**Final project documentation - Client-Server Architecture**

**Neha Ahmed Khan 14793**

This client-server system is a multi-threaded architecture where multiple clients can connect to a server and both are interactive meaning clients can send a set of commands to the server to perform such as arithmetic operations (add/sub/mult/div), run and kill any program, list of processes running for a client, send a message to server and exit. The server can also execute some commands such as request for a list of all clients running and their processes, send a message to all clients or to an individual client.

**Client-side:**

The client side first creates a socket so that 2-way communication occurs between client and the server. The socket has an Internet domain (AF\_INET) so that two processes(client and the server) on any 2 hosts can communicate over the internet and *stream socket* type.

The connect API is used to initiate a 3-way handshake which establishes a connection with the server’s socket. For this client architecture, port number of the server needs to be given that is provided by the server. This port number is converted into network format using htons() and then stored in the sockaddr structure. The IP address of the server is also needed which is converted from dotted notation into network format using gethostbyname() and then saved in the structure.

After connection is established, this client can request for services from the server such as add/sub/mult/div, run or kill a process, list of all processes of that client, send a message to the server and exit (where the client terminates). To perform this, the client is divided into 2 threads: one thread takes input from the client(keyboard) and writes it on the socket and the other thread reads from the socket and writes to the screen of the client. The thread that takes input checks on client-side only if these above mentioned commands are sent, otherwise writes a message on screen alerting of wrong command. This is done so that time is saved when wrong commands are sent to the server. When the exit command is sent, the client sends this to the server where its client handler terminates and then the client terminates too.

In the second thread that reads from the socket the results that the server has performed, they are simply written on the client’s screen. If the server is terminated then the clients get terminated too.

**Server-side:**

Just like in client, first a socket is created with the same domain and socket type as the client (AF\_INET & stream socket). Then the bind() api is used to define ports at the server. The address is set INADDR\_ANY to pick up any address of the server and port is set 0 to assign any port number (wildcard) for the server. The port number is displayed on the server screen to be used by the client. After bind, listen() api is used to check for connection requests and blocks until any request is received. Up to 10 connections can queue up for this server. As soon as any client connects then accept() api is used to accept that connection. A connection sets up and a connection handler which was originally a separate process is now a separate thread which communicates with the client using msgsock.

On the server, I have created 5 threads:

* First thread is for the connection handler that takes input (takeInput( ))
* Second thread is the client handler thread that reads msgsock for client inputs (takeClientInput( ))
* The third one is client handler thread that handles server commands (handleServerInput( ) )
* The 4th (readInput( )) and 5th (writeList( )) threads are created in the conn thread (1st thread) where one simply reads input from the server screen and writes into the pipe that goes to the client handler. The other thread prints the list of all clients by reading from the pipe that is sending data from the client handler where the lists are maintained. The reason to create these 2 threads separately is that both are supposed to be reading from the keyboard/pipe indefinitely and keeping them in one thread created errors for other server input commands to be never executed as one or the other read would block.

The function of the first thread is explained in the last point above where it creates two threads at the conn handler. One of its threads reads input from the server such as print msg to all clients, print message to a specific client and list of all client handlers. These commands are read and written to the pipe that connects conn to the client handler. Necessary checks such as if any other command than print, printclient or list is entered, an error message is displayed for it, if its a printclient command and a PID that does not exist in the client list is entered even then an error message is displayed. At the conn, a new struct (clientlist) is maintained that stores all client handlers PID and msgsock fds as well as their pipe fds that are used when communicating between conn and client handlers.

The second and third thread is of the client handler which is created after a client handler process is created.

The second thread as mentioned above handles client inputs (add/sub/mult/div/run/list/kill/exit) by reading from msgsock and then writing back to it after processing the request. In the arithmetic operations floating point numbers are accepted and division by zero is also handled. On exit command, not only the client exits, but also its client handler. On run command, if exec occurs successfully meaning if the process to run exists, then entry is made of that process in the list. Otherwise, an error message is sent to the client of an “invalid process name entered” and then the entry is not made in the list as well.

The third thread handles server commands which is basically used for communication between client handler and conn where client handler reads from the conn through a pipe. If the command it read is “print” or “printclient” then it writes it to the msgsock. If the command is of “list” then it writes back the list of all client handlers to the conn through a second pipe which is read in the 5th thread that we created.

If the server terminates due to ctrl C, this is handled by SIGINT signal handler where it informs all clients that the server has terminated. If a child process terminates externally, then a SIGCHLD handler is invoked where that specific child’s active status is set to zero from 1 and its end time and elapsed time are also calculated to update the list.

**Limitations in the code:**

* For add sub mult and div operations, the check to see if numbers are entered or alphabets is not applied
* Time is not converted to local time. It will show seconds elapsed since the epoch
* If a client exits, the server commands behave abnormally for e.g. sometimes it will show a complete list, sometimes just for the client that is still running and sometimes not at all. Otherwise it works fine.
* All commands need to be entered in lower case
* Multiple inputs from 1 client not accepted for e.g. if one client enters “add 2 2 ; run gedit ; sub 10 5” this wont work. One client sends one command at a time.