OPERATING SYSTEMS (COMP 310/ECSE 427) – Midterm – February 28, 2013

Name: **Answers** **Student ID:**

Provide **brief answers to all** questions. This exam is worth **100 points** in total.

1. This question has multiple parts **[20 points]**
    a. Briefly explain the necessity for splitting the address space and maintaining a kernel part and a user part in a modern operating system.

Splitting the address space into kernel and user part allows the kernel part to reside in the memory while the user part keeps changing with each context switch. Also, the split allows us to have different protection levels for the kernel part. To access the kernel memory, the process got to run at a higher privilege level, which allows protection of the kernel address space.

    b. Suppose a computer is running two applications A and B. Show the view of the virtual address space at different times while running the two applications.

[Kernel Address Space | Process A for User Space] while running process A
[Kernel Address Space | Process B for User Space] while running process B

    c. Why do we need two stacks per process?

One stack is used while the process is in user mode. The other stack is used while the process is in kernel mode. We need a separate kernel mode stack (that is held in the kernel address space) so that a process running in user mode cannot corrupt it.

    d. What is busy waiting? How can busy waiting lead to priority inversion?

Busy waiting is when a process that is unable to take a lock is repeatedly checking whether it can take the lock. Because processes do not go into waiting state when they are unable to take lock, we can have the following scenario. Suppose process X is busy waiting to get into a critical section, while process Y is occupying the critical section and X has higher priority than process Y. With a priority based scheduling policy, process X will not relinquish the processor for process Y to get out of the critical section. So, process X (high priority) is prevented from making progress by process Y (lower priority).

    e. What is spin lock? Why would an operating system use spin lock inside the kernel instead of blocking?

Memory variables are used as locks inside the kernel and hardware provided instructions such as Test-And-Lock or Compare-And-Swap is used to busy wait on the correct value on the memory variable. This is called spin lock. Operating systems use spin locking instead of blocking, it is more efficient to spin lock. When locks can be obtained in short time, it is better to spin lock, which is the case inside the kernel.

2. Deadlock problems [**25 points**]
    a. State the necessary and sufficient conditions for a deadlock to happen.

Circular wait.

Hold and wait.

No preemption.

Mutual exclusion.

    b. You are allocating resources for four processes using Banker's algorithm. There are eight units of resource of a single type in the system. The available at any time is less than eight because some might be already allocated. At a given time the allocation vector (hold vector) looks like [1, 0, 5, y], where y is unknown. The "Max" or "Claim" vector is [3, 2, 7, x], where x is another unknown. Find the minimum x (0 is a valid number) for which the system could go into unsafe state. Show the allocation vector at that point. Assume resources are allocated one unit at a time.

You could have the system going into unsafe state for x = 0. The allocation vector at that point looks like [2, 1, 5].

    c. Consider a situation where N processes are sharing M resources that can be reserved or released one at a time. The maximum claim (the **Max** vector in the slides) that is made by any process does not exceed M, and the sum of all maximum claims is less than M + N. Show that deadlock cannot occur. **Hint:** Use the banker's algorithm. At deadlock all resources are allocated because resources are allocated one at a time.

Total claim = Total Needed + Total Allocation (from banker's algorithm).

It is given that total claim < M + N.

Also, total allocation = M.

Therefore, Total Needed < N.

That is, at least one process does not need anything more. So, deadlock cannot occur.

3. Synchronization problems [**30 points**]
    a. Implement a barrier synchronization mechanism using a monitor. In barrier synchronization, a process calling the barrier() waits until all the processes have called the barrier(). Your barrier should be reusable (can be called multiple times).

```
Monitor Barrier {
        condvar notLast
        int n, nvalue;

        void init(val) {
                n = nvalue = val
        }

        void wait() {
                n = n -1
                if (n > 0)
                        notLast.wait()
                else
                        notLast.signal()
        }

        void reset() {
                n = nvalue;
        }
}
```

    b. Implement a modified semaphore using monitor. In this semaphore, the wait() can take a priority value. When a signal is made on the semaphore, it wakes up the process that has the highest priority among the ones already waiting. If no process is waiting, the semaphore remembers the signal and lets the next wait() to fall through whatever its priority is.

```
Monitor WeightedSemaphore {
        condvar semNotReady[N]
        int value
        int waitCount[N]

        init(int ival) {
                value = ival
                initialize waitCount array to 0.
        }
```

```
        void wait(int weight) {
                value = value -1
                if (value < 0) {
                        waitCount[weight]++;
                        semNotReady[weight].wait();
                }
        }

        void signal() {
                value = value + 1
                if (value >= 0) {
                        for I = N downto 0:
                                if (waitCount[i] > 0) {
                                        waitCount[i]—
                                        semNotReady[i].signal()
                                        break
                                }
                }
        }
}
```

4. Scheduling problems [**25 points**]
    a. Describe the Stride scheduling algorithm. Be brief and provide the key details.

Here is a simple description of stride scheduling.

Each process has a meter value. When a process runs on the CPU, it's meter value is incremented. The rate at which the meter is incremented is equal to 1/bribe, where bribe is the number of tickets the process has. The scheduler at any given time picks the process with the smallest meter value.

    b. Rate monotonic scheduler is asked to schedule a task with deadline 50 and service time 30. Give a second task it cannot schedule but an earliest deadline scheduler can schedule.

Consider another task with a deadline of 75 and service time of 25. This task has lower priority than the first task. The RMA will schedule the first task every 50 time units. In the first 75 units, there will be only 20 free time units. Therefore, RMA cannot schedule the second task. The earliest deadline first scheduler can schedule both tasks. In the first 150 time units, we have 30 + 25 + 30 + 25 + 30 = 140 time units. Therefore, EDF can find a feasible schedule.

c. Schedule the following tasks using a multi-level feedback queue scheduler. There are four levels in the scheduler and the maximum quantum increases from 1 (at the top) to 8 at the bottom. Task 1 (arrival 0, service 3), Task 2 (arrival 2, service 5), Task 3 (arrival 4, service 4), Task 4 (arrival 6, service 4)