Процессы и потоки

Цель занятия

После освоения темы вы:

- узнаете подходы к параллельному исполнению кода и термины, используемые в многопоточном и параллельном программировании;
- сможете создавать процессы и потоки, выполнять синхронизацию потоков и управлять их выполнением на Python.

План занятия

Используемые термины

Основные термины приведены в п. 1 «Подходы к параллельному исполнению кода».

Конспект занятия

1. Подходы к параллельному исполнению кода

Пусть требуется выполнить сразу несколько задач. Рассмотрим подходы, как это можно сделать.

1. **Выполнить задачи по очереди.** Так чаще всего и происходит. И если такой подход не создает проблем, то нужно им пользоваться — это самое простое решение. Если задачи длительные, и их выполнение зависит друг от друга, такое решение не будет лучшим.

2. **Выполнить разные задачи на разных компьютерах.** Тоже простое решение. Вполне рабочий вариант при наличии кластера из нескольких вычислительных серверов. Минусы решения — высокая стоимость используемого оборудования и проблема взаимодействия задач, выполняемых на разных компьютерах. Взаимодействовать задачи могут с помощью сетевых соединений, что требует дополнительного оборудования. Еще один недостаток сетевых соединений — они довольно медленные.

3. **Выполнить задачи на многопроцессорном компьютере (или на многоядерном процессоре).** Взаимодействие между задачами будет происходить по внутренним каналам. То есть гораздо быстрее, чем при использовании сетевых соединений.

   Реально данный подход применяется редко, поскольку процессоров (или ядер в одном процессоре) не настолько много. Количество одновременно выполняемых процессов всегда значительно больше, чем ядер в процессоре. Поэтому они никак не могут выполняться по-настоящему одновременно.

4. **Разбить задачи на небольшие «кусочки» и быстро переключаться между ними.** При таком подходе операционная система дает поработать каждому процессу несколько миллисекунд. Есть два важных момента. Во-первых, механизм переключения реализуется самой операционной системой, то есть нам его не требуется реализовывать. Во-вторых, мы не контролируем процесс переключения.

   Разбиение задач на «кусочки» и переключением между ними не дает ускорения — задачи выполняются столько же времени или даже больше. В этой реализации нет настоящей параллельности выполнения задач, она кажущаяся.

Введем основные термины.

Многозадачность / конкурентность (concurrent) — способность обрабатывать несколько задач «одновременно»: параллельно или чередуя выполнение.

Параллелизм — способность выполнять несколько вычислений одновременно на многопроцессорной системе.

Единица выполнения — объект, выполняющий код конкурентно.

Python поддерживает три вида единиц выполнения:

1. **Процесс** — экземпляр программы во время ее выполнения, которому выделена память и другие ресурсы.
2. **Поток** (англ. thread — буквально «нить») — минимальная единица выполнения, поддерживаемая операционной системой. Несколько потоков существуют в рамках одного процесса и разделяют одни и те же ресурсы.
3. **Сопрограмма** — функция, способная приостановить выполнение и продолжить позже.

Процессы и потоки предоставляет операционная система, сопрограмма — сущность языка Python.

Синхронное выполнение кода — выполнение кода «так, как написано» строчка за строчкой.

Асинхронное выполнение кода — последовательность выполнения инструкций определяется внешней системой. Например, операционной системой или циклом событий при внутренней асинхронности.

Рассмотрим небольшое образное сравнение, чтобы лучше понимать разницу между терминами. Пусть процесс — это исполнитель, обладающий некими ресурсами. Например, повар, который печет торт.

Если мы запускаем несколько процессов одновременно, это означает, что у нас есть несколько кухонь, на каждой из которых работает свой повар. У каждого исполнителя свой набор ресурсов, которым они не могут делиться с остальными.

Когда мы выполняем несколько потоков в рамках одного процесса, это означает, что есть одни ресурсы (одна духовка и один набор продуктов) и несколько исполнителей (поваров). Исполнители могут выполнять разные задачи и даже иногда конкурировать за использование доступных ресурсов.

Асинхронное выполнение — ситуация, когда один повар выполняет несколько задач: готовит несколько блюд одновременно.

