[**Теоретические вопросы к экзамену**](#_hfi2i74ofzkg) **4**

[1. Понятие виртуальной сети. Виды виртуальных сетевых компонентов.](#_weiaciqobx9w) 4

[2. Трансляция сетевых адресов. Виды NAT.](#_yasnnz166lt9) 5

[3. Использование Linux для разработки. Стандартные программные средства.](#_vt4d5o7hreu3) 6

[4. Интерпретатор Python. Использование, версии. Понятие виртуального окружения, настройка, использование.](#_gumvelv5yuj9) 6

[5. Структура проекта на Python. Организация модулей. Файл зависимостей.](#_yvqpa6b93f1q) 7

[6. Системы контроля версий. Примеры, назначение, общие понятия.](#_ddv1qtu9vgwi) 9

[7. Общий алгоритм работы с СКВ Git. Инициализация репозитория, добавление файлов, коммиты.](#_34pbj9aybrl5) 10

[8. Работа с ветвлением в Git. Назначение веток. Создание, переключение, объединение веток. Разрешение конфликтов слияния.](#_inr0u6oxrgf0) 12

[9. Работа с удаленными репозиториями. Клонирование и форк репозиториев. Отправка и получение изменений в удаленный репозиторий.](#_c0bj5geec48g) 13

[10. Современные методологии работы с Git в командном проекте. GitFlow.](#_6qsv4rsfogsl) 13

[11. Понятие сетевого сокета. Применение, виды, схема взаимодействия.](#_wt8xpp204b82) 19

[12. Блокирующие операции при обмене через сокеты. Возможные ошибки. Таймауты.](#_q8twjavnyn3r) 20

[13. Транспортные протоколы TCP и UDP. Принципы работы, сравнение.](#_vhkjbh4tliu0) 21

[14. Клиент-серверное взаимодействие.](#_wrb2n4ar7neb) 21

[15. Реализация сокетов в языке Python. Модуль socket.](#_jqy6rpboweed) 22

[16. Понятие программного потока. Процессы и потоки.](#_nr5cxgocg5q4) 23

[17. Асинхронное программирование. Основные понятия. Параллелизм и конкуррентность.](#_77bnu655jbn) 24

[18. Блокирующие и неблокирующие операции.](#_izx9i1ha3o4n) 30

[19. Алгоритмы, ограниченные процессором и вводом-выводом. Основные характеристики, особенности выполнения и распараллеливания.](#_sqozvhwl2ii2) 31

[20. Особенности реализации многопоточности в Python. Модуль threading.](#_kvxsaplz47m) 32

[21. Особенности организации многопроцессорной программы в Python. Модуль multiprocessing.](#_f4unciikd83a) 35

[22. Асинхронное программирование в Python. Использование asyncio.](#_u5n5v0yxtxap) 38

[23. Параллельное программирование. Достоинства и недостатки.](#_jhteijpr4qhr) 39

[24. Понятие потокобезопасности. Причины, проблематика, способы обеспечения.](#_a0d3j0dvy13t) 40

[25. Алгоритм выполнения многопоточной программы. Блокировка потоков.](#_5jmxdzfiztuz) 41

[26. Доступ к общим ресурсам в многопоточной программе. Механизмы блокировки ресурсов модуля threading.](#_jhxtgpnemt20) 42

[27. Работа с файловой системой в Python. Основные операции.](#_5x0d9i4rc75x) 46

[28. Понятие веб-технологий. Основные характеристики, история, назначение.](#_ucznyfu3smzb) 47

[29. Программное обеспечение, используемое для веб-технологий. Виды, назначение, примеры.](#_7jyzx9qcgw4d) 48

[30. Понятие URL: назначение, применение, состав.](#_a3j26p4d6ev8) 51

[31. Понятие веб-сервера. Цели, принцип работы.](#_si7k2kuq8c6) 54

[32. Протокол HTTP. Принцип работы, назначение, основные понятия.](#_fyn4the48mcb) 57

[33. Настройка веб-сервера. Основные конфигурационные файлы, понятия.](#_7xo6gjb8l2fu) 59

[34. Виртуальные хосты. Применение, настройка.](#_da982poazkbu) 61

[35. Понятие прокси-сервера. Настройка сервера nginx.](#_nigygigd3nqi) 62

[36. Основные понятия криптографических методов защиты информации](#_7ow7jhrku9m3) 64

[37. Симметричное шифрование. Примеры алгоритмов, общая схема, виды.](#_7ow7jhrku9m3) 66

[38. Асимметричное шифрование. Примеры алгоритмов, общая схема, преимущества и недостатки.](#_7ow7jhrku9m3) 71

[39. Алгоритмы хэширования. Примеры, назначение.](#_7ow7jhrku9m3) 72

[40. Протокол TLS/SSL. Общая схема взаимодействия, назначение.](#_7ow7jhrku9m3) 74

[41. Понятие SSL-сертификата. Назначение. Самоподписанные сертификаты. Центры сертификации.](#_7ow7jhrku9m3) 77

[**Примерные практические задания к экзамену**](#_gjdgxs) **81**

[1. Напишите программу, которая создает нить. Родительская и вновь созданная нити должны распечатать десять строк текста.](#_bqo29ix5dyu2) 81

[2. Напишите простой эхо-сервер, использующий неблокирующие сокеты и клиент к нему.](#_vl779a4apbl7) 81

[3. Напишите простой многопоточный загрузчик URL. Список URL скрипт принимает как аргументы командной строки.](#_ibktjfc5ldus) 81

[4. 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.](#_xe244mufqlfs) 81

[5. Напишите программу, которая вычисляет число Пи при помощи ряда Эйлера. Количество потоков программы должно определяться параметром командной строки.](#_o8jjypmlx37x) 81

[6. 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.](#_3qjwwh90w8mz) 81

[7. 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.](#_b6uuldlyutq) 81

[8. 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.](#_2nr23funchus) 81

[9. Напишите программу, которая выводит на экран список номеров открытых портов на данной машине. Использовать команду netstat.](#_geapa1und6yn) 82

[10. Напишите программу, которая копирует файл с удаленного хоста в текущую папку по SSH. Имя файла и адрес хоста принимать как параметры.](#_sg07cwdhsit8) 82

# Теоретические вопросы к экзамену

## 1. Понятие виртуальной сети. Виды виртуальных сетевых компонентов.

Виртуальная сеть -совокупность технологий, позволяющих обеспечить одно или несколько сетевых соединений поверх другой сети.

Виртуальные сети представляют собой самые обычные виртуальные коммутаторы. К каждому виртуальному коммутатору могут присоединяться как сетевые интерфейсы виртуальных машин, так и физические сетевые интерфейсы сервера.

Виртуальные сети (то есть виртуальные коммутаторы) бывают трех типов: External, Internal и Private.

**External** – виртуальная сеть, имеющая выход «во внешний мир».

**Internal** – внутренняя виртуальная сеть, к которой могут подключаться только виртуальные интерфейсы – виртуальных машин и хостовой ОС. К физическому интерфейсу сеть типа Internal не привязывается, и, соответственно, выхода «вовне» не имеет.

**Private** – то же самое, что и Internal, за исключением того, что к такой сети могут подключаться только виртуальные машины. Сеть типа Private не имеет доступа ни ко «внешнему миру», ни к хостовой ОС.

**Виртуальные сетевые компоненты**

***Виртуальный сетевой адаптер*** (**Virtual network adapter) -** программный эмулятор сетевой карты, устанавливаемый на гостевую ОС.

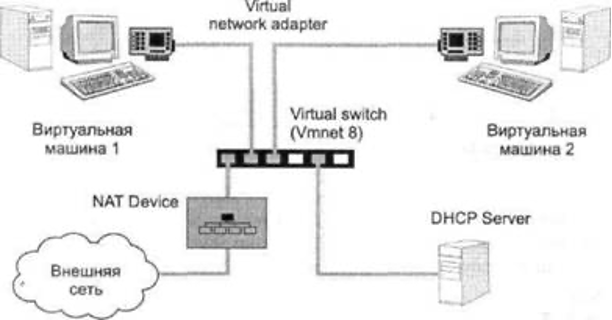
***Виртуальный адаптер хоста*** (**Host virtual adapter)-** виртуальный Ethernet-адаптер, устанавливаемый на хостовую ОС. Это виртуальное устройство служит для взаимодействия ВМ с хост-компьютером и включается в состав ВМ, когда для нее задается тип сетевого подключения **Host Only**

***Мост*** (**Bridge)-** программно реализованный сетевой мост, который позволяет подключать ВМ к реальной локальной сети, используя в качестве «посредника» хост-компьютер. Сетевой мост соединяет виртуальный сетевой адаптер с физическим Ethernet-адаптером хост-компьютера.

***Виртуальные коммутаторы***(**Virtual switches)-** эти устройства, подобно физическим коммутаторам, обеспечивают соединение между собой различных узлов сети.

***Устройство преобразования сетевых адресов***(**NAT Device)-** позволяет подключать ВМ к внешней сети (например, к Интернету), когда ВМ невозможно выделить собственный IP-адрес и приходится использовать IP-адрес, назначенный хост-компьютеру.

***DHCP-сервер***-программный компонент, обеспечивающий назначение сетевых IP-адресов виртуальным машинам в сети, в которой не используются подключения через мост



## 2. Трансляция сетевых адресов. Виды NAT.

Способы подключения к сети Интернет:

· Прямое IP подключение

· Подключение через NAT

· Подключение через прокси-сервер

**NAT=Network Address Translation** (“преобразование сетевых адресов”)-механизм в сетях TCP/IP, позволяющий преобразовывать IP адреса транзитных пакетов.

Функции NAT:

· Экономия IP адресов

· Ограничение доступа извне

· Сокрытие внутренних сервисов

Виды NAT:

· *Статический*-отображение незарегистрированного IP-адреса на зарегистрированный IP-адрес на основании один к одному. Особенно полезно, когда устройство должно быть доступным снаружи сети. Статический NAT использует сопоставление локальных и глобальных адресов один к одному. Статический NAT требует наличия достаточного количества общедоступных адресов для удовлетворения общего количества одновременных сеансов пользователя.

· *Динамический*-отображает незарегистрированный IP-адрес на зарегистрированный адрес из группы зарегистрированных IP-адресов. Динамический NAT использует пул публичных адресов и назначает их по принципу «первым пришел, первым обслужен». Когда внутреннее устройство запрашивает доступ к внешней сети, динамический NAT назначает доступный общедоступный IPv4-адрес из пула. Подобно статическому NAT, динамический NAT требует наличия достаточного количества общедоступных адресов для удовлетворения общего количества одновременных сеансов пользователя.

· *Перегруженный(маскарадный)-* форма динамического NAT, который отображает несколько незарегистрированных адресов в единственный зарегистрированный IP-адрес, используя различные порты. Известен также как [PAT](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_%D0%BF%D0%BE%D1%80%D1%82-%D0%B0%D0%B4%D1%80%D0%B5%D1%81) (Port Address Translation). При перегрузке каждый компьютер в частной сети транслируется в тот же самый адрес, но с различным номером порта.

## 3. Использование Linux для разработки. Стандартные программные средства.

Linux-подобные ОС предоставляют достаточно широкий выбор инструментов для разработки, а некоторые из них даже установлены по умолчанию (зависит от конкретного дистрибутива).

Стандартные:

* компиляторы C и C++ - gcc и g++
* какой-либо пакетный менеджер (apt, packman, yum)
* интерпретатор perl

То, что скорее всего будет в дистрибутиве:

* nano - простой текстовый редактор
* make и cmake - сборщик исходников для C-подобных языков программирования
* vim - более сложный текстовый редактор, есть возможность добавлять плагины
* интерпретатор python
* snap или flatpack (дистрибуция приложений в контейнерах)

Естественно, с помощью пакетных менеджеров можно поставить много стороннего софта, например:

* npm - пакетный менеджер js
* pip - пакетный менеджер python
* Pycharm (через snap) - IDE для работы с Python
* IDEA (через snap) - IDE для работы с Java
* Visual Studio Code (через snap) - удобный текстовый редактор на electorn с поддержкой множества плагинов

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

Интерпретатор Python после установки располагается, обычно, по пути /usr/local/bin/python. Кто установил интерпретатор на машину Unix, потенциально имеют поддержку библиотеки GNU readline, обеспечивающей усовершенствованное интерактивное редактирование и сохранение истории. Интерпретатор ведёт себя сходно шеллу Unix: если он вызван, когда стандартный ввод привязан к устройству tty — он считывает и выполняет команды в режиме диалога; будучи вызванным с именем файла в качестве аргумента или с файлом, назначенным на стандартный ввод — он читает и выполняет сценарий из этого файла.

У каждого проекта должен быть свой интерпретатор Python, со своей собственной изолированной директорией site-packages. Это и есть основная идея, стоящая за виртуальными окружениями. Виртуальное окружение — это самостоятельная копия интерпретатора со своими пакетами.

Настройка:

Virtualenv можно установить с использованием менеджера pip (ссылка на статью), либо скачать исходные коды проекта и установить приложение вручную.

pip install virtualenv

Виртуальное окружение создается следующей командой:

virtualenv \*имя окружения\*

После выполнения данной команды, в текущем каталоге будет создан новый каталог с именем окружения.

Для активации виртуального окружения:

source \*имя\*/bin/activate

Команда source выполняет bash-скрипт без запуска второго bash-процесса.

После завершения работы деактивировать виртуальную среду можно с помощью команды deactivate.

Введите ее и приставка \*имя\* пропадет. Вы вернетесь к использованию глобально версии python.

Удаление виртуальной среды:

rm -rf \*имя\*

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

**Модули**

Модуль в языке Python - это обычный файл с расширением .ру. Модуль может содержать любой программный код на языке Python.

Не все модули располагаются в файлах с расширением .ру, например, модуль sys встроен в Python, а некоторые модули написаны на других языках программирования (чаще всего на языке С). Однако большая часть библиотеки языка Python написана именно на языке Python.

При импорте модуля Python выполняет весь код в нём, который не находится в функциях. Условие if \_\_name\_\_ == “\_\_main\_\_” необходимо для того, чтоб код в нем не выполнялся при импорте файла в качестве модуля, НО при непосредственном вызове файла через интерпретатор Python отрабатывался.

**Порядок поиска файлов, содержащих модули**

Порядок следующий:

1.каталог, где находится сама программа, даже если она вызывается из другого каталога;

2.пути к каталогам из переменной окружения PYTHONPATH, если она определена;

3.пути к каталогам стандартной библиотеки языка Python - они определяются на этапе установки Python.

Существует 2 типа импорта в Python. Импорт всего модуля, либо каких-либо его конкретных методов. Эти типы также кто-то называет абсолютными и относительными.

**Абсолютный импорт:**

После импортирования модуля с помощью import появляется возможность получить доступ к его методам с помощью module, который становится доступен внутри исполняемого файла

impor trandom

print(type(random))#<class 'module'>

random\_number = random.randint(1,10)

print(random\_number)

**Относительный импорт:**

Импортировать определенные методы какого-либо модуля можно с помощью инструкции from.

from random import randint, shuffle

print(type(randint))#<class 'method'>

random\_number = randint(1,10)

print(random\_number)

Также существует **частный случай** **относительного импорта** с \*:

fromrandomimport\*

random\_number = randint(1,10)

print(random\_number)

Но так лучше не делать т.к. У разных модулей по теории могут быть одинаковые названия методов, что может привести к ошибкам в программе.

Использование конструкции as:

import random as koshkas

random\_value = koshkas.randint(1,6)

print(random\_value)

**Файл зависимостей**

При дистрибуции проекта настоятельно рекомендуется использовать файл зависимостей **requirements.txt**, который содержит в себе все библиотеки, которые используются текущим проектом

Чаще всего файл зависимостей используют с обособленным виртуальном окружением проекта, реже с глобальным окружением.

Для того, чтоб экспортировать названия всех библиотек текущего окружения в **requirements.txt**, используют команду:

pip3 freeze > **requirements.txt**

Для того, чтоб поставить все пакеты из файла **requirements.txt**, используют команду

pip3 install -r **requirements.txt**

Дистрибуция файла зависимостей с исходным кодом проекта - хорошая практика, желательно следовать ей.

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

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Все они делятся на два типа:

· распределенные (Git, Mercurial) – изменения хранятся в локальных хранилищах компьютеров и оттуда синхронизируются с другими компьютерами;

· централизованные (CVS, SVN) – все данные хранятся на центральном сервере, и оттуда каждый локальный компьютер получает обновленные данные.

Обычно вся работа над проектом (сайтом) со стороны разработчика сводится к трем простым операциям:

· удаление;

· добавление;

· изменение.

С приходом VCS работать стало гораздо проще:

· можно откатывать изменения, если смысла в их внедрении нет;

· можно быстро и безболезненно восстанавливать поврежденные файлы;

· можно определить, кто из команды писал определенный блок кода;

· можно следить за процессом, даже если в один и тот же момент над одним модулем работает несколько разработчиков или даже команд по всему миру.

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

**git add** Команда git add добавляет содержимое рабочей директории в индекс (staging area) для последующего коммита. По умолчанию git commit использует лишь этот индекс, так что вы можете использовать git add для сборки слепка вашего следующего коммита.

**git status** Команда git status показывает состояния файлов в рабочей директории и индексе: какие файлы изменены, но не добавлены в индекс; какие ожидают коммита в индексе. Вдобавок к этому выводятся подсказки о том, как изменить состояние файлов.

**git diff** Команда git diff используется для вычисления разницы между любыми двумя Git деревьями. Это может быть разница между вашей рабочей директорией и индексом (собственно git diff), разница между индексом и последним коммитом (git diff --staged), или между любыми двумя коммитами (git diff master branchB).

