

# 1. Этапы развития вычислительной техники и программного обеспечения.

	Время	Быстрота	Состоит из	Особенности	Прим.
I поколение Электронные лампы	1946 - 1959	10000 тыс. оп/сек	• Процессор • Опер. запомин. у-во • У-во вывода • Внешние запомин. устройства (магнитные ленты) • У-во ввода (перфокарт)	• Однопользовательский режим	ENIAC
Зарождение языков программирования				• Ассемблерные языки • Высоконивелированные языки • Простые языки	
II поколение Полупроводниковые транзисторы	1959 - 1969	Сотни тыс. оп/сек	• Печатные платы с 4-6 позиц. компонентами • Модем для соединения	• Пакетная обработка заданий • Разделение на запрограммированные и смешанные у-ва хранения и вычисления • Мультипрогр. • Язык Управления Заданиями • Первые графические системы	IBM 650
III поколение Интегральные схемы	1970 - 1979	Миллионы оп/сек	• Драйверы УС-В • Виртуальное УС-В • Устройства управления • Пакетные прикладные программы	• Начало аппаратной унификации узлов и у-ва • Модульный принцип изг. • Появление ОС UNIX	IBM 360
IV пок. и дальше	1980 +	10-100 млн оп/сек		• Персональное оборудование • Формирование ОС	

I поколение, неоднородные. Электро-вакуумные лампы, высокое энергопотребление → высокая энергопотребность → тепло → частый выход из строя из-за перегрева. Лампа подключается к плате через разъем, расширение / сужение отходов от контактов. Габаритные устройства → длинные проводники → скорость. Дороговизна. Недороговизна.

- 1) Возможность охлаждения: появляются надежные способы с точки зрения аппаратурного
- 2) Справочное функции, наличие ROM - энергонезависимого памяти
- 3) Развитие автокодов / языков ассемблера - средство наматывания программ в виде машинных обозначений машинных языков и операндов ⇒ нужна специальная программа, переводящая машинный код в машинный ⇒ транслайтер
- Ассемблер языком сложен в конкретной архитектуре
- Трудоемкое программирование
- 4) Однопользовательский режим, однопрограммный
- Трата времени

## II поколение. Транзисторы и ЗУОУ

- Устье вставлялось в плату
  - + габариты
  - + У энергопотребление / выделение
  - + число проблем отхода от контактов
  - + надежность
- Резкие пакетные бордюрные зацепления. В пакетах контакты есть программа - пакетный обработчик, работающий машинно - пакетный однопрограммный режим
- Появление микропрограммирования:
  - 1) программа выполняется
  - 2) машино-запуск обработка
  - 3) готовая к выполнению и ждет передачи управления
- + более эффективная загрузка процессора
- Виртуализация: виртуальное устройство ⇒ нужен элемент, связанный с организацией данных
- доступ к содержимому файлов посредством универсализированного файлового интерфейса
- Первые ОС - комплекс программы, обеспечивающей обработку программ пользователя и распределение между этими программами ресурсов системы
- Аппарат прерываний

## III поколение. Интегральные схемы малой интеграции

- + габариты
- + энергопотребление / выделение
- + надежность
- 1) аппаратная унификация узлов и устройств для разных производителей
- + взаимозаменяемость
- 2) возможность создания и унификации узлов компьютеров: условие программной преемственности
- семейства компьютеров: условие программной преемственности

1) 1е пок - электронно-вакуумные лампы, 1940-1950.

Зарождение класса сервисных, управляющих программ. Зарождение языков программирования. Однопользовательский, персональный режим.

2) 2е пок - полупроводниковые приборы: диоды и транзисторы, 1950-1960, пакетная обработка заданий, мультипрог., языки управления заданиями, файловые системы, виртуальные устройства, ОС.

3) 3е пок - интегральные схемы малой и средней интеграции, 1960-1970, аппаратная унификация узлов и устройств, создание семейств компьютеров, унификация компонентов прогр обеспечения.

4) 4е пок - большие и сверхбольшие интегральные схемы, 1970-х – настоящее время. «Дружественность» пользовательских интерфейсов, сетевые технологии, безопасность хранения и передачи данных.

### Первое поколение компьютеров:

середина 40-х – начало 50-х годов XX века.

- электронно-вакуумные лампы.

- В 1946 г. в Пенсильванском университете США ENIAC (Electronic Numerical Integrator and Computer) по заказу министерства обороны США и применялась для решения задач энергетики и баллистики.

- Производительность 100-1000 оп/сек

- ЦП, ОЗУ та и 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 ОП компьютера предварительно подготовленные на специальных носителях (перфокартах, перфоленте и пр.) программы и данные.

- однопользовательский режим.

Пользователь (программист) использовал аппаратную консоль (или пульт управления) компьютера для ввода и запуска программы чтения данных через устройства ввода. Результат выполнения программы выводился на устройство печати. Ошибка по выполнению команд - программы прерывалась, и возникшая ситуация отображалась на индикаторах пульта управления, содержание которых анализировалось программистом. Нет средств автоматизации программирования. Вручную на регистрах задать точку входа программы.

- Программирование в машинных кодах

привносило ряд проблем, связанных с тех. сложностями написания, модификации и отладки программы. Программист должен кодировать все необходимых операций ввода/вывода с помощью машинных команд управления внешними устройствами => Должен знать алгоритмы, знать организации и использования аппаратуры компьютера, системы команд и кодировки, используемой в данном компьютере, знание особенностей программирования устройств ввода/вывода (инженер-электр).

- Класс программ, обеспечивающих сервисные функции програм. — это ассемблеры (в комп с помощью загрузчика), 1-е языки высокого уровня и трансляторы для этих языков, а также простейшие средства организации и использования библиотек программ. Ассемблер – средство нотации программы в виде мнемонических обозначений машинных команд и операндов => необходимость сервисной программы, переводящих ассемблер в машинные команды.

- Зарождении класса сервисных, управляющих программ чтения и загрузки в ОП программ и данных с внешних устройств, предопределены фирмой-разработчиком компьютеров и вводились в ОП с использованием аппаратной консоли: возможно вручную последовательность команд (составл управляемую программу) и запустить. В случае если управляющая программа размещалась на ВЗУ, через аппаратную консоль вводилась последовательность команд, обеспечивающих чтение кода управляющей программы в ОП и передачу управления на ее точку входа.

#### **Минусы:**

- вручную загружается программа – опечатки, не понятно где ошибка.
- нужны охладители. Проблема надежности с точки зрения конструкции.
- С точки зрения сервисных функций.
- Нагревание – охлаждение – отходят контакты.
- Плохо интерпретируемый не наглядный вид программы (до ассемблеров)
- Ассемблер архитектурно зависимый, на нем медленно и неэффективно писать => 50-е годы фортран.
- Компьютеры строились, как единые, аппаратно-целостные устройства, комплектация и возможности которых были существенно предопределены на этапе их производства. Их аппаратная модификация, обычно, была крайне затруднительна.

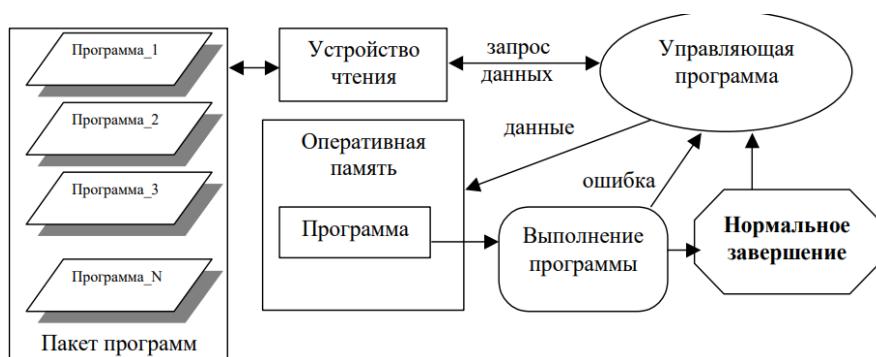
#### Компьютеры второго поколения: конец 50-х годов — вторая половина 60-х годов XX века.

- Полупроводниковые приборы — диодов и транзисторов, которые по функциям, размеру, энергопотреблению в 10-ки раз превосходили возможности электронно-вакуумных ламп => более развитые логическими возможностями и в 1000 раз превосходили компьютеры 1 пок по производительности.

- Распространение получили новые высокопроизводительные внешние устройства ЭВМ => совершенствование программного обеспечения и способов использования компьютеров.

- Аппаратная ф-ия обработки ошибок (развитие в аппарат прерываний).

- Пакетная обработка заданий: в компьютере работала специальная управляющая программа последовательно загружает в ОП и запускает на выполнение программы из заранее подготовленного пакета программ («большой» стопки перфокарт, в которой программы находятся последовательно). Завершение текущей программы -> чтение очередной прог через у-во считывания первокарт -> загрузка в ОП -> управления на фиксированную точку входа в программу (адрес памяти с которого должно начинаться выполнение программы). По завершении программы/ошибки, вызывающую аварийную остановку выполнения программы, управление передается в управляющую прогу. Вып до того, пока все проги из пакета не будут выполнены => прообразы ОС (мониторные системы/супервизоры/диспетчеры) => эффективность загрузка процессора.



- Аппаратная поддержка мультипрог (одновременно находились в обработке несколько программ). При этом в каждый момент времени команды одной из обрабатываемых программ выполнялись процессором, другие выполняли обмен данными с внешними устройствами, третьи были готовы к выполнению процессором и ожидали своей очереди.

- Развитие «дружественности» интерфейсов между пользователем и системой => развитие языков программирования и систем программирования: система команд компьютера - автокоды и ассемблеры - языки программирования высокого уровня - проблемно-ориентированные языки программирования.

- Развитие ОС как комплекс программ, обеспечивающий распределение ресурсов между программами: появились языки управления заданиями, которые позволяли пользователю до начала выполнения его программы сформировать набор требований по организации выполнения программы.

- Аппарат прерываний – аппаратно программные с-ва компа, предполагающие предопределенную реакцию на возникновение в компе/системе одного из некоторых заранее опр событий – прерываний.

- Виртуализация, средство создания и использования виртуальных ресурсов (часть эксплуатационных хар-к реализуются

программно). Прообразы современных файловых систем — систем, позволяющих систематизировать и упростить способы хранения и доступа пользователей к данным, размещенным на внешних запоминающих устройствах, что позволило пользователю работать с данными во внешней памяти в терминах имен или адресов некоторых наборов данных => возможность абстрагироваться от знания особенностей и способов организации хранения и доступа к данным конкретных физических устройств => основа для появления виртуальных устройств.

- Появление унифицированного файлового интерфейса доступа (для файловой системы).
- Компьютеры строились, как единые, аппаратно-целостные устройства, комплектация и возможности существенно предопределены на этапе их производства. Их аппаратная модификация, обычно, была крайне затруднительна.
- Автоматизация гражданских задач.

### Компьютеры третьего поколения: конец 60-х — начало 70-х годов XX века.

- В качестве элементной базы интегральные схемы малой интеграции => функциональное наполнение элементной базы, ↗ производительности компьютеров, надежность, ↓ размеров, веса, энергопотребления, появление новых, высокопроизводительных внешних устройств
- Начало аппаратной унификации их узлов и устройств => создание семейств компьютеров, аппаратная комплектация которых могла достаточно просто варьироваться владельцем компьютера (IBM-360 фирмы IBM, PDP-11 фирмы DEC).
- Компы строятся на модульном принципе => можно осуществлять замену и расширение состава внешних устройств, увеличивать размеры ОП, заменять процессор на более производительный => развитие и структуру ОС, которые потом приобрели модульную организацию с унификацией межмодульных интерфейсов. Семейство компьютеров – группа компов разной пр-сти, для которых выполняется условие программной преемственности снизу вверх => оптимизация в рамках предприятия расходов на технику.
- В ОС 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.
- Расширение круга проблем, специализация.
- Унификация компонентов ПО. Унификация и стандартизация языков программирования.
- Для обеспечения простоты и «дружественности» общения пользователя с различными устройствами компьютера 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. Новые режимы использования компьютеров: диалоговый режим доступа к компьютеру.
- ОС Unix (стал стандартом), которая открыла направление развития комплексной стандартизации пользовательских интерфейсов, как на уровне интерфейсов командных языков, так и на различных уровнях программных интерфейсов от правил взаимодействия с драйверами устройств до интерфейсов с прикладными системами. Завершение формирования сегодняшнего понятия ОС может быть связано с появлением четвертого и последующих поколений компьютеров, в построении которых использовалась элементная база, основанная на больших интегральных схемах. Система не экспериментальная, а широко распространенная ОС. POSIX.
- Язык Си, не является машинно-ориентированный, средствами которого можно эффективно программировать ПО (по производительности, размеру, скорости решений).
- В России тралы с элементной базой, отечественные проекты закрыты, копируем запад.

### Компьютеры четвертого поколения,

- Развитие элементной базы, первых микропроцессоров.
- Персональный комп => революция в массовом распространении ИТ => ряд проблем: совершенствование «дружественности» пользовательских интерфейсов => ОС Microsoft, которые в полном смысле слова совершили революцию в обеспечении массовости освоения компьютера.
- Расширилась область встроенных компов (управление технологическими процессами в режиме реального времени). Компы – бытовые ус-ва.
- Развились сетевые технологии => появлению сетевых и распределенных ОС => Internet => задачи обеспечения ОС безопасности хранения и передачи данных => определена типовая структура и задачи ОС и функции ее основных компонентов, принципиально новые разновидности ОС и режимов использования компьютеров.
- Сегодня аппаратура и программное обеспечение современных компьютеров представляют единую взаимозависимую вычислительную систему, в которой многие функции ОС нельзя рассматривать вне контекста аппаратной поддержки компьютера, а многие аппаратные возможности сложно рассматривать вне контекста ОС.
- Сверхвысокая интеграция (в одном компе очень много узлов).

---

## **2. Структура вычислительной системы. Ресурсы ВС - физические, виртуальные. Уровень ОС**

Функционирование компьютера = функционирование системы, в которой интегрированы аппаратура

компьютера и его ПО => вычислительная система (ВС), возможности и эксплуатационные качества которой определяются как аппаратурой компьютера, так и функционирующим на нем ПО.

**Вычислительная система** - совокупность аппаратных и программных средств, функционирующих в единой системе и предназначенных для решения задач определенного класса.

Некоторая иерархия уровней, нижестоящий уровень предоставляет интерфейса для вышестоящих.

Структура ВС. Каждый из уровней определяет свой уровень абстракции свойств ВС.

Взаимосвязи уровней иерархической структуры ВС, их характеристики могут проявляться как в виде непосредственных межуровневых интерфейсов, определенных однозначным набором правил использования объектов одного уровня другим, так и косвенным влиянием одного уровня на другой: влияние,

оказываемое на характеристики функционирования всей вычислительной системы в целом, производительности или емкости аппаратных компонентов компьютера (внешних устройств, процессора, оперативной памяти, линий связи и пр.).

**1. Аппаратный уровень ВС** определяется набором аппаратных компонентов и их характеристик, используемых вышестоящими уровнями иерархии и оказывающими влияние на эти уровни, предоставляют т.н. физические ресурсы, или физические устройства ВС.

- аппаратные компоненты компьютера: Процессор компьютера, ОП, внешние устройства, входящие в состав компьютера. Физический ресурс – то, что можно программно использовать, аппаратный компонент компьютера, обладающий **характеристиками**:

– правила программного использования, определяющие возможность корректного использования данного ресурса в программе (для процессора компьютера эти правила описывают **машинный язык** — систему команд данного компьютера, на основании которой возможно построение работающей программы, для внешнего устройства компьютера подобные правила описывают способы программного управления данным устройством, к примеру, команды ввода вывода процессора). Экран – физ ресурс, шнурок – нет (нет правил прогр исп.);

– параметры, характеризующие его объемные характеристики и/или производительность (для процессора компьютера таким параметром может служить его тактовая частота, а для ВЗУ – объем информации, которая может храниться на данном устройстве и скорость доступа);

– степень использования данного физического ресурса в ВС – это параметры, которые характеризуют степень занятости или используемости данного физического ресурса (для процессора компьютера такой

характеристикой является время его работы, затраченное на выполнение программ пользователей, для ОЗУ это будет объем используемой памяти, для линий связи — это ее загруженность).

- пользователю ВС предоставлены в качестве средств программирования система команд компьютера и аппаратные интерфейсы программного взаимодействия с физическими ресурсами = средства программирования на ранних этапах освоения компьютеров первого поколения.

## 2. Уровень управления физическими ресурсами — это 1й уровень системного ПО ВС, программный.

- назначение — систематизация и стандартизация правил программного использования физических ресурсов.
- пользователю доступна система команд компьютера

*Программирование управления физическими устройствами — кропотливая работа, необходимо учитывать сложную логику организации взаимодействия с конкретным устройством компьютера, нужна модификация кода программы в части, обеспечивающей это взаимодействие => трудозатраты, снижение надежности программы из-за роста риска внесения ошибок в логику ее работы => драйверы.*

- Драйвер физического устройства — программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством. Предоставляет вышестоящему уровню унифицированный программный интерфейс работы с каждым физическим устройством, скрывает от пользователя детальные элементы управления конкретным устройством и предоставляет пользователю упрощенный программный интерфейс работы с устройством. Интерфейс ориентирован на конкретные свойства устройства. Часто разрабатывается вместе с самим устройством.
- Совокупность драйверов физических устройств составляет уровень управления физическими устройствами.
- Стандартизует правила, по которым возможно внесение в систему новых драйверов устройств.
- В системе для одного и того же физического устройства возможно наличие нескольких различных драйверов, которые имеют различные пользовательские интерфейсы, а также предоставляют различные возможности.

Плюсы:

- + упростило процесс адаптации программы для работы с различными типами и разновидностями устройств,
- + повысило надежность программирования
- + снизило уровень требований к программисту о знании специфики управления конкретными устройствами.

Минусы:

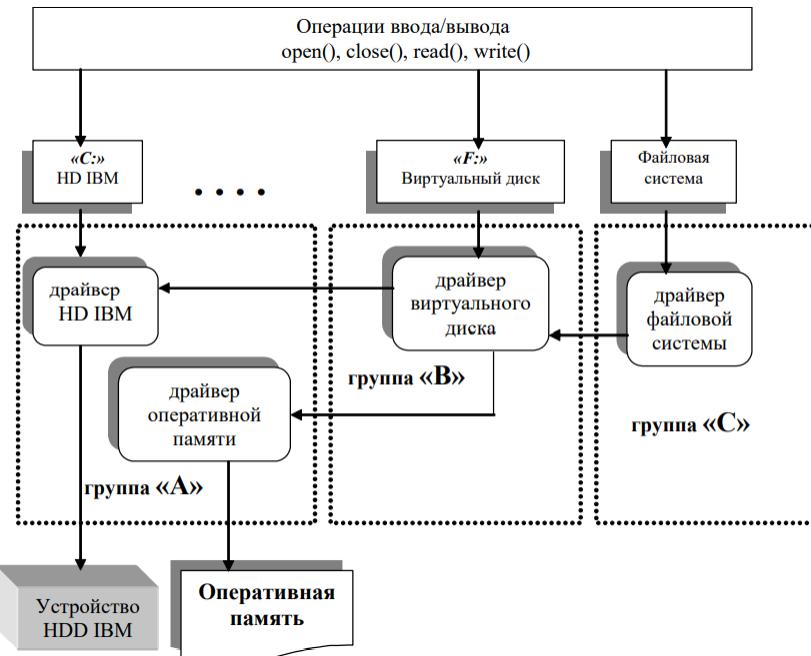
- программист должен быть «знаком» со всеми интерфейсами драйверов используемых физических устройств;
- программы пользователей, использующие конкретные драйверы физических устройств, должны модифицироваться каждый раз, когда возникает необходимость использовать другие физические устройства данного типа (это работа проще той, которая выполнялась, когда внешнее устройство непосредственно програмировалось в программе пользователя, но в программу необходимо внести изменения, позволяющие использовать другой драйвер с другими интерфейсами).

## 3. Управление логическими/виртуальными ресурсами.

- В основу легло обобщение особенностей физических устройств одного вида и создание драйверов, имеющих единые интерфейсы, посредством которых осуществляется доступ к различным физическим устройствам одного типа => в ВС возможность программного создания и использования логических, или виртуальных, ресурсов. **Логическое/виртуальное устройство (ресурс)** — это устройство/ресурс, некоторые эксплуатационные характеристики которого (возможно все) реализованы программным образом.

Виртуальный диск (вирт ресурс с унифицированным хар-ками дисковых устройств.) – надстройка, которая предоставляет на пользовательский уровень унифицированный интерфейс работы с диском.

- Современные ВС позволяют создавать разнообразные логические/виртуальные устройства и соответствующие им драйверы. **Драйвер логического/виртуального ресурса** — это программа, обеспечивающая существование и использование соответствующего ресурса. Для этих целей при его реализации возможно использование существующих драйверов физических и виртуальных устройств. Внутри драйвера виртуального диска прога разбирается, к какому реальному физическому устройству. Внутри драйвера виртуального диска прога разбирается, к какому реальному физическому устройству. Для этих целей при его реализации возможно использование существующих драйверов физических и виртуальных устройств. Внутри драйвера виртуального диска прога разбирается, к какому реальному физическому устройству.
- Возможно построение иерархии логических устройств.



А. Драйверы физических устройств — обеспечивают доступ к конкретным физическим устройствам. Например, драйвер жесткого диска фирмы IBM модели Deskstar.

Б. Драйверы виртуальных устройств определенного типа (например, драйвер виртуального диска), предоставляют обобщенные интерфейсы доступа к разнообразным физ устройствам данного типа. Имеют связи с драйверами конкретных физических устройств данного типа. Запрос к данному драйверу виртуального устройства обычно транслируется драйверу конкретного физического устройства и управляющие команды получит само устройство. Возможна «реализация» виртуального устройства определенного типа на устройствах других

типов, например, возможна организация работы с виртуальным диском, реализованном на пространстве ОП => драйвер виртуального устройства имеет связь с драйверами физ. устройств других типов.

С. Драйверы виртуальных устройств, которым затруднительно поставить в соответствие физическое устройство или группу физических устройств определенного типа - драйверы файловых систем (**файловая система** — программный компонент вычислительной системы, обеспечивающий именованное хранение и доступ к данным. Файл – нет прямого аппаратного аналога, пользователь не знает расположение файла в компе, тип компа и тд). Результат появления уровня управления виртуальными устройствами ВС - многоуровневая унификация интерфейсов доступа к ресурсам ВС, что существенно упростило проблему программирование устройств компьютера, а также предоставило качественно новые возможности в функционировании ВС и в создании их программного обеспечения. Драйвер файловой системы преобразует запрос к драйверу виртуального диска, который осуществляет трансляцию запроса к драйверу физ ус-в.

- Средства программирования, доступные на уровнях управления ресурсами ВС: система команд компа, программные интерфейсы драйверов ус-в (физ и вирт).

Ресурсы ВС – совокупность всех физ вирт ус-в.

ОС – комплекс программ, обеспечивающий управление ресурсами ВС.

- Одна из характеристик ресурсов - конечность.
- у вирт ресурса файловой системы - размер ФС на устройствах хранения данных, ограничения на предельное количество зарегистрированных в файловой системе файлов.

### **3. Структура вычислительной системы. Ресурсы ВС - физические, виртуальные. Уровень систем программирования.**

Традиционное, неформальное определение этапов жизненного цикла программы. Существуют международные стандарты, формализуют понимание жизненного цикла программы ("Information Technology – Software Life Cycle Processes").

1. Проектирование программной системы. Часто итерационный процесс, в котором возможны неоднократные возвраты к тем или иным шагам.

- Формализация и уточнение постановки задач. Исследование решаемой задачи, формирование концептуальных требований к разрабатываемой программной системе.
- Определение характеристик объектной ВС — характеристик аппаратных и программных компонентов ВС, в рамках которой будет работать создаваемая программная система.
- Построение моделей функционирования автоматизируемого объекта.

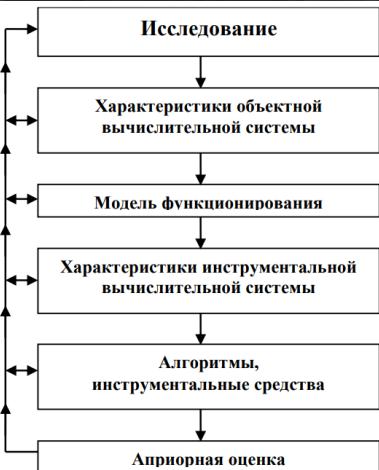


Рис. 7. Этапы проектирования.



**Рис. 8. Кодирование.**

может представляться в виде совокупности исходных модулей программы, объектных или библиотечных модулей, модулей исполняемого кода разрабатываемой программной системы.

Средства управления разработкой программных продуктов позволяют организовать эффективную коллективную работу над реализацией программного проекта, включают в себя:

- средства автоматизации контроля использования межмодульных интерфейсов, которые обеспечивают контроль правильности использования в программе спецификаций, регламентирующих межмодульные связи (кол-во, тип, права доступа к параметрам, обеспечивающим межмодульной взаимодействие в программе);
  - средства автоматизации получения объектных и исполняемых модулей программы, обеспечивающие автоматический контроль за соответствием исходных модулей объектным и исполняемым модулям (так, если в проекте появилась новая редакция некоторого исходного модуля, то при запуске этого средства автоматически произойдет последовательность действий, обновляющих объектные и исполняемые модули, зависящие от данного исходного модуля);

– Определение  
характеристик инструментальной  
ВС, которая будет использоваться  
при создании программной  
системы. Зачастую характеристики  
объектной и инструментальной ВС  
совпадают: тип ВС, на которых в  
дальнейшем будет работать  
программная система, совпадает с  
типов ВС, которая использовалась  
при разработке, в общем случае  
это не так.

– Выбор основных алгоритмов, инструментальных средств, методов (ресурсоемкие, влияют на платформу), которые будут использованы при

программировании, а также разработка архитектуры программного решения, включающей разбиение программного решения на основные модули и определение информационных связей между модулями системы, а также правила взаимодействия с объектной ВС.

– Априорная оценка ожидаемых результатов. Один из важнейших шагов проектирования программной системы, заключающийся в предварительной оценке характеристик проектируемого решения до начала его практической реализации, используются различные методы моделирования => повысить качество программного продукта, который будет создан на основании результатов этапа проектирования, а также сократить затраты на его создание.

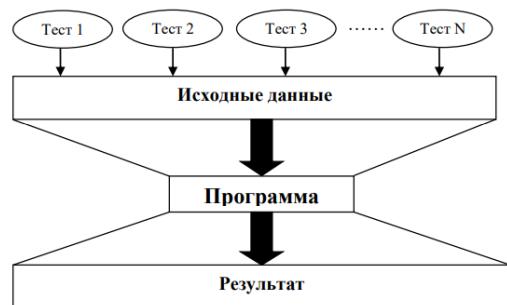
- Определение ее функционирования системы, по которым для входных данных формируются выходные данные (или результаты).

2. Кодирование (программная реализация). Этап построения кода программной системы на основании спецификаций, полученных при ее проектировании. Используются инструментальные средства: прог-я:

- трансляторы языков программирования, средства поддержки и использования библиотек программ, формирования модулей, которые могут исполняться в ВС;
- средства управления разработкой программных продуктов коллективом разработчиков.

Результат - реализация программной системы, которая

— **система поддержки версий** — система, позволяющая фиксировать состояние разработки прогресса проекта (создание версии проекта) и, при необходимости, возвращаться в разработке к той или иной версии проекта.



3. Этап тестирования и отладки программной системы. **Тестирование программы** — процесс проверки правильности функционирования программы на заранее определенных наборах входных данных — тестах, или тестовых нагрузках.

- выявляется работоспособность программы на данном teste (или на наборе тестов) или имеющаяся в программе ошибка.
- задачей в тестировании является решение проблемы формирования минимального набора тестов или тестовых нагрузок, наиболее полно проверяющих функциональность программы (**тестовое покрытие**).

- **Отладка** — это поиск, локализация и исправление зафиксированных

при тестировании или в процессе эксплуатации ошибок. Для обеспечения процесса отладки используются специальные программные средства — отладчики. Средства отладки зависят от типа и назначения создаваемой программной системы.

4. Этап ввода программной системы в эксплуатацию (внедрение) и сопровождения.

- Основное требование к программному продукту - возможность эксплуатации соответствующей программной системы без постоянного участия разработчика программы. Достигается надежностью программы (для этого программа должна быть максимально полно протестирована и устойчива к всевозможным комбинациям входных данных) и наличием подробной и адекватной программе документации, необходимой для всех категорий пользователей данной программной системы (пользователь, системный программист, администратор, оператор и т.п.).

**Сpirальная модель** основана на том, что процесс разработки программной системы складывается из последовательности "спиралей", каждая из которых включает этапы проектирования, кодирования, тестирования и получения результата. Результат — очередная детализация проекта и получение последовательности программ — прототипов. **Прототип** — программа, реализующая частичную функциональность и внешние интерфейсы разрабатываемой системы. Последовательность прототипов сходится к реализации программной системы. А детализации проекта превращаются в полный проект системы.

На каждом из шагов из жизненного цикла (где возможно) искать решения, позволяющий автоматизировать соотв действия и осуществл переход от 1 этапа жизненного цикла к другому «непрерывный».

**Система программирования** — комплекс программ, обеспечивающий поддержание этапов жизненного цикла программы в ВС. Этапы жизненного цикла программы не изменялись в течении времени, определение системы программирования изменялось постоянно вместе с появлением и развитием данных средств.

## Системы программирования: история

Начало 50-х годов XX века	Система программирования или система автоматизации программирования включала в себя ассемблер (или автокод) и загрузчик, появление библиотек стандартных программ и макрогенераторов.	- Основная функция первых систем программирования — предоставление программисту системы мнемонического обозначения компьютерных команд и данных, используемых в программах, а также предоставление возможности создавать и использовать библиотеки программ.
Середина 50-х – начало 60-х годов XX века	Появление и распространение языков программирования высокого уровня (Фортран, Алгол-60, Кобол и др.). Формирование концепций модульного программирования.	- Середина 50-х – начало 60-х годов XX века. Система программирования: макроассемблеры, трансляторы языков высокого уровня, редакторы внешних связей, загрузчики.
Середина 60-х годов – начало 90-х XX века	Развитие интерактивных и персональных систем, появление и развитие языков объектно-ориентированного программирования.	- Середина 60-х – начало 90-х годов XX века. Система программирования: трансляторы языков программирования, редакторы внешних связей, загрузчики.
90-е ХХ века – настоящее время	Появление промышленных средств автоматизации проектирования программного обеспечения, CASE-средств ( <i>Computer-Aided Software/System Engineering</i> ), унифицированного языка моделирования UML.	- Середина 60-х – начало 90-х годов ХХ века. Система программирования: трансляторы языков программирования, детализация проектирование тестирование отладка

кодирование прототипы программная система 22 редакторы внешних связей, загрузчики, средства поддержания библиотек программ, интерактивные и пакетные средства отладки программ, системы контроля версий, средства поддержки проектов.

- 90-е годы XX века — настоящее время. CASE системы — система, в которых делается попытка представления движения по жизненному циклу как единого целого. Системы программирования: интегрированные системы, предоставляющие комплексные решения в автоматизации проектирования, кодирования, тестирования, отладки и сопровождения программного обеспечения.

Функции конкретной СП определяются составом программных компонентов, которые могут использоваться для поддержания этапов жизненного цикла программы, и степенью интеграции этих компонентов.

Выбор конкретной СП во многом зависит как от масштабности и сложности решаемой задачи автоматизации, так и от квалификации программистов.

уровень СП основывается на доступе к виртуальным и физическим ресурсам, предоставляемым ОС (или уровнями управления физическими и виртуальными ресурсами), и предоставляет программистам инструментальные средства разработки программных систем, каждая из которых предназначена для решения своего круга задач.

---

#### **4. Структура ВС. Ресурсы ВС - физические, виртуальные. Уровень прикладных системы.**

**Прикладная система** — это программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области. РЕАЛИЗУЮТСЯ СРЕДСТВА ФОРМАЛИЗАЦИИ ВЗАИМОД С ПРОГР СИСТЕМАМИ, ИСПОЛЬЗУЕМЫМИ ДЛЯ РЕШЕНИЯ КОНКРЕТНЫХ ПРИКЛАДНЫХ ЗАДАЧ. Прагматическая основа всей ВС. История развития:

1. 1-е поколения: для автоматизации решения каждой конкретной задачи создавалась уникальная программная система, которая не предполагала возможность модификации функциональности, переноса с одной ВС на другую. Пользовательского интерфейса не было, как такового. Подавляющее большинство решаемых прикладных задач было связано с моделированием физ процессов, и, в свою очередь, результаты моделирования представлялись в виде последовательностей чисел и числовых таблиц. Уровень инструментальных средств программирования, доступных для решения прикладных задач, накладывал достаточно жесткие требования к квалификации специалистов, занимающихся автоматизацией решения прикладных задач.

2. 2-е пок — развитие систем программирования и появление средств создания и использования библиотек программ. Библиотеки прикладных программ => аккумулировать и многократно использовать практический опыт численного решения типовых задач из конкретных предметных областей. Составляющие библиотеку подпрограммы служили "строительными блоками", которые в интеграции с системами программирования использовались для разработки прикладных систем. Библиотеки — одни из первых программных систем, которые могли относиться к категории программных продуктов — документированных, прошедших детальное тестирование, распространенное в пользовательской среде, первые коммерческие программные продукты. Прикладные системы этого этапа создавались с использованием стандартных систем программирования и в большей части были уникальны: создавались для решения конкретной задачи в конкретных условиях.

3. Появление пакетов прикладных программ (ППП), которые включали в себя программные продукты для решения широкого комплекса задач из конкретной прикладной области (Microsoft Office) и обладающие свойствами:

- программные продукты имели развитые, стандартизованные пользовательские интерфейсы, не требующие высокой программистской квалификации от прикладного пользователя и значительных затрат на их освоение;
- функциональные возможности прикладных программ и их пользовательские интерфейсы позволяли решать разнообразные задачи данной прикладной области;
- возможно совместное использование программных продуктов, входящих в состав ППП при решении конкретных задач.

4. Современный этап — это этап комплексных, адаптируемых к конкретным условиям программных систем автоматизации прикладных процессов, построенных на основе развития концепций пакетов прикладных, интегрированных с современными системами программирования и использующих передовые технологии

проектирования и разработки программного обеспечения. Особое развитие получили системы автоматизации бизнес-процессов.

#### **Основные тенденции в развитии современных прикладных систем.**

1. 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 и, в свою очередь, появились прикладные системы, ориентированные на их автоматизацию.

- a. B2B-система (business to business), поддержка модели межкорпоративной торговли продукцией с использованием Internet (примером может служить электронные биржи);
- b. B2C-система (business to customer), обеспечивающая поддержку в Internet модели торговых отношений между предприятием и частным лицом — потребителем (примером может служить Интернетмагазин);
- c. ERP (Enterprise Resource Planning) — планирование ресурсов в масштабе предприятия, автоматизированная система управлением предприятием;
- d. CRM (Customer Relationship Management) — система управления взаимоотношениями с клиентами.

2. Открытость системы: потребителю системы открыты прикладные интерфейсы, обеспечивающие основную функциональность системы, а также стандарты организации внутренних данных. Прикладные интерфейсы (API — Application Programming Interface) совместно со стандартными средствами систем программирования, системы шаблонов и специализированные средства настройки прикладной системы позволяют адаптировать и развивать функциональные возможности прикладных систем к особенностям конкретного потребителя системы.

3. Использование современных технологий и моделей организации системы: Internet/Intranet-технологии, средства и методы объектно-ориентированного программирования (ООП), модель клиент/сервер, технологии организации хранилищ данных и аналитической обработки данных с целью выявления закономерностей и прогнозирования решений, и др.

Современная прикладная система предполагает глубокую интеграцию всех компонентов ВС => возможно разделение пользователей прикладной системы на категории:

- оператор или прикладной пользователь, оперируя средствами пользовательского интерфейса и функциональными возможностями системы, решает конкретные прикладные задачи.
- 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, доработку функциональных возможностей системы, удовлетворяющих потребностям и особенности эксплуатации.
- системный администратор обеспечивает выполнение текущих работ по поддержке функционирования программной системы в конкретных условиях: в их состав могут входить регистрация пользователей и распределение полномочий и прав между ними, контроль за обеспечение сохранности и целостности данных, фиксация проблем, возникающих в процессе эксплуатации, и обоснованное выполнение обновлений системы, поступающих от разработчика.

---

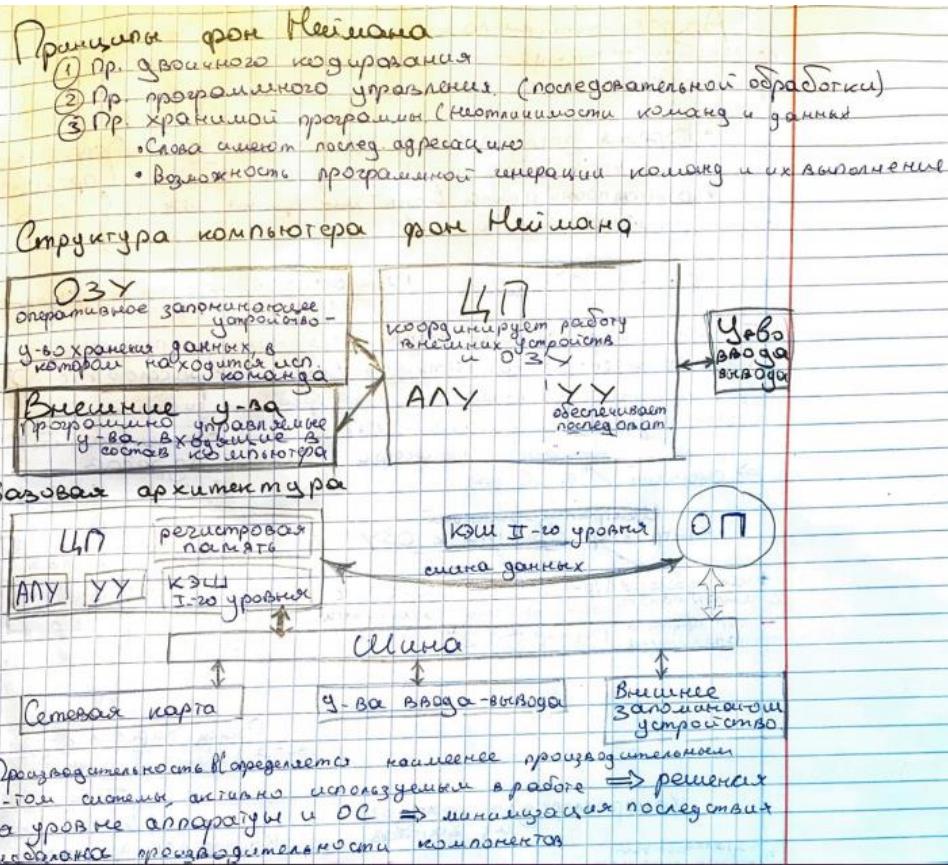
#### **5. Структура вычислительной системы. Понятие виртуальной машины.**

**Виртуальная машина** - программно реализованные аппаратные средства. Спецификация некоторой вычислительной среды

Не менее 90% современных пользователей персональных компьютеров не имеют представления о системе команд компьютера, о структуре компьютерных данных, об аппаратных интерфейсах управления физическими устройствами — все это скрывают расширения компьютера, которые образуются за счет соответствующих уровней вычислительной системы. Мы будем говорить, что каждый пользователь, работая в соответствующем расширении компьютера, работает в виртуальной машине или виртуальном компьютере. Реальный компьютер используется непосредственно исключительно на аппаратном уровне. Во всех остальных случаях пользователь работает с

программным расширением возможностей реального компьютера — с виртуальным компьютером. Причем "виртуальность" этого компьютера (или этих компьютеров) возрастает от уровня управления физическими ресурсами ВС до уровня прикладных систем.

## 6. Основы архитектуры компьютера. Основные компоненты и характеристики. Структура и функционирование ЦП.



### Принципы фон Неймана.

1. **Принцип двоичного кодирования информации.**

2. **Принцип программного управления (принцип последовательного выполнения команд).** Программа состоит из команд, в которых закодированы операция и операнды, над которыми должна выполниться данная операция. Выполнение компьютером программы — это автоматическое выполнение определенной последовательности команд, составляющих программу. В компьютере имеется устройство, обеспечивающее выполнение команд, — процессор.

Последовательность выполняемых процессором команд определяется последовательностью команд и данных, составляющих программу.

