CS3331 Concurrent Computing Solutions 2
Fall 2015

1. **Synchronization Basics**

(a) **[15 points]** The following is a solution to the critical section problem. It has two shared variables `Flag[ ]` and `turn` and a process **Scheduler** started before processes P_1 and P_2. Process **Scheduler** waits until `turn` becomes 0. Then, the `repeat-until` loop searches for a `j` such that `Flag[j]` is `TRUE`. Finally, `turn` is set to the value of `j` and loops back.

```
Boolean Flag[1..2] = { FALSE, FALSE } // note that there is no Flag[0]
int     turn = 0;
```

Process Scheduler

```
int  j;

j = 0;
repeat                      // repeat forever
   while (turn != 0)        // wait if turn is not 0
      ;                     //
   repeat                   // now turn = 0
      j = (j % 2) + 1;      //    search for a j such that
   until Flag[j];           //    Flag[j] is TRUE
   turn = j;                // set turn to j
until FALSE;                // loops back
```

Processes P_1 and P_2 are shown below. Both have very simple entry and exit sections.

Process P_1

```
Flag[1] = TRUE;     // interested
while (turn != 1)   // wait if not my turn
   ;
// Critical Section
Flag[1] = FALSE;    // no more interested
turn = 0;           // release my turn
```

Process P_2

```
Flag[2] = TRUE;     // interested
while (turn != 2)   // wait if not my turn
   ;
// Critical Section
Flag[2] = FALSE;    // no more interested
turn = 0;           // release my turn
```

Show rigorously that this solution satisfies the mutual exclusion and bounded waiting conditions. Moreover, state the *bound* first and prove the bounded waiting condition. **A vague and/or unconvincing proof receives no point**.

<u>**Answer**</u>: In this solution, P_1 and P_2 are two running processes scheduled by the **Scheduler** process. Note that `turn` is zero only if no one is in the critical section. As long as there is a process in its critical section, `turn` is non-zero and **Scheduler** executes its `while` loop. The `repeat-until` loop in **Scheduler** tries repeatedly to find a `j` such that `Flag[j]` is `TRUE`. This means that **Scheduler** tries to find a process P_j who is trying to enter its critical section. Note that if P_j is trying to enter its critical section, `Flag[j]` is set to `TRUE` in the entry section. Note also that if no process is interested in entering, **Scheduler** executes `repeat-until` until an interested process occurs. If there are entering processes, **Scheduler** will find one and set `turn` to that process. Because `turn` is non-zero if there are entering processes, when **Scheduler** loops back, its `while` loop blocks **Scheduler** from any further activity until no process is in the critical section. In this way, the "selected" process by the inner `repeat-until` loop of **Scheduler** is allowed to enter its critical section. Therefore, **Scheduler** plays the role of the thread/process scheduler trying to "schedule" a waiting process to enter its critical section.

<u>**Mutual Exclusion.**</u> From the above code, P_1 is in its critical section if P_1 has set `Flag[1]` to `TRUE` and breaks the `while` loop (*i.e.*, $\texttt{turn} = 1$). By the same reason, P_2 is in its critical section

if P_2 has set `Flag[2]` to `TRUE` and breaks the `while` loop (*i.e.*, **turn** $= 2$). Therefore, if P_1 and P_2 are both in their critical sections, `turn` must be both 1 and 2. This is impossible, and, hence, P_1 and P_2 cannot be in their critical sections at the same time. Consequently, this solution satisfies the mutual exclusion condition.

Bounded Waiting. The proof is identical to pp. 19–22 of `06-Sync-Soft-Hardware.pdf`. **The bound is 1.** In other words, a process waiting to enter its critical section only waits for no more than one turn. Suppose P_1 is entering. We know that `Flag[1]` is `TRUE` and P_1 is executing its `while` loop (*i.e.*, **turn** $\neq 1$). Meanwhile, P_2 may be in one of the following three situations: **(1)** P_2 is not interested; **(2)** P_2 is in its critical section; and **(3)** P_2 is also in its enter section. Let us prove each case separately.

