**Вопросы к экзамену**

*Возможны незначительные изменения в течение сентября 2023 года.*

1. Операционные системы: определение, назначение, состав, функции.

**Операционная система** — это комплект программ, которые служат интерфейсом между модулями вычислительных систем и прикладными программными приложениями, а также управляют компьютерным оборудованием и процессами вычислений, эффективным распределением вычислительных мощностей среди процессов вычислений.

**Вычислительная система** — взаимосвязанная совокупность аппаратных средств вычислительной техники и программного обеспечения, предназначенная для обработки информации и удобства работы с ней.

**Две основные функции (назначение) ОС**:

1) предоставлять пользователю некую расширенную виртуальную машину, с которой легче работать (легче программировать), чем непосредственно с аппаратурой реального компьютера или реальной сети;

2) управлять ресурсами вычислительной системы. Поэтому в специальной литературе ОС представляется всегда двояко: как расширенная виртуальная машина и как система управления ресурсами.

Функции ОС автономного компьютера обычно группируются в соответствии с типами локальных ресурсов, которыми управляет ОС. Такие группы называют подсистемами. **Наиболее важные из них**:

* подсистема управления процессами,
* подсистема управления памятью,
* подсистема управления файлами,
* подсистема управления внешними устройствами,
* подсистема пользовательского интерфейса,
* подсистема защиты данных и администрирования

Функции ОС:

* прием от пользователя (оператора) заданий или команд, сформулированных на соответствующих языках, и их обработка;
* загрузка в ОП программ и их исполнение;
* инициация программы (передача ей управления);
* прием и исполнение программных запросов на запуск, приостановку, остановку других программ;
* организация взаимодействия между задачами;
* идентификация всех программ и данных;
* обеспечение работы системы управления файлами и/или систем управления БД;
* обеспечение режима мультипрограммирования (многозадачности);
* планирование и диспетчеризация задач;
* обеспечение функций по организации и управлению операциями ввода/вывода;
* удовлетворение жестким ограничениям на время ответа в режиме реального времени (для соответствующих ОС);
* управление памятью, организация виртуальной памяти;
* организация механизмов обмена сообщениями и данными между выполняющимися программами;
* защита одной программы от влияния другой;
* обеспечение сохранности данных;
* аутентификация, авторизация и другие средства обеспечения безопасности;
* предоставление услуг на случай частичного сбоя системы;
* обеспечение работы систем программирования;
* параллельное исполнение нескольких задач.

2. Операционные системы: классификация, основные этапы развития, особенности современного этапа развития

Существует несколько классификаций ОС.

**В зависимости от способа организации вычислений**:

* Системы пакетной обработки – основной задачей является организация наибольшего количества вычислительных процессов за единицу времени. Определенные процессы объединяются в пакет, который затем обрабатывает ОС.
* Системы разделения времени – создание возможности единовременного взаимодействия с устройством сразу несколькими людьми. В порядке очереди каждый пользователь получает определенный промежуток процессорного времени.
* Системы реального времени – организация работы каждой задачи за определенный промежуток времени, присущий каждой конкретной задаче.

**В зависимости от типа ядра:**

* OС с монолитным ядром (ядро ОС выполняет все функции, включая управление памятью, устройствами ввода-вывода и файловой системой, в одном монолитном модуле);
* OС с микроядром (ядро ОС содержит только основные функции, такие как планирование задач, управление памятью и межпроцессное взаимодействие, а остальные функции реализуются в виде отдельных модулей, работающих в привилегированном или пользовательском режиме);
* OС с гибридным ядром (комбинация монолитного и микроядерного подходов, где некоторые компоненты, такие как драйверы устройств, могут работать в пространстве ядра, в то время как другие функции, такие как файловая система и сетевые протоколы, могут работать в виде отдельных модулей, загружаемых по требованию).

**В зависимости от количества единовременно решаемых задач:**

* однозадачные;
* многозадачные;

**В зависимости от количества пользователей:**

* однопользовательские;
* многопользовательские.

**В зависимости от количества поддерживаемых процессоров:**

* однопроцессорные
* многопроцессорные

**В зависимости от возможности работы в компьютерной сети:**

* локальные – автономные ОС, которые не позволяют работать с компьютерными сетями;
* сетевые – ОС с поддержкой компьютерных сетей.

**В зависимости от роли в сетевом взаимодействии:**

* серверные – ОС, открывающие доступ к ресурсам сети и осуществляющие управление сетевой инфраструктурой;
* клиентские – ОС, которые имеют возможность получения доступа к ресурсам сети.

**В зависимости от типа лицензии:**

* открытые – ОС с открытым исходным кодом, который можно изучать и редактировать;
* проприетарные – ОС, связанные с определенным правообладателем и, как правило, имеющие закрытый исходный код.

**В зависимости от сферы использования:**

* ОС мэйнфреймов – больших компьютеров;
* ОС серверов;
* ОС персональных компьютеров;
* OC мобильных устройств;
* встроенные OC;
* OC маршрутизаторов.

Основные этапы развития:

**Первое поколение (1945-1955 гг.)**

Первое поколение компьютеров строилось преимущественно на электронных лампах. Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно (такой режим работы называется последовательной обработкой данных).

В сущности, все решаемые задачи сводились к простым математическим и числовым расчетам, таким как уточнение таблиц синусов, косинусов и логарифмов или вычисление траекторий полета артиллерийских снарядов.

Когда в начале 1950-х годов появились перфокарты, положение несколько улучшилось. Появилась возможность вместо использования коммутационных панелей записывать программы на картах и считывать с них, но в остальном процедура работы не претерпела изменений

**Второе поколение (1955-1965 гг.)**

Второе поколение компьютеров характеризуется использованием транзисторов, что повысило их надёжность и продлило время непрерывной работы.

Машины, называемые теперь мэйнфреймами, располагались в специальных большихизалах с кондиционированием воздуха, где ими управлял целый штат профессиональных операторов.

Чтобы выполнить задание (то есть программу или комплект программ), программист сначала должен был записать его на бумаге (на Фортране или ассемблере), а затем перенести на перфокарты. После этого он должен был принести колоду перфокарт в комнату ввода данных, передать одному из операторов и идти пить кофе в ожидании, когда будет готов результат.

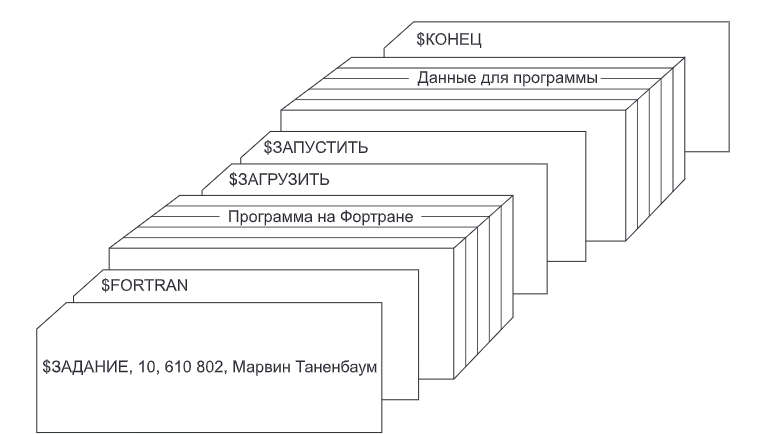
Когда компьютер заканчивал выполнение какого-либо из текущих заданий, оператор подходил к принтеру, отрывал лист с полученными данными и относил его в комнату для распечаток, где программист позже мог его забрать. Затем оператор брал одну из колод перфокарт, принесенных из комнаты ввода данных, и запускал ее на считывание.

Если в процессе расчетов был необходим компилятор языка Фортран, то оператору приходилось брать его из картотечного шкафа и загружать в машину отдельно. На одно только хождение операторов по машинному залу терялась впустую масса драгоценного компьютерного времени.

Если учесть высокую стоимость оборудования, неудивительно, что люди довольно скоро занялись поиском способа повышения эффективности использования машинного времени. Общепринятым решением стала **система пакетной обработки**. Первоначально замысел состоял в том, чтобы собрать полный поднос заданий (колод перфокарт) в комнате входных данных и затем переписать их на магнитную ленту



Структура типичного входного задания показана на рис. 1.4. Оно начиналось с карты $JOB, на которой указывались максимальное время выполнения задания в минутах, загружаемый учетный номер и имя программиста. Затем поступала карта $FORTRAN, дающая операционной системе указание загрузить

компилятор языка Фортран с системной магнитной ленты. За этой картой следовала программа, которую нужно было компилировать, а после нее — карта $LOAD, указывающая операционной системе загрузить только что скомпилированную объектную программу. (Скомпилированные программы часто записывались на рабочих лентах, данные с которых могли стираться сразу после использования, и их загрузка должна была выполняться в явном виде.) Следом шла карта $RUN, дающая операционной системе команду на выполнение программы с использованием данных, следующих за ней. Наконец, карта завершения $END отмечала конец задания. Эти примитивные управляющие перфокарты были предшественниками современных оболочек и интерпретаторов командной строки.

**Третье поколение (1965-1980 гг.)**

Компьютеры третьего поколения использовали малые **интегральные схемы**, что дало им преимущество в цене и качестве по сравнению с машинами второго поколения. Самым важным достижением **явилась многозадачность**. В компьютерах предыдущего поколения вычисления останавливались на время ввода-вывода. Проблема была решена разбиением памяти на несколько частей, называемых разделами, в каждом из которых выполнялось отдельное задание. При окончании выполнения каждого текущего задания операционная система могла загружать новое задание с диска в освободившийся раздел памяти и запускать это задание. Этот технический прием называется **подкачкой данных,** или спулингом (spooling — английское слово, которое произошло от Simultaneous Peripheral Operation On Line, то есть совместная периферийная операция в интерактивном режиме), и его также используют для выдачи полученных данных.

В системах третьего поколения промежуток времени между передачей задания и возвращением результатов часто составлял несколько часов, так что единственная поставленная не в том месте запятая могла стать причиной сбоя при компиляции, и получалось, что программист полдня тратил впустую. Программистам это очень не нравилось. Желание сократить время ожидания ответа привело к разработке **режима разделения времени** — варианту многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Если двадцать пользователей зарегистрированы в системе, работающей в режиме разделения времени, и семнадцать из них думают, беседуют или пьют кофе, то центральный процессор по очереди предоставляется трем пользователям, желающим работать на машине.

**POSIX (Portable Operating System Interface)** - набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем.

**Четвертое поколение (1980-2005 гг.)**

Разработка БИС (большие интегральные схемы, LSI, Large Scale Integration — кремниевые микросхемы, содержащие тысячи транзисторов на одном квадратном сантиметре) привела к появлению микрокомпьютеров. Операционные системы для микрокомпьютеров принято относить к четвертому поколению.

**Пятое поколение (2005 г. - по н.в.)**

Широкую популярность приобрела технология виртуализации — представление вычислительных ресурсов, абстрагированное от аппаратной реализации.

**Эксплуатационные требования к ОС**

К этим требованиям относятся: расширяемость (код операционной системы должен быть написан таким образом, чтобы можно было легко внести дополнения и изменения); переносимость (например, на другой процессор), надежность и отказоустойчивость; совместимость (соответствие стандартам, запуск программ для других ОС); безопасность (защита ресурсов одних пользователей от других); производительность.

**Особенности современного этапа развития операционных систем**

На современном этапе развития операционных систем на передний план вышли средства обеспечения безопасности. Это связано с возросшей ценностью информации, обрабатываемой компьютерами, а также с повышенным уровнем угроз, существующих при передаче данных по сетям, особенно по публичным, таким как Интернет. Многие операционные системы обладают сегодня развитыми средствами защиты информации, основанными на шифрации данных, аутентификации и авторизации. Современным операционным системам присуща многоплатформенность, т.е. способность работать на совершенно различных типах компьютеров.

**Перспективы развития операционных систем**

Эффективность работы человека становится основным фактором, определяющим эффективность вычислительной системы в целом. Пользовательский интерфейс операционной системы становится все более интеллектуальным, направляя действия человека в типовых ситуациях и принимая за него рутинные решения. Операционные системы будущего должны обеспечить высокий уровень прозрачности сетевых ресурсов, взяв на себя задачу организации распределенных вычислений, превратив сеть в виртуальный компьютер.

3. Компоненты архитектуры вычислительных систем, их назначение и взаимодействие.

**Шина** - это несколько проводников, соединяющих несколько устройств. Шины можно разделить на категории в соответствии с выполняемыми функциями. Они могут быть внутренними по отношению к процессору и служить для передачи данных в АЛУ и из АЛУ, а могут быть внешними по отношению к процессору и связывать процессор с памятью или устройствами ввода-вывода. Любая шина состоит из трех частей: **шина данных**, по которой передается информация; **шина адреса**, определяющая, кому передаются данные; **шина управления**, регулирующая процесс обмена информацией. Также может присутствовать шина питания.

**Контроллер** - специализированный процессор, управляющий работой внешнего устройства по специальным встроенным программам обмена.

Cхему легко пополнять новыми устройствами - это свойство называют **открытостью архитектуры**. Для пользователя открытая архитектура означает возможность свободно выбирать состав внешних устройств для своего компьютера, т.е. конфигурировать его в зависимости от круга решаемых задач

Классическая архитектура ЭВМ, построенная по принципу фон Неймана (фон-неймановская архитектура) и реализованная в вычислительных машинах двух (трех) поколений, представлена и содержит следующие основные блоки:

• арифметическо-логическое устройство (АЛУ), выполняющее арифметические и логические операции;

• управляющее устройство (УУ), организующее процесс выполнения программ;

• внешнее запоминающее устройство (ВЗУ), или память, для хранения программ и данных;

• оперативное запоминающее устройство (ОЗУ);

• устройства ввода и вывода информации (УВВ).



Современный компьютер в типичной конфигурации включает в себя:

• Системный блок

* блок питания
* корпус
* материнская плата
* процессор
* жесткий диск
* оперативная память
* видеоадаптер
* звуковой адаптер
* приводы дисков
* шлейфы, кабели для подключения и связи устройств между собой и т.д.

• Монитор

• Клавиатура

• Мышь

• Периферийные устройства

**Центральный процессор** выбирает команды из памяти и выполняет их. Обычный цикл работы центрального процессора выглядит так: выборка из памяти первой команды, ее декодирование для определения ее типа и операндов, выполнение этой команды, а затем выборка, декодирование и выполнение последующих команд. Этот цикл повторяется до тех пор, пока не закончится программа. У всех центральных процессоров есть несколько собственных регистров для хранения основных переменных и промежуточных результатов. Среди специальных регистров процессора стоит выделить **счетчик команд IP** (Instruction Pointer), который содержит адрес ячейки памяти со следующей выбираемой командой, **указатель стека SP** (Stack Pointer), который ссылается на вершину текущего стека в памяти и **слово состояния программы PSW** (Program Status Word).

Есть несколько архитектур процессоров с различными наборами команд. **Аббревиатура CISC обозначает Complex Instruction Set Computer, а RISC — Reduced Instruction Set Computer**

Процессоры CISC имеют большой набор разных команд, включая сложные, а длина команды в байтах для наиболее часто использующихся команд минимальна. Это было вызвано в первую очередь экономией оперативной памяти. Идея RISC заключается в замене сложных инструкций на комбинацию простых. Так не придется заниматься сложной отладкой микрокода. Вместо этого разработчики компилятора будут решать возникающие проблемы. Еще одна основная идея RISC - это конвейеризация. Инструкции разделены на этапы, каждый из которых выполняется примерно одинаковое количество времени.

Если рассмотреть ARM RISC-процессор, то мы обнаружим пятиступенчатый конвейер инструкций.

• **(Fetch)** Извлечение инструкции из памяти и увеличение счетчика команд, чтобы извлечь следующую инструкцию в следующем такте.

• **(Decode)** Декодирование инструкции — определение, что эта инструкция делает. То есть активация необходимых для выполнения этой инструкции частей микропроцессора.

• **(Execute)** Выполнение включает использование арифметико-логического устройства (АЛУ) или совершение сдвиговых операций.

• **(Memory)** Доступ к памяти, если необходимо. Это то, что делает инструкция load.

• **(Write Back)** Запись результатов в соответствующий регистр.

Второй основной составляющей любого компьютера является **память** . В идеале память должна быть максимально быстрой (работать быстрее, чем производится выполнение одной инструкции, чтобы работа центрального процессора не замедлялась обращениями к памяти), довольно большой и чрезвычайно дешевой. Никакая современная технология не в состоянии удовлетворить все эти требования, поэтому используется другой подход. Система памяти создается в виде иерархии уровней.

Верхний уровень состоит из внутренних **регистров процессора**. Они выполнены по той же технологии, что и сам процессор, и поэтому не уступают ему в быстродействии. Следовательно, **к ним нет и задержек доступа**.

Затем следует **кэш-память**, которая управляется главным образом аппаратурой. Оперативная память разделяется на кэш-строки, обычно по 64 байт, с адресами от 0 до 63 в кэш-строке 0, адресами от 64 до 127 в кэш-строке 1 и т. д. Наиболее интенсивно используемые кэш-строки оперативной памяти сохраняются в высокоскоростной кэш-памяти, находящейся внутри процессора или очень близко к нему. Когда программе нужно считать слово из памяти, аппаратура кэша проверяет, нет ли нужной строки в кэш-памяти. Если строка в ней имеется, то происходит результативное обращение к кэш-памяти (**cache hit — кэш-попадание),** запрос удовлетворяется за счет кэш-памяти без отправки запроса по шине к оперативной памяти. Занимает по времени два такта. Отсутствие слова в кэш-памяти вынуждает обращаться к оперативной памяти, что приводит к существенной потере времени. Кэш-память из-за своей высокой стоимости ограничена в объеме. Некоторые машины имеют два или даже три уровня кэша, причем каждый из последующих медленнее и объемнее предыдущего. Доступ к кэшу первого уровня осуществляется без задержек, а доступ к кэшу второго уровня требует задержки в один или два такта.

Следующей в иерархии **идет оперативная память**. Это главная рабочая область системы памяти машины. Оперативную память часто называют оперативным запоминающим устройством (ОЗУ), или памятью с произвольным доступом (Random Access Memory (RAM)).

Дополнительно к оперативной памяти многие компьютеры оснащены небольшой по объему неизменяемой памятью с произвольным доступом — постоянным запоминающим устройством (ПЗУ), оно же память, предназначенная только для чтения (Read Only Memory (ROM)). В отличие от ОЗУ она не утрачивает своего содержимого при отключении питания, то есть является энергонезависимой. ПЗУ программируется на предприятии-изготовителе и впоследствии не подлежит изменению. Эта разновидность памяти характеризуется высоким быстродействием и дешевизной.

**Магнитный жесткий диск** состоит из одной или нескольких металлических пластин, вращающихся со скоростью до 15 000 оборотов в минуту. Информация записывается на диск в виде последовательности концентрических окружностей. В каждой заданной позиции привода каждая из головок может считывать кольцеобразный участок, называемый **дорожкой**. Из совокупности всех дорожек в заданной позиции привода **составляется цилиндр**. Каждая дорожка поделена на определенное **количество секторов**, обычно по 512 байт на сектор.

С целью дальнейшей увеличения плотности записи дорожки стали частично накладывать друг на друга. Такие диски эффективны при последовательной записи данных и редком их чтении (например, системы видеорегистрации). **При черепичной магнитной записи** (SMR - Shingled Magnetic Recording) на жесткий диск дорожки размещаются друг над другом, подобно черепице на крыше. Это позволяет повысить плотность записи. Увеличивается количество дорожек на дюйм (TPI). При перпендикулярной магнитной записи (PMR), используемой в большинстве современных дисков, данные размещаются на параллельных дорожках. Увеличение TPI путем уменьшения расстояния между дорожками благодаря технологии SMR открывает огромный простор для роста емкости жестких дисков. Конечный продукт физически выглядит и ведет себя как обычный жесткий диск с PMR, но при этом имеет большую емкость.

**Твердотельные накопители** (SSD — Solid State Disks), которые хранят данных во флэш-памяти и не имеют движущихся частей. У них то же назначение, что и у магнитных дисков, и с точки зрения операционной системы выглядят так же. SSD-диски обладают большим быстродействием, чем магнитные, но и стоимость таких дисков выше. Гибридные диски используют обе технологии. SSD-диски могут в одной ячейке хранить несколько бит данных. Чем больше бит данных в одной ячейке, тем выше емкость, ниже цена и меньше количество циклов перезаписи. SLC-ячейки выдерживают до 100000 циклов перезаписи, но хранят в одной ячейке ровно один бит. MLC - 2 бита на ячейку, TLC - 3 бита на ячейку, QLC - 4 бита на ячейку, PLC - 5 битов на ячейку.

**Видеокарта** - устройство, преобразующее цифровую информацию в форму, пригодную для дальнейшего вывода на экран монитора. Обычно видеокарта выполнена в виде печатной платы (плата расширения) и вставляется в слот расширения материнской платы, универсальный либо специализированный (AGP, PCI Express), но может быть реализована и на системной плате. Видеокарты не ограничиваются простым выводом изображения, они имеют встроенный графический процессор, который может производить дополнительную обработку, снимая эту задачу с центрального процессора компьютера.

4. Задачи операционной системы по управлению и организации работы компьютера.

С точки зрения программиста, операционная система — это программа, добавляющая ряд команд и функций к командам и функциям, выполняемым аппаратным обеспечением

Выделим три важные группы таких функций.

1. **Виртуальная память**. Механизм виртуальной памяти используется многими операционными системами. Она позволяет создать впечатление, будто у машины больше памяти, чем есть на самом деле.

2. **Файловый ввод-вывод.** Это понятие более высокого уровня, чем команды ввода-вывода, которые мы рассматривали в предыдущей главе.

3. **Параллелизм** (как организовано одновременное выполнение нескольких процессов, обмен информацией и синхронизация). Под процессом можно понимать работающую программу и всю информацию об ее состоянии (памяти, регистрах, счетчике команд, вводе-выводе и т. д.).

Если рассматривать более подробно, то самыми важными задачами управления компьютерным оборудованием, осуществляемыми операционной системой, являются следующие:

* Параллельное функционирование модулей ввода, вывода информации и процессора.
* Организация кэширования данных и выполнение согласования скоростей информационного обмена.
* Разбиение модулей и информационных данных среди процессов.
* Организация удобной работы логического интерфейса между модулями и оставшейся частью системы.
* Организация поддержки различных устройств с обеспечением возможности просто их добавить.
* Режим динамической загрузки и выгрузки драйверов.
* Обеспечение поддержки набора файловых систем.

Существует два источника поступления прерываний (interrupt) и три типа особых ситуаций (исключений, exception). Кроме того, различают внутренние (программные, software) и внешние (аппаратные, hardware) источники генерации прерываний и особых ситуаций**.**

**В зависимости от источника**, прерывания делятся на:

* **аппаратные** - возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, клавиатура, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени;
* **программные** - вызываются искусственно с помощью соответствующей команды из программы (int), предназначены для выполнения некоторых действий операционной системы, являются синхронными;
* **исключения** - являются реакцией микропроцессора на нестандартную ситуацию, возникшую внутри микропроцессора во время выполнения некоторой команды программы (деление на ноль, прерывание по флагу TF (трассировка)).

**Маскируемые** прерывания генерируются контроллером прерываний по заявке определенных периферийных устройств.

**Немаскируемые** прерывания (говорят, что оно одно, т.к. подается на вывод микропроцессора NMI) инициируют источники, требующие безотлагательного вмешательства со стороны микропроцессора.

В зависимости от типа особой ситуации различается реакция процессора на ее возникновение.

• **Ошибка (Fault)** — это особая ситуация, которая может быть исправлена обработчиком особой ситуации. При встрече ошибки состояние процессора сохраняется в том виде, каким оно было до начала выполнения команды, инициировавшей генерацию ошибки, а значения CS:EIP, указывающие на эту команду сохраняются в стеке обработчика. Прерванная программа после исправления ошибки может быть продолжена непосредственно с команды, вызвавшей эту ошибку.

• **Ловушка (Trap)** — особая ситуация, которая генерируется после выполнения соответствующей команды. В этом случае сохраняемые в стеке значения CS:EIP, указывают на команду, которая будет выполняться вслед за командой, вызвавшей ловушку; например, если ловушка произошла во время команды JMP, то сохраненные значения CS:EIP указывают на команду, являвшуюся целью команды JMP.

