# Operating Systems

## Lecture 15: File Systems
## Part III: Log-structured File System

**Hong Xu**

https://github.com/henryhxu/CSCI3150

# Outline

- Key Idea: Writing Sequentially
- Indirect Mapping and Checkpoint Region
- Directories
- Garbage Collection
- Crash Recovery

# Overview

- In the early 90's, a new file system, the log-structured file system (LFS) was developed
- Motivation
  - Memory sizes were growing.
  - Large gap between random IO and sequential IO performance.
  - Existing File System perform poorly on common workloads.
- In this chapter, we study Log-Structured Filesystem (LFS).
  - How can a file system **transform all writes into sequential writes**?

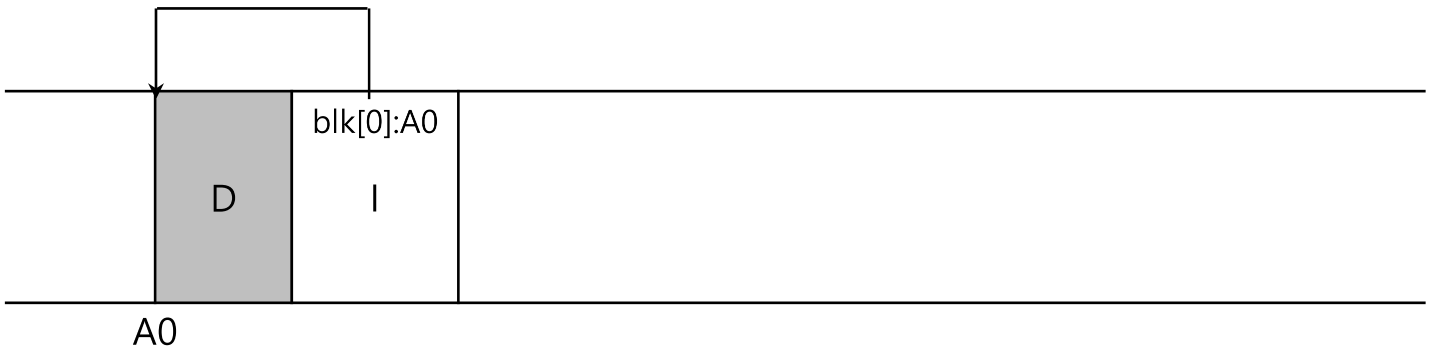
John Ousterhout

Mendel Rosenblum

# Writing to Disk Sequentially

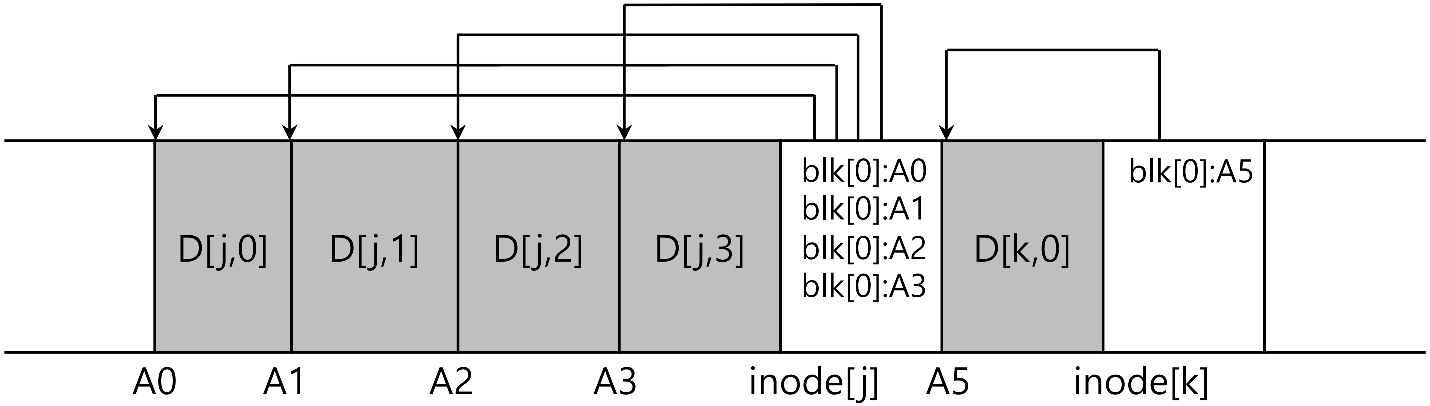
- How do we transform all updates to file-system state into a series of sequntial writes to disk?
  - data update


