Лабораторная работа №5

Тема: «Работа с потоками в Python»

Понятие потока

Поток (***thread***) – это абстракция уровня операционной системы, содержащая отдельный контекст (стек) выполнения, позволяющая выполнять несколько задач одновременно или переключаться между ними.

Множество потоков могут совместно использовать данные и ресурсы, получая преимущества так называемого пространства разделяемых сведений. Сами особенности потоков и процессов зависят от той ОС, в которой вы планируете запускать своё приложение, однако, в целом, можно постулировать, что некий поток содержится внутри какого-то процесса и что различные потоки при одних и тех же условиях процесса совместно разделяют некоторые ресурсы. В противоположность этому, различные процессы не разделяют свои собственные ресурсы с прочими процессами.

Некий поток состоит из трёх элементов: программных счётчиков, регистров и стека. Совместные с прочими потоками ресурсы в том же самом процессе, по существу, содержат данные и ресурсы ОС. Более того, потоки обладают своим собственным состоянием исполнения, а именно, состоянием потока, а также могут выполнять синхронизацию с другими потоками.

Состояниями потока могут быть ***ready***, ***running*** и ***blocked*** (рис. 1):

- при создании некого потока он входит в состояние ***ready***;
- поток планируется к выполнению своей ОС и, когда происходит его активация, он начинает своё выполнение, переходя в состояние ***running***. Это значит, что происходит выполнение инструкций;
- поток может ожидать исполнения некого условия, переходя из состояния ***running*** в состояние ***blocked***. В заблокированном состоянии процесс выполняет операции, которые не дают ему быть готовым к исполнению до тех пор, пока не произойдет какое-либо событие. Один из примеров – когда процесс инициализирует операцию IO, он становится заблокированным, и таким образом другой процесс может использовать процессор. Когда такое блокирующее условие прекращается, блокированный поток возвращается в состояние ***ready***.

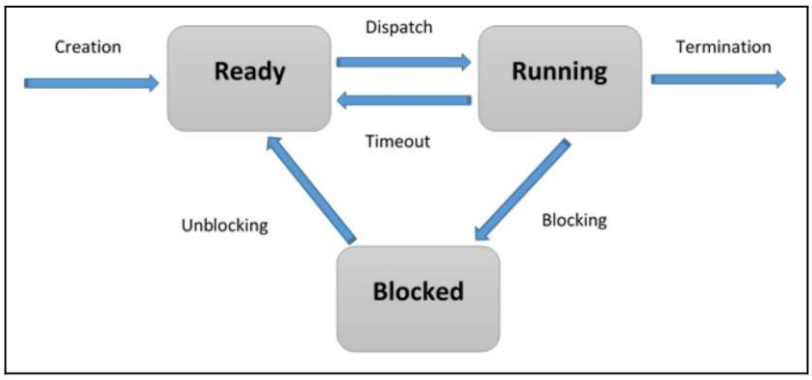

Рис. 1 – Жизненный цикл потока

Основное преимущество многопоточного программирования состоит в производительности, поскольку контекстное переключение между процессами включается намного тяжелее, чем контекстное переключение между потоками, которые относятся к одному и тому же процессу.

Модуль threading

Модуль **threading** впервые был представлен в Python 1.5.2 как продолжение низкоуровневого модуля потоков. Модуль **threading** значительно упрощает работу с потоками и позволяет программировать запуск нескольких операций одновременно. Обратите внимание на то, что потоки в Python лучше всего работают с операциями Input/Output (IO), такими как загрузка ресурсов из интернета или чтение файлов и папок на вашем компьютере. Это связано с тем, что IO операции (чтение/запись файла, чтение из сокета), запущенные интерпретатором Python, выполняются операционной системой. В это время интерпретатор Python может продолжать исполнять код в другом потоке, пока IO операция, выполняемая операционной системой, не завершится.

Если вам нужно работать с трудоёмкими операциями на CPU (например: циклы, обрабатывающие большое количество объектов, парсинг сложных форматов, сложные математические вычисления), тогда вам нужно обратить внимание на модуль **multiprocessing** вместо **threading**. Причина заключается в том, что Python содержит **Global Interpreter Lock (GIL)**.

Global Interpreter Lock (GIL)

GIL был представлен, чтобы сделать работу с памятью ***CPython*** проще и обеспечить наилучшую интеграцию с ***C*** (например, с расширениями). ***GIL*** – это механизм блокировки, когда интерпретатор ***Python*** запускает в работу только один поток за раз.

