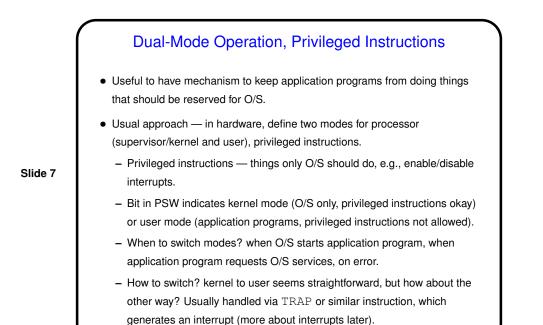
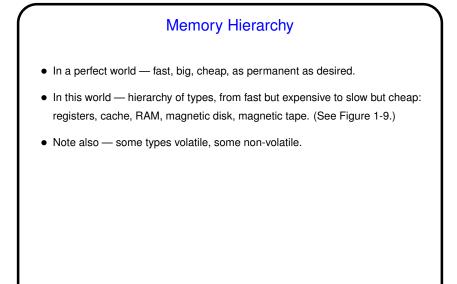


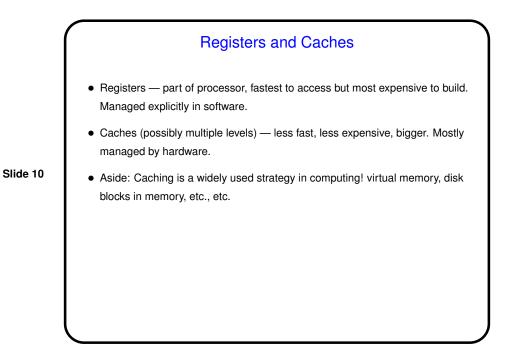
Processors, Continued

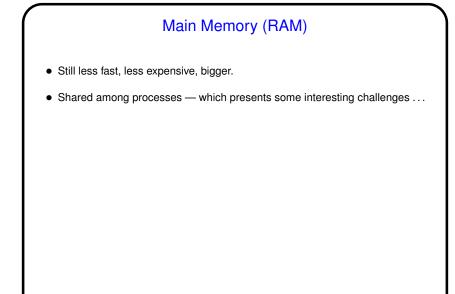


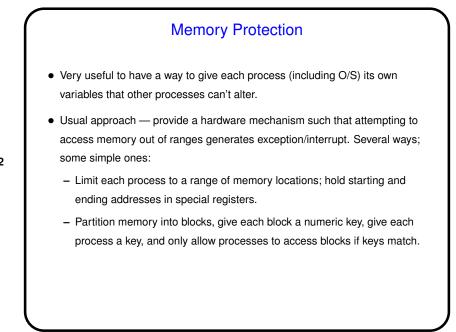
Multithreaded and Multicore Chips

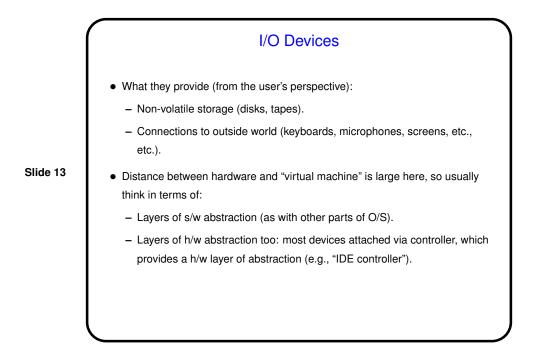
- For many years (at least 30, to my knowledge) advocates of parallel programming have been saying that eventually hardware designers would run out of ways to make single processors faster — and finally it seems to be happening.
- Basic idea number of transistors one can put on a chip kept increasing, and for a long time hardware designers used that to make single processors faster (e.g., with longer pipelines). But then they apparently ran out of ideas. So, instead, they chose to provide (more) hardware support for parallelism. Various approaches, including "hyperthreading" (fast switching among threads), "multicore" (multiple independent CPUs, possibly sharing cache), "GPGPU" (use of graphic card's many processors for computation).

