Введение в потоки в Python

02.04.2019 Editorial Team 0 Комментариев threading 10 391 просмотров

Статья рассказывает о базовом использование потоков в Python. Что это такое и как правильно с ними работать. Так же если вы думаете что для базового использования потоков достаточно знать как запустить **threading.Thread** , то это статья именно для вас. На самом деле все чуть более интереснее…. Оригинал статьи: Jim Anderson An Intro to Threading in Python

Потоки (thread) в Python позволяет одновременно запускать различные подпрограммы и могут упростить архитектуру проектов. Если у вас есть некоторый опыт работы с Python и вы хотите ускорить свою программу с помощью потоков, тогда это руководство для вас!

В этой статье вы узнаете:

- Что такое потоки
- Как создавать потоки и ждать их окончания
- Как использовать `ThreadPoolExecutor`
- Как избежать условий гонки (race conditions)
- Как использовать распространенные инструменты, которые предоставляют потоки Python

В этой статье предполагается, что у вас есть базовые знания Python и что для запуска примеров вы используете как минимум **версию 3.6** .

Все исходники, используемые в этой статье, доступны в репозитории *Real Python* GitHub repo.

Что такое Thread (Поток)?

Thread – это отдельный поток выполнения. Это означает, что в вашей программе могут работать две и более подпрограммы одновременно. Но разные потоки на самом деле не работают одновременно: это просто кажется.

Соблазнительно думать, что в вашей программе работают два (или более) разных процессора, каждый из которых выполняет независимую задачу одновременно. Это почти правильно, но это то, что обеспечивает **многопроцессорность** (**multiprocessing**).

Запуск потоков (**threading**) похожа на эту идею, но ваши программы работает только на одном процессоре. Различные задачи, внутри потоков выполняются на одном ядре, а операционная система управляет, когда ваша программа работает с каким потоком.

Лучшая реальная аналогия, которую я читал, – это введение Async IO в Python: полное прохождение, которое сравнивает этот процесс с шахматистом-гроссмейстером, соревнующимся одновременно со многими противниками. Это всего лишь один человек, но ему нужно переключаться между задачами (играми) и помнить состояние (обычно это называется **state**) для каждой игры.

Поскольку потоки выполняется на одном процессоре, они хорошо подходят для ускорения некоторых задач, но не для всех. Задачи, которые требуют значительных вычислений ЦП и тратят мало времени на ожидание внешних событий, очевидно используя многопоточность не будут выполняться быстрее, вместо этого следует использовать многопроцессорность (multiprocessing).

Архитектура вашей программы при использования многопоточности также может помочь в достижении более чистой архитектуры проекта. Большинство примеров, которые мы рассмотрим в этой статье, не обязательно будут работать быстрее используя потоки. Но использование потоков поможет сделать их архитектуру чище и понятнее.

Потоки (Thread)

Теперь, когда у вас есть представление о том, что такое поток, давайте узнаем, как его использовать. Стандартная библиотека Python предоставляет библиотеку **`threading`** , которая содержит необходимые классы для работы с потоками. Основной класс в этой библиотеки **Thread** .

Чтобы запустить отдельный поток, нужно создать экземпляр потока **Thread** и затем запустить его с помощью метода **.start()** :

```
import logging
import threading
import time
def thread_function(name):
```

```
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,))
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

В основном разделе **__main__** создается и запускается поток:

```
    x = threading.Thread(target=thread_function, args=(1,))
    x.start()
```

Когда вы создаете поток **Thread** , вы передаете ему функцию и список, содержащий аргументы этой функции. В нашем примере мы говорим **Thread** , чтобы он запустил функцию **thread_function()** и передаем ему 1 в качестве аргумента.

Сама по себе функция **thread_function()** мало что делает. Она просто выводит некоторые сообщения с промежутком **time.sleep()** между ними.

Когда вы запустите этот пример как есть (с закомментированной строкой), результат будет выглядеть как то так:

```
$ ./single_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
Thread 1: finishing
```

Вы заметите, что **Thread** завершил работу после выполнения **Main** . Давай рассмотрим что происходит в этот а так же поговорим о загадочной закоментированной строке.

Демоны потоков

В информатике `daemon` **(демон)** – это процесс, который работает в фоновом режиме.

Python потоки имеет особое значение для демонов. Демон потока (или как еще его можно назвать демонический поток) будет остановлен сразу после выхода из программы. Один из способов думать об этих определениях – считать демон потока как потоком, который работает в фоновом режиме, не беспокоясь о его завершении.

Если в программе запущены потоки, которые не являются демонами, то программа будет ожидать завершения этих потоков, прежде чем сможет завершится. Тем не менее, потоки, которые являются демонами, при закрытие программы просто убиваются, в каком бы они состояние ни находились.

Давайте посмотрим повнимательнее на вывод нашего примера. Нам интересны последние две строки. Когда вы запустите программу, вы можете заметить, паузу (около 2 секунд) после того, как **__main__** выведет все свои сообщения и до окончательного завершения потока.

Эта пауза, ожидания завершения **не-демонического потока** . Когда программа на Python завершается, частью процесса завершения работы является очистка подпрограммы потоков.

Если посмотрите на источники библиотеки `threading` Python, вы увидите, что **threading._shutdown()** проходит по всем запущенным потокам и вызывает метод **.join()** для всех, для которых не установлен флаг **daemon** .

Итак, наша программа ожидает выхода, потому что сам поток находится в спящем режиме. Как только он завершит работу и напечатает сообщение, вызовется **.join()** и программа сможет выйти.

Зачастую это именно то, что нам нужно, но есть и другие доступные нам варианты. Давайте сначала повторим наш пример но уже с демонически потоком. Это можно сделать, изменив способ создания **Thread** , добавив флаг **daemon** = **True** :

```
    x = threading.Thread(target=thread_function, args=(1,), daemon=T
```

Когда вы запустите пример сейчас, вы должны увидеть этот вывод:

```
$ ./daemon_thread.py
Main    : before creating thread
Main    : before running thread
```

```
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
```

Разница в том, что последняя строка вывода отсутствует (Thread 1: finishing). **thread_function()** не получил возможность завершиться. Это был демонический поток, поэтому, когда **__main__** достиг конца своего кода и программа захотела завершить работу, демон был просто убит.

join()

Демонические потоки удобны, но что делать, если вы хотите дождаться остановки потока? Как насчет того, что вы хотите сделать это и не выходить из программы? Теперь давайте вернемся к вашей исходной программе и посмотрим на закомментированную строку:

```
    # x.join()
```

Чтобы **указать одному потоку дождаться завершения другого потока** , вам нужно вызывать **.join()** . Если вы раскомментируете эту строку, основной поток остановится и будет ждать завершения потока x.

С каким потом это будет работать, с демоническим потоком или обычным? Оказывается, это не имеет значения. Если вы вызвали **.join()** , этот оператор будет ждать, пока не завершится любой вид потока.

Работа с несколькими потоками

До сих пор мы рассматривали пример только с двумя потоками: основным потоком и с потоком который мы создали с помощью объекта **threading.Thread** .

Зачастую вам нужно будет запускать несколько потоков. Давайте начнем с более сложного способа сделать это, а затем перейдем к более простому способу.

Более сложный способ запуска нескольких потоков – тот, который вы уже знаете:

```
import logging
import threading
import time
def thread_function(name):
    logging.info("Thread %s: starting", name)
```

```
    time.sleep(2)
    logging.info("Thread %s: finishing", name)
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    threads = list()
    for index in range(3):
        logging.info("Main    : create and start thread %d.", index)
        x = threading.Thread(target=thread_function, args=(index,))
        threads.append(x)
        x.start()
    for index, thread in enumerate(threads):
        logging.info("Main    : before joining thread %d.", index)
        thread.join()
        logging.info("Main    : thread %d done", index)
```

Этот код использует тот же механизм, который мы рассмотрели выше, чтобы запустить поток, создать объект **Thread** , а затем вызывается **.start()** . В примере мы размещаем все потоки **Thread** в списке **threads** , для того что бы можно было бы ожидать их позже, используя **.join()** .

После многократного запуска этого примера мы можем увидем примерно такой результат:

