за обработок процессов)
- Невозможно с ней обрабатывать программу разделения времени (допусти текстовый редактор)

## Системы разделения времени

Основывается на том как мы планируем распределение времени работы ЦП с определенным процессов

**Квант времени ЦП** – некоторый фиксированный ОС промежуток времени работы ЦП

ЦП предоставляет процессу на один квант времени. Меняя размер кванта можно получить различные характеристики ОС. Большой квант времени удобен для отладки.

Если квант времени устремить к нулю, то у пользователя создается впечатление, что он работает один на этой ОС. Это происходит потому, что критерий эффективности с точки зрения человека – через сколько компьютер реагирует на действия человека.

**Переключение выполнения процессов** происходит только в одном из случаев:

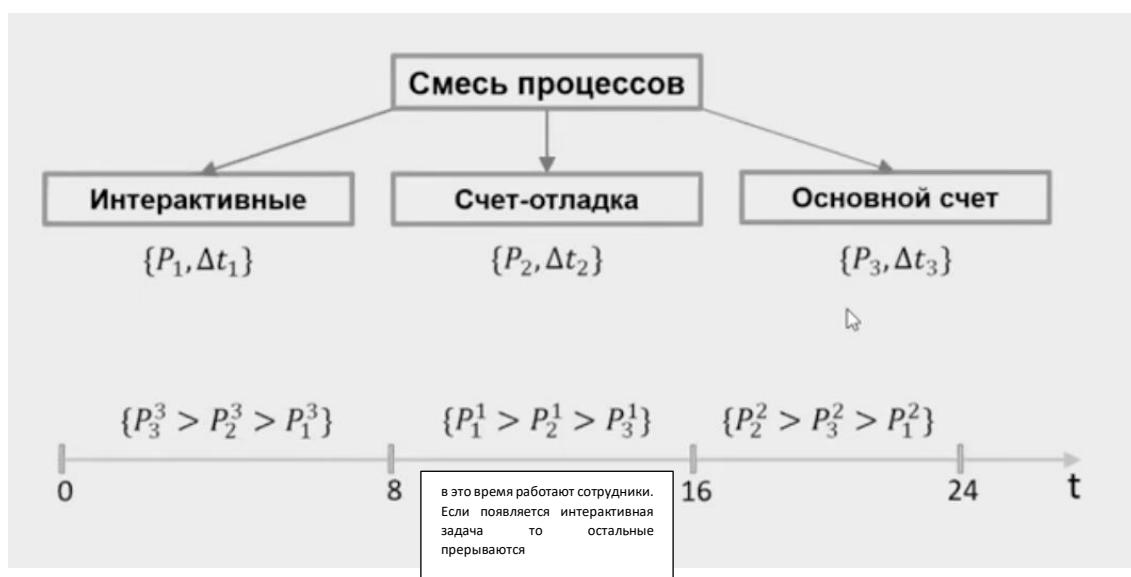
- Исчерпался выделенный квант времени
- Выполнение процесса завершено
- Возникло прерывание
- Был фиксирован факт зацикливания процесса

Минусы:

- Накладные расходы

Модель планирования процессов.

P – приоритет.



## Системы реального времени

являются специализированными системами в которых все функции планирования ориентированы на обработку некоторых событий за время, не превосходящее некоторого предельного значения

**Критерий качества** – обработка любого события за некоторый гарантированный промежуток времени (бортовой компьютер, автопилот – есть датчик высоты и есть время обработки высоты. Время обработки должно быть таким чтобы он удачно сел).

Реально (за исключением систем реального времени, которые могут быть разные по областям применения, важности серьезности и т.д.) используются комбинации пакетных и систем разделения времени друг в друге и с различными стратегиями

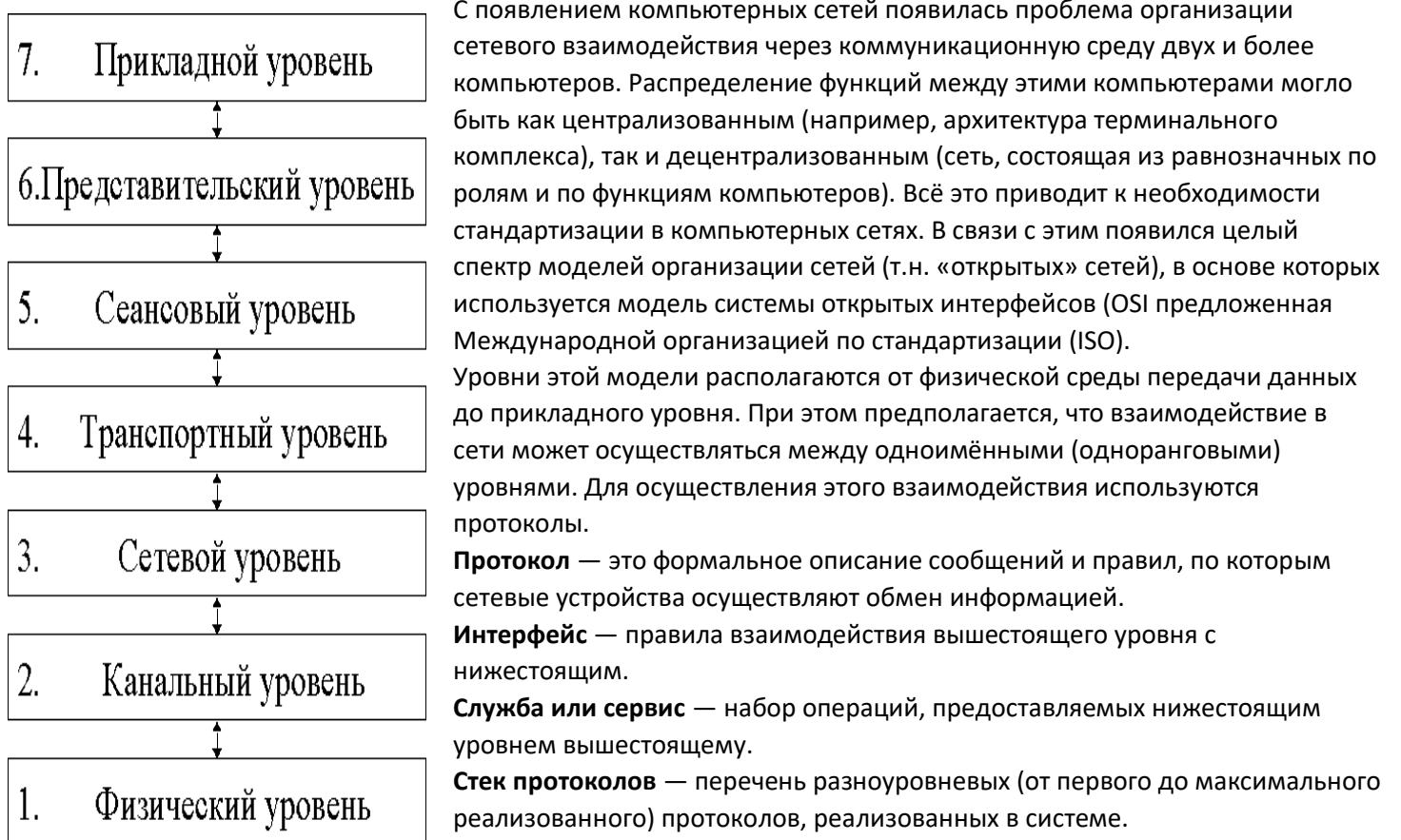
## Сетевые, распределенные ОС

Сетевая ОС – В первую категорию можно отнести т.н. сетевые ОС (Рис. 75). Сетевая операционная система – это система, обеспечивающая функционирование и взаимодействие вычислительной системы в пределах сети. Это означает, что сетевая ОС устанавливается на каждом компьютере сети и обеспечивает функционирование распределенных приложений, т.е. тех приложений, реализация функций которых распределена по разным компьютерам сети. ПРИМЕР – файловая система

Распределенная ОС – Вторую категорию составляют распределенные ОС (Рис. 76). Распределенной операционной системой считается система, функционирующая на многопроцессорном или многомашинном комплексе, в котором на каждом из узлов функционирует отдельное ядро, а сама система обеспечивает реализацию распределенных возможностей ОС. Пример – ОС на кластерном суперкомпьютере.

Проблема распределения файловой системы

## 19. Организация сетевого взаимодействия. Эталонная модель ISO/OSI. Протокол, интерфейс. Стек протоколов. Логическое взаимодействие сетевых устройств



**Канальный уровень** (или уровень передачи данных). На этом уровне решаются задачи обеспечения передачи данных по физической линии, обеспечения доступности физической линии, обеспечения синхронизации (например, передающего и принимающего узлов), а также задачи по борьбе с ошибками. Канальный уровень манипулирует порциями данных, которые называются кадрами. В кадрах присутствует избыточная информация для фиксации и устранения ошибок.

**Сетевой уровень.** На этом уровне решаются задачи взаимодействия сетей: обеспечивается управление операциями сети (в т.ч. адресация абонентов, маршрутизация (поиск маршрутов)), а также обеспечивается связь между взаимодействующими сетевыми устройствами.

**Транспортный уровень.** На данном уровне обеспечивается корректная транспортировка данных и

взаимодействие между программой-отправителем и программой-получателем данных, т.е. обеспечивается программное взаимодействие (а не взаимодействие устройств). На этом же уровне принимается решение о выборе типа транспортных услуг (транспортировка данных с установлением виртуального канала или же без этого).

**Сеансовый уровень.** Этот уровень обеспечивает управление сеансами связи. На этом уровне решаются задачи определения активной стороны, подтверждения полномочий и паролей, а также решаются задачи организации меток, или контрольных точек по сеансу, которые отражают состояние сеанса связи и позволяют в случае возникновения сбоя восстанавливать сеанс с последней контрольной точки (т.е. повторять передачу не с начала, а с последней установленной контрольной точки).

**Уровень представления данных** обеспечивает унификацию используемых в сети кодировок и форматов передаваемых данных.

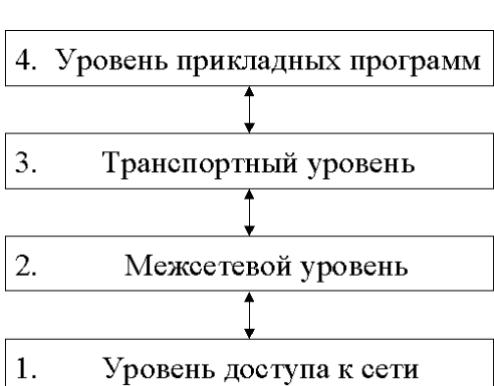
**Прикладной уровень** (уровень прикладных программ). На этом уровне формализуются правила взаимодействия с прикладными системами.



## Билет 20 Семейство протоколов TCP/IP

TCP/IP -> интернет (как следствие)

Рассмотрим еще одну модель организации сетевого взаимодействия — семейство протоколов TCP/IP (Рис. 68). Это классическая четырехуровневая модель организации сетевого взаимодействия. Протоколы семейства TCP/IP основаны на сети коммутации пакетов.



- **Уровень доступа к сети.** Этот уровень соответствует физическому и канальному уровням модели ISO/OSI. Уровень доступа к сети специфицирует доступ к физической сети. На этом уровне решаются проблемы сетевого адаптера, драйвера сетевого адаптера и проблемы среды передачи данных. (кабель, радио канал. У нас есть стандарты разъема. Как передаются сигналы и все такое) ПРИМЕР – протокол Ethernet (разработка с компанией Xerox). **Ethernet** – магистральная сеть, единый кусок кабеля, к которому подключаться сетевые устройства. Но если у нас в лок сети много компов, то начинаются конфликтные ситуации (аналогия с двумя вежливыми людьми и проемом)

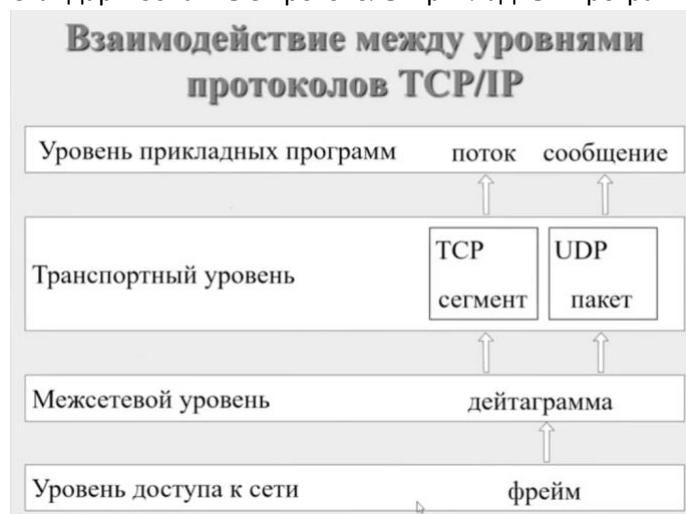
- **Межсетевой уровень** (или internet-уровень). В некотором смысле ему соответствует сетевой уровень модели ISO/OSI. Т.е. на этом уровне решаются проблемы адресации и маршрутизации по сети. В отличие от сетевого уровня модели OSI, межсетевой уровень модели TCP/IP не устанавливает соединений с другими машинами.
- **Транспортный уровень.** Этому уровню соответствуют сеансовый и транспортный уровни модели ISO/OSI. Транспортный уровень модели TCP/IP обеспечивает доставку данных от компьютера к

компьютеру, обеспечивает средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. На этом уровне имеется возможность использования протоколов передачи, которые устанавливают виртуальное соединение или не устанавливают его.

- **Уровень прикладных программ.** Этот уровень разрешает проблемы уровня представления и уровня прикладных программ модели ISO/OSI. Уровень прикладных программ модели TCP/IP состоит из прикладных программ и процессов, использующих сеть и доступных пользователю. В отличие от модели OSI, прикладные программы сами стандартизуют представление данных. Эти уровни модели TCP/IP являются пакетными: на каждом уровне система оперирует порциями данных, обладающими характеристиками соответствующего уровня

Итак, основные свойства протоколов семейства TCP/IP следующие:

- Устойчивая работа в недетерминированных условиях линий связи
- Открытость (доступность для использования) стандартов протоколов
- Независимость от аппаратного обеспечения сети передачи данных (за счёт наличия уровня доступа к сети)
- Унифицированная модель именования сетевых устройств
- Стандартизованные протоколы прикладных программ



Шина может быть перегружена и появляется очень много отказов. (по аналогии с UMA и NUMA)

**Межсетевой уровень.** Протокол IP (internet protocol) — один из основных протоколов, в честь которого получило название всё семейство TCP/IP. Это уникальное имя для устройства в сети. Данный протокол реализует следующие функции: – формирование дейтаграмм; – поддержание системы адресации; – обмен данными между транспортным уровнем и уровнем доступа к сети; – организация маршрутизации дейтаграмм; – разбиение и обратная сборка дейтаграмм.

# Система адресации протокола IP

32 бита



Класс А



Класс В



Класс С



Класс D

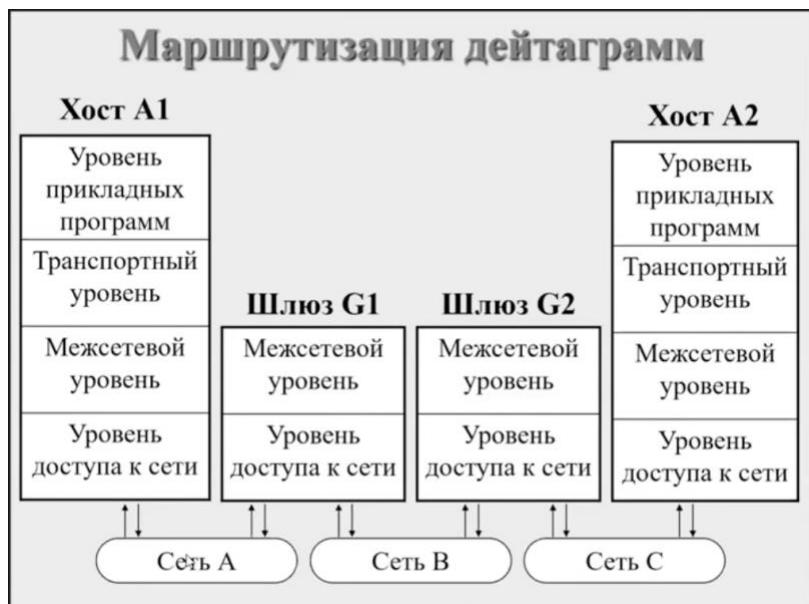


Класс Е



Класс А – это для ведущих мировых компаний. Хост – номера устройств ( $2^{24}$  устройств)

Класс В



Два компьютера A1 и A2. A1 хочет обратиться к A2. Он укажет его IP. Пакеты пойдут через шлюзы. Поможет в этом маршрутизатор. Устройства G1 и G2 имеют только два уровня сетевой организации. Им не нужно больше. Также они имеют IP так как имеют два и более сетевых устройства. У них будут адреса которые принадлежат сети A и B (они будут разные).

