Python

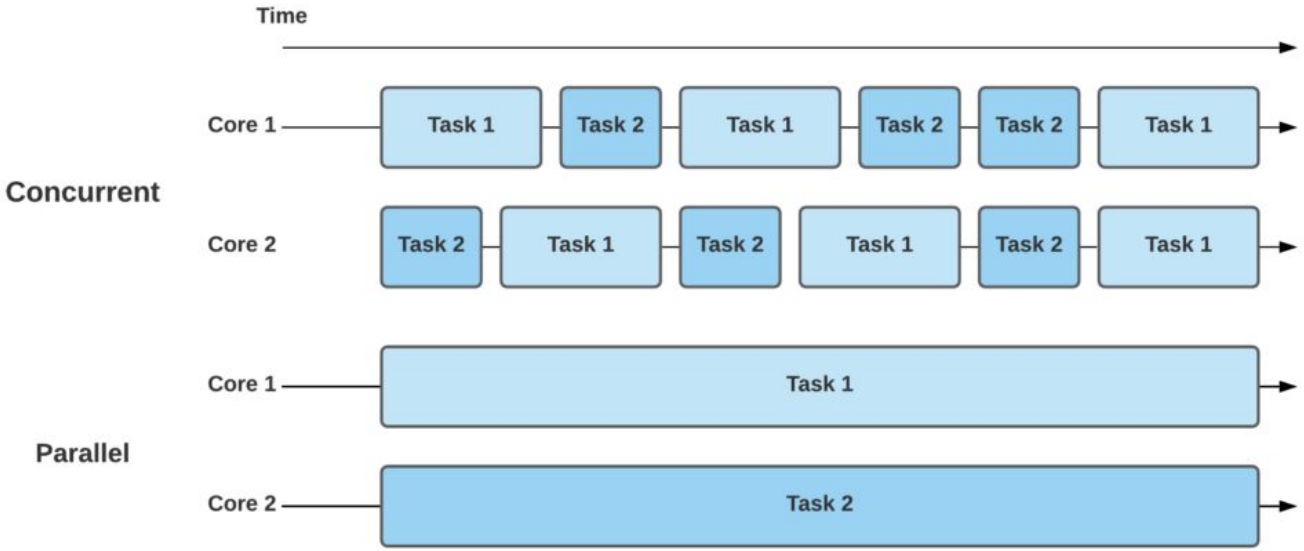
Лекция 8. Многозадачность

Сайфулин Дмитрий, Слободкин Евгений
ИТМО, 11 декабря 2023

MASTER OF
SOFTWARE
ENGINEERING

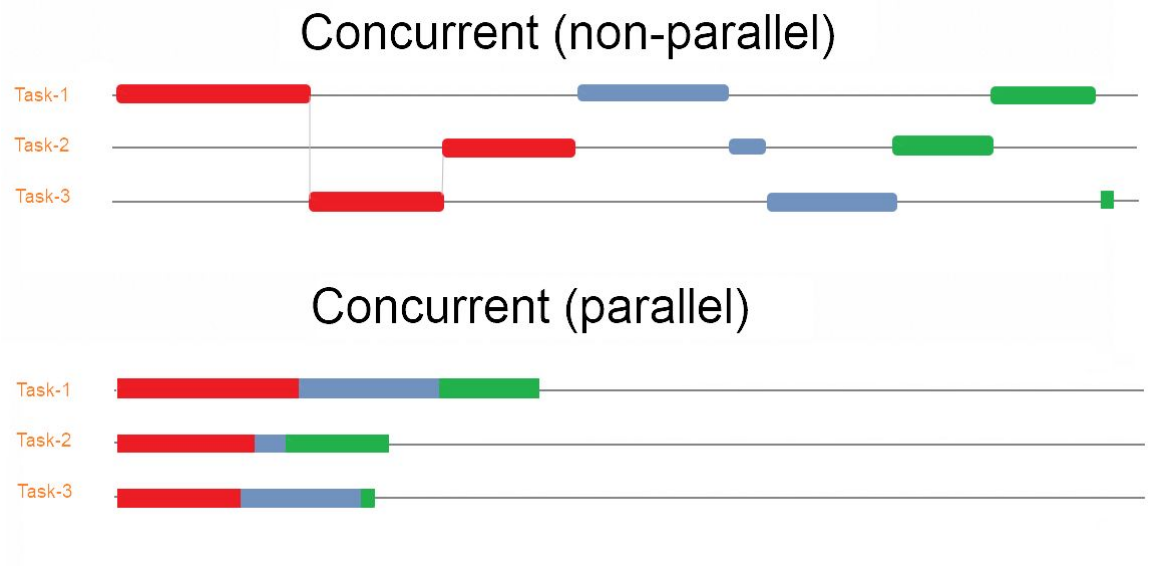
Конкурентность и параллелизм

Определимся с терминологией



Определимся с терминологией

```
# Конкурентность может быть и на одном процессоре, параллелизм — нет.
```



Многопроцессность

Пример

```
# Допустим у нас есть какая-то функция, которая генерирует отчёт.
# Построение одного отчёта занимает около 1 минуты.
# К нам пришли и просят посчитать 20 таких отчётов:
```

Пример

```
# Допустим у нас есть какая-то функция, которая генерирует отчёт.
# Построение одного отчёта занимает около 1 минуты.
# К нам пришли и просят посчитать 20 таких отчётов:

def generate_report(info):
    time.sleep(60)  # На самом деле, там реальный код построения отчёта.
    return report_path


report_paths = []
for info in infos:
    report_path = generate_report(info)
    report_paths.append(report_path)

print('Find your reports:', ', '.join(report_paths))
```

Пример

```
# Запустив данный скрипт мы потратим около 20 минут!

$ time python main.py > /dev/null
python main.py > /dev/null  …  20:00.52 total
```

Как можем это ускорить?

Как можем это ускорить?

```
# Запускать каждое построение отчёта в отдельном процессе.
```

Как можем это ускорить?

```
# Запускать каждое построение отчёта в отдельном ~~процессе~~. Или потоке? Нет,
всё-таки, в процессе. А чем они вообще отличаются? 🤨
```

Как можем это ускорить?

```
# Выбирать как масштабироваться в Python, потоками или процессами, нужно
исходя из того, к какому классу задач относится наша.

