# COMS W4118 Midterm
# March 13, 2008

Prof. Erich Nahum

1. (20 points) For each of the following, mark **TRUE** or **FALSE**. No justification is necessary (and no partial credit will be given for wrong answers). (2 points each)

   i. An OS can be viewed as a "resource multiplexor" sharing resources across multiple users and processes. **TRUE**

   ii. Counting semaphores can be implemented using regular binary semaphores combined with regular code. **TRUE**

   iii. SJF is primarily concerned with fairness. **FALSE**

   iv. A process control block would contain a stack pointer. **TRUE**

   v. A thread control block would contain a run queue. **FALSE**

   vi. All threads in a process share the heap. **TRUE**

   vii. All threads in a process share the stack. **FALSE**

   viii. Disabling interrupts is usually a priviledged operation. **TRUE**

   ix. Trapping into the kernel is usually a priviledged operation. **FALSE**

   x. If a process has only one thread, the library functions that it uses still need to be reentrant. **TRUE FALSE (full credit for either)**

2. (15 points) For each term below, give a short description (1-2 sentences) (3 points each):

   i. Deadlock

   *Deadlock* is when two processes or threads cannot make progress because each is waiting on the other to release a resource, typically a lock. Example: Process A acquires lock 1, B acquires lock 2, A tries to acquire lock 2, B tries to get lock 1.

   ii. Priority inversion

   *Priority inversion* is when a *higher priority process cannot complete* because it is *waiting on a lock held by a lower priority process* which in turn cannot run because it does not have sufficient priority. Example: A is high priority, B is medium priority, C is low priority. C acquires a lock, and is then prempted by B. A preempts B, but cannot acquire the lock. C cannot run since B is higher priority. The solution is *priority inheritance*, where A "lends" its higher priority to C until C releases the lock.

   iii. Non-preemptive scheduling

   *Non-preemptive scheduling* is when a scheduler *does not preempt a running process or thread*. Instead, the process or thread must *voluntarily relinquish control* of the CPU by blocking for I/O, exiting, or explictly yielding.

   iv. Interrupt

   An *Interrupt* is a *hardware or software event* that interrupts the *control flow of the processor*. Examples: a disk controller generates a hardware interrupt to notify the CPU that a disk block is ready; a system call in Linux is generated using a software interrupt instruction.

   v. `TASK_INTERRUPTIBLE`

   This is one of the *process states* in the Linux kernel. It indicates that a process is *blocked or sleeping on a wait queue*, but can be woken up by receipt of a *signal*.

3. (20 points) Given the following processes, run times, and arrival times, draw a Gannt diagram showing the scheduling using the shortest remaining processing time scheduling discipline. What is the average turnaround time from arrival to completion for all jobs? Assume context switching takes 0 time, and that all jobs do no I/O. *Show all your work to maximize any partial credit.*

| Process | Run Time | Arrival Time |
|---|---|---|
| P1 | 4 | 0 |
| P2 | 10 | 2 |
| P3 | 1 | 2 |
| P4 | 3 | 3 |
| P5 | 2 | 5 |

The solution is as follows:

| P1 | P1 | P3 | P1 | P1 | P5 | P5 | P4 | P4 | P4 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Time 00: P1 arrives and runs for 2 cycles.
Time 02: P2 arrives and waits, since it has 10 cycles remaining, more than P1.
Time 02: P3 arrives and preempts P1 (P1: 1 cycle remaining; P2: 2)
Time 03: P3 completes, P4 arrives. P1 resumes (P1: 2 cycles; P4: 3; P2: 10).
Time 05: P3 completes, P5 arrives. P5 runs ( P5:2; P4:3; P2:10).
Time 07: P5 completes. P4 runs.
Time 10: P4 completes. P2 runs.
Time 20: P2 completes.

Turnaround time is simply completion time minus arrival time for each job:
P1: Arrived 0, finished 05, total 05.
P2: Arrived 2, finished 20, total 18.
P3: Arrived 2, finished 03, total 01.
P4: Arrived 3, finished 10, total 07.
P5: Arrived 5, finished 07, total 02.

Total is (5+18+1+7+2) equals 33, divided by the number of jobs (5) equals an average turnaround time of 6.6 time units.

4. (20 points) You have been given the task to secure an intersection with a 4-way stop sign. The usual rules of the road apply: the first to come to the stop sign gets to cross the intersection and there can only be one car in the intersection at any one time. For politeness, if someone arrived at the intersection from another street before you arrived at yours, you should let them go first. Outline your solution using counting semaphores in pseudo-C. Assume semaphores have a FIFO queuing behavior, and that cars cannot arrive simultaneously (i.e., there are no ties in arrival time). Be careful to state any assumptions, as not all implementations have the same behavior.

   The solution is simple: one semaphore for each stop sign, and one semaphore for the intersection itself.

```
semaphore stop_sign[4] = {1} ; /* N, E, S, W */
semaphore intersection = 1;


my_direction = get_my_direction(); /* N, E, S, W */
P(stop_sign[my_direction]);
P(intersection);
// drive through intersection
V(intersection);
V(stop_sign[my_direction]);
```

5. (10 points) A computer has a floating point instruction called INV. Much like a hand-held calculator, INV takes a value in memory or a register, computes 1.0/X, and stores the result back into the same place. At the same time it clears the overflow bit in the floating point status register. If X was zero, INV stores a 1.0 into X and sets the overflow bit in the floating point processor status register. *This occurs atomically.*

   Assume that someone has made this atomic instruction available to high level languages such as C with the following library function:

   ```
   int invert(float * X)
   ```

   If you call this function, it will invert the passed floating point value pointed to by X. The return value of the call is the overflow bit: one if overflow occurs and zero if not. Show how to solve the critical section problem by providing CSEnter and CSEXit using INV.

   ```
   X = 0.0;

   csenter(float * X) {
    while (invert(X) == 0) ; /* busy wait */
   }


   csexit(float * X) {
    *X = 0.0;
   }
   ```

6. (10 points) The following questions concern the Linux O(1) scheduler:

   i. (2 points): What is the run queue? How is it different from the task list?

   The run queue is the list of all tasks that are able to run (with state `TASK_RUNNING`). The task list is all tasks in the system.

   ii. (2 points): How many real-time priorities are there? How many non real-time?

   Linux has 100 real-time priorities and 40 non real-time.

   iii. (3 points): Linux give processes with higher priority larger time slices. Why?

   The scheduling philosophy is that more important tasks get more opportunity to run. Thus higher priority tasks get larger time slices.

   iv. (3 points): Linux dynamically adjusts the priority of tasks based on sleep time. Why?

   Linux uses sleep time as an indication of interactivity. If a task is sleeping most of the time, the assumption is that it is probably waiting for the user (a mouse click or keyboard input). Linux tries to improve responsiveness to the user by boosting the priority of tasks it thinks are interactive.

7. (5 points) Linux passes system call arguments in registers, rather than on the stack like a regular program. Why?

   Full credit: Linux uses a separate kernel stack when in protected mode. When making a system call, the system is transitioning between the user mode stack and the kernel mode stack. Using 2 stacks is awkward, so here Linux doesn't use any.

   Partial credit: The kernel cannot trust the user mode stack, due to potentially malicious or buggy user code that may place bad values on the stack.

   Partial credit: To be consistent with how other interrupts, traps, and exceptions are handled.