CS4411 Intro. to Operating Systems Exam 1
Fall 2009

150 points – 9 pages

Name: _____

- Most of the following questions only require very short answers. Usually a few sentences would be sufficient. Please write to the point. If I don't understand what you are saying, I believe, in most cases, you don't understand the subject.
- To minimize confusion in grading, please write **readable** answers. Otherwise, it is very likely I will interpret your unreadable handwriting in my way.
- *Justify your answer with a convincing argument*. If there is no justification when it is needed, you will receive no point for that question even though you have provided a correct answer. *I consider a good and correct justification more important than a right answer. Thus, if you provide a very vague answer without a convincing argument to show your answer being correct, you will likely receive a low grade.*
- Do those problems you know how to do first. Otherwise, you may not be able to complete this exam on time.

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.**

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

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

2. **Fundamentals of Processes and Threads**

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

(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**.

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.**

(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.**

(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.**

(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.

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.

   Use this and next page for your answer

(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.

Use this blank page for your answer

Grade Report

Problem		*Possible*	*You Received*
1	a	10	
	b	10	
	c	10	
2	a	10	
	b	15	
3	a	10	
	b	10	
	c	10	
	d	15	
4	a	25	
	b	25	
Total		150	