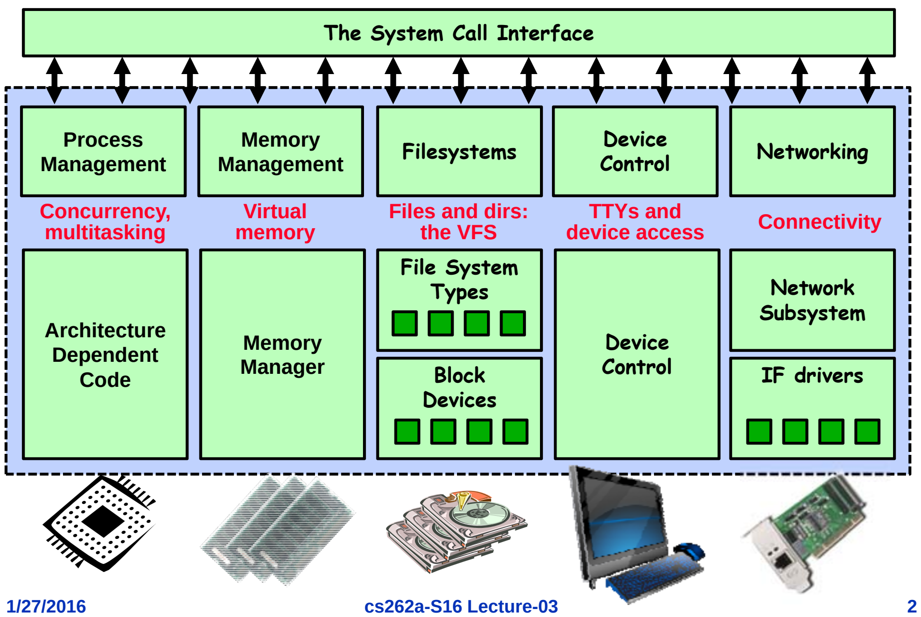
EECS 262a
Advanced Topics in Computer Systems
Lecture 3

Filesystems
January 27th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Kernel Device Structure

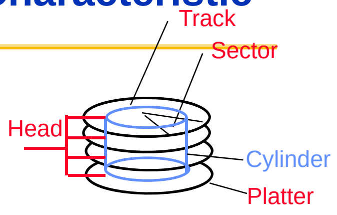


Today's Papers

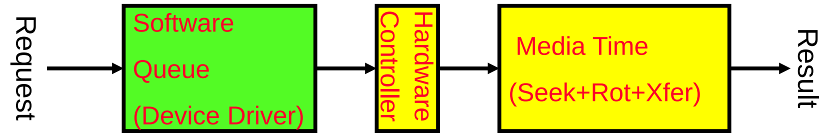
- A Fast File System for UNIX
  Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry. Appears in *ACM Transactions on Computer Systems* (TOCS), Vol. 2, No. 3, August 1984, pp 181-197
- Analysis and Evolution of Journaling File Systems
  Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, Appears in *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (ATEC '05), 2005

- System design paper and system analysis paper
- Thoughts?

Review: Magnetic Disk Characteristic



- Cylinder: 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 (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 = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time



