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

   This monitor will manage access to a stage, where a number of musicians (threads) are spontaneously forming jazz combos to play for a short period of time.[1] The rules for who can play are simple. There can be at most three musicians on the stage at a time, and there can't be two of the same instrument on stage at once.

   The monitor will support the following functions:

   - `void initCombonator()`
     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. 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 destroyCombonator()`
     This function frees any resources allocated by the monitor. This could include freeing dynamically allocted memory if needed or
   - `void startPlaying( char const *instrument )`
     Threads call this function when they want to take the stage and start playing. Inside the function, you'll have to make threads wait if there are already three musicians on stage, of if there's already a musician playing the given instrument.
   - `void stopPlaying( char const *instrument )`
     Threads call this function when they're done playing and want to take a break.

   Each time the set of musicians changes, you'll print out a message like the following, listing the instruments that are currently on stage.

   ```
   Now playing: cello trombone cowbell
   ```

   If the stage isn't full, just list the instruments that are on stage, like:

   ```
   Now playing: banjo marimba
   ```

   In particular, if the stage is empty, you'll print an empty list of instruments, like:

   ```
   Now playing:
   ```

   At program start-up (in your initCombonator() function) and each time a musician takes the stage or leaves, you'll print the new set of instruments that's playing.

[1] A jazz combo is a small collection of jazz musicians. That's where the name of our monitor, combo-nator, comes from. Plus, it's a play on words, with a combinator (e.g., the Y combinator) being a useful concept in semantics of programming languages.

Playtime Report

Whenever the set of musicians changes and at program termination (inside destroyCombonator()), your monitor will print a report of how long (in milliseconds) the most recent set of musicians was playing. So, as musicians enter and leave the stage, your program's output will look like.

```
Now playing: banjo trombone marimba
  4 ms
Now playing: banjo trombone
  0 ms
Now playing: banjo trombone serpent
  4 ms
```

This fragment of output shows that a banjo, trombone and marimba were playing for 4 milliseconds. Then, the marimba player left the stage, and, for a moment, there were just two musicians. Then, almost immediately afterward a serpent player took the stage. This new ensemble got to play for 4 more milliseconds.

Round the play time to the nearest millisecond, and, as shown above, indent the playtime report by two spaces.

To measure play time for each ensemble, you'll use the gettimeofday() system call. You'll need to read the online documentation for this call to figure out how to use it (e.g., you can enter 'man gettimeofday'). This call reports the current time, in seconds since January 1, 1970 along with a number of microseconds. You can use this to record the time when an ensemble starts playing. When the ensemble changes, you can check the time again to figure out how long they were playing. Keep in mind, this system call reports the time in seconds and microseconds, but your output just needs to be in milliseconds.

Play Time Summary

As different ensembles take the stage and leave, your monitor will record the total play time in milliseconds for each size of ensemble. At program termination (in the destroyCombonator() function), you'll report these totals. This will include the total time during which there were no musicians on stage (silence), one musician (a solo), two musicians (a duet) and three musicians (a trio). Your summary will look like:

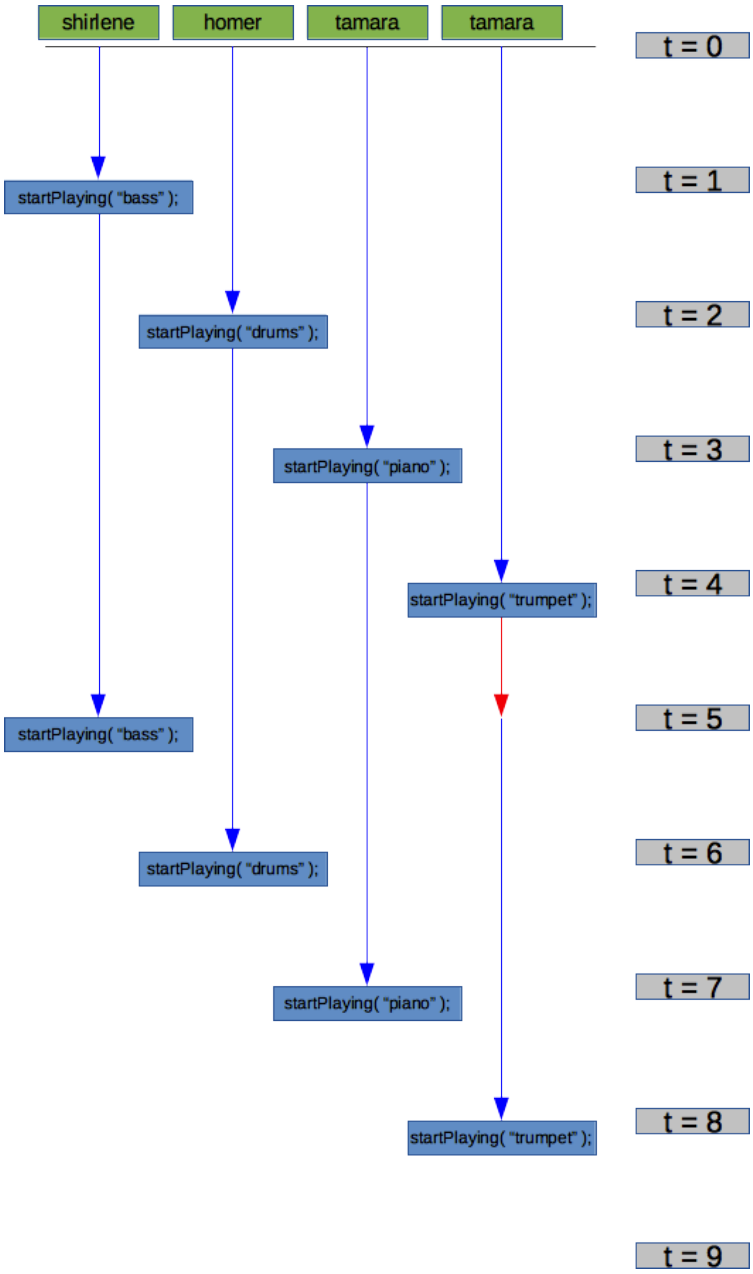
```
Summary
silence:    24 ms
   solo:   323 ms
   duet:  1867 ms
   trio:  7566 ms
```

