Дипломная работа по теме: Реализация проекта Python для сравнения моделей асинхронного программирования, демонстрирующих зависимость   
центрального процессорного устройства от нагрузки и влияние команд ввода/вывода.

Автор: Наседкин Дмитрий Игоревич

Оглавление дипломной работы:

Contents  
  
1. Введение. 2

Обоснование выбора темы. 2

Определение цели и задач исследования. 3

2. Основные понятия и определения. 3

3. Методы и подходы к разработке. 5

Выбор модели. 5

Архитектура приложения. 5

4. Обзор модулей asyncio, threading и multiprocessing в Python. 5

Asyncio. 5

Multiprocessing. 6

Threading. 6

5. Проектирование приложения. 6

Планирование и анализ требований. 6

Основные требования. 6

Технические требования. 7

6. Разработка в соответствии с созданной документацией. 7

Планирование разработки. 7

Разработка. 8

7. Анализ и интерпретация результатов. 16

Сравнение моделей. 17

Интерпретация результатов. 17

Рекомендации по выбору модели в зависимости от проекта. 17

8. Заключение. 18

Обзор выполненной работы. 18

Дальнейшие планы. 18

1.Введение

Обоснование выбора темы:

1. В нашем мире никто не любит ждать, поэтому нельзя писать код синхронно! Чтобы не отставать от современных тенденций, нужно освоить асинхронное программирование. В современном мире компьютерных технологий асинхронное программирование было реализовано и постоянно совершенствовалось. В 2015 году асинхронность стандартизовали в версии python 3.5, в которой появилась библиотека асинхронности -  asyncio,   
а также библиотека aiohttp, реализующая асинхронный доступ к веб-серверу, что позволило значительно ускорить и оптимизировать выполнение программ.

2. В настоящее время на рынке труда существует дефицит специалистов   
в области ИТ. Наравне с этим существует растущая потребность   
в специалистах, обладающих навыками применения асинхронного программирования. Компании стремятся использовать наиболее эффективные модели для улучшения качества своих продуктов и услуг.

3. Практическая значимость: Работа в области разработки веб-приложений   
и не только позволит применить полученные знания на практике, а также даст опыт использования асинхронного программирования. Это позволит создавать интересные и полезные для пользователей возможности.  
  
4. Личный интерес и потенциальные перспективы: Использование различных инструментов и создание продуктов сочетающих в себе разные технологии (например, веб-приложения и асинхронное программирование, хотя оно применяется практически повсеместно) позволяет развить и отработать опыт создания интересных и разносторонних проектов. Кроме того, такие навыки востребованы на рынке труда, что открывает перспективы для карьерного роста и развития.

Таким образом, выбор темы "Реализация проекта Python для сравнения моделей асинхронного программирования, демонстрирующих зависимость   
центрального процессорного устройства и влияние команд ввода/вывода   
при нагрузке" обусловлен ее актуальностью, потребностями рынка, практической значимостью и личным интересом, что делает эту тему подходящей для проведения дипломной работы.

**Определение цели и задач исследования:**

Цель исследования: Демонстрация моделей асинхронного программирования, отражающих зависимость центрального процессорного устройства и влияние команд ввода/вывода при нагрузке.

Задачи исследования:

1. Обзор различных подходов асинхронного программирования   
при проектировании приложений и необходимость использования в них асинхронных методов: Провести анализ существующих методов, выбрать наиболее подходящие под конкретные задачи.

2. Разработка критериев сравнения: Определить количественные   
и качественные показатели при использовании конкретного метода асинхронного программирования.

3. Создание проекта Python для проведение тестирования: реализовать проекта Python, на нем можно будет оценить эффективность различных моделей, и провести их тестирование.

4. Написание дипломной работы: Составить дипломную работу, включающую в себя введение, обзор литературы, методологию и результаты исследования, анализ результатов, выводы и рекомендации.  
  
Цели и задачи исследования направлены на получение практических результатов, которые позволят сформировать классификатор задач, рекомендуемых решением конкретным методом асинхронного программирования.

**2. Основные понятия и определения**

**Обзор основных понятий в области асинхронного программирования:**

1**. Фреймворк (англ. Framework)**: Программная платформа, которая предоставляет готовые компоненты и инструменты для разработки приложений. В контексте веб-приложений часто используются Django, FastAPI или Flask.

2. **Сопрограмма** **(англ. Coroutine)**: Программный модуль, особым образом организованный для обеспечения взаимодействия с другими модулями   
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.

3. **Веб-приложение (англ. Web Application)**: Программное приложение, которое работает на веб-сервере и доступно через браузер.

4. **Футура (англ. Future):** Конструкция языка программирования, которая формирует стратегию вычисления, применяемую для параллельных вычислений. С ее помощью описывается объект, к которому можно обратиться за результатом, вычисление которого может быть не завершено на данный момент.

5. **Асинхронность** **(англ. Asynchrony):** Неблокирующая архитектура, поэтому выполнение одной задачи не зависит от другой. Задачи могут выполняться одновременно.

6**. Синхронизация (англ. Synchronization):** Блокирующая архитектура, поэтому выполнение каждой операции зависит от завершения предыдущей. Каждая задача требует ответа перед переходом к следующей итерации.

7.  **Время отклика (англ. Response Time)**: Время, необходимое модели   
для обработки входных данных и выдачи результата. Время отклика является важной метрикой, особенно в приложениях, где скорость критична.

8. **Процесс (англ. Process)**: Экземпляр программы, который запущен   
и выполняется операционной системой.

