Slide 1

Administrivia

- Homework 6 to be on Web soon — I will send e-mail.
- I plan to be here Wednesday, but class will be optional. Attending will get you a small amount of extra credit on the attendance part of your grade.

Slide 2

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 3

I/O Hardware, Revisited

- 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.

  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 4

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.

Slide 5

Direct Memory Access (DMA)

- 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? 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.

Slide 6

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 7

Interrupts, Revisited, A Bit More

- Notice that 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 8

Mechanics of I/O — 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 9

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 10

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).

Slide 11

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.

Slide 12

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 13

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. Formerly often compiled into kernel, now usually loaded dynamically (details vary).

Slide 14

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 15

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 16

Minute Essay

- We talked about two approaches to communicating with I/O devices — special I/O instructions, and "memory-mapped I/O" (reading/writing particular memory locations). What implications do you think the two choices have for programmers' ability to write device drivers in a (moderately) high-level language such as C?

Minute Essay Answer

- With memory-mapped I/O it should be possible to write device drivers entirely in C; with special I/O instructions this would not be possible without compiler modifications or some amount of assembly-language code.