Оглавление

Неделя 5

Многопоточное и асинхронное программирование

5.1. Введение

Асинхронное и многопоточное программирование — это достаточно сложная тема для изучения. Наше обучение будет разбито на три основных части. В первой части мы рассмотрим выполнение "синхронных программ", и посмотрим на примеры их выполнения в процессах и потоках. Далее мы погрузимся в то, как устроены socket-ы, и рассмотрим выполнение различных сетевых запросов. В заключительной части нашего обучения мы углубимся в то, как устроены генераторы и корутины в языке Python и рассмотрим примеры работы с framework-ом asyncio. Полученные знания могут оказаться полезными для вас, даже если вы не разрабатываете на языке Python. Пройдя обучение, вы сможете увидеть, насколько это легко и удобно сделано в Python и, возможно, сравнить с другими языками программирования.

5.2. Процессы и потоки

5.2.1. Процесс и его характеристики

Процесс — это программа, которая запущена в оперативной памяти компьютера. Другими словами, процесс — это набор инструкций, которые выполняются последовательно. Каждый процесс, который запущен в операционной системе, имеет свои характеристики. Одна из главных характеристик — это идентификатор процесса или PID. Кроме того, каждый процесс занимает некий объем оперативной памяти, которую он запрашивает у системы. Система возвращает запрошенный объём памяти процессу и аллоцирует её. Также у процесса есть стек — он используется для вызова функций и создания локальных переменных у этих функций. И, наконец, у каждого процесса есть список открытых файлов, стандартный ввод и стандартный вывод.

В следующих примерах мы будем использовать операционную систему класса Linux и Python 3.

Давайте попробуем узнать, какие процессы запущены в операционной системе. Для этого нам потребуется консоль и команда `top` — она отображает список процессов, которые сейчас функционируют в операционной системе. В виде колонок в таблице мы видим характеристики процессов:

```
top - 15:07:23 up 1 day, 13:57,  2 users,  load average: 0,27, 0,10, 0,09
Tasks: 192 total,   1 running, 191 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,0 us,  5,3 sy,  0,0 ni, 79,4 id, 15,3 wa,  0,0 hi,  0,0 si,  0
КиБ Mem :  1022740 total,   213616 free,   375656 used,   433468 buff/ca
КиБ Swap:  1046524 total,   891248 free,   155276 used.   474276 avail M
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2308	shveenk+	20	0	9340	3712	3180	R	2,3	0,4	0:00.31	top
911	root	20	0	159188	28404	7376	S	0,6	2,8	9:19.43	Xorg
2001	shveenk+	20	0	307196	80848	16224	S	0,6	7,9	25:42.90	compiz
3	root	20	0	0	0	0	S	0,3	0,0	1:11.08	ksofti+
7	root	20	0	0	0	0	S	0,3	0,0	2:32.33	rcu_sc+
178	root	0	-20	0	0	0	S	0,3	0,0	1:44.35	kworke+
1499	root	20	0	81336	2496	2032	D	0,3	0,2	0:12.51	upowerd
1738	root	20	0	0	0	0	S	0,3	0,0	0:03.27	kworke+
1899	shveenk+	20	0	25664	148	0	S	0,3	0,0	0:55.86	gpg-ag+
1	root	20	0	25088	3528	2352	S	0,0	0,3	0:38.32	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.13	kthrea+
5	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	kworke+
8	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_bh
9	root	rt	0	0	0	0	S	0,0	0,0	0:00.00	migrat+
10	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	lru-ad+
11	root	rt	0	0	0	0	S	0,0	0,0	0:07.80	watchd+
12	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/0

В таблице указан PID (идентификатор процесса), пользователь, из-под которого был запущен процесс (определяет права, которые будут доступны этому процессу в операционной системе), размер виртуальной и физической памяти и т.д. Как мы видим, процессов в операционной системе достаточно много, и все они работают на первый взгляд параллельно. На самом деле это не так. Планировщик операционной системы выделяет небольшой квант времени каждому процессу, исполняет его и затем происходит переключение между процессами. Таким образом, процессы выполняются последовательно, но из-за того, что квант времени небольшой, нам кажется, что все они выполняются параллельно.

Давайте попробуем запустить наш первый Python процесс и изучим его характеристики средствами операционной системы. Сначала импортируем два модуля — `time` и `os`. Затем при помощи вызова функции модуля `os.getpid` мы получаем идентификатор процесса, запоминаем его в переменную `pid` и в бесконечном цикле выводим PID нашего процесса и системное время каждые 2 секунды:

```
# простой Python процесс

import time
import os

pid = os.getpid()

while True:
    print(pid, time.time())
    time.sleep(2)
```

```
python ex1.py
2321 1488521934.518766
2321 1488521936.520758
2321 1488521938.522762
...
```

Мы видим, что запустился процесс. Он вывел `pid 2321`, системное время и продолжает это делать бесконечно в цикле. Давайте попробуем найти наш процесс в списке всех процессов. Для этого нам поможет команда `ps` с флагами `aux` (подробную информацию о флагах можно посмотреть в документации). Эта команда отобразит список всех процессов. Для того чтобы найти конкретно наш процесс, можно отфильтровать выдачу `ps aux` при помощи команды `grep`. Мы увидим наш процесс (в данном случае в первой строчке):

```
shveenkov@coursera:~$ ps axu | grep ex1.py
shveenk+  2321  0.5  0.6  12548  6580 pts/21   S+   15:09   0:00 python3 ex
1.py
shveenk+  2339  0.0  0.0   6356   840 pts/22   S+   15:10   0:00 grep --col
or=auto ex1.py
```

Чтобы посмотреть название характеристик, можно вывести первую строчку от результатов вывода `ps` при помощи следующей команды:

```
shveenkov@coursera:~$ ps aux | head -1; ps axu | grep ex1.py
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
shveenk+  2321  0.3  0.6  12548  6580 pts/21   S+   15:09   0:00 python3 ex
1.py
shveenk+  2352  0.0  0.0   6356   900 pts/22   S+   15:10   0:00 grep --col
or=auto ex1.py
```

Итак, процесс с PID-ом 2321 виден в результатах вывода команды ps. Он потребляет немного центрального процессора и занимает 6 килобайт физической памяти. Также видим командную строчку, при помощи которой он был запущен (`python3 ex1.py`).

Какую последовательность команд выполняет наш процесс? Вообще, когда процессы выполняются в операционной системе, они делают системные вызовы. Системные вызовы выполняет непосредственно ядро операционной системы, а результаты этих системных вызовов возвращаются к процессу, который их вызвал. Например, вывод в консоль,

или стандартный вывод — это системный вызов. Чтобы посмотреть, какие системные вызовы делает наш процесс, можно воспользоваться командой `strace`, указав ей PID нашего процесса (для этой команды нужны дополнительные права):

```
shveenkov@coursera:~$ sudo strace -p 2321
sudo: unable to resolve host coursera: Время ожидания соединения истекло
[sudo] пароль для shveenkov:
strace: Process 2321 attached
_newselect(0, NULL, NULL, NULL, {1, 537604}) = 0 (Timeout)
clock_gettime(CLOCK_REALTIME, {1491653538, 554869392}) = 0
write(1, "2321 1491653538.5548694\n", 24) = 24
clock_gettime(CLOCK_MONOTONIC, {136954, 582212491}) = 0
_newselect(0, NULL, NULL, NULL, {2, 0}
) = 0 (Timeout)
clock_gettime(CLOCK_REALTIME, {1491653540, 606849458}) = 0
write(1, "2321 1491653540.6068494\n", 24) = 24
clock_gettime(CLOCK_MONOTONIC, {136956, 629594142}) = 0
 newselect(0, NULL, NULL, NULL, {2, 0}
```

В результате видим, что вызывается системный вызов `write`. У него в аргументах есть файловый дескриптор 1 — стандартный вывод. Также видим, что в стандартный вывод попадает PID нашего процесса и системное время, а для вызова `sleep` используются другие дополнительные системные вызовы. Для выхода используем Ctrl+C.

Итак, мы узнали, что процесс во время исполнения общается с операционной системой при помощи системных вызовов. Давайте посмотрим на список файлов, которые открыты в нашем процессе. Для этого можно воспользоваться командой `lsof`, указав PID процесса (`lsof -p 2321`). Мы увидим, что процесс использует множество Python-библиотек. Но самое главное, что нас сейчас интересует, — это стандартный поток ввода, вывода и поток ошибок (файловые дескрипторы 0, 1 и 2):

