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. One of the programming problems is in Java and the other is in C. 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. (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:

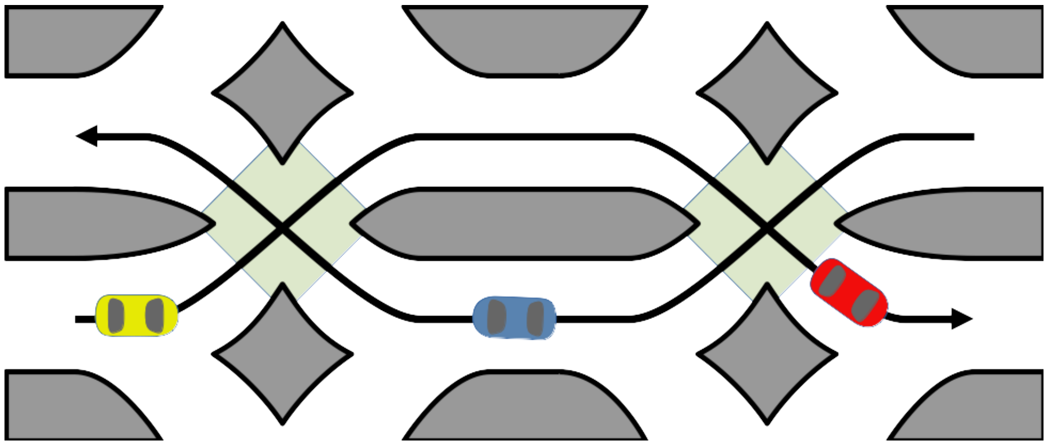
```
  ssh arc.csc.ncsu.edu
```

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. different).

```
[dbsturgi@login ~]$
```

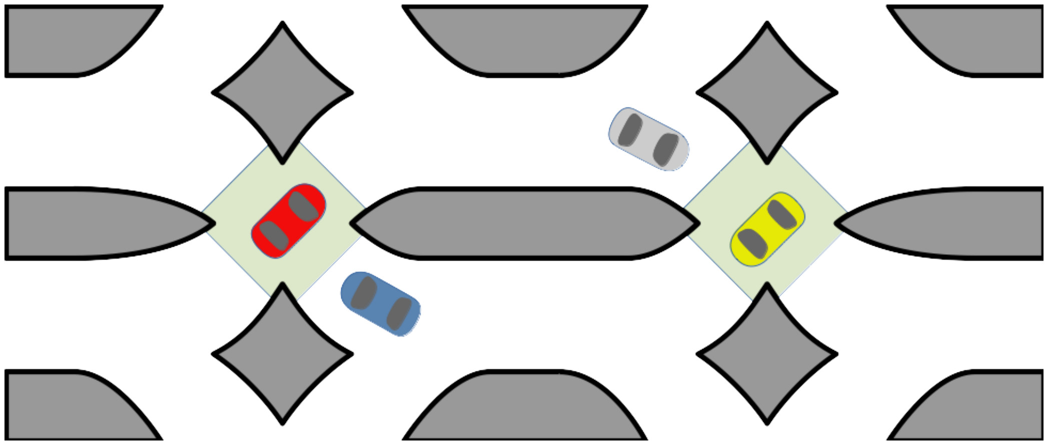
If this works, you should be in good shape for the next assignment. You can log out of the arc system by entering `exit`.

2. (40 pts) This problem is intended to give you some practice thinking about and implementing monitors. You are going to use POSIX mutex and condition variables to implement part of a simulation of a diverging diamond interchange, like the one shown below. If you've driven down Western Boulevard recently, you've had a chance to go through an interchange like this. They're building one at the intersection of Western and 440. For a smaller road, it temporarily exchanges the side of the road that you drive on so that it's easier to enter or exit from a larger road.



