**Что такое asyncio?**

В синхронном приложении код исполняется последовательно. Следующая строка кода выполняется после завершения предыдущей, и в каждый момент времени происходит что-то одно. Эта модель хорошо работает для многих, если не для большинства приложений. Но что, если какая-то одна строка кода занимает слишком много времени? В таком случае весь последующий код должен будет замереть, пока эта строка не соблаговолит завершиться. в каждый момент времени может выполняться только одна длительная операция, а она не дает приложению больше ничего делать.

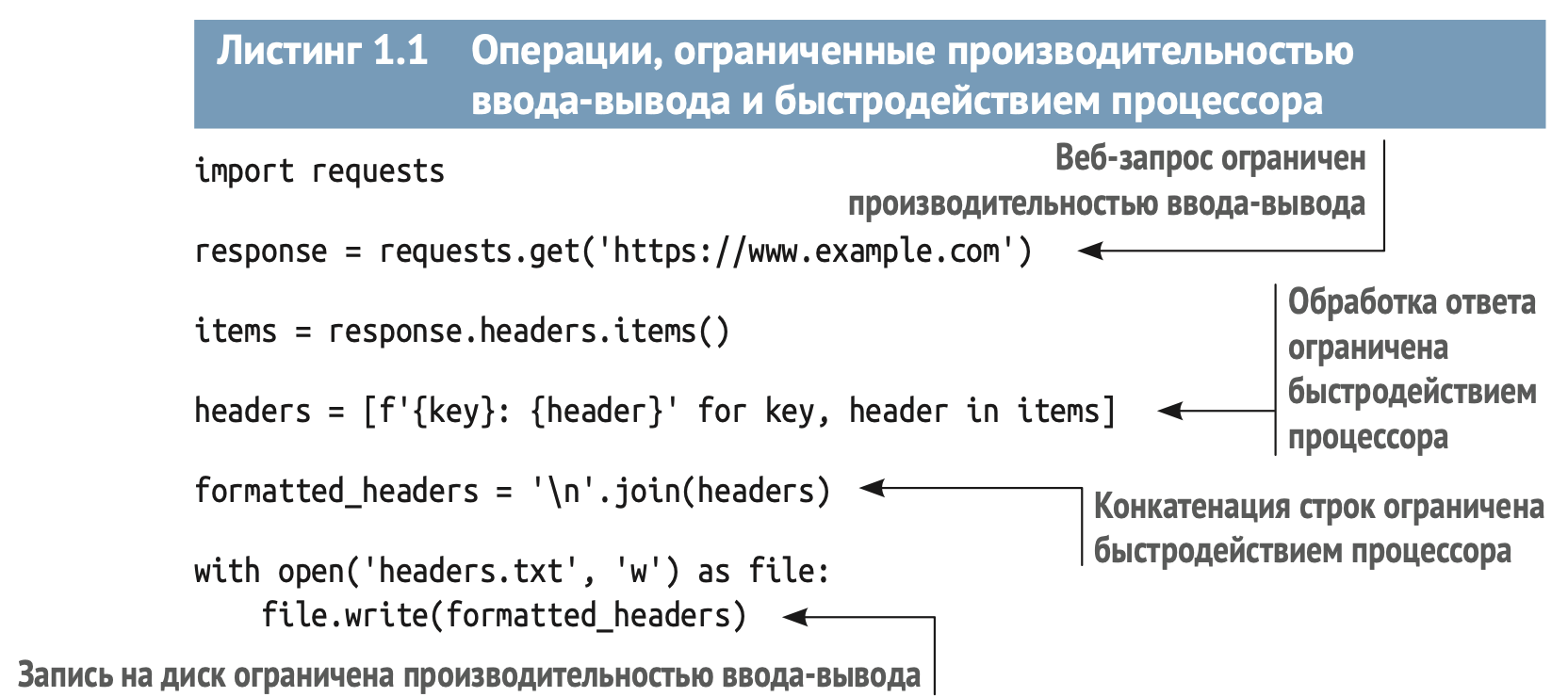
**Что означают слова «асинхронное программирование»**? Что длительную задачу можно выполнять в фоновом режиме отдельно от главного приложения. И система не блокируется в ожидании завершения этой задачи, а может заниматься другими вещами, не зависящими от ее исхода.

**Сопрограмма** - это метод, который можно приостановить, если имеется потенциально длительная задача, а затем возобновить, когда она завершится.

Библиотека asyncio исполняет сопрограммы асинхронно, пользуясь моделью конкурентности, получившей название **однопоточный цикл событий.**

Операция, ограниченная быстродействием процессора (счетная операция), завершилась бы быстрее, будь процессор более мощным.

Операция, ограниченная производительностью ввода-вывода (**IO - операция**), работала бы быстрее, если бы устройство могло обработать больше данных за меньшее время (например увеличить пропускную способность сети или установив более быструю сетевую карту)



**Асинхронный ввод-вывод** позволяет приостановить выполнение метода, встретив операцию ввода-вывода; ожидая завершения этой операции, работающей в фоновом режиме, мы можем выполнять какой-нибудь другой код.

**Конкурентность** - Несколько задач выполняются одновременно, но не параллельно (чередуются), только одна задача активна в каждый момент времени. Пример конкурентного поведения - делать что-то пока выполняется другой процесс (например замешивать тесто пока разогревается духовка, а не сидеть и ждать пока духовка разогреется )

**Параллелизм** - Задачи выполняются строго в одно и то же время, несколько задач активны одновременно. Пример - позвать второго пекаря. Два пекаря замешивают два разных теста, делают одновременно два разных дела.

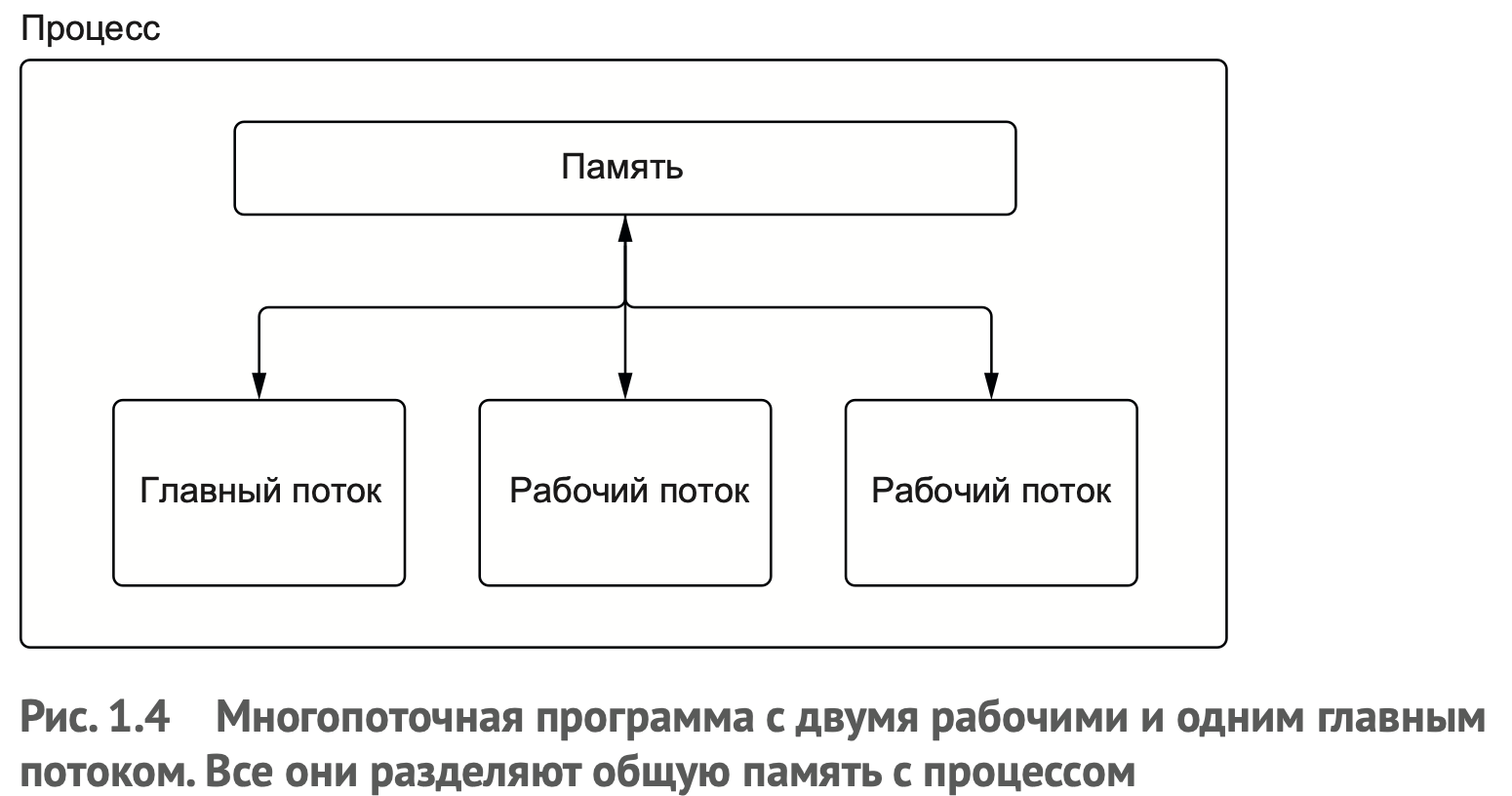
**Вытесняющая многозадачност**ь - операционная система решает как переключаться между выполняемыми задачами с помощью квантования времени.

