CS-446/646

Log-structured File System

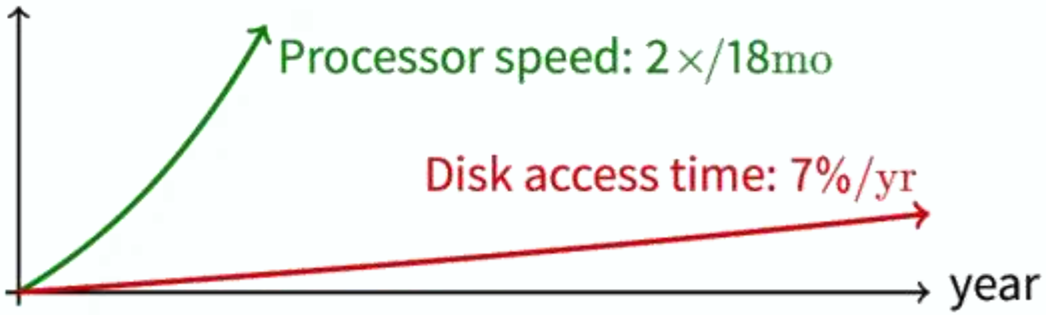
C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Log-structured File System

***Log-structured File System* (LFS)**

- Influential work designed by Mendel Rosenblum (VMWare co-founder) & John Ousterhout
  - 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

Log-structured File System

LSF Motivation - Problems with previous *Filesystems:*

- 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 (i.e. like *Sequential Logging* media)
  - Best performance we can get from Disk if when performing *Sequential* Access
    - Remember: FFS' insight about Disk: Leverage *Locality* via *Cylinder Groups*
- *Filesystem* **buffers** writes in Main *Memory* until "enough" data are in
  - Quantify how much "enough" is:
    - Enough to get good *Sequential* Bandwidth from Disk (MB/s)
  - Buffered Unit called a "*Segment*"

