Operating Systems, Assignment 4

This assignment includes a few programming problems. 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. (16 pts) Using your textbook, please give a short definition of the following terms, in your own words. Write up all your answers into a plain text file called `terms.txt` and submit it using WolfWare classic when you're done.

   (a) external fragmentation

   (b) TLB flush

   (c) Dynamic loading

   (d) Lazy swapper

2. (30 pts) This problem is intended to give you some practice thinking about and implementing monitors. You are going to create a monitor called buffet, using POSIX mutex and condition variables.

   The driver program, restaurant.c, uses your monitor to simulate a restaurant with a busy pizza buffet. Servers (threads) are periodically adding more slices to the buffet, while restaurant patrons (other threads) are taking slices from the buffet. Your monitor will need to keep up with the set of pizza slices currently on the buffet. A slice is just a string of up to 10 characters representing the type of slice (e.g., "cheese" or "works"); the header file for the monitor defines a Slice type for this. When patrons request a desired number of slices, the monitor will give them out starting with the oldest slices. The monitor will need to block patrons if they can't yet have all the slices they want, and it will need to block servers if the buffet is already full of pizza.

   Your monitor will support two kinds of patrons, vegetarian and non-vegetarian. A vegetarian will only take vegetarian slices from the buffet, where vegetarian is simply defined as either "cheese" or "veggie". A non-vegetarian can take any type of slice, but will only take a vegetarian slice if no vegetarians are waiting.

   Your monitor will support the following functions:

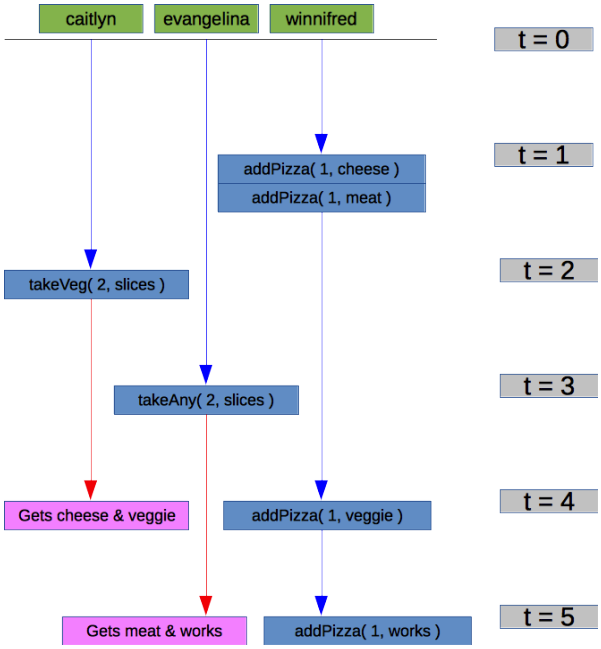
   - `void initBuffet();`
     To do any needed one-time initialization of the monitor when the program starts up.
   - `void destroyBuffet();`
     To free any resources created by the monitor at startup or while it was being used.
   - `bool takeAny( int desired, Slice slices[] );`
     Called by non-vegetarian patrons to request a desired number of slices. The monitor will store the type of slice the patron gets, oldest slice first, in the given slices array. This function will block the caller until they can have all the slices they want; it won't take any slices from the buffet until they can get everything they want. If a vegetarian is waiting for slices, this will only take non-vegetarian slices (slices other than cheese and veggie). Otherwise, this function will give them any type of slice. If the buffet is closed, this function will return false.
   - `bool takeVeg( int desired, Slice slices[] );`
     This function is like takeAny(), but it takes the oldest vegetarian slices and stores them in the given slices array. As with takeAny(), it will block the caller until they can have all the slices they want, and once the buffet is closed, this function will return false immediately.

