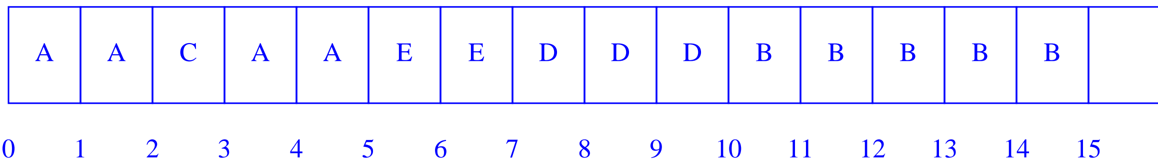
# COMS W4118: Operating Systems — March 2006

## Questions

**1.** (15 points) The following table shows the arrival times and run-times of several jobs. Draw a Gantt diagram showing the scheduling for shortest remaining time first. What is the average turnaround time from arrival to completion?

| | *Arrival* | *Run-time* |
|---|---|---|
| *A* | 0 | 4 |
| *B* | 2 | 5 |
| *C* | 2 | 1 |
| *D* | 3 | 3 |
| *E* | 5 | 2 |

Write your answer below, **not in the blue book**:




This chart show the run-time left for each process at the start of each time-tick; the * shows which process is going to run in that slot.
A arrives at $T = 0$ and finishes at $T = 5$, and hence took 5 ticks. B arrived at $T = 2$ and finished at $T = 15$ for 13 ticks. C was in the system for 1 tick, D for 7, and E for 2. The average is thus 28/5, or 5.6 ticks.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | 4* | | | | |
| 1 | 3* | | | | |
| 2 | 2 | 5 | 1* | | |
| 3 | 2* | 5 | - | 3 | |
| 4 | 1* | 5 | - | 3 | |
| 5 | - | 5 | - | 3 | 2* |
| 6 | - | 5 | - | 3 | 1* |
| 7 | - | 5 | - | 3* | - |
| 8 | - | 5 | - | 2* | - |
| 9 | - | 5 | - | 1* | - |
| 10 | - | 5* | - | - | - |
| 11 | - | 4* | - | - | - |
| 12 | - | 3* | - | - | - |
| 13 | - | 2* | - | - | - |
| 14 | - | 1* | - | - | - |
| 15 | - | - | | | |

**2.** (10 points) There are two principle uses for semaphores. What are they, and when you use each one?

Semaphores can be used for simple mutual exclusion or to count availability of some resource. The former, often called a "mutex", is by far the most common; it's used to regulate access to some resource. The latter is used for things like counting buffer slots in the producer/consumer problem.

Put another way, the second form of semaphore provides a way to block awaiting a resource that isn't as easily modeled as a simple mutex.

**3.** (10 points) Interrupt routines use both interrupt masking and spin locks. When should each be used?

Interrupt masking prevents dual access to resources on a single CPU. Spin locks are useful for contention on multiprocessors.

Note that you can't easily use a spin lock properly on a uniprocessor (unless you're using threading and have some way to surrender control). The main flow can grab the lock; suppose the interrupt occurs right after it does. How does the interrupt routine wait? It could be done with one of the deferred work mechanisms, but those are relatively heavy-wieght; spinlocks and interrupt masking are light-weight.

The usual sequence is to mask interrupts (to deal with contention on this CPU, grab a spin lock (if on a multiprocessor), do the work, release the spinlock, and unmask interrupts.

**4.** (10 points) Interrupt handlers 'have no process context". What does this mean, why don't interrupt handlers have one, and what are the implications?

Interrupts are asynchronous to any process, and are generally not related to whatever process is running at the time they occur. In fact, for some interrupts – say, a timer interrupt or an unsolicited (and possibly unwanted) network packet, there is *no* process that can be associated with it, even in principle.

Because of this, interrupt handlers can't do certain things. They can't check if the current process is running as root, nor can they sleep — after all, they'd be looking at the wrong process' UID or putting the wrong process to sleep.

**5.** (10 points) It's possible to detect and recover from deadlocks. Why is it so much harder (or impossible) to detect starvation?

Locking is generally done in the kernel, with the kernel aware of the locks; it can thus detect deadlocks, using the algorithms we've discussed in class. With starvation, the applications are running; they're not making progress (and hence wait again almost immediately), but the kernel can't know that that's what's happening.

**6.** (10 points) Why is priority aging needed?

Without priority aging (or some similar mechanism), lower-priority tasks will never run if the arrival rate of higher-priority tasks is high enough.

**7.** (15 points)

a. (5 points) Explain why reentrant subroutines are needed in a threaded environment

Two threads may need to call the same routine; if one is using a static variable when it blocks, that variable can be overwritten by the second thread.

b. (10 points) Not all subroutines need to be reentrant. If you're writing a subroutine library, how can you distinguish between the cases?

Threading is non-preemmptive; only routines that can block need to be reentrant. Note, though, that with kernel threading, any thread might block because of a virtual memory issue, so the larger question is whether or not the library routine uses any static variables.

**8.** (20 points) On some system, a CPU-bound process has been running for a long time. A network I/O interrupt occurs, marking the arrival of a packet that a high-priority process has been blocked while waiting for. Explain, in reasonable detail, all of the events that will occur starting from the instant of the interrupt until the machine starts executing something in user mode again. Note: I'm not interested in, say, Linux routine names; I'm interested in functional components and what they do.

When the interrupt occurs, the hardware traps to the kernel. The *first-level interrupt handler* is invoked. It saves registers and CPU state in some area private to the current process. It then invokes the *second-level interrupt handler* which does the actual interrupt servicing. At some point in this process, it notices that the high-priority process was waiting on a packet, so it *marks the process runnable* and passes control to the *scheduler*. The scheduler checks to see what the highest-priority runnable process is, and invokes it. It does this by *saving the current process' state* — on Linux, that's the stack pointer – and *loading the registers and state for the new process*. That's run in the kernel, where it *checks if the wait condition is satisfied*. If so, it issues a *return from interrupt* instruction to run that process at user level.