**UrbanUniversity**

Дипломная работа

**Сравнение различных подходов к реализации асинхронного программирования: asyncio, threading и multiprocessing**

**Смирнова Людмила**

**Г. Нижний Новгород, 2024**

**Содержание**

|  |  |
| --- | --- |
| **Введение** | **3** |
| **Виды нагрузок и подходы** | **4** |
| **Библиотека asyncio** | **5** |
| **Процесс и поток** | **9** |
| **Threading** | **10** |
| **Мультипроцессность** | **14** |
| **Практическое сравнение различных подходов** | **15** |
| **Заключение** | **18** |
| **Список литературы** | **19** |

**Введение**

Сейчас высокоуровневый язык программирования Python является одним из самых востребованных языков в сфере информационных технологий. На нем реализованы многие проекты, связанные с обработкой текстов на естественном языке, анализом больших данных, машинным обучением и пр.

В большинстве наших Python-проектов мы пишем обычный код, который выполняется строка за строкой. Если в каком-то месте программе нужно сделать запрос, например, на сервер погоды, она ждёт ответа, а затем продолжает работу. Такой код называется синхронным: программа ждёт, пока сервисы синхронизируются между собой, и продолжает работу после этого. В простых проектах, где мало запросов или время ожидания некритично, это работает. В высоконагруженных сервисах такой подход не годится. Поэтому в современных сложных Python-программах чаще всего применяют асинхронный код.

Асинхронное программирование — концепция [программирования](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), которая заключается в том, что результат выполнения функции доступен не сразу, а через некоторое время в виде некоторого асинхронного (нарушающего обычный порядок выполнения) вызова.

Использование технологий асинхронного программирования позволяет значительно увеличить скорость работы программ на Python, обходя различные ограничения. Асинхронное программирование применяется в решении многих задач, например, при создании веб-сервера, клиент-серверного приложения, при извлечении данных с информационного ресурса веб-краулером.

Для реализации асинхронного кода применяются различные подходы, такие как:

- asyncio - библиотека для асинхронного ввода-вывода в Python,

-threading - модуль для многопоточного программирования,

-multiprocessing - модуль для многопроцессорного программирования.

Целью данной работы является изучение различных подходов к асинхронному программированию, сравнение их производительности и уместности для выполнения разных видов задач.

**Виды нагрузок и подходы**

Выбор между процессами, потоками или применением асинхронного подхода зависит в первую очередь от нагрузки. Она бывает двух видов:

* **CPU bound** — нагрузка на процессор, когда он активно работает, например, при выполнении математических расчётов или вычислениях в «тяжёлых» компьютерных играх.
* **I/O bound** — процессор ожидает операции ввода-вывода, не слишком интенсивно работая. Из примеров можно привести запросы к базам данных или API каких-нибудь сервисов, то есть к внешним ресурсам. Другими словами, это нагрузка, длительность обработки которой не зависит от скорости работы процессора.

Есть три основных подхода к обработке нагрузки:

* **Асинхронность** (asyncio). В этом случае используют конкурентные потоки, которые выполняются НЕ параллельно. Можно сказать, что асинхронность — это параллельное ожидание. Работа в один поток накладывает некоторые ограничения на вид нагрузки. Например, главный поток асинхронного алгоритма можно заблокировать.
* **Многопоточность** (threading). Все потоки используют общую память. Нагрузка вида CPU bound обрабатывается процессором в один поток — самое интересное, что это работает так именно в Python. В различных языках программирования это может быть реализовано по-разному.
* **Многопроцессность** (multiprocessing). У каждого процесса своя область памяти. Более того, каждый процесс — это, в сущности, отдельный интерпретатор Python со своим GIL и соответствующими расходами ресурсов.

GIL - это встроенный механизм, который обеспечивает выполнение ровно одного потока в любой момент времени. GIL облегчает реализацию интерпретатора, защищая объекты от одновременного доступа из нескольких потоков. По этой причине, создание несколько потоков не приведет к их одновременному исполнению на разных ядрах процессора.

