Lab exercises on Socket Programming

Goal of the assignment

In this assignment, we will build a simple client-server system, where you use the client to chat with a dummy 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 are provided on the commandline).
- The client connects to the server, and then asks the user for input. The user types his message on the terminal (e.g., "Hi", "Bye", "How are you"). The user's input is sent to the server via the connected socket.
- The server reads the user's input from the client socket. If the user has typed "Bye" (without the quotes), the server must reply with "Goodbye". For any other message, the server must reply with "OK".
- The client then reads the reply from the server, and checks that it is accurate (either "OK" or "Goodbye").
- If the user had typed "Bye", and the server replied with a "Goodbye" correctly, the client quits. Otherwise, the client asks the user for the next message to send to the server.

You are provided with the client (source code). You will write the server code to communicate with the client. You are encouraged to solve the assignment in C.

The client

Here is a copy of the client source code. 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 (127.0.0.1 in this case, which is a special IP address that always points to the local machine) and port (5000) as commandline inputs. When the user enters messages like Hello or Hi, the server replies with OK. When the user says Bye, the server says Goodbye. The client program will exit after user enters "Bye" and server replies "Goodbye".

```
$ gcc client.c -o client
$ ./client 127.0.0.1 5000
Connected to server
Please enter the message to the server: Hello
Server replied: OK
Please enter the message to the server: Hi
Server replied: OK
Please enter the message to the server: Bye
Server replied: Goodbye
$
```

In another run of the client, we show what happens when the server sends a wrong reply, say, "NO" instead of "OK". Here, the client exits with error message. This behavior should not happen with your server code.

```
$ ./client 127.0.0.1 5000
Connected to server
Please enter the message to the server: Hello
Server replied: NO
ERROR wrong reply from server
```

The server

Now, you will write a simple server in a file called "server.c". The server program should take the port number from the commandline, and start a listening socket on that commandline. Whenever a client request arrives on that socket, the server should accept the connection, and store the client socket. The server must then read from the socket. When the client has sent a message, the server should reply "OK" or "Goodbye" based on what the client has sent. After replying to one message, the server should then wait to read the next message from the client.

Note that your simple server will not handle multiple client concurrently. That is, when the server is engaged with a client, another client that tries to chat with the same server will see an error message. You may verify this during your implementation and testing.

Below is sample output from our version of server (corresponding to the client output shown above). Your server should produce similar output (the exact words printed in the statements may differ). Note that our server is printing out the client socket file descriptor number for debugging.

```
$ gcc server.c -o server
$ ./server 5000
Connected with client socket number 4
Client socket 4 sent message: Hello
Replied to client 4
Client socket 4 sent message: Hi
Replied to client 4
Client socket 4 sent message: Bye
Replied to client 4
Client said bye; finishing
```

Programming Assignment 1: Socket Programming

Goal of the project

In this assignment, we will build a simple client-server system, where you use the client to chat with a dummy "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 arithmatic 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 **three** 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 server that will fork a process for every new client it receives. Multiple clients should be able to simultaneously chat with the server.
- Your server program "server3" will be a single process server that uses the "select" system call to handle multiple clients. Again, much like server2, server3 will also be able to handle multiple clients concurrently.

At the very minimum, all 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 may 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 confirm to this format. Handling non-integer operands, other arithmatic 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 select) 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++. Please talk to me if you have a compelling reason to use any other language.

The client

Here is a copy of the client source code. 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 commandline 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". The server1 program should take the port number from the commandline, 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 client 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 succed if the first client has terminated.

Part2: Multi-process server

Note: For parts 2 and 3 below, 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 part 2, you will write server2.c to be able to handle multiple clients simulatenously by forking a separate process for each client. You should be able to reuse most of the code from server1.c, but your server2.c should compile and run independent of your code in Part 1.

Part 3: Concurrent server with "select"

Now you will write server3.c to handle multiple calls using the "select" system call. The difference from part 2 is that you will not do a fork() to create a new process for every client, but you will handle all the client sockets in a single process. The behavior of the server with respect to handling multiple clients correctly is the same as the specification in part 2 above.

Submission instructions

You must do the project in groups of one or two. If you work with another student, both of you should contribute equally to the assignment (i.e., do not leave one person to write the code alone).

To submit your PA, create a submission folder, where the name is a concatenation of the roll numbers of your team members, separated by underscore ("_"). For example, your folder name could be "15000001_15000002". Place all your files in this folder, then create a tar gzipped file that has all roll numbers in the filename (e.g., "15000001_15000002.tgz") and submit on Moodle. For example, go to the directory with your submission folder, and do "tar -zcvf 15000001_15000002.tgz 15000001_15000002".

Your submission folder must contain the following files.

- server1.c, server2.c, server3.c as described above. **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 three 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 testcases 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 arithmatic 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 arithmatic operations for a single client, as in TEST1.
- [TEST5]: For server2, we will test that multiple clients can simultaneoulsy 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], [TEST8], [TEST9]: Repeats the above three tests for server3 as well.
- [TEST10]: We will test that server2 starts multiple processes for multiple clients, while server3 does not.

Grading

Below is the grading scheme for this assignment. This assignment carries 25 points (scaled to 10% of grade).

- 5 points for code inspection (readability, documentation, correctly following the spec).
- 10 points for the 10 test cases listed above.
- 10 points for written test question.

Note that the grade of the code submission and written test are not independent. For example, if you do badly in the written test, marks allocated for your code will also be penalized, as it is will be assumed that you did not solve the assignment yourself.