CS162 Operating Systems and Systems Programming Lecture 13

Disk/SSDs, File Systems

October 10, 2012
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Quiz 13.1: Synchronization

- Q1: True _ False _ During a critical section, a thread can be preempted by the CPU dispatcher
- Q2: True _ False _ If we use interrupts to implement locks we need to enable interrupts before going to sleep (in the lock() primitive)
- Q3: True _ False _ The order of sem.P() and sem.V() in a program is commutative
- Q4: True _ False _ With Mesa monitors, the program needs to check again the condition (on which it went to sleep) after waking up
- Q5: True _ False _ In a database (think of the Readers/Writers problem), a user can read while another one writes

Quiz 13.1: Synchronization

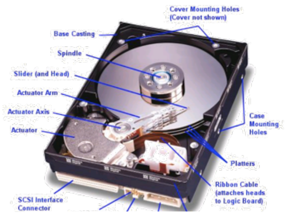
- Q1: True x False _ During a critical section, a thread can be preempted by the CPU dispatcher
- Q2: True _ False x If we use interrupts to implement locks we need to enable interrupts before going to sleep (in the lock() primitive)
- Q3: True x False _ The order of sem.P() and sem.V() in a program is commutative
- Q4: True x False _ With Mesa monitors, the program needs to check again the condition (on which it went to sleep) after waking up
- Q5: True _ False x In a database (think of the Readers/Writers problem), a user can read while another one writes

Goals for Today

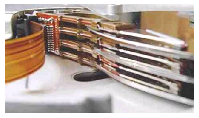
- Disks and SSDs
- Important System Properties
- File Systems
  - Structure, Naming, Directories, Caching

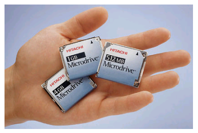
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Hard Disk Drives (HDDs)



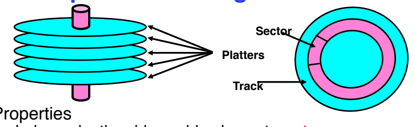
Western Digital Drive
http://www.storagereview.com/guide/


Read/Write Head Side View

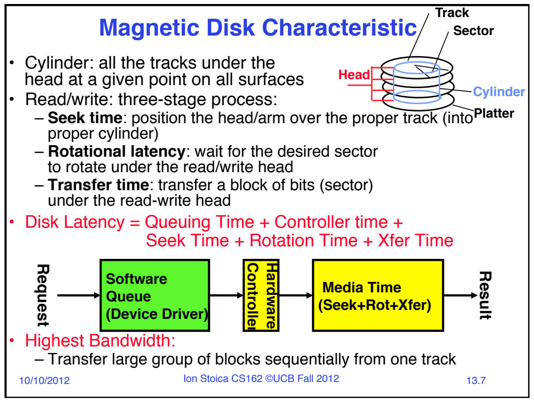

IBM/Hitachi Microdrive

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

Properties of a Magnetic Hard Disk

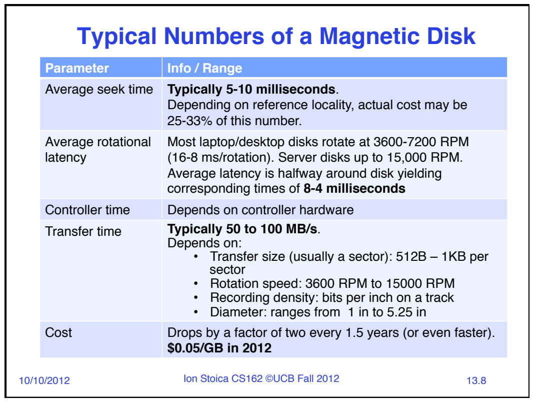


- Properties
  - Independently addressable element: sector
    - » OS always transfers groups of sectors together—"blocks"
  - A disk can access directly any given block either sequentially or randomly.
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
- Zoned bit recording
  - Constant bit density: more bits (sectors) on outer tracks
  - Apple][gs/old Macs: speed varies with track location



