Fast File, Log and Journaling File Systems

cs262a, Lecture 3

Ali Ghodsi and Ion Stoica

(based on presentations from John Kubiatowicz, UC Berkeley, and Arvind Krishnamurthy, from University of Washington)

From Last Lecture

Unix vs. System R: Philosophy

Bottom-Up (elegance of system) vs. Top-Down (elegance of semantics)

UNIX main function: *present hardware to computer programmers*

- small *elegant* set of mechanisms, and abstractions for developers (i.e. C programmers)

System R: *manage data for application programmer*

- complete system that insulated programmers (i.e. SQL + scripting) from the system, while guaranteeing clearly defined *semantics* of data and queries.

Unix vs. System R: Philosophy

Bottom-Up (elegance of system) vs. Top-Down (elegance of semantics)

Affects where the complexity goes: to system, or end-programmer?

Which one is better? In what environments?

Turing Awards

Relational databases related:

» 1981: Edgar Codd

» 1988: Jim Gray

» 2015: Mike Stonebraker

Unix related

» 1983: Ken Thomson and Dennis Ritchie

» 1990: Fernando Corbato

Today's Papers

A Fast File System for UNIX
Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry. Appears in *ACM Transactions on Computer Systems* (TOCS), Vol. 2, No. 3, August 1984, pp 181-197

Analysis and Evolution of Journaling File Systems
Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (ATEC '05), 2005

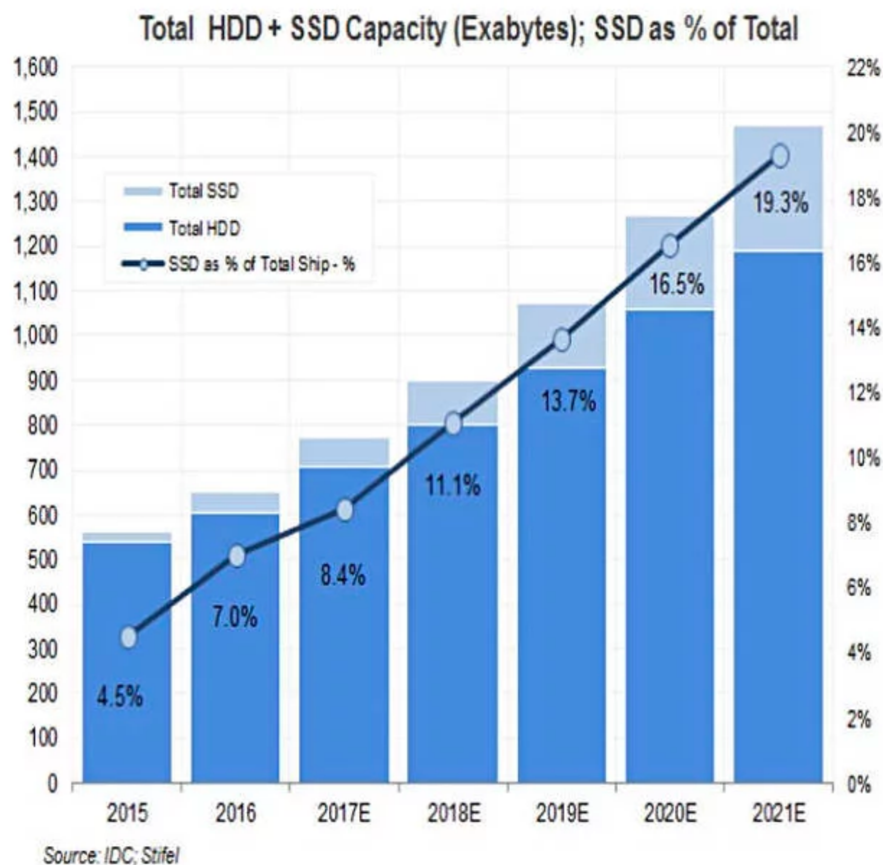
System design paper and system analysis paper

Why Today's Papers?

After all SSDs are taking over…

Why Today's Papers?

HDDs still used to store very large data sets cost effectively



Why Today's Papers?