  - metadata needs to be updated too. (Ex. inode)



# Writing Sequentially, and Effeciently!

- Writing to the disk sequentially is not enough to guarantee the efficient writes.
  - Disk may rotate between the writes. → loose a single revolution between the writes.
- Write buffering.
  - Segment: a set of sequential writes that are written to the disk with a single unit.
  - Keep track of updates in **memory buffer**. ( a few Mbyte)
  - Write them to disk all at once, when it has sufficient number of updates.



# Issue #1: How Much to Buffer

- Time to write D MB

$$T_{write} = T_{position} + \frac{D}{R_{peak}}$$

- Effective write bandwidth

$$R_{effecitve} = \frac{D}{T_{write}} = \frac{D}{T_{position+}\frac{D}{R_{peak}}}$$

- We'd like to make the effective write bandwidth close to peak bandwidth with some fraction F (0<F<1)

$$R_{effecitve} = \frac{D}{T_{position+}\frac{D}{R_{peak}}} = \mathrm{F\ x}\ R_{peak}$$

# Issue #1: How Much to Buffer

- Then, D can be computed as follows

$$D = F \text{ x } R_{peak} \text{ x } (T_{position} + \frac{D}{R_{peak}})$$

$$D = \left(F \text{ x } R_{peak} \text{ x } T_{position}\right) + \left(F \text{ x } R_{peak} \text{ x } \frac{D}{R_{peak}}\right)$$

$$D = \frac{F}{1-F} \text{ x } R_{peak} \text{ x } T_{position}$$

- Example: Positioning time 10ms, peak transfer rate 100MByte/sec, we like to achieve 90% of the peak rate

D = 0.9*0.1*100 Mbyte/sec * 0.01 secs = 9 Mbyte

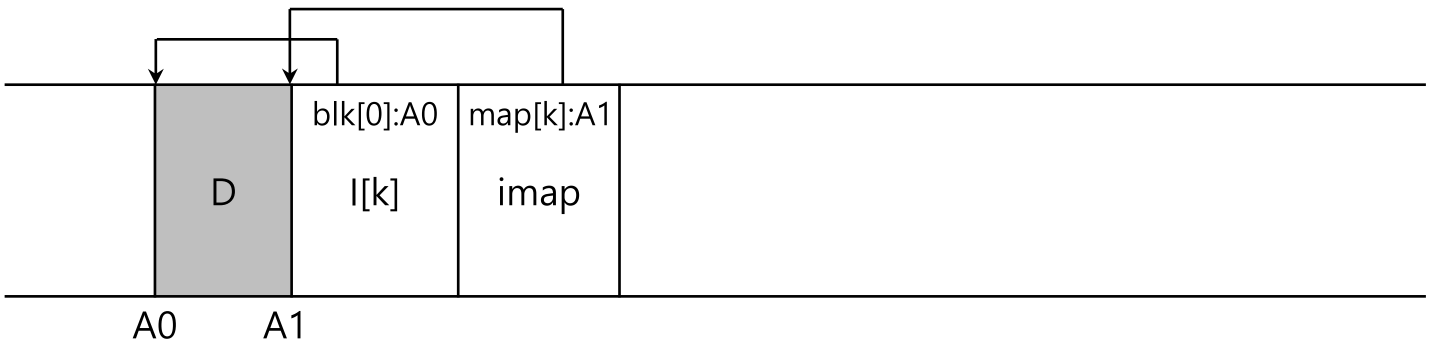
  - What is D if F = 0.95?