**git difftool** Команда git difftool просто запускает внешнюю утилиту сравнения для показа различий в двух деревьях, на случай если вы хотите использовать что-либо отличное от встроенного просмотрщика git diff.

**git commit** Команда git commit берёт все данные, добавленные в индекс с помощью git add, и сохраняет их слепок во внутренней базе данных, а затем сдвигает указатель текущей ветки на этот слепок.

**git reset** Команда git reset, как можно догадаться из названия, используется в основном для отмены изменений. Она изменяет указатель HEAD и, опционально, состояние индекса. Также эта команда может изменить файлы в рабочей директории при использовании параметра --hard, что может привести к потере наработок при неправильном использовании, так что убедитесь в серьёзности своих намерений прежде чем использовать его.

**git rm** Команда git rm используется в Git для удаления файлов из индекса и рабочей директории. Она похожа на git add с тем лишь исключением, что она удаляет, а не добавляет файлы для следующего коммита.

**git mv** Команда git mv — это всего лишь удобный способ переместить файл, а затем выполнить git addдля нового файла и git rm для старого.

**git clean**  Команда git clean используется для удаления мусора из рабочей директории. Это могут быть результаты сборки проекта или файлы конфликтов слияний.

**Команды ветвления и слияния.**

**git branch**

Команда git branch —  "менеджер веток". Она умеет перечислять ваши ветки, создавать новые, удалять и переименовывать их.

**git checkout**

Команда git checkout используется для переключения веток и выгрузки их содержимого в рабочую директорию.

**git merge**

Команда git merge используется для слияния одной или нескольких веток в текущую. Затем она устанавливает указатель текущей ветки на результирующий коммит.

**git mergetool**

Команда git mergetool просто вызывает внешнюю программу слияний, в случае если у вас возникли проблемы слияния.

**git log**

Команда git log используется для просмотра истории коммитов, начиная с самого свежего и уходя к истокам проекта. По умолчанию, она показывает лишь историю текущей ветки, но может быть настроена на вывод истории других, даже нескольких сразу, веток. Также её можно использовать для просмотра различий между ветками на уровне коммитов.

**git stash**

Команда git stash используется для временного сохранения всех незакоммиченных изменений для очистки рабочей директории без необходимости коммитить незавершённую работу в новую ветку.

**git tag**

Команда git tag используется для задания постоянной метки на какой-либо момент в истории проекта. Обычно она используется для релизов.

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

**Работа с ветвлением в Git. Назначение веток. Создание, переключение, объединение веток**

В Git принято использовать несколько веток (бранчей) для работы. Главная ветка называется master или main, от нее можно создать другие ветки, например, dev.

Чаще делают так, что каждая ветка отвечает за решение какого-либо таска/задачи (например, в Jira или Github issues), тогда алгоритм работы с git будет следующий:

1. Сделать бранч (ветку) от master с номером задачи, например, issue\_39594

git branch issue\_39594

1. Переключить ветку с master на issue\_39594

git checkout issue\_39594

1. Писать код и делать коммиты в ваш бранч issue\_39594

git add .

git commit -m "Какое-то изменение в коде"

1. Как код написан, осуществить пулл реквест с ветки issue\_39594 обратно в ветку master. После открытия пулл реквеста возможно осуществление автодеплоя и автотестов изменений разрабатываемого решения.
2. Если все тесты прошли, то какое-либо ответственное лицо осуществляет код ревью ваших изменений в пул реквесте, если что-то не так - решение отправляется на доработку
3. Ответственное лицо осуществляет мердж вашего пулл реквеста в master-ветку.

git checkout master

git merge hotfix

1. Пулл реквест закрывается
2. Бранч issue\_39594 удаляется (зависит от политики работы с репо).

git push origin --delete issue\_39594

Кстати, для работы с git необязательно работать напрямую через командную строку, можно с помощью GitKraken или Github Desktop.

## 

**Разрешение конфликтов слияния.**

При работе с ветками бывает такое, что возникают конфликты слияния. Это происходит потому, что в разных ветках были разные коммиты, связанные с одним и тем же файлом. В этом случае git поругается, что ничего сделать не может и далее вам придется вручную править файл для разрешения конфликта слияния, после чего сделать коммит.

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

Работа с удалёнными репозиториями

Для того, чтобы внести вклад в какой-либо Git-проект, вам необходимо уметь работать с удалёнными репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее.

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

Наиболее популярной СКВ для организации совместной разработки является Git. Ветвление и слияние обычно считаются опасными из-за конфликтов слияния и потому проводятся как можно реже.

Но с Git эти действия становятся исключительно простыми и дешёвыми, и потому на деле они становятся центральными элементами обычного *ежедневного* рабочего процесса.

Благодаря своей простоте и предсказуемости, ветвление и слияние больше не являются действиями, которых стоит опасаться. Теперь инструменты управления версиями способны помочь в ветвлении и слиянии больше, чем какие-либо другие.

Gitflow— это модель которая, по сути, является просто набором процедур, которые исполняет каждый член команды, чтобы все вместе могли достичь высокой управляемости процесса разработки.

Она предполагает выстраивание строгой модели ветвления вокруг релиза проекта, которая дает надежную схему управления крупными проектами.

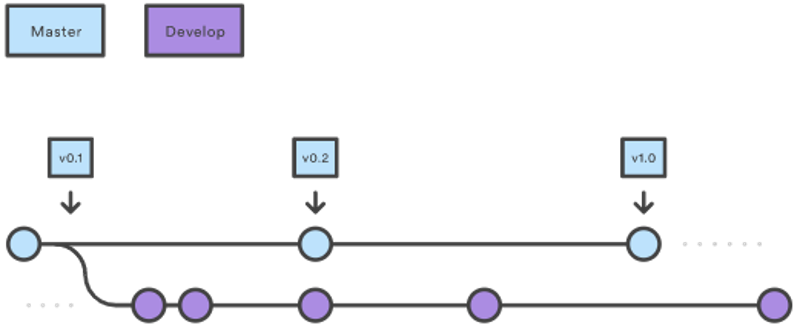
Кроме feature-веток в ней используются отдельные ветки для подготовки, поддержки и записи релиза.

Gitflow использует собственный набор инструментов git-flow, который легко интегрируется с Git, добавляя новые команды Git.

**Начало работы**

Набор инструментов git-flow нужно установить отдельно. После установки git-flow необходимо выполнить команду git flow init. Git-flow является оберткой для Git. Команда git flow init является расширением стандартной команды git init и ничего не меняет в вашем репозитории, кроме того, что создает ветки.

**Как это работает**

****

**Ветки master и develop**

Вместо использования одной ветки master, в этой модели используется две ветки для записи истории проекта. В ветке master хранится официальная история релиза, а ветка develop служит в качестве интеграционной ветки для новых функций. Также, удобно тегировать все коммиты в ветке master номером версии.

Первым шагом является создание ветки develop от ветки master. Проще всего это сделать одному разработчику, локально создав пустую ветку и отправив ее в центральный репозиторий:

git branch develop

git push -u origin develop

В этой ветке будет находиться вся история проекта, в то время как master содержит частичную историю. Остальные разработчики теперь должны клонировать центральный репозиторий и создать отслеживающую ветку для ветки develop.

При использовании библиотеки расширений git-flow, для создания ветки develop можно выполнить git flow init в существующем репозитории:

$ git flow init

Initialized empty Git repository **in** ~/project/.git/

No branches exist yet. Base branches must be created now.

Branch name **for** production releases: [master]

Branch name **for** "next release" development: [develop]

How to name your supporting branch prefixes?

Feature branches? [feature/]

Release branches? [release/]

Hotfix branches? [hotfix/]

Support branches? [support/]

Version tag prefix? []

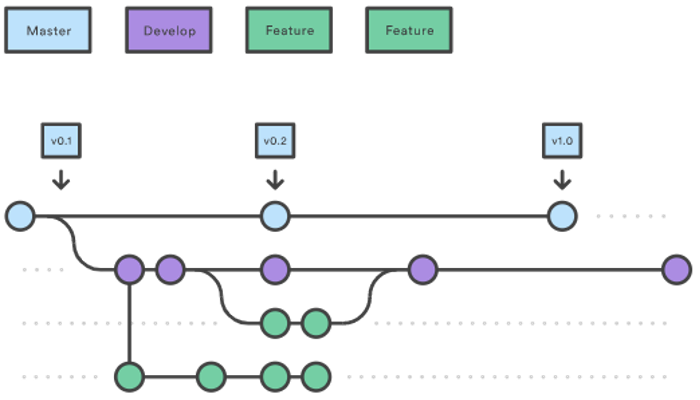
$ git branch

**\*** develop

master

**Ветки для функций (feature branches)**

Каждая новая функциональность должна разрабатываться в отдельной ветке, которую можно отправлять в центральный репозиторий для создания резервной копии/для совместной работы команды. Ветки функций создаются не на основе master, a на основе develop. Когда работа над новой функциональностью завершена, она вливается назад в develop. Новый код не должен отправляться напрямую в master.



Ветки функций объединяются с веткой develop как в модели Feature Branch Workflow. Но на этом работа по схеме Gitflow не заканчивается.

**Создание ветки функции**

Без использования расширений git-flow:

git checkout develop

git checkout -b feature\_branch

При использовании git-flow:

git flow feature start feature\_branch

Далее, продолжайте работу c Git как обычно.

**Окончание работы с веткой**

По окончании разработки новой функциональности следующим шагом следует объединить ветку feature\_branch c develop. Используйте команды:

Без использования расширений git-flow:

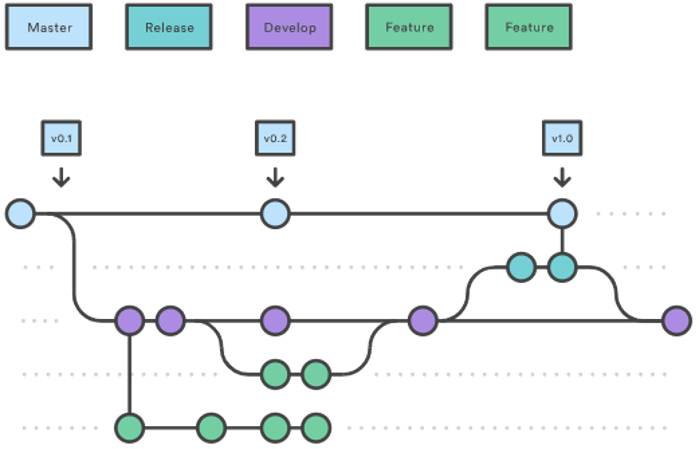
git checkout develop

git merge feature\_branch

При использовании git-flow:

git flow feature finish feature\_branch

**Ветки релиза**

****

Когда в ветку develop уже слито достаточно нового кода для релиза (или подходит установленная дата предрелиза), от ветки develop создается ветка release. Создание данной ветки означает начало следующего цикла релиза, в ходе которой новая функциональность уже не добавляется, а производится только отладка багов, создание документации и решение других задач, связанных с релизом. Когда все готово, ветка release сливается в master, и ей присваивается тег с версией. Кроме этого, она должна быть также слита обратно в ветку develop, в которой с момента создания ветки релиза могли добавляться изменения с момента создания ветки релиза.

Использование отдельной ветки для подготовки релиза позволяет одной команде дорабатывать текущий релиз пока другая команда уже работает над функциональностью для следующего релиза. Это также позволяет разграничить этапы разработки (например, легко сказать: «На этой неделе мы готовимся к версии 4.0» и фактически увидеть это в структуре репозитория).

Создание веток релиза – это еще одна простая операция ветвления. Как и ветки функций, ветки релизов основаны на ветке develop. Новая ветка release может быть создана с использованием следующих команд:

Без использования расширений git-flow:

git checkout develop

git checkout -b release/0.1.0

При использовании git-flow:

$ git flow release start 0.1.0

Switched to a new branch 'release/0.1.0'

Когда релиз готов к отправке, он сливается в master и develop, а ветка релиза удаляется. Важно влить ее обратно в develop, поскольку в ветку release могут быть добавлены критические обновления, и они должны быть доступны для новых функций. Если ваша команда делает акцент на проверку кода, этот момент идеален для пул-реквеста.

Для завершения работы на ветке релиза, используйте следующие команды:

Без использования расширений git-flow:

git checkout develop

git merge release/0.1.0

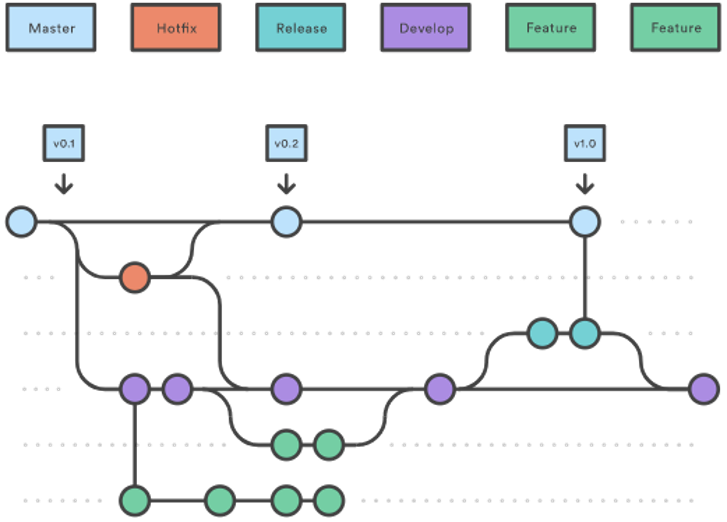
Или при использовании git-flow:

git checkout master

git checkout merge release/0.1.0

git flow release finish '0.1.0'

**Ветки hotfix**

****

Ветки hotfix используются для быстрого внесения исправлений в рабочую версию кода. Ветки hotfix очень похожи на ветки release и feature, за исключением того, что они созданы от master, а не от develop. Это единственная ветка, которая должна быть создана непосредственно от master. Как только исправление завершено, ветка hotfix должна быть объединена как с master, так и с develop (или с веткой текущего релиза), а master должен быть помечен обновленным номером версии.

Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла подготовки к релизу. Можно говорить о ветках hotfix как об особых ветках relese, которые работают напрямую с master. Ветка hotfix может быть создана с помощью следующих методов:

Без использования расширений git-flow:

git checkout master

git checkout -b hotfix\_branch

Или при использовании git-flow:

$ git flow hotfix start hotfix\_branch

Как и в работе с веткой release, ветка hotfix объединяется как с master, так и с develop.

git checkout master

git merge hotfix\_branch

git checkout develop

git merge hotfix\_branch

git branch -D hotfix\_branch

$ git flow hotfix finish hotfix\_branch

**Пример**

Пример команд, демонстрирующий полный цикл работы с веткой функции, выглядит следующим образом. Предположим, что у нас есть репозиторий с веткой master.

git checkout master

git checkout -b develop

git checkout -b feature\_branch

*# работа ведется на ветке feature*

git checkout develop

git merge feature\_branch

git checkout master

git merge develop

git branch -d feature\_branch

Помимо ветки функции и release, приведем пример создания ветки hotfix:

git checkout master

git checkout -b hotfix\_branch

*# работа сделана, коммиты добавлены в hotfix\_branch*

git checkout develop

git merge hotfix\_branch

git checkout master

git merge hotfix\_branch

**Ключевые идеи, которые нужно запомнить о Gitflow:**

· Данная модель отлично подходит для организации рабочего процесса на основе релизов.

· Gitflow предлагает создание отдельной ветки для исправлений ошибок в продуктовой среде.

**Последовательность работы при использовании модели Gitflow:**

1. Из master создается ветка develop.

2. Из develop создаются ветки feature.

3. Когда разработка новой функциональности завершена, она объединяется с веткой develop.

4. Из develop создается ветка release.

5. Когда ветка релиза готова, она объединяется с develop и master.

6. Если в master обнаружена проблема, из нее создается ветка hotfix.

7. Как только исправление на ветке hotfix завершено, она объединяется с develop и master.

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

**Сетевой сокет** — комбинация IP-адреса и номера порта, которые представляют собой способ адресации и обеспечивают нормальное взаимодействие большого количества приложений в рамках одной системы.

Существует **2 вида сокетов** – клиентские и серверные.

Применение: сокет однозначно идентифицирует прикладной процесс в сети TCP/IP.

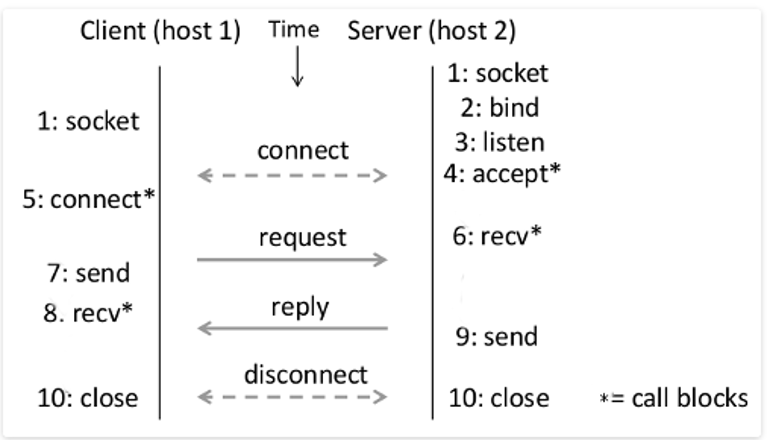
API сокетов – это название программного интерфейса, предназначенного для обмена данными между процессами, находящимися на одном или на разных объединенных сетью компьютерах.

API сокетов включает в себя функции создания сокета, установки параметров сокета (сетевой адрес, номер порта), функции создания канала и обмена данными между сокетами.