HDDs still used to store very large data sets cost effectively

More and more distributed storage systems

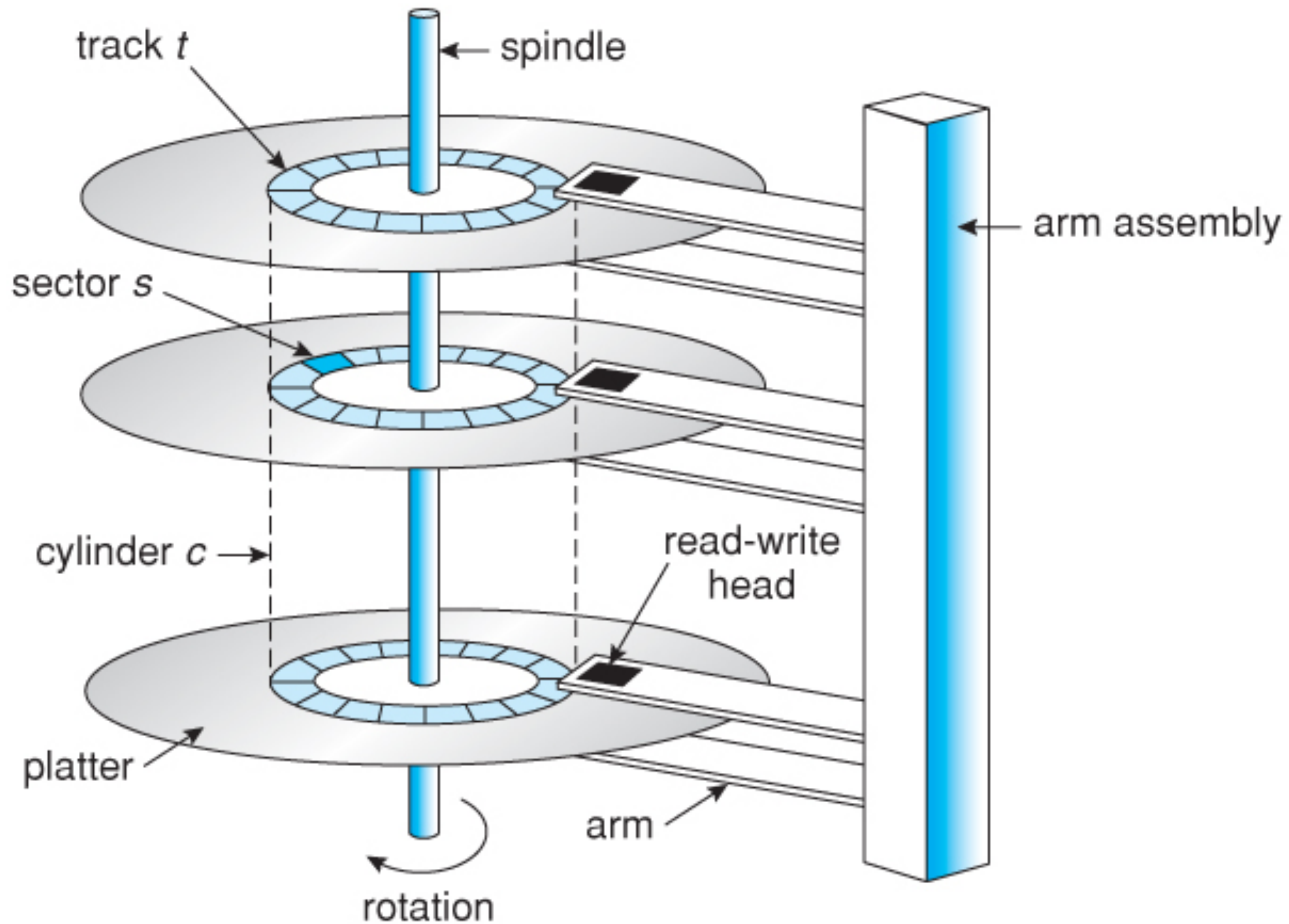
» Performance pattern resemble disks, e.g.,

