Operating Systems, Assignment 4

This assignment just includes a couple of programming problems, and an easy assignment to activate an account you'll need for homework assignment 5. 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. (10 pts) Arc account activation.

   On our next homework assignment, you'll need to use a high-performance computing cluster called arc. Before you can do this, you'll need to follow steps to activate your account on arc and set up a key pair so you can log in. If you don't complete arc activation on this assignment, there's a problem you won't be able to do on the next assignment. To help make sure everyone is ready, I have a low-point-value problem on this assignment, requiring you to activate your account on arc and log in on the head node of the arc cluster.

   About five days before this assignment is due, you should receive an email from the system administrator on arc. It will contain a link to more information about the arc cluster (most of which won't apply to you) and a link to enter a public key to activate your account and permit you to log in. This email will expire within about a week, so you need to activate your arc account soon after you get it or you won't be able to do one of the problems on the next assignment.

   First, you'll need to generate an RSA key pair. This will create a pair of files under a directory named `.ssh` inside your home directory. You will need access to these files whenever you want to log in to arc, and you can only log into arc from a machine on the university network. It's probably good to do this from a machine that has access to your AFS file space, so you can get to your new key pair from just about any university machine. Creating your key pair from one of the remote.eos.ncsu.edu machines should work fine. **Don't create your key pair on a VCL image.** Many of the VCL images are temporary systems that have a temporary filesystem. That filesystem will get deleted when the machine goes away. If you create your key pair on a system like this, then you won't be able to use it later, after your VCL image is deleted.

   If you've never generated a ssh key pair before, it's easy. If you've done it before, it's possible to use a key pair you've already made or to keep multiple key pairs around for different purposes. If you want to generate your key differently than what I'm suggesting here (e.g., naming it something different), that's fine, but you will need to be responsible for knowing what you're doing.

   To generate your key pair, log in on one of the EOS Linux systems, and enter the command below. It may ask you some questions about how to store your key. You can just take the default answers for these.

   ```
   ssh-keygen -t rsa -b 4096
   ```

   This command will create a 4096-bit RSA key pair, and store it in a couple of files in the .ssh directory under your home directory.

   Have a look at the public key of this pair:

   ```
   cat ~/.ssh/id_rsa.pub
   ```

   To register this public key with the arc system, you'll need to click on the hyperlink you got in your email from the arc administrator. To communicate with the machine at the end of this link, it looks like you need to have a campus IP address, so you can either do this from campus, or you can VPN into campus before following the rest of the instructions. If you've never used VPN to connect to campus before, there are instructions for installing the software you need and getting connected at:

`https://oit.ncsu.edu/campus-it/campus-data-network/vpn/`

Now, click the link in your email for activating your account. Your web browser will probably complain that it's seeing a self-signed certificate; that's OK. Once you get to the destination page, paste the contents of your `.ssh/id_rsa.pub` file into the text box the email link takes you to. Be sure to get the entire contents of your `.ssh/id_rsa.pub` file, including the ssh-rsa part at the beginning. Then, press the button at the bottom of the form and your account should be activated. ... but you haven't completed this part of the assignment yet.

To get credit for this part of the assignment, you just need to do one more thing. From one of the university Linux machines, you should be able to ssh into the head node on the arc system:

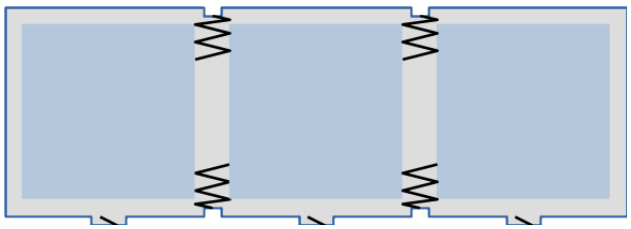
```
ssh arc.csc.ncsu.edu
```

If this works, you'll get a shell prompt on the arc system. You need to get as far as getting this shell prompt on arc to get credit for this problem. When I was on the arc system, I got a prompt that looked like the following. Yours should look the same, although you will see a different unity ID.