Т. е. только один поток может исполняться в байт-коде Python единовременно. GIL следит за тем, чтобы несколько потоков не выполнялись параллельно.

Краткие сведения о GIL:

- одновременно может выполняться один поток;
- интерпретатор Python переключается между потоками для достижения конкурентности;
- GIL делает однопоточные программы быстрыми;
- операциям ввода/вывода GIL обычно не мешает;
- GIL позволяет легко интегрировать непотокобезопасные библиотеки на C, благодаря GIL у нас есть много высокопроизводительных расширений/модулей, написанных на C;
- для CPU-зависимых задач интерпретатор делает проверку каждые N тиков и переключает потоки. Таким образом один поток не блокирует другие.

Тест производительности

Рассмотрим тривиальную CPU-зависимую функцию (т. е. функцию, скорость выполнения которой зависит преимущественно от производительности процессора):

```
def count(n):
    while n > 0:
        n -= 1
```

Сначала запустим ее дважды по очереди:

```
count(100000000)
count(100000000)
```

Теперь запустим ее параллельно в двух потоках:

```
from threading import Thread

t1 = Thread(target=count, args=(100000000,))
t1.start()
t2 = Thread(target=count, args=(100000000,))
t2.start()
t1.join()
t2.join()
```

Последовательный запуск – 10 сек.

Параллельный запуск – 12 сек.

В любой момент может выполняться только один поток Python. Глобальная блокировка интерпретатора – GIL – тщательно контролирует выполнение потоков. GIL гарантирует каждому потоку эксклюзивный доступ к переменным интерпретатора (и соответствующие вызовы C-расширений работают правильно).

Принцип работы прост. Потоки удерживают GIL, пока выполняются. Однако они освобождают его при блокировании для операций ввода-вывода. Каждый раз, когда поток вынужден ждать, другие, готовые к выполнению, потоки используют свой шанс запуститься (рис. 2).

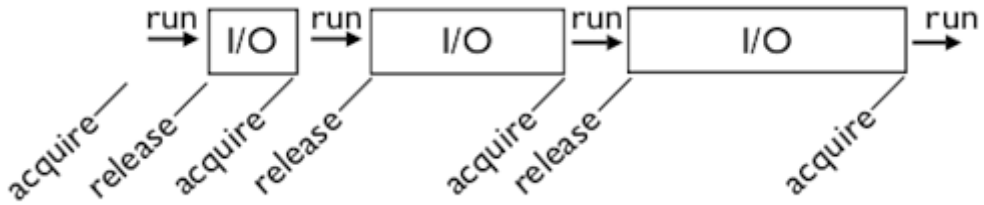

Рис. 2 – Схема удержания потоков

При работе с CPU-зависимыми потоками, которые никогда не производят операции ввода-вывода, интерпретатор периодически проводит проверку («the periodic check») (рис. 3).

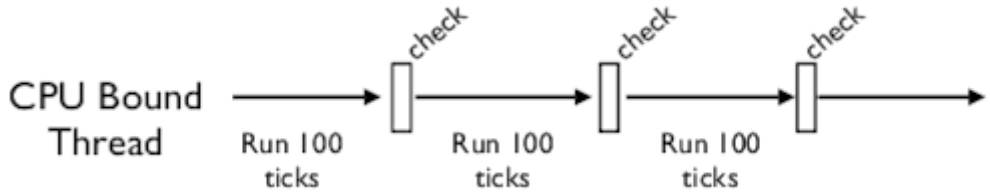

Рис. 3 – Периодическая проверка CPU-зависимых потоков

По умолчанию это происходит каждые 100 «тиков» (рис. 4), но этот параметр можно изменить с помощью `sys.setcheckinterval()`. Интервал проверки – глобальный счетчик, абсолютно независимый от порядка переключения потоков.

