**Над теорией работают:**

**Настюша, Анечка, Катенька, Майя, Арина, Андрей, Димка Богданов,Димка Кириллов, Никитка, Амир, Саша, Адель**

*пока нас маловато, но за сегодня, надеюсь, нам удастся найти побольше ответственных людей, после чего сможем подсчитать количество билетов на человека. записывайте свои имена, выбирайте цвет выделителя и разбирайте темы.*

**Литл напоминалка для билетиков:**

* выделяйте ключевую информацию *курсивом* или **жирным**
* если делаете списки, ставьте интервалы между абзацами в 1,5, чтобы было более читаемо
* проверяйте инфу и не халявьте: вам может достаться ваш же билет (:

### **Теоретические вопросы к экзамену**

1. Понятие виртуальной сети. Виды виртуальных сетевых компонентов.
2. Трансляция сетевых адресов. Виды NAT.
3. Использование Linux для разработки. Стандартные программные средства.
4. Интерпретатор Python. Использование, версии. Понятие виртуального окружения, настройка, использование.
5. Структура проекта на Python. Организация модулей. Файл зависимостей.
6. Системы контроля версий. Примеры, назначение, общие понятия.
7. Общий алгоритм работы с СКВ Git. Инициализация репозитория, добавление файлов, коммиты.
8. Работа с ветвлением в Git. Назначение веток. Создание, переключение, объединение веток. Разрешение конфликтов слияния.
9. Работа с удаленными репозиториями. Клонирование и форк репозиториев. Отправка и получение изменений в удаленный репозиторий.
10. Современные методологии работы с Git в командном проекте. GitFlow.
11. Понятие сетевого сокета. Применение, виды, схема взаимодействия.
12. Блокирующие операции при обмене через сокеты. Возможные ошибки. Таймауты.
13. Транспортные протоколы TCP и UDP. Принципы работы, сравнение.
14. Клиент-серверное взаимодействие.
15. Реализация сокетов в языке Python. Модуль socket.
16. Понятие программного потока. Процессы и потоки.
17. Асинхронное программирование. Основные понятия. Параллелизм и конкуррентность.
18. Блокирующие и неблокирующие операции.
19. Алгоритмы, ограниченные процессором и вводом-выводом. Основные характеристики, особенности выполнения и распараллеливания.
20. Особенности реализации многопоточности в Python. Модуль threading.
21. Особенности организации многопроцессорной программы в Python. Модуль multiprocessing.
22. Асинхронное программирование в Python. Использование asyncio.
23. Параллельное программирование. Достоинства и недостатки.
24. Понятие потокобезопасности. Причины, проблематика, способы обеспечения.
25. Алгоритм выполнения многопоточной программы. Блокировка потоков.
26. Доступ к общим ресурсам в многопоточной программе. Механизмы блокировки ресурсов модуля threading.
27. Работа с файловой системой в Python. Основные операции.
28. Понятие веб-технологий. Основные характеристики, история, назначение.
29. Программное обеспечение, используемое для веб-технологий. Виды, назначение, примеры.
30. Понятие URL: назначение, применение, состав.
31. Понятие веб-сервера. Цели, принцип работы.
32. Протокол HTTP. Принцип работы, назначение, основные понятия.
33. Настройка веб-сервера. Основные конфигурационные файлы, понятия.
34. Виртуальные хосты. Применение, настройка.
35. Понятие прокси-сервера. Настройка сервера nginx.
36. Основные принципы криптографии. Шифры. Исторические шифры.
37. Симметричное шифрование. Примеры алгоритмов, общая схема, виды.
38. Асимметричное шифрование. Примеры алгоритмов, общая схема, преимущества и недостатки.
39. Алгоритмы хэширования. Примеры, назначение.
40. Протокол TLS/SSL. Общая схема взаимодействия, назначение.
41. Понятие SSL-сертификата. Назначение. Самоподписанные сертификаты. Центры сертификации.

### **Примерные практические задания к экзамену**

1. Напишите программу, которая создает нить. Родительская и вновь созданная нити должны распечатать десять строк текста.

import threading

def output():

for i in range(5, 11):

print(f"{i} строка")

thread1 = threading.Thread(target=output)

thread1.start()

for i in range(5):

print(f"{i} строка")

1. Напишите простой эхо-сервер, использующий неблокирующие сокеты и клиент к нему.

**Сервер**

import socket

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server.bind(('', 8080))

server.listen(1)

conn, addr = server.accept()

server.setblocking(False)

msg = conn.recv(1024).decode()

print(msg)

conn.send(msg.upper().encode())

**Клиент**

import socket

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client.connect(("localhost", 8080))

msg = str(input("Отправленное сообщение: ")).encode()

client.send(msg)

data = client.recv(1024).decode()

print(f"Полученная строка: {data}")

client.close()

1. Напишите простой многопоточный загрузчик URL. Список URL скрипт принимает как аргументы командной строки.

import threading

import time

import urllib2

start = time.time()

urls = ["http://www.google.com", "http://www.apple.com", "http://www.microsoft.com", "http://www.amazon.com",

"<http://www.facebook.com>"]

def fetch\_url(url):

urlHandler = urllib2.urlopen(url)

html = urlHandler.read()

print("'%s\' fetched in %ss" % (url, (time.time() - start)))

threads = [threading.Thread(target=fetch\_url, args=(url,)) for url in urls]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

print("Elapsed Time: %s" % (time.time() - start))

1. Реализуйте простой HTTP-клиент. Он принимает один параметр командной строки - URL. Клиент делает запрос по указанному URL и выдает тело ответа на терминал как текст.

# возможно, это не совсем то, но другого варианта не придумал

import os

url = input("Введите url: ") #<https://www.google.com>

os.system('curl -I ' + str(url))

input()

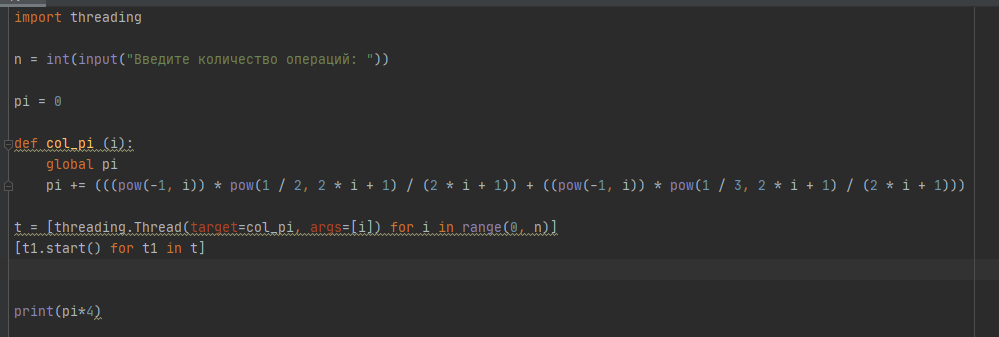
Решение 2

import sys

import os

os.system('curl ' + sys.argv[1])

1. Напишите программу, которая вычисляет число Пи при помощи ряда Эйлера. Количество потоков программы должно определяться параметром командной строки.



1. Дана функция calculate(x, y). Напишите программу, которая создает пул из 5 процессов и распределяет в этом пуле вычисление функции на промежутке х от 0 до 1 с шагом 0,1. у равняется 2 всегда.
2. Напишите программу, которая проверяет все числа от 0 на простоту и выводит простые числа на экран по мере нахождения. Числа должны проверяться в различных потоках (или процессах, по выбору студента) Программа должна работать до тех пор, пока ее не остановит пользователь.

import threading

def IsPrime(n):

k = 0

for i in range(2, n // 2+1):

if (n % i == 0):

k = k+1

if (k <= 0):

print(n, " Число простое")

i = 0

while True:

a = threading.Thread(target=IsPrime, args=(i, )).start()

i += 1

1. Напишите программу, которая обходит все файлы в директории, переданной ей как параметр и выводит на экран имена тех, чей размер задан как второй параметр. Реализовать рекурсивный обход поддиректорий.

import os

path = input("Введите полный путь директории: " + '\n')

size = int(input("Размер файла в байтах: " + '\n'))

def search(path, size):

for i in os.listdir(path):

if os.path.isfile(i) and os.path.getsize(i) == size:

print (i)

#print (i, path + "\\" + i) Если хотите получить формат ФАЙЛ АДРЕСС\_ФАЙЛА

if os.path.isdir(i):

search(path + "\\" + i, size)

search(path, size)

Вариант 2

import os

import sys

def search(path, size):

for i in os.listdir(path):

if os.path.isfile(path + "/" + i) and os.path.getsize(path + "/" + i) == size:

print (i)

if os.path.isdir(path + "/" + i):

search(path + "/" + i, size)

search(sys.argv[1], int(sys.argv[2]))

1. Напишите программу, которая выводит на экран список номеров открытых портов на данной машине. Использовать команду netstat.

import os

os.system('netstat - ano')

input()

# по хорошему надо написать:

netstat –ano| find /i “established”

Вариант 2

import os

os.system('netstat -ano | grep LISTEN')

1. Напишите программу, которая копирует файл с удаленного хоста в текущую папку по SSH. Имя файла и адрес хоста принимать как параметры.

import subprocess

def ssh\_func(FILE,USER\_PATH):

subprocess.run(["scp", FILE, USER\_PATH])

#subprocess.run(["scp", FILE, "USER@SERVER:PATH"])

#e.g. subprocess.run(["scp", "foo.bar", "joe@srvr.net:/path/to/foo.bar"])

**Ответы на теор. часть:**

1. Понятие виртуальной сети. Виды виртуальных сетевых компонентов.

**Виртуальной сетью** называется группа узлов сети, трафик которой, в том числе и широковещательный, на канальном уровне полностью изолирован от других узлов сети (рис. 4.39). Это означает, что передача кадров между разными виртуальными сетями на основании адреса канального уровня невозможна, независимо от типа адреса - уникального, группового или широковещательного. В то же время внутри виртуальной сети кадры передаются по технологии коммутации, то есть только на тот порт, который связан с адресом назначения кадра. Виртуальные сети могут пересекаться, если один или несколько компьютеров входят в состав более чем одной виртуальной сети.

**Виртуальные сетевые компоненты**:

* **Virtual network adapter** (Виртуальный сетевой адаптер) — программный эмулятор сетевой карты типа AMD PCNET PCI, устанавливаемый на гостевую ОС, Каждая ВМ может включать до трех сетевых адаптеров; один адаптер включается в состав ВМ сразу при ее создании, если был разрешен любой из вариантов сетевого подключения.
* **Host virtual adapter** (Виртуальный адаптер хоста) — виртуальный Ethernet-адаптер, устанавливаемый на хостовую ОС при установке VMware Workstation. В хостовой ОС семейства Windows он опознается в качестве сетевого адаптера как VMware Virtual Ethernet Adapter. Это виртуальное устройство служит для взаимодействия ВМ с хост-компьютером и включается в состав ВМ, когда для нее задается тип сетевого подключения **Host Only** или **Network Address Translation**.
* **Bridge** (Мост) — программно реализованный сетевой мост, который позволяет подключать ВМ к реальной локальной сети (Local Area Network, LAN), используя в качестве «посредника» хост-компьютер. Сетевой мост соединяет виртуальный сетевой адаптер с физическим Ethernet-адаптером хост-компьютера. Мост устанавливается во время установки VMware Workstation и включается в конфигурацию ВМ автоматически, если при ее создании был выбран тип сетевого подключения **Bridged Networking**. При необходимости в состав ВМ можно включить дополнительные мосты, если требуется обеспечить ее взаимодействие более чем с одним физическим Ethernet-адаптером.
* **Virtual switches** (Виртуальные коммутаторы) — эти устройства, подобно физическим коммутаторам, обеспечивают соединение между собой различных узлов сети.
* **NAT Device** (Устройство преобразования сетевых адресов) — позволяет подключать ВМ к внешней сети (например, к Интернету), когда ВМ невозможно выделить собственный IP-адрес и приходится использовать IP-адрес, назначенный хост-компьютеру. Компонент **NAT Device** выбирается также в том случае, если требуется обеспечить подключение не к Ethernet-сети, а, например, к сети с архитектурой Token Ring.
* **DHCP Server** (DHCP-сервер) — программный компонент, обеспечивающий назначение сетевых IP-адресов виртуальным машинам в сети, в которой не используются подключения через мост (то есть для подключений типа **Нost Only** или **Network Address Translation**).

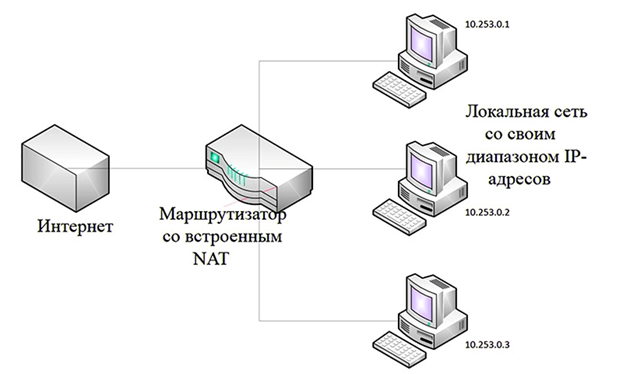
.

2. Трансляция сетевых адресов. Виды NAT.

Трансляция сетевых адресов (NAT) используется многими сервис провайдерами и частными пользователями для решения проблемы нехватки реальных IP-адресов и обеспечения безопасности локальных сетей, подключенных к Интернету. Например, предприятие может иметь выделенный диапазон реальных IP-адресов, но гораздо большее количество компьютеров, имеющих локальные IP-адреса, которым необходим доступ в Интернет. Для решения этой проблемы используется технология трансляции адресов, которая позволяет компьютерам локальной сети взаимодействовать с сетью Интернет, используя всего один внешний реальный IP-адрес. NAT решает эту проблему с помощью подмены локального IP-адреса на наружный общедоступный адрес. Заменяя внутренний IP-адрес и порт на внешний IP-адрес и порт, NAT сохраняет таблицу соответствия, затем при получении ответного пакета производится обратное преобразование.

К локальным IP-адресам относятся следующие диапазоны адресов: 10.ххх.ххх.ххх, 192.168.ххх.ххх, 172.16.ххх.ххх - 172.32.ххх.ххх.