9. **Поток (англ. Thread):** Минимальная единица выполнения работы   
в операционной системе.

10. **Аллокация:** Система распределения дефицитных ресурсов с тем, чтобы их получение было наиболее эффективным.

11. **IO-bound операция:** операция, в которой скорость выполнения ограничена скоростью ввода/вывода (например, запись на диск или запрос на сервер)

12. **CPU-bound операция:** операция, в которой скорость выполнения ограничена характеристиками CPU и оперативной памятью.

Данные термины помогают понять ключевые аспекты проверки эффективности работы асинхронной программы.

**3. Методы и подходы к разработке**

**Выбор модели**

Параллельное программирование — сложный, но очень полезный навык для программиста. Оно позволяет эффективно использовать мощности современных компьютеров с несколькими ядрами и процессорами. Это особенно важно при решении сложных задач, например, в инженерных расчетах, обработке мультимедийных данных, обучении нейросетей   
и многом другом.

Модель №1: модуль asyncio позволяет запускать функции программы асинхронно, не соблюдая последовательность.

Модель №2: модуль Threading демонстрирует мнимый параллелизм, создавая в основном процессе несколько рабочих потоков под управлением главного потока.

Модель №3: модуль Multiprocessing позволяет использовать истинный параллелизм, то есть создавать процессы, которые выполняются полностью независимо друг от друга.

Данные модели, принципы их работы реализованы в нашем проекте.   
  
**Архитектура приложения**

Для понятия принципа работы различных моделей асинхронного программирования будем использовать приложение PyСharm. В проекте будет 2 модуля с примером задач, решаемых соответствующим типом асинхронного программирования.

**4. Обзор модулей asyncio, threading и multiprocessing в Python**

Рассмотрим библиотеки asyncio, multiprocessing и модуль threading.

**Asyncio**  
  
Библиотека asyncio впервые появилась в версии Python 3.4 как еще один способ справляться с высокими конкурентными нагрузками, не прибегая к нескольким потокам или процессам. При правильном использовании эта библиотека может значительно повысить производительность и  уменьшить потребление ресурсов в  приложениях, выполняющих много операций ввода-вывода, поскольку позволяет запускать сразу много таких долго работающих задач.

**Multiprocessing**  
  
Библиотека multiprocessing будет запускать родительский процесс, который   
в свою очередь будет запускать дочерние процессы. В каждом дочернем процессе работает отдельный интерпретатор Python со своей GIL. В  предположении, что код выполняется на машине с  несколькими процессорными ядрами, это означает, что можно эффективно распараллелить счетные задачи. И даже если процессов больше, чем ядер, механизм вытесняющей многозадачности, встроенный в операционную систему, позволит выполнять задачи конкурентно. Такая конфигурация является конкурентной и параллельной.

**Threading**  
  
Модуль threading используется для создания потоков и управления ими. Он предлагает класс Thread, экземпляр которого принимает функцию, подлежащую выполнению в  отдельном потоке. Интерпретатор Python однопоточный в  том смысле, что в  каждый момент времени может выполняться только один участок байт-кода, даже если в  процессе работает несколько потоков. Глобальная блокировка интерпретатора не позволяет выполнять несколько потоков одновременно.

**5. Проектирование приложения**

**Планирование и анализ требований**

Определение структуры приложения: Разработка схемы архитектуры, включающей библиотеки asyncio, multiprocessing и модуль threading.  
  
Разработка прототипа

Реализация основного функционала: Создание базовой версии приложения   
с минимально необходимым функционалом для тестирования   
и демонстрации.

Разработка ключевого функционала

Импорт библиотек asyncio, multiprocessing и модуля threading. Подключение и настройка выбранных модулей для демонстрации их работы.

**Основные требования**

Генератор пакетов (задач): Пользователь должен иметь возможность выбирать количество пакетов, интервалы их генерации, продолжительность генерации пакетов.

Алгоритм обработки задач (пакетов): Приложение будет решать задачу обработки пакетов (задач) в соответствии с особенностями модуля   
или библиотеки.

Отображение результатов: Приложение должно отображать результаты обработки задач (пакетов).

**Технические требования**

Проект Python с использованием библиотек asyncio, multiprocessing и модуля threading.  
  
**6. Разработка в соответствии с созданной документацией**  
  
**Планирование разработки**

Разработка была разделена на несколько основных этапов:

1. создание модуля №1 с библиотекой multiprocessing и с модулем threading;
2. создание модуля №2 с библиотекам asyncio, multiprocessing и модулем threading.

**Разработка**  
  
Модуль №1: “multithreading\_project.py”

Работа с JSON или CPU-bound task

Рассмотрим функцию «job»:

# Работа с JSON или CPU-bound task. Скорость выполнения зависит от мощности процессора.  
**def** job(parametr):  
 json.dumps(list(range(parametr)))

Функция «job» выполняет следующее:

1. генерирует список;
2. превращает список в JSON.

Примечание: Подобный код - отличный пример CPU bound задачи - задачи, скорость выполнения которой зависит от мощности процессора.

Предположим, что выполнить вышеуказанную задачу необходимо N раз.

Рассмотрим решение этих задач различными способами с использованием различных методов асинхронного программирования.

Первый метод: стандартный, последовательный (без применения асинхронности).

Последовательный алгоритм будет выглядеть так:

# Последовательный алгоритм (на графике sync) будет выполнять задачу N раз.   
**def** serial(parametr, count):  
 **for** \_ **in** range(count):  
 job(parametr)

Исходя из нашего опыта и знаний о методах параллельного выполнения кода программы, очевидно, что последовательное выполнение кода неэффективно.

