Operating Systems, Assignment 4

This assignment includes a couple of programming problems and a simple exercise to get you to activate an account you'll need for the next homework assignment. As always, be sure your code compiles and runs on the EOS Linux machines before you submit. Also, remember that your programs need to be commented and consistently indented, and that you **must** document your sources and write at least half your code yourself from scratch.

1. (5 pts) Arc account activation.

   On our next homework assignment, we'll need to use a system called ARC to write programs that run on both the general-purpose CPU and the GPU. You'll need to follow some instructions in order to activate an account on this system. I have these instructions written up, but I want to check with the ARC system administrator to mak sure things still work the way I think they do. Once I do, I'll replace this with actual instructions for what you have to do.

2. (40 pts) This problem is intended to give you some practice thinking about and implementing monitors. You are going to create a monitor called pairMonitor, using POSIX mutex and condition variables.

   This monitor will manage threads that want to enter and leave a room. There's a function they call when they want to enter, and another they call when they're ready to leave. To make this an interesting synchronization problem, let's require that threads can only enter in pairs. So, if some thread, Bill, wants to enter, he has to wait for another thread, Mary, to enter before he can go in. After threads enter, they can stay in the room as long as they want, but they have to leave with the same partner they entered with. For example if Bill calls leave(), he will wait in the leave function until his partner, Mary, also calls leave(). Then, they can both return from the leave() function at the same time. Likewise, if Mary had called leave() before Bill, she should wait in the leave() function until Bill also called the leave() function.

   Threads get paired up with a partner when they enter(). If a thread (e.g., Bill) calls enter(), and there's already another thread, Ted waiting in the function, then Bill and Ted are partners until the leave the room. If Bill calls enter() first and there's no other thread waiting to enter, then he'll have to wait until some other thread chooses to enter.

   The room has limited capacity. This will make it easier to implement the data structure that keeps up with which threads are partners while they're in the room, but it will make the synchronization part a little more interesting. If the room full, threads will have to wait in the enter() function until some other threads leave (and, of course, they'll also have to wait for a partner).

   Partnership only lasts until threads leave the room. After that, they could get partnered with a different thread the next time they enter.

   I'm giving you a partial implementation with this assignment. You'll need to add code, and you can even change parts of the code I'm giving you, if there's another way you'd like to do things. Be sure you don't change the printouts, since we'll use those to see if your monitor looks like it's doing the right thing.

   You will need to complete the implementation of the following functions.

   - `void initPairMonitor( int capacity )`

     This function initializes the state of the monitor. It is called once at the start of program execution. Inside this function, you can allocate any state, condition variables, etc that you need to create your monitor and keep up with what's going on inside it. I've implemented a little bit of this function for you. You can see that I have a partial implementation of a data structure for keeping up with the names of all the threads that have entered, and who's partnered with whom.

Note that if you make condition variables in this function, you can't initialize them with the `PTHREAD_COND_INITIALIZER` constant. That only works for static variables.

- `void destroyPairMonitor()`

  This function frees any resources allocated by the monitor. This could include freeing dynamically allocated memory if needed or destroying condition variables and mutexes.

- `bool enter( const char *name )`

  This is the function threads call to enter the room. Before it returns, the function needs to make sure there's enough capacity for them to enter the room, and it needs to choose a partner thread for them to enter with.

  I've included a little bit of code to print the pair of threads that enter together, along with the time when they enter. These messages will help us see if your monitor is working correctly.

  This function returns true if the caller is able to successfully enter the room. At the end of execution, we may need to force threads out of this function, even if they never got a chance to enter the room with a partner. The terminate() function below takes care of this, and a return of false from enter() indicates that you didn't really get to enter the room (so, you shouldn't try to leave() later on).

- `void leave( const char *name )`

  Threads call this function when they want to leave the room. As described above, they have to wait for their partner to also call leave(, then they can both return at the same time. You can assume the threads will always tell you their correct, unique name when they leave (you don't have to worry about threads lying to you about their name when they leave).

- `void terminate()`

  When we're ready to terminate our program, we need to worry about the possibility that there could be some threads still in the enter() function, waiting for the population in the room to drop below the capacity or waiting for a partner (this one's the more difficult problem to deal with). If all the other threads terminate, a thread in enter() might be stuck there forever, never getting the partner they're waiting for.

  If a thread successfully got a partner before terminate() was called, then enter() should return successfully and they should be allowed to stay in the room as long as they want before calling leave() (kind of like lingering in a restaurant for a little bit after it closes. Other people can't enter, but you can stay for a little bit longer to finish up your dinner).

