CS3331 Concurrent Computing Exam 2 Solutions
Fall 2019

1. **Synchronization**

(a) **[15 points]** Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1, where `status[ ]` is a Boolean array of two elements and `turn` is an integer variable. Furthermore, there are three constants indicating the status of a process, where `COMPETING`, `IN_CS` and `OUT_CS` mean competing to enter the critical section, in the critical section, and out of the critical section. Note that `status[ ]` and `turn` are global variables shared by both processes.

```
int  status[2];   // status of a process initialized to OUT_CS
int  turn;        // initialized to either 0 or 1

         Process 0                          Process 1
         =========                          =========
 1 status[0] = COMPETING;             status[1] = COMPETING;
 2 while (status[1] == COMPETING) {   while (status[0] == COMPETING) {
 3    status[0] = OUT_CS;                status[1] = OUT_CS;
 4    repeat until (turn == 0);          repeat until (turn == 0 || turn == 1);
 5    turn = 0;                          turn = 1;
 6    status[0] = COMPETING;             status[1] = COMPETING;
 7 }                                  }
                         // critical section
 8 status[0] = OUT_CS;                status[1] = OUT_CS;
 9 turn      = 0;                     turn = 0;
```

Use **proof-by-contradiction** to show rigorously that this solution satisfies the mutual exclusion condition. *You will receive* **zero** *point if* **(1)** *you prove by example,* **(2)** *your proof is vague and/or unconvincing, or* **(3)** *you do not prove by contradiction.*

Answer: Checking the short code you should be able to see the following:

- When a process reaches its `while` loop, whether it is the first time or loops back, this process always has its own `status[ ]` set to `COMPETING` (Line 1 and Line 6).
- Then, if the other process is **NOT** `COMPETING`, this process breaks its own `while` loop and enters its critical section.
- So far, we know that P_0 is in its critical section `status[0]` is `COMPETING` **AND** `status[1]` is not `COMPETING`. By the same reason, if P_1 is in its critical section `status[1]` is `COMPETING` **AND** `status[0]` is not `COMPETING`. This is good enough to derive a contradiction for proving mutual exclusion.
- However, there is more to say. Let us look at the variable `turn`. Suppose P_1 is not `COMPETING`. In this case P_0 enters its critical section immediately and `turn` plays no role. On the other hand, if P_0 enters its `while` loop, P_0 sets `turn` to 0 before reaching the end of its `while` loop. Therefore, as long as P_0 enters its `while` loop, we have `status[0]` and `status[1]` being `COMPETING` and not `COMPETING`, respectively, and `turn` being 0. By the same reason, as long as P_1 enters its `while` loop, P_1 can enter its critical section, we have `status[0]` and `status[1]` being not `COMPETING` and `COMPETING`, respectively, and `turn` being 1.

Because we do not know whether a process enters its critical section without getting into its `while` loop, and because we do not know which process modifies `turn` first, the role of `turn` and `status[ ]` being `OUT_CS` are not useful and should not be used. Thus, we can only rely on `status[ ]` and we have:

- P_0 is in its critical section if and only if `status[0]` is `COMPETING` and `status[1]` is not `COMPETING`.
- P_1 is in its critical section if and only if `status[1]` is `COMPETING` and `status[0]` is not `COMPETING`.

If P_0 and P_1 are both in their critical section at the same time, then `status[0]` (and `status[1]`) must be `COMPETING` and not `COMPETING` at the same time, which is absurd. Thus, we have a contradiction and the mutual exclusion condition is met. ∎

