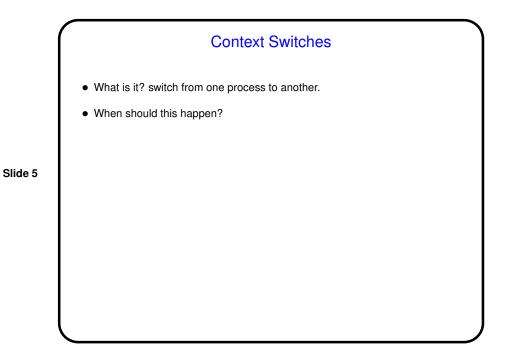


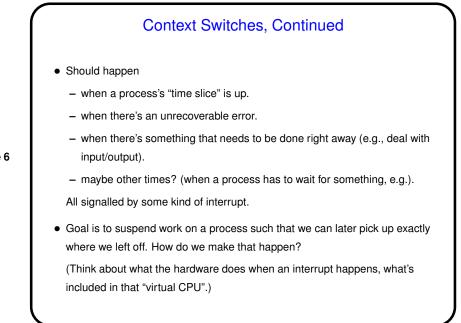
Process Abstraction, Continued

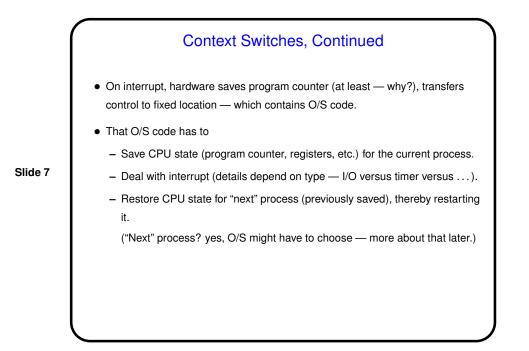
 Can also associate with process an "address space" — range of addresses the program can use. Simplifying a little, this is "virtual memory" (like the virtual CPU) that only this process can use. More (lots more) about this later. (Nitpick: Yes, we also want to be able to share memory among processes. More about that later too.)

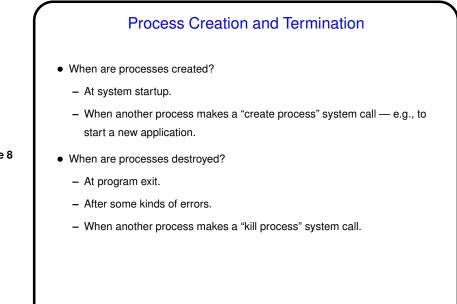
Slide 4

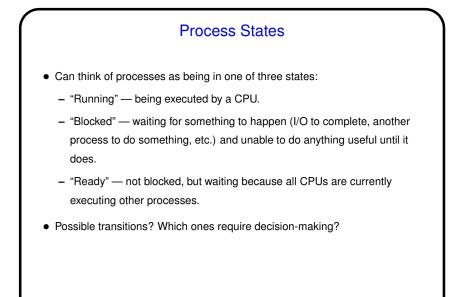
 How to map this to the real hardware? Chapter 2 talks about how to share the real CPU(s) among processes; chapter 3 talks about how to share the real memory.







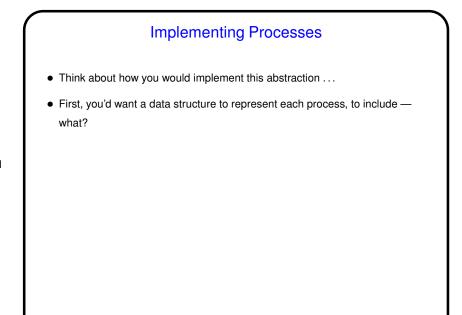


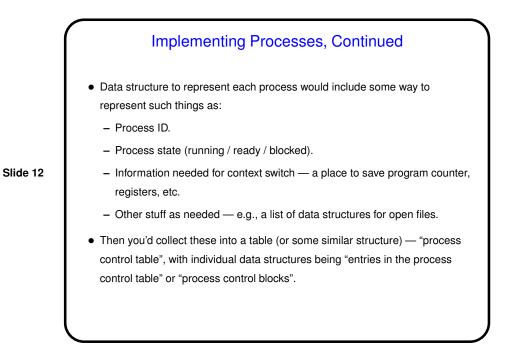


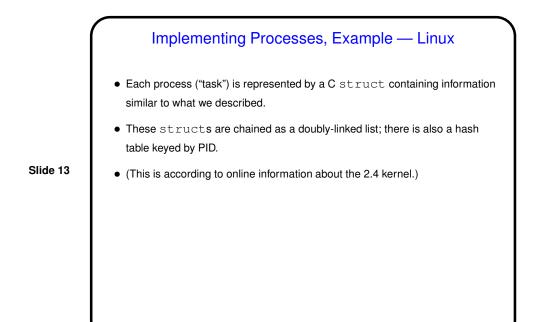
Process States, Continued
Possible transitions (Figure 2-2):

Running to blocked — happens when, e.g., a process makes an I/O request and can't continue until it's complete.
Blocked to ready — happens when the event the blocked process is waiting for occurs.
Running to ready, ready to running — needed if we want some sort of time-sharing (give all non-blocked processes "a turn" frequently).

