

Minute Essay From Last Lecture

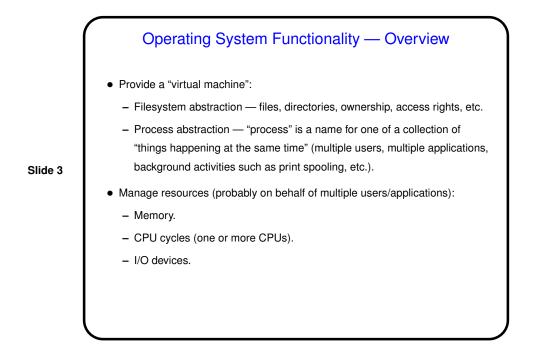
• Not everyone had exposure to anything very old/different.

Also a mention of iPhone 5 as being difficult to develop for!

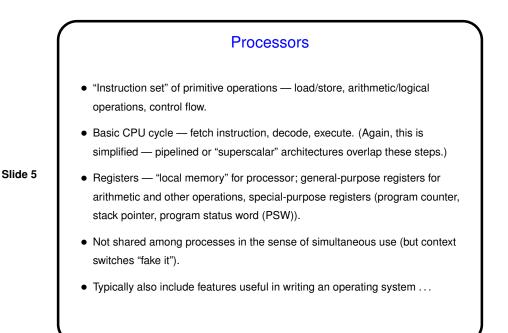
• But - Compaq with MS-DOS, custom OS on a friend's laptop (next bullet), Commodore 64 (two people!), multiple layers of virtual stuff. Also Windows ME and Windows 95.

Slide 2

• (There was a time when each computer hardware company had its own O/S!)



Overview of Hardware Simplified view of hardware (as it appears to programmers) — processor(s), memory, I/O devices, bus. Figure 1.6 shows simplified view of overall organization — components connected to a single bus. (Actual processors may have more than one bus.)



Dual-Mode Operation, Privileged Instructions

- Useful to have mechanism to keep application programs from doing things that should be reserved for O/S.
- Usual approach in hardware, define two modes for processor (supervisor/kernel and user), privileged instructions.
 - Privileged instructions things only O/S should do, e.g., enable/disable interrupts.
 - Bit in PSW indicates kernel mode (O/S only, privileged instructions okay) or user mode (application programs, privileged instructions not allowed).
 - When to switch modes? when O/S starts application program, when application program requests O/S services, on error.
 - How to switch? kernel to user seems straightforward, but how about the other way? Usually handled via TRAP or similar instruction, which generates an interrupt (more about interrupts later).

Multithreaded and Multicore Chips

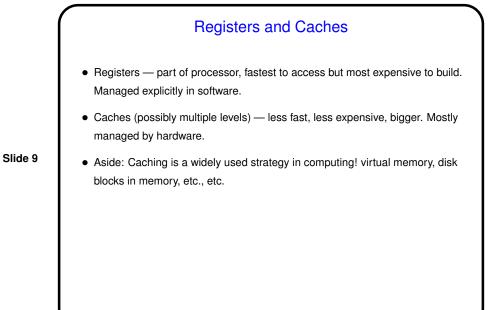
 For many years (at least 30, to my knowledge) advocates of parallel programming have been saying that eventually hardware designers would run out of ways to make single processors faster — and finally it seems to be happening.

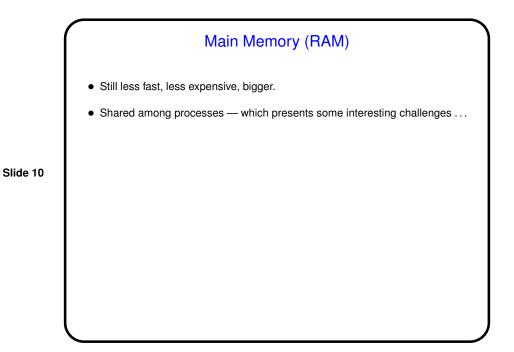
Slide 7

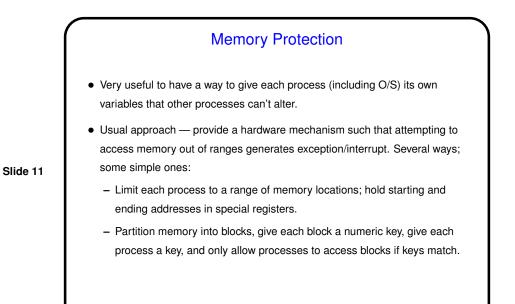
 Basic idea — number of transistors one can put on a chip kept increasing, and for a long time hardware designers used that to make single processors faster (e.g., with longer pipelines). But then they apparently ran out of ideas. So, instead, they chose to provide (more) hardware support for parallelism. 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).

