Read: A&D Ch 12
Recall: What’s a Bus?

- Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
  - Operations: e.g., Read, Write
  - Control lines, Address lines, Data lines
- Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address, length, data
- Very high BW close to processor (wide, fast, and inflexible), low BW with high flexibility out in I/O subsystem
Recall: Typical PCI Architecture

- CPU
- RAM
- Memory Bus
- Host Bridge
- ISA Bridge
- ISA Controller
  - Legacy Devices
- PCI Bridge
- PCI Slots
- USB Controller
- Root Hub
- Webcam
- Hub
- Scanner
- DVD ROM
- Hard Disk
- SATA Controller
- PCI #0
- PCI #1
- Keyboard
- Mouse
Recall: How does the Processor Talk to the Device?

• CPU interacts with a Controller
  • Contains a set of registers that can be read and written
  • May contain memory for request queues, etc.
• Processor accesses registers in two ways:
  • Port-Mapped I/O: in/out instructions
    • Example from the Intel architecture: out 0x21, AL
  • Memory-mapped I/O: load/store instructions
    • Registers/memory appear in physical address space
    • I/O accomplished with load and store instructions
Recall: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by HW jumpers or at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - Addr: 0x8000F000 — 0x8000FFFF
  - Writing graphics description to cmd queue
    - Say enter a set of triangles describing some scene
      - Addr: 0x80010000 — 0x8001FFFF
    - Writing to the command register may cause on-board graphics hardware to do something
      - Say render the above scene
      - Addr: 0x0007F004
  - Can protect with address translation
There’s more than just a CPU in there!
Chip-Scale Features of Skylake (x86 in 2015)

- Significant pieces:
  - Four OOO cores with deeper buffers
  - Integrated GPU, System Agent (Mem, Fast I/O)
  - Large shared L3 cache with on-chip ring bus
    - 2 MB/core instead of 1.5 MB/core
    - High-BW access to L3 Cache

- Integrated I/O
  - Integrated memory controller (IMC)
    - Two independent channels of DRAM
  - High-speed PCI-Express (for Graphics cards)
  - Direct Media Interface (DMI) Connection to PCH (Platform Control Hub)
Skylake I/O: Platform Controller Hub (PCH)

Sky Lake
System Configuration

Platform Controller Hub

- Connected to processor with proprietary bus
  - Direct Media Interface

- Types of I/O on PCH:
  - USB, Ethernet
  - Thunderbolt 3
  - Audio, BIOS support
  - More PCI Express (lower speed than on Processor)
  - SATA (for Disks)
Operational Parameters for I/O

• Data granularity: Byte vs. Block
  • Some devices provide single byte at a time (e.g., keyboard)
  • Others provide whole blocks (e.g., disks, networks, etc.)

• Access pattern: Sequential vs. Random
  • Some devices must be accessed sequentially (e.g., tape)
  • Others can be accessed “randomly” (e.g., disk, CD, etc.)
    • Fixed overhead to start transfers
    • Some devices require continual monitoring (polling)
    • Others generate interrupts when they need service

• Transfer Mechanism: Programmed I/O and DMA
Transferring Data To/From Controller

• **Programmed I/O:**
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size

• **Direct Memory Access:**
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly

• Sample interaction with DMA controller (from OSTEP book):
Transferring Data To/From Controller

• **Programmed I/O:**
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size

• **Direct Memory Access:**
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly

• **Sample interaction with DMA controller (from OSTEP book):**
Aside: Linux Memory Details

• Memory management in Linux considerably more complex than the examples we have been discussing

• Memory Zones: physical memory categories
  • ZONE_DMA: < 16MB memory, DMAable on ISA bus
  • ZONE_NORMAL: 16MB → 896MB (mapped at 0xC0000000)
  • ZONE_HIGHMEM: Everything else (> 896MB)

• Each zone has 1 freelist, 2 LRU lists (Active/Inactive)

• Many different types of allocation
  • SLAB allocators, per-page allocators, mapped/unmapped

• Many different types of allocated memory:
  • Anonymous memory (not backed by a file, heap/stack)
  • Mapped memory (backed by a file)

• Allocation priorities
  • Is blocking allowed/etc.
I/O Device Notifying the OS

• The OS needs to know when:
  • The I/O device has completed an operation
  • The I/O operation has encountered an error

• I/O Interrupt: Device generates interrupt when it needs service
  • Handles unpredictable events well, but high overhead

