CS3331 Concurrent Computing Exam 2 Solutions Spring 2015

1. **Synchronization Basics**

   (a) [**15 points**] Consider the following solution to the critical section problem for two processes P_0 and P_1.

```
bool  flag[2];   // global flags, initially FALSE
int   turn;      // global turn variable, initially 0 or 1

Process 0                Process 1

// enter protocol        // enter protocol
flag[0] = TRUE;          flag[1] = TRUE;        // I am interested
while (flag[1]) {        while (flag[0]) {      // wait if you are interested
  if (turn == 1) {         if (turn == 0) {     //   if it is your turn ...
    flag[0] = FALSE;         flag[1] = FALSE;   //     I am no more interested
    while (turn != 0)        while (turn != 1)  //     wait for my turn
      ;                        ;
    flag[0] = TRUE;          flag[1] = TRUE;    //     let me try again
  }                        }
}                        }

// in critical section   // in critical section

// exit protocol         // exit protocol
turn = 1;                turn = 0;              // it is your turn now
flag[0] = FALSE;         flag[1] = FALSE;       // I am not interested
```

   Show that this solution satisfies the mutual exclusion condition. *You will receive* **zero** *point if you provide a vague answer* **without** *a convincing argument and/or you prove this by example.*

   Answer: Process 0 can enter its critical section if the condition of the `while` loop fails. This means `flag[1]` must be `FALSE`. Furthermore, process 0 sets `flag[0]` to `TRUE` before entering or going back for the next iteration of the `while` loop. Therefore, process 0 can enter its critical section if `flag[0]` is `TRUE` **and** `flag[1]` is `FALSE`. Note that `turn` plays no role here. Suppose process 0 runs fast, so fast that it sets `flag[0]` to `TRUE` and sees `flag[1]` being `FALSE`. Then, process 0 enters no matter what value `turn` has. By the same reason, process 1 can enter its critical section if `flag[1]` is `TRUE` **and** `flag[0]` is `FALSE`.

   If process 0 and process 1 are both in their critical sections at the same time, `flag[0]` (and `flag[1]`) is both `TRUE` and `FALSE`, which is impossible. Consequently, process 0 and process 1 cannot be in their critical section at the time, and the mutual exclusion conditions is satisfied.

   This proof is exactly identical to the one shown on slide 7 of `06-Sync-Soft-Hardware.pdf`. ∎

   (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 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. **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.

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]** The `TS` (*i.e.*, Test-and-Set) instruction is an atomic instruction. It has the following form:

```
bool TS(bool *key)
{
     bool save = *key;
     key = TRUE;
     return save;
}
```

Consider the following implementation of mutual exclusion by the `TS` instruction:

```
bool  lock = FALSE;      // a global variable

do {
    // other work
    while (TS(&lock));   // entry section
        // critical section
    lock = FALSE;        // exit section
    // other work
} while (1);
```

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

Answer: This is a very simple problem as discussed on slide 26 of `06-Sync-Soft-Hardware.pdf`. We shall prove that the implementation satisfies the mutual exclusion condition.

Assume that processes P_0 and P_1 are in their critical sections at the same time. This means that both processes passed the `while` loop and saw the returned value of the `TS` instruction being `FALSE`. Since `TS` is an atomic instruction, P_0 and P_1 cannot execute it at the same time. Without loss of generality, assume that P_0 executes the `TS` first. In this case, P_0 saw the returned value of `TS` being `FALSE` and enters its critical section. Right after P_0 finishes the execution of `TS`, variable `lock` has been changed to `TRUE` by the `TS` instruction. By the time P_1 executes `TS`, the returned value has become `TRUE`, which causes P_1 to loop and cannot enter. Consequently, P_0 and P_1 cannot be in their critical sections at the same time. Thus, we have a contradiction and the mutual exclusion condition is met.

This proof was discussed in class and is on slide 26 of `06-Sync-Soft-Hardware.pdf`. ∎

(c) **[15 points]** Show that the 1-weirdo solution to the dining philosophers problem will not cause circular waiting and hence is deadlock free. **You should prove this rigorously. A vague and/or unconvincing argument is not acceptable and will receive no points.**

Answer: Suppose the weirdo is philosopher 5. We have two cases to consider: **(1)** if Philosopher 5 has his right chopstick, and **(2)** if Philosopher 5 does not have his right chopstick.

