Lecture 18: Log Structured Filesystem

601.418/618 Operating Systems

David Hovemeyer

April 15, 2024

Agenda

- Problem with writes
- Log Structured Filesystem
  - Segments
  - Data Structures
  - Cleaning

Acknowledgments: These slides are shamelessly adapted from Prof. Ryan Huang's Fall 2022 slides, which in turn are based on Prof. David Mazières's OS lecture notes.

File Systems Examples

BSD Fast File System (FFS)

- What were the problems with the original Unix FS?
- How did FFS solve these problems?

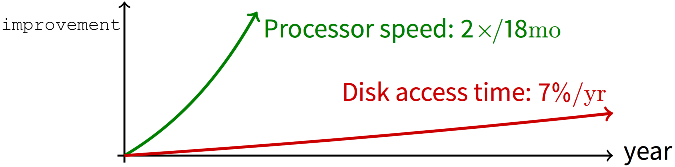
Log-Structured File System (LFS)

- What was the motivation of LFS?
- How did LFS work?

LFS: Log-structured File System

An influential work designed by Mendel Rosenblum (VMWare co-founder) and John Ousterhout

- A classic example of system designs driven by technology trends Motivation
- Faster CPUs: I/O becomes more and more of a bottleneck



- More memory: file cache is effective for reads
- **Implication**: writes compose most of disk traffic

Motivation

Problems with previous FS

- Perform many small writes
  - Good performance on large, sequential writes, but many writes are still small, random
- Synchronous operation to avoid data loss
- Depends upon knowledge of disk geometry (Fast File System)

LFS Idea

Insight: treat disk like a tape drive

- Best performance from disk for sequential access
- What is Fast File System's insight about disk?

File system buffers writes in main memory until "enough" data

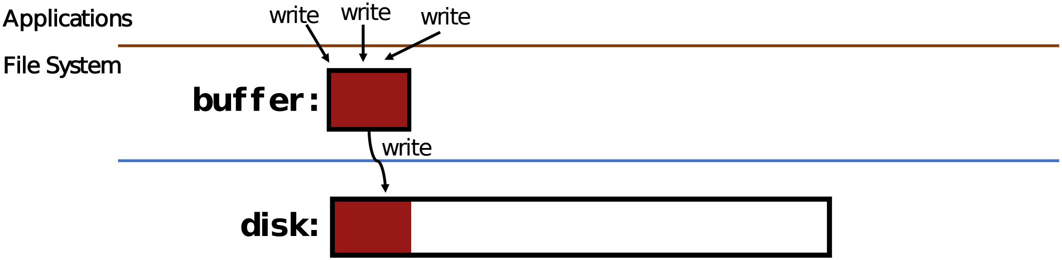
- How much is enough?
- Enough to get good sequential bandwidth from disk (MB)
- Unit called a "*segment*"

Write Data to a Sequential Log

Write buffered data to new segment on disk in a sequential log

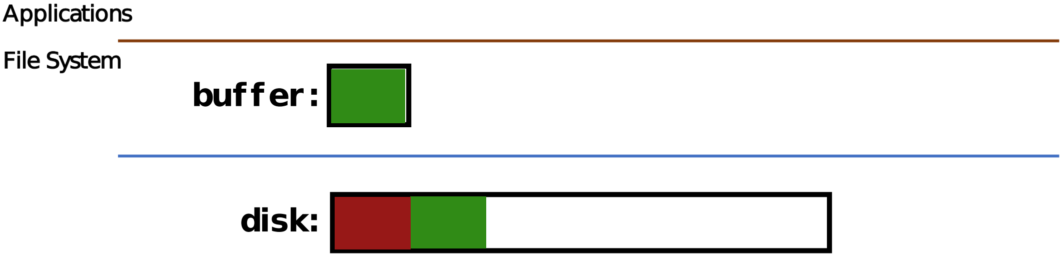
- Transfer all updates into a series of sequential writes
- **Do not overwrite old data on disk**
  - i.e., old copies left behind
- Write both data and metadata in one operation

Write in LFS



Absorb many small writes into one buffer write!