### 3. Принцип хранимой программы.

Для хранения команд и данных программы используется единое устройство памяти, которое представляется в виде вектора слов. Все слова имеют последовательную адресацию. Команды и данные представляются единым образом. Интерпретация информации памяти и, соответственно, ее идентификация как команды или как данных происходит неявно при выполнении очередной команды. То есть одна и та же область памяти в зависимости от команд в одном случае будет интерпретироваться как команда, в другом случае — как данные => возможность программной генерации команд с последующим их выполнением.

- **Оперативное запоминающее устройство (ОЗУ),** или основная память, — устройство хранения данных, в котором находится исполняемая в данный момент программа => ОП — свойство всё-таки системное, а не технологическое (т.е. на основе чего сделано это устройство — на той же элементной базе можно сделать устройство для хранения информации, но оно будет представляться системе не как ОП, а как внешнее устройство; тогда как программа будет исполняться из ОП).
- **Внешние устройства** — программируемые устройства, входящие в состав компьютера, т.е. устройства, с которыми выполняемая программа может осуществлять обмен данными.
- **Процессор, или центральный процессор (ЦП),** — основной компонент компьютера, обеспечивающий выполнение программ, процессор координирует работу внешних устройств и ОП. Процессор состоит из арифметико-логического устройства (АЛУ) и устройства управления (УУ). Устройство управления обеспечивает последовательную выборку команд, составляющих программу, из памяти, выделение и анализ кода операции, получение значений operandов. В зависимости от кода операции команда выполняется либо в устройстве управления (обычно это могут быть команды передачи управления), либо код операции и operandы передаются для выполнения в АЛУ. После чего выбирается из памяти следующая команда программы и т.д. В системе команд процессора предусмотрены средства для взаимодействия с внешними

устройствами. В АЛУ осуществляется реализация команд, предполагающая арифметическую/логическую обработку содержимого операндов.

Современные компьютеры по многим показателям не соответствуют принципам фон Неймана.

1. Существует комп в троичной сс.
2. Большинство компьютеров начинает обрабатывать команды «с забеганием» вперёд, то есть во время выполнения текущей команды последующие команды уже начинают выбираться (иногда эта работа может пойти наスマрку, например, в случае ветвления по условию).
3. В большинстве компьютеров ОЗУ хранит команды и данные «по-разному» для предотвращения ошибки потери управления.
4. Компы параллельные, команды выполняются параллельно.

В ОП находится исполняемая программа (т.е. все команды и данные последовательно выбираются из ОП). ЦП обеспечивает выборку, анализ и исполнение команд. Скорость обработки информации в процессоре, скорость доступа к данным, размещенным в оперативной памяти, и скорость обмена данными с внешними устройствами могут отличаться друг от друга => в системе не будут предусмотрены средства, компенсирующие этот дисбаланс => итоговая производительность будет определяться наименее производительным элементом, активно используемым в работе системы.

**1.2.2 Оперативное запоминающее устройство (RAM — Random-Access Memory, память с произвольным доступом)** — это устройство для хранения данных, в котором находится исполняемая программа. ОЗУ еще называют основной памятью, или ОП. Команды программы, исполняемые компьютером, поступают в процессор исключительно из ОЗУ.

Основным назначением ОП является хранение программы, которая выполняется в настоящее время компьютером. Оперативная память структурно состоит из ячеек памяти. **Ячейка памяти** — это устройство, в котором размещается информация. Ячейка памяти имеет предопределенное количество разрядов, имеет уникальное имя (адрес ячейки памяти, обычно - нумерация), может состоять из двух полей (2-е необяз.):

1. Первое поле — поле машинного слова. **Машинное слово** — поле программно изменяемой информации. В машинном слове могут располагаться машинные команды (или части машинных команд) или данные, с которыми может оперировать программа. Машинное слово имеет фиксированный для данной ЭВМ размер. Обычно под размером машинного слова понимается количество двоичных разрядов, размещаемых в машинном слове, кратна 8 двоичным разрядам.

2. Второе — тег. **Поле служебной информации — ТЭГ** (tag — ярлык, бирка) — 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).**

— Контроль за целостностью данных. Простейшая модель — это одноразрядный ТЭГ, который использовался для контроля чётности. Содержимое поля используется для контрольного суммирования кода, размещенного в машинном слове. При каждой записи информации в машинное слово автоматически происходит контрольное суммирование (количество единичек в записываемом коде) и формирование содержимого поля служебной информации (бит чётности или нечётности). При чтении данных из машинного слова также автоматически происходит контрольное суммирование кода, находящегося в машинном слове, а затем полученный код контрольной суммы сравнивается с кодом, размещенным в поле служебной информации. Совпадение кодов говорит о том, что данные, записанные в машинном слове, не потеряны. Несовпадение говорит о том, что произошел сбой в ОЗУ и информация, находящаяся в машинном слове, потеряна, в этом случае в процессоре происходит прерывание (прерывания будут рассматриваться несколько позднее).

— Контроль доступа к командам/данным. В машинах фон Неймана: 1) ситуация "потери" управления в программе (из-за ошибок в программе в качестве исполняемых команд начинают выбираться процессором и исполняться данные), 2) программа из-за ошибки сама затирает свою кодовую часть: на место команд записываются данные. Отладка трудоемка => контроль доступа к командам/данным обеспечивает защиту от возникновения подобных проблем. Когда мы размещаем программу в памяти и генерируем код, то мы сразу же «раскрашиваем» всю информацию на два цвета. При включении специального режима работы

процессора запись машинных команд в ОП сопровождается установкой в ТЕГе специального кода, указывающего, что в данном машинном слове размещена команда. Также соответствующий признак устанавливается при записи данных. При выборке очередной команды из памяти автоматически проверяется содержимое соответствующих разрядов ТЕГа => введение контроля за семантикой информации, размещенной в машинном слове для минимизации возможных ошибки в программе пользователя.

— Контроль доступа к машинным типам данных. Модель доступа к машинным типам данных (существуют группы машинных команд, которые оперируют с данными одного типа (целые, вещественные с фиксированной точкой, вещественные с плавающей точкой, символьные, логические)) => при выполнении команды используемые операнды интерпретируются согласно машинному типу данных в соответствии с типом команды. Контроль доступа осущ за счет фиксации в поле ТЕГа кода типа данных при их записи в машинное слово, а при использовании этих данных в качестве операндов команд осуществляется автоматическая проверка совпадения типа операнда и типа команды.

В ОЗУ все ячейки памяти имеют уникальные имена, имя — **адрес ячейки памяти**. Обычно адрес — это порядковый номер ячейки памяти (нумерация ячеек памяти возможна как подряд идущими номерами, так и номерами, кратными некоторому значению). Доступ к содержимому машинного слова осуществляется при непосредственном (например, считать содержимое слова с адресом А) или косвенном использовании адреса (например, считать значение слова, адрес которого находится в машинном слове с адресом В).

Проблема дисбаланса производительности процессора и доступа к ОП. 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 (решается на уровне архитектуры ЭВМ).

Производительность ОЗУ (определяет скорость доступа процессора к данным, размещенным в ОЗУ) определяется по параметрам:

- Первый — время доступа (access time — taccess) — это время между запросом на чтение слова из оперативной памяти и получением содержимого этого слова.
- Второй параметр — длительность цикла памяти (cycle time — tcycle) — это минимальное время между началом текущего и последующего обращения к памяти. Обычно, длительность цикла памяти существенно превосходит время доступа ( $tcycle > taccess$ ): устройства памяти устроены таким образом, что после чтения многие из устройств памяти требуют регенерации (т.е. при чтении информация из ячейки разрушается — для того, чтобы сохранить информацию, надо её записать). Реальные соотношения между длительностью цикла и временем доступа зависят от конкретных технологий, применяемых для организации ОЗУ (в некоторых ОЗУ  $tcycle/taccess > 2$ ) => возможна ситуация, при которой для чтения N слов из памяти потребуется времени больше, чем  $N \times taccess$ .

---

## **7. Основы архитектуры компьютера. Основные компоненты и характеристики. Оперативное запоминающее устройство. Расслоение памяти.**

$t_{cycle} > t_{access}$

После цикла инфы в ячейке разрушается  
=> нужна регенерация

=> Итоговая скорость выполнения команды производством должна меньше зависеть от  $t_{access}$  => расслоение ОЗУ.  
один из аппаратных путей решения проблемы дисбаланса в производительности доступа к данным в ОП и производительности процессора => скорость  $\uparrow$  в K раз

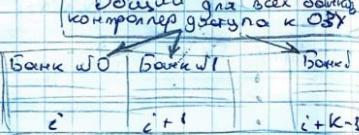
централизованный контроллер доступа к ОЗУ

+ нет проблем с циклом памяти под устройством памяти  
- нет эффекта разрыв  $K = 2^L$  при параллелизме

последовательный адрес

адресов

$i, i+1, \dots, i+K-1$

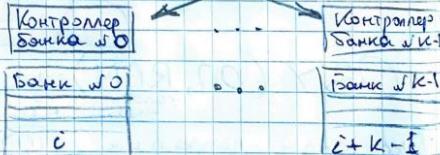


послед. адресов  $i, \dots, i+K-1$   
виде адрес номер банка

L разрядов

Контроллер доступа к ОЗУ

+ параллельный доступ к памяти до K слов



памяти, т.к. при расслоении ОЗУ задержки, связанные с циклом памяти, будут возникать только в тех случаях, когда подряд идущие обращения попадают в один и тот же банк памяти => увеличения скорости чтения из памяти при последовательном доступе, в идеальном случае в K раз.

#### Модели расслоения памяти:

1. Модель с общим централизованным контроллером доступа к памяти – одно ус-во управляет всеми банками => при последовательном обращении пока в банке А инфа регенерируется можно обращаться в другие банки => нет проблемы цикла памяти, т.к. соседние ячейки памяти находятся в разных банках; но нет эффекта при параллелизме.
2. Каждый банк имеет свой контроллер => параллельная обработка запроса.

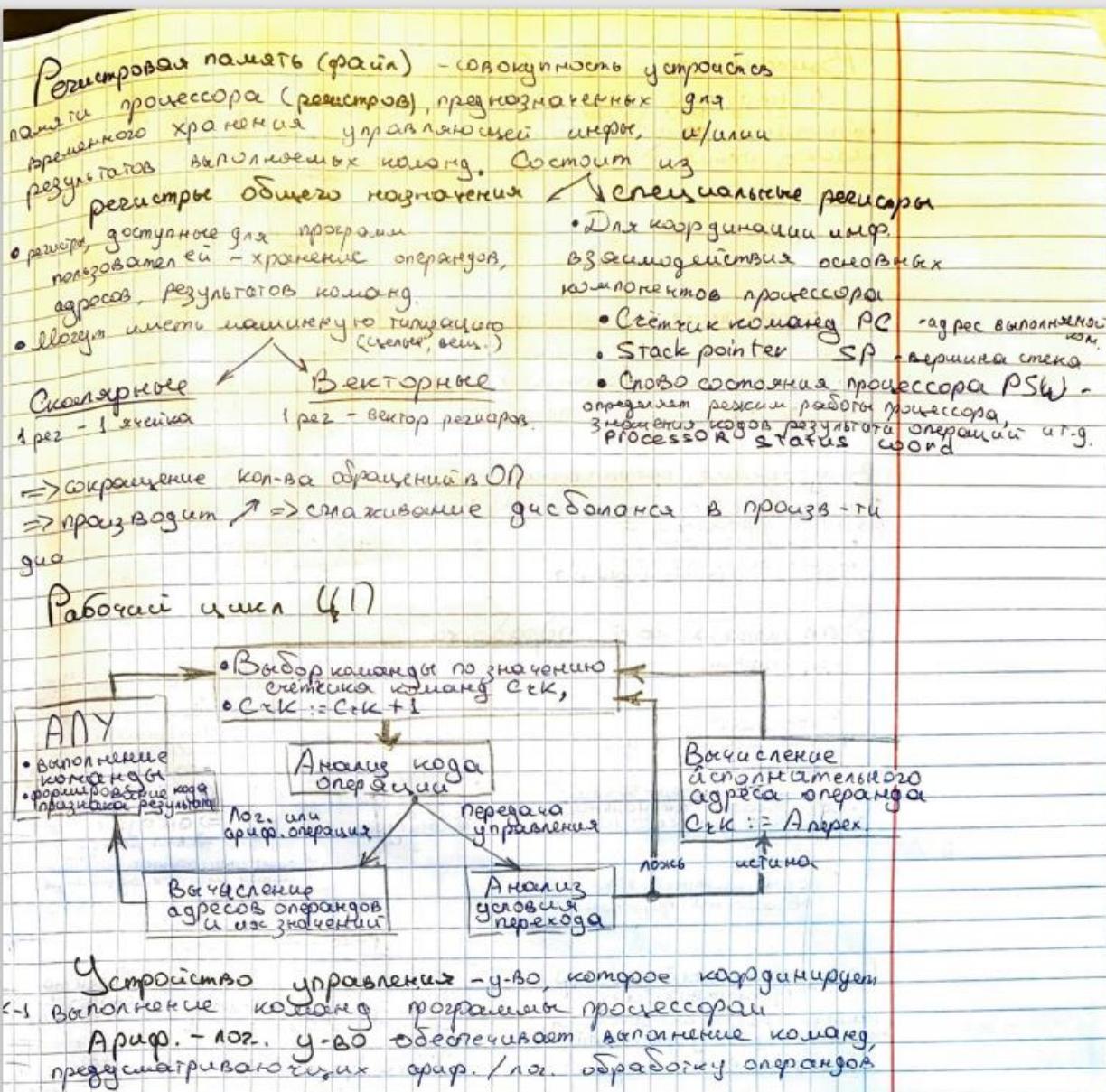
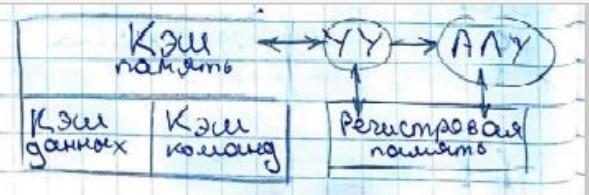
**Расслоение ОЗУ** – один из аппаратных путей решения проблемы дисбаланса в скорости доступа к данным, размещенным в ОП, и производительностью процессора.

- Всё пространство ОЗУ делится на K ( $K = 2^L$ ) независимых подустройств, которые называются банками памяти. Адресация в системе организована таким образом, что младшие L разрядов адреса содержат номер банка. Подряд идущие ячейки памяти распределены между банками таким образом, что у любой ячейки ее соседи размещаются в соседних банках (т.е. они находятся в разных банках).

- Каждый банк может работать независимо.

=> сократить задержки из-за несоответствия времени доступа и цикла памяти при выполнении последовательного доступа к ячейкам

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



**Регистровая память, или регистровый файл (register file)**, — совокупность устройств памяти процессора (так называемых регистров), предназначенных для временного хранения управляемой информации, operandов и/или результатов выполняемых команд. Регистровая память обычно включает в себя:

1. **Регистры общего назначения (РОН)** состоят из доступных для программ пользователей регистров, предназначенных для хранения operandов, адресов operandов, результатов выполнения команд.
  - РОН могут иметь машинную типизацию (например, регистры для хранения данных с плавающей точкой, с фиксированной точкой и т.д.).
  - РОН могут быть скалярными (когда с одним регистром ассоциируется только одна единица памяти) и векторными (например, с одним регистром может ассоциироваться вектор регистров из 64 элементов; примером классических векторных компьютеров являются компьютеры фирмы CRAY).
  - Могут быть регистры, связанные с внешними устройствами, через которые передается команда во внешние устройства.

- Назначение: Регистровая память работает в темпе процессора, т.е. скорость доступа к содержимому регистров сравнима со скоростью обработки информации процессором => сглаживание дисбаланса в производительности процессора и скорости доступа к ОП. РОН были первым аппаратным средством, которое предоставлялось пользователю для оптимизации своей программы. Наиболее часто используемые в программе операнды размещались на РОН => ↗ количества реальных обращений в ОП => ↗ суммарную производительность компьютера. Состав РОН зависит от архитектуры конкретного компьютера.

**2. Специальные регистры** предназначены для координации информационного взаимодействия основных компонентов процессора. Состав определяется архитектурой. В их составе могут быть:

- 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:

- (program counter) **Счетчик команд** — специальный регистр, в котором размещается адрес очередной выполняемой команды программы, изменяется в устройстве управления согласно алгоритму, заложенному в программу.
- (stack pointer) **Указатель стека** — регистр, содержимое которого в каждый момент времени указывает на адрес слова в области памяти, являющегося вершиной стека. Обычно данный регистр присутствует в процессорах, система команд которых поддерживает работу со стеком (операции чтения и записи данных из/в стек с автоматической коррекцией значения указателя стека).
- (processor status word). **Слово состояния процессора** — регистр, содержимое которого определяет режимы работы процессора, значения кодов результата операций и т.п.

**Устройство управления (control unit)** — устройство, которое координирует выполнение команд программы процессором.

**Арифметико-логическое устройство (arithmetic/logic unit)** обеспечивает выполнение команд, предусматривающих арифметическую или логическую обработку операндов.

Упрощенный алгоритм выполнения процессором программы (рабочий цикл ЦП).

Есть комп с ОП, пословная адресация: каждый адрес указывает на машинное слово, адрес следующего отличается от предыдущего на 1. Пусть есть счетчик команд. Действие ЦП:

1. Выборка машинного слова, которое считается, как команда, который считается из счетчика команд.
2. Помещается на специальный регистр процессора.
3. Счетчик команд автоматически увеличивается на 1.
4. Устройство упр., анализ кода операции, которые мы считали.

- Некорректный код/ несуществующая/ некорректная команда => Ошибка.

- Ариф/лог операция => вычисление исполнительных адресов операндов, их значения перемещаются регистры, команда передается для исполнения в АЛУ. При работе АЛУ мб фиксируется ошибка. Если все норм => возвращаемся в начальный этап.

- Команда передачи управления => анализ условия перехода. Если не вып => возвращаемся обратно.

Если вып => вычисляем исп адрес операнда, присваиваем счетчику команд этот адрес.

---

## 8. Основы архитектуры компьютера. Основные компоненты и характеристики. Кэширование ОЗУ.

Проблема несоответствия производительности ЦП и скорости доступа к информации, размещенной в ОП.

Самое эффективное решение основывается на аппаратных средствах, позволяющих при выполнении программы автоматически минимизировать количество реальных обращений в ОП за operandами и командами программы за счет кэширования памяти — размещения части данных в более высокоскоростном запоминающем устройстве. **КЭШ-память (cache memory)** — высокоскоростное устройство хранения данных, используемое для буферизации работы процессора с ОП. В общем случае, кэш - специализированная память, в которой аккумулируются наиболее часто используемые данные из ОП, размещенная в процессоре, которая организована как последовательность блоков фиксированного размера (степень 2).

- Скорость доступа к информации, размещенной в КЭШе, соизмерима со скоростью обработки информации в ЦП. Регистры и кэш 1-го уровня работают в темпе процессора, тк находятся вместе и работают на общих

технологиях.

- Обмен данными при выполнении программы (чтение команд, чтение значений операндов, запись результатов) происходит не с ячейками ОП, 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 ОП. Размещение и команд, и данных в одном КЭШе может приводить к тому, что команды и данные начинают вытеснять друг друга, увеличивая при этом обращения к ОП.
- Для исключения недетерминированной конкуренции в КЭШе между командами программы и обрабатываемыми данными => два независимых КЭШа: КЭШ данных и КЭШ команд, каждый из которых работает со своим потоком информации — потоком команд и потоком операндов.
- КЭШ НЕ БУДЕТ РАБОТАТЬ, если поток адресов, который формируется для обращения в память как для получения команд, так и операндов, случайный => слишком много процессов вытеснения.
- КЭШ БУДЕТ РАБОТАТЬ, если есть принцип локализации формирования потока запросов команд.

#### Общая схема работы КЭШа следующая.

1. Условно, вся память разделяется на блоки одинакового размера. Обмен данными между КЭШем и ОП осуществляется блоками фиксированного объема. Здесь мы можем видеть возможное проявление преимущества использования памяти с расслоением, так как загрузка блока из ОП. в КЭШ осуществляется с использованием параллелизма работы «расслоенной» ОП.
2. Каждый блок имеет спецификатор доступа – тэг, в котором находится служебная информация, характеризующая данный блок. В тэге может содержаться информация о том, какой области ОП соответствует содержимое данного блока, занят или свободен блок, производились изменения в данном блоке или нет. 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 => после вычисления Аисп 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 (hit)** – в этом случае данные берутся из КЭШа, и обращение в ОП не осуществляется. Если искомых данных нет в КЭШе, то фиксируется **промах (cache miss)**.
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.
  - вытеснение случайным образом.
  - наименее «популярного» блока КЭШа, т.е. блока, к содержимому которого происходило наименьшее число обращений (**LRU — Least-Recently Used**).

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.

1. **сквозное кэширование (write-through caching)**: при выполнении команды записи данных обновление происходит как в КЭШе, так и в ОП => при вытеснении блока из КЭШа происходит только загрузка содержимого нового блока. Данная стратегия оправдана, т.к. статистические исследования показывают, что частота чтения данных превосходит частоту их записи на порядок.

2. **кэширование с обратной связью (write-back caching)**, суть которой заключается в использовании специального тега модификации (**dirty bit**). При выполнении команды записи по адресу, содержимое которого кэшируется в одном из блоков, происходит обновление соответствующей этому адресу информации только в блоке КЭШа, а также установка в блоке тега модификации => при вытеснении блока осуществляется контроль за содержимым тега. Если тег модификации установлен, то содержимое блока перед вытеснением «сбрасывается» в память => ↓ частота выполнения операции записи в память.

Плюсы КЭШ-памяти:

1. Сокращается количество обращений к ОЗУ – обращений, как по выборке команд, так и по выборке операндов.
2. ↓ скорость доступа к памяти в случае использования ОЗУ с «расслоением», так как обмены блоков с памятью будут проходить практически параллельно (когда мы работаем с группой подряд идущих слов).

**3. Кэш + расслоение = максимальный эффект => взаимодействие между памятью и кэшем происходят блоками => обмены с блоками происходят параллельно, сокращаем кол-во промахов (СВОЙСТВО ЛОКАЛИЗАЦИИ) => максимальное развитие скорости.**

Минусы:

1. усложнение логики процессора. Организация и использование КЭШ-памяти развивает рабочий цикл процессора модельного компьютера: при выборке очередных команд, получении операндов команд и записи результатов выполнения команд в ОЗУ добавляются схемы организации использования КЭШ-памяти.
2. Если КЭШ один (т.е. потоки команд и данных приходятся на один КЭШ), то один из потоков может начать «довлечь» над другим, так как характеристики потоков команд и данных разные (поток команд обладает свойством локализации, поток данных этим свойством не обладает) => деление на 2 кэша => повысить производительность системы.

---

## **9. Основы архитектуры компьютера. Аппарат прерываний. Последовательность действий в вычислительной системе при обработке прерываний.**

В первых компьютерах происходила остановка работы компьютера и обработка ситуации, вызвавшей аварийную остановку (АВОСТ). Современные вычислительные системы не могут полностью останавливаться.

Проблема автоматизации обработки предопределённых событий, возникающих в вычислительной системе => аппарат прерываний.

- **Прерыванием** называется событие в компьютере, при возникновении которого в системе предусмотрена предопределенная последовательность действий, включающая стандартную реакцию процессора на прерывание и этап программной обработки прерываний (функция ОС).
- **Состав прерываний** — множество разновидностей событий, на возникновение которых предусмотрена стандартная реакция центрального процессора, — фиксирован и определяется конструктивно при разработке компьютера.
- **Аппарат прерываний** компьютера позволяет организовывать стандартную обработку прерываний, возникающих при функционировании ВС.

1. **Внутренние прерывания** инициируются схемами контроля работы процессора (деление на 0, overflow).
  2. **Внешние прерывания** — события, возникающие в компьютере в результате взаимодействия центрального процессора с внешними устройствами (ввод с клавиатуры).
- Обработка прерывания предполагает две стадии:
1. **аппаратную**, которая включает предопределенную реакцию процессора на возникновение прерывания,
  2. **программную**, которая предполагает выполнение специальной программы обработки прерывания, являющейся частью ОС.



Рис. 29. Схема обработки прерывания.

### Этап аппаратной обработки прерывания.

1. Завершается выполнение текущей команды (кроме случаев, когда прерывание возникает по причине некорректного выполнения команды). АТОМАРНАЯ КОМАНДА.
2. Обработка прерывания => остановку выполнения текущей программы, запуск специальной программы обработки прерывания, а затем, возможно, продолжение выполнения прерванной программы (с прерванного места) => необходимо запомнить актуальное состояние компьютера в момент прихода прерывания (т.к. для обработки прерывания будет работать другая программа – ОС, а, следовательно, актуальное состояние системы изменится) => перечень специальных регистров, которые автоматически будут сохранены процессором (счетчик команд, регистр результатов, регистры, содержащие режимы работы процессора), а также несколько регистров общего назначения, которые могут быть использованы программой

обработки прерываний в начальный момент времени. Буфер для сохранения актуального состояния в системе один => временно запретить запись информации относительно нового прерывания в этот буфер => **режим блокировки прерываний** - запрещается инициализация новых прерываний: возникающие в это время прерывания могут либо игнорироваться, либо откладываться (зависит от конкретной аппаратуры компьютера и типа прерывания). Это единственная гарантия того, что не придет новое прерывание, и при его обработке не потеряются данные, необходимые для продолжения прерванной программы (регистры, режимы, таблицы ЦП). После полного сохранения регистров происходит снятие режима блокировки прерываний, то есть включается стандартный режим работы процессора, при котором возможно появление прерываний.

3. Аппаратное копирование содержимого сохраняемых регистров («малое упрятывание»). Включенный режим блокировки прерывания гарантирует сохранность этих данных до момента завершения предварительной обработки прерывания и выключения блокировки прерываний. Перекачать содержимое всех регистров – долго, поэтому сохраняем минимальный набор.

4. Переход на программный этап обработки прерываний. Аппаратно передаём управление на некоторую фиксированную точку в ОЗУ, в которой предполагается наличие программы обработки прерываний ОС. Аппаратная передача ОС информации о типе прерывания => модели:

- использование специального **регистра прерываний**, каждый разряд которого соответствует конкретному типу прерывания,
- Для расширения числа обрабатываемых прерываний иерархическая модель регистров прерывания: в **главном регистре прерываний** выделяются разряды, отвечающие не только за появление конкретных прерываний, но и разряды, отвечающие за появление прерываний в периферийных регистрах. В данной модели управление передается в операционную систему на адрес входа в программу.
- использование **вектора прерываний**. Предполагается, что по количеству возможных прерываний в ОЗУ выделена группа машинных слов — вектор прерываний. Каждое слово вектора прерываний содержит адрес программы, обрабатывающей данное прерывание. При прерывании после сохранения регистров осуществляется передача прерывания по адресу, соответствующему номеру прерывания.



#### Этап программной обработки прерывания.

1. Управление передано на адрес программы ОС, занимающейся обработкой прерывания, часть ресурсов ЦП, используемых программами, свободна (из-за малого упрятывания).



Рис. 33. Программный этап обработки прерываний.

1. Анализ и предварительная обработка прерывания, идентификация типа прерывания, определяются причины.
  - Если в ОП по этому адресу нет программы, обрабатывающей данное прерывание => синий экран.
  - 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 от таймера для коррекции времени в системе).

- Если прерывание требует использования всех ресурсов ЦП, то переходим к следующему шагу

(обращение к области памяти, которая закрыта для обращения – чтение информации с внешнего носителя).

– Если **прерывание является «фатальным»** для программы, т.е. после этого прерывания продолжить выполнение программы невозможно (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 упаковывание»** - 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. ОС завершает обработку прерывания.

---

## 10. Основы архитектуры компьютера. Внешние устройства. Организация управления и потоков данных при обмене с внешними устройствами.

**Внешние запоминающие устройства (ВЗУ)** предназначены для организации хранения данных и программ, обмен с ВЗУ происходит некоторыми порциями данных - **записями**. Данные на ВЗУ - последовательности записей. Виды ВЗУ по обмену данными:

1. Блочные устройства (блокориентированные), которые допускают выполнение обменов исключительно записями фиксированного размера — блоками (магнитные диски). Размер блоков (физических блоков) определяется аппаратно и может зависеть от конкретной модели и типа устройства.
2. Устройства, аппаратно допускающие обмен записями произвольного размера (магнитные ленты).

ВЗУ по возможностям доступа к хранящимся данным:

1. Устройства, аппаратно допускающие как операции чтения, так и операции записи. Пример - жесткий диск.
2. Устройства, позволяющие выполнять только операции чтения данных, CD-ROM (compact disk read-only memory), DVD-ROM (digital video/versatile disc read-only memory).

ВЗУ по доступу:

1. **Устройства прямого доступа** - обеспечивает выполнение операций чтения/записи без считывания дополнительной (предыдущей) информации. Примером устройств прямого доступа могут служить магнитные диски, или дисковые устройства (магнитные диски, или дисковые устройства).
2. **Устройства последовательного доступа** - это устройства, при доступе к содержимому произвольной записи которых «просматриваются» все записи, предшествующие искомой (магнитные ленты). Считывать долго и неудобно, но надежно и можно хранить много инфы. Нужны для организации, архивирования, долгого хранения инфы.

**Этапы обработки** определяют производительность ВЗУ:

1. Механическая обработка заказа на обмен (прочесть/записать). Перед обменом механические действия. Для жесткого диска – перемещение блока головок на нужный цилиндр, фиксируется блок головок и дожидается, пока диск повернется. Для барабана: барабан прокрутится до позиции, с которой находятся данные, нужная позиция под головками => вкл нужную головку (убрали движение головок => быстрее).
2. этап обмена (большой у магнитной ленты).

На магнитной ленте каждая запись имеет специальные маркеры начала и конца, каждая запись на ленте имеет свой логический номер. При возникновении запроса на чтение записи с номером  $i$  выполняется следующая последовательность действий:

- устройство перематывает ленту до маркера начала ленты;
- осуществляется последовательный поиск маркеров начала записей, после нахождения  $i$ -го маркера считается, что устройство «вышло» на начало искомой записи;
- происходит чтение  $i$ -ой записи. Устройство прямого доступа обеспечивает выполнение операций чтения/записи без считывания дополнительной (предыдущей) информации.

## Магнитный диск:

- вал, вращающийся с достаточно высокой постоянной скоростью.
- на валу закреплены диски, поверхности которых покрыты материалом, способным на основе магнитоэлектрических эффектов сохранять информацию, кол-во варьируется.
- система головок чтения/записи. Кол-во головок = кол-ву поверхностей дисков, и каждая головка может работать со своей фиксированной поверхностью.
- Все головки устройства составляют блок головок магнитного диска. Блок головок может перемещаться от края поверхностей к центру. Перемещение блока головок осуществляется дискретно.
- каждая позиция остановки блока головок над поверхностями (с учетом вращения дисков) образует цилиндр => дисковое устройство характеризуется фиксированным количеством цилиндров, которые соответствуют позициям, на которых может размещаться блок головок. Все цилиндры пронумерованы.
- Условные линии пересечения цилиндров с поверхностями образуют дорожки. Дорожки, относящиеся к одному цилиндру пронумерованы. Дорожки, принадлежащие одной поверхности, формируют концентрические круги.
- Все дорожки разделены на фиксированное (для данного устройства) число равных частей — **секторов**. Секторы каждой дорожки пронумерованы. Начала одноименных секторов лежат в одной плоскости, проходящей через вал.
- Возможность индикации факта прохода блока головок через каждую точку начала сектора => блок головок всегда может «знать», над каким сектором он находится. В каждый момент времени в блоке головок может проходить обмен с одним из секторов.

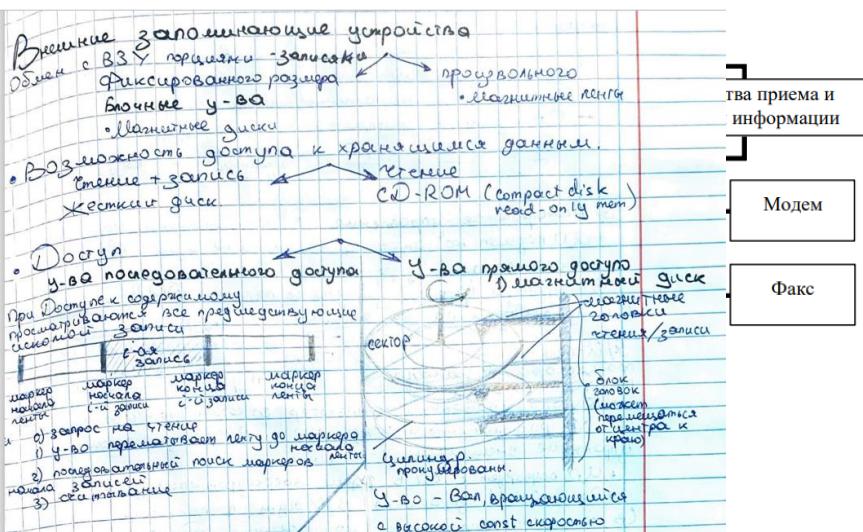


Рис. 34. Иерархия внешних устройств.

- В магнитном барабане** электродвигатель, к оси которого прикреплен массивный барабан, поверхность которого покрыта электромагнитным слоем. Двигатель раскручивает барабан до высокой постоянной скорости.
- Есть фиксированная штанга, на которой расположены головки чтения-записи. Под каждой головкой дорожка - трек. Так же, как и в диске, все дорожки разделены на сектора. Для адресации блока данных в этом случае используется только номер дорожки (Nтрека) и номер сектора (Nсектора).
  - Используются в основном лишь в больших специализированных

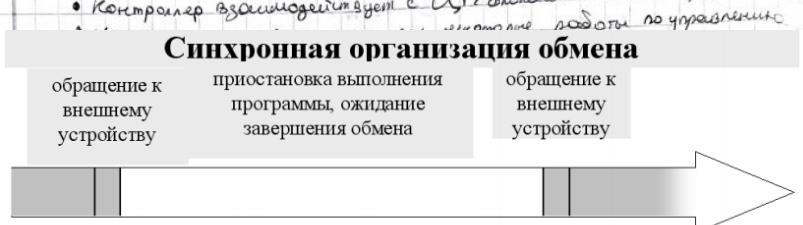
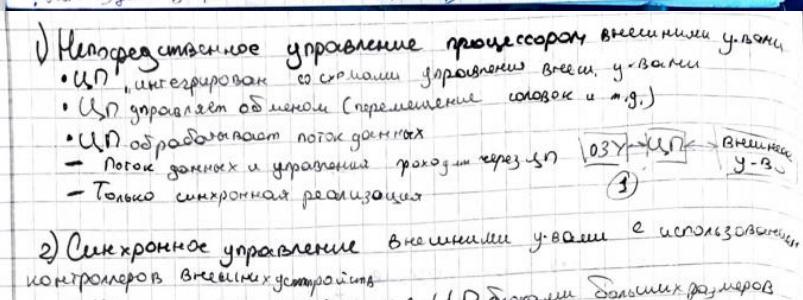
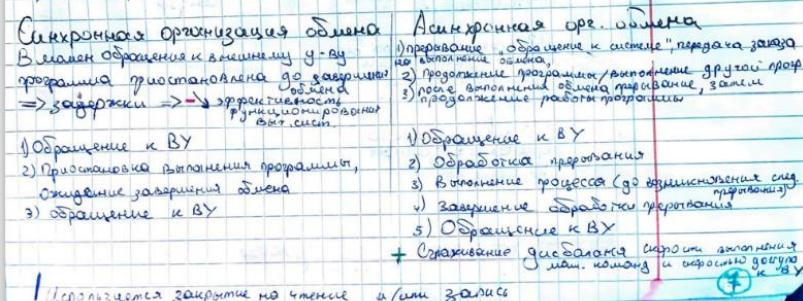
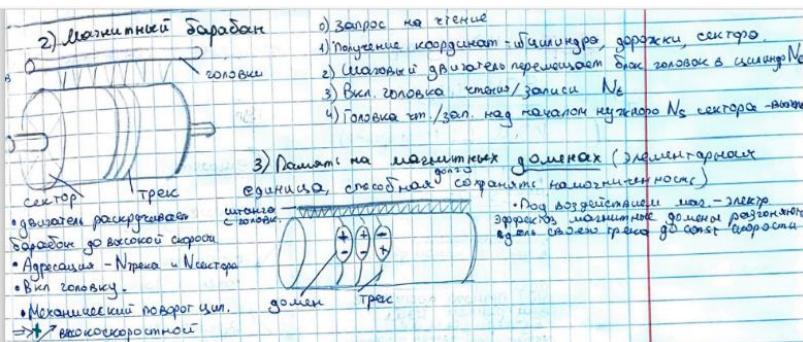
высокопроизводительных компьютерах обычно для временного хранения данных из ОП.

Хар-ка, опр эффективность функционирования ВС - **модель синхронизации**, поддерживаемая аппаратурой компьютера при взаимодействии ЦП с внешними устройствами. Для иллюстрации рассмотрим пример. Пусть выполняемой в компьютере программе необходимо записать блок данных на магнитный диск.

**1. Синхронная работа с ВУ:** в момент обращения к внешнему устройству программа будет приостановлена до момента завершения обмена => задержки снижают эффективность функционирования ВС.

**2. Асинхронная работа с ВУ:** пусть в системе прерываний компьютера имеется специальное внутреннее прерывание «обращение к системе», 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 ОС происходит обработка прерывания, при этом конкретному драйверу устройства передается заказ на выполнение обмена.



- После завершения обработки «обращения к системе» программа может продолжить свое выполнение, или может быть запущено выполнение другой программы.
- По завершении выполнения обмена происходит прерывание, после обработки которого программа, выполнявшая обмен, может продолжить свое выполнение.

Проблема: записывая область данных на ВЗУ после обработки заказа на обмен, но до завершения обмена, программа может попытаться обновить содержимое области => неккоректно => возможность аппаратного закрытия памяти на чтение и/или запись (при попытке обмена с закрытой областью памяти произойдет прерывание) => остановить выполнение программы до завершения обмена, если программа попытается выполнить неккоректные операции с областью памяти, находящейся в обмене (попытка чтения при незавершенной операции чтения с ВУ или записи при незавершенной операции записи данной области на ВУ)

## 2 потока инфы в компе:

1. поток управляющей информации,
2. второй поток — это поток данных, над которыми осуществляется обработка в программе.
- 3\*. В контексте ВЗУ: поток управляющей информации, включающий в себя команды, обеспечивающие управление внешним устройством, а также поток данных, перемещающихся между ВЗУ и ОП.

Модели организации управления ВЗУ.

### 1. Непосредственное управление процессором внешними устройствами (синхронная работа):

- ЦП «интегрирован» со схемами управления внешними устройствами, имеет специальные команды управления ими,
- путем интерпретации последовательности команд управления осуществляет управление обменом: ЦП подает команды устройству на перемещение головок обмена, на включение той или иной головки, на

<sup>1</sup> Примечание: процесс выполняется до возникновения следующего прерывания

Рис. 39. Синхронная и асинхронная работа с ВУ.

ожидание и синхронизации прихода содержательной информации и пр.

- обрабатывает и поток данных: считывает информацию со специальных регистров и переносит ее в ОП (или наоборот)
- Во время обмена вся система ждут, пока в регистре драйвера не появится код завершения обмена
- неэффективное использование компонентов системы, все ждут завершения обмена.
- трудоемкая задача, лишь синхронная реализация.

## 2. Синхронное управление внешними устройствами с использованием контроллеров внешних устройств (

- После появления устройств с электронными схемами управления — **контроллерами**, — взявшим на себя часть работ ЦП по управлению обменами.
- Контроллер взаимодействует с ЦП блоками больших размеров, при этом контроллер может самостоятельно выполнять некоторые работы по непосредственному управлению ВЗУ (попытка локализовать и исправить возможные ошибки).
- Исторически такой тип управления ВЗУ изначально был синхронным: процессор посыпал устройству команды на обмен и ожидает, когда этот обмен завершится.
- ЦП считывает поток данных со специальных регистров внешнего устройства и помещает их в ОП.

## 3. Асинхронное управление с использованием контроллеров ВЗУ.

- ЦП подает команду на обмен и не дожидается, когда эту команду отработают контроллер и устройство => процессор может продолжить обработку каких-то задач.
- Обращаемся к драйверу ус-ва, он передает инфу на аппаратуру управления магнитной ленты, после передачи команды управление возвращается в драйвер и в программу. Факт выполнения заказа передается в систему прерыванием.
- нужен аппарат прерываний.