**Схема работы NAT:**



\*\*\*на всякий случай:

Терминология NAT

В соответствии с принятой терминологией NAT внутренняя сеть понимается как набор сетей, которые подлежат терминологии. Под внешней понимают все другие сети.

Чтобы определить, что такое NAT-адрес, нужно учитывать его нахождение в частной сети или в интернете, а также тип трафика (входящий или исходящий). В зависимости от этих факторов устанавливаются следующие четыре обозначения:

* Inside local address (внутренний локальный) — видимый во внутренней сети адрес источника, собственный локальный адрес устройства.
* Inside global address (внутренний глобальный) — видимый из внешней сети адрес источника. При передаче трафика, например, с локального компьютера на веб-сервер, его Inside local address преобразуется маршрутизатором во внутренний глобальный адрес.
* Outside local address (внешний локальный) — видимый из внешней сети адрес получателя. Присвоенный хосту глобально маршрутизируемый адрес IPv4.
* Outside global address (внешний глобальный) — видимый из внутренней сети адрес получателя. Часто совпадает с локальным внешним адресом. \*\*\*

**Типы NAT**

* **Static NAT** — статическая адресная трансляция. Предусматривает сопоставление между глобальными и локальными адресами «один к одному».

Этот тип NAT использует принцип «один к одному» при сопоставлении локальных и глобальных адресов. Настройки сопоставлений, установленные сетевым администратором, остаются неизменными. При отправке сетевыми устройствами трафика в интернет выполняется преобразование их внутренних локальных адресов в настроенные администратором глобальные внутренние адреса. Для внешних сетей устройства, которые работают во внутренней сети со статическим NAT, имеют общедоступные адреса IPv4.

Static NAT Type — это то, что хорошо подходит для веб-серверов, а также для устройств, которым необходим доступный из интернета согласованный адрес.

* **Dynamic NAT** — динамическая адресная трансляция. Сопоставление адресов осуществляется по принципу «многие ко многим».

Тип динамического NAT работает с пулом публичных адресов, которые назначаются на основе принципа «первым пришел, первым обслужен». При запросе внутренним устройством доступа к интернету динамическим NAT производится назначение из пула общедоступного адреса IPv4. Как и в случае со статическим, для динамического типа NAT необходимо достаточное число общедоступных адресов, чтобы удовлетворить совокупное число одновременных пользовательских сеансов.

* **Port Address Translation (NAT Overload)** — трансляция с использованием портов. Предусматривается многоадресное сопоставление.

Это наиболее распространенный тип NAT. При использовании Port Address Translation NAT подключение осуществляет трансляцию нескольких приватных адресов на один или несколько общедоступных. Этот принцип используется в большинстве маршрутизаторов, которые устанавливаются дома у частных абонентов. Адрес назначается провайдером маршрутизатором. При этом одновременный доступ к интернету могут получать несколько домашних пользователей.

Применение **PAT** позволяет сопоставлять несколько адресов с одним или несколькими. Это возможно благодаря отслеживанию каждого приватного адреса по номеру порта. После начала сеанса TCP/IP устройство генерирует номер порта источника TCP или UDP, что позволяет идентифицировать сеанс. При получении пакета от клиента NAT-маршрутизатором, он однозначно идентифицирует перевод NAT, используя номер собственного исходного порта. Схема PAT гарантирует использование разных портов TCP устройствами для каждого сеанса. При поступлении ответа от сервера при этом типе NAT номер порта источника уже используется как номер порта получателя. Это позволяет маршрутизатору однозначно правильно передавать пакеты.

Маршрутизатор выполняет обработку каждого пакета и определяет устройство, с которого он выслан, используя номер порта. Адресом источника (Source Address) в этой схеме будет локальный адрес устройства с добавлением номера порта, который был назначен TCP/IP. Адресом назначения (Destination Address) при использовании TAP будет внешний локальный адрес с добавлением номера служебного порта, например, порта службы 80: HTTP. При ответе веб-сервера производится обратная трансформация адресов.

Между традиционным NAT и PAT есть существенные отличия. NAT осуществляет перевод IPv4-адреса на основе принципа «один к одному» между приватными и общедоступными IPv4-адресами. С другой стороны, PAT трансформирует и адрес, и номер порта. Перенаправление входящих пакетов в NAT интернете выполняется на заданный хостом IP-адрес источника. В схеме PAT предусматривается только один или небольшое число публично открытых адресов IPv4, поэтому перенаправление входящих данных осуществ

3. Использование Linux для разработки. Стандартные программные средства.

В линукс существует множество программ которые могут обеспечить комфортную разработку при использовании множества разных языков программирования. Дабы сузить рассматриваемые рамки, я опишу то, каким примерно должно быть рабочее окружение python.

Стоит отметить что в большинстве линукс дистрибутивов уже встроен компилятор С, и интерпретатор python второй и третьей версии.

При разработке на линукс могут быть полезны:

htop – программа которая позволяет мониторить состояние компьютера, следить за процессами и используемыми ресурсами

curl/wget – программа которая дает возможность скачать файлы из интернета посредством командной строки

build-essential – пакет который содержит служебные файлы необходимые для компиляции программ

git – СКВ, упрощает работу с кодом, и помогает организовать совместную работу.

python3-pip – средство которое помогает устанавливать питоновские библиотеки

Помимо всего этого желательно будет установить специализированный текстовый редактор или IDE. Безусловно можно работать в vim или даже nano. Но специализированные программные средства упрощают работу с кодом. Например:

sublime/atom/vscode – это текстовые редакторы поддерживающие множество языков программирования, в случае чего они еще могут быть улучшены посредством установки плагинов.

Pycharm/CLion и т.д. – пример кроссплатформенных IDE от jetbrains

При разработке на Линукс таже может быть полезно настроить рабочее окружение (verualenv), в случае с питоном, оно помогает установить только те модули которые нужны для определенного проекта, а не пользоваться системным интерпретатором и не нести за собой кучу нунжных модулей.

Стоит отметить что для всех этих действий можно создать сценарий, и при необходимости запустить его, тем самым настроив систему под себя.

4. Интерпретатор Python. Использование, версии. Понятие виртуального окружения, настройка, использование.

**Интерпретатор** - это такой модуль, который исполняет другие программы. Когда вы пишете код на языке Python, интерпретатор Python читает вашу программу и выполняет составляющие ее инструкции. По сути дела *интерпретатор - это слой программной логики между вашим программным кодом и аппаратурой вашего компьютера.*

В зависимости от **версии Python** сам интерпретатор может быть реализован как программа на языке С, как набор классов Java или в каком-либо другом виде.

*Независимо от используемой разновидности Python ваш программный код на этом языке всегда будет выполняться этим интерпретатором.* А чтобы обеспечить такую возможность, вы должны установить интерпретатор Python на свой компьютер.

**Виртуальное окружение** - это изолированное окружение среды (в нашем случае это окружение Python), которое позволяет нам использовать определенные версии приложений.

Все пакеты, работающие с *виртуальным окружением решают одну проблему* - они позволяют проектам, которые имеют различные (и часто конфликтующие) зависимости, сосуществовать на одной системе.

**Виртуальное окружение** автоматически создается в IDE PyCharm.

5. Структура проекта на Python. Организация модулей. Файл зависимостей.

**Что такое модуль?**

Под модулем в *Python* понимается файл с расширением *.py*. Модули предназначены для того, чтобы в них хранить часто используемые функции, классы, константы и т.п. Можно условно разделить модули и программы: программы предназначены для непосредственного запуска, а модули для импортирования их в другие программы. Стоит заметить, что модули могут быть написаны не только на языке *Python*, но и на других языках (например *C*).

Импортировать модуль:

import имя\_модуля

можно импортировать под другим именем:

import имя\_модуля as короткое\_имя

В таких случаях объекты и функции из модуля нужно указывать в виде имя\_модуля.название\_функции

Чтобы избежать этого можно использовать:

from имя\_модуля import название\_функции

**Файл зависимостей.**

В проекте может быть большое количество внешних пакетов. Список этих пакетов принято поставлять вместе с исходным кодом в файле requirements.txt в корне проекта. Это делается для того, чтобы было легче собрать требуемые пакеты для запуска проекта и работы над ним.

Формат файла: по одному пакету на строчке.

Например:

django==1.10.2

pillow==3.3.0

Все пакеты из этого файла можно установить через

pip install -r requirements.txt

6. Системы контроля версий. Примеры, назначение, общие понятия.

Системы контроля версий — это программные инструменты, помогающие командам разработчиков управлять изменениями в исходном коде с течением времени.

Зачем контролировать версии?

1. Вы разрабатываете большой и длительный проект.
2. Большинство работы происходит в текстовых файлах.
3. Вы часто вносите изменения и, бывает, откатываетесь назад.
4. Вам нужно запоминать некоторые состояния (например, гарантированно стабильно работающие, или окончательные).
5. Несколько человек работает одновременно над одним проектом.
6. Вы хотите проводить ревизию изменений другого участника.
7. Вы или другие работаете распределенно и вам нужно облачное хранилище
8. Как Timeshift в Linux.
9. Вы хотите отслеживать, кто вносит какие изменения.
10. Могут происходить конфликты версий

Как СКВ меняют рабочий процесс?

1. Ничего не удаляется. Информация не потеряется.
2. Нужно решить, как часто делать коммиты.
3. Позволяют создавать разные варианты одного документа, т. н. ветки, с общей историей изменений до точки ветвления и с разными — после неё.
4. Дают возможность узнать, кто и когда добавил или изменил конкретный набор строк в файле.
5. Ведут журнал изменений, в который пользователи могут записывать пояснения о том, что и почему они изменили в данной версии.
6. Контролируют права доступа пользователей, разрешая или запрещая чтение или изменение данных, в зависимости от того, кто запрашивает это действие.

Системы контроля версий можно разделить **на две группы:**

* распределенные (позволяют хранить репозиторий (его копию) у каждого разработчика, работающего с данной системой. При этом можно выделить центральный репозиторий (условно), в который будут отправляться изменения из локальных и, с ним же эти локальные репозитории будут синхронизироваться.

Пример: *Git, Mercurial*)

* централизованные (представляют собой приложения типа клиент-сервер, когда репозиторий проекта существует в единственном экземпляре и хранится на сервере. Доступ к нему осуществлялся через специальное клиентское приложение.

Пример: *CVS*, *Subversion*)

Какие есть СКВ?

* Subversion
* Mercurial
* Team Foundation Server
* Git - сегодня самая популярная. Была создана Линусом Торвальдсом, когда его достали человеки. Создана в первую очередь для программистов

Основные понятия СКВ:

1. Рабочая директория - папка, в которой хранятся все файлы проекта и в которой работает СКВ.
2. **Отслеживание файла** - файлы в рабочей директории могут быть добавлены под СКВ или нет. Это сделано для того, чтобы не отслеживать часто меняющиеся, но неважные для рабочего процесса файлы, например: объектные файлы, настройки сборщика и IDE, временные файлы, кэш и так далее.
3. **Состояние** - снимок рабочей директории в определенный момент времени.
4. **Репозиторий** - служебное хранилище СКВ, в котором хранится информация о всех зафиксированных состояниях рабочей директории. Обычно является скрытой подпапкой в рабочей директории, но не входит в нее.
5. **Коммит** - фиксация изменений в новое состояние.
6. **История** - упорядоченная последовательность коммитов конкретного проекта начиная от исходного.
7. **Удаленный репозиторий** - другой репозиторий, часто расположенный на другой машине, с которым можно наладить взаимодействие (синхронизацию). Используется для организации распределенной работы над проектом.
8. **Клонирование** - создание локальной копии удаленного репозитория. Может происходить по протоколу ssh, HTTPS или просто копированием архива.
9. **Апстрим** - обычное название репозитория, служащего исходным для текущего. Обычно обозначается origin.

7. Общий алгоритм работы с СКВ Git. Инициализация репозитория, добавление файлов, коммиты.

Git — система управления версиями с распределенной архитектурой.

С помощью Git можно откатить проект до более старой версии, сравнить, проанализировать или слить свои изменения в репозиторий.

Репозиторием называют хранилище кода и историю его изменений (или коллекцией всех изменений, которые были совершены на протяжении всего времени после инициализации репозитория). Git работает локально и все репозитории хранятся в определенных папках на жестком диске. Так же репозитории можно хранить и в интернете (например, GitHub). В Git каждая рабочая копия кода сама по себе является репозиторием. Это позволяет всем разработчикам хранить историю изменений в полном объеме.

Каждая точка сохранения проекта, фиксирующая изменения, носит название коммит (commit). У каждого commit-a есть hash (уникальный id) и комментарий. Из таких commit-ов собирается ветка. Ветка - это история изменений. У каждой ветки есть свое название. Репозиторий может содержать в себе несколько веток, которые создаются из других веток или вливаются в них.

Общий алгоритм работы с Git:

* Установить и настроить клиент git
* Создать репозиторий в папке с проектом
* Добавить необходимые файлы под СКВ
* Сделать первоначальный коммит
* Сделать логически завершенный участок работы
* Добавить необходимые файлы под СКВ
* Сделать новый коммит с поясняющим сообщением
* Повторить сколько нужно

Основные команды:

***git init*** – команда создает/инициализирует .git репозиторий в проекте.

***git add*** *имяФайла.расширение* – команда фиксирует изменения файла и добавляет файл в “staging area” (участок подготовки – секция, в которой файлы проходят подготовку к перемещению в репозиторий).

***git add*** *.* – добавление всех файлов из директории проекта в staging area.

***git status*** – отображение ранее добавленных файлов в staging area и файлов, которые были изменены и ждут перемещения в staging area.

***git reset*** *имяФайла.расширение* – удаление выбранного файла из staging area.

***git commit -m*** *"Описание коммита"* – фиксирование файлов из staging area в локальный репозиторий. В кавычки следует вставить краткое описание изменений для конкретного коммита.

8. Работа с ветвлением в Git. Назначение веток. Создание, переключение, объединение веток. Разрешение конфликтов слияния.