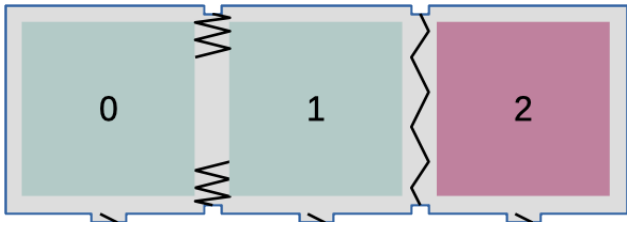
```
[sjiao2@login ~]$
```

If this works, you should be in good shape for the next assignment. You can log out of the arc system by entering `exit`.

2. (40 pts) For this problem, you are going to use POSIX mutex and condition variables to implement part of a simulation of conference hall allocation system. The picture below shows the type of room we'll be imagining. It's a long hall with partitions at regular intervals.



The partitions in the hall can be used to break it up into separate spaces. So it can be used by multiple organizations at the same time, each using a one or more of the spaces, with partitions between them. The figure below shows two spaces on the left, separated from a single space on the right.



The hall can contain an arbitrary number of spaces, n, given at start-up. The spaces are numbered from zero on the left up to $n-1$ on the right. When an organization wants to use part of the hall, they specify how many spaces wide it needs to be. Then, they wait until that many contiguous spaces are available. Once they have been assigned their requested number of spaces, they can use them for as long as they want. While one organization is using part of the hall, no other organization can use those

same spaces, but other spaces could be used by another organization. That's what the partitions are for. In the figure above, for example, one organization could be using spaces 0 and 1 while a different organization used space 2.

When an organization is done using part of the hall, they free it. Then, it's available for other organizations to use.

Space Allocation

In our simulation, the organizations are implemented as threads. They call functions on our system as they allocate and free space in the hall. If a thread can't have its requested space right away, the system will make them wait until enough space is available.

The system will use a first-fit policy for allocating space. It will always give the lowest-numbered spaces in response to each request. For example, imagine a hall that's 10 spaces wide, with spaces 2, 3, 6, 7 and 8 available and the other 5 spaces in use. If an organization needed a space of width 2, the system would give it spaces 2 and 3; that's the lowest numbered contiguous space that's large enough for the request. However, if an organization needed a space of width three, the system would have to allocate spaces 6, 7 and 8 instead.

The system can grant requests in any order; it doesn't have to grant them in the order the requests are made. However (unless you do the extra credit) it shouldn't leave spaces idle if that space could satisfy the request of a waiting thread.

Monitor Implementation

You are going to develop a synchronization mechanism called monitor to implement the conference hall allocation system. Some programming languages such as Java and C++ provide a built-in monitor, but C does not. A partial implementation of the monitor and some test programs are provided. The header file `hall.h` defines the interface you will implement. In `hall.c`, you'll implement the monitor's functions. In `hall.c`, you can also create any state you need for your monitor. This will include a mutex for controlling access to the monitor. Probably, you'll also need one or more condition variables, for making threads wait in the monitor when they need to. You can also have other variables to keep up with the current state of the monitor (e.g., what parts of the hall are in use). Define variables like these as static global variables in your monitor implementation file. That way, you can access them from any of your monitor's functions, but they won't collide with names used outside this one component.

For some functions, the monitor is expected to print out an *allocation report*. This is a sequence of n characters, where n is the width of the hall. Each character in the report shows the current state of one of the spaces, from left to right. An asterisk indicates that a space is free. For spaces that have been allocated, the corresponding character should give the first letter of the name of the organization currently using the space. For example, the report "**aabb*" represents a hall that's 7 spaces wide. The left two spaces and the rightmost space are available. Spaces 2 and 3 are in use by an organization that starts with 'a', and spaces 4 and 5 are in use by an organization that starts with 'b'.

