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

Содержание

[1. Введение 2](#_Toc186982508)

[1.1. Обоснование выбора темы 2](#_Toc186982509)

[1.2. Определение цели и задач исследования 3](#_Toc186982510)

[2. Основные понятия и определения 4](#_Toc186982511)

[2.1. Асинхронное программирование 4](#_Toc186982512)

[2.2. Многопоточность (threading) 5](#_Toc186982513)

[2.3. Многопроцессность (multiprocessing) 5](#_Toc186982514)

[2.4. Asyncio 6](#_Toc186982515)

[2.5. Сравнительные термины и понятия 7](#_Toc186982516)

[2.5.1. Блокирующие и неблокирующие операции 8](#_Toc186982517)

[2.5.2. Синхронное vs Асинхронное выполнение 8](#_Toc186982518)

[2.5.3. Конкурентность и Параллелизм 9](#_Toc186982519)

[2.5.4. Корутины 9](#_Toc186982520)

[2.5.5. Таблица сравнения: Синхронное vs Асинхронное выполнение 10](#_Toc186982521)

[3. Методы и подходы к разработке 11](#_Toc186982522)

[3.1. Методология исследования 11](#_Toc186982523)

[3.2. Критерии оценки производительности 12](#_Toc186982524)

[3.3. Инструменты и среды разработки 13](#_Toc186982525)

[4. Разработка с помощью разных подходов 14](#_Toc186982526)

[4.1. Обычная реализация без оптимизаций 14](#_Toc186982527)

[4.2. Реализация с использованием asyncio 15](#_Toc186982528)

[4.3. Реализация с использованием threading 17](#_Toc186982529)

[4.4. Реализация с использованием multiprocessing 19](#_Toc186982530)

[5. Анализ и интерпретация результатов 21](#_Toc186982531)

[5.1. Сравнение производительности 21](#_Toc186982532)

[5.2. Уместность подходов для различных типов задач 22](#_Toc186982533)

[5.3. Выводы по полученным результатам 23](#_Toc186982534)

[6. Заключение 24](#_Toc186982535)

[6.1. Обзор выполненной работы 24](#_Toc186982536)

[6.2. Дальнейшие планы 25](#_Toc186982537)

[Приложения 26](#_Toc186982538)

[Приложение 1. Синхронная реализация 26](#_Toc186982539)

[Приложение 2. Асинхронная реализация 28](#_Toc186982540)

[Приложение 3. Реализация с использованием threading 30](#_Toc186982541)

[Приложение 4. Реализация с использованием multiprocessing 33](#_Toc186982542)

# 1. Введение

## 1.1. Обоснование выбора темы

В современном мире программирования эффективность и производительность приложений играют ключевую роль в их успешности и конкурентоспособности. С ростом сложности задач и увеличением объема обрабатываемых данных разработчики сталкиваются с необходимостью оптимизации процессов выполнения программ. Асинхронное программирование предоставляет инструменты и подходы, позволяющие эффективно использовать ресурсы системы, улучшать отклик приложений и снижать время обработки задач.

Асинхронное программирование в Python реализуется различными методами, среди которых наиболее распространенными являются модули asyncio, threading и multiprocessing. Каждый из этих подходов имеет свои особенности, преимущества и ограничения, что делает актуальным их сравнение для выбора оптимального решения в зависимости от конкретных требований проекта.

Выбор подходящего метода асинхронного программирования напрямую влияет на производительность приложения, его масштабируемость и устойчивость к нагрузкам. Понимание различий между asyncio, threadingи multiprocessing позволяет разработчикам принимать обоснованные решения, адаптируя архитектуру приложений под специфические задачи и условия эксплуатации.

Таким образом, исследование сравнительных характеристик различных подходов к реализации асинхронного программирования на Python имеет значительную практическую ценность. Оно способствует развитию навыков выбора и применения наиболее эффективных инструментов для решения конкретных задач, что, в свою очередь, повышает качество и надежность разрабатываемого программного обеспечения.

## 1.2. Определение цели и задач исследования

**Цель исследования:** сравнение производительности и уместности использования asyncio, threading и multiprocessing для различных типов задач в Python.

**Задачи исследования:**

1. Анализ существующих подходов к асинхронному программированию:

* Изучение теоретических основ асинхронного программирования.
* Обзор модулей asyncio, threading и multiprocessing в контексте Python.
* Выявление ключевых особенностей, преимуществ и ограничений каждого подхода.

1. Реализация асинхронных задач с использованием asyncio, threading и multiprocessing:

* Разработка примеров приложений, реализующих идентичные функциональные задачи с использованием каждого из трех подходов.
* Обеспечение сопоставимости условий тестирования для корректного сравнения.

1. Сравнение производительности реализованных решений:

* Проведение тестов на выполнение разработанных приложений.
* Сбор и анализ метрик производительности, таких как время выполнения, использование ресурсов (CPU, память) и масштабируемость.

1. Оценка уместности каждого подхода для различных типов задач:

* Классификация задач на I/O-bound и CPU-bound.
* Определение, какой из подходов наиболее эффективно решает каждую категорию задач.
* Анализ ситуаций, в которых один подход может превосходить другие по эффективности и удобству использования.

1. Формулирование выводов и рекомендаций на основе проведенного анализа:

* Систематизация полученных результатов и их интерпретация.
* Разработка рекомендаций по выбору подходящего метода асинхронного программирования в зависимости от специфики задачи.
* Обсуждение возможных направлений дальнейших исследований и улучшений.

# 2. Основные понятия и определения

В данном разделе рассматриваются ключевые концепции и термины, связанные с асинхронным программированием, многопоточностью и многопроцессностью в контексте языка Python. Понимание этих понятий является основой для дальнейшего анализа и сравнения различных подходов к реализации асинхронного программирования.

## 2.1. Асинхронное программирование

Асинхронное программирование — это парадигма программирования, позволяющая выполнять несколько операций одновременно без блокировки основного потока выполнения. В отличие от синхронного программирования, где операции выполняются последовательно, асинхронное программирование позволяет эффективно использовать ресурсы системы, особенно в задачах, связанных с вводом-выводом (I/O-bound).

Ключевые принципы асинхронного программирования:

* **Неблокирующие операции:** Асинхронные функции не блокируют основной поток выполнения, позволяя другим задачам выполняться параллельно.
* **Конкурентность:** Возможность выполнения нескольких задач одновременно, что повышает общую производительность приложения.
* **Событийный цикл:** Центральный механизм, управляющий выполнением асинхронных задач, планируя их выполнение и обрабатывая события.
* **Корутины:** Специальные функции, которые могут приостанавливаться и возобновляться, обеспечивая асинхронное выполнение.

Асинхронное программирование особенно полезно в приложениях, требующих высокой отзывчивости и эффективного управления ресурсами, таких как веб-серверы, приложения реального времени и системы обработки больших объемов данных.

## 2.2. Многопоточность (threading)

Многопоточность — это метод параллельного выполнения нескольких потоков внутри одного процесса. В Python реализация многопоточности осуществляется с помощью модуля threading, который предоставляет инструменты для создания и управления потоками.