**Назначение веток:**

* Ветвление происходит, когда мы произвели два разных изменения, основываясь на одном и том же состоянии.
* Часто происходит при командной работе.
* Git позволяет работать с ветками комфортно и, при необходимости, объединять их.
* **Ветвление** - это способ изолировать одни изменения от других. Например, разработка одной фичи независимо от хода работ над другой.

**Создание веток:**

**Ветки** - это специальные указатели на коммиты, которые перемещаются с каждым новым коммитом.

Создать новую ветку можно командой **git branch <name>**

Команда git branch выводит все ветки.

Ветка по умолчанию по соглашению называется master.

Git хранит специальный указатель HEAD - он показывает, на какой ветке вы сейчас находитесь. Чтобы перейти на существующую ветку, вам надо выполнить команду **git checkout.** Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом. Теперь коммиты, которые мы будем делать, будут относиться к новой ветке.

**Переключение на другую ветку:**

Чтобы перейти на существующую ветку, вам надо выполнить команду **git checkout.**

В любой момент можно перейти в другую ветку и начать коммитить в нее. При переходе в другую ветку состояние рабочей директории приводится в соответствие с состоянием последнего коммита в этой ветке. Поэтому все изменения будут основываться на том состоянии.

Переход между ветками откатывает изменения, которые вы делали в другой ветке, но не удаляет их. Все фиксированные изменения сохраняются в истории веток.

При использовании веток история коммитов уже не будет линейной.

**Объединение веток:**

Две ветки можно объединить (слить). Это значит, что мы применяем к файлам изменения, которые произошли и в одной ветке и во второй.

Команда **git merge <branch>** сливает указанную ветку в текущую. Это значит, что создастся новый коммит в текущей ветке, учитывающий изменения в указанной.

* Коммиты слияния имеют двух родителей.
* Git автоматически определяет наилучшего общего предка для слияния веток.
* После слияния ветка не уничтожается и над ней можно продолжать работать.

**Разрешение конфликтов слияния**

Если вы изменили одну и ту же часть файла по-разному в двух ветках, которые собираетесь слить, Git не сможет сделать это чисто. **Такая ситуация называется конфликтом слияния.**

В таком случае, git приостанавливает слияние до тех пор, пока вы не разрешите конфликт.

Посмотреть, какие файлы не прошли слияние можно командой **git status.**

Содержимое спорных файлов в рабочей директории будет отражать оба конфликтующих варианта текста.Чтобы разрешить конфликт, вы должны либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению.После этого необходимо выполнить добавление этих файлов к индексу и коммит для завершения слияния веток.

9. Работа с удаленными репозиториями. Клонирование и форк репозиториев. Отправка и получение изменений в удаленный репозиторий.

**Работа с удаленными репозиториями:**

При командной разработке приходится синхронизировать работу нескольких программистов в одном месте. Даже одному разработчику иногда приходится работать и вносить изменения в проект из разных мест. Для этого в системах контроля версий предусмотрена возможность связать локальный репозиторий с другим - удаленным.

При связывании репозиториев возникает возможность отправить изменения, сделанные в локальном репозитории в удаленный и наоборот - скачать изменения из удаленного репозитория в локальный.

Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду **git remote.** Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы склонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, origin — это имя по умолчанию, которое Git присваивает серверу, с которого вы склонировали.

**Клонирование репозиториев:**

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду **git clone**.  
Вместо того, чтобы просто получить рабочую копию, **Git получает копию практически всех данных, которые есть на сервере.**

Эта команда создаёт директорию “libgit2”, инициализирует в ней поддиректорию .git, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новую директорию libgit2, то увидите в ней файлы проекта, готовые для работы или использования.

В Git реализовано несколько транспортных протоколов, которые вы можете использовать.

**Форк репозиториев:**

Если вы хотите вносить свой вклад в уже существующие проекты, в которых у нас нет прав на внесения изменений путем отправки (push) изменений, **вы можете создать свое собственное ответвление (“fork”) проекта**.

Это означает, что GitHub создаст вашу собственную копию проекта, данная копия будет находиться в вашем пространстве имен и вы сможете легко делать изменения путем отправки (push) изменений.

Таким образом, проекты не обеспокоены тем, чтобы пользователи, которые хотели бы выступать в роли соавторов, имели право на внесение изменений путем их отправки (push). Люди просто могут создавать свои собственные ветвления (fork), вносить туда изменения, а затем отправлять свои внесенные изменения в оригинальный репозиторий проекта путем создания запроса на принятие изменений (Pull Request)

Для того, чтобы создать ответвление проекта (fork), зайдите на страницу проекта и нажмите кнопку “Cоздать ответвление” (“Fork”), которая расположена в правом верхнем углу.

**Отправка изменений в удаленный репозиторий:**

$ git push origin master

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий.

Команда для этого действия: **git push [удал. сервер] [ветка].**

Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонён.

**Получение изменений в удаленный репозиторий:**

Можно использовать команду **git pull** чтобы автоматически получить изменения из удалённой ветви и слить их со своей текущей ветвью.

Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

10. Современные методологии работы с Git в командном проекте. GitFlow.

**Ветви:**

* Старайтесь давать ветвям в git короткие, описательные названия:
* Используйте разнообразные идентификаторы задач (например, задача, созданная в сервисе Github).
* Если в названии несколько слов, для разделения используйте дефис «-».
* Когда несколько человек одновременно работают над одним функционалом, можно создать отдельную ветку на каждого разработчика и общую ветвь, в которой хранятся все изменения, используя соответствующее наименование.   Все изменения из личных веток будут сливаться в общую, которая по завершению функционала будет, в свою очередь, добавлена в master. Если нет необходимости хранить изменения, сделанные в ветке, то после слияния ее необходимо удалить из удаленного репозитория.

**Коммиты:**

* Каждый коммит логически должен представлять собой одно законченное изменение. Не стоит включать несколько таких изменений в один коммит. И наоборот, не стоит разбивать одно логическое изменение на несколько коммитов.
* Постоянно сохраняйте свои изменения. Небольшие атомарные коммиты легче понять и отменить, если что-то пойдет не так.
* Коммиты должны идти в логическом порядке. Если коммит Х зависит от изменений, сделанных в коммите Y, то эти изменения нужно сохранить перед изменениями коммита Х.

**Сообщения к коммитам:**

* Для написания сообщения к коммиту лучше использовать текстовый редактор, а не консоль.  Сохранение изменений в консоли заставляет писать сообщения длинной не более одной строки, что в итоги выливается в неинформативные и неоднозначные сообщения к коммиту.
* Сообщение должно быть описательным и лаконичным. В идеале, оно не должно быть более 50 символов и начинаться с большой буквы.
* После названия необходимо дать более полное описание изменений, сделанных в коммите. При этом название и описание надо разделить пустой строкой. Описание должно быть ограничено 72 символами, кроме того, в нем нужно пояснить, для чего именно были внесены эти изменения, как это изменение решает поставленную задачу и какие побочные эффекты возможны при этом. Кроме того, в описание необходимо включить связанные с коммитом ресурсы (например, ссылку на соответствующую задачу в системе для отслеживания ошибок).
* Иногда первая строка коммита трактуется как тема электронного письма, а остальное - как его тело. Пустая линия, которая разделяет "тему" от "тела" письма необходима, иначе некоторые команды воспримут их как одну.
* Можно использовать ненумерованные списки с точкой, дефисом либо звездочкой в качестве маркера, отделяя при этом пункты пустыми строками.
* Если коммит А зависит от коммита В, то факт этой зависимости должен быть записан в сообщении к коммиту А. Для ссылки на коммиты используйте их хеш. Аналогично, если в коммите А был исправлен баг, допущенный в коммите В, то этот факт также должен быть упомянут в сообщении к коммиту А.

**Слияние изменений:**

* Не переписывайте историю репозитория, если она уже опубликована. По ней можно понять, что произошло на самом деле. Изменение истории репозитория может обернуться большими проблемами для всех, кто с ним работает.
* Не нужно переписывать историю веток.
* Старайтесь поддерживать историю коммитов ясной и простой.
* Если у вас в ветке больше одного коммита, не стоит делать автоматическое слияние.

Git-flow — альтернативная модель ветвления Git, в которой используются функциональные ветки и несколько основных веток. Она предполагает выстраивание строгой модели ветвления вокруг релиза проекта, которая дает надежную схему управления крупными проектами. Gitflow подходит для проектов, которые имеют спланированный цикл релиза.

В соответствии с этой моделью разработчики создают функциональную ветку и откладывают ее слияние с главной магистральной веткой до завершения работы над функцией. Такие долгосрочные функциональные ветки требуют тесного взаимодействия разработчиков при слиянии и создают повышенный риск отклонения от магистральной ветки. В них также могут присутствовать конфликтующие обновления.

Gitflow — это лишь методика работы с Git; в ней определяется, какие виды веток необходимы проекту и как выполнять слияние между ними. Набор инструментов git-flow представляет собой отдельную утилиту командной строки, которая требует установки. Пакеты команд git-flow доступны для многих операционных систем.

Порядок выполнения:

**Ветка разработки и главная ветка**

В этом рабочем процессе для регистрации истории проекта вместо одной ветки main используются две ветки. В главной ветке main хранится официальная история релиза, а ветка разработки develop предназначена для объединения всех функций. Кроме того, для удобства рекомендуется присваивать всем коммитам в ветке main номер версии.

Первый шаг рабочего процесса заключается в создании ветки develop от стандартной ветки main. Разработчику будет проще создать пустую ветку develop локально и отправить ее на сервер.

В этой ветке будет храниться полная история проекта, а в ветке main — сокращенная. Теперь другим разработчикам следует клонировать центральный репозиторий и создать отслеживающую ветку для ветки develop.

**Функциональные ветки (feature)**

Под каждую новую функцию нужно выделить собственную ветку, которую можно отправить в центральный репозиторий для создания резервной копии или совместной работы команды. Ветки feature создаются не на основе main, а на основе develop. Когда работа над функцией завершается, соответствующая ветка сливается с веткой develop. Функции не следует отправлять напрямую в ветку main.

Обратите внимание, что комбинация веток feature с веткой develop фактически представляет собой рабочий процесс с функциональными ветками. Но рабочий процесс Gitflow на этом не заканчивается. Как правило, ветки feature создаются на основе последней ветки develop.

**Ветки выпуска (release)**

Когда в ветке develop оказывается достаточно функций для выпуска (или приближается назначенная дата релиза), от ветки develop создается ветка release. Создание этой ветки запускает следующий цикл релиза, и с этого момента новые функции добавить больше нельзя — допускается лишь исправление багов, создание документации и решение других задач, связанных с релизом. Когда подготовка к поставке завершается, ветка release сливается с main и ей присваивается номер версии. Кроме того, нужно выполнить ее слияние с веткой develop, в которой с момента создания ветки релиза могли возникнуть изменения.

Благодаря тому, что для подготовки выпусков используется специальная ветка, одна команда может дорабатывать текущий выпуск, в то время как другая команда продолжает работу над функциями для следующего. Это также позволяет разграничить этапы разработки.Создание веток release — это еще одна простая операция ветвления. Как и ветки feature, ветки release основаны на ветке develop.

Когда подготовка к поставке завершается, релиз сливается с ветками main и develop, а ветка release удаляется. Важно слить ее с веткой develop, поскольку в ветку release могли добавить критические обновления, которые должны быть доступны для новых функций.

**Ветки исправления (hotfix)**

Ветки сопровождения или исправления (hotfix) используются для быстрого внесения исправлений в рабочие релизы. Ветки hotfix очень похожи на ветки release и feature. Отличие заключается в том, что они создаются на основе main, а не develop. Это единственная ветка, которую нужно обязательно создавать напрямую от main. Как только исправление завершено, эту ветку следует слить с main и develop (или текущей веткой release), а ветке main присвоить обновленный номер версии.

Благодаря специальной ветке для исправления ошибок команда может устранять проблемы, не прерывая остальную часть рабочего процесса и не дожидаясь следующего цикла релиза. По завершении работы с веткой hotfix ее сливают с main.

11. Понятие сетевого сокета. Применение, виды, схема взаимодействия.

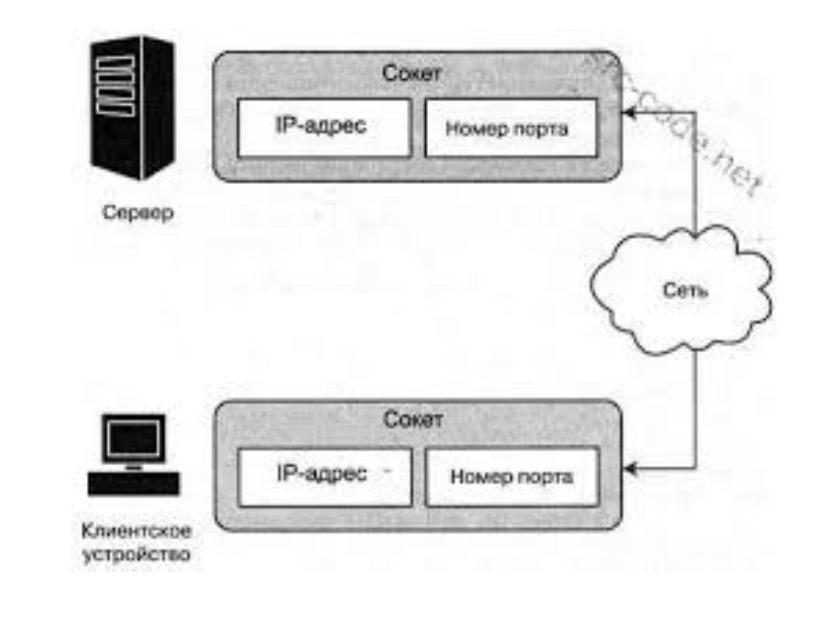
Сетевыми приложениями мы будем называть любые приложения, которые обмениваются данными, используя компьютерную сеть. В настоящее время все более или менее развитые приложения являются сетевыми в той или иной мере.