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

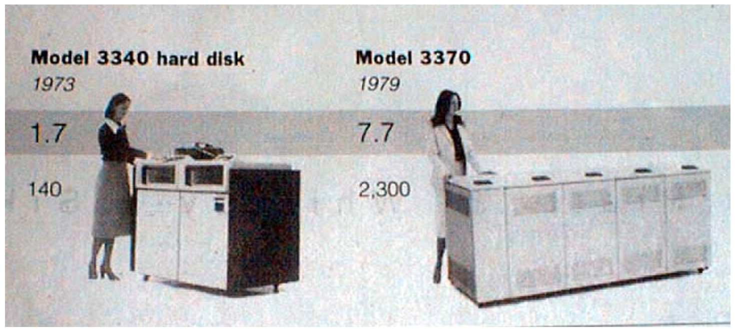
Historical Perspective

- 1956 IBM Ramac — early 1970s Winchester
  - Developed for mainframe computers, proprietary interfaces
  - Steady shrink in form factor: 27 in. to 14 in.
- Form factor and capacity drives market more than performance
- 1970s developments
  - 5.25 inch floppy disk formfactor (microcode into mainframe)
  - Emergence of industry standard disk interfaces
- Early 1980s: PCs and first generation workstations
- Mid 1980s: Client/server computing
  - Centralized storage on file server
    - » accelerates disk downsizing: 8 inch to 5.25
  - Mass market disk drives become a reality
    - » industry standards: SCSI, IPI, IDE
    - » 5.25 inch to 3.5 inch drives for PCs, End of proprietary interfaces
- 1900s: Laptops => 2.5 inch drives
- 2000s: Shift to perpendicular recording
  - 2007: Seagate introduces 1TB drive
  - 2009: Seagate/WD introduces 2TB drive
- 2014: Seagate announces 8TB drives

Disk History

Data density Mbit/sq. in.

Capacity of Unit Shown Megabytes

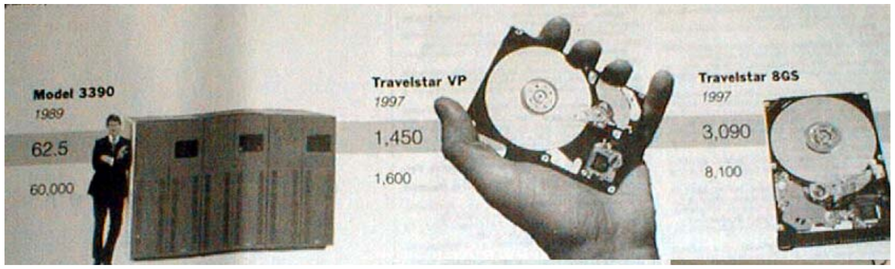


1973:
1. 7 Mbit/sq. in
140 MBytes

1979:
7. 7 Mbit/sq. in
2,300 MBytes

source: New York Times, 2/23/98, page C3,
“Makers of disk drives crowd even mroe data into even smaller spaces”

Disk History



1989:
63 Mbit/sq. in
60,000 MBytes

1997:
1450 Mbit/sq. in
2300 MBytes

1997:
3090 Mbit/sq. in
8100 MBytes

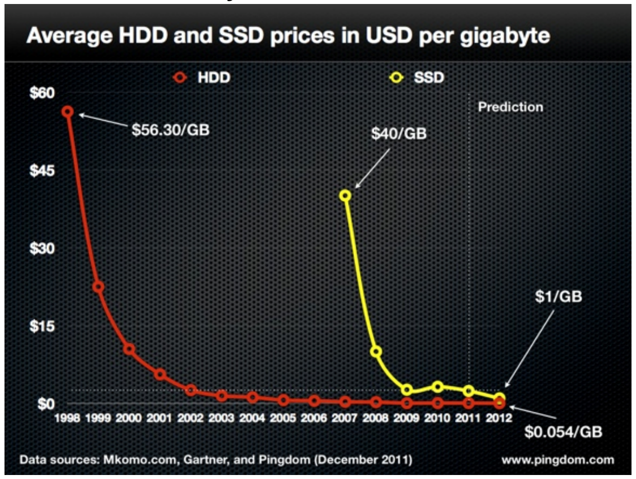
source: New York Times, 2/23/98, page C3,
“Makers of disk drives crowd even mroe data into even smaller spaces”

Recent: Seagate Enterprise (2014)

- 8TB! > 1.33 Tb/in^2 (announced Nov 2015)
- 6 (3.5”) platters, 2 heads each
- Perpendicular recording (not SMR!)
- 7200 RPM, 4.16ms latency
- 237MB/sec sustained transfer speed
- 256MB cache
- Error Characteristics:
  - MBTF: 2 x 10^6 hours
  - Bit error rate: 10^{-15}
- Special considerations:
  - Normally need special “bios” (EFI): Bigger than easily handled by 32-bit OSes.
  - Seagate provides special “Disk Wizard” software that virtualizes drive into multiple chunks that makes it bootable on these OSes.

