CS3331 Concurrent Computing Exam 2 Solutions Fall 2018

1. **Synchronization**

(a) **[15 points]** Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1, where `flag[ ]` is a Boolean array of two elements and `turn[ ]` is an `int` array, each of its two elements can only hold `0` or `1`. Note that `flag[ ]` and `turn[ ]` are global variables shared by both P_0 and P_1.

```
     int   flag[2] = { FALSE, FALSE};   // global flags
     int   turn[2] = { 0, 1 };   // global turn variables

    Process i (i = 0 or 1)

    // Enter Protocol
  1. flag[i] = TRUE;
  2. turn[i] = (turn[j] + i)%2;
  3. repeat
  4. until ( flag[j] == FALSE || turn[i] != (turn[j]+i)%2 );

    // critical section

    // Exit Protocol
  5. flag[i] = FALSE;
```

Prove rigorously that this solution satisfies the mutual exclusion condition. *You will receive* **zero** *point if* **(1)** *you prove by example, or* **(2)** *your proof is vague and/or unconvincing.*

Answer: If P_0 is in its critical section, P_0 must have passed the `repeat-until` loop and the following holds

```
flag[1] == FALSE || turn[0] != (turn[1]+0)%2
```

If P_1 is in its critical section, the following holds

```
flag[0] == FALSE || turn[1] != (turn[0]+1)%2
```

However, since `flag[i]` is set to `TRUE` at the beginning of the enter section, if P_0 and P_1 are both in their critical sections, the following must both hold.

```
turn[0]  !=  (turn[1]+0)%2     (1)
turn[1]  !=  (turn[0]+1)%2     (2)
```

Since the value of `turn[i]` can only be `0` or `1` because of the mod 2 operator, there are four possible combinations of `turn[0]` and `turn[1]`. As a result, we may do a complete enumeration as follows:

`turn[0]`	`turn[1]`	`(turn[1]+0)%2`	Equation (1)	`(turn[0]+1)%2`	Equation (2)
0	0	0	FALSE	1	TRUE
0	1	1	TRUE	1	FALSE
1	0	0	TRUE	0	FALSE
1	1	1	FALSE	0	TRUE

From the above table, we learn that Equation (1) and Equation (2) cannot hold at the same time. This is a contradiction, and, consequently, P_0 and P_1 cannot be in their critical sections at the same time.

Or, we could also argue in a slightly different way. Note that `turn[0]` can only have two possible values, 0 or 1. Now, we have the following:

- `turn[0]` **being 0**. From Equation (1), we have `0 != turn[1]`, which means `turn[1]` has to be 1. Plugging `turn[0] = 0` and `turn[1] = 1` into Equation (2) yields `1 != (0 + 1)%2`, which is impossible. Therefore, `turn[0]` cannot be 0.
- `turn[0]` **being 1**. Again, from Equation (1), we have `1 != turn[1]`, which means `turn[1]` has to be 0. Plugging `turn[0] = 1` and `turn[1] = 0` into Equation (2) yields `0 != (1 + 1)%2`, which is impossible. Therefore, `turn[0]` cannot be 1.

Thus, if we assume P_0 and P_1 are both in the critical section, `turn[0]` cannot be 0 or 1. By the same reason, we also obtain that `turn[1]` cannot be 0 or 1. This is absurd, because `turn[0]` and `turn[1]` must have 0 or 1 based on the program logic. Consequently, we have a contradiction and mutual exclusion is satisfied. ■

(b) **[15 points]** Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1, where `flag[2]` is a global Boolean array of two elements. Explain whether the progress condition is met. For each condition, either provide a rigorous proof showing the condition is met, or provide a **step-by-step execution sequence** showing that the condition is violated. **You will receive zero point if you do not use an execution sequence table and/or provide a vague argument.**

```
bool  flag[2];   // global flags

Process i (i = 0 or 1)

flag[i] = TRUE;        // I am interested
while (flag[j]) {      // while you are interested
    flag[i] = FALSE;   /   yield to you
    while (flag[j])    //   wait until you are not interested
        ;
    flag[i] = TRUE;    // then I will try again
}
// critical section
flag[i] = FALSE;           // I am done and not interested
```

Answer: Both progress and bounded waiting conditions are violated. If P_0 and P_1 starts at the same time and runs in exactly the same speed without interruption (perhaps on difference processors), the `while` loop will never exits. The following is a possible execution sequence.