**Пакет** – это блок данных, который передаётся вместе с информацией, необходимой для его корректной доставки. Каждый пакет перемещается по сети независимо от остальных.

**Дейтаграмма** – это пакет протокола IP. Контрольная информация занимает первые пять или шесть 32-битных слов дейтаграммы. Это её заголовок (header).

**Шлюз** – устройство, передающее пакеты между различными сетями

**Маршрутизация** – процесс выбора шлюза или маршрутизатора

## Транспортный уровень

**Протокол контроля передачи (TCP, Transmission Control Protocol)** - обеспечивает надежную доставку данных с обнаружением и исправлением ошибок и с установлением логического соединения. При TCP есть сеанс связи по виртуальному каналу. Пакетная передача. Он отправляет пакеты и отправив его, отправитель ждет пакет от получателя (приходит уведомление о том что пакет дошел). Если подтверждения не пришло, то отправка отправляется. **Из минусов:** Он нагружает этими уведомлениями сеть.

**Протокол пользовательских дейтаграмм (UDP, User Datagram Protocol)** - отправляет пакеты с данными, «не заботясь» об их доставке. Нет виртуального канала! Но имеет наименьшую нагрузку на сеть. Пример: телефонная связь.

## 21. Управление процессами. Определение процесса, типы. Жизненный цикл, состояния процесса. Свопинг. Модели жизненного цикла процесса. Контекст процесса

Под **процессом** понимается совокупность машинных команд и данных, обрабатываемая в вычислительной системе и обладающая правами на владение некоторым набором ресурсов ВС. Команды и данные которые находятся в трех состояниях – **исполнение, ожидание** завершения запроса к ОС (процессу что то потребовалось и он сделал запрос в ОС (запрос по обмену с ВЗУ)), процесс готов стать исполняемым

Также уже говорилось, что ресурсы могут декларироваться посредством различных стратегий, причем ресурс может эксклюзивно принадлежать одному единственному процессу (допустим процессор), а может быть **разделяемым**, когда ресурс может одновременно принадлежать нескольким процессам (). Ресурсы могут быть аппаратными (ОЗУ ЦП) так и программными (виртуальные ресурсы – открытие файла из ФС).

Когда процесс становится исполняемым то что ОС передала ресурс процессор. И процессор стал принадлежать этому процессу.

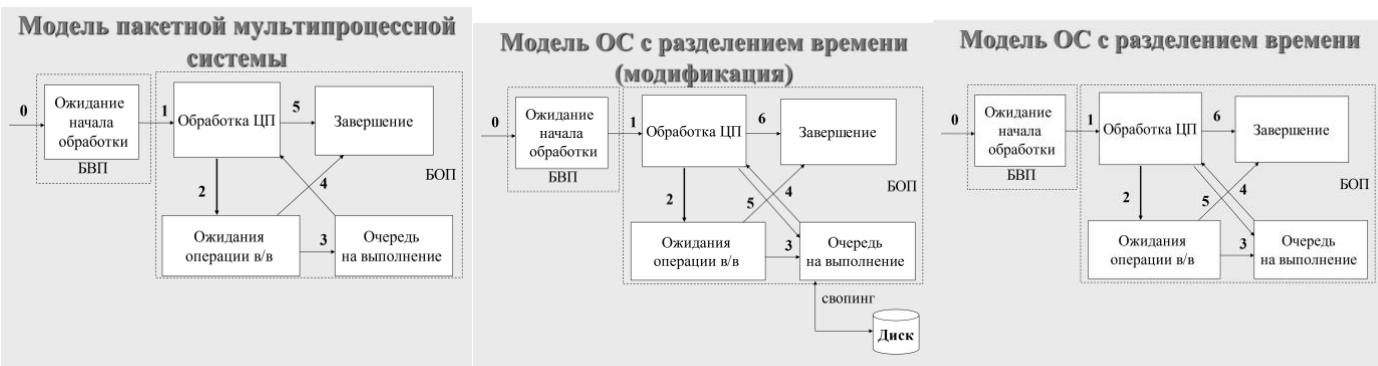
Одной из основных задач ОС является поддержание жизненного цикла процесса. **Жизненный цикл процесса** – это те этапы, через которые может проходить процесс с момента его создания, в ходе его обработки и до завершения в рамках вычислительной системы. Выделим следующие типовые этапы жизненного цикла процесса:

- образование процесса,
- обработка (выполнение) процесса на процессоре,
- ожидание постановки процесса на выполнение (обобщенное со вторым)
- завершение процесса (процесс завершается не мгновенно, мы освобождаем ресурсы)

Рассмотрим Модельную ОС

Будем считать, что ОС обеспечивает существование процессов в двух состояниях. Первое состояние – это размещение процесса, или программы, в **буфере ввода процессов** (БВП). В этом буфере размещаются процессы с момента их формирования, или ввода в систему, до начала обработки его центральным процессором. Второе состояние объединяет состояния процесса, связанные с размещением процесса в **буфере обрабатываемых процессов** (БОП), т.е. будем считать, что все процессы, которые начали обрабатываться центральным процессором, размещаются в данном буфере

Начнем рассмотрение с модели **пакетной однопроцессной системы** (Этапы – формирование процесса и ожидание начала обработки, – обработка (переход из БВП в БОП) – завершение процесса, освобождение системных ресурсов)



Свопинга- механизм откачки процесса во внешнюю память. Что освобождать ресурсы ОЗУ

Процесс (или **полновесный процесс**) – является объектом планирования и выполняется внутри защищённой области памяти. Альтернативой являются т.н. **легковесные процессы**, известные также как **нити** (или потоки), – это процессы, которые могут активироваться внутри полновесного процесса, могут быть объектами планирования, и при этом они могут функционировать внутри общей (т.е. незащищённой от других нитей) области памяти.

Тогда определение процесса можно обобщить – понятие «процесса» включает в себя следующее:

- исполняемый код;
- собственное адресное пространство, представляющее собой множество виртуальных адресов, которые может использовать процесс;
- ресурсы системы, которые назначены процессу операционной системой;
- хотя бы одну выполняемую нить.

Под **контекстом процесса** будем понимать совокупность данных, характеризующих актуальное состояние процесса. Обычно контекст процесса состоит из трёх компонент:

- **пользовательская составляющая** — это текущее состояние программы (т.е. совокупность машинных команд и данных, размещённых в ОЗУ и характеризующих выполнение данного процесса);
- **аппаратная составляющая** — отражает актуальное состояние центрального процессора в момент выполнения данного (информация о содержимом регистров)
- **системная составляющая** — это структуры данных операционной системы, содержащие характеристики процесса (PID процесса)

## 22 Реализация процессов в ОС UNIX. Определение процесса. Контекст, тело процесса. Состояния процесса. Аппарат системных вызовов в ОС UNIX.

### Понятие «процесс»

Понятие «**процесс**» включает в себя следующее:

- исполняемый код
- собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс
- ресурсы системы, которые назначены процессу ОС
- хотя бы одну выполняемую нить

### Определение процесса Unix

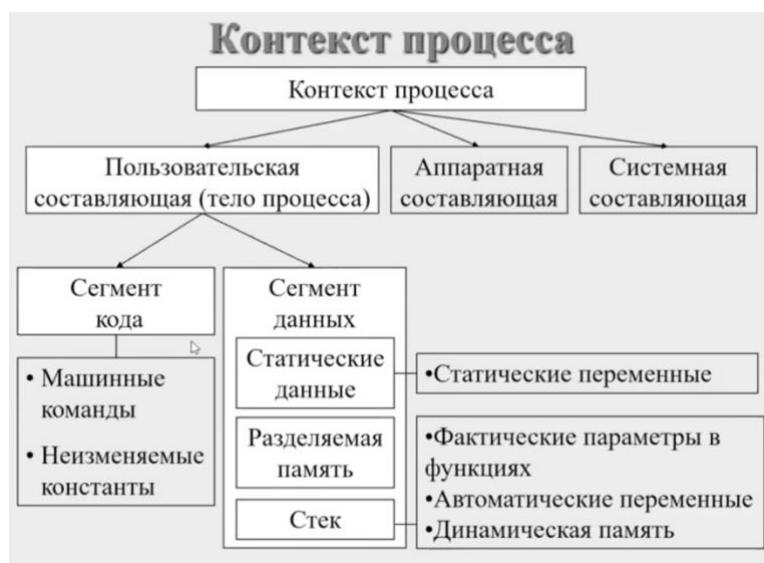


С точки зрения понимания термина процесса в ОС Unix, данное понятие можно определить двояко. С одной стороны, **процесс UNIX** можно определить как объект, зарегистрированный в таблице процессов операционной системы. **Контекст процесса – совокупность данных, характеризующих состояние процесса.** **Таблица процессов** – одна из специальных системных таблиц, которая является программной таблицей. Второе определение объявляет **процессом** объект, порожденный системным вызовом `fork()`.

Таблица процессов обеспечивает уникальное именование процессов: она устроена позиционным образом, т.е. именование процесса осуществляется посредством номера записи таблицы, соответствующей данному процессу (нумерация строится от нуля до некоторого фиксированного значения). Этот номер записи называется **идентификатором процесса. (PID )**



Каждая запись таблицы процессов имеет ссылку на контекст процесса, который структурно состоит из пользовательской, системной и аппаратной составляющих.



**Пользовательская (программная) составляющая** — это тело процесса. Обычно тело процесса состоит из двух частей: сегмент кода и сегмент данных. **Сегмент кода** содержит машинные команды и неизменяемые константы; сегмент кода — это обычно не изменяемая программным способом часть тела. Также в пользовательскую составляющую входит сегмент данных, включающий область статических данных процесса, область разделяемой памяти, а также область стека.

В ОС Unix реализована такая возможность, как **разделение сегмента кода**

**Аппаратная составляющая** включает в себя все регистры, аппаратные таблицы процессора и т.д., характеризующие актуальное состояние процесса в момент его выполнения на процессоре. (счетчик команд, регистр состояния процессора)

**Системная составляющая** содержит системную информацию об идентификации процесса системную информацию об открытых и используемых в процессе файлах и пр., а также сохраненные значения аппаратной составляющей. Итак, в системной составляющей контекста процесса содержатся различные

- идентификатор родительского процесса
- текущее состояние процесса
- приоритет процесса
- реальный и эффективный идентификаторы пользователя-владельца
- реальный и эффективный идентификатор идентификатор группы, к которой принадлежит владелец
- список областей памяти
- таблица открытых файлов процесса
- информация об установленной реакции на тот или иной сигнал
- информация о сигналах, ожидающих доставки в данный процесс
- сохраненные значения аппаратной составляющей

атрибуты процесса, такие как: •идентификатор родительского процесса; •текущее состояние процесса; •приоритет процесса и тд

Следуя второй трактовке, процессом называется объект, порожденный **системным вызовом fork()**. Этот системный вызов обеспечивает создание копии текущего процесса. Под **системным вызовом** понимается средство ОС, предоставляемое пользователям (а точнее, процессам), посредством которого процессы могут обращаться к ядру операционной системы за выполнением тех или иных функций. При этом выполнение системных вызовов происходит в привилегированном режиме (поскольку непосредственную обработку системных вызовов производит ядро), даже если сам процесс выполняется в пользовательском режиме.

## 23. Реализация процессов в ОС UNIX. Базовые средства управления процессами в ОС UNIX. Загрузка ОС UNIX, формирование нулевого и первого процессов.

Для порождения новых процессов в UNIX существует единая схема, с помощью которой создаются все процессы, существующие в работающем экземпляре ОС UNIX, за исключением первых двух процессов (0-го и 1-го). Для создания нового процесса в операционной системе UNIX используется системный вызов **fork()**. При

### Создание нового процесса

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork ( void );
```

- При удачном завершении возвращается:
  - сыновнему процессу значение 0
  - родительскому процессу PID порожденного процесса
- При неудачном завершении возвращается -1, код ошибки устанавливается в переменной **errno**
- Заносится новая запись в таблицу процессов
- Новый процесс получает уникальный идентификатор
- Создание контекста для нового (сыновьего) процесса

обращении процесса к данному системному вызову, операционная система создает копию текущего процесса, т.е. появляется еще один процесс, тело которого полностью идентично исходному процессу. Это означает, что система заносит в таблицу процессов новую запись, тем самым новый порождённый процесс получает уникальный идентификатор.

Сыновний процесс наследует от родительского процесса большую часть контекста, а именно:

- **окружение** — при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением

- файлы, открытые в процессе-отце, за исключением тех, которым было запрещено передаваться процессам-потомкам с помощью задания специального параметра при открытии.
- способы обработки сигналов;
- разрешение переустановки эффективного идентификатора пользователя;
- разделяемые ресурсы процесса-отца; • и т.д.

## Семейство системных вызовов exec()

```
#include <unistd.h>
int execl (const char *path, char *arg0, ..., char *argn, 0);
```

- **path** — имя файла, содержащего исполняемый код программы
- **arg0** — имя файла, содержащего вызываемую на выполнение программу
- **arg1, ..., argn** — аргументы программы, передаваемые ей при вызове

Возвращается:  
в случае ошибки

Обычно к нему происходит обращение в связи с одним из семейства системных вызовов exec(). Последние обеспечивают смену тела текущего процесса. Прежде всего, речь пойдет о **завершении процесса\_exit()**. Процесс-предок имеет возможность получить информации о статусе завершения/приостановки своего потомка. Для этого служит системный вызов **wait()**.

## Получение информации о завершении своего потомка

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait ( int *status );
```

**status** по завершению содержит:

- в старшем байте — код завершения процесса-потомка (**пользовательский код завершения процесса**)
- в младшем байте — индикатор причины завершения процесса-потомка, устанавливаемый ядром UNIX (**системный код завершения процесса**)

Возвращается: PID завершенного процесса или –1 в случае ошибки или прерывания

## Формирование процесса 1 и 0

Нажали на включение.

**Начальная загрузка** — загрузка ядра системы в оперативную память, запуск ядра.

1. Чтение нулевого блока системного устройства аппаратным загрузчиком (аппаратный загрузчик это BIOS). Определяет активное системное устройство, откуда будет подгружаться ОС.

2. В этом нулевом блоке поиск и считывание в память файла */unix*
3. Запуск на исполнение файла */unix*

В самом начале ядром выполняются определенные действия по инициализации системы, а именно:

- устанавливаются системные часы (для генерации прерываний)
- формируется диспетчер памяти
- формируются значения некоторых структур данных (наборы буферов блоков, буфера индексных дескрипторов) и ряд других
- По окончании этих действий происходит инициализация процесса с номером 0. Это ядро (это делается не штатным образом)

По понятным причинам для этого невозможно использовать методы порождения процессов, изложенные выше, т. е. с использованием функций **fork()** и **exec()**. При инициализации этого процесса резервируется память под его контекст и формируется нулевая запись в таблице процессов.

Основными отличиями нулевого процесса являются следующие моменты:

1. Данный процесс не имеет кодового сегмента — это просто структура данных, используемая ядром и процессом его называют потому, что он каталогизирован в таблице процессов.

