CSC 262
Homework #7
Due at the start of class on Wednesday, April 4.

1. **Race Conditions**. For each of the following programs, state the possible values for *x* and *y* **if** the program terminates. Express your answer as a set of *(x,y)* pairs. For example, the starting configuration should be written as {(5,4)}.

In addition, state whether the program will always terminate, sometimes terminate, or never terminate.

You may assume that the variables are initialized as follows:

x : integer **init** 5
y : integer **init** 4
m0 : general semaphore **init** 0
m1 : general semaphore **init** 1
b0 : binary semaphore **init** 0
b1 : binary semaphore **init** 1

a). `cobegin x := x-y; // y := y-x; coend`

b).
```
cobegin
   DOWN(m1); x := x-y; UP(m1);
      //
   DOWN(m1); y := y-x; UP(m1);
coend
```

c).
```
cobegin
   DOWN(m0); x := x-y; UP(m0);
      //
   DOWN(m0); y := y-x; UP(m0);
coend
```

d).
```
cobegin
   DOWN(m1); x := x-y; UP(m0);
      //
   DOWN(m0); y := y-x; UP(m1);
coend
```

e).
```
cobegin
   B_UP(b1); x := x-y; B_DOWN(b0);
      //
   B_UP(b0); y := y-x; B_DOWN(b1);
coend
```

2. **Synchronization Protocols**. Canaryville has an old covered bridge in the center of town. The bridge is one lane only, and due to weight restrictions only two vehicles may be on the bridge at the same time. The village council has hired you to come up with a synchronization mechanism to protect their historic bridge and prevent any accidents.

Vehicles traveling north can be represented by a process:

```
V_N:  process
   while true do
      {entry_N}
      {traverse bridge driving north}
      {exit_N}
      {travel someplace else}
   end;
```

Similarly, vehicles traveling south can also be represented by a process:

```
V_S:  process
   while true do
      {entry_S}
      {traverse bridge driving south}
      {exit_S}
      {travel someplace else}
   end;
```

a. Using general semaphores, devise routines for $\texttt{entry}_N$, $\texttt{exit}_N$, $\texttt{entry}_S$, and $\texttt{exit}_S$ to ensure the following: (1) at any time, all traffic traversing the bridge is going in the same direction, (2) no more than two vehicles are on the bridge at once, (3) a vehicle traveling "someplace else" cannot prevent other vehicles from traversing the bridge, (4) if multiple vehicles are contending to traverse the bridge, then eventually one will succeed, and (5) the bridge is used efficiently – if only one vehicle is on the bridge in a particular direction, then a new vehicle wanting to travel in the same direction is not blocked.

> Some hints: First, design a protocol that ensures that traffic travels only in one direction at a time, using the bridge efficiently. The restriction to two vehicles on the bridge at a time can be added in at the end. Now test your protocol in a number of scenarios. If a car is on the bridge southbound, does an incoming northbound car wait (as it should)? Does an incoming southbound car enter immediately (as it should)? When the last southbound car crosses the bridge, do *two* waiting northbound cars get released, if appropriate? If there aren't any northbound cars waiting, does your algorithm return to its starting condition (where an incoming car in either direction will enter and cause traffic in the opposite direction to wait)? If your algorithm fails any of these tests, modify it and check each test again. This is essentially a programming exercise, only on paper – proper debugging is essential.
>
> A study of protocols for similar problems may be helpful; some examples are found in sections 5.7 and 6.6, of the text, and in section 2.3 of Tanenbaum & Woodhull (on reserve).

b. Discuss whether an attempt by any specific vehicle to traverse the bridge is guaranteed by your protocols, assuming that traversing the bridge takes each vehicle a finite time. Is your protocol starvation-free? Justify your conclusion.

3. **Message Passing**. The town leaders of Canaryville are so impressed with your work on the protocol for their historic covered bridge that they want to hire you for a new project. The highway department has recently built a new one-lane bridge in town that runs from east to west. This bridge also needs a protocol to prevent accidents, although it doesn't have any weight restriction. (In other words, any number of cars may be on the bridge at once.) The town leaders have heard that message passing protocols are in vogue, and they have asked you to design a protocol based upon message passing so that their new bridge will be as modern as possible.

The protocol will establish a single process whose sole purpose is to exercise authority over the bridge. All driver processes will apply to the authority for permission to cross. When they receive permission from the authority, they begin cross the bridge, and when they have finished they inform the authority that they are done. The authority's job is to tell driver processes when it is safe to cross, keeping track of traffic flow so as to prevent accidents.

The message passing system available uses a non-blocking send and a blocking receive. (In other words, a process may send messages without blocking, but an attempt to receive will cause the process to block unless there is a sent message already waiting.) Furthermore, instead of direct process-to-process communication, messages are sent via mailboxes. For this problem, assume that any process may send to a mailbox, causing the message to be appended to a queue. Any process may receive from a mailbox, retrieving (and removing) the first message on the queue or blocking if the queue is empty. (This is a commonly used message-passing variant.)

a.) Write the protocol in pseudocode for the central authority. Assume that the driver processes are executing the following protocol:

```
⟨eastbound⟩:
  while true do
    send(inbox,"EAST-ENTER");
    receive(east,msg);
      // block until message arrives
    {cross bridge}
    send(inbox,"EAST-EXIT");
    {drive elsewhere}
  end;
```

```
⟨westbound⟩:
  while true do
    send(inbox,"WEST-ENTER");
    receive(west,msg);
      // block until message arrives
    {cross bridge}
    send(inbox,"WEST-EXIT");
    {drive elsewhere}
  end;
```

Note the use of the message passing functions *send(mailbox_name,msg)* and *receive(mailbox_name,msg)*. You will use these in your solution. There are three mailboxes involved: *inbox* (for sending messages to the bridge authority), *east* (where the authority sends permission to go east) and *west* (where the authority sends permission to go west).

For convenience, you may also want to assume the existence of functions that parse the messages from *inbox*. For example, *direction(msg)* returns *EAST* or *WEST*, and *action(msg)* returns *ENTER* or *EXIT*.

A hint: You will need to establish internal variables so the authority can keep track of the traffic situation. All variables are local in a message-passing system, so you don't need to worry about mutex protection for these variables. Your protocol should loop continuously, waiting for an

incoming message, updating its internal variables, and (if appropriate) issuing messages telling driver processes to proceed.

b.) This bridge will have more traffic than the old one. The finished protocol should allow traffic flow to switch directions periodically (prevent *starvation*). In other words, if cars are traveling eastbound, and a westbound car arrives, no more eastbound cars should start to cross the bridge until the waiting westbound car(s) have gone. If your protocol does not prevent starvation, modify it so that it does.

c.) Present a brief but convincing argument that your protocol will prevent both collisions and starvation.