- **Case (1): P_2 is not interested**. In this case, P_2 set `Flag[2]` to `FALSE` and `turn` to 0 upon exit. Process **Scheduler** sees `turn` being 0 and executes the `repeat-until` loop. Then, it will find `Flag[1]` being `TRUE` (*i.e.*, `j` $= 1$) because P_1 made it so at the very beginning when it reaches its entry section. As a result, process **Scheduler** breaks the `repeat-until` loop and sets `turn` to 1. After this, **Scheduler** goes back to the beginning and executes the `while` loop until `turn` becomes 0 again. At the same time, P_1 sees `turn` being 1 in its `while` loop and enters its cortical section. Therefore, in this case, P_1 waits for zero turn.
- **Case (2): P_2 is in its critical section**. Because P_2 is in its critical section, we know from P_2's code that `Flag[2]` and `turn` are `TRUE` and 2, respectively. Moreover, **Scheduler** busy waits in its `while` loop because `turn` is 2, and P_1 busy waits in its `while` loop because **turn** $\neq 1$. Eventually, P_2 will set `Flag[2]` to `FALSE` and `turn` to zero upon exit. This brings us back to Case **(1)** in which P_2 is not interested and P_1 is entering. Therefore, P_1 will enter its critical section and waits for zero turn.
- **Case (3): P_1 and P_2 are both in the entry sections.** If P_2 is also in its entry section, then `Flag[2]` is `TRUE`. In this case, because P_1 and P_2 are both entering, `Flag[1]` and `Flag[2]` are both `TRUE`. Note that at this point `turn` is zero because there is no process in its critical section. As a result, **Scheduler** breaks the `while` loop and executes the `repeat-until`. Depending on which `j` is selected such that `Flag[j]` is `TRUE`, either P_1 or P_2 (but not both) can enter its critical section because this solution satisfies the mutual exclusion condition. If P_1 enters, it waits for zero turn. If P_2 enters, we have Case **(2)**, which means P_2 will eventually exits so that P_1 can enter. In this case, P_1 waits for one turn.

In summary, after at most one turn, a waiting process can enter its critical section. Note that this solution works for more than two processes. The modifications are: **(a)** change array `Flag[1..2]` to `Flag[1..n]`, where `n` is the number of involved processes, **(b)** replace the `2` in **Scheduler**'s `repeat-until` loop by `n`, and **(c)** process P_i sets `Flag[i]` to `TRUE` and executes its `while` loop until `turn` becomes `i`. This n-process version satisfies all three conditions, and the bound for he bounded waiting condition is $n-1$. The same argument above applies to this n-process version. ∎

(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. 5–10 of `05-Sync-Basics.pdf`. ■

2. **Synchronization**

(a) **[10 points]** Consider the following implementation of mutual exclusion with a semaphore `X`.

```
Semaphore X = 1;

Process 1         Process 2
---------         ---------
while (1) {       while (1) {
   .....             .....
   Wait(X);          Wait(X);
   // CS             // CS
   Signal(X);        Signal(X);
   ...               .....
}                 }
```

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

Answer: The proof is similar to the one for the `TS` instruction (pp. 25–26 of `06-Sync-Soft-Hardware.pdf`). Suppose processes P_1 and P_2 are both in their critical sections. We have two cases to consider: **(1)** they enter their critical sections at the same time, and **(2)** P_1 and P_2 enter sequentially.

Case **(1)** is impossible because the `Wait()` method is atomic and hence P_1 and P_2 cannot reach the entry sections and then enter their critical sections at the same time. In other words, P_1 and P_2 must enter their critical sections sequentially.

Suppose P_1 enters first. Right after P_1 enters, the counter of semaphore `X` becomes zero. This means when P_2 reaches its entry section later it will be blocked due to the semaphore counter being zero. Therefore, case **(2)** is also impossible, and P_1 and P_2 cannot be in their critical sections at the same time. Note that this argument applies to multiple processes. ∎

(b) **[10 points]** A programmer designed a FIFO semaphore so that the waiting processes can be released in a first-in-first-out order. This FIFO semaphore has an integer counter `Counter`, a queue of semaphores, and procedures `FIFO_Wait()` and `FIFO_Signal()`.

