COSC 3750
Linux Programming, Spring 2023
Homework 7

1 Basic info

This is a first pass at interprocess communication. It is relatively simple. There is almost no "protocol" to go along with this. The program will of course be be written in C.

2 What you will need

You will need the files *socketfun.c* and *socketfun.h*, which are linked in the module. You should additionally need to include *unistd.h* but nothing else (for the socket code). Those files plus the information you already have should be enough.

Do not "include" *socketfun.c* in **your** source code files, only *socketfun.h*. AND the compile line should, for the client, look like

```
${CC} ${CFLAGS} wytalkC.c socketfun.c -o wytalkC
```

Note that there are NO **.h** files on the compile line.

3 What you will create

1. There will be two source code files and a Makefile you will turn in. One source code file will be named **wytalkD.c** and the other **wytalkC.c**.

2. Now **wytalkD.c** will the be the "server" process. In the programming world these usually run in the background and are called *daemons*. If you look that up on Wikipedia you will get all kinds of information and some of it may even be true including the idea that it is an acronym for "disk and execution monitor". In any event, when *wytalkD* is run it will open a socket on **port 51100** and accept a connection when a connection is requested. Once the connection is established, *wytalkD* will receive data from the connection and echo it to the screen, then receive data from the user (at the keyboard) and send that out on the connection. Please note that even though daemons traditionally run as background processes, these will not. If you run them as background processes you cannot easily interact with them through the keyboard. But, this process **must** be started BEFORE *wytalkC*.

3. **wytalkC.c** is the client version. It will take a single command line argument that is the **name** of the **host** that *wytalkD* is running on, then it will request a connection to **port 51100** on that host. Once that connection is established, *wytalkC* will listen to the user (at the keyboard), take his/her input a line at a time (use newlines to determine this) and send it to *wytalkD*. Once a line has been sent, *wytalkC* will receive any data from *wytalkD* (the socket), echo it to the screen, then listen to the user again.

4. The protocol then is reduced to this:

   (a) *wytalkD* opens a port 51100 connection and listens for a client.

   (b) *wytalkC* connects to the *wytalkD* host on port 51100.

   (c) When the connection is made:

      i. *wytalkC* reads from user and sends a line of user input to *wytalkD*,

      ii. *wytalkD* echoes the data from *wytalkC* to the monitor,

      iii. *wytalkD* reads from the user and sends a line of user input to *wytalkC*, and

      iv. *wytalkC* echoes the data from *wytalkD* to the monitor.

   (d) repeat from i) to iv) until:

      i. The user closes the input. If this happens, if any data is unsent (which should not happen), send it and then listen for any reply, which should be echoed to the monitor. Once that is completed, the program exits.

      ii. There is an error from user input or socket input. Output any data (monitor or socket as appropriate), print an error message, and exit.

      iii. The other process closes the socket. Output any data (to the monitor) and exit.

5. All actions should be robust (check for errors/overflow/underflow) and failures should be graceful.

6. Writes will NOT be a character at a time. Always a "buffer" full.

7. Under Linux, once the connections are established, you can use standard file handling functions to read/write on the sockets, read() and write(), or you can use send() and recv(). I insist that you do NOT use *sendto()* or *recvfrom()*.

8. Do NOT **print anything at all** except the messages we are communicating and error messages prior to exiting. **Do not prompt the user or print out any additional text, such "User typed:" or "Remote sent:" or some prompt string. Do NOT print extra blank lines.**

9. **Do NOT read one character at a time from the terminal. Read a line. The *fgets()* function is safe and works very well. Use it.**

10. **DO read 1 character at a time from the socket. It is maybe not efficient BUT it allows you to easily find the newline character that ends the transmission. I do not care what Stackoverflow says.**

11. Get started. This is not a big assignment and the worst of the work is done for you in *socketfun.c*. **But**, I will guarantee that you will **not** make it work correctly the first time you get it to compile. One of the biggest problems is always interfacing with the user and correctly interpreting when the user is doing. This is **never** with a timeout.

12. In case you are wondering, if you do this correctly, you don't even have to have two separate computers. Just run the programs between two different terminal windows on the same machine. But, you had **better** test it between two separate machines as well, I know that I will.

    If some this does not work on your Mac, not my problem. Some have had issues in the past, others not. BUT fixes to make it work on a Mac will probably make it NOT work on Linux. Be warned this is the Linux programming class.

4 What to turn in

There are three files to turn in: **Makefile**, **wytalkC.c**, **wytalkD.c**. DO NOT submit the *socketfun.?* files. Use *tar* to store those three files in an archive named **wytalk.tar** or **wytalk.tgz** (if compressed). Yes, if you have some extra files of your own that must be there, submit those in the archive as well. I am not requiring any pseudocode for this one because I gave you the basic protocol. You probably should do some planning though.