Данная статья была переведена в рамках "Конкурса на лучший перевод статьи" на сервисе Quizful. Ссылка на оригинал находится внизу страницы.]

Источник: http://www.quizful.net/post/thread-synchronization-in-python

Синхронизация доступа к разделяемым ресурсам в Python

Одним из важных моментов при использовании потоков является избежание конфликтов доступа, когда более одному потоку требуется доступ к переменной или какому-либо ресурсу. Если не проявить осторожность, одновременный доступ и модификация из нескольких потоков может вызвать всевозможные проблемы, и, что ещё хуже, эти проблемы имеют свойство проявляться лишь под большой нагрузкой, или на продакшн-серверах, или на более быстром железе, которое используют ваши клиенты. (Сейчас более-менее устоялось название ["heisenbugs"](http://en.wikipedia.org/wiki/Heisenbug) для трудновоспроизводимых багов - прим. перев.)

Для примера представьте программу на Python, обрабатывающую какие-нибудь данные и отслеживающую, сколько элементов было обработано:

counter = 0

def process\_item(item):

global counter

... do something with item ...

counter += 1

Если вызывать эту функцию из более чем одного потока, можно обнаружить, что счетчик **counter** не обязательно точен. Код будет работать верно в большинстве случаев, но иногда пропускать один или несколько элементов. Причина в том, что операция сложения на самом деле выполняется в три шага: интерпретатор получает текущее значение счетчика, вычисляет новое значение, и наконец, записывает его в переменную.

Если другой поток получает управление в тот момент, когда текущий поток получил значение переменной, он может выполнить вышеописанные 3 шага до того, как их выполнит текущий поток. А так как оба потока получили одно и то же начальное значение переменной, то переменная будет увеличена лишь на 1.

Другая распространённая проблема заключается в возникновении объектов в неполном или неконсистентном состоянии. Объект в таком состоянии может возникнуть, когда поток, производящий инициализацию или обновление сложной структуры данных прерывается другим потоком, который пытается эту структуру использовать во время обновления или инициализации.

Атомарные операции

Простейший способ синхронизировать доступ к разделяемым данным или ресурсам - положиться на атомарные операции в интерпретаторе. Атомарная операция - операция, выполняемая за 1 шаг, без возможности прерывания её другим потоком.

В целом, этот подход работает только если разделяемый ресурс состоит из единственного экземпляра встроенного типа, например, строки, числа, списка или словаря. Вот некоторые потокобезопасные операции:

* чтение или изменение одного атрибута объекта
* чтение или изменение одной глобальной переменной
* выборка элемента из списка
* модификация списка "на месте" (т.е. с помощью метода append)
* выборка элемента из словаря
* модификация словаря "на месте" (т.е. добавление элемента, или вызов метода **clear**)

Заметьте, что, как было сказано ранее, операции, которые читают переменную или атрибут, меняют их и затем записывают их обратно - не потокобезопасные. Другой поток может изменить переменную после того как её прочитал текущий поток, но перед тем как текущий поток изменил её.

Заметьте также, что какой-то код может выполняться при уничтожении объектов, и потому даже операции, свиду кажущиеся безопасными, могут заставить выполняться другие потоки и потому вызывать конфликты. Если сомневаетесь, используйте явные блокировки.

Блокировки в Python

Блокировки - наиболее фундаментальный механизм синхронизации предоставляемый модулем **threading** В каждый момент времени блокировка может принадлежать не более чем одному потоку. Если поток пытается получить блокировку, уже принадлежащую другому потоку, выполнение первого потока приостанавливается до тех пор, пока блокировка не будет освобождена.

Обычно блокировки используются для доступа к разделяемым ресурсам. Для каждого разделяемого ресурса создавайте объект типа Lock. Когда требуется доступ к ресурсу, следует вызывать метод **acquire** чтобы захватить блокировку (вызов в случае необходимости подождёт освобождения блокировки), и вызывать метод **release** чтобы освободить блокировку.

lock = Lock()

lock.acquire() # заблокирует выполнение, если блокировка кем-то захвачена

... доступ к разделяемому ресурсу

lock.release()

Для корректного выолнения важно освобождать блокировки даже в случае возникновения ошибок. Для этого можно использовать **try-finally**

lock.acquire()

try:

... доступ к разделяемому ресурсу

finally:

lock.release() # освободить блокировку, что бы ни произошло

В Python 2.5 и более поздних версиях можно использовать оператор **with**. Если оператор **with** работает с блокировкой, то он автоматически захватывает блокировку перед входом в блок, и освобождает её после выхода из блока:

from \_\_future\_\_ import with\_statement # только в версии 2.5

with lock:

... доступ к разделяемому ресурсу

Метод **acquire** принимает необязательный флажок ожидания, который можно использовать для избежания блокирования, если блокировка уже кем-то захвачена. Если установить флажок в **False**, метод не блокируется, но вернёт **False**, если блокировка кем-то захвачена:

if not lock.acquire(False):

... не удалось заблокировать ресурс

else:

try:

... доступ к разделяемому ресурсу

finally:

lock.release()

Можно использовать этот метод для выяснения, будет ли вызов **acquire** блокировать выполнение потока или нет, однако, другой поток может захватить блокировку между вызовом метода и следующим оператором.

if not lock.locked():

# другой поток может начать выполняться перед тем как мы перейдём к следующему оператору

lock.acquire() # всё равно может заблокировать выполнение

Проблемы с простыми блокировками в Python

Обычной блокировке (объекту типа **Lock**) всё равно, кто её захватил; если блокировка захвачена, любой поток при попытке её захватить будет заблокирован, даже если этот поток уже владеет этой блокировкой в данный момент. Рассмотрим пример:

lock = threading.Lock()

def get\_first\_part():

lock.acquire()

try:

... получить данные первой части разделяемого объекта

finally:

lock.release()

return data

def get\_second\_part():

lock.acquire()

try:

... получить данные второй части разделяемого объекта

finally:

lock.release()

return data

В данном примере мы имеем разделяемый объект, к разным частям которого обращаются две функции. Обе используют блокировки чтобы предотвратить изменение объекта другими потоками пока осуществляется доступ.

Теперь, если мы хотим добавить третью функцию, котора я получает обе части объекта, могут возникнуть проблемы. Наивный подход заключается в последовательном вызове функций и возврате общего результата:

def get\_both\_parts():

first = get\_first\_part()

second = get\_second\_part()

return first, second

Проблема заключается в том, что если какой-то другой поток изменит объект между двумя вызовами функций, наши данные утратят целостность.

Очевидное решение - захватить блокировку во внешней функции:

def get\_both\_parts():

lock.acquire()

try:

first = get\_first\_part()

second = get\_second\_part()

finally:

lock.release()

return first, second

Однако, такой код не будет работать; функции доступа к частям объекта заблокируются, т.к. внешняя функция уже захватила ту же блокировку, которую используют внутренние. Чтобы обойти эту проблему, можно добавить флажки для функций доступа к частям объекта, чтобы позволить внешней функции отключить блокировку в функциях доступа, но этот путь почти гарантированно ведёт к ошибкам.

К счастью, модуль **threading** предоставляет более удобную реализацию блокировок: реентерабельные (c повторным входом) блокировки ("реентерабельный" - код с возможностью одновременного выполнения разными потоками)

Блокировки с повторным входом (r-блокировки)

Класс **RLock** - вариант простой блокировки, которая блокирует поток только в том случае, если блокировка захвачены *другим*потоком, в то время как обычные блокировки могут заблокировать тот же поток, если тот попытается захватить её повторно.

Если поток пытается захватить блокировку, которой он уже владеет, выполнение продолжается как обычно:

lock = threading.Lock()

lock.acquire()

lock.acquire() # вызов заблокирует выполнение

lock = threading.RLock()

lock.acquire()

lock.acquire() # вызов не заблокирует выполнение

r-блокировки полезны для вложенного доступа к разделяемым ресурсам, например так, как было показано в примере для предыдущего раздела. Чтобы исправить поведение методов доступа к частям объекта, следует просто заменить объект типа **Lock** на объект типа **RLock**, всё остальное остаётся без изменений:

lock = threading.RLock()

def get\_first\_part():

... см. выше

def get\_second\_part():

... см. выше

def get\_both\_parts():

... см. выше

Таким образом можно делать выборку отдельных частей или всех частей объекта сразу без угрозы получить нецелостные данные или заблокироваться.

Заметьте, что эта блокировка (типа **RLock**) отслеживает глубину рекурсии, поэтому так же как в случае обычных блокировок следует вызывать функцию **release** на каждый вызов функции **acquire**

Семафоры

Семафоры - более сложный и совершенный механизм блокировок. Внутри семафора - счетчик, в отличии от объекта блокировки, в которой просто флажок. Семафор блокирует поток только когда более заданного числа потоков пытаются захватить семафор.