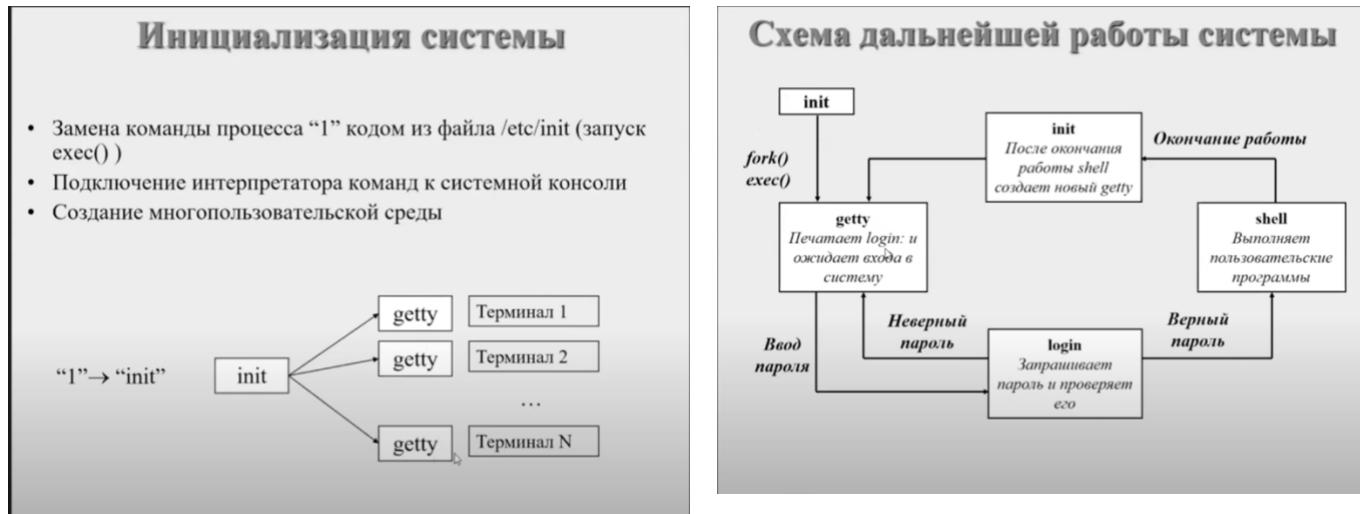
2. Он существует в течении всего времени работы системы (чисто системный процесс) и считается, что он активен, когда работает ядро ОС.

Создание ядром первого процесса

- Копируется процесс “0” (запись таблицы процессов)
- Создание области кода процесса “1”
- Копирование в область кода процесса “1” программы, реализующей системный вызов exec(), который необходим для выполнения программы /etc/init.

Смена тела на программу init. Далее он для каждого терминала запускает getty. Кол-во терминалов зависит от кол-ва пользователей.

Терминал для каждого пользователя. Окончание работы ctrl+D. Shell дает нам интерпретатор команд. И мы можем программировать свои проги



## 24. Реализация нитей в UNIX. Примеры программирования нитей

написать

## 25. Взаимодействие процессов. Разделяемые ресурсы. Критические секции. Взаимное исключение. Тупики.

**Параллельные процессы** — процессы, выполнение (обработка) которых хотя бы частично перекрывается по времени.

- **Независимые процессы** используют независимое множество ресурсов
- **Взаимодействующие процессы** используют ресурсы совместно, и выполнение одного процесса может оказать влияние на результат другого

**Разделение ресурса** — совместное использование несколькими процессами ресурса ВС.

**Критические ресурсы** — разделяемые ресурсы, которые должны быть доступны в текущий момент времени только одному процессу.

**Критическая секция или критический интервал** часть программы (фактически набор операций), в которой осуществляется работа с критическим ресурсом.

Выделяют две важнейшие задачи ОС: распределение ресурсов между процессами и организация корректного доступа (т.е. организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов). Результат выполнения процессов не должен зависеть от порядка переключения выполнения между процессами.

Ситуация, когда процессы конкурируют за разделяемый ресурс, называются **гонкой процессов**

Единственный способ избежать гонок при использовании разделяемых ресурсов — контролировать доступ к любым разделяемым ресурсам в системе. При этом необходимо организовать **взаимное исключение** — т.е. такой способ работы с разделяемым ресурсом, при котором постулируется, что в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

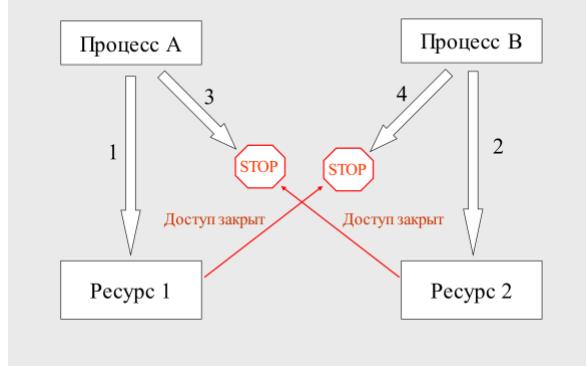
Проблему организации взаимного исключения можно сформулировать в более общем виде: задача взаимного исключения сводится к тому, чтобы не допускать ситуации, когда два процесса одновременно находятся в критических секциях, связанных с одним и тем же ресурсом.

Проблемы, которые могут возникать при организации взаимного исключения, — это **блокировки и тупики**.

**Блокировка** — это ситуация, когда доступ к разделяемому ресурсу одного из взаимодействующих процессов не обеспечивается из-за активности других, более приоритетных процессов.

**Тупик, или deadlock**, — это ситуация, когда (из-за некорректной организации доступа и разделения ресурсов) конкурирующие за критический ресурс процессы вступают в клинч — происходит взаимоблокировка.

### Тупики (Deadlocks)



## 26. Взаимодействие процессов. Некоторые способы реализации взаимного исключения: семафоры Дейкстры, мониторы, обмен сообщениями.

В настоящий момент известно множество средств организации взаимного исключения, среди которых мы рассмотрим **семафоры Дейкстры, мониторы Хоара и аппарат передачи сообщений**.

**Семафоры Дейкстры.** Эта модель базируется на следующей концепции. Имеется специальный тип данных — т.н. семафор. Переменные типа семафор могут принимать целочисленные значения. Над этими переменными определены следующие атомарные (их нельзя прервать) (неделимые) операции: опустить семафор `down(S)` (или `P(S)`) и поднять семафор `up(S)` (или `V(S)`). Пример атомарной команды – машинная команда

Операция `down(S)` проверяет значение семафора `S` и, если оно больше нуля, то уменьшает его на 1. Если же это не так, процесс блокируется, причем связанная с заблокированным процессом операция `down` считается незавершенной. Операция `up(S)` увеличивает значение семафора на 1. При этом если в системе присутствуют процессы, заблокированные ранее при выполнении `down` на этом семафоре, то один из них разблокируется и завершает выполнение операции `down`, т.е. вновь уменьшает значение семафора. (Ещё раз отметим, что операции `up` и `down` являются атомарными (неделимыми), т.е. их выполнение не может быть прервано прерыванием.)

Пример **двоичного семафора** — семафора, максимальное значение которого равно 1. Этот тип семафоров и обеспечивает взаимное исключение. Таким образом, **семафоры Дейкстры** — это низкоуровневые средства синхронизации, для корректной практической реализации которых необходимо наличие специальных атомарных семафорных машинных команд.

**Монитор Хоара** — совокупность процедур и структур данных, объединенных в программный модуль специального типа. (аналог с телефонной кабинкой). Он антипод. И он высокоуровневый в отличии от семафоров.

Процесс захватывает монитор и в этом мониторе может работать только один процесс. Монитор захватывает при обращении в любую его процедуру

- Структуры данных монитора доступны только для процедур, входящих в этот монитор
- Процесс «входит» в монитор по вызову одной из его процедур
- В любой момент времени внутри монитора может находиться не более одного процесса

Сразу отметим, что механизм передачи сообщений является универсальным в том смысле, что он предоставляет, как средства организации взаимодействия между процессами, так и средства синхронизации. **Механизм передачи сообщений** основан на двух функциональных примитивах: `send` (отправить сообщение) и `receive` (принять сообщение). Данные операции можно разделить по трем характеристикам: модель синхронизации, адресация и формат сообщения.

**Синхронизация.** Операции посылки/приема сообщений могут быть блокирующими и неблокирующими. Рассмотрим различные комбинации. **Блокирующий send:** процесс-отправитель будет заблокирован до тех пор, пока посланное им сообщение не будет получено. **Блокирующий receive:** процесс-получатель будет заблокирован до тех пор, пока не будет получено соответствующее сообщение. Соответственно, **неблокирующие операции**, как следует из названия, происходят без блокировок.

**Адресация** может быть **прямой** или **косвенной**. При **прямой** адресации указывается конкретный адрес получателя и/или отправителя (например, когда получатель ожидает сообщения от конкретного отправителя, игнорируя сообщения других отправителей). В случае **косвенной** адресации не указывается адрес конкретного получателя при отправке или адрес конкретного отправителя при получении; сообщение «бросается» в некоторый общий пул, в котором могут быть реализованы различные стратегии доступа.

## 27. Взаимодействие процессов. Классические задачи синхронизации процессов. “Обедающие философы”

**Обедающие философы.** Пусть существует круглый стол, за которым сидит группа философов: они пришли пообщаться и покушать. Кушают они спагетти, которое находится в общей миске, стоящей в центре стола. Для приема пищи они пользуются двумя вилками: одна в левой руке, другая — в правой. Вилки располагаются по одной между каждыми двумя философами. Каждый из философов некоторое время размышляет, затем берет две вилки и ест спагетти, затем кладёт вилки на стол и опять размышляет, и так далее. Каждый из них ведет себя независимо от других. Философы должны совместно использовать имеющиеся у них вилки (ресурсы). Задача состоит в организации доступа к вилкам.

```
# define N 5
# define LEFT (i-1)%N      /* номер левого соседа для i-ого философа */
# define RIGHT (i+1)%N     /* номер правого соседа для i-ого философа*/
# define THINKING 0         /* философ думает */
# define HUNGRY 1            /* философ голоден */
# define EATING 2             /* философ ест */
typedef int semaphore;    /* тип данных «семафор» */
int state[N];             /* массив состояний философов */
semaphore mutex=1;        /* семафор для критической секции */
semaphore s[N];           /* по одному семафору на философа */
void philosopher (int i)
    /* i : номер философа от 0 до N-1 */
{
while (TRUE)           /* бесконечный цикл */
{
    think();           /* философ думает */
    take_forks(i);    /* философ берет обе вилки или блокируется */
    eat();              /* философ ест */
    put_forks(i);     /* философ освобождает обе вилки */
}
}

void take_forks(int i)
    /* i : номер философа от 0 до N-1 */
{
down(&mutex);          /* вход в критическую секцию */
state[i] = HUNGRY;      /*записываем, что i-ый философ голоден */
test(i);                /* попытка взять обе вилки */
up(&mutex);            /* выход из критической секции */
down(&s[i]);           /* блокируемся, если вилок нет */
}

void put_forks(i)
    /* i : номер философа от 0 до N-1 */
{
down(&mutex);          /* вход в критическую секцию */
state[i] = THINKING;   /* философ закончил есть */
    test(LEFT);
/* проверить может ли левый сосед сейчас есть */
    test(RIGHT);
/* проверить может ли правый сосед сейчас есть*/
up(&mutex);            /* выход из критической секции */
}

void test(i)
    /* i : номер философа от 0 до N-1 */
{
if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
{
    state[i] = EATING;
    up (&s[i]);
}
}
```

## 28. Взаимодействие процессов. Классические задачи синхронизации процессов. “Читатели и писатели”

Представим произвольную систему резервирования ресурса. Например, это может быть система резервирования места в гостинице. В данной системе существует два типа процессов для работы с информацией. Одни процессы могут читать информацию, а другие — ее изменять, корректировать. Соответственно, возникает все тот же вопрос, как организовать корректную совместную работу этих процессов. Это означает, что в любой момент времени читать данные могут любое количество процессов-читателей, но если процесс-писатель начал свою работу, то все остальные процессы (и читатели, и писатели) будут блокированы на входе в систему. Задача заключается в планировании работы такой системы.

Рассмотрим модельную реализацию данной задачи при выбранной следующей стратегии: будем считать, что наиболее приоритетными являются читающие процессы. То есть процесс-писатель будет ожидать момента, когда все желающие процессы-читатели окончат свои действия в системе и покинут ее.

```
typedef int semaphore;      /* тип данных «семафор» */
semaphore mutex = 1;        /* контроль за доступом к «rc» (разделяемый
                           ресурс) */
semaphore db = 1;           /* контроль за доступом к базе данных */
int rc = 0;                 /* кол-во процессов читающих или пишущих */

void reader (void)
{
    while (TRUE)           /* бесконечный цикл */
    {
        down (&mutex);     /* получить эксклюзивный доступ к «rc»*/
        rc = rc + 1;         /* еще одним читателем больше */
        if (rc == 1) down (&db); /* если это первый читатель, нужно
                               заблокировать эксклюзивный доступ к
                               базе */
        up (&mutex);        /* освободить ресурс rc */
        read_data_base();   /* доступ к данным */
        down (&mutex);     /* получить эксклюзивный доступ к «rc»*/
        rc = rc - 1;         /* теперь одним читателем меньше */
        if (rc == 0) up (&db); /* если это был последний читатель,
                               разблокировать эксклюзивный доступ к
                               базе данных */
        up (&mutex);        /* освободить разделяемый ресурс rc */
        use_data_read();    /* некритическая секция */
    }
}

void writer (void)
{
    while (TRUE)           /* бесконечный цикл */
    {
        think_up_data();   /* некритическая секция */
        down (&db);        /* получить эксклюзивный доступ к данным*/
        write_data_base();  /* записать данные */
        up (&db);          /* отдать эксклюзивный доступ */
    }
}
```

## 29. Взаимодействие процессов. Классические задачи синхронизации процессов. «Спящий парикмахер».

Рассмотрим парикмахерскую, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер засыпает прямо на своем рабочем месте. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания (т.е. это стратегия обслуживания с отказами). Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

```
#define CHAIRS 5
typedef int semaphore; /* тип данных «семафор» */
semaphore customers = 0;      /* посетители, ожидающие в очереди */
semaphore barbers = 0;        /* парикмахеры, ожидающие посетителей */
semaphore mutex = 1;          /* контроль за доступом к переменной
                               waiting */

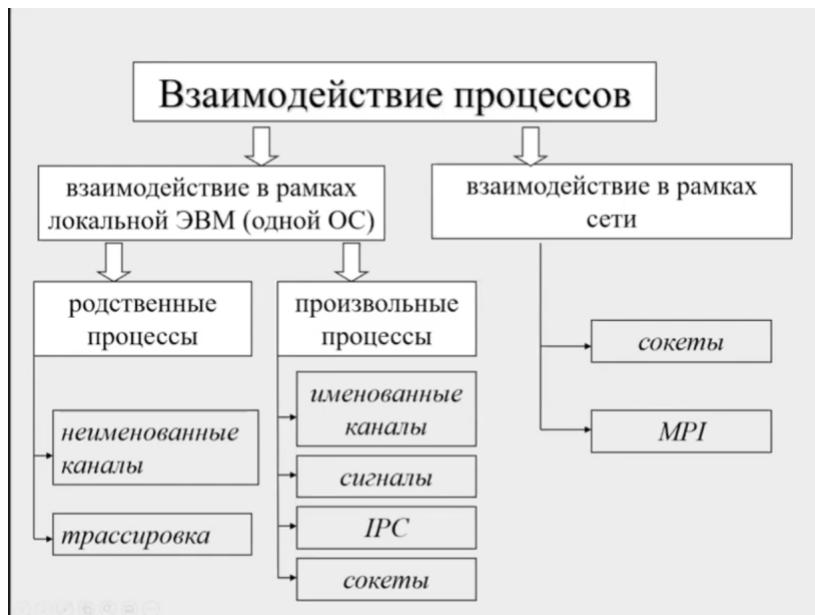
int waiting = 0;
void barber()
{
while (true) {
    down(customers); /* если customers == 0, т.е.
                       посетителей нет, то заблокируемся до появления
                       посетителя */
    down(mutex);       /* получаем доступ к waiting */
    waiting = waiting - 1; /* уменьшаем кол-во ожидающих
                           клиентов */
    up(barbers);      /* парикмахер готов к работе */
    up(mutex);         /* освобождаем ресурс waiting */
    cut_hair();        /* процесс стрижки */
}
void customer()
{
    down(mutex); /* получаем доступ к waiting */
    if (waiting < CHAIRS) /* есть место для ожидания */
    {
        waiting = waiting + 1; /* увеличиваем кол-во
                                 ожидающих клиентов */
        up(customers); /* если парикмахер спит, это
                         его разбудит */
        up(mutex);      /* освобождаем ресурс waiting */
        down(barbers); /* если парикмахер занят, переходим
                         в состояние ожидания, иначе – занимаем
                         парикмахера*/
        get_haircut();   /* процесс стрижки */
    }
    else
    {
        up(mutex); /* нет свободного кресла для ожидания –
                     придется уйти */
    }
}
```

## 30. Базовые средства взаимодействия процессов в ОС UNIX. Сигналы.

### Примеры программирования.

**Сигнал** – средство асинхронного уведомления процесса о наступлении некоторого события в системе.

Симметрично это когда у процессов равные возможности. Ассиметричные это когда разные наборы возможности у процессов.