**Кооперативная (невытесняющая) многозадачность** - явно в коде определяются точки где можно уступить выполнение другой задаче.

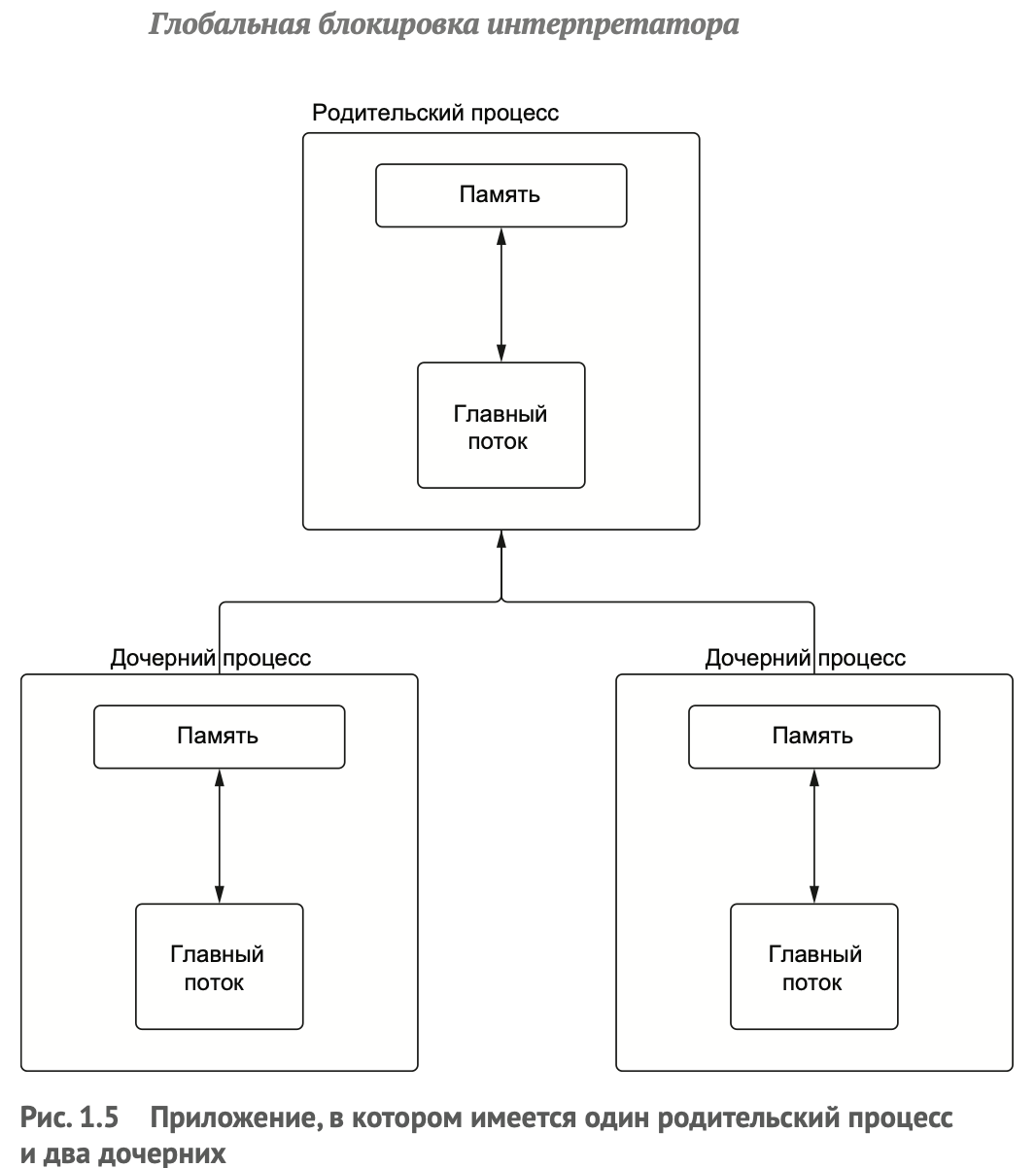
**Процесс** - работающее приложение которому выделена область памяти, недоступная другим приложениям. На одной машине могут работать несколько процессов. Если машина оснащена процессором с несколькими ядрами, то несколько процессов могут работать одновременно. Если процессор имеет только одно ядро, то все равно можно выполнять несколько приложений конкурентно, но уже с применением квантования времени. При использовании квантования операционная система будет автоматически вытеснять работающий процесс по истечении некоторого промежутка времени и передавать процессор другому процессу. Алгоритмы, определяющие, в какой момент переключать процессы, зависят от операционной системы.

**Поток** - У потоков нет своей памяти, они пользуются памятью создавшего их процесса. На один процесс минимум один поток, называемый главным. Дополнительные потоки созданные процессом называются рабочими или фоновыми потоками (выполняют работу конкурентно).

**Многопоточность** - потоки созданные процессом и разделяющие его память, конкурентно выполняют другую работу.



**Многопроцессность** - несколько конкурентных процессов. Родительский процесс создает дочерние процессы и распределяет между ними работу. Многопроцессность обычно предпочтительна, когда имеется счетная задача.



**Глобальная блокировка интерпретатора** (global interpreter lock - **GIL**) - GIL не дает Python-процессу исполнять более одной команды байт-кода в каждый момент времени. Это означает, что, даже если имеется несколько потоков на многоядерной машине, интерпретатор сможет в каждый момент исполнять только один поток.

GIL освобождается при вводе-выводе, но не освобождается для счетных задач. Все дело в системных вызовах, которые выполняются за кулисами. В случае ввода-вывода низкоуровневые системные вызовы работают за пределами среды выполнения Python. GIL захватывается снова, только когда полученные данные переносятся в объект Python.

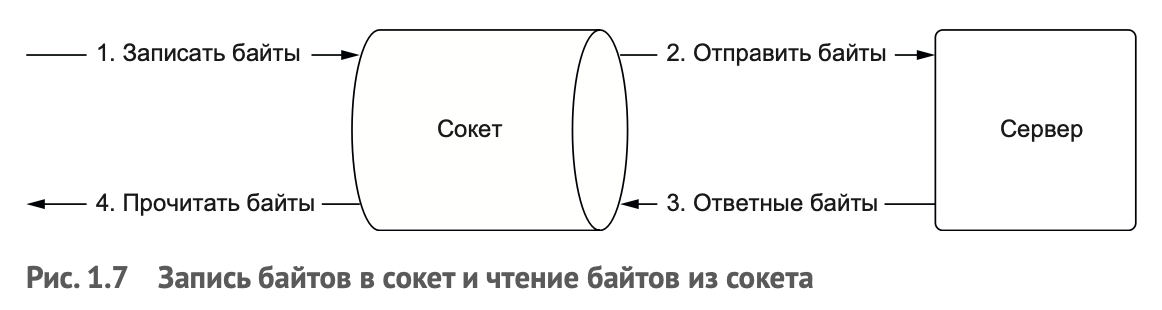
В Python лучшее, на что можно рассчитывать, - конкурентность операций ввода-вывода, поскольку в любой момент может выполняться только один кусок написанного на Python кода.

**Состояние гонки** - когда два потока одновременно обращаются к одному объекту python и увеличивают счетчик ссылок, может случится что счетчик обнулится.