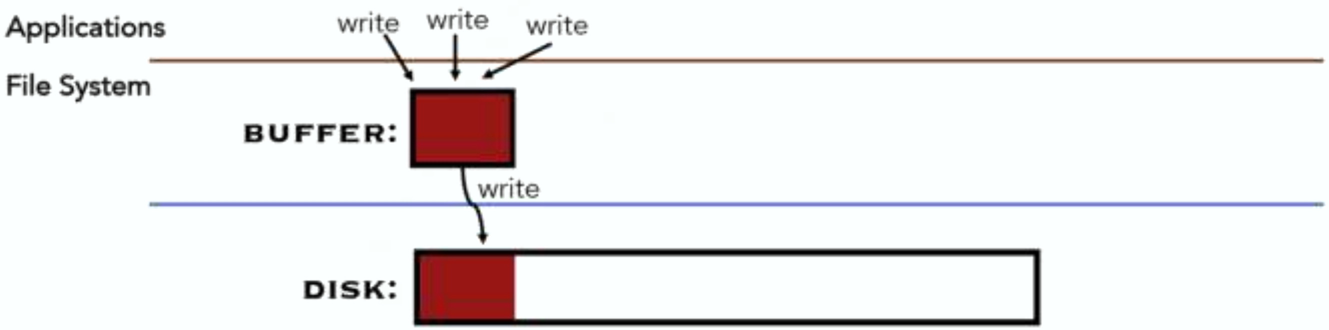
Log-structured File System

Writing Data to a *Sequential Log*

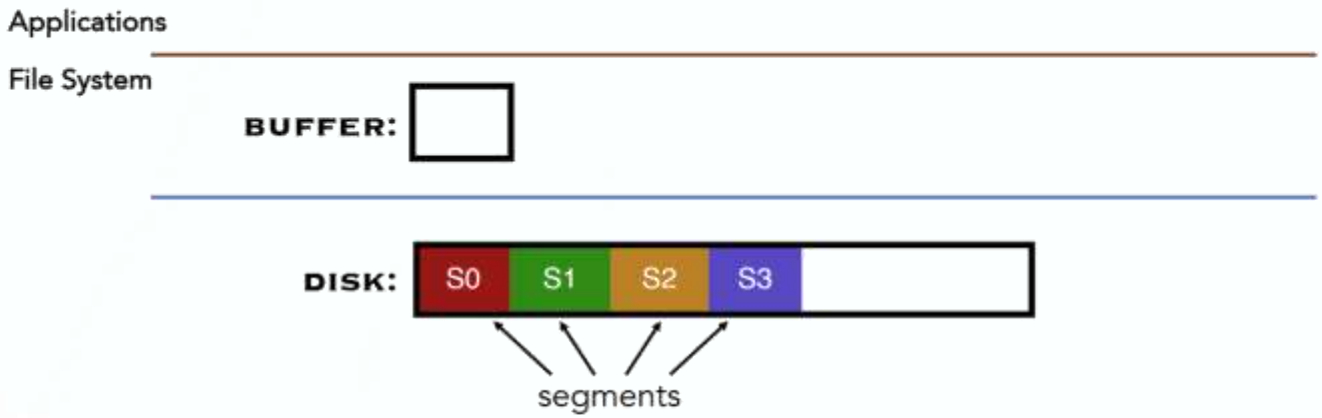
- Write buffered data in a *Sequential Log*
  - All updates (write operations) are delayed, and take place in a series of *Sequential* writes
  - Write both Data and Metadata for all *Files* in one intermixed operation
  - **Do not overwrite old data on Disk**
    - i.e. old copies left behind

Write in LFS

- Absorb many small writes into one buffered write



- Data written in *Segments*



Log-structured File System

Write in LFS

Why is buffering is required? (i.e. instead of directly writing on Disk, just sequentially?)

- *Sequential* writing alone is not enough
  - Disk is constantly spinning
  - To be efficient, must issue a **burst** of contiguous writes

Advantages

- Always large *Sequential* writes → Good Performance
- No need for knowledge of Disk Geometry
  - Scheme assumes *Sequential* will "naturally" exhibit better Performance than *Random*

Potential problems

- How do you find the Data you want to read?
  - *Remember*: Updates written *Sequentially* (like *Sequential Logging* on a Tape), with old copies left behind
    i.e. what happens with *File* Metadata?
- What happens when Disk is filled-up?

Log-structured File System

Read in LFS

- Same basic structures as prior (Unix) *Filesystems*
  - *Directories*, *inodes*, *Indirect Blocks*, *Data Blocks*
  - Reading *Data Block* implies first finding the *File*'s *inode*
    - Unix *Filesystem* / FFS: *inodes* in a fixed *Region* (*inode Table*) on Disk
    - LFS: *inodes* spread on *Disk* (Remember: *Sequential Logging* of Data and Metadata)

Solution – One slightly different structure

- An *inode Map* (*imap*) : Holds the (most recent) Disk offset for each *inode*
  - *inode Map* small enough to keep in *Memory*
  - *inode Map* which maintains the state of our *Sequentially-Logged Filesystem* has to be written to Disk as well:
    - Periodically written to known checkpoint *Location* on Disk for Crash Recovery

Log-structured File System

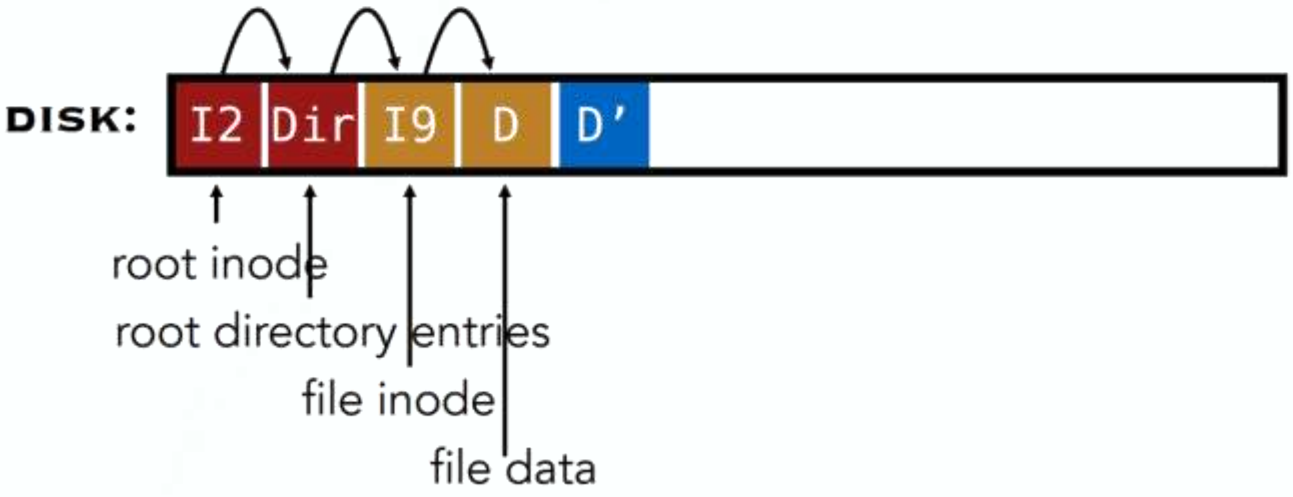
Data Structures for LFS – Why *imap*? (Attempt 1)