В ОС Unix присутствует т.н. аппарат сигналов, позволяющий одним процессам оказывать воздействия на другие процессы. Сигналы могут рассматриваться как средство уведомления процесса о наступлении некоторого события в системе. Инициатором отправки сигнала процессу может быть как процесс, так и сама ОС. Сигналы, посылаемые ОС, уведомляют о наступлении некоторых строго предопределенных ситуаций (как, например, завершение порожденного процесса, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал и т.п.), при этом каждой такой ситуации сопоставлен свой сигнал.

Аппарат сигналов является механизмом асинхронного взаимодействия, момент прихода сигнала процессу заранее неизвестен. Однако процесс может предвидеть возможность получения того или иного сигнала и установить определенную реакцию на его приход.

2 - SIGINT /\*прерывание\*/

3 - SIGQUIT /\*аварийный выход\*/

9 - SIGKILL /\*уничтожение процесса\*/

14 - SIGALRM /\*прерывание от таймера\*/

18 – SIGCHLD /\* процесс-потомок завершился \*/

При получении процессом сигнала возможны три типа реакции на него . Во-первых, это **обработка сигнала по умолчанию**. Во-вторых, процесс может **перехватывать обработку пришедшего сигнала**. Если процесс получает сигнал, то вызывается функция, принадлежащая телу процесса, которая была специальным образом зарегистрирована в системе как обработчик сигнала. В-третьих, сигналы можно **игнорировать**, т.е. приход некоторых сигналов процесс может проигнорировать.

### Работа с сигналами

```
#include <sys/types.h>
#include <signal.h>

int kill (pit_t pid, int sig);

pid – идентификатор процесса, которому посыпается сигнал
sig – номер посыпаемого сигнала
```

При удачном выполнении возвращает 0, в противном случае возвращает -1

### Пример. Программа “будильник”.

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void alarm (int s)
{
    printf("\n жду имя \n");
    alarm(5);
}
```

```
int main(int argc, char **argv)
{ char s[80];
    signal(SIGALRM, alarm); alarm(5);
    printf("Введите имя \n");
    for (;;) {
        printf("имя:");
        if (gets(s) != NULL) break;
    };
    printf("OK! \n");
    return 0;
}
```

### Работа с сигналами

```
#include <signal.h>
void (*signal ( int sig, void (*disp)(int)) (int)
```

**sig** – номер сигнала, для которого устанавливается реакция  
**disp** – либо определенная пользователем функция – обработчик сигнала, либо одна из констант:  
**SIG\_DFL** – обработка по умолчанию  
**SIG\_IGN** - игнорирование

При успешном завершении функция возвращает указатель на предыдущий обработчик данного сигнала.

Signal возвращает указатель на функцию с одним параметром int

## 31. Базовые средства взаимодействия процессов в ОС UNIX. Неименованные каналы. Примеры программирования

**Неименованный канал** (или программный канал) представляется в виде области памяти (на внешнем запоминающем устройстве), управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы, т.е. эта область памяти является разделяемым ресурсом. Это ресурс системы. А файловый дескриптор это ресурс процесса

Для доступа к неименованному каналу система ассоциирует с ним два файловых дескриптора. Один из них предназначен для чтения информации из канала, т.е. с ним можно ассоциировать файл, открытый только на чтение. Другой дескриптор предназначен для записи информации в канал. Соответственно, с ним может быть ассоциирован файл, открытый только на запись.

Дисциплина доступа к информации, записанной в программный канал, - FIFO, т.е. информация, первой записанная в канал, будет прочитана из канала первой.

Для создания неименованного канала служит системный вызов pipe().

```
#include <unistd.h>
```

```
int pipe(int *fd)
```

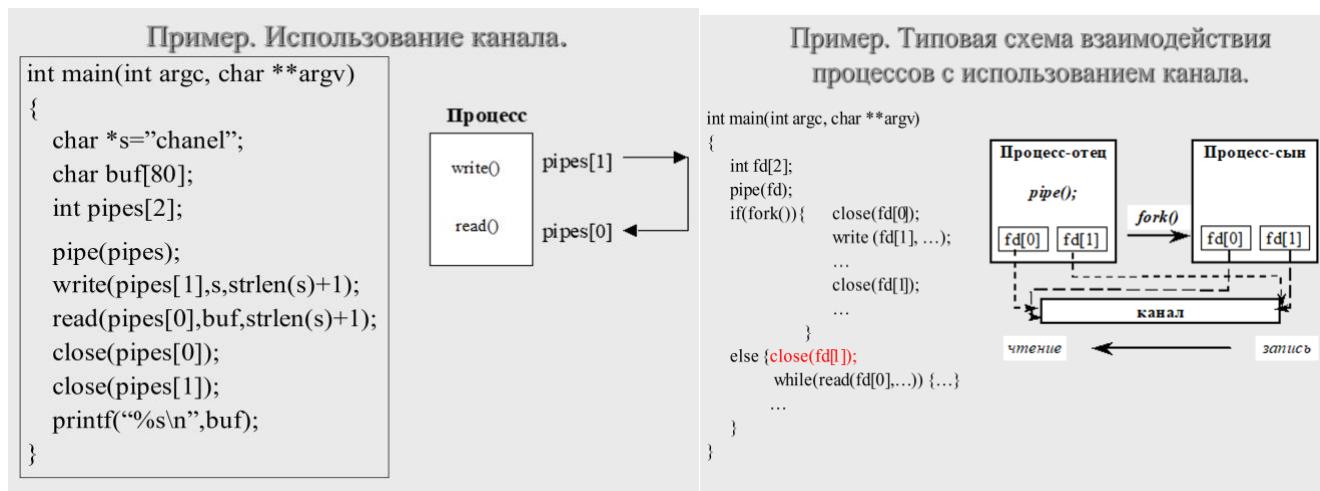
Аргументом данного системного вызова является указатель на массив fd из двух целочисленных элементов.

Если системный вызов `pipe()` прорабатывает успешно, то он возвращает код ответа, равный нулю, а массив будет содержать два открытых файловых дескриптора. Соответственно, в `fd[0]` будет содержаться дескриптор чтения из канала, а в `fd[1]` — дескриптор записи в канал.

`pipe` может не сработать если превышен лимит дескрипторов.

Однако следует четко понимать различия между обычным файлом и каналом. Основные отличительные свойства канала следующие: - В отличие от файла, к неименованному каналу невозможен доступ по имени, - В отличие от файлов неименованный канал существует в системе, до тех пор, пока существуют процессы, его использующие. - Канал – это структура данных, которая реализует модель последовательного доступа к данным (FIFO), т.е. данные из канала можно прочитать только в той же последовательности, в какой они были записаны в канал.

Неименованные каналы в общем случае предназначены для синхронизации и организации взаимодействия родственных процессов, осуществляющегося за счет передачи по наследству ассоциированных с каналом файловых дескрипторов.



В приведенном ниже примере производится копирование текстовой строки с использованием канала. Схема организации взаимодействия процессов с использованием канала

Именованные каналы (FIFO-файлы) расширяют свою область применения за счет того, что подключиться к ним может любой процесс в любое время, в том числе и после создания канала. Это возможно благодаря наличию у них имен.

## 32. Базовые средства взаимодействия процессов в ОС UNIX. Взаимодействие процессов по схеме "подчиненный-главный". Общая схема трассировки процессов.

Достаточно часто при организации многопроцессной работы необходимо наличие возможности, когда один процесс является главным по отношению к другим процессам (трассировка процессов). В частности, это необходимо для организации средств отладки, когда есть процесс-отладчик и отлаживаемый процесс. Для механизма отладки полезно, чтобы отладчик мог в произвольные моменты времени останавливать отлаживаемый процесс и, когда отлаживаемый процесс остановлен, осуществлять действия по его отладке: просматривать содержимое тела процесса, при необходимости корректировать тело процесса и т.д. Также является полезным возможность установки контрольных точек в отлаживаемом процессе. Очевидно, что полномочия процесса-отладчика по отношению к отлаживаемому процессу являются полномочиями главного, т.е. отладчик может осуществлять управление, в то время как отлаживаемый процесс может лишь подчиняться. В UNIX трассировка возможна только между родственными процессами: процесс-родитель может вести трассировку только непосредственно порожденных им потомков, при этом трассировка начинается только после того, как процесс-потомок дает разрешение на это.

Далее схема взаимодействия процессов путем трассировки такова: выполнение отлаживаемого процесса-потомка приостанавливается всякий раз при получении им какого-либо сигнала, а также при выполнении вызова exec(). Если в это время отлаживающий процесс осуществляет системный вызов wait(), этот вызов немедленно возвращает управление. В то время, как трассируемый процесс находится в приостановленном состоянии, процесс-отладчик имеет возможность анализировать и изменять данные в адресном пространстве отлаживаемого процесса и в пользовательской составляющей его контекста. Далее, процесс-отладчик возобновляет выполнение трассируемого процесса до следующей приостановки.

Для организации взаимодействия «главный–подчиненный» ОС Unix предоставляет системный вызов ptrace().  
**#include <sys/ptrace.h>** cmd обозначает код выполняемой команды, pid — идентификатор процесса-потомка  
**int ptrace(int cmd, int pid, int addr, int data);** addr — некоторый адрес в адресном пространстве процесса-потомка, data — слово информации.

Посредством системного вызова ptrace() можно решать две задачи. С помощью этого вызова подчиненный процесс может разрешить родительскому процессу проводить свою трассировку: для этого в качестве параметра cmd необходимо указать команду PTRACE\_TRACEME. С другой стороны, с помощью этого же системного вызова процесс отладчик может манипулировать отлаживаемым процессом.

**Системный вызов ptrace() позволяет выполнять следующие действия:**

1. читать данные из сегмента кода и сегмента данных отлаживаемого процесса (только когда процесс остановлен);

2. читать некоторые данные из контекста отлаживаемого процесса;

3. осуществлять запись в сегмент кода, сегмент данных и в некоторые области контекста отлаживаемого процесса;

4. продолжать выполнение отлаживаемого процесса с прерванной точки или с предопределенного адреса сегмента кода; (могу поменять регистры общего доступа) (тоже при приостановке)

5. исполнять отлаживаемый процесс в пошаговом режиме (процесс приостанавливается после каждой машинной команды – получает сигнал),

продолжить выполнение с точки остановки или с заданного адреса, завершить процесс

**Пошаговый режим** — это режим, обеспечиваемый аппаратурой компьютера, который вызывает прерывание после исполнения каждой машинной команды отлаживаемого процесса.

Отладчики бывают двух типов: **адресно-кодовыми и символьными**. Адреснокодовые отладчики оперируют адресами тела отлаживаемого процесса, в то время как символьные отладчики позволяют оперировать объектами языка, т.е. переменными и операторами языка.



### 33. Система межпроцессного взаимодействия ОС UNIX. Именование разделяемых объектов. Очереди сообщений. Пример.

Система IPC (известная так же, как IPC System V) позволяет обеспечивать взаимодействие произвольных процессов в пределах локальной машины. Для этого она предоставляет взаимодействующим процессам возможность использования общих, или разделяемых, ресурсов. Эти ресурсы могут существовать в системе с момента создания до момента их принудительного удаления либо в течение сеанса работы операционной системы (вне зависимости от наличия процессов, которые их используют). Такие разделяемые ресурсы можно подразделить на три типа:

– **Очередь сообщений** — это разделяемый ресурс, позволяющий организовывать очереди сообщений: один процесс может в эту очередь положить сообщение, а другой процесс — прочитать его. Данный механизм имеет возможность блокировок, поэтому его можно использовать и как средство передачи информации между взаимодействующими процессами, и как средство их синхронизации.

Для организации совместного использования разделяемых ресурсов необходим некоторый **механизм именования ресурсов**, используя который взаимодействующие процессы смогут работать с данным конкретным ресурсом. Для этих целей используется **система ключей**

#### Организация очереди сообщений по принципу FIFO и Использование типов сообщений

##### Создание/доступ к очереди сообщений

###### Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
int msgget ( key_t key, int msgflag )
```

###### Параметры

**key** — ключ

**msgflag** — флаги, управляющие поведением вызова

###### Возвращаемое значение

В случае успеха возвращается положительный дескриптор очереди, в случае неудачи возвращается –1.

У данной функции два параметра: ключ и флаги. В случае успешного выполнения функция возвращает положительный дескриптор очереди, иначе возвращается -1. Рассмотрим теперь функции отправки и приема сообщений. Для отправки сообщений служит функция msgsnd().

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,
int msgflg);
```

Для получения сообщений используется функция msgrcv().

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long
msgtyp, int msgflg);
```

Следующая группа функций — это функции управления ресурсом — очередью сообщений. Эти функции обеспечивают в общем случае изменение режима функционирования ресурса, в т.ч. и удаление ресурса

```
int msgctl(int msqid, int cmd, struct msgid_ds *buf);
```

##### Пример: «Клиент-сервер»

###### Сервер

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msgh>
#include<stdlib.h>
#include<string.h>
int main ( int argc, char ** argv )
{ struct {
    long mestype ;
    char mes [ 100 ] ;
} messageto ;
struct {
    long mestype ;
    long mes ;
} messagefrom ;
key_t key ;
int mesid ;
```

```
key = ftok ( "example", 'r' ) ;
mesid = msgget ( key, 0666 | IPC_CREAT |
IPC_EXCL ) ;
while(1)
{
    msgrcv ( mesid, & messagefrom, sizeof(
        messagefrom )-sizeof( long ), 1, 0 ) ;
    messageto . mestype = messagefrom . mes ;
    strcpy ( messageto . mes, "Message for client" ) ;
    msgsnd ( mesid, & messageto, sizeof(messageto )
        - sizeof( long ), 0 ) ;
}
return 0 ;
```

##### Пример: «Клиент-сервер»

###### Клиент

```
long pid = getpid () ;
key = ftok ( "example", 'r' ) ;
mesid = msgget ( key, 0666 ) ;
messageto . mestype = 1 ;
messageto . mes = pid ;
msgsnd ( mesid, & messageto, sizeof(
(messageto )- sizeof ( long ), 0 ) ;
msgrcv ( mesid, & messagefrom, sizeof(
messagefrom )- sizeof ( long ), pid, 0 ) ;
printf ( "%s", messagefrom . mes ) ;
return 0 ;
```

```
key_t key;
int mesid;
```

## 34. Система межпроцессного взаимодействия ОС UNIX . Именование разделяемых объектов. Разделяемая память. Пример

Система IPC (известная так же, как IPC System V) позволяет обеспечивать взаимодействие произвольных процессов в пределах локальной машины. Для этого она предоставляет взаимодействующим процессам возможность использования общих, или разделяемых, ресурсов. Эти ресурсы могут существовать в системе с момента создания до момента их принудительного удаления либо в течение сеанса работы операционной системы (вне зависимости от наличия процессов, которые их используют). Такие разделяемые ресурсы можно подразделить на три типа:

— **Общая, или разделяемая, память**, которая представляется процессу как указатель на область памяти, которая является общей для двух и более процессов. Т.е. внутри процесса некоторый указатель можно установить на начало данной области и работать далее с этой областью, как с массивом. Все изменения, которые сделает данный процесс, будут видны другим процессам. Разделяемая память IPC почти не обладает никакими средствами синхронизации (т.е. существует очень слабо развитый механизм взаимных блокировок, но мы на нем не будем останавливаться).

Для организации совместного использования разделяемых ресурсов необходим некоторый **механизм именования ресурсов**, используя который взаимодействующие процессы смогут работать с данным конкретным ресурсом. Для этих целей используется **система ключей**.

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти (Рис. 100). Благодаря этому, данные, находящиеся в этой области памяти, будут доступны для чтения и модификации всем процессам, подключившимся к данной области памяти. Процесс, подключившийся к разделяемой памяти, может затем получить указатель на некоторый адрес в своем виртуальном адресном пространстве, соответствующий данной области разделяемой памяти. После этого он может работать с этой областью памяти аналогично тому, как если бы она была выделена динамически (например, путем обращения к malloc()), однако, как уже говорилось, разделяемая область памяти не уничтожается автоматически даже после того, как процесс, создавший или использовавший ее, перестанет с ней работать.

### IPC: разделяемая память

Создание общей памяти

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget ( key_t key, int size, int shmflg )
```

#### Параметры

**key** — ключ для доступа к разделяемой памяти  
**size** — размер области памяти  
**shmflg** — флаги управляющие поведением вызова

#### Возвращаемое значение

дескриптор области памяти, в случае неудачи — -1.