# Традиционно, выделяют два класса задач:
# 1. CPU-bound — вычислительно-сложные задачи.
# 2. IO-bound — задачи, в которых основную часть мы ждём данных (ввод-вывод).


# Допустим, что построение нашего отчёта — CPU-bound задача (например, мы
делаем в нём кластеризацию).
```

multiprocessing.Process

```
import multiprocessing as mp
import time

def hello():
    time.sleep(5)  # Вычислительно-сложная задача.
    p = mp.current_process()
    print(f'hello from process {p.pid} {p.name}')
```

multiprocessing.Process

```
import multiprocessing as mp
import time

def hello():
    time.sleep(5)  # Вычислительно-сложная задача.
    p = mp.current_process()
    print(f'hello from process {p.pid} {p.name}')


if __name__ == '__main__':
    hello_process()

    p1 = mp.Process(target=hello, name='custom pname')
    p2 = mp.Process(target=hello)

    p1.run()
    p2.run()
    p1.join()
    p2.join()
```

multiprocessing.Process

```
import multiprocessing as mp
import time

def hello():
    time.sleep(5)
    p = mp.current_process()
    print(f'hello from process {p.pid} {p.name}')

if __name__ == '__main__':
    hello_process()

    p1 = mp.Process(target=hello, name='custom pname')
    p2 = mp.Process(target=hello)

    p1.run()
    p2.run()
    p1.join()
    p2.join()
```

```
hello from process 47781 MainProcess
hello from process 47781 MainProcess
hello from process 47781 MainProcess
Traceback (most recent call last):
  ...
  /python3.11/multiprocessing/process.py", line 148, in join
    assert self._popen is not None, 'can only join a started process'
           ^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: can only join a started process
```