A semaphore `Mutex` with initial value 1 is also used. `FIFO_Wait()` uses `Mutex` to lock the procedure and checks `Counter`. If `Counter` is positive, `FIFO_Wait()` decreases `Counter` by one, unlocks the procedure, and returns. If `Counter` is zero, a semaphore `X` with initial value 0 is allocated and added to the end of the queue of semaphores. Then, `FIFO_Wait()` releases the procedure, and lets the caller wait on `X`.

Procedure `FIFO_Signal()` first locks the procedure, and checks if the semaphore queue is empty. If the queue is empty, `FIFO_Signal()` increases `Counter` by one, unlocks the procedure, and returns. If there is a waiting process in the queue, the head of the queue is removed and signaled so that the *only* waiting process on that semaphore can continue. Then, this semaphore node is freed and the procedure is unlocked.

Finally, the initialization procedure `FIFO_Init()`, not shown below, sets the counter to an initial value and the queue to empty.

```
Semaphore  Mutex = 1;
int        Counter;
```

```
FIFO_Wait(...)
{
     Wait(Mutex);
     if (Counter > 0) {
          Counter--;
          Signal(Mutex);
     }
     else { /* must wait here */
          allocate a semaphore node, X=0;
          add X to the end of queue;
          Signal(Mutex);
          Wait(X);
     }
}
```

```
FIFO_Signal(...)
{
     Wait(Mutex);
     if (queue is empty) {
          Counter++;
          Signal(Mutex);
     }
     else { /* someone is waiting */
          remove the head X;
          Signal(X);
          free X;
          Signal(Mutex);
     }
}
```

Discuss the correctness of this solution. If you think it correctly implements a first-in-first-out semaphore, provide a convincing argument to justify your claim. Otherwise, discuss why it is wrong with an execution sequence.

Answer: As soon as you see a shared item being accessed by multiple threads/processes without a proper protection, it is a sign of possible race conditions. In this particular case, the allocation/free of and waiting on semaphore of `X` are trouble spots, because before a process can wait on `X`, `X` may have been freed by `FIFO_Signal()`.

Suppose process *A* calls `FIFO_Wait()` and sees `Counter` being zero. Then, this process allocates a semaphore node `X` with initial value 0, adds it to the semaphore queue, and releases the lock `Mutex`.

But, right before *A* can wait on semaphore `X`, it is switched out and *B* is switched in. Process *B* calls `FIFO_Signal()`. This *B* sees the semaphore queue being non-empty, and, as a result, removes the head semaphore node from the queue. If this is the only semaphore node, it has to be the semaphore `X` that process A is about to wait on. (Keep in mind that A was switched out before it can actually wait on this semaphore.) Then, B signals semaphore `X`. Now, what if B continues and A does not run immediately? In this case, semaphore `X` is freed before A can continue. Consequently, when A executes `Wait(X)`, semaphore `X` has already gone and A has no semaphore to wait on. The following execution sequence illustrates this execution sequence. We assume that there is no process waiting on the FIFO semaphore initially.

No.	*Process A*	*Process B*
1	calls `FIFO_Wait()`	
2	`Wait(Mutex)`	
3	sees `Counter` = 0	
4	allocates semaphore node `X`	
5	adds `X` into queue	
6	`Signal(Mutex)`	
7		calls `FIFO_Signal()`
8		`Wait(Mutex)`
9		queue is not empty
10		removes the head `X`
11		`Signal(X)`
12		free `X`
13	`Wait(X)`. Oops! where is `X`?	

You may wonder why we choose the switching point between `Signal(Mutex)` and `Wait(X)` in `FIFO_Wait()`. The reason is simple. Any context switch between `Wait(Mutex)` and `Signal(Mutex)` has no impact because `Mutex` is locked and no other process can be in its critical section protected by `Mutex`.