(b) **[15 points]** Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1, where `status[ ]` is a Boolean array of two elements and `turn` is an integer variable. Furthermore,

there are three constants indicating the status of a process, where `COMPETING`, `IN_CS` and `OUT_CS` mean competing to enter the critical section, in the critical section, and out of the critical section. Note that `status[ ]` and `turn` are global variables shared by both processes.

```
int  status[2];   // status of a process
int  turn;        // initialized to either 0 or 1

         Process 0                        Process 1
         =========                        =========
 1 status[0] = COMPETING;              status[1] = COMPETING;
 2 do {                                do {
 3    while (turn != 0) {                 while (turn != 1) {
 4       status[0] = OUT_CS;                 status[1] = OUT_CS;
 5       if (status[turn] == OUT_CS)         if (status[turn] == OUT_CS)
 6          turn = 0;                           turn = 1;
 7    }                                   }
 8    status[0] = IN_CS;                  status[1] = IN_CS;
 9 } while (status[1] == IN_CS);       } while (status[0] == IN_CS);
                      // critical section
10 status[0] = OUT_CS;                 status[1] = OUT_CS;
```

Use a clear and convincing execution sequence to show that this solution does not satisfy the **bounded waiting** condition. A convincing argument **is** required. *You will receive* **zero** *point if* **(1)** *you do not use a valid execution sequence, or* **(2)** *your execution sequence is vague and/or unconvincing.* **Hint:** the value of `turn` plays a significant role.

Answer: As indicated in the hint, the role of `turn` is significant. Note that `turn` is only set to 0 and 1 by P_0 and P_1 (line 6), respectively. Now, if P_0 comes back fast before P_1 can set `turn` to 1, P_0 can enter immediately. Upon exiting its critical section, P_0 sets `turn` to 0 because of the `while` (line 3). Variable `turn` is 0 either because of its initial value or because being set to 0 (line 6). Therefore, if P_0 (*resp.*, P_1) exits its critical section, `turn` is 0 (*resp.*, 1). This is the major cause of the failure of the bounded waiting condition. In other words, this solution is in favor of the just-exited process.

	P_0	P_1	`turn`	`status[0]`	`status[1]`	*Comment*
1			0			
2	`s[0]=C`	`s[1]=C`	0	`C`	`C`	Entering
3	`while`	`while`	0	`C`	`C`	P_0 breaks `while`
4	`s[0]=IN`		0	`IN`	`C`	P_0 about to enter
5		`s[1]=OUT`	0	`IN`	`OUT`	P_0 about to enter
6	**P_0 enters its critical section**					
7	`s[0]=OUT`	`if`	0	`OUT`	`OUT`	P_0 enters CS
8	**P_0 comes back**					
9	`s[0]=C`	`while` loops back	0	`C`	`OUT`	P_0 entering
10		`if`	0	`C`	`OUT`	`if` is false
11	`while`	`while`	0	`C`	`OUT`	P_1 loops back
12	`s[0]=IN`		0	`IN`	`OUT`	P_0 about to enter
13	**P_0 enters its critical section**					

Suppose that `turn` is 0, meaning P_0 may just exit its critical section. See the execution sequence above. To save space, we use `s[ ]`, `C`, `IN` and `OUT` to indicate `status[ ]`, `COMPETING`, `IN_CS` and `OUT_CS`, respectively. From the above observation, if P_0 is fast enough so that every time before P_1 can test the value of `status[turn]` P_0 sets `status[0]` to either `COMPETING` or `IN_CS`, P_0 enters. In this execution sequence, P_0 simply repeats the action between line 6 and line 12 in the above execution sequence. As a result, P_1 will starve, which means P_0 simply keeps entering the critical section and P_1 does not have any chance. Of course, the bounded waiting condition fails. ∎

(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**

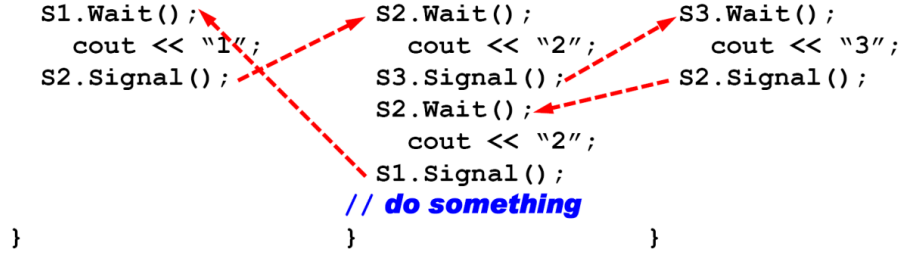
(a) **[10 points]** Consider the following code:

```
Thread 1              Thread 2              Thread 3
========              ========              ========
while (1) {           while (1) {           while (1)
   // do something      // do something        // do something
   cout << "1";         cout << "2";          count << "3";
   // do something      // do something        // do something
}                     }                     }
```

There are three threads. The first, second, and third thread prints `1`, `2` and `3`, respectively. Declare and add semaphores to the above code so that the output of these three concurrently running threads is `1`, `2`, `3`, `2`, `1`, `2`, `3`, `2`, `1`, ...