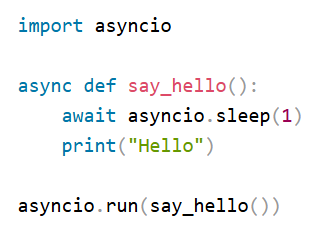
**Библиотека asyncio**

Это стандартная библиотека Python для реализации алгоритмов асинхронного программирования. В нее входят инструменты для работы с корутинами, менеджерами контекста, асинхронными генераторами.

**Корутины**

Это особые функции,  определяемые словом async def, способные сначала остановить свое выполнение с помощью ключевого слова await, что позволяет другим корутинам выполняться в это время. Так как корутины запускаются параллельно в одном потоке, не занимая дополнительной памяти, ресурсы расходуются экономнее, чем при последовательном выполнении задач. В то же время в asyncio существует свой пул потоков — так называемых «исполнителей» (executors). Если запустить в нем параллельно слишком много процессов, есть риск чрезмерного расхода и последующей нехватки ресурсов.

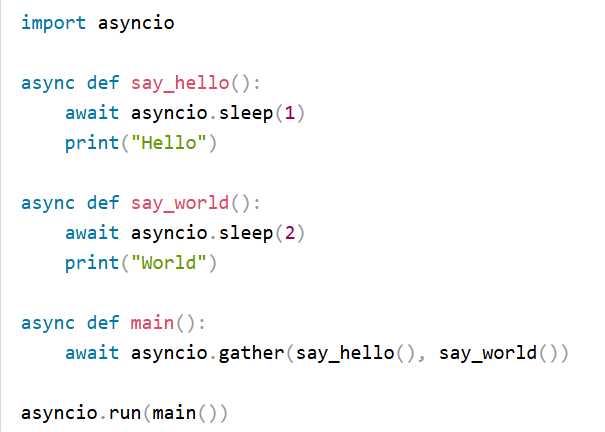
Асинхронные функции часто используются для выполнения операций с задержкой, таких как ожидание ответа от сервера или чтение данных из файла.

****

В этом примере функция say\_hello приостанавливает выполнение на одну секунду с помощью await asyncio.sleep(1), а затем выводит "Hello". Это позволяет другим задачам выполняться параллельно, пока текущая задача ожидает завершения задержки.

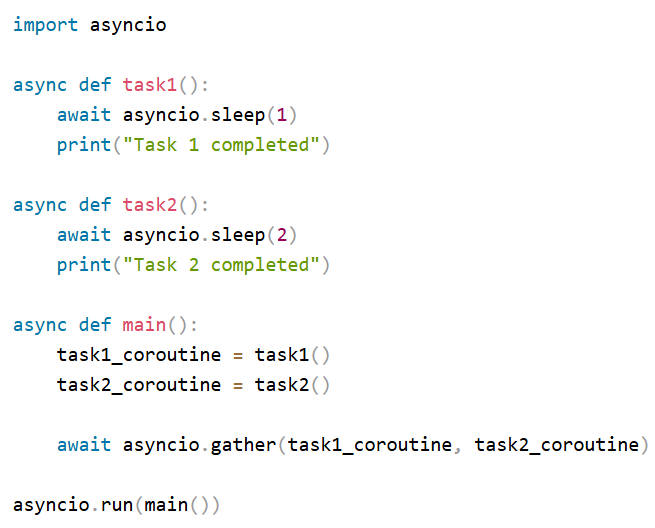
**Асинхронная функция с несколькими задачами**

Асинхронное программирование позволяет выполнять несколько задач одновременно, что значительно улучшает производительность.



В этом примере функции say\_hello и say\_world выполняются одновременно, но say\_hello завершится первой, так как её задержка меньше. Использование asyncio.gather позволяет запускать несколько асинхронных задач параллельно и ожидать их завершения.

