Hello!

PERSISTENCE: LOG-STRUCTURED FILESYSTEM

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

Project 5, 6 grading → Project 5 – day or two

Project 7 out!

Project 8 update! → Extra credit 4%

Midterm 3 conflicts 5/8 7:25 – 9:25 pm

AGENDA / LEARNING OUTCOMES

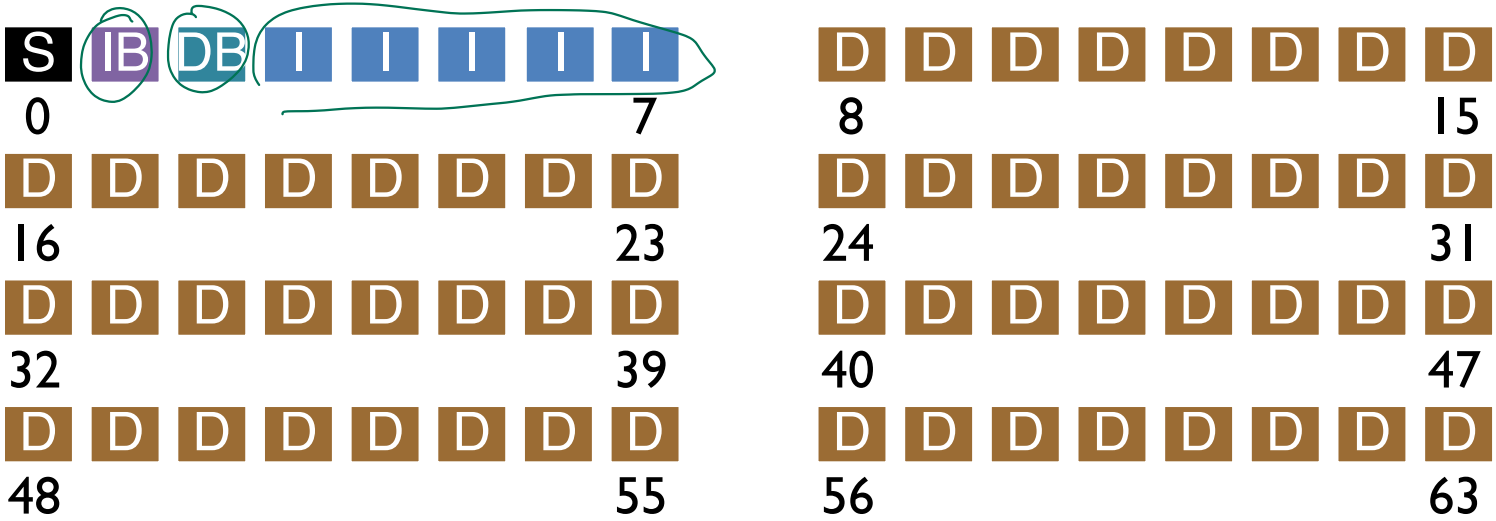
How to design a filesystem that performs better for small writes?

What are some similarities or differences with FFS?

RECAP

FS STRUCTS

Simple FS layout
→ FFS layout



HOW CAN FILE SYSTEM FIX INCONSISTENCIES?

Solution #1:

FSCK = file system checker

Strategy:

After crash, scan whole disk for contradictions and "fix" if needed

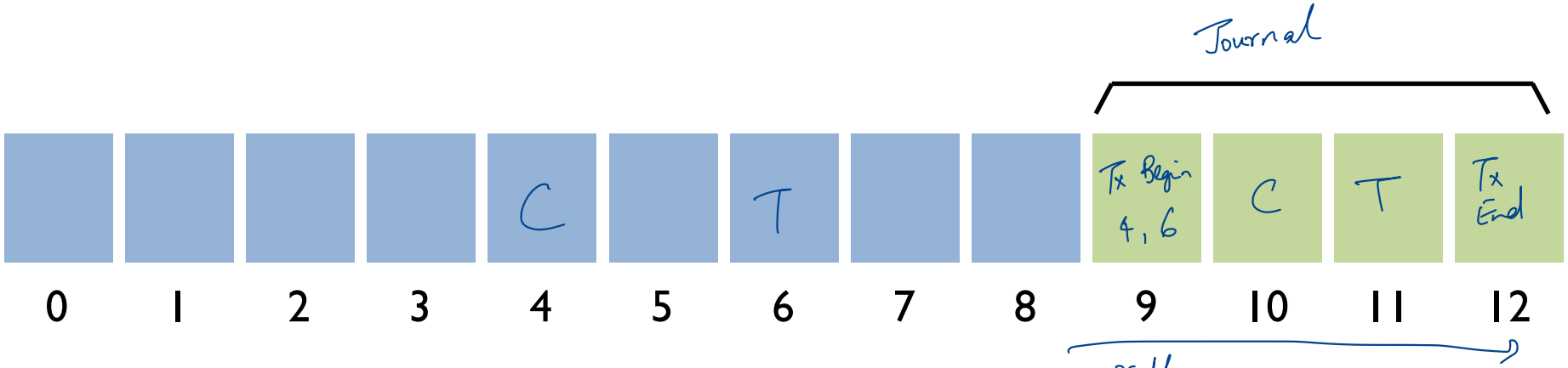
Keep file system off-line until FSCK completes

