CS4410 Homework 3

Due 9am Tuesday February 24

100 points as shown below

Short answer questions, 5 points each (no answer should need more than two or three sentences).

1. Name three ways in which the processor can transition into kernel mode.

   Interrupt, system call (trap), exceptions

2. What needs to be saved and restored on a context switch between two threads in the same process?

   Registers

3. What is the difference between deadlock prevention and deadlock avoidance? What category does Bankers algorithm fall in and why?

   As used in cs4410 (and in the book), deadlock prevention is a term that includes allowing a deadlock to occur but providing some mechanism to "undo" the damage, such as releasing locks or rolling back a computation. Deadlock avoidance completely avoids deadlocks. The Banker's Algorithm never allows a deadlock to arise, so it falls into the latter category.

4. Suppose you have a concurrent system with locks: Lock.acquire() blocks until the Lock is available and then acquires it. Lock.release() releases the Lock. There is also a Lock.isFree(), that does not block and returns true if the Lock is available; otherwise, returns false.

   Is the following code guaranteed never to block?

```
if(Lock.isFree())
      Lock.acquire();
```

   A thread could preempt execution in between the if test and the Lock.acquire operation, hence even if the lock was free when the if executed, it might not be free anymore by the time Lock.acquire is executed.

Ithaca's one-lane bridges

Write a class in Java to coordinate cars trying to cross one-lane bridges. The class should be instantiated once per bridge. A car wishing to cross the bridge will call bridge.cross(int direction), where direction is either 0 or 1, representing the two possible directions. This method blocks the car until it is safe for it to cross the bridge. Having finished crossing, the car will call bridge.done(). *Only one car can be on the bridge at a time.*

5. (25 points) Code a solution that solves the problem without worrying about fairness.

```
public class Bridge {
    Mutex lock;
    public void cross(int direction)
```

```
        {
                Lock.acquire();
        }

        public void done()
        {
                Lock.release();
        }
}
```

6. (25 points) Give a solution that will allow a maximum of 3 cars in a row to cross in any given direction and then switches to the opposite direction. It should only behave this way when there is actually “two way traffic” – your solution should not *assume* that there are cars trying to cross in both directions.

