(/learn)

Asynchronous with Executors

In this lesson we introduce executors within an asynchronous program and examine how the right choice can significantly improve performance.

Asynchronous with Executors

In the previous section, we wrote an asynchronous version of our service that used the same event loop to juggle between incoming requests and performing calculations. A long-running request made the service completely unavailable. One mitigation we can apply is to offload the calculation work to thread or process pools, same as what we did in our multithreaded approach. The changes required to our service are minimal. Instead of scheduling the prime number calculation on the same event loop, we handoff the work to either a process or thread pool. The changes are shown below:

```
class PrimeService:

    def __init__(self, server_host, server_port):
        self.server_host = server_host
        self.server_port = server_port
        self.executor = ThreadPoolExecutor()

    async def handle_client(self, reader, writer):
        global requests

        while True:
            data = (await reader.read(4096)).decode()
            nth_prime = int(data)

            current_loop = asyncio.get_event_loop()
            prime = await current_loop.run_in_executor(self.ex
ecutor, find_nth_prime, nth_prime)

            writer.write(str(prime).encode())
            await writer.drain()

            requests += 1
```

The rest of the infrastructure remains the same. The code widget below runs the simulation for a few seconds and the results are similar to what we saw in the multithreaded approach. The number of requests completed doesn't drop to absolute zero but are down more than 99% from before the long-running request gets submitted. Also, since we are using a thread pool, the `sys.setswitchinterval()` will have similar effects on the number of requests completed as we saw in the previous multithreaded approach. The code in the widget runs without tinkering with the switch interval but feel free to change that value and observe the outcome.

The code in the widget below has slightly different syntax to account for

Python 3.5.

```
from threading import Thread
from threading import Lock
from concurrent.futures import ThreadPoolExecutor

import socket, time, random, sys, asyncio

requests = 0


def find_nth_prime(nth_prime):
    i = 2
    nth = 0

    while nth != nth_prime:
        if is_prime(i):
            nth += 1
            last_prime = i

        i += 1

    return last_prime


def is_prime(num):
    if num == 2:
        return True

    div = 2

    while div <= num / 2:
        if num % div == 0:
            return False
        div += 1
    return True


def monitor_requests_per_thread():
    global requests
    while 1:
        time.sleep(1)
        print("{0} requests/secs".format(requests), flush=True)
        requests = 0



class PrimeService:

    def __init__(self, server_host, server_port):
        self.server_host = server_host
        self.server_port = server_port
        self.executor = ThreadPoolExecutor()
```

```
    async def handle_client(self, reader, writer):

        global requests

        while True:
            data = (await reader.read(4096)).decode()
            nth_prime = int(data)

            current_loop = asyncio.get_event_loop()
            prime = await current_loop.run_in_executor(self.executor, find_nth_prim

            writer.write(str(prime).encode())
            await writer.drain()

            requests += 1


def run_simple_client(server_host, server_port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.connect((server_host, server_port))

    while 1:
        server_socket.send("1".encode())
        server_socket.recv(4096)


def run_long_request(server_host, server_port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.connect((server_host, server_port))

    while 1:
        server_socket.send("100000".encode())
        server_socket.recv(4096)


async def main():
    server = PrimeService(server_host, server_port)
    await asyncio.start_server(server.handle_client, "127.0.0.1", server_port,)


def server_code():
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    asyncio.ensure_future(main())
    loop.run_forever()


if __name__ == "__main__":
    sys.setswitchinterval(0.002)
    server_port = random.randint(10000, 65000)
    server_host = "127.0.0.1"

    Thread(target=server_code, daemon=True).start()
```

```
    time.sleep(0.1)

    Thread(target=monitor_requests_per_thread, daemon=True).start()
    Thread(target=run_simple_client, args=(server_host, server_port), daemon=True).

    time.sleep(2)
    Thread(target=run_long_request, args=(server_host, server_port), daemon=True).s

    time.sleep(10)
```

Next, we'll switch out the thread pool for a process pool and see significant improvement in service performance. When the long-running request gets submitted, the number of requests completed make a slight drop demonstrating that the GIL is not a limiting factor anymore.

```
from threading import Thread
from threading import Lock
from concurrent.futures import ProcessPoolExecutor

import socket, time, random, sys, asyncio

requests = 0


def find_nth_prime(nth_prime):
    i = 2
    nth = 0

    while nth != nth_prime:
        if is_prime(i):
            nth += 1
            last_prime = i

        i += 1

    return last_prime


def is_prime(num):
    if num == 2:
        return True

    div = 2

    while div <= num / 2:
        if num % div == 0:
            return False
        div += 1
    return True


def monitor_requests_per_thread():
    global requests
    while 1:
        time.sleep(1)
        print("{0} requests/secs".format(requests), flush=True)
        requests = 0



class PrimeService:

    def __init__(self, server_host, server_port):
        self.server_host = server_host
        self.server_port = server_port
        self.executor = ProcessPoolExecutor()
```

```
    async def handle_client(self, reader, writer):

        global requests

        while True:
            data = (await reader.read(4096)).decode()
            nth_prime = int(data)

            current_loop = asyncio.get_event_loop()
            prime = await current_loop.run_in_executor(self.executor, find_nth_prim

            writer.write(str(prime).encode())
            await writer.drain()

            requests += 1


def run_simple_client(server_host, server_port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.connect((server_host, server_port))

    while 1:
        server_socket.send("1".encode())
        server_socket.recv(4096)


def run_long_request(server_host, server_port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.connect((server_host, server_port))

    while 1:
        server_socket.send("100000".encode())
        server_socket.recv(4096)


async def main():
    server = PrimeService(server_host, server_port)
    await asyncio.start_server(server.handle_client, "127.0.0.1", server_port,)


def server_code():
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    asyncio.ensure_future(main())
    loop.run_forever()


if __name__ == "__main__":
    server_port = random.randint(10000, 65000)
    server_host = "127.0.0.1"

    Thread(target=server_code, daemon=True).start()

    time.sleep(0.1)
```

```
    time.sleep(0.1)

    Thread(target=monitor_requests_per_thread, daemon=True).start()
    Thread(target=run_simple_client, args=(server_host, server_port), daemon=True).

    time.sleep(2)
    Thread(target=run_long_request, args=(server_host, server_port), daemon=True).s

    time.sleep(10)
```

In summary, one should be cognizant of the performance hit a program can experience because of the GIL and appropriately choose designs that can scale and be responsive with different workload profiles.

Back
Asynchronous

Next
Blocking Queue | Bounded Buffer | Co...

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/asynchronous-with-executors__global-interpreter-lock__python-concurrency-for-senior-engineering-interviews)