

Outline for Today's Lecture

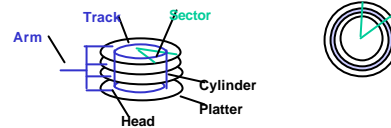
Administrative:

- Assignment 5 – start NOW. Read the newsgroup. Justin will be out of town this weekend.
- Handout – Nachos guide formatted!

Objective:

- Disk dilemmas

Rotational Media

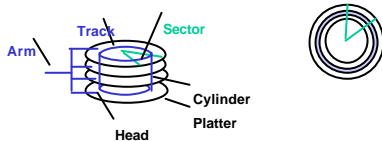


Access time = seek time + rotational delay + transfer time

seek time = 5-15 milliseconds to move the disk arm and settle on a cylinder
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.

Rotational Media



Access time = seek time + rotational delay + transfer time + spinup time

seek time = 5-15 milliseconds to move the disk arm and settle on a cylinder
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s
Spinup/spindown time = ~1 second

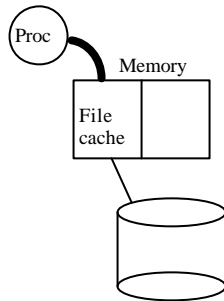
Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.

What to do about Disks?

- Avoid them altogether! Caching
- Disk scheduling
 - Idea is to reorder outstanding requests to minimize seeks.
- Layout on disk
 - Placement to minimize disk overhead
- Build a better disk (or substitute)
 - Example: RAID

File Buffer Cache

- Avoid the disk for as many file operations as possible.
- Cache acts as a filter for the requests seen by the disk – reads served best.
- Delayed writeback will avoid going to disk at all for temp files.



Handling Updates in the File Cache

1. Blocks may be modified in memory once they have been brought into the cache.

Modified blocks are *dirty* and must (eventually) be written back.

2. Once a block is modified in memory, the write back to disk may not be immediate (*synchronous*).

Delayed writes absorb many small updates with one disk write.

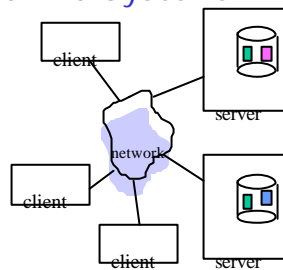
How long should the system hold dirty data in memory?

Asynchronous writes allow overlapping of computation and disk update activity (*write-behind*).

Do the **write** call for block $n+1$ while transfer of block n is in progress.

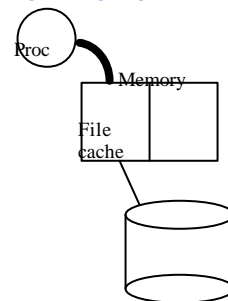
Distributed File Systems

- ✓ Naming
 - Location transparency/independence
- Caching
 - Consistency
- Replication
 - Availability and updates



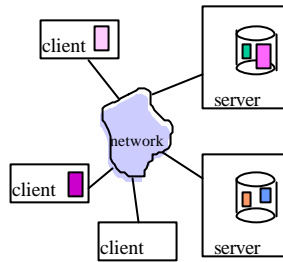
Caching was "The Answer"

- Avoid the disk for as many file operations as possible.
- Cache acts as a filter for the requests seen by the disk – reads served best.
- Delayed writeback will avoid going to disk at all for temp files.



Caching in Distributed F.S.

- Location of cache on client - disk or memory
- Update policy
 - write through
 - delayed writeback
 - write-on-close
- Consistency
 - Client does validity check, contacting server
 - Server call-backs



File Cache Consistency

Caching is a key technique in distributed systems.

The *cache consistency problem*: cached data may become *stale* if cached data is updated elsewhere in the network.

Solutions:

Timestamp invalidation (NFS).

Timestamp each cache entry, and periodically query the server: "has this file changed since time t ?"; invalidate cache if stale.

