Filesystems
Lecture 13

Credit: some slides by John Kubiatowicz and Anthony D. Joseph

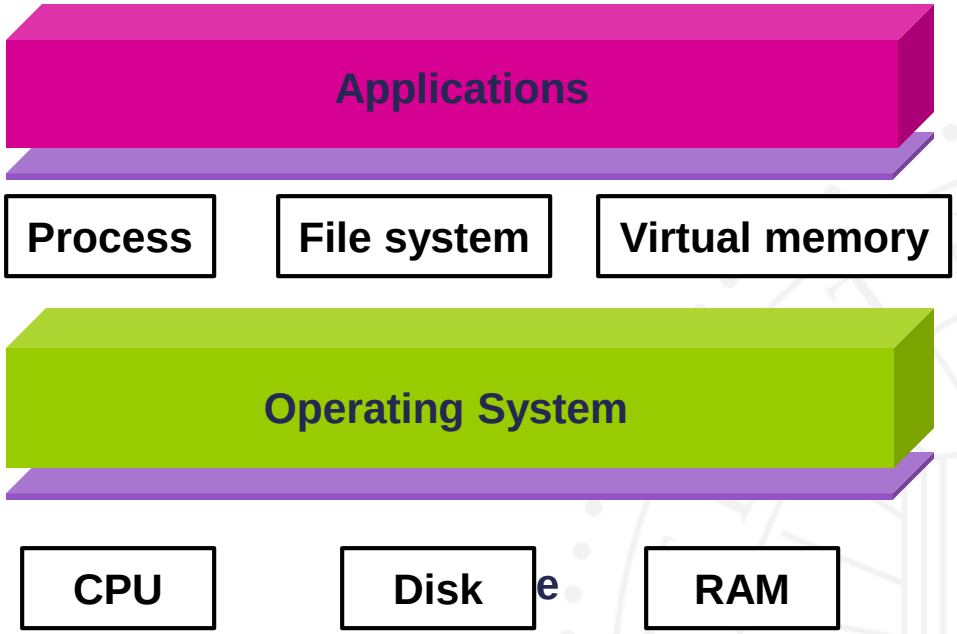
UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

Today and some of next class

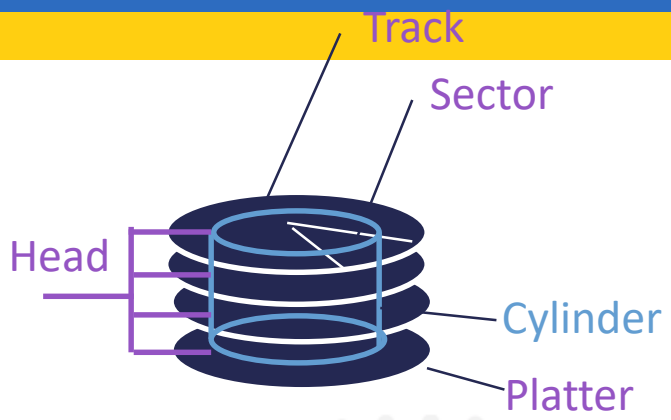
- Overview of file systems
- Papers on basic file systems
  - A Fast File System for UNIX
    Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry. Appears in *ACM Transactions on Computer Systems* (TOCS), Vol. 2, No. 3, August 1984, pp 181-197
  - Log Structured File Systems (LFS), Ousterhout and Rosenblum