**Asyncio** - конкурентность в одном потоке. При работе с Asyncio создаются сопрограммы. Сопрограммы - это облегченные потоки. Может быть несколько сопрограмм работающих бок о бок. asyncio не обходит GIL. Если имеется счетная задача, то для ее конкурентного выполнения все равно нужно заводить отдельный процесс.

**Сокет** - это низкоуровневая абстракция отправки и получения данных по сети. С ее помощью производится обмен данными между клиентами и серверами.

Низкоуровневую концепцию сокетов проще понять, если представлять их как почтовые ящики. Вы можете положить письмо в почтовый ящик, почтальон заберет его и доставит в почтовый ящик получателя. Получатель откроет его и достанет ваше письмо. В зависимости от содержания письма получатель может отправить в ответ. Здесь письмо является аналогом данных или байтов, которые мы хотим отправить. Можно рассматривать помещение письма в почтовый ящик как запись байтов в сокет, а извлечение его из ящика - как чтение байтов из сокета. Почтальон тогда является аналогом механизма передачи через интернет, который маршрутизирует данные до адреса назначения.



По умолчанию **сокеты блокирующие** - значит приложение не может ничего делать, пока не придут данные от сервера или произойдет ошибка или случится тайм-аут.

Сокеты могут работать в **неблокирующем режиме**, когда мы просто начинаем операцию чтения или записи и забываем о ней, а сами занимаемся другими делами. Но позже операционная система уведомляет нас о том, что байты получены, и в этот момент мы можем уделить им внимание. Это позволяет приложению не ждать прихода байтов, а делать что-то полезное. Для реализации такой схемы применяются различные системы уведомления с помощью событий, разные в разных ОС.

Библиотека asyncio абстрагирует различия между системами уведомления, а именно:

- kqueue - FreeBSD MacOS;

﻿﻿ - epoll - Linux;

- ﻿﻿IOCP (порт завершения ввода-вывода) - Windows.

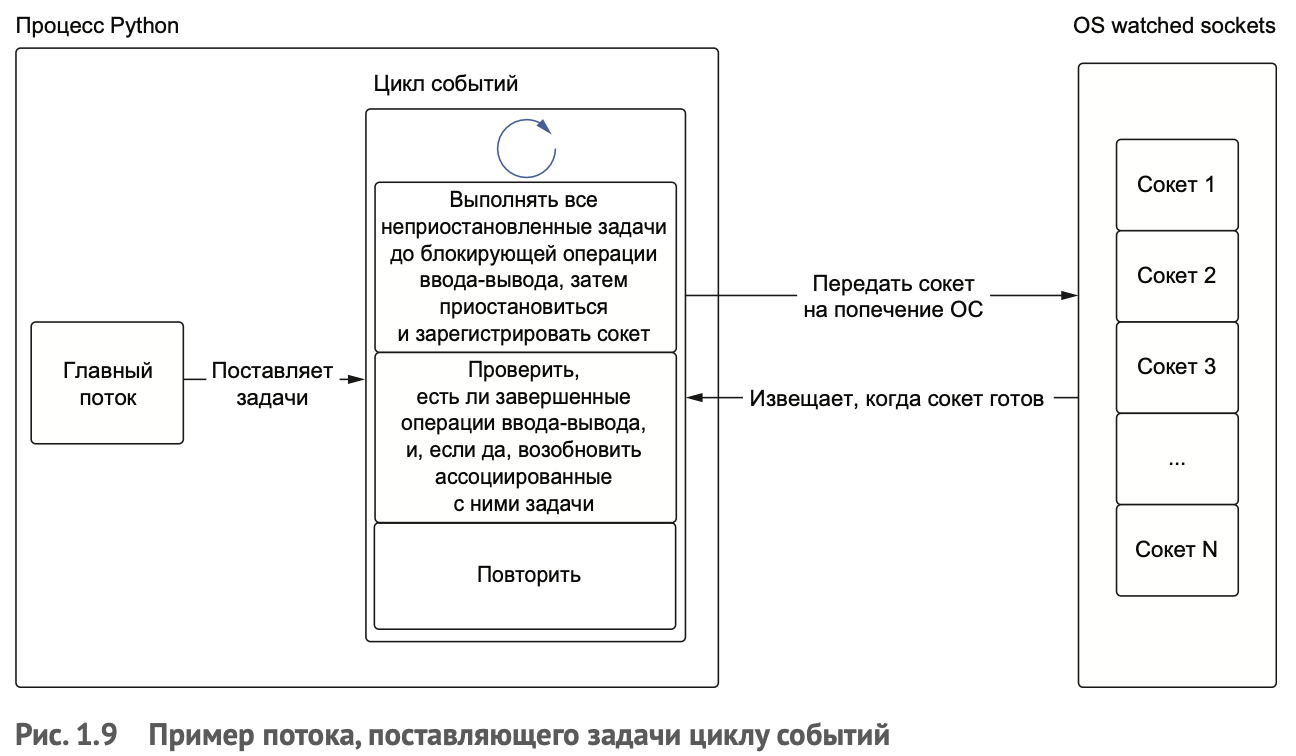
Эти системы наблюдают за неблокирующими сокетами и уведомляют нас, когда с сокетом можно начинать работу.

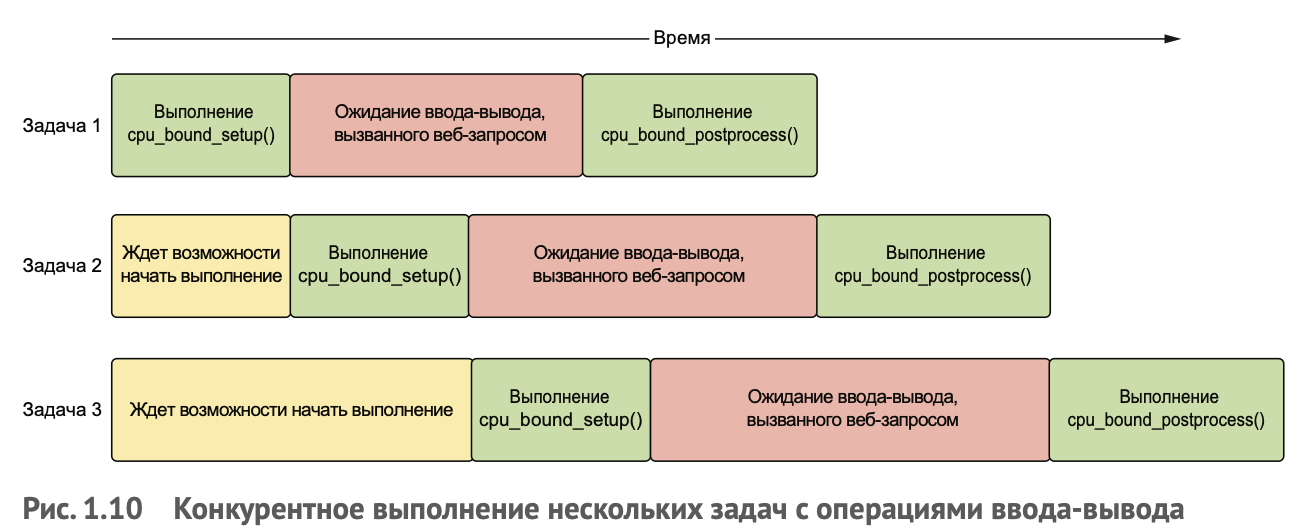
Именно они лежат в основе модели конкурентности в asyncio. В этой модели имеется только один поток, исполняющий

Python-код. Встретив операцию ввода-вывода, интерпретатор передает ее на попечение системы уведомления, входящей в состав ОС. Совершив этот акт, поток Python волен исполнять другой код или открыть другие неблокирующие сокеты, о которых позаботится ОС. По завершении операции система «пробуждает» задачу, ожидающую результата, после чего выполняется код, следующий за этой операцией.

Как отличить задачи ожидающие завершения ввода - вывода от тех, которые просто выполняют python код и ничего не ждут? Решение - конструкция **event - loop (цикл событий)**

B asyncio цикл событий управляет очередью задач. Задача - это обертка вокруг сопрограммы. Сопрограмма может приостановить выполнение, встретив операцию ввода-вывода, и дать циклу событий возможность выполнить другие задачи, которые не ждут завершения ввода-вывода. Создавая цикл событий, мы создаем пустую очередь задач. Затем добавляем в эту очередь задачи для выполнения. На каждой итерации цикла проверяется, есть ли в очереди готовая задача, и если да, то она выполняется, пока не встретит операцию ввода-вывода. В этот момент задача приостанавливается, и мы просим операционную систему наблюдать за ее сокетами. А сами тем временем переходим к следующей готовой задаче. На каждой итерации проверяется, завершилась ли какая-нибудь операция ввода-вывода; если да, то ожидавшие ее завершения задачи пробуждаются и им предоставляется возможность продолжить работу.