For example, how to tell if data bitmap block is consistent?

Read every valid inode+indirect block

If pointer to data block, the corresponding bit should be 1; else bit is 0

ORDERING FOR CONSISTENCY



Transaction: write C to block 4; write T to block 6

write order

9,10,11

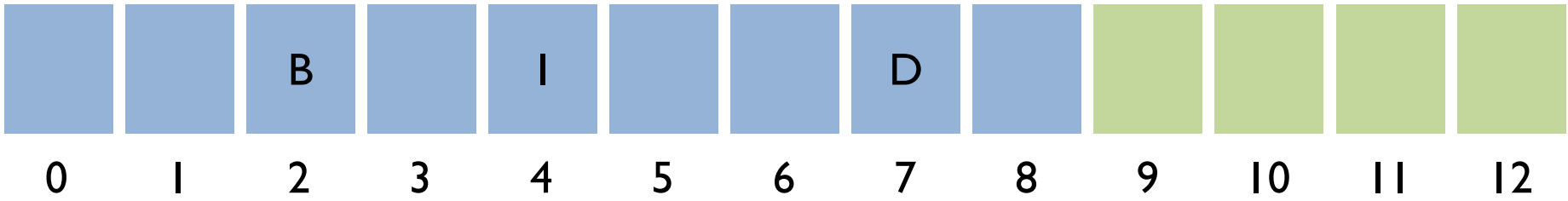
12

4,6

flush journal to disk

flush Tx End

ORDERED JOURNAL



Append to a file
Data (D) in block 7
Inode (I) in block 4
Bitmap (B) in block 2

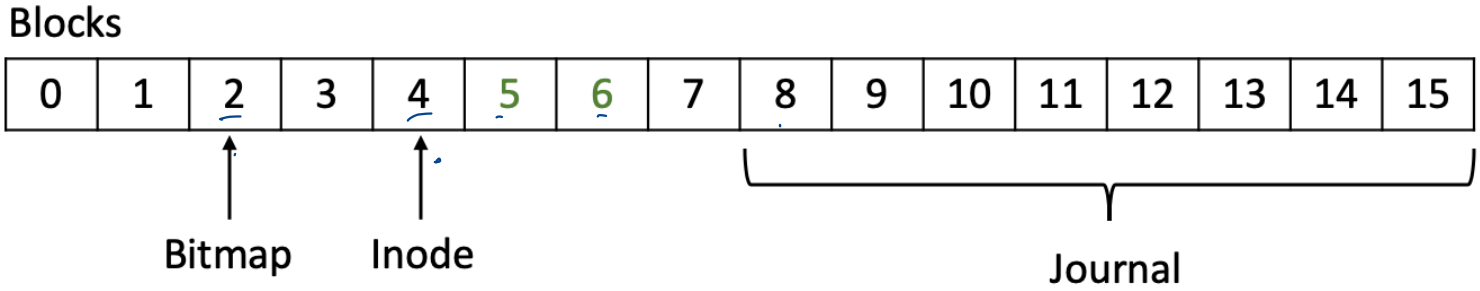
only journal metadata entries written to disk

