CS3331 Concurrent Computing Final Exam Fall 2013

200 points – 11 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 very likely I will 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.
- Problems marked with an * are reused problems. They will be graded with a stricter standard.
- Do those problems you know how to do first. Otherwise, you may not be able to complete this exam on time.

1. **Processes and Threads**

   (a) **[10 points]*** Draw the state diagram of a process from its creation to termination, including all transitions. Make sure you will elaborate **every state** and **every transition** in the diagram.

2. **Synchronization**

   (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) [**15 points**] Consider the following solution to the critical section problem using the atomic `TS` (i.e., test-and-set) instruction. Shared variables `waiting[]` and `lock` are initialized to `FALSE`. This solution works for n processes, where $n \geq 2$ is a known integer.

```
boolean waiting[n];    // shared variable with initial value FALSE
boolean lock;          // shared lock variable with initial value FALSE

Process i (i = 0, 1, ..., n-1)

// Enter protocol
waiting[i] = TRUE;             // I am waiting to enter
key        = TRUE;             // set my local variable key to TRUE
while (waiting[i] && key)      // wait and keep trying to
   key = TS(&lock);            //    lock the shared lock variable
waiting[i] = FALSE;            // no more waiting and I am in.

// critical section

// Exit protocol
j = (i+1) % n;                 // scan the waiting status of other processes
while ((j != i) && !waiting[j]) // loop as long as process j is not waiting
   j = (j+1) % n;              //    move to next process
if (j == i)                    // no one is waiting
   lock = FALSE;               //    set the lock to FALSE
else                           // process j is waiting to enter
   waiting[j] = FALSE;         //    release j so that it can enter
```

It is obvious that this solution satisfies the mutual condition as discussed in class. Show that this solution also satisfies the progress and bounded waiting conditions. *You will receive* **zero** *point if you provide a vague answer* **without** *a convincing argument and/or you prove this by example.*

3. **Semaphores**

   (a) [**15 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, it is decreased by one, and releases the procedure. 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(...)                             FIFO_Signal(...)
{                                          {
     Wait(Mutex);                               Wait(Mutex);
     if (Counter > 0) {                         if (queue is empty) {
          Counter--;                                 Counter++;
          Signal(Mutex);                             Signal(Mutex);
     }                                          }
     else { /* must wait here */                else { /* someone is waiting */
          allocate a semaphore node, X=0;            remove the head X;
          add X to the end of queue;                 Signal(X);
          Signal(Mutex);                             free X;
          Wait(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.

(b) [**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.

4. **Monitors**

   (a) [**10 points**] Suppose in a Hoare type monitor a thread waits on condition variable `cond` if the value of an expression `expr` is negative. The value of `expr` may be modified by other threads, and the thread makes `expr` non-negative signals `cond` to release a waiting thread. There are two different ways to signal *all* threads that are waiting on `cond`. Suppose we know that the total number of waiting threads is `n`. The first method, **Method 1**, does not have to know the value of `n` and uses cascading signal as follows. In this way, a released thread can signal `cond` immediately to release another.

```
signal thread                 waiting thread
--------------               -------------------
cond.Signal();               if (expr < 0) {
                                  cond.Wait();
                                  cond.Signal();
                             }
```

   The second method, **Method 2**, uses a `for` loop as shown below:

```
signal thread                 waiting thread
--------------               -------------------
for (i=1; i <= n; i++)       if (expr < 0)
     cond.Signal();               cond.Wait();
```

   Study both methods and answer this question: In a Hoare type monitor, which method can guarantee that the expression `exp` is non-negative when a waiting thread is released?

You have to indicate which method (or both methods) satisfies the requirement and provide a convincing argument. Without doing so, you will risk low or even no credit.

(b) [**10 points**] Enumerate and elaborate all major differences between a semaphore wait/signal and a condition variable wait/signal. Vague answers and/or inaccurate or missing elaboration receive **no** credit.

5. **Deadlock**

(a) [**10 points**] What are the necessary conditions for a deadlock to occur? Name these conditions and provide an elaboration. Stating conditions without elaboration or stating a vague elaboration receives **no** credit.

(b) [**10 points**] Three processes share four resource units that can be acquired and released only one at a time. Each process needs maximum of two units at any time. We also know that these resource units must be used in a mutual exclusive way and that they are non-preemptable. Show that this is deadlock free system. **Hint**: Think about the four deadlock conditions.

(c) [**10 points**] Explain what is a livelock and explain the major difference between a deadlock and a livelock.

6. **Channels**

(a) [**10 points**] What is a *rendezvous* in message passing? **You must provide the context of a rendezvous and a clear explanation. Otherwise, you receive low or no credit.**

7. **Problem Solving**

   (a) [**25 points**] A movie theater with m seats shows movies non-stop. Customers can come and go at anytime; but, the theater cannot seat more than m customers. For some unknown reason, this theater has a design flaw. The entrance door allows customers to get in one-by-one without affecting movie showing in any way. But, opening the exit door would cause too much light into the theater. Therefore, the owner of this theater made the following rules:

   - Customers can get into the theater one at any time
   - Customers leaving the theater must be in a group of n so that the exit door will not be opened very often.
   - The entrance door cannot be used for exit and only allows customers to enter one-by-one.
   - The exit door is exit only and cannot be used to enter the theater.

   Each customer is represented as a thread as follows:

```
Customer_thread()
{
     while (1) {
          // arrival
          // enter the theater
          // have a seat and enjoy movie
          // exit
     }
}
```

   Complete the above code segment with semaphores to implement the theater's policy. **Note:** You must prevent a fast exiter who may come back into the theater, join the current exiting group, and exit again. In other word, members of the current exiting group must exit completely before admitting the next m exiting customers.

(b) [**25 points**] Each thread in a system has a unique ID, which is a positive integer. The system also has a shared file that can be accessed by multiple threads simultaneously as long as the sum of the ID's of all threads that are currently accessing the file is less than a predefined value `MAXIMUM`.

Design a Hoare monitor `Strange` and monitor procedures `Access(id)` and `Release(id)`, where `id` is the ID of the calling thread. Monitor procedure `Access(id)` allows the caller to access the file if the sum of the all ID's and `id` is less than `MAXIMUM`. In this case, `Access(id)` returns. Otherwise, the caller is blocked until the condition will be met in the future. On the other hand, when a thread finishes accessing the shared file, it calls monitor procedure `Release(id)` to release the file.

Use ThreadMentor syntax to write the monitor code.

Hint: Watch out the way of releasing threads.

(c) [**25 points**] Use semaphores to design an *asynchronous* `MailBox` class of capability `n` >0 and its two methods `Send()` and `Receive`. The `Send(int x)` takes an `int` as its argument and sends the `int` to the mailbox, and the `Receive(int *x)` receives an `int` from the mailbox. You may use a simplified syntax for semaphore declaration and operations. For example, `Sem X = 0` declare a semaphore `X` with initial value `0`, and `Signal(X)` and `Wait(X)` signals and waits on semaphore `X`, respectively.

Grade Report

Problem		*Possible*	*You Received*
1	a	10	
2	a	15	
	b	15	
3	a	15	
	b	10	
4	a	10	
	b	10	
5	a	10	
	b	10	
	c	10	
6	a	10	
7	a	25	
	b	25	
	c	25	
Total		200	