- **Philosopher 5 Has His Right Chopstick:**
  In this case, Philosopher 1 does not have his left chopsticks and cannot eat. See Figure (a) below. The worst case is that philosophers 2, 3 and 4 have their left chopsticks and are waiting for their right ones. Depending on if Philosopher 5 has his left chopstick, we have the following two possibilities.
  - **Philosopher 5 Has His Left Chopstick:**
    In this case, Philosopher 5 has both chopsticks and can start eating. There is no deadlock. See Figure (b) below.
  - **Philosopher 5 Does Not Have His Left Chopstick:**
    In this case, Philosopher 5's left chopstick is being held by Philosopher 4 as his right chopstick, and, of course, Philosopher 4 can eat. There is no deadlock either. See Figure (c) below.
- **Philosopher 5 Does Not Have His Right Chopstick:**
  In this case, Philosopher 5's right chopstick is being held by Philosopher 1 as his left chopstick,

and, hence, Philosopher 5 cannot eat. See Figure (d) below. The worst case possible is that Philosopher 1 to Philosopher 4 all have their left chopsticks. Since Philosopher 5 cannot eat, his left chopstick is free and can be used by Philosopher 4 as his right chopstick. Therefore, Philosopher 4 can eat, and there is no deadlock.

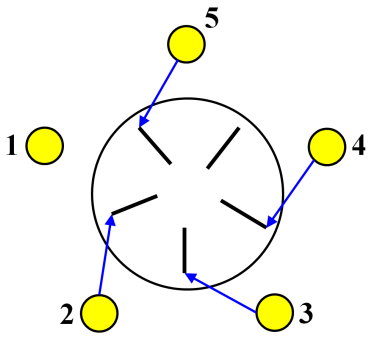


Figure (a)

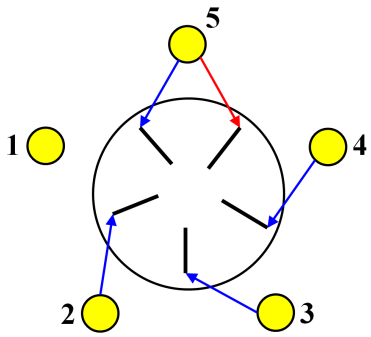


Figure (b)

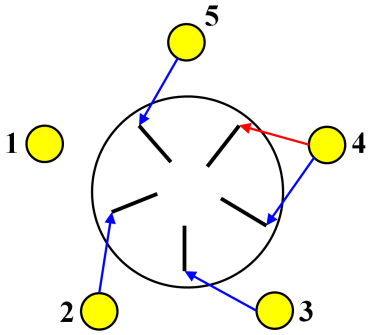


Figure (c)

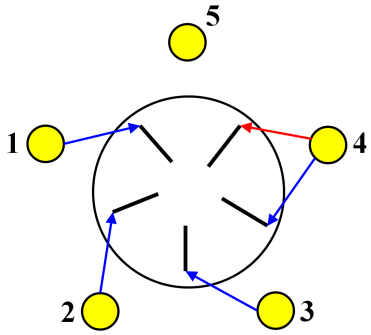


Figure (d)

Since the above enumeration is exhaustive and since none of these case can cause a deadlock, we conclude that the 1-weirdo version is deadlock free. ∎

3. **Problem Solving:**

(a) **[20 points]** A restaurant has n tables, each of which can only sit one customer, and has the following rules.

- Initially, the restaurant has no customers and all tables are free.
- When a customer arrives, if there is a free table and no one is waiting, he could sit down and order food.
- When a customer arrives, if all tables are occupied or there are waiting customers, he must wait until all eating customers finish eating and leave.
- After finishing his meal, a customer leaves.

Design a customer thread with semaphores to simulate this activity. Your implementation should not have any busy waiting, race conditions, and deadlocks, and should aim for maximum parallelism. **You must provide a convincing elaboration to show the correctness of your program.**

Hint: This problem is very similar to the readers-writers problem discussed in class, except that there is no writer involved. Think about what a customer must do when he arrives and when he leaves. You may have to keep track two counters, one counting the number of waiting customer and the other counting the number of eating customers. The rest is similar to what a reader does in the readers-writers problem.

