Первое поколение компьютеров: середина 40-х — начало 50-х:

- электронно-вакуумных лампы
- ENIAC (46 США) – одна из первых ЭВМ. Производительность таких компьютеров измерялась от сотен до тысяч команд (операций) в секунду. Компьютер = процессор + ОЗУ + примитивных ВУ: вывода цифровой информации на бумажную ленту и ввода в ОП подготовленных данных и программ на специальных носителях (перфокартах, перфоленте и пр.) + ВЗУ — аппаратных средств хранения готовых к исполнению программы и данных (магнитные ленты).
- однопользовательский, персональный режим
- программа и необходимые данные представлены в машинных кодах в двоичном представлении
- пользователь (программист) использовал аппаратную консоль (или пульт управления) компьютера для ввода и запуска программы для чтения данных через устройства ввода. Результат выполнения программы выводился на устройство печати. В случае ошибки работа прерывалась, варнинг на индикаторах пульта управления

- программировать
- модифицировать и отлаживать программы
- кодировать все операции ввода/вывода с помощью специальных машинных команд управления ВУ

Сложно => пользователь должен был быть программистом, алгоритмистом и инженером-электронщиком.
Появляются:

- появляются ассемблеры – программы с определенными сервисными функциями программирования,
- первые языки высокого уровня
- трансляторы для этих языков
- простейшие средства организации и использования библиотек программ
- **управляющие программы**: чтения и загрузки в ОП программ и данных с ВУ (были определены фирмой-разработчиком и вводились в ОП с использованием аппаратной консоли). При помощи консоли было возможно вручную ввести в ОП последовательность команд (управляющую программу), запустить ее. Если управляющая программа размещалась на ВЗУ, через аппаратную консоль вводилась последовательность команд по чтению кода управляющей программы в ОП и управления передавалось на ее точку входа.

Компьютеры второго поколения: конец 50-х — вторая половина 60-х.

- диоды и транзисторы - полупроводниковые приборы, круче э-в ламп в десятки раз по функциональной емкости, размеру и энергопотреблению. В тысячи раз круче первого поколения по производительности.
- зародилось понятие ОП (**мониторные системами**, **супервизорами** или **диспетчерами)** в связи с появлением **пакетной обработки заданий**: специальная управляющая программа, которая последовательно загружала программы из заранее подготовленного пакета (стопки перфокарт) в ОП и запускала их (передавала управление на фиксированную точку входа в программу - адрес памяти, с которого должно начинаться выполнение программы). По завершении программы или при возникновении ошибки (аварийная остановка программы), управление возвращается управляющей программе.
- позднее в них реализовано **мультипрограммирование** – одновременно в обработке находится не одна, а несколько программ
- развитие «дружественности» интерфейсов между пользователем и системой: *система команд компьютера* ➔ *автокоды и ассемблеры* ➔ *языки программирования высокого уровня* ➔ *проблемно-ориентированные языки программирования*.
- **языки управления заданиями** позволяли пользователю до начала выполнения его программы сформировать набор требований по организации ее выполнения
- прообразы **ФС** — систем, которые позволяют систематизировать и упростить способы хранения и доступа к данным на ВЗУ (теперь пользователь может работать с данными во внешней памяти не в смысле физических устройств и координат данных на них, а в терминах имен или адресов)
- **виртуальные устройств**а - пользователь может забить на знание особенностей хранения и доступа к данным конкретных физических устройств
- в качестве элементной базы используются интегральные схемы:
    - увеличение производительности компьютеров
    - снижение их размеров, веса
    - появление новых, высокопроизводительных внешних устройств
- *начало* аппаратной унификации их узлов и устройств для возможной модифицируемости аппаратной комплектации

Компьютеры первых двух поколений - единые, аппаратно-целостные устройства, комплектация и возможности которых были существенно предопределены на этапе их производства.
- аппаратная модификация затруднительна

Третье поколение – модульная архитектура:
+ можно заменять и расширять состав ВУ

+ увеличивать размеры ОП
+ заменять процессор на более производительный
и модульные ОП с унификацией межмодульных интерфейсов.

- **драйверы устройств** – специальные программы управления устройствами со стандартными интерфейсами, позволяли работать с новыми или модифицированными устройствами при аппаратной модификации компьютера *драйверы виртуальных устройств предоставляли пользователю набор единых правил работы с группой ВУ. Это позволило создавать программы, не зависящие от типов используемых ВУ - простота и «дружественность» общения пользователя с различными устройствами
- новые режимы использования компьютеров (например, диалоговый).
- родилась Unix - стандартизация пользовательских интерфейсов

четвертое и последующие поколения:

- элементная база на больших интегральных схемах.
- компьютер из инструмента прикладного программиста стал повседневным, массово распространенным и доступным оборудованием => необходимо совершенствование «дружественности» пользовательских интерфейсов, упрощающих взаимодействие пользователя и ОС (Microsoft).
- Активное развитие получили сетевые технологии => появились сетевые и распределенные ОС
- всемирная сеть Internet => вопросы безопасности хранения и передачи данных

Основы компьютерной архитектуры

Середина 40-х годов 20-го века - начало ВС (отчет фон Неймана по компьютеру EDVAC): появление основополагающих принципов построения ВС – "принципов фон Неймана":

1. **Принцип двоичного кодирования информации**: вся информация, которая поступает и обрабатывается в компьютере, кодируется в двоичной системе счисления (нарушение – Сетунь Н.П.Брусенцова 3-ная система)

2. **Принцип программного управления (последовательной обработки)**: программа состоит из команд, в которых закодированы операции и операнды(аргументы) операции. Выполнение компьютером программы — это автоматическое выполнение определенной последовательности команд, составляющих программу. Устройство выполнения команд - процессор. Он последовательно обрабатывает команды (нарушение - большинство компьютеров начинает обрабатывать команды «с забеганием» вперёд, то есть во время выполнения текущей команды последующие команды уже начинают выбираться)

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. Одна и та же область памяти в зависимости от команд в одном случае будет интерпретироваться как команда, в другом случае – как данные (происходит неявно) (нарушение - в большинстве компьютеров ОЗУ хранит команды и данные «немного по-разному»)

Упрощенная структура компьютера фон Неймана:
ОЗУ + ВУ + ЦП (ЦП = **АЛУ + АУ**).

Оперативное запоминающее устройство (RAM — Random-Access Memory, память с произвольным доступом) – это устройство для хранения данных, в котором находится исполняемая программа. Команды программы, исполняемые компьютером, поступают в процессор исключительно из ОЗУ. ОП состоит из ячеек памяти. Ячейка памяти — это устройство, в котором размещается информация = поле машинного слова и поле служебной информации (ТЕГ).

1. **Машинное слово** — поле программно изменяемой информации. Тут могут располагаться машинные команды (или их части) или данные, с которыми может оперировать программа. Машинное слово имеет фиксированный для данной ЭВМ размер - количество двоичных разрядов. Пример: в 16-ти разрядном компьютере ОП имеет *машинные слова* размером 16 бит.
2. **тег** (tag — ярлык, бирка) - поле служебной информации для:

Контроль за целостностью данных (например одноразрядный ТЭГ чётности: при каждой записи в машинное слово происходит суммирование кода в машинном слове и формирование тега, при чтении - суммирование и сравнение с тегом, несовпадение - сбой в ОЗУ, данные потеряны – прерывание)
Контроль доступа к командам/данным. Чтобы не перепутать данные и команды «раскрашиваем» все машинные слова тегом
Контроль доступа к машинным типам данных. Существуют группы машинных команд, которые оперируют с данными одного типа. Соответственно, тег раскраски команд в зависимости от типа данных, с которыми они работают и самих данных. При выполнение чекается совпадение тегов (если не совпали – прерывание)

Типы данных (5): целые / вещественные с фиксированной точкой / вещественные с плавающей точкой / символьные / логические

Адрес ячейки памяти в ОЗУ – уникальное имя каждой ячейки. Обычно это порядковый номер ячейки (нумерация возможна как подряд идущими номерами, так и кратными некоторому значению). Доступ к содержимому машинного слова осуществляется при непосредственном или косвенном использовании адреса (например, в слове по адресу B хранится адрес A, значение которого надо прочесть).

Скорость доступа процессора к данным, размещенным в ОЗУ определяется производительностью:
1. **время доступа (access time)** — время между запросом на чтение слова из памяти и получением его содержимого.
2. **длительность цикла памяти (cycle time)** — это минимальное время между началом текущего и последующего обращения к памяти.

Проблемы:
1. $t_{cycle} > t_{access}$ (регегнерация: после операции чтения информация из ячейки разрушается и ее надо перезаписать для сохранения)
2. $t_{process} >> t_{access}$ (дисбаланс производительности аппаратных компонентов: скорость обработки данных в процессоре в несколько раз превышает скорость доступа к информации, размещенной в ОП)

Расслоение ОЗУ – всё пространство ОЗУ делится на **K** независимых подустройств - **банков памяти** размерами обычно **K** = 2^L. Адресация : младшие L разрядов - номер банка. Соседи любой ячейки размещаются в соседних банках. Зачем это нужно? При обращении к какой-то ячейке из одной общей кучи никакие другие обращения к памяти не могут производиться. При расслоении же можно производить несколько простых последовательных обращений к основной памяти одновременно, так как соседние ячейки располагаются в разных банках.
+ задержки, связанные с циклом памяти возникают только когда подряд идущие обращения попадают в один и тот же банк памяти => скорость чтения из памяти при последовательном доступе выше
в идеале при параллельной работе банков можно повысить производительность работы ОЗУ в **K** раз:
1.с централизованным контроллером доступа к памяти - один контроллер управляет всеми банками
+ в этом случае нет проблемы цикла памяти, т.к. соседние ячейки памяти находятся в разных банках
- нет эффекта при параллелизме
2.с контроллерами для каждого из банков
+ параллельный доступ к памяти (одновременно можно считать порцию данных до **K** слов)

Центральный процессор выполняет машинные команды, из которых состоит программа, размещенная в ОП. Термин «центральный процессор» соответствует ситуации сегодняшнего дня, когда современный компьютер – это многопроцессорная система (при этом компьютер будет называться однопроцессорным, поскольку в выполнении программы принимает участие только один процессор). Практически любой современный компьютер имеет в своем составе значительное количество специализированных управляющих компьютеров.

Регистровая память / **регистровый файл** (register file), — совокупность устройств памяти процессора (**регистров**) для временного хранения управляющей информации, операндов и/или результатов выполняемых команд:

Регистры общего назначения (РОН) доступны для программ пользователей и предназначены для хранения операндов, адресов операндов, результатов выполнения команд. РОН могут:

- иметь *машинную типизацию* (например, хранить данные с плавающей точкой, с фиксированной точкой)
- быть *скалярными* (с одним регистром ассоциируется только одна единица памяти)
- быть *векторными* (с одним регистром может ассоциироваться вектор регистров из 64 элементов)

+ скорость доступа к содержимому регистров сравнима со скоростью обработки информации процессором (сглаживание дисбаланса)
+ уменьшается количество обращений в ОП так как наиболее часто используемые в программе операнды размещются на РОН

Специальные регистры координируют информационное взаимодействие основных компонентов процессора:
Счетчик команд — в нем размещается адрес очередной выполняемой команды программы
Указатель стека — его содержимое в каждый момент времени указывает на адрес слова в области памяти, являющегося вершиной стека.
Слово состояния процессора — его содержимое определяет режимы работы процессора, значения кодов результата операций и т.п.

Устройство управления координирует выполнение команд программы процессором. **Арифметико-логическое устройство** выполняет команды, связанные с арифметической или логической обработкой операндов.

Выполнения процессором программы (рабочий цикл ЦП): в начальный момент времени в счетчике команд **СчК** находится адрес первой команды программы, а любая команда размещается в одном машинном слове и адреса соседних машинных слов отличаются на единицу.
1. по содержимому счетчика команд **СчК** из ОП выбирается команда для выполнения
2. формируется адрес следующей команды: **СчК = СчК + 1**.
3. анализируется код операции:

арифметическая или логическая
1. вычисляются исполнительные адреса операндов
2. выбираются значения операндов
3. команда передается для исполнения в АЛУ (передается код операции и значения операндов)
4. АЛУ выполняет команду и формирует код признака результата в регистре слова состояния процессора или в специальном регистре результата.

команда передачи управления:
анализируются условия перехода: если содержимое кода признака результата предыдущей арифм-лог команды совпадает с условиями перехода, соответствующими команде, вычисляется исполнительный адрес операнда **Аперехода**, и **СчК = Аперехода**

команда загрузки данных из памяти в РОН:
1. вычисляются исполнительные адреса операндов
2. выбираются значения операндов из памяти
3. значения записываются в соответствующие регистры.

КЭШ-память - размещение части данных в более высокоскоростном запоминающем устройстве, аппаратной «емкости», в которой аккумулируются наиболее часто используемые данные из ОП.

- 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

Это наиболее эффективное на сегодняшний день решение проблемы дисбаланса
+ варьируя размеры КЭШа, можно минимизировать частоту реальных обращений к ОП
+ скорость доступа к информации в КЭШе, соизмерима со скоростью обработки информации в ЦП

- 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.
Решение: два кэша - **КЭШ данных** и **КЭШ команд**
- усложнение логики процессора

Общая схема работы:

- вся память разделяется на блоки одинакового размера
- обмен данными между КЭШем и ОП осуществляется ***блоками*** фиксированного объёма (проявляются + использования памяти с расслоением, т.к. загрузка блока из ОП в КЭШ осуществляется с использованием параллелизма)
- Каждый блок имеет спецификатор доступа – ***тэг*** со служебной информацией:
  1. какой области ОП соответствует содержимое данного блока
  2. занят он или свободен
  3. производились ли в нем изменения

Когда процессор обращается за командой или за данными в ОП, сначала он обращается к КЭШу
1. По содержимому адресного тэга можно однозначно адресовать содержимое блока (анализ тэгов блоков КЭШа производится аппаратно)
2. По исполнительному адресу устройство управления определяет, находится ли соответствующая информация в одном из блоков КЭШ-памяти или нет:
   попадание (hit) и данные берутся из КЭШа, обращение в ОП не осуществляется
   промах (cache miss) и ***вытеснение*** – обновление содержимого КЭШа - выбирается блок-претендент на вытеснение, содержимое которого будет заменено: *случайным образом* или *вытеснение наименее «популярного» блока КЭШа*.

вытеснения блока в КЭШе данных - содержимое блоков КЭШа может не соответствовать содержимому памяти (это возникает при обработке команд записи данных в память):
- **сквозное кэширование**: 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
- **кэширование с обратной связью**: специальный *тег модификации* (dirty bit). При выполнении команды записи данных обновление происходит только в КЭШе, но устанавливается тег модификации. При вытеснении блока: если тег модификации установлен, то содержимое блока перед вытеснением «сбрасывается» в память.

Аппарат прерываний. В первых компьютерах в случаях ошибки работы всего компьютера прекращалась, чего не могут позволить себе современный ВС.
Прерыванием - событие, при возникновении которого в системе предусмотрена предопределенная последовательность действий = стандартная реакция процессора на прерывание + его программная обработка
Состав прерываний 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 или обращение к несуществующей области памяти)
2. Внешние прерывания — события, возникающие в компьютере в результате взаимодействия ЦП с ВУ (ввод символа завершения работы с клавиатуры)

Обработка:
аппаратная:
1. текущая **команда** завершается (кроме случаев, когда прерывание возникает по причине некорректного выполнения команды)
2. текущая **программа** останавливается
3. фиксируется актуальное состояние программы (сохранение регистров: счетчик команд, регистр результатов, регистры, содержащие режимы работы процессора, а так же некоторые РОН).
*буфер для сохранения актуального состояния в системе один, поэтому, если пришло прерывание, то другие прерывания нужно запретить => **включается режим блокировки прерываний** (либо игнорируются, либо откладываются)
3. управление передается специальной программе обработки прерывания. Способы передачи аппаратурой ОС информации о типе прерывания:

- **регистр прерываний**, каждый разряд которого соответствует конкретному типу прерывания. Для расширения числа обрабатываемых прерываний возможно использование иерархической модели регистров прерывания. Она предполагает, что имеется **главный регистр прерывания** и **периферийные**. В главном регистре прерывания выделяются разряды, отвечающие не только за появление конкретных прерываний, но и разряды, отвечающие за появление прерываний в периферийных регистрах. В данной модели управление ОС передается по адресу входа в программу.
- **вектор прерываний**: по количеству возможных прерываний в ОЗУ выделена группа машинных слов — *вектор прерываний*. Каждое слово вектора прерываний содержит адрес программы, обрабатывающей данное прерывание. При возникновении прерывания после сохранения регистров осуществляется передача прерывания по адресу, соответствующему номеру прерывания.
- **регистр слова состояния процессора**. В этом случае в данном регистре резервируется часть разрядов – поле, в которое передаётся номер возникшего прерывания. В этой модели управление передается на фиксированный адрес входа в программу обработки прерываний.

программная: в связи с упрятыванием регистров, часть ресурсов ЦП оказывается освобождена => программа обработки может использовать только освобожденные ресурсы.

1. анализ и предварительная обработка прерывания:
   - «**короткое**» прерывание - обработка не требует дополнительных ресурсов ЦП и времени (например прерывание от таймера для коррекции времени в системе)
     1. прерывание обрабатывается,
     2. выключается режим блокировки прерываний,
     3. восстанавливается состояние процессора, соответствующее точке прерывания исходной программы, и передается управление на прерванную точку.
   - «**фатальное**» прерывание - продолжить выполнение программы невозможно (например, 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.
   - «**полное упрятывание**» - если прерывание не короткое и не фатальное (например, обращение к области памяти, которая закрыта для обращения - чтение информации с внешнего носителя)
     1. все ресурсы цп (содержимое регистровой/кэш памяти и все остальное), использовавшиеся данной программой, сохраняются/копируются в специальную программную таблицу
     2. программе обработки прерываний становятся доступны все ресурсы ЦП
     3. прерванная программа получает статус ожидания завершения обработки прерывания (таким программ может быть несколько)

2. снятие режима блокировки прерываний (включается стандартный режим работы процессора, при котором возможно появление прерываний)
3. ОС завершает обработку прерывания и, возможно, возвращается в исходную программу

Внешние устройства

Внешние запоминающие устройства (ВЗУ) предназначены для хранения данных и программ. Обычно обмен с ВЗУ происходят некоторыми порциями данных, которые называются ***записями***. Данные, размещенные на ВЗУ, представляются в виде последовательности записей:

по размеру обмениваемых данных:

- **блочные устройства** - фиксированный размер — ***блоки*** (размер блоков (**физических блоков**) определяется аппаратно) – магнитные диски
- **устройства с произвольным** размера - магнитные ленты

по типу работы с данными:

- допускающие как операции чтения, так и операции записи (жесткий диск)
- позволяющие выполнять только операции чтения (CD-ROM, DVD-ROM)

по типу доступа к данным:

- **последовательный** - при доступе к содержимому произвольной записи «просматриваются» все записи, предшествующие искомой

магнитная лента:
** каждая лента имеет специальные маркеры начала и конца ленты.*
** каждая запись имеет специальные маркеры начала и конца записи и свой логический номер. При запросе 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.

- **прямой** - обеспечивает чтение/запись без считывания дополнительной (предыдущей) информации

Магнитный диск - самое распространенное ВЗУ:
Обмен данными в одном из секторов:
Для задания координат конкретного сектора нужно передать в устройство управления магнитным диском:
– номер цилиндра данного сектора N_C
– номер дорожки сектора N_t
– номер сектора N_S
После получения координат сектора (N_C, N_t, N_S)
– двигатель перемещает блок головок в цилиндр N_C
– включается головка чтения/записи, соответствующая номеру дорожки N_t
– как только головка чтения/записи позиционируется над началом искомого сектора N_S, запускается выполнение операции чтения (или записи)

Производительность внешнего запоминающего устройства — время доступа к хранящейся информации — определяется наличием и продолжительностью механических операций, которые необходимо провести при обмене. Так, время обмена с магнитным диском = время выдвижения блока головок в соответствующий цилиндр (не больше времени перемещения блока головок из начального положения к цилиндру с максимальным номером) + время установки головки в начало сектора, с которым будет осуществляться обмен (не больше времени полного оборота вала).

магнитный барабан:
Для адресации блока данных в этом случае используется только номер дорожки (Nтрека) и номер сектора (Nсектора). Для чтения или записи:
- устройство управления должно включить головку, соответствующую указанному номеру дорожки;
- а после этого происходит ожидание механического поворота цилиндра до выхода головки на начало искомого сектора.
Таким образом, по сравнению с жесткими дисками, в этом устройстве отсутствует механическая составляющая выхода головки на нужный трек, поэтому данный тип устройств считается более высокоскоростным.

память на магнитных носителях (доменах) – магнитноэлектронное ВЗУ прямого доступа. ***Домен*** **-** некоторая элементарная единица, способная сохранять свою намагниченность в течение длительного промежутка времени. Домен может быть намагничен одним из двух способов (либо как «плюс-минус», либо как «минус-плюс»). Под воздействием магнитно-электронных эффектов магнитные домены

разгоняются вдоль своего трека до некоторой постоянной скорости. В остальном же принцип работы данного класса устройств ничем не отличается от работы магнитных барабанов. Соответственно, из-за того, что в данном устройстве нет механической составляющей, оно является еще более высокоскоростным по сравнению с предыдущими устройствами. Для считывания или записи информации на данный носитель устройство управления включает необходимую головку, которая по таймеру синхронизируется с «приходом» начала искомого сектора, после чего происходит обмен с найденным сектором.

Модели синхронизации при обмене с внешними устройствами

1. Синхронная работа с ВУ: в момент обращения к ВУ программа будет приостановлена до момента завершения обмена
- задержки снижают эффективность ВС

2. Асинхронная работа с ВУ:

1. Пусть в системе прерываний есть специальное внутреннее прерывание по команде. Программа инициирует это прерывание и передает заказ на выполнение обмена.
2. В ОС происходит обработка прерывания, при этом конкретному драйверу устройства передается заказ на выполнение обмена.
3. После завершения обработки прерывания программа может продолжить свое выполнение, или может быть запущено выполнение другой программы.
4. По завершении обмена происходит прерывание, после обработки которого программа, выполнявшая обмен, может продолжить свое выполнение.

+ сглаживается дисбаланс в скорости выполнения машинных команд и скоростью доступа к ВУ
- программа может попытаться обновить содержимое на ВЗУ после заказа на обмен, но до его завершения, что является некорректным (аппаратное закрытие ВЗУ на чтение и/или запись)

Потоки данных. Организация управления ВУ.

Есть два потока информации - управляющая информация и данные.
*если рассматривать потоки в контексте ВЗУ, то можно выделить также поток управляющей информации с командами управления ВУ, а также поток данных, перемещающихся между ВЗУ и ОП. Рассмотрим теперь различные модели организации управления ВЗУ.

Модели организации управления ВЗУ:

1. ЦП управляет ВУ **непосредственно:** и **управляет** (спецкоманды перемещения головок, включения той или иной головки, ожидания прихода содержательной информации) и **выполняет обмен** информацией (интерпретирует последовательность команд управления - считывает информацию, участвующую в обмене, со специальных регистров и переносит ее в ОП или же производит обратные манипуляции).
- оба потока проходят через ЦП - трудоемкая задача
- подразумевает лишь синхронную реализацию

2. синхронное управление ВУ с использованием контроллеров ВУ. ***Контроллер*** - электронная схема управления данным устройством. Она берет на себя часть работ по управлению или обмену данными (например, локализовать и исправить возможные ошибки, которые могут случиться при чтении или записи данных)
- исторически такой тип управления ВЗУ изначально был синхронным: процессор посылает устройству команды на обмен и ожидает, когда этот обмен завершится
- поток данных по-прежнему проходит через ЦП - он считывает их со специальных регистров ВУ и помещает в ОП

3. **асинхронное управление** с использованием контроллеров ВЗУ. В этом случае ЦП подает команду на обмен и не дожидается, когда эту команду отработают контроллер и устройство (может продолжить обработку каких-то задач). Но для осуществления указанной модели необходимо, чтобы в системе был реализован аппарат прерываний.

4. **контроллеры прямого доступа к памяти** DMA. Они исключили ЦП из обработки потока данных, взяв эту функцию на себя. ЦП занимается лишь потоком управляющей информации.

5. Использование **процессора** или **канала ввода-вывода** - специализированный компьютер со своим процессорным элементом, своей ОП, под управлением своей ОС, располагается логически между ЦП и ВУ. Он осуществляет высокоуровневое управление ВУ. Тогда ЦП оперирует с ВУ в форме высокоуровневых заказов на обмен. Соответственно, 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.).

Иерархия памяти

1. Память, которая размещается в ЦП (это **регистровая память** и **КЭШ первого уровня** (**L1**)) - самая высокопроизводительная и дорогостоящая память.

2. **КЭШ второго уровня** (**L2**) логически располагается между ЦП и ОП – среднее по производительности и стоимости.
3. **ОЗУ или ОП** - в ней располагается исполняемая ЦП программа (именно оттуда процессор «берет» очередные операнды и команды для исполнения)
4. устройства, предназначенные для оперативного хранения программной информации пользователей и ОС - **ВЗУ прямого доступа:**

1. с внутренней КЭШ-буферизацией для наиболее оперативного обмена (на них ОС может размещать свои всякого рода информационные таблицы)

2. **без КЭШ-буферизации**, которые также обеспечивают оперативных доступ, но уже на более низких скоростях. На подобных устройствах может находиться 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).
5. **ВЗУ долговременного хранения данных**. Это системы резервирования, системы архивирования и т.д.
- низкая скорость доступа к данным
+ низкая стоимость хранения единицы информации.

Аппаратная поддержка ОС и систем программирования

Цель производителей первых компьютеров - создание автоматических вычислителей, однако со временем появилась необходимость управлять этими вычислениями:
Первый шаг – это появление программируемого компьютера, т.е. устройства, которое умеет исполнять программу.
Второй шаг – это появление регистров общего назначения, которые позволили на программном уровне сгладить разницу производительности центрального процессора и оперативной памяти.
Третий шаг – это появление аппарата прерываний, который позволил организовать асинхронную работу с внешними устройствами и т.д.

Требования к аппаратуре для поддержки мультипрограммного режима

Мультипрограммный режим – в каждый момент времени в системе могут обрабатываться две и более программ пользователей (эти программы иногда называют *смесью программ*), режим наиболее эффективной загрузки ЦП). Каждая из этих программ может находиться в одном из *трех состояний*:

1. выполняться на процессоре (т.е. ее команды исполняются центральным процессором),
2. ожидать завершения запрошенного ею обмена (для продолжения ее выполнения необходимо окончания обмена)
3. ожидать освобождения центрального процессора (эти программы готовы к выполнению на процессоре, но процессор в данный момент занят иной программой).

схема организации мультипрограммного режима: в начальный момент времени на процессоре обрабатывается *Программа 1*, которая в некоторый момент времени *t1* выдает запрос на обмен, при этом дальнейшая обработка на процессоре невозможна до завершения этого обмена. В случае синхронной организации *Программа 1* будет приостановлена, и процессор будет простаивать до завершения обмена *Программы 1*. Вместо этого пока *Программа 1* ожидает завершения своего обмена, будет запущена, например, *Программа 2*, которая выполняется до некоторого момента времени *t2*, после чего она приостанавливается по тем или иным причинам, и запускается *Программа 3*. После завершения обмена на обработку вновь ставится *Программа 1*, сменяя *Программу 3* в момент времени *t3*.

Под **корректным** функционированием мы будем понимать, что в независимости от степени мультипрограммирования (от количества обрабатываемых в системе программ) результат работы конкретной программы не зависит от наличия и деятельности других программ.

Проблемы мультипрограммного режима (в первую очередь, связанные с конкуренцией между обрабатываемыми программами за доступ к одним и тем же ресурсам ВС).

1. **влияние программ друг на друга**: одна программа может обратиться в адресное пространство другой программы и считать оттуда или записать туда данные => нужен ***аппарат защиты памяти*** обрабатываемых программ:
в процессоре могут быть специальные регистры - ***регистры границ*** диапазона доступных адресов ОП для исполняемой задачи. Соответственно, когда УУ в ЦП вычисляет очередной исполнительный адрес (адрес следующей команды или адрес необходимого операнда), **автоматически** проверяется, принадлежит ли полученный адрес заданному диапазону: да - продолжается обработка задачи, нет - в системе возникает ***прерывание по защите памяти***.

Пример1: если в мульти-системе имеется единственный принтер и несколько программ, которые выводят свои данные на печать (юзают специальные машинные команды управления принтером), то при совместной работе в режиме мультипрограммирования на выводе будет каша.
Пример2: Значения регистров границ аппарата защиты памяти устанавливаются посредством специальных машинных команд. Если же к этим командам смогут обращаться произвольные программы, то они смогут обойти этот режим защиты подменой своих регистров границ.

2. **неограниченный доступ к машинным командам**: система должна каким-то способом ранжировать команды и в соответствии с ранжированием ограничивать доступ пользователей различных категорий к машинным командам.
Решением стала **аппаратная** возможность работы ЦП в двух режимах:

- ***режиме работы ОП*** (или ***привилегированном режиме***, или ***режиме супервизора***), когда процессор может исполнить любую из своих машинных команд
- ***пользовательском режиме*** (или ***непривилегированном режиме***) - программе доступно для исполнения лишь некоторое подмножество машинных команд (иначе прерывание по запрещенной команде). Тогда программа, работающая в непривилегированном режиме, может командами обратиться к ОС, а через параметры передать необходимые данные. Тогда ОС для каждой из программы формирует некоторую таблицу или область памяти, в которой будет аккумулироваться информация из данных (на печать например). Реально печать будет выполнена, если:
    1. программа, посылающая данные, успешно завершилась и гарантированно не будет больше посылать данные на печать.
    2. в программе обнаружилась фатальная ошибка, что ведет к безусловному завершению этой программы, что опять-таки гарантирует отсутствие будущих запросов на печать.
    3. ОС система может получить (от некоторого виртуального оператора – т.н. *планировщика*) команду разгрузить буфер печати данной конкретной программы.

3. **программа зациклилась**: необходима функция управления временем (ОС должна контролировать время использования ЦП программами пользователей - требуется **прерывание по таймеру**: зацикленная программа автоматически будет периодически прерываться, управление периодически будет передаваться ОС и та уже решать – поставить на счет другую программу / снять со счета (например, по команде пользователя) эту зависшую программу).
Существуют две стратегии работы со временем:

1. любой процесс берёт время ЦП до тех пор, пока его не убьют. Тем не менее, при этой стратегии зацикленная программа не будет никому мешать, т.к. ОС будет периодически получать управление и переключать процессы. (используется в Windows, в «ширпотребовской» Unix)
2. процесс предварительно оговаривает, сколько времени ему может потребоваться. Если процесс пытается превысить это время, то ОС считает, что процесс зациклился. (применяется для высокопроизводительных машин)

Включение режима супервизора зависит от архитектуры конкретной системы: в некоторых архитектурах считается, что ОС занимает некоторое предопределенное адресное пространство физической памяти, и если управление попадает на эту область, то включается режим ОС. А вот выключение режима ОС может происходить программно: например, ОС, запуская процесс, может предварительно программным способом установить его в непривилегированный режим.

Проблемы, возникающие при исполнении программ

1. Вложенные обращения к подпрограммам: при выполнении математических вычислений порядка 70% времени тратится на обработку входов и выходов из подпрограмм, когда необходимо зафиксировать адрес возврата, сформировать параметры, передаваемые подпрограмме, сохранить регистровый контекст.
Решение: использование в современных процессорах ***регистровых окон*** для сохранения/ восстановления регистров. Пусть в процессоре имеется K физических РОН для использования в пользовательских программах. В каждый момент времени программе доступно *регистровое окно*, состоящее из L регистров ($L<K$). Нулевое окно: регистры от 0 до $L–1$. Первое окно: L регистров от $L–1$ (нумерация внутри окна от 0 до $L–1$). При обращении из текущей программы в другую программу автоматически происходит смена окна на следующее. Структура регистрового окна:

1 группа – регистры формальных параметров (включая адреса возврата)
2 группа – регистры локальных параметров
3 группа – регистры фактических параметров.

- каждый из регистров окна отображается на один из регистров базового регистрового файла.
- следующее регистровое окно, которое получит программа при обращении к подпрограмме, связано с текущим окном пересечением началом и концом регистровых окон. Т.е подпрограмма в своём окне получит фактические параметры программы через соответствующие формальные параметры (локальные регистры текущей подпрограммы сохранять не нужно)
- все окна расположены в циклическом списке: нулевое окно пересекается с первым, первое — со вторым… N–1-е с нулевым.
- имеется команда смены окна - при обращении к подпрограмме через пересекающиеся точки передаются адреса возвратов, а

внутри окна можно работать с регистрами, причем при обращении к подпрограмме не встает необходимость их сохранения.