• Polling: OS periodically checks device-specific status register
  • Low overhead, but may waste cycles for infrequent or unpredictable I/O

• Actual devices combine both polling and interrupts
  • E.g., high-bandwidth network adapter
Kernel Device Structure
Device Drivers

• Device-specific code in the kernel that interacts directly with the device hardware
  • Supports a standard, internal interface
  • Same kernel I/O system can interact easily with different device drivers
  • Special device-specific configuration supported with the ioctl() system call

• Device Drivers typically divided into two pieces:
  • Top half: accessed in call path from system calls
    • implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl()
      This is the kernel’s interface to the device driver
    • Top half will start I/O to device, may put thread to sleep until finished
  • Bottom half: run as interrupt routine
    • Gets input or transfers next block of output
    • May wake sleeping threads if I/O now complete
  • In Linux, this convention is reversed!
Recall: Life Cycle of an I/O Request
The Goal of the I/O Subsystem

• Provide Uniform Interfaces, Despite Wide Range of Different Devices
  • This code works on many different devices:
    ```c
    FILE fd = fopen("/dev/something", "rw");
    for (int i = 0; i < 10; i++) {
      fprintf(fd, "Count %d\n", i);
    }
    close(fd);
    • Why? Because code that controls devices ("device driver") implements standard interface

• We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  • Can only scratch surface!
Want Standard Interface to Devices

• **Block Devices:** *e.g.* disk drives, tape drives, DVD-ROM
  • Access blocks of data
  • Commands include `open()`, `read()`, `write()`, `seek()`
  • Raw I/O or file-system access
  • Memory-mapped file access possible

• **Character Devices:** *e.g.* keyboards, mice, serial ports, some USB devices
  • Single characters at a time
  • May not be buffered like block devices
  • Libraries layered on top allow line editing

• **Network Devices:** *e.g.* Ethernet, Wireless, Bluetooth
  • Different enough from block/character devices to have an extended interface
  • Unix and Windows include `socket` interface
    • Separates network protocol from network operation
    • Includes `select()` functionality
  • Usage: pipes, FIFOs, streams, queues, mailboxes
How Does User Deal with I/O Timing?

• **Blocking Interface: “Wait”**
  • When request data (e.g. read syscall), put process to sleep until data is ready
  • When write data (e.g. write syscall), put process to sleep until device is ready for data

• **Non-blocking Interface: “Don’t Wait”**
  • Returns quickly from read or write request with count of bytes successfully transferred
  • Read may return nothing, write may write nothing

• **Asynchronous Interface: “Tell Me Later”**
  • When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  • When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Storage Devices
Hard Disk Drivers (HDDs)

Western Digital Drive
http://www.storagereview.com/guide/

IBM/Hitachi Microdrive

IBM Personal Computer/AT (1986)
30 MB hard disk - $500
30-40ms seek time
0.7-1 MB/s (est.)
The Amazing Magnetic Disk

• Unit of Transfer: Sector
  • Ring of sectors form a track
  • Stack of tracks form a cylinder
  • Heads position on cylinders

• Disk Tracks ~ 1µm (micron) wide
  • Wavelength of light is ~ 0.5µm
  • Resolution of human eye: 50µm
  • 100K tracks on a typical 2.5” disk

• Separated by unused guard regions
  • Reduces likelihood neighboring tracks are corrupted during writes (still a small non-zero chance)
The Amazing Magnetic Disk

• Track length varies across disk
  • Outside: More sectors per track, higher bandwidth
  • Disk is organized into regions of tracks with same # of sectors/track
  • Only outer half of radius is used
    • Most of the disk area in the outer regions of the disk

• Disks so big that some companies (like Google) reportedly only use part of disk for active data
  • Rest is archival data
Shingled Magnetic Recording (SMR)

- Overlapping tracks yields greater density, capacity
- Restrictions on writing, complex DSP for reading
Review: Magnetic Disks

- **Cylinders**: all the tracks under the head at a given point on all surfaces
- **Read/write data** is a three-stage process:
  - **Seek time**: position the head/arm over the proper track
  - **Rotational latency**: wait for desired sector to rotate under r/w head
  - **Transfer time**: transfer a block of bits (sector) under r/w head

Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time
Disk Performance Example

• Assumptions:
  • Ignoring queuing and controller times for now
  • Avg seek time of 5ms,
  • 7200RPM ⇒ Time for rotation: 60000 (ms/min) / 7200(rev/min) ≈ 8ms
  • Transfer rate of 50MByte/s, block size of 4Kbyte ⇒
    4096 bytes/50×10^6 (bytes/s) = 81.92 × 10^-6 sec ≈ 0.082 ms for 1 sector

