**Southern Methodist University**
**Bobby B. Lyle School of Engineering**
**Department of Computer Science**
**CS 7343/5343 Operating Systems and System Software**

- Include a front page with course title, your name, Student ID, your e-mail address, and the course number (e.g. CS 5343 or CS 7343). You must also indicate whether you are a distance student or an in-person student.
- Each answer should begin in a new page.
- **Note**:
  - There will be 10 points deduction if this information is missing.
  - Late Homework submission must be sent directly to the grader via email.

  - **All students who are signed for this course at the CS 7343 must answer all questions.**
  - **All students who are signed for this course at the CS 5343 must answer exactly four questions.**

1. Consider the following program:

```
const int n = 50;
int tally;
void total()
{
   int count;
   for (count = 1; count <= n; count++){
      tally++;
   }
}
```

```
void main()
{
   tally = 0;
   parbegin (total (), total ());
   write (tally);
}
```

```
void main()
{
   tally = 0;
   parbegin (total (), total ());
   write (tally);
}
```

The key word **Parabegin** indicates that both calls to the function total are executed concurrently. Determine the proper lower bound and upper bound on the final value of the shared variable **tally** output by this concurrent program. Assume 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. You must explain how you've arrived to the final answer.

Hints: Notice that we have two threads (call them T1 and T2) running simultaneously. This is evident from the two concurrent invocation of the function total() two times. Also, notice that the variable **tally** is shared. This implies that both threads have access to the same variable and, hence, they may modify the value of tally independent of each other.

Under the best scenario, both T1 and T2 will run sequentially. In this case, T1 will run into completion followed by T2 as in

```
void main()
{
   tally = 0;

    total ()

    total ()

 }
```

On the other hand, Thread T1 and T2 could be running in such away so that they will

Tally++

Assembly: In this assembly code tally is a location in memory and R0 of he CPU registers

1. ***load tally to R0***
2. ***increment R0***
3. ***move R0 to tally***

Consider the following senario

1. Thread A loads the value of `tally`, increments *tally*, but then loses the processor (it has incremented its register to 1, but has not yet stored this value.
2. Thread B loads the value of `tally` (still zero) and performs forty-nine complete increment operations, losing the processor after it has stored the value 49 into the shared variable `tally`.
3. Thread A regains control long enough to perform its first store operation (replacing the previous `tally` value of 49 with 1) but is then immediately forced to relinquish the processor.

There are three execution steps that Thread A and Thread B could execute that result in the minimum value of tally. Try to deduce these steps.

**4.** Thread B resumes long enough to load 1 (the current value of *tally*) into its register, but then it too is forced to give up the processor (note that this was B's final load).

**5.** Thread A is rescheduled, but this time it is not interrupted and runs to completion, performing its remaining 49 load, increment, and store operations, which results in setting the value of `tally` to 50.

**6.** Thread B is reactivated with only one increment and store operation to perform before it terminates. It increments its register value to 2 and stores this value as the final value of the shared variable.

2. Examine the following pseudo code in which p and q, defined as shown below. A, B, C, D, and E atomic (indivisible) statements. Assume that the main program execute these two processes concurrently.

```
void p()
{
    A;
    B;
    C;
}
```

```
void q()
{
    D;
    E;
}
```

```
Void main()
  {
    Parbegin p(), q() //Call method p() and method q()  simultaneously
  }
```

   a. Show all possible execution paths of this program. For example, a possible execution path would be ABCDE.

answer: ABCDE; ABDCE; ABDEC; ADBCE; ADBEC; ADEBC; DEABC; DAEBC; DABEC; DABCE

b. Show **5 impossible paths** of this program. For example, an impossible path would be EDABC.

