PERSISTENCE: LFS
COPY-ON-WRITE FILE SYSTEMS

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

Make-up Points for Project 4

Due November 25

Project 6: Due Friday (5pm or midnight)

Project 7: Available immediately after Thanksgiving

xv6 File systems: Improvements + checker?

Will have Specification Quiz

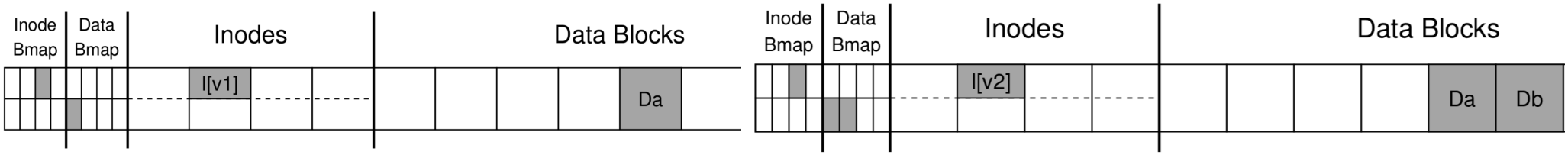
AGENDA / LEARNING OUTCOMES

Besides Journaling, how else can disks be updated atomically?

- Does on-disk **log** help performance of writes or reads?
- How to **find inodes** in on-disk log?
- How to **recover** from a crash in LFS?
- How to **garbage collect** dead information in LFS?

RECAP

FILE APPEND EXAMPLE



Written to disk	Result
Db	Lost data (nothing bad)
I[v2]	Point to garbage; another file could use data block
B[v2]	Space leak
I[v2] + B[v2]	Point to garbage data
I[v2] + Db	Another file could use same datablock
B[v2] + Db	Space leak

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

FSCK CHECKS

Do superblocks match?

Is the list of free blocks correct?

Do number of dir entries equal inode link counts?

Do different inodes ever point to same block?

Are there any bad block pointers?

Do directories contain "." and ".."?

…

CONSISTENCY SOLUTION #2: JOURNALING

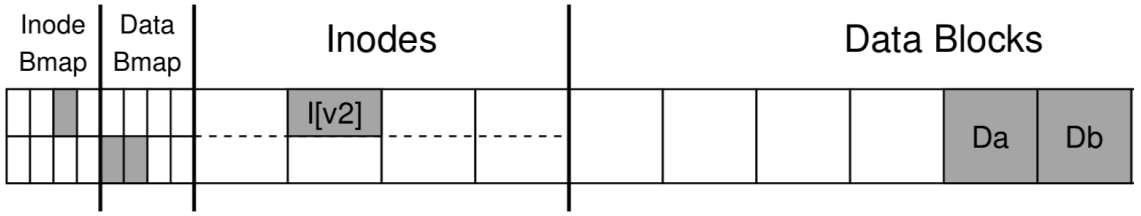
Goals

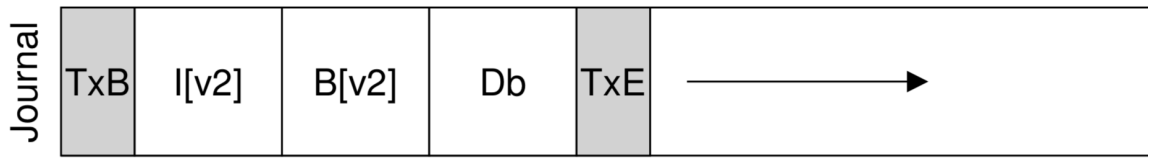
- Ok to do some **recovery work** after crash, but not to read entire disk
  - Recovery: Replaying committed transactions in log
- Don't move file system to just any consistent state, get **correct** state

Atomicity

- Definition of atomicity for **concurrency:** operations in critical sections are not interrupted by operations on related critical sections
- Definition of atomicity for **persistence:** collections of writes are not interrupted by crashes; either (all new) or (all old) data is visible