Каждый момент времени работает только один счетный кусок кода, но при этом конкурентно выполняются одна или две операции ввода-вывода. Именно из-за такого перекрытия ожидания ввода-вывода asyncio и достигает экономии во времени.

**Сопрограммы**. ! Сопрограммы не выполняются, если их вызвать напрямую. Вместо этого возвращается объект сопрограммы, который будет выполнен позже. Чтобы выполнить сопрограмму, мы должны явно передать ее циклу событий!

**asyncio.run** - для запуска сопрограммы. (создает новое событие - выполняет код сопрограммы -> возвращает результат

-> подчищает все то что могло остаться после завершения сопрограммы -› останавливает и завершает цикл событий).

Она задумана как главная точка входа - выполняет только одну сопрограмму, которая может содержать другие сопрограммы.

Ключевое слово **async** говорит, что это сопрограмма, а не обычная функция.

Встретив выражение **await**, интерпретатор приостанавливает родительскую сопрограмму и выполняет сопрограмму в выражении await.

**asyncio.sleep** - моделирование длительных операций. asyncio.sleep сама является сопрограммой, поэтому вызывать ее следует с помощью await.

**asyncio.create\_task** - создание задач, ей передается подлежащая выполнению сопрограмма, в ответ она немедленно возвращает объект задачи.

Доходим до create\_task -> создаем задачи. После первой встречи с await начинают выполняться ВСЕ ожидающие задачи.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

import asyncio  
from util import delay  
  
async def main():  
 delay\_task = asyncio.create\_task(delay(2))  
 try:  
 result = await asyncio.wait\_for(delay\_task, timeout=1)  
 print(result)  
 except asyncio.exceptions.TimeoutError:  
 print('Got a timeout!')  
 print(f'Was the task cancelled? {delay\_task.cancelled()}')  
  
asyncio.run(main())

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Снятие задачи**. Чтобы остановить задачу нужно вызвать метод cancel у объекта задачи. В результате снятия задачи сработает исключение CancelledError, когда мы ждем ее с помощью await. Это исключение можно обрабатывать.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(Код)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

long\_task.done()- возвращает True, если задача выполнена успешно

long\_task.cancel() - снимаем задачу по условию -› Далее возникает исключение Cancelled Error.

Вызов cancel не прерывает задачу, делающую дело; он снимает ее, только если она уже находится в точке ожидания или когда дойдет до следующей такой точки. Вызов исключения CancelledError возможен только внутри await.

Задать тайм-аут и снять задачу с помощью **asyncio.wait\_for**. Она принимает объект сопрограммы или задачи и таймаут в сек. и возвращает сопрограмму к которой можно применить await. (result = await asyncio.wait\_for(delay\_task, timeout=1)) Если задача не завершилась в отведенное время, возникает исключение TimeoutError и задача автоматически снимается.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(Код)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Чтобы задача не была снята, а пользователь получил сообщение что задача все еще работает, можно обернуть задачу функцией **shield().**В блоке обработки исключения мы печатаем полезное сообщение пользователю, в котором говорим, что задача еще работает, после чего ждем ее завершения с помощью await.

Объект **future** в Python содержит одно значение, которое мы ожидаем получить в будущем, но пока еще, возможно, не получили. В момент создания обычно объект future - неполный или неготовый, т.е. не обертывает никакого значения потому что его еще не существует. Получив результат он устанавливается в объект future и он становится полным, из него можно извлечь результат.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(Код)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Если вызвать result раньше чем он установлен, то возникнет исключение InvalidState.

Будущие объекты также можно использовать в выражениях await. Это означает «я посплю, пока в будущем объекте не будет установлено значение, с которым я могу работать, а когда оно появится, разбуди меня и дай возможность его обработать».

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(Код)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Task** является комбинацией сопрограммы и future. Создавая задачу, мы создаем пустой объект future и запускаем сопрограмму. А когда сопрограмма завершится с результатом или вследствие исключения, мы записываем этот результат или объект-исключение во future.



Измерение времени выполнения сопрограммы с помощью декораторов.

**Декоратор** в Python - это паттерн, который позволяет расширять функциональность существующей функции, не изменяя ее код. Мы можем «перехватить» вызываемую функцию и применить декоратор до или после вызова.

Примерами **блокирующих API** является библиотека requests или функция time.sleep.

**Отладочный режим**. При работе в отладочном режиме печатаются полезные сообщения, когда сопрограмма или задача работают больше 100 мс. Кроме того, если для некоторой сопрограммы отсутствует await, то возбуждается исключение, показывающее, в каком месте следовало бы добавить await. asyncio.run(coroutine(), debug=True) - вариант включения отладочного режима python3 -X dev program.py - через терминал (добавляем -Х dev перед именем приложения)

**Сокеты и Вебсокеты**

Сокет — это программный интерфейс для обеспечения информационного обмена между процессами.

Сокеты работают на транспортном уровне протоколв и соответственно бывают 2 типов:

Потоковые (на основе TCP, в коде обозначаются SOCK\_STREAM) - сокеты с установленным соединением на основе протокола TCP, передают поток байтов, который может быть двунаправленным - т.е. приложение может и получать и отправлять данные.

Дейтаграммные (на основе UDP, в коде обозначаются SOCK\_DGRAM) - сокеты, не требующие установления явного соединения между ними. Сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

Сокет состоит из IP-адреса и порта.

IP-адрес - уникальный сетевой адрес узла в компьютерной сети, построенной по протоколу IP. В версии протокола IPv4 IP-адрес имеет длину 4 байта (например, 192.168.0.3), а в версии протокола IPv6 IP-адрес имеет длину 16 байт (например, 2001:0db8:85a3:0000:0000:8a2e:0370:7334). IP-адрес должен быть уникален.

Порт - натуральное число, записываемое в заголовках протоколов транспортного уровня (TCP, UDP и др.). Порт используется для определения процесса-получателя пакета в пределах одного хоста.

Функции объекта socket

socket.bind(address) - Привязывает сокет к адресу address (инициализирует IP-адрес и порт). Сокет не должен быть привязан до этого.

socket.listen([backlog]) - Переводит сервер в режим приема соединений. Параметр``backlog (int)`` – количество соединений, которые будет принимать сервер.

socket.accept() - Принимает соединение и блокирует приложение в ожидании сообщения от клиента. В результате возвращает кортеж:

conn: объект соединения (сокет), который можно использовать для отправки/получения данных;

address: адрес клиента.

socket.recv(bufsize[, flags]) - Читает и возвращает данные в двоичном формате (набор байтов) из сокета. Параметр bufsize (int) – максимальное количество байтов в одном сообщении.

socket.send(bytes[, flags]) - Отправляет данные клиенту и возвращает количество отправленных байт. Параметр bytes (bytes) – двоичные данные.

socket.close() - Закрывает сокет.

Работа с сокетом во многом схожа с работой с файловым объектом. Принцип - открыли соединение - считали данные - закрыли соединение.

метод encode('utf-8') - позволяет перевести данные из строки в байты (str -> bytes);

метод decode('utf-8') - позволяет перевести данные из байт в строку (bytes -> str);

**socket.socket()** - создаётся объект сокета

**with** socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) **as** s:  
 s.bind((HOST, PORT))

Так как используем with… , то вызывать s.close() не нужно

**AF\_INET** — это семейство интернет-адресов для IPv4

**SOCK\_STREAM** — это тип сокета для TCP и протокол, который будет использоваться для передачи сообщений в сети

**.bind()** применяется для привязки сокета к конкретному сетевому интерфейсу и номеру порта

**.listen(5)** – принимаются подключения на, а сам сервер становится «прослушиваемым» сокетом. 5 – количество одновременно ожидающих обработки исключений, все остальные будут отбрасываться.

