**Дипломная работа**

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

задача: Реализовать асинхронные задачи с использованием asyncio, threading и multiprocessing, сравнить их производительность и уместность для различных типов задач.

Автор: Иванов Анатолий Леонидович

Оглавление

[1. Введение 3](#_Toc184072607)

[1.1. Актуальность исследования 3](#_Toc184072608)

[1.2. Цель и задачи работы 3](#_Toc184072609)

[1.3. Основные понятия и определения 4](#_Toc184072610)

[2. Анализ подходов к асинхронному программированию 4](#_Toc184072611)

[2.1. История и эволюция асинхронных методов в Python 5](#_Toc184072612)

[2.2. Характеристики и ограничения современных подходов 5](#_Toc184072613)

[2.2.1. Модуль asyncio 5](#_Toc184072614)

[2.2.2. Модуль threading 6](#_Toc184072615)

[2.2.3. Модуль multiprocessing 6](#_Toc184072616)

[2.3. Сравнение подходов: теоретические аспекты 7](#_Toc184072617)

[3. Методология исследования 8](#_Toc184072618)

[3.1. Типы задач для сравнения (IO-bound, CPU-bound) 8](#_Toc184072619)

[3.2. Методы измерения производительности и анализа ресурсов 9](#_Toc184072620)

[3.3. Инструменты разработки и тестирования 9](#_Toc184072621)

[4. Реализация задач 10](#_Toc184072622)

[4.1. Описание тестовых сценариев 10](#_Toc184072623)

[4.1.1. Задачи ввода-вывода (IO-bound) 10](#_Toc184072624)

[4.1.2. Задачи вычислений (CPU-bound) 11](#_Toc184072625)

[4.2. Реализация сценариев с использованием подходов 11](#_Toc184072626)

[4.2.1. Реализация на asyncio 12](#_Toc184072627)

[4.2.2. Реализация на threading 12](#_Toc184072628)

[4.2.3. Реализация на multiprocessing 12](#_Toc184072629)

[5. Анализ и интерпретация результатов 13](#_Toc184072630)

[5.1. Сравнение производительности 13](#_Toc184072631)

[5.1.1. IO-bound задачи 13](#_Toc184072632)

[5.1.2. CPU-bound задачи 13](#_Toc184072633)

[5.2. Потребление системных ресурсов 14](#_Toc184072634)

[5.3. Применимость подходов в реальных проектах 14](#_Toc184072635)

[6. Выводы и рекомендации 15](#_Toc184072636)

[6.1. Основные выводы исследования 15](#_Toc184072637)

[6.2. Рекомендации по выбору подхода 16](#_Toc184072638)

[6.3. Перспективы дальнейшего изучения 16](#_Toc184072639)

[Заключение 17](#_Toc184072640)

[Список литературы 18](#_Toc184072641)

[Приложения 19](#_Toc184072642)

[1. Реализация IO-bound задачи 19](#_Toc184072643)

[1.1 Реализация на asyncio 19](#_Toc184072644)

[1.2 Реализация на threading 20](#_Toc184072645)

[1.3 Реализация на multiprocessing 21](#_Toc184072646)

[Сравнение подходов (резюме) 23](#_Toc184072647)

[2. Реализация CPU-bound задачи 23](#_Toc184072648)

[2.1 Реализация на asyncio 23](#_Toc184072649)

[2.2 Реализация на threading 24](#_Toc184072650)

[2.3 Реализация на multiprocessing 25](#_Toc184072651)

[Сравнение подходов (резюме) 26](#_Toc184072652)

1. Введение

Асинхронное программирование является важным инструментом в современном программировании, особенно в условиях роста требований к производительности программного обеспечения. С его помощью решаются задачи оптимизации времени выполнения приложений, повышения масштабируемости и эффективного использования ресурсов. Разработка программ с использованием асинхронных подходов становится критически важной для создания высоконагруженных веб-приложений, обработки данных в реальном времени и других систем, ориентированных на интенсивную работу с вводом-выводом или многозадачностью.

Python предоставляет разработчикам широкий выбор инструментов для реализации асинхронного программирования, среди которых особо выделяются asyncio, threading и multiprocessing. Каждый из этих подходов имеет свои сильные и слабые стороны, которые важно учитывать при выборе метода решения задач.

В данном исследовании проводится сравнительный анализ трех подходов, рассматриваются их возможности и ограничения в контексте различных типов задач.

Итоги анализа позволят сформулировать рекомендации по их применению, что делает проект ценным для практической деятельности разработчиков программного обеспечения.

1.1. Актуальность исследования

В современных программных системах необходимость выполнения нескольких задач одновременно встречается практически повсеместно:

* Обработка многочисленных запросов в веб-серверах.
* Параллельная обработка больших объемов данных.
* Выполнение сложных вычислений в условиях ограничений по времени.

Python, благодаря своей универсальности и широкой экосистеме библиотек, стал популярным языком для таких задач. Однако выбор подходящего инструмента — asyncio, threading или multiprocessing — зависит от особенностей задачи. Введение новых возможностей и оптимизаций в Python делает актуальным исследование их производительности и применимости.

Этот проект нацелен на удовлетворение потребности разработчиков в структурированных данных о производительности разных методов асинхронного программирования, что имеет практическое значение для повышения качества и эффективности программного обеспечения.

1.2. Цель и задачи работы

Цель исследования — провести сравнительный анализ подходов asyncio, threading и multiprocessing для реализации асинхронного программирования в Python, определить их производительность и оценить уместность использования для разных типов задач.

Для достижения цели были поставлены следующие задачи:

* Изучить теоретические основы асинхронного программирования и особенности реализации каждого из подходов.
* Разработать тестовые сценарии для задач, связанных с обработкой ввода-вывода (IO-bound) и вычислений (CPU-bound).
* Реализовать эти сценарии с использованием подходов asyncio, threading и multiprocessing.
* Провести измерение и анализ производительности каждого подхода.
* Сформулировать рекомендации по выбору подхода для различных типов задач.