Доступ к разделяемой памяти **char \*shmat(int shmid, char \*shmaddr, int shmflg);**

Соответственно, для открепления разделяемой памяти от адресного пространства процесса используется функция **shmdt() int shmdt(char \*shmaddr);**

И, напоследок, рассмотрим функцию **shmctl()** управления ресурсом разделяемая память.

**int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);**

### Пример. Работа с общей памятью в рамках одного процессса

```
int main ( int argc, char ** argv )
{
    key_t key;
    char * shmaddr ;

    key = ftok ( "/tmp/ter", 'S' );
    shmid = shmget ( key, 100, 0666 | IPC_CREAT | IPC_EXCL );
    shmaddr = shmat ( shmid, NULL, 0 );
    putm ( shmaddr );
    .....
    shmctl ( shmid, IPC_RMID, NULL );
    exit ();
}
```

## 35. Система межпроцессного взаимодействия ОС UNIX . Именование разделяемых объектов. Массив семафоров. Пример.

Система IPC (известная так же, как IPC System V) позволяет обеспечивать взаимодействие произвольных процессов в пределах локальной машины. Для этого она предоставляет взаимодействующим процессам возможность использования общих, или разделяемых, ресурсов. Эти ресурсы могут существовать в системе с момента создания до момента их принудительного удаления либо в течение сеанса работы операционной системы (вне зависимости от наличия процессов, которые их используют). Такие разделяемые ресурсы можно подразделить на три типа:

— **Массив семафоров** — ресурс, представляющий собой массив из N элементов, где N задается при создании данного ресурса, и каждый из элементов является семафором IPC (а не семафором Дейкстры: семафор Дейкстры так или иначе является формализмом, не опирающимся ни на какую реализацию, а семафор IPC — конкретной программной реализацией семафора в ОС). Семафоры IPC предназначены, в первую очередь, для использования в качестве средств организации синхронизации.

Семафоры представляют собой одну из форм IPC и обычно используются для синхронизации доступа нескольких процессов к разделяемым ресурсам, так как сами по себе другие средства IPC не предоставляют механизма синхронизации.

Как уже говорилось, семафор представляет собой особый вид числовой переменной, над которой определены две неделимые операции: уменьшение ее значения с возможным блокированием процесса и увеличение значения с возможным разблокированием одного из ранее заблокированных процессов. Объект System V IPC представляет собой набор семафоров. Как правило, использование семафоров в качестве средства синхронизации доступа к другим разделяемым объектам предполагает следующую схему:

- с каждым разделяемым ресурсом связывается один семафор из набора;
- положительное значение семафора означает возможность доступа к ресурсу (ресурс свободен), неположительное – отказ в доступе (ресурс занят);
- перед тем как обратиться к ресурсу, процесс уменьшает значение соответствующего ему семафора, при этом, если значение семафора после уменьшения должно оказаться отрицательным, то процесс будет заблокирован до тех пор, пока семафор не примет такое значение, чтобы при уменьшении его значение оставалось неотрицательным;
- закончив работу с ресурсом, процесс увеличивает значение семафора (при этом разблокируется один из ранее заблокированных процессов, ожидающих увеличения значения семафора, если такие имеются);
- в случае реализации взаимного исключения используется двоичный семафор, т.е. такой, что он может принимать только значения 0 и 1: такой семафор всегда разрешает доступ к ресурсу не более чем одному процессу одновременно

Для получения доступа к массиву семафоров (или его создания) используется системный вызов `semget()`:

`int semget (key_t key, int nsems, int semflag);`

Используя полученный дескриптор, можно производить изменять значения одного или нескольких семафоров в наборе, а также проверять их значения на равенство нулю, для чего используется системный вызов `semop()`: `int semop (int semid, struct sembuf *semop, size_t nops)`

С помощью этого системного вызова можно запрашивать и изменять управляющие параметры разделяемого ресурса, а также удалять его. `int semctl (int semid, int num, int cmd, union semun arg)`

В рассматриваемом примере моделируется двухпроцессная система, в которой первый процесс создает ресурсы разделяемая память и массив семафоров. Затем он начинает читать информацию со стандартного устройства ввода, считанные строки записываются в разделяемую память. Второй процесс читает строки из разделяемой памяти. Таким образом, мы имеем критический участок в момент, когда один процесс еще не дописал строку, а другой ее уже читает. Данная задача требует синхронизации, которая будет осуществляться на основе механизма семафоров.

## 36. Сокеты. Типы сокетов. Коммуникационный домен. Схема работы с сокетами с установлением соединения.

Сокеты представляют собой в определенном смысле обобщение механизма каналов, но с учетом возможных особенностей, возникающих при работе в сети. Кроме того, они предоставляют больше возможностей по передаче сообщений, например, могут поддерживать передачу экстренных сообщений вне общего потока данных. Общая схема работы с сокетами любого типа такова: каждый из взаимодействующих процессов должен на своей стороне создать и отконфигурировать сокет, после чего процессы должны осуществить соединение с использованием этой пары сокетов. По окончании взаимодействия сокеты уничтожаются.

Два основных типа коммуникационных соединений и, соответственно, сокетов представляет собой **соединение с использованием виртуального канала** и **датаграммное соединение**.

**Соединение с использованием виртуального канала** – это последовательный поток байтов, гарантирующий надежную доставку сообщений с сохранением порядка их следования. Данные начинают передаваться только после того, как виртуальный канал установлен, и канал не разрывается, пока все данные не будут переданы.

**Датаграммное соединение** используется для передачи отдельных пакетов, содержащих порции данных – датаграмм. Для датаграмм не гарантируется доставка в том же порядке, в каком они были посланы.

Поскольку сокеты могут использоваться как для локального, так и для удаленного взаимодействия, встает вопрос о пространстве адресов сокетов. При создании сокета указывается так называемый **коммуникационный домен**, к которому данный сокет будет принадлежать. Коммуникационный домен определяет конкретную модель именования (форматы адресов, правила их интерпретации), а также область взаимодействия процессов. Мы будем рассматривать два основных домена: для локального взаимодействия – домен AF\_UNIX и для взаимодействия в рамках сети – домен AF\_INET (префикс AF обозначает сокращение от «address family» – семейство адресов). В домене AF\_UNIX формат адреса – это допустимое имя файла, в домене AF\_INET адрес образуют комбинация IP адреса и номера порта (порт – виртуальная точка соединения, которая позволяет адресовать конкретный процесс извне).

Имеется возможность организации взаимодействия с **предварительным установлением соединения**. Данная модель ориентирована на организацию клиент-серверных систем, когда организуется один серверный узел процессов, который принимает сообщения от клиентских процессов и их как-то обрабатывает. Заметим, что тип сокета в данном случае не важен.



В данной модели можно выделить две группы процессов: процессы серверной части и процессы клиентской части. На стороне сервера открывается основной сокет.

Поскольку необходимо обеспечить возможность другим процессам обращаться к серверному сокету по имени, то в данном случае необходимо связывание сокета с именем (вызов bind()). Затем серверный процесс переводится в режим прослушивания (посредством системного вызова listen()): это означает, что данный процесс может принимать запросы на соединение с ним от клиентских процессов. При этом, в вызове listen() оговаривается очередь запросов на соединение, которая может формироваться к

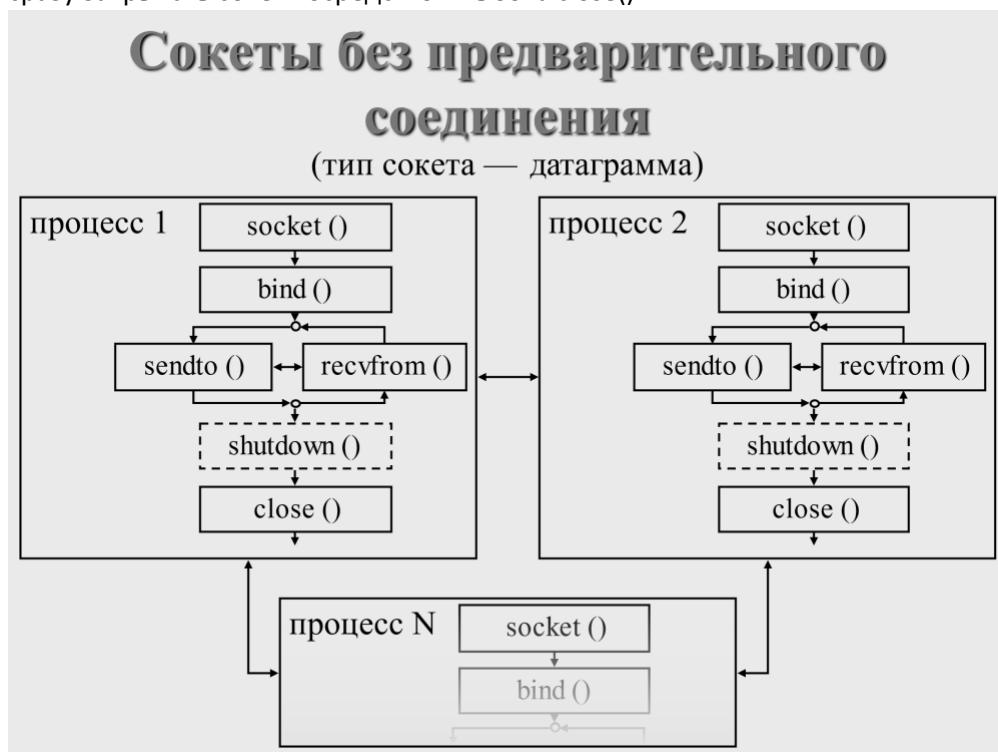
данному процессу серверной части. Каждый клиентский процесс создает свой сокет. Затем клиентский процесс может передать серверному процессу сообщение, что он с ним хочет соединиться, т.е. передать запрос на соединение (посредством системного вызова connect()). Итак, данная схема организована таким образом, что к одному серверному узлу может быть множество соединений. Для организации этой модели имеется системный вызов accept(), который работает следующим образом. Завершение работы состоит из

двух шагов. Первый шаг заключается в отключении доступа к сокету посредством системного вызова `shutdown()`. Можно закрыть сокет на чтение, на запись, на чтение-запись. Тем самым системе передается информация, что сокет более не нужен. С помощью этого вызова обеспечивается корректное прекращение работы с сокетом (в частности, это важно при организации виртуального канала, который должен гарантировать доставку посланных данных). Второй шаг заключается в закрытии сокета с помощью системного вызова `close()`

## 37. Сокеты. Схема работы с сокетами без установления соединения

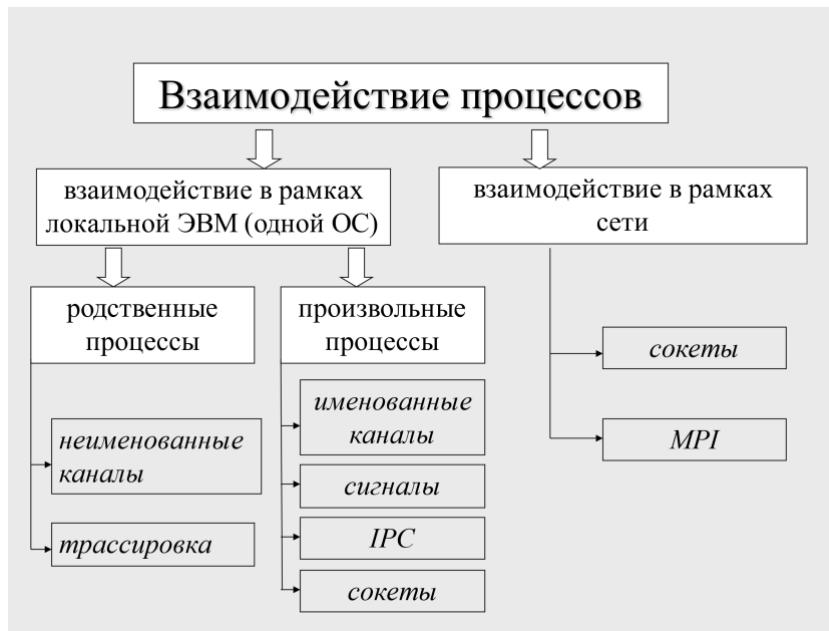
Теперь рассмотрим модель сокетов **без предварительного соединения**. В этой модели используются лишь дейтаграммные сокеты. В отличие от предыдущей модели, которая была иерархически организованной, то эта модель обладает произвольной организацией взаимодействия. Это означает, что в данной модели у каждого взаимодействующего процесса имеется сокет, через который он может получать информацию от различных источников, в отличие от предыдущей модели, где имеется «главный» известный всем клиентам сокет сервера, через который неявно передается управляющая информация (заказы на соединение), а затем с каждым клиентом связывается один локальный сокет. Этот механизм позволяет серверу взаимодействовать с клиентом, не зная его имени явно. В текущей модели ситуация симметричная: любой процесс через свой сокет может послать информацию любому другому сокету. Это означает, что механизм отправки имеет соответствующую адресную информацию (т.е. информацию об отправителе и получателе).

Поскольку в данной модели используются дейтаграммные сокеты, то необходимость обращаться к вызову `shutdown()` отпадает: в этой модели сообщения проходят (и уходят) одной порцией данных, поэтому можно сразу закрывать сокет посредством вызова `close()`.



## 38. Мьютексы

## 39. Общая классификация средств взаимодействия процессов в ОС UNIX



Прежде всего, возникает общая для обеих упомянутых групп **проблема именования взаимодействующих процессов**, которая заключается в ответе на вопрос, как, т.е. посредством каких механизмов, взаимодействующие процессы смогут «найти друг друга». В рамках взаимодействия под управлением одной ОС можно выделить две основные группы решений данной задачи.

Первая группа решений основана на **взаимодействии родственных процессов**. При взаимодействии родственных процессов, т.е. процессов, связанных некоторой иерархией родства, ОС обычно предоставляет возможность наследования сыновними процессами

некоторых характеристик родительских процессов. И именно за счет наследования различных характеристик возможно реализовать то самое именование. В данном случае именование будет **неявным**.

Следующая группа — это **взаимодействие произвольных процессов** в рамках одной локальной машины. Во-первых, **прямое именование**, когда процессы для указания своих партнеров по взаимодействию используют уникальные имена партнеров. Во-вторых, это может быть **взаимодействие посредством общего ресурса**. Но в этом случае встает проблема именования этих общих ресурсов.

**Неименованный канал** — это некоторый ресурс, наследуемый сыновьями процессами, причем этот механизм может быть использован для организации взаимодействия произвольных родственников. Другой моделью взаимодействия является несимметричная модель, которую иногда называют **модель «главный–подчиненный»**. В этом случае среди взаимодействующих процессов можно выделить процессы, имеющие больше полномочий, чем у остальных. Соответственно, у главного процесса (или процессов) есть целый спектр механизмов управления подчиненными процессами.

**Именованные каналы** — это ресурс, принадлежащий взаимодействующим процессам, посредством которого осуществляется взаимодействие. При этом не обязательно знать имена процессов-партнеров по взаимодействию.

Передача **сигналов** — это средство оказания воздействия одним процессом на другой процесс в общем случае (в частности, одним из процессов в этом виде взаимодействия может выступать процесс операционной системы). При этом используются непосредственные имена процессов.

Система **IPC** (Inter-Process Communication) предоставляет взаимодействующим процессам общие разделяемые ресурсы (среди которых ниже будут рассмотрены **общая память, массив семафоров и очередь сообщений**), посредством которых осуществляется взаимодействие процессов.

**Аппарат сокетов** — унифицированное средство организации взаимодействия. На сегодняшний момент сокеты — это не только средства ОС Unix, сколько стандартизованные средства межмашинного взаимодействия. В аппарате сокетов именование осуществляется посредством связывания конкретного процесса (его идентификатора PID) с конкретным сокетом, через который и происходит взаимодействие.