2. Создание потоков

Рассмотрим пример программы, которая загружает информацию с разных адресов в Интернет. Сетевые соединения весьма медленны. Поэтому, возможно, для такой программы мы можем получить некоторое преимущество в производительности, если будем выполнять запросы параллельно. В примере программы запросы выполняются последовательно к адресам example.com, example.net, example.org, example.edu:

```
import time
import httpx

base_url = 'http://example'
domains = ['.com', '.net', '.org', '.edu']

def get_page(domain):
    print(f'connecting {domain}...')
    r = httpx.get(base_url + domain)
    r.raise_for_status()
    print(f'response from {domain}: {r}')
    return r.text

start = time.perf_counter()
for d in domains:
    get_page(d)
end = time.perf_counter()
print(f'{len(domains)} pages loaded in {end-start:0.2f} sec.')
```

Выполнение программы выглядит примерно так:

```
connecting .com...
response from .com: <Response [200 OK]>
connecting .net...
response from .net: <Response [200 OK]>
connecting .org...
response from .org: <Response [200 OK]>
connecting .edu...
response from .edu: <Response [200 OK]>
4 pages loaded in 1.52 sec.
```

Рассмотрим, как запустить загрузку параллельно с помощью функций модуля `threading`. Сама функция остается такой же — это большой плюс многопоточного программирования, но не всегда так работает.

В цикле мы проходим по списку доменных имен и создаем потоки: `t = threading.Thread(target=get_page, args=(d,))`.

Из модуля `threading` мы создаем класс `Thread`, в который передаем параметры: `target` — функция, которая будет запущена в потоке, и `args` — аргумент(ы) в виде кортежа. Поток запускается с помощью метода `start`. Сам поток сохраняется в список `threads`.

В следующем цикле мы запускаем метод `join` для потоков из списка `threads`, который дожидается, когда потоки завершат свое выполнение:

```
import threading


# то же самое

start = time.perf_counter()
threads = []
for d in domains:
    t = threading.Thread(target=get_page, args=(d,))
    t.start()
    threads.append(t)
for t in threads:
    t.join()
end = time.perf_counter()
print(f'{len(domains)} pages loaded in {end-start:0.2f} sec.')
```

Важно использовать метод `join` в отдельном цикле. Иначе мы получаем последовательное выполнение запросов — запустили поток, дождались завершения его выполнения, запустили следующий поток и так далее.

Результат запуска программы — программа выполняется в другом порядке, время выполнения программы уменьшилось:

```
connecting .com...
connecting .net...
connecting .org...
connecting .edu...
response from .net: <Response [200 OK]>
response from .org: <Response [200 OK]>
response from .edu: <Response [200 OK]>
response from .com: <Response [200 OK]>
4 pages loaded in 0.42 sec.
```

Важно! Никогда нельзя математически точно вычислить выигрыш при использовании многопоточности. Во-первых, существует разница в скорости загрузки всех четырех страниц. Во-вторых, существуют накладные расходы на создание потоков и управление ими.

Важно обратить внимание на методы, используемые в многопоточном коде выше:

- Метод `start` запускает новый поток, метод `run` выполняет функцию в текущем потоке.
- Метод `join` необходим, чтобы дождаться завершения потоков.
- Значение, возвращаемое из функции, запущенной в потоке, исчезает. Чтобы отдать результат, обычно его сохраняют в каком-нибудь списке — глобальном или переданным в качестве одного из аргументов.

Рассмотрим, будет ли программа дожидаться выполнения завершения потока, если мы не используем метод `join`. Это зависит от того, будет ли объявлен поток демоном. Идея использования термина «демон» идет от греческого языка, в переводе с которого термин означает «ангел-хранитель». Это что-то, что присутствует в программе постоянно.

Технически использование потока-демона означает, что такой поток будет завершен «молча», когда программа закончит работу. То есть программа не будет дожидаться его завершения.

Если запустить поток как обычно, программа дождется завершения всех потоков.

Рассмотрим пример программы, использующей потоки-демоны. Программа содержит функцию, распечатывающую 10 чисел с небольшой задержкой, чтобы мы могли заменить разницу — завершилась программа или нет.

