Spring 2015 **EECS 338 Assignment 3 Solns** **Due:** February 26th, 2015

G. Ozsoyoglu *Concurrent Programming with Monitors*

One-Lane Bridge Problem (Monitor-based solution).
Cars (i.e., processes) coming from the east and the west must pass a bridge across a river. Unfortunately, the bridge is a one-lane bridge. So, at any moment, the bridge can be crossed only by *one to four* cars coming from the same direction, but *not* from the opposite directions. For bounded waiting, the direction of traffic across the bridge changes each time five cars have crossed from one direction as long as one or more cars are waiting to cross from the opposite direction.

```
type One-Lane Bridge= monitor;      //Type specification.
var XingCnt:int;  XedCnt:int; EastBndWaitCnt:int; WestBndWaitCnt:int;
     EastBound:condition; WestBound:condition;   XingDrctn:Enumerated {None, EastBnd, WestBnd};

procedure entry EastBoundCar-Enter()
if ((XingDrctn=EastBnd or  XingDrctn=None) and XingCount<4 and (XedCount+XingCount)<5))
       {XingDrctn:=EastBnd; XingCount++}
  else {EastBndWaitCount++; EastBound.wait; EastBndWaitCount−−; XingCount++;  XingDrctn:=EastBnd}

procedure entry EastBoundCar-Exit()
{ XedCount++;   XingCount−−;
  if (EastBndWaitCount≠0 and ((XedCount+XingCount)<5) or WestBndWaitCount=0)) EastBound.signal;
  else if (XingCount=0 and WestBndWaitCount≠0 and (EastBndWaitCount=0 or XedCount≥5))
            {XingDrctn:=WestBnd; XedCount:=0; WestBound.signal}
       else if (XingCount=0 and EastBndWaitCount=0 and WestBndWaitCount=0)
            {XingDrctn:=None; XedCount:=0}}

procedure entry WestBoundCar-Enter()
if ((XingDrctn=WestBnd or  XingDrctn=None) and XingCount<4 and (XedCount+XingCount)<5))
       {XingDrctn:=WestBnd; XingCount++}
  else {WestBndWaitCount++; WestBound.wait; WestBndWaitCount−−; XingCount++;  XingDrctn:=WestBnd}

procedure entry WestBoundCar-Exit()
{ XedCount++;   XingCount−−;
  if (WestBndWaitCount≠0 and ((XedCount+XingCount)<5) or EastBndWaitCount=0)) WestBound.signal;
  else if (XingCount=0 and EastBndWaitCount≠0 and (WestBndWaitCount=0 or XedCount≥5))
            {XingDrctn:=EastBnd; XedCount:=0; EastBound.signal}
       else if (XingCount=0 and EastBndWaitCount=0 and WestBndWaitCount=0)
            {XingDrctn:=None; XedCount:=0}}

begin  XingCnt := XedCnt:=EastBndWaitCnt:=WestBndWaitCnt:=0;   XingDrctn := None end.

Monitor Use: var MyOneLaneBridge:OneLaneBridge;

EastBoundCar:  {//Drive; EastBoundCar-Enter(pid); CROSS; EastBoundCar-Exit(pid); DRIVE-AWAY}
WestBoundCar:  {//Drive; WestBoundCar-Enter(pid); CROSS; WestBoundCar-Exit(pid); DRIVE-AWAY}

Signaling/signaled process policy: Signaling process continues to execute; and, signaled process is blocked until
signaling process is blocked or leaves the monitor.
```

2) Drinking Philosophers Problem (Monitor-based solution).

An undirected social network graph G is given for philosophers. Philosophers (i.e., processes) are associated with nodes of the graph G, and can communicate only with immediate neighbors. A bottle is associated with each edge of G where originally none of the bottles is "acquired" by any philosopher. Each philosopher cycles between three states: *tranquil*, *thirsty*, and *drinking*. A tranquil philosopher may become thirsty at random times. Before drinking, the philosopher must acquire each and every bottle associated with edges connected to the philosopher's node in G (and make a mixed drink out of them). After drinking, a philosopher releases the bottles, and becomes tranquil again.

a. Give a monitor-based algorithm to the drinking philosophers problem that is deadlock-free.

Assume that

- *G has n nodes (philosophers), and m edges (jugs),*
- *All jugs are numbered from 1 to m, and*
- *All philosophers request their jugs in the increasing order of their adjacent jug numbers (to prevent deadlocks).*

*Procedure GetMyJugNumbers(**int** i, **int** k, **array** MyJugArray) takes as input the integer i (the process id of philosopher i), and returns as output (i) the number of edges (i.e., the no of jugs) k adjacent to philosopher i, and (ii) jug numbers of the k jugs in the array MyJugArray, in increasing order.*

```
type DrinkingPhilosophers= monitor;      //Type specification.
var  k:int;  jug:array[1..m] of condition;  MyJugArray:array [1..m] of int;  JugAcquired:array[1..m] of bool

procedure entry Acquire-Jugs(k, MyJugArray)
    for s:int from 1 to k do if not JugAcquired[MyJugArray[s]]  JugAcquired[MyJugArray[s]]:=True
                                 else {jug[MyJugArray[s]].wait; JugAcquired[MyJugArray[s]]:=True}

procedure entry Release-Jugs(k, MyJugArray)
    for s:int from 1 to k do {JugAcquired[MyJugArray[s]]:=False; jug[MyJugArray[s]].signal;

begin JugAcquired[..]:=False end.    //No initializations.

    Monitor Use: var MyDrinkingPhilosophers:DrinkingPhilosophers;

    Philosopher i:
    {MyJugArray:array [1..m] of int; i:int; k:int; state:enumerated{tranquil, thirsty, drinking}
     i:=getpid();   GetMyJugNumbers(i, k, MyJugArray);
     repeat
         THINK;  state:=thirsty;
         Acquire-Jugs(k, MyJugArray); state:=drinking;
         MIX-AND-DRINK;
         Release-Jugs(k, MyJugArray); state:=tranquil;
     until False }
```

Signaling/signaled process policy: *Signaling process continues to execute; and, signaled process is blocked until signaling process is blocked or leaves the monitor.*

b. Does your solution allow livelocks? If no, why? If yes, explain where and how livelocks occur in your solution.

This solution does NOT have livelocks. Assume that, at time t, (i) thirsty philosopher i has successfully acquired all of its adjacent jugs that are numbered less than j, and is waiting for jug j, (ii) there are already v philosophers waiting for jug j, and (iii) philosopher x is drinking from jug j. Once x releases j, philosopher i waits for the other v philosophers to acquire (drink, and release) (1) the jug j (i.e., a bounded wait of v acquisitions/releases), and, (2) the remainder of their jugs (i.e., a bounded wait of the size of integer f(v) where f is a monotone function). Thus, to eventually acquire j, i waits for a finite (i.e., v.f(v)) number of jug acquisitions (and releases).