кафедра Алгоритмов
и Технологий
Программирования

Многопоточное программирование с python

Емельянов А. А.
login-const@mail.ru

2017

https://github.com/king-menin/python-course

Что такое процесс?

- Процесс — программа, которая выполняется в текущий момент. Стандарт ISO 9000:2000 определяет процесс как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

- Компьютерная программа сама по себе — это только пассивная последовательность инструкций, в то время как процесс — это непосредственное выполнение этих инструкций.

Что такое процесс

- Также, процессом называют выполняющуюся программу и все её элементы:
  - Идентификатор процесса (PID),
  - Адресное пространство,
  - Глобальные переменные,
  - Регистры,
  - Стек,
  - Открытые файлы,
  - Ввод/вывод…

Как создавать процессы 1

```
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()
```

```
os.system("dir")
```

```
0
```

Создание потока в python: subprocess

- The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions.
- Класс Popen – весь необходимый функционал для работы с под-процессами.

```
class Popen(object):
    def __init__(self, args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None,
                 preexec_fn=None, close_fds=True, shell=False, cwd=None,
                 env=None, universal_newlines=False, startupinfo=None, creationflags=0,
                 restore_signals=True, start_new_session=False, pass_fds=(),
                 *, encoding=None, errors=None)
```

- То, что действительно используют:

```
class Popen(object):
    def __init__(self, args, *, stdin=None, stdout=None, stderr=None,
                 cwd=None, env=None)
    ...
```

subprocess.Popen

- Пример вызова

```
import subprocess as sp
```

Popen

```
p = sp.Popen(["git", "help", "-a"])
p
```

```
<subprocess.Popen at 0xe8bad33080>
```

subprocess.Popen

- Получение результатов подпроцесса.
- Потоки ввода (stdin), вывода (stdout) и ошибок (stderr).

```
p.stdout
```

```
p = sp.Popen(["git", "help", "-a"], stdout=sp.PIPE)
p.wait()
list(p.stdout)[:3]
```

```
[b'usage: git [--version] [--help] [-C <path>] [-c name=value]\n',
 b'           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]\n',
 b'           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]\n']
```

- subprocess.PIPE – специальная переменная для извлечения данных из потоков.

subprocess.run

- run – это обертка надо Popen, возвращает специальный объект subprocess.CompletedProcess

```
# Файл sum.cpp
"""
#include <iostream>

int Sum(int first, int second) {
    return first + second;
}

int main() {
    int first = 0;
    int second = 0;
    std::cin >> first >> second;
    std::cout << Sum(first, second) << std::endl;
}
"""

sp.run(['g++', 'sum.cpp', '-o', 'sum.out'], stderr=sp.PIPE)
```

```
CompletedProcess(args=['g++', 'sum.cpp', '-o', 'sum.out'], returncode=0, stderr=b'')
```

```
sp.run(['./sum.out'], stdout=sp.PIPE, input=b'2 3')
```

```
CompletedProcess(args=['./sum.out'], returncode=0, stdout=b'5\r\n')
```

Поток ошибок в subprocess. Пример.

```
# Файл bad_sum.cpp
"""
// #include <iostream>

int Sum(int first, int second) {
    return first + second;
}

int main() {
    int first = 0;
    int second = 0;
    std::cin >> first >> second;
    std::cout << Sum(first, second) << std::endl;
}
"""

sp.run(['g++', 'bad_sum.cpp', '-o', 'sum.out'], stderr=sp.PIPE)
```

```
CompletedProcess(args=['g++', 'bad_sum.cpp', '-o', 'sum.out'], returncode=1, stderr=b"b
\nbad_sum.cpp:10:5: error: 'cin' is not a member of 'std'\r\n     std::cin >> a >> b;\r
ut' is not a member of 'std'\r\n     std::cout << Sum(a, b) << '\\n';\r\n     ^\r\n")
```

Модуль shlex

- С помощью функции split можно легко задавать аргументы.

```
import shlex
sp.run(shlex.split('g++ sum.cpp -o sum.out'))
```

```
CompletedProcess(args=['g++', 'sum.cpp', '-o', 'sum.out'], returncode=0)
```