- LFS vs FFS: Data structures we can get rid of:
  - Allocation structs: *inode Bitmaps* and *Data*
    - *Remember:* Information is written (i.e. updated) *Sequentially*
- New structure of our Data on Disk becomes more complicated
  - (Updated) *inodes* are no longer at fixed offset!
  - For internal OS handling of names, instead of an *inode's* index in the *inode Table* (no longer exists), would have to use the current offset on Disk as its *i-number*
    - But: When updating *inode*, the *i-number* (Disk offset by this approach) has to change!

Log-structured File System

Data Structures for LFS – Why *imap*? (Attempt 1)

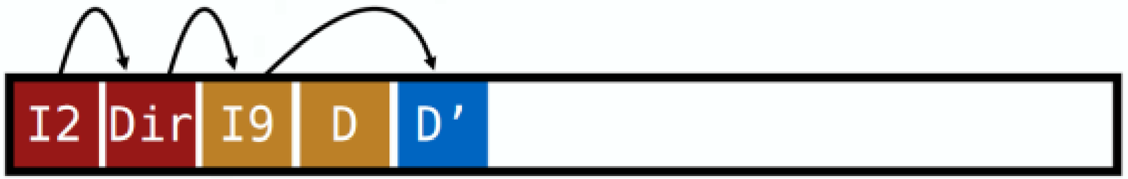
- *Example:* Overwrite data in **`/file.txt`**
- Remember: Now in a *Directory (File)*, each *inode Entry* becomes: **`<name, offset#>`** instead of **`<name, inode#>`**



- How to update *inode#* 9 to point to new **`D'`** ?

Data Structures for LFS – Why *imap*? (Attempt 1)

- *Example:* Overwrite data in `/file.txt`
- Remember: Now in a *Directory (File)*, each *inode Entry* becomes: `<name, offset#>` instead of `<name, inode#>`

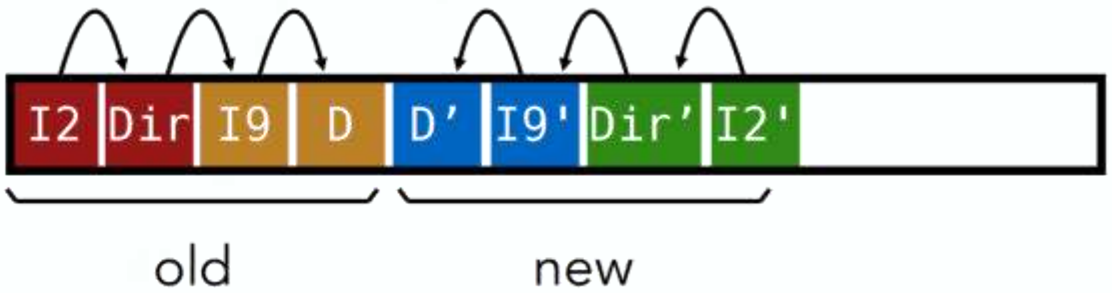


- LFS cannot update *inode# 9 Entry* to point to new `D'`
  - Not a *Sequentially-Logged* write
    - (a *Random* Access write)

Log-structured File System

Data Structures for LFS – Why *imap*? (Attempt 1)

- *Example:* Overwrite data in **`/file.txt`**
- Remember: Now in a *Directory (File)*, each *inode Entry* becomes: **`<name, offset#>`** instead of **`<name, inode#>`**



- Must update all structures in *Sequential* order to our *Sequential Log*

Problem:

- For every Data update, must propagate updates all the way up *Directory* Tree to *Root Dir*
  - Why?
    When we copy & modify the *inode*, its *Location* (Disk offset) changes

