

# CS167 Homework Assignment 1

*Due: 11:59pm, September 29, 2008*

1. The text, on page 3-10, describes how to switch from one thread to another. The implicit assumption is that the thread being switched to in the *switch* routine has sometime earlier yielded the processor by calling *switch* itself. Suppose, however, that this thread is newly created and is being run for the first time. Thus when its creator calls *switch* to enter the new thread's context, the new thread should start execution as if its first routine had just been called. Show what the initial contents of its stack should be to make this happen. Assume an x86-like assembler language, as used in Chapter 3 of the text.
2. We've seen that Unix uses file-descriptor tables, system open-file tables, and active inode tables. Suppose we have multithreaded processes on a multiprocessor (with a preemptible kernel — this means that even kernel threads may be forced to yield to other runnable threads at any time). Consider the system calls *open*, *close*, *dup*, *read*, and *write*. Describe how mutexes and other synchronization constructs would be used in these tables with these system calls so as to achieve maximal parallelism. Be sure to indicate for each of the system calls which mutexes it would use and for how long it would hold them.
3. Slides VI-5 and VI-6 (from pages 3-23 and 3-24 of the text) show the assembler code of *main.s* and *subr.s*. Describe what changes are made to the corresponding machine code (in *main.o* and *subr.o*) when these routines are processed by *ld*, forming *prog*. Be specific. I.e., don't say "the address of X goes in location Y," but say, for example, "the value 11346 goes in the four-byte field starting at offset 21 in xyz.o." Note that for the x86 instructions used in this example, an address used in an instruction appears in the four-byte field starting with the second byte of the instruction.
4. The following code is an alternative to the implementation of mutexes in slide VIII-21. Does it work? Explain why or why not.

```
kmutex_lock(mutex_t *mut) {
    if (mut->locked)
        sleep_on(mut->wait_queue);
    mut->locked = 1;
}

kmutex_unlock(mutex_t *mut) {
    mut->locked = 0;
    wakeup_on(mut->wait_queue);
}
```