Write in LFS



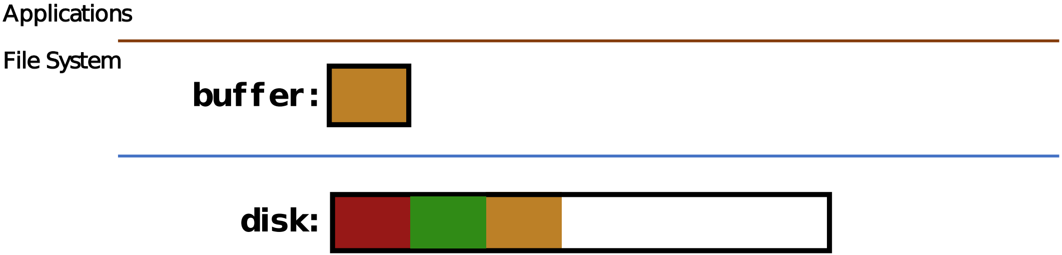
Applications

File System

buffer:

disk:

Write in LFS



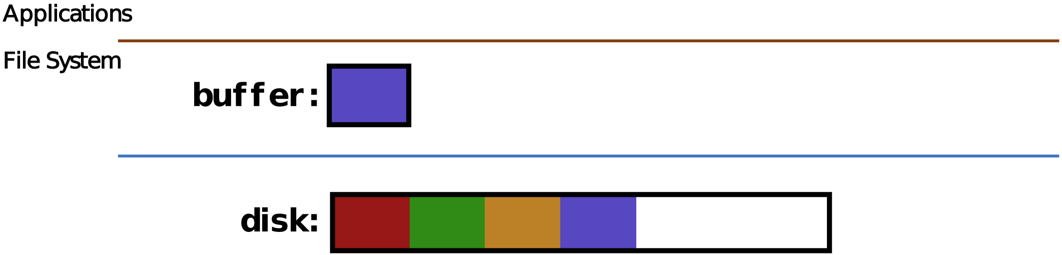
Applications

File System

buffer:

disk:

Write in LFS



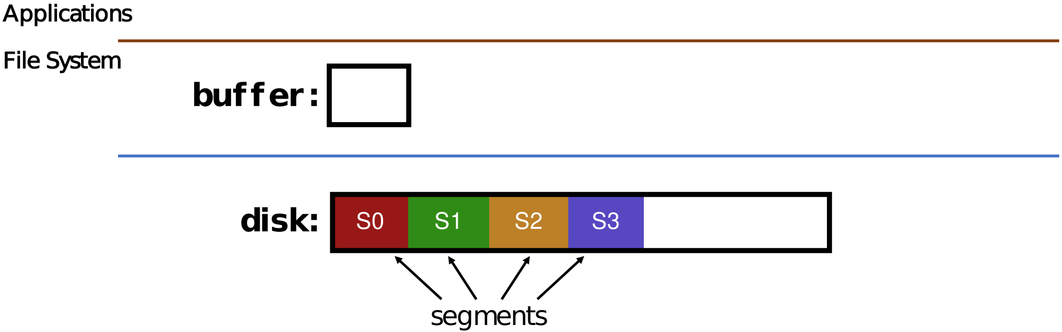
Applications

File System

buffer:

disk:

Write in LFS



Applications

File System

buffer:

disk:

S0

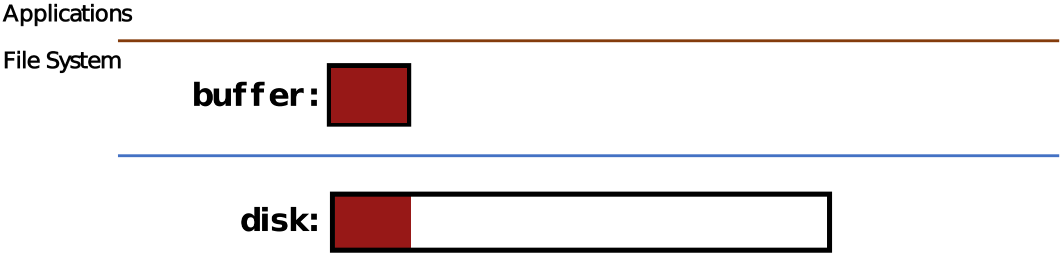
S1

S2

S3

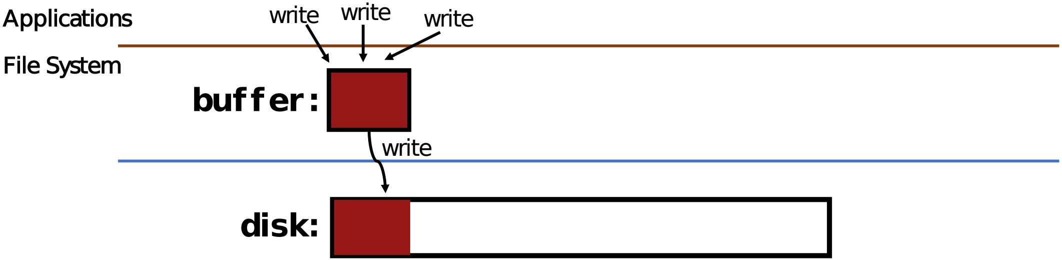
segments

Write in LFS

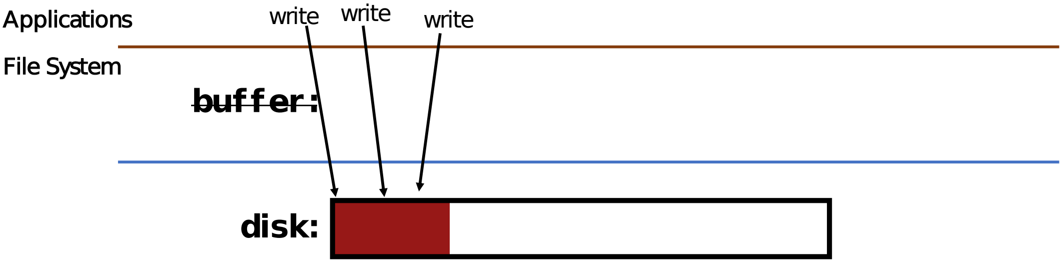


Why do we buffer the write?

Write in LFS


Applications
write
write
write
File System
buffer:
write
disk:

Write in LFS



Why not directly write to the log on disk sequentially?

- Sequential write alone is not enough
- Disk is constantly rotating!
- Must issue a large number of **contiguous** writes

Pros And Cons

Pros

- Always large sequential writes → good performance
- No knowledge of disk geometry
  - Assume sequential better than random

Potential problems

- How do you find data to read?
- What happens to metadata during write?
- What happens when you fill up the disk?

Read in LFS

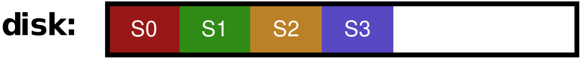
Same basic structures as Unix

- Directories, inodes, indirect blocks, data blocks
- Reading data block implies finding the file's inode
    - Unix FS: inodes in a fixed region (array) on disk
    - LFS: inodes spread around on disk

Solution: inode map (*imap*) indicates where each inode is stored

- Can keep cached copy in memory
- inode map written to log with everything else
- Periodically written to known checkpoint location on disk for crash recovery

Attempt 1: Data Structures for LFS



What data structures from FFS can LFS remove?

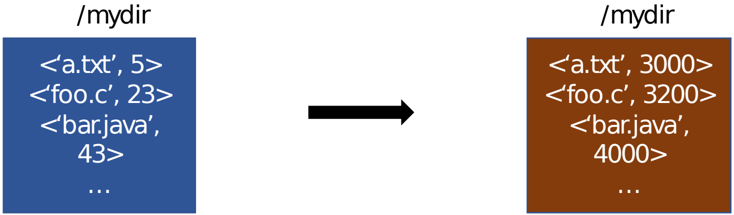
- allocation structs: data + inode bitmaps (why?)

What type of structure is much more complicated?

- Inodes are no longer at fixed offset!
- Use current offset on disk instead of table index for name
- Note: when inode updated, inode number changes! (why?)

Attempt 1: Data Structures for LFS

Directory Entry



/mydir

<‘a.txt’, 5>
<‘foo.c’, 23>
<‘bar.java’,
43>
...

/mydir

<‘a.txt’, 3000>
<‘foo.c’, 3200>
<‘bar.java’,
4000>
...

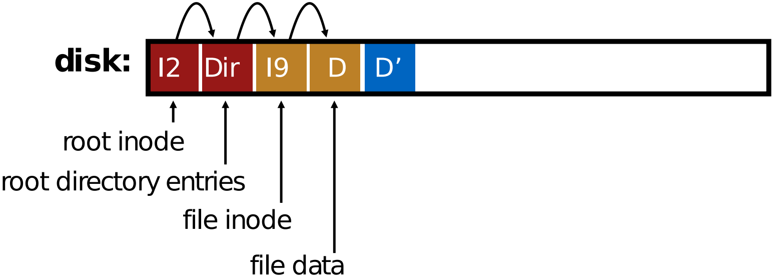
Previously,
each dir entry is
<name, inode #>

Now,
each dir entry is
<name, disk offset>

Would this attempt work?

