

## CS4411 Intro. to Operating Systems Exam 1 Solutions Fall 2005

### 1. Basic Concepts

- (a) [8 points] Explain *interrupts* and *traps*, and provide a detailed account of the procedure that an operating system handles an interrupt.

**Answer:** An *interrupt* is an event that requires the attention of the operating system. These events include the completion of an I/O, a key press, the alarm clock going off, division by zero, accessing a memory area that does not belong to the running program, and so on. A *trap* is an interrupt generated by software.

When an interrupt occurs, the following steps will take place to handle the interrupt:

- The executing program is suspended and the control is transferred to the operating system. Mode switch may be needed.
- A general routine in the operating system examines the received interrupt and calls the interrupt-specific handler.
- After the interrupt is served, a context switch transfers the control to a suspended process. Of course, mode switch may be needed.

See p. 7 of our textbook and class notes. ■

- (b) [8 points] What is the *supervisor* or *kernel* mode? What is the *user* mode. What are the differences? Why are they needed?

**Answer:** Modern CPUs have two execution modes, the *kernel mode* and *user mode*. In addition to general instructions, *privileged instructions* can only be used in the kernel mode. These privileged instructions help the CPU access sensitive information (*e.g.*, clear the cache) and carry out vital operations (*e.g.*, I/O). In the *user mode*, only general instructions can be executed. If a privileged instruction is executed in the user mode, the hardware does not execute the instruction, but rather treats the instruction as an illegal one and traps to the operating system.

An operating system usually runs in the kernel mode so that it can have access to all hardware components and no user can affect the execution of the OS. This is for protection purpose. Note that not all components of an OS would run in the kernel mode.

This was covered in class. Refer to **pages 18** of our text and class notes details. ■

- (c) [8 points] Define the meaning of *mechanism* and *policy* in the separation of mechanism and policy principle.

**Answer:** Mechanisms determine how to do something, and policies determine what will be done. They are usually separated for flexibility.

See p. 56 of text and class notes details. ■

- (d) [6 points] Under a Unix/Linux shell, what do  $A < B$ ,  $A > B$ ,  $A | B$  and  $A \& B$  mean, where  $A$  and  $B$  are two programs or files.

**Answer:** Here are the answers:

- $A < B$ : sets the `stdin` (*i.e.*, standard input) of program  $A$  to file  $B$ . Thus, program  $A$  takes its input from file  $B$ .
- $A > B$ : sets the `stdout` (*i.e.*, standard output) of program  $A$  to file  $B$ . Thus, program  $A$  send its output to file  $B$ .
- $A | B$ : programs  $A$  and  $B$  are run concurrently, and the standard input of program  $B$  is identified to the standard output of  $A$ . Thus, program  $B$  reads its standard input from program  $A$ 's standard output.

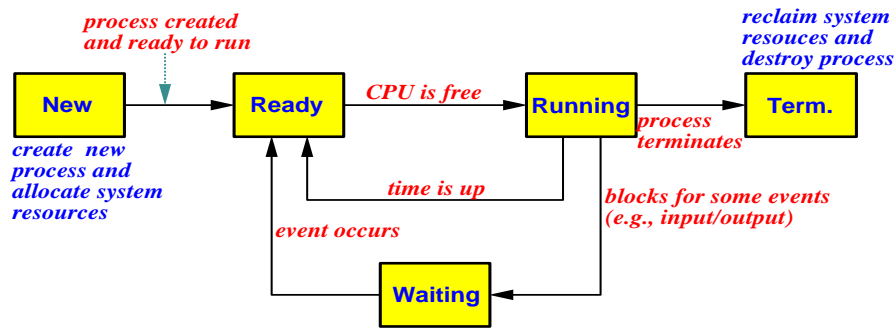
- A & B: programs A and B are run concurrently with A is the background and B in the foreground.

This is program in the second weekly reading list. ■

2. Fundamentals of Processes and Threads

- (a) [15 points] Draw the state diagram of a process from its creation to termination, including all transitions, and briefly elaborate each state and each transition.

**Answer:** The following state diagram is taken from my overhead which was covered in class. Fill in the elaboration for each state and transition by yourself.



