Overview

Hardware
- How hardware works

Operating Systems Layer
- What the kernel does

API
- What the programmer does

Hardware

Kinds
- Block devices: read/write a block independent of all others
  - Disk drive
  - Floppy drive
  - USB pendrive
- Character devices: read/write one or more bytes
  - Keyboard
  - Mouse
  - Serial
  - Network
  - Printer
- Miscellaneous
  - Memory-mapped video
  - Video chips
  - Clock

Bus: a set of wires and protocol for communicating
- ISA
- PCI
- Memory bus

Daisy chain: bus with cables going from one device to the next
- USB
- SCSI
- Firewire
Hardware Controllers

**Simple**
- Serial controller
  - has buffer of several bytes

**Complex**
- SCSI controller
  - reads data from drive serially into block of memory inside controller
  - Returns that block of data once block is completely read

Communication between CPU and controller
- Registers
  - Data in
  - Data out
  - Status
  - Control
- How to access register?
  - Memory-mapped I/O
    - Registers are in a portion of memory space. Device controller looks for memory accesses in its own area of space.
  - Using I/O ports
    - Separate instructions for reading and writing port address space. (Separate bit on the bus specifies I/O space rather than memory space).

Communicating with a Hardware Controller

**Polling write (busy-wait):**

**CPU:**
- loop while busy bit in status register is set
- Set write bit in control register.
- Write data byte into data-out register
- Set command-ready bit in control register
- loop while busy bit in status register is set

**Controller:**
- loop while command-ready bit in control register is not set
- Set busy bit in status register
- Since write bit in control register is set, read data-out register
do I/O
  - Clear command-ready bit in control register
  - Clear error bit in status register
  - Clear busy bit in status register
  - Check error

Communicating with a Hardware Controller

**Interrupt-driven I/O**

**CPU:**
- loop while busy bit in status register is set
- Set write bit in control register.
- Write data byte into data-out register
- Set command-ready bit in control register
- Continue processing

**Controller:**
- loop while command-ready bit in control register is not set
- Set busy bit in status register
- Since write bit in control register is set, read data-out register
do I/O
  - Clear command-ready bit in control register
  - Clear error bit in status register
  - Clear busy bit in status register
  - Process interrupt

Interrupts

**Interrupt Levels**
- Specifies priority of an interrupt; higher-level interrupts are serviced before lower-level ones
- Non- Maskable Interrupt (NMI)
  - One that the processor can’t ignore. Debugging button, for example

**Interrupt Controller will:**
- Be notified by hardware controller that it wishes to interrupt
- Check to see whether there’s an already-pending interrupt of same or higher priority
- If not, put interrupt address (small number) on bus, and generate interrupt signal to CPU

**CPU will:**
- Check for interrupt signal after every instruction
- On interrupt (if not masked), save minimal state
  - Use interrupt vector (table at well-known location) to jump to interrupt service routine (ISR)
  - ISR will acknowledge the interrupt when it is ready to handle another interrupt
  - Handle condition that caused interrupt (wake sleeping process)
  - Restore state
Direct Memory Access (DMA)

Without DMA
- CPU must transfer information from device controller to main memory by reading/writing data one byte/word at a time (programmed I/O)

Idea: Have DMA controller that will do the transfer on behalf of the CPU
- Sometimes build into device controller (must be a bus master)
- CPU programs the DMA controller to specify source, destination, and amount
- DMA controller loops:
  - Seize memory bus (cycle stealing)
  - Tell device controller to write to memory address
  - Release memory bus
- When complete, issue interrupt to CPU

Usually used for device generating lots of data
- Disk, video
- Not keyboard, for example

Device Driver Functionality

Initialize
- Probe whether device is there
- Figure out interrupt address (usually known by device controller)
- Allocate data structures

Open
- Install ISR
- Enable interrupts on controller

Close
- Disable interrupts on controller
- Uninstall ISR

Read
- Read data from device

Write
- Write data to device

ioctl
- Device-specific functionality
  - Set baud rate, for example

ISR
- Not called directly, but only in response to an interrupt

Uniformity

Uniform interface between OS and drivers
- Unix character devices
  - read
  - write
  - open
  - close
  - ioctl
- Unix block devices (buffered in kernel memory)
  - open
  - close
  - strategy
    - Given a buffer header with address, read/write bit, block number, word count, major/minor device number

Uniform interface between user programs and devices
- Unix, for example
  - inode specifics block vs. character and major/minor device numbers
  - Major number used as index into table of drivers
  - Minor number used to specify which device
    - which partition on disk
    - which serial port
    - …