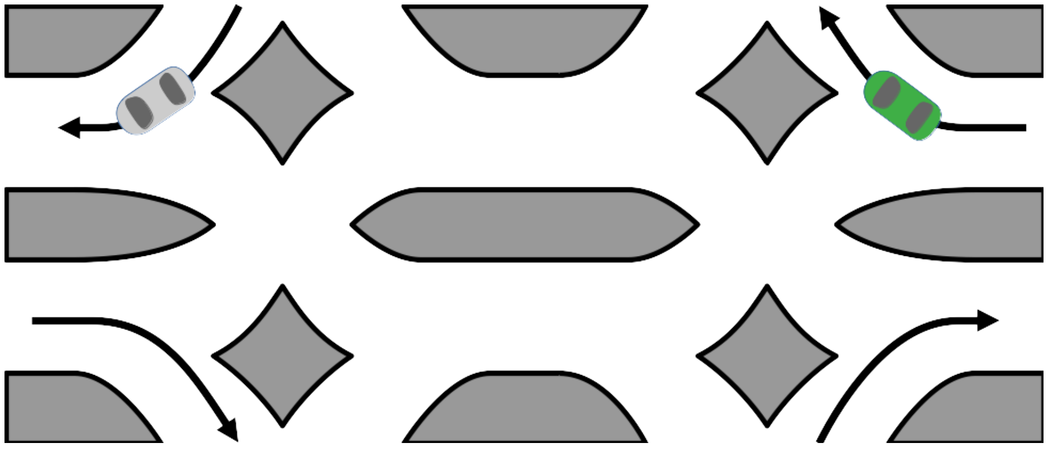
We'll imagine the intersection being organized like the one shown above. Cars can go through east-to-west or west-to-east. Cars going in opposite directions will cross over in two places, an east-side intersection and a west-side intersection. These are shown in pale green in the figure.

We need to make sure we never have multiple cars in either the west-side intersection or the east-side intersection going in different directions. Like the figure below shows, if the red car is in the west-side intersection going north-east, then the blue car must wait for it to pass through the west-side intersection before it can drive through going north-west. The rules are similar for the east-side intersection. If the yellow car is driving through going south-west, then the gray car will have to wait before it can enter this side of the intersection going south-east. It's OK if multiple cars are driving through one side of the intersection going in the same direction. Also, the two sides of the intersection are independent, so a car can be going north-east on the west side while another car is driving through the east-side intersection in a different direction.

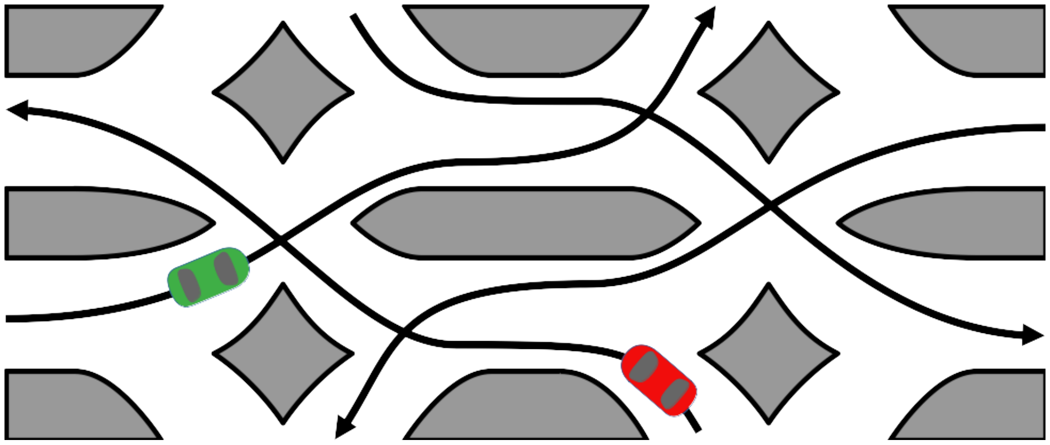


Normally, we would have a traffic light for each side intersection, permitting traffic in only one direction at a time. For this assignment, we'll create a monitor to solve the problem. Our monitor will decide when a car can enter the west-side and the east-side intersection. It will make sure cars wait to enter if there's already a car driving through in a different direction. It will also make sure cars eventually get a chance to drive through.