- имеются два управляющих регистра: указатель текущего окна (**CWP)** и указатель сохранённого окна **(SWP)**. Если глубина вложенности больше, чем количество регистровых окон, то возникает проблема. В этом случае какое-то окно необходимо сохранить в ОП или стеке, чтобы потом его использовать. При этом эффективность, естественно, начнёт падать. Считается, что наиболее оптимальный эффект оптимизации достигается при четырех окнах – это означает, что средний уровень вложенности подпрограмм не более четырех.
  - фиксированный размер каждого окна
  - неоптимально (т.к. иногда требуется больше регистров, иногда — меньше)
- **Модель организации регистровой памяти в Intel Itanium**. В современных компьютерах имеется возможность варьирования размера регистрового окна. В частности, в 64-разрядных процессорах Itanium компании Intel размер окна *динамический*. В данном процессоре в регистровом файле, состоящем из 128 регистров, первые 32 регистра (с номерами от 0 до 31) являются общими, а на регистрах с номерами от 32 по 127 организуются регистровые окна, причем окно может быть произвольного размера.

2. Накладные расходы при смене обрабатываемой программы. *ОС* должна сохранить контексты процессов. А одновременных процессов в современных компьютерах много.
Решение: аппаратная поддержка стека - имеется регистр, который ссылается на вершину стека, и есть некоторый механизм, который поддерживает работу со стеком. Если в системе возникает прерывание, процессор просто сохраняет в стеке содержимое необходимых регистров («малое упрятывание»). Если же возникнет второе прерывание, то процессор поверх предыдущих данных скинет в стек новое содержимое регистров, чтобы обработать вновь пришедшее прерывание. Но так как стек располагается в ОП, то при каждой обработке прерывания процессору придется обращаться к ней, что сильно снижает производительность системы при частых возникновениях прерываний. Решения:

1. В процессоре могут использоваться специальные регистры, исполняющие роль буфера, аккумулирующего вершину стека непосредственно в процессоре.
2. Работу со стеком можно организовать посредством буферизации в КЭШе первого уровня (L1), но при кэшировании стека мы добавляем ещё один поток информации.

3. Перемещаемость программы по ОЗУ. Компилятор из исходного текста программы (кода) образует **объектный модуль** - файл с промежуточным представлением отдельного модуля программы. Объектный файл содержит в себе особым образом подготовленный код (часто называемый *двоичным* или *бинарным*), который может быть объединён с другими объектными файлами при помощи редактора связей (**компоновщика**) для получения готового **исполнимого модуля** либо библиотеки. Связывание со статическими библиотеками выполняется так же компоновщиком, а с ОС и динамическими библиотеками связывание выполняется при исполнении программы, после её загрузки в ОП. Возникает проблема привязки исполняемого модуля к тому адресному пространству, в котором он будет исполняться. Исторически исполняемые модули настраивались на те адреса ОП, где они должны были исполняться (если память в данный момент занята другой программой, то эту программу поставить на счет не удастся). Возникает проблема перемещаемости программы по ОЗУ: ресурс свободной памяти в ОЗУ может быть достаточно большим, чтобы в ней разместилась вновь запускаемая программа, но в силу фиксированности адресов исполнения ее запустить не удается.

Решение: аппарат **виртуальной памяти** - аппаратное средство процессора, которое устанавливает соответствия виртуальных адресов, используемых внутри программы, и физических адресов ОП, в которой размещается программа во время выполнения. **Виртуальное адресное пространство** - адресное пространство, которое используется внутри программ. Виртуальные адреса существуют «вне машины». Аппарат **базирования адресов**: получаемый в ходе выполнения программы исполнительный адрес ($А_{исп.}^{вирт.}$) можно интерпретировать как **абсолютный исполнительный адрес**, когда физический адрес в некотором смысле соответствует исполнительному адресу программы ($А_{исп.}^{физ.} = А_{исп.}^{вирт.}$) или как **относительный адрес (**адрес относительно точки загрузки программы). Иными словами, имеется ОП с ячейками от 0 до некоторого L–1, и, начиная с некоторого адреса *K*, расположена программа. Тогда адрес $А_{исп.}^{вирт.}$ - отступ от физической ячейки с адресом K на величину $А_{исп.}^{прог.}$. Для реализации модели базирования используется специальный ***регистр базы***, в который в момент загрузки процесса в ОП ОС записывает начальный адрес загрузки (K). Тогда реальный физический адрес получается, исходя из формулы $А_{исп.}^{физ.} = А_{исп.}^{прог.} +$ <Rбазы>.

\+ решает проблему перемещаемости программ по ОЗУ, поскольку процесс можно загрузить в любую область памяти.

4. Фрагментация памяти: исполняемый модуль нужно грузить в свободный и непрерывный фрагмент памяти, которого может не быть, не смотря на то что в сумме свободного места достаточно. ОС способна оценивать свободное пространство ОП и из буфера программ, готовых к исполнению, выбирать те, что влезут в свободный фрагмент памяти. Но зачастую их размер даже меньше свободного места, т.е. формируются «излишки» - **фрагментации ОП** и система начинает деградировать.

Решение: аппарат **страничной организации памяти**. Все адресное пространство ОП - последовательность блоков фиксированного размера (кратно степени 2, пусть = 2^k), называемых страницами. Тогда структура виртуального адреса: правые *k* разрядов представляют адрес внутри страницы, а оставшиеся разряды отвечают за номер страницы.

Виртуальное адресное пространство — это множество виртуальных страниц, доступных для использования в программе.
Физическое адресное пространство — это ОП, подключенная к данному компьютеру. Физическая память может иметь произвольный размер по отношению к размеру виртуальной памяти (физических страниц >= виртуальных страниц).

В ЦП имеется аппаратная (регистровая) таблица - **таблица страниц** (строк = максимальное число виртуальных страниц)

- каждой виртуальной странице исполняемой программы соответствует строка таблицы с тем же номером
- внутри каждой записи таблицы страниц находится номер физической страницы, в которой размещается соответствующая виртуальная страница программы

Все что нужно – корректно создать и заполнить для процесса, который ставится на обработку в ЦП, таблицу страниц.

Схема преобразования адресов: Пусть в таблице страниц имеется N строк. Содержимое каждой i-ой строки таблицы — α_i – определяется ОС в момент запуска процесса. Если $\alpha_i >= 0$, то это номер физической страницы, которая соответствует i-ой виртуальной странице. Если $\alpha_i < 0$, то это означает, что данной страницы у программы нет (при попытке обратиться - прерывание по защите памяти, управление получает ОС).
Причины возникновения прерывания:

1. действительно i-ой виртуальной страницы у программы нет, что свидетельствует об ошибке в программе
2. соответствующей страницы нет в ОП – она расположена на ВЗУ (данная i-ая виртуальная страница легальна, но в данный момент ее нет в ОЗУ. Тогда ОС подкачивает из ВЗУ нужную страницу в ОЗУ)

Достоинства:

1. **защита памяти** - процесс никогда не сможет обратиться к «чужой» странице, но также имеется возможность разделять некоторые страницы между несколькими процессами (в этом случае ОС каждому из процессов допишет в таблицу страниц номер общей страницы)
2. **высокая производительность** - используются регистры
3. **нет фрагментации** - все программы оперируют в терминах страниц (каждая из которых имеет фиксированный размер), и мы можем размещать программу по страницам в произвольном порядке
4. **перемещаемость по ОЗУ** - даже в рамках одной программы соответствие между виртуальными и физическими страницами может оказаться произвольным: нулевая вс может располагаться в одной физической странице, первая - в другой (совершенно не связанной с первой) физической странице
5. **нет необходимости держать в ОП весь исполняемый процесс** - повысить 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 => размеры физической и виртуальной памяти могут быть произвольными. Может оказаться, что физической памяти в компьютере больше, чем размеры адресного пространства виртуальной памяти, а может оказаться и наоборот: физической памяти существенно меньше виртуальной. Но во всех этих случаях система окажется работоспособной.

Недостатки:

1. **Внутристраничная или внутренняя фрагментация**: если в странице используется хотя бы один байт, то вся страница отводится процессу и считается занятой
2. если таблица страниц целиком располагается на регистровой памяти, то в силу дороговизны последней **размеры подобной таблицы должны быть слишком малы** (а следовательно, будет невелико количество физических страниц)
3. при смене процессов таблицу страниц сначала обязательно надо сохранить (скопировать в программную таблицу), а после этого восстановить содержимое таблицы страниц той программы, которая будет исполняться следующей.– дополнительные **накладные расходы**

На размер виртуального адресного пространства влияет разрядность исполнительных адресов, получаемых в ходе обработки программы на ЦП. Размер физического пространства определяется характеристикой компьютера: зависит от того, сколько физически можно подключить памяти к машине, и какова разрядность внутренней аппаратной шины. Но и то, и другое являются аппаратными характеристиками компьютера.

Многомашинные, многопроцессорные ассоциации

Говоря об ЭВМ, мы подразумеваем машину в некотором окружении и взаимодействии с другими машинами. В зависимости от степени интегрированности машин в рамках одного комплекса различают:

1. **многопроцессорные** ассоциации, где степень связанности машин довольно велика
2. **многомашинные** ассоциации, в которых наблюдаются слабые связи между машинами

Возможны некоторые оптимизации потоков (команд и данных): В потоке команд — это 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); в потоке данных — это исключение участия ЦП в обменах между ВУ и ОП.

Классы многопроцессорных архитектур по Флинну:

ОКОД (одиночный поток команд, одиночный поток данных, или **SISD** — single instruction, single data stream) -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. В некотором смысле сюда можно отнести специализированные системы обработки видео- и аудиоинформации (DSP-процессоры), а также конвейерные системы.

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.

Среди систем МКМД можно выделить два подкласса:

1. **системы с общей ОП:** любой процессор имеет **непосредственный** доступ к любой ячейке этой общей ОП.
2. **системы с распределенной памятью** - объединение компьютерных узлов. Под узлом понимается самостоятельный процессор со своей локальной ОП. В данных системах любой процессор **не может** произвольно обращаться к памяти другого процессора.

*на практике обычно встречаются промежуточные решения.

Системы с общей ОП:

UMA(uniform memory access) — система с однородным доступом к памяти.
Подвид UMA-систем - **SMP** (symmetric multiprocessor — симметричная мультипроцессорная система) - к общей системной шине, или магистрали, подсоединяются несколько процессоров и блок общей ОП.

Недостатки:

1. Количество подключаемых процессорных элементов ограничено «шириной» общей шины (обычно 2, 4, 8, вплоть до 32).
2. Возникают дополнительные проблемы синхронизации КЭШей первого уровня каждого процессора:

Решения:

1. не использовать КЭШ
2. реализовать КЭШ-память со слежением: каждый КЭШ слушает шину и реагирует на ситуацию в системе.
   a. Промах при чтении – запись соответствующего блока в локальный КЭШ через общую магистраль, которую слушают другие КЭШи; никаких действий КЭШи других процессоров не производят.
   b. Попадание при чтении - мы используем КЭШ, и эта операция не даёт никакой нагрузки на общую шину.
   c. Промах при записи - осуществляется запись в память; все другие КЭШи слушают информацию о записи и проверяют наличие соответствующего блока у себя – если блок есть, то они удаляют его из своих КЭШей (обновление памяти)
   d. Попадание при записи ситуация аналогичная, только запись осуществляется ещё и в локальный КЭШ.

NUMA (non-uniform memory access) — система с неоднородным доступом к памяти. Степень параллелизма в NUMA-системах выше, чем в SMP.

- процессорные элементы работают на общем адресном пространстве;
- характеристики доступа процессора к области оперативной памяти зависят от того, 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.

Недостатки:
- синхронизации КЭШа. Решение:
1. не использовать кэш
2. использовать системы с когерентными КЭШами –
можно отслеживать соответствие локальных КЭШей друг
другу и состояние всей системы в целом.
Вытекающая проблема: размер шины и архитектура ограничивает число подключаемых процессоров (до сотни) появляются системные потоки служебной информации, что ведет к дополнительным накладным расходам – загрузке общей шины служебной информацией.

системы с распределенной оперативной памятью - объединение вычислительных узлов, каждый из которых состоит из процессора и ОЗУ, непосредственный доступ к которой имеет только «свой» процессорный элемент.
Основные классы:
1. **MPP** (Massively Parallel Processors — процессоры с массовым параллелизмом)
   + высокая эффективность при решении определённого класса задач
   - дорогостоящие и узкоспециализированные => нет массового применения

2. **COW** (Cluster of Workstations — кластеры рабочих станций) - многомашинные системы, состоящие из множества узлов.

   - обыкновенный компьютер может быть узлом (компьютерные сети)
   - минимальным узлом может быть процессор со своей локальной ОП и аппаратурой сопряжения с другими вычислительными узлами.
   - для сопряжения вычислительных узлов в кластере используются специализированные компьютерные сети
   - в кластере может быть до нескольких тысяч узлов

   Цели кластера:
   1. **надежность**: решение 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; Для построения используют Windows-системы.
   2. **высокопроизводительная ВС** (критерий эффективности - скорость обработки информации). Каждый вычислительный узел многопроцессорной системы– это компьютер традиционной архитектуры. Связь узлов - специальные высокоскоростные сети. Для построения вычислительных кластеров зачастую используют Unix-системы.

   + высокая эффективность при решении широкого круга задач (за приемлемую цену)
   + унифицированные средства программирования (типа MPI).
   - отвод тепла
   - коммуникация (если будет использоваться единственная магистраль, то она «захлебнется» от потоков передаваемой информации, а большинство узлов будут простаивать).

1.2.8 Терминальные комплексы (ТК)

Современные многопроцессорные системы строятся как специализированные компьютерные сети.
Терминальный комплекс — это многомашинная ассоциация, которая предназначена для организации массового доступа пользователей (удаленных и локальных) к ресурсам некоторой ВС.
Компоненты ТК:
– основная ВС, к которой обеспечивается доступ;
– локальные мультиплексоры;
– локальные терминалы;
– модемы;
– удаленные терминалы;
– удаленные мультиплексоры.

Считается, что пользователь может использовать:

терминальное устройство (алфавитно-цифровой терминал, т.е. дисплей с клавиатурой):

1. **локальные** – подключение:
   a. непосредственно к ВС по локальному каналу связи
   b. через **мультиплексор** - устройство, позволяющее через один высокоскоростной канал связи подключить N менее скоростных каналов
2. **удаленные** - подключаются к ВС через коммуникационную среду (раньше - обычные телефонные сети, которые исторически основывались на аналоговом способе передачи информации; компьютерные сети основаны на цифровом (дискретном) способе передачи данных. Для передачи цифровой информации через аналоговые сети необходимы аналогово-цифровой и цифро-аналоговый преобразователи (модем — **мо**дулятор-**дем**одулятор).

- печатающее устройство
- внешнее запоминающее устройство (например, магнитная лента).

линии связи/каналы. Классификация по…

по организации:

Коммутируемый канал — это линия, выделяемая на весь сеанс работы терминального устройства (телефонный разговор).

Выделенный канал обеспечивает связь терминального устройства с ВС на постоянной основе:
1. можно протянуть между терминальным устройством и ВС физический провод.
- дорого
+ наилучший способ обеспечить выделенный канал
2. можно взять один из коммутируемых маршрутов телефонной сети
+ постоянная готовность
+ детерминированное качество соединения.
+ дешевле предыдущего
- уменьшает количество коммутируемых маршрутов в телефонной сети (негативно сказывается на качестве ее работы «по прямому назначению» — на обеспечении телефонной (голосовой) связи).

по **количеству участников общения:**

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

по движению информации в канале:

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

Компьютерные сети - терминальные комплексы с компьютерами вместо терминальных устройств. **Компьютерная сеть** - это объединение компьютеров (или ВС), взаимодействующих через **коммуникационную среду** - каналы и средства передачи данных.
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 и **автономными**, если один компьютер не может управлять работой другого
2. сеть предполагает возможность **распределенной обработки информации**
3. **расширяемость сети** – можно увеличить протяженность, расширить пропускную способность каналов
4. **возможность применения симметричных интерфейсов обмена информацией между компьютерами сети** - позволяют произвольным способом распределять функции сети между компьютерами.

Компьютеры в сети:
1. **абонентские машины** (или основные компьютеры — **хосты**)
2. **коммуникационные** (или вспомогательные) компьютеры (шлюзы, маршрутизаторы и пр.) – вспомогательные функции, обеспечивающие выбор маршрута, передачу информации и т.д..

Взаимодействие компьютеров в сети - модели организации каналов
Сообщение - логически целостная порция данных, имеющая произвольный размер.
Сеанс связи - последовательность обменов сообщениями между абонентами сети.

1. сети коммутации каналов - коммутация каналов происходит на весь сеанс связи.

+ канал всегда находится в состоянии готовности
+ требования к оборудованию минимальны: буферизация не нужна
+ обмены происходят сообщениями, значит меньше накладных расходов по передаче информации
+ детерминированная пропускная способность сети

– дороговизна: выделенные каналы дорогие
– 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 оперируют термином «**передача сообщения**», а не «сеанс связи». Абонент-отправитель передаёт сообщение в сеть по первому доступному свободному каналу. 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. Если свободен, то сообщение передается далее, иначе происходит 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;

– в связи с тем, что сообщения могут быть произвольного размера, возникает необходимость наличия в коммутационных узлах средств буферизации (в общем случае неизвестно, какой мощности, поскольку сообщение имеет произвольную длину) – таким образом, данная 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. сети коммутации пакетов. Предполагается использования в сети ненадежных средств связи. Каждое сообщение разбивается на блоки фиксированного размера - ***пакеты***. **Передача сообщения** – это передача цепочки пакетов (движение пакетов по сети – аналогично предыдущей модели, но существенное различие между этими моделями заключается в детерминированности размера пакета). Соответственно, на стороне отправителя происходит разбиение сообщения на пакеты, а на стороне получателя — сборка. Любой пакет помимо непосредственно самого сообщения (или его части) = *служебная информация* (которая обычно представлена в заголовке пакета и обеспечивает внутреннюю целостность пакета - контрольная сумма пакета и пр.) + *адресная составляющая* (данные об отправителе и адресате) + *информацию для сборки* сообщения из пакетов.
При передаче пакета используется следующая стратегия: любой узел, получив пакет, пытается сразу от него избавиться. Модель допускает буферизацию в узлах передачи: пакет, придя на узел, может быть послан несколько позже, если все необходимые выходные каналы заняты. Но период занятости канала при известной стратегии обработки буфера предопределен, поэтому можно оценить *предельный размер буфера*, а также *предельные периоды ожидания* пакетов при передаче их по сети. Таким образом, если известна стратегия передачи, пропускная способность является детерминированной величиной.

+ детерминированность
+ 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.
- проблемой, связанной с разбиением сообщения на пакеты, является их сборка — это аккумуляция пакетов, а также сама сборка (необходимо обеспечить наличие всех переданных пакетов и их правильный порядок).

Организация сетевого взаимодействия. Эталонная модель ISO/OSI

Распределение функций между компьютерами в сети:
- централизованное (например, архитектура терминального комплекса)
- децентрализованное (сеть, состоящая из равнозначных по ролям и по функциям компьютеров)

Раньше архитектура и структура компьютера была монолитной
- сложно подключать к компьютеру новые устройства (нет взаимозаменяемости)
- нет программной преемственности (переносимость программ со старых компьютеров на новые)
- нет возможности децентрализовать производство компьютера (разные производители центральной части, периферийного оборудования, программного обеспечения и т.д.)

Та же самая проблема стандартизации прослеживалась и в программном обеспечении. В первую очередь подверглись стандартизации языки программирования. Описание любого языка нестрогое – 95% в нём совершенно чёткой и однозначной декларации синтаксиса и семантики языка, а 5% чётко не продекларированы, что могло приводить ко многим проблемам.

Важный этап стандартизации связан с появлением Unix (от стандартизации интерфейсов системных вызовов до стандартизации интерфейсов систем обработки команд). Появился стандарт POSIX (Portable Operating System Interface), который сейчас используется при разработке ОС.

(70-е – 80-е) **Международная организация по стандартизации** (ISO — International Organization for Standardization) предложила модель открытых интерфейсов (OSI — Open Systems Interconnection). Цель - построение стандарта, на основе которого можно было создавать компьютерные сети, открытые к расширению и модификации.
Модель ISO/OSI рассматривает сеть и взаимодействие компьютеров в сети в виде семи функциональных уровней. Взаимодействие в сети может осуществляться между одноимёнными (одноранговыми) уровнями. Для осуществления этого взаимодействия используются протоколы.
Протокол — это формальное описание сообщений и правил, по которым сетевые устройства (ВС) осуществляют обмен информацией. Протокол обеспечивает взаимодействие между различными сетевыми устройствами на одноименных уровнях. 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) протоколов, реализованных в системе. Стек может быть произвольной глубины (мб там протоколы не всех уровней).

При осуществлении взаимодействия информация должна быть сначала передана с текущего на первый уровень на данном сетевом устройстве, затем передана по коммуникационной среде, принята на другом сетевом устройстве, и, наконец, поднята до соответствующего уровня на другом сетевом устройстве.

1. Физический уровень.
- происходит передача неструктурированного потока двоичной информации через конкретную физическую среду (кабель, радиоволны и т.п.).
- декларируется стандартизация сигналов и соединений

2. Канальный уровень (или уровень **передачи данных**) - манипулирует порциями данных, которые называются **кадрами**. В кадрах присутствует избыточная информация для фиксации и устранения ошибок.
- обеспечивается доступность физической линии и передачу данных по физической линии
- обеспечивается синхронизацию (например, передающего и принимающего узлов)
- происходит борьба с ошибками
- делает линию связи надежной.

3. Сетевой уровень. На этом уровне решаются задачи взаимодействия сетей:
- обеспечивается управление операциями сети (адресация абонентов, маршрутизация)
- обеспечивается связь между взаимодействующими сетевыми устройствами
- происходит управление движением пакетов, и при необходимости поддерживается их буферизация.

4. Транспортный уровень.

- обеспечивается корректная транспортировка данных и взаимодействие между программой-отправителем и программой-получателем - обеспечивается программное взаимодействие (а не взаимодействие устройств)
- принимается решение, транспортировать данные с установлением виртуального канала или без него
    - Если да - осуществляется контроль за фактом доставки и обработка ошибок (программа-отправитель и программа-получатель взаимодействуют сообщениями)
    - Если нет - уровень не несет ответственности за доставку пакетов
- может обеспечиваться выявление и исправление ошибок при передаче

5. Сеансовый уровень.

- обеспечивается управление сеансами связи
- определяется активная сторона
- подтверждаются полномочия и пароли
- решаются задачи организации меток (контрольных точек по сеансу), которые отражают состояние сеанса связи и позволяют в случае возникновения сбоя восстанавливать сеанс с последней контрольной точки (повторять передачу не с начала, а с последней установленной контрольной точки)

6. Уровень представления данных

- обеспечивается унификация используемых в сети кодировок и форматов передаваемых данных

7. Прикладной уровень (уровень прикладных программ)

- формализуются правила взаимодействия с прикладными системами (например, с веббраузером). Ради этого уровня выстраивается вся структура организации сетевого взаимодействия.

Семейство протоколов TCP/IP - это четырехуровневая модель организации сетевого взаимодействия
Протоколы семейства TCP/IP основаны на сети коммутации пакетов. Цель – создание устойчивой децентрализованной сети, которая могла бы функционировать в коммуникационной среде, имеющей недетерминированную надёжность и производительность.

1. Уровень доступа к сети (физический и канальный ISO/OSI)

- специфицирует доступ к физической сети

Протоколы данного уровня обеспечивают передачу данных другим устройствам в сети.

Пример: семейство протоколов **Ethernet**. Эти протоколы предполагают наличие широковещательной сети, использующей единую магистраль (общую шину). Для сетевых устройств, подключаемых к этой магистрали, обеспечивается режим, называемый **множественный доступ с контролем несущей и обнаружением конфликтов** (CSMA/CD):
Широковещательная сеть - все сообщения, которые перемещаются по общей шине, видны всем адаптерам сетевых устройств
Множественный децентрализованный доступ - в любой момент любое из устройств может попытаться выдать сообщение на общую шину
Контроль несущей - каждый абонент, «слушая» сеть, распознает, свободна она или занята

- каждый адаптер слушает шину и забирает те сообщения, которые адресованы соответствующему сетевому устройству
- как только сеть становится свободной, устройство может «закидывать» очередную порцию данных
- устройство «слушает» как свою передачу, так и передачи других абонентов
- бросая сообщение в сеть, устройство способно распознать 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

- при интенсивной работе часто возникает ситуация, когда общая шина занята
- при интенсивной работе возрастает частота конфликтов => падает производительность системы

Виды физической среды передачи данных:

- «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

2. Межсетевой уровень (или **internet-уровень**) (сетево1 ISO/OSI)

- решаются проблемы адресации и маршрутизации по сети
- однако соединение с другими машинами не устанавливается

Как отмечалось выше, каждый из уровней взаимодействует с соседними уровнями (в соответствии с теми или иными протоколами) порциями данных, имеющими специфичные для каждого уровня названия. Так, для межсетевого уровня пакет называется **дейтаграммой**.

Протокол IP — один из основных протоколов.

- формирует дейтаграммы
- поддерживает систему адресации
- реализует обмен данными между транспортным уровнем и уровнем доступа к сети
- маршрутизирует дейтаграммы
- разбивает и обратно собирает дейтаграммы
- не обеспечивает обнаружение и исправление ошибок.

- 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.

Класс	Первые биты	Номер сети, бит	Диапазон сетей	Максимальное число сетей	Максимальное число хостов в сети
A	0	8	1.0.0.0 – 126.0.0.0	126	16 777 214
B	10	16	128.0.0.0 – 191.255.0.0	16 382	65 534
C	110	24	192.0.0.0 – 223.255.255.0	2 097 150	254
D	1110	-	224.0.0.0 – 239.255.255.255	Групповые адреса	
E	11110	-	240.0.0.0- 255.255.255.255	Зарезервировано	

*класс A принадлежит межнациональным корпорациям
*среди IP-адресов классов A, B и C имеются предопределённые (если номер сети и номер сетевого устройства равны нулю - это IP-адрес текущего сетевого устройства)
*класс D предназначен для многоадресной рассылки.
*есть возможность экономного расходования IP-адресов.
Стратегия выделения:
1. долговременная (постоянные IP-адреса)
2. кратковременная (IP-адреса выделяются на время сеанса)

1. организация адресации в локальной сети, в рамках которой происходит взаимодействие (несложно)
2. организация адресации между различными сетями (используются шлюзы – устройства (спец.компьютеры), которые одновременно принадлежат разным сетям и физически объединяют сети, а также маршрутизаторы, которые решают, через какой шлюз необходимо отправить пакет)

*одна и та же машина может быть одновременно и шлюзом, и маршрутизатором, и хостом.
**на шлюзах реализовано только два уровня протоколов (больше и не надо: из одной сети (протокол1) в другую (протокол2))
***по имеющему IP-адресу получателя определяется маршрут следования пакета – маршрутизация дейтаграмм

3. Транспортный уровень (сеансовый и транспортный ISO/OSI)

- обеспечивается доставка данных от компьютера к компьютеру
- обеспечиваются поддержка логических соединений между прикладными программами
- не всегда происходит контроль за ошибками и их коррекция

Два основных протокола транспортного уровня – **TCP** и **UDP**.

TCP (Transmission Control Protocol — протокол управления передачей данных) оперирует передачей сообщений. Сообщения разбиваются на пакеты, и эти пакеты выбрасываются в сеть. Принимающая сторона на каждый пришедший пакет отправляет подтверждение и собирает соответствующие пакеты в сообщение. При этом порядок отправки пакетов может не соответствовать порядку их прибытия на принимающую сторону.

- устанавливается виртуальный канал
- обеспечивается последовательная передача пакетов
- контролируется доставку пакетов
- отрабатываются сбои (пакет либо не доставляется, либо доставляется в целостном состоянии)
- 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

- протокол подразумевает отправку подтверждающей информации по сети => пропускная способность может сильно падать, особенно на линиях связи с плохими техническими характеристиками

(альтернатива) **UDP** (User Datagram Protocol — протокол пользовательских дейтаграмм)

- оперирует передачей пакетов
- 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 (уровень представления данных и уровень прикладных программ 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

На этом уровне находятся протоколы, часть которых опираются на протокол TCP, а часть — на UDP (зависит от критичности потери информации, и степени детерминированности качества линий связи)

Протоколы, которые основываются **на принципах работы протокола *TCP*** обеспечивают доступ и работу с заведомо корректной информацией, причем именно в среде межсетевого взаимодействия (internet), и эти протоколы требуют корректной доставки.
1.TELNET (Network Terminal Protocol) — прикладной протокол, эмулирующий терминальное устройство (сетевой терминал)
2.FTP (File Transfer Protocol) - протокол межсетевого перемещения файлов
3.SMTP (Simple Mail Transfer Protocol) - протокол передачи почтовых сообщений

Протоколы, которые основываются **на принципах работы протокола UDP**:
+ оказываются относительно быстрыми, поскольку максимально снижены накладные расходы на передачу
- допускают наличие ошибок

Часть подобных протоколов действуют в рамках локальной сети, где качество линий связи детерминировано.
Пример: **NFS** (Network File System) сетевой файловой системы функционирует именно в рамках *локальной сети*, и очень редко его запускают в межсетевом режиме.

Другая часть протоколов должна контролироваться, с одной стороны, на прикладном уровне, а с другой стороны, эти протоколы предполагают обмен очень небольшими порциями данных. Пример: **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 (пример – *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.). Существует организация, которая распределяет доменные имена первого уровня. Владелец доменного имени i-го уровня может по своему усмотрению распределять доменные имена (i+1)-го уровня.

Эти уровни модели TCP/IP являются **пакетными**: 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.

+ 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

Основы архитектуры операционных систем

Операционная система — это комплекс программ, который контролирует:
1. **Существование** виртуальных ресурсов
2. **Наличие** доступа к физическим *Для любого ресурса степень его доступности зависит от ОС
3. **Использование** ресурсов
4. **Распределение**: ОС выбирает стратегию распределения.

Процесс – одна из базовых сущностей любой ОС, которая обозначает исполняемую программу, задачу или задание и определяет некоторый процесс исполнения последовательности команд (может быть как одна единственная ветвь вычислений, так и несколько параллельных ветвей сразу).
Процесс — это совокупность машинных команд и данных, обрабатывающаяся в рамках ВС и обладающая правами на владение некоторым набором ресурсов ВС.

Совокупности машинных команд и данных - то, что принято называть исполняемой программой (это код и операнды, используемые в этом коде). Далее, 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, третье, понятие **обладания правами на владение некоторым набором ресурсов** обозначает возможность доступа.

Типы ресурсов, выделяемых процессам:
1. на эксклюзивных правах - ресурс принадлежит только тому процессу, который им владеет (на время владения)
2. разделяемые ресурсы могут принадлежать многим процессам одновременно (то, что разделяемый ресурс может одновременно принадлежать нескольким процессам, не означает, что к нему возможен одновременный доступ)

Стратегии выделения ресурсов:
1. предварительная декларация ресурсов. Перечень ресурсов (объемы ОП, время работы), которыми будет обладать процесс ОС формирует заранее и старается их выделить до ввода программы в систему и запуска процесса. Если в системе нет заказанного ресурса, то она, скорее всего, не станет запускать процесс, который запросил этот ресурс.
2. динамическое пополнение списка ресурсов. ОС выделяет ресурсы процессу уже во время его выполнения по мере необходимости. Запуск процесса происходит с минимально необходимой ему областью виртуальной памяти.

ОС обеспечивает:
1. надежность - число программных ошибок в системе должно минимально и соизмеримо с количеством возможных аппаратных сбоев
2. защиту информации и ресурсов от несанкционированного доступа
3. эффективность - функционирование системы должно удовлетворять некоторым требованиям – критериям эффективности
4. предсказуемость системы - при возникновении разного рода форс-мажорных обстоятельств ОС должна вести себя строго определенным способом. Это свойство должно очерчивать круг всевозможных проблем, которые могут возникнуть в той или иной ситуации, предопределять последствия этих проблем, а также подразумевать устойчивость системы к возникновению сбоев (как аппаратных, так и программных)

Структура ОС

1. Ядро (kernel) - постоянно размещаемая в ОП часть, которая реализует некоторую базовую функциональность ОС, работает в режиме супервизора (в привилегированном режиме) и может включать в свой состав драйверы физических или виртуальных устройств

2. Уровень динамически подгружаемых драйверов физических и виртуальных устройств - в зависимости от ситуации состав этих драйверов при загрузке системы может меняться
1. **резидентные драйверы** подгружаются в систему в процессе ее загрузки и находятся в ней до завершения ее работы (драйвер физического диска т.к. это устройство оперативного доступа, поэтому к моменту полной загрузки системы все должно быть готово)
2. **нерезидентные (динамически подгружаемые) драйверы** вызываются ОС на сеанс работы с соответствующими устройствами, а до этого находятся в ОЗУ или отключены (сканер - используется нечасто и скорость работы много меньше скорости загрузки драйвера из внешней памяти в оперативную)

Стратегии выбора:
1. может быть явное указание системе списка драйверов, которые необходимо подгрузить (в этом случае, если в списке что-то будет указано неправильно, то соответствующее устройство, возможно, просто 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).

С**истемный вызов -** средство обращения процесса к ядру ОС за выполнением той или иной функции (открытие файла, чтение/запись в него, порождение процесса и т.д).