- System design paper and system analysis paper

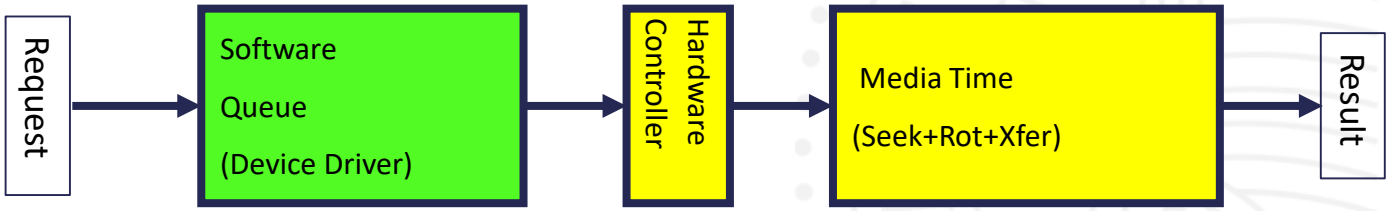
OS Abstractions



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 = Queuing Time + Controller time + Seek Time + Rotation Time + Xfer Time



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

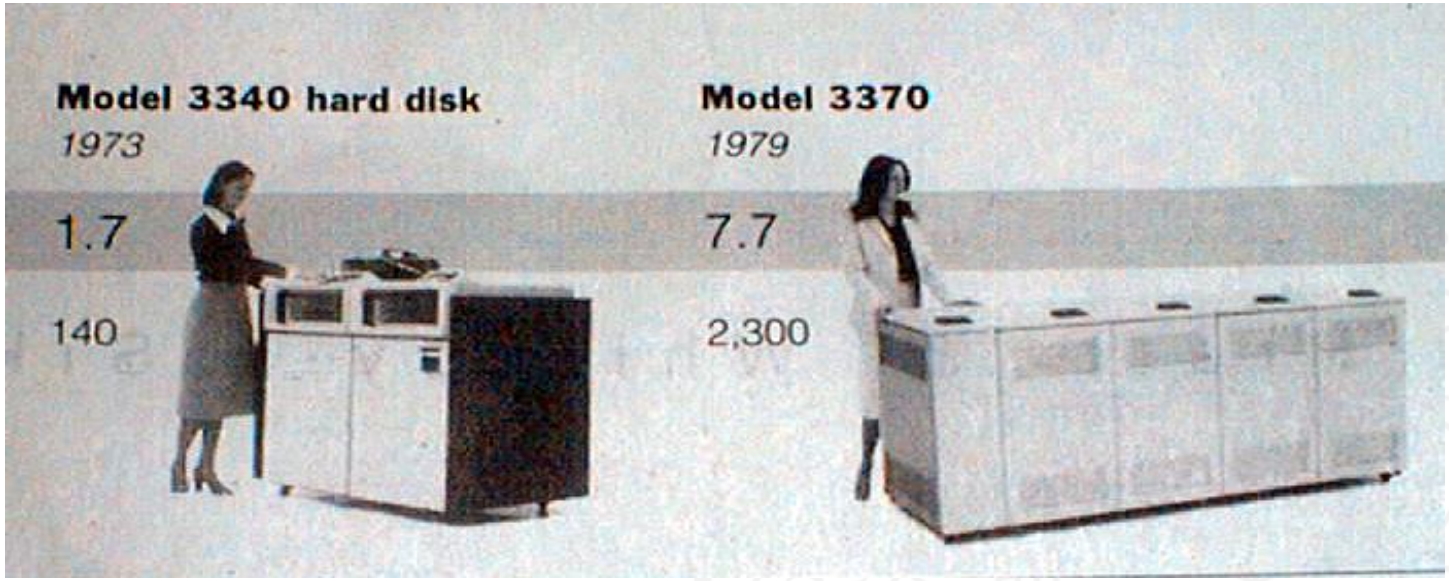
Historical Perspective