```
python3 2321 shveenkov    0u   CHR 136,21      0t0     24 /dev/pts/21
python3 2321 shveenkov    1u   CHR 136,21      0t0     24 /dev/pts/21
python3 2321 shveenkov    2u   CHR 136,21      0t0     24 /dev/pts/21
```

Мы видим, что эти файловые дескрипторы равны терминалу. Если мы будем делать стандартный вывод в файл (при помощи команды `python3 exq.py > log.txt`), мы обнаружим, что стандартный вывод у нашего процесса поменялся на файл:

```
python3 2367 shveenkov    0u   CHR 136,21      0t0     24 /dev/pts/21
python3 2367 shveenkov    1w   REG    8,1        0 545637 /home/shveenkov/c
oursera/01/log.txt
python3 2367 shveenkov    2u   CHR 136,21      0t0     24 /dev/pts/21
```

5.2.2. Создание процессов

Поговорим про создание процессов в Python. В этом разделе мы узнаем, как создать дочерний процесс, как работает системный вызов `fork`, а также рассмотрим примеры со-

здания процессов при помощи модуля `multiprocessing`.

Процесс в операционной системе создается при помощи системного вызова `fork`. Давайте рассмотрим программу, которая создает дочерний процесс при помощи системного вызова `fork`. Импортируем пару модулей `time` и `os`, затем вызываем системный вызов `fork`:

```
# Создание процесса на Python

import time
import os

pid = os.fork()
if pid == 0:
    # дочерний процесс
    while True:
        print("child:", os.getpid())
        time.sleep(5)
else:
    # родительский процесс
    print("parent:", os.getpid())
    os.wait()
```

```
parent: 2411
child: 2412
```

Системный вызов `fork` создает точную копию родительского процесса. Это означает, что вся память, все файловые дескрипторы и все ресурсы, которые были доступны в родительском процессе, будут целиком и полностью скопированы в дочерний процесс. С того момента, как системный вызов `fork` отработал, у нас имеется два одинаковых процесса в операционной системе. Единственное отличие заключается в том, что системный вызов `fork` в родительский процесс вернет PID дочернего процесса, а в дочернем процессе переменная `pid` будет равна нулю. Код в блоке `if` будет исполнен в дочернем процессе, а код, который находится за веткой `else`, будет исполнен в родительском процессе. Также в родительском процессе мы вызываем системный вызов `os.wait`, это еще один дополнительный системный вызов и он позволяет нам дожидаться завершения созданного дочернего процесса. А в дочернем процессе в бесконечном цикле выводится PID этого процесса каждые пять секунд.

Можно отобразить результаты команды ps в иерархическом виде, чтобы посмотреть, какой из процессов родительский, а какой дочерний. Делается это при помощи дополнительного флага `f`:

```
shveenkov@coursera:~$ ps axf | grep ex2.py
 2411 pts/21   S+     0:00                    |   \_ python3 ex2.py
 2412 pts/21   S+     0:00                    |       \_ python3 ex2.py
 2418 pts/22   S+     0:00                    |   \_ grep --color=auto ex
2.py
```

Если мы вызовем команду `strace`, увидим, что наш созданный дочерний процесс делает системный вызов `write` и выводит информацию в стандартный поток вывода. А родительский процесс сделал системный вызов `wait`, и операционная система сама оповестит его о том, когда дочерний процесс завершится.

Давайте остановимся ещё раз на памяти в родительском и дочернем процессах и рассмотрим пример. Итак, у нас есть программа, мы объявили в ней переменную `foo`, присвоили ей значение `"bar"` и делаем системный вызов `fork`:

```
# Память родительскогo и дочернего процесса

import os

foo = "bar"

if os.fork() == 0:
    # дочерний процесс
    foo = "baz"
    print("child:", foo)
else:
    # родительский процесс
    print("parent:", foo)
    os.wait()
```

После того, как отработал системный вызов `fork`, как уже было сказано, вся память целиком и полностью будет скопирована из родительского процесса в дочерний. Значит, переменная `foo` будет доступна в дочернем процессе со значением `"bar"`. Но если мы изменим значение `foo` в дочернем процессе, это никак не повлияет на переменную `foo`, которая была объявлена в родительском процессе:

```
parent: bar
child: baz
```

Итак, мы узнали, что память в дочерний процесс копируется, и что дочерний и родительский процессы пользуются *разной памятью*.

То же самое относится и к файловым дискрипторам. Предположим, у нас есть небольшой файл с двумя строчками: `example string1` и `example string2`. Открываем файл на чтение и читаем в переменную `foo` одну строчку. После того, как мы считали одну строчку, делаем системный вызов `fork`. После этого у нас создается точная копия родительского процесса:

```
# Файлы в родительском и дочернем процессе

# cat data.txt
# example string1
# example string2

import os

f = open("data.txt")
foo = f.readline()

if os.fork() == 0:
    # дочерний процесс
    foo = f.readline()
    print("child:", foo)
else:
    # родительский процесс
    foo = f.readline()
    print("parent:", foo)
```

Если мы в дочернем процессе снова вызовем метод `readline()` у объекта `f`, то мы прочитаем уже вторую строчку из этого файла. Но это никак не повлияет на родительский процесс. В родительском процессе, если мы вызовем `readline()`, то мы точно так же считаем вторую строчку:

```
parent: example string2
child: example string2
```

Итак, ещё раз обращаем внимание, что не только память, но и файловые дискрипторы целиком и полностью копируются в дочернем процессе, когда мы делаем системный вызов `fork`.

Все эти примеры носят обучающий характер, и обычно код с использованием системных вызовов `fork` немного сложнее. `fork` может вернуть ошибку, которую нужно проверять, поэтому обычно в Python-е для создания процессов используют модуль `multiprocessing`. Для того, чтобы запустить процесс таким способом, необходимо импортировать класс `Process` из модуля `multiprocessing`, создать объект класса `Process`, передать ему в конструктор функцию, которую мы хотим исполнить в отдельном дочернем процессе и аргументы этой функции. Процесс будет создан тогда, когда мы вызовем метод `start` нашего объекта. Внутри метода `start` будет вызван системный вызов `fork` и исполнена наша функция `f` в отдельном процессе. Очень важно ожидать завершения всех созданных дочерних процессов. Для этого можно воспользоваться удобной функцией `join`:

```
# Создание процесса, модуль multiprocessing

from multiprocessing import Process

def f(name):
    print("hello", name)

p = Process(target=f, args=("Bob",))
p.start()
p.join()
```

```
hello Bob
```

Как мы видим, системные вызовы `fork` и `wait` спрятаны внутри красивых оберток. Вообще, не в каждой операционной системе есть системный вызов `fork`, и поэтому в `multiprocessing` все аккуратно сделано за вас.

Существует также альтернативный метод создания процесса при помощи `multiprocessing` — используя механизм наследования. Для этого мы объявляем свой класс, наследуемся от класса `Process`. В конструктор передаем нужные параметры для функции, которая должна быть запущена в дочернем процессе и переопределяем метод run. В методе `run` мы реализуем код, который должен выполнять дочерний прочесс. Далее создаем объект нашего класса `PrintProcess`, передаем туда параметры, вызываем метод `start`. Метод `start` вызовет `fork` и выполнит наш код в дочернем процессе. Для завершения дочернего процесса мы вызываем метод `join`:

```
# Создание процесса, модуль multiprocessing

from multiprocessing import Process

class PrintProcess(Process):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        print("hello", self.name)

p = PrintProcess("Mike")
p.start()
p.join()
```

```
hello Mike
```

Очень важно ожидать завершения всех дочерних процессов, чтобы контролировать освобождение всех ресурсов.

5.2.3. Создание потоков

В этом разделе мы поговорим о потоках. Мы обсудим создание потоков при помощи модуля `threading` и использование класса `ThreadPoolExecutor`. С прикладной точки зрения поток целиком и полностью напоминает процесс. Он имеет свою последовательность инструкций для исполнения, у каждого потока есть свой собственный стек, но все потоки выполняются в рамках одного процесса. Этим они отличаются от процессов. Если, когда мы говорили о процессах, у каждого процесса были свои ресурсы и память, то все созданные потоки разделяют память процесса и все его ресурсы. Управлением и выполнением потоков занимается операционная система. Но в Python есть свои ограничения для потоков — их мы обсудим отдельно.

Итак, давайте рассмотрим пример создания потока на Python. Всё будет очень похоже на создание процессов. Во-первых, используем модуль `threading` и импортируем из него класс `Thread`. Далее мы объявляем функцию, которую хотим исполнить в отдельном потоке и создаем объект класса `Thread`, передав в него нашу функцию `f` и аргументы, с которыми эта функция должна быть вызвана. После того как мы создали объект, никакого потока запущено не будет — он будет запущен, когда мы вызовем метод `start` у этого объекта. Также очень важно дожидаться выполнения завершения всех созданных потоков при помощи метода `join`:

```
# Создание потока

from threading import Thread

def f(name):
    print("hello", name)

th = Thread(target=f, args=("Bob",))
th.start()
th.join()
```

```
hello Bob
```

Существует также альтернативный метод создания потока при помощи наследования. Опять же, всё очень похоже на использование модуля `multiprocessing`. Мы объявляем свой класс, наследуемый от класса `Thread` и в конструктор передаем нужные аргументы, чтобы выполнить функцию в отдельном потоке:

```
# Создание потока

from threading import Thread

class PrintThread(Thread):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        print("hello", self.name)

th = PrintThread("Mike")
th.start()
th.join()
```

```
hello Mike
```

В Python3 появился очень удобный класс для создания пула потоков — `ThreadPoolExecutor` из модуля `concurrent.futures`. Предположим, у нас есть некий массив чисел, и нам нужно с помощью ограниченного количества потоков рассчитать квадраты этих чисел. Для этого можно использовать контекстный менеджер, указать в нем вызов `ThreadPoolExecutor` с параметром `max_workers`, который как раз отвечает за максимальное количество потоков, которые будут созданы в этом блоке `with`. Нужное количество потоков будет создано автоматически, и при завершении контекстного менеджера будет вызвана функция `shutdown`, которая дождется завершения всех созданных потоков. Основная функция у `ThreadPoolExecutor` — это метод `submit`. Он создает объект класса
`concurrent.futures.Future` — это такой объект, который еще не завершился, но выполняется и будет завершен в будущем. При помощи удобного метода `as_completed` из модуля `concurrent.futures` мы можем дождаться завершения всех объектов и получить результаты по мере завершения всех созданных нами потоков:

```
# Пул потоков, concurrent.futures.Future

from concurrent.futures import ThreadPoolExecutor, as_completed

def f(a):
    return a * a

# .shutdown() in exit
with ThreadPoolExecutor(max_workers=3) as pool:
    results = [pool.submit(f, i) for i in range(10)]

    for future in as_completed(results):
        print(future.result())
```

```
0
1
4
9
...
```

5.2.4. Синхронизация потоков

В этом видео мы поговорим про синхронизацию потоков и обсудим очереди, блокировки и условные переменные. Если вы запустите несколько потоков для решения своей задачи, то вам рано или поздно придётся обмениваться данными между потоками. Часто для синхронизации потоков используют блокировки. Но любые блокировки замедляют выполнение программы. Лучше не использовать блокировки и отдавать предпочтение обмену данными через очереди.

Давайте для начала разберёмся, как можно использовать модуль `queue` и очереди для обмена данными между потоками. Использование очередей выглядит достаточно простым. В следующем примере мы создаём объект типа очередь с максимальным размером 5. Для помещения элементов в очередь необходимо использовать метод `put` объекта `Queue`. Обращаем ваше внимание, что если в очереди будет уже пять элементов, то вызов метода `put` заблокирует выполнение потока, который вызвал этот метод, и будет ждать, пока в очереди не появится свободное место. Итак, для обработки сообщений этой очереди мы создаём пару потоков — объектов класса `Thread`. Передаём в этот объект функцию `worker`, которой передаём нашу очередь. Итак, функция `worker` будет выполняться в двух независимых параллельных потоках. Каждый поток в бесконечном цикле будет получать сообщение из очереди при помощи вызова метода `get` у объекта `q`:

```
# 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()
```

```
process data: 1 0
process data: 1 1
process data: 2 3
...
```

Большое внимание нужно уделить правильному завершению потока. Ресурсами процесса, то есть выделенной памятью или открытым файлом владеет сам процесс. Но процесс ничего не знает о том, что делает с этими ресурсами поток. И если поток завершить аварийно, то файл может остаться незакрытым, блокировка — невысвобожденной, что теоретически может привести к непредвиденным последствиям. Поэтому в Python не существует функции аварийного завершения потока. Очень важно делать это правильно в функции самого потока. На приведённом выше примере в очередь помещается специальное значение None, и функция потока при проверке условия завершает свою работу.

Использование очередей делает код выполняемой программы более простым. И, по возможности, лучше разрабатывать код таким образом, чтобы не было глобального разделяемого ресурса или состояния. Тем не менее, иногда приходится использовать блокировки.

Давайте рассмотрим пример. Предположим, у нас есть класс `Point`, и у класса `Point` есть координаты `x` и `y`. Также у этого класса есть метод `get`, который возвращает координаты, и метод `set`, который задаёт новые координаты. Предположим, что мы создали объект класса `Point` и используем этот объект в большом количестве потоков. Некоторые потоки вызывают метод `get`, другие вызывают метод `set`. Если бы не было блокировок, то могла возникнуть ситуация, когда один поток изменил значение координаты `x`, а другой поток в это время вернул координаты `x` и `y`. Мы получили неконсистентное состояние

объекта, у которого одна координата изменена, а вторая нет.

```
# Синхронизация потоков, race condition

import threading

class Point(object):
    def __init__(self, x, y):
        self.set(x, y)

    def get(self):
        return (self.x, self.y)

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

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

Чтобы избежать подобных ситуаций, нужны блокировки. Для создания блокировки используют метод `threading.RLock()`. Создаём объект блокировки, и теперь при входе в контекстный менеджер мы захватываем блокировку, а при выходе блокировка высвобождается:

```
# 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()
```

Таким образом можно легко и удобно создавать блокировки на Python. Подобные ситуации иногда называют гонкой за ресурсами, или race condition.

Давайте рассмотрим ещё один вариант применения блокировок. Их можно использовать без контекстного менеджера. На примере мы создаём объекты класса `RLock` и затем вызываем методы `acquire`, чтобы получить или захватить блокировку, и метод `release` для того, чтобы высвободить её:

```
# Синхронизация потоков, блокировки

import threading


a = threading.RLock()
b = threading.RLock()

def foo():
    try:
        a.acquire()
        b.acquire()
    finally:
        a.release()
        b.release()
```

Если мы запустим подобный код в большом количестве процессов, то рано или поздно это приведёт к ситуации, которая называется deadlock. Дело в том, что мы освобождаем блокировки в неправильной последовательности. Нужно учитывать это в своих программах и отдавать предпочтение использованию контекстного менеджера при работе с блокировками. Также в Python существует ещё и объект класса `Lock`, а не `RLock`, но предпочтительнее использовать объекты `RLock` — они позволяют в одном потоке получить блокировку дважды.

В Python существует ещё один механизм для синхронизации потоков — он называется *условные переменные*. Давайте рассмотрим класс `Queue`. Это очередь, с которой нужно будет работать в большом количестве потоков. У неё есть операции `put` и `get`, и, конечно же, у неё есть размер. Если мы выполним операцию `put`, а в очереди уже достаточно большое количество элементов, то нам необходимо ждать пока это количество уменьшится. Вопрос — сколько ждать? Неизвестно. Для решения подобной задачи можно использовать условные переменные. Условная переменная `threading.Condition` получает в конструктор объект блокировки `self._mutex` (он есть по умолчанию, но если условные переменные взаимозависимые, то необходимо использовать общую блокировку). И при помощи этих условных переменных легко и удобно ожидать событий при помощи вызова `wait` и оповещать все потоки, которые сейчас ждут наступления этого события с помощью функции `notify`:

```
# Синхронизация потоков, условные переменные

class Queue(object):
    def __init__(self, size=5):
        self._size = size
        self._queue = []
        self._mutex = threading.RLock()
        self._empty = threading.Condition(self._mutex)
        self._full = threading.Condition(self._mutex)

    def put(self, val):
        with self._full:
            while len(self._queue) >= self._size:
                self._full.wait()

            self._queue.append(val)
            self._empty.notify()

    def get(self):
        with self._empty:
            while len(self._queue) == 0:
                self._empty.wait()

            ret = self._queue.pop(0)
            self._full.notify()
            return ret
```

Таким образом можно реализовать очередь в Python, которая работает в многопоточной программе.