Answer: This is a very simple problem. It was discussed in class to some degree and was an exercise on a weekly reading list. The following is a possible solution:

```
Semaphore  S1 = 1, S2 = 0, S3 = 0;

   Thread 1                Thread 2                Thread 3
while (1) {             while (1) {             while (1) {
                        // do something
  S1.Wait();              S2.Wait();              S3.Wait();
    cout << "1";            cout << "2";            cout << "3";
  S2.Signal();            S3.Signal();            S2.Signal();
                          S2.Wait();
                            cout << "2";
                          S1.Signal();
                        // do something
}                       }                       }
```



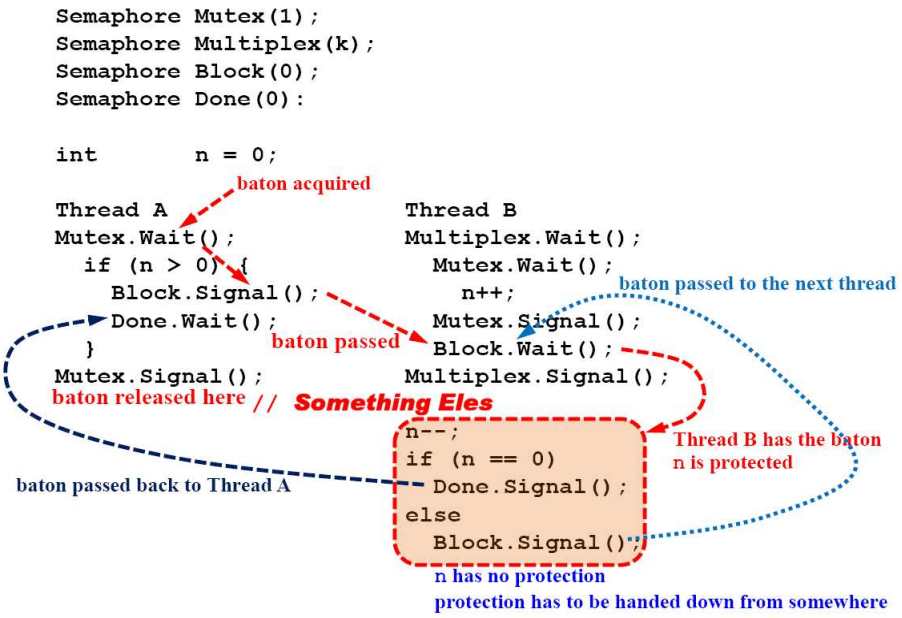
In the above code, going from thread 1 to thread 2 and going from thread 2 to thread 3 are exactly the same as the "`1 2 1 2 ...`" pattern discussed in class. What is really needed is a going-back pattern from thread 3 to thread 2 and then from thread 2 to thread 1. Because thread 3 has printed "`3`", we need to add a section of code to bridge between thread 3 and thread 1. That is it! ■

(b) **[10 points]** The following code shows a single thread `Thread A` and a `Thread B` group. All threads in `Thread B` Group execute the same code. There are four semaphores, `Mutex`, `Multiplex`, `Block` and `Done`. There is also a global variable `n` shared by `Thread A` and all threads and Group *B*.

```
Semaphore    Mutex(1);
Semaphore    Multiplex(k);  // a being a large positive integer
Semaphore    Block(0);
Semaphore    Done(0);

int          n = 0;

Thread A                    Thread B Group
========                    ==============
Mutex.Wait();               Multiplex.Wait();
  if (n > 0) {                Mutex.Wait();
    Block.Signal();             n++;
    Done.Wait();              Mutex.Signal();
  }                           Block.Wait();
Mutex.Signal();             Multiplex.Signal();

// Something Else           // Something Else

                            n--;
                            if (n == 0)
                              Done.Signal();
                            else
                              Block.Signal();

                            // Something Else
```

