PERSISTENCE: LOG-STRUCTURED FILESYSTEM

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

Project 5: One week to go! ← Due next Thursday

Discussion today: Prep for final exam

5:30 pm Canvas Quiz Practice

35 mins to do it Not graded

~ 6:10 pm Blackboard

AGENDA / LEARNING OUTCOMES

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

What are some similarities or differences with FFS?

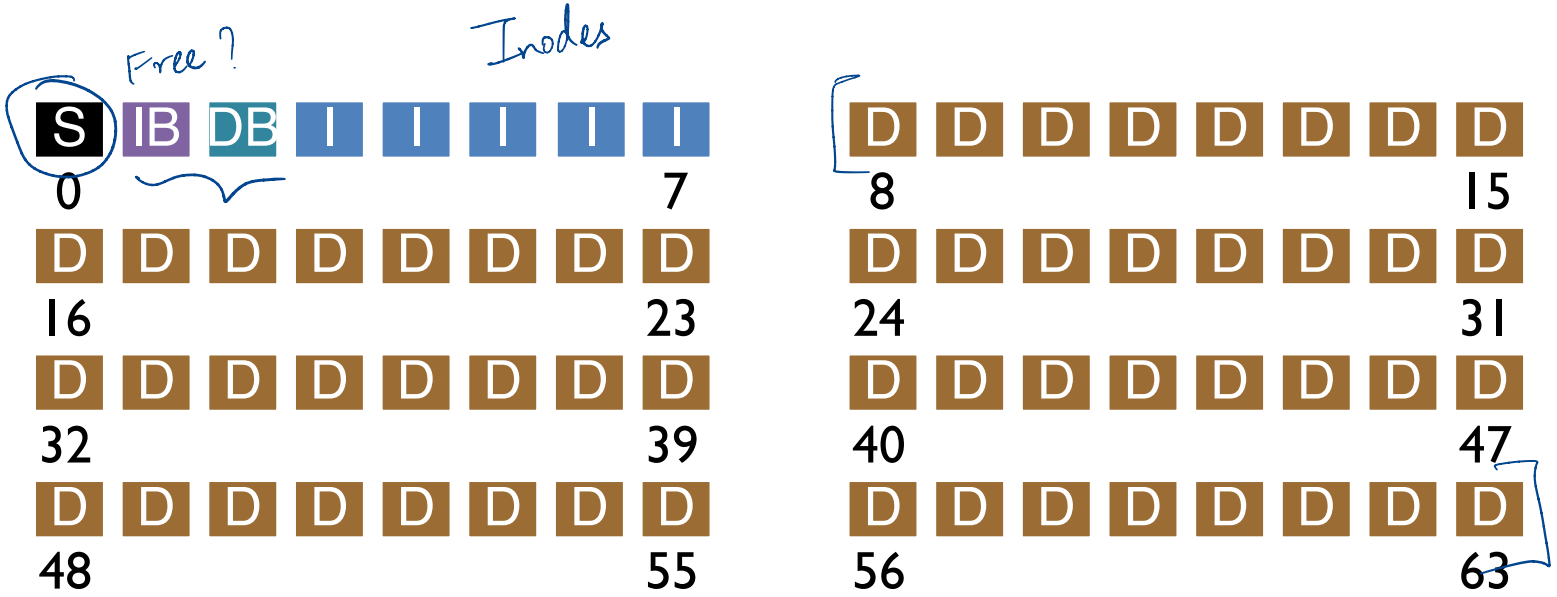
Very Simple FS ⟶ FFS

Log Structured FS

RECAP

FS STRUCTS

FileSystem
↳ collection of files



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

- Can be slow esp for large drive
- Consistent not necessarily correct

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

Read every valid inode+indirect block

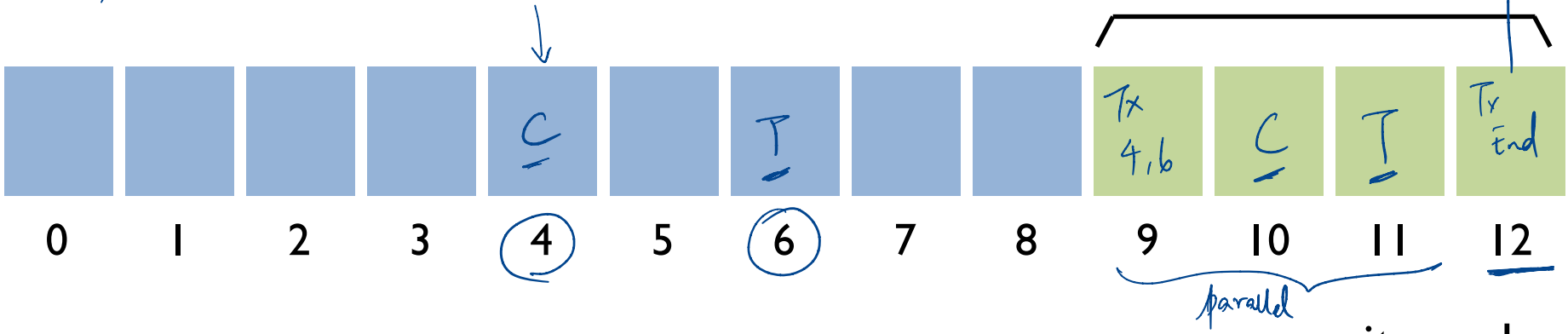
If pointer to data block, the corresponding bit should be 1; else bit is 0

