EECS 338 Assignment #3 solutions: *Concurrent Programming Algorithms #1 2011*

On the due date (Feb. 18th), submit your solution document (pdf or txt only please; no word documents!) to blackboard-assignment #3. To help you prepare your assignment document, you can download a word document of this assignment from blackboard-assignment #3.

(15 pts) **(1)** Consider the Bakery algorithm given below.

```
Pi: repeat
       choosing[i] := true;
A ---- number[i] := max (number[0], number[1], ..., number[n-1])+1;
       choosing[i] := false;
       for j:=0 to n-1 do begin
          while choosing[j] do skip;----------------------(*)
          while (number[j], j) < (number[i], i)
                    and number[j] ≠ 0 do skip;
       endfor
       Critical Section
       number[i] := 0;
       Remainder Section
    until false;
```

Initially, all choosing values are false, and all number values are zeroes.

(a) Assume that the line (*) is removed. Describe a scenario in which the algorithm does not work (i.e., it does not solve the critical section problem). Briefly, explain what goes wrong and why.

Violation of ME can occur.
Example:
P1: Wants to enter CS. Number[1]=0. Leaves CPU at A, after computing the max() function, but before executing the assignment statement.
P2: Wants to enter CS. Picks number[2] as, say, 250. Goes into CS.
P1: Picks number[1] as 250 also. Goes into CS. Violation of ME!

(b) Is the Bakery algorithm FIFO (First-In-First-Come) with respect to CS entry? If yes, explain why. If no, give an example illustrating the non-FIFO behavior.
No. Consider the scenario:
P1: Wants to enter CS. Number[1]=0. Leaves CPU at A, after computing the max() function, but before executing the assignment statement.
P2: Wants to enter CS. Goes into CS. Leaves CS.
P1: Picks number[1] and goes into its CS.

P2 arrives after P1, but goes into its CS first.

(c) Does the Bakery algorithm satisfy the bounded waiting property? If yes, what is the bounded waiting value? If no, how so?
BW is satisfied, and the BW bound is (n-1) where n is the number of processes. This happens when all processes pick the same number, and, by luck, the process that comes in first is Pn.

(10 pts) **(2)** Assume that you have a sequential programming language of your choice, the concurrent statement (i.e., the parbegin/parend statement), semaphores and wait/signal semaphore primitives. Assume that integer variables A, B, C, D, E and F are already initialized. Write a concurrent program fragment that computes the formula below (without algebraic transformations on the formula) with maximum concurrency.

$$((A * B) + (C * D)) / (E * F)$$

You can use temporary variables to hold intermediate results.

(Binary or nonbinary) semaphores a, b, c, d are all set to 0 initially.

```
parbegin
    begin X := A * B;   signal(a);  end;
    begin Y := C * D;  signal(b);  end;
    begin   wait(a);   wait(b);   Z := X + Y;   signal(c);  end;
    begin V := E * F; signal(d);  end;
    begin  wait(c);   wait(d);   Out := Z / V;   end;
parend;
```

(5 pts) **(3)** Consider the monitor-based Dining Philosophers solution below (discussed in the class). Can livelocks occur? If yes, give an example. If no, explain why.

```
Type dining-philosophers = monitor
    var state : array [0..4] of (thinking, hungry, eating);
    var self : array [0..4] of condition;

    procedure entry pickup (i: 0..4);
    begin
      state[i] := hungry;
      test (i);
      if state[i ] ≠ eating then self[i ].wait;
    end;

    procedure entry putdown (i: 0..4);
    begin
      state[i] := thinking;
      test (i+4 mod 5);
      test (i+1 mod 5);
    end;

    procedure test (k: 0..4);
    begin
       if state[k+4 mod 5] ≠ eating
          and state[k] = hungry
          and state[k+1 mod 5] ≠ eating
       then begin
             state[k ] := eating;
             self[k ].signal;
             end;
    end;


begin
     for i := 0 to 4  do state[i] := thinking;

end.

                                    USE:  var dp: dining-philosopher;
                                      ...
                                      dp.pickup(i);
                                      EAT
                                      dp.putdown(i);
                                       ...
```

Answer: Livelocks can occur. Here is a scenario: (a) All philosophers are thinking. (b)P0 is eating. (c) P2 is eating. (d) P1 wants to eat; and cannot. P1 is hungry. (e) P0 is thinking. (f) P0 is eating. (g) P2 is thinking. (h) P2is eating.   Then, steps e, f, g, and h keeps getting repated. → P1 can never eat, and stays hungry.

(70 pts) **(4) Railroad Crossing Problem.** Consider a single-track railroad with multiple trains using the track and a two-way car crossing where cars can cross the railroad in both directions at the same time.