See p. 83 of our text and class notes. ■

- (b) [10 points] What is a *context*? Provide a detailed description of *all* activities of a *context switch*.

**Answer:** A process needs some system resources to run successfully, which include process ID, registers, memory areas (for instructions, local and global variables, stack, heap and so on), various tables (*i.e.*, process table), and a program counter to indicate the next instruction to be executed. They constitute the *environment* or *context* of a process. The steps of switching process *A* to process *B* are as follows:

- Suspend *A*'s execution
- Transfer the control to the CPU scheduler. A CPU mode switch may be needed.
- Save *A*'s context to its PCB and other tables.
- Load *B*'s context to register, etc.
- Resume *B*'s execution of the instruction at *B*'s program counter. A CPU mode switch may be needed.

See p. 85 and p. 89 of our text and class notes. ■

- (c) [10 points] What is *thread cancelation*? Define the term and discuss the two commonly used versions and their differences.

**Answer:** *Thread cancelation* means terminating a thread before it completes. There are two commonly used cancelation schemes, *asynchronous* and *deferred*.

*Asynchronous cancelation* means the target thread terminates immediately. *Deferred cancelation* means the target thread can periodically check if it should terminate, allowing the target thread an opportunity to terminate itself in an orderly fashion.

See p. 139 of our text and class notes. ■

## 3. Synchronization

- (a) [10 points] Define the meaning of *race condition*? Answer the question first and use an execution sequence to illustrate your answer. **You will receive no credit if only an example is provided without an elaboration.**

**Answer:** A *race condition* is a situation in which more than one processes or threads are executing and accessing a shared data item *concurrently*, and the result depends on the order of execution. The following is a very 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;

Thread_1(...)      Thread_2(...)
{                  {
    // do something    // do something
    count++;          count--;
}                  }

```

The following execution sequence shows a race condition. There are two threads running concurrently (condition 1). Both threads 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 table below shows the result being 9; however, if the two **SAVE** instructions are switched (*i.e.*, *B*'s runs first and *A*'s second), the result would be 11. Since all three conditions of a race conditions are met, we have a race condition.

Thread_1	Thread_2	Comment
do something	do something	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++`” would produce different results and hence a race condition is incorrect, because the threads do not access the shared variable `count` at the same time (*i.e.*, Condition 2).

See p. 193 of our text and class notes. ■

- (b) [5 points] Programming Assignment #1 has three steps: (1) thread  $T_i$  ( $0 \leq i < n$ ) initializes array element `w[i]` to 1; (2) threads  $T_{i,j}$  writes a 0 into `w[i]` (*resp.*, `w[j]`) if  $x_i < x_j$  (*resp.*,  $x_i > x_j$ ) holds, where  $x_i$ 's are input values; and (3) thread  $S_i$  prints  $x_i$  if `w[i]` is 1. Of these three steps, which step or steps could have potential race conditions? **You should provide a clear answer with a convincing argument. Otherwise, you will receive no credit.**

**Answer:** There is no race condition in this program.

- **Step 1:** Since each thread only accesses its own memory (*i.e.*,  $T_i$  uses `w[i]`), Condition 2 does not hold.
- **Step 2:** The value to be written into `w[i]` is 0. If `w[0]` receives a 0,  $x_i$  is not the maximum. Hence, the order of execution will not change this fact as the `w[]` entry corresponding to the maximum element never receives anything. As a result, no race condition will occur.
- **Step 3:** Since this step only contains the testing of `w[i]`, no race condition will occur. Keep in mind that Programming Assignment #1 has *distinct* integers as the input. This means exactly one thread sends information to the standard output.

In summary, there is no race condition in Programming Assignment #1. ■

- (c) [10 points] The methods `Wait()` and `Signal()` must be atomic to ensure a correct implementation of mutual exclusion. Use an execution sequence to show that if `Wait()` is not atomic then mutual exclusion cannot be maintained. **You must use an execution sequence as we did many times in class to present your answer. Otherwise, you risk low or no credit.**

**Answer:** If `Wait()` is not atomic, its execution may be switched in the middle. If this happens, mutual exclusion will not be maintained. The following is a possible execution sequence, where `Count = 1` is the counter variable of the involved semaphore.

<i>Process A</i>	<i>Process B</i>	<i>Count</i>	<i>Comment</i>
		1	Initial value
LOAD Count		1	A executes Count-- of Wait()
SUB #1		1	
	LOAD Count	1	B executes Count-- of Wait()
	SUB #1	1	
	SAVE Count	0	B finishes Count--
SAVE Count		0	A finishes Count--
if (Count < 0)		0	It is false from A
	if (Count < 0)	0	It is false from B
Both A and B enter the critical section			

This problem was assigned in class. ■

- (d) [10 points] Consider the solution to the mutual exclusion problem for two processes  $P_0$  and  $P_1$ , where `flag[2]` is a Boolean array of two elements and `turn` is an integer variable with an initial value 0 or 1. Both are global to processes  $P_0$  and  $P_1$ . Show, with a step-by-step execution sequence table, that mutual exclusion is implemented incorrectly. **You will risk low or very low grade if you do not use an execution sequence table.**