```
$ ./multiple_threads.py
Main    : create and start thread 0.
Thread 0: starting
Main    : create and start thread 1.
Thread 1: starting
Main    : create and start thread 2.
Thread 2: starting
Main    : before joining thread 0.
Thread 2: finishing
Thread 1: finishing
Thread 0: finishing
Main    : thread 0 done
Main    : before joining thread 1.
Main    : thread 1 done
Main    : before joining thread 2.
Main    : thread 2 done
```

Если вы внимательно просмотрите выходные данные, вы увидите, что все три потока запускаются в том порядке, в котором мы ожидаем, но заканчиваются в случайном порядке! Многократные запуски покажут различные результаты.

Порядок выполнения потоков определяется операционной системой и его может быть довольно сложно предсказать. Он может (и, вероятно, будет) изменяться от запуска к запуску, поэтому вам следует помнить об этом при разработке алгоритмов, использующих многопоточность.

К счастью, Python предоставляет вам несколько примитивов, которые мы рассмотрим позже, чтобы помочь координировать потоки и запускать их вместе. Но перед этим давайте посмотрим, как немного упростить управление группой потоков.

Использование ThreadPoolExecutor

Есть более простой способ запустить группу потоков, чем тот, который вы видели выше. Он называется **ThreadPoolExecutor** и является частью стандартной библиотеки в **`concurrent.futures`** (начиная с Python 3.2).

Самый простой способ создать его – использовать диспетчер контекста, используя оператор **with** для управления созданием и удалением пула.

Вот **__main__** для последнего примера, переписанного для использования **ThreadPoolExecutor** :

```
import concurrent.futures
[rest of code]
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as exe
        executor.map(thread_function, range(3))
```

Код создает **ThreadPoolExecutor** как менеджер контекста, сообщая ему, сколько рабочих потоков он хочет в пуле. Затем он использует **.map()** для пошагового прохождения итерируемой объкта, в нашем случае **range(3)** , передавая каждый поток в пул.

Конец блока **with** заставляет **ThreadPoolExecutor** выполнить **.join()** для каждого из потоков в пуле. Настоятельно рекомендуется использовать **ThreadPoolExecutor** в качестве диспетчера контекста, чтобы вы никогда не забыли про использование **.join()** .

Примечание . Использование **ThreadPoolExecutor** может привести к ошибкам. Например, если вы вызываете функцию, которая не принимает параметров, но передаете ей параметры в **.map()** , в этом случае поток выдаст исключение. К сожалению, **ThreadPoolExecutor** скрывает это исключение, и (в случае выше) программа завершается без вывода. Это может стать довольно запутанным при отладке.

Запуск нашего исправленного примера приведет к выводу, который может выглядит следующим образом:

```
$ ./executor.py
Thread 0: starting
Thread 1: starting
Thread 2: starting
Thread 1: finishing
Thread 0: finishing
Thread 2: finishing
```

Опять же, обратите внимание, как закончился **Thread 1** и **Thread 0** . Планирование потоков выполняется операционной системой и не будет предсказуемым.

Условия гонки (Race Conditions)

Прежде чем перейти к некоторым другим функциям потоков, давайте немного поговорим об одной из самых сложных проблем, с которыми вы столкнетесь при написании многопоточных программ: **условия гонки** (race conditions).

После того, как мы расскажем, о том что такое состояние гонки, и рассмотрим, то что происходит в этот момент, мы перейдем к некоторым из объектов, предоставляемых стандартной библиотекой, чтобы предотвратить возникновение условий гонки.

Условия гонки могут возникать, когда два или более потока обращаются к общему фрагменту данных или ресурсу. В следующим примере мы создадим состояние гонки, но имейте в виду, что часто условие гонки не так очевидны и они могут привести к сбивающим с толку результатам. Как вы можете себе представить, все это делает их довольно сложными для отладки.

В нашем примере мы напишем класс, который будет имитировать обновление базы данных. Наша база данных будет называться **FakeDatabase** и у нее будет методы **.__init__()** и **.update()** :

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
    def update(self, name):
        logging.info("Thread %s: starting update", name)
        local_copy = self.value
        local_copy += 1
        time.sleep(0.1)
        self.value = local_copy
        logging.info("Thread %s: finishing update", name)
```

FakeDatabase отслеживает одно число: **value** . Это будут общие данные, на которых мы увидим состояние гонки.

.__init__() просто инициализирует значение **value** в ноль.

.update () выглядит немного странно. Этот метод имитирует чтение значения из базы данных, делается некоторые вычисления с числом, а затем записывает новое значение обратно в базу данных.

В нашем случае чтение из базы данных означает просто копирование **self.value** в локальную переменную. Вычисление состоит только в том, чтобы добавить единицу к значению, а затем вызов паузы методом **.sleep()** на некоторое время. Наконец, мы записываем значение обратно, копируя локальное значение обратно в . **value** .

Вот как мы будем использовать **FakeDatabase** :

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    database = FakeDatabase()
    logging.info("Testing update. Starting value is %d.", database.v
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as exe
        for index in range(2):
            executor.submit(database.update, index)
    logging.info("Testing update. Ending value is %d.", database.val
```

Программа создает **ThreadPoolExecutor** с двумя потоками, а затем вызывает метод **.submit()** для каждого из них, приказывая им запустить **database.update()** .

Метод **.submit()** имеет формат, который позволяет передавать как позиционные, так и именованные аргументы функции, выполняющейся в потоке:

```
.submit(function, *args, **kwargs)
```

index передается в качестве первого и единственного позиционного аргумента в **database.update()** . Позже в этой статье вы увидите, что вы можете передать несколько аргументов аналогичным образом.

Поскольку каждый поток выполняет **.update()** , а **.update()** добавляет один к значению **.value** , мы можем ожидать, что **database.value** будет равен 2, когда он будет выведен на экран в конце выполнения программы. Но мы бы не рассматривали бы этот пример, если бы это было так. Если вы запустите приведенный выше код, результат будет выглядеть как то так:

```
$ ./racecond.py
Testing unlocked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
Testing unlocked update. Ending value is 1.
```

Возможно, вы ожидали, что это произойдет, но давайте посмотрим на детали того, что на самом деле здесь происходит, так как это облегчит понимание проблемы.

Один поток

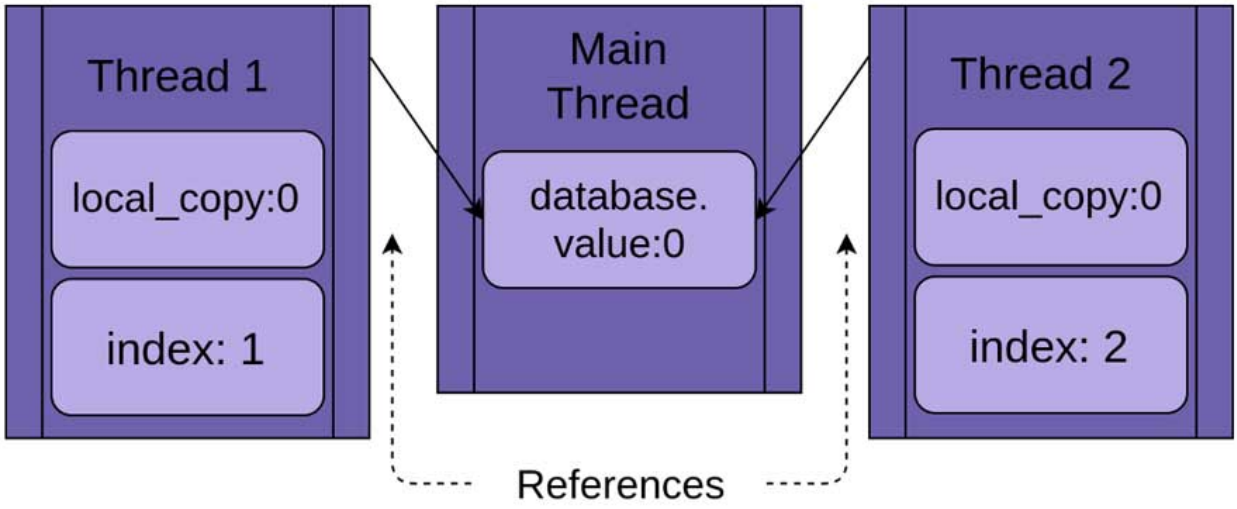
Прежде чем углубиться в эту проблему с использованием двух потоков, давайте вернемся назад и немного поговорим о некоторых деталях работы потоков.

Мы не будем углубляться во все детали, так как это не важно на данном этапе. Мы также упростим несколько вещей, которые не будут технически точными, но дадут вам правильное представление о том, что происходит.

Когда вы указываете **ThreadPoolExecutor** запустить каждый поток, вы указываете, какую функцию запускать и какие параметры передавать в нее: **executor.submit (database.update, index)** .

Результатом этого кода будет то, что каждый из потоков в пуле будет вызывать **database.update(index)** . Обратите внимание, что **database** является ссылкой на экземпляр класса **FakeDatabase** , созданный в **__main__** . А вызов **.update()** для этого объекта вызывает метод экземпляра этого класса.

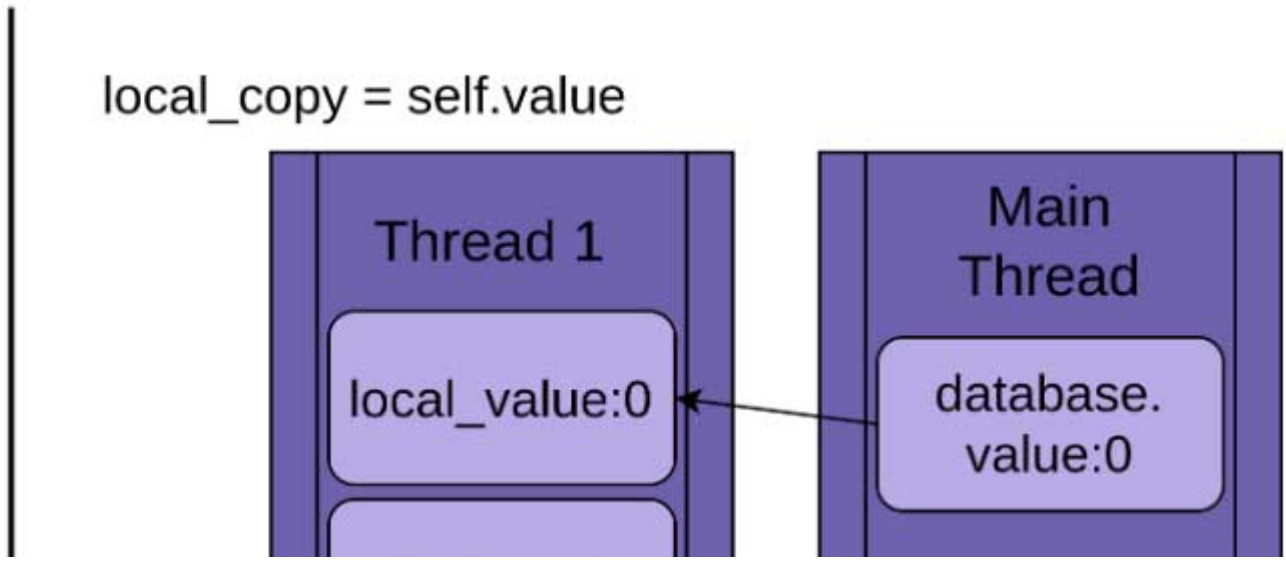
Каждый поток будет иметь ссылку на один и тот же объект **FakeDatabase** , **database** . Каждый поток также будет иметь уникальное значение **value** и индекс **index** :



Когда поток начинает выполнять **.update()** , он имеет свою собственную версию всех данных, локальных для функции. В случае **.update()** это **local_copy** . Обычно это определенно хорошо. В противном случае два потока, выполняющие одну и ту же функцию, всегда будут смешивать друг друга. Но это так же означает, что все переменные (локальные) для функции являются поточно-ориентированными (**thread-safe**).

Теперь рассмотрим, что происходит, если мы запустим вышеописанную программу с одним потоком и одним вызовом **.update()** .

Изображение ниже показывает выполнение **.update()** , если запущен только один поток. Слева показан поток **Thread 1** , в нем значения в **local_value** потока и совместно используемой **database.value** :



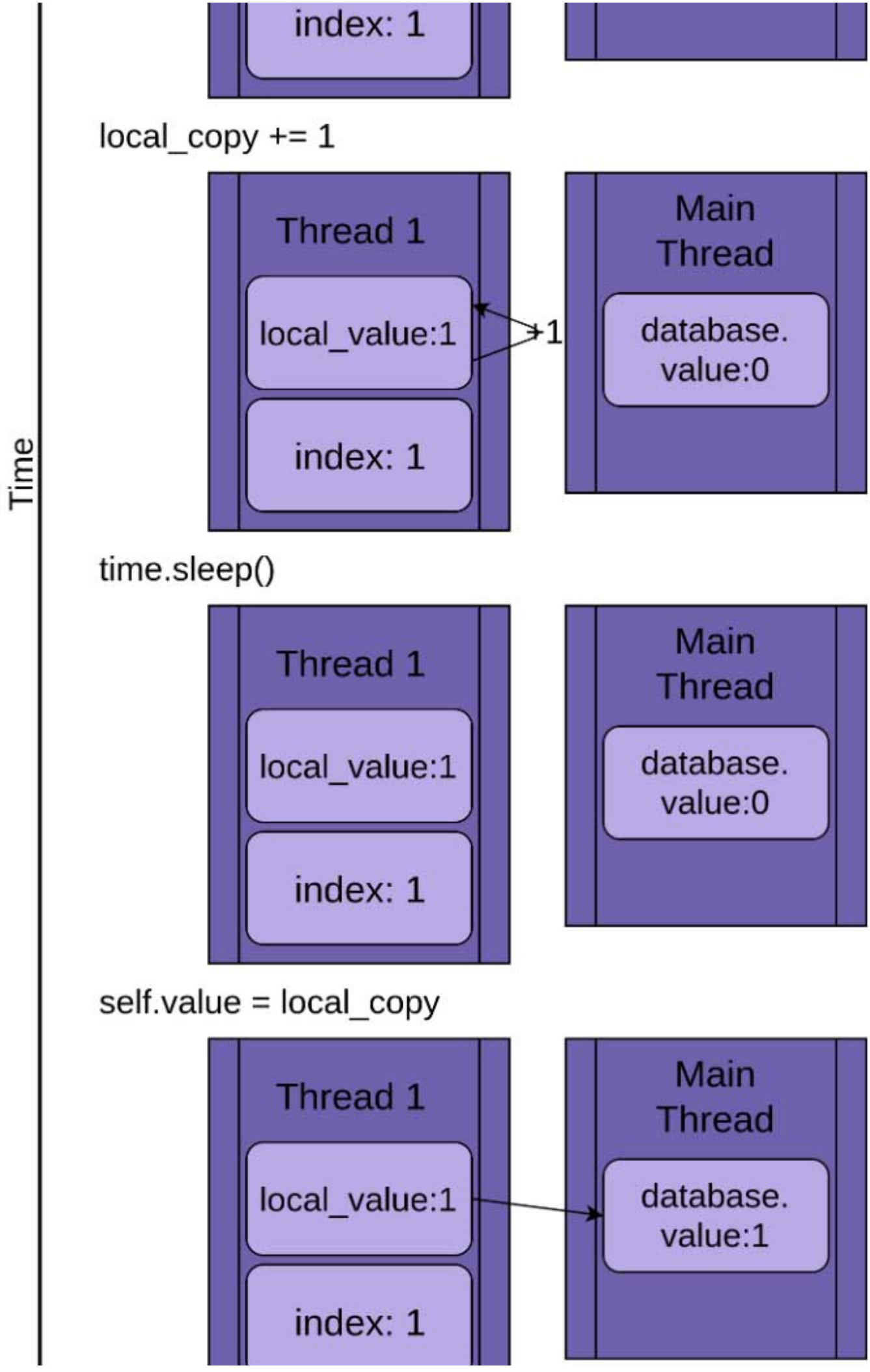

index: 1
local_copy += 1
Thread 1
local_value:1
index: 1
1
Main Thread
database. value:0
Time
time.sleep()
Thread 1
local_value:1
index: 1
Main Thread
database. value:0
self.value = local_copy
Thread 1
local_value:1
index: 1
Main Thread
database. value:1