Характеристики многопоточности:

1. **Совместное использование памяти:** Все потоки внутри процесса имеют доступ к общей памяти, что упрощает обмен данными между ними.
2. **Легковесность:** Потоки легче процессов, что позволяет создавать большое количество потоков с минимальными накладными расходами.
3. **Глобальная блокировка интерпретатора (GIL):** В Python существует GIL, который ограничивает выполнение байт-кода Python одним потоком одновременно. Это снижает эффективность многопоточности в задачах, связанных с интенсивными вычислениями (CPU-bound).

Применение многопоточности:

* **I/O-bound задачи:** Такие задачи, как сетевые операции, ввод-вывод с диска и взаимодействие с пользователем, могут эффективно использовать многопоточность для повышения производительности.
* **Параллельное выполнение задач:** Многопоточность позволяет выполнять несколько задач одновременно, улучшая отзывчивость приложений.
* **Обработка событий:** В приложениях с большим количеством событий и асинхронных операций многопоточность может быть использована для управления различными потоками событий.

## 2.3. Многопроцессность (multiprocessing)

Многопроцессность предполагает параллельное выполнение нескольких процессов, каждый из которых имеет собственное пространство памяти. В Python реализация многопроцессности осуществляется с помощью модуля multiprocessing, который предоставляет интерфейсы для создания и управления процессами.

Характеристики многопроцессности:

1. **Изолированная память:** Каждый процесс имеет собственное пространство памяти, что обеспечивает безопасность и предотвращает конфликты при доступе к данным.
2. **Отсутствие влияния GIL:** В отличие от многопоточности, многопроцессность не ограничена GIL, что позволяет эффективно использовать несколько ядер процессора для выполнения CPU-bound задач.
3. **Высокие накладные расходы:** Создание и управление процессами требует больше ресурсов по сравнению с потоками, что может ограничивать количество одновременно выполняемых процессов.

Применение многопроцессности:

* **CPU-bound задачи:** Интенсивные вычисления, такие как обработка данных, численные расчеты и машинное обучение, могут эффективно использовать многопроцессность для ускорения выполнения.
* **Параллельная обработка данных:** Многопроцессность позволяет распределять обработку больших объемов данных между несколькими процессами, улучшая общую производительность.
* **Изоляция задач:** В системах, где требуется изолировать задачи друг от друга для повышения надежности и безопасности, многопроцессность является предпочтительным подходом.

## 2.4. Asyncio

asyncio — это стандартная библиотека Python, предназначенная для реализации асинхронного программирования. Она предоставляет инструменты для создания событийных циклов, корутин, задач и управления асинхронными операциями.

Характеристики asyncio:

1. **Событийный цикл:** Центральный компонент asyncio, управляющий выполнением корутин и обработкой событий.
2. **Корутины и задачи:** Корутины — это функции, определенные с использованием ключевого слова async, которые могут приостанавливаться и возобновляться. Задачи представляют собой планируемые корутины, которые выполняются в событийном цикле.
3. **Неблокирующие операции:** asyncio позволяет выполнять I/O-bound задачи без блокировки основного потока выполнения, обеспечивая высокую отзывчивость приложений.
4. **Поддержка асинхронных библиотек:** Множество библиотек и фреймворков интегрированы с asyncio, что упрощает разработку асинхронных приложений.

Применение asyncio:

* **Веб-серверы и сетевые приложения:** asyncio используется в популярных веб-фреймворках, таких как aiohttp, для создания высокопроизводительных веб-серверов.
* **Обработка событий и задач реального времени:** Приложения, требующие быстрой обработки событий и низкой задержки, могут эффективно использовать asyncio.
* **Асинхронные API и микросервисы:** asyncio облегчает создание асинхронных интерфейсов и микросервисной архитектуры, повышая масштабируемость и производительность.

## 2.5. Сравнительные термины и понятия

В данной подчасти рассматриваются ключевые термины и понятия, которые играют важную роль в понимании различий между синхронным и асинхронным программированием. Эти понятия помогают лучше ориентироваться в подходах к реализации многозадачности и выбора оптимального метода для конкретных задач.

### 2.5.1. Блокирующие и неблокирующие операции

**Блокирующие операции** — это такие операции, при выполнении которых выполнение программы приостанавливается до тех пор, пока операция не завершится. Примеры блокирующих операций включают чтение или запись файлов, сетевые запросы и другие операции ввода-вывода (I/O). Во время выполнения блокирующей операции основной поток программы не может выполнять другие задачи, что может привести к снижению производительности и отзывчивости приложения.

**Неблокирующие операции** позволяют программе продолжать выполнение других задач, не дожидаясь завершения текущей операции. Это достигается путем использования специальных механизмов, таких как асинхронные вызовы, события или коллбеки. Неблокирующие операции особенно полезны в асинхронном программировании, так как они позволяют эффективно использовать ресурсы системы и повышать производительность приложений.

Пример:

**Блокирующая операция:** Чтение большого файла синхронно, где программа ждет окончания чтения файла, прежде чем перейти к следующей задаче.

**Неблокирующая операция:** Чтение того же файла асинхронно, позволяя программе выполнять другие задачи параллельно с чтением файла.

### 2.5.2. Синхронное vs Асинхронное выполнение

**Синхронное выполнение** подразумевает, что задачи выполняются последовательно, одна за другой. Каждая задача должна завершиться перед началом выполнения следующей. Такой подход прост в реализации и понимании, но может быть неэффективным в условиях высокой нагрузки или при выполнении I/O-bound задач, где время ожидания может существенно замедлить общее выполнение программы.

**Асинхронное выполнение** позволяет выполнять несколько задач одновременно или конкурентно, без необходимости ожидания завершения предыдущих задач. Это достигается с помощью механизмов управления задачами, таких как события, корутины или потоки. Асинхронное выполнение повышает производительность и отзывчивость приложений, особенно в условиях многозадачности и высокой нагрузки.

Пример:

**Синхронное выполнение:** Веб-сервер обрабатывает запросы последовательно, каждый запрос блокирует выполнение следующего до завершения.

**Асинхронное выполнение:** Веб-сервер обрабатывает несколько запросов одновременно, позволяя одному запросу выполняться, пока другой ожидает ответа от базы данных.

### 2.5.3. Конкурентность и Параллелизм

**Конкурентность** — это способность системы управлять несколькими задачами одновременно, переключаясь между ними. В контексте программирования это означает, что несколько задач могут быть в стадии выполнения или ожидания одновременно, но не обязательно выполняются одновременно.

**Параллелизм** — это одновременное выполнение нескольких задач на разных процессорах или ядрах. Параллелизм позволяет действительно выполнять несколько задач одновременно, что особенно полезно для CPU-bound задач, требующих интенсивных вычислений.

Различие:

**Конкурентность** фокусируется на управлении множеством задач, обеспечивая их эффективное выполнение без необходимости параллельного выполнения.

**Параллелизм** требует наличия нескольких процессорных ядер и позволяет выполнять задачи одновременно, что приводит к значительному ускорению выполнения CPU-bound задач.

### 2.5.4. Корутины

**Корутины** — это специальные функции, которые могут приостанавливать свое выполнение и возобновляться позже. В Python корутины определяются с помощью ключевого слова async и работают в рамках асинхронного программирования. Они позволяют создавать неблокирующие операции и эффективно управлять асинхронными задачами.