Отличие обращения к системному вызову от обращения к библиотеке программ заключается в том, что библиотечная программа присоединяется к исполняемому коду процесса и вычисление библиотечных функций будет происходить в рамках процесса. Обращение к системному вызову — это вызов тех команд, которые инициируют обращение к системе. Инициацией обращения к ОС может служить либо прерывание, либо исполнение специальной команды. Например, осуществляя работу с файлом, имеется возможность работы с ним посредством обращения к системным вызовам либо посредством использования библиотеки ввода-вывода. В последнем случае в тело процесса включаются дополнительные функции из данной библиотеки, а уже внутри данных функций происходит обращение к необходимым системным вызовам.

Структурная организация ОС:

1. монолитное ядро:

- представляет собою единую монолитную программу, в которой отсутствует явная структуризация (напоминает физическую организацию первых компьютеров: в них также нельзя было выделить отдельные физические функциональные блоки — все было единым, монолитным и интегрированным друг с другом)
- содержит фиксированное число реализованных в нем базовых функций

- модификация функционального набора достаточно затруднительна (необходима практически полная переделка ядра)
+ минимальное количество интерфейсных сочленений, связанных со структуризацией
- отсутствует универсальность, и внутренняя организация ядра рассчитана на конкретную реализацию
- необходимость перепрограммировать ядро при внесении изменений > разработчик > дорого

2. многослойные ОС.

- все уровни разделяются (индивидуально для каждой ОС) на некоторые функциональные слои
- между слоями имеются фиксированные интерфейсы (взаимодействие слоев)
- каждый слой предоставляет определенный сервис вышестоящему слою

+ модернизация подобных систем сводится к модернизации соответствующих слоев
- 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)

3. микроядерная архитектура

- выделяется минимальный набор функций (например, первичная обработка прерываний и некоторые функции управления процессами), которые включаются в ядро
- вся остальная функциональность представляются в виде драйверов, которые подключаются к ядру посредством некоторого стандартного интерфейса

+ расширяема
+ почти не имеет никаких ограничений по количеству подключаемых драйверов и их функциональному наполнению
+ требуется только соблюдение драйвером интерфейса для обращения к микроядру
+ высокотехнологична
+ хорошо подходит для применения в современных многопроцессорных ВС (например, в SMP-системах - можно распределять драйверы по различным процессорами и получать соответствующую эффективность)
- большие накладные расходы:

процесс системным вызовом обращается к ФС, чтобы произвести обмен с конкретным файлом	драйвер ФС, получив запрос, перерабатывает его в последовательность запросов на обмен с диском (пусть это будут виртуальный диск)	ФС обращается к микроядру
драйвер передаёт запросы физическому диску по той же схеме	драйвер виртуального диска определяет, с каким физическим диском будет происходить обмен, и трансформирует поступивший ему запрос в запросы к этому физическому диску	микроядро находит драйвер виртуального диска и передает ему соответствующий запрос

Таким образом, один запрос распадается на множество подзапросов, следующих от драйвера через микроядро к другому драйверу, из-за чего эффективность системы снижается.

Логические функции ОС

1. **управления процессами**
- формирование процессов
- поддержание жизненного цикла процесса
- организация взаимодействия процессов с системой в целом и с другими процессами в частности
- работа процессов с ресурсами

2. **управления ОП**
- выбор стратегии организации ОП (в частности, реализуется поддержка аппарата виртуальной памяти)
- защита ОП от несанкционированного доступа
- обеспечение корректности работы процесса с выделенной ему ОП
- выделение и изъятие памяти у процессов (пересекаются с функциями планирования)

3. **планирования**
- регламентация доступа конкурирующих процессов к ресурсам - обработка очередей запросов к ресурсам
- планирование доступа процессов к ЦП
- распределение времени ЦП
- обработка очередей обмена. В процессе функционирования системы формируется поток запросов на обмен, и очень часто этот поток может превышать пропускную способность устройства - образуется конкуренция по доступу к устройству, выстраивается очередь запросов на обмен. Необходимо учитывать приоритетность при обработке категорий запросов, которые идут от ОС и которые идут от пользователей.
- обработка прерываний. При обработке прерываний также есть приоритетность, так как прерывание иногда требует достаточно больших ресурсов системы и может возникать очередь прерываний.

4. **управление ВУ и ФС** - достаточно значимое для ОС виртуальное устройство
- Управление устройствами, которые не требуют оперативного доступа
- Управление устройствами, которые требуют оперативный доступ (которые могут содержать важную информацию и которые должны обеспечивать надёжность сохранения этой информации)

Организация управления внешними устройствами должна осуществляться надёжно и эффективно - многоуровневая буферизация запросов к ВУ и сложные схемы обеспечения надёжности обменов с ВУ (например, программные средства дублирования) и т.д.

5. **обеспечение сетевого взаимодействия**. Практически любая современная ОС должна иметь средства взаимодействия с другими компьютерами в сети, то есть ОС должна обеспечивать функционирование и реализацию сетевых протоколов.

6. **обеспечения безопасности** – чтобы дин зарегистрированный в системе пользователь системы не мог добраться до информации другого пользователя и чтобы незарегистрированный пользователь не мог получить доступ к системе. Развитие сетевого взаимодействия существенно усложнило проблему безопасности, так как из-за концепции открытых интерфейсов (ISO/OSI, TCP/IP) у компьютера появилось множество логических «входов», через которые программы других компьютеров могут связываться с данным. Появилось понятие компьютерной атаки на сетевое устройство с целью нарушения защиты (либо с целью несанкционированного доступа к данным, либо с целью нарушения функциональности устройства)
- устойчивость системы к возможным атакам
- анализ функционирования системы и выявление попыток вторжения в систему (это также важно, поскольку очень редко, когда взлом происходит сразу)
- минимизация ущерба в случае вторжения.

Типы ОС

1. пакетная ОС для решения расчетных задач (требующих определенного объема времени работы ЦП)
Пакет программ — это некоторая совокупность программ, которые системе необходимо обработать. Переключение ЦП с одного процесса на другой происходит только…

1. завершение выполнения процесса (в силу успешного перехода на точку завершения программы или возникновения ошибки)
2. обращение к ВУ с целью осуществления обмена, т.е. возникновение прерывания по вводу-выводу, поскольку операция обмена так или иначе требует какого-то минимального интервала времени
3. фиксация факта зацикливания процесса (часто зацикливанием считают исчерпание процессорного времени)

+ минимизируются накладные расходы так как редки обращения к функции ОС смены процесса
+ степень полезной загрузки ЦП от 90%
+ время работы процессора ~ время исполнения пользовательских программ

2. система разделения времени: пакетная система + для каждого процесса в системе определяется **квант процессорного времени** - некоторый фиксированный промежуток времени работы ЦП. Переключение процессов происходит по тем же причинам, что и в пакетных системах + исчерпался выделенный квант времени.
- минимизация времени отклика системы на запрос пользователя (набор текста – набранные символы отображались на экране достаточно быстро)
- большие накладные расходы из-за частой смены контекстов
- эффективность 30–40%, 60–70% - накладные расходы

Увеличивая квант времени до некоторого среднего размера (порядка нескольких секунд), можно получить пакетную систему, ориентированную на обработку отладочных программ. А если увеличить размер кванта до бесконечности, получится пакетная система в чистом виде.

При организации планирования времени ЦП необходимо:

1. разделить все процессы на группы по некоторым критериям (например, интерактивные, отладочные и т.д.).
2. для каждой из этих групп определить квант времени ЦП, который будет выделяться процессу из конкретной группы.
3. определить приоритеты для каждой из категорий процессов (в критерии смены обрабатываемого процесса можно добавить ещё один пункт – появление процесса из более приоритетной группы)
4. приоритеты категорий процессов должны определяться по расписанию (например, в зависимости от времени суток)

3. реального времени - специализированные системы, которые будут работать на ВС, управляющих какими-то технологическими процессами. Все функции планирования ориентированы на обработку некоторого фиксированного набора событий. Для любого события гарантируется его обработка в пределах заданного промежутка времени.

Пример: процесс кипячения молока. Если емкость с молоком постоянно нагревать, то через некоторое время оно начинает кипеть, а еще через некоторый достаточно короткий период оно «убегает» (после чего вообще начинает подгорать). Процесс кипячения молока можно автоматизировать, если в сосуд с молоком поместить датчик температуры, который снимает текущее значение температуры молока и передает это значение компьютеру. Соответственно, ставится задача «поймать» момент фиксации температуры кипения молока, причем среагировать необходимо за некоторый фиксированный промежуток времени. Если реакция произойдет, положим, через минуту, то молоко «убежит», и, соответственно, польза от такой системы будет минимальной.

Выделяют различные группы систем реального времени: жесткого времени (например, управление бортовой системой самолёта), мягкого времени и пр.

4. сетевые ОС обеспечивают функционирование и взаимодействие ВС в пределах сети. ОС устанавливается на каждом компьютере сети и обеспечивает работу распределенных приложений (реализации функций находятся в разных компьютерах сети).

5. распределенные ОС функционируют 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). Примером распределенных функций может служить функция управления заданиями (в кластерных системах задание может представлять собою целое множество процессов, и ставится задача распределить эти процессы по имеющимся процессорным узлам). Другим примером может служить распределенная файловая система. Традиционная ФС ОС Unix просто не справится с потоками информации между узлами многопроцессорных систем, поэтому необходимы принципиально новые решения организации хранения и доступа к файлам.

Управление процессами

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:
– образование
– выполнение на процессоре,
– ожидание постановки на исполнение (ожидание какого-либо события: окончания обмена, выделения ресурса ЦП и пр.)
– завершение (возврат ресурсов)

Пусть ОС обеспечивает существование процессов в двух состояниях:
1. размещение процесса или программы в **буфере ввода процессов** (**БВП**). Там процессы хранятся до начала обработки ЦП.
2. размещением процесса в **буфере обрабатываемых процессов** (**БОП**). Там находятся все процессы, которые начали обрабатываться ЦП.

Различные модели ОС и управление жизненным циклом процесса:

пакетная однопроцессная
1. формирование и ожидание начала обработки (БВП)
2. обработка (переход из БВП в БОП)
3. завершение процесса, освобождение системных ресурсов

*нет ожидания готовых процессов или ожидания ввода-вывода - это однопроцессная система, которая обрабатывает один процесс, причем все обмены синхронные, и процесс никогда не откладывается

пакетная мультипроцессная:

выходя из БОП**,** процесс может либо завершиться, либо перейти в состояние ожидания ввода-вывода, если процесс обращается к операции обмена. В этот момент система может запустить некоторый процесс либо из БВП, либо из очереди готовых на выполнение процессов. Соответственно после того, как процесс завершил обмен, он меняет свой статус и попадает в очередь на выполнение, из которой позже он попадет снова на выполнение.

с разделением времени
+ возможность перехода из состояния обработки ЦП в очередь готовых на выполнение процессов (система имеет возможность прервать выполнение текущего процесса (если исчерпался выделенный квант времени) и поместить процесс в указанную очередь)

- модель не предполагает свопинга (механизма откачки процесса во внешнюю память). Такую возможность можно добавить, тогда появляется еще одно состояние процесса - откачанный во внешнюю память, сюда могжно попасть только из очереди готовых на выполнение процессов, а процессы, ожидающие окончания ввода-вывода, свопироваться не могут, иначе в системе будут «зависать» заказы на обмен.

Типы процессов

Процесс (или **полновесный процесс**) – является объектом планирования и выполняется внутри защищённой области памяти. Альтернативой являются **легковесные процессы** / **нити** / **потоки** - это процессы другого иерархического уровня, которые могут активироваться внутри одного полновесного процесса (тогда они работают в одном адресном пространстве и незащищены друг от друга) и могут быть объектами планирования (планировщик может осуществлять переключение с нити на нить).
+ минимизация накладных расходов за счет уменьшения количества смен контекстов
+ хорошо подходит современным многопроцессорным системы (например, SMP-системы) (иногда повышается эффективность системы)
+ удобно осуществлять взаимодействие нитей в рамках одного процесса
- возникает задача ОС управления нитями

Тогда процесс = исполняемый код + собственное адресное пространство, представляющее собой множество виртуальных адресов, которые может использовать процесс + ресурсы системы, которые ОС дала процессу + хотя бы одна выполняемую нить

Контекст процесса - совокупность данных, характеризующих актуальное состояние процесса.

1. пользовательская (программная) составляющая — это текущее состояние программы (совокупность машинных команд и данных, размещённых в ОЗУ и характеризующих выполнение данного процесса), тело процесса:

1. Сегмент кода содержит машинные команды и неизменяемые константы - неизменяемая программно часть тела процесса (в принципе системные вызовы в силе изменить ее)
2. Сегмент данных = область статических данных процесса (в т.ч. статические переменные) + область разделяемой памяти (может принадлежать двум и более процессам одновременно) + область стека, на котором в системе реализуется передача фактических параметров функциям, реализуются автоматические и регистровые переменные, а также в этой области организуется динамическая память - куча.

Разделение сегмента кода Unix: оптимизация использования ОП. Допустим, в системе в одним и тем же текстовым редактором работает 100 пользователей => в системе обрабатываются 100 копий редактора. Разделение - каждый процесс текстового редактора в пользовательской составляющей хранит ссылку на единственную копию сегмента кода редактора, а сегмент данных у каждого из процессов свой (так не приходится держать в ОЗУ все сегменты кода для этих 100 процессов) => в памяти будут находиться один сегмент кода и 100 сегментов данных.
* только в том случае, когда сегмент кода нельзя изменить: он закрыт на запись.

2. аппаратная составляющая – отражает актуальное состояние ЦП в момент выполнения данного процесса: актуальное состояние регистров, аппаратные таблицы процессора, настроек процессора и т.д.; Конкретная структура зависит от конкретного процессора; обычно она включает счётчик команд, регистр состояния процессора, аппарат виртуальной памяти, регистры общего назначения и т.д.

3. системная составляющая – это структуры данных ОС:

- идентификация процесса (точнее пользователя, сформировавшего процесс):
  - идентификатор родительского процесса
  - реальный идентификатор пользователя-владельца (идентификатор 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)
- содержимое регистров (РОН, индексные регистры, флаги)
- информация для управления процессом (состояние процесса, приоритет)
  - текущее состояние процесса
  - приоритет процесса
  - список областей памяти
- информация об открытых и используемых в процессе файлах и прочее
- таблица дескрипторов открытых файлов процесса (именно дескрипторов, т.к. один и тот же файл может быть открыт в системе многократно)
- 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). Системная составляющая процесса содержит копию аппаратной составляющей, если процесс остановлен => когда процесс выполняется на процессоре, то актуальна аппаратная составляющая, когда отложен — актуальна системная составляющая.

Исполняемый файл – 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
Формирование процесса в Unix - запуск исполняемого файла на выполнение.

- Каждый пользователь системы имеет свой идентификатор (UID — User ID).
- Каждый файл имеет своего владельца, т.е. для каждого файла определен UID пользователя-владельца (но разрешено запускать и файлы, не принадлежащие конкретному пользователю)
- Большинство команд ОС Unix представляют собой исполняемые файлы, принадлежащие системному администратору (суперпользователю)

Соу при запуске файла определены фактически два пользователя: пользователь-владелец файла и пользователь, запустивший файл (пользователь-владелец процесса). И эта информация хранится в контексте процесса, как реальный идентификатор — идентификатор владельца процесса, и эффективный идентификатор — идентификатор владельца файла.

А дальше возможно следующее: можно подменить права процесса по доступу к файлу с реального идентификатора на эффективный идентификатор. Соответственно, 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. И в этом случае права рядового пользователя заменятся на права администратора, поэтому пользователь сможет сохранить новый пароль в системной таблице (в соответствующем файле).

Реализация процессов в ОС Unix

Двойное определение процесса:

1. Объект, зарегистрированный в таблице процессов ОС. **Таблица процессов** – одна из специальных системных (программных) таблиц, предназначенная для регистрации всех существующих в данный момент процессов. Ее размер - параметр ОС, и, соответственно, количество процессов в системе является системным ресурсом. Таблица процессов каждому процессу дает имя-номер в таблице от 0 до N – 1. Этот номер - **идентификатор процесса** (PID — Process Identifier).

2. Объект, порожденный системным вызовом fork() - он обеспечивает создание копии текущего процесса. **Системный вызов** - средство ОС, которое предоставляется пользователям (процессам), с помощью которого они обращаются к ядру ОС за выполнением каких-то функций. Выполнение системных вызовов происходит в привилегированном режиме (поскольку их непосредственную обработку производит ядро), даже если сам процесс выполняется в пользовательском режиме. С.в. может считаться как специфическим прерыванием, так и как команда обращения к ОС.

Базовые средства управления процессами в ОС Unix

Все процессы в Unix (кроме 0 и 1) порождаются одинаково - системный вызов fork(). При обращении процесса к данному системному вызову:

1. ОС создает копию текущего процесса (тело копии полностью идентично исходному процессу)
2. ОС делает новую запись в таблице процессов
3. новый порождённый процесс получает уникальный идентификатор
4. для нового процесса создается контекст - большая часть содержимого идентична контексту родителя: сегмент кода и данных.

Наследство сына:

- 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. (Речь идет о том, что в системе при открытии файла с ним ассоциируется некоторый атрибут, который определяет правила передачи этого открытого файла сыновним процессам. По умолчанию открытые в «отце» файлы можно передавать «потомкам», но можно изменить значение этого параметра и блокировать передачу открытых в процессе-отце файлов.);
- способы обработки сигналов;
- разрешение переустановки эффективного идентификатора пользователя;
- разделяемые ресурсы процесса-отца;
- текущий рабочий и домашний каталоги;

Сын 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.

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

pid_t fork(void);
```

По завершении *fork()* отец и сын, получив управление, продолжает выполнение с одной и той же инструкции одной и той же программы, а именно, с той точки, где происходит возврат из *fork()*. Вызов *fork()* в случае успешного завершения возвращает сыновнему процессу значение 0, а родительскому процессу — *PID* порожденного процесса. В случае неудачного завершения (т.е. если сыновний процесс не был порожден, например, по причине отсутствия свободного места в таблице процессов), системный вызов *fork()* возвращает –1, а код ошибки устанавливается в переменной *errno*.
*отец и сын работают независимо с точки зрения системного управления процессами: в частности, порядок их обработки на

процессоре в общем случае пользователю неизвестен и зависит от той или иной реализованной в системе стратегии планирования времени процессора.

Системные вызовы семейства *exec()* – спутники *fork(),* обеспечивают замену тела текущего процесса на тело, образованное в результате загрузки исполняемого файла, и управление передается на точку входа в новое тело.
Суффикс – уточнение сигнатуры того или иного вызова.

```
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);
```

Параметры:
1. имя файла программы, подлежащей исполнению
   - файл должен быть исполняемым
   - пользователь-владелец процесса должен иметь право на исполнение данного файла

Для функций с суффиксом «p» в названии имя файла может быть кратким, при этом при поиске нужного файла будет использоваться переменная окружения PATH.

2. аргументы командной строки для вновь запускаемой программы, которые отобразятся в ее массив *argv*:
   - в виде списка аргументов переменной длины для функций с суффиксом «l»
   - в виде вектора строк для функций с суффиксом «v».

В любом случае, в списке аргументов должно присутствовать как минимум 2 аргумента: имя программы, которое отобразится в элемент *argv[0]*, и значение NULL, завершающее список.

3. В функциях с суффиксом «e» имеется также дополнительный аргумент, описывающий переменные окружения для вновь запускаемой программы – это массив строк вида *name=value*, завершенный значением NULL.

Пусть сын *экзеклит* ls -l (отображает содержимое текущего каталога). Реализация данной команды хранится в файле /bin/ls. После успешного завершения *execl(),* сын будет содержать реализацию команды ls, и управление в нем будет передано на точку входа (т.е. запустится функция *main() из файла* bin/ls). Возврат к первоначальной программе происходит только в случае ошибки *exec()*.

Выполнение «нового» тела происходит в рамках уже существующего процесса, т.е. после вызова *exec()* сохраняется идентификатор процесса, и идентификатор родительского процесса, 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 следующая 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 *close-on-exec*. Соответствующие файловые дескрипторы будут помечены как свободные.

Завершение процесса:
1. Сигнал процессу (программный аналогом прерывания)
2. Системный вызов завершения процесса:
   a. явный: ***_exit().*** Он никогда не завершается неудачно => нет возвращаемого значения. С помощью единственного параметра *status* процесс может передать породившему его процессу информацию о статусе своего завершения - программный код завершения процесса. Возврат 0 – успешное завершении процесса, не 0 - ошибочное завершение.
   b. неявный: *return* языка С внутри функции main() или выход на закрывающую скобку функции main(). В последнем случае компилятор заменит действие оператора return обращением к *exit()*.

Так как процесс не может завершиться мгновенно, он переходит состояние ***зомби***:
- корректно освобождаются ресурсы (закрываются все открытые дескрипторы файлов, освобождаются сегмент кода и

сегмент данных процесса и пр.)

- 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
- если остаются сироты, их усыновляет процесс с номером 1 (в Unix)
- отцу умирающего процесса передается сигнал SIGCHLD (в большинстве случаев его игнорируют).

Процесс-предок имеет возможность получить информации о статусе завершения/приостановки своего потомка. Для этого служит системный вызов *wait()*:

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

pid_t wait(int *status);
```

1. если к моменту обращения какие-то дети уже завершились, то родитель получит информацию об одном из этих детей
2. если у процесса нет детей то *wait()* вернет -1
3. если у процесса ни один из детей еще не завершился, то при обращении к wait() отец будет блокирован до завершения любого из своих детей (если отец хочет получить информацию о завершении каждого из своих потомков, он должен несколько раз обратиться к вызову *wait()*).

В случае успешного завершения возвращается *PID* завершившегося потомка или -1 в случае ошибки или прерывания. А через параметр *status* передается указатель на целочисленную переменную, в которой система возвращает процессу информацию о причине завершения потомка: *пользовательский код завершения процесса* (старший байт), передаваемый в качестве параметра *_exit() + системный код завершения процесса* (младший байт) - индикатор причины завершения сына, устанавливается ядром Unix и хранит номер сигнала, приход которого в сыновний процесс вызвал его завершение.
*В случае если отец 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). С другой стороны, имеется возможность изменить режим работы системного вызова *wait()* таким образом, чтобы отцовский процесс не блокировался в ожидании завершения одного из потомков, а сразу получал соответствующий код ответа.

- после передачи родителю статуса завершения все структуры, связанные с процессом-зомби, освобождаются, и запись о нем удаляется из таблицы процессов.

Становление зомби необходимо, чтобы процесс-предок мог получить информацию о судьбе своего завершившегося потомка, независимо от того, вызвал он wait() до или после его завершения. Если предок вообще не обращался к *wait()* и/или завершился раньше потомка, то для всех его потомков отцом становится процесс с идентификатором 1, и он уже вызывает *wait()*, освобождая все структуры, связанные с потомками зомби.

Жизненный цикл процесса. Состояния процесса

1. Создание: *fork()* создает новый процесс в системе (иных способов создать процесс в Unix не существует).
2. Практически сразу он попадает в **очередь** процессов, **готовых к выполнению**.
3. По решению планировщика процесс может выйти из очереди в…
   a. **состояние выполнения**, а затем обратно в состояние **готовности к выполнению** (например, в случае исчерпания кванта времени обработки на процессоре). Переходы между этими двумя состояниями основываются на приоритете процесса: чем дольше процесс выполняется, тем ниже становится его приоритет, но чем дольше процесс находится в состоянии готовности к выполнению, тем выше становится его приоритет. В состоянии выполнения процесс может работать как в пользовательском режиме, так и в режиме ядра (супервизора). В режиме ядра процесс работает при обращении к системному вызову. Тогда ядро ОС выполняет (на своих ресурсах) некоторые действия для конкретного процесса.
   b. **состояние блокировки** или ожидания завершения некоторого действия – например, завершения взаимодействия с внешним окружением. Соответственно, при возникновении соответствующего события процесс переходит из состояния блокировки в состояние готовности к выполнению.
4. *exit()* - «зомби» - состояние завершения существования процесса
5. после получения отцом информации о завершении данного процесса он завершает свой жизненный цикл.

Упрощения:
- нет свопинга - откачки процесса во внешнюю память
- реальные современные Unix-системы оперируют с нитями => особенности в жизненный цикл

Формирование процессов 0 и 1 - два системных процесса, которые существуют в течение всего времени работы системы

1.Включение компуктера. Практически во всех компьютерах имеется **постоянное запоминающее устройство (ПЗУ)** - область памяти, способная постоянно хранить информацию. В этой области памяти постоянно находится программа, которая называется **аппаратный загрузчик компьютера**. При включении компьютера схемы управления запускают работу компьютера с адреса точки входа в этот аппаратный загрузчик. Он знает список системных устройств компьютера, которые априори могут содержать ОС, и их приоритеты. **Системное устройство** – это блок-ориентированное устройство прямого доступа, на котором может размещаться ОС. В соответствии с приоритетом загрузчик проходит по списку и выбирает, откуда грузить ОС.

Например, бывает полезно сделать floppy-диск наиболее приоритетным для загрузки ОС, так как в нормальном режиме аппаратный загрузчик этот диск не обнаружит, и загрузка будет осуществляться с жёсткого диска, а в случае «гибели» жёсткого диска мы как раз воспользуемся возможностью загрузки с floppy-диска.)

Аппаратный загрузчик «знает» структуру системного устройства - обычно в нулевом блоке системного устройства находится **программный загрузчик**, который знает, содержатся ли ОС на этом устройстве и какие они. **Раздел системного устройства** — это последовательность подряд идущих блоков (выделенная на внешнем запоминающем устройстве), внутри которых используется виртуальная нумерация этих блоков (каждый раздел начинается с нулевого блока). Соответственно, если ОС несколько, то программный загрузчик может предложить пользователю компьютера выбрать, какую систему загружать. После этого программный загрузчик обращается к соответствующему разделу данного системного устройства и из нулевого блока выбранного раздела считывает загрузчик конкретной ОС, после чего начинает работать программный загрузчик конкретной ОС. Этот загрузчик, в свою очередь, «знает» структуру раздела, структуру ФС и находит в соответствующей ФС файл, который должен быть запущен в качестве ядра ОС. Что касается Unix-систем, то программный загрузчик ОС осуществляет поиск, считывание в память и запуск на исполнение файла */unix*, который содержит исполняемый код ядра ОС Unix.

2. Действия ядра при запуске

1. инициализация аппаратных и программных компонентов системы
2. установка начальных параметров в аппаратных интерфейсах:
   - устанавливаются системные часы (для генерации прерываний)
   - формируется диспетчер оперативной памяти
   - устанавливаются средства защиты оперативной памяти.
3. формирование системных программных структур данных:
   - создается таблица процессов
   - устанавливается размер КЭШ-буфера
4. создание нулевого процесса:
   - резервируется память под его контекст
   - формируется нулевая запись в таблице процессов и более ничего - это и есть ***создание нулевого процесса***. Этому нулевому процессу в общем случае соответствует ядро ОС (это процесс ядра), но он имеет особенность – отсутствие в контексте процесса сегмента кода (отметим, что это вовсе не означает, что нулевой процесс не имеет кода). Это означает, что нулевая запись таблицы процессов ссылается на контекст, в котором отсутствует ссылка на сегмент кода процесса
5. создание первого процесса - init:
   - копирование нулевой записи (в таблице процессов) в первую
   - для первого процесса выделяется пространство оперативной памяти
   - в эту память загружается программа исполнения системного вызова *exec()*
   - внутри первого процесса происходит обращение к этому системному вызову с параметром */etc/init*.

*сам первый процесс формируется нестандартным путем, но тело его в конце формируется уже «правильным» образом посредством вызова *exec()*

<u>режимы работы системы</u>: (определяется на стадии загрузки ядра и инициализации системы, система опознает один из подключенных терминалов как системную консоль)

1. **однопользовательский**: интерпретатор команд подключается к системной консоли (консоль регистрируется с корневыми привилегиями и доступ по каким-либо другим линиям связи невозможен)
2. **многопользовательский**: **init** обращается к системной таблице терминалов, хранящей все терминальные устройства, которые могут быть в системе, и для каждого готового к работе терминала из этого перечня он запускает процесс **getty**. Он обеспечивает работу конкретного терминала (сеанс работы пользователя). В свою очередь, процесс **getty** печатает на экране приглашение ввести логин. После того, как пользователь вводит логин, процесс **getty** загружает на свое место процесс подтверждения пароля **login**. Соответственно, процесс **login** запрашивает ввода пароля, после чего проверяет его. Верно - начинается сеанс работы с пользователем. Сеанс работы с пользователем, как и сеанс работы всей системы, определяется конфигурационной информацией. Эта информация может храниться в зашифрованном виде в файле регистрации паролей *passwd*. Каждая запись этого файла содержит имя зарегистрированного пользователя, пароль,

учётную информацию и некоторые настройки работы пользователя:

- домашний каталог - каталог файловой системы, который станет текущим, когда пользователь начинает сеанс работы
- информация о том интерпретаторе команд (**shell**), который должен быть загружен в начале сеанса работы пользователя.

То есть, с одной стороны, система позволяет варьировать интерпретаторы команд для каждого из пользователей, а, с другой стороны, в качестве интерпретатора команд можно запустить любую программу (это может быть, например, программа, проверки целостности ФС).
С точки зрения интерпретатора команд, сеанс работы пользователя представляется в виде обменов с файлом (операций чтения и записи). Работа пользователя с системой заканчивается закрытием файла – подачей EOF (end of file) (код EOF вводится путём нажатия комбинации клавиш Ctrl+D на клавиатуре). После получения от пользователя EOF интерпретатор завершает свою работу. Процесс **init** фиксирует эту ситуацию и снова запускает процесс **getty.**

Взаимодействие процессов

Параллельные процессы – процессы, выполнение которых 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 (**псевдопараллелизм**, 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.

Разделение ресурса - совместное использование ресурса ВС двумя и более параллельными процессами, когда каждый из процессов некоторое время владеет этим ресурсом (физ./ виртуал.).
Критический ресурс - разделяемый ресурс, который в каждый момент времени может быть доступен только одному из взаимодействующих процессов (внешнее устройство, некая переменная, значение которой может изменяться разными процессами).
Критическая секция (интервал) - часть программы (фактически набор операций), в рамках которой осуществляется работа с критическим ресурсом.

Задачи:

1. распределение ресурсов между процессами
2. организация корректного доступа к разделяемым ресурсам (организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов)
3. организация независимости работы параллельных процессов от порядка и интенсивности доступа этих процессов к разделяемым ресурсам

Пример: имеется некоторая общая переменная (разделяемый ресурс) **in** и два процесса, которые работают с этой переменной. Пусть в некоторый момент времени процесс **A** присвоил переменной **in** значение X. Затем в некоторый момент процесс **B** присвоил значение Y этой же переменной **in**. Далее оба процесса читают эту переменную, и в обоих случаях процессы прочтут значение Y. То есть в этом случае символ, считанный процессом **A**, был потерян, а символ, считанный процессом **B**, был выведен дважды. Результат выполнения процессов здесь зависит от того, в какой момент осуществляется переключение процессов, и от того, какой конкретно процесс будет выбран следующим для выполнения. Таким образом, требование независимости работы параллельных процессов нарушается. Такие ситуация, когда процессы конкурируют за разделяемый ресурс, называются **гонкой процессов**.

Единственный способ избежать конкуренции за неразделяемый, глобальный ресурс (уст-во ввода/вывода) и/или гонок при использовании разделяемых ресурсов - контролировать доступ к любым разделяемым ресурсам в системе. При этом необходимо организовать **взаимное исключение** — если один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ = не допускать ситуации, когда два процесса одновременно находятся в критических секциях, связанных с одним и тем же ресурсом.

Проблемы:
Блокировка - доступ к разделяемому ресурсу одного из взаимодействующих процессов не обеспечивается из-за активности других, более приоритетных процессов.

Пример: пусть в нашей модели более приоритетный запрос на обращение к ресурсу будет обработан быстрее, чем менее приоритетный. И пусть в этой модели работают два процесса, у которых приоритеты доступа к критическому ресурсу разные. Тогда, если более приоритетный будет «часто» выдавать запросы на обращение к ресурсу, может возникнуть ситуация, когда второй процесс будет «вечно» (или достаточно долго) ожидать обработки каждого своего запроса, т.е. этот менее приоритетный процесс будет блокирован.

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

Пример: пусть есть два процесса **A** и **B**, а также пара критических ресурсов. Пусть в некоторый момент времени процесс **A** вошел в критическую секцию работы с ресурсом 1. (доступ любого другого процесса к данному ресурсу будет блокирован пока процесс **A** не закончит с ним работать). Пусть также в это время процесс **B** войдет в критическую секцию ресурса 2. Пусть процесс **A**, не выходя из критической секции ресурса 1, пытается захватить ресурс 2 => процесс **A** блокируется. Аналогично с **B** => каждый из процессов ожидает освобождения недостающего ресурса, но оба ресурса никогда не будут освобождены => тупик (перезапуск системы или уничтожение обоих/одного из процессов).