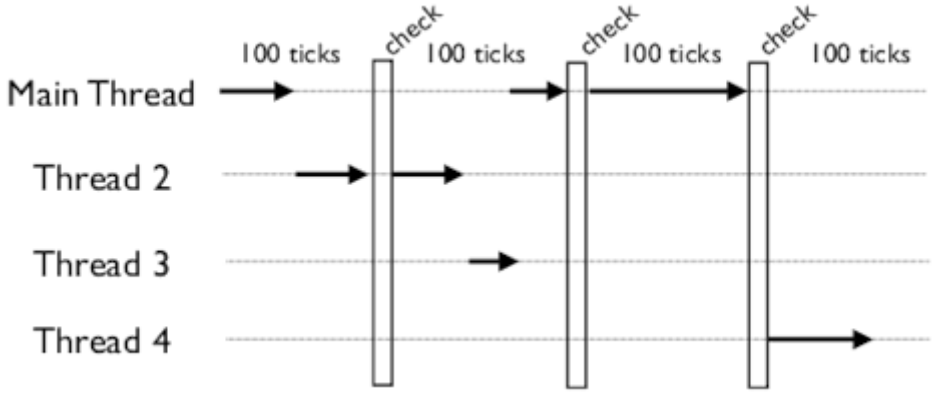

Рис. 4 – Схема переключения потоков

При периодической проверке в главном потоке запускаются обработчики сигналов, если таковые имеются. Затем GIL отключается и включается вновь. На этом этапе обеспечивается возможность переключения нескольких CPU-зависимых потоков (при кратком освобождении GIL другие потоки имеют шанс на запуск).

Тики примерно соответствуют выполнению инструкций интерпретатора. Они *не основываются* на времени. Фактически, длинная операция может заблокировать всё.

Планировщик потоков

У Python нет средств для определения, какой поток должен запуститься следующим. Нет приоритетов, вытесняющей многозадачности, round-robin и т. п. Эта функция целиком возлагается на операционную систему. Это одна из причин странной работы сигналов: интерпретатор никак не может контролировать запуск потоков, он просто переключает их как можно чаще, надеясь, что запустится главный поток.

Ctrl-C часто не срабатывает в многопоточных программах, потому что главный поток обычно заблокирован непрерываемым **thread-join** или **lock**. Пока он заблокирован, он не сможет запуститься. Как следствие, он не сможет выполнить обработчик сигнала.

В качестве дополнительного бонуса интерпретатор остается в состоянии, где он пытается переключить поток после каждого тика. Мало того, что вы не можете прервать программу, она еще и работает медленнее.

Процессы (Processes)

Чтобы достичь параллелизма в Python был добавлен модуль **multiprocessing**, который предоставляет API, и выглядит очень похожим, если вы использовали **threading** раньше.

По функциональности модуль **multiprocessing** напоминает **threading**. Например, процессы можно создавать точно так же из обычных функций. Методы работы с

процессами почти все те же самые, что и для потоков из модуля **threading**. А вот для синхронизации процессов и обмена данными принято использовать другие инструменты. Речь идет об очередях (***Queue***) и каналах (***Pipe***). Впрочем, аналоги локов, событий и семафоров, которые были в **threading**, здесь тоже есть. Создание процесса с точки зрения потребления ресурсов очень дорогостоящая операция. Поэтому для операций ввода/вывода в основном выбираются потоки.

Кроме того, в модуле **multiprocessing** есть механизм работы с общей памятью. Для этого в модуле есть классы переменной (***Value***) и массива (***Array***), которые можно «обобщать» (***share***) между процессами. Для удобства работы с общими переменными можно использовать классы-менеджеры (***Manager***). Они более гибкие и удобные в обращении, однако более медленные. Нельзя не отметить приятную возможность делать общими типы из модуля **ctypes** с помощью модуля **multiprocessing.sharedctypes**.

Изменим предыдущий пример. Теперь модифицированная версия использует **Процесс** вместо **Потока**.

```
import multiprocessing


t1 = multiprocessing.Process(target=count, args=(100000000,))
t1.start()
t2 = multiprocessing.Process(target=count, args=(100000000,))
t2.start()
```

Что же изменилось? Мы просто импортировали модуль **multiprocessing** вместо **threading**. А затем вместо потока использовали процесс. Теперь вместо множества потоков мы используем процессы, которые запускаются на разных ядрах CPU (если, конечно, у вашего процессора несколько ядер).

С помощью класса **Pool** мы также можем распределить выполнение одной функции между несколькими процессами для разных входных значений. Пример из официальных документов:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

Здесь вместо того, чтобы перебирать список значений и вызывать функцию `f` по одному, мы фактически запускаем функцию в разных процессах. Один процесс

выполняет `f(1)`, другой `f(2)`, а третий `f(3)`. Наконец, результаты снова объединяются в список. Это позволяет нам разбить тяжелые вычисления на более мелкие части и запускать их параллельно для более быстрого расчета.