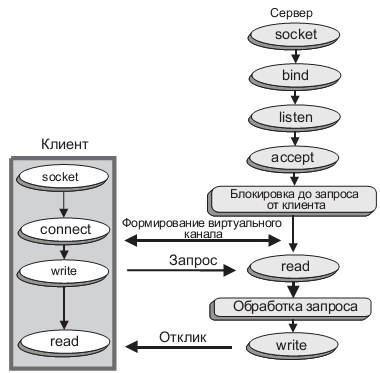
Сетевые приложения могут обмениваться информацией с другими, сторонними приложениями либо строить взаимодействие по сети между компонентами одного и того же приложения. Сетевые приложения используются для обращения к сторонним сервисам, создания публичного сервиса, многопользовательских приложений и т. д.

Со́кет — название программного интерфейса для обеспечения обмена данными между процессами. Сокеты не требуют специального программного обеспечения.

Пара IP-адрес и порт характеризуют сокет (гнездо) - начальную или конечную точку сетевой коммуникации. Для создания соединения нужно два сокета: один на локальной машине, а другой - на удаленной.

Схема:





1. Алгоритм установки соединения:
2. Сначала должен быть создан серверный сокет.
3. Он привязывается к определенному номеру порта и начинает ждать соединения.
4. Затем создается клиентский сокет.
5. Он присоединяется к серверному по адресу хоста и номеру порта
6. После этого устанавливается двунаправленное соединение
7. Используется механизм системных вызовов операционной системы

Виды сокетов:

1. **Потоковые сокеты (stream socket)**

Потоковый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т, е. через эту конечную точку приложение может и передавать, и получать данные.

Потоковый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, сдублированных данных. Потоковый сокет также подходит для передачи больших объемов данных, поскольку накладные расходы, связанные с установлением отдельного соединения для каждого отправляемого сообщения, может оказаться неприемлемым для небольших объемов данных. Потоковые сокеты достигают этого уровня качества за счет использования протокола Transmission Control Protocol (TCP). TCP обеспечивает поступление данных на другую сторону в нужной последовательности и без ошибок.

Для этого типа сокетов путь формируется до начала передачи сообщений. Тем самым гарантируется, что обе участвующие во взаимодействии стороны принимают и отвечают. Если приложение отправляет получателю два сообщения, то гарантируется, что эти сообщения будут получены в той же последовательности.

Однако, отдельные сообщения могут дробиться на пакеты, и способа определить границы записей не существует. При использовании TCP этот протокол берет на себя разбиение передаваемых данных на пакеты соответствующего размера, отправку их в сеть и сборку их на другой стороне. Приложение знает только, что оно отправляет на уровень TCP определенное число байтов и другая сторона получает эти байты. В свою очередь TCP эффективно разбивает эти данные на пакеты подходящего размера, получает эти пакеты на другой стороне, выделяет из них данные и объединяет их вместе.

Потоки базируются на явных соединениях: сокет А запрашивает соединение с сокетом В, а сокет В либо соглашается с запросом на установление соединения, либо отвергает его.

1. **Дейтаграммные сокеты (datagram socket)**

Дейтаграммные сокеты иногда называют сокетами без организации соединений, т. е. никакого явного соединения между ними не устанавливается — сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

Потоковые сокеты по сравнению с дейтаграммными действительно дают более надежный метод, но для некоторых приложений накладные расходы, связанные с установкой явного соединения, неприемлемы (например, сервер времени суток, обеспечивающий синхронизацию времени для своих клиентов). В конце концов на установление надежного соединения с сервером требуется время, которое просто вносит задержки в обслуживание, и задача серверного приложения не выполняется. Для сокращения накладных расходов нужно использовать дейтаграммные сокеты.

Использование дейтаграммных сокетов требует, чтобы передачей данных от клиента к серверу занимался User Datagram Protocol (UDP). В этом протоколе на размер сообщений налагаются некоторые ограничения, и в отличие от потоковых сокетов, умеющих надежно отправлять сообщения серверу-адресату, дейтаграммные сокеты надежность не обеспечивают. Если данные затерялись где-то в сети, сервер не сообщит об ошибках.

1. **Сырые сокеты (raw socket)**

Главная цель использования сырых сокетов состоит в обходе механизма, с помощью которого компьютер обрабатывает TCP/IP. Это достигается обеспечением специальной реализации стека TCP/IP, замещающей механизм, предоставленный стеком TCP/IP в ядре — пакет непосредственно передается приложению и, следовательно, обрабатывается гораздо эффективнее, чем при проходе через главный стек протоколов клиента.

По определению, сырой сокет — это сокет, который принимает пакеты, обходит уровни TCP и UDP в стеке TCP/IP и отправляет их непосредственно приложению.

При использовании таких сокетов пакет не проходит через фильтр TCP/IP, т.е. никак не обрабатывается, и предстает в своей сырой форме. В таком случае обязанность правильно обработать все данные и выполнить такие действия, как удаление заголовков и разбор полей, ложится на получающее приложение — все равно, что включить в приложение небольшой стек TCP/IP.

Однако нечасто может потребоваться программа, работающая с сырыми сокетами. Если вы не пишете системное программное обеспечение или программу, аналогичную анализатору пакетов, вникать в такие детали не придется. Сырые сокеты главным образом используются при разработке специализированных низкоуровневых протокольных приложений. Например, такие разнообразные утилиты TCP/IP, как trace route, ping или arp, используют сырые сокеты.

Работа с сырыми сокетами требует солидного знания базовых протоколов TCP/UDP/IP.

**4. SOCK\_SEQPACKET**

Тип сокета SOCK\_SEQPACKET подобен типу SOCK\_STREAM и также ориентирован на соединение. Обеспечивает работу последовательного двустороннего канала для передачи датаграмм с поддержкой соединений; датаграммы имеют ограниченную длину; от получателя требуется за один раз прочитать целый пакет.

Единственное различие между этими типами заключается в том, что границы записи поддерживаются с использованием типа SOCK\_SEQPACKET. Запись может быть отправлена ​​с использованием одной или нескольких операций вывода и получена с использованием одной или нескольких операций ввода, но одна операция никогда не передает части более чем одной записи. Границы записи видны получателю через флаг MSG\_EOR во флагах полученных сообщений, возвращаемых функцией recvmsg(). Установление максимального размера записи зависит от протокола.

**5. SOCK\_RDM**

Обеспечивает надежную доставку датаграмм без гарантии, что они будут расположены по порядку.

12. Блокирующие операции при обмене через сокеты. Возможные ошибки. Таймауты.

Сокеты могут работать в одном из двух режимов, обычно указывается блокирующим или неблокирующим при помощи функций **fcntl()** или **ioctl()**.

**Блокирующий сокет** не возвращает контроль, пока не отправит или пока не получит все данные, указанные для операции.

На время выполнения операции с блокирующим сокетом программа блокируется. Например, если вы вызвали recv, а данных на вашем конце соединения нет, то в ожидании их прихода ваша программа "засыпает" и тд.

**Неблокирующие сокеты** – инструмент для работы из одного потока с несколькими сокетами сразу или с несколькими операциями над одним сокетом сразу.

Стандартные операции над сокетом, переведенным в неблокирующий режим, никогда не приводят к блокировке. Вместо этого они завершаются со специальным кодом ошибки (**EWOULDBLOCK** или **EAGAIN** в unix), который означает, что операция не может быть выполнена в данный момент. Таким образом, программа не лишается управления при временной невозможности выполнить операцию, а лишь информируется об этом, и может повторить ее успешно позже.

**Таймауты.**

Сокетам можно назначить таймаут для блокировки операций. В Python это делается методом объект\_сокета.settimeout(). В режиме тайм-аута операция завершается неудачно, если она не может быть завершена в течение времени, указанного для сокета или если система возвращает ошибку.

**Возможные ошибки.**

Во время работы с сокетами возможно возникновение ошибок, в частности, в библиотеке socket на Python есть три основных ошибки: socket.timeout, socket.gaierror и socket.herror. Слава богам, что все они наследуются от стандартного Python OSError и в коде их можно перехватывать от except OSError.

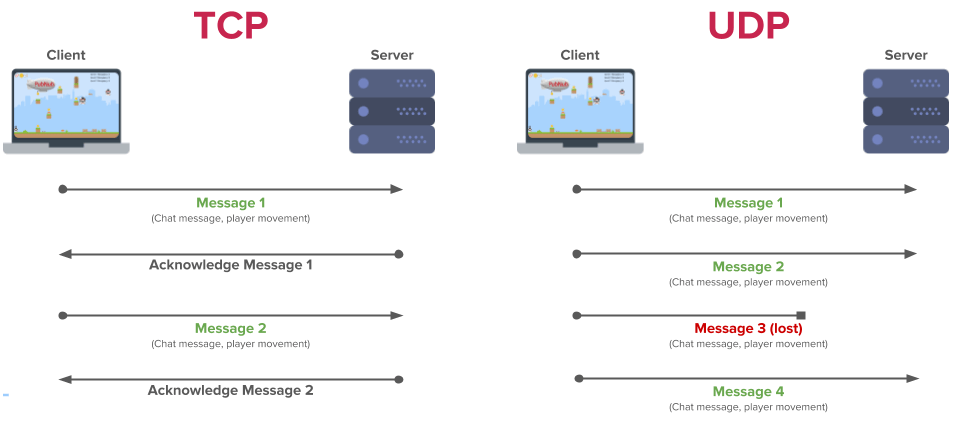
* **socket.herror** [ошибки функций gethostbyname\_ex() и gethostbyaddr()](https://docs-python.ru/standart-library/modul-socket-setevoj-interfejs-python/oshibki-iskljuchenija-opredeljaemye-modulem-socket/#socket.herror),
* **socket.gaierror** [ошибки функций getaddrinfo() и getnameinfo()](https://docs-python.ru/standart-library/modul-socket-setevoj-interfejs-python/oshibki-iskljuchenija-opredeljaemye-modulem-socket/#socket.gaierror),
* **socket.timeout** [ошибки функций settimeout() и setdefaulttimeout()](https://docs-python.ru/standart-library/modul-socket-setevoj-interfejs-python/oshibki-iskljuchenija-opredeljaemye-modulem-socket/#socket.timeout).

**socket.timeout** возникает тогда, когда операция над сокетом не завершилась в течении указанного времени через settimeout.

**socket.gaierror** возникает тогда, когда имя хоста является недействительным (gai расшифровывается как getaddrinfo()

**socket.herror** возникает для ошибок, связанных с адресом хоста

13. Транспортные протоколы TCP и UDP. Принципы работы, сравнение.



**Протокол TCP (Transmission Control Protocol)** – это сетевой протокол, который «заточен» под соединение. Иными словами, прежде, чем начать обмен данными, данному протоколу требуется установить соединение между двумя хостами. Данный протокол имеет высокую надежность, поскольку позволяет не терять данные при передаче, запрашивает подтверждения о получении от принимающей стороны и в случае необходимости отправляет данные повторно. При этом отправляемые пакеты данных сохраняют порядок отправки, то есть можно сказать, что передача данных упорядочена. Минусом данного протокола является относительно низкая скорость передачи данных, за счет того что выполнение надежной и упорядоченной передачи занимает больше времени, чем в альтернативном протоколе UDP.

**Протокол UDP (User Datagram Protocol)**, в свою очередь, более прост. Для передачи данных ему не обязательно устанавливать соединение между отправителем и получателем. Информация передается без предварительной проверки готовности принимающей стороны. Это делает протокол менее надежным – при передаче некоторые фрагменты данных могут теряться. Кроме того, упорядоченность данных не соблюдается – возможен непоследовательный прием данных получателем. Зато скорость передачи данных по данному транспортному протоколу будет более высокой.

**Различия:**

*Надежность:* в этом случае предпочтительнее будет протокол TCP, за счет подтверждения получения данных, повторной отправки в случае необходимости, а также использованию такого инструмента как тайм-аут. Протокол UDP такого инструментария не имеет, а потому при получении отправленные данные могут приходить не полностью;

*Упорядоченность:* опять будет предпочтительнее TCP, поскольку этот протокол гарантирует передачу пакетов данных именно в том порядке, в котором они были отправлены. В случае с UDP такой порядок не соблюдается;

*Скорость:* здесь уже лидировать будет UDP, так как более тяжеловесному TCP-протоколу будет требоваться больше времени для установки соединения, подтверждения получения, повторной отправки данных и т.д. ;

*Метод передачи данных:* в случае с TCP данные передаются потоково, границы фрагментов данных не имеют обозначения. В случае с UDP данные передаются в виде датаграмм – проверка пакетов на целостность осуществляется принимающей стороной только в случае получения сообщения. Также пакеты данных имеют определенные обозначения границ;

TCP применяется там, где требуется точная и подтверждаемая передача данных – например, отправка фотографий, или переписка между пользователями. UDP, в свою очередь, нужен для общения в голосовом формате, или при передаче потокового видео, например, с веб-камер или IP-камер.

14. Клиент-серверное взаимодействие.

**Клиент-серверное взаимодействие** — взаимодействие двух программных продуктов между собой, один из которых выступает в качестве сервера, а другой в качестве клиента. Клиент посылает запрос, а сервер отвечает ему.

**Клиент** — это программная оболочка, с которой взаимодействует пользователь.

**Сервер** — это та часть программного обеспечения, которая выполняет все основные функции (хранит данные, выполняет расчеты).

Пользователь видит программу, которая, допустим, работает с какими-то данными, которые хранятся в базе данных, тем самым он видит всего лишь интерфейс этой программы, а все самое основное выполняет сервер, и процесс когда пользователь оперирует данными через интерфейс программы, при котором клиентская часть взаимодействует с серверной, и называется Клиент-Сервер.

В качестве клиента не обязательно должен выступать интерфейс, который видит пользователь, в некоторых случаях в качестве клиента может выступать и просто программа или скрипт, например, данные на сайте хранятся в базе данных, соответственно скрипты, которые будут обращаться к базе данных и будут являться клиентом в данном случае, хотя и сами эти скрипты являются сервером для клиентской часть сайта (интерфейса).

**Алгоритм взаимодействия:**

1. Сначала должен быть создан серверный сокет.
2. Он привязывается к определенному номеру порта и начинает ждать соединения.
3. Затем создается клиентский сокет.
4. Он присоединяется к серверному по адресу хоста и номеру порта
5. После этого устанавливается двунаправленное соединение
6. Используется механизм системных вызовов операционной системы.

15. Реализация сокетов в языке Python. Модуль socket.

Сокеты в Python реализованы с помощью модуля socket, который обеспечивает доступ к интерфейсу сокета BSD. Он доступен во всех современных ОС. При использовании модуля socket выполняются вызовы API сокетов операционной системы.

Он включает в себя функции создания объекта сокета Socket, который и обрабатывает канал данных, а также функции, связанных с сетевыми задачами, такими как преобразование имени сервера в IP адрес и форматирование данных для отправки по сети.

Интерфейс Python представляет собой прямую трансляцию системного вызова Unix и интерфейса библиотеки для сокетов в объектно-ориентированный стиль Python. Функция socket.socket() возвращает объект Socket, методы которого реализуют различные системные вызовы сокетов.

Сокеты можно настроить для работы в качестве сервера и прослушивания входящих сообщений или для подключения к другим приложениям в качестве клиента. После подключения обоих концов сокета TCP/IP обмен данными становится двунаправленным.

Вызов метода sock.listen(1) переводит сокет в режим сервера, а метод sock.accept() ожидает входящего соединения. Целочисленный аргумент у метода .listen - это количество соединений, которые система должна поставить в очередь в фоновом режиме, прежде чем отклонять новых клиентов. В этом примере предполагается, что одновременно будет работать только одно соединение.

Метод sock.accept() возвращает открытое соединение между сервером и клиентом вместе с адресом клиента. На самом деле соединение представляет собой другой сокет на другом порту (назначенный ядром). Данные считываются из соединения с помощью метода sock.recv() и передаются с помощью sock.send().

Когда общение с клиентом завершено, соединение необходимо очистить с помощью sock.close().

Клиентская программа настраивает свой сокет иначе, чем сервер. Вместо привязки к порту и прослушивания он использует метод sock.connect() для подключения сокета непосредственно к удаленному адресу.

После установления соединения данные могут быть отправлены через сокет с помощью метода sock.send() и получены с помощью sock.recv(), как и на сервере. Когда все сообщения отправлены, а копия получена, то сокет закрывается, чтобы освободить порт.

Клиент и сервер должны запускаться в отдельных окнах терминала, чтобы они могли взаимодействовать друг с другом.

16. Понятие программного потока. Процессы и потоки.

**Программный поток** – поток в котором выполняются задачи программы. Все они выполняются последовательно. С появлением многоядерных процессоров стала общеупотребительной практика распространять нагрузку на все доступные ядра. Существует два основных подхода в распределении нагрузки: использование процессов и потоков. Использование нескольких процессов фактически означает использование нескольких программ, которые выполняются независимо друг от друга.

В Python за это отвечают модули **subprocessing** и **multiprocessing**.

Import multiprocessing

Import os

def foo(n):

print(n, os.getpid())

If \_\_name\_\_ “\_\_main\_\_”:

For n in range(4):

multiprocessing.Process(target=poo, args = n).start()

For I in range(4):

Для увеличения скорости работы программы используются потоки и процессы. Если нужно, чтобы ваше приложение выполняло несколько задач в одно и то же время, то можете воспользоваться потоками (threads). **Потоки** позволяют приложениям выполнять в одно и то же время множество задач. Многопоточность (multi-threading) важна во множестве приложений, от примитивных серверов до современных сложных и ресурсоёмких игр. За многопоточность отвечает модуль **Threading**. Создание отдельного потока:

import Threading

def foo(n):

print(n)

threading.thread(target=foo, args=10).start()

17. Асинхронное программирование. Основные понятия. Параллелизм и конкуррентность.

Прежде чем осмыслить это, читайте вопрос 18. Блокирующие и неблокирующие операции.

**Параллелизм и конкурентность**

**Параллелизм –** когда несколько задач выполняются одновременно(параллельно). Такая форма обычно контролируется системой.

**Конкурентность** – когда несколько задач выполняются **совместно, но не одновременно**. Такую форму использует **асинхронность**. Задача может разбиваться на несколько подзадач.

Параллелизм подразумевает конкурентность (Читайте про GIL, замки, гонки и т.д.).

Но конкурентность не всегда подразумевает параллелизм.

**Асинхронное программирование**

**Асинхронный код** убирает блокирующие операции из основного потока.

**Асинхронные приложения** **производительные** из за того, что приложение берет на себя ответственность переключения задач.

**Асинхронный код –** это набор абстрактных парадигм, которые вместе дают асинхронное (конкурентное) выполнение.

**Асинхронность** достигается с помощью **Event Loop** – цикл событий.

В **Event Loop** кладуться **coroutines (**сопрограммы или корутины), которые выполняются в нем. **Coroutines** способны приостанавливаться в тот момент, когда **Event Loop** прикажет это сделать.

**Task** (задача) – позволяет запускать **coroutine** на фоне.

**Future** (футура) – будущий результат выполнения **coroutine**. Возвращает футуру, значение которой еще не подсчитано. Программа может или подождать(допустим в бесконечном цикле) выполнения футуры или пойти заниматься другими делами. **Future** – наследник **Task**.

**Проблемы** **асинхронного программирования**:

- Синхронный и асинхронный код не живут вместе. Решения:

- Найти в интернете асинхронную реализацию

- Написать самому асинхронную реализацию

- **ThreadPool**

- Не для всего есть асинхронные интерфейсы (допустим открыть файл асинхронно можно). Решение:

- **ThreadPool**

- **CPU-bound** задачи. **Event-loop** кто то должен вращать, если его кто-то заблокирует – встанет все приложение. Решения:

- **Process Pool**

- Отдельный микросервис

- **RPS** может быть ограничен не только сервером (например медленный интернет). Решение только 1 – думать над архитектурой. There is no magic here.

Когда использовать **асинхронный код**:

- Микросервисы (**I/O-bound**, не **CPU-bound**)

- Долгоживущие соединения (**websocket**, раздача файлов)

- Производительная инфраструктура (шардирование базы, кеши, write-heavy очереди)

- Экономия ресурсов серверов

Когда **не** использовать **асинхронный код**:

- **CPU-bound**

- Боттлнек в инфраструктуре (1 инстанс базы), но можно пережить, если нагрузка не упирается в небеса (<1800 RPS), а инстанс выдерживает.

- Вы очень богаты и можете позволить себе сколько угодно железа для сервера

Все дальше относится к asyncio!

Асинхронное программирование – набор парадигм, а asyncio это Python библиотека помогающая их использовать для решения своих проблем.

Что есть **asyncio**?

- Библиотека для написания конкурентных задач в **Python** 3.4+

- Учитывает опыт **Twisted, Tornado, Tulip** (PEP 3156), **greenlet** и прочих, оглядывается на **Curio** и **Trio**

- Привнес синтаксис **async** / **await** (**Python** 3.5+)

- Прослойка между расширениями, работающими на функциях обратного вызова и **async** / **await**

Что такое стандарт **asyncio**?

- Фундамент для асинхронных фреймворков

- Базовые абстракции (**Future/Coroutine/Task/AbstractEventLoop**)

- Высокоуровневый API

- Сопрограммы (**coroutine, generator coroutine**), задачи (**Task**)

- **Streams**, примитивы для синхронизации, **Queues**

- **API** для работы с процессами и межпроцессного взаимодействия

- Низкоуровневый API

- **loop.\*, asyncio.Future**

- Транспорты и протоколы

**Awaitable-объекты**

- Можно использовать в выражении **await**

- Использование объекта в выражении **await** означает, что текущая **сопрограмма** переключит **контекст** и будет ожидать, пока выражение не будет выполнено

- Существует 3 встроенных типа **awaitable** объектов: **coroutine, Task, Future**

Что такое **сопрограмма**?

Методика связи программных модулей друг с другом по принципу кооперативной многозадачности: модуль приостанавливается в определённой точке, сохраняя полное состояние (включая стек вызовов и счётчик команд), и передаёт управление другому. Тот, в свою очередь, выполняет задачу и передаёт управление обратно, сохраняя свои стек и счётчик. © Wikipedia

**Запуск сопрограмм:**

- **asyncio.run()**:

- запуск event loop

- ожидания окончания работы асинхронной функции

- завершение работы event loop и отмена всех порожденных асинхронных задач

- **await** – запуск асинхронного кода с явным переключением контекста

- **asyncio.create\_task()** – запуск задачи в фоновом режиме

**asyncio Coroutine** вPython **–** генератор, а с 3.5+ объект который ведет себя как генератор

- Объект, который имеет ряд инструкций, умеет хранить свое состояние и может переключать контекст (передавать управление)

- Является более обобщенной формой подпрограмм (интерпретатор может выходить в подпрограмму в 1 точке и выходить в другой. В сопрограммах может быть несколько точек входа, выхода и возврата).

**asyncio Coroutine function** в Python:

- Функция, возвращающая объект coroutine

- Определяется ключевыми словами async def

- Может содержать ключевые слова await, async for, async with

**asyncio Task** – сопрограмма, запущенная или запланированая для запуска в цикле событий и контексте:

- Позволяет запускать задачи в фоновом режиме

- Создается с помощью **asyncio.create\_task()** или **loop.create\_task()**, которые оборачивают сопрограмму в объект Task и планирует ее выполнение в цикле событий на ближайшее время

- Абстракция, позволяющая отменить/прервать выполнение сопрограммы с помощью **.cancel()**

**asyncio Future**

- Специальный низкоуровневый **awaitable** объект, представляющий конечный результат выполнения асинхронной операции

- Позволяет использовать низкоуровневый код, реализованный на функциях обратного вызова с высокоуровневым кодом на async/await

- Создается с помощью **loop.create\_future()**

**asyncio.sleep()** приостанавливает действие текущей задачи на

указанное время, позволяя выполнять другие задачи

*Если указан параметр result, его значение возвращается*

*вызывающему объекту по завершении работы сопрограммы*

**asyncio.gather()** запускает указанные **awaitable** объекты в

конкурентном режиме и возвращает результаты выполнения в том же

порядке

Оборачивает объекты **coroutine** в **asyncio Task**

В случае отмены **asyncio.gather()** отменяются все запущенные (но

еще не завершенные) задачи

**asyncio.shield()** защищает awaitable объект от отмены

**asyncio.wait\_for()** ожидает выполнения задачи в течение указанного времени, если задача не успевает выполнится - она отменяется и бросается asyncio.TimeoutError

- Если кто-то отменяет wait\_for(), то и обернутый им awaitable обьект тоже отменяется

- Отмену задачи можно предотвратить с помощью asyncio.shield()

**asyncio.as\_completed()** запускает awaitable объекты, возвращает итератор по результатам в порядке выполнения (сначала - самые быстро вычисленные)

**asyncio.current\_task()** вернет выполняющуюся в данный момент задачу или None

**asyncio.all\_tasks()** вернет все незаконченные задачи, запущенные в цикле событий

**asyncio Policy** (политика)

- Глобальный объект для каждого процесса, отвечает за выбор, настройку и управление циклом событий

- Определяет понятие контекста и управляет отдельным циклом событий для каждого контекста (по умолчанию контекст - текущий поток)

- По умолчанию используется **DefaultEventLoopPolicy**, использует **SelectorEventLoop** на **\*nix** и **ProactorEventLoop** на **Windows**

- Есть альтернативные **WindowsSelectorEventLoopPolicy** и **WindowsProactorEventLoopPolicy**

- Можно получить текущую с помощью **asyncio.get\_event\_loop\_policy()**

- Настраивается с помощью **asyncio.set\_event\_loop\_policy(policy)**

**Цикл событий**

- Ядро любого **asyncio** приложения

- Выполняет асинхронные задачи и функции обратного вызова из очереди, выполняет сетевой **I/O**, управляет выполнением подпроцессов.

**SelectorEventLoop** использует модуль **selectors**, который включает в себя

- SelectSelector

- PollSelector

- EpollSelector

- DevpollSelector

- KqueueSelector

- DefaultSelector

**Умеет**:

- Планировать обратных вызовов

- Открывать сетевые подключения, в т.ч. защищенные (**TLS**)

- Создавать сетевые серверы

- Эффективно передавать файлы (**sendfile**)

- Мониторить файловые дескрипторы

- Напрямую работать с объектами **socket**

- Резолвить **DNS** (в потоках, потому что “unix плох”)

- Обрабатывать сигналы операционной системы (\*nix)

- Выполнять код в пулах потоков или процессов

- Выполнять подпроцессы

Альтернативные циклы: **uvloop**

- Реализован поверх **libuv**, стабильный

- Может дать хороший прирост производительности, если есть очень много сетевого **I/O**

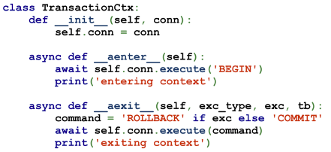
**Асинхронный менеджер контекста:**

- Способен приостановить выполнение в **\_\_aenter\_\_** и **\_\_aexit\_\_** методах

- Как и с обычными менеджерами контекста, можно использовать несколько объектов с оператором async with

- PEP 492

Пример:



**Асинхронные итераторы:**

- Можно вызывать асинхронный код

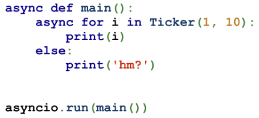
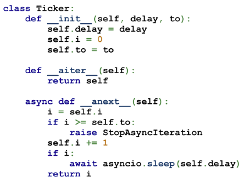
- Итерируемый объект должен реализовывать метод **\_\_aiter\_\_**

- Итератор должен реализовывать асинхронный метод **\_\_anext\_\_**

- По завершении метод **\_\_anext\_\_** должен бросить исключение **StopAsyncIteration**

- PEP 492

Пример:



**Асинхронные генераторы**

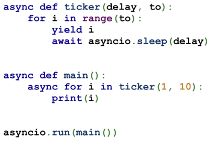
- Асинхронная функция, в которой используется **yield**

- Вместо **send** и **throw** - асинхронные **asend()** и **athrow()**

- Можно использовать с **async for**

- Не поддерживают **yield from**

- PEP 525



**Asynchronous comprehensions**

- В python 3.6+ поддерживаются все compherensions:

- Множетсво set(): {i async for i in agen()}

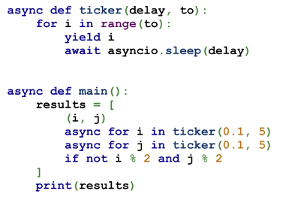
- Список list(): [i async for i in agen()]

- Словарь dict(): {i: i \*\* 2 async for i in agen()}

- Генератор generator(): (i \*\* 2 async for i in agen())

- Можно сочетать с for и условиями if

- PEP 530



18. Блокирующие и неблокирующие операции.