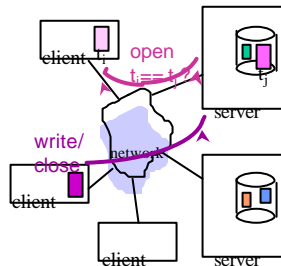
Callback invalidation (AFS).

Request notification (callback) from the server if the file changes; invalidate cache on callback.

Leases (NQ-NFS) [Gray&Cheriton89]

Sun NFS Cache Consistency

- Server is **stateless**
- Requests are self-contained.
- **Blocks** are transferred and cached in memory.
- Timestamp of last known mod kept with cached file, compared with "true" timestamp at server on Open. (Good for an interval)
- Updates delayed but flushed before Close ends.



45

File Handles

Question: how does the client tell the server which file or directory the operation applies to?

- Similarly, how does the server return the result of a *lookup*?
 - More generally, how to pass a pointer or an object reference as an argument/result of an RPC call?

In NFS, the reference is a **file handle** or **handle**, a 32-byte token/ticket whose value is determined by the server.

- Includes all information needed to identify the file/object on the server, and get a pointer to it quickly.

volume ID	inode #	generation #

NFS: From Concept to Implementation

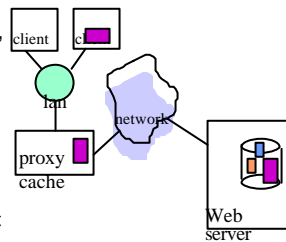
- How do we make it fast?
 - Answer: caching, read-ahead, and write-behind.
- How do we make it reliable? What if a message is dropped? What if the server crashes?
 - Answer: client retransmits request until it receives a response.
- How do we preserve file system semantics in the presence of failures and/or sharing by multiple clients?
 - Answer: well, we don't, at least not completely.
- What about security and access control?

NQ-NFS Leases

- In NQ-NFS, a client obtains a *lease* on the file that permits the client's desired read/write activity.
- "A lease is a ticket permitting an activity; the lease is valid until some expiration time."
- A *read-caching lease* allows the client to cache clean data.
 - Guarantee**: no other client is modifying the file.
 - A *write-caching lease* allows the client to buffer modified data for the file.
 - Guarantee**: no other client has the file cached.
- Leases may be revoked by the server if another client requests a conflicting operation (server sends *eviction notice*).
 Since leases expire, losing "state" of leases at server is OK.

Cache Consistency for the Web

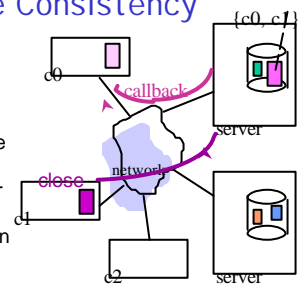
- Time-to-Live (TTL) fields - HTTP "expires" header
- Client polling - HTTP "if-modified-since" request headers
 - polling frequency? possibly *adaptive* (e.g. based on age of object and assumed stability)



50

AFS Cache Consistency

- Server keeps state of all clients holding copies (copy set)
- **Callbacks** when cached data are about to become stale
- Large units (whole files or 64K portions)
- Updates propagated upon close
- Cache on local disk & memory
- If client crashes, revalidation on recovery (lost callback possibility)



51

Coda - Using Caching to Handle Disconnected Access

- Single location-transparent UNIX FS.
 - Scalability - coarse granularity (whole-file caching, volume management)
 - First class (server) replication and client caching (second class replication)
 - Optimistic replication & consistency maintenance.
- Designed for **disconnected operation** for mobile computing clients

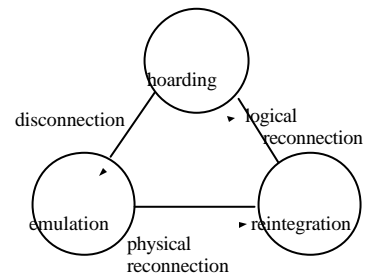
Explicit First-class Replication

- File name maps to set of replicas, one of which will be used to satisfy request
 - Goal: availability
- Update strategy
 - Atomic updates - all or none
 - Primary copy approach
 - Voting schemes
 - Optimistic, then detection of conflicts

Optimistic vs. Pessimistic

- High availability
Conflicting updates are the potential problem - requiring detection and resolution.
- Avoids conflicts by holding of shared or exclusive locks.
- How to arrange when disconnection is involuntary?
- Leases [Gray, SOSP89] puts a time-bound on locks but what about expiration?

