Name:

Operating Systems
Fall 2015
Test 2
November 9, 2015

Closed books, notes, cell phones, PDAs, iPods, laptops, etc. No headphones, please. You do not need to use a calculator.

You have 75 minutes to solve 6 problems, with an optional 7th problem for extra credit. As with any exam, you should read through the questions first and start with those that you are most comfortable with.

Partial credit will be offered only for meaningful progress towards solving the problems.

Please read and sign below if you agree with the following statement:

In recognition of and in the spirit of the academic code of honor, I certify that I will neither give nor receive unpermitted aid on this exam.

Signature:_____

1	/20
2	/10
3	/10
4	/10
5	/10
6	/15
Total	**/75**
7 (extra)	/15

1. [20 points] Short Answer Questions:

a. The term busy waiting refers to a technique in which a process can do nothing until it gets permission to enter its critical section but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance.

b. ~~*True or*~~ *False:* Race condition is a situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

c. A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution is a race condition.

d. Atomicity is when the sequence of instruction is guaranteed to execute as a group, or not execute at all.

e. ~~*True or*~~ *False:* A process that is waiting for access to a critical section does not consume processor time when mutual exclusion is implemented with *CompareAndSwap*-based locks.

f. *True* ~~*or False*~~*?* It is possible for one process to lock the mutex and for another process to unlock it.

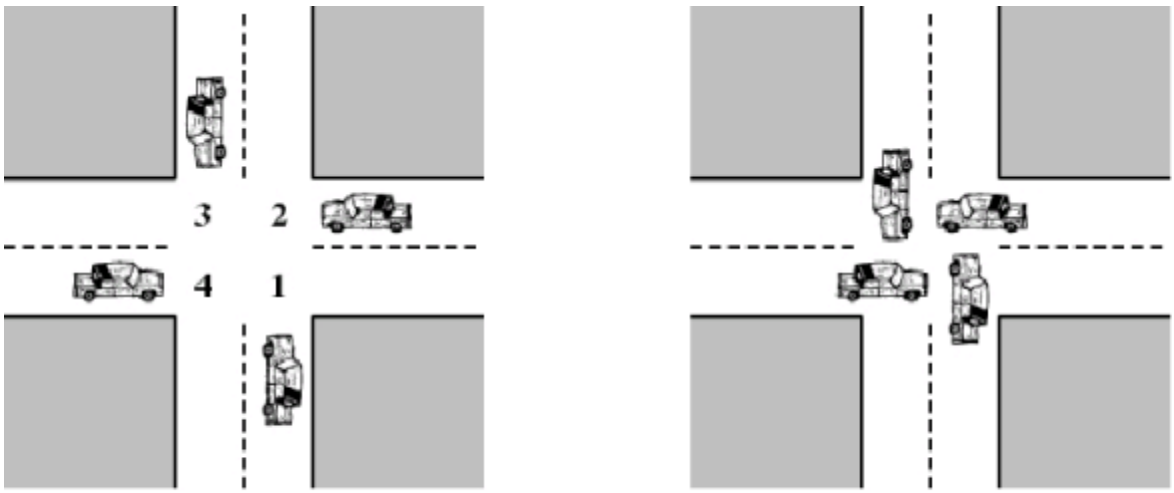
g. ~~*True or*~~ *False?* Deadlock avoidance is more restrictive than deadlock prevention.

h. *True* ~~*or False*~~*?* A process that is waiting for access to a critical section protected by semaphores does not consume processor time.

i. ~~*True or*~~ *False?* Multiple threads in the same process share all code and data resources.

j. ~~*True or*~~ *False?* Hardware support is always needed to synchronize more than two processes.

2. [10 points] Show that the four conditions of deadlock apply to the situation depicted below. Consider the intersection partitioned in 4 areas (or resources), numbered 1 to 4.