P_0	P_1	`flag[0]`	`flag[1]`	Comment
`flag[0] = TRUE`	`flag[1] = TRUE`	`TRUE`	`TRUE`	Both processes start
`while (flag[1])`	`while (flag[0])`	`TRUE`	`TRUE`	Test other's flag
`flag[0] = FALSE`	`flag[1] = FALSE`	`FALSE`	`FALSE`	Both yield
`while (flag[1])`	`while (flag[0]`	`FALSE`	`FALSE`	Test other's flasg again
`flag[0] = TRUE`	`flag[1] = TRUE`	`TRUE`	`TRUE`	Both reset own flag
Both P_0 and P_1 start the next iteration				

In this way, P_0 and P_1 loop indefinitely, and the progress condition is violated because both P_0 and P_1 are trying to enter but fail to make a decision in finite time. Because both P_0 and P_1 are looping indefinitely, none of them will eventually enter the critical section, violating the bounded waiting condition.

This is the same problem on page 5 and page 6 in `06-Sync-Soft-Hardware.pdf`. ■

(c) **[10 points]*** Define the meaning of a *race condition*? Answer the question first and use execution sequences with a clear and convincing argument to illustrate your answer. **You must explain step-by-step why your example causes a race condition.**

Answer: A *race condition* is a situation in which *more than one* processes or threads manipulate a shared resource *concurrently*, and the result depends on *the order of execution*.

The following is a simple counter updating example discussed in class. The value of `count` may be 9, 10 or 11, depending on the order of execution of the **machine instructions** of `count++` and `count--`.

```
int         count = 10;  // shared variable

Process 1                Process 2

count++;                 count--;
```

The following execution sequence shows a race condition. Two processes run concurrently (condition 1). Both processes access the shared variable `count` concurrently (condition 2) because `count` is accessed in an interleaved way. Finally, the computation result depends on the order of execution of the `SAVE` instructions (condition 3). The execution sequence below shows the result being 9; however, switching the two SAVE instructions yields 11. Since all conditions are met, we have a race condition. **Note that you have to provide TWO execution sequences, one for each possible result, to justify the existence of a race condition.**

Thread_1	Thread_2	*Comment*
do somthing	do somthing	count = 10 initially
LOAD count		Thread_1 executes count++
ADD #1		
	LOAD count	Thread_2 executes count--
	SUB #1	
SAVE count		count is 11 in memory
	SAVE count	Now, count is 9 in memory

Stating that "count++ followed by count--" or "count-- followed by count++", even using machine instructions, produces different results and hence a race condition is **incomplete**, because the two processes do not access the shared variable count concurrently. Note that the use of higher-level language statement interleaved execution may not reveal the key concept of "sharing" as discussed in class. Therefore, use instruction level interleaved instead.

See pp. 5–12 of 05-Sync-Basics.pdf. ■

2. **Semaphores**

(a) **[10 points]** The semaphore methods Wait() and Signal() must be atomic to ensure a correct implementation of mutual exclusion. Use execution sequences to show that if Wait() is not atomic then *mutual exclusion cannot be maintained*. In other words, use execution sequences to show that, if the implementation of Wait() is not atomic, the internal counter of a semaphore cannot be correct and multiple processes can enter their critical sections at the same time. **You must show clearly what the intended *mutual exclusion* is and how the mutual exclusion condition is violated with execution sequences and provide a convincing explanation. Otherwise, you will risk a lower score. Note also that this question asks for a possible violation of mutual exclusion rather than having a race condition.**

Answer: If Wait() is not atomic, its execution may be switched in the middle. If this happens, mutual exclusion will not be maintained. Consider the following solution to the critical section problem:

```
Semaphore S = 1;

Process A              Process B
---------              ---------

Wait(S);               Wait(S);
             // in critical section
Signal(S);             Signal(S);
```

The following is a possible execution sequence, where Count = 1 is the internal counter variable of the involved semaphore S.

