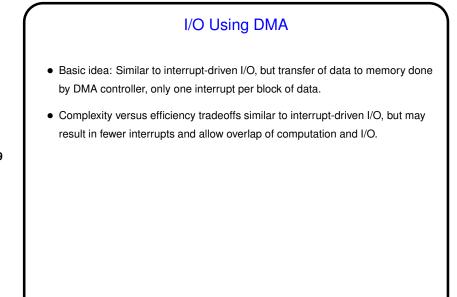
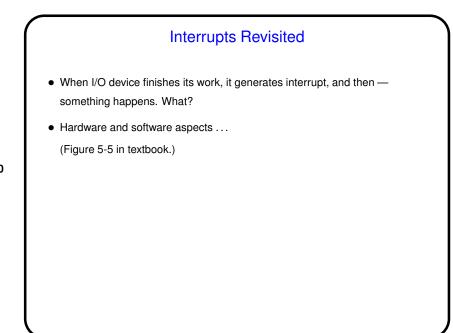
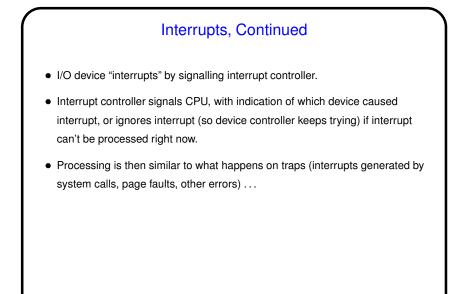


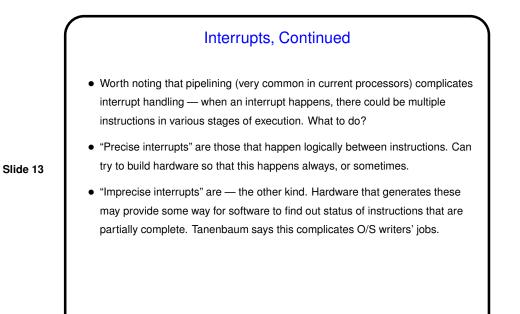
Interrupt-Driven I/O Basic idea: Program tells controller what to do and then blocks. While it's blocked, other processes run. When requested operation is done, controller generates interrupt. Interrupt handler unblocks original program (which, on a read operation, would then obtain data from device controller). Slide 8 More complex, but allows other processing to happen while waiting, so potentially more efficient for system as a whole. Could, however, result in lots of interrupts. (Tanenbaum says one per character/byte. Can that be true for disks?? Open question ...)



