



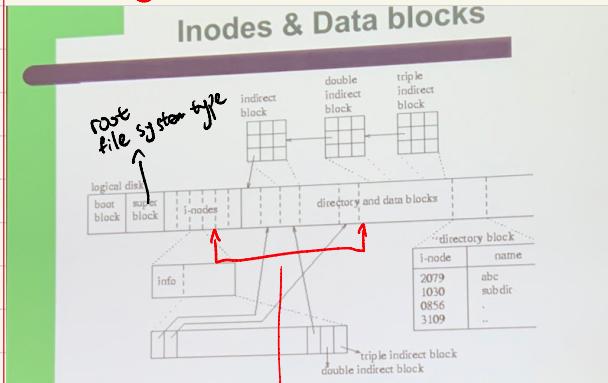
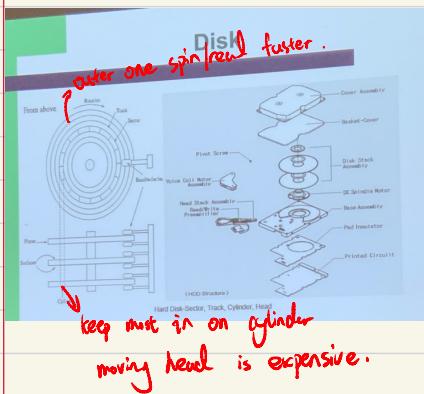
11/4

- Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, "A Fast File System for Unix," ACM Transactions on Computer Systems, 2(3), August 1984, pp. 181-197.

Q: In FFS, reading is always at least as fast as writing. In old UFS, writing was 50% faster. Why is this?

- Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System," Proceedings of the 13th ACM Symposium on Operating Systems Principles, December 1991.

Q: When we want to read a block in LFS on disk, how do we figure out where it is?



- ### Main Point
- What is the main point of this paper?
 - to describe performance improvements of the Unix file system
 - also new features that have been waiting in the wings
 - What is the main constraint on the changes made?
 - maintain standard Unix I/O programming interface
 - reduces # programs that have to be changed
- keep same interface.*

Problems in Previous FS (Group)

Problems	Solutions in FS
Random data block placement in aging file system	allocate data in same file close together (Cylinder groups)
Inodes allocated far from data blocks	inodes and data close together
Low bandwidth utilization	larger block size (1024 to 4096)
Small max file size	larger block size (w/ 4096, only need two levels of indirection to reach 2^32 fragments (different sizes))
Waste from large block size	
recovery	replicated superblocks (superblocks not placed in same track to minimize effect of platter failures)
Device oblivious	parameterize file systems to device characteristics - reformat dat (solaris) allocate rotationally optimal blocks

abandon since disk are shorter or tall. disk need to relocate head to a diff plate

e.g. Giving some padding since disk is spinning.

FFS Allocation

- FFS has a two-level allocator, global and local. What does the global allocator try to achieve?
 - localize related data within a cylinder group
 - e.g., all inodes for a directory in same cylinder group
 - distribute unrelated data across cylinder groups
 - e.g., new directories placed in cyl group with >avg #nodes
- What does the local allocator try to achieve?
 - optimal local placement
 - rotationally optimal block in same cylinder
 - block in same cyl group
 - search through cyl groups
- What is one requirement of the new FFS?
 - requires a minimum of free space so that cyl block allocations can be done: 10%
 - this is why you will see some file systems with >100% utilization

11/4/21

Performance

- The paper states that the throughput rate is strongly tied to the amount of free space maintained. Why?
 - if free space is low, blocks have to be spread across cyl groups
- In FFS, reading is always at least as fast as writing. In old UFS, writing was 50% faster. Why is this?
 - writes are async. FS can **reorder requests** to minimize total seek time
 - In FFS, blocks are placed on disk in a much better layout
 - even with sync reads, placement roughly corresponds to a good ordering of disk requests
 - writes have higher overhead because they write to new blocks