Диаграмма построена так, что время увеличивается при движении сверху вниз. Он начинается, когда начинается **Thread 1** , и заканчивается, когда он завершается.

Когда **Thread 1** запускается, **FakeDatabase.value** равен нулю. Первая строка кода в методе **local_copy = self.value** копирует нулевое значение в локальную переменную. Затем увеличивается значение **local_copy** с помощью оператора **local_copy + = 1** . Вы можете увидеть **.value** в **Thread 1** , установленной в единицу.

На следующим шаге вызывается **time.sleep()** , который останавливает текущий поток и позволяет другим потокам работать. Так как в этом примере есть только один поток, это не имеет никакого эффекта.

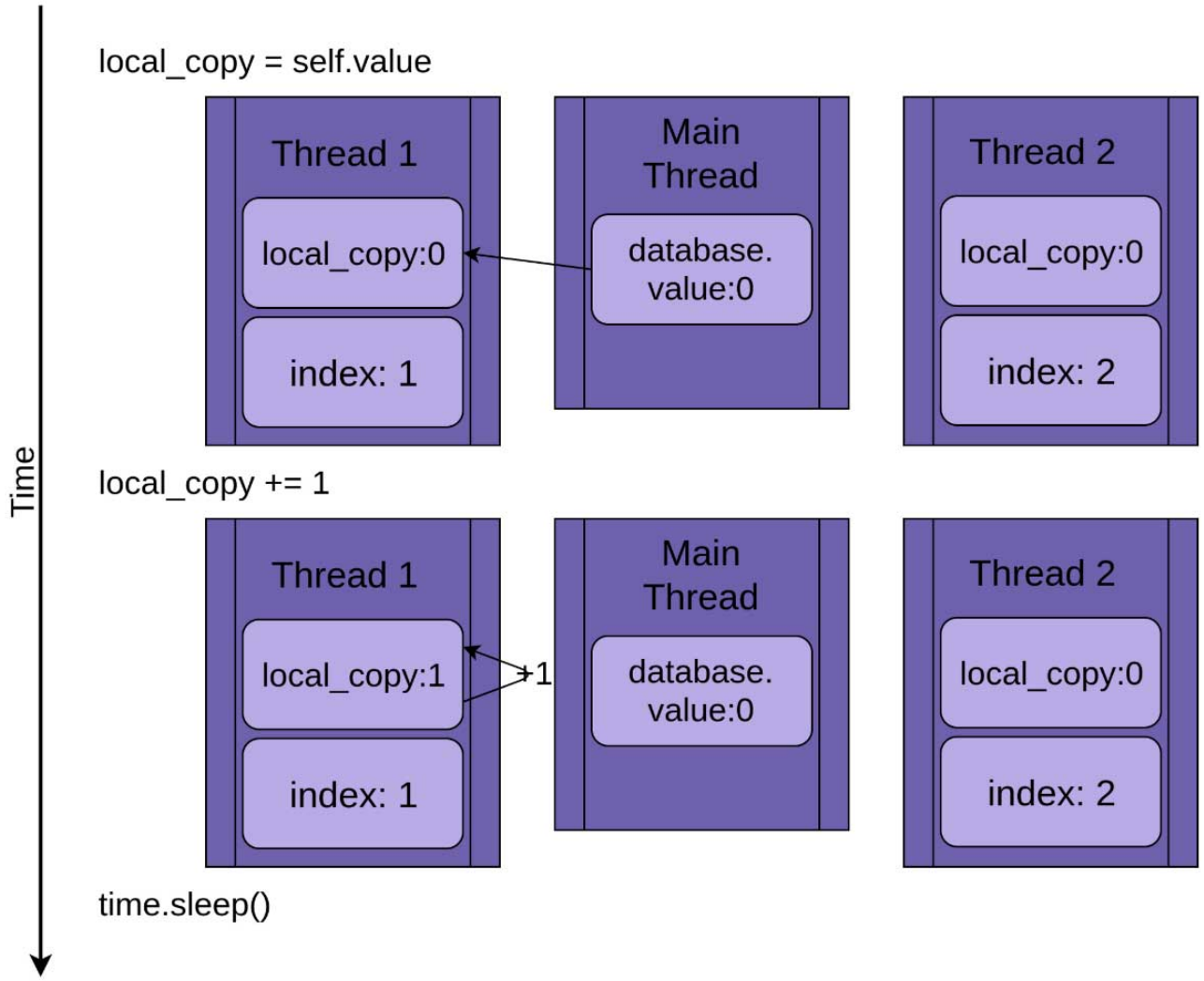
Когда **Thread 1** просыпается и продолжает работу, он копирует новое значение из **local_copy** в **FakeDatabase.value** , и затем поток завершается. Вы можете видеть, что значение **database.value** установлено в единицу.

Все как обычно. Вы запустили **.update()** один раз, и **FakeDatabase.value** был увеличен до единицы.

Два потока

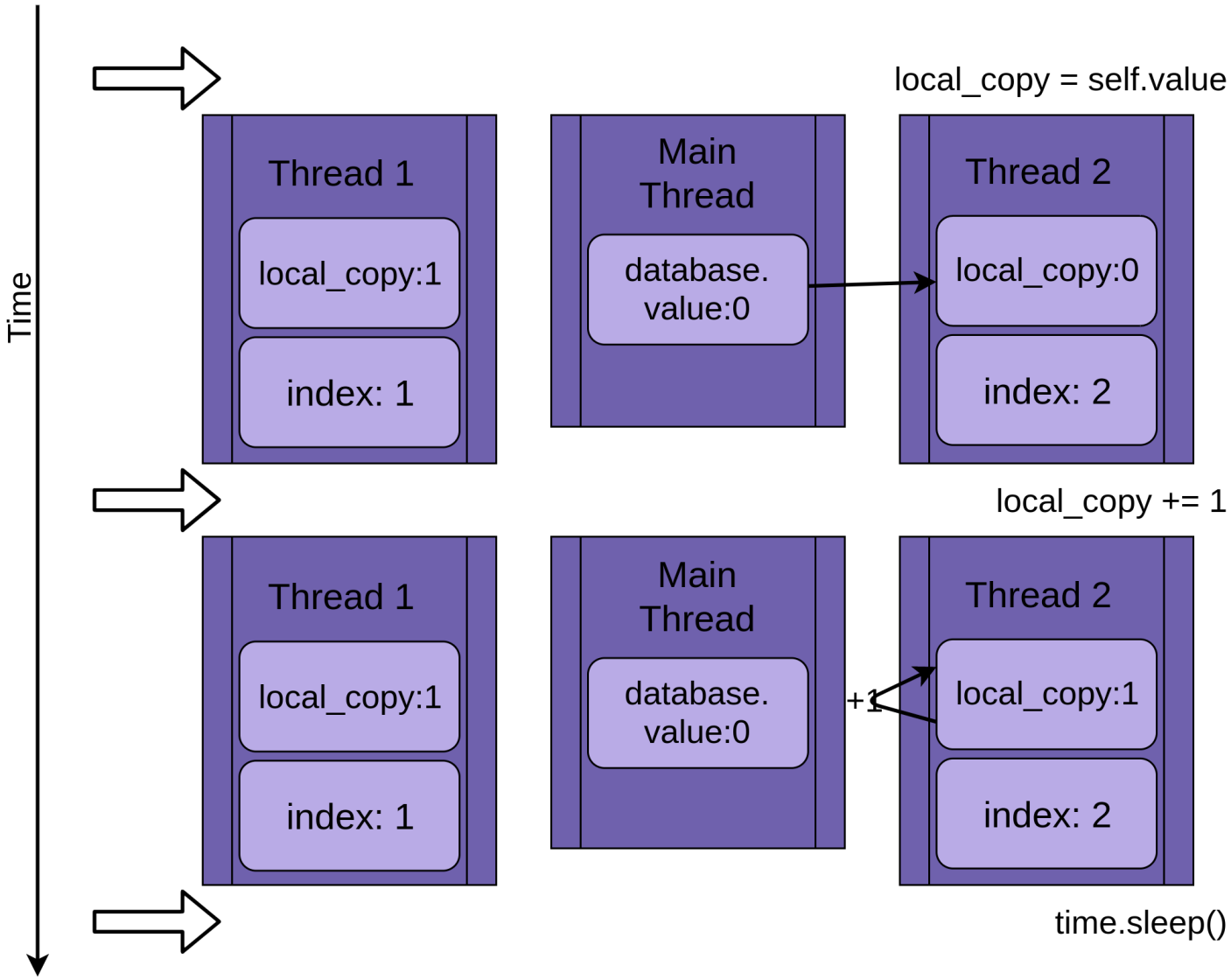
Возвращаясь к состоянию гонки, два потока работают одновременно, но не одномоментно. Каждый из них будет иметь свою собственную версию **local_copy** и при этом указывают на одну и ту же базу данных. Именно этот общий объект базы данных будет вызывать проблемы.

Программа запускается с **Thread 1** , выполняющим **.update()** :



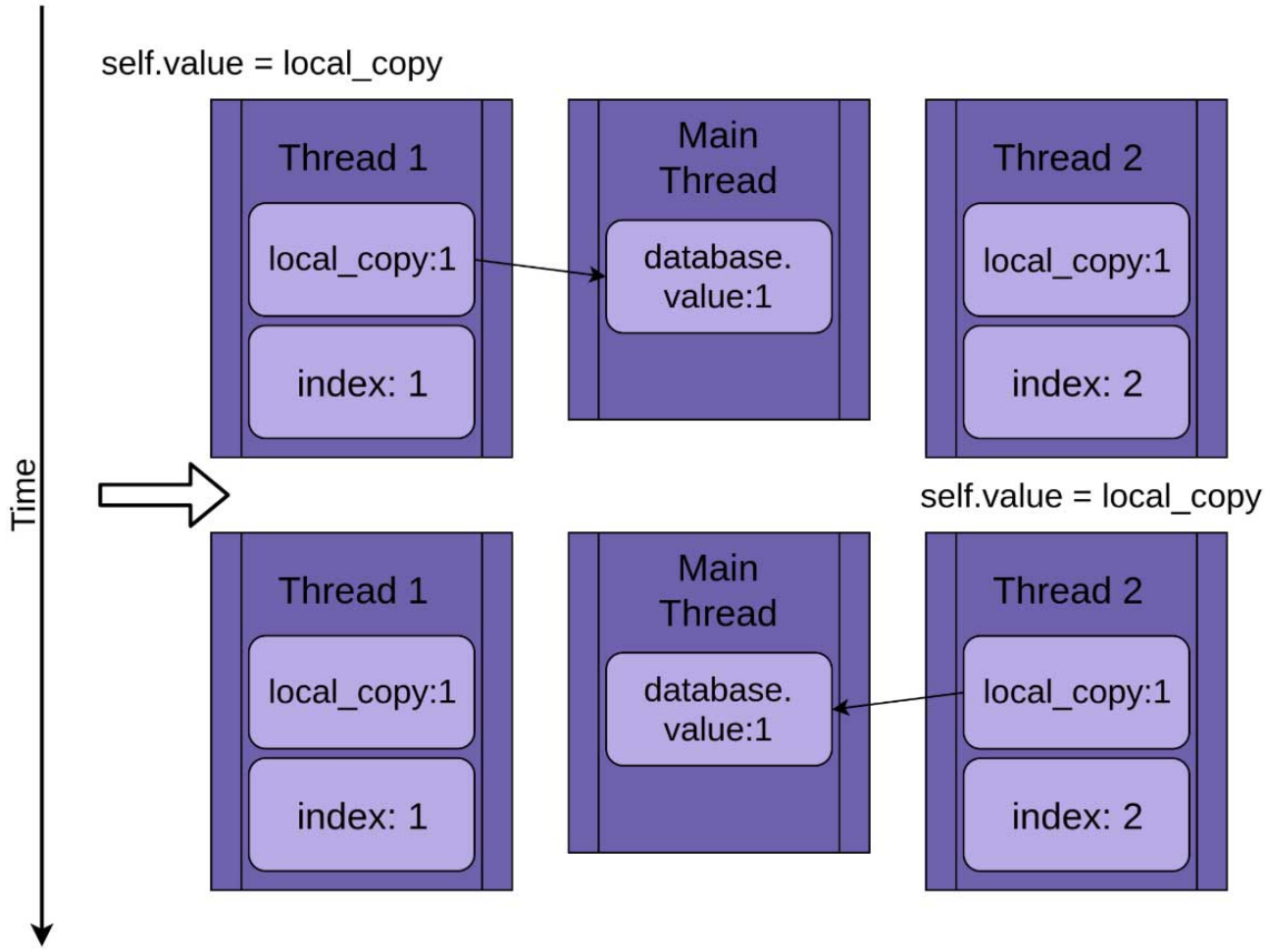
Когда **Thread 1** вызывает **time.sleep()** , он позволяет другому потоку начать работу. Здесь все становятся сложнее.

Thread 2 запускается и выполняет те же операции. Он также копирует **database.value** в свою приватную **local_copy** , в то время как общий **database.value** еще не обновлен:



Когда **Thread 2** , наконец, переходит в спящий режим, общий **database.value** по-прежнему пока еще не изменяется, и обе частные версии **local_copy** имеют одно значение один.

Thread 1 теперь просыпается и сохраняет свою версию **local_copy** , а затем завершает свою работу, давая **Thread 2** последний шанс на запуск. **Thread 2** не знает, что **Thread 1** отработал и обновил **database.value** , пока он еще спал. Теперь он сохраняет свою версию **local_copy** в **database.value** , также устанавливая ее в единицу:



Два потока имеют чередующийся доступ к одному общему объекту, перезаписывая результаты друг друга. Подобные условия могут возникнуть, когда один поток освобождает память или закрывает дескриптор файла, прежде чем другой поток завершит доступ к нему.