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 combonator.c driver1.c -o driver1 -lpthread -lm
```

The driver1.c program is designed to make sure your monitor only lets three musicians at a time onto the stage. As illustrated below, it creates four threads. After a second, the first one takes the stage (playing bass). A second later, the next one takes the stage (playing drums). A second after that, a third thread starts playing piano. Then, a final thread tries to take the stage playing trumpet. Since the stage is full, it has to wait for a thread to leave (shown as a red arrow in the figure). After waiting for a second, the bass player leaves and the trumpet player gets to take the stage.

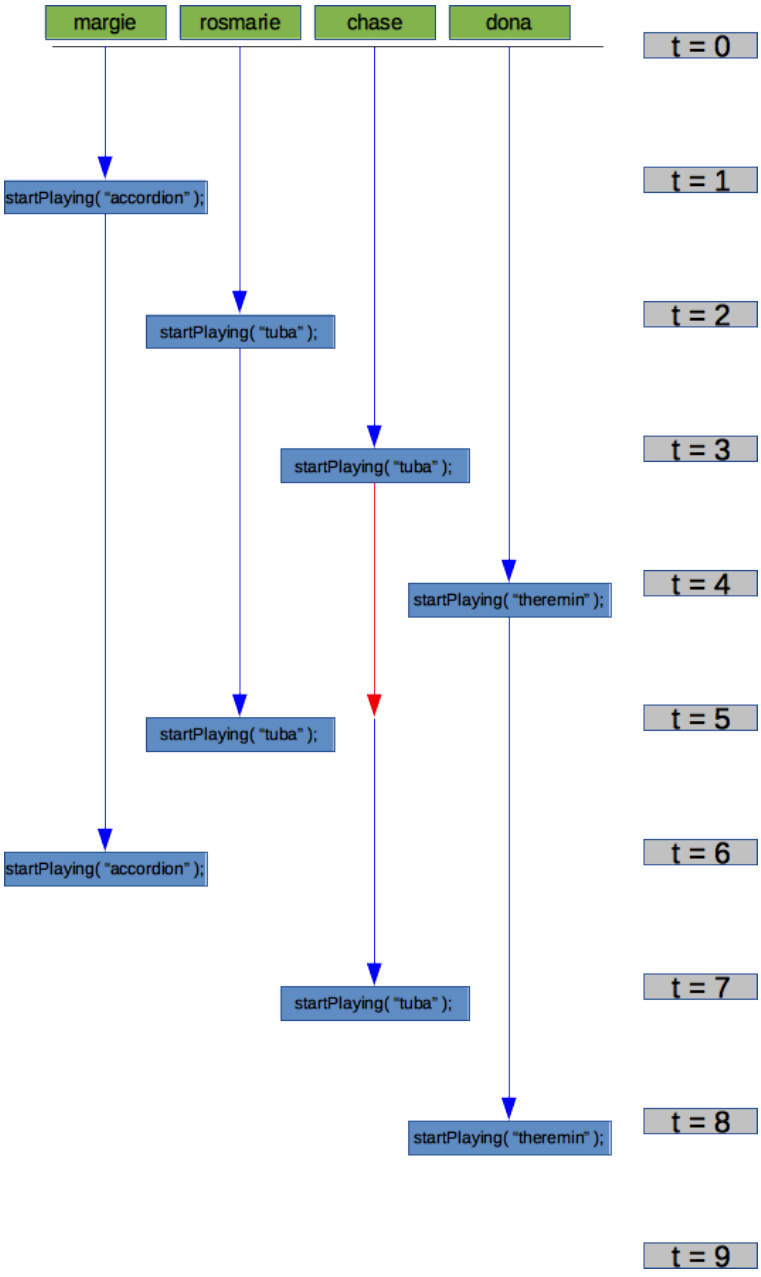


If you compile and run this driver, you should get output like the following (maybe with some very small variation in the timing).