- Low throughput for random access, high throughput for sequential access

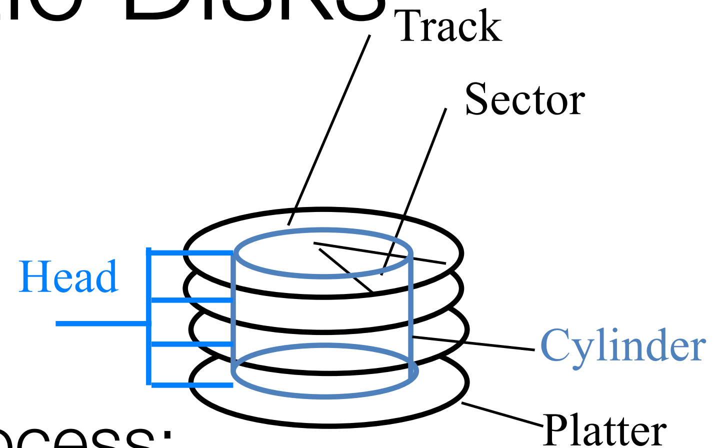
Great examples of system engineering

» Valuable lessons for other application domains

Review: Magnetic Disks



Review: Magnetic Disks



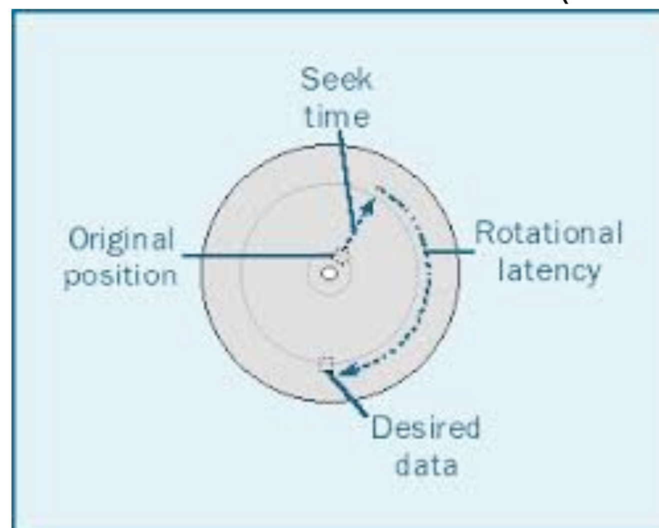
Cylinders: all the tracks under the head at a given point on all surface

Read/write data is a three-stage process:

» Seek time: position the head/arm over the proper track

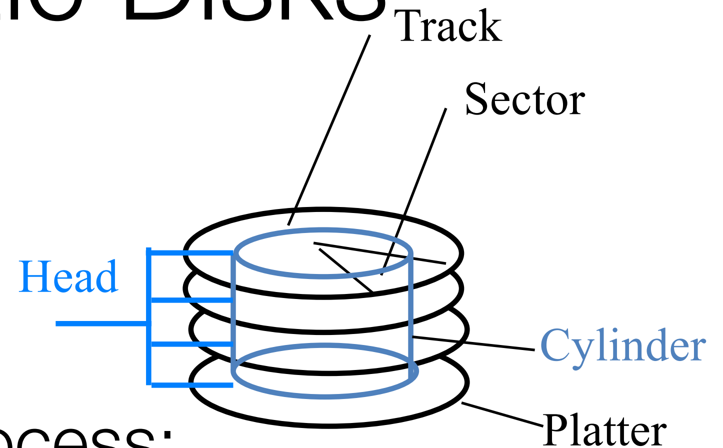
» Rotational latency: wait for desired sector to rotate under r/w head

» Transfer time: transfer a block of bits (sector) under r/w head



Seek time = 4-8m
One rotation = 1-2ms
(3600-7200 RPM)

