Netlog

CS3411 Spring 2020
Program Five
Due: Sunday Apr 26, 2020, AoE

In this project, we will develop a *Logging Server*. The server will provide a recording of messages sent to it and will be able to return the recorded log file upon a request.

The server

The server should open a socket and listen on an available port. You should always ask for a random port from `bind`. The server program upon starting should print the port number it is using on the standard output. This port number is then manually passed as a command line argument to a client (i.e., telnet program) to establish the connection. Once the connection is established, the server should send the message:

```
Log #:
```

From this point on, the client can send one of two commands, the command *log* followed by a text, or the command *list*. If the command is *log*, the text following the command should be appended to a logfile, known only to the server. If the command is *list*, the server should send the accumulated log file, one block at a time to the client, each time waiting for a newline to send the next block of the log.

Restrictions

The program must follow these restrictions:

1. No stdio calls, fopen, printf etc are permitted. You can use sprintf to format messages into a buffer and use write kernel call to write it. Using perror is acceptable.
2. Server has to be single-threaded. This means it cannot fork/exec etc. Threaded libraries cannot be used.
3. Server MUST use the *select* kernel call.
4. No software can be borrowed from any source except software discussed in class and illustrated on slides and from manual pages directly.

Approaching the problem

Following the steps below will help you get to your target quickly. Your program will get 5 pts if all your submitted programs compile correctly.

1. Create a program called step1.c. Create a server which can obtain a TCP port and print the port number on the stdout. There are plenty of examples for this in the course slides. Do not implement select or accept calls yet (5 pts).
2. Make a copy of step1.c and call it step2.c. Add an *accept* call to the program and test it using the command `telnet <hostname> <port number>`. Once you connect successfully, terminate the connection by killing the server (5 pts).
3. Make a copy of step2.c and call it step3.c. Add a select call which waits for two descriptors, the socket descriptor marked using the *listen* call and the standard input. When there is input on the listened socket, execute an accept. When there is input on stdin, read it and write it back to stdout. This way you will see that your select call is functioning properly by sending some input on the terminal where the server is executed. You should be able to connect to the server from a different host/terminal. Test the server by using the telnet program (10 pts).

4. Make a copy of step3.c and call it step4.c. Keep a vector which tells the status of a given connection. Assume there will not be more than 32 file descriptors open at any time. So, it will be a good idea to identify each connection using the socket descriptor number returned by accept. Initialize the vector to a value which represents *not connected*. Other possible values of *state* would be *connected* and *listing* (no points – interim step).
5. Add the following feature to step4.c. Keep another vector of integers which shows the file position while listing the file to a client (no points – interim step).
6. Make a copy of step4.c and call it step5.c. Now modify the program so that each time an accept is executed, it writes the message `log #:` to the returned descriptor. Test the program using telnet. You will see that the prompt is displayed. You can now add code to mark the corresponding entry in the status vector to *connected* and initialize the position vector's corresponding entry to zero (10 pts).
7. Make a copy of step5.c and call it step6.c. Modify the program so that whenever a user closes the connection, corresponding state vector entry is set to *not connected*. You can test this by using telnet program's control character `ctrl ]` and entering *close* from any telnet connection. This will result in the select call waking up for the corresponding descriptor for input, but the read returns end of file (5 pts).
8. Make a copy of step6.c and call it step7.c Modify the program so that it can work with an arbitrary number of connections. This will require adding each of the *connected* descriptors to the select sets. Test using telnet and multiple terminal windows (10 pts).
9. Make a copy of step7.c and call it step8.c. Implement the command handling. If the message received is *log*, write back *#log: logging*. If it is *list* then write back *#log: listing* and modify the status of the connection to *listing*. If the input is a newline character and the status is listing write back *#log: more*. For all other inputs return the text *#log: Command not recognized*. (15 pts).
10. Make a copy of step8.c and call it step9.c. Add code to open with create and truncate the server log file. Implement the *log* command in full and test using multiple connections and checking the file locally where the server runs (15 pts).
11. Make a copy of step9.c and call it logger.c. Add code to implement the list command. The server should return the log file contents 128 characters at a time expecting a newline character from the client each time. You will need to update the position vector for that entry so that the next block will be correctly sent to the client that is listing the log. The last block may be shorter. When the end of file is reached, the status should be modified to *connected* again (20 pts).

Submission Requirements

Your submission must be written in C.
Use Canvas to submit a tar file named `prog5.tgz` that contains:

1. Your final program source logger.c.
2. A Makefile.
3. Any of the *step* files *step1.c*, *step2.c* etc. If you do not submit any of the steps, you will not be able to get partial credit for it!
4. A README file detailing which steps work.

Your makefile should include "all", and "clean" labels. When I type `make` or `make all` in the directory containing your recovered submission, a binary file named `logger`, as well as the binaries step1, step2, .. ,step9 should be created. Typing `make clean` should remove all the object files and the created binary `logger`.

In order to tar the files, store all your files into the directory project-5, cd into this directory and use the command:

```
tar -czvf prog5.tgz *
```

Ground Rules and Restrictions

This assignment may be provided with updates as the project goes on. Make sure that you check Canvas for announcements watch the class mailing list.
Your program MUST work on colossus or guardian (or other IT supplied remotely accessible lab machine equivalent). Any program which was not tested on these machines will not be graded. Please contact us ASAP if you have trouble with connections to these remote servers, or need advice on how to program on these servers remotely.
You may not borrow or reuse any code from other resources, and all the code must be your own. In addition, the following rules apply:

1. You may discuss the program with others.
2. You may not bring any printed/written or otherwise recorded material into the discussion with you. (You may not show anyone your code; you may not view the code of anyone else. This includes others both enrolled in the course and others not enrolled in the course.)
3. You may generate recorded material during the discussion, but it must be destroyed immediately afterwards. Don't save copies of anything from the discussion.
4. If you are in doubt about the acceptability of any collaboration activity, I expect you to check with me before engaging in the activity.