Your monitor will provide the following functions:

- `void initMonitor( int n )`
  This function initializes the monitor, allocating any heap memory it needs and initializing it state where necessary. The n parameter is the total width of the hall, the number of spaces it contains.

- `void destroyMonitor()`
  This function is called after we're done using the monitor. It should free any resources used by the monitor.
- `int allocateSpace( char const *name, int width )`
  A thread calls this function when it wants to use a space in the hall. The first parameter is the name of the organization for the thread, and the second, width, parameter is the width of the contiguous space it needs. The function returns a value named `start` described below.

  If there isn't enough space available, this function will make the thread wait until it can have the space it needs. If a thread has to wait in this function, the monitor should print a message like the following, where *name* is the name given by the thread, and *current-allocation* is an allocation report for the hall. Only print this line if the thread has to wait. Omit it if the thread can immediately have the space it's requesting.

  name waiting: *current-allocation*

  Once the thread can have its requested space, the allocateSpace() function should print a line like the following, giving the name of the thread and the (updated) allocation report for the hall.

  name allocated: *current-allocation*

  The start value returned by this function should be the index of the leftmost space assigned to the thread. This tells the thread which part of the hall it was given. The thread is expected to pass this value (along with it's name and the width of the allocation) back to the monitor when it frees the space.
- `void freeSpace( char const *name, int start, int width )`
  A thread calls this function when it is done using the space it was given in a previous call to allocateSpace(). The name field is the name of the thread's organization, the start is the index of the leftmost space allocated to the thread, and width is the number of contiguous spaces allocated to the thread. After calling this function, the spaces from start up to start + width - 1 are considered available, and may be given to some other thread.

  This function should print out a line like the following, where *name* is the name of the thread that freed the space and *current-allocation* is an allocation report for the hall (immediately after the spaces are freed).

  name freed: *current-allocation*

Extra Credit

There's a risk of starvation with this monitor. Threads that need a very wide space could starve. For **up to 8 points of extra credit**, build your monitor so it prevents starvation.

To do this, you'll use aging. The age of any thread, t, will be defined as the number of other threads that return from allocateSpace() wile t is waiting in allocateSpace() (i.e., the number of other threads that get space before t). So, as threads have to wait, their age will go up.

If you do the extra credit, you will need to report an extra value in the output for your allocated message printed by allocateSpace(). Your message should look like the following. The value a in parentheses should give the age of each thread when it gets its requested space. This will help us check your extra credit, and it will probably help you debug as you're developing the extra credit part of your solution.

name allocated (a): *current-allocation*

A thread's age starts at zero each time a thread calls allocateSpace(). So, threads don't accumulate age from one request to the next.

We'll prevent starvation by making threads wait for their requested space (even if space is available) when there's a much older thread waiting. Specifically, the monitor should make thread t_b wait for space (even if space is available) whenever there's another thread, t_a with an age that's more than 100 greater than t_b.

For example, if thread t_a is waiting in allocateSpace() with an age of 101, a new thread, t_b calling allocateSpace() wouldn't be able to get its space right away (even if space was available). The new thread, t_b would have an age of zero and t_a would have an age that's more than 100 greater. The t_b thread would have to wait for t_a. Of course, some other waiting thread, t_c could still get its space right away, provided t_c had an age of 1 or greater.

For the extra credit, you don't get to change the driver programs. You should be able to implement aging using only code in the monitor. In fact, your starvation-free monitor should work with other test drivers that aren't given out with the assignment.