  This function helps with that. Calling terminate() should force all threads waiting in enter() to return, even if they never got a partner. The return value of enter() will tell them they didn't successfully enter and it's time for them to terminate so we can exit the program. The starter code has a variable, running, that you can use to indicate that terminate has been called, but you'll still need to add some more code to wake up threads that might be blocked inside enter().

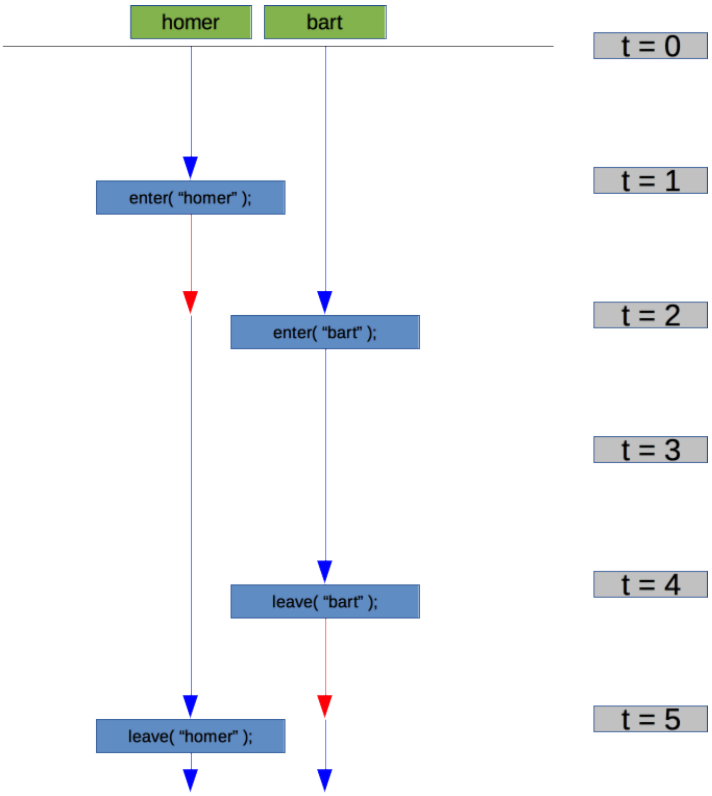
Driver Programs

I'm providing a few driver programs to help you develop and test your monitor. You should be able to compile your monitor with one of these drivers using a command like the following (replacing the driver1 part with whatever driver you want to try):

```
gcc -Wall -std=c99 -D_XOPEN_SOURCE=500 pairMonitor.c driver1.c -o driver1 -lpthread
```

The driver1.c program is designed to make sure your monitor lets threads in in pairs. It uses a room with a capacity for just one pair of threads. As illustrated below, it creates two threads, homer and bart. After a second, the homer calls enter(). He doesn't have a partner so he has to wait in the