Все механизмы блокировки и обмена данными между потоками имеют место и для процессов, но вместо модуля `threading` нужно использовать `multiprocessing`.

5.2.5. Глобальная блокировка интерпретатора

Поговорим о том, что такое глобальная блокировка интерпретатора, или, как её сокращенно называют, GIL. GIL очень тесно связан с выполнением потоков. Многие разработчики знают о GIL лишь то, что это какая-то штука, которая не позволяет одновременно двум потокам выполняться на одном ядре процессора, даже если этих ядер у процессора несколько. Тем не менее, GIL в первую очередь предназначен для защиты памяти интерпретатора от разрушений и делает все операции с памятью атомарными. Давайте рассмотрим следующую программу:

```
# cpu bound programm

from threading import Thread
import time

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

# series run
t0 = time.time()
count(100_000_000)
count(100_000_000)
print(time.time() - t0)

# parallel run
t0 = time.time()
th1 = Thread(target=count, args=(100_000_000,))
th2 = Thread(target=count, args=(100_000_000,))

th1.start(); th2.start()
th1.join(); th2.join()
print(time.time() - t0)
```

Можем посмотреть при помощи команды `top`, что этот пример потребляет почти 100% CPU на одном ядре. Давайте вернемся к программе и рассмотрим, что она делает. Прежде всего, у нас есть функция `count`, которая в цикле уменьшает значение счетчика до нуля. Нам необходимо выполнить два вызова этой функции со значением `100_000_000` и засечь, сколько времени займет выполнение двух функций с этим счетчиком. Функция потребляет только центральный процессор и не делает никаких операций ввода-вывода,

не ходит в сеть. Для сравнения мы выполним эту функцию в потоке. Создадим два потока при помощи уже известных нам ранее методов модуля `threading`. Передадим туда эту функцию, те же самые аргументы, запустим наши потоки, подождем, пока они завершатся при помощи метода `join`, и выведем количество секунд, которое было потрачено на выполнение работы этих двух потоков:

```
15.281397104263306
15.86177659034729
```

Видим, что параллельное выполнение при помощи потоков заняло больше времени. Как же так? Тогда зачем нужны потоки, и почему так происходит? Всё дело в глобальной блокировке интерпретатора.

Дело в том, что потоки при выполнении своего кода каждый раз получают блокировку интерпретатора. Если у нас задача CPU-bound (так называют задачи, которые потребляют только ресурсы процессора), то код, написанный с использованием тредов в Python, будет неэффективным. Он будет работать медленнее, чем код, который запущен последовательно. Тем не менее, если мы код нашей функции заменим, например, на задачу, которая требует операции ввода-вывода, то мы заметим большой прирост в итоговом времени выполнения, если сравнивать последовательное выполнение и выполнение в тредах.

Схематично изобразим процесс выполнения потока. Есть поток, в котором выполняется Python-код, и каждый раз Python-интерпретатор пробует получить глобальную блокировку интерпретатора. Если Python выполняет операцию ввода-вывода или системный вызов, он снимает блокировку, и далее выполнение происходит без блокировки. Поэтому если таких будет потоков много, все задачи с вводом-выводом, с ожиданием завершения для операций ввода-вывода будут очень хорошо параллелиться:

```
# как выполняется поток?

a      r      a            r              a          r    a
  run  |------|    run     |--------------|   run    |----| run
------>|  IO  |----------->|      IO      |--------->| IO |----->
       |------|            |--------------|          |----|
a      r      a            r              a          r    a

a - acquire GIL
r - release GIL
```

Это нужно учитывать в задачах, в которых вы будете применять потоки или процессы. GIL реализован внутри как обычная нерекурсивная блокировка или объект класса `threading.Lock`. Все потоки спят пять миллисекунд в ожидании получения блокировки, и если работает один главный поток, то он не требует освобождения этой глобальной блокировки интерпретатора. Итак, в этом видео мы обсудили, что такое GIL и какое отношение он имеет к потокам в Python. Так, Python-потоки — это обычные потоки, или POSIX threads, но с ограничениями в виде глобальной блокировки интерпретатора. Все потоки

выполняются с захватом GIL, но для системных вызовов и операций ввода-вывода GIL не нужен. Итак, мы рассмотрели вопросы про то, как работают потоки и процессы, и в следующих разделах мы рассмотрим, как устроены сокеты и как работать с сетью с применением полученных знаний о потоках и процессах.

5.3. Работа с сетью, сокеты

5.3.1. Сокеты, клиент-сервер

В следующих разделах мы будем изучать то, как устроены сокеты и как работают сетевые программы. Сокеты — это кросс-платформенный механизм для обмена данными между отдельными процессами. Эти процессы могут работать на разных серверах, они могут быть написаны на разных языках, и, прежде всего, программа на Python, которая использует механизм сокетов, осуществляет системные вызовы и взаимодействие с ядром операционной системы. Как правило, для организации сетевого взаимодействия нужен сервер, который изначально создает некое соединение и начинает «слушать» все запросы, которые поступают в него, и программа-клиент, которая присоединяется к серверу и отправляет ему нужные данные.

Давайте рассмотрим пример серверной программы. Чтобы создать сокет, мы должны импортировать модуль `socket`. Далее мы должны создать объект типа `socket` из модуля `socket`. В него необходимо передать некоторые параметры. В данном случае это некоторое семейство address family `AF_INET`, а также тип сокета (потоковый сокет). (Полную информацию по типам сокетов и по типам address family можно посмотреть в документации Python, либо в документации про то, как устроена сеть в операционной системе Linux.) Итак, мы создали объект `socket`. Далее мы должны вызвать метод `bind`. В метод `bind` мы должны передать некую адресную пару — это хост и порт. В качестве хоста в данном случае мы передаем 127.0.0.1 — наш сервер будет слушать все входящие соединения только локально на одной машине. Если мы укажем пустую строчку, либо адрес 0.0.0.0, то наш сервер будет слушать входящие соединения со всех интерфейсов. Порт — это некая целочисленная константа, существуют некоторые зарезервированные порты, например, 80-й порт (обычно на нем работает HTTP-сервер), 43-й порт, 443-й порт. Как правило, порты с номерами до 2000 являются системными, и мы должны использовать адреса больше значений 2000, но максимальное значение для порта — это 65535.

Итак, системный вызов `bind` зарегистрировал нашу адресную пару в операционной системе. Далее, для того чтобы начать принимать соединения, мы должны вызвать метод `listen`. У метода `listen` есть необязательный параметр — это так называемый backlog, или размер очереди входящих соединений, которые еще не обработаны, для которых не был вызван метод `accept`. Если наш сервер будет не успевать принимать входящие соединения, то все эти соединения будут копиться в backlog, и если он превысит максимальное значение, то операционная система выдаст ошибку `ConnectionRefused` для клиентской программы. Далее мы должны вызвать метод `accept`, чтобы начать принимать входящее клиентское соединение. Системный вызов `accept` по умолчанию заблокируется, до тех пор, пока не появится клиентское соединение. Итак, если клиент вызовет

метод connect, то наш метод accept вернет нам объект, который будет являться полнодуплексным каналом. У этого объекта будут доступны методы записи в этот канал и методы чтения. В нашем примере мы в бесконечном цикле будем вызывать чтение из нашего полнодуплексного канала. Если мы ничего не прочитали, это будет означать, что клиент закрыл соединение и серверу тоже необходимо прекратить работу. Данные, которые мы прочитали с канала, мы выводим в консоль. После того как мы закончили работу с нашим клиентом, мы вызываем метод close для нашего объекта, который представляет собой полнодуплексный канал, а также закрываем сокет, который слушает новые соединения со стороны клиента:

```
# создание сокета, сервер

import socket

# https://docs.python.org/3/library/socket.html
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(("127.0.0.1", 10001))   # max port 65535
sock.listen(socket.SOMAXCONN)

conn, addr = sock.accept()
while True:
    data = conn.recv(1024)
    if not data:
        break
    # process data
    print(data.decode("utf8"))

conn.close()
sock.close()
```

Давайте рассмотрим код на стороне клиента. Чтобы установить соединение с сервером, мы должны создать объект типа socket.socket. По умолчанию создается потоковый сокет с семейством address family AF_INET. После этого мы должны вызвать метод connect. Connect заблокируется до тех пор, пока сервер со своей стороны не вызовет метод accept. После того как системный вызов connect отработал, наш сокет готов к работе, и для него можно вызывать методы send, sendall или recv, чтобы получать данные с сервера. После того как мы завершили работу с нашим клиентским сокетом, необходимо вызвать метод close:

```
# создание сокета, клиент

import socket

sock = socket.socket()
sock.connect(("127.0.0.1", 10001))
sock.sendall("ping".encode("utf8"))
sock.close()
```

В Python существует более короткая запись для создания клиентского сокета — это вызов метода модуля `socket` `create_connection`. В `create_connection` мы передаем адресную пару и необязательный `timeout` (про который мы ещё будем говорить в следующих разделах). Этот вызов возвращает нам соединение, готовое для того, чтобы делать отправку или прием данных.

```
# создание сокета, клиент
# более короткая запись

sock = socket.create_connection(("127.0.0.1", 10001))
sock.sendall("ping".encode("utf8"))
sock.close()
```

Теперь мы можем запустить сервер и затем клиент, который отправит серверу байтовую строку `"ping"` (обращаем ваше внимание, что при работе с данными по сети мы вынуждены отправлять именно байты, а не строки).

