Welcome to CS162!

About your TA

Name:	Angela Juang	
Email:	ajuang@berkeley.edu (Put [CS 162] in the subject line)	
Website:	http://inst.eecs.berkeley.edu/~cs162-ta	
Sections:	Tuesday	10-11am, 320 Soda
		11-12pm, 320 Soda (may be changed)
Office Hours:	Thursday	10-11am, 651 Soda
	Friday	2-3pm, 651 Soda

Announcements

- Section signup is still not finalized…look out for an announcement in tomorrow's lecture!
- We will likely move one of the 11-12pm sections to another time, since it's less popular
- Project 1 starts next week, 09/15
    - Specs aren't up yet – keep an eye out for updates
    - Familiarize yourself with Java Threads (`java.lang.Thread`)

Don't take this out of context…

"Context switch" is a general term to describe the process of saving and restoring CPU state so that streams of execution can be resumed later in time. There are two main types of context switches: hardware context switches and software context switches. Hardware context switches, also called interrupts, are caused by a signal from hardware to the kernel indicating that some event (keystroke, mouse, timer, etc.) has occurred. When a hardware interrupt happens, execution goes directly to the interrupt handler. In contrast, a software context switch occurs when a thread yields execution to another thread. In this case, the kernel is responsible for choosing the next thread to run.

Each thread has its own execution stack; a context switch from one thread to another will not affect the previous thread's stack outside of taking the necessary steps to yield. This allows threads to resume execution where they left off the next time they get CPU time. But how does each thread know where in the its execution to resume? Each thread has a stack pointer that is located at the bottom of its stack (remember, the stack grows down!). This stack pointer is saved and restored in the TCB during a context switch, and points to the bottom of a set of stack frames that have been built up over the course of the thread's execution so far. A stack frame is created every time a function is called, and stores parameters to the function, local variables, and a return address (the address to which control should return once the function exits).

When a context switch happens, the stack pointer, program counter, and registers are replaced with the corresponding values for the next thread to execute. Assuming this thread had stopped executing because it had performed a context switch at some earlier point in time, this allows the thread to "return" out of the context switch and continue execution from where it left off.

CS 162 – Fall 2011 TA: Angela Juang
Discussion Notes Week #02

A Tale of Two Threads

Last week we introduced the concept of processes and threads, and the fact that the kernel is in charge of scheduling threads (allocating CPU time to each thread). The threads that the kernel knows about and schedules are called *kernel threads*; each process has at least one kernel thread associated with it. However, user-level processes can be multithreaded as well by creating *user threads*. User threads are managed by a user threading library such as Java, POSIX, etc., and are not directly scheduled by the kernel. All user threads must be associated with corresponding kernel threads so the kernel can assign them to the CPU. There are several ways to do this:

- *One-to-one.* Each user thread is associated with exactly one kernel thread. In this case, all scheduling is done by the kernel. The advantage of this approach is that all threads can take advantage of multiple cores, since any thread can be assigned to any open CPU. However, it also means that all context switches are now kernel thread context switches, which are more expensive than user thread context switches because they require switching between user and kernel mode.
- *One-to-many.* One kernel thread has many user threads. In this case, most or all scheduling is done completely at user level, which is less expensive than kernel level scheduling. However, from the kernel's point of view, only one thread is running. This means that if one of the user threads blocks on a syscall (for example, I/O), then none of the other threads will get a chance to run until the syscall is finished.
- *Many-to-many.* Several kernel threads are associated with several user threads. With this system, we can avoid some of the shortcomings of the above two options. However, this kind of association makes scheduling much more complicated and involves some kind of coordination between user and kernel level schedulers.