Arif Ayarci

CIS-3207-01

4/13/2022

README Document

**Project 3: Multi-Threaded Echo Server**

**Introduction:**

In this multi-threaded echo server project, a server-client program will be implemented much like a producer-consumer problem. Two .c files will be implemented in this project, one for the server and one for the clients that connect to that server. The server program will create a socket and bind that socket to a server and listen for client connections. The client will create a socket and connect to that server. The server program will accept those incoming connections. The client will be given a scanner for a message input and that message will be sent to the server to be echoed back to the client.

**Design Description:**

* In this multi-threaded echo server project, a client server program will be implemented much like a producer-consumer problem. Two .c files will be implemented in this project, one for the server and one for the clients that connect to that server.
* The multi-threaded echo server will make use of five command line arguments, two of them being optional and three of them being required. The length and port will be optional and if not defined will be defined by default in the program. The number of workers, buffer size, and terminator character will be defined in the command line, else, an error will occur.
* The multi-threaded echo server will make use of three threads and a parsing function that makes up the entire program. There is the main thread, the worker threads, and the log thread. The parsing function will be called at the beginning of the main function followed by the main thread that will be used to call the worker threads that will be used to call the log thread. The threads will be created with the pthread\_create() function and executed one by one with the pthread\_join() function.
* The parsing function, parse\_cmdline(), will make use of argc and argv as calls. The function will make use of flags to specify the variable and strcmp() to identify those flags. The atoi() function will be used in the parser to convert the arguments after the flags to an int as argv is a double char and store them into their respective variables. The variables will be global as so they can be passed into main and every thread.
* The main thread will use a while(1) loop to repeat infinitely. The main thread will be used to accept the incoming clients and create the worker threads. The worker threads will take the accepted client sockets as an argument.
* On the client side, the input will be taken with the fgets() function that takes stdin as a call and accounts for spaces. The messages will be sent to the server with the send() function taking the server socket as a call. If the client message is equal to the terminating character the server will close the socket with the close() function and the client will exit with exit(1). The messages will be compared for this using the strcmp() function and taking the terminating character as a call.
* The worker threads will be used to receive the messages inputted by the clients within a while loop using the recv() function. The server will also make use of the send() function to echo back the message to the client and likewise, the client will also make use of the recv() function to receive that message and for the echo to come in full cycle.
* The client will also make use of a while(1) loop to infinitely ask for a message input until it is terminated with the terminating character. The client will make use of the puts() function to print the echoed messages.
* When the message has been echoed the worker thread will call the log thread that takes the echoed message as a call. The log thread will store the echoed message and the time the message was echoed in a log.txt file. The log thread will make use of the fopen() function to open up “log.txt” for appending. The log thread will also make use of the time(), asctime(), and localtime() functions to print the current calendar time.
* After the log thread has been called, the worker thread will make use of the bzero() function taking the message as a call to clear the memory of the client message.
* When all of the worker threads have been created and joined the thread will make use of a wait condition variable to wait for a client to exit. If a client does exit the worker thread will use a signal condition variable to tell the main thread that a client as exited and a worker thread will be created to account for this. Mutex lock and unlock will be used alongside these condition variables for mutual exclusion.

**Testing, Results, and Debugging Description:**

* In this multi-threaded echo server project, Ubuntu software was used as the project would need to be done in a Linux environment. The program makes use of libraries and functions exclusive to Linux and not available for Windows like <sys/socket.h> and <netinet/in.h>.
* A makefile was also implemented in this multi-threaded echo server project to easily compile both the server file and client file at the same time.
* For testing the server and client files I opened up two terminals, one to run the server and another to run the client. The server file will be executed first and only one server file will be executed. Then in the other terminal the client file will be executed to connect to the server.
* To test working with many clients, I simply executed more client files by opening more terminals and executing the client file in those terminals to connect to the one server.
* To test a single work echo I would simply have the client send a single word to the server to be echoed, to test a sequence of phrases I would simply have the client type a sentence to the server to be echoed. Both of these tests were a success.
* When testing, I noticed that when having one client write multiple messages, those messages would start to overlap each other and the memory for the client message would stay to the next messages. To debug this I used the bzero() function at the end of the worker thread to reset the data for the client message.
* After resetting the message, I ran through another error where the client message would not print to the log file as the memory would reset. To debug this I would use the strcpy() function to copy the client message to a log message before bzero() was called and passed the log message to the log thread.

**Results of Testing with many Clients:**

* When testing with many clients, all of the clients within the number of workers would have their respected messages serviced immediately by the worker threads as it should. The clients that connect after the worker threads would have their messages be put on hold and in wait. As soon as one of the worker thread clients exits, the next client in wait will be executed immediately as it should. Only one message from a client can be processed by the server at a time as it should.
* To test and see if my queues were working, I ran my server with an argument for 2 workers. I then opened three other terminals and ran all of them for the client. Two of the clients would have their respected messages echoes back to them while the third client would have their message be stuck at “client: data sent.”. When exiting one of the worker thread clients, the server would accept the third client in wait and immediately service its respected message resulting in a success. If a fourth client terminal was present, it would remain in wait until another worker thread client exited.

**Discussion and Analysis:**

All in all, the multi-threaded echo server project appears to be doing its job well. It is actually multi-threading in the sense that it is handling multiple clients at the same time. When all the threads are being handled by clients, the rest of the remaining clients are waiting to be handled. When a handled client exits, the next client in line that compiled first is handled immediately. The server can handle echoing single worded messages and even multiple phrased messages. The worker threads nicely call for the log thread to print the contents of the client message plus the time the message was echoed. Though global variables are quite dangerous and were implemented in this project, the parser had no problem updating the parameters and the variables have no trouble being used across main and the three threads of this project. The server nicely handles mutex locks/unlocks and condition variables wait and signal in order for the worker threads and main thread to communicate with each other whenever a client exits the server via the terminator character. Best of all, the echo server can compile error free with the -Wall and -Werror flags included in the makefile.