



86.51

Рейтинг

Подписаться

## SimbirSoft

Лидер в разработке современных ИТ-решений на заказ



SSul 23 ноя 2022 в 07:50

# Как работать с процессами и потоками в Python

🕒 16 мин 👁 69K

Блог компании SimbirSoft, Python\*, Программирование\*, Параллельное программирование\*

Раскрывать тему параллельного или асинхронного программирования непросто. Во-первых, она перегружена терминологией и трудна для понимания. Как правило, тонкости и особенности работы с языками усваиваются, лишь когда столкнешься с ними на практике. Во-вторых, в контексте Python тоже много своих подводных камней. Но сегодня почти любой современный web-сервис сталкивается с необходимостью многопоточности или асинхронности. Поскольку это многопользовательская среда, мы хотим направить всю процессорную мощность не на ожидание, а на решение прикладных задач бизнеса, чтобы все пользователи вовремя получили необходимые данные.

Эта статья будет полезна тем разработчикам, которые хотят выполнять больше работы за одно и то же время, и задействовать все ресурсы своего железа. Проще говоря, делать

## ИНФОРМАЦИЯ

**Сайт** [www.simbirsoft.com](http://www.simbirsoft.com)

**Дата регистрации** 2 ноября 2015

**Дата основания** 21 февраля 2001

**Численность** 1 001–5 000 человек

**Местоположение** Россия

## ССЫЛКИ

SimbirSoft  
[www.simbirsoft.com](http://www.simbirsoft.com)

YouTube  
[www.youtube.com](http://www.youtube.com)

Вакансии  
[www.simbirsoft.com](http://www.simbirsoft.com)

## ВКОНТАКТЕ



+16



233



16

SimbirSoft

# Как работать с потоками и процессами в Python

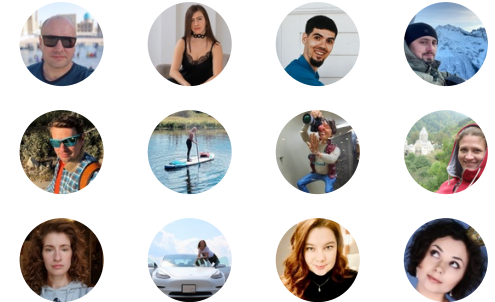


\*Кадр из фильма «Астерикс на Олимпийских играх»



SimbirSoft

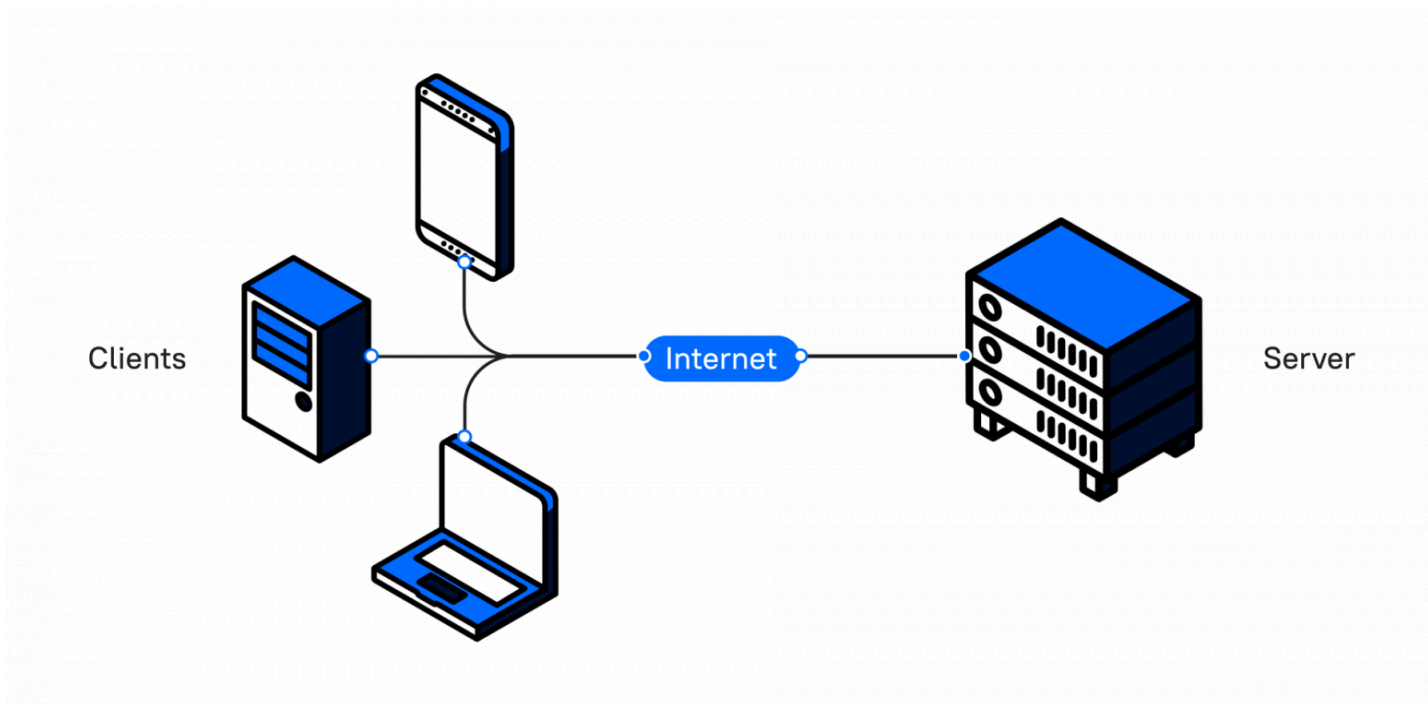
11 500 подписчиков



[Подписаться на новости](#)

ВИДЖЕТ

Давайте возьмем за отправную точку ситуацию, когда у нас есть приложение, которое работает по стандартной схеме **клиент – сервер**:

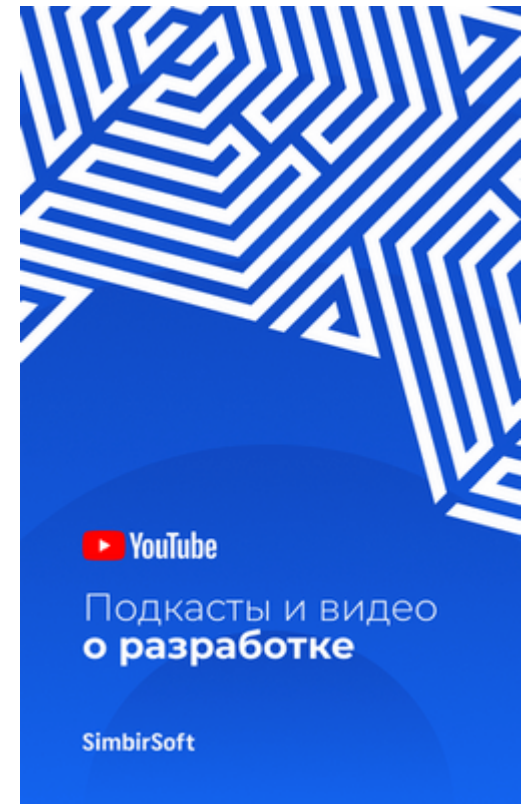


Клиент посылает запрос и получает ответ. А теперь представьте, что в нашем приложении есть кнопка, которая формирует большой отчет. Когда пользователь нажимает на нее, программа долго обрабатывает запрос. Клиент ждет ответа, и пока отчет не будет сформирован, он не сможет пользоваться интерфейсом приложения.