- 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

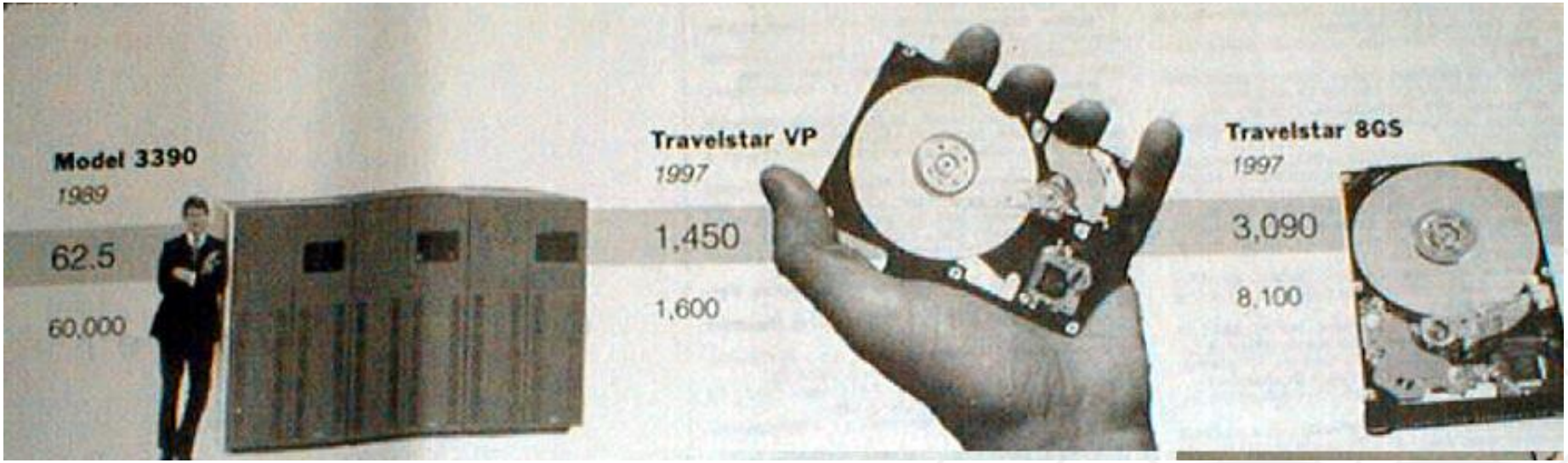


1973:
1. 7 Mbit/sq. in
140 MBytes

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

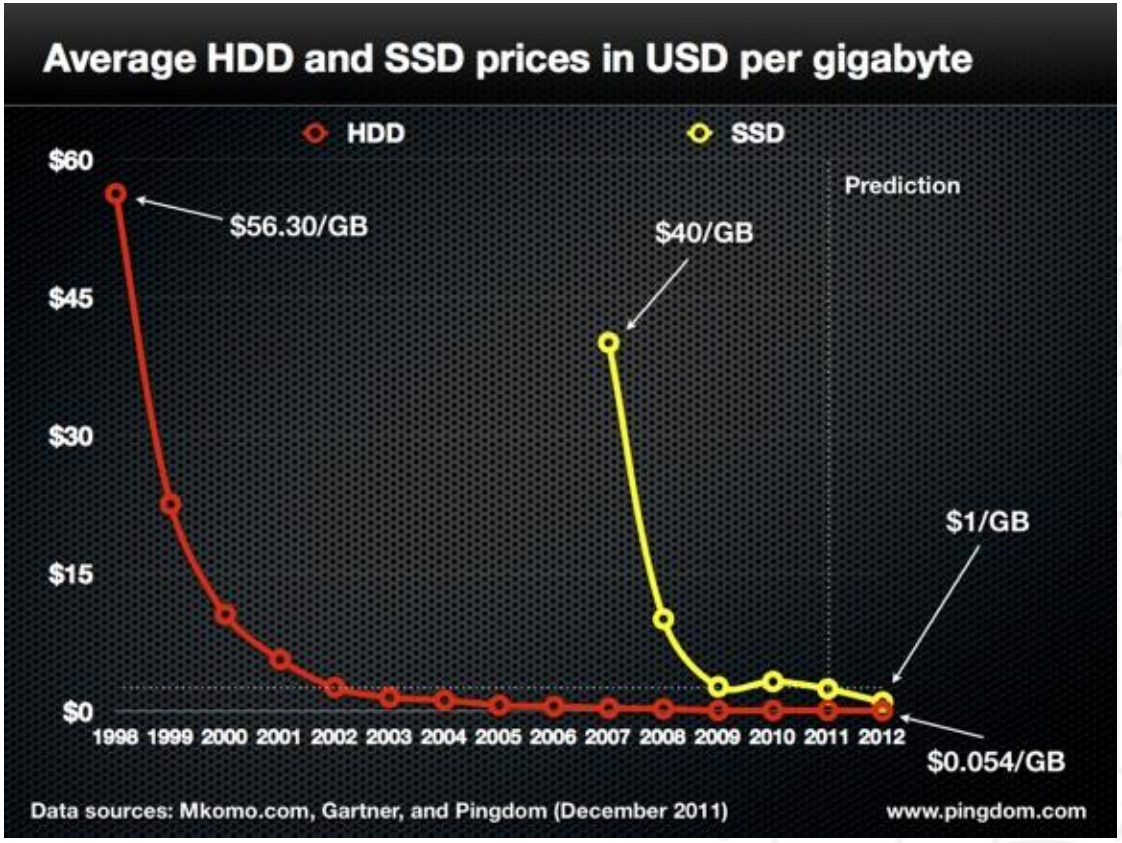
source: New York Times, 2/23/98, page C3,
"Makers of disk drives crowd even mroe data into even smaller spaces"