Какие варианты ускорить программу можем применить?

Вариант №1 - создаем по одному **потоку** на каждый вызов функции и попробуем добиться параллелизма:

# Создадим по одному потоку на каждый вызов функции и попробуем добиться параллелизма  
**def** threads\_start(parametr, execution\_Count):  
 threads = [threading.Thread(target=job, args=( parametr,)) **for** \_ **in** range(execution\_Count)]  
  
 **for** thread **in** threads:  
 thread.start()  
   
 **for** thread **in** threads:  
 thread.join()

Вариант №2 - вместо потоков создаем **процессы**:

# Создадим по одному процессу на каждый вызов функции и попробуем добиться параллелизма  
**def** processes\_start(parametr, execution\_Count):  
 processes = [multiprocessing.Process(target=job, args=( parametr,)) **for** \_ **in** range(execution\_Count)]  
   
 **for** process **in** processes:  
 process.start()  
   
 **for** process **in** processes:  
 process.join()

Для сбора данных, результаты вычислений функций, с последующим построением графика зависимостей создадим:

#список для задач, которые будут обрабатываться различными способами

x = [ ]

#общий список всех результатов обработки всеми способами

y = [ ]

#список с временами решения задач синхронным способом (sync)

serial\_result = [ ]

#список с временами решения задач многопоточным способом (multithreading)

threads\_start\_result = [ ]

#список с временами решения задач многопроцессным способом (multiprocessing)

processes\_start\_result = [ ]

Код тестирующий производительность всех реализаций:

**if** \_\_name\_\_ == '\_\_main\_\_':  
 # Отключаем автоматическую сборку мусора для качественного анализа производительности рассматриваемых методов   
 gc.disable()

# Устанавливаем количество аргументов  
 jsonSize = 1000000

# Подставляем значения в кортеж при 10 итерациях и получаем список из 10 кортежей  
 box = [(jsonSize, i) **for** i **in** range(1, 11)]

# Список функций, применяемых для анализа различных подходов параллельного программирования  
 options = [serial, threads\_start, processes\_start]

# Запускаем 10 итераций кортежа для создания потоков от 1 до 10 и 1000000 аргументов  
 **for** i, q **in** enumerate(box):  
 parametr, execution\_Count = q  
 print(f"Parallelism: {execution\_Count}, JSON Size: { parametr}")

x.append(execution\_Count)

# Функция принимает итерируемый объект и начальный индекс = 0  
 **for** j, option **in** enumerate(options):   
 start = time.perf\_counter()

# Передача значений в функции, реализующие параллелизм  
 option(parametr, execution\_Count)  
 end = time.perf\_counter()  
 time\_out = round(end – start, 2)  
 print(f"Способ работы: {option.\_\_name\_\_}, время выполнения: {time\_out} сек.")  
 # Наполняем списки результатами работы для каждого способа соответственно

if option.\_\_name\_\_ == ‘serial’:

serial\_result.append(time\_out)

if option.\_\_name\_\_ == ‘threads\_start’:

threads\_start\_result.append(time\_out)

if option.\_\_name\_\_ == ‘processes\_start’:

processes\_start\_result.append(time\_out)

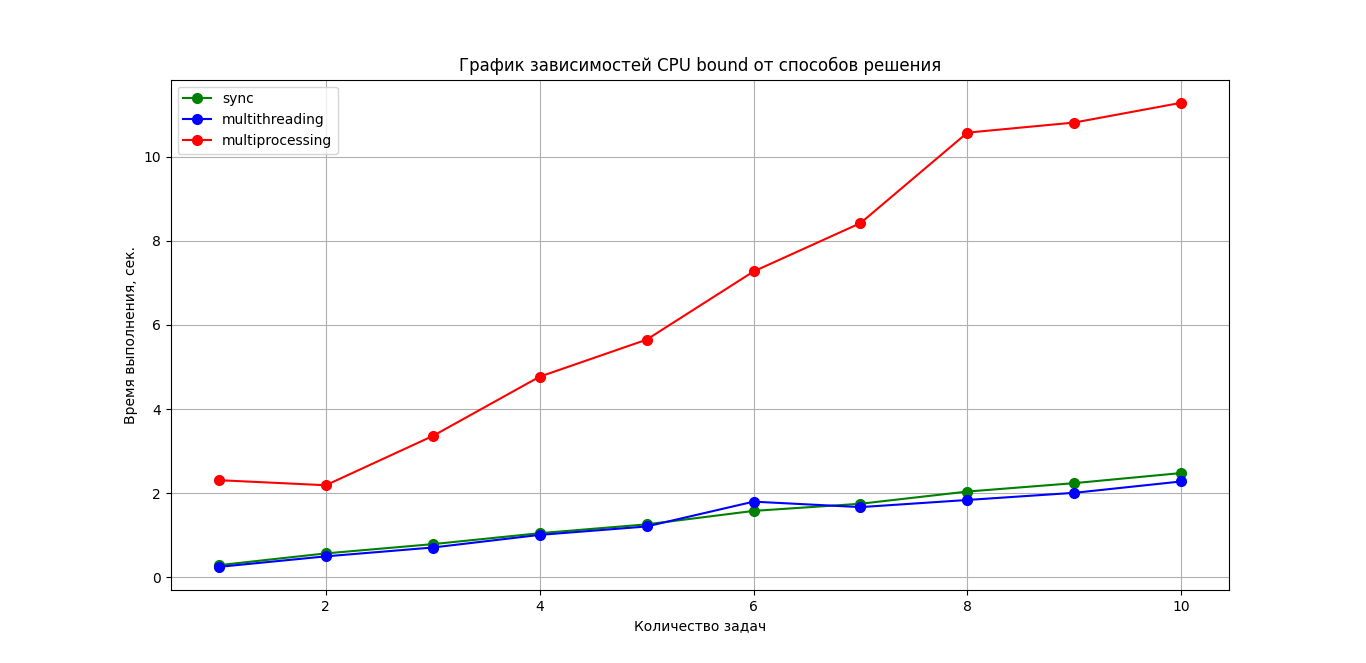
# Выведем результаты расчетов всех способов

print(serial\_result)

print(threads\_start\_result)

print(processes\_start\_result)