Пример: обработка запросов на вики

```
class WikiReader(object):
    ...
    def run_async_sub_proc(self):
        def communicate(sub_process):
            out, _ = sub_process.communicate()
            return out.decode("utf-8").split("\n")

        sub_processes = [
            sp.Popen(["python", "./reader/search_text.py", self.__tmp, query],
                     stdout=sp.PIPE, stderr=sp.PIPE)
            for query in self.__queries]
        res = []
        for sub_process, query in zip(sub_processes, self.__queries):
            res.append(communicate(sub_process))
        if not all(map(lambda x: x[0][0].strip() == x[1], zip(res, self.__queries))):
            print("Failed on results order")
        return dict(zip(self.__queries, list(map(lambda x: x[1], res))))
    ...
```

```
%%timeit
_ = wr.run_async_sub_proc()
```

```
1 loop, best of 3: 4.62 s per loop
```

VS

```
def run_synchronously(self):
    return dict(zip(self.__queries, map(self.search_text,
                                        zip([self.__tmp] * len(self.__queries), self.__queries))))
```

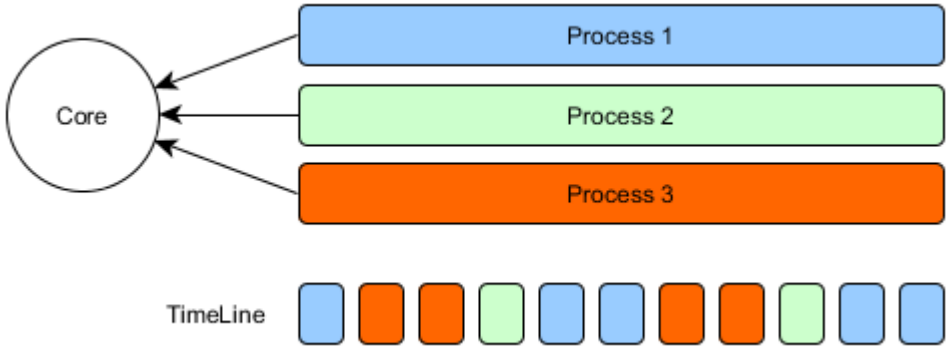
```
%%timeit
_ = wr.run_synchronously()
```

```
1 loop, best of 3: 4.9 s per loop
```

Как происходит работа процессора

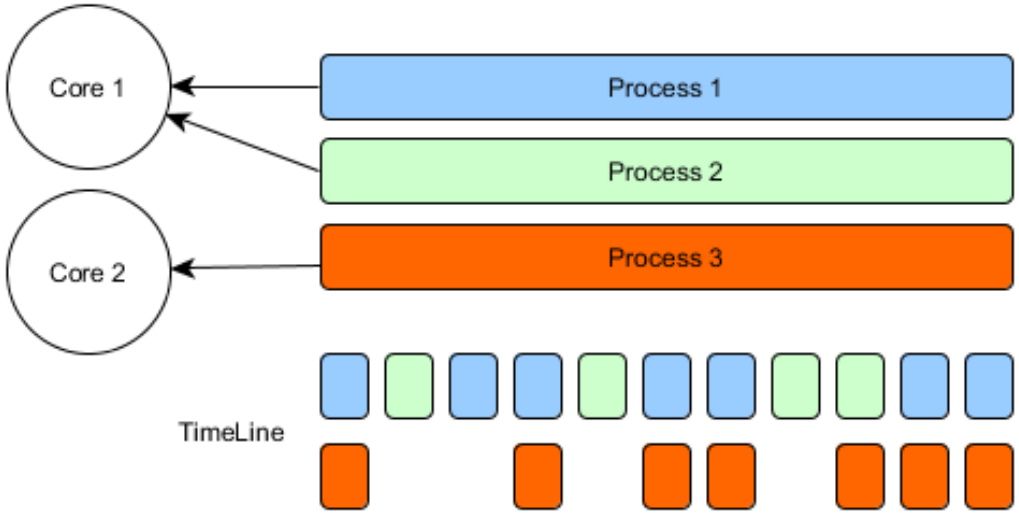
- Поочередное выполнение частей каждого процесса, которые сейчас работают.

- Физически на одном ядре может выполняться только один процесс.