Преимущества корутин:

**Эффективное использование ресурсов:** Корутины позволяют выполнять множество задач в одном потоке без необходимости переключения контекста.

**Простота написания асинхронного кода:** Корутины обеспечивают синтаксически понятный способ написания асинхронных операций.

**Высокая производительность для I/O-bound задач:** Корутины идеально подходят для задач, связанных с вводом-выводом, где время ожидания может быть использовано для выполнения других задач.

### 2.5.5. Таблица сравнения: Синхронное vs Асинхронное выполнение

| **Параметр** | **Синхронное выполнение** | **Асинхронное выполнение** |
| --- | --- | --- |
| **Блокирующие операции** | Да | Нет |
| **Неблокирующие операции** | Нет | Да |
| **Управление задачами** | Последовательное выполнение | Параллельное или конкурентное выполнение |
| **Использование ресурсов** | Менее эффективно для I/O-bound задач | Высокая эффективность для I/O-bound задач |
| **Сложность реализации** | Простая реализация и отладка | Более сложная реализация и отладка |
| **Производительность** | Может снижаться при высокой нагрузке или большом количестве задач | Повышенная производительность при правильном использовании |
| **Применимость** | Подходит для простых приложений с небольшим количеством задач | Подходит для приложений с высокой нагрузкой и множеством I/O-bound задач |
| **Примеры реализации** | Традиционные функции и методы | asyncio, threading, multiprocessing |

В этом разделе были рассмотрены основные понятия и определения, связанные с асинхронным программированием, многопоточностью и многопроцессностью в Python. Понимание различий между этими подходами, их характеристик и областей применения является фундаментом для дальнейшего анализа и сравнения их эффективности и уместности в различных типах задач. Эти знания позволят обоснованно выбирать наиболее подходящий метод разработки в зависимости от специфики проекта и требований к производительности.

# 3. Методы и подходы к разработке

В данном разделе рассматриваются методы и подходы, используемые для реализации и сравнения различных методов асинхронного программирования в Python. Он включает описание методологии исследования, критериев оценки производительности, а также инструментов и сред разработки, применяемых в процессе разработки и тестирования.

## 3.1. Методология исследования

Методология исследования определяет общий план и подходы, используемые для достижения целей и решения поставленных задач дипломной работы. В данном исследовании применяются следующие методы:

Аналитический метод:

* Изучение существующих научных работ, статей, документации и других источников информации, касающихся асинхронного программирования, многопоточности и многопроцессности в Python.
* Анализ различных подходов к асинхронному программированию, выявление их преимуществ и ограничений.

Экспериментальный метод:

* Создание примеров приложений с использованием asyncio, threading и multiprocessing, реализующих идентичные функциональные задачи для обеспечения сопоставимости результатов.
* Проведение тестов на выполнение разработанных приложений с целью измерения производительности и оценки использования ресурсов.

Статистический метод:

* Сбор количественных данных о времени выполнения, использовании CPU и памяти для каждого подхода.
* Обработка и интерпретация собранных данных с использованием статистических методов для выявления значимых различий и тенденций.

## 3.2. Критерии оценки производительности

Для объективного сравнения производительности различных подходов к асинхронному программированию в Python используются следующие критерии и метрики:

**Время выполнения:**

* Измеряется время от начала до завершения выполнения задачи.
* Время, необходимое для обработки отдельного запроса или операции.

**Использование ресурсов:**

* Процент использования CPU во время выполнения задачи.
* Объем оперативной памяти, используемый приложением во время выполнения задачи.

**Масштабируемость -** оценка, как производительность меняется при увеличении нагрузки или числа параллельных задач.

**Эффективность использования ресурсов** - соотношение между временем выполнения и потребляемыми ресурсами.

**Отзывчивость** - способность системы быстро реагировать на входные данные или события.

**Надежность и устойчивость:**

* Способность системы корректно обрабатывать ошибки и исключения без сбоев.
* Способность системы сохранять производительность и стабильность при увеличении нагрузки.

**Удобство разработки и поддержки:**

* Оценка трудоемкости и сложности кода для каждого подхода.
* Наличие инструментов для отладки и мониторинга выполнения приложений.

## 3.3. Инструменты и среды разработки

Для реализации и тестирования различных подходов к асинхронному программированию в Python используются следующие инструменты и среды разработки:

Язык программирования:

**Python 3.12 или выше:** Используется последняя стабильная версия Python для обеспечения доступа к актуальным функциям и библиотекам, включая улучшения в asyncio и многопоточности.

Библиотеки и модули:

**Asyncio:** Стандартная библиотека Python для асинхронного программирования.

**Threading:** Стандартный модуль для реализации многопоточности.

**Multiprocessing:** Стандартный модуль для реализации многопроцессности.

**Psutil:** Библиотека для мониторинга системных ресурсов (CPU, память и т.д.).

**Aiohttp:** Библиотека для асинхронных HTTP-запросов.

**Requests:** Библиотека для синхронных HTTP-запросов.

В данном разделе были подробно описаны методы и подходы, используемые для реализации и сравнения различных методов асинхронного программирования в Python. Методология исследования включает аналитические, экспериментальные и статистические методы, обеспечивающие всесторонний анализ выбранной темы. Критерии оценки производительности определяют ключевые метрики для объективного сравнения подходов, а описание инструментов и сред разработки демонстрирует техническую базу, необходимую для успешного проведения экспериментов и тестирования. Этот комплекс методов и инструментов обеспечивает надежную основу для проведения дальнейшего исследования и достижения поставленных целей дипломной работы.

# 4. Разработка с помощью разных подходов

В данном разделе описан процесс разработки приложения для взаимодействия с API списка университетов России с использованием различных методов асинхронного программирования. Целью является реализация идентичных функциональных задач разными способами для последующего сравнения их производительности и эффективности.

## 4.1. Обычная реализация без оптимизаций

В этом подразделе представлена базовая, синхронная реализация приложения, которая последовательно выполняет HTTP-запросы к API списка университетов Российской Федерации. Приложение измеряет время выполнения запросов, использование CPU и памяти, а затем записывает результаты в файл для последующего анализа. Такой подход служит отправной точкой для сравнения с асинхронными методами.

**Алгоритм работы:**

1. Отправка HTTP-запроса для получения списка всех университетов.
2. Получение и обработка ответа от API.
3. Вывод информации пользователю.
4. Тестирование производительности путем выполнения нескольких запросов.
5. Запись результатов в файл для дальнейшего сравнения.

**Преимущества:**

* Простота реализации: Код легко понять и отладить.
* Легкость отладки: Отладка последовательного кода менее сложна по сравнению с многопоточными или асинхронными реализациями.

**Ограничения:**

* Низкая производительность при большом количестве запросов: Последовательное выполнение запросов может занимать значительное время.
* Отсутствие параллелизма: Нет возможности выполнять несколько запросов одновременно, что приводит к увеличению общего времени выполнения.
* Возможные задержки при обработке I/O-операций: Ожидание ответа от API блокирует выполнение следующих операций.

Пример реализации указан в Приложении 1. Результат работы кода указан на Рис.1.