data blocks directly

→ bitmaps
inode
directories

QUIZ 29

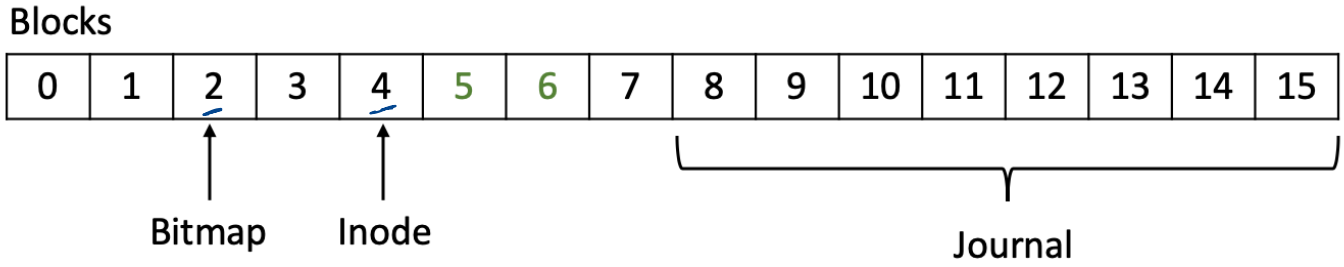
https://tinyurl.com/cs537-sp23-quiz29



Write 5,6
Write 8, 9, 10
Barrier
Write 11
Barrier
Write 4, 2

→ writing data blocks first

In ordered journal → only journal metadata



Write 8, 9, 10, 11, 12
Barrier
Write 13
Barrier
Write 2, 4, 5, 6

Journaling everything

If txn will get replayed

→ Data journaling

Write 8, 9, 10, 11, 12, 13
Barrier
Write 2, 4, 5, 6

include "hash" of 8-12

checksum optimization!

LOG STRUCTURED FILE SYSTEM (LFS)

LFS PERFORMANCE GOAL

Motivation:

- Growing gap between sequential and random I/O performance
- Especially true in SSDs!
- RAID-5 especially bad with small random writes

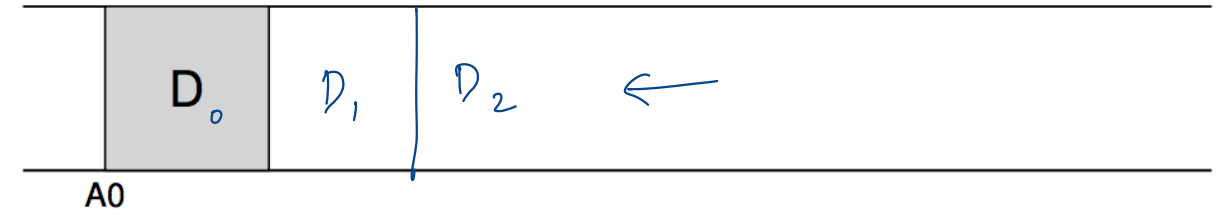
Idea: use disk purely sequentially

Design for writes to use disk sequentially – how?

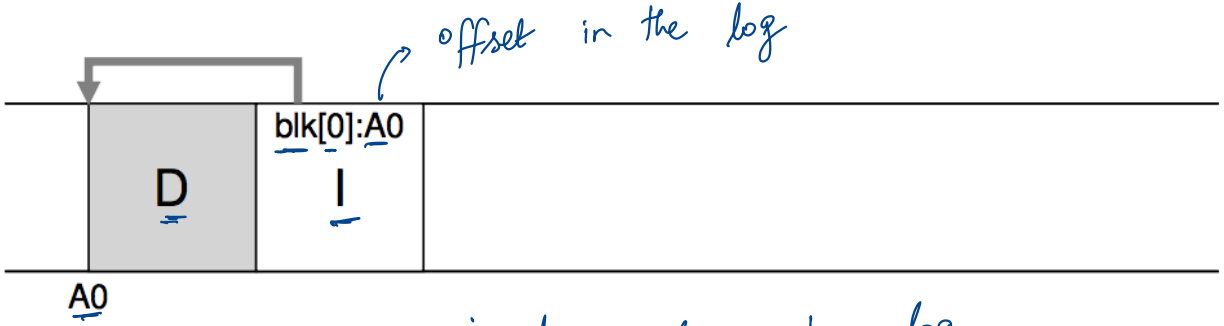
late 80s

Write slow in FFS design → large number of random writes

WHERE DO INODES GO?