Блокировки

В многопоточной программе доступ к объектам иногда нужно синхронизировать. Часто для синхронизации потоков используют блокировки. Любые блокировки замедляют выполнение программы. Лучше избегать использование блокировок и отдавать предпочтение обмену данными через очереди.

```
# Очереди, модуль queue
from queue import Queue
from threading import Thread

def worker(q, n):
    while True:
        item = q.get()
        if item is None:
            break
        print("process data:", n, item)

q = Queue(5)
th1 = Thread(target=worker, args=(q, 1))
th2 = Thread(target=worker, args=(q, 2))
th1.start(); th2.start()

for i in range(50):
    q.put(i)

q.put(None); q.put(None)
th1.join(); th2.join()
```

```
> $ python example.py
> process data: 1 0
> process data: 1 1
> process data: 2 3
...
```

Создаем очередь с максимальным размером 5. Используем методы `put()` для того, чтобы поместить данные в очередь, и `get()` для того, чтобы забрать данные из очереди. Использование очередей делает код выполняемой программы более простым. И по возможности лучше разрабатывать код таким образом, чтобы не было глобального разделяемого ресурса, или состояния.

```
# Синхронизация потоков, блокировки

import threading

class Point(object):
    def __init__(self, x, y):
        self.mutex = threading.RLock()
        self.set(x, y)
```

```
    def get(self):
        with self.mutex:
            return (self.x, self.y)

    def set(self, x, y):
        with self.mutex:
            self.x = x
            self.y = y

# use in threads
my_point = Point(10, 20)
my_point.set(15, 10)
my_point.get()
```

Этот код гарантирует что если объект класса **Point** будет использоваться в разных потоках, то изменение **x** и **y** будет всегда атомарным. Работает все это так: – при вызове метода берем блокировку через `with self._mutex`. Весь код внутри `with` блока будет выполняться только в одном потоке. Другими словами, если два разных потока вызовут `.get` то пока первый поток не выйдет из блока, второй будет его ждать – и только потом продолжит выполнение.

Зачем это все нужно? Координаты нужно менять одновременно: ведь точка – это атомарный объект. Если позволить одному потоку поменять **x**, а другой в это же время поправит **y**, логика алгоритма может сломаться.

Модуль concurrent.futures

Пример использования класса **ThreadPoolExecutor**

```
import concurrent.futures
import urllib.request


def get_status(link):
    response = urllib.request.urlopen(link)
    return response.status


urls = [
    "https://miet.ru",
    "https://ya.ru",
    "https://google.com",
    "https://vk.com",
]

with concurrent.futures.ThreadPoolExecutor() as executor:
    statuses = {executor.submit(get_status, url): url for url in urls}

    for future in concurrent.futures.as_completed(statuses):
        url = statuses[future]

        print(url, future.result())
```

1. `concurrent.futures` импортируется, чтобы получить доступ к классу `ThreadPoolExecutor` и методу `as_completed`.
2. Конструкция `with … as …` используется для создания объекта класса `ThreadPoolExecutor` и очистки потоков после выполнения.
3. В `executor.submit` отправляются задачи (по одному для каждого `url` из списка `urls`).
4. Каждый вызов `submit` возвращает объект класса `Future`.
5. Метод `as_completed` ожидает каждого вызова `get_status` для выполнения.

Пример использования класса **ProcessPoolExecutor**

```
    import concurrent.futures
    import math

    PRIMES = [
        112272535095293,
        112582705942171,
        112272535095293,
        115280095190773,
        115797848077099,
        1099726899285419
    ]

    def is_prime(n):
        if n < 2:
            return False
        if n == 2:
            return True
        if n % 2 == 0:
            return False

        sqrt_n = int(math.floor(math.sqrt(n)))
        for i in range(3, sqrt_n + 1, 2):
            if n % i == 0:
                return False
        return True

    def main():
        with concurrent.futures.ProcessPoolExecutor() as executor:
            for number, prime in zip(PRIMES, executor.map(is_prime,
PRIMES)):
                print('%d is prime: %s' % (number, prime))

    if __name__ == '__main__':
        main()
```

Требования к выполнению лабораторной работы №6

1. Изучите теоретическую часть к шестой лабораторной работе:
    a. Теоретическая часть к шестой лабораторной работе.
    b. Лекция №6.
2. Создайте новый проект.
3. Запустите примеры из лабораторной работы.
4. Выполните задание согласно вашему варианту:
    a. Вычислите свой вариант (*согласно формуле ниже*). Если сделали не свой вариант => работа не засчитывается.
    b. Каждое задание представляет собой отдельный скрипт формата: `lab_{`**`номер_ЛР`**`}_{`**`номер_задания`**`}_{`**`номер_варианта`**`}.py`, пример: `lab_6_1_2.py`
    c. Отправьте выполненное задание в ОРИОКС (*раздел Домашние задания*).