Результаты: выведем на график зависимостей



Результаты выполнения теста. Ось X - количество задач. Ось Y - время выполнения в секундах

Краткие выводы на основе результата теста:

* Использование потоков для параллельного исполнения CPU bound задач не дает явного преимущества, так как результаты сопоставимы с кодом не использующим потоки и работающим последовательно. Все дело в GIL - механизме синхронизации потоков, который не позволяет программе задействовать всю мощность процессора и выполнять потоки параллельно.
* Использование процессов заметно замедляет выполнение с увеличением создаваемых дочерние процессов, так как мы начинаем увеличивать расход ресурса процессора по максимуму и как следствие видим уменьшение времени исполнения на графике. На процессы действие GIL не распространяется.
* Максимальный параллелизм, которого можно добиться c помощью процессов, равен количеству физических ядер CPU. На графике можно заметить, что время исполнения теста multiprocessing растет приблизительно лесенкой с шагом 2 - это количество физических ядер CPU на моем ноутбуке.

Модуль №2: “in\_output.py”

Работа с внешним API или IO-bound task

На ряду с CPU bound задачами, существуют еще задачи IO bound.   
В таких задачах производительность зависит не от процессора,  
 а от подсистемы ввода-вывода, а также устройств с которыми мы взаимодействуем, например файловая система или сеть.

В примере будем эмулировать IO нагрузку через sleep (вместо sleep   
в реальности может быть, например запрос по HTTP к API сервиса).

**def** job(parametr):  
 time.sleep(parametr)

#Создадим N вызовов функции последовательным способом выполнения задачи

**def** serial(parametr, execution\_Count):  
 **for** \_ **in** range(execution\_Count):  
 job(parametr)

**async** **def** async\_io\_tasks(parametr, execution\_Count):  
 tasks = [asyncio.create\_task(async\_job(parametr)) **for** \_ **in** range(execution\_Count)]  
 **await** asyncio.gather(\*tasks)

#Конкурентная реализация на основе asyncio

**def** async\_io\_start(parametr, execution\_Count):  
 asyncio.run(async\_io\_tasks(parametr, execution\_Count))  
  
**async** **def** async\_job(parametr):  
 **await** asyncio.sleep(parametr)

#Конкурентная реализация на потоках:  
**def** threads\_start(parametr, execution\_Count):  
 **with** concurrent.futures.ThreadPoolExecutor(max\_workers=execution\_Count) **as** executor:  
 executor.map(job, [parametr] \* execution\_Count)

#Конкурентная реализация на процессах  
**def** processes\_start(parametr, execution\_Count):  
 **with** concurrent.futures.ProcessPoolExecutor() **as** executor:  
 executor.map(job, [parametr] \* execution\_Count)

#список для задач, которые будут обрабатываться различными способами

x = [ ]

#общий список всех результатов обработки всеми способами

y = [ ]

#список с временами решения задач асинхронным способом (async)

async\_io\_start\_result = [ ]

#список с временами решения задач многопоточным способом (multithreading)

threads\_start\_result = [ ]

#список с временами решения задач многопроцессным способом (multiprocessing)

processes\_start\_result = [ ]

#Сравним разницу между подходами, протестируя от 1 до 1001 конкурентных задач с шагом 200

**if** \_\_name\_\_ == '\_\_main\_\_':  
 box = [ (0.1, i) **for** i **in** range(1, 10002, 200)]  
 options = [processes\_start, threads\_start, async\_io\_start]  
   
 **for** i, q **in** enumerate(box):  
 parametr, execution\_Count = q  
 print(f"Кол-во задач: {execution\_Count}")

x.append(execution\_Count)  
   
 # Функция принимает итерируемый объект и начальный индекс = 0  
 **for** j, option **in** enumerate(options):   
 start = time.perf\_counter()

# Передача значений в функции, реализующие параллелизм  
 option(parametr, execution\_Count)  
 end = time.perf\_counter()  
 time\_out = round(end – start, 2)  
 print(f"Способ работы: {option.\_\_name\_\_}, время выполнения: {time\_out} сек.")  
 # Наполняем списки результатами работы для каждого способа соответственно

if option.\_\_name\_\_ == ‘async\_io\_start’:

async\_io\_start\_result.append(time\_out)

if option.\_\_name\_\_ == ‘threads\_start’:

threads\_start\_result.append(time\_out)

if option.\_\_name\_\_ == ‘processes\_start’:

processes\_start\_result.append(time\_out)

