

Slide 1

Administrivia

- Reminder: Homework 4 due today.

Slide 2

I/O Subsystem, Continued

- Goal of this part of the o/s is like for other parts — manage physical resources on behalf of (all) user programs, and provide a nicer “virtual machine”.
- Hardware is diverse and complicated, so in discussing it we try to find basic principles, talk about what’s available to o/s-level programmer.

Goals of I/O Software

Slide 3

- 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.

Layers of I/O Software

Slide 4

- 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):
 - 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).

User-Space Software

Slide 5

- 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 6

- 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 7

- 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 8

- 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 9

- 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 10

- 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 is available in `/users/cs4320/LinuxSource`.)

Slide 11

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 12

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 13

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 14

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

- None — sign in.

Slide 15