```

bool flag[2]; // global flags
int turn; // global turn variable

Process i (i = 0 or 1)

flag[i] = TRUE; // I am interested
while (turn != i) { // while it is not my turn
    while (flag[j]) // while you are interested
        ; // do nothing: busy waiting
    turn = i; // because your are not interested, it is my turn
}
// critical section
flag[i] = FALSE // I am done and not interested
    
```

**Answer:** Suppose `turn` is initially 0. Consider the following execution sequence:

$P_0$	$P_1$	flag[0]	flag[1]	turn
		0	0	0
enters its critical section		1	0	0
	flag[1] = TRUE	1	1	0
	executes the outer while	1	1	0
	executes the inner while	1	1	0
exits its critical section		1	1	0
flag[0] = FALSE		0	1	0
	exits the inner while	0	1	0
flag[0] = TRUE		1	1	0
enters its critical section		1	1	0
	turn = 1	1	1	1
	enters its critical section	1	1	1

Therefore, both processes are in their critical sections and mutual exclusion is violated.

This is a problem in weekly reading list #4. ■

#### 4. [25 points] Problem Solving: I

- (a) 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 proper initial values, and add `Wait()` and `Signal()` calls to thread  $T_i$  to make sure it will compute the result correctly. The syntax is unimportant. For example, you can declare and initialize a semaphore  $S$  with “`Sem S = 1`” and use `Wait(S)` and `Signal(S)` for semaphore wait and semaphore signal. Your implementation should not have any busy waiting, race condition, and deadlock.

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

**Answer:** If you look at the code, you will see that thread  $T_i$ ,  $1 < T_i < n-1$ , uses  $a[i-1]$  and  $a[i]$ , and the latter is used by  $T_{i+1}$ . Therefore,  $T_i$  must wait until  $T_{i-1}$  completes its update to  $a[i-1]$  before using it in the computation of  $a[i] = f(a[i], a[i-1])$ .  $T_i$  must also wait until  $T_{i+1}$  finishes using  $a[i]$  before computing a new value for  $a[i]$ . Otherwise, race condition can occur. Therefore, we need a semaphore  $s[i-1]$  for protecting  $a[i-1]$  and a semaphore  $s[i]$  for protecting  $a[i]$ .

Next, we consider  $T_1$  and  $T_{n-1}$ .  $T_1$  uses  $a[0]$  and  $a[1]$ . However, since 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  $a[n-2]$ . Since there is no thread using  $a[n-1]$  other than  $T_{i-1}$ ,  $T_{n-1}$  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 only  $T_i$  uses  $b[i]$  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 in the above solution. It has no race condition either because each shared data item (*i.e.*,  $a[i-1]$  and  $a[i]$  for thread  $T_i$ ) is protected by mutual exclusion. Is deadlock possible? This solution is deadlock free.

We first show that if there is a deadlock, every  $T_i$  must be blocked on its first semaphore wait (*i.e.*,  $s[i-1].Wait()$ ). If  $T_i$  is blocked on the first wait, it is blocked on semaphore  $s[i-1]$ . If  $T_{i-1}$  is blocked on the second wait, the semaphore is also  $s[i-1]$ . Thus, both  $T_i$  and  $T_{i-1}$  are waiting on semaphore  $s[i-1]$ . However, since the initial value of  $s[i-1]$  is 1, and will be signaled by  $T_i$  and  $T_{i-1}$  after its use, either  $T_i$  or  $T_{i-1}$ , but not both, can continue. Therefore, if the system has a deadlock, every thread must be blocked on the first wait. However, this is also impossible. If  $T_i$  waits for  $T_{i-1}$ ,  $T_{i-1}$  waits for  $T_{i-2}$ , ...,  $T_3$  waits for  $T_2$ , and  $T_2$  waits for  $T_1$ , since  $T_1$  does not have to wait for  $a[0]$  to be released,  $T_1$  can continue and this program is deadlock free.

**Incorrect Solution 1:** 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 threading at all. In other words, since 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 single semaphore.

**Incorrect Solution 2:** Since  $T_i$  uses  $a[i-1]$  to update  $a[i]$  and  $T_1$  uses  $a[0]$  that is not changed, the following “solution” seems correct. The problem is that this solution forces the execution to be  $T_1, T_2, \dots, T_{n-1}$  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.*,  $T_1, T_2, \dots, T_{n-1}, T_1, T_2, \dots, T_{n-1}$ ) and at any time only one thread can be in execution (*i.e.*, no threading).

