CMPE476 - Distributed Systems
Project
Due April 21, 23:59

In this project, you **<u>are allowed to use</u>** ChatGPT or a similar large language model (LLM). However, make sure you cite the parts you acquired from LLM properly using inline comments. Also, provide the prompt you used to get that help from LLM. Note that you will not lose points depending on how much help you received from LLM, but I would like to see how good you are also in writing the prompts. Note that LLM systems often give incorrect results, so it is your responsibility to check it.

The programs should be developed using C language (using any version of gcc) on Linux. Note that handling sockets in Windows is quite tricky, so I suggest that you use a virtual Linux machine if you are using Windows (even Mac OSX).

The project is composed of two mandatory parts (Part_A and Part_B) and a bonus part (Part_Bonus).

PART_A: SIMPLE CLIENT-SERVER SYSTEM

The aim of the mandatory part is to get the students to learn the basics of implementing a very simple protocol for a client-server system. You are going to write two separate files, a client and a server. You will start a single copy of the server in one terminal window (in Linux) and several copies of the client (each client in a separate terminal window).

The system works as defined below:

- The server will **<u>not</u>** fork any child processes for each client connecting. The connection attempts will be served directly by the server process, but up to 3 client processes may be kept waiting for the server. If more clients attempt to connect, their attempts should simply be denied.
- The client learns its client id from the command line argument. (Search for the use of argc and argv if you don't know how to do so.) The first message sent by the client to the server is this client id. The server records this client id and prepends all of its output on the screen with this child id (in the form "Message from client #3: …") for clarity.
- Each client process will read an integer from the keyboard, send it to the server, and display the response it receives on the screen. The client process will repeat doing this until a negative value is entered.
- The response sent from the server is simply the square of the number it receives from the child process.
- When the child process reads a negative value from the input, it sends that value to the server so that the server is also aware of the end of communication. The server will not respond to this request. Both the client and the server drop the connection at this stage and the child process terminates.

The output of a sample run should be similar to below. Note that the user can enter input from any client window in the order he/she wishes. Therefore, the requests arrive at the server in different orders.

Server Window

```
$ ./server
Server has started
Waiting for connections
Incoming request from client #1
(client #1) Request=5
(client #1) Request=9
(client #1) Request=-1 dropping
connection
(client #2) Request=3
(client #1) Request=4
(client #1) Request=-5 dropping
connection
```

Client #1 Window

```
$ ./client 1
This is client #1
Enter request (negative to terminate): 5
    Result: 25
Enter request (negative to terminate): 9
    Result: 81
Enter request (negative to terminate): -1
    Will terminate
$ _
```

Client #2 Window

```
$ ./client 2
This is client #2
Enter request (negative to terminate): 3
    Result: 9
Enter request (negative to terminate): 4
    Result: 16
Enter request (negative to terminate): -5
    Will terminate
$ _
```

PART_B: MULTIPLE CLIENTS AND ONE SERVER SYSTEM

This part is a direct extension to the previous. You are going to re-write the previous part as two separate files, a client and a server. You will start a single copy of the server in one terminal window and several copies of the client (each client in a separate terminal window).

The system works as defined below:

- The server will fork a client for each client that connects. (Assume we will test it with at most 10 clients.) The client will be served by the child process of the client, and then the child process will terminate.
- The client learns its client id from the command line argument. (Search for the use of argc and argv if you don't know how to do so.) The first message sent by the client to the server is this client id. The server (actually the child process) records this client id and prepends all of its output on the screen with this child id (in the form "Message from client #3: …") for clarity. Note that, since all child processes are created by a parent running on a specific terminal window, their output will be interleaved on the same window.
- Each client process will read an integer from the keyboard, send it to the server (to the corresponding child process), and display the response it receives on the screen. The client process will repeat doing this until a negative value is entered.
- The response sent from the server is simply the square of the number it receives from the child process.
- When the child process reads a negative value from the input, it sends that value to the server so that the server is also aware of the end of communication. The server will not respond to this request. Both the client and the server (the corresponding child process terminate at this stage).