Как мы уже говорили, `socket` — это кроссплатформенный механизм и необязательно программа-клиент и сервер должны быть написаны на одном и том же языке. Наш пример будет работать и в том случае, если мы воспользуемся вместо клиента программой telnet.

Соединения и сокеты необходимо корректно закрывать. Поэтому в Python существует более удобный механизм для работы с сокетами в виде контекстных менеджеров. Давайте рассмотрим пример, в котором мы выполняем те же самые задачи, но используем контекстный менеджер. Итак, мы используем на стороне сервера конструкцию `with` `socket.socket`, создаём наш объект типа `socket`, вызываем методы `bind`, `listen`. Затем в бесконечном цикле вызываем `accept` и получаем новые соединения от клиентов. Для полученного объекта мы опять используем контекстный менеджер и мы не заботимся о вызовах метода `close`:

```
# создание сокета, контекстный менеджер
# сервер
import socket

with socket.socket() as sock:
    sock.bind(("", 10001))
    sock.listen()

    while True:
        conn, addr = sock.accept()
        with conn:
            while True:
                data = conn.recv(1024)
                if not data:
                    break
                print(data.decode("utf8"))
```

После того как контекстный менеджер завершит свою работу, он автоматически вызовет метод `close` для нужных нам объектов. Это очень удобно, и это позволяет допускать вам меньшее количество ошибок при работе с сокетами. На стороне клиента мы снова используем контекстный менеджер для вызова `socket.create.connection`:

```
# клиент
import socket

with socket.create_connection(("127.0.0.1", 10001)) as sock:
    sock.sendall("ping".encode("utf8"))
```

Предпочтительнее работать с контекстными менеджерами при написании клиент-серверных программ на языке Python.

5.3.2. Таймауты и обработка сетевых ошибок

Сеть не всегда может работать стабильно, поэтому в сетевых программах необходимо обрабатывать различные ошибки. Давайте рассмотрим обучающий пример с обработкой ошибок. Рассмотрим немного изменённый код сервера из предыдущего раздела. Здесь после того, как мы получили соединение с помощью команды `sock.accept()`, мы устанавливаем timeout командой `settimeout`. Значение таймаута по умолчанию — это None, а значение 0 переведёт сокет в неблокирующий режим, про который мы будем говорить позднее. Мы установили таймаут, равный 5 секундам:

```
# создание сокета, таймауты и обработка ошибок
# сервер
import socket

with socket.socket() as sock:
    sock.bind(("", 10001))
    sock.listen()
    while True:
        conn, addr = sock.accept()
        conn.settimeout(5)  # timeout := None|0|gt 0
        with conn:
            while True:
                try:
                    data = conn.recv(1024)
                except socket.timeout:
                    print("close connection by timeout")
                    break

                if not data:
                    break
                print(data.decode("utf8"))
```

Теперь, если после вызова `recv` нам не поступит данных в течение 5 секунд, будет сгенерировано исключение `socket.timeout`. Затем мы закрываем соединение.

Рассмотрим код на клиенте. На клиенте существует connect timeout и socket read timeout. Connect timeout мы задаём в методе `create_connection`, он будет распространяться только на установку соединения с нашим сервером. После того, как соединение будет установлено, мы можем задать socket read timeout на все операции с нашим сокетом:

```
# создание сокета, таймауты и обработка ошибок
# клиент
import socket

with socket.create_connection(("127.0.0.1", 10001), 5) as sock:
    # set socket read timeout
    sock.settimeout(2)
    try:
        sock.sendall("ping".encode("utf8"))
    except socket.timeout:
        print("send data timeout")
    except socket.error as ex:
        print("send data error:", ex)
```

Также могут возникнуть другие исключения, которые тоже нужно обрабатывать. Базовый класс для этих исключений — это `socket.error`.

5.3.3. Обработка нескольких соединений

Итак, в предыдущих разделах мы рассматривали простые программы типа клиент-сервер и пробовали организовать взаимодействие между двумя процессами. А что делать, если этих процессов будет несколько или не несколько, а очень много? Если мы приняли соединение и начинаем его обработку в том же самом потоке управления, мы не можем принимать новые соединения. Если у нас будет большое количество клиентов, то все остальные клиенты будут вынуждены ждать, пока мы закончим работу с первым соединением. Какие подходы существуют для решения данной задачи?

Конечно, мы можем создать процесс или поток для обработки отдельного соединения и выполнить в этом процессе или потоке код по его обработке. Если мы создадим 10,000 процессов, это будет иметь ряд своих минусов. Как минимум, это потребует очень больших ресурсов от нашей операционной системы, а также само создание процесса является "дорогой" операцией. Тем не менее, такой подход иногда используется, и, если у вас небольшое количество соединений, плюсом будет то, что вы можете использовать все ядра операционной системы и распределять обработку по всем ядрам на сервере.

Если же мы будем обрабатывать новые соединения в потоках, то, как мы помним, все потоки работают в Python на одном ядре, и они ограничены GIL. Рано или поздно мы упремся в то, что нам не хватает одного ядра, и наш сервер будет отвечать медленно.

Тем не менее, и на потоках, особенно если они требуют операции ввода-вывода, можно получить достаточно производительный сервер.

Давайте рассмотрим пример одновременной обработки сетевых запросов при помощи потоков. Итак, мы создаем `socket`, вызываем нами известные методы `bind` и `listen`. Затем в бесконечном цикле принимаем входящее соединение от клиента. Как только мы приняли это входящее соединение, мы должны создать поток. Делаем это мы при помощи модуля `threading`, создаем объект класса `Thread`, передаём ему в качестве аргумента функцию и наше соединение, с которым будем дальше в этой функции работать. Запускаем поток, а в основном потоке продолжаем акцептить новые соединения:

```
# обработка нескольких соединений одновременно, потоки

import socket
import threading

def process_request(conn, addr):
    print("connected client:", addr)
    with conn:
        while True:
            data = conn.recv(1024)
            if not data:
                break
            print(data.decode("utf8"))

with socket.socket() as sock:
    sock.bind(("", 10001))
    sock.listen()
    while True:
        conn, addr = sock.accept()
        th = threading.Thread(target=process_request,
                              args=(conn, addr))
        th.start()
```

Если процесс обработки этих соединений будет заниматься вводом-выводом, то такой код будет достаточно производительным. Тем не менее, если будет недостаточно одного ядра операционной системы, можно этот процесс распараллелить. Давайте рассмотрим пример, когда можно использовать и потоки, и процессы одновременно. Итак, для того чтобы обрабатывать одно соединение в нескольких процессах, нам нужно выполнить небольшой трюк. После того как мы вызвали метод `listen`, мы должны создать несколько процессов, сделать `fork`. После того как мы сделаем вызов `fork`, все ресурсы родительского процесса будут целиком и полностью скопированы в дочерние процессы, тем самым в наших дочерних процессах будет тот же самый `socket`. Если мы в этом `socket`-е сделаем вызов `accept` и будем ждать нового соединения от клиента, то системный вызов `accept` распределит равномерно между всеми дочерними процессами новые входящие соединения, а уже дальше в этих дочерних процессах, когда мы поймали новые соединения, мы уже сможем создать поток и обработать новые соединения. Опишем эту схему

комментариями в коде:

```
# обработка нескольких соединений одновременно, процессы и потоки
import socket

with socket.socket() as sock:
    sock.bind(("", 10001))
    sock.listen()
    # создание нескольких процессов
    while True:
        # accept распределится "равномерно" между процессами
        conn, addr = sock.accept()
        # поток для обработки соединения
        with conn:
            while True:
                data = conn.recv(1024)
                if not data:
                    break
                print(data.decode("utf8"))
```

Если, опять же, мы создадим несколько процессов, которые все одновременно делают системный вызов `accept`, то по умолчанию все они будут спать, а операционная система не будет потреблять никаких ресурсов. Но если будет приходить новое входящее соединение, операционная система разбудит все наши процессы. В этом месте есть небольшой overhead, нужно это понимать.