multiprocessing.Process

```
import multiprocessing as mp
import time

def hello():
    time.sleep(5)  # Вычислительно-сложная задача.
    p = mp.current_process()
    print(f'hello from process {p.pid} {p.name}')


if __name__ == '__main__':
    hello_process()

    p1 = mp.Process(target=hello, name='custom pname')
    p2 = mp.Process(target=hello)

    p1.start()  # API скопировано из Thread из Java... В threading такое же.
    p2.start()
    p1.join()
    p2.join()
```

multiprocessing.Process

```
import multiprocessing as mp
import time

def hello():
    time.sleep(5)
    p = mp.current_process()
    print(f'hello from process {p.pid} {p.name}')

if __name__ == '__main__':
    hello_process()

    p1 = mp.Process(target=hello, name='custom pname')
    p2 = mp.Process(target=hello)

    p1.start()  # API скопировано из Thread из Java... В threading такое же.
    p2.start()
    p1.join()
    p2.join()
```

```
hello from process 47662 MainProcess
hello from process 47670 Process-2
hello from process 47669 custom pname
```

multiprocessing и concurrent.futures

```
# Каждый `start` Process-а запускает новый процесс в ОС с питоновским интерпретатором. Если у нас есть 100 задач, то мы реально запустим 100 процессов!

# Чтобы не иметь огромного числа запущенных процессов, обычно пользуются пулом процессов:

def print_n(n):
    time.sleep(5)  # Вычислительно-сложная задача.
    print(f"{n}, PID: {mp.current_process().pid}")


if __name__ == "__main__":
    with mp.Pool(processes=5) as pool:
        pool.map(print_n, list(range(100)))
```

multiprocessing и concurrent.futures

```
# Каждый `start` Process-а запускает новый процесс в ОС с питоновским
интерпретатором. Если у нас есть 100 задач, то мы реально запустим 100
процессов!
# Чтобы не иметь огромного числа запущенных процессов, обычно пользуются
пулом процессов
def print_n(n):
    time.sleep(5)
    print(f"{n} PID: {mp.current_process().pid}")
if __name__ == "__main__":
    with mp.Pool(processes=5) as pool:
        pool.map(print_n, list(range(100)))
```

```
0, PID: 48559
5, PID: 48560
10, PID: 48561
15, PID: 48562
20, PID: 48563
# 5 seconds later...
1, PID: 48559
6, PID: 48560
11, PID: 48561
16, PID: 48562
21, PID: 48563
```

multiprocessing и concurrent.futures

```
# В concurrent.futures имеется ProcessPoolExecutor. Он пользуется сущностями
из multiprocessing и предоставляет более простой интерфейс:


from concurrent.futures import ProcessPoolExecutor


def print_n(n):
    time.sleep(5)  # Вычислительно-сложная задача.
    print(f"{n}, PID: {mp.current_process().pid}")


if __name__ == "__main__":
    with ProcessPoolExecutor(processes=5) as pool:
        pool.map(print_n, list(range(100)))
```

Межпроцессное взаимодействие

```
# Процессам каким-то образом нужно общаться друг с другом. Какие вы знаете IPC?

# В multiprocessing имеются классы Pipe (двунаправленный канал) и Queue (если есть много consumer-ов и producer-ов).

# Как мы передаём данные из одного процесса в другой? На самом деле, этот вопрос должен был появиться ранее.
```

Многопоточность

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

А почему мы не масштабировались потоками?

```
# Пример взят из https://www.dabeaz.com/python/GIL.pdf:


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

if __name__ == "__main__":
    count(100000000)
    count(100000000)


# $ time python main.py > /dev/null
# 4.89s user 0.01s system 98% cpu 4.956 total
```

А почему мы не масштабировались потоками?

```
# Пример взят из https://www.dabeaz.com/python/GIL.pdf:


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

if __name__ == "__main__":
    t1 = threading.Thread(target=count, args=(100000000,))
    t2 = threading.Thread(target=count, args=(100000000,))

    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

А почему мы не масштабировались потоками?