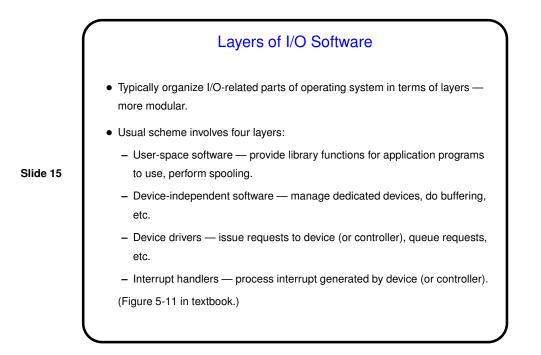


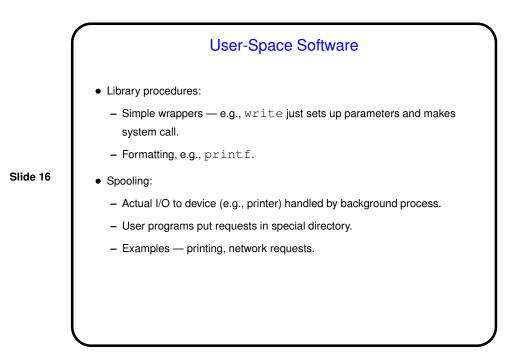


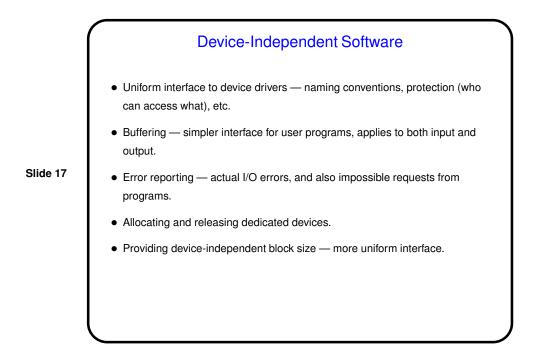
Interrupts, Continued On interrupt, hardware locates proper interrupt handler (probably using interrupt vector), saves critical info such as program counter, and transfers control (switching into supervisor/kernel mode). Interrupt handler saves other info needed to restart interrupted process, tells interrupt controller when another interrupt can be handled, and performs minimal processing of interrupt.

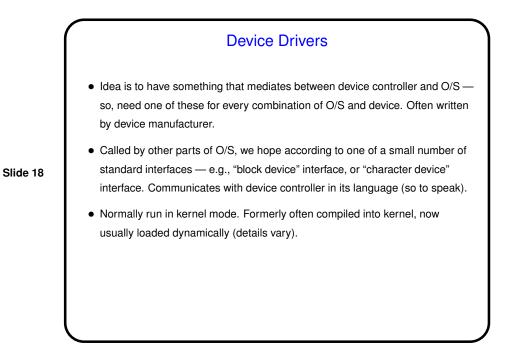


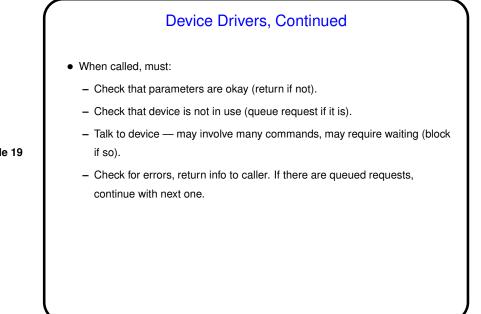
Goals of I/O Software
Device independence — application programs shouldn't need to know what kind of device.
Uniform naming — conventions that apply to all devices (e.g., UNIX path names, Windows drive letter and path name).
Slide 14
Error handling — handle errors at as low a level as possible, retry/correct if possible.
"Synchronous interface to asynchronous operations."
Buffering.
Device sharing / dedication.