Вот так может выглядеть код нашего сервера на процессах:

```
# обработка нескольких соединений одновременно, процессы и потоки

import socket
import threading
import multiprocessing

with socket.socket() as sock:
    sock.bind(("", 10001))
    sock.listen()

    workers_count = 3
    workers_list = [multiprocessing.Process(target=worker,
                                            args=(sock,))
                    for _ in range(workers_count)]

    for w in workers_list:
        w.start()

    for w in workers_list:
        w.join()
```

Как обычно, мы создаем socket, вызываем методы `bind` и `listen`. Затем мы должны при помощи модуля `multiprocessing` создать несколько объектов `worker`, которые будут обрабатывать новые соединения.

Давайте рассмотрим код наших `worker`-ов. Итак, каждый `worker`, который будет запущен в отдельном процессе, делает системный вызов `accept`. Все входящие соединения будут равномерно распределены между `worker`-ами при помощи операционной системы. И после того как соединение попало в наш процесс, необходимо создать поток и передать ему метод `process_request`, который обрабатывает данное соединение:

```
# обработка нескольких соединений одновременно, процессы и потоки

def worker(sock):
    while True:
        conn, addr = sock.accept()
        print("pid", os.getpid())
        th = threading.Thread(target=process_request,
                              args=(conn, addr))
        th.start()

def process_request(conn, addr):
    print("connected client:", addr)
    with conn:
        while True:
            data = conn.recv(1024)
            if not data:
                break
            print(data.decode("utf8"))
```

Таким образом, используя одновременно процессы и потоки, мы сможем решить проблему с GIL и проблему с памятью.

5.4. Асинхронное программирование

5.4.1. Исполнение кода в одном потоке, модуль select

В операционной системе существует модуль `select`, который позволяет организовать работу с неблокирующим вводом-выводом. Отдельно хочется сказать, что существует несколько механизмов опроса всех файловых дескрипторов для организации неблокирующего ввода-вывода: `select.select`, `select.poll`, `select.epoll` и `select.kqueue`. Все эти методы связаны с особенностями операционных систем. Например, в Linux, как правило, используют `epoll`, и мы будем рассматривать примеры на основе `epoll`.

Вспомним, что у нас была проблема создания сокета и обработки нескольких входящих соединений одновременно. Давайте попробуем сделать это при помощи модуля `select`. Итак, предположим, мы создали объект класс `socket`, вызвали метод `bind`, вызвали метод `listen` и смогли вызвать метод `accept` для двух соединений. Теперь у нас есть два объекта соединения — `conn1` и `conn2`, и нам необходимо одновременно читать или записывать данные из этих соединений без использования потоков или процессов. Для этого необходимо перевести наше соединение, во-первых, в неблокирующий режим при помощи вызова `setblocking(0)` (равносильно тому, что мы сделаем вызов `settimeout(0)`).

Теперь наши сокеты в неблокирующем режиме, и если мы попробуем что-то прочитать из этого сокета, а там данных нет, то наш вызов `recv` не заблокируется, а вернет некую системную ошибку. Но как узнать, какие сокеты готовы читать, а какие сокеты готовы записывать данные? Для этого как раз нам понадобится объект `epoll`. Мы регистрируем в

объекте `epoll` наши файловые дескрипторы от созданных коннектов, а также говорим, на какие события подписываемся от этих файловых дескрипторов. В данном случае это чтение из сокетов и запись в сокет (`select.EPOLLIN | select.EPOLLOUT`). Далее для организации цикла опроса событий нам необходимо запомнить наши объекты и смапить их по файловым дескрипторам. То есть формируем словарь `conn_map`, в него записываем файловые дескрипторы и объекты наших соединений:

```
# Неблокирующий ввод/вывод, обучающий пример

import socket
import select

sock = socket.socket()
sock.bind(("", 10001))
sock.listen()

# как обработать запросы для conn1 и conn2
# одновременно без потоков?
conn1, addr = sock.accept()
conn2, addr = sock.accept()

conn1.setblocking(0)
conn2.setblocking(0)

epoll = select.epoll()
epoll.register(conn1.fileno(), select.EPOLLIN | select.EPOLLOUT)
epoll.register(conn2.fileno(), select.EPOLLIN | select.EPOLLOUT)

conn_map = {
    conn1.fileno(): conn1,
    conn2.fileno(): conn2
}
```

Давайте рассмотрим цикл обработки событий для нашего `epoll`, который иногда называют event loop. Итак, мы в бесконечном цикле постоянно опрашиваем наш объект `epoll`. Это системный вызов, который нам возвращает список событий, и эти события содержат файловый дескриптор и непосредственно то событие, которое произошло с этим файловым дескриптором. После того как `epoll` вернул этот список, мы должны проверить, что за событие пришло, из нашего словаря получить нужный объект для работы с ним и выполнить определённые операции. В данном случае, например, мы читаем данные из этого сокета и выводим их в консоль. Если пришло событие, которое говорит нам, что сокет готов принять данные от нас, мы должны записать данные в этот сокет:

```
# Неблокирующий ввод/вывод, обучающий пример
# Цикл обработки событий в epoll

while True:
    events = epoll.poll(1)

    for fileno, event in events:
        if event & select.EPOLLIN:
            # обработка чтения из сокета
            data=conn_map[fileno].recv(1024)
            print(data.decode("utf8"))
        elif event & select.EPOLLOUT:
            # обработка записи в сокет
            conn_map[fileno].send("pong".encode("utf8"))
```

Такой код иногда называют асинхронным программированием, или мультиплексированием ввода/вывода. Код уже не выглядит слишком простым (хотя в нем нет создания потоков или процессов, нет обработки закрытия сокетов, отсутствует обработка новых входящих соединений). Также мы не тратим память на создание процессов, нет расходов на создание потоков и их синхронизацию, нет проблем с GIL.

Но если код будет решать настоящие задачи, то увеличится кол-во операторов if или callback-ов. Кроме того, нам придётся изменять код, если в обработке запроса появятся вызовы сторонних библиотек. Для этого есть решение: спрятать вызовы `select.epoll` в функции библиотеки.

Так поступили и написали существующие в Python фреймворки. Наиболее популярным был долгое время фреймворк Twisted, его работа очень похожа на наш пример, но код на Twisted, затем Gevent и Tornado. После Tornado в Python3 появился фреймворк asyncio, и он сейчас является мейнстримом. В его основе, в принципе, лежит то же самое, что и в Tornado, но тем не менее asyncio поставляется вместе с Python3 Core и также основан на работе генераторов. Его поддерживают официальные сообщества, и поэтому все наши дальнейшие разговоры о программировании в один поток будут привязаны к фреймворку asyncio.

5.4.2. Итераторы и генераторы, в чём разница

Чтобы лучше понимать, как устроен asyncio, нужно сначала разобраться в том, как устроены генераторы и итераторы, в чём их сходство и различие.

Давайте рассмотрим пример с итератором, который генерирует последовательность 0, 1, 2. Чтобы решить эту задачу, необходимо создать класс и переопределить у него методы `__init__`, `__iter__` и `__next__`. Если мы вызовем итератор в цикле `for`, то сначала вызовется метод `__iter__`, а затем будет последовательно вызываться метод `__next__`. Когда выполнится условие `if self.current >= self.top` вызовется исключение, и итератор прекратит работу:

```
# Итераторы

class MyRangeIterator:
    def __init__(self, top):
        self.top = top
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.top:
            raise StopIteration

        current = self.current
        self.current += 1
        return current

counter = MyRangeIterator(3)
counter
```

```
<__main__.MyRangeIterator object at 0xb671b5cc>
```

```
for it in counter:
    print(it)
```

```
0
1
2
```

Чтобы создать генератор, нужно написать обычную функцию с командой `yield`. Реализуем тот же фукционал с помощью генератора. На самом деле, когда мы вызовем эту функцию, создастся объект, по которому можно итерироваться:

```
# Генераторы

def my_range_generator(top):
    current = 0
    while current < top:
        yield current
        current += 1

counter = my_range_generator(3)
counter
```

```
<generator object my_range_generator at 0xb67170ec>
```

```
for it in counter:
    print(it)
```

```
0
1
2
```

На самом деле, при вызове `yield` генератор замораживает свой стек и все значения локальных переменных в объекте `counter`. При повторном вызове функций стек и локальные переменные восстанавливаются, и мы продолжаем выполнять следующие инструкции.

