Computing Science 300 Quiz 2 Spring 2019

1. Consider a process.
   a) **[9 points]** For each of the items in the table below one of the following statements is true
      i. One copy of the item is shared between all threads in the process
      ii. One copy of the item exists for each thread in the process

      For each item in the table below indicate which of the two statements above is true by placing an ✓in the column for your chosen statement.

Item	One copy shared between all threads	One copy per thread
Local variables		✓
Global variables	✓	
Program instructions	✓	
Registers		✓
File descriptors	✓	
Process control block	✓	
Thread control block		✓
Dynamic variables created before any threads are created	✓	
Stack		✓

   b) **[8 points]** A process begins to run its original thread. In the process X is a global variable. Y is a local (automatic) variable. In each row of the following table is an action taken by the process. Indicate the values of the two variables X and Y in each thread after the action listed in the first column.

	X in original thread	X in thread 1	X in thread 2	Y in original thread	Y in thread 1	Y in thread 2
Thread 1 is created	10	10	--------	20	20	-------
Thread 2 is created	10	10	10	20	20	20
Thread 2 sets Y=30	10	10	10	20	20	30
Thread 1 sets X=50	50	50	50	20	20	30
Thread 1 sets Y=70	50	50	50	20	70	30
Thread 2 sets X=13	13	13	13	20	70	30

c) **[10 points]** Consider a multithreaded process. The process states can include RUNNING, BLOCKED, or READY to run. Each thread within the threaded process has a thread state. The thread state of each thread in a process includes the states RUNNING, BLOCKED or READY. Fill in the first empty column in the table by indicating if the statement in each row is true or false for System level threads. Fill in the second empty column in the table by indicating if the statement in each row is true of false for user level threads.

Statement	System level threads	User level threads
If the one thread in the process is in RUNNING state then the process will be RUNNING State	True	False
If one thread in the process is in BLOCKED state then no other thread in the process can run	False	True
Threads of the same process may be scheduled on different cores of a multicore processor	True	False
The OS schedules each thread in the process	True	False
The OS knows how many threads there are in each process	True	False

2. Consider the following program.

```
int sum=0;
int main()
{
    /*  creates two threads
        thread 1 runs increase()
        thread 2 runs decrease()  */
}
```

```
void increase( ) {
    int count;
    for (count = 0; count < 5; count++) {
            sum += count;
    }
}
```

```
void decrease() {
        int count;
        for(count=5; count > 0; count--) {
                sum -= count;
        }
}
```

Assume the two threads can execute in any possible order and that a value of sum can only be incremented after it has been loaded into a register by a separate machine instruction. The lines of code **sum+=count** and **sum-=count** are each executed in three machine instructions

- load value of **sum** into a register from memory
- add value of **count** to the register
- load the result from the register into the variable **sum** in memory

You may assume that all other instructions (other than modifying sum) are never interrupted between the load and increment or between the increment and load back steps

For parts a) and b) below you are asked to fill in one row of a table for each time through the for loop in increase or the for loop in decrease. However, if the thread stops executing part way through the three steps listed above, that is part way through filling in a line of the table, leave the columns for the steps not executed blank. Then when the thread begins running again leave the already completed portion of the line empty and fill in results only for the parts of the loop executed after the process begins again. For example

Process running (increase or decrease)	Value of count	Value of sum loaded from memory to register	New value of sum Calculated in register	Value of sum copied from register to memory
decrease	5	0	-5	
increase	0	0	0	0
increase	1	0		
decrease				-5
Increase			1	1

a) **[15 points]** Determine the upper bound on the final value of the shared variable sum after both increase() and decrease() have run to completion. For an order of execution that produces the upper bound, give a step by step time ordered description by filling in the table below. The number of available rows in the table may not be the same as the number of steps in your solution.

Process running (increase or decrease)	Value of count	Value of sum loaded from memory to register	New value of sum Calculated in register	Value of sum copied from register to memory
Increase	0	0	0	
decrease	5	0	-5	-5
decrease	4	-4	-9	-9
decrease	3	-9	-12	-12
decrease	2	-12	-14	-14
decrease	1	-14	-15	-15
Increase				0
Increase	1	0	1	1
increase	2	1	3	3
increase	3	3	6	6
increase	4	6	10	10

b) **[20 points]** Determine the order of execution through the pair of processes that produces the answer sum=0 after both increase() and decrease() have run to completion. For the order of execution that produces tally=5 give a step by step time ordered description detailing the lines of code executed by each process each time it is in the CPU. The number of available rows in the table may not be the same as the number of steps in your solution.

