CS3331 Concurrent Computing Final Exam
Fall 2015

200 points – 13 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]*** 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) **[15 points]** The following is a simple solution to the critical section problem for n processes P_1, P_2, ..., P_n, where $n \geq 2$ is a known integer. The general code for process P_i ($1 \leq i \leq n$) is:

```
Bool  x[1..n] = FALSE;

Process Pi
----------
L : x[i] = TRUE;                    // I am interested
    for (j = 1; j <= i-1; j++) {    // check all processes Pj (1 <= j <= i-1)
       if (x[j]) {                  //    if Pj is interested
          x[i] = FALSE;             //       yield the CS
          while (x[j]);             //       busy wait until Pj not interested
          goto L;                   //       restart myself
       }
    }
    for (j = i+1; j <= n; j++)      // check all processes Pj (i+1 <= j <= n)
       while (x[j]);                //    if Pj is interested, busy wait
// in critical section              // none of the Pj's (j != i) is interested
                                    // Pi enters its CS
x[i] = FALSE;                       // Pi not interested
```

   Convert the above code to a 3-processes version for P_1, P_2 and P_3:

```
Bool  x[1..3] = FALSE;

Process 1                  Process 2              Process 3
---------                  ---------              ---------
x[1] = TRUE;               L: x[2] = TRUE;        L: x[3] = TRUE;
                                                     for (j = 1; j <= 2; j++) {
                              if (x[1]) {               if (x[j]) {
                                 x[2] =FALSE;              x[3] = FALSE;
                                 while (x[1]);             while (x[j]);
                                 goto L;                   goto L;
                              }                         }
                                                     }
for (j = 2; j <= 3; j++)
   while (x[j]);              while (x[3]);
                           // in critical section
x[1] = FALSE;                 x[2] = FALSE;          x[3] = FALSE;
```

Prove rigorously that the above algorithm for three processes P_1, P_2 and P_3 satisfies the mutual exclusion requirement. **A vague or enumeration type of proof receives NO point. An unconvincing argument receives low or even no points.**

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

```
boolean waiting[n] = FALSE;
boolean lock       = 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 was shown in class that this solution satisfies the mutual condition. 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) **[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.**

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 or methods can guarantee that the expression `exp` is non-negative when a waiting thread is released? **You have to name the method (or methods) that satisfies the requirement and provide a convincing argument. You must also indicate why the other method does not work if you indicate only method works. Without doing so, you will risk low or even no credit.**

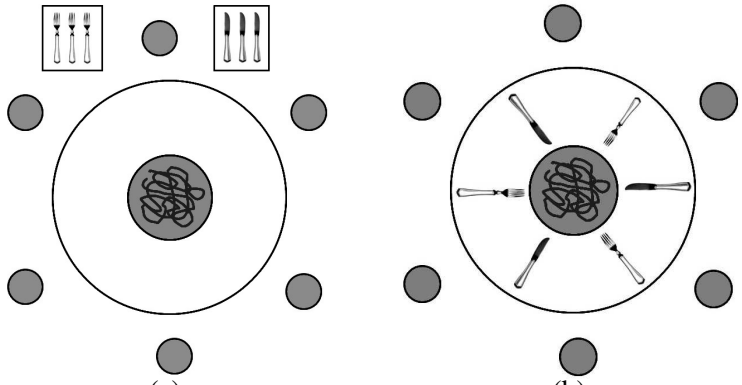
(b) **[10 points]** Enumerate and elaborate all major differences between a semaphore wait/signal and a condition variable wait/signal. Vague answers and/or inaccurate or missing elaboration receive **no** credit.

(c) **[10 points]** Why is calling a monitor procedure from within another monitor (*i.e.*, nested monitor call) not a good programming practice?

5. **Deadlock**

   (a) **[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 necessary condition of a deadlock.

   (b) **[15 points]** There are six philosophers seated at a circular table. There are three knives and three forks available. The philosophers are quite hungry, but require both a fork and a knife to eat. Do



(a) (b)

   the following two problems.

   - Each philosopher goes to the tray (Figure (a) above) and grabs any knife until he is successful. He then grabs any fork until he is successful. Once he has a fork and a knife, he eats and finally returns the fork and knife.
   - If the forks and knives are arranged in an alternating way as shown in Figure (b). Each philosopher flips a coin to determine if he is going to first try for a fork or a knife. He grabs his choice until he is successful, and grabs the other type of utensil until he is successful. Then, he eats and returns the utensil after eating.

   For each problem, answer if a deadlock is possible. If it is not, explain clearly why. If it can cause a deadlock, use an execution sequence to show the existence of a deadlock. **Make sure you state the problem and its answer clearly to avoid confusion.**

Use the space on the next page for your answer

(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

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.**

<u>Use the space on the next page for your answer</u>

6. **Channels**

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

7. **Problem Solving**

   (a) **[25 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.

(b) **[25 points]** A party room has a capability of *m* people. Each party goer can bring a number of people. However, if the sum of the number of people in a new group and the number of people in the party room exceeds the maximum capacity *m*, the new group will not be admitted and must wait outside of the party room. People in the party room must also exit in a group.

Design a Hoare monitor `PartyRoom` and monitor procedures `Admit(u)` and `Exit(v)`, where `u` is the number of people in the group requesting to enter the party room and `v` is the number of people in the group leaving the party room. When a group of `u` people arrives, a representative of this group calls `Admit(u)` to make a request to enter. If this request cannot be granted, all `u` people wait as a group. When a group of `v` people leave the party, a representative of this group calls `Exit(v)`. Because the party room has `v` people less, your monitor must allow some (or none) of those waiting groups to enter and make sure the number of people in the party room will not be larger than *m*. You may assume that entering and exit groups form automatically and can be considered as a single unit. Use ThreadMentor-like syntax to write this monitor.

(c) **[25 points]** It is not difficult to see that the `ENTRY` and `ACCEPT` pair in Ada forms a many-to-one channel, where `ENTERY` and `ACCEPT` represent blocking send and blocking receive, respectively. We also mentioned in class that semaphores, monitors and channels are equivalent to each other under the shared memory model. ThreadMentor has a synchronous many-to-one channel type as follows:

```
SynManyToOneChannel  X(*,5);
      // channel from everyone to user defined thread ID 5
```

This synchronous many-to-one channel uses the send and receive methods for sending and receiving messages:

```
X.Send(*MsgAddress, MsgSize);
X.Receive(*MsgAddress, MsgSize);
```

Based on the idea you learn from Ada and the many-to-one channel and thread capability of ThreadMentor to implement a mutex lock. More precisely, define a new class `newMutex` that has two methods `Lock()` and `Unlock()`. Then, a thread can use the following for mutual exclusion purpose:

```
myLock  newMutex;

myLock.Lock();
// critical section
myLock.Unlock();
```

Here are some important notes. **(1)** The actual ThreadMentor syntax is not so important as long as I know what you are doing; but, ambiguity can cause lower grade for sure. (2) You can only use many-to-one channels. The use of any other synchronization primitives will receive **zero** point. (3) You must use the idea of Ada that was discussed in class. Otherwise, you receive no point. (4) Provide a convincing argument to show that your implementation is correct. You will receive a low to very low score if you do not have a convincing argument.

Grade Report

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