*write Cache, write and
write it once.*

*read
don't
double*

New Features

- Interesting to see that these features were not in the original Unix FS implementation:
 - long file names
 - file locking
 - with advent of NFS, became much less useful
 - symbolic links
 - rename!
 - quotas

Discussion

- Some of these things are obsolete. Which ones?
 - device parameters
 - cylinders in today's disks vary in size depending upon radius
 - host no longer knows cylinder and track boundaries
 - rotation allocation not done
 - blocks allocated sequentially
 - drives do read ahead if there is a gap in seq commands
- Current Unix filesystems (and NT) are extent-based w/ journaling
 - extends
 - allocate multiple consecutive blocks on disk
 - can read/write multiple blocks at once to amortize seek (cf VMS, LFS)
 - journaling: as described in related work in LFS paper

//Second

Before we get into this paper...

- LFS spawned a vocal controversy between Ousterhout and Seltzer about whether LFS was better than an extent-based FFS
- The main complaint was that cleaning required too much overhead, and that it was a source of hidden costs.
- The issue has never been settled definitively, although most disk-based file systems today are extent-based FFS rather than LFS.

Overview

- What is the main point of this paper?
 - file system designed to exploit h/w and workload trends
- What are these trends?
 - disk b/w is scaling, but latency is not
 - large main memories
 - large file buffer caches
 - absorb a greater fraction of read requests
 - use for write buffers as well
 - coalesce small writes into large writes
 - workloads
 - small file access

CSE221 - Operating Systems

Say: ok

Small I/O: stop, get some help

Buffer cache \Rightarrow hit them, no need worry!

Motivation

- What are problems with FFS?
 - layout
 - possibly related files are physically separated
 - inodes are separate from files
 - directory entries are separate from inodes
 - bottom line is only 5% disk b/w utilization
 - but FFS says 15-50%... what gives?
 - small vs large I/Os, accounting for seeks to inodes then data blocks
 - synchronous writes
 - metadata requires synchronous writes
 - w/ small files, most writes are metadata and hence
 - sync writes defeat write caches

synch

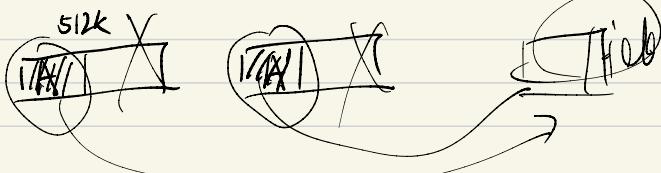
→ has to follow the sequence.

A New Motivation

- New type of memory technology
 - SSD, Flash Memory, etc
- 
- 
- (+) Low cost per bit
 - (-) Mechanical movement (SPM & VCM)
 - (-) High power consumption (10-15W)
 - (-) Heavy weight compared to flash
 - (+) Erase before Write
 - (+) Erasing operation in the unit of block (not page)
 - (+) Maximum # of erase operations per cell
 - (+) High cost per bit

erase then write
so, if there is a hot spot, frequently update.
data will not be readable

LFS avoid hot spot
no overwirte.



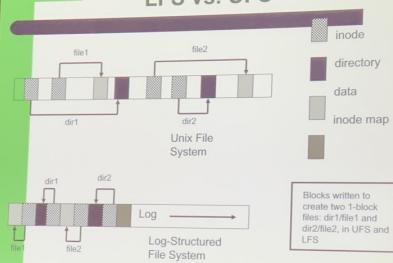
LFS

- At a high-level, how does LFS work?
 - treat disk as a single log for appending
 - buffer small writes into very large writes to maximize use of disk b/w
 - all info written to disk is appended to log
 - data, attrs, inodes, dirs, all other metadata
- What are the two key challenges to implementing LFS?
 - how to find data in the log
 - reading data, invalidating deleted or overwritten data
 - how to manage free space on disk
 - will run out of disk space at some point
 - need to recover deleted blocks in old log

→ the IMAP (all pointer to INODE)

will always write to the end of the disk.