## 4. Контроллеры прямого доступа к памяти (DMA — Direct Memory Access).

Контроллеры обрабатывают поток данных сами

- ЦП занимается лишь обработкой потоком управляющей информации, а данные перемещаются между ВЗУ и ОЗУ уже без его участия. Использование контроллера прямого доступа к памяти (DMA) или процессора (канала) ввода-вывода при обмене. И, наконец, последняя модель основана на
- Основное преимущество при использовании контроллеров прямого доступа при управлении внешними устройствами (DMA)? Исключается непосредственное участие процессора при переносе данных из ВЗУ в ОЗУ (и обратно)
- Основной недостаток использования модели непосредственного управления внешних устройств центральным процессором по сравнению с использованием DMA? Избыточная нагрузка на ЦП за счёт обработки потока данных, связанного с обменами.

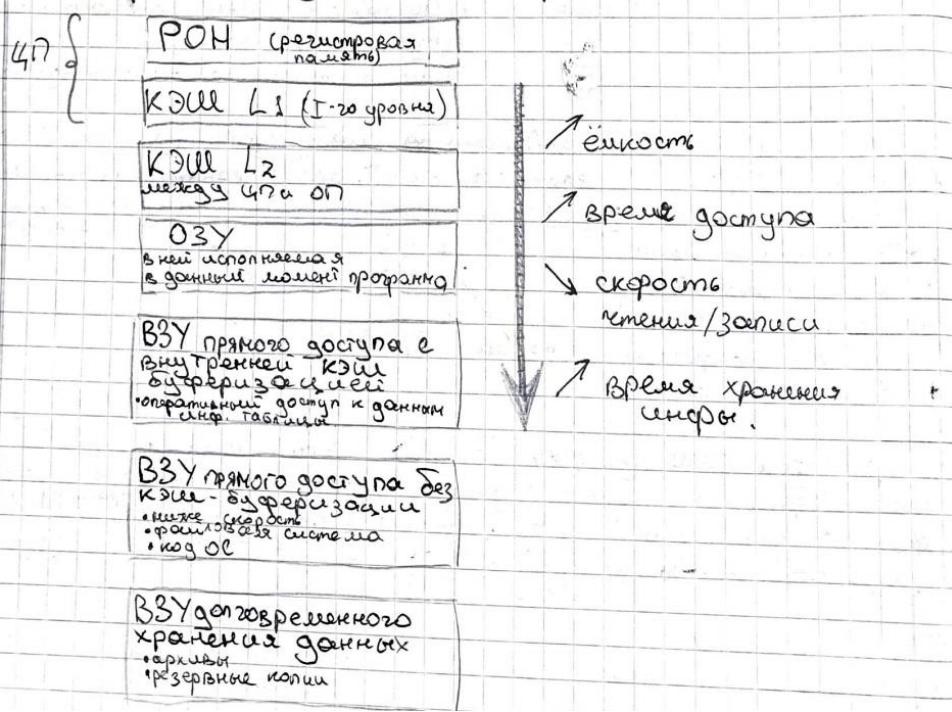
## 5. Использование процессора или канала ввода-вывода.

- предполагается наличие специализированного компьютера, который имеет свой процессорный элемент, свою ОП и ОС, компьютер располагается логически между ЦП и внешними устройствами.
- В функции подобных процессоров входит высокоуровневое управление внешних устройств.
- ЦП оперирует с внешними устройствами в форме высокоуровневых заказов на обмен => реализация непосредственного управления конкретным ВЗУ осуществляется в процессоре ввода-вывода (в частности, в нем может происходить многоуровневая фиксация ошибок, он может осуществлять аппаратное кэширование обменов к конкретному устройству и пр.).

---

## **11. Основы архитектуры компьютера. Иерархия памяти.**

## Иерархия устройств хранения информации



- Самой дорогостоящая и высокопроизводительная память размещается в ЦП (это регистровая память и КЭШ первого уровня (L1)).
- КЭШ второго уровня (L2) логически располагается между ЦП и ОП более дешевое и менее скоростное, чем КЭШ первого уровня, но более дорогое и более скоростное, чем ОЗУ, которое располагается на следующей ступени иерархии.
- В ОЗУ располагается исполняемая ЦП в данный момент программа, т.е. процессор «берет» очередные операнды и команды для исполнения именно из ОП. Ниже ОЗУ в иерархии следуют устройства, предназначенные для оперативного хранения программной информации пользователей и ОС:

- ВЗУ прямого доступа с внутренней КЭШ буферизацией - дорогостоящие устройства для наиболее оперативного обмена (всякого рода информационные таблицы).
- ВЗУ прямого доступа без КЭШ-буферизации, которые обеспечивают оперативных доступ, но уже на более низких скоростях (ФС пользователей, код ОС (поскольку для системного устройства, с которого происходит загрузка ОС, скорость не особенно актуальна в отличие от устройства, хранящего данные работающей ОС)).
- ВЗУ долговременного хранения данных.

## 12. Аппаратная поддержка ОС. Мультипрограммный режим.

**Мультипрограммный режим** – режим, при котором возможна организация переключения выполнения с 1 программы на другую. Предполагает функционирование ВС и обработку одновременно 2х и более программ пользователя. **Корректное функционирование** - в независимости от степени мультипрограммирования (от количества обрабатываемых в системе программ) результат работы конкретной программы не зависит от наличия и деятельности других программ.

Категории программ/процессов:

- Выполняются: машинные команды поступают в ЦП.
- Заблокированы по причине подачи заказа на обмен, ждут завершения обмена.
- Ждут, пока им будет предоставлен процессор (уже завершили обмен, если был запрос).

**Проблема несанкционированного доступа.** Особые требования к корректности этой системы. Система должна обеспечить, чтобы 1 программа не оказывала нелегального действия на систему и другие процессы => аппаратная поддержка корректного мультипрог.:

- Аппарат защиты памяти.** Выполняемая программа должна быть защищена от несанкционированного доступа других программ. Контроль принадлежности адреса адресному пр-ву, выделенному исполняемой программе/процессу (если нет => прерывание). Нужны регистры границ, в которых устанавливаются границы диапазона доступных для исполняемой задачи адресов ОП.
- Специальный режим ОС** – привилегированный/режим супервизора.
- Аппарат прерываний (как минимум, по таймеру)



Рис. 46. Процесс получения исполняемого модуля программы.

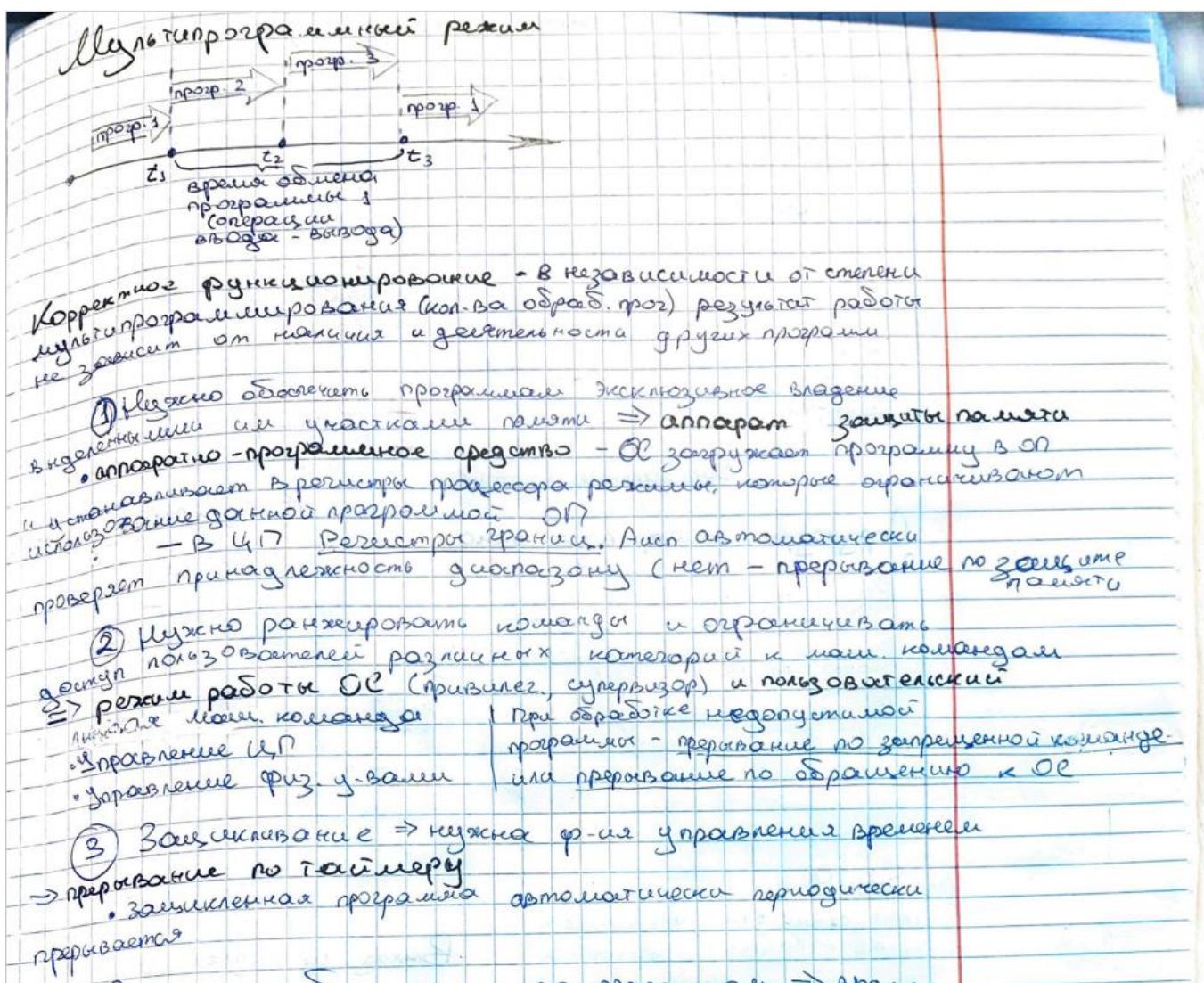
**Проблема: накладные расходы при смене обрабатываемой программы.** Это аналогичная проблема, связанная со сменой обрабатываемых программ (или процессов): ОС должна сохранить контексты процессов. К этому необходимо добавить, что в современных компьютерах количество одновременно обрабатываемых процессов очень велико, что лишь увеличивает объем возникающих накладных расходов.

В исполняемом модуле решены проблемы привязки по адресам. В ОС загружает в ОП нужные программы, потом в ОП места достаточно, но нет ни одного исп. модуля, который мог бы быть загружен в свободные диапазоны. Физически свободная память есть, но нет адресов, на которые настроены исполняемые модули (те адреса, которые использует 4-я программа, занимает 2-я). Мы должны поместить исполняемый модуль в оперативной памяти => **проблема перемещаемости памяти**, при этом не запускать каждый раз загрузчик.

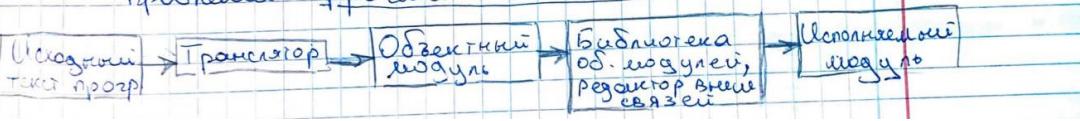


Соответствие адресов, используемых в программе, областям ОЗУ, в которой будет размещена данная программа

Рис. 47. Перемещаемость программы по ОЗУ.



- Вложенные обращения к подпрограммам  $\Rightarrow$  время на сохранение регистрарного контекста, адреса возврата и тд.
  - Расходы при смене обработчиком программы
  - Проблема привязки исполняемого модуля к адресному пр-ву, где он будет исполняться  $\Rightarrow$  проблема переноса семантики
- пр-ву, где он будет исполняться  $\Rightarrow$  проблема переноса семантики  
программы по ОЗУ (в силу привязки каждой программы к конкретным адресам ОЗУ с блочным ресурсом свободной памяти  
когда её выполняют)
- $\Rightarrow$  решение: исполняемый модуль и б загружены в производимое место ОЗУ для дальнейшего выполнения, но программа загружается в непрерывной фрагмент памяти
- $\Rightarrow$  - размер загружаемой программы < свободного фрагмента, проблемы фрагментации ОП  $\Rightarrow$  усечение когтей

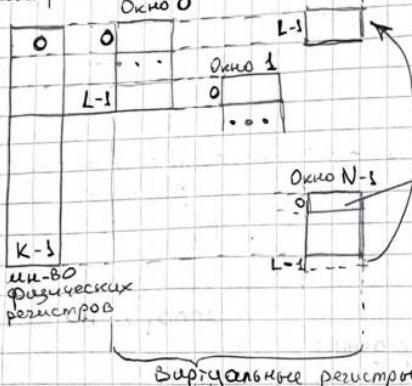


- Проблема программ из большого кол-ва модулей и подпрограмм – накладные расходы при входе и выходе из подпрограмм. В общем случае в модуле используются все регистры  $\Rightarrow$  регистровые окна.
- В процессоре физически есть К регистров общего назначения,
  - через систему команд в программе доступна только часть этих регистров.
  - Аппарат регистровых окон позволяет реализовывать команды, которые при входе в подпрограмму перемещают окно доступных регистров из отображения в одни реальные регистры в другие регистры  $\Rightarrow$  фактически доступны виртуальные регистры  $\Rightarrow$  виртуализация регистров

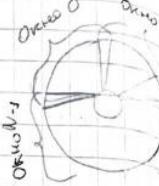
### 13. Аппаратная поддержка ОС и систем программирования. Организация регистровой памяти ЦП.

#### Способы решения проблем мультипрог.

① Регистровые окна – решение проблем сохр./восстановленных регистров. Все окна в циклическом списке, переключаются. Есть команда смены окна



- CWP – указатель текущего окна
- SWP – указатель сохраненного окна



- Фиксированное окно
- Фиксированное количество окон

Вход в подпрограмму

$$CWP = (CWP + 1) \bmod N$$

$$CWP = SWP$$

за

прерывание

или таймер

Откатка окна

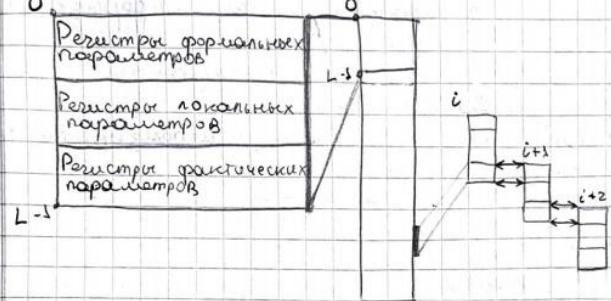
$$SWP = SWP + \% N$$

Использованные

окна CWP, вызов

функции

#### Структура регистрового окна



Какое-то окно сохраняется в ОР

CWP – указатель текущего окна, SWP – указатель сохраненного окна

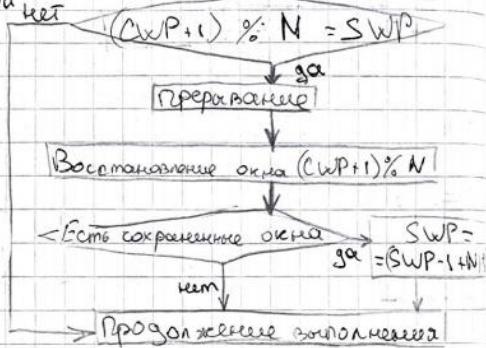
Какое-то окно сохраняется в ОП или стеке для дальнейшего использования, т.к. кол-во зарегистрированных окон ограничено  
→ настк CWP и SWP

Работа с зарегистрированными окнами имеет  
2 окна,  $A_1(A_2(A_3(A_4(A_5)))$ )

Окна CWP, вниз  
открываются

Выход из подпрограммы

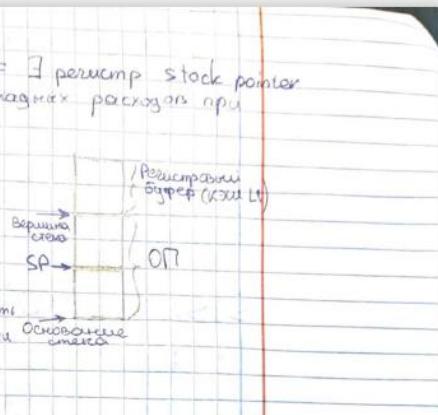
$$CWP = (CWP - 1 + N) \% N$$



## ② Системный стек

Аппаратная поддержка стека = регистр stack pointer  
- решение проблем минимизация накладных расходов при  
использовании обработавшей программы

- Стек в ОП  $\Rightarrow$  производительность
  - при прерываниях
  - в ЦП могут быть регистры - буферы, вершины стека
  - акумулирование вершин стека
  - работу с стеком можно организовать
- Буферизация в КЭШах, при кэшировании основные стеки  
стека добавить ещё в поток и крат.

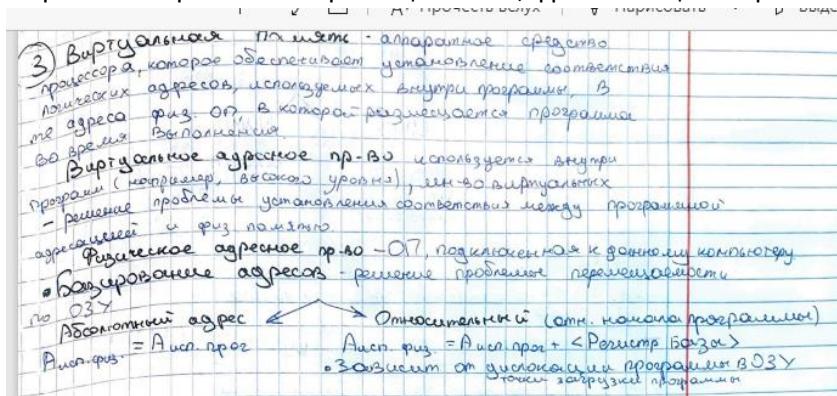


## 14. Аппаратная поддержка ОС. Виртуальная оперативная память.

Проблема фрагментации. Решение – виртуальная память – адресное пр-во, которое используется в программе. Уникально для каждой программы в исполняемом модуле. Модели виртуальной памяти:

### 1. Базирование адресов.

- В ЦП выделяется специальный регистр базы (мб >1).
- Если мы имеем Аисп, он либо абсолютный, либо относительный (тогда к нему добавляется значение регистра базы)  $\Rightarrow$  физический адрес
- **Аппарат виртуальной памяти** – аппаратное средство компьютера, осуществляющее преобразование программного (виртуального, используемый в программе) адреса в физический адрес.  
 $\Rightarrow$  решение проблемы перемещаемости, фрагментация не решается.



## 15. Аппаратная поддержка ОС. Пример организации страничной виртуальной памяти.

**«Аппарат» виртуальной памяти** - Аппаратно-программные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе с адресами

физической памяти, в которой размещена программа во время выполнения.

**Аппарат виртуальной памяти** – аппаратные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе адресам физической памяти в которой размещена программа при выполнении.

## 2. Страницчная организация памяти

- Все адресное про-во представляется как совокупность страниц – область памяти фиксированного размера ()
- Физическое про-во разрезается по страницам.
- Виртуальное адресное про-во тоже разрезается
- Если мы возьмем любой адрес и вырежем из него К крайних левых (младших) разрядов, то разряды с К и правее – номер страницы, а младшие К разрядов – номер в странице.
- Таблица страниц процессора имеет фикс размер
- Колво строк в таблице – предельное колво страниц в виртуальном адресном про-ве
- i-я строка таблицы соответствует i-й виртуальной странице обрабатываемого в настоящий момент процесса
- содержание i-й строки – номер физ страницы ОП, в котором размещена i-я страница.
- Алгоритм работы с таблицей:

1. Получаем Аисп виртуальный при выполнении программы,
2. Вырезаем из Аисп номер вирт страницы (с k до самого левого) => получили номер вирт страницы.
3. Индексируемся по таблице страниц (берем строку, номер которой равен номеру вирт страницы).
4. Находим строку, содержимое – номер физ страницы, в которой размещена соотв вирт страница процесса.

### Плюсы:

1. Механизм защиты памяти (в этой схеме процесс никогда не сможет обратиться к «чужой» странице),
2. Возможность разделять некоторые страницы между несколькими процессами (в этом случае ОС каждому из процессов допишет в таблицу страниц номер общей страницы).
3. Производительность, поскольку ее функционирование основано на использовании регистров.
4. Решение проблемы фрагментации, поскольку все программы оперируют в терминах страниц (каждая из которых имеет фиксированный размер), и мы можем размещать программу по страницам в произвольном порядке.
5. Проблема перемещаемости программ по ОЗУ, причем даже в рамках одной программы соответствие между вирт и физ страницами может оказаться произвольным: Одна виртуальная страница программы может располагаться в одной физической странице, другая – в другой (совершенно не связанной с первой) физической странице, и т.д.
6. Нет необходимости держать в ОП весь исполняемый процесс => ↑ эффективность использования ОП. Реально в ОЗУ может находиться лишь незначительное число страниц, в которых расположены команды и требуемые для текущих вычислений операнды, а все оставшиеся страницы могут находиться на внешней памяти – в областях подкачки.
7. размеры физической и виртуальной памяти могут быть произвольными. Но во всех этих случаях система окажется работоспособной.

### Минусы:

1. Внутренняя фрагментация памяти.
2. Модель вырождена: если таблица страниц целиком располагается на регистровой памяти, то в силу дороговизны последней размеры подобной таблицы должны быть слишком малы (=> невелико количество физических страниц).

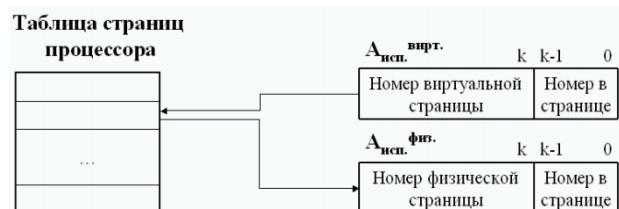


Рис. 57. Страницчная организация памяти. Преобразование виртуального адреса в физический.



Рис. 58. Модельный пример организации страницочной виртуальной памяти. Схема преобразования адресов.

3. При смене процессов таблицу страниц сначала обязательно надо сохранить, а потом обновить – дополнительные накладные расходы.

- Преимущества сегментной организации памяти по сравнению со страничной?

1. Простая аппаратная реализация (аппаратная таблица сегментов)

2. Возможность адресной поддержки логической структуры виртуальной памяти процесса (кодовая часть, данные, стек, динамическая память и пр.)

- Преимущества страничной организации памяти по сравнению с сегментной?

1. Решение проблемы фрагментации памяти.

2. Оптимальность использования физической памяти – возможность «откачки» значительной части неиспользуемых страниц.

## 16. Многомашинные, многопроцессорные ассоциации. Классификация. Примеры.

Классификация многопроцессорных систем по Флинну (M.Flynn), основанная на анализе некоторых характеристик потоков информации в машине.

- Основная концепция – перебор всевозможных характеристик потока команд (инструкций) и потока данных.

- Обработка каждого из этих потоков может быть одиночная либо множественная.

- В контексте машины можно выделить: **поток управления** (для передачи управляющих воздействий на конкретное устройство), переход от команд низкого уровня к высокоуровневым (когда ЦП вместо работы с микрокомандами начинает вырабатывать высокоуровневые команды, которые передаются «умному» устройству управления, непосредственно реализующему данные команды); и **поток данных** (циркулирующий между оперативной памятью и внешними устройствами), это исключение участия ЦП в обменах между внешними устройствами и оперативной памятью.

- Рассматривать компьютер с позиции 2 потоков:

- поток команд: выбор одной или группы команд

- поток данных, операндов: с выполнением каждой команды выбирается либо единичная, либо множественная порция данных.



Четыре класса архитектур:

– ОКОД (одиночный поток команд, одиночный поток данных, или SISD — single instruction, single data stream)

– 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 — single instruction, multiple data stream) — 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.

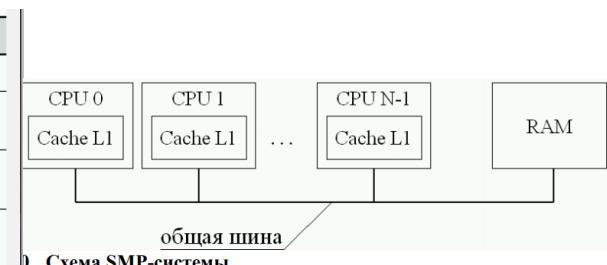
– **МКОД (множественный поток команд, одиночный поток данных, или MISD** – multiple instruction, single data stream) — имеется смесь команд, которая оперирует над одними и теми же данными. Этот класс архитектур является спорным. Существуют различные точки зрения о существовании каких-либо систем данного 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. Вырожденная группа.

– **МКМД (множественный поток команд, множественный поток данных, или MIMD** – multiple instruction, multiple data stream) — 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. Два подкласса:

1. **системы с общей ОП**, любой процессор имеет непосредственный доступ к любой ячейке этой общей ОП, любой адрес может появляться в произвольной команде в любом из устройств управления.

- **UMA (uniform memory access)** — MIMD система с общей ОП, имеющая однородный доступ в память. Характеристики доступа любого процессорного элемента в любую точку ОЗУ не зависят от конкретного элемента и адреса (все процессоры равнозначны относительно доступа к памяти). Работа с КЭШ: Хотим записать по адресу A, в локальном КЭШе нет строчки с таким адресом => промах => инициируем запись операнда с информацией с адресом A. Все процессы увидели, что на шине идет обращение в ОП => чекают свои КЭШи и удаляют. Алгоритм ориентирован на то, что чаще идет обращение по чтению.

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



- Подвидом UMA-систем является модель **SMP (symmetric multiprocessor** — симметричная мультипроцессорная система). К общей системной шине, или магистрали, подсоединяются несколько процессоров и блок общей ОП.

- **NUMA (non-uniform memory access** — система с неоднородным доступом к памяти) - MIMD система с общей ОП, имеющая неоднородный доступ в память. Процессорные элементы работают на общем адресном пространстве, но характеристики доступа процессора к ОЗУ зависят от того, в какую часть адресного пространства он обращается. Степень параллелизма в NUMA-системах выше, чем в SMP. Свойства:

процессорные элементы работают на общем адресном пространстве;

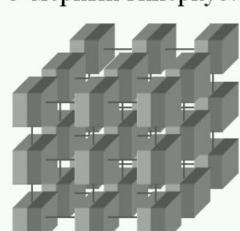
характеристики доступа процессора к области ОП зависят от того, к каким областям идет обращение (к локальной или нелокальной памяти). Контроллер памяти позволяет осуществлять взаимодействие между вычислительными узлами. Доступ процессора к своей ОП осуществляется через свой контроллер, к чужой — через два контроллера. При обращении не к своей памяти контроллер выбрасывает запрос на общую шину, целевой контроллер его принимает и возвращает результат.



Макроконвейер:



3-мерный гиперкуб:



Процессорный элемент с локальной памятью

Межэлементные коммуникации, определяющие топологию

Примеры топологий MPP-систем.

2. **системы с распределенной памятью MPP, COW** -

объединение компьютерных узлов (самостоятельный процессор со своей локальной ОП). В данных системах любой процессор не может произвольно обращаться к памяти другого процессора.

Наиболее перспективные с точки зрения их массового распространения и использования. Два основных класса:

- **MPP (Massively Parallel Processors** — процессоры с массовым параллелизмом) – специализированные дорогостоящие ВС, могут выстраиваться, процессорные элементы могут объединяться в различные топологии, которые определяются специализацией компьютера. Эффективные системы для решения определенного класса прикладных задач.

- **COW (Cluster of Workstations** — кластеры рабочих станций). Суперкомпьютеры для бедных. Дешевые рабочие офисные станции, соединять их таким образом, что на этой системе можно решать задачи на распараллельном режиме. Это многомашинные системы, состоящие из множества узлов, каждый из которых может быть обычным компьютером. Мин узел может выступать процессор со своей локальной ОП и аппаратурой сопряжения с другими вычислительными узлами. Для сопряжения вычислительных узлов в кластере используются специализированные компьютерные сети. Цели создания кластеров:

1. Кластер как основа для построения суперкомпьютера (вычислительный узел, высокопроизводительная система) (критерием эффективности выступает скорость обработки информации). Это многопроцессорная система, состоящая из вычислительных узлов. Каждый вычислительный узел – это компьютер традиционной архитектуры. Все узлы связаны друг с другом через специальные высокоскоростные сети, и вся система в целом используется для достижения максимальной производительности. Для построения вычислительных кластеров зачастую используют Unix-системы, а для кластеров надежности – Windows-системы.

2. Кластеры, обеспечивающий надежность (сохранение работоспособности при возможном снижении производительности). Программноаппаратные комплексы для обеспечения устойчивого функционирования системы для решения конкретной прикладной задачи (например, сервер базы данных авиабилетов), при этом выход из строя некоторых узлов не означает отказ всей системы: система продолжает функционировать, пусть и со сниженной производительностью;

#### Недостатки UMA:

1. централизованная система, и общая шина в ней является «узким горлом» => существенные ограничения на количество подключаемых процессорных элементов (обычно 2, 4, 8, вплоть до 32).

2. возникают дополнительные проблемы синхронизации КЭШей 1го уровня каждого процессора. Решения:  
не использовать КЭШ,

реализовать КЭШ-память со слежением: каждый КЭШ слушает шину и реагирует на ситуацию в системе.

3. Существенно ограничивается параллелизм (чтобы ограничить кол-во обращений к общейшине для предотвращения конфликтов). Решение – увеличить скорость шины.

#### Плюсы NUMA по сравнению с UMA:

Потенциально большая степень параллелизма (в UMA системах существенно ограничено количество процессорных элементов).

#### Недостатки NUMA:

1. Проблема синхронизации КЭШа, использование когерентных КЭШей загружает шину служебной информации. Решения:

- использовать процессоры без КЭШа (использовать только Cache L2);
- использовать модель **ccNUMA (Cache coherent NUMA)** – это NUMA-система с когерентными КЭШами. CcNUMA-системы позволяют отслеживать соответствие локальных КЭШей друг другу и состояние всей системы в целом.

2. CcNUMA-системы подключают несколько сотен процессоров, но остаются ограничения, связанные с централизацией – использованием системной шины, а также возникают ограничения, связанные с архитектурой: появляются системные потоки служебной инфы => накладные расходы – загрузка общей шины служебной информацией.

#### Преимущества MPP-систем:

- высокая эффективность при решении определённого класса задач.

#### Минусы:

1. Дорого, сложно.
2. Узкоспециализированные многопроцессорными системами, не находят массового применения.

## Преимущества кластерных систем:

1. высокая эффективность при решении широкого круга задач (за приемлемую цену);
2. используют унифицированные средства программирования (типа MPI).

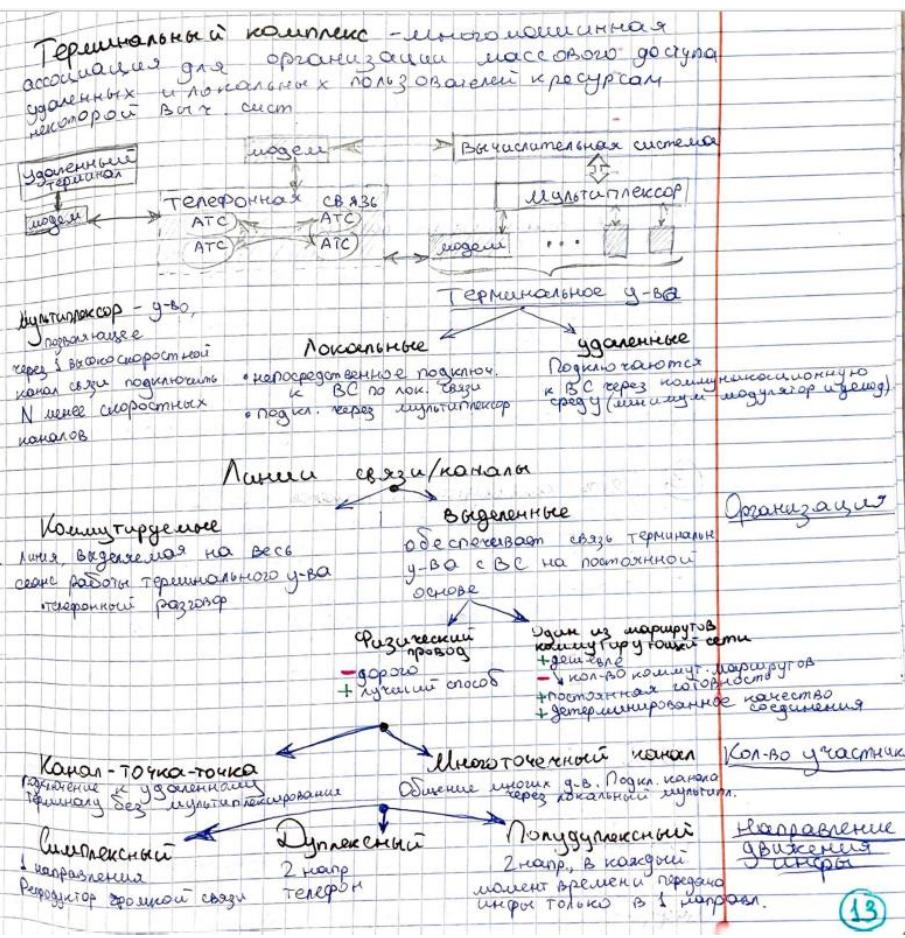
## Недостатки:

1. отвод тепла и коммуникация (если будет использоваться единственная магистраль, то она «захлебнется» от потоков передаваемой информации, а большинство узлов будут простаивать).

Различие между COW и MPP.

- В кластере каждая машина в значительной степени независима от других с точки зрения памяти, диска и тд. Они связаны между собой с использованием некоторых вариаций в обычной сети. Кластер существует в основном в уме программиста и в том, как он / она решает распределить работу.
- В MPP действительно есть только одна машина с тысячами тесно связанных между собой процессоров. MPP имеют экзотические архитектуры памяти, позволяющие чрезвычайно быстро обмениваться промежуточными результатами с соседними процессорами.

## **17. Многомашинные, многопроцессорные ассоциации. Терминалные комплексы. Компьютерные сети.**



### **Терминальный комплекс –**

многомашинная ассоциация (объединение комп устройств), предназначенный для организации удаленного доступа к ресурсам ВС (высокая производительность рассматриваемой системы или же информационный ресурс) значительного кол-ва пользователей, которые могут работать с ресурсом на любом расстоянии от ресурса (удаленных и локальных пользователей).

### **Локальные терминальные устройства**

имеют либо непосредственное подключение к ВС по локальному каналу связи, либо подключение через мультиплексор (устройство, позволяющее через один высокоскоростной канал связи подключить N менее скоростных каналов). размещаться на небольшом расстоянии от компа, не требует сильных средств для усиления передачи сигнала. Подключение локального терминала к компьютерной системе может подключаться через интерфейсы компьютерной системы. При

таком подключении затруднительно использовать массовость.

**Удаленные терминальные устройства** подключаются к ВС через коммуникационную среду (изначально обычные телефонные сети, аналоговый способ передачи инфы; компьютерные сети основаны на цифровом (дискретном) способе передачи данных. Для передачи цифровой информации через аналоговые сети необходимы аналого-цифровой и цифро-аналоговый преобразователи (**модем** — модулятор-демодулятор) — устройства, предназначенные для преобразования двоичной инфы в непрерывный сигнал. Для удаленного соединения необходимы > 2 модема. Удалённые терминалы могут также мультиплексироваться (многоуровнево). Используются **мультиплексоры** — аппаратные устройства, осуществляющие подключение группы терминалов к компьютерной системе через 1 аппаратный интерфейс => увеличиваем кратно кол-во подключаемых устройств, но снижается скорость.

**Линии связи/каналы.** Критерии, по которым можно классифицировать каналы.

1. **коммутуемые и выделенные.** **Коммутуемый канал** — это линия, выделяемая на весь сеанс работы терминального устройства. Физически коммутуемая линия может иметь различные топологии.

Телефонный разговор. **Выделенный канал** (фиксируется соединение терминального устройства и компьютера) обеспечивает связь терминального устройства с ВС на постоянной основе:

- можно протянуть между терминальным устройством и ВС физический провод (дорого, но это наилучший способ обеспечить выделенный канал),
- можно взять один из коммутируемых маршрутов телефонной сети (дешевле предыдущего, но уменьшает количество коммутируемых маршрутов в телефонной сети => ухудшение работы «по прямому назначению» — на обеспечении телефонной (голосовой) связи).

### 2. Кол-во 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.

### 3. По направлению движения информации в канале:

- **симплексный** — канал позволяет осуществлять передачу инфы только в одном направлении (например, репродуктор громкой связи на вокзале или в организациях; идеальная лекция);
- **дуплексный** (двунаправленный) — передаёт инфу в двух направлениях (например, телефон);
- **полудуплексный** — позволяет передавать инфу в двух направлениях, но в каждый момент времени возможна передача информации лишь в одном направлении (например, радио).

### Преимущества выделенного канала:

1. постоянная готовность (отсутствие отказа на соединение),
2. детерминированное качество соединения

Развитие терминальных комплексов => развития компьютерных сетей => замена терминальных устройств компьютерами. **Компьютерная сеть** — это объединение компьютеров (или ВС), взаимодействующих через коммуникационную среду (Рис. 64). **Коммуникационная среда** — каналы и средства передачи данных.

### Свойства компьютерных сетей:

1. логически сеть может состоять из значительного числа связанных между собой автономных компьютеров, каждый из которых обеспечивает решение определённых задач. **Два компьютера называются связанными** между собой, если они могут обмениваться информацией. Требование автономности => исключить из рассмотрения компьютерные системы, в которых один компьютер может принудительно запустить, остановить другой компьютер или управлять его работой.
2. компьютерная сеть предполагает возможность распределенной обработки инфы, когда пользователь компьютерной сети может получать в качестве результата данные, обработанные за счет интеграции этапов обработки данных на разных компьютерах сети.
3. расширяемость сети => сеть должна обеспечивать возможность развития по протяженности, по расширению пропускной способности каналов, по составу и производительности компонентов сети.
4. возможность применения симметричных интерфейсов обмена информацией между компьютерами сети. Эти интерфейсы позволяют произвольным способом распределять функции сети между компьютерами.

Логически (на деле могут выполнять оба) компьютеры, составляющие сеть, можно подразделить на:

1. абонентские машины (или основные компьютеры — хосты)
2. коммуникационные (или вспомогательные) компьютеры (шлюзы, маршрутизаторы и пр.) - выполняют фиксированные функции по обеспечению функций сети (выбор маршрута, передача информации и т.д.).

Сетевые устройства **взаимодействуют друг с другом посредством передачи сообщений**. Сообщение может иметь произвольный размер. Сетевые устройства взаимодействуют друг с другом в рамках **сеансов связи** – периода времени, в течение которого сетевые устройства обмениваются сообщениями. С сеансов связи ассоциируется функция начал и завершения сеанса связи. Под **сообщением** будем понимать логически целостную порцию данных, имеющую произвольный размер.

### 3 модели сетей:

1. **В сети коммутации каналов** коммутация каналов происходит на весь сеанс связи.
  - + канал всегда находится в состоянии готовности;
  - + требования к коммуникационному оборудованию минимальны: по сути, нет требований к обеспечению

буферизации;

+ обмены происходят сообщениями => ↘ накладных расходов по передаче информации;

+ детерминированная пропускная способность сети.

– дороговизна: любой выделенный канал требует больших материальных затрат;

– наличие высокой избыточности в сети: должно быть либо много каналов, чтобы не было коллизий, либо в сети будут коллизии – при этом период ожидания свободного канала недетерминирован;

– неэффективность использования выделенного коммутационного канала: в отдельно взятом сеансе может быть низкая интенсивность обмена сообщениями;

– при сбоях или отказах повторение переданной информации является сложной задачей.

**2. Сети коммутации сообщений** — это сети, которые оперируют термином «передача сообщения», а не «сеанс связи». Абонент-отправитель передаёт сообщение в сеть по 1му свободному каналу, который у этого абонента есть. Выделение канала для передачи каждого сообщения происходит поэтапно (по мере продвижения сообщения) от одного узла к другому. На каждом узле на пути следования принимается решение, свободен ли канал к следующему узлу. Свободен => сообщение передается далее, иначе => ожидание освобождения канала

+ отсутствие выделенного канала и, соответственно, занятости ресурса коммутируемого канала на недетерминированный промежуток времени, т.е. устранена деградация системы, возникающая при организации сетей коммутации каналов;

– сообщения могут быть произвольного размера => необходимость наличия в коммутационных узлах средств буферизации (в общем случае неизвестно, какой мощности, поскольку сообщение имеет произвольную длину) => сеть имеет недетерминированные характеристики (скорость передачи по сети и т.д.);

– обеспечение буферизации требует дорогостоящего коммутационного оборудования и ПО;

– необходимость повторения сообщения в случае сбоя при передаче, что само по себе является сложной задачей, хотя менее трудоемкой, чем для сетей коммутации каналов.

