Operating Systems, Assignment 5

This assignment includes a programming problem in C. As always, be sure your code compiles and runs on the EOS Linux machines before you submit. Also, remember that your work needs to be done individually and that your programs need to be commented and consistently indented. You **must** document your sources and, if you use sources, you must still write at least half your code yourself from scratch.

1. (40 pts) Teachers of North Carolina State University are having a busy schedule in this Thanksgiving month and their schedules are frequently overlapping. That's why In this assignment we are going to implement a multi threaded server/client Class Room booking system in C that uses TCP/IP socket mechanism for communicating. To handle synchronization you have to integrate appropriate code (i.e POSIX monitor).

   For this program it is sufficient to write a server program that will be handling requests **classRoomScheduler.c**. No need for implementing client side program (Teachers), we are going to use a general purpose program for network communication , `nc`. Note that in this assignment we are referring Teacher as the client.

   Once you get your server program running it will keep on handling incoming requests from the Teachers (Clients). For connecting to the server the Teacher will need the hostname (which is localhost in our case) and a port number (54321 an arbitrary value defined in the static section of the code provided in the starter code, but students will be assigned separate port number which is mentioned later in this document).

   Our solution is pretty straight forward. There will be a number of classrooms initialized in the beginning (say 5) with some specific names (i.e ROOM1201). A Teacher can book a class room if it is available. When he is done he can leave the room making it available for another Teacher. If a class room is taken by a Teacher and another Teacher wants it, he has to wait unfortunately until the room is vacant again.

   While a Teacher is connected to the system, the server will repeatedly prompt for a command using the prompt "`>` ". We are simply echoing whatever the Teacher types in the skeleton code. However in the final version of it you need to implement some specific commands that corresponds to the following actions:

   - `book` *Class Room*
     Wait until it's possible to book a room with the given name, if it is booked, remove it from the listed available rooms. After booking a room a proper message feedback should be given to the Teacher (i.e Class Room Booked!). If an invalid room name is given proper feedback should be also given(i.e Class Room does not exist).
   - `free` *Class Room*
     Takes a Class Room Name as argument and vacates the class room with the given name (i.e., add it to the set of available room). If any other Teacher was waiting to book this class room, this should wake up one of them up so they can book it. If invalid room name is given proper feedback should be given to the Teacher.
   - `available`
     Lists out the available rooms.
   - `reserved`
     Lists out the class rooms that are already reserved.
   - `quit`
     This is a request for the server to terminate the connection with this client. This behavior has already been implemented for you.

Client/Server Interaction

Client/server communication is all done in text. We're using fdopen() to create a FILE pointer from the socket file descriptor. This lets us send and receive messages the same way we would write and read files using the C standard I/O library. Of course, we could just read and write directly to the socket file descriptor, but using the C I/O functions should make sending and receiving text a little bit easier.

The partial server implementation just repeatedly prompts the client for commands and terminates the connection when the client enters "quit". You will extend the server to handle the commands described above. There's a sample execution below to help demonstrate how the server is expected to respond to these commands.

Multi-Threading and Synchronization

Right now, the server uses just the main thread to accept new client connections and to communicate with the client. It can only interact with one client at a time. You're going to fix this by making it a multi-threaded server. Each time a client connects, you will create a new thread to handle interaction with that client, using the pthread argument-passing mechanism to give the new thread the socket associated with that client's connection.

With each client having its own thread on the server, there could be race conditions when threads try to access any state stored in the server (e.g., the set of class rooms that are currently booked). You'll need to add synchronization support to your server to prevent potential race conditions. You can use POSIX semaphores or POSIX monitors for this (I suggest monitors). That way, if two clients enter a command at the same time, the server will make sure their threads can't access shared server state at the same time.

The commands described above require that the server makes a Teacher wait if they try to book a class room that's already occupied. So, in your implementation of the book command, you'll check to see if the requested class room is currently in the set of reserved rooms. If it is, the server will force the thread associated with that Teacher wait until the given class room is vacated by that teacher (who booked it). Whenever an room is vacated, you server will remove it from the set of reserved rooms and wake a thread that was waiting to book that room (if such a thread exists).

Detaching Threads

Previously, we've always had the main thread join with the threads it created. Here, we don't need to do that. The main thread can just forget about a thread once it has created it. Each new thread can communicate with its client and then terminate when it's done.

For this to work properly, after creating a thread, the server needs to detach it using `pthread_detach()`. This tells pthreads to free memory for the given thread as soon as it terminates, rather than keeping it around for the benefit of another thread that's expected to join with it.

Port Numbers

Since we may be developing on multi-user systems, we need to make sure we all use different port numbers. That way, a client written by one student won't accidentally try to connect to a server being written by another student. To help prevent this, I've assigned each student their own port number. Visit the following URL to find out what port number your server is supposed to use.

`people.engr.ncsu.edu/dbsturgi/class/info/246/`

Sample Execution

You should be able to run your server with any number of concurrent clients, each handled by its own thread in the server. Below, we're looking at three terminal sessions, with the server running in the left-hand column and clients running in the other two columns (be sure to use our own port number, instead of 54321). I've interleaved the input/output to illustrate the order of interaction with each client, and I've added some comments to explain what's going on. If you try this out on your own server, don't enter the comments, since the server doesn't know what to do with them.

```
$ ./classRoomScheduler

                    # Run one client and book a
                    # couple of rooms.
                    $ nc localhost 28123
                    > book ROOM1201
                    >Class room booked!
                    > book ROOM1202
                    >Class room booked!

                                                  # Run another client
                                                  $ nc localhost 28123
                                                  # See what items are available
                                                  > available
                                                  ROOM1203
                                                  ROOM1204
                                                  ROOM1205

                                                  # book another room.
                                                  > book ROOM1203
                                                  >Class room booked!

                                                  # Client will wait if it tries
                                                  # to book a room that's
                                                  # already reserved.
                                                  > book ROOM1201
                                                  (Thread waiting)

                    # Now, three rooms are booked
                    > reserved
                    ROOM1201
                    ROOM1202
                    ROOM1203

                    # Vacating this room
                    # Will let the other client
                    # continue
                    > free ROOM1201


                                                  >Class room booked!
                                                  > book ROOM1204
                                                  >Class room booked!
```

```
# This will block this client.
> book ROOM1204
(Thread waiting)
                                    # Until we vacate this room
                                    > free ROOM1204

>Class room booked!
> quit


                                    > quit
```

Submitting your Work

When you're done, submit your `classRoomScheduler.c` source file using Wolfware Classic.