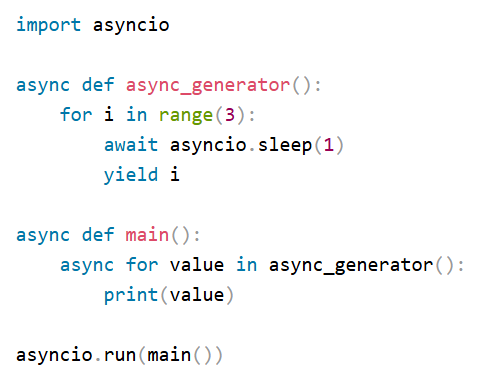
**Создание и выполнение асинхронных задач** с помощью asyncio позволяет эффективно управлять многозадачностью.



В этом примере создаются две асинхронные задачи task1 и task2, которые выполняются параллельно с помощью asyncio.gather. Это позволяет значительно сократить общее время выполнения задач.

**Асинхронные генераторы**

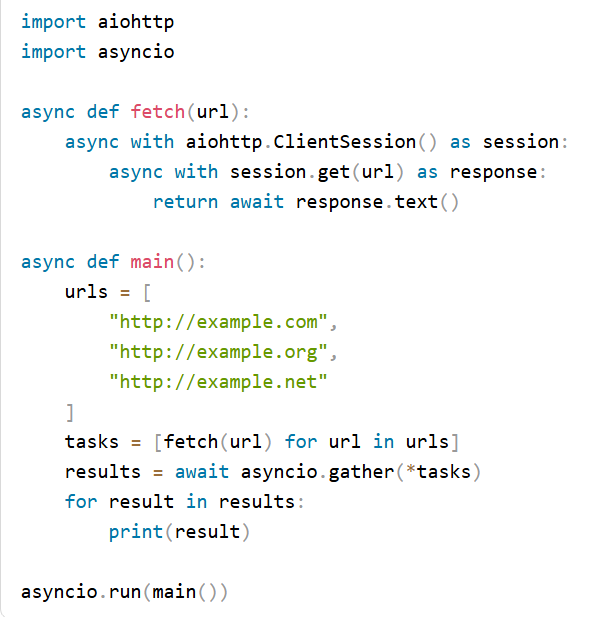
Асинхронные генераторы позволяют создавать итераторы, которые могут использовать await внутри себя. Это особенно полезно для работы с потоками данных, которые поступают асинхронно.



В этом примере асинхронный генератор async\_generator создает итератор, который возвращает значения с задержкой в одну секунду. Асинхронный цикл async for позволяет обрабатывать эти значения по мере их поступления.

**Асинхронные сетевые запросы**

Асинхронные сетевые запросы могут значительно ускорить работу с API и другими сетевыми ресурсами. Использование асинхронных библиотек, таких как aiohttp, позволяет выполнять множество запросов параллельно.



В этом примере асинхронная функция fetch выполняет сетевой запрос и возвращает текст ответа. Функция main создает задачи для каждого URL и выполняет их параллельно с помощью asyncio.gather. Это позволяет значительно сократить время выполнения множества сетевых запросов.

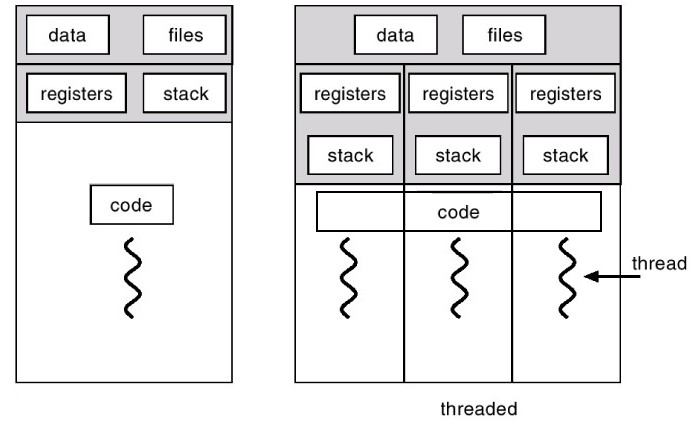
**Процесс и поток**

Процесс - исполняемый экземпляр какой-либо программы. Каждый процесс состоит из следующих элементов:

* образ машинного кода;
* область памяти, в которую включается исполняемый код, данные процесса (входные и выходные данные), стек вызовов и куча (для хранения динамически создаваемых данных);
* дескрипторы операционной системы (например, файловые дескрипторы);
* состояние процесса.