**3. Модель сети коммутации пакетов** строится на предположении использования в сети ненадежных средств связи. Функционирует по принципу горячей картошки. Каждое сообщение разбивается на блоки фиксированного размера, которые называются пакетами.

То есть передача сообщения – это передача цепочки пакетов (движение пакетов по сети – аналогично предыдущей модели, но существенное различие между этими моделями заключается в детерминированности размера пакета). На стороне отправителя происходит разбиение сообщения на пакеты, а на стороне получателя — сборка. Любой пакет помимо непосредственно самого сообщения (или его части) имеет служебную информацию (которая обычно представлена в заголовке пакета), обеспечивающую внутреннюю целостность пакета (контрольная сумма пакета и пр.); адресную составляющую (данные об отправителе и адресате), а также информацию для сборки сообщения из пакетов. Стратегия передачи: любой узел, получив пакет, пытается сразу от него избавиться. Любая сеть имеет фиксированную топологию, а также фиксированное количество и расположение абонентов, то возможно просчитать ее характеристики и предъявить требования к коммуникационным узлам. Данная модель допускает буферизацию в узлах передачи: пакет, прийдя на узел, может быть послан несколько позже, если все необходимые выходные каналы заняты. Но период занятости канала при известной стратегии обработки буфера предопределен => можно оценить предельный размер буфера, а также предельные

периоды ожидания пакетов при передаче их по сети => если известна стратегия передачи, пропускная способность является детерминированной величиной.

+ детерминированность (детерминированность перемещений, детерминированность требований к коммуникационному оборудованию),

+ при сбое достаточно заново послать потерянные пакеты, а не все сообщение целиком.

- увеличение трафика из-за того, что по сети перемещается накладная информация, которая прибавляется к каждому пакету при разбиении сообщения на пакеты. Как известно, эта служебная информация занимает 80-90% пропускной способности канала.

- разбиением сообщения на пакеты => сборка — это аккумуляция пакетов, а также сама сборка (необходимо обеспечить наличие всех переданных пакетов и их правильный порядок).

**Операционная система** — это комплекс программ, в функции которого входит обеспечение контроля за существованием, использованием и распределением ресурсов ВС. ВС может включать в свой состав как физические, так и виртуальные ресурсы.

1. ОС обеспечивает контроль за существованием ресурсов => обеспеч реализации виртуальных ресурсов и предоставление средств доступа к физическим ресурсам. Степень доступности ресурса зависит от ОС: файловая система: этого ресурса может и не быть в операционной системе, может существовать одна модель, или другая модель, или сразу несколько моделей.

2. использование ресурсов. Здесь имеется в виду, что ОС предоставляет все средства, обеспечивающие доступность ресурсов ВС пользователю (точнее программам). При использовании любых ресурсов ВС может возникнуть конкуренция.

3. распределение: под этим будем понимать выбор стратегии распределения и обеспечение всевозможных моделей регламентации доступа. Решается 2мя моделями:

- Предварительное выделение потребителю запрашиваемого ресурса.
- Выделение ресурса по запросу

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

1. Понятие совокупности машинных команд и данных обозначает то, что принято называть исполняемой программой (т.е. это код и операнды, используемые в этом коде).

2. Под термином обработки в рамках ВС будем понимать, что эта программа сформирована и находится в системе в режиме обработки (это может быть и ожидание, и исполнение на процессоре, и т.п.).

3. Понятие обладания правами на владение некоторым набором ресурсов обозначает, по сути, возможность доступа. Процесс можно определить как исполняемую программу, которая введена в систему для ее обработки и с которой ассоциированы некоторые ресурсы ВС. Ресурсы, выделяемые процессам, могут быть двух типов.

1. ресурсы, которые выделяются процессу на эксклюзивных правах -
2. те ресурсы, которые одновременно могут принадлежать двум и более процессам, – **разделяемые ресурсы**. Разделяемый ресурс может одновременно принадлежать нескольким процессам != к нему возможен одновременный доступ.

С точки зрения выделения ресурса процессу используются две стратегии организации выделения.

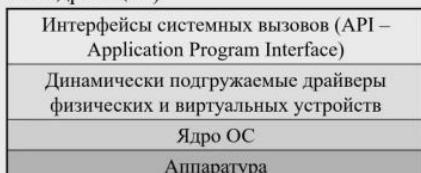
## Свойства ОС:

**1. Надежность** - число программных ошибок в системе должно быть сведено к минимуму и должно быть соизмеримо с количеством возможных аппаратных сбоев.

**2. Защита информации и ресурсов от несанкционированного доступа.** Эта проблема на сегодняшний день

## **Структура ОС**

**Ядро (Kernel)** — резидентная часть ОС, работающая в режиме супервизора (обычно работает в режиме физической адресации).



#### Динамически подгружаемые драйверы устройств:

- резидентные / нерезидентные
  - работают в пользовательском / привилегированном режиме

**Системный вызов** — обращение к ОС за предоставление той или иной функции (возможности, услуги, сервиса).

## Структура ОС

## Пример структурной организации классической системы UNIX



- Резидентные драйверы подгружаются в систему в процессе ее загрузки и находятся в ней до завершения ее работы (драйвер физического диска).

- Нерезидентные драйверы вызываются ОС на сеанс работы с соответствующими устройствами (драйвер флэш-памяти).

Большинство ОС имеют набор драйверов широкого спектра конкретных физических устройств => при смене устройства драйвер менять не надо: он уже есть в системе. Но при этом системе незачем держать драйвера всех устройств в ОП => следуя той или иной стратегии, будут загружаться драйверы тех физических устройств, которые реально будут обслуживаться системой.

1. может быть явное указание системе списка драйверов, которые необходимо подгрузить (в этом случае, если в списке что-то будет указано неправильно, то соответв. устройство м.б. не будет работать).

2. система при загрузке самостоятельно сканирует подключенное к ней оборудование и выбирает те драйверы, которые должны быть подгружены для обслуживания найденного оборудования

### **3. Интерфейсы системных вызовов (API – Application Program Interface).**

- **системный вызов** - средство обращения процесса к ядру ОС за выполнением той или иной функции (возможности, услуги, сервиса). Отличие обращения к системному вызову от обращения к библиотеке программ заключается в том, что библиотечная программа присоединяется к исполняемому коду процесса => вычисление библиотечных функций будет происходить в рамках процесса. Обращение к системному вызову — это вызов тех команд, которые инициируют обращение к системе => либо прерывание, либо исполнение специальной команды. Тело и реализация системного вызова остается в ядре, в процессе – только обращение. Ресурс, который потребляет системный вызов при обработке запросов к нему - ресурс ОС.

### - Структурная организация ОС:

1. Классический подход на использовании **монолитного ядра**, ядро ОС представляет собою единую монолитную программу, в которой отсутствует явная структуризация, хотя, конечно, в ней есть логическая структуризация => ядро содержит фиксированное число реализованных в нем базовых функций => модификация функционального набора затруднительна (необходима практически полная переделка ядра).

+ для конкретного состава функциональности и логики ядра это будет наиболее эффективное решение (т.к. оно имеет минимальное количество интерфейсных сочленений, связанных со структуризацией).

- отсутствует универсальность, и внутренняя организация ядра рассчитана на конкретную реализацию.

- необходимость перепрограммировать ядро при внесении изменений => для внесения новой функциональности пользователю системы приходится обращаться к разработчику, что зачастую ведет к материальным затратам

2. **многослойные ОС**, все уровни разделяются на функциональные слои. Здесь можно провести аналогию с моделью сетевых протоколов. Между слоями имеются фиксированные интерфейсы. Управление происходит посредством взаимодействия соседних слоев. Поскольку любая структуризация снижает эффективность (программа, написанная в виде одной большой функции, работает быстрее, чем аналогичная программа, разбитая на подпрограммы, т.к. любое обращение к подпрограмме ведет к накладным расходам), то подобные системы обладают более низкой эффективностью. Итак, каждый слой предоставляет определенный сервис вышестоящему слою. Деление на слои является индивидуальным для каждой конкретной операционной системы. Это может быть слой файловой системы, слой управления внешними устройствами и т.д. Тогда модернизация подобных систем сводится к модернизации соответствующих слоев. Вследствие чего проблема несколько упрощается, но при этом остаются ограничения на структурную

организацию (например, имея слой файловой системы, можно заменить его другим вариантом этого слоя, но использовать одновременно две различные файловые системы не представляется возможным). Третий подход предлагает использовать **микроядерную архитектуру** (Рис. 74). 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), которые

### **Логические функции ОС:**

1. Управление процессами: создать процесс, выделить ресурсы, защитить, освободить ресурсы.
2. Управление ОП
3. Планирование. Использование ресурсов значительного кол-ва потребителей => очередь и конкуренция. Одна из основных ф-ий ОС – планирование => Дисциплина предоставления ресурсов.
4. Управление устройствами и ФС =>
5. Сетевое взаимодействие. Любая современная система ориентирована на сетевое взаимодействие.
6. Безопасность, защита от несанкционированного доступа.

---

## **19. Операционные системы. Пакетная ОС, ОС разделения времени, ОС реального времени.**

Модели функционирования ОС с точки зрения критериев эффективности и стратегий использования ЦП.

**1. Пакетная** - система, критерием эффективности функционирования которой является максимальная загрузка ЦП (т.е. минимизация накладных расходов) => минимизировать накладные расходы снижением использования ОС. Иными словами, отношение всего времени работы процессора ко времени исполнения пользовательских программ должно быть близко к единице. Для решения расчетных задач, т.е. задач, требующих определенного объема времени работы процессора. Как следует из названия, эти системы оперируют термином пакет программ.

- **Пакет программ** — это некоторая совокупность программ, которые системе необходимо обработать.

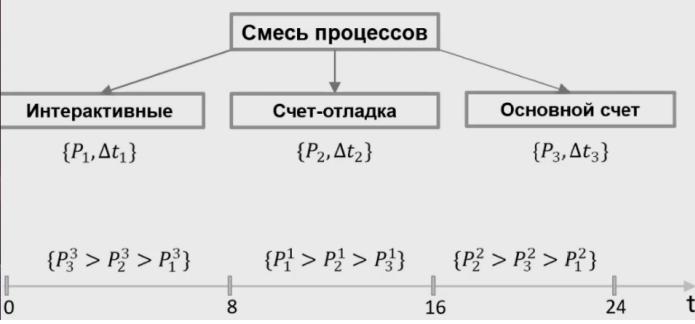
- Переключение выполнения:

Первая причина — завершение выполнения процесса (в силу успешного перехода на точку завершения программы или же в силу возникновения ошибки).

Вторая причина — обращение к внешнему устройству с целью осуществления обмена, т.е. возникновение прерывания по вводу-выводу, поскольку операция обмена так или иначе требует какого-то минимального интервала времени.

Третья причина — фиксация факта зацикливания процесса. В принципе точно определить факт зацикливания программы сложно, но все-таки возможно. На практике зачастую под фактом зацикливания считают исчерпание процессорного времени

## Модель организации планирования процессов



**2. Система разделения времени.** Развитие модели пакетных систем - для каждого процесса в системе определяется квант процессорного времени - некоторый фиксированный ОС промежуток времени работы процессора => переключение процессов происходит по тем же причинам, что и в пакетных системах (завершение процесса, возникновение прерывания, фиксация факта зацикливания) + исчертался выделенный квант времени.

- Критерий эффективности - минимизация времени отклика системы на запрос пользователя (набранные символы отображались на экране достаточно быстро, иначе работать с системой ему будет неудобно).
- частая смена контекстов => накладными расходами.

- Варьируя размерами кванта времени, можно получать системы для решения тех или иных задач.

Во-первых, разделить все процессы на группы по некоторым критериям (например, интерактивные, отладочные и т.д.).

Во-вторых, для каждой из этих групп определить квант времени ЦП, который будет выделяться процессу из конкретной группы.

В-третьих, решить вопрос справедливой организации вычислительного процесса, т.е. определить приоритеты для каждой из категорий процессов. + Критерий смены процесса - появление процесса из более приоритетной группы.

В-четвёртых, приоритеты категорий процессов должны определяться по расписанию (например, в зависимости от времени суток).

**3. Система реального времени** - это специализированные системы, которые предназначены для функционирования в рамках ВС, обеспечивающих управление и взаимодействие с различными технологическими процессами. При разработке подобных систем все функции планирования ориентированы на обработку некоторого фиксированного набора событий, при возникновении любого из которых гарантируется обработка этого события за некоторый промежуток времени, не превосходящий определенного предельного значения.

Различные группы систем реального времени:

1. жесткого времени (например, управление бортовой системой самолёта),
2. мягкого времени и пр.;

Процессорный элемент - составляющая многопроцессорных или многомашинных ассоциаций => категории.

1. сетевые ОС (Рис. 75). **Сетевая операционная система** — это система, обеспечивающая функционирование и

взаимодействие ВС в пределах сети, устанавливается на каждом компьютере сети и обеспечивает функционирование распределенных приложений, т.е. тех приложений, реализация функций которых распределена по разным компьютерам сети. Примеров можно привести достаточно много. Так, почтовое приложение может быть распределенным: есть функции перемещения, есть сервер-получатель, есть клиентская часть, обеспечивающая интерфейс работы пользователя с указанным сервером.

2. Распределенные ОС. 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 (т.н. сервисы или услуги) (функция управления заданиями, распределенная файловая система.)

---

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

Появление компьютерных сетей => проблема организации сетевого взаимодействия через коммуникационную среду > 2x компьютеров. Распределение функций между компьютерами: централизованное (архитектура терминального комплекса), так и децентрализованное (сеть, состоящая из равнозначных по ролям и по функциям компьютеров) => проблемы аппаратного взаимодействия устройств и сопоставления взаимодействующих программ.

Монолитная структура и архитектура компьютера:

- крайне сложно подключать к компьютеру новые устройства (должна была проводиться очень тяжёлая аппаратная проработка).
- появление новых компьютеров практически зачёркивало всё то, что было для старых – отсутствовала программная преемственность (переносимость программ).  
=> необходимость стандартизации (компьютерных комплектующих, программных интерфейсов) => возможность взаимозаменяемости компонентов и их работоспособности
- + децентрализовать производство компьютера (появились независимые производители центральной части, периферийного оборудования, программного обеспечения и т.д.),
- + возможность достаточно простой стыковки адаптации, модернизации техники.

Стандартизация позволяет современным производствам и организациям производственных процессов быть развивающими, ремонтоспособными, обслуживаемыми. Стандартизация языки программирования => появление Unix => многоуровневая стандартизация интерфейсов ОС (от стандартизации интерфейсов системных вызовов до стандартизации интерфейсов систем обработки команд) => стандарт POSIX (Portable Operating System Interface),

Терминальные комплексы, строятся по внутрикорпоративным правилам => корпоративные стандарты на подключение оборудования, передачу данных, правила взаимодействия компьютеров и программ в сети. Развитие сетей => массовость их использования => необходимость создания сетей, которые могли бы достаточноочно прочно расширяться без привлечения существенных переделок, взаимодействовать друг с другом, модернизироваться, в которых могло бы меняться ПО, добавляться новые службы => спектр моделей организации сетей (т.н. «открытых» сетей), в основе **модель системы открытых интерфейсов (OSI — Open Systems Interconnection)**, предложенная ISO — International Organization for Standardization.

- Взаимодействие в сети может осуществляться между одноимёнными (одноранговыми) уровнями. Для осуществления этого взаимодействия используются **протоколы** — это формальное описание сообщений и правил, по которым сетевые устройства (вычислительные системы) осуществляют обмен информацией => протокол обеспечивает взаимодействие в сети между различными сетевыми устройствами на одноимённых уровнях. Это правила взаимодействия одноименных, или одноранговых, уровней. Любой из уровней может содержать произвольное число протоколов, но общаться могут лишь протоколы одного уровня.
- **Интерфейс** — правила взаимодействия вышестоящего уровня с нижестоящим.
- **Служба или сервис** — набор операций, предоставляемых нижестоящим уровнем вышестоящему.
- **Стек протоколов** — перечень разноуровневых (от первого до максимального реализованного) протоколов, реализованных в системе. Стек может быть произвольной глубины, т.е. в нем, возможно, не будут представлены протоколы некоторых уровней модели ISO/OSI. При осуществлении взаимодействия

информация должна быть сначала передана с текущего на 1й уровня на данном сетевом устройстве, затем передана по коммуникационной среде, принята на другом сетевом устройстве, и, наконец, поднята до соответствующего уровня на другом сетевом устройстве.

2 сетевые ус-ва подключаются к физической среде передачи данных. В них должны быть реализованы уровни – стек протоколов. Уровни сетевых ус-в могут взаимодействовать с одноименными уровнями других устройств (сетевые протоколы), правила взаимодействия – сетевые протоколы. Для реализации пути используются сетевые интерфейсы.

**Физический уровень**, непосредственно передача неструктурированного потока двоичной информации. Для передачи используется конкретная физическая среда (кабель, радиоволны и т.п.). На данном уровне декларируется стандартизация сигналов и аппаратных соединений.

**Канальный уровень (или уровень передачи данных)**, 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 и м.б. задача внутренней адресации устройств в рамках локальной сети.

- Уровень манипулирует порциями данных (в терминах пакетов), которые называются **кадрами**. В кадрах присутствует избыточная информация для фиксации и устранения ошибок.

- основная задача канального уровня – обеспечение надёжной линии связи. На канальном уровне также **Сетевой уровень**, задачи взаимодействия сетей:

- обеспечивается управление операциями сети (в т.ч. адресация абонентов, маршрутизация),
- обеспечивается связь между взаимодействующими сетевыми устройствами.

- управление движением пакетов, и при необходимости поддерживается их буферизация.

**Транспортный уровень**, обесп корректная транспортировка данных и взаимодействие между программой-отправителем и программой-получателем данных.

- решение о выборе типа транспортных услуг (транспортировка данных с установлением виртуального канала или же без оного).

- В случае установления виртуального канала осуществляется контроль за фактом доставки и обработка ошибок (при этом взаимодействие программы-отправителя и программы-получателя обеспечивается в терминах сообщений). Много накладных расходов. Надежный, хорошо работает в условиях недетерминированных коммуникаций.

- Если же виртуальный канал не устанавливается, то уровень не несет ответственности за доставку пакетов.

- М.б. выявление и исправление ошибок при передаче.

**Сеансовый уровень**, обеспечивает управление сеансами связи.

- задачи определения активной стороны,
- подтверждения полномочий и паролей,
- 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).
- средства диагностирования и обработки ошибок.

- обеспечивает унификацию используемых в сети кодировок и форматов передаваемых данных.

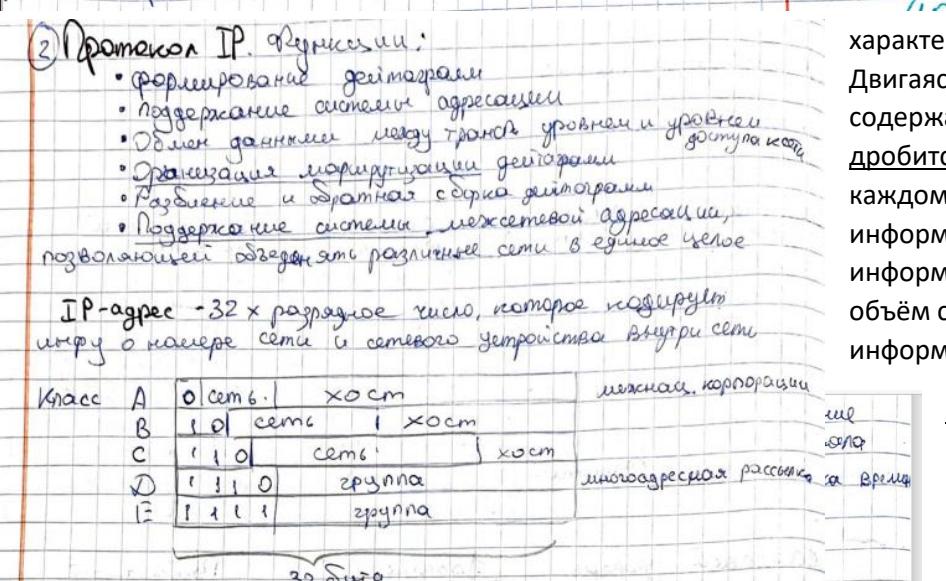
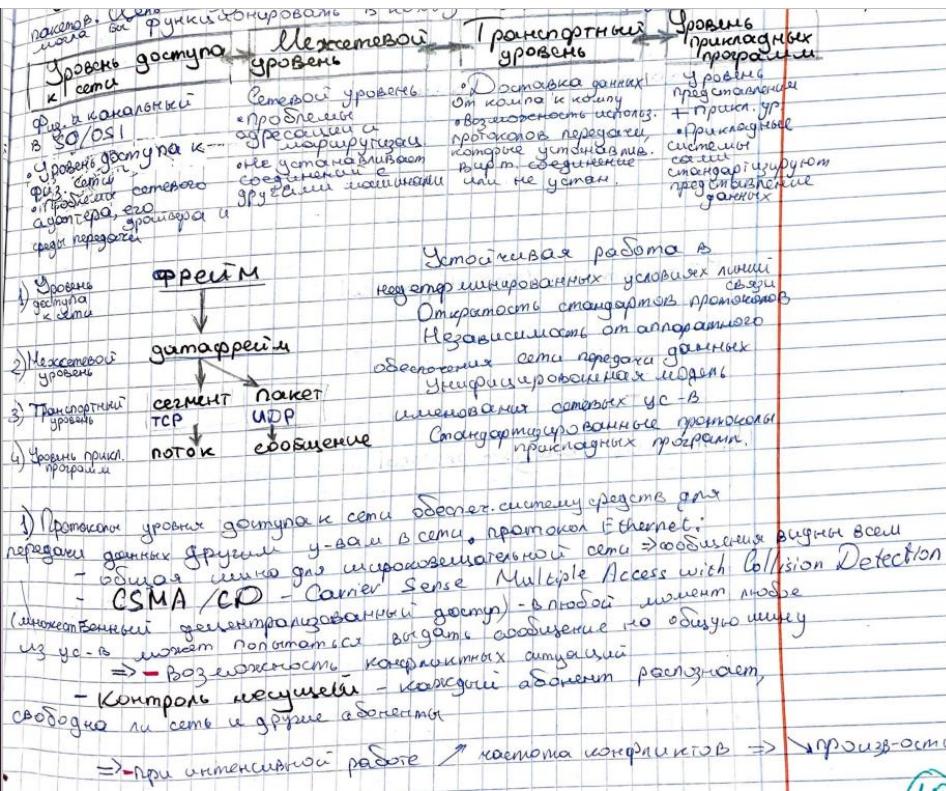
**Представительский уровень (уровень представления)**. Для унификации представления данных (кодировки).

**Прикладной уровень** (уровень прикладных программ), формализуются правила взаимодействия с прикладными системами (например, с веббраузером). Ради этого уровня выстраивается вся структура организации сетевого взаимодействия.

---

**21. Организация сетевого взаимодействия. Семейство протоколов TCP/IP, соответствие модели ISO/OSI.**

**Взаимодействие между уровнями протоколов семейства TCP/IP. IP адресация.**



Дейтограммы - пер. пакет, которым опирают межсет. уровень  
 Шлюзы - компьютеры, предназначенные для организ. обмена различными различными сетями; принадлежит  $\geq 2$  сетям и имеет  $\geq 2$  сетевых адаптеров

Задача маршрутизации: по IP-адресу получателю определяют маршрут

- 1) Организация адресации в лок. сети
- 2) Друг. // между различными сетями

$\Rightarrow$  шлюза + маршрутизаторы

Стек протоколов  $\Rightarrow$  одна машина - хост + шлюз + марш.

- NFS (Network File System) - протокол сетевой файловой системы функционирует именно в рамках лок. сети
- DNS (Domain Name Service) - один из базовых протокол

(Carrier Sense Multiple Access with Collision Detection — CSMA/CD).

Эти сетевые устройства работают в следующем режиме. Так как сеть широковещательная  $\Rightarrow$  сообщения, которые перемещаются по общейшине, видны всем адаптерам сетевых устройств. Каждый адаптер слушает

Это классическая четырехуровневая модель организации сетевого взаимодействия.

- Протоколы семейства TCP/IP основаны на сети коммутации пакетов.
- Изначально - стандарт военных протоколов в агентстве перспективных разработок DARPA министерства обороны США. Цель этой разработки - создание устойчивой децентрализованной сети, которая могла бы функционировать в коммуникационной среде, имеющей недетерминированную надёжность и производительность.

Реальные сетевые устройства связаны друг с другом только на физическом уровне. Уровни **пакетные**: на каждом уровне система оперирует порциями данных, обладающими

характеристиками соответствующего уровня.

Двигаясь от верхнего уровня модели к нижнему, содержательная информация при необходимости дробится на пакеты фиксированного размера, и к каждому из них добавляется заголовочная информация. При этом пока содержательная информация доходит до уровня доступа к сети, объём соответствующей ей служебной информации становится достаточно большим.

### Проведём сопоставление моделей TCP/IP и ISO/OSI.

#### 1. Уровень доступа к сети = физическому и канальному уровням модели ISO/OSI.

- обеспечивают систему средств для передачи данных другим устройствам в сети.

- специфицирует доступ к физической сети.  
 - решаются проблемы сетевого адаптера, драйвера сетевого адаптера и проблемы среды передачи данных.

#### Семейство протоколов Ethernet

предполагают наличие широковещательной сети, использующей единую магистраль (общую шину). Для подключаемых к этой магистрали сетевых устройств обеспечивается режим - множественный доступ с контролем несущей и обнаружением конфликтов

шину и забирает те сообщения, которые адресованы соответствующему сетевому устройству.

- **Множественный децентрализованный доступ** - в любой момент любое из устройств может попытаться выдать сообщение на общую шину. При этом возможно возникновение конфликтных ситуаций.

- **Контроль несущей** - каждый абонент, «слушая» сеть, распознает, свободна она или занята. Как только сеть становится свободной, устройство может «закидывать» очередную порцию данных. При этом устройство «слушает» как свою передачу, так и передачи других абонентов. «Бросая» сообщение в сеть, устройство способно распознать искажения, которые означают, что какое-то еще устройство также пытается послать данные в сеть => оба абонента прекращают вещание и берут тайм-аут на некоторый случайный промежуток времени (чтобы минимизировать повторные коллизии), а затем повторяют свои попытки.

- Физическая среда передачи данных: это может быть «толстый» Ethernet (к толстой медной полосе адаптеры подключаются с помощью «клёпочного» механизма), «тонкий» Ethernet (аналог домашнего телевизионного кабеля), витая пара (скрутка служит для погашения электромагнитных помех), оптоволокно, радиосигнал.

#### Недостатки:

- при интенсивной работе часто возникает ситуация, когда общая шина занята.

- Также при интенсивной работе возрастает частота конфликтов => ↓ производительности системы.

## **2. Межсетевой уровень (или internet-уровень)** = сетевой уровень модели ISO/OSI.

- решаются проблемы адресации и маршрутизации по сети.

- В отличие от сетевого уровня модели OSI не устанавливает соединений с другими машинами.

- Протокол IP 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.

- протокол IP – без установления логич соединения, и он не обеспеч обнаружение и исправление ошибок.

- Основная функция — поддержание системы межсетевой адресации (internet-адресации), позволяющей объединять различные (гетерогенные) сети в единое целое, а также организация маршрутизации.

- **IP-адрес** — это 32-разрядное число, которое кодирует информацию о номере конкретной сети и номере сетевого устройства внутри этой сети. Интерпретация адреса происходит по префиксным двоичным разрядам. Классы IP-адресов:

1. Формат класса A позволяет задавать адреса до 126 сетей с 16 млн. хостов в каждой (сетью класса A обладают ведущие межнациональные корпорации),

2. класса B — до 16382 сетей с 65536 хостами,

3. класса C — 2 млн. сетей с 254 хостами в каждой.

Среди IP-адресов классов A, B и C имеются предопределённые (например, если номер сети и номер сетевого устройства равны нулю, то считается, что это IP-адрес текущего сетевого устройства).

4. Формат класса D предназначен для многоадресной рассылки.

5. Остальные адреса используются для служебных целей.

Система TCP/IP предполагает возможность экономного расходования IP-адресов, т.е. имеется возможность выделения IP-адресов по двум стратегиям: **долговременная** (постоянные IP-адреса) и **кратковременная** (IP-адреса выделяются на время сеанса).

- Каждый из уровней взаимодействует с соседними уровнями (в соответствии с теми или иными протоколами) порциями данных, имеющими специфичные для каждого уровня названия, для межсетевого уровня пакет называется **дейтаграммой**. Одна из основных задач протокола IP – это маршрутизация дейтаграмм.

- Протокол IP подразумевает использование компьютеров, предназначенных для организации физического единения различных сетей - **шлюзов** (устройство, которое единовременно может принадлежать двум и более сетям). В общем случае шлюз имеет два и более сетевых адаптера, на которых функционирует соответствующее число стеков протоколов.

- Перед межсетевым уровнем стоит задача маршрутизации — по имеющему IP-адресу получателя необходимо определить маршрут следования пакета.

1. подзадача — это проблема организации адресации в локальной сети, в рамках которой происходит

взаимодействие => специфика межсетевого уровня позволяет относительно просто организовать взаимодействие машин в рамках одной локальной сети.

2. подзадача — это организация адресации между различными сетями => используются шлюзы, которые одновременно принадлежат разным сетям, а также маршрутизаторы, которые решают задачу, через какой шлюз необходимо отправить пакет. Одна и та же машина может быть одновременно и шлюзом, и маршрутизатором, и хостом, причем работающий за ней пользователь может не догадываться об организации локальной сети, в которой он работает.

### 3. Транспортный уровень = сеансовый и транспортный уровни модели ISO/OSI.

- 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 не всегда входят контроль за ошибками и их коррекция.

- есть возможность использования протоколов передачи, которые устанавливают виртуальное соединение или не устанавливают его.

- **TCP (Transmission Control Protocol — протокол управления передачей данных)** — это протокол, обеспечивающий установление виртуального канала => он обеспечивает последовательную передачу пакетов, контролирует доставку пакетов и отрабатывает сбои (пакет либо не доставляется, либо доставляется в целостном состоянии). Для обеспечения заявленных качеств данный протокол подразумевает отправку по сети подтверждающей инфы => пропускная способность может очень сильно падать, особенно на линиях связи с плохими техническими характеристиками.

- Для каждого полученного пакета адресат обязан отправить подтверждение о доставке.

- В данном протоколе действует поддержка времени: если через некоторое время после отправки пакета подтверждение так и не пришло, то считается, что отправленный пакет пропал, и начинается повторная посылка пропавшего пакета.

- Альтернатива - **протокол UDP (User Datagram Protocol** — протокол пользовательских дейтаграмм). UDP не обеспечивает установление виртуального соединения, подразумевает отправку пакетов по сети без гарантии их доставки (он выбрасывает пакет и сразу же «забывает» о нем)



### 4. Уровень прикладных программ = представления и уровня прикладных программ модели ISO/OSI.

- состоит из прикладных программ и процессов, использующих сеть и доступных пользователю.

- В отличие от модели OSI, прикладные программы сами стандартизуют представление данных.

- специфицирует протоколы построения распределённых сетевых приложений.

протоколы, часть которых опираются на протокол TCP, а часть — на UDP. Выбор между TCP и UDP основывается на том, насколько критична потеря информации, и на сфере реализации приложений (детерминированное или недетерминированное качество линий связи).

- Протоколы на основе TCP, обеспечивают доступ и работу с заведомо корректной информацией, причем именно в среде межсетевого взаимодействия (internet), и эти протоколы требуют корректной доставки.

**TELNET (Network Terminal Protocol)** — прикладной протокол, эмулирующий терминальное устройство (сетевой терминал); протокол межсетевого перемещения файлов FTP (File Transfer Protocol); протокол передачи почтовых сообщений SMTP (Simple Mail Transfer Protocol).

- На основе UDP - быстрые, поскольку максимально снижены накладные расходы на передачу, но они допускают наличие ошибок. Часть подобных протоколов действуют в рамках локальной сети, где качество линий связи детерминировано. **NFS (Network File System)**.

- Другая часть протоколов должна контролироваться на прикладном уровне, а с другой стороны, эти протоколы предполагают обмен очень небольшими порциями данных. **DNS (Domain Name Service)**, который позволяет мнемоническим способом именовать сетевые устройства. В частности, этот протокол осуществляет преобразования IP-адресов в мнемонические имена и обратно. Мнемонический адрес строится справа

налево перечислением доменных имён соответствующих уровней (`jaffar.mlab.cs.msu.su`). Доменные имена первого уровня определяют принадлежность данного имени по двум категориям: национальной (когда доменное имя определяет страну – `fi`, `ru`, `su`, `de` и др.) и по принадлежности компьютера к организации, занимающейся определённой деятельностью (`com`, `org`, `gov`, `net` и др.).

Свойства TCP/IP:

- 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

---

## **22. 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 ВС и обладающая правами на владение некоторым набором ресурсов ВС. Ресурсы могут декларироваться посредством различных стратегий, причем ресурс может эксклюзивно принадлежать одному единственному процессу, а может быть разделяемым, принадлежать нескольким процессам.

**Жизненный цикл процесса** — это те этапы, через которые может проходить процесс с момента его создания, в ходе его обработки и до завершения в рамках ВС.

Этапы жизненного цикла процесса:

- образование (порождение или формирование) процесса,
  - обработка (выполнение) процесса на процессоре,
  - ожидание постановки процесса на исполнение — обычно это ожидание какого-либо события: ожидание окончания обмена, ожидание выделения ресурса центрального процессора и пр.,
  - завершение процесса — этап, связанный с возвратом процессом принадлежащих ему ресурсов.
- Будем считать, что ОС обеспечивает существование процессов в двух состояниях:

1. 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 ЦП.

Отметим, что буфер ввода процессов является неотъемлемой частью пакетных систем.

2. объединяет состояния процесса, связанные с размещением процесса в **буфере обрабатываемых процессов** (БОП).

Модельные примеры совокупности состояний процессов в зависимости от типа ОС.

### **1. Пакетная однопроцессная система**

- жизненный цикл: формирование процесса и ожидание начала обработки — обработка (переход из БВП в

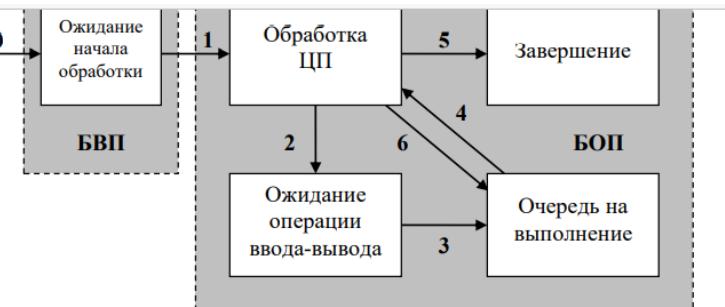


Рис. 79. Модель ОС с разделением времени. 6 — процесс прекращает обработку ЦП, но в любой момент может быть продолжен (истек квант времени ЦП, выделенный процессу). Поступает в очередь процессов, ожидающих продолжения выполнения центральным процессором (БОП).

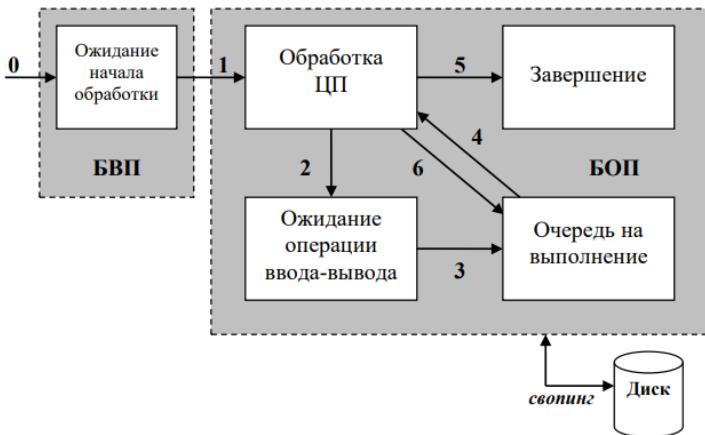


Рис. 80. Модель ОС с разделением времени (модификация). Заблокированный процесс может быть откачен (свопирован) на внешний носитель, а на освободившееся место может быть подкачен процесс с внешнего носителя, который был откачен ранее, либо взят новый.

обработки ЦП в очередь готовых на выполнение процессов. Выполнение текущего процесса (если исчерпался выделенный квант времени) и поместить процесс в указанную очередь.

- не предполагает **свопинга** - механизма откачки процесса во внешнюю память.

### 3\*. модель ОС с разделением времени и свопингом.

- еще одно состояние, характеризующее процесс, как откаченный во внешнюю память.
- в новое состояние могут переходить процессы лишь из очереди готовых на выполнение процессов, а процессы, ожидающие окончания ввода-вывода, свопироваться не могут, иначе в системе будут «зависать» заказы на обмен.
- Переход в свопинг не из состояния ожидания в/в, чтобы не зависали операции в/в: заказы на обмен внутри системы могли всегда завершиться.

---

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

Внутри процесса может быть > 1 объектов планирования.

**Процесс (или полновесный процесс)** – является объектом планирования и выполняется внутри защищённой области памяти.

**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.

- нити внутри одного полновесного процесса работают в едином адресном пространстве,
- они незащищены друг от друга.

- планированию подвергается каждая нить, планировщик может осуществлять переключение с нити на нить.

**Корректное мультипроцессирование** – защита полновесных процессов (принадл разным пользователям).

Нити внутри одного процесса принадлежат 1 процессу, ответственность за доступ одной нити в адресное пр-

БОП) — завершение процесса, освобождение системных ресурсов.

- не имеет ожиданий готовых процессов или ожиданий ввода-вывода — это однопроцессная система, которая обрабатывает один процесс, причем все обмены синхронные, и процесс никогда не откладывается.

**2. Следующая модель — пакетная мультипроцессная система**, более богатый набор состояний процесса. Есть состояние ожидания начала обработки в БВП, после которого процесс попадает в БОП на обработку ЦП. Пакетная система => обрабатываемый процесс может либо завершиться, либо перейти в состояние ожидания ввода-вывода (если процесс обращается к операции обмена).

- Когда процесс переходит из состояния обработки на процессоре, система может поставить на счет некоторый процесс либо из БВП, либо из очереди готовых на выполнение процессов в зависимости от той или иной реализованной стратегии
- после того, как процесс завершил обмен, он меняет свой статус и попадает в очередь на выполнение, из которой позже он попадет снова на выполнение.

### 3. Пакетная мультипроцессная система => модель ОС с разделением времени

- Добавилась возможность перехода из состояния

т.е. система имеет возможность прервать

во к другой нити лежит на пользователе.

**Причина использования многонитевой организации** – это минимизация накладных расходов. Уменьшение смен контекстов => мультипроцессирование внутри полновесного процесса эффективно.

Многонитевые процессы хорошо ложатся на современные многопроцессорные системы (например, SMP-системы), повышается эффективность.

механизм нитевой организации позволяет осуществлять взаимодействие нитей в рамках одного процесса, причем адресное пространство, посредством которого они взаимодействуют, остается защищенным от других процессов в системе => перед ОС, помимо управления полновесными процессами, планирования и выделения им ресурсов, возникает задача управления нитями.

Процесс включает в себя:

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

Использование нитей => оптимизация (переключения полновесных процессов – меняем ВЕСЬ контекст, т.ч настройки управления памятью).

Поддержка ОС для функционирования процессов. **Контекст процесса** - 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 процесса (т.е. это актуальное состояние регистров, настроек процессора и т.д.);
- **системная составляющая** – это структуры данных операционной системы, содержащие характеристики процесса, содержат информацию идентификационного характера (PID процесса, PID «родителя» и т.д.); информацию о содержимом регистров (РОН, индексные регистры, флаги и т.д.); а также информацию, необходимую для управления процессом (состояние процесса, приоритет и т.д.).

Отметим, что системная составляющая процесса содержит копию аппаратной составляющей, если процесс остановлен => когда процесс выполняется на процессоре, то актуальна аппаратная составляющая, когда процесс отложен — актуальна системная составляющая.

