Лекция 6. Подходы к параллельному программированию

**Оглавление**

[**Введение** 3](#_Toc188258871)

[**1.** **Понятие GIL** 5](#_Toc188258872)

[**2.** **Многопоточность** 8](#_Toc188258873)

[**3.** **Многопроцессорность** 13](#_Toc188258874)

[**4.** **Асинхронное программирование** 16](#_Toc188258875)

[**Заключение** 19](#_Toc188258876)

[**Список литературы** 20](#_Toc188258877)

**Понятие класса и объекта класса**

**Цель:** формирование знаний по основам параллельного программирования на Python, принципам и областям применения каждого подхода.

**План лекции:**Введение.1. Понятие GIL.  
2. Многопоточность.  
3. Многопроцессорность.4. Асинхронное программирование.Заключение.Список литературы.

# **Введение**

Современная разработка программного обеспечения требует создания эффективных приложений, способных справляться с высокими нагрузками и выполнять множество операций одновременно. Программы, способные выполнять несколько задач параллельно или асинхронно, становятся неотъемлемой частью таких областей, как веб-разработка, обработка больших данных, научные вычисления и автоматизация процессов.

Python, будучи универсальным языком программирования, предлагает три основные модели для работы с параллелизмом и конкурентностью:

1. **Многопоточность**: Механизм, позволяющий выполнять несколько потоков (threads) в рамках одного процесса. Подходит для задач, связанных с вводом-выводом (I/O), но ограничивается Global Interpreter Lock (GIL), который не позволяет потокам одновременно выполнять Python-байт-код на разных ядрах процессора.
2. **Многопроцессорность**: Использование нескольких процессов, каждый из которых имеет собственную копию интерпретатора Python и памяти. Этот подход обходит ограничения GIL и идеально подходит для ресурсоёмких операций, нагружающих процессор.
3. **Асинхронное программирование**: Метод, основанный на неблокирующем выполнении задач с использованием корутин и цикла событий. Асинхронное программирование эффективно для задач, связанных с большим количеством операций ввода-вывода, таких как сетевые запросы или работа с базами данных.

Каждый из этих подходов имеет свои преимущества, ограничения и области применения. Выбор правильной модели зависит от характера задачи, требуемой производительности и структуры приложения. Например, многопоточность удобна для быстрого прототипирования и простых I/O операций, многопроцессорность подходит для параллельных вычислений, а асинхронное программирование — для высоконагруженных сетевых приложений.

В этой статье мы подробно рассмотрим каждый из этих подходов, их реализацию в Python, основные инструменты и примеры использования. Вы узнаете, как правильно выбрать подход для решения ваших задач и как максимально эффективно использовать возможности Python для работы с параллельными и асинхронными процессами.

# **Понятие GIL**

GIL (Global Interpreter Lock, глобальная блокировка интерпретатора) — это механизм, используемый в интерпретаторе CPython, который обеспечивает безопасность выполнения потоков, но одновременно ограничивает параллельное выполнение потоков в многопоточном Python-коде. Понимание GIL важно для правильного использования многопоточности и многопроцессорности в Python.

#### **Что такое GIL**

GIL — это глобальная блокировка, которая позволяет только одному потоку выполнять Python-байт-код в любой момент времени. Это означает, что даже если на вашей машине несколько ядер процессора, потоки Python не смогут одновременно выполнять код на разных ядрах.

GIL был введён для упрощения работы с управлением памятью в CPython. Благодаря этому механизму разработчикам интерпретатора не нужно было внедрять сложные механизмы синхронизации для объектов, так как GIL предотвращает одновременный доступ потоков к общей памяти.

#### **Как работает GIL**

1. **Выделение потока:** GIL блокирует выполнение всех других потоков, пока один поток выполняет Python-байт-код.
2. **Ожидание завершения:** Поток освобождает GIL, когда:
   * Выполняет ввод-вывод (I/O), например, чтение из файла или сети.
   * Явно вызывается блокировка, например, time.sleep().
3. **Переключение контекста:** GIL периодически освобождается для выполнения других потоков. По умолчанию это происходит после определённого числа операций ("счётчик переключений контекста").

