«Urban University»

Дипломный проект

на тему

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

Исполнитель: Кандинский К.А.

Томск

2024

Содержание

1. Введение

2. Основные понятия и определения

3. Методы и подходы к разработке

4. Проектирование приложения

5. Реализация приложения

6. Сбор и анализ данных.

6.1 Задача с вводом-выводом:

6.2 Задача вычисление числа пи методом Лейбница.

6.3 Задача с высокой параллельностью

7. Обобщаем результаты: преимущества и недостатки каждого подхода.

8. Рекомендации по применению и ограничения.

9. Заключение

10. Источники

**1. Введение**

Обоснование выбора темы

Асинхронное программирование стало важным компонентом в разработке современных приложений, особенно в условиях растущих требований к скорости обработки данных и масштабируемости сервисов. Среди разработчиков на Python существуют разные подходы к организации многозадачности, такие как asyncio, threading и multiprocessing. Каждый из них обладает своими преимуществами и ограничениями в зависимости от специфики задач.

Рассмотрение различных методов асинхронного программирования позволяет не только углубить понимание их внутренней работы, но и выбрать наиболее эффективный способ решения сложных вычислительных задач. Это особенно актуально в контексте постоянно растущих объёмов данных и требований к времени отклика систем.

Таким образом, тема является актуальной и релевантной для понимания того, как эффективно разрабатывать приложения, которые могут быстро и эффективно обрабатывать большие объёмы информации, обеспечивая тем самым конкурентоспособность и стабильность современных информационных систем.

Определение цели и задач исследования

Целью данного исследования является изучение и сравнение различных подходов к асинхронному программированию в Python, таких как asyncio, threading и multiprocessing. Мы стремимся определить, какой из подходов наиболее подходит для различных типов задач, будь то операции ввода-вывода, параллельное выполнение независимых задач или интенсивные вычисления.

Для достижения этой цели были поставлены следующие задачи:

- Изучить основы и ключевые аспекты каждого подхода к многозадачности.

- Реализовать примеры асинхронных задач для каждого из исследуемых методов.

- Проанализировать производительность и временные характеристики выполнения задач.

- Оценить уместность каждого подхода для определённых видов вычислительных проблем.

Основные вопросы исследования

- Какие ключевые отличия существуют между asyncio, threading и multiprocessing?

- В каких случаях использование одного подхода предпочтительнее другого?

- Как каждый подход влияет на производительность и потребление ресурсов?

Исследование нацелено на предоставление чётких и практичных рекомендаций для разработчиков при выборе подхода к решению конкретных задач в области асинхронного программирования.

**2. Основные понятия и определения**

Асинхронное программирование

Асинхронное программирование — это парадигма разработки программного обеспечения, позволяющая выполнять операции без необходимости блокировки основного потока исполнения программы. Это важная концепция, позволяющая повысить отзывчивость и производительность приложений, особенно тех, которые работают с большим количеством операций ввода-вывода.

Основная идея заключается в том, чтобы позволить приложению продолжать выполнение других операций вместо того, чтобы ждать завершения медленных задач, таких как чтение и запись данных, сетевые запросы или любые другие операции, которые могут занять неопределённое время.

Основные подходы в Python

1. Asyncio

Asyncio — это библиотека в языке Python, поддерживающая структуру написания асинхронного кода. Она предоставляет средства для работы с корутинами и циклами событий, которые позволяют выполнять задачи в асинхронном режиме, эффективно управляя задачами ввода-вывода без создания новых потоков или процессов.

Asyncio поддерживает концепцию генераций задач с использованием ключевых слов async и await, что делает код более читаемым и удобным для поддержки. Этот подход идеально подходит для задач, связанных с посещением большого количества разрозненных источников данных, например, получения данных с веб-сервисов или работы с асинхронными сетевыми сокетами.

2. Threading

Threading — это традиционный способ управления многопоточностью в Python, который позволяет запустить несколько потоков внутри одного процесса. Потоки разделяют одно и то же пространство памяти, что позволяет эффективно передавать данные между ними, но это же может быть источником сложности при управлении состояниями в многопоточной среде.

Этот подход чаще всего используется в приложениях, где требуется выполнение множества малых задач одновременно, которые могут взаимодействовать друг с другом, например, в пользовательских интерфейсах или небольших серверах.