Study the above code and identify **ALL** possible pass-the-baton patterns. You should indicate the following as clear as possible: **(1)** for each baton, where is it acquired? **(2)** for each baton, where is it passed? Also indicate its receiver of this baton; and **(3)** for each baton, where is it released? Draw arrows directly on the code like we did in class and on slides to indicate the path of baton passing. A convincing elaboration is **REQUIRED**. You will receive **zero** point if you do not provide a convincing elaboration/

Answer: The following shows possible pass-the-baton patterns:



Note that in the second half of Thread *B*, shared variable `n` is not protected. This means that this protection has to be handed down from somewhere. How could the execution reach `n--`? It is easy to see. A thread in Thread *B* can reach `n--` only if it is released from `Block.Wait()`, because Thread *A* calls `Block.Signal()`. However, when Thread *A* reaches `Block.Signal()`, it has the baton (*i.e.*, `Mutex`). As a result, Thread *A* passes the baton (*i.e.*, `Mutex`) to a thread in Thread *B*. Therefore, the released thread from `Block.Wait()` has the baton `Mutex` and can have exclusive access to `n`.

After the released thread in Thread *B* executing `n--`, it may either hand the baton to the next thread released from `Block.Wait` if `n` is greater than 0. Or, the released thread returns the baton to Thread *A* by releasing Thread *A* which was blocked by `Done.Wait()`. Finally, Thread *A* releases the baton `Mutex`. ∎

(c) **[10 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 all have their left chopsticks and are waiting for their right ones in order to eat. Otherwise, there is at least one philosopher is eating, and the system does not have a deadlock. Depending on whether 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 and wait for their right 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.

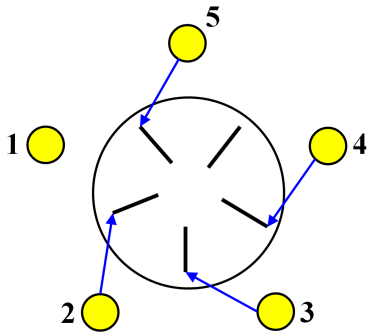


Figure (a)

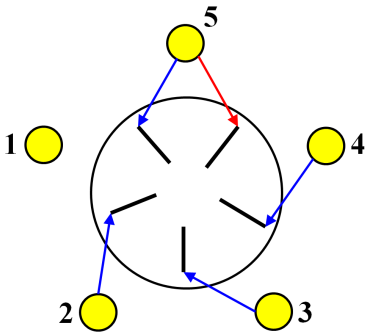


Figure (b)

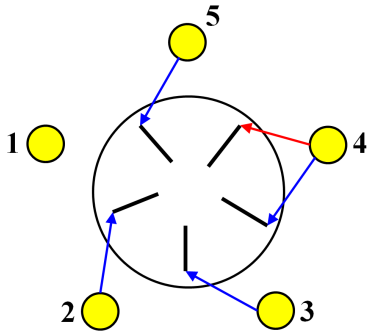


Figure (c)

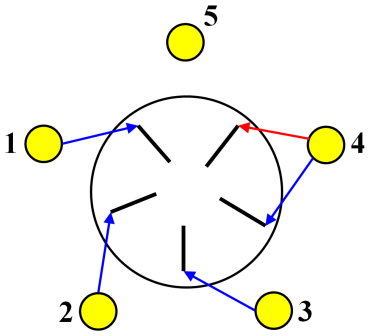


Figure (d)