Способы организации взаимного исключения

Семафоры Дейкстры — это формальная модель организации доступа, предложенная в середине 60-х гг. голландским ученым Дейкстрой. Имеется специальный целочисленный тип данных - **семафор** и определены следующие атомарные (неделимые, их выполнение не может быть прервано прерыванием) операции над переменными этого типа: опустить семафор **down(**S**)** / **P(**S**)** и поднять семафор **up(**S**)** / **V(**S**)**.

down(S) проверяет значение семафора **S** и, если он больше нуля, уменьшает его на **1,** иначе процесс блокируется и эта операция **down** считается незавершенной

up(S) увеличивает значение семафора на **1** и 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. Увеличение значения семафора и, возможно, разблокирование одного из процессов и уменьшение значения являются атомарной неделимой операцией. Выбор процесса для разблокирования никак не оговаривается.

Пример: универсам. Пусть на момент открытия имеется N свободных тележек, и посетителей пускают только с тележкой. Каждый очередной посетитель берет тележку и проходит в зал. (N+1)-ый посетитель уже не сможет войти и будет ждать свободную тележку. Если приходят еще покупатели, то они также ожидают свободной тележки.
*прибывающие в магазин покупатели не становятся в очередь, а стоят «беспорядке» и как только один из покупателей с тележкой покидаетмагазин, происходит операция **up** – появляется одна свободная тележка и ее забирает один из ожидающих. Т.е. один из заблокированных клиентов разблокировался и продолжил работу, остальные же продолжают ждать в заблокированном состоянии.
Если тележка была бы одна, то это был бы режим взаимного исключения = **двоичный семафор** (макс.знач. = 1)

- требование атомарности операций **down** и **up** накладывает ограничения на реализацию семафоров Дейкстры =>
- это низкоуровневые средства синхронизации, для корректной практической реализации которых необходимо наличие специальных атомарных семафорных машинных команд.
+попытка обойти требование аппаратной поддержки атомарности операций семафора
+высокоуровневая конструкция (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.

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 одного 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. Таким образом, 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. Поскольку организация взаимного исключения в данном случае возлагается на компилятор, количество программных ошибок, связанных с организацией взаимного исключения, сводится к минимуму
Бытовой иллюстрацией монитора может служить кабина таксофонного аппарата.
Повторим, что монитор — это языковая конструкция с централизованным управлением (в отличие от семафоров, которые не обладают централизацией). Семафоры и мониторы являются, главным образом, средствами организации работы в однопроцессорных системах либо в многопроцессорных системах с общей памятью. Для многопроцессорных систем с

распределенной памятью эти средства не очень подходят.

Для них в настоящий момент наиболее часто используется механизм **передачи сообщений**.
Это и средство организации межпроцессного взаимодействия, так и средства синхронизации. Механизм основан на двух функциональных примитивах: **send** (отправить сообщение) и **receive** (принять сообщение). Данные операции можно разделить по трем характеристикам: модель синхронизации, адресация и формат сообщения.

Синхронизация.
Операции посылки/приема сообщений могут быть **блокирующими** и **неблокирующими**.
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 происходят без блокировок.
Комбинируя различные операции send и receive, получаем 4 различных модели синхронизации.

Адресация:
Прямая – указывается конкретный адрес получателя и/или отправителя (например, когда получатель ожидает сообщения от конкретного отправителя, игнорируя сообщения других отправителей).
Косвенная - 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.

передача сообщений = средство передачи информации + средство синхронизации. Этот аппарат является базовым средством организации взаимодействия процессов в многопроцессорных системах с распределенной памятью.
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.

Классические задачи синхронизации процессов

Обедающие философы - круглый стол, философы, спагетти, по вилке между ними, для еды нужно две, в перерывах думают. Каждый из них ведет себя независимо от других. Задача организовать совместный доступ к вилкам (ресурсам).

Решения:
1.

```
#define N 5 // количество философов
void Philosopher(int i) // i – номер философа
от 0 до 4
{
 while(TRUE)
 {
  Think();// философ думает
  TakeFork(i); // взятие левой вилки
  TakeFork((i + 1) % N); // взятие правой
вилки
  Eat();// философ ест
  PutFork(i); // освобождение левой вилки
  PutFork((i + 1) % N); // освобождение правой
вилки
 }
}
```

Функция take_fork(i) описывает поведение философа по захвату вилки: он ждет, пока указанная вилка не освободится, и забирает ее.
- если все философы захотят есть в одно и то же время – тупик: каждый получит доступ к своей левой вилке и будет находиться в состоянии ожидания второй вилки до бесконечности.

2. только четырем из пяти философов доступны вилки (тогда по крайней мере один будет иметь доступ к двум вилкам) => нет тупика

```
#define N 5 // количество философов
#define LEFT (i-1)%N // номер левого соседа для i-го философа
#define RIGHT (i+1)%N // номер левого соседа для i-го философа //состояния
философов:«думает», «голоден», «кушает»
#define THINKING 0 HUNGRY 1 EATING 2
```

```
typedef int semaphore; // определяем тип СЕМАФОР
int state[N]; //массив состояний философов, иниц. нулями
semaphore mutex = 1; // семафор для доступа в критическую секцию
semaphore s[N]; //массив семафоров по одному на каждого из философов, инициализированный
нулями

void Philosopher(int i) // Процесс-философ (i = 0..N-1)
{
      while(TRUE)
      {
       Think();
       TakeForks(i); // философ берёт обе вилки или блокируется
       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]);
}
```

- опускается семафор mutex (синхронизация входа в критическую секцию)
- внутри критической секции состояние философа меняется на HUNGRY
- вызывается функция Test: если i-ый философ голоден, а его соседи в данный момент не едят (правая и левая вилки свободны)
    - философ начинает прием пищи (состояние EATING)
    - его семафор поднимается (изначально 0)
    - возврат в TakeForks
    - выход из критической секции - поднимается семафор mutex
    - опускается семафор этого философа.
- Если внутри функции Test философу удалось начать прием пищи, то семафор поднят, и операция down обнулит его, не блокируясь. Если же функция Test не изменит состояние философа, а также не поднимет его семафор, то операция down (в функции TakeForks) в этой точке заблокируется до тех пор, пока оба соседа не освободят вилки

```
//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);
}
```

- опускается семафора mutex (происходит вход в критическую секцию)
- статус философа меняется на THINKING
- проверка соседей: если любой из них был заблокирован лишь из-за того, что наш i-ый философ забрал его вилку, то мы его разблокируем, и он начинает прием пищи.
- подъем семафора mutex (выход из критической секции)

```
void Test(int i)//проверка, можно ли взять вилки - проверяется
состояние соседей данного философа
{
      if(state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING)
      {
            state[i] = EATING;
            up(&s[i]);
       }
}
```

механизм взаимоисключающего нахождения внутри критической секции (за счет семафора mutex) гарантирует, что не возникнет ситуация, когда функция Test в процессах двух соседних философов разрешит каждому из них начать прием пищи. Без этого механизма возможно, что процессы на процессоре сменяться после первой команды if-блока первого философа. Второй процес проходит if-блок с положительным результатом и переходит к первой инструкции блока. Дальнейшая работа будет некорректной.

Задача «читателей и писателей». 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.
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.
Пусть читатели наиболее приоритетны (писатель будет ожидать момента, когда все желающие процессы-читатели окончат свои действия в системе и покинут ее)

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

```
void Reader(void)// читатель
{
while(true)
{
 down(&mutex); // получить эксклюзивный доступ к «rc»
 rc++; // еще одним читателем больше

//Первый читатель блокирует 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--; // теперь одним читателем меньше

последний читатель разблокирует экс-доступ к базе
if(rc == 0) up(&db);

 up(&mutex); //освободить разделяемый ресурс rc
 UseDataRead(); //некритическая секция
}
}
```

1. вход в критическую секцию (опускается семафор mutex),
2. увеличение счетчика читателей в хранилище на 1
3. проверка, является ли он первым читателем (он единственный клиент в хранилище на данный ммомент)
   a. да и семафор db поднят => опускает его (препятствует писателям войти в систему)
   b. да, но семафор *db* уже был опущен (в хранилище присутствует писатель) => этот первый читатель заблокируется, ожидая выхода писателя из системы. (блокировка происходит внутри критической секции, поэтому остальные читатели будут блокироваться на опускании семафора mutex)
4. выход из критической секции (поднимается семафор mutex)
5. чтение информации из хранилища
6. обратные действия по выходу из хранилища, которые также происходят внутри критической секции
7. уменьшение число читателей в хранилище
8. проверка, является ли он последним читателем в библиотеке: да => поднятие семафора db, разрешая работу писателям (которые к этому моменту могли быть заблокированы на входе)
9. обработка полученных данных из хранилища
10. цикл

```
void Writer(void) //писатель
{
while(TRUE)
{
ThinkUpData(); // некритическая секция
down(&db); //получить эксклюзивный доступ к данным
WriteDataBase(); // записать данные
up(&db); // отдать эксклюзивный доступ
}
}
```

1. подготовка данных для сохранения
2. попытка входа в хранилище (опускается семафор db)
   a. если в хранилище кто-то есть, то он будет ожидать, пока последний клиент (независимо от того, читатель это или писатель) не покинет его
3. корректировка данных в хранилище
4. выход из хранилища (подъем семафора db)

если хотя бы один читатель находится внутри системы:

- любой следующий читатель беспрепятственно в нее попадет
- любой писатель же будет ожидать, когда все посетители покинут хранилище

реализована стратегия приоритетности читателя перед писателем

* можно модифицировать алгоритм так, чтобы в случае, если имеется хотя бы один ожидающий писатель, новые читатели не получали доступа к ресурсу, а ожидали, когда писатель обновит данные.
- снижается производительность читателей (ждут в тот момент, когда ресурс не занят в эксклюзивном режиме

Задача о «спящем парикмахере».
Парикмахерская, один парикмахер, одно кресло для стрижки и несколько кресел в приемной для ожидающих.
нет посетителей - парикмахер засыпает (первый посетитель будит)
новые посетители:

- ждут своей очереди
- покидают парикмахерскую, если в приемной нет свободного кресла для ожидания (стратегия с ограничением на длину очереди клиентов и отказами).

.

Семафоры:

1. *customers* — подсчитывает количество посетителей, ожидающих в очереди
2. *barbers* — статус парикмахера (0 - спит, 1 -работает) – но можно и расширить до N парикмахеров, тогда barbers – кол-во свободных парикмахеров
3. *mutex* — синхронизация доступа к разделяемой переменной *waiting*.

Переменная *waiting*, как и семафор *customers*, содержит количество посетителей, ожидающих в очереди (нужна, чтобы знать, имеется ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется) - является разделяемым ресурсом, и доступ к ней охраняется семафором *mutex*.

```
#define CHAIRS 5 //количество стульев в комнате ожидания
typedef int semaphore; //переопределение типа СЕМАФОР
semaphore customers = 0; //наличие ждущих посетителей
semaphore barbers = 0; //парикмахер спит (0) или работает (1)
semaphore mutex = 1; семафор доступа в критическую секцию - контроль за доступом к
переменной waiting
int waiting = 0; количество ожидающих посетителей
```

```
void Barber(void) //парикмахер
{
while(TRUE)
{
down(&customers); если customers == 0
(посетителей нет) - блок до появления посетителя
down(&mutex); // доступ к waiting
waiting--; уменьшаем кол-во ожидающих клиентов
up(&barbers); // парикмахер готов к работе
up(&mutex); // освобождаем ресурс waiting
CutHair(); // процесс стрижки
}
}
```

1. опускает семафор *customers* (уменьшает тем самым количество ожидающих посетителей на 1)
2. если в комнате ожидания никого нет, то он «засыпает» в своем кресле, пока не появится клиент, который его разбудит.
3. входит в критическую секцию
4. уменьшает счетчик ожидающих клиентов
5. поднимает семафор *barbers* (сигнал клиенту о своей готовности его обслужить)
6. выходит из критической секции
7. стрижет волосы посетителю.

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

1. входит в критическую секцию
2. проверяет, есть ли свободные места в зале ожидания
    a. нет – уходит: покидает критическую секцию, поднимая семафор *mutex*
    b. да - увеличивает счетчик ожидающих процессов и поднимает семафор customers.
3. Если же этот посетитель является единственным в данный момент клиентом брадобрея, то он этим действием разбудит брадобрея.
4. выход из критической секции
5. «захват» брадобрея (опуская семафор *barbers*)
6. если семафор опущен, то клиент будет дожидаться, когда брадобрей его поднимет, известив тем самым, что готов к работе

Реализация межпроцессного взаимодействия в ОС Unix

полновесные процессы - обрабатываемой в системе программы, обладающей эксклюзивными правами на оперативную память, а также правами на некоторые дополнительные ресурсы.
группы взаимодействия:

1. взаимодействие процессов, функционирующих под управлением одной ОС на одной локальной ЭВМ:

взаимодействие родственных процессов (процессов, связанных некоторой иерархией родства):

1. ОС позволяет детям наследовать некоторые характеристики отцов и за счет этого можно реализовать то самое именование – неявное.
2. взаимодействие по цепочке предок–потомок (известно, кто есть кто) => предок может к своему потомку посредством явного именования. Имя - идентификатор процесса (PID). А потомок, зная имя предка, может также к нему обратиться.

Варианты**:**

1. **Неименованный канал** – симметричная модель (равные права на ресурс благодаря наследованию). Неименованный канал - это некоторый ресурс, который наследуют дети. (этот механизм может быть использован для организации взаимодействия произвольных родственников (между «сыном» и его «племянником»).
2. **Трассировка**
3. **«главный–подчиненный»** - несимметричная модель: среди взаимодействующих процессов можно выделить процессы, имеющие больше полномочий, чем у остальных. А у главного процесса (или процессов) есть целый спектр механизмов управления подчиненными.

взаимодействие произвольных процессов в рамках одной локальной машины.

1. **прямое именование,** когда процессы для указания своих партнеров по взаимодействию используют их уникальные имена (например, PID привязывается к некоторому новому уникальному имени, и обращение при взаимодействии происходит с использованием системы этих новых имен).
2. взаимодействие посредством **общего ресурса** (но тогда проблема именования этих общих ресурсов)

Варианты *преобладают средства симметричного взаимодействия (процессы имеют равные права).

1. **Именованные каналы** — это общий ресурс взаимодействующих процессов, посредством которого осуществляется взаимодействие (не обязательно знать имена процессов-партнеров по взаимодействию)
2. Передача **сигналов** — это средство воздействия одного процесса на другой в общем случае (в частности, одним из процессов может выступать процесс ОС). При этом используются непосредственные имена процессов.
3. Система **IPC** (Inter-Process Communication) предоставляет взаимодействующим процессам общие разделяемые ресурсы (**общая память**, **массив семафоров** и **очередь сообщений**), (альтернатива именованных каналов)
4. Аппарат **сокетов** — унифицированное средство организации взаимодействия. В аппарате сокетов именование осуществляется посредством связывания конкретного процесса (его PID) с конкретным сокетом, через который и происходит взаимодействие.

2. взаимодействие в пределах сети

Решение проблемы именования: пусть у нас есть две машины, имеющие сетевые имена A и B. 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»: (A–P1) и (B–P2).

Однако! В рамках сети могут взаимодействовать машины, находящиеся под управлением ОС различного типа (т.е. в сети могут оказаться Windows-машины, FreeBSD-машины, Macintosh-машины и пр.) => система именования должна быть стандартизованным (унифицированным) средством.

Решения:

1. **Аппарат сокетов** – механизм уровня протоколов взаимодействия. Взаимодействие осуществляется между сокетами (можно организовать общение одного сокета со многими, можно установить связь один–к–одному и т.д.). Именование сокетов также зависит от задачи: иногда это точные имена взаимодействующих сокетов, а иногда произвольными (клиент–серверная архитектура - имена клиентских сокетов могут быть любыми).
2. **Система MPI** (**интерфейс передачи сообщений**). Система MPI может работать на локальной машине, в многопроцессорных системах с распределенной памятью (может работать в кластерных системах), в сети в целом (в частности, в т.н. GRID-системах)

Сигналы - средство уведомления процесса о наступлении некоторого события в системе. Аналогия аппарата прерываний (также есть уведомление системы о том, что в ней произошло некоторое событие + приход сигнала в процесс, как и прерывания, вызывает в нем определенную последовательность действий)

Инициатор отправки сигнала процессу:
1. **Сама ОС**. Она уведомляет о наступлении некоторых строго предопределенных ситуаций (завершение порожденного процесса, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал), при этом каждой такой ситуации сопоставлен свой сигнал.

Пример: в ходе выполнения некоторого процесса произошло деление на ноль, вследствие чего в системе происходит прерывание, управление передается операционной системе. ОС «видит», что это прерывание «деление на ноль», и отправляет сигнал процессу, в теле которого произошла данная ошибка. Дальше процесс реагирует на получение сигнала.

2. **другой процесс.**

Пример: ОС Unix запустил некоторый процесс, который в некоторый момент времени зацикливается. Чтобы снять этот процесс со счета, пользователь может послать ему сигнал об уничтожении (например, нажав на клавиатуре комбинацию клавиш Ctrl+C, а это есть команда интерпретатору команд послать код сигнала SIGINT). В данном случае процесс интерпретатора команд пошлет сигнал пользовательскому процессу.

- асинхронного взаимодействия, момент прихода сигнала процессу заранее неизвестен.
- процесс может предвидеть возможность получения того или иного сигнала и установить определенную реакцию на его приход (аналог прерываний)

Unix-системы имеют фиксированный набор сигналов в файле **<signal.h>** : набор пар «имя сигнала — его целочисленное значение»
2 - SIGINT /*прерывание*/
3 - SIGQUIT /*аварийный выход*/
9 - SIGKILL /*уничтожение процесса*/
14 - SIGALRM /*прерывание от таймера*/
18 – SIGCHLD /* процесс-потомок завершился */
SIGUSR1 /*используется для синхронизации взаимодействующих процессов. Обычно в ОС есть сигналы, которые не ассоциированы с событиями, происходящими в системе. Эти сигналы процессы могут использовать по своему усмотрению (количество пользовательских сигналов зависит от конкретной реализации).

Реакция процесса на сигнал:
1. **обработка сигнала по умолчанию**.
   Чаще всего это завершение процесса. Системный код завершения - номер пришедшего сигнала.
2. **перехватывать обработку пришедшего сигнала**.
   Вызывается функция, принадлежащая телу процесса, которая была специальным образом зарегистрирована в системе как **обработчик сигнала**. Важно, часть реализованных в ОС сигналов перехватывать нельзя (**SIGKILL** (код 9) - безусловное уничтожение процесса).
3. **Игнорировать**. Опять же, часть сигналов игнорировать нельзя (**SIGKILL**).

Несколько тонкостей:
- Если пришло несколько сигналов, то порядок их не определен.
- Если же обработки ждут несколько экземпляров одного и того же сигнала, то ответ на вопрос, сколько экземпляров будет доставлено в процесс – все или один – зависит от конкретной реализации ОС.
- Если сигнал приходит в момент выполнения системного вызова, то в зависимости от версии UNIX:
  - обработка может быть отложена до завершения системного вызова
  - системный вызов автоматически перезапускается после его прерывания сигналом
  - системный вызов вернет –1, а в переменной **errno** будет установлено значение **EINTR**. (будем считать так)

Отправка сигнала в ОС Unix:

```
#include <sys/types.h>, <SIGNAL.H>
INT KILL (PID_T PID, INT SIG);
```

кому посылается (можно себе/всей группе кроме 0 и 1 – 0), номер сигнала (0 – проверка корректности kill() в частности, существование процесса с идентификатором pid)
*Если отправитель не обладает правами привилегированного пользователя, то он может отправить сигнал только тем процессам, у которых такой же реальный или эффективный идентификатор владельца процесса.
При успешном завершении системный вызов **`kill()`** возвращает 0, иначе возвращается -1.

Установка реакции на сигнал:

```
<SIGNAL.H>
void (*signal  ( int sig, void (*disp) (int))) (int);
```

номер сигнала, реакцию на приход которого надо установить / новая реакция на приход указанного сигнала (либо определенная пользователем функция-обработчик сигнала, либо одна из констант: **SIGDFL** и **SIGIGN** - по умолчанию/игнорирование)
* при успешном завершении возвращается указатель на предыдущий обработчик данного сигнала (для восстановления прежней реакции на сигнал).
** определенная пользователем функция-обработчик сигнала должна принимать один целочисленный аргумент (в нем будет передан номер обрабатываемого сигнала), и не возвращать никаких значений.

Если мы успешно установили в качестве обработчика сигнала свою функцию, то при возникновении сигнала:
1.выполнение процесса прерывается
2.фиксируется точка возврата
3.управление в процессе передается данной функции, при этом в качестве фактического целочисленного параметра передается номер пришедшего сигнала (тем самым возможно использование одной функции в качестве обработчика нескольких сигналов).
4. управление передается в точку возврата при выходе из функции-обработчика
5.процесс продолжает свою работу.

- ресурсоемкий, ибо отправка сигнала – это системный вызов, а доставка сигнала - прерывание выполнения процесса-получателя
- вызов функции-обработчика и возврат требует операций со стеком
- особенность в ранних версиях UNIX: если процесс желает многократно обрабатывать сигнал своим собственным обработчиком, он должен каждый раз при обработке сигнала заново устанавливать реакцию на него, так как каждый раз при получении сигнала его диспозиция (т.е. действие при получении сигнала) сбрасывается на действие по умолчанию.

Надежные сигналы.

(UNIX System V.2 и раньше):
Если между моментом вызова пользовательского обработчика и определением следующей реакцию на этот сигнал, снова пришел этот сигнал. Тогда он не будет перехвачен, так как на момент его прихода действует реакция по умолчанию.

UNIX (BSD UNIX 4.2 и System V.4) (новые версии):
+ ядро не меняет диспозицию перехватываемого сигнала, тем самым появляется гарантия перехвата всех экземпляров сигнала
+ ядро блокирует доставку других экземпляров того же сигнала в процесс до тех пор, пока функция-обработчик не завершит свое выполнение. Это избавляет от нежелательных эффектов при рекурсивном вызове обработчика для множества экземпляров одного и того же сигнала.

+ есть возможность на время блокировать доставку того или иного вида сигналов в процесс
Отличие блокировки от игнорирования в том, что пришедшие экземпляры сигнала не будут потеряны, а произойдет лишь откладывание их обработки на тот период времени, пока процесс не разблокирует данный сигнал. Таким образом процесс может оградить себя от прерывания сигналом на тот период, когда он выполняет какие-либо критические операции.
+ есть возможность получить информацию о причине отправления сигнала.

Сигнальная маска описывает, какие сигналы из посылаемых процессу блокируются. Процесс наследует свою сигнальную маску от родителя при порождении, но может изменить ее в процессе выполнения.

Сигнальная маска описывается битовым полем типа **sigset_t**. Для управления сигнальной маской процесса служит системный вызов:

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *old_set);
```

how влияют на характер изменения маски сигналов:

- **SIG_BLOCK** – к текущей маске добавляются сигналы, указанные в наборе **set**;
- **SIG_UNBLOCK** – из текущей маски удаляются сигналы, указанные в наборе **set**;
- **SIG_SETMASK** – текущая маска заменяется на набор **set**.

Если в качестве аргумента **set** передается NULL-указатель, то сигнальная маска не изменяется, значение аргумента **how** при этом игнорируется.

В последнем аргументе возвращается прежнее значение сигнальной маски до изменения ее вызовом **sigprocmask()** или NULL-указатель, если нет необходимость знать прежнее значение.

Тонкости:
1.Если один/несколько заблокированных сигналов будут разблокированы через **sigprocmask()**, то для их обработки будет использована диспозиция сигналов, действовавшая до вызова **sigprocmask()**.
2.Если за время блокирования процессу пришло несколько экземпляров одного и того же сигнала, то количество сигналов, которые будут доставлены (все или один) зависит от реализации конкретной ОС.

Вспомогательные функции для формирования битового поле **sigset_t**:
int sigemptyset(sigset_t *set); – инициализация битового набора - очищение всех битов
int sigfillset(sigset_t *set); наоборот устанавливает все биты в наборе
int sigaddset *или* sigdelset (sigset_t *set, int signo); – добавить или удалить флаг, соответствующий сигналу, в наборе, второй арг. – номер сигнала
int sigismember(sigset_t *set, int signo); – проверка, установлен ли в наборе флаг, соответствующий сигналу, указанному в качестве параметра (возвращает 1, если в маске **set** установлен флаг, соответствующий сигналу **signo**, иначе 0)
int sigpending(sigset_t *set); – какие из заблокированных сигналов ожидают доставки (через аргумент возвращается набор сигналов, ожидающих доставки)

sigaction() : аналогичная функции **signal()**, позволяющая установить обработку сигнала, узнать ее текущее значение, приостановить получение сигналов:
int sigaction(int sig, const struct sigaction *act, struct *oact)
1.номер сигнала
2.структура, описывающая новую реакцию на сигнал
3.структура, через которую возвращается прежний метод обработки сигнала, или NULL-указатель

Поля структуры **sigaction**:

```
void (*sa_handler)(int),
void (sa_sigaction)(int, siginfo_t*, void*), sigset_t sa_mask,
int sa_flags
```

sa_handler - функция-обработчик сигнала, либо константы **SIG_IGN** или **SIG_DFL** (игнорирование/обработка по умолчанию)
sa_mask - набор сигналов, которые будут добавлены к маске сигналов на время работы функции обработчика. Сигнал, для которого устанавливается функция-обработчик, также будет заблокирован на время ее работы. При возврате из функции-обработчика маска сигналов возвращается в первоначальное состояние.
sa_flags – флаги, модифицирующие доставку сигнала (например, **SIGINFO** - если он установлен, то при получении этого сигнала будет вызван обработчик **sa_sigaction**, ему помимо номера сигнала также будет передана дополнительная информация о причинах получения сигнала и указатель на контекст процесса)

Неименованные каналы или ***программные канал***ы - это область памяти (на ВЗУ), которая управляется ОС и осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы. То есть эта область памяти является разделяемым ресурсом. Для доступа к неименованному каналу система ассоциирует с ним два файловых дескриптора. Один из них предназначен для чтения информации из канала (как файл, открытый только на чтение). Другой дескриптор предназначен для записи информации в канал (как файл, открытый только на запись).
Тонкости:

- Поступление информации по FIFO, т.е. информация, записанная в канал первой и будет прочитана из канала первой. Таким образом для данных дескрипторов неприменимы системные вызовы перемещения файлового указателя.
- Предельный размер канала определяется параметрами настройки ОС.
- В общем случае нужен для взаимодействия родственных каналов, но иногда встречаются вырожденные случаи использования неименованного канала в рамках одного процесса - фактически осуществляется посылка данных самому себе.
- В общем случае возможна многонаправленная работа процессов с каналом (с одним и тем же каналом взаимодействуют два и более процесса, и каждый из них пишет информацию или читает ее из канала). Но традиционной схемой организации работы с каналом является однонаправленная организация, когда канал связывает два (в большинстве случаев) или несколько взаимодействующих процессов, каждый из которых может либо читать, либо писать в канал.

Для создания неименованного канала служит системный вызов *pipe()*.

```
#include <unistd.h>
int pipe(int *fd)
```

- указатель на массив *fd* из двух целочисленных элементов.
Если системный вызов *pipe()* прорабатывает успешно, то он возвращает код ответа, равный нулю, а массив будет содержать два открытых файловых дескриптора. Соответственно, в fd[0] будет содержаться дескриптор чтения из канала, а в fd[1] — дескриптор записи в канал. После этого с дескрипторами можно работать как с файлами (по стратегии FIFO - любые операции работы с файлами кроме перемещений файлового указателя).

Различия между обычным файлом и каналом:
1. В отличие от файла, к неименованному каналу невозможен доступ по имени, т.е. единственная возможность использовать канал – это те файловые дескрипторы, которые с ним ассоциированы
2. В отличие от файлов неименованный канал существует в системе, до тех пор, пока существуют процессы, его использующие. То есть для существования канала необходим процесс, который его создал либо получил в наследство. Отметим, что канал может быть доступен как для процесса, который его создал, так и для процессов, которые унаследовали открытые файловые дескрипторы, ассоциированные с этим каналом. За счёт этой возможности наследования, неименованный канал превращается в средство взаимодействия родственных процессов.
3. Нельзя использовать lseek()

При чтении из канала:
- Если из канала читается порция данных меньшая, чем находящаяся в канале, то эта порция считывается по стратегии FIFO, а оставшаяся порция непрочитанных данных остается в канале.
- Если делается попытка прочитать больше данных, чем имеется в канале, и при этом в системе имеются открытые дескрипторы записи, ассоциированные с этим каналом, то будет прочитано (т.е. изъято из канала) доступное количество данных, после чего читающий процесс блокируется до тех пор, пока в канале не появится достаточное количество данных для завершения операции чтения.
- Процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова **`fcntl()`**. В неблокирующем режиме будет так же прочитано доступное количество данных, но управление будет сразу возвращено процессу.
- Отметим, что блокировка происходит лишь при условии, что есть хотя бы один открытый дескриптор записи в канал. Если закрывается последний дескриптор записи в данный канал, то в канал помещается код конца файла EOF. В этом случае процесс, осуществляющий чтение, может выбрать из канала все оставшиеся данные и признак конца файла, благодаря которому блокирования при чтении в этом случае не происходит. Соответственно, если заблокированы два и более процесса на чтение, то порядок разблокировки определяется конкретной реализацией.

При записи в канал:
- Если процесс пытается записать большее число байтов, чем доступное свободное пространство канала (но не превышающее предельный размер канала), то записывается возможное количество данных, после чего процесс, осуществляющий запись, блокируется до тех пор, пока в канале не появится достаточное количество места для завершения операции записи.
- Процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова **`fcntl()`**. В неблокирующем режиме будет записано то же возможное количество данных, но управление будет сразу возвращено процессу.
- Если же процесс пытается записать в канал порцию данных, превышающую предельный размер канала, то будет записано доступное количество данных, после чего процесс заблокируется до появления в канале свободного места любого размера (пусть даже и всего 1 байт), затем процесс разблокируется, вновь производит запись на доступное место в канале, и если данные для записи еще не исчерпаны, вновь блокируется до появления свободного места и т.д., пока не будут записаны все данные, после чего происходит возврат из вызова **`write()`.**
- Если процесс пытается осуществить запись в канал, с которым не ассоциирован ни один дескриптор чтения, то он получает сигнал **`SIGPIPE`** (тем самым ОС уведомляет его о недопустимости такой операции)**.**
- В стандартной ситуации (при отсутствии переполнения) система гарантирует атомарность операции записи, т. е. при одновременной записи нескольких процессов в канал их данные не перемешиваются.

Недостаток: так как доступ к неименованным каналам возможен только через дескрипторы, то взаимодействие процессов возможно только если они все обладают доступом к этим дескрипторам. Т.е при рождении процесса их необходимо передавать по наследству. – проблема именования.

Решение: **именованные каналы** (те же FIFO-файлы, хранящиеся в ВЗУ). К такому каналу может подключиться любой процесс в любое время (и после создания канала), зная имя канала.

Создание файлов FIFO в различных реализациях используются разные системные вызовы, одним из которых может являться *mkfifo():*

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

1. имя создаваемого канала
2. устанавливаются:
    a. режимы открытия
    b. права доступа к каналу для владельца/группы/прочих пользователей
    c. флаг, указывающий на то, что создаваемый объект является именно FIFO-файлом (в разных версиях ОС он может иметь разное символьное обозначение – **`S_IFIFO`** или **`I_FIFO`**).

После создания именованного канала любой процесс может установить с ним связь посредством системного вызова ***open().***

1. если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись;
2. если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение;
3. процесс может избежать такого блокирования, указав в вызове *open()* специальный флаг (в разных версиях ОС он может иметь разное символьное обозначение - O_NONBLOCK или O_NDELAY). Тогда *open()* сразу же вернет управление процессу.

Нелокальные переходы

Дополнительные возможности по организации управления ходом процесса в UNIX, а именно возможность передачи управления в точку, расположенную вне данной функции.

Оператор **`goto`** позволяет осуществлять безусловный переход только внутри одной функции, так как необходимо сохранять целостность стека: в момент входа в функцию в стеке отводится место, называемое стековым кадром (фрейм), где записываются адрес возврата, фактические параметры, отводится место под автоматические переменные. Фрейм освобождается при выходе из функции. Соответственно, если при выполнении безусловного перехода процесс минует тот фрагмент кода, где происходит освобождение фрейма, и управление перейдет в другую часть программы, то фактическое состояние стека не будет соответствовать текущему участку кода, и тем самым стек подвергнется разрушению.

Однако в случае возникновения ошибки в рекурсивной функции, после обработки ошибки имеет смысл перейти в основную функцию, которая может находиться на несколько уровней вложенности выше текущей. (Нельзя ни **`return`**, ни **`goto`**). Возможность передавать управление в точку, находящуюся в одной из вызывающих функций,

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

1. регистрация некоторой точки кода, которая в дальнейшем будет использоваться в качестве пункта назначения для нелокального перехода
2. сохранение параметров текущей точки кода (значения счетчика адреса, позиции стека, регистров процессора и реакций на сигналы) в структуре типа **`jmp_buf`**, которая передается вызову **`setjmp()`** в качестве параметра.
3. При этом вызов **`setjmp()`** возвращает 0.