Contrarian View

- FFS doesn't matter anymore!



- What about Journaling? Is it still relevant?

Storage Performance & Price

	Bandwidth (sequential R/W)	Cost/GB	Size
HHD	50-100 MB/s	\$0.05-0.1/GB	2-8 TB
SSD[1]	200-500 MB/s (SATA) 6 GB/s (PCI)	\$1.5-5/GB	200GB-1TB
DRAM	10-16 GB/s	\$5-10/GB	64GB-256GB

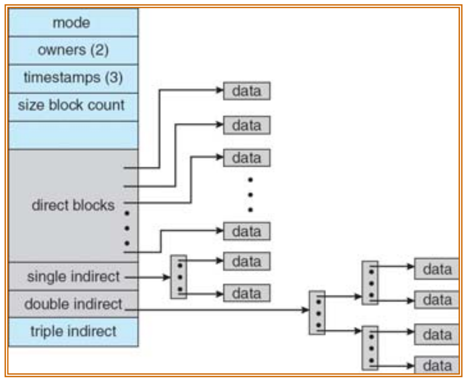
[1]http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives/

BW: SSD up to x10 than HDD, DRAM > x10 than SSD

Price: HDD x30 less than SSD, SSD x4 less than DRAM

Filesystems Background

- i-node: structure for per-file metadata (unique per file)
  - contains: ownership, permissions, timestamps, about 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

A Fast File System for UNIX

- Original UNIX FS was simple and elegant, but slow
- Could only achieve about 20 KB/sec/arm; ~2% of 1982 disk bandwidth
- Problems:
  - Blocks too small
    - » 512 bytes (matched sector size)
  - Consecutive blocks of files not close together
    - » Yields random placement for mature file systems
  - i-nodes far from data
    - » All i-nodes at the beginning of the disk, all data after that
  - i-nodes of directory not close together
  - no read-ahead
    - » Useful when sequentially reading large sections of a file

FFS Changes

- Aspects of new file system:
  - 4096 or 8192 byte block size (why not larger?)
  - large blocks and small fragments
  - disk divided into cylinder groups
  - each contains superblock, i-nodes, bitmap of free blocks, usage summary info
  - Note that i-nodes are now spread across the disk:
    - » Keep i-node near file, i-nodes of a directory together (shared fate)
  - Cylinder groups ~ 16 cylinders, or 7.5 MB
  - Cylinder headers spread around so not all on one platter
- Two techniques for locality:
  - Lie – don't let disk fill up (in any one area)
  - Paradox: to achieve locality, must spread unrelated things far apart
  - Note: new file system got 175KB/sec because free list contained sequential blocks (it did generate locality), but an old system has randomly ordered blocks and only got 30 KB/sec (fragmentation)

FFS Locality Techniques

- Goals
  - Keep directory within a cylinder group, spread out different directories
  - Allocate runs of blocks within a cylinder group, every once in a while switch to a new cylinder group (jump at 1MB)
- Layout policy: global and local
  - Global policy allocates files & directories to cylinder groups – picks "optimal" next block for block allocation
  - Local allocation routines handle specific block requests – select from a sequence of alternative if need to

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 unusable (except at 50% performance price)
- Could have done more; later versions do

FFS System Interface Enhancements

- Really a second mini-paper!
- Long file names (14 → 255 characters)
- Advisory file locks (shared or exclusive)
  - Process id of holder stored with lock => can reclaim the lock if process is no longer around
- Symbolic links (contrast to hard links)
- Atomic rename capability
  - The only atomic read-modify-write operation, before this there was none
- Disk quotas
- Could probably have gotten copy-on-write to work to avoid copying data from user → kernel (would need to copies only for parts that are not page aligned)
- Over-allocation would save time; return unused allocation later Advantages:
  - 1) less overhead for allocation
  - 2) more likely to get sequential blocks