3. Multiprocessing

Multiprocessing предлагает ещё один подход к параллельному выполнению задач в Python, но здесь каждое отдельное задание выполняется в своём процессе с индивидуальным пространством памяти. Это позволяет избежать проблем с конкурентным доступом к данным, характерных для многопоточности, но делает необходимым использование инструментов межпроцессного взаимодействия.

Процессы управляются с помощью multiprocessing библиотеки Python, и это делает данный подход более подходящим для задач, требующих интенсивных вычислений, поскольку процессы могут быть распределены по разным ядрам процессора.

Основные различия

- Управление ресурсами: Asyncio более ресурсоэффективен для задач, ориентированных на ввод-вывод, в то время как threading и multiprocessing могут быть более полезными для параллельного вычисления.

- Безопасность потоков: Multiprocessing обеспечивает независимость процессов, избегая блокировок или состояний гонки, тогда как threading требует осторожного управления доступом к общим ресурсам.

**3. Методы и подходы к разработке**

Использование asyncio

Asyncio — это основной инструмент для реализации асинхронного программирования в Python. Он предоставляет мощные абстракции для создания и управления асинхронными задачами. Основные компоненты asyncio включают корутины, задачи и цикл событий.

- Корутина: Это функции, объявленные с использованием ключевого слова async. Они могут приостанавливать своё выполнение с помощью await, позволяя другим корутинам выполняться.

- Задача (Task): Asyncio позволяет запускать корутины как задачи, которые управляются циклом событий. Это позволяет планировать выполнение нескольких операций ввода-вывода параллельно.

Использование threading

Threading позволяет запускать параллельные потоки в одном процессе. Python предоставляет модуль threading, который делает выполнение многопоточности доступным.

- Запуск потока: Каждый поток можно запускать с указанием функции и её аргументов, которые будут выполняться в этом новом потоке.

- Управление потоками: Несмотря на простоту, каждый поток требует аккуратности в управлении общими ресурсами, чтобы избежать состояния гонки.

Использование multiprocessing

Multiprocessing позволяет выполнять задачи в отдельных процессах, что позволяет Python программы использовать все ресурсы процессора.

- Создание процессов: Библиотека multiprocessing предоставляет простой API для создания новых процессов.

- Межпроцессное взаимодействие: Отдельные процессы имеют своё адресное пространство, что делает необходимым использование механизмов для обмена данными, таких как очереди или каналы.

Сравнение подходов

Каждый из описанных подходов имеет свои преимущества и ограничения, которые делают его более подходящим для определённых задач:

- Вычислительные задачи: Подход multiprocessing выигрывает, позволяя использовать все ресурсы процессора.

- Операции ввода-вывода: Asyncio позволяет эффективно обрабатывать множество соединений, избегая накладных расходов потоков или процессов.

- Параллельные задачи: Threading может быть проще в настройке для быстрого параллелизма, когда задачи не конкурируют за общий ресурс.

**4. Проектирование приложения**

Планирование и анализ требований

Перед началом разработки важно определить основные цели тестирования и разработать план экспериментов:

1. Тип задачи: Мы выберем стандартные задачи, которые включает в себя как операции ввода-вывода, так и вычислительные компоненты, для более полного сравнения.

2. Критерии производительности: Время выполнения и количество использованной памяти.

3. Среда тестирования: однородная среда для каждого подхода, чтобы результаты были сопоставимы.

Основные требования

- Реализовать три задачи с использованием всех трех подходов

Первая задача: загрузка данных с веб-сервера.

Вторая задача: вычисление числа пи методом Лейбница.

Третья задача: с высокой параллельностью с использованием функции time.sleep(0.1).

Технические требования

- Язык программирования: Python

- Библиотеки: asyncio, threading, multiprocessing

- Мониторинг производительности: psutil для измерения использованной памяти, time для измерения затраченного времени.