```
void longjmp(jmp_buf env, int val);
```

4. переход в одну из ранее зарегистрированных с помощью **`setjmp()`** конечных точек через ее структуру (с зафиксированными атрибутами нужной нам точки назначения).
5. После осуществления вызова **`longjmp()`** процесс продолжит выполнение с зафиксированной точки кода, т.е. с того места, где происходит возврат из функции **`setjmp()`**, но в отличие от первого обращения к **`setjmp()`**, возвращающим значением **`setjmp()`** станет не 0, а значение параметра **`val`** в вызове **`longjmp()`**, который произвел переход.

Тонкости:

- если необходимо определить в программе несколько точек назначения для нелокальных переходов, каждая из них должна быть зарегистрирована в своей структуре типа **`jmp_buf`**.
- на одну и ту же точку назначения можно переходить из разных мест программы, при этом, чтобы различить, из какой точки был произведен нелокальный переход, следует указывать при переходах разные значения параметра

val. В любом случае, при вызове **longjmp()** значение параметра **val** не должно быть нулевым (даже если оно есть 0, то возвращаемое значение **setjmp()** будет установлено в 1).

- переход должен производиться только на такие точки, которые находятся в коде одной из вызывающих функций для той функции, откуда осуществляется переход (в том числе, переход может быть произведен из функции обработчика сигнала). При этом в момент перехода все содержимое стека, используемое текущей функцией и всеми вызывающими, вплоть до необходимой, освобождается.

Трассировка процессов – модель межпроцессного взаимодействия «главный-подчиненный»

Достаточно часто при организации многопроцессной работы необходимо наличие возможности, когда один процесс является главным по отношению к другим процессам (трассировка процессов). В частности, это необходимо для организации средств отладки, когда есть процесс-отладчик и отлаживаемый процесс. Для механизма отладки полезно, чтобы отладчик мог в произвольные моменты времени останавливать отлаживаемый процесс и, когда отлаживаемый процесс остановлен, осуществлять действия по его отладке: просматривать содержимое тела процесса, при необходимости корректировать тело процесса и т.д. Также является полезным возможность установки контрольных точек в отлаживаемом процессе. Очевидно, что полномочия процесса-отладчика по отношению к отлаживаемому процессу являются полномочиями **главного**, т.е. отладчик может осуществлять управление, в то время как отлаживаемый процесс может лишь **подчиняться**.

В UNIX трассировка возможна только между родственными процессами: процесс-родитель может вести трассировку только непосредственно порожденных им потомков, при этом трассировка начинается только после того, как процесс-потомок дает разрешение на это.

Далее схема взаимодействия процессов путем трассировки такова (трассируемый процесс – танцор, отладчик – хореограф)

- если танцор получает сигнал или делает **exec()**, то его выполнение (танец) приостанавливается
- хореограф выполняет свою работу : анализирует и изменяет данные в адресном пространстве процесса и в пользовательской составляющей его контекста
- если хореограф делает **wait()** то управление сразу передается процессу и тот продолжает выполняться

Для организации взаимодействия «главный–подчиненный» ОС Unix предоставляет системный вызов *ptrace()*.

```
#include <sys/ptrace.h>
int ptrace(int cmd, int pid, int addr, int data);
```

(код выполняемой команды / идентификатор танцора / некоторый адрес в адресном пространстве танцора / слово информации)

1. с помощью этого вызова танцор может разрешить хореографу проводить свою трассировку: для этого в качестве параметра ***cmd*** необходимо указать команду **PTRACE_TRACEME.**
2. с помощью этого же системного вызова хореограф может манипулировать танцором: для этого используются остальные значения параметра ***cmd***.

ptrace() позволяет:

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). Следует отметить, что 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).

Рассмотрим основные коды операций (параметр *cmd*) системного вызова *ptrace()*:

cmd = PTRACE_TRACEME
танцор вызывает хореографа в самом начале работы, все остальные обращения к вызову **ptrace()** осуществляет хореограф.

cmd = PTRACE_PEEKDATA
чтение слова из адресного пространства танцора по адресу **addr**, возвращается его значение.

cmd = PTRACE_PEEKUSER — чтение слова из контекста танцора **(**доступ к пользовательской составляющей контекста танцора, сгруппированной в некоторую структуру в заголовочном файле **<sys/user.h>**). **addr** указывает смещение относительно начала этой структуры. В ней хранятся регистры, текущее состояние танцора, счетчик адреса и так далее. Возвращается значение считанного слова.

cmd = PTRACE_POKEDATA — запись данных, размещенных в **data**, по адресу **addr** в адресном пространстве танцора.

cmd = PTRACE_POKEUSER — запись слова из **data** в контекст танцора со смещением **addr**. Так можно, например, изменить счетчик адреса танцора, и при последующем возобновлении танцора его выполнение начнется с инструкции, находящейся по заданному адресу.

cmd = PTRACE_GETREGS, PTRACE_GETFREGS — чтение регистров общего назначения (в т.ч. с плавающей точкой) танцора и запись их значения по адресу **data**.

cmd = PTRACE_SETREGS, PTRACE_SETFREGS — запись в регистры общего назначения (в т.ч. с плавающей точкой) танцора данных, расположенных по адресу **data** в отладчике.

cmd = PTRACE_CONT — возобновление выполнения танцора**.** Танцор будет выполняться до тех пор, пока не получит какой-либо сигнал, либо пока не завершится.

cmd = PTRACE_SYSCALL, PTRACE_SINGLESTEP —аналогично **PTRACE_CONT**, возобновляет выполнение танцора, но при этом произойдет ее остановка после того, как выполнится одна инструкция. Используя **PTRACE_SINGLESTEP**, можно организовать пошаговую отладку. С помощью команды **PTRACE_SYSCALL** возобновляется выполнение танцора вплоть до ближайшего входа или выхода из системного вызова. Идея использования **PTRACE_SYSCALL** в том, чтобы иметь возможность контролировать значения аргументов, переданных в системный вызов танцором, и возвращаемое значение, переданное ему из системного вызова.

cmd = PTRACE_KILL — завершение выполнения танцора.

Отладчики бывают:

1. **адресно-кодовыми** - оперируют адресами тела отлаживаемого процесса
2. **символьные** - оперируют объектами языка (переменными и операторами)

Механизм организации контрольной точки в адресно-кодовом отладчике: при приходе управления в эту точку программы процесс всегда приостанавливался, и управление передавалось процессу отладчику.
В отладчике имеется таблица контрольных точек, в каждой строке которой присутствует адрес некоторой контрольной точки и оригинальный код (содержимое) танцора, взятый по данному адресу.

Для установки контрольной точки по адресу A необходимо:

1. тем или иным способом остановить танцора (либо при входе, либо отладчик шлет сигнал)
2. отладчик читает из сегмента кода машинное слово по адресу A (посредством обращения к системному вызову *ptrace()*), которое он записывает в соответствующую строку таблицы контрольных точек, тем самым, сохраняя оригинальное содержимое тела танцора.
3. по адресу A в сегмент кода записывается команда, которая вызывает прерывание, и, соответственно, приход предопределенного события (сигнала) (например, деление на ноль).

Итак, танцор выполняется до тех пор, пока управление, наконец, не передается на машинное слово по адресу A (деление на ноль):

1. происходит прерывание.
2. система передает сигнал.
3. отладчик через системный вызов *wait()* получает код возврата и «понимает», что в танцоре случилось деление на ноль.
4. отладчик посредством системного вызова *ptrace()* читает адрес останова в контексте танцора.
5. анализируется причина останова:
    a. это действительно деление на ноль как ошибка
    b. это деление на ноль как контрольная точка.

    Для определения отладчик обращается к таблице контрольных точек и ищет там адрес останова подчиненного процесса. Если он его нашел, значит, танцор пришел на контрольную точку (иначе деление на ноль отрабатывается как ошибка).
6. находясь в контрольной точке, отладчик производит различные манипуляции с трассируемым процессом (читает данные, устанавливает новые контрольные точки).

Корректное продолжение танцора:

1. в сегмент кода по адресу A записывается оригинальное машинное слово (которое берётся из таблицы контрольных точек).
2. отладчик через *ptrace()* включает пошаговый режим выполнения танцора.
3. танцор выполняет одну команду (которую только что записали по адресу A) и останавливается.
4. Затем отладчик выключает режим пошаговой отладки и запускает процесс с текущей точки.

Организация контрольных точек в символьных отладчиках:
Используется информация, собранная на этапах компиляции и редактирования связей. Если с компилятором связан символьный отладчик, то компилятор формирует некоторую специализированную базу данных, в которой находится информация по всем именам, используемым в программе. Для каждого имени определены диапазоны видимости и существования этого имени, его тип (статическая переменная, автоматическая переменная, регистровая переменная и т.п.). А также данная база содержит информацию обо всех операторах (диапазон начала и конца оператора и т.п.).

Чтобы просмотреть содержимое некоторой переменной *v* нужно:
1. остановить танцора
2. по адресу останова можно определить, в какой точке программы он произошел.
3. на основе информации об этой точке программы можно (обратившись к содержимому базы данных) определить то пространство имен, которое доступно из этой точки.

Если *v* оказалась доступна, происходит обращение к базе данных и определяется тип данной переменной:
1. **статическая** - в соответствующей записи будет адрес, по которому размещена данная переменная (этот адрес станет известным на этапе редактирования связей). И с помощью *ptrace()* отладчик берет содержимое по этому адресу. Также из базы данных берется информация о типе переменной (char, float, int и т.п.), и на основе этой информации пользователю соответствующим образом отображается значение указанной переменной *v*.
2. **автоматическая** или **формальный параметр** (обычно размещаются в блоке на фиксированном смещении относительно вершины стека, т.е. для этих переменных в качестве адреса фиксируется смещение от вершины стека). Для чтения необходимо:
    a. обратиться к контексту процесса
    b. считать значение регистра-указателя стека (адрес вершины стека)
    c. к содержимому этого регистра прибавить смещение
    d. по получившемуся адресу обратиться к соответствующему сегменту
3. **регистровая**
    a. происходит обращение к базе данных
    b. считывается номер регистра
    c. происходит обращение к сегменту кода
    d. считывается содержимое нужного регистра

*При записи значений в переменные происходит та же последовательность действий.
**Для установки контрольной точки на оператор:
    1. через базу данных определяются диапазон адресов оператора и начальный адрес
    2. производятся действия по описанной выше схеме

Система межпроцессного взаимодействия IPC (Inter-Process Communication)

- сигналы несут в себе слишком мало информации и не могут использоваться для передачи сколь либо значительных объемов данных
- неименованный канал должен быть создан до того, как рождается процесс, который будет осуществлять коммуникацию
- именованный канал, хотя и лишен этого недостатка, требует одновременной работы с ним обоих процессов, участвующих в коммуникации

Решение: система IPC (IPC System V) позволяет обеспечивать взаимодействие произвольных процессов в пределах локальной машины. Для этого она дает взаимодействующим процессам использовать **общие**, или **разделяемые**, ресурсы (могут существовать в системе с момента создания до момента их принудительного удаления либо в течение сеанса работы ОС (вне зависимости от наличия процессов, которые их используют).

Типы таких ресурсов:
1. **Очередь сообщений**. Один процесс может в эту очередь положить сообщение, а другой процесс - прочитать его. Механизм умеет блокировать, поэтому его можно использовать и для передачи информации между процессами, и для их синхронизации.
2. **Массив семафоров** - массив из N элементов (задается при создании данного ресурса) и каждый из элементов является семафором **IPC** (а **не семафором Дейкстры**: семафор Дейкстры так или иначе является формализмом, не опирающимся ни на какую реализацию, а семафор IPC — конкретной программной реализацией семафора в ОС). Семафоры IPC нужны для синхронизации.
3. **Общая**, или **разделяемая**, **память**, которая дается процессу как указатель на область памяти, которая является общей для двух и более процессов (внутри процесса некоторый указатель можно установить на начало данной области и работать далее с этой областью, как с массивом). Все изменения, которые сделает данный процесс, будут видны другим процессам. Разделяемая память IPC почти не обладает никакими средствами синхронизации (существует, но очень слабый механизм взаимных блокировок)

Для совместного использования разделяемых ресурсов необходим **механизм именования ресурсов** - система ключей: при создании ресурса с ним ассоциируется конкретный **ключ** — целочисленное значение **key**. Теперь все процессы, желающие работать с этим ресурсом:

1. должны обладать соответствующими правами
2. должны заявить, что они желают работать с конкретным ресурсом P с ключом **key**

Жесткого ограничения к выбору ключей нет, но проблемно, если они совпадают у разных ресурсов, так что имеет место унификация именования IPC-ресурсов. Для генерации уникальных ключей в системе имеется библиотечная функция ftok().

```
#include <sys/types.h>, <sys/ipc.h>
key_t ftok(char *filename, char proj);
```

(имя существующего файла, информация
(значение символьной переменной) может использоваться
для поддержания разных версий программы).

1. обращается к указанному файлу
2. считывает его атрибуты*
3. генерирует уникальный ключ (целое число), используя второй аргумент

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.

Тонкости:

- Каждый ресурс IPC имеет атрибут владельца. Владельцем считается тот процесс (его идентификация), который создал данный ресурс, и, соответственно, удалить данный ресурс может только владелец. Но существует механизм передачи прав владения от процесса процессу.
- С каждым ресурсом, так же как и с файлами, связаны три категории прав (права владельца, группы и остальных пользователей). Но в каждой категории имеются лишь права на чтение и запись: право на выполнение нет.
- Ресурсы IPC могут существовать без процессов, их создавших. Это означает, что созданный разделяемый ресурс будет существовать до тех пор, пока его явно не удалят либо до перезапуска системы.

Функции работы с разными ресурсами различны, но их структура идентична. В частности, можно выделить функции, имеющие суффикс get (префикс - аббревиатуру имени ресурса).

```
<ResourceName>get(key, ..., flags);
```

В зависимости от flags функции работают либо в режиме создания ресурса, либо в режиме подключения к существующему IPC-ресурсу с указываемым ключом.
flags задаёт права доступа к ресурсу и режимы работы с ресурсом (может быть комбинацией через |):

IPC_PRIVATE - создание частного IPC-ресурса, доступного только процессу, который его создал, не доступного остальным процессам. Т.е. функция get всегда открывает новый ресурс, к которому никто другой не может подключиться. Данный флаг позволяет использовать разделяемый ресурс между родственными процессами (поскольку дескриптор созданного ресурса передается при наследовании сыновним процессам).

IPC_CREAT – его отсутствие означает, что процесс хочет подключиться к существующему ресурсу. Если такой ресурс существует и права доступа позволяют к нему обратиться, то процесс получит дескриптор ресурса и продолжит работу, иначе вернется -1, а переменная errno будет содержать код ошибки. Если же при вызове функции get данный флаг установлен, то функция работает на создание или подключение к существующему ресурсу. (мб ошибка с правами на существующий файл)

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 функция get вернет -1, а переменная errno будет содержать код соответствующей ошибки.

Ошибки при вызове функции get, возвращаемых в переменной errno:
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 IPC - функциональное расширение канала, отличие - сообщения в очереди сообщений IPC типизированы: помимо своей содержательной части, оно имеет атрибут **тип сообщения** (неотрицательное целое значение). очередь сообщений:

- с **одной** стороны, это **сквозная** очередь (когда все сообщения, вне зависимости от типа, рассматриваются как единая очередь)
- с **другой**, это **суперпозиция** очередей однотипных сообщений

При этом способ интерпретации допускает одновременно различные типы интерпретации. Непосредственный выбор интерпретации определяется в момент считывания сообщения из очереди.

Функции: **`#include <sys/types.h>`**, **`<sys/ipc.h>`**

Создание/получение доступа к очереди: `#include <sys/message.h>`

```
int msqid = int msgget(key_t key, int msgflag);
```

В случае успешного выполнения возвращается положительный дескриптор (идентификатор) очереди **msqid**, иначе -1.

Отправка и получение сообщений: `#include <sys/msg.h>` тут определена константа `MSGMAX` (максимальный размер тела сообщения. При попытке отправить сообщение, у которого число элементов в массиве `msgtext` превышает это значение, системный вызов вернет –1)

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

1. идентификатор очереди,
2. указатель на буфер, содержащий реальные данные и тип сообщения, подлежащего посылке в очередь,
3. размер буфера,

для отправки:

4. флаг (если 0 - вызов будет блокироваться, если для отправки недостаточно системных ресурсов. Флаг IPC_NOWAIT позволяет работать без блокировки: в случае возникновения ошибки при отправке сообщения, вызов вернет -1, а переменная errno будет содержать соответствующий код ошибки)

для получения:

4. тип сообщения, которое процесс желает получить (0 - любое, >0 - указанного типа, <0 - наименьшее значение среди типов, которые меньше модуля)
5. комбинация | флагов. Если среди флагов не указан IPC_NOWAIT, и в очереди не найдено ни одного сообщения, удовлетворяющего критериям выбора, процесс будет заблокирован до появления такого сообщения. (Однако, 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. Процесс может также указать 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.

- при доступе используется стратегия FIFO.

Буфер – структура с полями:
long msgtype — тип сообщения (только положительное длинное целое);
har msgtext[] — данные (тело сообщения).

В случае успешного завершения функция возвращает количество успешно прочитанных байтов в теле сообщения.

управление ресурсом – изменение режима функционирования ресурса (например удаление ресурса)

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

(идентификатор ресурса, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры очереди)

msgid_ds – структура с полями, описывающими параметры очереди: права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возможные значения аргумента **cmd:**

- **IPC_STAT** - скопировать структуру по адресу, указанному в параметре **buf**;
- **IPC_SET** - заменить указанную структуру на структуру, находящуюся по адресу, указанному в параметре **buf**;
- **IPC_RMID** - удалить очередь (сделать это может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя)

Благодаря типизации сообщений, очередь сообщений позволяет мультиплексировать сообщения от различных процессов, при этом каждая пара взаимодействующих через очередь процессов может использовать свой тип сообщений, и таким образом, их данные не будут смешиваться.

Разделяемая память IPC

Механизм позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти. Данные из этой области памяти, будут доступны для чтения и модификации всем процессам, подключившимся к данной области. Процесс, подключившийся к разделяемой памяти, может затем получить указатель на некоторый адрес в своем виртуальном адресном пространстве, соответствующий данной области разделяемой памяти. После этого он может работать с ней так, как если бы она была выделена динамически (например, malloc()), однако, разделяемая область памяти не уничтожается автоматически даже после прекращения процесса работать с ней.

Создание/подключение к ресурсу разделяемой памяти IPC

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

(ключ, размер области памяти, к которой процесс желает получить доступ (если больше, чем доступная память вернется -1) / которую создаст **shmget()**, флаги, управляющие поведением вызова)
В случае успешного завершения вызов возвращает положительное число — дескриптор области памяти, в случае неудачи возвращается -1.

Но даже с дескриптором разделяемой памяти процесс не может работать с ресурсом, поскольку при работе с памятью процесс работает в терминах адресов. Поэтому…
Присоединение полученной разделяемой памяти к виртуальному адресному пространству процесса

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

1.дескриптор памяти
2.виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память:

- 0 - система сама может выбрать адрес начала разделяемой памяти
- целое > 0 - конкретный адрес (возможны коллизии с имеющимся адресным пространством)

3.комбинация флагов: SHM_RDONLY – только читать (если аппаратура не поддерживает защиту памяти от записи, то при установке флага SHM_RDONLY ошибка модификации содержимого памяти не будет обнаружена - поскольку программно невозможно выявить, в какой момент происходит обращение на запись в данную область памяти).
Эта функция возвращает адрес (указатель) в виртуальном адресном пространстве процесса, начиная с которого будет отображаться присоединяемая разделяемая память. И с этим указателем можно работать стандартными средствами языка С.
В случае неудачи вызов возвращает -1.
Теперь процесс может читать и модифицировать данные, находящиеся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.

Открепление разделяемой памяти от адресного пространства процесса:

```
int shmdt(char *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.

Управление разделяемой памятью: получение или изменение процессом управляющих параметров области разделяемой памяти, наложение и снятие блокировки на нее, ее уничтожение.

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

(дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти)

Тип **shmid_ds** описан в заголовочном файле **<sys/shm.h>,** и представляет собой структуру, в полях которой хранятся права доступа к области памяти, ее размер, число процессов, подсоединенных к ней в данный момент, и статистика обращений к области памяти.
Возможные значения аргумента **cmd**:
IPC_STAT / IPC_SET / IPC_RMID
Выполнить 2 и 3 операцию может процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя, при этом процесс может изменить только владельца области памяти и права доступа к ней.

SHM_LOCK / **SHM_UNLOCK** – блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя.

3.2.3 Массив семафоров IPC

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

Семафор - особый вид числовой переменной, над которой определены две неделимые операции:

1. уменьшение ее значения с возможным блокированием процесса
2. увеличение значения с возможным разблокированием одного из ранее заблокированных процессов

Объект System V IPC представляет собой набор семафоров:

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

Создание/доступ к набору семафоров:

```
#include <sys/types.h>, <sys/ipc.h>, <sys/sem.h>
int semget (key_t key, int nsems, int semflag);
```

(ключ для доступа к разделяемому ресурсу, количество семафоров в создаваемом наборе (длина массива семафоров), флаги, управляющие поведением вызова)

Тонкости:
- 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.
- если среди флагов указан **IPC_CREAT**, **nsems** - положительное число, если же этот флаг не указан, значение **nsems** игнорируется
- в заголовочном файле **<sys/sem.h>** определена константа **SEMMSL** - максимально возможное число семафоров в наборе. Если значение аргумента **nsems** больше этого значения, вызов **semget()** завершится неудачно.

В случае успеха вызов **semget()** возвращает положительный дескриптор созданного разделяемого ресурса, в случае неудачи -1.

Изменение значения одного или нескольких семафоров в наборе, а также проверять их значения на равенство нулю:

```
int semop (int semid, struct sembuf *semop, size_t nops)
```

(дескриптор массива семафоров, массив из объектов типа **struct sembuf**, каждый из которых задает одну операцию над семафором, длина массива **semop**)
количество семафоров, над которыми процесс может одновременно производить операцию в одном вызове **semop()**, ограничено константой **SEMOPM**, описанной в файле **<sys/sem.h>**, если процесс попытается вызвать **semop()** с параметром **nops**, большим этого значения, этот вызов вернет неуспех.

Структура имеет **sembuf** вид:

```
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 (**sem_op**.) не равно нулю - оценивается значение суммы **sem_val + sem_op**.
   - >= 0 : **sem_val = sem_val + sem_op**
   - < 0, то действие процесса будет приостановлено до тех пор, пока значение не станет >= 0, после чего **sem_val = sem_val + sem_op**
2. Если код операции **sem_op** равен нулю:
   - если при этом значение семафора (**sem_val**) равно нулю, происходит немедленный возврат из вызова
   - иначе происходит блокирование процесса до тех пор, пока значение семафора не обнулится, после чего происходит возврат из вызова

Таким образом, ненулевое значение поля **sem_op** обозначает необходимость прибавить к текущему значению семафора значение **sem_op**, а нулевое – дождаться обнуления семафора.

Итак, порядок работы с семафором можно записать в виде следующей схемы:

```
Если sem_op = 0 то
  если sem_val ≠ 0 то
    пока (sem_val ≠ 0) [процесс блокирован]
      [возврат из вызова]
Если sem_op ≠ 0 то
  если sem_val + sem_op < 0 то
    пока (sem_val+sem_op< 0) [процесс блокирован]
  sem_val = sem_val + sem_op
```

Поле **sem_flg** в структуре **sembuf** содержит комбинацию флагов, влияющих на выполнение операции с семафором. В этом поле может быть установлен флаг **IPC_NOWAIT**, который предписывает соответствующей операции над семафором не блокировать процесс, а сразу возвращать управление из вызова **semop()**. Вызов **semop()** в такой ситуации вернет –1. Кроме того, в этом поле может быть установлен флаг **SEM_UNDO**, в этом случае система запомнит изменение значения семафора, произведенные данным вызовом, и по завершении процесса автоматически ликвидирует это изменение. Это предохраняет от ситуации, когда процесс уменьшил значение семафора, начав работать с ресурсом, а потом, не увеличив значение семафора обратно, по какой-либо причине завершился. В этом случае остальные процессы, ждущие доступа к ресурсу, оказались бы заблокированы навечно.

Управление состоянием массива семафоров

запрос и изменение управляющих параметров разделяемого ресурса, а также удаление его:

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

(дескриптор массива семафоров, индекс семафора в массиве, код операции, которая должна быть выполнена над данным семафором, аргумент типа **union semun**, который используется для считывания или задания управляющих параметров одного семафора или всего массива, в зависимости от значения аргумента **cmd**)

В случае успешного завершения возвращается значение, соответствующее конкретной выполнявшейся операции (0, если не оговорено иное), в случае неудачи -1.

Тип данных **union semun** определен в файле **<sys/sem.h>** 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
}
```

struct semid_ds – структура, описанная в том же файле, в полях которой хранится информация о всем наборе семафоров в целом, а именно, количество семафоров в наборе, права доступа к нему и статистика доступа к массиву 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 обладать правами привилегированного пользователя, при этом процесс может изменить только владельца массива семафоров и права доступа к нему.

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

GETVAL – возвратить значение семафора с номером **num**. Последний аргумент вызова игнорируется.

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

Сокеты — унифицированный интерфейс программирования распределенных систем

Средства межпроцессного взаимодействия ОС UNIX, представленные в системе IPC, решают проблему взаимодействия процессов, выполняющихся в рамках одной ОС. Однако, очевидно, их невозможно использовать, когда требуется организовать взаимодействие процессов в рамках сети. Это связано как с принятой системой именования, которая обеспечивает уникальность только в рамках данной системы, так и вообще с реализацией механизмов разделяемой памяти, очереди сообщений и семафоров, – очевидно, что для удаленного взаимодействия они не годятся. Следовательно, возникает необходимость в каком-то дополнительном механизме, позволяющем общаться двум процессам в рамках сети. Однако если разработчики программ будут иметь два абсолютно разных подхода к реализации взаимодействия процессов, в зависимости от того, на одной машине они выполняются или на разных узлах сети, им, очевидно, придется во многих случаях создавать два принципиально разных куска кода, отвечающих за это взаимодействие. Понятно, что это неудобно и хотелось бы в связи с этим иметь некоторый унифицированный механизм, который в определенной степени позволял бы абстрагироваться от расположения процессов и давал бы возможность использования одних и тех же подходов для локального и нелокального взаимодействия. Кроме того, как только мы обращаемся к сетевому взаимодействию, встает проблема многообразия сетевых протоколов и их использования. Понятно, что было бы удобно иметь какой-нибудь общий интерфейс, позволяющий пользоваться услугами различных протоколов по выбору пользователя.
Обозначенные проблемы был призван решить механизм, впервые появившийся в UNIX – BSD (4.2) и названный сокетами (**sockets**).

Сокеты - обобщение механизма каналов, но с учетом особенностей, возникающих при работе в сети. Кроме того, они поддерживают передачу экстренных сообщений вне общего потока данных.

Типы сокетов. Коммуникационный домен

Сокеты подразделяются на несколько типов в зависимости от типа коммуникационного соединения:

1. Соединение с использованием виртуального канала – это последовательный поток байтов, гарантирующий надежную доставку сообщений с сохранением порядка их следования. Данные начинают передаваться только после того, как виртуальный канал установлен, и канал не разрывается, пока все данные не будут переданы (каналы в UNIX, телефонный разговор)
   - границы сообщений при таком виде соединений не сохраняются, т.е. приложение, получающее данные, должно само определять, где заканчивается одно сообщение и начинается следующее
   - можно передавать экстренные сообщения вне основного потока данных, если протокол позволяет

2. Датаграммное соединение - передача отдельных пакетов, содержащих порции данных – **датаграмм**.
   - Не гарантируется доставка вообще и доставка в том же порядке, в каком они были посланы
   - надежность соединения ниже
   + выше скорость соединения
   (обычная почта: письма и посылки могут приходить адресату не в том порядке, в каком они были посланы, а некоторые из них могут и совсем пропадать)

Поскольку сокеты могут использоваться как для локального, так и для удаленного взаимодействия, встает вопрос о пространстве адресов сокетов. При создании сокета указывается **коммуникационный домен**, к которому данный сокет будет принадлежать. Коммуникационный домен определяет конкретную модель именования (форматы адресов, правила их интерпретации), а также область взаимодействия процессов.

1. **`AF_UNIX`** для локального взаимодействия, формат адреса – это допустимое имя файла, используются внутренние протоколы ОС
2. **`AF_INET`** для взаимодействия в рамках сети, адрес образуют имя хоста + номер порта (порт – виртуальная точка соединения, которая позволяет адресовать конкретный процесс извне), используются протоколы семейства TCP/IP

*Современные системы поддерживают и другие коммуникационные домены, например BSD UNIX поддерживает также третий домен – **`AF_NS`**, использующий протоколы удаленного взаимодействия Xerox NS.

Создание и конфигурирование сокета

Создание сокета

```
#include <sys/types.h>, <sys/socket.h>
int socket (int domain, int type, int protocol)
```

1.константа коммуникационного домена, к которому должен принадлежать создаваемый сокет (**`AF_UNIX`** / **`AF_INET`**
2.тип соединения, которым будет пользоваться сокет (и, соответственно, тип сокета) - константы **`SOCK_STREAM`** для соединения с установлением виртуального канала и **`SOCK_DGRAM`** для датаграмм
3.конкретный протокол, который будет использоваться в рамках данного коммуникационного домена для создания

соединения. 0 - система автоматически выберет подходящий протокол / константы для протоколов, используемых в домене **AF_INET:**
IPPROTO_TCP –протокол TCP (корректно при создании сокета типа **SOCK_STREAM**)
IPPROTO_UDP –протокол UDP (корректно при создании сокета типа **SOCK_DGRAM**)

В случае успеха возвращает положительное целое число – дескриптор сокета (фактически файловый дескриптор - он 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)
В ошибки (например, при некорректном сочетании коммуникационного домена, типа сокета и протокола), возвращается –1.

Связывание - чтобы к созданному сокету мог обратиться какой-либо процесс извне, необходимо связать с этим сокетом некоторое имя (адрес). Формат адреса зависит от коммуникационного домена, в рамках которого действует сокет, и может представлять собой либо путь к файлу, либо сочетание IP-адреса и номера порта.

```
int bind (int sockfd, struct sockaddr *myaddr, int addrlen)
```
(дескриптор сокета, указатель на структуру, содержащую адрес сокета, размер этой структуры)

Для **AF_UNIX** формат структуры описан в **<sys/un.h>**

```
struct sockaddr_un {
       short sun_family;
       char sun_path[108];
};
```

Для **AF_INET** формат структуры описан в **<netinet/in.h>**

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

Важно отметить, что если мы имеем дело с доменом **AF_UNIX** и адрес сокета представляет собой имя файла, то при выполнении функции **bind()** система в качестве побочного эффекта создает файл с таким именем. Поэтому для успешного выполнения **bind()** необходимо, чтобы такого файла не существовало к данному моменту. Это следует учитывать, если мы «зашиваем» в программу определенное имя и намерены запускать нашу программу несколько раз на одной и той же машине – в этом случае для успешной работы **bind()** необходимо удалять файл с этим именем перед связыванием. Кроме того, в процессе создания файла, естественно, проверяются права доступа пользователя, от имени которого производится вызов, ко всем директориям, фигурирующим в полном путевом имени файла, что тоже необходимо учитывать при задании имени. Если права доступа к одной из директорий недостаточны, вызов **bind()** завершится неуспешно.
В случае успешного связывания **bind()** возвращает 0, в случае ошибки -1**.**

Предварительное установление соединения.

Сокеты с установлением соединения. Запрос на соединение.

1. **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. Если тип сокета – виртуальный канал, то сокет **должен** устанавливать соединение
2. **сокеты без установления соединения**, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением. Датаграммы, как правило, не устанавливают соединения, но это не является требованием

Установление соединения:

```
int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);
```

(дескриптор сокета, указатель на структуру, содержащую адрес сокета, с которым производится соединение, в необходимом формате, реальная длина этой структуры)
Функция возвращает 0 в случае успеха и –1 в случае неудачи + в **errno.**

Заметим, что в рамках модели «клиент-сервер» клиенту, вообще говоря, не важно, какой адрес будет назначен его сокету, так как никакой процесс не будет пытаться непосредственно установить соединение с сокетом клиента. Поэтому клиент может не вызывать предварительно функцию **bind()**, в этом случае при вызове **connect()** система автоматически выберет приемлемые значения для локального адреса клиента. Однако сказанное справедливо только для взаимодействия в рамках домена **AF_INET**, в домене **AF_UNIX** клиентское приложение само должно позаботиться о связывании сокета.

Сервер: прослушивание сокета и подтверждение соединения.