Запускать поток как демон или нет, определяет параметр командной строки `sys.argv[1] == '-d'`:

```
# daemons_test.py
import sys
import time
import threading

def f():
    for i in range(10):
        print(i)
        time.sleep(0.1)

daemon = None
if len(sys.argv) > 1 and sys.argv[1] == '-d':
    daemon = True
threading.Thread(target=f, daemon=daemon).start()
```

Посмотрим на два варианта запуска программы. В первом случае программа запускается как обычно. Программа дожидается выполнения потока — это видно по тому, что был напечатан весь список чисел:

```
$ python daemons_test.py
0
1
2
3
4
5
6
7
8
9
```

Если мы запускаем поток как демон, программа успевает распечатать только 0, после чего завершается:

```
$ python daemons_test.py -d
0
```

Важно! Программа распечатает 0 всегда. Дело в том, что при запуске нового потока, остальные потоки на некоторое время блокируются, пока не встретится строка `time.sleep`. Если бы этой строки в коде не было, поток успел бы распечатать все десять чисел. Это связано с глобальной блокировкой интерпретатора.

Рассмотрим функции модуля `threading`, которые позволяют получить информацию об исполняемых потоках.

В коде ниже приведена функция `active_count`, которая говорит, сколько активных потоков выполняется в данный момент:

```
>>> import threading
>>> threading.active_count()
1
```

Можно получить ссылку на текущий поток, в котором мы находимся и из которого происходит вызов функции:

```
>>> threading.current_thread()
<_MainThread(MainThread, started 10716)>
```

и получить имя потока:

```
>>> _.name
'MainThread'
```

Можно обратиться к операционной системе и узнать `id` текущего потока:

```
>>> threading.get_native_id()
10716
```

Можно получить ссылку на главный поток:

```
>>> threading.main_thread()
<_MainThread(MainThread, started 10716)>
```

3. Создание процессов

Когда нужно создавать процессы. Создание процессов имеет смысл для счетных задач, т. е. задач с большим объемом вычислений. В отличие от задач по загрузке из Интернета, где длительность загрузки определяется скоростью соединения. Потоки в этом случае не помогут.

Рассмотрим пример. Напишем функцию, которая определяет, является ли число простым:

```
import math
import time
def is_prime(n):
      if n == 2:
            return True
      if n % 2 == 0:
            return False
      for d in range(3, math.isqrt(n) + 1, 2):
            if n % d == 0:
                  return False
      return True
```

Задача достаточно сложная и требует большого объема работы процессора. Если замерить время работы такой программы, то мы получим, что каждое из чисел проверяется разное время. Какое-то очень быстро, какое-то достаточно долго, а общая сумма времени на все числа порядка 128 секунд.

Как запустить эту программу с использованием нескольких процессов? Для этого нам нужен модуль `multiprocessing`, который копирует интерфейс модуля `threading`. Но в отличие от запуска потоков в процессах мы должны запустить функцию `main()`, выполнив проверку:

```
import multiprocessing

a = []

def f():
    a.append(1)

def main():
    procs = []
    for i in range(10):
        p = multiprocessing.Process(target=f)
        p.start()
        procs.append(p)
```

```
    for p in procs:
        p.join()
    print(a)  # []

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

Запустив код, мы увидим, что порядок вывода чисел поменялся. Этот порядок соответствует времени, которое было необходимо на проверку. Общее время выполнения = 44,48 секунды.

Передача данных между процессами. Если нам нужно, чтобы процессы взаимодействовали друг с другом, это будет сложной задачей. Разные процессы — это разные программы, работающие отдельно друг от друга. У них нет разделяемой памяти, они имеют независимую память.

Чтобы процессы передавали друг другу данные, нам нужно обратиться к специальным механизмам. Например, используем очередь:

```
import multiprocessing as mp

q = mp.Queue()

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Нужно передать процесс в очередь в качестве документа.

Функции модуля `os` (функции операционной системы) для работы с процессами:

- `os.getpid()` — получение `id` текущего процесса;
- `os.getppid()` — получение `id` родительского процесса;
- `os.abort()` — принудительно завершить текущий процесс.

Базовые функции для запуска процессов:

- `os.exec*(...)` — запустить команду вместо текущего процесса;
- `os.popen(cmd)` — запустить команду в дочернем процессе;
- `os.fork()` — запустить еще один экземпляр текущего процесса;
- `os.kill(pid, sid)` — послать сигнал `sid` процессу `pid`.

Они лежат в основе модуля `multiprocessing`.

Fork-бомба — это процесс, который бесконечно плодит свои копии, а эти копии тоже плодят свои копии. Процесс бесконечный, и загрузит вашу операционную систему мгновенно. Не запускайте этот код у себя!

Пример, кода, реализующего fork-бомбу:

```
#НЕ ЗАПУСКАТЬ!
import os
while True:
      x = os.fork()
```

Для запуска внешних программ в дочерних процессах существует модуль `subprocess`. Это бывает полезно — например, для тестирования чужого кода. В модуле `subprocess` основная функция — `run()` — запуск команды.