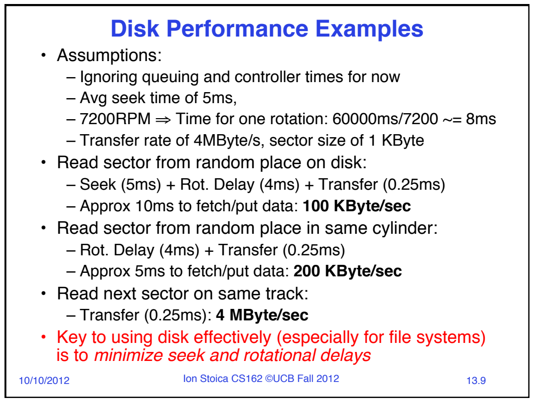
Magnetic Disk Characteristic

- Cylinder: all the tracks under the head at a given point on all surfaces
- Read/write: three-stage process:
  – **Seek time**: position the head/arm over the proper track (into proper cylinder)
  – **Rotational latency**: wait for the desired sector to rotate under the read/write head
  – **Transfer time**: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queuing Time + Controller time + Seek Time + Rotation Time + Xfer Time
- Highest Bandwidth:
  – Transfer large group of blocks sequentially from one track



Typical Numbers of a Magnetic Disk

Parameter	Info / Range
Average seek time	**Typically 5-10 milliseconds**. Depending on reference locality, actual cost may be 25-33% of this number.
Average rotational latency	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 yielding corresponding times of **8-4 milliseconds**
Controller time	Depends on controller hardware
Transfer time	**Typically 50 to 100 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	Drops by a factor of two every 1.5 years (or even faster). **$0.05/GB in 2012**

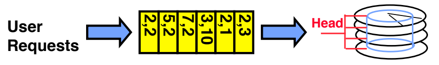


Disk Performance Examples

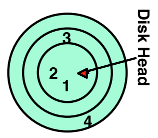
- Assumptions:
  – Ignoring queuing and controller times for now
  – Avg seek time of 5ms,
  – 7200RPM ⇒ Time for one rotation: 60000ms/7200 ~= 8ms
  – Transfer rate of 4MByte/s, sector size of 1 KByte
- Read sector from random place on disk:
  – Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  – Approx 10ms to fetch/put data: **100 KByte/sec**
- Read sector from random place in same cylinder:
  – Rot. Delay (4ms) + Transfer (0.25ms)
  – Approx 5ms to fetch/put data: **200 KByte/sec**
- Read next sector on same track:
  – Transfer (0.25ms): **4 MByte/sec**
- Key to using disk effectively (especially for file systems) is to *minimize seek and rotational delays*

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?
  - Request denoted by (track, sector)

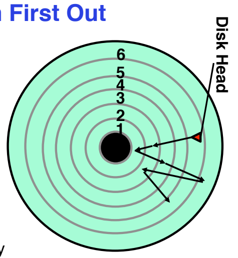


- Scheduling algorithms:
  - First In First Out (FIFO)
  - Shortest Seek Time First
  - SCAN
  - C-SCAN
- In our examples we will ignore the sector
  - Consider only track #



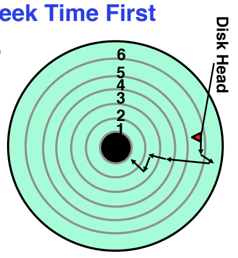
FIFO: First In First Out

- Schedule request in the order they arrive in the queue
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 2, 1, 3, 6, 2, 5
  - Pros: Fair among requesters
  - Cons: Order of arrival may be to random spots on the disk ⇒ Very long seeks