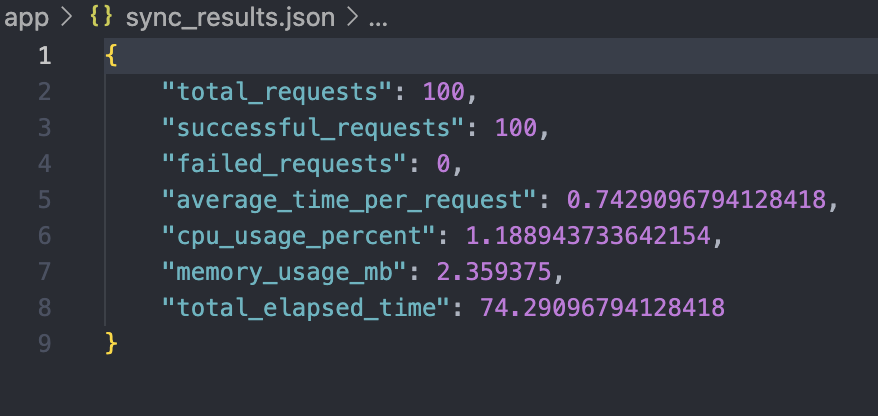


Рисунок 1. Результаты работы синхронных запросов

* total\_requests — общее количество выполненных запросов.
* successful\_requests — количество успешных запросов.
* failed\_requests — количество неудачных запросов.
* average\_time\_per\_request — среднее время выполнения одного запроса (в секундах).
* cpu\_usage\_percent — изменение использования CPU (%) во время теста.
* memory\_usage\_mb — изменение использования памяти (в МБ) во время теста.
* total\_elapsed\_time — общее затраченное время на тестирование (в секундах).

## 4.2. Реализация с использованием asyncio

В этом подразделе представлена асинхронная реализация приложения, использующая библиотеку asyncio для параллельного выполнения HTTP-запросов к API списка университетов Российской Федерации. Асинхронный подход позволяет значительно повысить производительность при обработке большого количества запросов за счет неблокирующего выполнения операций ввода-вывода.

**Алгоритм работы**

1. Изучение asyncio: Понимание основных концепций asyncio, таких как событийный цикл, корутины и задачи.
2. Модификация запросов: Переписывание функций отправки HTTP-запросов с использованием асинхронных библиотек (aiohttp).
3. Параллельное выполнение задач: Организация параллельного выполнения нескольких запросов для повышения производительности.
4. Обработка результатов: Асинхронная обработка полученных данных, управление исключениями.
5. Тестирование производительности: Выполнение серии асинхронных запросов и сбор метрик.
6. Запись результатов: Сохранение полученных метрик в файл для дальнейшего сравнения.

**Преимущества:**

* Высокая производительность при обработке большого количества запросов: Асинхронный подход позволяет выполнять множество запросов одновременно, значительно снижая общее время выполнения.
* Эффективное использование ресурсов: Неблокирующие операции позволяют системе более эффективно использовать CPU и память.
* Возможность масштабирования: Легко масштабируется для обработки увеличивающегося числа запросов без существенного увеличения затрат на ресурсы.

**Ограничения:**

* Более сложная реализация по сравнению с синхронным подходом: Асинхронный код требует понимания концепций asyncio, таких как корутины и событийный цикл.
* Необходимость понимания асинхронных концепций: Разработка и отладка асинхронного кода может быть сложнее, особенно для начинающих разработчиков.
* Зависимость от асинхронных библиотек: Для работы с HTTP-запросами используется aiohttp, что добавляет зависимости и требует дополнительной настройки.

Пример реализации указан в Приложении 2. Результат работы кода указан на Рис.2.

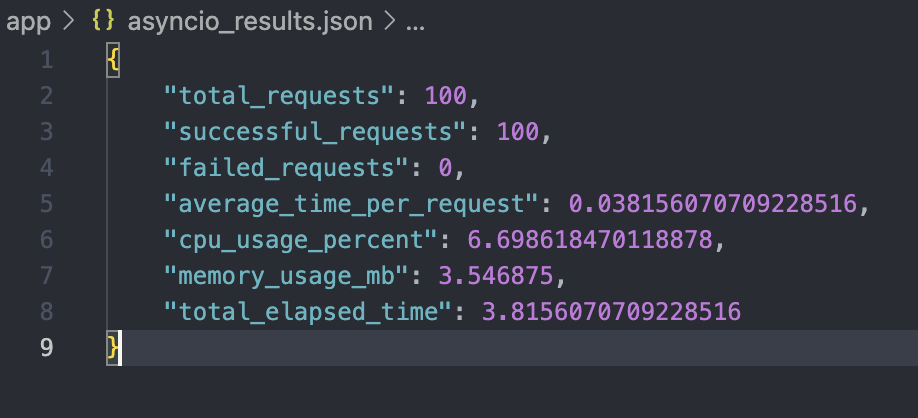


Рисунок 2. Результаты работы при помощи асинхронных запросов

## 4.3. Реализация с использованием threading

В этом подразделе представлена реализация приложения, использующая модуль threading для параллельного выполнения HTTP-запросов к API списка университетов Российской Федерации. Многопоточность позволяет одновременно выполнять несколько запросов, что может повысить производительность при обработке I/O-bound задач. Однако из-за глобальной блокировки интерпретатора (GIL) эффективность многопоточности ограничена для CPU-bound задач.

**Алгоритм работы:**

1. Изучение threading: Понимание работы потоков в Python, механизмов синхронизации и управления потоками.
2. Создание потоков: Реализация параллельных потоков для выполнения нескольких HTTP-запросов одновременно.
3. Управление потоками: Обработка завершения потоков, синхронизация данных между ними.
4. Обработка результатов: Сбор и обработка данных, полученных из различных потоков.
5. Тестирование производительности: Выполнение серии многопоточных запросов и сбор метрик.
6. Запись результатов: Сохранение полученных метрик в файл для дальнейшего сравнения.

**Преимущества:**

* Упрощение параллельного выполнения I/O-bound задач: Многопоточность позволяет одновременно выполнять несколько запросов, что может значительно снизить общее время выполнения.
* Улучшение отзывчивости приложения: Позволяет приложению продолжать выполнять другие задачи, пока потоки обрабатывают запросы.
* Простота использования: Модуль threading предоставляет простой интерфейс для создания и управления потоками.

**Ограничения:**

* Низкая эффективность для CPU-bound задач из-за GIL: В Python глобальная блокировка интерпретатора (GIL) ограничивает выполнение байткода одним потоком одновременно, что снижает эффективность многопоточности для задач, требующих интенсивных вычислений.
* Возможные сложности с синхронизацией данных между потоками: Необходимо использовать механизмы синхронизации (например, блокировки) для безопасного доступа к разделяемым данным, что может усложнить код и привести к ошибкам, таким как взаимные блокировки (deadlocks).
* Большие накладные расходы при большом количестве потоков: Создание слишком большого количества потоков может привести к значительному увеличению потребления памяти и снижению производительности из-за переключения контекста.

Пример реализации указан в Приложении 3. Результат работы кода указан на Рис.3.