"tail ptr" and I append data blocks



- Write inode also to log
  - First data block, add ptr from inode

LFS STRATEGY

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

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

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

Never overwrite old info: old copies left behind

4 MB

Large Sequential writes

segment: sequence of writes that are flushed to disk as one unit

↳ append data to file needs new version of inode

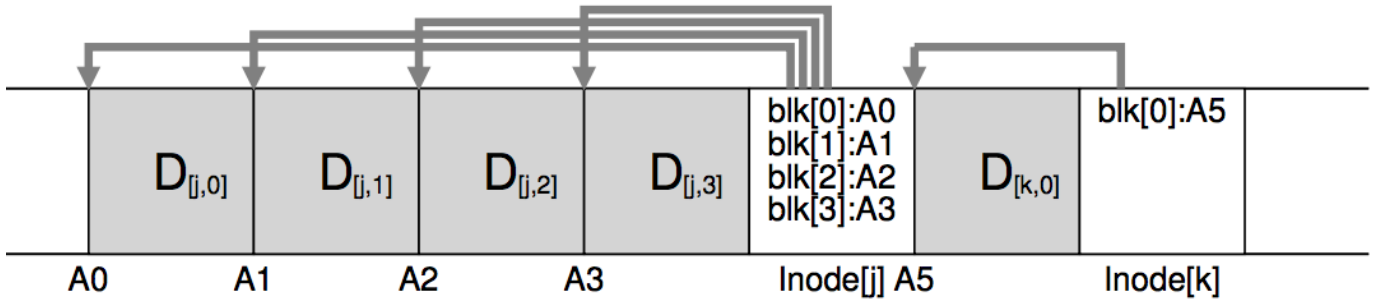
↳ How do we track last segment written?

BUFFERED WRITES

4 blocks to file 1

1 block to file

all of them are in this segment



How do we know where is latest inode?

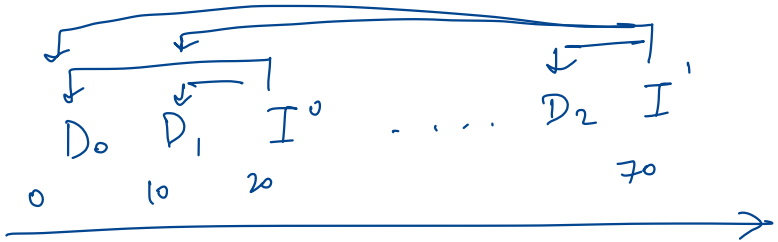
new inode version

old and new ptrs

WHAT ELSE IS DIFFERENT FROM FFS?

What data structures has LFS removed?

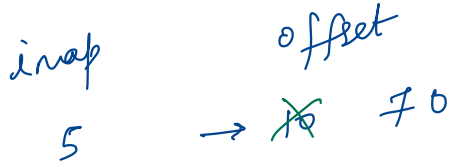
allocation structs: data + inode bitmaps



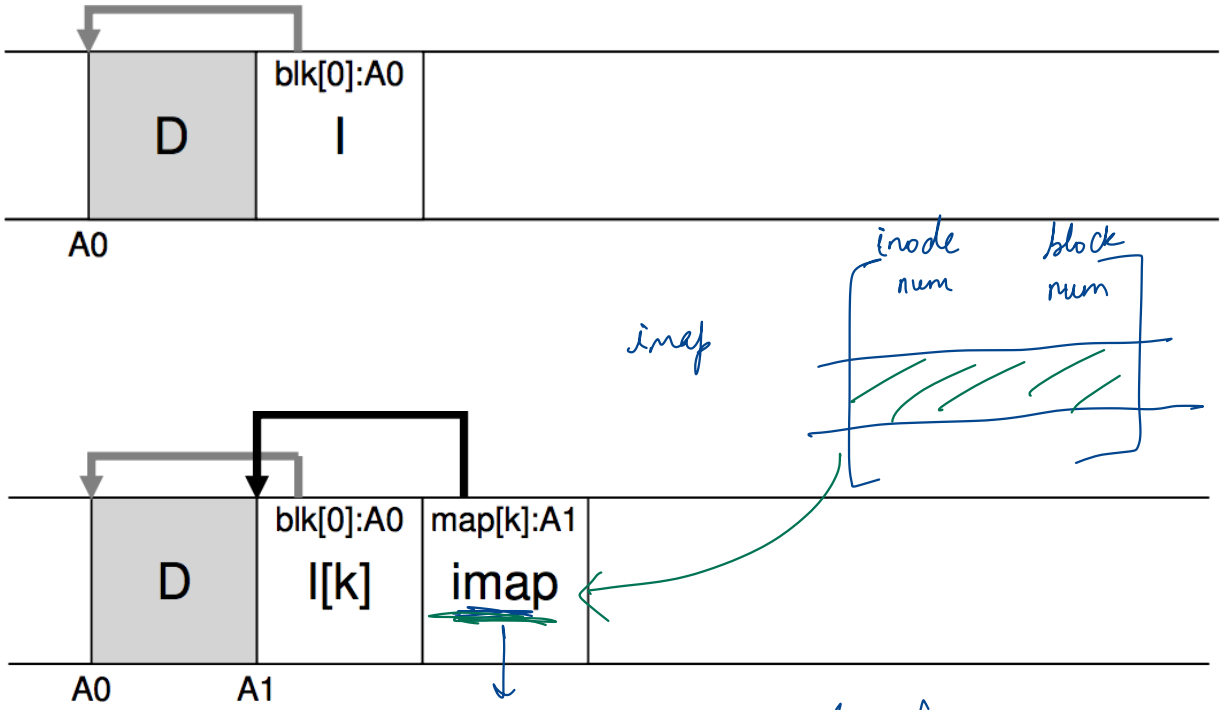
How to do reads?