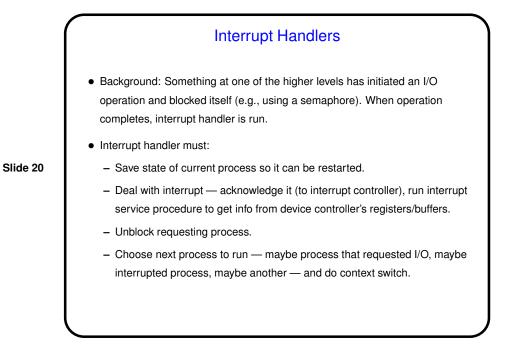


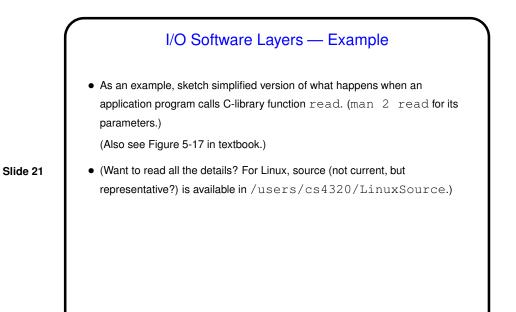




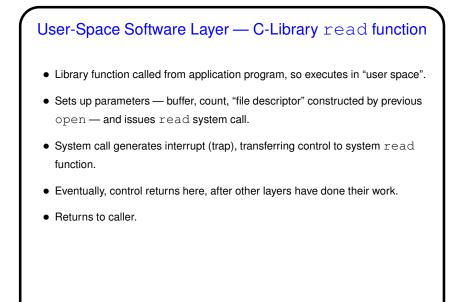


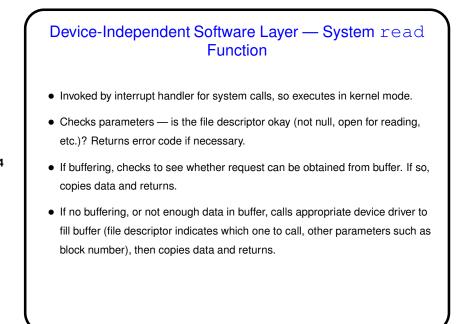


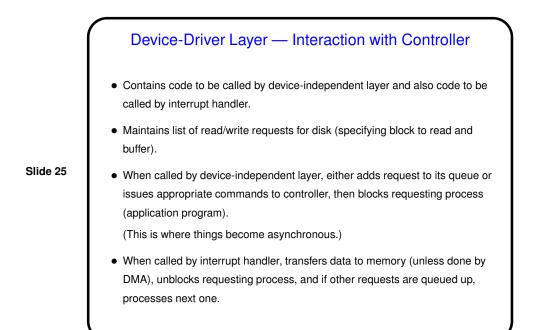




Sidebar: "Opening" Files
(This is really kind of part of the discussion of filesystems?)
You know that in most programming languages you have to "open" a file before working with it. What does that do?
in UNIX/Linux, ultimately results in making an "open file" system call, which builds a system-specific data structure in the O/S's memory, adds it to the list of open files for this process, and returns to the program the index into this list (called a "file descriptor").
What's in that data structure? as best I can tell, function pointers for code to perform operations such as read and write. More about these functions soon.



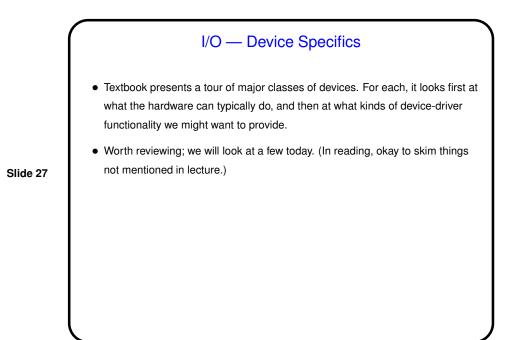




Interrupt-Handler Layer — Processing of I/O Interrupt

- Gets control when requested disk operation finishes and generates interrupt.
- Gets status and data from disk controller, unblocks waiting user process.
 At this point, "call stack" (for user process) contains C library function, system read function, and a device-driver function. We return to the device-driver function and then unwind the stack.

Slide 26



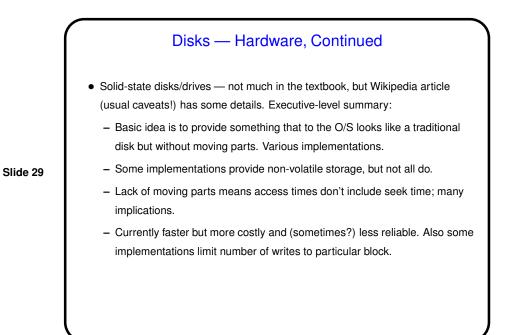
Disks — Hardware
Magnetic disks:

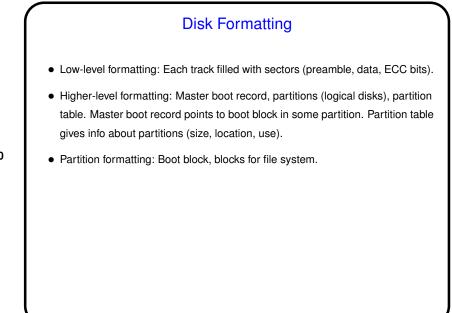
Cylinder/head/sector addressing may or may not reflect physical geometry — controller should handle this.
Controller may be able to manage multiple disks, perform overlapping seeks.