Recent: Seagate Enterprise (2015)

- 10TB! 800 Gbits/inch^2
- 7 (3.5”) platters, 2 heads each
- 7200 RPM, 8ms seek latency
- 249/225 MB/sec read/write transfer rates
- 2.5million hours MTBF
- 256MB cache
- $650

Contrarian View

- FFS doesn't matter in 2012!



- What about Journaling? Is it still relevant?

60 TB SSD ($20,000)

Storage Performance & Price

	Bandwidth (sequential R/W)	Cost/GB	Size
HHD	50-100 MB/s	$0.05-0.1/GB	2-4 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

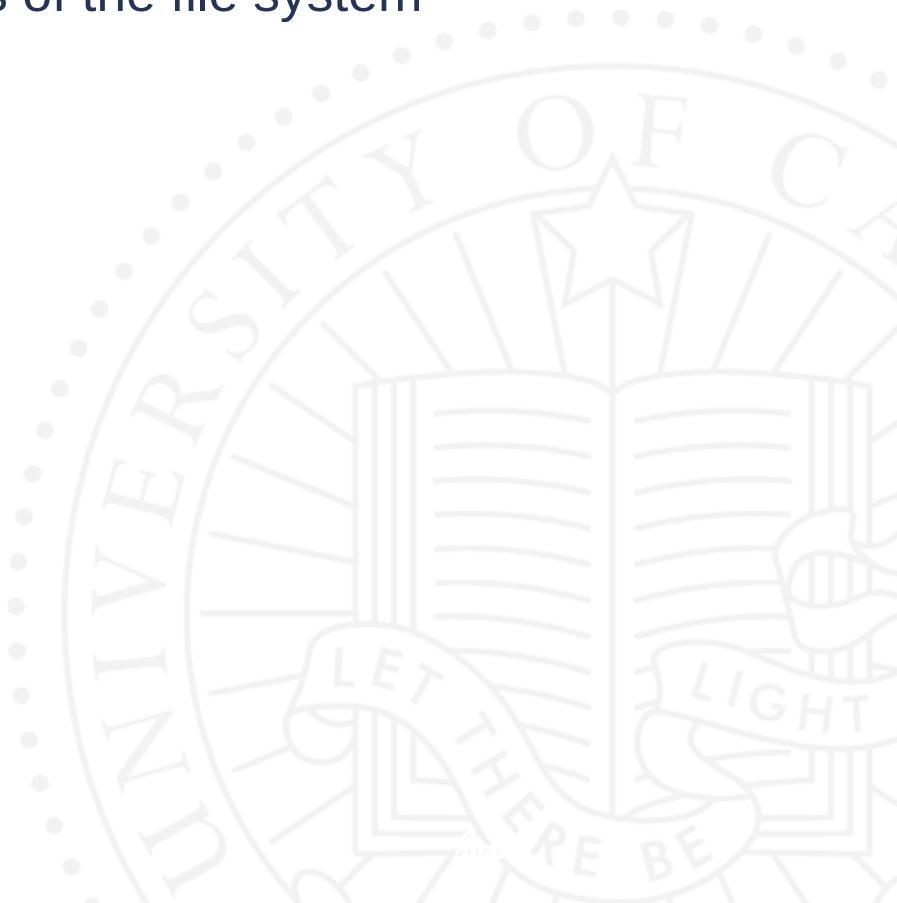
Price: HDD x30 less than SSD, SSD x4 less than DRAM

