CS3331 Concurrent Computing Exam 2 Solutions Spring 2014

1. **Synchronization Basics**

(a) [**15 points**] Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1. A process can be making a request `REQUESTING`, executing in the critical section `IN_CS`, or having nothing to do with the critical section `OUT_CS`. This status information, which is represented by an `int`, is saved in `flag[i]` of process P_i. Moreover, variable `turn` is initialized elsewhere to be `0` or `1`. Note that `flag[]` and `turn` are global variables shared by both P_0 and P_1.

```
int   flag[2];   // global flags
int   turn;      // global turn variable, initialized to 0 or 1

Process i (i = 0 or 1)

// Enter Protocol
repeat                                        // repeat the following
   flag[i] = REQUESTING;                      // making a request to enter
   while (turn != i && flag[j] != OUT_CS)    // as long as it is not my turn and
      ;                                       //    the other is not out, wait
   flag[i] = IN_CS;                           // OK, I am in (well, maybe); but,
until flag[j] != IN_CS;                       //    must wait until the other is not in
turn = i;                                     // the other is out and it is my turn!

// critical section

// Exit Protocol
turn = j;                                     // yield the CS to the other
flag[i] = OUT_CS;                             // I am out of the CS
```

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 not convincing.*

Answer: A process that enters its critical section must first set `flag[i] = IN_CS` and then see `flag[j] != IN_CS` being true at the end of the `repeat-until` loop. Therefore, if process P_0 is in the critical section, it had executed `flag[0] = IN_CS` followed by seeing `flag[1] != IN_CS`. By the same reason, if process P_1 is in the critical section, it had executed `flag[1] = IN_CS` followed by seeing `flag[0] != IN_CS`. Consequently, if P_0 and P_1 are both in the critical section, we have `flag[0] = IN_CS` and `flag[1] != IN_CS` (from P_0's point of view) and `flag[1] = IN_CS` and `flag[0] != IN_CS` (from P_1's point of view). As a result, `flag[0]` and `flag[1]` are equal to `IN_CS` and not equal to `IN_CS` at the same time. This is impossible, and the mutual exclusion condition holds.

See page 8 of `06-Sync-Soft-Hardware.pdf`. This is exactly the same technique we used to show that Peterson's algorithm satisfies the mutual exclusion condition. ∎

(b) [**10 points**]* Define the meaning of a *race condition*? Answer the question first and use an execution sequence 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 access 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` at the same time (condition 2). 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.

`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++`" produces different results and hence a race condition is at least **incomplete**, because the two processes do not access the shared variable `count` at the same time (*i.e.*, condition 2).

See page 5 to page 9 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.

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.

This problem was assigned as an exercise in class. ■

(b) **[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

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

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. ∎

(c) **[15 points]** A programmer used two semaphores to design a class `Barrier`, a constructor, and method `Barrier_wait()` that fulfills the following specification:

- The constructor `Barrier(int n)` takes a positive integer argument `n`, and initializes a private `int` variable in class `Barrier` to have the value of `n`.
- Method `Barrier_wait(void)` takes no argument. A thread that calls `Barrier_wait()` blocks if the number of threads being blocked is less than `n`-1, where `n` is the initialization value and will not change in the execution of the program. Then, the `n`-*th* calling thread releases all `n`-1 blocked threads and all `n` threads continue.

This programmer came up with the following solution. However, he found his solution could react strangely because sometimes the same thread may be released multiple times in the same batch. Of course, this is wrong.

```
Semaphore  Mutex       = 1;
Semaphore  WaitingList = 0;

Barrier::Barrier_wait()
{
   int  i;

   Mutex.Wait();                 // lock the counter
   if (count == Total-1) {       // I am the n-th one
      count = 0;                 // reset counter
      Mutex.Signal();            // release the lock
      for (i=1; i<=Total-1; i++) // release all waiting threads
           WaitingList.Signal();
   }                             // I am done
   else {                        // otherwise, I am not the last one
      count++;                   // one more waiting threads
      Mutex.Signal();            // release the mutex lock
      WaitingList.Wait();        // block myself
   }
}
```

Help this programmer pinpoint the problem with an execution sequence plus a convincing explanation.

Answer: This solution does not have a mechanism to prevent fast-runners from coming back, and, as a result, a thread just released may come back and get released again.

If there are $n-1$ threads waiting on semaphore `WaitingList`, where n is `Total` in the program. Then, thread T_n comes, locks `Mutex`, sets `count` to 0, unlocks `Mutex`, and starts signaling semaphore `WaitingList`. Right after a thread, say T_i, is released from semaphore `WaitingList` and before T_n can signal the second time, T_n is switched out and T_i is switched in. As a result, T_i runs, comes back, sees `count` being 0, increases `count` by 1, and waits on semaphore `WaitingList` again. Now, thread T_n resumes, and signals the second time. Since there is no pareticular order for releasing threads from a semaphore, T_i may be released the second time.