Memory Hierarchy

- In a perfect world fast, big, cheap, as permanent as desired.
- In this world hierarchy of types, from fast but expensive to slow but cheap: registers, cache, RAM, magnetic disk, magnetic tape. (See Figure 1-9.)
- Note also some types volatile, some non-volatile.





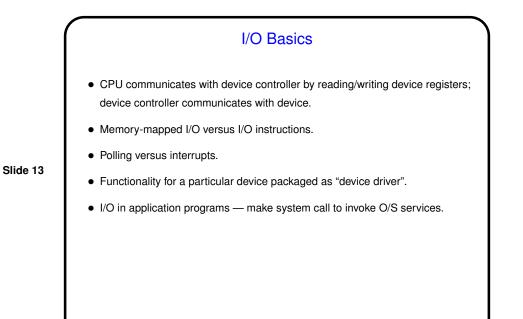


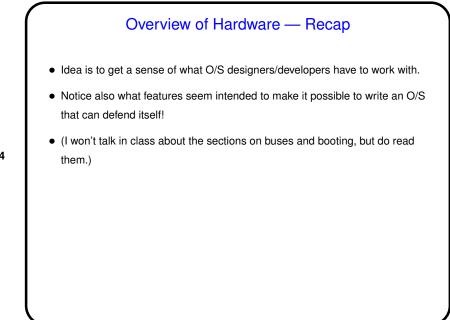
I/O Devices
What they provide (from the user's perspective):

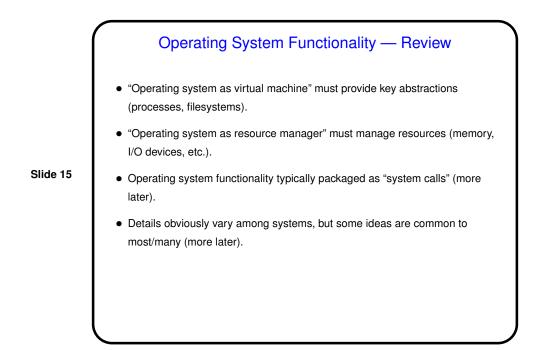
Non-volatile storage (disks, tapes).
Connections to outside world (keyboards, microphones, screens, etc., etc.).

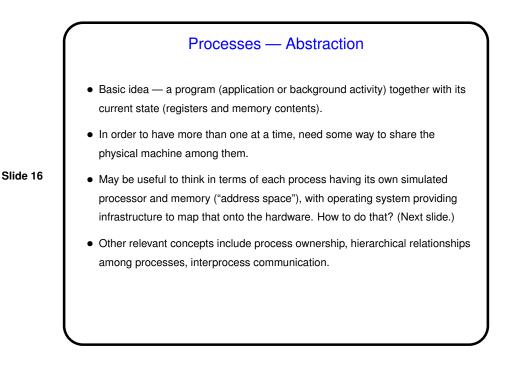
Distance between hardware and "virtual machine" is large here, so usually think in terms of:

Layers of s/w abstraction (as with other parts of O/S).
Layers of h/w abstraction too: most devices attached via controller, which provides a h/w layer of abstraction (e.g., "IDE controller").









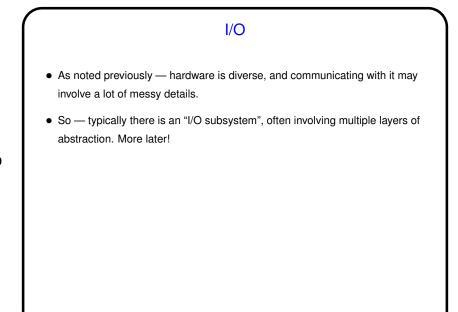
Processes — Implementation

 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".

Slide 17

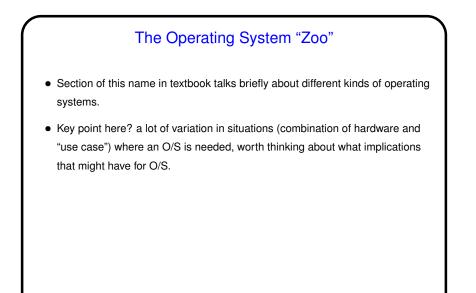
 Managing the "address space" aspect requires some way to partition physical memory among processes. To get a system that can defend itself (and keep applications from stepping on each other), memory protection is needed probably via hardware assist. Some notion of address translation may also be useful, as may a mechanism for using RAM as a cache for the most active parts of address space, with other parts kept on disk.