File system abstractions

- How do users/user programs interact with the file system?
  - Files
  - Directories
  - Links
  - Protection/sharing model
- Accessed and manipulated by a virtual file system set of system calls
- File system implementation:
  - How to map these abstractions to the storage devices
  - Alternatively, how to implement those system calls

File system basics

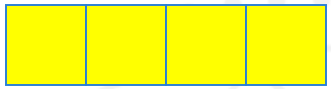
- Virtual file system abstracts away concrete file system implementation
  - Isolates applications from details of the file system
- Linux vfs interface includes:
  - creat(name)
  - open(name, how)
  - read(fd, buf, len)
  - write(fd, buf, len)
  - sync(fd)
  - seek(fd, pos)
  - close(fd)
  - unlink(name)



Disk Layout Strategies

- Files span multiple disk blocks
- How do you find all of the blocks for a file?

1. Contiguous allocation
   - Like memory
   - Fast, simplifies directory access
   - Inflexible, causes fragmentation, needs compaction

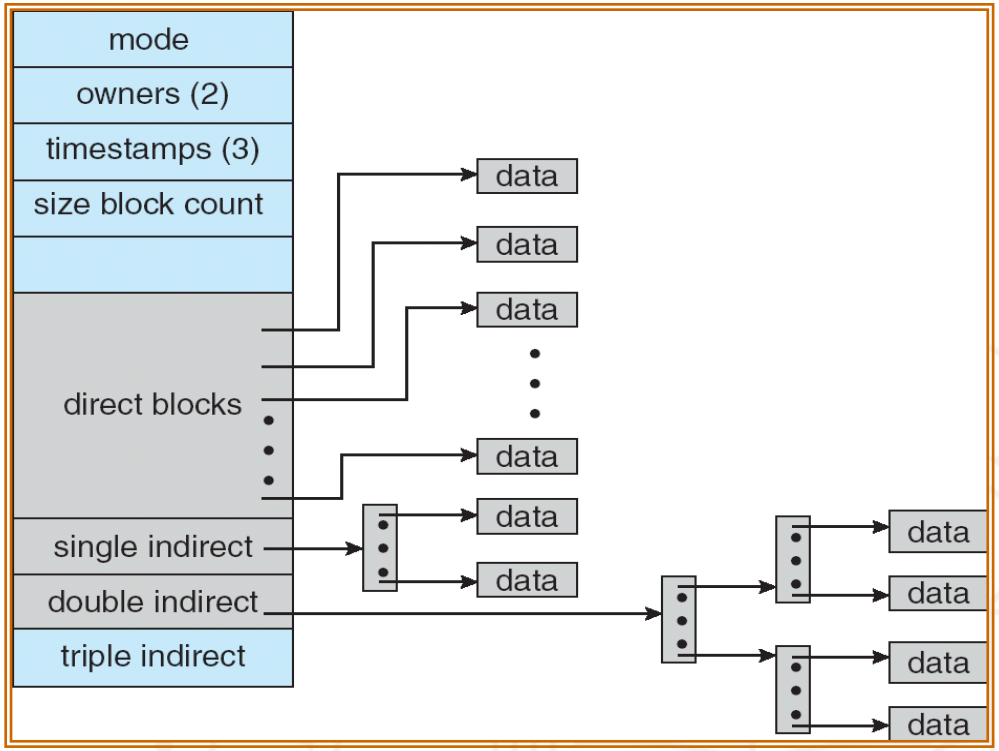


2. Linked structure
   - Each block points to the next, directory points to the first
   - Bad for random access patterns