1.3. Основные понятия и определения

В рамках данного исследования используются следующие ключевые понятия:

* Асинхронное программирование — метод программирования, позволяющий выполнять несколько задач одновременно без блокировки выполнения, что повышает производительность программ при работе с вводом-выводом и многозадачными операциями.
* IO-bound задачи — задачи, основное время выполнения которых связано с операциями ввода-вывода, например, чтение и запись данных в файловую систему, запросы к сетевым ресурсам.
* CPU-bound задачи — задачи, ограниченные производительностью центрального процессора, такие как математические вычисления, обработка данных и другие ресурсоемкие операции.
* Threading (многопоточность) — подход, при котором выполнение задач происходит в отдельных потоках одного процесса. Эффективен для IO-bound задач, но ограничен GIL (Global Interpreter Lock) в Python.
* Multiprocessing (многопроцессорность) — подход, при котором задачи выполняются в отдельных процессах, позволяя задействовать несколько процессорных ядер и обходить ограничения GIL.
* Asyncio — библиотека Python, реализующая асинхронное программирование на основе кооперативной многозадачности через использование событийного цикла и ключевых слов async и await.
* **Корутина** — это программная конструкция, которая позволяет приостанавливать выполнение функции и возобновлять её позже с того же места. Коррутины похожи на функции, но отличаются тем, что могут приостанавливать своё выполнение и возвращать управление вызывающей стороне, а затем продолжать работу.
* GIL — это глобальная блокировка интерпретатора Python, которая ограничивает выполнение только одного потока Python-кода за раз, даже на многоядерных процессорах. Эта блокировка необходима для обеспечения безопасности потоков при работе с объектами Python.

Данные определения составляют основу исследования и используются для описания, сравнения и анализа представленных подходов.

2. Анализ подходов к асинхронному программированию

Асинхронное программирование представляет собой концепцию, которая позволяет выполнять несколько операций одновременно или частично перекрывать их во времени. Это особенно важно для задач, требующих одновременной обработки множества операций ввода-вывода или вычислений. В Python асинхронное программирование поддерживается несколькими подходами, каждый из которых имеет свои сильные и слабые стороны. В данном разделе рассматриваются история появления этих методов, их ключевые характеристики и ограничения.

2.1. История и эволюция асинхронных методов в Python

Асинхронное программирование в Python прошло долгий путь развития.

Ранние версии Python (до 3.0):

В ранних версиях Python разработчики полагались на базовые потоки (threading) и процессы (multiprocessing) для выполнения задач параллельно. Эти подходы работали, но имели свои недостатки:

threading страдал от ограничений GIL (Global Interpreter Lock), что снижало эффективность на многопроцессорных системах.

multiprocessing позволял обходить GIL, но требовал больших затрат на создание процессов и межпроцессное взаимодействие.

Проблемы, связанные с GIL:

Он ограничивает производительность многопоточных программ для задач, требующих интенсивного использования процессора (CPU-bound задачи), так как потоки вынуждены ожидать освобождения блокировки.

Для IO-bound задач (например, сетевых операций), где потоки часто находятся в состоянии ожидания, влияние GIL минимально.

Из-за этих ограничений разработчики начали искать альтернативные способы реализации асинхронности, что привело к созданию asyncio.

Введение библиотеки asyncio (Python 3.3):

С выходом Python 3.3 была представлена библиотека asyncio, которая реализует асинхронное программирование на основе событийного цикла. Это позволило:

Использовать ключевые слова async и await для удобного написания асинхронного кода.

Эффективно решать задачи ввода-вывода (IO-bound) без необходимости создавать потоки или процессы.

Современные улучшения (Python 3.5 и выше):

Постепенное развитие синтаксиса и улучшение стандартной библиотеки сделали asyncio основным выбором для задач асинхронного программирования в Python. Также библиотека concurrent.futures добавила удобства для работы с потоками и процессами, предоставляя унифицированный интерфейс.

Таким образом, Python предоставляет гибкий инструментарий для реализации асинхронного программирования, который охватывает сценарии ввода-вывода, многозадачности и параллельных вычислений.

2.2. Характеристики и ограничения современных подходов

Каждый из методов асинхронного программирования в Python подходит для решения определенных типов задач. В данном разделе подробно рассматриваются модули asyncio, threading и multiprocessing.

2.2.1. Модуль asyncio

Характеристики:

Основан на концепции событийного цикла.

Позволяет выполнять задачи параллельно, переключаясь между ними на операциях ожидания (await).

Использует кооперативную многозадачность, где задачи добровольно уступают управление другим задачам.

Эффективен для IO-bound задач, таких как сетевые запросы, работа с файлами или базами данных.

Преимущества:

Легковесность: не создает потоков или процессов, что экономит системные ресурсы.

Удобный синтаксис с использованием ключевых слов async и await.

Хорошая масштабируемость для обработки тысяч или даже миллионов параллельных задач.

Ограничения:

Не подходит для CPU-bound задач, так как операции вычислений блокируют события цикла.

Требует адаптации традиционного кода, чтобы стать асинхронным.

2.2.2. Модуль threading

Характеристики:

Использует потоки (threads) для выполнения задач в рамках одного процесса.

Позволяет запускать несколько потоков, которые работают параллельно, но взаимодействуют с одной памятью.

Преимущества:

Простота реализации для параллельного выполнения кода.

Удобство для задач, которые часто ждут ввода-вывода (например, сетевых операций).

Ограничения:

Ограничен GIL, что мешает полноценному использованию нескольких процессоров для CPU-bound задач.

Потенциальные проблемы с синхронизацией потоков и состояние гонки (race condition).

2.2.3. Модуль multiprocessing

Характеристики:

Создает отдельные процессы, каждый из которых имеет свою память и выполняется независимо от других.

Позволяет эффективно использовать ресурсы многопроцессорных систем.

Преимущества:

Отсутствие GIL: подходит для CPU-bound задач, требующих высокой вычислительной мощности.

Возможность распараллеливания вычислений между несколькими процессами.

Ограничения:

Более высокая стоимость создания процессов по сравнению с потоками.

Сложности в обмене данными между процессами (требуются межпроцессные механизмы, такие как очереди или пайпы).

2.3. Сравнение подходов: теоретические аспекты

Для понимания сильных и слабых сторон подходов asyncio, threading и multiprocessing, важно проанализировать их теоретические характеристики. Эти подходы различаются в способах управления задачами, использовании ресурсов и применимости к разным типам нагрузок.

**Критерии сравнения**

**Природа многозадачности**

asyncio: реализует кооперативную многозадачность, где задачи добровольно передают управление друг другу через ключевое слово await. Задачи выполняются последовательно, но переключаются на операциях ожидания (например, сетевой ввод-вывод).

threading: использует конкурентную многозадачность, позволяя потокам работать независимо, но все они делят одно адресное пространство в рамках процесса. Управление потоками выполняется операционной системой.

multiprocessing: обеспечивает полную параллельность за счет создания отдельных процессов. Каждый процесс имеет собственное адресное пространство, что исключает конфликты между задачами, но усложняет обмен данными.

**Ограничения GIL**

asyncio: не использует потоки и не сталкивается с ограничениями GIL. Асинхронные задачи выполняются в одном потоке.

threading: выполнение потоков ограничено GIL, что делает его менее подходящим для CPU-bound задач.

multiprocessing: каждый процесс работает независимо, обходя GIL и эффективно используя ресурсы многоядерных процессоров.