# Outline

- Key Idea: Writing Sequentially
- Indirect Mapping and Checkpoint Region
- Directories
- Garbage Collection
- Crash Recovery

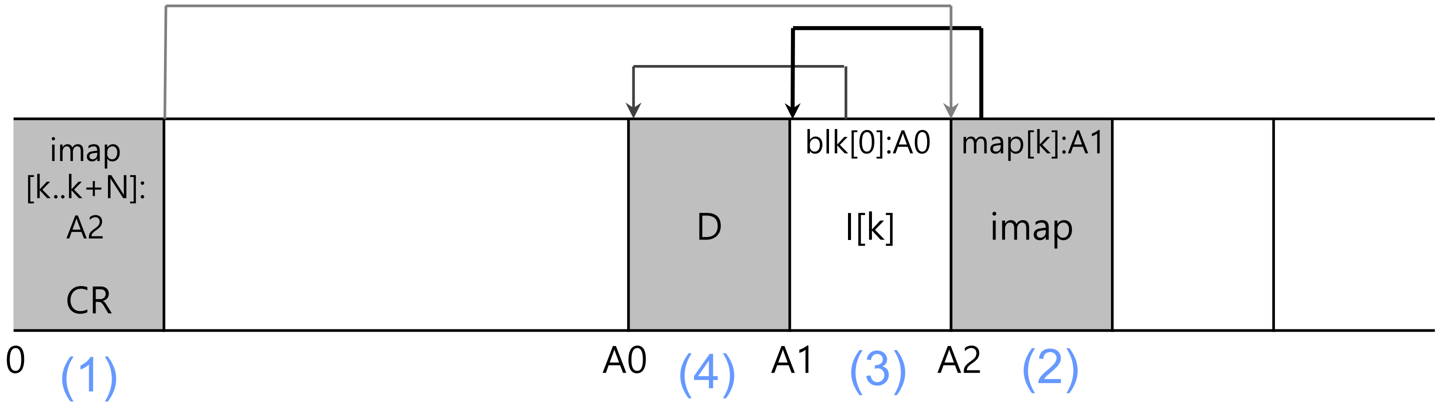
# Issue #2: How to Find Inodes?

- The position of the inodes keep changing.
- The Inode Map
  - A data structure that contains the location of the most recent inode for a given inode number.
  - Places the chunk of updated inode map right next to the updated inode (why?)
  - Where to find the inode map?



# Issue #2: How to Find Inodes?

- How to find the inode map spread across the disk?
  - The LFS must have a fixed location on disk to begin a file lookup.
- **Checkpoint Region**
  - A fixed location in the LFS partition.
  - Contain the pointers to the latest of the inode map.
- Wait, then every update still needs to seek to this CR!
  - Inode map also helps to solve the recursive update problem (coming up soon)



# Reading a file from the disk

- Reading a file block
  - Read a checkpoint region
  - Read inode map
  - Read inode
  - Read data block
- What about sequential read?
  - It may become random read.

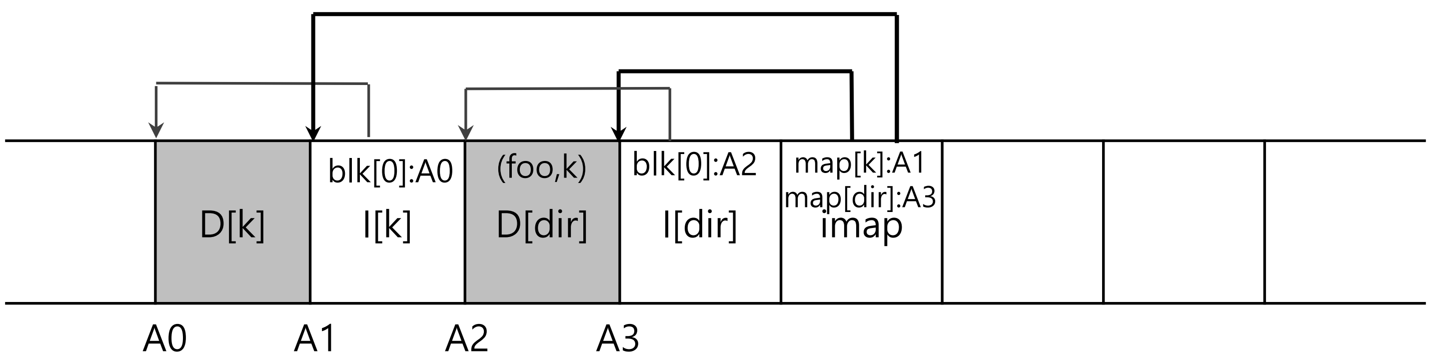
LFS is optimized for writes.

