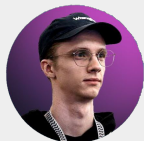


Конкурентное выполнение кода в Python



Денис Воронкин

Терминология

Конкурентность

Конкурентность - это подход, при котором вычисления выполняются в пересекающиеся периоды времени, а не последовательно, когда одно завершается до того, как начнется другое.

Конкурентная система - это система, в которой вычисления могут выполняться не дожидаясь завершения остальных.

Многозадачность

Многозадачность - конкурентное выполнения множества задач в течение определенного периода времени.

Не требует одновременного выполнения выполнения задач. Простой пример - возможность запустить несколько программ одновременно на одноядерных компьютерах.

Параллелизм

Параллелизм - это свойство системы, при котором несколько вычислений выполняются одновременно.

Возможен только в многоядерных системах, когда каждое вычисление выполняется на отдельном ядре процессора.

Асинхронность

Асинхронность - это подход, при котором результат выполнения функции доступен не сразу, а через некоторое время в виде некоторого асинхронного (нарушающего обычный порядок выполнения) вызова.

В асинхронном программировании длительные операции запускаются без ожидания их завершения и не блокируют дальнейшее выполнение программ.

Multiprocessing

Disclaimer: информация относится к Linux.

Процесс - это экземпляр выполняемой программы. Это базовая единица работы в операционной системе.

Ключевые особенности

- уникальный идентификатор (PID)
- виртуальное адресное пространство, отделенное от памяти других процессов
- управляются сигналами (SIGTERM, SIGKILL и тд)
- может находиться в одном из состояний:
 - запущен (running)
 - ожидает (sleeping)
 - остановлен (stopped)
 - зомби (zombie)
 - завершен (terminated)
- один процесс может создавать другой

Информация о процессах (top, htop, btop)

0[3.3%] Tasks: 50, 122 thr, 97 kthr; 1 running									
1[6.0%] Load average: 0.30 0.14 0.10									
Mem[1.02G/1.83G] Uptime: 27 days, 21:52:19									
Swp[0K/1024M]									
Main 170																			
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command								
905537	root	20	0	1209M	17248	10240	S	0.0	0.9	0:00.00	/usr/bin/containerd-shim-runc-v2 --namespace moby --id 8879f5b5b7e6f0a0ec32c677d4ad2a8c14208fe8c214e414836ec8704b5d6031 --ad								
905549	70	20	0	168M	25344	23680	S	0.0	1.3	0:01.00	postgres								
905584	root	20	0	1209M	17248	10240	S	0.0	0.9	0:00.92	/usr/bin/containerd-shim-runc-v2 --namespace moby --id 8879f5b5b7e6f0a0ec32c677d4ad2a8c14208fe8c214e414836ec8704b5d6031 --ad								
905585	root	20	0	1209M	17248	10240	S	0.0	0.9	0:01.15	/usr/bin/containerd-shim-runc-v2 --namespace moby --id 8879f5b5b7e6f0a0ec32c677d4ad2a8c14208fe8c214e414836ec8704b5d6031 --ad								
905618	root	20	0	1632M	5704	3200	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip 0.0.0.0 --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905628	root	20	0	1632M	5704	3200	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip 0.0.0.0 --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905629	root	20	0	1632M	5704	3200	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip 0.0.0.0 --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905630	root	20	0	1632M	5704	3200	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip 0.0.0.0 --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905631	root	20	0	1632M	5704	3200	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip 0.0.0.0 --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905632	root	20	0	1632M	5828	3328	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip :: --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905633	root	20	0	1632M	5704	3200	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip 0.0.0.0 --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905634	root	20	0	1632M	5704	3200	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip 0.0.0.0 --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905636	root	20	0	1632M	5828	3328	S	0.0	0.3	0:00.01	/usr/bin/docker-proxy --proto tcp --host-ip :: --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905637	root	20	0	1632M	5828	3328	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip :: --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905638	root	20	0	1632M	5828	3328	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip :: --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905639	root	20	0	1632M	5828	3328	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip :: --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905640	root	20	0	1632M	5828	3328	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip :: --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905642	root	20	0	1632M	5828	3328	S	0.0	0.3	0:00.00	/usr/bin/docker-proxy --proto tcp --host-ip :: --host-port 4200 --container-ip 172.18.0.4 --container-port 4200								
905664	root	20	0	1208M	16852	10240	S	0.0	0.9	0:00.08	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905666	root	20	0	1208M	16852	10240	S	0.0	0.9	0:01.12	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905667	root	20	0	1208M	16852	10240	S	0.0	0.9	0:00.92	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905668	root	20	0	1208M	16852	10240	S	0.0	0.9	0:00.00	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905669	root	20	0	1208M	16852	10240	S	0.0	0.9	0:00.58	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905670	root	20	0	1208M	16852	10240	S	0.0	0.9	0:00.00	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905671	root	20	0	1208M	16852	10240	S	0.0	0.9	0:01.22	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905672	root	20	0	1208M	16852	10240	S	0.0	0.9	0:00.50	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905673	root	20	0	1208M	16852	10240	S	0.0	0.9	0:00.00	/usr/bin/containerd-shim-runc-v2 --namespace moby --id e0c7067bd41a5d76fa22ded1e3d612fedcc7412310c68f0b2dd019ffeb9e8361 --ad								
905705	70	20	0	168M	15448	13696	S	0.0	0.8	0:00.47	postgres: checkpoint								
905706	70	20	0	168M	8280	6656	S	0.0	0.4	0:00.16	postgres: background writer								
905714	70	20	0	168M	8792	7168	S	0.0	0.5	0:01.69	postgres: walwriter								
905715	70	20	0	170M	7000	4992	S	0.0	0.4	0:00.03	postgres: autovacuum launcher								
905716	70	20	0	170M	6104	4224	S	0.0	0.3	0:00.00	postgres: logical replication launcher								

Дочерние процессы

В создании дочерних процессов участвуют два системных вызова: **exec** и **fork**.

Fork

Создает точную копию родительского процесса с новым PID, копируя код, память*, файловые дескрипторы и другие ресурсы.

Exec

Не создает новый процесс, а заменяет текущее содержимое процесса новой программой, загружая ее из исполняемого файла. PID остается прежним.

* обычно память копируется не сразу благодаря Copy-on-Write, но в Python с этим есть сложности
<https://instagram-engineering.com/copy-on-write-friendly-python-garbage-collection-ad6ed5233ddf>

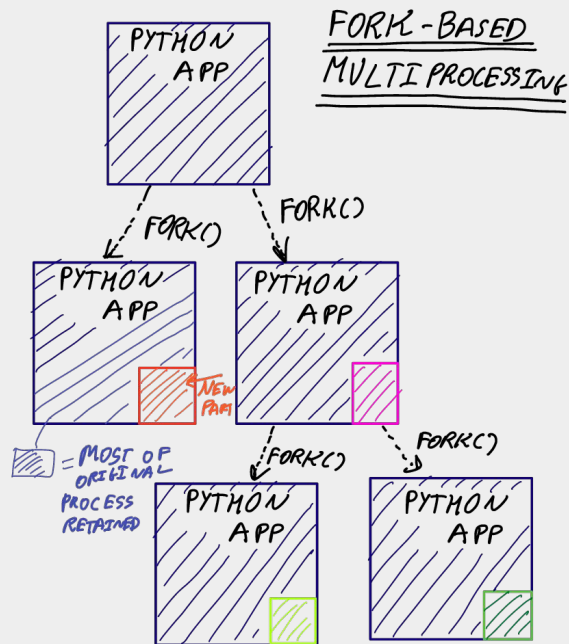
А что в Python?

В Python существует три способа создания дочерних процессов:

- `fork`
- `spawn`
- `forkserver`

Fork

Создает дочерний процесс идентичный родительскому (все ресурсы копируются). Используется по умолчанию при запуске приложения на linux.



