Slide 1

Administrivia

- Reminder: Homework 6 on Web; due Friday. One more homework (about I/O, etc.) coming soon.

Slide 2

I/O Software Layers — Example

- As an example, sketch simplified version of what happens when an application program calls C-library function `read`. (`man 2 read` for its parameters.)
- (Want to read all the details? For Linux, source (not current, but representative) is available in `/users/cs4320/LinuxSource`.)

Slide 3

User-Space Software Layer — C-Library `read` function

- Library function called from application program, so executes in "user space".
- Sets up parameters — "file descriptor" constructed by previous `open` (more about files in next chapter), buffer, count — and issues `read` system call.
- System call generates interrupt (trap), transferring control to system `read` function.
- Eventually, control returns here, after other layers have done their work.
- Returns to caller.

Slide 4

Device-Independent Software Layer — System `read` Function

- Invoked by interrupt handler for system calls, so executes in kernel mode.
- Checks parameters — is the file descriptor okay (not null, open for reading, etc.)? Returns error code if necessary.
- If buffering, checks to see whether request can be obtained from buffer. If so, copies data and returns.
- If no buffering, or not enough data in buffer, calls appropriate device driver (file descriptor indicates which one to call, other parameters such as block number) to fill buffer, then copies data and returns.

Slide 5

Device-Driver Layer — Read Disk Block

- Contains code to be called by device-independent layer and also code to be called by interrupt handler.
- Maintains list of read/write requests for disk (specifying block to read and buffer).
- When called by device-independent layer, either adds request to its queue or issues appropriate commands to controller, then blocks requesting process (application program).

  (This is where things become asynchronous.)
- When called by interrupt handler, transfers data to memory (unless done by DMA), unblocks requesting process, and if other requests are queued up, processes next one.

Slide 6

Interrupt-Handler Layer — Read Disk Block

- Gets control when requested disk operation finishes and generates interrupt.
- Gets status and data from disk controller, unblocks waiting user process.

  At this point, "call stack" (for user process) contains C library function, system `read` function, and a device-driver function. We return to the device-driver function and then unwind the stack.

Slide 7

I/O Continued — Device Specifics

- Next, a tour of major classes of devices. For each, we look first at what the hardware can typically do, and then at what kinds of device-driver functionality we might want to provide.

Slide 8

Disks — Hardware

- Magnetic disks:
  - Cylinder/head/sector addressing may or may not reflect physical geometry — controller should handle this.
  - Controller may be able to manage multiple disks, perform overlapping seeks.
- RAID (Redundant Array of Inexpensive/Independent Disks):
  - Basic idea is to replace single disk and disk controller with “array” of disks and RAID controller.
  - Two possible payoffs — redundancy and performance (parallelism).
  - Six “levels” (configurations) defined. Read all about it in textbook if interested.
- Optical disks — CD, CD-R, CD-RW, DVD. Okay to skim details!

Slide 9

Disk Formatting

- Low-level formatting — each track filled with sectors (preamble, data, ECC bits).
- Higher-level formatting — master boot record, partitions (logical disks), partition table. Master boot record points to boot block in some partition. Partition table gives info about partitions (size, location, use).
- Partition formatting — boot block, blocks for file system.

Slide 10

Disk Arm Scheduling Algorithms

- A little more about hardware: Time to read a block from disk depends on seek time, rotational delay, and data transfer time. First two usually dominate.
- Earlier we said that typical device driver for disk maintains a queue of pending requests (one per disk, if controller is managing more than one). What order to process them in? several “disk arm scheduling algorithms”:
  - FCFS (first come, first served).
  - SSF (shortest seek first).
  - Elevator.

  How do they compare with regard to ease of implementation, efficiency?

Slide 11

Disk Error Handling

- Almost all disks have sectors with defects. Some controllers can recognize them (repeated failures) and avoid them; if not, o/s (device driver) must do this.
- Other kinds of errors also possible, e.g., failure to correctly position read/write head; also must be handled either by controller (if possible) or o/s.

Slide 12

Clocks — Hardware

- System clock — can be simple or programmable. Programmable clock can generate either one interrupt after specified interval or periodic interrupts ("clock ticks").
- Backup clock — usually battery-powered, used at startup and perhaps periodically thereafter.

Slide 13

Clocks — Software

- Clock(s) can be treated as I/O devices, with device driver(s). Functions to provide:
  - Maintain time of day.
  - Enforce time limits on processes.
  - Provide timer / alarm-clock function.
  - Do accounting, profiling, monitoring, etc.
  - Do anything required by page replacement algorithm (turn off R bits in page table entries, e.g.).
- Provide this functionality in code to be called on clock-tick interrupts.

Slide 14

Character-Oriented Terminals — Hardware Overview

- Hardware consists of character-oriented display (fixed number of rows and columns) and keyboard, connected to CPU by serial line.
- Actual hardware no longer common (except in mainframe world), but emulated in software (e.g., UNIX xterm) so old programs still work. (Why does anyone care? some of those old programs are still useful — e.g., text editors — and usually very stable.)

Slide 15

Character-Oriented Terminals — Keyboard

- Hardware transmits individual ASCII characters.
- Device driver can pass them on one by one without processing, or can assemble them into lines and allow editing (erase, line kill, suspend, resume, etc.). Typically provide both modes.
- Device driver should also provide:
  - Buffering, so users can type ahead.
  - Optional echoing.

Slide 16

Character-Oriented Terminals — Display

- Hardware accepts regular characters to display, plus escape sequences (move cursor, turn on/off reverse video, etc.).

  In the old days, escape sequences for different kinds of terminals were different — hence the need for a `termcap` database that allows calling programs to be less aware of device-specific details.
- Device driver should provide buffering.

Slide 17

GUIs — Hardware Overview

- PC keyboard — sends very low-level detailed info (keys pressed/released); contrast with keyboard for character-oriented terminal.
- Mouse — sends (delta-x, delta-y, button status) events.
- Display can be vector graphics device (rare now, works in terms of lines, points, text) or raster graphics device (works in terms of pixels). Raster graphics device uses graphics adapter, which includes:
  - Video RAM, mapped to part of memory.
  - Video controller that translates contents of video RAM to display. Has two modes, text and bitmap.

Slide 18

GUI Software — Basic Concepts

- "WIMP" — windows, icons, menus, pointing device.
- Can be implemented as integral part of o/s (Windows) or as separate user-space software (UNIX).

Slide 19

GUIs — Keyboard

- Hardware delivers very low-level info (individual key press/release actions).
- Device driver translates these to character codes, typically using configurable keymap.

Slide 20

GUIs — Display (Windows Approach)

- Each window represented by an object, with methods to redraw it.
- Output to display performed by calls to GDI (graphics device interface) — mostly device-independent, vector-graphics oriented. A `.wmf` file (Windows `metafile`) represents a collection of calls to GDI procedures.

Slide 21

GUIs — Display (UNIX Approach)

- X Window System (its real name) designed to support both local input/output devices and network terminals, in terms of:
  - Programs that want to do GUI I/O.
  - Program that provides GUI services. Can run on the same system as applications, a different UNIX system, an X terminal (where it's the "o/s"), or under another o/s ("X emulators" for Windows — e.g., Exceed, XFree86).

  Which is the "client" and which the "server"?
- Core system is client/server communication protocol (input, display events akin to those in Windows) and windowing system. "Window manager" and/or "desktop environment" is separate, as are "widget" libraries. Modularity makes for flexibility and portability, at a cost in performance.

Slide 22

GUI-Based Programming

- Input from keyboard and mouse captured by o/s and turned into messages to process owning appropriate window.
- Typical structure of GUI-based program is a loop to receive and dispatch these messages — "event-driven" style of programming.
- Details vary between Windows and X, but overall idea is similar. See example programs in textbook (or GUI parts of Mandelbrot program for my CSCI 3366 class).

Slide 23

Network Terminals — Hardware

- Keyboard, mouse, and display as described previously, plus local processor; connected to remote system.
- Local processor can be very capable (X terminal, or even PC configured to run as one) or more primitive (SLIM terminal).

Slide 24

Other I/O-Related Topics

- “Stable storage” — use two disks to provide what appears to be a single more reliable one (i.e., write either succeeds or leaves old data in place).
- Power management significant — some devices have “sleeping” and “hibernating” states, o/s can try to determine when it would make sense to use them. Example — screen blanking.

Slide 25

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. Major device number used to locate specific function.
- For block devices, buffer cache contains blocks recently/frequently used.
- For character devices, optional line-discipline layer provides some of what we described for text-terminal keyboard driver.
- Streams provide additional layer of abstraction for callers — can interface to files, terminals, etc. (This is what you access with `*scanf`, `*printf`.) (Aside: How do you get the man page for the `printf` function? (`man printf` gives you something else.) Can be several man pages for given name, in different "sections". Get all of them with `man -a`.)

Slide 26

I/O in Windows

- Hardware Abstraction Layer (HAL) attempts to insulate rest of o/s from some low-level details — e.g., I/O using ports versus memory-mapped I/O.
- Standard interface to device drivers — Windows Driver Model. Drivers are passed I/O Request Packet objects.

Slide 27

Minute Essay

- Last year I argued with a Windows person about schemes for representing devices: UNIX uses “special files”, normally in `/dev` but can be anywhere, identifiable as different from normal files; Windows puts them all at the top level, prefix similar to drive letter.

  Which seems more logical to you, and why? from the standpoint of end users, application programmers, o/s developers?