```
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) [25 points] **Problem Solving: II**

There are two groups of threads  $A$  and  $B$  with unspecified number of threads in each group. Two threads from group  $A$  and one thread from group  $B$  are required to establish a *rendezvous* in order to perform a specific task. Thus, each thread has the following execution pattern:

```
thread-in-A(.....)      thread-in-B(.....)
{                        {
  while (1) {           while (1) {
    // does something   // does something
    A-rendezvous(..);   B-rendezvous(..);
    // does something else // does something else
  }
}                        }
```

When a thread in group  $A$  (*resp.*,  $B$ ) reaches the rendezvous point, it calls function `A-rendezvous()` (*resp.*, `B-rendezvous()`). This function blocks the calling thread until a rendezvous of two- $A$ -one- $B$  can be made. Once such a rendezvous occurs, two threads in  $A$  and one thread in  $B$  return from `A-rendezvous()` and `B-rendezvous()`, respectively.

Use semaphores to write functions `A-rendezvous()` and `B-rendezvous()`. The syntax is unimportant. For example, you may declare and initialize a semaphore  $S$  with “Sem  $S = 1$ ” and use `Wait(S)` and `Signal(S)` for semaphore wait and semaphore signal. Your implementation should not have any busy waiting, race condition, and deadlock.

**Answer:** The following is a possible solution. This solution allows threads in group  $B$  to take control. This will make program logic easier because we do not have to worry about the number of  $A$  threads waiting for a rendezvous. Semaphore `A-enter` has an initial value of 2 so that only two threads in group  $A$  can pass `A-enter` without blocking. To prevent threads in group  $A$  running so fast that could come back for another rendezvous before the current rendezvous completes, we need a semaphore `A-exit` to block the two threads in group  $A$  that are in the process of forming a rendezvous. Semaphore `B-mutex` establishes a critical section so that at any time only one thread in group  $B$  can be in the process of forming a rendezvous. In this way, once a rendezvous starts, there is only one thread in group  $B$  involves in the rendezvous. Semaphore `B-enter` blocks the thread in group  $B$  that is in the process of a rendezvous. As a result, each thread in group  $A$  that passes semaphore `A-enter` should inform a thread in  $B$  through semaphore `B-enter`.

```

Semaphore A-enter = 2;      // only allow 2 A's to make a rendezvous
Semaphore A-exit  = 0;      // block the 2 A's so that they won't
                           // come back for another rendezvous too soon

Semaphore B-enter = 0;      // let B to wait for 2 A's
Semaphore B-mutex = 1;      // only allow 1 B to make a rendezvous

A-rendezvous(..)          // rendezvous point for threads in group A
{
    Wait(A-enter);         // A arrives. Only 2 A's can pass through
    Signal(B-enter);       // let B know an A is available
    Wait(A-exit);          // wait for a rendezvous
}

B-rendezvous(..)          // rendezvous point for threads in group B
{
    Wait(B-mutex);         // an exclusive rendezvous
    Wait(B-enter);         // wait for the 1st A
    Wait(B-enter);         // wait for the 2nd A
    Signal(A-exit);        // both are there, release the 1st A
    Signal(A-exit);        // release the 2nd A
    Signal(A-enter);       // rendezvous done and release the lock
    Signal(A-enter);       // so two more A's can go for their rendezvous
    Signal(B-mutex);       // release the mutual exclusion lock
}

