CS4411 Intro. to Operating Systems Exam 1 Solutions
Fall 2009

1. **Basic Concepts**

   (a) **[10 points]** What is the *supervisor* or *kernel* mode? What is the *user* mode? What are the differences? Why are they needed? **Note that there are four questions.**

   Answer: Modern CPUs have two execution modes, the *kernel mode* and *user mode*, indicated by a *mode bit*. The kernel mode can use all general instructions as well as *privileged instructions*. Privileged instructions help the CPU access sensitive information (*e.g.*, clear the cache) and carry out vital operations (*e.g.*, I/O). On the other hand, only general instructions are available in the *user mode*. If a privileged instruction is executed in the user mode, the CPU 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.

   This was covered in class. Refer to **p. 3 and p. 19** of our text for more details. ∎

   (b) **[10 points]** There are three types of schedulers. What are they and what are their assigned tasks?

   Answer: There are three types of schedulers: long-term, short-term and medium-term schedulers. The *long-term* scheduler, or job scheduler, selects jobs from the job pool, loads them into memory, and converts them to processes for execution. The *short-term* scheduler, or CPU scheduler, selects processes from the ready queue, and allocates the CPU to them. The *medium-term* scheduler removes (*i.e.*, swaps out) processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, a process can be reintroduced into memory and resume its execution from where it left off (*i.e.*, swapped in). This scheme is also referred to as *swapping*.

   See **p. 108** of our text for the details. ∎

   (c) **[10 points]** Explain the Unix operators `<`, `>`, `|` and `&`.

   Answer: Operators `<` and `>` are input and output redirection operators, respectively. "`A < B`" (*resp.*. "`A > B`") sets the `stdin` (*resp.*, `stdout`) of program `A` to file `B`. In "`A & B`" programs `A` and `B` are run concurrently; however, `A` runs in background and `B` in foreground. In "`A | B`", program `A`'s `stdout` is "connected" to program `B`'s `stdin`. In this way, program `A`'s output is "piped" to program `B`'s input. ∎

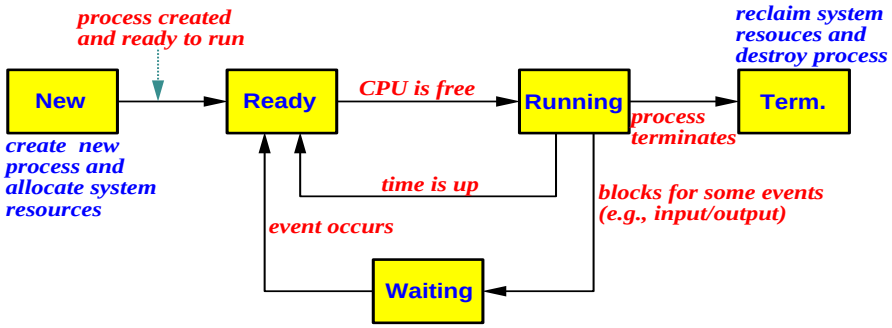
2. **Fundamentals of Processes and Threads**

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

   Answer: A process needs some system resources to run successfully. These system resources include process ID, registers, memory areas (for instructions, local and global variables, stack 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.

   This was covered in class. ∎



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

Answer: The following state diagram is taken from my class note and was discussed in class. Fill in the elaboration for each state and transition by yourself.

See **p. 103** of our text and class notes. ∎

3. **Synchronization**

(a) **[10 points]** Define the meaning of a *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 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;

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

The following execution sequence shows a race condition. Two threads run 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 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.

`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++`" 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) **[10 points]** A good solution to the critical section problem must satisfy three conditions: *mutual exclusion*, *progress* and *bounded waiting*. Explain the meaning of the *progress* condition. Does starvation violate the progress condition? **Note that there are two problems. You should provide a clear answer with a convincing argument. Otherwise, you will receive no credit.**

Answer: If no process is executing in its critical section and some processes are waiting to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next (*i.e.*, only those wanted to enter can join the competition and none of the other processes can have any influence), and this selection cannot be postponed indefinitely.

The progress condition only guarantees the decision of selecting one process to enter a critical section will not be postponed indefinitely. It does not mean a waiting process will enter its critical section eventually. Therefore, starvation is not a violation of the progress condition. Instead, starvation violates the bounded waiting condition.

See **p. 194** of our textbook. ∎

(c) **[10 points]** The semaphore 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 to present your answer as we did in class. Otherwise, you risk low or zero point.**

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.

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

Note that this question asks you to demonstrate a violation of mutual exclusion. Consequently, you receive low grade if your demonstration is not a violation of mutual exclusion.

This problem was assigned as an exercise in class. ∎

(d) **[15 points]** We cannot assume any particular order when a process (or thread) is released from a semaphore. A programmer wishes to overcome this problem by implementing a FIFO semaphore so that the waiting processes will be released in a first-in-first-out order. The following is his attempt. In this programmer's mind, a FIFO semaphore consists of an integer counter that is used to simulate a semaphore's counter, and a semaphore queue which is used to enforce the first-in-first-out order. The two corresponding procedures are `FIFO_Wait(..)` and `FIFO_Signal()`.