The easiest way is to graphically show all possible paths and subtract from them the legal paths stated in part a. Or simply all paths except those in part a are impossible

3. For each of the following thread state transitions, say whether the transition is legal *and* how the transition occurs or why it cannot.

   a. Change from thread state BLOCKED to thread state RUNNING

      Illegal. The scheduler selects threads to run from the list of ready (or runnable) threads. A blocked thread must first be placed in the ready queue before it can be selected to run.

   b. Change from thread state RUNNING to thread state BLOCKED

      Legal. A running thread can become blocked when it requests a resource that is not immediately available (disk I/O, lock, etc).

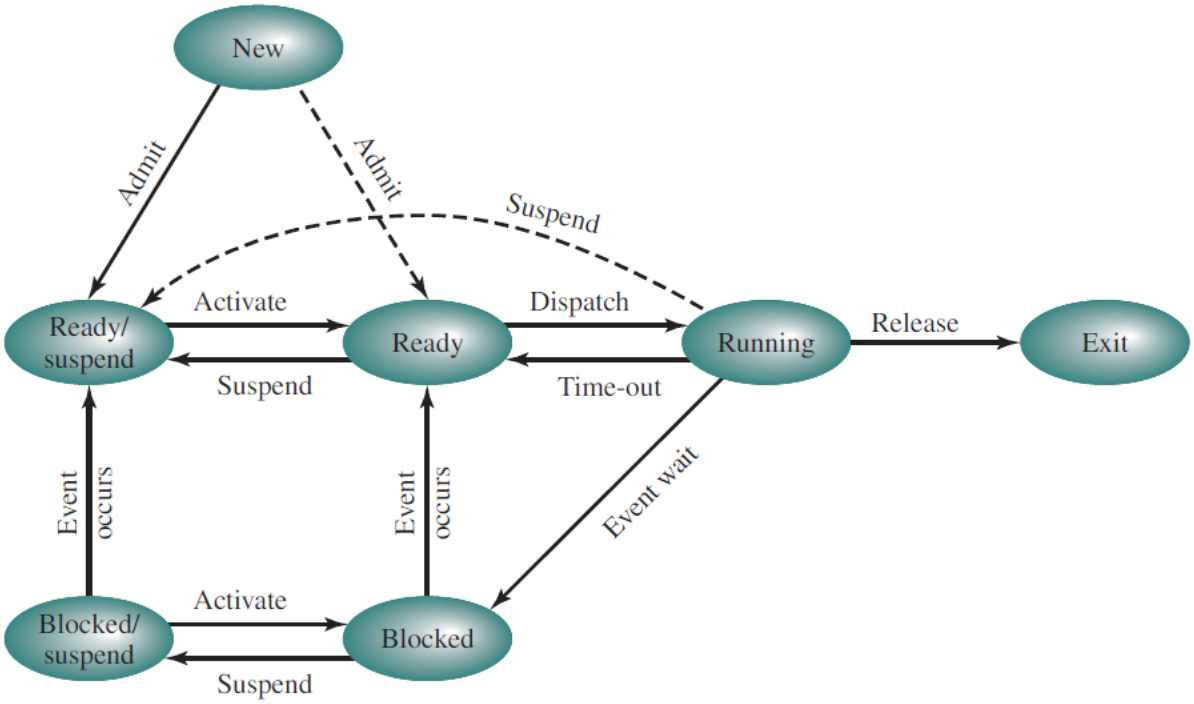
   c. Change from thread state RUNNABLE (i.e., in the ready queue) to thread state BLOCKED

      Illegal. A thread can only transition to BLOCKED from RUNNING. It cannot execute any statements when still in a queue (i.e. the ready queue).

4. Consider an environment in which there is an equivalence mapping between user-level and kernel-level threads (i.e. that is we have a one-to-one mapping between user threads and kernel threads) that allows one or more threads within a process to issue blocking system calls while other threads continue to run. Will this approach make multithreaded programs run faster than their single-threaded counterparts on a uniprocessor computer?