```
# Пример взят из https://www.dabeaz.com/python/GIL.pdf :


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

if __name__ == "__main__":
    t1 = threading.Thread(target=count, args=(100000000,))
    t2 = threading.Thread(target=count, args=(100000000,))

    t1.start()
    t2.start()
    t1.join()
    t2.join()

# $ time python main.py > /dev/null
# 4.67s user 0.02s system 97% cpu 4.797 total  😤
```

GIL

Почитать подробнее про GIL: https://realpython.com/python-gil

GIL

```
# Глобальная блокировка интерпретатора.
# Нужна для того, чтобы защитить консистентность состояния интерпретатора.

# Например, основной сборщик мусора в CPython основан на подсчёте ссылок. Без
GIL вам будет сложнее контролировать его корректность.
```

GIL

```
# Глобальная блокировка интерпретатора.
# Нужна для того, чтобы защитить консистентность состояния интерпретатора.

# Например, основной сборщик мусора в CPython основан на подсчёте ссылок.
Без GIL вам будет сложнее контролировать его корректность.


# Итог: в каждый момент времени интерпретатор выполняет Python код ровно из одного потока.
```

Не всё так страшно

```
# Чтобы поток не держал GIL постоянно, он его отпускает после нескольких
инструкций (за это отвечает `sys.getcheckinterval()`).

# При системных вызовах GIL тоже отпускается.

# Основные части numpy, keras, matplotlib, ... написаны на C.
```

Нужно ли синхронизировать потоки с GIL-ом?

Нужно ли синхронизировать потоки с GIL-ом?

```
# Обязательно! ☝

# В threading много примитивов синхронизации:

  -   threading.Semaphore
  -   threading.RLock
  -   threading.Condition
  -   threading.Barrier
  -   ...
```

Проект Python nogil

```
# PEP 703: https://peps.python.org/pep-0703/
# GitHub: https://github.com/colesbury/nogil
```

Проект Python nogil

```
# PEP 703: https://peps.python.org/pep-0703/
# GitHub: https://github.com/colesbury/nogil

# Кому мешает GIL?
```

Проект Python nogil

```
# PEP 703: https://peps.python.org/pep-0703/
# GitHub: https://github.com/colesbury/nogil

# Кому мешает GIL?
# Сколько библиотек сейчас завязано на GIL?
```

Проект Python nogil

```
# PEP 703: https://peps.python.org/pep-0703/
# GitHub: https://github.com/colesbury/nogil

# Кому мешает GIL?
# Сколько библиотек сейчас завязано на GIL?
# Во время перехода будем жить с GIL-ом под флагом...
```

А если сегодня нужно ускорять?

Без GIL:

- numba: https://numba.pydata.org/
- Jython: https://www.jython.org/
- IronPython: https://ironpython.net/
- PyPy-stm: https://doc.pypy.org/en/latest/stm.html

Компиляция в C:

- Cython: https://cython.org/
- mypyc: https://mypyc.readthedocs.io/en/latest/introduction.html

Субинтерпретаторы

https://peps.python.org/pep-0554/
https://peps.python.org/pep-0684/

На один интерпретатор — один GIL

```
# Субинтерпретаторы в Python есть аж с 1.5. Но непублично!
# В 3.12 зафиксировали C API для них.
# В 3.13 выйдут официально!
```

На один интерпретатор — один GIL

```
# Каждый интерпретатор имеет свой GIL, GC, копии модулей/классов/переменных.
# При этом файловые дескрипторы, статические данные модулей и синглтоны
глобальны.
```

Пример использования

```
import _xxsubinterpreters

class InterpreterThread(threading.Thread):
    def __init__(self):
        super().__init__()
        self._interp = _xxsubinterpreters.create()

    def run(self):
        code = textwrap.dedent("""
            n = 50_000
            fact = 1
            for i in range(1, n + 1):
                fact = fact * i
        """)

        _xxsubinterpreters.run_string(self._interp, code)
        _xxsubinterpreters.destroy(self._interp)
```

Пример взят из https://conf.python.ru/moscow/2021/abstracts/7896

Корутины

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

Вспомним генераторы

```
# Мы использовали yield from, чтобы «вытянуть» значение из генератора:

def g1():
    yield 1

def g2():
    yield from g1()
```

Вспомним генераторы

```
# Мы использовали yield как выражение, чтобы передавать значение в генератор
с помощью send.

def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        if val is not None:
            i = val
        else:
            i += 1

>>> it = counter(10)
>>> next(it)
0
>>> it.send(8)
8
```

Мы уже знаем что такое корутина!

