CS3331 Concurrent Computing Exam 2 Solutions Fall 2014

1. **Synchronization Basics**

(a) [**15 points**] Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1, where `flag[2]` is a Boolean array of two elements and `turn[2]` is an `int` array, each of its two elements can only hold `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[2];   // global turn variables

Process i (i = 0 or 1)

// Enter Protocol
flag[i] = TRUE;
turn[i] = (turn[j] + i)%2;
repeat
until ( flag[j] == FALSE || turn[i] != (turn[j]+i)%2 );

// critical section

// Exit Protocol
flag[i] = FALSE;
```

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 unconvincing.*

Answer: If P_0 is in its critical section, P_0 must have passed the `repeat-until` loop and the following holds

`flag[1] == FALSE || turn[0] != (turn[1]+0)%2`

If P_1 is in its critical section, the following holds

`flag[0] == FALSE || turn[1] != (turn[0]+1)%2`

However, since `flag[i]` is set to `TRUE` at the beginning of the enter section, if P_0 and P_1 are both in their critical sections, the following must both hold.

`turn[0]  !=  (turn[1]+0)%2` (1)

`turn[1]  !=  (turn[0]+1)%2` (2)

Since the value of `turn[i]` can only be `0` or `1` because of the mod 2 operator, there are four possible combinations of `turn[0]` and `turn[1]`. As a result, we may do a complete enumeration as follows:

`turn[0]`	`turn[1]`	`(turn[1]+0)%2`	Equation (1)	`(turn[0]+1)%2`	Equation (2)
0	0	0	FALSE	1	TRUE
0	1	1	TRUE	1	FALSE
1	0	0	TRUE	0	FALSE
1	1	1	FALSE	0	TRUE

From the above table, we learn that Equation (1) and Equation (2) cannot hold at the same time. This is a contradiction, and, consequently, P_0 and P_1 cannot be in their critical sections at the same time. ∎

(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. **Note that you have to provide TWO execution sequences showing difference results to justify the existence of 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` concurrently. Note that the use of higher-level language statement interleaving may not reveal the key concept of "sharing" as discussed in class. Therefore, use instruction level interleaving instead.

See page 5 to page 10 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**] A programmer wrote a program in which child processes are created and communicate using a shared memory segment. This programmer uses the semaphore capability of ThreadMentor to avoid potential race conditions as follows:

```
Semaphore  Lock(1)  // lock open initially;

int main(void)
{
   int status, *ShmPTR;

   // create and attached a shared memory segment
   // let the pointer be ShmPTR
   if (fork() == 0) {      // child process
      Child(ShmPTR); exit();
   }
   else if (fork() == 0) { // child process
      Child(ShmPTR); exit();
   }
   else {
      wait(&status); wait(&status);
      // remove shared memory
   }
   exit();
}

int Child(int* ShmPTR)   // code for the 1st child
{
   while (1) {
      // other task
      Lock.Wait();
         // access the shared memory segment using ShmPTR
      Lock.Signal();
      // other task
   }
}
```

However, even though the initialization, process creation and shared memory section are correct, this program can never run properly. Identify the problem as clear as possible and provide a convincing explanation. **Use execution sequences if needed.**

Answer: This is an easy problem. In Unix, a child process receives an **identical** but **separate** copy of the address space of its parent. Thus, after the two `fork()` system calls, there are **three** copies of `Lock`: one is in the parent's address space, and each child process has a copy in its address space. In this way, the two child processes access their own copy of `Lock` and certainly do not have any synchronization effect. ■

(c) **[15 points]** At a child care center, state regulations require that there is always one adult present for every three children. When an adult comes to the child care center, a thread is created to simulate that adult. Similarly, when a child arrives at the center, a thread is created to simulate that child. A programmer suggested the following solution using three semaphores. His code looks like the following:

```
Semaphore  Enter = 0, Center = 0, Mutex = 1;

Adult Thread                               Child Thread
------------                               ------------
// arrive at the center                    // arrive at the center
Enter.Signal();  // admit 1st child        Enter.Wait();   // wait for an adult
Enter.Signal();  // admit 2nd child
Enter.Signal();  // admit 3rd child        // enter and play at the center
// start child care service                Center.Signal(); // done playing
Mutex.Wait();    // lock the sequence      // leave center
  Center.Wait(); // 1st child done playing
  Center.Wait(); // 2nd child done playing
  Center.Wait(); // 3rd child done playing
Mutex.Signal().  // release the lock
// leave center
```

The programmer insisted that the lock `Mutex` cannot be eliminated, because a deadlock may occur when the child care center has a certain number of adults and children (*e.g.*, 3 children and 2 adults). Find and explain this deadlock with an execution sequence and provide a convincing argument.

Answer: Suppose the program does not have the `Mutex` protection as follows:

```
Adult Thread                               Child Thread
------------                               ------------
// arrive at the center                    // arrive at the center
Enter.Signal();  // admit 1st child        Enter.Wait();   // wait for an adult
Enter.Signal();  // admit 2nd child
Enter.Signal();  // admit 3rd child        // enter and play at the center

// start child care service                Center.Signal(); // done playing
                                           // leave center
Center.Wait(); // 1st child done playing
Center.Wait(); // 2nd child done playing
Center.Wait(); // 3rd child done playing
// leave center
```

Since each adult needs three `Center.Wait()` calls to leave and since there are two adults, the total number of signals to release both adults is six. However, since there are three children, the total number of signals can be generated is only three. As a result, if an adult receives two signals and the other receives one, they will wait for the needed signals which will never come.

The following is an execution sequence that shows this problem. In the following, we use `E` and `C` to denote semaphores `Enter` and `Center`, respectively.

Child C_1	Child C_2	Child C_3	Audult A_1	Adult A_2	Comment
`E.Wait()`	`E.Wait()`	`E.Wait()`			All children arrive
			`E.Signal()`		A_1 arrives
$\downarrow$					C_1 released
			`E.Signal()`		A_1 signals 2nd time
	$\downarrow$				C_2 released
			`E.Signal()`		A_1 signals 3rd time
		$\downarrow$			C_3 released
			`C.Wait()`		A_1 waits 1st time
				`E.Signal()`	A_2 arrives
				`E.Signal()`	A_2 signals 2nd time
				`E.Signal()`	A_2 signals 3rd time
`C.Signal()`					C_1 exits, releases A_1
			`C.Wait()`		A_1 waits 2nd time
	`C.Signal()`				C_2 exits, releases A_1
			`C.Wait()`		A_1 waits 3rd time
				`C.Wait()`	A_2 waits 1st time
		`C.Signal()`			C_3 exits, releases A_2
				`C.Wait()`	A_2 waits 2nd time

Now, all children exit; but, adults A_1 and A_2 are blocked by their third and second `Center.Wait()`, respectively. Therefore, we have a deadlock. ■

3. **Problem Solving:**

   (a) **[20 points]** Let T_0, T_1, …, T_{n-1} be n threads, and let `a[ ]` be a global `int` array. Moreover, thread T_i only has access to `a[i-1]` and `a[i]` if $0 < i \leq n-1$ and thread T_0 only has access to `a[n-1]` and `a[0]`. Thus, array `a[ ]` is "circular." Additionally, each thread knows its thread ID, which is a positive integer, and is only available to thread T_i. All thread IDs are distinct. Initially, `a[i]` contains the thread ID of thread T_i. With these assumptions, we hope to find the largest thread ID of these threads.

   A possible algorithm for thread T_i ($0 < i \leq n-1$) goes as follows. Thread T_i takes T_{i-1}'s information from `a[i-1]`. If this number is smaller than T_i's thread ID, T_i ignores it as T_i has a larger thread ID. If this information is larger than T_i's thread ID, T_i saves it to `a[i]` for thread T_{i+1} to use. In this way, thread ID's are circulated and smaller ones are eliminated. Finally, if a thread sees the "received" information being equal to its own, this must be the largest thread ID among all thread ID's. Algorithm for T_0 can be obtained with simple and obvious modifications.

Thread T_i

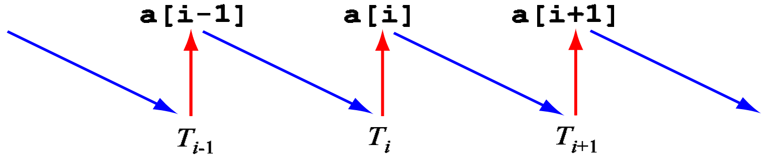
```
Thread T_i(...)
{
     // initialization: my thread ID is in a[i]
     // TID_i is my thread ID