5. Реализация приложения

1 Задача с вводом-выводом Task1

2. Задача с большим количеством вычислений Task2

3. Задача с высокой параллельностью Task3

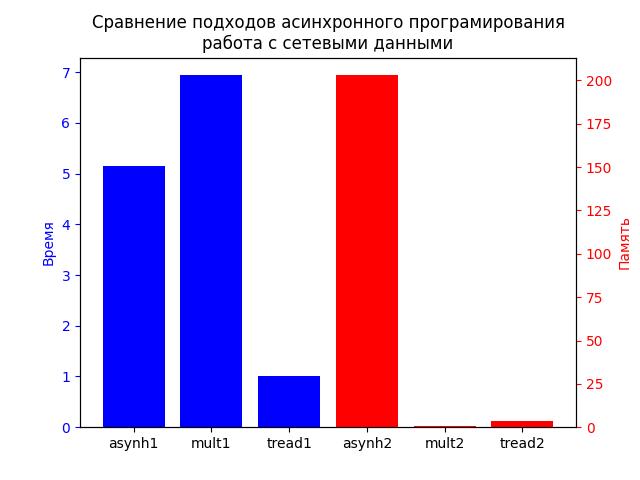
4. Лог файл для каждой задачи и для каждого подхода, в котором храниться время и количество использованной памяти.

5. Визуализация результатов.

**6. Сбор и анализ данных.**

В каждой задаче все три подхода были поочередно выполнены 10 раз.

**6.1 Задача с вводом-выводом:**



Asinhio показал среднее время работы и максимальное количество используемой памяти.

- Среднее время выполнения: Asyncio оптимально справляется с задачами ввода-вывода, такими как чтение данных с веб-сайтов, через использование цикла событий и корутин. Это позволяет выполнять множество сетевых операций параллельно, с минимальной блокировкой. Среднее время выполнения может указывать на эффективное управление множественными запросами.

- Максимальное использование памяти:

- Если все 50 запросов выполняются параллельно, а их результат как-то временно хранится, то общее использование памяти может возрасти из-за одновременной работы множества корутин.

- Корневые библиотеки, используемые для работы с сетью, могут также иметь память под буферы для обработки множества открытых соединений.

Multiprocessing показал максимальное время время выполнения и минимальное количество используемой памяти.

Максимальное время выполнения и минимальное использование памяти:

- Максимальное время выполнения: Multiprocessing менее эффективен для ввода-вывода, потому что открытие и завершение процесса требует значительных временных затрат. Кроме того, межпроцессное взаимодействие при передаче данных между процессами может быть узким местом.

- Минимальное использование памяти:

- Каждый процесс имеет своё независимое пространство памяти. Если процессы не хранят много данных, а ресурсы размежевания используются эффективно, реальный показатель памяти может быть низким.

Treading – показал минимальное время выполнения и среднее использование памяти.

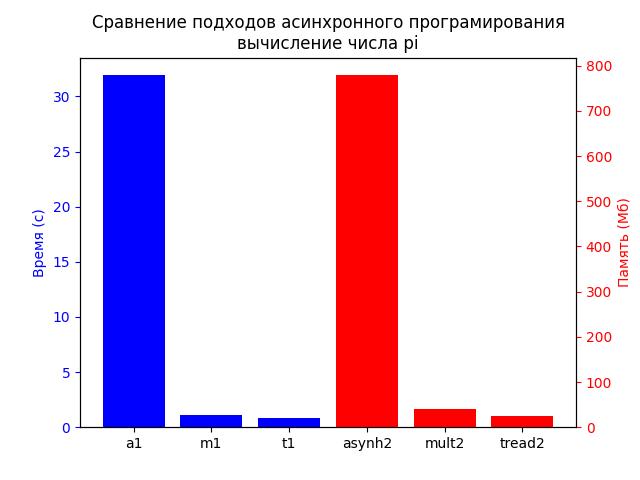
- Минимальное время выполнения: Threading может показывать хорошие результаты в задаче, где большая часть времени тратится на ожидание ввода-вывода, потому что потокам легче переключаться между собой, если одна задача ожидает завершения.

- Среднее использование памяти:

- Поскольку потоки совместно используют память, есть накладные расходы, но они ниже, чем у процесса. Это могут быть буферы или разделяемые структуры данных, которые могут объяснять средний уровень памяти.

Вывод: Для задачи, нацеленной на ввод-вывод, asyncio и threading могут показать лучшее время выполнения по сравнению с multiprocessing, так как они предлагают меньшие накладные расходы для управления конкурентными задачами, такими как одновременные соединения с сайтом.