```
# На генераторах мы уже смогли построить функцию, которая может поработать,
остановиться и потом продолжит своё выполнение.
```

Мы уже знаем что такое корутина!

```
# Во многом async/await — синтаксическая обёртка над генераторами:

# Было:
def g1(x):
    yield x

def g2(x):
    return 1 + (yield from g1(x))
```

Мы уже знаем что такое корутина!

```
# Во многом async/await — синтаксическая обёртка над генераторами:

# Было:
def g1(x):
    yield x

def g2(x):
    return 1 + (yield from g1(x))


# Стало:
async def g1(x):
    return x

async def g2(x):
    return 1 + await g1(x)
```

Мы уже знаем что такое корутина!

```
# На генераторах мы уже смогли построить функцию, которая может поработать,
остановиться и потом продолжит своё выполнение.
```

Мы уже знаем что такое корутина!

```
# На генераторах мы уже смогли построить функцию, которая может поработать, остановиться и потом продолжит своё выполнение. Потом что-то опять запустит эту функцию с того места, на котором она остановилась. Плюс, этот кто-то/что-то может ещё и передать что-либо в функцию.
```

Event Loop

```
# Всем этим занимается Event Loop. А, именно, он запускает корутину, управляет
какую корутину сейчас запустить и что ей передать.
```

Event Loop

```
# Всем этим занимается Event Loop. А, именно, он запускает корутину,
управляет какую корутину сейчас запустить и что ей передать.
```

 # Реализаций данного цикла событий много:

- **Стандартный asyncio: https://docs.python.org/3/library/asyncio.html**
- **uvloop от авторов async/await: https://github.com/MagicStack/uvloop**
- **trio: https://github.com/python-trio/trio**

 # Доклад David Beazley, в котором он пишет свой цикл:
https://www.youtube.com/watch?v=MCs5OvhV9S4&ab_channel=PyCon2015

asyncio

```
# Предоставляет абстракции для работы с асинхронным кодом:


import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

main()
```

asyncio

```
# Предоставляет абстракции для работы с асинхронным кодом:


import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

main()
```

```
RuntimeWarning: coroutine 'main' was never awaited
  main()
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
```

```
# «Hello World» не было напечатано! Почему?
```

asyncio

```
# Предоставляет абстракции для работы с асинхронным кодом:


import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

main()
```

```
>>> type(main)
<class 'function'>

>>> type(main())
<class 'coroutine'>
```

asyncio

```
# Предоставляет абстракции для работы с асинхронным кодом:


import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())

# Теперь увидим «Hello World»!
```

async vs sync

```
# Из-за того, что у вас отличается способ работы с асинхронной и синхронной функций, то у вашего кода может появиться два «режима работы».


# Например, SQLAlchemy имеет две функции create_engine и create_async_engine.
```

Почему всё не писать асинхронно?

Почему всё не писать асинхронно?

```
# Точнее вопрос звучит: «Для какого типа задач лучше подходит асинхронность?»


async def foo():
    hard_rabota()  # Нас тут не снимут с исполнения.
    await client.get_user(user_id)
    hard_rabota_2()  # Будем блокировать остальные корутины.
```

Итоги

Когда что использовать?

Модуль	Конкурентность	Параллелизм
threading	+	-
multiprocessing	+	+
asyncio	+	-
subinterpreters	+	+

Когда что использовать?

Модуль	CPU-bound	IO-bound
threading	-	+
multiprocessing	+	-
asyncio	-	+
subinterpreters	+	-+