Some may came up with the following execution sequence, claiming that the FIFO order may be violated, where the initial value of `Count` is 1:

Process A_1	Process A_2	Process B	*Comment*
		`FIFO_Wait()`	B tries to enter its critical section
		......	B is in its critical section, `Count = 0`
`FIFO_Wait()`			A_1 tries to enter its critical section

`Wait(Mutex)`
......
`Signal(Mutex)`
			A_1 in `FIFO_Wait()`

	`FIFO_Wait()`		A_2 tries to enter its critical section

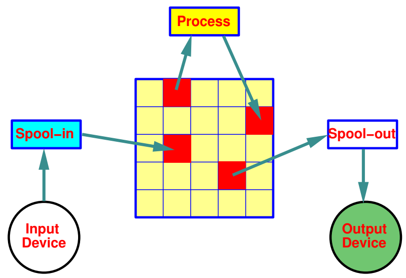
	`Wait(Mutex)`
	......
	`Signal(Mutex)`
			A_2 in `FIFO_Wait()`

	`Wait(X)`		A_2 waits
`Wait(X)`			A_1 waits
		`FIFO_Signal()`	B releases its critical section B is in `FIFO_Signal()`
		`Signal(X)`	B releases an A

Some may claim that the `Signal(X)` will cause A_2 to be released because A_2 executed `Wait(X)` before A_1 did, and, hence, FIFO order is violated. This is, of course, incorrect, because the semaphore that A_1 waits on was added to the semaphore queue before A_2 did, even though A_1

executes a wait later than A_2. ■

(c) **[15 points]** A simplified SPOOL system has three processes: *Spool-in*, *Spool-out* and *Process*. They share a spool device, say a disk. *Spool-in* reads in input from a slow input device and copies it to the spool device, *Spool-out* sends the print output from the spool device to a slow output device, and *Process* is a user program that reads in its input from and writes its output to the spool device. To be more efficient, the spool device is divided into a number of slots and each read and write operation reads and writes exactly one slot. Once a slot is read (by *Process*) or printed (by *Spool-out*) the space occupied by this slot is considered free and can be re-used.



The following are the "rules" for performing a spooling operation:

- Initially, the spool device is empty.
- As long as the spool device has an empty slot, *Spool-in* will read the input and copy it to the spool device. If all slots are used, *Spool-in* blocks until there are free slots.
- *Process* reads its input from the spool device if there are slots that have been filled with input data by *Spool-in*; otherwise, *Process* blocks until new input data become available. After reading an input, *Process* will generate some output, one slot at a time. *Process* also blocks until there are empty slots for output.
- As long as the spool device has output slots, *Spool-out* will read and send them to the output device. *Spool-out* blocks until output data become available.
- Reading from and writing into a slot is guaranteed to be mutually exclusive.

Under what condition(s) this system will have a deadlock. You should provide an execution sequence that can lead to a deadlock. Elaborate your answer; otherwise, you may receive **low** or even **no** credit.

Answer: Note the following observations:

- *Spool-out* never blocks as long as the spool device has at least one output slot.
- *Spool-in* does not block if the spool device is full of output data because *Spool-out* will eventually print some of them.
- *Process* blocks when it prints and the spool device is full of input data. *Process* does not block if the spool device is full of output data because *Spool-out* will eventually print some of them.

Therefore, the system has a deadlock only if **(1)** the spool device is full of input data and **(2)** *Process* is trying to print. This situation can happen as follows:

- *Spool-in* fills all empty slots of the spool device
- *Process* reads one slot. The spool device now has one empty slot.
- *Spool-in* fills this only empty slot fast before *Process* can puts output in.