• Read block from random place on disk:
  • Seek (5ms) + Rot. Delay (4ms) + Transfer (0.082ms) = 9.082ms
  • Approx 9ms to fetch/put data: 4096 bytes/9.082×10^-3 s ≈ 451KB/s

• Read block from random place in same cylinder:
  • Rot. Delay (4ms) + Transfer (0.082ms) = 4.082ms
  • Approx 4ms to fetch/put data: 4096 bytes/4.082×10^-3 s ≈ 1.03MB/s

• Read next block on same track:
  • Transfer (0.082ms): 4096 bytes/0.082×10^-3 s ≈ 50MB/sec

• Key to using disk effectively (especially for file systems) is to minimize seek and rotational delays
Lots of Intelligence in the Controller

• Sectors contain sophisticated error correcting codes
  • Disk head magnet has a field wider than track
  • Hide corruptions due to neighboring track writes

• Sector sparing
  • Remap bad sectors transparently to spare sectors on the same surface

• Slip sparing
  • Remap all sectors (when there is a bad sector) to preserve sequential behavior

• Track skewing
  • Sector numbers offset from one track to the next, to allow for disk head movement for sequential ops
## Typical Numbers for Magnetic Disk

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Info/Range</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Space/Density</td>
<td>Space: 14TB (Seagate), 8 platters, in 3½ inch form factor! <strong>Areal Density: ≥ 1 Terabit/square inch! (PMR, Helium, …)</strong></td>
</tr>
<tr>
<td>Average Seek Time</td>
<td>Typically 4-6 milliseconds</td>
</tr>
<tr>
<td>Average Rotational Latency</td>
<td>Most laptop/desktop disks rotate at 3600-7200 RPM (16-8 ms/rotation). Server disks up to 15,000 RPM. Average latency is halfway around disk so 4-8 milliseconds</td>
</tr>
<tr>
<td>Controller Time</td>
<td>Depends on controller hardware</td>
</tr>
</tbody>
</table>
| Transfer Time           | Typically 50 to 250 MB/s. Depends on:  
  • Transfer size (usually a sector): 512B – 1KB per sector  
  • Rotation speed: 3600 RPM to 15000 RPM  
  • Recording density: bits per inch on a track  
  • Diameter: ranges from 1 in to 5.25 in                                                           |
| Cost                    | Used to drop by a factor of two every 1.5 years (or faster), now slowing down                                                               |
Hard Drive Prices over Time

Disk cost-per-byte

- actual data points 1990-2013
- linear fit to data points 1990-2010
- range of industry projections 2013-2020
Example of Current HDDs

• Seagate Exos X14 (2018)
  • 14 TB hard disk
    • 8 platters, 16 heads
    • Helium filled: reduce friction and power
  • 4.16ms average seek time
  • 4096 byte physical sectors
  • 7200 RPMs
  • 6 Gbps SATA /12Gbps SAS interface
    • 261MB/s MAX transfer rate
    • Cache size: 256MB
  • Price: $615 (< $0.05/GB)

• IBM Personal Computer/AT (1986)
  • 30 MB hard disk
  • 30-40ms seek time
  • 0.7-1 MB/s (est.)
  • Price: $500 ($17K/GB, 340,000x more expensive !!)
Solid State Disks (SSDs)

• 1995 – Replace rotating magnetic media with non-volatile memory (battery backed DRAM)
• 2009 – Use NAND Multi-Level Cell (2 or 3-bit/cell) flash memory
  • Sector (4 KB page) addressable, but stores 4-64 “pages” per memory block
  • Trapped electrons distinguish between 1 and 0
• No moving parts (no rotate/seek motors)
  • Eliminates seek and rotational delay (0.1-0.2ms access time)
  • Very low power and lightweight
  • Limited “write cycles”
• Rapid advances in capacity and cost ever since!
SSD Architecture – Reads

- **Read 4 KB Page: ~25 usec**
  - No seek or rotational latency
  - Transfer time: transfer a 4KB page
    - SATA: 300-600MB/s => ~4 x10^3 b / 400 x 10^6 bps => 10 us
  - Latency = Queuing Time + Controller time + Xfer Time
  - Highest Bandwidth: Sequential OR Random reads
SSD Architecture – Writes