Review: Magnetic Disks



Cylinders: all the tracks under the head at a given point on all surface

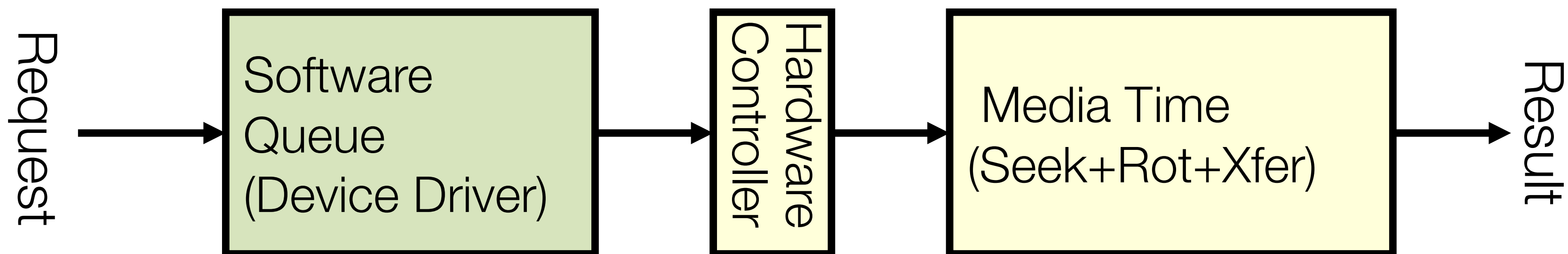
Read/write data is a three-stage process:

» Seek time: position the head/arm over the proper track

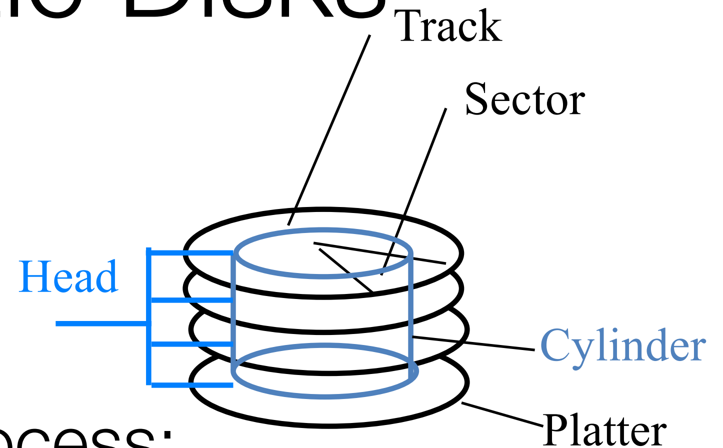
» Rotational latency: wait for desired sector to rotate under r/w head

» Transfer time: transfer a block of bits (sector) under r/w head

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



Review: Magnetic Disks



Cylinders: all the tracks under the head at a given point on all surface

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

Highest Bandwidth: Transfer large group of blocks sequentially from one track

Historical Perspective

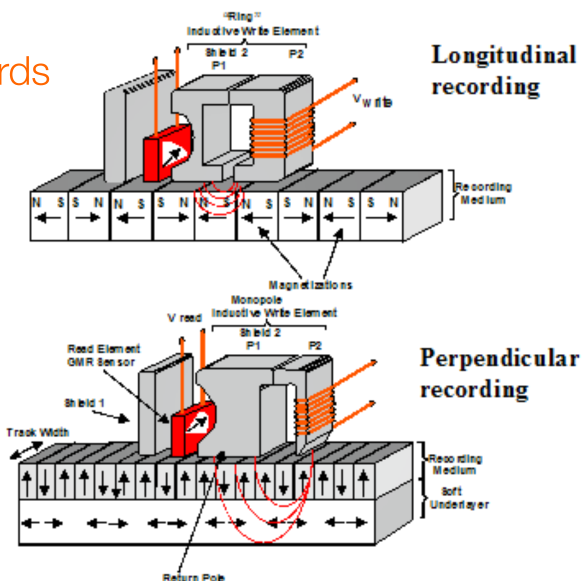
1956 IBM Ramac — early 1970s Winchester
- » Developed for mainframe computers, proprietary interface
- » Steady shrink in form factor from 24in to 17in

1970s developments
- » 8 inch floppy disk form factor (microcode into mainframe)
- » Emergence of industry standard disk interfaces

1980s: PCs, first generation workstations, and client server computing
- » Centralized storage on file server
  - Accelerate disk downsizing: 8inch inch to 5.25 inch
- » Mass market disk drives become a reality
  - Industry standards: SCSI, IDI; end of proprietary standards
  - 5.25 inch to 3.5 inch drives PCs

1990s: Laptops → 2.5inch

2000s: Switch to perpendicular (vs longitudinal) recording
- » 2007: 1TB
- » 2009: 2TB
- » 2016: 12TB



