Slide 1

Administrivia

- Reminder: Homework 5 programming problems due today. Homework 6 due Monday.

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.
- 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?)

Slide 7

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 8

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 9

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 10

Interrupts Revisited

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

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).

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.)
- (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 (file descriptor indicates which one to call, other parameters such as block number) to fill buffer, 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.

Slide 27

Minute Essay

- Anything noteworthy about the Homework 5 programming problems?