Now, *Process* is waiting for an empty slot to write its output, *Spool-out* is waiting for *Process*'s output, and *Spool-in* is waiting for *Process* and/or *Spool-out* to empty a slot. None of these processes can continue, and we have a deadlock. The following is a possible execution sequence:

Spool-in	*Process*	*Spool-out*	*Comment*
Read input until the sppol is full			*Spool-in* fills up the spool
	Read in one slot		*Process* consumes one slot
Read one input into the emptied slot			The spool is full again
Spool-in, *Process* and *Spool-out* all block			**Deadlock!**

Note that you cannot say **(1)** *Spool-in* fills all empty slots, **(2)** *Process* has no place to write, and **(3)** a deadlock occurs. You **must** show that the scenario **can** occur. ■

3. **Problem Solving:**

(a) **[20 points]** A multithreaded program has two global arrays and a number of threads that execute concurrently. The following shows the global arrays, where `n` is a constant defined elsewhere (*e.g.*, in a `#define`):

```
int  a[n], b[n];
```

Thread T_i ($0 < i \leq n-1$) runs the following (pseudo-) code, where function `f()` takes two integer arguments and returns an integer, and function `g()` takes one integer argument and returns an integer. Functions `f()` and `g()` do not use any global variable.

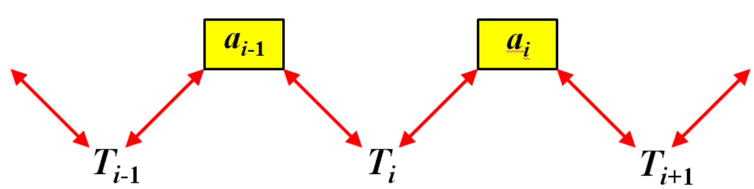
```
while (not done) {
   a[i] = f(a[i], a[i-1]);
   b[i] = g(a[i]);
}
```

More precisely, thread T_i passes the value of `a[i-1]` computed by T_{i-1} and the value of `a[i]` computed by T_i to function `f()` to compute the new value for `a[i]`, which is then passed to function `g()` to compute `b[i]`.

Declare semaphores with initial values, and add `Wait()` and `Signal()` calls to thread T_i so that it will compute the result correctly. Your implementation should not have any busy waiting, race condition, and deadlock, and should aim for **maximum parallelism**.

A convincing correctness argument is needed. Otherwise, you will receive no credit for this problem.

Answer: If you look at the code carefully, you will see that thread T_i ($1 < T_i < n-1$) shares `a[i-1]` and `a[i]` with T_{i-1} and T_{i+1}, respectively. See the diagram below. Therefore, T_i must wait until T_{i-1} finishes using `a[i-1]` before using it in the computation of `a[i] = f(a[i], a[i-1])`. Similarly, T_i must wait until T_{i+1} finishes using `a[i]` before computing a new value for `a[i]`. To this end, we need a semaphore `s[i-1]` for protecting `a[i-1]` and a semaphore `s[i]` for protecting `a[i]`. Isn't this very similar to the philosophers problem if you considered T_i as a philosopher and `a[i-1]` and `a[i]` as T_i's chopsticks? The major difference is that the philosophers are "circular" while the T_i's are "linear."



Next, we consider T_1 and T_{n-1}. T_1 uses `a[0]` and `a[1]`. Because no thread updates `a[0]`, T_1 only needs a semaphore `s[1]` to protect `a[1]`. Similarly, thread T_{n-1} uses `a[n-1]` and `a[n-2]`.

Because there is no thread using `a[n-1]` other than T_{n-1} itself, T_{n-1} only needs a semaphore `s[n-2]` to protect `a[n-2]`.