**.accept()** - выполнение блокируется, и ожидается входящее подключение. – **блокирующая функция**

**s.sendall(data)** - гарантирует отправку всех данных, вызывает низкоуровневый s.send(data) пока не будет отправлено все

**s.send(data)** – нужно самостоятельно управлять повторными попытками отправки данных, если они дошли не все.

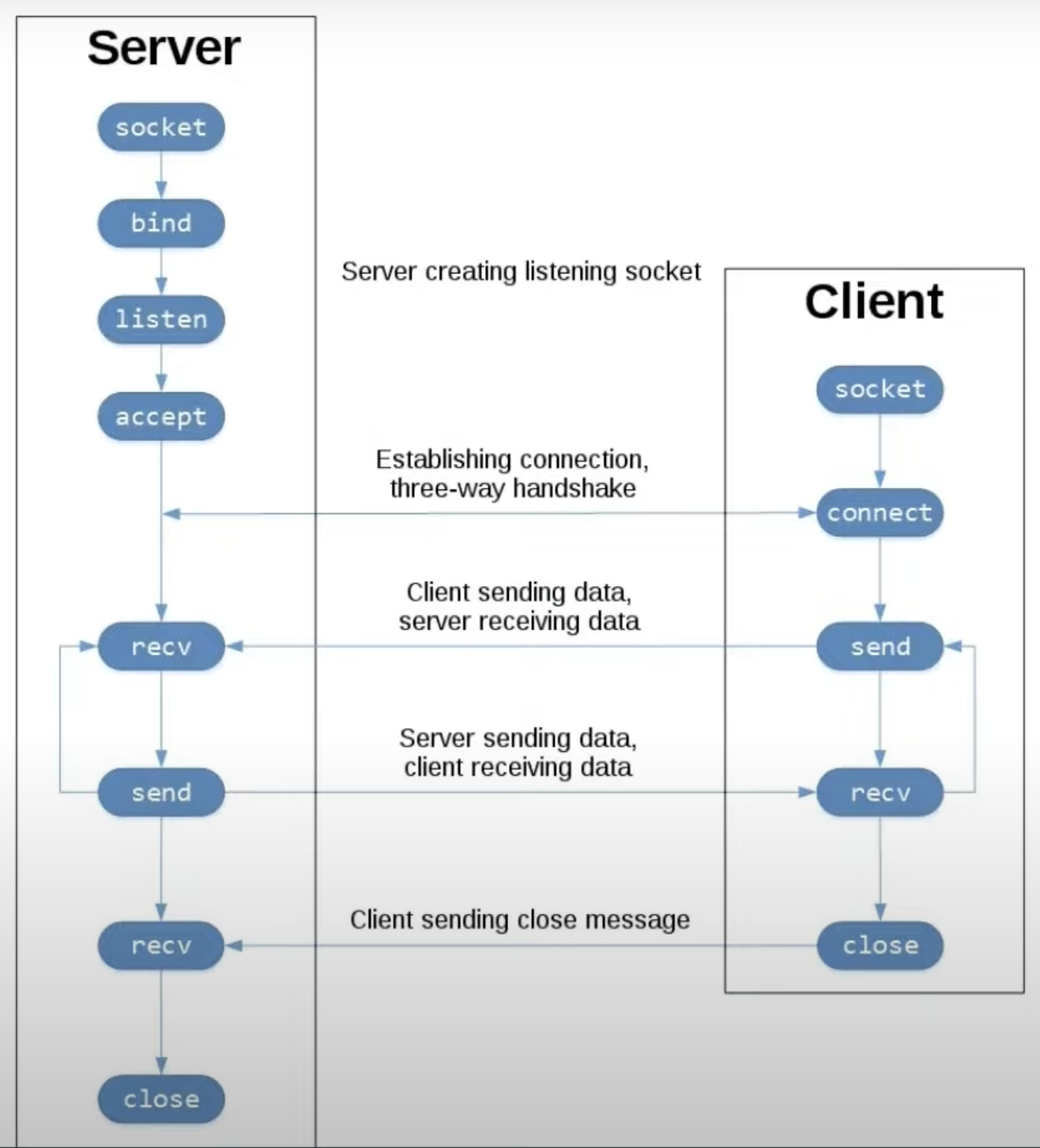
**Увеличение размера блока данных**, который читаем и отправляем за раз, может увеличить скорость передачи данных, поскольку сокращается количество операций чтения и записи. Однако это также может увеличить использование памяти, поскольку придется хранить больший блок данных в памяти за один раз.

data = conn**.recv(**1400000**)** – принять данные из сокета в размере 1400000 – **блокиующая функция**

for data in iter(lambda: f.read(1400000), b''):

**Lambda функции**: Это анонимные функции в Python, которые объявляются с помощью ключевого слова lambda. Они могут принимать любое количество аргументов, но могут иметь только одно выражение. lambda: f.read(1400000) это функция без имени, которая читает 1400000 байтов из файла f.

**Функция iter()**: Эта функция в Python возвращает итератор. Она принимает два аргумента: функцию и “стоп-значение”. Она продолжает вызывать переданную функцию до тех пор, пока она не вернет “стоп-значение”. iter(lambda: f.read(1400000), b'') будет продолжать читать 1400000 байтов из файла, пока не достигнет конца файла (когда f.read(1400000) вернет пустой байтовый объект b'').



**struct.pack()** - функция в Python, которая преобразует Python значения в байты, используя формат, указанный в первом аргументе.

msg = struct.pack('>I', len(msg)) + msg - создаем новое сообщение, которое начинается с четырехбайтового префикса, содержащего длину оригинального сообщения, а затем следует само оригинальное сообщение. Это позволяет получателю сначала прочитать префикс, чтобы узнать, сколько байт следует прочитать для получения полного сообщения.

Функция **struct.unpack**('>I', raw\_msglen)[0] :

struct.unpack(): - Функция unpack() преобразует байты обратно в исходное значение с использованием указанного формата

'>I': Это строка формата, которая указывает, как данные должны быть распакованы. Символ '>' указывает на “big-endian” порядок байтов (сначала идут старшие байты), а 'I' означает беззнаковое целое число.

raw\_msglen: Это байтовые данные, которые нужно распаковать. В данном случае, это 4 байта, которые были прочитаны из сокета и которые представляют собой длину остального сообщения.

[0]: Поскольку struct.unpack() возвращает кортеж, даже если он содержит только один элемент, используем индекс [0] для извлечения этого одного элемента из кортежа.

msglen = struct.unpack('>I', raw\_msglen)[0] распаковывает 4 байта raw\_msglen обратно в исходное целое число, которое представляет собой длину остального сообщения. Это число затем сохраняется в переменной msglen.

Кодирование размера передаваемого изображения в 4 байта связано с выбором формата для функции struct.pack(). В данном случае, '>I' означает 32-битное беззнаковое целое число в big-endian порядке байтов. Это позволяет кодировать размеры изображений до 4294967295 байтов (или примерно 4 ГБ), что обычно достаточно для большинства изображений.

Выбор 4 байтов для кодирования размера изображения является компромиссом между максимально возможным размером изображения и количеством данных, которые нужно передать для указания этого размера.

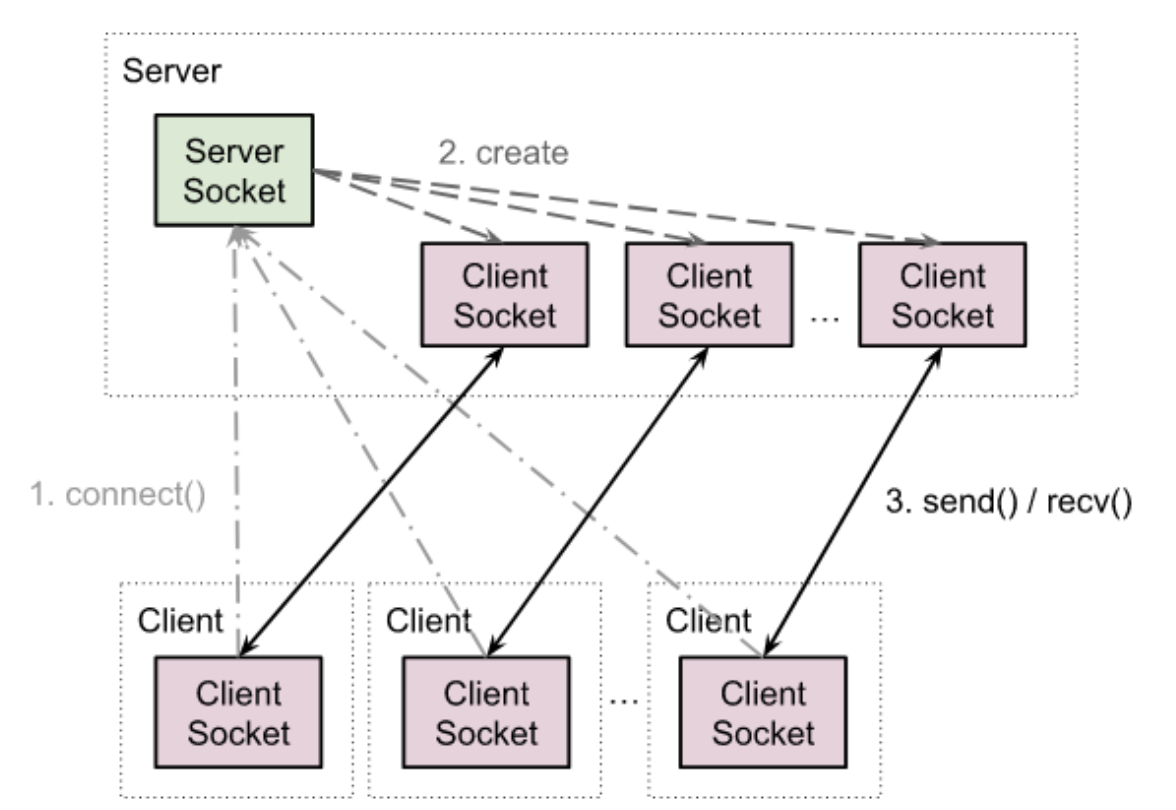
**https://habr.com/ru/articles/676110/**