**Список функций:**

|  |  |
| --- | --- |
| **Общие** |  |
| **Socket** | Создать новый сокет и вернуть файловый дескриптор |
| **Send** | Отправить данные по сети |
| **Receive** | Получить данные из сети |
| **Close** | Закрыть соединение |
| **Серверные** |  |
| **Bind** | Связать сокет с IP-адресом и портом |
| **Listen** | Объявить о желании принимать соединения. Слушает порт и ждет когда будет установлено соединение |
| **Accept** | Принять запрос на установку соединения |
| **Клиентские** |  |
| **Connect** | Установить соединение |

**Схема взаимодействия:**



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

**Блокирующие операции при обмене через сокеты**

Сокеты могут работать в одном из двух режимов: блокирующем или неблокирующем.

**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 UDP** обеспечивает ненадежную доставку датаграмм и не поддерживает соединений из конца в конец. Другими словами, его пакеты могут быть потеряны, продублированы или прийти не в том порядке, в котором они были отправлены.

**Протокол TCP** предоставляет транспортные услуги, отличающиеся от услуг UDP. Вместо ненадежной доставки датаграмм без установления соединений, он обеспечивает гарантированную доставку с установлением соединений между прикладными процессами в виде байтовых потоков.

**Разница между протоколами TCP и UDP** – в так называемой “гарантии доставки”. TCP требует отклика от клиента, которому доставлен пакет данных, подтверждения доставки, и для этого ему необходимо установленное заранее соединение. Также протокол TCP считается надежным, тогда как UDP получил даже именование “протокол ненадежных датаграмм. TCP исключает потери данных, дублирование и перемешивание пакетов, задержки. UDP все это допускает, и соединение для работы ему не требуется. Процессы, которым данные передаются по UDP, должны обходиться полученным, даже и с потерями. TCP контролирует загруженность соединения, UDP не контролирует ничего, кроме целостности полученных датаграмм.

## 14. Клиент-серверное взаимодействие.

Это взаимодействие двух программных продуктов между собой, один из которых выступает в качестве сервера, а другой соответственно в качестве клиента. Клиент посылает запрос, а сервер отвечает ему. А что такое клиент и что такое сервер? Спросите Вы. Клиент это программная оболочка, с которой взаимодействует пользователь. А сервер это та часть программного обеспечения, которая выполняет все основные функции (хранит данные, выполняет расчеты). Другими словами, 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).

## 15. Реализация сокетов в языке Python. Модуль socket.

Сокет — название программного интерфейса для обеспечения обмена данными между процессами. Существуют клиентские и серверные сокеты. Серверный сокет прослушивает определенный порт, а клиентский подключается к серверу. После того, как было установлено соединение начинается обмен данными. Сокет на языке Python позаимствован из языка программирования C и операционной системы Unix. Для работы с сокетами нужен клиент(ы) и сервер.

В Python для работы с сокетами используется модуль socket:

1. Подключение библиотеки для работы с сокетами. import socket

2. Создание сокета: sock = socket.socket()

3. Определяемся с хостом и портом сервера sock.bind(‘localhost’,6789)

4. Указываем количество клиентов, которые будет слушать наш сервер. Sock.listen(5)

5. Client, addr = sock.accept()

· Sock.recv – получает данные

· Sock.send – отправляет данные.

Мы можем построить наши сервера как по udp, так и по tcp протоколу.

**UDP** отправляет сообщения, но их размер ограничен и не гарантируется, что они достигнут места назначения. UDP не гарантирует в каком порядке придут сообщения. **TCP** вместо сообщений отправляет потоки байтов. Нельзя сказать, сколько байтов отправит или получит система с каждым вызовом. TCP доставляет данные в том порядке, в котором они были отправлены.

## 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 модулю 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

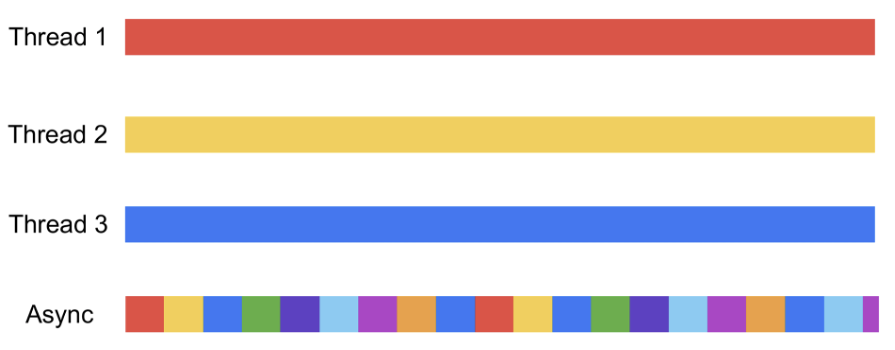
## 17. Асинхронное программирование. Основные понятия. Параллелизм и конкуррентность.

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

**Параллелизм и конкурентность**

**Параллелизм –** когда несколько задач выполняются одновременно(параллельно). На картинке Thread 1-3. Такая форма обычно контролируется системой

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

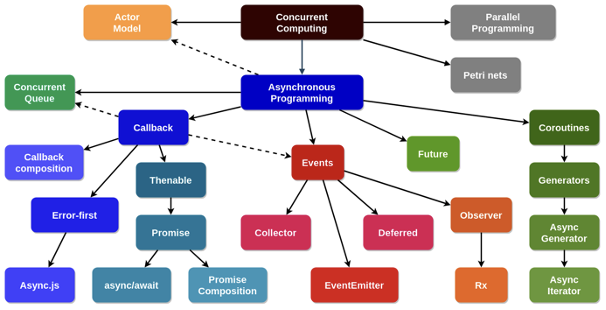
Параллелизм подразумевает конкурентность (Читайте про GIL, замки, гонки и т.д.). Но конкурентность не всегда подразумевает параллелизм.

**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:

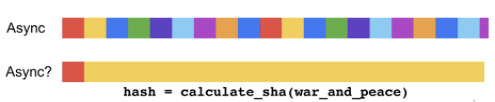
- **ThreadPool**

- **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)

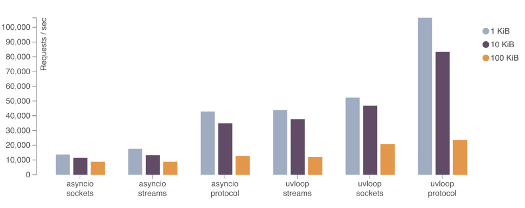
- Выполнять код в пулах потоков или процессов

- Выполнять подпроцессы

Альтернативные циклы: **uvloop**

- Реализован поверх **libuv**, стабильный

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



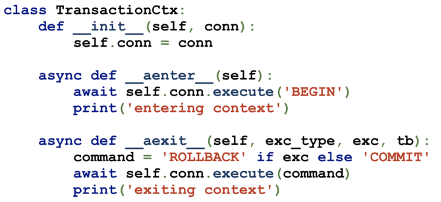
**Асинхронный менеджер контекста:**

- 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

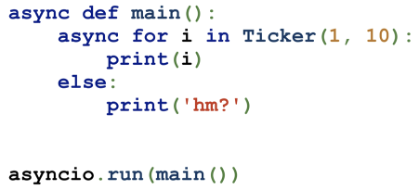
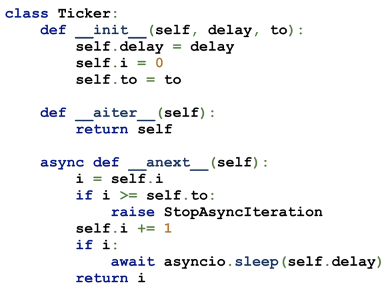
- Итерируемый объект должен реализовывать метод **\_\_aiter\_\_**

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

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

- PEP 492

Пример:

****

**Асинхронные генераторы**

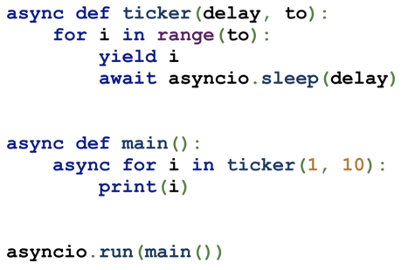
- Асинхронная функция, в которой используется **yield**

- 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()}

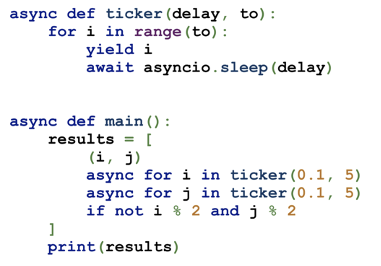
- Список list(): [i async for i in agen()]

- 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.

В синхронном коде каждая операция ожидает окончание предыдущей, по этому программа может не отвечать пока выполняется какая то большая операция.

**Блокирующие операции** – операции которые мешают выполнению других операций.

Например:

- Если браузер будет работать в 1 потоке, то загрузка файла будет блокировать выполнение других функций браузера.

- I/O – ввод/вывод – взаимодействие с “Миром”, пример ниже.

- conn\_data = conn.read(1024) # network

- file\_data = fileobj.read(1024) # filesystem

Решение проблем блокировки:

- **Процесс** - отдельная **подпрограмма**(**процесс**) от текущей программы. Требует большие затраты на создание, требует очень много памяти, сложно связать с основной программой, оверхед со стороны **ОС** (**user-space to scheduling, context switching**)

- **Поток** - подпрограмма, которая выполняется внутри текущей(в текущем процессе). Требует большие затраты на создание(меньше по сравнению с процессами). При работе в высоконагруженных систем выполнения могут произойти например **race conditional, dedlock, starvation**, да еще и **GIL**. Если не ограничить **Pool Threads**, программа встанет из за переполнения файлового дескриптора или ограничений системы.

- **Асинхронность** – когда программа работает используя **Event Loop** (цикл событий), где **Event** это подпрограммы (**coroutine**), а **Loop** место где происходит их хранение и выполнение путем прохода по всем событиям.

Рассмотрим данные операции на примере сокетов. Сокеты могут работать в одном из двух режимов: блокирующем или неблокирующем.

Сокет 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.

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

Процессы можно классифицировать как те, которые *ограничены скоростью ввода-вывода* (*I/O-bound*), и те, которые *ограничены скоростью процессора*(*processor-bound*). К первому типу относятся процессы, которые большую часть своего времени выполнения тратят на отправку запросов на ввод-вывод информации и на ожидание ответов на эти запросы. Следовательно, такие процессы часто готовы к выполнению, но могут выполняться только в течение короткого периода времени, так как в конце концов они блокируются в ожидании выполнения ввода-вывода (имеются в виду не только дисковые операции ввода-вывода, но и любой другой тип ввода-вывода информации, как, например, работа с клавиатурой).

Процессы, ограниченные скоростью процессора, наоборот, большую часть времени исполняют программный код. Такие процессы обычно выполняются до того момента, пока они не будут вытеснены, так как эти процессы не блокируются в ожидании на запросы ввода-вывода. Поскольку такие процессы не влияют на скорость ввода-вывода, то для обеспечения нормальной скорости реакции системы не требуется, чтобы они выполнялись часто. Стратегия планирования процессов, ограниченных скоростью процессора, поэтому предполагает, что такие процессы должны выполняться реже, но более продолжительный период времени. Конечно, оба эти класса процессов взаимно не исключают друг друга. Пример процесса, ограниченного скоростью процессора, — это выполнение бесконечного цикла.

Распараллеливание программ — процесс адаптации [алгоритмов](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), записанных в виде программ, для их эффективного исполнения на вычислительной системе [параллельной архитектуры](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) (в последнее время, как правило, на [многопроцессорной](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) [вычислительной системе](https://ru.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C)). Заключается либо в переписывании программ на специальный язык, описывающий параллелизм и понятный [трансляторам](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80) целевой вычислительной системы, либо к вставке специальной разметки (например, инструкций [MPI](https://ru.wikipedia.org/wiki/Message_Passing_Interface) или [OpenMP](https://ru.wikipedia.org/wiki/OpenMP)).

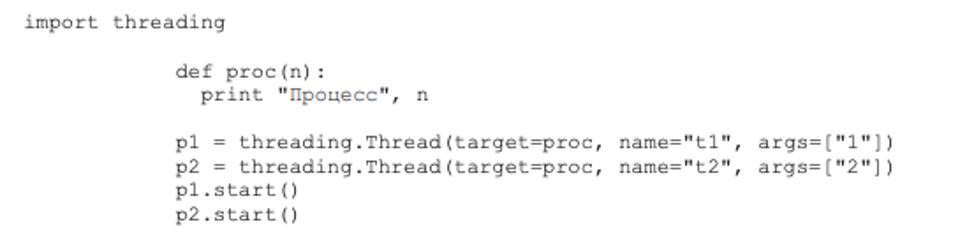
Распараллеливание может быть ручным, автоматизированным и полуавтоматизированным. Для оценки эффективности его качества применяются следующие критерии:

Ускорение -  время исполнения распараллеленной программы на p процессорах, T1— время исполнения исходной программы. В идеальном случае (отсутствие накладных расходов на организацию параллелизма) равна p.

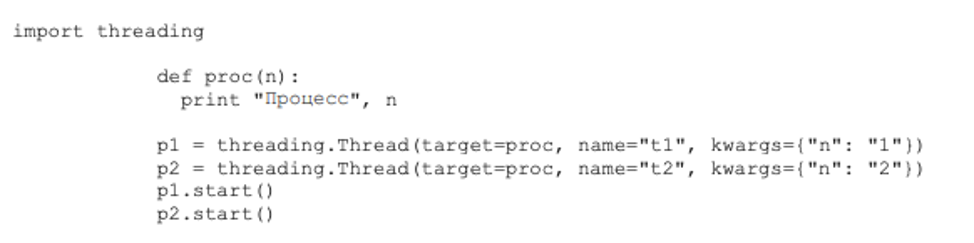
Загруженность , показывающая долю использования процессоров. В идеальном случае равна 1, или 100 %. Эта величина зачастую гораздо более наглядно характеризует эффективность параллелизма в серии испытаний при разных p, чем Sp, особенно на графиках.

## 20. Особенности реализации многопоточности в Python. Модуль threading.

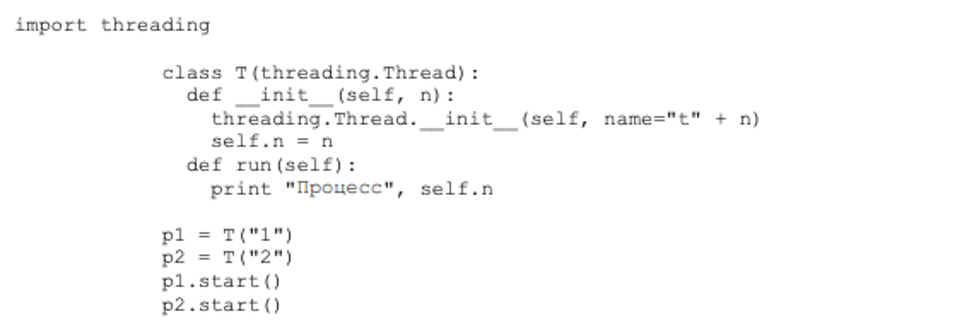
В современной операционной системе, даже не выполняющей ничего особенного, могут одновременно работать несколько процессов (processes). Например, при запуске программы запускается новый процесс. Функции для управления процессами можно найти в стандартном модуле os языка Python. Здесь же речь пойдет о потоках. Потоки управления (threads) образуются и работают в рамках одного процесса. В однопоточном приложении (программе, которая не использует дополнительных потоков) имеется только один поток управления. Говоря упрощенно, при запуске программы этот поток последовательно исполняет встречаемые в программе операторы, направляясь по одной из альтернативных ветвей оператора выбора, проходит через тело цикла нужное число раз, выбирается к месту обработки исключения при возбуждении исключения. В любой момент времени интерпретатор Python знает, какую команду исполнить следующей. После исполнения команды становится известно, какой команде передать управление. Эта ниточка непрерывна в ходе выполнения программы и обрывается только по ее завершении. Теперь можно представить себе, что в некоторой точке программы ниточка раздваивается, и каждый поток идет своим путем. Каждый из образовавшихся потоков может в дальнейшем еще несколько раз раздваиваться. (При этом один из потоков всегда остается главным, и его завершение означает завершение всей программы.) В каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет кванты времени каждому потоку. Такое, казалось бы, незначительное усложнение механизма выполнения программы на самом деле требует качественных изменений в программе - ведь деятельность потоков должна быть согласована. Нельзя допускать, чтобы потоки одновременно изменяли один и тот же объект, результат такого изменения, скорее всего, нарушит целостность объекта. Одним из классических средств согласования потоков являются объекты, называемые семафорами. Семафоры не допускают выполнения некоторого участка кода несколькими потоками одновременно. Самый простой семафор - замок (lock) или mutex (от английского mutually exclusive, взаимоисключающий). Для того чтобы поток мог продолжить выполнение кода, он должен сначала захватить замок. После захвата замка поток выполняет определенный участок кода и потом освобождает замок, чтобы другой поток мог его получить и пройти дальше к выполнению охраняемого замком участку программы. Поток, столкнувшись с занятым другим потоком замком, обычно ждет его освобождения. Поддержка многопоточности в языке Python доступна через использование ряда модулей. В стандартном модуле threading определены нужные для разработки многопоточной (multithreading) программы классы: несколько видов семафоров (классы замков Lock, RLock и класс Semaphore ) и другие механизмы взаимодействия между потоками (классы Event и Condition ), класс Timer для запуска функции по прошествии некоторого времени. Модуль Queue реализует очередь, которой могут пользоваться сразу несколько потоков. Для создания и (низкоуровневого) управления потоками в стандартном модуле thread определен класс Thread. Пример многопоточной программы В следующем примере создается два дополнительных потока, которые выводят на стандартный вывод каждый свое:



Сначала получается два объекта класса Thread, которые затем и запускаются с различными аргументами. В данном случае в потоках работает одна и та же функция proc(), которой передается один аргумент, заданный в именованном параметре args конструктора класса Thread. Нетрудно догадаться, что метод start() служит для запуска нового потока. Таким образом, в приведенном примере работают три потока: основной и два дополнительных (с именами "t1" и "t2" ). Функции модуля threading В модуле threading, который здесь используется, есть функции, позволяющие получить информацию о потоках: · activeCount() Возвращает количество активных в настоящий момент экземпляров класса Thread. Фактически, это len(threading.enumerate()). · currentThread() Возвращает текущий объект-поток, то есть соответствующий потоку управления, который вызвал эту функцию. Если поток не был создан через модуль threading, будет возвращен объект-поток с сокращенной функциональностью (dummy thread object). · enumerate() Возвращает список активных потоков. Завершившиеся и еще не начатые потоки не входят в список. Класс Thread Экземпляры класса threading.Thread представляют потоки Python-программы. Задать действия, которые будут выполняться в потоке, можно двумя способами: передать конструктору класса исполняемый объект и аргументы к нему или путем наследования получить новый класс с переопределенным методом run(). Первый способ был рассмотрен в примере выше. Конструктор класса threading.Thread имеет следующие аргументы: Thread(group, target, name, args, kwargs) Здесь group - группа потоков (пока что не используется, должен быть равен None ), target - объект, который будет вызван в методе run(), name - имя потока, args и kwargs - последовательность и словарь позиционных и именованных параметров (соответственно) для вызова заданного в параметре target объекта. В примере выше были использованы только позиционные параметры, но то же самое можно было выполнить и с применением именованных параметров:



То же самое можно проделать через наследование от класса threading.Thread с определением собственного конструктора и метода run():



Самое первое, что необходимо сделать в конструкторе - вызвать конструктор базового класса. Как и раньше, для запуска потока нужно выполнить метод start() объекта-потока, что приведет к выполнению действий в методе run(). Жизнью потоков можно управлять вызовом методов: · start() Дает потоку жизнь. · run() Этот метод представляет действия, которые должны быть выполнены в потоке. · join([timeout]) Поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван. Параметр timeout (число с плавающей точкой) позволяет указать время ожидания (в секундах), по истечении которого приостановленный поток продолжает свою работу независимо от завершения потока, чей метод join был вызван. Вызывать join() некоторого потока можно много раз. Поток не может вызвать метод join() самого себя. Также нельзя ожидать завершения еще не запущенного потока. Слово "join" в переводе с английского означает "присоединить", то есть, метод, вызвавший join(), желает, чтобы поток по завершении присоединился к вызывающему метод потоку. · getName() Возвращает имя потока. Для главного потока это "MainThread". · setName(name) Присваивает потоку имя name. · isAlive() Возвращает истину, если поток работает (метод run() уже вызван, но еще не завершился). · isDaemon() Возвращает истину, если поток имеет признак демона. Программа на Python завершается по завершении всех потоков, не являющихся демонами. Главный поток демоном не является. · setDaemon(daemonic) Устанавливает признак daemonic того, что поток является демоном. Начальное значение этого признака заимствуется у потока, запустившего данный. Признак можно изменять только для потоков, которые еще не запущены. В модуле Thread пока что не реализованы возможности, присущие потокам в Java (определение групп потоков, приостановка и прерывание потоков извне, приоритеты и некоторые другие вещи), однако они, скорее всего, будут созданы в недалеком будущем.

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

Multiprocessing

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​

Создание нескольких процессов​

import os​

from multiprocessing import Process ​

​

def doubler(number):​

result = number \* 2​

proc = os.getpid()​

print('{0} doubled to {1} by process id: {2}'.format(​

number, result, proc)) ​

​

if \_\_name\_\_ == '\_\_main\_\_':​

numbers = [5, 10, 15, 20, 25]​

procs = []​

​

for index, number in enumerate(numbers):​

proc = Process(target=doubler, args=(number,))​

procs.append(proc)​

proc.start()​

for proc in procs:​

proc.join()

Использование замков

from multiprocessing import Process, Lock ​

​

def printer(item, lock):​

""" Выводим то что передали​

"""​

lock.acquire()​

try:​

print(item)​

finally:​

lock.release() ​

​

if \_\_name\_\_ == '\_\_main\_\_':​

lock = Lock()​

items = ['tango', 'foxtrot', 10]​

​

for item in items:​

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

p.start()

Создание пула процессов

from multiprocessing import Pool​

​

​

def doubler(number):​

return number \* 2​

​

​

if \_\_name\_\_ == '\_\_main\_\_':​

numbers = [5, 10, 20]​

pool = Pool(processes=3)​

print(pool.map(doubler, numbers))

Использование очереди

from multiprocessing import Process, Queue ​

sentinel = -1 ​

def creator(data, q):​

print('Creating data and putting it on the queue')​

for item in data:​

q.put(item) ​

def my\_consumer(q):​

while True:​

data = q.get()​

print('data found to be processed: {}'.format(data)) ​

processed = data \* 2​

print(processed) ​

if data is sentinel:​

break ​

if \_\_name\_\_ == '\_\_main\_\_':​

q = Queue()​

data = [5, 10, 13, -1] ​

process\_one = Process(target=creator, args=(data, q))​

process\_two = Process(target=my\_consumer, args=(q,)) ​

process\_one.start()​

process\_two.start() ​

q.close()​

q.join\_thread() ​

process\_one.join()​

process\_two.join()

Модуль multiprocessing был добавлен в Python версии 2.6. Изначально он был определен в PEP 371 Джесси Ноллером и Ричардом Одкерком. Модуль multiprocessing позволяет вам создавать процессы таким же образом, как при создании потоков при помощи модуля threading. Суть в том, что, в связи с тем, что мы теперь создаем процессы, вы можете обойти GIL (Global Interpreter Lock) и воспользоваться возможностью использования нескольких процессоров на компьютере. Пакет multiprocessing также включает ряд API, которых вообще нет в модуле threading. Например, есть очень удобный класс Pool, который вы можете использовать для параллельного выполнения функции между несколькими входами.

**Особенности организации многопроцессорной программы в Python.**

Это использование двух или более процессорных блоков в одной компьютерной системе. Это лучший способ получить полный потенциал от нашего оборудования, используя полное количество процессорных ядер, доступных в нашей компьютерной системе.



**Устранение влияния глобальной блокировки интерпретатора (GIL)**

При работе с параллельными приложениями в Python есть ограничение, называемое GIL (Global Interpreter Lock) . GIL никогда не позволяет нам использовать несколько ядер CPU, и поэтому мы можем сказать, что в Python нет настоящих потоков. GIL – мьютекс – блокировка взаимного исключения, которая делает вещи безопасными. Другими словами, мы можем сказать, что GIL препятствует параллельному выполнению кода Python несколькими потоками. Блокировка может удерживаться только одним потоком за раз, и если мы хотим выполнить поток, он должен сначала получить блокировку.

Используя многопроцессорность, мы можем эффективно обойти ограничение, вызванное GIL –

* Используя многопроцессорность, мы используем возможности нескольких процессов и, следовательно, мы используем несколько экземпляров GIL.
* В связи с этим нет ограничений на выполнение байт-кода одного потока в наших программах одновременно.

Используя многопроцессорность, мы используем возможности нескольких процессов и, следовательно, мы используем несколько экземпляров GIL.

В связи с этим нет ограничений на выполнение байт-кода одного потока в наших программах одновременно.

## 22. Асинхронное программирование в Python. Использование asyncio.

Асинхронное программирование является некоторой моделью программирования, которая сосредоточена на координации различных задач в некотором приложении. Её цель состоит в предоставлении гарантии того, что данное приложение завершит исполнять такие задачи за наименьший возможный промежуток времени. С этой точки зрения асинхронное программирование состоит в переключении с одной задачи на другую когда это соответствует созданию совпадению между временем ожидания и временем обработки, а следовательно, сокращению общего времени, требуемого для завершения данной программы в целом.

Общие сведения

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.

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.​

​

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;​

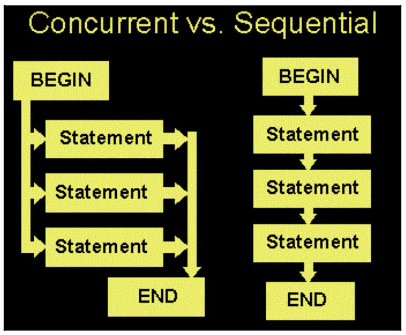
если сопрограмма встречает блокирующий код (I/O, sleep), текущая сопрограмма приостанавливается, и управление возвращается в event loop;​

event loop получает следующие задачи из очереди 2, ...n, затем event loop возвращается к задаче 1, с которой он был прерван.

## 

## 23. Параллельное программирование. Достоинства и недостатки.

Параллельное программирование - техника программирования, при которой используются все преимущества многоядерных или многопроцессорных систем. При этом параллельное программирование является одним из методов параллелизма, наравне с распределенным программированием.



Различие между параллельной и последовательной программой.

Преимущества:

· **Более быстрое время исполнения**: Одним из основных преимуществ параллельного программирования является достигаемое ускорение. Отдельные потоки в одной и той же программе могут исполняться совместно или параллельно, когда они достаточно независимы друг от друга.

· **Быстрота отклика**:

· **Эффективность в потреблении ресурсов**: программы могут совместно разделять одни и те же ресурсы и осуществлять к ним доступ. Вследствие этого, параллельные программы могут обслуживать и обрабатывать множество запросов клиентов к данным для совместной обработки, используя значительно меньше ресурсов.

**Минусы:**

· **Крушения**: отдельная недопустимая операция может отрицательно сказываться на имеющейся обработке программы и может вызывать в результате крушение всей программы целиком.

· **Синхронизация**

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

16. Понятие потокобезопасности. причины, проблематика, способы обеспечения.

Потоковая безопасность — это концепция программирования, применимая к многопоточным программам. Код потокобезопасен, если он функционирует исправно при использовании его из нескольких потоков одновременно. В частности, он должен обеспечивать правильный доступ нескольких потоков к разделяемым данным.

Способы обеспечения потокобезопасности.

1.Замки

lock = Lock()

lock.acquire()

... access shared resource

lock.release()

lock.acquire()

try:

... access shared resource

finally:

lock.release()

with lock:

... access shared resource

2.Дедлоки

import threading

l = threading.Lock()

print("before first acquire")

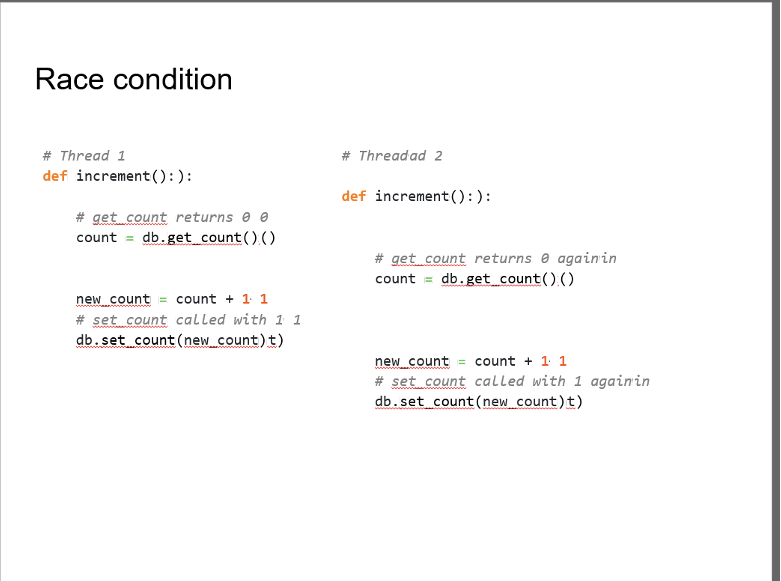
l.acquire()

print("before second acquire")

l.acquire()

print("acquired lock twice")

3.



Так же используеются семафоры, события и очереди и концепция

GIL.

## 25. Алгоритм выполнения многопоточной программы. Блокировка потоков.

- потоки управления (threads) образуются и работают в рамках одного процесса.

- в однопоточной программе используется только 1 главный поток (некоторая “нить” последовательно исполняемых операторов, которая начинается при запуске программы, является непрерывной и обрывается только по завершении программы)

- при использовании многопоточности, «нить» (главный поток) в некоторой точке программы делится на две, три и т.д. нити, которые в дальнейшем так же могут порождать другие потоки от себя. Но при этом **важно знать**, что один из потоков **всегда остается главным**, и его завершение означает завершение всей программы!

- в каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет кванты времени каждому потоку. Несмотря на внешнюю простоту такого механизма, деятельность потоков при этом должна быть полностью согласованна. Одновременная попытка изменения двумя и более потоками объекта скорее всего приведет к нарушению его целостности и другим ошибкам исполнения программы.

- Классическим средством согласования потоков являются **семафоры**. Самый простой – **lock(замок)**  или **mutex(взаимоисключения)**. Если в некоторой части программы нам необходимо сохранить целостность данных согласовав потоки, то логично использовать замки. Принцип их работы можно описать как эстафету с палочкой. Пока данными пользуется один поток, он забирает «эстафетную палочку» , в то время как следующий поток ожидает передачи «палочки» от первого потока, чтобы потом начать использовать те же данные.

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

Одновременная попытка изменения двумя и более потоками объекта в многопоточной программе скорее всего приведет к нарушению его целостности и другим ошибкам исполнения программы.

Поддержка многопоточности в языке Python доступна через использование ряда модулей. В стандартном модуле threading определены нужные для разработки многопоточной (multithreading) программы классы: несколько видов семафоров (классы замков Lock, RLock и класс Semaphore ) и другие механизмы взаимодействия между потоками (классы Event и Condition ), класс Timer для запуска функции по прошествии некоторого времени. Модуль Queue реализует очередь, которой могут пользоваться сразу несколько потоков.

**Замки** Простейший замок может быть реализован на основе класса Lock модуля threading. Замок имеет два состояния: он может быть или открыт, или заперт. В последнем случае им владеет некоторый поток. Объект класса Lock имеет следующие методы:

· acquire([blocking=True]) Делает запрос на запирание замка. Если параметр blocking не указан или является истиной, то поток будет ожидать освобождения замка. Если параметр не был задан, метод не возвратит значения. Если blocking был задан и истинен, метод возвратит True (после успешного овладения замком). Если блокировка не требуется (то есть задан blocking=False ), метод вернет True, если замок не был заперт и им успешно овладел данный поток. В противном случае будет возвращено False.

· release() Запрос на отпирание замка.

· locked() Возвращает текущее состояние замка ( True - заперт, False - открыт). Следует иметь в виду, что даже если состояние замка только что проверено, это не означает, что он сохранит это состояние до следующей команды.

Имеется еще один вариант замка - threading.RLock, который отличается от threading.Lock тем, что некоторый поток может запрашивать его запирание много раз. Отпирание такого замка должно происходить столько же раз, сколько было запираний. Это может быть полезно, например, внутри рекурсивных функций.

Замки позволяют ограничивать вход в некоторую область программы одним потоком. Замки могут потребоваться для обеспечения целостности структуры данных. Например, если для корректной работы программы требуется добавление определенного элемента сразу в несколько списков или словарей, такие операции в многопоточном приложении следует обставить замками. Вокруг атомарных операций над встроенными типами (операций, которые не вызывают исполнение какого-то другого кода на Python) замки ставить необязательно. Например, метод append() (встроенного) списка является атомарной операцией, а тот же метод, реализованный пользовательским классом, может требовать блокировок. В случае сомнений, конечно, лучше перестраховаться и поставить замки, однако следует минимизировать общее время действия замка, так как замок останавливает другие потоки, пытающиеся попасть в ту же область программы. Отсутствие замка в критической части программы, работающей над общими для двух и более потоков ресурсами, может привести к случайным, трудноуловимым ошибкам.

**Семафоры** являются более общим механизмом синхронизации потоков, нежели замки. Семафоры могут допустить в критическую область программы сразу несколько потоков. Семафор имеет счетчик запросов, уменьшающийся с каждым вызовом метода acquire() и увеличивающийся при каждом вызове release(). Счетчик не может стать меньше нуля, поэтому в таком состоянии потокам приходится ждать, как и в случае с замками, пока значение счетчика не увеличится.

Конструктор класса threading.Semaphore принимает в качестве (необязательного) аргумента начальное состояние счетчика (по умолчанию оно равно 1, что соответствует замку класса Lock ).

Методы acquire() и release() действуют аналогично описанным выше одноименным методам у замков. Семафор может применяться для охраны ограниченного ресурса. Например, с его помощью можно вести пул соединений с базой данных. Пример такого использования семафора (заимствован из документации к Python) дан ниже: from threading import BoundedSemaphore maxconnections = 5 # Подготовка семафора pool\_sema = BoundedSemaphore(value=maxconnections) # Внутри потока: pool\_sema.acquire() conn = connectdb() # ... использование соединения ... conn.close() pool\_sema.release() Таким образом, применяется не более пяти соединений с базой данных. В примере использован класс threading.BoundedSemaphore. Экземпляры этого класса отличаются от экземпляров класса threading.Semaphore тем, что не дают сделать release() больше, чем сделан acquire().