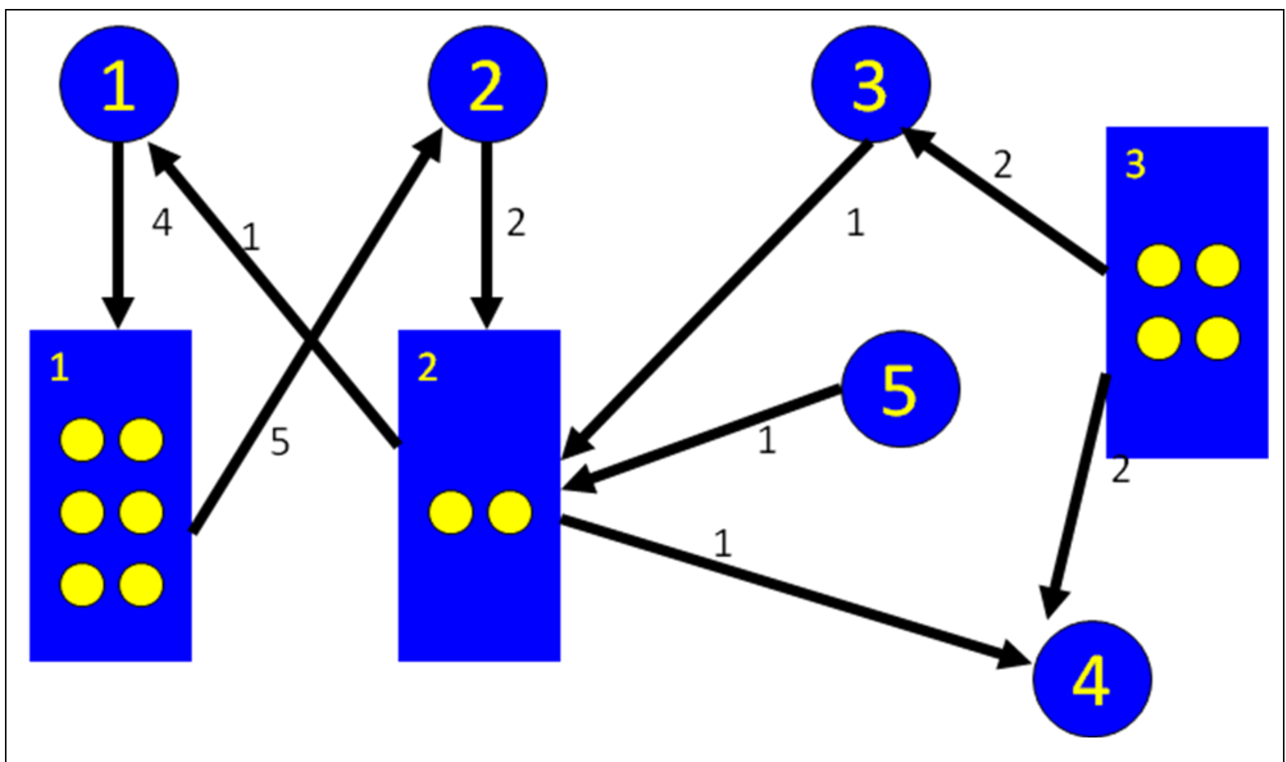
You can solve this in lots of ways. Our solution is actually based on the critical section code above, with some logic added to handle the various rules. The basic idea here is to have each driver pull up to the bridge and then sort of “nap” for 100ms at a time, periodically waking up and trying to grab the bridge lock. But when you hold the lock you don’t necessarily cross – instead you follow the policy, which is to let 3 cars pass in each direction, if there actually are 3 cars trying to pass. Our code counts cars waiting (like being able to see cars on the other side). Cur_dir is the direction that the last car crossed in (-1 initially) and in_a_row counts how many cars have passed in a row, but only if someone was actually waiting to cross in the other direction.

```
    public class Bridge {
        Mutex lock;
        Int curr_dir = -1, in_a_row = 0;
        Int[] waiting = new int[2];
        Boolean BridgeBusy = FALSE;

        public void cross(int dir)
        {
                lock.acquire();
                // Counts cars waiting in each direction
                waiting [dir]++;
                while(BridgeBusy ||
                         (waiting[dir^1] > 0 && (dir == cur_dir? in_a_row == 3: in_a_row < 3)))
                {
                        // Let someone go in the other direction if:
                        //   there is someone waiting, and
                                // a) the last car passed in my direction, and was the 3rd in a row, or
                                // b) it passed in the other direction but wasn't the 3rd in a row yet
                        lock.release();
                        Thread.sleep(100ms);
                        lock.acquire();
                }
                // I'm off!  And so I'm no longer a waiting car…
                waiting[dir]--;
                // Also make note of my direction
                if(cur_dir != dir)
```

```
            {
                // reset the in-a-row counter when we switch directions
                in_a_row = 0;
                cur_dir = dir;
            }
            // only count cars while someone is waiting in the other direction
            if(waiting[dir^1] > 0)
                in_a_row++;
            // Now I'll cross the bridge
            BridgeBusy = TRUE;
            Lock.release;
        }

        public void done()
        {
            // On the other side I release the lock and the next car can have at it
            Lock.acquire();
            BridgeBusy = FALSE;
            lock.release();
        }
}
```

7. (30 points) For the resource-wait graph shown below, either show a graph-reduction sequence that fully reduces the graph, or prove that the system is deadlocked by showing us an irreducible subgraph.

After reducing in the order P3, P4 and P5, we are left with an irreducible subgraph consisting of the resources and P1 and P2. This proves that the system is deadlocked.



Notation reminder: a process Pi is a blue circle with the number i inside, in yellow. Resource Rj is a blue box, with the value of j in the top left corner. The maximum total number of units of a resource is shown as a number of yellow circles. For example, below, there are a total of 4 units of R3. An arrow from a process to a resource, labeled, means the process is requesting that many units of the resource: P2 "wants" 2 units of R2. An arrow from the resource to a process means that the process owns that many units of the resource: P3 and P4 each own 2 units of R3. Notice that to figure out how many units of a resource are available, you'll need to sum up the labels on out-edges from that box and subtract this from the number of units shown inside the box. For example, all of R3 is currently allocated.