- `bool addPizza( int count, Slice stype );`
  This function adds count slices, all of the given slice type, stype, to the buffet. The buffet only has a capacity of 20 slices, If there isn't enough room for all the slices, it will add as many as it can to the buffet and then block the calling thread until there's room for more (note that this is different fom the behavior of takeAny() an takeVeg(), where you don't take any slices until you can have all of them. This function immediately returns false when the buffet is closed.
- `void closeBuffet();`
  This function marks the buffet as closed, causing threads blocked in takeAny(), takeVeg() and addPizza() to to immediately return false (and threads subsequently calling these function to immediately return false.

The header file, buffet.h defines the interface described above as a collection of function declarations. You will implement your monitor in a file called buffet.c. You won't need to change the header file; just implement the monitor functions in buffet.c. If your monitor needs condition variables or other state information (it probably will), just make these static global variables in buffet.c.

I'm providing you with a few client programs that use your monitor. The restaurant.c file creates 8 threads (4 servers and 4 patrons) who access the monitor at random intervals. The main thread lets these threads access the buffet for 10 seconds, then it calls closeBuffet() to force them all to exit. This test driver is good for stress-testing your monitor.

I'm also providing a few smaller test drivers, test1.c, test2.c, test3.c and test4.c. These try to set up specific situations for the monitor, to see if it handles each situation correctly. The behavior of test1.c, for example, is illustrated below. There's a timeline for what happens to each thread going from top to bottom, with seconds marked to the right. The blue arrows show when a thread is sleeping, and the red arrows show when it's blocked inside the monitor. A thread, winnifred waits for a second, then adds a cheese and a meat slice to the buffet. A second later, a different thread, caitlyn, tries to take 2 vegetarian slices. There's only one, so this thread has to wait. Another second later, a thread, evangelina, tries to take two slices. There are two available, but since there's a vegetarian waiting, evangelina can't take the vegetarian slice, so she waits. Another second later, winnifred adds a veggie slice. This lets caitlyn have two vegetarian slices she wanted, so she returns. Another second later, winnifred adds a slice of "works" pizza, so evangelina is able to get the two slices she wanted.



You should be able to compile your monitor with any of these driver programs. To compile with restaurant.c, you can use a command like:

```
gcc -std=c99 -g -Wall -D_BSD_SOURCE -o restaurant restaurant.c buffet.c -lpthread
```

When run with the restaurant.c driver, I got output like the following, with somewhere in the neighborhood of 350,000 lines of output. Variations in timing will give you different output each time, and when I let the output got to the terminal window, it probably didn't run as fast.

```
$ ./restaurant
Laverne gets: works works
Daron gets: works works
Michal gets: veggie veggie veggie
Kelsey gets: cheese
Laverne gets: pepperoni pepperoni

< lots of lines deleted >

Daron gets: works works
Laverne gets: cheese cheese
Michal gets: veggie veggie cheese
Kelsey gets: veggie
Daron gets: pepperoni pepperoni
```

When you run the test1.c driver, it should print out:

```
Caitlyn gets: cheese veggie   <- after 4 seconds
Evangelina gets: meat works   <- after 5 seconds
```

When you run the test2.c driver, it should print out:

```
Laurie gets: veggie ham       <- after 2 seconds
Ladonna gets nothing          <- after 6 seconds
```

When you run the test3.c driver, it should print something like the following. The last two lines could print in the opposite order.

```
Scot is dropped off 9 slices.       <- after 1 second
Marnie gets: squid squid            <- after 3 seconds
Shirlee is dropped off 12 slices.   <- after 3 seconds
```

When you run the test4.c driver, it should print out:

```
Dorothea gets: asbestos                    <- after 3 seconds
Neville gets: silt anchovies pepperoni     <- after 5 seconds
```

If you'd like some hints, I can tell you something about how I implemented my solution. I used a mutex to implement the monitor (of course) and three condition variables, along with several other global variables to keep up with what was on the buffet, and what the threads were doing. I used one condition variable to block vegetarian threads when they could get the slices they wanted, another to

block non-vegetarian threads and one more to block servers when the buffet was full. You don't have to implement your solution the way I did, but this may give you some ideas if you're having trouble getting started.

