

Administrivia

- Reminder: Homework 3 due today.
- Homeworks 4 and 5 on Web. Due after the holiday. There will likely be one more short homework, plus a set of optional extra-credit problems.

Slide 1

Layers of I/O Software — Review

- Typically organize I/O-related parts of operating system in terms of layers — more modular.
- Usual scheme involves four layers . . .

Slide 2

User-Space Software

Slide 3

- 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.

Device-Independent Software

Slide 4

- 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.

Device Drivers

Slide 5

- 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).

Device Drivers, Continued

Slide 6

- 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.

Interrupt Handlers

Slide 7

- 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.

I/O Software Layers — Example

Slide 8

- 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 9

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 10

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 11

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 12

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.

Minute Essay

- A year or two ago 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?

Slide 13