Administrivia

- Reminder: Homework 5 programming problems due today.

- Draft version of Homework 6 on the Web. Final version to be posted tomorrow I hope. Due next Wednesday. One more homework after that.
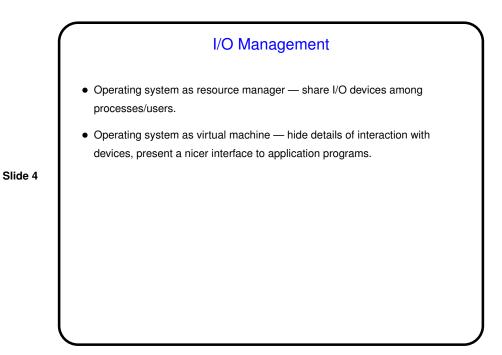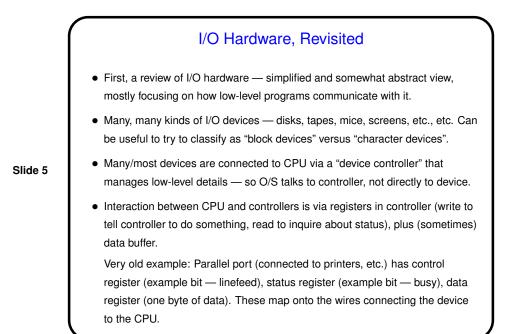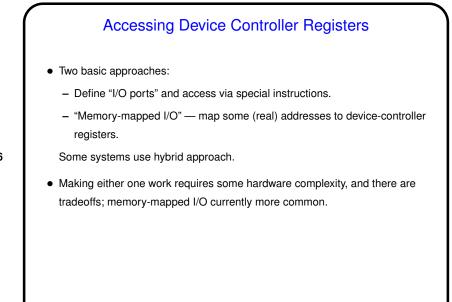
Slide 1

Minute Essay From Last Lecture

- One person asked whether Mac OS X uses a UNIX filesystem. According to Wikipedia, no. But no current system uses exactly the filesystem described for UNIX V7 last time.

Slide 2

Filesystems — What Do Current Systems Use?

Slide 3

- Linux — default is now probably ext4, successor to ext2 and ext3 with journalling. Very much like UNIX V7 conceptually, though with support for much longer filenames. Other filesystems possible/supported, and support for accessing various Windows filesystems provided via Samba.

- Mac OS X ("macOS"?) — Apple File System, externally pretty UNIX-like, possibly internal differences.

- Windows — NTFS is default, support still provided for FAT-xx.

I/O Management

Slide 4

- Operating system as resource manager — share I/O devices among processes/users.

- Operating system as virtual machine — hide details of interaction with devices, present a nicer interface to application programs.

I/O Hardware, Revisited

Slide 5

- First, a review of I/O hardware — simplified and somewhat abstract view, mostly focusing on how low-level programs communicate with it.

- Many, many kinds of I/O devices — disks, tapes, mice, screens, etc., etc. Can be useful to try to classify as "block devices" versus "character devices".

- Many/most devices are connected to CPU via a "device controller" that manages low-level details — so O/S talks to controller, not directly to device.

- Interaction between CPU and controllers is via registers in controller (write to tell controller to do something, read to inquire about status), plus (sometimes) data buffer.

  Very old example: Parallel port (connected to printers, etc.) has control register (example bit — linefeed), status register (example bit — busy), data register (one byte of data). These map onto the wires connecting the device to the CPU.

Accessing Device Controller Registers

Slide 6

- Two basic approaches:
  - Define "I/O ports" and access via special instructions.
  - "Memory-mapped I/O" — map some (real) addresses to device-controller registers.

  Some systems use hybrid approach.

- Making either one work requires some hardware complexity, and there are tradeoffs; memory-mapped I/O currently more common.

Direct Memory Access (DMA)

Slide 7

- When reading more than one byte (e.g., from disk), device controller typically reads into internal buffer, checking for errors. How to then transfer to memory?

- One way: CPU makes transfer, byte by byte.

- Another way: DMA controller makes transfer, having been given a target memory location and a count.
  (Figure 5-4 in textbook.)

- Which is better? consider speed of DMA versus speed of CPU, potential for overlapping data transfer and computation. DMA is extra hardware and could be slower than CPU, but would appear to offer potential to overlap transfer and computation.

Polling Versus Interrupts

Slide 8

- Three basic approaches to writing programs to do I/O: "programmed", "interrupt-driven", and using DMA.

- Which to use — it depends. (No surprise, right?)

Programmed I/O

- Basic idea: Program tells controller what to do and busy-waits until it says it's done.

- Simple but potentially inefficient — for the system as a whole, anyway.

Slide 9

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 10

- 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 . . .)

I/O Using DMA

- Basic idea: Similar to interrupt-driven I/O, but transfer of data to memory done by DMA controller, only one interrupt per block of data.

- Complexity versus efficiency tradeoffs similar to interrupt-driven I/O, but may result in fewer interrupts and allow overlap of computation and I/O.

Slide 11

Interrupts Revisited

- When I/O device finishes its work, it generates interrupt, and then — something happens. What?

- Hardware and software aspects . . .

  (Figure 5-5 in textbook.)

Slide 12

Interrupts, Continued

Slide 13

- I/O device "interrupts" by signalling interrupt controller.

- Interrupt controller signals CPU, with indication of which device caused interrupt, or ignores interrupt (so device controller keeps trying) if interrupt can't be processed right now.