Fork

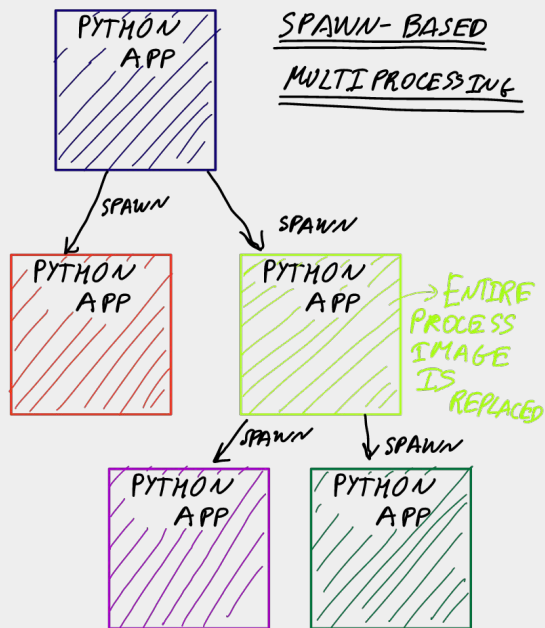
Считается небезопасным, т.к. может приводить к сбоям дочерних процессов. Из-за этого как минимум на macOS по умолчанию используется `spawn`.

Changed in version 3.8: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess as macOS system libraries may start threads. See [bpo-33725](#).

Source: <https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods>

Spawn

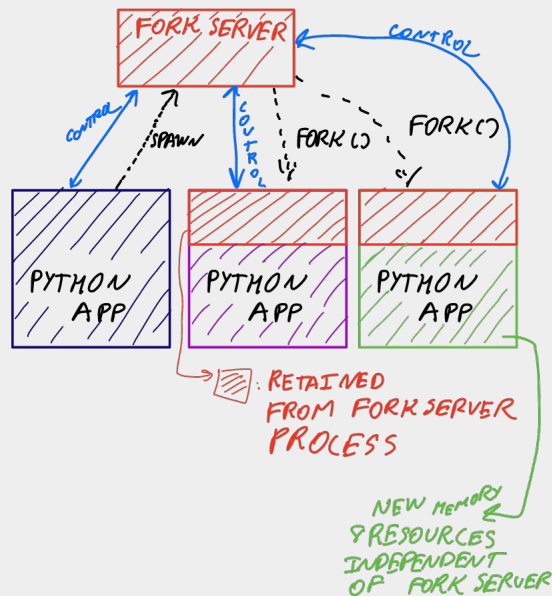
Создает чистый процесс. Копируются только те данные, которые необходимы для его запуска (сериализуя их с помощью pickle).



Forkserver

Подходит для случаев, если у родительского процесса довольно сложное состояние, но при этом хочется оптимизировать скорость создания дочернего процесса.

FORKSERVER - BASED
MULTI PROCESSING



Sources

Блог, из которого я позаимствовал схемы:

<https://bnikolic.co.uk/blog/python/parallelism/2019/11/13/python-forkserver-preload.html>

Тред, в котором неплохо объясняется принцип работы forkserver:

<https://stackoverflow.com/questions/63424251/multiprocessing-in-python-what-gets-inherited-by-forkserver-process-from-parent>

Запуск дочернего процесса в Python



```
1 from os import getpid, getppid
2 from multiprocessing import Process
3
4 def print_pid():
5     print(f"Child PID: {getpid()} | Parent PID: {getppid()}")
6
7 def main() -> None:
8     print(f"Main PID: {getpid()}")
9     process = Process(target=print_pid)
10    process.start()
11
12 if __name__ == "__main__":
13     main()
```

Output




Main PID: 22256

Child PID: 22258 | Parent PID: 22256

Выполнение расчетов в дочернем процессе

```
1 from multiprocessing import Process, Queue
2
3
4 def count_factorial(n: int) -> int:
5     if n == 0 or n == 1:
6         return 1
7     return n * count_factorial(n - 1)
8
9
10 def run_factorial_count(queue: Queue, number: int) -> None:
11     factorial = count_factorial(number)
12     queue.put(factorial)
13
14
15 def main() -> None:
16     number = 5
17     queue = Queue()
18     process = Process(
19         target=run_factorial_count,
20         kwargs={"queue": queue, "number": number},
21     )
22
23     process.start() # Запуск процесса
24     process.join() # Ждем завершения
25
26     print(f"Factorial of {number} = {queue.get()}")
27
28     queue.close()
29
30
31 if __name__ == "__main__":
32     main()
33
```

Output



```
Factorial of 5 = 120
```

ProcessPoolExecutor

```
1 from concurrent.futures import ProcessPoolExecutor
2
3
4 def count_factorial(n: int) -> int:
5     if n == 0 or n == 1:
6         return 1
7     return n * count_factorial(n - 1)
8
9
10 def main() -> None:
11     number = 5
12
13     with ProcessPoolExecutor(max_workers=1) as executor:
14         future = executor.submit(
15             count_factorial,
16             n=number,
17         )
18
19         print(f"Factorial of {number} = {future.result()}")
20
21
22 if __name__ == "__main__":
23     main()
```

Output



Factorial of 5 = 120

ProcessPoolExecutor.map()

```
1 from concurrent.futures import ProcessPoolExecutor
2 from random import randint
3
4
5 def count_factorial(n: int) -> int:
6     if n == 0 or n == 1:
7         return 1
8     return n * count_factorial(n - 1)
9
10
11 def main() -> None:
12     numbers = [randint(1, 10) for _ in range(10)]
13
14     with ProcessPoolExecutor(max_workers=1) as executor:
15         results = executor.map(
16             count_factorial,
17             numbers,
18         )
19
20         for number, factorial in zip(numbers, results):
21             print(f"Factorial of {number} = {factorial}")
22
23
24 if __name__ == "__main__":
25     main()
```

Timeit output



```
Avarage time: 0.042336508398875594
```


Вопрос

У ноутбука, на котором я запускал этот код 10 ядер CPU. Насколько изменится среднее время, если увеличить количество воркеров в пулле до 10?


Ответ

Почти вдвое!.. Медленнее?



```
Avarage time: 0.07140122079872526
```

Однопоточное решение



```
1 from random import randint
2
3 def count_factorial(n: int) -> int:
4     if n == 0 or n == 1:
5         return 1
6     return n * count_factorial(n - 1)
7
8
9 def main() -> None:
10     numbers = [randint(1, 10) for _ in range(10)]
11     factorials = (count_factorial(n) for n in numbers)
12
13     for number, factorial in zip(numbers, factorials):
14         print(f"Factorial of {number} = {factorial}")
15
16 if __name__ == "__main__":
17     main()
```