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

3. **Problem Solving:**

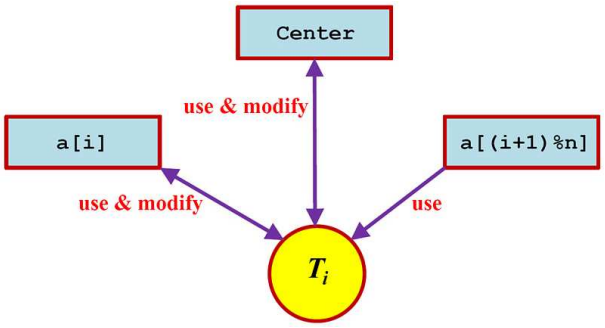
(a) **[15 points]** Let $T_0, T_1, \ldots, T_{n-1}$ be n threads, and let `a[ ]` be a global `int` array of n elements. Moreover, thread T_i only uses `a[i]` and `a[(i+1)%n]` for $0 \leq i \leq n-1$. Thus, array `a[ ]` is "circular." Additionally, `Center` is a global variable shared by all threads.

The code below processes `Center` and `a[ ]` **without** synchronization,

```
int  Center = ...       // some initial value
int  a[n]   = { .... }  // some initial values

Thread i
========

1. while (1) {
2.    // other irrelevant computation
3.    a[i]   = f(a[i], a[(i+1)%n]); // f() does not use a[ ] and Center
4.    Center = a[i] + Center;
5.    // other irrelevant computation
6. }
```

Declare and insert semaphores to the above code so that the indicated task can be performed correctly. You may use as many semaphores as you want. However, you should avoid **busy waiting**, **race conditions**, and **deadlocks**. Moreover, your modification should aim for **maximum concurrency**. A code with close to minimum concurrency receives **zero** point.

Answer: The code template shows three important facts: **(1)** variable `Center` is being shared by all threads, **(2)** array element `a[(i+1)%n]` is being used by thread T_i and $T_{(i+1)\%n}$, and **(3)** array element `a[i]` is **modified** by thread T_i. See the diagram below.



Therefore, array `a[ ]` resembles the chopsticks in the Dining Philosophers problem, and each array element (*i.e.*, chopstick) has to be protected by a semaphore. Thread i (*i.e.*, philosopher i) uses `a[i]` and `a[(i+1)% n]`. Finally, another semaphore is needed to protect `Center`. Let these semaphores be `S_Center` and `S_a[ ]`. We have the following:

```
int  Center = ...       // some initial value
int  a[n]   = { .... }  // some initial values

Semaphore S_Center = 1;
Semaphore S_a[n]   = { 1, 1, ....., 1 };
```

It is important to note that thread T_i only reads `a[(i+1)% n]`. It is not worth to lock `a[(i+1)% n]` while in function `f( )`, which could be time consuming. As a result, we could lock `a[(i+1)% n]` and copy its value to a local variable `Local`. In this way, `a[(i+1)% n]` is free for thread $T_{(i+1)\%n}$ to use (actually modify). Because only T_i modifies `a[i]` after calling `f( )`, the execution of `f(a[i],Local)` does not affect the value of `a[i]` and no protection for `f( )` is needed. When saving the result of `f( )` to `a[i]`, protection is needed. The final version looks like the following:

```
int  Center = ...       // some initial value
int  a[n]   = { .... }  // some initial values

Semaphore S_Center = 1;
Semaphore S_a[n]   = { 1, 1, ....., 1 };

Thread i
========
1.  int Local, fx;

2.  while (1) {
3.     // other irrelevant computation
4.     S_a[(i+1)%n]).Wait();         // Lock a[(i+1)%n]
5.        Local = a[(i+1)%n];       // Make a copy
6.     S_a[(i+1)%n]).Signal();       // Release a[(i+1)%n]
7.     fx = f(a[i], Local);          // f() does not use a[ ] and Center
8.     S_a[i].Wait();                // Lock a[i]
9.        a[i] = fx;                 // Update a[i]
10.    S_a[i].Signal();              // Unlock a[i]
11.    S_Center.Wait();              // Lock Center
12.       Center = fx + Center;      // Update Center
13.    S_Center.Signal();            // Release Center
14.    // other irrelevant computation
15. }
```