---

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

**1. Процесс - объект, зарегистрированный в таблице процессов ОС. Таблица процессов** – одна из специальных системных таблиц, которая является программной таблицей, предназначенная для регистрации всех существующих в данный момент процессов в системе. Размер ОР ОС, количество процессов в системе – системный ресурс. Таблица процессов обеспечивает 的独特ое именование процессов: она устроена **позиционным образом**, номер записи называется **идентификатором процесса** (PID – Process Identifier). 2 первые записи таблицы предопределены и используются для системных нужд. Каждая запись таблицы имеет ссылку на контекст процесса, который структурно состоит из:

- **Пользовательской (программной) составляющей** – это тело процесса: сегмент кода и сегмент данных. **Сегмент кода** содержит машинные команды и неизменяемые константы – это обычно не изменяющаяся программным способом часть тела процесса (отметим, что в принципе изменение может осуществляться посредством системных вызовов). **Сегмент данных** содержит область статических данных процесса, область разделяемой памяти (т.е. область памяти, которая может принадлежать двум и более процессам одновременно), а также область стека. На стеке в системе реализуется передача фактических параметров функциям, реализуются автоматические и регистровые переменные, а также в этой области организуется динамическая память (т.н. куча). В ОС Unix реализована возможность разделения сегмента кода для оптимизации использования ОП, механизм может иметь место в ОС только в том случае, когда сегмент кода нельзя изменить: он закрыт на запись.

- **Аппаратная составляющая** включает в себя все регистры, аппаратные таблицы процессора и т.д., характеризующие актуальное состояние процесса в момент его выполнения на процессоре. Конкретная структура аппаратной составляющей зависит от конкретного процессора; обычно она включает счётчик команд, регистр состояния процессора, аппарат виртуальной памяти, регистры общего назначения и т.д.

- **Системная составляющая** содержит системную инфу об идентификации процесса (т.е. идентификация пользователя, сформировавшего процесс), 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 атрибуты процесса, такие как:

- идентификатор родительского процесса;

- текущее состояние процесса;

• приоритет процесса. В UNIX используется **динамическая система приоритетов процесса**, которая позволяет относительно справедливо распределять ресурсы системы. Если процесс готов к исполнению и ожидает выделения ресурсов, его приоритет начинает расти. Если процесс в состоянии исполнения, его приоритет убывает.

- реальный идентификатор пользователя-владельца (ID пользователя, сформировавшего процесс);

• эффективный идентификатор пользователя-владельца (идентификатор пользователя, по которому определяются права доступа процесса к файловой системе);

• реальный идентификатор группы, к которой принадлежит владелец (идентификатор группы, к которой принадлежит пользователь, сформировавший процесс);

• эффективный идентификатор группы, к которой принадлежит владелец (идентификатор группы «эффективного» пользователя, по которому определяются права доступа процесса к файловой системе);

- список областей памяти;

• таблица дескрипторов открытых файлов процесса (именно дескрипторов, т.к. один и тот же файл может быть открыт в системе многократно); **файловый дескриптор** – это системная структура данных, отражающая актуальное состояние работы процесса с файлом;

• информация о том, какая реакция установлена на тот или иной сигнал (аппарат сигналов позволяет передавать воздействия от ядра системы процессу и от процесса к процессу);

- информация о сигналах, ожидающих доставки в данный процесс;

- сохраненные значения аппаратной составляющей (когда выполнение процесса приостановлено).

**2. Процесс - объект, порожденный системным вызовом fork(). Формированием процесса** считается запуск исполняемого файла на выполнение. **Исполняемым** считается файл, имеющий установленный соответствующий бит исполнения в правах доступа к нему, при этом файл может содержать либо исполняемый код, либо набор команд для командного интерпретатора.

- Каждый пользователь системы имеет свой идентификатор (UID — User ID).

- Каждый файл имеет своего владельца, т.е. для каждого файла определен UID пользователя-владельца.

- В системе имеется возможность разрешать запуск файлов, которые не принадлежат конкретному пользователю.

=> при запуске файла определены фактически два пользователя: **пользователь-владелец файла и пользователь, запустивший файл** (т.е. пользователь-владелец процесса), информация хранится в контексте процесса, как **реальный идентификатор** — идентификатор владельца процесса, и **эффективный идентификатор** — идентификатор владельца файла => можно подменить права процесса по доступу к файлу с реального идентификатора на эффективный идентификатор => если пользователь системы хочет изменить свой пароль доступа к системе, хранящийся в файле, который принадлежит лишь суперпользователю и только им может модифицироваться, то этот пользователь запускает процесс passwd, у которого эффективный идентификатор пользователя — это идентификатор суперпользователя (UID = 0), а реальным идентификатором будет UID данного пользователя.

- fork() обеспечивает создание копии текущего процесса. Под **системным вызовом** понимается средство ОС, предоставляемое пользователям (а точнее, процессам), посредством которого процессы могут обращаться к ядру ОС 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.

## Определение процесса в UNIX

Процесс в UNIX — объект, зарегистрированный в таблице процессов UNIX.



## Разделение сегмента кода



## Контекст процесса



## Контекст процесса



**FORK()**. При обращении процесса к системному вызову fork(), ОС создает копию текущего процесса, тело которого полностью идентично исходному процессу => система заносит в таблицу процессов новую запись => новый порождённый процесс получает уникальный идентификатор. Для этого нового процесса в системе создается контекст. Сыновний процесс наследует от родительского процесса большую часть контекста:

- окружение — при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.);
- 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;
- текущий рабочий и домашний каталоги;

Отметим, что при вызове fork() сыновний процесс не наследует:

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);
- сигналы, ждущие доставки в родительский процесс;
- время посылки ожидающего сигнала, установленное системным вызовом alarm();
- блокировки файлов, установленные родительским процессом.

По завершении системного вызова fork() каждый из процессов — родительский и порожденный, — получив управление, продолжает выполнение с одной и той же инструкции одной и той же программы, с той точки, где происходит возврат из системного вызова fork().

Вызов fork() в случае успешного завершения возвращает сыновнему процессу значение 0, а родительскому процессу — PID порожденного процесса. Может вернуть -1, если ЗАПОЛНЕНА ТАБЛИЦА ПРОЦЕССОВ.

**EXEC()**. Системные вызовы обеспечивают смену тела текущего процесса. В это семейство входят вызовы, у которых в названии префиксная часть обычно представлена как exec, а суффиксная часть служит для уточнения сигнатуры того или иного системного вызова. В качестве иллюстрации приведем определение системного вызова exec(). Через параметры вызова передается указание на имя некоторого исполняемого файла, а также набор аргументов, которые передаются внутрь при запуске этого исполняемого файла. При выполнении данных системных вызовов происходит замена тела текущего процесса на тело, образованное в результате загрузки исполняемого файла, и управление передается на точку входа в новое тело.

```
#include <unistd.h>
int execl(const char *path, char *arg0,...);
int execlp(const char *file, char *arg0,...);
int execle(const char *path, char *arg0,..., const char **env);
int execv(const char *path, const char **arg);
int execvp(const char *file, const char **arg);
int execve(const char *path, const char **arg, const char **env);
```

Заметим, что выполнение «нового» тела происходит в рамках уже существующего процесса, т.е. после вызова exec() сохраняется идентификатор процесса, и PPID, таблица дескрипторов файлов (за исключением, быть может, файлов, открытых в специальном режиме), приоритет и большая часть других атрибутов процесса. Фактически происходит замена сегмента кода и сегмента данных. Изменяется системная информация, которая должна корректироваться при смене тела процесса:

- режимы обработки сигналов: для сигналов, которые перехватывались, после замены тела процесса будет установлена обработка по умолчанию, т.к. в новой программе могут отсутствовать указанные функции-обработчики сигналов
- эффективные идентификаторы владельца и группы могут измениться, если для новой выполняемой программы установлен s-бит
- перед началом выполнения новой программы могут быть закрыты некоторые файлы, ранее открытые в процессе. Это касается тех файлов, для которых при помощи системного вызова fcntl() был установлен флаг closeon-exec. Соответствующие файловые дескрипторы будут помечены как свободные.

**\_EXIT()**. Системный вызов, сопутствует базовому системному вызову управления процессами - завершении процесса. Процесс может из-за:

1. Первая причина связана с возникновением в процессе сигнала. Сигнал - программный аналог прерывания,
2. Вторая причина связана с обращением к системному вызову завершения процесса. При этом обращение может быть **явным**, когда в теле программы встречается обращение к системному вызову \_exit(), или **неявным**, если происходит выполнение оператора return языка С внутри функции main() или выход на закрывающую скобку функции main(). Компилятор заменит действие оператора return обращением к системному вызову \_exit().

```
#include <unistd.h>
void _exit(int status);
```

Системный вызов \_exit() никогда не завершается неудачно. С помощью параметра status процесс может передать породившему его процессу информацию о статусе своего завершения - т.н. программный код завершения процесса. Сиюминутно процесс не может завершиться => переходит в переходное состояние — т.н. **состояние зомби**. Совокупность действий в системе:

1. корректно освобождаются ресурсы (закрываются все открытые дескрипторы файлов, освобождаются сегмент кода и сегмент данных процесса и пр.).
2. освобождается большая часть контекста процесса, однако сохраняется запись в таблице процессов и та часть контекста, в которой хранится статус завершения процесса и статистика его выполнения.
3. все сыновние процессы, чьи родители завершились, усыновляются процессом с номером 1.
4. процессу-предку от данного завершающегося процесса передается сигнал SIGCHLD, но в большинстве случаев его игнорируют. Процесс-предок имеет возможность получить информации о статусе завершения/приостановки своего потомка.

## WAIT() – получение информации о состоянии своего потомка

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

```
#include <sys/wait.h>
```

```
pid_t wait(int *status)
```

1. если к моменту обращения к этому системному вызову какие-то сыновние процессы уже завершились, то процесс получит информацию об одном из этих процессов.
2. если у процесса нет сыновних процессов, то, обращаясь к системному вызову wait(), процесс получит -1.
3. если у процесса имеются сыновние процессы, но ни один из них не завершился, то при обращении к указанному системному вызову данный отцовский процесс будет блокирован до завершения любого из своих сыновних процессов (т. е., если процесс хочет получить информацию о завершении каждого из своих потомков, он должен несколько раз обратиться к вызову wait()).

- В случае успешного завершения возвращается идентификатор PID завершившегося процесса, или же -1 — в случае ошибки или прерывания.

- через параметр status передается указатель на целочисленную переменную, в которой система возвращает процессу информацию о причине завершения сыновнего процесса.

- Данный параметр содержит в старшем байте код завершения процесса-потомка (пользовательский код завершения процесса), передаваемый в качестве параметра системному вызову \_exit(), а в младшем байте — индикатор причины завершения процесса-потомка, устанавливаемый ядром ОС Unix (системный код завершения процесса).

- Системный код завершения хранит номер сигнала, приход которого в сыновний процесс вызвал его завершение.

- wait() не всегда отрабатывает на завершении сыновнего процесса. В случае если отцовский процесс производит трассировку сыновнего процесса, то посредством системного вызова wait() можно фиксировать факт приостановки сыновнего процесса, причем сыновний процесс после этого может быть продолжен (т.е. не всегда он должен завершиться, чтобы отцовский процесс получил информацию о сыне).

- возможность изменить режим работы wait() таким образом, чтобы отцовский процесс не блокировался в ожидании завершения одного из потомков, а сразу получал соответствующий код ответа.

- после передачи информации родительскому процессу о статусе завершения все структуры, связанные с процессом-зомби, освобождаются, и запись о нем удаляется из таблицы процессов => переход в состояние зомби необходим именно для того, чтобы процесс-предок мог получить информацию о судьбе своего завершившегося потомка, независимо от того, вызвал он wait() до или после его завершения.

- При завершении процесса отцом для всех его потомков становится процесс с идентификатором 1. Он и осуществляет системный вызов wait(), тем самым, освобождая все структуры, связанные с потомками зомби.

## Жизненный цикл процессов



## Начало работы системы.

1. При включении компьютера управление передается на предопределенный адрес ОЗУ, начиная с которого начинается постоянное запоминающее устройство, там – **аппаратный загрузчик**.
2. Выбирает приоритетное готовое к работе системное устройство (обычно жесткий диск), читает предопределенный блок с этого устройства (обычно 0). Несколько устройств для того, чтобы можно было работать с устройством, если одно из них вышло из строя.
3. Загрузчик ЮНИКС знает структуру системного диска, знает, что в корне ФС есть исполняемый файл с ядром

ОС. Обычно название связано с название ОС. Загрузчик загружает его в физическую память и передает управление на точку входа. Ядро приводит аппаратуру и ПО в каноническую форму: инициирует программные устройства (часы и тд). Ядро иниц системные структуры данных (таблицы процессов и тд).

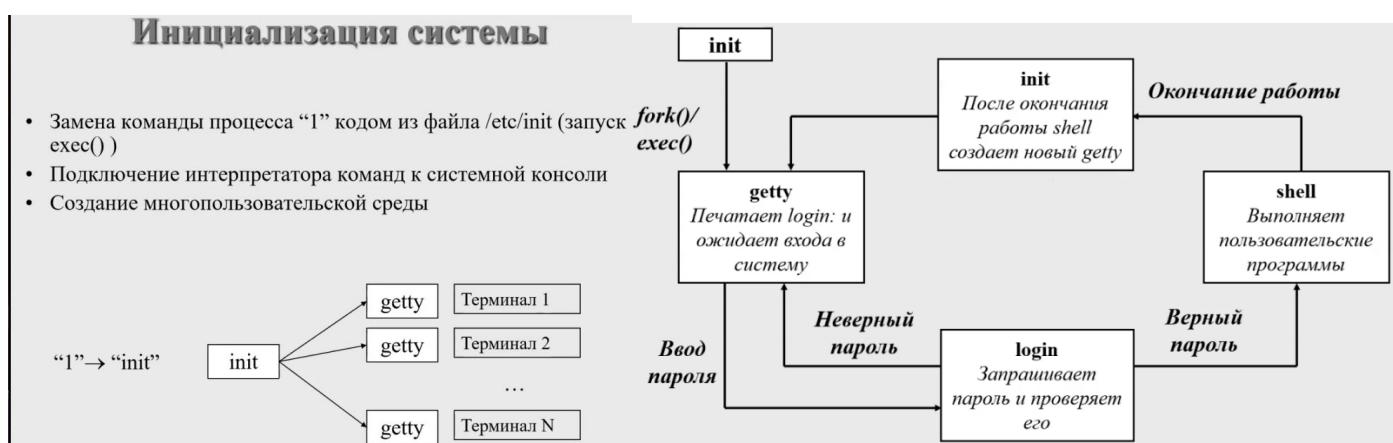
4. Формирование процесс с номером 0 (запись инфы в 0-ю строку таблицы), у него нету контекста, кода и тд. Соответствует ядру ОС. Процесс существует всегда.

5. Создается 1й процесс (не через форк, просто заполняет таблицу), он корректен с точки зрения структуры: у него есть контекст и тд. Тело заменяется на init(). Он существует всегда.

6. Init смотрит системные данные и запускает режим однопольз/многополз). Многопользовательский – обращается к текстовому файлу, где зарег. все терминалы, запускает по форк–экзек getty(), он запросит ввод.

7. Гетти последовательно видоизменяется. Он запрашивает имя и пароль. Passwd – файл регистрации пользователей. Для каждого пользователя есть строка с паролем (в зашифрованном виде), домашний каталог, который становится текущим при успешном входе пользователя в аккаунт. Там имя исп фала. Запускается интерпретатор команд.

8. С точки зрения интерпретатора команд, сеанс работы пользователя представляется в виде обменов с файлом (т.е. в виде операций чтения и записи). Работа пользователя с системой заканчивается закрытием файла – подачей EOF.



Взаимодействующие процессы могут использовать часть своих ресурсов совместно, и при этом работа 1 процесса оказывает влияние на работу другого процесса.

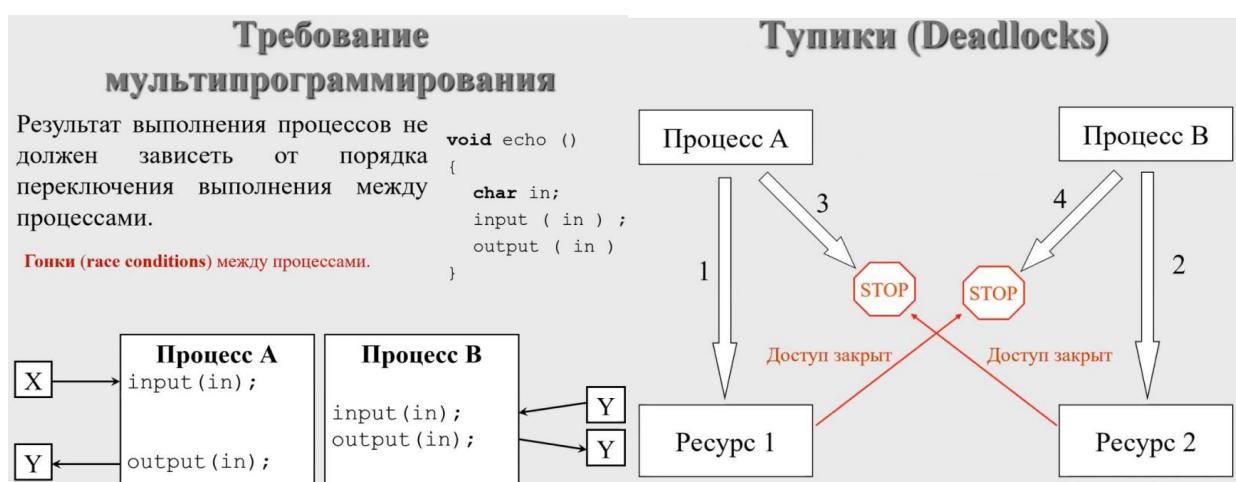
## 25. Взаимодействие процессов. Разделяемые ресурсы. Критические секции. Взаимное исключение. Тупики.

Результат работы процессов не должен зависеть от порядка переключения между процессами. Результат зависит => гонки.

**Взаимное исключение** – если 1 процесс работает с критической секцией, то другие не имеют к ней доступа.

1. **Тупики** – несколько процессов, работают в режиме взаимного исключения с разделяемыми ресурсами, из-за своей активности друг друга блокируют.

2. Блокирование. Активность одного процесса не позволяет работать другим процессам.



Процессы называются **параллельными**, если их выполнение хотя бы частично перекрывается по времени. Т.е. можно говорить, что все процессы, находящиеся в буфере обрабатываемых процессов, являются параллельными, т.к. в той или иной степени времена их выполнения перекрываются друг с другом. Речь идет лишь о **псевдопараллелизме**, поскольку реально на процессоре может исполняться только один процесс.

**Независимые процессы** используют независимые множества ресурсов; т.е. множества ресурсов, которые принадлежат независимым процессам, в пересечении дают пустое множество.

**Взаимодействующие процессы** совместно используют ресурсы, и выполнение одного процесса может оказывать влияние на результат другого процесса, участвующего в этом взаимодействии.

Совместное использование ресурса ВС двумя и более параллельными процессами, когда каждый из процессов некоторое время владеет этим ресурсом, называется **разделением ресурса**. Разделению подлежат как аппаратные, так и программные (виртуальные) ресурсы.

Разделяемый ресурс, который в каждый момент времени может быть доступен только одному из взаимодействующих процессов, называется **критическим ресурсом**. (Таковыми ресурсами могут быть как внешнее устройство, так и некая переменная, значение которой может изменяться разными процессами.) Часть программы (фактически набор операций), в рамках которой осуществляется работа с критическим ресурсом, называется **критической секцией** (или критическим интервалом).

Проблемы мультипрограммирования связаны с обеспечением корректного доступа к разделяемым ресурсам. Выделяют две задачи ОС:

1. распределение ресурсов между процессами
2. организация корректного доступа (т.е. организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов). В частности, под организацией корректного доступа может пониматься требование, декларирующее необходимость обеспечения независимости работы параллельных процессов от порядка и интенсивности доступа этих процессов к разделяемым ресурсам.

---

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

Средства, позволяющих организовать работу с критическими ресурсами, т.е. организовать такой способ работы с разделяемым ресурсом, при котором в любой момент времени к нему может иметь доступ не более 1 процесса.

**Семафоры Дейкстры** — это формальная модель организации доступа, базируется на следующей концепции.

- Имеется специальный тип данных — т.н. **семафор**.
- Переменные типа семафор могут принимать целочисленные значения.
- Над этими переменными определены следующие **атомарные (неделимые) операции**: их выполнение не может быть прервано прерыванием.
- Операция **down(S)** проверяет значение семафора S и, если оно больше 0, 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, блокированные ранее при выполнении down на этом семафоре, то один из них разблокируется и завершает выполнение операции down, т.е. вновь уменьшает значение семафора.
- Выбор процесса для разблокирования никак не оговаривается. Ещё раз отметим, что операции up и down являются атомарными (неделимыми), т
- **Двоичный семафор** — максимальное значение которого равно 1, обеспечивает взаимное исключение.
- Существуют различные программные реализации этих операций, но в них атомарность не всегда присутствует.

В отличие от семафора, **монитор Хоара** является высокоуровневой конструкцией (можно говорить, что это конструкция уровня языка программирования), реализация которой поддерживается системой программирования (компилятором).

- Монитор — это специализированный модуль, включающий в себя совокупность процедур и функций, а также структуры данные, с которыми работают эти процедуры и функции. Свойства:

1. структуры данных монитора доступны только через обращения к процедурам или функциям этого монитора (т.е. монитор представляет собой некоторый аналог объекта в объектно-ориентированных языках и реализует инкапсуляцию данных);

2. считается, что процесс занимает (или входит в) монитор, если он вызывает одну из процедур или функций монитора;

3. в каждый момент времени внутри монитора может находиться не более 1 процесса. Если процесс пытается попасть в монитор, в котором уже находится другой процесс, то (в зависимости от используемой стратегии) он либо получает отказ, либо блокируется, становясь в очередь.

=> чтобы защитить разделяемые структуры данных, их достаточно поместить внутрь монитора вместе с процедурами, представляющими критические секции для обработки этих структур данных.

- Монитор представляет собой конструкцию языка программирования и компилятору известно о том, что входящие в него процедуры и данные имеют особую семантику, поэтому первое условие может проверяться еще на этапе компиляции, кроме того, код для процедур монитора тоже может генерироваться особым образом, чтобы удовлетворялось третье условие.

- организация взаимного исключения возлагается на компилятор => количество программных ошибок, связанных с организацией взаимного исключения, сводится к минимуму

Семафоры и мониторы являются средствами организации работы в однопроцессорных системах либо в многопроцессорных системах с общей памятью. Для многопроцессорных систем с распределенной памятью - механизм передачи сообщений (универсальный, предоставляет, как средства организации взаимодействия между процессами, так и средства синхронизации).

- Основан на 2x функциональных примитивах: **send** (отправить сообщение) и **receive** (принять сообщение).

Данные операции можно разделить по трем характеристикам:

### 1. Синхронизация.

- Операции посылки/приема сообщений могут быть блокирующими и неблокирующими.

- 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.

- неблокирующие операции.

### 2. 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 (FIFO, LIFO и т.д.). Этим пулом может выступать очередь сообщений (FIFO) или почтовый ящик, в котором может быть реализована любая модель доступа.

### 3. Формат сообщения.

Механизм передачи сообщений реализуется на базе интерфейсов передачи сообщений MPI. На основе этих интерфейсов строятся почти все кластерные системы (т.е. системы с распределенной памятью), а также MPI может работать и в системах с общей памятью.

+ Гибкая и широкоприменимая концепция.

---

## 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.

Итак, данная задача иллюстрирует модель доступа равноправных процессов к общему ресурсу, и ставится вопрос, как организовать корректную работу такой системы. Рассмотрим простейшее решение данной задачи, использующее семафоры. Когда один из философов хочет есть, он берет вилку слева от себя, если она в наличии, а затем — вилку справа от себя. Закончив есть, он возвращает обе вилки на свои места.

```
{  
    while(TRUE)  
    {  
        Think();  
        /* философ берёт обе вилки или блокируется */  
        TakeForks(i);  
        Eat();  
        PutForks(i);  
    }  
}  
  
/* получение вилок */  
void TakeForks(int i)  
{  
    /* вход в критическую секцию */  
    down(&mutex);  
    state[i] = HUNGRY;  
    Test(i);  
    /* выход из критической секции */  
    up(&mutex);  
    down(&s[i]);  
}  
  
/* освобождение вилок */  
void PutForks(int i)  
{  
    /* вход в критическую секцию */  
    down(&mutex);  
    state[i] = THINKING;  
    Test(LEFT);  
    Test(RIGHT);  
    /* выход из критической секции */  
    up(&mutex);  
}  
/* функция проверки возможности получения вилок —  
проверяется состояние соседей данного философа */  
void Test(int i)  
{  
    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.

```

/* переопределение типа семафор */
typedef int semaphore;
/* семафор для доступа в критическую секцию - контроль за
доступом к «rc» (разделямый ресурс) */
semaphore mutex = 1;
/* семафор для доступа к базе данных */
semaphore db = 1;
/* количество читателей внутри хранилища */
int rc = 0;

/* процесс-читатель */

```

128

```

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

/* процесс-писатель */
void Writer(void)
{
    while(TRUE)
    {
        ThinkUpData(); /* некритическая секция */
        down(&db); /* получить эксклюзивный доступ к данным*/
        WriteDataBase(); /* 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.

Данная задача является иллюстрацией модели клиент-сервер с ограничением на длину очереди клиентов.

Понадобится 3 семафора: `customers` — подсчитывает количество посетителей, ожидающих в очереди, `barbers` — статус парикмахера (0 – спит/занят, 1 – готов к работе) 1 и `mutex` — используется для синхронизации доступа к разделяемой переменной `waiting`. Переменная `waiting`, как и семафор `customers`, содержит количество посетителей, ожидающих в очереди. Эта переменная используется в программе для того, чтобы иметь возможность проверить, имеется ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется. Заметим, что как и в предыдущем примере, эта переменная является разделяемым ресурсом, и доступ к ней охраняется семафором `mutex`.

```

/* количество стульев в комнате ожидания */
#define CHAIRS 5
/* переопределение типа СЕМАФОР */
typedef int semaphore;
/* наличие посетителей, ожидающих парикмахера */
semaphore customers = 0;
/* состояние парикмахера: спит/занят или готов к работе (0 или
1)*/
semaphore barbers = 0;
/* семафор для доступа в критическую секцию - контроль за
доступом к переменной waiting */
semaphore mutex = 1;
/* количество ожидающих посетителей */

```

<sup>1</sup> В принципе возможно расширение задачи для случая N парикмахеров, в этом случае незначительными коррекциями программ, barbers – количество свободных парикмахеров.

130

```

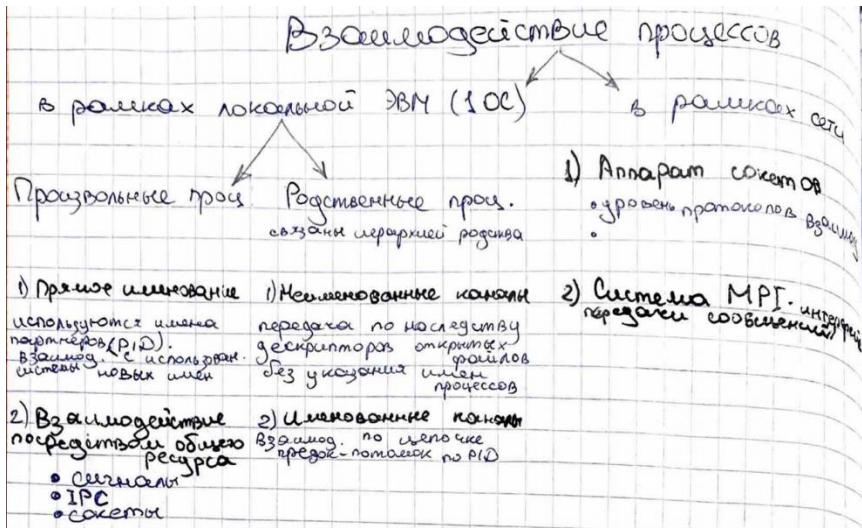
int waiting = 0;

/* Брадобрей */
void Barber(void)
{
    while(TRUE)
    {
        /* если customers == 0, т.е. посетителей нет, то
        заблокируемся до появления посетителя */
        down(&customers);
        down(&mutex);
        /* получаем доступ к waiting */
        /* уменьшаем кол-во ожидающих клиентов */
        waiting = waiting - 1;
        up(&barbers); /* парикмахер готов к работе */
        up(&mutex); /* освобождаем ресурс waiting */
        CutHair(); /* процесс стрижки */
    }
}

/* Посетитель */
void Customer(void)
{
    down(&mutex); /* получаем доступ к waiting */
    if(waiting < CHAIRS) /* есть место для ожидания */
    {
        /* увеличиваем кол-во ожидающих клиентов */
        waiting = waiting + 1;
        /* если парикмахер спит, это его разбудит */
        up(&customers);
        up(&mutex); /* освобождаем ресурс waiting */
        /* если парикмахер занят, переходим в состояние
        ожидания, иначе – занимаем парикмахера */
        down(&barbers);
        /* занять место и перейти к стрижке */
        GetHaircut();
    }
    else
    {
        /* нет свободного кресла для ожидания – придется уйти */
        up(&mutex);
    }
}

```

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



Взаимодействие родственных процессов: в ОС Unix можно передавать по наследству от отца сыну дескрипторы открытых файлов. В данном случае именование будет **неявным**, поскольку не указываются непосредственно имена процессов.

Процессы используют PID партнеров => при завершении 1 процесса, на его место встанет другой процесс => некорректно.

**Неименованный канал** — это некоторый ресурс, наследуемый

сыновьями процессами, причем этот механизм может быть использован для организации взаимодействия произвольных родственников (т.е., условно говоря,

- Неименованные каналы — пример симметричного взаимодействия, т.е., несмотря на то, что ресурс неименованного канала передается по наследству, взаимодействующие процессы в общем случае, абстрагируясь от семантики программы, имеют идентичные права.

- несимметричная модель «главный–подчиненный». В этом случае среди взаимодействующих процессов можно выделить процессы, имеющие больше полномочий, чем у остальных, 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), 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. Альтернатива именованным каналам.

**Аппарат сокетов** — унифицированное средство организации взаимодействия. На сегодняшний момент сокеты — это не столько средства ОС Unix, сколько стандартизированные 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, задача организовать взаимодействие процессов, находящихся на разных машинах под управлением различных ОС. Та же проблема именования процессов в рамках сети решается достаточно просто. Пусть у нас есть две машины, имеющие сетевые имена А и В. Пусть на этих машинах работают процессы P1 и P2 соответственно. Тогда, чтобы именовать процесс в сети, достаточно использовать связку «сетевое имя машины + имя процесса на этой машине». В нашем примере это будут пары (A–P1) и (B–P2) => проблема: рамках сети могут взаимодействовать машины, находящиеся под управлением ОС различного типа, система именования должна быть построена так, чтобы обеспечить возможность взаимодействия произвольных машин, т.е. это должно быть стандартизованным (унифицированным) средством.

1. **Аппарат сокетов** можно рассматривать как базовое средство организации взаимодействия, механизм лежит на уровне протоколов взаимодействия. Топология взаимодействующих процессов зависит от задачи (можно организовать общение одного сокета со многими, можно установить связь один–к–одному и т.д.).  
2. **Система MPI** (интерфейс передачи сообщений) также является достаточно распространенным средством организации взаимодействия в рамках сети. Эта система иллюстрирует механизм передачи сообщений. Система MPI может работать на локальной машине, в многопроцессорных системах с распределенной памятью (т.е. может работать в кластерных системах), в сети в целом (в частности, в т.н. GRID-системах). Далее речь пойдет о конкретных средствах взаимодействия процессов (как в ОС Unix, так и в некоторых других).

**Сигналы** - средство уведомления процесса о произошедшем событии. В современном Юниксе их около 30 и все они описаны в `<signal.h>`

Есть три варианта реакции на пришедший сигнал:

- дефолтная обработка
- игнорирование
- кастомный обработчик который программист задает сам

(на некоторые сигналы запрещено менять обработчик, например на SIGKILL и SIGSTOP)

Обработка прихода сигнала во время системного вызова(то есть попытка прервать атомарную операцию) зависит от конкретной реализации

```
int kill(pid_t pid, int sig)
```

Отправка сигнала по определенному pidу. Возвращает -1 и errno в случае ошибки, иначе 0. Если pid равен 0, то сигнал пошлется всей группе, если -1, то всем процессам до которого данный процесс сможет достучаться с текущим уровнем прав(подробнее читай в мане). Если sig равен 0, то kill() выполнит все необходимые действия для проверки существования сигнала, но сам сигнал не пошлет(так можно проверить, существует ли еще процесс).

```
void signal(*signal(int sig, void (*disp) (int))) (int)
```

Устанавливаем новый обработчик на сигнал

sig - номер сигнала для реакции

disp - функция-обработчик сигнала, или SIG\_DFL(дефолтная обработка) или

SIG\_IGN(игнорирование сигнала)

В ранних версиях Юникса обработчик сбрасывался после обработки одного сигнала(по стандарту System V). В современном Юниксе такого нет

**Аппарат сигналов** позволяет одним процессам оказывать воздействия на другие процессы. Сигналы могут рассматриваться как средство уведомления процесса о наступлении некоторого события в системе. Аналогия с аппаратом прерываний, поскольку последний есть также уведомление системы о том, что в ней произошло некоторое событие. Прерывание вызывает определенную детерминированную последовательность действий системы, точно так же приход сигнала.

UNIX система позволяет осуществлять передачу взаимодействий от имени ОС к процессу/ от процесса к процессу. Взаимодействия передаются асинхронно, Сигналы, которые приходят в процесс от имени ОС, уведомляющие о событии в данном процессе/системе.

Процесс получает этот сигнал. З Варианта обработки сигнала:

1. Действие по умолчанию. Часто процесс начинает завершаться с кодом, равным номеру сигнала.
2. Игнорирование, приход сигнала не вызовет ничего.
3. Внутри процесса заранее объявлено, что в случае прихода конкретного сигнала/сигналов обращение к некоторой заранее предопределенной функции – обработчику.

Чтобы установить реакцию процесса на приходящий сигнал, используется **системный вызов signal()**.

`void (*signal ( int sig, void (*disp) (int))) (int);`

sig - номер сигнала, реакцию на приход которого надо установить. Второй аргумент disp определяет новую реакцию на приход указанного сигнала. То есть disp – это либо определенная пользователем функция-обработчик сигнала, либо одна из констант: SIG\_DFL и SIG\_IGN. При успешном завершении системный вызов возвращает указатель на предыдущий обработчик данного сигнала (он может использоваться процессом, например, для восстановления прежней реакции на сигнал).

- Возвращает предыдущий обработчик
- Сигналы, которые нельзя игнорировать: SIGKILL.
- Сигналы, которые можно игнорировать:
- Если мы используем функцию – обработчик, и этот сигнал позволяет устанавливать обработку, она вызывается асинхронно в тот момент, когда приходит сигнал, для которого эта функция определена.
- Возврат из функции осуществляется в точку прихода сигнала – аналог прерывания.

Для отправки сигнала процессу в ОС Unix имеется **системный вызов kill()**. #include INT KILL (PID\_T PID, INT SIG);

- Первый параметр вызова – идентификатор процесса, которому посыпается сигнал (в частности, процесс может послать сигнал самому себе). Если значение этого параметра есть 0, сигнал будет передан всем процессам, которые принадлежат той же группе, что и процесс, посыпающий сигнал, за исключением процессов с идентификаторами 0 и 1.
- Во втором параметре передается номер посыпаемого сигнала. Если этот параметр равен 0, то будет выполнена проверка корректности обращения к kill() (в частности, существование процесса с идентификатором pid), но никакой сигнал в действительности посыпаться не будет.
- Если процесс-отправитель не обладает правами привилегированного пользователя, то он может отправить сигнал только тем процессам, у которых реальный или эффективный идентификатор владельца процесса совпадает с реальным или эффективным идентификатором владельца процесса-отправителя.
- При успешном завершении системный вызов kill() возвращает 0, иначе возвращается -1.

---

### 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 ОС, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы, т.е. эта область памяти является разделяемым ресурсом.

- Для доступа к неименованному каналу система ассоциирует с ним два файловых дескриптора: для чтения и для записи информации в канал. Единственная возможность использовать канал – это те файловые дескрипторы, которые с ним ассоциированы
- Дисциплина доступа к информации – **FIFO => неприменимы** системные вызовы перемещения файлового

указателя.

- Предельный размер канала декларируется параметрами настройки ОС
- неименованный канал существует в системе, до тех пор, пока существуют процессы, его использующие.
- Системный вызов pipe(). #include int pipe(int \*fd)
- Система выделяет в буфере ресурс неименованный канал и связывает этот ресурс двумя файловыми дескрипторами.
- Нулевой элемент массива fd[0] – читаем, fd[1] – пишем.
- Причины возврата -1: нет свободных файловых дескрипторов. Для каждого процесса определено предельное кол-во файлов, которые могут быть в нем открыты. При обращении к системному вызову должно быть хотя бы 2 свободных строки.
- Если из канала читается порция данных, меньшая по размеру, чем содержимое канала. Нет синхронизации по количеству чтения и записи.
- Если мы читаем из канала порцию данных, превосходящую по объему, и в системе есть хотя бы 1 открытый дескриптор, который может писать в этот канал, то данные, которые присутствуют в канале, будут перенесены в читающий процесс, а он будет заблокирован. Условия разблокировки (оба должны быть выполнены):
  1. В канале появилась недостающая порция данных.
  2. В системе закроется последний дескриптор, который может писать в канал.
- Если мы хотим писать большую порцию данных, большую размера свободного места: мы запишем ту порцию, равную объему свободного пространства, процесс будет ожидать освобождения, процесс заблокируется, пока в канале не появится место, снова запишет, и тд (аналогично чтению).
- В стандартной ситуации (при отсутствии переполнения) система гарантирует атомарность операции записи, т. е. при одновременной записи нескольких процессов в канал их данные не перемешиваются.

При порождении процесса в ОС Unix он заведомо получает три открытых файловых дескриптора:

1. дескриптор стандартного ввода (этот дескриптор имеет нулевой номер), обычно данные с клавиатура.
2. дескриптор стандартного вывода (имеет номер 1), обычно отображается на дисплей монитора
3. дескриптор стандартного потока ошибок (имеет номер 2).

**int pipe(int fd[2])**

Создает неименованный канал из двух дескрипторов, fd[0] для записи, fd[1] для чтения(запомнить легко - соответствие с 0 и 1 дескрипторами). Работа с этими дескрипторами не отличается от работы с дескрипторами регулярных файлов за исключением, наверное, lseek(). Возвращает 0 при успехе, -1 и errno при ошибке.

Ряд нюансов:

- при чтении из пустого процесс блокируется(если не открыт как O\_NONBLOCK)
- если пытаемся прочитать порцию меньшую чем доступна, то читаем сколько указано, остальное остается в канале
- если пытаемся прочитать больше чем есть в канале, то будет изъято доступное количество
- (по отношению к предыдущему пункту) если при прочтении дескриптор записи не закрыт, то читающий процесс блокируется. Если дескриптор записи закрыт, то в канал поступит EOF, и процесс разблокируется и продолжит читать дальше.
- если пишем больше чем доступно, то записывается возможное количество и пишущий процесс блокируется пока не появится достаточно места чтобы записать
- если пишем в канал у которого закрыт дескриптор чтения, получаем SIGPIPE в ебало

Обмен данных по именованному каналу может происходить не только между отцом-сыном, но и в принципе между родственными процессами.

---

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

Необходимо для организации средств отладки, когда есть процесс-отладчик и отлаживаемый процесс:

- отладчик может в произвольные моменты времени останавливать отлаживаемый процесс и, когда отлаживаемый процесс остановлен, осуществлять действия по его отладке: просматривать содержимое тела процесса, при необходимости корректировать и т.д.

- возможность установки контрольных точек в отлаживаемом процессе.

=> 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.

Схема взаимодействия процессов путем трассировки:

- 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. Пока трассируемый процесс приостановлен, 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 (либо, при пошаговом выполнении, для выполнения одной инструкции).

```
#include <sys/ptrace.h>
```

```
int ptrace(int cmd, int pid, int addr, int data);
```

cmd обозначает код выполняемой команды,

pid — идентификатор процесса-потомка (который мы хотим трассировать),

addr — некоторый адрес в адресном пространстве процесса-потомка, и, наконец,

data — слово информации.

Решаемые задачи:

1. подчиненный процесс может разрешить родительскому процессу проводить свою трассировку: для этого в качестве параметра cmd необходимо указать команду PTRACE\_TRACEME.