• **Сбой (Abort)** — это особая ситуация, которая не допускает точную локализацию вызвавшей ее команды и не допускает перезапуска. Сбои используются для сообщений о некоторых ошибках, таких как: технические неисправности и наличие некорректных значений в системных таблицах.

Обработка прерывания в реальном режиме производится в три этапа:

1. **Прекращение выполнения текущей программы**. Для этого необходимо сохранить содержимое регистров, так как они являются ресурсами, разделяемыми между программами. Обязательными для сохранения являются регистры cs, ip, flags (пара CS:IP содержит адрес команды, с которой необходимо начать выполнение после возврата, flags — состояние флагов после выполнения последней команды прерванной программы). Эти регистры сохраняются микропроцессором автоматически. Сохранение остальных регистров - должно обеспечиваться программистом. Наиболее удобным местом хранения регистров является стек. После сохранения регистров в стеке микропроцессор сбрасывает бит флага IF (т.е.=0) Этим предотвращается возможность возникновения вложенных внешних прерываний и порча регистров исходной программы вследствие неконтролируемых действий со стороны программы - обработчика вложенного прерывания. После того как необходимые действия по сохранению контекста завершены, обработчик аппаратного прерывания может разрешить вложенные прерывания командой sti.

2. **Переход к выполнению и выполнение программы обработки прерывания.** Здесь определяется источник прерывания и вызывается соответствующий обработчик прерывания. В реальном режиме микропроцессора допускается 256 источников - по количеству элементов таблицы векторов прерываний. Структура одного элемента в таблице векторов прерываний: WORD (2 байта) — значение смещения начала программы-обработчика прерывания от начала кодового сегмента WORD (2 байта) — значение базового адреса сегмента, в котором находится программа-обработчик.

Фактически, на втором этапе микропроцессор:

• по номеру источника прерывания определяет смещение в таблице векторов прерываний;

• помещает первые два байта в регистр IP;

• помещает вторые два байта в регистр CS;

• передает управление по адресу CS:IP.

Далее выполняется сама программа обработки прерывания. (Она тоже может быть прервана поступлением запроса от более приоритетного источника. Все источники прерывания имеют приоритеты.)

**3. Возврат управления прерванной программе** Необходимо привести стек в состояние, в котором он был сразу после передачи управления данной процедуре. Для этого программист должен указать необходимые действия по восстановлению регистров и очистке стека. Этот участок необходимо защитить от возможного искажения содержимого регистров (в результате появления аппаратного прерывания) с помощью команды cli. Последние команды в обработчике прерывания — sti, iret (sti — разрешить аппаратные прерывания, устанавливает флаг IF=1, не имеет операндов), iret — извлечь последовательно три слова из стека и поместить их соответственно в регистры ip, cs, flags.

5. Структура ядра и его функции. Объекты ядра. Основные операции над объектами ядра.

Под архитектурой ОС обычно понимают структурную организацию ОС на основе программных модулей. Современные ОС представляют собой хорошо структурированные модульные системы. Единой архитектуры ОС не существует, но существуют универсальные подходы к структурированию ОС

Наиболее общим подходом к структуризации ОС является подразделение модулей две группы:

• модули, выполняющие основные функции ОС — ядро ОС;

• модули, выполняющие вспомогательные функции ОС.

Модули ядра выполняют базовые функции ОС

• управление процессами;

• управление памятью;

• управление устройствами ввода-вывода.

Функции, входящие в состав ядра, можно разделить на два класса.

**1 класс. Функции для решения внутрисистемных задач организации вычислительного процесса** (переключение контекстов процессов, загрузка/выгрузка страниц, обработка прерываний). Эти функции недоступны для приложений.

**2 класс. Функции для поддержки приложений** (доступны приложениям). Эти функции создают для приложений так называемую прикладную программную среду и образуют интерфейс прикладного программирования — API. Приложения обращаются к ядру с запросами — системными вызовами. Функции API обслуживают системные вызовы — предоставляют доступ к ресурсам системы в удобной и компактной форме, без указания деталей их физического расположения.

**Ядро (kernel)**— низкоуровневая основа любой операционной системы, выполняемая аппаратурой в особом привилегированном режиме. Ядро загружается в память один раз и находится в памяти резидентно (постоянно), по одним и тем же адресам.

**Объект** — это коллекция данных, являющихся частью режима ядра операционной системы, которыми управляет Диспетчер объектов Windows (Windows Object Manager).

В качестве примера конкретных объектов можно привести: маркеры доступа (access token objects), файлы (file objects), проекции файлов (file-mapping objects), порты завершения ввода вывода (I/O completion port objects), задания (jobs), почтовые ящики (mailslot objects), мьютексы (mutex objects), каналы (pipe objects), процессы (thread objects) и ожидаемые таймеры (waitable timer objects).

Для взаимодействия с объектами ядра у операционной системы предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Например, когда мы создаем объект ядра в операционной системе Windows, функция возвращает описатель, идентифицирующий созданный объект (HANDLE). Все операции с текущим объектом ядра возможны только при указании этого описателя управляющей функции.

**Объекты ядра принадлежат ядру, а не процессу**. Это говорит о том, что, завершая работу с процессом, мы не обязательно разрушаем объект ядра. В большинстве случаев объект разрушается, но, если созданный вами объект ядра используется другим процессом, ядро запретит разрушение объекта до тех пор, пока от него не откажется последний пользователь.

Создание новых объектов, или открытие по имени уже существующих, приложение может осуществить при помощи Win32-функций, таких, как CreateFile, CreateSemaphore, OpenSemaphore и т.д.

6. Утилиты. Системные обрабатывающие программы. Библиотеки процедур. Программы дополнительных услуг.

**Утилиты** — это системное программное обеспечение, которое помогает поддерживать правильное и бесперебойное функционирование компьютерной системы.Утилиты выполняют определенные задачи, такие как обнаружение, установка и удаление вирусов, резервное копирование данных, удаление ненужных файлов и т. д.

Утилиты могут входить в состав операционных систем, идти в комплекте со специализированным оборудованием или распространяться отдельно. **Интегрированные пакеты утилит** — набор нескольких программных продуктов объединенных в единый удобный инструмент.

Типы утилит

**1. Антивирусы** Вирус — это вредоносное программное обеспечение, которое попадает в систему вместе с хост-программой. **Антивирус** — это служебное программное обеспечение, которое помогает защитить компьютер от вирусов. Более того, он уведомляет об обнаружении любого вредоносного файла и удаляет такие файлы. Кроме того, он сканирует любое новое устройство, подключенное к компьютеру, и удаляет любые вирусы, если они есть. Более того, он также время от времени сканирует систему на наличие угроз и уничтожает их.

**2. Системы управления файлами** Эти служебные программы используются для управления файлами компьютерной системы. Поскольку файлы являются важной частью системы, поскольку все данные хранятся в файлах. Таким образом, это служебное программное обеспечение помогает просматривать, искать, упорядочивать, находить информацию и быстро просматривать файлы системы. Проводник Windows — это инструмент управления файлами по умолчанию, присутствующий в системе. Другими примерами инструментов управления файлами являются Google Desktop, Double Commander, Directory Opus и т. д.

**3. Инструменты сжатия** Важной частью компьютера является место для хранения данных, очень важно поддерживать это хранилище. Поэтому мы используем определенные служебные программы для сжатия больших файлов и уменьшения их размера. Это инструменты сжатия. Формат файлов изменяется во время сжатия, и мы не можем получить к ним доступ или редактировать их напрямую. Кроме того, мы можем легко распаковать файл и вернуть исходный файл.

**4. Инструменты управления дисками** Эти служебные программы используются для управления данными на дисках. Более того, они выполняют такие функции, как разбиение устройств на разделы, управление дисками и т. д. Примерами инструментов управления дисками являются MiniTool Partition Wizard, Paragon Partition Manager и т. д.

**5. Инструменты очистки диска** Эта служебная программа помогает освободить дисковое пространство. Кроме того, с диска удаляются файлы, которые больше не используются. Примеры: Razer Cortex, Piriform CCleaner и т. д.

**6. Дефрагментация диска** Эта служебная программа помогает уменьшить фрагментацию и, следовательно, снижает скорость доступа. Дефрагментация означает переупорядочение файлов и сохранение их в смежных местах памяти. Более того, экономится время при чтении файлов и записи файлов на диск. Примерами дефрагментаторов дисков являются Perfect disk, Deflaggler и т. д.

**7. Утилита резервного копирования** Эта служебная программа помогает создавать резервные копии файлов, папок, баз данных или целых дисков. Более того, под резервным копированием подразумевается дублирование информации на диске, чтобы данные можно было восстановить в случае потери данных.

**Обрабатывающие системные программы** отличаются от управляющих программ как по своим функциям, так и по способу их инициирования (запуска).

Основные функции обрабатывающих программ:

1) перенос информации. Перенос может выполняться между различными устройствами или в пределах одного устройства. При этом под устройствами понимаются: ОП, устройства ВП, устройства ввода-вывода;

2) преобразование информации. То есть после считывания информации с устройства обрабатывающая программа преобразует эту информацию, а только затем записывает ее на это же или на другое устройство.

В зависимости от того, какая из этих двух функций является основной, обрабатывающие системные программы делятся **на утилиты и лингвистические процессоры.**

Основной функцией утилиты является перенос информации, а основная **функция лингвистического процессора** – перевод описания алгоритма с одного языка на другой. Сущность алгоритма при этом сохраняется, но форма его представления, ориентированная на программиста, преобразуется в форму, ориентированную на ЦП.

Лингвистические процессоры делятся на **трансляторы и интерпретаторы**. В результате работы транслятора алгоритм, записанный на языке программирования (исходная виртуальная программа), преобразуется в алгоритм, записанный на машинном языке. (На самом деле, машинная программа является результатом совместной работы нескольких лингвистических процессоров.)

**Трансляторы** делятся на компиляторы и ассемблеры. Исходная программа для транслятора–ассемблера записывается на языке-ассемблере. Один оператор данного языка транслируется в одну машинную команду. Исходная программа для компилятора записывается на языке программирования высокого уровня. Каждый оператор такого языка транслируется в несколько машинных команд. Примерами языков высокого уровня являются Паскаль и Си.

**Интерпретатор** в отличие от транслятора не выдаёт машинную программу целиком. Выполнив перевод очередного оператора исходной программы в соответствующую совокупность машинных команд, интерпретатор обеспечивает их выполнение. Затем преобразуется тот исходный оператор, который должен выполняться следующим по логике алгоритма, и так далее.

**Библиотеки процедур и функций** различного назначения включены в категорию вспомогательных модулей операционной системы. К ним можно отнести библиотеки математических функций, библиотеки функций ввода-вывода и т.д. Библиотеки — это набор функций, которые могут использоваться в различных программах. Библиотеки могут быть статические (библиотека привязывается к определенной программе или софт содержит данную библиотеку в своем теле.) и динамическими (библиотеки грузятся в оперативную память и используются).

В категорию **программ дополнительных услуг** входит большое количество разнообразных программ: специальный вариант пользовательского интерфейса, калькулятор, некоторые игры (какие, например, поставляются в составе ОС).

7. Понятие процесса. Системные и пользовательские процессы. Операции над процессами.

**Процесс (process)** - это некоторая часть (единица) работы, создаваемая операционной системой. Чтобы некоторую часть работы можно было назвать процессом, она должна иметь **адресное пространство**, назначаемое операционной системой, и **идентификатор**, или идентификационный номер (id процесса). Процесс должен обладать определенным статусом и иметь свой элемент в таблице процессов. В соответствии со стандартом POSIX он должен содержать один или несколько потоков управления, выполняющихся в рамках его Процесс состоит из множества выполняющихся инструкций, размещенных в адресном пространстве этого процесса. Адресное пространство процесса распределяется между инструкциями, данными, принадлежащими процессу, и стеками, обеспечивающими вызовы функций и хранение локальных переменных.

Процессы, которые выполняют системный код, называются **системными** и применяются к системе в целом. Они занимаются выполнением таких служебных задач, как распределение памяти, обмен страницами между внутренним и вспомогательным запоминающими устройствами, контроль устройств и т.п.

**Пользовательские процессы** выполняют собственный код и иногда обращаются к системным функциям. Выполняя собственный код, пользовательский процесс пребывает в пользовательском режиме (user mode).

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается ОС, совершая операции над ними. Удобно объединить их в три пары:

• создание процесса – завершение процесса;

• приостановка процесса – запуск процесса;

• блокирование процесса – разблокирование.

Еще одна операция, не имеющая парной: **изменение приоритета процесса**. Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции являются многоразовыми

**Основные операции над процессами:**

**Запуск процесса** (из числа процессов, находящихся в состоянии готовность, операционная система выбирает один процесс для последующего исполнения).

**Приостановка процесс**а (работа процесса, находящегося в состоянии исполнение, приостанавливается в результате какого-либо прерывания).

**Блокирование процесса** (процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе).

**Разблокирование процесса** (после возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло).

**Переключение контекста.**

Информация, хранимая в блоке управления процессом, делится на две части: **регистровый контекст и системный контекст**. **Регистровым контекстом процесса** называется содержимое всех регистров общего назначения процессора (включая значение программного счетчика). Содержимое остальных регистров процессора называется **системным контекстом процесса**. Информации, получаемой с регистровых и системных контекстов, достаточно для управления работой процесса в операционной системе. С точки зрения пользователя, наибольший интерес вызывает вычислительная деятельность процесса, последовательность преобразования данных и полученные результаты. **Пользовательским контекстом** называются данные, находящиеся в адресном пространстве процесса. **Совокупность регистрового, системного и пользовательского контекстов процесса называется контекстом процесса**. В любой момент времени процесс полностью характеризуется своим контекстом

8. Организация межпроцессного взаимодействия в ОС. Сигналы. Каналы. Классические проблемы межпроцессного взаимодействия.

При выполнении параллельных процессов может возникать проблема, когда каждый процесс, обращающийся к разделяемым данным, исключает для всех других процессов возможность одновременного с ним обращения к этим данным - это **называется взаимоисключением (mutual exclusion**). Ресурс, который допускает обслуживание только одного пользователя за один раз, называется **критическим ресурсом**. Если несколько процессов хотят пользоваться критическим ресурсом в режиме разделения времени, им следует синхронизировать свои действия таким образом, чтобы этот ресурс всегда находился в распоряжении не более чем одного их них.

Для организации коммуникации между одновременно работающими процессами применяются **средства межпроцессного взаимодействия** (Interprocess Communication - IPC).

Выделяются три уровня средств IPC:

• локальный;

• удаленный;

• высокоуровневый.

Средства **локального уровня** IPC привязаны к процессору и возможны только в пределах компьютера. К этому виду IPC принадлежат практически все основные механизмы IPC UNIX, а именно, каналы, разделяемая память и очереди сообщений. Коммуникационное пространство этих IPC поддерживаются только в пределах локальной системы. Из-за этих ограничений для них могут реализовываться более простые и более быстрые интерфейсы.

**Удаленные IPC** предоставляют механизмы, которые обеспечивают взаимодействие как в пределах одного процессора, так и между программами на различных процессорах, соединенных через сеть. Сюда относятся удаленные вызовы процедур (Remote Procedure Calls - RPC), сокеты Unix, а также TLI (Transport Layer Interface - интерфейс транспортного уровня) фирмы Sun.

**Под высокоуровневыми IPC** обычно подразумеваются пакеты программного обеспечения, которые реализуют промежуточный слой между системной платформой и приложением.

**Методы IPC и их применимость к разным ОС**

* Файл. Все операционные системы.
* Сигнал. Большинство операционных систем; некоторые системы, как например, Windows, только реализуют сигналы в библиотеке запуска Си, но не обеспечивают их полноценной поддержки для использования методов IPC.
* Сокет. Большинство операционных систем.
* Канал. Все системы, соответствующие POSIX.
* Именованный канал. Все системы, соответствующие POSIX.
* Семафор. Все системы, соответствующие POSIX.
* Разделяемая память. Все системы, соответствующие POSIX.
* Обмен сообщениями (без разделения). Используется в парадигме MPI, Java RMI, CORBA и других.
* Проецируемый в память файл. Все системы, соответствующие POSIX; несет риск появления состояния гонки в случае использования временного файла. Windows также поддерживает эту технологию, но использует API отличный от POSIX.
* Очередь сообщений. Большинство операционных систем.
* Почтовый ящик. Некоторые операционные системы.

**Сигнал в операционных системах семейства Unix** — асинхронное уведомление процесса о каком-либо событии, один из основных способов взаимодействия между процессами. Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный обработчик сигнала, операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Отдельные **сигналы подразделяются на три класса**:

• системные сигналы (ошибка аппаратуры, системная ошибка и т.д.);

• сигналы от устройств;

• сигналы, определенные пользователем.

**Механизм передачи сигналов** состоит из следующих частей:

• установление и обозначение сигналов в форме целочисленных значений;

• маркер в строке таблицы процессов для прибывших сигналов;

• таблица с адресами функций, которые определяют реакцию на прибывающие сигналы.

Как только сигнал приходит, он отмечается записью в таблице процессов. Если этот сигнал предназначен для процесса, то по таблице указателей функций в структуре описания процесса выясняется, как нужно реагировать на этот сигнал. При этом номер сигнала служит индексом таблицы.

Известно **три варианта реакции на сигналы**:

• вызов собственной функции обработки;

• игнорирование сигнала (не работает для SIGKILL);

• использование предварительно установленной функции обработки по умолчанию.

**Канал (pipe)** представляет собой средство связи стандартного вывода одного процесса со стандартным вводом другого. Каналы старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX. Для реализации IPC возможно использование полудуплексных и/или именованных каналов (FIFO).

После создания канала, процесс может при помощи обычного системного вызова write() выводить данные в него, а затем вводить их, вызывая соответственно функцию read(). При выполнении вызова fork() дескрипторы канала наследуются процессом-"потомком". Таким образом, оба процесса получают возможность обмениваться данными.

**Ограничением в данном случае является то, что канал должен работать лишь в одну сторону.** Либо "родитель" должен писать, а "потомок" читать, либо наоборот. Если предполагается передавать данные в обе стороны, необходимо создать два канал.

**Проблемы**

**Синхронный доступ.** Если все процессы считывают данные из файла, то в большинстве случае проблем не возникает. Однако, при попытке одним из процессов изменить этот файл, могут возникнуть так называемые конкурентные ситуации (race conditions).

**Дисциплина доступа.** Если нужно, чтобы как можно большее количество пользователей могли записывать данные, организуется так называемая очередь (по правилу «один пишет, несколько читают»). Практически организуется две очереди: одна — для чтения, другая — для записи. Такую дисциплину доступа можно организовать с помощью **барьеров** (блокировок). При этом создается общий барьер для считывателей, так как несколько процессов могут одновременно считывать данные, а также отдельный барьер для процесса-писателя. Такая организация предотвращает взаимные помехи в работе.

**Голодание процессов.** Организация дисциплины доступа может привести к ситуации, когда процесс будет длительно ждать своей очереди для записи данных. Поэтому иногда нужно организовывать очереди с приоритетами. Если нельзя точно определить, какой из процессов запрашивает или возвращает свои данные в нужный компьютер первым, используется так называемое взаимодействие по модели "клиент-сервер". При этом используются один или несколько клиентов и один сервер. Клиент посылает запрос серверу, а сервер отвечает на него. После этого клиент должен дождаться ответа сервера, чтобы продолжать дальнейшее взаимодействие. Такое поведение называется **управлением потоком**. При одновременном доступе здесь также могут использоваться очереди с приоритетами.

**Классический тупик**(deadlock) возникает, если процесс A получает доступ к файлу A и ждет освобождения файла B. Одновременно процесс B, получив доступ к файлу B, ждет освобождения файла A. Оба процесса теперь ждут освобождения ресурсов другого процесса и не освобождают при этом собственный файл.

**Отображение файла в память (на память)** - это такой способ работы с файлами в некоторых операционных системах, при котором всему файлу или некоторой непрерывной части этого файла ставится в соответствие определенный участок памяти (диапазон адресов оперативной памяти). При этом чтение данных из этих адресов фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл.

**Разделяемая память** может быть наилучшим образом описана как отображение участка (сегмента) памяти, которая будет разделена между более чем одним процессом. Это гораздо более быстрая форма IPC, потому что здесь нет никакого посредничества (т.е. каналов и т.п.). Вместо этого, информация отображается непосредственно из сегмента памяти в адресное пространство вызывающего процесса. Сегмент может быть создан одним процессом и впоследствии использован для чтения/записи любым количеством процессов.

9. Концепция потока. Параллельное исполнение потоков. Главный поток процесса.

**Поток выполнения (англ. thread — нить)** — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени)

**Преимущества потоков**

1. Код программы может выполняться, даже если некоторые действия заблокированы.

2. Легкость и быстрота их создания (по сравнению с процессами).

3. Потоки довольно просто обмениваются данными по сравнению с процессами.

4. Улучшает общую производительность системы при ожиданиях завершения операций ввода-вывода.

5. На многопроцессорных системах могут повысить быстродействие за счет параллельных вычислений.

**Недостатки потоков**

1. При программировании приложения с множественными потоками необходимо постоянно думать о потокобезопасности (thread safety).

2. Один неправильно работающий поток может повредить остальные, так как потоки делят общее адресное пространство.

3. Потоки конкурируют друг с другом в адресном пространстве. Стек и thread-local storage, занимая часть виртуального адресного пространства процесса, тем самым делают его недоступным для других потоков.

**POSIX Threads**

Это стандарт POSIX-реализации потоков. Стандарт POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) определяет API для управления потоками, их синхронизации и планирования. Библиотеки, реализующие этот стандарт (и функции этого стандарта), обычно **называются pthreads (**функции имеют приставку pthread\_). Некоторые из вызовов функций пакета Pthread: pthread\_create - создание нового потока, pthread\_exit - завершение работы вызвавшего потока, pthread\_join - ожидание выхода из указанного потока, pthread\_yield - освобождение центрального процессора, позволяющее выполняться другому потоку, pthread\_attr\_init - создание и инициализация структуры атрибутов потока, pthread\_attr\_destroy - удаление структуры атрибутов потока.

В среде Microsoft Windows концепция иная, там **процесс — это контейнер для потоков**. Процесс-контейнер содержит как минимум один поток. Если потоков в процессе несколько, приложение (процесс) становится многопоточным. Процесс — это исполнение программы. Операционная система использует процессы для разделения исполняемых приложений. Поток — это основная единица, которой операционная система выделяет время процессора.

**Под словами "параллельный поток"** или просто "поток" мы будем понимать объект, выполняемый параллельно с основным потоком приложения и с другими параллельными потоками

**Многопоточность:** процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. Модель конкурентного выполнения сопрограмм (concurrency) - это немного более широкий термин, чем параллелизм (parallelism). Данная модель предполагает, что несколько задач могут выполняться одновременно, но не говорит, как это должно быть достигнуто. Резюмируя вышесказанное, **многопроцессность** — это форма параллелизма (которого также можно добиться многопоточностью), **а параллелизм** в свою очередь, это подмножество конкурентного выполнения сопрограмм.

Среди множества различных видов взаимодействия параллельных потоков обзорно рассмотрим только несколько наиболее важных видов: совместное использование разделяемых данных, асинхронное взаимодействие, синхронное взаимодействие

Задача, которую нужно решить **при совместном использовании данных** заключается в обеспечении атомарности (неделимости) изменения данных, что иногда обозначается термином "транзакция". Иными словами, необходим взаимно-исключающий доступ к данным - недопустимо, чтобы один поток читал данные, в то время как другой поток их изменял. Наиболее простое и эффективное средство - это использование **критической секции**. У критической секции есть одно очень важное достоинство - высокая эффективность, поскольку критическая секция - это не объект ядра операционной системы, а запись (record) в адресном пространстве приложения, именно поэтому критическая секция решает задачу синхронизации потоков только в рамках одного приложения. Альтернативное и более универсальное средство - мьютекс

**Асинхронное взаимодействие** Это очень универсальный и весьма эффективный способ взаимодействия параллельных потоков. Суть его состоит во введении посредника-буфера между параллельными потоками. Иногда этот способ называют взаимодействием с помощью обмена сообщениями. Поток-писатель, желающий послать другому потоку некоторые данные, записывает их в очередь. Поток-читатель ожидает появления данных в очереди, получает их, обрабатывает и, если нужно, посылает результат в очередь потока-писателя. Принципиальный недостаток асинхронного взаимодействия - необходимость буфера неопределенного размера. Этот недостаток можно сгладить, если ввести ограничение на размер очереди - в этом случае поток-писатель должен приостанавливаться, если очередь достигла своего максимального размера. Важная проблема эффективности многопоточной обработки возникает в том случае, если поток должен обрабатывать несколько очередей входящих сообщений или несколько событий. Единственный способ достигнуть этого - ожидать именно первого из многих возможных событий.

