Fall 2004 CS414 Prelim 1

1. The **Sim City Smoking Ban** problem:

In the Sim-City community Woobish most people smoke, but the laws of Sim City require that non-smokers be protected from passive smoke. So Woobish has a law under which people can only smoke in a bar if everyone in the bar is ok with it. If a designated non-smoker is in the bar, nobody can light up.

Assume that customers are modeled as threads:

- *smoking threads* call **enter_bar(true)** before entering the bar (the flag is *true* to indicate that the thread is a smoker), then repeatedly call **want_smoke()** before lighting up, and **done_smoking()** after they finish, and finally call **leave_bar(true)** when leaving the bar.
- *non-smoking threads* call **enter_bar(false)** to enter (the flag is *false* to indicate a non-smoker), and **leave_bar(false)** on its way out.

a) Write the code for a monitor implementing these rules. You can assume that periodically, there won't be any non-smokers. This would make sense, at least in the first few years after Woobish passes the law, since non-smokers tend to leave the bar quickly (you would too, with all those angry nicotine-crazed smokers glaring at you!)

```
// First solution: non-smokers can enter while people
// are smoking

public class SmokingRules: monitor {
     condition want_to_smoke;
     int non_smokers_in_bar = 0;

     Public void enter_bar(Boolean smoker)
     {
          if(smoker == false)
               ++non_smokers_in_bar;
     }

     Public void leave_bar(Boolean smoker)
     {
          if(smoker == false)
               if(--non_smokers_in_bar == 0)
                    want_to_smoke.signal();
     }

     Public void want_smoke()
     {
          if(non_smokers_in_bar)
               want_to_smoke.wait();
          // Wake next waiting smoker, if any
```

```
          want_to_smoke.signal();
     }

     Public void done_smoking()
     {
     }
}

// Second solution: in this one, they wait if
// someone is smoking before entering.  This is a
// better solution, but because of the way the problem
// was worded, we'll accept the simpler one as correct
// too.
public class SmokingRules: monitor {
     condition want_to_smoke, want_to_enter;
     int non_smokers_in_bar = 0;
     int smoker_count = 0;

     Public void enter_bar(Boolean smoker)
     {
          if(smoker == false)
          {
               if(smoker_count > 0)
                    want_to_enter.wait();
               ++non_smokers_in_bar;
               want_to_enter.signal();
          }
     }

     Public void leave_bar(Boolean smoker)
     {
          if(smoker == false)
                if(--non_smokers_in_bar == 0)
                    want_to_smoke.signal();
     }

     Public void want_smoke()
     {
          if(non_smokers_in_bar)
               want_to_smoke.wait();
          ++smoker_count;
          // Wake next waiting smoker, if any
          want_to_smoke.signal();
     }

     Public void done_smoking()
     {
```

```
            if(--smoker_count == 0)
                want_to_enter.signal();
        }
    }
```

b) Define the following properties in a line or two each. For each, tell us if the property was satisfied by your solution under the assumptions of part (a), and if not, why not.

a. Mutual exclusion. *Definition: We'll say that two threads conflict if errors can arise should they concurrently enter some code segment, or concurrent access some resource. Mutual exclusion involves guaranteeing that conflicting threads will be synchronized so that conflicts cannot occur.*

The first solution given in part (a) doesn't provide mutual exclusion because a non-smoker can enter while someone is smoking. However, the second solution shown above has additional logic, so that a non-smoker wishing to enter the bar would wait until all smokers stopped. That would provide for mutual exclusion between smoker threads and non-smoker threads.

An interesting observation is that there can be multiple smokers and multiple non-smokers: mutual exclusion doesn't mean "one at a time", it means "there can't be concurrent activity by members of two conflicting sets of threads." Thus one could have two smokers, or two non-smokers, but not some of each.

b. Progress. *Definition: This is a guarantee that when threads are waiting to obtain mutual exclusion, some thread will eventually enter the critical section or take control of the shared resource. Our solution guarantees progress because if a smoker is waiting, either more non-smokers will enter (nothing stops them), or eventually all the non-smokers will leave and in this case the smokers can smoke. In either case some threads can make progress.*

c. Bounded waiting. *Definition: This is a guarantee that if a thread is waiting, the length of its weight will be bounded – eventually, it will be able to make progress. (Progress is a statement that some thread will make progress, but not a guarantee that every single thread will do so). Our solution can't guarantee progress because if a unlikely event occurs and an endless stream of non-smokers shows up, smokers will wait indefinitely in want_smoke(). Similarly, the second solution can leave a non-smoker stuck at the door if smokers endlessly light up.*

2. **Atomic Instructions**

A computer has a floating point instruction called **INV**. Much like a hand-held calculator **INV** takes a value in memory or a register, computes **1.0/X** and stores the result back into the same place. At the same time it clears the overflow bit in the floating point status register. If **X** was zero, **INV** stores a 1.0 into **X** and sets the overflow bit in the floating-point processor status register. *This occurs atomically.*

Assume that someone has implemented a function callable from high level languages; if you call **Boolean INV(ref float X)** the function uses this instruction to invert **X** (the argument is by reference). The call returns the overflow bit: true if overflow occurs and false if not.