ORDERING FOR CONSISTENCY

Data journaling

Journal commit

Journal



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

Atomicity

Data written twice in disk

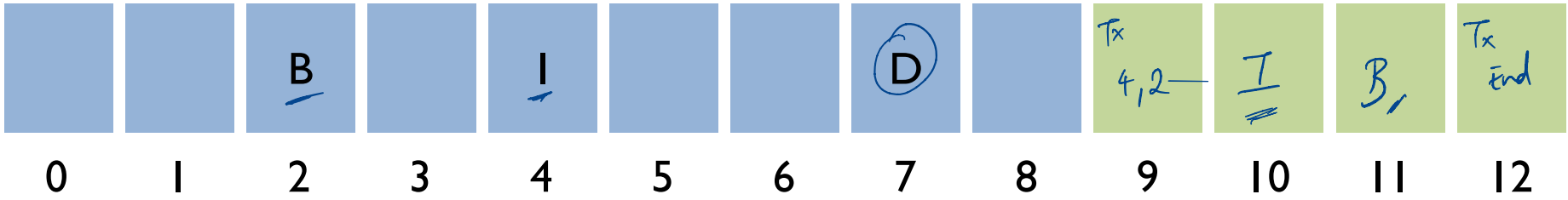
Barriers

write order
9,10,11
12
4,6

ORDERED JOURNAL

Only the journal metadata

Journal can be reused for other transaction



First data block is written 7

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

9, 10, 11

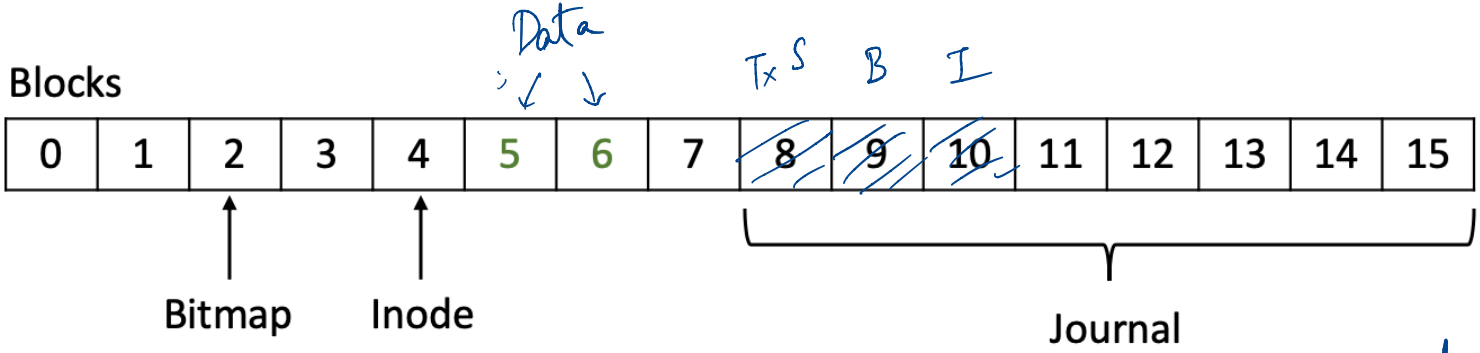
12

2, 4

Reuse Journal

QUIZ 28

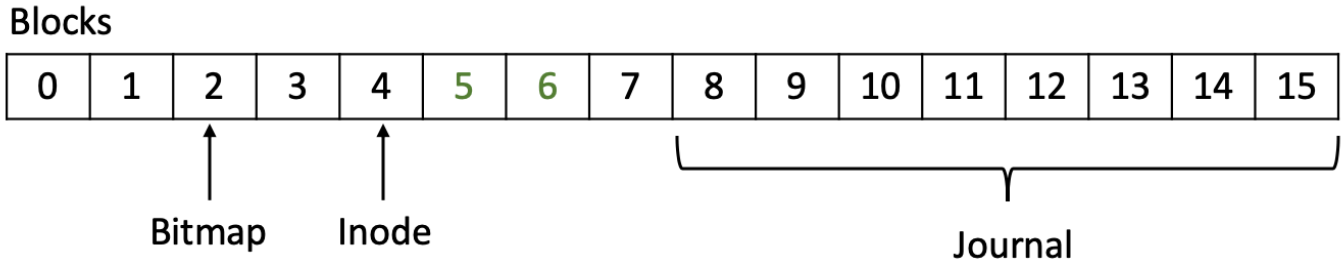
https://tinyurl.com/cs537-sp20-quiz28



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

→ Data blocks in-place
Journal/ start Txn
Txn End
Update metadata in place

Metadata
Ordered journaling



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

Txn Start 4 blocks
Barrier
Txn End
In-place updates

Data journaling

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

Txn Start 4 blocks Tx End
In-place updates

Checksum in Tx End → remove barrier

LOG STRUCTURED FILE SYSTEM (LFS)

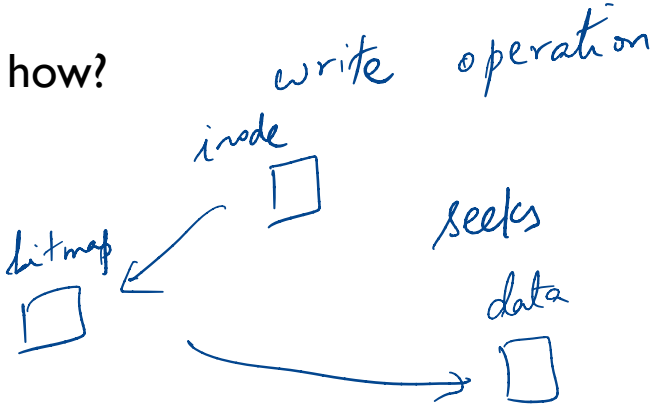
LFS PERFORMANCE GOAL

Motivation:

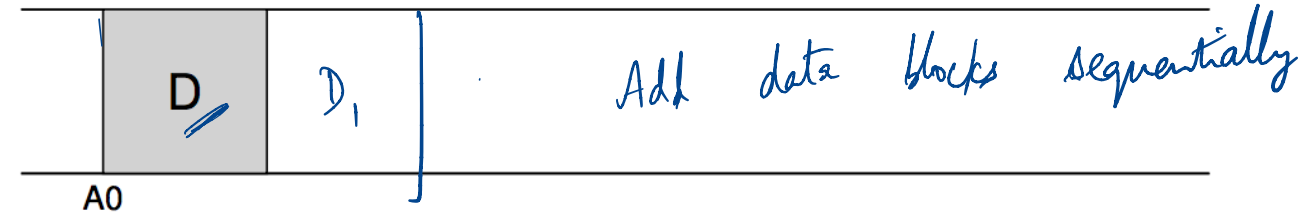
- Growing gap between sequential and random I/O performance
- RAID-5 especially bad with small random writes
  RAID-4

Idea: use disk purely sequentially

Design for writes to use disk sequentially – how?



WHERE DO INODES GO?



Address

Inode is written after a data blocks

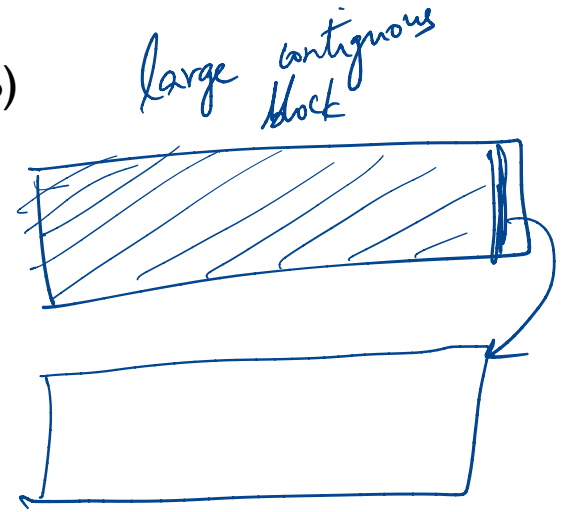
LFS STRATEGY

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

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

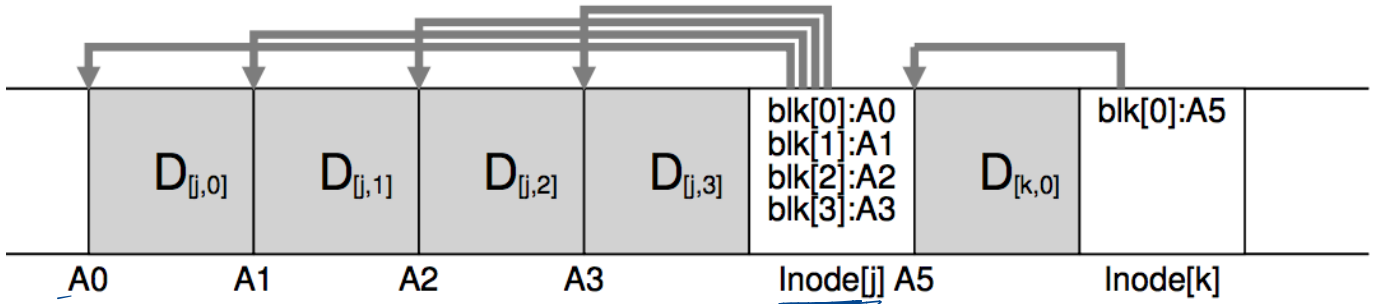
Write buffered data sequentially to new **segment** on disk

Never overwrite old info: old copies left behind



BUFFERED WRITES

Amortization
↳ If buffer size ~10 MBs ⇒ then we get most benefits of sequential



4 blocks to file 1 — 1 block to file 2

WHAT ELSE IS DIFFERENT FROM FFS?

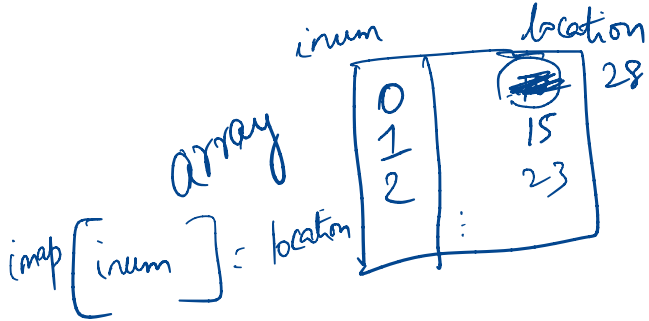
What data structures has LFS removed?