The output of a sample run should be similar to below. Note that the user can enter input from any client window in the order they wish. Therefore, the requests arrive at the server in different orders. Also, this time we are prepending the server output with the strings "(parent)" and "child #X)" since multiple processes write on the same console.

Server Window

```
$ ./server
(parent) Server has started
(parent) Waiting for connections
(child #1) Child created for incoming request
(child #1) Request=5
(child #1) Request=9
(child #2) Child created for incoming request
(child #2) Request=3
(child #1) Request=4
(child #2) Request=2
(child #1) Request=-1 Will terminate
(child #3) Child created for incoming request
(child #3) Request=4
(child #3) Request=-5 Will terminate
(child #2) Request=-4 Will terminate
```

Client #1 Window

```
$ ./client 1
This is client #1
Enter request (negative to terminate): 5
    Result: 25
Enter request (negative to terminate): 9
    Result: 81
Enter request (negative to terminate): 4
    Result: 16
Enter request (negative to terminate): -1
    Will terminate
$ _
```

Client #2 Window

```
$ ./client 2
This is client #2
Enter request (negative to terminate): 3
    Result: 9
Enter request (negative to terminate): 2
    Result: 4
Enter request (negative to terminate): -4
    Will terminate
$ _
```

Client #3 Window

```
$ ./client 3
This is client #3
Enter request (negative to terminate): 4
    Result: 16
Enter request (negative to terminate): -5
    Will terminate
$ _
```

Note that when a child process terminates, a zombie process remains as long as the parent is alive. (You can check that with the command "**`ps -ef | grep server`**" in another window while the server process is still running.) Zombie processes disappear when their parent terminates. You will receive 5 extra points if you can catch and terminate the extra points while the server process is alive so that they do not accumulate. Search the Internet to find how you can take care of zombie processes.

PART_BONUS: INETD-LIKE SYSTEM (60 POINTS)

inetd is a daemon that captures the ports of several well-known but rarely used services so that the corresponding server processes are created only when there is an incoming request. The use of *inetd* allows the system to host many rarely used services by using minimal resources.

For the bonus part, you will implement two simple server programs: *square* (finds and returns the square of the number sent by the client) and *cube* (finds and returns the cube of the number sent by the client). These server processes will be the simplified version of the server in the mandatory part. These server programs will **not** fork child processes. An incoming client request will be served by the server process and the server will terminate after it serves the client. If another client tries to use the same service, the server will be spawned again by inetd. (Yes, it will be slow and there is too much overhead, but we expect these requests to arrive rarely.) Your *inetd* server should listen to port #5010 for *square* service and port #5020 for *cube* service.

The output of a sample run should be similar to below. Note that the user can enter input from any client window in the order he/she wishes. Therefore, the requests arrive at the server in different orders.

Server Window

```
$ ./inetd
(inetd) inetd has started
(inetd) Waiting for ports 5010 & 5020
(inetd) Connection request to square service
(square) Request=6
(square) Reply sent as 36. Terminating...
(inetd) Connection request to cube service
(cube) Request=2
(cube) Reply sent as 8. Terminating...
(inetd) Connection request to cube service
(cube) Request=3
(cube) Reply sent as 27. Terminating...
(inetd) Connection request to square service
(square) Request=8
(square) Reply sent as 64. Terminating...
```

Client Window

```
$ ./client 5010
Enter request: 6
    Result: 36
$ ./client 5020
Enter request: 2
    Result: 8
$ ./client 5020
Enter request: 3
    Result: 27
$ ./client 5010
Enter request: 8
    Result: 64
$ _
```

Submission

You will submit the whole project as a single ZIP file in Moodle, where each part is in a separate directory named Part_A, Part_B, Part_Bonus. Please stick with the naming convention. For each part, you will provide a separate Makefile, which compiles and links all modules when “make” is called, and removes all binary files when “make clean” is called.

You will also submit a report in the root folder, named report.pdf, which states the LLM prompts you used for each part and any problems you experienced. For example, if a specific LLM prompt caused an incorrect result, state it.