CS3331 Concurrent Computing Solutions 2
Spring 2016

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.*

   Answer: A process that enters its critical section must first set `flag[i] = IN_CS` and then see `flag[j] != IN_CS` being true at the end of the `repeat-until` loop. Therefore, if process P_0 is in the critical section, it had executed `flag[0] = IN_CS` followed by seeing `flag[1] != IN_CS`. By the same reason, if process P_1 is in the critical section, it had executed `flag[1] = IN_CS` followed by seeing `flag[0] != IN_CS`. Consequently, if P_0 and P_1 are both in the critical section, we have `flag[0] = IN_CS` and `flag[1] != IN_CS` (from P_0's point of view) and `flag[1] = IN_CS` and `flag[0] != IN_CS` (from P_1's point of view). As a result, `flag[0]` and `flag[1]` are equal to `IN_CS` and not equal to `IN_CS` at the same time. This is impossible, and the mutual exclusion condition holds.

   Note that the variable `turn` does not play a role here. Right after P_0 and P_1 pass their `repeat-until` loop, they will store some value to `turn`. At this point, because P_0 and P_1 will be in their critical sections without any obstruction, and the value in `turn` does not matter.

   See page 7 of `06-Sync-Soft-Hardware.pdf`. This is the same technique as the one we used to show that Attempt II satisfies the mutual exclusion condition. ∎

   (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, one for each result, 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 we have 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 machine instruction level interleaving instead.

See pp. 5–12 of `05-Sync-Basics.pdf`. ■

2. **Synchronization**

(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. **You have to show clearly how the mutual exclusion condition is violated with execution sequences and provide a convincing explanation. Otherwise, you will risk a lower score.**

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. Additionally, if you

failed to indicate how the needed critical sections that require mutual exclusion is formed, you also risk a lower grade.
This problem was assigned as an exercise in class. See slide 8 of `08-Semaphores.pdf`. ∎

(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:

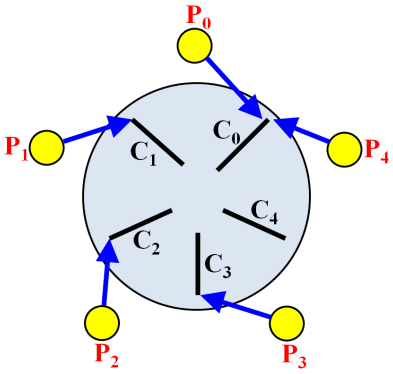
```
Semaphore  Lock(1)  // lock open initially;

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) **[20 points]** The following problems are about the dining philosophers problem. However, all of them are variations of the original version. **You have to answer each question as precisely as possible, and prove your claim with rigor. A vague answer receive zero point**.

i. **[5 points]** Suppose all five chopsticks are placed in a tray at the center of the table and any philosopher can pick up any chopstick. Now, all philosophers must pick up his first chopstick, followed by his second chopstick. In other words, each philosopher must pick up his two chopsticks one by one. Is deadlock possible? If you think this solution is deadlock-free, prove it rigorously. If you think a deadlock is possible, use an explicit and valid execution sequence to demonstrate how this deadlock happens.
Answer: This is a very simple problem, which is a variation of the original version. **A deadlock is possible.** If every philosopher picks up a chopstick from the tray at the center of the table, then each philosopher has one chopstick and is waiting for the other. Hence, a deadlock occurs. ∎

ii. **[5 points]** Suppose each chopstick is assigned a number. For example, C_i means chopstick i, where $i = 0, 1, 2, 3, 4$. Each philosopher must pick up a chopstick of lower number, and then a chopstick of higher number after he has successfully obtained one chopstick. For example, philosopher 3 picks up chopstick C_3 and then chopstick C_4. Is deadlock possible? If you think this solution is deadlock-free, prove it rigorously. If you think a deadlock is possible, use an explicit and valid execution sequence to demonstrate how this deadlock happens.
Answer: **This solution is the 1-weirdo version stated in a different way. Therefore, it is deadlock free.** The problem states that a philosopher picks up a lower number chopstick first and then a higher number one. More specifically, philosopher P_i picks up chopstick C_i first, followed by chopstick C_{i+1} if C_i is available. The following diagram shows the situation.



Philosopher P_0 first picks up chopstick C_0 followed by C_1. Philosopher P_1 first picks up chopstick C_1 followed by C_2. Philosopher P_2 first picks up chopstick C_2 followed by C_3. Philosopher P_3 first picks up chopstick C_3 followed by C_4. Philosopher P_4 must have C_4 and C_0. Because the numbers of C_4 and C_0 are 4 and 0, respectively, philosopher P_4 must pick chopstick C_0 followed by C_4. Therefore, philosopher P_4 is the weirdo (*i.e.*, righty) in the 1-weirdo version. Because we know that the 1-weirdo version is deadlock free, this version is also deadlock free. ∎

iii. **[5 points]** Adding more resources usually can avoid possible deadlocks. Suppose an additional (*i.e.*, sixth) chopstick is placed at the center of the table. A philosopher picks up his left chopstick as usual, and then competes to grab the chopstick at the center of the table. If he can get both, then he eats. Is deadlock possible? If you think this solution is deadlock-free, prove it rigorously. If you think a deadlock is possible, use an explicit and valid execution sequence to demonstrate how this deadlock happens.
Answer: **This approach is deadlock free.** If a philosopher has picked up his left chopstick, he must compete to get the center one. As long as a philosopher could get the center one, this philosopher can eat. In fact, there is always one and only one philosopher can get the chopstick at the center due to mutual exclusion. Hence, the system is deadlock free. ∎

iv. **[5 points]** Suppose the above problem is modified a little bit. A philosopher picks up his left chopstick as usual, and then competes to grab the chopstick at the center of the table. **If the chopstick at the center of the table has been taken, the philosopher tries to grab his right chopstick.** Is deadlock possible? If you think this solution is deadlock-free, prove it rigorously. If you think a deadlock is possible, use an explicit and valid execution sequence to demonstrate how this deadlock happens. Moreover, use semaphores to design a code segment so that each philosopher can test to see if the chopstick at the center of the table is available, and give up (if the center chopstick is taken) or grab it (if the center chopstick is available).
Answer: **This approach is deadlock free.** If a philosopher has picked up his left chopstick, he then competes to get the center one. Because each philosopher has one chopstick and because the center one is available to one and only one philosopher, the philosopher who has the center one can eat. For the remaining philosophers, if he fails to get the center one, this philosopher eats if he can pick up his right chopstick, or this philosopher waits. Therefore, at any time, at least one philosopher can eat. After eating, a philosopher puts back his left and right chopsticks or his left and the center chopsticks, and another philosopher who has his left can compete for the center one. Hence, this is a deadlock free approach.
The following pseudo code illustrates the thinking and eating cycle for philosopher i.

Philosopher *i*

```
bool        CenterChop = FREE;      // state of the center chopstick
Semaphore   Chop[5] = { 1, 1, 1, 1, 1 };  // mutual exclusion
Semaphore   Center = 1;             // lock to protect the state of
                                    //     the center chopstick

while (1)
{
    // thinking

    Chap[i].Wait();                  // get the left chopstick
    Center.Wait();                   // lock the CenterChop state variable
    if (CenterChop == FREE) {        // chopstick at center is available
        CenterChop = TAKEN;          //     take it
        // eat
        CenterChop = FREE;           // set it to free after eating
        Center.Signal();             // release the center one
        Chop[i].Signal();            // release his left
    }
    else {                           // center one not available
        Center.Signal();             // release the lock
        Chop[(i+1)%5].Wait();        // get the right one
        // eat
        Chop[(i+1)%5].Signal();      // release right
        Chop[i].Signal();            // release left
    }
}
```

Note that this is a complete code. However, it is easy for you to see how to test whether the center chopstick is available. ■

3. **Problem Solving:**

(a) **[15 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. Otherwise, you will receive very low grade or even zero point.**

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 Reader-Writer 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 reader-writer 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. ∎

(b) **[20 points]** A barber shop has one barber, one barber chair, and *n* chair for waiting customers to sit in. The barber's activities are as follows:

- The barber waits for customers.
- If there is a customer, the barber brings one of the waiting customers to the barber chair and cuts his hair.
- After serving a customer, the barber sleeps.
- If there are no waiting customers, the barber simply sleeps.

The activities of each customer are shown below:

- When a customer arrives, he looks to see whether there is a free chair in the waiting room.
- If there is a free chair in the waiting room, the customer tells the barber that there is a new customer, sits in, and waits for his turn .
- If there is no free chair, this customer leaves.

The barber is simulated by a thread. When he comes to work, he just calls function `Barber()`. The customers are simulated by dynamically created threads. When a customer needs a hair cut, he calls function `Customer()`. Note that `cut_hair()` and `get_haircut()` are known functions that do not require your attention and implementation.

Barber