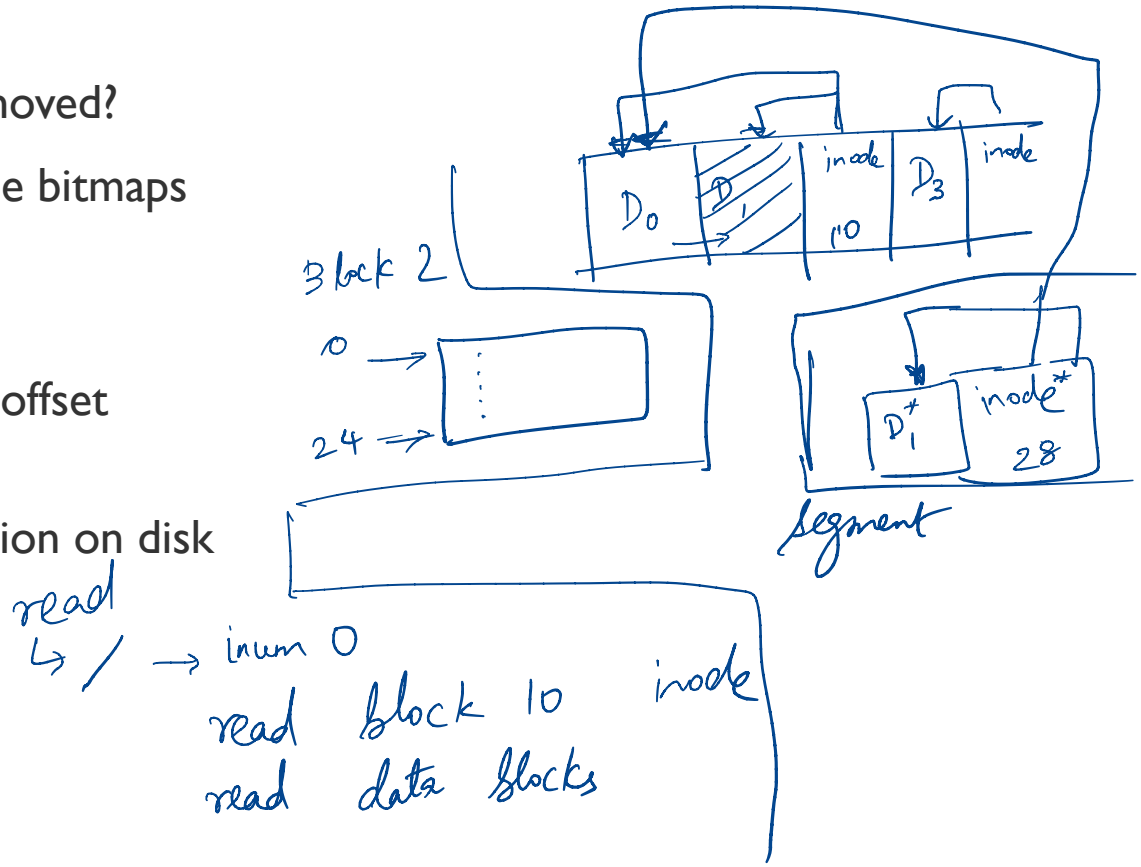
allocation structs: data + inode bitmaps

How to do reads?

Inodes are no longer at fixed offset

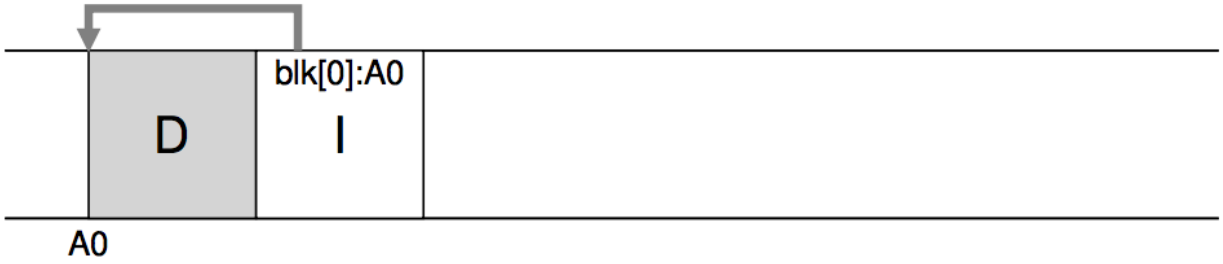
Use imap structure to map:
inode number => inode location on disk

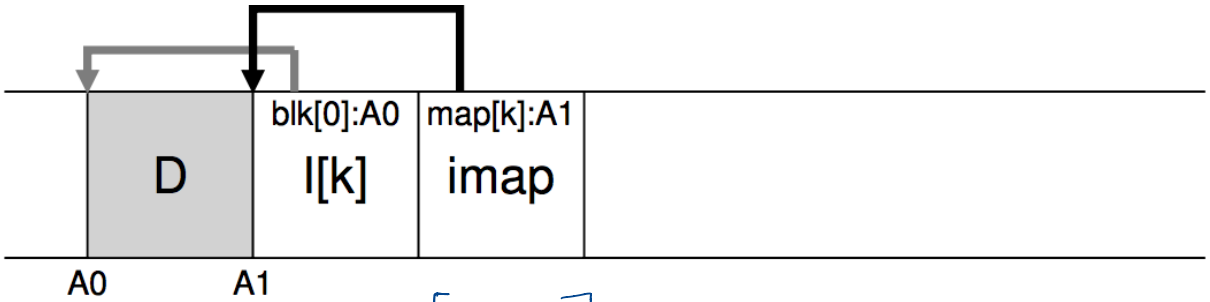




read
↳ / → inum 0
read block 10 inode
read data blocks

IMAP EXPLAINED





[0 : 28]

imap

inum [0 1 2 . . . . .]

addr [~~10~~ 15 23 28]

Cache things in memory

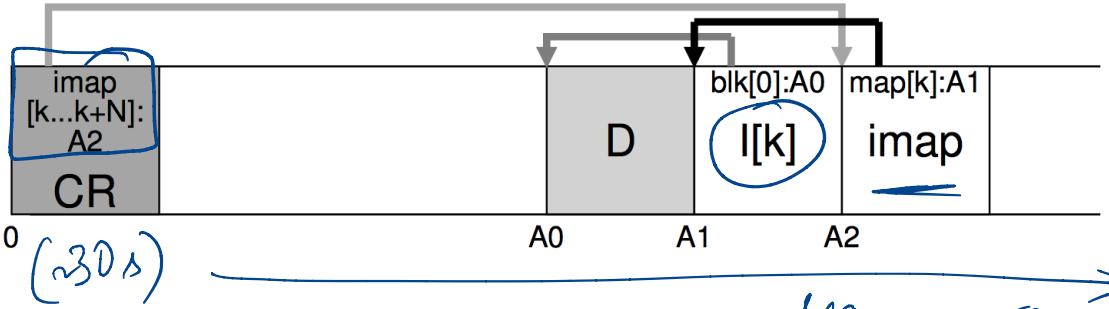
imap cached in memory

Very good write performance

READING IN LFS

imap range | location
0:10 [100]
11:20 142
⋮

Checkpoint region (CR)