Почему это пример типичной ситуации

Приведенный выше пример создан для того, чтобы условия гонки возникали при каждом запуске вашей программы. Поскольку операционная система может поменять поток в любое время, то возможно прерывание работы любой вычислительной операции, например такой как **x = x + 1** , после того, как было прочитано значение **x** , но до того, как было записано увеличенное значение.

Теперь, когда вы увидели состояние гонки в действии, давайте выясним, как решить эту проблему!

Базовая синхронизация с использованием блокировки

Есть много способов избежать создания условия гонки. Вы не будем рассматривать их все, но есть пара, которая часто используется. Давайте начнем с блокировки (**Lock**).

Чтобы решить вышеупомянутое условие гонки, нам нужно найти способ разрешить только один поток за раз в операции чтения-изменения-записи. Наиболее распространенный способ сделать это называется **Lock** in Python. В некоторых других языках эта же идея называется **mutex** (мьютексом). **Mutex** происходит от MUTual EXclusion, что то же самое что и Lock.

Lock – это объект, который действует как коридор в зал. Только один поток за раз может использовать Lock. Любой другой поток, который захочет использовать **Lock** , должен ждать, пока текущий владелец **Lock** не откажется от нее.

Основными функциями для этого являются **.acquire()** и **.release()** . Чтобы получить блокировку, потоку нужно вызвать **my_lock.acquire()** . Если блокировка уже удерживается, вызывающий поток будет ждать, пока она не будет снята. Здесь есть важный момент. Если один поток получает блокировку, но никогда не возвращает ее, ваша программа зависнет.

К счастью, в Python **Lock** может также работать как менеджер контекста, так что вы можете использовать его с **with** , и он будет освобожден автоматически, когда блок **with** по какой-либо причине завершится.

Давайте посмотрим на базу данных **FakeDatabase** с использованием **Lock** . Вызывающая функция остается прежней:

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()
    def locked_update(self, name):
        logging.info("Thread %s: starting update", name)
        logging.debug("Thread %s about to lock", name)
        with self._lock:
            logging.debug("Thread %s has lock", name)
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.debug("Thread %s about to release lock", name)
        logging.debug("Thread %s after release", name)
        logging.info("Thread %s: finishing update", name)
```