```
void Barber(void)
{
   while (1) {
      // barber may be sleeping
      // barber wakes up to work
      cut_hair();
   }
}
```

Customer

```
void Customer(void)
{
   // comes to the shop
   // follow the steps discussed above
   get_haircut();
   // it is done.  good bye
}
```

Write the code for functions `Barber()` and `Customer()` and add semaphores and variables as needed. 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: The description has provided you with all ingredients for you to solve this problem. What you need is adding appropriate semaphore operations at the right places. Let us analyze the activities of the barber first:

- The barber waits (or sleeps) until a customer comes. We need a semaphore for the barber to block on and for a customer to wake up the barber. Let this semaphore be `CustomerReady`. Its initial value is `0` because the barber shop has no customers at the beginning.
- By the same reason, when the barber is ready to do the next hair-cut, he needs a semaphore to notify a customer so that the customer could proceed to the barber's chair. Let this semaphore be `BarberReady` with initial value `0` because the barber is not ready initially.
- Since customers are sitting on chairs while they are waiting, each customer (or the barber) must increase or decrease the number of available chairs. To avoid possible race conditions, we need a third semaphore to protect this counting (*i.e.*, mutual exclusion). Let the chair counter be `Chairs` with an initial value `n`, where `n` is a positive integer, and the mutex for protection be `Mutex` with initial value `1` so that the first one arrives could manipulate `Chairs`.

In summary, we have the following declarations:

```
Semaphore  CustomerReady = 0;    // blocks the barber until a customer comes
Semaphore  BarberReady = 0;      // blocks customers until the barber is ready
Semaphore  Mutex = 1;            // mutual exclusion protecting variable Chairs

int        Chairs = n;           // number of chairs, where n is a positive integer
```

With the above semaphores and counter in hand, the barber code is straightforward:

```
void Barber(void)
{
   while (1) {
      Wait(CustomerReady);        // wait for a customer
      Wait(Mutex);                // barber has been awaken by a customer
                                  //    and gets a customer
      Chairs++;                   // the barber increases the number of free chairs
      Signal(BarberReady);        // the barber releases a customer for hair-cutting
      Signal(Mutex);              // the barber releases the mutex lock
      cut_hair();
   }
}
```

The customer part is also easy. Each customer has the following activities:

- A customer first checks to see if there is a free chair. This customer must lock `mutex` and examine the value of `Chairs`. She/He joins the game only if `Chairs` is positive (*i.e.*, a free chair is available). Otherwise, s/he just unlocks `Mutex` and leaves.
- If there are free chairs, this customer decreases `Chairs` by `1` and wakes up the barber (*i.e.*, `Signal(CustomerReady)`).
- Then, this customer releases the lock `Mutex` and waits for the barber's call. This is done with `Signal(Mutex)` for the former and `Wait(BarberReady)` for the latter.
- Once this customer is released from `Wait(BarberReady)`, s/he is ready for a hair cut.

```
void Customer(void)
{
   Wait(Mutex);                   // wait to access variable Chairs
   if (Chairs > 0) {              // free chairs are available
      Chairs--;                   // get a chair
      Signal(CustomerReady);      // tells the barber that a customer is waiting
      Signal(Mutex);              // release the lock
      Wait(BarberReady);          // waits for the barber's call
   }
   else                           // there is no free chairs
      Signal(Mutex);              // release the lock and go away
}
```

As you can see from the above, all key elements have been discussed extensively in class through various examples. The semaphore `Mutex` serves as a lock for the purpose of mutual exclusion, and semaphores `CustomerReady` and `BarberReady` are used for notification between the barber and customers. If you follow the lectures, understand the concepts behind the many examples discussed in class, and do the exercises, this barber shop problem is not difficult. In fact, in terms of complexity, this problem is easier than the readers-writers problem and the roller-coaster problem. As mentioned many times in class, you should avoid the use of switches to record the states of an event such as the barber or customers are waiting, because this is still a sequential way of thinking, and, more importantly, because you will have more chances to put race conditions into your solution. ■