Process A	*Process B*	Count	*Comment*
		1	Initial value
LOAD Count		1	*A* executes Count-- of Wait()
SUB #1		1	
	LOAD Count	1	*B* executes Count-- of Wait()
	SUB #1	1	
	SAVE Count	0	*B* finishes Count--
SAVE Count		0	*A* finishes Count--
if (Count < 0)		0	It is false for *A*
	if (Count < 0)	0	It is false for *B*
Both *A* and *B* enter the critical section			

Note that this question asks you to demonstrate a violation of mutual exclusion. Consequently, you receive low grade if your demonstration is not a violation of mutual exclusion. Additionally, if you

failed to indicate how the needed critical sections that require mutual exclusion is formed, you also risk a lower grade.

This problem was assigned as an exercise in class. See slide 8 of `08-Semaphores.pdf`. ■

(b) **[10 points]** As discussed in class, it is very typical that a thread must do something (*e.g.*, the entry protocol) before doing its core task, and do something else upon exit (*e.g.*, the exit protocol). The entry and exit protocols must perform some locking and unlocking activities in order to access some shared data items to avoid potential race conditions. However, too many locking and unlocking activities can be rather inefficient, and the *passing the baton* technique may be used to cut some redundant locking and unlocking activities.

The following shows an entry protocol and an exit protocol without the use of the passing the baton technique. Modify it to use the passing the baton technique. **Just directly edit the code below, and indicate clearly how the baton is passed. You will receive zero point if you provide no elaboration or a vague elaboration.** This is really a simple problem if you attended my lecture.

```
 1 int        Waiting = 0;         // the number of waiting thread
 2 Bool       Must_Wait = FALSE;   // variable to be set somewhere in the code
 3 Semaphore  Mutex = 1;           // mutual exclusion lock
 4 Semaphore  Delay = 0;           // semaphore to block thread if it must wait
   // Entry Protocol
 5 Mutex.Wait();                   // lock the system
     // determine if this thread has to wait by setting Must_wait
 6   if (Must_Wait) {              // if I have to wait, then
 7      Waiting++;                 //    one more waiting threads
 8      Mutex.Signal();            //    release the lock
 9      Delay.Wait();              //    wait here!
10      Mutex.Wait()               //    get the lock back
11   }
12   Waiting--;                    // one less waiting threads
13 Mutex.Signal();                 // unlock the system
   // Whatever
   // Exit Protocol
14 Mutex.Wait();                   // upon exit, lock the system
15    if (Waiting > 0)             // if there are waiting thread
16       Delay.Signal();           //    release one of them
17 Mutex.Signal();                 // unlock the system
```

Answer: The "passing the baton" pattern means that before a process in the critical section (*i.e.*, the baton) exits, it releases a waiting to enter process and gives that process the right to using the critical section (*i.e.*, the baton). In this way, the exiting process does not have to release the lock for the critical section, and the release process does not have to lock the critical section because the lock is passed by the exiting one.

Note that `Waiting` counts the number of processes waiting on semaphore `Delay` (line 7). In the entry section a process is blocked by semaphore `Delay.Wait()` (line 9) after unlocking the `Mutex`that is used to protect variable `Waiting`. This released process tries to lock the mutex again (line 10) in order to regain the access to `Waiting` (line 12). On the other hand, in the exit section, the exiting process signals semaphore `Delay` to release a process if `Waiting` is not 0 (line 16). Then, it unlocks the critical section and leaves (line 17).

The process has the baton is the exiting one. This exiting process can simply hand the baton over to the released one without executing `Mutex.Signal()` (line 17), and the released process does not have to lock the critical section again because the `Mutex` was given by the exiting process. The released process unlocks the lock (line 13) after finishing updating `Waiting`.

Once you understand the key concept of passing the baton, the modification is very simple as follows:

- Remove `Mutex.Wait()` on line 10, because the released process will be given the lock.
- Change the `Mutex.Signal()` on line 17 so that it is not called if a waiting process is released (*i.e.*, passing the baton to the released one).