Answer: The following is a possible solution.

```
int  Waiting = Eating = 0;  // counting waiting and eating customers
int  i, k;                  // working variables
Bool Must_Wait = FALSE;     // variable indicating if a newcomer must wait

Semaphore  Mutex = 1;       // lock for protecting Waiting, Eating and Must_Wait
Semaphore  Table = 0;       // block customers

// A customer thread

Mutex.Wait();               // newcomer must lock the variables
if (Must_Wait) {            // there are waiting customers or no table is available?
   Waiting++;               //    must wait and increase Waiting
   Mutex.Signal();          //    release the lock before waiting
   Table.Wait();            //    and join the waiting line
}
else {                      // no waiting customers and a table available
   Eating++;                // this customer can have a table
   if (Eating == n)         // if he is the last one who gets a table
      Must_Wait = TRUE;     //    he must set Must_Wait to TRUE
   Mutex.Signal();          // release the lock and go eat
}

// This customer has a table and easts

Mutex.Wait();               // finish eating and lock variables
Eating--;                   // decrease Eating
if (Eating == 0) {          // if this is the last customer
   k = (Waiting <= n) ? Waiting : n;  // allow no more than n to go
   Waiting -= k;            //    release k waiting customers
   Eating += k;             //    they can have tables
   Must_Wait = (k == n)     //    if this is a full count, newcomers must wait
   for (i = 1; i <= k; i++) //    release those k waiting customers
      Table.Signal();
}
Mutex.Signal();             // unlock variables
```

The following is the list of requirements:

- Initially, the restaurant has no customers and all tables are free.
- When a customer arrives, if there is a free table and no one is waiting, he could sit down and order food.
- When a customer arrives, if all tables are occupied or there are waiting customers, he must wait until all eating customers finish eating and leave.
- After finishing his food, a customer leaves.

The second and third are the key elements. A customer can have a table only if there is a free one and no one is waiting. A customer must wait if there are other customers waiting, even though there is a free table. Therefore, we have to keep track two counters as stated in the hint, one counting the number of eating customers (`Eating`), and the other counting the number of waiting customers (`Waiting`). Because every customer thread must update these counters, we need a semaphore `Mutex` to ensure mutual exclusion. Semaphore `Mutex` also protects a shared variable `Must_Wait` which is used to tell if a new customer must wait or not. Since a customer blocks if there are waiting customers or no table is available, we need a semaphore `Table`, with initial value 0, for this purpose. The following shows the declarations:

```
int  Waiting = Eating = 0;  // counting waiting and eating customers
int  i, k;                  // working variables
Bool Must_Wait = FALSE;     // variable indicating if a newcomer must wait

Semaphore  Mutex = 1;       // lock for protecting Waiting, Eating and Must_Wait
Semaphore  Table = 0;       // block customers
```

A new customer checks `Must_Wait` to see if she must wait. `Must_Wait` is `TRUE` if there are waiting customers and/or all tables are allocated. If this customer must wait, she updates `Waiting`, releases the lock `Mutex`, and waits on semaphore `Table`. Otherwise (*i.e.*, `Must_Wait` being `FALSE`), there are no waiting customer and there is an empty table. This customer can have a table and eat. She updates `Eating`, sets `Must_Wait` to `TRUE` if she is the last one who can get a table, and unlocks `Mutex`.

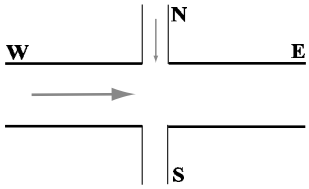
```
Mutex.Wait();               // newcomer must lock the variables
if (Must_Wait) {            // there are waiting customers or no table is available?
   Waiting++;               //    must wait and increase Waiting
   Mutex.Signal();          //    release the lock before waiting
   Table.Wait();            //    and join the waiting line
}
else {                      // no waiting customers and a table available
   Eating++;                // this customer can have a table
   if (Eating == n)         // if he is the last one who gets a table
      Must_Wait = TRUE;     //    he must set Must_Wait to TRUE
   Mutex.Signal();          // release the lock and go eat
}
```