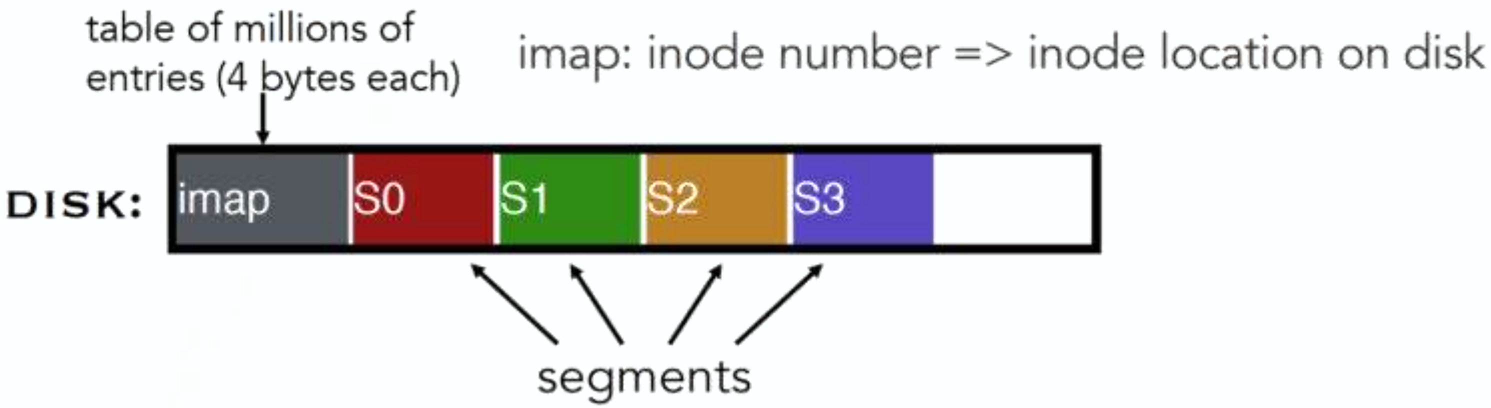
Data Structures for LFS – Why *imap*? (Attempt 2)

- Solution: Keep *inode #s (i-numbers)* constant; don't base name on Disk offset
- LFS vs FFS: Data structures we can get rid of:
  - Allocation structs: *inode Bitmaps* and *Data*
    - *Remember:* Information is written (i.e. updated) *Sequentially*
- New structure of our Data on Disk becomes more complicated
  - (Updated) *inodes* are no longer at fixed offset!
  - ~~For internal OS handling of names, instead of an *inode's* index in the *inode Table* (no longer exists), have to use the current offset on Disk as its *i number*~~
  - Keep *inode #* (*i-number*) in *Directory Entry* constant
  - Use *imap* structure to map: *inode #* → most recent *inode* offset on Disk
- FFS found *inodes* from *inode Table* – vs – LFS uses the *imap*

Where to keep *imap*

Where should the *imap* be stored? Dilemma:

- 1. *imap* too large to keep in *Memory*
- 2. Don't want to perform *Random* Access writes to update the *imap*