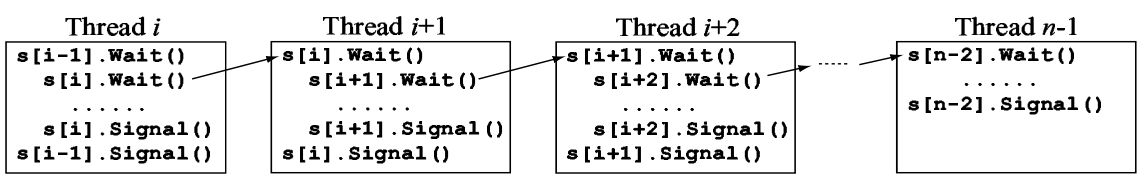
The following is a possible solution. Note that `b[i]` is not protected by any semaphore because **(1)** only T_i uses `b[i]` and **(2)** only T_i writes into `a[i]`, and once `a[i]` is correctly computed one can compute `b[i]` correctly.

```
Sem s[1..n-1] = { 1, 1, 1, ..., 1 };  // all semaphores = 1 for mutual exclusion

Thread-1:                     Thread-i:                       Thread-(n-1):
while (not done) {            while (not done) {              while (not done) {
   s[1].Wait();                  s[i-1].Wait();                  s[n-2].Wait();
      a[1] = f(a[0], a[1]);      s[i].Wait();                       a[n-1] = f(a[n-2], a[n-1]);
   s[1].Signal();                   a[i] = f(a[i-1], a[i]);      s[n-2].Signal();
   b[1] = g(a[1]);               s[i].Signal();                  b[n-1] = g(a[n-1]);
}                                s[i-1].Signal();             }
                                 b[i] = g(a[i]);
                              }
```

Obviously, there is no busy waiting. The above solution has no race conditions either because each shared data item (*i.e.*, `a[i-1]` and `a[i]` for thread T_i) is protected by mutual exclusion.

This solution is deadlock free. What we have to show is that no semaphore would block threads indefinitely. Consider thread T_i, which may be blocked on semaphores `s[i-1]` or `s[i]`. If T_i is blocked on semaphore `s[i-1]`, this means thread T_{i-1} is now in its critical section. Thread T_{i-1} will eventually exit its critical section and execute `s[i-1].Signal()`. At this moment, thread T_i could continue although T_{i-1} may come back fast enough and execute `s[i-1].Wait()` successfully again. Whatever the case is, T_i has the chance to continue, and, the worst case is starvation rather than a deadlock.



If thread T_i is blocked on semaphore `s[i]`, this means thread T_{i+1} executed its `s[i].Wait()`, followed by `s[i+1].Wait()`. See the figure above. Thread T_{i+1} may or may not be blocked by `s[i+1].Wait()`. If thread T_{i+1} is not blocked by `s[i+1].Wait()`, it will eventually signal it, allowing T_i to continue. If thread T_{i+1} is blocked by `s[i+1].Wait()`, it is because thread T_{i+2} is blocked by `s[i+2].Wait()`. With the same reasoning, the worst case is that this chain of "waiting" could continue until thread T_{n-1}. However, if T_{n-1} can pass `s[n-2].Wait()`, which causes thread T_{n-2} to wait, it will signal `s[n-2]` later, and, as a result, thread T_{n-2} has a chance to run. Thread T_{n-2} will signal `s[n-3]`, allowing thread T_{n-3} to run. This backward chain of "signal" will eventually return to thread T_{i+1}, which will signal `s[i]`. Therefore, thread T_i always has a chance to pass semaphore `s[i].Wait()` and enters its critical section. This means, again, thread T_i does not involve in a deadlock. At worst, it only has starvation.

<u>Incorrect Solution 1:</u> There are some incorrect solutions and the following is one of them.

```
Sem s = 1;

Thread i:
while (not done) {
   s.Wait();
      a[i] = f(a[i], a[i-1]);
      b[i] = g(a[i]);
   s.Signal();
}
```

This "solution" uses semaphore `s`, and thread T_i acquires `s` before updating `a[i]` and `b[i]`. The problem is that there is virtually no concurrency at all. In other words, because at any moment there is at most one thread can be in its critical section, at any moment there is only one T_i in execution. As a result, the original requirement of multithreading is destroyed by the use of single semaphores.

Some even move `s.Wait()` and `s.Signal()` outside of the `while` loop. In this case, you are guaranteed that the thread that is lucky enough to enter the critical section will run forever and no other threads can have a chance to execute.