Второй блок организации взаимодействия — это **взаимодействие в пределах сети**. На сегодняшний день наиболее распространенными являются аппарат **сокетов** и система **MPI**.

**Система MPI** (интерфейс передачи сообщений) также является достаточно распространенным средством организации взаимодействия в рамках сети. Эта система иллюстрирует механизм передачи сообщений, речь о котором шла выше.

## 40. Файловые системы. Структурная организация файлов. Атрибуты файлов. Основные правила работы с файлами. Типовые программные интерфейсы работы с файлами.

Под **файловой системой** (ФС) будем понимать часть операционной системы, представляющую собой совокупность организованных наборов данных, хранящихся на внешних запоминающих устройствах, и программных средств, гарантирующих именованный доступ к этим данным и их защите. **файловая система** — это компонент операционной системы, обеспечивающий **корректный именованный доступ** к данным пользователя.

1. Первой моделью файла явилась модель файла **как последовательности байтов**. В этом случае содержимое файла представляется как неинтерпретируемая информация (или интерпретируемая примитивным образом). Задача интерпретации данных ложится на пользователя. 2. Следующие модели представляют файл как **последовательность записей переменной и постоянной длины**. Первая из этих моделей является аналогом магнитной ленты. Соответственно, эта организация файла и рассчитана на работу с магнитными лентами. Модель файла как последовательности записей постоянной длины является также аппаратно-ориентированной: она является аналогом перфокарты. 3. И, наконец, **модель иерархической организации файла**. В данной модели организация файла имеет сложную логическую структуру, позволяющую организовывать динамическую работу с данными. Одной из наиболее распространенных структур является дерево, в узлах которого расположены записи.

Каждый файл обладает фиксированным набором параметров, характеризующих свойства и состояния файла, причем и долговременное (стратегическое), и оперативное состояния. Совокупность этих параметров называют **атрибутами файла**: — Под **именем файла** понимается последовательность символов, используя которую организуется именованный доступ к данным файла. — **Права доступа**. Данный атрибут характеризует возможность доступа к содержимому файла различным категориям пользователей. — **персонификация** — связан с предыдущим. Соответственно, данный атрибут содержит информацию о принадлежности файла. — **Тип файла** — информация о способе организации файла и интерпретации его содержимого. **Размер записи**. В системе имеется возможность указать, что данный файл организован в виде последовательности блоков данного размера, при этом размер определяется пользователем. Размер может быть **стационарным**, когда при создании файла указывается фиксированный размер блоков, и **нестационарным**, когда размер блока задается каждый раз при открытии файла. **Размер файла**. Данный атрибут имеет достаточно простой смысл; заметим, что обычно размер файла задается в байтах. **Указатель чтения/записи** — это указатель, относительно которого происходит чтение или запись информации. В общем случае с каждым файлом ассоциируются два указателя (и на чтение, и на запись).

Практически все файловые системы при организации работы с файлами действуют по схожим сценариям, которые в общем случае состоят из трех основных блоков действий. Первый этап — это **начало работы с файлом** (или **открытие файла**). В частности, для каждого открытого файла создается т.н. **файловый дескриптор** — системная структура данных, содержащая информацию об актуальном состоянии открытого файла (режимы, позиции указателей и т.п.). Второй блок действий образуют **операции по работе с содержимым файла** (чтение и запись), также **операции, изменяющие атрибуты файла** (режимы доступа, изменение указателей чтения/записи и т.п.). Последний этап — это **закрытие файла**: уведомление системе о закрытии процессом файлового дескриптора (а не файла).

Структурно каждая операционная система предлагает унифицированный набор интерфейсов, посредством которых можно обращаться к системным вызовам работы с файлами. Обычно этот набор содержит следующие основные функции: — **open** — открытие/создание файла; — **close** — закрытие; — **read/write** — чтение/запись (относительно положения указателя чтения/записи соответственно); — **delete** — удаление файла из файловой системы; — **seek** — позиционирование указателя чтения/записи; —

**read\_attributes/write\_attributes** — чтение/модификация некоторых атрибутов файла.

**Каталог** — это системная структура данных файловой системы, в которой находится информация об именах файлов, а также информация, обеспечивающая доступ к атрибутам и содержимому файлов: **одноуровневая файловая система, двухуровневая файловая система, иерархическая файловая система**.

## Модельная организация каталогов файловых систем

**Каталог** — компонент файловой системы, содержащий информацию о содержащихся в файловой системе файлах. Каталоги являются специальным видом файлов.

### Модель одноуровневой файловой системы

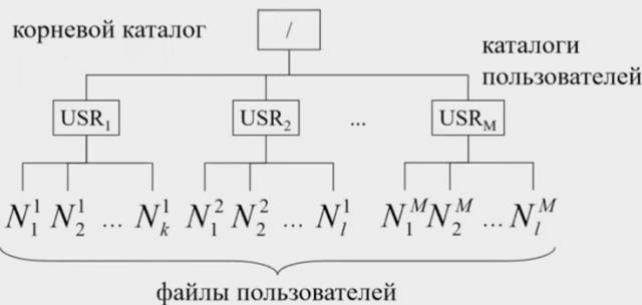
корневой каталог



NAME1 NAME2 NAME3 ...

## Модельная организация каталогов файловых систем

### Модель двухуровневой файловой системы



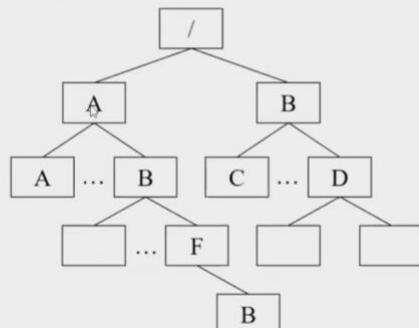
## Модельная организация каталогов файловых систем

### Иерархические файловые системы

#### Понятия

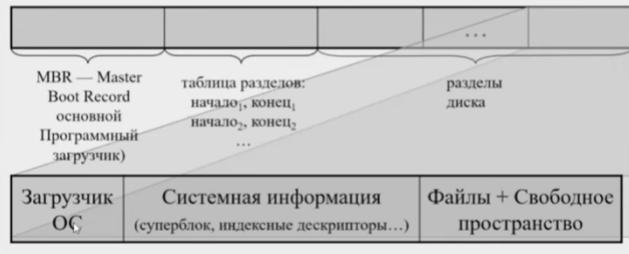
- имя файла
- полное имя файла
- относительное имя
- домашний каталог
- текущий каталог

/A/B/F/B  
B  
B/F/B



## Подходы в практической реализации файловой системы

### Структура «системного» диска



Блок физического HDD → Блок виртуального HDD  
Блок файловой системы  
Блок файла

## 41. Файловые системы. Модели реализации файловых систем. Понятие индексного дескриптора.

Первой тривиальной и самой эффективной с точки зрения минимизации накладных расходов является **модель непрерывных файлов**. Данная модель подразумевает размещение каждого файла в непрерывной области внешнего устройства. Эта организация достаточно простая: для обеспечения доступа к файлу среди атрибутов должны присутствовать имя, блок начала и длина файла.

Следующей моделью является **модель файлов, имеющих организацию связанного списка**. В этой модели файл состоит из блоков, каждый из которых включает в себя две составляющие: данные, хранимые в файле, и ссылка на следующий блок файла. Эта модель также является достаточно простой, достаточно эффективной

при организации последовательного доступа, а также эта модель решает проблему фрагментации свободного пространства.

Другой подход иллюстрирует модель, основанная на использовании т.н. **таблицы размещения файлов** (File Allocation Table — FAT, Рис. 109). В этой модели операционная система создает программную таблицу, количество строк в которой совпадает с количеством блоков файловой системы, предназначенных для хранения пользовательских данных. Также имеется отдельный каталог (или система каталогов), в котором для каждого имени файла имеется запись, содержащая номер начального блока. Соответственно, таблица размещения имеет позиционную организацию: i-ая строка таблицы хранит информацию о состоянии i-ого блока файловой системы, а, кроме того, в ней указывается номер следующего блока файла. Чтобы получить список блоков файловой системы, в которых хранится содержимое конкретного файла, необходимо по имени в указанном каталоге найти номер начального блока, а затем, последовательно обращаясь к таблице размещения и извлекая из каждой записи номер следующего блока, возможно построение искомого списка.

Следующая модель — модель организации файловой системы с использованием т.н. **индексных узлов** (дескрипторов). Принцип модели состоит в том, что в атрибуты файла добавляется информация о номерах блоков файловой системы, в которых размещается содержимое файла. Это означает, что при открытии файла можно сразу получить перечень блоков. Соответственно, необходимость использования FAT-таблицы

## Индексные узлы (дескрипторы)

Name	Номер 0 <sup>о</sup> блока файла
	Номер 1 <sup>о</sup> блока файла
	Номер 2 <sup>о</sup> блока файла
	Номер 3 <sup>о</sup> блока файла
	...
	Номер последнего блока файла

### Индексный узел (дескриптор)

— системная структура данных, содержащая информацию о размещении блоков конкретного файла в файловой системе.

#### Достоинства

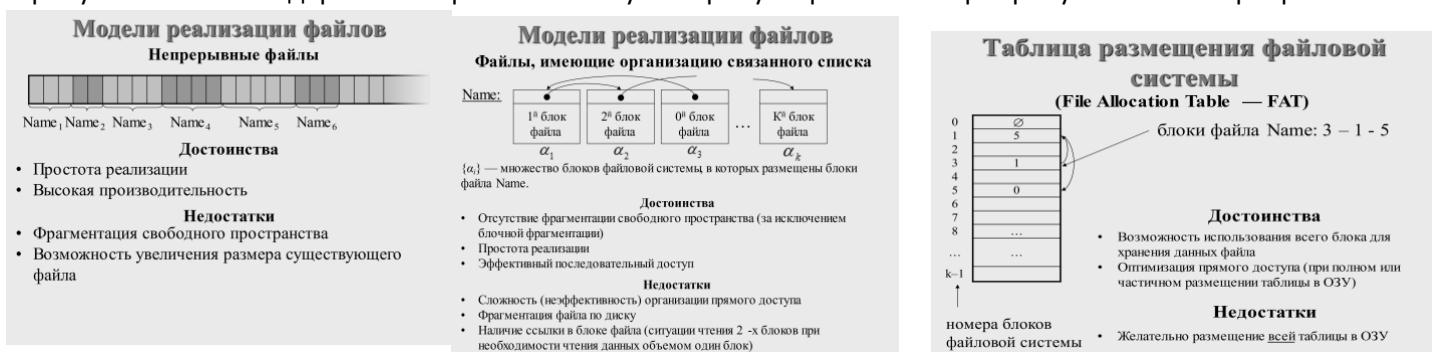
- Нет необходимости в размещении в ОЗУ информации всей FAT о всех файлах системы, в памяти размещаются атрибуты, связанные только с открытыми файлами
- Размер файла и размер индексного узла (в общем случае привязан к размерам таблицы размещения).

**Решение:**

- ограничение размера файла
- иерархическая организация индексных узлов

отпадает, зато, с другой стороны, при предельных размерах файла размер индексного дескриптора становится соизмеримым с размером FAT-таблицы. Для разрешения этой проблемы существует два принципиальных решения. Во-первых, это тривиальное ограничение на максимальный объем файла. Во-вторых, это построение иерархической организации данных о блоках файла в индексном дескрипторе. В последнем случае вводятся иерархические уровни представления информации: часть (первые N) блоков перечисляются непосредственно в индексном узле, а оставшиеся представляются в виде косвенной ссылки

**Модели организации каталогов** 1 Записи каталога фиксированного размера, содержат имя файла и все его атрибуты. 2 Каталог содержит имя файла и ссылку на атрибуты файла. Размер атрибутов может варьироваться.



## 42. Файловые системы.

**Координация использования пространства внешней памяти. Квотирование пространства ФС. Надежность ФС. Проверка целостности ФС.**

С точки зрения организации использования пространства внешней памяти файловой системой, существует несколько аспектов, на которые необходимо обратить внимание. Первый момент связан с проблемой выбора размера блока файловой системы. Задача определения оптимального размера блока не имеет четкого решения. Еще одна проблема, на которую стоит обратить внимание, — это **проблема учета свободных блоков** файловой системы. Первый подход заключается в том, что вся совокупность свободных блоков помещается в единый список, т.е. номера свободных блоков образуют **связный список**, который

располагается в нескольких блоках файловой системы. Для более эффективной работы первый блок, содержащий начальную часть списка, должен располагаться в ОЗУ, чтобы файловая система могла к нему оперативно обращаться. Вторая модель основана на использовании **битовых массивов**. В этом случае каждому блоку файловой системе ставится в соответствие двоичный разряд, сигнализирующий о незанятости данного блока. Для организации данной модели необходимо подсчитать количество блоков файловой системы, рассчитать количество разрядов массива, а также реализовать механизм пересчета номера разряда в номер блока и наоборот.

Как отмечалось выше, файловая система должна обеспечивать контроль использования двух видов системных ресурсов — это регистрация файлов в каталогах и контроль свободного пространства. Для решения поставленных задач в файловой системе вводятся квотирование имен файлов и квотирование блоков.

В общем случае модель квотирования может иметь два типа лимитов: **жесткий и гибкий**. Для каждого пользователя при его регистрации в системе определяются два типа квот. **Жесткий лимит** — это количество имен в каталогах или количество блоков файловой системы, которое пользователь превзойти не может: если происходит превышение жесткого лимита, работа пользователя в системе блокируется. **Гибкий лимит** — это значение, которое устанавливается в виде лимита; с ним ассоциировано еще одно значение, называемое счетчиком предупреждений (гибкий лимит превышать можно, но после этого включается обратный счётчик предупреждений).

Защита от потери информации в результате аппаратного или программного сбоя и Случайное удаление файлов. **Резервное копирование (архивирование)**: Копируются не все файлы файловой системы (избирательность архивирования по типам файлов), **Инкрементное архивирование** (резервное копирование) — единожды создается «полная» копия, все последующие включают только обновленные файлы, Использование **компрессии** при архивировании (риск потери всего архива из-за ошибки в чтении/записи сжатых данных), Проблема **архивирования «на ходу»** (во время копирования происходят изменения файлов, создание, удаление каталогов и т.д.), Распределенное хранение резервных копий

**Стратегии архивирования** : **Физическая архивация** -«Один в один» - Интеллектуальная физическая архивация (копируются только использованные блоки файловой системы)- Проблема обработки дефектных блоков. **Логическая архивация** — копирование файлов (а не блоков), модифицированных после заданной даты.

**Проблема** — при аппаратных или программных сбоях возможна потеря информации: потеря модифицированных данных в «обычных» файлах, потеря системной информации (содержимое каталогов, списков системных блоков, индексные узлы и т.д.). Контроль целостности или непротиворечивости файловой системы. Для выявления непротиворечивости и исправления возможных ошибочных ситуаций файловая система использует избыточную информацию, т.е. данные тем или иным образом (явно или косвенно) дублируются. **Контроль непротиворечивости блоков файловой системы**:

1Формируются две таблицы: таблица занятых блоков, таблица свободных блоков (размеры таблиц соответствуют размеру файловой системы — число записей равно числу блоков ФС)Изначально все записи таблиц обнуляются. 2. Анализируется список свободных блоков. Для каждого номера свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных 3. Анализируются все индексные узлы. Для каждого блока, встретившегося в индексном узле, увеличивается его счетчик на 1 в таблице занятых блоков 4. Анализ содержимого таблиц и коррекция ситуаций

## 43. Примеры реализаций файловых систем. Организация файловой системы ОС UNIX. Виды файлов. Права доступа. Логическая структура каталогов.



**Файл ОС Unix** — это специальным образом именованный набор данных, размещенных в файловой системе. Файлы ОС Unix могут быть разных типов: — **обычный файл** — это те файлы, с которыми регулярно имеет дело пользователь в своей повседневной работе (например, текстовый файл, исполняемый файл и т.п.); — **каталог** — файл данного типа содержит имена и ссылки на атрибуты, которые содержатся в данном каталоге; — **специальный файл устройств** — каждому устройству, с которым работает ОС Unix, должен быть поставлен в соответствие файл данного типа. Через имя файла устройства происходит обращение к устройству, а через содержимое данного файла (которое достаточно специфично) можно обратиться к конкретному драйверу этого устройства; — **именованный канал, или FIFO-файл** — **файл-ссылка, или символьская связь** —; — **сокет** — файлы данного типа используются для реализации унифицированного интерфейса программирования распределенных систем.