Client-cache State Transitions



Prefetching

- To avoid the access *latency* of moving the data in for that first cache miss.
- **Prediction!** “Guessing” what data will be needed in the future.
 - It’s not for free:
Consequences of guessing wrong
Overhead

Hoarding - Prefetching for Disconnected Information Access

- Caching for availability (not just latency)
- Cache misses, when operating disconnected, have no redeeming value.
(Unlike in connected mode, they can’t be used as the triggering mechanism for filling the cache.)
- How to preload the cache for subsequent disconnection? Planned or unplanned.
- What does it mean for replacement?

Hoard Database

- Per-workstation, per-user set of pathnames with priority
- User can explicitly tailor HDB using scripts called *hoard profiles*
- Delimited observations of reference behavior (snapshot spying with bookends)

Coda Hoarding State

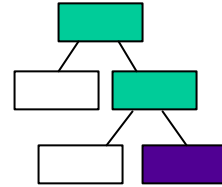
- Balancing act - caching for 2 purposes at once:
 - performance of current accesses,
 - availability of future disconnected access.
- Prioritized algorithm -
Priority of object for retention in cache is **f(hoard priority, recent usage)**.
- Hoard walking (periodically or on request) maintains equilibrium - no uncached object has higher priority than any of cached objects

The Hoard Walk

- Hoard walk - phase 1 - reevaluate name bindings (e.g., any new children created by other clients?)
- Hoard walk - phase 2 - recalculate priorities in cache and in HDB, evict and fetch to restore equilibrium

Hierarchical Cache Mgt

- Ancestors of a cached object must be cached in order to resolve pathname.
- Directories with cached children are assigned infinite priority



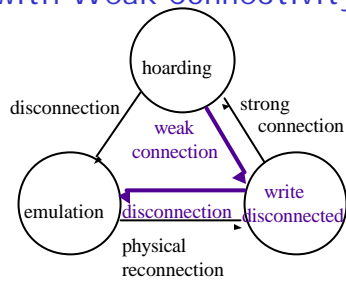
Callbacks During Hoarding

- Traditional callbacks - invalidate object and refetch on demand
- With threat of disconnection
 - Purge files and refetch on demand or hoard walk
 - Directories - mark as stale and fix on reference or hoard walk, available until then just in case.

Emulation State

- Pseudo-server, subject to validation upon reconnection
- Cache management by priority
 - modified objects assigned infinite priority
 - freeing up disk space - compression, replacement to floppy, backout updates
- Replay log also occupies non-volatile storage (RVM - recoverable virtual memory)

Client-cache State Transitions with Weak Connectivity



Cache Misses with Weak Connectivity

- At least now it's possible to service misses but \$\$\$ and it's a foreground activity (noticeable impact). Maybe **not**
- User patience threshold - estimated service time compared with what is acceptable
- Defer misses by adding to HDB and letting hoard walk deal with it
- User interaction during hoard walk.

What to do about Disks?

- ✓ Avoid them altogether! Caching
- Disk scheduling
 - Idea is to reorder outstanding requests to minimize seeks.
- Layout on disk
 - Placement to minimize disk overhead
- Build a better disk (or substitute)
 - Example: RAID

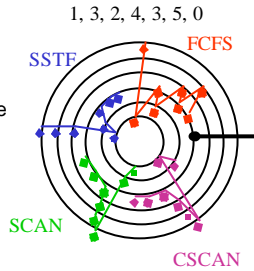
Disk Scheduling

- Assuming there are sufficient outstanding requests in request queue
- Focus is on seek time - minimizing physical movement of head.
- Simple model of seek performance