Driver Programs

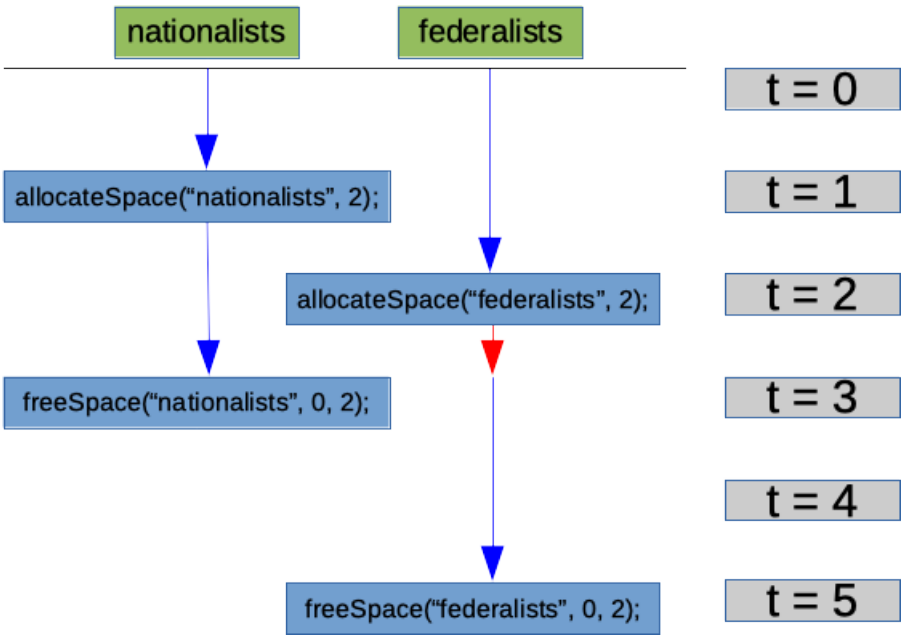
We're providing four test driver programs to help you exercise your monitor, driver1.c, driver2.c, driver3.c and driver4.c. There's also a driver you can use if you're trying the extra credit option. There are other drivers that we're not providing with the assignment. We'll use these when we're testing your monitors. Take a look at drivers 1, 2 and 3 and you'll see you could easily write your own driver to test out particular behaviors from your monitor. That's what we'll do when we're testing your solution.

To compile one of the drivers with your monitor implementation, you should be able to use a command like the following. This compiles in `driver1.c` with your monitor implementation and writes an executable named `driver1`. To try out a different driver, just change the name of the driver source file and the executable name (the name given after the -o option).

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

The driver1.c program is designed to make sure two organizations aren't permitted to use the same space at the same time. The following figure shows what should happen when you run it with a working monitor. The organizations are named after historic political parties. We make a hall that's three spaces wide.