**Синхронное взаимодействие** будем рассматривать только применительно к двум параллельным потокам. Ограничимся только простым случаем парного синхронного взаимодействия - асимметричное рандеву. **Суть этого способа состоит в том, что оба взаимодействующих потока подходят к точке синхронизации, обмениваются данными и затем продолжают работать параллельно и независимо.** Если один из потоков подошел к точке синхронизации раньше, то он дожидается партнера. Асимметричность выражается в том, что потоки играют при встрече разные роли. Поток-отправитель несет основную, активную, нагрузку, а поток-получатель более пассивен. Рандеву фактически означает совмещение синхронизации и обмена данными. Предлагаемая реализация рандеву представляет собой **объект-канал**, по которому происходит взаимодействие. Обмен данными в этой реализации состоит в том, что поток-отправитель вызывает у потока-получателя некоторый метод, передавая данные как аргумент и принимая реакцию как результат

По умолчанию процесс создается с одним потоком, **называемым главным** или основным потоком. В Windows процесс может породить практически неограниченное количество потоков. Для этого используется функция CreateThread. Процесс будет активен, пока активен хотя бы один поток. Но при этом следует различать фоновые и обычные, не фоновые потоки.

10. Диаграммы состояния потоков. Понятие контекста и переключения контекста.

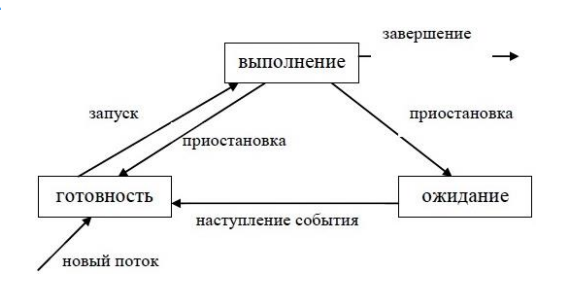
Для каждого созданного потока в системе предусматриваются три возможных его состояния:

• состояние выполнения, когда код потока выполняется процессором; на однопроцессорных платформах в этом состоянии в каждый момент времени может находиться только один поток;

• состояние готовности к выполнению, когда поток готов продолжать свою работу и ждет освобождения ЦП;

• состояние ожидания наступления некоторого события; в этом случае поток не претендует на время ЦП, пока не наступит определенное событие (завершение операции ввода/вывода, освобождение необходимого потоку занятого ресурса, сигнала от другого потока), часто такие потоки называют блокированными.

Изменение состояния потока происходит в результате соответствующих действий. Удобно для этих целей использовать следующую диаграмму состояний и переходов



**Переключение контекста (context switch)** — в многозадачных ОС и средах — процесс прекращения выполнения процессором одной задачи (процесса, потока, нити) с сохранением всей необходимой информации и состояния, необходимых для последующего продолжения с прерванного места, и восстановления и загрузки состояния задачи, к выполнению которой переходит процессор.

С точки зрения прикладного уровня переключение контекста можно разделить **на добровольное (voluntary) и принудительное (non-voluntary):** выполняющийся процесс/поток может сам передать управление другому потоку либо ядро может насильно отобрать у него управление.

11. Многозадачность в ОС. Типы многозадачности.

**Многозадачность (multitasking)** — свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач. Истинная многозадачность операционной системы возможна только в распределённых вычислительных системах

Предположим, что процесс проводит часть своего времени p в ожидании завершения операций ввода-вывода. При одновременном присутствии в памяти n процессов вероятность того, что все n процессов ожидают завершения ввода-вывода (в случае чего процессор простаивает), равна p·n. Тогда время задействования процессора вычисляется по формуле:

**Время задействования центрального процессора = 1 – p^n .**

Существует два типа многозадачности:

• Процессная многозадачность (основанная на процессах — одновременно выполняющихся программах). Здесь программа — наименьший элемент управляемого кода, которым может управлять планировщик операционной системы. Более известна большинству пользователей (работа в текстовом редакторе и прослушивание музыки).

• Поточная многозадачность (основанная на потоках). Наименьший элемент управляемого кода — поток (одна программа может выполнять 2 и более задачи одновременно).

**Многопоточность** — специализированная форма многозадачности.

К псевдопараллельной многозадачности можно отвести следующие типы: простое переключение, совместная (или кооперативная) многозадачность и вытесняющая (или приоритетная) многозадачность

**Простое переключение** Тип многозадачности, при котором операционная система одновременно загружает в память два или более приложений, но процессорное время предоставляется только основному приложению. Для выполнения фонового приложения оно должно быть активизировано. **Преимущества**: можно задействовать уже работающие программы, написанные без учёта многозадачности. **Недостатки:** невозможно в неинтерактивных системах, работающих без участия человека. Взаимодействие между программами крайне ограничено.

**Совместная (или кооперативная) многозадачность** Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам. При кооперативной многозадачности приложение может захватить фактически столько процессорного времени, сколько оно считает нужным. Все приложения делят процессорное время, периодически передавая управление следующей задаче. **Преимущества кооперативной многозадачности**: отсутствие необходимости защищать все разделяемые структуры данных объектами типа критических секций и мьютексов, что упрощает программирование, особенно перенос кода из однозадачных сред в многозадачные**. Недостатки:** неспособность всех приложений работать в случае ошибки в одном из них, приводящей к отсутствию вызова операции «отдать процессорное время». Крайне затрудненная возможность реализации многозадачной архитектуры ввода-вывода в ядре ОС, позволяющей процессору исполнять одну задачу, в то время как другая задача инициировала операцию ввода-вывода и ждет её завершения

**Вытесняющая (или приоритетная) многозадачность** Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определённый приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами.  Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя. В основе метода лежат два важнейших и достаточно понятных принципа: квантование времени ЦП и приоритеты потоков.

**Квантование** означает, что каждому потоку система выделяет определенный интервал времени (квант), в течение которого процессор потенциально может выполнять код этого потока. По завершении выделенного кванта планировщик принудительно переключает процессор на выполнение другого готового потока (если, конечно, такой есть), переводя старый активный поток в состояние готовности. Это гарантирует, что ни один поток не захватит ЦП на непозволительно большое время (как было в более ранних системах с так называемой невытесняющей или кооперативной многозадачностью). Конечно, выделенный квант поток может и не использовать до конца, если в процессе своего выполнения он нормально или аварийно завершится, или потребует наступления некоторого события, или будет прерван системой. Типичный диапазон изменения кванта - от 10 до 50 миллисекунд.

**Приоритет** определяет важность потока и влияет на частоту запуска потока и, возможно, на величину выделяемого кванта. Интуитивно понятно, что потоки могут иметь разную степень важности: системные – более высокую (иначе ОС не сможет решать свои задачи), прикладные – менее высокую. Многие ОС позволяют группировать потоки по их важности, выделяя три группы, или класса: потоки реального времени с максимально высоким уровнем приоритета; системные потоки с меньшим уровнем приоритета; прикладные потоки с самым низким приоритетом. Если приоритет потока может меняться системой, то такие приоритеты называют динамическими, иначе - фиксированными.

Компьютерная программа в целом или её отдельная процедура называется **реентерабельной** (reentrant — повторно входимый), если она разработана таким образом, что одна и та же копия инструкций программы в памяти может быть совместно использована несколькими пользователями или процессами При этом второй пользователь может вызвать реентерабельный код до того, как с ним завершит работу первый пользователь и это как минимум не должно привести к ошибке, а при корректной реализации не должно вызвать потери вычислений. Для обеспечения реентерабельности необходимо выполнение нескольких условий: никакая часть вызываемого кода не должна модифицироваться; вызываемая процедура не должна сохранять информацию между вызовами; если процедура изменяет какие-либо данные, то они должны быть уникальными для каждого пользователя; процедура не должна возвращать указатели на объекты, общие для разных пользователей.

12. Иерархия, приоритеты и планирование потоков. Динамические уровни приоритетов.

13. Синхронизация и взаимоблокировка ресурсов. Механизмы синхронизации.  
Ситуации, в которых два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой из них был первым, **называются состояниями состязания**

Чтобы избежать состязательной ситуации, нужно найти способ взаимного исключения, то есть некий способ, обеспечивающий правило, при котором если общие данные или файл используются одним процессом, возможность их использования всеми другими процессами исключается.

Та часть программы, в которой используется доступ к общей памяти, называется **критической областью или критической секцией.** Критическая область — часть программы, в которой есть обращение к совместно используемым данным. Соответственно, критический ресурс — тот ресурс, к которому осуществляется одновременный доступ

**Условия для эффективной совместной работы процессов**

1. Два процесса не могут одновременно находиться в своих критических областях.

2. Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.

3. Никакие процессы, выполняемые за пределами своих критических областей, не могут блокироваться любым другим процессом.

4. Процессы не должны находиться в вечном ожидании входа в свои критические области.

Для достижения режима взаимного исключения (при котором, пока один процесс занят обновлением общей памяти и находится в своей критической области, никакой другой процесс не сможет войти в свою критическую область и создать проблему) можно предложить различные решения.

**Запрещение прерываний** В однопроцессорных системах простейшим решением является запрещение всех прерываний каждым процессом сразу после входа в критическую область и их разрешение сразу же после выхода из критической области. При запрещении прерываний не могут осуществляться никакие прерывания по таймеру. Поскольку центральный процессор переключается с одного процесса на другой в результате таймерных или каких-нибудь других прерываний, то при выключенных прерываниях он не сможет переключиться на другой процесс. Поскольку процесс запретил прерывания, он может исследовать и обновлять общую память, не опасаясь вмешательства со стороны любого другого процесса. Но применение такого метода в пользовательском режиме вполне может привести к краху системы при некорректном поведении приложения. А на многопроцессорных системах он и вовсе бесполезен, так прерывания будут запрещены на одном процессоре, а другие процессоры смогут обратиться к критическим ресурсам.

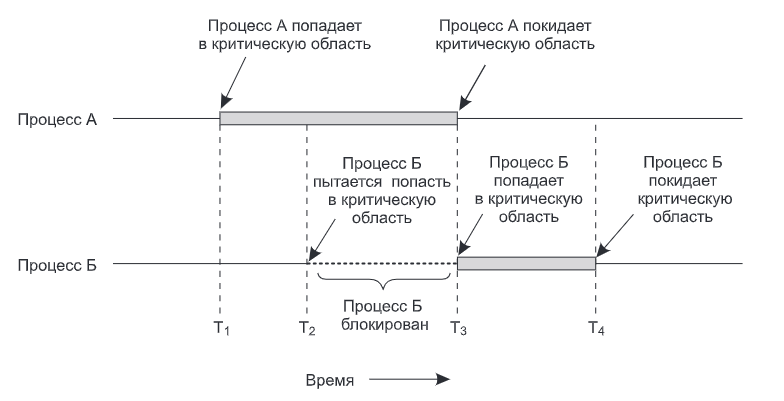
**Блокирующие переменные** В качестве второй попытки рассмотрим программное решение, в котором используется одна общая (блокирующая) переменная, исходное значение которой равно нулю. Когда процессу требуется войти в свою критическую область, сначала он проверяет значение блокирующей переменной. Если оно равно 0, процесс устанавливает его в 1 и входит в критическую область. Если значение уже равно 1, процесс просто ждет, пока оно не станет равно нулю. Таким образом, нулевое значение показывает, что ни один из процессов не находится в своей критической области, а единица — что какой-то процесс находится в своей критической области. На самом деле такое решение не защищает от состязания: существует некоторый момент времени между проверкой значения переменной и ее изменением, в который может вклиниться другой процесс.

**Синхронизация предполагает наличие общей памяти**. В случае потоков проблемы нет, так как потоки разделяют адресное пространство процесса. Но у процессов разобщенные адресные пространства, поэтому нужно найти какое-то решение. Во-первых, некоторые общие структуры данных, например семафоры, могут храниться в ядре и быть доступны только через системные вызовы. Такой подход позволяет устранить проблему. Во-вторых, большинство современных операционных систем (включая UNIX и Windows) предлагают процессам способ, позволяющий использовать некоторую часть их адресного пространства совместно с другими процессами. В этом случае допускается совместное использование буферов и других структур данных. В худшем случае, когда нет доступа ни к чему другому, можно воспользоваться общим файлом

Windows предоставляет четыре других объекта, предназначенных для синхронизации потоков и процессов. Три из них — мьютексы, семафоры и события — являются объектами ядра, имеющими дескрипторы. События используются также для других целей, например, для асинхронного ввода/вывода. Мы начнем обсуждение с четвертого объекта, а именно, объекта критического участка кода CRITICAL\_SECTION.

Та часть программы, в которой используется доступ к общей памяти, называется критической областью или критической секцией.

Для инициализации и удаления переменной типа CRITICAL\_SECTION используются, соответственно, функции InitializeCriticalSection и DeleteCriticalSection. Функция EnterCriticalSection блокирует поток, если на данном критическом участке кода присутствует другой поток. Ожидающий поток разблокируется после того, как другой поток выполнит функцию LeaveCriticalSection. Говорят, что поток получил права владения объектом CS, если произошел возврат из функции EnterCriticalSection, тогда как для уступки прав владения используется функция LeaveCriticalSection



**Мьютекс** — это совместно используемая переменная, которая может находиться в одном из двух состояний: заблокированном или незаблокированном. Следовательно, для их представления нужен только один бит, но на практике зачастую используется целое число, при этом нуль означает незаблокированное, а все остальные значения—заблокированное состояние. Объект взаимного исключения (mutual exception), или мьютекс (mutex), обеспечивает более универсальную функциональность по сравнению с объектом CRITICAL\_SECTION. Поскольку мьютексы могут иметь имена и дескрипторы, их можно использовать также для синхронизации потоков, принадлежащих различным процессам.

При работе с мьютексами мы будем пользоваться функциями CreateMutex, ReleaseMutex и OpenMutex. Функция OpenMutex открывает существующий именованный мьютекс. Эта функция дает возможность потокам, принадлежащим различным процессам, синхронизироваться так, как если бы они принадлежали одному и тому же процессу. Вызову функции OpenMutex в одном процессе должен предшествовать вызов функции CreateMutex в другом процессе. Функция ReleaseMutex освобождает мьютекс, которым владеет вызывающий поток. Если мьютекс не принадлежит потоку, функция завершается с ошибкой.

**Семафор**

Объекты второго из трех упомянутых в начале главы типов объектов синхронизации ядра — семафоры (semaphores), поддерживают счетчики, и когда значение этого счетчика больше 0, объект семафора находится в сигнальном состоянии. Если же значение счетчика становится нулевым, объект семафора переходит в несигнальное состояние. Потоки и процессы организуют ожидание обычным способом, используя для этого одну или несколько функций ожидания. При разблокировании ожидающего потока значение счетчика уменьшается на 1.

К функциям управления семафорами относятся CreateSemaphore, OpenSemaphore и ReleaseSemaphore, причем последняя функция может инкрементировать значение счетчика на 1 и более. Эти функции аналогичны своим эквивалентам, предназначенным для управления мьютексами. Семафор может быть освобожден любым потоком, а не только тем, который ожидает

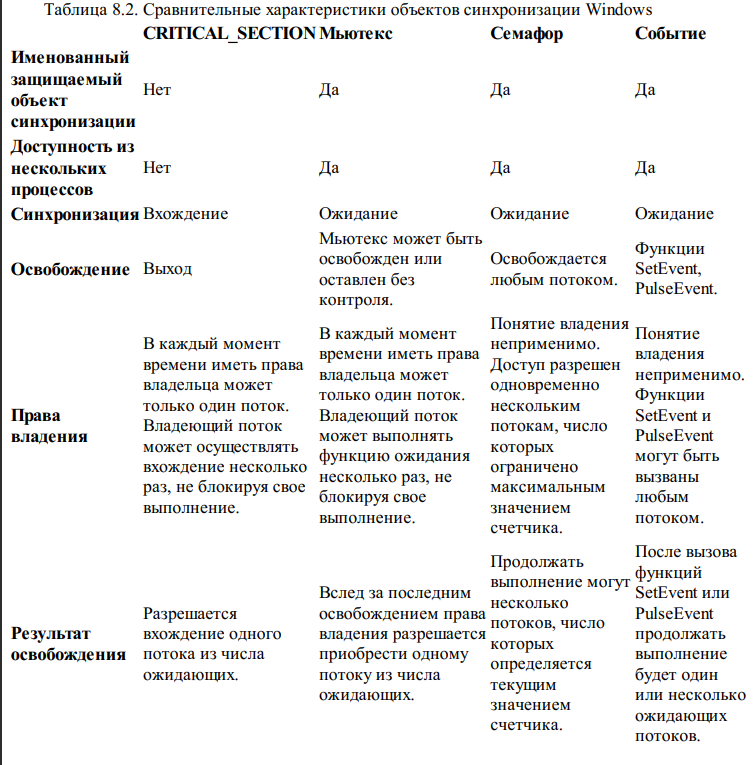
**Последним из рассматриваемых нами типов объектов синхронизации ядра являются события (events).** Объекты события используются для того, чтобы сигнализировать другим потокам о наступлении какого-либо события, например, о появлении нового сообщения. Важной дополнительной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько потоков. Объекты события делятся на сбрасываемые вручную и автоматически сбрасываемые, и это их свойство устанавливается при вызове функции CreateEvent.

• Сбрасываемые вручную события (manual-reset events) могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в несигнальное состояние программно.

• Автоматически сбрасываемые события (auto-reset event) сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в сигнальное состояние.

События используют пять новых функций: CreateEvent, OpenEvent, SetEvent, ResetEvent и PulseEvent. Для открытия именованного объекта события используется функция OpenEvent, причем это может сделать и другой процесс. Поток может установить событие в сигнальное состояние, используя функцию SetEvent. Если событие является автоматически сбрасываемым, то оно автоматически возвращается в несигнальное состояние уже после освобождения только одного из ожидающих потоков. В отсутствие потоков, ожидающих наступления этого события, оно остается в сигнальном состоянии до тех пор, пока такой поток не появится, после чего этот поток сразу же освобождается.

С другой стороны, если событие является сбрасываемым вручную, то оно остается в сигнальном состоянии до тех пор, пока какой-либо поток не вызовет функцию ResetEvent, указав дескриптор этого события в качестве аргумента. В это время все ожидающие потоки освобождаются, но до выполнения такого сброса события другие потоки могут как переходить в состояние его ожидания, так и освобождаться. Функция PulseEvent освобождает все потоки, ожидающие наступления сбрасываемого вручную события, но после этого событие сразу же автоматически сбрасывается. В случае же использования автоматически сбрасываемого события функция PulseEvent освобождает только один ожидающий поток, если таковые имеются.



14. Взаимоблокировка ресурсов в многозадачных системах. Решение задачи взаимоблокировки ресурсов.

**Взаимоблокировка** в группе процессов возникает в том случае, если каждый процесс из этой группы ожидает события, наступление которого зависит исключительно от другого процесса из этой же группы.

Коффман (Coffman et al., 1971) показал, что для возникновения ресурсных взаимоблокировок должны выполняться четыре условия:

1. Условие взаимного исключения. Каждый ресурс либо выделен в данный момент только одному процессу, либо доступен.

2. Условие удержания и ожидания. Процессы, удерживающие в данный момент ранее выделенные им ресурсы, могут запрашивать новые ресурсы.

3. Условие невыгружаемости. Ранее выделенные ресурсы не могут быть принудительно отобраны у процесса. Они должны быть явным образом высвобождены тем процессом, который их удерживает.

4. Условие циклического ожидания. Должна существовать кольцевая последовательность из двух и более процессов, каждый из которых ожидает высвобождения ресурса, удерживаемого следующим членом последовательности. Для возникновения ресурсной взаимоблокировки должны соблюдаться все четыре условия. Если одно из них не соблюдается, ресурсная взаимоблокировка невозможна.

**Чаще всего для борьбы с взаимными блокировками используются четыре стратегии:**

1. **Игнорирование проблемы**. Может быть, если вы проигнорируете ее, она проигнорирует вас.

2. **Обнаружение и восстановление**. Дайте взаимоблокировкам проявить себя, обнаружьте их и выполните необходимые действия.

3. **Динамическое уклонение** от них за счет тщательного распределения ресурсов.

4. **Предотвращение за счет структурного подавления одного из четырех условий**, необходимых для их возникновения.

**Обнаружение взаимоблокировок и восстановление работоспособности** При использовании этой технологии система не пытается предотвращать взаимоблокировки. Она позволяет им произойти, пытается обнаружить момент их возникновения, а затем предпринимает некоторые действия по восстановлению работоспособности.

**Восстановление за счет приоритетного овладения ресурсом**. Возможность отобрать ресурс у процесса, позволить использовать его другому процессу, а затем вернуть его без извещения процесса во многом зависит от природы этого ресурса. Восстановление этим способом зачастую затруднено или вовсе невозможно. Выбор процесса для приостановки обусловлен тем, какой именно процесс обладает тем ресурсом, который у него можно легко отобрать.

**Восстановление путем отката.** Если разработчики системы и операторы вычислительной машины знают о том, что есть вероятность возникновения взаимоблокировки, они могут организовать периодическое создание процессами контрольных точек. Это означает, что состояние процесса записывается в файл, что позволит осуществить его последующий перезапуск. Чтобы выйти из взаимоблокировки, процесс, владеющий необходимым ресурсом, откатывается назад к точке, предшествующей получению данного ресурса, для чего он запускается из одной из своих контрольных точек. Вся работа, выполненная после этой контрольной точки, теряется. Восстановление путем уничтожения процессов. Самым грубым, но и самым простым способом прервать взаимоблокировку является уничтожение одного или нескольких процессов. Можно уничтожить процесс, находящийся в цикле взаимоблокировки. Если повезет, то другие процессы смогут продолжить свою работу. Если это не поможет, то все можно повторить, пока цикл не будет разорван

**Уклонение от взаимоблокировок** Мы предполагали, что, когда процесс запрашивает ресурсы, он просит их все сразу. Но в большинстве систем ресурсы запрашиваются по одному. Система должна уметь принимать решение, представляет выделение ресурса опасность или нет, и выделять его только в том случае, если это безопасно. И при определенных условиях взаимоблокировки можно избежать, но только если заранее будет доступна вполне определенная информация

**Предотвращение взаимоблокировки** Если мы сможем гарантировать, что хотя бы одно из четырех условий возникновения взаимоблокировок никогда не будет выполнено, то взаимоблокировки станут структурно невозможными.

15. Компьютерное время. Ожидаемые таймеры.

**Unix-время (также POSIX-время)** — система описания моментов во времени, принятая в Unix и других POSIX-совместимых операционных системах. Определяется как количество секунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года; этот момент называют эпохой Unix. Время отсчитывается по часовому поясу Гринвичского меридиана без перехода на летнее время (так называемое «Универсальное координированное время» UTC — Coordinated Universal Time). В настоящее время в POSIX-системах тип time\_t эквивалентен типу long. Диапазона значений типа long (32 бита) достаточно для представления дат, лежащих в отрезке длиной примерно 136 лет. Поскольку отрицательные числа используются для представления дат до 1970 года, переполнение текущего представления типа time\_t может произойти в 2038 году, если, конечно, до этого момента представление не будет изменено. Для более удобного представления календарного времени используется тип struct tm, в котором каждый компонент даты и времени хранится в отдельном целочисленном поле.

**Функция time** позволяет получить текущее системное календарное время. **Функции localtime, gmtime** преобразовывают дату из секундного представления time\_t в структурное представление struct tm. **Функция mktime** конвертирует календарное время местного часового пояса в развёрнутом структурном формате в секундное представление. Функции ctime и asctime конвертируют календарное время в символьную строку. Функция strftime позволяет получать строковое представление времени согласно заданной форматной строке.

**Ядро таймеров POSIX.1b представляет собой набор часов, которые используются как привязка ко времени**. Linux обеспечивает поддержку следующих часов:

**CLOCK\_REALTIME**: общесистемные часы реального времени, видимые для всех процессов, работающих в системе. Часы измеряют количество времени в секундах и наносекундах с начала эпохи. Разрешение часов равно 1/HZ секунд. Таким образом, если HZ равно 100, то разрешение часов составляет 10 мс. Если HZ равно 1000, то разрешение часов составляет 1 мс. Так как это время базируются на времени настенных часов, оно может быть изменено.