Блокирующие операции — операции, которые не могут быть выполнены полностью локально на компьютере. Для продолжения работы программы необходимо получить ответ от другого компьютера в сети (с которым и выполняется блокирующая операция). Примеры таких операций в Python: accept, connect, recv, send. (необходимо импортировать socket). Рассмотрим подробнее, что происходит во время использования Send: данная операция является блокирующей, если локальный буфер будет заполнен (как следствие, нет места для записи новых данных). В этом случае необходимо подождать, когда информация из буфера будет удалена. Далее можно будет выполнять следующие в программе операции.

Соответственно, неблокирующие (слитно пишется) операции, не прерывают программу по мере ее выполнения.

В синхронных операциях задачи выполняются друг за другом. В асинхронных задачи могут запускаться и завершаться независимо друг от друга. Одна асинхронная задача может запускаться и продолжать выполняться, пока выполнение переходит к новой задаче. Асинхронные задачи не блокируют (не заставляют ждать завершения выполнения задачи) операции и обычно выполняются в фоновом режиме.

Алгоритмы, ограниченные процессором и вводом-выводом. Основные характеристики, особенности выполнения и распараллеливания.

20. Особенности реализации многопоточности в Python. Модуль threading.

Модуль threading является самым основным и простым при построении многопоточных программ. Этот модуль дает возможность создавать новые потоки параллельно основному. Сам модуль threading может помочь, например, с построением сетевых приложений например, можно обслуживать клиента в отдельном потоке.

Многопоточная программа — это программа, которая в момент создания потока как бы “разделяет” программу на две части. Т. Е. главный поток не зависит от другого потока, и если в другом потоке программа висит, то в главном потоке она продолжает выполняться.

К примеру, чтобы создать поток достаточно использовать конструктор Thread из модуля threading, указав target - часть программы (функция) которая будет выполняться в другом потоке. И запустить с помощью метода start() -

t1 = threading.Thread(target = proc)

t1.start()

Одна из особенностей потоков, это что они выполняются не в гарантированном порядке, так например, запустив в цикле 10 потоков мы не сможем точно сказать в каком порядке они будут выполняться. Но с помощью с помощью метода join() можно сделать так чтобы основная программа не продолжала выполняться до завершения потока -

t1.join()

К особенностям threading’a еще можно отнести глобальную блокировку интерпретатора, т. е. threading не может быть использован для ускорения вычислений за счет использования многоядерности. Threading на питоне производит вычисления исключительно на одном ядре.

21. Особенности организации многопроцессорной программы в Python. Модуль multiprocessing.

Многопроцессорность в Python – это способность системы запускать несколько процессов параллельно.

Multiprocessing - это способ повысить производительность путем создания параллельного кода. Производители процессоров делают это возможным, добавляя больше ядер к своим процессорам. В многопроцессорной системе приложения разбиваются на более мелкие подпрограммы для самостоятельной работы.

Общие сведения

* Входит в стандартную библиотеку Python
* Позволяет запускать задачи в разных процессах
* Процессы управляются операционной системой
* Каждый процесс имеет свою копию интерпретатора и всех ресурсов
* Позволяет получить прирост производительности на многоядерных системах

Для использования многопроцессорности необходимо импортировать соответствующий модуль:  
from multiprocessing import Process, Pool

о классе Process

Это абстракция для настройки другого процесса, позволяющая родительскому приложению контролировать выполнение. Здесь мы наблюдаем методы start() и join().

Пример:

import multiprocessing

from multiprocessing import Process

def testing():

print("Объектов")

def square(n):

print("Число квадратов в ",n\*\*2)

def cube(n):

print("Количество кубиков в ",n\*\*3)

if \_\_name\_\_=="\_\_main\_\_":

p1=Process(target=square,args=(7,))

p2=Process(target=cube,args=(7,))

p3=Process(target=testing)

p1.start()

p2.start()

p3.start()

p1.join()

p2.join()

p3.join()

print("Мы закончили")

Process() позволяет нам создать экземпляр класса Process. start() говорит Python начать обработку. Но тогда, если мы позволим этому быть, это потребляет ресурсы, и мы можем исчерпать их в более поздний момент времени. Это потому, что он позволяет процессу оставаться бездействующим и не завершаться. Чтобы избежать этого, мы вызываем метод join(). При этом нам не нужно убивать их вручную. Присоединение останавливает выполнение текущей программы до завершения процесса. Это гарантирует, что программа ожидает завершения p1 и затем завершения p2. Затем он выполняет следующие операторы программы.

Использование замков

Как и многопоточный модуль, многопроцессорная обработка в Python поддерживает блокировки. Процесс включает в себя импорт Lock, его использование, выполнение чего-либо, а затем освобождение.

В следующем фрагменте кода мы заставляем процесс получить блокировку, пока он выполняет свою работу.

from multiprocessing import Process, Lock

lock=Lock()

def printer(item):

lock.acquire()

try:

print(item)

finally:

lock.release()

if \_\_name\_\_=="\_\_main\_\_":

items=['nacho','salsa',7]

for item in items:

p=Process(target=printer,args=(item,))

p.start()

Блокировка не позволяет потокам мешать друг другу. Следующий процесс ожидает снятия блокировки, прежде чем продолжить.

Создание пула процессов Pool

Этот класс представляет пул рабочих процессов; его методы позволяют переложить задачи на такие процессы.

from multiprocessing import Pool

def double(n):

return n\*2

if \_\_name\_\_=='\_\_main\_\_':

nums=[2,3,6]

pool=Pool(processes=3)

print(pool.map(double,nums))

Мы создаем экземпляр пула и создаем процесс с 3 работниками. map() отображает функцию double и повторяемость для каждого процесса.

22. Асинхронное программирование в Python. Использование asyncio.

**Асинхронное программирование** — это особенность современных языков программирования, которая позволяет выполнять операции, не дожидаясь их завершения.

**Asyncio** – это библиотека Python, которая используется для запуска параллельного кода с использованием async / wait.

**Общие сведение про модуль asyncio:**

* Введен в Python с версии 3,5
* Позволяет писать конкурентные корутины
* Включает собственную реализацию блокирующих операций
* Позволяет писать ясный код
* Цикл выполнения контролируется интерпретатором
* Позволяет явно прописывать места переключения потока выполнения
* High-level и low-level API

Основные понятия:

* **корутины** — специальные функции, похожие на генераторы python, от которых ожидают (await), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий
* **task** в asyncio – это объект, который оборачивает coroutine, предоставляя методы для контроля ее выполнения и запроса ее статуса. task может быть создан с помощью asyncio.create\_task() или asyncio.gather().
* В asyncio **event loop** управляет планированием и передачей ожидаемых объектов. Каждая программа asyncio имеет как минимум один event loop. Можно иметь несколько циклов, но в Python 3.7 настоятельно рекомендуется использовать только один.

Асинхронность больше всего подходит для таких сценариев:

* Программа выполняется слишком долго.
* Причина задержки — не вычисления, а ожидания ввода или вывода.
* Задачи, которые включают несколько одновременных операций ввода и вывода.

Это могут быть:

* Парсеры,
* Сетевые сервисы.

Дополнительно:

Awaitable:  
Любой объект, который можно ожидать прерывание своего процесса выполнения, называется awaitable.  
Ключевое слово await приостанавливает выполнение текущей подпрограммы (coroutine) и вызывает указанное ожидание awaitable.  
Пример:  
async def my\_coro():  
 pass

Цикл обработки событий:

* event loop выполняется в потоке;
* получает данные из очереди;
* каждая задача вызывает следующий шаг сопрограммы;
* если сопрограмма вызывает другую сопрограмму (await <имя\_сопрограммы>), текущая сопрограмма приостанавливается, и происходит переключение контекста. Контекст текущей сопрограммы (переменные, состояние) сохраняется и загружается контекст вызванной сопрограммы;
* если сопрограмма встречает блокирующий код (I/O, sleep), текущая сопрограмма приостанавливается, и управление возвращается в event loop;
* event loop получает следующие задачи из очереди 2, ...n;
* затем event loop возвращается к задаче 1, с которой он был прерван.

Параллельное программирование. Достоинства и недостатки.

24. Понятие потокобезопасности. Причины, проблематика, способы обеспечения.

Проблема потокобезопасности проявляется в том что потоки не согласованны друг с другом, и при условном создании потоков в цикле и обращении к общим ресурсам, все ресурсы программы будут использоваться одновременно несколькими потоками. Может случиться так, что какой-нибудь баланс клиента банка ушел бы в минус, так как списание средств происходило бы одновременно после блокирующей операции (например при обращении к базе данных), а при проверке средств, изначально все потоки видели положительный баланс, и не видели никаких преград, но в итоге все обернулось множественным списанием.

Для того чтобы обойти проблему множественного списания и обеспечить потокобезопасность используются замки. Добавив замок (thread.Lock), и метод acquire в функцию потока, мы устанавливаем ограничение, на поток, сделав так чтобы он не выполнялся, в случае если замок закрыт (другие потоки заняты), и только когда замок будет открыт (в нужном месте добавив метод release), потоки продолжат выполняться. Поставив замки мы обеспечиваем потокобезопасность, так как вся работа с ресурсами ставится на замок.

25. Алгоритм выполнения многопоточной программы. Блокировка потоков.

Многопоточная программа выполняется так же, как и все программы, то есть последовательно, но тут следует учесть пару моментов. Так например, в моменте создания нового потока, программа как бы разделяется на две части, и исходный поток работает независимо от того что происходит в дочернем потоке. Потоки не согласованны друг с другом. В этом случае может возникнуть проблема того, что какие-то действия в главном потоке будут выполняться быстрее чем в других потоках, хотя по задумке программы, основной поток должен быть выполнен после всех остальных. Чтобы урегулировать эту ситуацию существует множество разных приемов. Например, в питоновской библиотеке threading у потоков есть метод join(), который блокирует текущий поток, в котором мы находимся, пока не будут завершены дочерние потоки. Также блокировку потоков можно обеспечить посредством использования объекта Lock (thread.Lock()), и в нужном месте прописать lock.aquire, а в конце lock.release(). Но стоит помнить, что абсолютно ВСЕ потоки будут ждать открытия замка в месте его использования, в случае если он закрыт.

Пример простой программы с использованием join:

def proc(n):

print(f“Thread {n}”)

t = [threading.Thread(target = proc, args = [i]) for i in range 5]

[t1.start() for t1 in t]

[t1.join() for t1 in t]

print(“Done”)

26. Доступ к общим ресурсам в многопоточной программе. Механизмы блокировки ресурсов модуля threading.

Для доступа к общим ресурсам программы потоки используют глобальные переменные через ключевое слово global. Существует множество методов для обеспечения блокировки ресурсов, но одним из самых известных, наверное, является блокировка ресурсов посредством замков. Для того чтобы заблокировать общий ресурс, следует объявить замок используя threading.Lock, затем в функции которую мы хотим сделать отдельным потоком следует объявить метод acquire(), затем произвести все необходимые манипуляции с общей переменной, после чего в конце открыть замок с помощью метода release(). В целом, эти два действия можно заменить использовав with, которая еще и помогает в случае чего не забыть открыть замок. Пример простой программы:

count = 0

t\_lock = thread.Lock()

def proc():

global count

with t\_lock:

count += 1

t = [threading.Thread(target = proc) for i in range 5]

[t1.start() for t1 in t]

27. Работа с файловой системой в Python. Основные операции.

в python существует несколько модулей для организации удобной работы с файловой системой.

Самые популярные модули это os и shutil которые предоставляет удобные команды для манипуляции с файлами и директориями

Основные манипуляции с файловой системой это создание, удаление папки и файлов, запись текста в файл, перемещение между папками, просмотр содержимого текстового файла,

копирование файлов, перемещение файлов из одной папки в другую, переименование файлов и просмотр текущей директории.

Модуль os дает возможность создавать файлы и папки, перемещаться между папками, посмотреть текущую директорию, добавлять содержимое в файл и читать содержимое из файла, а также удалять файлы и пустые папки.

os.mkdir – создание директории

os.chdir – смена директории

os.rmdir – удаление директории

os.remove – удаление файла

os.lisdir – содержимое текущей директории

os.getcwd – показывает путь к текущей директории

Когда как модуль shutil обеспечивает перемещение самих файлов между папками, их копирование и переименование, а также он позволяет удалять папки со всеми файлами внутри.

shutil.rmtree – удаление папки со всеми файлами внутри

shutil.copy – копирование файлов

shutil.move – перемещение и переименование файлов

Эти команды являются основными и они обеспечивают базовую работу с файловой системой.

28.Понятие веб-технологий. Основные характеристики, история, назначение.

Web-технологиями является весь набор средств, позволяющих организовать **WWW (World Wide Web)**, то есть всемирную паутину. Так как каждый сеанс является взаимодействием двух сторон, а именно, сервера и клиента, то и Web-технологии делятся на следующие группы: **серверная сторона/ сторона клиента.**

Технологии клиентской стороны включают в свой состав весь набор технологий по созданию веб-страниц **(HTML, JavaScript, DHTML и другие)**, а технологии серверной стороны состоят из технологий доступа к информационным базам данных в сети интернет **(CGI, PHP).**

**Принцип работы/Назначение**

Серверные программы обеспечивают предоставление тех или иных ресурсов клиентским программам. Клиенты, когда им требуется какой-либо файл или просто какая-то информация от сервера, вырабатывают специальный запрос клиента и отправляют его серверу. Серверная программа выполняет обработку запроса и отправляет ответ сервера, который содержит запрошенную информацию или же извещение об ошибке, в случае недоступности требуемых данных. Данная компьютерная организация, или по другому, принципы формирования вычислительных систем или сетей, именуется **архитектурой «клиент-сервер»** или **двухзвенной организацией.** Как раз на базе двухзвенной архитектуры работают практически все интернет - сервисы, включая и WWW.