The following is the answer, with the original line numbers unchanged.

```
 1 int        Waiting = 0;         // the number of waiting thread
 2 Bool       Must_Wait = FALSE;   // variable to be set somewhere in the code

 3 Semaphore  Mutex = 1;           // mutual exclusion lock
 4 Semaphore  Delay = 0;           // semaphore to block thread if it must wait

   // Entry Protocol

 5 Mutex.Wait();                   // lock the system

     // determine if this thread has to wait by setting Must_wait

 6   if (Must_Wait) {              // if I have to wait, then
 7      Waiting++;                 //    one more waiting threads
 8      Mutex.Signal();            //    release the lock
 9      Delay.Wait();              //    wait here!
10 //   Mutex.Wait()               // don't need it as the baton will be passed to me
11   }
12   Waiting--;                    // one less waiting threads
13 Mutex.Signal();

   // Whatever

   // Exit Protocol

14 Mutex.Wait();                   // upon exit, lock the system
15    if (Waiting > 0)             // if there are waiting thread
16       Delay.Signal();           //    release one of them
                                   // no need to signal due to passing the baton
17    else                         // do this only if no process is released
18       Mutex.Signal();           // unlock the mutex only no process was released
```

This simple concept is discussed on pp. 46–48 and on pp. 50-54 of `08-Semaphores.pdf`. This problem is just a rewording of the example on pp. 47-48. ∎

(c) **[10 points]** Three ingredients are needed to make a cigarette: tobacco, paper and matches. An agent has an infinite supply of all three. Each of the three smokers has an infinite supply of one ingredient only. That is, one of them has tobacco, the second has paper, and the third has matches. The following solution uses three semaphores, each of while represents an ingredient, and a fourth one to control the table. A smoker waits for the needed ingredients on the corresponding semaphores, signals the table semaphore to tell the agent that the table has been cleared, and smokers for a while.

```
   Semaphore  Table = 0;               // table semaphore
   Semaphore  Sem[3] = { 0, 0, 0 };    // ingredient semaphores

   int        TOBACCO = 0, PAPER = 1, MATCHES = 2

   Smoker_Tobacco            Smoker_Paper              Smoker_Matches

1. while (1) {               while (1) {               while (1) {
      // other work
2.    Sem[PAPER].Wait();        Sem[TOBACCO].Wait();      Sem[TOBACCO].Wait();
3.    Sem[MATCHES].Wait();      Sem[MATCHES].Wait();      Sem[PAPER].Wait();
4.    Table.Signal();           Table.Signal();           Table.Signal();
      // smoke
5. }                         }                         }
```

The agent adds two randomly selected different ingredients on the table, and signals the corresponding semaphores. This process continues forever.

```
while (1) {
   // generate two different random integers in the range of 0 and 2,
   //     say X and Y
   Sem[X].Signal();
   Sem[Y].Signal();
   Table.Wait();
   // do some other tasks
}
```

Show, using execution sequences, that this solution can have a deadlock. **You will receive zero point if you do not use valid execution sequences.**

Answer: This is a simple problem and was discussed in class when we talked about the smokers problem. The following shows a possible execution sequence:

Smoker 1	Smoker 2	Smoker 3	Agent	Comment
Wait on `PAPER`	Wait on `TOBACCO`	Wait on `TOBACCO`		All three smokers wait
			Signal `TOBACCO`	Tobacco available
		Wait in `PAPER`		Smoker 3 released
			Signal `PAPER`	Paper available
Wait on `MATCHES`				Smoker 1 released
			Wait on `TABLE`	Agent blocks and waits on `TABLE`
All smokers and the agent blocks				Deadlock occurs

There are other execution sequences based on the same idea. ■

3. **Problem Solving:**

(a) **[15 points]** A multithreaded program has two global arrays and a number of threads that execute concurrently. The following shows the global arrays, where `n` is a constant defined elsewhere (*e.g.*, in a `#define`):

```
int  a[n], b[n];
```

Thread T_i ($0 < i \leq n-1$) runs the following (pseudo-) code, where function `f()` takes two integer arguments and returns an integer, and function `g()` takes one integer argument and returns an integer. Functions `f()` and `g()` do not use any global variable.

```
while (not done) {
   a[i] = f(a[i], a[i-1]);
   b[i] = g(a[i]);
}
```

More precisely, thread T_i passes the value of `a[i-1]` computed by T_{i-1} and the value of `a[i]` computed by T_i to function `f()` to compute the new value for `a[i]`, which is then passed to function `g()` to compute `b[i]`.