The following is an execution sequence:

Thread T_i	Thread T_n	`count`	Comment
		$n-1$	$n-1$ threads waiting
	`Mutex.Wait()`	$n-1$	T_n arrives
	`if ...`	$n-1$	T_n sees a full count
	`count = 0`	0	T_n resets `count`
	`WaitingList.Signal()`	0	T_n signals once
`Mutex.Wait()`		0	T_i is released and comes back
`count++`		1	T_i increases `count`
`Mutex.Signal()`		1	T_i releases `count`
`WaitingList.Wait()`		1	T_i blocks
	`WaitingList.Signal()`	1	T_n signals the second time
Thread T_i may be released again!			

In this way, the fast runner T_i is released twice. ∎

3. **Problem Solving:**

(a) **[20 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. Therefore, T_i must wait until T_{i-1} completes its update to `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]`. Since 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]`. Since 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]`.

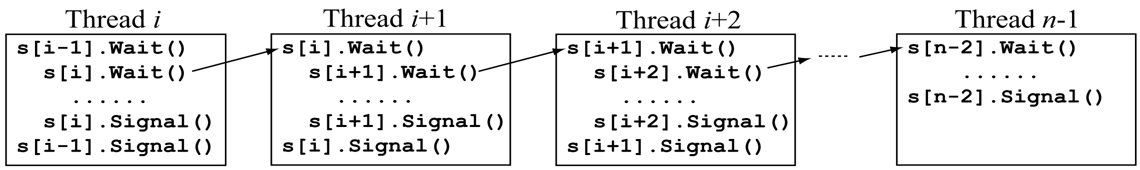
The following is a possible solution. Note that `b[i]` is not protected by any semaphore because (1) only T_i uses `b[i]` and (2) only T_i writes into `a[i]`, and once `a[i]` is correctly computed one can compute `b[i]` correctly.

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

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

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

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

Obviously, there is no busy waiting. The above solution has no race condition 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.

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 threading at all. In other words, since 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 single semaphore.

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: Since 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 threading).

```
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) **[20 points]** A unisex bathroom is shared by men and women. A man or a woman may be using the room, waiting to use the room, or doing something else. They work, use the bathroom and come back to work. The rule of using the bathroom is very simple: *there must never be a man and a woman in the room at the same time; however, people with the same gender can use the room at the same time.*

Man Thread

```
void Man(void)
{
   while (1) {
      // working
      // use the bathroom
}
```

Woman Thread

```
void Woman(void)
{
   while (1) {
      // working
      // use the bathroom
}
```

Declare semaphores and other variables with initial values, and add `Wait()` and `Signal()` calls to the threads so that the man threads and woman threads will run properly and meet the requirement. 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: This is a simple variation of the reader-priority readers-writers problem. More precisely, we allow the "writers" to write simultaneously. Therefore, the writers have the same structures as the readers. We need to maintain two counters, one for the males `MaleCounter` and the other for the females `FemaleCounter`. Of course, we need two Mutexes `MaleMutex` and `FemaleMutex` for mutual exclusion. In addition, there is a semaphore `BathRoom` to block the males (*resp.*, females) if the room is being used by the females (*reap.*, males). Note that the male thread and female thread are symmetric.

```
int       MaleCounter = 0, FemaleCounter = 0;   // male and female counters
Semaphore MaleMutex = 1, FemaleMutex = 1;       // male and female counters
Semaphore BathRoom = 1;                         // the bathroom is empty initially

Male Thread                     Female Thread

while (1) {                     while(1) {
   // working                      // working

   MaleMutex.Wait();               FemaleMutex().Wait();       // update counter
      MaleCounter++;                  FemaleCounter--;
      if (MaleCounter == 1)           if (FemaleCounter == 1)   // if I am the first
         BathRoom.Wait();                BathRoom.Wait();       //    yield to other
   MaleMutex.Signal();             FemaleMutex.Signal();

   // use the bathroom             // use the bathroom

   MaleMutex.Wait();               FemaleMutex.Wait();        // update counter
      MaleCounter--;                  FemaleCounter--;
      if (MaleCounter == 0)           if (FemaleCounter == 0)  // if I am the last one
         BathRoom.Signal();              BathRoom.Signal();    //    let the other group know
   MaleMutex.Signal();             FemaleMutex.Signal();
}                               }
```

Refer to the class note for the solution to the reader-priority version of the readers-writers problem for the details. ∎