**События.** Еще одним способом коммуникации между объектами являются события. Экземпляры класса threading.Event могут быть использованы для передачи информации о наступлении некоторого события от одного потока одному или нескольким другим потокам. Объекты-события имеют внутренний флаг, который может находиться в установленном или сброшенном состоянии. При своем создании флаг события находится в сброшенном состоянии. Если флаг в установленном состоянии, ожидания не происходит: поток, вызвавший метод wait() для ожидания события, просто продолжает свою работу. Ниже приведены методы экземпляров класса threading.Event:

· set() Устанавливает внутренний флаг, сигнализирующий о наступлении события. Все ждущие данного события потоки выходят из состояния ожидания.

· clear() Сбрасывает флаг. Все события, которые вызывают метод wait() этого объекта-события, будут находиться в состоянии ожидания до тех пор, пока флаг сброшен, или по истечении заданного таймаута.

· isSet() Возвращает состояние флага.

· wait([timeout]) Переводит поток в состояние ожидания, если флаг сброшен, и сразу возвращается, если флаг установлен. Аргумент timeout задает таймаут в секундах, по истечении которого ожидание прекращается, даже если событие не наступило. Составить пример работы с событиями предлагается в качестве упражнения.

**Условия.**

Более сложным механизмом коммуникации между потоками является механизм условий. Условия представляются в виде экземпляров класса threading.Condition и, подобно только что рассмотренным событиям, оповещают потоки об изменении некоторого состояния. Конструктор класса threading.Condition принимает необязательный параметр, задающий замок класса threading.Lock или threading.RLock. По умолчанию создается новый экземпляр замка класса threading.RLock. Методы объекта-условия описаны ниже:

· acquire(...) Запрашивает замок. Фактически вызывается одноименный метод принадлежащего объекту-условию объекта-замка.

· release() Снимает замок.

· wait([timeout]) Переводит поток в режим ожидания. Этот метод может быть вызван только в том случае, если вызывающий его поток получил замок. Метод снимает замок и блокирует поток до появления объявлений, то есть вызовов методов notify() и notifyAll() другими потоками. Необязательный аргумент timeout задает таймаут ожидания в секундах. При выходе из ожидания поток снова запрашивает замок и возвращается из метода wait().

· notify() Выводит из режима ожидания один из потоков, ожидающих данные условия. Метод можно вызвать, только овладев замком, ассоциированным с условием. Документация предупреждает, что в будущих реализациях модуля из целей оптимизации этот метод будет прерывать ожидание сразу нескольких потоков. Сам по себе метод notify() не приводит к продолжению выполнения ожидавших условия потоков, так как этому препятствует занятый замок. Потоки получают управление только после снятия замка потоком, вызвавшим метод notify().

· notifyAll() Этот метод аналогичен методу notify(), но прерывает ожидание всех ждущих выполнения условия потоков.

В следующем примере условия используются для оповещения потоков о прибытии новой порции данных (организуется связь производитель - потребитель, producer - consumer):

import threading

cv = threading.Condition()

class Item:

"""Класс-контейнер для элементов, которые будут потребляться в потоках"""

def \_\_init\_\_(self):

self.\_items = []

def is\_available(self):

return len(self.\_items) > 0

def get(self):

return self.\_items.pop()

def make(self, i):

self.\_items.append(i)

item = Item()

def consume():

"""Потребление очередного элемента (с ожиданием его появления)"""

cv.acquire()

while not item.is\_available():

cv.wait()

it = item.get()

cv.release()

return it

def consumer():

while True:

print consume()

def produce(i):

"""Занесение нового элемента в контейнер и оповещение потоков"""

cv.acquire()

item.make(i)

cv.notify()

cv.release()

p1 = threading.Thread(target=consumer, name="t1")

p1.setDaemon(True)

p2 = threading.Thread(target=consumer, name="t2")

p2.setDaemon(True)

p1.start()

p2.start()

produce("ITEM1")

produce("ITEM2")

produce("ITEM3")

produce("ITEM4")

p1.join()

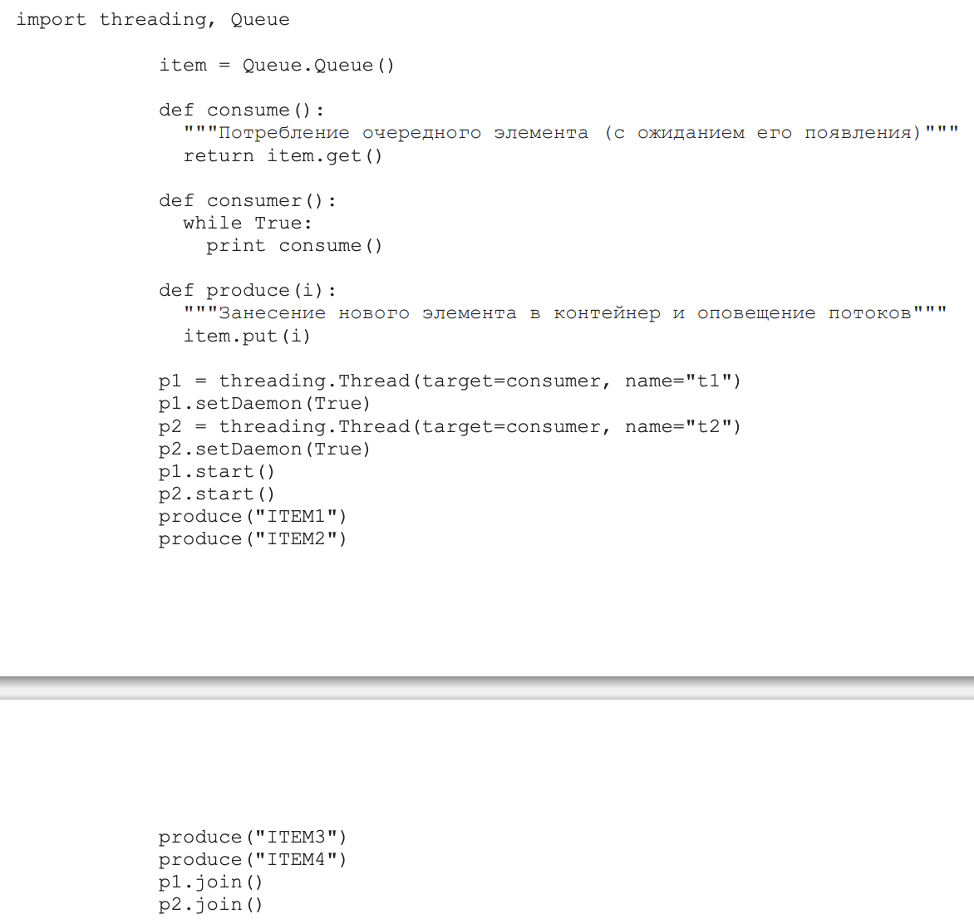
p2.join()

В этом примере условие cv отражает наличие необработанных элементов в контейнере item. Функция produce() "производит" элементы, а consume(), работающая внутри потоков, "потребляет". Стоит отметить, что в приведенном виде программа никогда не закончится, так как имеет бесконечный цикл в потоках, а в главном потоке - ожидание завершения этих потоков. Еще одна особенность - признак демона, установленный с помощью метода setDaemon() объекта-потока до его старта.

**Очередь.** Процесс, показанный в предыдущем примере, имеет значение, достойное отдельного модуля. Такой модуль в стандартной библиотеке языка Python есть, и он называется Queue.

Помимо исключений - Queue.Full (очередь переполнена) и Queue.Empty (очередь пуста) - модуль определяет класс Queue, заведующий собственно очередью.

Собственно, здесь можно привести аналог примера выше, но уже с использованием класса Queue.Queue:



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

Модуль OS в python предоставляет функции для взаимодействия с операционной системой. OS, поставляется под стандартные служебные модули Python. Этот модуль предоставляет портативный способ использования функциональных возможностей, зависящих от операционной системы. Модули \* os \* и \* os.path \* включают в себя множество функций для взаимодействия с файловой системой.

**os.name** - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

**os.environ** - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

**os.getlogin**() - имя пользователя, вошедшего в терминал (Unix).

**os.getpid**() - текущий id процесса.

**os.uname**() - информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется реализацией), release - релиз, version - версия, machine - идентификатор машины.

**os.access**(path, mode, \*, dir\_fd=None, effective\_ids=False, follow\_symlinks=True) - проверка доступа к объекту у текущего пользователя. Флаги: **os.F\_OK** - объект существует, **os.R\_OK** - доступен на чтение, **os.W\_OK** - доступен на запись, **os.X\_OK** - доступен на исполнение.

**os.chdir**(path) - смена текущей директории.

**os.chmod**(path, mode, \*, dir\_fd=None, follow\_symlinks=True) - смена прав доступа к объекту (mode - восьмеричное число).

**os.chown**(path, uid, gid, \*, dir\_fd=None, follow\_symlinks=True) - меняет id владельца и группы (Unix).

**os.getcwd**() - текущая рабочая директория.

**os.link**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None, follow\_symlinks=True) - создаёт жёсткую ссылку.

**os.listdir**(path=".") - список файлов и директорий в папке.

**os.mkdir**(path, mode=0o777, \*, dir\_fd=None) - создаёт директорию. OSError, если директория существует.

**os.makedirs**(path, mode=0o777, exist\_ok=False) - создаёт директорию, создавая при этом промежуточные директории.

**os.remove**(path, \*, dir\_fd=None) - удаляет путь к файлу.

**os.rename**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает файл или директорию из src в dst.

**os.renames**(old, new) - переименовывает old в new, создавая промежуточные директории.

**os.replace**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает из src в dst с принудительной заменой.

**os.rmdir**(path, \*, dir\_fd=None) - удаляет пустую директорию.

**os.removedirs(path)** - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

**os.symlink**(source, link\_name, target\_is\_directory=False, \*, dir\_fd=None) - создаёт символическую ссылку на объект.

**os.sync**() - записывает все данные на диск (Unix).

**os.truncate**(path, length) - обрезает файл до длины length.

**os.utime**(path, times=None, \*, ns=None, dir\_fd=None, follow\_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

**os.walk**(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

**os.system**(command) - исполняет системную команду, возвращает код её завершения (в случае успеха 0).

**os.urandom**(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

[os.path](https://pythonworld.ru/moduli/modul-os-path.html) - модуль, реализующий некоторые полезные функции на работы с путями.

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

**Веб-технологии** — это коммуникационные, информационные и иные технологии и сервисы, основываясь на которые осуществляется деятельность в Интернете или с помощью него. В первую очередь, это, конечно, сайты, а также: чаты, почта, Интернет-магазины, форумы и т.д.

Веб–технологии — это логическая составляющая Интернет-технологий, которые включают в себя:

**1) Интернет-сервисы**

● WWW — Всемирная паутина

**2) Работа в Интернет**

● Браузеры.

● Поисковые системы.

● Просмотр страниц в браузере.

**3) Информационные ресурсы Интернет**

● Веб-страницы, интернет-магазины, интернет-порталы.

● URL и протоколы передачи данных, адресация.

● Создание сайтов.

● Языки Веб-программирования.

**Основные характеристики**

· Работает WWW по принципу клиент- серверы. То есть клиент делает запрос на серверы, серверы выдают клиенту гипермедийный текст (тексты, фото, видео, звук).

· Основные составляющие при работе с WWW: HTML (специальный язык разметки гипертекста), URL (адресная ссылка на ресурсы и их части: протокол, хост, порт, путь до ресурса, запрос), HTTP (протокол передачи гипертекста).

**История**

С **1962** года министерство обороны США активно вкладывало деньги в разработки технологий взаимодействия между компьютерами. Благодаря этим исследованиям, в **1969**году, американскими военными был создана локальная сеть - предшественница Интернета. С 1972 года функционирует всеми нами любимая электронная почта.

Переломным этапом в развитии интернет-технологий стали 90-е годы. С изобретением в **1993 году первого браузера**, предполагаемые возможности Интернета стали приобретать глобальный характер. Появились многочисленные телекоммуникационные операторы, компании электронной торговли, разработчики программного обеспечения и т.п.

**Назначение**

Позволяет пользователю разместить, найти, обрабатывать практически любую информацию в сети Интернет

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

Создание web-сайтов и других web-технологий, а также их поддержка и развитие осуществляется с помощью специализированного ПО.



**Серверные ОС**

Назначение серверной операционной системы: под управлением этих ОС выполняются приложения, обслуживающие всех пользователей корпоративной сети, а нередко и внешних пользователей. К таким приложениям относятся современные системы управления базами данных, средства управления сетями и анализа событий в сети, службы каталогов, средства обмена сообщениями и групповой работы, Web-серверы, почтовые серверы, серверы приложений разнообразного назначения.

*Примеры серверных ОС*: Solaris, Linux, MS Windows Server, HP UX.

**Web-браузер**

Должно быть ПО, которое сможет запросить информацию, получить ее, обработать и отобразить на экран пользователя. Именно этим и занимается браузер.

Браузер - окно просмотра, программа для просмотра текстового и графического содержания (контента) web-ресурсов Интернет, проигрывания мультимедийной информации (звука, видео, виртуальной реальности), передачи управляющей, почтовой, форм-интерактивной и иных форм информации от пользователя Интернет, поиска информации в Интернет с помощью поисковых систем.

Происходит это с помощью **http запросов** к серверу и получения от него данных, которые обрабатываются по специальным утвержденным стандартам и таким образом формируется веб-страница.

Браузер представляет собой компьютерную программу, - так называемый "тонкий клиент", - в системе клиент-серверной технологии современного сетевого обмена информацией, когда "тонкому клиенту" делегируется лишь незначительная часть полномочий по управлению информационным процессом, не позволяя влиять на жизненно важные его элементы, от которых зависит работоспособность информационной системы.

**Web-сервер**

Чтобы опубликовать веб-сайт, нужен либо статический, либо динамический веб-сервер.

***Как он работает***

Простыми словами, когда браузеру нужен файл, размещенный на веб-сервере, браузер запрашивает его через HTTP. Когда запрос достигает нужного веб-сервера (железо), сервер HTTP (ПО) передает запрашиваемый документ обратно, также через HTTP.

1. При получении запроса, HTTP сервер сначала проверяет существует ли ресурс по данному URL.
2. Если это так, веб-сервер отправляет содержимое файла обратно в браузер. Если нет, сервер приложений создает необходимый ресурс.
3. Если это невозможно, веб-сервер возвращает сообщение об ошибке в браузер, чаще всего «404 Not Found». (Эта ошибка настолько распространена, что многие веб-дизайнеры тратят большое количество времени на разработку 404 страниц об ошибках.)

***Примеры:*** Apache, Nginx, Microsoft-IIS

Apache и Nginx — 2 самых широко распространенных веб-сервера с открытым исходным кодом в мире. Вместе они обслуживают более 50% трафика во всем интернете. Оба решения способны работать с разнообразными рабочими нагрузками и взаимодействовать с другими приложениями для реализации полного веб-стека.

***Преимущество Apache:***

Администраторы часто выбирают Apache из-за его гибкости, мощности и широкой распространенности. Он может быть расширен с помощью системы динамически загружаемых модулей и исполнять программы на большом количестве интерпретируемых языков программирования без использования внешнего программного обеспечения.

***Преимущество Nginx:***

Администраторы часто выбирают Nginx из-за его эффективного потребления ресурсов и отзывчивости под нагрузкой, а также из-за возможности использовать его и как веб-сервер, и как прокси.

**Серверы приложений**

Предназначены выполнять обширный набор специализированных функций — от обеспечения гибких средств электронных коммуникаций до управления реляционными базами данных.ё

· *Пример: WebSphere Application Server, Tomcat*

**Tomcat** позволяет запускать [веб-приложения](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) и содержит ряд программ для самоконфигурирования.

**Tomcat** используется в качестве самостоятельного [веб-сервера](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80), в качестве сервера контента в сочетании с веб-сервером [Apache HTTP Server](https://ru.wikipedia.org/wiki/Apache_HTTP_Server), а также в качестве контейнера сервлетов в [серверах приложений](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) [JBoss](https://ru.wikipedia.org/wiki/JBoss) и [GlassFish](https://ru.wikipedia.org/wiki/GlassFish).

## 30. Понятие URL: назначение, применение, состав.

**URL** – это адрес документа или файла. Аббревиатура URL составлена из начальных букв исходного сочетания Uniform Resource Locator, что в переводе с английского языка означает понятие Единый указатель ресурсов.

Другими словами, он определяет местонахождение ресурса либо документа в сети, его сетевой локатор, ссылка на него.

Каждая интернет-страница расположена на собственном уникальном адресе, который пользователь может видеть в специальном поле – адресной строке браузера – он и указывает, где хранятся эти файлы в сети. И урл есть не что иное, как общепринятая стандартная форма записи этого местонахождения во Всемирной паутине.

Чтобы осуществлять взаимодействие в глобальной сети, обмениваться данными, иметь доступ к искомому URL веб-ресурса, используют протоколы – так называется свод правил и очередности действий, который позволяет устанавливать соединение устройств в Интернете.

Практически мгновенная предоставление информации посредством Всемирной паутины обеспечивается благодаря поддержке протоколами TCP/IP, ATM, MPLS и прочими.

Самым распространенным среди них является набор протоколов передачи данных TCP/IP.

В их числе наибольшей популярностью в сети пользуются:

* HTTP — это протокол передачи гипертекста. Он применяется при обмене данными с серверов – подключенному к Интернету компьютеру пользователя по его запросу;
* FTP — протокол, назначение которого — передача документов с сервера на персональное устройство пользователя. FTP позволяет обмениваться документами любым компьютерам сети.

Каждый URL **состоит** из определенной последовательности символов. Это не хаотичный набор знаков, а определенная структура, которая составляется по четкой схеме.

По задумке создателей, сетевой локатор необходимо максимально упростить, чтобы тот включал в себя небольшое число символов.

