Assignment 4: Network Programming

Due Dec 5, 2018 by 4:59pm **Points** 10 **Available** after Nov 17, 2018 at 12am

Assignment A4: Network Programming

Due: 4:00 PM on Wednesday December 5th

Introduction

In this assignment, you will implement a network version of the help centre program that you wrote for Assignment 2. You will write a server to which clients can connect and issue commands. You will **not** write a client. The commands supported in A4 will be a simplified version of those from A2.

Your Ports

To avoid port conflicts, please use the same port numbers as you use for the online lab in week 10. You should also make sure to add the following lines to your server code (after creating sock_fd and before you call bind), so that the port will be released as soon as your server process terminates.

```
int on = 1;
int status = setsockopt([sock_fd], SOL_SOCKET, SO_REUSEADDR,
                        (const char *) &on, sizeof(on));
if(status == -1) {
    perror("setsockopt -- REUSEADDR");
}
```

Playing Around

Before detailing your tasks, let's explore the help centre server to get an idea of how it works. On `wolf.teach.cs.toronto.edu` port 8888, I have a help centre server running. To connect to it, type the following at the shell: nc -C wolf.teach.cs.toronto.edu 8888. (You must do this from cdf; only the `cdf` machines are allowed to connect to this server.) This command runs the netcat program to connect to the help centre server and the -C flag tells netcat to send network newlines.

Once you've connected, you'll be asked for your user name. Type a name and hit enter. Next you will be asked your role. If you indicate that you are a TA, then the server will tell you the valid commands for a TA which include stats and next. If you indicate that you are a student, then the server will ask you the course for which you want to ask a question and then tell you that you can use the command stats while you wait. Both TAs and students can leave the system by typing Control-C to close nc.

If you try out both roles, you will see that the results of calling stats are different depending on your role. Students see the list of TAs and who is currently being served. This is the result of the

print_currently_serving function from hcq.c. TAs see the queue of students waiting which is the result of the function print_full_queue. Once a TA begins helping a student (by saying next), that student is sent a message to tell them to go see the TA and then is disconnected from the server. The nc window will remain open even though the connection has been terminated from the server. You can kill the nc process with Control-C.

Try connecting to the server. Open up at least one terminal where you connect as a TA and a few others where you connect as a student. Remember that there may be other classmates also connecting in the two roles. There is only one common queue on the server.

Open yet another terminal window and connect. This time give a name that is longer than 30 characters. Notice that the server disconnects. Another acceptable option would be to write your server so that it truncates the name to 30 characters.

While you still have at least two clients connected, start typing something in client 1, then switch to client 2 and type a command (or response). Notice that the server responds to client 2; it uses `select` to avoid waiting for any particular client, and uses separate buffers for each client so that client text is maintained independently.

Functionality

Your task is to implement the help centre server, as outlined above and further detailed below. You must support the same functionality offered by the sample implementation. When a new client arrives, your server must add them to a data structure of active clients, and ask for and store their name, their role and in the case of a student, their course. Then your server must respond to commands from the client. Try the sample server to see the required behaviour when the user enters valid or invalid information.

Your Tasks

You are required to implement a server that works like the sample server. Although the basic idea is the same as in A2, the functionality is significantly stripped down. We have removed any queue statistics, any time information and the by-course queues. TAs can no longer ask for the next student from a particular course and the courses are no longer configurable.

We have provided starter code for the assignment that is stripped-down version of the A2 solution. We have removed the functions that you don't need for A4 and have changed the three larger data-structures (the array of courses, the queue of students and the list of TAs) to be global variables. We didn't change the signatures of the functions in hcq.h in light of this, but you are welcome to do so if you wish.

Step 1:

Modify the `print_full_queue` and `print_currently_serving` functions to return *dynamically allocated* strings. This means changing the signatures of these functions and changing their code. These commands require a string to be created based on the nodes in a linked list. However, prior to looping through the list, you don't know how much memory to allocate for the string that you will build. If you are concatenating strings together, then one approach is to loop through the list twice: once to add up all of the string lengths, and again to copy those strings into a buffer of sufficient size. You may find `asprintf` helpful. Read the man pages to see what header files and constants you need to use this function. Warning: code that compiled just fine on my mac did not compile on teach.cs without adding extra header lines for asprintf.

Another acceptable approach is to set a maximum size for the output buffer and truncate the output of these functions if they will exceed this maximum. The import thing is that you not overflow the memory and write to memory that you have not allocated. If you chose this second approach, use 1024 for your maximum output buffer size.