**Использование памяти и ресурсов**

asyncio: наиболее легковесный вариант, так как не создает дополнительных потоков или процессов.

threading: потребляет больше ресурсов из-за контекста потоков и необходимости синхронизации.

multiprocessing: требует значительных ресурсов, так как каждый процесс запускает отдельный интерпретатор Python и выделяет память под свое окружение.

**Простота разработки**

asyncio: требует понимания асинхронного синтаксиса и преобразования существующего кода для использования async и await.

threading: легко интегрируется в традиционные программы, так как потоки работают параллельно с привычным синхронным кодом.

multiprocessing: сложнее в использовании из-за необходимости явного управления процессами и организации межпроцессного взаимодействия.

**Типы задач, для которых подход наиболее эффективен**

asyncio: подходит для IO-bound задач, где основное время работы программы связано с ожиданием ввода-вывода (сетевые запросы, взаимодействие с базами данных).

threading: также хорошо справляется с IO-bound задачами, но менее эффективен из-за большего потребления ресурсов.

multiprocessing: оптимален для CPU-bound задач, так как позволяет эффективно использовать многоядерные системы.

#### **Таблица сравнения**

| **Критерий** | **asyncio** | **threading** | **multiprocessing** |
| --- | --- | --- | --- |
| Тип многозадачности | Кооперативная | Конкурентная | Полная параллельность |
| Ограничение GIL | Нет | Есть | Нет |
| Потребление ресурсов | Низкое | Среднее | Высокое |
| Простота разработки | Средняя | Высокая | Низкая |
| Типы задач | IO-bound | IO-bound | CPU-bound |
| Масштабируемость | Высокая | Средняя | Высокая |

**Основные выводы**

Asyncio наиболее подходит для задач, связанных с вводом-выводом, особенно если требуется обработка большого количества параллельных соединений. Однако оно требует асинхронного подхода к разработке, что может быть непривычным для разработчиков, привыкших к традиционному синхронному коду.

Threading является хорошим выбором для IO-bound задач в случаях, когда важно сохранить совместимость с существующими синхронными программами. Однако ограничения GIL снижают его эффективность в задачах, требующих интенсивного использования процессора.

Multiprocessing оптимален для CPU-bound задач, так как использует преимущества многоядерных процессоров. Но сложность разработки и затраты на управление процессами делают его менее подходящим для задач с частыми операциями ввода-вывода.

3. Методология исследования

Для выполнения сравнительного анализа подходов asyncio, threading и multiprocessing необходимо разработать набор тестовых сценариев и определить методы измерения производительности. Данный раздел описывает типы задач, выбранных для экспериментов, применяемые метрики и используемые инструменты.

3.1. Типы задач для сравнения (IO-bound, CPU-bound)

Для полноты анализа исследование охватывает два ключевых типа задач, которые встречаются в программировании:

*IO-bound задачи:*

Эти задачи характеризуются длительными операциями ввода-вывода, такими как:

Сетевые запросы (например, HTTP-запросы).

Чтение и запись данных из файлов или баз данных.

Основное время выполнения таких задач связано с ожиданием завершения операций, что делает их идеальными для асинхронного или многопоточного выполнения.

Пример задачи: загрузка данных с веб-сайтов (несколько HTTP-запросов).

*CPU-bound задачи:*

Задачи, ограниченные вычислительными ресурсами, такие как:

Обработка больших массивов данных.

Математические вычисления или алгоритмы машинного обучения.

Для выполнения таких задач важно максимально использовать возможности многоядерных процессоров.

Пример задачи: вычисление факториалов или обработка больших наборов данных.

Оба типа задач будут реализованы для каждого из подходов, чтобы оценить их применимость и производительность.

3.2. Методы измерения производительности и анализа ресурсов

Для анализа производительности выбранные подходы оцениваются по следующим метрикам:

1. *Время выполнения задачи (Execution Time):*

* Общее время, затраченное на выполнение всей задачи.
* Измеряется с помощью функции time.time() или аналогов.

1. *Использование процессора (CPU Utilization):*

* Процент загруженности процессора в процессе выполнения задачи.
* Измеряется с помощью системных утилит или библиотек, таких как psutil.

1. *Использование памяти (Memory Usage):*

* Количество оперативной памяти, используемой задачей.
* Для измерения используются библиотеки tracemalloc или psutil.

1. *Количество одновременно выполняемых задач:*

Оценка способности подхода обрабатывать множество задач параллельно без значительного падения производительности.

1. *Сложность разработки:*

Качественная метрика, основанная на субъективной оценке читаемости и сложности кода, необходимого для реализации задачи.

Для каждого подхода проводится несколько запусков тестов, чтобы исключить влияние случайных факторов и получить усредненные результаты.

3.3. Инструменты разработки и тестирования

Для реализации и тестирования сценариев исследования используются следующие инструменты и библиотеки:

1. Среда разработки:

Python 3.12 (или более новая версия) для обеспечения доступа к актуальным функциям асинхронного программирования.

1. IDE:

PyCharm, VS Code или любая другая подходящая среда.

1. Библиотеки для работы с подходами:

Asyncio: стандартная библиотека asyncio для реализации асинхронных задач.

Threading: стандартная библиотека threading для работы с потоками.

Multiprocessing: стандартная библиотека multiprocessing для работы с процессами.

1. Библиотеки для тестирования и анализа:

Psutil: для мониторинга использования ресурсов (CPU, памяти, потоков и процессов).

Timeit: для точного измерения времени выполнения кода.

Requests или aiohttp: для создания IO-bound задач (например, выполнения HTTP-запросов).

Numpy: для создания вычислительно сложных задач (например, работы с массивами).

1. Системные утилиты:

Встроенные утилиты операционной системы (например, top, htop на Linux, диспетчер задач Windows) для мониторинга ресурсов.

Обоснование выбора инструментов

Выбранные библиотеки и утилиты являются стандартными и широко применимыми в сообществе Python-разработчиков. Их использование обеспечивает точность измерений и удобство в реализации тестовых сценариев.

4. Реализация задач

В данном разделе описывается процесс реализации тестовых сценариев, направленных на сравнение производительности подходов asyncio, threading и multiprocessing. Тестовые сценарии разделены на две категории: задачи ввода-вывода (IO-bound) и задачи вычислений (CPU-bound).

4.1. Описание тестовых сценариев

Для выполнения сравнительного анализа реализованы два типа задач, каждая из которых адаптирована под особенности тестируемых подходов.