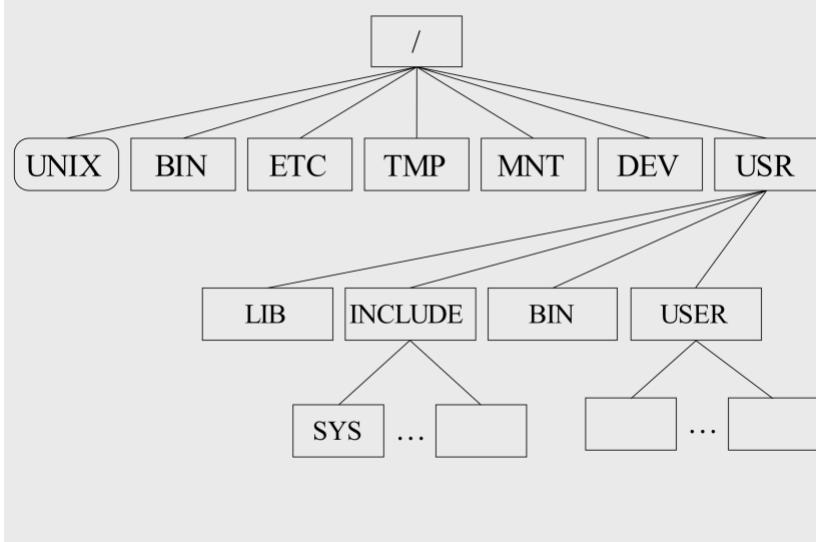
С каждым файлом также ассоциированы такие характеристики, как **права доступа к файлу**, которые регламентируют чтение содержимого файла, запись и исполнение файла.

Права на доступ к файлу разделяются на три категории пользователей — это права пользователя-владельца файла, права группы, к которой принадлежит владелец файла, исключая этого владельца, и, наконец, права всех остальных пользователей системы (без указанной группы владельца). При работе с каталогом его владельцу, члену соответствующей группы, и всем остальным пользователям может разрешаться чтение, запись и выполнение.

Одной из характеристик ОС Unix является характеристика, кажущаяся на первый взгляд достаточно странной: система рекомендует размещать системную и пользовательскую информацию по некоторым правилам.

Прежде всего, необходимо отметить, что файловая система ОС Unix является иерархической древовидной файловой системой, т.е. у нее есть корневой каталог /, из которого за счет каталогов разных уровней вложенности «вырастает» целое дерево имен файлов. Система предполагает, что в корневом каталоге всегда

## Логическая структура каталогов



расположен некоторый файл, в котором размещается код ядра операционной системы. Сразу оговоримся, что мы рассматриваем некоторую модельную систему.

В каталоге **/bin** находятся команды общего пользования. Каталог **/etc** содержит системные таблицы и команды, обеспечивающие работу с этими таблицами. Каталог **/tmp** является каталогом временных файлов, т.е. в этом каталоге система и пользователи могут размещать свои файлы на некоторый ограниченный промежуток времени.

Каталог **/mnt** традиционно используют для

монтирования различных файловых систем к данной системе. Операция монтирования в общих чертах заключается в том, что корень монтируемой файловой системы ассоциируют с данным каталогом (или с одним из его подкаталогов), после чего доступ к файлам подмонтированной системы осуществляется уже через этот каталог (т.н. **точку монтирования**). В каталоге **/dev** размещаются специальные файлы устройств, посредством которых осуществляется регистрация обслуживаемых в системе устройств и связь этих устройств с тем или иным драйвером. Каталог **/usr** можно охарактеризовать как каталог пользовательской информации.

## 44. Примеры реализаций файловых систем Внутренняя организация ФС.

### Модель версии UNIX SYSTEM V.

Рассмотрение внутренней организации файловой системы мы начнем с модели файловой системы, реализованной в ОС Unix версии System V. Данная файловая система была реализована одной из первых в ОС Unix и имеет название **s5fs**.

#### Модель версии System V Структура ФС

Суперблок	Область индексных дескрипторов	Блоки
-----------	--------------------------------	-------

**Суперблок** файловой системы содержит оперативную информацию о текущем состоянии файловой системы, а также данные о параметрах настройки.

**Индексный дескриптор** — специальная структура данных ФС, которая ставится во взаимооднозначное соответствие с каждым файлом.

**Блоки** — свободные, занятые под системную информацию, занятые файлами.

Файловая система занимает часть того раздела, в котором она находится (назовем его **системным разделом**, чтобы отличать его от разделов с другими файловыми системами, имеющими схожую организацию и которые можно примонтировать к данной системе), начиная с нулевого блока и заканчивая некоторым фиксированным блоком. Эта часть состоит из трех подпространств: суперблока, области индексных дескрипторов и блоков файлов.

**Достоинства ФС модели версии System V:** Оптимизация в работе со списками номеров свободных индексных дескрипторов и блоков,

Организация косвенной адресации блоков файлов

**Недостатки ФС модели версии System V:** Концентрация важной информации в суперблоке ,Проблема надежности , Фрагментация файла по диску, Ограничения на возможную длину имени файла

## 45. Примеры реализаций файловых систем Внутренняя организация ФС. Принципы организации системы FFS UNIX BSD.

46. Управление внешними устройствами. Архитектура организации управления внешними устройствами, основные подходы, характеристики. Первой исторической моделью стало **непосредственное управление** центральным процессором внешними устройствами, когда процессор на уровне микрокоманд полностью обеспечивал все действия по управлению внешними устройствами. Иными словами, поток управления полностью шел через ЦПУ, а наравне с ним через процессор шел и поток данных. Эта модель иллюстрирует синхронное управление: если начался обмен, то, пока он не закончится, процессор не продолжает вычисления.

Вторая модель, появившаяся с развитием вычислительной техники, связана с появлением **специализированных контроллеров устройств**, которые концептуально располагались между центральным процессором и соответствующими внешними устройствами (Рис. 144.Б). Контроллеры позволяли процессору работать с более высокоуровневыми операциями при управлении внешними устройствами. Таким образом, процессор частично освобождался от потока управления внешними устройствами за счет того, что вместо большого числа микрокоманд конкретного устройства он оперировал меньшим количеством более высокоуровневых операций. Но и эта модель оставалась синхронной.

Следующим этапом стало развитие предыдущей модели до **асинхронной**, осуществление которой стало возможным благодаря появлению аппарата прерываний. Данная модель позволяла запустить обмен для одного процесса, после этого поставить на счет другую задачу (или же текущий процесс может продолжить выполнять какие-то свои вычисления), а по окончании обмена (успешного или неуспешного) в системе возникнет прерывание, сигнализирующее возможность дальнейшего выполнения первого процесса. Но эти две модели предполагали, что поток данных идет через процессор.

Кардинальным решением проблемы перемещения обработки потока данных из процессора стало использование появившихся **контроллеров прямого доступа** к памяти (или DMA-контроллеров). Процессор генерировал последовательность управляющих команд, указывая координаты в оперативной памяти, куда

надо положить или откуда надо взять данные, а DMA-контроллер занимался перемещением данных между ОЗУ и внешним устройством. Таким образом, поток данных шел в обход процессора.

И, наконец, можно отметить последнюю модель, основанную на том, что управление внешними устройствами осуществляется с использованием **специализированного процессора** (или даже **специализированных компьютеров**) или **каналов ввода-вывода**. Данная модель подразумевает снижение нагрузки на центральный процессор с точки зрения обработки потока управления: ЦПУ теперь оперирует функционально-емкими макрокомандами.

Рассмотрим архитектуру программного управления внешними устройствами, которую можно представить в виде некоторой иерархии. В основании лежит **аппаратура**, а далее следуют программные уровни: **программы обработки прерываний, драйверы физических устройств**, а на вершине иерархии лежит уровень **драйверов логических устройств**, причем каждый уровень строится на основании нижележащего уровня.

Можно выделить следующие цели программного управления устройствами:

- унификация программных интерфейсов доступа к внешним устройствам (унификация именования, абстрагирование от свойств конкретных устройств) (стандартизация правил использования различных устройств)
- обеспечение конкретной модели синхронизации при выполнении обмена (синхронный, асинхронный обмен)
- обработка возникающих ошибок (индикация ошибки, локализация ошибки, попытка исправления ситуации);
- буферизация обмена
- обеспечение стратегии доступа к устройству (распределенный доступ, монопольный доступ);
- планирование выполнения операций обмена

## 47. Управление внешними устройствами. Планирование дисковых обменов, основные алгоритмы.

Рассмотрим различные стратегии организации планирования дисковых обменов. Будем рассматривать некоторое дисковое устройство, обмен с которым осуществляется дорожками (т.е. происходит обращение и считывание соответствующей дорожки). Пусть имеется очередь запросов к следующим дорожкам: 4, 40, 11, 35, 7, 14 и пусть изначально головка дискового устройства позиционирована на 15-ой дорожке. Замети, что время на обмен складывается из трех компонентов: выход головки на позицию, вращение диска и непосредственно обмен. Для оценки эффективности алгоритмов будем подсчитывать суммарный путь (выраженный в количестве дорожек), который пройдет головка для осуществления всех запросов на обмен из указанной очереди.

Первая стратегия, которую мы рассмотрим, — **стратегия FIFO**. Эта стратегия основывается лишь на порядке появления запроса в очереди. В нашем случае головка сначала начинает двигаться с 15 дорожки на 4, потом на 40 и т.д. После обработки всей указанной очереди суммарная длина пути составляет 135 дорожек, что в среднем можно охарактеризовать, как 22,5 дорожки на один обмен.

Альтернативой FIFO является **стратегия LIFO**. Этот алгоритм в нашем случае имеет примерно те же характеристики, что и FIFO. Но данная стратегия оказывается полезной, когда поступают цепочки связанных обменов: процесс считывает информацию, изменяет ее и обратно записывает. Для таких процессов эффективнее всего будет выполнение именно цепочки обмена, иначе после считывания он не сможет продолжаться, т.к. будет ожидать записи.

Следующая стратегия — **SSTF** основана на «жадном» алгоритме. Термин «жадного» алгоритма обычно применяется к итерационным алгоритмам для выделения среди них класса алгоритмов, которые на каждой

итерации ищут наилучшее решение. но данный алгоритм имеет существенный недостаток: ему присуща проблема голодания крайних дорожек.

Еще один алгоритм — **алгоритм, основанный на приоритетах процессов (PRI)**. Данный алгоритм подразумевает, что каждому процессу присваивается некоторый приоритет, тогда в заказе на обмен присутствует еще и приоритет. И, соответственно, очередь запросов обрабатывается согласно приоритетам. Здесь встает серьезная проблема корректной организации выдачи приоритета, иначе будут возникать случаи голодания низкоприоритетных процессов.

«**Лифт**» — сначала «движение» в одну сторону до «упора», затем в другую, также до «упора»

**Циклическое сканирование** Сканирование в одном направлении. Ищем минимальный номер дорожки, затем «движемся наверх»

Для решения проблемы зависания при интенсивном обмене с локальной областью диска применяются многошаговый алгоритм (**N-step-SCAN**).

Разделение очереди на подочереди длины  $\leq N$  запросов каждая (из соображений FIFO). Последовательная обработка очередей. Обрабатываемая очередь не обновляется. Обновление очередей, отличных от обрабатываемой.

Борьба с «залипанием» головки (интенсивный обмен с одной и той же дорожкой).

## 48. Управление внешними устройствами. Организация RAID систем, основные решения, характеристики.

Аббревиатура RAID может раскрываться двумя способами. RAID — Redundant Array of Independent (Inexpensive) Disks, или избыточный массив независимых (недорогих) дисков. На сегодняшний день обе расшифровки не совсем корректны. Понятие недорогих дисков родилось в те времена, когда большие быстрые диски стоили достаточно дорого, и перед многими организациями, желающими сэкономить, стояла задача построения такой организации набора более дешевых и менее быстродействующих и емких дисков, чтобы их суммарная эффективность не уступала одному «дорогому» диску.

**RAID-система** — это совокупность физических дисковых устройств, которая представляется в операционной системе как одно устройство, имеющее возможность организации параллельных обменов. Помимо этого образуется избыточная информация, используемая для контроля и восстановления информации, хранимой на этих дисках.

RAID-системы предполагают размещение на разных устройствах, составляющих RAID-массив, порций данных фиксированного размера, называемых **полосами**, которыми осуществляется обмен в таких системах. Размер полосы зависит от конкретного устройства.

**RAID 0.** В этой модели полоса соизмерима с дисковыми блоками, соседние полосы находятся на разных устройствах. Организация RAID нулевого уровня чем-то напоминает расслоение оперативной памяти. С каждым диском обмен может происходить параллельно. Каждое устройство независимое, т.е. движение головок в каждом из устройств не синхронизировано. Данная модель не хранит избыточную информацию, поэтому в ней могут храниться данные объемом, равным суммарной емкости дисков, при этом за счет параллельного выполнения обменов доступ к информации становится более быстрым.

**RAID 1, или система зеркалирования.** Предполагается наличие двух комплектов дисков. При записи информации она сохраняется на соответствующем диске и на диске-дублере. При чтении информации запрос направляется лишь одному из дисков. К диску-дублеру происходит обращение при утере информации на первом экземпляре диска.

Следующие две модели (**RAID 2 и RAID 3**) — это модели с т.н. синхронизированными головками, что, в свою очередь, означает, что в массиве используются не независимые устройства, а специальным образом синхронизированные. Эти модели обычно имеют полосы незначительного размера (например, байт или слово). Данные модели содержат избыточную информацию, позволяющую восстановить данные в случае выхода из строя одного из устройств. В частности, RAID 2 использует коды Хэмминга (т.е. коды, исправляющие одну ошибку и выявляющие двойные ошибки). Модель RAID 3 более проста, она основана на четности с чередующимися битами. Для этого один из дисков назначается для хранения избыточной информации — полос, дополняющих до четности соответствующие полосы на других дисках.

**RAID 4** является упрощением RAID 3. Это массив несинхронизированных устройств. Соответственно, появляется проблема поддержания в корректном состоянии диска четности. Для этого каждый раз происходит пересчет по соответствующей формуле.

Модели **RAID 5 и RAID 6** спроектированы так, чтобы повысить надежность системы по сравнению с RAID 3 и 4 уровней. Оказывается опасным хранить важную информацию (в данном случае полосы четности) на одном носителе, т.к. при каждой записи происходит обращение всегда к одному и тому же устройству, что может спровоцировать его скорейший выход из строя. Надежнее разнести служебную информацию по разным дискам. Соответственно, RAID 5 распределяет избыточную информацию по дискам циклическим образом, а RAID 6 использует двухуровневую избыточную информацию (которая также разнесена по дискам).

## 49. Внешние устройства в ОС UNIX. Типы устройств, файлы устройств, драйверы.

Как уже неоднократно упоминалось, одной из основных особенностей ОС Unix является концепция файлов: практически все, с чем работает система, представляется в виде файлов. Внешние устройства не являются исключением и также представлены в системе в виде специальных файлов устройств, хранимых обычно в каталоге /dev.

С точки зрения интерфейсов организации работы с внешними устройствами система делит абсолютно все устройства на две категории: **байт-ориентированные и блок-ориентированные устройства**. С блок-ориентированными устройствами обмен осуществляется порциями данных фиксированной длины, называемыми блоками. Обычно размер блока кратен степени двойки, а зачастую кратен 512 байтам. Все остальные устройства относятся к байт-ориентированным. Такие устройства позволяют осуществлять обмен порциями данных произвольного размера (от 1 байта до некоторого k).

Но, говоря о блок- и байт-ориентированных устройствах, следует помнить, что за регистрацию устройств в системе в конечном счете отвечает **драйвер** устройства: именно он определяет тип интерфейса устройства.

В системе имеется специальный каталог устройств, в котором располагаются файлы особого типа — специальные файлы устройств. Эти файлы обеспечивают решение следующих задач: – именование устройств (если быть более точными, то именование драйверов устройств); – связывание имени, выбранного для именования устройства, с конкретным драйвером.