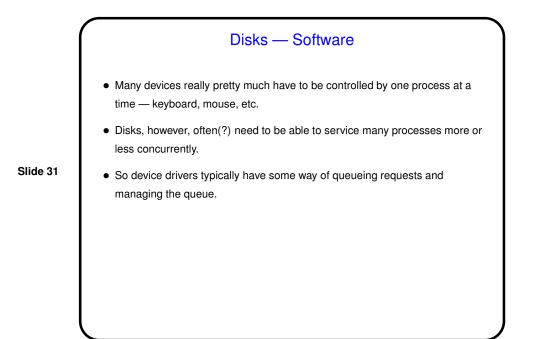
RAID (Redundant Array of Inexpensive/Independent Disks):

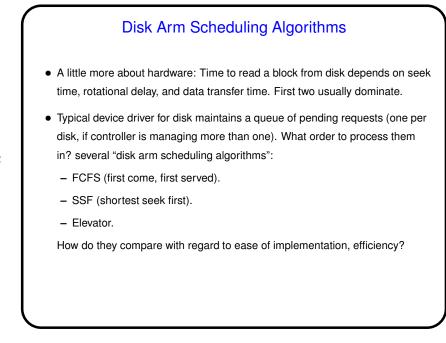
Basic idea is to replace single disk and disk controller with "array" of disks plus RAID controller.
Two possible payoffs: Redundancy and performance (parallelism).
Six "levels" (configurations) defined. Read all about it in textbook if interested.