Attempt 1: Overwrite Data in LFS

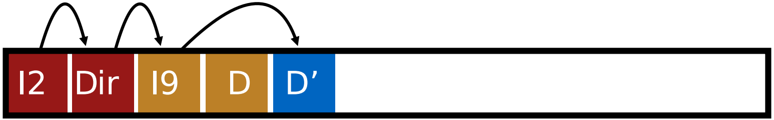
Overwrite data in `/file.txt`:



How to update inode 9 to point to new D’ ?

Attempt 1: Overwrite Data in LFS

Overwrite data in `/file.txt`:

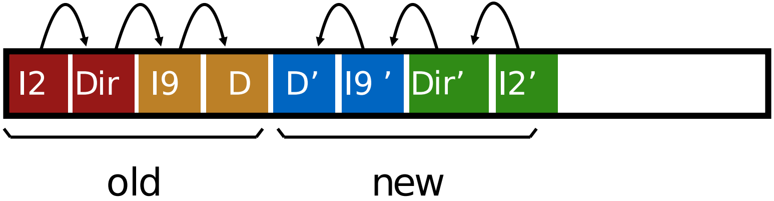


Can LFS update inode 9 to point to new D’?

- NO! This would be a random write…

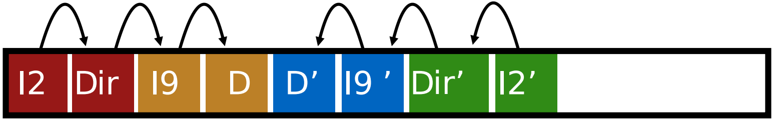
Attempt 1: Overwrite Data in LFS

Overwrite data in `/file.txt`:



Must update **all** structures in sequential order to log

Attempt 1: Problem w/ Using Offset



Problem:

- For every data update, must propagate updates all the way up directory tree to root

Why?

- When we copy & modify the inode, its location (disk offset) changes

Solution:

- Keep inode numbers constant; don't base name on disk offset

Data Structures for LFS (attempt 2)

What data structures from FFS can LFS remove?

- allocation structs: data + inode bitmaps

What type of struct is much more complicated?

- Inodes are no longer at fixed offset
- ~~Use current offset on disk instead of table index for name~~
- Keep inode number in dir constant
- Use *imap* structure to map *inode number* → **most recent** inode location on disk

FFS found inodes with math. How now?

- *imap*

Where to keep imap?

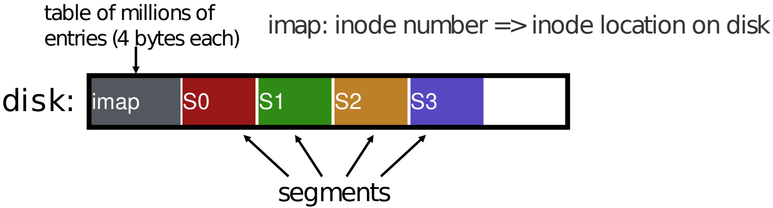


table of millions of entries (4 bytes each)
imap: inode number => inode location on disk
disk:
imap
S0
S1
S2
S3
segments

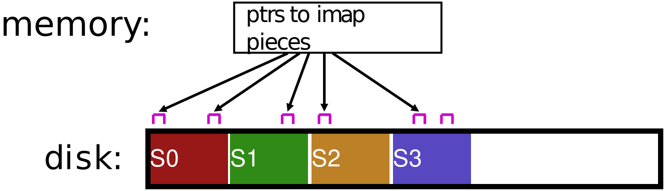
Where can imap be stored? Dilemma:

1. imap too large to keep in memory
2. don't want to perform random writes for imap

Solution: Write imap in segments

- Keep pointers to pieces of imap in memory

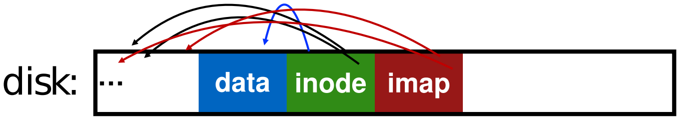
Solution: imap in segments



Solution:

- Write imap in segments
- Keep pointers to pieces of imap in memory
- Keep recently accessed imap cached in memory

Example Write



Solution:

- Write imap in segments
- Keep pointers to pieces of imap in memory
- Keep recently accessed imap cached in memory

Disk Cleaning

When disk runs low on free space

- Run a disk cleaning process
- Compacts live information to contiguous blocks of disk

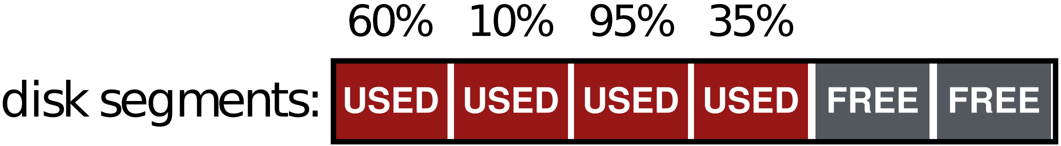
Problem: long-lived data repeatedly copied over time

- Solution: partition disk into segments
- Group older files into same segment

LFS reclaims segments (not individual inodes and data blocks)

- Want future overwrites to be to sequential areas
- Tricky, since segments are usually partly valid

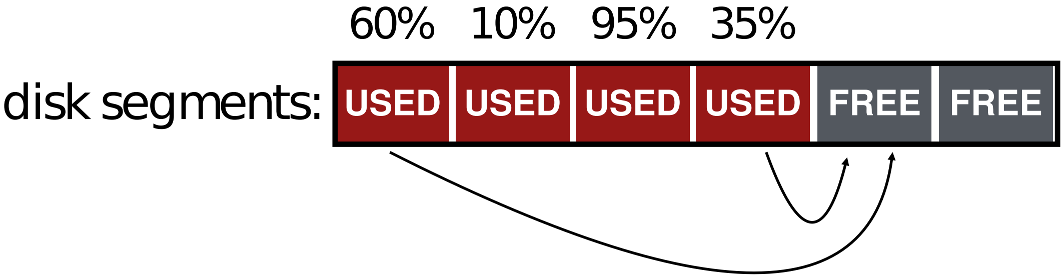
Cleaning: Copy & Compact Segments



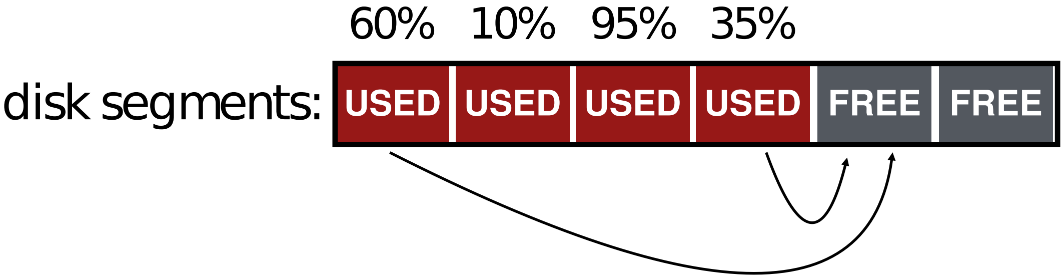
60% 10% 95% 35%

disk segments: USED USED USED USED FREE FREE

Cleaning: Copy & Compact Segments



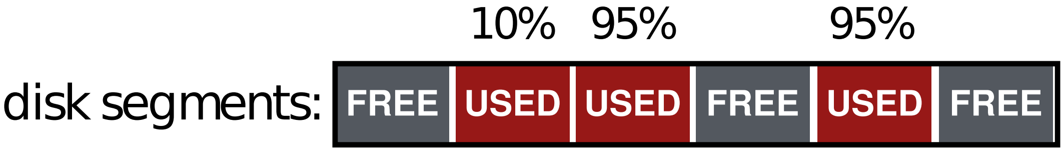
Cleaning: Copy & Compact Segments



compact 2 segments to one

- When move data blocks, copy new inode to point to it
- When move inode, update imap to point to it

Cleaning: Copy & Compact Segments



disk segments:

	10%	95%		95%	
FREE	USED	USED	FREE	USED	FREE

release the two input segments

Next Time

`fsck`, journaled filesystems