Slide 1

Administrivia

- (Via e-mail.)

Slide 2

“Persistence”

- This section merges what in traditional O/S textbooks might be called “I/O management” and “filesystems”.
- Good section, but (I think) too Linux-centric. But maybe if combined with some high-level material from previous textbook (Tanenbaum) . . .

Slide 3

Overview of I/O Hardware

- First, a little about I/O hardware — simplified and somewhat abstract view, mostly focusing on how low-level programs communicate with it. Overall view of the system is as presented in Chapter 36.
- Many, many kinds of I/O devices — disks, tapes, keyboards, mice, screens, etc., etc. Can be useful to try to classify as “block devices” versus “character devices”.
- Many devices also connected to CPU via a “device controller” that manages low-level details — so O/S talks to controller, not directly to device. (In a way, this is a hardware “layers of abstraction” idea?)

Slide 4

Overview of I/O Hardware, Continued

- 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.

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. (Alluded to in section on virtualizing memory.)
- Making either one work requires some hardware complexity, and there are tradeoffs. Note that memory-mapped I/O makes it possible to write device drivers entirely in C! (Though not without breaking a few C-standard rules.)

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? DMA is extra hardware and could be slower than CPU, but offers potential to overlap transfer and computation.

Slide 7

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 8

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. But a good choice if wait time is small.

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).
- 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.

Slide 10

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 . . .

Slide 12

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 for other interrupts (system calls, page faults, etc.).

Slide 13

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.

Slide 14

Goals of I/O Software (Tanenbaum)

- 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." (Interrupt-driven I/O is inherently asychronous, but application programs want to call library functions that hide that.)
- Buffering. (Examples: For disk I/O, faster to read/write at least a block at a time. For keyboard input, nice to let user type head.)
- Device sharing / dedication. (Some devices — e.g., disks — can be used concurrently by multiple processes, but others can't.)

Slide 15

Layers of I/O Software (Tanenbaum)

- 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).
- Possibly useful to review to get a sense of what this part of the O/S does?

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.

Slide 17

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 18

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).
- Code for device drivers contributes to total lines of O/S code way out of proportion to its importance, probably because there are so many devices! Similarly for its contribution to system crashes.

Slide 19

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 20

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 21

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`.)
- Streams provide additional layer of abstraction for callers — can interface to files, terminals, etc. (This is what you access with `*scanf`, `*printf`.)

Slide 22

Minute Essay

- Questions? Does this give you some sense of how O/S's deal with I/O devices?