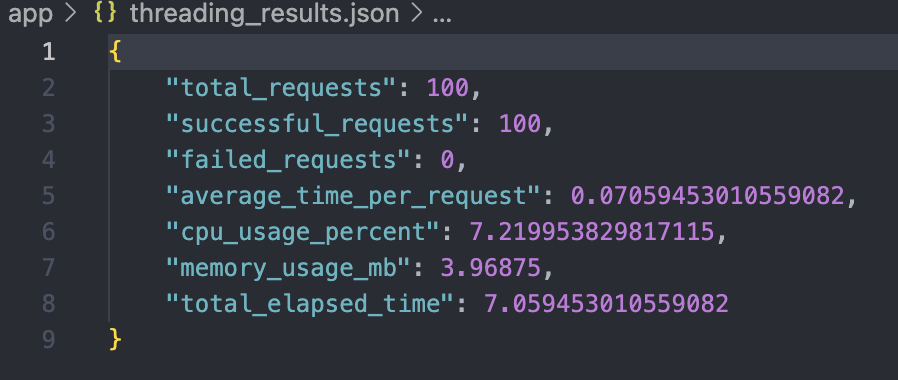


Рисунок 3. Результаты работы при помощи модуля threading

## 4.4. Реализация с использованием multiprocessing

В этом подразделе представлена реализация приложения, использующая модуль multiprocessing для параллельного выполнения HTTP-запросов к API списка университетов Российской Федерации. Модуль multiprocessing позволяет создавать отдельные процессы, каждый из которых работает независимо и имеет собственное пространство памяти. Это обходит ограничения глобальной блокировки интерпретатора (GIL) и позволяет эффективно использовать многоядерные процессоры как для I/O-bound, так и для CPU-bound задач.

Примерный алгоритм работы

1. Изучение multiprocessing: Понимание основ работы процессов в Python, механизмов межпроцессного взаимодействия и передачи данных между процессами.
2. Создание процессов: Реализация параллельных процессов для выполнения нескольких HTTP-запросов одновременно с использованием multiprocessing.Pool.
3. Управление процессами: Координация процессов, сбор результатов, обработка ошибок и контроль над количеством одновременно работающих процессов.
4. Обработка результатов: Объединение данных, полученных из разных процессов, для дальнейшего использования и анализа.

**Преимущества:**

* Эффективное использование многоядерных процессоров: Позволяет параллельно выполнять задачи на нескольких ядрах, что значительно ускоряет обработку.
* Отсутствие влияния GIL: Обходит глобальную блокировку интерпретатора (GIL), обеспечивая высокую производительность для задач, требующих интенсивных вычислений.
* Подходит как для I/O-bound, так и для CPU-bound задач: Универсальность использования позволяет эффективно применять multiprocessing в различных сценариях.

**Ограничения:**

* Высокие накладные расходы на создание и управление процессами: Запуск отдельных процессов требует больше системных ресурсов по сравнению с потоками.
* Сложности во взаимодействии между процессами: Обмен данными и синхронизация между процессами требуют дополнительных механизмов, таких как очереди или пайпы, что усложняет разработку.
* Увеличенное потребление памяти: Каждый процесс имеет своё собственное пространство памяти, что может привести к большему расходу памяти при большом количестве процессов.

Пример реализации указан в Приложении 4. Результат работы кода указан на Рис.4.

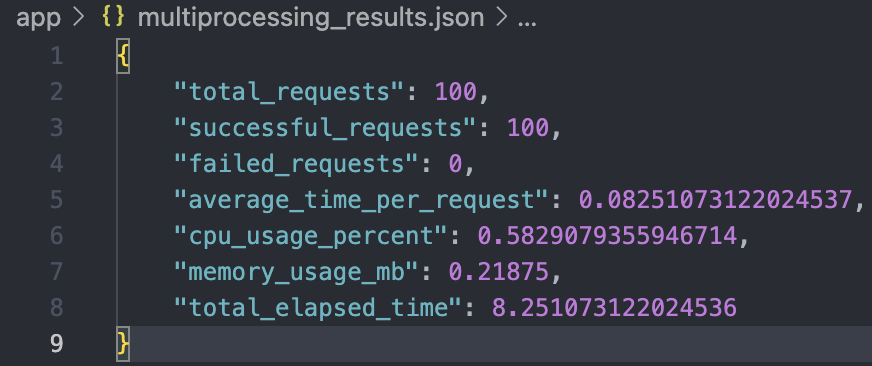


Рисунок 4. Результаты работы при помощи multiprocessing

# 5. Анализ и интерпретация результатов

## 5.1. Сравнение производительности

Сравнительный анализ времени выполнения:

* sync (синхронный):

Общее время: ~74.29 с

Среднее время на запрос: ~0.743 с

* asyncio (асинхронный):

Общее время: ~3.82 с

Среднее время на запрос: ~0.038 с

* threading (многопоточность):

Общее время: ~7.06 с

Среднее время на запрос: ~0.071 с

* multiprocessing (многопроцессность):

Общее время: ~8.25 с

Среднее время на запрос: ~0.0825 с

По этим данным видно, что асинхронный подход (asyncio) завершает все запросы быстрее всего, за ним следует многопоточность (threading), далее многопроцессность (multiprocessing) и самым длительным остаётся синхронный (sync) метод.

Сравнительный анализ использования ресурсов (CPU, память):

* sync: CPU ~1.19%, память ~2.36 МБ
* asyncio: CPU ~6.70%, память ~3.55 МБ
* threading: CPU ~7.22%, память ~3.97 МБ
* multiprocessing: CPU ~0.58%, память ~0.22 МБ
* asyncio и threading показывают более высокие значения загрузки CPU (около 6–7%), что объясняется активным управлением большим количеством задач параллельно.
* multiprocessing показывает меньшее время (около 8.25 c), но при этом использует CPU на уровне ~0.58% и память около 0.22 МБ. Подход всё равно обходится дороже в плане создания процессов, чем асинхронная или многопоточная модель, но может быть выгоден при определённых сценариях.
* sync задействует процессор минимально (~1.19%), но общее время выполнения самое большое.

## 5.2. Уместность подходов для различных типов задач

I/O-bound задачи:

* asyncio отлично подходит для большого количества сетевых операций, поскольку эффективно обрабатывает I/O без ожидания завершения каждого запроса.
* threading тоже даёт заметное ускорение, особенно когда количество запросов велико, но при интенсивной работе с большим количеством потоков может возрастать потребление памяти.
* multiprocessing для чисто I/O-задач может оказаться менее выгодным по сравнению с asyncio и threading из-за накладных расходов на создание процессов и межпроцессное взаимодействие. Однако показатели времени (около 8.25 с) всё же лучше, чем у синхронного метода (около 74 с), что может быть полезно, если нужна процессная изоляция или смешанные сценарии.
* sync — не лучший выбор для большого количества запросов, так как каждый запрос блокирует выполнение последующих.

CPU-bound задачи:

* multiprocessing эффективнее при вычислительно трудоёмких операциях, так как обходит GIL и способен задействовать все ядра.
* asyncio и threading при CPU-bound операциях могут упираться в GIL, что снизит выгоду от параллелизации.
* sync для CPU-bound не даёт выигрыша в параллелизме, но простой в реализации.