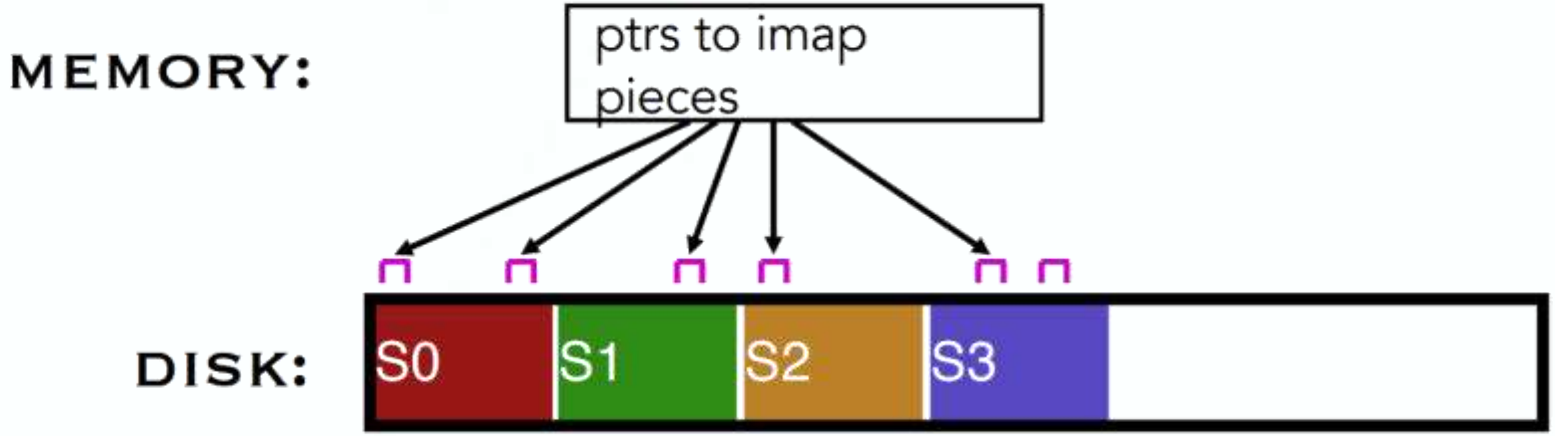


Solution: **Piecewise** write of the *imap* inside the *Segments*

- Keep Pointers to **pieces** of the *imap* in *Memory*

Log-structured File System

Solution: **Piecewise *imap* inside *Segments***



Solution:

- Piecewise write of the *imap* inside the *Segments*
- Keep Pointers to pieces of the *imap* in *Memory*
- Keep recently accessed *imap* parts cached in *Memory*

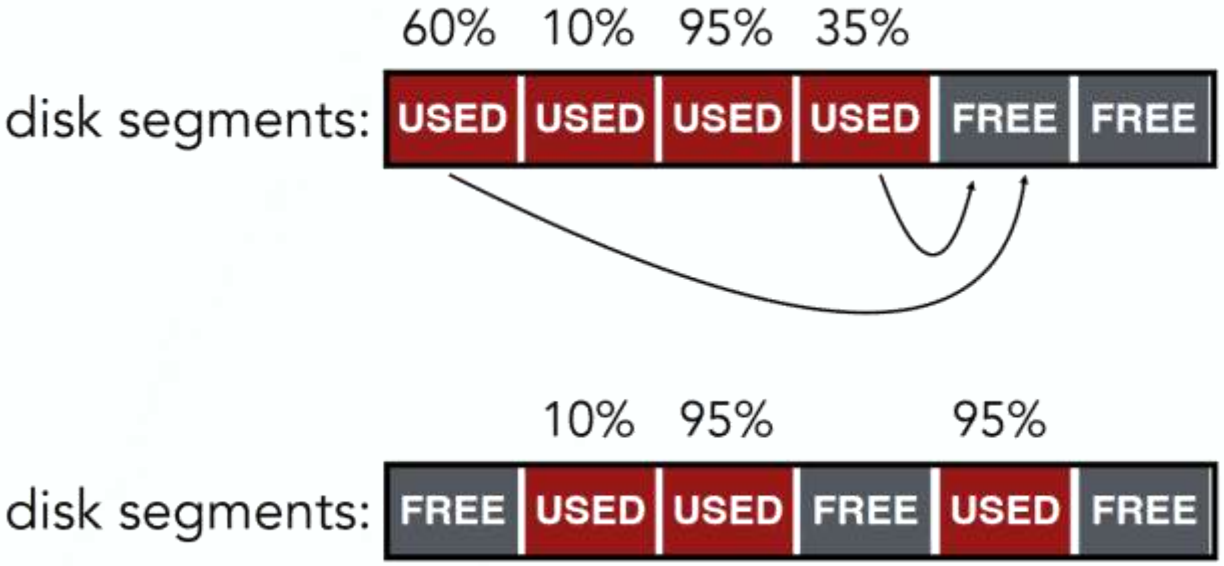
Disk Cleaning

Sequential Logging fills up Disk space fast

- When Disk runs low on free space
  - Run a Disk Cleaning utility
  - Compact 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*
    - Do not clean *Segments* with older files
- Disk Cleaner utility runs when Disk is not being used

Disk Cleaning – Copy & Compact *Segments*

- LFS reclaims *Segments* (not individual *inodes* and *Data Blocks*)
  - Want future overwrites to be to *Sequential* areas
  - Tricky, since *Segments* are usually partly valid



Compact 2 *Segments* into 1

- When moving *Data Blocks*, copy new *inode* to point to it
- When moving *inode*, update the *imap* to point to it

Release the 2 input *Segments*

CS-446/646

Time for Questions !