**CLOCK\_MONOTONIC**: время непрерывной работы системы, видимое всем процессам в системе. В Linux оно измеряется как количество времени в секундах и наносекундах после загрузки системы. Его разрешение равно 1/HZ с. Это время не может быть изменено каким-либо процессом.

**CLOCK\_PROCESS\_CPUTIME\_ID**: часы, измеряющие время работы процесса. Время текущего процесса, потраченное на выполнение в системе, измеряется в секундах и наносекундах. Разрешение равно 1/HZ. Это время может быть изменено.

**CLOCK\_THREAD\_CPUTIME\_ID**: То же, что и CLOCK\_PROCESS\_CPUTIME\_ID, но для текущего потока. Обычно CLOCK\_REALTIME используется для указания абсолютного времени ожидания. CLOCK\_MONOTONIC используется для относительного времени ожидания и периодических задач. Поскольку время этих часов не может быть изменено, периодическим задачам не нужно беспокоиться о преждевременном или задержанном пробуждении, которое могло бы произойти с CLOCK\_REALTIME. Двое других часов могут использоваться для целей учёта.

 Персональные компьютеры для измерения времени использует или системный таймер (меньшая точность) или часы реального времени (большая точность).

**Системный таймер** служит для отслеживания хода времени ядром. Независимо от аппаратной платформы, идея, которая лежит в основе системного таймера, одна и та же - это обеспечение механизма управления прерываниями, которые возникают периодически с постоянной частотой. Для некоторых аппаратных платформ это реализуется с помощью электронных часов, которые генерируют колебания с программируемой частотой. В других аппаратных платформах используется декрементный счетчик (decrementer), куда можно записать некоторое начальное значение, которое будет периодически, с фиксированной частотой, уменьшаться на единицу, пока значение счетчика не станет равным нулю. Когда значение счетчика становится равным нулю, генерируется прерывание. В любом случае эффект получается один и тот же.

**Часы реального времени** (ЧРВ, RTC - Real Time Clock) — электронная схема, предназначенная для учёта хронометрических данных (текущее время, дата, день недели и др.), представляет собой систему из автономного источника питания и учитывающего устройства. Чаще всего часы реального времени встречаются в вычислительных машинах, хотя на самом деле ЧРВ присутствуют практически во всех электронных устройствах, которые должны хранить время.

**HPET** (High Precision Event Timer, таймер событий высокой точности) - тип таймера, используемый в PC-совместимых компьютерах. Совместно разработан Intel и Microsoft, и стал внедряться в чипсеты с 2005 года.

**Счетчик меток времени (TSC)** - это 64-битный регистр, присутствующий во всех процессорах x86, начиная с Pentium. Он подсчитывает количество циклов процессора с момента его сброса. Инструкция RDTSC возвращает TSC в EDX:EAX. В режиме x86-64 RDTSC также очищает старшие 32 бита RAX и RDX.

**Время Windows** - это количество времени в миллисекундах, прошедшее с момента последнего запуска системы. Функция GetTickCount или GetTickCount64 возвращают текущее время Windows.

**Время прерывания** - это время с момента последнего запуска системы в интервалах по 100 наносекунд. Счетчик времени прерываний начинается с нуля при запуске системы и увеличивается при каждом прерывании часов на длину такта часов. Точная длина тактов часов зависит от базового оборудования и может отличаться в разных системах. В отличие от системного времени, количество прерываний не подлежит корректировке пользователями или службой времени Windows, что делает его лучшим выбором для измерения коротких длительностей.

**Объект таймера ожидания** - это объект синхронизации, состояние которого по достижении указанного срока устанавливается в значение Signaled. Существует два типа таймеров ожидания, которые можно создать: **сброс вручную и синхронизация.** Таймер любого типа также может быть периодическим.

**Таймер сброса вручную** - таймер, состояние которого остается сигнальным до вызова SetWaitableTimer, чтобы установить новое время выполнения.

**Таймер синхронизации** - таймер, состояние которого остается сигнальным до тех пор, пока поток не завершит операцию ожидания в объекте таймера.

**Периодический таймер** - таймер, который повторно активируется каждый раз, когда истечет указанный период, пока таймер не будет сброшен или отменен.

**Периодический таймер** - это либо периодический таймер сброса вручную, либо периодический таймер синхронизации.

Поток использует функцию CreateWaitableTimer или CreateWaitableTimerEx **для создания объекта таймера.** Поток создания указывает, является ли таймер таймером сброса вручную или таймером синхронизации. Создающий поток может указать имя объекта таймера. Потоки в других процессах могут открывать дескриптор для существующего таймера, указывая его имя в вызове функции OpenWaitableTimer. Любой поток с дескриптором объекта таймера может использовать одну из функций ожидания для ожидания, пока состояние таймера будет задано как сигнальное. Поток вызывает функцию SetWaitableTimer для активации таймера. Поток может использовать функцию CancelWaitableTimer для установки таймера в неактивное состояние. Чтобы сбросить таймер, вызовите SetWaitableTimer. Завершив работу с объектом таймера, вызовите CloseHandle, чтобы закрыть дескриптор для объекта таймера.

16. Управление памятью: адресное пространство процесса, организация памяти, основные механизмы управления памятью, концепция рабочего множества.

Часть операционной системы, которая управляет иерархией памяти (или ее частью**), называется менеджером, или диспетчером, памяти**. Он предназначен для действенного управления памятью и должен следить за тем, какие части памяти используются, выделять память процессам, которые в ней нуждаются, и освобождать память, когда процессы завершат свою работу

**Адресное пространство** - это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется собственное адресное пространство, независимое от того адресного пространства, которое принадлежит другим процессам (за исключением тех особых обстоятельств, при которых процессам требуется совместное использование их адресных пространств).

Если память распределяется в динамическом режиме, то управлять этим должна операционная система.

Когда процессы и пустые пространства содержатся в списке отсортированными по адресам, то для выделения памяти создаваемому процессу (или существующему процессу, загружаемому в результате свопинга с диска) могут быть использованы несколько алгоритмов. Предположим, что диспетчер памяти знает, сколько памяти нужно выделить. Простейший алгоритм **называется «первое подходящее».** Диспетчер памяти сканирует список сегментов до тех пор, пока не найдет пустое пространство подходящего размера. Затем пустое пространство разбивается на две части: одна для процесса и одна для неиспользуемой памяти, за исключением того статистически маловероятного случая, когда процесс в точности помещается в пустое пространство.

Незначительной вариацией алгоритма «первое подходящее» является алгоритм **«следующее подходящее».** Он работает так же, как и «первое подходящее», за исключением того, что отслеживает свое местоположение, как только находит подходящее пустое пространство. При следующем вызове для поиска пустого пространства он начинает поиск в списке с того места, на котором остановился в прошлый раз, а не приступает к поиску с самого начала, как при работе алгоритма «первое подходящее».

Другой хорошо известный и широко используемый алгоритм **— «наиболее подходящее».** При нем поиск ведется по всему списку, от начала до конца, и выбирается наименьшее соответствующее пустое пространство. Вместо того чтобы разбивать большое пустое пространство, которое может пригодиться чуть позже, алгоритм «наиболее подходящее» пытается подыскать пустое пространство, близкое по размеру к необходимому, чтобы наилучшим образом соответствовать запросу и имеющимся пустым пространствам.

При попытке обойти проблему разбиения практически точно подходящих пространств памяти на память, отводимую под процесс, и небольшие пустые пространства можно прийти к идее алгоритма **«наименее подходящее»,** то есть к неизменному выбору самого большого подходящего пустого пространства, чтобы вновь образующееся пустое пространство было достаточно большим для дальнейшего использования.

Еще один алгоритм распределения памяти называется **«быстро искомое подходящее»**. Его использование предусматривает ведение отдельных списков для некоторых наиболее востребованных искомых размеров.

В основе **виртуальной памяти** лежит идея, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые страницами. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы одновременное присутствие в памяти всех страниц необязательно. Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету. Когда программа ссылается на часть своего адресного пространства, которое не находится в физической памяти, операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду.

Виртуальные адреса не выставляются напрямую на шине памяти. Вместо этого они поступают в **диспетчер памяти (Memory Management Unit (MMU))**, который отображает виртуальные адреса на адреса физической памяти

**Блок управления памятью (MMU)** — это аппаратный компонент компьютера, который обрабатывает все операции с памятью и кэшированием, связанные с процессором. Другими словами, MMU отвечает за все аспекты управления памятью. Обычно он интегрирован в процессор, хотя в некоторых системах он занимает отдельную интегральную схему (ИС).

Основные различия между логической и физической памятью заключаются в следующем:

**Логическая память.** Адреса логической памяти, также называемые виртуальным адресным пространством, создаются ЦП и выполняют множество различных функций. Логическую память также называют виртуальной памятью, поскольку ее размер может динамически изменяться в зависимости от требований обработки данных пользователя.

**Физическая память**. Напротив, MMU вычисляет физические адреса. Затем они загружаются в регистр адреса памяти подсистемы памяти, который также называется физической памятью. Когда физические адреса содержат логические адреса, они называются физическим адресным пространством. В отличие от логических адресов, физические адреса не меняются.

**Адресное пространство процесса** состоит из диапазона адресов, которые выделены процессу, и, что более важно, в этом диапазоне выделяются адреса, которые процесс может так или иначе использовать.

**Рабочий набор процесса** — это набор страниц в виртуальном адресном пространстве процесса, которые в настоящее время находятся в физической памяти. Рабочий набор содержит только страничные выделения памяти. В рабочий набор не включаются ресурсы памяти, не допускающие подкачку, такие как расширения адресных окон (AWE) или большие страницы

Память, принадлежащая процессу, неявно защищена его частным виртуальным адресным пространством.

**Защита памяти (Memory protection)** — это способ управления правами доступа к отдельным регионам памяти. Используется большинством многозадачных операционных систем. Основной целью защиты памяти является запрет доступа процессу к той памяти, которая не выделена для этого процесса. Такие запреты повышают надёжность работы как программ, так и операционных систем, так как ошибка в одной программе не может повлиять непосредственно на память других приложений.

17. Классификация запоминающих устройств. Иерархия памяти. Оперативные и постоянные запоминающие устройства.

**Запоминающее устройство** — носитель информации, предназначенный для записи и хранения данных. В основе работы запоминающего устройства может лежать любой физический эффект, обеспечивающий приведение системы к двум или более устойчивым состояниям.

**Классификация запоминающих устройств по устойчивости записи и возможности перезаписи:**

• Постоянные (ПЗУ), содержание которых не может быть изменено конечным пользователем (например, BIOS). ПЗУ в рабочем режиме допускает только считывание информации.

• Записываемые (ППЗУ), в которые конечный пользователь может записать информацию только один раз (например, CD-R).

• Многократно перезаписываемые (ПППЗУ) (например, CD-RW).

• Оперативные (ОЗУ) — обеспечивают режим записи, хранения и считывания информации в процессе её обработки. Быстрые, но дорогие ОЗУ (SRAM) строят на триггерах, более медленные, но более дешёвые разновидности ОЗУ — динамические ЗУ (DRAM) строят на элементах, состоящих из ёмкости (конденсатора) и полевого транзистора, используемого в качестве ключа разрешения записи-чтения. В обоих видах ЗУ информация исчезает после отключения от источника питания (например, тока).

**По типу доступа ЗУ делятся на**:

* устройства с последовательным доступом (например, магнитные ленты).
* устройства с произвольным доступом (RAM) (например, оперативная память).
* устройства с прямым доступом (например, жесткие магнитные диски).
* устройства с ассоциативным доступом (специальные устройства, для повышения производительности БД)

**Классификация запоминающих устройств по геометрическому исполнению:**

* дисковые (магнитные диски, оптические, магнитооптические); • ленточные (магнитные ленты, перфоленты);
* барабанные (магнитные барабаны);
* карточные (магнитные карты, перфокарты, флэш-карты, и др.)
* печатные платы (карты DRAM).

**Классификация запоминающих устройств по физическому принципу:**

* перфорационные (перфокарта; перфолента);
* с магнитной записью (ферритовые сердечники, магнитные диски, магнитные ленты, магнитные карты);
* оптические (CD, DVD, HD-DVD, Blu-ray Disc);
* использующие эффекты в полупроводниках (флэш-память) и другие.

Современная система памяти образует иерархию от быстрых типов памяти маленького размера до медленных типов памяти большого размера.

На вершине иерархии находятся **регистры процессора**. Доступ к ним занимает 0 тактов, но их всего несколько штук.

Далее идёт несколько килобайт **кэш-памяти** первого уровня, доступ к которой занимает примерно 4 такта. Потом идёт пара сотен килобайт более медленной кэш-памяти второго уровня. Потом несколько мегабайт кэш-памяти третьего уровня. Она гораздо медленней, но всё равно быстрее оперативной памяти.

Далее расположена относительно медленная **оперативная память**. Оперативную память можно рассматривать как кэш для локального диска (локальные диски тоже можно разделить на отдельные категории).

**Локальный диск** сам может рассматриваться как кэш для данных, расположенных на удалённых серверах. Резервные копии данных можно хранить на **магнитных лентах.**

В ПЗУ находятся:

* программа управления работой процессора;
* программа запуска и останова компьютера;
* программы тестирования устройств, проверяющие при каждом включении компьютера правильность работы его блоков;
* программы управления дисплеем, клавиатурой, принтером, внешней памятью;
* информация о том, где на диске находится операционная система.

**Оперативная память** (также оперативное запоминающее устройство, ОЗУ) - предназначена для временного хранения данных и команд, необходимых процессору для выполнения им операций. Оперативная память передаёт процессору данные непосредственно, либо через кэш-память. Каждая ячейка оперативной памяти имеет свой индивидуальный адрес. Устанавливается в виде модулей объемом до 64 ГБ. Наиболее распространенный тип модулей — DIMM, тип памяти — DDR.

18. Виртуальная память. Структуризация адресного пространства виртуальной памяти. Задачи управления виртуальной памятью.

**Виртуальная память** - Метод управления памятью процессора, предназначенный для выполнения программ, которым выделяется адресное пространство, превышающее доступный физический объем памяти компьютера.

**Свопинг (подкачка)** - Механизм обмена (вытеснения и загрузки) содержимым блоков оперативной физической памяти компьютера с устройством хранения данных с целью расширения адресуемого объема оперативной памяти компьютера. Механизм является аппаратно-программным.

**Страничная память** - Реализации виртуальной памяти, при которой физическая память и адресное пространство разбивается на блоки (страницы), а также осуществляется страничный свопинг. Размеры страниц для X86-64: 4K, 2MB, 1GB.

**Диспетчер памяти** - Memory Management Unit (MMU) - аппаратное (программируемое) устройство, входящее в состав процессора и предназначенное для трансляции виртуальных адресов оперативной памяти в реальные.

**Буфер быстрого преобразования адреса** - Translation Lookaside Buffer (TLB) - ассоциативная память (параллельный поиск), для архитектуры x86\_64 имеет объем 64 строки.

**Уровни привилегий и кольца защиты**

В архитектуре x86 четыре кольца защиты (самый привилегированный 0). В ARM аналогичный механизм называется уровнями исключений, здесь нулевой **уровень EL0:** область пользователя. **EL1:** ядро («супервизор» в терминологии ARM). **EL2:** гипервизоры, например Xen. **Гипервизор для ОС** — то же самое, что ОС для пользовательской среды. Например, Xen позволяет одновременно запускать несколько операционных систем, таких как Linux или Windows, в одной системе и изолирует операционные системы друг от друга для обеспечения безопасности и простоты отладки, точно так же, как Linux делает это для пользовательских программ. Гипервизоры являются ключевой частью современной облачной инфраструктуры: они позволяют нескольким серверам работать на одном оборудовании, сохраняя использование оборудования всегда близким к 100% и экономя много денег. **EL3**: еще один уровень.

**Кольца защиты с отрицательными номерами**

Условное обозначение более привилегированных уровней на архитектуре X86-64 при использовании тех или иных форм виртуализации. Не имеют физического представления. В архитектуре ARM в них нет необходимости, так как там используется более естественная нумерация уровней исключений (в порядке возрастания).

**Текущий уровень привилегий** Current Privilege Level (CPL) - уровень привилегий, на котором в данный момент исполняется задача. Значение CPL хранится в поле RPL селектора сегмента кода, который помещен в регистр CS. Обычно это значение соответствует уровню привилегий дескриптора исполняемого сегмента кода. Уровень привилегий меняется, когда управление передается сегменту кода с другим значением DPL (за исключением подчиняемых сегментов кода).

**Уровень привилегий дескриптора** Descriptor Privilege Level (DPL) - наименее привилегированный уровень, на котором задача может получить доступ к сегменту или шлюзу, связанному с этим дескриптором. Уровень DPL определяется битами 46 и 45 дескриптора.

**Запрашиваемый уровень привилегий** Requested Privilege Level (RPL) используется для временного понижения своего уровня привилегий при обращении к памяти. RPL заносится в младшие биты селектора.

**Дескриптор** - это 8-байтная единица описательной информации, распознаваемая устройством управления памятью в защищенном режиме, хранящаяся в дескрипторной таблице. Дескриптор сегмента содержит базовый адрес описываемого сегмента, предел сегмента и права доступа к сегменту. Дескрипторы являются основой защиты и мультизадачности.

**Селектор** - это 16-битный идентификатор сегмента, помещаемый в сегментный регистр. Он содержит индекс дескриптора в дескрипторной таблице, бит определяющий, к какой дескрипторной таблице производится обращение (LDT или GDT), а также запрашиваемые права доступа к сегменту.

**Сегментный регистр** – это 80-битный регистр, в котором доступны только младшие 16 бит, которые и содержат селектор. Остальные 64 бита называются Shadow register или «дескрипторным кэшем». Они и содержат базовый адрес, права доступа и лимит. Поэтому, как только мы загружаем в селектор соответствующее значение, процессор сразу же по полю индекс читает базу, лимит и уровень привилегий из дескриптора и заносит их в дескрипторный кэш сегментного регистра.

**Страничная память** — способ организации виртуальной памяти, при котором виртуальные адреса отображаются на физические постранично.

Виртуальная память делится на страницы. Размер размера страницы задается процессором и обычно на x86-64 составляет 4 KiB. Это означает, что управление памятью в ядре выполняется с точностью до страницы. Когда вам понадобится новая память, ядро предоставит вам одну или несколько страниц. При освобождении памяти вы вернёте одну или несколько страниц

Ставится задача транслировать 48-битный виртуальный адрес в физический. Она решается аппаратным обеспечением — блоком управления памятью (memory management unit, MMU). Этот блок является частью процессора. Чтобы транслировать адреса, он использует структуры данных в оперативной памяти, называемые таблицами страниц. Вместо двухуровневой системы таблиц страниц, используемой системами с 32-битной архитектурой x86, системы, работающие в длинном режиме, используют четыре уровня таблицы страниц. Возможные размеры страниц:

• 4 КБ (212 байт) — наиболее часто используется (как и в x86)

• 2 МБ (221 байт)

• 1 ГБ (230 байт)

Таблицы страниц хранятся в оперативной памяти. Если при каждом обращении по виртуальному адресу выполнять полностью трансляцию адресов, это будет работать очень медленно. Поэтому в процессоре реализуется специальный кеш под названием **«буфер ассоциативной трансляции»**

В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в память. С появлением мультипрограммирования перед ОС были поставлены новые задачи, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами

Функциями ОС по управлению памятью в мультипрограммной системе являются:

* отслеживание свободной и занятой памяти;
* выделение памяти процессам и освобождение памяти по завершении процессов;
* вытеснение кодов и данных процессов из оперативной памяти на диск (полное или частичное), когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
* настройка адресов программы на конкретную область физической памяти.

**Виртуальная память решает следующие задачи**:

• размещает данные в запоминающих устройствах разного типа, например, часть программы в оперативной памяти, а часть на диске;

• перемещает по мере необходимости данные между запоминающими устройствами разного типа, например, подгружает нужную часть программы с диска в оперативную память;

• преобразует виртуальные адреса в физические. Все эти действия выполняются автоматически, без участия программиста, то есть механизм виртуальной памяти является прозрачным по отношению к пользователю.

19. Подкачка. Алгоритмы замещения страниц. Куча (heap). Стек.

**Файл подкачки или виртуальная память** — это способ системы виртуальной памяти увеличить оперативную память, когда ее не хватает для совершения операций. Система автоматически задействует файл подкачки, когда приложениям не хватит системной памяти ОЗУ. Хотя система сама регулирует объем файла подкачки иногда может понадобиться вручную увеличить виртуальную память.

При возникновении **ошибки отсутствия страницы** операционная система должна выбрать выселяемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначенная для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть переписана на диске, чтобы привести дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности и перезапись не требуется. Тогда считываемая страница просто пишется поверх выселяемой

**Использование битов состояния в алгоритмах исключения страниц** Чтобы позволить операционной системе осуществить сбор полезной статистики востребованности страниц, большинство компьютеров, использующих виртуальную память, имеют два бита состояния, **R и M**, связанных с каждой страницей. **Бит R** устанавливается при каждом обращении к странице (при чтении или записи**). Бит M** устанавливается, когда в страницу ведется запись (то есть когда она модифицируется). Эти биты присутствуют в каждой записи таблицы страниц. Важно усвоить, что эти биты должны обновляться при каждом обращении к памяти, поэтому необходимо, чтобы их значения устанавливались аппаратной частью. После установки бита в 1 он сохраняет это значение до тех пор, пока не будет сброшен операционной системой. Биты R и M могут использоваться для создания следующего простого алгоритма замещения страниц. При запуске процесса оба страничных бита для всех его страниц устанавливаются операционной системой в 0. Время от времени (например, при каждом прерывании по таймеру) бит R сбрасывается, чтобы отличить те страницы, к которым в последнее время не было обращений, от тех, к которым такие обращения были. При возникновении ошибки отсутствия страницы операционная система просматривает все страницы и на основе текущих значений принадлежащих им битов R и M делит их на четыре категории:

1. Класс 0: в последнее время не было ни обращений, ни модификаций.

2. Класс 1: обращений в последнее время не было, но страница модифицирована.

3. Класс 2: в последнее время были обращения, но модификаций не было.

4. Класс 3: в последнее время были и обращения, и модификации.

**Алгоритм исключения давно использовавшейся страницы** Алгоритм исключения давно использовавшейся страницы (Not Recently Used (NRU)) удаляет произвольную страницу, относящуюся к самому низкому непустому классу. В этот алгоритм заложена идея, суть которой в том, что **лучше удалить модифицированную страницу, к которой не было обращений по крайней мере за последний такт системных часов (обычно это время составляет около 20 мс), чем удалить интенсивно используемую страницу.** Главная привлекательность алгоритма NRU в том, что его нетрудно понять, сравнительно просто реализовать и добиться от него производительности, которая, конечно, не оптимальна, но может быть вполне приемлема.

**Алгоритм «первой пришла, первой и ушла»** Другим низкозатратным алгоритмом замещения страниц является алгоритм FIFO (First In, First Out — «первым пришел, первым ушел»). Операционная система ведет список всех страниц, находящихся на данный момент в памяти, причем совсем недавно поступившие находятся в хвосте, поступившие раньше всех — в голове списка. При возникновении ошибки отсутствия страницы удаляется страница, находящаяся в голове списка, а к его хвосту добавляется новая страница. Принцип FIFO в чистом виде используется довольно редко.

**Алгоритм «второй шанс»** Простой модификацией алгоритма FIFO, исключающей проблему удаления часто запрашиваемой страницы, может стать проверка бита R самой старой страницы. Если его значение равно нулю, значит, страница не только старая, но и невостребованная, поэтому она тут же удаляется. Если бит R имеет значение 1, он сбрасывается, а страница помещается в конец списка страниц и время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается. Страницы с A по H содержатся в связанном списке отсортированными по времени их поступления в память. Предположим, что ошибка отсутствия страницы возникла на отметке времени 20. Самой старой является страница A, время поступления которой соответствует началу процесса и равно 0. Если бит R для страницы A сброшен, страница либо удаляется из памяти с записью на диск (если она измененная), либо просто удаляется (если она неизмененная). Но если бит R установлен, то A помещается в конец списка и ее «время загрузки» переключается на текущее (20). Алгоритм «второй шанс» занимается поиском ранее загруженной в память страницы, к которой за только что прошедший интервал времени таймера не было обращений. **Если обращения были ко всем страницам, то алгоритм «второй шанс» превращается в простой алгоритм FIFO**.