## 5.3. Выводы по полученным результатам

Интерпретация данных:

* asyncio показывает наименьшее общее время выполнения при обработке I/O-bound запросов за счёт неблокирующего управления событиями.
* threading даёт схожий выигрыш по времени, но может потребовать больше ресурсов (в частности, памяти) при масштабировании.
* multiprocessing завершает все запросы примерно за 8.25 с и использует CPU (~0.58%) и память (~0.22 МБ) довольно экономно. Это подтверждает, что метод имеет накладные расходы на создание процессов, но может быть приемлем для некоторых I/O-сценариев и особенно выгоден при CPU-bound.
* sync максимально проста, но уступает по скорости при большом количестве запросов.

Возможные причины различий в производительности:

* Архитектура подходов: asyncio и threading эффективно обрабатывают множество I/O-запросов, а multiprocessing обходится дороже при большом числе процессов.
* GIL в Python: влияет на threading и asyncio при CPU-bound, однако для I/O-bound подходит очень хорошо.
* Накладные расходы: межпроцессное взаимодействие, переключение контекста потоков, создание корутин и т.д.

Ограничения исследования:

* Стабильность сети и доступность API: результаты зависят от текущих сетевых условий, которые могут меняться.
* Окружение и конфигурация системы: объём ОЗУ, версия Python, версия psutil и прочее могут влиять на измерения.
* Тип нагрузки (I/O vs CPU): тест в основном проверяет скорость сетевых операций (I/O-bound). При CPU-bound результаты будут другими.

Таким образом, для **I/O-bound** задач (например, массовых HTTP-запросов) лучшим выбором являются **asyncio** или **threading**, причём asyncio демонстрирует лучшее время выполнения. Для **CPU-bound** операций предпочтителен **multiprocessing**, так как он эффективно использует все ядра и обходит GIL. Синхронный подход удобен для небольших и простых задач, но неэффективен при больших объёмах запросов.

# 6. Заключение

## 6.1. Обзор выполненной работы

В ходе исследования были рассмотрены четыре основных подхода к реализации асинхронных и параллельных операций в Python:

* Синхронный (sync)
* Асинхронный (asyncio)
* Многопоточный (threading)
* Многопроцессный (multiprocessing)

Каждый подход был протестирован на примере отправки 100 HTTP-запросов к удалённому API списка университетов Российской Федерации. Собранные метрики (время выполнения, использование CPU, потребление памяти) позволили сравнить эффективность каждого способа в контексте I/O-bound задач. Анализ показал, что asyncio и threading наиболее продуктивны для сценариев, где требуется параллельная обработка множества запросов ввода-вывода, тогда как multiprocessing обычно эффективнее при CPU-bound операциях, но показал более высокие накладные расходы при I/O-bound задачах. Синхронный подход продемонстрировал самую простую реализацию, однако затратил значительно больше времени на выполнение всех запросов.

Таким образом, достигнута цель исследования — провести сравнительный анализ разных методов асинхронного программирования в Python и определить наиболее подходящие подходы для различных типов задач.

## 6.2. Дальнейшие планы

1. Возможности для дальнейшего исследования

* Проверка производительности на CPU-bound задачах. Текущие эксперименты в основном касались операций ввода-вывода (сетевых запросов).
* Расширение тестовых сценариев. Можно увеличить число запросов, усложнить обработку полученных данных или протестировать другие API, чтобы увидеть, как меняются метрики производительности.

1. Рекомендации по улучшению реализованных решений

* Оптимизация сетевых параметров: Для asyncio — правильная настройка количества одновременно выполняемых задач; для threading и multiprocessing — балансировка числа потоков/процессов в зависимости от ресурсов системы.
* Добавление кеширования и retries: Реализация дополнительного механизма повторных попыток для сокрытия нестабильности сети.

1. Перспективы применения результатов в практических проектах

* Выбор оптимального подхода: Результаты исследования помогут разработчикам выбирать метод асинхронности, исходя из характера задач (I/O- или CPU-bound), требований к производительности и архитектуре системы.
* Масштабирование высоконагруженных сервисов: Применение asyncio или threading может значительно ускорить обработку большого числа запросов в веб-сервисах и микросервисных архитектурах.
* Повышение надёжности и отзывчивости приложений: Правильная настройка числа потоков/процессов, retries и кеширования позволит повышать отказоустойчивость и эффективность систем, работающих в реальных производственных условиях.

В целом, результаты проведённого исследования создают основу для более глубокого изучения и совершенствования подходов к асинхронному программированию в Python, а также помогают формировать практические рекомендации для реальных проектов.

# Приложения

## Приложение 1. Синхронная реализация

import requests

import time

import psutil

import json

import os

# Путь к файлу для записи результатов

RESULTS\_FILE = 'app/sync\_results.json'

# URL API списка университетов Российской Федерации

API\_URL = 'http://universities.hipolabs.com/search?country=Russian+Federation'

# Количество повторений запроса для тестирования

NUM\_REQUESTS = 100

def fetch\_universities():

"""

Отправляет синхронный HTTP-запрос к API и возвращает статус выполнения.

"""

try:

response = requests.get(API\_URL, timeout=10)

response.raise\_for\_status()

return True

except requests.RequestException as e:

print(f"Ошибка при запросе к API: {e}")

return False

def perform\_test():

"""

Выполняет серию синхронных запросов и собирает метрики производительности.

"""

results = {

'total\_requests': NUM\_REQUESTS,

'successful\_requests': 0,

'failed\_requests': 0,

'average\_time\_per\_request': 0.0,

'cpu\_usage\_percent': 0.0,

'memory\_usage\_mb': 0.0,

'total\_elapsed\_time': 0.0

}

# Получаем текущие значения CPU и памяти до теста

process = psutil.Process(os.getpid())

cpu\_times\_before = process.cpu\_times()

total\_cpu\_time\_before = cpu\_times\_before.user + cpu\_times\_before.system

mem\_before = process.memory\_info().rss / (1024 \* 1024) # в МБ

start\_time = time.time()

for i in range(NUM\_REQUESTS):

success = fetch\_universities()

if success:

results['successful\_requests'] += 1

else:

results['failed\_requests'] += 1

print(f"Запрос {i+1}/{NUM\_REQUESTS} выполнен.")

end\_time = time.time()

# Получаем текущие значения CPU и памяти после теста

cpu\_times\_after = process.cpu\_times()

total\_cpu\_time\_after = cpu\_times\_after.user + cpu\_times\_after.system

mem\_after = process.memory\_info().rss / (1024 \* 1024) # в МБ

# Вычисляем разницу в CPU времени и общее время выполнения

delta\_cpu\_time = total\_cpu\_time\_after - total\_cpu\_time\_before

delta\_wall\_time = end\_time - start\_time

# Вычисляем процент использования CPU

cpu\_usage\_percent = (delta\_cpu\_time / delta\_wall\_time) \* 100

# Обработка результатов

results['total\_elapsed\_time'] = delta\_wall\_time

results['average\_time\_per\_request'] = delta\_wall\_time / NUM\_REQUESTS

results['cpu\_usage\_percent'] = cpu\_usage\_percent

results['memory\_usage\_mb'] = mem\_after - mem\_before

return results

def save\_results(results):

"""

Сохраняет результаты тестирования в JSON-файл.

