CS3331 Concurrent Computing Exam 2
Fall 2015

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 execution sequences to answer a problem if you are asked to do so. In addition, you must include 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 a nearly 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]** The following is a solution to the critical section problem. It has two shared variables `Flag[ ]` and `turn` and a process **Scheduler** started before processes P_1 and P_2. Process **Scheduler** waits until `turn` becomes 0. Then, the `repeat-until` loop searches for a `j` such that `Flag[j]` is `TRUE`. Finally, `turn` is set to the value of `j` and loops back.

```
Boolean Flag[1..2] = { FALSE, FALSE } // note that there is no Flag[0]
int     turn = 0;
```

Process Scheduler

```
int  j;

j = 0;
repeat                        // repeat forever
   while (turn != 0)          // wait if turn is not 0
      ;                       //
   repeat                     // now turn = 0
      j = (j % 2) + 1;        //    search for a j such that
   until Flag[j];             //    Flag[j] is TRUE
   turn = j;                  // set turn to j
until FALSE;                  // loops back
```

Processes P_1 and P_2 are shown below. Both have very simple entry and exit sections.

Process P_1

```
Flag[1] = TRUE;    // interested
while (turn != 1)  // wait if not my turn
   ;
// Critical Section
Flag[1] = FALSE;   // no more interested
turn = 0;          // release my turn
```

Process P_2

```
Flag[2] = TRUE;    // interested
while (turn != 2)  // wait if not my turn
   ;
// Critical Section
Flag[2] = FALSE;   // no more interested
turn = 0;          // release my turn
```

Show rigorously that this solution satisfies the mutual exclusion and bounded waiting conditions. Moreover, state the *bound* first and prove the bounded waiting condition. **A vague and/or unconvincing proof receives no point**.

(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. **Synchronization**

(a) **[10 points]** Consider the following implementation of mutual exclusion with a semaphore X.

```
Semaphore X = 1;

Process 1         Process 2
---------         ---------
while (1) {       while (1) {
   .....             .....
   Wait(X);          Wait(X);
   // CS             // CS
   Signal(X);        Signal(X);
   ...               .....
}                 }
```

Show rigorously that the above implementation satisfies the mutual exclusion condition. **A vague and/or unconvincing proof receives no point.**

(b) **[10 points]** A programmer designed a FIFO semaphore so that the waiting processes can be released in a first-in-first-out order. This FIFO semaphore has an integer counter `Counter`, a queue of semaphores, and procedures `FIFO_Wait()` and `FIFO_Signal()`.

A semaphore `Mutex` with initial value 1 is also used. `FIFO_Wait()` uses `Mutex` to lock the procedure and checks `Counter`. If `Counter` is positive, `FIFO_Wait()` decreases `Counter` by one, unlocks the procedure, and returns. If `Counter` is zero, a semaphore `X` with initial value 0 is allocated and added to the end of the queue of semaphores. Then, `FIFO_Wait()` releases the procedure, and lets the caller wait on `X`.

Procedure `FIFO_Signal()` first locks the procedure, and checks if the semaphore queue is empty. If the queue is empty, `FIFO_Signal()` increases `Counter` by one, unlocks the procedure, and returns. If there is a waiting process in the queue, the head of the queue is removed and signaled so that the *only* waiting process on that semaphore can continue. Then, this semaphore node is freed and the procedure is unlocked.

Finally, the initialization procedure `FIFO_Init()`, not shown below, sets the counter to an initial value and the queue to empty.

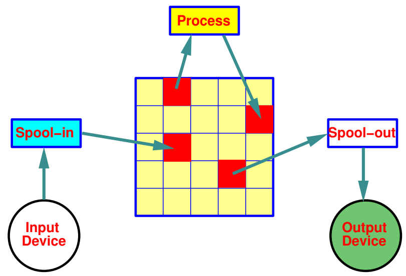
```
Semaphore  Mutex = 1;
int        Counter;
```

```
FIFO_Wait(...)
{
     Wait(Mutex);
     if (Counter > 0) {
          Counter--;
          Signal(Mutex);
     }
     else { /* must wait here */
          allocate a semaphore node, X=0;
          add X to the end of queue;
          Signal(Mutex);
          Wait(X);
     }
}
```

```
FIFO_Signal(...)
{
     Wait(Mutex);
     if (queue is empty) {
          Counter++;
          Signal(Mutex);
     }
     else { /* someone is waiting */
          remove the head X;
          Signal(X);
          free X;
          Signal(Mutex);
     }
}
```

Discuss the correctness of this solution. If you think it correctly implements a first-in-first-out semaphore, provide a convincing argument to justify your claim. Otherwise, discuss why it is wrong with an execution sequence.

(c) **[15 points]** A simplified SPOOL system has three processes: *Spool-in*, *Spool-out* and *Process*. They share a spool device, say a disk. *Spool-in* reads in input from a slow input device and copies it to the spool device, *Spool-out* sends the print output from the spool device to a slow output device, and *Process* is a user program that reads in its input from and writes its output to the spool device. To be more efficient, the spool device is divided into a number of slots and each read and write operation reads and writes exactly one slot. Once a slot is read (by *Process*) or printed (by *Spool-out*) the space occupied by this slot is considered free and can be re-used.



The following are the "rules" for performing a spooling operation:

- Initially, the spool device is empty.
- As long as the spool device has an empty slot, *Spool-in* will read the input and copy it to the spool device. If all slots are used, *Spool-in* blocks until there are free slots.
- *Process* reads its input from the spool device if there are slots that have been filled with input data by *Spool-in*; otherwise, *Process* blocks until new input data become available. After reading an input, *Process* will generate some output, one slot at a time. *Process* also blocks until there are empty slots for output.
- As long as the spool device has output slots, *Spool-out* will read and send them to the output device. *Spool-out* blocks until output data become available.
- Reading from and writing into a slot is guaranteed to be mutually exclusive.

Under what condition(s) this system will have a deadlock. You should provide an execution sequence that can lead to a deadlock. Elaborate your answer; otherwise, you may receive **<u>low</u>** or even **<u>no</u>** credit.

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]** Design a class `Group` in C++, a constructor, and method `Group_wait()` that fulfill the following specification:

- The constructor `Group(int n)` takes a positive integer argument `n`, and initializes a private `int` variable in class `Group` to have the value of `n`. The value of `n` will not change in the execution of the program.
- Method `Group_wait(void)` takes no argument. A thread that calls `Group_wait()` blocks if the number of threads being blocked is less than `n`-1. Then, the `n`-*th* calling thread releases all `n`-1 blocked threads and all `n` threads continue. Note that the system has more than `n` threads. For example, suppose `n` is initialized to 3. The first two threads that call `Group_wait()` block. When the third thread calls `Group_wait()`, the two blocked threads are released, and all three threads continue. Note that your solution **cannot** assume `n` to be 3. Otherwise, you will receive **zero** point.

Use semaphores only to implement class `Group` *and method* `Group_wait()`. Otherwise, you will receive **zero** point. Use `Sem` for semaphore declaration and initialization (*e.g.*, "`Sem S = 0;`"), `Wait(S)` on a semaphore `S`, and `Signal(S)` to signal semaphore `S`.

Your implementation should not have any busy waiting, race condition and deadlock. You should explain why your implementation is correct in detail. A vague discussion or no discussion receives no credit.

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	