4.1.1. Задачи ввода-вывода (IO-bound)

**Описание сценария:**

Для оценки производительности подходов в обработке IO-bound задач используется сценарий, связанный с загрузкой данных из нескольких источников. Каждый подход будет выполнять множество HTTP-запросов к фиктивным или реальным API и измерять суммарное время выполнения всех операций.

**Условия эксперимента:**

Отправляется 500 HTTP-запросов к серверу с фиктивной задержкой. Если мы хотим максимально точно измерить производительность подходов, лучше использовать тестовые API (например, https://httpbin.org/delay/1) или API с известной задержкой. Это позволяет избежать влияния внешних факторов, таких как скорость сети или задержки серверов.

Все запросы независимы и могут выполняться параллельно.

Оцениваются время выполнения и количество используемых ресурсов (CPU, память).

**Реализация:**

Asyncio: Используется библиотека aiohttp для выполнения асинхронных HTTP-запросов.

Threading: Библиотека requests в сочетании с потоками для обработки запросов.

Multiprocessing: Библиотека requests в сочетании с процессами, каждый из которых выполняет свою часть запросов.

**Код для каждого подхода:**

* **asyncio:**
* **threading:**
* **multiprocessing:**

находиться в приложениях.

4.1.2. Задачи вычислений (CPU-bound)

**Описание сценария:**

Для оценки производительности в обработке CPU-bound задач используется сценарий вычисления факториала для большого числа (например, 200 000) с многократным повторением операции.

**Условия эксперимента:**

Выполняется 1000 вычислений факториала для числа 200 000.

Оцениваются время выполнения и степень загрузки CPU.

**Реализация:**

Asyncio: С использованием асинхронных функций, разделяющих вычисления на небольшие части для выполнения в цикле событий.

Threading: Потоки выполняют расчеты параллельно, но ограничены GIL.

Multiprocessing: Каждое вычисление выполняется в отдельном процессе, что позволяет эффективно использовать многоядерный процессор.

**Код для каждого подхода:**

* **asyncio:**
* **threading:**
* **multiprocessing:**

находиться в приложениях.

4.2. Реализация сценариев с использованием подходов

В данном разделе приводится реализация тестовых сценариев для IO-bound и CPU-bound задач, описанных ранее, с использованием подходов asyncio, threading и multiprocessing. Каждая реализация подробно рассматривается с акцентом на особенности выбранного подхода.

4.2.1. Реализация на asyncio

IO-bound задачи:

Асинхронный подход использует событийный цикл для обработки множества задач ввода-вывода. Библиотека aiohttp позволяет эффективно управлять асинхронными HTTP-запросами.

CPU-bound задачи:

Для выполнения вычислений используется асинхронная функция, которая может разделять выполнение задач на более мелкие этапы.

Особенность реализации на asyncio: для IO-bound задач подход демонстрирует высокую эффективность благодаря кооперативной многозадачности. Однако для CPU-bound задач эффективность ограничивается однопоточностью.

4.2.2. Реализация на threading

IO-bound задачи:

Потоковый подход использует библиотеку requests для выполнения HTTP-запросов. Каждый поток отвечает за один запрос, что позволяет выполнять их конкурентно.

CPU-bound задачи:

Задачи с интенсивными вычислениями выполняются параллельно в потоках. Однако из-за GIL потоки не могут эффективно использовать многопроцессорность.

Особенность реализации на threading: потоки эффективны для IO-bound задач, но ограничение GIL снижает их производительность для CPU-bound задач.

4.2.3. Реализация на multiprocessing

IO-bound задачи:

Для выполнения HTTP-запросов в отдельных процессах используется библиотека multiprocessing. Каждый процесс выполняет свою часть задач.

CPU-bound задачи:

Каждое вычисление выполняется в отдельном процессе, что позволяет эффективно использовать многоядерный процессор.

Особенность реализации на multiprocessing: данный подход показывает высокую эффективность для CPU-bound задач за счет обхода GIL и параллельного выполнения процессов. Однако затраты на создание и управление процессами могут быть значительными для IO-bound задач.

**Выводы по реализации**

Каждый подход имеет свои сильные и слабые стороны:

Asyncio — лучший выбор для IO-bound задач, так как обеспечивает легковесное управление асинхронными операциями.

Threading хорошо справляется с IO-bound задачами, но менее эффективен для CPU-bound задач из-за GIL.

Multiprocessing оптимален для CPU-bound задач благодаря параллельной обработке, но имеет высокие затраты на управление процессами.

5. Анализ и интерпретация результатов

В данном разделе представлены результаты тестирования трех подходов (asyncio, threading, multiprocessing) на задачах двух типов (IO-bound и CPU-bound). Анализ включает сравнение производительности, оценку использования системных ресурсов и выводы о применимости каждого подхода в реальных проектах.

5.1. Сравнение производительности

Сравнение производительности основано на замерах времени выполнения задач. Для объективности каждый тест запускался несколько раз, и рассчитывалось среднее время выполнения.

5.1.1. IO-bound задачи

Для IO-bound задач, включающих 500 HTTP-запросов, результаты демонстрируют следующую картину:

| **Подход** | **Среднее время выполнения (с)** | **Примечания** |
| --- | --- | --- |
| **Asyncio** | ~ 33.96 | Высокая производительность благодаря кооперативной многозадачности. |
| **Threading** | ~ 59.73 | Эффективен, но уступает asyncio из-за управления потоками. |
| **Multiprocessing** | ~ 62.32 | Заметно медленнее из-за накладных расходов на создание процессов. |

Выводы:

Asyncio оказался наиболее эффективным для IO-bound задач благодаря отсутствию затрат на создание потоков или процессов.

Threading демонстрирует конкурентоспособные результаты, но уступает в легковесности.

Multiprocessing не подходит для IO-bound задач из-за высоких накладных расходов.

5.1.2. CPU-bound задачи

Для CPU-bound задач, включающих 1000 вычислений факториала, результаты следующие:

| **Подход** | **Среднее время выполнения (с)** | **Загрузка CPU** | **Примечания** |
| --- | --- | --- | --- |
| **Asyncio** | ~ 465.74 | 0.6% | Эффективность ограничена GIL, используется только одно ядро. |
| **Threading** | ~ 430.13 | 0.2% | Незначительно быстрее asyncio, но также ограничен GIL. |
| **Multiprocessing** | ~ 127.68 | 14.5% | Наиболее эффективен благодаря параллельной обработке на всех ядрах CPU. |

Выводы:

Multiprocessing значительно превосходит другие подходы для CPU-bound задач благодаря обходу GIL и эффективной многопроцессорной обработке.

Asyncio и Threading не подходят для CPU-bound задач, так как их производительность ограничена однопоточностью.