Помимо добавления нескольких журналов отладки, чтобы мы могли видеть блокировку, большое изменение здесь заключается в добавлении переменной с именем **._lock** , которая является объектом **threading.Lock()** .
Этот **._lock** инициализируется в разблокированном состоянии, блокируется и освобождается оператором **with** .

Здесь стоит отметить, что поток, выполняющий эту функцию, будет удерживать эту блокировку до тех пор, пока он полностью не завершит обновление базы данных. В этом случае это означает, что он будет удерживать блокировку, пока он копирует, обновляет, спит, а затем записывает значение обратно в базу данных.

Если вы запустите эту версию с журналированием, установленным на уровень предупреждения, вы увидите что типа такого:

```
$ ./fixrace.py
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

Вы можете включить полное ведение журнала, установив уровень **DEBUG** , добавив этот оператор после настройки вывода журнала в **__main__** :

```
    logging.getLogger().setLevel(logging.DEBUG)
```

Запуск этой программы с включенным ведением журнала DEBUG будет выглядет следующим образом:

```
$ ./fixrace.py
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 0 about to lock
Thread 0 has lock
Thread 1: starting update
Thread 1 about to lock
Thread 0 about to release lock
Thread 0 after release
Thread 0: finishing update
Thread 1 has lock
Thread 1 about to release lock
Thread 1 after release
```

```
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

В этом выводе вы можете видеть, что **Thread 0** получает блокировку и все еще удерживает ее, когда переходит в режим сна. Затем **Thread 1** запускается и пытается получить такую же блокировку. Поскольку **Thread 0** все еще удерживает его, **Thread 1** должен ждать. Это взаимное исключение, которое обеспечивает **Lock** .

Во многих примеров в оставшейся части этой статьи уровень ведение журнала установлен в **WARNING** и **DEBUG** . Обычно мы показываем только выходные данные уровня **WARNING** , поскольку журналы **DEBUG** могут быть довольно длинными.

Deadlock

Прежде чем двигаться дальше, мы должны рассмотреть общую проблему при использовании **Lock** . Как вы видели, если блокировка уже была получена, второй вызов **.acquire()** будет ожидать, пока поток, удерживающий блокировку, не вызовет **.release()** . Как вы думаете, что происходит, когда вы запускаете этот код:

```
import threading
l = threading.Lock()
print("before first acquire")
l.acquire()
print("before second acquire")
l.acquire()
print("acquired lock twice")
```

Когда программа вызывает **l.acquire()** во второй раз, она зависает в ожидании снятия блокировки. В этом примере вы можете устранить взаимоблокировку, удалив второй вызов, но взаимоблокировки обычно происходят из одной из двух причин:

1. Ошибка реализации, при которой блокировка не освобождается должным образом
2. Проблема проектирования, когда служебная функция должна вызываться функциями, которые могут иметь или не иметь блокировку

Первая ситуация иногда случается, но использование блокировки в качестве диспетчера контекста значительно снижает вероятность ее появления. Рекомендуется по возможности писать код с использованием менеджеров контекста, поскольку они помогают избежать ситуаций, когда исключение всегда вызовет **.release()** .

Проблема дизайна может быть немного сложнее в некоторых языках. К счастью, в потоках в Python есть второй объект, называемый **RLock** , который предназначен именно для этой ситуации. Это позволяет потоку вызывать **.acquire()** **RLock** несколько раз, прежде чем он вызовет **.release()** . Но при этому поток все еще должен вызывать **.release()** столько же раз, сколько вызывал **.acquire()** .

Lock и **RLock** – это два основных инструмента, используемых в многопоточном программировании для предотвращения условий гонки. Есть несколько других, которые работают по-другому. Прежде чем мы рассмотрим их, давайте перейдем к немного другой проблемной области.

Потоки Производитель-Потребитель (Producer-Consumer Threading)

Проблема «производитель-потребитель» (Producer-Consumer Problem) – это стандартная проблема в области компьютерных наук, используемая при рассмотрение проблем с многопоточностью или синхронизацией процессов.

В нашем следующем примере мы рассмотрим программу, которая должна читать сообщения из сети и записывать их на диск. Программа не может запрашивать сообщения, когда захочет. Она должна прослушивать и принимать сообщения по мере их поступления. Сообщения не будут поступать в обычном темпе, а будут приходить очередями. Эта часть программы называется продюсером (**producer**).

С другой стороны, когда у вас есть сообщение, вам нужно записать его в базу данных. Доступ к базе данных медленный, но достаточно быстрый, чтобы соответствовать среднему темпу сообщений. Это не достаточно быстро, чтобы не отставать, когда приходит поток сообщений. Эта часть является потребителем (**consumer**).

Между производителем и потребителем мы создадим **конвейер** (**Pipeline**), который будет частью, которая будет меняться, когда мы разберем различных объектах синхронизации.
Это основной макет. Давайте посмотрим на решение с помощью блокировки. Это

решение не будет работать идеально, но оно использует инструменты, которые мы уже знаем, так что это хорошее выбор для начала.

Producer-Consumer используя Lock

Поскольку это статья о потоках в Python и вы только что прочитали о примитиве **Lock** , давайте попробуем решить эту проблему с двумя потоками, используя **Lock** .

Общий дизайн состоит в том, что существует поток производителя (**producer**), который читает из фальшивой сети и помещает сообщение в Pipeline:

```
SENTINEL = object()
def producer(pipeline):
    """Pretend we're getting a message from the network."""
    for index in range(10):
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        pipeline.set_message(message, "Producer")
    # Send a sentinel message to tell consumer we're done
    pipeline.set_message(SENTINEL, "Producer")
```

Чтобы создать поддельное сообщение, производитель получает случайное число от одного до ста. Затем он вызывает **.set_message()** в конвейере, чтобы отправить его потребителю.

Производитель (**producer**) также использует значение **SENTINEL** , чтобы дать сигнал потребителю остановиться после того, как он отправил десять значений. Так делать немного нехорошо, но не волнуйтесь, вы увидите способы избавиться от этого значения **SENTINEL** после проработки этого примера.

На другой стороне pipeline находится потребитель (consumer):

```
def consumer(pipeline):
    """ Pretend we're saving a number in the database. """
    message = 0
    while message is not SENTINEL:
        message = pipeline.get_message("Consumer")
        if message is not SENTINEL:
            logging.info("Consumer storing message: %s", message)
```

Потребитель читает сообщение из конвейера и записывает его в поддельную базу данных, которая в этом случае просто выводит значение на дисплей. Если он получает значение SENTINEL, он возвращается из функции, которая завершит поток.

Прежде чем вы посмотрите на действительно интересную часть, **Pipeline** , вот раздел **__main__** , который порождает эти потоки:

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    # logging.getLogger().setLevel(logging.DEBUG)
    pipeline = Pipeline()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as exe
        executor.submit(producer, pipeline)
        executor.submit(consumer, pipeline)
```

Этот код должен выглядеть довольно знакомым, поскольку он похож на код **__main__** в предыдущих примерах.

Помните, что вы можете включить ведение журнала **DEBUG** , чтобы увидеть все сообщения журнала, раскомментировав эту строку:

```
    # logging.getLogger().setLevel(logging.DEBUG)
```

Возможно, стоит просмотреть сообщения журнала **DEBUG** , чтобы точно определить, где каждый поток получает и снимает блокировки.

Теперь давайте посмотрим на конвейер, который передает сообщения от производителя к потребителю:

```
class Pipeline:
    """Class to allow a single element pipeline between producer and
    """
    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()
    def get_message(self, name):
        logging.debug("%s:about to acquire getlock", name)
        self.consumer_lock.acquire()
        logging.debug("%s:have getlock", name)
```

```
        message = self.message
        logging.debug("%s:about to release setlock", name)
        self.producer_lock.release()
        logging.debug("%s:setlock released", name)
        return message
    def set_message(self, message, name):
        logging.debug("%s:about to acquire setlock", name)
        self.producer_lock.acquire()
        logging.debug("%s:have setlock", name)
        self.message = message
        logging.debug("%s:about to release getlock", name)
        self.consumer_lock.release()
        logging.debug("%s:getlock released", name)
```

Ого! Это много кода. Довольно большой процент от этого – просто логгирование операций, чтобы было легче видеть, что происходит, когда вы запускаете их. Вот тот же код со всеми удаленными инструкциями регистрации:

```
class Pipeline:
    """Class to allow a single element pipeline between producer and
    """
    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()
    def get_message(self, name):
        self.coNsumer_lock.acquire()
        message = self.message
        self.producer_lock.release()
        return message
    def set_message(self, message, name):
        self.producer_lock.acquire()
        self.message = message
        self.consumer_lock.release()
```

Это кажется немного более управляемым. Конвейер в этой версии вашего кода состоит из трех элементов:

1. **`.message`** сохраняет сообщение для передачи.
2. **`.producer_lock`** является объектом **threading.Lock** , который ограничивает доступ к сообщению потока производителя
3. **`.consumer_lock`** также является **threading.Lock** , который ограничивает доступ к сообщению потока потребителя.

__init__() инициализирует эти три элемента а затем вызывает **.acquire()** для **.consumer_lock** . Это состояние, в котором мы хотите начать работку программы. Производителю разрешено добавлять новое сообщение, а потребителю нужно подождать, пока сообщение не появится

.get_message() и **.set_messages()** почти противоположны. **.get_message()** вызывает **.acquire()** для **customer_lock** . Это вызов, который заставит потребителя ждать, пока сообщение не будет готово.

Как только потребитель получает **.consumer_lock** , он копирует значение в **.message** и затем вызывает **.release()** для **.producer_lock** . Освобождение этой блокировки позволяет производителю вставить следующее сообщение в конвейер.

Прежде чем перейти к **.set_message()** , в **.get_message()** происходит нечто особое, что довольно легко пропустить. Может показаться заманчивым избавиться от сообщения и просто завершить функцию возвращением **self.message** . Посмотрите, сможете ли вы понять, почему мы не хотим этого делать, прежде чем двигаться дальше.

Вот ответ. Как только потребитель вызывает **.producer_lock.release()** , он может быть заменен, и производитель может начать работать. Это может произойти до того, как вызовется **.release()** ! Это означает, что существует небольшая вероятность того, что когда функция вернет **self.message** , это может быть следующее сгенерированное сообщение, поэтому мы можем потерять первое сообщение. Это еще один пример состояния гонки.

Переходя к **.set_message()** , вы можете увидеть противоположную сторону транзакции. Производитель передает сообщение. Он получит **.producer_lock** , установит **.message** и вызовет **.release()** для параметра **customer_lock** , что позволит потребителю прочитать это значение.

Давайте запустим код с журналированием, установленным **WARNING** , и посмотрим, как это выглядит:

```
$ ./prodcom_lock.py
Producer got data 43
Producer got data 45
Consumer storing data: 43
Producer got data 86
Consumer storing data: 45
Producer got data 40
Consumer storing data: 86
```

```
Producer got data 62
Consumer storing data: 40
Producer got data 15
Consumer storing data: 62
Producer got data 16
Consumer storing data: 15
Producer got data 61
Consumer storing data: 16
Producer got data 73
Consumer storing data: 61
Producer got data 22
Consumer storing data: 73
Consumer storing data: 22
```

Поначалу может показаться странным, что производитель получает два сообщения еще до того, как потребитель запускается. Если вы посмотрите на производителя и **.set_message()** , вы заметите, что единственное место, где он будет ожидать блокировки, это когда он пытается поместить сообщение в конвейер. Это делается после того, как производитель получает сообщение и регистрирует его наличие.

Когда производитель пытается отправить второе сообщение, он вызовет **.set_message()** во второй раз и заблокирует поток.

Операционная система может поменять потоки в любое время, но, как правило, она позволяет каждому потоку иметь достаточно времени для запуска, прежде чем его менять. Вот почему производитель обычно работает, пока не блокируется во втором вызове **.set_message()** .

Однако, как только поток заблокирован, операционная система всегда поменяет его и найдет другой поток для запуска. В этом случае единственным другим потоком, который должен что-либо делать, является **производитель** .

Производитель вызывает **.get_message()** , который читает сообщение и вызывает **.release()** для **.producer_lock** , что позволяет производителю запускаться снова при следующей перестановке потоков.

Обратите внимание, что первое сообщение было 43, и это именно то, что получил потребитель, даже если производитель уже сгенерировал сообщение 45.

Хотя все это работает для этого ограниченного примера, этот код не является отличным решением проблемы производитель-потребитель в целом, поскольку он

допускает только одно значение в конвейере за раз. Когда продюсер получит пакет сообщений, ему некуда будет их отправлять.

Давайте перейдем к лучшему способу решения этой проблемы, используя **Queue** (Очередь).

Producer-Consumer с использованием Queue

Если вы хотите иметь возможность обрабатывать более одного значения в конвейере одновременно, вам понадобится структура данных для конвейера, которая позволяет увеличивать и уменьшать число при резервном копировании данных от производителя.

Стандартная библиотека Python имеет модуль **queue** , который, в свою очередь, имеет класс **Queue** . Давайте изменим конвейер, так чтобы использовать очередь вместо просто переменной, защищенных **Lock** . Мы также будем использовать другой способ остановить рабочие потоки, используя другой примитив из threading Python – **Event** .

Давайте начнем с **Event** . Объект **threading.Event** позволяет одному потоку сигнализировать о событии, в то время как другие потоки будут ожидать этого события. Основаная идея в этом коде состоит в том, что потокам, ожидающим событие, не обязательно нужно останавливать то, что они делают, они могут просто проверять состояние **Event** время от времени.

Триггером запуском события может быть много вещей. В нашем примере основной поток просто будет некоторое время спать, а затем запустит **.set()** :

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    # logging.getLogger().setLevel(logging.DEBUG)

    pipeline = Pipeline()
    event = threading.Event()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as
        executor.submit(producer, pipeline, event)
        executor.submit(consumer, pipeline, event)

        time.sleep(0.1)
```

```
14          logging.info("Main: about to set event")
15          event.set()
```

Единственными изменениями здесь являются создание объекта события в строке 8, передача события в качестве параметра в строках 10 и 11 и заключительный раздел в строках 13-15, который запускает паузу на секунду, регистрируют сообщение и затем вызывают **.set()** для запуска события.

Производителя тоже не пришлось сильно менять:

```
1 def producer(pipeline, event):
2     """Pretend we're getting a number from the network."""
3     while not event.is_set():
4         message = random.randint(1, 101)
5         logging.info("Producer got message: %s", message)
6         pipeline.set_message(message, "Producer")
7 
8     logging.info("Producer received EXIT event. Exiting")
```

Теперь он будет зацикливаться до тех пор, пока не увидит, что событие было установлено в строке 3. Он также больше не помещает значение **SENTINEL** в конвейер.

Потребителя пришлось немного изменить:

```
def consumer(pipeline, event):
    """ Pretend we're saving a number in the database. """
    while not event.is_set() or not pipeline.empty():
        message = pipeline.get_message("Consumer")
        logging.info(
            "Consumer storing message: %s  (queue size=%s)",
            message,
            pipeline.qsize(),
        )
    logging.info("Consumer received EXIT event. Exiting")
```

В то время как мы должны были извлечь код, связанный со значением **SENTINEL** , нам пришлось создать несколько более сложное условие **while** . Он не только выполняет цикл до тех пор, пока не будет установлено событие, но также должен продолжать цикл до тех пор, пока конвейер не будет очищен.

Перед тем, как потребитель завершит работу, убедитесь, что очередь пуста, чтобы избежать еще одной забавной проблемы. Если потребитель действительно завершает

работу, пока в конвейере есть сообщения, могут произойти две неприятные вещи. Во-первых, мы можем потерять последние сообщения, но более серьезным является то, что производитель может закциклиться из-за **producer_lock** и никогда не завершится.

Это происходит, если событие запускается после того, как производитель проверил условие **.is_set()** , но до того, как оно вызвало **pipe.set_message()** .

Если это произойдет, производитель может проснуться и выйти, удерживая **.producer_lock** . Затем производитель попытается **.acquire()** для **.producer_lock** , но потребитель уже вышел и никогда не выполнит **.release()** .

