CSSE332: Exam 2

This exam is meant to test your skills in creatively using condition variables and mutex locks. The problems are designed to be solved using condition variables and mutex locks, however, **unless otherwise stated** you can also attempt to use semaphores, or any combination thereof, to solve these problems.

Problem 1: Priority condition variables (30 points, Medium)

In this problem, you are only allowed to use condition variables and mutex locks.

In this problem, we would like to implement `pr_cond_t`, a priority condition variable that we are going to add in this class. There are 5 priorities and each thread will be assigned a priority at creation time (you don't have to worry about assigning priorities, it is already done for you).

I suggest you read through the code and the documentation stubs for each function before you start reading the next steps.

Step 1: Creating the structure and initializing it

In this first step, we will create the structure `pr_cond_t`. Think about how many condition variables you might need and what other things you might need to remember about each priority condition variable.

Step 2: Initializing the priority condition variable

In this step, you must write the appropriate code to initialize your priority condition variable. This should be a straightforward task that mimics what we normally do with `pthread_cond_init`.

Step 3: Waiting with a given priority

In this step, implement `pr_cond_wait`. In this function, the calling thread will wait on your priority condition variable but it will first register itself as holding the priority `unsigned priority`.

Step 4: Signaling the thread with the highest priority

In this step, implement `pr_cond_signal`. When this function is called, you are to wake up the thread **with the highest prioirity** that is waiting on the condition variable. Make sure that you only wake up **one** thread and that it is indeed the thread with the hightest priority.

Step 5: Testing

In the `main` function, uncomment the line `test_wait_signal()` to test your implementation so far. The output exepected looks something like the following (the threads should come out in descending order):

```
####################################################################
Thread with pr=0 waiting
Thread with pr=1 waiting
Thread with pr=2 waiting
Thread with pr=3 waiting
Thread with pr=4 waiting
Thread with pr=0 waiting
Thread with pr=4 awake
Thread with pr=3 awake
Thread with pr=2 awake
Thread with pr=1 awake
Thread with pr=0 awake
Thread with pr=0 awake
Everything done ...
####################################################################
```

Step 6: Destroy the condition variable

In this step, implement `pr_cond_destroy` to destroy the condition variable that you have created. **This step is crucial to run several tests at the same time**.

Step 7: Broadcast

In this step, implement `pr_cond_broadcast` in which you wake up *ALL* threads waiting on the condition variable, regardless of their priority levels.

Step 8: Testing

In the `main` function, uncomment the line `test_wait_broadcast` to test your implementation so far. The output expected looks something like the following:

```
####################################################################
Thread with pr=0 waiting
Thread with pr=1 waiting
Thread with pr=2 waiting
Thread with pr=3 waiting
Thread with pr=4 waiting
Thread with pr=0 waiting
Thread with pr=3 awake
Thread with pr=0 awake
Thread with pr=1 awake
Thread with pr=2 awake
Thread with pr=4 awake
Thread with pr=0 awake
Everything done ...
###################################################################
```

Step 9: Signal a specific priority

In this step, implement `pr_cond_signal_pr`. In this function, you must wake up **one** waiting thread with the given priority. In other words, even if the priority I pass to you is not the highest, you still wake up the one with that passed priority and not the highest one.

Step 10: Testing

In the `main` function, uncomment the line `test_wait_signal_pr` to test your implementation so far. Your output **must match** the output below:

```
###################################################################
Thread with pr=0 waiting
Thread with pr=2 waiting
Thread with pr=1 waiting
Thread with pr=3 waiting
Thread with pr=4 waiting
Thread with pr=0 waiting
Thread with pr=4 awake
Thread with pr=0 awake
Thread with pr=1 awake
Thread with pr=3 awake
Thread with pr=2 awake
Thread with pr=0 awake
Everything done ...
###################################################################
```

Problem 2: Salad bar (30 points, Easy)

In this problem, there are two threads: a cook and a farmer.

The farmer produces tomatoes and cucumbers whenever they are asked to until they have produced 30 cucumbers and 30 tomatoes. However, as long as there are 2 cucumbers and 3 tomatoes available, the farmer refuses to produce (because you know, capitalism).

On the other hand, the cook uses up the tomatoes and the cucumbers to make a salad. The cook uses 2 cucumbers and 3 tomatoes at a time and then asks the farmer to produce more vegetables when they run out of supplies.

Put another way, this is a glorified producers/consumers problem. Implement the semantics above using condition variables and mutex lock, or semaphores (whichever you think is easier for you).

Problem 3: H2O Bonding (30 points, Hard)

In this problem, there are two types of threads: oxygen and hydrogen.

In order to create water molecules, we need two hydrogen molecules and one oxygen molecule. To assemble these water molecules, the two hydrogen molecules and the oxygen molecule must all be present at the same time. In other words, we must create a barrier at which the individual molecules will wait for each other before proceeding to bond. However, there can be **at most one** H2O molecule being created at a time. In other words, we cannot create two or more H2O molecules simultaneously.

Implement `oxy_thread` and `hydro_thread` to satisfy the above requirements.

> Hint: In my solution, I used two condition variables, one mutex lock, and 6 state variables. You don't have to use the same solution as I did; you can use less/more condition variables and mutex locks.
>
> Hint: Here is a breakdown of what happens when an oxygen thread arrives. If the oxygen thread arrives and there are no other threads there, it must wait for two more hydrogen threads. If it arrives and there are two hydrogen threads, then it kickstarts the bonding. If it arrives and there is a bonding happening, then it will have to wait regardless.
>
> Hint: It might be useful to remember the number of oxygen threads needed as well as the total number of oxygen threads arriving. Same for hydrogen.
>
> Hint: Please attempt this problem and do not leave it blank. I want to see your thought process as much as I want to see you successfully complete this problem.