В зависимости от того, как семафор будет инициализирован, он может позволить нескольким потокам выполнять один и тот же код одновременно.

semaphore = threading.BoundedSemaphore()

semaphore.acquire() # уменьшает счетчик

... доступ к общему ресурсу

semaphore.release() # увеличивает счетчик

Счетчик уменьшается когда семафор захватывается и увеличивается когда семафор освобождается. Если значение счетчик становится равно 0, захват семафора приведёт к блокировке потока. Если счетчик вновь будет увеличен, то один из заблокированных на семафоре потоков сможет захватить семафор и продолжит работу.

Обычно семафоры используются чтобы лимитировать доступ к ограниченному ресурсу типа сетевых подключений или сервера баз данных. Инициализируем семафор максимальным числом потоков, которые должны иметь доступ к ресурсу - и реализация семафора позаботится обо всём остальном.

max\_connections = 10

semaphore = threading.BoundedSemaphore(max\_connections)

Если не передавать в **BoundedSemaphore** параметр, семафор будет инициализирован 1 (и таким образом станет обычной блокировкой)

Модуль **threading** в **Python** предоставляет две реализации семафоров; класс **Semaphore** предоставляет неограниченный семафор, позволяющий вызвать **release** сколько угодно раз для увеличения счетчика. Чтобы избежать простых ошибок обычно лучше использовать класс **BoundedSemaphore**, который считает ошибкой вызовы release, для которых до этого не был вызван метод acquire.

Синхронизация между потоками

Блокировки можно также использовать для синхронизации потоков. Модуль **threading** содержит ещё несколько классов, созданных для этой цели.

События (Events)

Событие - простейший объект синхронизации, являющийся, по сути, флажком. Поток может ожидать установки этого флажка, или устанавливать и сбрасывать его самостоятельно.

event = threading.Event()

# поток клиента может подождать пока флажок будет установлен

event.wait()

# серверный поток может установить или сбросить флажок

event.set()

event.clear()

Если флажок установлен, метод **wait** ничего не сделает. Если флажок сброшен, **wait** заблокирует выполнение потока до тех пор, пока флажок вновь не будет установлен (другим потоком). Любое количество потоков могут ожидать одно и то же событие одновременно.

Условные переменные (Conditions или conditional variable)

Условная переменная - более совершенный вариант события (event). Условная переменная представляет собой нечто типа изменения состояния в приложении, которого может ожидать какой-нибудь поток, или оповестить, что это состояние наступило. Рассмотрим простой пример consumer/producer (производитель/потребитель).

Для начала нам нужен объект условной переменной:

# представляет добавление элемента к ресурсу

condition = threading.Condition()

Производящему потоку (producer) необходимо захватить объект **condition**, после этого он может уведомить потоки-потребители (consumer threads) что доступен новый элемент.

# производящий поток

... сгенерировать элемент

condition.acquire()

... добавить данные к ресурсу

condition.notify() # уведомить, что новые данные доступны

condition.release()

Потоки-потребители должны захватить условную переменную (и связанную с ней блокировку), и затем уже делать выборку данных из общего ресурса.

# Поток-потребитель

condition.acquire()

while True:

... Получить элемент (item) данных из общего ресурса

if item:

break

condition.wait() # ждать, пока данные не станут доступными

condition.release()

... обработать элемент item

Метод **wait** освобождает блокировку, блокирует текущий поток, пока другой поток не позовёт **notify** или **notifyAll** на этой же условной переменной, затем снова захватывает блокировку условной переменной. Если изменение состояния условной переменной ждут несколько потоков, метод **notify** разблокирует ("разбудит") только один из ждущий потоков, тогда как метод **notifyAll** всегда "будит" все ждущие потоки.

Чтобы избежать блокирования потока при вызове метода **wait**, можно передать время ожидания, вещественное число в секундах. Если оно задано, то метод возвращается не позднее чем через заданное время, даже в случае, если метод **notify** не был вызван. При использовании таймаута необходимо убедиться, что ожидаемое изменение ресурса действительно наступило.

Следует отметить, что объект условной переменной связан с блокировкой и потому блокировка должна быть захвачена прежде чем осуществляется доступ к условной переменной. Аналогично, эта блокировка должна быть освобождена, когда поток больше не использует условную переменную. В реальных приложениях следует использовать блоки **try-finally** или **with**, как было показано ранее, чтобы избежать ошибок.

Чтобы связать условную переменную с существующей блокировкой, передайте объект блокировки в конструктор условной переменной. Это полезно и в том случае, если требуется использовать несколько условных переменных для синхронизации доступа к одному ресурсу.

lock = threading.RLock()

condition\_1 = threading.Condition(lock)

condition\_2 = threading.Condition(lock)

Надеюсь, статья была вам интересна и что она пролила свет на механизмы синхронизации потоков в Python.