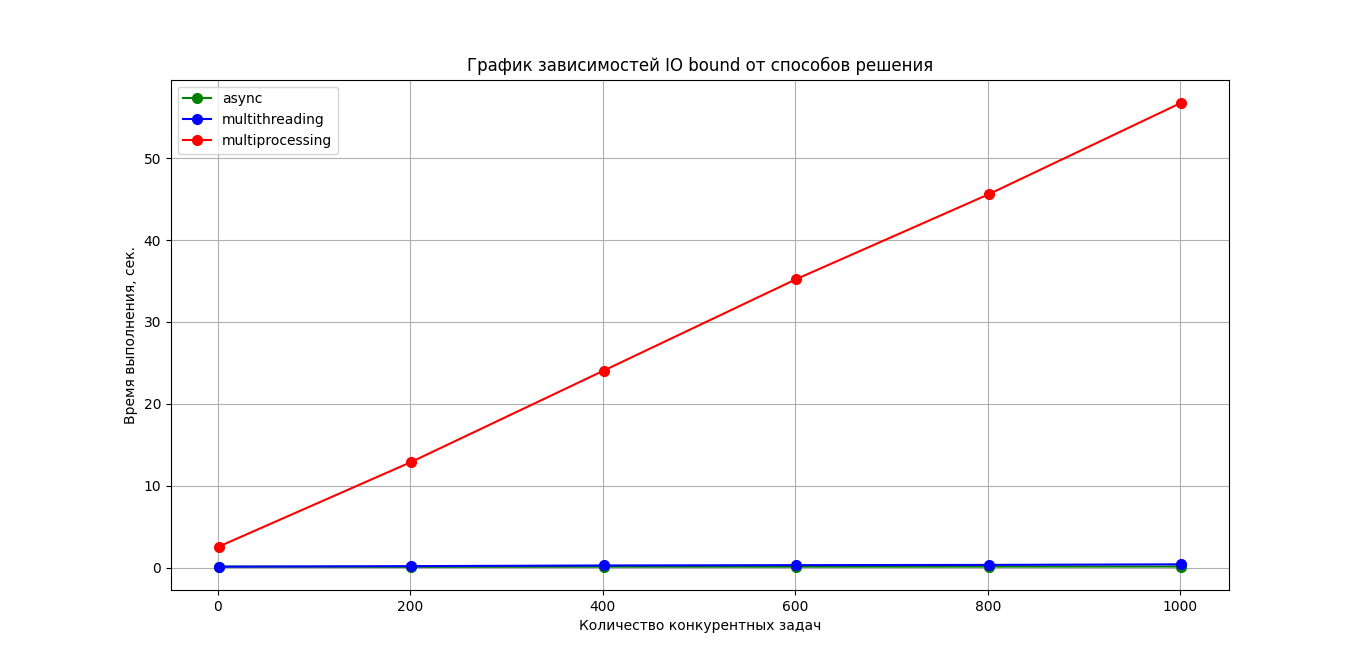
# Выведем результаты расчетов всех способов

print(async\_io\_start\_result)

print(threads\_start\_result)

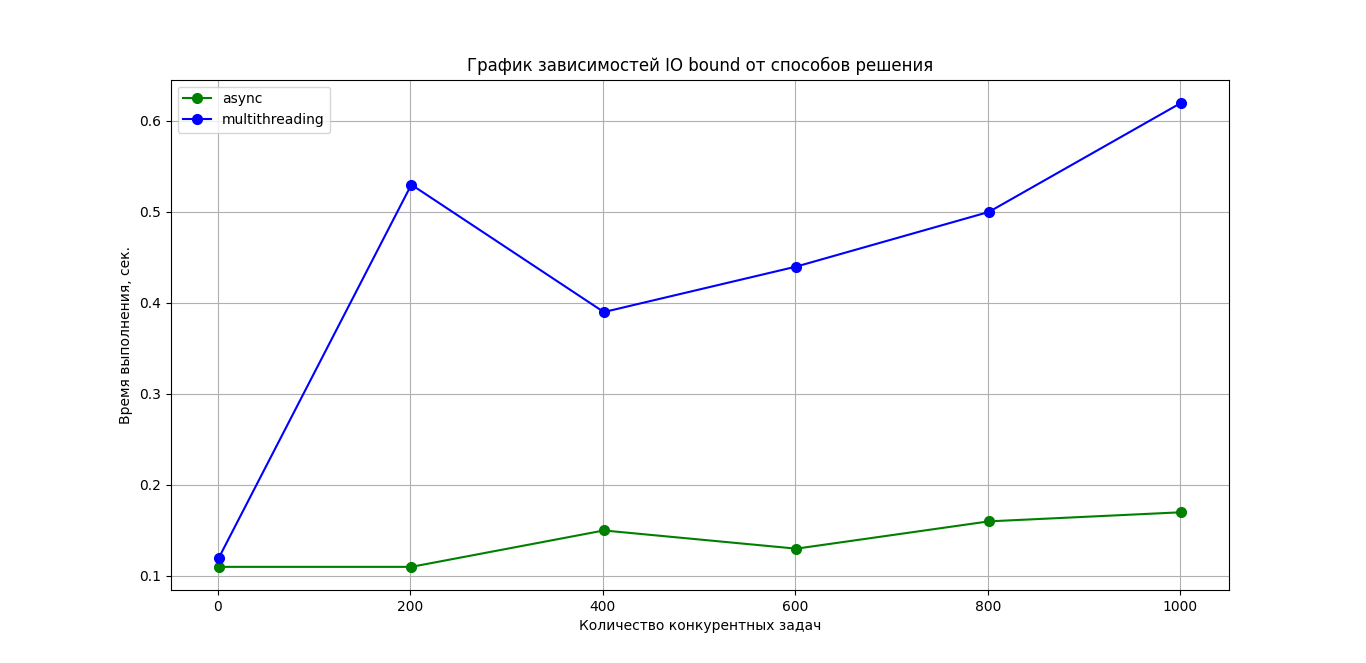
print(processes\_start\_result)

Результаты:



Результаты выполнения теста. Ось X - количество конкурентных задач (процессов и потоков). Ось Y - время выполнения в секундах.