(a) Deadlock possible (b) Deadlock

The 4 conditions that must happen in order for deadlock to have a chance to occur are:

- Mutual exclusion: this refers to the fact that the resources are accessed in a mutually exclusive way. Specifically, in this case it means that one partition of the intersection can only be used by one car at any time.
- Hold-and-wait: this refers to the fact that once it entered the intersection (as in picture b), a car holds the resources it acquired (e.g., the partition 2 for the right-hand side car) and waits for access to the other resource(s) it needs (in this case, partition 3 of the intersection).
- Non-preemption: no car frees its partition before it reaches its objective of crossing the intersection.
- Circular wait: each car waits for a resource held by the car to the right (assuming they all want to go straight ahead), which leads to a circle of processes waiting for each other.

Grading note: there were many wrong explanations of the terms used, in which case I gave only 1 point for mentioning each of the conditions, but did not give credit for wrong application of the term to the intersection context.

3. [10 points] Suppose we have the philosophers at a table as in the Dinning Philosophers problem, but the chopsticks are placed in a tray at the center of the table when not in use. As in the original problem, any philosopher can eat only with two chopsticks, but unlike in the original problem, any two available chopsticks will do.

   One way to prevent deadlock in this system is to provide sufficient resources. For a system with *n* philosophers, what is the *minimum* number of chopsticks that ensures deadlock freedom? Why?

   The minimum number of chopsticks is n+1. Since they are placed in the middle, at least one philosopher will have access to 2 chopsticks at any time. She will eat and when done, she will release the chopsticks, enabling at least one other philosopher to eat, and so on.

4. [10 points] Process A is already written and requests the printer, the plotter, the tape drive, and the robotic arm, in that order. You need to develop processes B and C. Both need the printer and the arm, B needs the tape drive, and C needs the plotter. In what order should the requests be made? Why?

Since A is already written and the resources it requires are in the order (1) printer, (2) plotter, (3) tape drive, (4) robotic arm, then let's consider them indexed in this order from 1 to 4. In order to avoid circular wait (and thus deadlock), we need to make sure these resources are requested only in increasing order or their indices. Consequently, B will make its resource requests in the order printer (1), tape drive (3) and the robotic arm (4); C will request the printer (1), the plotter (2), and the robotic arm (4).

Note that the problem does not specify whether processes keep all the resources until they finish executed or release some of them before they take need another resources. If the latter is true, than a solution that does not follow the order above may lead to deadlock in the following case. If the printer is released immediately after use but the robotic arm and the tape are used together, and B requests: (1) printer; (4) robotic arm; (2) tape, then it can deadlock with A.

5. [10 points] Consider the following program executed by two different processes P1 and P2:

```
{
      shared int x;                          //s1
      x = 10;                                //s2
      while (1) {
            x = x - 1;                       //s3
            x = x + 1;                       //s4
            if (x != 10)                     //s5
                  printf("x is %d",x)        //s6
      }
}
```

Consider that the processes P1 and P2 are executed on a uniprocessor system. Note that the scheduler in a uniprocessor system would implement pseudoparallel execution of these two concurrent processes by interleaving their instructions, without restriction on the order of the interleaving.

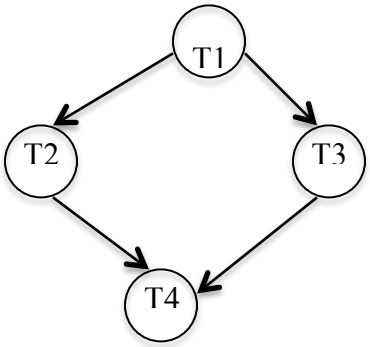
a. Show a sequence (i.e., trace the sequence of interleavings of statements) such that the statement "x is 10" is printed. Suggested format:
   Pi: <instruction> <relevant new value >
b. Show a sequence that leads to the statement "x is 8" be printed.