"""

try:

with open(RESULTS\_FILE, 'w', encoding='utf-8') as f:

json.dump(results, f, ensure\_ascii=False, indent=4)

print(f"Результаты тестирования сохранены в файл {RESULTS\_FILE}.")

except IOError as e:

print(f"Ошибка при сохранении результатов: {e}")

def main():

print("Начало синхронного тестирования...")

results = perform\_test()

print("Тестирование завершено.")

print(json.dumps(results, indent=4, ensure\_ascii=False))

save\_results(results)

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Приложение 2. Асинхронная реализация

import asyncio

import aiohttp

import time

import psutil

import json

import os

# Путь к файлу для записи результатов

RESULTS\_FILE = 'app/asyncio\_results.json'

# URL API списка университетов Российской Федерации

API\_URL = 'http://universities.hipolabs.com/search?country=Russian+Federation'

# Количество повторений запроса для тестирования

NUM\_REQUESTS = 100

# Максимальное количество одновременных запросов

MAX\_CONCURRENT\_REQUESTS = 10

async def fetch\_universities(session, request\_number):

"""

Отправляет асинхронный HTTP-запрос к API и возвращает статус выполнения.

"""

try:

async with session.get(API\_URL, timeout=10) as response:

response.raise\_for\_status()

await response.json()

print(f"Запрос {request\_number}/{NUM\_REQUESTS} выполнен.")

return True

except aiohttp.ClientError as e:

print(f"Ошибка при запросе {request\_number}: {e}")

return False

except asyncio.TimeoutError:

print(f"Тайм-аут запроса {request\_number}.")

return False

async def perform\_test():

"""

Выполняет серию асинхронных запросов и собирает метрики производительности.

"""

results = {

'total\_requests': NUM\_REQUESTS,

'successful\_requests': 0,

'failed\_requests': 0,

'average\_time\_per\_request': 0.0,

'cpu\_usage\_percent': 0.0,

'memory\_usage\_mb': 0.0,

'total\_elapsed\_time': 0.0

}

# Получаем текущие значения CPU и памяти до теста

process = psutil.Process(os.getpid())

cpu\_times\_before = process.cpu\_times()

total\_cpu\_time\_before = cpu\_times\_before.user + cpu\_times\_before.system

mem\_before = process.memory\_info().rss / (1024 \* 1024) # в МБ

start\_time = time.time()

connector = aiohttp.TCPConnector(limit=MAX\_CONCURRENT\_REQUESTS)

async with aiohttp.ClientSession(connector=connector) as session:

tasks = [

asyncio.create\_task(fetch\_universities(session, i + 1))

for i in range(NUM\_REQUESTS)

]

responses = await asyncio.gather(\*tasks)

end\_time = time.time()

# Получаем текущие значения CPU и памяти после теста

cpu\_times\_after = process.cpu\_times()

total\_cpu\_time\_after = cpu\_times\_after.user + cpu\_times\_after.system

mem\_after = process.memory\_info().rss / (1024 \* 1024) # в МБ

# Вычисляем разницу в CPU времени и общее время выполнения

delta\_cpu\_time = total\_cpu\_time\_after - total\_cpu\_time\_before

delta\_wall\_time = end\_time - start\_time

# Вычисляем процент использования CPU

cpu\_usage\_percent = (delta\_cpu\_time / delta\_wall\_time) \* 100

# Обработка результатов

results['successful\_requests'] = sum(responses)

results['failed\_requests'] = NUM\_REQUESTS - results['successful\_requests']

results['total\_elapsed\_time'] = delta\_wall\_time

results['average\_time\_per\_request'] = delta\_wall\_time / NUM\_REQUESTS

results['cpu\_usage\_percent'] = cpu\_usage\_percent

results['memory\_usage\_mb'] = mem\_after - mem\_before

return results

def save\_results(results):

"""

Сохраняет результаты тестирования в JSON-файл.

"""

try:

with open(RESULTS\_FILE, 'w', encoding='utf-8') as f:

json.dump(results, f, ensure\_ascii=False, indent=4)

print(f"Результаты тестирования сохранены в файл {RESULTS\_FILE}.")

except IOError as e:

print(f"Ошибка при сохранении результатов: {e}")

def main():

print("Начало асинхронного тестирования с использованием asyncio...")

results = asyncio.run(perform\_test())

print("Тестирование завершено.")

print(json.dumps(results, indent=4, ensure\_ascii=False))

save\_results(results)

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Приложение 3. Реализация с использованием threading

import requests

import time

import psutil

import json

import os

from concurrent.futures import ThreadPoolExecutor, as\_completed

import threading

import random

# Монкипатчинг для подавления ошибок в threading python 3.13

def safe\_del(self):

pass

if hasattr(threading, '\_DeleteDummyThreadOnDel'):

threading.\_DeleteDummyThreadOnDel.\_\_del\_\_ = safe\_del

# Путь к файлу для записи результатов

RESULTS\_FILE = 'app/threading\_results.json'

# URL API списка университетов Российской Федерации

API\_URL = 'http://universities.hipolabs.com/search?country=Russian+Federation'

# Количество повторений запроса для тестирования

NUM\_REQUESTS = 100

# Максимальное количество одновременных потоков

MAX\_THREADS = 10

# Счетчики успешных и неудачных запросов

successful\_requests = 0

failed\_requests = 0

lock = threading.Lock()

def fetch\_universities(request\_number, max\_retries=3):

"""

Отправляет синхронный HTTP-запрос к API и обновляет счетчики успеха/неудачи.

Добавляет повторные попытки в случае неудачи.

"""

global successful\_requests, failed\_requests

for attempt in range(1, max\_retries + 1):

try:

response = requests.get(API\_URL, timeout=10)

response.raise\_for\_status()

with lock:

successful\_requests += 1

print(f"Запрос {request\_number}/{NUM\_REQUESTS} выполнен.")

return True

except requests.RequestException as e:

print(f"Ошибка при запросе {request\_number} (попытка {attempt}): {e}")

if attempt < max\_retries:

sleep\_time = 2 \*\* attempt + random.uniform(0, 1)

print(f"Повторная попытка через {sleep\_time:.2f} секунд...")

time.sleep(sleep\_time)

with lock:

failed\_requests += 1

print(f"Запрос {request\_number}/{NUM\_REQUESTS} не удался после {max\_retries} попыток.")

return False

def perform\_test():

"""

Выполняет серию многопоточных запросов и собирает метрики производительности.

