CS3331 Concurrent Computing Exam 2 Solutions Spring 2019

1. **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, initialized to OUT_CS
   int   turn;      // global turn variable, initialized to 0 or 1

   Process i (i = 0 or 1)

   // Enter Protocol
1. repeat                                  // repeat the following
2.    flag[i] = REQUESTING;                // making a request to enter
3.    while (turn != i && flag[j] != OUT_CS)  // as long as it is not my turn and
4.       ;                                 //    the other is not out, wait
5.    flag[i] = IN_CS;                     // OK, I am in (well, maybe); but,
6. until flag[j] != IN_CS;                 //    must wait until the other is not in
7. turn = i;                               // the other is out and it is my turn!

   // critical section

   // Exit Protocol
8. turn = j;                               // yield the CS to the other
9. 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` (line 5) and then see `flag[j] != IN_CS` being true at the end of the `repeat-until` loop (line 6). Therefore, we have the following:

- **Condition for P_0 to enter its critical section**: If process P_0 is in the critical section, it had executed `flag[0] = IN_CS` followed by seeing `flag[1] != IN_CS`. That is, `flag[0] = IN_CS` **and** `flag[1] != IN_CS` are true.
- **Condition for P_1 to enter its critical section**: If process P_1 is in the critical section, it had executed `flag[1] = IN_CS` followed by seeing `flag[0] != IN_CS`. Hence, `flag[1] = IN_CS` **and** `flag[1] != IN_CS` both hold.
- **Prove by Contradiction**: If P_0 and P_1 are both in their critical sections, then (`flag[0] = IN_CS` **and** `flag[1] != IN_CS`) **AND** (`flag[1] = IN_CS` **and** `flag[0] != IN_CS`) hold at the same time. However, this means that `flag[0]` and `flag[1]` are equal to and not equal to `IN_CS` at the same time. This is absurd. Consequently, mutual exclusion hold.

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 10 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) **[15 points]** Consider the following solution to the critical section problem. The shared variables `flag[0]` and `flag[1]` are both initialized to `FALSE`.

```
   bool  flag[2] =  {FALSE, FALSE};   // global flags

   Process 0                Process 1
   ---------                ---------
1. flag[0] = TRUE;        flag[1] = TRUE;      // I am interested
2. while (flag[1]) {      while (flag[0]) {    // wait if you are interested
3.   flag[0] = FALSE;        flag[1] = FALSE;  //    then play nice to you
4.   while (flag[1])         while (flag[0])   //    wait until the other not interested
5.        ;                     ;
6.   flag[0] = TRUE;         flag[1] = TRUE;   //    OK.  try again!
7. }                      }
8.               // in critical section
9. flag[0] = FALSE;       flag[1] = FALSE;     // I am not interested
```

Show that this solution **does not** satisfy the *progress* condition. You have to construct an execution sequence (in the table format used in class) to show that *progress* is not met. *You will receive* **zero** *point if* **(1)** *your proof is vague and/or unconvincing, or* **(2)** *you did not provide a valid execution sequence.*

Answer: This is a problem similar to Attempt II. See page 11 of `06-Soft-Hardware.pdf` for what we have discussed in class. Both processes may execute in a fully synchronized way. As a result, they will loop in the outer `while` indefinitely. Because the outer `while` determines which process can enter the critical section, this fully synchronized execution will cause the finite decision criterion to fail. In other words, both processes are waiting to enter, but none of them can. The following execution sequence shows you why.

Line No.	P_0	P_1	`flag[0]`	`flag[1]`	*Comments*
1	`flag[0] = TRUE`	`flag[1] = TRUE`	`TRUE`	`TRUE`	
2	`while (flag[1])`	`while (flag[0])`	`TRUE`	`TRUE`	into outer `while`
3	`flag[0] = FALSE`	`flag[1] = FALSE`	`FALSE`	`FALSE`	
4	`while (flag[1])`	`while (flag[0])`	`FALSE`	`FALSE`	skips the inner `while`
6	`flag[0] = TRUE`	`flag[1] = TRUE`	`TRUE`	`TRUE`	
7	loops back to Line 2		`TRUE`	`TRUE`	

In this way, both processes will iterate in the outer `while` indefinitely. Therefore, the finite decision time criterion is violated and this solution does not satisfy the *Progress* condition. ∎

(c) **[10 points]*** Define the meaning of a *race condition*? Answer the question first and use execution sequences 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 manipulate 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` concurrently (condition 2) because `count` is accessed in an interleaved way. 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 possible 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++`", even using machine instructions, produces different results and hence a race condition is **incomplete**, because the two processes do not access the shared variable `count` concurrently. Note that the use of higher-level language statement interleaved execution may not reveal the key concept of "sharing" as discussed in class. Therefore, use instruction level interleaved instead.