Timeit output



```
Avarage time: 0.000000760000
```

Почему процессы обходятся дорого

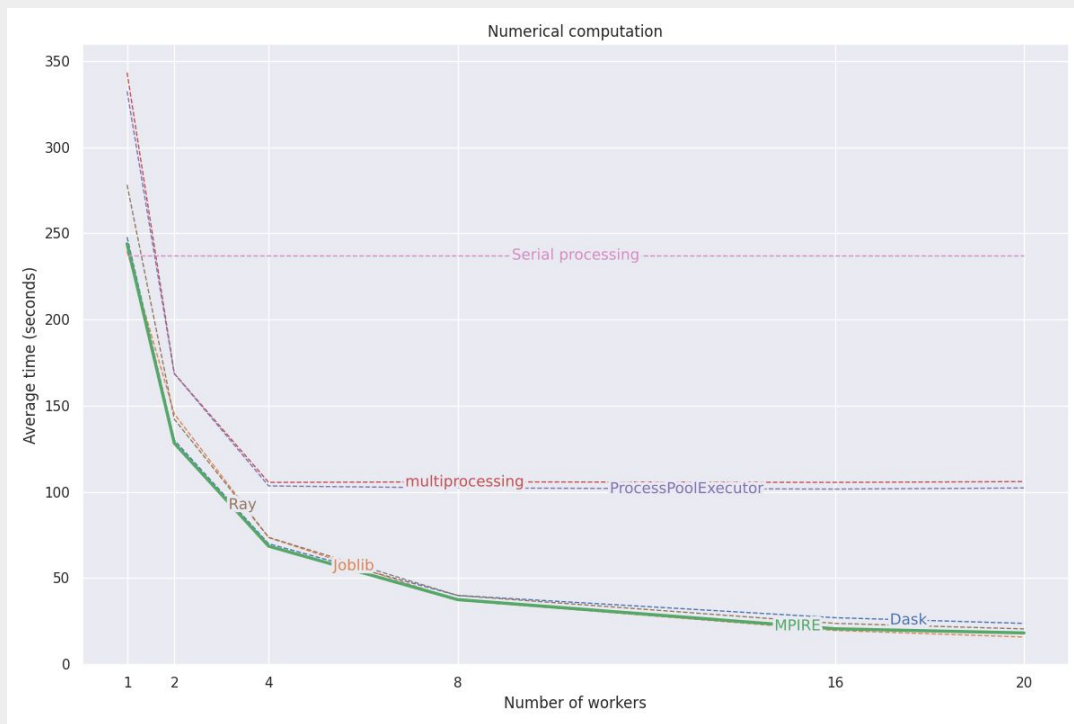
При создании нового процесса операционная система:

- создает новое виртуальное адресное пространство в оперативной памяти
- копирует данные из основного процесса
- добавляет новый PID в список активных процессов
- ...

Выводы

- Не стоит параллелить процессами любую CPU-bound задачу
- Могут быть сложности с запуском на разных ОС
- Подходят, если нужно использовать все ядра процессора

Mpire



Source: <https://github.com/sybrenjansen/mpire>

Multithreading

Поток (или легковесные процессы, LWP) - это последовательность инструкций, которая выполняется независимо, но, в отличие от процессов, делит ресурсы (память, файлы и тд) с другими потоками той же программы.

Ключевые особенности

Линукс работает с потоками очень похожим на процессы образом. Они, как и процессы:

- имеют идентификатор (Thread ID, TID)
- управляются планировщиком ОС
- могут выполняться на разных ядрах процессора (можно ограничить)
- могут находиться в разных состояниях вроде: running, sleeping, waiting

ThreadPoolExecutor

```
1 from concurrent.futures import ThreadPoolExecutor
2 from random import randint
3
4
5 def count_factorial(n: int) -> int:
6     if n == 0 or n == 1:
7         return 1
8     return n * count_factorial(n - 1)
9
10
11 def main() -> None:
12     numbers = [randint(1, 10) for _ in range(10)]
13
14     with ThreadPoolExecutor(max_workers=10) as executor:
15         results = executor.map(
16             count_factorial,
17             numbers,
18         )
19
20         for number, factorial in zip(numbers, results):
21             print(f"Factorial of {number} = {factorial}")
22
23
24 if __name__ == "__main__":
25     main()
```

Timeit output

1 воркер



```
Avarage time: 0.0001681334018940106
```

10 воркеров



```
Avarage time: 0.00030792499892413617
```

Profit?

Да:

Простая замена `ProcessPoolExecutor` на `ThreadPoolExecutor` дала прирост скорости в 100 раз.

Но:

В Python потоки не работают параллельно при выполнении CPU-bound задач.

GIL

GIL (Global Interpreter Lock) - глобальная блокировка интерпретатора, которая позволяет только одному потоку выполнять байт-код и любой момент времени.

Когда GIL не помеха

I/O операции

```
1 class AiofilesContextManager(Awaitable, AbstractAsyncContextManager):
2     """An adjusted async context manager for aiofiles."""
3
4     __slots__ = ("_coro", "_obj")
5
6     def __init__(self, coro):
7         self._coro = coro
8         self._obj = None
9
10    def __await__(self):
11        if self._obj is None:
12            self._obj = yield from self._coro.__await__()
13        return self._obj
14
15    async def __aenter__(self):
16        return await self
17
18    async def __aexit__(self, exc_type, exc_val, exc_tb):
19        await get_running_loop().run_in_executor(
20            None, self._obj._file.__exit__, exc_type, exc_val, exc_tb
21        )
22        self._obj = None
```

Source: <https://github.com/Tinche/aiofiles/blob/main/src/aiofiles/base.py#L58>

Код на C

Примеры:

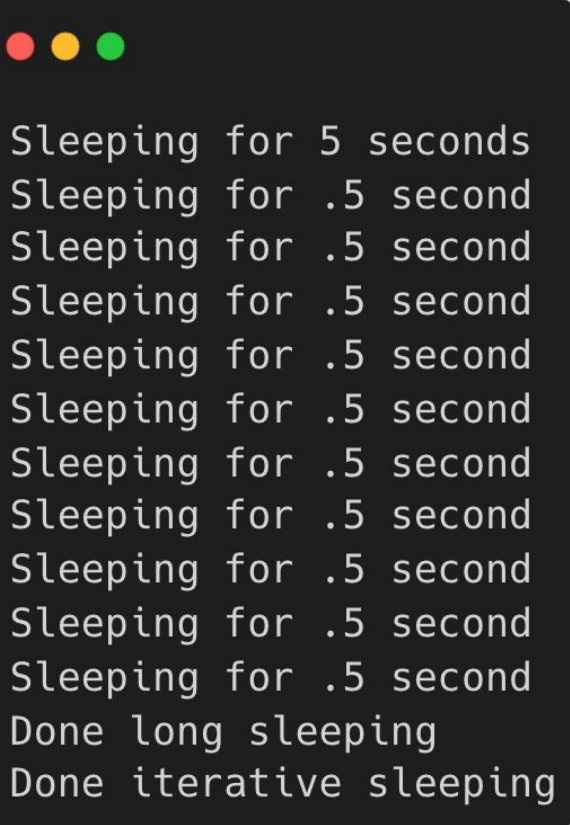
https://numpy.org/doc/stable/reference/global_state.html#number-of-threads-used-for-linear-algebra

https://pandas.pydata.org/docs/user_guide/enhancingperf.html

time.sleep()

```
1 import time
2 from concurrent.futures import ThreadPoolExecutor
3
4
5 def long_sleep() -> None:
6     print("Sleeping for 5 seconds")
7     time.sleep(5)
8     print("Done long sleeping")
9
10
11 def short_sleep() -> None:
12     for _ in range(10):
13         print("Sleeping for .5 second")
14         time.sleep(.5)
15     print("Done iterative sleeping")
16
17 def main() -> None:
18     with ThreadPoolExecutor() as executor:
19         executor.submit(long_sleep)
20         executor.submit(short_sleep)
21
22 if __name__ == "__main__":
23     main()
```

Output



```
Sleeping for 5 seconds  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Sleeping for .5 second  
Done long sleeping  
Done iterative sleeping
```

Низкоуровневые подробности о GIL

Разбор от разработчика Python Никиты Соболева:

https://t.me/opensource_findings/887

https://t.me/opensource_findings/888

Но точно ли без GIL никак?