**Алгоритм «часы»** При всей своей логичности алгоритм «второй шанс» слишком неэффективен, поскольку он постоянно перемещает страницы в своем списке. Стрелка указывает на самую старую страницу. При возникновении ошибки отсутствия страницы проверяется та страница, на которую указывает стрелка. Если ее бит R имеет значение 0, страница выселяется, на ее место в «циферблате» вставляется новая страница и стрелка передвигается вперед на одну позицию. Если значение бита R равно 1, то он сбрасывается и стрелка перемещается на следующую страницу. Этот процесс повторяется до тех пор, пока не будет найдена страница с R = 0. Неудивительно, что этот алгоритм называется «часы».

**Алгоритм замещения наименее востребованной страницы** В основе неплохого приближения к оптимальному алгоритму лежит наблюдение, что страницы, интенсивно используемые несколькими последними командами, будут, скорее всего, снова востребованы следующими несколькими командами. И наоборот, долгое время не востребованные страницы наверняка еще долго так и останутся невостребованными. Эта мысль наталкивает на вполне реализуемый алгоритм: при возникновении ошибки отсутствия страницы нужно избавиться от той страницы, которая длительное время не была востребована. Эта стратегия называется замещением наименее востребованной страницы (Least Recently Used (LRU)). Теоретически реализовать алгоритм LRU вполне возможно, но его практическая реализация дается нелегко. Для поиска страницы в списке, ее удаления из него и последующего перемещения этой страницы вперед потребуется довольно много времени, даже если это будет возложено на аппаратное обеспечение (если предположить, что такое оборудование можно создать). Так как вряд ли будет создано оборудование с такой функциональностью, нам понадобится решение, которое может быть реализовано в программном обеспечении.

**Алгоритм нечастого востребования** Одно из возможных решений называется алгоритмом нечастого востребования (Not Frequently Used (NFU)). Для его реализации потребуется программный счетчик с начальным нулевым значением, связанный с каждой страницей. При каждом прерывании от таймера операционная система сканирует все находящиеся в памяти страницы. Для каждой страницы к счетчику добавляется значение бита R, равное 0 или 1. Счетчики позволяют приблизительно отследить частоту обращений к каждой странице. При возникновении ошибки отсутствия страницы для замещения выбирается та страница, чей счетчик имеет наименьшее значение. Основная проблема при использовании алгоритма NFU заключается в том, что он никогда ничего не забывает. К примеру, при многопроходной компиляции те страницы, которые интенсивно использовались при первом проходе, могут сохранять высокие значения счетчиков и при последующих проходах. Фактически если на первый проход затрачивается больше времени, чем на все остальные проходы, то страницы, содержащие код для последующих проходов, могут всегда иметь более низкие показатели счетчиков, чем страницы, использовавшиеся при первом проходе. Вследствие этого операционная система будет замещать нужные страницы вместо тех, надобность в которых уже отпала.

**Алгоритм старения** К счастью, небольшая модификация алгоритма NFU позволяет довольно близко подойти к имитации алгоритма LRU. Модификация состоит из двух частей. Во-первых, перед добавлением к счетчикам значения бита R их значение сдвигается на один разряд вправо. Во-вторых, значение бита R добавляется к самому левому, а не к самому правому биту. Предположим, что после первого прерывания от таймера бит R для страниц от 0-й до 5-й имеет значения соответственно 1, 0, 1, 0, 1 и 1 (для страницы 0 оно равно 1, для страницы 1 — 0, для страницы 2 — 1 и т. д.). Иными словами, между прерываниями от таймера, соответствующими тактам 0 и 1, было обращение к страницам 0, 2, 4 и 5, в результате которого их биты R были установлены в 1, а у остальных страниц их значение осталось равным 0. После того как были смещены значения шести соответствующих счетчиков и слева вставлено значение бита R, они приобрели значения, показанные на рис. 3.2.19, а. В четырех оставшихся столбцах показаны состояния шести счетчиков после следующих четырех прерываний от таймера. Отличия алгоритма старения от полноценного LRU: неразличимость во времени обращений к страницам внутри интервала между двумя последовательными прерывания таймера и ограниченный горизонт (учитываются только несколько последних прерываний таймера)

**Алгоритм «рабочий набор»** При использовании замещения страниц в простейшей форме процессы начинают свою работу, не имея в памяти вообще никаких страниц. Как только центральный процессор попытается извлечь первую команду, он получает ошибку отсутствия страницы, заставляющую операционную систему ввести в память страницу, содержащую первую команду. Обычно вскоре за этим следуют ошибки отсутствия страниц с глобальными переменными и стеком. Через некоторое время процесс располагает большинством необходимых ему страниц и приступает к работе, сталкиваясь с ошибками отсутствия страниц относительно редко. Эта стратегия называется **замещением страниц по требованию (demand paging),** поскольку страницы загружаются только по мере надобности, а не заранее. Но, как правило, процессы в конкретный момент работают только с небольшой частью памяти. Если обеспечить ее наличие в памяти, количество страничных ошибок резко уменьшится. **Набор страниц, который процесс использует в данный момент, известен как рабочий набор**. Если в памяти находится весь рабочий набор, процесс будет работать, не вызывая многочисленных ошибок отсутствия страниц, пока не перейдет к другой фазе выполнения. Многие системы замещения страниц пытаются отслеживать рабочий набор каждого процесса и обеспечивать его присутствие в памяти, перед тем как позволить процессу возобновить работу. Такой подход называется моделью рабочего набора. Он был разработан для существенного сокращения количества ошибок отсутствия страниц. Загрузка страниц до того, как процессу будет позволено возобновить работу, называется также **опережающей подкачкой страниц (prepaging).** Следует заметить, что со временем рабочий набор изменяется. Но обычно он изменяется медленно, поэтому появляется возможность выстроить разумные предположения о том, какие страницы понадобятся при возобновлении работы программы, основываясь на том, каков был ее рабочий набор при последней приостановке ее работы. **Опережающая подкачка страниц как раз и заключается в загрузке этих страниц перед возобновлением процесса.** Для реализации модели рабочего набора необходимо, чтобы операционная система отслеживала, какие именно страницы входят в рабочий набор. При наличии такой информации тут же напрашивается возможный алгоритм замещения страниц: при возникновении ошибки отсутствия страницы нужно выселить ту страницу, которая не относится к рабочему набору.

Алгоритм рабочего набора обеспечивает приемлемую производительность, но его реализация обходится слишком дорого. Алгоритм WSClock является вариантом, который не только обеспечивает неплохую производительность, но и может быть эффективно реализован. В итоге наиболее приемлемыми алгоритмами являются алгоритм старения и алгоритм WSClock. Они основаны на LRU и рабочем наборе соответственно. Оба обеспечивают неплохую производительность страничной организации памяти и могут быть эффективно реализованы. Существует также ряд других хороших алгоритмов, но эти два, наверное, имеют наибольшее практическое значение.

**Куча (heap)** — это сегмент памяти, для которого не устанавливается постоянный размер перед компиляцией и который может динамически управляться программистом. То есть, это «свободный пул» памяти, который можно использовать при запуске приложения. Размер кучи приложения определяется физическими ограничениями оперативной памяти (оперативной памяти) и обычно намного больше размера стека. Память из кучи выделяется с помощью функции malloc(). malloc возвращает указатель void, который затем нужно привести к правильному типу. После использования блока памяти его нужно освободить помощью функции free().

Каждый процесс имеет кучу по умолчанию, предоставляемую системой. Приложения, которые часто выделяют ресурсы из кучи, могут повысить производительность с помощью частных куч**. Частная куча** — это блок одной или нескольких страниц в адресном пространстве вызывающего процесса. После создания частной кучи процесс использует такие функции, как HeapAlloc и HeapFree, для управления памятью в этой куче. Функции кучи также можно использовать для управления памятью в куче процесса по умолчанию с помощью дескриптора, возвращаемого функцией GetProcessHeap. Для этой цели новые приложения должны использовать функции кучи вместо глобальных и локальных функций. Нет различий между памятью, выделенной из частной кучи, и памятью, выделенной с помощью других функций выделения памяти. Функция HeapCreate создает частный объект кучи, из которого вызывающий процесс может выделять блоки памяти с помощью функции HeapAlloc. После фиксации страницы не удаляются до завершения процесса или до тех пор, пока куча не будет уничтожена путем вызова функции HeapDestroy. Функция HeapAlloc выделяет указанное количество байтов из частной кучи и возвращает указатель на выделенный блок. Этот указатель можно использовать в функциях HeapFree, HeapReAlloc, HeapSize и HeapValidate.

**Стек** — это сегмент памяти, в котором данные, такие как локальные переменные и вызовы функций, добавляются и/или удаляются по принципу «последним пришел — первым вышел» (LIFO). Вообще говоря, стек — это структура данных, которая хранит значения данных в памяти последовательно. Однако, в отличие от массива, вы получаете доступ (чтение или запись) к данным только на «верхней части» стека. Чтение из стека называется «извлечение» (pop), а запись в стек — «вталкивание» (push).

20. Типы устройств ввода/вывода. Обработка внешних прерываний. Синхронный и асинхронный ввод/вывод. (система прерываний в 4 вопросе)

**Устройства ввода/вывода** — компонент типовой архитектуры ЭВМ, предоставляющий компьютеру возможность взаимодействия с внешним миром и, в частности, с пользователями. По направлению передачи информации подразделяются на: устройства ввода, устройства вывода, устройства ввода и вывода. Устройства ввода/вывода (клавиатура, мышь, принтер и т.д.) обычно состоят из механического и электронного компонента, где электронный компонент называется контроллером устройства.

**Регистры устройства** У каждого контроллера для связи с центральным процессором имеется несколько регистров. Путем записи в эти регистры операционная система может давать устройству команды на предоставление данных, принятие данных, включение, выключение или выполнение каких-нибудь других действий. Считывая данные из этих регистров, операционная система может узнать о текущем состоянии устройства, о том, готово ли оно принять новую команду, и т. д. Для доступа к этим регистрам используются два подхода. 1. Каждый регистр получает уникальный номер порта в пространстве ввода-вывода. 2. Регистры отображаются на пространство памяти. Возможен гибридный вариант. Второй подход более удобен для программирования, но при наличии отдельной шины памяти нужно принимать дополнительные меры.

**Драйверы устройств** — это программные модули, которые можно подключить к операционной системе для работы с определенным устройством. Операционная система получает помощь от драйверов устройств для работы со всеми устройствами ввода-вывода.

**Система ввода-вывода в Linux** довольно проста и не отличается от присущих другим UNIX-системам. Как правило, все устройства ввода-вывода выглядят как файлы и доступ к ним осуществляется с помощью тех же системных вызовов read и write, которые используются для доступа ко всем обычным файлам. В некоторых случаях должны быть заданы параметры устройства — это делается при помощи специального системного вызова. В операционной системе Linux все устройства интегрируются в файловую систему в виде так называемых специальных файлов (special files). Каждому устройству ввода-вывода назначается путь (обычно в каталоге /dev). Например, диск может иметь путь /dev/hd1, у принтера может быть путь /dev/lp, а у сети — /dev/net.

**Специальные файлы** подразделяются на две категории: блочные и символьные. Блочный специальный файл (block special file) состоит из последовательности пронумерованных блоков. Основное свойство блочного специального файла заключается в том, что к каждому его блоку можно адресоваться и получить доступ отдельно. Блочные специальные файлы обычно используются для дисков. Символьные специальные файлы (character special files) обычно используются для устройств ввода или вывода символьного потока. Символьные специальные файлы используются такими устройствами, как клавиатуры, принтеры, сети, мыши, плоттеры и т. д. С каждым специальным файлом связан драйвер устройства, осуществляющий работу с соответствующим устройством. Каждый драйвер разделен на две части, причем обе они являются частью ядра Linux и работают в режиме ядра. Верхняя часть драйвера работает в контексте вызывающей стороны и служит интерфейсом к остальной системе Linux. Нижняя часть работает в контексте ядра и взаимодействует с устройством. Драйверам разрешается делать вызовы процедур ядра для выделения памяти, управления таймером, управления DMA и т. д.

**Система ввода-вывода в Windows** состоит из служб Plug-and-Play, диспетчера электропитания, менеджера ввода-вывода, а также модели драйвера устройств. Plug-and-Play обнаруживает изменения в конфигурации аппаратного обеспечения, создает (или уничтожает) стеки устройств (для каждого устройства), а также загружает и выгружает драйверы устройств. Диспетчер электропитания настраивает состояние электропитания устройств ввода-вывода, чтобы уменьшить потребление энергии системой, когда устройства не используются. Диспетчер ввода-вывода предоставляет поддержку манипулирования объектами ядра для ввода-вывода, а также операций типа IoCallDrivers и IoCompleteRequest. Однако большая часть работы по поддержке ввода-вывода в Windows реализована в самих драйверах устройств. Чтобы гарантировать, что драйверы устройств хорошо работают с Windows, компания Microsoft описала модель WDM (Windows Driver Model), которой должны соответствовать драйверы устройств. Существует набор разработчика (Windows Driver Kit), который содержит документацию и примеры, помогающие создавать драйверы, соответствующие WDM. Большинство драйверов Windows начинается с копирования подходящего образцового драйвера из WDK и его модификации создателем нового драйвера.

**Устройства в Windows** представлены объектами устройств. Объекты устройств используются для представления аппаратных средств (таких, как шины), а также как программные абстракции (наподобие файловых систем, сетевых протоколов и расширений ядра вроде антивирусных драйверов-фильтров). Все они формируют то, что Windows называет стеком устройств.

**Обслуживание прерывания в Windows**

Обслуживание прерывания состоит из двух, а иногда и трех этапов. 1. Быстрое сохранение непостоянной информации (например, регистр содержимого) в процедуре прерывания, которая выполняется в IRQL = DIRQL. 2. Обработка сохраненных переменных данных в отложенном вызове процедуры (DPC), который выполняется в IRQL = DISPATCH\_LEVEL. 3. Выполнение дополнительных работ в IRQL = PASSIVE\_LEVEL, если это необходимо.

**Асинхронный ввод-вывод используется там, где можно оптимизировать производительность приложения**. При асинхронном вводе-выводе приложение инициирует операцию ввода-вывода, а затем может продолжить свою работу (во время выполнения этого запроса). При синхронном вводе-выводе приложение блокируется до завершения выполнения операции ввода-вывода. С точки зрения вызывающего потока асинхронный ввод-вывод более эффективен, поскольку позволяет продолжать выполнение, в то время как операция ввода-вывода ставится диспетчером ввода-вывода в очередь и впоследствии выполняется. Однако приложение, использующее асинхронный ввод-вывод, требует механизма определения завершенности этой операции.

**Буферизация** — метод организации обмена, в частности, ввода и вывода данных в компьютерах и других вычислительных устройствах, который подразумевает использование буфера для временного хранения данных. При вводе данных одни устройства или процессы производят запись данных в буфер, а другие — чтение из него, при выводе — наоборот. Процесс, выполнивший запись в буфер, может немедленно продолжать работу, не ожидая, пока данные будут обработаны другим процессом, которому они предназначены. В свою очередь, процесс, обработавший некоторую порцию данных, может немедленно прочитать из буфера следующую порцию. Таким образом, буферизация позволяет процессам, производящим ввод, вывод и обработку данных, выполняться параллельно, не ожидая, пока другой процесс выполнит свою часть работы. Поэтому буферизация данных широко применяется в многозадачных ОС.

**Дисковое планирование**

Способ оптимизации обращения к данным, находящимся на разных участках жесткого диска (обычно магнитного). Известны следующие алгоритмы: случайное планирование, «Первым вошел — первым вышел» (FIFO), с использованием приоритета (PRI), «Последним вошел — первым вышел» (LIFO), стратегия выбора наименьшего времени обслуживания (SSTF), стратегия SCAN, стратегия C-SCAN, стратегия N-step-SCAN, стратегия FSCAN.

21. Файловые системы. Файлы и директории. Управление внешней памятью.

**Дескриптор файла** - это управляющий блок, содержащий информацию, необходимую ОС для различных операций с файлом. Конкретный вид дескриптора файла определяется используемой ОС. Другие названия дескриптора - блок управления файлом, элемент каталога файлов. Типичный дескриптор файла содержит следующие данные: символическое имя файла; тип файла; атрибуты файла; данные о размещении файла во внешней памяти и др.

**Файловая система (file system)** — порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах, а также в другом электронном оборудовании

Список файловых систем, которые поддерживаются ядром, находится в файле /proc/filesystems. **Виды файловых систем, предлагаемых при установке ОС на базе Linux:** Ext, Ext2, Ext3, Ext4, JFS, XFS, Btrfs, Swap.

**Типы файловых систем Windows включают**: таблица размещения файлов (FAT), FAT32 и расширенная таблица размещения файлов (exFAT); файловая система NT (NTFS); устойчивая файловая система (ReFS).

**VFS** определяет набор основных абстракций файловой системы и разрешенные с этими абстракциями операции. Системные вызовы обращаются к структурам данных VFS, определяют тип файловой системы (к которой принадлежит нужный файл) и при помощи хранящихся в структурах данных VFS указателей на функции запускают соответствующую операцию в указанной файловой системе.

**Файл** - поименованная совокупность данных, обычно размещаемая на внешних запоминающих устройствах.

Каждому файлу на диске соответствует один и только один **индексный дескриптор файла**, который идентифицируется своим порядковым номером - индексом файла (inode). Это означает, что число файлов, которые могут быть созданы в файловой системе, ограничено числом индексных дескрипторов, которое либо явно задается при создании файловой системы, либо вычисляется исходя из физического объема дискового раздела.

**Кеширование** в операционной системе служит для минимизации количества операций передачи данных. Для достижения данной цели в Linux-системах между дисковыми драйверами и файловой системой имеется кэш. Кэш представляет собой таблицу в ядре, в которой хранятся тысячи недавно использованных блоков. Когда файловой системе требуется блок диска (например, блок i-узла, каталога или данных), то сначала проверяется кэш. Если нужный блок есть в кэше, он берется оттуда, при этом обращения к диску удается избежать (что значительно улучшает производительность системы). Если же блока в кэше страниц нет, то он считывается с диска в кэш, а оттуда копируется туда, куда нужно. Поскольку в кэше страниц есть место только для фиксированного количества блоков, то используется один из описанных ранее алгоритмов замещения страниц.

22. Принципы организации и структура ОС Windows. Обзор версий Windows. Методы инсталляции ОС Windows.

Архитектура операционной системы Windows включает **ядро операционной системы, системные службы и приложения**

На самом низком уровне операционной системы **ядро операционной системы** состоит из самого ядра Windows и драйверов устройств низкого уровня. Ядро отвечает за прием запросов операционной системы от системных служб. Затем ядро преобразует эти запросы в инструкции для аппаратного обеспечения компьютера, включая центральный процессор (ЦП), память и аппаратные устройства. При запуске операционной системы сначала инициализируются ядро и связанные с ним низкоуровневые драйверы устройств, а затем службы операционной системы.

**Службы операционной системы** являются частью операционной системы, а не компонентами, которые вы устанавливаете после развертывания операционной системы. Кроме того, службы операционной системы функционируют без каких-либо действий со стороны пользователя. Фактически они начинаются до того, как пользователь войдет в систему. И службы операционной системы, и драйверы устройств являются программным обеспечением. Однако разница между ними заключается в том, что драйверы устройств напрямую взаимодействуют с аппаратными устройствами или компонентами. Как правило, системная служба взаимодействует с другими программными компонентами операционной системы.

На верхнем уровне операционной системы **приложения** работают путем взаимодействия с пользователем компьютера, а на нижнем уровне — путем интеграции со службами операционной системы. Вы устанавливаете приложения после установки операционной системы, и для их использования вам необходимо запускать приложения вручную.

**Отличия службы от приложения**. Служба – процесс, работающий, даже когда никто не зарегистрирован в системе. Служба использует несколько потоков: один поток принимает команды от операционной системы (Start, Stop, Pause, Resume и т.п.), второй поток является основным рабочим потоком (обычно он создает еще множество потоков для выполнения отдельных задач). Завершение этого потока завершает работу всей службы. Управляется диспетчером управления службами (Service Control Manager, SCM). Физически может находиться в .EXE или .DLL-файле, в одном файле может находиться множество служб. Если служба находится в .DLL-файле, ее запускает svchost.exe.

**Реестр** — это база данных, в которой Windows хранит параметры конфигурации пользователя и компьютера. Всякий раз, когда вы вносите изменения в конфигурацию Windows, это изменение фиксируется в реестре. Реестр Windows организован иерархически. На верхнем уровне имеется пять кустов реестра, которые представляют собой отдельный набор связанных параметров, структурированных как серия ключей, подразделов и значений.

Операционная система Windows имеет **модульную структуру,** что обеспечивает ее работу на самых разных аппаратных платформах, с самыми разными приложениями. Кроме того, этим достигается относительно эффективная возможность модернизации ОС (либо путем замены каких-либо отдельных структурных блоков, либо путем включения новых блоков)

Принцип модульной структуры состоит в том, что за реализацию каждой из функций ОС отвечает отдельный модуль. Все остальные части системы и приложения обращаются к этой функции через стандартный интерфейс. А доступ к основным системным данным можно получить только через определенные функции.

Самый удивительный аспект состоит в том, что ядро Windows практически не меняется в зависимости от всех этих архитектур и SKU. Ядро динамически масштабируется в зависимости от архитектуры и процессора, на котором оно работает, так, чтобы пользоваться всеми возможностями оборудования. Конечно, в ядре присутствует определённое количество кода, связанного с конкретной архитектурой, однако его там минимальное количество, что позволяет Windows запускаться на разнообразных архитектурах.

**Клиентские версии Windows**

Различные выпуски Windows удовлетворяют потребности потребителей, от частных лиц до крупных предприятий.

* **Home** - Индивидуальное домашнее использование.
* **Pro** - Малый и средний бизнес, продвинутые пользователи.
* **Pro для рабочих станций** - Пользователи с повышенными требованиями к производительности и хранилищу.
* **Enterprise** - Крупные коммерческие организации.
* **Enterprise LTSC** - Крупные корпоративные организации с ограничительными требованиями к изменениям.
* **Pro Education** - Сравнимо с Pro для сотрудников школы, администраторов, учителей и учащихся.
* **Education** - Сравнимо с Enterprise для школьного персонала, администраторов, учителей и учащихся.
* **IoT Core/ Enterprise** - Устройства стационарного назначения и бытовые устройства.

**Серверные версии Windows Server**

Последняя LTSC-версия Windows Server сегодня — это **Windows Server 2022**. Также поддерживается и Windows Server 2019. Windows Server 2022 обеспечивает расширенную многоуровневую безопасность, гибридные возможности с Azure и гибкую платформу приложений. Он имеет встроенные возможности защищенного ядра, помогающие защитить оборудование, встроенное ПО и возможности ОС Windows Server от расширенных угроз безопасности. Существует три редакции Windows Server 2022: **Standard, Datacenter, Datacenter: Azure Edition.** В портфолио Microsoft есть еще одна операционная система, основанная на Windows Server, но имеющая другое имя. Это **Azure Stack HCI**. Она имеет собственную сертификацию оборудования, лицензируется по подписке и может быть установлен только на «голое железо», но не на виртуальную машину.

Существует несколько разных вариантов развертывания сервера Windows и настольных операционных систем бизнес-класса. Это могут быть ручная установка, полностью автономные (не обслуживаемые) установки, автономные готовые или дополнительно настроенные установки, а также развертывания заготовленных и, возможно, специально настроенных образов операционной системы.

**Устанавливать систему вручную** не очень сложно. Нужно воспользоваться установочным носителем и пройти все этапы установки, документируя по мере продвижения все настройки. Этот способ иногда необходим, когда у администратора нет образа, подходящего для определенной аппаратной платформы, или когда регулярно развертывается только небольшое количество систем, а время, необходимое на создание автоматических установок или установок с использованием образов, не ограничено и не является ключевым фактором для организации.

**Автономные установк**и могут быть полезными при развертывании большого количества настольных компьютеров и серверов, на которых установлено одинаковое оборудование. Файл автономной установки представляет собой текстовый XML-файл, содержащий ответы на все вопросы, задаваемые во время ручной установки.