When the user types a command in the client, it will be sent to the server. The server will call the appropriate function, and will send a message back to the client to be printed on stdout to the user. The functions in hcq.c should **never** print anything to standard out. Strings sent back to the client should end with a network newline.

Because you are writing a server that can be accessed by malicious users across the internet, it is a requirement of the assignment that **your code never overwrite the end of a character array.** Be careful to truncate or refuse names (or commands) that are longer than 30 characters.

Although you do not need to submit it for marking, it is a wise idea to check your changes to `hcq.c` by modifying the `helpcentre.c` starter code so that it works again after you have made the required changes to your functions. For example, instead of

```
if (strcmp(cmd_argv[0], "print_currently_serving") == 0 && cmd_argc == 1) {
       print_currently_serving(ta_list);
}
```

you might have something like

```
if (strcmp(cmd_argv[0], "print_currently_serving") == 0 && cmd_argc == 1) {
    buf = print_currently_serving(ta_list);
    printf("%s",buf);
    free(buf);
 }
```

Advice: Be sure that your changes to `hcq.c` work properly before you move on to the next step!

Step 2: Build the server so that it works with one client

Add the socket code so that you can have one client connecting to the server. Take advantage of the example code you have been given. You are welcome to use without attribution any code from PCRS videos or lecture or labs.

Partial reads

Calling `read` on a socket is not guaranteed to return a full line typed in by the client, so you will need to keep track of how much is read each time and check to see if you have received a newline which indicates the end of a command. (Section 61.1 in Kerrisk describes this in detail.) Lab 10 should also be helpful here.

Advice: It will be helpful to think of a client as having a *state* (i.e. waiting for the name, waiting for the role, ...) in order for your server to interpret the information the client is sending.

Step 3: Support multiple clients using `select`

The server must never block waiting for input from a particular client or the listening socket. After all, it can't know whether a client will talk next or whether a new client will connect. And, in the former case, it can't know **which** client will talk next. This means that you must use `select` rather than blocking on one file descriptor. There will be a deduction if your code could block on a read or accept call.

Note that you will need some kind of data structure to keep track of clients, including a buffer to hold partial reads. Any per-client information should be stored in the element in the data structure that represents a single client.

Step 4 (Step 0): Makefile

This should really be Step 0, since you will save yourself time if you create a Makefile right from the start and use it as you are developing your solution.

Create a Makefile that compiles a program called hcq_server. It must use the gcc flags `-std=c99` and `-Wall`. In addition to building your code, your Makefile must permit choosing a port at compile-time. To do this, first add a `#define` to your program to define the port number on which the server will expect connections (this is the port based on your student number):

```
#ifndef PORT
  #define PORT <x>
#endif
```

Then, in your Makefile, include the following code, where y should be set to your student number port plus 1:

```
PORT=<y>
CFLAGS= -DPORT=\$(PORT) -g -Wall
```

Now, if you type `make PORT=53456` the program will be compiled with `PORT` defined as `53456`. If you type just `make`, `PORT` will be set to `y` as defined in the makefile. Finally, if you use `gcc` directly and do not use `-D` to supply a port number, it will still have the `x` value from your source code file. This method of setting a port value will make it possible for us to test your code by compiling with our desired port number. (It's also useful for you to know how to use `-D` to define macros at commandline.)

Testing

Since you're not writing a client program, the `nc` tool mentioned above can be used to connect clients to your server. Your server is required to work in noncanonical mode (stty -icanon).

To use nc, type `nc -C hostname yyyyy`, where `hostname` is the full name of the machine on which your server is running, and `yyyyy` is the port on which your server is listening. If you aren't sure which machine your server is running on, you can run `hostname -f` to find out. If you are sure that the server and client are both on the same server, you can use localhost in place of the fully specified host name.

To test if your partial reads work correctly, you can set the terminal to send characters to the socket as soon as you type it, by running stty -icanon. When you want to go back to the normal line buffered mode, type stty icanon at the shell to return the terminal back to canonical mode. If you don't do this, programs will work in unexpected ways. (For example, do a stty -icanon to get back to noncanonical mode, then type cat. You'll notice that instead of waiting for full lines, each character is echoed by cat as soon as you type it.) When you are using non-canonical mode, characters get sent immediately to the server, so it is more apparent that the server is waiting to see a newline character before it processes the input.

What to submit

Commit all the files required to build your executable. That includes .h files, .c files, and your Makefile. Make sure your code compiles on teach.cs before your final submission.