В целях стабильности и безопасности, в современных операционных системах каждый процесс имеет прямой доступ только с своим собственным ресурсам. Доступ к ресурсам другого процесса возможен через межпроцессное взаимодействие (например, посредством файлов, при помощи именованных и неименованных каналов и другие).

Сам процесс может быть разделен на так называемые потоки. Поток (поток выполнения, thread) - наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. В отличии от нескольких процессов, потоки существуют внутри одного процесса и имеют доступ к ресурсам этого процесса. Каждый поток обладает собственным набором регистров и собственным стеком вызова, но доступ к ним имеют и другие потоки.



При работе с потоками стоит учесть несколько моментов:

* одно ядро процессора в один момент может исполнять только один поток;
* потоки одного процесса могут исполняться физически одновременно (на разных ядрах);
* бессмысленно порождать потоков больше, чем у вас есть ядер.

Потоки имеют несколько применений. Первое - ускорение работы программы. Ускорение достигается за счет параллельного выполнения независимых друг от друга вычислений. Например, при численном интегрировании область интегрирования может быть разбита на 3 участка. На каждый участок создается свой поток, в котором численно вычисляется интеграл для конкретного участка. Второе - независимое исполнение операций. Отличие этого случая от первого хорошо видно на следующем примере. Пусть есть приложение с графическим интерфейсом, где весь код выполняется в одном потоке. При выполнении какой-нибудь долгой операции (например, копирование файла) интерфейс приложения просто перестанет отвечать до тех пор, пока долгий процесс не завершится. В таком случае в один поток помещается работа графического интерфейса, в другой - остальные вычисления. В таком случае интерфейс позволит проводить другие операции даже во время выполнения долгой операции в другом потоке (например, заполнение прогресс бара в процессе копирования файла).

**T**[**hreading**](https://cs.mipt.ru/advanced_python/lessons/lab11.html#toc-entry-2)

В Python работа с потоками осуществляется при помощи стандартной библиотеки threading. В библиотеке представлен класс Thread для создания потока выполнения. У него есть 4 основных метода:

* start(): Запускает выполнение потока.
* join(): Текущий поток приостанавливается и ждёт завершения запущенного потока.
* is\_alive(): Возвращает True, если поток выполняется.
* run(): Метод, содержащий код, который будет выполняться в потоке. Переопределяется при наследовании от класса Thread

Конструктор класса Thread имеет следующие аргументы:

* group должно быть None;
* target является исполняемым объектом (по умолчанию равен None, ничего не исполняется);
* name обозначет имя потока (по умолчанию имя генерируется автоматически);
* args - кортеж аргументов для исполняемого объекта;
* kwargs - словарь именованных аргументов для исполняемого объекта;
* daemon равное True обозначет служебный поток (служебные потоки завершаются принудительно при завершении процесса); по умолчанию False.
* В Python выполнение программы заканчивается, когда все неслужебные потоки завершены.

**Понятие синхронизации потоков**

**Синхронизация потоков** является неким понятием из информатики, которое определяет различные механизмы гарантии того, что за раз не более одной одновременно исполняемого потока может обработать и исполнить некую определённую часть кода; такая часть кода именуется **критическим разделом.**

В некоторой конкретной программе, когда какой- то поток осуществляет доступ/ исполняет соответствующий критический раздел данной программы, все прочие потоки обязаны дожидаться, пока этот поток не завершит исполнение. Наиболее типичной целью синхронизации потоков является избежание потенциальной противоречивости данных при организации доступа множеством потоков разделяемых ими ресурсов; позволяя только одному потоку исполнять данный критический раздел вашей программы за раз чтобы гарантировать что в многопоточном приложении не произойдёт никаких конфликтов с данными.