Довольно давно ведется работа над двумя разными подходами:

- **NoGIL**
- **Subinterpreters**

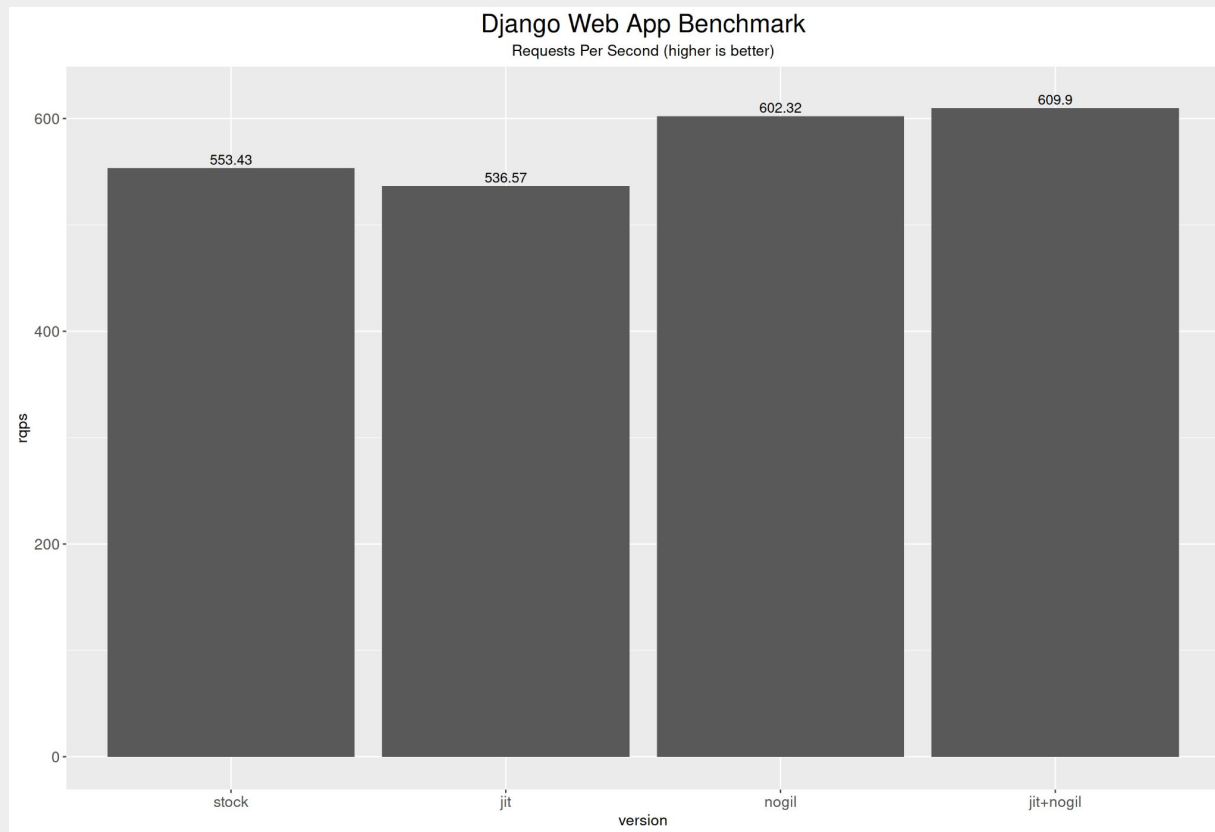
NoGIL

PEP 703 предлагает сделать **GIL** опциональным добавив флаг **--disable-gil** для запуска программы без глобального лока.

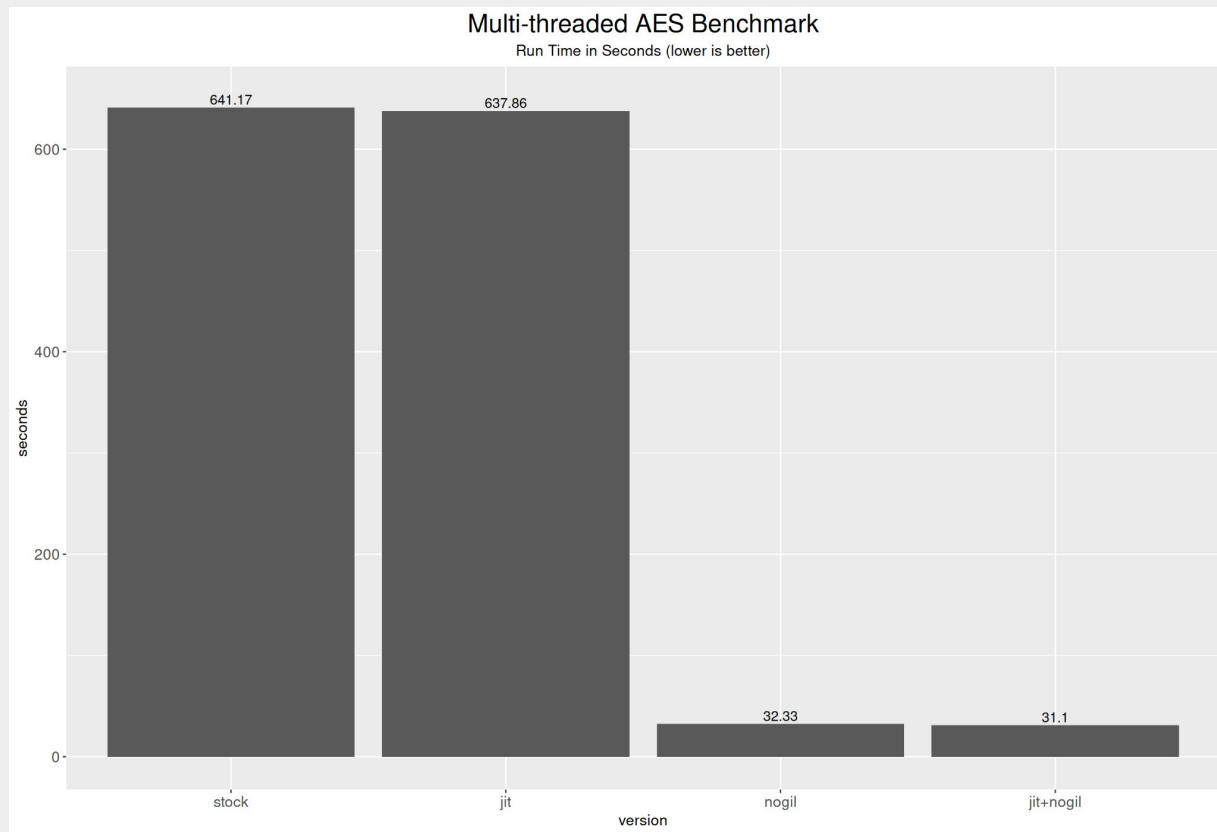
Доступен при сборке интерпретатора из исходников.

