Multiple Threads

This lesson discusses the implementation of the web-service using multiple threads.

Multiple Threads

To mitigate the issues we experienced in the previous section, we'll modify our service to spawn a new thread to deal with each new client request. We can use a **`ThreadPoolExecutor`** to handle new client connections but for now, we'll simply spawn threads ourselves. The required changes are:

```
    def run_service(self):
        connection = socket.socket()
        connection.bind(('localhost', self.server_port))

        # put the socket into listening mode
        connection.listen(5)

        while True:
            client_socket, addr = connection.accept()
            Thread(target=self.handle_client, args=(client_soc
ket,), daemon=True).start()
```

```
from threading import Thread
from threading import Lock

import socket, time, random, sys


class PrimeService():
```

```python

    def __init__(self, server_port):
        self.server_port = server_port
        self.requests = 0

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

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

            i += 1

        return last_prime

    def is_prime(self, num):
        if num == 2 or num == 3:
            return True
```

The output from the above program shows that our multithreaded approach doesn't alleviate the ills of our previous approach. The number of requests, though, don't drop to zero, but are significantly reduced nevertheless. In fact, the reduction in requests completed is less than 99%. The only reason that the server is able to handle a few requests while one of the threads calculates the 10,000th prime is because the Python interpreter schedules other threads for execution so that all threads get a chance to make progress. However, from the drop in the number of requests completed, we can reason that the interpreter favors the thread involved in the long-running task.

Switch Interval

Python allows us to tweak the switch interval, which is the ideal thread switching delay inside the interpreter. It is the time interval after which the Python interpreter schedules another thread for execution. If we reduce it to a very small value, say 2 milliseconds, we should see more number of requests being completed even with the long-running task. The switch interval can be controlled using the API `sys.setswitchinterval()`. Run the code widget below which changes the switch interval to 2 milliseconds and examine the output.

```
from threading import Thread
from threading import Lock

import socket, time, random, sys


class PrimeService():

    def __init__(self, server_port):
        self.server_port = server_port
        self.requests = 0

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

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

            i += 1

        return last_prime

    def is_prime(self, num):
        if num == 2 or num == 3:
            return True

        div = 2

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

    def monitor_requests_per_thread(self):

        while 1:
            time.sleep(1)
            print("{0} requests/min".format(self.requests), flush=True)
            self.requests = 0

    def handle_client(self, client_socket):
        while 1:
            data = client_socket.recv(4096)
            nth_prime = int(data)
            prime = self.find_nth_prime(nth_prime)
            client_socket.send(str(prime).encode())
            self.requests += 1
```

```
    def run_service(self):

        connection = socket.socket()
        connection.bind(("127.0.0.1", self.server_port))

        # put the socket into listening mode
        connection.listen(5)

        while True:
            client_socket, addr = connection.accept()
            Thread(target=self.handle_client, args=(client_socket,), daemon=True).s


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)


if __name__ == "__main__":

    # change the switch interval to 2 milliseconds
    sys.setswitchinterval(0.002)

    server_port = random.randint(10000, 65000)
    server_host = "127.0.0.1"
    server = PrimeService(server_port)

    server_thread = Thread(target=server.run_service, daemon=True)
    server_thread.start()

    monitor_thread = Thread(target=server.monitor_requests_per_thread, daemon=True)
    monitor_thread.start()

    simple_req_thread = Thread(target=run_simple_client, args=(server_host, server_
    simple_req_thread.start()

    time.sleep(3)

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

    time.sleep(1000)
```

```
    time.sleep(1000)
```

The output shows that the number of requests completed once a long-running request has already been submitted is roughly double than the number of requests completed without tweaking the switch interval. On the flip side, if we increase the switch interval to say 1 second, the number of requests completed would drop significantly.

```
from threading import Thread
from threading import Lock

import socket, time, random, sys


class PrimeService():

    def __init__(self, server_port):
        self.server_port = server_port
        self.requests = 0

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

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

            i += 1

        return last_prime

    def is_prime(self, num):
        if num == 2 or num == 3:
            return True

        div = 2

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

    def monitor_requests_per_thread(self):

        while 1:
            time.sleep(1)
            print("{0} requests/min".format(self.requests), flush=True)
            self.requests = 0

    def handle_client(self, client_socket):
        while 1:
            data = client_socket.recv(4096)
            nth_prime = int(data)
            prime = self.find_nth_prime(nth_prime)
            client_socket.send(str(prime).encode())
            self.requests += 1
```

```
    def run_service(self):

        connection = socket.socket()
        connection.bind(("127.0.0.1", self.server_port))

        # put the socket into listening mode
        connection.listen(5)

        while True:
            client_socket, addr = connection.accept()
            Thread(target=self.handle_client, args=(client_socket,), daemon=True).s


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)


if __name__ == "__main__":

    # change the switch interval to 1 second
    sys.setswitchinterval(1)

    server_port = random.randint(10000, 65000)
    server_host = "127.0.0.1"
    server = PrimeService(server_port)

    server_thread = Thread(target=server.run_service, daemon=True)
    server_thread.start()

    monitor_thread = Thread(target=server.monitor_requests_per_thread, daemon=True)
    monitor_thread.start()

    simple_req_thread = Thread(target=run_simple_client, args=(server_host, server_
    simple_req_thread.start()

    time.sleep(3)

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

    time.sleep(1000)
```

```
    time.sleep(1000)
```

Unfortunately, even if we have more than one processor on the machine running our program, the additional CPUs can't lend us a helping hand because of GIL. Were we to write this program in Java, we would not see such an abysmal decline in performance because such a program could still schedule shorter-running threads on idle CPUs. When designing multithreaded Python programs we can't ignore the effects of GIL especially in case of programs that involve long-running tasks.

In the next section, we'll see how the same program fares when retrofitted with the **`multiprocessing`** module.

Back

Single Thread

Next

Multiple Processors

Mark as Completed

Report an Issue

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