**6.2 Задача вычисление числа пи методом Лейбница.**



Asinhio показал максимальное время и использование памяти.

Асимметрия для вычислительных задач: Asyncio лучше всего проявляет себя в задачах ввода-вывода, где период ожидания может быть заполнен другими задачами. Для интенсивных вычислений он не получает значительных преимуществ, поскольку asyncio не использует множества процессоров и не оптимизирован для вычислительных задач.

- Максимальное использование памяти: Когда вычисления выполняются в рамках одного процесса с мерами asyncio, может наблюдаться большое использование памяти из-за управленческих затрат, либо временных объектов, создаваемых при суммировании и обработке чисел.

Multiprocessing показал среднее время и среднее использование памяти.

- Выполнение процессов: Multiprocessing позволяет распределить расчёты между несколькими процессами, которые могут использовать разные ядра процессора. Это помогает избегать блокировок, связанных с GIL (Global Interpreter Lock), что делает многопроцессность более подходящей для уровня высокой производительности в вычислениях.

- Среднее использование памяти: Каждый процесс выполняется в собственном пространстве памяти, но накладные расходы управления процессами и межпроцессного взаимодействия могут приводить к дополнительной нагрузке на память.

Treading показал минимальное время и минимально использование памяти.

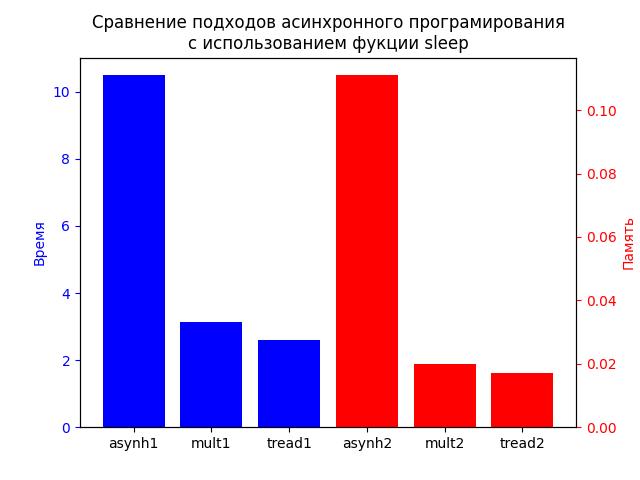
- Преодоление GIL: Для простой задачи, не слишком интенсивной в плане ввода-вывода или сложной логики, Threading может по своему управлению потоками оказываться быстрее, особенно если каждую часть работы можно быстро завершить и объединить.

- Минимальное использование памяти: Потокам доступны общие структуры данных, но при использовании многопоточности GIL остаётся неподвижным в Python, что может быть ограничением для высокоинтенсивных задач, хотя в этой задаче они могут оказаться более управляемыми.

Вывод: -Для задачи с интенсивными вычислениями Multiprocessing подходит лучше, чем asyncio, поскольку он может использовать все доступные ядра процессора, предоставляя явные выгоды для тяжёлых вычислений.

- Threading может проявлять себя хорошо для задач, где GIL не становится узким местом, но при более сложных и интенсивных расчетах его эффективность может снизиться.

**6.3 Задача с высокой параллельностью**



Функция time.sleep(0.1) практически не требует обработки. Она просто приостанавливает выполнение программы на указанный интервал времени, уходя в режим ожидания. Таким образом, задача является скорее симуляцией высокой параллельности ожидания, чем ресурсо-интенсивной операцией.

Asinhio показал самое большое время и использование памяти.

- Задержка на обработку контекста: Asyncio не подходит для таких задач, так как он стремится к управлению операциями ввода-вывода, пропуская задачи, ожидающие завершения. time.sleep() не является асинхронной функцией и не использует await, что приводит к неэффективности внутри цикла событий asyncio, блокируя его и вызывая накладные расходы.

- Максимальное использование памяти: Если задачи выполняются как часть большого числа корутин, это может потребовать большего объёма памяти на управление циклами и состояниями для каждой задачи, особенно если они тщательно обрабатываются в ожидании.

Multiprocessing показал среднее время и использование памяти.

- Процессы и время на управление: Каждый процесс multiprocessing работает в своём адресном пространстве. time.sleep(0.1) действительно эффективно просто удерживает процесс в ожидании. Однако из-за необходимости создания процесса и ресурсов управления всеми процессами, может происходить увеличение накладных расходов.