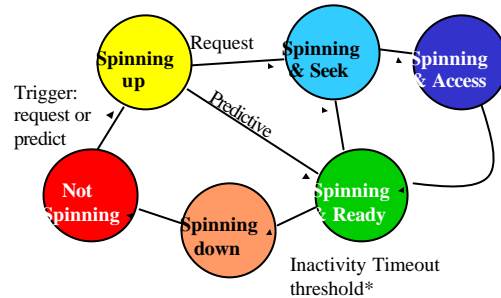
$$\text{Seek Time} = \text{startup time (e.g. 3.0 ms)} + N \text{ (number of cylinders) } * \text{per-cylinder move (e.g. .04 ms/cyl)}$$

Policies

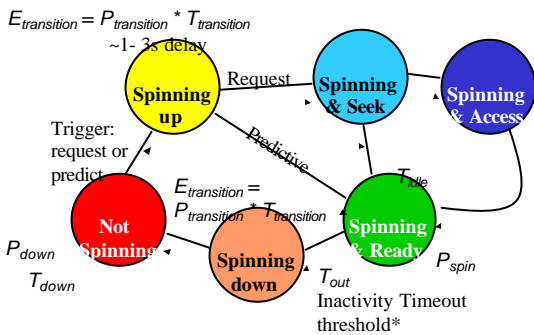
- Generally use FCFS as baseline for comparison
- Shortest Seek First (SSTF) - closest
 - danger of starvation
- Elevator (SCAN) - sweep in one direction, turn around when no requests beyond
 - handle case of constant arrivals at same position
- C-SCAN - sweep in only one direction, return to 0
 - less variation in response



Spin-down Disk Model



Spin-down Disk Model



Reducing Energy Consumption

$$Energy = \sum_{i \in \text{power states}} Power_i \times Time_i$$

To reduce energy used for task:

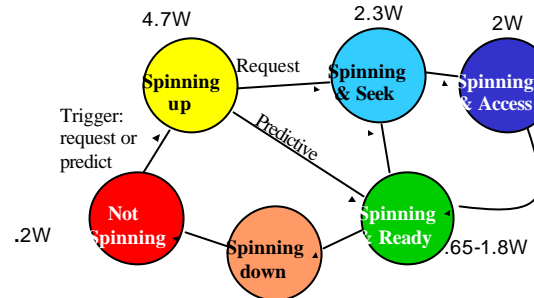
- Reduce power cost of power state / through better technology.
- Reduce time spent in the higher cost power states.
- Amortize transition states (spinning up or down) if significant.

$$P_{down} * T_{down} + 2 * E_{transition} + P_{spin} * T_{out} < P_{spin} * T_{idle}$$

Power Specs

- | | |
|--|--|
| <p>IBM Microdrive (1inch)</p> <ul style="list-style-type: none"> • writing 300mA (3.3V) 1W • standby 65mA (3.3V) .2W | <p>IBM TravelStar (2.5inch)</p> <ul style="list-style-type: none"> • read/write 2W • spinning 1.8W • low power idle .65W • standby .25W • sleep .1W • startup 4.7 W • seek 2.3W |
|--|--|

Spin-down Disk Model



Spin-Down Policies

- Fixed Thresholds
 - $T_{out} = \text{spin-down cost s.t. } 2 * E_{transition} = P_{spin} * T_{out}$
- Adaptive Thresholds: $T_{out} = f(\text{recent accesses})$
 - Exploit burstiness in T_{idle}
- Minimizing Bumps (user annoyance/latency)
 - Predictive spin-ups
- Changing access patterns (making burstiness)
 - Caching
 - Prefetching

What to do about Disks?