• Writing data is complex! (~200μs – 1.7ms )
  • Can only write empty pages in a block
  • Erasing a block takes ~1.5ms
  • Controller maintains pool of empty blocks by coalescing used pages (read, erase, write), also reserves some % of capacity
• Rule of thumb: writes 10x reads, erasure 10x writes

SSD Architecture – Writes

• SSDs provide same interface as HDDs to OS – read and write chunk (4KB) at a time
• But can only overwrite data 256KB at a time!
• Why not just erase and rewrite new version of entire 256KB block?
  • Erasure is very slow (milliseconds)
  • Each block has a finite lifetime, can only be erased and rewritten about 10K times
  • Heavily used blocks likely to wear out quickly
Solution – Two Systems Principles

1. Layer of Indirection
   • Maintain a Flash Translation Layer (FTL) in SSD
   • Map virtual block numbers (which OS uses) to physical page numbers (which flash mem. controller uses)
   • Can now freely relocate data w/o OS knowing

2. Copy on Write
   • Don’t overwrite a page when OS updates its data
   • Instead, write new version in a free page
   • Update FTL mapping to point to new location
Flash Translation Layer

• No need to erase and rewrite entire 256KB block when making small modifications

• SSD controller can assign mappings to spread workload across pages
  • Wear Levelling

• What to do with old versions of pages?
  • Garbage Collection in background
  • Erase blocks with old pages, add to free list
Some “Current” 3.5in SSDs

• Seagate Nytro SSD: 15TB (2017)
  • Dual 12Gb/s interface
  • Seq reads 860MB/s
  • Seq writes 920MB/s
  • Random Reads (IOPS): 102K
  • Random Writes (IOPS): 15K
  • Price (Amazon): $6325 ($0.41/GB)

• Nimbus SSD: 100TB (2019)
  • Dual port: 12Gb/s interface
  • Seq reads/writes: 500MB/s
  • Random Read Ops (IOPS): 100K
  • Unlimited writes for 5 years!
  • Price: ~ $50K? ($0.50/GB)
SSD prices drop much faster than HDD
SSD Summary

• Pros (vs. hard disk drives):
  • Low latency, high throughput (eliminate seek/rotational delay)
  • No moving parts:
    • Very light weight, low power, silent, very shock insensitive
    • Read at memory speeds (limited by controller and I/O bus)

• Cons
  • Small storage (0.1-0.5x disk), expensive (3-20x disk)
    • Hybrid alternative: combine small SSD with large HDD
  • Asymmetric block write performance: read pg/erase/write pg
    • Controller garbage collection (GC) algorithms have major effect on performance
  • Limited drive lifetime
    • 1-10K writes/page for MLC NAND
    • Avg failure rate is 6 years, life expectancy is 9–11 years

• These are changing rapidly!

No longer true!
Announcements

Quiz 2 is graded! Scores will be released tonight.
Announcements

• Homework 4 is due tonight!

• Project 2: code is due tomorrow, final report/scheduling lab due Wednesday

• Homework 5 is released tomorrow

• Updated homework plan
  • Homework 5 will have two parts; each will count as a homework
  • Homework 6 will be optional
A Bit of I/O Performance
Recall: Times (s) and Rates (op/s)

- **Latency** – time to complete a task
  - Measured in units of time (s, ms, us, …, hours, years)

- **Response Time** - time to initiate and operation and get its response
  - Able to issue one that *depends* on the result
  - Know that it is done (anti-dependence, resource usage)

- **Throughput** or **Bandwidth** – rate at which tasks are performed
  - Measured in units of things per unit time (ops/s, GLOP/s)

- Performance???
  - Operation time (4 mins to run a mile…)
  - Rate (mph, mpg, …)
Basic Performance Concepts

• **Response Time or Latency**: Time to perform an operation(s)

• **Bandwidth or Throughput**: Rate at which operations are performed (op/s)
  • Files: MB/s, Networks: Mb/s, Arithmetic: GFLOP/s

• **Start up or “Overhead”**: time to initiate an operation

• Most I/O operations are roughly linear in $b$ bytes
  • Latency(b) = Overhead + $b$/TransferCapacity
Example (Fast Network)

- Consider a 1 Gb/s link ($B = 125$ MB/s) with startup cost $S = 1$ ms
  - Latency: $L(b) = S + \frac{b}{B}$
  - Effective Bandwidth:
    \[
    E(b) = \frac{b}{S + \frac{b}{B}} = \frac{B \cdot b}{B \cdot S + b} = \frac{B}{b} + 1
    \]
  - Half-power Bandwidth: $E(b) = \frac{B}{2}$
  - For this example, half-power bandwidth occurs at $b = 125$ KB
Example: 10 ms Startup Cost (e.g., Disk)

- Half-power bandwidth at $b = 1.25$ MB

- Large startup cost can degrade effective bandwidth

- Amortize it by performing I/O in larger blocks

Half-power $b = 1,250,000$ bytes!
What Determines Peak BW for I/O?