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

2. **Semaphores**

(a) **[10 points]** Suppose a system in which there are two types of processes, type *A* processes and type *B* processes. All processes of type *A* execute the same code, and all processes of type *B* execute the same code. The code for each process type is shown below.

A Processes	*B* Processes

`Wait(X)`
`Signal(Y)`
	`Wait(Y)`
	`Wait(Y)`
	`Signal(X)`
	`Signal(Y)`

Here, `X` and `Y` are semaphores. `X` is initialized to 2, and `Y` is initialized to 0. Suppose three processes of type *A* and two processes of type *B* are brought into execution simultaneously. Answer the following two questions:

- **[7 points]** Is it possible for processes to finish in the order of *AABAB*? If so, show an execution sequence that results in this order. If not, explain why as accurate as possible.
- **[3 points]** Is it possible for processes to finish in the order *AABBA*. If so, show an execution sequence that results in this order. If not, explain why as accurate as possible.

You should provide a valid and understandable execution sequence to answer this problem. (Yes, for both parts.) Without showing a valid execution sequence, you receive 0 point.

Answer: The first execution order *AABAB* is possible, but the second one *AABBA* is impossible.

- The following is an execution sequence showing that *AABAB* is possible:

A_1	A_2	B_1	A_3	B_2	*Semaphore X*	*Semaphore Y*
					2	0
`Wait(X)`					1	0
`Signal(Y)`					1	1
	`Wait(X)`				0	1
	`Signal(Y)`				0	2
		`Wait(Y)`			0	1
		`Wait(Y)`			0	0
		`Signal(X)`			1	0
		`Signal(Y)`			1	1
			`Wait(X)`		0	1
			`Signal(Y)`		0	2
				`Wait(Y)`	0	1
				`Wait(Y)`	0	0
				`Signal(X)`	1	0
				`Signal(Y)`	1	1

This execution sequence shows clearly that processes A_1, A_2 and A_3 in group A, and processes B_1 and B_2 in group B can indeed produce the order *AABAB*.

- The sequence *AABBA* is impossible. From the above execution sequence, after *AA* semaphores X and Y have counters 0 and 2. If B follows, X and Y will be 1 and 1. Because Y's counter is 1, the next B can only pass its first `Wait(Y)` and will be blocked by the second `Wait(Y)`. Then, the second B will be blocked by its first `Wait(Y)`, and, as a result, the order of *AABBA* is impossible.

∎

(b) **[10 points]** As discussed in class, it is very typical that a thread must do something (*e.g.*, the entry protocol) before doing its core task, and do something else upon exit (*e.g.*, the exit protocol). The entry and exit protocols must perform some locking and unlocking activities in order to access some shared data items to avoid potential race conditions. However, too many locking and unlocking activities can be rather inefficient, and the *passing the baton* technique may be used to cut some redundant locking and unlocking activities.

The following shows an entry protocol and an exit protocol without the use of the passing the baton technique. Modify it to use the passing the baton technique. **Just directly edit the code below, and indicate clearly how the baton is passed. You will receive zero point if you provide no elaboration or a vague elaboration.** This is really a simple problem if you attended my lecture.

```
 1 int        Waiting = Newcomers = 0;  // counters
 2 int        i;
 3 Semaphore  Mutex = 1, Empty = 0;

   // Entry
 4 Mutex.Wait();                        // lock "Newcomers" and "Waiting"
 5 if (Newcomers == n || Waiting > 0) { // is the waiting condition met?
 6    Waiting++;                        // no. waiting is increased by 1
 7    Mutex.Signal();                   // unlock the Mutex
 8       Empty.Wait();                  // wait for releasing
 9       Mutex.Wait();                  // get the lock again if released
10          Waiting--;                  // released, waiting decreases by 1
11          Newcomers++;                // but newcomer increased by 1
12       Mutex.Signal();                // release the Mutex
13 }
14 else {                               // condition not met.  go ahead
15    Newcomers++;                      // one more newcomers
16    Mutex.Signal();                   // release the Mutex lock
17 }
   // do business
   // Exit
18 Mutex.Wait();                        // lock the Mutex
19    Newcomers--;                      // one less newcomers
20    if (Newcomers == 0) {             // if no newcomers, then ...
21       for (i = 1; i <= Waiting; i++) // release some waiting one
22          Empty.Signal();             // signal them
23    }
24 Mutex.Signal();                      // release the Mutex
```

