

**Некоммерческое образовательное учреждение
Учебно-научно-производственный комплекс
«Международный Университет Кыргызстана»
Среднее профессиональное образование**

«Nomad» Колледж



Тротченко Андрей Александрович

КУРСОВАЯ РАБОТА

На тему: «Исследование многозадачности с использованием threading и multiprocessing»

Руководитель курсовой работы:
преп.

_____ Уралбек уулу С.

“ ____ ” _____ 2024 г.

Бишкек, 2024 г.

Содержание:

Введение	3
Цель курсовой работы	4
ГЛАВА 1. Что такое многозадачность в Python и на чем основана?	4
1.1 Многозадачность с использованием Threading	5
1.2 Многозадачность с использованием Multiprocessing	6
1.3 Управление и взаимодействие между задачами	6
1.4 Главные качества и сферы применения	7
ГЛАВА 2. Что такое threading в Python?	9
2.1 Средства синхронизации	10
ГЛАВА 3. Что такое multiprocessing в python?	13
3.1 Основные компоненты модуля multiprocessing	13
3.2 Применения multiprocessing	14
3.3 Отличительные свойства threading от multiprocessing	16
Заключение	19
Список используемой литературы	21
Приложение	22

Введение

В современном обществе информационные технологии стали неотъемлемой частью нашей повседневной жизни и связаны с самыми разными сферами деятельности. От общения и развлечений до науки и бизнеса - ИТ проникают во все аспекты нашей деятельности, улучшая и ускоряя процессы. В этом мире программирование является ключевым элементом, позволяющим создавать приложения, веб-сервисы и инструменты, составляющие основу цифровой эпохи и много всего другого.

Среди различных языков программирования Python выделяется своей универсальностью, читаемостью кода и большим сообществом разработчиков. Его простота и гибкость делают его идеальным инструментом для разработки широкого спектра проектов.

В данной курсовой работе рассматривается одна из ключевых особенностей Python - многозадачность, которая оказывает значительное влияние на производительность приложений.

Для этого анализируется способность Python реализовывать параллельные вычисления с помощью потоков и многопроцессорности. Далее в работе рассматриваются принципы многопроцессорной обработки, преимущества и ограничения каждого метода, а также анализируется их эффективность в различных сценариях.

Эффективное использование многозадачности в Python важно, когда приложениям необходимо обрабатывать несколько задач одновременно для обеспечения высокой скорости отклика и производительности.

Потоки и многопроцессорность - два основных подхода к решению этой проблемы в Python.

Цель курсовой работы

Целью данной курсовой работы является изучение того, как язык программирования Python выполняет многозадачные задачи с использованием многопоточности и многопроцессорной обработки. Основной упор делается на выявление преимуществ, ограничений и эффективности каждого из подходов в различных сценариях разработки приложений.

ГЛАВА 1. Что такое многозадачность в Python и на чем основана?

Многозадачность (multitasking) — это возможность одновременной работы с несколькими задачами. Она основана на двух принципах: потоки (threading) и процессы (multiprocessing).

Потоки (threading) — это логические потоки выполнения, которые выполняют одну и ту же задачу, но с разными данными. В отличие от процессов, потоки используют общее пространство памяти и обмениваются данными между собой напрямую. Смотреть рис. 1

Процессы (multiprocessing) — это полностью независимые процессы, которые выполняют свою задачу. В отличие от потоков, процессы не используют общее пространство памяти и обмениваются данными через интерфейс ввода-вывода (stdin, stdout, stderr). Смотреть рис. 2

Где используется многозадачность в Питоне:

1. Обработка изображений и видео (OpenCV, PIL).
2. Машинное обучение и анализ данных (Scikit-learn, TensorFlow).
3. Веб-сервисы и веб-приложения, требующие больших вычислительных ресурсов (Flask, Django).

Когда нужно использовать threading, а когда multiprocessing?

Потоки могут быть более подходящими, если задачи требуют много времени на выполнение, так как процессы занимают больше ресурсов операционной системы и медленнее создаются и уничтожаются.

Процессы могут быть более подходящими, если задачи должны выполняться в отдельных пространствах памяти, так как они не используют общее пространство памяти и могут общаться только через интерфейс ввода-вывода.

Однако следует учесть, что в большинстве случаев для работы с большими массивами данных или параллельным выполнением задач следует использовать процессы, так как они могут лучше использовать многоядерные процессоры.

Помимо того, потоки и процессы могут иметь различные ограничения в использовании общей памяти и других ресурсов, что также может влиять на выбор между ними.

1.1 Многозадачность с использованием Threading

Потоки (Threads): Threading в Python предоставляет легковесные потоки выполнения в пределах одного процесса. Каждый поток имеет свой собственный стек и выполняет независимый код.

Совместное использование ресурсов: Потоки в одном процессе совместно используют ресурсы, такие как память, что может привести к более эффективному использованию ресурсов.

Общий процессор: Потоки внутри одного процесса выполняются на общем процессоре, что подходит для задач, где множество операций может быть выполнено параллельно.

1.2 Многозадачность с использованием Multiprocessing

Процессы (Processes): Multiprocessing предоставляет возможность создания отдельных процессов с собственными областями памяти. Каждый процесс работает независимо друг от друга.

Параллельная обработка: Процессы в Multiprocessing могут выполняться параллельно, что особенно полезно на многоядерных системах. Каждый процесс имеет свой собственный интерпретатор Python.

Изоляция данных: из-за изолированных областей памяти процессы предоставляют более высокую степень защиты данных и избегают проблем с разделяемыми ресурсами.

1.3 Управление и взаимодействие между задачами

Locks и Semaphores: Threading предоставляет механизмы синхронизации, такие как блокировки (locks) и семафоры, для избежания одновременного доступа нескольких потоков к общим ресурсам.

Queue и Pipe: Multiprocessing предоставляет инструменты для обмена данными между процессами, такие как Queue и Pipe (Функция pipe создает канал между двумя процессами и возвращает два дескриптора файла. (Дескриптор файла — это целое число без знака, с помощью которого процесс обращается к открытому файлу.)), что упрощает взаимодействие и передачу данных.

Daemon Threads/Processes: Threading и Multiprocessing позволяют создавать фоновые потоки (Daemon Threads – потоки-демоны чаще всего запускаются во время загрузки системы. С технической точки зрения демоном считается процесс, который не имеет интерфейса для управления.)/процессы, которые завершают свою работу, когда основной поток/процесс завершается.

Управление приоритетами: Оба подхода предоставляют средства управления приоритетами выполнения потоков и процессов для более гибкой настройки работы приложений.

Thread/Process States: Threading и Multiprocessing предоставляют функции для отслеживания состояния выполнения потоков и процессов, что облегчает мониторинг и отладку.

1.4 Главные качества и сферы применения

Параллелизм: Одной из ключевых особенностей многозадачности Python является возможность запускать задачи параллельно или одновременно. Многопоточность и multiprocessing предоставляют инструменты для эффективного использования вычислительных ресурсов.

Использование многоядерных систем: Multiprocessing позволяет распараллеливать задачи в разных процессах. Это особенно важно для эффективного использования многоядерных систем и повышения общей производительности.

Улучшение реакции приложения: потоки помогают выполнять фоновые задачи, такие как загрузка данных из сети и обновление пользовательского интерфейса, тем самым делая приложение более отзывчивым.

Изоляция и безопасность данных: Multiprocessing обеспечивает отдельные области памяти для каждого процесса, обеспечивая более высокий уровень безопасности данных, особенно при обработке конфиденциальных данных.

Веб-разработка: Многопоточность и multiprocessing могут ускорить обработку запросов на стороне сервера в веб-приложениях и обеспечить параллельную обработку нескольких запросов.

Научные исследования: В областях научных исследований, требующих обработки больших объемов данных, многозадачность может значительно ускорить вычисления.

Обработка и анализ данных: Многопоточность и многопроцессорность обычно используются при обработке и анализе данных, чтобы обеспечить эффективную обработку больших объемов информации.

Искусственный интеллект и машинное обучение: В сфере искусственного интеллекта и машинного обучения, требующей обучения на больших объемах данных, многозадачность может значительно ускорить процесс обучения модели.

Игровая индустрия: Разработка компьютерных игр часто требует одновременной обработки нескольких задач, а многозадачность можно использовать для повышения производительности и скорости реагирования игровых приложений.

Сетевое программирование: В сетевом программировании, где приложение может иметь большое количество соединений, многозадачность позволяет обрабатывать запросы параллельно, что приводит к более эффективной работе.

Гибкость и универсальность многозадачности Python делает его незаменимым во многих областях программирования, где требуется эффективное выполнение параллельных задач.

ГЛАВА 2. Что такое threading в Python?

Модуль `threading` является одним из способов реализации многопоточности в Python. Это означает, что с его помощью вы можете выполнять несколько операций одновременно, используя разные потоки. Каждый поток выполняет независимую задачу, и все они могут выполняться параллельно.

В Python для работы с потоками используется встроенный модуль `threading`, который предоставляет классы и функции для создания и управления потоками.

Для создания нового потока вам потребуется импортировать модуль `threading` и использовать его класс `Thread`. Класс `Thread` принимает два основных аргумента: `target` — функция, которую нужно выполнить в новом потоке, и `args` — кортеж аргументов, передаваемых этой функции. Пример создания и запуска потока. Смотреть рис. 3

Чтобы дождаться завершения потока, вы можете использовать метод `join()`. Это полезно, когда вам нужно убедиться, что все потоки выполнили свою работу перед тем, как продолжить выполнение основной программы. Пример использования метода `join()`. Смотреть рис. 4

При работе с многопоточностью важно учитывать, что потоки могут одновременно обращаться к одной и той же глобальной переменной. В таких случаях возможны «гонки» между потоками, поэтому рекомендуется использовать механизмы синхронизации, такие как блокировки (`Lock`). Пример использования блокировки при работе с глобальной переменной. Смотреть рис. 5

`Semaphore` и другие механизмы синхронизации: кроме блокировок, `threading` предоставляет другие механизмы синхронизации, такие как семафоры, условия и мьютексы.

2.1 Средства синхронизации

Синхронизация потоков в Python. Это механизм синхронизации потоков, который гарантирует, что никакие два потока не могут одновременно выполнять определенный сегмент внутри программы для доступа к общим ресурсам. Ситуацию можно назвать критическими участками. Мы используем состояние гонки, чтобы избежать состояния критического раздела, когда два потока не обращаются к ресурсам одновременно.

Семафор — это объект, который контролирует доступ к общему ресурсу в многозадачной среде. Он поддерживает операции захвата (acquire) и освобождения (release), что позволяет управлять доступом нескольких потоков к общему ресурсу.

Семафор можно рассматривать как переменную, отражающую количество существующих в настоящее время ресурсов. Например, есть несколько слотов, доступных на определенном уровне на стоянке торгового центра, который является семафором.

Семафоры используются для контроля доступа к ограниченному числу ресурсов. Они представляют собой объекты, имеющие счетчик, который уменьшается при каждом захвате ресурса и увеличивается при его освобождении.

Класс Semaphore состоит из конструктора и двух функций, Acquire() и Release() соответственно.

Функция Acquire() используется для уменьшения счетчика семафора в случае, если счетчик больше нуля. В противном случае он блокируется, пока счетчик не станет больше нуля.

Функция release() используется для увеличения счетчика семафора и пробуждения одного из потоков, ожидающих семафор.

В приведенном рис. 6 `object_name` является объектом класса `Semaphore`. Параметр `count` – это количество потоков, которым разрешен одновременный доступ. Значение этого параметра по умолчанию – 1.

Всякий раз, когда функция `Acquire()` выполняется потоком, значение параметра «`count`» будет уменьшаться на единицу. Каждый раз, когда функция `release()` выполняется потоком, значение параметра «`count`» увеличивается на единицу. Этот оператор подразумевает, что всякий раз, когда мы вызываем метод `Acqua()`, значение параметра «`count`» будет уменьшаться, тогда как при вызове метода `release()` значение параметра «`count`» будет увеличиваться. Пример использования смотреть рис. 7.

Мы импортировали необходимые модули и создали объект для класса `Semaphore` со значением счетчика 4. Мы определили функцию, используя для этого объекта функцию `Acquire()`. Затем мы использовали цикл `for` для перебора значения до 6. Затем мы вызвали функцию `release()` и создали несколько потоков. Наконец, мы вызвали потоки с помощью функции `start()`.

Условие — это объект, который предоставляет механизм для ожидания событий и их оповещения внутри многозадачной программы. Оно используется для синхронизации потоков в различных сценариях. Пример использования смотреть рис. 8.

Примечание: методы `.wait()` и `.notify_all()` не снимают блокировку. Это означает, что пробужденный поток или потоки не вернутся из своего вызова `.wait()` сразу, а только тогда, когда поток, который вызвал `.notify()` или `.notify_all()` окончательно снимет/откажется от блокировки.

Программирование с использованием переменных условий использует блокировку для синхронизации доступа к некоторому общему состоянию. Потоки, которые заинтересованы в конкретном изменении этого состояния, повторно вызывают метод `.wait()`, пока не увидят желаемое состояние, в то

время как потоки, которые изменяют состояние, вызывают методы `.notify()` или `.notify_all()`, когда изменяют состояние таким образом, чтобы оно могло быть приемлемо для одного из ждущих потоков. Например, следующий код представляет собой общую ситуацию производитель/потребитель с неограниченной емкостью буфера. Смотреть рис. 9.

Проверка цикла `while` на условие необходима приложению, потому что метод `.wait()` может возвращать результат через произвольно долгое время, а условие, вызвавшее метод `.notify()`, может больше не выполниться. Это присуще многопоточному программированию. Метод `.wait_for()` может использоваться для автоматизации проверки условий и упрощает вычисление тайм-аутов. Смотреть рис. 10.

Чтобы выбрать между использованием методов `.notify()` и `.notify_all()`, необходимо понять, может ли одно изменение состояния быть интересным только для одного или нескольких ожидающих потоков. Например, в типичной ситуации производитель/потребитель, добавление одного элемента в буфер обмена требует пробуждения только одного потока-потребителя.

Мьютекс — это объект, который обеспечивает эксклюзивный доступ к общему ресурсу. Только один поток может владеть мьютексом в любой момент времени. `Lock` (мьютекс) обеспечивает эксклюзивный доступ к общему ресурсу, чтобы избежать конфликтов доступа.

Мьютекс (от англ. «**M**U**T**ual **E**X**cl**usion») — это специальный вид семафора, используемый для предотвращения одновременного доступа к критическому ресурсу. В отличие от семафора, мьютекс имеет только два состояния: заблокирован и разблокирован. Пример использования смотреть рис. 11.

ГЛАВА 3. Что такое multiprocessing в python?

Модуль multiprocessing в Python предоставляет средства для параллельного выполнения кода с использованием многозадачности на уровне процессов. Этот модуль создает отдельные процессы, в каждом из которых выполняется код, что позволяет использовать многозадачность на нескольких ядрах процессора.

3.1 Основные компоненты модуля multiprocessing

Класс Process: Этот класс используется для создания объектов процессов. Каждый объект Process представляет отдельный процесс, в котором может выполняться некоторый код. Пример создания и запуска процесса смотреть рис. 12.

Обмен данными между процессами: Модуль предоставляет несколько способов обмена данными между процессами, включая разделяемые объекты (Value, Array), механизмы межпроцессного взаимодействия (IPC) через очереди (Queue) и трубы (Pipe), а также разделяемую память (shared_memory). Смотреть рис. 13.

Пул процессов (класс Pool): Pool предоставляет удобный интерфейс для распределения задач по нескольким процессам и сбора результатов. Смотреть рис. 14.

Использование Manager: Класс Manager предоставляет интерфейс для управления общими объектами и данными между процессами. Смотреть рис. 15.

Создание подпроцессов: multiprocessing также позволяет создавать подпроцессы, которые могут быть использованы для выполнения задач внутри основных процессов. Подпроцесс относится к дополнительным процессам, созданным из основного процесса. Подпроцесс представляет собой отдельное выполнение кода, независимое от основного процесса и других подпроцессов.

Подпроцессы позволяют организовать параллельное выполнение задач, ускоряя общий процесс выполнения программы. Смотреть рис. 16.

Использование Value и Array: multiprocessing предоставляет классы Value и Array для создания разделяемых объектов (числа, массивы) между процессами.

Модуль multiprocessing предоставляет обширные возможности для организации параллельного выполнения кода в Python, позволяя эффективно использовать несколько процессорных ядер и решать задачи, которые требуют параллельного выполнения. Смотреть рис. 17.

3.2 Применения multiprocessing

Распределение вычислительных задач: используется для параллельного выполнения вычислительных задач на нескольких процессорных ядрах.

Обработка данных: эффективно применяется при обработке больших объемов данных, когда можно разбить задачу на независимые части и обработать их параллельно.

Создание многозадачных приложений: помогает в создании многозадачных приложений, где каждая задача выполняется в отдельном процессе, что обеспечивает изоляцию и стабильность.

Параллельное выполнение тестов: широко используется в тестировании для параллельного выполнения тестов, что может ускорить процесс тестирования.

Использование нескольких ядер процессора: отлично подходит для использования нескольких ядер процессора и улучшения производительности при решении задач, требующих вычислительных ресурсов.

Создание демонов: multiprocessing может использоваться для создания демонов, которые выполняются в фоновом режиме, обрабатывая определенные задачи.

Избегание GIL: используется для выполнения задач, избегая ограничений Global Interpreter Lock (GIL) в Python.

Обработка ввода/вывода (I/O)-интенсивных задач: при работе с операциями ввода/вывода, такими как чтение/запись файлов, запросы к базам данных или работа с сетевыми операциями, использование multiprocessing может улучшить производительность программы, так как процессы могут эффективно ждать завершения операций I/O (Input/Output).

Распределение задач в сети: Модуль multiprocessing может быть использован для распределения задач на различные узлы сети, где каждый узел выполняет задачи параллельно. Это полезно, например, в сценариях, где есть несколько компьютеров, способных выполнять задачи.

Выполнение параллельных вычислений: применяется для параллельного выполнения вычислительных задач, таких как сложные математические вычисления или обработка больших объемов данных. Процессы могут эффективно использовать несколько ядер процессора.

Параллельное выполнение тестов: В тестировании программного обеспечения может использоваться для параллельного выполнения тестов, что может значительно ускорить процесс тестирования.

Асинхронная обработка: В сценариях, где нужна асинхронная обработка задач, multiprocessing может быть использован для создания асинхронных воркеров, которые выполняют задачи параллельно.

Решение задач с высоким уровнем параллелизма: В случае, когда задачи могут быть разделены на множество независимых частей, multiprocessing

позволяет эффективно использовать многозадачность на уровне процессов, особенно при наличии нескольких ядер процессора.

Создание параллельных алгоритмов: В разработке параллельных алгоритмов, например, в области машинного обучения или обработки сигналов, где можно разделить задачу на подзадачи, каждую из которых обрабатывает свой процесс.

Создание параллельных веб-серверов: может использоваться для создания веб-серверов, способных обрабатывать несколько запросов одновременно.

Параллельная обработка событий: В приложениях, где обработка различных событий требует значительных вычислительных ресурсов, multiprocessing может помочь в обработке событий параллельно.

Параллельное выполнение фоновых задач: Модуль multiprocessing можно использовать для создания фоновых задач, которые выполняются параллельно с основным процессом и не прерывают его работу.

3.3 Отличительные свойства threading от multiprocessing

Threading и multiprocessing — это два различных подхода к параллельному программированию в Python, и каждый из них имеет свои отличительные особенности. Вот некоторые из ключевых различий между ними:

Модель выполнения: threading использует многозадачность на уровне потоков внутри одного процесса. Однако из-за Global Interpreter Lock (GIL) в CPython, только один поток может выполнять байт-код Python в один момент времени, что может ограничивать эффективность использования нескольких потоков для CPU-интенсивных задач.

Multiprocessing использует многозадачность на уровне процессов, каждый процесс имеет свой отдельный интерпретатор Python и собственный GIL. Это позволяет использовать несколько процессорных ядер и обойти ограничения GIL, делая multiprocessing более подходящим для CPU-интенсивных задач.

Создание и управление: В threading создание и управление потоками более легкое и менее затратное с точки зрения ресурсов. В multiprocessing создание и управление процессами более затратное, так как каждый процесс имеет свой собственный интерпретатор Python и память.

Обмен данными между потоками/процессами: В threading потоки могут легко обмениваться данными, поскольку они разделяют общее пространство памяти. В multiprocessing процессы имеют собственное пространство памяти, и обмен данными между процессами может быть более сложным. Для этого могут использоваться механизмы, такие как разделяемые объекты (Value, Array) или механизмы межпроцессного взаимодействия (IPC) как очереди (Queue), трубы (Pipe) и разделяемая память (shared_memory).

Стойкость к сбоям: Из-за общего пространства памяти в threading, сбой одного потока может повлиять на другие потоки в том же процессе. В multiprocessing каждый процесс работает в своем собственном адресном пространстве, поэтому сбой одного процесса обычно не влияет на другие процессы.

Кроссплатформенность: Оба подхода (многозадачность на уровне потоков и на уровне процессов) кроссплатформенны и могут использоваться на различных операционных системах.

Оперативная память: из-за того, что потоки в threading используют общее пространство памяти, они могут эффективно обмениваться данными.

Однако, это также означает, что необходимо бережно использовать синхронизацию для избежания конфликтов при доступе к данным.

В multiprocessing каждый процесс имеет свою собственную область памяти, что устраняет проблемы с GIL и обеспечивает более безопасное параллельное выполнение. Однако обмен данными между процессами может быть менее эффективным.

Сериализация объектов: при использовании multiprocessing, объекты передаются между процессами путем сериализации и десериализации. Это может потребовать, чтобы объекты были сериализуемыми. В threading объекты передаются по общему адресному пространству и не требуют сериализации.

Сложность программирования: В threading часто проще программировать, поскольку потоки совместно используют общее пространство памяти и могут легко обмениваться данными. В multiprocessing программирование может быть сложнее из-за необходимости управления процессами, обмена данными между процессами и решения проблем, связанных с параллельным выполнением.

Производительность: для некоторых видов задач threading может быть быстрее из-за меньшей накладной работы при создании потоков.

В случае CPU-интенсивных задач multiprocessing может предоставить лучшую производительность из-за возможности использования нескольких процессорных ядер. Выбор между threading и multiprocessing зависит от конкретных требований вашей задачи и особенностей вашего приложения.

Заключение

В ходе выполнения курсовой работы было проведено подробное исследование возможностей модулей `threading` и `multiprocessing` в языке программирования Python с целью изучения многозадачности. В процессе работы были выделены следующие ключевые моменты:

Threading: Модуль `threading` предоставляет удобные средства для создания легковесных потоков.

Особенности GIL (Global Interpreter Lock) влияют на эффективность параллельного выполнения кода в нескольких потоках, что делает этот подход более пригодным для I/O-интенсивных задач.

Multiprocessing: Модуль `multiprocessing` предоставляет механизмы для создания отдельных процессов, обходя ограничения GIL. Эффективен для CPU-интенсивных задач, позволяя параллельное выполнение кода на нескольких ядрах процессора.

Применение: `Threading` оказывается полезным для асинхронных операций, обработки ввода/вывода и выполнения блокирующих операций. `Multiprocessing` применяется в случаях, требующих параллельного выполнения CPU-интенсивных задач, таких как обработка данных, вычислительные задачи и другие.

Рекомендации: при выборе между `threading` и `multiprocessing` рекомендуется учитывать характер задачи и требования к производительности.

Для I/O-интенсивных задач, где не блокируется GIL, `threading` может быть удобным в использовании.

Для CPU-интенсивных задач, требующих эффективного использования нескольких ядер процессора, multiprocessing представляется более подходящим вариантом.

Общий вывод: Исследование многозадачности в Python с использованием модулей threading и multiprocessing позволило лучше понять, как эти инструменты могут быть применены в различных сценариях.

Результаты исследования позволяют разработчикам принимать обоснованные решения при выборе подходящего механизма параллельного выполнения в зависимости от задачи и характеристик системы.

Исследование многозадачности в Python расширило понимание работы потоков и процессов, что является важным вкладом в область разработки программного обеспечения и оптимизации производительности приложений.

Список используемой литературы

1. <https://pythonpip.ru/osnovy/semafor-v-python-sinhronizatsiya-potokov-i-protssessov>
2. <https://sky.pro/media/kak-rabotat-s-modulem-threading-v-python/>
3. <https://docs-python.ru/standart-library/modul-threading-python/>
4. [https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_\(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5\)#cite_note-:8-1](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-:8-1)
5. <https://docs-python.ru/standart-library/modul-threading-python/klass-condition-modulja-threading/>
6. <https://docs-python.ru/standart-library/modul-threading-python/klass-condition-modulja-threading/>
7. <https://sky.pro/media/kak-rabotat-s-semaforami-i-myuteksami-v-python/>
8. <https://habr.com/ru/companies/otus/articles/769448/>
9. <https://habr.com/ru/articles/164325/>
10. <https://pythonstart.ru/osnovy/mnogopotochnost-i-mnogoprotsessornost-v-python>

Ссылка на GitHub:

<https://github.com/Kompanis/Kurovaya/tree/master>

Приложение

```
1  import threading
2  import time
3
4  def print_numbers():
5      for i in range(10):
6          print(i)
7          time.sleep(1)
8
9  def print_letters():
10     for i in range(65, 75):
11         print(chr(i))
12         time.sleep(1)
13
14  t1 = threading.Thread(target=print_numbers)
15  t2 = threading.Thread(target=print_letters)
16
17  t1.start()
18  t2.start()
19
20  t1.join()
21  t2.join()
22
```

Рис. 1

import threading

import time

def print_numbers():

for i in range(10):

print(i)

time.sleep(1)

def print_letters():

for i in range(65, 75):

print(chr(i))

time.sleep(1)

```

if __name__ == "__main__":

    t1 = threading.Thread(target=print_numbers)

    t2 = threading.Thread(target=print_letters)

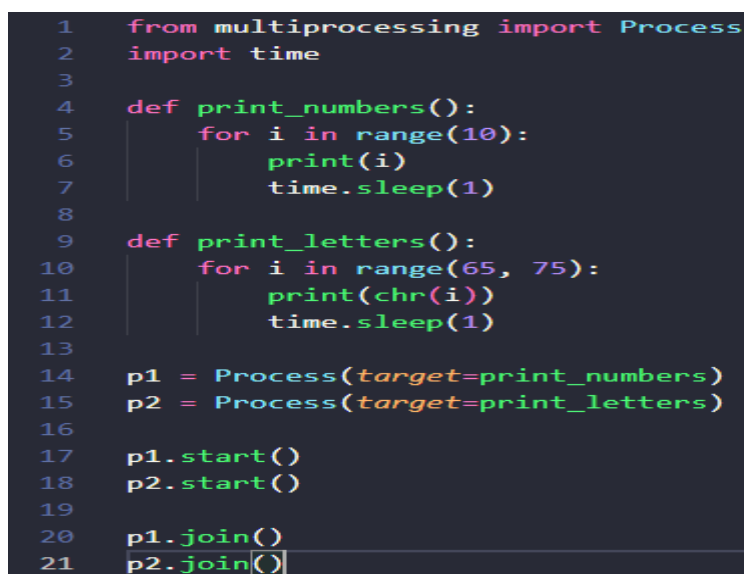

    t1.start()

    t2.start()


    t1.join()

    t2.join()

```



```

1  from multiprocessing import Process
2  import time
3
4  def print_numbers():
5      for i in range(10):
6          print(i)
7          time.sleep(1)
8
9  def print_letters():
10     for i in range(65, 75):
11         print(chr(i))
12         time.sleep(1)
13
14     p1 = Process(target=print_numbers)
15     p2 = Process(target=print_letters)
16
17     p1.start()
18     p2.start()
19
20     p1.join()
21     p2.join()

```

Рис. 2

```

from multiprocessing import Process

import time


def print_numbers():

```

```

for i in range(10):

    print(i)

    time.sleep(1)

def print_letters():

    for i in range(65, 75):

        print(chr(i))

        time.sleep(1)

if __name__ == "__main__":

    p1 = Process(target=print_numbers)

    p2 = Process(target=print_letters)

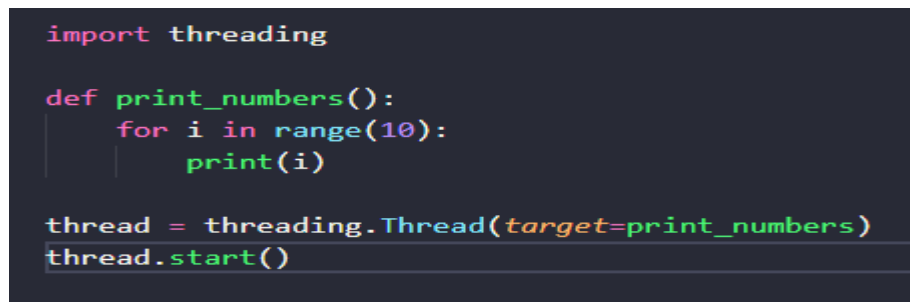
    p1.start()

    p2.start()

    p1.join()

    p2.join()

```



```

import threading

def print_numbers():
    for i in range(10):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()

```

Рис. 3


```
import threading
```

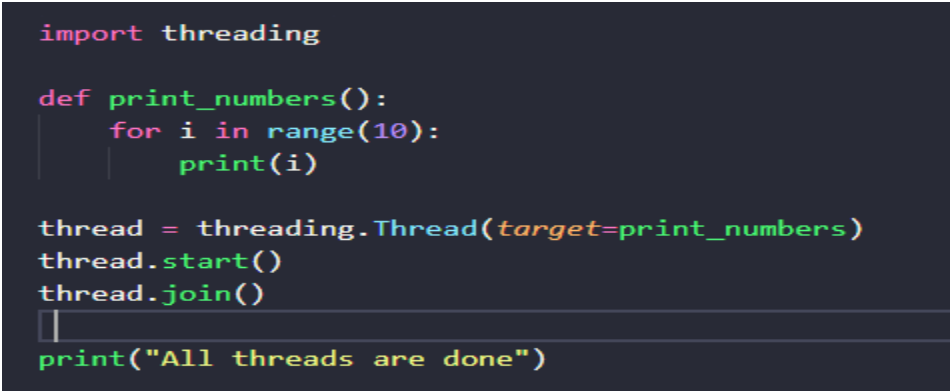
```
def print_numbers():
```

```
    for i in range(10):
```

```
        print(i)
```

```
thread = threading.Thread(target=print_numbers)
```

```
thread.start()
```

A screenshot of a code editor with a dark background. The code is written in a light-colored font with syntax highlighting. It shows the same Python code as the previous blocks, but with an additional line: `print("All threads are done")` at the end. The code is as follows:

```
import threading

def print_numbers():
    for i in range(10):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()
thread.join()

print("All threads are done")
```

Рис. 4

```
import threading
```

```
def print_numbers():
```

```
    for i in range(10):
```

```
        print(i)
```

```
thread = threading.Thread(target=print_numbers)
```

```
thread.start()
```

```
thread.join()
```

```
print("All threads are done")
```

```

import threading

counter = 0
lock = threading.Lock()

def increment_counter():
    global counter
    with lock:
        for _ in range(10000):
            counter += 1

threads = [threading.Thread(target=increment_counter) for _ in range(10)]

for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print(f"Final counter value: {counter}")

```

Рис. 5

```
import threading
```

```
counter = 0
```

```
lock = threading.Lock()
```

```
def increment_counter():
```

```
    global counter
```

```
    with lock:
```

```
        for i in range(10000):
```

```
            counter += 1
```

```
threads = [threading.Thread(target=increment_counter) for i in range(10)]
```

for thread in threads:

 thread.start()

for thread in threads:

 thread.join()

print(f'Final counter value: {counter}')

```
object_name = Semaphore(count)
```

object_name = Semaphore(count)

Рис. 6

```
import threading
import time

def show(name):
    for i in range(6):
        print(f'Javatpoint, {name}')
        time.sleep(1)

my_obj = threading.Semaphore(4)

threads = [threading.Thread(target=show, args=(f'Thread {i}',)) for i in range(6)]

for thread in threads:
    my_obj.acquire()
    thread.start()

for thread in threads:
    thread.join()
```

Рис. 7

import threading

import time

```

def show(name):

    for i in range(6):

        print(f'Javatpoint, {name}")

        time.sleep(1)

my_obj = threading.Semaphore(4)

threads = [threading.Thread(target=show, args=(f'Thread {i}"),) for i in range(6)]

for thread in threads:

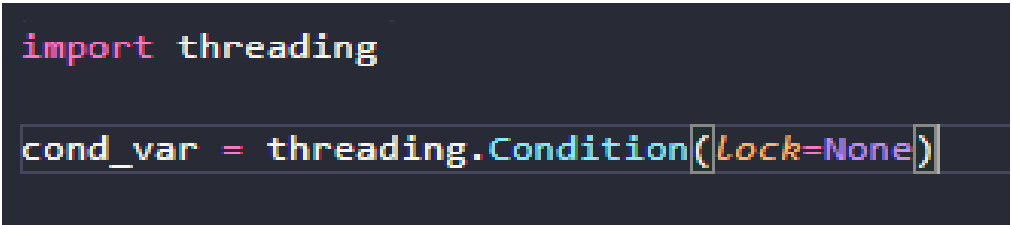
    my_obj.acquire()

    thread.start()

for thread in threads:

    thread.join()

```



```

import threading

cond_var = threading.Condition(lock=None)

```

Рис. 8

```

import threading

cond_var = threading.Condition(lock=None)

```

```

import threading
cond_var = threading.Condition(Lock=None)

# Потребитель порции данных
with cond_var:
    while not an_item_is_available():
        cond_var.wait()
    get_an_available_item()

# Производитель порции данных
with cond_var:
    make_an_item_available()
    cond_var.notify()

```

Рис. 9

```

import threading

cond_var = threading.Condition(Lock=None)

# Потребитель порции данных

with cond_var:

    while not an_item_is_available():

        cond_var.wait()

    get_an_available_item()

# Производитель порции данных

with cond_var:

    make_an_item_available()

    cond_var.notify()

```

```
import threading
cond_var = threading.Condition(lock=None)

# Потребитель порции данных
with cond_var:
    cond_var.wait_for(an item is available)
    get_an_available_item()
```

Рис. 10

```
import threading

cond_var = threading.Condition(lock=None)

# Потребитель порции данных

with cond_var:

    cond_var.wait_for(an_item_is_available)

    get_an_available_item()
```

```

import threading

mutex = threading.Lock()
shared_resource = 0

def access_resource():
    global shared_resource
    print(f'{threading.current_thread().name} ожидает доступ к ресурсу')
    mutex.acquire()
    print(f'{threading.current_thread().name} получил доступ к ресурсу')
    shared_resource += 1
    mutex.release()
    print(f'{threading.current_thread().name} освободил ресурс')

threads = []
for i in range(5):
    thread = threading.Thread(target=access_resource)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print(f'Значение shared_resource: {shared_resource}')

```

Рис. 11

```

import threading

mutex = threading.Lock()

shared_resource = 0

def access_resource():

    global shared_resource

    print(f'{threading.current_thread().name} ожидает доступ к ресурсу')

    mutex.acquire()

    print(f'{threading.current_thread().name} получил доступ к ресурсу')

    shared_resource += 1

    mutex.release()

```

```

print(f'{threading.current_thread().name} освободил ресурс')

threads = []

for i in range(5):

    thread = threading.Thread(target=access_resource)

    threads.append(thread)

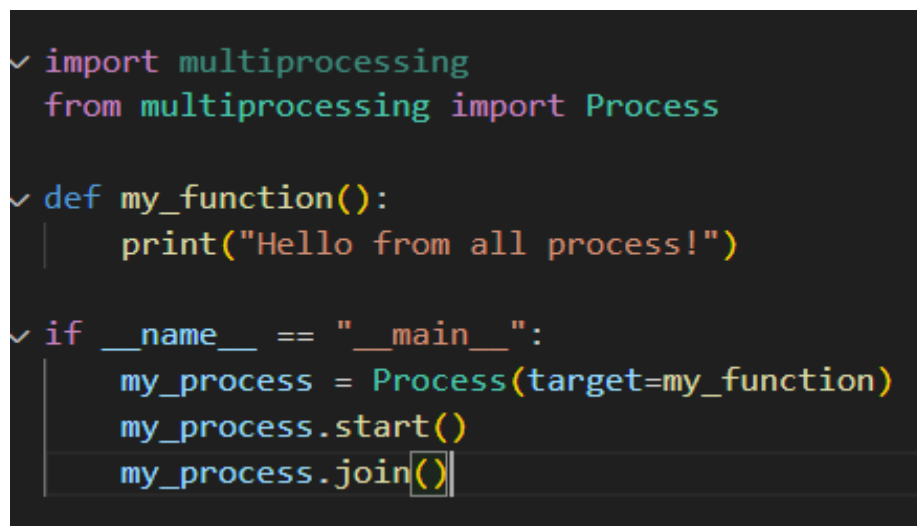
    thread.start()


for thread in threads:

    thread.join()


print(f'Значение shared_resource: {shared_resource}')

```



```

✓ import multiprocessing
  from multiprocessing import Process

✓ def my_function():
  |     print("Hello from all process!")

✓ if __name__ == "__main__":
  |     my_process = Process(target=my_function)
  |     my_process.start()
  |     my_process.join()

```

Рис. 12

```

import multiprocessing

from multiprocessing import Process

```



```
def my_function():

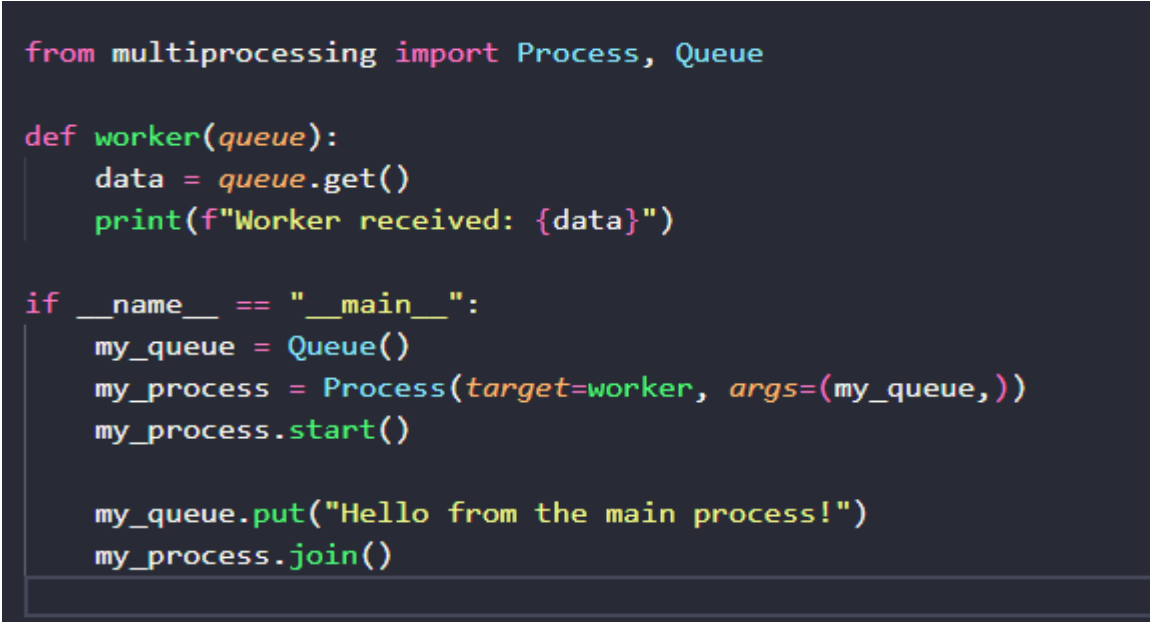
    print("Hello from all process!")

if __name__ == "__main__":

    my_process = Process(target=my_function)

    my_process.start()

    my_process.join()
```



```
from multiprocessing import Process, Queue

def worker(queue):
    data = queue.get()
    print(f"Worker received: {data}")

if __name__ == "__main__":
    my_queue = Queue()
    my_process = Process(target=worker, args=(my_queue,))
    my_process.start()

    my_queue.put("Hello from the main process!")
    my_process.join()
```

Рис. 13

```
from multiprocessing import Process, Queue
```

```
def worker(queue):

    data = queue.get()

    print(f"Worker received: {data}")
```

```
if __name__ == "__main__":
```

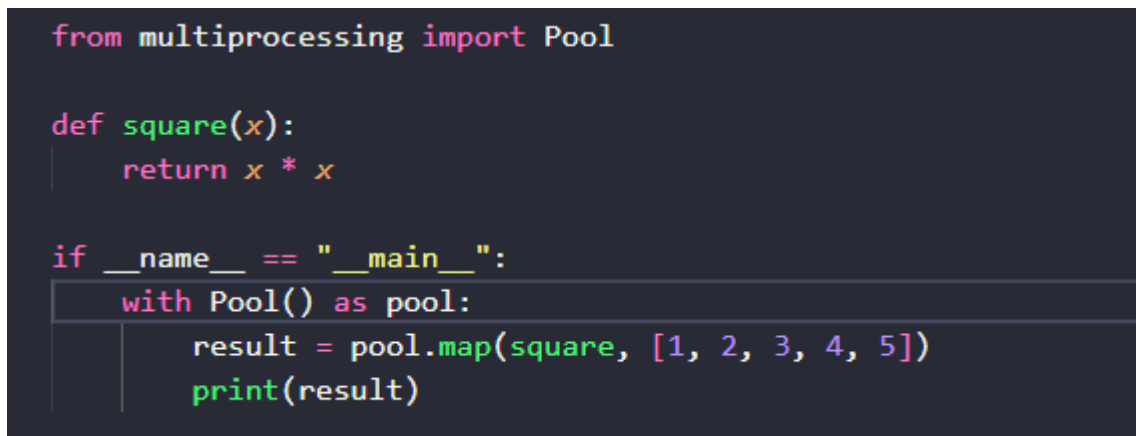
```
my_queue = Queue()

my_process = Process(target=worker, args=(my_queue,))

my_process.start()


my_queue.put("Hello from the main process!")

my_process.join()
```

A screenshot of a code editor with a dark background and light-colored text. The code is a Python script that demonstrates the use of the multiprocessing.Pool class. It imports Pool from multiprocessing, defines a square function, and then uses a Pool object to map the square function over a list of numbers [1, 2, 3, 4, 5]. The result is printed. The code is as follows:

```
from multiprocessing import Pool

def square(x):
    return x * x

if __name__ == "__main__":
    with Pool() as pool:
        result = pool.map(square, [1, 2, 3, 4, 5])
        print(result)
```

Рис. 14

```
from multiprocessing import Pool
```

```
def square(x):
```

```
    return x * x
```

```
if __name__ == "__main__":
```

```
    with Pool() as pool:
```

```
        result = pool.map(square, [1, 2, 3, 4, 5])
```

```
        print(result)
```

```

from multiprocessing import Process, Manager
def worker(shared_list):
    shared_list.append("Hello from worker!")
if __name__ == "__main__":
    with Manager() as manager:
        shared_list = manager.list()
        my_process = Process(target=worker, args=(shared_list,))
        my_process.start()
        my_process.join()
        print(shared_list)

```

Рис. 15

```

from multiprocessing import Process, Manager

def worker(shared_list):

    shared_list.append("Hello from worker!")

if __name__ == "__main__":

    with Manager() as manager:

        shared_list = manager.list()

        my_process = Process(target=worker, args=(shared_list,))

        my_process.start()

        my_process.join()

        print(shared_list)

```

```

from multiprocessing import Process
def worker():
    print("Hello from a subprocess!")
if __name__ == "__main__":
    my_process = Process(target=worker)
    my_process.start()
    my_process.join()

```

Рис. 16

```

from multiprocessing import Process

def worker():

    print("Hello from a subprocess!")

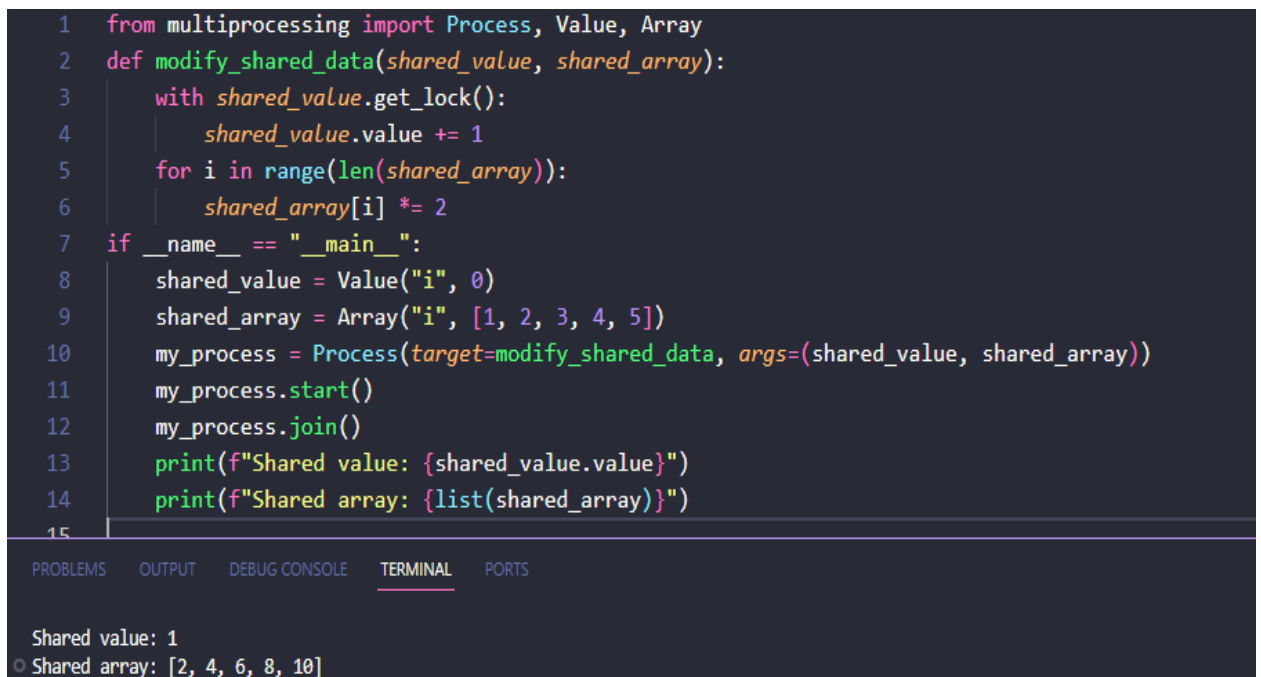
if __name__ == "__main__":

    my_process = Process(target=worker)

    my_process.start()

    my_process.join()

```



The screenshot shows a code editor with a dark background. The code defines a function `modify_shared_data` that uses a lock to increment a shared value and double the elements of a shared array. The main block creates a `Process` object, starts it, joins it, and prints the final state of the shared data. The terminal output at the bottom shows the results: the shared value is 1 and the shared array is [2, 4, 6, 8, 10].

```

1  from multiprocessing import Process, Value, Array
2  def modify_shared_data(shared_value, shared_array):
3      with shared_value.get_lock():
4          shared_value.value += 1
5          for i in range(len(shared_array)):
6              shared_array[i] *= 2
7  if __name__ == "__main__":
8      shared_value = Value("i", 0)
9      shared_array = Array("i", [1, 2, 3, 4, 5])
10     my_process = Process(target=modify_shared_data, args=(shared_value, shared_array))
11     my_process.start()
12     my_process.join()
13     print(f"Shared value: {shared_value.value}")
14     print(f"Shared array: {list(shared_array)}")
15
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
Shared value: 1
Shared array: [2, 4, 6, 8, 10]

```

Рис. 17

```

from multiprocessing import Process, Value, Array

def modify_shared_data(shared_value, shared_array):

    with shared_value.get_lock():

        shared_value.value += 1

    for i in range(len(shared_array)):

```

```
    shared_array[i] *= 2

if __name__ == "__main__":
    shared_value = Value("i", 0)
    shared_array = Array("i", [1, 2, 3, 4, 5])

    my_process = Process(target=modify_shared_data, args=(shared_value,
shared_array))

    my_process.start()

    my_process.join()

    print(f"Shared value: {shared_value.value}")
    print(f"Shared array: {list(shared_array)}")
```