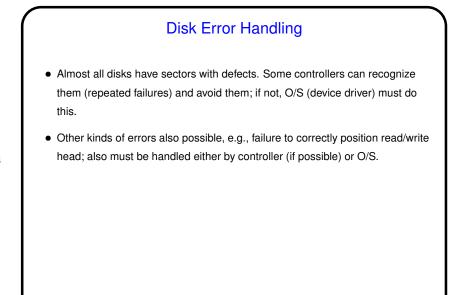
Slide 28

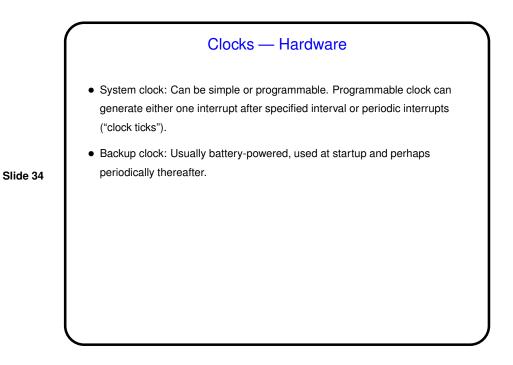


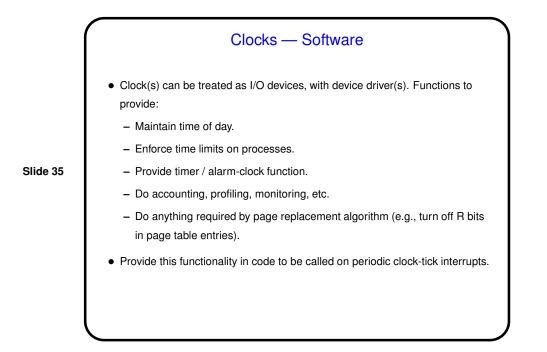






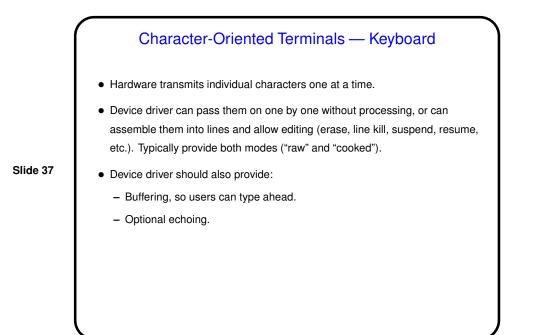


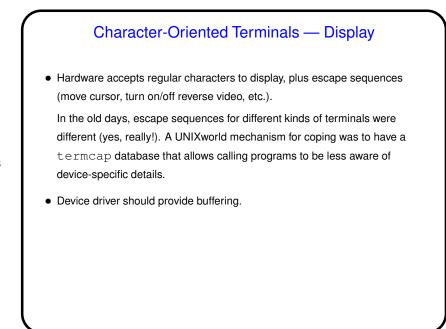


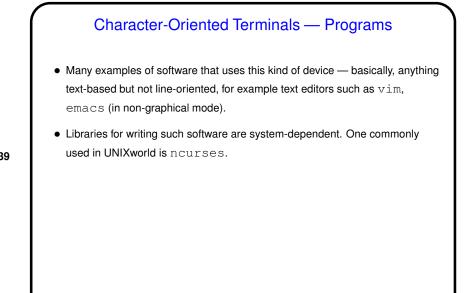


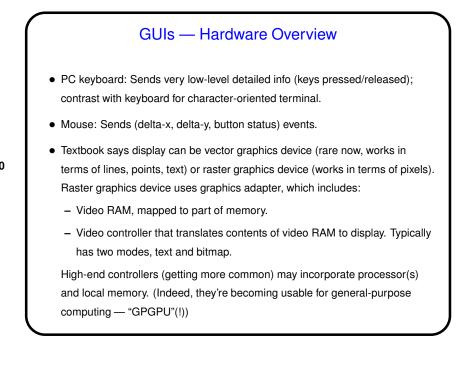
Character-Oriented Terminals — Hardware Overview

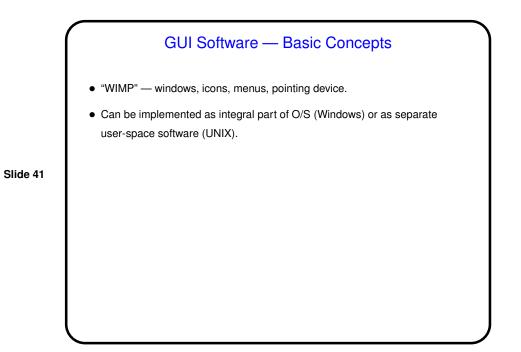
- Hardware consists of character-oriented display (fixed number of rows and columns) and keyboard, connected to CPU by serial line.
- Actual hardware no longer common (except possibly in mainframe world), but emulated in software (e.g., UNIX/Linux terminal windows) so old programs still work. (Why does anyone care? those "old programs" include command shells, text editors, etc., which some of us claim are still useful, and likely to be stable.)

