CS3331 Concurrent Computing Exam 2
Spring 2014

100 points – 8 pages

Name: ____________________________

- Most of the following questions only require short answers. Usually a few sentences would be sufficient. Please write to the point. If I don't understand what you are saying, I believe, in most cases, you don't understand the subject.
- To minimize confusion in grading, please write **readable** answers. Otherwise, it is likely I may interpret your unreadable handwriting in my way.
- *Justify your answer with a convincing argument*. An answer **must** include a convincing justification. You will receive no point for that question even though you have provided a correct answer. *I consider a good and correct justification more important than just providing a right answer. Thus, if you provide a very vague answer without a convincing argument to show your answer being correct, you will likely receive a low to very low grade.*
- You must use an execution sequence to answer each problem in this exam with a convincing argument. You will receive zero point if you do not provide a needed execution sequence, you do not elaborate your answer, and your answer is not clear or vague.
- Repeated/Recycled problems are marked with * and will be graded with the all-or-nothing principle.
- The syntax of semaphores is unimportant. You may declare and initialize a semaphore `S` with "`Sem S = 1`" and use `Wait(S)` and `Signal(S)` for semaphore wait and semaphore signal.
- Do those problems you know how to do first. Otherwise, you may not be able to complete this exam on time.

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.*

(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.**

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.

(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 <u>zero</u> point if you do not use valid execution sequences.**

(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.

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 <u>no</u> credit for this problem.

(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.

Grade Report

Problem		*Possible*	*You Received*
1	a	15	
	b	10	
2	a	10	
	b	10	
	c	15	
3	a	20	
	b	20	
Total		100	