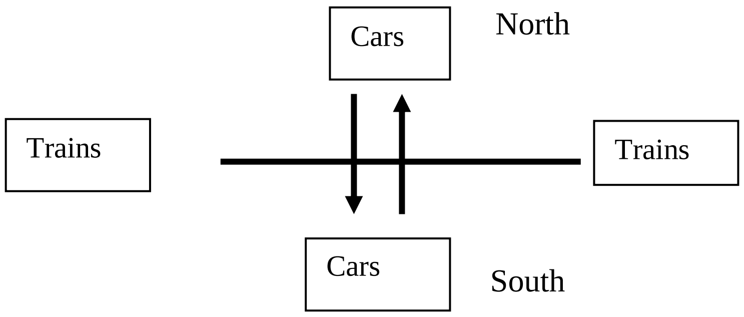
The crossing is controlled by a "crossing light". The light always rotates from green to yellow to red, and then to green. When the light is green, at most four cars—in any direction--can be crossing the railroad at the same time. That is, the number of (north-to-south, south-to-north) car combinations crossing the railroad can be (4, 0), (3, 1), (2, 2), (1, 3), or (0, 4).

When the light is yellow, no new cars are allowed to cross the railroad, and the cars that are in the process of crossing the railroad track complete their crossing. When the light is red, no cars are allowed to cross the railroad.

We assume that there is enough time between the event that the light changes from green to yellow and the event that the light changes from yellow to red so that all cars that are in the process of crossing the railroad complete the crossing process safely and, when the light turns red, there are no more cars that are crossing the railroad.

Multiple trains can use the railroad at the same time, and arrive at the crossing one after another—but, always in the same direction.

At most four cars can cross the railroad crossing at the same time and in any direction. Cars can starve while crossing the railroad tracks; trains never do.



Your task is to give a concurrent process management algorithm to the Railroad Crossing Problem where trains and cars are processes. The train approaching the crossing activates the light that changes from green to yellow. The change of the light from yellow to red is activated either by the last car crossing the railroad (at most four at a time) or, if no car is passing the crossing at the time, by the train approaching the crossing. The change of the light from red to green is activated by the train that is finishing its crossing. Your algorithm does not know the total number of trains and cars in the system.

Give a semaphore-based algorithm (solution) to the Railroad Crossing Problem. Explain your algorithm, and explicitly specify any assumptions you make about the model.

Hints: It is a good idea to

(1) Use a shared enumerated variable *light* (with values green, yellow, and red) that maintains the state of the light.

(2) Maintain shared variables to keep counts of
(i) cars "waiting to cross the track",
(ii) cars that are "in the process of crossing the track", and
(iii) trains that are "approaching" the crossing.

Define and employ "a number of" semaphores that will block cars and trains when needed, and provide mutual exclusion.

Railroad Crossing Problem: Semaphore-Based Solution

Variables:
Nonbinary Semaphores: mutex, car-wait, train-wait, initialized to 1, 0, and 0 respectively.
Integers: WaitCarCnt, PassCarCnt, TrainCnt, all initialized to zeroes.
Enumerated variable light *with values* {green, yellow, red}

CAR:

```
wait(mutex);
if (PassCarCnt < 4 and light=green)
      then   PassCarCnt++;            //Fix the count and start crossing the intersection.
      else { WaitCarCnt++;             //Can't pass. Fix the count and block self.
              signal (mutex);         // By releasing mutex, we allow other cars or trains come in.
              wait (car-wait);        // Wait for your turn
              wait(mutex);            // Cleared to cross.
              WaitCarCnt--;           //  But, first, adjust the counts mutually exclusively.
              PassCarCnt++;
            }
signal(mutex);

              CAR-PASS-CROSSING;

wait(mutex);                                                        //Done crossing.
PassCarCnt--;                                                       // Fix the counts.
if (light = green and WaitCarCnt > 0)   then signal(car-wait);      // if light is green, release another waiting car.
 signal(mutex);
```

TRAIN:

```
wait(mutex);
TrainCnt++;
 if(TrainCnt = 1)
    then {light := yellow; signal(mutex) }      //if first train then change light to yellow.
    else   { signal(mutex); wait(train-wait)}    //Otherwise, wait for your turn to approach the crossing.

        TRAIN-APPROACH-CROSSING;

wait(mutex);                              //About to enter the crossing; no cars are crossing (assumed);
light:=red;                               //so, change light to red.
signal(mutex);

        TRAIN-PASS-CROSSING;

wait(mutex);                          //Done crossing; do housekeeping work mutually exclusively and exit.
TrainCnt--;                           // Fix the count.
if (TrainCnt ≥1) then {light := yellow; signal(train-wait)};    //Release waiting trains first (cars starve).
  else {light := green;                                         //No waiting trains. Release waiting cars.
      if (WaitCarCnt=1) then {signal(car-wait)}
        else if (WaitCarCount=2) then {signal(car-wait); signal(car-wait)}
            else if (WaitCarCnt=3) then {signal(car-wait); signal(car-wait); signal(car-wait)}
                else if (WaitCarCnt ≥ 4) then {signal(car-wait); signal(car-wait); signal(car-wait); signal(car-wait)}
signal(mutex);
```