Layers of I/O Software

User Space
- Rest of the process
- User-level I/O software

Kernel
- Device-Independent I/O Software
  - IDE Driver
  - Keyboard Driver
- IDE ISR
- Keyboard ISR

Hardware
- bus
- IDE Controller
- USB Controller
- disk
- keyboard
Buffering

Unbuffered
- Driver reads directly into user space, one byte at a time
  - `read(fd, &ch, 1)`;
- Disadvantage: must wake up user process on every byte

Buffered in user space
- Driver reads directly into a buffer in user space
  - `read(fd, buffer, sizeof(buffer))`;
- Disadvantage: must lock the page in memory

Buffered in kernel
- Driver reads into a buffer in the kernel
- When buffer is full, copied to user space
- Disadvantage: copying takes time. Buffer may overflow

Double-buffering in kernel
- Driver reads into a buffer in the kernel
- When buffer is full, copied to user space
- While copying (or waiting for user space page to be paged in), use separate buffer for incoming data
- Disadvantage: more memory used in the kernel

User-space I/O software

Buffering in user-space
- All FILE* routines: `fputc`, `putchar`, `getchar`

Formatting routines:
- `sprintf`, `printf`, etc.

Library interfaces to system calls
- `write`
  - Small assembly-language stub that marshals parameters and issues system call
- `read`
- etc.

Spooling
- printing a file
- UUCP (Unix to Unix Copy)

Disks

Hardware

Time to access a block (sector)
- Seek time (time to move the head in or out to the appropriate track)
- Rotational latency (time for the disk to spin so that the beginning of the sector is under the head)
- Transfer time (time for the data to be read from the sector).
Disk Specs

<table>
<thead>
<tr>
<th></th>
<th>Western Digital Raptor X WD1500AHFD</th>
<th>Hitachi Deskstar 7K250</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>72 GB</td>
<td>250 GB</td>
</tr>
<tr>
<td>Rotational Speed</td>
<td>10,000 RPM</td>
<td>7,200 RPM</td>
</tr>
<tr>
<td>Average rotational latency</td>
<td>3 ms</td>
<td>4 ms</td>
</tr>
<tr>
<td>Average seek time</td>
<td>5 ms</td>
<td>8.5 ms</td>
</tr>
<tr>
<td>Average sustained transfer rate</td>
<td>84 MB/s</td>
<td>60 MB/s</td>
</tr>
<tr>
<td>Buffer size</td>
<td>16 MB</td>
<td>8 MB</td>
</tr>
</tbody>
</table>

RAID

Redundant Array of Inexpensive Disks (compared to SLED: Single large Expensive Disk)
- Bunch of disks controlled by a RAID card
- Looks like SLED to operating system, but provides better performance and better reliability

Level 0
- Striping
- Parallel read (for large read requests)
- Worse reliability

<table>
<thead>
<tr>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1-k</td>
<td>LBN 1-k</td>
<td>LBN k-2k</td>
</tr>
<tr>
<td>Block k-2k</td>
<td>LBN 3k-4k</td>
<td>LBN 4k-5k</td>
</tr>
<tr>
<td>Block 2k-3k</td>
<td>LBN 6k-7k</td>
<td>LBN 7k-8k</td>
</tr>
</tbody>
</table>

Level 1
- Every strip is written twice
- Read from one disk while busy with another

<table>
<thead>
<tr>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1-k</td>
<td>LBN 1-k</td>
<td>LBN k-2k</td>
</tr>
<tr>
<td>Block k-2k</td>
<td>LBN 3k-4k</td>
<td>LBN 4k-5k</td>
</tr>
<tr>
<td>Block 2k-3k</td>
<td>LBN 6k-7k</td>
<td>LBN 7k-8k</td>
</tr>
</tbody>
</table>

Level 2
- Break each word into bits
- Add Hamming code
  - can correct any 1-bit error
  - can detect any 2-bit error
- Can lose any one drive
- Disks must be synchronized (rotational position and head location)
- Throughput increased by 16x

<table>
<thead>
<tr>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
<th>...</th>
<th>Disk 21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 1</td>
<td>En bit</td>
<td>En bit</td>
<td>Bit 1</td>
<td>...</td>
</tr>
<tr>
<td>Bit 2</td>
<td>En bit</td>
<td>En bit</td>
<td>Bit 1</td>
<td>...</td>
</tr>
<tr>
<td>Bit 3</td>
<td>En bit</td>
<td>En bit</td>
<td>Bit 1</td>
<td>...</td>
</tr>
</tbody>
</table>