Inodes are no longer at fixed offset

Use imap structure to map:
inode number => inode location on disk



IMAP EXPLAINED

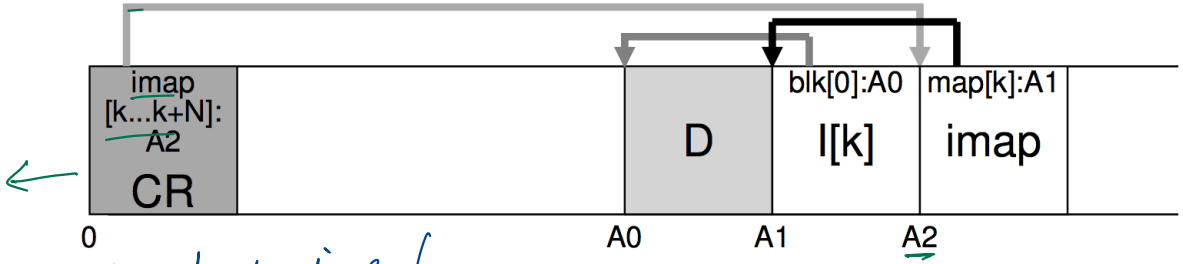


imap
- cached in memory
- Construct imap on boot
- imap updates are written to the log

READING IN LFS

Similar

"Superblock"



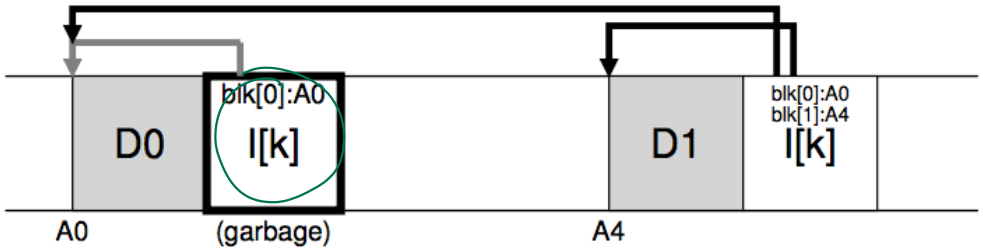
only once at boot time / mount time

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

for every file read

Boot up machine - nothing is in memory

GARBAGE COLLECTION



Append D1 to file

↳ Created new inode version

↳ old inode is garbage

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" → every file has version number
- Example: Dropbox

Approach 2: garbage collection

↳ remove garbage blocks from the log

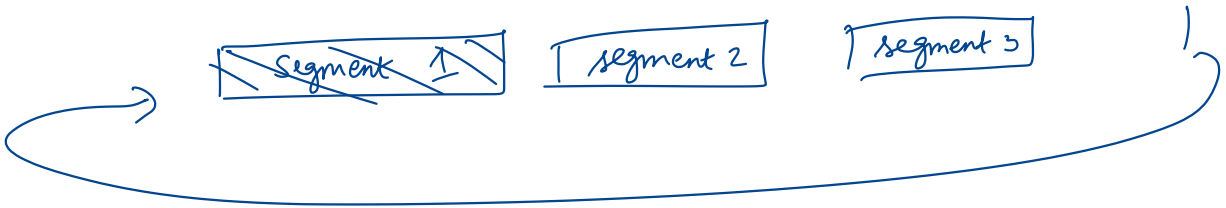
GARBAGE COLLECTION

Need to reclaim space:

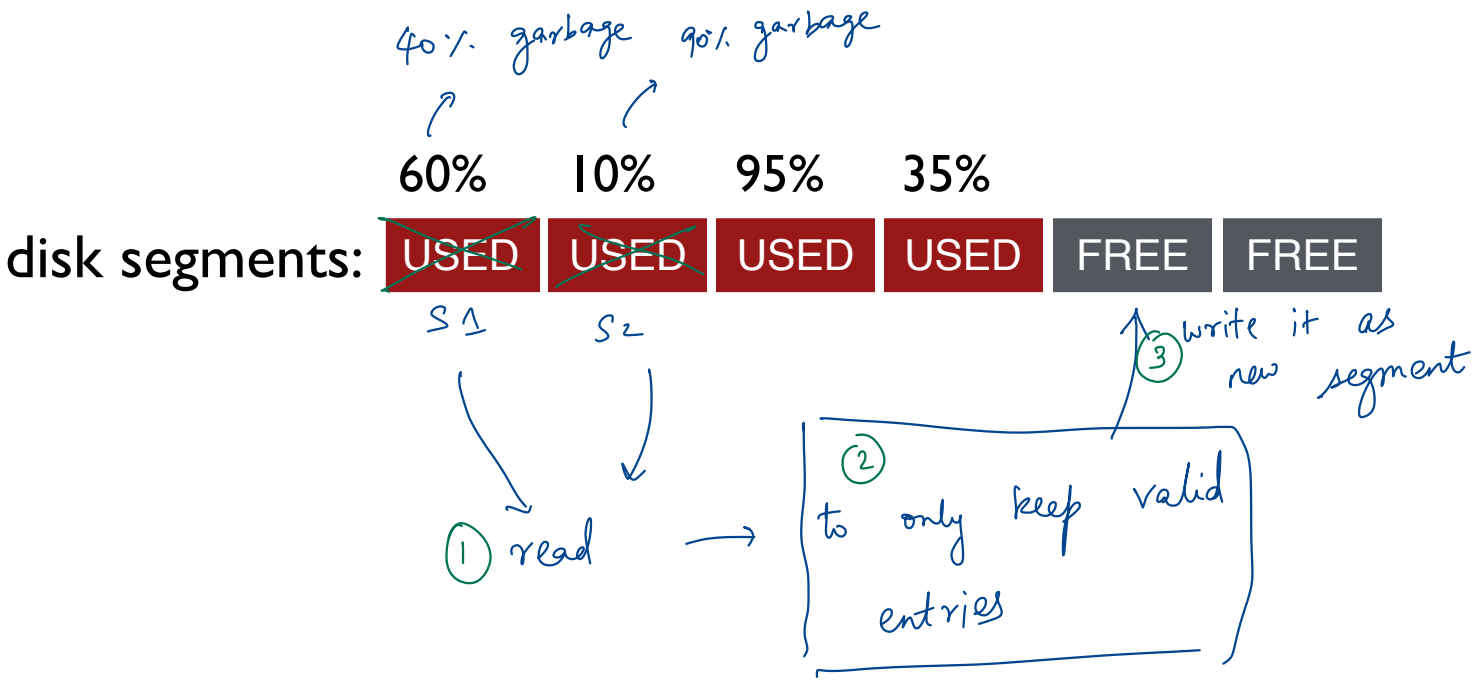
1. When no more references (any file system)
2. After newer copy is created (COW file system)