#### **Преимущества GIL**

1. **Простота реализации:** Упрощает работу интерпретатора CPython, особенно в части управления памятью.
2. **Совместимость с расширениями:** Многие библиотеки, написанные на C, рассчитывают на GIL, чтобы избежать сложной синхронизации при доступе к объектам Python.
3. **Безопасность потоков:** GIL предотвращает конфликты при одновременном доступе к данным из нескольких потоков.

#### **Ограничения GIL**

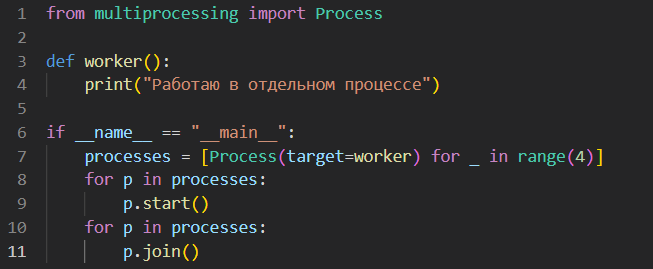
1. **Отсутствие параллелизма:** Потоки в Python не могут одновременно использовать несколько ядер процессора для выполнения байт-кода. Это делает многопоточность неэффективной для CPU-зависимых задач.
2. **Снижение производительности:** GIL может стать узким местом в многопоточных приложениях, особенно если задачи требуют интенсивных вычислений.
3. **Неэффективное использование ресурсов:** На многопроцессорных системах GIL ограничивает использование доступных ядер.

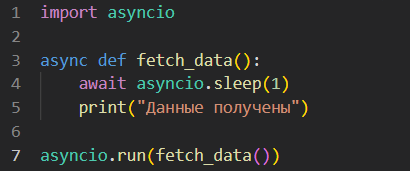
#### **Когда GIL не мешает**

GIL не является значительным ограничением для:

* **I/O-зависимых задач:** Ввод-вывод освобождает GIL, что позволяет другим потокам продолжать выполнение. Например, работа с файлами, сетевыми запросами и базами данных.
* **Однопоточных программ:** Программы, которые не используют многопоточность, не ощущают влияния GIL.
* **Расширений на C:** Библиотеки, такие как NumPy, выполняют вычисления на C-уровне, освобождая GIL.

#### **Как обойти ограничения GIL**

* **Многопроцессорность:** Использование модуля multiprocessing позволяет запускать несколько процессов, каждый из которых имеет собственный интерпретатор Python и, следовательно, собственный GIL. 
* **Асинхронное программирование:** Использование asyncio для задач, связанных с вводом-выводом.



* **Расширения на C:** Использование библиотек, которые освобождают GIL для выполнения интенсивных вычислений, таких как NumPy или SciPy.
* **Переписывание на другие языки:** Для вычислительно сложных задач можно использовать более подходящие языки, такие как C++, Java или Go.

#### **Будущее GIL**

В сообществе Python ведутся активные обсуждения о замене или удалении GIL. Например, существуют проекты, такие как PyPy (альтернативный интерпретатор Python), которые используют другой подход к управлению памятью и не имеют GIL. Также разрабатываются патчи для CPython, чтобы устранить эту блокировку, но это требует значительных изменений в экосистеме Python.

#### **Итог**

GIL — это компромисс между простотой реализации интерпретатора и ограничениями производительности в многопоточных приложениях. Понимание его работы помогает разработчикам выбирать правильные инструменты и подходы для решения задач. Если ваша программа требует высокой производительности и параллелизма, вы можете использовать многопроцессорность, асинхронное программирование или сторонние библиотеки, обходящие ограничения GIL.

# **Многопоточность (Multitreading)**

Многопоточность — это механизм, который позволяет программе выполнять несколько потоков выполнения (threads) одновременно. Потоки — это независимые последовательности выполнения инструкций, которые делят общую память и ресурсы процесса.