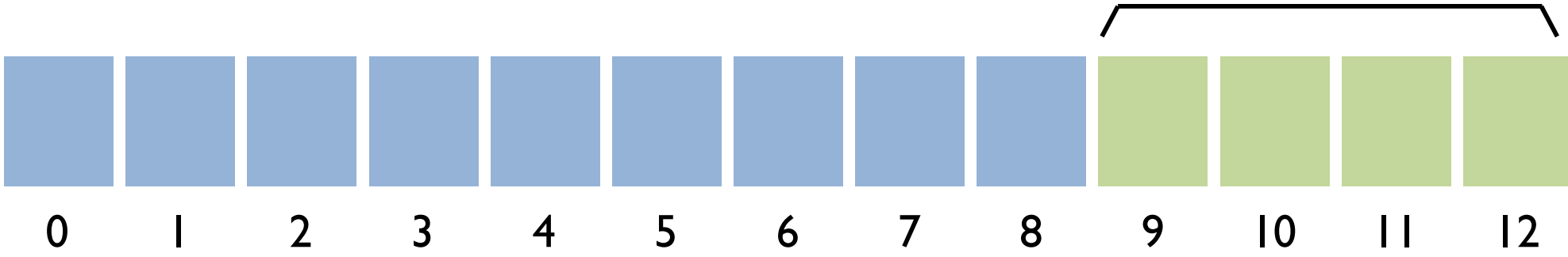
JOURNAL LAYOUT





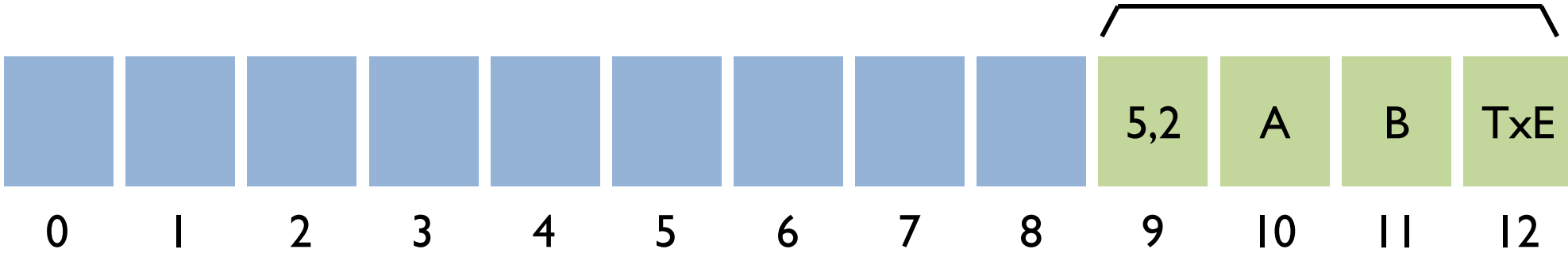
Transaction

DATA JOURNAL WRITE AND CHECKPOINTS



transaction: write A to block 5; write B to block 2

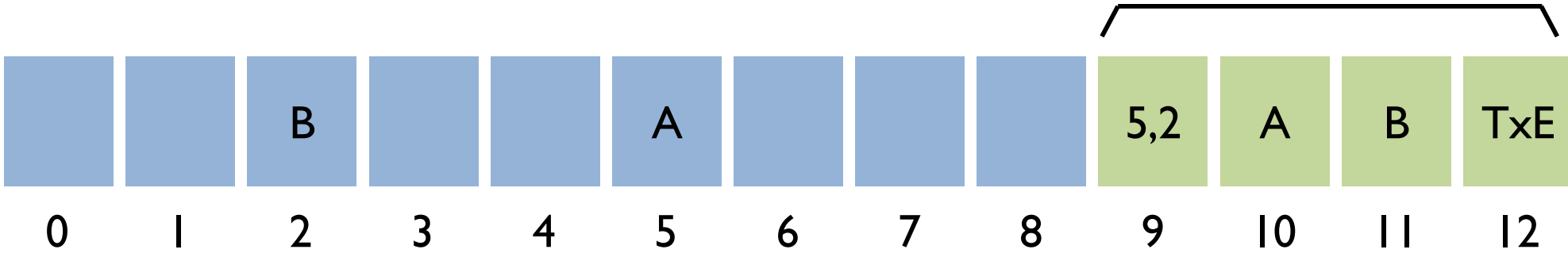
DATA JOURNAL WRITE AND CHECKPOINTS



Journal descriptor block: What is in this transaction?
TxE: Journal commit

Checkpoint: Writing new data to in-place locations

DATA JOURNAL WRITE AND CHECKPOINTS

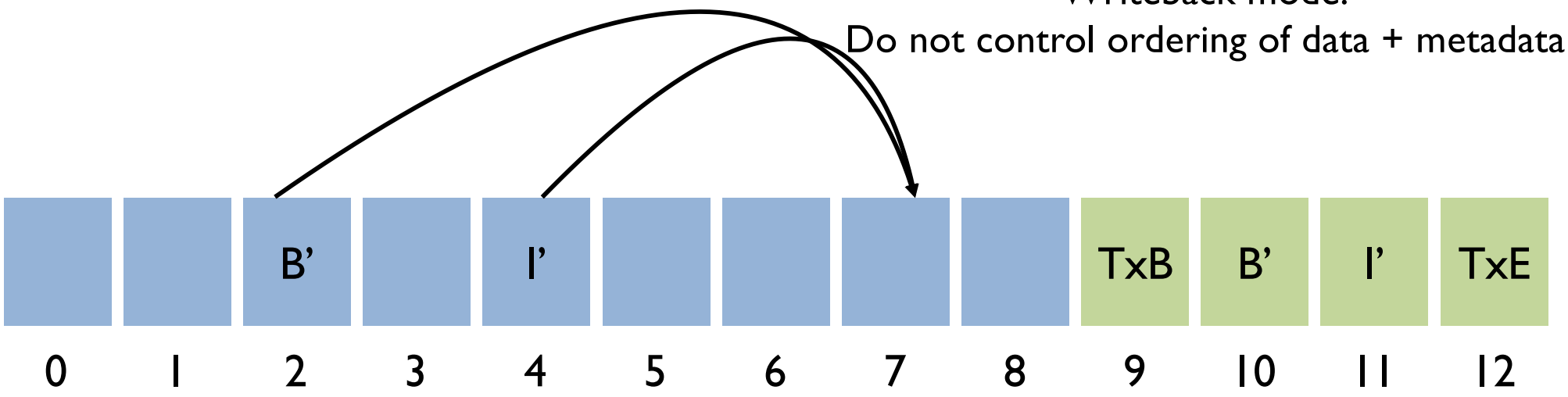


Journal descriptor block: What is in this transaction?
TxE: Journal commit

Checkpoint: Writing new data to in-place locations

METADATA JOURNALING

Writeback mode:
Do not control ordering of data + metadata

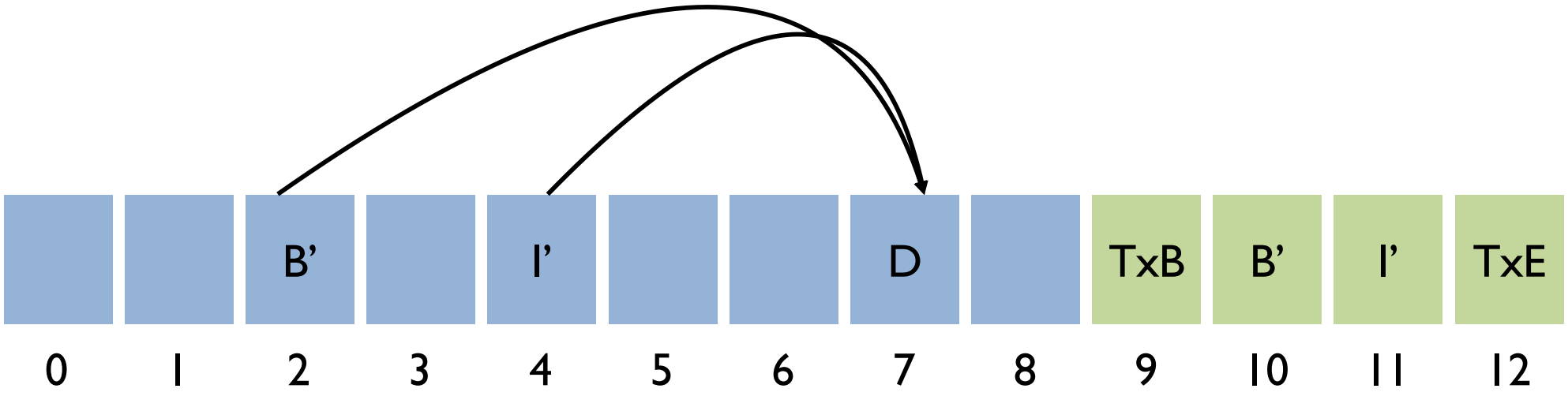


transaction: append to inode I
Ensures B' and I' are updated atomically

Crash !?!

Point to garbage data, but data block won't be allocated to another file

ORDERED (META-DATA) JOURNALING



transaction: append to inode I
Ensures B' and I' are updated atomically AFTER D is on disk

Crash !?!

Everything is all good; if didn't clear TxE, will replay transaction, extra work but no problems

IDENTIFY THE KIND OF JOURNALING?

We need to write data in block 5,6. Inode is block 4, bitmap in block 2 (in-place blocks)
Journal is from blocks 8 to 15

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