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

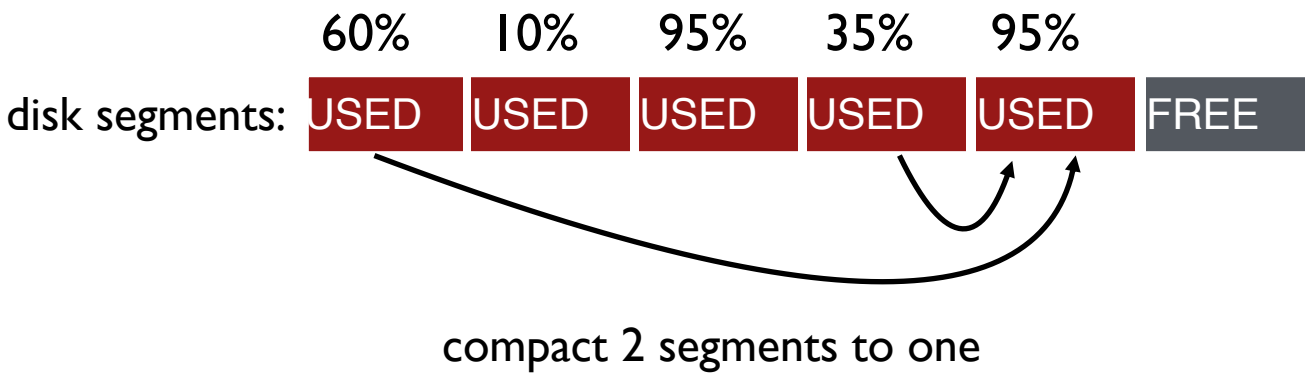
- Want future overwites to be to sequential areas
- Tricky, since segments are usually partly valid



GARBAGE COLLECTION



GARBAGE COLLECTION



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

similar to appending new data

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?

↳ Segments which are oldest

Segments with most garbage

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!

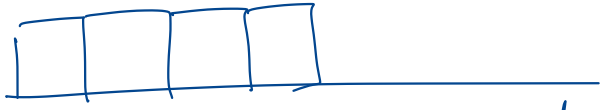
How to track information more efficiently?

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

inode : ptrs to data blocks

seg summary : ptrs from data blocks to inodes

Segment :



for each block is this valid or not.

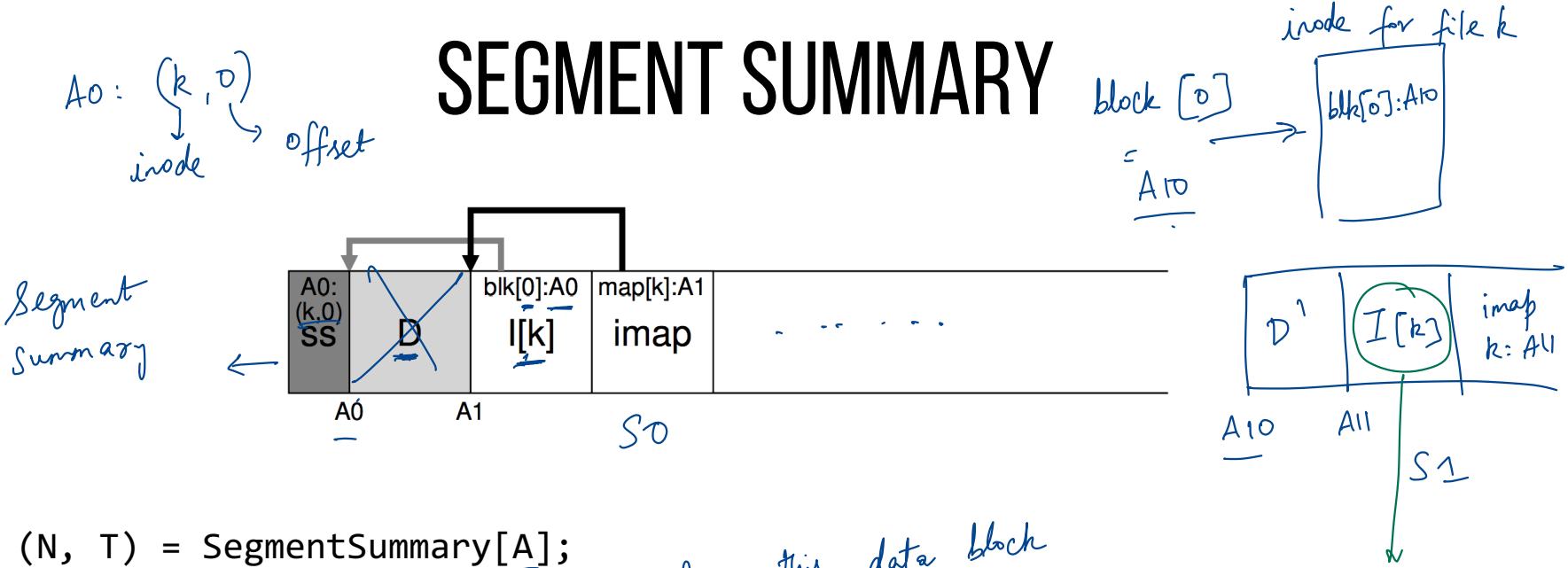
Data blocks

- Is valid if some inode points to it
- Slow
  → requires scanning all the inodes

