Slide 1

Administrivia

- Reminder: Homework 3 due Monday. Written part due at 5pm, programming part at 11:59pm.
- Homework 4 to be on the Web soon; due in about a week.

Slide 2

Minute Essay From Last Lecture — Some Responses

- How do Linux/UNIX systems avoid filesystem fragmentation?

  (Not sure, but supposedly has to do with strategy for allocating blocks for new/expanded files. One could do this a simple dumb way, or a smarter way that tries to find groups of contiguous blocks. ?)

Slide 3

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 4

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 5

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 6

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 7

Interrupts

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

Slide 8

Interrupts, Continued

- 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) . . .

Slide 9

Interrupts, Continued

- On interrupt, 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 10

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.

Polling Versus Interrupts

- Three basic approaches to writing programs to do I/O — “programmed”, “interrupt-driven”, and using DMA.
- Which to use — it depends. (No surprise, right?)

Slide 11

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 12

Slide 13

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).
- 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 . . .)

Slide 14

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 15

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 16

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).
- To be continued . . .

Slide 17

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?

Slide 18

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.