To guarantee atomic execution of these two procedures, a semaphore `Mutex` is used with initial value 1. `FIFO_Wait()` uses `Mutex` to lock the procedure and checks the semaphore counter `Counter`. If this counter is greater than zero, it is decreased by one. Then, `FIFO_Wait()` releases the procedure and we are all set. If `Counter` is zero, then the caller is queued. To this end, a semaphore `X` with initial value 0 is allocated, and added to the end of a semaphore queue. Then, `FIFO_Wait()` releases the procedure, and lets the caller wait on `X`.

On the other hand, `FIFO_Signal()` first locks the procedure, and checks if the semaphore queue is empty. If the queue is empty, the counter is increased by one, unlocks the procedure, and returns.

However, 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, it allocates a semaphore node `X`, 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`?	

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 uses to wait was added to the semaphore queue before A_2 did, even though A_1 executes a wait later than A_2. ∎

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

   (a) **[25 points]** Design a class `Barrier` in C++, a constructor, and method `Barrier_wait()` that fulfill the following specification:

   - The constructor `Barrier(int n)` takes a positive integer argument `n`, and initializes a private `int` variable in class `Barrier` to have the value of `n`.
   - Method `Barrier_wait(void)` takes no argument. A thread that calls `Barrier_wait()` blocks if the number of threads being blocked is less than `n`-1, where `n` is the initialization value and will not change in the execution of the program. 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. Suppose `n` is initialized to 3. The first two threads that call `Barrier_wait()` block. When the third thread calls `Barrier_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 `Barrier` *and method* `Barrier_wait()`. Without observing this rule, you will receive **zero** point. 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`.

   You should explain why your implementation is correct in some details. A vague discussion or no discussion receives no credit.

   Answer: This is a simple variation of the readers-writers 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.

   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 `Barrier_wait()`, it locks the counter, and checks to see if it is the `n`-th one. If it is not the `n`-th one, 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 will occur. (Why?)

If the thread is the `n`-th one, it must release all waiting threads that were blocked on semaphore `WaitingList`. Since we know there are exactly $n-1$ waiting threads, executing $n-1$ 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 `Barrier` class:

```
class Barrier {
   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:
      Barrier(int n) { Total = n; count = 0 };  // constructor
      Barrier_wait();                    // the wait method
};

Barrier::Barrier_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
   }
}
```

Note that 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 problems. **First**, we may release a newcomer rather than the threads that were waiting in the barrier prior to the release. **Second**, threads just released may come back 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 `Barrier_wait()` method is rewritten as the following to "increase efficiency," we will have problems:

```
Barrier::Barrier_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 `Barrier_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. Since 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! ∎

(b) **[25 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. 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 ($0 < i \leq n-1$) is as follows. Thread T_i receives T_{i-1}'s information from `a[i-1]`. If this information 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 passes it to T_{i+1} with `a[i]`. In this way, thread ID's are circulated and smaller ones are eliminated. Finally, if a thread sees the "received" information is 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
}
```

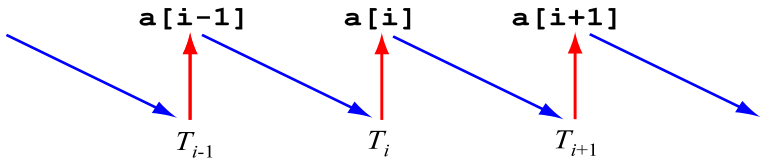
Obviously, race conditions are everywhere. Declare and add semaphores to the above code section so that the indicated task can be performed properly. You may use as many semaphores as you want. However, thread T_i can only share its resource, semaphores included, with its left neighbor T_{i-1} and right neighbor T_{i+1}. To make this problem a bit less complex, you may ignore thread T_0.

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`.

You should explain why your implementation is correct in some details. A vague discussion or no discussion receives no credit.

Answer: This is a very easy problem and a simple extension to the ping-pong notification example discussed in class.

Since thread T_i takes information out of `a[i-1]` and stores new information into `a[i]`, it is clear that `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 access `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

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

          if (in == END_GAME) {
               // game over, my thread ID is not the largest
               break;
          }
          else if (in == TID_i) {
               // my TID is the largest, break
               Wait(S[i]);            // lock a[i]
                    a[i] = END_GAME;  // tell everyone to break
               Signal(S[i]);
               printf("My thread ID %d is the largest\n", TID_i);
               break;
          }
          else if (in > TID_i) {
               // someone's thread ID is larger than mine
               Wait(S[i]);            // lock a[i]
                    a[i] = in;        // pass it to my neighbor
               Signal(S[i]);
          }
          else {
               // so far, my thread ID is still the largest
               // do nothing
          }
     }
     // do something else
}
```

The above solution uses a local variable `in` to store the value of `a[i-1]` at the beginning of each iteration to avoid locking `a[i-1]` for a long time. In this way, after retrieve 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. 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 (in == END_GAME) {
               // game over, my thread ID is not the largest
               break;
          }
          else if (in == 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 (in > TID_i) {
               // someone's thread ID is larger than mine
               a[i] = in;        // pass it to my neighbor
          }
          else {
               // so far, my thread ID is still the largest
               // do nothing
          }
          Signal(S);
     }
     // do something else
}
```