Мы можем передать в программу что-то на вход (обязательно поток байтов).

Параметр `capture_output` сохранит в отдельный файл то, что программы выводит на экран. Если это не указать, то все будет выведено просто на экран (для случаев, когда нужно, чтобы это видел пользователь).

Мы также указываем `timeout`, чтобы ограничить время выполнения программы. В данном случае мы указали 1 секунду. Если программа не завершается через 1 секунду, то программа выдаст ошибку.

Параметр `check` проверяет, с каким кодом ошибки завершилась программа. Программа, которая завершается нормально, должна выдать код 0. В Python это делается автоматически. Если завершится с ошибкой, то программа выведет ошибку.

4. Синхронизация потоков

Главная проблема многопоточного программирования состоит в том, что мы не можем контролировать, в каком порядке выполняются операции. Это касается не только «крупных» операций (вызов функции), но и мелких атомарных шагов (чтение нескольких байт из оперативной памяти).

Запуск нескольких потоков может привести к **состоянию гонки** — ошибке в проектировании программы, при которой результат работы программы зависит от порядка переключения между потоками.

Рассмотрим, как избежать состояния гонки, если потоки должны работать с одним и тем же ресурсом. Например, с потоком стандартного вывода.

Простой способ наладить взаимодействие — **блокировка**. Это механизм, позволяющий запретить нескольким потокам делать одно и то же — то есть работать с одним и тем же ресурсом. Работа с одним и тем же ресурсом может привести к неприятным последствиям работы не только программы, но и процессора.

Как это написать в коде? В модуле `threading` есть класс `Lock` (на примере процессов — все то же самое, отличие в использовании модуля `multiprocessing`).

В коде ниже мы создаем блокировку `print_lock`, которая будет использоваться как контекстный менеджер:

```
from threading import Thread, Lock

print_lock = Lock()


def print_multiplication_table(n):
    for i in range(1, 10):
        with print_lock:
            print(n, '*', i, '=', i * n)


threads = [
    Thread(target=print_multiplication_table, args=(i,))
    for i in range(1, 10)
]
for t in threads:
    t.start()
```

Перейдем к более сложному механизму взаимодействия потоков.

Семафор — это механизм, позволяющий ограничить количество потоков, одновременно работающих с одним ресурсом. Семафор не блокирует полностью работу, а только ограничивает.

```
import threading
import time
import socket

domains = [line.strip() for line in open('domains.csv')]
max_connections = 4

connection_sem = threading.BoundedSemaphore(value=max_connections)


def check_free(domain):
    ok = True
    addr = 'python' + domain
    with connection_sem:
```

```
        try:
            socket.getaddrinfo(addr, 80)
        except OSError:
            ok = False
    print(addr + ': ' + ('занято' if ok else 'свободно') + '\n',
end='')


start = time.perf_counter()
threads = []
for d in domains:
    t = threading.Thread(target=check_free, args=(d,))
    t.start()
    threads.append(t)
for t in threads:
    t.join()
end = time.perf_counter()
print(f'{len(domains)} domains in {end-start:0.2f} sec.')

# 1   - 32.86
# 2   - 16.84
# 3   - 15.73
# 4   - 11.45
# 5   - 10.74
# 10  - 11.39
# 100 - 10.61
# 500 - 11.27
# no semaphore - 11.24

# python.mil 11.062690199999906
# python.ac 11.053585699999985
# python.sg 11.051137899999958

# 1   - 6.37
# 2   - 2.92
# 3   - 1.25
# 4   - 1.82
# 5   - 0.97
# 10  - 1.07
# 100 - 4.10
# 500 - 4.14
# no semaphore - 4.18
```

В модуле `threading` есть класс `Semaphore` и класс `BoundedSemaphore`. Разница между ними состоит в том, что `BoundedSemaphore` не может увеличить количество одновременных соединений, которые закладывали изначально в коде, а `Semaphore` может. Лучше использовать `BoundedSemaphore`, потому что необходимость увеличивать количество одновременных соединений, скорее всего, говорит об ошибках в программе, которые класс `BoundedSemaphore` покажет.