**Программа** — это определенная последовательность инструкций и данных, решающих определенную задачу, которая хранится в памяти. Программа, которая выполняется, называется **процессом**. Самый главный процесс — это операционная система (ОС). Она в свою очередь может запускать другие программы как процессы ОС. Программы, которые являются частью ОС, называются системными. Все остальные — прикладными, или приложениями. Они запускаются пользователем и предназначены для решения его задач.

Программа может сама управлять только ходом собственного исполнения (инструкции if, else, while, for и так далее) и вычислениями (+, -, \*, /). Всё же остальное — выделение или освобождение блока памяти, вывод на экран, обращение к мыши и клавиатуре, сети, файлам, жестким дискам и другим устройствам ввода-вывода — программа может получить, только обратившись к ОС через системный вызов.

**межпроцессное взаимодействия (Inter-process communication, IPC)** - для разных способов обмена данных между процессами.

Сокеты бывают двух типов: клиентские и серверные. **Серверные** не делают ничего, кроме приема запросов на установление соединения и создания клиентского сокета для каждого нового подключения. **Клиентские** **сокеты** не делают ничего, кроме как обмениваются данными. И на сервере, и на клиенте клиентские сокеты идентичны. Соединение между клиентскими сокетами является одного ранга — peer-to-peer.



**Сервер** отличается от клиента наличием серверного сокета. Из этого вытекает необходимость поддерживать на сервере одновременную работу нескольких клиентских сокетов, тогда как в клиентском приложении достаточно одного сокета. Поэтому даже на сервере основная работа приходится именно на клиентские сокеты, а не на серверные.

Слово сервер имеет несколько значений. Прежде всего, так мы называем программу, на которой запущены серверные сокеты. Кроме того, так называют еще и машину, на которой запущена такая программа. Аналогично и с клиентом. Этим словом называют программу, использующую только клиентские сокеты, и машину, на которой она запущена. Вся эта схема, вместе взятая, образует **клиент-серверную модель взаимодействия**. Суть ее в том, что разные клиенты подключаются к одному серверу, который координирует и управляет работой клиентов.

**AF\_INET** — означает интернет сокет по протоколу IPv4 (также существуют: AF\_INET6 — для IPv6, AF\_IPX — для IPX, AF\_UNIX — для Unix-сокетов).

**SOCK\_STREAM** — говорит о том, что в качестве транспортного протокола будет использоваться TCP. Второй распространенный вариант — это UDP (SOCK\_DGRAM). Первый используется там, где важна надежность, второй — где скорость. TCP гарантирует доставку всех сообщений, причем в том же порядке, в котором они были переданы. UDP возвращает сообщения сразу по прибытии — без проверки, обработки и подтверждений приема. Поэтому он гораздо быстрее TCP.

**HOST** — это IP адрес системы, где запущен сервер. Если сокет должен быть виден только снаружи, из других систем, то нужно взять значение из server\_sock.gethostname(). Если он должен быть виден только изнутри, то необходимо задать "localhost" или "127.0.0.1". Значение "" осуществляет привязку ко всем доступным для данной машины адресам: и локальным, и глобальным. Заметим, что использование "localhost" для локального доступа позволяет пропустить несколько слоев сетевого кода, что быстрее, чем использовать HOST="".

**PORT** — порт сервера. Так как малые значения (до 1024) большей частью зарезервированы, то рекомендуется использовать любое 4-значное число свыше 1024.

При вызове **send()** переданная в аргументе последовательность байтов добавляется в **буфер отправки** для соответствующего соединения. Когда именно буфер будет отправлен, зависит от операционной системы и уровня ее загруженности.

Аналогично, существует буфер приема данных, который пополняется с каждым новым полученным по сети пакетом. В методе **recv()** мы передаем максимальное число байтов, которое мы желаем получить. Если в буфере меньше данных, чем мы запрашиваем, то функция возвращает, сколько есть. Лучше всего, если число будет степенью двойки и не меньше максимальной ожидаемой длины сообщений, чтобы не требовалось несколько раз вызывать recv() для получения одного и того же сообщения.

**Полный же жизненный цикл сокета** завершается с закрытием соединения. Закрывать соединение необходимо всегда явным образом — вызовом метода **close()**. Даже не смотря на то, что close() вызывается автоматически при сборе мусора (периодической очистки памяти от ранее удаленных переменных). Если, например, убить клиентский процесс без вызова close(), то на сервере об этом не узнают и так и будут ждать от него сообщений хоть целую вечность.

Метод close() высвобождает ресурсы, выделенные сокету, но еще не гарантирует немедленного закрытия соединения. Если момент закрытия важен, то нужно явно вызывать метод shutdown(how) с одним из значений SHUT\_RD=0, SHUT\_WR=1, SHUT\_RDWR=2. Так, например, при отсылке HTTP-запроса клиент вызовом shutdown(SHUT\_WR) говорит серверу, что передача закончилась и больше данных не будет — можно начинать обработку запроса. При этом тот же сокет все еще может получать данные с сервера и ждет ответ. На сервере recv() при этом возвращает b"" — значение длиной в 0 байтов, что означает состояние End of File (EOF).

Чтобы не вызывать каждый раз close(), весь блок кода, работающего с сокетами, можно также обернуть в конструкцию with

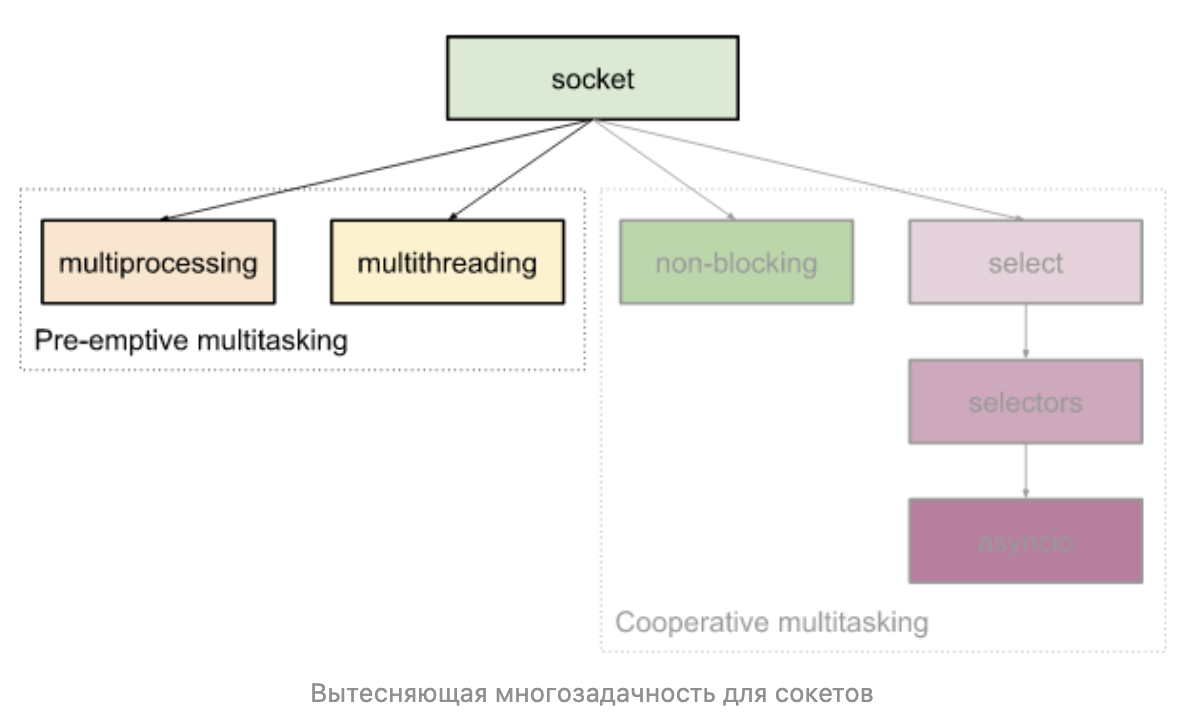
Методы **accept(), connect() и recv()** класса socket являются блокирующими. Это значит, что после их вызова выполнение программы остановится и не возобновится до тех пор, пока не поступят соответствующие данные из сети. А если программа не выполняется, то и другие соединения и задачи также не обрабатываются.