<u>**Answer**</u>: The baton passing scheme is basically giving the lock (*i.e.*, the baton) to a released waiting thread without releasing the lock. Therefore, as discussed in class, the thread that receives the baton gets the lock (*i.e.*, the baton) without actually executing `Wait()` on that lock. In this example, a thread gets the lock of `Mutex` as the baton, and, hence, `Mutex.Wait()` (line 9) could be eliminated. Then, who is going to pass the baton to a thread waiting on `Mutex`? Look at line 18 where a thread acquires `Mutex` and then releases threads blocked by `Empty` (line 22). The released threads from `Empty.Signal()` will try to lock `Mutex` (line 9). Consequently, the thread that acquired `Mutex` on line 18, after releasing a thread blocked by `Empty.Wait()` (line 8), could pass the baton to that thread so that it can modify `Waiting` and `Newcomers` on line 10 and line 11, respectively.

In this way, the `Mutex.Wait()` on line 9 is not needed. The entry section is modified as follows:

```
 1 int        Waiting = Newcomers = 0;  // counters
 2 int        i;
 3 Semaphore  Mutex = 1, Empty = 0;

   // Entry
 4 Mutex.Wait();                        // lock "Newcomers" and "Waiting"
 5 if (Newcomers == n || Waiting > 0) { // is the waiting condition met?
 6    Waiting++;                        // no. waiting is increased by 1
 7    Mutex.Signal();                   // unlock the Mutex
 8       Empty.Wait();                  // wait for releasing
 9                                      // have the baton. no need to Wait() on Mutex
10          Waiting--;                  // have the lock for accessing Waiting and Newcomers
11          Newcomers++;                // but newcomer increased by 1
12       Mutex.Signal();                // release the Mutex
13 }
14 else {                              // condition not met.  go ahead
15    Newcomers++;                      // one more newcomers
16    Mutex.Signal();                   // release the Mutex lock
17 }
```

But, in the above code, both the `then` and the `else` have something in common: `Newcomers++` and `Mutex.Signal`. Therefore, we could move them out of the `if`:

```
 1 int        Waiting = Newcomers = 0;  // counters
 2 int        i;
 3 Semaphore  Mutex = 1, Empty = 0;

   // Entry
 4 Mutex.Wait();                        // lock "Newcomers" and "Waiting"
 5 if (Newcomers == n || Waiting > 0) { // is the waiting condition met?
 6    Waiting++;                        // no. waiting is increased by 1
 7    Mutex.Signal();                   // unlock the Mutex
 8       Empty.Wait();                  // wait for releasing
 9                                      // have the baton. no need to Wait() on Mutex
10          Waiting--;                  // have the lock for accessing Waiting and Newcomers
11 }
15 Newcomers++;                         // one more newcomers
16 Mutex.Signal();                      // release the Mutex lock
```

Next, let us look at the exit section.

```
18 Mutex.Wait();                        // lock the Mutex
19    Newcomers--;                      // one less newcomers
20    if (Newcomers == 0) {             // if no newcomers, then ...
21       for (i = 1; i <= Waiting; i++) // release some waiting one
22          Empty.Signal();             // signal them
23    }
24 Mutex.Signal();                      // release the Mutex
```

The exiting thread locks `Mutex` (line 18) to protect `Newcomers` (line 19) and `Waiting` (line 21). Here, if there is no `Newcomers` this thread will signal `Empty` and pass the baton (*i.e.*, no need to release the lock) to it. On the other hand, if there are `Newcomers` threads blocked by `Empty`, this thread simply skips the release step and proceeds to unlock `Mutex`. Therefore, a naive solution looks like the following:

```
18 Mutex.Wait();                         // lock the Mutex
19    Newcomers--;                       // one less newcomers
20    if (Newcomers == 0) {              // if no newcomers, then ...
21       for (i = 1; i <= Waiting; i++)  // release some waiting one
22          Empty.Signal();              // signal them
23    }
24a   else
24b      Mutex.Signal();                 // release the Mutex
```