Как мы можем помочь пользователю продолжить взаимодействие с нашим приложением, пока формируется отчет? Мы можем создать отдельный процесс, отдельный поток, и выполнять код асинхронно.

Рассмотрим каждое понятие отдельно.

## Процессы



### БЛОГ НА ХАБРЕ

3 апр в 14:10

**Применяем стандартные алгоритмы в C++. Семь примеров**

👁 10K 💬 9 +9

19 мар в 14:36

**Быстрый старт, или Как ускорить запуск iOS-приложений**

Процессы являются контейнерами. Их основная задача – изолировать программы друг от друга, чтобы одна не могла получить доступ к памяти другой.

В контексте Python каждому процессу выделен свой интерпретатор. Когда мы запускаем несколько процессов из кода, то мы обнаруживаем такое же количество процессов в мониторинге системы.

Небольшой пример создания процессов:

```
from multiprocessing import Process

def print_word(word):
    print('hello,', word)

if __name__ == '__main__':
    p1 = Process(target=print_word, args=('bob',), daemon=True)
    p2 = Process(target=print_word, args=('alice',), daemon=True)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

Процессы представлены как экземпляр класса Process из встроенной библиотеки multiprocessing.

У нас есть функция, которая принимает 1 параметр и печатает приветствие с переданным параметром. Внутри конструкции if мы создаем два процесса p1 и p2 в качестве параметров, то есть мы передаем:

**target** – с названием выполняемой функции,

 1.8K  5 +5

12 мар в 11:40

Как использовать ChatGPT для разработки и учебы. Четыре сценария

 12K  0

29 фев в 12:53

Хватит маппить все руками, используй Mapster

 19K  24 +24

20 фев в 13:10

Автоматизируем проверку содержимого PDF-файлов с помощью pdf-test

 3.5K  1 +1

**args** – параметры для функции, которую мы будем вызывать,

**daemon** – с флагом True, который говорит нам, что процесс будет являться «демоном» – об этом чуть позже.

Для того чтобы процесс стартовал, мы вызываем у каждого метод `.start()`.

Но ниже мы вызываем еще и метод `.join()`.

## Для чего нужен `join()` и что такое **daemon**? Или основные и фоновые процессы

У нас есть основной (**главный**) процесс, который содержит весь код нашей программы, и два дополнительных (**фоновых**) `p1`, `p2`. Их мы создаем, когда мы прописываем параметр `daemon=True`. Так мы как раз и указываем, что эти два процесса будут второстепенными. Если мы не вызовем метод `join` у фонового процесса, то наша программа завершит свое выполнение, не дожидаясь выполнения `p1` и `p2`.

## Немного теории о процессах

Процессы не могут работать параллельно на одноядерной машине.

Параллельное вычисление – выполнение двух и более задач одновременно, когда каждое ядро процессора берет задачу и выполняет ее. На многоядерной машине параллельное вычисление – нормальная практика. Однако количество ядер у нас ограничено, причем весьма сильно, а процессов в системе работает много.

Познакомимся с еще одним термином — вытесняющая многозадачность.

**Вытесняющая многозадачность** — это такой способ управления задачами, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого принимается планировщиком операционной системы.

Предположим, что у нас одноядерный процессор и ему приходится выполнять работу множества программ одновременно. Как он это делает?

В этом случае каждой программе выделяется небольшой промежуток времени, то есть программы конкурируют за доступ к ядру. Процессор сам переключает контекст выполнения, и таким образом создается впечатление, что программы работают одновременно. Но это не совсем так.

Проще говоря, одна программа поработала какое-то время, и процессор переключает контекст на другую, чтобы она выполнила запланированные действия, передала обратно и так далее.

Когда количество процессов превышает количество ядер, на помощь приходит конкурентное вычисление.

## Потоки

Первое, о чем хотим сказать про потоки — интерфейсы работы с процессами и потоками в Python очень похожи.

Потоки живут внутри процессов, потребляют меньше ресурсов и разделяют общую память внутри процесса. Во многих языках программирования потоки создавались именно для того, чтобы выполнять задачи параллельно, но не в Python. А виноват в этом GIL.

**GIL (Global interpreter lock)** следит за тем, чтобы в один момент времени работал лишь один поток. Механизм похож на то, как процессы конкурируют за ядро. Но в отличие от процессов GIL освобождается при вызове блокирующей функции операций ввода/вывода. Другой механизм его освобождения – `time.sleep()`. Об этом позже.

```
import threading

def greet(name):
    print('hello: ', name)

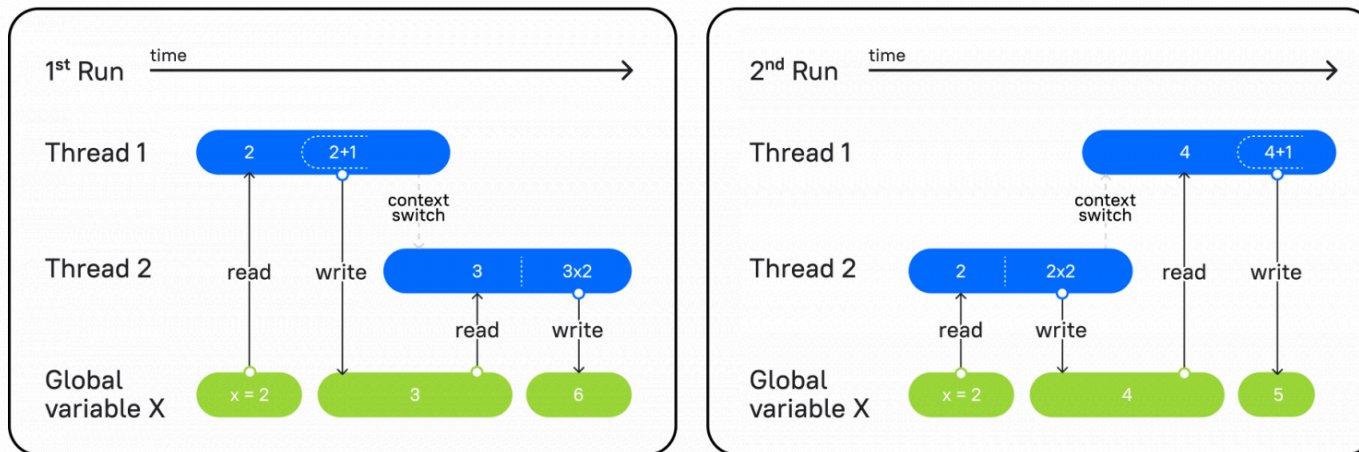
if __name__ == '__main__':
    t1 = threading.Thread(target=greet, args=('bob',), daemon=True)
    t2 = threading.Thread(target=greet, args=('alice',), daemon=True)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

Как видно, процесс создания потоков идентичен алгоритму формирования процессов.