- ✓ Avoid them altogether! Caching
- ✓ Disk scheduling
 - Idea is to reorder outstanding requests to minimize seeks.
- Layout on disk
 - Placement to minimize disk overhead
- Build a better disk (or substitute)
 - Example: RAID

Layout on Disk

- Can address both seek and rotational latency
- Cluster related things together (e.g. an inode and its data, inodes in same directory (ls command), data blocks of multi-block file, files in same directory)
- Sub-block allocation to reduce fragmentation for small files
- Log-Structured File Systems

The Problem of Disk Layout

- The level of indirection in the file block maps allows flexibility in file layout.
 - "File system design is 99% block allocation." [McVoy]
- Competing goals for block allocation:
 - *allocation cost*
 - *bandwidth* for high-volume transfers
 - *stamina*
 - *efficient directory operations*
- **Goal**: reduce disk arm movement and seek overhead.
 - metric of merit: *bandwidth utilization*

FFS and LFS

Two different approaches to block allocation:

- Cylinder groups in the Fast File System (FFS) [McKusick81]
 - clustering enhancements [McVoy91], and improved cluster allocation [McKusick: Smith/Seltzer96]
 - FFS can also be extended with metadata logging [e.g., Episode]
- Log-Structured File System (LFS)
 - proposed in [Douglis/Ousterhout90]
 - implemented/studied in [Rosenblum91]
 - BSD port, sort of maybe: [Seltzer93]
 - extended with self-tuning methods [Neefe/Anderson97]
- Other approach: *extent-based* file systems

FFS Cylinder Groups

- FFS defines *cylinder groups* as the unit of disk locality, and it factors locality into allocation choices.
 - typical: thousands of cylinders, dozens of groups
 - **Strategy** place "related" data blocks in the same cylinder group whenever possible.
 - seek latency is proportional to seek distance
 - Smear large files across groups:
 - Place a run of contiguous blocks in each group.
 - Reserve inode blocks in each cylinder group.
 - This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files).



FFS Allocation Policies

1. Allocate file inodes close to their containing directories.
 - For *mkdir*, select a cylinder group with a more-than-average number of free inodes.
 - For *creat*, place inode in the same group as the parent.
2. Concentrate related file data blocks in cylinder groups.
 - Most files are read and written sequentially.
 - Place initial blocks of a file in the same group as its inode.
 - How should we handle directory blocks?
 - Place adjacent logical blocks in the same cylinder group.
 - Logical block $n+1$ goes in the same group as block n .
 - Switch to a different group for each indirect block.

Allocating a Block

1. Try to allocate the rotationally optimal physical block after the previous logical block in the file.
 - Skip *rotdelay* physical blocks between each logical block. (rotdelay is 0 on trackcaching disk controllers.)
2. If not available, find another block a nearby rotational position in the same cylinder group
 - We'll need a short seek, but we won't wait for the rotation.
 - If not available, pick any other block in the cylinder group.
3. If the cylinder group is full, or we're crossing to a new indirect block, go find a new cylinder group.
 - Pick a block at the beginning of a run of free blocks.

Representing Small Files

- Internal fragmentation in the file system blocks can waste significant space for small files.
 - E.g., 1KB files waste 87% of disk space (and bandwidth) in a naive file system with an 8KB block size.
 - Most files are small: one study [Irlam93] shows a median of 22KB.
- FFS solution: optimize small files for space efficiency.
 - Subdivide blocks into 2/4/8 *fragments* (or just *frags*).
 - Free block maps contain one bit for each fragment.
 - To determine if a block is free, examine bits for all its fragments.
 - The last block of a small file is stored on fragment(s).
 - If multiple fragments they must be contiguous.

Small Files with Bigger Blocks

- Internal fragmentation in the file system blocks can waste significant space for small files.
 - E.g., 1KB files waste 87% of disk space (and bandwidth) in a naive file system with an 8KB block size.
 - Most files are small: one study [Irlam93] shows a median of 22KB.
- FFS solution: optimize small files for space efficiency.
 - Subdivide blocks into 2/4/8 *fragments* (or just *frags*).
 - Free block maps contain one bit for each fragment.
 - To determine if a block is free, examine bits for all its fragments.
 - The last block of a small file is stored on fragment(s).
 - If multiple fragments they must be contiguous.