• Bus Speed
  • PCI-X: 1064 MB/s = 133 MHz x 64 bit (per lane)
  • ULTRA WIDE SCSI: 40 MB/s
  • Serial Attached SCSI & Serial ATA & IEEE 1394 (firewire): 1.6 Gb/s full duplex (200 MB/s)
  • USB 3.0 – 5 Gb/s
  • Thunderbolt 3 – 40 Gb/s

• Device Transfer Bandwidth
  • Rotational speed of disk
  • Write / Read rate of NAND flash
  • Signaling rate of network link

• Whatever is the bottleneck in the path...
Overall I/O performance

• Performance of I/O subsystem
  • Metrics: Response Time, Throughput
  • Effective BW = transfer size / response time
  • Contributing factors to latency:
    • Software paths (can be loosely modeled by a queue)
    • Hardware controller
    • I/O device service time

• Queuing behavior:
  • Can lead to big increases of latency as utilization increases
  • Solutions?
Recall: I/O and Storage Layers

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

Streams

File Descriptors

open(), read(), write(), close(), ...

Open File Descriptions

Files/Directories/Indexes

Commands and Data Transfers

Disks, Flash, Controllers, DMA

What we covered in Week #2

What we just covered...

What we will cover next...
From Storage to File Systems

I/O API and syscalls

Variable-Size Buffer

Memory Address

Logical Index, Typically 4 KB

File System

Block

Hardware Devices

Sector(s)

Physical Index, 512B or 4KB

HDD

Flash Trans. Layer

SSD

Phys. Block

Erasure Page

Phys Index., 4KB
Building a File System

Classic OS situation

*Take limited hardware interface (array of blocks) and provide a more convenient/useful interface with:*

- Naming: Find file by name, not block numbers
- Organize file names with directories
- Organization: Map files to blocks
- Protection: Enforce access restrictions
- Reliability: Keep files intact despite crashes, hardware failures, etc.
Translation from User to System View

What happens if user says: “give me bytes 2 – 12?”
• Fetch block corresponding to those bytes
• Return just the correct portion of the block
• What about writing bytes 2 – 12?
  • Fetch block, modify relevant portion, write out block

Everything inside file system is in terms of whole-size blocks
• Actual disk I/O happens in blocks
• read/write smaller than block size needs to translate and buffer
Disk Management

• The disk is accessed a linear array of sectors.
• How to identify a sector?
  • Physical position
    • Sector is a vector [cylinder, surface, sector]
    • Not used anymore
    • OS/BIOS must deal with bad sectors
  • Logical Block Addressing (LBA)
    • Every sector has an integer address
    • Control translates from address to physical position
    • Shields OS from disk structure
What Does the File System Need?

• Track free disk blocks
  • Need to know where to put newly written data
• Track which blocks contain data for which files
  • Need to know where to read a file from
• Track files in a directory
  • Find list of file's blocks given its name
• Where do we maintain all of this?
  • Somewhere on disk
Data Structures on Disk

• Bit different than data structures in memory
• Access a block at a time
  • Can't efficiently read/write a single word
  • Have to read/write full block containing it
  • Ideally want **sequential** access patterns

• Durability
  • Ideally, file system is in meaningful state upon shutdown
  • This obviously isn't always the case…
Critical Factors in File System Design

• (Hard) Disks Performance !!!
  • Maximize sequential access, minimize seeks

• Open before Read/Write
  • Can perform protection checks and look up where the actual file resource are, in advance

• Size is determined as they are used !!!
  • Can write (or read zeros) to expand the file
  • Start small and grow, need to make room

• Organized into directories
  • What data structure (on disk) for that?