5.2. Потребление системных ресурсов

Оценка использования системных ресурсов (CPU, память) проводилась с помощью psutil и встроенных системных утилит.

| **Подход** | **Использование CPU** | **Использование памяти** | **Примечания** |
| --- | --- | --- | --- |
| **Asyncio** | Низкое | Низкое | Идеален для задач, требующих минимальных ресурсов. |
| **Threading** | Умеренное | Умеренное | Более затратный по сравнению с asyncio. |
| **Multiprocessing** | Высокое | Высокое | Высокая загрузка ресурсов оправдана для CPU-bound задач. |

Выводы:

Asyncio минимально нагружает систему, что делает его подходящим для задач с ограниченными ресурсами.

Threading потребляет больше ресурсов, но остается умеренно эффективным.

Multiprocessing использует максимум системных ресурсов, что оправдано только в задачах с высоким вычислительным требованием.

5.3. Применимость подходов в реальных проектах

На основе проведенных экспериментов можно сделать выводы о применимости подходов в реальных сценариях:

Asyncio

Рекомендуется для:

IO-bound задач, требующих обработки множества сетевых запросов или операций ввода-вывода.

Приложений с ограниченными ресурсами (например, серверов с низким объемом памяти).

Не подходит для:

CPU-bound задач из-за ограничений однопоточности.

Threading

Рекомендуется для:

IO-bound задач, где требуется высокая совместимость с синхронными библиотеками.

Не подходит для:

CPU-bound задач, поскольку GIL ограничивает производительность.

Multiprocessing

Рекомендуется для:

CPU-bound задач, требующих интенсивных вычислений и эффективного использования всех ядер процессора.

Не подходит для:

IO-bound задач из-за высоких накладных расходов на управление процессами.

**Общий вывод по анализу результатов**

Выбор подхода зависит от характера задачи:

Для IO-bound задач безоговорочно лидирует asyncio.

Для CPU-bound задач наилучшим выбором является multiprocessing.

Threading занимает промежуточную позицию и может быть использован в задачах, требующих простоты реализации и совместимости с синхронным кодом.

6. Выводы и рекомендации

6.1. Основные выводы исследования

В результате проведенного анализа и тестирования были получены следующие выводы:

Различия в эффективности подходов зависят от типа задачи.

IO-bound задачи: подход asyncio оказался наиболее производительным благодаря низким накладным расходам на управление задачами.

CPU-bound задачи: подход multiprocessing лидирует за счет эффективной параллелизации вычислений на уровне процессов, что позволяет использовать все доступные ядра CPU.

Ограничение GIL значительно влияет на производительность threading и asyncio для CPU-bound задач.

Потоки и асинхронные функции не могут полноценно использовать многоядерность, что делает их менее эффективными для интенсивных вычислений.

Ресурсоемкость подходов различается.

Asyncio использует минимальное количество системных ресурсов, что делает его предпочтительным для ограниченных по ресурсам сред.

Multiprocessing, хотя и наиболее производителен для CPU-bound задач, требует больших объемов памяти и процессорного времени.

Простота реализации зависит от контекста.

Threading проще в реализации для задач, где требуется синхронная совместимость.

Asyncio требует изучения специфических подходов к асинхронному программированию.

Multiprocessing требует аккуратной работы с процессами и возможностью их синхронизации.

6.2. Рекомендации по выбору подхода

На основе проведенного исследования можно дать следующие рекомендации:

Для IO-bound задач:

Рекомендуется использовать asyncio для задач, связанных с большим количеством операций ввода-вывода (например, сетевые запросы, взаимодействие с файловыми системами).

Если требуется интеграция с синхронными библиотеками, можно рассмотреть threading, но с пониманием возможных ограничений.

Для CPU-bound задач:

Использование multiprocessing предпочтительно для выполнения ресурсоемких вычислений, таких как обработка больших массивов данных или сложные математические расчеты.

Подходы asyncio и threading не рекомендуются для таких задач из-за низкой производительности.

Для гибридных задач:

Если задачи сочетают IO-bound и CPU-bound операции, можно комбинировать подходы, используя, например, asyncio для ввода-вывода и multiprocessing для вычислений.

Ограниченные ресурсы:

В условиях ограниченного объема памяти и вычислительных мощностей стоит отдавать предпочтение asyncio, особенно для IO-bound задач.

6.3. Перспективы дальнейшего изучения

Исследование альтернативных библиотек и инструментов.

Существуют другие подходы к параллелизации в Python, такие как использование фреймворков (Ray, Dask) и языков с нативной поддержкой многопоточности (например, Rust или Go). Их включение в будущие исследования может дать более полное представление о возможностях асинхронного программирования.

Оптимизация multiprocessing для IO-bound задач.

Проведение экспериментов по оптимизации использования процессов для ввода-вывода, например, с учетом распределенной обработки запросов.

Исследование гибридных подходов.

Разработка архитектур, которые сочетают преимущества asyncio для ввода-вывода и multiprocessing для вычислений в единой системе.

Влияние специфики аппаратного обеспечения.

Изучение производительности подходов на различных типах устройств (например, серверы с большим количеством ядер, устройства с низким энергопотреблением).

Изучение новых возможностей Python.

Анализ нововведений в Python (например, улучшения в многопоточности или оптимизация GIL) может дать новые перспективы для улучшения существующих подходов.

Заключение

Проведенное исследование подтвердило, что выбор подхода к асинхронному программированию в Python должен основываться на характере решаемых задач. Методы asyncio, threading и multiprocessing обладают уникальными особенностями, которые делают их более или менее подходящими для различных сценариев.

Для задач, связанных с интенсивным вводом-выводом (IO-bound), asyncio демонстрирует наивысшую эффективность благодаря низким накладным расходам и кооперативной многозадачности. В то же время, для вычислительно сложных задач (CPU-bound), multiprocessing обеспечивает наибольшую производительность, эффективно обходя ограничения GIL и используя все доступные ядра процессора. Подход threading, хотя и является компромиссным решением, сохраняет актуальность в задачах, где требуется взаимодействие с синхронными библиотеками или простой способ реализации параллелизма.

Результаты анализа и тестирования подчеркивают важность осознанного выбора инструментов программирования. Правильно подобранный подход позволяет не только значительно сократить время выполнения задач, но и оптимально использовать системные ресурсы.

Будущие исследования могут быть направлены на изучение новых инструментов и библиотек для асинхронного программирования, а также на разработку гибридных подходов, объединяющих преимущества разных методов. Это позволит сделать приложения еще более производительными, гибкими и масштабируемыми.

Таким образом, настоящее исследование представляет собой основу для принятия обоснованных решений при выборе подходов к асинхронному программированию, что способствует разработке эффективных и устойчивых приложений в современных условиях.