Формат защиты лабораторных работ:

1. Продемонстрируйте выполненные задания.
2. Ответьте на вопросы по вашему коду.
3. При необходимости выполните дополнительное (*дополнительные*) задание от преподавателя.
4. Ответьте (*устно*) преподавателю на контрольные вопросы.

Список вопросов

1. Что такое поток? В чём состоит ключевое отличие потока от процесса?
2. Что такое процесс? В чём состоят ключевые отличия между процессами и потоками?
3. Что такое многопроцессорность? В чём состоят ключевые отличия между многопроцессорностью и многопоточностью?
4. Что предлагают параметры, предлагаемые модулем threading из Python?
5. Какие варианты API предоставляются модулем multiprocessing?
6. Что такое процесс демона?
7. Что представляет собой GIL?
8. Принцип работы GIL.
9. Для каких задач стоит использовать модуль threading?
10. Для каких задач стоит использовать модуль multiprocessing?
11. Когда не стоит использовать ни threading, ни multiprocessing?
12. Что стоит за основной идеей синхронизации потоков при помощи блокировки?
13. Что составляет процесс реализации синхронизации потоков с применением блокировки в Python?
14. В чём основная идея структуры данных очереди?
15. Что является основным приложением очередей для совместного программирования?
16. Что составляет центральное отличие между обычными очередями и очередями с приоритетами?

Задания

Задание №1

Самостоятельно изучите документацию по модулям ***threading***, ***multiprocessing,*** ***concurrent.futures***:

https://docs.python.org/3/library/threading.html

https://docs.python.org/3/library/multiprocessing.html

https://docs.python.org/3/library/concurrent.futures.html

Задание №2

Самостоятельно изучите все основные способы синхронизации потоков.

Продемонстрируйте один из способов синхронизации потоков.

№ Варианта = *номер_студенческого* % **5** + 1

Вариант	**Способ синхронизации**
1	С помощью семафора (Semaphore)
2	Условием
3	Событием (Event Objects)
4	Барьером (Barrier Objects)
5	Очередью (Queue Objects)

Задание №3

Даны два набора по N чисел: $P = \{p_1, p_2, \dots, p_N\}$ и $Q = \{q_1, q_2, \dots, q_N\}$. Необходимо построить матрицу R размерностью $N \times N$, каждый элемент $r_{i,j}$ которой вычисляется по формуле.

№ Варианта = *номер_студенческого* % **5** + 1

Вариант	Формула
1	$r_{i,j} = \frac{1}{1 + \lVert q_j - p_i \rVert}, i = 1..N, j = 1..N, N = 5000$
2	$r_{i,j} = \frac{1}{1 + (q_j - p_i)^2}, i = 1..N, j = 1..N, N = 5000$
3	$r_{i,j} = \sqrt{(q_j - p_i)^2}, i = 1..N, j = 1..N, N = 5000$
4	$r_{i,j} = \sqrt{q_j^2 + p_i^2}, i = 1..N, j = 1..N, N = 5000$
5	$r_{i,j} = \sqrt{\lVert q_j - p_i \rVert}, i = 1..N, j = 1..N, N = 5000$

Сравните результат и время исполнения программы с использованием модуля `concurrent.futures` и без.

Задание №4

Выберите задание согласно варианту. Выполните задание в одном и в нескольких потоках. Сравните результаты и время выполнения программы.

№ Варианта = *номер_студенческого* % **3** + 1

Вариант	**Задание**
1	Получите содержимое 10 сайтов и запишите результат в файлы.
2	Выполните поиск содержимого текущей директории (включая поддиректории) и найдите все файлы с расширением ***txt*** в которых есть ключевое слово ***key***.
3	Выполните поиск содержимого текущей директории (включая поддиректории) и создайте в каждой поддиректории (включая текущую папку) файл ***size.txt***, в который запишите количество файлов, содержащихся в поддиректории.