After eating, a customer must decrease `Eating`. If she is the last one, she is responsible to release enough number of waiting customers. As a result, this last eating customer must check the number of waiting customers. If there are more than `n` customers, only `n` of them can be released. On the other hand, if there are less than `n` waiting customers, all of them can be released. This is the reason that counter `Waiting` is needed. In the above solution, the last eating customer determines `k`, which is the number of customers to be released. Since these `k` customers are about to eat, `Waiting` decreases by `k` and `Eating` increases by `k`. However, if `k` is equal to `n` (*i.e.*, full count), the released customers will occupy all tables, and other customers, those new arrivals included, must wait. Therefore, `Must_Wait` must be set to `TRUE`. Then, this last customer signals semaphore `Table` `k` times, allowing `k` customers to be released. Finally, she releases `Mutex` so that other customers can acquire and proceed.

```
Mutex.Wait();               // finish eating and lock variables
Eating--;                   // decrease Eating
if (Eating == 0) {          // if this is the last customer
   k = (Waiting <= n) ? Waiting : n;  // allow no more than n to go
   Waiting -= k;            //    release k waiting customers
   Eating += k;             //    they can have tables
   Must_Wait = (k == n)     //    if this is a full count, newcomers must wait
   for (i = 1; i <= k; i++) //    release those k waiting customers
      Table.Signal();
}
Mutex.Signal();             // unlock variables
```

From this discussion, you should be able to see that this problem is very similar to the readers-writers problem. Although there is no writers, the concept of having the first thread and the last thread doing the control is similar. ∎

(b) **[20 points]** A main highway cuts through a rural road as shown below. East-bound vehicles are on the highway, while south-bound vehicles are on the rural road. To avoid delay on the highway, the following traffic regulations are implemented:



- As long as there are east-bound vehicles crossing the intersection, east-bound vehicles do not have to stop, and they just follow the traffic flow. In this case, south-bound vehicles must stop at the intersection.
- If there is a south-bound vehicle crossing the intersection, all east-bound vehicles must stop at the intersection.
- To prevent south-bound vehicles from blocking the highway, only **one** south-bound vehicle can proceed and enter the intersection. On the other hand, multiple east-bound vehicles may cross the intersection at the same time.
- If east-bound vehicles and south-bound vehicles approach the intersection at the same time, only one can proceed. This vehicle may be east-bound or south-bound.

The east-bound and south-bound vehicles are run as threads and use the following template:

```
east-bound(...)                          south-bound(...)
{                                        {
      // approach the intersection               // approach the intersection
      wait or proceed.  add code here.           wait or proceed.  add code here.
      // exit the intersection                   // exit the intersection
}                                        }
```

Write the code for `east-bound()` and `south-bound()` 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: This is a rephrased Readers-Writers problem. The east-bound vehicles are readers and the south-bound vehicles are writers. Since an east-bound vehicle can cross the intersection as long as there are vehicles in the same direction crossing the intersection, they are the readers. On the other hand, since south-bound vehicles must cross the intersection exclusively, they are writers.

```
semaphore Mutex = 1, Intersection = 1;
int       Count;

        East-Bound                      South-Bound
        ----------                      -----------
while (1) {                         while (1) {
   Mutex.Wait();
      Count++;
      if (Count == 1)
         Intersection.Wait();          Intersection.Wait();
   Mutex.Signal();

   // cross intersection

   Mutex.Wait();
      Count--;
      if (Count == 0)
         Intersection.Signal();        Intersection.Signal();
  Mutex.Signal();
}                                   }
```

Here is a quick review of this solution. `Mutex` is a semaphore protecting the counter `Count`. `Count` counts the number of east-bound vehicles crossing the intersection, and, hence, as long as `Count` is non-zero an east-bound vehicle can proceed. However, the first vehicle tries to cross the intersection yields the intersection to itself or to a south-bound vehicle. After crossing the

intersection, if `Count` becomes zero, this "last" vehicle must allow those vehicles waiting to cross to proceed.

Lesson Learned: Understand the concept of each classical problem rather than only memorizing the code. ∎