PERSISTENCE: LOG-STRUCTURED FILESYSTEM

Sujay Yadalam

CS 537, Fall 2024

ADMINISTRIVIA

Project 5 due today!

Project 6 will be out!

Midterm re-grades will be done soon

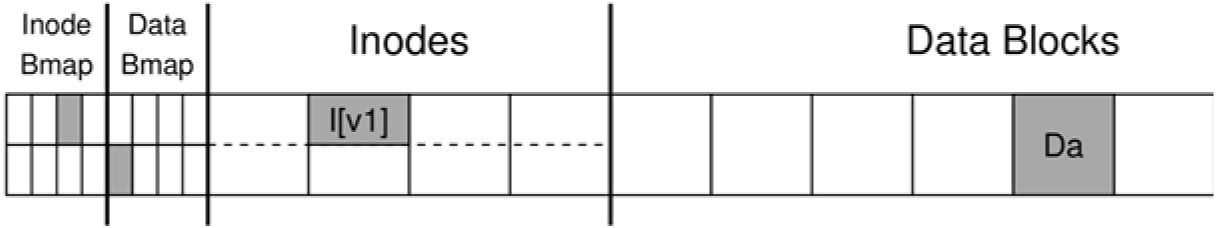
AGENDA / LEARNING OUTCOMES

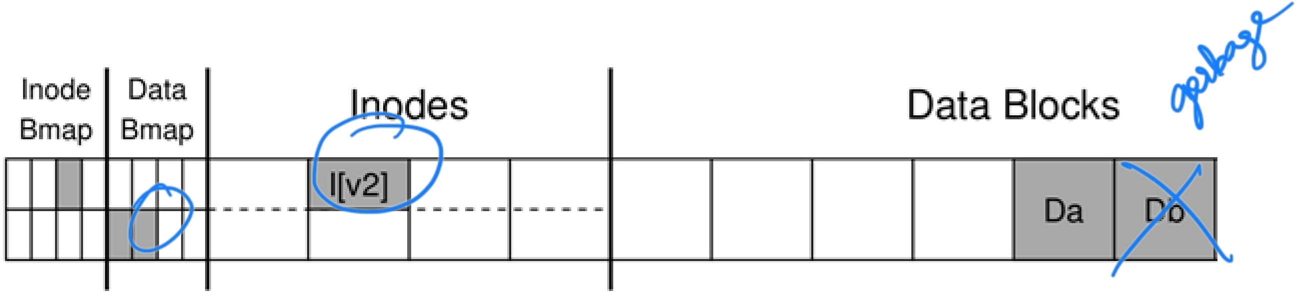
How to optimize a filesystem that performs better for writes?

What are some challenges and how to overcome them?