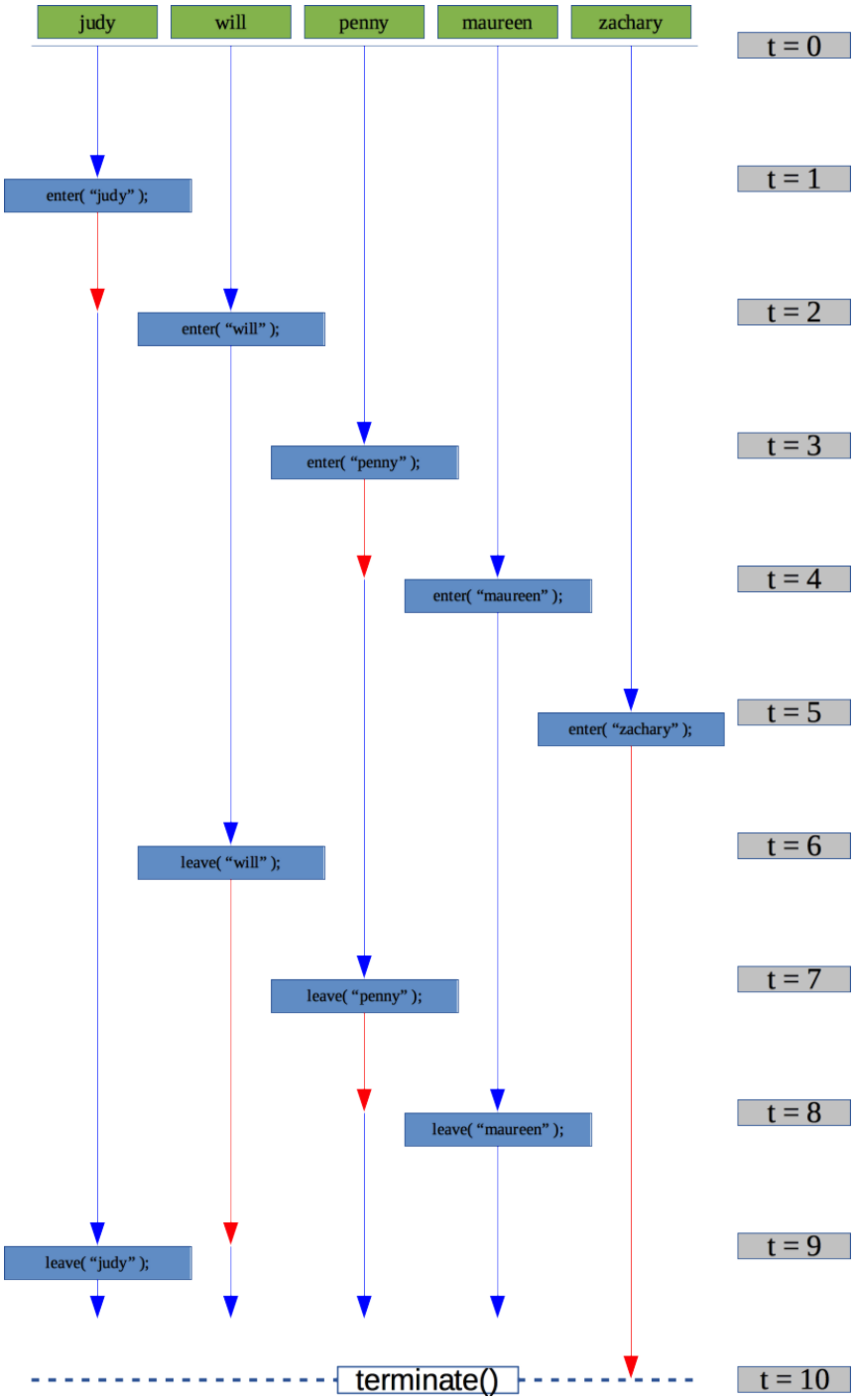
monitor (indicated by the red arrow) until bart also calls enter(). Then, they can both wait a little while before calling leave(). Bart calls it after 2 seconds, but homer doesn't call it until a second later, so at about 5 seconds into the program execution, they both return from leave().



If you compile and run this driver, you should get output like the following (maybe with some very small variation in the timing). The printouts I included in the starter show you that bart and homer entered about 2 seconds after the start of execution, and they left about 5 sections from the start of execution.

```
$ ./driver1
Enter: homer bart (2.001)
Leave: homer bart (5.001)
```

The driver2.c program uses a room of capacity 2 (so two pairs of threads can be in the room at the same time). The first two are judy and will, so they get paired up. Then, penny and maureen are the next two. The zachary thread can't enter, since he doesn't have a partner and the room is already at capacity. Will is the first to try to leave, but he can't leave until judy also calls leave() 3 seconds later. The penny thread leaves earlier, but that's not the thread will is partnered with, so he has to wait for penny. After judy, will, penny and maureen have exited, zachary is still stuck in the enter function. That's what the terminate() function is for. The main() function calls terminate() before trying to join with all the threads. This should force eenter() to return false to zachary, telling him he didn't get to enter and he should terminate.



If you run this driver with your monitor, you should get output like the following (maybe with some very small variation in the timing).

```
$ ./driver2
Enter: judy will (2.001)
Enter: penny maureen (4.000)
Leave: penny maureen (8.001)
```

```
Leave: judy will (9.001)
```

The driver3.c program, creates 10 threads that repeatedly try to enter() and leave() waiting for a very short time in between. The room only has capacity for 3 pairs, so threads will sometimes have to wait in enter(), either because the room is full or just to wait for a partner.