Специальные файлы устройств не имеют блоков файла, хранимых в рабочем пространстве файловой системы. Вся содержательная информация файлов данного типа размещается исключительно в соответствующем

индексном дескрипторе. Индексный дескриптор состоит из перечня стандартных атрибутов файла, среди которых, в частности, указывается тип этого файла, а также включает в себя некоторые специальные атрибуты. Эти атрибуты содержат следующие поля: тип файла устройства (блок- или байт-ориентированное), а также еще 2 поля, позволяющие осуществлять работу с конкретным драйвером устройства, — это т.н. **старший номер и младший номер**. Старший номер — это номер драйвера в таблице драйверов, соответствующей типу файла устройства. А младший номер — это некоторая дополнительная информация, передаваемая драйверу при обращении. За счет этого реализуется механизм, когда один драйвер может управлять несколькими схожими устройствами.

Для регистрации драйверов в системе используются две системные таблицы: таблица блок-ориентированных устройств — **bdevsw**, и таблица байт-ориентированных устройств — **cdevsw**. Соответственно, старший номер хранит ссылку на драйвер, находящийся в одной из таблиц; тип таблицы определяется типом файла устройств.

Каждая запись этих таблиц содержит структуру специального формата, называемую **коммутатором устройства**. Коммутатор устройства хранит указатели на все возможные точки входа в соответствующий драйвер, либо же в соответствующей записи таблицы вместо указанной структуры хранится специальная ссылка-заглушка на точку ядра. Стоит отметить следующие типовые имена точек входа в драйвер: —  $\beta$ open(),  $\beta$ close(); —  $\beta$ read(),  $\beta$ write(); —  $\beta$ ioctl(); —  $\beta$ intr().

**Ситуации, вызывающие обращение к функциям драйвера:** Старт системы, определение ядром состава доступных устройств, Обработка запроса ввода/вывода, Обработка прерывания, связанного с данным устройством, Выполнение специальных команд управления

Изначально Unix-системы (как и большинство систем) предполагали «жесткое» статические встраивание драйверов в код ядра. Это означало, что при добавлении нового драйвера или удалении существующего необходимо было выполнить достаточно трудоемкую операцию перетрансляции (когда ядро создается «с нуля») или, как минимум, перекомпоновку ядра (когда есть готовые объектные модули).

Альтернативной моделью, существующей и по сей день, является модель динамического связывания драйверов. В этом случае в системе присутствуют программные средства, позволяющие динамически, «на лету» подключить к операционной системе тот или иной драйвер.

## 50. Внешние устройства в ОС UNIX. Системная организация обмена с файлами. Буферизация обменов с блокоориентированными устройствами.

Для организации операций обмена в ОС Unix используются системные таблицы и структуры, часть которых ассоциирована с каждым процессом, а часть — с самой ОС.

**Таблица открытых файлов** (ТОФ) создается в адресном пространстве процесса. Каждая запись этой таблицы соответствует открытому в процессе файлу. Говоря о номере дескриптора открытого в процессе файла (т.н. файлового дескриптора); подразумевается соответствующий номер записи в таблице открытых файлов процесса. Размер данной таблицы определяется при настройке операционной системы: этот параметр декларирует предельное количество открытых в одном процессе файлов.

Каждая запись ТОФ содержит целый набор атрибутов, который в данный момент нам не интересен, но в этом наборе имеется один достаточно важный атрибут — это ссылка на номер записи в **таблице файлов** операционной системы (ТФ). Таблица файлов ОС является системной таблицей, она представлена в системе в единственном экземпляре. В этой таблице происходит регистрация всех открытых в системе файлов.

В таблице файлов ОС помимо прочего содержатся такие атрибуты, как **указатель чтения/записи** (ссылающийся на позицию в файле, начиная с которой будет происходить, соответственно, чтение или запись), счетчик кратности (речь о нем пойдет ниже) и ссылка на таблицу индексных дескрипторов открытых файлов.

**Таблица индексных дескрипторов открытых файлов** (ТИДОФ) также является системной структурой данных, содержащей перечень индексных дескрипторов всех файлов, открытых в данный момент в системе. Каждая

запись этой таблицы содержит актуальную копию открытого в системе индексного дескриптора. Здесь также хранится целый набор параметров, среди которых имеется и счетчик кратности.

Одним из достоинств ОС Unix является организация многоуровневой буферизации при выполнении неэффективных действий. В частности, для организации блокориентированных обменов система использует стандартную стратегию кэширования.

Для буферизации используют пул буферов, размером в один блок каждый. Вкратце рассмотрим алгоритм, состоящий из пяти действий.

1. Поиск заданного блока в буферном пуле. Если удачно, то переход на п.4
2. Поиск буфера в буферном пуле для чтения и размещения заданного блока.
3. Чтение блока в найденный буфер.
4. Изменение счетчика времени во всех буферах.
5. Содержимое данного буфера передается в качестве результата.
  - Оптимизация работы ОС, за счет минимизации реальных обращений к физическому устройству
  - Недостатки:
    - Критичность к несанкционированным отключениям питания
    - Разорванность во времени факта обращения к системе за обменом и реальным обменом

### **Борьба со сбоями**

- Наличие параметра, определяющего периоды времени, через которые осуществляется сброс системных данных, который может оперативно меняться
- Пользовательская команда SYNC
- Избыточность системы, позволяющая восстанавливать информацию

## **51. Управление оперативной памятью. Одиночное непрерывное распределение. Распределение разделами. Распределение перемещаемыми разделами**

Основные задачи:

1. Контроль состояния каждой единицы памяти (свободна/распределена)
2. Стратегия распределения памяти (кому, когда и сколько памяти должно быть выделено)
3. Выделение памяти (выбор конкретной области, которая должна быть выделена)
4. Стратегия освобождения памяти (процесс освобождает, ОС “забирает” окончательно или временно)

**Одиночное непрерывное распределение.** Данная модель распределения оперативной памяти является одной из самых простых и основывается на том, что все адресное пространство подразделяется на два компонента. В одной части памяти располагается и функционирует операционная система, а другая часть выделяется для выполнения прикладных процессов. Весь процесс помещается в ОЗУ. Нельзя его куками записывать. Также остается много неиспользованной памяти.

**Необходимые аппаратные средства:** Регистр границ + режим ОС / режим пользователя

- Если ЦП в режиме пользователя попытается обратиться в область ОС, то возникает прерывание

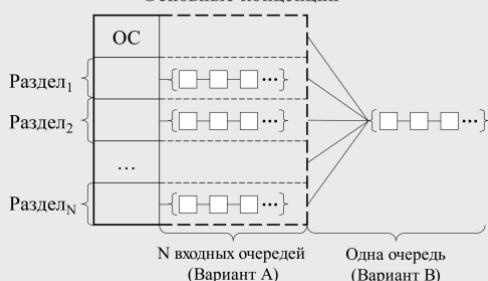
### Одиночное непрерывное распределение

Основные концепции



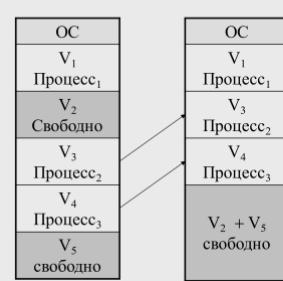
### Распределение неперемещаемыми разделами

Основные концепции



### Распределение перемещаемыми разделами

Основные концепции



Виртуальная память  
Прогресс<sub>4</sub>  
(например, V = V<sub>2</sub> + ½ V<sub>5</sub>)

**Распределение неперемещаемыми разделами:** Опять же, все адресное пространство оперативной памяти делится на две части. Одна часть отводится под операционную систему, все оставшееся пространство отводится под работу прикладных процессов, причем это пространство заблаговременно делится на N частей (назовем их разделами), каждая из которых в общем случае имеет произвольный фиксированный размер. Эта настройка происходит на уровне операционной системы. Соответственно, очередь прикладных процессов разделяется по этим разделам.

Существуют концептуально два варианта организации этой очереди. Первый вариант (вариант Б) предполагает наличие единственной сквозной очереди, которая по каким-то соображениям распределяется между этими разделами. Второй вариант (вариант А) организован так, что с каждым разделом ассоциируется своя очередь и поступающий процесс сразу попадает в одну из этих очередей.

**Необходимые аппаратные средства:** Существуют несколько способов аппаратной реализации данной модели. С одной стороны, это использование двух регистров границ, один из которых отвечает за начало, а второй — за конец области прикладного процесса. Выход за ту или иную границу ведет к возникновению прерывания по защите памяти.

**Распределение перемещаемыми разделами** Данная модель распределения разрешает загрузку произвольного (нефиксированного) числа процессов в оперативную память, и под каждый процесс отводится раздел необходимого размера. Соответственно, система допускает перемещение раздела, а, следовательно, и процесса. Такой подход позволяет избавиться от фрагментации.

**Необходимые аппаратные средства:** аппаратные средства защиты памяти (Регистры границ + регистр базы) и аппаратные средства, позволяющие осуществлять перемещение процессов (Ключи + регистр базы)

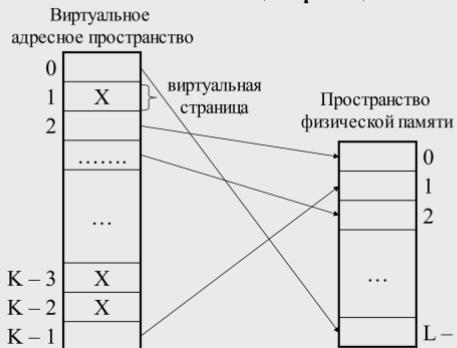
## 52. Управление оперативной памятью. Страницочное распределение.

Данная модель основывается на том, что все адресное пространство может быть представлено совокупностью блоков фиксированного размера, которые называются **страницами**. Есть **виртуальное адресное пространство** — это то пространство, с адресами которого оперирует программа, и **физическое адресное пространство** — это то пространство, которое есть в наличии в компьютере. Соответственно, при

### Страницочное распределение

Основные концепции

Таблица страниц



страницочном распределении памяти существуют программно-аппаратные средства, позволяющие устанавливать соответствие между виртуальными и физическими страницами. Механизм преобразования виртуального адреса в физический обсуждался ранее, он достаточно прост: берется номер виртуальной страницы и заменяется соответствующим номером физической страницы. Также отмечалось, что для этих целей используется т.н. **таблица страниц**, которая целиком является аппаратной, что на самом деле является большим упрощением.

Проблема:

1. Размер таблицы страниц (количество 4КБ страниц при 32-х разрядной адресации — 1 000 000; любой процесс имеет собственную таблицу страниц)
2. Скорость отображения (сама таблица страниц находится в ОЗУ)
3. Проблемы при замене контекста процесса

### Необходимые аппаратные средства

1. Полностью аппаратная таблица страниц (стоимость, полная перегрузка при смене контекстов, скорость преобразования)
2. Таблица страниц в ОЗУ + Регистр начала таблицы страниц в памяти (простота, управление смены контекстов, медленное преобразование)
3. Гибридные решения

### Страницочное распределение

#### Алгоритмы и организация данных

Размер и организация таблицы страниц ???

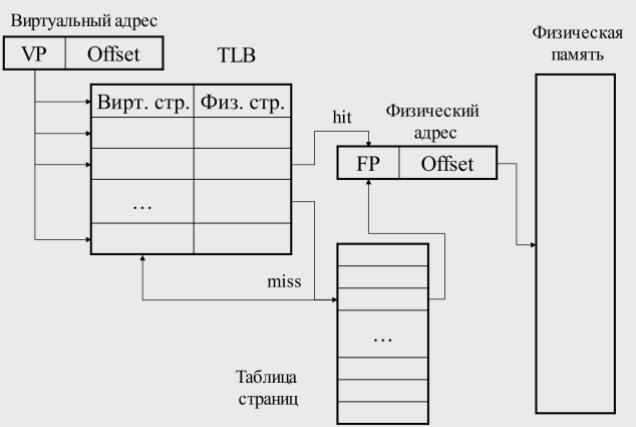
#### Модельная структура записи таблицы страниц

$\varepsilon$	$\delta$	$\gamma$	$\beta$	$\alpha$	Номер физической страницы
$\alpha$	— присутствие/отсутствие				
$\beta$	— защита (чтение, чтение/запись, выполнение)				
$\gamma$	— изменения				
$\delta$	— обращение (чтение, запись, выполнение)				
$\varepsilon$	.....				

В качестве одного из первых решений оптимизации работы с памятью стало использование т.н. **TLB-таблиц** (Translation Lookaside Buffer — таблица быстрого преобразования адресов, Рис. 136). Данный метод подразумевает наличие аппаратной таблицы относительно небольшого размера (порядка 8 – 128 записей). Данная таблицы концептуально содержит три столбца: первый столбец — это номер виртуальной страницы, второй — это номер физической страницы, в которой находится указанная виртуальная страница, а третий столбец содержит упомянутые выше атрибуты. Эта таблица находится в процессоре. Регистровая память. Она реализована аппаратно. Если miss, то фиксируется страницочное прерывание. ОС залезает в таблицу страниц и записывает

нужную VP в TLB.

### TLB (Translation Lookaside Buffer)



Одним из решений, позволяющих снизить размер таблицы страниц, является **модель иерархической организации таблицы страниц**. В этом случае информация о странице представляется не в виде одного номера страницы, а в виде совокупности номеров, используя которые, можно получить номер соответствующей физической страницы, посредством обращения к соответствующим таблицам, участвующим в иерархии (это может быть 2-х-, 3-х- или даже 4-х уровневая иерархия). Таблица первого уровня в памяти всегда, а второго можно подгружать. При смене процессоров нам не надо всю загружать. => снижаем размер таблицы страниц



### Инвертированные таблицы страниц



**Проблема** — поиск по таблице (хэширование)

**Решение** проблемы перезагрузки таблицы страниц при смене обрабатываемых ЦП процессов

Инвертированная таблица страниц — одна для всех. Ее не надо менять

## 53. Управление оперативной памятью.

### Сегментное распределение.

Недостатком страничного распределения памяти является то,

что при реализации этой модели процессу выделяется единый диапазон виртуальных адресов: от нуля до некоторого предельного значения. С одной стороны, ничего плохого в этом нет, но это свойство оказывается неудобным по следующей причине. В процессе есть команды, есть статические переменные, которые, по сути, являются разными объединениями данных с различными характеристиками использования. Еще большие отличия в использовании иллюстрируют существующие в процессе стек и область динамической памяти, называемой также кучей. И модель страничной организации памяти подразумевает статическое разделение единого адресного пространства: выделяются область для команд, область для размещения данных, а также область для стека и кучи. При этом зачастую стек и куча размещаются в единой области, причем стек прижат к одной границе области, куча — к другой, и «растут» они навстречу друг другу. Соответственно, возможны ситуации, когда они начинают пересекаться (ситуация переполнения стека). Или даже если стек будет располагаться в отдельной области памяти, он может переполнить выделенное ему пространство. Таковы основные недостатки страничного распределения памяти.

Избавиться от указанных недостатков на концептуальном уровне призвана модель сегментного распределения памяти. Данная модель представляет каждый процесс в виде совокупности сегментов, каждый из которых может иметь свой размер. Каждый из сегментов может иметь собственную функциональность: существуют сегменты кода, сегменты статических данных, сегмент стека и т.д. Для организации работы с сегментами может использоваться некоторая таблица, в которой хранится информация о каждом сегменте (его номер, размер и пр.). Тогда виртуальный адрес может быть проинтерпретирован как номер сегмента и величина смещения в нем.

Модель сегментного распределения может иметь достаточно эффективно работающую аппаратную реализацию. Существует аппаратная таблица сегментов с фиксированным числом записей. Каждая запись этой таблицы соответствует своему сегменту и хранит информацию о размере сегмента и адрес начала сегмента (т.е. адрес базы), а также тут могут присутствовать различные атрибуты, которые будут оговаривать права и режимы доступа к содержимому сегмента.

Итак, имея виртуальный адрес, система аппаратным способом извлекает из него номер сегмента  $i$ , обращается к  $i$ -й строке таблицы, из которой извлекается информация о сегменте. После этого происходит

проверка, не превосходит ли величина сегмента размера самого сегмента. Если превосходит, то происходит прерывание; иначе, складывая базу со смещением, вычисляется физический адрес.



54. Вычислительная система. Кэширование информационных потоков на уровнях аппаратуры и ОС

55. Язык С