В Python есть несколько способов реализовать **многозадачность**:

- Процессы (создаются копии самого процесса приложения).

- Потоки (выполняются в одном процессе).

- Системный вызов select и его аналоги (выполняются в одном потоке).



Код по созданию нового подключения достаточно стандартный, и чтобы его каждый раз не дублировать, разработчикам пошли на встречу и сделали специальные классы TCPServer, UDPServer и их Threading- и Forking-версии:

**from** socketserver **import** ForkingTCPServer  
**from** socketserver **import** ThreadingTCPServer  
  
HOST, PORT = "", 50007  
**if** \_\_name\_\_ == "\_\_main\_\_":  
 # with ForkingTCPServer((HOST, PORT), ConnectionHandler) as server:  
 **with** ThreadingTCPServer((HOST, PORT), ConnectionHandler) **as** server:  
 server.serve\_forever()

Функция-обработчик соединения была обернута в класс, что позволяет нам теперь хранить состояние и создавать более сложную логику, состоящую из нескольких методов, которые можно наследовать и переопределять:

**Блокировщик**

нужно навсего заблокировать доступ для других потоков к переменной, пока все изменения окончательно не будут записаны в память.

lock.acquire() # Wait if already acquired  
x = x + 1  
lock.release()

Блокировщик устроен так, чтобы метод acquire() нельзя было вызывать 2 раза подряд без вызова release() в промежутке. При первом вызове устанавливается флаг — блокировщик занят. Если из другого потока будет вызван acquire() того же объекта lock, выполнение потока остановится, пока в первом потоке выполнение не дойдет до вызова release(). Тогда только объект lock освободится, и его сможет занять другой поток.

Это называется **синхронизацией потоков.**

Lock реализован как менеджер контекста и может использоваться с оператором with, то запись можно упростить:

# Thread 1  
**with** lock:  
 x = x + 1

# Thread 2  
**with** lock:  
 x = x \* 2

Функции acquire() и release() будут вызваны автоматически при входе и выходе из блока with, соответственно, так как:

**class** Lock:  
 # ...  
 **def** \_\_enter\_\_(self):  
 self.acquire()  
  
 **def** \_\_exit\_\_(self, \*args):  
 self.release()

Но тут появляется другая опасность. Если, допустим, используется несколько блокировщиков, есть вероятность взаимной блокировки двух потоков и их полной остановки на веки вечные. Например, первый поток занимает (acquire) блокировщик A, а второй — B, после чего продолжают выполнение. Потом, не освободив предыдущие, они доходят до блокировщиков B и A, соответственно. Первый никогда не пройдет через B, потому что он уже занят вторым потоком, а второй — A, потому что тот занят первым. Такая ситуация называется взаимная блокировка, или **deadlock**

Чтобы такого не случалось, нужно создать иерархию блокировщиков и не использовать нижележащие без блокировки сначала вышележащих. То есть блокировщик B всегда должен идти после A, даже если A тут по логике и не нужен:

# Thread 2 (no deadlocks)  
**with** lock\_a:  
 **with** lock\_b:  
 x = x \* 2

https://habr.com/ru/articles/676118/

**Мультиплексирование с помощью select**

Другое дело — использование системного вызова select (в Python он вызывается функцией select.select()). Тут переключение между задачами осуществляется вручную и полностью управляется из кода. Никаких неожиданных перескакиваний из задачи в задачу, как это было в потоках или процессах, нет. Поэтому здесь не нужно никакой синхронизации и защиты от взаимных блокировок. Весь код выполняется синхронно — инструкция за инструкцией. Переключение задач — это тоже просто очередная последовательность инструкций, и нам заранее известно, когда она выполнится.

Принцип работы функции select() довольно простой — в аргументах передается список всех имеющихся сокетов, а возвращается список всех готовых. В соответствии с полученным сокетом мы сами выбираем, какую следующую задачу нужно выполнять. Когда задача выполнилась или перешла в режим ожидания, функция select() вызывается снова, и все повторяется.

Так, все процессорное время распределяется между разными задачами. То есть несколько потоков выполнения мультиплексируются в одном потоке выполнения процессора. Никаких случайных перескакиваний и при этом — никаких простоев. Если есть хоть один готовый сокет, всегда будет что-то выполняться. Так, с помощью полностью синхронного кода мы добились реальной асинхронности в выполнении задач. Вот так синхронность перешла в асинхронность.

Многозадачность, реализованная через select(), называется совместной, или кооперативной (cooperative), потому что, прежде, чем начала выполняться следующая задача, нужно, чтобы предыдущая явно объявила о своей готовности отдать процессорное время другим задачам. То есть задачи очевидным образом согласуют между собой момент переключения задач.

В противоположность кооперативной, процессы и потоки реализуют вытесняющую (pre-emptive) многозадачность. Она так называется, потому что если один из процессов или потоков выполняется слишком долго, то он может быть прерван при окончании выделенного ему интервала времени или при исчерпании лимита выполненных инструкций (time-slicing). Один процесс, таким образом, вытесняет другой, даже если тот еще может выполняться и выполняться.

В реальной разработке необходимость осуществлять синхронизацию потоков может быть весьма дорогим и сомнительным удовольствием. Во-первых, это дополнительная работа и напряжение серых клеточек. А кто хочет работать (или платить) больше? Во-вторых, это сложно, а, значит, тут обязательно будут ошибки. А в случае потоков — трудноповторимые и трудноуловимые. Так что многочасовой багфиксинг обеспечен, при том — на пустом месте. Не говоря уже о постоянных сомнениях: в логике ли баг на этот раз или опять ошибка в синхронизации? И что самое неприятное, можно 1000 раз прогнать этот код и не получить вообще никакой ошибки. Так что никогда нет уверенности, что программа написана правильно, даже если 100% кода покрыта unit-тестами и все они зеленые.

Именно по этой причине использование select() и решений, построенных на его основе, оказывается куда предпочтительнее потоков. Ведь момент переключения выполнения всегда известен заранее, а значит, никакой синхронизации проводить не нужно. Все уже синхронизировано по определению. А использование интерпретатором GIL лишает потоки единственного их преимущества — параллельности исполнения.

Вот почему потоки, как и процессы, нужно знать, но никогда не использовать! А применять функцию select() и асинхронное программирование.