Увеличим масштаб и сравним потоки и корутины:



Результаты выполнения теста. Ось X - количество конкурентных задач (процессов и потоков). Ось Y - время выполнения в секундах.

Краткие выводы на основе результата теста:

* Процессы в чистом виде непригодны для подобного класса задач, слишком большая затрата ресурсов на создание процесса и его “убийство”, взаимодействовать с ОС так часто – “дорогое” удовольствие.
* Потоки справились намного лучше процессов, и GIL в данной задаче не стал помехой, так как он запрещает "исполняться" нескольким потокам одновременно, а в случае IO нагрузки потоки дольше находятся в ожидании, чем в работе. Но замедление работы   
  по сравнению с корутинами присутствует, так как мы взаимодействуем с ОС и аллоцируем память.
* Корутины - абсолютный лидер, за счет того что ими управляет рантайм языка, а не ОС, к тому же с точки зрения памяти корутине нужно ее меньше, чем потоку.

**Бэкенд-разработка:**  
  
Реализован проект с использованием приложения PyCharm.  
  
Настроены параметры для обработки задач:

В модуле №1 (multithreading\_project.py) следующими способами: синхронным, многопоточным и многопроцессным.

В модуле №2 (in\_output.py) следующими способами: асинхронным, многопоточным и многопроцессным.

В обоих модулях есть возможность изменения входных данных (количество задач и аргументов) для отражения на графиках зависимости способов решения задач от их количества.

На основе полученных результатов можно сделать оптимальный выбор   
в пользу метода с наилучшими показателями, отраженными на графике.

Важно помнить, что результаты работы многопроцессного способа зависят от технических характеристик компьютера, на которой запускают данную программу. К таким характеристикам относят: количество процессоров, ядер, их тактовая частота, объем кэш-памяти и оперативная памяти.

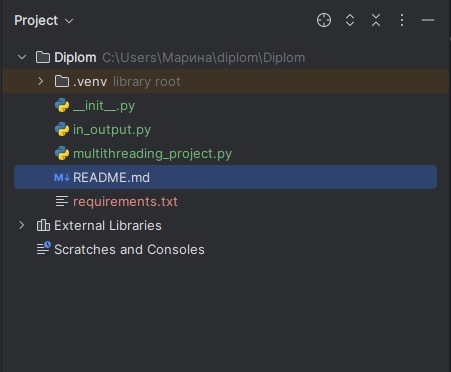
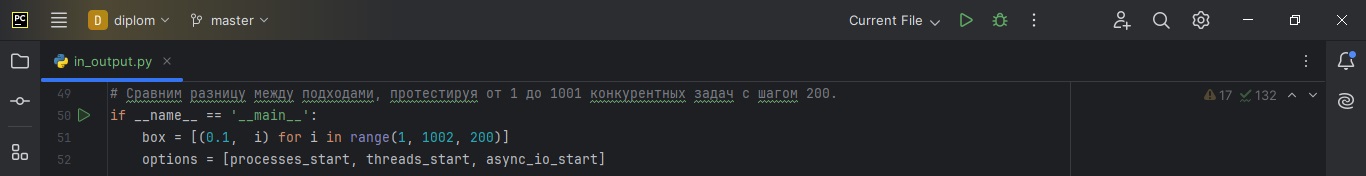


Рис. 1 Файловая структура проекта

Рис. 2 Ввод данных в файле in\_output.py

В строке 51 в методе range вводим (начало диапазона количества задач; конец диапазона количества задач; шаг изменения количества задач)