In lots of my monitor functions, I found the pthread_cond_broadcast() function to be very convenient, like when a new vegetarian slice arrived and I wanted to wake up all the waiting vegetarian threads. It's probably easier to solve this problem using this broadcast function, but, if you can do it without the broadcast, using only the signal and wait functions, I'll give you 5 points of **extra credit**.

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

3. (30 pts) This problem is intended to help you think about avoiding deadlocks in your own multi-threaded code, and the performance trade-offs for different deadlock prevention techniques. I've written a simulator for a busy kitchen with lots of chefs who need to use a shared set of cooling appliances. It's kind of like the dining philosophers problem, except that number of appliances used by a chef isn't limited to two.

   You'll find my implementation of this problem on the course website, `Kitchen.java`. If you look at my source code, you'll see that we have ten chefs and eight appliances. Each chef acquires a lock on the appliances he or she needs before they start cooking. Then, once they're done, they release the locks and rest for a while. The program counts how many dishes are prepared and reports a total for each chef and a global total at the end of execution.

   Unfortunately, my program deadlocks in its current state. I'd like you to fix this program using three different techniques. You're not going to change the appliances used by each chef or the timings for preparing each chef's dish. You're just going to change how the locking is done.

   (a) First, rename the program to `Global.java`. In `Global.java`, get rid of the code to lock individual appliances. Instead, let's implement a policy that only one chef at a time can cook. Just use one object for synchronization and have every chef lock that one object before they cook. That way, we can be certain that two chefs can't use the same appliance at the same time (only one at a time can be cooking), and we should be free from deadlock.

   This solution should work, but it's not ideal. It prohibits concurrency. When I tried this technique, I only got around 280 total dishes prepared (on an EOS Linux machine).

   (b) Instead of forcing the chefs to cook just one at a time, we should be able to prevent deadlock by just having them all lock the appliances in a particular order. Copy the original program to `Ordered.java` and apply this deadlock prevention technique. Choose an order for the appliances that you believe will maximize throughput, one that will maximize the total number of dishes prepared. You may want to experiment with a few different orders as you try to find a good one. Or, you can just think about which appliances should be locked early and which ones can be locked later.

   This solution is a little more complicated than Global.java, but it should perform better since it permits concurrent cooking among some subsets of the chefs. When I implemented this technique, I got around 750 total dishes prepared (on an EOS Linux machine). You should try to do as well in your solution.

   (c) Instead of applying the no-circular-wait deadlock prevention technique, we can apply the no-hold-and-wait technique; chefs will allocate all their needed appliances at once, only taking them if they are all available. Copy the program to `TakeAll.java`. Replace objects used to acquire locks for each appliance with a boolean variable for each appliance. We'll use these variables as flags, keeping up with whether or not each appliance is currently in use.

   Also, replace the lock-acquiring code with a new synchronized block. The call to cook() will be after this block (outside the synchronized block), and you'll need one new object to control entry

to this synchronized block. Inside the block, you'll check to see if all the needed appliances are available. If they are, you'll mark them all as in use, leave the synchronized block and then call cook(). If some of the needed appliances are in use, you'll use to wait() to block until a chef finishes with the needed appliance.

After cooking, each chef will enter another synchronized block (essentially, re-entering the monitor) to mark its appliances as no longer in use. While inside, the chef can use notifyAll() to wake any chef that may have been waiting for one of the appliances that are no longer in use.

In terms of performance, my TakeAll.java implementation did a little better than either of my other two solutions. On an EOS linux machine, I got about 780 total dishes prepared, but, often, some chefs got to do very little cooking. I'd say this is a consequence of the starvation risk in this solution. No chef was prevented from cooking, but some (one in particular) didn't seem to get treated very fairly.

Once all three of your deadlock-free programs are working run them each and write up a report in a file called kitchen.txt. In your report, you just need to (clearly) report how many dishes were prepared in each of your three programs.

When you're done, submit electronic copies of your **Global.java**, your **Ordered.java**, your **TakeAll.java** and your **kitchen.txt**.