# 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.

```
public class TollBooth: monitor {
        enum {OPEN, CLOSED} state = OPEN;
        int linelen = 0;
        condition WaitQueue;

        public void wait()
        {
                if(linelen++ == 0)
                        return;
                WaitQueue.wait();
        }

        public void pay()
        {
                … move some money …
                –linelen;
                WaitQueue.signal();
        }
}
TollBooth[] tollbooth = new TollBooth[B];

// Code for car k:
int which_booth = 0;
for(int  b = 1; b < B; b++)
        if(tollbooth[b].linelen < tollbooth[which_booth].linelen)
                which_booth = b;
tollbooth[b].wait();
tollbooth[b].pay();
```

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).

*If we've already looked at process 0, we know that any turn value process 0 could pick will be larger than the turn value that was picked by process 2, since process 0 wasn't in the process of chosing (we checked when we initially looked at j=0). Thus process 0 will "see" turn[2] and hence pick a value for turn[0] greater than that of turn[2].*

(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).

*One solution works by modifying lines [2]-[4] this way:*

```
[*]    do
[*]    {
[*]        turn[i] = 0;
[*]        if(max(turn[0], . . ., turn[N]) == MAX)
[*]                while(max(turn[0], . . ., turn[N]) != 0)
continue;
[2]        chosing[i] = true;
[3]        turn[i] = max(turn[0], . . ., turn[N])+1;
[4]        chosing[i] = false;
[*]    }
[*]    while(turn[i] >= MAX);
```

*that is, we keep picking a turn value until the value we pick is small enough. The reason for resetting turn[i] to 0 is so that the number won't just keep ratcheting to bigger and bigger values.*

(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.

*If we get past the loop, turn[i] will be larger than any existing turn value and yet smaller than MAX, hence the code continues to satisfy Mutual Exclusion. It no longer guarantees progress or bounded waiting, because scenarios can arise in which all the processes in the system get stuck in the chose-a-value loop, although this is very unlikely to occur.*

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).

*This is basically Readers and Writers but with a tricky change to deal with the "fairness" issue. The code is a bit easier to write as a monitor, in fact, and you won't see such a hard semaphore problem on the prelim. But we wanted you to have a chance to really dig your teeth into a semaphore synchronization problem, and this is a classic. The code for Guys and for Girls is symmetric.*

```
Semaphore Mutex = 1, TheLine = 1, OpenTheDoor = 1, RoomLimit = 3;
Integer  GuyCount = 0, GirlCount = 0;

GuyEnters()
{
    wait(RoomLimit); // This is sort of "separate" logic to respect room capacity limit
    wait(TheLine);
      // Only one person gets here at a time!  Others line up on "TheLine"
    wait(Mutex);
    // Mutex needed because of people leaving
    if(GuyCount++ == 0 || GirlCount > 0)
    {
            signal(Mutex);
            // If we get here, the bathroom is currently empty (or will be soon)
            // Notice that while waiting on this semaphore anyone else who shows
            // up gets stuck waiting on TheLine and won't get past that spot until this
```

```
            // person (who could be a guy or a girl, since this section of the code looks
            // the same in both cases)  is actually in the bathroom.
            wait(OpenTheDoor);
    }
    else
            signal(Mutex);
    signal(TheLine); // Now we can let another person get off TheLine
}

GuyLeaves()
{
    signal(RoomLimit);
    wait(Mutex);
    if(--GuyCount == 0)   // Think of this next line as "last person out closes the door"
            signal(OpenTheDoor);
    signal(Mutex);
}

GirlEnters()
{
     wait(RoomLimit);
    wait(TheLine);
     wait(Mutex);
    if(GirlCount++ == 0 || GuyCount > 0)
    {
            signal(Mutex);
            wait(OpenTheDoor);
    }
    else
            signal(Mutex);
   signal(TheLine);

}

GirlLeaves()
{
    signal(RoomLimit);
    wait(Mutex);
    if(--GirlCount == 0)
            signal(OpenTheDoor);
    signal(Mutex);
}
```

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?

*We'll try to reenter the CSEnter code when we're already in the critical section. This can result in violations of the Mutual Exclusion property because process i will change its turn value to a larger one when it is already inside the critical section – some other process could then get in.*

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

*In this case, process i can hang and the system as a whole would deadlock.*

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

```
void recurse(args, has_mutex)  // invoke with has_mutex = false
{
   if(some condition)
        recurse(args', has_mutex);
   else if(some other condition)
        return;
   else
   {
        if(has_mutex == false) CSEnter(my-process-id);
        recurse(args'',  true);    //  recursion  with  mutual
        exclusion:
        if(has_mutex == false) CSExit(my-process-id);
   }
}
```