In this assignment, we will build a simple client-server system, where you use the client to chat with a "math" server. The protocol between the client and server is as follows.

- The server is first started on a known port.
- The client program is started (server IP and port is provided on the commandline).
- The client connects to the server, and then asks the user for input. The user enters a simple arithmetic expression string (e.g., "1 + 2", "5 - 6", "3 * 4"). The user's input is sent to the server via the connected socket.
- The server reads the user's input from the client socket, evaluates the expression, and sends the result back to the client.
- The client should display the server's reply to the user, and prompt the user for the next input, until the user terminates the client program with Ctrl+C.

You are provided with the client (source code). You will write two versions of the server:

- Your server program "server1" will be a single process server that can handle only one client at a time. If a second client tries to chat with the server while one client's session is already in progress, the second client's socket operations should see an error.
- Your server program "server2" will be a multi-process or multi-threaded server that will fork a process for every new client it receives. Multiple clients should be able to simultaneously chat with the server.

At the very minimum, both your servers should be able to handle addition, multiplication, subtraction, and division operations on two integer operands. You may also assume that the two operands are separated by a space. So, some sample test cases are:

- User types: 1 + 2, server replies 3
- User types: 2 * 3, server replies 6
- User types: 4 - 7, server replies -3
- User types: 30 / 10, server replies 3

Note that the actual test cases we will use would be different from the ones shown above: your servers should correctly work with any numbers, not just the ones shown above, as long as they conform to this format. Handling non-integer operands, other arithmetic operations, or operations with more than 2 operands (e.g., "1 + 2 + 3") is purely optional.

Several good tutorials and useful documentation on socket programming and designing servers for concurrent clients (using both fork and multithreading) are available online. Please make use of these resources to learn the intricacies of socket programming on your own during this assignment.

Please write your code for this assignment in C/C++/Python.

The client

The copy of the client source code is provided with this assignment. Below is a sample run of the client. For this example, the client code is first compiled. Then a server is run on port 5000 in another terminal. The client program is then given the server IP (localhost 127.0.0.1 in this case) and port (5000) as command line inputs.

```
$ gcc client.c -o client
$ ./client 127.0.0.1 5000

Connected to server
Please enter the message to the server: 22 + 44
Server replied: 66
Please enter the message to the server: 3 * 4
Server replied: 12
...
...
```

In parallel, here is how the output at the server looks like this (you may choose to print more or less debug information). Note that the server's output shown below is only for illustration, and we will not grade you based on your server's debug output. We will primarily grade you based on whether your server returns correct responses to the clients or not.

```
$ gcc server1.c -o server1
$ ./server1 5000
Connected with client socket number 4
Client socket 4 sent message: 22 + 44
Sending reply: 66
Client socket 4 sent message: 3 * 4
Sending reply: 12
...
…
```

The servers

Part1: Single process server

First, you will write a simple server in a file called "server1.c" or “server1.py”. The server1 program should take the port number from the command line, and start a listening socket on that commandline. Whenever a client request arrives on that socket, the server should accept the connection, read the client's request, and return the result. After replying to one message, the server should then wait to read the next message from the client, as long as the client

wishes to chat. Once the client is terminated (socket read fails), the server should go back to waiting for another client. The server should terminate on Ctrl+C.
Your simple server1 should NOT handle multiple clients concurrently (only one after the other). That is, when server1 is engaged with a client, another client that tries to chat with the same server must see an error message. However, the second client should succeed if the first client has terminated.

Part2: Multi-process server

For this part, your server should behave like any real server. It should be able to handle several clients concurrently. It should work fine as clients come and go. Your server should always keep running (until terminated with Ctrl+C), and should not quit for any other reason. If you are in doubt about any functionality of the server, think of what a real server would do, and use that as a guide for your implementation.

For this part, you will write “server2.c” or “server2.py” to be able to handle multiple clients simultaneously by forking a separate process for each client, or creating a new thread for each client. You should be able to reuse most of the code from server1, but your server2 should compile and run independent of your code in Part 1.

Submission instructions

This is an individual assignment. You must do it yourself. You can look up the web for syntactical help but the logic should be totally yours. If a part of your code is found to be similar to your colleagues, it would be considered plagiarism.

To submit your assignment, create a folder, where the name is your roll number. Place all your files in this folder, then compress the folder as a tar gzipped file or a zip file and submit on Google Classroom.

Your submission folder must contain the following files.

- server1.c and server2.c as described above (*.cpp for C++ and *.py for Python). Please follow the guidelines for filenames strictly.
- An optional make/build script to compile your code. Otherwise simple gcc will be used as shown in the sample output above.
- testcases.txt comprehensively describing the various scenarios you have tested your two servers with. Especially highlight any optional cases (e.g., operations with 3 operands such as "1 + 2 + 3") that you have tested your code with.

Note: Please document your code properly. Since we will read through your code while grading, a cleanly-written and well-documented code will fetch you more marks, in addition to making the grader's job easy.

Testing

We will run the following tests to grade your assignment. It is strongly encouraged that you perform similar tests on your servers as well before submission. Please list which of these test cases you have successfully tested in your testcases.txt submission. In addition, please list any additional tests you may have come up with on your own.

- [TEST1]: For server 1, we will start a single client, connect to server, and test all 4 arithmetic operations (+,-,*,/) with two operands each. We will check that the results returned by the server to the client are correct.
- [TEST2]: For server1, we will start a client, do some math operations (like TEST1), then terminate the client, start a second client, and check that the second client can chat with the server as well.
- [TEST3]: For server 1, we will try to connect a second client when the first one is still connected, and check that its socket operations fail.
- [TEST4]: For server 2, we will check the correctness of arithmetic operations for a single client, as in TEST1.
- [TEST5]: For server2, we will test that multiple clients can simultaneously connect and chat with the server correctly.
- [TEST6]: For server2, we will connect a client, then connect and disconnect a second client. The first client should continue to function correctly.
- [TEST7]: We will test that server2 starts multiple processes for multiple clients.