Список литературы

1. Бейли Д. **Асинхронное программирование в Python. Практическое руководство** / Перевод с англ. — М.: ДМК Пресс, 2020.
2. Свиридов А. **Параллельное и многопоточное программирование в Python. Основы и примеры** — М.: Питер, 2021.
3. Лутц М. **Изучаем Python. Том 2: Инструменты для профессионалов** / Перевод с англ. — СПб.: Питер, 2022.
4. Саммерфилд М. **Программирование на Python 3. Подробное руководство** / Перевод с англ. — М.: БХВ-Петербург, 2021.
5. Грёнлунд А. **Высокопроизводительное программирование в Python** / Перевод с англ. — М.: Эксмо, 2022.
6. Фаррелл Э. **Python для профессионалов** / Перевод с англ. — СПб.: Питер, 2021.
7. Кузнецов М., Артамонов А. **Асинхронное программирование: концепции и примеры на Python** // Журнал "Системное администрирование", №6, 2022.
8. Трофимов А. **Асинхронное программирование в Python: от asyncio до multiprocessing** // Хабр, 2021. [Доступно онлайн: <https://habr.com/>]
9. В. Козырев. **Асинхронное программирование и обработка данных в Python** // Журнал "Программирование и компьютеры", №4, 2023.
10. Официальная документация Python // [<https://docs.python.org/3/library/asyncio.html>]
11. Романов С. **Оптимизация работы с потоками в Python** // Канал на YouTube "Технострим Mail.Ru Group", 2022.
12. Шульгин Д. **Современные подходы к асинхронному программированию в Python** // Материалы конференции PyCon Russia, 2023.

Приложения

1. Реализация IO-bound задачи

Пример:

Выполнение 500 HTTP-запросов к тестовому API (https://httpbin.org/delay/1), возвращающему ответ с задержкой в 1 секунду. Реализовано с использованием модулей asyncio, threading, и multiprocessing.

1.1 Реализация на asyncio

import asyncio # Модуль для асинхронного программирования  
import aiohttp # Библиотека для асинхронных HTTP-запросов  
import psutil # Для мониторинга системных ресурсов  
import time # Для измерения времени выполнения  
  
  
# Асинхронная функция для выполнения одного HTTP-запроса  
async def fetch(url, success\_counter):  
 async with aiohttp.ClientSession() as session: # Создаем асинхронную HTTP-сессию  
 try:  
 async with session.get(url) as response: # Выполняем GET-запрос  
 await response.text() # Асинхронно считываем ответ (для проверки успешного получения)  
 success\_counter["success"] += 1 # Увеличиваем счетчик успешных запросов  
 except Exception as e:  
 print(f"Ошибка запроса: {e}") # Логируем ошибки  
  
  
# Основная корутина для выполнения всех запросов  
async def main\_io\_bound():  
 url = "https://httpbin.org/delay/1" # Тестовый URL с задержкой в 1 секунду  
 num\_requests = 500 # Количество запросов  
 success\_counter = {"success": 0} # Счетчик успешных запросов  
  
 # Замер времени выполнения  
 start\_time = time.time() # Засекаем начальное время  
  
 # Создаем список задач  
 tasks = [fetch(url, success\_counter) for \_ in range(num\_requests)]  
  
 # Асинхронно выполняем все задачи  
 await asyncio.gather(\*tasks)  
  
 # Засекаем конечное время выполнения  
 end\_time = time.time()  
  
 # Вывод результатов  
 print(f"Отправлено запросов: {num\_requests}")  
 print(f"Успешных ответов: {success\_counter['success']}")  
 print(f"Время выполнения: {end\_time - start\_time:.2f} секунд")  
  
 # Получение загрузки CPU  
 cpu\_usage = psutil.cpu\_percent(interval=1) # Измеряем загрузку CPU за 1 секунду  
 print(f"Загрузка CPU: {cpu\_usage}%")  
  
 # Получение информации о памяти  
 memory\_info = psutil.virtual\_memory()  
 print(f"Использовано памяти: {memory\_info.used / (1024 \*\* 2):.2f} МБ")  
 print(f"Свободно памяти: {memory\_info.available / (1024 \*\* 2):.2f} МБ")  
  
  
# Запускаем основную корутину  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main\_io\_bound()) # Запуск событийного цикла

**Комментарии к ключевым частям кода:**

* aiohttp: используется для асинхронных HTTP-запросов, что позволяет эффективно работать с IO-bound задачами.
* async def fetch(url): определяет корутину для выполнения одного запроса.
* asyncio.gather(\*tasks): запускает все запросы параллельно в одном событийном цикле.
* asyncio.run(): стандартный способ запуска корутин в Python 3.7+.

**Особенности:**

1. **Параллельное выполнение:** Асинхронный подход позволяет обрабатывать все 500 запросов параллельно, используя событийный цикл.
2. **Мониторинг успешных запросов:** Показывает, сколько запросов завершились успешно, что важно для проверки надежности.
3. **Потребление ресурсов:** Асинхронный подход минимально нагружает CPU, но может потреблять больше памяти для хранения большого числа задач.

****1.2 Реализация на**** threading

import threading # Модуль для работы с потоками  
import requests # Синхронная библиотека для HTTP-запросов  
import time # Для измерения времени выполнения  
import psutil # Для мониторинга системных ресурсов  
  
# Общий счетчик успешных запросов  
success\_counter = 0  
lock = threading.Lock() # Блокировка для синхронизации доступа к счетчику  
  
  
# Функция для выполнения одного HTTP-запроса  
def fetch(url):  
 global success\_counter # Используем глобальный счетчик  
 try:  
 response = requests.get(url) # Выполняем синхронный GET-запрос  
 with lock: # Гарантируем безопасный доступ к счетчику  
 success\_counter += 1 # Увеличиваем счетчик успешных запросов  
 except Exception as e:  
 print(f"Ошибка запроса: {e}") # Логируем ошибку  
  
  
# Основная функция для запуска задач в потоках  
def main\_io\_bound():  
 global success\_counter # Используем глобальный счетчик  
 url = "https://httpbin.org/delay/1" # Тестовый URL с задержкой 1 секунда  
 num\_requests = 500 # Количество запросов  
 threads = [] # Список для хранения потоков  
  
 # Замер времени выполнения  
 start\_time = time.time() # Засекаем начальное время  
  
 # Создаем и запускаем 100 потоков  
 for \_ in range(num\_requests):  
 thread = threading.Thread(target=fetch, args=(url,)) # Создаем поток  
 threads.append(thread) # Добавляем поток в список  
 thread.start() # Запускаем поток  
  