Конфигурационные файлы автономной установки обычно называют **файлами ответов.** Опции в некоторых файлах ответов автономной установки могут включать принятие лицензионного соглашения конечным пользователем, ввод лицензионного ключа, вариант форматирования жесткого диска, определение раздела или размера тома для операционной системы и многое другое. Теперь это называется файлом необслуживаемой установки (unattended installation file). Чтобы выбрать файл ответов во время установки, нужно загрузить среду предустановки Windows и выполнить команду setup.exe с параметром /unattend:filename. При отсутствии параметра /unattend программа установки будет искать файл с именем Autounattend.xml в корневых папках всех доступных дисков. Поэтому проще всего записать файл ответа на USB-устройство флэш-памяти, вставить его в компьютер, а затем загрузить этот компьютер с установочного диска Windows.

**Создание копий или образов систем** может быть полезно при развертывании нескольких идентичных настольных компьютеров и серверов. Вы создаете настольный компьютер или сервер, подготавливаете систему для клонирования или создания образа и копируете или захватываете образ системы с помощью средств сторонних разработчиков или средств развертывания от Microsoft, таких как WDS.

**WDS** — это серверная роль в Windows Server, которая спроектирована для оказания помощи организациям, использующим доменную службу Active Directory, при развертывании систем Windows. Система WDS обычно настраивается для обеспечения служб хранения и извлечения образов, необходимых для их развертывания, клиентских компонентов, таких как образы загрузки РХЕ, и управляющих компонентов, используемых для конфигурирования параметров WDS, в том числе добавление образов на сервер WDS и создание мультивещательных передач.

**WIM** — это формат образа диска на основе файлов, который впервые был представлен в Windows Vista. Файлы WIM представляют собой сжатые пакеты, содержащие несколько связанных файлов. Формат WIM-файла оптимизирован для максимального сжатия с использованием LZX или для быстрого сжатия с использованием XPRESS. Структура файла WIM содержит до шести типов ресурсов: заголовок, файловый ресурс, ресурс метаданных, таблицу поиска, данные XML и таблицу целостности. Для работы с WIM-файлами используют программы ImageX и DISM из состава WADK. DISM также входит в состав Windows.

23. ОС Windows: организация рабочей среды пользователя, работа с учетными записями пользователей и групп, работа с профилями пользователей.

Рабочая среда пользователя состоит из настроек рабочего стола, например, цвета экрана, настроек мыши, размера и расположения окон, из настроек процесса обмена информацией по сети и с устройством печати, переменных среды, параметров реестра и набора доступных приложений

Для управления средой пользователя предназначены следующие средства Windows:

**Сценарий входа в сеть** (сценарий регистрации) представляет собой командный файл, имеющий расширение .bat, или исполняемый файл с расширением .ехе, который выполняется при каждой регистрации пользователя в сети. Сценарий может содержать команды операционной системы, предназначенные, например, для создания соединения с сетью или для запуска приложения. Кроме того, с помощью сценария можно устанавливать значения переменных среды, указывающих пути поиска, каталоги для временных файлов и другую подобную информацию.

**Профили пользователей**. В профиле пользователя хранятся все настройки рабочей среды компьютера, на котором работает Windows 2000, определенные самим пользователем. Это могут быть, например, настройки экрана и соединения с сетью.

**Сервер сценариев Windows** (Windows Scripting Host, WSH). Сервер сценариев независим от языка и предназначен для работы на 32-разрядных платформах Windows. Он включает в себя как ядро сценариев Visual Basic Scripting Edition (VBScript), так и JScript. Сервер сценариев Windows предназначен для выполнения сценариев прямо на рабочем столе Windows или на консоли команд. При этом сценарии не надо встраивать в документ НТМL.

Большинство настроек рабочей среды пользователя можно настроить с помощью **групповых политик**.

**Локальные учетные записи** пользователей определяются локально на устройстве и могут назначаться только на этом устройстве. **Локальные учетные записи пользователей** — это субъекты безопасности, которые используются для защиты и управления доступом к ресурсам на устройстве для служб или пользователей.

**Учетные записи локальных пользователей по умолчанию** — это встроенные учетные записи, которые создаются автоматически при установке операционной системы. Учетные записи локальных пользователей по умолчанию не могут быть удалены или удалены и не предоставляют доступ к сетевым ресурсам. Учетные записи локальных пользователей по умолчанию используются для управления доступом к ресурсам локального устройства на основе прав и разрешений, назначенных учетной записи.

Учетные записи локальных пользователей по умолчанию и создаваемые локальные учетные записи пользователей находятся в папке Пользователи. Папка Пользователи находится в папке Локальные пользователи и группы в локальной консоли управления Майкрософт (MMC). **Управление компьютером** — это набор средств администрирования, которые можно использовать для управления локальным или удаленным устройством. Вы также можете управлять локальными пользователями с помощью NET.EXE USER и управлять локальными группами с помощью NET.EXE LOCALGROUP или с помощью различных командлетов PowerShell и других технологий сценариев.

**Учетная запись локального администратора по умолчанию** — это учетная запись пользователя для системного администрирования. У каждого компьютера есть учетная запись администратора (SID S-1-5-домен-500, отображаемое имя — Администратор). Учетная запись администратора — это первая учетная запись, которая создается при установке Windows. Учетная запись администратора имеет полный контроль над файлами, каталогами, службами и другими ресурсами на локальном устройстве. Учетная запись администратора может создавать других локальных пользователей, назначать права пользователей и назначать разрешения. Учетная запись администратора может в любое время управлять локальными ресурсами, изменив права и разрешения пользователя. Учетную запись администратора по умолчанию нельзя удалить или заблокировать, но ее можно переименовать или отключить.

**Гостевая учетная запись** позволяет случайным или разовым пользователям, у которых нет учетной записи на компьютере, временно войти на локальный сервер или клиентский компьютер с ограниченными правами пользователя. По умолчанию учетная запись гостя отключена и имеет пустой пароль. Так как учетная запись гостя может предоставлять анонимный доступ, она считается угрозой безопасности. По этой причине рекомендуется оставить учетную запись гостя отключенной, если только ее использование не требуется. По умолчанию учетная запись "Гость" является единственным членом группы SID S-1-5-32-546 "Гости" по умолчанию, которая позволяет пользователю войти на устройство.

**Система создает профиль пользователя** при первом входе пользователя на компьютер. При последующих входах система загружает профиль пользователя, а затем другие системные компоненты настраивают среду пользователя в соответствии с информацией в профиле. Типы профилей пользователей: локальные, перемещаемые, обязательные, временные. Профиль пользователя состоит из следующих элементов: куста реестра HKEY\_CURRENT\_USER и набора папок, хранящегося в файловой системе.

**Локальный профиль пользователя** создается при первом входе пользователя на компьютер. Профиль хранится на локальном жестком диске компьютера. Изменения, внесенные в профиль локального пользователя, относятся к пользователю и компьютеру, на котором внесены изменения.

**Перемещаемый профиль пользователя** — это копия локального профиля, которая копируется в общую папку сервера и хранится в ней. Этот профиль загружается на любой компьютер, на который пользователь входит в сеть. Изменения, внесенные в перемещаемый профиль пользователя, синхронизируются с серверной копией профиля при выходе пользователя из системы. Преимущество перемещаемых профилей пользователей заключается в том, что пользователям не нужно создавать профиль на каждом компьютере, который они используют в сети.

**Обязательный профиль пользователя** — это тип профиля, который администраторы могут использовать для указания параметров для пользователей. Только системные администраторы могут вносить изменения в обязательные профили пользователей. Изменения, внесенные пользователями в параметры рабочего стола, теряются при выходе пользователя из системы.

**Временный профиль** выдается каждый раз, когда ошибка препятствует загрузке профиля пользователя. Временные профили удаляются в конце каждого сеанса, а изменения, внесенные пользователем в параметры рабочего стола и файлы, теряются при выходе пользователя из системы. Временные профили доступны только на компьютерах под управлением Windows 2000 и более поздних версий.

24. Системный реестр ОС Windows, его назначение и использование.

**Реестр** — это база данных, в которой Windows хранит параметры конфигурации пользователя и компьютера. Всякий раз, когда вы вносите изменения в конфигурацию Windows, это изменение фиксируется в реестре

Реестр Windows организован иерархически. На верхнем уровне имеется пять кустов реестра, которые представляют собой отдельный набор связанных параметров, структурированных как серия ключей, подразделов и значений:

**HKEY\_CLASSES\_ROOT** Этот куст содержит информацию о ассоциации файлов и определяет, какое приложение открывается, когда пользователь дважды щелкает определенный тип файла в файловой системе. Например, он определяет, что приложением для файлов .xlsx является Microsoft Excel. Этот куст заполняется на основе настроек, связанных с компьютером и пользователем, которые хранятся в разделах HKEY\_LOCAL\_MACHINE\Software\Classes и HKEY\_CURRENT\_USER\Software\Classes. Обычно вы не будете вносить изменения в этот куст.

**HKEY\_CURRENT\_USER** Этот куст содержит информацию о конфигурации для текущего пользователя, вошедшего в систему. Такие элементы, как цветовая схема Windows пользователя и настройки шрифта, хранятся в соответствующих значениях под этим кустом. При ссылке на этот куст при редактировании реестра этот куст иногда называют HKCU. Этот куст представляет собой ярлык ключа, хранящегося в HKEY\_USERS.

**HKEY\_LOCAL\_MACHINE** Вероятно, это самый важный куст, в который вы, вероятно, будете вносить больше всего изменений. Этот куст, иногда называемый сокращенно HKLM, хранит все параметры конфигурации, связанные с компьютером.

**HKEY\_USERS** Этот куст содержит коллекцию всей информации о конфигурации для всех пользователей, выполнивших вход локально на компьютер, включая текущего пользователя, вошедшего в систему. Фактически, один из ключей под этим кустом — это ключ текущего пользователя, вошедшего в систему, который отображается как куст HKEY\_CURRENT\_USER. Важно знать, что вы, скорее всего, будете вносить прямые изменения в настройки пользователя только для текущего пользователя, вошедшего в систему.

**HKEY\_CURRENT\_CONFIG** Этот куст содержит информацию о текущем профиле оборудования, который локальный компьютер использовал во время запуска системы. Обычно вы не вносите изменения в этот куст.

Для поддержания структуры базы данных аналогичные настройки хранятся в папках и подпапках, известных как **ключи и подразделы**. Это упрощает ссылку на определенное значение реестра. Значения определяют поведение операционной системы и хранятся в разделах и подразделах. Существует много типов значений, в зависимости от типа данных, которые каждое из них хранит. Ниже перечислены наиболее распространенные типы значений реестра:

**REG\_BINARY**. Необработанные двоичные данные. Эти значения обычно отображаются в шестнадцатеричном формате. Информация об оборудовании часто хранится в значениях REG\_BINARY.

**REG\_DWORD**. 4-байтовые числа (32-битное целое число). Многие значения, связанные с драйверами устройств и службами, хранятся в значениях REG\_DWORD.

**REG\_SZ.** Текстовая строка фиксированной длины.

**REG\_EXPAND\_SZ.** Текстовая строка переменной длины.

**REG\_MULTI\_SZ**. Несколько строковых значений.

25. ОС Windows: планирование и назначение разрешений NTFS.

26. Средства автоматической настройки в ОС Windows.

**В Windows есть две оболочки командной строки: командная оболочка cmd и PowerShell**. Каждая оболочка — это программная программа, которая обеспечивает прямую связь между оператором и операционной системой или приложением, предоставляя среду для автоматизации ИТ-операций. Командная оболочка cmd была первой оболочкой, встроенной в Windows, для автоматизации повседневных задач, с пакетными (.bat или .cmd) файлами. С помощью Windows Script Host можно запускать более сложные скрипты в командной оболочке (команды cscript и vscript). PowerShell был разработан для расширения возможностей командной оболочки для выполнения команд PowerShell, называемых командлетами. Командлеты похожи на команды Windows, но предоставляют более расширяемый язык сценариев. Команды Windows и командлеты PowerShell можно запускать в PowerShell, но командная оболочка может выполнять только команды Windows, а не командлеты PowerShell.

**Инструментарий управления Windows (WMI)** — это инфраструктура для управления данными и операциями в операционных системах Windows. Это расширенная и адаптированная под Windows реализация стандарта WBEM, принятого многими компаниями, в основе которого лежит идея создания универсального интерфейса мониторинга и управления различными системами и компонентами распределённой информационной среды предприятия с использованием объектно-ориентированных идеологий и протоколов HTML и XML. В основе структуры данных в WBEM лежит Common Information Model (CIM), реализующая ООП-подход к представлению компонентов системы. CIM является расширяемой моделью, что позволяет программам, системам и драйверам добавлять в неё свои классы, объекты, методы и свойства. WMI, основанный на CIM, также является открытой унифицированной системой интерфейсов доступа к любым параметрам операционной системы, устройствам и приложениям.

27. Основные понятия системы UNIX. Пользователи системы, атрибуты пользователя. Файловая структура ОС.

Операционная система UNIX всегда была интерактивной системой, разработанной для одновременной поддержки множества процессов и множества пользователей. Она была разработана программистами и для программистов. От системы требовалось быть простой, элегантной и совместимой. В системе должно быть небольшое количество базовых элементов, которые можно комбинировать, чтобы приспособить их для конкретного приложения. **Другие характерные особенности**: использование простых текстовых файлов для настройки и управления системой; широкое применение утилит, запускаемых из командной строки; взаимодействие с пользователем посредством виртуального устройства — терминала; представление физических и виртуальных устройств и некоторых средств межпроцессного взаимодействия в виде файлов; использование конвейеров из нескольких программ, каждая из которых выполняет одну задачу.

Дуглас Макилрой (Douglas McIlroy), изобретатель каналов UNIX и один из основателей традиции UNIX, **обобщил философию следующим образом: «Философия UNIX гласит:** пишите программы, которые делают что-то одно и делают это хорошо; пишите программы, которые бы работали вместе; пишите программы, которые бы поддерживали текстовые потоки, поскольку это универсальный интерфейс».

В 1994 году Майк Ганцарз вывел девять основных принципов:

1. Простые решения красивы (Small is Beautiful).

2. Пусть каждая программа делает что-то одно, но хорошо.

3. Стройте прототип программы как можно раньше.

4. Предпочитайте переносимость эффективности.

5. Храните данные в простых текстовых файлах.

6. Извлекайте пользу из уже существующих программных решений.

7. Используйте скриптовые языки для уменьшения трудозатрат и улучшения переносимости.

8. Избегайте пользовательских интерфейсов, ограничивающих возможности пользователя по взаимодействию с системой.

9. Делайте каждую программу «фильтром».

**Комитет POSIX разработал стандарт 1003.1**, который определяет набор библиотечных процедур, которые должна обеспечивать каждая соответствующая данному стандарту система UNIX. Большая часть этих процедур делает системный вызов, но некоторые из них могут быть реализованы вне ядра. **Идея стандарта POSIX** заключается в том, что производитель программного обеспечения, который при написании программы использует только описанные в стандарте 1003.1 процедуры, может быть уверен, что его программа будет работать на любой системе UNIX, соответствующей данному стандарту. Также был принят ряд сопутствующих документов, которые стандартизируют потоки, утилиты, сетевое программное обеспечение и многие другие функции системы UNIX. Кроме того, язык C также был стандартизирован Национальным институтом стандартизации США (ANSI) и Международной организацией по стандартизации ISO.

**Unix** - это семейство операционных систем (ОС), обладающих сходной архитектурой и интерфесом с пользователем. Unix как явление зародилось в начале 70-х годов и развивается до сих пор. **Основные современные варианты UNIX:** Linux, BSD (FreeBSD, NetBSD, OpenBSD), AIX, HPUX, Solaris, SCO.

Важнейшие стандарты, обеспечивающие целостность семейства UNIX: POSIX - Portable Operating System Interface, ANSI C (c89 и с99).

**Классическая архитектура UNIX двухуровневая**: **ядро** - управляет ресурсами компьютера и предлагает программам базовый набор услуг (системные вызовы), **системные программы** (управление сетью, терминалами, печатью), прикладные программы (редакторы, утилиты, компиляторы и т.д.). Операционная система UNIX обладает классическим монолитным ядром, в котором можно выделить следующие основные части: файловая подсистема, управление процессами и драйверы устройств.

**Существует два основных объекта операционной системы** UNIX, с которыми приходится работать пользователю – **файлы и процессы**. Эти объекты сильно связаны друг с другом, и в целом организация работы с ними как раз и определяет архитектуру операционной системы.

Все данные пользователя хранятся в файлах; доступ к периферийным устройствам осуществляется посредством чтения и записи специальных файлов; во время выполнения программы, операционная система считывает исполняемый код из файла в память и передает ему управление. С другой стороны, вся функциональность операционная определяется выполнением соответствующих процессов. В частности, обращение к файлам на диске невозможно, если файловая подсистема операционной системы (совокупность процессов, осуществляющих доступ к файлам) не имеет необходимого для этого кода в памяти.

В операционной системе содержится большое количество стандартных программ, некоторые из них указаны в стандарте POSIX 1003.2, тогда как другие могут различаться от версии к версии. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускает пользователь с клавиатуры. Графические интерфейсы пользователя поддерживает оконная система X Windowing System, которую обычно называют Х11 (или просто Х). Она определяет обмен и протоколы отображения для управления окнами на растровых дисплеях UNIX-подобных систем.

**Пользователем** является объект, который обладает определенными правами и может запускать на выполнение программы и владеть файлами. Пользователями могут быть отдельные клиенты, удаленные компьютеры или группы пользователей с одинаковыми правами и функциями.

**В системе существует один пользователь, обладающий неограниченными правами это суперпользователь или администратор системы** (обычно с именем root). Каждый пользователь имеет уникальное или регистрационное имя, но система различает пользователей по идентификатору пользователя UID. Идентификаторы также должны быть уникальны. Пользователи являются членами одной или нескольких групп. Информация о пользователях обычно хранится в специальном файле: /etc/passwd. Этот файл доступен только для чтения. Писать в него может только администратор.

**Группа** — список пользователей, имеющих сходные задачи. Принадлежность к группе определяется дополнительными правами, которыми обладают все пользователи группы. Каждая группа имеет уникальное имя, а система различает группы по групповому идентификатору (GID). Информация о группах хранится в специальном файле: /etc/group.

Как правило, все **атрибуты** пользователя хранятся в файле /etc/passwd. Каждая строка файла является записью конкретного пользователя и имеет следующий формат: name:passwd-encod:UID:GID:comments:home-dir:shell — всего семь полей (атрибутов), разделенных двоеточиями. **name** - регистрационное имя пользователя. **passwd-encod** - пароль пользователя в закодированном виде. Пароль часто хранят в отдельном файле, а в поле passwd-encod ставится символ х. **UID** - числовой идентификатор пользователя, является внутренним представлением пользователя в системе**. GID** - определяет идентификатор первичной группы пользователя. **comments** - обычно это полное "реальное" имя пользователя. **home-dir** - домашний каталог пользователя. **shell** - имя программы, которую UNIX использует в качестве командного интерпретатора.

**Файловая структура ОС**

Файлы ядра можно обнаружить в каталоге /boot. В заархивированном виде ядро системы Линукс располагается в vmlinuz, где z - символ, прямо указывающий на то, ядро сжато до минимального размера с целью экономии свободного пространства **Файл initrd.img – это первичная файловая система**, монтирующая в первую очередь, прежде чем будет осуществлено подключение реальных физических накопители к виртуальной файловой системе VFS. В этом же месте можно обнаружить все другие дополняющие модули ядра, следовательно, данный файл в плане размерности может быть ощутимо больше самого ядра Линукс.

**В файле system.map** не составит труда обнаружить различные функций. Программы устанавливаются в различные директории в файловой системе. Одна из наиболее распространенных директорий для установки программ — это **/usr/bin.** Здесь обычно размещаются исполняемые файлы программ, доступные для всех пользователей системы. Эта директория имеет преимущество перед другими в пути поиска исполняемых файлов, что позволяет пользователю запускать программы из любой директории.

Другая важная директория для установки программ — это **/opt.** Она служит для установки приложений, которые не являются частью основной операционной системы. Здесь можно разместить программы, которые вы загрузили из Интернета или с которыми не связаны зависимости от других программ. Установка программ в эту директорию обеспечивает лучшую изоляцию и восстановление системы после обновления или удаления программ.

Наконец, важно упомянуть о **/usr/local** — директории, в которой обычно размещаются локально установленные программы. Здесь могут находиться программы, установленные самим пользователем или администратором системы и которые не входят в стандартный пакет установки Linux. Размещение программ в этой директории позволяет отделить их от программ, поставляемых с операционной системой, и облегчает их управление и обновление.

28. ОС UNIX: особенности процессов, сигналы, обработка сигналов.

**Основными активными сущностями в системе Linux являются процессы.** Каждый процесс выполняет одну программу и изначально получает один поток управления. Иначе говоря, у процесса есть один счетчик команд, который отслеживает следующую исполняемую команду. Linux позволяет процессу создавать дополнительные потоки (после того, как он начинает выполнение). Несколько независимых процессов могут работать одновременно. Более того, у каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Фактически на большинстве однопользовательских рабочих станций работают десятки **фоновых процессов, называемых демонами (daemons).** Они запускаются при загрузке системы из сценария оболочки. Процессы создаются в операционной системе Linux с помощью системного **вызова fork,** который создает точную копию исходного процесса, называемого **родительским процессом (parent process).** Новый процесс **называется дочерним процессом (child process**). У родительского и у дочернего процессов есть собственные (приватные) образы памяти. Открытые файлы используются родительским и дочерним процессами совместно. Процессы именуются своими PID-идентификаторами. При создании процесса его PID выдается родителю нового процесса. Если дочерний процесс желает узнать свой PID, то он может воспользоваться системным вызовом g**etpid.**

**Типы процессов**

* **Системные процессы** являются частью ядра и всегда расположены в оперативной памяти.
* **Демоны** — это неинтерактивные процессы, которые запускаются обычным образом — путем загрузки в память соответствующих им программ (исполняемых файлов), и выполняются в фоновом режиме.
* К **прикладным процессам** относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы.

**Сигналы** — это способ информирования процесса со стороны ядра о происшествии некоторого события. По сути, сигналы являются программными прерываниями. Смысл термина «сигнал» состоит в том, что сколько бы однотипных событий в системе не произошло, по поводу каждой такой группы событий процессу будет подан ровно один сигнал. То есть, сигнал означает, что определяемое им событие произошло, но не несет информации о том, сколько именно произошло однотипных событий.

Сигналы могут инициироваться одними процессами по отношению к другим процессам с помощью специального системного вызова kill. Процессы могут сообщить системе, какие действия следует предпринимать, когда придет входящий сигнал.

**Варианты такие**: проигнорировать сигнал, перехватить его, позволить сигналу убить процесс (действие по умолчанию для большинства сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуру обработки сигналов. Когда сигнал прибывает, управление сразу же передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова передается в то место, в котором оно находилось, когда пришел сигнал (это аналогично обработке аппаратных прерываний ввода-вывода).

В системе Linux процессы могут общаться друг с другом с помощью некой формы передачи сообщений. Можно создать **канал** между двумя процессами, в который один процесс сможет писать поток байтов, а другой процесс сможет его читать. Эти каналы иногда называют трубами (pipes). Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется. При помощи каналов организуются конвейеры оболочки. **Неименованный канал** (или программный канал) представляется в виде области памяти на внешнем запоминающем устройстве, управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы, т.е. это область памяти является разделяемым ресурсом.

Файловая система ОС Unix поддерживает специальные файлы, которые называются FIFO-файлами (или **именованными каналами**). Файлы этого типа очень схожи с обыкновенными файлами (в них можно писать и из них можно читать информацию), за исключением того факта, что они организованы по стратегии FIFO (т.е. невозможны операции, связанные с перемещением файлового указателя). Таким образом, файлы FIFO могут использоваться для организации взаимодействия процессов, при этом в отличие от неименованных каналов эти файлы могут существовать независимо от процессов, взаимодействующих через них. Эти файлы хранятся на внешних запоминающих устройствах, поэтому возможно открыть этот файл, записать в него информацию, а через любой промежуток времени (в течение которого допустимы перезагрузки системы) прочитать записанную информацию.

29. Основные принципы функционирования Linux. Основные компоненты Linux. Дистрибутивы Linux. Файловая система Linux.