The following version is not very efficient. Because `Center` is being used by all threads, at any time there is one and only one thread can be executing Line 6-7 below. Consequently, this serializes the execution of all threads, and the execution of these n threads becomes sequential. This means that there is no concurrency at all.

```
int  Center = ...       // some initial value
int  a[n]   = { .... }  // some initial values

Semaphore S_Center = 1;
Semaphore S_a[n]   = { 1, 1, ....., 1 };

Thread i
========
1. while (1) {
2.    // other irrelevant computation
3.    S_a[(i+1)%n]).Wait();        // Lock a[(i+1)%n]
4.       S_a[i].Wait();            // Lock a[i]
5.          S_Center.Wait();       // Lock Center;
6.             a[i] = f(a[i], a[(i+1)%n]);
7.             Center = a[i] + Center;
8.          S_Center.Signal();     // Release Center;
9.       S_a[i].Signal             // Release a[i]
10.   S_a[(i+1)%n].Signal          // Release a[(i+1)%n]
11.   // other irrelevant computation
12. }
```

The following version is even worse:

```
int  Center = ...       // some initial value
int  a[n]   = { .... }  // some initial values

Semaphore S = 1;

Thread i
========
1. while (1) {
2.    // other irrelevant computation
3.    S.Wait();
4.       a[i] = f(a[i], a[(i+1)%n]);
5.       Center = a[i] + Center;
6.    S.Signal();
7.    // other irrelevant computation
8. }
```

In terms of concurrency, this version is the same as the above one because there is one and only one thread can be executing in Line 4-5. ■

(b) **[15 points]** A unisex bathroom is shared by men and women. A man or a woman may be using the room, waiting to use the room, or doing something else. They work, use the bathroom and come back to work. The rule of using the bathroom is very simple: *there must never be a man and a woman in the room at the same time; however, people with the same gender can use the room at the same time*.

Man Thread

```
void Man(void)
{
   while (1) {
      // working
      // use the bathroom
}
```

Woman Thread

```
void Woman(void)
{
   while (1) {
      // working
      // use the bathroom
}
```

Declare semaphores and other variables with initial values, and add `Wait()` and `Signal()` calls to the threads so that the man threads and woman threads will run properly and meet the requirement. 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: This is a simple variation of the reader-priority readers-writers problem. More precisely, we allow the "writers" to write simultaneously, the "readers" and "writers" cannot do their work at the same time. Therefore, the writers have the same structures as the readers. We need to maintain two counters, one for the males `MaleCounter` and the other for the females `FemaleCounter`. Of course, we need two Mutexes `MaleMutex` and `FemaleMutex` for mutual exclusion. In addition, there is a semaphore `BathRoom` to block the males (*resp.*, females) if the room is being used by the females (*reap.*, males). Note that the male thread and female thread are symmetric.

```
int       MaleCounter = 0, FemaleCounter = 0;   // male and female counters
Semaphore MaleMutex = 1, FemaleMutex = 1;       // male and female counters
Semaphore BathRoom = 1;                         // the bathroom is empty initially

Male Thread                     Female Thread

while (1) {                     while(1) {
   // working                       // working

   MaleMutex.Wait();                FemaleMutex().Wait();        // update counter
      MaleCounter++;                   FemaleCounter--;
      if (MaleCounter == 1)            if (FemaleCounter == 1)   // if I am the first
         BathRoom.Wait();                 BathRoom.Wait();       //    yield to other
   MaleMutex.Signal();              FemaleMutex.Signal();

   // use the bathroom             // use the bathroom

   MaleMutex.Wait();                FemaleMutex.Wait();         // update counter
      MaleCounter--;                   FemaleCounter--;
      if (MaleCounter == 0)            if (FemaleCounter == 0)  // if I am the last one
         BathRoom.Signal();               BathRoom.Signal();    //    let the other group know
   MaleMutex.Signal();              FemaleMutex.Signal();
}                               }
```

Refer to the class note for the solution to the reader-priority version of the readers-writers problem for the details. ∎