Asynchronous

This lesson demonstrates how we can convert our threaded design to an asynchronous one.

Asynchronous

We have seen our service go through various transformations and now we'll modify our service once more to make it run on asyncio's event loop. The loop runs in a single thread and all tasks get executed by the same thread. First, let us see the changes required for the service.

```
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)

            prime = await asyncio.create_task(find_nth_prime(n
th_prime))

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

            requests += 1
```

Note we have factored out the methods that crunch numbers and also we don't need to setup the server to accept incoming connections. We'll pass the **`handle_clients()`** method as a callback to the asyncio's **`start_server()`** method. The snippet appears below:

```
async def main():
    service = PrimeService(server_host, server_port)
    server = await asyncio.start_server(service.handle_client,
 "localhost", server_port)
    await server.serve_forever()
```

Since **`main()`** is a coroutine we start it using:

```
def server_code():
    asyncio.run(main())
```

For our simulation, we'll have one thread that runs the **`server_code()`** method and also becomes the event loop thread. We also spawn other threads that make client requests. The complete code is given below:

```
requests = 0


async 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))
        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)

            prime = await asyncio.create_task(find_nth_prime(n
th_prime))

            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)
        server_socket.recv(4096)


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


def server_code():
    asyncio.run(main())


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

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

    time.sleep(0.1)

    # run the monitor thread
    Thread(target=monitor_requests_per_thread, daemon=True).st
art()

    # run a simple request
    Thread(target=run_simple_client, args=(server_host, server
_port), daemon=True).start()

    # issue a long-running request after three minutes
    time.sleep(3)
    Thread(target=run_long_request, args=(server_host, server_
port), daemon=True).start()

    time.sleep(10)
```

Again, we don't concern ourselves with graceful shutdown as that is tangential to our discussion. The widget would timeout after running the simulation for a few seconds. Note that the code that appears in the code

widget is slightly different so that it can work with Python 3.5 but the intention is the same as we described.

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

import socket, time, random, sys, asyncio

requests = 0


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

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

            prime = await asyncio.ensure_future(find_nth_prime(nth_prime))

            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)
    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)
```

The output shows that, as soon as the long-running request gets submitted, the number of requests being completed per second drops to zero. This result is consistent with the behavior we witnessed in the case of a single thread server. Since there's a single event loop on which all coroutines get executed, a CPU-bound task brings the entire application to a stop.

In the next section we'll tweak our example to handoff the calculation to an executor.

Back
Multiple Processors

Next
Asynchronous with Executors

Mark as Completed

Report an Issue

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