Организация программно-аппаратных средств во всех UNIX-совместимых системах организована **по принципу клиент-сервер**. С точки зрения распределения функций, возложенных на систему, все компьютеры в сети работают как один большой компьютер, который может быть легко дополнен аппаратными ресурсами, когда к сети подключается новый компьютер. Каждый пользователь работает с системой через виртуальный терминал, которых может быть до 12-ти в зависимости от версии Linux (на экране обозначается как tty1..tty12**). В ОС Linux все файлы организованы в непрерывный поток байтов**. Данные, вводимые с клавиатуры, представляют собой входной поток данных, а отображаемые данные – выходной поток. Поскольку процедуры ввода и вывода организованы также, как и файлы, то они могут свободно взаимодействовать с файлами. Чтобы передать данные из одной команды в другую, можно использовать конвейеры. **Файл может быть каталогом или исполняемой программой (командой).** Команда file помогает определить, для чего используется данный файл (например, текстовый файл или каталог). Файловая система в ОС Linux как и в большинстве других систем имеет иерархическую (древовидную) структуру. Вверху дерева всегда находится корневой каталог ROOT. В этой операционной системе также справедливо понятие текущего каталога. Каждый файл имеет относительное имя пути, которое определяет его принадлежность к какому-либо каталогу, и абсолютное имя пути, которое показывает весь путь файла, начиная от корневого каталога.

**Название Linux в первую очередь относится к ядру операционной системы**. Вообще говоря, в операционных системах этого семейства можно выделить следующие компоненты: **ядро, модули, планировщик, файл подкачки, файловые системы, механизмы защиты, инструменты администрирования, серверные возможности, графический интерфейс пользователя, система управления пакетами** и т. д.

**В ядре** собрана основная функциональность для работы с памятью, управления процессами и т.д. Когда ядру требуется дополнительные функции, он обращается к модулям. В модулях, например, может содержаться код для работы с оборудованием. Файлы ядра Linux находятся в /boot.

Так как ядро Linux обеспечивает одновременную работу нескольких процессов от нескольких пользователей (с поддержкой нескольких процессоров), операционная система нуждается в средствах управления многопоточностью**. Планировщик Linux** назначает процессам приоритеты и определяет, какой процесс выполняется на конкретном процессоре (если в системе установлено несколько процессоров). Планировщик можно настроить для работы в системах различного типа. При правильной настройке более важные процессы получают более быструю реакцию процессора. Например, планировщик Linux на настольном компьютере предоставляет больший приоритет задаче перемещения окна и меньший — задаче фонового копирования файлов.

Ядро Linux старается держать работающие в данный момент процессы в оперативной памяти. Простаивающие процессы перемещаются **в файл подкачки**, представляющий собой выделенную область на жестком диске, которая используется для хранения не перемещающихся в оперативную память данных и процессов. При переполнении оперативной памяти процессы выносятся в файл подкачки. При переполнении файла подкачки (но этого допускать нельзя) новые процессы не запускаются.

Ядро Linux поддерживает тысячи аппаратных устройств. При этом за счет включения в работающее ядро только актуальных драйверов размер ядра удается сохранять на приемлемом уровне. Использование загружаемых модулей позволяет добавить в ядро поддержку дополнительных устройств**. Модули** можно загружать и выгружать по запросу в результате подключения или отключения устройства.

**Файловые системы.** Как и UNIX, операционная система Linux изначально создавалась для обеспечения, одновременного многопользовательского доступа. Для защиты пользовательских ресурсов каждому файлу назначаются наборы разрешений на чтение, запись и выполнение, которые определяют права доступа. В стандартной системе Linux пользователь root имеет доступ ко всей системе без ограничений, специальные регистрационные записи могут управлять определенными службами (например, службами Web-сервера Apache), а пользователям могут присваиваться разрешения по отдельности или в составе групп. Последние нововведения, например, Security-Enhanced Linux (SELinux), поддерживают более тонкую настройку и защиту безопасных сред обработки информации.

**Механизмы защиты.** Как и UNIX, операционная система Linux изначально создавалась для обеспечения, одновременного многопользовательского доступа. Для защиты пользовательских ресурсов каждому файлу назначаются наборы разрешений на чтение, запись и выполнение, которые определяют права доступа. В стандартной системе Linux пользователь root имеет доступ ко всей системе без ограничений, специальные регистрационные записи могут управлять определенными службами (например, службами Web-сервера Apache), а пользователям могут присваиваться разрешения по отдельности или в составе групп. Последние нововведения, например, Security-Enhanced Linux (SELinux), поддерживают более тонкую настройку и защиту безопасных сред обработки информации.

**Инструменты администрировани**я включают в себя сотни (а возможно и тысячи) команд и графических утилит, которые позволяют добавлять пользователей, управлять дисками, следить за состоянием сети, устанавливать программное обеспечение, а также гарантировать безопасность и управлять ресурсами компьютера.

**Серверные возможности** позволяют компьютеру под управлением Linux предоставлять службы для клиентов в сети. Иными словами, кроме установки Web-обозревателей для просмотра Web-страниц, компьютер можно превратить в сервер, который предоставляет Web-страницы другим компьютерам. При этом среди популярных серверных функций можно назвать Web-серверы, серверы электронной почты, баз данных, печати, файловые серверы, серверы DNS и DHCP.

**Графический интерфейс** пользователя состоит из графической инфраструктуры (обычно это X WindowSystem), оконных менеджеров, панелей, пиктограмм и меню. Графический интерфейс пользователя позволяет применять комбинацию мыши и клавиатуры вместо простого ввода команд с клавиатуры.

**Система управления пакетами** - это набор инструментов, предназначенных для автоматизации процессов установки, обновления, конфигурирования и удаления пакетов программного обеспечения определенного формата. Наиболее известными (или распространенными) системами управления пакетами являются: RPM/YUM — менеджер пакетов Red Hat; dpkg/APT — система управления пакетами \*.deb дистрибутива Debian; tgz или tar.gz — стандартный набор из двух программ tar + gzip; система портежей дистрибутива Gentoo; YaST - утилита, разработанная Novell и используемая в дистрибутиве SuSE.

**Дистрибутивы** состоят из отдельных пакетов, каждый из которых содержит какое-то приложение, утилиту или сервис. Отдельный пакет может содержать, например, веб-браузер, библиотеку для работы с графическими файлами в формате PNG, набор шрифтов и так далее. Программное обеспечение, содержащееся в пакете, поставляется в одном из двух основных видов: в виде бинарных файлов, которые предназначены для непосредственной установки в вашу систему, без какой-либо дополнительной обработки (например, компиляции); в виде исходных текстов, которые обычно содержат текст на каком-то языке программирования, заархивированный в формате tar и упакованный программой gzip, а также вспомогательные файлы, необходимые для компиляции приложения из файлов пакета.

Мы можем распределить дистрибутивы Linux на три группы:

• Enterprise Grade Linux Distributions

* Red Hat Enterprise Linux
* CentOS
* SUSE Linux Enterprise Server
* Debian GNU/Linux
* Ubuntu LTS

• Consumer Grade Linux Distributions

* Fedora
* Ubuntu non-LTS
* openSUSE

• Experimental and Hacker Linux Distributions

* Arch
* Gentoo

**Дистрибутивы Enterprise Grade Linux** предназначены для развертывания в крупных организациях с использованием оборудования предприятия. Они обычно включают более старые выпуски ядра и другое программное обеспечение, которое, как известно, работает надежно. Часто дистрибутивы портируют важные обновления, такие как исправления безопасности, на эти стабильные версии**.**

**Дистрибутивы Consumer Grade Linux** больше ориентированы на малый бизнес или домашних пользователей и любителей. Они подготовлены для использования новейшего оборудования, установленного в системах потребительского уровня. Этим системам потребуются новейшие драйверы, чтобы максимально использовать новое оборудование.

**Дистрибутивы Experimental and Hacker Linux** используют самые современные технологии. Они содержат самые последние версии программного обеспечения, даже если эти версии все еще содержат ошибки и непроверенные **функции.**

**Файловая система Linux**

В операционной системе GNU/Linux все объекты являются файлами, в том числе и директории, которые служат для организации доступа к файлам. Существуют следующие типы файлов: обычные файлы, каталоги, символьные ссылки, блочные устройства, символьные устройства, сокеты, каналы. В любой файловой системе Linux всегда есть только один корневой каталог, который называется /. Пользователь Linux всегда работает с единым деревом каталогов, даже если данные расположены на разных носителях: жёстких или съёмных дисках, CD-ROM, сетевых дисках и т. д.

Положение любого каталога в дереве каталогов точно и однозначно описывается при помощи полного пути. Полный путь всегда начинается от корневого каталога и состоит из перечисления всех вершин, встретившихся при движении по рёбрам дерева до искомого каталога включительно. Названия соседних вершин разделяются символом наклонной черты / (слэш). Для каждого процесса Linux определена текущая директория, с которого система начинает относительный путь при выполнении файловых операций. Между полным путём и относительным есть только одно существенное различие: относительный путь начинается от текущей директории, в то время как полный путь всегда начинается от корневой директории. То есть, первый символ полного пути всегда будет /. Точка . в начале имени файла делает его скрытым, то есть, он не показывается в выводе команды ls.

**Домашний каталог** (домашняя папка, домашняя директория) – предназначен для хранения собственных данных пользователя Linux и личных настроек для программ. Как правило, становится текущим непосредственно после регистрации пользователя в системе. Полный путь к домашнему каталогу хранится в переменной окружения HOME, в полном пути к файлу можно заменять на знак тильды ~. Для обычных пользователей домашний каталог находится в директории /home.

**FHS (Filesystem Hierarchy Standard)** – стандарт файловой системы, специфичной для GNU/Linux, создан в 1994-1996 гг. Текущая версия стандарта FHS 3.0 выпущена 03.06.2015 г. Некоторые Linux-системы отвергают FHS и следуют своему собственному стандарту.В FHS все файлы и каталоги находятся внутри корневого каталога, даже если они расположены на различных физических носителях. В корневой директории должны быть следующие директории: bin, boot, dev, etc, lib, media, mnt, opt, run, sbin, srv, tmp, usr, var. При наличии соответствующих подсистем добавляются папки home, root и lib64.

30. ОС Linux: управление процессами, выполнение задач в фоновом режиме, изменение приоритетов выполняющихся программ.

Процессы существуют в иерархии: после загрузки ядра в память запускается первый процесс (init или systemd), который, в свою очередь, запускает другие процессы, которые, опять же, могут запускать другие процессы. Каждый раз, когда пользователь вводит команду, запускается программа и генерируется один или несколько процессов. **Каждый процесс имеет уникальный идентификатор (PID) и идентификатор родительского процесса (PPID)**. Это положительные целые числа, которые назначаются в последовательном порядке.

**Команда top** динамически отображает все запущенные процессы

Альтернативные команды – htop и atop. **Команда ps** выводит статическую информацию о процессах. Без опций выводит только процессы, относящиеся к текущей оболочке. Команда поддерживает три стиля параметров, причем они не совпадают. Например, ps aux не эквивалентна ps –aux, но ps -e эквивалентна ps ax. Иерархию процессов можно увидеть с **помощью команды pstree**. Завершить процесс можно с помощью **команды kill.**

**Для выполнения команды в фоновом режиме достаточно добавить в конце символ амперсанда &.** В выводе терминала будут отображены порядковый номер задачи (в квадратных скобках) и идентификатор процесса. Работая в фоновом режиме, команда все равно продолжает выводить сообщения в терминал, из которого была запущена. Для этого она использует потоки stdout и stderr, которые можно закрыть при помощи следующего синтаксиса: $ command > /dev/null 2>&1 &

Узнать состояние всех остановленных и выполняемых в фоновом режиме задач в рамках текущей сессии терминала можно при помощи **утилиты jobs** c использованием опции -l. Вывод содержит порядковый номер задачи, идентификатор фонового процесса, состояние задачи и название команды, которая запустила задание. В любое время можно вернуть процесс из фонового режима на передний план. Для этого **служит команда fg.** Если в фоновом режиме выполняется несколько программ, следует также указывать номер.

Для завершения фонового процесса применяют команду kill с номером программы. Если изначально процесс был запущен обычным способом, его можно перевести в фоновый режим, выполнив следующие действия:

• Остановить выполнение команды, нажав комбинацию клавиш Ctrl+Z.

• Перевести процесс в фоновый режим при помощи команды bg.

**Утилита nice** — программа, предназначенная для запуска процессов с изменённым приоритетом nice. Приоритет nice (целое число) процесса используется планировщиком процессов ядра ОС при распределении процессорного времени между процессами. **Приоритет nice** — число, указывающее планировщику процессов ядра ОС приоритет, который пользователь хотел бы назначить процессу.

Утилита nice, запущенная без аргументов, выводит приоритет nice, унаследованный от родительского процесса. nice принимает аргумент «смещение» в диапазоне от -20 (наивысший приоритет) до +19 (низший приоритет). Если указать смещение и путь к исполняемому файлу, утилита nice получит приоритет своего процесса, изменит его на указанное смещение и использует системный вызов семейства exec() для замещения кода своего процесса кодом из указанного исполняемого файла. Команда nice сделает то же, но сначала выполнит системный вызов семейства fork() для запуска дочернего процесса (sub-shell). Если смещение не указано, будет использовано смещение +10. Привилегированный пользователь (root) может указать отрицательное смещение.

Для того, чтобы изменить приоритет у существующего процесса (т.е. такого процесса, который ранее был уже запущен), необходимо воспользоваться **командой renice** [значение приоритета] -p [id процесса]. У запущенной программы с помощью команды renice можно изменить назначенный приоритет. Согласно правилам, обычный пользователь может только увеличивать значение nice (уменьшать приоритет) любого процесса.

31. Понятие безопасности ОС. Основные угрозы безопасности ОС. Методы и защитные механизмы операционных систем.

Во многих работах, посвященных безопасности, безопасность информационных систем разбита на три компонента: **конфиденциальность, целостность и доступность**. Вместе все три компонента часто называют CIA (Confidentiality, Integrity, Availability). **Конфиденциальность** направлена на сохранение секретности данных. **Целостность** означает, что пользователи, не обладающие необходимыми правами, не должны иметь возможности изменять какие-либо данные без разрешения их владельцев (в этом контексте изменение данных включает в себя не только внесение в них изменений, но и их удаление или добавление в них ложных данных). **Доступность** означает, что никто не может нарушить работу системы и вывести ее из строя. Позже были добавлены дополнительные свойства, такие как аутентичность (authenticity), идентифицируемость (accountability), неотвергаемость (nonrepudiability), закрытость (privacy) и др.

**Защищаемая информация** — информация, подлежащая защите в соответствии с требованиями нормативных правовых актов или требованиями, устанавливаемыми обладателем информации.

**Обладатель информации** — лицо, самостоятельно создавшее информацию либо получившее на основании закона или договора право разрешать или ограничивать доступ к информации, определяемой по каким-либо признакам. Обладателями информации могут быть: государство, юридическое лицо, группа физических лиц, отдельное физическое лицо.

**Организация защиты информации** — совокупность действий, направленных на выявление угроз безопасности информации, планирование, реализацию мероприятий по защите информации и контроль состояния защиты информации.

**Система защиты информации** — совокупность органов и (или) исполнителей, используемой ими техники защиты информации, а также объектов защиты информации, организованная и функционирующая в соответствии с требованиями о защите информации.

**Классификация угроз по цели атаки**:

• несанкционированное чтение информации;

• несанкционированное изменение информации;

• несанкционированное уничтожение информации;

• полное или частичное разрушение операционной системы.

**Классификация угроз по принципу воздействия на операционную систему:**

• использование известных (легальных) каналов получения информации, например, угроза несанкционированного чтения файла, доступ пользователей к которому определен некорректно – разрешен доступ пользователю, которому, согласно политике безопасности, доступ должен быть запрещен;

• использование скрытых каналов получения информации, например, угроза использования злоумышленником недокументированных возможностей операционной системы;

• создание новых каналов получения информации с помощью программных закладок.

**Классификация угроз по типу используемой злоумышленником уязвимости защиты***:*

• неадекватная политика безопасности, в том числе и ошибки администратора системы;

• ошибки и недокументированные возможности программного обеспечения операционной системы, в том числе и так называемые люки – случайно или преднамеренно встроенные в систему «служебные входы», позволяющие обходить систему защиты;

• ранее внедренная программная закладка.

**Классификация угроз по характеру воздействия на ОС**:

• активное воздействие – несанкционированные действия злоумышленника в системе;

• пассивное воздействие – несанкционированное наблюдение злоумышленника за процессами, происходящими в системе.

**Операционная система может подвергнуться следующим типичным атакам**: сканирование файловой системы, подбор пароля, кража ключевой информации, сборка мусора, превышение полномочий, программные закладки, жадные программы.

**Операционную систему называют защищенной**, если она предусматривает средства защиты от основных классов угроз. Защищенная операционная система обязательно должна содержать средства разграничения доступа пользователей к своим ресурсам, а также средства проверки подлинности пользователя, начинающего работу с операционной системой. Кроме того, защищенная операционная система должна содержать средства противодействия случайному или преднамеренному выводу операционной системы из строя. Если операционная система предусматривает защиту не от всех основных классов угроз, а только от некоторых, такую ОС называют **частично защищенной.**

Существует два основных подхода к созданию защищенных операционных систем – фрагментарный и комплексный

**При фрагментарном подходе** вначале организуется защита от одной угрозы, затем от другой. Примером фрагментарного подхода может служить ситуация, когда за основу берется незащищенная операционная система, на нее устанавливают антивирусный пакет, систему шифрования, систему регистрации действий пользователей и т. д. При применении фрагментарного подхода подсистема защиты операционной системы представляет собой набор разрозненных программных продуктов, как правило, от разных производителей. Эти программные средства работают независимо друг от друга, при этом практически невозможно организовать их тесное взаимодействие. Кроме того, отдельные элементы такой подсистемы защиты могут некорректно работать в присутствии друг друга, что приводит к резкому снижению надежности системы.

**При комплексном подходе** защитные функции вносятся в операционную систему на этапе проектирования архитектуры операционной системы и являются ее неотъемлемой частью. Отдельные элементы подсистемы защиты, созданной на основе комплексного подхода, тесно взаимодействуют друг с другом при решении различных задач, связанных с организацией защиты информации, поэтому конфликты между ее отдельными компонентами практически невозможны. Подсистема защиты, созданная на основе комплексного подхода, может быть устроена так, что при фатальных сбоях в функционировании ее ключевых элементов она вызывает крах операционной системы, что не позволяет злоумышленнику отключать защитные функции системы. При фрагментарном подходе такая организация подсистемы защиты невозможна. Как правило, подсистему защиты операционной системы, созданную на основе комплексного подхода, проектируют так, чтобы отдельные ее элементы были заменяемы. Соответствующие программные модули могут быть заменены другими модулями.

Программно-аппаратные средства защиты операционной системы обязательно должны дополняться **административными мерами защиты**. Без постоянной квалифицированной поддержки со стороны администратора даже надежная программно-аппаратная защита может давать сбои.

Перечислим основные административные меры защиты.

* Постоянный контроль корректности функционирования операционной системы, особенно ее подсистемы защиты.
* Организация и поддержание адекватной политики безопасности.
* Осведомление пользователей операционной системы о необходимости соблюдения мер безопасности при работе с ОС и контроль за соблюдением этих мер.
* Регулярное создание и обновление резервных копий программ и данных ОС.
* Постоянный контроль изменений в конфигурационных данных и политике безопасности ОС.

**Архитектура подсистемы защиты операционной системы**

Подсистема защиты ОС выполняет следующие основные функции: **идентификация и аутентификация; разграничение доступа; аудит; управление политикой безопасности; криптографические функции; сетевые функции.** Подсистема защиты обычно не представляет собой единый программный модуль. Как правило, каждая из перечисленных функций подсистемы защиты решается одним или несколькими программными модулями. Некоторые функции встраиваются непосредственно в ядро ОС.

**Идентификация** субъекта доступа заключается в том, что пользователь (субъект) сообщает операционной системе идентифицирующую информацию о себе (имя, учетный номер) и таким образом идентифицирует себя.

**Аутентификация** субъекта доступа заключается в том, что субъект предоставляет операционной системе, помимо идентифицирующей информации, еще и аутентифицирующую информацию, подтверждающую, что он действительно является тем субъектом доступа, к которому относится идентифицирующая информация.

**Авторизация** субъекта доступа происходит после успешной идентификации и аутентификации. При авторизации субъекта ОС выполняет действия, необходимые для того, чтобы субъект мог начать работу в системе. Авторизация субъекта не относится напрямую к подсистеме защиты операционной системы. В процессе авторизации решаются технические задачи, связанные с организацией начала работы в системе уже идентифицированного и аутентифицированного субъекта доступа.

**Криптография** – это метод защиты информации путем использования закодированных алгоритмов, хэшей и подписей. Информация может находиться на этапе хранения (например, файл на жестком диске), передачи (например, электронная связь между двумя или несколькими сторонами) или использования (при применении для вычислений). Криптография использует некоторые низкоуровневые криптографические алгоритмы для достижения одной или нескольких из этих целей информационной безопасности. Среди этих инструментов – алгоритмы шифрования, алгоритмы цифровой подписи, алгоритмы хэширования и другие функции.

**Алгоритм шифрования** – это процедура, которая преобразует сообщение в формате неформатированного текста в зашифрованный текст. Современные алгоритмы используют сложные математические вычисления и один или несколько ключей шифрования. Благодаря этому можно относительно легко зашифровать сообщение, но практически невозможно расшифровать его, не зная ключей. В зависимости от того, как действуют ключи, технологии шифрования делятся на две категории: симметричные и асимметричные.

**Алгоритмы шифрования с симметричным ключом** используют одни и те же криптографические ключи для шифрования простого текста и расшифровки зашифрованного. При использовании симметричного шифрования все получатели сообщения должны иметь доступ к общему ключу.

**Схема шифрования называется асимметричной,** если в ней один ключ (открытый) используется для шифрования данных, а другой, но математически связанный (частный) – для их расшифровки. Необходимо, чтобы было невозможно вычислить частный ключ, если известен только открытый. Поэтому общий открытый ключ можно передавать, а частный – держать в тайне и в безопасности. Эти ключи называются парой ключей.

Сочетание криптографии с открытым ключом для обмена ключей и симметричного шифрования для пакетного шифрования данных называется **гибридным шифрованием.** В гибридном шифровании используются уникальные свойства криптографии с открытым ключом для обмена секретной информацией по недоверенному каналу с эффективностью симметричного шифрования. Оно представляет собой практически применимое сквозное решение для обеспечения конфиденциальности данных. Гибридное шифрование широко используется в протоколах передачи данных для Интернета, таких как протокол TLS (безопасность транспортного уровня). Когда вы подключаетесь к веб-сайту, который использует HTTPS (безопасный HTTP с TLS), браузер согласовывает криптографические алгоритмы, защищающие соединение. Это алгоритмы обмена ключами, симметричного шифрования и цифровой подписи.

**Схемы цифровых подписей** – это тип криптографии с открытым ключом, который гарантирует целостность, подлинность и обеспечение невозможности отказа. Процесс подписания можно воспринимать как шифрование файла с использованием частного ключа. Лицо, подписывающее цифровой документ, например, файл или фрагмент кода, использует для создания «подписи» свой частный ключ. Эта подпись является уникальной для пары документ-частный ключ и может прикрепляться к документу и проверяться с использованием открытого ключа лица, ставящего подпись. Двумя распространенными алгоритмами цифровой подписи являются RSA с вероятностной схемой подписи (RSA-PSS) и алгоритм цифровой подписи (DSA).

**Код аутентификации сообщения (MAC)** – это симметричная версия цифровой подписи. При использовании MAC две или больше сторон совместно используют ключ. Одна сторона создает тег MAC, который является симметричной версией цифровой подписи, и прикрепляет его к документу. Другая сторона может проверить целостность сообщения с использованием того же ключа, который использовался для создания тега.Обратите внимание, что несколько сторон совместно используют ключ, с помощью которого создавались теги MAC, поэтому MAC невозможно использовать для аутентификации или обеспечения невозможности отказа, потому что неизвестно, какая сторона создала тег.

**Криптографическая хэш-функция** – это инструмент для преобразования произвольных данных в «отпечаток» фиксированной длины. Хэш-функции создаются таким образом, чтобы было сложно найти два различных набора входных данных, дающих один и тот же отпечаток, и чтобы было сложно найти сообщение, отпечаток которого совпадает с фиксированным значением. В отличие от схем шифрования, схем подписей и MAC, хэш-функции не имеют ключа. Кто угодно может вычислить хэш для данного входного значения, и хэш-функция всегда будет генерировать одно и то же самое выходное значение для одного и того же входного.