LOG STRUCTURED FILE SYSTEM (LFS)

FILE-SYSTEM CASE STUDIES

Local

- **FFS**: Fast File System
- ext3, ext4: Journaling File Systems
- **LFS**: Log-Structured File System;
  - Copy-On-Write (COW) (ZFS, btrfs)

Network

- **NFS**: Network File System
- **AFS**: Andrew File System

GENERAL STRATEGY FOR CRASH CONSISTENCY

Never delete ANY old data, until ALL new data is safely on disk

Implication:

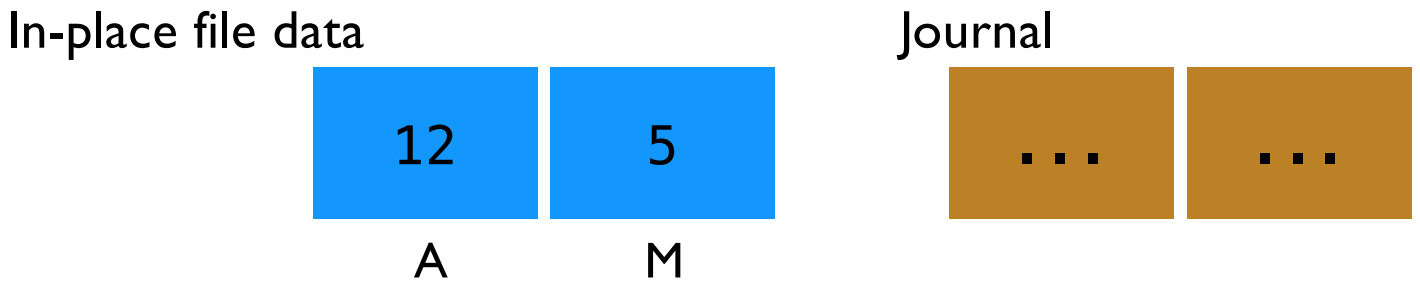
At some point in time, all old AND all new data must be on disk

Two techniques popular in file systems:

1. **journal** make note of new info, then overwrite old info with new info **in place**