Подводя итоги, скажем, что в генераторах заложены большие возможности для написания concurrency кода.

5.4.3. Генераторы и сопрограммы

В этом разделе мы познакомимся с сопрограммами (или корутинами) и выясним, в чем отличие и сходство между генераторами и сопрограммами в Python. Предположим, наша цель — фильтровать входной поток при помощи функции `grep`. В функцию `grep` мы передаем строку-паттерн, далее мы должны в эту функцию перевадать строчки и выводить на экран только те из них, в которых присутствует заданный паттерн. Давайте реализуем эту функцию как корутину.

Мы в бесконечном цикле вызываем такую конструкцию `line` присвоить `yield` (в отличие от генераторов в корутинах мы переменной присваиваем результат работы конструкции `yield`). В данном случае функция заморозит своё состояние и будет ожидать ввода данных при помощи метода `send`. Итак, если мы вызовем функцию `grep`, будет создана корутина. Для того чтобы запустить нашу корутину, необходимо вызвать метод `next`. После того, как метод `next` будет вызван, запустится код нашей функции, выведется строчка `start grep`, запустится бесконечный цикл, код дойдет до инструкции `yield`, и здесь управление вернется в основной поток. После этого в основном потоке мы отправляем данные нашей корутине, и код функции выполняется дальше:

```
# Сопрограммы (корутины)

def grep(pattern):
    print("start grep")
    while True:
        line = yield
        if pattern in line:
            print(line)

g = grep("python")
next(g) # g.send(None)
```

```
start grep
```

```
g.send("golang is better?")
g.send("python is simple!")
```

```
python is simple!
```

Таким образом, можно сделать вывод, что генераторы производят значения, а корутины их потребляют с помощью одного и того же метода `yield`.

Иногда необходимо остановить запущенную корутину. Делается это при помощи вызова метода `close` для объекта корутины. Метод `close` будет вызван автоматически сборщиком мусора, но если нам нужно самим остановить корутину, то можно руками вызвать метод `close`. Метод `close` сгенерирует исключение генератора в том месте, где функция заморозила свое значение. Это исключение нельзя игнорировать, его нужно обрабатывать — например, при помощи блока `try except`:

```
# Сопрограммы, вызов метода close()

def grep(pattern):
    print("start grep")
    try:
        while True:
            line = yield
            if pattern in line:
                print(line)
    except GeneratorExit:
        print("stop grep")

g = grep("python")
next(g)  # g.send(None)
```

```
start grep
```

```
g.send("python is the best!")
```

```
python is the best!
```

```
g.close()
```

```
stop grep
```

Иногда необходимо передать исключения в саму корутину. Это делается при помощи вызова метода `throw`. Пример точно такой же:

```
# Сопрограммы, генерация исключений

def grep(pattern):
    print("start grep")
    try:
        while True:
            line = yield
            if pattern in line:
                print(line)
    except GeneratorExit:
        print("stop grep")

g = grep("python")
next(g) # g.send(None)
g.send("python is the best!")
g.throw(RuntimeError, "something wrong")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: something wrong
```

Давайте посмотрим еще более сложный пример. У нас есть корутина `grep`, содержащая внутри инструкцию `yield`, и мы хотим вызвать корутину `grep` в другой корутине. Как это сделать? Если мы напишем код, который приведен на слайде, и потом сделаем данный вызов, то будет ли переменная `g` являться корутиной? Так как она не содержит инструкции `yield`, она выполнится сразу, т.к. будет являться обычной функцией:

```
# Вызовы сопрограмм, PEP 380

def grep(pattern):
    print("start grep")
    while True:
        line = yield
        if pattern in line:
            print(line)

def grep_python_coroutine():
    g = grep("python")
    next(g)
    g.send("python is the best!")
    g.close()

g = grep_python_coroutine()  # is g coroutine?
```

```
start grep
python is the best!
```

Для того чтобы было удобно вызывать из одних корутин другие, в Python разработали стандарт PEP 0380 и в Python 3 его реализовали. В новом стандарте появилась инструкция `yield from`. При помощи неё можно выполнить делегирование вызова одной корутины в другой. Наш пример можно переписать следующим образом мы используем инструкцию `yield from` и указываем объект в виде другой корутины:

```
# Сопрограммы, yield from PEP 0380

def grep(pattern):
    print("start grep")
    while True:
        line = yield
        if pattern in line:
            print(line)

def grep_python_coroutine():
    g = grep("python")
    yield from g

g = grep_python_coroutine()  # is g coroutine?
g
```

```
<generator object grep_python_coroutine at 0x7f027eec03b8>
```

```
g.send(None)
```

```
start grep
```

```
g.send("python wow!")
```

```
python wow!
```

Теперь, если мы попробуем вызвать нашу функцию `grep_python_coroutine`, она будет являться корутиной или генератором, и для того, чтобы продолжить и выполнить код, который находится у нее внутри, необходимо вызвать метод `send`, передать туда значение None и далее вызвать метод `send` и передать нужную строчку. Таким образом, в Python 3 стало очень легко и удобно вызывать из одной корутины другую.

Для обычных генераторов инструкцию `yield from` можно использовать как замену цикла `for`, внутри которого вызывается инструкция `yield`. Рассмотрим пример. Например, у нас есть два объекта, по которым возможно осуществлять итерацию — `a` и `b`. Далее в цикле вызываем функцию `chain`, передаем туда эти списки и выводим то, что нам генерирует наша функция `chain`. В функции `chain` мы используем конструкцию `yield from` и передаем туда объект, по которому возможна итерация. Функция `the_same_chain` работает так же, но реализована при помощи циклов:

```
# PEP 380, генераторы

def chain(x_iterable, y_iterable):
    yield from x_iterable
    yield from y_iterable

def the_same_chain(x_iterable, y_iterable):
    for x in x_iterable:
        yield x

    for y in y_iterable:
        yield y

a = [1, 2, 3]
b = (4, 5)
for x in chain(a, b):
    print(x)
```

```
1
2
3
4
5
```

5.4.4. Первые шаги с asyncio

`asyncio` — это библиотека, которая стала частью Python 3. Она отвечает за неблокирующий ввод/вывод, на этом фреймворке можно написать сервис, который работает с десятками тысяч соединений одновременно. В основе работы этого фреймворка лежат генераторы и корутины, о чем мы уже говорили в предыдущих видео.

В следующем примере мы объявляем функцию и добавляем к этой функции декоратор `asyncio.coroutine`, тем самым делая нашу функцию корутиной. Далее, мы в бесконечном цикле выполняем вывод строчки `"Hello World"` в консоль и делаем вызов `yield from asyncio.sleep(1.0)`, то есть "засыпаем" на одну секунду. Обратите внимание, что мы используем не привычный нам `time.sleep` вызов, а специальную конструкцию `yield from` для того, чтобы наша корутина приостановила свою работу, тем самым давая возможность поисполняться другим корутинам:

```
# asyncio, Hello World

import asyncio

@asyncio.coroutine
def hello_world():
    while True:
        print("Hello World!")
        yield from asyncio.sleep(1.0)
```

Весь код в asyncio строится на основе понятия цикла обработки событий или, как еще его иногда называют, event loop. event loop — это своего рода планировщик задач или корутин, которые в нем исполняются. Он отвечает за ввод/вывод, управление сигналами, всеми сетевыми операциями и переключает контекст между всеми корутинами, которые в нем зарегистрированы и выполняются. Если одна корутина ожидает завершения какой-то сетевой операции, то в этот момент event loop может переключиться на другую корутину и продолжить ее выполнение. Продолжим наш пример. При помощи вызова `asyncio.get_event_loop` мы получаем цикл обработки событий. Это объект, который исполняет корутины (обычные функции с помощью него исполнять нельзя, нужно использовать именно корутины):

```
loop = asyncio.get_event_loop()
loop.run_until_complete(hello_world())
```

```
Hello World!
Hello World!
...
```