Его составляющие в ниже представленной последовательности могут быть следующими:

способ обмена данными с ресурсом или тип сетевого протокола;

логин, то есть то имя учетной записи пользователя для обращения к ресурсу;

пароль для входа в учетную запись;

хост или целиком вписанное доменное имя;

порт для подключения в пределах обозначенного хоста;

URL-путь – содержит конкретизирующую информацию о заданных характеристиках места расположения ресурса;

строка запроса, содержащая передаваемые на сервер параметры;

«якорь», в функцию которого входит возможность ссылаться на определенную часть документа, страницы.

Такая последовательность указывает компьютерной системе, где конкретно хранится необходимый пользователю ресурс и путь к нему.

Если на заре создания URL служил для определения пути к определенным файлам, то в настоящее время посредством ссылок обозначается расположение любого типа информации:

* интернет-ресурсов;
* веб-страниц;
* изображений;
* видео.

Например, чтобы просмотреть видео на сайте, компьютер обращается к точному адресу места хранения документа. Если файл удалятся правообладателем или URL изменяют, просмотреть данное видео пользователь не сможет.

Применение идентификатора URL дает возможность создавать комфортные, понятные для восприятия пользователя конструкции символов. Благодаря чему в этом случае исчезает необходимость использовать программные коды.

Вместе с тем помимо этих достоинств существуют некоторые ограничения. В URL разрешается пользоваться определенным набором знаков:

* латиница (или символы английского алфавита);
* цифры длиной до 225 символов;
* определенные знаки пунктуации (дефис, нижнее подчеркивание);
* специальные символы (в частности, «/» — слеш, «\_» — нижнее подчеркивание).

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

Веб-сервер — это сложный программный комплекс, реализующий огромный набор функций и поддерживающий все нюансы современных версий протокола HTTP.

Веб-сервер – это программа, которая принимает входящие HTTP-запросы, обрабатывает эти запросы, генерирует HTTP-ответ и отправляет его клиенту. – это и есть **принцип работы**.

Веб-сервер (web-server) – это сервер, отвечающий за прием и обработку запросов (HTTP-запросов) от клиентов к веб-сайту. В качестве клиентов обычно выступают различные веб-браузеры. В ответ веб-сервер выдает клиентам HTTP-ответы, в большинстве случаев – вместе с HTML-страницей, которая может содержать: всевозможные файлы, изображения, медиа-поток или любые другие данные.

Также веб-сервер выполняет функцию исполнения скриптов, например, таких как CGI, JSP, ASP и PHP, которые отвечают за организацию запросов к сетевым службам, базам данных, доступу к файлам, пересылке электронной почты и другим приложениям электронной коммерции.

Термин “веб-сервер” также применяется к техническим устройствам и программному обеспечению, которые выполняют функции веб-сервера. Это может быть какой-нибудь компьютер, который специально выделен из группы персональных компьютеров или рабочая станция, на которых установлено и работает сервисное программное обеспечение.

В случае малой организации веб-сервер может быть целостной системой, которая будет состоять из: HTTP-сервера – служит для запросов к веб-страницам; FTP-сервера – применяется для загрузки файлов через Интернет; NNTP-сервера – выполняет доступ к группам новостей; SMTP-сервера – для электронной почты.

**Все основные и дополнительные цели(функции) веб-сервера:**

· Прием запросов от веб-браузеров по протоколу стандарта HTTP с использованием сетевых протоколов TCP/IP;

· Выполнение поиска и отсылки файлов с гипертекстом или каких-либо документов в браузер по протоколу HTTP;

· Обслуживание и обработка запросов, типа: mailto, FTP, Telnet и т. п.;

· Запуск прикладных программ на веб-сервере с последующей передачей и возвратом параметров обработки через стандарт интерфейса CGI;

· Работа и обслуживание навигационных карт изображений (Image map);

· Загрузка Java-приложений;

· Администрация и оперативное управление сервером;

· Авторизация пользователей и их аутентификация;

· Ведение регистрационного журнала обращений пользователей к различным ресурсам;

· Автоматизированная работа веб-страниц;

· Поддержка страниц, которые генерируются динамически;

· Поддержка работы протокола HTTPS для защищенных соединений с клиентами.

**Описание (принцип) работы веб-сервера**

Веб-браузеры поддерживают связь с веб-серверами с помощью протокола передачи гипертекстовых сообщений (HypertextTransferProtocol, HTTP). Это простой протокол запросов и ответов для пересылки информации с использованием протокола TCP/IP. Веб-сервер получает запрос, обнаруживает файл, посылает его браузеру, а затем разрывает соединение. Графическая информация, которая имеется на странице, обрабатывается таким же образом. Далее настает очередь веб-браузера – вывести на монитор пользователя загруженный из сети HTML-документ.

Кроме HTML-страниц и графики, веб-серверы могут хранить любые файлы, в том числе текстовые документы, документы текстовых процессоров, видеофайлы и аудиоинформацию. На сегодняшний день, если не учитывать анкет, которые заполняют пользователи, основная часть веб-трафика передается в одном направлении – браузеры считывают файлы с веб-сервера. Но это положение изменится после общего принятия описанного в проекте HTTP 1.1 метода PUT, который позволяет записывать файлы на веб-сервер. Сегодня метод PUT используется в основном пользователями, создающими веб-страницы, но в перспективе он может пригодиться и остальным пользователям для обратной связи с информационными центрами. Запросы методом PUT намного проще, чем обыкновенная POST загрузка файлов на веб-сервер.

На веб-сервере также выполняют свою работу различные приложения, наибольшую популярность среди которых получили поисковики и средства связи с базами данных. Для разработки этих приложений применяются такие стандарты, как общий шлюзовой интерфейс (CommonGatewayInterface, CGI), языки сценариев JavaScript, а также языки программирования Java и VisualBasic. Кроме интерфейса стандарта CGI, некоторые фирмы-разработчики веб-серверов создали интерфейсы прикладного программирования (API) такие как, например, Netscape Server API и Internet Server API, которые созданы компаниями Microsoft и Process Software AG. Эти интерфейсы позволяют разработчикам непосредственно обращаться к конкретным функциям веб-сервера. Некоторые веб-серверы обладают связующим программным обеспечением (middleware) для подключения к базам данных, работа с которыми может потребовать профессиональных знаний в программировании.

Базовые функции поиска помогают пользователям отсортировывать нужную им информацию, а утилиты для связи с базами данных предоставляют пользователям веб-браузеров доступ к этой информации.

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

HTTP (HyperText Transfer Protocol) – это протокол [прикладного уровня](https://en.wikipedia.org/wiki/Application_Layer) для передачи гипермедиа документов, таких как HTML. Он был разработан для связи между веб-браузерами и веб-серверами, но также может использоваться для других целей. HTTP следует классической [модели клиент-сервер:](https://en.wikipedia.org/wiki/Client%E2%80%93server_model) клиент открывает соединение, чтобы сделать запрос, а затем ждет, пока не получит ответ.

URL & URI





- URL

Типичные задачи, которые решает **HTTP протокол**: протокол HTTP осуществляет доступ к веб-ресурсам и обмен данными между пользовательскими приложениями.

**Передача данных по HTTP протоколу** осуществляется через TCP/IP соединение. Машина, которая выступает в роли сервера использует восьмидесятый TCP порт или порт 8080.

Функции, управляемые с помощью HTTP:

1) Кеширование

2) Аутентификация

3) Proxy

4) *Ослабление ограничения источника.*

*5) Сеансы*

**Структура протокола.**

Структура протокола определяет, что каждое HTTP-сообщение состоит из трёх частей, которые передаются в следующем порядке:

1. Запрос — определяет тип сообщения;
2. Поля запроса — характеризуют тело сообщения, параметры передачи и прочие сведения;
3. Тело сообщения (англ. Message Body) — непосредственно данные сообщения. Обязательно должно отделяться от заголовков пустой строкой.

Структура http-запроса:

Запрос – GET/wiki/ HTTP/1.1 Статусная строка: GET/ HTTP/1.1 (get – метод, / - путь, перед HTTP/1.1 (версия запроса) всегда пробел ).

Поля запроса:

Host:ru.wikipedia.org

User-Agent:Mozila/5.0

Gecko/8080 - port

Connection: close

(пустая строка)

**Методы протокола**

Метод HTTP (англ. HTTP Method) — последовательность из любых символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно метод представляет собой короткое английское слово, записанное заглавными буквами. Названия метода чувствительны к регистру.

HTTP [метод](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods) , как правило , имеет вид - [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET), [POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST) , [OPTIONS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS) или [HEAD](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD) что определяет операцию клиент хочет выполнить. Как правило, клиент хочет получить ресурс (используя GET) или опубликовать значение [HTML-формы](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms) (используя POST), хотя в других случаях может потребоваться больше операций.

Примеры методов протокола:

1) GET

2) POST (Запостить файл – создать файл)

3) PUT

4) DELETE (удалить) и др.

**Код состояния** информирует клиента о результатах выполнения запроса и определяет его дальнейшее поведение. Набор кодов состояния является стандартом, и все они описаны в соответствующих документах RFC.

Каждый код представляется целым трехзначным числом. Первая цифра указывает на класс состояния, последующие - порядковый номер состояния. За кодом ответа обычно следует краткое описание на английском языке.

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

1. Установка

apt install apache2

После завершения установки откроем браузер и перейдем по ссылке “http://[ip\_адрес\_сервера]”

2. Создание тестовой страницы

По умолчанию корневым каталогом для размещения сайта является директория “/var/www/html”, именно там находится страница приветствия. Создадим отдельную директорию “/var/www/sites” для размещения виртуальных хостов и вложенную папку “/var/www/sites/site1” с индексной страницей тестового сайта.

cd /var/www/

mkdir -p sites/site1

echo "<H1>Welcome</H1>" > sites/site1/index.html

В результате файл “/var/www/sites/site1/index.html” будет содержать одну html-строку:

<H1>Welcome</H1>

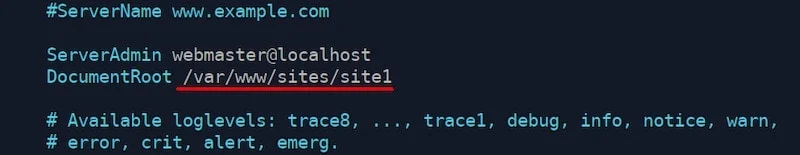
3. Конфигурация Apache-сервера

Конфигурационные файлы сайтов находятся в каталоге “/etc/apache2/sites-available/”. Создадим конфигурационный файл для нового виртуального хоста взяв за основу конфигурацию по умолчанию из файла “000-default.conf”

cd /etc/apache2/sites-available/

cp 000-default.conf site1.conf

Откроем файл “site1.conf” и изменим параметр “DocumentRoot”. В качестве значения нужно указать путь к новому сайту, в нашем случае это “/var/www/sites/site1”



На данном этапе нам не требуется настройка одновременной работы нескольких сайтов, поэтому отключим сайт по умолчанию и включим новый сайт. Для применения изменений перезагружаем конфигурацию сервера.

a2dissite 000-default

a2ensite site1

systemctl reload apache2

Снова переходим по ссылке “http://[ip\_адрес\_сервера]” и убеждаемся, что вместо стандартной страницы приветствия отображается наша новая страница.

Настройка HTTP-сервера завершена.

В зависимости от версии Apache и вашего дистрибутива, **конфигурационные файлы** Apache могут находиться в следующих каталогах: /etc/apache, /etc/apache2, /etc/httpd или /etc/httpd2. Основные конфигурационные файлы называются httpd.conf, httpd2.conf или apache.conf и apache2.conf.

Первым делом откройте конфигурационный файл и найдите директиву:

**#ServerName new.host.name**

Нужно ее раскомментировать и указать имя сервера, которое будут задавать пользователи в строке браузера. Данное имя должно быть зарегистрировано в DNS-сервере вашей сети (или указано в файле /etc/hosts каждого компьютера сети). Обычно здесь указывается имя компьютера, например: **ServerName user-desktop**

После этого можно будет обращаться к серверу по адресу <http://user-desktop/>.

**Основные директивы и их описание**

**ServerName** имя - задает имя Web-сервера, имя должно быть зарегистрированным на DNS-сервере, то есть обычно это доменное имя сервера

**ServerAdmin** e-mail - задает e-mail администратора сервера

**ServerRoot** каталог - определяет каталог с конфигурационными файлами сервера

**DocumentRoot** каталог - Позволяет задать каталог, в котором хранятся документы Web-сервера - это корневой каталог документов.

**ErrorLog файл** - задает журнал ошибок

**User пользователь**

**Group группа**

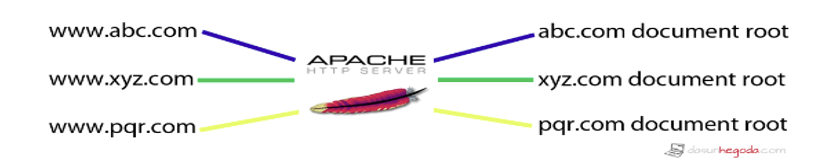
## 34. Виртуальные хосты. Применение, настройка.

Веб хостинг – услуга, предост. ресурсов для размещения информации на сервере, постоянно, находящихся в сети. – размещение сайтов и приложений.

Базовая часть, которая отвечает за отдельный сайт или домен называется виртуальным хостом.

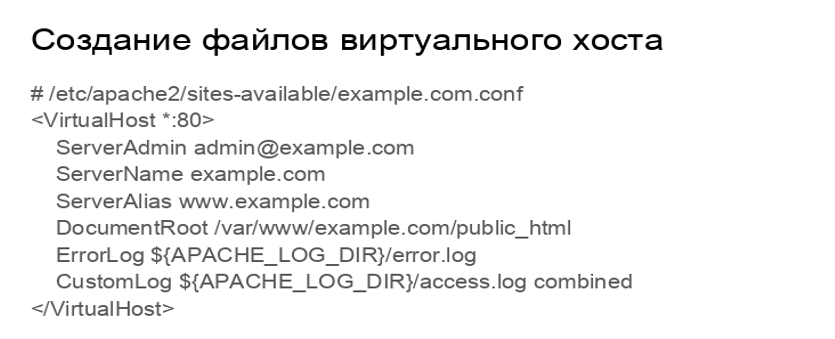
Эта система позволяет использовать один сервер, чтобы раздавать несколько сайтов используя один интерфейс или IP.

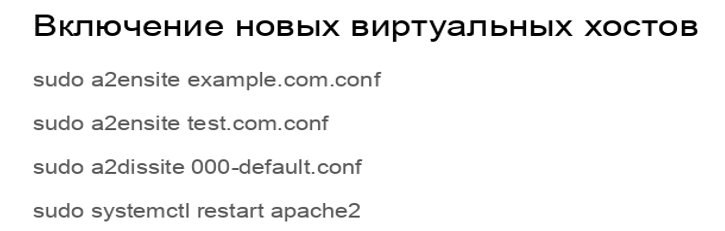
Каждый настроенный соответствующим образом домен будет направлять пользователя к определенной директории сервера.

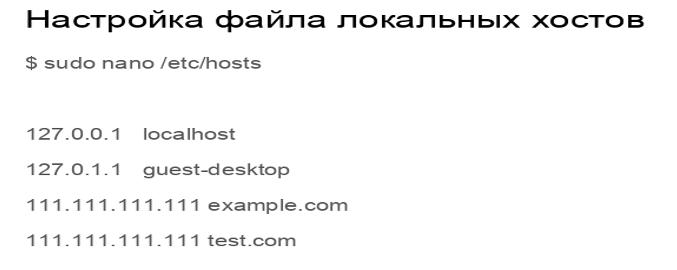


**Настройка:**

****

****

****

****

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

Рroxy-сервер – это дополнительная служба в сети интернет, позволяющая пользователям выполнять косвенные запросы к различным веб-ресурсам и сетевым службам. Данный ресурс выступает в качестве посредника, причем на оперативности и качестве работы это отражается только в лучшую сторону, за счет использования отдельного оптоволокна (выделенного канала) с максимально возможной скоростью обмена данными.

**Основные и наиболее часто используемые возможности прокси-серверов:**

· Ускорение работы с интернет-ресурсами за счет кэширования (большинство контента, фото- и видеоматериала уже загружено на дополнительном промежуточном сервере).

· Защита программного обеспечения компьютера пользователя от всевозможных сетевых атак (вирусов).

· Анонимность посещения всех интернет ресурсов. При использовании proxy-серверов все запросы на веб-ресурсы будут проходить под IP адресом прокси-сервера и информация о пользователе (страна, регион, провайдер, номер компьютера и адрес) будет завуалирована.

· Открытый доступ пользователя к социальным сетям, чатам и сайтам, даже если аккаунт заблокирован системным администратором (модератором) на уровне IP-адреса. Система будет воспринимать данный запрос к доступу как от нового пользователя, так как IP адрес будет изменен.

· Доступ к информации на сайтах с ограничениями на просмотр. Некоторые сайты могут устанавливать блокировку по определенному критерию или географическому положению. Пользователю достаточно подключиться к прокси-серверу данной страны, и система проверки IP адреса не распознает подмены, так как IP адрес будет соответствовать географическому положению.

· Фильтрация рекламы (автоматическое удаление рекламного контента с просматриваемых страниц).

**Классификация Proxy-серверов**

Условно прокси-серверы можно классифицировать на:

· FTP-proxy, которые работают непосредственно с FTP. Используются исключительно для доступа к определенному ресурсу.

· ProxySocks (4/5) сервис используется для обработки любого контента, но к сожалению, не все компьютерные программы его поддерживают.

· Proxy http – активно используется пользователями для скачивания файлов и просмотра веб-страниц с любым контентом (поддерживается большинством компьютерных программ и различными браузерами).

· Proxi CGI (анонимайзеры) – предназначены для работы с веб-браузерами.