**Класс threading.Lock**

Один из наиболее общих путей применения синхронизации потоков состоит в реализации некоторого механизма блокировки. В нашем модуле threading, класс threading.Lock предоставляет некий образец и интуитивно понятного подхода по созданию блокировок и работе с ними. Его основное использование включает в себя такие методы:

* threading.Lock(): Этот метод инициализирует и возвращает некий объект блокировки.
* acquire(blocking): При вызове данного метода все имеющиеся потоки будут запущены синхронно (то есть, только один поток может исполнять данный критический раздел в определённый момент времени):
* Необязательный аргумент blocking позволяет нам определять должен ли данный текущий поток выполнять ожидание чтобы получить данную блокировку.
* Когда blocking = 0, данный текущий поток не дожидается данной блокировки и просто возвращает 0, если такая блокировка не может быть получена этим потоком или 1 в противоположном случае.
* Когда blocking = 1, данный текущий поток выполняет блокирование и дожидается пока эта блокировка не освободится и получает её после этого.
* release(): При вызове данного метода его блокирование высвобождается.

Широко применяемым как в не параллельном, так и в совместном программировании понятием является применение очередей. Некая **очередь** является абстрактной структурой данных, которая является коллекцией различных элементов для сопровождения некоего установленного порядка; эти элементы могут быть иными объектами в какой- то программе.