Для того, чтобы завершить работу с циклом обработки событий, необходимо вызвать метод `close` для нашего объекта `loop`:

```
loop.close()
```

Начиная с версии Python 3.5, появился новый PEP 492, в котором был введен специальный синтаксис для написания корутин — с инструкциями `async def` и `await`. Этот синтаксис выглядит более лаконично и красиво по сравнению с предыдущим, кроме того, объявление функции через конструкцию `async def` гарантирует нам, что эта функция является корутиной. Если мы используем этот синтаксис, то внутри мы не можем использовать конструкцию `yield from`, мы обязаны использовать вызов `await`:

```
# asyncio, async def / await; PEP 492 Python3.5

import asyncio

async def hello_world():
    while True:
        print("Hello World!")
        await asyncio.sleep(1.0)

loop = asyncio.get_event_loop()
loop.run_until_complete(hello_world())
loop.close()
```

Давайте рассмотрим более сложный пример и напишем свой TCP-сервер, который обрабатывает несколько входящих соединений одновременно. Итак, мы получаем наш event loop, делаем вызов `start_server`, передаём в этот вызов корутину. В функциию `start_server` мы должны еще передать параметры в виде хоста и порта, на котором мы будем слушать соединение. Далее мы запускаем установку этого соединения и делаем вызов `loop.run_forever`. Тем самым мы будем обрабатывать все входящие соединения, и после того, как мы заакцептили соединение, для каждого соединения будет создана отдельная корутина, и в этой корутине будет выполнена наша функция. При помощи конструкции `await reader` мы можем читать данные из нашего сокета, и также существует `writer` (если нам будет необходимо, мы сможем записывать данные в наш сокет):

```
# asyncio, tcp сервер

import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(1024)
    message = data.decode()
    addr = writer.get_extra_info("peername")
    print("received %r from %r" % (message, addr))
    writer.close()

loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo, "127.0.0.1", 10001,
                            loop=loop)
server = loop.run_until_complete(coro)
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Давайте рассмотрим код клиента, который тоже может быть асинхронным. Допустим, мы клиент будет передавать строчку. В корутину `tcp_echo_client` передаём эту строчку (`message`) и наш event loop. Далее, для того, чтобы создать соединение, мы должны вызвать метод `asyncio.open_connection`. В этом вызове мы должны отправить адресную пару и вызов `await` вернет нам `reader` и `writer`. Это два объекта, при помощи которых можно взаимодействовать с нашим удаленным сервером. То есть, при помощи объекта `reader` можно читать данные с сервера, при помощи объекта `writer` можно записывать данные на сервер:

```
# asyncio, tcp клиент

import asyncio

async def tcp_echo_client(message, loop):
    reader, writer = await asyncio.open_connection("127.0.0.1",
                                                   10001, loop=loop)

    print("send: %r" % message)
    writer.write(message.encode())
    writer.close()

loop = asyncio.get_event_loop()
message = "hello World!"
loop.run_until_complete(tcp_echo_client(message, loop))
loop.close()
```

Вы можете легко создать несколько таких асинхронных клиентов и одновременно выполнять запросы на разные сервера, при этом не делая никаких потоков или процессов. Это очень удобно, просто, и код получается достаточно производительным.

5.4.5. Работа с asyncio

В этом последнем разделе мы обсудим, что такое `asyncio.Future` и поговорим о том, как создавать объекты типа `asyncio.Task`. Также мы рассмотрим проблему запуска синхронных функций в цикле обработки событий и немного обсудим библиотеки, которые существуют для работы с фреймворком `asyncio`.

Давайте перейдем к примеру. В нём мы объявляем корутину `slow_operation`, в неё передаем некий объект `asyncio.Future()`, который исполняется, и его выполнение ещё не завершено. Интерфейс этого объекта целиком и полностью соответствует интерфейсу `concurrent.futures.Future` (мы разбирали его при знакомстве с потоками).

В корутине мы выполнили `sleep` на одну секунду, и после этого при помощи `set_result` выставили результат в наш объект типа `future`:

```
### asyncio.Future, аналог concurrent.futures.Future

import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result("Future is done!")

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))

loop.run_until_complete(future)
print(future.result())
loop.close()
```

```
Future is done!
```

Обратите внимание — в основной программе мы создаем объект `future`, далее мы создаем нашу корутину при помощи `ensure_future`, а в основном цикле обработки событий мы ожидаем завершения объекта `future` (а не функции `slow_operation`!). Таким образом, при помощи объектов класса `future` можно выстраивать цепочки не только из двух объектов, но и более сложные цепочки, и очень удобно дожидаться завершения выполнения всех объектов.

Давайте посмотрим, как можно запустить несколько корутин в одном event loop. Для этого, как правило, используется объект типа `asyncio.Task`, который является наследником класса `asyncio.Future` с рядом дополнительных методов. Итак, напрямую объект типа `asyncio.Task` создавать не нужно, стоит использовать метод `create_task` и передавать в него корутину (у каждого объекта типа `Task` есть собственная корутина, которую он исполняет). Итак, мы создаем список из двух тасков. Запоминаем его в виде списка объектов, и далее при помощи метода `asyncio.wait` мы исполняем список наших тасков в event loop:

```
### asyncio.Task, запуск нескольких корутин

import asyncio

async def sleep_task(num):
    for i in range(5):
        print(f"process task: {num} iter: {i}")
        await asyncio.sleep(1)

    return num

# ensure_future or create_task
loop = asyncio.get_event_loop()

task_list = [loop.create_task(sleep_task(i)) for i in range(2)]
loop.run_until_complete(asyncio.wait(task_list))

loop.close()
```

```
process task: 0 iter: 0
process task: 1 iter: 0
process task: 0 iter: 1
process task: 1 iter: 1
process task: 0 iter: 2
process task: 1 iter: 2
process task: 0 iter: 3
process task: 1 iter: 3
process task: 0 iter: 4
process task: 1 iter: 4
```

Мы видим, что у нас сначала один таск выполняет нулевую итерацию, затем второй таск с номером выполняет свою нулевую итерацию и т.д. То есть корутины исполняются в event loop-е одновременно, а код при этом последовательный.

Также можно выполнить несколько тасков при промощи удобной функции `asyncio.gather`, которая позволяет не вызывать отдельно метод `create_task`:

```
loop.run_until_complete(asyncio.gather(sleep_task(10),
                                       sleep_task(20)))
```

Но что если нам необходимо исполнить синхронную функцию в event loop? Как правило, таких задач не должно возникать, но если вдруг они и возникли, то они будут представлять из себя небольшую сложность. event loop постоянно переключает контекст между всеми нашими корутинами и исполняет их последовательно — пока одна корутина ожидает ввода-вывода, вторую корутину event loop благополучно исполняет. Если код, который будет исполняться в корутине, будет блокирующим, то event loop не сможет делать переключения контекста. Для решения этой проблемы в `asyncio` существует метод

run_in_executor. Он означает запустить код буквально в пуле потоков, который внутри этого event loop-а автоматически будет создан (можно использовать собственный пул потоков или уже готовый по умолчанию).

На этом примере можно наблюдать, как функция urlopen, которая открывает некий url, который ей передали, скачивает результаты в отдельном потоке. Для этого мы вызываем метод run_in_executor. Внутри него будет создано нужное количество потоков, и наша функция sync_get_url с параметром url будет выполнена в отдельном потоке. Для того, чтобы дождаться результата выполнения функции в отдельном потоке, мы используем конструкцию await и передаем туда объект future. Мы будем открывать страничку google.com и выводить на экран количество байт, которые она занимает:

```
# loop.run_in_executor, запуск в отдельном потоке

import asyncio
from urllib.request import urlopen

# a synchronous function
def sync_get_url(url):
   return urlopen(url).read()

async def load_url(url, loop=None):
    future = loop.run_in_executor(None, sync_get_url, url)
    response = await future
    print(len(response))

loop = asyncio.get_event_loop()
loop.run_until_complete(load_url("https://google.com", loop=loop))
```

```
11055
```

Как правило, такие задачи будут возникать у вас, если в некоторой библиотеке не будет поддержки работы с asyncio. Это может сказаться негативно на производительности, потому что, как мы помним, потоки в Python запускаются и работают с ограничением GIL. В последнее время появляются библиотеки с поддержкой asyncio — aiohttp, aiomysql, aiomcache и многие другие.