3. Indexed structure (indirection, hierarchy)
   - An “index block” contains pointers to many other blocks
   - Handles random better, still good for sequential
   - May need multiple index blocks (linked together)

Zooming in on i-node

- 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

Unix Inodes and Path Search

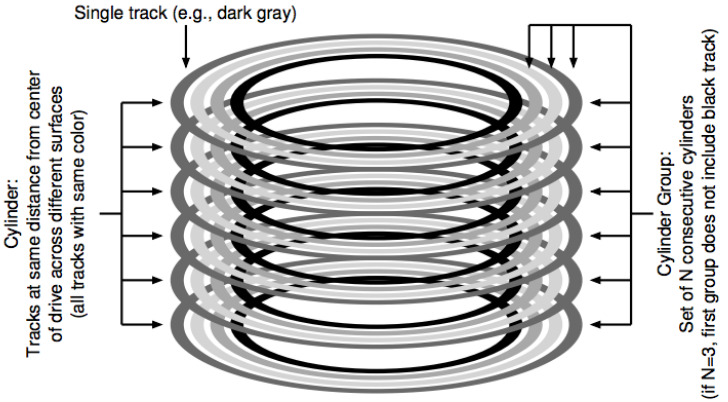
- Unix Inodes are not directories
- Inodes describe where on disk the blocks for a file are placed
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- Directory entries map file names to inodes
  - To open "/one", use Master Block to find inode for "/" on disk
  - Open "/", look for entry for "one"
  - This entry gives the disk block number for the inode for "one"
  - Read the inode for "one" into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file
- This is why we have *open* in addition to *read* and *write*

FFS – what's wrong with original unix FS?

- 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 -- Locality is important



- 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

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
- Watershed moment for OS designers– File system matters

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

File operations still expensive

- How many operations (seeks) to create a new file?
  - New file, needs a new inode
  - But at least a block of data too
  - Check and update the inode and data bitmap (eventually have to be written to disk)
  - Not done yet – need to add it to the directory (update the directory inode and the directory data block – may need to split if its full)…
  - Whew!! How does all of this even work?

- So what is the advantage?
  - Not removing any operations
  - Seeks are just shorter…

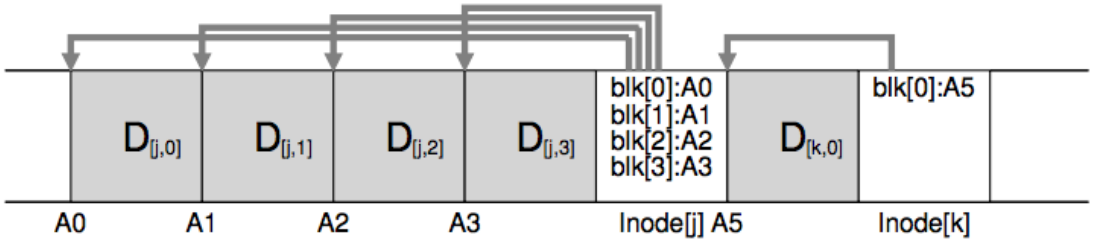
Log-Structured/Journaling File System

- 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)
    6. (not to mention bitmap updates)

LFS Basic Idea

- Log all data and metadata with efficient, large, sequential writes
  - Do not update blocks in place – just write new versions in the log
- Treat the log as the truth, but keep an index on its contents
- Not necessarily good for reads, but trends help
  - 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!