**Права доступа**

**Домен (domain)** представляет собой множество пар (объект, права доступа). Каждая пара определяет объект и некоторое подмножество операций, которые могут быть выполнены в отношении этого объекта. **Права доступа (rights)** означают в данном контексте разрешение на выполнение той или иной операции. Зачастую домен соотносится с отдельным пользователем, сообщая о том, что может, а что не может сделать этот пользователь, но он может также иметь и более общий характер, распространяясь не только на отдельного пользователя. К примеру, сотрудники одной группы программистов, работающие над одним и тем же проектом, могут целиком принадлежать к одному и тому же домену и иметь доступ к файлам проекта.

**Объектом доступа** (или просто объектом) называют любой элемент операционной системы, доступ к которому пользователей и других субъектов доступа может быть произвольно ограничен. Возможность доступа к объектам ОС определяется не только архитектурой операционной системы, но и текущей политикой безопасности. Под объектами доступа понимают, как ресурсы оборудования, так и программные ресурсы. В качестве примера ресурсов оборудования можно привести процессор, принтер, жесткие диски и ленты. Каждый объект имеет уникальное имя, отличающее его от других объектов в системе, и может быть доступен через хорошо определенные и значимые операции.

**Методом доступа к объекту** называется операция, определенная для объекта. Тип операции зависит от объектов. Например, процессор может только выполнять команды, сегменты памяти могут быть записаны и прочитаны, считыватель магнитных карт может только читать, а для файлов могут быть определены методы доступа «чтение», «запись» и «добавление» (дописывание информации в конец файла).

**Субъектом доступа** называют любую сущность, способную инициировать выполнение операций над объектами (обращаться к объектам по некоторым методам доступа). Обычно полагают, что множество субъектов доступа и множество объектов доступа не пересекаются.

Д**ля объекта доступа может быть определен владелец – субъект,** которому принадлежит данный объект и который несет ответственность за конфиденциальность содержащейся в объекте информации, а также за целостность и доступность объекта. Обычно владельцем объекта автоматически назначается субъект, создавший данный объект; в дальнейшем владелец объекта может быть изменен с использованием соответствующего метода доступа к объекту. На владельца, как правило, возлагается ответственность за корректное ограничение прав доступа к данному объекту других субъектов.

**Разграничением доступа субъектов** к объектам является совокупность правил, определяющая для каждой тройки субъект–объект–метод, разрешен ли доступ данного субъекта к данному объекту по данному методу. При избирательном разграничении доступа возможность доступа определена однозначно для каждой тройки субъект–объект–метод, при полномочном разграничении доступа ситуация несколько сложнее.

Субъекта доступа называют **суперпользователем**, если он имеет возможность игнорировать правила разграничения доступа к объектам.

**Существуют две основные модели разграничения доступа**:

* избирательное (дискреционное) разграничение доступа;
* полномочное (мандатное) разграничение доступа.

При **избирательном** разграничении доступа (Discretionary Access Control) определенные операции над конкретным ресурсом запрещаются или разрешаются субъектам или группам субъектов. Большинство операционных систем реализуют именно избирательное разграничение доступа.

**Полномочное разграничение доступа** заключается в том, что все объекты могут иметь уровни секретности, а все субъекты делятся на группы, образующие иерархию в соответствии с уровнем допуска к информации. Иногда эту модель называют моделью многоуровневой безопасности, предназначенной для хранения секретов.

**ACL (Access Control List)** - упорядоченный список, содержащий все домены, которым разрешен доступ к данному объекту, а также тип доступа.

**При использовании изолированной программной** среды права субъекта на доступ к объекту определяются не только правами и привилегиями субъекта, но и процессом, с помощью которого субъект обращается к объекту. Можно, например, разрешить обращаться к файлам с расширением .doc только программам Word, Word Viewer и WPview.

32. Механизмы безопасности в операционных системах семейства Windows.

**Идентификатор безопасности (SID)** — это уникальное значение переменной длины, используемое для идентификации доверенного лица.

Идентификатор безопасности используется для уникальной идентификации субъекта безопасности или группы безопасности. Субъекты безопасности могут представлять любую сущность, которая может быть проверена операционной системой, например учетной записью пользователя, учетной записью компьютера или потоком или процессом, выполняющимся в контексте безопасности учетной записи пользователя или компьютера. Каждая учетная запись или группа или каждый процесс, выполняемый в контексте безопасности учетной записи, имеет уникальный идентификатор безопасности, выданный центром, например контроллером домена Windows. Идентификатор безопасности хранится в базе данных безопасности. Система создает идентификатор безопасности, определяющий определенную учетную запись или группу во время создания учетной записи или группы. Если идентификатор безопасности используется в качестве уникального идентификатора для пользователя или группы, он никогда не может использоваться повторно для идентификации другого пользователя или группы.

**Маркер доступа** — это объект, описывающий контекст безопасности процесса или потока. При каждом входе пользователя система создает маркер доступа для этого пользователя. Маркер доступа содержит идентификатор безопасности пользователя, права пользователя и идентификаторы SID для всех групп, к которому принадлежит пользователь. Этот маркер предоставляет контекст безопасности для любых действий, выполняемых пользователем на этом компьютере.

Маркеры доступа содержат следующие сведения:

• Идентификатор безопасности (SID) для учетной записи пользователя

• Идентификаторы безопасности для групп, членом которых является пользователь

• Идентификатор безопасности входа, который идентифицирует текущий сеанс входа в систему.

• Список привилегий, которыми пользовались пользователи или группы пользователей.

• Идентификатор безопасности владельца

• Идентификатор безопасности для основной группы

• DACL по умолчанию, используемый системой, когда пользователь создает защищаемый объект без указания дескриптора безопасности.

• Источник маркера доступа

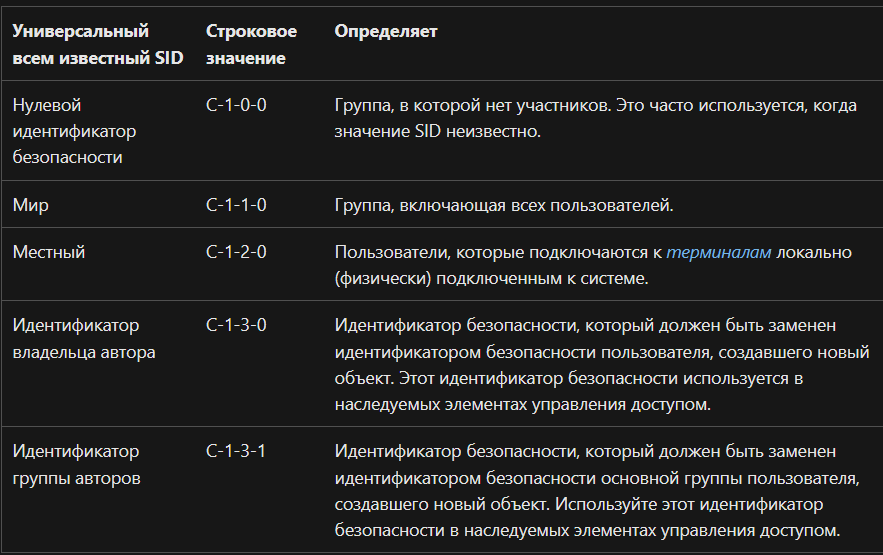
• Является ли маркер основным или маркером олицетворения

• Необязательный список ограничений идентификаторов безопасности

• Текущие уровни олицетворения

• Другая статистика.

Используйте функцию **OpenProcessToken**, чтобы получить дескриптор основного маркера процесса. Используйте функцию **OpenThreadToken** для получения дескриптора маркера олицетворения потока.



**Дескриптор безопасности** содержит сведения о безопасности, связанные с защищаемым объектом. Дескриптор безопасности состоит из структуры SECURITY\_DESCRIPTOR и связанных с ней сведений о безопасности.

Дескриптор безопасности может содержать следующие сведения о безопасности:

• Идентификаторы безопасности для владельца и основной группы объекта.

• DACL, указывающий права доступа, разрешенные или запрещенные определенным пользователям или группам.

• SACL, указывающий типы попыток доступа, которые создают записи аудита для объекта.

• Набор битов элементов управления, которые определяют значение дескриптора безопасности или его отдельных членов.

**Списки управления доступом (ACL)** состоят из записей управления доступом (ACE). Каждый ACE в ACL идентифицирует доверенного лица и указывает права доступа, разрешенные, запрещенные или регистрируемые для этого доверенного лица. Дескриптор безопасности для защищаемого объекта может содержать два типа списков управления доступом: **DACL и SACL**

**Список управления доступом на уровне пользователей (DACL)** определяет доверенных лиц, которым разрешен или запрещен доступ к защищаемому объекту. Когда процесс пытается получить доступ к защищаемому объекту, система проверяет ACE в DACL объекта, чтобы определить, следует ли предоставить доступ к нему. Если у объекта нет DACL, система предоставляет полный доступ всем пользователям. Если DACL объекта не имеет ACE, система отклоняет все попытки доступа к объекту, так как DACL не разрешает какие-либо права доступа. Система проверяет ACE в последовательности, пока не найдет один или несколько ACE, которые разрешают все запрошенные права доступа, или пока не будет отказано в любом из запрошенных прав доступа.

**Системный список управления доступом (SACL)** позволяет администраторам регистрировать попытки доступа к защищенному объекту. Каждый ACE указывает типы попыток доступа со стороны указанного доверенного лица, которые приводят к созданию системой записи в журнале событий безопасности. ACE в SACL может создавать записи аудита при сбое попытки доступа и/или при успешном выполнении.

**Каталог** — это иерархическая структура, в которой хранятся сведения об объектах в сети. Служба каталогов, например, службы домен Active Directory (AD DS), предоставляет методы хранения данных каталога и предоставления этих данных сетевым пользователям и администраторам.

**Логическая структура Active Directory** представляет собой иерархическую структуру доменов Active Directory и подразделений в лесу. Взаимосвязь компонентов логической структуры контролирует доступ к хранимым данным, а также то, как информация реплицируется между различными контроллерами домена в лесу

Active Directory также включает следующие компоненты:

• Схема — набор правил, который определяет классы объектов и атрибутов, содержащихся в каталоге, ограничения и ограничения экземпляров этих объектов, а также формат их имен.

• Глобальный каталог, содержащий сведения о каждом объекте в каталоге. Это позволяет пользователям и администраторам находить сведения о каталоге независимо от того, какой домен в каталоге фактически содержит данные.

• Механизм запроса и индекса, чтобы объекты и их свойства могли быть опубликованы и найдены сетевыми пользователями или приложениями.

• Служба репликации, которая синхронизирует данные каталога по сети. Все контроллеры домена в домене участвуют в репликации и содержат полную копию всех сведений о каталоге для своего домена. Любые изменения данных каталога реплицируются в домене на все контроллеры домена.

**Структура и архитектура хранения Active Directory состоят из четырех частей**:

• Домены и леса Active Directory. Леса, домены и организационные единицы (OU) составляют основные элементы логической структуры Active Directory. Лес определяет один каталог и представляет собой границу безопасности. Леса содержат домены.

• Поддержка системы доменных имен (DNS) для Active Directory. DNS предоставляет службу разрешения имен для определения местоположения контроллера домена и иерархическую структуру, которую Active Directory может использовать для обеспечения соглашения об именах, которое может отражать организационную структуру.

• Схема. Схема предоставляет определения объектов, которые используются для создания объектов, хранящихся в каталоге.

• Хранилище данных. Хранилище данных — это часть каталога, которая управляет хранением и получением данных на каждом контроллере домена.

**Active Directory имеет два типа групп**:

• Группы безопасности. Используйте для назначения разрешений общим ресурсам.

• Группы рассылки: создание списков рассылки электронной почты

Группы безопасности могут обеспечить эффективный способ назначения доступа к ресурсам в сети.

Группы рассылки можно использовать только для отправки электронной почты в коллекции пользователей с помощью почтового приложения, например, Exchange Server. Группы рассылки не включены в систему безопасности, поэтому их нельзя включить в списки управления доступом.

33. Механизмы безопасности в операционных системах семейства Linux.

В GNU/Linux как и других Unix-подобных операционных системах понятие типа файла не связано с расширением файла (несколькими буквами после точки в конце имени), как это обстоит в Windows. Поэтому тип файла в Linux – это скорее тип объекта, но не тип данных как в Windows. В операционной системе GNU/Linux существуют следующие типы файлов:

• обычные файлы • каталог • символьные ссылки • блочные устройства • символьные устройства • сокеты • каналы Каждый тип имеет собственное обозначение одним символом.

**Обычные файлы(-)** Сюда относятся все файлы с данными, играющими роль ценной информации сами по себе. Linux не различает текстовые файлы, исполняемые или картинки. В любом случае это будет обычный (regular) файл. Все они обозначаются знаком "-". Остальные типы файлов считаются специальными (special).

**Каталоги**(d) В Linux каталог представляет собой такой тип файла, данными которого является список имен других файлов и каталогов, вложенных в данный каталог. Напрямую, то есть через какой-либо редактор, пользователь не может редактировать данные файла-каталога. Редактированием занимается ядро операционной системы, получая, в том числе от пользователя, команды создания файла, удаления и др. В файле каталога осуществляется связь между именами файлов (словесного обозначения для людей) и их индексными дескрипторами (истинным именем-числом, которым оперирует ОС). В Unix-подобных системах один и тот же файл может существовать под разными именами и/или в разных каталогах: все имена будут связаны с одним и тем же индексным дескриптором (механизм жестких ссылок). Также следует, что файлы всегда должны содержаться в каталогах, иначе станут недоступны, так как нигде не будет содержаться записи о них.

**Символьные ссылки(l**) Символьная ссылка – это файл, в данных которого содержится адрес другого файла по его имени (а не индексному дескриптору). Выполнение символьной ссылки приводит к открытию файла, на который она указывает. Это аналог ярлыков в операционной системе Windows. Если удалить исходный файл, то символьная ссылка продолжит существовать. Она по-прежнему будет указывать на файл, которого уже нет. Символьные ссылки не содержат атрибутов файлов, на которые они указывают. У них есть собственные атрибуты (свое время создания, размер, права доступа).

В системе Linux есть два типа ссылок: **Символьные ссылки (**symbolic links, или soft links) указывают путь к другому файлу. Если удалить файл, на который указывает ссылка (называемая target), ссылка все равно будет существовать, но перестанет «работать», поскольку теперь она указывает на несуществующий файл. Может указывать на файл в другой файловой системе. **Жесткие ссылки (hard links)** указываюn на то же место (inode) на диске, где находится исходный файл. После создания жесткой ссылки она становится неотличимой от оригинального имени файла, то есть как бы становится дополнительным именем файла. После удаления исходного файла жесткая ссылка продолжает существовать. Файл удаляется, когда количество жестких ссылок на него становится равным нулю. Должны указывать на файл в этой же файловой системе.

**Символьные (c) и блочные устройства (b)** Файлы устройств предназначены для обращения к аппаратному обеспечению компьютера (дискам, принтерам, терминалам и др.). Когда происходит обращение к файлу устройства, то ядро операционной системы передает запрос драйверу этого устройства. К символьным устройствам обращение происходит последовательно (символ за символом). Примером символьного устройства может служить терминал. Считывать и записывать информацию на блочные устройства можно в произвольном порядке, причем блоками определенного размера. Пример: жесткий диск.

**Сокеты (s) и каналы (p)** Чтобы понять, что такое канал и сокет, необходимо понимание процессов в операционной системе. И каналы, и сокеты организуют их взаимодействие. Пользователь с данными типами файлов почти не сталкивается. Ключевым отличием канала от сокета является то, что канал однонаправлен. Через канал один процесс всегда передает данные второму, но не наоборот. Сокеты позволяют передавать данные в разных направлениях, т. е. осуществляют связь. Также следует отметить, что канал представлен в структуре каталогов файлом, только если он именован. Когда возникают безымянные каналы, то они существуют только внутри ядра Linux.

**Временные файлы** — это файлы, используемые программами для хранения данных, которые нужны только в течение короткого времени. Это могут быть данные о запущенных процессах, журналы ошибок, промежуточные файлы, используемые при преобразовании данных, файлы кэша и т.д.

**В Linux у каждого файла и каждого каталога есть два владельца: пользователь и группа.** Эти владельцы устанавливаются при создании файла или каталога. Пользователь, который создаёт файл становится владельцем этого файла, а первичная группа, в которую входит этот же пользователь, так же становится владельцем этого файла. Чтобы определить, есть ли у вас как у пользователя права доступа к файлу или каталогу, оболочка проверяет владение ими

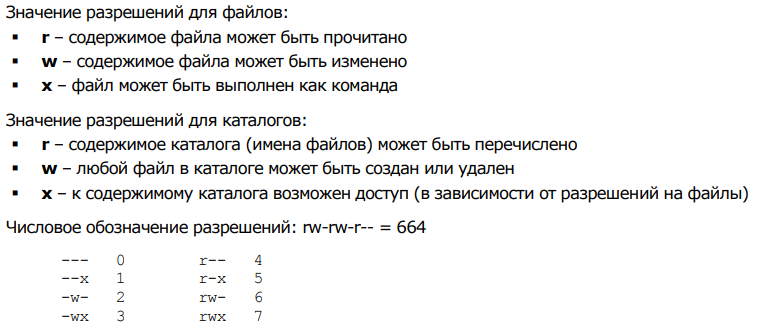
Это происходит в следующем порядке:

1. Оболочка проверяет, являетесь ли вы владельцем файла, к которому вы хотите получить доступ. Если вы являетесь этим владельцем, вы получаете разрешения и оболочка прекращает проверку.

2. Если вы не являетесь владельцем файла, оболочка проверит, являетесь ли вы участником группы, у которой есть разрешения на этот файл. Если вы являетесь участником этой группы, вы получаете доступ к файлу с разрешениями, которые для группы установлены, и оболочка прекратит проверку.

3. Если вы не являетесь ни пользователем, ни владельцем группы, вы получаете права других пользователей (Other).

Команда **chown** позволяет сменить владельца (команда chgrp – только группу-владельца). Имя пользователя-владельца и группы владельца разделяется двоеточием; Если текущий пользователь linda хочет изменить эффективную первичную группу, он будет использовать **команду newgrp**, за которой следует имя группы, которую он хочет установить в качестве новой эффективной первичной группы. После изменения действующей основной группы все новые файлы, созданные пользователем, получат эту группу в качестве группы-владельца. Чтобы вернуться к исходной настройке первичной группы, используйте **exit**

****

Команда **chmod** позволяет изменять разрешения двумя способами:

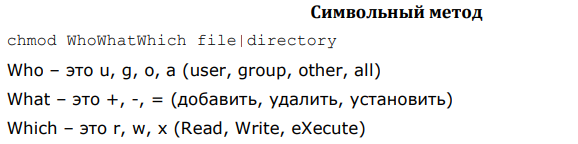
**символьным** chmod a+x cat.txt

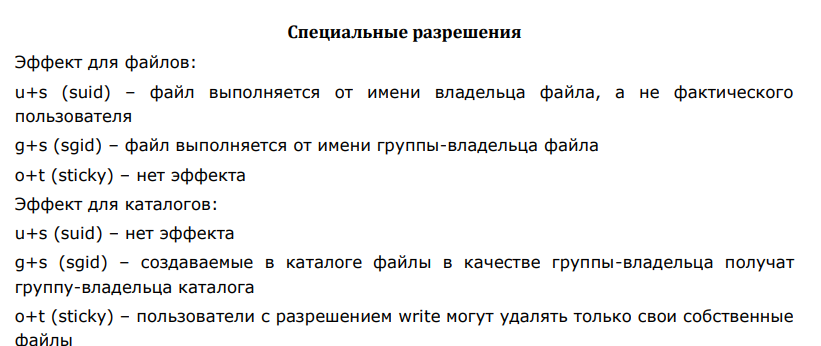
**числовым** chmod 750 animals

Некоторые опции:

-R, --recursive изменять файлы и директории рекурсивно

-v, --verbose выводить диагностику о каждом обрабатываемом файле

-c, --changes выводить диагностику только для изменяемых файлов 



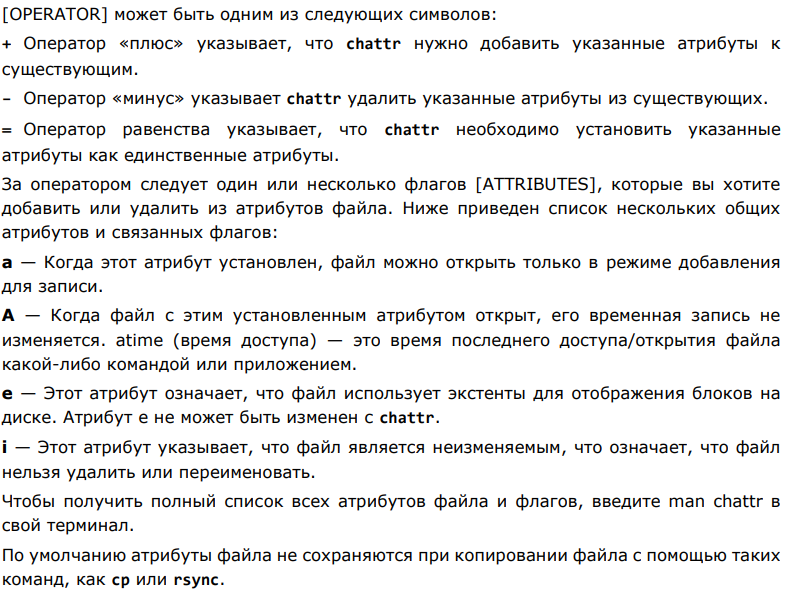
При создании файла или директории, среда операционной системы присваивает им определенные права доступа по умолчанию**. umask - пользовательская маска** (user mask), которая используется для определения конечных прав доступа. После процедуры начальной инсталляции Linux по умолчанию она равна 0022 (022) или 0002 (002).

В операционной системе Linux базовые права для директории равны 0777 (rwxrwxrwx), а для файла 0666 (rw-rw-rw) По умолчанию umask 0002 используется для обычного пользователя. С этой маской права по умолчанию для директории равны 775, а для файла - 664 Для суперпользователя (root) umask по умолчанию равна 0022. С этой маской права по умолчанию для директории равны 755, а для файла – 644.

**Специальное разрешение «sticky bit»**, которое применяется только к каталогам, запрещает пользователям удалять или переименовывать файл в таком каталоге, если они не владеют файлом В выводе команды ls -l каталоги с установленным sticky bit обозначаются буквой t вместо x в последней группе разрешений.

**В Linux атрибуты файла** — это свойства метаданных, которые описывают поведение файла. Например, атрибут может указывать, сжат ли файл, или указывать, можно ли удалить файл. Некоторые атрибуты, такие как неизменяемость, могут быть установлены или очищены, в то время как другие, такие как шифрование, доступны только для чтения и могут быть только просмотрены. Поддержка определенных атрибутов зависит от используемой файловой системы





У каждого файла доступны следующие метки времени:

• Доступ

• Модифицирован

• Изменён

• Создан

Временные метки файла являются частью функциональности файловой системы. Следовательно, на различных файловых системах некоторые временные метки могут быть недоступны.

**Доступ (Access - last access)** — время, когда файл был прочитан последний раз. Это время меняется при доступе таких системных вызовов как mknod(2), utimes(2) и read(2). Если это текстовый файл, то дата последнего доступа обновляется при каждом его открытии. Если это исполнимый файл, то дата доступа обновится при его запуске. При некоторых опциях монтирования диска (noatime или relatime) это значение может быть неточным**. Модифицирован** (Modify - last modified) — время последнего изменения содержимого файла. То есть если это текстовый файл, то время модификации поменяется когда вы его откроете и удалите какое-то слово или что-то допишите. Меняется системными вызовами mknod(2), utimes(2) и write(2). **Изменён** (Change - last changed) — Время последнего изменения метаданных файлов в файловой системе. То есть если в файле изображения вы измените EXIF метаданные — это будет модификация (поскольку по сути поменяется содержимое файла). Примером Изменения файла является смена разрешений доступа к нему (чтение, запись, выполнение), смена владельца, группы и т. д. Меняется с chmod(2), chown(2), link(2), mknod