Проблема «мёртвой блокировки» (dead lock). Если для операции использовать несколько блокировок, которые будут использоваться несколькими потоками, то программа может просто зависнуть. Она зависнет, потому что потоки запущены параллельно и захватывают обе блокировки, не имея возможности перейти ко второй, так как она захвачена другим потоком. Время ожидания будет бесконечным. Ситуация достаточно специфическая и лучше ее избегать:

```
import threading
import time


lock_1 = threading.Lock()
lock_2 = threading.Lock()


def f1():
    with lock_1:
        time.sleep(0.1)
        with lock_2:
            print('OK')


def f2():
    with lock_2:
        time.sleep(0.1)
        with lock_1:
            print('OK')


t1 = threading.Thread(target=f1)
t2 = threading.Thread(target=f2)
t1.start()
t2.start()
t1.join()
t2.join()
```

Помимо блокировок и семафоров в модуле `threading` существуют другие механизмы взаимодействия потоков, такие как события и условия. О них можно почитать в документации.

Вы не будете реализовывать вручную рассмотренные выше примеры, потому что многопоточное программирование — это особый подход, который нужно тщательно и глубоко изучать. Здесь рассмотрены его механизмы и структура для общего понимания.

5. Очереди и поддержание пула потоков

Рассмотрим пример, когда нам нужно обработать несколько сотен адресов, но при этом нужно поддерживать только 4–5 соединений. Нельзя ли обойтись 4–5 потоками и разделить между ними все задачи? Как реализовать такой случай?

Если разделить весь список на 4–5 частей и каждому потоку дать свою часть, то хорошей конкурентности мы не получим. Один поток может закончить свою работу гораздо быстрее другого, один из потоков может повиснуть на домене, который не отвечает. То есть такое решение не будет эффективным. Для этого нам понадобятся очереди.

Рассмотрим пример создания очереди:

```
from threading import Thread
import time
import socket
from queue import Queue

def main():
    domains = [line.strip() for line in open('domains.csv')]
    max_workers = 4
    start = time.perf_counter()
    jobs = Queue()
    results = Queue()
    create_threads(jobs, results, max_workers)
    process(jobs, domains)
    print_results(results)
    end = time.perf_counter()
    print(f'{len(domains)} domains in {end-start:0.2f} sec.')
```

В функции main мы создаем список всех доменов, задаем максимальное количество «рабочих» (`max_workers`), которые будут выполнять работу и создаем две очереди — очередь задач и очередь результатов. Потоки из первой очереди будут читать задачи, которые необходимо выполнить, а во вторую очередь писать результат.

После этого мы создаем потоки, запускаем выполнение и распечатываем результат.

Чтобы создать очередь, мы используем класс `Queue` из модуля `queue`.

Важно! Если хотим создать очередь для процессов, ее нужно импортировать из модуля `multiprocessing`.

Функция создания потоков сводится к тому, что в цикле мы создаем потоки и запускаем их. При этом в потоке работает функция `worker`, которая будет

рассмотрена ниже. Функции `worker` необходимо передать очередь задач и очередь результатов, с которыми она будет взаимодействовать:

```
def create_threads(jobs, results, max_workers):
    for _ in range(max_workers):
        thread = Thread(target=worker,
                        args=(jobs, results),
                        daemon=True)
        thread.start()
```

Функция выполнения работы `process` состоит в том, чтобы положить в очередь задачи, которые необходимо выполнить, и дождаться их выполнения с помощью метода `join` у самой очереди. Мы не запускаем метод `join` у потоков. Более того, сами потоки в программе объявлены как демоны:

```
def process(jobs, domains):
    for d in domains:
        jobs.put(d)
    jobs.join()
```

В функции распечатки результата мы получаем результаты из очереди `results` и распечатываем:

```
def print_results(results):

    while not results.empty():
        domain, free = results.get()
        print(f'python{domain} is '
            + 'free' if free else 'occupied')
```

Функция `worker` запускает бесконечный цикл, в котором из очереди `jobs` получает очередную задачу, выполняет ее с помощью функции `check_free` и результат записывает в очередь `results`. Важно, что результат записывается в виде двух значений — домена и самого результата. После того как задача выполнена, у очереди `jobs` выполняется метод `task_done`, который сообщает очереди, что одна из задач очереди выполнена. Таким образом, по значению счетчика задач очередь в определенный момент понимает, что все задачи были выполнены:

```
def worker(jobs, results):
    while True:
        domain = jobs.get()
        free = check_free(domain)
        results.put((domain, free))
        jobs.task_done()
```

Поскольку функция `worker` работает в бесконечном цикле, мы должны создать поток как демона. Этот поток завершит работу только в момент завершения работы программы.