Теперь, когда мы познакомились с основными понятиями, продемонстрируем несколько проблем, которые встречаются в многопоточном программировании.

Первая проблема – **Race Condition** или **состояние гонки**.





На изображении мы видим два запуска одной и той же программы, в которой есть два потока: в первом функция увеличивает переданное число на единицу, а во втором — мы умножаем число на 2.

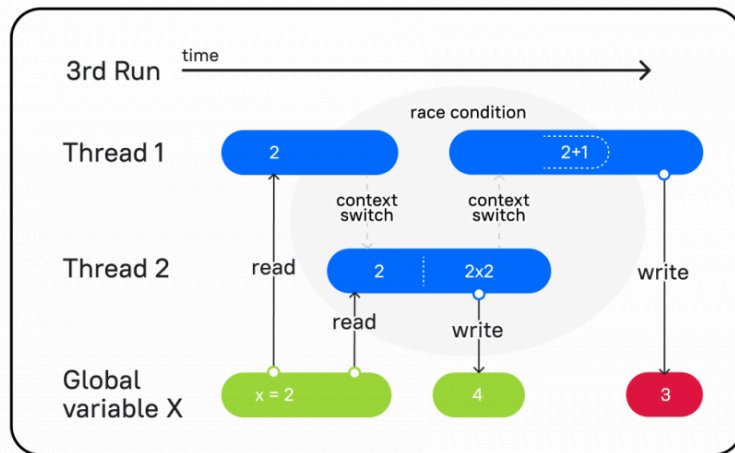
**Слева** вы видите первый запуск программы. Первый поток берет значение из глобальной переменной  $x$ , прибавляет 1 и записывает в  $x$  результат  $= 3$ . Затем второй поток начинает работу. Он берет из переменной  $x$  значение 3, умножает на 2 и записывает результат  $= 6$ .

На **правой** схеме – второй запуск программы, где сперва в работу вступает поток 2, он выполняет те же операции, берет  $x = 2$ , умножает на 2 и фиксирует результат 4. Затем вступает поток 1, читает 4 из  $x$ , увеличивает на единицу и записывает 5.

Так как оба потока меняли порядок работы программы, но выполняли ее по очереди, у нас не возникало никакого конфликта, и мы получали ожидаемый результат.

**Но давайте посмотрим на такой поток выполнения:**





Поток 1 вступает в работу, читает переменную  $x$  и переключает контекст на поток 2 (context switch). Затем поток 2 берет значение из  $x = 2$ , умножает на 2 и записывает в  $x = 4$ . Процессор переключает контекст на поток 1, а в потоке 1, как мы помним, сохранено значение  $x = 2$ . В итоге он увеличивает значение на единицу и записывает в  $x = 3$ , а значит, на выходе мы получаем 3.

Один поток обогнал другой при переключении контекста, и мы получили непредсказуемый результат. Такое событие называется **Race condition**. Как тогда быть уверенным в том, что поток, взявший в работу какие-то данные, выполнит свою работу, перед тем как переключит свой контекст на другой потоку?

Вот пример:

```
...
from threading import Thread
from time import sleep

counter = 0
```

```
def increase(by):  
    global counter  
  
    local_counter = counter  
    local_counter += by  
  
    sleep(0.1)  
  
    counter = local_counter  
    print(f'{counter=}')  
  
t1 = Thread(target=increase, args=(10,))  
t2 = Thread(target=increase, args=(20,))  
  
t1.start()  
t2.start()  
  
t1.join()  
t2.join()  
````
```

Посмотрим на результат:

```
````  
counter=10  
counter=20  
````
```

Вместо 30 получаем 20.

На помощь нам может прийти такое понятие как **Lock**.

**Lock (замок)** – объект, который захватывает поток, и пока поток не освободит (release) Lock, другие потоки не смогут ничего сделать с этими данными, захваченными при помощи замка.

```
'''  
from threading import Thread, Lock  
from time import sleep  
  
counter = 0  
  
def increase(by, lock: Lock):  
    global counter  
  
    lock.acquire()  
  
    local_counter = counter  
    local_counter += by  
  
    sleep(0.1)  
  
    counter = local_counter  
    print(f'{counter=}')  
  
    lock.release()  
  
lock = Lock()
```

```
t1 = Thread(target=increase, args=(10, lock,))
t2 = Thread(target=increase, args=(20, lock,))

t1.start()
t2.start()

t1.join()
t2.join()
'''
```

Вот теперь как и должно быть:

```
'''
counter=10
counter=30
'''
```

Несмотря на то, что **Lock** помогает решить проблему с **Race condition**, он может привести к другой сложной ситуации, когда один поток ждет освобождение одного замка, а другой ждет освобождение от первого. Такое ожидание приводит к ситуации взаимного тупика, известного как **Deadlock**.

```
'''

from threading import Thread, Lock
from time import sleep

a = 5
```

```
b = 10
```

```
a_lock = Lock()
```

```
b_lock = Lock()
```

```
def function_a():
```

```
    global a
```

```
    global b
```

```
    a_lock.acquire()
```

```
    print('Функция a, a_lock = заблокирован')
```

```
    sleep(1)
```

```
    b_lock.acquire()
```

```
    print('Функция a, b_lock = заблокирован')
```

```
    sleep(1)
```

```
    a_lock.release()
```

```
    print('Функция a, a_lock = разблокирован')
```

```
    b_lock.release()
```

```
    print('Функция a, b_lock = разблокирован')
```

```
def function_b():
```

```
    global a
```

```
    global b
```

```
    b_lock.acquire()
```

```
    print('Функция b, b_lock = заблокирован')
```

```
    a_lock.acquire()
```

```
    print('Функция b, a_lock = заблокирован')
```

```
sleep(1)

b_lock.release()
print('Функция b, b_lock = разблокирован')
a_lock.release()
print('Функция b, a_lock = разблокирован')

t1 = Thread(target=function_a)
t2 = Thread(target=function_b)

t1.start()
t2.start()

t1.join()
t2.join()

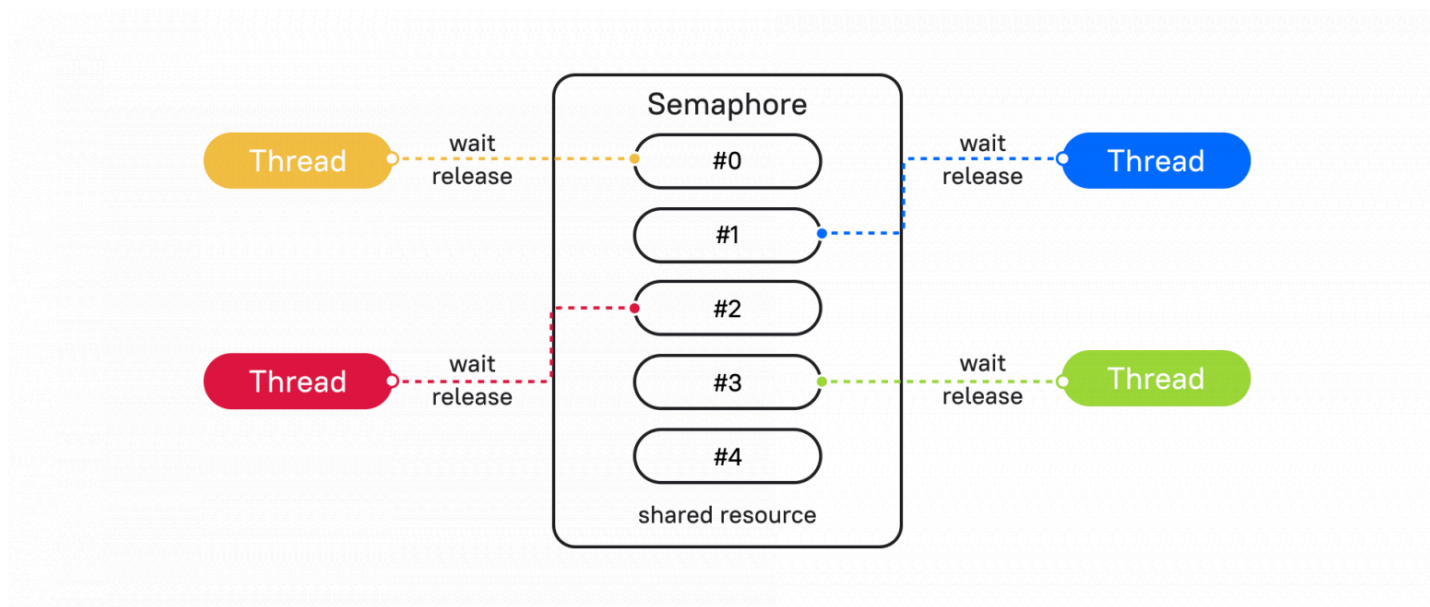
print('Готово')
'''
```

И теперь посмотрим результат:

```
'''
Функция a, a_lock = заблокирован
Функция b, b_lock = заблокирован
'''
```

Наша программа зависает в ожидании разблокировки, которая никогда не произойдет. Так же **Deadlock** произойдет при попытке заблокировать наш **Lock** повторно в том же потоке.

Решить проблему с **Deadlock** могут помочь различные механизмы синхронизации потоков. Разберем один из таких примеров – **Semaphore (Семафор)**.



**Semaphore** прост в понимании, если его представить в виде объекта, который ограничивает выполнение блока кода установленным количеством, по умолчанию это 1. При каждом вхождении в блок кода **Semaphore** счетчик уменьшается. Если счетчик дошел до 0, все потоки блокируются, и пока поток не освободит семафор, другие будут ждать разрешения подключиться.

Посмотрим **Semaphore** на примере реализации очереди из реального кейса.

```
...  
  
import datetime  
from threading import Semaphore, Thread  
from time import sleep
```



```

s = Semaphore(3)

def semaphore_func(payload: int):
    s.acquire()
    now = datetime.datetime.now().strftime('%H:%M:%S')
    print(f'{now=}, {payload=}')
    sleep(2)
    s.release()

threads = [Thread(target=semaphore_func, args=(i,)) for i in range(7)]

for t in threads:
    t.start()

for t in threads:
    t.join()

```

В результате увидим, что функция выполнялась группами по 3 потока. То есть одновременно не может выполняться кусок кода с блокировкой через **Semaphore** больше, чем указан в инициализации класса **Semaphore**. Видим паузы в 2 секунды между блокировками.

```

...
now='00:49:51', payload=0
now='00:49:51', payload=1
now='00:49:51', payload=2
now='00:49:53', payload=3
now='00:49:53', payload=5
now='00:49:53', payload=4

```