- Среднее использование памяти: Но каждая инструкция sleep() делает все процессы неактивными на 0.1 секунды, и потенциального высвобождения основного процессорного времени для других задач, использование памяти могло бы быть чуть более оптимальным из-за разграниченного использования пространства памяти.

Treading показал минимальное время и использование памяти.

- Преимущества в параллельном ожидании: Threading максимально использует себя в задачах, где требуется ожидание завершения, например time.sleep(), поскольку приостановка основного потока позволяет другим потокам немедленно начать выполнение. Это даёт возможность минимизации блокировок и позволяет распределить задачи по потокам.

- Минимальное использование памяти: Поскольку потоки совместно используют память, тратимая память будет оптимизироваться, так как затраты на управление потоками и синхронизация общих ресурсов значительно снижены.

Вывод: - Asyncio не подходит для задач, в которых ожидается блокировка без ввода-вывода, из-за того, что цикл событий в таких обстоятельствах становится узким местом.

- Multiprocessing "перебор" небольших задач и ожиданий не может быть оправданным, из-за ресурсоемкости работы процессов.

- Threading фактически становится наиболее подходящим по времени и памяти, так как эффективно управляет состояниями ожидания.

В данном тесте видно что Multiprocessing и Threading выполняются параллельно в 4 потока, поэтому время в 4 раза меньше чем у Asinhio.

**7. Обобщаем результаты: преимущества и недостатки каждого подхода.**

Asyncio

Преимущества:

- Подходит для ввода-вывода: Asyncio наиболее эффективен для долгосрочных задач, связанных с вводом-выводом, где время ожидания велико, например, сетевые операции, взаимодействие с базами данных или файловые операции.

- Лёгкое управление конкурентностью: Корутинная модель позволяет управлять многими задачами без необходимости создания множества потоков или процессов.

Недостатки:

- Не подходит для вычислительных задач: Асинхронность не может обойти Python GIL, что делает её непрактичной для чисто вычислительных задач, нелегко реализуется через await.

- Сложность синтаксиса: Новый стиль программирования с async и await может быть менее интуитивным для новых разработчиков.

Возможности и простота:

- Простота ожидания: С самим понятием await простое ожидание длительных асинхронных операций позволяет избежать блокировок.

- Читаемость кода: Код при должной структуре может быть доволен понятным, но требует некоторого опыта с новой парадигмой.

Threading

Преимущества:

- Хорош для смешанных задач: Различные задачи, которые не требуют высокоинтенсивных вычислений, выигрывают от многопоточности, удачно справляясь с простой параллелизацией.

- Подходит для задач с ожиданием: Threading хорош для задач, которые тратят много времени в состоянии ожидания.

Недостатки:

- GIL ограничение: При высокоинтенсивных вычислениях одновременно работающие потоки на Python могут быть ограничены GIL.

- Сложность управления потоками: Особенно для потокобезопасности и синхронизации общего состояния данных, нужно выстраивать архитектуру кода.

Возможности и простота:

- Встроенные инструменты: Threading хорош для быстрого начала работы с параллелизмом и часто проще в настройке.

- Читаемость кода: Потоковый код может стать сложным из-за требуемой синхронизации доступа к общим ресурсам.

Multiprocessing

Преимущества:

- Очень подходит для вычислительных задач: Может использовать все ядра процессора, так как процессы не подвластны ограничению GIL.

- Изоляция процесса: Каждый процесс имеет своё отдельное пространство памяти, что упрощает управление ресурсами и снижает риск состояния гонки.

Недостатки:

- Большая нагрузка на создание и переключение между процессами: Процессы более тяжёлые по сравнению с потоками или корутинами, что ограничивает эффективность для задач ввода-вывода.

- Интер-процессное общение: Требует явных механизмов обмена данными между процессами, что может быть сложно в реализации.

Возможности и простота:

- Изолированность процесса: Проще обезопасить данные из-за отсутствия общего состояния данных.

- Сложная настройка: Добиться эффективного обмена данными может быть непросто, и может потребоваться значительное архитектурное планирование.

