Background

In class, we learned a bit about mutexes and condition variables. I have provided a small program for you which is broken, and you will use these things to fix it.

Starting point

Get the broken program and compile it like so:

```
$ mkdir lab8
$ cd lab8
$ cp ~jfb42/public/lab8.c .
$ gcc -o lab8 -Wall -Werror -lpthread lab8.c
```

Now try running it several times. Sometimes the output is right, but most of the time it’s just... weird. For example, this is what I got:

```
$ ./lab8
this is the correct order for output!
$ ./lab8
the ut!this order for is rect
$ ./lab8
correct the ut!order this
```

What the program does

The program wants to print the following message:

```
this is the correct order for output!
```

(It does this in a very silly and inefficient way but just roll with it, okay?) It spawns 7 threads, one for each word. Then the threads all write to the output_buffer variable. Finally, it waits for the threads to complete, and prints the contents of the buffer.

But currently, it uses multiple unsynchronized threads. It just prints confusing garbage. (Try running it.)

There are two problems you must solve:

- The threads are interrupting each other while writing to the output buffer.
- The threads are running out of order.
Making the threads share the buffer nicely

The first and simpler problem to solve is making the threads synchronize their access to the output buffer. You’ll change the copy_message function to solve this.

There is a mutex, output_buffer_mutex, which you can use to synchronize access. Remember the names of the pthread functions we use to lock and unlock mutexes?

You only have to add a couple lines here. When done correctly, the program will now print all seven words, but in a random order. What you’ve done is made the threads take turns writing to the buffer, but now they’re just running in an arbitrary order.

Making the threads run in the right order

Sharing a resource is a good fit for a mutex. But making the threads depend on each other involves making the threads communicate with one another, and a simple way of doing that is through a condition variable.

Each thread has an id number, 0 through 6. Each thread should wait until the previously-numbered thread has finished. If you look at thread_main, you’ll see that it calls wait_for_thread if the id is not 0.

The way we do this is have each thread set the last_thread_finished variable to its id when it finishes. Of course, since this variable is shared state, what do you have to do to make sure the threads access it properly?

Here’s how we want things to work:

1. In wait_for_thread, we should wait on the finished_cond variable. This lets each thread wait without spin-waiting. Don’t forget what else you have to do in order to wait on a condition variable.
   - Note: like the condvars.c example I gave you (linked from the site), you will have to use a loop to check the condition instead of a simple if statement.
   - You will only have to add a few (4-5) lines!

2. In finish_thread, first synchronize access to the last_thread_finished variable like you did with output_buffer. Then, signal the finished_cond variable.
   - There are two functions you could use to do this: pthread_cond_signal and pthread_cond_broadcast. Try both, and see what happens. If your program hangs, use ctrl+C.
   - You will only have to add 3 lines!

Once everything is properly written, you should get the message in the right order!
What to submit

Tar/zip/submit the directory containing your lab8.c and lab8 executable.

```
$ cd ..
$ ls
lab8/
$ tar cvf xyz00_lab8.tar lab8/
$ ls
lab8/ xyz00_lab8.tar
$ gzip xyz00_lab8.tar
$ ls
lab8/ xyz00_lab8.tar.gz
$ cp xyz00_lab8.tar.gz ~jfb42/submit/449
```