```
$ ./driver1
Now playing:
```

```
  1000 ms
Now playing: bass
  1000 ms
Now playing: bass drums
  1000 ms
Now playing: bass drums piano
  2000 ms
Now playing: drums piano
  0 ms
Now playing: trumpet drums piano
  1000 ms
Now playing: trumpet piano
  1000 ms
Now playing: trumpet
  1000 ms
Now playing:
  1000 ms
Summary
silence:  2000 ms
   solo:  2000 ms
   duet:  2000 ms
   trio:  3000 ms
```

The driver2.c program checks to make sure two of the same instrument won't be permitted to play at the same time. As illustrated below, an accordion and a tuba player take the stage. When another tuba player wants to play, he has to wait, but a theremin player is permitted to start playing a second later (since there's space on the stage, and the theremin isn't a duplicate instrument). After the first tuba player leaves at time 5, the second one is permitted to start playing.



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
Now playing:
  1001 ms
Now playing: accordion
  1000 ms
Now playing: accordion tuba
  2000 ms
Now playing: accordion tuba theremin
```

```
  1000 ms
Now playing: accordion theremin
  0 ms
Now playing: accordion tuba theremin
  1000 ms
Now playing: tuba theremin
  1000 ms
Now playing: theremin
  1000 ms
Now playing:
  1000 ms
Summary
silence:  2001 ms
   solo:  2000 ms
   duet:  3000 ms
   trio:  2000 ms
```

The driver3.c program, creates 10 threads that repeatedly try to take a short break, then perform on stage. Each thread has an instrument that it plays, but there are two copies of some of the instruments (e.g., two banjos). This will let you test out your monitor's behavior of preventing multiple copies of the same instrument from being on stage at the same time.

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
Now playing:
  19 ms
Now playing: cowbell
  0 ms
Now playing: cowbell cello
  2 ms
Now playing: cowbell cello flute
  8 ms
Now playing: cowbell flute
  0 ms

... lots of output omitted ...

Now playing: trombone flute
  10 ms
Now playing: flute
  3 ms
Now playing:
  0 ms
Summary
silence:    28 ms
   solo:   336 ms
   duet:  1887 ms
   trio:  7590 ms
```

Be sure your program doesn't deadlock when you run it with this driver. Your summary at the bottom should be similar to the example above, and you can check the number of output lines you get to see if

your solution seems to be working. Run as follows, I got around 10,700 output lines every time I ran my solution (with some variation from run to run).

```
$ ./driver3 | wc
```

Monitor Implementation

I'm providing the header file, combonator.h, that describes the monitor interface, but you get to create the implementation file, combonator.c yourself. Inside this file, you'll implement all the functions prototyped in the header. You can also implement other (static) functions if you want, to help simplify your monitor and re-use code.

You'll implement your monitor using the POISIX monitor API. Inside your monitor, you'll implement a mechanism for keeping up which instruments are currently playing, the total play time for each ensemble size, and you'll need synchronization objects (mutex and condition variables) to control access to the monitor and to make threads wait when they need to. Your monitor can use (static) global variables to maintain its state.

You can use `pthread_cond_broadcast()` for this problem if you want. But, if you can solve the problem more efficiently, using pthread_cond_signal() to wake just one thread at a time (i.e., waking a specific thread that's ready to make progress), then you can earn up to 6 points of **extra credit**.

Submitting your Work

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

2. (40 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 cooking 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 10 chefs and 8 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 and release it when they're done. 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 concurrent cooking by chefs who don't need any of the same appliances. When I tried this technique, I only got around 320 total dishes prepared (on my 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 760 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 original 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 of the chef's 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 typically got over 900 total dishes prepared, but, often, some chefs got to do a lot less cooking than some others. I'd say this is a consequence of the starvation risk in this solution. No chef was prevented from cooking, but some 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. For each version, also report the minimum number of dishes prepared by any chef (i.e. for the chef that prepared the fewest dishes, how many did they prepare?) and the maximum number number of dishes prepared by any of the chefs. This will help to show how fair each solution was among the chefs.

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