Clustering in FFS

- *Clustering* improves bandwidth utilization for large files read and written sequentially.
 - Allocate clumps/clusters/runs of blocks contiguously; read/write the entire clump in one operation with at most one seek.
 - Typical cluster sizes: 32KB to 128KB.
- FFS can allocate contiguous runs of blocks “most of the time” on disks with sufficient free space.
 - This (usually) occurs as a side effect of setting *rotdelay*=0.
 - Newer versions may relocate to clusters of contiguous storage if the initial allocation did not succeed in placing them well.
 - Must modify buffer cache to group buffers together and read/write in contiguous clusters.

Effect of Clustering

Access time = seek time + rotational delay + transfer time

average seek time = 2 ms for an intra-cylinder group seek, let's say
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

8 KB blocks deliver about 15% of disk bandwidth.
64KB blocks/clusters deliver about 50% of disk bandwidth.
128KB blocks/clusters deliver about 70% of disk bandwidth.

Actual performance will likely be better with good disk layout, since most seek/rotate delays to read the next block/cluster will be “better than average”.

Log-Structured File Systems

- Assumption: Cache is effectively filtering out reads so we should optimize for writes
- Basic Idea: manage disk as an append-only *log* (subsequent writes involve minimal head movement)
- Data and meta-data (mixed) accumulated in large segments and written contiguously
- Reads work as in UNIX - once inode is found, data blocks located via index.
- Cleaning an issue - to produce contiguous free space, correcting fragmentation developing over time.

Log-Structured File System (LFS)

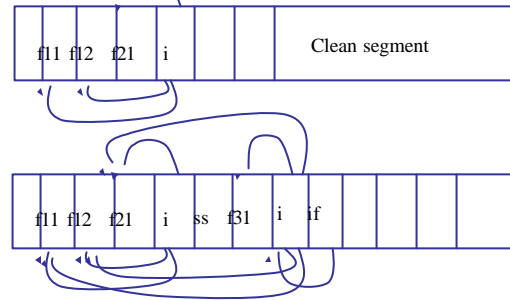
In LFS, all block and metadata allocation is log-based.

- LFS views the disk as “one big log” (logically).
- All writes are clustered and sequential/contiguous.
 - Intermingles metadata and blocks from different files.
- Data is laid out on disk in the order it is written.
- No-overwrite allocation policy: if an old block or inode is modified, write it to a new location at the *tail* of the log.
- LFS uses (mostly) the same metadata structures as FFS; only the allocation scheme is different.
 - Cylinder group structures and free block maps are eliminated.
 - Inodes are found by indirecting through a new map (the *ifile*).

Writing the Log in LFS

- LFS "saves up" dirty blocks and dirty inodes until it has a full *segment* (e.g., 1 MB).
 - Dirty inodes are grouped into block-sized clumps.
 - Dirty blocks are sorted by (*file, logical block number*).
 - Each log segment includes summary info and a checksum.
- LFS writes each log segment in a single burst, with at most one seek.
 - Find a free segment "slot" on the disk, and write it.
 - Store a back pointer to the previous segment.
 - Logically the log is sequential, but physically it consists of a chain of segments, each large enough to amortize seek overhead.