Следующие два вызова используются сервером только в том случае, если используются сокеты с предварительным установлением соединения.
Вызов, который использует процесс-сервер, для сообщения системе о своей готовности обрабатывать запросы на соединение, поступающие на данный сокет. До тех пор, пока процесс – владелец сокета не вызовет **listen(),** все запросы на соединение с данным сокетом будут возвращать ошибку.

```
int listen (int sockfd, int backlog);
```

(дескриптор серверного сокета, максимальный размер очереди запросов на соединение)
ОС буферизует приходящие запросы на соединение, выстраивая их в очередь до тех пор, пока процесс не сможет их обработать. В случае если очередь запросов на соединение переполняется, поведение ОС зависит от того, какой протокол используется для соединения:

- если он не поддерживает возможность перепосылки (retransmission) данных, то соответствующий вызов **connect()** вернет ошибку **ECONNREFUSED**
- если же перепосылка поддерживается (как, например, при использовании TCP), ОС просто выбрасывает пакет, содержащий запрос на соединение, как если бы она его не получала вовсе. А пакет будет присылаться повторно до тех пор, пока очередь запросов не уменьшится и попытка соединения не увенчается успехом, либо пока не произойдет тайм-аут, определенный для протокола. В последнем случае вызов **connect()** завершится с ошибкой **ETIMEDOUT** (позволяет клиенту отличить, был ли процесс-сервер слишком занят или он не функционировал. В большинстве систем максимальный допустимый размер очереди равен 5)

Конкретное соединение - удовлетворение поступившего клиентского запроса на соединение с сокетом, который сервер к тому моменту уже прослушивает (т.е. предварительно была вызвана функция **listen()**)

```
int accept (int sockfd, struct sockaddr *addr, int *addrlen);
```

(дескриптор серверного сокета, указатель на структуру, в которой возвращается адрес клиентского сокета, с которым установлено соединение, указатель переменную, в которой возвращается реальная длина этой структуры)
* 2-й параметр позволяет серверу знать, куда ему в случае надобности следует послать ответное сообщение. Если адрес клиента нас не интересует, в качестве второго аргумента можно передать **NULL**.

1. извлекается первый запрос из очереди запросов, ожидающих соединения, и устанавливает с ним соединение (если к моменту вызова **accept()** очередь запросов на соединение пуста, процесс, вызвавший **accept()**, блокируется до поступления запросов)
2. создается новый серверный сокет, который будет использоваться для работы с данным соединением
3. возвращается дескриптор этого нового сокета, соединенного с сокетом клиентского процесса
4. через новый сокет осуществляется обмен данными
5. старый серверный сокет продолжает оставаться в состоянии прослушивания и обработки других поступающих запросов на соединение (именно этот сокет связан с адресом, известным клиентам, поэтому все клиенты могут слать запросы только на соединение с этим сокетом) - позволяет процессу-серверу поддерживать несколько соединений одновременно: обычно это 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.

Прием и передача данных

обмен только через сокет с предварительно установленным соединением

```
int send(int sockfd, const void *msg, int len, unsigned int flags);
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

(дескриптор сокета, через который передаются данные, **msg** и **len** - сообщение и его длина

- если сообщение слишком длинное для того протокола, который используется при соединении, оно не передается и вызов возвращает ошибку **EMSGSIZE**
- если же сокет окажется переполнен, т.е. в его буфере не хватит места, чтобы поместить туда сообщение, выполнение процесса блокируется до появления возможности поместить сообщение.

/ buf и **len** – указатель на буфер для приема данных и его первоначальная длина,
И комбинация опций **flags:**
MSG_OOB – этот флаг сообщает ОС, что процесс хочет осуществить прием/передачу экстренных сообщений
MSG_PEEK – данный флаг может устанавливаться при вызове **recv()**. Процесс может прочитать порцию данных, не удаляя ее из сокета, так, что последующий вызов **recv()** вновь вернет те же самые данные.
Возвращают количество переданных/считанных байт в случае успеха и -1 в случае неудачи + **errno.**

Другая пара функций, которые могут использоваться при работе с сокетами с предварительно установленным соединением – это обычные **`read()`** и **`write()`**, в качестве дескриптора которым передается дескриптор сокета.

И, наконец, пара функций, которая может быть использована как с сокетами с установлением соединения, так и с сокетами без установления соединения:

```
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);
```

Первые 4 аргумента у них такие же, как и у рассмотренных + **`sendto`** - указатель на структуру, содержащую адрес получателя, и ее размер, **`recvfrom()`** – возвращает соответственно указатель на структуру с адресом отправителя и ее реальный размер.
Отметим, что перед вызовом **`recvfrom()`** параметр **`fromlen`** должен быть установлен равным первоначальному размеру структуры **`from`.** Здесь, как и в функции **`accept`**, если нас не интересует адрес отправителя, в качестве **`from`** можно передать **`NULL`.**

Завершение работы с сокетом

– процесс закончил прием либо передачу данных, ему следует закрыть соединение