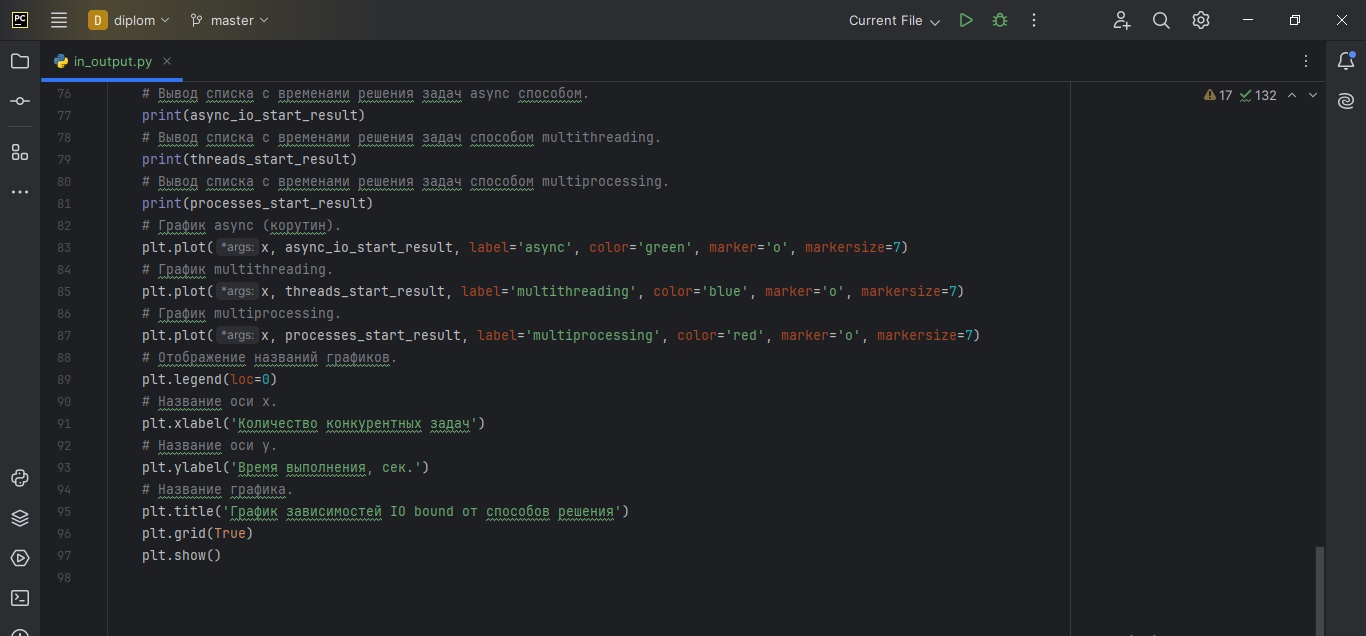


Рис. 3 Вывод данных на графике в файле in\_output.py

В строке 83 реализован метод отображения на графике результата работы метода async. Если результат не нужно отображать, то просто ее закомментируем, поставив в начале строки #.

В строке 85 реализован метод отображения на графике результата работы метода multithreading. Если результат не нужно отображать, то просто ее закомментируем, поставив в начале строки #.

В строке 87 реализован метод отображения на графике результата работы метода multiprocessing. Если результат не нужно отображать, то просто ее закомментируем, поставив в начале строки #.

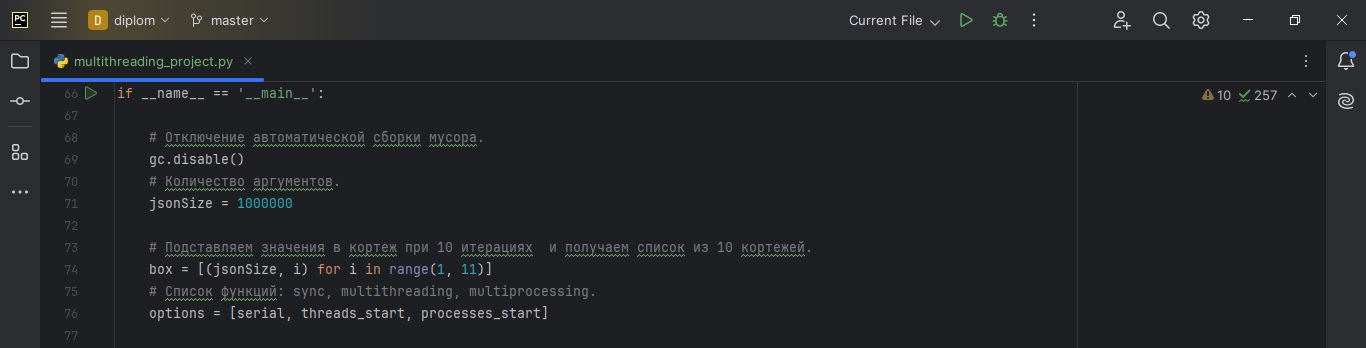


Рис. 4 Ввод данных в файле multithreading\_project.py

В строке 71 в переменную jsonSize вводим (количество аргументов, над которыми проводится вычисления в наших методах). Чем больше значение, тем большую нагрузку даем для вычислений каждому методу, соответственно увеличивается время решения

В строке 74 в методе range вводим количество задач, которых будет создано и решено различными способами (начальное количество создаваемы задач, конечное количество создаваемы задач).