FFS Summary

- 3 key features:
  - Parameterize FS implementation for the hardware it's running on
  - Measurement-driven design decisions
  - Locality "wins"
- Major flaws:
  - Measurements derived from a single installation
  - Ignored technology trends
- A lesson for the future: don't ignore underlying hardware characteristics
- Contrasting research approaches: improve what you've got vs. design something new

Is this a good paper?

- What were the authors' goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

BREAK

Quick Aside: Log-Structured/Journaling File System

- Radically different file system design
- Technology motivations:
  - CPUs outpacing disks: I/O becoming more-and-more of a bottleneck
  - Large RAM: file caches work well, making most disk traffic writes
- Problems with (then) current file systems:
  - Lots of little writes
  - Synchronous: wait for disk in too many places – makes it hard to win much from RAIDs, too little concurrency
  - 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 efficient, large, sequential writes
- Treat the log as the truth, but keep an index on its contents
- Rely on a 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 better? Because caching helps reads but not writes!
- Two potential problems:
  - Log retrieval on cache misses
  - Wrap-around: what happens when end of disk is reached?
    - » No longer any big, empty runs available
    - » How to prevent fragmentation?

LFS Log Retrieval

- Keep same basic file structure as UNIX (inode, indirect blocks, data)
- Retrieval is just a question of finding a file’s inode
- UNIX inodes kept in one or a few big arrays, LFS inodes must float to avoid update-in- place
- Solution: an *inode map* that tells where each inode is (Also keeps other stuff: version number, last access time, free/allocated)
- inode map gets written to log like everything else
- Map of inode map gets written in special checkpoint location on disk; used in crash recovery

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
  - Try to collect long-lived info into segments that never need to be cleaned
  - Note there is not free list or bit map (as in FFS), only a list of clean segments

LFS Segment Cleaning

- Which segments 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, since those are trivial to clean…
  - If utilization of segments being cleaned is U:
    - » write cost = (total bytes read & written)/(new data written) = 2/(1-U) (unless U is 0)
    - » write cost increases as U increases: U = .9 => cost = 20!
    - » Need a cost of less than 4 to 10; => U of less than .75 to .45
- 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 #}

Analysis and Evolution of Journaling File Systems

- Write-ahead logging: commit data by writing it to log, synchronously and sequentially
- Unlike LFS, then later moved data to its normal (FFS-like) location – this write is called *checkpointing* and like segment cleaning, it 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, which is only read for crash recovery and checkpointing
- Much better than FFS (fsck) for crash recovery (covered below) because it is much faster
- Ext3/ReiserFS/Ext4 filesystems are the main ones in Linux

Three modes for a JFS

- *Writeback mode*:
  - Journal only metadata
  - Write back data and metadata independently
  - Metadata may thus have dangling references after a crash (if metadata written before the data with a crash in between)
- *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 the tail/head of the log
- Scan log to detect whole committed transactions (they have 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
- How to find the head of the log?

Some Fine Points

- Can group transactions together: fewer syncs and fewer writes, since hot metadata may changes several times within one transaction
- Need to write a commit record, so that you can tell that all of the compound transaction made it to disk
- ext3 logs whole metadata blocks (physical logging); JFS and NTFS log logical records instead, which means less journal traffic

Some Fine Points

- Head of line blocking:
  - Compound transactions can link together concurrent streams (e.g., from different apps) and hinder asynchronous apps performance (Figure 6)
  - This is like having no left turn lane and waiting on the car in front of you to turn left, when you just want to go straight
- Distinguish
  - Between ordering of writes and durability/persistence – careful ordering means that after a crash the file system can be recovered to a consistent past state.
  - But that state could be far in the past in the case of JFS
  - 30 seconds behind is more typical for ext3 – if you really want something to be durable you must flush the log synchronously

