Name: Brian Jon Stout
Date: 13MAY2017
Current Module: Operating System
Project Name: Relay

Project Goals

The goal of the project is to create two programs using IPC processes so that one acts as a server and multiple version can exist of the other, acting as a client. The server must take a message from STDIN and broadcast it to all version of the client running at the time. Ctrl+D on the server should exit the program cleaning

Considerations:

- CTRL+D should exit the program cleanly
- The client executable should be able to connect to the server from any where on the device

Initial Design:

The program starts by creating a linked list of ints which contains the sockets. Afterwards it creates the two threads, the first for receiving input and the second for accepting connections to the mast socket.

The first thread is a looped fgets call, which only exits on ctrl+D. The pthread function passes a pointer to the list of sockets which is used to call a function that iterates through all the sockets and broadcasts the message written from STDIN by fgets.

The second thread is also passed the pointer to the list of sockets. Its primary job is to add new sockets to the linked list. At the start it initiates the master socket, binds, and sets it to listen. After this it gets into an infinite for loop that looks for sockets to accept.

When the first thread gets completed, the main program closes the second thread with pthread_close() because I have no idea how to look at the connection queues for the socket (which I know exists because of the listen() command) and just accept all the connections in the queue. As a result the accept for loop is infinite and there's no good way to exit out of it that I could find.

When the second thread is forcibly shut down the program runs a destructor for the linked lists and frees all the memory.

The client program is very simple. It sets up a socket that's set up to remote to the stream stored in /tmp/ It sits in an infinite while loop until it receives data from the server, then prints it out. If a socket receives no data from the server (meaning it's not serving anymore) it exists out. The program is a modified version of the examples given in the shared folder for echos.c and echoc.c.

Communication Protocol:

AF_UNIX SOCKET STREAMS

Potential Pitfalls:

The infinite loop of the accept function made it difficult to do anything cleanly.

The memory allocated in the threads caused some hard to debug valgrind errors.

Test Plan:

User Test:

Run the server. Ctrl+D it. Type message in it.

Do the same except with a listener.

Do the same except with multiple listeners.

Do the same except with multiple listeners from different parts of the OS.

Test Cases:

The program works well. There might be some issues with concurrency (probably some issues with concurrency.)

Conclusion:

Turns out using AF_UNIX sockets without having a full understanding of networking functions in unix (sockets, send, receive, bind, listen, etc ,etc.) Was a bad idea. Resources on the subject weren't too great, since they probably rely on the fact that the programmer has experience with network sockets already. There was some functions called select which might have helped with the infinite accept() loop since it polls the sockets for events and reacts accordingly, but I wasn't sure if it could detect a socket attempting to connect and wasn't able to rework other people's code since I didn't understand it completely.

In hindsight, using shared memory space, FIFO, or message queue probably would have been a lot easier, even though most programmer recommend using UNIX sockets for a many to one IPC client-server relationship. Having the listeners just loop through and read-only a binary file containing a struct of the message number and the message itself would have been simple enough, having the server lock the file while it was writing to it.