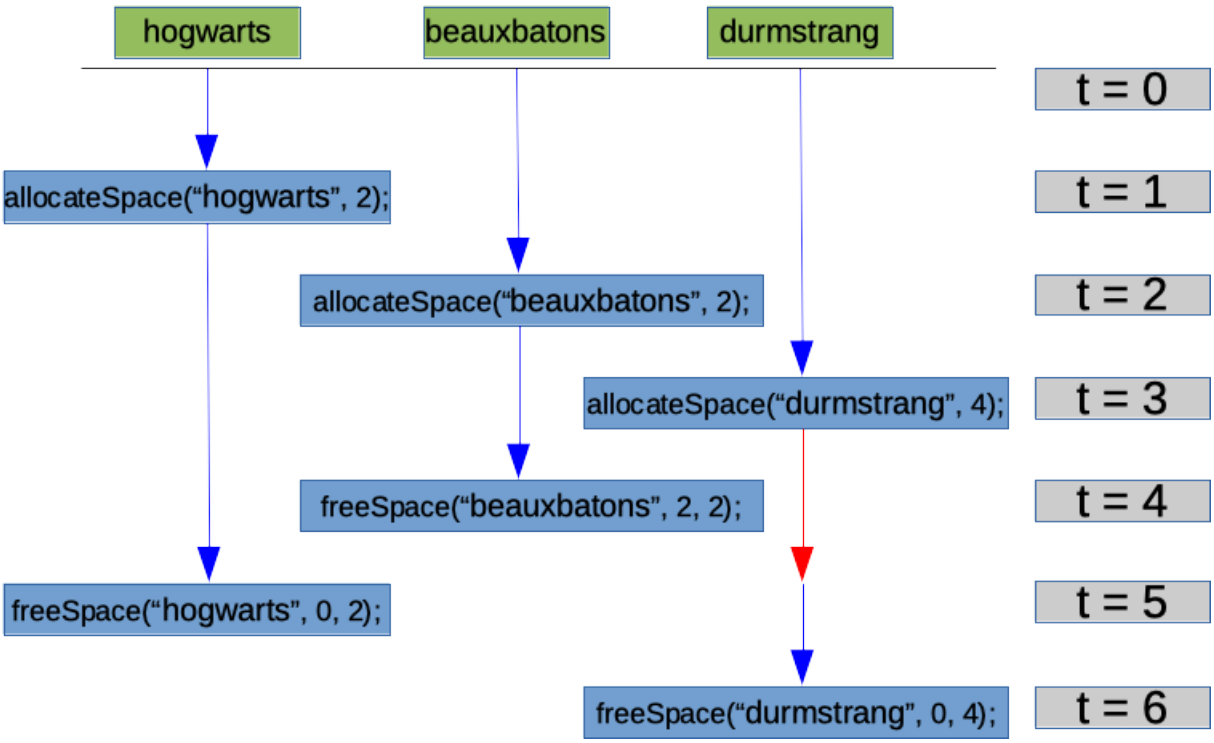
The two vertical lines show the actions of two threads, nationalists and federalists. Blue boxes show when these threads call functions in the monitor and the red arrow shows where a thread has to wait inside the monitor function before returning. The expected behavior is ordered top-to-bottom by time (with particular seconds shown on the right). After a second, the nationalists thread tries to allocate a space of width 2. It should immediately get the two spaces on the left (0 and 1). A second later, the federalists thread tries to allocate a space of width 2, but it has to wait. When the nationalists thread frees its space at time 3, the federalists thread should get two spaces (again, the leftmost two spaces because of the first-fit policy). It uses these spaces for two seconds then frees them.



You can run your program and watch the output as it's produced to check the timing. I found it useful to hit return as I counted seconds. This put blank lines at (approximately) one-second intervals in the output. It made it easy to look at the terminal output and see when each message was printed as well as when multiple messages get printed at about the same time. Here's the output you should expect from this first driver (along with approximately when you should see each message in parentheses).

```
nationalists allocated: nn*        (at 1 second)
federalists waiting: nn*           (at 2 seconds)
nationalists freed: ***            (at 3 seconds)
federalists allocated: ff*         (at 3 seconds)
federalists freed: ***             (at 5 seconds)
```

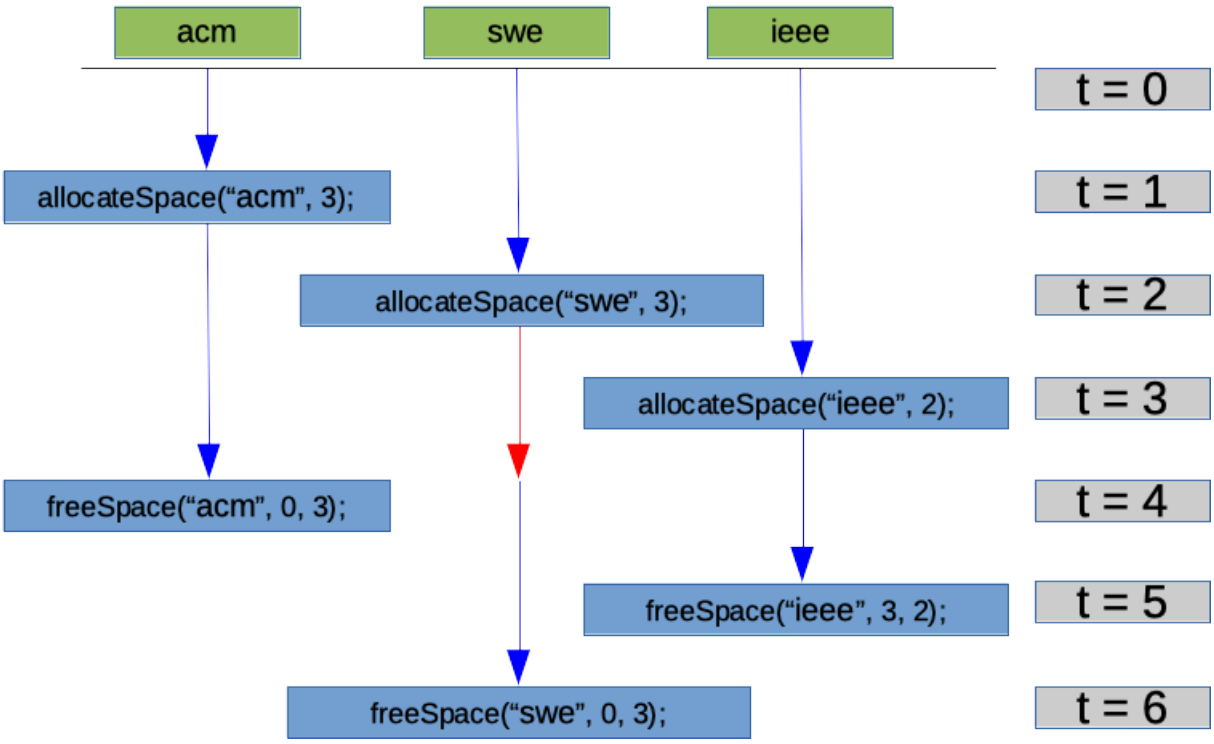
The driver2.c program checks to make sure space is given to an organization only after all other organizations get out of the way. It uses schools of witchcraft and wizardry as the organizations, with a hall that's 4 spaces wide. After a second, hogwarts allocates a space of width 2. Then, beauxbatons allocates the remaining two spaces. At time 3, durmstrang tries to allocate a width-four space, but nothing is available, so they have to wait. At time 4, beauxbatons frees their space, but durmstrang still has to wait another second for hogwarts to give up the remaining two spaces. Durmstrang holds all four spaces for a second then frees them.



If you build and run your monitor with driver2.c, you should get output like the following (with the times shown at the right).

```
hogwarts allocated: hh**          (at 1 second)
beauxbatons allocated: hhbb       (at 2 seconds)
durmstrang waiting: hhbb          (at 3 seconds)
beauxbatons freed: hh**           (at 4 seconds)
hogwarts freed: ****              (at 5 seconds)
durmstrang allocated: dddd        (at 5 seconds)
durmstrang freed: ****            (at 6 seconds)
```

The driver3.c program uses professional organizations in computing and engineering to show that requests for a small amount of space may be granted before requests for more space. It uses a hall of width 5. After a second, the acm thread gets the first three spaces. A second later, the swe thread asks for a space of width 3, but it has to wait, because only two spaces remain. A second later, the ieee thread is able to get its request granted immediately, since it only needs two spaces. Later, the swe thread is able to get its requested space after the acm thread frees its space.



If you build and run your monitor with driver3.c, you should get output like the following:

```
acm allocated: aaa**          (at 1 second)
swe waiting: aaa**            (at 2 seconds)
ieee allocated: aaaii         (at 3 seconds)
acm freed: ***ii              (at 4 seconds)
swe allocated: sssii          (at 4 seconds)
ieee freed: sss**             (at 5 seconds)
swe freed: *****              (at 6 seconds)
```

The driver4.c program is a stress test. It uses 20 threads named after 1970s bands to allocate and free spaces of various widths over and over, waiting a random fraction of a second between actions. The program behavior is random, and variations in timing will give you different output on each execution. However, there are some things you can look for. First, check to make sure your solution doesn't deadlock. The first 4 threads need wide spaces, but the last four need much less space, so they will probably get what they're asking for more quickly. From my sample output below, you can see the first band, Aerosmith, got their requested space only 408, but Yes got theirs 968 times. Overall, I got almost 14,000 total reservations when I ran this driver on one of the remote EOS Linux machines.

```
Ocean allocated: OOO*****************
Queen allocated: OOOQQ***************
Heart allocated: OOOQQHHHH***********
Ramones allocated: OOOQQHHHHRR*********
Carpenters allocated: OOOQQHHHHRRCCCCC****
Journey allocated: OOOQQHHHHRRCCCCCJJJ*
M waiting: OOOQQHHHHRRCCCCCJJJ*
```

```
Delfonics waiting: OOOQQHHHHRRCCCCCJJJ*
Journey freed: OOOQQHHHHRRCCCCC****
M allocated: OOOQQHHHHRRCCCCCMMM*
Heart freed: OOOQQ****RRCCCCCMMM*
U2 allocated: OOOQQU***RRCCCCCMMM*
Toto allocated: OOOQQUT**RRCCCCCMMM*
Genesis waiting: OOOQQUT**RRCCCCCMMM*
Ocean freed: ***QQUT**RRCCCCCMMM*
Queen freed: *****UT**RRCCCCCMMM*
Delfonics allocated: DDDDDUT**RRCCCCCMMM*
Journey waiting: DDDDDUT**RRCCCCCMMM*

... lots of output omitted ...

Eagles freed: ******MMM****GGGG***
M freed: **************GGGG***
Genesis freed: ********************
Aerosmith made 408 reservations
Bread made 409 reservations
Carpenters made 397 reservations
Delfonics made 411 reservations
Eagles made 555 reservations
Foghat made 553 reservations
Genesis made 564 reservations
Heart made 545 reservations
Isis made 709 reservations
Journey made 698 reservations
M made 707 reservations
Ocean made 696 reservations
Parliament made 838 reservations
Queen made 830 reservations
Ramones made 857 reservations
SANTANA made 834 reservations
Toto made 978 reservations
U2 made 949 reservations
Wings made 955 reservations
YES made 968 reservations
Total: 13861
```

Extra Credit Test

The driver10.c program is for the extra credit part of the assignment. It's a lot like driver4.c, but one thread, Aerosmith, repeatedly requests all the space, while all the other threads just request spaces of width 1. The thread with the wide request might not get the space it needs until all the other threads exit. That's what seems to have happened in the sample output below.

```
Ocean allocated: O*******************
Ramones allocated: OR******************
Heart allocated: ORH*****************

... lots of output omitted ...

Aerosmith allocated: AAAAAAAAAAAAAAAAAAA
Aerosmith freed: ********************
```

```
Aerosmith made 1 reservations
Bread made 996 reservations
Carpenters made 989 reservations
Delfonics made 977 reservations
Eagles made 1001 reservations
Foghat made 991 reservations
Genesis made 982 reservations
Heart made 1009 reservations
Isis made 980 reservations
Journey made 1001 reservations
M made 998 reservations
Ocean made 972 reservations
Parliament made 1006 reservations
Queen made 987 reservations
Ramones made 944 reservations
SANTANA made 994 reservations
Toto made 969 reservations
U2 made 1002 reservations
Wings made 988 reservations
YES made 999 reservations
Total: 18786
```

If you implement the extra credit, you should see that it fixes this. It will reduce overall performance a little but even the Aerosmith thread should get its requested space occasionally. Here's what I got when I ran driver10.c with my extra credit solution:

```
Ocean allocated (0): O******************
Queen allocated (0): OQ*****************
Heart allocated (0): OQH****************

... lots of output omitted ...

Aerosmith allocated (16): AAAAAAAAAAAAAAAAAA
Aerosmith freed: ********************
Aerosmith made 140 reservations
Bread made 892 reservations
Carpenters made 867 reservations
Delfonics made 903 reservations
Eagles made 901 reservations
Foghat made 890 reservations
Genesis made 888 reservations
Heart made 892 reservations
Isis made 894 reservations
Journey made 901 reservations
M made 883 reservations
Ocean made 902 reservations
Parliament made 910 reservations
Queen made 891 reservations
Ramones made 895 reservations
SANTANA made 890 reservations
Toto made 877 reservations
U2 made 891 reservations
Wings made 897 reservations
YES made 880 reservations
Total: 17084
```

Submitting your Work

When you're done, submit just your monitor implementation in **hall.c** under the assignment named HW4_Q2 on Gradescope. You shouldn't need to change the header or driver files, so you don't need to submit copies of these.

3. (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.c`. 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. This is implemented with a mutex. Before cooking, each chef does a pthread_mutex_lock() on the mutex for each of the appliances they need. They quickly prepare a dish. Then they release the locks on the appliances they were using and rest for a little while before preparing another dish. They do this over and over for 10 seconds; then they all terminate. The program counts how many dishes are prepared and reports a total for each chef and a global total at the end of execution. This is like a measure of performance. We'd like chefs to be able to prepare as many dishes as possible.

   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 time they need to cook or rest on each iteration. You're just going to change how the locking of appliances is done.

   (a) First, copy the original `kitchen.c` program to `global.c`. In `global.c`, get rid of the mutex variables for each appliance and the code to lock appliances individually. Instead, let's implement a policy that only one chef at a time can cook. Just use one mutex for synchronization. Have every chef lock that one mutex before they cook and unlock it when they're done (i.e., while they're resting). 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 at all. With just one mutex, our solution should be free of potential deadlocks.

   This solution should work, but it's not ideal. It prohibits concurrent cooking by chefs, even if they don't need any of the same appliances. When I tried this technique, I only got between about 300 and 320 total dishes prepared on one of the remote Linux machines. There was some variation from execution to execution.

   (b) Instead of forcing the chefs to cook just one at a time, we should be able to prevent deadlock by having them all lock the appliances individually, but in a particular order. Copy the original `kitchen.c` program to `ordered.c` 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 can be locked early and which ones should be locked as late as possible.

   This solution is a little more complicated than `global.c`, but it should perform better since it permits concurrent cooking among some subsets of the chefs, provided they don't need any of the same appliances. When I implemented this technique, I got between about 780 to 825 total dishes prepared (on an EOS remote Linux machine). You should try to do at least 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.c`. Replace mutexes representing

locks for the appliances with boolean variables 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 block protected by a mutex. Inside this block, you will check to see if all the appliances needed by a particular chef are available. If they are, you'll set the flags for all these appliances, showing that they are now being used. If the needed appliances aren't all available, you will use a condition variable to wait until an appliance is released. Protecting this code with a mutex ensures that only one thread at a time can be looking at appliances and changing their flags.

Once a chef has all the appliances they need (marked them as in use), they will cook. Since the cook function outside the mutex block will let multiple chefs cook at the same time, as long as they all have the appliances they need.

After cooking, the chef will enter another block of code protected by the same mutex. In this block, they will mark all the appliances they were using as available and then wake up any other chefs that were waiting on an appliance. To wake the remaining chefs, you can use the pthread_cond_broadcast() function. We could solve the problem without broadcast, but using this function will make it easier.

After releasing all the appliances, a chef can call rest().

In terms of performance, my `takeAll.c` implementation did a bit better than either of my other two solutions. On an EOS Linux machine, I typically got between about 985 and 1010 total dishes prepared.

However, there was typically a larger gap between the chefs who got to prepare a lot of dishes and those who didn't prepare as many. I'd say this is a consequence of the starvation risk in this solution. No chef was completely prevented from cooking, but some didn't seem to get to acquire their needed appliances as often.

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.c**, your **ordered.c**, your **takeAll.c** and your **kitchen.txt** under the assignment named HW4_Q3 on Gradescope.