**8. Рекомендации по применению и ограничения.**

Asyncio

Рекомендации по применению:

1. Асинхронные операции ввода-вывода:

- Используйте asyncio, когда основная часть вашей программы связана с операциями ввода-вывода, такими как сетевые запросы, взаимодействие с базой данных или файловые операции.

2. Высокая конкурентность с минимальными затратами:

- Эффективен для выполнения большого количества мелких задач одновременно, особенно при обработке событий, таймеров и задержек.

3. Серверные приложения и микросервисы:

- Хорош для построения веб-сервисов и серверных приложений благодаря несинхронной обработке запросов (например, с использование FastAPI).

4. Ограничения:

- Не рекомендуется для задач, требующих тяжёлых вычислений, поскольку async задачи разделяют процесс, не обходя GIL.

Threading

Рекомендации по применению:

1. Лёгкие вычислительные задачи:

- Хорош для параллельных задач, которые больше связаны с состояниями и ожиданиями, чем с тяжёлыми вычислениями.

2. Интерфейсы пользователя:

- Часто используется для повышения отзывчивости UI, позволяя интерфейсу оставаться активным, пока длительные процессы выполняются на фоне.

3. Секционирование задач:

- Полезен, когда необходимо разделить задачу на несколько потоков для повышения скорости выполнения без значительных накладных расходов на создание новых объектов.

4. Ограничения:

- Могут быть проблемы управления, такие как состояние гонки и мьютексы, особенно при доступе к разделяемым ресурсам из нескольких потоков.

Multiprocessing

Рекомендации по применению:

1. Интенсивные вычисления:

- Подходит для задач, требующих больших вычислительных ресурсов, поскольку может обойти ограничение GIL, распределяя задачи по нескольким ядрам процессора.

2. Задачи с высокой независимостью:

- Хорошо подходит для задач, которые можно легко разделить на несколько независимых частей, так как процессы не разделяют память.

3. Долговременные вычисления:

- Применим для задач второстепенного плана и моторно значительных сервісов, где высокий уровень параллелизма улучшает производительность.

4. Ограничения:

- Высокие накладные расходы на создание и управление процессами, особенно для задач с короткими временными рамками.

- Сложности в межпроцессовом взаимодействии, если требуется обмен большими объёмами данных.

**9. Заключение**

Обобщение подходов

- Asyncio оптимален для ввода-вывода: элегантен для сетевых операций и где требуется много асинхронного работы.

- Threading удобен для задач с состоянием и ожиданиями: помогает быстро внедрить параллелизм.

- Multiprocessing отлично справляется с расчётами: позволяет выжать максимум производительности из нескольких ядер процессора.

Читаемость и сложность

- Asyncio может потребовать привыкания к новым принципам синтаксиса и парадигмам.

- Threading предоставляет хороший старт для начинающих в многозадачности.

- Multiprocessing даёт отличные результаты, но требует знания о безопасности межпроцессных взаимодействий и управления памятью.

Дополнительные подходы к асинхронному программированию в Python

1. Twisted:

- Описание: Это один из старейших фреймворков для асинхронного программирования в Python. Twisted предлагает широкий набор инструментов для создания сетевых приложений, поддерживает множество протоколов, таких как HTTP, SMTP, DNS и другие.

- Особенности: Использует цикл событий как и asyncio, но предоставляет более высокий уровень абстракций. Хорошо подходит для сложных сетевых приложений.

2. Tornado:

- Описание: Это асинхронный веб-фреймворк и библиотека сетевых приложений, которая может справляться с тысячами одновременных соединений, что делает его подходящим для WebSockets и реального времени.

- Особенности: Несмотря на то что Tornado поддерживает asyncio, он предлагает свой собственный цикл событий и может использоваться независимо.

3. Curio и Trio:

- Curio: Акцентируется на корутинах и использует собственный цикл событий для управления асинхронными задачами.

- Trio: Позиционируется как фреймворк для упрощённого и безопасного асинхронного программирования. Trio делает большой упор на улучшение управления задачами и исключением.

**10. Источники**

<https://habr.com/ru/articles/421625/>

<https://it4each.com/blog/threading-multiprocessing-i-asyncio-v-python/>

<https://simple-blog.vercel.app/2021/02/11/threading-asyncio-multiprocessing/>

ChatGPT-4o

Giga.chat