Cars may not drive all the way through east-to-west or west-to east. If they enter from the west, they may turn south before they get to either of the intersections. Or, they could enter from the east and then turn north before reaching either intersection. Likewise, they could enter from the south and turn east or enter from the north and turn west, avoiding either intersection. For our monitor, we'll ignore cars like this. They don't pass though either intersection so they don't interfere with other cars going through the interchange.

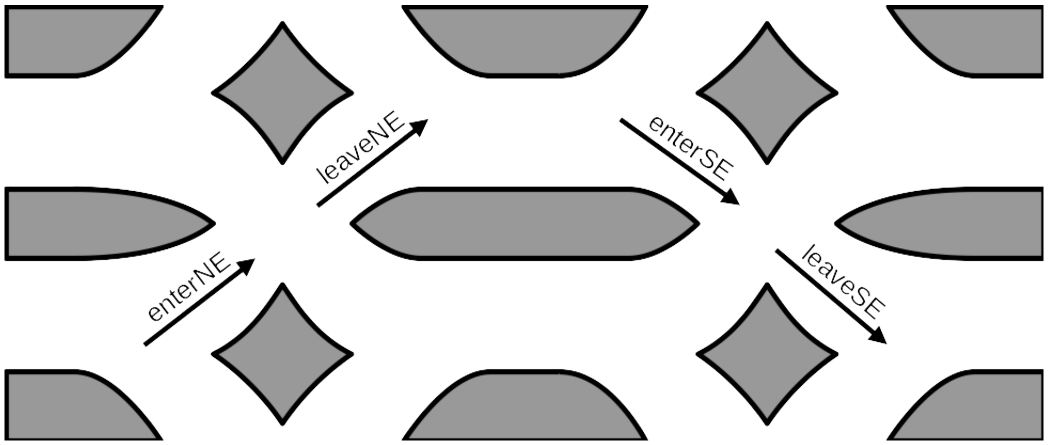


Cars could go through just one side of the interchange. They could enter from the south and then go west, they could enter from the north and then go east, they could enter from the east and then go south or they could enter from the west and then go north. The following figure shows the paths for cars like these. We will include cars like them in our simulation.

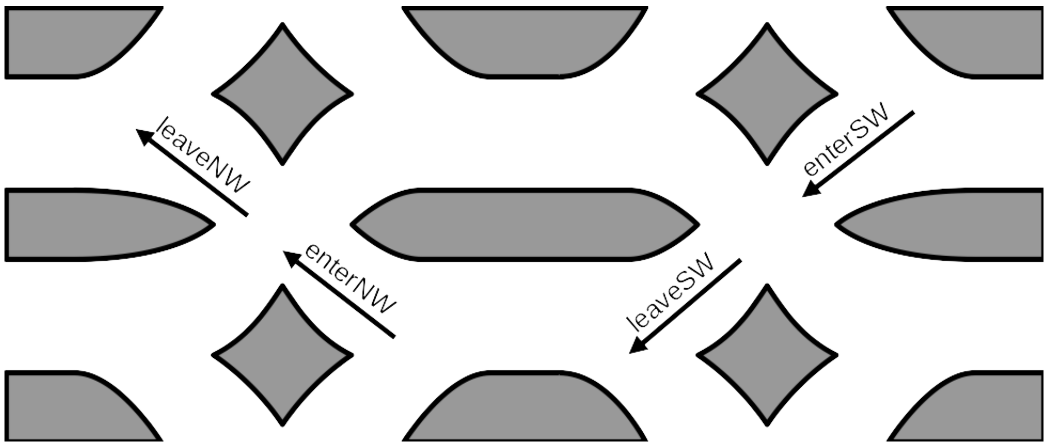


Interchange Management

In our implementation, the cars are implemented as threads. They call functions on our monitor as they pass through the west-side and east-side intersections. The monitor will have eight functions for doing this. A thread will call enterNE() when it wants to enter the west-side intersection going north-east, and it will call leaveNE() when it leaves this part of the interchange. There will be two more functions, enterSE() and leaveSE() for entering and leaving the east-side intersection going south-east.