2. процесс отладчик может манипулировать отлаживаемым процессом, используя значения параметра cmd.

ptrace() позволяет выполнять:

1. читать данные из сегмента кода и сегмента данных отлаживаемого процесса;

2. читать некоторые данные из контекста отлаживаемого процесса (в частности, имеется возможность чтения содержимого регистров);

3. осуществлять запись в сегмент кода, сегмент данных и в некоторые области контекста отлаживаемого процесса (в т.ч. модифицировать содержимое регистров). Производить чтение и запись данных (а также осуществлять большинство управляющих команд над отлаживаемым процессом) можно лишь тогда, когда трассируемый процесс приостановлен;

4. продолжать выполнение отлаживаемого процесса с прерванной точки или с предопределенного адреса сегмента кода;

5. исполнять отлаживаемый процесс в пошаговом режиме. **Пошаговый режим** — это режим, обеспечиваемый аппаратурой компьютера, который вызывает прерывание после исполнения каждой машинной команды отлаживаемого процесса (т.е. после исполнения каждой машинной команды процесс приостанавливается).

cmd = **PTRACE\_TRACEME** — ptrace() с таким кодом операции сыновний процесс вызывает в самом начале своей работы, позволяя тем самым трассировать себя. Все остальные обращения к вызову ptrace() осуществляет процесс-отладчик.

cmd = **PTRACE\_PEEKDATA** — чтение слова из адресного пространства отлаживаемого процесса по адресу addr, ptrace() возвращает значение этого слова.

cmd = **PTRACE\_PEEKUSER** — чтение слова из контекста процесса. Речь идет о доступе к пользовательской составляющей контекста данного процесса, сгруппированной в некоторую структуру, описанную в заголовочном файле . В этом случае параметр addr указывает смещение относительно начала этой структуры. В этой структуре размещена такая информация, как регистры, текущее состояние процесса, счетчик адреса и так далее. ptrace() возвращает значение считанного слова.

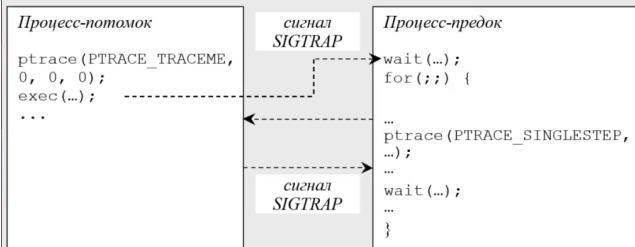
cmd = **PTRACE\_POKEDATA** — запись данных, размещенных в параметре data, по адресу addr в адресном

пространстве процесса-потомка.

cmd = **PTRACE\_POKEUSER** — запись слова из data в контекст трассируемого процесса со смещением addr.

Таким образом можно, например, изменить счетчик адреса

## Общая схема трассировки процессов



```
if ((pid = fork()) == 0)
{
    ptrace(PTRACE_TRACEME, 0, 0, 0);
    /* сыновний процесс разрешает трассировать себя */
    exec("трассируемый процесс", 0);
    /* замещается телом процесса, который необходимо
     * трассировать */
}
else
{
    /* это процесс, управляющий трассировкой */
    wait((int) 0);
    /* процесс приостанавливается до тех пор, пока от
     * трассируемого процесса не придет сообщение о том, что
     * он приостановился */
    for(;;)
    {
        ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
        /* возобновляем выполнение трассируемой программы
         */
        wait((int) 0);
        /* процесс приостанавливается до тех пор, пока от
         * трассируемого процесса не придет сообщение о том,
         * что он приостановился */
    }
}
```

161



```
...
ptrace(cmd, pid, addr, data);
/* теперь выполняются любые действия над
 * трассируемым процессом */
...
}
```

Организация контрольных точек (для отладчика):

1. Считываем из сегмента кода информацию по адресу A.
2. Сохраняем ее в таблице контрольных точек.
3. По адресу контрольной точки записываем команду, вызывающую прерывание и возникновение сигнала.
4. Даем команду подчиненному процессу через птреис продолжить работу. Когда он дойдет до адреса A, вызовет прерывание.
5. Отладчик должен проверить причину прерывания (по табличке, смотрим адрес останова и сверяемся по таблице. Если в таблице нет этого адреса => в программе ошибка).
6. Сделали действия, нужно продолжить работу. Восстановили команду, переключили работу подчиненного режим на пошаговый, когда после каждой команды он посылает сигнал.
7. Передаем птреисом управление на предыдущую контрольную точку, прошли контрольную точку в шаговом режиме, снимаем шаговый режим и продолжаем работу.

**FIFO-файл (именованный канал)** представляет собой специальный тип файла в файловой системе UNIX, который обладает всеми атрибутами файла, такими как имя владельца, права доступа и размер. В нем невозможны операции, связанные с перемещением файлового указателя.

- Могут использоваться для организации взаимодействия процессов,
- в отличие от неименованных каналов эти файлы могут существовать независимо от процессов, взаимодействующих через них, хранятся на ВЗУ => возможно открыть этот файл, записать в него информацию, а через любой промежуток времени (в течение которого допустимы перезагрузки системы) прочитать записанную информацию.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (char *pathname, mode_t mode);
```

- Первый аргумент представляет собой имя создаваемого канала,
- во втором аргументе указываются режимы открытия, устанавливаются права доступа к каналу для владельца, группы и прочих пользователей, и кроме того, устанавливается флаг, указывающий на то, что создаваемый объект является именно FIFO-файлом (`S_IFIFO` или `I_FIFO`).
- После создания именованного канала любой процесс может установить с ним связь посредством системного вызова `open()`, правила:

- если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись;
- если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение;
- процесс может избежать такого блокирования, указав в вызове `open()` специальный флаг (`O_NONBLOCK` или `O_NDELAY`). В этом случае в ситуациях, описанных выше, вызов `open()` сразу же вернет управление процессу.

Правила работы с именованными каналами аналогичны правилам работы с неименованным каналами.

```
int pid;
mkfifo("fifo", S_IFIFO | 0666);
/* создали специальный файл FIFO с открытыми для всех
правами доступа на чтение и запись*/
fd = open("fifo", O_RDONLY | O_NONBLOCK);
/* открыли канал на чтение*/
while (read(fd, &pid, sizeof(int)) == -1){
    printf("Server %d got message from %d !\n",
    getpid(), pid);
    .....
}
close(fd);
unlink("fifo"); /*уничтожили именованный канал*/
return 0;
}

/*
 * процесс-клиент*
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char **argv)
{
    int fd;
    int pid = getpid();
    fd = open("fifo", O_RDWR);
    write(fd, &pid, sizeof(int));
    close(fd);
    return 0;
}

/* процесс-сервер*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    int fd;
```

---

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

IPC поддерживает 3 типа ресурсов:

- Разделяемый ресурс очередь сообщений: модель send-receive. **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.
- **Распределенная память** — позволяет создавать некоторую область ОП и предоставляется возможность подключения к этой ОП разных процессов => общее пространство ОП. Все, что 1 процесс в нее запишет, доступно другим процессам, имеющим доступ к этой области.
- **Массив** (последовательность однотипных элементов) **семафоров** => семафорную операцию можно сделать атомарно для всей группы.

Позволяет организовывать взаимодействие произвольных процессов процессов => **проблема именования**: как передавать системе информацию о процессе, с которыми взаимодействуем => модель именования, основанная на использовании ключей. При создании ресурса с ним ассоциируется **ключ** — целочисленное значение.

- Теперь все процессы, желающие работать с той же общей памятью, должны:

1. обладать соответствующими правами,
2. заявить, что они желают работать с ресурсом общая память с ключом.

- возможны коллизии из-за случайного совпадения ключей различных ресурсов => унификация именования IPC-ресурсов => ftok(). Функция обращается к указанному файлу, считывает его атрибуты и, используя значение второго аргумента, генерирует уникальный ключ (уникальное целое число).

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

```
#include <sys/ipc.h>
```

```
key_t ftok(char *filename, char proj);
```

- Первый параметр данной функции — полное имя существующего файла. При генерации ключа используются атрибуты файла => если один процесс для генерации ключа ссылается на некоторый файл, создает ресурс, потом этот файл уничтожается и создается другой файл с тем же именем (но, скорее всего, с другими атрибутами), то другой процесс, желающий получить ключ к созданному ресурсу, не сможет этого сделать, т.к. функция ftok() будет генерировать иное значение

- Второй параметр — это уточняющая информация (значение символьной переменной; в частности, этот параметр может использоваться для поддержания разных версий программы).

- При создании и/или подключение через системы IPC использовать только те ключи, которые мы сгенеририровали при обращении к ftok.

- Каждый ресурс IPC имеет атрибут владельца - тот процесс (его идентификация), который создал данный ресурс => удалить данный ресурс может только владелец.

- существует механизм передачи прав владения от процесса процессу.

- С каждым ресурсом связаны три категории прав (права владельца, группы и остальных пользователей). Но в каждой категории имеются лишь права на чтение и запись: права на выполнение нет.

- Ресурсы IPC могут существовать без процессов, их создавших, пока его явно не удалят либо до перезапуска системы.

- Функциональная структура работы с IPC идентична, среди этих наборов есть функции с суффиксом get:  
<ResourceName>get(key, ..., flags);

- key — ключ, который мы хотим ассоциировать с разделяемым ресурсом.

- флаги (flags) задаёт права доступа к ресурсу и режимы работы с ресурсом. Этот параметр может быть комбинацией различных флагов.

- **IPC\_PRIVATE** — данный флаг определяет создание частного IPC-ресурса, доступного только процессу, который его создал, не доступного остальным процессам. Т.е. функция get при наличии данного флага всегда открывает новый ресурс, к которому никто другой не может подключиться. Используется для переключения

взаимодействия только родственных процессов (ресурс можно только передавать по наследству).

– **IPC\_CREAT** — если данного флага нет среди параметров функции get, то это означает, что процесс хочет подключиться к существующему ресурсу. В этом случае, если такой ресурс существует и права доступа позволяют к нему обратиться, то процесс получит дескриптор ресурса и продолжит работу, иначе ошибка. Если же при вызове функции get данный флаг установлен, то функция работает на создание или подключение к существующему ресурсу => возможно возникновение ошибки из-за нехватки прав доступа в случае существования ресурса. Но при установленном флаге встает вопрос, кто является владельцем ресурса, и кто его должен удалять => для разрешения проблемы используется следующий флаг.

– **IPC\_EXCL** — используя данный флаг в паре с флагом IPC\_CREAT, функция get будет работать только на создание нового ресурса. Если же ресурс будет уже существовать, то ошибка.

Возврат функции get:

- ENOENT — ресурс не существует, и не указан флаг IPC\_CREAT;
- EEXIST — ресурс существует, и установлены флаги IPC\_CREAT | IPC\_EXCL;
- EACCESS — не хватает прав доступа на подключение.

**Очередь сообщений** - некоторого функционально расширенного аналога канала

- модель FIFO последовательность всех сообщений, который в него записывается.

- записываем информацию и тип (неотрицательное целое).

- Рассматриваем очередь как нетипизированную => работа по модели FIFO.

- Рассматриваем очередь как объединение подочередей, содержащие сообщения одного типа.

## 1. Функция создания/доступа к очереди сообщений. msgget

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

В случае успешного выполнения функция возвращает положительный дескриптор очереди (на самом деле файловый дескриптор).

Причины ошибки: ресурса нет, таблица файлов заполнена, недостаточно прав.

## 2. Функции отправки и приема сообщений.

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

Первый аргумент — идентификатор очереди, полученный в результате вызова msgget().

Второй аргумент — указатель на буфер, содержащий реальные данные и тип сообщения, подлежащего посылке в очередь, в третьем аргументе указывается размер буфера – структуры с полями:

- long msqtype — тип сообщения (только положительное длинное целое);
- char msgtext[] — данные (тело сообщения). При попытке отправить сообщение, у которого число элементов в массиве msgtext превышает MSGMAX, системный вызов вернет -1.

Последний аргумент функции — это флаги, можно выделить те, которые определяют режим блокировки при отправке сообщения. IPC\_NOWAIT, который позволяет работать без блокировки:

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

Первые три аргумента аналогичны аргументам предыдущего вызова.

Четвертый аргумент указывает тип сообщения, которое процесс желает получить. При доступе используется стратегия FIFO.

= 0 => будет получено сообщение любого типа (т.е. идет работа со сквозной очередью).

> 0 => будет извлечено сообщение указанного типа.

< 0 => наименьший по модулю тип

Последним аргументом является комбинация (побитовое сложение) флагов.

- Если среди флагов не указан IPC\_NOWAIT, в очереди не найдено ни одного нужного сообщения => процесс будет заблокирован до появления такого сообщения.
- если такое сообщение существует, но его длина превышает указанную в аргументе msgsz, то процесс заблокирован не будет, и вызов сразу вернет -1; сообщение при этом останется в очереди.
- Если же флаг IPC\_NOWAIT указан, и в очереди нет ни одного необходимого сообщения, то вызов сразу

вернет -1.

- флаг MSG\_NOERROR: в этом случае он может прочитать сообщение, даже если его длина превышает указанную емкость буфера. Тогда в буфер будет записано первые msgsz байт тела сообщения, а остальные данные отбрасываются. В случае успешного завершения функция возвращает количество успешно прочитанных байтов в теле сообщения.

**3. Функции управления ресурсом** – очередью сообщений, обеспечивают в общем случае изменение режима функционирования ресурса, в т.ч. и удаление ресурса.

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

Аргументы — идентификатор ресурса, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры очереди: msgid\_ds, в полях которой хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возможные значения аргумента cmd:

- IPC\_STAT — скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре buf;
- IPC\_SET — заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре buf;
- IPC\_RMID — удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

```
strcpy(messageto.mes, "Message for client");
msgsnd (mesid, &messageto, sizeof(messageto) -
        sizeof(long), 0);
}

/*
 * КЛИЕНТ *
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    struct
    {
        long mestype;
        long mes;
    } 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, messagefrom.mes);
        msgsnd(mesid, &messageto, sizeof(messageto) -
                sizeof(long), 0);
    }
}
```

---

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

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти => 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 => он может работать с этой областью памяти аналогично тому, как если бы она была выделена динамически

(например malloc()), но разделяемая область памяти не уничтожается автоматически даже после того, как процесс, создавший или использовавший ее, перестанет с ней работать.

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

Аргументы: key — ключ для доступа к разделяемой памяти;

size задает размер области памяти, к которой процесс желает получить доступ.

В результате вызова shmget() создана новая область разделяемой памяти => ее размер будет соответствовать значению size. В случае успешного завершения вызов возвращает положительное число — дескриптор области памяти, в случае неудачи возвращается -1

Процесс подключается к существующей области разделяемой памяти => значение size должно быть не более ее размера, иначе вызов вернет -1. Size меньшее фактического размера => процесс сможет получить доступ только к первым size байтам этой области.

Третий параметр определяет флаги, управляющие поведением вызова.

Наличие у процесса дескриптора разделяемой памяти не дает ему возможности работать с ресурсом (при работе с памятью процесс работает в терминах адресов) => нужна **функция, которая присоединяет полученную разделяемую память к адресному пространству процесса**, — это функция shmat().

```
char *shmat(int shmid, char *shmaddr, int shmflg);
```

- Процесс подсоединяет область разделяемой памяти, дескриптор которой указан в shmid, к своему виртуальному адресному пространству.
- Возвращает указатель на начало разделяемой памяти.
- После выполнения процесс сможет читать и модифицировать данные, находящиеся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.
- В качестве второго аргумента процесс может указать виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Если 0 => система сама может выбрать адрес начала разделяемой памяти.
- Передача конкретного адреса (положительного целого) в этом параметре имеет смысл лишь в определенных случаях => процесс желает связать начало области разделяемой памяти с конкретным адресом => учитывать возможные коллизии с имеющимся адресным пространством.

Третий аргумент представляет собой комбинацию флагов. SHM\_RDONLY - подсоединяемая область будет использоваться только для чтения.

Для **открепления разделяемой памяти** от адресного пространства процесса используется функция shmdt().

```
int shmdt(char *shmaddr);
```

- Параметр shmaddr — адрес прикрепленной к процессу памяти, который был получен при вызове shmat().
- В случае успешного выполнения функция вернет значение 0, в случае неудачи возвращается -1.

**shmctl() управления ресурсом разделяемая память.**

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения.
- Аргументы вызова — дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти.
- **Тип shmid\_ds** - структура, в полях которой хранятся права доступа к области памяти, ее размер, число процессов, подсоединеных к ней в данный момент, и статистика обращений к области памяти.
- cmd: IPC\_SET – скопировать структуру, описывающую управляющие параметры области памяти по адресу, указанному в параметре buf

IPC\_SET – заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре buf. Выполнить может процесс с соотв/привилегиями доступа.

IPC\_RMID – удалить очередь, может только процесс, у которого эффективный идентификатор

пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

SHM\_LOCK, SHM\_UNLOCK – блокировать или разблокировать область памяти, привилегия. Единственные средства синхронизации в данном ресурсе, их реализация должна поддерживаться аппаратурой.

```
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);

    /*
```

176

```
/* работаем с разделяемой памятью, как с обычной */
putm(shmaddr);
waitprocess();
shmctl(shmid, IPC_RMID, NULL);
return 0;
}
```

---

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

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.

Каждый из семафоров массива может иметь текущее значение, с каждым семафором можем выполнять семафорные операции – аналог операций Дейкстры . Они **псевдоатомарные** (атомарность обеспечивается не аппаратными средствами, как инструкции процессора, они реализованы программно). Семафорные операции над семафорами, составляющие массив, выполняются **единовременно**.

1. Для получения доступа к массиву семафоров (или создания) int semget (key\_t key, int nsems, int semflag);  
key – ключ для доступа к разделяемому ресурсу,  
nsems - количество семафоров в создаваемом наборе (длина массива семафоров)  
semflags – флаги, управляющие поведением вызова.  
- процесс, имеющий право доступа к массиву семафоров по чтению, может проверять значение семафоров;  
- процесс, имеющий право доступа по записи, может как проверять, так и изменять значения семафоров  
- есть флаг IPC\_CREAT => nsems должен > 0, иначе значение nsems игнорируется.  
- SEMMSL задает макс возможное число семафоров в наборе.  
- В случае успеха вызов semget() возвращает положительный дескриптор созданного разделяемого ресурса, в случае неудачи -1.

2. Используя полученный дескриптор, можно **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)

semid – дескриптор массива семафоров;

semop – указатель на массив из объектов типа struct sembuf, каждый из которых задает одну операцию над семафором;

nops – длина массива semop.

- Количество семафоров, над которыми процесс может одновременно производить операцию в одном вызове ограничено SEMOPM.

struct sembuf {

```
    short sem_num; /* номер семафора в векторе */
    short sem_op; /* производимая операция */
    short sem_flg; /* флаги операции */
```

}

**Поле операции** в структуре интерпретируется следующим образом. Пусть значение семафора с номером `sem_num` равно `sem_val`.

1. если значение операции не равно нулю:

♣ оценивается значение суммы `sem_val + sem_op`.

♣ если эта сумма больше либо равна нулю, то значение данного семафора устанавливается равным этой сумме:  $sem\_val = sem\_val + sem\_op$

♣ если же эта сумма меньше нуля, то действие процесса будет приостановлено до тех пор, пока значение суммы `sem_val + sem_op` не станет больше либо равно нулю, после чего значение семафора устанавливается равным этой сумме:  $sem\_val = sem\_val + sem\_op$

2. Если код операции `sem_op` равен нулю:

♣ Если при этом значение семафора (`sem_val`) = 0, происходит немедленный возврат из вызова

♣ Иначе происходит блокирование процесса до тех пор, пока значение семафора не обнулится, после чего происходит возврат из вызова

- флаг `IPC_NOWAIT` предписывает соответствующей операции над семафором не блокировать процесс, а сразу возвращать управление из вызова `semop()`.

- флаг `SEM_UNDO`, в этом случае система запомнит изменение значения семафора, произведенные данным вызовом, и по завершении процесса автоматически ликвидирует это изменение => предохраняет от ситуации, когда процесс уменьшил значение семафора, начав работать с ресурсом, а потом, не увеличив значение семафора обратно, по какой-либо причине завершился => остальные процессы оказались бы заблокированы навечно.

### 3. Запрашивать и изменять управляющие параметры разделяемого ресурса, а также удалять его

`int semctl (int semid, int num, int cmd, union semun arg)`

- Параметр `num` представляет собой индекс семафора в массиве,

- параметр `cmd` задает операцию, которая должна быть выполнена над данным семафором.

- `union semun` и используется для считывания или задания управляющих параметров одного семафора или всего массива, в зависимости от значения аргумента `cmd`. Тип данных `union semun` определен в файле и выглядит следующим образом:

```
union semun {  
    int val; /* значение одного семафора */  
    struct semid_ds *buf; /* параметры массива семафоров в целом */  
    ushort *array; /* массив значений семафоров */  
}
```

`cmd`: `IPC_STAT` – скопировать управляющие параметры набора семафоров по адресу `arg.buf`

`IPC_SET` – заменить управляющие параметры набора семафоров на те, которые указаны в `arg.buf`. Процесс должен быть владельцем или создателем массива семафоров, либо привил., при этом процесс может изменить только владельца массива семафоров и права доступа к нему.

`IPC_RMID` – удалить массив семафоров, процесс должен быть владельцем или создателем массива семафоров, либо привилег.

`GETALL`, `SETALL` – считать / установить значения всех семафоров в массив, на который указывает `arg.array`

```
1й процесс:  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/ipc.h>  
#include <sys/sem.h>  
#include <string.h>  
#define NMAX 256  
int main(int argc, char **argv)  
{  
    key_t key;  
    int semid, shmid;  
    struct sembuf sops;  
    char *shmaddr;  
    char str[NMAX];  
  
    key = ftok("/usr/ter/exmpl", 'S');  
    /* создаем уникальный ключ */  
    semid = semget(key, 1, 0666 | IPC_CREAT);  
    /* создаем один семафор с определенными правами доступа */  
    shmid = shmget(key, NMAX, 0666 | IPC_CREAT);  
    /* создаем разделяемую память на 256 элементов */
```

`GETVAL` – возвратить значение семафора с номером `num`.

Последний аргумент вызова игнорируется.

`SETVAL` – установить значение семафора с номером `num` равным `arg.val`

**Пример:** 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

```

shmaddr = shmat(shmid, NULL, 0);
/* подключаемся к разделу памяти, в shaddr - 
указатель на буфер с разделяемой памятью */
semctl(semid, 0, SETVAL, (int) 0);
/* инициализируем семафор значением 0 */
sops.sem_num = 0;
sops.sem_flg = 0;
do { /* запуск цикла */
    printf("Введите строку:");
    if (fgets(str, NMAX, stdin) == NULL)
    {
        /* окончание ввода */
        /* пишем признак завершения - строку "Q" */
        strcpy(str, "Q");
    }
    /* в текущий момент семафор открыт для этого
процесса */
    strcpy(shmaddr, str); /* копируем строку в разд.
память */
    /* предоставляем второму процессу возможность
войти */
    sops.sem_op = 3; /* увеличение семафора на 3 */
    semop(semid, &sops, 1);
    /* ждем, пока семафор будет открыт для 1го
процесса - для следующей итерации цикла */
    sops.sem_op = 0; /* ожидание обнуления семафора */
    semop(semid, &sops, 1);
} while (str[0] != 'Q');
/* в данный момент второй процесс уже дочитал из
разделяемой памяти и отключился от нее - можно ее
удалять*/
shmdt(shmaddr); /* отключаемся от разделяемой памяти */
shmctl(shmid, IPC_RMID, NULL);
/* уничтожаем разделяемую память */
semctl(semid, 0, IPC_RMID, (int) 0);
/* уничтожаем семафор */
return 0;
}

```

требует синхронизации, которая будет осуществляться на основе механизма семафоров. Стоит обратить внимание на то, что с одним и тем же ключом одновременно создаются ресурсы двух разных типов (в случае использования ресурсов одного типа подобные действия некорректны).

#### 2й процесс:

```

/* необходимо корректно определить существование ресурса,
если он есть - подключиться */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX 256

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

```

181

```

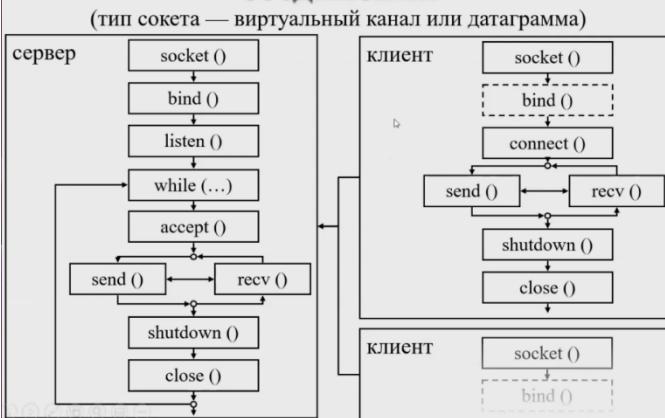
    int semid, shmid;
    struct sembuf sops;
    char *shmaddr;
    char str[NMAX];

    key = ftok("/usr/ter/exmpl",'S');
    /* создаем тот же самый ключ */
    semid = semget(key, 1, 0666 | IPC_CREAT);
    shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
    /* аналогично предыдущему процессу - инициализации
ресурсов */
    shmaddr = shmat(shmid, NULL, 0);
    sops.sem_num = 0;
    sops.sem_flg = 0;
    /* запускаем цикл */
    do
    {
        printf("Waiting... \n"); /* ожидание на семафоре */
        sops.sem_op = -2;
        /* будем ожидать, пока "значение семафора" +
"значение sem_op" не станет положительным*/
        semop(semid, &sops, 1);
        /* теперь значение семафора равно 1 */
        strcpy(str, shmaddr); /* копируем строку из
разд.памяти */
        /*критическая секция - работа с разделяемой
памятью - в этот момент первый процесс к
разделяемой памяти доступа не имеет*/
        if (str[0] == 'Q')
        {
            /* завершение работы - освобождаем
разделяемую память */
            shmdt(shmaddr);
        }
        /*после работы - обнулим семафор*/
        sops.sem_op=-1;
        semop(semid, &sops, 1);
        printf("Read from shared memory: %s\n", str);
    } while (str[0] != 'Q');
    return 0;
}

```

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

## Предварительное установление соединения



Аппарат сокетов позволяет организовывать взаимодействие процессов в рамках сети.

**Сокеты** представляют собой в определенном смысле обобщение механизма каналов, но с учетом возможных особенностей, возникающих при работе в сети, 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.

- Стоит для каждого нового сокета создавать сыновий процесс, чтобы не заполнить таблицу открытых файлов для 1 процесса.

### «клиент-сервер».

1. Процесс-сервер запрашивает у ОС сокет.
2. Получает, присваивает ему некоторое имя (адрес), которое предполагается заранее известным всем клиентам, которые захотят общаться с данным сервером.
3. сервер переходит в режим ожидания и обработки запросов от клиентов.
4. Клиент, со своей стороны, тоже создает сокет и запрашивает соединение своего сокета с сокетом сервера, имеющим известное ему имя (адрес).
5. Установление соединения => клиент и сервер могут обмениваться данными через соединенную пару сокетов.

В сервере создается сокет, ассоциируем с ним некоторое имя. В клиенте создаем свой сокет, через который хотим осуществлять взаимодействие с сервером (необязательно присваивать имя). Клиент обращается к connect и запрашивает возможность установки соединения Сервер сделал именование, если уже вызвал listen после connect в клиенте (до => connect вернет ошибку). Сервер обратился к Listen и установил очередь сообщений. Очередь занята => отказ connects, иначе в сервере подтверждаем соединение accept (создается новый сокет, который связывается с клиентом) происходит соединение.

**Какое минимальное количество открытых сокетов имеет сервер при реализации модели клиент – сервер средствами аппарата сокетов?** Количество активных присоединённых клиентов + 1

**1. Создание сокета int socket (int domain, int type, int protocol).** Возвращает в случае успеха дескриптор сокета, которое может быть использовано в дальнейших вызовах при работе с данным сокетом. Дескриптор сокета - файловый дескриптор, является индексом в таблице файловых дескрипторов процесса, и может использоваться в дальнейшем для операций чтения и записи в сокет, которые осуществляются подобно операциям чтения и записи в файл.

**Domain.** Локализация процессов определяется коммуникационным доменом – то, какие процессы могут взаимодействовать:

- 1.в рамках одной сети, локально – AF\_UNIX,
2. в рамках сети – AF\_INET.

Тип сокета **type**:

1. SOCK\_STREAM Виртуальный канал – надежное взаимодействие, контроль и гарантированная доставка (аналог TCP).
2. SOCK\_DGRAM Дейтаграммное соединение для передачи отдельных пакетов, содержащих порции данных – датаграмм – без гарантии доставки пакетов (UDP). Более быстрые.

Тип протокола **protocol**, связан с типом сокета.

1. IPPROTO\_TCP.
2. IPPROTO\_UDP, не может устанавливаться для виртуального канала.

Для того, чтобы организовывать взаимодействие между процессами => сокет можно именовать =>  
int bind (int sockfd, struct sockaddr \*myaddr, int addrlen).

1. Дескриптор сокета
  2. Указатель на структуру, где размещается адрес, который ставим в соответствие сокету
  3. Размер адреса.
- Имя/адрес зависит от коммуникационного домена.

Для домена AF\_UNIX

```
struct sockaddr_un {  
    short sun_family; /* == AF_UNIX */  
    char sun_path[108];  
};
```

Для домена AF\_INET формат структуры описан в и выглядит следующим образом:

```
struct sockaddr_in {  
    short sin_family; /* == AF_INET */  
    u_short sin_port; /* port number */  
    struct in_addr sin_addr; /*  
        host IP address */  
    char sin_zero[8]; /* not used */  
};
```

**Сокеты с предварительным установлением соединения**, когда до начала передачи данных устанавливаются адреса сокетов отправителя и получателя данных – такие сокеты соединяются друг с другом и остаются соединенными до окончания обмена данными. Обязательно для виртуальных каналов.

**Сокеты без установления соединения**, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением.

int connect (int sockfd, struct sockaddr \*serv\_addr, int addrlen) - Для установления соединения.

первый аргумент – дескриптор сокета,

второй аргумент – указатель на структуру, содержащую адрес сокета, с которым производится соединение, третий аргумент содержит реальную длину этой структуры

Если используются сокеты с предварительным установлением соединения: listen(),

int listen (int sockfd, int backlog);

- используется процессом-сервером для того, чтобы сообщить системе о том, что он готов к обработке запросов на соединение, поступающих на данный сокет. До тех пор, пока процесс – владелец сокета не вызовет listen(), все запросы на соединение с данным сокетом будут возвращать ошибку.

- backlog, содержит максимальный размер очереди запросов на соединение.

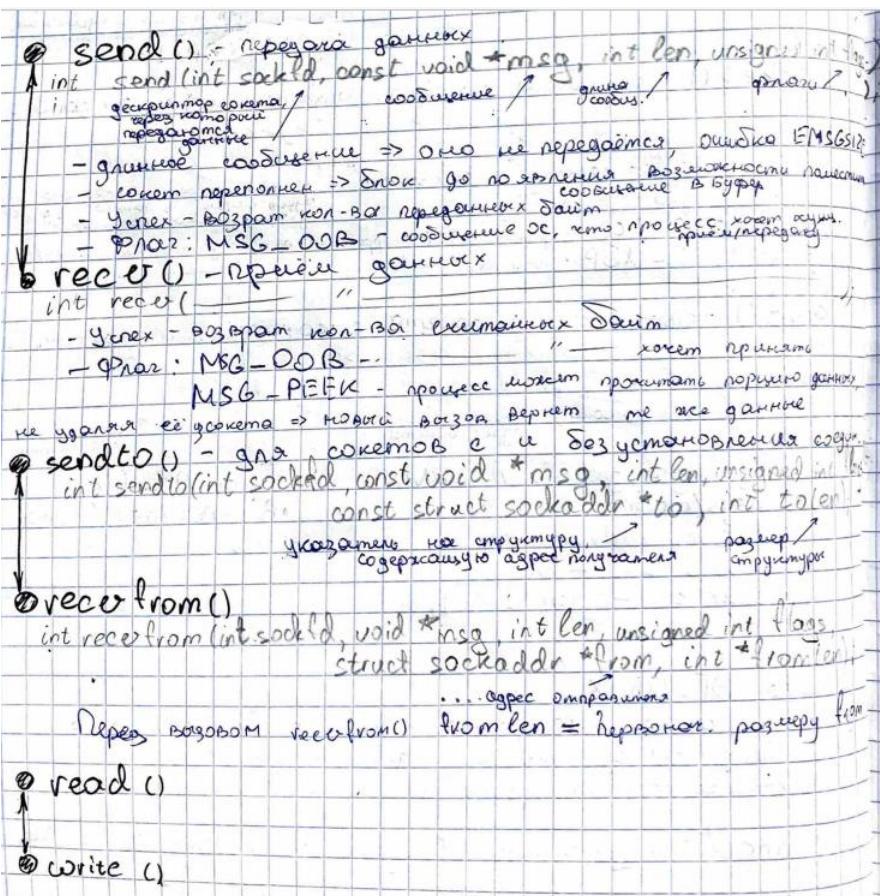
int accept (int sockfd, struct sockaddr \*addr, int \*addrlen); применяется сервером для удовлетворения поступившего клиентского запроса на соединение с сокетом, который сервер к тому моменту уже прослушивает (т.е. предварительно была вызвана функция listen()).

- Когда запрос поступает и соединение устанавливается, accept() создает новый сокет, который будет использоваться для работы с данным соединением, и возвращает дескриптор этого нового сокета.

- первоначальный сокет продолжает оставаться в состоянии прослушивания.

- Через новый сокет осуществляется обмен данными, в то время как старый сокет продолжает обрабатывать другие поступающие запросы на соединение.

=> Это позволяет процессу-серверу поддерживать несколько соединений одновременно, обычно реализуется путем порождения для каждого установленного соединения отдельного процесса-потомка, который занимается собственно обменом данными только с этим конкретным клиентом, в то время как процесс-родитель продолжает прослушивать первоначальный сокет и порождать новые соединения.



Для приема и передачи данных через сокет используются

### 1. Send() – receive() лишь

int send(int sockfd, const void \*msg, int len, unsigned int flags);

int recv(int sockfd, void \*buf, int len, unsigned int flags); Эти функции используются для обмена только через сокет с предварительно установленным соединением

### 2. обычные read() и write(), в качестве дескриптора которым передается дескриптор сокета.

### 3. sendto, receivefrom может быть использована как с сокетами с установлением соединения и без:

int sendto(int sockfd, const void \*msg, int len, unsigned int flags, const struct sockaddr \*to, int tolen);

int recvfrom(int sockfd, void \*buf, int len, unsigned int flags, struct sockaddr \*from, int \*fromlen);

- При использовании сокетов с установлением виртуального соединения границы сообщений не сохраняются, поэтому приложение, принимающее сообщения, может принимать данные совсем не теми же порциями, какими они были посланы.

- Вся работа по интерпретации сообщений возлагается на приложение.

int shutdown (int sockfd, int mode); закрывает соединение, важен в 1ю очередь для закрытия соединения сокета с использованием виртуального канала.

mode=0, то сокет закрывается для чтения, дальнейшие попытки чтения будут возвращать EOF.

mode=1, то сокет закрывается для записи, и при осуществлении в дальнейшем попытки передать данные будет выдан кода неудачного завершения (-1).

mode=2, то сокет закрывается и для чтения, и для записи.

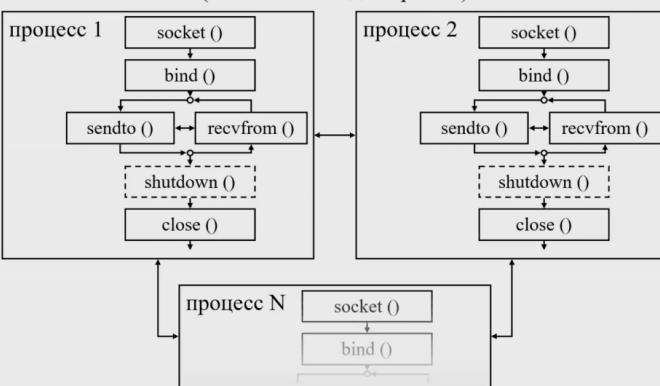
- Аналогично файловому дескриптору, дескриптор сокета освобождается системным вызовом close(). Даже если до этого не был вызван shutdown(), соединение будет закрыто.

- Если используемый для соединения протокол гарантирует доставку данных (т.е. тип сокета – виртуальный канал), то вызов close() будет блокирован до тех пор, пока система будет пытаться доставить все данные, находящиеся «в пути» (если таковые имеются), в то время как вызов shutdown() извещает систему о том, что эти данные уже не нужны и можно не предпринимать попыток их доставить, и соединение закрывается немедленно.

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

#### Сокеты без предварительного соединения

(тип сокета — датаграмма)



3. `sendto`, `receivefrom` может быть использована как с

сокетами с установлением соединения и без:

`int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);`

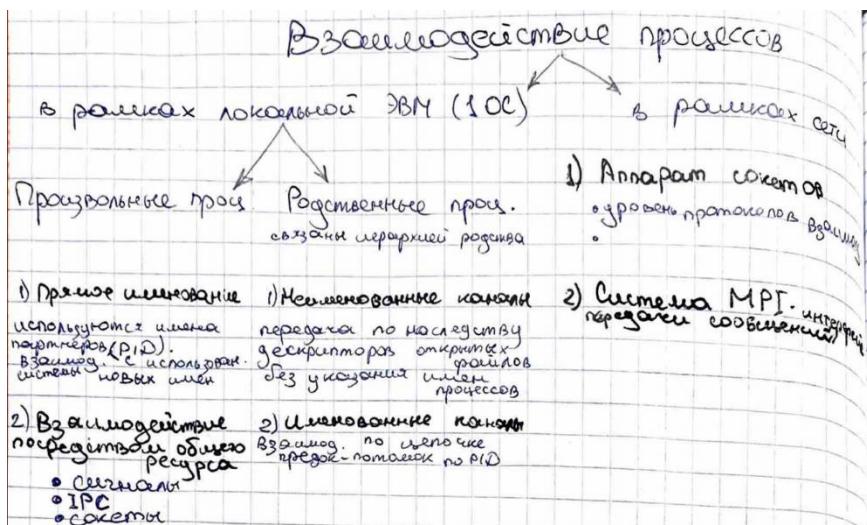
`int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);`

- При использовании сокетов с установлением виртуального соединения границы сообщений не сохраняются, поэтому приложение, принимающее сообщения, может принимать данные совсем не теми же порциями, какими они были посланы.

- Вся работа по интерпретации сообщений возлагается на приложение.

**Сокеты без установления соединения**, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением.

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



### 39. 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 ОС, представляющую собой совокупность организованных наборов данных, хранящихся на внешних запоминающих устройствах, и программных средств, гарантирующих именованный доступ к этим данным и их защите. Первый виртуальный ресурс.

#### Модели структуры организации файлов:

1. **последовательности байтов.** В этом случае содержимое файла представляется как неинтерпретируемая информация (или интерпретируемая примитивным образом). Задача интерпретации данных ложится на пользователя.

2. **последовательность записей переменной и постоянной** (не требуют разделителей) длины, аппаратно-ориентированна, аналог перфокарты, организация файла и рассчитана на работу с магнитными лентами. Проблемы, такие как коррекция данных в середине файла, вследствие чего меняется размер файла, и появляется необходимость сдвигать «хвост» файла на ленте.

3. **иерархической организации файла.** Сложная структура, позволяет организовывать динамическую работу с данными. Популярно дерево, в узлах которого расположены записи. Каждая запись состоит из двух полей:

поле ключа (мб номер записи) и поле данных. Удобная для редактирования файла, требует сложной реализации.

С каждым файлом как объектом ФС связывается ряд характеристик – **атрибуты файлов**:

1. Под именем файла понимается последовательность символов, используя которую организуется именованный доступ к данным файла.
2. Права доступа - возможность доступа к содержимому файла различным категориям пользователей.
3. персонификация — связан с предыдущим, инфа о принадлежности файла: о создателе файла, владельце файла
4. Тип файла — инфа о способе организации файла и интерпретации его содержимого. **Файлы устройств**, соответствующие тем устройствам, которые обслуживает данная ОС, через них происходит обращение к драйверам устройств. **Регулярные файлы** могут хранить различную информацию (текстовую, графическую и пр). **Исполняемый файл** можно запустить как процесс, в отличие от неисполняемого. Интерпретация: явная/неявная.
5. Размер записи (или размер блока) - данный файл организован в виде последовательности блоков данного размера, размер определяется пользователем, может быть **стационарным**, когда при создании файла указывается фиксированный размер блоков, и **нестационарным**, когда размер блока задается каждый раз при открытии файла.
6. Размер файла в байтах.
7. Указатель чтения/записи — это указатель, относительно которого происходит чтение или запись информации.
8. Возможны атрибуты, отражающие системную и статистическую информацию о файле: например, время последней модификации, время последнего обращения, предельный размер файла и т.д.
9. Инфа о размещении содержимого файла, т.е. где в ФС организовано хранение данных файла, и как оно организовано.

**1. Начало работы с файлом** (или открытие файла), операции открытия файла, процесс передает файловой системе запрос на работу с конкретным файлом. Получив запрос, файловая система проверяет возможности (в т.ч. на наличие полномочий) работы с файлом; в случае успеха выделяет внутри себя необходимые ресурсы для работы процесса с указанным файлом. 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.), он может размещаться как в адресном пространстве процесса, так и в пространстве памяти ОС => при открытии файла процесс получает либо номер файлового дескриптора, либо указатель на начало данной структуры.

2. Операции по работе с содержимым файла (чтение и запись), также операции, изменяющие атрибуты файла (режимы доступа, изменение указателей чтения/записи и т.п.).

3. Закрытие файла: уведомление системе о закрытии процессом файлового дескриптора (а не файла). Даже в рамках одного процесса можно открыть один и тот же файл два и более раза, и на каждое открытие будет предоставлен новый файловый дескриптор. ОС выполняет действия по корректному завершению работы с файловым дескриптором, а если закрывается последний открытый дескриптор — то осуществляет корректное завершение работы с файлом (освобождаются системные ресурсы, разгрузка кэш-буферов файловых обменов и т.д.).

ОС предлагает **унифицированный набор интерфейсов**: – open, close, read/write, delete — удаление файла из файловой системы; – seek — позиционирование указателя чтения/записи; – read\_attributes/write\_attributes — чтение/модификация некоторых атрибутов файла (в файловых системах, рассматривающих имя файла не как атрибут, возможна дополнительная функция переименования файла — rename).

В ФС есть компонент, посредством которого можно установить соответствие между именем файла и его атрибутами, получить информацию о размещении данных в файловой системе и организовать доступ к данным - **каталог** — это системная структура данных файловой системы (специальный вид файлов), в которой находится информация об именах файлов, а также инфа, обеспечивающая доступ к атрибутам и содержимому файлов. Модели организации каталогов

1. **одноуровневая файловая система** (система с одноуровневым каталогом), единственный каталог, в

котором перечислены всевозможные имена файлов, находящихся в данной системе: для каждого файла хранится информация об его имени, расположении первого блока и размере файла. Используется в бытовой технике.

- + простота доступа к информации файлов,
- не предполагает многопользовательской работы, коллизии имен

**2. двухуровневая файловая система**, предполагает работу нескольких пользователей, позволяет группировать файлы по принадлежности тому или иному пользователю.

- + не возникают коллизии имен файлов разных пользователей.
- неудобно и нежелательно расположение всех файлов одного пользователя в одном месте
- остается проблема коллизии имен для файлов одного пользователя.

**3. двух-, трех- и вообще N-уровневые** (N – фиксированное) актуальны и по сей день. Объясняется простотой по своей структуре и организации работы с ними (простейший мобильный телефон).

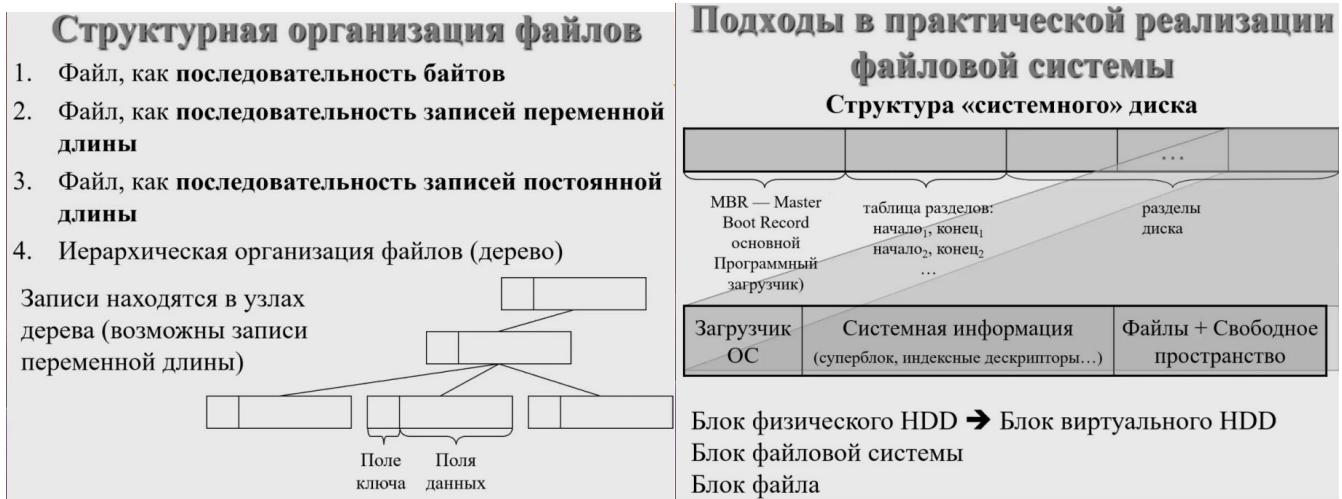
**4. иерархическая файловая система**. Вся информация в иерархической файловой системе представляется в виде дерева, имеющего корень - **корневая файловая система**.

- В узлах дерева, отличных от листьев, находятся каталоги, содержат информацию о размещенных в них файлах. Иерархические файловые системы обычно имеют специальный тип файлов – **каталогов**, не как отдельная выделенная структура данных, а как файл особого типа.
- Листом дерева может быть либо файл-каталог, либо любой файл файловой системы.
- Возможность использования 的独特的命名文件, основывается на том, что в дереве существует единственный путь от корня до любого узла.
- Характеристики:

**Текущий каталог** — это каталог, на работу с которым в данный момент настроена файловая система, Файлы в текущем каталоге доступны «просто» по имени => **имя файла** — это имя файла относительно текущего каталога, а **полное имя файла** — это перечень всех имен файлов от корня до узла с данным файлом, содержит префиксный символ, обозначающего корневой каталог (в ОС Unix "/"). Можно использовать **относительные имена файлов** — это путь от некоторого каталога до данного файла, необходимо указать явно или неявно каталог, относительно которого строится это именование.

**Домашний каталог** - для каждого зарегистрированного в системе пользователя (или для всех пользователей) задается полное имя каталога, который должен стать текущим каталогом при входе пользователя в систему.

- + Уникальное именование имен



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

**Системные устройства** — устройства, на котором, как считается аппаратурой компьютера, должна присутствовать ОС.

- можно определить цепочку внешних устройств, которые при загрузке компьютера могут рассматриваться

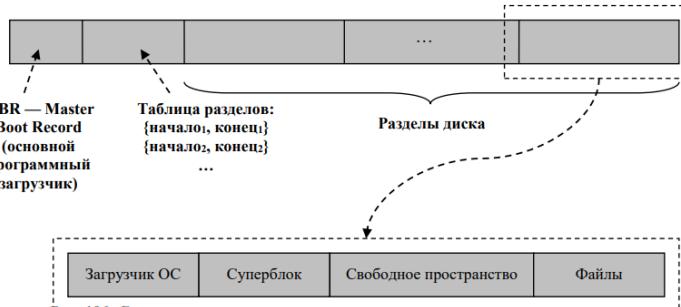


Рис. 106. Структура «системного» диска.

как системные устройства.

- В цепочке приоритет, во время старта ВС перебирает данную цепочку в порядке убывания приоритета до тех пор, пока не обнаружит готовое к работе устройство, предполагает, что на этом устройстве имеется необходимая системная информация и пытается загрузить с него ОС.

- Пусть первым системным устройством является флоппи-дисковод (в нем - дискета), вторым — дисковод оптических дисков (CDROM), а третьим — жесткий диск. Аппаратный загрузчик обращается к первому системному устройству => в нем дискета => дисковод готов к работе => попытка загрузки ОС с флоппи-диска. Неуспешно => сообщение об ошибке загрузки системы. Дискеты во флооппи-дисководе не будет, но будет находиться диск в CDROM => попытка загрузить ОС с оптического диска.

- Работа аппаратного загрузчика основана на предположении о том, что любое системное устройство имеет предопределенную структуру. В начальном блоке системного устройства **располагается основной программный загрузчик (MBR — Master Boot Record)**. Основной программный загрузчик - загрузчик конкретной ОС или унифицированный загрузчик, имеющий информацию о структуре системного диска и способный загружать по выбору пользователя одну из альтернативных ОС.

- После блока основного программного загрузчика на диске следует последовательность блоков, в которых находится т.н. **таблица разделов (partition)** (аналогично сегментной памяти). Каждый раздел рассматривается как виртуальный диск. Некоторые разделы могут быть выбраны как системные, внутри каждого раздела может быть помещена в свою ОС => границы каждого раздела регистрируются в указанной таблице разделов. Диски имеют большую емкость => для адресации произвольной точки диска не хватает разрядной сетки ЦП => за счет косвенной адресации (адресации относительно начала раздела) использование подобной таблицы позволяет решить данную проблему.

- **Логическая структура раздела.** В начальном блоке раздела находится загрузчик конкретной ОС. Остальное пространство занимает ФС, иногда в ней часть пространства выделяется т.н. **суперблоку**, в котором хранятся настройки (размеры блоков, режимы работы и т.п.) и информация об актуальном состоянии (информация о свободных и занятых блоках и т.п.) ФС.

При включении компьютера управление передается аппаратному загрузчику, который просматривает (согласно приоритетам) список системных устройств, определяет готовое к работе устройство и передает управление основномуциальному загрузчику этого устройства - программному компоненту и может загрузить конкретную ОС, а может являться **мультисистемным загрузчиком**, способным предложить пользователю выбрать, какую из ОС, расположенных в различных разделах диска. Мультисистемный загрузчик владеет информацией, какая ОС в каком разделе диска находится.

- После выбора загрузчик по таблице разделов определяет координаты соответствующего раздела и передает управление загрузчику ОС указанного раздела. Загрузчик ОС производит непосредственную загрузку этой ОС.

**Иерархия блоков.** Большинство современных ОС поддерживают целую иерархию блоков, используемую при организации работы с блок-ориентированными устройствами. Размер блока влияет на эффективность работы, если размеры всех блоков будут хотя бы кратны друг другу.

1. В основе лежат **блоки физического устройства**, т.е. это те порции данных, которыми можно совершать обмен с данным физическим устройством. Размер блока физического устройства зависит от конкретного устройства

2. следующий уровнем абстракции — **блоки виртуального диска**. Размер блока - стационарная характеристика, определяемая при настройке системы, и в динамике эта характеристика не меняется.

3. **уровень блоков файловой системы**, используются при организации структуры ФС. Размер блока - стационарная характеристика,

4. **блоки файла**. Размер данных блоков может определить пользователь при открытии или создании файла.

## Модели реализации файлов

### Непрерывные файлы



- Name<sub>1</sub> Name<sub>2</sub> Name<sub>3</sub> Name<sub>4</sub> Name<sub>5</sub> Name<sub>6</sub>
- Достоинства**
- Простота реализации
  - Высокая производительность
- Недостатки**
- Фрагментация свободного пространства
  - Возможность увеличения размера существующего файла

**1. модель непрерывных файлов** - размещение каждого файла в непрерывной области внешнего устройства.

+ простая и эффективная: для обеспечения доступа к файлу среди атрибутов должны присутствовать имя, блок начала и длина файла.

+ нет накладных расходов, самая большая производительность.

+ отсутствие фрагментации файла по диску: поскольку файл хранится в единой непрерывной области диска, то при считывании файла головка жесткого диска совершает минимальное количество механических движений, что означает более высокую производительность системы

- внутренняя фрагментация (хотя это проблема почти всех блокориентированных устройств).

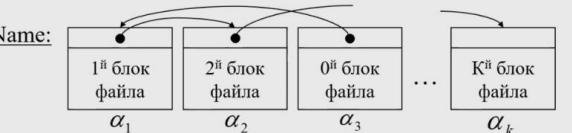
- фрагментация между файлами.

- Должна решаться проблема, возникающая при модификации файла, в частности, при увеличении его содержательной части => при создании файла к запрошенному объему добавляют некоторое количество свободного пространства (например, 10% от запрошенного объема) => ↗ внутренней фрагментации.

- фрагментация свободного пространства: в ходе эксплуатации фрагментация свободного пространства ↗ => на диске множество мелких свободных участков, имеющих очень большой суммарный объем, но из-за своего небольшого размера эти участки использовать не представляется возможным => необходимо запускать процесс компрессии (дефрагментации), который сдвигает файлы с учетом зарезервированного для каждого файла «запаса», «прижимая» их друг к другу => трудоемкая, продолжительная и опасная, поскольку при перемещении файла возможен сбой.

## Модели реализации файлов

### Файлы, имеющие организацию связанного списка



{ $\alpha_i$ } — множество блоков файловой системы, в которых размещены блоки файла Name.

#### Достоинства

- Отсутствие фрагментации свободного пространства (за исключением блочной фрагментации)
- Простота реализации
- Эффективный последовательный доступ

#### Недостатки

- Сложность (неэффективность) организации прямого доступа
- Фрагментация файла по диску
- Наличие ссылки в блоке файла (ситуации чтения 2-х блоков при необходимости чтения данных объемом один блок)

### 2. модель файлов, имеющих организацию связанного списка,

файл состоит из блоков, каждый из которых включает в себя: данные, хранимые в файле, и ссылка на следующий блок файла.

В каждом блоке имеется фиксированное пространство, используется для указания на следующий блок.

+ простая, достаточно эффективная при организации последовательного доступа,

+ решает проблему фрагментации свободного пространства (за исключением блочной фрагментации).

- не предполагает прямого доступа: чтобы обратиться к i-ому блоку, необходимо последовательно просмотреть все предыдущие.

- фрагментация файла по диску, т.е. содержимое файла может быть рассредоточено по всему дисковому пространству => при последовательном считывании содержимого файла добавляется механическая составляющая, ↘ эффективность доступа, диск быстро выходит из строя.

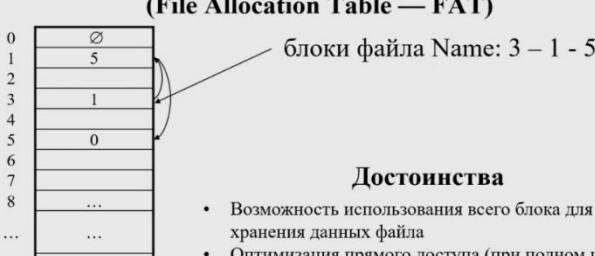
- в каждом блоке присутствует и системная, и пользовательская информация => при необходимости считывать данные, объемом в один логический блок, реально происходит чтение двух блоков.

последовательном считывании содержимого файла добавляется механическая составляющая, ↘

эффективность доступа, диск быстро выходит из строя.

## Таблица размещения файловой системы

### (File Allocation Table — FAT)



#### Достоинства

- Возможность использования всего блока для хранения данных файла
- Оптимизация прямого доступа (при полном или частичном размещении таблицы в ОЗУ)

#### Недостатки

- Желательно размещение всей таблицы в ОЗУ

### 3. таблицы размещения файлов (File Allocation Table — FAT),

фактически модификация предыдущего метода. ОС создает программную таблицу, количество строк в которой совпадает с количеством блоков ФС, предназначенных для хранения пользовательских данных. Есть отдельный каталог (или система каталогов), в котором для каждого имени файла имеется запись, содержащая номер начального блока => таблица размещения имеет позиционную организацию: i-ая строка таблицы хранит информацию о состоянии i-ого блока файловой системы, в ней указывается номер следующего блока файла. По номеру начального блока можем обратиться к FAT таблице и прочесть содержимое записи, которая соответствует начальному блоку. Если у

файла есть еще блоки, то следующие с

- + весь блок используется полностью для хранения содержимого файла.
- + нет накладных расходов, связанных с непроизводительными обменами.
- + при открытии файла можно составить список блоков данного файла => осуществлять прямой доступ.
- для максимальной эффективности необходимо, чтобы эта таблица целиком размещалась в ОП, но для современных огромных дисков, таблица будет иметь большой размер.
- ограничение на размер файла в силу ограниченности длины списка блоков, принадлежащих данному файлу.

**4. модель организации файловой системы с использованием индексных узлов (дескрипторов).** Для каждого файла система формирует специальную структуру данных, содержащую инфу размещении блоков файла. Это системная инфа размещается в пространстве, выделенном в разделе, предназначенная для хранения системной информации в блоке. Совокупность характеристик и перечень блоков файла.

При открытии файла ФС находит его индексный дескриптор. Открывается сеанс работы с файлов, индексный дескриптор помещается в ОП. Каждый обмен проходит через инфу индексного дескриптора. В ОП, принадлежащей ОС, храним все индексные дескрипторы. + Нет необходимости в размещении в ОЗУ информации всей FAT-таблицы обо всех файлах системы. В памяти размещаются атрибуты, связанные только с открытыми файлами.

- + При этом индексный дескриптор имеет фиксированный размер, а файл может иметь практически «неограниченную» длину.
- При предельных размерах файла размер индексного дескриптора становится соизмеримым с размером FAT-таблицы => тривиальное ограничение на максимальный объем файла, либо построение иерархической организации данных о блоках файла в индексном дескрипторе: часть (первые N) блоков перечисляются непосредственно в индексном узле, а оставшиеся представляются в виде косвенной ссылки.

Аналогично для каталогов. Системе всегда удобнее работать с таблицами с записями фиксированного размера => детерминированное условие, можем оценивать характеристики программ.

1. Каталог может организовываться **как записи фикс размера**, содержат поле имени файла и поле с его атрибутами. При длинных именах в каталоге может быть префиксная часть имени, а суффиксная – в атрибутах  
Объем атрибутов может быть достаточно большой =>
2. Второе поле – ссылка на атрибуты файла, где то в данных на ВЗУ.

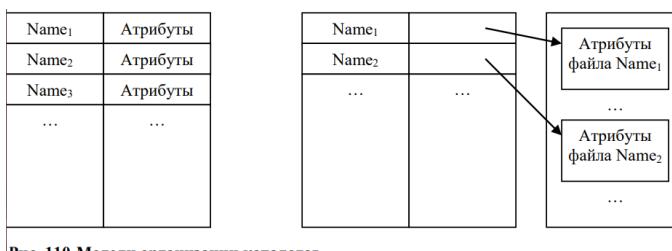


Рис. 110. Модели организации каталогов.

Соотношение имени и атрибутов файла.

1. Взаимно-однозначное соответствие между именем и содержанием файла.

2. У содержимого может быть >2 имени, эти имена могут использоваться в различных стратегиях:

- жесткая связь между именем и содержимым => имя – один из атрибутов файла. Атрибутов может быть произвольное количество. Каждое из имен равноценно: можно работать с содержимым файла равноценно через любой из этих имен. В древообразной иерархии в вершинах хранятся имена. В иерархических каталогах системы можем ссылаться на одно и то же содержимое. Системный вызов «удалить файл» = удалить имя файла, если у файла больше имен нет => удаляется содержимое => кроме имен в атрибутах файла должен быть счетчик имен.

- мягкая символическая связь. Есть файл специального типа, содержимое которого есть полный путь на некоторой существующей файловой системе. Мы можем работать с содержимого оригинального файла через файл – ссылку при использовании системных вызовов обмена. Когда мы удаляем этот файл, удаляется ссылка, содержимое не меняется. При удалении файла, на который ссылается ссылка, то она становится **подвешенным** => несимметричный набор операций, они работают по разному.

Как организовывать инфу об имеющихся и используемых в системе ресурсов:

1. Учет свободных блоков ФС. Организация списка свободных блоков в виде связного списка. В каждом блоке перечислены номера свободных блоков и ссылка на следующий блок. Когда нам нужны блоки – обращаемся к головному, обнуляем ссылки, дошли до конца => больше не создаем файлы. Эти блоки размещаются в пространстве, выделенному для хранения блоков файлов. Никак не противоречит ресурсам, выделенным для хранения блоков файла. Он не занимает содержательные блоки.

2. Использование битового массива. Берем массив, содержащий любые переменные. В нем мы перенумеровываем все двоичные разряды. Программно получить i-го двоичного бита в массиве. Состояние любого блока определяется содержимым бита с номером каждого блока. Если свободен => бит = 1.  
+ Основное преимущество использования битовых массивов для учёта свободных блоков ФС? Возможность оптимизации размещения данных файлов, минимизирующих фрагментацию данных по устройству (мы можем видеть и использовать свободные области, состоящие из групп свободных блоков)  
+ Информация получается более компактная, меньше памяти.  
+ Лучше списка: можем бороться с фрагментацией по устройству.

---

#### 41. Файловые системы. Координация использования пространства внешней памяти. Квотирование пространства ФС. Надежность ФС. Проверка целостности ФС.

Квотирование/бюджетирование пространства ФС.

Пусть есть сервер, на нем работает ФС и ОС

Учет использования квот на блоки	{	Гибкий лимит блоков
		Жесткий лимит блоков
		Использовано блоков
		Счетчик предупреждений
Учет использования квот на число файлов	{	Гибкий лимит числа файлов
		Жесткий лимит числа файлов
		Использовано файлов
		Счетчик предупреждений

Рис. 113. Квотирование пространства файловой системы.

Ограничено:

1. Кол-во файлов.
2. Размер файла (ограничен организацией ФС).

=> нужны средства регламентации ресурсов. Для каждого из пользователей/групп пользователей выделяется ресурс, который пользователь может использовать.

**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 ассоциирован **счетчиком предупреждений** (гибкий лимит превышать можно, но после этого включается обратный счётчик предупреждений).

При входе пользователя в систему происходит подсчет соответствующего ресурса (числа имен файлов либо количества используемых пользователем блоков файловой системы).

- Если вычисленное значение <= гибкий лимит => счетчик предупреждений сбрасывается на начальное значение, пользователь продолжает свою работу.

- Если значение > гибкий лимит => значение счетчика предупреждений уменьшается на единицу, затем происходит проверка равенства его значения нулю.

- Если значение счётика = нулю => вход пользователя в систему блокируется,

- больше 0 => пользователь получает предупреждение о том, что соответствующий гибкий лимит израсходован, после чего пользователь может работать дальше.

=> система позволяет пользователю привести свое «файловое пространство» в порядок в соответствии с установленными квотами.

**ПРОБЛЕМА РЕЗЕРВНОГО КОПИРОВАНИЯ - МИНИМИЗАЦИЯ**  
**ОБЪЕМА КОПИРУЕМОЙ ИНФЫ БЕЗ ПОТЕРИ КАЧЕСТВА**

- 1) Выборочное копирование. Не коп. восстановление  
файлов. исп. файлы ОС, систем программирования, присущих  
архивации.
- 2) Инкрементное архивирование - создание в 1-е  
архивирование полной копии всех файлов (Master - copy).  
следующее архив. включает новые /измененные файлы
- 3) Компрессия

### ПРОБЛЕМА КОПИРОВАНИЯ НА ХОДУ

- 1) Грамотный выбор момента архивирования
- 2) Распределенное хранение резервных копий

### Физическое копирование

Подложное коп. данных с концом  
- коп. и свободные блоки

### Логическое копирование

Копирование не блоков,  
а файлов

### Интеллектуальное физ. коп.

- Проблема обработки дефектных блоков,  
которые невозможно связать с конкретными  
файлами

+ большая эффективность при  
использовании именно пары этих  
параметров.

- лишь гибкий лимит =>  
пользовательский процесс может  
«забыть» все свободное пространство  
ФС => решает жесткий лимит.
- лишь жесткий лимит => ситуации,  
когда пользователь получает отказ от  
системы, поскольку он  
«неумышленно» превзошел  
указанную квоту.

Надежность - минимизация  
вероятность безвозвратной потери  
информации.

Нарушение надежности (ошибки в  
ПО, случайно) => потери  
пользовательской инфы

### => резервное копирование и архивирование.

**1. Выборочное копирование.** Копия  
категорий файлов (например, только  
все исходные файлы).

**2. Инкрементное архивирование** -  
создается серия копий. С  
мастеркопией ассоциируется дата и  
время создания. По желанию  
пользователя создание  
инкрементных копий (чреда  
изменений относительно  
предыдущего копирования). Уровень  
потерь зависит от частоты  
архивирования.

- Компрессия при архивировании  
(риск потери всего архива из-за

- 1) Потеря актуального содержимого открытых файлов
- 2) Во время сбоя может нарушиться корректность  
системной инфы

### Организация контроля целостности блоков

- 1) В системе формируется таблица занятых и свободных  
блоков в разрезе реального коп-ва блоков файловой  
системы. Допускается
- 2) Система запускает процесс анализа блоков по  
некоторой, увелич. таблице свободных блоков на 1
- 3) Аналогично для индексных узлов
- 4) Процесс анализа содержимого этих таблиц и коррекции  
ошибочных ситуаций

### \* Непроторачиваемость

- Задание пропажа блока
- Инфа о блоке выше
- Инфа о блоке ниже
- Позже

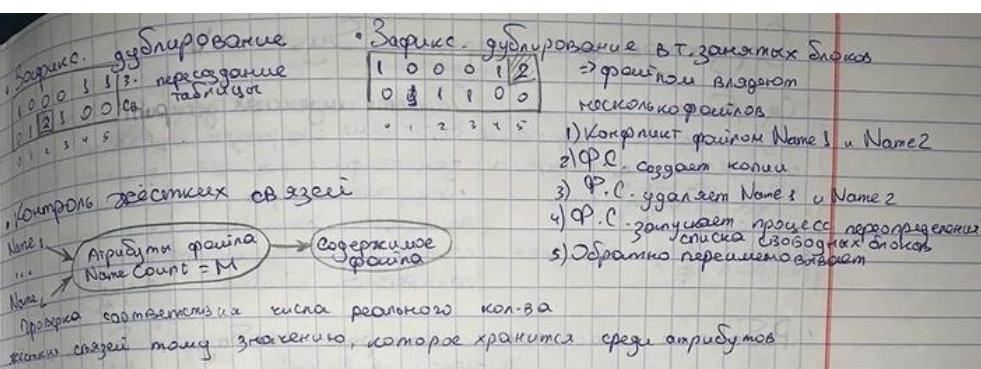
1 1 0 1 0 1	1 0 1 0 1 1
0 0 1 0 1 0	0 0 0 1 0 0
0 1 2 3 4 5	0 1 2 3 4 5
Tабл. занятых бл.	Tабл. свобод. бл.

### ошибки в чтении/записи сжатых данных)

- Проблема архивирование на ходу (во время копирования меняются файлы, создаются и удаляются каталоги).
- Распределенное хранение копий.
- Физическая архивация: «один в один», интеллектуальная физическая архивация (только использованные блоки ФС), проблема обработки дефектных блоков.

- Логическая архивация -  
копирование файлов (а не блоков),  
модифицированных после опр даты)

Потеря системной информации =>  
потеря данных для ОС и  
пользователей => тестирование  
системной информации и  
автоматизированного восстановления  
инфы => нужна избыточность



информации.

## Проверка целостности. Контроль непротиворечивости блоков ФС.

1. В системе формируются две таблицы, каждая из которых имеет размеры, соответствующие реальному количеству блоков ФС - таблица занятых блоков, таблица свободных блоков. Изначально содержимое таблиц обнуляется.
2. система запускает процесс анализа блоков на предмет их незанятости. Для каждого свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных блоков.
3. Аналогично для анализа индексных узлов.
4. процесс анализа содержимого этих таблиц и коррекции ошибочных ситуаций.

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

Файл ЮНИКС – специальным образом именованный набор данных, размещенных в системе. Поддерживает разные типы файлов. Файлы по-разному могут организовывать хранение и интерпретацию значений.

- Обычный файл как последовательность байт (regular file), как текстовый, бинарный (файл определенного типа).
- Каталог – предназначен для организации связи между именем файла и его содержимым.
- Файл устройств (special device file) – установление соотношения между именем устройства и зарегистрированным в системе драйвером.

Файл ОС Unix – специальным образом именованный набор данных, размещенных в ФС:

- Обычный файл (regular file) – файл, который имеет для пользователя в последовательной форме
- Каталог (directory) – содержит имена и ссылки на директории, содержащиеся в данном каталоге
- Спец-файл устройства (special device file) – связь в соответствствие каждому устройству через имя можно обратиться и устройству через содержащуюся в драйвере именованную папку /FIFO-файл (named pipe, FIFO file)
- файл – ссылка / символическая связь (link)
- socket (socket) – для реализации универсализированного интерфейса программирования распределенных систем

• FIFO именованный канал.

• Ссылка link

• Сокет socket.

• Нет команд.

• Те команды, с которыми мы оперируем, могут быть исполняемыми файлами, либо командой интерпретатора команд.

1. Пользователь

2. Группа пользователей (за исключением пользователя)

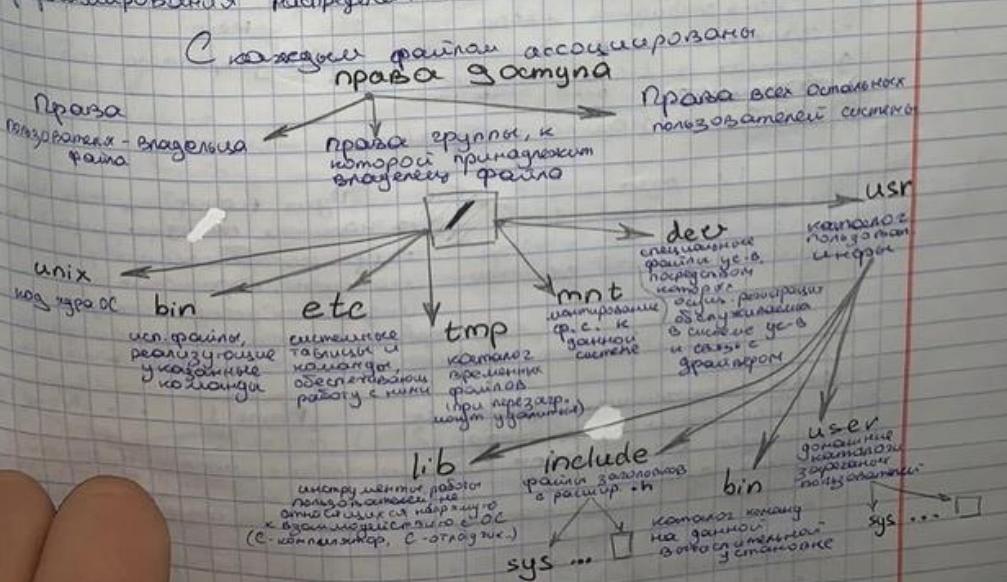
3. Все пользователи системы (за исключением группы).

Права доступа:

1. На чтение.

2. На запись.

3. На исполнение. Внутри исполнения есть магическое число – код, который считывает система перед выполнением. Если его нет => файл не может быть исполнен.



## 43. Примеры реализаций файловых систем Внутренняя организация ФС. Модель версии UNIX SYSTEM V.

Файловая система занимает часть того раздела, в котором она находится (**системным разделом** системного устройства ОС System V), начиная с нулевого блока и заканчивая некоторым фиксированным блоком. Эта часть состоит из трех подпространств:

1. **Суперблок** – часть ФС, которая резидентно находится в ОП, содержит данные, определяющие статические параметры и характеристики данной ФС (информация о размере блока файла, информация о размере всей файловой системы в блоках или байтах или же информация о количестве индексных дескрипторов в системе), хранит информацию об оперативном состоянии ФС, о наличии свободных ресурсов ФС.

- В нем находится массив фиксированного размера, в котором перечислены свободные блоки ФС.

# Структура Ф.С. версии System V

## Суперблок

- содержит данные, определяющие статические параметры и каркас линейной Ф.С.
- адреса для оперативного состояния Ф.С.
- информация о наличии свободных ресурсов Ф.С.

**Область индексных дескрипторов**  
системные структуры данных фиксированного размера

- содержит информацию о размещении, актуальном состоянии и содержимом конкретного файла

**Блоки файлов**  
рабочее пространство файловой системы

- блоки файлов с информацией о системной информации, которая не логична в предыдущих блоках

Если массива не хватает для перечисления, то один из элементов массива содержит номер блока, в котором находится продолжение массива => связанный список свободных блоков.

### - массив свободных индексных дескрипторов

со свободным ИД. Освобождение ИД => есть свободное место (номер -> элемент массива)/нет (номер забывается). Запрос ИД: поиск в массиве, массив пустой => обновление массива (не пусток => OK).

**2. область индексных дескрипторов.** – специальная структура данных Ф.С., которая ставится во взаимнооднозначное соотв с каждым файлом, содержащих комплексную актуальную информацию о размещении, актуальном состоянии и содержимом конкретного файла.

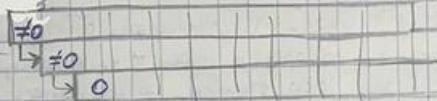
**3. блоки файлов** (рабочее пространство Ф.С.). Здесь размещаются блоки файлов (с содержимым этих файлов), а также системная информация, которая не поместились в суперблоке и области индексных дескрипторов.

**Перечислить модельную последовательность действий с ресурсами (данными) файловой системы и файлами Unix при выполнении команды удаления файла.** Например, для случая: в текущем каталоге имеется файл NAME и мы выполняем команду rm NAME

## Работа с массивами номеров свободных блоков

N блоков свободны