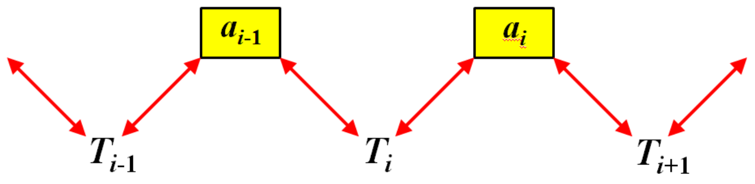
Declare semaphores with initial values, and add `Wait()` and `Signal()` calls to thread T_i so that it will compute the result correctly. Your implementation should not have any busy waiting, race condition, and deadlock, and should aim for **maximum parallelism**.

A convincing correctness argument is needed. Otherwise, you will receive no credit for this problem.

Answer: If you look at the code carefully, you will see that thread T_i ($1 < T_i < n-1$) shares `a[i-1]` and `a[i]` with T_{i-1} and T_{i+1}, respectively. See the diagram below. Therefore, T_i must wait until T_{i-1} finishes using `a[i-1]` before using it in the computation of `a[i] = f(a[i], a[i-1])`. Similarly, T_i must wait until T_{i+1} finishes using `a[i]` before computing a new value for `a[i]`. To this end, we need a semaphore `s[i-1]` for protecting `a[i-1]` and a semaphore `s[i]` for protecting `a[i]`. Isn't this very similar to the philosophers problem if you considered T_i as a philosopher and `a[i-1]` and `a[i]` as T_i's chopsticks? The major difference is that the philosophers are "circular" while the T_i's are "linear."

Next, we consider T_1 and T_{n-1}. T_1 uses `a[0]` and `a[1]`. Because no thread updates `a[0]`, T_1 only needs a semaphore `s[1]` to protect `a[1]`. Similarly, thread T_{n-1} uses `a[n-1]` and `a[n-2]`. Because there is no thread using `a[n-1]` other than T_{n-1} itself, T_{n-1} only needs a semaphore `s[n-2]` to protect `a[n-2]`.

The following is a possible solution. Note that `b[i]` is not protected by any semaphore because **(1)** only T_i uses `b[i]` and **(2)** only T_i writes into `a[i]`, and once `a[i]` is correctly computed one can compute `b[i]` correctly.

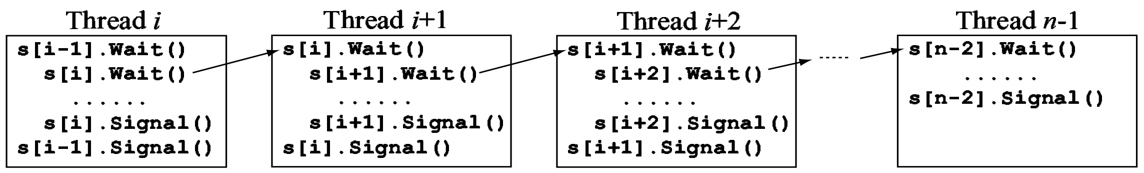


```
Sem s[1..n-1] = { 1, 1, 1, ..., 1 };  // all semaphores = 1 for mutual exclusion

Thread-1:                     Thread-i:                      Thread-(n-1):
while (not done) {            while (not done) {             while (not done) {
   s[1].Wait();                  s[i-1].Wait();                 s[n-2].Wait();
      a[1] = f(a[0], a[1]);      s[i].Wait();                      a[n-1] = f(a[n-2], a[n-1]);
   s[1].Signal();                   a[i] = f(a[i-1], a[i]);     s[n-2].Signal();
   b[1] = g(a[1]);               s[i].Signal();                 b[n-1] = g(a[n-1]);
}                                s[i-1].Signal();            }
                                 b[i] = g(a[i]);
                              }
```

Obviously, there is no busy waiting. The above solution has no race conditions either because each shared data item (*i.e.*, `a[i-1]` and `a[i]` for thread T_i) is protected by mutual exclusion.