Source: <https://peps.python.org/pep-0703/>

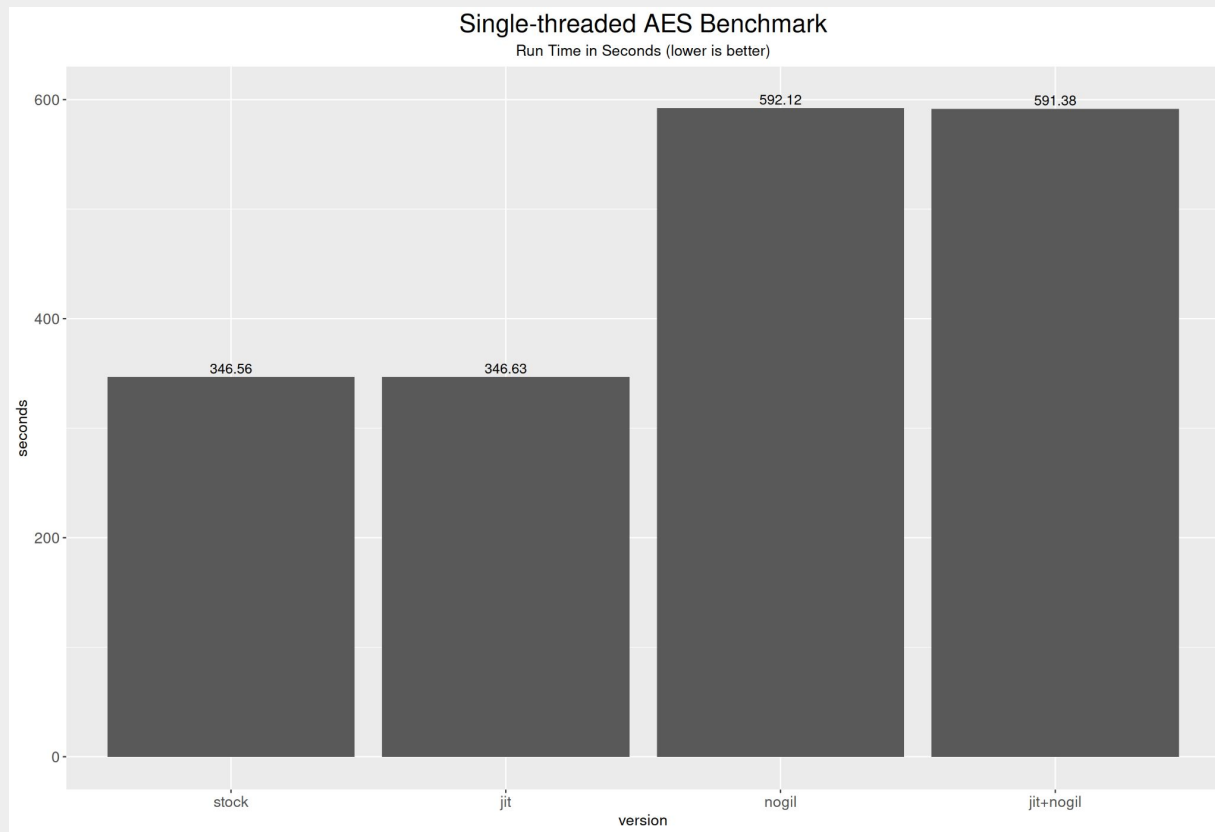
Бенчмарк веб сервера на Django



Многопоточный бенчмарк



Однопоточный бенчмарк



Sources

https://github.com/lip234/python_313_benchmark

Subinterpreters

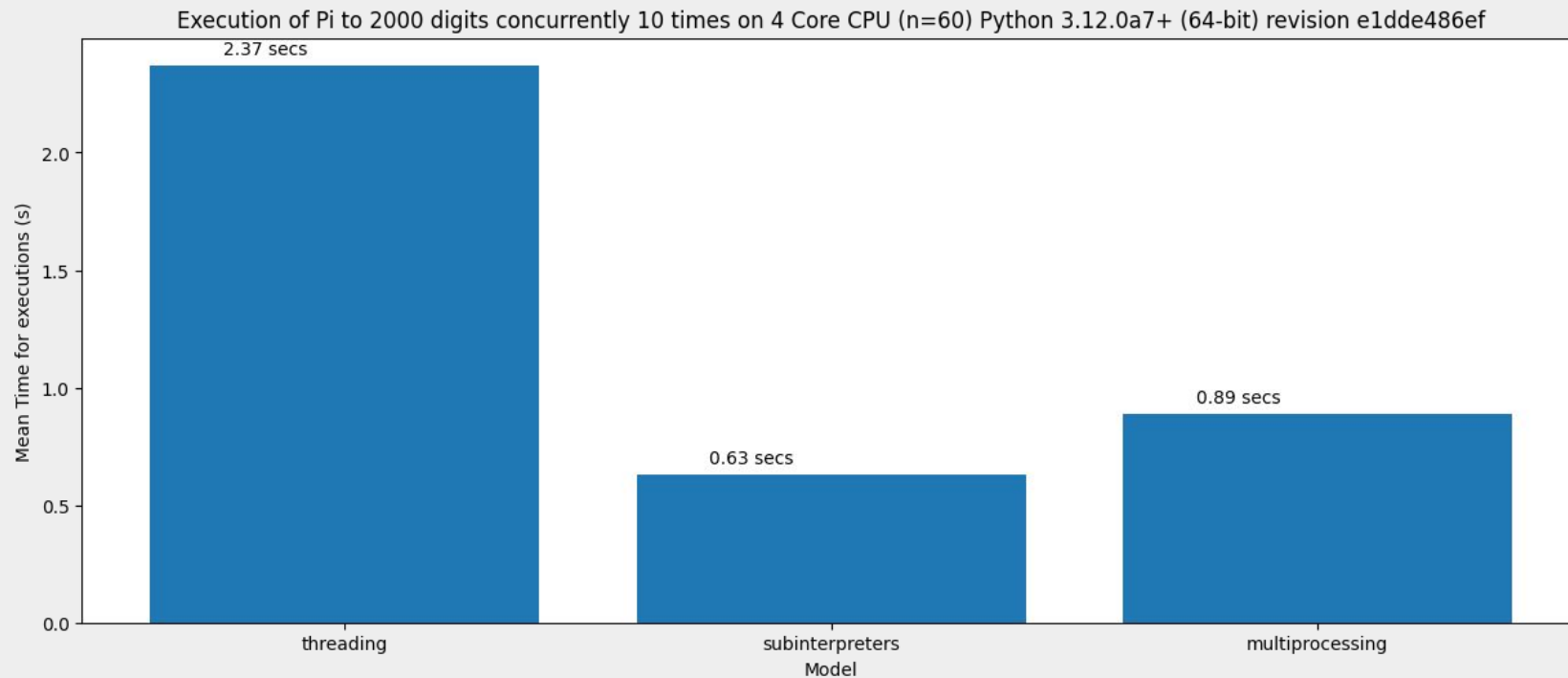
Если **NoGIL** продвигает возможность отключение глобального лока, то PEP 554 и PEP 684 предлагают запускать несколько интерпретаторов в одном процессе, каждый со своим набором глобальных переменных, стеком вызовов и GIL.

Sources:

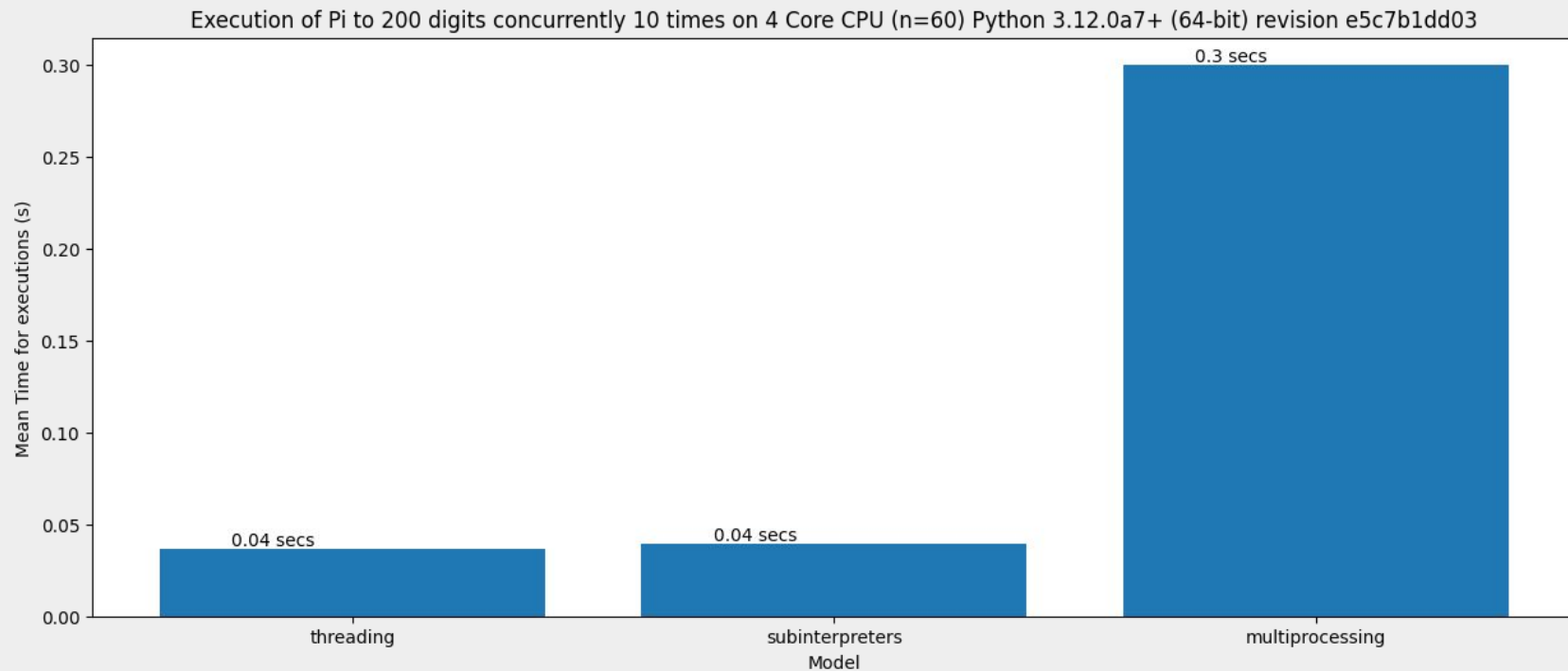
<https://peps.python.org/pep-0554/>

<https://peps.python.org/pep-0684/>

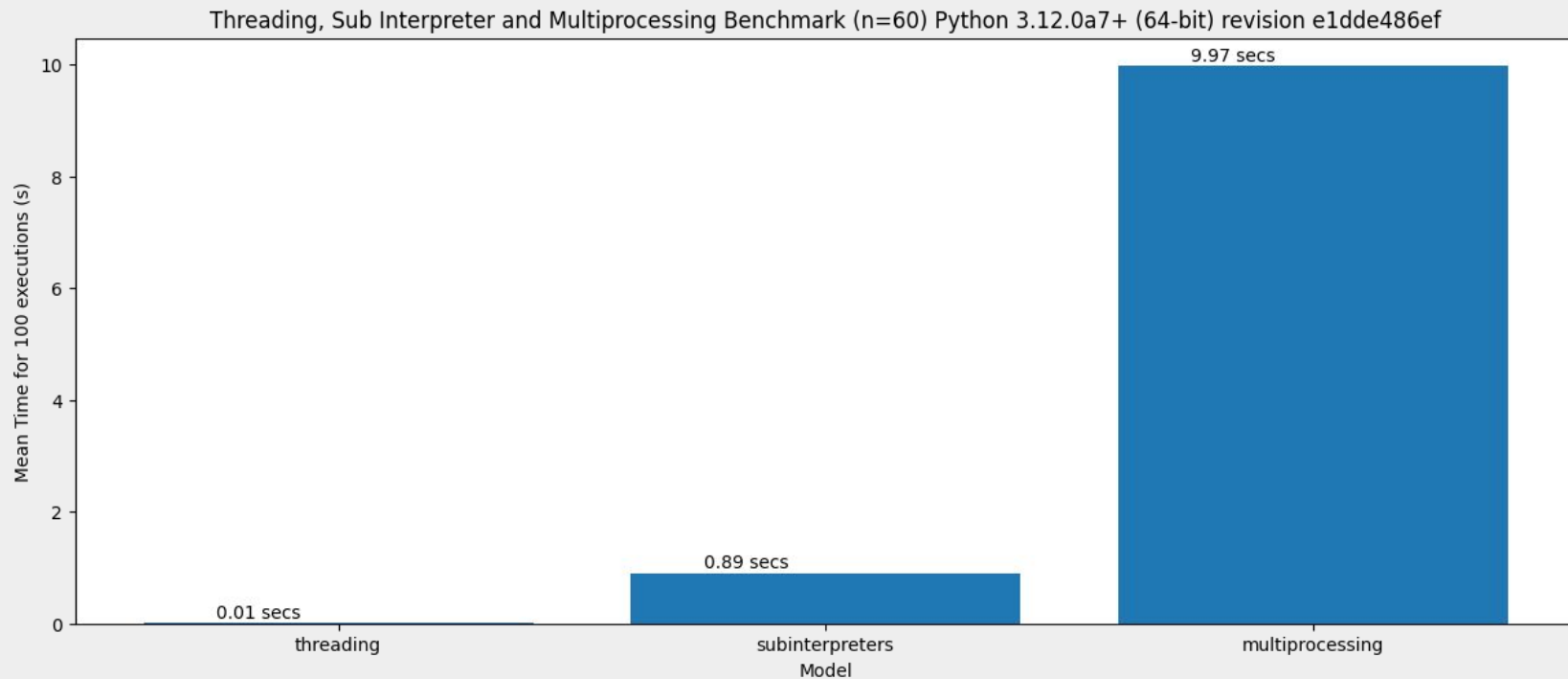
Расчет 2000 знаков числа Пи



Расчет 200 знаков числа Пи



Время создания



Sources

<https://tonybaloney.github.io/posts/sub-interpreter-web-workers.html>

Async

Disclaimer: дальше речь пойдет именно об асинхронном I/O и `asyncio` в частности.

В Python 3.4 добавили библиотеку **`asyncio`**, а в версии 3.5 появился **`async/await`** синтаксис. Зачем это нужно, если потоки уже отлично справлялись с тем, чтобы конкурентно выполнять I/O операции?

Подходы для реализации асинхронности

Существует довольно много подходов. И использование потоков - один из них. Альтернатива - корутины и цикл событий.

В то время уже существовал целый зоопарк библиотек, которые работали по тому же принципу. PEP-492 и PEP-3156 предложили стандартизировать этот функционал в языке введя отдельную библиотеку и синтаксис.

Важной особенностью также стало то, что `async/await` никак не привязан к `asyncio` (как, например, в js), что позволяет использовать его с любой другой библиотекой.

Sources:

https://en.wikipedia.org/wiki/Asynchronous_I/O#Forms

<https://peps.python.org/pep-0492/>

<https://peps.python.org/pep-3156/>

Event Loop

Цикл событий в **asyncio** - ключевая сущность, которая отвечает за:

- управление задачами
- обработка I/O
- управление таймерами
- обработку сигналов (SIGTERM, SIGKILL и тд)
- ...

По аналогии с тем, что `async/await` не привязан к конкретной библиотеке, `asyncio` может использовать различные реализации цикла событий помимо встроенной. Достаточно реализовать интерфейс базового класса.

Низкоуровневые подробности

Для того, чтобы разобраться в принципе работы цикла событий, нужно познакомиться с двумя концепциями: **сетевой сокет** и **селектор**.

Socket

Сетевой сокет - это низкоуровневая абстракция, которая позволяет программам обмениваться данными по сети на транспортном уровне (TCP/UDP).

Любая библиотека, которая отправляет или принимает сетевые запросы, использует сокеты под капотом. По сути, это прослойка между вашим кодом и операционной системой. Для работы с ними в питоне существует библиотека **socket**.