Во всемирной сети интернет существуют как платные, так и бесплатные прокси-серверы. Безусловно, у каждого сервера есть свои достоинства и недостатки. Так, например, бесплатные прокси-серверы обеспечивают высокую скорость доступа к интернет ресурсам, но при этом они не анонимны, ведь их принцип действия основан на кэшировании информации. Все бесплатные дополнительные ресурсы существуют сравнительно короткий период, а потом либо переходят на коммерческую основу или исчезают. Бесплатную базу proxy-серверов никто не чистит и не обновляет, поэтому примерно 70% сервисов неактивные. Платные прокси не имеют всех вышеперечисленных недостатков, но их минус – достаточно высокая абонплата за пользование услугой.

**Понятие Прокси-сервера**

Прокси-сервер — удаленный компьютер, который, при подключении к нему вашей машины, становится посредником для выхода абонента в интернет. Прокси передает все запросы программ пользователя в сеть, и, получив ответ, отправляет его обратно.

Каждому компьютеру, подключенному к интернет, присваивается уникальный ip-адрес, который несет информацию о стране и регионе абонента, номере его провайдера и номере компьютера в сети.

Прокси-сервер также имеет свой ip-адрес. Подключившись к прокси, вы передаете все запросы в интернет через него, при этом проверка показывает ip прокси-сервера, а вы остаетесь анонимным.

**Настройка nginx**

Nginx доступен в стандартных репозиториях ubuntu, а установить его мы сможем при помощи пакетов APT. Для начала обновим список пакетов apt, а потом установим nginx:

sudo apt update && sudo apt install nginx

Далее необходимо настроить файрвол. При установки Nginx регистрируется в сервисе файрвола ufw. Поэтому настройка доступа осуществляется достаточно просто. Для вывода настроек доступа введем команду

sudo ufw app list

В результате будет выведен список профилей приложений:

Nginx Full

Nginx HTTP

Nginx HTTPS

Выбираем нужный, и пишем команду:

sudo ufw allow 'Nginx HTTP'

Далее ubuntu запустит nginx автоматически. Убедимся, введя команду

systemctl status nginx

Таким образом, сервер запущен и работает. Тем не менее, лучше проверить его работоспособность путем запроса веб-страницы.

Для остановки, старта, и перезапуска сервера необходимо ввести следующие команды:

sudo systemctl stop nginx

sudo systemctl start nginx

sudo systemctl restart nginx

## 36. Основные понятия криптографических методов защиты информации

Шифрование - преобразовательный процесс: исходный текст, который носит также название открытого текста, заменяется шифрованным текстом.

Дешифрование - обратный шифрованию процесс. На основе ключа шифрованный текст преобразуется в исходный.

Криптографические методы защиты информации - это специальные методы шифрования, кодирования или иного преобразования информации, в результате которого ее содержание становится недоступным без предъявления ключа криптограммы и обратного преобразования.

**Принцип криптографии**

Использование криптографической системы помогает зашифровать изначальный текст (открытый текст), чтобы в конечном итоге получилась совершенно другой текст (шифртекст или криптограмма), смысл которого сможет понять только тот, для кого этот текст предназначался. Причем, получатель должен полученный текст дешифровать, чтобы понять смысл исходного текста. Кроме того, если шифртекст попадет в руки третьего лица, который не должен был ее получить, то он должен быть не способен ее дешифровать.

Надежность систем защиты информации - это очень сложный и запутанный вопрос. Проблема заключается в том, что почти невозможно проверить степень защищенности информации. Кроме того, проблема криптографии в том, что порой на дешифрование необходимо затратить намного больше ресурсов и усилий, нежели на процесс шифрования. Но не исключено, что какой-нибудь школьник не сможет случайным образом расшифровать ваш шифртекст. Ведь современные дети обладают необычайным складом ума и логики.

Несмотря на существенные успехи математики за пару тысячелетий, тайнопись не совершила значительных улучшений вплоть до середины 20 века. К примеру, в 20 веке в качестве шифра часто использовали популярное печатное издание, дабы можно было спрятать шифртекст у всех на виду, но никто бы и не подумал о передаче тайной информации.

**Задачи:**

· Обеспечение секретности

· Аутентификация сторон

· Обеспечение целостности

**Шифры:**

**Шифрование** – процесс **криптографического** преобразования текста на основе определенного параметра (ключа) и алгоритма. **Расшифрование** – **криптографическое** преобразование шифрованного текста в исходный. **Шифр** – совокупность обратимых преобразований исходных данных в скрытый текст.

**Виды:**

Симметричное шифрование (шифрование с закрытым ключом)

Есть ключ шифрования. С его помощью данные шифруются по какому-то алгоритму. Тот, кто обладает ключом и знает алгоритм, может расшифровать сообщение.

Асимметричное шифрование (шифрование с открытым ключом)

Асимметричное шифрование — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом.

Электронные подписи(MAC, HMAC)

Электронная подпись — это асимметричное шифрование наоборот: вы зашифровываете закрытым ключом, а расшифровать может кто угодно с помощью открытого ключа, который доступен всем.

Хэширование(MD5, SHA1, SHA256)

Криптографические хэши используются везде, от хранения паролей до систем проверки файлов. Основная идея состоит в том, чтобы использовать детерминированный алгоритм (алгоритмический процесс, который выдает уникальный и предопределенный результат для задачи входных данных), который принимает один вход и создает строку фиксированной длины каждый раз.

**Исторические шифры:**

Шифр Цезаря

Классический шифр Цезаря предполагает смещение каждой буквы текста на следующую за ней через три. Последние буквы смещаются в начало алфавита по кольцу.

Шифр Вижинера

Фатальный недостаток шифра Цезаря - то, что каждый символ текста преобразуется в один и тот же символ шифротекста. Это можно исправить используя ключ длиной не в одно число, а в несколько. Тогда первый символ смещается на первое число в ключе, второй - на второе и так далее. Дополнительно, такой ключ тоже можно воспринимать как строку - выполнив преобразование из массива чисел в массив символов.

Такой шифр имеет абсолютную криптографическую стойкость, если ключ равен по длине тексту и используется только один раз. На практике это очень неудобно и приходится использовать ключ гораздо короче, чем сам текст. В таком случае ключ просто “размножается” до нужной длины.

Шифр Энигмы

Энигма — это машина, которая использовалась нацистами во времена Второй Мировой для шифрования сообщений.

Есть несколько колёс и клавиатура. На экране оператору показывалась буква, которой шифровалась соответствующая буква на клавиатуре. То, какой будет зашифрованная буква, зависело от начальной конфигурации колес.

Существовало более ста триллионов возможных комбинаций колёс, и со временем набора текста колеса сдвигались сами, так что шифр менялся на протяжении всего сообщения.

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

Симметричное шифрование - это метод шифрования, при котором для защиты информации используется ключ, зная который любой может расшифровать или зашифровать данные.

Симметричное шифрование остаётся самым актуальным и криптографически гарантированными методом защиты информации. В симметричном шифровании, основанном на использовании составных ключей, идея состоит в том, что секретный ключ делится на две части, хранящиеся отдельно. Каждая часть сама по себе не позволяет выполнить дешифрование.

Основным недостатком симметричного шифрования является то, что секретный ключ должен быть известен и отправителю, и получателю. Это создает проблему распространения ключей. Получатель на основании наличия зашифрованного и расшифрованного сообщения не может доказать, что он получил это сообщение от конкретного отправителя, поскольку такое же сообщение он мог сгенерировать самостоятельно.

Единственным недешифруемым шифром является, так называемая, лента одноразового использования (One-time Pad), когда открытый текст шифруется с помощью случайного ключа такой же длины. Это обстоятельство делает абсолютно стойкий шифр очень дорогим в эксплуатации.

Математическое описание шифра замены выглядит следующим образом. Пусть Х и Y - два текста (открытый и шифрованный соответственно), Х взаимно однозначно отображается в текст Y. Действие шифра замены можно представить как преобразование открытого текста X = (x1,x2,…xn) в шифрованныйтекст Y = g(X)=g(x1,x2,…,xn).

Математическое описание шифра перестановки выглядит следующим образом. Пусть длина отрезков, на которые разбивается открытый текст, равна m, а S - взаимно однозначное отображение X = (x1,x2,…xm) в себя. Шифр перестановки преобразует отрезок открытого текста x1,x2,…xm в отрезок шифрованного текста S(x1,x2,…,xm).

Общая схема шифрования выглядит следующим образом:

Классическими примерами таких алгоритмов являются симметричные криптографические алгоритмы, перечисленные ниже:

· Простая перестановка

· Одиночная перестановка по ключу

· Двойная перестановка

· Перестановка «Магический квадрат»

**Простая перестановка**

Простая перестановка без ключа — один из самых простых методов шифрования. Сообщение записывается в таблицу по столбцам. После того, как открытый текст записан колонками, для образования шифртекста он считывается по строкам. Для использования этого шифра отправителю и получателю нужно договориться об общем ключе в виде размера таблицы. Объединение букв в группы не входит в ключ шифра и используется лишь для удобства записи несмыслового текста.

**Одиночная перестановка по ключу**

Более практический метод шифрования, называемый одиночной перестановкой по ключу, очень похож на предыдущий. Он отличается лишь тем, что колонки таблицы переставляются по ключевому слову, фразе или набору чисел длиной в строку таблицы.

**Двойная перестановка**

Для дополнительной скрытности можно повторно шифровать сообщение, которое уже было зашифровано. Этот способ известен под названием двойная перестановка. Для этого размер второй таблицы подбирают так, чтобы длины её строк и столбцов отличались от длин в первой таблице. Лучше всего, если они будут взаимно простыми. Кроме того, в первой таблице можно переставлять столбцы, а во второй строки. Наконец, можно заполнять таблицу зигзагом, змейкой, по спирали или каким-то другим способом. Такие способы заполнения таблицы если и не усиливают стойкость шифра, то делают процесс дешифрования гораздо более занимательным.

**Перестановка «Магический квадрат»**

Магическими квадратами называются квадратные таблицы со вписанными в их клетки последовательными натуральными числами от 1, которые дают в сумме по каждому столбцу, каждой строке и каждой диагонали одно и то же число. Подобные квадраты широко применялись для вписывания шифруемого текста по приведенной в них нумерации. Если потом выписать содержимое таблицы по строкам, то получалась шифровка перестановкой букв. На первый взгляд кажется, будто магических квадратов очень мало. Тем не менее, их число очень быстро возрастает с увеличением размера квадрата. Так, существует лишь один магический квадрат размером 3 х 3, если не принимать во внимание его повороты. Магических квадратов 4 х 4 насчитывается уже 880, а число магических квадратов размером 5 х 5 около 250000. Поэтому магические квадраты больших размеров могли быть хорошей основой для надежной системы шифрования того времени, потому что ручной перебор всех вариантов ключа для этого шифра был немыслим.

В квадрат размером 4 на 4 вписывались числа от 1 до 16. Его магия состояла в том, что сумма чисел по строкам, столбцам и полным диагоналям равнялась одному и тому же числу — 34. Впервые эти квадраты появились в Китае, где им и была приписана некоторая «магическая сила».

16 3 2 13

5 10 11 8

9 6 7 12

4 15 14 1

Шифрование по магическому квадрату производилось следующим образом. Например, требуется зашифровать фразу: «ПриезжаюСегодня.». Буквы этой фразы вписываются последовательно в квадрат согласно записанным в них числам: позиция буквы в предложении соответствует порядковому числу. В пустые клетки ставится точка.

16. 3 и 2 р 13 д

5 з 10 е 11 г 8 ю

9 С 6 ж 7 а 12 о

4 е 15 я 14 н 1 П

После этого шифрованный текст записывается в строку (считывание производится слева направо, построчно):

.ирдзегюСжаоеянП

При расшифровывании текст вписывается в квадрат, и открытый текст читается в последовательности чисел «магического квадрата». Программа должна генерировать «магические квадраты» и по ключу выбирать необходимый. Размер квадрата больше чем 3х3.

В настоящее время **симметричные шифры — это**:

* блочные шифры. Обрабатывают информацию блоками определённой длины (обычно 64, 128 бит), применяя к блоку ключ в установленном порядке, как правило, несколькими циклами перемешивания и подстановки, называемыми раундами. Результатом повторения раундов является лавинный эффект — нарастающая потеря соответствия битов между блоками открытых и зашифрованных данных.
* поточные шифры, в которых шифрование проводится над каждым битом либо байтом исходного (открытого) текста с использованием гаммирования. Поточный шифр может быть легко создан на основе блочного (например, ГОСТ 28147-89 в режиме гаммирования), запущенного в специальном режиме

**Виды симметричных шифров:**

блочные шифры

* AES (англ. Advanced Encryption Standard) — американский стандарт шифрования
* ГОСТ 28147-89 — советский и российский стандарт шифрования, также является стандартом СНГ
* DES (англ. Data Encryption Standard) — стандарт шифрования данных в США
* 3DES (Triple-DES, тройной DES)
* RC2 (Шифр Ривеста (Rivest Cipher или Ron’s Cipher))
* RC5
* Blowfish
* Twofish
* NUSH
* IDEA (International Data Encryption Algorithm, международный алгоритм шифрования данных)
* CAST (по инициалам разработчиков Carlisle Adams и Stafford Tavares)
* CRAB
* 3-WAY
* Khufu и Khafre
* Kuznechik

потоковые шифры

* RC4 (алгоритм шифрования с ключом переменной длины)
* SEAL (Software Efficient Algorithm, программно-эффективный алгоритм)
* WAKE (World Auto Key Encryption algorithm, алгоритм шифрования на автоматическом ключе)

**Сравнение с асимметричными криптосистемами:**

Достоинства:

* скорость
* простота реализации (за счёт более простых операций)
* меньшая требуемая длина ключа для сопоставимой стойкости
* изученность (за счёт большего возраста)

Недостатки:

* сложность управления ключами в большой сети
* сложность обмена ключами. Для применения необходимо решить проблему надёжной передачи ключей каждому абоненту, так как нужен секретный канал для передачи каждого ключа обеим сторонам

Для компенсации недостатков симметричного шифрования в настоящее время широко применяется комбинированная (гибридная) криптографическая схема, где с помощью асимметричного шифрования передаётся сеансовый ключ, используемый сторонами для обмена данными с помощью симметричного шифрования.

Важным недостатком симметричных шифров является невозможность их использования в механизмах формирования электронной цифровой подписи и сертификатов, так как ключ известен каждой стороне.

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

Асимметричное шифрование — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом. Открытый и закрытый ключи — это очень большие числа, связанные друг с другом определенной функцией, но так, что, зная одно, крайне сложно вычислить второе.

Асимметричное шифрование используется для защиты информации при ее передаче, также на его принципах построена работа электронных подписей.

**Принцип действия 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.

**Наиболее распространенные алгоритмы асимметричного шифрования:**

* RSA (аббревиатура от Rivest, Shamir и Adelman, фамилий создателей алгоритма) — алгоритм, в основе которого лежит вычислительная сложность факторизации (разложения на множители) больших чисел. Применяется в защищенных протоколах SSL и TLS, стандартах шифрования, например в PGP и S/MIME, и так далее. Используется и для шифрования данных, и для создания цифровых подписей.
* DSA (Digital Signature Algorithm, «алгоритм цифровой подписи») — алгоритм, основанный на сложности вычисления дискретных логарифмов. Используется для генерации цифровых подписей. Является частью стандарта DSS (Digital Signature Standard, «стандарт цифровой подписи»).
* Схема Эль-Гамаля — алгоритм, основанный на сложности вычисления дискретных логарифмов. Лежит в основе DSA и устаревшего российского стандарта ГОСТ 34.10–94. Применяется как для шифрования, так и для создания цифровых подписей.
* ECDSA (Elliptic Curve Digital Signature Algorithm) — алгоритм, основанный на сложности вычисления дискретного логарифма в группе точек эллиптической кривой. Применяется для генерации цифровых подписей, в частности для подтверждения транзакций в криптовалюте Ripple.

**Общая схема:**

Плюсы и минусы асимметричного шифрования:

Плюс таких алгоритмов в том, что для передачи зашифрованных сообщений можно использовать открытый канал связи. Даже если злоумышленник перехватит сообщение, он не сможет прочитать его без секретного ключа. Но чтобы всё было именно так, нужно, чтобы ключ был достаточно длинный — 1024 бит и выше.

Минус асимметричного шифрования очевиден — оно работает только в одну сторону. Чтобы такое общение было двусторонним, каждый должен предоставить другому свой открытый ключ.

## 

## 39. Алгоритмы хэширования. Примеры, назначение.

Хеширование (или хэширование, англ. hashing ) – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хеш-функциями или функциями свертки, а их результаты называют хешем, хеш-кодом, хеш-таблицей или дайджестом сообщения (англ. message digest ).

**SHA2**

Изначально алгоритм SHA (алгоритм безопасного хеширования или алгоритм безопасного хеширования) был создан NSA и NIST с целью генерации хэшей или уникальных кодов на основе стандарта. В 1993 году родился первый протокол SHA, также названный SHA-0, но он почти не использовался и не оказал большого влияния. Пару лет спустя был выпущен более надежный и безопасный улучшенный вариант SHA-1, который в течение многих лет использовался для подписания цифровых сертификатов SSL / TLS для миллионов веб-сайтов. Спустя несколько лет был создан SHA-2, который имеет четыре варианта в зависимости от количества выходных битов: SHA2-224, SHA2-256, SHA2-384 и SHA2-512 . В настоящее время из соображений безопасности SHA1 больше не используется, но настоятельно рекомендуется использовать SHA2 или SHA3 (в семействе SHA).