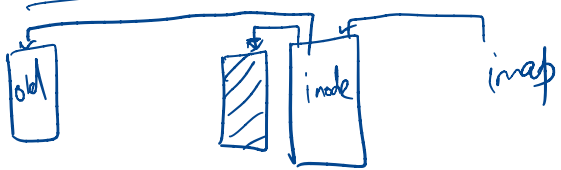
(~30s)

Get the imap in memory

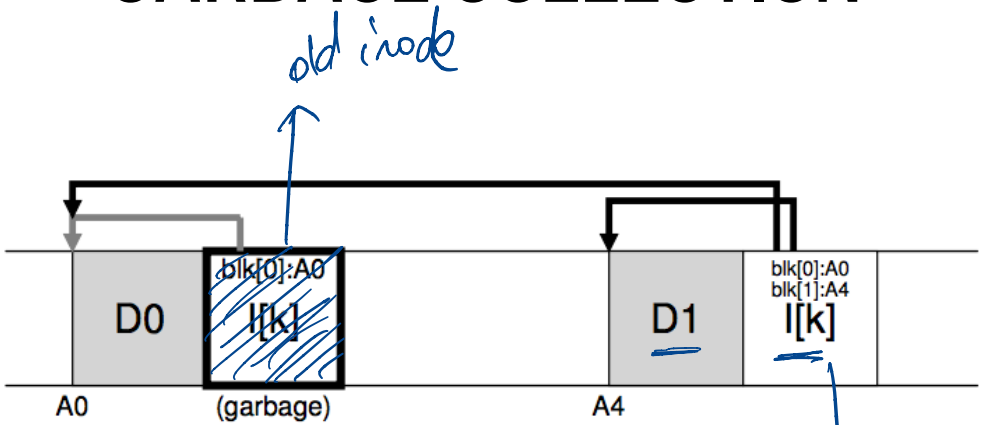
Scan reconstruct the imap

very slow

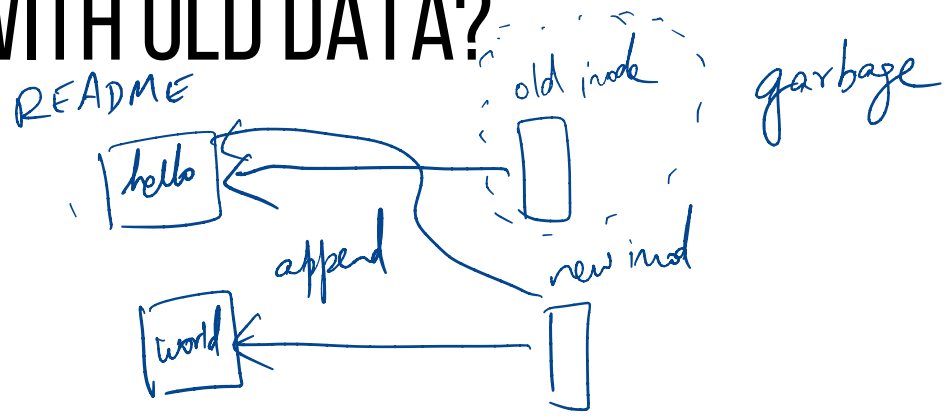
1. Read the Checkpoint region
2. Read all imap parts, cache in mem
3. To read a file: → read inode num: 1
   1. Lookup inode location in imap → inode is at addr 33
   2. Read inode → list of block addrs → direct block at 31
   3. Read the file block → read 31



GARBAGE COLLECTION



WHAT TO DO WITH OLD DATA?



Old versions of files → garbage

Approach 1: garbage is a feature!

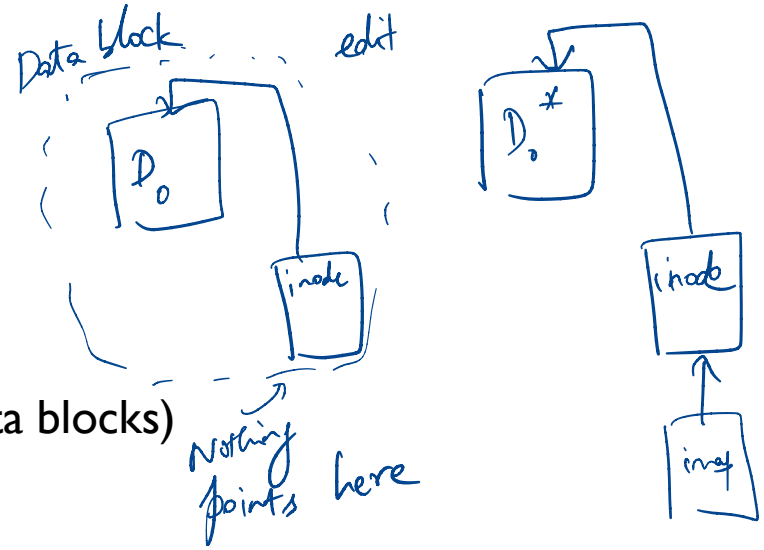
- Keep old versions in case user wants to revert files later
- Versioning file systems
- Example: Dropbox

