Slide 1

Administrivia

- Reminder: Please keep the lab doors closed after hours. There have been reports of thefts.
- Homework 4 is on the Web. Due next Tuesday. Optional programming problems may be added soon.

Slide 2

Administrivia

- Dr. Lewis's course next term (CSCI 3294, "Effective OO Programming"):

  "You've learned how to do object-oriented programming in C++/Java, but do you do it well? Do you want to be better? This course will explore how to effectively write code in C++ and Java and how to use design patterns to improve the structure of your OO programs in general. The course will involve reading and a significant amount of coding where you try to apply the things that you have learned. Students in the course will be working with other students to produce a large project."
- My course next term (CSCI 3190, "Unix Power Tools") — description linked from my home page.

Slide 3

Minute Essay From Last Lecture

- Once upon a time, in a mainframe shop . . . (details as presented in class). What was wrong? What should the systems people do?

  The system is thrashing! so they should reduce the number of processes (short term), install more memory (long term), etc.

Slide 4

I/O Management

- 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.

Slide 5

I/O Hardware, Revisited

- 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.

  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.

Slide 6

Accessing Device Controller Registers

- 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. (Notice implications for writing device drivers — which scheme allows writing them without assembly language?)

Slide 7

Direct Memory Access, Revisited

- 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.
- Which is better? DMA is extra hardware and usually slower than CPU, but would appear to offer potential to overlap transfer and computation.

Slide 8

Interrupts, Revisited

- When I/O device finishes its work, it generates interrupt, typically actually 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 now similar to what happens on traps (interrupts generated by system calls, page faults, other errors):

  Hardware locates proper interrupt handler (probably using interrupt vector), saves critical info such as program counter, and transfers control (probably switching into supervisor 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 9

Interrupts, Revisited, A Bit More

- Notice how pipelining complicates things — restarting is much easier with precise interrupts (all instructions before interrupted one complete, none past interrupted one complete, etc.), but these are difficult to get with pipelined processor.

Slide 10

I/O Software, Overview

- 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 11

Polling Versus Interrupts

- Programmed I/O: Program tells controller what to do and busy-waits until it says it's done.

  Simple but potentially inefficient.
- Interrupt-driven I/O: 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,
- I/O using DMA: Similar to interrupt-driven I/O, but transfer of data to memory done by DMA controller, only one interrupt per block of data.

Slide 12

Layers of I/O Software

- Typically organize I/O-related parts of operating system in terms of layers — more modular.
- Usual scheme — see figure on p. 288. Looking at these one at a time, bottom to top . . .

Slide 13

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 14

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

  Check for errors, return info to caller. Manage queued requests.

Slide 15

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.
- 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.

Slide 16

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.

Minute Essay

- How are you liking the textbook so far?
- Are the notes I post to the Web in a format that works for you? PDF is not really negotiable, but I could do two per page or full size — ?