Отправка http запроса с aiohttp

```
1 import aiohttp
2 import asyncio
3
4
5 async def main() -> None:
6     async with aiohttp.ClientSession() as session:
7         async with session.get("http://example.com") as response:
8             status = response.status
9             text = await response.text()
10
11             print(f"Status code: {status}")
12             print(f"Content: {text[:200]}") # Первые 200 символов ответа
13
14
15 if __name__ == "__main__":
16     asyncio.run(main())
```

Отправка http запроса с socket

```
1 import socket
2
3 # Создаём сокет
4 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5
6 # Подключаемся к серверу (example.com, порт 80 для HTTP)
7 sock.connect(("example.com", 80))
8
9 # Формируем HTTP-запрос вручную
10 request = "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
11 sock.sendall(request.encode("utf-8"))
12
13 # Получаем ответ
14 response = b""
15 while True:
16     data = sock.recv(1024) # Читаем по 1024 байта
17     if not data: # Если данных больше нет, выходим
18         break
19     response += data
20
21 # Закрываем сокет
22 sock.close()
23
24 # Выводим результат (декодируем байты в строку)
25 print(response.decode("utf-8")[:200]) # Первые 200 символов
```

Режимы работы сокетов

Сокеты умеют работать в двух режимах:

- блокирующем: **sock.recv()** блокируется до тех пор, пока не придут данные или не будет превышен таймаут
- неблокирующем: **sock.recv()** вызовет исключение, если в нем не будет данных, которые можно прочитать

Selector

Селектор - механизм, который позволяет проверять один и более сокетов на возможность получения или передачи данных через них.

Используя селектор, вместо того, чтобы ждать пока данные придут в один сокет, мы можем сразу пойти читать их из того, в котором уже получен ответ.

Event Loop

Похожим образом работает цикл событий в `asynio`. Он не проверяет каждое соединение отдельно на наличие ответа. Вместо этого он регистрирует все открытые сокеты в селекторе, чтобы сразу получить список тех, которые готовы для выполнения операции.

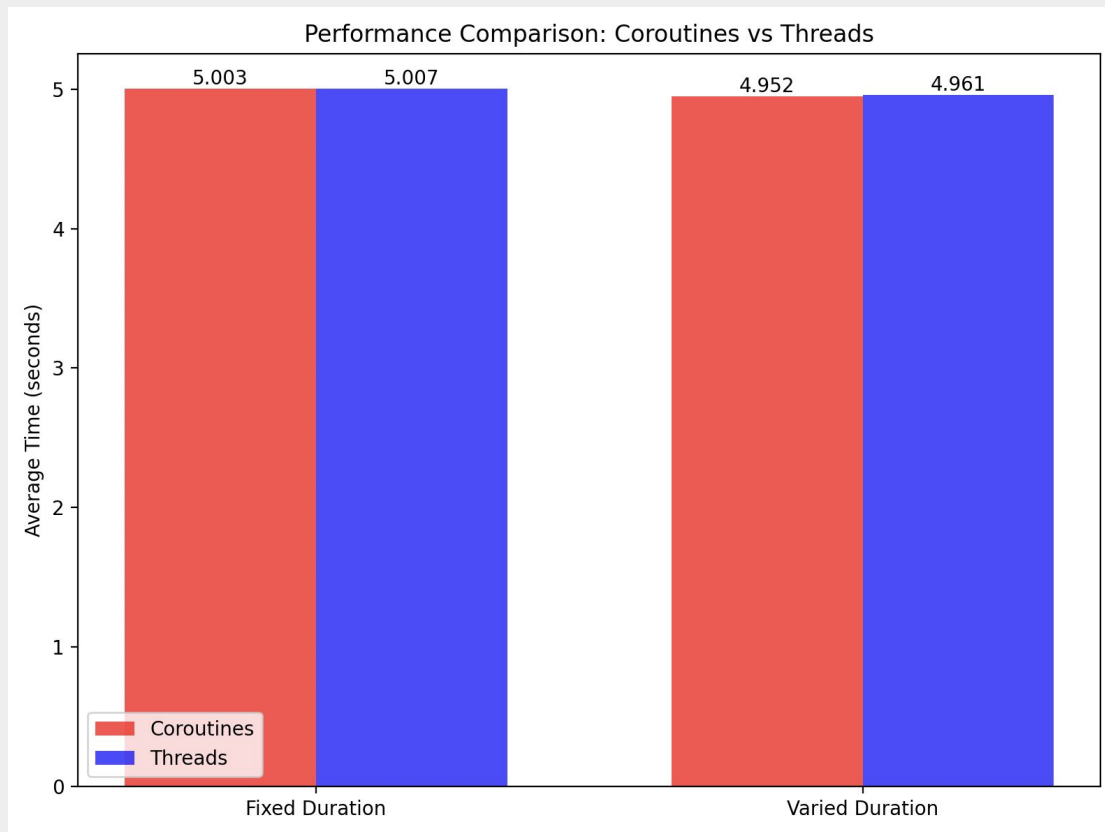
Хоть он и выполняет эти проверки множество раз в секунду, важно учитывать, что цикл однопоточный. Любая блокирующая операция приведет к полной остановке цикла, пока эта операция не завершится.

Сравнение async и multithreading

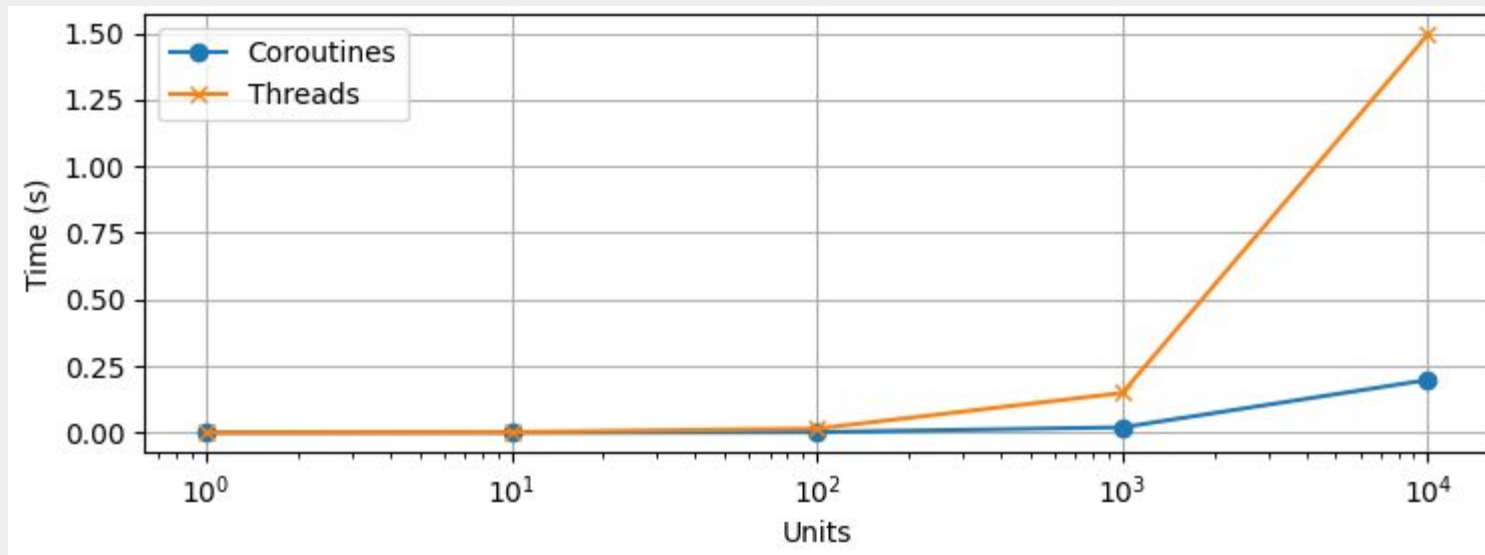
Если задуматься, то асинхронность и многопоточность работают похожим образом. В обоих случаях, программа не блокируется при выполнении запроса, а продолжает работу обрабатывая результат тех операций, которые уже завершились.

Какой подход эффективнее?