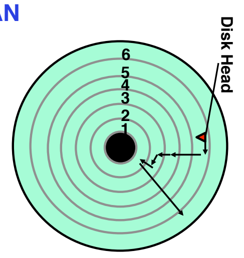


SSTF: Shortest Seek Time First

- Pick the request that's closest to the head on the disk
  - Although called SSTF, include rotational delay in calculation, as rotation can be as long as seek
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 5, 6, 3, 2, 2, 1
- Pros: reduce seeks
- Cons: may lead to starvation

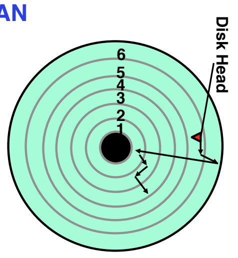


SCAN



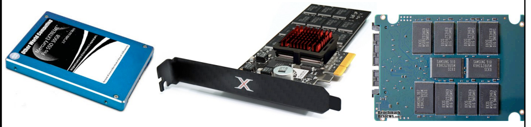
- Implements an Elevator Algorithm: take the closest request in the direction of travel
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head is moving towards center
  - Scheduling order: 5, 6, 3, 2, 2, 1
- Pros:
  - No starvation
  - Low seek
- Cons: favor middle tracks

C-SCAN



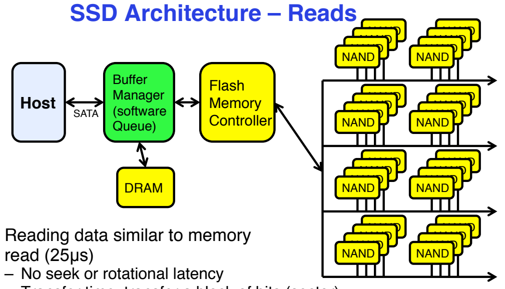
- Like SCAN but only serves request in only one direction
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head only servers request on its way from center towards edge
  - Scheduling order: 5, 6, 1, 2, 2, 3
- Pros:
  - Fairer than SCAN
- Cons: longer seeks on the way back

Solid State Disks (SSDs)



- 1995 – Replace rotating magnetic media with non-volatile memory (battery backed DRAM)
  - Since 2009, use NAND Flash: Single Level Cell (1-bit/cell), Multi-Level Cell (2-bit/cell)
- Sector addressable, but stores 4-64 "sectors" per memory page
- No moving parts (no rotate/seek motors)
  - Eliminates seek and rotational delay (0.1-0.2ms access time)
  - Very low power and lightweight

SSD Architecture – Reads



Reading data similar to memory read (25μs)
- No seek or rotational latency
- Transfer time: transfer a block of bits (sector)
  - » Limited by controller and disk interface (SATA: 300-600MB/s)
- 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 (erase takes ~1.5ms)
  - Controller maintains pool of empty pages by coalescing used sectors (read, erase, write), also reserve some % of capacity
- Typical steady state behavior when SSD is almost full
  - One erase every 64 or 128 writes (e.g., 4KB/32B = 128)
- Write and erase cycles require "high" voltage
  - Damages memory cells, limits SSD lifespan
  - Controller uses ECC, performs wear leveling
- Result is very workload dependent performance
  - Latency = Queuing Time + Controller time (Find Free Block) + Xfer Time
  - Highest BW: Seq. OR Random writes (limited by empty pages)

Rule of thumb: writes 10x more expensive than reads, and erases 10x more expensive than writes

Storage Performance & Price

	Bandwidth (sequential R/W)	Cost/GB	Size
HDD	50-100 MB/s	$0.05-0.1/GB	2-4 TB
SSD[1]	200-600 MB/s (SATA) 6 GB/s (PCI)	$1-1.5/GB	200GB-1TB
DRAM	10-16 GB/s	$5-10/GB	64GB-256GB

[1]http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives/

BW: SSD up to x10 than HDD, DRAM > x10 than SSD
Price: HDD x20 less than SSD, SSD x5 less than DRAM