```
now='00:49:55', payload=6
...
```

Это удобно использовать, например, в таком виде: если база данных может держать не более 30 соединений, то инстанцируем **Semaphore** со значением 30. Блокируем, когда поднимаем соединение и разблокируем, когда освобождаем.

Есть несколько способов синхронизации потоков, которые подходят для тех или иных ситуаций. Примеры можно посмотреть [в документации](#).

Теперь поговорим об освобождении GIL.

CPython управляет памятью с помощью подсчета ссылок. То есть для каждого объекта Python подсчитывается, сколько на него указывается ссылок с других объектов, использующих его в данный момент. При добавлении ссылки счетчик увеличивается, при удалении ссылки счетчик уменьшается. А когда счетчик ссылок становится 0 — это означает, что объект больше не нужен, и его можно удалить из памяти.

Следовательно, если не будет GIL, который запрещает Python процессу выполнять более одной команды байт-кода в каждый момент времени, то при подсчете ссылок может случиться Race-condition, с подсчетом ссылок на объекты, как это было в примере с переменными выше.

Итак, раз GIL запрещает одновременное выполнение Python кода, из этого следует, что он высвобождается, когда Python код не выполняется. Когда мы ждем, например, пока считывается файл с диска или придет ответ на запрос к сайту. Так как в этом случае низкоуровневые системные вызовы работают за пределами Python кода и среды выполнения, и код операционной системы не взаимодействует напрямую с объектами Python, соответственно, они не увеличивают и не уменьшают счетчик ссылок. GIL захватывается снова, когда данные переносятся в объект Python.

Стало быть, если мы сделаем библиотеку, даже с CPU-bound нагрузкой, где мы не взаимодействуем с объектами Python (словарями, списками, целыми числами и т. д.) или большая часть библиотеки не взаимодействует, то мы можем освободить GIL. Например, библиотеки `hashlib` и `NumPy` выполняют расчеты на чистом C и освобождают GIL.

`time.sleep()` — реализация этой функции освобождает GIL и выполняется на уровне системы и работает вне кода Python.

Как видите, в многопоточности существует огромное количество нюансов и проблем. В реальных больших программах будет непросто понять, где происходит ошибка. Рассмотрим, как можно распараллелить выполнение программ. В этом поможет асинхронность.

## Асинхронность

Для того чтобы лучше понять асинхронность, окунемся в далекий 1992 год. Тогда была выпущена операционная система Windows 3.1 которая использовала кооперативную многозадачность.

**Кооперативная многозадачность** — это тип многозадачности, при котором фоновые задачи выполняются только во время простоя основного процесса и только в том случае, если на это получено разрешение основного процесса.

То есть время, когда исполняемая программа управляет передачей управления другому процессу и передачей процессорного времени.

**Недостатком** такого исполнения является то, что если одна задача зависла. Зависает вся система.

А вот **преимущества** такого решения: разработчик программы отдает управление тогда, когда он посчитает это нужным.

Теперь мы подобрались к понятию асинхронного программирования.

**Асинхронное программирование** — выполнение программы в неблокирующем режиме системного вызова, что позволяет потоку программы продолжить работу.

Благодаря асинхронному программированию в одном процессе и даже потоке мы можем выполнять сразу множество задач. Как же это происходит?

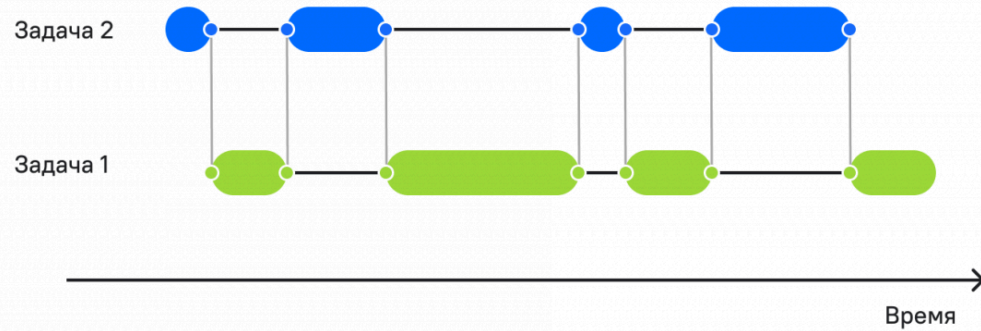
В реальном программировании, а особенно в web-разработке мы очень часто чего-то ждём и не делаем полезной работы. Вот несколько примеров:

- Отправили запрос на сторонний ресурс и ждем ответа.
- Отправили запрос в базу данных и ждем результата запроса.
- Читаем или записываем файл на диск.
- И так далее.

Получается что мы ждем, ждем и ждем. А в это время наша программа могла бы выполнить множество полезной нагрузки. И мы как разработчики ПО точно знаем, где мы будем ожидать. Ничего не напоминает? Да! Похоже на **кооперативную многозадачность**, но только не на уровне операционной системы, а на уровне процесса.



На рисунке видно, что периодов ожидания много. А что будет если во время ожидания мы будем выполнять полезную работу?



Как видно на рисунке, в моменты ожидания мы выполняем уже две задачи за то же самое время. Чем быстрее мы выполняем работу и чем дольше мы ожидаем, тем больше задач мы можем сделать за одно и то же время.

Для реализации такого поведения асинхронности есть несколько подходов:

- Реализация на основе коллбэков.
- Реализация на основе корутин.

Оба подхода имеют место. Например, мощный фреймворк TORNADO реализован именно на основе коллбэков.

**У этого подхода есть ряд недостатков:**

- Код перестает выглядеть как синхронный, что усложняет отладку.
- Ад коллбэков, в котором будет сложно разобраться. Просто погуглите фразу “callback hell”.

Если после этих минусов желание попробовать ещё осталось, то можно в подходе легко разобраться.

А вот подход на основе корутин мы разберем более глубоко. У него также есть ряд преимуществ и недостатков:

**Плюсы:**

- Асинхронный код выглядит как синхронный.
- Нет проблем с общей памятью, и избавляемся от синхронизаций.

- Не нужно переключать контекст между задачами, что экономит ресурсы нашего компьютера.
- Теперь нам не нужны коллбэки, но их также можно использовать.

### Минусы:

- Чуть более сложный подход для понимания.

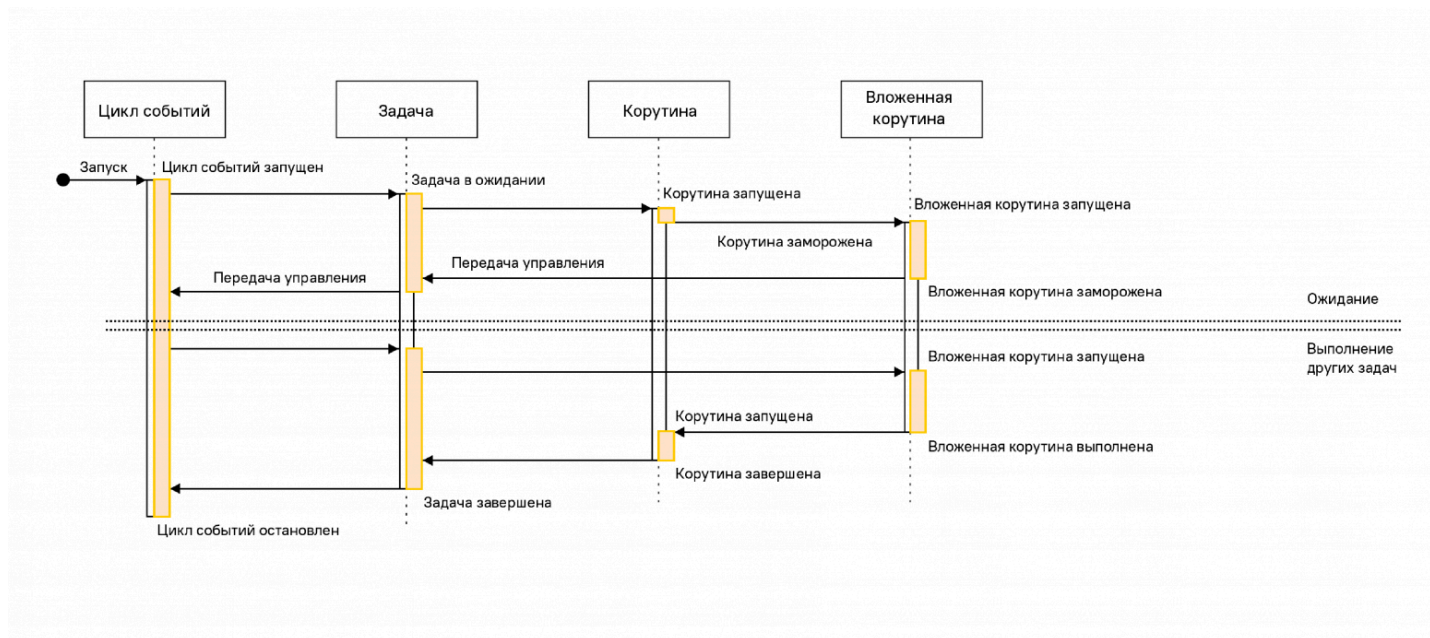
В Python есть ряд библиотек, которые позволяют работать с асинхронностью:

- **asyncio** — основная библиотека для работы с асинхронным программированием,
- **aiohttp** — для асинхронной работы с запросами,
- **aiofiles** — для работы с файловой системой.

Как вы наверное заметили, у библиотек есть префикс **aio (asynchronous input output, асинхронный ввод-вывод)**. Тут как раз решается проблема ожидания. Такие задачи называют **IO bound**.

Рассмотрим термины, которые нам помогут во всём разобраться.





**Event loop (цикл событий)** — ядро каждого приложения `asyncio`. Циклы событий запускают асинхронные задачи и обратные вызовы, выполняют операции сетевого ввода-вывода и запускают подпроцессы. Официальную документацию можно прочесть [тут](#).

**Корутины** — это специальные функции, которые запускаются, используя цикл событий. У них есть особенность — они говорят, когда они будут ждать и передают управление обратно, чтобы другая задача могла выполняться во время ожидания.

**Футуры** — это определение обычно воспринимается тяжелее всего, но я постараюсь объяснить как можно проще. Это объект, в котором хранится результат и состояние задачи:

+ ожидание (pending)

+ выполнение (running)

+ выполнено (done)

+ отменено (cancelled)

То есть в процессе работы мы можем управлять задачами в зависимости от футуры (статус/результат) задачи.

Корутины могут быть реализованы с использованием генераторов или `async/await`. Мы выбираем второй вариант как более лаконичный.

Посмотрим, как это выглядит в коде.

Создадим первую корутину:

```
'''
import asyncio

async def hello():
    print('Запуск функции hello')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждем
    print('Переключение контекста в функцию hello')
'''
```

Теперь у нас есть асинхронная функция. Научимся теперь её запускать. Первое, что хочется сделать — вызвать её как обычную функцию. Давайте попробуем:

```
'''
import asyncio
```

```
async def hello():
    print('Запуск функции hello')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждём
    print('Переключение контекста в функцию hello')

hello()
...
```

При выполнении ничего не произошло. А вот наш друг интерпретатор выдал предупреждение.

```
...
RuntimeWarning: coroutine 'hello' was never awaited
    hello()
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
...
```

Тут из сообщения становится понятно, что при вызове таким образом асинхронной функции она превращается в асинхронную корутину.

Как же можно запустить корутину?

- Из другой корутины.
- Обернуть в задачу.
- Запустить через метод `asyncio.run` и `run_until_complete` из цикла событий.

```
'''
import asyncio

async def hello():
    print('Запуск функции hello')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждём
    print('Переключение контекста в функцию hello')

asyncio.run(hello())
'''
```

И получили результат, который ожидали.

```
'''
Запуск функции hello
Переключение контекста в функцию hello
'''
```

Вызов метода **asyncio.run(hello())** принимает корутину, которую необходимо выполнить, открывает цикл событий, выполняет корутину и закрывает цикл событий.

Что делать, если необходимо запустить две задачи конкурентно?

Это поможет нам сделать **asyncio.gather**, но раз функция **asyncio.run** принимает только одну корутину, создадим новую корутину, которая будет запускать конкурентно несколько задач.

```
'''
import asyncio

async def hello():
    print('Запуск функции hello')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждём
    print('Переключение контекста в функцию hello')

async def starter():
    await asyncio.gather(hello(), hello())

asyncio.run(starter())
'''
```

И получаем тот результат, который ожидали.

```
'''
Запуск функции hello
Запуск функции hello
Переключение контекста в функцию hello
Переключение контекста в функцию hello
'''
```

Время выполнения около 5 секунд. Если бы две функции выполнялись синхронно, то время выполнения составило около 10 секунд.

А если нам необходимо выполнить 10 тысяч раз, сколько времени это займёт?

Видоизменяем код:

```
'''
import asyncio
import time

start = time.time() ## точка отсчета времени

async def hello():
    print('Запуск функции hello')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждём
    print('Переключение контекста в функцию hello')

async def starter():
    await asyncio.gather(*[hello() for i in range(10000)])

asyncio.run(starter())

end = time.time() - start
print(end)
'''
```

Получаем результат. Посмотрим на вывод последних нескольких строк, которые нам говорят, сколько минут выполнялся код.

```
'''
...
'''
```

```
Переключение контекста в функцию hello
Переключение контекста в функцию hello
Переключение контекста в функцию hello
5.27926778793335
...
```

Неплохо. Чуть больше тех же самых 5 секунд.

**Что же это значит?** Представьте, что запрос на сторонний сайт занимает порядка 5 секунд. И нам необходимо получить результат тех же самых 10000 запросов. Используя асинхронное программирование, 10 тысяч запросов сеть будут выполняться чуть больше 5 секунд. Правда, здорово?

Но мы пойдем дальше и будем уже более гибко и детально работать с асинхронным выполнением:

```
...

import asyncio

async def hello():
    print('Запуск функции hello')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждём
    print('Переключение контекста в функцию hello')

async def bye():
    print('Запуск функции bye')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждём
    print('Переключение контекста в функцию bye')
```



```

ioloop = asyncio.get_event_loop()
tasks = [ioloop.create_task(hello()), ioloop.create_task(bye())]
tasks_for_wait = asyncio.wait(tasks)
ioloop.run_until_complete(tasks_for_wait)
ioloop.close()
'''

```

В этом примере мы более гибко управляем циклом событий. Сначала получаем/создаем основной цикл событий. Затем создаем задачи и объединяем их запускаем на выполнение, пока не завершится. Затем уже закрываем цикл событий. Нужно помнить, что порядок выполнения задач при конкурентном выполнении мы не можем гарантировать, и необходимо разрабатывать приложения с учетом этой особенности.

Теперь давайте попробуем управлять выполнениями задач и рассмотрим код ниже:

```

'''
import asyncio

async def hello():
    print('Запуск функции hello')
    await asyncio.sleep(5) # Отдаем управление обратно в Event loop пока ждём
    print('Переключение контекста в функцию hello')
    return 'Выполнена функция hello'

async def bye():
    print('Запуск функции bye')

```

```
await asyncio.sleep(2) # Отдаем управление обратно в Event loop пока ждём
print('Переключение контекста в функцию bye')
return 'Выполнена функция bye'
```

```
async def starter(ioloop):
    tasks = [ioloop.create_task(hello()), ioloop.create_task(bye())]
    done, pending = await asyncio.wait(tasks, return_when=asyncio.FIRST_COMPLETED)
    result = done.pop().result()

    for pending_future in pending:
        pending_future.cancel()

    print(result)

ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(starter(ioloop))
ioloop.close()
'''
```

Результат будет таким:

```
'''
Запуск функции hello
Запуск функции bye
Переключение контекста в функцию bye
Выполнена функция bye
'''
```

Теперь только представьте, какие возможности у нас открылись! Например, мы можем запрашивать курсы валют сразу с нескольких ресурсов, и принимать результат того, который быстрее ответит. Чувствуете, как растёт скорость и устойчивость приложения?

Или ещё такой пример. Мы можем динамически добавлять новые задачи, когда одна из задач выполнена. Например, парсить сайт в 20 задач. Только в этом случае добавляем к футурам в статусе **pending** новую задачу.

А самое приятное — наши асинхронные задачи выглядят как синхронные:

- Работая в один поток, можно делать больше работы;
- Удобная отладка;
- Нет проблем с блокировками;
- Можем использовать обратные вызовы (коллбэки) и отложенные обратные вызовы вдобавок к нашему асинхронному коду. Для этого посмотрите на методы цикла событий **call\_soon**, **call\_later**, **call\_at**.

Для работы с конкурентностью есть различные библиотеки, которые решают самые востребованные задачи IO:

- **aiohttp** — работа с HTTP запросами;
- **aiofiles** — работа с файлами.

Мы рассмотрели темы асинхронного и параллельного программирования. Теперь осталось дело за малым, опробовать всё это на практике.

## Итого

## **Отдельные процессы**

### **Плюсы:**

- + Работают параллельно.
- + Используют все ресурсы ядра процессора.
- + Можно загрузить все ядра процессора.
- + Изолированная память.
- + Независимые системные процессы.
- + Подходит для CPU bound операций.

### **Минусы:**

- Если необходимо использовать общую память, то необходимо синхронизировать, так как нет общих переменных.
- Требуют больших ресурсов, так как запускают отдельный интерпретатор.

**Используем там, где обрабатываемые данные не зависят от других процессов и данных. Например:**

- + Расчет нейронных сетей.
- + Обработка изолированных фотографий.
- + Архивирование изолированных файлов.

+ Конвертация форматов файлов.

## **Отдельные потоки**

### **Плюсы:**

+ Работают параллельно.

+ Используют немного памяти.

+ Общая память.

### **Минусы:**

- Одновременный доступ к памяти может приводить к конфликтам.
- Сложный код.

**Используем там, где код много раз ожидает, пока выполнится задача. Например:**

+ Работа с сетью.

## **Асинхронность**

### **Плюсы:**

+ Работает в одном процессе и в одном потоке.

+ Экономное использование памяти.

+ Подходит для I/O bound операций.

+ Работает конкурентно.

### **Минусы:**

- Сложность отладки.
- CPU bound операции блокируют все задачи.

### **Используем там, где код много раз ожидает. Например:**

+ Работа с сетью.

+ Работа с файловой системой.

Основываясь на конкретных плюсах и минусах, нам становится легче выбирать подход и грамотно использовать процессорное время и память. Хотя Python является мультипарадигменным языком общего назначения, на нем можно писать практически любые программы, используя любой подход. Но особенно приятно, когда ваш веб-сервис может держать в сотню раз больше соединений или обрабатывать запросы в 8 раз быстрее, обходясь меньшим количеством памяти.

**Спасибо за внимание! Надеемся, что этот материал был полезен для вас.**

**Авторские материалы для разработчиков мы также публикуем в наших соцсетях – [VK](#) и [Telegram](#).**

**Теги:** python, многопоточность, многопоточное программирование, асинхронность, асинхронное программирование

**Хабы:** Блог компании SimbirSoft, Python, Программирование,  
Параллельное программирование

## Редакторский дайджест



Присылаем лучшие статьи раз в месяц

Электронная почта



SimbirSoft

Лидер в разработке современных ИТ-решений на заказ

[Сайт](#) [ВКонтакте](#) [Telegram](#) [Telegram](#) [ВКонтакте](#)



83

5

Карма Рейтинг



Подписаться

@SSul

Пользователь

## Комментарии 16




**outlingo** 23 ноя 2022 в 08:19

Семафор не избавляет от дедлока. Можно точно также в разных тредах полностью захватить семафор и при попытке захвата второго получить дедлок. И что намного более важно -



блокировка (Lock) позволяет защитить фрагмент кода от параллельного выполнения полностью, а семафор ограничивает количество параллельных выполнений, но не препятствует им.

 **+1** Ответить  


○  **SSul** 24 ноя 2022 в 05:32 

Решить проблему с Deadlock могут помочь 2 правила:

Так как Deadlock — это взаимная блокировка, т.е. захват в прямом и обратном направлении в разных потоках, отсюда вытекает правило. Если в процессах нужно захватывать Lock, то делаем это всегда в одном порядке (направлении).

И второе правило — освобождаем блокировки в обратном порядке их захвата, подобно стеку. Первым захватили — последним отпустили.

 **+1** Ответить  

○  **sci\_nov** 24 ноя 2022 в 07:45 

А что значит направление захвата?

 **0** Ответить  

○  **fishHook** 23 ноя 2022 в 13:47

Для чего нужен `join()` и что такое **daemon**? Или основные и фоновые процессы

У нас есть основной (**главный**) процесс, который содержит весь код нашей программы, и два дополнительных (**фоновых**) `p1`, `p2`. Их мы создаем, когда мы прописываем параметр `daemon=True`. Так мы как раз и указываем, что эти два процесса будут второстепенными. Если мы не вызовем метод `join` у фонового процесса, то наша программа завершит свое выполнение, не дожидаясь выполнения `p1` и `p2`.

То ли я тупой, то ли у автора дислексия, то ли он сам не понимает, что он пишет. Но вот где, в какой строке этого сумбурного текста даётся определение, что же такое **демон**? Ну дружище, блин, ты же пишешь инженерную статью. Есть документация на сайте python.org. Ваша статья должна давать что-то большее, чем эта документация, иначе какой в ней смысл? Ну то есть, какие-то тонкие и непонятные моменты должны быть разъяснены, правильно? Ну давайте вчитаемся в то, что написано в этом рандомно взятом абзаце.

> процесс, который содержит весь код нашей программы и два дополнительных

То есть делаем вывод, что подпроцессы не содержат весь код программы, правильно? А что же они тогда содержат?

> Их мы создаем, когда мы прописываем параметр `daemon=True`

Ага, то есть если в вашем же примере