Hint: Remember that the increment/decrements at the source language level are not done atomically, e.g., the assembly language code below implements the single C increment instruction (x = x + 1).

```
LD R0,X /* load R0 from memory location x */
INCR R0 /* increment R0 */
STO R0,X /* store the incremented value back in X */
```

P1: x=x-1; //x=9
P2: x=x-1; //x=8
P1: x=x+1; //x=9
P1: if (x!=10) //true
P2: x=x+1; //x=10
P1: printf ("x is 10")
	P1: x=x-1; //x=9
	P2: LD R0, x
	DECR R0 //R0=8
	P1: x=x+1; //x=10
	P2: STO R0, x //x=8
	P1: printf("x is %d", x);

6. [15 points] Assume you are given a graph that represents the relationship between four threads (T1, T2, T3, T4). An arrow from one thread (Tx) to another (Ty) means that thread Tx must finish its computation before Ty starts. Assume that the threads can arrive in any order. Use semaphores to enforce this relationship specified by the graph. Be sure to show the initial values and the locations of the semaphore operations.



// Semaphores definitions and their initial values

```
init(sem1,0); //initialize sem1 to 0
init(sem2,0); //initialize sem2 to 0
```

T1(void){	T2(void){	T3(void){	T4(void){

//T1 computation
post(sem1);
post(sem1);
}
	wait(sem1);
	//T2 computation
	post(sem2);
	}
		wait(sem1);
		//T3 computation
		post(sem2);
		}
			wait(sem2);
			wait (sem2);
			//T4 computation
			}

Most people did some variation of this with 3 semaphores and received full credit. Some people used 4 semaphores in a correct solution, and were deducted 1 point for not finding a simpler solution.

7. **Extra credit [15 points]: This is a more difficult problem. Credit is given only for meaningful progress towards a correct solution.**

   Assume you are building a special operating system that requires executing the expression:

$$z = F3(F1(x),F2(F3(y)));$$

where x, y, and z are integers and F1, F2, F3 are functions.

The functions F1() and F3() must execute on thread T1 while the function F2 needs to execute on thread T2.

a. Your job is to write the code to force the expression to be evaluated regardless of the order they are run by the CPU scheduler. Note you will have to add code to all three functions below.

There are many different ways of solving this problem. A common solution is given here:

```
int z, x, y; //shared variables

//Declare shared variables and semaphore with initial values here.

//shared variables and semaphores with initial values:
semaphore T1_sem(0); //initialized to zero
semaphore T2_sem(0); //initialized to zero
semaphore done(0); //initialized to zero
```

`void ComputeZ(){`
 	
`StartThread(T1);`
 	
`StartThread(T2);`
 	
`sem_wait(done);`
 	
`return;`
 	
`}`
	`void T1(){`
	
	`y = F3(y);`
	
	`sem_post(T2_sem);`
	
	`x=F1(x);`
	
	`sem_wait(T1_sem);`
	
	`z = F3(x,y);`
	
	`sem_post(done);`
	`}`
		`void T2(){`
		
		`sem_wait(T2_sem);`
		
		`y = F2(y);`
		
		`sem_up(T1_sem);`
		`}`

Syntax didn't matter in this question as long as it was clear. 2 points were deducted for every race condition.

b. Does your solution handle the following cases? If so, explain why. If not, explain the problem and how you could fix it.

i. Would your solution handle the case when ComputeZ() is called twice in a row like:

```
x=...;  y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z);
x=...;  y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z);
```

Yes, the previous solution works well when running ComputeZ() twice sequentially. All semaphore values end up as zero at the end of ComputeZ(), and when ComputeZ() returns, z is guaranteed to be correct.

ii. Would your solution handle the case when ComputeZ() is called twice from two different threads? For example:

```
void Z1() {
x=...; y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z); }
void Z2() {
x=...; y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z); }
StartThread(Z1);
StartThread)(Z2);
```

There are multiple problems with the solution proposed in a) under this scenario. First, there are data inconsistency issues that occur inside T1() or T2() if ComputeZ() is called twice in parallel. Second, there are also race conditions inherent in Z1() and Z2() running in parallel when x and y are set (e.g., x could be set in Z1(), then overwritten by switching to Z2(), and then we might switch back to Z1() and run ComputeZ() using Z2()'s x value). To fix the second problem, you either need to run Z1() and Z2() atomically, or make x,y, and z local variables that are passed in/returned from ComputeZ instead.