• Need to carefully allocate / free blocks
  • Such that access remains efficient
FAT: File Allocation Table

- MS-DOS, 1977
- Still widely used!
**FAT (File Allocation Table)**

- Assume (for now) we have a way to translate a path to a “file number”
  - i.e., a directory structure

- Disk Storage is a collection of Blocks
  - Just hold file data (offset $o = <B, x>$)

- Example: `file_read 31, < 2, x >`
  - Index into FAT with file number
  - Follow linked list to block
  - Read the block from disk into memory
**FAT (File Allocation Table)**

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  - Could require scan to find
  - Or, could use a free list
FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  - Could require scan to find
  - Or, could use a free list
FAT (File Allocation Table)

• Where is FAT stored?
  • On disk

• How to format a disk?
  • Zero the blocks, mark FAT entries “free”

• How to quick format a disk?
  • Mark FAT entries “free”

• Simple: can implement in device firmware
FAT Discussion

Suppose you start with the file number:

• Time to find block?
• Block layout for file?
• Sequential access?
• Random access?
• Fragmentation?
• Small files?
• Big files?
How to get the File Number?

• Look up in *directory structure*

• A directory is a file containing `<file_name : file_number>` mappings
  • File number could be a file or another directory
  • Operating system stores the mapping in the directory in a format it interprets
  • Each `<file_name : file_number>` mapping is called a directory entry

• Process isn’t allowed to read the raw bytes of a directory
  • The `read` function doesn’t work on a directory
  • Instead, see `readdir`, which iterates over the map without revealing the raw bytes

• Why shouldn’t the OS let processes read/write the bytes of a directory?
FAT: Directories

• A directory is a file containing `<file_name: file_number>` mappings

• Free space for new entries

• In FAT: file attributes are kept in directory (!!!)

• Each directory a linked list of entries

• Where do you find root directory ("/")?
FAT Directory Structure

• How many disk accesses to resolve “/my/book/count”?
  • Read in file header for root (fixed spot on disk)
  • Read in first data block for root
    • Table of file name/index pairs. Search linearly – ok since directories typically very small
  • Read in file header for “my”
  • Read in first data block for “my”; search for “book”
  • Read in file header for “book”
  • Read in first data block for “book”; search for “count”
  • Read in file header for “count”

• Current working directory: Per-address-space pointer to a directory used for resolving file names
  • Allows user to specify relative filename instead of absolute path (say CWD=“/my/book” can resolve “count”)

7/27/2020 Kumar CS 162 at UC Berkeley, Summer 2020
Many Huge FAT Security Holes!

• FAT has no access rights

• FAT has no header in the file blocks

• Just gives an index into the FAT
  • (file number = block number)
Conclusion: I/O Devices

• I/O Devices Types:
  • Many different speeds (0.1 bytes/sec to GBytes/sec)
  • Different Access Patterns:
    • Block Devices, Character Devices, Network Devices
  • Different Access Timing:
    • Blocking, Non-blocking, Asynchronous

• I/O Controllers: Hardware that controls actual device
  • Processor Accesses through I/O instructions, load/store to special physical memory

• Notification mechanisms
  • Interrupts
  • Polling: Report results through status register that processor looks at periodically

• Device drivers interface to I/O devices
  • Provide clean Read/Write interface to OS above
  • Manipulate devices through PIO, DMA & interrupt handling
  • Three types: block, character, and network
Conclusion: Storage Devices

• Disk Performance:
  • Queuing time + Controller + Seek + Rotational + Transfer
  • Rotational latency: on average ½ rotation
  • Transfer time: spec of disk depends on rotation speed and bit storage density

• Devices have complex interaction and performance characteristics
  • Response time (Latency) = Queue + Overhead + Transfer
    • Effective BW = BW * T/(S+T)
  • HDD: Queuing time + controller + seek + rotation + transfer
  • SDD: Queuing time + controller + transfer (erasure & wear)

• Systems (e.g., file system) designed to optimize performance and reliability
  • Relative to performance characteristics of underlying device

• Bursts & High Utilization introduce queuing delays
Conclusion: File Systems

• File System:
  • Transforms blocks into Files and Directories
  • Optimize for size, access and usage patterns
  • Maximize sequential access, allow efficient random access
  • Projects the OS protection and security regime (UGO vs ACL)

• Naming: translating from user-visible names to actual sys resources
  • Directories provide naming for local file systems
  • Linked or tree structure stored in files

• Components: directory, index, storage blocks, free list

• File Allocation Table (FAT) – simple and primitive
  • I-number = FAT index, FAT 1–1 with disk blocks, file is singly-link list of FAT=Blocks, directory is essentially file of <name, index, attributes> at known location
  • Linear search – for block, for i-number