This solution is deadlock free. What we have to show is that no semaphore would block threads indefinitely. Consider thread T_i, which may be blocked on semaphores `s[i-1]` or `s[i]`. If T_i is blocked on semaphore `s[i-1]`, this means thread T_{i-1} is now in its critical section. Thread T_{i-1} will eventually exit its critical section and execute `s[i-1].Signal()`. At this moment, thread T_i could continue although T_{i-1} may come back fast enough and execute `s[i-1].Wait()` successfully again. Whatever the case is, T_i has the chance to continue, and, the worst case is starvation rather than a deadlock.



```
Thread i               Thread i+1             Thread i+2             Thread n-1
s[i-1].Wait()          s[i].Wait()            s[i+1].Wait()          s[n-2].Wait()
  s[i].Wait()            s[i+1].Wait()          s[i+2].Wait()          ......
    ......                 ......                 ......             s[n-2].Signal()
  s[i].Signal()          s[i+1].Signal()        s[i+2].Signal()
s[i-1].Signal()        s[i].Signal()          s[i+1].Signal()
```

If thread T_i is blocked on semaphore `s[i]`, this means thread T_{i+1} executed its `s[i].Wait()`, followed by `s[i+1].Wait()`. See the figure above. Thread T_{i+1} may or may not be blocked by `s[i+1].Wait()`. If thread T_{i+1} is not blocked by `s[i+1].Wait()`, it will eventually signal it, allowing T_i to continue. If thread T_{i+1} is blocked by `s[i+1].Wait()`, it is because thread T_{i+2} is blocked by `s[i+2].Wait()`. With the same reasoning, the worst case is that this chain of "waiting" could continue until thread T_{n-1}. However, if T_{n-1} can pass `s[n-2].Wait()`, which causes thread T_{n-2} to wait, it will signal `s[n-2]` later, and, as a result, thread T_{n-2} has a chance to run. Thread T_{n-2} will signal `s[n-3]`, allowing thread T_{n-3} to run. This backward chain of "signal" will eventually return to thread T_{i+1}, which will signal `s[i]`. Therefore, thread T_i always has a chance to pass semaphore `s[i].Wait()` and enters its critical section. This means, again, thread T_i does not involve in a deadlock. At worst, it only has starvation.

Incorrect Solution 1: There are some incorrect solutions and the following is one of them.

```
Sem s = 1;

Thread i:
while (not done) {
   s.Wait();
      a[i] = f(a[i], a[i-1]);
      b[i] = g(a[i]);
   s.Signal();
}
```

This "solution" uses semaphore `s`, and thread T_i acquires `s` before updating `a[i]` and `b[i]`. The problem is that there is virtually no concurrency at all. In other words, because at any moment there is at most one thread can be in its critical section, at any moment there is only one T_i in execution. As a result, the original requirement of multithreading is destroyed by the use of single semaphores.

Some even move `s.Wait()` and `s.Signal()` outside of the `while` loop. In this case, you are guaranteed that the thread that is lucky enough to enter the critical section will run forever and no other threads can have a chance to execute.

Incorrect Solution 2: Because T_i uses `a[i-1]` to update `a[i]` and T_1 uses `a[0]`, which is not changed, the following "solution" seems correct. The problem is that this solution forces the execution to be T_1, T_2, ..., T_{n-1} and then the system deadlocks because no one allows T_1 to run.

```
Sem s[n] = { 1, 0, ..., 0};  // allows T1 to start first
                             // and all remaining Ti's to block

Thread 1:                         Thread i:
while (not done) {                while (not done) {
   s[1].Wait();                      s[i].Wait();
      a[1] = f(a[1], a[0]);             a[i] = f(a[i], a[i-1]);
      b[1] = g(a[1]);                   b[i] = g(a[i]);
   s[2].Signal();                    s[i+1].Signal();
}                                 }
```