For cars traveling west, we will have similar functions for regulating their trips through the interchange. They will call enterSW() and leaveSW() as they enter and leave the east-side intersection going south-west, and they will call enterNW() and leaveNW() as they enter and leave the west-side intersection going north-west.



To keep cars from hitting each other, the four enter functions will make threads wait when they need to, either because a car is already in the intersection going the other way or to make sure every car eventually gets a chance.

When a thread, T calls one of the enter functions, T will have to wait if the intersection is already occupied by one or more cars going in a different direction than the direction T is driving. It's OK if the intersection has multiple cars, provided they are going in the same direction. If T calls one of the enter functions and the intersection is empty or occupied by cars going in the same direction as T, then T can enter if there are no cars waiting to enter from the other direction.

If cars are waiting to enter an intersection from both directions, then the monitor will force them to take turns. For example, if a thread is driving trough the east-side intersection going south-east and other threads are waiting to go through in the south-west direction, then no other threads can enter going south-east until a south-west threads has had a turn. So, you could have some threads waiting to enter from both directions. The example program, `driver3.c`, demonstrates this. If, for example, there are waiting threads and the intersection is currently occupied by cars going south-east, then the next thread permitted to enter should be going south-west, but it doesn't matter which thread; any waiting south-west thread is OK. Then, a south-east thread will be given a turn, then a south-west thread and so-on. After the last thread heading a particular direction has left left the intersection, all remaining threads waiting to go in the other direction can enter. They no longer need to take turns if there are no threads waiting to go in the other direction.

The east-side and west-side intersections are independent. If, for example, a car wants to enter the east-side intersection, then it doesn't matter what cars are in the west-side intersection or waiting to enter it.

Monitor Implementation

The header file `diamond.h` defines the interface you will implement. In `diamond.c`, you'll implement the monitor's functions. In this implementation file, 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 a few condition variables, for making threads wait when they need to. You can also have other variables to keep up with the current state of the monitor (e.g., what types of cars are in the two intersections and what cars are waiting). 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.

Your monitor will provide the following functions:

- `void initMonitor()`
  This function initializes the monitor, allocating any heap memory it needs and initializing it state where necessary.
- `void destroyMonitor()`
  This function is called after we're done using the monitor. It should free any resources used by the monitor.
- `void enterNE( char const *name )`
  `void enterNW( char const *name )`
  `void enterSE( char const *name )`
  `void enterSW( char const *name )`
  A thread calls this function when it wants to enter an intersection in the interchange. It will make the thread wait until it can enter the intersection. Before returning, the function will print one of the following messages (depending on the direction), where *name* is the name of thread passed in as the function parameter.

  Entering NE: *name*
  Entering NW: *name*
  Entering SE: *name*
  Entering SW: *name*

- `void leaveNE( char const *name )`
  `void leaveNW( char const *name )`
  `void leaveSE( char const *name )`
  `void leaveSW( char const *name )`
  A thread calls this function when it is ready to leave the intersection. It won't need to make the caller wait, but it may permit a different thread to enter the intersection. The function will print one of the following messages (depending on the direction), where *name* is the name of thread passed in as the function parameter.

  Leaving NE: *name*
  Leaving NW: *name*
  Leaving SE: *name*
  Leaving SW: *name*