"""

global successful\_requests, failed\_requests

results = {

'total\_requests': NUM\_REQUESTS,

'successful\_requests': 0,

'failed\_requests': 0,

'average\_time\_per\_request': 0.0,

'cpu\_usage\_percent': 0.0,

'memory\_usage\_mb': 0.0,

'total\_elapsed\_time': 0.0

}

# Получаем текущие значения CPU и памяти до теста

process = psutil.Process(os.getpid())

cpu\_times\_before = process.cpu\_times()

total\_cpu\_time\_before = cpu\_times\_before.user + cpu\_times\_before.system

mem\_before = process.memory\_info().rss / (1024 \* 1024) # в МБ

start\_time = time.time()

with ThreadPoolExecutor(max\_workers=MAX\_THREADS) as executor:

futures = [executor.submit(fetch\_universities, i + 1) for i in range(NUM\_REQUESTS)]

for future in as\_completed(futures):

pass

executor.shutdown(wait=True)

end\_time = time.time()

# Получаем текущие значения CPU и памяти после теста

cpu\_times\_after = process.cpu\_times()

total\_cpu\_time\_after = cpu\_times\_after.user + cpu\_times\_after.system

mem\_after = process.memory\_info().rss / (1024 \* 1024) # в МБ

# Вычисляем разницу в CPU времени и общее время выполнения

delta\_cpu\_time = total\_cpu\_time\_after - total\_cpu\_time\_before

delta\_wall\_time = end\_time - start\_time

# Вычисляем процент использования CPU

cpu\_usage\_percent = (delta\_cpu\_time / delta\_wall\_time) \* 100

# Обработка результатов

results['successful\_requests'] = successful\_requests

results['failed\_requests'] = failed\_requests

results['total\_elapsed\_time'] = delta\_wall\_time

results['average\_time\_per\_request'] = delta\_wall\_time / NUM\_REQUESTS

results['cpu\_usage\_percent'] = cpu\_usage\_percent

results['memory\_usage\_mb'] = mem\_after - mem\_before

return results

def save\_results(results):

"""

Сохраняет результаты тестирования в JSON-файл.

"""

try:

with open(RESULTS\_FILE, 'w', encoding='utf-8') as f:

json.dump(results, f, ensure\_ascii=False, indent=4)

print(f"Результаты тестирования сохранены в файл {RESULTS\_FILE}.")

except IOError as e:

print(f"Ошибка при сохранении результатов: {e}")

def main():

print("Начало многопоточного тестирования с использованием ThreadPoolExecutor...")

results = perform\_test()

print("Тестирование завершено.")

print(json.dumps(results, indent=4, ensure\_ascii=False))

save\_results(results)

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Приложение 4. Реализация с использованием multiprocessing

import requests

import time

import psutil

import json

import os

from multiprocessing import Pool, Manager

import random

# Путь к файлу для записи результатов

RESULTS\_FILE = 'app/multiprocessing\_results.json'

# URL API списка университетов Российской Федерации

API\_URL = 'http://universities.hipolabs.com/search?country=Russian+Federation'

# Общее число запросов

NUM\_REQUESTS = 100

# Максимальное число одновременно работающих процессов

MAX\_PROCESSES = 10

# Сколько раз в целом можно пытаться заново запустить неудачные запросы

TOTAL\_MAX\_RETRIES = 5

def fetch\_universities(args):

"""

Делает до max\_retries локальных попыток отправить запрос.

Если ни одна не удалась, возвращает False.

При успехе возвращает True.

"""

request\_number, max\_retries, counter = args

for attempt in range(1, max\_retries + 1):

try:

response = requests.get(API\_URL, timeout=5)

response.raise\_for\_status()

with counter['lock']:

counter['successful\_requests'] += 1

print(f"[Запрос #{request\_number}] Успех на попытке {attempt}.")

return True

except requests.RequestException as e:

print(f"[Запрос #{request\_number}] Ошибка (попытка {attempt}): {e}")

if attempt < max\_retries:

sleep\_time = 0.5 + random.uniform(0, 0.5)

print(f"Повтор через {sleep\_time:.2f} с...")

time.sleep(sleep\_time)

with counter['lock']:

counter['failed\_requests'] += 1

print(f"[Запрос #{request\_number}] Не удался после {max\_retries} локальных попыток.")

return False

def perform\_test():

"""

Механизм:

1) Имеем список из 100 запросов (1..100).

2) В каждой итерации создаём пул, обрабатываем только текущий список неудачных.

3) Если остались неуспешные - снова пытаемся (до TOTAL\_MAX\_RETRIES итераций).

"""

manager = Manager()

shared\_counter = manager.dict({

'successful\_requests': 0,

'failed\_requests': 0,

'lock': manager.Lock()

})

results = {

'total\_requests': NUM\_REQUESTS,

'successful\_requests': 0,

'failed\_requests': 0,

'average\_time\_per\_request': 0.0,

'cpu\_usage\_percent': 0.0,

'memory\_usage\_mb': 0.0,

'total\_elapsed\_time': 0.0

}

process = psutil.Process(os.getpid())

cpu\_before = process.cpu\_times()

total\_cpu\_time\_before = cpu\_before.user + cpu\_before.system

mem\_before = process.memory\_info().rss / (1024 \* 1024)

start\_time = time.time()

all\_requests = list(range(1, NUM\_REQUESTS + 1))

current\_retry = 0

while all\_requests and current\_retry < TOTAL\_MAX\_RETRIES:

print(f"\nИтерация {current\_retry+1}/{TOTAL\_MAX\_RETRIES}, запросов осталось: {len(all\_requests)}")

success\_before\_iter = shared\_counter['successful\_requests']

fail\_before\_iter = shared\_counter['failed\_requests']

args = [(req\_num, 3, shared\_counter) for req\_num in all\_requests]

with Pool(processes=MAX\_PROCESSES) as pool:

results\_map = pool.map(fetch\_universities, args)

iteration\_new\_success = shared\_counter['successful\_requests'] - success\_before\_iter

iteration\_new\_fail = shared\_counter['failed\_requests'] - fail\_before\_iter

print(f"Итерация {current\_retry+1} завершена. Новых успехов: {iteration\_new\_success}, новых неудач: {iteration\_new\_fail}.")

all\_requests = [args[i][0] for i, success in enumerate(results\_map) if not success]

current\_retry += 1

end\_time = time.time()

cpu\_after = process.cpu\_times()

total\_cpu\_time\_after = cpu\_after.user + cpu\_after.system

mem\_after = process.memory\_info().rss / (1024 \* 1024)

delta\_cpu\_time = total\_cpu\_time\_after - total\_cpu\_time\_before

delta\_wall\_time = end\_time - start\_time

cpu\_usage\_percent = (delta\_cpu\_time / delta\_wall\_time) \* 100

# Итоговые результаты

results['successful\_requests'] = shared\_counter['successful\_requests']

results['failed\_requests'] = NUM\_REQUESTS - shared\_counter['successful\_requests']

results['total\_elapsed\_time'] = delta\_wall\_time

results['average\_time\_per\_request'] = delta\_wall\_time / NUM\_REQUESTS

results['cpu\_usage\_percent'] = cpu\_usage\_percent

results['memory\_usage\_mb'] = mem\_after - mem\_before

return results

def save\_results(results):

try:

with open(RESULTS\_FILE, 'w', encoding='utf-8') as f:

json.dump(results, f, ensure\_ascii=False, indent=4)

print(f"Результаты тестирования сохранены в {RESULTS\_FILE}.")

except IOError as e:

print(f"Ошибка при сохранении результатов: {e}")

def main():

print("Начало многопроцессного тестирования...")

results = perform\_test()

print("Тестирование завершено.")

print(json.dumps(results, indent=4, ensure\_ascii=False))

save\_results(results)

if \_\_name\_\_ == "\_\_main\_\_":

main()