Pipeline сильно изменился:

```
class Pipeline(queue.Queue):
    def __init__(self):
        super().__init__(maxsize=10)
    def get_message(self, name):
        logging.debug("%s:about to get from queue", name)
        value = self.get()
        logging.debug("%s:got %d from queue", name, value)
        return value
    def set_message(self, value, name):
        logging.debug("%s:about to add %d to queue", name, value)
        self.put(value)
        logging.debug("%s:added %d to queue", name, value)
```

Вы можете видеть, что класс **Pipeline** является подклассом **queue.Queue** . Очередь имеет необязательный параметр при инициализации для указания максимального размера очереди.

Если вы назначите положительное число для **maxsize** , оно ограничит очередь этим количеством элементов, в результате чего запуск **.put()** будет невозможен, если число элементов будет больше **maxsize** . Если вы не укажете **maxsize** , очередь будет увеличится до предела памяти вашего компьютера.

.get_message() и **.set_message()** стали намного меньше. Они в основном заключают в очередь **.get()** и **.put()** . Вы можете быть удивлены, куда ушел весь код блокировки, который предотвращает возникновение состояний в потоках.

Разработчики ядра, написавшие стандартную библиотеку, знали, что очередь часто используется в многопоточных средах, и включали весь этот код блокировки в самой очереди. Очередь потокобезопасна.

Запуск этой программы выглядит следующим образом:

```
$ ./prodcom_queue.py
Producer got message: 32
Producer got message: 51
Producer got message: 25
Producer got message: 94
Producer got message: 29
Consumer storing message: 32 (queue size=3)
Producer got message: 96
Consumer storing message: 51 (queue size=3)
Producer got message: 6
Consumer storing message: 25 (queue size=3)
Producer got message: 31
[many lines deleted]
Producer got message: 80
Consumer storing message: 94 (queue size=6)
Producer got message: 33
Consumer storing message: 20 (queue size=6)
Producer got message: 48
Consumer storing message: 31 (queue size=6)
Producer got message: 52
Consumer storing message: 98 (queue size=6)
Main: about to set event
Producer got message: 13
Consumer storing message: 59 (queue size=6)
Producer received EXIT event. Exiting
Consumer storing message: 75 (queue size=6)
Consumer storing message: 97 (queue size=5)
Consumer storing message: 80 (queue size=4)
Consumer storing message: 33 (queue size=3)
Consumer storing message: 48 (queue size=2)
Consumer storing message: 52 (queue size=1)
Consumer storing message: 13 (queue size=0)
Consumer received EXIT event. Exiting
```

Если вы прочитаете вывод в моем примере, вы увидите, что происходит кое-что интересное. Справа вверху видно, что производитель должен создать пять сообщений и поместить четыре из них в очередь. Он был заменен операционной системой до того, как смог разместить пятую.

Затем потребитель запустился и вытащил первое сообщение. Затем распечатал это сообщение так же как размер очереди:

```
Consumer storing message: 32 (queue size=3)
```

Вот откуда мы знаем, что пятое сообщение еще не попало в конвейер. После удаления одного сообщения очередь уменьшается до трех значений. Мы также знаем, что очередь может содержать десять сообщений, поэтому поток производителя не был заблокирован этой очередью. Он был заменен операционной системой.

Примечание : ваш вывод будет другим. Вывод будет меняться от запуска к запуску. Это забавная часть работы с потоками!

Когда программа начинает закрываться, мы видим что основной поток, генерирующий событие, которое приводит к немедленному завершению работы производителя. Потребителю все еще предстоит выполнить много работы, поэтому он продолжает работать до тех пор, пока не очистит pipeline.

Попробуйте поиграть с разными размерами очереди и вызовами **time.sleep()** в производителе или потребителе, чтобы имитировать более длительное время доступа к сети или диску соответственно. Даже небольшие изменения в этих элементах программы приведут к значительным изменениям в ваших результатах.

Это гораздо лучшее решение проблемы **производитель-потребитель** , но вы можете упростить его еще больше. Pipeline действительно не нужен для этой проблемы. Как только вы убираете запись, она просто становится очередью.

Вот как выглядит окончательный код с использованием **queue.Queue** напрямую:

```
import concurrent.futures
import logging
import queue
import random
import threading
import time
def producer(queue, event):
    """Pretend we're getting a number from the network."""
    while not event.is_set():
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        queue.put(message)
    logging.info("Producer received event. Exiting")
def consumer(queue, event):
```

```
    """ Pretend we're saving a number in the database. """
    while not event.is_set() or not pipeline.empty():
        message = queue.get()
        logging.info(
            "Consumer storing message: %s (size=%d)", message, queue
        )
    logging.info("Consumer received event. Exiting")
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    pipeline = queue.Queue(maxsize=10)
    event = threading.Event()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as exe
        executor.submit(producer, pipeline, event)
        executor.submit(consumer, pipeline, event)
        time.sleep(0.1)
        logging.info("Main: about to set event")
        event.set()
```

Это проще для чтения и показывает, как использование встроенных объектов Python может упростить сложную проблему.

Lock и **Queue** – удобные классы для решения проблем параллелизма, но есть и другие, предоставляемые стандартной библиотекой. Прежде чем закончить эту статью, давайте кратко рассмотрим некоторые из них.

Объекты Threading

Есть еще несколько объектов, предлагаемых модулем потоков Python. Хотя в приведенных выше примерах они вам не нужны, они могут пригодиться в разных случаях, поэтому полезно с ними ознакомиться.

Semaphore

Первым объектом потока Python, который нужно рассмотреть, является **threading.Semaphore** . **Semaphore** – это счетчик с несколькими особыми свойствами. Во-первых, счет считается атомарным. Это означает, что есть гарантия, что операционная система не будет менять поток в середине увеличения или уменьшения счетчика.

Внутренний счетчик увеличивается при вызове **.release()** и уменьшается при вызове **.acquire()** .

Следующее специальное свойство заключается в том, что если поток вызывает **.acquire()** , когда счетчик равен нулю, этот поток будет блокироваться, пока другой поток не вызовет **.release()** и увеличит счетчик до единицы.

Semaphore часто используются для защиты ресурса с ограниченными возможностями. Например, если у вас есть пул соединений и вы хотите ограничить размер этого пула определенным числом.

Timer

Threading.Timer – это способ запланировать функцию, которая будет вызвана по истечении определенного времени. Вы создаете таймер, передавая количество секунд ожидания и функцию для вызова:

```
t = threading.Timer(30.0, my_function)
```

Таймер запускается методом **.start()** . Функция будет вызвана в новом потоке в определенный момент после указанного времени, но имейте в виду, что нет никаких обещаний, что она будет вызвана именно в то время, когда вы захотите.

Если вы хотите остановить уже запущенный таймер, вы можете отменить его, вызвав **.cancel()** . Вызов **.cancel()** после срабатывания таймера ничего не делает и не вызывает исключения.

Таймер может использоваться для запроса действий пользователя через определенное время. Если пользователь выполняет действие до истечения таймера, может быть вызван **.cancel()** .

Barrier

Threading.Barrier можно использовать для синхронизации фиксированного количества потоков. При создании **Barrier** вызывающая сторона должна указать, сколько потоков будет синхронизироваться на нем. Каждый поток вызывает **.wait()** в **Barrier** . Все они останутся заблокированными до тех пор, пока не будет найдено указанное количество потоков, а затем все они будут освобождены одновременно.

Помните, что потоки планируются операционной системой, поэтому, даже если все потоки освобождаются одновременно, они будут запланированы для запуска по

одному за раз.

Одно из применений **Barrier** – разрешить инициализации пула потоков. Наличие потоков, ожидающих **Barrier** после их инициализации, гарантирует, что ни один из потоков не запустится до того, как все потоки завершат свою инициализацию.

Заключение: Потоки в Python

Теперь вы познакомились со многими возможностями Python, а также с примерами создания многопоточных программ и проблемами, которые они решают. Вы также видели несколько примеров проблем, возникающих при написании и отладке многопоточных программ.

Если вы хотите изучить другие варианты параллелизма в Python, ознакомьтесь с разделом «Ускорение работы программы на Python с помощью параллелизма» (Speed Up Your Python Program With Concurrency).

Если вы заинтересованы в глубоком погружении в модуль asyncio, прочитайте Async IO в Python: полное руководство (Async IO in Python: A Complete Walkthrough).

Что бы вы ни выбрали, теперь у вас есть достаточно информации, необходимой для написания программ с использованием потоков Python!