Computing Science 300 **Quiz 2** **Sample problem solutions**

1. Consider a process running on a LINUX system. Answer each of the following questions. The answer to each question inside each part of this problem should be no more than two sentences long.
   a) [6 points] The first thread in the process sets a global variable, myVariable, to have a value 3. The process then creates a second thread. Is the value of the global variable, myVariable, in the first thread the same as the value of the global variable, myVariable, in the second thread immediately after the second thread is created? Does the value of the global variable, myVariable, always have the same value in the second thread and in the first thread? Briefly Explain why?
   The value of the global variable be the same immediately after the second thread is created. The value of the global variable will always be the same for all threads.
   Global variables are shared between threads so when one thread changes the global variable, the changed value is visible to all other threads.
   b) [6 points] The first thread in the process sets an automatic local variable, myVariable, to have a value 3. The process then creates a second thread. Is the value of myVariable in the first thread the same as the value of the variable myVariable, in the second thread immediately after the second thread is created? The second thread runs, sets myVariable to a value of 23 and then tells the first thread to run again. What is the value of the local myVariable, in the first thread when it begins to run for the second time? Why?
   Yes, the values are the same immediately after the second thread is created are both 3
   When the new thread is created it creates its own copy of the local variables in the function the thread is about to execute. Those variables will have the initial values defined in the function to be executed. This is true for the first thread created and for any subsequent thread created.
   When the first thread begins to run after the second thread has completed its execution the value in myVariable in the first thread is 3.
   Each thread has its own copy of the local variables so if the 2nd thread changes its copy of a local variable it does not change the value of the variable in the 1st thread. The value or myVariable in the first thread has not be modified so its value is still 3.
   c) [8 points] Consider scheduling of processes and user level threads. Is it possible for the threads in the process to be scheduled onto different processors in a multiprocessor CPU? Briefly explain why.
   NO
   When user level threads are used the scheduling of threads is managed using the thread library in user space. The OS does not know the specifics of what code it is being run inside the process, it is only aware of system function calls. Since user level threads are part of the code running inside the process, and the user level thread libraries do not make system calls to system level thread system functions the OS is not even aware that the process contains threads. A single process is that does not contain system level threads is always run on a single processor.

2. Consider the following program.

```
const int n=10;
int tally=0;
void total( ) {
    int count;
    for (count = 0; count < n; count++) {
            tally ++;
    }}
```

The main program using this function sets tally to 0, then starts two concurrent copies of process total. (Process total runs function total). Assume the two processes can execute at any relative speed and that a value can only be incremented after it has been loaded into a register by a separate machine instruction. (That is tally++ is executed in three machine instructions, load into register, then increment, then load back into variable in memory). You may assume that all other instructions (other than incrementing tally) are never interrupted between the load and increment or between the increment and load back steps

a) **[10 points]** Determine the upper bound on the final value of the shared variable tally after both copies of total( ) have run to completion. For the order of execution that produces the upper bound give a step by step time ordered description detailing the lines of code executed by each process each time it is in the CPU .
b) **[30 points]** Determine the order of execution through the pair of processes that produces the answer tally = 5 after both copies of the process total() 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 . Give you answer as a list of steps or a table of steps.

The maximum value of tally can be produced if the two processes run in many different orders. In order to calculate the maximum (the correct solution 20) the only constraint is that the three machine instructions to increment tally must never be interrupted by the other process. This means that If those three machine instructions always execute in one single block then the maximum value of tally would be calculated. The other process can be started after any machine instruction in the process except the load tally into register or the increment the register containing tally instructions.

For example the maximum value of tally would be determined if the two processes (copy1 of total() and copy2 of total( )) run as follows

1. ***Copy 1 runs until the for loop is entered (tally has not been incremented the first time)***
2. ***Copy 2 runs until it completes the loop.***
3. ***Copy 1 runs until it terminates.***
4. ***Copy 2 terminates***

The maximum value of tally is 20

To produce tally =5 the two processes (copy1 of total() and copy2 of total( )) might run as follows

1. *Copy 1 runs until it enters the for loop then loads the value of tally into the register, increments it and copies it back to the variable tally, repeating the process twice so that the value of tally is 3. Tally is then loaded in the register and incremented to 4*
2. *The scheduler replaces copy1 with copy 2 before copy 1 writes the value of tally back to the variable tally.*
3. *Therefore, the current value of tally when copy2 reads it for the first time is 3. Copy2 runs until the value placed in variable tally is 12 and is then interrupted. (interrupted immediately after writing 12 to tally)*
4. *The scheduler replaces copy2 with copy1, which immediately writes the value 4 to the variable tally, and is interrupted*
5. *The scheduler replaces copy1 with copy2 which immediately reads tally and gets the value 4, then increments to 5 and is interrupted.*
6. *The scheduler replaces copy2 with copy1 which then runs to completion leaving a value of 10 in variable tally.*
7. *The scheduler then brings back copy1 which writes 5 to the variable tally and ends*