© 2005, Hitachi Global Storage Technologies

Recent: Western Digital Ultrastar

12 TB (April, 2017)

8 platters, 16 heads

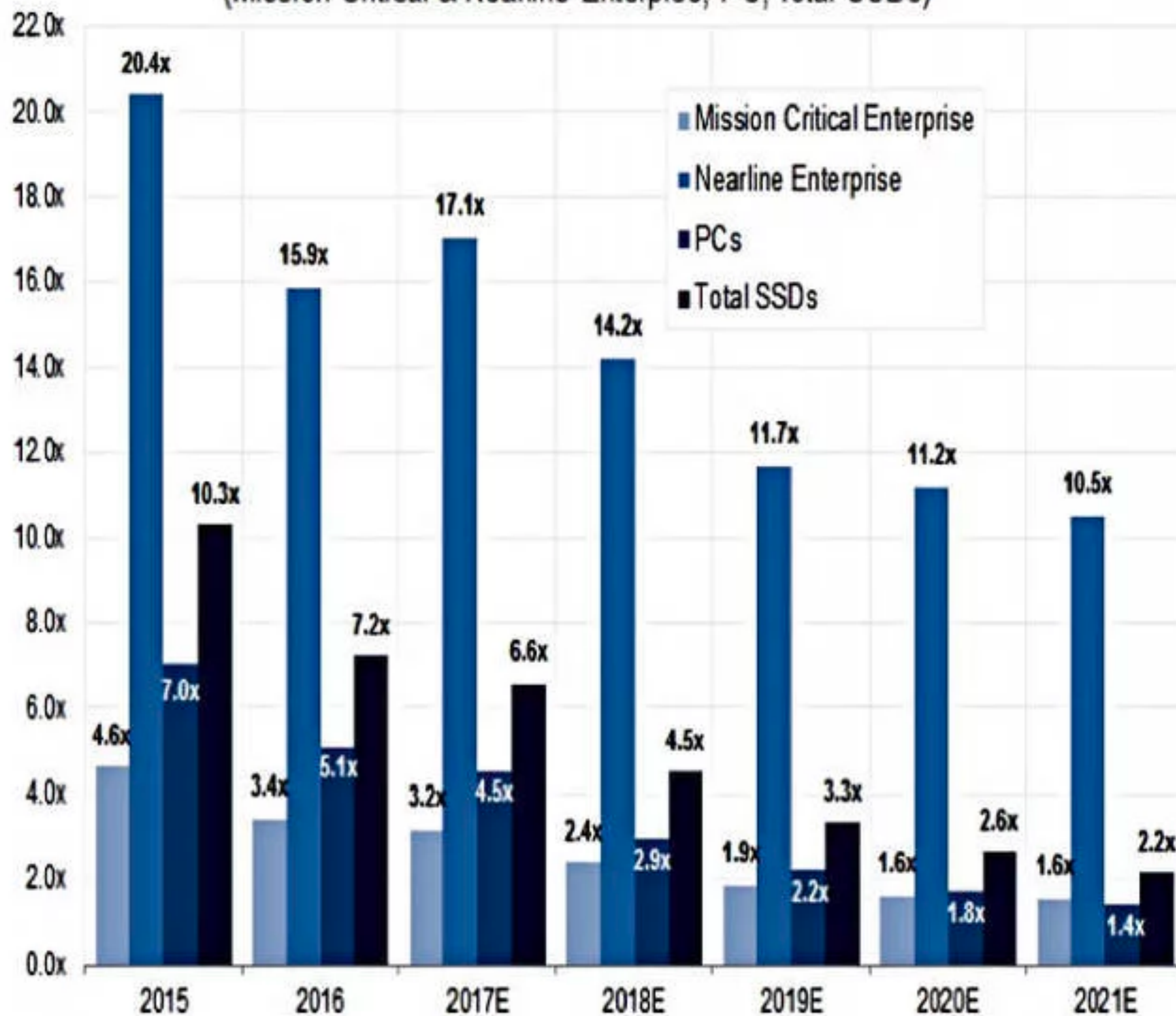
7200 RPMs

Max 255MB/s transfer rate

Average latency: 4.2ms

Seek time: 8ms

Helium filled: reduce friction and power usage



Total SSD $/TB Premium vs. HDDs
(Mission-Critical & Nearline Enterpise, PC, Total SSDs)
22.0x
20.0x
18.0x
16.0x
14.0x
12.0x
10.0x
8.0x
6.0x
4.0x
2.0x
0.0x
Mission Critical Enterprise
Nearline Enterprise
PCs
Total SSDs
20.4x
4.6x
7.0x
10.3x
15.9x
3.4x
5.1x
7.2x
17.1x
3.2x
4.5x
6.6x
14.2x
2.4x
2.9x
4.5x
11.7x
1.9x
2.2x
3.3x
11.2x
1.6x
1.8x
2.6x
10.5x
1.6x
1.4x
2.2x
2015
2016
2017E
2018E
2019E
2020E
2021E
Source: IDC; Stifel

Largest SSDs

60TB (Seagate, August 16)

Dual 16Gbps

Seq reads: 1.5GB/s

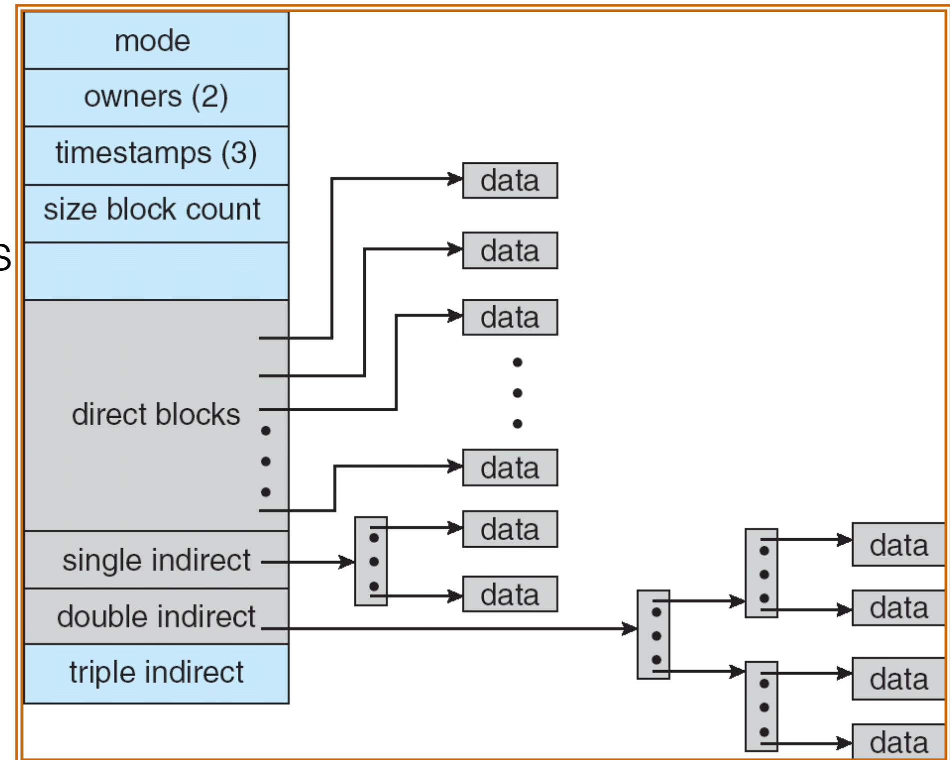
Seq writes: 1GB/s

Random Read Ops (IOPS): 150K

Review: Unix File Systems

i-node: per-file metadata (unique per file)

- » contains: ownership, permissions, timestamps, ~10 data-block pointers
- » i-nodes form an array, indexed by "i-number" – so each i-node has a unique i-number
- » Array is explicit for FFS, implicit for LFS (its i-node map is cache of i-nodes indexed by i-number)



Indirect blocks:

- » i-node only holds a small number of data block pointers (direct pointers)
- » For larger files, i-node points to an indirect block containing 1024 4-byte entries in a 4K block
- » Each indirect block entry points to a data block
- » Can have multiple levels of indirect blocks for even larger files

Original Unix File System

Simple and elegant, but slow

- » 20KB/s, only 2% from disk theoretical bandwidth (1MB/s)

Challenges:

- » Blocks too small: 512 bytes (matched sector size)
- » Many seeks
  - Consecutive blocks of files not close together
  - All i-nodes at the beginning of the disk, all data after that
  - i-nodes of directory not close together
- » Transfer only one block at a time

Old File System

Increase block size to 1024: over 2x higher throughput (why?)