#### **Что такое многопоточность**

В Python многопоточность предоставляется с помощью модуля threading. Поток представляет собой более легковесную единицу выполнения, чем процесс, и позволяет реализовать параллельность в рамках одного процесса. Однако из-за GIL (Global Interpreter Lock) в Python потоки не могут одновременно выполнять байт-код Python. GIL ограничивает производительность многопоточных программ в CPU-зависимых задачах. Тем не менее многопоточность остаётся полезной для задач, связанных с вводом-выводом (I/O), таких как обработка сетевых запросов или операций с файлами.

#### **Где применяется многопоточность**

1. **Веб-разработка**: Для одновременной обработки множества запросов. Например, веб-серверы могут использовать потоки для обслуживания нескольких клиентов.
2. **Сетевое программирование**: Обработка входящих и исходящих данных через сеть.
3. **Чтение и запись данных**: Обработка больших файлов, запросы к базам данных или веб-скрейпинг.
4. **Графический интерфейс пользователя (GUI)**: Обеспечение отзывчивости интерфейса при выполнении долгих операций.

#### **Как применяется многопоточность**

Для создания и управления потоками в Python используется модуль threading. Основные этапы работы с потоками:

1. Создание потока.
2. Запуск потока.
3. Синхронизация потоков, если это необходимо.

#### **Основные классы и методы модуля threading:**

* *threading.Thread*: Основной класс для создания потоков.
* *start*(): Запускает выполнение потока.
* *join*(): Блокирует выполнение основного потока, пока поток-объект не завершится.
* *Lock*: Используется для синхронизации доступа к общим ресурсам.

#### **Примеры использования**

##### Пример 1: Простое создание потока

##### Пример 2: Использование Lock для синхронизации потоков

##### Пример 3: Многопоточность для сетевых запросов



#### **Ограничения многопоточности в Python**

1. **GIL (Global Interpreter Lock)**: Из-за этого ограничения настоящая параллельность невозможна для задач, нагружающих процессор. В таких случаях рекомендуется использовать модуль *multiprocessing*.
2. **Сложность отладки**: Потокобезопасность и синхронизация могут усложнять разработку.
3. **Проблемы производительности**: Избыточное создание потоков может привести к накладным расходам.

#### **Альтернативы многопоточности**

* **Многопроцессорность**: Модуль *multiprocessing* позволяет использовать несколько процессов для обхода GIL.
* **Асинхронное программирование**: Использование модуля *asyncio* для работы с корутинами, обеспечивающими эффективную обработку I/O задач.

#### **Итог**

Многопоточность в Python — это мощный инструмент для повышения производительности и создания более отзывчивых приложений. Однако из-за ограничений GIL её использование эффективно в основном для задач ввода-вывода. Для CPU-зависимых задач лучше рассмотреть многопроцессорность или асинхронное программирование.

# **Многопроцессорность**

Многопроцессорность — это механизм, позволяющий программе запускать несколько процессов одновременно, что обеспечивает настоящую параллельность выполнения. В отличие от потоков, каждый процесс имеет собственное пространство памяти и не ограничен GIL (Global Interpreter Lock), что делает его идеальным для задач, нагружающих процессор.

#### **Что такое многопроцессорность**

В Python модуль multiprocessing предоставляет инструменты для создания и управления процессами. Каждый процесс выполняется независимо и имеет собственную память. Для взаимодействия между процессами используются очереди (*Queues*), каналы (*Pipes*) и другие механизмы межпроцессного взаимодействия (*IPC*).

#### **Где применяется многопроцессорность**

1. **Научные вычисления**: Обработка больших массивов данных или выполнение сложных математических расчетов.
2. **Машинное обучение**: Распараллеливание обучения моделей и обработки данных.
3. **Рендеринг и обработка изображений**: Ускорение обработки больших графических данных.
4. **Автоматизация и обработка данных**: Эффективное выполнение ресурсоемких задач, таких как обработка файлов, шифрование и компиляция.

#### **Как применяется многопроцессорность**

Для работы с процессами в Python используются классы и методы из модуля *multiprocessing*. Основные шаги включают:

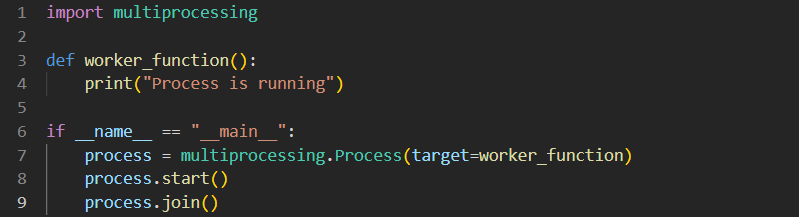
1. Создание и запуск процессов.
2. Обмен данными между процессами.
3. Синхронизацию процессов, если это необходимо.

#### **Основные классы и методы модуля multiprocessing:**

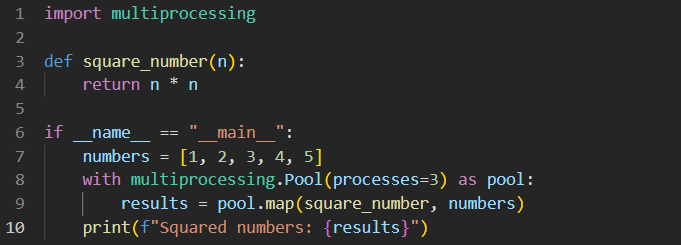
* *multiprocessing.Process*: Основной класс для создания процессов.
* *start*(): Запускает выполнение процесса.
* *join*(): Ожидает завершения процесса.
* *Pool*: Предоставляет удобный интерфейс для работы с пулом процессов.
* *Queue* и *Pipe*: Используются для обмена данными между процессами.

#### **Примеры использования**

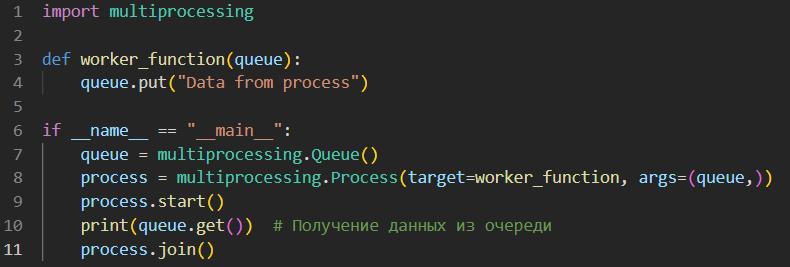
##### Пример 1: Простое создание и запуск процесса



##### Пример 2: Использование пула процессов



##### Пример 3: Обмен данными между процессами с использованием Queue



#### **Преимущества многопроцессорности**

1. **Настоящая параллельность**: Отсутствие ограничений GIL позволяет задействовать все ядра процессора.
2. **Изоляция процессов**: Каждый процесс имеет собственное пространство памяти, что снижает вероятность конфликтов.
3. **Увеличение производительности**: Особенно для задач, нагружающих CPU.

#### **Ограничения многопроцессорности**

1. **Высокие накладные расходы**: Создание процессов требует больше ресурсов, чем потоков.
2. **Сложность обмена данными**: Поскольку память не общая, взаимодействие между процессами требует дополнительного кодирования.
3. **Платформенные различия**: Некоторые функции модуля *multiprocessing* работают по-разному на Windows и Unix.

#### **Альтернативы многопроцессорности**

* **Многопоточность**: Для задач ввода-вывода, где высокая производительность CPU не требуется.
* **Асинхронное программирование**: Модуль *asyncio* обеспечивает эффективную обработку задач с ожиданием.

#### **Итог**

Многопроцессорность в Python — это мощный инструмент для выполнения вычислительно сложных задач с использованием всех доступных ресурсов процессора. Хотя она требует большего внимания к синхронизации и обмену данными, её преимущества делают её незаменимой для задач, где требуется высокая производительность.

# **Асинхронное программирование**

Асинхронное программирование — это подход к разработке программного обеспечения, позволяющий выполнять операции, не блокируя основной поток выполнения. Этот метод особенно эффективен для задач, связанных с вводом-выводом (I/O), таких как обработка сетевых запросов, работы с файлами или базами данных.