 # Дожидаемся завершения всех потоков  
 for thread in threads:  
 thread.join() # Блокируем основной поток до завершения текущего  
  
 end\_time = time.time() # Засекаем конечное время выполнения  
  
 # Вывод результатов  
 print(f"Отправлено запросов: {num\_requests}")  
 print(f"Успешных ответов: {success\_counter}")  
 print(f"Время выполнения: {end\_time - start\_time:.2f} секунд")  
  
 # Получение загрузки CPU  
 cpu\_usage = psutil.cpu\_percent(interval=1) # Измеряем загрузку CPU за 1 секунду  
 print(f"Загрузка CPU: {cpu\_usage}%")  
  
 # Получение информации о памяти  
 memory\_info = psutil.virtual\_memory()  
 print(f"Использовано памяти: {memory\_info.used / (1024 \*\* 2):.2f} МБ")  
 print(f"Свободно памяти: {memory\_info.available / (1024 \*\* 2):.2f} МБ")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main\_io\_bound() # Запускаем основную функцию

**Комментарии к ключевым частям кода:**

* threading.Thread: создает новый поток для выполнения функции fetch.
* thread.start(): запускает выполнение потока.
* thread.join(): блокирует выполнение основного потока, пока текущий поток не завершится.
* Потоки работают параллельно, но синхронная природа requests не позволяет эффективно использовать GIL.

**Особенности**

1. **Параллельное выполнение:** Потоки позволяют обрабатывать несколько запросов одновременно, но их производительность может быть ограничена из-за GIL.
2. **Мониторинг ресурсов:** Добавлены замеры времени выполнения, загрузки CPU и использования памяти.
3. **Потокобезопасность:** Использование lock обеспечивает правильную работу счетчика успешных запросов в многопоточном окружении.

****1.3 Реализация на**** multiprocessing

import multiprocessing # Модуль для работы с процессами  
import requests # Синхронная библиотека для HTTP-запросов  
import time # Для измерения времени выполнения  
import psutil # Для мониторинга системных ресурсов  
  
  
# Функция для выполнения одного HTTP-запроса  
def fetch(url, success\_counter):  
 try:  
 response = requests.get(url) # Выполняем синхронный GET-запрос  
 success\_counter["success"] += 1 # Увеличиваем счетчик успешных запросов  
 except Exception as e:  
 print(f"Ошибка запроса: {e}") # Логируем ошибку  
  
  
# Основная функция для запуска задач в процессах  
def main\_io\_bound():  
 url = "https://httpbin.org/delay/1" # Тестовый URL с задержкой 1 секунда  
 num\_requests = 500 # Количество запросов  
 processes = [] # Список для хранения процессов  
 success\_counter = multiprocessing.Manager().dict() # Общий словарь для подсчета успешных запросов  
 success\_counter["success"] = 0 # Инициализируем счетчик успешных запросов  
  
 # Замер времени выполнения  
 start\_time = time.time() # Засекаем начальное время  
  
 # Создаем и запускаем 100 процессов  
 for \_ in range(num\_requests):  
 process = multiprocessing.Process(target=fetch, args=(url, success\_counter)) # Создаем процесс  
 processes.append(process) # Добавляем процесс в список  
 process.start() # Запускаем процесс  
  
 # Дожидаемся завершения всех процессов  
 for process in processes:  
 process.join() # Блокируем основной процесс до завершения текущего  
  
 end\_time = time.time() # Засекаем конечное время выполнения  
  
 # Вывод результатов  
 print(f"Отправлено запросов: {num\_requests}")  
 print(f"Успешных ответов: {success\_counter['success']}")  
 print(f"Время выполнения: {end\_time - start\_time:.2f} секунд")  
  
 # Получение загрузки CPU  
 cpu\_usage = psutil.cpu\_percent(interval=1) # Измеряем загрузку CPU за 1 секунду  
 print(f"Загрузка CPU: {cpu\_usage}%")  
  
 # Получение информации о памяти  
 memory\_info = psutil.virtual\_memory()  
 print(f"Использовано памяти: {memory\_info.used / (1024 \*\* 2):.2f} МБ")  
 print(f"Свободно памяти: {memory\_info.available / (1024 \*\* 2):.2f} МБ")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main\_io\_bound() # Запускаем основную функцию

**Комментарии к ключевым частям кода:**

* multiprocessing.Process: создает новый процесс для выполнения функции fetch.
* process.start(): запускает выполнение процесса.
* process.join(): блокирует выполнение основного процесса, пока текущий процесс не завершится.
* Хотя процессы обходят ограничения GIL, они менее эффективны для IO-bound задач из-за высоких накладных расходов на создание и управление процессами.

**Особенности**

1. **Параллельное выполнение:** Каждый запрос выполняется в отдельном процессе, что может обходить ограничения GIL, но увеличивает потребление памяти и накладные расходы на создание процессов.
2. **Мониторинг успешных запросов:** Счетчик позволяет убедиться, что все запросы успешно обработаны, или выявить возможные ошибки.
3. **Использование ресурсов:** Этот метод активно использует как память, так и процессорное время, особенно при большом количестве запросов.

Сравнение подходов (резюме)

* **Asyncio**: наиболее эффективный подход для IO-bound задач, требует минимальных накладных расходов.
* **Threading**: средний по эффективности, подходит для задач, совместимых с синхронным кодом.
* **Multiprocessing**: неэффективен для IO-bound задач, но остается полезным для CPU-bound задач.

1. Реализация CPU-bound задачи

Пример:

Выполнение 1000 вычислений факториала числа 200000, требующих интенсивного использования процессора. Реализация представлена с использованием модулей asyncio, threading и multiprocessing.

2.1 Реализация на asyncio

import asyncio # Библиотека для асинхронного программирования  
import math # Для вычисления факториала  
import psutil # Для мониторинга системных ресурсов  
import time # Для измерения времени выполнения  
  
  
# Функция для вычисления факториала числа  
def calculate\_factorial(n):  
 return math.factorial(n) # Используем встроенную функцию для вычисления факториала  
  
  
# Асинхронная обертка для вычисления факториала  
async def async\_factorial(n):  
 return calculate\_factorial(n) # Синхронное вычисление в асинхронной обертке  
  
  
# Основная корутина для выполнения всех задач  
async def main\_cpu\_bound():  
 n = 200000 # Число, для которого будет вычисляться факториал  
 tasks = [async\_factorial(n) for \_ in range(1000)] # Создаем 1000 задач для вычисления факториала  
  
 # Замер времени выполнения  
 start\_time = time.time() # Засекаем начальное время выполнения  
 results = await asyncio.gather(\*tasks) # Асинхронно выполняем все задачи  
 end\_time = time.time() # Засекаем конечное время выполнения  
  
 # Вывод времени выполнения  
 print(f"Время выполнения: {end\_time - start\_time:.2f} секунд")  
  
 # Получение загрузки CPU  
 cpu\_usage = psutil.cpu\_percent(interval=1) # Измеряем загрузку CPU за 1 секунду  
 print(f"Загрузка CPU: {cpu\_usage}%")  
  
 # Получение информации о памяти  
 memory\_info = psutil.virtual\_memory()  
 print(f"Использовано памяти: {memory\_info.used / (1024 \*\* 2):.2f} МБ")  
 print(f"Свободно памяти: {memory\_info.available / (1024 \*\* 2):.2f} МБ")  
  
   
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main\_cpu\_bound()) # Запуск событийного цикла

**Комментарии к ключевым частям кода:**

* calculate\_factorial: синхронная функция для вычисления факториала.
* async\_factorial: асинхронная обертка, обеспечивающая совместимость с asyncio.
* Использование asyncio не дает прироста производительности из-за блокирующего характера calculate\_factorial и действия GIL.