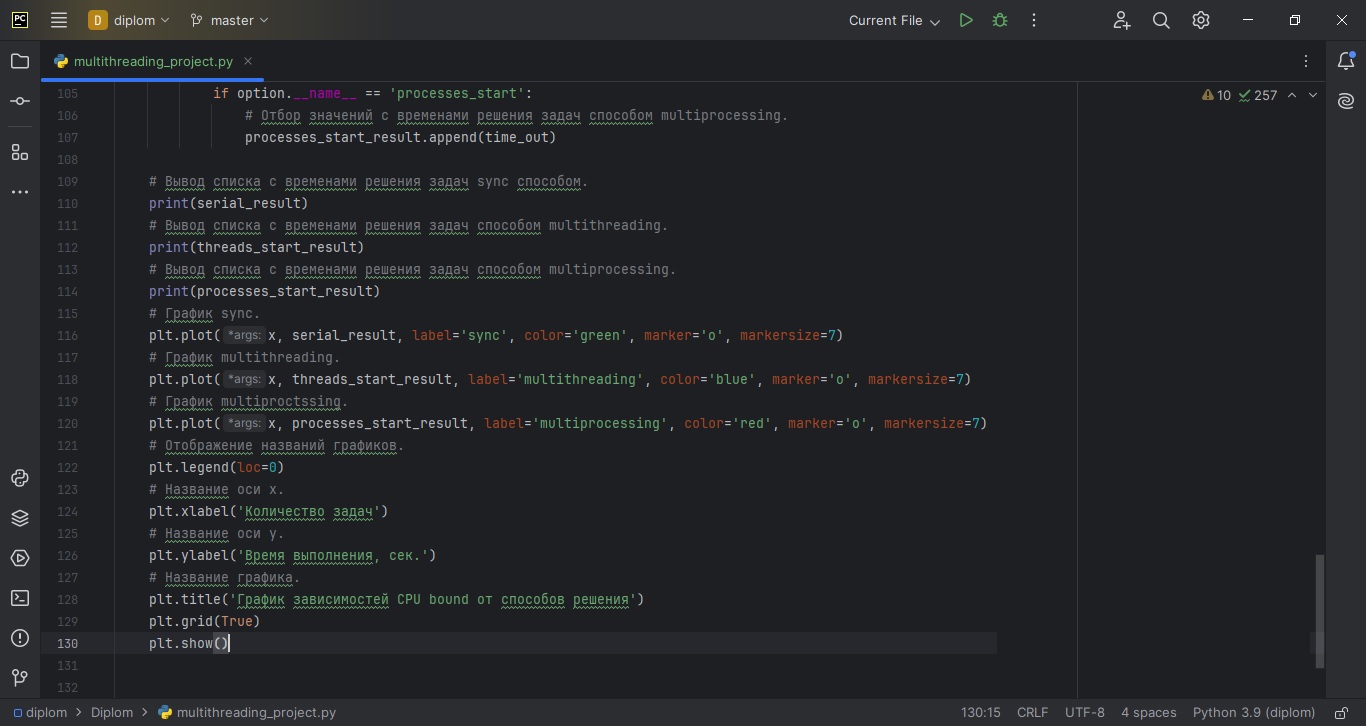


Рис. 5 Вывод данных на графике в файле multithreading\_project.py

В строке 116 реализован метод отображения на графике результата работы метода sync. Если результат не нужно отображать, то просто ее закомментируем, поставив в начале строки #.

В строке 118 реализован метод отображения на графике результата работы метода multithreading. Если результат не нужно отображать, то просто ее закомментируем, поставив в начале строки #.

В строке 120 реализован метод отображения на графике результата работы метода multiprocessing. Если результат не нужно отображать, то просто ее закомментируем, поставив в начале строки #.

**7. Анализ и интерпретация результатов**  
  
Вывод по ограничению количества процессов/потоков/корутин

1. Сколько можно создавать процессов и потоков в программе?

2. Чем мы платим за их создание?

Ответ: мы платим оперативной памятью нашего устройства (ПЭВМ, ноутбук, гаджет и т.п.). Для каждого процесса и потока ОС резервирует определенное количество памяти, а так как память конечна, то без остановки создавая процесс или поток мы рано или поздно столкнемся с OOM (Out of memory error). Чтобы избегать подобных ошибок в production системах ограничивается количество активных процессов и потоков с помощью популярных механизмов ThreadPool и ProcessPool.

Итоги

В дипломной работе рассмотрели практическое применение потоков, процессов и корутин. На примерах разобрались, в каких задач применим тот или иной инструмент. Для того чтобы упростить код были выбраны простые синтетические примеры, но при этом не противоречащие production коду который мы встречаем ежедневно в работе.

**Сравнение моделей**

Были протестированы различные подходы к применению потоков, процессов и корутин.

Основным рассматриваемым критерием являлась скорость выполнения конкретно способа параллельного вычисления, зависящего от количества решаемых задач. На основе полученных данных можем оценить производительность каждого способа моделей.

**Интерпретация результатов**

При решении CPU bound задач в реализации на моем компьютере, наибольшей эффективности достиг многопоточный метод, хотя не на много лучше последовательного метода, из-за ограничения GIL. Хотя имея мощный компьютер, можно достичь значительного ускорения многопроцессного метода.

При решении IO bound задач корутины (метод async), за счет того что ими управляет рантайм языка, а не ОС, и памяти нужно меньше, чем потоку является наилучшим вариантом. Чуть медленнее отработал многопоточный метод из-за взаимодействия с ОС, по сравнению с корутинами. Многопроцессность при решении таких задач не рекомендуется   
из-за необоснованности затрат ресурсов, без выигрыша в скорости по сравнению с многопоточностью и корутинами.

**Рекомендации по выбору метода параллельной работы в зависимости от проекта**  
  
Для задач, требующих выполнения сложных математические вычислений, операций шифрования и дешифрования, сортировки больших списков, рекомендуется использовать многопроцессность, только если у вас мощный компьютер. В противном случае применяйте обычный последовательный метод, так как многопоточность в скорости не даст большого изменения.

Для задач, где **требуется** обработка ввода с клавиатуры или вывод информации на экран, чтение и запись данных в базу данных, выполнение SQL-запросов, транзакций, **сетевых операций и взаимодействие с API л**учше всего подойдет использование асинхронности – корутинов.

**8. Заключение**

**Обзор выполненной работы**

Разработка проекта Python для сравнения моделей асинхронного программирования, демонстрирующих зависимость центрального процессорного устройства от нагрузки и влияние команд ввода/вывода   
успешно завершены в соответствии с изначально созданной документацией. Приложение включает функционал обработки генератора задач   
с использованием различных методов параллельной работы и графическим анализом результатов их работы.

Реализованный проект соответствует требованиям и демонстрирует зависимость метода параллельной от работы от типа решаемых задач   
и технических характеристик компьютера.

Методы параллельной работы действительно ускоряют процесс выполнения программы. Главное на этапе проектирования учесть все необходимые требования к вычислительным средствам, где разрабатываемое приложение будет использоваться и какие именно задачи будет решать.

**Дальнейшие планы**

В развитии данной работы сделать дополнительный модуль, который   
на основе результата выдавал бы подсказку о применяемости конкретного метода параллельной работы.

Создать эмулятор работы различных методов параллелизма при решении задач CPU bound, с заданием технических характеристик вычислительному устройству, где планируется применяться разрабатываемое приложение.