**Как работает SHA2**

Алгоритмы хеширования работают только в одном направлении: мы можем сгенерировать хэш любого контента или отпечаток пальца, но с помощью хеша или отпечатка пальца нет возможности сгенерировать исходный контент. Единственный способ сделать это - использовать словарь или грубую силу, поэтому получение исходной информации может занять тысячи лет (в настоящее время).

Среди множества различных способов создания хэшей алгоритм SHA2-256 является одним из наиболее часто используемых благодаря своему балансу между безопасностью и скоростью, это очень эффективный алгоритм и имеет высокую устойчивость к коллизиям, что очень важно для поддержания безопасности. . этого алгоритма хеширования. Чтобы алгоритм хеширования был безопасным, не должно быть известно о коллизиях. Например, метод проверки биткойнов основан на SHA2-256.

**Характеристики различных типов SHA2**

* Выходной размер : это размер символов, которые образуют хеш.
* Размер внутреннего состояния : это внутренняя хеш-сумма после каждого сжатия блока данных.
* Размер блока : размер блока, обрабатываемого алгоритмом.
* Максимальное сообщение size: это максимальный размер сообщения, к которому мы применяем алгоритм.
* Word длина: это длина в битах операции, применяемой алгоритмом в каждом раунде.
* Взаимодействия или раунды : это количество операций, которые алгоритм выполняет для получения окончательного хеша.
* Поддерживаемые операции : это операции, выполняемые алгоритмом для получения окончательного хеша.

**SHA-256**

Он имеет 256-битный выходной размер, 256-битный размер внутреннего состояния, 512-битный размер блока, максимальный размер сообщения, который он может обработать, составляет 2 64 - 1, длина слова составляет 32 бита, а количество примененных раундов - 64, а также операции, применяемые к хешу: +, и, или, xor, shr и rot. Длина хэша всегда одинакова, независимо от того, насколько велик контент, который вы используете для генерации хэша: будь то всего одна буква или образ ISO 4 ГБ, результатом всегда будет последовательность из 40 букв и цифр.

**SHA2-384**

Этот алгоритм отличается по характеристикам, но принцип его действия такой же. Он имеет размер вывода 384 бит, размер внутреннего состояния 512 бит, размер блока 1024 бит, максимальный размер сообщения, который он может обработать, составляет 2 128 - 1, длина слова - 64 бита, количество примененных раундов - 80, а также операции, применяемые к хешу: +, и, или, xor, shr и rot. Этот алгоритм является более безопасной версией, чем SHA2-256, поскольку применяется больше раундов операций, и он также может применяться к более обширной информации. Этот алгоритм хеширования обычно используется для проверки целостности и подлинности сообщений в виртуальных частных сетях. Отрицательным моментом является то, что он несколько медленнее, чем SHA2-256, но в определенных обстоятельствах это может быть очень хорошим вариантом для его использования.

**SHA2-512**

Как и во всех SHA2, операция одинакова, они меняют только одну характеристику. Его выходной размер составляет 512 бит. Все остальные функции такие же, как у SHA2-384. 512 бит размера внутреннего состояния, 1024 бит размера блока, 2 128 - 1 для максимального размера сообщения, 64 бита длины слова и 80 - количества раундов, примененных к нему. Этот алгоритм также применяет одни и те же операции к каждому раунду +, и, или, xor, shr и rot.

**SHA2-224**

Мы не комментировали этот алгоритм как основной, потому что его старший брат (SHA2-256) используется гораздо чаще, поскольку вычислительная разница между ними смехотворна, а SHA2-256 гораздо более стандартизирован. Мы упоминаем об этом, потому что, по крайней мере, до сих пор для этого алгоритма не было обнаружено коллизий, что делает его безопасным и удобным вариантом.

**SHA-3**

SHA3 ​​- это новейший алгоритм хеширования семейства SHA, он был опубликован NISH в 2015 году, но пока не получил широкого распространения. Хотя он принадлежит к одному семейству, его внутренняя структура совершенно другая. Этот новый алгоритм хеширования основан на «Конструкция из губки . » Конструкция этой губки основана на случайной функции или случайной перестановке данных, она позволяет вводить любой объем данных и генерировать любой объем данных, кроме того, функция является псевдослучайной по отношению ко всем предыдущим записям. Это позволяет SHA-3 иметь большую гибкость, цель состоит в том, чтобы заменить SHA2 в типичном TLS или [VPN](https://itigic.com/ru/tag/vpn/) протоколы, которые используют этот алгоритм хеширования для проверки целостности данных и их подлинности.

Хэш-функции имеют разнообразные применения при проведении статистических экспериментов, при тестировании логических устройств, при построении алгоритмов быстрого поиска и проверки целостности записей в базах данных. Основным требованием к хэш-функциям является равномерность распределения их значений при случайном выборе значений аргумента.

**Назначение**: проверка целостности файлов (отпечаток файла), хеширование паролей на сайтах в БД

## 40. Протокол TLS/SSL. Общая схема взаимодействия, назначение.

Протоколы SSL и TLS обеспечивают аутентификацию сервера и клиента и шифрование соединений. SSL был впервые введен компанией Netscape Communication в 1994 году и дважды пересматривался (последней версией SSL является версия 3). В 1996 году IETF основало рабочую группу TLS, чтобы определить протокол SSL в качестве стандарта Интернета.

**Протокол TLS преследует три цели:**

* обеспечение безопасности соединения
* идентификация сторон
* обеспечение надежной передачи данных.

Для работы TLS/SSL использует комбинацию открытого сертификата и закрытого ключа. Закрытый ключ хранится на сервере и не разглашается. SSL-сертификат используется открыто и доступен всем пользователям, запрашивающим контент.

**Схема работы SSL/TLS:**

Протокол реализуется на базе стека TCP/IP. Устанавливает алгоритмы шифрования и ключи на обоих сторонах и прокидывает шифрованный туннель, по которому могут передаваться другие протоколы (например HTTP)

Принцип работы SSL состоит из двух фаз: фаза рукопожатия и фаза передачи данных. Во время фазы рукопожатия клиент и сервер используют шифрование открытым ключом для того, чтобы определить параметры секретного ключа, используемого клиентом и сервером для шифрования во время фазы передачи данных.

Клиент инициирует рукопожатие посылая “hello”-сообщение серверу. Такое сообщение содержит список алгоритмов симметричного шифрования (cipher specs), поддерживаемых клиентом. Сервер отвечает похожим “hello”-сообщением, выбрав при этом наиболее подходящий алгоритм шифрования из полученного списка. Далее сервер отправляет сертификат, который содержит его публичный ключ.

Сертификат - это набор данных, который подтверждает подлинность. Подтвержденная третья сторона, известная как центр сертификации (CA), генерирует сертификат и проверяет его подлинность. Чтобы получить сертификат сервер должен использовать безопасные каналы для отправки своего публичного ключа в центр сертификации. Он генерирует сертификат, который содержит его собственный ID, ID сервера, публичный ключ сервера и другую информацию. А также центр сертификации создает отпечаток (digest) сертификата, который, по сути, является контрольной суммой. Далее центр сертификации создает подпись сертификата (certificate signature), которая формируется путем шифрования отпечатка сертификата приватным ключом центра сертификации.

Для проверки сертификата сервера клиент использует публичный ключ центра сертификации для расшифровки подписи. Затем клиент самостоятельно считает отпечаток сертификата сервера и сверяет с расшифрованным. Если они не совпадают, то сертификат был подделан. Естественно, для расшифровки подписи у клиента должен быть публичный ключ центра авторизации. Поэтому клиент хранит у себя список публичных ключей подтвержденных центров сертификации. По факту, многие браузерные приложения имеют подобный список, находящийся непосредственно в их коде. Когда клиент установил подлинность сервера (сервер также может запросить сертификат у клиента), сервер использует шифрование открытым ключом для определения секретного ключа для обмена информацией.

Фаза рукопожатия завершается отправкой “finished”-сообщений, как только обе стороны готовы начать использование секретного ключа. Начинается фаза передачи данных, в ходе которой каждая сторона разбивает исходящие сообщения на фрагменты и прикрепляет к ним коды авторизации сообщений MAC (message authentication code). Код авторизации сообщения это зашифрованный отпечаток, вычисленный на основе содержимого сообщений. Из соображений безопасности, он не совпадает с секретным ключом и вычисляется вместе с секретным ключом на стадии рукопожатия. Для получения полноценного SSL пакета каждая из сторон объединяет данные фрагмента, код авторизации сообщения, заголовки сообщения и шифруют с использованием секретного. При получении пакета каждая из сторон расшифровывает его и сверяет полученный код авторизации сообщения со своим. Если они не совпадают, то пакет был подделан.

Протокол TLS делится на два слоя: TLS Record и TLS Handshake.

**Подтверждение связи (handshake)**

1. Клиент посылает сообщение ClientHello, указывающее версию SSL или TLS и поддерживаемые клиентом методы шифрования (англ. *CipherSuite*). Это сообщение также содержит случайное число (набор байт), которое используется в последующих вычислениях. Протокол также позволяет указать поддерживаемые клиентом методы сжатия данных.
2. Сервер отвечает сообщением ServerHello, которое содержит метод шифрования, выбранный сервером из списка, предложенного клиентом, а также идентификатор сессии и еще одно случайное число. Также сервер посылает свой цифровой сертификат. Если серверу нужен сертификат для аутентификации клиента, на этом шаге он может послать клиенту запрос такого сертификата.
3. Клиент проверяет сертификат сервера.
4. Клиент отправляет случайное число, которое клиент и сервер используют для шифрования последующих сообщений. Сама строка из байт шифруется публичным ключом сервера.
5. Если сервер потребовал у клиента сертификат, клиент отсылает набор байт, зашифрованный его секретным ключом, и свой цифровой сертификат, или оповещение об отсутствии сертификата.
6. Сервер проверяет сертификат клиента.
7. Клиент и сервер отправляют друг другу сообщение ChangeCipherSpec, объявляя об изменении режима передачи данных с незащищенного на защищенный.
8. Клиент отправляет сообщение Finished, зашифрованное секретным ключом, и таким образом завершает подтверждение связи со своей стороны.
9. Аналогичные действия производит сервер.
10. На протяжении данной сессии клиент и сервер могут обмениваться сообщениями, зашифрованными секретным ключом.

**Возобновление сессии**

1. Клиент посылает сообщение ClientHello, используя ID сессии, которую нужно возобновить.
2. Сервер проверяет, есть ли у него в кэше соответствующий идентификатор. Если есть и сервер способен возобновить сессию, он отсылает клиенту сообщение ServerHello с этим же ID сессии. Если нет, сервер генерирует новый ID сессии и выполняет процедуру handshake с клиентом.
3. Клиент и сервер обмениваются сообщениями ChangeCipherSpec, а затем Finished.
4. Передача данных по защищенному каналу возобновляется.

**Протокол записи (TLS Record)**

Этот слой защищает данные с помощью ключей, полученных при подтверждении связи, и проверяет целостность и источник входящих сообщений. Он выполняет следующие функции:

* Разбиение исходящих сообщений на блоки нужного размера и "склеивание" входящих сообщений.
* Сжатие исходящих сообщений и распаковку входящих (используется не всегда).
* Применение кода аутентификации к исходящим сообщениям и проверку входящих с помощью MAC.
* Шифрование исходящих сообщений и дешифровку входящих.

После обработки протоколом TLS Record зашифрованные данные передаются на слой TCP для передачи.

**Состав записи**

* *Content type*: тип сообщения — подтверждение связи (22), обычное сообщение (23) или оповещение (21).
* *Version*: версия SSL/TLS.
* *Length*: длина оставшейся части сообщения.
* *Payload*: собственно зашифрованные данные.
* *MAC*: код аутентификации.
* *Padding*: "отступ" для получения нужного размера сообщения.

**Назначение:**

Сетевые протоколы SSL и TLS являются криптографическими протоколами, обеспечивающими аутентификацию и защиту от несанкционированного доступа, нарушения целостности передаваемых данных. Протоколы SSL/TLS предназначены для исключения подмены идентификатора на клиентской или серверной стороне, раскрытия или искажения данных.

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

**SSL-сертификат** – это цифровой сертификат, удостоверяющий подлинность веб-сайта и позволяющий использовать зашифрованное соединение. Аббревиатура SSL означает Secure Sockets Layer – протокол безопасности, создающий зашифрованное соединение между веб-сервером и веб-браузером.

Компаниям и организациям необходимо добавлять SSL-сертификаты на веб-сайты для защиты онлайн-транзакций и обеспечения конфиденциальности и безопасности клиентских данных.

SSL обеспечивает безопасность интернет-соединений и не позволяет злоумышленникам считывать или изменять информацию, передаваемую между двумя системами. Если в адресной строке рядом с веб-адресом отображается значок замка, значит этот веб-сайт защищен с помощью SSL.

С момента создания протокола SSL около 25 лет назад, он был доступен в нескольких версиях. При использовании каждой из этих версий в определенный момент возникали проблемы безопасности. Затем появилась обновленная переименованная версия протокола – TLS (Transport Layer Security), которая используется до сих пор. Однако аббревиатура SSL прижилась, поэтому новая версия протокола по-прежнему часто называется старым именем.

**Зачем нужен SSL-сертификат:**

SSL-сертификаты сайтов требуются для обеспечения безопасности данных пользователей, подтверждения прав собственности на сайт, предотвращения создание поддельной версии сайта злоумышленниками и обеспечения доверия со стороны пользователей.

Если использование веб-сайта предполагает вход в систему, ввод личных данных, таких как номера кредитных карт, или просмотр конфиденциальной информации, такой как данные медицинской страховки, или финансовой информации, то важно сохранить конфиденциальность этих данных. SSL-сертификаты помогают сохранить конфиденциальность онлайн-транзакций и гарантируют пользователям, что веб-сайт является подлинным и безопасным для ввода личных данных.

Для бизнеса более актуален тот факт, что SSL-сертификаты требуются для веб-адресов HTTPS. HTTPS – это безопасная форма HTTP, то есть трафик веб-сайтов HTTPS зашифрован с помощью SSL. Большинство браузеров помечают сайты HTTP, не имеющие SSL-сертификатов, как небезопасные. Это сигнал пользователям о том, что сайт может быть небезопасным, а для компаний это стимул перейти на HTTPS.

**SSL-сертификат помогает защитить такую информацию, как:**

* Учетные данные для входа в систему.
* Операции по кредитной карте и информацию о банковском счете.
* Личную информацию: полное имя, адрес, дату рождения, номер телефона.
* Юридические документы и контракты.
* Медицинские документы.
* Конфиденциальную информацию.

**Самоподписанные сертификаты:**

Самозаверяющий сертификат SSL - это сертификат, подписанный лицом, которое его создало, а не доверенным центром сертификации. Самозаверяющие сертификаты могут иметь тот же уровень шифрования, что и доверенный CA-подписанный SSL-сертификат.

Самозаверяющие сертификаты, признанные действительными в любом браузере. Если вы используете самозаверяющий сертификат, веб-браузер покажет посетителю предупреждение о том, что сертификат веб-сайта невозможно проверить.

Самозаверяющие сертификаты в основном используются для целей тестирования или внутреннего использования. Вы не должны использовать самозаверяющий сертификат в производственных системах, которые подключены к Интернету.

Чтобы создать новый самоподписанный сертификат SSL, используйте openssl req команду:

openssl req -newkey rsa:4096 \

-x509 \

-sha256 \

-days 3650 \

-nodes \

-out example.crt \

-keyout example.key

**Что такое центры сертификации (CA)?**

Это организация, которая обладает правом выдачи цифровых сертификатов. Она производит проверку данных, содержащихся в CSR, перед выдачей сертификата. В самых простых сертификатах проверяется только соотвествие доменного имени, в самых дорогих производится целый ряд проверок самой организации, которая запрашивает сертификат. Об этом мы поговорим ниже.

Так вот, разница между самоподписными бесплатным и платными сертификатами, выданными центром сертификации как раз и заключается в том, что данные в сертификате проверены центром сертификации и при использовании такого сертификата на сайте ваш посетитель никогда не увидит огромную ошибку на весь экран.

Говоря в общем, SSL сертификаты содержат и отображают (как минимум одно из) ваше доменное имя, ваше название организации, ваш адрес, город и страницу. Также сертификат всегда имеет дату окончания и данные о центре сертификации, ответственного за выпуск сертификата. Браузер подключается к защищенному сайту, получает от него SSL сертификат и делает ряд проверок: он не просрочен ли сертификат, потом он проверяет, выпущен ли сертификат известным ему центром сертификации (CA) используется ли сертификат на сайте, для которого он был выпущен.

Если один из этих параметров не проходит проверку, браузер отображает предупреждение посетителю, чтобы уведомить, что этот сайт не использует безопастное соединение SSL. Он предлагает покинуть сайт или продолжить просмотр, но с большой осторожностью. Это последнее, что вы должны увидеть ваши потенциальные клиенты.

**Центров сертификации существует достаточно много, вот перечень самых популярных:**

Comodo — работает с 1998 штабквартира в Jersey City, New Jersey, США.

Geotrust — основан в 2001, в 2006 продан Verisign, штабквартира Mountain View, California, США

Symantec — бывший Verisign в состав которого входит и Geotrust. Купил всех в 2010 году.

Thawte — основан в 1995, продан Verisign в 1999.

Trustwave — работает с 1995, штабквартира Chicago, Illinois, США.

Как видим самый крупный игрок на рынке SSL сертификатов это Symantec, который владеет тремя крупнейшими центрами сертификации — Thawte, Verisgin и Geotrust.

## 

# Примерные практические задания к экзамену

Решения размещены тут:

<https://github.com/StudOborona/UnixSummer2021>

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

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

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

## 4. 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.

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

## 6. 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.

## 7. 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.

## 8. 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.

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

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