Example of log growth



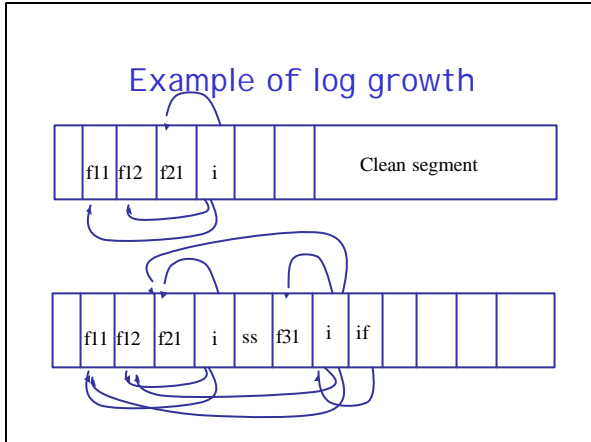
Writing the Log: the Rest of the Story

- LFS cannot always delay writes long enough to accumulate a full segment; sometimes it must push a *partial segment*.
 - fsync, update daemon, NFS server, etc.
 - Directory operations are synchronous in FFS, and some must be in LFS as well to preserve failure semantics and ordering.
- LFS allocation and write policies affect the buffer cache, which is supposed to be filesystem-independent.
 - Pin (*lock*) dirty blocks until the segment is written; dirty blocks cannot be recycled off the free chain as before.
 - Endow indirect blocks with permanent logical block numbers suitable for hashing in the buffer cache.

Cleaning in LFS

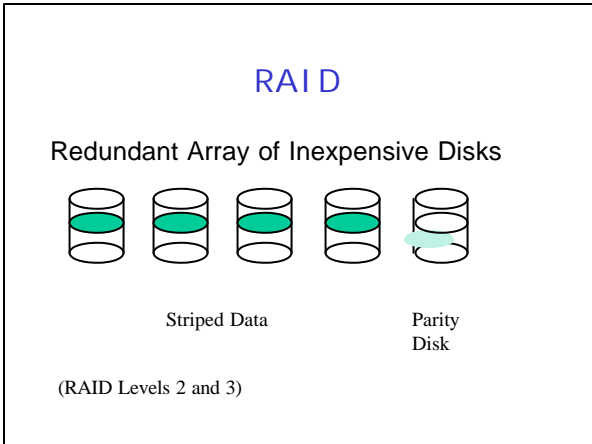
What does LFS do when the disk fills up?

- As the log is written, blocks and inodes written earlier in time are superseded ("killed") by versions written later.
 - files are overwritten or modified; inodes are updated
 - when files are removed, blocks and inodes are deallocated
- A cleaner daemon compacts remaining live data to free up large hunks of free space suitable for writing segments.
 - look for segments with little remaining live data
 - benefit/cost analysis to choose segments
 - write remaining live data to the log tail
 - can consume a significant share of bandwidth, and there are lots of cost/benefit heuristics involved.

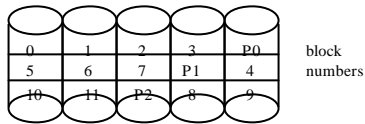


- ### What to do about Disks?
- ✓ Avoid them altogether! Caching
 - ✓ Disk scheduling
 - Idea is to reorder outstanding requests to minimize seeks.
 - ✓ Layout on disk
 - Placement to minimize disk overhead
 - Build a better disk (or substitute)
 - Example: RAID

- ### Build a Better Disk?
- “Better” has typically meant density to disk manufacturers - bigger disks are better.
 - I/O Bottleneck - a speed disparity caused by processors getting faster more quickly
 - One idea is to use parallelism of multiple disks
 - **Striping** data across disks
 - Reliability issues - introduce redundancy