```
int shutdown (int sockfd, int mode);
```

Помимо дескриптора сокета, ей передается целое число, которое определяет режим закрытия соединения:

- **0** - сокет закрывается для чтения, при этом все дальнейшие попытки чтения будут возвращать **`EOF`**
- **1** - сокет закрывается для записи, и при осуществлении в дальнейшем попытки передать данные будет выдан кода неудачного завершения (-1).
- **2** - сокет закрывается и для чтения, и для записи.

Аналогично файловому дескриптору, дескриптор сокета освобождается системным вызовом **`close()`** и соединение будет закрыто даже без **`shutdown()`**. Однако, если используемый для соединения протокол гарантирует доставку данных (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. Таким образом, вызов **`shutdown()`** важен в первую очередь для закрытия соединения сокета с использованием виртуального канала.

Резюме: общая схема работы с сокетами

Имеется возможность организации взаимодействия с **предварительным установлением соединения** - ориентировано на клиент-серверные системы, когда организуется один серверный узел процессов (не процесс, а именно узел процессов), который принимает сообщения от клиентский процессов и их как-то обрабатывает. Тип сокета в данном случае не важен, - можно использовать как сокеты, являющиеся виртуальными каналами, так и дейтаграммные сокеты.
<u>В данной модели можно выделить две группы процессов: процессы серверной части и процессы клиентской части.</u>
На **сервере**:

1. открывается основной сокет и bind() связывает его с именем, чтобы другие процессы могли обратиться
2. серверный процесс переводится listen() в режим прослушивания - данный процесс может принимать запросы на соединение с ним от клиентских процессов. При этом, в *listen()* оговаривается очередь запросов на соединение/.

На **клиентском хосте**:

1. клиентский процесс создает свой сокет (его связывать необязательно, т.к. сервер может работать с любым клиентом, в частности, и с «анонимным»)
2. клиентский процесс может передать серверному процессу сообщение, что он с ним хочет соединиться - *connect()*:
    a. клиент обращается к *connect()* до того, как сервер перешел в режим прослушивания. В этом случае клиент получает отказ с уведомлением, что сервер в данный момент не прослушивает сокет.
    b. клиент может обратиться к *connect()*, а у серверного процесса в данный момент сформирована очередь необработанных запросов, и эта очередь пока не насыщена. В этом случае запрос на соединение встанет в очередь, и клиентский процесс заблокируется и будет ожидать обслуживания.
    c. указанная очередь переполнена. В этом случае клиент получает отказ с соответствующим уведомлением, что сервер в данный момент занят.

Схема в системе «клиент-сервер» (в остальных так же):
1. процесс-сервер запрашивает у ОС сокет
2. получив его, присваивает ему некоторое имя (адрес), заранее известное всем клиентам, которые захотят общаться с данным сервером
3. сервер переходит в режим ожидания и обработки запросов от клиентов
4. клиент тоже создает сокет со своей стороны и запрашивает соединение своего сокета с сокетом сервера, зная его имя (адрес).
5. после установления соединения, клиент и сервер могут обмениваться данными через соединенную пару сокетов
6. сокеты уничтожаются

Итак, данная схема организована таким образом, что к одному серверному узлу может быть множество соединений. Для организации этой модели имеется системный вызов *accept()*:

- если в очереди имеется необработанная заявка на соединение, создается новый локальный сокет, который связывается с клиентом
- клиент может посылать сообщения серверу через этот новый сокет
- сервер, получая через данный сокет сообщения от клиента, «понимает», от какого клиента пришло это сообщение (т.е. клиент получает некоторое внутрисистемное именование, даже если он не делал связывание своего сокета). И, соответственно, в этом случае сервер может посылать ответные сообщения клиенту.

Завершение:
1. отключение доступа к сокету через *shutdown()* (закрыть сокет на чтение, на запись, на чтение-запись) - обеспечивается корректное прекращение работы с сокетом (в частности, это важно при организации виртуального канала, который должен гарантировать доставку посланных данных).
2. закрытие сокета через *close()* (т.е. закрытие сокета как файла).

Также сервер может порождать сыновний процесс для каждого вызова *accept()*, и тогда все действия по работе с данным клиентом ложатся на этот сыновний процесс. На родительский процесс возлагается задача прослушивание «главного» сокета и обработка поступающих запросов на соединение - не один процесс на сервере, а о серверный узел, в котором может находиться целый набор процессов.

Сокеты ***без предварительного соединения*** (лишь дейтаграммные сокеты) не иерархически организована, как клиент-сервер, а произвольно - у каждого взаимодействующего процесса имеется сокет, через который он может получать информацию от различных источников. Если же сервер мог взаимодействовать с клиентом, не зная его имя явно, то тут любой процесс через свой сокет может послать информацию любому другому сокету, то есть механизм отправки имеет соответствующую адресную информацию (имя грубо говоря).

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

Файловая система (ФС) - часть ОС, программный компонент, появившийся в машинах 2-го поколения. ФС – это организованные наборы данных, хранящихся на ВЗУ + программные средства, гарантирующие **корректный именованный** доступ к этим данным и их защиту. Данные в ФС представляются в виде файлов, каждый из которых имеет имя. Главными характеристиками в определении ФС являются **именованный доступ** и **корректная работа** (ФС обеспечивает корректное управление свободным и занятым пространством на ВЗУ (не обязательно физическом устройстве, может и на виртуальном), а также защиту информации от несанкционированного доступа. Большинство современных ФС обеспечивают корректную организацию распределенного доступа к одному и тому же файлу (когда с ним могут работать два и более пользователя). Это не означает, что система будет отвечать за корректную семантику данных внутри файла: гарантируется, что система обеспечит корректный доступ пользователей к файлу с точки зрения системной организации. Также многие современные ФС поддерживают возможность синхронизации доступа к информации.

С точки зрения пользователя, файловая система является первым виртуальным ресурсом (который появился в ОС).
1. Проблема размещения данных во внешней памяти - необходимо сохранять данные: «одноразовых» задач уже нет, современные задачи требуют больших объемов начальных данных (результатов решений других задач), которые надо где-то хранить (без программ, которые их используют)
2. Как следствие, проблема эффективности доступа к этим данным.
3. Само сохранение информации (и программ, и данных). Имеется в виду, сам факт долгосрочного хранения информации.

Структурная организация файлов

1. **Файл как последовательность байтов** (аналог магнитной ленты). Содержимое файла - неинтерпретируемая информация (или интерпретируемая примитивным образом). Задача интерпретации данных ложится на пользователя. Наиболее распространена.
   + программные интерфейсы, обеспечивающие доступ к содержимому файла, позволяют считывать и записывать произвольные порции данных
   - коррекция данных в середине файла приводит к изменению размера файла - необходимосто сдвигать «хвост» файла на ленте

2. **Файл как последовательность записей переменной и постоянной длины** (аналог перфокарты)
   + и – перфокарт

3. **Иерархическая организация файла.** Одной из наиболее распространенных структур является дерево, в узлах которого расположены записи. Каждая запись состоит из двух полей: поле ключа (например, номер записи) и поле данных
   + можно организовывать динамическую работу с данными
   + удобное редактирование файла
   - сложная реализация

*Еще одной исторической характеристикой файлов были режимы доступа - прямого и последовательного. Режим доступа задавался на этапе создания файла. В современных файловых системах эти режимы не используются. Зато актуальны режимы доступа с точки зрения разрешения или запрета определенные операций: возможно иметь доступ только по чтению, только по записи или по чтению-записи информации в файл.

Атрибуты файлов - фиксированный набор параметров, которыми обладает файл, характеризующих его свойства и состояния, причем и долговременное (стратегическое), и оперативное. Точный состав атрибутов зависит от конкретной реализации системы.

- **имя файла** - последовательность символов, с помощью которой организуется именованный доступ к данным файла. В одних файловых системах имя файла -атрибут, в других нет (разделяют файл (его содержимое), имя и отдельно набор атрибутов)
- **права доступа -** определяет доступ к содержимому файла различным категориям пользователей. Структура категорий пользователей зависит от конкретной операционной системы (существуют ОС, в которых прав доступа нет: файлы доступны любому пользователю системы)
- **персонификация** (создатель/владелец) - содержит информацию о принадлежности файла (В общем случае несколько параметров: информация о создателе файла, о владельце и т.п.)
- **тип файла** - информация о способе организации файла и интерпретации его содержимого. ФС ОС Unix поддерживает разные типы файлов: ***файлы устройств***, соответствующие тем устройствам, которые обслуживает данная ОС; и через эти файлы устройств происходит фактически обращение к драйверам устройств. ***Регулярные файлы*** - хранят различную информацию (текстовую, графическую и пр.). Интерпретация может быть ***явной*** и ***неявной*** (можно указать, является ли данный файл **исполняемым** (можно запустить как процесс) или **неисполняемым**.
- **размер записи (блока)** - Можно указать, что данный файл организован в виде последовательности блоков данного размера, при этом размер определяется пользователем (пользовательским процессом). Размер может быть ***стационарным*** (при создании файла указывается фиксированный размер блоков) или ***нестационарным*** (размер блока задается каждый раз при открытии файла)
- **размер файла** (обычно в байтах)
- **указатель чтения/записи** - относительно него происходит чтение или запись информации. В общем случае с каждым файлом ассоциируются два указателя (и на чтение, и на запись), хотя бывают файловые системы, в которых используется единый указатель чтения/записи.
- **время создания, время последней модификации, время последнего обращения, предельный размер файла** и т.п. - атрибуты, отражающие системную и статистическую информацию о файле
- атрибуты, хранящие **информацию о размещении содержимого файла**, т.е. где в файловой системе организовано хранение данных файла, и как оно организовано.

Основные правила работы с файлами. Типовые программные интерфейсы

1. **начало работы с файлом** (или **открытие файла**):
    a. посредством специальной операции открытия процесс передает ФС запрос на работу с конкретным файлом
    b. получив запрос, ФС проверяет возможности (на наличие полномочий) работы с файлом
    c. в случае успеха выделяет внутри себя необходимые ресурсы для работы процесса с указанным файлом. Для каждого открытого файла создается **файловый дескриптор** – системная структура данных, содержащая информацию об актуальном состоянии открытого файла (режимы, позиции указателей и т.п.). Размещается как в адресном пространстве процесса, так и в пространстве памяти ОС. Соответственно, 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 (процесс прекращает работу не с файлом, а с конкретным файловым дескриптором, поскольку даже в одном процессе можно открыть один и тот же файл два и более раза, и на каждое открытие будет предоставлен новый файловый дескриптор).
4. ОС корректно завершает работу с файловым дескриптором, а если закрывается последний открытый дескриптор — то корректно завершает работу с файлом: освобождаются системные ресурсы, выполняется разгрузка кэш-буферов файловых обменов и т.д.

<u>унифицированный набор интерфейсов, посредством которых можно обращаться к системным вызовам работы с файлами:</u>
open — открытие/создание файла
close — закрытие
read/write — чтение/запись (относительно указателей чтения/записи)
delete — удаление файла из файловой системы
seek — позиционирование указателя чтения/записи
read_attributes/write_attributes — чтение/модификация некоторых атрибутов файла (в ФС, рассматривающих имя файла не как атрибут, возможна дополнительная функция переименования файла — **rename**).

Каталог – системная структура данных ФС, которая устанавливает соответствие между именем файла, его атрибутами и содержимым. (обеспечивает доступ по имени). Каталоги являются специальным видом файлов.

<u>Модели организации каталогов:</u>

1. **одноуровневая файловая 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.
    + простота организации и доступа к информации файлов
    - не предполагает многопользовательской работы (коллизии имен)
2. **двухуровневая файловая система** - группировка файлов по принадлежности тому или иному пользователю.
    + не возникают коллизии имен файлов разных пользователей
    + простота организации и доступа
    + многопользовательская работа
    - коллизии имен для файлов одного пользователя.
    - неудобно и даже нежелательно расположение всех файлов одного пользователя в одном месте (в одном каталоге)
3. **иерархическая файловая система** - вся информация представляется в виде дерева, имеющего корень - **корневая файловая система**. В узлах дерева, отличных от листьев, находятся каталоги, которые содержат информацию о размещенных в них файлах. Такие ФС имеют специальный тип файлов-каталогов (не отдельная выделенная структура данных, а файл особого типа). Листом дерева может быть либо файл-каталог, либо любой файл ФС.
    + уникальное именование файлов (т.к. существует единственный путь от корня до любого узла)
    Текущий каталог — это каталог, на работу с которым в данный момент настроена ФС. Файлы, находящиеся непосредственно в текущем каталоге, доступны «просто» по имени.
    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.
    Имя файла — это имя файла относительно текущего каталога
    Полное имя файла — это перечень всех имен файлов от корня до узла с данным файлом (признак полного имени - префиксный символ, обозначающий корневой каталог (в ОС Unix “/”).
    Относительное имя файла — это путь от некоторого указанного явно (по сути дано полное имя) или неявно (каталог сделан текущим) каталога до данного файла. (если полное имя **/A/B/C/D**, то относительно каталога **B** имя **C/D)**.

Подходы в практической реализации ФС

Системное устройство — устройство, на котором, как считает аппаратура компьютера, присутствует ОС. Почти в любом компьютере можно определить некоторую цепочку внешних устройств с приоритетами, которые при загрузке компьютера могут рассматриваться как системные устройства. Во время старта ВС перебирает данную цепочку в порядке убывания приоритета до тех пор, пока не обнаружит готовое к работе устройство. Система предполагает, что на этом устройстве имеется необходимая системная информация и пытается загрузить с него ОС.
Обычно в штатном режиме загрузка происходит с жесткого диска, но в ситуации, например, краха системы и невозможности загрузиться с жесткого диска приведенная модель позволяет загрузить систему со съемного носителя и произвести некие действия по восстановлению работоспособности поврежденной системы.

Аппаратный загрузчик предполагает определенную структуру каждого системного устройства:

1. в начальном блоке располагается **основной программный загрузчик** (MBR — Master Boot Record) – либо загрузчик конкретной ОС, либо унифицированный мультисистемный загрузчик, знающий структуру системного диска (какая операционная система в каком разделе диска находится) и способный загружать по выбору пользователя одну из альтернативных ОС (по таблице разделов определяет координаты соответствующего раздела и передает управление загрузчику конкретной ОС указанного раздела, который уже загружает эту ОС)
2. последовательность блоков, в которых находится т.н. **таблица разделов**

В современных жестких дисках все физическое пространство диска можно разбить на некоторые области - **разделы** (**partition**). Внутри каждого раздела может быть помещена в общем случае своя ОС. Границы каждого раздела регистрируются в таблице разделов. К тому же современные диски имеют настолько большие емкости, что для адресации произвольной точки диска не хватает разрядной сетки процессора и за счет косвенной адресации (адресации относительно начала раздела) использование этой таблицы позволяет решить проблему.

Структура раздела:

1. в начальном блоке раздела находится ***загрузчик*** конкретной ОС.
2. все остальное пространство раздела обычно занимает ФС:
    a. ***суперблок,*** 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
    b. все оставшееся пространство ФС состоит из свободных и занятых блоков, хранящих системные и пользовательские данные.

Иерархия блоков: (блоки файла != блоки ФС != блоки устройств) современные ОС поддерживают целую иерархию блоков, используемую при организации работы с блок-ориентированными устройствами:

1. Наверху - блоки физического устройства (порции данных, которыми обмениваются с данным ФУ). Размер блока ФУ зависит от конкретного устройства. Блоки ФУ скрываются за блоками виртуального диска.
2. Середина - блоки ФС (организация структуры ФС). Размер блока ФС, равно как и виртуального диска, является стационарной характеристикой, определяемой при настройке системы.
3. Нижний уровень - блоки файла. Размер данных блоков может определить пользователь при открытии или создании файла (как об этом говорилось выше).

* размер блока влияет на эффективность работы, которая будет выше, если размеры всех блоков будут хотя бы кратны друг другу.

Модели реализации файлов

модель непрерывных файлов: каждый файл размещен в непрерывной области ВУ. Для обеспечения доступа к файлу среди атрибутов должно быть имя, блок начала и длина файла.
+ простая организация
+ минимальны накладных расходов является
+ отсутствие фрагментации файла по диску приводит к минимизации механических движений головки, считывающей информацию, что означает более высокую производительность системы
- внутренняя фрагментация (хотя это проблема почти всех блок-ориентированных устройств): если хранить всего один байт, то для этого будет выделен целый блок
- фрагментация между файлами
- проблема модификация файла (увеличение его содержательной части)
Решение: при создании файла к запрошенному объему добавляют немного свободного пространства (например, 10% от запрошенного объема), но тогда увеличивается внутренняя фрагментация.
- фрагментация свободного пространства: в ходе эксплуатации диска она увеличивается, и в итоге появляется много мелких свободных участков, имеющих большой суммарный объем, но недостаточный размер для использования.
Решение: процесс компрессии (дефрагментации) - сдвигает файлы с учетом зарезервированного для каждого файла «запаса», «прижимая» их друг к другу. Эта операция трудоемкая, продолжительная и опасная, поскольку при перемещении файла возможен сбой.

модель файлов, имеющих организацию связанного списка: каждый файл состоит из блоков, составленных из данных, хранимых в файле, и ссылки на следующий блок файла.
+ простая и эффективная организация 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 (содержимое файла может быть рассредоточено по всему дисковому пространству, тогда при последовательном считывании содержимого файла добавляется значительная механическая составляющая, снижающая эффективность доступа)
- в каждом блоке присутствует и системная, и пользовательская информация. Можно вместо 1 логического блока прочитать два.

Таблицы размещения файлов (FAT-таблица): ОС создает программную таблицу, количество строк = количество блоков ФС, предназначенных для пользовательских данных (вразнобой). Также имеется отдельный каталог (или система каталогов), в котором для каждого имени файла имеется запись с номером (по таблице) начального блока файла. В записи таблицы по этому номеру хранится содержимое начального блока и номер следующего блока файла. То есть чтобы получить список блоков ФС, в которых хранится содержимое конкретного файла, надо по имени в указанном каталоге найти номер начального блока, а затем, последовательно обращаясь к таблице размещения за номером следующего блока, построить искомый список.
+весь блок используется полностью для хранения содержимого файла
+ при открытии файла можно составить список блоков данного файла и, следовательно, осуществлять прямой доступ
*Заметим, что для максимальной эффективности необходимо, чтобы эта таблица целиком размещалась в ОП
- современные диски имеют огромные объемы, значит и таблица имеет большой размер
- ограничение на размер файла в силу ограниченности длины списка блоков, принадлежащих данному файлу.

Индексные узлы (дескрипторы) : в атрибуты файла добавляется информация о номерах блоков ФС, в которых размещается содержимое файла. То есть при открытии файла можно сразу получить перечень блоков.
+ необходимость использования FAT-таблицы отпадает
- при предельных размерах файла размер индексного дескриптора становится соизмеримым с размером FAT-таблицы
Решение:
1. тривиальное ограничение на максимальный объем файла
2. построение иерархической организации данных о блоках файла в индексном дескрипторе: первые N блоков перечисляются непосредственно в индексном узле, а оставшиеся представляются в виде косвенной ссылки.

+ нет необходимости в размещении в ОЗУ информации всей FAT-таблицы обо всех файлах системы.
+ в памяти размещаются атрибуты, связанные только с открытыми файлами
+ практически «неограниченная» длина файла (но индексный дескриптор фиксированного размера)

Модели реализации каталогов

1. каталог в виде таблицы, у которой в одной колонке находятся имена файлов, а в остальных — все атрибуты.
   + все необходимые данные оперативно доступны
   - размер записи в таблице каталога, да и сама таблица, может быть большой (например, из-за большого числа атрибутов), что влечет за собой долгий поиск в каталоге.

2. каталог в виде таблицы, в которой один столбец хранит имена, а в другом хранится ссылка на системную таблицу, содержащую атрибуты соответствующего файла.
   + для разных типов файлов можно иметь различный набор атрибутов.

Проблемы организации каталогов:
- проблема длины имени
Решение: (раньше) имя файла было ограничено шестью или восемью символами. В современных системах разрешается присваивать файлам достаточно длинные имена.

- проблема: для эффективной работы с каталогом необходимо, чтобы информация в каталогах хранилась в сжатом виде, в т.ч. и имена файлов должны быть короткими

Соответствие имени файла и его содержимого

- проблема соответствия между именем файла и содержимым этого файла.

1. **взаимно-однозначное соответствие** - однако современные ФС позволяют давать для одного и того же содержимого файла два и более имен

2. **симметричное** (или равноправное) именование: с одним и тем же содержимым ассоциируется группа имен, каждое из которых равноправное (посредством любого из имен, ассоциированных с данным содержимым, можно выполнить все операции, и они будут выполнены одинаково). Такая модель реализована в некоторых ФС посредством установления **жесткой связи** - среди атрибутов есть счетчик имен, ссылающихся на данное содержимое. При уничтожении файла с одним из этих имен ФС уменьшает счетчик имен на единицу; если счетчик обнулился, то в этом случае удаляется содержимое, если нет - удаляется только указанное имя из соответствующего каталога.
   - древовидность ФС нарушается (имена, ассоциированные с одним и тем же содержимым, можно разместить в разных узлах дерева - каталогах файловой системы)
   + сохраняется древовидность именования файлов

3. **несимметричное** именование или «мягкая» ссылка / **символическая связь.**

Пусть существует содержимое файла, с которым жесткой связью ассоциировано некоторое имя (Name2). К этому файлу теперь можно организовать доступ через файл-ссылку. То есть создается еще один файл некоторого специального типа (типа файл-ссылка), в атрибутах которого указывается его тип и то, что он ссылается на файл с именем Name2. Теперь можно работать с содержимым файла Name2 посредством косвенного доступа через файл-ссылку. Но некоторые операции с файлом-ссылкой будут происходить иначе. Например, если вызывается операция удаления файла-ссылки, то удаляется именно файл-ссылка, а не файл с именем Name2. Если же явно удалить файл Name2, то в этом случае файл-ссылка окажется висячей ссылкой.

Координация использования пространства внешней памяти

проблема выбора размера блока ФС: если ФС дает квотировать размер блока, то надо учитывать, что
Больший размер блока ведет к увеличению…
+ производительности ФС (поскольку данные файла оказываются локализованными на жестком диске, при доступе к ним снижается количество перемещений считывающей головки)
- внутренней фрагментации, а, следовательно, возникает неэффективность использования пространства ВЗУ
Альтернативой являются блоки меньшего размера, которые снижают внутреннюю фрагментацию, но и повышают накладные расходы при доступе к файлу в связи фрагментация файла по диску.

проблема учета свободных блоков ФС:

1. номера свободных блоков образуют **связный список**, который располагается в нескольких блоках файловой системы. Для более эффективной работы первый блок должен располагаться в ОЗУ, чтобы ФС могла к нему оперативно обращаться.
   *список может достигать больших размеров: если размер блока 1 Кбайт = 256 * 4 байта => может содержать в себе 255 номеров свободных блоков и одну ссылку на следующий блок со списком => тогда для жесткого диска, емкостью 16 Гбайт, потребуется 16794 элемента списка (блоков). Но размер списка не столь важен, поскольку по мере использования свободных блоков этот список сокращается, при этом освобождающиеся блоки, хранившие указанный список, ничем не отличаются от других свободных блоков файловой системы, а значит, их можно использовать для хранения файловых данных.
2. **битовые массивы**: каждому блоку ФС соответствует двоичный бит(разряд) занятости данного блока.
   - операция пересчета номера разряда в номер блока и наоборот достаточно трудоемка
   - необходимо выделять под массив стационарный ресурс (немалого объема)

Квотирование пространства файловой системы

ФС должна контролировать использование двух видов системных ресурсов — количества имен файлов, которое можно зарегистрировать в каталоге и свободного пространства (чтобы не возникла ситуация, когда один процесс заполнил все свободное пространство, тем самым не давая другим пользователям возможность сохранять свои данные).
Решение - квотирование файловых имен (их числа) и квотирование блоков
Типы лимитов:

1. **жесткий** - количество имен в каталогах или количество блоков ФС, которое пользователь превзойти не может, иначе его работа в системе блокируется
2. **гибкий** – некоторое значение устанавливается в виде лимита, а с ним сравнивается еще одно значение - **счетчик 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 ФС).
    - Если вычисленное значение не превосходит гибкий лимит, то счетчик предупреждений сбрасывается на начальное значение, и пользователь продолжает свою работу
    - Если же превосходит, то значение счетчика предупреждений уменьшается на единицу, затем происходит проверка равенства его значения нулю:
        - 0 - вход пользователя в систему блокируется
        - > 0 - пользователь получает предупреждение о том, что соответствующий гибкий лимит израсходован, после чего пользователь может работать дальше

*наибольшая эффективность при совместной работе этих параметров:

- Если в системе только гибкий лимит, то пользовательский процесс может «забить» все свободное пространство файловой системы. Данную проблему решает жесткий лимит.
- Если же в системе только жесткий лимит, то пользователь может получить отказ от системы из-за «неумышленного» нарушения лимита (например, из-за ошибки в программе был сформирован очень большой файл). Данную проблему решает жесткий лимит.

Надежность ФС - системные данные ФС должны обладать избыточной информацией (в случае аварийной ситуации минимизировать ущерб - потерю информации).
Решение: системы **архивирования**, или **резервного копирования** (происходит как автоматически по инициативе некоторого программного робота, так и по запросу пользователя).

Проблемы резервного копирования:

1. целиком каждый раз копировать всю ФС неэффективно и дорого – нужно минимизировать объем копируемой информации без потери качества:
    a. **избирательное копирование**: намеренно не копируются файлы, которые заведомо восстанавливаются:
        - исполняемые файлы ОС, систем программирования, прикладных систем и т.д (считается, что есть дистрибутивные носители, с которых можно восстановить эти файлы)
        - исполняемые файлы, если для них имеется в наличии дистрибутив или исходных код, который можно откомпилировать и получить данный исполняемый файл
        - файлы пользователей определенных категорий (если их можно достаточно легко восстановить, переписав).
    b. **инкрементном архивировании**. При первом архивировании создается полная копии всех файлов — это т.н. **мастер-копия** (**master-copy**). Каждая следующая копия будет включать в себя только те файлы, которые изменились или были созданы с момента предыдущего архивирования.
    c. **компрессия**. Дилемма: с одной стороны сжатие данных при архивировании дает выигрыш в объеме резервной копии, с другой стороны компрессия крайне чувствительна к потере информации. Потеря или приобретение лишнего бита в сжатом архиве может привести к порче всего архива.

2. **копирование на ходу**: во время резервного копирования какого-то файла пользователь начинает с ним работать (модифицировать, удалять и т.п.) - необходимо грамотно выбирать моменты для архивирования: ночные часы, когда в системе почти нет пользователей.

3. **распределенное хранение резервных копий**. Всегда желательно иметь две копии, причем храниться они должны в совершенно разных местах.

Стратегии **копирования:**

1. **физическое** - поблочное копирование данных с носителя («один в один»)
   - неэффективно, поскольку копируются и свободные блоки
2. **интеллектуальное** физическое копирование лишь занятых блоков
   - обработка дефектных блоков: сталкиваясь при копировании с физически дефектным блоком, невозможно связать данный блок с конкретным файлом
3. **логическое** - копирование не блоков, а файлов (например, файлов, модифицированных после заданной даты)

Проверка и контроль целостности ФС (не файлов)

Если произошел сбой (например, сломался ЦП или ОП), то гарантированно потери будут, и эти потери будут двух типов:
1. потеря актуального содержимого одного или нескольких открытых файлов
2. может нарушиться корректность системной информации (более существенно)

Для выявления непротиворечивости и исправления возможных ошибочных ситуаций ФС использует избыточную информацию (данные тем или иным образом (явно или косвенно) дублируются)
1. в системе формируются две таблицы с размером по количеству блоков ФС: **таблица занятых блоков** и **таблицей свободных блоков**
2. изначально содержимое таблиц обнуляется
3. система запускает процесс анализа блоков на предмет их незанятости: для каждого свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных блоков
4. запускается аналогичный процесс, но уже для анализа индексных узлов: для каждого блока, номер которого встретился в индексном дескрипторе, увеличивается на 1 соответствующая ему запись в таблице занятых блоков
5. запускается процесс анализа содержимого этих таблиц и коррекции ошибочных ситуаций

Возможные ситуации:
- Если при анализе таблиц для каждого номера блока сумма содержимого ячеек с данным номером дает 1, то считается, что система не выявила противоречий.
- Если же находится блок, о котором нет информации ни в таблице свободных, ни в таблице занятых блоков (стоят нули), то считается, что этот блок потерян из списка свободных блоков - не катастрофически, не требует оперативного разрешения: информацию о данном блоке система может внести в таблицу свободных блоков спустя некоторое время.
- Если в ходе анализа блок получается свободным, но индекс свободности его больше 1, то считается, что нарушен список свободных блоков, и начинается процесс пересоздания таблицы свободных блоков.
- Если же возникает аналогичная ситуация, но уже для таблицы занятых блоков, то это означает, что данным файлом владеют несколько файлов, что является ошибкой. Автоматически определить, какой из файлов ошибочно хранит ссылку на этот блок, не представляется возможным: необходимо анализировать содержимое этих файлов. Для разрешения данной проблемы ФС:
  1. создает копии конфликтующих файлов (с именами соответственно $Name1^2$ и $Name2^2$)
  2. удаляет файлы с исходными именами Name1 и Name2
  3. запускает процесс переопределения списка свободных блоков
  4. переименовывает копии с фиксацией факта их возможной некорректности

Для проверки корректности ФС может выполняться проверка соответствия числа реального количества жестких связей тому значению, которое хранится среди атрибутов файла. Если эти значения совпадают, то считается, что данный файл находится в корректном состоянии, иначе происходит коррекция атрибута-счетчика жестких связей.

Примеры реализаций ФС

Рассмотрим файловые системы, реализованные в ОС Unix. Два главных понятия – файл (всё то, с чем мы работаем) и процесс. Реализованы унифицированные интерфейсы доступа и организации информации.

Достоинства:
1. одна из первых ОС, в которой была реализована иерархическая файловая система.
2. одна из первых ОС, открытых для расширения набора команд системы. До нее команды, которые были доступны пользователю, - наборы жестких правил взаимодействия с системой (как современные интерпретаторы команд), модифицировать этот набор мог только разработчик, который, модифицируя, по сути, создавал новую систему. В ОС Unix все исполняемые команды принадлежат одной из двух групп: команды, встроенные в интерпретатор команд (например, pwd, cd и т.д.), и команды, реализация которых представляется в виде исполняемых файлов, расположенных в одном из каталогов ФС (это либо исполняемый бинарный файл, либо текстовый файл с командами для исполнения интерпретатором команд). Тогда, варьируя правами доступа к файлам, уничтожая при необходимости или добавляя новые исполняемые файлы, пользователь способен самостоятельно выстраивать функциональное окружение, необходимое для решения своих задач.
3. элегантная организация идентификации доступа и прав доступа к файлам

Организация файловой системы ОС Unix. Виды файлов. Права доступа

Файл ОС Unix — это специальным образом именованный набор данных, размещенных в ФС.

Типы файлов:

1. **обычный или регулярный файл** – с ним пользователь регулярно имеет дело в повседневной работе (текстовый файл, исполняемый файл и т.п.)
2. **каталог** (**directory**) - содержит имена и ссылки на атрибуты, которые содержатся в данном каталоге
3. **специальный файл устройств** - каждому устройству, с которым работает ОС Unix, соответствует файл данного типа. Через имя файла устройства происходит обращение к устройству, а через содержимое данного файла (которое достаточно специфично) можно обратиться к конкретному драйверу этого устройства
4. **именованный канал**, или **FIFO-файл** (**named pipe**, **FIFO file**)
5. **файл-ссылка**, или **символическая связь** (**link**)
6. **сокет** (**socket**)

права доступа к файлу – характеристика файла, регламентирующая чтение содержимого файла, запись в него и исполнение для разных категорий пользователей (владельца файла, группы, к которой принадлежит владелец файла, исключая владельца, всех остальных пользователей системы без указанной группы владельца и него) (для регулярных файлов), для других типов интерпретация прав доступа отличается: при работе с каталогом его владельцу, члену соответствующей группы, и всем остальным пользователям может разрешаться чтение, запись и выполнение. Однако, эти разрешения интерпретируются не так, как для обычных файлов. Разрешение на чтение из каталога означает, что разрешено открытие каталога и чтение из него. Разрешение на запись предоставляет возможность создавать и уничтожать файлы. Разрешение на выполнение - система может выполнять поиск в каталоге с целью обнаружения какого-либо файла. Если вместо простого имени используется составное, то в каждом из указанных в нем каталогов выполняется поиск имени файла, стоящего следующим в составном имени.

Логическая структура каталогов

ОС Unix является иерархической древовидной файловой системой. Система предполагает, что в корневом каталоге всегда расположен некоторый файл, в котором размещается код ядра ОС. Вообще говоря, в корневом каталоге можно размещать любые файлы (с учетом прав доступа), но система предполагает наличие совокупности каталогов с предопределенными именами.

Основные каталоги системы:

1. В **/bin** находятся команды общего пользования (точнее говоря, исполняемые файлы этих команд)
2. В **/etc** находятся системные таблицы и команды, обеспечивающие работу с этими таблицами (например, таблица (файл) passwd с инфо-й о зарегистрированных в системе пользователях)
3. В **/tmp** находятся временные файлы (система и пользователи размещают тут файлы на времнннное хранение; нет гарантии сохранения при перезагрузке системы)
4. **/mnt** традиционно используют для монтирования различных ФС к данной системе. Операция монтирования - корень монтируемой ФС ассоциируют с данным каталогом (или с одним из его подкаталогов), после чего доступ к файлам подмонтированной ФС осуществляется уже через этот каталог (т.н. **точку монтирования**).
5. **В /dev** находятся специальные файлы устройств. Все устройства, с которыми работает ОС, именуются посредством имен этих специальных файлов устройств.
6. В **/usr** находится пользовательская информация:
    a. **/usr/lib** содержит инструменты работы пользователей, не относящихся напрямую к взаимодействию с ОС (например системы программирования, С-компилятор, С-отладчик и т.п.)
    b. **/usr/include** содержит файлы заголовков (include-файлы) с расширением *.h. Именно тут препроцессор С-компилятора ищет соответствующие файлы заголовков, указанные в программе
    c. **/usr/bin** содержит команды, которые введены на данной ВС (например, команды, связанные с непосредственной деятельностью организации)
    d. **/usr/user** размещаются домашние каталоги зарегистрированных в системе пользователей.

Внутренняя организация ФС: модель версии System V

Структура ФС в ОС Unix версии System V - **s5fs**. ФС находится в **системном разделе** (надо отличать от разделов с другими ФС схожей организации, которые можно примонтировать к данной системе), начиная с нулевого блока и заканчивая некоторым фиксированным блоком. Часть, которую занимает эта ФС состоит из трех подпространств:

1. **Суперблок** – часть ФС, которая резидентно находится в ОП. Он хранит:
   a. данные, определяющие статические параметры и характеристики данной ФС:
      i. информация о размере блока файла
      ii. информация о размере всей ФС в блоках или байтах
      iii. информация о количестве индексных дескрипторов в системе
   b. информацию об оперативном состоянии ФС
   c. информацию о наличии свободных ресурсов ФС - свободных блоков в рабочем пространстве ФС и свободных индексных дескрипторов (массив номеров свободных блоков и массив индексных дескрипторов)
2. **область индексных дескрипторов** - системные структуры данных фиксированного размера, содержащих комплексную информацию о размещении, актуальном состоянии и содержимом конкретного файла.
3. **блоки файлов** (**рабочее пространство** ФС) с содержимым, а также системная информация, которая не поместилась в суперблоке и области индексных дескрипторов.

Работа с массивами номеров свободных блоков

Изначально номера всех свободных блоков файловой системы выстраиваются в единый связный список, который размещается в нескольких блоках. Первый блок располагается в суперблоке (а значит, в ОП). Каждый блок хранит номера свободных блоков, а также номер следующего блока данного массива. При запросе на получение свободного блока:

1. поиск в первом блоке массива ячейки с содержательной (ненулевой) информацией
2. обнуление найденной ячейки
3. блок с найденным номером выдается в ответ на запрос
4. если это была последняя ячейка блока, ссылающаяся на следующий блок массива, то предварительно содержимое этого блока загружается в суперблок и используется уже как первый блок этого массива.

*Если же какой-то блок освобождается, то выполняются противоположные действия в обратном порядке.

** Если нет свободных блоков, - тогда нет и номеров свободных блоков, а значит, нет и блоков, хранящих эти номера, т.е. файловая система занята на 100%.

Работа с массивом свободных индексных дескрипторов

Массив номеров свободных индексных дескрипторов состоит из фиксированного количества элементов и заполнен номерами свободных индексных дескрипторов.

<u>Освобождение индексного дескриптора (удаление файла) - обращение к данному массиву:</u>

1. если в массиве есть свободные места, то номер освободившегося индексного дескриптора записывается в первое встретившееся свободное место массива
2. если нет - номер дескриптора «забывается».

<u>Создание файла - обращение к массиву:</u>

1. если он не пуст, то из него изымается первый содержательный элемент, который представляет собой номер свободного индексного дескриптора
2. если же пуст, а в суперблоке присутствует информация о наличии свободных индексных дескрипторов, то система запускает процесс обновления рассматриваемого массива:
   a. процесс обращается к области индексных дескрипторов
   b. последовательно перебирает их и в зависимости от содержимого делает однозначный вывод о занятости или свободности дескриптора
   c. номера свободных индексных дескрипторов процесс помещает в массив.

Рассмотренные массивы свободных блоков и свободных индексных дескрипторов исполняют роль специализированных КЭШей: происходит буферизация обращений к системе за свободным ресурсом.

Индексные дескрипторы. Адресация блоков файла

индексный дескриптор – системная структура данных с атрибутами файла и всей оперативной информацией об организации и размещении данных. Между содержимым файла и его индексным дескриптором существует **взаимнооднозначное соответствие**.

*содержимое файла не обязательно размещается в рабочем пространстве файловой системы: существуют некоторые типы файлов, для которых содержимое хранится в самом индексном дескрипторе. Примером тут может послужить тип специального файла устройств.

Для каждого индексного дескриптора существует, по меньшей мере, одно имя, зарегистрированное в каталогах ФС (древовидность ФС - не с точки зрения размещения файла, а с точки зрения размещения имен файлов)

Индексный дескриптор хранит:

1. информацию о типе файла, правах доступа, информацию о владельце файла, размере файла в байтах, количестве имен, зарегистрированных в каталогах файловой системы и ссылающихся на данный индексный дескриптор. В частности, признаком свободного индексного дескриптора является нулевое значение последнего из указанных атрибутов.
2. Различную статистическую информацию о времени создания, времени последней модификации, времени последнего доступа.
3. массив блоков файла, который состоит из 13 элементов: первые 10 используются для указания номеров первых десяти блоков файла, оставшиеся три элемента используются для организации косвенной адресации блоков (11 ссылается на массив из N номеров блоков файла, 12 — на массив из N ссылок, каждая из которых ссылается на массив из N блоков файла, 13 используется уже для трехуровневой косвенной адресации блоков)

Пример: размер блока равен 512 байтам:

- Если блоков файла больше 10, то сначала используется косвенная адресация первого уровня - в 11 элементе хранится номер блока, состоящий из 128 номеров блоков файла, которые следуют за первыми десятью блоками (11 элемент адресует 11-ый – 138-ой блоки файла).
- Если блоков больше, чем 138, то используется косвенность второго уровня - 12 элемент массива содержит номер блока, в котором могут находиться до 128 номеров блоков, в каждом из которых может находиться до 128 номеров блоков файла.
- Если размеры файла оказываются настолько большими, что для хранения номеров его блоков не хватает двойной косвенной адресации, используется 13 элемент массива и косвенная адресация третьего уровня.

В данном случае максимальный размер файла может достигать $(10+128+128^2+128^3)*512$ байт. На сегодняшний день файловые системы с таким размером блока не используются, наиболее типичны размеры блока 4, 8, и т.д. вплоть до 64 кбайт.

\+ компактно и эффективно, поскольку для обращения к блоку файла с использованием тройной косвенности потребуется всего три обмена, а если учесть, что в системе реализована буферизация блочных обменов, то накладные расходы становятся еще меньше.

Файл-каталог

Каталог файловой системы версии System V — это файл специального типа; его содержимое так же, как и у регулярных файлов, находится в рабочем пространстве ФС и по организации данных ничем не отличается от организации данных регулярных файлов.

- каждая запись в нем имеет фиксированный размер (будем считать,16 байт). Первые два байта - номер индексного дескриптора файла, а оставшиеся 14 - имя файла.
- при создании каталога он получает две предопределенные записи, которые невозможно модифицировать и удалять: ссылка на сам этот каталог (унифицированное имя “**.**”, хранит номер индексного дескриптора данного файла-каталога) и ссылка на родительский каталог (унифицированное имя “**..**”, хранится номер индексного дескриптора родительского каталога)

* многие более развитые ФС ОС Unix поддерживают **установление связей** между индексным дескриптором и именами файла:

- жесткие связи - с одним индексным дескриптором можно связать два и более равноправных имени. При удалении имени, участвующего в жесткой связи:
  - удаляется имя из каталога
  - уменьшается счетчик жестких связей в индексном дескрипторе
  - если он обнулился, то содержимое файла удаляется и данный индексный дескриптор освобождается
- символические связи - создается файл специального типа - ссылка. Он содержит полный путь к тому файлу, на который ссылается данный файл-ссылка. Используя такую косвенную адресацию, можно добраться до целевого файла - ассиметричное именование (права файла ссылки будут отличаться от прав файла, на который он ссылается).

Достоинства и недостатки ФС модели System V

+ иерархичная
+ за счет использования системного кэширования оптимизирована работа с массивом свободных блоков и свободных индексных дескрипторов
+ удачное решение организации блоков файлов за счет использования «нарастающей» косвенности адресации
- в суперблоке концентрируется ключевая информация файловой системы => потеря суперблока приводит к достаточно серьезным проблемам
- в суперблоке концентрируется ключевая информация файловой системы => несмотря на то, что суперблок резидентно размещается в ОП, система периодически «сбрасывает» его копию на диск — это делается для того, чтобы при сбое минимизировать потери актуальной информации из суперблока. Это, в свою очередь, означает, что система регулярно обращается к одной и той же точке дискового пространства, и, соответственно, вероятность выхода из строя именно данной области диска со временем сильно увеличивается.
- фрагментация блоков файла по диску (при интенсивной работе ФС (когда в ней со временем создается, модифицируется и уничтожается достаточно большое число файлов) блоки одного файла оказываются разбросанными по всему доступному дисковому пространству) => если потребуется прочитать последовательные блоки файла (что бывает достаточно часто), то головка жесткого диска начинает совершать довольно много механических передвижений, что отрицательно сказывается на эффективности работы файловой системы.
- ограничение, накладываемое на длину имени файла (6, 8, 14 байт)

Внутренняя организация ФС: модель версии Fast File System (FFS) BSD

Раздел - последовательность дисковых цилиндров, разбитая на порции фиксированного размера – кластеры. Разбиение происходит аппаратно-зависимо так, чтобы суперблоки не оказывались на «опасно близком» расстоянии (например, на одной поверхности). Это обеспечивает большую надежность ФС. В каждом из образовавшихся кластеров размещается:

1. копия суперблока,
2. блоки файлов (которые мы назвали рабочим пространством ФС),
3. информация об индексных дескрипторах, ассоциированных с данным кластером
4. информация о свободных ресурсах этого кластера

Стратегии размещения. Концепции.

1. **оптимизация размещения каталога**. При создании каталога система осуществляет поиск кластера, наиболее свободного в данный момент с точки зрения использования индексных дескрипторов (ищутся кластеры, количество свободных индексных дескрипторов в которых превосходит некоторую среднюю величину, и среди найденных кластеров выбирается кластер с наименьшим количеством каталогов)
2. **равномерность использования блоков данных**. Во время создания файла он делится на несколько частей. Часть файла, которая имела непосредственную адресацию из индексного дескриптора, по возможности размещается в том же кластере, что и индексный дескриптор. Оставшиеся части файла делятся на равные порции, которые ФС размещает в отдельных кластерах.
   + борьба с фрагментацией файла по разделу: файл либо целиком размещается в одном кластере, либо размещается в нескольких кластерах, но большими фрагментами подряд идущих блоков.
3. **размещение последовательных блоков файлов**: если необходимо прочитать два последовательных блока с магнитного диска => нужны два последовательных системных вызова. Между окончанием физического считывания первого блока и началом считывания второго потратится некоторое время Δt на накладные расходы - вход и выход из системного вызова. Хоть и мало, но за данный промежуток диск успеет повернуться на угол $\omega * \Delta t$ (где ω — скорость вращения диска). Если следующий второй блок расположен на диске непосредственно за первым, то за время Δt головка пропустит начало второго блока, и когда будет предпринята попытка физически прочесть второй блок, то придется ожидать полного оборота диска, что является относительно долгим по времени. Чтобы избежать подобных накладных расходов, связанных с необходимостью ожидать полного оборота диска, необходимо расположить второй блок с некоторым отступом от первого. В этом и заключается технологическое размещение блоков на диске.

Внутренняя организация блоков

Размер блока в файловой системе FFS может варьироваться в достаточно широком диапазоне: предельный размер блока — 64 Кбайт. Но какой оптимальный? В данной ФС выбрано увеличение размера блока:
+ уменьшается фрагментация файла по диску
+ уменьшаются накладные расходы при чтении подряд идущих данных файла (эффективнее считать за 1 раз большую порцию информации, чем два раза считать по «половинке»)
- сильная внутренняя фрагментация
Решение: еще один уровень структурной организации: каждый блок ФС поделен на фиксированное количество **фрагментов** (обычно число фрагментов в блоке кратно степени 2 — 2, 4, 8 и т.д.).

Размещение файла по блокам ФС: пусть все блоки файла кроме последнего (он тоже, но необязательно) целиком заполнены содержимым и их номера хранятся среди атрибутов файла. А вот для последнего блока помимо номера хранятся номера занятых в нем фрагментов, принадлежащих данному файлу. Способы представления информации о блоках и фрагмента: двоичная маска, номер первого фрагмента в этом блоке, занятым данным файлом (количество фрагментов тогда можно вычислить на основании длины файла в байтах), и т.д.

Выделение пространства для файла

Пример: пусть блок ФС поделен на 4 фрагмента, а в системе хранятся файлы petya.txt и vasya.txt, для которых в соответствующих индексных дескрипторах хранится информация об их размерах и номерах блоков, принадлежащих файлам, в виде стартовых фрагментов. Соответственно, файл petya.txt расположен в нулевом (стартовый фрагмент № 00), первом (стартовый фрагмент № 04) и втором блоке (стартовый фрагмент № 08). Если учесть длину файла (5120 байт), то получается, что во втором блоке этот файл занимает 08 и 09 фрагменты. Файл vasya.txt расположен в третьем блоке (стартовый фрагмент № 12), четвертом (стартовый фрагмент № 16) и втором блоке (стартовый фрагмент № 10); при этом во втором блоке файлу принадлежит только 10-ый фрагмент (т.к. размер файла 4608 байт).

Итак, очевидно, что данная система нарушает концепцию ФС ветви System V, в которой каждый блок мог принадлежать только одному файлу; в FFS последний блок можно разделять между различными файлами.