Quiz 12.3: HDDs and SSDs

- Q1: True _ False _ The block is the smallest addressable unit on a disk
- Q2: True _ False _ An SSD has zero seek time
- Q3: True _ False _ For an HDD, the read and write latencies are similar
- Q4: True _ False _ For an SSD, the read and write latencies are similar
- Q5: Consider the following sequence of requests (2, 4, 1, 8), and assume the head position is on track 9. Then, the order in which SSTF services the requests is _________

Quiz 12.3: HDDs and SSDs

- Q1: True _ False **x** The block is the smallest addressable unit on a disk
- Q2: True **x** False _ An SSD has zero seek time
- Q3: True **x** False _ For an HDD, the read and write latencies are similar
- Q4: True _ False **x** For an SSD, the read and write latencies are similar
- Q5: Consider the following sequence of requests (2, 4, 1, 8), and assume the head position is on track 9. Then, the order in which SSTF services the requests is **(8, 4, 2, 1)**

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), very expensive (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
    - 50-100K writes/page for SLC, 1-10K writes/page for MLC

Administrivia

- Midterm Monday 10/15 at 4-5:30PM in 120 Latimer
  - Closed-book, 1 double-sided page of handwritten notes
  - Covers lectures/readings #1-12 (Mon 10/8) and project one
  - Midterm review session **Friday** 7-9PM in 306 Soda Hall
  - Please remember your **class login**: you need to write it down on the exam!

5min Break

Quiz 13.3: Deadlocks

- Q1: True _ False _ If a resource type (e.g., disk) has multiple instances we cannot have deadlock
- Q2: True _ False _ Deadlock implies starvation
- Q3: True _ False _ Starvation implies deadlock
- Q4: True _ False _ If resources can be preempted from threads we cannot have deadlock
- Q5: True _ False _ Assume a system in which each thread is only allowed to either allocate all resources it needs or none of them. In such a system we can still have deadlock.

Quiz 13.3: Deadlocks

- Q1: True _ False x If a resource type (e.g., disk) has multiple instances we cannot have deadlock
- Q2: True x False _ Deadlock implies starvation
- Q3: True _ False x Starvation implies deadlock
- Q4: True x False _ If resources can be preempted from threads we cannot have deadlock
- Q5: True _ False x Assume a system in which each thread is only allowed to either allocate all resources it needs or none of them. In such a system we can still have deadlock.

Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

- File System Components
  - Disk Management: organizing disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc.

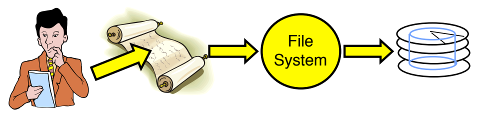
User vs. System View of a File

- User's view:
  - Durable Data Structures

- System's view (system call interface):
  - Collection of Bytes (UNIX)
  - Doesn't matter to system what kind of data structures you want to store on disk!

- System's view (inside OS):
  - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  - Block size ≥ sector size; in UNIX, block size is 4KB

Translating 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: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks

Disk Management Policies

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible mapping of names to files
- Access disk as linear array of sectors.
  - Logical Block Addressing (LBA): Every sector has integer address from zero up to max number of sectors
    - » OS/BIOS must deal with bad sectors
  - Controller translates from address ⇒ physical position
    - » Hardware shields OS from structure of disk

Disk Management Policies (cont’d)

- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk
- Need way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
- Optimize placement of files’ disk blocks to match access and usage patterns

Designing the File System: Access Patterns

- Sequential Access: bytes read in order (“give me the next X bytes, then give me next, etc.”)
  - Most of file accesses are of this flavor
- Random Access: read/write element out of middle of array (“give me bytes i—j”)
  - Less frequent, but still important, e.g., mem. page from swap file
  - Want this to be fast – don’t want to have to read all bytes to get to the middle of the file
- Content-based Access: (“find me 100 bytes starting with ION”)
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don’t provide this; instead, build DBs on top of disk access to index content (requires efficient random access)
  - Example: Mac OSX Spotlight search (do we need directories?)