# Outline

- Key Idea: Writing Sequentially
- Indirect Mapping and Checkpoint Region
- Directories
- Garbage Collection
- Crash Recovery

# Issue #3: What About Directories?

- Directory: a set of <inode, filename>
- How does LFS store directory data?
- Creating a file: `foo`
  - Update the directory inode. (inode #: dir)
  - Update the directory entry. (`foo`, k)
  - Update inode for the created file. (inode #: k)
  - Update the data block for the created file.



# Issue #3: What About Directories?

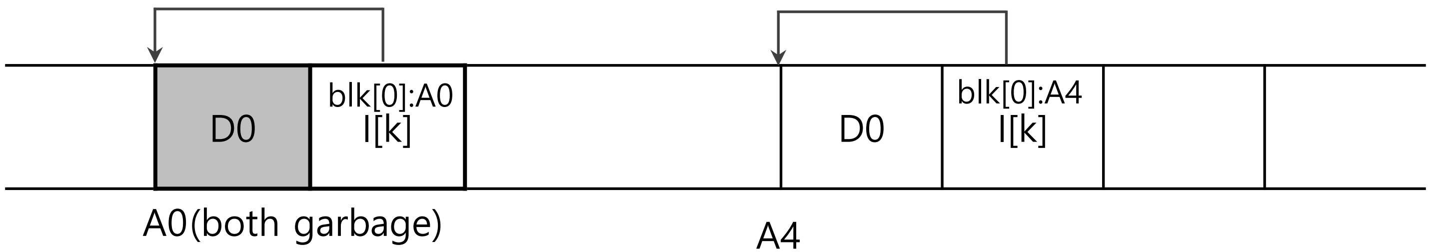
- Recursive update: A serious problem in any FS that never updates in place
  - Whenever an inode is updated, its location on disk changes
    - To keep track of inodes, a directory may have to record a collection of (name, inode-location) pairs instead
  - This would have also entailed recursive updates to the directory that points to the file, the parent of that directory, …, all the way up the file system tree
- LFS cleverly avoids this problem with imap
  - The directory is still (name, inode-num) pairs.
  - The imap keeps the inode-num to inode-location mappings
    - The change in inode location is never reflected in the directory itself

# Outline

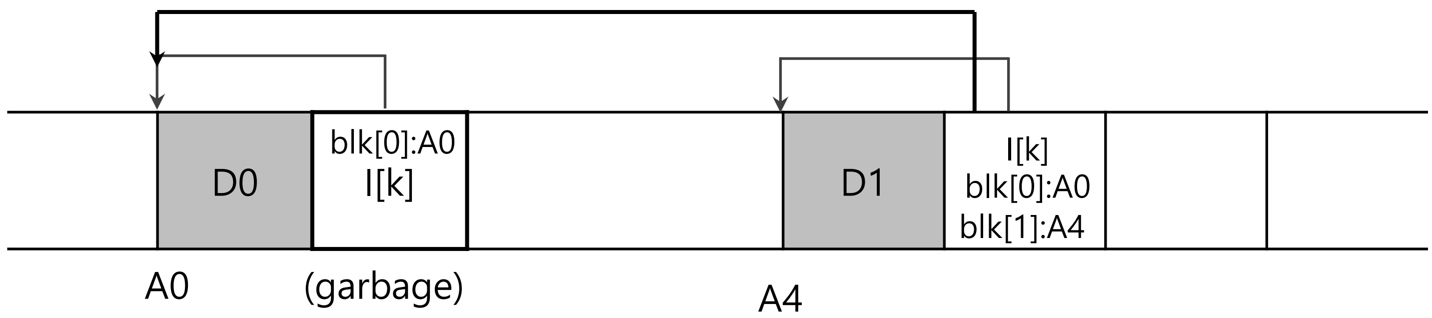
- Key Idea: Writing Sequentially
- Indirect Mapping and Checkpoint Region
- Directories
- Garbage Collection
- Crash Recovery