Approach 2: garbage collection

GARBAGE COLLECTION

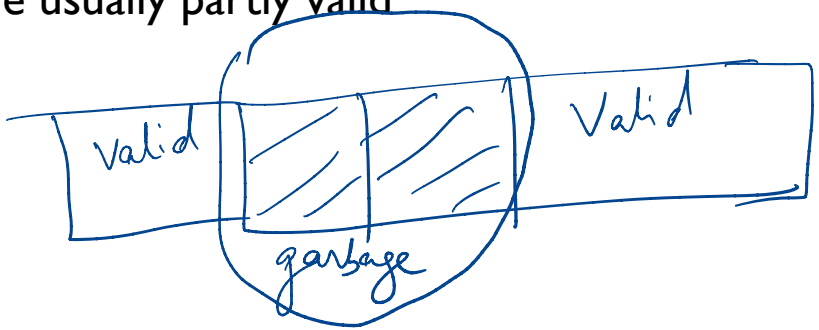
Need to reclaim space:

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

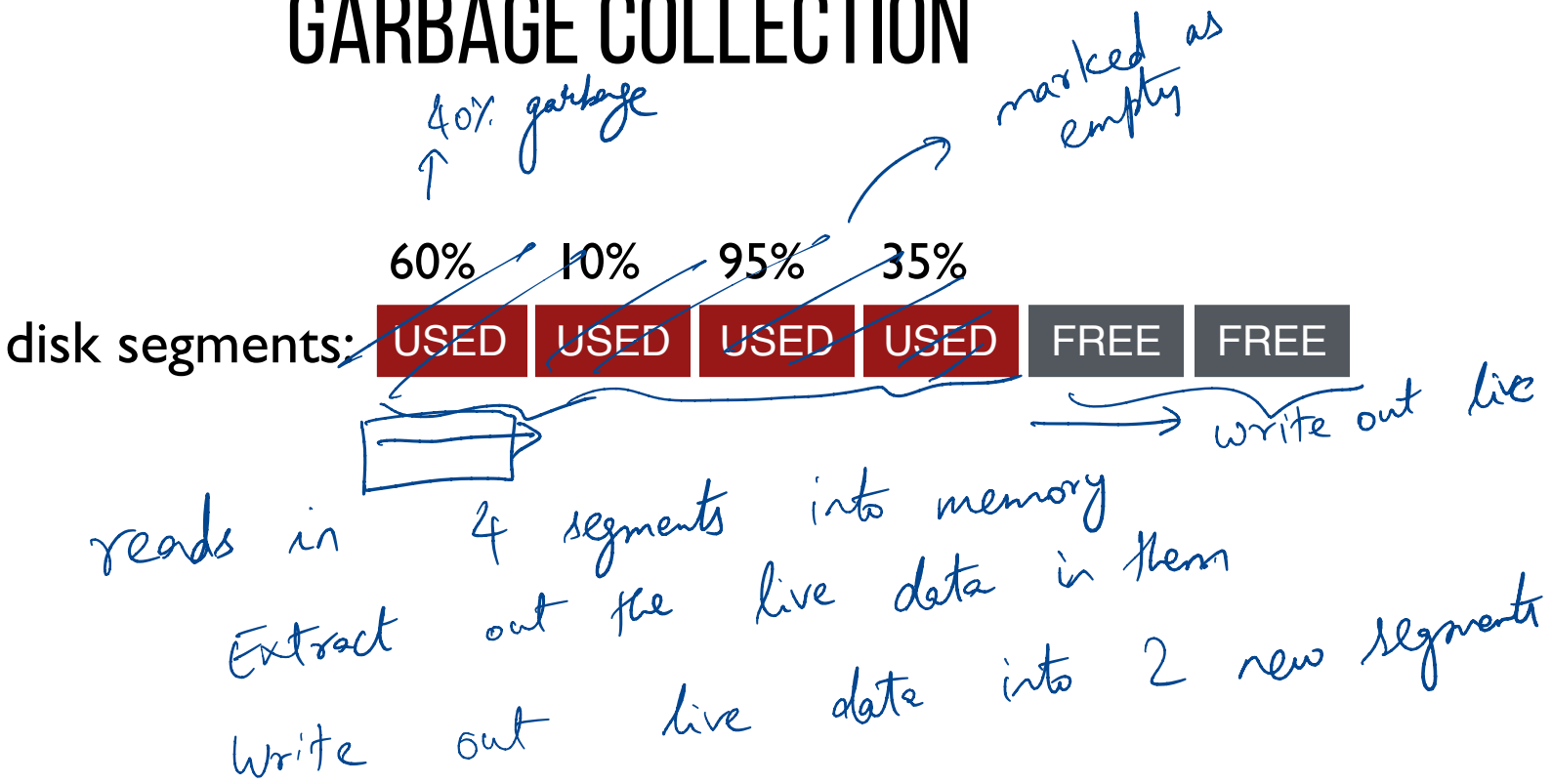


LFS reclaims **segments** (not individual inodes and data blocks)

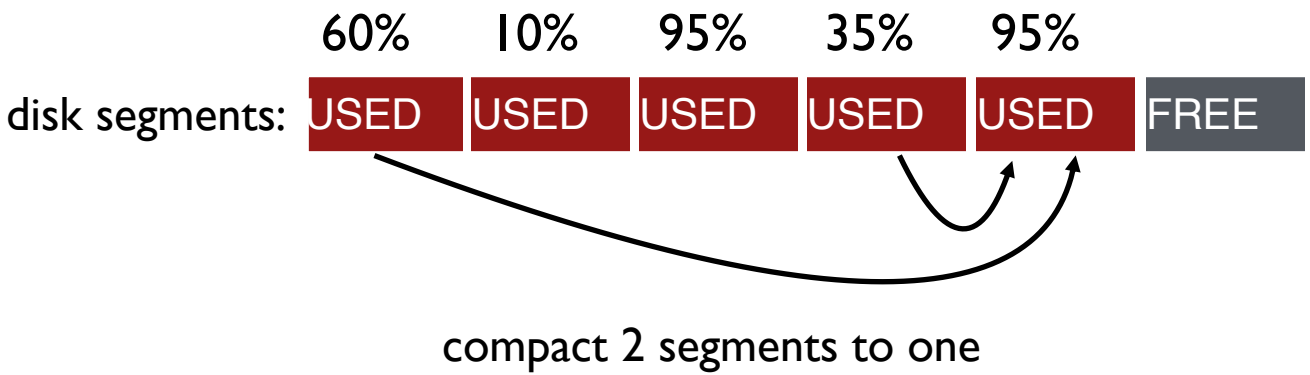
- Want future overwites to be to sequential areas
- Tricky, since segments are usually partly valid



