Slide 1

Administrivia

- Homework 3 is on the Web. Due next Tuesday.
- My course next term (CSCI 3294, “Unix Power Tools”) — will be similar to my CSCI 3190 last spring, follow “Old course materials” link from my home page to syllabus, etc.

Slide 2

Minute Essay From Last Lecture

- Consider the following partial program:

```
double a[N][N];
int i, j, k;
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
        a[i][j] = i + j;
    }
}
```

- Reversing the order of the loops can have a big effect on execution time. Why? (Actually there are two explanations, depending on the size of N.)

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

- 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.

  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

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.

  Some systems use hybrid approach.
- Making either one work requires some hardware complexity, and there are tradeoffs; memory-mapped I/O currently more common. (Notice implications for writing device drivers — which scheme allows writing them without assembly language?)

Slide 6

Direct Memory Access, Revisited

- 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.

Slide 7

Interrupts, Revisited

- When I/O device finishes its work, it generates interrupt, typically actually 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 now similar to what happens on traps (interrupts generated by system calls, page faults, other errors):

  Hardware locates proper interrupt handler (probably using interrupt vector), saves critical info such as program counter, and transfers control (probably switching into supervisor 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 8

Interrupts, Revisited, A Bit More

- Notice how pipelining complicates things — restarting is much easier with precise interrupts (all instructions before interrupted one complete, none past interrupted one complete, etc.), but these are difficult to get with pipelined processor.

Slide 9

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 10

Mechanics of I/O — Polling Versus Interrupts

- Programmed I/O: Program tells controller what to do and busy-waits until it says it's done.

  Simple but potentially inefficient.
- Interrupt-driven I/O: 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,
- I/O using DMA: Similar to interrupt-driven I/O, but transfer of data to memory done by DMA controller, only one interrupt per block of data.

Slide 11

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 (see figure on p. 288):
  - Interrupt handlers — process interrupt generated by device (or controller).
  - Device drivers — issue requests to device (or controller), queue requests, etc.
  - Device-independent software — manage dedicated devices, do buffering, etc.
  - User-space software — provide library functions for application programs to use, perform spooling.

Slide 12

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 13

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. Once compiled into kernel, now usually loaded dynamically (details vary).
- 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).

Slide 14

  - Check for errors, return info to caller. Manage queued requests.

Slide 15

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 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.

I/O Software Layers — Example

- 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.)
- (Want to read all the details? Explore `/usr/src/linux*`.)

Slide 17

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 18

Slide 19

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 20

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.

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 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.

Minute Essay

- None — sign in.