ТЕХНОАТОМ

Углубленный Python

Лекция 9: асинхронное программирование

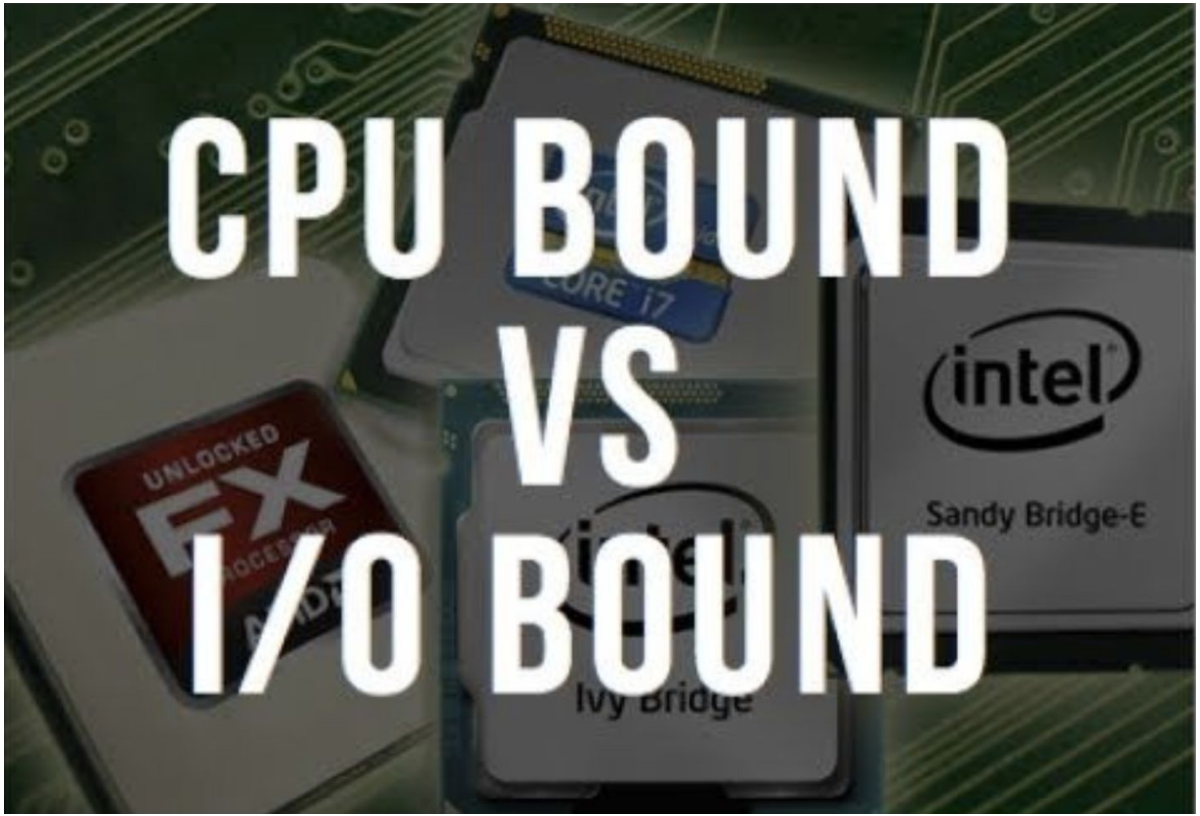
Кандауров Геннадий

Не забудьте отметиться на занятии!