LFS vs. UFS



JFS: Both used

has a buffer like LFS
other file are normal FFS

Locaton

- How does LFS solve this problem?
 - level of indirection
 - use index tables (inode maps)
 - inode maps map file # to inode location
 - they are also written into the log
 - pointers to all inode map blocks are kept in checkpoint region
 - checkpoint region has a fixed location, so is easy to find
- Level of indirection adds overhead.
- How does LFS minimize this overhead?
 - keep inode map blocks in memory (they are small)

Free Space Management

- Problem is that append-only, you will run out of disk space. Hence, need to identify and reclaim free space.
- What are the two choices for reusing free space?
 - 1) Threading
 - move parts of the log between existing data
 - no data is ever moved
 - Problem?
 - high fragmentation, small extents, no large writes possible
 - 2) Copying
 - move data by copying
 - compact live data, free up large extents
 - Problem?
 - copying data that never goes away has a high overhead

Solution

LFS Segment Cleaning

- What is the solution in LFS?
 - coarse techniques
 - divide log into segments (512KB or 1MB)
 - thread segments within disk
 - do free space management at granularity of segments
 - to reuse blocks, do segment clearing
 - mark segments
 - copy live data to end of log
 - free up the segment
- Need to identify
 - which blocks (of size 512 usually) are live in this segment
 - the file to which the blocks belong so that inode is updated

threading between segments

copy on segment

guarantee to have at least one seq

Cleaning Policy

- policy issues to decide:
 - 1) when do cleaning?
 - 2) how many segments to clean at a time?
 - 3) which segments should be cleaned?
 - 4) how to group live blocks when writing out?
- 1 and 2 are used simply.
- Ideally, which segments to clean?
 - want to clean segments with very little live data
 - this minimizes overhead
- How to achieve this?
 - segregate segments into mostly full, mostly empty
 - only clean mostly empty ones

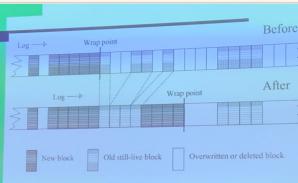
It is because

recent one might be
replace quickly,

just write and you
done even need cleaning!

Which one to clean?

- First try was a greedy algorithm:
 - choose least-utilized segments to clean
- Why didn't this work out well with workloads with locality?
 - cold segments are modified infrequently
 - hence, cold segments tie up a lot of free disk space
 - because they are cold, this space is tied up for a long time
- Lesson:
 - free space in cold segments is more valuable than free space in hot segments



?

•

Crash Recovery

- What are the two mechanisms for recovery in LFS?
 - checkpoints
 - roll-forward
- Checkpoints are every 30 seconds.
 - probably good enough
- Wanted to use roll-forward to increase times between checkpoints.
 - complicated, though

Experience

- What do you think about the use of microbenchmarks here?
 - good use of microbenchmarks to focus on specific behaviors
- Why does LFS do worse vs. FFS on random write/sequential read?
 - FFS organizes file blocks sequentially, no matter how written
 - LFS organizes file blocks in order written
- One of the good things about this paper is that they report usage statistics from a production system. Such evaluations of research prototypes are few and far between. What did they find in the production system?
 - cleaning overheads not as bad as predicted
- Why is that?
 - long files written and deleted as a whole
 - cold files in reality are much colder than in simulation

Discussion

- So what do you think about the need for cleaning? Is it the Achilles heel of LFS?
- Do you believe in the LFS approach?
- NetApp's WAFL is the one file system that has LFS-like features.
 - WAFL is unusual because it is a combination of NFS server + file system + RAID storage system all wrapped up together.
 - Like LFS, it does not do updates in place as typical file systems do. It collects updates and writes them as a single I/O to new free locations on disk rather than the original locations. A reason for doing so is to support snapshots of the file system over time, like Plan9. If you're going to have snapshots, then you want to keep the old data around and not overwrite it.

Checkpoint is at fixed location

Since we may circle the disk,
we need to scan the whole disk.

the checkpoint may not be the latest, but we can scan and catch from that point.