The for loop (line 21–22) releases Waiting number of threads from semaphore Empty. In the original form, there is a Mutex.Wait() (line 9) to force the released threads to access Waiting and Newcomers in a mutually exclusive way. However, if the Mutex.Wait() (line 9) is removed, there would be all Waiting threads to get through, and, as a result, all of them can have access to Waiting and Newcomers at the same time, creating race conditions. Consequently, we can only release one thread from Empty at a time so that race condition will not happen. This is our current version of the exit section:

```
18 Mutex.Wait();                         // lock the Mutex
19    Newcomers--;                       // one less newcomers
20    if (Newcomers == 0)                // if no newcomers, then ...
21                                       // we can only pass the baton to ONE thread!!!
22       Empty.Signal();                 // signal them
24a   else
24b       Mutex.Signal();                // release the Mutex
```

In this version, an exiting thread finds out if there are Newcomers threads waiting on Empty, it allows a waiting thread to go (*i.e.*, passing the baton). Otherwise, the exiting thread simply releases the lock Mutex. This is what I expect from you. Is this a good solution? **No, it is not a good solution, even though it successfully performs the "pass-the-baton" technique.** The reason is simple. In the original version the exiting thread **must** signal Waiting number of threads. However, in this version every exiting thread only releases one and only one thread from Empty, which is different from the original and is a deficiency.

How do we overcome this problem so that the solution resemble the original? The answer is also easy. **Because a thread released from Empty gets the baton, why don't we allow this released thread to release the next thread from Empty and pass the baton to it?**

Let use return to the original version of the entry section as shown below:

```
   // Entry
 4 Mutex.Wait();                         // lock "Newcomers" and "Waiting"
 5 if (Newcomers == n || Waiting > 0) {  // is the waiting condition met?
 6    Waiting++;                         // no. waiting is increased by 1
 7    Mutex.Signal();                    // unlock the Mutex
 8       Empty.Wait();                   // wait for releasing
 9                                       // have the baton. no need to Wait() on Mutex
10          Waiting--;                   // have the lock for accessing Waiting and Newcomers
11          Newcomers++;                 // but newcomer increased by 1
12       Mutex.Signal();                 // release the Mutex
13 }
14 else {                                // condition not met.  go ahead
15    Newcomers++;                       // one more newcomers
16    Mutex.Signal();                    // release the Mutex lock
17 }
```

After a thread is released from Empty.Wait() (line 8), this released thread holds the baton (*i.e.*, critical section defined by Mutex) and has exclusive access to Waiting and Newcomers. Therefore, this released thread could check Waiting and release the next thread blocked by Empty and pass the baton to it. In this case, this released thread does not release the baton. On the other hand, if Waiting is 0, which means no waiting threads blocked by Empty, this thread simply releases lock Mutex and goes away. Therefore, the entry section looks like the following:

```
// Entry
Mutex.Wait();                          // lock "Newcomers" and "Waiting"
if (Newcomers == n || Waiting > 0) {   // is the waiting condition met?
   Waiting++;                          // no. waiting is increased by 1
   Mutex.Signal();                     // unlock the Mutex
      Empty.Wait();                    // wait for releasing
                                       // have the baton. not need to Wait() on Mutex
         Waiting--;                    // have the lock for accessing Waiting and Newcomers
         Newcomers++;                  // but newcomer increased by 1
         if (Waiting != 0)             // release the next thread from Empty
            Empty.Signal();            //    and pass the baton to it
         else                          // no one is waiting on Empty
            Mutex.Signal();            // release the Mutex and go away.
}
else {                                 // condition not met.  go ahead
   Newcomers++;                        // one more newcomers
   Mutex.Signal();                     // release the Mutex lock
}
```

Combining everything we learned so far together, a better solution is shown below:

```
int        Waiting = Newcomers = 0;  // counters
int        i;
Semaphore  Mutex = 1, Empty = 0;

// Entry
Mutex.Wait();                          // lock "Newcomers" and "Waiting"
if (Newcomers == n || Waiting > 0) {   // is the waiting condition met?
   Waiting++;                          // no. waiting is increased by 1
   Mutex.Signal();                     // unlock the Mutex
      Empty.Wait();                    // wait for releasing
                                       // have the baton. not need to Wait() on Mutex
         Waiting--;                    // have the lock for accessing Waiting and Newcomers
         Newcomers++;                  // but newcomer increased by 1
         if (Waiting != 0)             // release the next thread from Empty
            Empty.Signal();            //    and pass the baton to it
         else                          // no one is waiting on Empty
            Mutex.Signal();            // release the Mutex and go away.
}
else {                                 // condition not met.  go ahead
   Newcomers++;                        // one more newcomers
   Mutex.Signal();                     // release the Mutex lock
}
// do business
// Exit
Mutex.Wait();                          // lock the Mutex
   Newcomers--;                        // one less newcomers
   if (Newcomers == 0)                 // if no newcomers, then ...
                                       // we can only pass the baton to ONE thread!!!
      Empty.Signal();                  // signal them
   else
      Mutex.Signal();                  // release the Mutex
```