Show how to implement CSEnter and CSEXit using INV.

```
Float X = 0.0;

CSEnter()
{
        while(INV(ref X) != true) continue;
}

CSExit()
{
        X = 0.0;
}
```

b) What might happen if INV was replaced by this C# code (compiled normally, not using special atomic instructions)?

```
boolean INV(ref float X)
{
     if(X != 0.0)
     {
          X = 1.0/X;
          return(false);
     }
     else
     {
          X = 1.0;
          return(true);
     }
}
```

If your algorithm from part (a) is still correct when this code is compiled with it, explain why. If not, explain why not. A sentence or two will be fine!

If we compile with this segment of code, all sorts of problems can occur. For example, thread a could test to see if X is 0.0 and see that it is, but then suspend, and thread b could also test X. In this case both a and b return true and both get past CSEnter concurrently. Thus with the atomic instruction, the solution given in part (a) works, but with the equivalent non-atomic segment of code, the solution would not be correct.

3. **Bakery Algorithm.**

Here's the bakery algorithm, with line numbers. *You may assume that turn never overflows. This question has nothing to do with overflow.*

```
[0]  CSEnter(i):
[1]       chosing[i] = true;
[2]       turn[i] = max(turn[0], . . ., turn[N])+1;
[3]       chosing[i] = false;
[4]       for(j = 0; j < N; j++) {
[5]            while(chosing[j] == true) continue;
[6]            while(turn[j] > 0 &&
                     (turn[j],j)<(turn[i],i)) continue;
[7]       }


[8]  CSExit(i):
[9]       turn[i] = 0;
```

Suppose that N = 5 and that processes P0 and P3 try to enter the critical section simultaneously while process P1 is already inside. Process P1 has turn[1] = 2. By "try to enter simultaneously" we mean that both processes have called CSEnter and neither has executed line 1 yet. Processes P2 and P4 are doing something else.

a) Can we be sure that process P0 will enter the critical section before process P3? Explain.

We have no control over the scheduler, hence P3 could execute lines 1-3 before P0 is able to execute line 1. In such a case P3 would be certain to enter the critical section before P0. So the answer is ***no.***

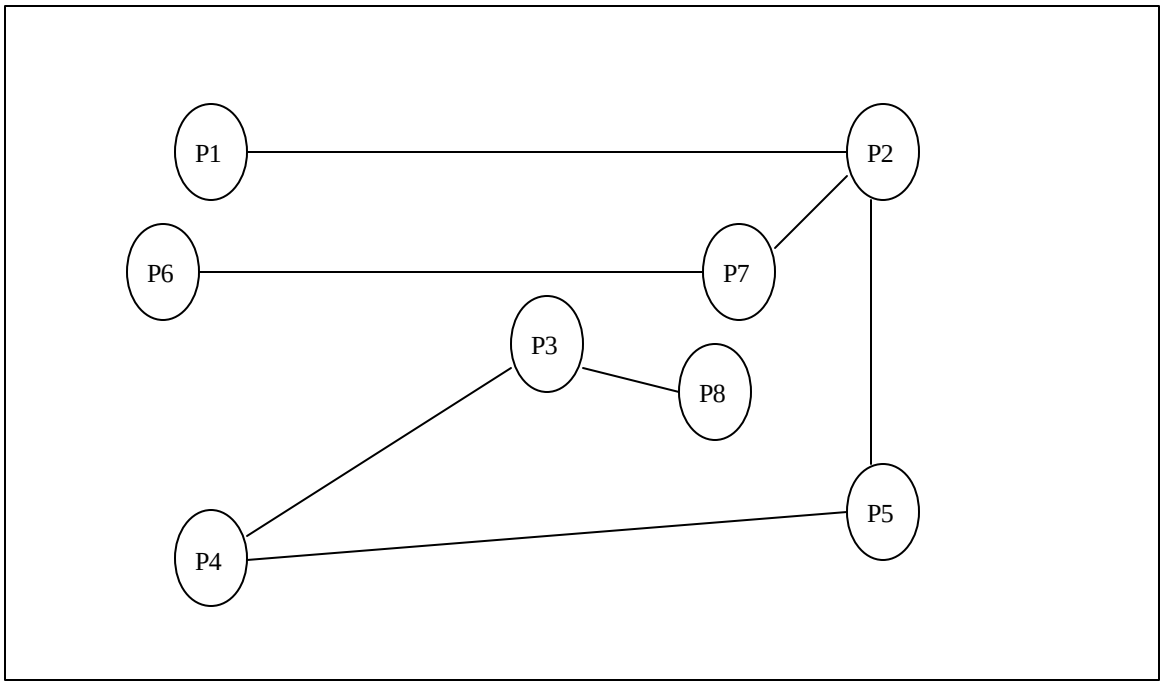
b) Suppose that process P0 picks turn[0] = 3 and process P3 picks turn[3] = 4. Could a situation arise in which P0 would wait for process P3 to enter the critical section and then exit the critical section before P0 does? Explain why this can't happen, or how it could.