Level 3
- One extra parity drive
- Similar to level 2
- Can't correct silent errors

<table>
<thead>
<tr>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
<th>...</th>
<th>Disk 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 1</td>
<td>En bit</td>
<td>En bit</td>
<td>...</td>
<td>Byte 16</td>
</tr>
<tr>
<td>Bit 2</td>
<td>En bit</td>
<td>En bit</td>
<td>...</td>
<td>Byte 16</td>
</tr>
<tr>
<td>Bit 3</td>
<td>En bit</td>
<td>En bit</td>
<td>...</td>
<td>Byte 16</td>
</tr>
</tbody>
</table>

Level 4
- Like level 0, but with extra parity drive
- On write, must re-read all strips to recalculate parity
  - Or, can pre-read old parity and data to compute new parity
- Parity drive may become bottleneck (used on every write)

<table>
<thead>
<tr>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
<th>Disk 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1-k</td>
<td>LBN 1-k</td>
<td>LBN k-2k</td>
<td>LBN 3k-4k</td>
</tr>
<tr>
<td>Block k-2k</td>
<td>LBN 3k-4k</td>
<td>LBN 4k-5k</td>
<td>LBN 5k-6k</td>
</tr>
<tr>
<td>Block 2k-3k</td>
<td>LBN 6k-7k</td>
<td>LBN 7k-8k</td>
<td>LBN 8k-9k</td>
</tr>
</tbody>
</table>

Level 5
- Like level 4, but with parity strip spread across drives
**Disk-Arm Scheduling Algorithms**

Reducing seek time will increase system performance

**First Come First Serve (FCFS)**
- Can require lots of seeking (imagine queue containing cylinder 1, cylinder n, cylinder 2, cylinder n-5)

**Shortest Seek First (SSF)**
- From among the cylinders in the queue, go to the one that’s nearest
- Can increase throughput
- Disadvantage:
  - Imaging queue contains request for block in cylinder 1, but other requests keep coming in that are closer to the disk arm. A request can be indefinitely delayed.

**Elevator (or SCAN)**
- Keep state of moving up or down. Move the disk arm to next-closest requested higher cylinder (if moving up, otherwise lower cylinder). If no higher cylinder, switch direction.

**Circular Scan (CSCAN)**
- Like Elevator, but always go up. When at end, move to lowest requested cylinder without servicing requests in-between.

**Anticipatory Scheduling**

Scenario:
- Imagine two processes p and q each writing to 10 MB disk file. If running alone, each takes 5 seconds.
- How long to run together?
  - p and q will issue requests
  - If p is satisfied first, scheduling algorithm will try to satisfy q’s request.
  - When p runs, it’ll issue another write request
  - By that time, the disk head has moved way off.

Solution:
- Add anticipation to any disk-scheduling algorithm.
- Simple explanation
  - The last process that issued a request will probably issue another one soon (in same general location)
  - If request we’d otherwise do is not close, instead of moving the disk head, delay and do nothing for a short time
  - If request comes in from process during that time, stop waiting
  - and, presumably, read/write that requested (close) disk block
  - If timer expires, continue with disk-scheduling algorithm

Benefits
- Decreased latency, increased bandwidth

**Reference**

**Implementation: Linux**
- Streaming read while streaming write taking place
  - Without anticipatory: 42, 48, 47 seconds
  - With anticipatory: 3.8 seconds
- Reading many small files while streaming write taking place
  - Without anticipatory: >15:00, 7:27, 9:55 minutes:seconds
  - With anticipatory: 17 seconds

More formally
- benefit = CalculateSeekTime(candidate) - Expected seek time of process
- cost = max(0, expected median think time of process - elapsed time)
- duration = max(0, expected 95th percentile think time of process - elapsed time)
- if (benefit > cost)
  - time_to_wait = duration
- else
  - don’t wait

**Bad Blocks**

Low-level format (at factory) reserves blocks
- Not all at beginning or end, but spread out across disk

Controller will detect bad blocks and either:
- Spare a block (for example, if block 17 is bad, remap it to block 120)
- Skip a contiguous section of blocks
  - If block 17 is bad and first reserved block is 120, remap blocks 17-119 to 18-120 (and move blocks 18-119 up one)

Drive controller can report information on bad blocks, etc. to OS
- SMART (Self-Monitoring, Analysis and Reporting Technology)