   The issue here is that a machine spends a considerable amount of its waking hours waiting for I/O to complete. In a multithreaded program, one KLT can make the blocking system call, while the other KLTs can continue to run. On uniprocessors, a process that would otherwise have to block for all these calls can continue to run its other threads.

5. Consider the figure below. Show all possible transitions and give a scenario in which each transition could occur.



a. **New → Ready or Ready/Suspend**: covered in text
**Ready → Running or Ready/Suspend**: covered in text
**Ready/Suspend → Ready**: covered in text
**Blocked → Ready or Blocked/Suspend**: covered in text
**Blocked/Suspend → Ready /Suspend or Blocked**: covered in text
**Running → Ready, Ready/Suspend, or Blocked**: covered in text
**Any State → Exit**: covered in text

b. **New → Blocked, Blocked/Suspend, or Running**: A newly created process remains in the new state until the processor is ready to take on an additional process, at which time it goes to one of the Ready states.
**Ready → Blocked or Blocked/Suspend:** Typically, a process that is ready cannot subsequently be blocked until it has run. Some systems may allow the OS to block a process that is currently ready, perhaps to free up resources committed to the ready process.
**Ready/Suspend → Blocked or Blocked/Suspend:** Same reasoning as preceding entry.
**Ready/Suspend → Running:** The OS first brings the process into memory, which puts it into the Ready state.
**Blocked → Ready /Suspend**: this transition would be done in 2 stages. A blocked process cannot at the same time be made ready and suspended, because these transitions are triggered by two different causes.
**Blocked → Running**: When a process is unblocked, it is put into the Ready state. The dispatcher will only choose a process from the Ready state to run
**Blocked/Suspend → Ready**: same reasoning as Blocked → Ready /Suspend
**Blocked/Suspend → Running**: same reasoning as Blocked → Running
**Running → Blocked/Suspend**: this transition would be done in 2 stages
**Exit → Any State**: Can't turn back the clock

a. **New → Ready or Ready/Suspend**: covered in text
**Ready → Running or Ready/Suspend**: covered in text
**Ready/Suspend → Ready**: covered in text
**Blocked → Ready or Blocked/Suspend**: covered in text
**Blocked/Suspend → Ready /Suspend or Blocked**: covered in text
**Running → Ready, Ready/Suspend, or Blocked**: covered in text
**Any State → Exit**: covered in text

b. **New → Blocked, Blocked/Suspend, or Running**: A newly created process remains in the new state until the processor is ready to take on an additional process, at which time it goes to one of the Ready states.
**Ready → Blocked or Blocked/Suspend:** Typically, a process that is ready cannot subsequently be blocked until it has run. Some systems may allow the OS to block a process that is currently ready, perhaps to free up resources committed to the ready process.
**Ready/Suspend → Blocked or Blocked/Suspend:** Same reasoning as preceding entry.
**Ready/Suspend → Running:** The OS first brings the process into memory, which puts it into the Ready state.
**Blocked → Ready /Suspend**: this transition would be done in 2 stages. A blocked process cannot at the same time be made ready and suspended, because these transitions are triggered by two different causes.
**Blocked → Running**: When a process is unblocked, it is put into the Ready state. The dispatcher will only choose a process from the Ready state to run
**Blocked/Suspend → Ready**: same reasoning as Blocked → Ready /Suspend
**Blocked/Suspend → Running**: same reasoning as Blocked → Running
**Running → Blocked/Suspend**: this transition would be done in 2 stages
**Exit → Any State**: Can't turn back the clock

6. Consider the Shell sort algorithm as explained in https://www.youtube.com/watch?v=ddeLSDsYVp8 Is it possible to use two or more threads to implement this algorithm. Explain why/why not.

   Yes. Each subset of the unsorted list can be independently sorted by a separate thread.

7. Suppose a process P spawns one or more threads. If the process P terminates and exits, will these threads continue to run? Explain.
   No. When a process exits, it takes everything with it—the KLTs, the process structure, the memory space, everything—including threads.