```
if __name__ == '__main__':  
    p1 = Process(target=print_word, args=('bob',), daemon=True)  
    p2 = Process(target=print_word, args=('alice',), daemon=True)
```

сделать `daemon=False`, мы **не** создадим два дополнительных процесса? А что создадим?

Лучше ничего не писать, чем писать такое

 **+1** [Ответить](#)  

 **select26** 23 ноя 2022 в 19:49 

А мне статья понравилась.

Хорошее дополнение к документации, несмотря на огрехи. Которые, кстати, легко исправить.

0 Ответить

gosox 25 ноя 2022 в 08:17

Так напишите статью лучше, а мы оценим.

Как по мне, статья качественная.

+1 Ответить

mrkaban 24 ноя 2022 в 07:27

Спасибо за интересную статью, а если помимо параллельного выполнения задач нужно, чтобы они между собой обменивались данными? и при этом графический интерфейс отображал прогресс?

+1 Ответить

SSul 24 ноя 2022 в 12:48

Спасибо!

Чтобы обмениваться, multithreading и асинхронка уже используют общую память.

Для обмена между процессами можно использовать очереди (Queue) или каналы данных (pipe).

0 Ответить

TyVik 11 дек 2022 в 17:10

Обмен данными между процессами можно организовать ещё через сокеты. Так что смотрите на семантику того, чего отображаете. Если это какая-то отдельная программа или упираться в gil, то лучше через процессы. Вариантов много.

Хотя, кажется в 3.11 хотели переместить gil в каждый поток, созданный python. Но это не точно.

0 Ответить

isBlaze 24 ноя 2022 в 08:32

Вы упомянули GIL, пообещали позже рассказать про его освобождение через `time.sleep()`, и забыли про это. А ещё в итогах не упомянули этот вполне себе существенный минус. Фактически, все потоки работают на одном ядре. И если нам надо чем то загрузить процессор, мы не получим выигрыша от потоков. Как и от AIO впрочем.

+1 Ответить

GothicJS 24 ноя 2022 в 15:54

Отдельные потоки

**Плюсы:**

+ Работают параллельно.

Но в пайтоне есть GIL, и потоки работают конкурентно. Тогда правильно ли я понимаю, что в пайтоне при наличии асинхронности теперь вообще нет смысла использовать многопоточность ?

0 Ответить

SSul 23 дек 2022 в 10:42


Есть смысл. Так как при использовании асинхронности мы самостоятельно переключаем контекст и поток исполнения. А при многопоточности этим занимается операционная система.

0 Ответить

○  **TimurBaldin** 28 ноя 2022 в 10:52

Не совсем понимаю....если существует GIL , который не дает потокам работать одновременно, то в чем смысл тогда блокировок и прочего ? если в один момент времени, будет работать только один поток, а все остальные будут спать

◆ 0 Ответить  

○  **fireSparrow** 28 ноя 2022 в 12:00  

Ну, например для того, чтобы безопасно сделать неатомарные манипуляции с общими ресурсами.

Например, у вас есть некий словарь. И вам нужно брать из него значения, производить на основе этого значения какое-то новое значение и сохранять под тем же ключом.

Вы пишете так:

```
value = my_dict[key]
new_value = calculate_new_value(value)
my_dict[key] = new_value
```

Теперь представьте, что где-то между первой и третьей строчкой выполнение переключилось на другой поток, и этот другой поток обновил значение `my_dict[key]`. Когда выполнение вернётся к первому потоку, он просто перезапишет поверх него какое-то уже неактуальное значение. А если бы перед записью в словарь наш код брал бы лок, такого бы не случилось.

◆ +2 Ответить  

○  **iosuslov** 5 дек 2022 в 09:59

Очередная статья про "потoki, процессы и асинхронность" - зачем их столько? Хотя бы примеры какие-то из жизни привели, но нет все какая-то синтетическая копия паста.

Всю эту историю можно свести к тому, что там где код упирается в await - происходит переключение на следующую функцию с периодическим возвратом, пока она не вернет ответ.

А процессы работают через ProcessPoolExecutor, который инициализируется при старте приложения и, по необходимости, в него записываются нужные функции (и тоже периодически проверяется готовность результата)

 0 Ответить  ...

 **releyshic** 21 янв 2023 в 01:22

какие есть способы сделать 2 окна активными? В одном будет видео из OpenCV, а в другом элементы управления (ползунки кнопки)

 0 Ответить  ...

Зарегистрируйтесь на Хабре, чтобы оставить комментарий

## Публикации

ЛУЧШИЕ ЗА СУТКИ    ПОХОЖИЕ

 **AKlimenkov** 21 час назад

### Почему FAR — центр моей компьютерной вселенной

 4 мин     22K

Мнение

♦ +97    📖 94    💬 212 +212



difhel 23 часа назад

## Вы должны перестать вручную писать Dockerfile'ы

💧 Средний    ⌚ 3 мин    👁 22K

Из песочницы

Перевод

♦ +45    📖 179    💬 35 +35



Erwinmal 20 часов назад

## Как пытались (пере)программировать мозги, и что из этого получилось? Часть 4: Нуарный коп, оргии и тюремные эксперименты

🟢 Простой    ⌚ 10 мин    👁 3.4K

Ретроспектива

♦ +39    📖 24    💬 4 +4



Bright\_Translate вчера в 15:00

## Не бойтесь бросать свои пет-проекты

🟢 Простой    ⌚ 6 мин    👁 4K

Мнение

Перевод

♦ +39    📖 35    💬 14 +14



myswordishatred вчера в 15:10

## Театр образования



Простой



6 мин



2K

Мнение



+18



13



14

+14



Andrey\_Biryukov 18 часов назад

## Тсrdump на разных уровнях



4 мин



3.1K



+16



83



3

+3



OldfagGamer 2 часа назад

## Мах Рауне: хороша ли неонуарная классика сегодня?



Простой



13 мин



710

Ретроспектива



+13



2



1

+1



MimusTriurus 3 часа назад

## Интерактивный NPC на Unreal Engine



5 мин



919

Из песочницы



◆ +9    📌 9    💬 1 +1



linuxacademy 18 часов назад

## Дайджест полезных материалов из мира Golang за неделю (6.04.24-13.04.24)

🟢 Простой    ⌚ 2 мин    👁 1.4K

Дайджест

◆ +9    📌 11    💬 0



Seleditor 1 час назад

## Импортозамещение по-китайски: Huawei разрабатывает инструменты для производства современных чипов

⌚ 3 мин    👁 564

◆ +8    📌 3    💬 1 +1

Показать еще

### ВАКАНСИИ КОМПАНИИ «SIMBIRSOFT»

#### Golang разработчик

SimbirSoft · Можно удаленно

Ваш аккаунт

- Войти
- Регистрация

Разделы

- Статьи
- Новости
- Хабы
- Компании
- Авторы
- Песочница

Информация

- Устройство сайта
- Для авторов
- Для компаний
- Документы
- Соглашение
- Конфиденциальность

Услуги

- Корпоративный блог
- Медийная реклама
- Нативные проекты
- Образовательные программы
- Стартапам