The behavior of the code depends on when the two processes pick these values. As stated, it sounds as if P3 picked its turn value after P0 did, and in that case P0 gets in first. But the situation is really more complicated, because we were only told that the two processes are poised at line 1. In this case you could imagine that P3 gets to run first, picks turn[3] = 4, gets in, then the turns go down to 0, and then they count up to 2 again. At that point perhaps P0 finally gets to execute. Now P0 would have turn 3, yet would be entering after P3. Thus, it can happen that P3 enters before P0 does, but only if P0 was delayed by the scheduler before executing line 1. Once P0 has executed line 1, if these were the values picked, P0 will certainly enter before P3.

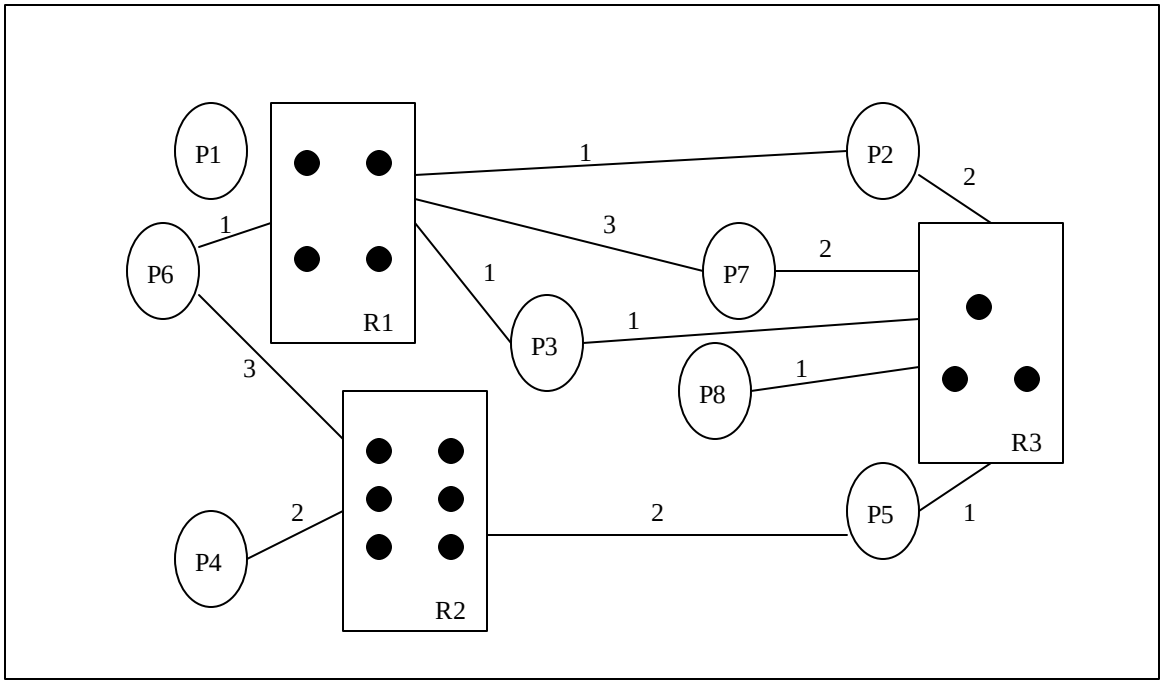
The graders are concerned that some students assumed that by "pick values" we implied that the processes were both at line 3. Answers that assumed this will also be graded as correct.

4. Deadlock Detection

Below is one process-wait graph and one resource wait graph, depicting <u>different</u> system states. For each graph, if the state shown is not a deadlock, list the order in which we can fully reduce the graph (e.g. P_4, P_1, …., meaning "first erase the node for process P_4, then the one for P_1, etc). If a deadlock is present, prove this by showing us a fully-reduced but non-trivial component.



The graph can be reduced in the order P4, P3, P8, P5, P2, P7, P6, P1 (this is just one of about a half dozen equivalent orders). This implies that there is no deadlock in the state shown.



A reduction order is P1, P4, P5, P8, P3, P6, P2, P7. This fully reduces the graph, so no deadlock is present.

5. The village well is large enough so that four people can draw water at a time. However, if more than four try to do so a fight could break out.

Give code for two procedures, **WaitMyTurn(int pid)** and **Finished(int pid)** to enforce these rules first using semaphores. The process id is an integer from 0... N-1, where N is the population of the village.

```
Semaphore MaxPeople = 4;

WaitMyTurn(int pid)
{
        MaxPeople.wait();
}

Finished(int pid)
{
        MaxPeople.signal();
}
```