Basic idea

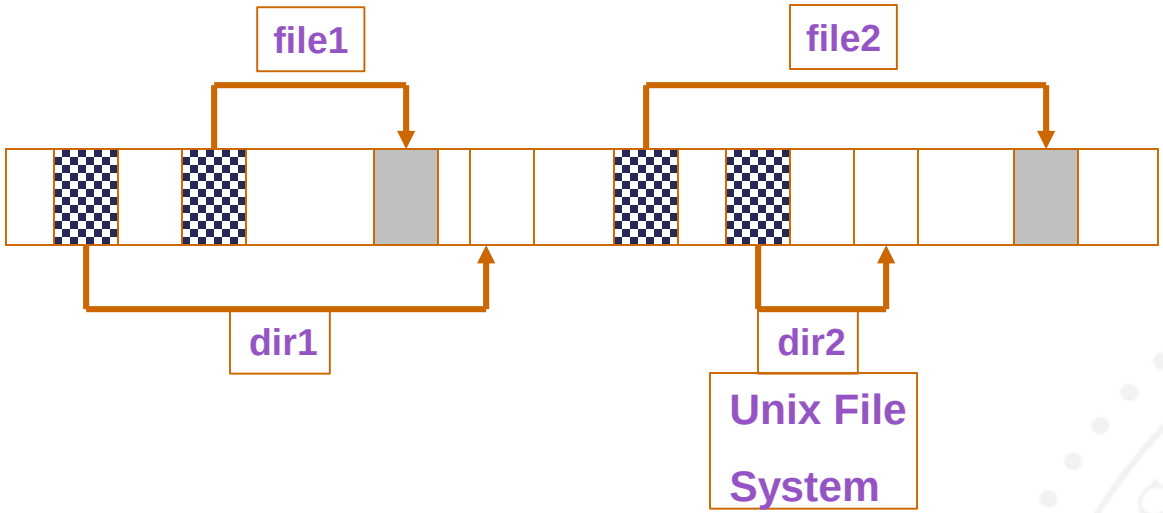


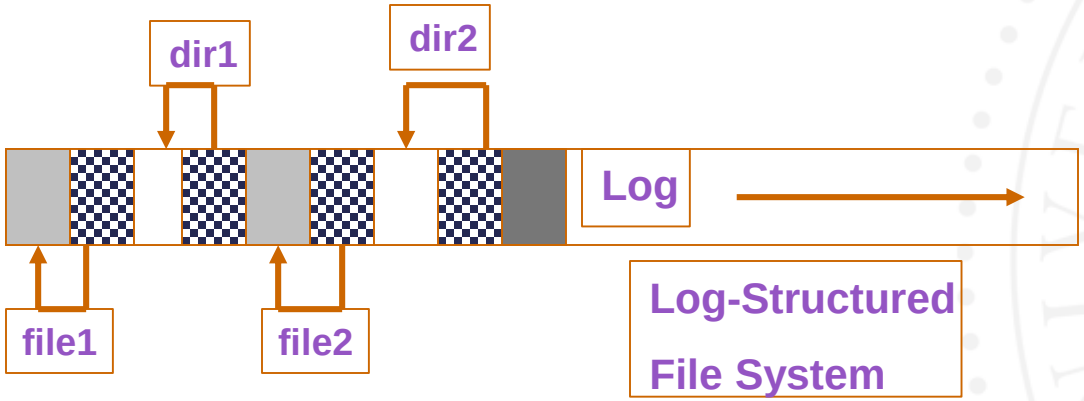
- We buffer all updates, and write them together in one big sequential write
  - Good for the disk
  - Example above, writes to two different files were written together (along with the new version of i-node) in one write
  - How much should we buffer?
    - What happens if too much? If too little?
- But how do we find a file??
  - All problems in CS solved with another level of indirection ☺