The output from this driver depends on the timing of the execution, so it can vary each time you run it. With a working monitor, you should get output that looks something like:

```
$ ./driver3
Enter: Madeleine Elouise (0.001)
Enter: Sharon Trish (0.001)
Enter: Mohamed Jessica (0.001)
Leave: Madeleine Elouise (0.002)
Enter: Beverly Eleanore (0.002)
Leave: Sharon Trish (0.002)
Enter: Renate Madeleine (0.002)
Leave: Mohamed Jessica (0.002)
Enter: Elouise Judy (0.002)
Leave: Beverly Eleanore (0.003)

... lots more output ...
```

Be sure your program doesn't deadlock when you run it with this driver. Run as follows, I got around 54,500 output lines on a multi-core EOS Linux machine (with some variation from run to run).

```
$ ./driver3 | wc
  54442  217771 1654784
```

You can also try checking how many times each thread gets to enter. Madeleine has short waiting times, so she gets to enter a lot, but Eleanore has longer wait times, so she doesn't show up as much in the output:

```
$ ./driver3 | grep Madeleine | wc
  14239   56956  461933
$ ./driver3 | grep Eleanore | wc
   9278   37112  293120
```

Monitor Implementation

I'm providing the header file, pairMonitor.h, that describes the monitor interface. I'm also giving you a partial implementation containing a struct that you can use to keep up with which threads are partnered while they're in t he room. You'll need to add some code to get this implementation working, and, as noted above, you can change the representation if you want.

You'll need to create a mutex for controlling entry into your monitor. You can use (static) global variables for this and other parts of your monitor's state.

You can use `pthread_cond_broadcast()` for this problem if you want. This is kind of like the Java approach to creating a monitor. When something of interest happens, you can wake up all the threads in the monitor and let them decide if they can make progress. Or, if you want up to 6 points of **extra credit**, solve the problem without broadcast. You can use condition variables more surgically, having different condition variables for the different things a thread could be waiting for and using `pthread_cond_signal()` to wake up a particular thread that can now make progress.

Submitting your Work

When you're done, submit just your monitor implementation, **pairMonitor.c**. You shouldn't need to change the header or driver files, so you don't need to submit copies of these.

3. (40 pts) We're going to implement a multi-threaded Unix client/server program in C using TCP/IP sockets for communication. The program will let clients perform basic 32-bit integer math operations, storing the results in a collection of 26 variables named `a`, `b` ... `z`.

   I'm providing you a skeleton of the server to help get you started, `calcServer.c`. You'll complete the implementation of this server, adding support for multi-threading and synchronization, and building some representation for the variables and their values. You won't need to write a client; for that, we're going to use a general-purpose program for network communication, `nc`.

   Once started, your server will run perpetually, accepting connections from clients until the user kills it with ctrl-C. The program we're using as a client, `nc`, will take the server's hostname and port number as command-line arguments. After connecting to the server, nc will just echo to the screen any text sent by the server and send any text the user types to the server.

   While a client is connected, the server will repeatedly prompt for a command using the prompt "`cmd>` ". In the starter code, the server just echoes each command back to the client, but you're going to modify it to support the following commands. These commands are deliberately simple, making the server easy-ish to code, but making the program a little more difficult to use. For example, if you want to multiply the value of a variable by 3, you have to put the value 3 into some other variable, then multiply by that. There's an extra credit option below to implement a slightly more general syntax for the server's input language.

   - `set` *var value*
     This tells the sever to set the given variable to the given integer value. Here, *var* must be a single lower-case letter (the name of a variable).
   - `print` *var*
     This prints out an output line to the client, giving the current value of the given variable.
   - `add` *var1 var2*
     This adds the value of variables *var1* and *var2*, storing the result in *var1*.
   - `subtract` *var1 var2*
     This subtracts the value of variable *var2* from *var1*, storing the result in *var1*.
   - `multiply` *var1 var2*
     This multiplies the value of variable *var1* and *var2*, storing the result in *var1*.
   - `divide` *var1 var2*
     This divides the value of variable *var1* by *var2*, storing the result in *var1*. We're doing integer math, so division should truncate, like it does in C or Java. If the user tries to divide by zero, your program should leave the variable values unchanged and print (to the client) a line with the message "Invalid command".
   - `quit`
     This is a request for the server to terminate the connection with this client. This behavior has already been implemented for you.

   Variables all start out with the value of zero. You're expected to prevent divide-by-zero, but you don't have to prevent overflow (for example, if the user multiplies to big numbers, the result may not be mathematically correct, if it doesn't fit in a signed, 32-bit integer).

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 values of all the variables). You'll need to add synchronization support to your server to prevent potential race conditions. This is a great opportunity to use multiple locks (kind of like dining philosophers). Make a POSIX semaphore for each variable. Then, when you're about to access or modify the values of one or more variables, acquire()/wait() on the semaphore first and release()/post() on it when you're done. This way, two clients that try to access different variables could possibly have their commands execute at the same time. If two clients try to access the same variable or variables, locking them first will force the clients to take turns.