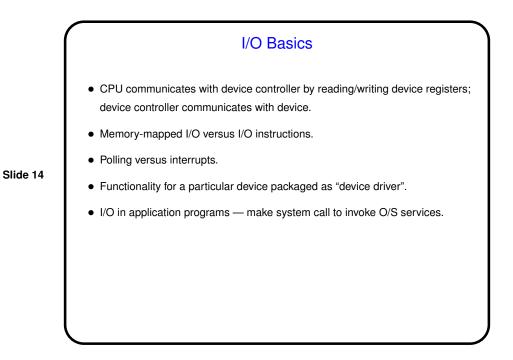


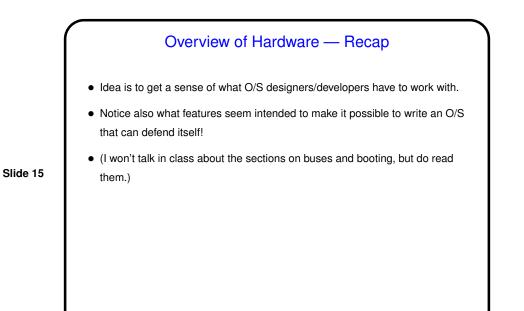




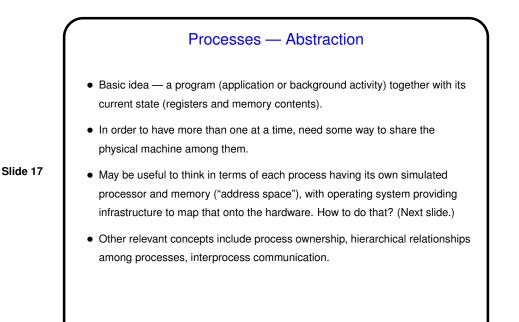






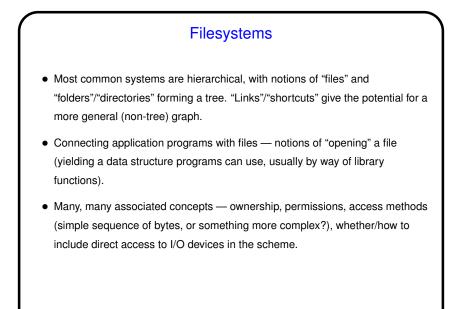


Operating System Functionality — Review
"Operating system as virtual machine" must provide key abstractions (processes, filesystems).
"Operating system as resource manager" must manage resources (memory, I/O devices, etc.).
Operating system functionality typically packaged as "system calls" (more later).
Details obviously vary among systems, but some ideas are common to most/many (more later).



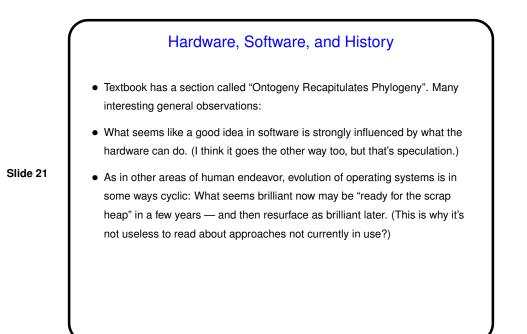
## Processes — Implementation

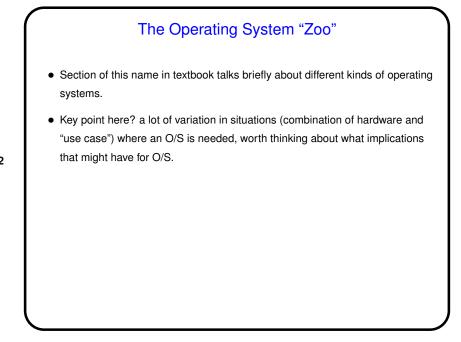
- Managing the "simulated processor" aspect requires some way to timeshare physical processor(s). Typically do that by defining a per-process data structure that can save information about process. Collection of these is a "process table", and each one is a "process table entry".
- Managing the "address space" aspect requires some way to partition physical memory among processes. To get a system that can defend itself (and keep applications from stepping on each other), memory protection is needed probably via hardware assist. Some notion of address translation may also be useful, as may a mechanism for using RAM as a cache for the most active parts of address space, with other parts kept on disk.