Copy1				*Copy 2*			
Value copied into register	*Value in register after increment*	*Value copied back into variable*	*count*	*Value copied into register*	*Value in register after increment*	*Value copied back into variable*	*count*
0	*1*	*1*	*0*				
1	*2*	*2*	*1*				
2	*3*	*3*	*2*				
3	*4*		*3*				
SWAP TO COPY 2							
				3	*4*	*4*	*0*
				4	*4*	*5*	*1*
				5	*6*	*6*	*2*
				6	*7*	*7*	*3*
				7	*8*	*8*	*4*
				8	*9*	*9*	*5*
				9	*10*	*10*	*6*
				10	*11*	*11*	*7*
				11	*12*	*12*	*8*
				SWAP TO COPY 1			
		4	*3*				
SWAP TO COPY 2							
				4	*5*		*9*
				SWAP TO COPY 1			
4	*5*	*5*	*4*				
5	*6*	*6*	*5*				
6	*7*	*7*	*6*				
7	*8*	*8*	*7*				
8	*9*	*9*	*8*				
9	*10*	*10*	*9*				
COPY 1 FINISHED SWAP TO COPY 2							
						5	*9*
COPY 2 FINISHED							

3. **[40 points]** Consider the pseudo code below. The pseudo-code is given for parts of two processes P0 and P1 that will be sharing the CPU in a multiprocessing operating system. Mutual exclusion is being used to share a resource between these two processes. Assume that when the first block of code below to run begins to execute its loop that both P0_wants and P1_wants are zero.
   For each part of this problem give a time ordered instruction by instruction list of a particular path of execution through processes P0 and P1. Each of these paths of execution must describe
   - Which instructions each process runs when it begins a quanta in the CPU and which instructions it partially completes or completes during the quanta before it leaves the CPU.
   - The tasks completed by the instructions that are executed

   a) Give a path through the two processes for which the resource is successfully protected by mutual exclusion. The path of execution you give MUST include each process leaving the CPU at least twice.

   b) Give a path through the two processes for which the resource is NOT successfully protected by mutual exclusion because deadlock occurs. The path of execution you give MUST include each process leaving the CPU at least twice.

```
1) int P0_wants;
2) while(true) {
3)       P0_wants = true;
4)       while (P1_wants ) { };
5)       CriticalSection();
6)       PO_wants = false;
7)       NonCriticalSection();
8) }
```

```
1) int P1_wants;
2) while(TRUE) {
3)       P1_wants = TRUE;
4)       while (P0_wants ) { };
5)       CriticalSection();
6)       P1_wants = FALSE;
7)       NonCriticalSection();
8) }
```

SOLUTION

a)

Begin with process0 (PO) and process 1 (P1) having no intent to enter a critical region

- **P1 begins to run (executes line 2)**
- **P1 declares it intent to use the resource (executes line 3)**
- **P1 checks and finds P0_wants is false so it does not busy wait (executes line 4)**
- **P1 begins to execute its critical section (begins executing line 5)**
- **P1 is swapped out of the CPU when it quanta is complete**
- **P0 is loaded into the CPU and begins to execute.at line 2**
- **P0 declares its intent to enter a critical region by setting P0=true (executes line 3)**
- **P0 "busy waits" until the scheduler removes it from the CPU (executes line 4)**
- **P1 completes its critical section, then the scheduler removes it from the CPU (finishes executing line 5)**
- **P0 is swapped into the CPU and continues to busy wait until its quanta ends and it is swapped out of the CPU (executes line 4)**
- **P1 indicates it no longer wants the resource by setting P1_wants to false (executes line6)**
- **P1 runs its Noncritical region and finishes execution (executes line 7)**
- **PO is swapped into the CPU**

- **P0's "busy wait" ends since P1_wants is now false. P0 completes its critical region then sets P0_wants to 0 and continues with its non- critical region until it completes execution (executes lines 4-7)**
- **Both processes were able to complete their critical regions so that for this case the two processes are successfully sharing the resource protected by mutual exclusion/**

b)

Begin with process0 (PO) and process 1 (P1) having no intent to enter a critical region

- **P1 begins to run (executes line 2)**
- **P1 declares it intent to use the resource (executes line 3)**
- **P1 is swapped out of the CPU when it quanta is complete**
- **P0 is loaded into the CPU and begins to execute.at line 2**
- **P0 declares its intent to enter a critical region by setting P0=true (executes line 3)**
- **P0 "busy waits" until the scheduler removes it from the CPU (executes line 4)**
- **P1 is loaded into the CPU and begins to execute line 3. Because P0_wants is true P1 busy waits until its quanta is over and it is swapped out of the CPU**
- **P0 is swapped into the CPU. P1_wants is true so P0 continues to busy wait until its quanta ends and it is swapped out of the CPU (executes line 4)**
- **P1 is loaded into the CPU. P0_wants is true so P1 continues to busy wait until its quanta is over and it is swapped out of the CPU. (executes line4)**
- **P0 is swapped into the CPU and continues to busy wait until its quanta ends and it is swapped out of the CPU (executes line 4)**
- **Processes 1 is waiting for process 2. Process 2 is waiting for process1.**
- **Process 1 and 2 are deadlocked and cannot proceed. Therefore, the two processes are not successfully sharing the resource and mutual exclusion is broken**