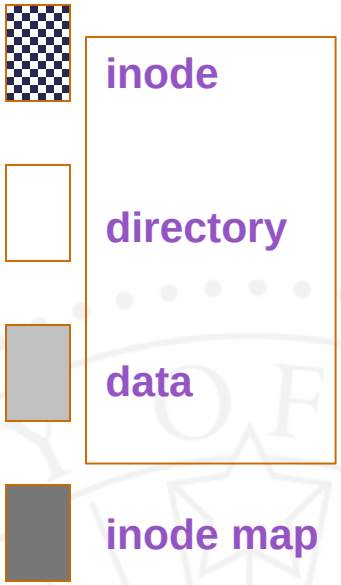
Devil is in the details

- Two potential problems:
  - Log retrieval on cache misses – how do we find the data?
  - Wrap-around: what happens when end of disk is reached?
    - No longer any big, empty runs available
    - How to prevent fragmentation?

LFS vs. UFS

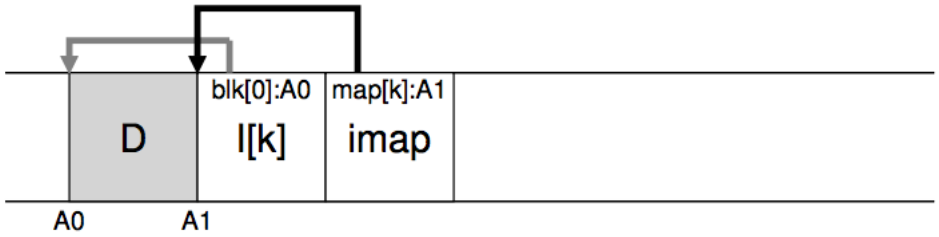






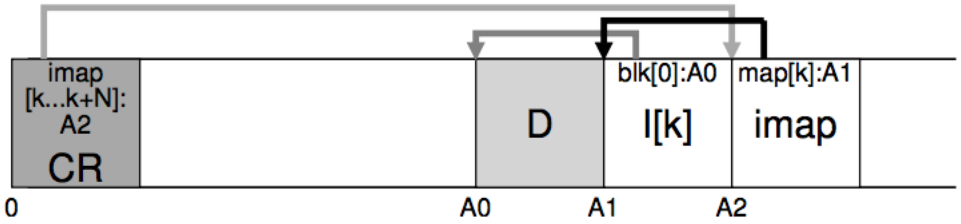
Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

i-node map



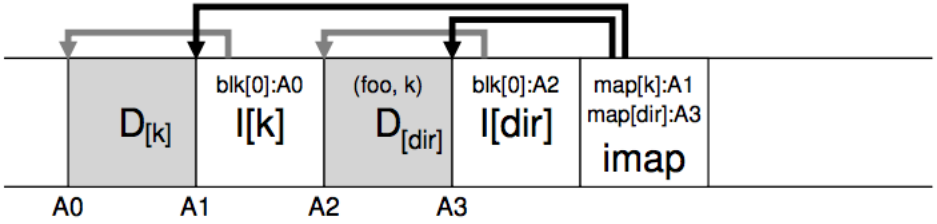
- A map keeping track of the location of i-nodes
- Anytime an i-node is written to disk, the imap is updated
  - But is that any better? In a second
- Most of the time the imap is in memory, so access is fast
- Updated imap is saved as part of the log!
  - but how do we find it!

Final piece to the solution



- Checkpoint region is written to point to the location of the imap
  - Also serves as an indicator of a stable point in the file system for crash recovery
- So, to read a file from LFS:
  - Read the CR, use it to read and cache the imap
  - After that, it is identical to FFS
  - Are reads fast?

What about directories?



- When a file is updated, its inode changes (new copy)
  - We need to update the directory inode (also creating a copy)
  - We need to update its parent directory
- Ugh….what to do?
  - Inode map helps with that too – just keep track of inode number and resolve it through inode map

LFS Disk Wrap-Around/Garbage collection

- 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 #}
  - Through i-map you check if this block is still being used for the (i-number, block #)

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?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?