

April 19, 2007, COSC 416 Exam 2 Post-mortem

1. a. F. Pp 145–146 Java threads can map to kernel threads. And in fact must, even if only many-to-one, since the whole JVM is a process, hence a thread. Even if you were to simulate multi-threading in software, which Peterson’s solution can do on a uniprocessor, it would run on at least one kernel thread of the host.
- b. F. Java threads may instead implement Runnable, thereby allowing the thread to inherit from another class, because of the Java rule “only one parent class.”
- d. F. You need not execute sleep(); you may execute yield() instead.
- e. T. We synchronize against a specific lock, either explicitly as in synchronized (myObject) or implicitly when a method of a class is synchronized, implied against the lock of the object that is created of that class, so like a synchronized(this).
- f. T or F. I allowed either answer. That Java supports thread priorities like Thread.MAX_PRIORITY makes this clearly true at a higher level of abstraction. See Slide 40 in the PowerPoint for Chapter 5, linked from course web page, ch05new.ppt. But in the absence of explicit thread priorities, different JVMs may schedule threads in any of the following ways: FCFS, RR, SJF, SRTF, two of which (the underlined) are not priority-based. That’s at the lowest level of abstraction of a JVM.
- g. F. They are also called acquire and release, but that doesn’t make the given names false. I gave credit if you corrected to the underlined terms. See 1(j) below for a similar distinction.
- h. T. I said this explicitly in class. You may argue that it’s always bad.. But remember that you are not disabling throughout the critical region, only for the two machine language commands necessary to simulate an atomic test-and-set. Those who said F but reminded us that we are assuming a uniprocessor system got credit, since disabling interrupts on a multiprocessor system is not a good idea.
- j. T or F. I allowed either answer. I intended the answer to be true. But the book calls getAndSet what I called TestAndSet. If you look at the code for getAndSet on p. 216, you’ll see that it returns a boolean, so it tests whether a boolean is true or false by getting it! Other texts use testAndSet instead of getAndSet for the terminology. I didn’t realize that some of you would answer based on the names rather than on the concepts. As with Question g above, I wasn’t nit-picking names, I was testing the concepts behind the names, whatever they happened to be called. Watch the typeface now: Had I used “test and set” and “swap” they would have clearly been semantic. Had I used testAndSet(boolean b) and swap(boolean a, boolean b) they would have clearly been syntactic. But I used a typeface convention half-way between them.
- k. F. P. 180, Fig. 5.5
- l. T. P. 192, Fig. 5.12

2. A good answer would say why i/o does not need to consume the CPU. For example, it may use DMA, or it may be so slow that even if it does use the CPU, it doesn’t do that very often at the timescale at which a CPU executes. To say simply “They can alternate” doesn’t explain throughput, merely restates the question..The real issue is that the CPU can be kept busy while i/o is completing, in overlapping fashion..

3. I was looking for the notion of whether the processes **communicate** directly **with each other** in a way that provides information from one proc to another, to help the other make a decision about what to do next. Cooperative procs do that. Competitive (i.e. non-cooperative) ones instead fly blind, and so benefit from being maximally greedy. Ben and Josh mentioned as their example that preempting is a kind of cooperation. But this is subtle. If the preemption is based on something in the procs (like setting a priority), then that’s a good example. If the preemption is based on the operating system just throwing a proc out ready-or-not, then the procs are certainly not cooperating.

A clearer example would be: Competing procs don’t yield(). Don’t say that non-cooperative procs take all the (non-CPU) resources that they need. That’s actually a form of cooperation to guarantee no deadlock. It would be non-cooperative to take a few now and a few later without regard to whether other procs need resources. You can say that (non-CPU) resources are held inefficiently in the competitive case.

Also, even non-cooperative procs can and do share data. All three classic synchronization problems (bounded buffer, dining philosophers, and readers-writers) are non-cooperative if their solutions do not involve `yield()`.

6. The issue is not between actual and theoretical data, but recent actual data and long-term history of actual data started with one theoretical value. That is to say, tau can be expanded in terms only of t's, except for tau zero, which must be theoretically supplied. For alpha = 0.8 we don't learn much from history; for alpha = 0.2 we don't much trust our wild current guess of shortest job. I worked really hard in creating this question to avoid giving it away by using the words "past history." Good answers, however, did use the words. Think of the taus as giving better and better theoretical models (hence my choice of term) because they learn from history. I especially liked those answers that said that alpha = 0.2 gives a more gently changing graph.

7. **Indefinite postponement:** Selecting a proc to **restart**-from-ready that is **in** its critical section must happen promptly, and must be based only on the ready procs that are **in** their critical sections. **Bounded waiting:** If a **running** proc **not in** its critical section attempts to enter its critical section, there is a maximum number of other procs **not in** their critical sections also wanting to attempt to enter their critical sections that can go ahead of it. (I think I made this clearer than I did in lecture and than the text does, so I was lenient in deducting points on this question.)

8.

b. Volatile variables are potentially changed by more than one thread, hence need to be updated promptly.