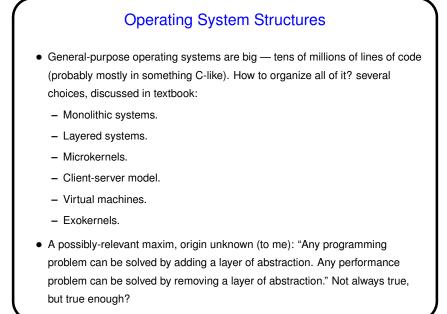


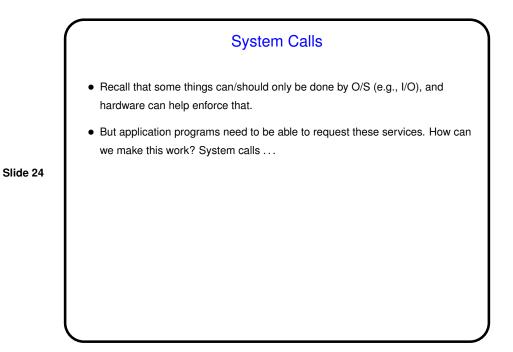
I/O
As noted previously — hardware is diverse, and communicating with it may involve a lot of messy details.
So — typically there is an "I/O subsystem", often involving multiple layers of abstraction. More later!

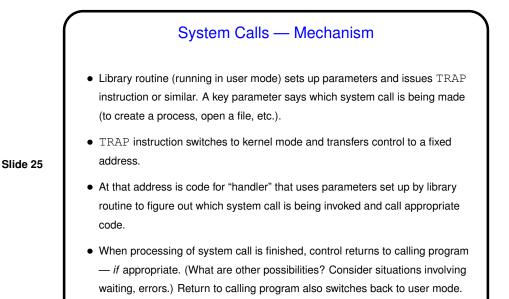
Slide 20











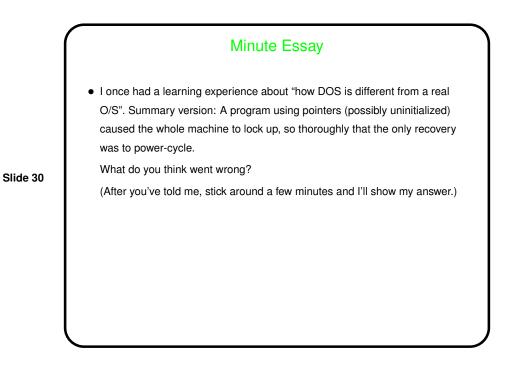
## System Calls — Services Provided Typical services provided include creating processes, creating files and directories, etc., etc. — details depend on (and in some ways define, from application programmer's perspective) operating system. Examples discussed in textbook: POSIX (Portable Operating System Interface (for UNIX)) — about 100 calls. Win32 API (Windows 32-bit Application Program Interface) — thousands of calls. Worth noting that the actual number of system calls is likely smaller — interface may contain function calls that are implemented completely in user space (no TRAP to kernel space).

Interrupts
Processing of TRAP instructions is similar to interrupts, so worth mentioning here:
Very useful to have a way to interrupt current processing when an unexpected or don't-know-when event happens — error occurs (e.g., invalid operation), I/O operation completes.
On interrupt, goal is to save enough of current state to allow us to restart current activity later:

Save old value of program counter.
Disable interrupts.
Transfer control to fixed location ("interrupt handler" or "interrupt vector") — normally O/S code that saves other registers, re-enables interrupts, decides what to do next, etc.

Example: System Calls in MIPS
MIPS instruction set includes syscall instruction that generate a system-call exception. MIPS interrupts/exceptions use special-purpose registers to hold type of exception and address of instruction causing exception. Before issuing syscall program puts value indicating which service it wants in register \$v0. Parameters for system call are in other registers (can be different ones for different calls).
Interrupt handler for system calls looks at \$v0 to figure out what service is requested, other registers for other parameters.
When done, it uses rfe instruction to restore calling program's environment, then returns to caller using value from EPC register.

```
Example: System Calls in MIPS/SPIM
• SPIM simulator — a primitive O/S! — defines a short list of system calls.
Example code fragment:
    la $a0, hello
    li $v0, 4 # "print string" syscall
    syscall
    ....
    .data
hello: .asciiz "hello, world!\n";
```



## Minute Essay Answer

The program changed memory at the addresses pointed to by the uninitialized pointers — and this memory was being used by the O/S, possibly to store something related to interrupt handling. A "real" O/S wouldn't allow this!
 (Then again, the version of MS-DOS in question was supposedly written to run on hardware that didn't provide memory protection, so maybe it's not DOS's fault.)