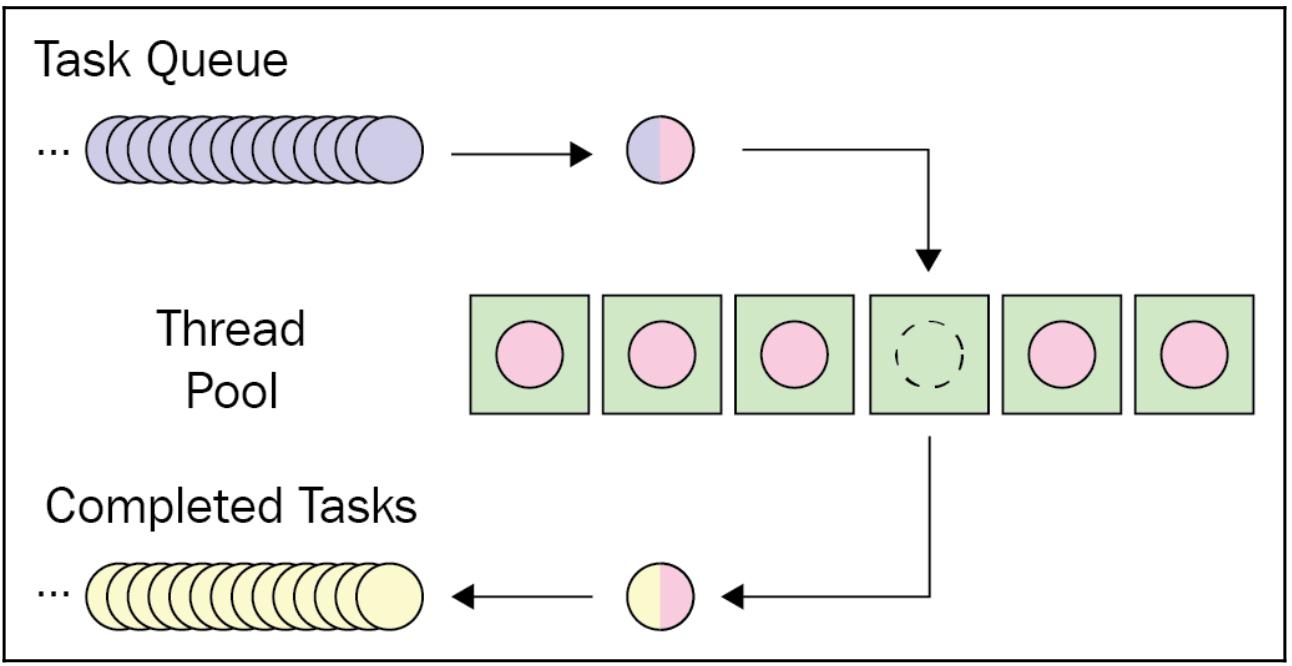
**Модуль queue**

Модуль queue в Python предоставляет некую простую реализацию такой структуры данных очереди. Каждая очередь из имеющегося класса queue.Queue может содержать определённое количество элементов и может иметь следующие методы:

* get(): Данный метод возвращает следующий элемент вызывая объект queue и удаляя его из этого объекта queue
* put(): Этот метод добавляет новый элемент в данный вызываемый объект queue
* qsize(): Этот метод возвращает общее число текущих элементов в данном вызываемом объекте queue (то есть его размер).
* empty(): Такой метод возвращает некое Булево значение, указывающее является ли вызываемый объект queue пустым.
* full(): Данный метод возвращает некое Булево значение, указывающее является ли вызываемый объект queue заполненным.

У нас есть возможность спроектировать некую структуру, в которой наш пул потоков не будет сохранять какую бы то ни было информацию, относящуюся к задачам, которые ему следует исполнять, а вместо этого сами задачи хранятся в некоторой очереди (иными словами, создают очередь задач), и соответствующие элементы из этой очереди будут питать персональных участников имеющегося пула потоков. По мере того как выбранная задача завершается неким участником нашего пула потоков, если присутствующая очередь задач всё ещё содержит подлежащие обработке элементы, тогда следующий элемент из этой очереди будет отправлен в тот поток, который только что освободился.

Приводимая ниже схема иллюстрирует такую настройку:



**Мультипроцессность**

Одно приложение может работать сразу в нескольких процессах. В каждом из этих процессов будет работать один или несколько потоков. Многопроцессорность в Python обеспечивается модулем **multiprocessing.** Он схож по функциональности с модулем **threading**, в частности имеет классы (**Lock**, **Event**, **Condition**, **Semaphore**и т.д.). Плюсом многопроцессорной программы является то, что каждый процесс работает в рамках своего GIL. Поэтому процессы могут работать с ЦПУ параллельно (на нескольких ядрах). А недостатком процессов является их тяжеловесность по сравнению с потоками.

Для создания процессов используют класс **Process** модуля **multiprocessing**. При запуске процессов нужно обязательно использовать конструкцию **if name == «main»**. Создание процесса ничем не отличается от создания потока. В параметр target мы передаём запускаемую функцию. А в параметр args — передаём массив параметров функции. Так как мы должны передать массив, а у нас всего 1 параметр, то после него ставим запятую. Точно также как поток, процесс можно запускать в режиме демона, для этого имеется параметр **daemon=True**.

Класс **Queue** - это очередь на основе FIFO (первый пришёл, первый ушёл). В очередь можно что-то поместить (**put()**) или что-то вытащить (**get()**).

При помещении в очередь объект блокируется, пока не станет доступен следующий слот. То есть в один слот два разных процесса одновременно не смогут поместить два разных объекта.

Метод **get()** не просто возвращает элемент из очереди, он его от туда удаляет, то есть освобождает слот.

**Класс Manager** — позволяет сделать **namespace**, в котором будут находится общие объекты. Различные процессы могут использовать этот **namespace** и получать доступ к общим объектам.