#### **Что такое асинхронное программирование**

В Python асинхронное программирование реализуется с помощью модуля asyncio, который предоставляет инструменты для создания и управления асинхронными задачами (coroutines). Основное преимущество подхода — возможность обрабатывать множество операций одновременно без необходимости создания потоков или процессов. Вместо этого используется цикл событий (event loop), который переключается между задачами, пока они ожидают выполнения внешних операций.

#### **Где применяется асинхронное программирование**

1. **Веб-разработка**: Асинхронные веб-фреймворки, такие как FastAPI или AIOHTTP, обрабатывают множество запросов одновременно, эффективно используя ресурсы.
2. **Сетевое программирование**: Асинхронные библиотеки для работы с HTTP, WebSocket или сокетами.
3. **Обработка файлов**: Чтение и запись больших объёмов данных без блокировки основного выполнения программы.
4. **Работа с базами данных**: Асинхронные драйверы для взаимодействия с базами данных.
5. **Автоматизация и веб-скрейпинг**: Эффективное выполнение множества запросов к веб-сайтам.

#### **Как применяется асинхронное программирование**

Асинхронное программирование строится вокруг ключевых концепций:

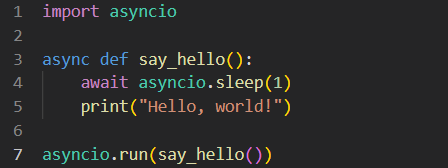
1. **Корутины (coroutines)**: Асинхронные функции, определяемые с помощью ключевого слова ***async def***.
2. **Цикл событий (event loop)**: Координирует выполнение корутин.
3. **Задачи (tasks)**: Корутины, которые были запущены в цикле событий.

#### **Основные функции и классы модуля asyncio:**

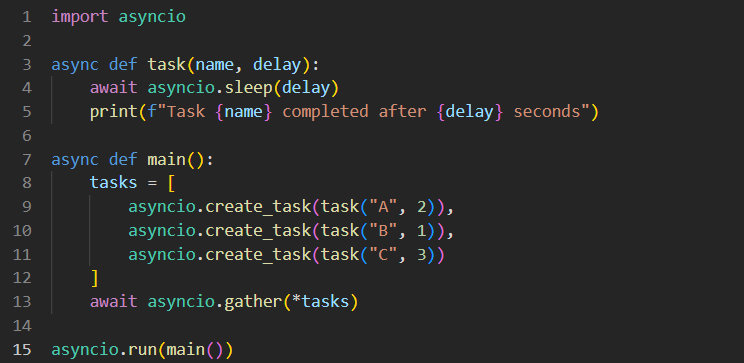
* *asyncio.run*(): Запускает асинхронную программу.
* *await*: Приостанавливает выполнение текущей корутины до завершения другой асинхронной операции.
* *asyncio.create\_task*(): Создаёт задачу для выполнения корутины.
* *asyncio.gather*(): Запускает несколько корутин одновременно и ожидает их завершения.

#### **Примеры использования**

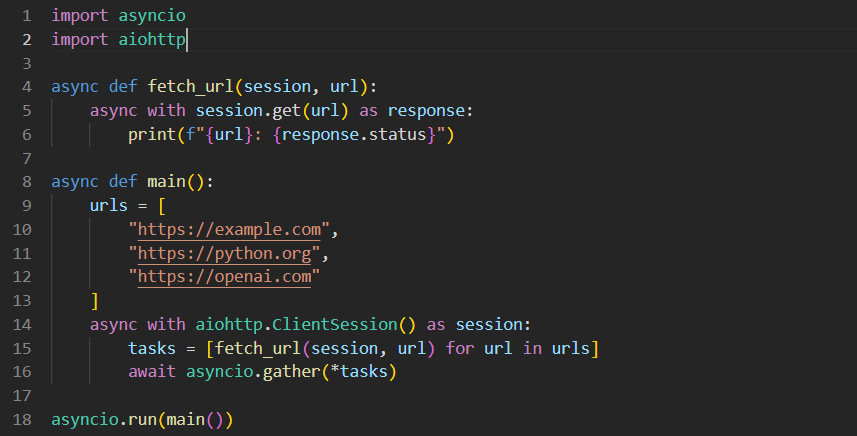
##### Пример 1: Простая асинхронная функция