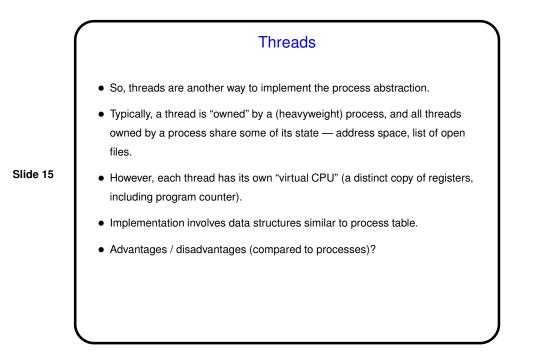
Notice that moving to and from "blocked" state doesn't involve decision-making, but ready/running transitions do.
The decision-maker — "scheduler" (to be discussed later). Often "running to ready" is triggered by an interrupt (I/O, timer, etc.), and "ready to running" involves this scheduler.

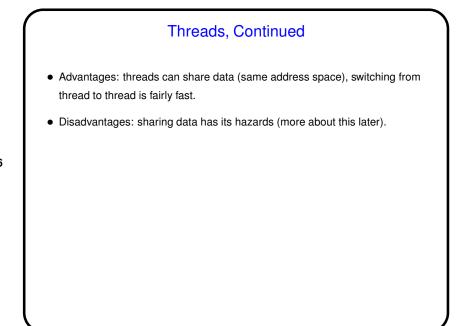


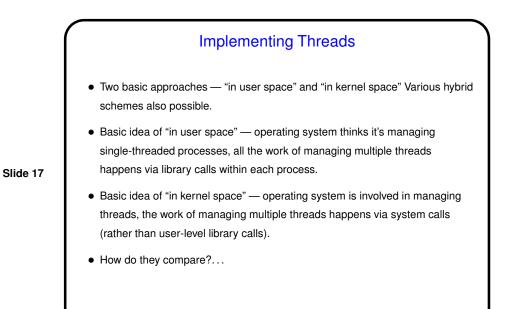




Processes Versus Threads
So far I've used "process" in an abstract/general way.
In typical implementations, though, "process" is more specific — something that has its own address space, list of open files, etc. Often these are called "heavyweight processes".
Advantages — such processes don't interfere with each other.
Disadvantages — they can't easily share data, switching between them is expensive ("a lot of state" to save/restore).
For some applications, might be nice to have something that implements the abstract process idea but allows sharing data and faster context switching — "threads".



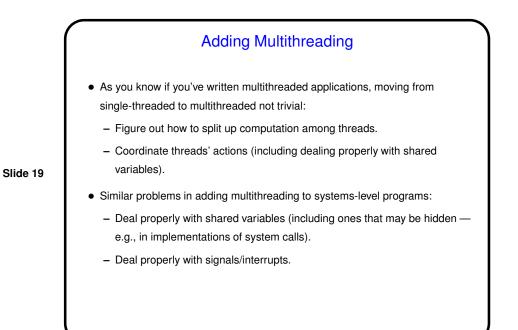


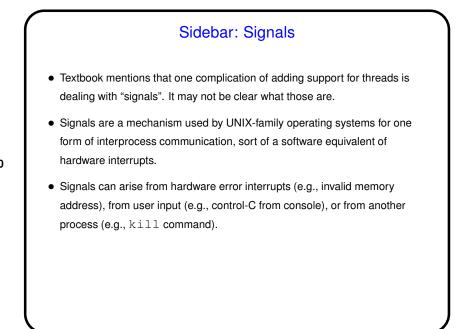


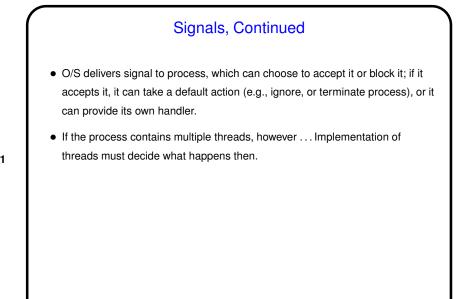
Implementing Threads, Continued
Implementing in user space is likely more efficient — fewer system calls, so less overhead.
Implementing in kernel space avoids some problems, though:

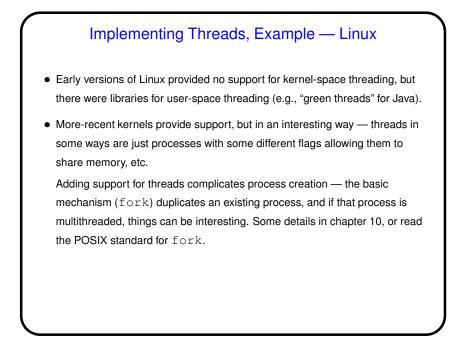
If a thread blocks, it may do so in a way that blocks the whole process.
Preemptive multitasking is difficult/impossible without help from the kernel, as is using multiple CPUs.

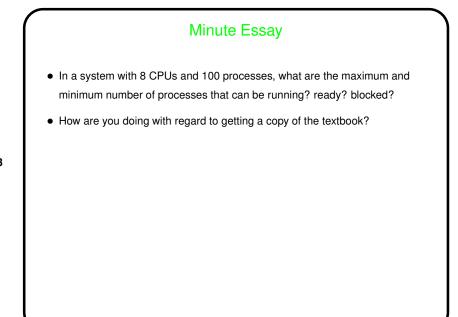
Slide 18











Minute Essay Answer Blocked: Maximum of 100 (unless you assume that there's an "idle" operating system process that runs when nothing else does and never blocks, and maybe one of these is needed for every CPU). Minimum of 0. Running: Maximum of 8, because there are 8 CPUs. Minimum of 0 (again unless you assume that there's an O/S process that runs when nothing else does). Ready: Maximum of 92, since all CPUs will be running processes if there are any that can be run. (Depending on details, you might have to add "except during context switches, when the scheduler is choosing the next process to run on a CPU".) Minimum of 0, since they could all be blocked or running.