This is also a great opportunity to think about the possibility of deadlock. Depending on how you write your code, it might be easy for two threads in the server to try to lock the same two variable, but in different orders. For example, running the command "add a b" on one client and running "add b a" on a different client at about the same time could leave the server in a deadlock. We could easily fix this by applying the no-circular-wait deadlock prevention technique. No matter what command the client gives you, make sure all the threads in your server lock variables in a standard order. Then, your server should be deadlock free.

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.

Buffer Overflow

In its current state, the server is vulnerable to buffer overflow where it reads commands from the client. You'll need to fix this vulnerability in the existing code and make sure you don't have potential buffer overflows in any new code you add. Your server only needs to be able to handle the commands listed

above. For inputs that are too long, you can do whatever you want, print an error message, ignore the command, terminate the client connection, whatever. No matter what a client sends it, it shouldn't permit a buffer overflow, since that could let an attacker get unintended behavior out of the server.

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. For this assignment, you only need one port number, so just use the first of the two on this page.

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

Sad Truths about Sockets on EOS Linux Hosts

Since we're using TCP/IP for communication, we can finally run our client and server programs on different hosts. You should be able to run your server on one host and then start nc on any other internet-connected system in the world, giving it the server's hostname and your port number on the command line. However, if you try this on a typical university Linux machine, you'll probably be disappointed. These systems have a software firewall that blocks most IP traffic. As a result, if you want to develop on a university system, you will still need to run the client and server on the same host. If you have your own Linux machine, you should be able to run your server on it and connect from anywhere (depending on how your network is set up).

If you'd like to use a machine in the Virtual Computing Lab (VCL), you should be able to run commands as the administrator, so you can disable the software firewall. I had success with a reservation for a CentOS 7 Base (64 bit VM) system. After logging in, uploading and building my server, I ran the following command to permit TCP connections via my port. Then, if I ran a server on the virtual machine, I was able to connect to it, even from off campus.

sudo /sbin/iptables -I INPUT 1 -p tcp --dport *my-port-number* -j ACCEPT

Also, if your server crashes, you may temporarily be unable to bind to your assigned port number, with an error message "Can't bind socket." Just wait a minute or two and you should be able to run the server again. I added some code to try to prevent this, so maybe you won't see it.

Implementation

You will complete your server by extending the file, `calcServer.c` from the starter. You'll need to compile as follows:

```
gcc -Wall -g -std=gnu99 -o calcServer calcServer.c -lpthread
```

Extra Credit

If you'd like 6 points of **extra credit**, implement your server so it can handle a more flexible input syntax. In addition to the format of the standard commands above, you should be able to handle commands that accept either a variable or a value in certain places:

4. `set` *var1 var2*
   This form of the set command will let you copy a value from one variable (var2) to another (var1). For example:

   ```
   set x y
   ```

5. `add` *var value*
   `subtract` *var value*
   `multiply` *var value*
   `divide` *var value*
   This form of the four arithmetic command lets you give a second parameter that's just an integer value. This let's you do some math operations without having to copy every value to a variable first. For example:

   ```
   add a 30
   divide b 3
   ```

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 28123). 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.

When I run the following on a Linux machine, nc doesn't seem to exit after I type in the quit command. The server closes the client connection, but nc continues running. I'll try to see if there's a good way to fix this. I don't have the same problem when I run the client on my osX laptop, but it's using a different version of nc. For now, you can press ctrl-C in the client window, after you enter the quit command, or you can terminate the cleint with ctrl-D (sending the EOF condition). This seems to close the client connection and get nc to terminate.

```
$ ./calcServer
```

```
# Connect as a client and modify
# some variables.
$ nc localhost 28123
cmd> set x 25
cmd> print x
25
cmd> set y 36
```

```
# Make the server talk to two clients
# at once, see we can see the variable
# values made by the other client.
cmd> add z x
cmd> add z y
cmd> print z
61
```

```
                    # Show off some more of the
                    # math commands.
                    cmd> set a 3
                    cmd> set b 5
                    cmd> subtract z b
                    cmd> print z
                    56
                    cmd> multiply b a
                    cmd> print b
                    15

                                        # One more for the second client.
                                        cmd> divide z b
                                        cmd> print z
                                        3

                    # quit the first client.
                    quit

                                        # quit the other client
                                        quit

# Kill the server with ctrl-C
^C
```

Submitting your Work

When you're done, submit your **calcServer.c** source file using the **assignment_4** assignment on Moodle.