Slide 1

Administrivia

- (None today.)

Slide 2

Minute Essay From Last Lecture

- How are you liking the textbook so far?

  (Most people do. Good!)
- 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 — ?

  (No strong preference, but several people said fewer slides per page would be good. I'll try two per page.)

Slide 3

I/O Software Layers – Recap

- I/O-related code in operating system is typically organized into 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).
- As an example, sketch simplified version of what likely happens when an application program calls C-library function `read`. (`man 2 read` for its parameters.)

  (Aside: `man -a read` will show you all man pages named `read`.)

Slide 4

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.

Device-Independent Software Layer —System `read` function

Slide 5

- 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.

Device-Driver Layer —Read Disk Block

Slide 6

- 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.

Interrupt-Handler Layer —Read Disk Block

Slide 7

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

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

Device Specifi cs

Slide 8

- 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 9

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 10

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 11

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 12

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.

Character-Oriented Terminals —Display

Slide 13

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

  In olden 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.

GUIs —Hardware Overview

Slide 14

- 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 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.

CSCI 4320 November 6, 2003

GUI Software —Basic Concepts

Slide 15

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

GUIs —Keyboard

Slide 16

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

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 17

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 18

GUIs —Display (Unix Approach)

- X Window System designed to support both local input/output devices and network terminals, in terms of:
  - "X clients" are programs that want to do GUI I/O.
  - "X server" provides GUI services. Can run on the same system as clients, a different Unix system, an X terminal (where it's the "o/s"), or under another o/s ("X servers" for Windows — e.g., Exceed, XFree86).
- Core system is client/server communication protocol (input, display events akin to those in Windows) and windowing system. "Window manager" is separate, as are "widget" libraries. Modularity makes for flexibility and portability, at a cost in performance.

Slide 19

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.

Slide 20

Slide 21

Minute Essay

- None — just sign in on board.