Inode blocks

- Check imap points to this inode
- Fast, imap is in memory

SEGMENT SUMMARY




```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

for this data block

read imap for inode num

offset T inside latest inode

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…

CRASH RECOVERY

What data needs to be recovered after a crash?

- Need imap (lost in volatile memory)

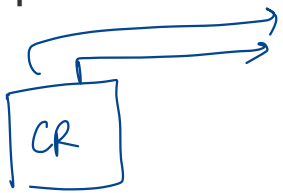
imap persist & recover

Better approach?

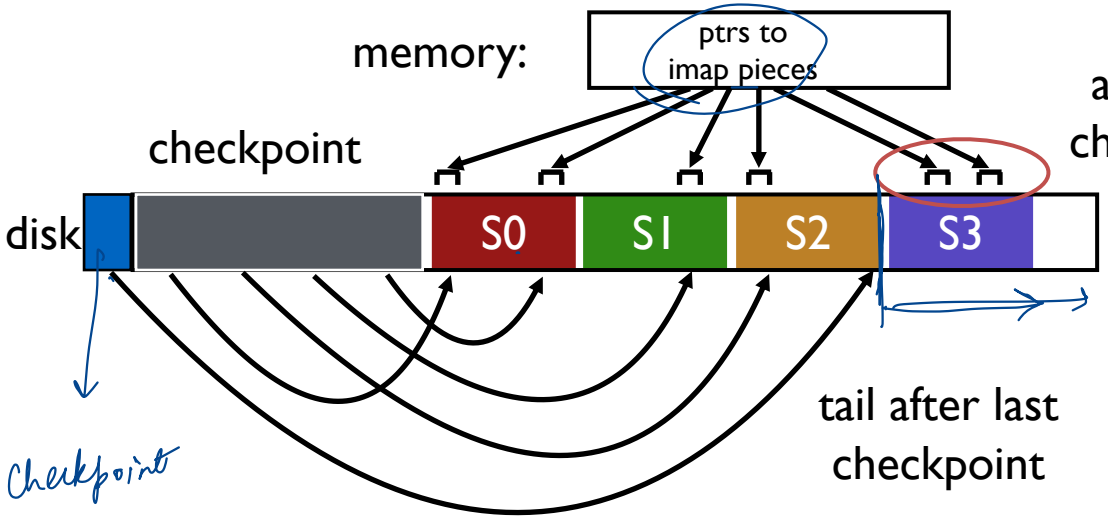
- Occasionally save to checkpoint region the pointers to imap pieces

How often to checkpoint?

- Checkpoint often: random I/O → negate benefits
- 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

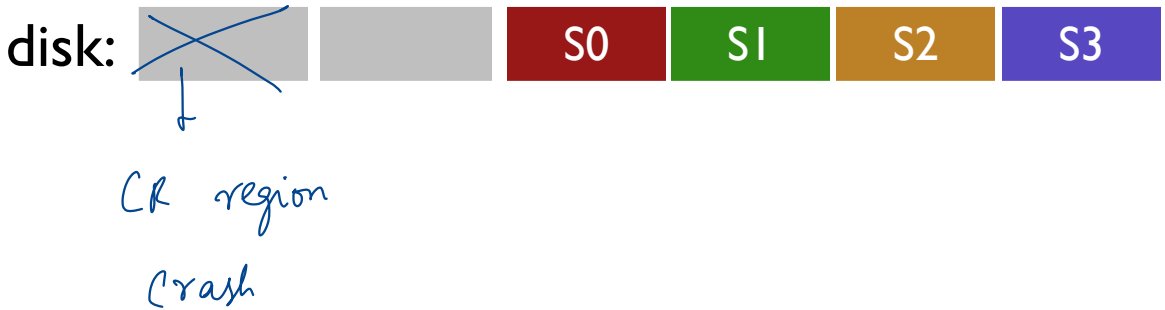
What if crash during checkpoint?

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!