# Issue #4: Garbage Collection

- LFS keeps writing newer versions of a file.
- Garbage: LFS leaves the older versions of file structures all over the disk.
- An example of garbage
  - Overwrite the data block:



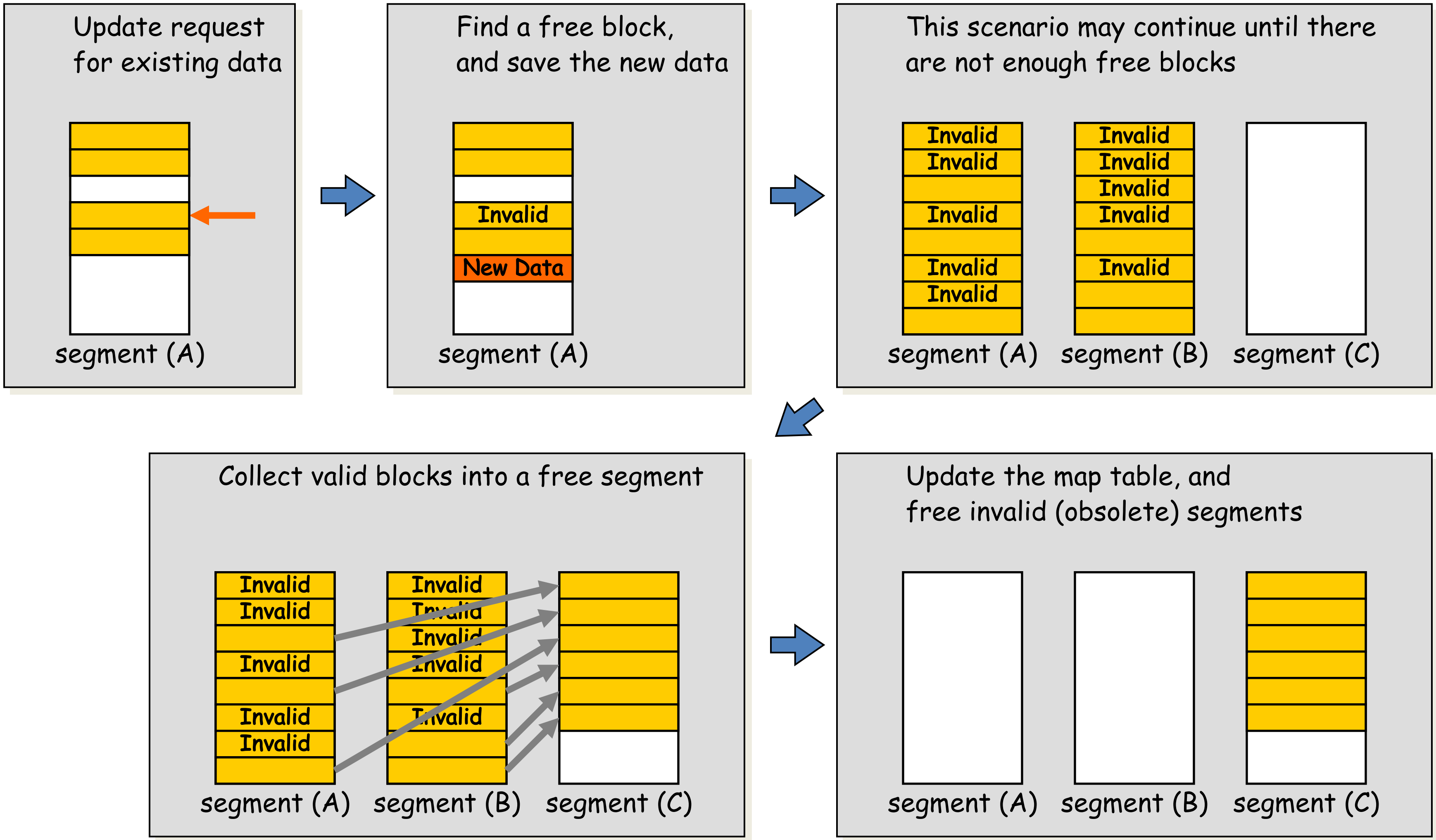

  - Append a block to that original file k:



# Issue #4: Garbage Collection

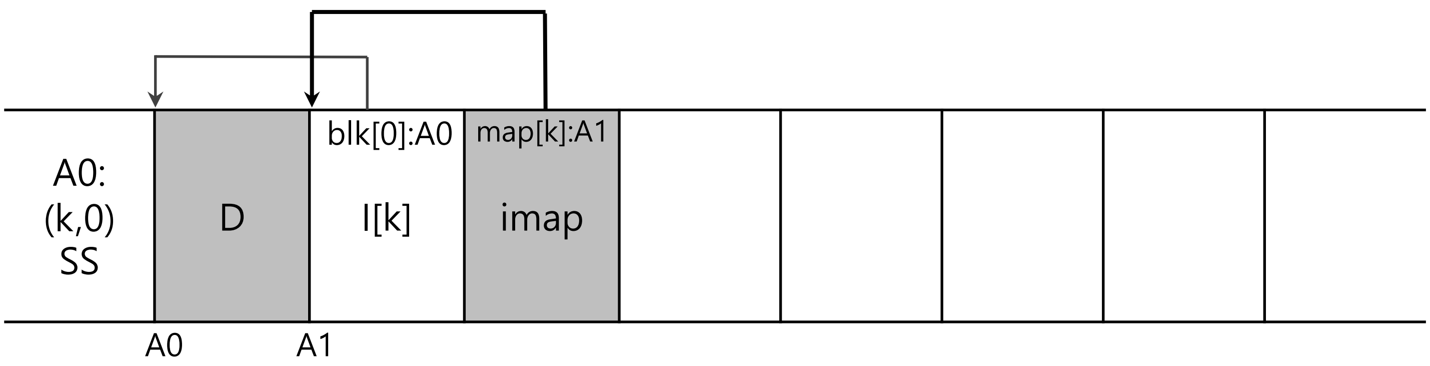
- What to do with the older versions of the block
  - Versioning filesystem: keep the old blocks and allow the users to restore to the older version of the filesystem status.
  - LFS: periodically clean the older versions of the file data, inodes and other structures.
- Unit of garbage collection: Segment
  - Read a number of old segments, M segments.
  - Identify the valid blocks.
  - Write them to a number of new segments (in memory), N segments.
  - Write N segments to the disk.
  - Then, $N < M$.

# Issue #4: Garbage Collection



# Mechanism: Segment Summary Block

- Store the inode number and the offset for each data block in it.
- In garbage collection, we need to identify the obsolete blocks.
- Compare the block address of file k, block offset 0, based upon the Segment Summary and based upon the in-memory imap. If they coincide, the block is alive



# Policy of Garbage Collection

- When to clean
  - Periodically
  - When a system is idle
  - When the disk is full
- Which block to consolidate? (heuristic from the original LFS paper)
  - Hot segment: the blocks are updated periodically
  - Cold segment: the blocks are not updated.
  - Hot segment: clean later.
  - Cold segment: clean sooner.
- Other policies are possible