RAID Level 5

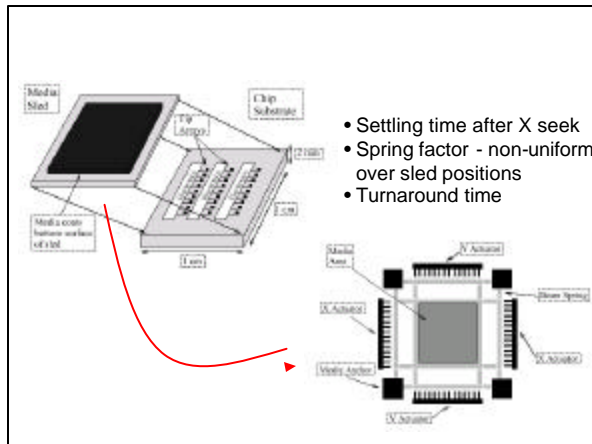


Distribute parity info as well as data over all disks

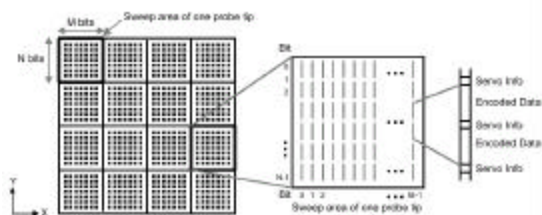
MEMS-based Storage

Griffin, Schlosser, Ganger, Nagle

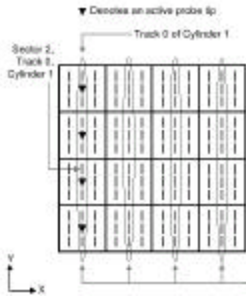
- Paper in OSDI 2000 on OS Management
- Comparing MEMS-based storage with disks
 - Request scheduling
 - Data layout
 - Fault tolerance
 - Power management



Data on Media Sled



Disk Analogy



- 16 tips
- $M \times N = 3 \times 280$
- Cylinder – same x offset
- 4 tracks of 1080 bits, 4 tips
- Each track – 12 sectors of 80 bits (8 encoded bytes)
- Logical blocks striped across 2 sectors

Logical Blocks and LBN



- Sectors are smaller than disk
- Multiple sectors can be accessed concurrently
- Bidirectional access

Comparison

MEMS

- Positioning – X and Y seek (0.2-0.8 ms)
- Settling time 0.2ms
- Seeks near edges take longer due to springs, turnarounds depend on direction – it isn't just distance to be moved.
- More parts to break
- Access parallelism

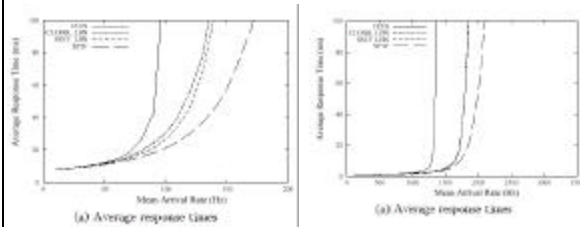
Disk

- Seek (1-15 ms) and rotational delay
- Settling time 0.5ms
- Seek times are relatively constant functions of distance
- Constant velocity rotation occurring regardless of accesses

Specific Parameters for Simulation Study

device capacity	3.2 GB
number of tips	6400
maximum concurrent tips	1280
stated acceleration	800.6 m/s ²
stated access speed	28 mm/s
constant settling time	0.22 ms
spring factor	75%
per-tip data rate	0.7 Mbit/s
media bit cell size	40x40 nm
bits per tip region (MxN)	2500x2440
data encoding overhead	2 bits per byte
servo overhead per 8 bytes	10 bits (11%)
command processing overhead	0.2 ms/request
on-board cache memory	0 MB
external bus bandwidth	100 MB/s

Request Scheduling

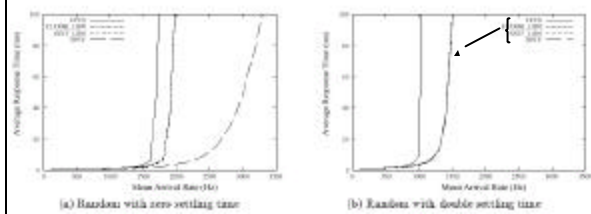


Quantum Atlas 10K Disk

MEMS

Random Workload

Impact of Settling Time



(a) Random with zero settling time

(b) Random with double settling time

Y matters – only captured by SPTF

X dominates

Data Placement

- Offset from center matters to seek time – small data are placed in centermost subregions
- Positioning is relatively insignificant for large transfers – sequential streaming data placed in outer subregions
- Compared to organpipe policy– most frequently accessed data in middle disk tracks
- All better than nothing at all

Failure and Power

- Error Correcting Code computed horizontally across tips (missing sector, bad tip) and vertically within sector (bad sector)
- Remap sector under spare tip allocated in each track
- Idle mode stops sled and powers down electronics
- Restart is fast 0.5ms and no power spike to “spin up.” Immediate-idle (no timeout policy).