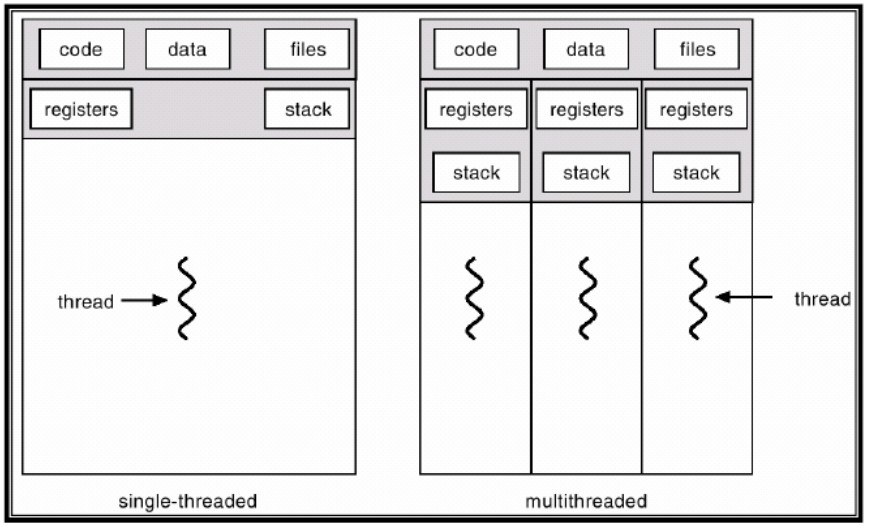
Как происходит работа процессора

- В случае нескольких ядер, параллельно (в физическом смысле) могут работать несколько процессов.



Поток

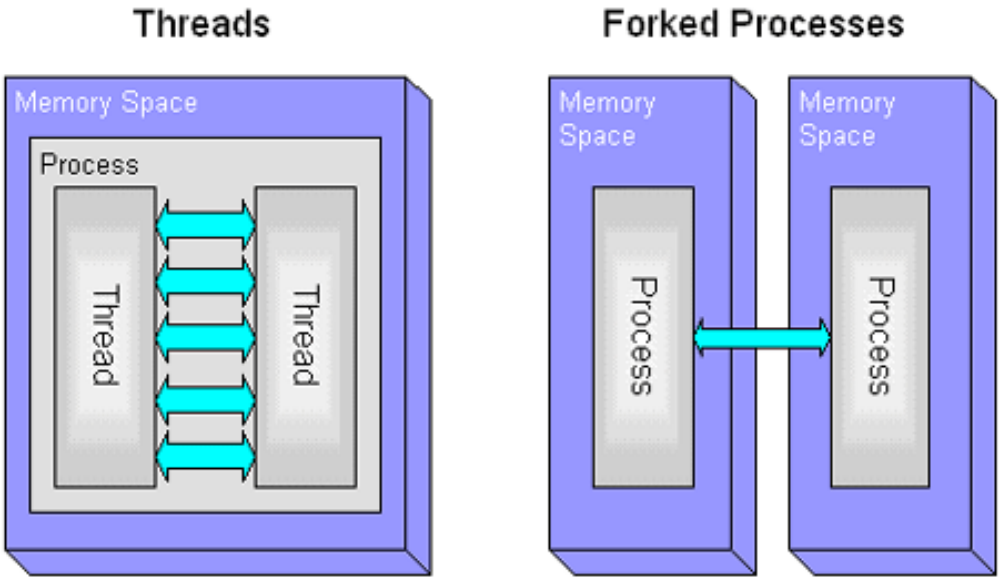
- Поток – это тоже программа, однако в рамках выполнения одного процесса их может быть несколько (несколько потоков выполнения).
- Поток имеет свой стек и адресное пространство (на физическом уровне):
  - регистры и т. д.
- Поток является зависимой сущностью от процесса, однако существенно влияет на другие потоки в рамках одного процесса
- Потоки, созданные в рамках одного процесса имеют следующие общие элементы:
  - код,
  - данные,
  - файлы



1. Картинка - https://sites.google.com/site/sureshdevang/thread-vs-process

Потоки vs процессы

- Процессы почти ничего не знают про другие процессы.
- Могут управлять только дочерними процессами.
- Являются независимыми.



1. Картинка - https://www.perl.com/pub/2002/09/04/threads.html

Создание потока в python: threading

- start – запускает поток.