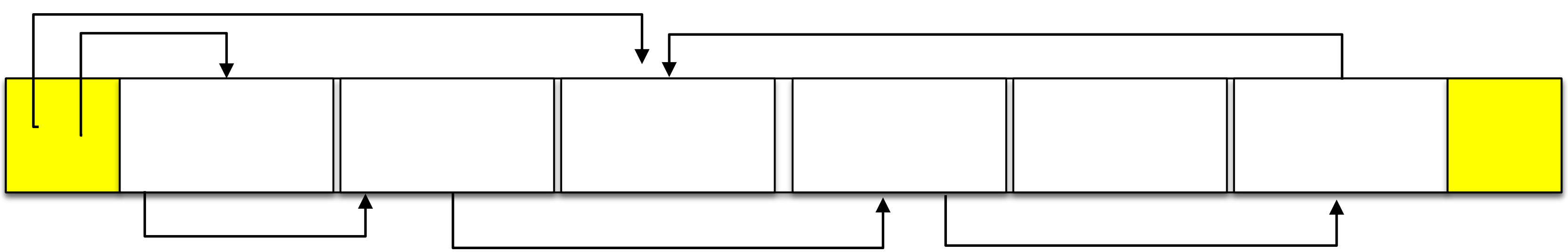
# Outline

- Key Idea: Writing Sequentially
- Indirect Mapping and Checkpoint Region
- Directories
- Garbage Collection
- Crash Recovery

# Crash recovery

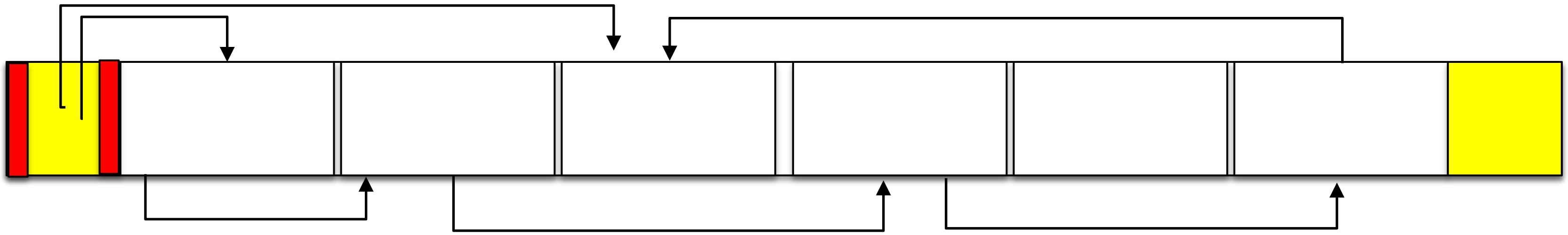
- What if the crash happens when the LFS is in the middle of writing the segment to the disk?
- LFS maintains a set of segments as a linked list in memory

- LFS organizes the filesystem partition as a log (a linked list of the segments).
  - Two checkpoint regions: one at the beginning and the one at the end.



# Crash recovery

- Consistent Update on CR
  - Write timestamp at the beginning of CR.
  - Write CR body.
  - Write time stamp at the end of the CR.
  - When crash occurs, chooses the most recent CR with valid consistent time stamps.



- Crash recovery
  - Read the CR and rebuild imap.
  - Perform roll-forward.
    - Start from the first segment in CR.
    - Scan the valid segment following the “next segment” pointer and update the imap.

# Summary

- Introduce a new approach to updating the disk.
  - **Shadow paging** in database system, **Copy-on-Write** in file system.
- Gather all updates into an in-memory segment.
  - Write them out together sequentially.
- LFS-style is excellent for performance on many different devices.
  - Hard drives, parity-based RAIDs, even Flash-based SSDs.
- Some modern commercial filesystems adopt a similar copy-on-write approach even though it generates garbage.
  - NetApp's **WAFL,** Sun's **ZFS** and Linux **btrfs**
  - In particular, WAFL turns cleaning problem into a feature, by providing old versions of the file system via **snapshots.**