Метод `get` очереди — блокирующий. При вызове метода, он блокирует выполнение потока до того момента, пока в очереди что-то появится. Метод не выдаст ошибку, если очередь будет пуста, а будет ждать появление задач.

Время работы абсолютно аналогично результатам работы программы с использованием семафоров:

Кол-во	Время
1	5.30
2	4.07
4	1.19
5	1.56
10	2.01
20	1.12
50	2.46
100	4.14

Преимущество использования очереди состоит в том, что мы не создаем большое количество потоков, а ограничиваем их количество. При создании очереди и потоков нужно иметь в виду, что поток занимает значительный объем оперативной памяти (как минимум 2 Мб).

6. Блокировка GIL в Python

Рассмотрим, как устроено управление памятью в Python. Когда мы создаем объект, под него выделяется память, и в ней сохраняются необходимые данные под объект. Просто так освободить память из-под объекта не получится. Например, в случае удаления, поскольку могла быть создана копия объекта.

Управление памятью в Python осуществляется с помощью счетчика ссылок. При создании каждой новой ссылки на объект, в этом объекте записывается еще одна единица в счетчике ссылок. Интерпретатор всегда знает, сколько существует действующих ссылок на тот или иной объект. Когда все ссылки освобождаются, счетчик ссылок сбрасывается в ноль. И интерпретатор знает, что можно освободить память из-под объекта.

Ниже в коде на C приведен фрагмент программы интерпретатора Python. Код состоит в том, что мы берем некий объект, обращаемся к его свойству `ob_refcnt`, и увеличиваем его на единицу (операция ++).

```
static inline void Py_INCREF(PyObject *op)
{
    op->ob_refcnt++;
}
```

Проблема состоит в операции увеличения на единицу, поскольку мы должны выполнить не одну операцию, а три:

1. Прочитать значение из оперативной памяти.
2. Увеличить его на единицу.
3. Полученное значение записать обратно.

Представим ситуацию, когда два потока выполняют операцию увеличения на единицу одновременно. Например, есть функция, в которую передается аргумент. При создании каждого нового аргумента в двух потоках нужно увеличить счетчик ссылок дважды.

При этом может произойти следующее. Первый поток прочитал значение счетчика ссылок. В этот момент операционная система переключилась на выполнение другого потока, который читает значение, увеличивает его на единицу и записывает в память. После чего операционная система возвращает управление первому потоку, который не знает о том, что значение в счетчике было увеличено на единицу. В итоге вместо увеличения на 2, счетчик увеличивает значение на 1. Единица прибавилась два раза, но записалось в счетчик одно и то же число.

Таким образом, мы получили, что счетчик ссылок имеет некорректное состояние. При уменьшении значения счетчика ссылок до нуля может оказаться, что на самом деле ссылка на объект еще есть. Если ссылка есть, то по ней можно обратиться к памяти — прочитать или записать данные. Но эта память уже освобождена. Обращение к уже освобожденной памяти в языке C — «неопределенное поведение».

Лучшее, что может быть при неопределенном поведении — программа «упадет» с ошибкой. Но может оказаться и так, что мы прочитаем какие-то данные из памяти, и они будут корректными для нашей программы. В этом случае программа начинает выполнять что-то непредсказуемое. Например, если в области лежит байт-код, который не предназначен для выполнения. То есть программа становится уязвимой для злоумышленников.

В Python подобная проблема решается с помощью механизма GIL.

Глобальная блокировка интерпретатора (Global interpreter lock — GIL) — механизм, реализованный в Python, который в каждый момент времени позволяет только одному потоку выполнять байт-код. Переключение между потоками происходит в моменты, когда байт-код уже выполнен.

Этот подход ограничивает возможности многопоточного программирования и не дает выигрыша в производительности.

GIL снимается на время системных вызовов — например, вызова функции `time.sleep`, операций чтения-записи (файлов, пользовательского ввода или сетевых соединений). Если этого не происходит, примерно каждые 5 мс текущий поток прерывается, чтобы дать поработать другим потокам. Также GIL может сниматься в коде, написанном на C (в стандартной библиотеке или дополнительных библиотеках).

Рассмотрим пример работы GIL. В коде представлены две функции, которые работают с одной и той же переменной — одна из функций прибавляет к переменной единицу, вторая вычитает единицу.

