CS 414 Fall 2004: Homework 3 (due Sept. 28)

Please submit solutions using CMS

1. Write a monitor solving the *toll plaza* problem. Car "processes" approach the toll plaza. The plaza consists of a set of *b* toll booths, each of which has a state, `tollbooth[b].state` with values OPEN and CLOSED and a current line length `tollbooth[b].linelen`. A car scans the booths and picks a booth and then asks to be added to the line; this causes the line length to be incremented (method `tollbooth[b].wait()`) and leaves the car process waiting for its turn. When a process reaches the front of the line it pays (`tollbooth[b].pay()`) and then can drive away. You should implement the *tollbooth* class and show us code that might be executed by car *k*. If you find it necessary to change the proposed interface you have our permission to do so.

2. Here's code for the Bakery Algorithm:

```
[1]  CSEnter(i):
[2]       chosing[i] = true;
[3]       turn[i] = max(turn[0], . . ., turn[N])+1;
[4]       chosing[i] = false;
[5]       for(j = 0; j < N; j++)
[6]       {
[7]            while(chosing[j]) continue;
[8]
     while(turn[j]>0&&(turn[j],j)<(turn[i],i))continue;
[9]       }
          . . . process i can enter the critical section once
it gets here . . .

[10] CSExit(i):
[11]      turn[i] = 0;
```

(a) Suppose that process 2 is trying to enter the critical section and that N=10. When line 8 is executed, suppose that turn[0] is equal to zero. But later when line 8 is executed for turn[4] suppose that we need to spend a long time in the while loop. Why isn't it necessary to start the "for" loop again from j=0 and re-check the processes that previously had turn[j] equal to zero, like process 0? Try and express your answer like a "proof" if you can (e.g. "prove" that you don't need to revisit a value of j once you have already done so).

(b) Modify the code to create a "Bounded Bakery Algorithm" in which turn values can never exceed a fixed upper limit, MAX (you can assume that MAX is larger than N).

(c) We showed that the original Bakery Algorithm satisfied Mutual Exclusion, Progress and Bounded Waiting. Does your modified algorithm satisfy these properties? Prove that it does, or explain why you can't provide one or another of these guarantees. *Note: we're not looking for a formal proof, but try and be as clear and specific as you can. A solution that does satisfy as*

many of the guarantees as possible is preferred to one that tosses one or another out the window. All solutions must still satisfy the mutual exclusion property.

3. Due to a last minute funding cuts, Ellison University's dorms only have a single shared bathroom on each floor, even though the dorms are coed. Using semaphores, solve the *unisex bathroom* problem. Specifically, design procedures `GuyEnters(), GuyLeaves(), GirlEnters(), GirlLeaves()` such that: (a) there are never more than 3 people in the bathroom, (b) if a guy is in, girls can't enter and vice-versa, and (c) If a guy is waiting and girls are inside, the next person to get in will be a guy, and vice versa (all of these 3 properties should hold at the same time).

4. Suppose that a recursive procedure contains a critical section and, to make matters worse, that the recursion sometimes (not always) occurs from within the critical code. For example, here's a skeleton of how such a procedure might look:

```
void recurse(args)
{
   if(some condition)
        recurse(args');
   else if(some other condition)
        return;
   else
   {
        CSEnter(my-process-id);
        recurse(args'');
        CSExit(my-process-id);
   }
}
```

(a) Suppose that CSEnter/Exit are implemented using the Bakery Algorithm. What issues might arise?

(b) Same question but now answer for the case where CSEnter/Exit are implemented with semaphores.

(c) Show how recurse can be modified to avoid these potential problems.