     while (not done) {
          if (a[i-1] == -1) {
               // game over, my thread ID is not the largest
               break;
          }
          else if (a[i-1] == TID_i) {
               // my TID is the largest, break
               a[i] = -1;  // tell everyone to break
               printf("My thread ID %d is the largest\n", TID_i);
               break;
          }
          else if (a[i-1] > TID_i) {
               // someone's thread ID is larger than mine
               a[i] = a[i-1];   // pass it to my neighbor
          }
          else {
               // so far, my thread ID is still the largest
               // do nothing
          }
     }
     // do something else
}
```

Obviously, race conditions are everywhere. Declare and add semaphores to the above code section so that the indicated task can be performed correctly. You may use as many semaphores as you want. However, thread T_i can only share its resource, semaphores included, with its left neighbor T_{i-1} and right neighbor T_{i+1}. To make this problem a bit less complex, you may ignore thread T_0.

You may use type `Sem` for semaphore declaration and initialization (*e.g.*, "`Sem S = 0;`"), `Wait(S)` on a semaphore `S`, and `Signal(S)` to signal semaphore `S`.

You should explain why your implementation is correct in details. A vague discussion or no discussion receives <u>zero</u> point.

<u>Answer</u>: This is a very easy problem and is similar to the dining philosophers problem. Since thread T_i (*i.e.*, a philosopher) takes information out of `a[i-1]` (*i.e.*, left fork) and stores new information into `a[i]` (*i.e.*, right fork), `a[i-1]` is shared between threads T_{i-1} and T_i and `a[i]` is shared between threads T_i and T_{i+1}. See the diagram below. Therefore, `a[i]` should be protected by a semaphore, say `S[i]` with initial value `1`. Thread T_i must lock semaphore `S[i-1]` (*resp.*, `S[i]`) before accessing `a[i-1]` (*resp.*, `a[i]`), and release the semaphore after its access.



The following is a possible solution:

Thread T_i

```
#define    END_GAME    (-1)

Semaphore  S[n] = { 1, 1, ..., 1 };

Thread T_i(...)
{
     int   in;                        // local variables

     // initialization: my thread ID is in a[i]
     // TID_i is my thread ID

     while (not done) {
          Wait(S[i-1];                // locks a[i-1]
             in = a[i-1];             // retrieve information
          Signal(S[i-1]);             // release a[i-1]

          if (in == END_GAME) {
               // game over, my thread ID is not the largest
               break;
          }
          else if (in == TID_i) {
               // my TID is the largest, break
               Wait(S[i]);            // lock a[i]
                    a[i] = END_GAME;  // tell everyone to break
               Signal(S[i]);
               printf("My thread ID %d is the largest\n", TID_i);
               break;
          }
          else if (in > TID_i) {
               // someone's thread ID is larger than mine
               Wait(S[i]);            // lock a[i]
                    a[i] = in;        // pass it to my neighbor
               Signal(S[i]);
          }
          else {
               // so far, my thread ID is still the largest
               // do nothing
          }
     }
     // do something else
}
```

The above solution uses a local variable `in` to store the value of `a[i-1]` at the beginning of each iteration to avoid locking `a[i-1]` for a long time. In this way, after retrieving the current value of `a[i-1]`, it is released for thread T_{i-1} to have a new update.

A few of you used one and only one semaphore `S` with initial value `1` as follows. This is a terribly wrong solution. **First**, this solution serializes the whole system. In other words, only one thread can execute the `while` loop at any time, which violates the maximum parallelism requirement. **Second**, this solution is incorrect because it violates the "passing the information along" requirement. As long as thread T_i passes `Wait(S)`, it retrieves `a[i-1]`, which may or may not be updated by thread T_{i-1}. Thus, T_i may have to iterate for an unknown number of iterations to get an updated `a[i-1]`. This, of course, wastes CPU time, in addition to serialization. **Third**, an extreme case is that the same thread T_i keeps entering its critical section. In this case, no other threads can update their information, and, the system will not be able to complete its required task! Consequently, this is an <u>**incorrect**</u> solution.

Thread T_i

```
#define    END_GAME    (-1)

Semaphore  S = 1;

Thread T_i(...)
{
     int   in;                        // local variables

     // initialization: my thread ID is in a[i]
     // TID_i is my thread ID

     while (not done) {
          Wait(S);
          if (in == END_GAME) {
               // game over, my thread ID is not the largest
               break;
          }
          else if (in == TID_i) {
               // my TID is the largest, break
               a[i] = END_GAME;       // tell everyone to break
               printf("My thread ID %d is the largest\n", TID_i);
               break;
          }
          else if (in > TID_i) {
               // someone's thread ID is larger than mine
               a[i] = in;        // pass it to my neighbor
          }
          else {
               // so far, my thread ID is still the largest
               // do nothing
          }
          Signal(S);
     }
     // do something else
}
```

(b) [**20 points**] Three kinds of threads share access to a singly-linked list: *searchers*, *inserters* and *deleters*. Searchers only examine the list, and can execute concurrently with each other. Inserters append new nodes to the end of the list. Insertions must be mutually exclusive to preclude two inserters from inserting new nodes at about the same time. However, one insertion can proceed in parallel with any number of searches. Finally, deleters remove nodes from anywhere in the list. At most one deleter can access the list at a time, and deletion must also be mutually exclusive with searches and insertions.

Obviously, searchers and deleters are exactly the readers and writers, respectively, in the readers-writers problem. The following shows the code for searcher and deleter. They are actually a line-by-line translation of the readers-writers solution.