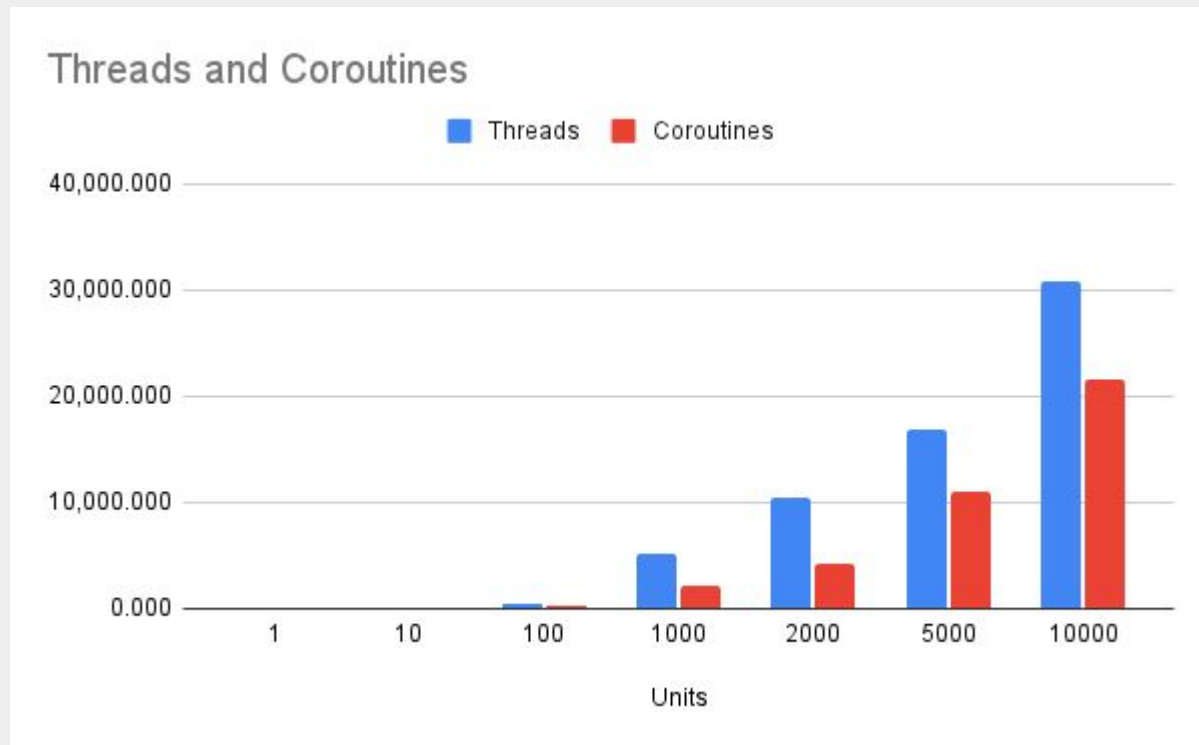
Время обработки операций



Время создания корутин и потоков



Потребление памяти (в килобайтах)



Sources

<https://superfastpython.com/asyncio-coroutines-faster-than-threads/>

За пределами стандартной библиотеки

Асинхронный туллинг развивается продолжительное время. С появлением **asyncio** хоть и пропала необходимость во многих библиотеках, но некоторые все же заслуживают внимания.

Gevent

Gevent основана на другой библиотеке - **greenlet**. Гринлеты - это так называемые "зеленые потоки". Они позволяют вручную переключаться между разными функциями в рамках одного системного потока, сохраняя при этом состояние каждой функции.

Gevent же в свою очередь берет на себя управление переключением между функциями. Она делает **monkey-patching** сокетов и других блокирующих вызовов на свои асинхронные аналоги.

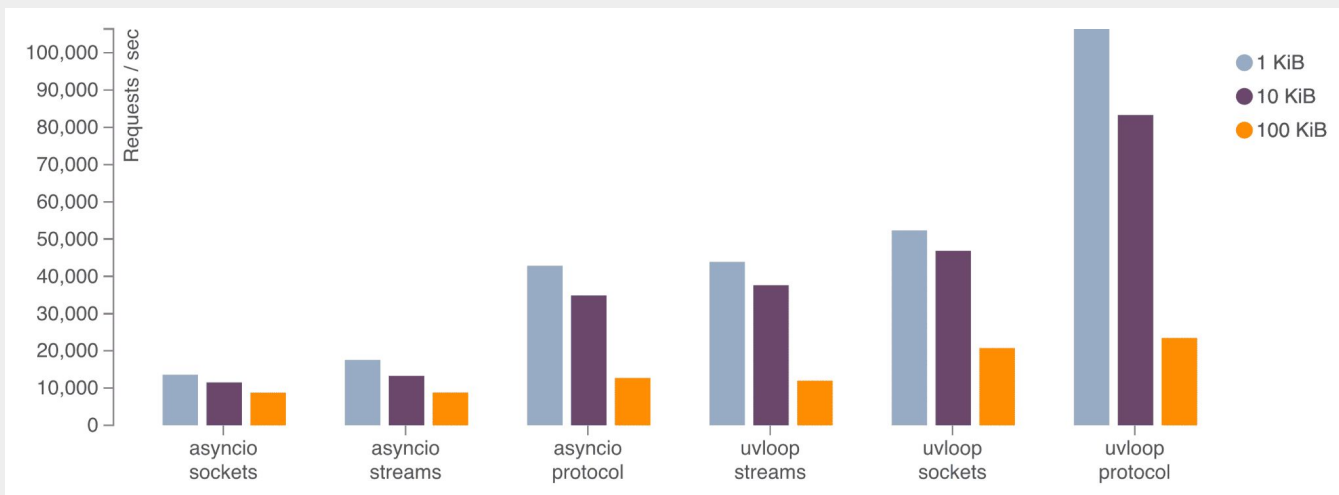
Сейчас вполне поддерживается и даже используется на проектах:

<https://youtu.be/JUGjCc-cRLk?si=GiS7mlwGTSyJfk4Y>

Source: <https://www.gevent.org/>

uvloop

uvloop - более производительный цикл событий для **asyncio** на основе **libuv**. Это кроссплатформенная библиотека, которая изначально разрабатывалась для Node.js.



Source: <https://uvloop.readthedocs.io/>

AnyIO

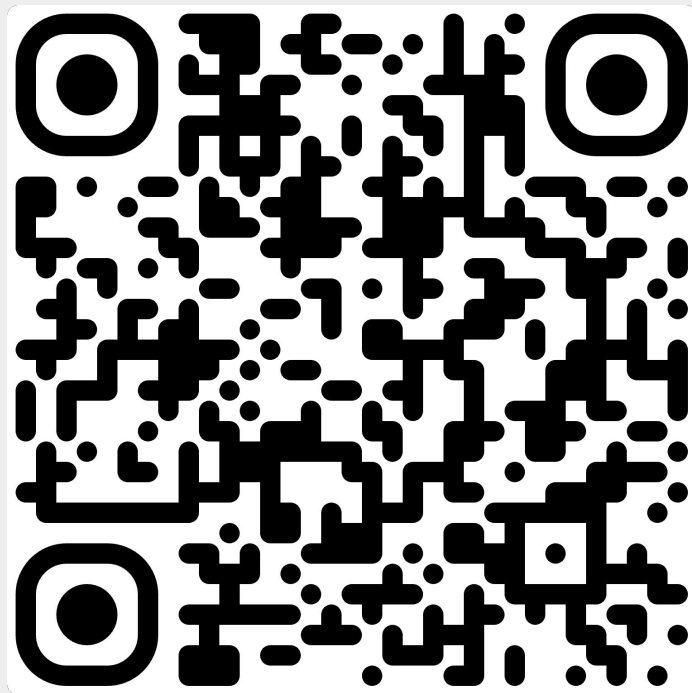
AnyIO предоставляет унифицированный асинхронный интерфейс для работы с I/O операциями.

Из интересных фич:

- асинхронная работа с файлами
- возможность использования **trio** вместо **asyncio**
- возможность настройки различных бекендов и циклов событий для тестов

Source: <https://anyio.readthedocs.io/en/stable/>

Вопросы?



Ссылка на файл с презентацией