Week 2 - Concurrency and Parallelism

> Distributed and Networking Programming - Spring 2025

Your tasks for this lab:

1. Multithreading: write a multi-threaded TCP server that communicates with a given client;
2. Multiprocessing: compute prime numbers using `multiprocessing`.

Task: multithreading

The server should do the following:

1. Accept a new connection from a client;
2. Spawn a new thread to handle the connection;
3. Wait for client to send data;
4. Recalculate the mean for all data from clients that has been already sent;
5. Wait for ready message from the client;
6. Send the mean of all received data from all clients.

Additional requirements:

- The server should stay listening all the time and should not terminate unless a `KeyboardInterrupt` is received;
- The server should be able to handle multiple connections simultaneously;
- The server socket should be marked for address reuse so that the OS would immediately release the bound address after server termination. You can do so by calling the `setsockopt` on the server socket before binding the address as follows:

  ```
  server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
  server_socket.bind((SERVER_IP, SERVER_PORT))
  ```

Client Implementation

The client does the following:

1. Connects to the TCP server;
2. Sends user-provided data (passed as argument, see examples);
3. Waits 5 seconds then sends "ready" message;
4. Receives "mean" data from the server.

Example Run

Single client run:

```
$ python3 server.py 8081
Server: listening on 0.0.0.0:8081
Server: got connection from client ('127.0.0.1', 62000)
Server: received 2 from ('127.0.0.1', 62000), new mean_data 2.0
```

```
Server: waiting for ready message from ('127.0.0.1', 62000)
Server: got ready message from ('127.0.0.1', 62000), sending 2.0
^CServer: stopped

$ python3 client.py 127.0.0.1:8081 2
Client 0: connecting to ('127.0.0.1', 8081)
Client 0: sending data: 2
Client 0: sleeping for 5 seconds
Client 0: sending ready message
Client 0: receving mean data
Client 0: mean_data 2.0
```

Multiple clients run in sequence:

```
$ python3 server.py 8081
Server: listening on 0.0.0.0:8081
Server: got connection from client ('127.0.0.1', 62928)
Server: received 3 from ('127.0.0.1', 62928), new mean_data 3.0
Server: waiting for ready message from ('127.0.0.1', 62928)
Server: got ready message from ('127.0.0.1', 62928), sending 3.0
Server: got connection from client ('127.0.0.1', 62968)
Server: received 5 from ('127.0.0.1', 62968), new mean_data 4.0
Server: waiting for ready message from ('127.0.0.1', 62968)
Server: got ready message from ('127.0.0.1', 62968), sending 4.0
Server: got connection from client ('127.0.0.1', 63024)
Server: received 7 from ('127.0.0.1', 63024), new mean_data 5.0
Server: waiting for ready message from ('127.0.0.1', 63024)
Server: got ready message from ('127.0.0.1', 63024), sending 5.0
^CServer: stopped

$ python3 client.py 127.0.0.1:8081 3 --number 1 && python3 client.py
127.0.0.1:8081 5 --number 2 && python3 client.py 127.0.0.1:8081 7 --number 3
Client 1: connecting to ('127.0.0.1', 8081)
Client 1: sending data: 3
Client 1: sleeping for 5 seconds
Client 1: sending ready message
Client 1: receving mean data
Client 1: mean_data 3.0
Client 2: connecting to ('127.0.0.1', 8081)
Client 2: sending data: 5
Client 2: sleeping for 5 seconds
Client 2: sending ready message
Client 2: receving mean data
Client 2: mean_data 4.0
Client 3: connecting to ('127.0.0.1', 8081)
Client 3: sending data: 7
Client 3: sleeping for 5 seconds
Client 3: sending ready message
Client 3: receving mean data
Client 3: mean_data 5.0
```

Multiple clients run in parallel:

```
$ python3 server.py 8081
```

```
Server: listening on 0.0.0.0:8081
Server: got connection from client ('127.0.0.1', 63948)
Server: got connection from client ('127.0.0.1', 63947)
Server: received 7 from ('127.0.0.1', 63948), new mean_data 7.0
Server: waiting for ready message from ('127.0.0.1', 63948)
Server: got connection from client ('127.0.0.1', 63949)
Server: received 5 from ('127.0.0.1', 63947), new mean_data 6.0
Server: waiting for ready message from ('127.0.0.1', 63947)
Server: received 3 from ('127.0.0.1', 63949), new mean_data 5.0
Server: waiting for ready message from ('127.0.0.1', 63949)
Server: got ready message from ('127.0.0.1', 63947), sending 5.0
Server: got ready message from ('127.0.0.1', 63948), sending 5.0
Server: got ready message from ('127.0.0.1', 63949), sending 5.0
^CServer: stopped

$ python3 client.py 127.0.0.1:8081 3 --number 1 & python3 client.py
127.0.0.1:8081 5 --number 2 & python3 client.py 127.0.0.1:8081 7 --number 3
[1] 4931
[2] 4932
Client 2: connecting to ('127.0.0.1', 8081)
Client 3: connecting to ('127.0.0.1', 8081)
Client 1: connecting to ('127.0.0.1', 8081)
Client 3: sending data: 7
Client 2: sending data: 5
Client 1: sending data: 3
Client 1: sleeping for 5 seconds
Client 2: sleeping for 5 seconds
Client 3: sleeping for 5 seconds
Client 2: sending ready message
Client 1: sending ready message
Client 3: sending ready message
Client 2: receving mean data
Client 1: receving mean data
Client 3: receving mean data
Client 2: mean_data 5.0
Client 3: mean_data 5.0
Client 1: mean_data 5.0
[2]  + 4932 done       python3 client.py 127.0.0.1:8081 5 --number 2
[1]  + 4931 done       python3 client.py 127.0.0.1:8081 3 --number 1
```

Task: multiprocessing

Find all prime numbers amongst first 1,000,000 numbers using multiprocessing. You should provide both single-processed solution and "optimized" one (multiprocessed).

Instructions

1. Write basic single-process solution (the most basic one - bruteforce, **do not use** Sieve of Eratosthenes or some other optimizations) in `nonoptimized.py` file;
2. Modify the script to use multiprocessing to process multiple numbers concurrently and write your solution into `optimized.py` file;
3. Ensure that first solution write primes to `primes_nonoptimized.txt` file, while second one to `primes_optimized.txt` (and also there shouldn't be a new line `\n` at the end of the file, check `primes.txt` for reference);

4. Compare the execution time of the optimized script with the original version (it should decrease);

Checklist and Grading Criteria

Submitted solution should satisfy the requirements listed below. Failing to satisfy an item will result in partial grade deduction or an assignment failure (depending on the severity).

- [ ] Required files are pushed to classrooms repository on time. Other files are not modified.
    - [ ] `task_multithreading/server.py`
    - [ ] `task_multiprocessing/optimized.py`
    - [ ] `task_multiprocessing/nonoptimized.py`
- [ ] Code runs successfully under the latest stable Python interpreter
- [ ] Code only imports dependencies from the Python standard library (no external dependencies allowed)
- [ ] Code does not import `concurrent.futures`
- [ ] Server opens a thread for each connection
- [ ] For multiprocessing, execution time is improved after using `multiprocessing` properly
- [ ] Source code is readable and nicely formatted (according to PEP8)
- [ ] Source code is the author's original work. Both parties will be penalized for detected plagiarism)

The list may not be fully complete. We preserve the right to deduct points for any other non-listed issues.