```
Semaphore  Mutex = 1;            // for locking the Counter
Semaphore  listProtection = 1;   // for list protection
int        Count = 0;

Searcher                               Deleter
--------                               -------
while (1) {                            while (1) {
   Wait(Mutex);                           Wait(listProtection);
      Count++;                               // delete a node
      if (Count == 1)                     Signal(listProtection);
         Wait(listProtection);            // do something
   Signal(Mutex);                      }

   // do search work

   Wait(Mutex);
      Count--;
      if (Count == 0)
         Signal(listProtection);
   Signal(Mutex);

   // use the data
}
```

Write the code for the inserter and add semaphores and variables as needed. **You are not supposed to modify the Searcher and Deleter.** You may use the simple syntax discussed in class (and in class notes) rather than that of ThreadMentor. **You must provide a convincing elaboration to show the correctness of your program.**

Answer: Look at the code carefully and you should be able to see that the searchers and deleters are exactly the readers and writers, respectively, in the Readers-Writers problem. Searchers have concurrent access to the linked-list, while deleters must acquire an exclusive use. The new inserters are actually special searchers, because an inserter runs concurrently with others searchers; but only one inserter can run at any time. In other words, inserters are hybrid searchers and deleters. Therefore, we may reuse the code of searcher for concurrent access with a lock to ensure only one inserter can run at any time.

The following is a possible solution. Semaphore `insertProtection` makes sure only one inserter can have access to the list. Mutual exclusion among all threads (*i.e.*, searchers, inserters, and deleters) are enforced by semaphore `listProtection` as in the readers-writers problem.

```
Semaphore  Mutex = 1;              // for locking the Counter
Semaphore  listProtection = 1;     // for list protection
Semaphore  insertProtection = 1;   // for blocking other inserters
int        Count = 0;

while (1) {
   Wait(insertProtection);         // mutual exclusion for inserter
      Wait(Mutex);
         Count++;
         if (Count == 1)
            Wait(listProtection);
      Signal(Mutex);

   // do insertion

      Wait(Mutex);
         Count--;
         if (Count == 0)
            Signal(listProtection);
      Signal(Mutex);
   Signal(insertProtection);

   // do other thing
}
```

In this way, one inserter and multiple searchers can run concurrently. They use semaphore `Mutex` to maintain the counter `Count` and communicate with deleters with semaphore `listProtection` so that a deleter has exclusive access to the list.

Some may suggest the following solution. The only difference is moving the inserter lock from the beginning to very close to the insert operation.

```
Semaphore  Mutex = 1;             // for locking the Counter
Semaphore  listProtection = 1;    // for list protection
Semaphore  insertProtection = 1;  // for blocking other inserters
int        Count = 0;

while (1) {
  Wait(Mutex);
     Count++;
     if (Count == 1)
        Wait(listProtection);
  Signal(Mutex);

  Wait(insertProtection);        // mutual exclusion for inserter
   // do insertion
  Signal(insertProtection);

  Wait(Mutex);
     Count--;
     if (Count == 0)
        Signal(listProtection);
  Signal(Mutex);

  // do other thing
}
```

The problem is that inserters and searchers may compete to lock the counter, and, as a result, may clog the system. On the other hand, the original version guarantees that no more than one inserter can join the competition, and is more efficient.

Now consider another problematic solution:

```
Semaphore  Mutex = 1;             // for locking the Counter
Semaphore  listProtection = 1;    // for list protection
Semaphore  insertProtection = 1;  // for blocking other inserters
int        Count = 0;

while (1) {
  Wait(Mutex);
     Wait(insertProtection);      // insert protection wait moved here
     if (Count == 0)
        Wait(listProtection);
  Signal(Mutex);

  // do insertion

  Wait(Mutex);
     Signal(insertProtection);    // insert protection signal moved here
     if (Count == 0)
        Signal(listProtection);
  Signal(Mutex);

  // do other thing
}
```

This is a terribly wrong solution. While an inserter is inserting, all searchers could finish their work and `Count` becomes 0 since this inserter does not update *Count*. Then, the last searcher signals `listProtection` allowing a deleter to delete. In this way, an inserter and a deleter could run at the same time, violating the given condition. ∎