В программе мы запускаем два потока. Пока один из потоков жив (метод `is_alive`), мы время от времени распечатываем значение переменной `x`:

```
x = 0

def add():
    global x
    for i in range(10_000_000):
        x += 1

def sub():
    global x
    for i in range(10_000_000):
        x -= 1

t1 = Thread(target=add)
t2 = Thread(target=sub)
t1.start(); t2.start()
while t1.is_alive() or t2.is_alive():
    print(x)
    time.sleep(0.05)
print(x)
```

Данная программа будет работать по-разному в зависимости от версии Python. В версии 3.10 были введены некоторые дополнительные механизмы безопасности при переключении потоков, и представленный выше код будет работать корректно. В любом случае представленный пример — плохой пример многопоточной программы.

GIL — простой способ обеспечения безопасности многопоточного кода, но достаточно накладный, так как не дает в полной мере использовать все ресурсы многопроцессорной системы. Однако любые другие способы сильно замедляют работу в однопоточном режиме (то есть в 99% случаев).

Вопрос использования GIL активно обсуждается в сообществе разработчиков на Python. В том числе предлагаются варианты обхода GIL и другие варианты обеспечения безопасного многопоточного кода. Но пока другого хорошего решения не предложено.

7. Использование универсального интерфейса `concurrent.futures`

Модуль `concurrent.futures` дает простой интерфейс для запуска пула процессов или потоков.

Рассмотрим пример, когда нам нужно запустить несколько функций в многопоточном режиме. В коде ниже рассмотрена функция проверки свободен или занят адрес. После выполнения функция распечатывает требуемую информацию.

Чтобы запустить потоки, нам понадобится класс `ThreadPoolExecuto`.Задаем для него количество «работников», запущенных в потоках и выполняем метод `map`.

Метод `map` похож на обычную функцию `map`. Он принимает функцию и набор данных, на которых эту функцию нужно выполнить. Класс `executor` используется как контекстный менеджер, при выходе из него все потоки будут завершены.

```
from concurrent.futures import ThreadPoolExecutor
import socket

def check_free(domain):
    addr = 'python' + domain
    try:
        socket.getaddrinfo(addr, 80)
    except OSError:
        print(addr + ' is free')

domains = [line.strip() for line in open('domains.csv')]
with ThreadPoolExecutor(max_workers=4) as executor:
    executor.map(check_free, domains)
```

Важно! Если используются потоки, то необходимо использовать класс `ProcessPoolExecutor`.

Более сложный случай — нам нужно получить возвращаемое значение из функции, которое в дальнейшем будет использоваться. Функцию `map` мы не можем использовать, поскольку никаких результатов она не возвращает.

Для данного примера также воспользуемся классом `ThreadPoolExecutor`, но применим метод `submit`. Метод позволяет запустить в исполнителе функцию с набором аргументов, которые передаются методу как параметры.

Метод `submit` возвращает специальный объект `future` — некий объект, который выполняет определенную работу. Но возможно, эту работу объект еще не выполнил.

Мы можем получить результат из объекта с помощью метода `result`. Метод `result` — блокирующий. Поэтому правильным решением для получения результата будет применить специальную функцию `as_completed`. Она отслеживает список будущих объектов и выдает их по мере того, как они выдают результат. Для того чтобы впоследствии мы могли адекватно обработать объекты, мы записываем их в словарь, где каждому «будущему объекту» ставится в соответствие домен, с которым он работал:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import socket

def check_free(domain):
    addr = 'python' + domain
    try:
        socket.getaddrinfo(addr, 80)
    except OSError:
        return True
    return False

domains = [line.strip() for line in open('domains.csv')]
with ThreadPoolExecutor(max_workers=4) as executor:
    future_to_addr = {executor.submit(check_free, d): d
                      for d in domains}
    for f in as_completed(future_to_addr.keys()):
        result = f.result()
        addr = future_to_addr[f]
        print(addr, result)
```

На практике интерфейсов `ThreadPoolExecutor` и `ProcessPoolExecutor` в большинстве случаев будет достаточно, чтобы запустить многопоточный код.

Особенно хорошо будут работать классы для простых случаев. Если задача более сложная, нужно отнестись к проектированию программы внимательно. И прежде всего решить, действительно ли нужен многопоточный код и можно ли обойтись более простыми средствами.

Дополнительные материалы для самостоятельного изучения

1. Concurrent Execution