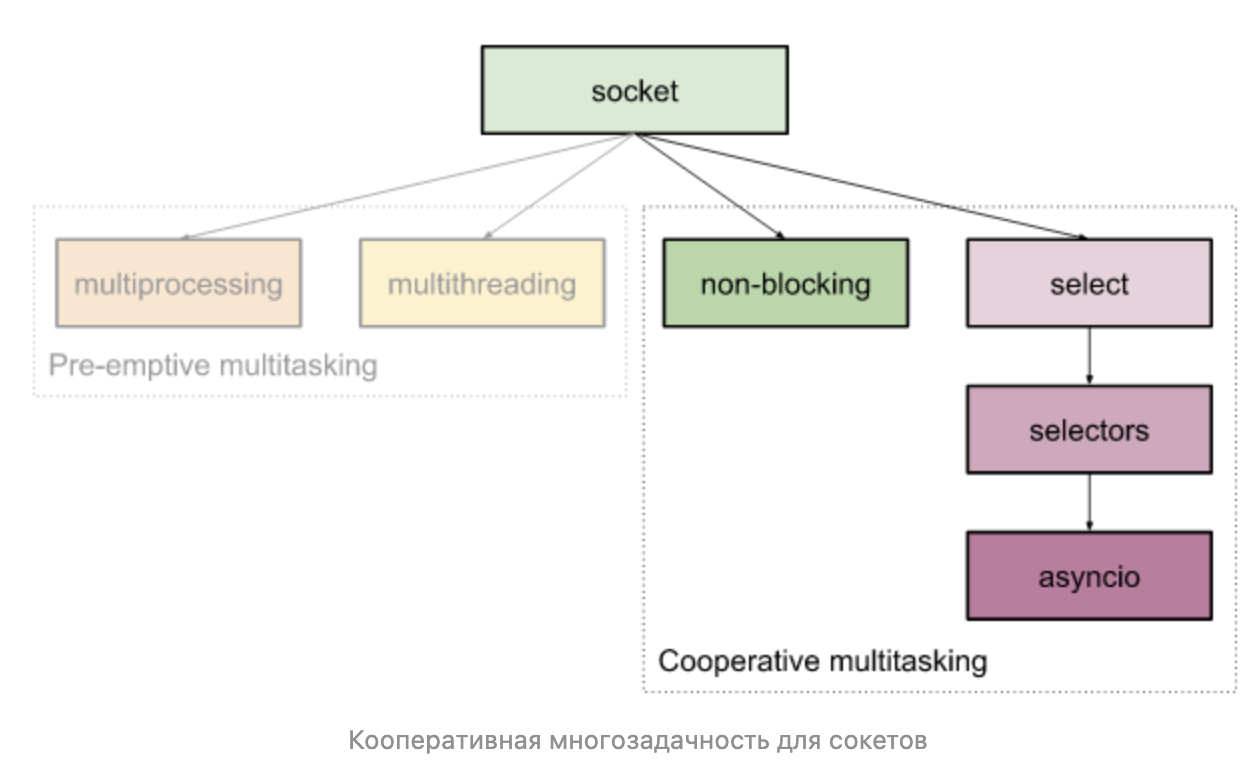
Вытесняющая многозадачность:

- Процессы (выполняются в одной ОС).

- Потоки (выполняются в одном процессе).

Кооперативная многозадачность:

- Сопрограммы (выполняются в одном потоке).



**Неблокирующие сокеты**

сокеты допускают переключение в неблокирующий режим работы. Для этого нужно перед первым использованием сокета вызвать у него метод setblocking(False). Тогда ранее блокирующие методы будут сразу или возвращать данные, если они есть в наличии, или генерировать исключение BlockingIOError, если их нет. Поэтому мы в бесконечном цикле можем опрашивать все соединения по очереди на предмет готовности данных. Если возникает исключение BlockingIOError, мы просто переходим к следующему сокету:

**import** socket  
  
HOST, PORT = "", 50007  
connections = []  
**if** \_\_name\_\_ == "\_\_main\_\_":  
 **with** socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) **as** serv\_sock:  
 serv\_sock.bind((HOST, PORT))  
 serv\_sock.listen(1)  
 serv\_sock.setblocking(**False**) # Important!  
 **while** **True**:  
 **try**:  
 # print("Try to accept a new connection...")  
 sock, addr = serv\_sock.accept()  
 sock.setblocking(**False**)  
 print("Connected by", addr)  
 connections.append((sock, addr))  
 **except** BlockingIOError:  
 # print("No connections are waiting to be accepted")  
 **pass**  
 **for** sock, addr **in** connections.copy():  
 print("Try to receive data from:", sock, addr)  
 **try**:  
 data = sock.recv(1024)  
 **except** ConnectionError:  
 print(f"Client suddenly closed while receiving from {addr}")  
 connections.remove((sock, addr))  
 sock.close()  
 **continue**  
 **except** BlockingIOError:  
 # No data received  
 **continue**  
 print(f"Received: {data} from: {addr}")  
 **if** **not** data:  
 connections.remove((sock, addr))  
 sock.close()  
 print("Disconnected by", addr)  
 **continue**  
 data = data.upper()  
 print(f"Send: {data} to: {addr}")  
 **try**:  
 sock.sendall(data)  
 **except** ConnectionError:  
 print(f"Client suddenly closed, cannot send to {addr}")  
 connections.remove((sock, addr))  
 sock.close()  
 **continue**

Не трудно догадаться, что такое решение будет полностью загружать процессор, как и всякий другой бесконечный цикл, вне зависимости от того, есть там что обрабатывать или нет. При этом чаще всего данные будут поступать медленнее, чем будет проходить полный цикл опроса всех клиентских сокетов. А значит, большая часть кода будет исполняться впустую — sock.recv() → BlockingIOError → sock.recv() → BlockingIOError и т.д.

Также, если первый сокет в списке готов, а мы сейчас опрашиваем только второй, то придется пройтись по всему списку неготовых сокетов, пока мы дойдем до первого — реально готового соединения.

Я уже не говорю про затраты времени на обработку исключений, которые являются неотъемлемой составляющей данного способа. Даже если один из сокетов был бы всегда готов, когда бы его не спросили, то прежде, чем к нему попасть, пришлось бы поймать и обработать множество исключений BlockingIOError от других сокетов.

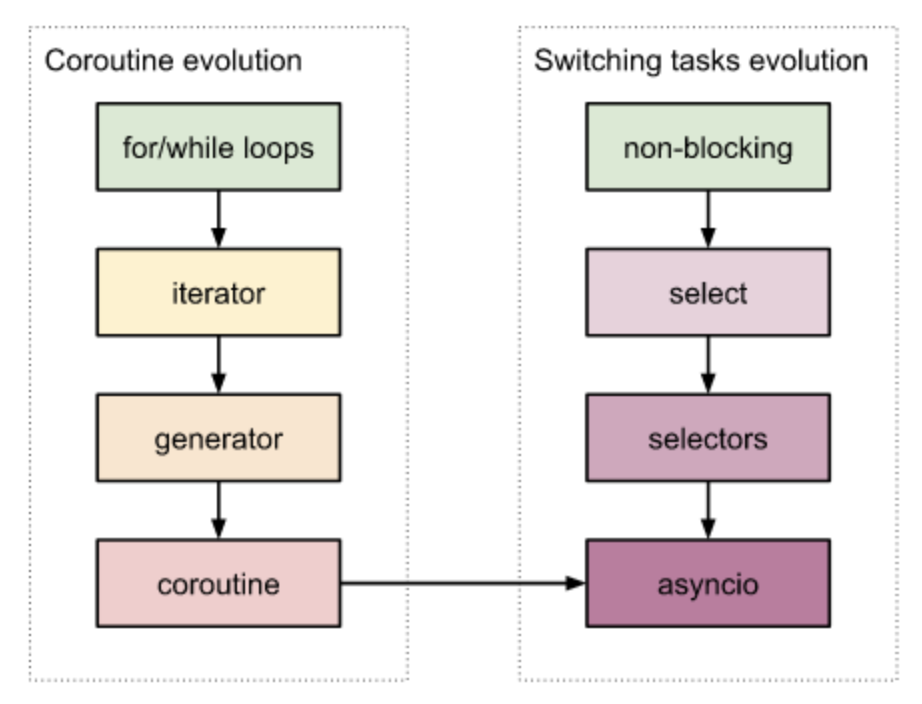
Вот бы каким-нибудь образом опрашивать только те сокеты, про которых достоверно известно, что данные для них уже пришли и готовы для обработки, чтобы не перебирать все сокеты подряд. Это сэкономило бы нам немало процессорного времени. Специально для таких случаев в ОС и создали системный вызов select.

**Системный вызов select**

Системный вызов select — это такой вызов, которые принимает в качестве параметров список дескрипторов потоков ввода-вывода, которые нужно проанализировать, а на выходе возвращает список тех из них, которые готовы к работе.

Каждый **дескриптор** — это уникальное целое число, которое возвращает система при открытии файла, сокета или другого потока ввода-вывода. С помощью этого числа потоки можно отличать друг от друга. Класс socket в Python — это просто удобная обертка вокруг такого числа-дескриптора.

Если выразиться проще, то передавая в select() список всех сокетов, мы на выходе получим список тех из них, которые готовы к обработке.



Итерирование, Итераторы, генераторы, вложенные генераторы, сопрограммы, колбеки, селекты, селекторы,