Incorrect Solution 2: Because T_i uses `a[i-1]` to update `a[i]` and T_1 uses `a[0]`, which is not changed, the following "solution" seems correct. The problem is that this solution forces the execution to be T_1, T_2, ..., T_{n-1} and then the system deadlocks because no one allows T_1 to run.

```
Sem s[n] = { 1, 0, ..., 0};  // allows T1 to start first
                             // and all remaining Ti's to block

Thread 1:                          Thread i:
while (not done) {                 while (not done) {
   s[1].Wait();                       s[i].Wait();
      a[1] = f(a[1], a[0]);              a[i] = f(a[i], a[i-1]);
      b[1] = g(a[1]);                    b[i] = g(a[i]);
   s[2].Signal();                     s[i+1].Signal();
}                                  }
```

Incorrect Solution 3: One might suggest allowing T_{n-1} to signal T_1 as shown below. The situation is improved just a little; but, the execution would still be fixed (*i.e.*, T_1, T_2, ..., T_{n-1}, T_1, T_2, ..., T_{n-1}, ...) and at any time only one thread can be in execution (*i.e.*, no concurrency).

```
Sem s[n] = { 1, 0, ..., 0};  // allows T1 to start first
                             // and all remaining Ti's to block

Thread 1:                    Thread i:                      Thread n-1:
while (not done) {           while (not done) {             while (not done) {
   s[1].Wait();                 s[i].Wait();                   s[n-1].Wait();
      a[1] = f(a[1], a[0]);        a[i] = f(a[i], a[i-1]);        a[n-1] = f(a[n-1], a[n-2]);
      b[1] = g(a[1]);              b[i] = g(a[i]);                b[n-1] = g(a[n-1]);
   s[2].Signal();               s[i+1].Signal();               s[1].Signal();
}                            }                              }
```

■

(b) **[20 points]** Design a class `Group` in C++, a constructor, and method `Group_wait()` that fulfill the following specification:

- The constructor `Group(int n)` takes a positive integer argument `n`, and initializes a private `int` variable in class `Group` to have the value of `n`. The value of `n` will not change in the execution of the program.

- Method `Group_wait(void)` takes no argument. A thread that calls `Group_wait()` blocks if the number of threads being blocked is less than `n`-1. Then, the `n`-*th* calling thread releases all `n`-1 blocked threads and all `n` threads continue. Note that the system has more than `n` threads. For example, suppose `n` is initialized to 3. The first two threads that call `Group_wait()` block. When the third thread calls `Group_wait()`, the two blocked threads are released, and all three threads continue. Note that your solution **cannot** assume `n` to be 3. Otherwise, you will receive **zero** point.

Use semaphores only to implement class `Group` *and method* `Group_wait()`. Otherwise, you will receive **zero** point. Use `Sem` for semaphore declaration and initialization (*e.g.*, "`Sem S = 0;`"), `Wait(S)` on a semaphore `S`, and `Signal(S)` to signal semaphore `S`.

Your implementation should not have any busy waiting, race condition and deadlock. You should explain why your implementation is correct in detail. A vague discussion or no discussion receives no credit.

Answer: This is a simple variation of the reader-writer problem, because the last thread must activate/do something. Compare the task of the *n*-th thread with what the last reader should do, and you should be able to see the similarity, although the situation of this problem is different.

It is obvious that we need a counter `count` to count the number of waiting threads. Initially, `count` should be 0. Based on the specification, we need two semaphores: `Mutex` for protecting the counter `count`, and `WaitingList` for blocking threads.

When a thread calls `Group_wait()`, it locks the counter, and checks to see if it is the `n`-th one. If it is not, the thread releases the lock and waits on semaphore `WaitingList`. This portion is trivial. Note that the order of "releasing the lock" and "waiting on semaphore `WaitingList`" is important. Otherwise, a deadlock could occur. (Why?)