This solution shows that when passing the baton one has to be very careful to ensure that only one thread can be allowed to go and take the baton. The remaining threads can follow a cascading release pattern until all threads that have to be released will be released one by one so that each of the released threads can get the baton and pass it to the next released thread. ■

(c) **[10 points]** At a child care center, state regulations require that there must always be one adult present for every three children. Each adult (*resp.*, child) is simulated by a `Adult` (*resp.*, `Child`) thread. 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) **[15 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

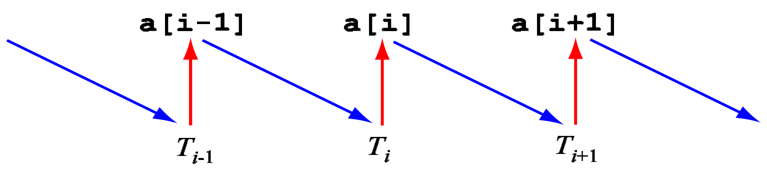
```
Thread T_i(...)
{
     // initialization: my thread ID is in a[i]
     // TID_i is my thread ID

     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
}
```

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. **Note that you do not have to rewrite the above code. What you need to do is modifying the above code by adding the needed semaphore operations. You should explain why your implementation is correct in details. A vague discussion or no discussion receives <u>zero</u> point. Moreover, maximum parallelism is always a requirement. Without observing the maximum parallelism requirement you will risk a <u>VERY</u> low score.**

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`.

<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

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

     5.             if (in == END_GAME) {
                         // game over, my thread ID is not the largest
     6.                  break;
     7.             }
     8.             else if (in == TID_i) {
                         // my TID is the largest, break
     9.                  Wait(S[i]);          // lock a[i]
    10.                       a[i] = END_GAME;  // tell everyone to break
    11.                  Signal(S[i]);
    12.                  printf("My thread ID %d is the largest\n", TID_i);
    13.                  break;
    14.             }
    15.             else if (in > TID_i) {
                         // someone's thread ID is larger than mine
    16.                  Wait(S[i]);          // lock a[i]
    17.                       a[i] = in;      // pass it to my neighbor
    18.                  Signal(S[i]);
    19.             }
    20.             else {
                        // so far, my thread ID is still the largest
                        // do nothing
    21.             }
    22.        }
               // do something else
    23. }
```

The above solution uses a local variable `in` to store the value of `a[i-1]` at the beginning of each iteration (line 3) to avoid locking `a[i-1]` for a long time. It also avoids a possible race condition when testing the value of `a[i-1]` in the three `if` statements (lines 5, 8 and 15). 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 (line 2 and line 16). 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 **incorrect** 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 a[i-1] == END_GAME) {
               // 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] = END_GAME;       // 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
          }
          Signal(S);
     }
     // do something else
}
```

(b) **[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 (line 2). Mutual exclusion among all threads (*i.e.*, searchers, inserters, and deleters) are enforced by semaphore `listProtection` (line 6) 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;

 1. while (1) {
 2.    Wait(insertProtection);        // mutual exclusion for inserter
 3.       Wait(Mutex);
 4.          Count++;
 5.          if (Count == 1)
 6.             Wait(listProtection);
 7.       Signal(Mutex);

       // do insertion

 8.       Wait(Mutex);
 9.          Count--;
10.          if (Count == 0)
11.             Signal(listProtection);
12.       Signal(Mutex);
13.    Signal(insertProtection);

       // do other thing
14. }
```

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 (line 7 and line 8).

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

    1. while (1) {
    2.   Wait(Mutex);
    3.      Count++;
    4.      if (Count == 1)
    5.         Wait(listProtection);
    6.   Signal(Mutex);

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

    9.   Wait(Mutex);
   10.      Count--;
   11.      if (Count == 0)
   12.         Signal(listProtection);
   13.   Signal(Mutex);

         // do other thing
   14. }
```

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. ■