Semantic Block-level Analysis (SBA)

- Nice idea: interpose special disk driver between the file system and the real disk driver
- Pros: simple, captures ALL disk traffic, can use with a black-box filesystem (no source code needed and can even use via VMWare for another OS), can be more insightful than just a performance benchmark
- Cons: must have some understanding of the disk layout, which differs for each filesystem, requires a great deal of inference; really only useful for writes
- To use well, drive filesystem with smart applications that test certain features of the filesystem (to make the inference easier)

Semantic Trace Playback (STP)

- Uses two kinds of interposition:
  - 1) SBA driver that produces a trace, and
  - 2) user-level library that fits between the app and the real filesystem
- 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
- Claim:
  - Faster to modify the trace than to modify the filesystem 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)

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Extra Slides on LFS

LFS i-node and Block Cleaning

- To clean an i-node:
  - Just check to see if it is the current version (from i-node map)
  - If not, skip it; if so, write to head of log and update i-node map
- To clean a file block, must figure out it if is still live
  - First check the UID, which only tells you if this file is current (UID only changes when is deleted or has length zero)
  - Note that UID does not change every time the file is modified (since you would have to update the UIDs of all of its blocks)
  - Next, walk through the i-node and any indirect blocks to get to the data block pointer for this block number
    - » If it points to this block, then move the block to the head of the log

Simulation of LFS Cleaning

- Initial model: Uniform random distribution of references; greedy algorithm for segment- to-clean selection
- Why does the simulation do better than the formula?
  - Because of variance in segment utilizations
- Added locality (i.e., 90% of references go to 10% of data) and things got worse!

LFS Cleaning Solution #1

- First solution: Write out cleaned data ordered by age to obtain hot and cold segments
  - What prog. language feature does this remind you of? (Generational GC)
  - Only helped a little
- Problem:
  - Even cold segments eventually have to reach the cleaning point, but they drift down slowly. tying up lots of free space
  - Do you believe that’s true?

LFS Cleaning Solution #2

- Second Solution:
  - It's worth paying more to clean cold segments because you get to keep the free space longer
- Better way to think about this:
  - Don't clean segments that have a high d-free/dt (first derivative of utilization)
  - If you ignore them, they clean themselves!
  - LFS uses age as an approximation of d-free/dt, because the latter is hard to track directly
- New selection function:
  - MAX(T*(1-U)/(1+U))
  - Resulted in the desired bi-modal utilization function
  - LFS stays below write cost of 4 up to a disk utilization of 80%

LFS Recovery Techniques

- Three techniques:
  - Checkpoints
  - Crash Recovery
  - Directory Operation Log

LFS Checkpoints

- LFS Checkpoints:
  - Just an optimization to roll forward
  - Reduces recovery time
- Checkpoint contains: pointers to i-node map and segment usage table, current segment, timestamp, checksum (?)
- Before writing a checkpoint make sure to flush i-node map and segment usage table
- Uses "version vector" approach:
  - Write checkpoints to alternating locations with timestamps and checksums
  - On recovery, use the latest (valid) one

LFS Crash Recovery

- Unix must read entire disk to reconstruct meta data
- LFS reads checkpoint and rolls forward through log from checkpoint state
- Result: recovery time measured in seconds instead of minutes to hours
- Directory operation log == log *intent* to achieve atomicity, then redo during recovery, (undo for new files with no data, since you can't redo it)

LFS Directory Operation Log

- Example of “intent + action”:
  - Write the intent as a “directory operation log”
  - Then write the actual operations (create, link, unlink, rename)
- This makes them atomic
- On recovery, if you see the operation log entry, then you can REDO the operation to complete it (For new file create with no data, you UNDO it instead)
- => “logical” REDO logging

LFS Summary

- Key features of paper:
  - 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 to obtain 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
- Some flaws:
  - 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

- An 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
- A Lesson:
  - 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