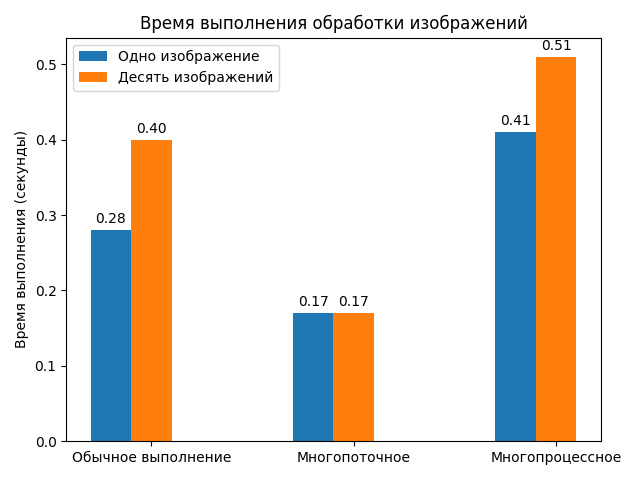
**Кдассы Listen** и **Client** — позволяют реализовать сетевое взаимодействие между процессами. Процессы могут, с помощью методов **send()** и **recv()**, отправлять и получать данные по сети.

**Практическое сравнение различных подходов**

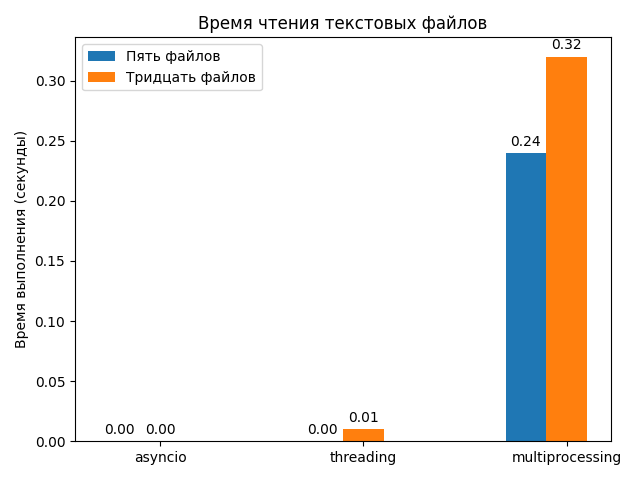
Как мы говорили в начале, выбор между процессами, потоками или применением асинхронного подхода зависит в первую очередь от нагрузки: **CPU bound** — нагрузка на процессор, когда он активно работает, например, при выполнении математических расчётов, **и I/O bound** — процессор ожидает операции ввода-вывода(запросы к базам данных или API каких-нибудь сервисов, то есть к внешним ресурсам).

В своей работе я постаралась применить несколько задач к сравнению подходов, используя в качестве инструмента сравнения производительности время, за которое тот или иной подход справлялся с поставленной задачей.

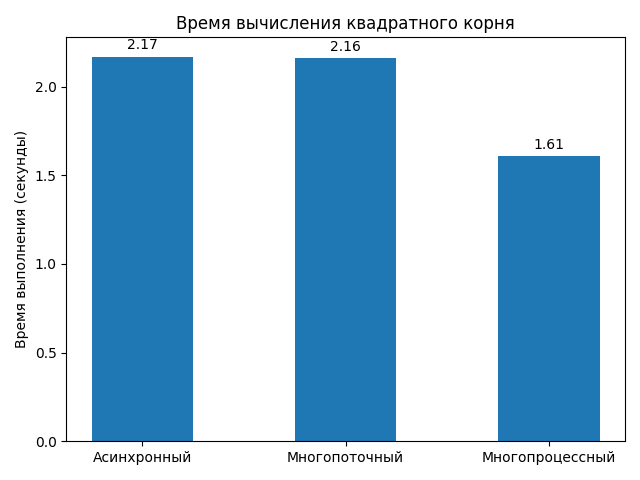
Первой задачей было редактирование одного и нескольких(десяти) изображений при помощи библиотеки Pillow. У изображений изменяли цветовую модель, размер и накладывали фильтр. При обычном выполнении кода, многопоточном и многопроцессном для одного изображения время выполнения обычным способом составило: 0.28 секунды, время выполнения многопоточным способом: 0.17 секунд, время выполнения многопроцессным способом: 0.41 секунды. Для десяти изображений время базового выполнения: 0.40 секунды, время выполнения многопоточным способом: 0.17 секунд, время выполнения многопроцессным способом: 0.51 секунды. Здесь потоки оказались более эффективны.