Если, например, размер файла petya.txt увеличивается на столько, что конец файла не помещается в 08 и 09 фрагментах, то система начинает поиск блока с тремя **подряд идущими** свободными фрагментами. (Соответственно, если размер файла увеличивается на большую величину, то сначала для него отводятся полностью свободные блоки, в которых файл занимает все фрагменты, а для размещения последних фрагментов ищется блок с соответствующим числом подряд идущих свободных фрагментов.) Когда система находит такой блок, то происходит перемещение последних фрагментов файла petya.txt в этот блок.

Структура каталога FFS

Каталог файловой системы FFS:

- позволяет использовать имена файлов, длиной до 255 символов
- состоит из записей переменной длины. Начальная запись содержит:
    - номер индексного дескриптора,
    - размер записи (ссылку на последний элемент записи)
    - длину имени файла
    - дополненное до кратности в 4 байта имя файла
- если происходит удаление файла из каталога, то освобождающееся пространство, занимаемое раньше записью данного файла, присоединяется к предыдущей записи - размер предыдущей записи увеличивается, но длина хранимого в ней имени не меняется (остается реальной)
- если происходит удаление первой записи, то номер индексного дескриптора в этой записи обнуляется.

+ можно не заботиться о высвобождаемом пространстве внутри файла-каталога: получаемые при удалении «дыры» ликвидируются не за счет компрессии, а за счет тривиального «склеивания» с предыдущей записью.
+ можно использовать имена файлов произвольной длины вплоть до 255 символов (удобно для пользователя)
- система несет накладные расходы как по использованию дискового пространства (в каталогах присутствует внутренняя фрагментация), так и по времени (осуществление поиска в системах с фиксированными размерами записей в большинстве случаев эффективнее, чем в системах с переменными размерами записей).

Блокировка доступа к содержимому файла

Организация ФС ОС Unix позволяет:

- открывать и работать с одним и тем же файлом произвольному числу процессов
- многократно открыть один и тот же файл в рамках одного процесса

При этом система поддерживает модель синхронизации работы с файлами - системный вызов **fcntl()**, который может обеспечивать блокировку как файла в целом, так и отдельных областей внутри файла Типы блокировок:

1. **исключающая блокировка** (**exclusive lock**) — это «жесткая» блокировка: другой процесс не сможет осуществить операции обмена с данной областью (будет либо приостановлен в ожидании разблокирования области, либо получит отказ, в зависимости от установленного режима работы). Это блокировка с монополизацией: области с исключающими блокировками пересекаться не могут.
2. **распределенная блокировка** (**shared lock**), или «мягкая», рекомендательная блокировка: другие процессы при работе могут не обращать внимания на нее (чтение и запись информации из блокированной области разрешены). Области с рекомендательными блокировками могут пересекаться.

Управление оперативной памятью - задачи:

1. осуществление **контроля использования ресурсов** (учет состояния каждой доступной в системе единицы памяти (свободна она или распределена))
2. **выбор стратегии распределения памяти** (какому процессу, в течение какого времени и в каком объеме должен быть выделен соответствующий ресурс)
3. **конкретное выделение ресурса тому или иному потребителю** (после решения, какому процессу сколько выделить памяти и на какое время (в соответствии с наличием ресурса), следует операция непосредственного выделения - для предоставляемого ресурса идет корректировка системных данных (например, изменение статуса занятости), а затем выдача его потребителю.
4. **выбор стратегии освобождения памяти**: с одной стороны, это окончательное освобождение памяти (завершения процесса и высвобождения ресурса), с другой стороны это действия по освобождению физической памяти из-под какого-то процесса за счет откачивания во внешнюю память (на освободившееся место поместить данные другого процесса). Тут нужно решить, память какого процесса необходимо откачать и какая именно область памяти у выбранного процесса будет освобождаться. (весь процесс откачивать неэффективно)

Модели распределения ОП:

1. Одиночное непрерывное распределение. Все адресное пространство подразделяется на два компонента: в одной части памяти располагается и функционирует ОС, а другая часть выделяется для выполнения прикладных процессов. Границу (для корректной работы) определяет один специальный регистр: если получаемый исполнительный адрес оказывается меньше значения этого регистра, то это адрес в пространстве ОС, иначе – в пространстве процесса.

+ очень простая
+ не возникает особых организационных трудностей.
+ может сочетаться с аппаратной поддержкой двух режимов функционирования: пользовательского и режима ОС. Если в режиме пользователя происходит попытка обратиться в область памяти ОС – прерывание
+ минимальные аппаратные требования или отсутствие таковых
- неэффективное использование физического ресурса: часть памяти, выделяемой под процесс, никогда реально не используется
- процесс занимает всю память полностью на все время выполнения (но реально оказывается, что зачастую обращения процесса к памяти происходят в достаточно локализованные участки, а равномерное обращение ко всему адресному пространству процесса случается очень редко)
- неэффективна: под адресное пространство процесса отводится сразу все необходимое физическое пространство, хотя реально процесс работает лишь с локальными участками.
- модель жестко ограничивает размер прикладного процесса

2. **Распределение неперемещаемыми разделами**. Все адресное пространство ОП делится на две части: под ОС и под прикладные процессы, причем это пространство заблаговременно делится на N частей (назовем их **разделами**), каждая из которых в общем случае имеет произвольный фиксированный размер (на уровне ОС). Соответственно, очередь прикладных процессов разделяется по этим разделам.

a. единственная сквозная очередь, которая по каким-то соображениям распределяется между этими разделами
b. с каждым разделом ассоциируется своя очередь и поступающий процесс сразу попадает в одну из этих очередей

Способы аппаратной реализации данной модели:

a. **использование двух регистров границ**, один из которых отвечает за начало, а второй — за конец области прикладного процесса. Выход за ту или иную границу ведет к возникновению прерывания по защите памяти.
b. **механизм ключей защиты** (PSW — process[or] status word), которые находятся в слове состояния процесса и в слове состояния процессора. Каждому разделу ОЗУ ставится в соответствие некоторый ключ защиты. Если аппаратура поддерживает, то в процессоре имеется слово состояния, в котором может находиться ключ защиты доступного в данный момент раздела. Соответственно, у процесса также есть некоторый ключ защиты, который тоже хранится в некотором регистре. Если при обращении к памяти эти ключи защиты совпадают, то доступ считается разрешенным, иначе возникает прерывание по защите памяти.

Модель с N очередями. Сортировка входной очереди процессов по отдельным очередям к разделам: приходящий процесс размещается в разделе минимального, но достаточного размера.
- в общем случае не гарантируется равномерная загрузка всех очередей, что ведет к неэффективности использования памяти. Возможны ситуации, когда к некоторым разделам имеются большие очереди, а к разделам большего размера очередей вообще нет, т.е. возникает проблема недозагрузки некоторых разделов.

Другая модель с единой очередью процессов
+ более гибкая
- проблема выбора процесса из очереди для размещения его в только что освободившийся раздел.

Решения:

1. из очереди выбирается первый процесс, помещающийся в освободившемся разделе
+ просто и не требует просмотра всей очереди процессов
- возможно несоответствие размеров процесса и раздела, когда процесс намного меньше освободившегося раздела - маленькие процессы будут «подавлять» более крупные процессы, которые могли бы поместиться в освободившемся разделе

2. искать в очереди процесс максимального размера, помещающийся в освободившийся раздел.
- требует просмотра всей очереди процессов
+ достаточно эффективно обходит проблему фрагментации раздела
- дискриминация «маленьких» процессов при выборе очередного процесса для постановки на исполнение.

Решение последней проблемы: для каждого процесса имеется счетчик дискриминации. **Дискриминация** - в системе освободился раздел, достаточный для загрузки некоторого процесса, но система планирования ОС его пропустила. Соответственно, при каждой такой дискриминации из счетчика дискриминации данного процесса вычитается единица. Тогда при просмотре очереди планировщик сначала проверяет значение этого счетчика: если оно равно равно нулю и процесс помещается в освободившемся разделе, то планировщик обязан загрузить данный процесс в этот раздел.
+ простота аппаратных средств организации мультипрограммирования (например, использование двух регистров границ)
+ простота используемых алгоритмов.
- внутреннюю фрагментацию в разделах, поскольку зачастую процесс, загруженный в раздел, оказывается меньшего размера, чем данный раздел
- ограничение предельного размера прикладных процессов размером максимального физического раздела ОЗУ
- весь процесс размещается в памяти, что может привести к неэффективному использованию ресурса (поскольку, как упоминалось выше, зачастую процесс работает с локализованной областью памяти).
*В модели с N очередями нет никаких дополнительных требований к реализации не возникает (можно так все организовать, что подготавливаемый процесс в зависимости от его размера будет настраиваться на соответствующий раздел).
**В модели с единой очередью процессов появляется требование к перемещаемости кода, это же требование добавляется и к аппаратной части. В данном случае это регистр базы, который может совпадать с одним из регистров границ

3. **Распределение перемещаемыми разделами**: разрешается загрузка произвольного (нефиксированного) числа процессов в ОП, и под каждый процесс отводится раздел необходимого размера. Соответственно, система допускает перемещение раздела, а, следовательно, и процесса. Таким образом, хоть ОП и становится все более и более фрагментированным с каждым завершенным процессом, возможность перемещения исполняемого кода дает использовать компрессию и избавиться от фрагментации.

Стратегии компрессии:

- **локальная** - для высвобождения пространства передвигает небольшое количество процессов (например, два процесса)

- **глобальная** - в некоторый момент система приостанавливает выполнение всех процессов и начинает их перемещать, например, к начальному адресу ОП, тогда в конце ОЗУ окажется вся свободная память
  - требуются аппаратные средства защиты памяти (регистры границ или же ключи защиты) и аппаратные средства, позволяющие осуществлять перемещение процессов (в большинстве случаев для этих целей используется регистр базы, который в некоторых случаях может совпадать с одним из регистров границ).
  + нет фрагментации памяти.
  * для систем, ориентированных на работу в мультипрограммном пакетном режиме (когда почти каждый процесс является более или менее большой вычислительной задачей), задача дефрагментации (или компрессии) не имеет существенного значения, поскольку для многочасовых вычислительных задач редкая минутная приостановка для совершения компрессии на эффективность системы не влияет. Соответственно, данная модель хорошо подходит для такого класса систем.
  **Если же система предназначена для обработки большого потока задач пользователей, работающих в интерактивном режиме, то компрессия будет достаточно частой, а продолжительность компрессии будет достаточно большой => низкая эффективность подобной системы.
  - ограничение предельного размера прикладного процесса размером физической памяти
  - накладные расходы, связанные с компрессией

3. Страничное распределение. Все адресное пространство может быть представлено совокупностью блоков фиксированного размера - **страницами**. Есть **виртуальное адресное пространство** — это то пространство, с адресами которого оперирует программа, и **физическое адресное пространство** — это то пространство, которое есть в наличии в компьютере. Соответственно, при страничном распределении памяти существуют программно-аппаратные средства, устанавливающие соответствие между виртуальными и физическими страницами. Механизм преобразования виртуального адреса в физический: берется номер виртуальной страницы и заменяется соответствующим номером физической страницы. Виртуальный адрес состоит из номера виртуальной страницы (**VP**) и смещения в ней (**offset**). Используется **таблица страниц**, которая целиком является аппаратной.

Однако современные машины обладают очень большим объемом виртуального адресного пространства => таблица страниц будет очень большой. Варианты установления соответствия в этом случае:

1. полное размещение таблицы преобразования адресов в аппаратной части компьютера, но это решение применимо лишь в тех системах, где количество страниц незначительное.
   - для большой таблицы стоимость аппаратной поддержки высока, а при необходимости полной перезагрузки таблицы при смене контекстов начинается ужас.
   + скорость преобразования высока
2. хранение таблицы в ОП
   - каждое преобразование происходит через обращение к ОЗУ, что совсем неэффективно.
   Требования к аппаратуре: должен быть регистр, ссылающийся на начало таблицы в ОЗУ, а также поддержка обращения в ОП по адресу, хранящемуся в указанном регистре, извлечения данных из таблицы и осуществления преобразования.

   Возможно оптимизировать рассмотренный подход за счет использования кэширования L1 или L2. *поскольку к таблице страниц происходит постоянное обращение, странички из данной таблицы «зависают» в КЭШе
   - но, если в компьютере используется всего один КЭШ и для потока управления, и для потока данных, то в этом случае через него направляется еще и поток преобразования страниц, что снижает эффективность системы.
   - для больших таблиц это дорого – хранить всю таблицу в ОП

В принципе, можно хранить не всю, а только оперативную часть этой таблицы, но тогда то возникают проблемы, связанные со сменой процессов: необходимо будет часть таблицы откачивать во внешнюю память, а часть — наоборот, подкачивать, что является достаточно трудоемкой задачей - проблема организации эффективной работы с таблицей страниц, чтобы возникающие накладные расходы не приводили к деградации системы.

Современные решения гибридны и основаны на иерархической организации таблиц. Типовая структура записи таблицы страниц содержит информацию о номере физической страницы, а также совокупность атрибутов, необходимых для описания статуса данной страницы:

- атрибут присутствия/отсутствия страницы
- атрибут режима защиты страницы (чтение, запись, выполнение)
- флаг модификации содержимого страницы
- атрибут, характеризующий обращения к данной странице, чтобы иметь возможность определения «старения» страницы
- атрибут блокировки кэширования

TLB-таблица (Translation Look-aside Buffer — таблица быстрого преобразования адресов. Подразумевается наличие аппаратной таблицы относительно небольшого размера (порядка 8 – 128 записей). Данная таблицы концептуально содержит три столбца:

1. номер виртуальной страницы
2. номер физической страницы, в которой находится указанная виртуальная страница
3. атрибуты

1. Страница изымает из виртуального адреса номер виртуальной страницы и осуществляет оптимизированный поиск (не последовательный, а параллельный) этого номера по TLB-таблице:
   a. Если искомый номер найден, система проверяет соответствия атрибутов, и если проверка успешна, то происходит подмена номера виртуальной страницы номером физической страницы, и, таким образом, получается физический адрес.
   b. Если же не найден – промах:
      Модели обработки промаха:
      - без прерывания: система самостоятельно, имея регистр начала программной таблицы страниц, обращается к этой таблице и осуществляет в ней поиск)
      - с прерыванием: управление передается ОС, которая затем начинает работать с программной таблицей страниц (увеличение накладных расходов)
        i. система обращается в программную таблицу
        ii. выкидывает самую старую запись из TLB
        iii. загружает в нее найденную запись из программной таблицы
        iv. вычисляется физический адрес

TLB-таблица является некоторым КЭШем

- проблема, связанная с большим размером таблицы страниц: при смене контекста система так или иначе обязана поменять эту таблицу, а также содержимое TLB, т.к. там хранится информацию об одном процессе, а во-вторых, это проблема, связанная с организацией мультипроцессирования, — необходимо решать, где размещать все таблицы различных процессов.

Решение, позволяющее снизить размер таблицы страниц:

1. **иерархическая организация таблицы страниц**. В этом случае информация о странице представляется не в виде одного номера страницы, а в виде совокупности номеров, используя которые, можно получить номер соответствующей физической страницы, посредством обращения к соответствующим таблицам, участвующим в иерархии (это может быть 2-х-, 3-х- или даже 4-хуровневая иерархия).

   обычно: 32-разрядный виртуальный адрес = 20 на номер виртуальной страницы и 12 на смещение двухуровневая иерархическая организация = 10 на индекс во «внешней» таблице групп (или кластеров) страниц, 10 на индекс в таблице второго уровня + 12 на смещение в физической странице.

   Соответственно, чтобы получить номер физической страницы необходимо по индексу во «внешней» таблице страниц найти начальный адрес таблицы второго уровня, затем по этому адресу и по индекса по таблице второго уровня находится нужная запись в таблице страниц второго уровня, которая уже и содержит номер соответствующей физической страницы.
   + всю таблицу страниц хранить в памяти необязательно: из-за принципа локализации будет достаточно хранить сравнительно небольшую «внешнюю» таблицу групп страниц и некоторые таблицы второго уровня (они также имеют незначительные размеры); все необходимые таблицы второго уровня можно подкачивать по мере надобности.
   - эффективность системы начинает сильно падать с ростом числа уровней иерархии (из-за различных накладных расходов), поэтому обычно число уровней ограничено четырьмя.

2. использование **хеширования** (на использовании **хеш-таблиц** и **хеш-функции**): пусть имеется некоторое множество значений, которое необходимо каким-то образом отобразить на множество фиксированного размера. Для осуществления этого отображения используют функцию, которая по входному значению определяет номер позиции (номер кластера, куда должно попасть это значение). Но эта функция имеет свои особенности: при ее использовании возможны коллизии, связанные с тем, что различные значения могут оказаться в одном и том же кластере.
   Модель преобразования адресов, основанная на хешировании, достаточно проста. Из виртуального адреса аппаратно извлекается номер виртуальной страницы, который подается на вход некоторой хеш-функции, отображающей значение на аппаратную таблицу (т.н. хеш-таблицу) фиксированного размера. Каждая запись в данной таблице хранит начало списка коллизий, где каждый элемент списка является парой: номер виртуальной страницы — соответствующий ему номер физической страницы. Итак, перебирая соответствующий список коллизий, можно найти номер исходной виртуальной страницы и соответствующий номер физической страницы. Подобное решение имеет свои достоинства и недостатки: в частности, возникают проблемы с перемещением списков коллизий.

3. использования **инвертированных таблиц страниц**. Главной сложностью данного решения является требование к процессору на аппаратном уровне работать с идентификаторами процессов (их PID). Примерами таких процессоров могут служить процессоры серий SPARC и PowerPC. В этой модели виртуальный адрес трактуется как тройка значений: PID процесса, номер виртуальной страницы и смещение в этой странице. При таком подходе используется единственная таблица страниц для всей системы, и каждая строка данной таблицы соответствует физической странице (с номером, равным номеру этой строки). При этом каждая запись данной таблицы содержит информацию о том, какому процессу принадлежит данная физическая страница, а также какая виртуальная страница этого процесса размещена в данной физической странице. Итак, имея пару PID процесса и номер виртуальной страницы, производится поиск ее в таблице страниц, и по смещению найденного результата определяется номер физической страницы.
   + единственной таблицы страниц, обновление которой при смене контекстов сравнительно нетрудоемкое: операционная система производит обновление тех строк таблицы, для которых в соответствующие физические страницы происходит загрузка процесса. Отметим, что «тонким местом» данной модели является организация поиска в таблице. Если будет использоваться прямой поиск, то это приведет к существенным накладным расходам. Для оптимизации этого момента возможна надстройка над этим решением более интеллектуальных моделей — например, модели хеширования и/или использования TLB-таблиц.
   + процесс может использовать очень незначительную часть физического ресурса памяти, а все остальные его страницы могут размещаться во внешней памяти (быть откачанными)
   - фрагментации внутри страницы.
   - проблема выбора той страницы, которая должна быть откачана во внешнюю память при необходимости загрузить какую-то страницу из внешней памяти
   <u>Выбор откачиваемой страницы:</u>

1. алгоритм **NRU** (Not Recently Used — не использовавшийся в последнее время): с любой страницей ассоциируются два признака, один отвечает за обращение на чтение или запись к странице (R-признак), а второй — за модификацию страницы (M-признак), когда в страницу что-то записывается.
   0. Изначально для всех страниц процесса признаки R и M - 0 (происходит по таймеру или из-за событий)
   1. Категории страниц процесса:

   – **Класс 0**: R = 0, M = 0. В последнее время к ним не обращались и не меняли
   – **Класс 1**: R = 0, M = 1. В последнее время к ним не обращались, но изменили
   – **Класс 2**: R = 1, M = 0. В последнее время их читали (происходило обращение без изменений)
   – **Класс 3**: R = 1, M = 1. В последнее время в них записывали

   Откачивается случайная страница из минимального по номеру непустого класса

2. алгоритм **FIFO**: при загрузке очередной страницы в память ОС фиксирует время этой загрузки. Откачивается та страница, которая наиболее долго располагается в ОЗУ.
   - откачка интенсивно используемой страницы:
   Модификации:
   a. Проверяется R-признак самой «старой» страницы: если R = 0, то эта страница откачивается, если же R = 1, то признак обнуляется, а время загрузки данной страницы меняется на текущее (в конец очереди). Цикл.
      - рост накладных расходов при перемещении страниц по очереди.
      Модернизация: **«Часы»** - все страницы образуют циклический список. Имеется некоторый маркер, ссылающийся на некоторую страницу в списке, и этот маркер может перемещаться, например, только по часовой стрелке. Если значение R-признака в обозреваемой маркером странице равно нулю, то эта страница выгружается, а на ее место помещается новая страница, после чего маркер сдвигается. Если же R = 1, то этот признак обнуляется, а маркер сдвигается на следующую позицию.

b. **LRU** (Least Recently Used — «наименее недавно» – наиболее давно используемая страница). Пусть имеется N страниц и битовая изначально обнуленная матрица N × N. При обращении к i-ой странице, все биты i-ой строки устанавливаются в 1, а весь i-ый столбец обнуляется. Откачивается страница с минимальным двоичным числом по строке.
+ адекватно учитывает интенсивность использования страниц
- требует сложной аппаратной реализации.

c. **NFU** (Not Frequently Used — редко использовавшаяся страница) с каждой физической страницей с номером i ассоциирован программный счетчик Counti (изначально 0). А затем, по таймеру счетчик увеличивается на R-признак: Counti = Counti + Ri. Откачивается страница с минимальным значением счетчика.
- если какая-то страница в некоторый период времени интенсивно использовалась, то значение счетчика стало настолько большим, что при прекращении работы с данной страницей значение счетчика достаточно долго не даст откачать эту страницу
- при очень интенсивном обращении к странице возможно переполнение счетчика.
*модификация: по таймеру значение счетчика сдвигается на 1 разряд влево, после чего последний (правый) разряд устанавливается в значение R-признака.

- для страничного распределения: процессу выделяется единый диапазон виртуальных адресов: от нуля до некоторого предельного значения. Однако в процессе есть статические переменные, а есть команды, или есть стек, а есть куча (область динамической памяти) – объединения различных данных (с различными характеристиками использования). Но между всеми ими одинаково делится единого адресного пространства: выделяются область для команд, область для размещения данных, а также область для стека и кучи. При этом зачастую стек и куча размещаются в единой области, причем стек прижат к одной границе области, куча — к другой, и «растут» они навстречу друг другу. Соответственно, возможны ситуации, когда они начинают пересекаться (ситуация переполнения стека). Или даже если стек будет располагаться в отдельной области памяти, он может переполнить выделенное ему пространство.

4. Сегментное распределение: каждый процесс - совокупность сегментов своего размера и функциональности: сегменты кода, сегменты статических данных, сегмент стека и т.д. Для организации работы с сегментами используется некоторая таблица с информацией о каждом сегменте (его размер, адрес начала – адрес базы, дополнительные атрибуты (права и режимы доступа к содержимому). Тогда виртуальный адрес = номер сегмента и смещение в нем. Т.е. система аппаратно извлекает из в.а. номер сегмента i, обращается к i-ой строке таблицы, из которой извлекается информация о сегменте. После этого происходит проверка, не превосходит ли величина сегмента размера самого сегмента (прерывание), сложение базы со смещением, вычисление физического адреса.
+ простота организации (развитие модели распределения разделов: там каждому процессу выделяется только один сегмент (раздел), а тут совокупность сегментов, каждый из которых будет иметь свои функциональные обязанности)
- каждый сегмент должен целиком размещаться в памяти (неявная неэффективности из-за принципа локальности)
- откачка/подкачка: подкачка осуществляется всем процессом или, по крайней мере, целым сегментом – неэффективно
- поскольку каждый сегмент так или иначе должен быть размещен в памяти, то возникает ограничение на предельный размер сегмента.

5. Сегментно-страничное распределение
виртуальный адрес = номер сегмента и смещение в нем. Имеется также аппаратная таблица сегментов, посредством которой из виртуального адреса получается т.н. **линейный адрес**, который, в свою очередь, представляется в виде номера страницы и величины смещения в ней. А затем, используя таблицу страниц, получается непосредственно физический адрес. Модель сочетает логическое сегментирование (в процессе имеется ряд виртуальных сегментов) со страничной организацией (сегменты дробятся на страницы)
+ можно работать с отдельными страницами памяти, не требуя при этом полного размещения сегмента в ОЗУ

Пример от Intel: Виртуальный адрес - **селектор** (информация о локализации сегмента) + смещение в сегменте. В модели Intel сегмент может быть:

- **Локальным**: описывается в таблице локальных дескрипторов **LDT** (Local Descriptor Table) и доступен лишь данному процессу
- **Глобальным**: описывается в таблице глобальных дескрипторов **GDT** (Global Descriptor Table) и может разделяться между процессами.

Каждая запись таблиц LDT и GDT хранит полную информацию о сегменте (адрес базы, размер и пр.). Итак, в селекторе хранится тип сегмента, после которого следует номер сегмента (номер записи в соответствующей таблице). Помимо перечисленного, селектор хранит различные атрибуты, касающиеся режимов доступа к сегменту.

На основе виртуального адреса, посредством использования информации из таблиц LDT и GDT, получается 32-разрядный линейный адрес, который интерпретируется в терминах двухуровневой иерархической страничной организации. Последние 12 разрядов отводятся под смещение в физической странице, а первые 20 разрядов интерпретируют как 10-разрядный индекс во «внешней» таблице групп страниц и 10-разрядное смещение в соответствующей таблице второго уровня иерархии.

\+ можно «выключать» страничную функцию, и тогда модель Intel начинает работать по сегментному распределению
\+ можно не использовать сегментную организацию процесса, и тогда данная реализация будет работать по страничному распределению памяти

Управление внешними устройствами (ВУ)

Архитектура организации управления внешними устройствами

При взаимодействии работы процессора и ВУ есть два потока информации: поток управляющей информации (поток команд какому-либо устройству управления) и поток данных (поток информации, участвующей в обмене обычно между ОЗУ и ВУ).
Модели управления (исторически):

1. **ЦП непосредственно управляет** ВУ (на уровне микрокоманд) - поток управления и поток данных полностью шли через ЦПУ. **Синхронное** управление: если начался обмен, то, пока он не закончится, процессор не продолжает вычисления (поскольку занят обменом).
2. **ЦП управляет ВУ с помощью специализированных контроллеров** устройств, которые концептуально располагались между ЦП и соответствующими ВУ и позволяли ЦП работать с более высокоуровневыми операциями при управлении ВУ. То есть процессор частично освобождался от потока управления - вместо большого числа микрокоманд конкретного устройства он оперировал небольшим числом более высокоуровневых операций. Так же **синхронная** модель.
3. **Модернизация 2 с помощью аппарата прерываний** позволяла параллельно обмену для одного процесса поставить на выполнение другую задачу (или же текущий процесс может продолжить выполнять какие-то свои вычисления), а по окончании обмена (успешного или неуспешного) в системе возникнет прерывание, сигнализирующее возможность дальнейшего выполнения первого процесса. Но и эти две модели предполагали, что поток данных идет через процессор. Асинхронное управление.
4. ЦП управляет ВУ с помощью **контроллеров прямого доступа** к памяти (или **DMA-контроллеров** - Direct Memory Access). Процессор 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. Таким образом, поток данных шел в обход процессора.
5. Управление ЦП с помощью **специализированного процессора** (или даже **специализированных компьютеров**) или **каналов ввода-вывода**. снижение нагрузки на центральный процессор с точки зрения обработки потока управления: ЦПУ теперь оперирует функционально-емкими макрокомандами. Решение задачи осуществления непосредственного обмена, а также решение всех связанных с обменом вопросов (в т.ч. оптимизация операций обмена, например, за счет использования аппаратной буферизации в процессоре ввода-вывода) ложится «на плечи» специализированного процессора.

Программное управление ВУ

Архитектура программного управления ВУ:
аппаратура, а далее следуют программные уровни: **программы обработки прерываний**, **драйверы физических устройств**, а на вершине иерархии лежит уровень **драйверов логических устройств**, причем каждый уровень строится на основании нижележащего уровня.

Цели программного управления устройствами:

1. унификация программных интерфейсов доступа к внешним устройствам - стандартизация правил использования различных устройств (если данная цель достигнута, то, например, пользователю, пожелавшему распечатать текстовый файл, не надо будет заботиться об организации управления конкретным печатающим устройством - ему достаточно воспользоваться некоторым общим программным интерфейсом)	**2. буферизация обмена** — связана с различной производительностью основных компонентов системы: ВУ медленнее центральной части компьютера, т.е. нужно сгладить разброс производительностей различных компонентов. Если речь идет об устройствах не оперативного доступа (к ним не идет массовое обращение на обмен от процессов), то для них это второстепенная проблема (принтеру не особо требуется реализация буфера, а вот магнитному диску да, ибо обмен должен быть быстрым – массовое использование). Решение – кэш.
3. выявление и локализация ошибок, а также **устранение их последствий**. Чем сложнее система, тем больше статистически в ней возникает сбоев. То есть система должна уметь выявить момент сбоя, и либо самостоятельно обойти эту ситуацию, либо известить пользователя.	**4. обеспечение конкретной модели синхронизации при выполнении обмена** (синхронный или асинхронный обмен) - цель поддерживать оба вида обмена, а выбор конкретного типа зависит от пользователя.
5. обеспечение стратегии доступа к устройству (распределенный или монопольный доступ): один и тот же файл может быть доступен через множество файловых дескрипторов, которые могут быть распределены между различными процессами, т.е. файл может быть многократно открыт в системе, и система позволяет организовывать распределенный доступ к его информации. Система позволяет организовать управление этим доступом и синхронизацию, а так же монопольный доступ к устройству.	**6. планирование выполнения операций обмена**. Это важная проблема, поскольку от качества планирования может во многом зависеть эффективность функционирования вычислительной системы. Неправильно организованное планирование очереди заказов на обмен может привести к деградации системы, связанной, к примеру, с началом голодания каких-то процессов и, соответственно, зависания их функциональности.

Планирование дисковых обменов

Пусть обмен с некоторым диском осуществляется дорожками и имеется очередь запросов к следующим дорожкам: 4, 40, 11, 35, 7, 14 и пусть изначально головка дискового устройства позиционирована на 15-ой дорожке. Время на обмен = выход головки на позицию + вращение диска + непосредственно обмен.

Стратегии обмена:

1. **FIFO** (First In First Out): считывание начнется с певрой в списке дорожки (4), суммарная длина пути составляет 135 дорожек, что в среднем 22,5 дорожки на один обмен.
2. **LIFO** (Last In First Out): аналогично предыдущей. Полезная, когда поступают цепочки связанных обменов: процесс считывает информацию, изменяет ее и обратно записывает (если фифо, то после считывания он не сможет продолжаться, т.к. будет ожидать записи).
3. **SSTF** (Shortest Service Time First) — основана на «жадном» алгоритме (итерационный алгоритм, который на каждой итерации ищут наилучшее решение, т.е. на каждом шаге осуществляет поиск в очереди запросов номера дорожки, требующей минимального перемещения головки диска)
   + быстро
   - крайние дорожки голодают
4. **PRI** - каждому процессу присваивается некоторый приоритет и очередь запросов обрабатывается согласно приоритетам.
   - необходим контроль корректной выдачи приоритета, иначе голодание низкоприоритетных процессов

5. **SCAN** или лифтовый - головка диска перемещается сначала в одну сторону до границы диска, выбирая каждый раз из очереди запрос с номером обозреваемой головкой дорожки, а затем — в другую.
   + для любого набора запросов потребуется перемещений не больше удвоенного числа дорожек на диске
   - возможна деградация системы вследствие голодания некоторых процессов в случае, когда идет интенсивный обмен с некоторой локальной областью диска
6. **C-SCAN** или циклического сканирования - сканирование всегда происходит в одном направлении. В очереди запросов ищется запрос с минимальным (или максимальным) номером, головка передвигается к дорожке с этим номером, а затем за один проход по диску обрабатывается вся очередь запросов. Но проблемы остаются те же самые, что и для алгоритма сканирования.
7. **N-step-SCAN** или многошаговый - очередь запросов каким-либо образом (по локализации запросов, по времени поступления) делится на N подочередей, затем по какой-либо стратегии (например, по порядку формирования) выбирается очередь, которая будет обрабатываться и начинается ее обработка. Во время обработки очереди блокируется попадание новых запросов в эту очередь (тогда эти запросы могут сформировать новую очередь), другие очереди могут получать заявки. Сама обработка может осуществляться, например, по алгоритму сканирования.

+ нет голодания

Все это упрощение - если идет заказ на считывание данных, потом процесс их изменяет, а затем обратно записывает, то эти считывание и запись могут следовать лишь в одном порядке.

RAID-системы. Уровни RAID

RAID — Redundant Array of Independent (Inexpensive) Disks - избыточный массив независимых (недорогих) дисков. На сегодняшний день обе расшифровки не совсем корректны. Понятие недорогих дисков родилось в те времена, когда большие быстрые диски стоили достаточно дорого, и перед многими организациями, желающими сэкономить, стояла задача построения такой организации набора более дешевых и менее быстродействующих и емких дисков, чтобы их суммарная эффективность не уступала одному «дорогому» диску. На сегодняшний день цены между различными по характеристикам дисками более сглажены, но бывают и исключения, когда новейший диск чуть ли не на порядок опережает по цене своих предыдущих собратьев. Что касается независимости дисков, то она соблюдается не всегда.

RAID-система —совокупность физических дисковых устройств, которая представляется в ОС как одно устройство, имеющее возможность организации параллельных обменов.

- присутствует избыточная информация для контроля и восстановления информации, хранимой на этих дисках.
- на разных устройствах, составляющих RAID-массив, размещаются порции данных фиксированного размера – **полосы**. Ими осуществляется обмен в таких системах. Размер полосы зависит от конкретного устройства

(при обсуждении файловых систем была упомянута иерархия различных блоков; RAID добавляет в эту иерархию дополнительный уровень — уровень полос RAID).

Модели RAID-систем: 0 и 1 - могут реализовываться программным способом.
RAID 0 - полоса соизмерима с дисковыми блоками, соседние полосы находятся на разных устройствах. ~ расслоение ОП. С каждым диском обмен может происходить параллельно. Каждое устройство независимое (движение головок в каждом из устройств не синхронизировано).
+ не хранит избыточную информацию
+ для данных доступен объем, равный суммарной емкости дисков
+ за счет параллельного выполнения обменов доступ к информации становится более быстрым

RAID 1 – два комплектов дисков. При записи информации она сохраняется на соответствующем диске и на диске-дублере. При чтении информации запрос направляется лишь одному из дисков. К диску-дублеру происходит обращение при утере информации на первом экземпляре диска.
- дорогая (не стоимость, а цена обслуживания - диски занимают много места, потребляют довольно приличную мощность и выделяют много тепла и требуют расходов на обеспечение резервного питания, чтобы эти диски не отключались при перебоях с электроэнергией).

RAID 2 и **RAID 3** - синхронизированные головки, т.е. в массиве используются не независимые устройства, а специальным образом синхронизированные. Эти модели обычно имеют полосы незначительного размера (например, байт или слово).
- содержат избыточную информацию, позволяющую восстановить данные в случае выхода из строя одного из устройств: RAID 2 использует коды Хэмминга (коды, исправляющие одну ошибку и выявляющие двойные ошибки), RAID 3 - четность с чередующимися битами (один из дисков назначается для хранения избыточной информации — полос, дополняющих до четности соответствующие полосы на других дисках (т.е., по сути, в каждой позиции суммарное число единиц на всех дисках должно быть четным).

RAID 4 является упрощением RAID 3 - массив несинхронизированных устройств. Однако необходимо поддерживать в корректном состоянии диска четности - каждый раз происходит пересчет по соответствующей формуле.

RAID 5 и **RAID 6** более надежные RAID 3 и 4: опасно хранить важную информацию (в данном случае полосы четности) на одном носителе, т.к. при каждой записи происходит обращение всегда к одному и тому же устройству, что может спровоцировать его скорейший выход из строя. Надежнее разнести служебную информацию по разным дискам. Соответственно, RAID 5 распределяет избыточную информацию по дискам циклическим образом, а RAID 6 использует двухуровневую избыточную информацию (которая также разнесена по дискам).

Работа с ВУ в ОС Unix

Файлы устройств, драйверы

*практически все, с чем работает система - файлы. ВУ не исключение и также представлены в системе в виде специальных файлов устройств, хранимых обычно в каталоге /dev.

1. **байт-ориентированные** - обмен осуществляется порциями данных фиксированной длины, называемыми блоками. Обычно размер блока кратен степени двойки, а зачастую кратен 512 байтам. Помимо физических устройств (с которыми можно осуществлять обмен) к ним могут относиться и устройства, с которыми обмен не осуществим (таймер)
2. **блок-ориентированные** - обмен порциями данных произвольного размера (от 1 байта до некоторого k). Априори считается, что устройства, на которых может располагаться ФС блок-ориентированные.

за регистрацию устройств в системе в конечном счете отвечает **драйвер** устройства: именно он определяет тип интерфейса устройства. Бывают ситуации, когда одно и то же устройство рассматривается системой и как байт-ориентированное, и как блок-ориентированное: например ОП Заведомо ОЗУ является байт-ориентированным устройством, но при организации обменов или при развертывании в ОП виртуальной ФС ОЗУ может рассматриваться уже как блок-ориентированное устройство.

Специальные файлы устройств:

- задачи:
  – именование устройств (точнее именование драйверов устройств)
  – связывание имени устройства, с конкретным драйвером
- не имеют блоков файла, хранимых в рабочем пространстве ФС
- вся содержательная информация файлов данного типа размещается исключительно в соответствующем индексном дескрипторе, который состоит из перечня стандартных атрибутов файла:
  - тип этого файла
  - специальные атрибуты:
    - тип файла устройства (блок- или байт-ориентированное)
    - **старший номер** - номер драйвера в таблице драйверов, соответствующей типу файла устройства
    - **младший номер** - некоторая дополнительная информация, передаваемая драйверу при обращении.

За счет этого реализуется механизм, когда один драйвер может управлять несколькими схожими устройствами.

Системные таблицы драйверов устройств

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

Каждая запись этих таблиц содержит либо коммутатор устройства либо специальная ссылка-заглушка на точку ядра.
Коммутатор – специальная структура, которая хранит указатели на всевозможные точки входа (т.е. реализуемые функции) в соответствующий драйвер.
Типовые имена точек входа в драйвер:

– **βopen()**, **βclose()**;
– **βread()**, **βwrite()**;
– **βioctl()** - разного рода настройки и управление драйвером
– **βintr()** - вызывается при поступлении прерывания, ассоциированного с данным устройством

Символ β является аббревиатурой имени устройства: обычно в Unix-системах для именования устройства используют двухсимвольные имена. Например, lp — принтер, mt — магнитная лента и т.п.

В общем случае система специфицирует почти все функции драйвера, которые доступны пользователю. Если какая-либо функция отсутствует, то на ее месте в коммутаторе может стоять заглушка. Заглушки могут быть двух типов:
nulldev() - при обращении сразу возвращает управление
nodev() - приобращении возвращает управление с кодом ошибки.
*Например, для таймера скорее всего будут отсутствовать функции чтения и записи, а при попытке чтения или записи система должна «ругнуться» (т.е. заглушка типа nodev()).
Традиционно часть функций драйверов может быть реализована синхронным способом, а другая часть — асинхронным способом. Соответственно, синхронная часть драйвера называется **top half, а асинхронная — **bottom half**.

Ситуации, вызывающие обращение к функциям драйвера

1. старт системы и инициализация устройств и драйверов: система просматривает перечень (содержится в каталоге /dev) устройств, которые могут быть к ней подключены, находит те, что есть в наличии и подключает их вызовом функции коммутатора (функции βioctl()).
2. обработка запросов на обмен: если процессу необходимо произвести считывание или запись данных, то происходит обращение к соответствующей точке входа в драйвер.
3. обработка прерывания, связанного с данным устройством (например, закончился обмен или же по линии связи пришел какой-то сигнал, который необходимо обработать). В этом случае возникает прерывание, обработка которого происходит в соответствующем драйвере.
4. выполнение специальных команд управления устройством

Включение, удаление драйверов из системы

Изначально Unix-системы (как и большинство систем) предполагали «жесткое» статические встраивание драйверов в код ядра. Это означало, что при добавлении нового драйвера или удалении существующего необходимо было выполнить достаточно трудоемкую операцию перетрансляции (когда ядро создается «с нуля») или, как минимум, перекомпоновку ядра (когда есть готовые объектные модули). Соответственно, эти операции требовали серьезных навыков от системного администратора. Чтобы минимизировать число перекомпоновок ядра, надо было максимизировать число драйверов, встроенных в систему. Но такая модель была неэффективной, поскольку в системе присутствовали драйверы, которые никак не используются.
Решение: динамическое связывание драйверов - в системе присутствуют программные средства, позволяющие динамически, «на лету» подключить к ОС тот или иной драйвер.
Предполагается:

- решение задачи именования устройства
- инициализация драйвера (формирование системных областей данных и т.п.) и инициализация устройства (приведение устройства в начальное состояние)
- добавление данного драйвера в соответствующую таблицу драйверов устройств (либо блок-, либо байт-ориентированных)
- «установка» обработчика прерывания, т.е. предоставление ядру информации, что при возникновении определенного прерывания управление необходимо передать в соответствующую точку входа в данный драйвер.

Организация обмена данными с файлами

Для организации операций обмена в ОС Unix используются системные таблицы и структуры, часть которых ассоциирована с каждым процессом (т.е. они располагаются в адресном пространстве процесса), а часть — с самой ОС.

Таблица открытых файлов (**ТОФ**) создается в адресном пространстве процесса. Каждая запись этой таблицы соответствует открытому в процессе файлу. Номер **файлового дескриптора** - номер записи в таблице открытых файлов процесса. Размер таблицы определяется при настройке ОП: этот параметр определяет максимум открытых в одном процессе файлов.
Каждая запись ТОФ содержит целый набор атрибутов, в том числе ссылку на номер записи в **таблице файлов** операционной системы (**ТФ**). Таблица файлов ОС является системной таблицей в единственном экземпляре. В ней происходит регистрация всех открытых в системе файлов.

В ТФ ОС помимо прочего содержатся:

- **указатель чтения/записи** (ссылающийся на позицию в файле, начиная с которой будет происходить, соответственно, чтение или запись)
- **счетчик кратности** (речь о нем пойдет ниже)
- **ссылка** на таблицу индексных дескрипторов открытых файлов

Таблица индексных дескрипторов открытых файлов (**ТИДОФ**) также является системной структурой данных, содержащей перечень индексных дескрипторов всех файлов, открытых в данный момент в системе. Каждая запись этой таблицы содержит актуальную копию открытого в системе индексного дескриптора. Здесь также хранится целый набор параметров, среди которых имеется и счетчик кратности.

Пример: пусть запущен Процесс1, для которого система при его создании сформировала ТОФ1. Затем этот процесс, посредством обращения к системному вызову open(), открывает файл с именем name. Это означает, что в свободном месте этой таблицы заводится файловый дескриптор для работы с данным файлом. В этой записи ТОФ хранится ссылка на соответствующую запись в ТФ. Если файл открывается впервые в системе, то в ТФ заводится новая запись для работы с этим файлом. В данной записи хранится указатель чтения/записи, а также коэффициент кратности, который в начале устанавливается в значение 1 — это означает, что с данной записью ТФ ассоциирована единственная запись из какой-либо ТОФ. И, конечно, в данной записи ТФ хранится ссылка на запись в ТИДОФ, содержащую актуальную копию индексного дескриптора обрабатываемого файла. Таблицы ТФ и ТИДОФ хранят оперативную информацию, поэтому они располагаются в ОЗУ. Соответственно, ФС, работая с блоками открытого файла, оперирует данными, хранимыми именно в ТИДОФ. Пусть в системе позже был запущен Процесс2, который также открыл файл name. В этом случае в ТФ заводится новая запись, в которой устанавливается свой указатель чтения/записи, но эта запись ТФ будет ссылаться на тот же номер записи в ТИДОФ. Такой механизм позволяет корректно (с системной точки зрения) обрабатывать ситуации одновременной работы с одним и тем же файлом: поскольку в итоге все сводится к единственной актуальной копии индексного дескриптора в ТИДОФ, то работа ведется с соответствующими блоками файла. При этом данные процессы работают с файлом каждый «по-своему», т.к. каждый из них оперирует независимыми указателями чтения/записи, хранимыми в различных записях ТФ. Теперь предположим, что после открытия файла name, Процесс1 обращается к системному вызову fork() и порождает своего потомка — Процесс3. При обращении к системному вызову fork() ТОФ родительского процесса копируется в ТОФ сыновнего процесса. Соответственно, все записи ТОФ3 будут ссылаться на те же записи ТФ, что и записи ТОФ1. Это означает, что при порождении сыновнего процесса в соответствующих записях ТФ происходит увеличение на 1 счетчика кратности. Заметим, что подобный механизм наследования подразумевает, что сыновний процесс будет работать с теми же указателями чтения/записи, что и родительский процесс.

+ТИДОФ располагается в ОП
+ становится эффективнее работа с ФС (уменьшается число обращений к пространству индексных дескрипторов ФС, т.е. этот механизм можно считать кэшированием системных обменов)
- некорректное завершение работы ОС: если в системе происходит сбой, то содержимое ТИДОФ будет потеряно, а это означает, что будут потери и в ФС.

Буферизация при блок-ориентированном обмене

В ОС Unix возможна организация многоуровневой буферизации при выполнении неэффективных действий: для организации блок-ориентированных обменов система использует стандартную стратегию кэширования. Все действия тут те же самые (вплоть до отдельных нюансов). Цель кэширования — минимизация обменов с ВУ.

Для буферизации используют пул буферов, размером в один блок каждый.

1. Поиск заданного блока в буферном пуле. Если удачно, то переход на п.4
2. Поиск буфера в буферном пуле для чтения и размещения заданного блока.
3. Чтение блока в найденный буфер.
4. Изменение счетчика времени во всех буферах.
5. Содержимое данного буфера передается в качестве результата.

+за счет минимизации реальных обращений к физическим устройствам работа системы более эффективная
- кэширование дисковых обменов приводит к тому, что имеется несоответствие реального содержимого диска и того

содержимого, которое должно быть на нем
- при сбое системы возможна потеря информации в КЭШах, расположенных в ОП (потеря индексного дескриптора. Конечно, во время работы система сбрасывает актуальную информацию по местам дислокации, но этого недостаточно. Если теряется индексный дескриптор, то теряется список блоков файла. За счет использования избыточной информации можно организовать и восстановление. Но заметим, что при сбое теряется лишь файл, - работоспособность системы остается)

Альтернативными являются системы, работающие без буферизации, когда при каждом обмене происходит реальное обращение к физическому устройству. Эти системы более устойчивы к сбоям в аппаратуре. Примером такой системы может служить Microsoft DOS. Соответственно, при развертывании на ненадежной аппаратуре ОС Unix многие ее положительные качества могли теряться.

Борьба со сбоями

1. в системе может быть задан параметр, определяющий промежутки времени, через которые осуществляется сброс системных данных по местам дислокации
2. в системе доступна команда sync, позволяющая осуществлять в любой момент этот сброс информации по желанию пользователя
3. система использует избыточную информацию, позволяющую восстанавливать данные. Поскольку практически весь ввод-вывод сводится к обменам с ФС (т.е., по сути, идет борьба за сохранность файлов и файловой системы), то использование избыточных данных позволяет восстанавливать системную информацию. Обычно безвозвратные потери происходят с частью пользовательской информации; системная информация почти всегда восстанавливаема.