```

Consider function `A-rendezvous()`. The caller is blocked by semaphore `A-enter` if there have been two threads in group *A* passing `A-enter`. Each of these two threads signals semaphore `B-enter` so that a thread in group *B* can complete this rendezvous. Then, the threads in group *A* are blocked on semaphore `A-exit` waiting for signal (from *B*) to indicate that a rendezvous completes. Once a thread in group *B* completes its rendezvous, it releases the two threads in group *A* blocked on semaphore `A-exit`. If there are waiting threads on a semaphore, a signal will release one thread from that semaphore, and both the released and the signaling threads will continue. By the time, the two `Signal(A-enter)` calls execute, the two (and only two) waiting threads will be released and a successful rendezvous is made.

Now consider function `B-rendezvous()`. A thread in group *B* must perform a rendezvous in a mutual exclusive way. Once it is the only thread in group *B* involving in a rendezvous, it waits on semaphore `B-enter` for two signals from threads in group *A*. After this thread finishes the two `Wait(B-enter)`'s, we know that there are two threads in group *A* waiting on semaphore `A-exit`, and no other threads in group *A* can join this rendezvous. Then, this thread in group *B* releases both threads in group *A* that are blocked on semaphore `A-exit`. Finally, this *B* thread signals semaphore `A-enter` allowing the next two threads to form a rendezvous. Note that the `Signal(A-exit)` should not be in function `A-rendezvous()` because a faster *A* may re-enter causing a “confused” rendezvous. See below.

In this solution, semaphore `A-enter` is a count-down lock, semaphore `B-mutex` is a simple lock, and semaphores `A-exit` and `B-enter` are for notification purpose. Here are some notes:

- The `Signal(A-enter)` call is in `A-rendezvous()` rather than in `B-rendezvous()`:

```

A-rendezvous(..)          B-rendezvous(..)
{                          {
    Wait(A-enter);         Wait(B-mutex);
    Signal(B-enter);       Wait(B-enter);
    Wait(A-exit);         Wait(B-enter);
    Signal(A-enter);       Signal(A-exit);
                          Signal(A-exit);
                          Signal(B-mutex);
}                          }

```

The following execution sequence shows that it is possible a fast  $A_1$  may have an incorrect

rendezvous of  $A_1$ ,  $A_2$  and  $B$ :

$A_1$	$A_2$	$B$	<i>Comment</i>
Wait(A-enter)			$A_1$ enters for a rendezvous
Signal(B-enter)			$A_1$ tells $B$ it is there
Wait(A-exit)			$A_1$ waits for $B$
	Wait(A-enter)		$A_2$ enters for a rendezvous
	Signal(B-enter)		$A_2$ tells $B$ it is there
	Wait(A-exit)		$A_2$ waits for $B$
		Wait(B-mutex)	$B$ arrives
		Wait(B-enter)	$B$ picks up an $A$
		Wait(B-enter)	$B$ picks up the 2nd $A$
		Signal(A-exit)	$B$ releases 1st $A$ , say $A_1$
Signal(A-enter)			$A_1$ continues
Wait(A-enter)			$A_1$ comes back!
Signal(B-enter)			
Wait(A-exit)			$A_1$ blocks and waits for $B$
		Signal(A-exit)	$B$ could release $A_1$ again!
$A_1$ continues			$A_1$ , $A_2$ and $B$ have a rendezvous!