Второй задачей было чтение текстовых файлов, для сравнения пяти и тридцати файлов и вывод первых 50 символов. При использовании асинхронности, многопоточности и мультипроцессности для чтения пяти файлов время выполнения asyncio: 0.00 секунд, время выполнения threading: 0.00 секунд, время выполнения multiprocessing: 0.24 секунд. При чтении тридцати файлов время выполнения asyncio: 0.00 секунд, время выполнения threading: 0.01 секунды, время выполнения multiprocessing: 0.32 секунды. Асинхронное выполнение оказалось самым быстрым.



Третьей задачей шло вычисление квадратного корня числа. Для наглядности увеличили число задач и итераций в цикле. Время выполнения кода асинхронным способом составило: 2.17 секунд, многопоточным способом: 2.16 секунды, многопроцессным способом: 1.61 секунды.



Объяснением подобных результатов может служить то, что  процессы лучше справляются с интенсивными вычислительными задачами, такими как вычисление квадратного корня или любые операции, требующие значительных ресурсов CPU. Это связано с тем, что процессы используют отдельные ядра процессора, обходя ограничения GIL в Python.

**При редактировании изображений** потоки могли оказаться более эффективными, потому что операции включали в себя повторяющиеся действия (например, изменение размера, наложение фильтров). Python-потоки хорошо работают с задачами, где значительная часть нагрузки связана с вводом-выводом или менее ресурсоемкими вычислениями.

Асинхронный подход показал преимущество при операции ввода-вывода (например, чтение файлов), так как такие задачи обычно "ждут", пока операционная система завершит запрос. Асинхронность позволяет выполнять другие задачи в это время, что и приводит к ускорению.

**Заключение**

Каждый используемый нами подход эффективен для решения конкретных задач, имеет свои плюсы и минусы.

Асинхронность-работает в одном процессе и в одном потоке, экономно использует память, подходит для I/O bound операций, работает конкурентно, но у нее существует сложность отладки, а CPU bound операции блокируют все задачи. Используется там, где код много раз ожидает. Например, в нашем случае чтение файлов.

Потоки - работают параллельно, используют немного памяти, используют общую память, но одновременный доступ к памяти может приводить к конфликтам. В нашей задаче потоки показали себя эффективными в выполнении повторяющихся действий в операциях по редактированию изображения.

Процессы работают параллельно, используют все ресурсы ядра процессора, их все можно загрузить, изолированная память, независимые системные процессы, подходят для CPU bound операций, но если необходимо использовать общую память, то необходимо синхронизировать, так как нет общих переменных. Требуют больших ресурсов, так как запускают отдельный интерпретатор.

**Используем там, где обрабатываемые данные не зависят от других процессов и данных, в нашем случае при математических вычислениях.**

**Список литературы**

<https://habr.com/ru/companies/simbirsoft/articles/701020/>

<https://superfastpython.com/asyncio-vs-threading/>

<https://bvm84.livejournal.com/42630.html>

<https://habr.com/ru/articles/773376/>

<https://pythonru.com/osnovy/python-asyncio>

<https://sky.pro/wiki/python/raznitsa-multiprocessing-multithreading-asyncio-v-python/>

<https://cs.mipt.ru/advanced_python/lessons/lab11.html>

<https://habr.com/ru/companies/simbirsoft/articles/701020/>

<http://onreader.mdl.ru/MasteringConcurrencyInPython/content/Ch03.html>

<https://habr.com/ru/companies/sberbank/articles/829098/>

<https://javarush.com/quests/lectures/ru.javarush.python.core.lecture.level14.lecture02>

<https://sky.pro/wiki/python/primery-asinhronnogo-koda-v-python/>

<https://gitverse.ru/blog/articles/development/262-asinhronnyj-python-dlya-nachinayushih-ot-teorii-k-praktike>

https://habr.com/ru/companies/rosbank/articles/736698/