##### Пример 2: Запуск нескольких задач одновременно



##### Пример 3: Асинхронная работа с HTTP запросами



#### **Преимущества асинхронного программирования**

1. **Высокая производительность**: Эффективное использование ресурсов для I/O задач.
2. **Уменьшение накладных расходов**: Нет необходимости создавать множество потоков или процессов.
3. **Простота управления**: Использование *await* делает код более читаемым по сравнению с обратными вызовами (callbacks).

#### **Ограничения асинхронного программирования**

1. **Сложность отладки**: Асинхронный код может быть сложнее для понимания и отладки.
2. **Требования к библиотекам**: Асинхронное программирование требует использования специализированных библиотек (например, *aiohttp* вместо *requests*).
3. **Необходимость обучения**: Разработка асинхронного кода требует изучения новых концепций и синтаксиса.

#### **Альтернативы асинхронному программированию**

* **Многопоточность**: Для задач, которые требуют совместного использования памяти.
* **Многопроцессорность**: Для ресурсоемких операций, нагружающих процессор.

#### **Итог**

Асинхронное программирование в Python позволяет эффективно обрабатывать множество операций, связанных с вводом-выводом, минимизируя время простоя. Несмотря на то, что освоение подхода может потребовать времени, его преимущества делают его важным инструментом для современных разработчиков.

# **Заключение**

Параллельное программирование — это мощный инструмент для повышения производительности приложений, особенно в условиях современного высоконагруженного мира, где обработка данных и выполнение сложных операций требует максимальной эффективности. Python предлагает три основных подхода к параллелизму: многопоточность, многопроцессорность и асинхронное программирование. Каждый из этих методов имеет свои преимущества, недостатки и оптимальные области применения.

**Многопоточность** является подходом для обработки задач, которые сильно зависят от операций ввода-вывода. Благодаря простоте реализации и поддержке стандартной библиотекой Python, она позволяет быстро организовать обработку сетевых запросов, работу с файлами или взаимодействие с базами данных. Однако её ограничения, вызванные GIL (Global Interpreter Lock), делают многопоточность малоэффективной для вычислительно сложных задач.

**Многопроцессорность** решает проблему GIL, предоставляя возможность использовать несколько процессов вместо потоков. Каждый процесс выполняется независимо, что позволяет задействовать все доступные ядра процессора. Этот подход отлично подходит для ресурсоёмких задач, таких как машинное обучение, научные вычисления и обработка больших данных. Основным недостатком является больший расход памяти и накладные расходы на создание процессов.

**Асинхронное программирование** предлагает уникальный способ обработки задач с минимальными ресурсными затратами. Используя неблокирующий ввод-вывод и цикл событий, оно идеально подходит для создания высокопроизводительных сетевых приложений, веб-серверов и микросервисов. Хотя асинхронный подход требует изучения новых концепций и библиотек, таких как asyncio, его преимущества делают его незаменимым для современных разработчиков.

Каждый из подходов имеет своё место в экосистеме Python, и выбор зависит от характера задачи. Например, если приложение должно эффективно обрабатывать множество входящих сетевых запросов, асинхронное программирование будет наилучшим выбором. Для интенсивных вычислений подойдут многопроцессорные решения, а для простых сценариев ввода-вывода — многопоточность.

В результате, комбинируя эти подходы или выбирая наиболее подходящий инструмент для конкретной задачи, вы сможете создавать приложения, которые эффективно используют ресурсы системы и отвечают современным требованиям производительности.

# **Список литературы**

1. П.Берри. Изучаем программирование на Python. Глава 113/4.
2. https://en.wikipedia.org/wiki/Global\_interpreter\_lock
3. https://habr.com/ru/companies/sberbank/articles/829098/