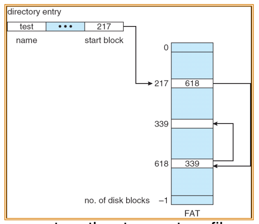
Designing the File System: Usage Patterns

- Most files are small (for example, .login, .c, .java files)
  - A few files are big – executables, swap, .jar, core files, etc.; the .jar is as big as all of your .class files combined
  - However, most files are small – .class, .o, .c, .doc, .txt, etc
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware!
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?

File System Goals

- Maximize sequential performance
- Efiicient random access to file
- Easy management of files (growth, truncation, etc)

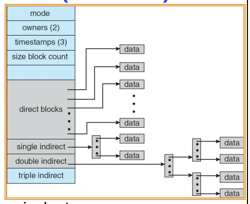
Linked Allocation: File-Allocation Table (FAT)



- MSDOS links pages together to create a file
  - Links not in pages, but in the File Allocation Table (FAT)
    - » FAT contains an entry for each block on the disk
    - » FAT Entries corresponding to blocks of file linked together
  - Access properties:
    - » Sequential access expensive unless FAT cached in memory
    - » Random access expensive always, but *really* expensive if FAT not cached in memory

Multilevel Indexed Files (UNIX 4.1)

- Multilevel Indexed Files: (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files



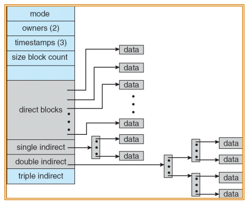
- File hdr contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an "inode" in UNIX
- File Header format:
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)

Multilevel Indexed Files (UNIX 4.1): Discussion

- Basic technique places an upper limit on file size that is approximately 16Gbytes
  - Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
  - Fallacy: today, Facebook gets hundreds of TBs of logs every day!
- Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks
  - On small files, no indirection needed

Example of Multilevel Indexed Files

- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data



- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect blocks (four I/O's per block!)

UNIX BSD 4.2

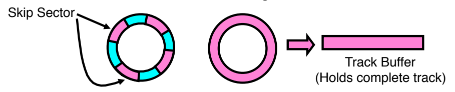
- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from Cray-1 DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space (mentioned next slide)
  - Skip-sector positioning (mentioned in two slides)
- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other

How to Deal with Full Disks?

- In many systems, disks are always full
  - EECS department growth: 300 GB to 1TB in a year (now 10s TB)
  - How to fix? Announce disk space is low, so please delete files?
    - » Don't really work: people try to store their data faster
  - Sidebar: Perhaps we are getting out of this mode with new disks… However, let's assume disks are full for now
- Solution:
  - Don't let disks get completely full: reserve portion
    - » Free count = # blocks free in bitmap
    - » Scheme: Don't allocate data if count < reserve
  - How much reserve do you need?
    - » In practice, 10% seems like enough
  - Tradeoff: pay for more disk, get contiguous allocation
    - » Since seeks so expensive for performance, this is a very good tradeoff

Attack of the Rotational Delay

- Problem: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block!



  - Solution 1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution 2: Read ahead: read next block right after first, even if application hasn't asked for it yet
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

Summary (1/2)

- Hard (Magnetic) Disk Performance:
  - Latency = Queuing time + Controller + Seek + Rotational + Transfer
  - Rotational latency: on average ½ rotation
  - Transfer time: depends on rotation speed and bit density
- SSD Performance:
  - Read: Queuing time + Controller + Transfer
  - Write: Queuing time + Controller (Find Free Block) + Transfer
  - Find Free Block time: depends on how full SSD is (available empty pages), write burst duration, …
  - Limited drive lifespan

Summary (2/2)

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
- File (and directory) defined by header, called "inode"
- Multilevel Indexed Scheme
  - Inode contains file info, direct pointers to blocks,
  - indirect blocks, doubly indirect, etc..
- 4.2 BSD Multilevel index files
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization