Slide 1

Administrivia

- (By e-mail.)

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.

  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 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.

  (Figure 5-4 in textbook.)
- 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

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?)

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 7

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 8

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 9

Interrupts Revisited

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

  (Figure 5-5 in textbook.)

Slide 10

Slide 11

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 12

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 13

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.

Slide 14

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 15

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).

(Figure 5-11 in textbook.)

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).

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 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.)

  (Also see Figure 5-17 in textbook.)

- (Want to read all the details? For Linux, source (not current, but representative?) is available in `/users/cs4320/LinuxSource`.)

Slide 22

Sidebar: “Opening” Files

- (This is really kind of part of the discussion of filesystems?)
- You know that in most programming languages you have to “open” a file before working with it. What does that do?
- in UNIX/Linux, ultimately results in making an “open file” system call, which builds a system-specific data structure in the O/S’s memory, adds it to the list of open files for this process, and returns to the program the index into this list (called a “file descriptor”).
- What’s in that data structure? as best I can tell, function pointers for code to perform operations such as read and write. More about these functions soon.

Slide 23

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

- Library function called from application program, so executes in "user space".
- Sets up parameters — buffer, count, "file descriptor" constructed by previous `open` — 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 24

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 to fill buffer (file descriptor indicates which one to call, other parameters such as block number), then copies data and returns.

Slide 25

Device-Driver Layer — Interaction with Controller

- 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 26

Interrupt-Handler Layer — Processing of I/O Interrupt

- 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.

I/O — Device Specifics

Slide 27

- Textbook presents a tour of major classes of devices. For each, it looks first at what the hardware can typically do, and then at what kinds of device-driver functionality we might want to provide.
- Worth reviewing; we will look at a few today. (In reading, okay to skim things not mentioned in lecture.)

Disks — Hardware

Slide 28

- 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 plus RAID controller.
  - Two possible payoffs: Redundancy and performance (parallelism).
  - Six “levels” (configurations) defined. Read all about it in textbook if interested.

Slide 29

Disks — Hardware, Continued

- Solid-state disks/drives — not much in the textbook, but Wikipedia article (usual caveats!) has some details. Executive-level summary:
  - Basic idea is to provide something that to the O/S looks like a traditional disk but without moving parts. Various implementations.
  - Some implementations provide non-volatile storage, but not all do.
  - Lack of moving parts means access times don't include seek time; many implications.
  - Currently faster but more costly and (sometimes?) less reliable. Also some implementations limit number of writes to particular block.

Slide 30

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 31

Disks — Software

- Many devices really pretty much have to be controlled by one process at a time — keyboard, mouse, etc.
- Disks, however, often(?) need to be able to service many processes more or less concurrently.
- So device drivers typically have some way of queueing requests and managing the queue.

Slide 32

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.
- 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 33

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 34

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 35

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 (e.g., turn off R bits in page table entries).
- Provide this functionality in code to be called on periodic clock-tick interrupts.

Slide 36

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 possibly in mainframe world), but emulated in software (e.g., UNIX/Linux terminal windows) so old programs still work. (Why does anyone care? those “old programs” include command shells, text editors, etc., which some of us claim are still useful, and likely to be stable.)

Slide 37

Character-Oriented Terminals — Keyboard

- Hardware transmits individual characters one at a time.
- 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 (“raw” and “cooked”).
- Device driver should also provide:
  - Buffering, so users can type ahead.
  - Optional echoing.

Slide 38

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 (yes, really!). A UNIXworld mechanism for coping was to have a `termcap` database that allows calling programs to be less aware of device-specific details.
- Device driver should provide buffering.

Slide 39

Character-Oriented Terminals — Programs

- Many examples of software that uses this kind of device — basically, anything text-based but not line-oriented, for example text editors such as `vim`, `emacs` (in non-graphical mode).
- Libraries for writing such software are system-dependent. One commonly used in UNIXworld is `ncurses`.

Slide 40

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.
- Textbook says 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. Typically has two modes, text and bitmap.

  High-end controllers (getting more common) may incorporate processor(s) and local memory. (Indeed, they're becoming usable for general-purpose computing — "GPGPU"(!))

Slide 41

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 42

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 43

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.

Slide 44

GUIs — Display (Traditional UNIX Approach)

- X Window System (the pedantic call it that and not “X Windows”) designed to support both local input/output devices and network terminals, based on a client/server model.
- “Clients” here are programs that want to do GUI I/O; “server” is a program that provides GUI services. An “X server” can run on the same system as the clients, a different UNIX system, an “X terminal (where it’s the “O/S”), or under another O/S (“X emulators” for Windows).

  (Figure 5-33 in textbook.)

Slide 45

X Window System, Continued

- Core system is client/server communication protocol (input and 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. Some Linux distributions moving toward alternatives (presumably to emphasize performance over flexibility).

Slide 46

GUIs — Programs

- Of course, many examples of software using this kind of device.
- Libraries for writing such software vary by language:
- Java and Scala include lots of library classes, mostly fairly high-level/abstract.
- Nothing standard in C, but most platforms offer various libraries. Lowest-level one in UNIXworld is “X11”.

GUI-Based Programming

Slide 47

- 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. (Examples in Figures 5-34 and 5-36.) I’ve also written programs using the fairly low-level X11 interface, but — maybe not. But it’s doable, even from C, though of course not completely portable.

Network Terminals — Hardware

Slide 48

- 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.

Slide 49

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 50

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`.)
- 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`.)

Slide 51

Minute Essay

- Questions?