Лекция 9: повестка дня

ТЕХНОАТОМ

1. Асинхронное программирование
2. event loop
3. корутины/нативные корутины
4. asyncio
5. web фреймворки


CPU BOUND
VS
I/O BOUND
UNLOCKED
FX
intel
Sandy Bridge-E
Ivy Bridge

```
import socket

server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_sock.bind(('localhost', 15000))
server_sock.listen()

while True:
    client_sock, addr = server_sock.accept()

    while True:
        data = client_sock.recv(4096)
        if not data:
            break
        else:
            client_sock.send(data.decode().upper().encode())

    client_sock.close()
```

- connect, accept, recv, send - блокирующие операции
- C10k problem, http://kegel.com/c10k.html
- Потоки дорого стоят (CPU & RAM)
- Потоки простаивают часть времени

Системные вызовы:

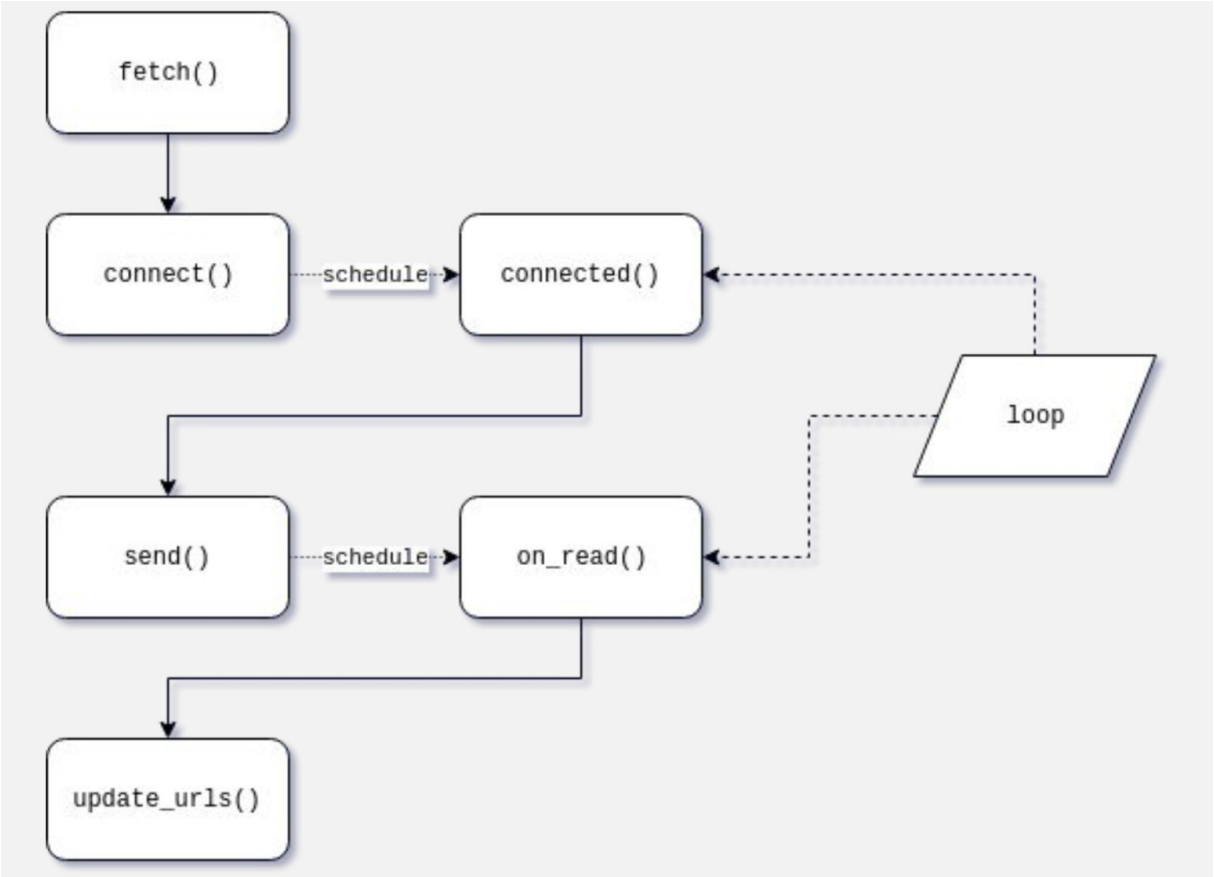
- select (man 2 select)
- poll (man 2 poll)
- epoll (man 7 epoll)
- kqueue

python:

- select
- selectors

```
def event_loop():
    while True:
        ready_to_read, _, _ = select(to_monitor, [], [])

        for sock in ready_to_read:
            if sock is server_sock:
                accept_conn(sock)
            else:
                respond(sock)
```

```
import selectors

selector = selectors.DefaultSelector()
selector.register(server_sock, selectors.EVENT_READ, accept_conn)


def event_loop():
    while True:
        events = selector.select()  # (key, events_mask)

        for key, _ in events:
            # key: NamedTuple(fileobj, events, data)
            callback = key.data
            callback(key.fileobj)
            # selector.unregister(key.fileobj)
```


fetch()
connect()
schedule
connected()
loop
send()
schedule
on_read()
update_urls()

Дэвид Бизли (David Beazley), "Python Concurrency From the Ground Up: LIVE!"

```
def event_loop():
   while any([tasks, to_read, to_write]):
       while not tasks:
           ready_to_read, ready_to_write, _ = select(to_read, to_write, [])
           for sock in ready_to_read:
               tasks.append(to_read.pop(sock))
           for sock in ready_to_write:
               tasks.append(to_write.pop(sock))
       try:
           task = tasks.pop(0)
           op_type, sock = next(task)
           if op_type == 'read':
               to_read[sock] = task
           elif op_type == 'write':
               to_write[sock] = task
       except StopIteration:
           pass
```

```
def grep(pattern):
    print('start grep for', pattern)
    while True:
        s = yield
        if pattern in s:
            print('found!', s)
        else:
            print('no %s in %s' % (pattern, s))

g = grep('python')
next(g)
g.send('data')
g.send('deep python')
```

```
$ python generator_socket.py
start grep for python
no python in data
found! deep python
```

- использование ***yield*** более обобщенно определяет корутину
- не только генерируют значения
- потребляют данные, отправленные в них через ***.send***
- отправленные данные возвращаются через ***data = yield***

coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the async def statement. See also **PEP 492**.

```
import asyncio, time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")
    await say_after(1, 'hello')
    await say_after(2, 'world')
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

```
>run.py
started at 16:42:46
hello
world
finished at 16:42:49
```

- 1 процесс
- 1 поток
- кооперативная многозадачность (vs вытесняющая)
- передача управления в event loop на ожидающих операциях
- async/await это API Python, а не часть asyncio

Event loop:

coroutine > Task (Future)

- ***Future*** представляет ожидаемый в будущем (eventual) результат асинхронной операции;
- ***Task*** это *Future-like* объект, запускающий корутины в событийном цикле;
- ***Task*** используется для запуска нескольких корутин в событийном цикле параллельно.

High-level APIs

- Coroutines and Tasks
- Streams
- Synchronization Primitives
- Subprocesses
- Queues
- Exceptions

Low-level APIs

- Event Loop
- Futures
- Transports and Protocols
- Policies
- Platform Support

- asyncio.create_task
- asyncio.sleep
- asyncio.gather
- asyncio.shield
- asyncio.wait_for
- asyncio.wait
- asyncio.Queue
- asyncio.Lock
- asyncio.Event

- aiohttp https://docs.aiohttp.org/en/stable/
- sanic https://sanic.readthedocs.io/en/latest/

Лекция 9: домашнее задание

ТЕХНОАТОМ

1. Написать скрипт для обкачки списка урлов с возможностью задавать количество одновременных запросов. Клиент можно использовать любой, например, из aiohttp. Например, 10 одновременных запросов могут задаваться так:

   ```
   python fetcher.py -c 10 urls.txt
   ```

2. Улучшить пример про выкачку картинок (**`fetcher.py`**) так, чтобы запись на диск выполнялась асинхронно: без использования сторонних либ типа aiofiles - полный балл, если совсем никак, то с либой, но -2 балла в итог.

ТЕХНОАТОМ

Спасибо за внимание!

Кандауров Геннадий

g.kandaurov@corp.mail.ru