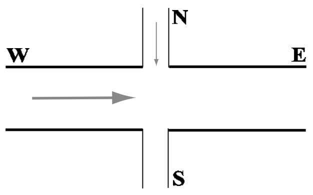
Incorrect Solution 3: One might suggest allowing T_{n-1} to signal T_1 as shown below. The situation is improved just a little; but, the execution would still be fixed (*i.e.*, T_1, T_2, ..., T_{n-1}, T_1, T_2, ..., T_{n-1}, ...) and at any time only one thread can be in execution (*i.e.*, no concurrency).

```
Sem s[n] = { 1, 0, ..., 0};  // allows T1 to start first
                             // and all remaining Ti's to block

Thread 1:                    Thread i:                    Thread n-1:
while (not done) {           while (not done) {           while (not done) {
   s[1].Wait();                 s[i].Wait();                 s[n-1].Wait();
      a[1] = f(a[1], a[0]);        a[i] = f(a[i], a[i-1]);      a[n-1] = f(a[n-1], a[n-2]);
      b[1] = g(a[1]);              b[i] = g(a[i]);              b[n-1] = g(a[n-1]);
   s[2].Signal();               s[i+1].Signal();             s[1].Signal();
}                            }                            }
```

■

(b) **[15 points]** A main highway cuts through a rural road as shown below. East-bound vehicles are on the highway, while south-bound vehicles are on the rural road. To avoid delay on the highway, the following traffic regulations are implemented:



- As long as there are east-bound vehicles crossing the intersection, east-bound vehicles do not have to stop, and they just follow the traffic flow. In this case, south-bound vehicles must stop at the intersection.
- If there is a south-bound vehicle crossing the intersection, all east-bound vehicles must stop at the intersection.
- To prevent south-bound vehicles from blocking the highway, only **one** south-bound vehicle can proceed and enter the intersection. On the other hand, multiple east-bound vehicles may cross the intersection at the same time.
- If east-bound vehicles and south-bound vehicles approach the intersection at the same time, only one can proceed. This vehicle may be east-bound or south-bound.

The east-bound and south-bound vehicles are run as threads and use the following template:

```
east-bound(...)                          south-bound(...)
{                                        {
      // approach the intersection             // approach the intersection
      wait or proceed.  add code here.         wait or proceed.  add code here.
      // exit the intersection                 // exit the intersection
}                                        }
```

Write the code for `east-bound()` and `south-bound()` and add semaphores and variables as needed. You may use the simple syntax discussed in class (and in class notes) rather than that of ThreadMentor. **You must provide a convincing elaboration to show the correctness of your program.**

Answer: This is a rephrased Readers-Writers problem. The east-bound vehicles are readers and the south-bound vehicles are writers. Since an east-bound vehicle can cross the intersection as long as there are vehicles in the same direction crossing the intersection, they are the readers. On the other hand, since south-bound vehicles must cross the intersection exclusively, they are writers.

```
semaphore Mutex = 1, Intersection = 1;
int       Count;

        East-Bound                      South-Bound
        ----------                      -----------
while (1) {                         while (1) {
   Mutex.Wait();
      Count++;
      if (Count == 1)
         Intersection.Wait();          Intersection.Wait();
   Mutex.Signal();

   // cross intersection

   Mutex.Wait();
      Count--;
      if (Count == 0)
         Intersection.Signal();        Intersection.Signal();
  Mutex.Signal();
}                                   }
```

Here is a quick review of this solution. `Mutex` is a semaphore protecting the counter `Count`. `Count` counts the number of east-bound vehicles crossing the intersection, and, hence, as long as `Count` is non-zero an east-bound vehicle can proceed. However, the first vehicle tries to cross the intersection yields the intersection to itself or to a south-bound vehicle. After crossing the intersection, if `Count` becomes zero, this "last" vehicle must allow those vehicles waiting to cross to proceed.

Lesson Learned: Understand the concept of each classical problem rather than only memorizing the code.
■