- join – ждет окончания выполнения потока (в месте вызова инструкции).

```
import threading
```

```
def thread_job(number):
    # time.sleep(np.random.rand())
    print(" Hi! I am {} ".format(number), end = " | ")


def run_threads(count):
    threads = [
        threading.Thread(target=thread_job, args=(i,))
        for i in range(count)
    ]
    for thread in threads:
        thread.start()
    # ALL NON-DAEMON THREADS MUST BE JOINED
    for thread in threads:
        thread.join()

run_threads(4)
```

```
 Hi! I am 0  Hi! I am 1  Hi! I am 2  Hi! I am 3  |  |  |  |
```

Проблемы с потоками: общая память и прерывания

```
import random
import time

counter = 0
def thread_job():
    global counter
    old_counter = counter
    time.sleep(random.randint(0, 1))
    counter = old_counter + 1
    print('{} '.format(counter), end='')

threads = [threading.Thread(target=thread_job) for _ in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
counter
```

```
1 2 3 4 5 2 5 5 6 5

5
```

Блокировки

- Решение проблемы доступа к общей памяти могут являться блокировки.

```
counter = 0
def thread_job(lock):
    time.sleep(random.randint(0, 1))
    lock.acquire()
    global counter
    counter += 1
    print('{} '.format(counter), end='')
    lock.release()

lock = threading.Lock()
threads = [
    threading.Thread(target=thread_job, args=(lock,))
    for i in range(10)
]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

counter
```

```
1 2 3 4 5 6 7 9 8 10

10
```

Рекурсивная блокировка

- Позволяет вызывать блокировку «внутри блокировки».
- Если вдруг используете блокировки, то только RLock[1].

```
class ColoredPoint(Point):
    def __init__(self):
        super(ColoredPoint, self).__init__()
        self._color = 'green'

    @property
    def color(self):
        with self._mutex:
            return self._color

    @color.setter
    def color(self, val):
        with self._mutex:
            self._color = val

    def do(self, observer):
        with self._mutex:
            if self._color == 'red':
                observer(self.get())
```

1. http://asvetlov.blogspot.ru/2010/11/2.html

Global Interpreter Lock (GIL)

- Прежде всего GIL — это блокировка, которая обязательно должна быть взята перед любым обращением к Питону (а это не только исполнение питоновского кода а еще и вызовы Python C API). Строго говоря, единственные вызовы, доступные после запуска интерпретатора при незахваченном GIL — это его захват.

- Нарушение правила ведет к мгновенному аварийному завершению (лучший вариант) или отложенному краху программы (куда более худший и труднее отлаживаемый сценарий).

Как работает GIL (начиная с python 3.2)

- Захват GIL зеркально отражает его освобождение. Сначала ждем, пока GIL не освободится. Если ждем долго (больше 5 мс по умолчанию) и при этом не произошло переключения (не важно, на нас или какой другой поток) — выставляем запрос на переключение.
- Дождавшись наконец свободного GIL, захватываем его и сигналим отдавшему потоку что передача состоялась. Естественно, все обращения защищены блокировками.

- Что получилось в итоге[1]:
  - поток, владеющий GIL, не отдает его пока об этом не попросят.
  - если уж отдал по просьбе, то подождет окончания переключения и не будет сразу же пытаться захватить GIL назад.
  - поток, у которого сразу не получилось захватить GIL, сначала выждет 5 мс и лишь потом пошлет запрос на переключение, принуждая текущего владельца освободить ценный ресурс. Таким образом переключение осуществляется не чаще чем раз в 5 мс, если только владелец не отдаст GIL добровольно перед выполнением системного вызова.

1. http://asvetlov.blogspot.ru/2011/07/gil.html

Какие бывают потоки

- Потоки в питоне делятся на два типа:
  - «Обычные» потоки – их необходимо завершать самостоятельно, иначе основной процесс не сможет закончить работу.
  - Демоны (daemon) – потоки, которые работают бесконечно и завершаются автоматически при завершении процесса ("не считаются").