Выделение свободных блоков



## Блоки файлов

- номера свободных блоков в длинном связном списке в нескольких блоках
- 1-й блок в суперблоке => AOP
- Каждый блок хранит

- 1) Запрос на получение свободного блока (При освобождении изображения)
- 2) Поиск в 1-м блоке массива ячейки с содержательной информацией
- 3) Обнуление найденной ячейки
- 4) Блок с найденным номером выдается в ответ на запрос
- 5) Обнуление последней ячейки блока сформированной на след. блок массива => предварительно содержимое этого блока загружается в суперблок и исп. как 1-й блок массива

## Работа с массивом свободных индексных дескрипторов

- рядок кол-во эл-тов заполнен номерами свободных индекс
- 1) Освобождение индексного дескр. (удаление ячейки) => обращение к массиву

Если свободные места => запись номера освободившегося дескриптора в 1-е вытесненное окно массива

## 2) Создание фрейла => обращение к массиву

Не путать => из него извлекается 1-й содержательный эл-т – номер свободного индексного дескриптора

Рут, в суперблоке есть инфо о наличии свободных индексных дескрипторов => процесс обновления расположения массива, обратившись к индексному дескр. Помимо этого, номера свободных индексных дескрипторов помещаются в массив

Но между содержимым фрейла и его индексным дескриптором существует взаимодействие.

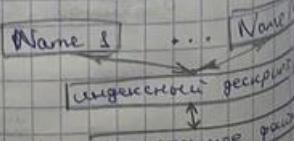
### Виды дескрипторов

- информация о типе файла
- права доступа
- временные файлы
- размер в байтах
- количество ссылок

имя и ссылка на инд. деск. (= > инд. деск. свободен)

статическая ячейка (время создания, последней модификации)

массив блоков файла



## 1. Получаем номер индексного дескриптора из записи каталога

## 2. Уменьшаем значение поля Count, содержащее количество ссылок из каталогов к данному ИД

## 3. Если значение Count = 0, освобождаем блоки удалённого файла (номера поступают в список свободных блоков), освобождаем данный ИД (он считается свободным, как только счётчик стал равен нулю, пытаемся поместить номер свободного индексного дескриптора в массив суперблока), удаляем запись в текущем каталоге. Стоп

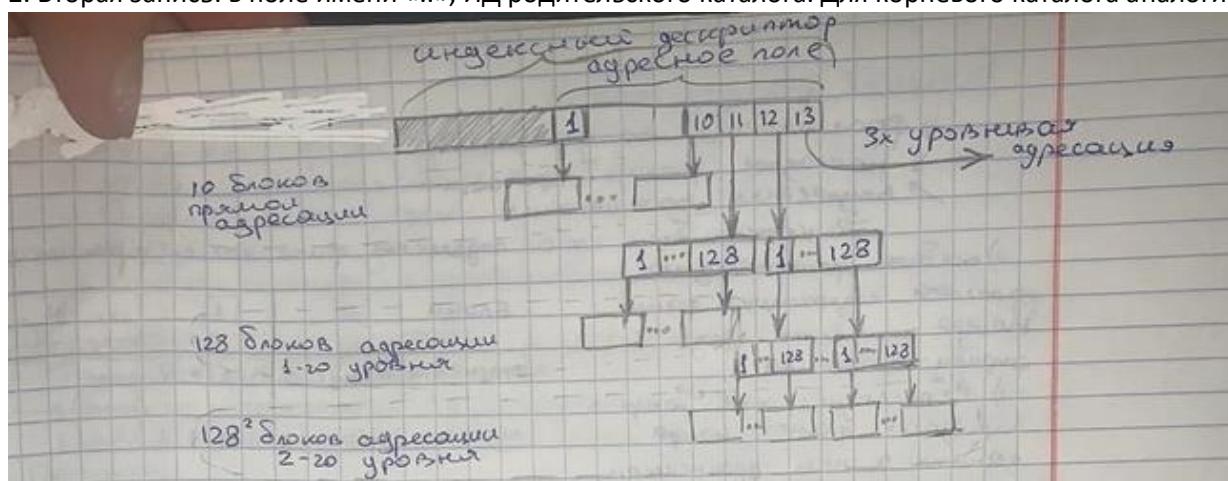
## 4. Если значение счётчика больше нуля, удаляем запись в текущем каталоге. Стоп

**Каталог Ф.С. версии System V** – это файл специального типа; его содержимое так же, как и у регулярных файлов, находится в рабочем пространстве Ф.С и по организации данных ничем не отличается от организации

данных регулярных файлов. Ф записи 2 поля фикс размера: поле ИД, имя файла.

1. Первая запись: в поле имени – имя «..». ИД самого каталога.

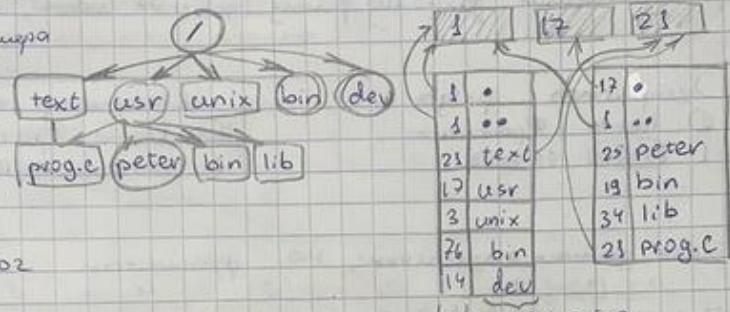
2. Вторая запись: в поле имени «..», ИД родительского каталога. Для корневого каталога аналогично первой.



### Каталог

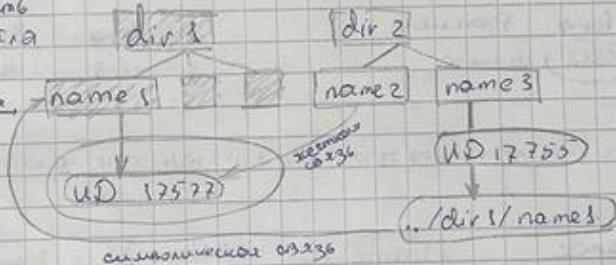
- запись фикс размера
- при создании получаем 2 предопр. записи:

'.' - ссылка на same каталог  
'..' - ссылка на родительский каталог



### Средства установления связей

- Жёсткая связь позволяет
- с 1 инд. деск. связь
- > 2 равноправных роли
- Для связи связь создаётся физич. ссылка, содержащим путь к файлу

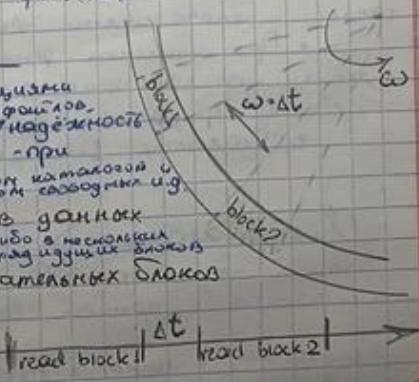


### System V

- + иерархична
- + системное кэширование  $\Rightarrow$  оптимизация работы с массивом свободных блоков и свободных инд. деск.
- + "Нарастающая" коэффициент адресации
- В суперблоке хранится информация о файловой системе  $\Rightarrow$  его потеря - проблема
- Система обрабатывает только суперблока на диске  $\Rightarrow$  обращение к 1 и той же логике дискового пр-ва  $\Rightarrow$  выход из строя этого блока
- При интенсивном работе блоки в файле разбросаны по всему доступному дисковому пр-ву  $\Rightarrow$  много ненужных движений головки  $\Rightarrow$   $\downarrow$  эффективность работы ФС
- Ограничение на длину имени файла

### Fast File System (FFS) BSK

- Разбиение последоват. дисковых блоков на порции или блоки, в каждом - конкт. суперблоко, конкт. файлов, блоков, инфа о свободных блоках  $\Rightarrow$  надёжность
- 1) Оптимизация размещения каталогов - при создании каталога выбирается класс с мин. кол-вом свободных блоков
  - 2) Равночленость использования блоков данных
  - 3) Размещение последовательных блоков отступом на с. д.т.



## Большой размер блока

- ↓ фрагментации по диску
- ↓ начальные расходы при чтении
- внутренняя фрагментация
- ⇒ каждый блок ф.с. поделен на фрагменты

о) все блоки кроме последнего

заполняют содержимым файла.

Номера этих блоков среди

выделяются фрагментами

и) в последнем блоке есть номер в

записях файла, номера

записей в нем фрагментов,

принадлежащих дальному файлу (единичная маска / в 2-го запись про-

Блоки	0	1	2	3	...		N		
Фрагменты	0	1	2	3	4	5	6	7	...
Маска	0	0	0	1	0	1	1	1	...

## Дескриптор ретуа.txt

ID Розмер Фрагмент

1 5120 00 04 08

## Дескриптор насыа.ехт

ID Розмер Фрагмент

2 4608 12 16 10

Блоки данных, разделенные на фрагменты

0	1	2	3	4	5
00 01 02 03 04 05 06 07 08 09 10 11	12 13 14 15 16 17 18 19 20 21 22 23				



ID Розмер Фрагмент

5 5632 00 04 20

ID Розмер Фрагмент

2 4608 12 16 10

0	1	2	3	4	5
00 01 02 03 04 05 06 07 08 09 10 11	12 13 14 15 16 17 18 19 20 21 22 23				

### Команды:

Индексного дескриптора

Размер записи

длина имени файла

Имя файла  
до 255 символов

Name 1 } Size(Name 1)

Name 2 }

Name 3 }

Name 4 }

Name 5 }

... =

Name 6 } Size(Name 6)

Name 7 }

Name 8 } Size(Name 8)

Name 9 }

Name 10 }

... =

RM Name 2  
RM Name 4

Name 3 }

Name 5 }

... =

### Команды:

Состоит из записи презентной длины

На только одна запись содержит номер индексного дескриптора

запись и имя файла

При удалении файла из каталога освобождающееся место

записи пр-ко присваивается к предыдущей записи

размер записи, длина хранимого имени не меняется

Удаление 5й записи ⇒ обнуление индексного деск

rent()

Исключительная блокировка (exclusive lock)

Любой другой процесс не может заниматься с данной блокировкой

Блокировка с блокировками не пересекаются

Распределенная блокировка

Области могут пересекаться

В ФС s5fs утерян суперблок. Какая минимальная информация необходима для полного автоматического восстановления всех файлов?

- Размер области индексных дескрипторов (или их предельное количество) Какая информация может размещаться в области блоков ФС fs5? х Содержимое блоков файлов, списки свободных блоков ФС, ссылки на номера блоков, занимаемых файлами.

#### **44. Управление внешними устройствами. Архитектура организации управления внешними устройствами, основные подходы, характеристики.**

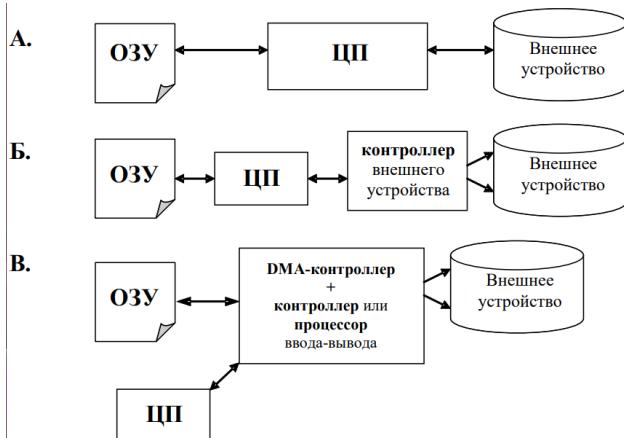


Рис. 144. Модели управления внешними устройствами: непосредственное (А), синхронное/асинхронное (Б), с использованием контроллера прямого доступа или процессора (канала) ввода-вывода (В).

**1. Непосредственное управление ЦП внешними устройствами**, когда процессор на уровне микрокоманд полностью обеспечивал все действия по управлению внешними устройствами. Синхронное управление: если начался обмен, то, пока он не закончится, процессор не продолжает вычисления (поскольку занят обменом).

**2. Использование специализированных контроллеров устройств**, между ЦП и соответствующими внешними устройствами => ЦП может работать с более высоконивневыми операциями при управлении внешними устройствами => он частично освобождался от потока управления внешними устройствами за счет того, что вместо большого числа микрокоманд конкретного устройства он оперировал меньшим количеством более

высоконивневых операций. Синхронная. Поток данных идет через процессор.

**3. Развитие предыдущей модели до асинхронной** благодаря появлению аппарата прерываний. Можно 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 в системе возникнет прерывание, сигнализирующее возможность дальнейшего выполнения 1-го процесса. Поток данных идет через процессор.

**4. Использование контроллеров прямого доступа к памяти (DMA-контроллеров, Direct Memory Access).**

Решение проблемы перемещения обработки потока данных из процессора. ЦП генерировал последовательность управляющих команд, указывая координаты в ОП, куда надо положить или откуда надо взять данные, а DMA-контроллер занимался перемещением данных между ОЗУ и внешним устройством => поток данных шел в обход процессора.

**5. Модель, основанная на том, что управление внешними устройствами осуществляется с использованием специализированного процессора** (или даже специализированных компьютеров) или каналов ввода-вывода => снижение нагрузки на ЦП с точки зрения обработки потока управления, он теперь оперирует функционально-емкими макрокомандами.

Архитектура программного управления внешними устройствами, каждый уровень строится на основании нижележащего уровня.

1. В основании лежит **аппаратура**, а далее следуют программные уровни:
2. программы обработки прерываний,
3. драйверы физических устройств,
4. уровень драйверов логических устройств.

Программное управление внешними устройствами:

1. Унификация программных интерфейсов доступа к внешним устройствам (унификация именования, абстрагирование от свойства конкретных устройств).
2. Обеспечение конкретной модели синхронизации при выполнении обмена (минхронный, асинхронный обмен)
3. Обработка возникающих ошибок (индикация ошибки, локализация, попытка исправления ситуации)
4. Буферизация обмена (может быть внутренняя => скрытый буфер => есть специальные диски, куда

скидывается информация в случае сбоя. Буферизация ОС, она сама организует буфер).

5. Обеспечение стратегии доступа к устройству (распределенный доступ – распределенный диск, монопольный доступ – ус-во предоставляет в монопольном режиме).

6. Планирование операции обмена.

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

Стратегии организации планирования дисковых обменов

- Очередь запросов: 4, 40, 11, 35, 7, 14
- Изначально на 15-й дорожке
- (FIFO) - порядок появления запросов в очереди

Путь головки	L
15 → 4	33
4 → 40	36
40 → 11	29
11 → 35	24
35 → 7	28
7 → 14	7

Итого: 135 дорожек  
Средний путь: 22.5

2) LIFO

+ Цепочки связанных обменов:  
Считывание → изменение → перезапись

3) SSTF - shortest Service Time First | 15-14-11-7-4-35-40 | 67

Поиск дорожки с мин. перемещением  
- Голодание крайних дорожек

4) PRIO - приоритеты процессов  
- Проблема организации в подаче приоритетов

5) SCAN-алгоритм сканирования | 15-35-40-14-11-7-4 | 61

Головка перемещается сначала в одну сторону диска, выбирая каждый раз из очереди запрос с номером ближайшей головкой дорожки. Затем в другую  
- Может привести к деградации системы из-за голодания процессов из-за интенсивного обмена с неколькими областями

6) C-SCAN - циклический запрос с min/max номером | 15-4-2-11-14-35-40 | 47

7) N-step SCAN

- 1) очередь делится на N подочередей
- 2) выбирается и обрабатывается очередь
- 3) во время обработки попадание в эту очередь новых запросов блокируется
- 4) другие очереди получают залывки

+ Нет проблем голодания

Основное преимущество алгоритма планирования дисковых обменов 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)

Основная характеристика алгоритма SSTF – «жадный» алгоритм планирования дисковых обменов? x На каждом шаге выбирается обмен, требующий минимального перемещения головок диска.

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

**RAID** (Redundant Array of Independent Disks) - система из нескольких дисковых УС-В, которая представляется в ОС как 1 УС-БО, имеющей логическую структуру организации параллельных объемов и обработки избыточной информации для контроля и восстановления информации. Различается на разных УС-Вах, состоящих из этих дисковых полосок RAID имеет, горизонтальных файлов различного размера, которые осуществляются объемом в таких системах.

RADIO:

- С некоторыми дисками можно добиться параллельности обеих

- Ус-ва независимост
  - Нем избогатилось в широте  $\rightarrow$  Довідкові джерел

0	1	2	3
4	5	6	7
8	9	10	11

RAID 3 - 2 накопителя гибкое:

- При записи сокращение на диске и будапер
  - При чтении запрос к 1 из дисков
  - При утере обращение к будаперу
  - Дорого (из-за обслуживания)

0	0
1	1
2	2
3	3

RAID2 - модель с синхронизацией  
• Код Хемминга

- RAID3 под. с синх. головками  
один из дисков хранит полото, дополнительные

$b_0$	$b_1$	$b_2$	$b_3$	$f_0(b)$	$f_1(b)$	$f_2(b)$
-------	-------	-------	-------	----------	----------	----------

RAID3 изг. с синх. головками

- Очки из дисков хранят номера, дополнительные  
ко кемпингу группе номера
  - $X_{\text{с}}(i) = X_{\text{в}}(i) \oplus X_{\text{с}}(i) \oplus X_{\text{з}}(i) \oplus X_{\text{б}}(i)$
  - При номере данных  $X_{\text{с}}(i) = X_{\text{в}}(i) \oplus X_{\text{с}}(i) \oplus X_{\text{з}}(i) \oplus X_{\text{б}}(i)$

$b_0 \ b_1 \ b_2 \ b_3 \ p(b)$

RAID4 изображение RAID3

- классифицированы по времени выполнения задач
  - $X_4(i) = X_0(i) \oplus X_3(i)$
  - $X_4\text{new} = X_4(i) \oplus X_1(i) \oplus X_{\text{new}}(i)$  после добавления нового

0	1	2	3	P(0-3)
4	5	6	7	P(4-7)

RAID5 распределяет избыток избыточным обр.

- RAINS распределен  
импульс по дискам акустическим др.

4	5	6	$P(4-2)$	?
8	9	$P(2-4)$	10	11

ищет по дискам + надежность

- RAD6 - двухуровневая избыток.

0	1	2	3	$P(0)$
4	5	6	$P(4-7)$	$P(4-7)$
8	9	$P(8-12)$	$\sim 1$	$\sim 1$

RAID6 - двухуровневый избыточный.

- инфо  
+ / настройка

0	1	2	3	$P(0-3)$	$Q(0-3)$
4	5	6	$P(4-7)$	$Q(4-7)$	7
8	9	$P(8-10)$	$Q(8-11)$	10	11

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

**Драйвер** фильтрует  $y=0$  в системе и определяет  
имперфекции

**Блок-ориентированные** виды  $y=0$ , с которыми  
он взаимодействует блоками  $bdelsw$

**БЛОК-ориентирован**  
 $y=0$ , содержащий различные структуры  
Подсистема  $bdelsw$  генерирует блоки

Оба рассматриваются как блоки и блоки -  $0$ .

В системе имеется с драйвером  $y=0$ :

- Инициализация (запуск драйвера  $y=0$ )
- Регистрирование инициализации с драйвером
- На момент блоков фазного хранения в реальном времени  $y=0$   $PC$ .
- Содержательная информация только в фазовом времени: 2 горизонт.
- Стартовый цикл - из драйвера в подсистему  $dp$ , содержит  
инициализацию  $y=0$
- Инициализации момент - гор. информ., передаваемая драйверу  
при обращении

$\Rightarrow$  1 драйвер может управлять несколькими  $y=0$  блоками

Каждый блок Хранит конфигурацию  $y=0$  с  
указанием типа на действующие точки входа (реализуемые функции)  
и соответствующий драйвер / ссылки-запущенные на момент запрашивающий

Типовые функции точек входа в драйвер

- 1) open(), close()
- 2) read(), write()
- 3) ioctl() - настройка и управление драйвером
- 4) interrupt() - при поступлении прерывания, ассоциированного с  $y=0$

Все устройства, с которыми легально работает система, регистрируются с помощью рег своих драйверов и ассоциированных с ними файлов устройств. В нем нет блоков, вся инфа находится в индексном дейскрипторе: владелец, права, статистика, нет адресной инфы => вместо не инфа об устройстве. Система рассматривает **блок-ориентированные устройства** (диск, обмен порциями байтов) и **байт-ориентированные устройства** (ОП, обмен порциями байтов). Одно и т же физическое устройство может представляться и так, и так.

Инфа, которая размещается в ИД файла устройств кроме кол-ва ссылок на это устройство есть инфа:

1. Старший номер – целочисленное число. Данное устройство ассоциируется с драйвером, зарегестр в системе под этим номеров
  2. Тип устройства блок-ор, байт-ор., для каждого устройства своя

таблица: bdevsw, cdevsw.

### 3. Младший номер устройства.

Каждая запись – **коммутатор** – структура фикс размера, которая содержит перечень точек входа в соотв драйвер => адреса точек входа в драйвер – набор системных вызовов для взаимодействия с ФС (открыть закрыть читать и тд). Драйверы каких то устройств могут обрабатывать и реализовывать все точки входа. Если какие то входы не задействованы в коммутаторе, там стоит заглушка (вызовет ошибку, 0 и др).

В точках входа есть стандартные операции работы с устройством (read/write)

Вход для обеспечения настроек драйвера

Есть вход для прерывания

При регистрации драйвера система запоминает жти точки и передает туда управление при возникновении прерывания.

### Обращения к драйверам:

1. Старт системы
2. Обработка ввода/вывода
3. Обработка прерывания, связанного с данным устройством.
4. Выполнение специальных команд управления.

**Ручкики драйвера**

- 1) синхронные – top half
- 2) асинхронные – bottom half

**Семафоры, вызываемые обращение к ф-ям драйвера**

- 1) start system, инициализация ус-ва и драйвера
- 2) Обработка запросов на обмен
- 3) Обработка прерываний, связанного с данным ус-вом
- 4) Выполнение специальных команд управления ус-вом

**Модель динамического связывания драйверов**

в системе присутствуют программные средства, позволяющие динамически подключить др. к ОС. Решают задачи:

- 1) именование драйвера
- 2) иниц. драйвера (организование системных областей др.)
- 3) иных устройств (приведение ус-ва в начальное состояние)
- 4) добавление драйвера в соответств. таблицу драйверов
- 5) установка обработчика прерываний.

**Если ор-ка отсутствует => на ее месте в коммутаторе стоят заглушки**

- 1) nulldev() – при обращении сразу возвращают управление
- 2) nodev() – возврат управления с кодом ошибки

**Ручкики драйвера**

- 1) синхронные – top half
- 2) асинхронные – bottom half

**Семафоры, вызываемые обращение к ф-ям драйвера**

- 1) start system, инициализация ус-ва и драйвера
- 2) Обработка запросов на обмен
- 3) Обработка прерываний, связанного с данным ус-вом
- 4) Выполнение специальных команд управления ус-вом

**Модель динамического связывания драйверов**

в системе присутствуют программные средства, позволяющие динамически подключить др. к ОС. Решают задачи:

- 1) именование драйвера
- 2) иниц. драйвера (организование системных областей др.)
- 3) иных устройств (приведение ус-ва в начальное состояние)
- 4) добавление драйвера в соответств. таблицу драйверов
- 5) установка обработчика прерываний.

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

1. Таблица индексных дескрипторов открытых файлов (в памяти ядра ОС). ТИДОФ.
2. Таблица файлов (в памяти ОС). ТФ. Фактически количество открытых единовременно в системе файлов.
3. Таблица открытых файлов ТОФ.

Таблицы создаются на ресурсах ОС.

**Таблица открытия файлов** создается в адресном пространстве процесса, каждая запись имеет уникальный идентификатор = № записи в ТОР

Запись содержит ссылку на номер записи в **Таблице файлов ОС** - системная таблица, в которой регистрируются все открытые в системе файлы. Хранится в ОЗУ. Содержит:

- указатель чтения/записи
- ссылка на позицию в файле, начиная с котрой ...

- статистика яркости

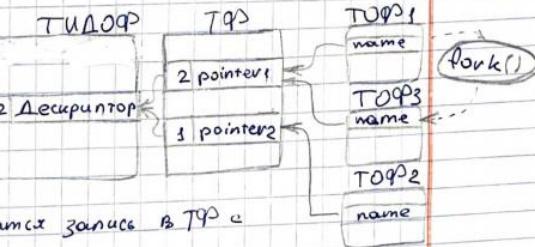
- ссылка на таблицу ид. дескр. откр. файлов

**Таблица индексных дескрипторов открытых файлов** - системная структура данных. Хранится в ОЗУ. Содержит:

- активная копия открытого в системе иду. дескр.

- Порядок (т.е. статистика яркости)

- Запускем процесс 1
- Система сформировала ТОР
- Пр. обращается к open() и открывает файл name
- В Таблице ТИДОР заводится файловый дескриптор для родителя с данным файлом



- Файл открыт впервые => заводится запись в ТОР с ссылкой на запись в ТИДОР
- указатели копий/записи копир. яркости: 1 => в записи ТОР ассоциирована 1 запись ТИДОР

- В ТИДОР активная копия индексного дескриптора открытого файла

запущенный оперирует с данными из ТИДОР

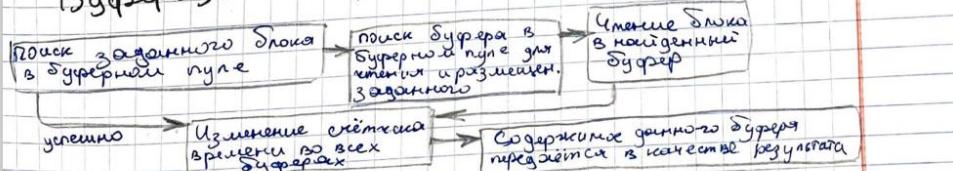
+ТИДОР в ОЗУ => числа обращений к пр-ву

индексных дескрипторов ф. с. => эффективность

- При сбое в ОС содержимое ТИДОР потерянно

=> потеря в ф. с.

### Буферизация при блок-ор. обмене



### Кэширование дисковых обменов

+ минимизация реальных обращений к диску у-важ

=> эффективность

- несогласованное реальное содержимое диска и

мено, что должно на нем быть

- => при сбое системы возможны потери информации в ОП

(теряется индексный дескриптор => теряется список блоков файла)

Предположим в ОС UNIX System V заблокировано кэширование обменов с блокориентированными устройствами. Какое предельное количество непроизводительных обменов (дополнительных обменов) может произойти при чтении блока N файла (считаем, что файл уже открыт).

Три. Система адресации блоков файлов в индексном дескрипторе использует тройную косвенную адресацию.

### 49. Управление оперативной памятью. Одиночное непрерывное распределение. Распределение разделами. Распределение перемещаемыми разделами.

Управление ОП.

ОС обеспечивает распределение и контроль за ресурсами ОП:

1. Контроль состояния каждой единицы памяти (свободна/распределена)
2. Стратегия распределения памяти (кому когда и сколько)
3. Выделение памяти (выбор выделяемой области)
4. Стратегия освобождения памяти (после освобождения процессов ОС забирает ее окончательно или временно).

1. Аппаратная составляющая: те аппаратные интерфейсы, которые позволяют управлять ОП.
2. Программная реализация – все алгоритмы, программы ОС, которые используют аппаратные интерфейсы для решения задач.

#### **Стратегии и методы управления:**

##### **1. Одиночное непрерывное распределение**

- Предполаг, что имеется физическое адресное пр-во. Часть выделено под ОС, часть – для выполнения 1 процесса. Нужны:

1. Регистр границ – в который мы можем установить адрес границ ОС.

2. Режим ОС. Если в УУ формируется адрес, заходящий в ОС, то команда может выполняться только в режиме ОС. Иначе – прерывание.

+ Простота.

- Часть памяти не используется.

- Нет мультипроцессирования

- Ограничение на размеры ОП для программы

- Незащищенность ОС: процесс потенциально может залезть в адресное пр-во ОС (MS-DOS, незащищена). Не критично, т.к. система однопроцессная.

##### **2. Распределение неперемещаемыми разделами**

Есть выделенная область, в которой находится ОС

- Оставшееся пространство разделяем на разделы предопределенного размера

- В каждый раздел может быть загружен только 1 процесс

Нужны 2 регистра границ: в каждый момент времени содержат адрес начала и адрес конца раздела исполняемого в данный момент процесса (для предотвращения несанкционированного доступа в соседние процессы).

Нужны ключи защиты.

Алгоритмы:

1. для каждого раздела формируем свою очередь процессов. ОС оценивает необходимую для процесса ОП, помещает процесс в ту или иную очередь. FIFO, при освобождении раздела. М.б. незагруженные и перегруженные очереди => неравномерная загрузка процессов и простой.

2. Одна очередь процессов.

- Освобождается раздел => ОС смотрит очередь с головы и помещает первый процесс в этот раздел (который может быть загружен) => несоответствие размеров. Большой раздел занят маленьким разделом => деградация системы.

- После освобождения ищем по всей очереди процесс максимального размера, который туда помещается => голодание, дискриминация маленьких процессов.

- Оптимизация. Каждый процесс имеет счетчик дискриминации.

- + мультипроцессирование.
- + простая аппаратная поддержка
- + простые алгоритмы
- фрагментация
- Ограничение размерами физической памяти
- Статичность разделов.

##### **3. Распределение перемещаемыми разделами.** По сути первая модель с виртуальной памятью.

Берем систему, ее загружаем процессами. Во времени разделы, занятые процессами, заканчиваются и освобождают память =>

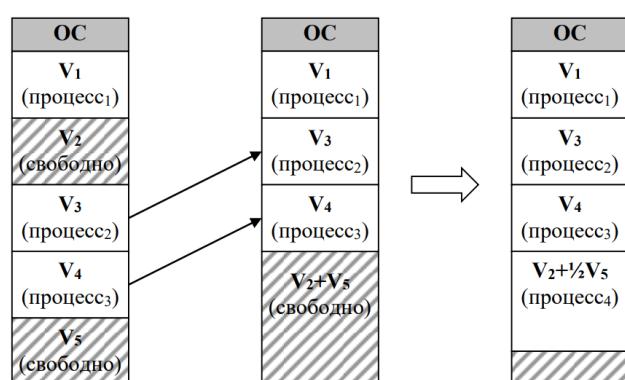


Рис. 133. Распределение перемещаемыми разделами.

проблема фрагментации => системы адресации с базированием:

- В ЦП выделяется специальный регистр базы (мб >1).
- Если мы имеем Аисп, он либо абсолютный, либо относительный (тогда к нему добавляется значение регистра базы) => физический адрес.

Аппаратные средства: регистр раниц + регистр базы

+ борьба с фрагментацией (можно запускать процесс компрессии).

- Ограничение доступной физической памяти.

- Процесс загружает весь и находится в ОП на протяжении всей обработки. Реальный процесс никогда не использует всю ОП, а только локализованно, часть.

- Затраты на перекомпоновку.

- Не годится для интерактивного процессирования с пользователем (попали на перекомпоновку -> паузы и задержки).

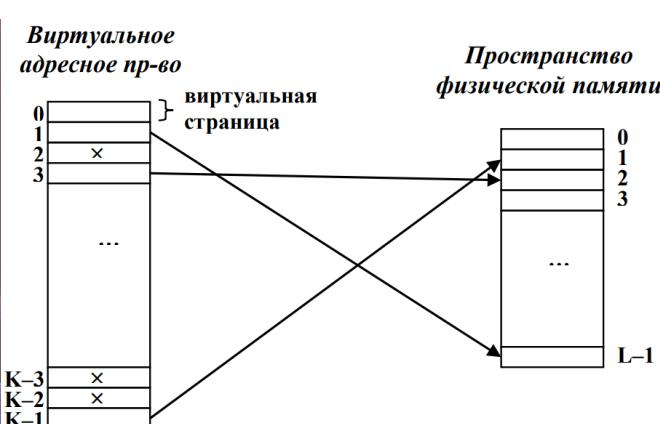
## 50. Управление оперативной памятью. Страницное распределение.

- Есть виртуально адресное пространство – модель адресации, которая используется внутри одного процесса. Размер определяется разрядностью системы  $2^L$ , L – разрядность процессора. Пространство каждого из процессов.

- Есть пространство физической памяти. Предельный объем физической памяти, которая подключена к компьютеру, определяется шиной адресов: размер специального регистра, на который попадает физический адрес.

Объём физ и вирт соотносятся как угодно.

В каждый момент времени в системе используется вирт адресное пространства исполняемого в данный момент процесса, и физическая память.



Стр и вирт и физ адресное пр-во разделены на блоки фиксированного размера (размер – степень 2) – страницы. Размеры этих страниц одинаковые.

Позволяет проводить отображение вирт адреса (неявное базирование, номер вирт страницы + смещение от этой страницы) в физ адреса (номер физ страницы + смещение). По сути отображение номера виртуальной страницы в номер физ. Используется таблица страниц, размер (гигантский) = кол-во виртуальных страниц в виртуальном адресном пространстве, предопределено. I-я строкка соотв информации об i-й виртуальной страницы выполняемого в данный момент страницы. Содержимое – либо номер физической страницы, либо инфы о том, что этой страницы в памяти нет.

Большой размер таблицы => при смене процессов мы должны

менять таблицу страниц, таблицу изменяемого процесса мы должны сохранить => долго.

Держать все таблицы в ОП плохо (реальных процессов много, преобразование виртуального адреса в физический + обращение в ОП = полное погашение производительности).

Необходимые аппаратные средства:

1. Полностью аппаратная таблица страниц => быстрое преобразование, но проблема при смене процессов (сохранение исполнявшегося процесса, загрузка новой таблицы) => накладные расходы.

2. Таблица страниц в ОЗУ + регистр начала таблицы страниц в ОП исполняемого процесса => простота, управление смены контекстов, медленное преобразование.

3. Гибридные решения:

**TLB – буфер быстрого преобразования адресов.** На самом деле кэш.

1. Пусть в ЦП есть аппаратная таблица. Размер мб не очень большой (...-32-64 строки). Каждая строка содержит номер вирт и номер физ страницы. Виртуальный адрес состоит из номера виртуальной страницы (VP) и смещения в ней (offset).

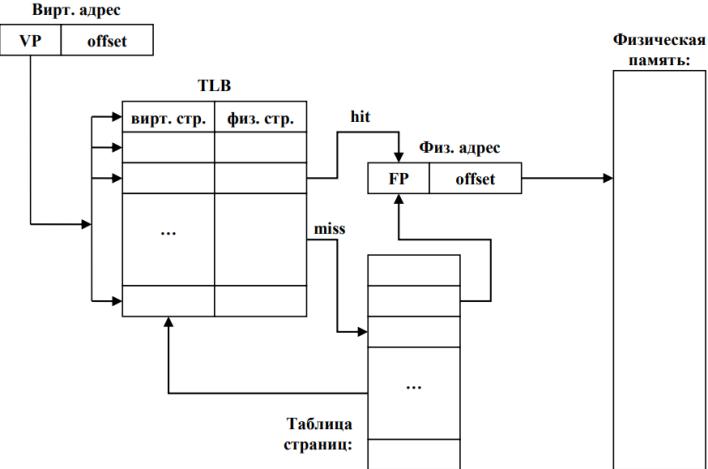


Рис. 136. TLB-таблица (Translation Look-aside Buffer).

Есть виртуальный испр. адрес, это адрес страничный = предопределенная структура. Какое то кол-во выделено под смещение внутри страницы, какое – то – под номер страницы.

УУ ЦП вытаскивает номер виртуальной страницы.

В ЦП есть аппаратный блок сопоставления полученного номера виртуальной страницы с содержимым TLB. Успешный поиск => заменяем номер вирт страницы на номер физ страницы с помощью таблицы. Если там нет такой страницы происходит обращение к таблице страниц процесса в ОП, находим строку в этой таблице, индексируемся, обновляем таблицу (выкинуть строку и заменить на новую).

Какую запись обновлять: 1. Случайный выбор. 2.

Добавить колонку «интенсивность использования», туда записывать характеристики использования.

Возможна организация отработки промаха без прерываний, когда система самостоятельно, имея регистр начала программной таблицы страниц, обращается к этой таблице и осуществляет в ней поиск.

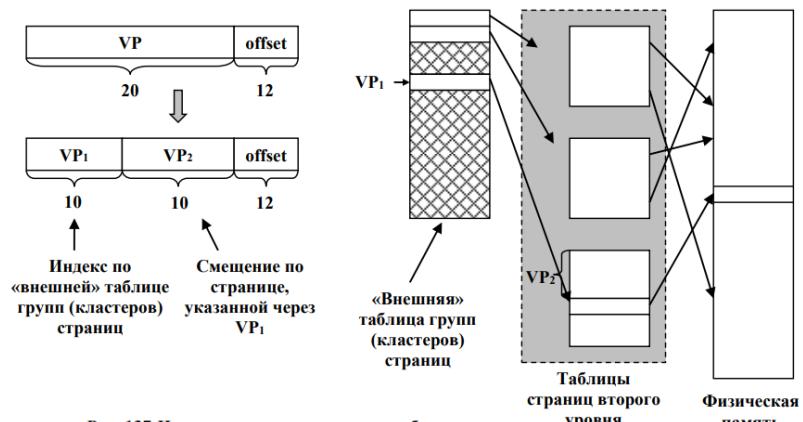


Рис. 137. Иерархическая организация таблицы страниц.

Возможна модель с прерыванием, когда при промахе возникает прерывание, управление передается ОС, которая затем начинает работать с программной таблицей страниц, и т.д. (менее эффективная, прерывание => накладные расходы).

**Модель иерархической организации таблицы страниц** - одно из решений, позволяющих снизить размер таблицы страниц. 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.

Мы разделяем поле виртуальной страницы пополам (не обязательно пополам) => подполе размером 10 бит. Инфа таблицы страниц: Есть внешняя таблица страниц. По ней мы будем индексироваться первым полем, полученным в результате разделения. Размер внешней таблицы страниц –  $2^{10}$ . Таблица второго уровня –  $2^{10}$ . Когда начинаем преобразовывать виртуальный адрес в физический, мы считаем, что внешняя таблица всегда в ОП. Мы индексируемся по ней по значению 1-го разделенного поля. В этой строчке инфа либо о том, что таблицы второго уровня нет в памяти => ее нужно подкачать в ОП. Если есть, то значение в этой строчке – адрес начала соотв таблицы страниц 2-го уровня => по этой странице мы индексируемся по значению 2-го поля => получаем номер физической страницы (либо инфу о том, что ее нет) => преобразовываем.

От инфы таблицы страниц всегда в ОП таблица первого уровня.

Есть множество таблиц 2-го уровня, но они не обяз. в ОП (там некоторое кол-во страниц, зависит от настроек).

+ Нет необходимости держать в памяти всю таблицу

+ Локализация при формировании потока запросов перегрузки происходят не так часто. При смене процессов мы не перемещаем 2 таблицы из миллионов строк, а перемещаем 1 маленькую таблицу.

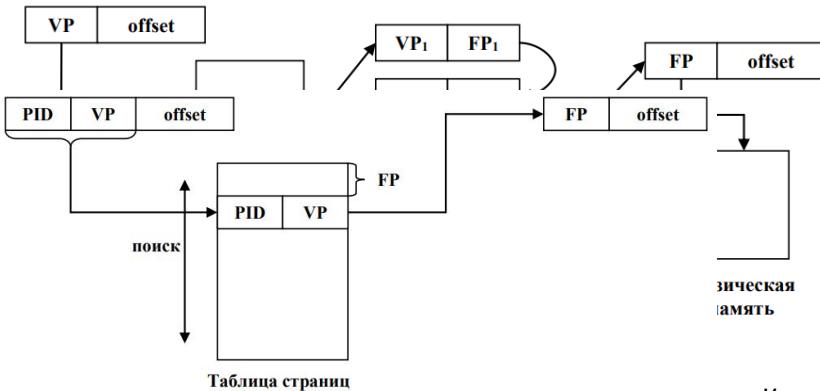


Рис. 139. Инвертированные таблицы страниц.

**Использование хеширования.** Модель основывается на функции расстановки (хеш функции – функции расстановки), отображает аргумент в ограниченное множество значений. Хеш таблица ограниченного размера, определяет ограничение на функцию размера. Меньше размер => больше вероятность коллизий. Используется список из пар вирт-физ страницы, в этом списке функция расстановки дает для всех вирт страниц одно и то же значение.

+ Качество зависит от хеш таблицы. Чем меньше, тем больше проблем.

### Инвертированные таблицы страниц.

- Каждая запись в таблице страниц содержит pid и номер виртуальной страницы. Номер записи = номер физической страницы.
- Преобразование: получаем вирт адрес.

Что описывает содержимое инвертированной таблицы страниц? Распределение физических страниц компьютера между виртуальными страницами обрабатываемых процессов.

Что определяет количество записей (размер) инвертированной таблицы страниц? Количество записей определяется количеством имеющихся в компьютере физических страниц.

- + Единая таблица, которая дает информацию, в какой физической странице находится вирт страница какого процесса – нет таблиц страниц процессов.

+ Решение проблем смены страниц при смене процессов.

- Проблема поиска по таблице (одно из решений - хеширование)

- Проблема замещения страниц, когда надо загрузить страницу извне в ОП => нужно что-то выкинуть:

1. **Not recently used** – не использовавшийся в последнее время. Используются биты R (обращения) и M (изменения). Значения признаков устанавливаются аппаратно при чтении/записи, программно мы можем их обнулять.

Изначально при запуске процессов все признаки обнуляем.

Через предопределенные промежутки времени по прерыванию по таймеру ОС обнуляет все виды R

– Класс 0: R = 0, M = 0, не происходило обращение и изменение в последнее время

– Класс 1: R = 0, M = 1. – Класс 2: R = 1, M = 0. – Класс 3: R = 1, M = 1.

При возникновении ситуации принятия решения о замещении анализируем признаки всех страниц.

Выбираем случайную страницу из непустого класса с минимальным номером.

2. **FIFO** – для каждой страницы формируется время пребывания в таблице. Самую старую страницу выкидывают. Не обязательно самая старая страница не используется => неправильное решение.

3. **Модификация FIFO** – очередь из страниц, выбираем самую старую страницу. Tckb K = 0 => замещаем, =1 => обнуляем и переходим в след.

4. **Модификация часы.** Все страницы размещены по круговому списку. Есть маркер, передвигается по часовой стрелке. Если маркер смотрит на стр, R = 0 => замещение, = 1 => замещение и идем дальше.

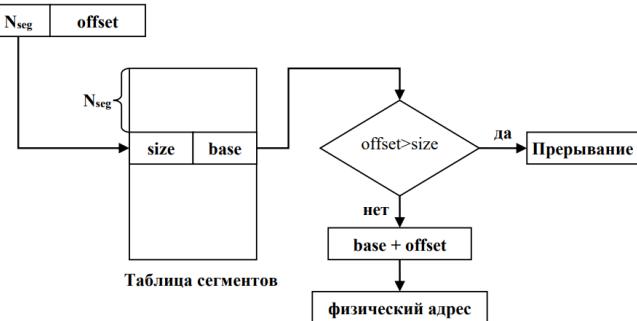
5. **Not frequently used.** Определим счетчик для каждой страницы. По таймеру будем прибавлять к счетчику соотв значение признака R => счетчик аккумулирует интенсивность работы.

При прерывании выбираем страницу с мин.значением

Алгоритм помнит старую активность. При таком сложении возможно переполнение.

6. **Модификация NFU:** сдвиг на 1 разряд и добавляем логически значение R.

## 51. Управление оперативной памятью. Сегментное распределение.



ис. 141. Сегментное распределение.

При реализации страничной организации процессу выделяется единый диапазон виртуальных адресов от 0 до предельного значения => в процессе есть команды, статические переменные (разные объединения данных с различными харками использования), есть куча => логическая структура памяти процесса НЕОДНОРОДНО => статическое разделение единого адресного пространства: выделяются область для команд, область для размещения данных, область для стека и кучи. Часто стек и куча размещаются в единой области памяти, причем стек прижат к одной границе области, куча — к другой, и «растут» они навстречу друг другу => 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.

**Сегментное разделение памяти** представляет каждый процесс в виде совокупности сегментов, каждый из которых может иметь свой размер и виртуальную адресацию (0 до N-1). 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.) => виртуальный адрес = номер сегмента и величина смещения в нем.

В ЦП есть аппаратная таблица сегментов, кол-о строк = предельному кол-ву сегментов, которые обрабатываются и/или реализуются в данном компе. Каждая строка этой таблицы содержит пару значений:

1. Адрес начала сегмента в физической памяти (база)
2. Размер сегмента.

### Алгоритм

1. Есть исполнительный виртуальный адрес = номер сегмента + смещение.
2. ЦП выбирает номер сегмента, инлексируется по таблице сегментов => строка-описание данного сегмента.
3. Проверяем, не выходит ли виртуальный адрес за пределы сегмента, больше => прерывание. Меньше либо равно => к базовому адресу добавляем смещение, получаем физический адрес.

- + Простая, просто реализуется.
- + Быстрая.
- Весь сегмент должен находиться в ОП.
- Сегментная память возвращает актуальность решения проблемы фрагментации.
- Весь сегмент в памяти на протяжении всего процесса находится в памяти.
- Накладные расходы при своппинге – процессе перемещения информации о программе/процессе во внешнюю память и обратно. Своппинг происходит сегментами.

# Сегментно-страничная организация памяти

Основные концепции:

Nseg | Npage | Offset

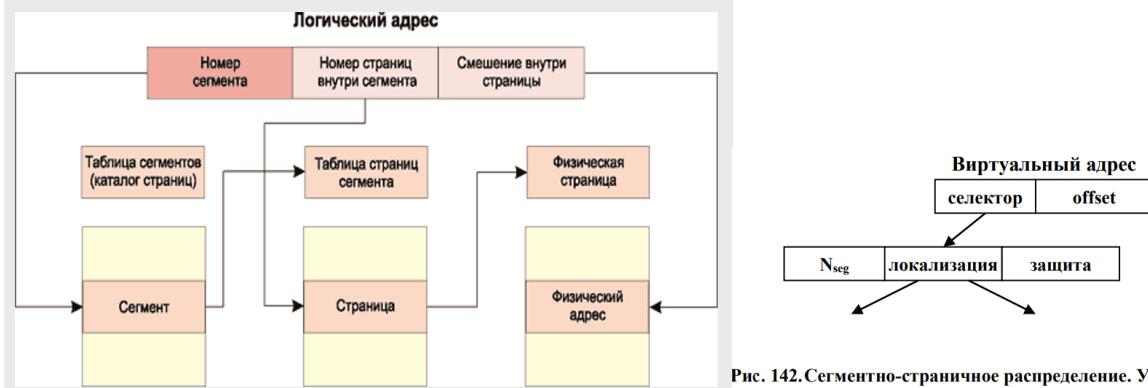


Рис. 142. Сегментно-страничное распределение. Упрощенная модель Intel.

## Сегментно-страничная организация.

Виртуальное адресное про-во процесса представляется в виде сегментов, где каждый из сегментов имеет страничную организацию. Виртуальный адрес состоит из

- Номер сегмента
- Номер страницы в сегменте- Смещение по странице.

Получая виртуальный исполнительный адрес сначала индексируемся по номеру сегмента, из номера сегмента получаем таблицу страниц сегмента, далее действуем как в страничной памяти этого сегмента.

- Преимущества сегментной организации памяти по сравнению со страничной?
    1. Простая аппаратная реализация (аппаратная таблица сегментов)
    2. Возможность адресной поддержки логической структуры виртуальной памяти процесса (кодовая часть, данные, стек, динамическая память и пр.)
  - Преимущества страничной организации памяти по сравнению с сегментной?
    1. Решение проблемы фрагментации памяти.
    2. Оптимальность использования физической памяти – возможность «откачки» значительной части неиспользуемых страниц.
- + гибкая и перенастраиваемая модель

---

## 52. Вычислительная система. Кэширование информационных потоков на уровнях аппаратуры и ОС.

---

## 53. Язык программирования С. Общая характеристика. Типы, данные, классы памяти. Правила видимости. Структура программы. Препроцессор. Интерфейс с ОС UNIX.

проблема несоответствия производительности центрального процессора и скорости доступа к информации, размещенной в оперативной памяти => кэш память (сократить обращения в ОП), расслоение памяти (+ кэш = запараллелить обращения), регистры общего назначения (сократить обращения в ОП)

проблем автоматизации обработки предопределённых событий, возникающих в вычислительной системе => аппарат прерываний.