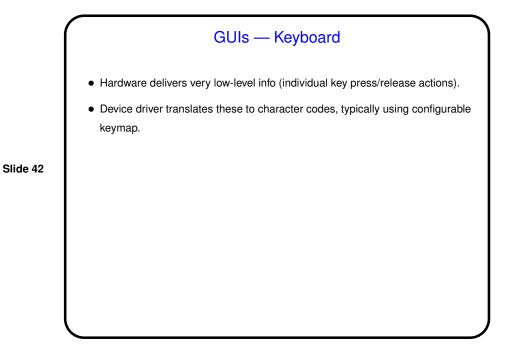


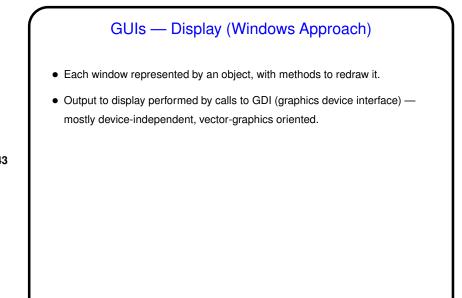












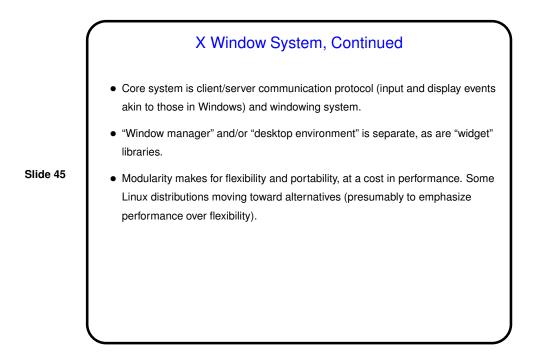
GUIs — Display (Traditional UNIX Approach)

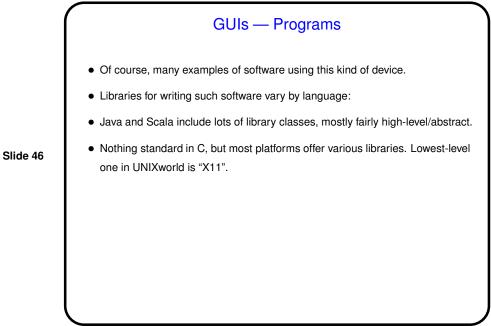
• X Window System (the pedantic call it that and not "X Windows") designed to support both local input/output devices and network terminals, based on a client/server model.

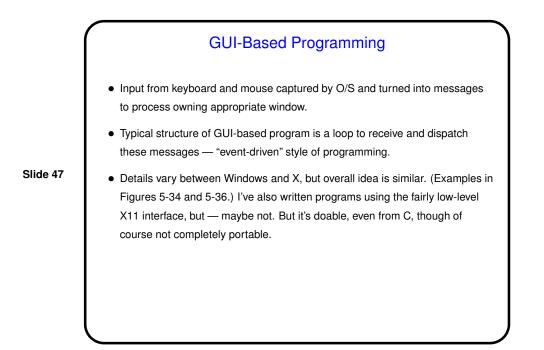
Slide 44

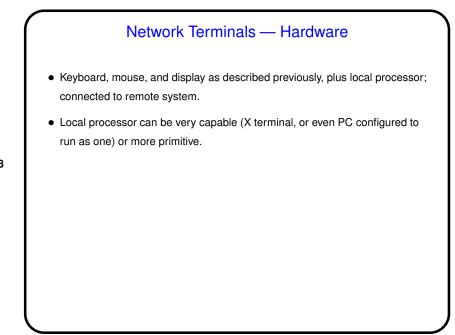
 "Clients" here are programs that want to do GUI I/O; "server" is a program that provides GUI services. An "X server" can run on the same system as the clients, a different UNIX system, an "X terminal (where it's the "O/S"), or under another O/S ("X emulators" for Windows).

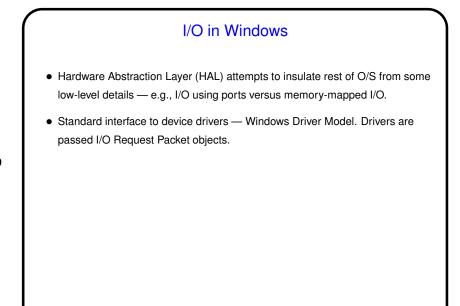
(Figure 5-33 in textbook.)











I/O in UNIX/Linux Access to devices provided by special files (normally in /dev/*), to provide uniform interface for callers. Two categories, block and character. Each defines interface (set of functions) to device driver. Associated with each special file are major and minor device numbers, with major device number used to locate specific function. (Look at some output of ls -l /dev.) For block devices, buffer cache contains blocks recently/frequently used. For character devices, optional line-discipline layer provides some of what we described for text-terminal keyboard driver. Streams provide additional layer of abstraction for callers — can interface to files, terminals, etc. (This is what you access with *scanf, *printf.)