- » 2x higher transfer rate
- » Fewer indirect blocks, so fewer seeks

Challenges: in time, free list becomes random so performance degrades (because of seeks)

FFS Changes

4096 or 8192 byte block size (why not larger?)

Use cylinder groups (why?)

- » Each contains superblock, i-nodes, bitmap of free blocks, usage summary info

Blocks divided into small fragments (why?)

Don’t fill entire disk (why?)

Parametrized Model

New file try to allocate new blocks on same cylinder (why?)

Don’t allocate consecutive blocks (why?)

- » How do you compute the space between two blocks that are consecutive in the file?

Layout

Principles:

- » Locality of reference to minimize seek latency
- » Improve the layout of data to optimize larger transfers

inodes of files in same directory together (why?)

New directories in cylinder groups that have higher than average free blocks (why?)

Try to store all data blocks of a file in the same cylinder group

- » Preferably at rotationally optimal positions in the same cylinder

Move to other cylinder group when files grow (which one?)
Greedy algorithm:

- » same cylinder
- » same cylinder group
- » quadratically hash cylinder group number
- » exhaustive search;

FFS Results

20-40% of disk bandwidth for large reads/writes

10-20x original UNIX speeds

Size: 3800 lines of code vs. 2700 in old system

10% of total disk space overhead

FFS System Interface Enhancements

Long file names: from 14 to 255 characters

Advisory file locks (shared exclusive):

- » Requested by the program: shared or exclusive
- » Effective only if all programs accessing the same file use them
- » Process id of holder stored with lock => can reclaim the lock if process is no longer around

Long file names: from 14 to 255 characters

Symbolic links (contrast to hard links)

Atomic rename capability

- » The only atomic read-modify-write operation, before this there was none

Disk quotas

FFS Summary

3 key features:

- » Optimize FS implementation for hardware
- » Measurement-driven design decisions
- » Locality “wins”

Limitations:

- » Measurements derived from a single installation
- » Ignored technology trends

Lessons:

- » Don’t ignore underlying hardware characteristics
- » Contrasting research approaches: improve what you’ve got vs. design something new (e.g., Log File Systems)

Log Structured & Journaling File Systems

Log-Structured/Journaling File System

Radically different file system design

Technology motivations:

» CPUs outpacing disks: I/O becoming more-and-more a bottleneck

» Large RAM: file caches work well, making most disk traffic writes

Problems with (then) existing file systems:

» Lots of little writes

» Synchronous: wait for disk in too many places

» 5 seeks to create a new file (rough order):

1. file i-node (create)
2. file data
3. directory entry
4. file i-node (finalize)
5. directory i-node (modification time)

LFS Basic Idea

Log all data and metadata with large, sequential writes

Keep an index on log's contents

Use large memory to provide fast access through caching

Data layout on disk has "temporal locality" (good for writing), rather than "logical locality" (good for reading)

» Why is this a good idea?

Two Potential Problems

Floating inodes

Wrap-around: what happens when running out of space?

- » No longer any big, empty runs available
- » How to prevent fragmentation?

Floating inodes

When you create a small file (less than a block):

- » Write data block to memory log
- » Write file inode to memory log
- » Write directory block to memory log
- » Write directory inode to memory log

When memory accumulates to say 1MB or 30s have elapsed, write log to disk as a single write

No seek for writes, but inodes are now floating!

Solving floating inode problem

Need to keep track of current position of inodes
- » Solution: use an “inode-map”!

inode-map could be large (as many entries as there are files in the file system)
- » Break inode-map into chunks and cache them

Write out on the log those chunks that have changed

But, how do do you find the chunks of inode-map?

Finding chunks of inode-map?

Solution inode-map-map: map of inode map

Have we solved the problem now?

Yes!