**Основные понятия:**

Все ресурсы в сети обладают своим адресом, который можно закодировать при посредстве универсального ресурсного идентификатора **URI (Universal Resource Identifier).**

Языком общения компьютерного оборудования является протокол.

Веб –сервер и веб –обозреватель применяют для информационного обмена протокол **HTTP (HyperText Transfer Protocol),** то есть протокол обмена гипертекстом. Это протокол высокого уровня, работающий «сверху» стандартного протокола низкого уровня **TCP/IP (Transfer Control Protocol/Internet Protocol)**, то есть протокол управления обменом/протокол Интернета. Протокол трансляции гипертекстовых данных HTTP служит для пересылки гипертекстовой документации от севера до клиента. Протокол HTTP считается протоколом прикладного уровня.

**История:**

1990 – интернет только в форме протоколов. В конце года создан первый веб сайт (только текст).

1995 – JS, PHP, HTML редактор FrontPage.

1966 – CSS, Macromedia Flash

1999 – Internet Explorer

2002 – Facebook, MySpace (популяризация дизайна веб страниц)

2004 – FireFox – конкурент IE

2007 – Iphone и первый мобильный браузер Safari

2008 – Chrome

2014 – HTML 5

29.Программное обеспечение, используемое для веб-технологий. Виды, назначение, примеры.

30.Понятие URL: назначение, применение, состав.

URL – это механизм, используемый браузерами для получения любого опубликованного во Всемирной сети ресурса.

URL обозначает Uniform Resource Locator. URL это лишь адрес, который выдан уникальному ресурсу в интернете. В теории, каждый корректный URL ведёт на уникальный ресурс. Такими ресурсами могут быть HTML-страница, CSS-файл, изображение и т.д. На практике, существуют некоторые исключения, когда, например, URL ведёт на ресурс, который больше не существует или который был перемещён. Поскольку ресурс, доступный по URL, а также сам URL обрабатываются веб-сервером, его владелец должен внимательно следить за размещаемыми ресурсами и связанными с ними URL. URL используется как стандарт записи ссылок на объекты в Интернете (Гипертекстовые ссылки во «всемирной паутине» www).

Состав URL:

URL состоит из различных частей, некоторые из которых являются обязательными, а некоторые - факультативными.

На примере: *http://\_www.example.com\_:80\_/path/to/myfile.html\_?key1=value1&key2=value2\_#SomewhereInTheDocument*

**Protocol**

*http://* это протокол. Он отображает, какой протокол браузер должен использовать. Обычно это HTTP-протокол или его безопасная версия - HTTPS. Интернет требует эти 2 протокола, но браузеры часто могут использовать и другие протоколы, например mailto: (чтобы открыть почтовый клиент) или ftp: для запуска передачи файлов, так что не стоит удивляться, если вы вдруг увидите другие протоколы.

**Domaine Name**

*www.example.com* это доменное имя. Оно означает, какой веб-сервер должен быть запрошен. В качестве альтернативы может быть использован и IP-адрес, но это делается редко, поскольку запоминать IP сложнее, и это не популярно в интернете.

**Port**

:*80* это порт. Он отображает технический параметр, используемый для доступа к ресурсам на веб-сервере. Обычно подразумевается, что веб-сервер использует стандартные порты HTTP-протокола (80 для HTTP и 443 для HTTPS) для доступа к своим ресурсам. В любом случае, порт - это факультативная составная часть URL.

**Path to the file**

*/path/to/myfile.html* это адрес ресурса на веб-сервере. В прошлом, адрес отображал местоположение реального файла в реальной директории на веб-сервере. В наши дни это чаще всего абстракция, позволяющая обрабатывать адреса и отображать тот или иной контент из баз данных.

**Parameters**

*?key1=value1&key2=value2* это дополнительные параметры, которые браузер сообщает веб-серверу. Эти параметры - список пар ключ/значение, которые разделены символом &. Веб-сервер может использовать эти параметры для исполнения дополнительных команд перед тем как отдать ресурс. Каждый веб-сервер имеет свои собственные правила обработки этих параметров и узнать их можно, только спросив владельца сервера.

**Anchor**

#*SomewhereInTheDocument* это якорь на другую часть того же самого ресурса. Якорь представляет собой вид "закладки" внутри ресурса, которая переадресовывает браузер на "заложенную" часть ресурса. В HTML-документе, например, браузер может переместиться в точку, где установлен якорь; в видео- или аудио-документе браузер может перейти к времени, на которое ссылается якорь. Важно отметить, что часть URL после #, которая также известна как идентификатор фрагмента, никогда не посылается на сервер вместе с запросом.

Понятие веб-сервера. Цели, принцип работы.

Протокол HTTP. Принцип работы, назначение, основные понятия.

Настройка веб-сервера. Основные конфигурационные файлы, понятия.

Виртуальные хосты. Применение, настройка.

Понятие прокси-сервера. Настройка сервера nginx.



# 31. Понятие веб-сервера. Цели, принцип работы.

**Понятия:**

* С точки зрения железа Веб-сервер — это компьютер, который хранит ресурсы сайта (HTML документы, CSS стили, JavaScript файлы и другое) и доставляет их на устройство конечного пользователя (веб-браузер и т.д.). Обычно он подключен к сети Интернет и может быть доступен через доменное имя, например, mozilla.org.
* Веб-сервер — сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными. ... Ресурсы — это HTML-страницы, изображения, файлы, медиа-потоки или другие данные, которые необходимы клиенту.

**Цель** веб-сервера проста - обслуживать одновременно большое количество клиентов, максимально эффективно используя hardware.

**Главная задача** веб сервера принимать HTTP-запросы от пользователей, обрабатывать их, переводить в цифровой компьютерный код. Затем выдавать HTTP-ответы, преобразуя их из миллионов нолей и единичек в изображения, медиа-потоки, буквы, HTML страницы.

Любой веб сервер, для удобства его использования пользователями, должен иметь удобный веб-браузер. Он передает веб серверу запросы, преобразованные в URL-адреса интернет - ресурсов.

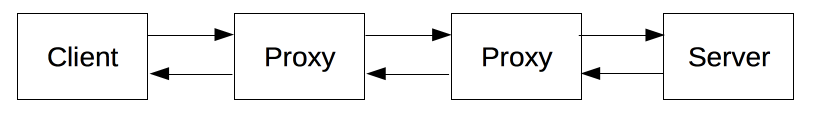
На обычном компьютере, в большинстве случаев, это работает примерно так. ОС (точнее драйвер сетевой карты) получает аппаратное прерывание, когда приходят данные на карту. Прерывания будут приходить не на каждый кусок данных полученных картой, а только тогда, когда софт разрешит себя прерывать. То есть: получил прерывание, запретил прерывания, начал обрабатывать все что пришло и приходит, закончил, разрешил прерывания. В зависимости от системы или ее настроек, драйвер уже может что-то сделать с данными. Намеренно не употребляю слово "пакет", ибо это пока только данные, лежащие в буфере. Потом после всех проверок, они могут быть переданы сетевому стеку OS. Сетевой стэк, собственно как и драйвер, как правило является модулем ядра. На каждом уровне, в зависимости от его настройки и/или типа данных может приниматься решение, что с ним делать дальше. Серверы могут его тоже обрабатывать по разному. Apache например, по крайней мере ранние его версии, выделяли каждому соединению по потоку, загружали туда что-то, что обработает этот запрос, и уходили слушать дальше. Но сильно-нагруженные серверы, так работать не смогут, ибо потоки много "весят", поэтому есть еще куча, более продвинутых, способов обработки запросов сервером.

# 32. Протокол HTTP. Принцип работы, назначение, основные понятия.

# **HTTP** — это протокол, позволяющий получать различные ресурсы, например HTML-документы. Протокол HTTP лежит в основе обмена данными в Интернете. HTTP является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно веб-браузером (web-browser). Полученный итоговый документ будет (может) состоять из различных поддокументов, являющихся частью итогового документа: например, из отдельно полученного текста, описания структуры документа, изображений, видео-файлов, скриптов и многого другого.

HTTP — это клиент-серверный протокол, то есть запросы отправляются какой-то одной стороной — участником обмена (user-agent) (либо прокси вместо него). Чаще всего в качестве участника выступает веб-браузер, но им может быть кто угодно, например, робот, путешествующий по Сети для пополнения и обновления данных индексации веб-страниц для поисковых систем.

Каждый запрос (англ. *request*) отправляется серверу, который обрабатывает его и возвращает ответ (англ. *response*). Между этими запросами и ответами как правило существуют многочисленные посредники, называемые прокси, которые выполняют различные операции и работают как шлюзы или кэш, например.



Обычно между браузером и сервером гораздо больше различных устройств-посредников, которые играют какую-либо роль в обработке запроса: маршрутизаторы, модемы и так далее. Благодаря тому, что Сеть построена на основе системы уровней (слоёв) взаимодействия, эти посредники "спрятаны" на сетевом и транспортном уровнях. В этой системе уровней HTTP занимает самый верхний уровень, который называется "прикладным" (или "уровнем приложений"). Знания об уровнях сети, таких как представительский, сеансовый, транспортный, сетевой, канальный и физический, имеют важное значение для понимания работы сети и диагностики возможных проблем, но не требуются для описания и понимания HTTP.

Понятия:

Участник обмена (user agent) — это любой инструмент или устройство, действующие от лица пользователя. Эту задачу преимущественно выполняет веб-браузер; в некоторых случаях участниками выступают программы, которые используются инженерами и веб-разработчиками для отладки своих приложений.

Прокси - Между веб-браузером и сервером находятся большое количество сетевых узлов, передающих HTTP сообщения. Из-за слоистой структуры большинство из них оперируют также на транспортном сетевом или физическом уровнях, становясь прозрачным на HTTP слое и потенциально снижая производительность. Эти операции на уровне приложений называются **прокси**. Они могут быть прозрачными или нет, (изменяющие запросы не пройдут через них), и способны исполнять множество функций:

* caching (кеш может быть публичным или приватными, как кеш браузера)
* фильтрация (как сканирование антивируса, родительский контроль, …)
* выравнивание нагрузки (позволить нескольким серверам обслуживать разные запросы)
* аутентификация (контролировать доступом к разным ресурсам)
* протоколирование (разрешение на хранение истории операций)

Веб-сервер - **(ПОНЯТИЕ 31 ВОПРОС)**

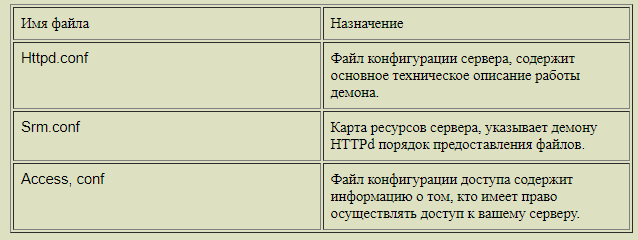
На другой стороне коммуникационного канала расположен сервер, который обслуживает (англ. *serve*) пользователя, предоставляя ему документы по запросу. С точки зрения конечного пользователя, сервер всегда является некой одной виртуальной машиной, полностью или частично генерирующей документ, хотя фактически он может быть группой серверов, между которыми балансируется нагрузка, то есть перераспределяются запросы различных пользователей, либо сложным программным обеспечением, опрашивающим другие компьютеры (такие как кеширующие серверы, серверы баз данных, серверы приложений электронной коммерции и другие).

Сервер не обязательно расположен на одной машине, и наоборот - несколько серверов могут быть расположены (поститься) на одной и той же машине. В соответствии с версией HTTP/1.1 и имея Host заголовок, они даже могут делить тот же самый IP-адрес.

# 33. Настройка веб-сервера. Основные конфигурационные файлы, понятия.

Сервер Apache имеет три файла конфигурации, они находятся в каталоге **/usr/local/etc/httpd/conf.** Эти файлы позволяют настроить все стороны функционирования сервера. После того как вы отредактируете эти файлы в соответствии с вашими требованиями, можете запускать сервер. Никакой другой настройки не требуется.

Прежде всего нужно перейти в каталог, содержащий файлы конфигурации, **/usr/local/etc/httpd/conf.** Здесь вы найдете три файла конфигурации.



Файл **httpd.conf** содержит конфигурацию сервера. Откройте его в редакторе и измените указанные ниже строки, если это необходимо. Приведенные ниже описания послужат вам в качестве указаний. Они не являются исчерпывающими инструкциями, однако затрагивают те изменения, которые вам, возможно, придется внести, чтобы заставить сервер работать. Единственной строкой, которую вы *обязаны,* изменить, является строка, начинающаяся с ⌠ServerAdmin■. Остальным директивам по умолчанию присвоены разумные значения, поэтому взгляните на них, чтобы убедиться в правильности значений, но вносить в них изменения вряд ли потребуется.

[Ссылка для Apache](https://www.opennet.ru/docs/RUS/apache/)

# 

# 34. Виртуальные хосты. Применение, настройка.

# 

Первоначальный виртуальный хост начинался с цели размещения на одной машине большего количества веб-сайтов (например, website1.example.com, website2.example.com и т. Д.). Это также означает совместное использование ресурсов одной машины, таких как память и процессор. Ресурсы распределяются и используются таким образом, что достигается максимальная эффективность.

Теперь, с изобретением облачных вычислений, виртуальный теперь служит больше, чем когда-либо, таким как такие решения, как хостинг виртуальных приложений, хостинг виртуальных серверов, хостинг виртуальных хранилищ, а иногда и виртуальный / весь хостинг центров обработки данных.

Существует много способов настройки виртуального хоста, и большинство способов, которые используются сегодня, приведены ниже:

* На основе IP
* Порт-Based
* На основании имени

#### **На основе IP:**

Это один из самых простых методов среди трех, и его можно использовать для применения различных директив на основе IP-адреса. В виртуальном хостинге на основе IP мы используем разные IP для каждого домена.

Несколько IP-адресов будут фактически указывать на уникальные домены на сервере, и для сервера будет только один IP-адрес.

Этот виртуальный хостинг достигается путем создания нескольких IP-адресов для одного сервера.

**Порт-Based:**

Виртуальный хостинг на основе портов также похож на виртуальный хостинг на основе IP, разница в том, что вместо использования разных IP-адресов для каждого из виртуальных хостов мы используем разные порты, в которых серверы настроены для ответа на несколько веб-сайтов, которые зависит от порта сервера.

#### **На основании имени:**

Виртуальные хосты на основе имен - это наиболее распространенная и часто используемая техника виртуального хостинга. Виртуальный хост на основе имени будет использовать один IP-адрес для всех доменов на данном сервере. Когда браузер пытается подключиться к серверу, он отправит на сервер сообщение с информацией о доменном имени, к которому он пытается подключиться. Когда указано доменное имя, сервер проверяет конфигурацию хоста и тем самым возвращает запрос с правильным веб-сайтом.

# 35. Понятие прокси-сервера. Настройка сервера nginx.

**Прокси-сервер** — промежуточный сервер (комплекс программ) в компьютерных сетях, выполняющий роль посредника между пользователем и целевым сервером (при этом о посредничестве могут как знать, так и не знать обе стороны), позволяющий клиентам как выполнять косвенные запросы (принимая и передавая их через прокси-сервер) к другим сетевым службам, так и получать ответы. Сначала клиент подключается к прокси-серверу и запрашивает какой-либо ресурс (например e-mail), расположенный на другом сервере. Затем прокси-сервер либо подключается к указанному серверу и получает ресурс у него, либо возвращает ресурс из собственного кэша (в случаях, если прокси имеет свой кэш). В некоторых случаях запрос клиента или ответ сервера может быть изменён прокси-сервером в определённых целях. Прокси-сервер позволяет защищать компьютер клиента от некоторых сетевых атак и помогает сохранять анонимность клиента, но также может использоваться мошенниками для скрытия адреса сайта, уличённого в мошенничестве, изменения содержимого целевого сайта (подмена), а также перехвата запросов самого пользователя.

Настройка Nginx:

Рассмотрим главный конфигурационный файл nginx — /etc/nginx/nginx.conf.

**Как выглядит можно не писать**

По умолчанию он выглядит следующим образом:

user www-data;

worker\_processes auto;

pid /run/nginx.pid;

include /etc/nginx/modules-enabled/\*.conf;

events {

worker\_connections 768;

}

http {

sendfile on;

tcp\_nopush on;

tcp\_nodelay on;

keepalive\_timeout 65;

types\_hash\_max\_size 2048;

include /etc/nginx/mime.types;

default\_type application/octet-stream;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;

ssl\_prefer\_server\_ciphers on;

access\_log /var/log/nginx/access.log;

error\_log /var/log/nginx/error.log;

gzip on;

include /etc/nginx/conf.d/\*.conf;

include /etc/nginx/sites-enabled/\*;

}

Существует два вида директив – простые и блочные. Простая директива состоит из имени и параметров, разделённых пробелами, и в конце строки ставится точкой с запятой (;). Блочная директива устроена так же, как и простая директива, но вместо точки с запятой после имени и параметров следует набор дополнительных инструкций, помещённых внутри фигурных скобок ({ и }). Рассмотрим те, которые пригодятся нам для примера:

* user – пользователь, от имени которого работает nginx, здесь это www-data;
* worker\_processes – количество процессов сервера, значение выставляется равным количеству ядер процессора, auto – сервер определит автоматически;
* pid – файл, внутри которого хранится идентификатор запущенного главного процесса (PID);
* include – подключаемый файл или файлы конфигурации;
* events – блок директив, определяющих работу с сетевыми соединениями;
* worker\_connections – максимальное количество одновременных соединений;
* http – блок директив http сервера;
* sendfile – метод отправки данных, включаем значением on;
* tcp\_nopush и tcp\_nodelay – параметры, положительно влияющие на производительность, оставляем значение on;
* keepalive\_timeout – время ожидания keepalive соединения до его разрыва со стороны сервера;
* types\_hash\_max\_size – регламентирует максимальный размер хэш таблиц типов;
* default\_type – указывает тип MIME ответа по умолчанию;
* ssl\_protocols – включает указанные протоколы;
* ssl\_prefer\_server\_ciphers – указывает, что серверное шифрование; предпочтительнее клиентского, при использовании SSLv3 и TLS протоколов;
* access\_log – задает путь к файлу лога доступа, при выставлении значения в off, запись в журнал доступа будет отключена;
* error\_log – путь к журналу регистрации ошибок;
* gzip – при помощи этой директивы можно включать или отключать сжатие.

В конфигурационных файлах nginx допустимо пользоваться встроенными переменными. Преимущественно это переменные, представляющие собой поля заголовка запроса клиента, такие как $remote\_addr, $server\_name. Все переменные начинаются со знака $, с полным перечнем можно ознакомиться в документации, на официальном сайте.

36.Основные принципы криптографии. Шифры. Исторические шифры.

Задачи:

* Обеспечение секретности
* Аутентификация сторон
* Обеспечение целостности

Шифр – совокупность заранее оговоренных способов преобразования исходного секретного сообщения с целью его защиты. Исходные сообщения обычно называют открытыми текстами.Сообщение, полученное после преобразования с использованием любого шифра, называется шифрованным сообщением (закрытым текстом, криптограммой).Преобразование открытого текста в криптограмму называется зашифрованием. Обратное действие называется расшифрованием. Ключ – информация, необходимая для шифрования и расшифрования сообщений.

Информация в процессе хранения, передачи и преобразования подвергается воздействию различных атак. Основными нарушениями безопасности являются раскрытие информационных ценностей (потеря конфиденциальности) , модификация без разрешения автора (потеря целостности) или неавторизованная потеря доступа к этим ценностям (потеря доступности) .

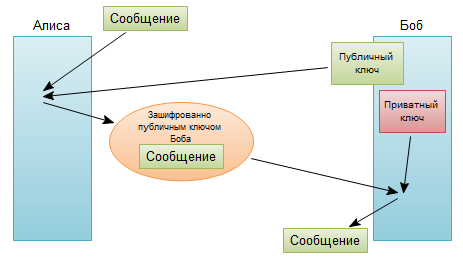
Виды шифрования:

* Симметричное шифрование (шифрование с закрытым ключом)
* Асимметричное шифрование (шифрование с открытым ключом)
* Электронные подписи (MAC, HMAC)
* Хэширование (MD5, SHA1, SHA256)

Если способ шифрования подразумевает, что как шифрование, так и расшифровка ведутся одним и тем же ключом, то такой шифр называется симметричным.

Если алгоритм предусматривает шифровку одним ключом, а расшифровку – другим, то такие шифры называю асимметричными.

Шифрование публичным ключом (асимметричное шифрование)



Алгоритм шифрования, применяющийся сегодня буквально во всех компьютерных системах. Есть два ключа: открытый и секретный. Открытый ключ — это большое число, имеющее только два делителя, помимо единицы и самого себя. Эти два делителя являются секретным ключом, и при перемножении дают публичный ключ. Например, публичный ключ — это 1961, а секретный — 37 и 53.

Открытый ключ используется, чтобы зашифровать сообщение, а секретный — чтобы расшифровать.

Исторические шифры

Шифр сдвига, шифр Юлия Цезаря

Каждая буква сообщения заменяется на другую, которая в русском алфавите отстоит от исходной на три позиции дальше. Таким образом, буква A заменяется на Г , Б на Д и так далее вплоть до буквы Ь , которая заменялась на Я , затем Э на A , Ю на Б и, наконец, Я на В . Предположим, что буквы сдвигаются не на три знака вправо, а на n (0<n<33). В этом случае в системе шифрования появляется ключ – число n – параметр сдвига.

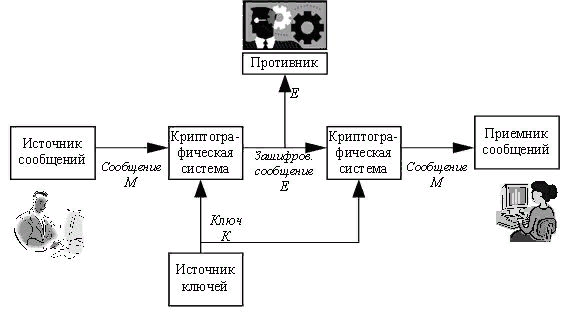
Шифр замены

Основной недостаток шифра сдвига заключается в том, что есть всего 25 возможных значений ключа. Для того чтобы воспользоваться этим алгоритмом, создается таблица с исходным алфавитом и, непосредственно под ним, тот же алфавит, но с переставленными буквами (или любой другой набор знаков.

# 37.Симметричное шифрование. Примеры алгоритмов, общая схема, виды.

**Симметричная схема шифрования** - криптографический примитив, определяемый следующими тремя компонентами: алгоритм **шифрования**, секретный ключ, алгоритм расшифрования. Долгое время **симметричные** схемы были единственным средством обеспечения конфиденциальности информации. **Шифрование** и расшифрование в такой схеме происходит с использованием одного и того же ключа.

**Общая схема симметричного шифрования:**

Классическая, или одноключевая криптография опирается на использование **симметричных алгоритмов шифрования**, в которых шифрование и расшифрование отличаются только порядком выполнения и направлением некоторых шагов. Эти алгоритмы используют один и тот же секретный элемент (ключ), и второе действие (расшифрование) является простым обращением первого (шифрования). Поэтому обычно каждый из участников обмена может как зашифровать, так и расшифровать сообщение. Схематичная структура такой системы представлена на рисунке:

На передающей стороне имеются источник сообщений и источник ключей. Источник ключей выбирает конкретный ключ К среди всех возможных ключей данной системы. Этот ключ К передается некоторым способом принимающей стороне, причем предполагается, что его нельзя перехватить, например, ключ передается специальным курьером (поэтому симметричное шифрование называется также шифрованием с *закрытым ключом*). Источник сообщений формирует некоторое сообщение М, которое затем шифруется с использованием выбранного ключа. В результате процедуры шифрования получается зашифрованное сообщение Е (называемое также криптограммой). Далее криптограмма Е передается по каналу связи. Так как канал связи является открытым, незащищенным, например, радиоканал или компьютерная сеть, то передаваемое сообщение может быть перехвачено противником. На принимающей стороне криптограмму Е с помощью ключа расшифровывают и получают исходное сообщение М.

Если М – сообщение, К – ключ, а Е – зашифрованное сообщение, то можно записать в виде: E=f(M,K)

то есть зашифрованное сообщение Е является некоторой функцией от исходного сообщения М и ключа К. Используемый в криптографической системе метод или алгоритм шифрования и определяет функцию f в приведенной выше формуле.

**В зависимости от принципа работы алгоритмы симметричного шифрования делятся на два типа**:

* *блочные;*
* *потоковые*.

***Блочные* алгоритмы** шифруют данные блоками фиксированной длины (64, 128 или другое количество бит в зависимости от алгоритма). Если все сообщение или его финальная часть меньше размера блока, система дополняет его предусмотренными алгоритмом символами, которые так и называются *дополнением*.

**К актуальным блочным алгоритмам относятся**:

* [AES](https://encyclopedia.kaspersky.ru/glossary/aes/)
* ГОСТ 28147-89
* RC5
* Blowfish
* Twofish

**Потоковое шифрование** данных предполагает обработку каждого бита информации с использованием *гаммирования*, то есть изменения этого бита с помощью соответствующего ему бита псевдослучайной секретной последовательности чисел, которая формируется на основе ключа и имеет ту же длину, что и шифруемое сообщение. Как правило, биты исходных данных сравниваются с битами секретной последовательности с помощью логической операции XOR (исключающее ИЛИ, на выходе дающее 0, если значения битов совпадают, и 1, если они различаются).

Потоковое шифрование в настоящее время используют следующие алгоритмы:

* [RC4](https://encyclopedia.kaspersky.ru/glossary/rc4/)
* Salsa20
* HC-256
* WAKE

38.Асимметричное шифрование. Примеры алгоритмов, общая схема, преимущества и недостатки.

**Асимметричное шифрование** — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого.

**Общая схема асимметричного шифрования:**

Схема передачи данных между двумя субъектами (А и Б) с использованием открытого ключа выглядит следующим образом:

* Субъект А генерирует пару ключей, открытый и закрытый (публичный и приватный).
* Субъект А передает открытый ключ субъекту Б. Передача может осуществляться по незащищенным каналам.
* Субъект Б шифрует пакет данных при помощи полученного открытого ключа и передает его А. Передача может осуществляться по незащищенным каналам.
* Субъект А расшифровывает полученную от Б информацию при помощи секретного, закрытого ключа.

В такой схеме перехват любых данных, передаваемых по незащищенным каналам, не имеет смысла, поскольку восстановить исходную информацию возможно только при помощи закрытого ключа, известного лишь получателю и не требующего передачи.

**Примеры алгоритмов:**

* **RSA (Rivest, Shamir и Adleman)** — первая и наиболее распространённая криптосистема которая стала пригодной и для шифрования, и для цифровой подписи, использует факторизацию очень больших простых чисел в качестве отношения между двумя ключами.
* **Схема Эль-Гамаля (Elgamal)** -криптосистема основанная на вычислительной сложности проблемы дискретного логарифмирования. Долгое время была стандартом в США и России.
* **Алгоритм Диффи — Хеллмана** — криптографический протокол, позволяющий двум и более сторонам получить общий секретный ключ, используя незащищенный от прослушивания канал связи.
* **DSA (Digital Signature Algorithm**) — криптографический алгоритм с использованием открытого ключа для создания электронной подписи, но не для шифрования. Подпись создается секретно, но может быть публично проверена. Это означает, что только один субъект может создать подпись сообщения, но любой может проверить её корректность. Алгоритм основан на вычислительной сложности взятия логарифмов в конечных полях. Является частью стандарта цифровой подписи США.

### **Плюсы и минусы асимметричного шифрования:**

**Достоинства**:

* для передачи ключа не нужен закрытый канал связи
* открытый ключ может быть свободно распространен, это позволяет принимать данные от всех пользователей

**Недостатки:**

* ресурсоемкий алгоритм шифрования/ дешифрования
* трудно внести изменения
* используются более длинные ключи

Алгоритмы хэширования. Примеры, назначение.

Протокол TLS/SSL. Общая схема взаимодействия, назначение.

Понятие SSL-сертификата. Назначение. Самоподписанные сертификаты. Центры сертификации.

# 