	Filesystems
Slide 18	 Most common systems are hierarchical, with notions of "files" and "folders"/"directories" forming a tree. "Links"/"shortcuts" give the potential for a more general (non-tree) graph.
	 Connecting application programs with files — notions of "opening" a file (yielding a data structure programs can use, usually by way of library functions).
	 Many, many associated concepts — ownership, permissions, access methods (simple sequence of bytes, or something more complex?), whether/how to include direct access to I/O devices in the scheme.
	 Relevant system calls — create file, create directory, remove file, open, close, etc., etc.

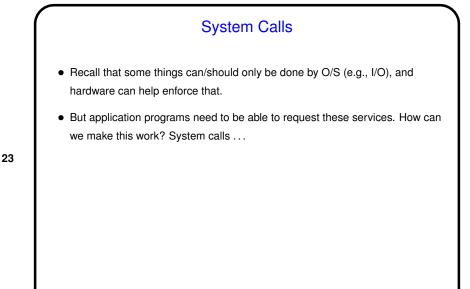


Slide 20

Hardware, Software, and History • Textbook has a section called "Ontogeny Recapitulates Phylogeny". Many interesting general observations: • What seems like a good idea in software is strongly influenced by what the hardware can do. (I think it goes the other way too, but that's speculation.) • As in other areas of human endeavor, evolution of operating systems is in some ways cyclic: What seems brilliant now may be "ready for the scrap heap" in a few years — and then resurface as brilliant later. (This is why it's not useless to read about approaches not currently in use?)

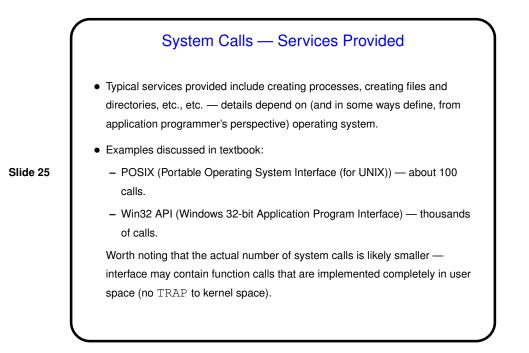


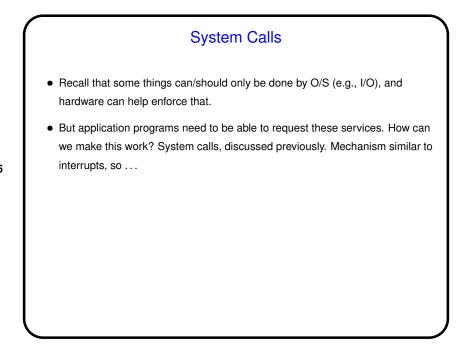
Operating System Structures • General-purpose operating systems are big — tens of millions of lines of code (probably mostly in something C-like). How to organize all of it? several choices, discussed in textbook: - Monolithic systems. - Layered systems. - Microkernels. Client-server model. _ Virtual machines. _ - Exokernels. • A possibly-relevant maxim, origin unknown (to me): "Any programming problem can be solved by adding a layer of abstraction. Any performance problem can be solved by removing a layer of abstraction." Not always true, but true enough?



Slide 24

System Calls — Mechanism Library routine (running in user mode) sets up parameters and issues TRAP instruction or similar. A key parameter says which system call is being made (to create a process, open a file, etc.). TRAP instruction switches to kernel mode and transfers control to a fixed address. At that address is code for "handler" that uses parameters set up by library routine to figure out which system call is being invoked and call appropriate code. When processing of system call is finished, control returns to calling program — *if* appropriate. (What are other possibilities? Consider situations involving waiting, errors.) Return to calling program also switches back to user mode.



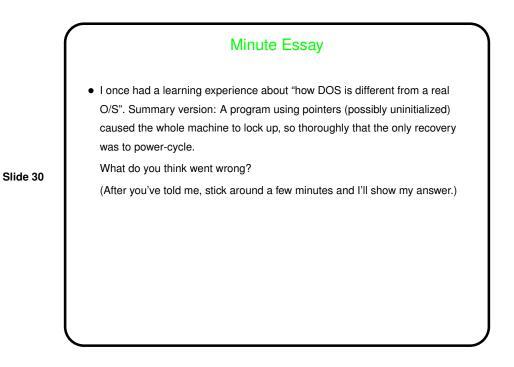


Interrupts
Processing of TRAP instructions is similar to interrupts, so worth mentioning here:
Very useful to have a way to interrupt current processing when an unexpected or don't-know-when event happens — error occurs (e.g., invalid operation), I/O operation completes.
On interrupt, goal is to save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