- Processing is then similar to what happens on traps (interrupts generated by system calls, page faults, other errors) . . .
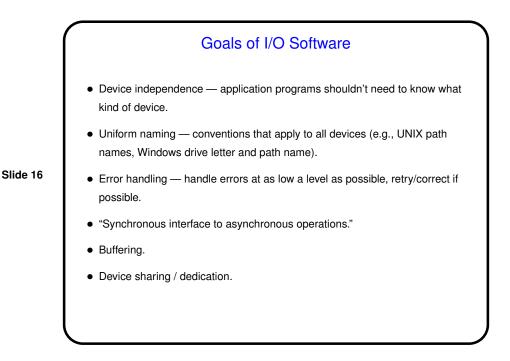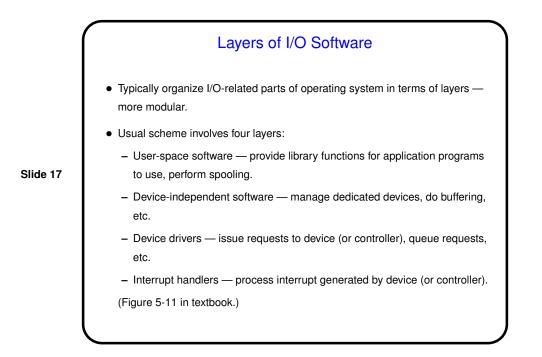
Interrupts, Continued

Slide 14
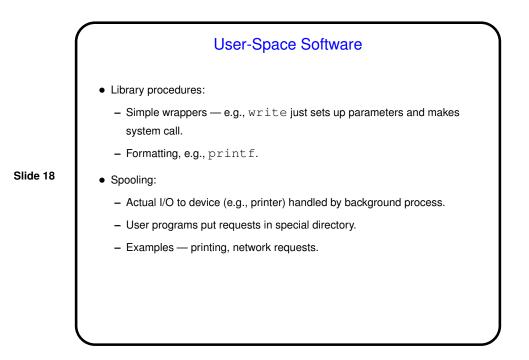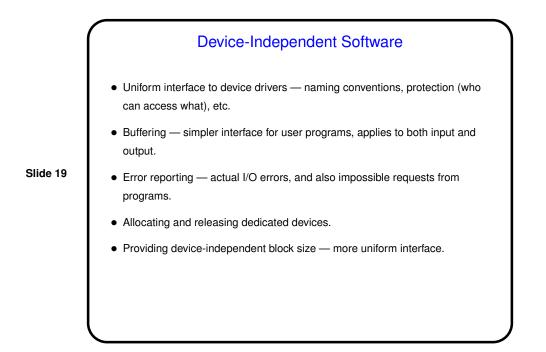
- On interrupt, hardware locates proper interrupt handler (probably using interrupt vector), saves critical info such as program counter, and transfers control (switching into supervisor/kernel mode).

- 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.

Slide 15

Interrupts, Continued

- Worth noting that pipelining (very common in current processors) complicates interrupt handling — when an interrupt happens, there could be multiple instructions in various stages of execution. What to do?

- "Precise interrupts" are those that happen logically between instructions. Can try to build hardware so that this happens always, or sometimes.

- "Imprecise interrupts" are — the other kind. Hardware that generates these may provide some way for software to find out status of instructions that are partially complete. Tanenbaum says this complicates O/S writers' jobs.

Slide 16

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).

- Error handling — handle errors at as low a level as possible, retry/correct if possible.

- "Synchronous interface to asynchronous operations."

- Buffering.

- Device sharing / dedication.

Slide 17

Layers of I/O Software

- Typically organize I/O-related parts of operating system in terms of layers — more modular.

- Usual scheme involves four layers:

  - User-space software — provide library functions for application programs to use, perform spooling.

  - Device-independent software — manage dedicated devices, do buffering, etc.

  - Device drivers — issue requests to device (or controller), queue requests, etc.

  - Interrupt handlers — process interrupt generated by device (or controller).

  (Figure 5-11 in textbook.)

Slide 18

User-Space Software

- Library procedures:

  - Simple wrappers — e.g., `write` just sets up parameters and makes system call.

  - Formatting, e.g., `printf`.

- Spooling:

  - Actual I/O to device (e.g., printer) handled by background process.

  - User programs put requests in special directory.

  - Examples — printing, network requests.

Device-Independent Software

- Uniform interface to device drivers — naming conventions, protection (who can access what), etc.

- Buffering — simpler interface for user programs, applies to both input and output.

Slide 19

- Error reporting — actual I/O errors, and also impossible requests from programs.

- Allocating and releasing dedicated devices.

- Providing device-independent block size — more uniform interface.

Device Drivers

- Idea is to have something that mediates between device controller and O/S — so, need one of these for every combination of O/S and device. Often written by device manufacturer.

- Called by other parts of O/S, we hope according to one of a small number of standard interfaces — e.g., "block device" interface, or "character device" interface. Communicates with device controller in its language (so to speak).

Slide 20

- Normally run in kernel mode. Formerly often compiled into kernel, now usually loaded dynamically (details vary).

Device Drivers, Continued

- When called, must:
  - Check that parameters are okay (return if not).
  - Check that device is not in use (queue request if it is).
  - Talk to device — may involve many commands, may require waiting (block if so).
  - Check for errors, return info to caller. If there are queued requests, continue with next one.

Slide 21

Interrupt Handlers

- Background: Something at one of the higher levels has initiated an I/O operation and blocked itself (e.g., using a semaphore). When operation completes, interrupt handler is run.

- Interrupt handler must:
  - Save state of current process so it can be restarted.
  - Deal with interrupt — acknowledge it (to interrupt controller), run interrupt service procedure to get info from device controller's registers/buffers.
  - Unblock requesting process.
  - Choose next process to run — maybe process that requested I/O, maybe interrupted process, maybe another — and do context switch.

Slide 22

Slide 23

Minute Essay

- Questions? otherwise just sign in.