If the thread is the `n`-th one, it must release all waiting threads that were blocked on semaphore `WaitingList`. Because we know there are exactly $\mathtt{n}-1$ waiting threads, executing $\mathtt{n}-1$ signals to semaphore `WaitingList` will release them all. Then, the `n`-th thread resets the counter and releases the lock.

Based on this idea, the following is the `Group` class:

```
class Group {
   private:
      int       Total;           // total number of threads in a batch
      int       count;           // counter that counts blocked threads
      Semaphore WaitingList(0);  // the waiting list
      Semaphore Mutex(1);        // mutex lock that protects the counter
   public:
      Group(int n) { Total = n; count = 0 };  // constructor
      Group_wait();                           // the wait method
};

Group::Group_wait()
{
   int  i;

   Mutex.Wait();                  // lock the counter
   if (count == Total-1) {        // if I am the n-th one
      for (i=0; i<Total-1; i++)   // release all waiting threads
           WaitingList.Signal();
      count = 0;                  // reset counter
      Mutex.Signal();             // release the lock
   }                              // I am done
   else {                         // otherwise, I am not the last one
      count++;                    // one more waiting threads
      Mutex.Signal();             // release the mutex lock
      WaitingList.Wait();         // block myself
   }
}
```

The protection of the counter `count` must start at the very beginning and extend to the very end so that the blocked `n`-1 threads can be released in a "single" batch. In other words, when the execution flow enters the "then" part of the `if`, all blocked `n`-1 threads are released as a single group. Otherwise, we may have the following issues. **First**, we may release a newcomer rather than the threads that were waiting in the group prior to the release. **Second**, threads just released may come back (*i.e.*, fast-runner) and be released again. In this case, the same thread is released twice and one of the originally blocked threads is not released. Hence, this violates the specification that the blocked $n-1$ threads must be released.

If the `Group_wait()` method is rewritten as follows to "increase efficiency" by doing `Mutex.Signal()` before releasing the `n`−1 blocked threads, we will have problems:

```
Group::Group_wait()               // incorrect version
{
   int  i;

   Mutex.Wait();                  // lock the counter
   if (count == Total-1) {        // I am the n-th one
      count = 0;                  // reset counter
      Mutex.Signal();             // release the lock
      for (i=1; i<=Total-1; i++)  // release all waiting threads
           WaitingList.Signal();
   }                              // I am done
   else {                         // otherwise, I am not the last one
      count++;                    // one more waiting threads
      Mutex.Signal();             // release the mutex lock
      WaitingList.Wait();         // block myself
   }
}
```

With this version, a thread just released from semaphore `WaitingList` may come back and call `Group_wait()` again. This thread can immediately change the value of `count` and wait on `WaitingList` again while the original `n`-th thread is still in the process of releasing the blocked `n`-1 threads. Because we cannot make any assumption about the order used for releasing threads, it is possible that a fast running thread is released again and one of the originally blocked thread will be blocked and released the next run. Or, it may be blocked forever!

The following is a similar solution. In this solution the `n`-th thread signals semaphore `WaitingList` `n` times so that it will release itself at the end. This "solution" does have the *fast-runner* problem. Suppose `n` is 2. The first thread blocks as usual. When the second comes, it signals `WaitingList` twice, releases the `Mutex` lock, and is switched out by a context switch. Because the first thread was released by one of the two signals, it may come back, go through all steps, and wait on semaphore `WaitingList` faster than the second thread does. Because `WaitingList` was signaled twice, this returning thread is not blocked and can pass through. As a result, the same thread is released twice and the releasing thread is blocked. Of course, this is terribly wrong!

```
Group::Group_wait()                // incorrect version
{
   int  i;

   Mutex.Wait();                   // lock the counter
   if (count == Total-1) {         // I am the n-th one
      count = 0;                   // reset counter
      for (i=1; i<=Total; i++)     // release all waiting threads
           WaitingList.Signal();   //    including myself
   }
   else                            // otherwise, I am not the last one
      count++;                     // one more waiting threads
   Mutex.Signal();                 // release the mutex lock
   WaitingList.Wait();             // block myself
}
```

■