- Semaphore **B-mutex** that enforces mutual exclusion among threads in group  $B$  cannot be eliminated. Otherwise, a deadlock may occur. Suppose we have only four threads  $A_1$  and  $A_2$  in group  $A$  and  $B_1$  and  $B_2$  in group  $B$ . The following execution sequence shows a deadlock.

$A_1$	$A_2$	$B_1$	$B_2$	<i>Notes</i>
Wait(A-enter)	Wait(A-enter)			$A_1$ and $A_2$ both pass
Signal(B-enter)	Signal(B-enter)			$A_1$ and $A_2$ inform $B$ about their arrival
		1st Wait(B-enter)	1st Wait(B-enter)	$B_1$ and $B_2$ both pass the 1st Wait
		2nd Wait(B-enter)	2nd Wait(B-enter)	$B_1$ and $B_2$ are both blocked by the 2nd Wait
Wait(A-exit)	Wait(A-exit)			$A_1$ and $A_2$ block on A-exit
$A_1$ , $A_2$ , $B_1$ and $B_2$ are all blocked				

- Semaphore **A-exit** is required. Otherwise, the same thread may come back and signal **B-enter**. Should this happen, a thread in group  $B$  will form a rendezvous with the same thread in group  $A$ , which is incorrect. The following is the version without the use of semaphore **A-exit** and a possible execution sequence.

```

Semaphore A-enter = 2;    // only allow 2 A's to make a rendezvous
Semaphore B-enter = 0;    // let B to wait for 2 A's
Semaphore B-mutex = 1;   // only allow 1 B to make a rendezvous

A-rendezvous(..)        // rendezvous point for threads in group A
{
    Wait(A-enter);       // A arrives. Only 2 A's can pass through
    Signal(B-enter);     // let B knows an A is available
    Signal(A-enter);     // rendezvous done and release the lock
}

B-rendezvous(..)        // rendezvous point for threads in group B
{
    Wait(B-mutex);       // an exclusive rendezvous
    Wait(B-enter);       // wait for the 1st A
    Wait(B-enter);       // wait for the 2nd A
    Signal(B-mutex);     // release the mutual exclusion lock
}
    
```

$A_1$	$B_1$	A-enter	B-enter	Notes
		2	0	Initial values
Wait(A-enter)		1		$A_1$ arrives
Signal(B-enter)			1	Tells $B_1$ about $A_1$ 's arrival
Signal(A-enter)		2		$A_1$ returns
$A_1$ loops back				
Wait(A-enter)		1		$A_1$ arrives
Signal(B-enter)			2	Tells $B_1$ about $A_1$ 's arrival
Signal(A-enter)		2		$A_1$ exits
	Wait(B-enter)		1	$B_1$ has the 1st $A_1$
	Wait(B-enter)		0	$B_1$ has the 2nd $A_1$
$A_1, A_1$ and $B_1$ have a rendezvous!				

- **Food for Thought:** Is the following correct?

```

int count = 0;
Semaphore A-mutex = 1; // locks A
Semaphore B-mutex = 1; // locks B
Semaphore A-ready = 0; // blocks the 1st A
Semaphore Mutex = 1; // mutex for count
Semaphore Rendezvous = 0; // rendezvous made

A-rendezvous(..)
{
    Wait(A-mutex); // only 1 A can be let in
    Wait(Mutex); // lock the counter
    count++; // one more A
    if (count == 2) // if 2 A are in
        Signal(A-ready); // tells B
    else // otherwise
        Signal(A-mutex); // let the 2nd A in
    Signal(Mutex); // release the lock
    Wait(Rendezvous); // wait to be released by B
    Wait(Mutex); // lock count
    count--; // decrease count
    if (count == 0) // am I the last 1?
        Signal(A-mutex); // yes, let the next 2A's in
    Signal(Mutex); // release count
}

B-rendezvous(..)
{
    Wait(B-mutex); // same as A
    Wait(A-ready); // wait for 2 A's
    Signal(Rendezvous); // release 1st A
    Signal(Rendezvous); // release 2nd A
    Signal(B-mutex);
}
    
```