Очереди задач для потоков: Queue

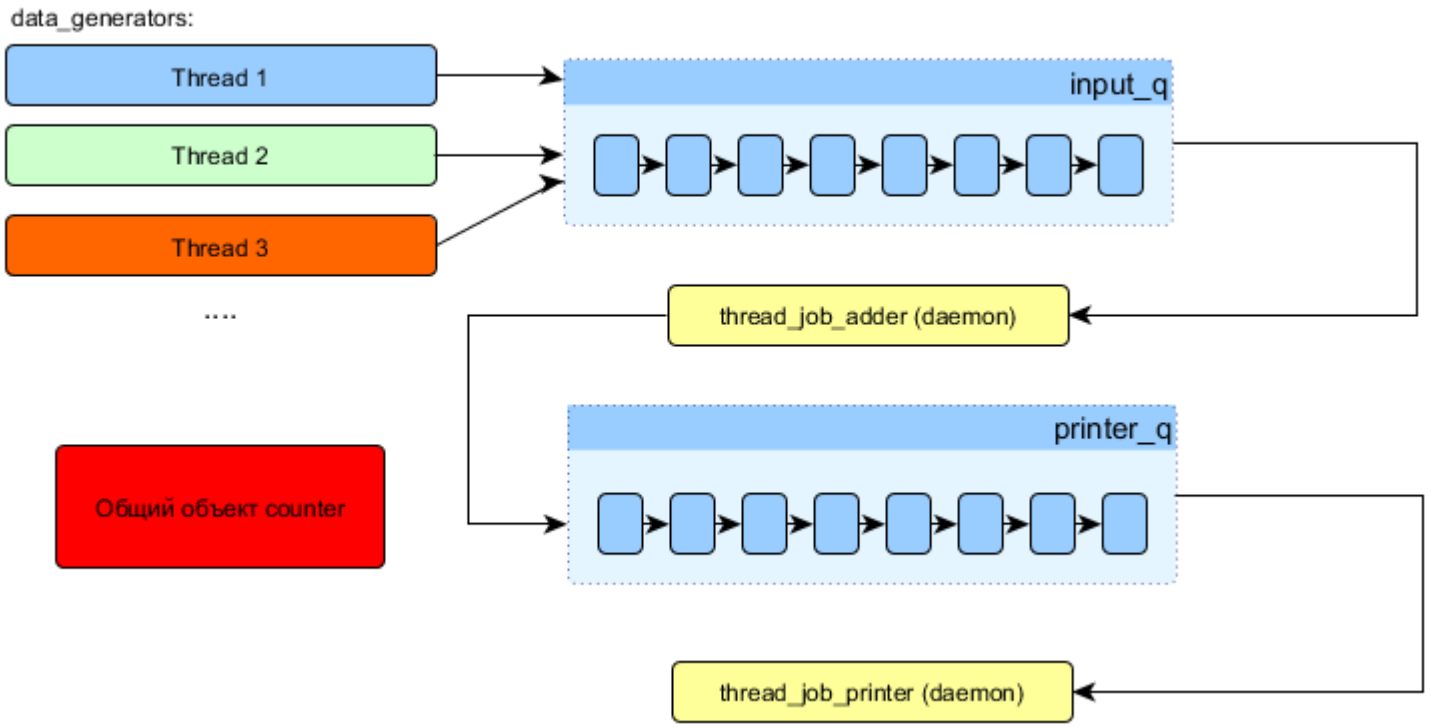
```
from queue import Queue
import threading
class Counter(object):
    def __init__(self, value=0):
        self.value = value
def thread_job_adder(counter, input_q, printer_q):
    while True:
        counter.value += input_q.get()
        input_q.task_done()
        printer_q.put(counter.value)
def thread_job_printer(printer_q):
    while True:
        print(printer_q.get(), end=' ')
        printer_q.task_done()
def data_generator(input_q):
    input_q.put(1)
def get_counted(size=10):
    counter = Counter()
    input_q = Queue()
    printer_q = Queue()
    adder_daemon = threading.Thread(target=thread_job_adder,
                                    args=(counter, input_q, printer_q), daemon=True)
    adder_daemon.start()

    printer_daemon = threading.Thread(target=thread_job_printer,
                                      args=(printer_q, ), daemon=True)
    printer_daemon.start()

    threads = [threading.Thread(target=data_generator, args=(input_q, ))
                    for query in range(size)]
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
    input_q.join()
    printer_q.join()
```

Очереди задач для потоков: Queue

- Результат предыдущего примера:

```
: get_counted()
1 2 3 4 5 6 7 8 9 10
```

- Схема работы:



Пример: сумма элементов массива

```
size = 10 * 1000 * 1000
arr = [1 for _ in range(size)]
process_count = 8
part_size = size // process_count
data = [arr[i * part_size: (i + 1) * part_size] for i in range(process_count)]
```

```
import queue

def thread_job(arr, results_queue):
    results_queue.put(sum(arr))

def sum_using_threads(data):
    results_queue = queue.Queue()
    threads = [
        threading.Thread(target=thread_job, args=(batch, results_queue))
        for batch in data]
    for thread in threads:
        thread.start()

    results = []
    for thread in threads:
        results.append(results_queue.get())
        thread.join()

    return sum(results)
```

```
%%timeit
sum(arr)
```

```
10 loops, best of 3: 110 ms per loop
```

```
%%timeit
sum_using_threads(data)
```

```
10 loops, best of 3: 114 ms per loop
```