2.2 Реализация на threading

import threading # Модуль для работы с потоками  
import math # Для вычисления факториала  
import psutil # Для мониторинга системных ресурсов  
import time # Для измерения времени выполнения  
  
  
# Функция для вычисления факториала числа  
def calculate\_factorial(n):  
 return math.factorial(n) # Вычисляем факториал числа n  
  
  
# Задача для выполнения в потоке  
def thread\_task(n):  
 result = calculate\_factorial(n) # Вычисляем факториал  
  
  
# Основная функция для запуска потоков  
def main\_cpu\_bound():  
 n = 200000 # Число, для которого будет вычисляться факториал  
 threads = [] # Список для хранения потоков  
  
 # Замер времени выполнения  
 start\_time = time.time() # Засекаем начальное время  
  
 # Создаем и запускаем 1000 потоков  
 for \_ in range(1000):  
 thread = threading.Thread(target=thread\_task, args=(n,)) # Создаем поток для задачи  
 threads.append(thread) # Добавляем поток в список  
 thread.start() # Запускаем поток  
  
 # Дожидаемся завершения всех потоков  
 for thread in threads:  
 thread.join() # Ожидаем завершения потока  
  
 end\_time = time.time() # Засекаем конечное время выполнения  
  
 # Вывод времени выполнения  
 print(f"Время выполнения: {end\_time - start\_time:.2f} секунд")  
  
 # Получение загрузки CPU  
 cpu\_usage = psutil.cpu\_percent(interval=1) # Измеряем загрузку CPU за 1 секунду  
 print(f"Загрузка CPU: {cpu\_usage}%")  
  
 # Получение информации о памяти  
 memory\_info = psutil.virtual\_memory()  
 print(f"Использовано памяти: {memory\_info.used / (1024 \*\* 2):.2f} МБ")  
 print(f"Свободно памяти: {memory\_info.available / (1024 \*\* 2):.2f} МБ")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main\_cpu\_bound() # Запуск основной функции

**Комментарии к ключевым частям кода:**

* thread\_task: обрабатывает вычисление факториала в отдельном потоке.
* GIL ограничивает одновременное выполнение потоков, что делает подход неэффективным для вычислительных задач.

**Особенности:**

1. Потоки эффективно используют многозадачность, но из-за GIL (Global Interpreter Lock) Python не позволяет потокам одновременно выполнять операции на нескольких ядрах процессора.
2. Этот код теперь позволяет не только выполнить задачи, но и мониторить, как потоки влияют на ресурсы системы.

2.3 Реализация на multiprocessing

import multiprocessing # Модуль для работы с процессами  
import math # Для вычисления факториала  
import psutil # Для мониторинга системных ресурсов  
import time # Для измерения времени выполнения  
  
  
# Функция для вычисления факториала числа  
def calculate\_factorial(n):  
 return math.factorial(n) # Вычисляем факториал числа n  
  
  
# Задача для выполнения в процессе  
def process\_task(n):  
 result = calculate\_factorial(n) # Вычисляем факториал  
  
  
# Основная функция для запуска процессов  
def main\_cpu\_bound():  
 n = 200000 # Число, для которого будет вычисляться факториал  
 processes = [] # Список для хранения процессов  
  
 # Замер времени выполнения  
 start\_time = time.time() # Засекаем начальное время  
  
 # Создаем и запускаем 1000 процессов  
 for \_ in range(1000):  
 process = multiprocessing.Process(target=process\_task, args=(n,)) # Создаем процесс для задачи  
 processes.append(process) # Добавляем процесс в список  
 process.start() # Запускаем процесс  
  
 # Дожидаемся завершения всех процессов  
 for process in processes:  
 process.join() # Ожидаем завершения процесса  
  
 end\_time = time.time() # Засекаем конечное время выполнения  
  
 # Вывод времени выполнения  
 print(f"Время выполнения: {end\_time - start\_time:.2f} секунд")  
  
 # Получение загрузки CPU  
 cpu\_usage = psutil.cpu\_percent(interval=1) # Измеряем загрузку CPU за 1 секунду  
 print(f"Загрузка CPU: {cpu\_usage}%")  
  
 # Получение информации о памяти  
 memory\_info = psutil.virtual\_memory()  
 print(f"Использовано памяти: {memory\_info.used / (1024 \*\* 2):.2f} МБ")  
 print(f"Свободно памяти: {memory\_info.available / (1024 \*\* 2):.2f} МБ")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main\_cpu\_bound() # Запуск основной функции

**Комментарии к ключевым частям кода:**

* Каждый процесс работает независимо, эффективно используя многопоточность и обходит GIL.
* Это делает multiprocessing оптимальным выбором для задач, требующих высокой вычислительной мощности.

**Особенности:**

1. **Параллельное выполнение:** Использование процессов позволяет обойти GIL, так как каждый процесс запускается в своем экземпляре интерпретатора Python.
2. **Потребление ресурсов:** Этот метод обычно требует больше памяти, чем threading, но позволяет полностью задействовать ядра процессора.
3. **Информативный мониторинг:** Время выполнения, загрузка CPU и использование памяти дают полное представление о том, как процессный подход справляется с задачей.

Сравнение подходов (резюме)

1. **Asyncio**:
   * Подходит для задач с низкими вычислительными нагрузками.
   * GIL препятствует одновременной работе нескольких корутин на одном ядре.
2. **Threading**:
   * Удобен для задач, требующих простого параллелизма.
   * GIL снижает производительность для CPU-bound задач.
3. **Multiprocessing**:
   * Наиболее эффективный подход для интенсивных вычислений, так как использует несколько ядер процессора.
   * Более высокие накладные расходы на запуск процессов компенсируются увеличением производительности.

### 