Process running (increase or decrease)	Value of count	Value of sum loaded from memory to register	New value of sum Calculated in register	Value of sum copied from register to memory
Increase	0	0	0	
decrease	5	0	-5	-5
decrease	4			
Increase				0
increase	1	0	1	1
Increase	2	1	3	3
Increase	3	3	6	6
Increase	4	6	10	10
decrease		10	6	6
decrease	3	6	3	3
decrease	2	3	2	1
decrease	1	1	0	0

3. Consider the concept of mutual exclusion. Multiple processes share a resource. The portion (as small as possible) of each process that uses the resource is called a critical region. When mutual exclusion is used only one process at a time may be within the critical region for the resource being shared. For processes sharing one resource, resource A, the process can be represented by the following pseudo code:

```
ExecuteNonCriticalRegion()
StartCriticalRegionResourceA()
CriticalRegion()
EndCriticalRegionResourceA()
ExecuteNonCriticalRegion2()
```

Consider two processes. These processes share 2 resources, resource C and resource B.

a) **[10 points]** The **`StartCriticalRegion()`** and **`EndCriticalRegion()`** functions can be implemented in several ways including using semaphores, signals, and interrupts. Consider using interrupts to implement these functions. Give a step by step (maximum 2 short steps each function) description of using interrupts to implement each of the two functions. Be sure to indicate which interrupts (what type of interrupts) are used. Then, consider each of the situations below and indicate if interrupts would be a good choice for the implementation of mutual exclusion in each of the described situations. For each situation where interrupts would not a good choice give one reason why.

- An operating system running a multiple user system on a single core CPU. The operating system includes a user mode and a system mode
- An operating system running on a multiple core CPU.
- A custom embedded operating system running on one single core dedicated to controlling and managing a particular device.

`StartCriticalRegion()`
Turn off all non maskable interrupts (those that can be disabled)
Turn back on any interrupts needed by this function

`EndCriticalRegion()`
Re-enable all disabled interrupts

A multiple user operating system on a single core CPU is not a good candidate for using interrupts to implement mutual exclusion. If one user is able to access masking of interrupts (a system mode function) then that user can override (turn off) interrupts (timer interrupts) used for scheduling and allow their own code to run without being interrupted by other processes. The user could also turn on interrupts if they had been turned off to protect another users code.

An operating system running on a multiple core CPU is not a good candidate for using interrupts to implement mutual exclusion. When interrupts are disabled, they are disabled on one core not an all cores. This means that the resource being protected could be accessed

simultaneously by two processes each running on different cores. Mutual exclusion does not work.
A custom embedded operating system running on one single core dedicated to controlling and managing a particular device is an appropriate place to use interrupts to implement mutual exclusion.

b) **[9 points]** Assume the critical regions for resource C and resource B cannot be separated. Pseudo code for this situation is given below. Give a path through the pseudo code for the two processes below for which the two processes become deadlocked. The path of execution you give MUST include each process leaving the CPU at least twice before deadlock occurs.

PROCESS 1	PROCESS 2
1) `ExecuteNonCriticalRegion0()`	1) `ExecuteNonCriticalRegion0()`
2) `StartCriticalRegionResourceC()`	2) `StartCriticalRegionResourceB()`
3) `StartCriticalRegionResourceB()`	3) `StartCriticalRegionResourceC()`
4) `ExecuteCriticalRegionCandB()`	4) `ExecuteCriticalRegionCandB()`
5) `EndCriticalRegionResourceB()`	5) `EndCriticalRegionResourceB()`
6) `EndCriticalRegionResourceC()`	6) `EndCriticalRegionResourceC()`
7) `ExecuteNonCriticalRegion2()`	7) `ExecuteNonCriticalRegion2()`

Execute PROCESS 2 until line 1 is complete
Execute PROCESS 1 until line 1 is complete
Execute PROCESS 2 until line 2 is complete
Execute PROCESS 1 until line 2 is complete
Execute PROCESS 2 line 3, because resource A is being used by PROCESS 1, PROCESS 2 blocks
Execute PROCESS 1 line 3, because resource B is being used by PROCESS 2, PROCESS 1 blocks
DEADLOCK

c) **[4 points]** Given the constraint that you must use the ALL the same functions as used in part b) what would you change in the pseudocode for PROCESS 2 to fix the deadlock you illustrated in part b). Fixing the deadlock in part b) should not introduce any other deadlocks.

Exchange lines 2 and 3

d) **[15 points]** Assume that PROCESS 1 uses resource C then resource B and PROCESS 2 uses resource B then resource C. In each process the critical regions for resource C and resource B are separated by an additional non critical region. The pseudo code function that executes the additional non critical region is **`ExecuteNonCriticalRegion1().`** The pseudo code functions that execute the separated critical regions are **`ExecuteCriticalRegionC()`** and **`ExecuteCriticalRegionB().`** Write pseudo code for PROCESS 1 and PROCESS 2. You may use all functions mentioned in part b) and part d).

PROCESS 1

```
1) ExecuteNonCriticalRegion0()
2) StartCriticalRegionResourceC()
3) ExecuteCriticalRegionC()
4) EndCriticalRegionResourceC()
5) ExecuteNonCriticalRegion1()
6) StartCriticalRegionResourceB()
7) ExecuteCriticalRegionB()
8) EndCriticalRegionResourceB()
9) ExecuteNonCriticalRegion2()
```

PROCESS 2

```
1) ExecuteNonCriticalRegion0()
2) StartCriticalRegionResourceB()
3) ExecuteCriticalRegionB()
4) EndCriticalRegionResourceB()
5) ExecuteNonCritcalRegion1()
6) StartCriticalRegionResourceC()
7) ExecuteCriticalRegionC()
8) EndCriticalRegionResourceC()
9) ExecuteNonCriticalRegion2()
```