Пример: обработка запросов на вики

```
class WikiReader(object):
    ...
    def run_async_thread(self):
        res = list()

        def thread_job(tmp, query):
            res.append((query, self.search_text((tmp, query))))

        threads = [threading.Thread(target=thread_job, args=(self.__tmp, query,))
                   for query in self.__queries]

        for thread in threads:
            thread.start()

        for thread in threads:
            thread.join()

        if not all(map(lambda x: x[0][0].strip() == x[1], zip(res, self.__queries))):
            print("Failed on results order")

        return dict(zip(self.__queries, list(map(lambda x: x[1], res))))
    ...
```

```
%%timeit
_ = wr.run_async_thread()
```

```
Failed on results order
Failed on results order
Failed on results order
Failed on results order
1 loop, best of 3: 2.69 s per loop
```

Использование потоков, вывод:

- Использование потоков полезно в IO bound задачах (в коде есть блокирующие операции, например запрос к сайту или бд).

- Использование потоков бесполезно в cpu bound задачах (сложные мат. вычисления).

Потоки: бонус concurrent.futures

- Хорошая библиотека для работы с потоками.

- Объект future – поток, который выполнился, или выполняется.

```
# since Python 3.2
from concurrent.futures import ThreadPoolExecutor
%%timeit
executor = ThreadPoolExecutor(max_workers=8)
list(executor.map(sum_using_threads, data))
```

```
%%timeit
from concurrent.futures import ThreadPoolExecutor

executor = ThreadPoolExecutor(max_workers=8)
futures = []
for batch in data:
    future_result = executor.submit(sum, batch)
    futures.append(future_result)
results = [f.result() for f in futures]
```

```
10 loops, best of 3: 116 ms per loop
```

Библиотека multiprocess

- Позволяет работать с процессами как с потоками.

```
from multiprocessing import Process

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


p = Process(target=f, args=('bob',))
p.start()
p.join()
```

multiprocess.Queue

- Аналогичный интерфейс (очереди) поддерживается и с процессами.

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])
    print("task_done()")
    q.task_done()

q = Queue()
p = Process(target=f, args=(q,))
print("p")
p.start()
print("q get")
print(q.get())
print("before")
q.join()
p.join()
```

multiprocessing.Pool

- Еще более удобный способ работы с процессами

```
import multiprocessing
```

```
size = 10 * 1000 * 1000
arr = [1 for _ in range(size)]
process_count = 8
part_size = size // process_count
data = [arr[i * part_size: (i + 1) * part_size] for i in range(process_count)]
```

```
%%timeit
sum(arr)
```

```
10 loops, best of 3: 109 ms per loop
```

```
%%timeit
with multiprocessing.Pool(process_count) as p:
    _ = sum(p.map(sum, data))
```

```
1 loop, best of 3: 1.16 s per loop
```

Пример: обработка запросов на вики

```
def run_async_multiprocess(self):
    with multiprocessing.Pool(8) as executor:
        res = list(executor.map(thread_job_multi_proc,
                                zip([self.__tmp] * len(self.__queries), self.__queries)))
    if not all(map(lambda x: x[0][0].strip() == x[1], zip(res, self.__queries))):
        print("Failed on results order")
    return dict(res)
```

```
%%timeit
res = wr.run_async_multiprocess()
```

```
1 loop, best of 3: 4.37 s per loop
```

Рекомендации

- Несмотря на то, что GIL позволяет работать только одному питоновскому потоку на запущенный процесс, существуют способы нагрузить все имеющиеся ядра процессора.

  – Во первых, если поток не делает вызовов Python C API — то GIL ему не нужен. Так можно держать много параллельно работающих потоков-числодробилок плюс несколько медленных питоновских потоков для управления всем хозяйством. Конечно, для этого нужно уметь писать Python C Extensions.

  – Второй способ еще лучше. Замените «поток» на «процесс». По настоящему высоконагруженная система в любом случае должна строится с учетом масштабируемости и высокой надежности. На эту тему можно говорить очень долго, но хорошая архитектура автоматически позволяет вам запускать несколько процессов на одной машине, которые общаются между собой через какую-либо систему сообщений. В качестве одного из приятных бонусов получается избавление от "проклятия GIL" — у каждого процесса он только один, но процессов много!