- inode-map-map is small enough to be always cached in memory
- it is small enough to be written to a fixed (and small position) on the disk (checkpoint region)
- write the inode-map-map when file system is unmounted

LFS Disk Wrap-Around

Compact live info to open up large runs of free space

» Problem: long-lived information gets copied over-and-over

Thread log through free spaces

» Problem: disk fragments, causing I/O to become inefficient again

Solution: *segmented log*

» Divide disk into large, fixed-size segments

» Do compaction within a segment; thread between segments

» When writing, use only clean segments (i.e. no live data)

» Occasionally clean segments: read in several, write out live data in compacted form, leaving some fragments free

» Collect long-lived info into segments that never need to be cleaned

» No free list or bit map (as in FFS), only a list of clean segments

Analysis and Evolution of Journaling File Systems

Write-ahead logging: commit data by writing it to log, synchronously and sequentially

Later move data to its normal (FFS-like) location
- » called *checkpointing*; makes room in the (circular) journal

Better for random writes, slightly worse for big sequential writes

All reads go the the fixed location blocks, not the journal
- » Journal only read for crash recovery and checkpointing

Much better than FFS (fsck) for crash recovery (Why?)

Ext3/ReiserFS/Ext4 filesystems are the main ones in Linux

How do you commit data and metadata?

Three modes:

- » Writeback mode
- » Ordered mode
- » Data journaling mode

Writeback Mode

Journal only metadata

Write back data and metadata independently

Metadata may have dangling references after a crash (if crash between metadata and data writes)

Ordered Mode

Journal only metadata, but always write data blocks before their referring metadata is journaled

This mode generally makes the most sense and is used by Windows NTFS and IBM's JFS

Data Journaling Mode

Write both data and metadata to the journal

Huge increase in journal traffic; plus have to write most blocks twice, once to the journal and once for checkpointing (why not all?)

JFS Crash Recovery

Load superblock to find tail/head of the log

Scan log to detect whole committed transactions
- » There is a commit record

Replay log entries to bring in-memory data structures up to date
- » This is called “redo logging” and entries must be “idempotent”

Playback is oldest to newest; tail of the log is the place where checkpointing stopped

Semantic Block-level Analysis (SBA)

Idea: interpose special disk driver between the file system and real disk driver

Pros:

- captures ALL disk traffic
- can use with a black-box file system (no source code needed and can even use via VMWare for another OS)
- more insightful than just performance benchmark

Cons:

- Must understand disk layout
- Really only useful for writes

To use well, drive file system with smart applications that test certain features of the file system (to make the inference easier)

Semantic Trace Playback (STP)

Uses two kinds of interposition:

- » SBA driver that produces a trace, and
- » user-level library between app and real file system

User-level library traces dirty blocks and app calls to fsync

Playback:

- » Given the two traces, STP generates a timed set of commands to the raw disk device – this sequence can be timed to understand performance implications

Semantic Trace Playback (STP)

Claim:

- Faster to modify the trace than to modify the file system and simpler and less error-prone than building a simulator

Limited to simple FS changes

Best example usage:

- Showing that dynamically switching between ordered mode and data journaling mode actually gets the best overall performance (use data journaling for random writes)

LFS Summary

CPUs outpacing disk speeds; implies that I/O is becoming more-and-more of a bottleneck

Write FS information to a log and treat the log as the truth; rely on in-memory caching for speed

Hard problem: finding/creating long runs of disk space to (sequentially) write log records to

- » Solution: clean live data from segments, picking segments to clean based on a cost/benefit function

Limitations:

- » Assumes that files get written in their entirety; else would get intra-file fragmentation in LFS
- » If small files "get bigger" then how would LFS compare to UNIX?

LFS Observations

Interesting point:

» LFS' efficiency isn't derived from knowing the details of disk geometry; implies it can survive changing disk technologies (such variable number of sectors/track) better

Lessons:

» Rethink your basic assumptions about what's primary and what's secondary in a design

» In this case, they made the log become the truth instead of just a recovery aid

Backup

Exercise

How would you design differently file-system for SSDs or 3DXPoint?

LFS Segment Cleaning

How to clean a segment?

- » Segment summary block contains map of the segment
- » Must list every i-node and file block
- » For file blocks you need {i-number, block #}

Which segment to clean?

Keep estimate of free space in each segment to help find segments with lowest utilization

Always start by looking for segment with utilization=0…

If utilization of segments being cleaned is U:

- write cost = (bytes read & written)/(new bytes written)
  = 2/(1 – U), if U > 0
- write cost increases with U: U = .9 => cost = 20!
- need a cost <= 4; → U <= .5