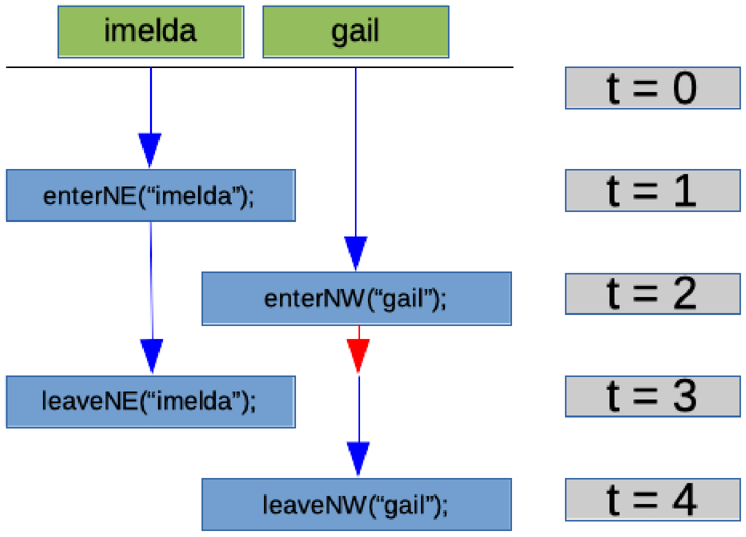
Driver Programs

We're providing four test driver programs to help you exercise your monitor, driver1.c, driver2.c, driver3.c and driver4.c. There are also some 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 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 diamond.c driver1.c -o driver1 -lpthread
```

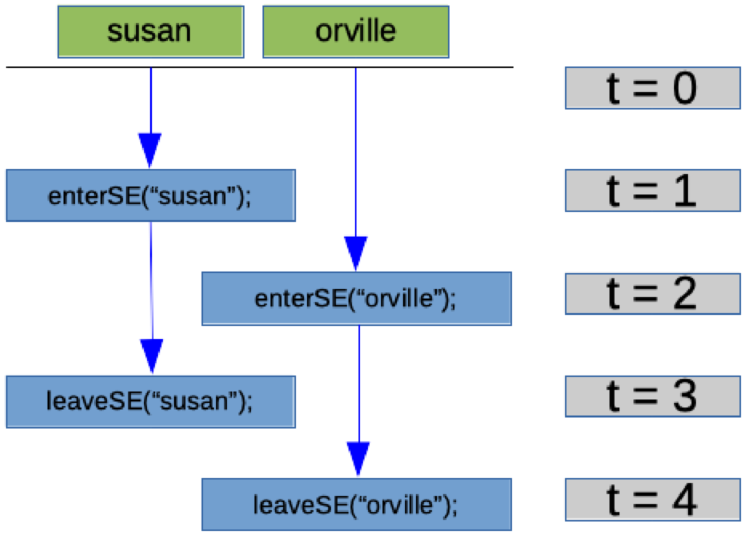
The driver1.c program is designed to make sure two threads can't be in the west-side intersection at the same time if they're going in different directions. The following figure shows what it does. The two vertical lines show the actions of two threads, imelda and gail. Blue boxes show when these threads 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). Here, the imelda thread waits a second, then enters going north-east. A second later, gail tries to enter the same intersection going north-west. Gail has to wait until imelda leaves at time 3. Then, gail can enter, so she returns from the enterNW() function, drives across the intersection and leaves a second later.



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).

```
Entering NE: imelda          (at 1 second)
Leaving NE: imelda           (at 3 seconds)
Entering NW: gail            (at 3 seconds)
Leaving NW: gail             (at 4 seconds)
```

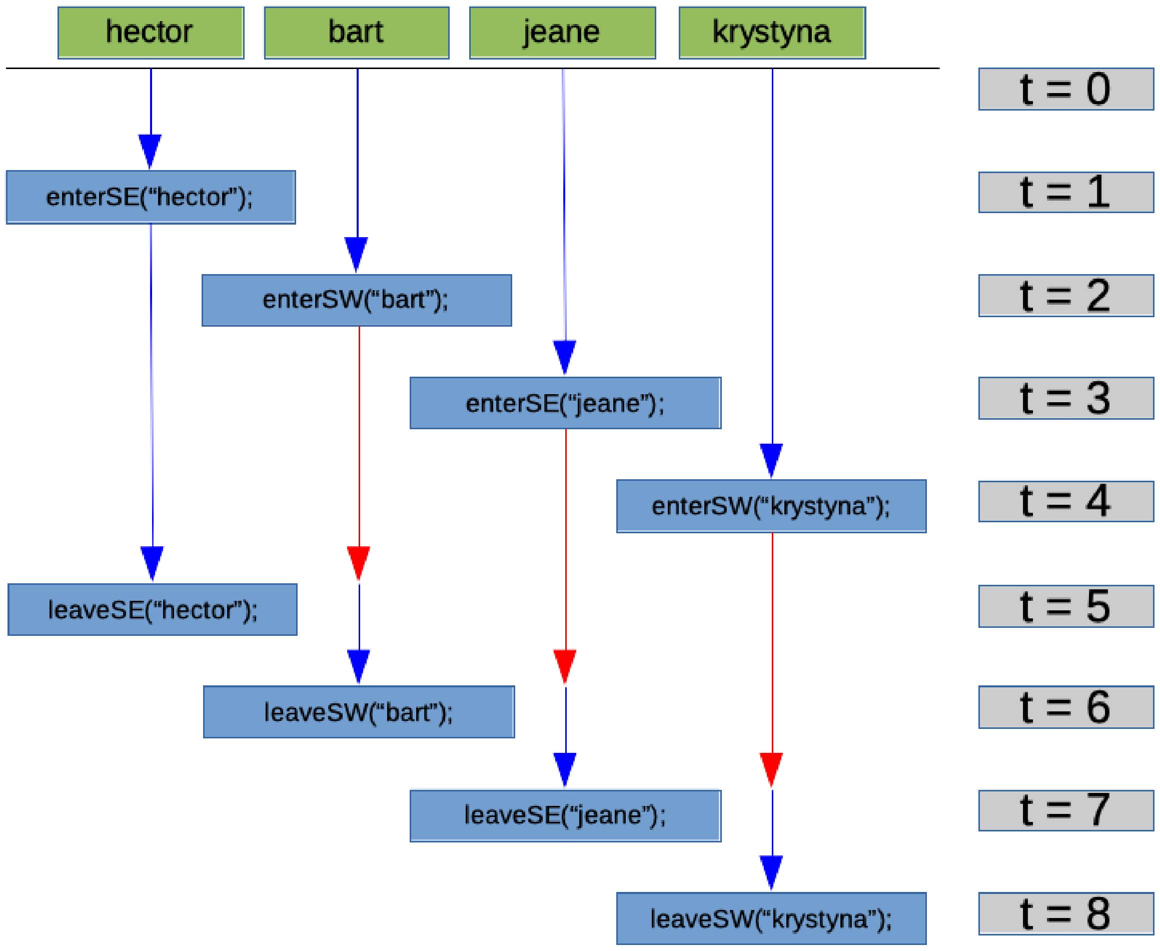
The driver2.c program checks to make sure cars can occupy the east-side intersection at the same time if they're going in the same direction. Susan enters going south-east a second after the program starts. A second later, orville is able to enter going in the same direction. Each car takes two seconds to drive through the intersection then leaves.



If you build and run your monitor with driver2.c, you should get output like the following:

```
Entering SE: susan            (at 1 second)
Entering SE: orville          (at 2 seconds)
Leaving SE: susan             (at 3 seconds)
Leaving SE: orville           (at 4 seconds)
```

The driver3.c program makes sure threads take turns if they want to go through the same intersection in different directions. Hector enters the east-side intersection after a second and then takes four seconds to drive through going south-east. While hector is in the intersection, bart starts waiting to go through this intersection going south-west. After this, two more threads line up waiting to go through the intersection in each direction. After hector leaves, they start taking turns. Hector was going south-east, so a south-west thread gets the next turn. It could be either bart or krystyna; the monitor isn't required to admit threads in the order they arrived. Let's say bart gets to go next. Bart takes a second to drive through, then a south-east thread (jeane) gets the next turn. After this, a south-west thread (krystyna) gets a turn.



If you build and run your monitor with driver3.c, you should get output like the following. In your output, it's OK if bart and krystyna are swapped, since either of them could have been admitted after hector leaves.

```
Entering SE: hector          (at 1 second)
Leaving SE: hector           (at 5 seconds)
Entering SW: bart            (at 5 seconds)
Leaving SW: bart             (at 6 seconds)
Entering SE: jeane           (at 6 seconds)
Leaving SE: jeane            (at 7 seconds)
Entering SW: krystyna        (at 7 seconds)
Leaving SW: krystyna         (at 8 seconds)
```

The driver4.c program is a stress test. It uses 24 threads to drive through the interchange over and over in different directions, with random timing for each stage of their trip. 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 8 threads are going all the way through the interchange east-to-west or west-to-east. They should all get to make about the same number of trips. As you can see from the partial output below, I got about 730 trips each through the interchange. The remaining 16 threads are just passing through the east-side intersection or the west-side intersection over and over. They will get to make more trips. I got about 1460 each on one of the remote EOS Linux machines.

```
Entering SW: Peter
Entering NE: Bradly
Entering NE: Arturo
Leaving NE: Arturo
Leaving NE: Bradly
Entering NW: Boris
Leaving SW: Peter
Leaving NW: Boris
Entering NE: Verona
Leaving NE: Verona
Entering NE: Desmond
Entering NE: Enedina
Leaving NE: Desmond
Entering SW: Kent
Leaving NE: Enedina
Leaving SW: Kent
Entering SW: Georgiana
Leaving SW: Georgiana
Entering SW: Rosanna

... lots of output omitted ...

Leaving NW: Orval
Entering NW: Hollie
Leaving NW: Hollie
Daisy drove through 716 times
Enedina drove through 721 times
Bradly drove through 727 times
Marcella drove through 734 times
Orval drove through 729 times
Hollie drove through 738 times
Amos drove through 730 times
Peter drove through 736 times
Desmond drove through 1453 times
Verona drove through 1468 times
Leonel drove through 1450 times
Arturo drove through 1498 times
Ellie drove through 1476 times
Boris drove through 1464 times
Lorretta drove through 1442 times
Monroe drove through 1473 times
Irene drove through 1460 times
Jeffery drove through 1498 times
Lucius drove through 1477 times
Jess drove through 1498 times
Kent drove through 1519 times
Georgiana drove through 1437 times
Lily drove through 1481 times
Rosanna drove through 1479 times
Total: 29404
```

Submitting your Work

When you're done, submit just your monitor implementation in **diamond.c**. 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.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. They do this over and over for 10 seconds; they 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.

   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, make a copy the program named `Global.java`. You'll also need to change the name of the main class inside the source file to get this to work with the new filename. 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). With just one lock, our solution should be free of potential deadlocks.

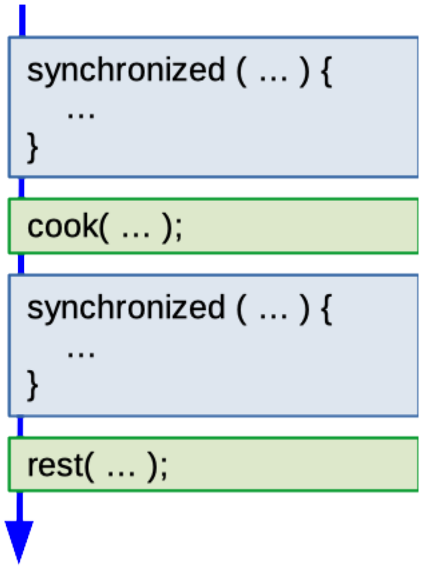
   This solution should work, but it's not ideal. It prohibits concurrent cooking by chefs, even if they don't need any of the same appliances. When I tried this technique, I only got between about 280 and 320 total dishes prepared on one of the remote Linux machines. There was some variation from execution to execution.

   (b) Instead of forcing the chefs to cook just one at a time, we should be able to prevent deadlock by 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 between about 830 to 860 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.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 good bit better than either of my other two solutions. On an EOS Linux machine, I typically got between about 1000 and 1050 total dishes prepared.

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**.