GARBAGE COLLECTION



GARBAGE COLLECTION



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:

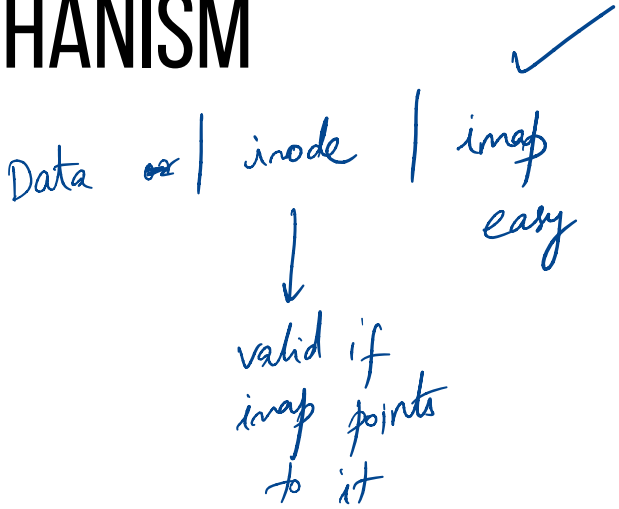
How does LFS know whether data in segments is valid?

Policy:

Which segments to compact?

When

GARBAGE COLLECTION MECHANISM



Is an inode the latest version?

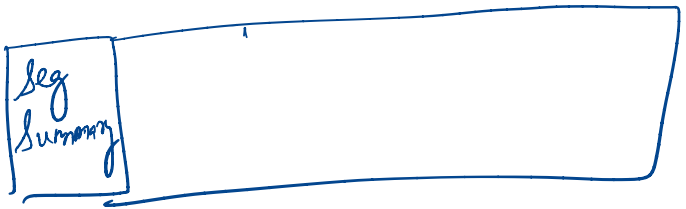
- Check imap to see if this inode is pointed to
- Fast!

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)

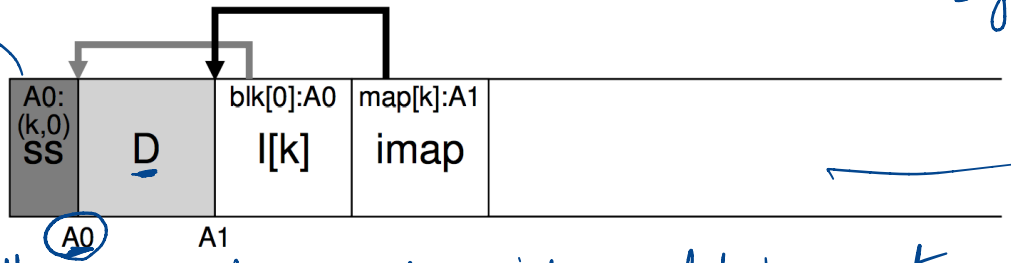


SEGMENT SUMMARY

where is data block → inum → offset

$A_0 \rightarrow (k, 0)$

→ Append segments to disk
→ Data blocks, Inode, imap fragments
→ Segment summary to check data blocks is valid



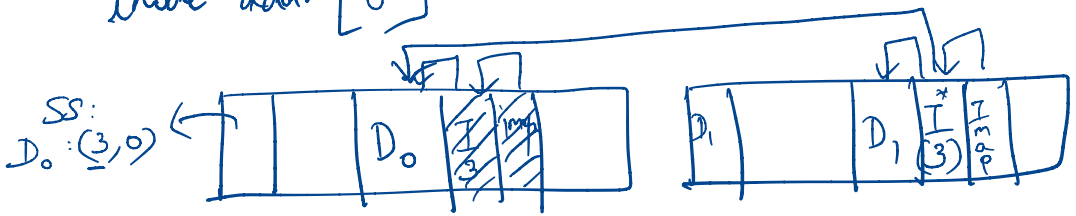
Garbage Collection: to check if block at addr A is valid?

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

inode = Read(imap[N]);

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

$(k, 0) \leftarrow SS[A_0]$

inode ← Read(imap[K]) → inode loc of K

inode.addr[0] == A_0



GARBAGE COLLECTION

General operation:

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

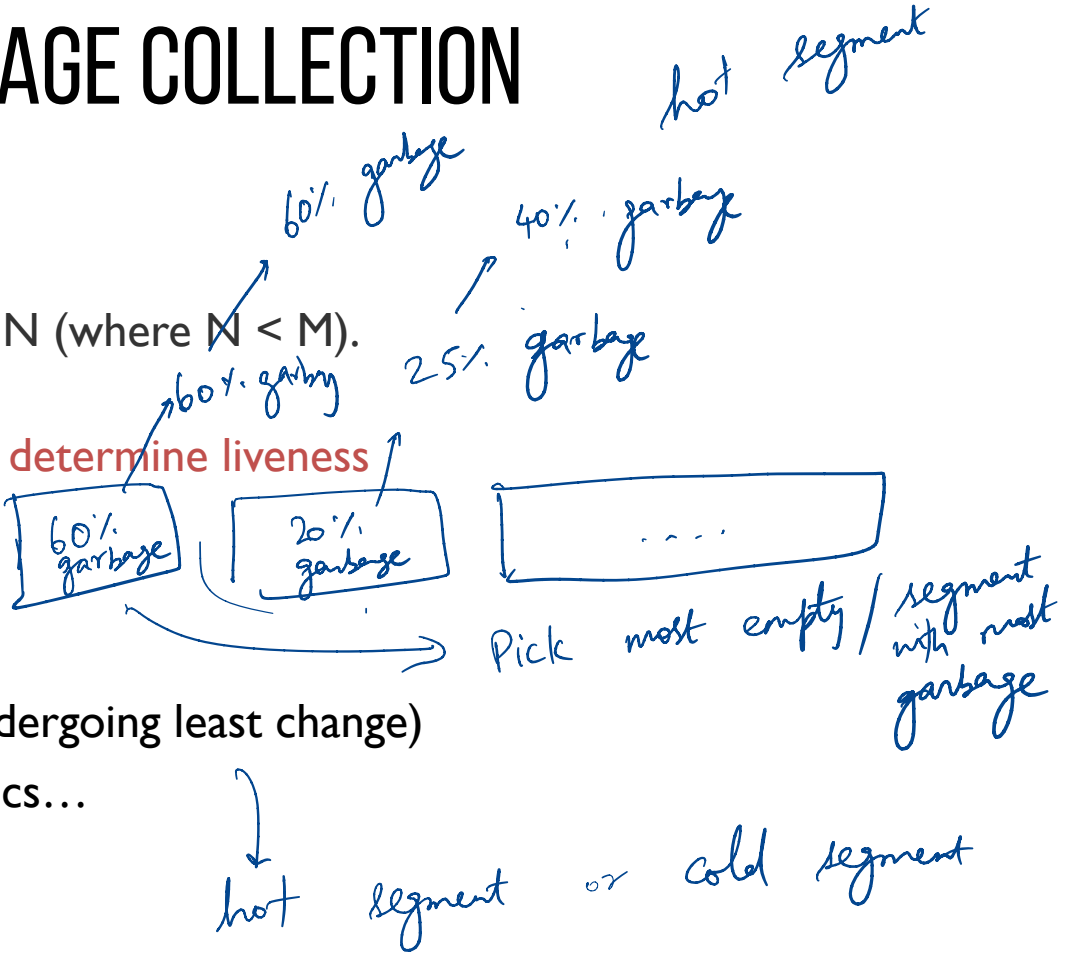
Mechanism:

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)

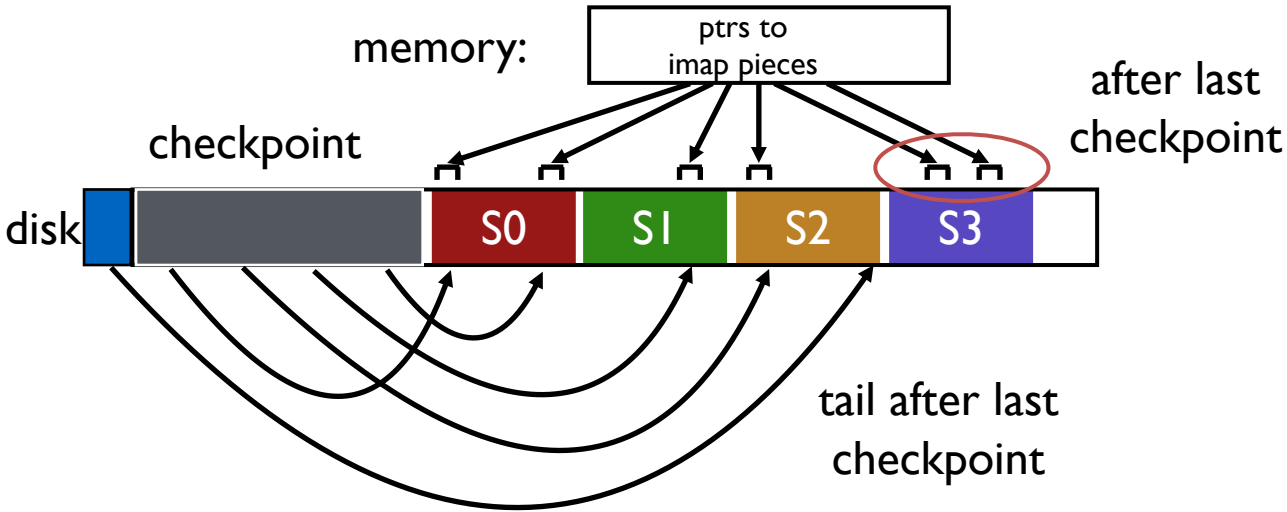
Better approach?

- Occasionally save to checkpoint region the pointers to imap pieces

How often to checkpoint?

- Checkpoint often: 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 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: FFS

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

QUIZ 29

https://tinyurl.com/cs537-sp20-quiz29

```
block 100: [("." 0), (".." 0), ("foo" 1)]   // a data block
block 101: [size=1,ptr=100,type=d]          // an inode
block 102: [size=0,ptr=-,type=r]            // an inode
block 103: [imap: 0->101,1->102]            // a piece of the imap
```

```
block 104: [SOME DATA]                          // a data block
block 105: [SOME DATA]                          // a data block
block 106: [size=2,ptr=104,ptr=105,type=r]      // an inode
block 107: [imap: 0->101,1->106]                // a piece of the imap
```

If we fill out 100 segments in a newly created LFS, how long does it take to complete a write?

If we read this file (reads do not hit cache), how long does it take to read the entire file?

If we now read this file backwards, one segment at a time (and reads do not hit in cache), how long does this backwards read take?

NEXT STEPS

Project 5 is one week away!

Discussion: Final practice quiz