RECAP

FILE APPEND EXAMPLE



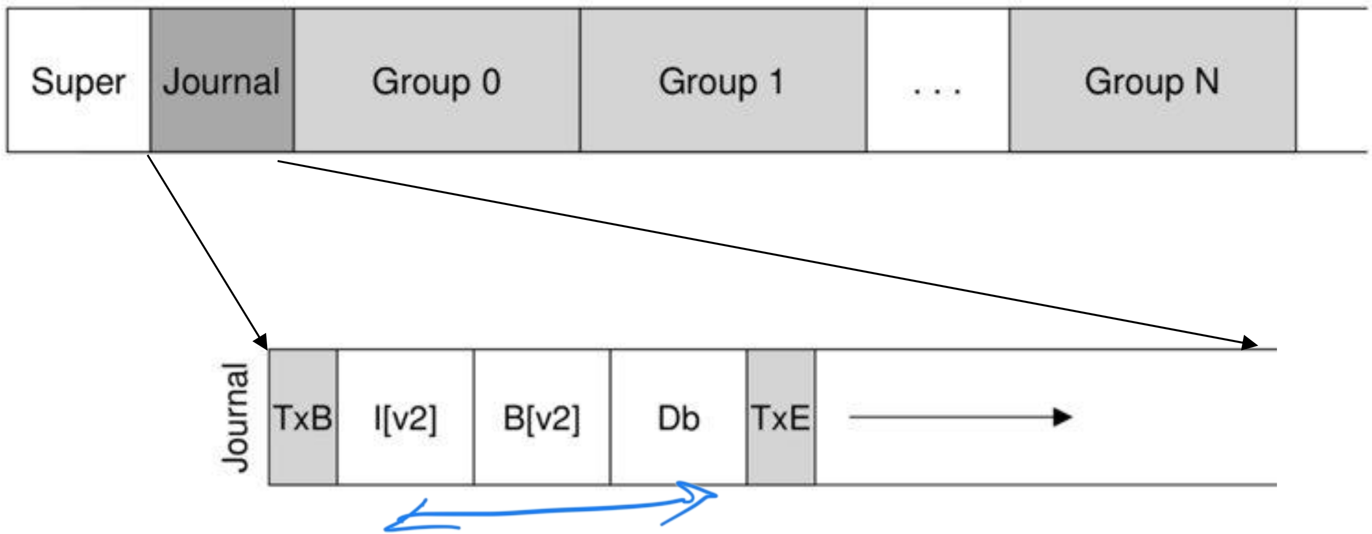


HOW TO FIX INCONSISTENCIES?

1) FSCK = file system checker
   - Read entire disk and fix issues: too slow
   - Do not know "correct" state; knows a consistent state

2) Journaling
   - Write to a special journal before writing in-place

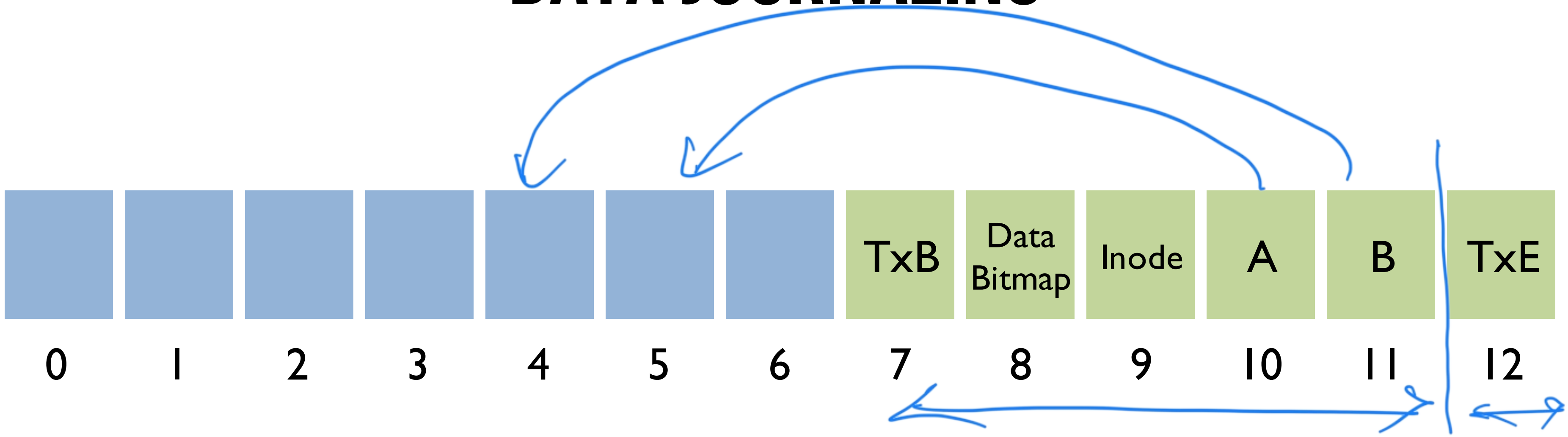
JOURNAL LAYOUT



IN-CLASS QUIZ

https://tinyurl.com/cs537-fa24-q18

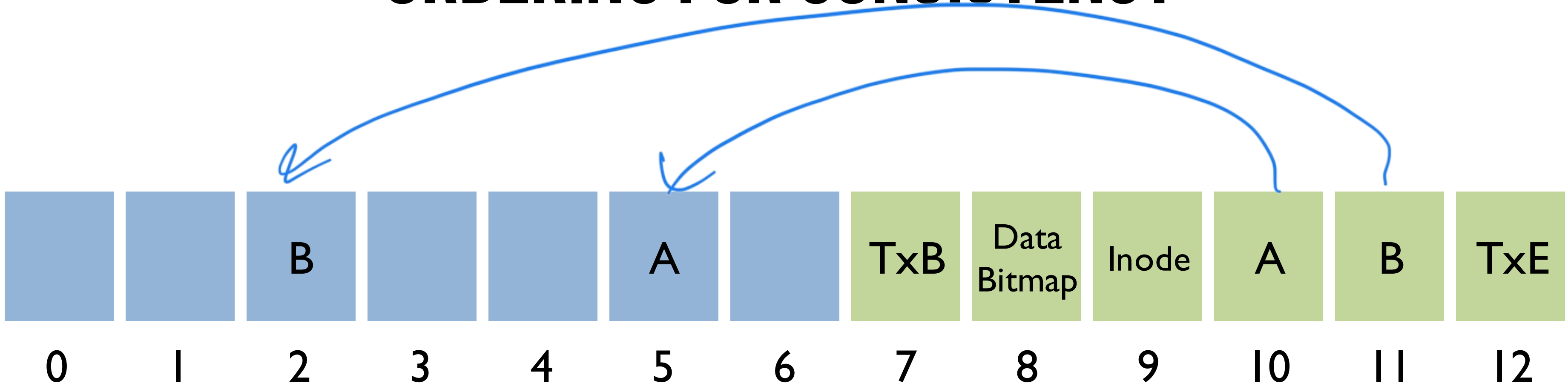
DATA JOURNALING



Checkpoint: Writing new data to in-place locations

After checkpointing, journal re-used for next transaction

ORDERING FOR CONSISTENCY



write order

7,8,9,10,11

TxE 12

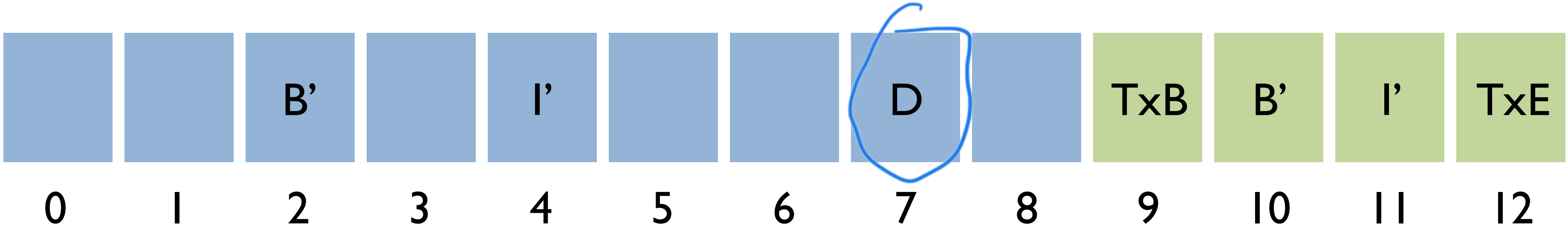
4,6

Barriers

1) Before journal commit, ensure journal entries complete
2) Before checkpoint, ensure journal commit complete
3) Before free journal, ensure in-place updates complete

METADATA JOURNALING

transaction: append to inode I



Idea: avoid data journaling; only journal metadata; But write data **before** the transaction!

LOG STRUCTURED FILE SYSTEM (LFS)

LFS PERFORMANCE GOAL

Motivation:

- Single operation (create a new file) requires multiple random writes
- RAID-4 and RAID-5 random write performance is poor

- Large gap between sequential and random I/O performance

LFS PERFORMANCE GOAL

Motivation:

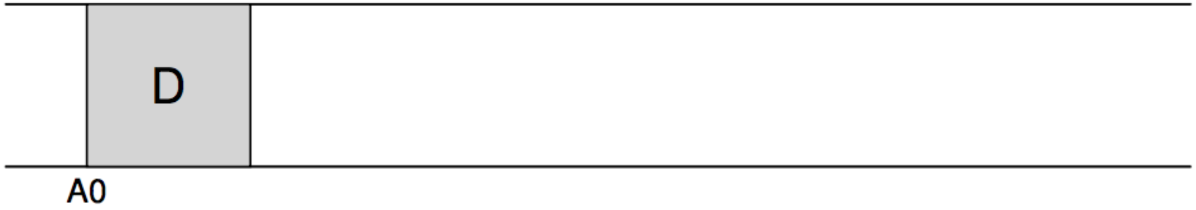
- Single operation (create a new file) requires multiple random writes
- RAID-4 and RAID-5 random write performance is poor

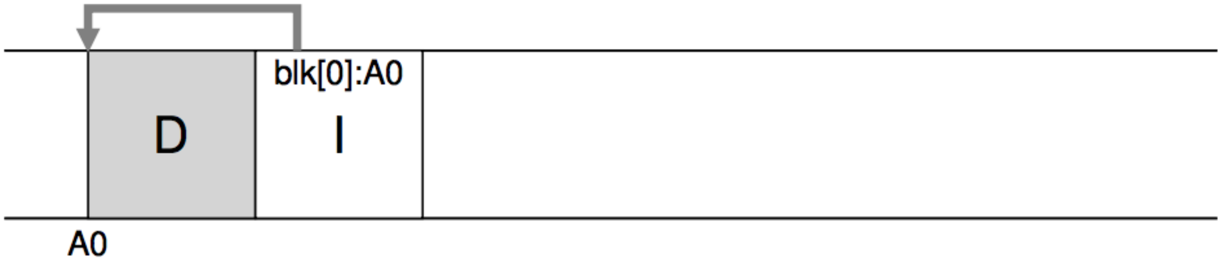
- Large gap between sequential and random I/O performance

Idea: use disk purely sequentially

No random writes!

WHERE DO INODES GO?





IS WRITING SEQUENTIALLY SUFFICIENT?

IS WRITING SEQUENTIALLY SUFFICIENT?

No!

Example:

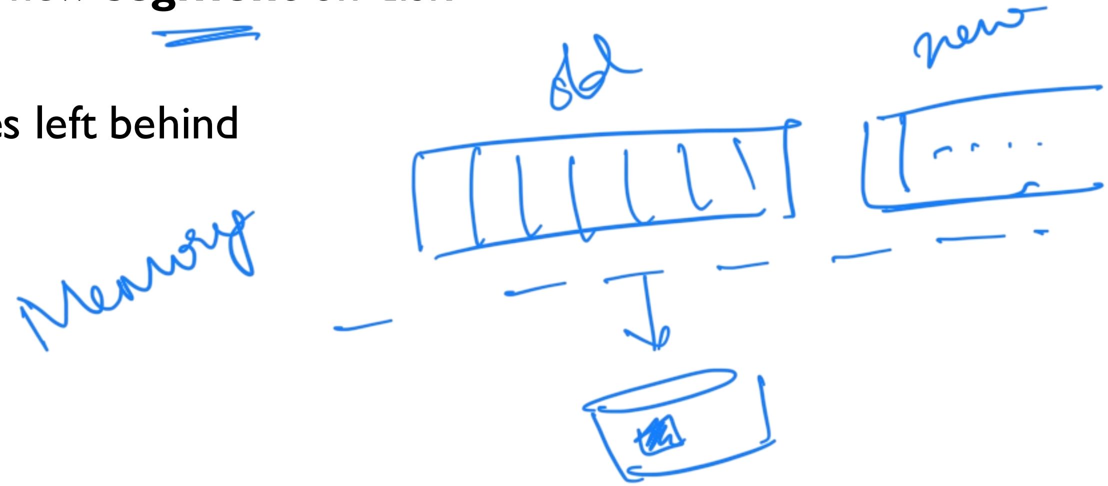
- Write block
- Perform computation
- Write block (but disk has already rotated past the desired block)

LFS STRATEGY

File system buffers writes in main memory until “enough” data

Write buffered data sequentially to new **segment** on disk

Never overwrite old info: old copies left behind



LFS STRATEGY

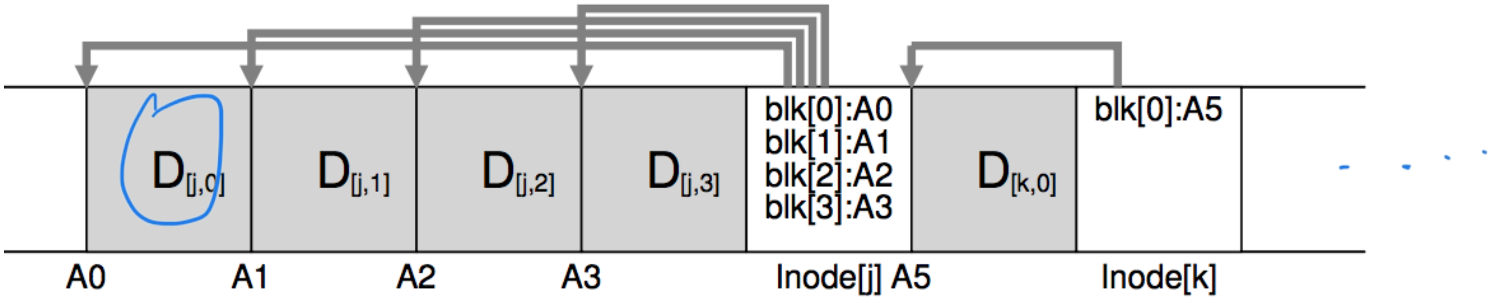
File system buffers writes in main memory until "enough" data

Write buffered data sequentially to new **segment** on disk

Never overwrite old info: old copies left behind

- How much to buffer?
- Enough to get good sequential bandwidth from disk (MB)

BUFFERED WRITES



WHAT IS DIFFERENT FROM FFS?

1) What data structures has LFS removed?

WHAT IS DIFFERENT FROM FFS?

1) What data structures has LFS removed?

   allocation structs: data + inode bitmaps

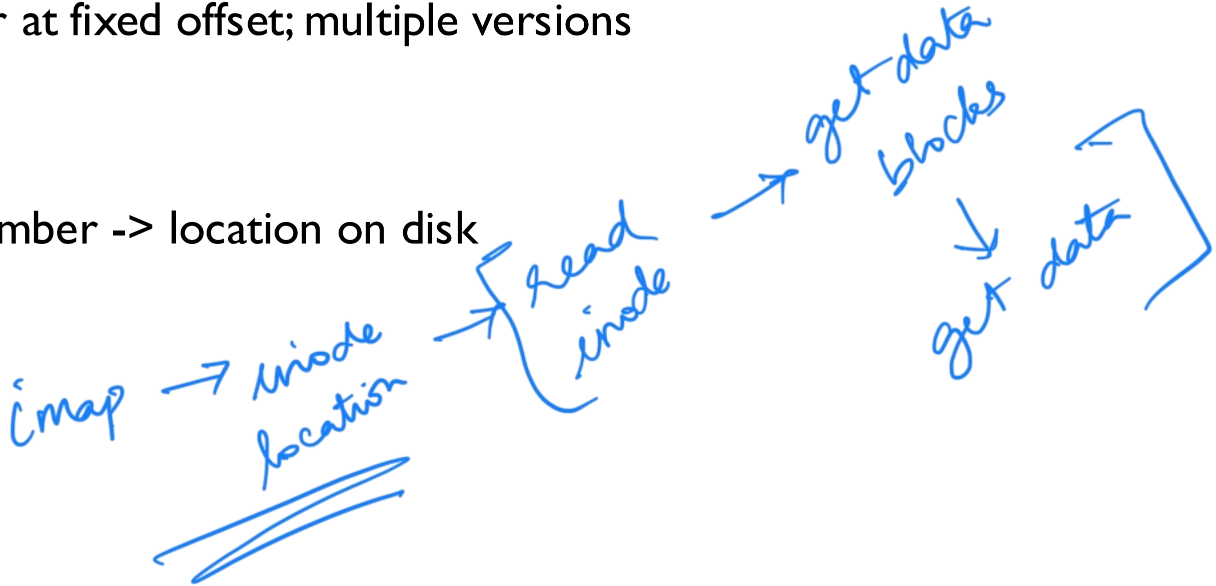
last_block_written_addr

CHALLENGE 1: HOW TO LOCATE LATEST INODES?

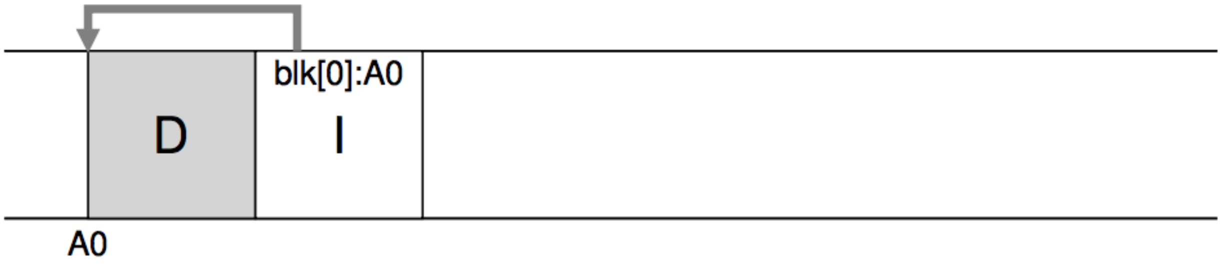
Problem: Inodes are no longer at fixed offset; multiple versions

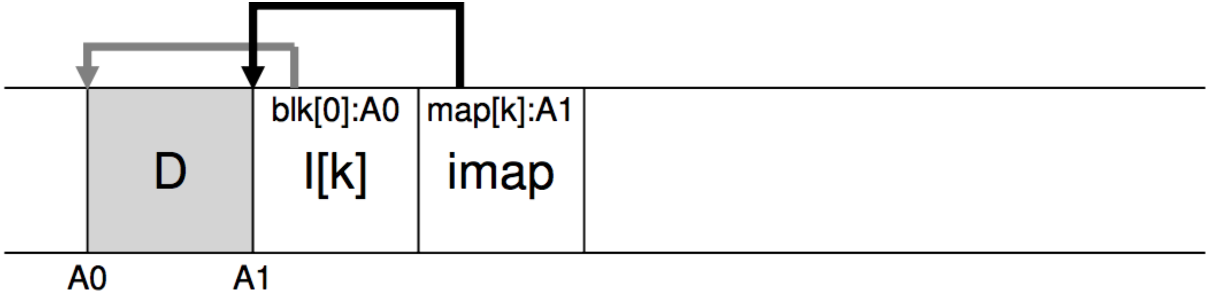
Solution: Use imap structure

imap = maps inode number -> location on disk

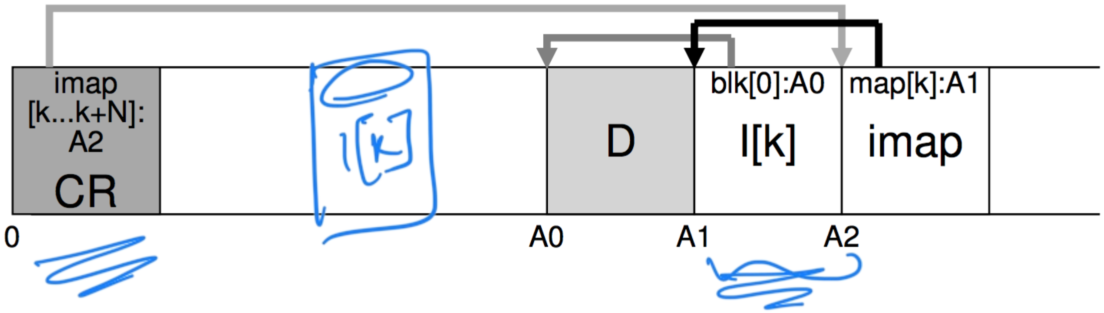


IMAP EXPLAINED





READING IN LFS



1. Read the Checkpoint region
2. Read all imap parts, cache in mem
3. To read a file:
    1. Lookup inode location in imap
    2. Read inode
    3. Read the file block

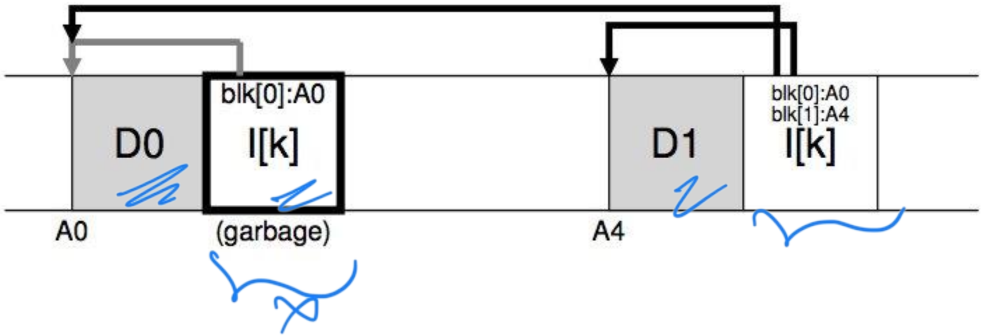
CHALLENGE 2: WHAT TO DO WITH OLD DATA?

Old versions of files → garbage

Approach 1: garbage is a feature!

- Keep old versions in case user wants to revert files later
- Versioning file systems
- Example: Dropbox

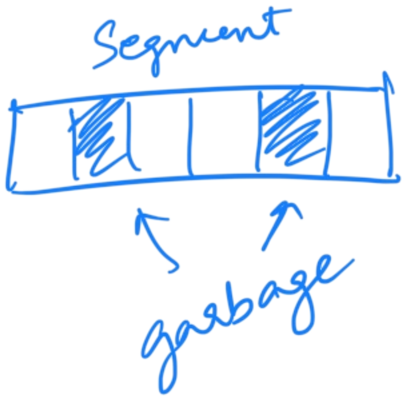
Approach 2: garbage collection



GARBAGE COLLECTION

Need to reclaim space:

1. When no more references (any file system)
2. After newer copy is created (COW file system)



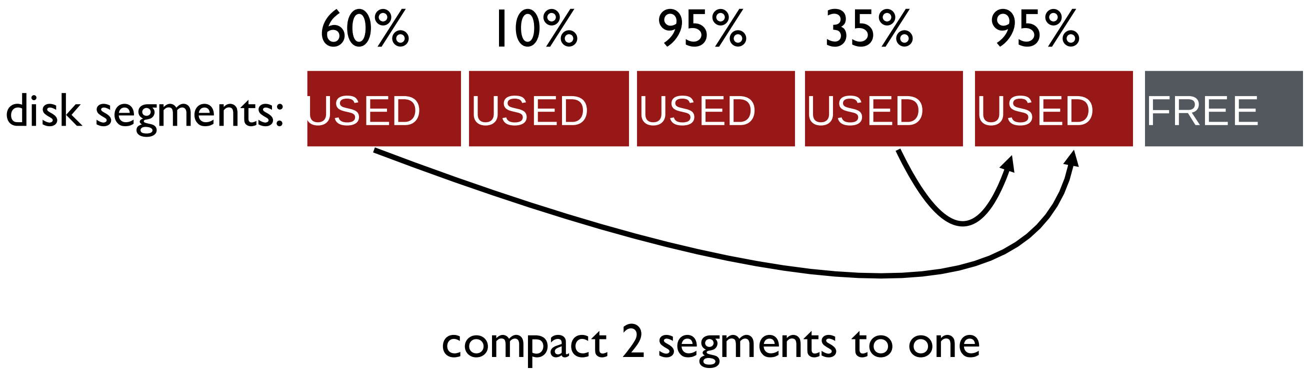
LFS reclaims **segments** (not individual inodes and data blocks)

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

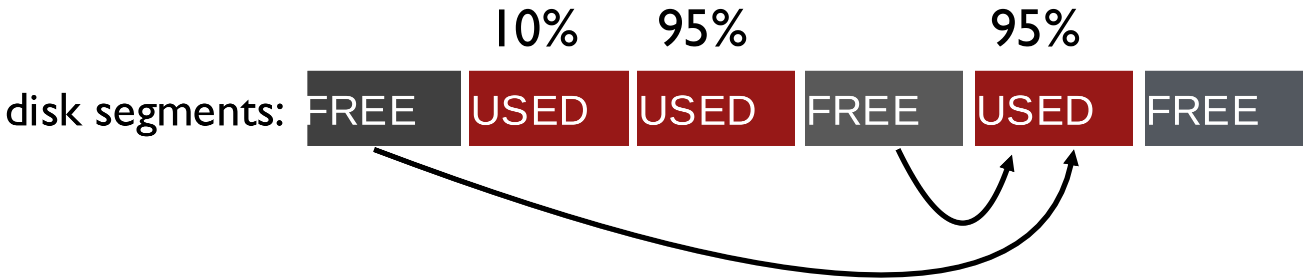
GARBAGE COLLECTION

	60%	10%	95%	35%		
disk segments:	USED	USED	USED	USED	FREE	FREE

GARBAGE COLLECTION



GARBAGE COLLECTION



compact 2 segments to one

1> Identify garbage blocks
2> Compact segments

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

GARBAGE COLLECTION

General operation:

Pick M segments, compact into N (where N < M).

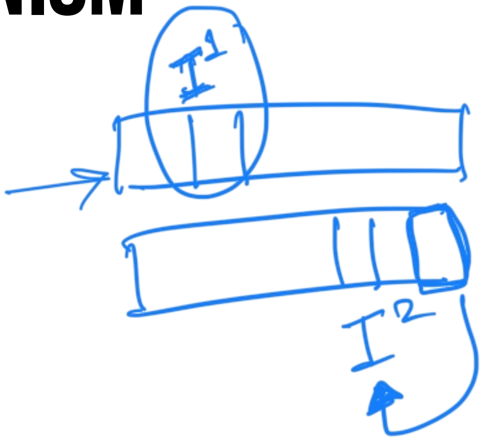
Mechanism:

How does LFS know whether data in segments is valid?

Policy:

Which segments to compact?

Garbage Collection Mechanism



Is an inode the latest version?

- Check imap to see if this inode is pointed to
- Fast!

Is a data block the latest version?

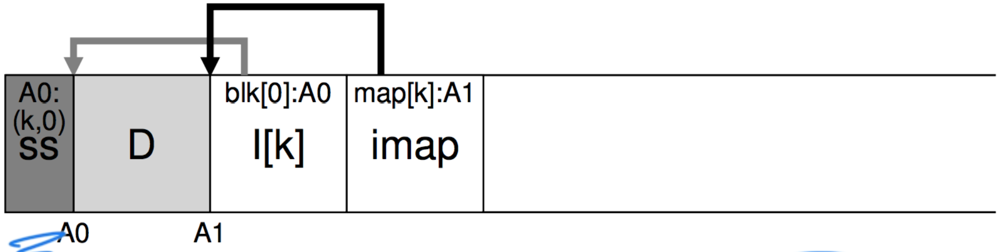
- Scan ALL inodes to see if any point to this data
- Very slow!

How to track information more efficiently?

- **Segment summary** lists inode and data offset corresponding to each data block in segment (reverse pointers)

SEGMENT SUMMARY

= data → inode mapping




```
(N, T) = SegmentSummary[A];

inode = Read(imap[N]);

if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Block 10000 belongs to inode 70 and is mapped at offset 1024

* SS helps narrow down the inode to check

GARBAGE COLLECTION

General operation:

Pick M segments, compact into N (where N < M).

Mechanism:

Use segment summary, imap to determine liveness

Policy:

Which segments to compact?

- clean most empty first
- clean coldest (ones undergoing least change)
- more complex heuristics…

CHALLENGE 3: CRASH RECOVERY

What data needs to be recovered after a crash?

- Need imap (lost in volatile memory)

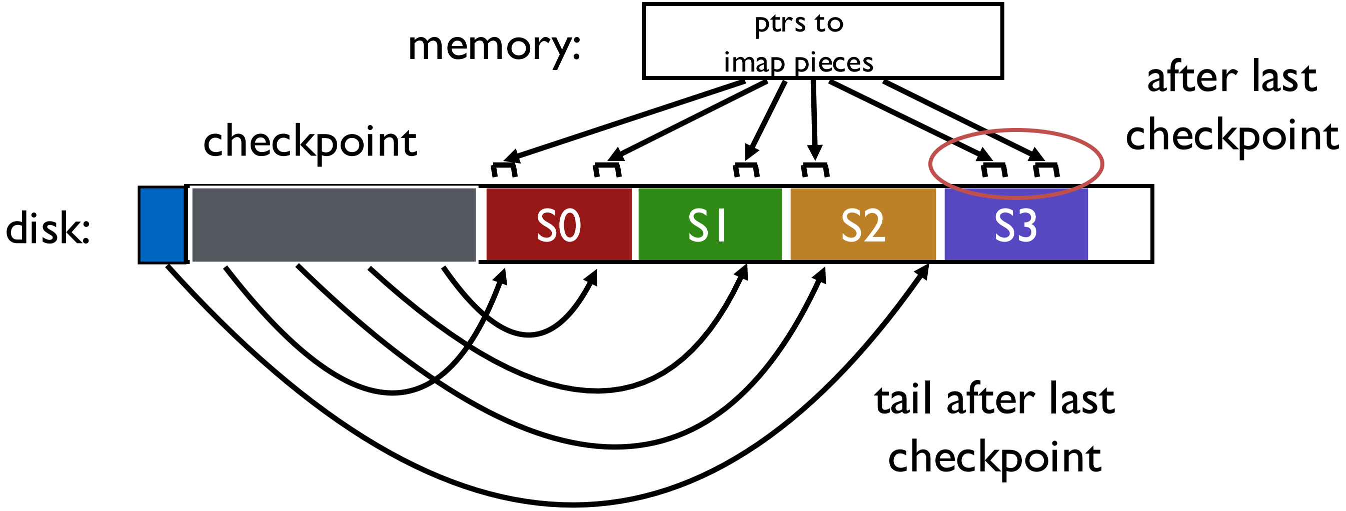
Better approach?

- Occasionally save to checkpoint region the pointers to imap pieces

How often to checkpoint?

- Checkpoint often: poor performance (random I/O)
- Checkpoint rarely: lose more data, recovery takes longer
- Example: checkpoint every 30 secs

CRASH RECOVERY



CHECKPOINT SUMMARY

Checkpoint occasionally (e.g., every 30s)

Upon recovery:

- read checkpoint to find most imap pointers and segment tail
- find rest of imap pointers by reading past tail

What if crash <u>during</u> checkpoint?

CHECKPOINT STRATEGY

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

LFS SUMMARY

Journaling:

Put final location of data wherever file system chooses

(usually in a place optimized for future reads)

LFS:

Puts data where it's fastest to write, assume future reads cached in memory

Other COW file systems: WAFL, ZFS, btrfs

NEXT STEPS

Next class: SSDs!