2. **copy-on-write**: write new info to new location, discard old info (update pointers)

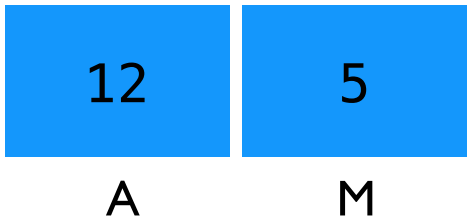
REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE



Goal: Write 10 to Block A and 7 to Block M

REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE

In-place file data



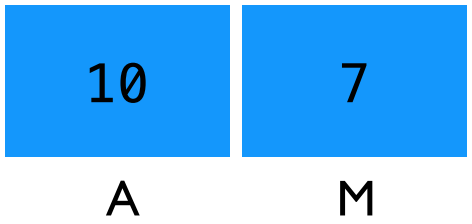
Journal

Imagine journal header describes in-place destinations

Imagine journal commit block designates transaction complete

REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE

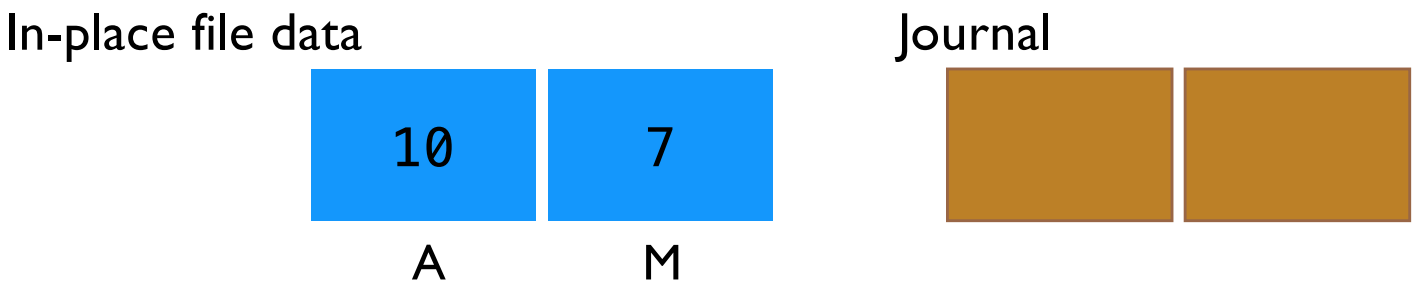
In-place file data



Journal

Perform checkpoint to write data to in-place destinations

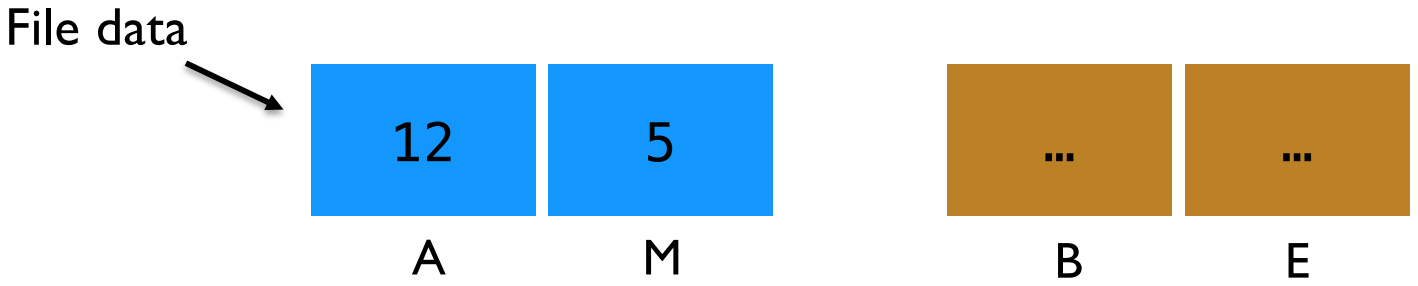
REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE



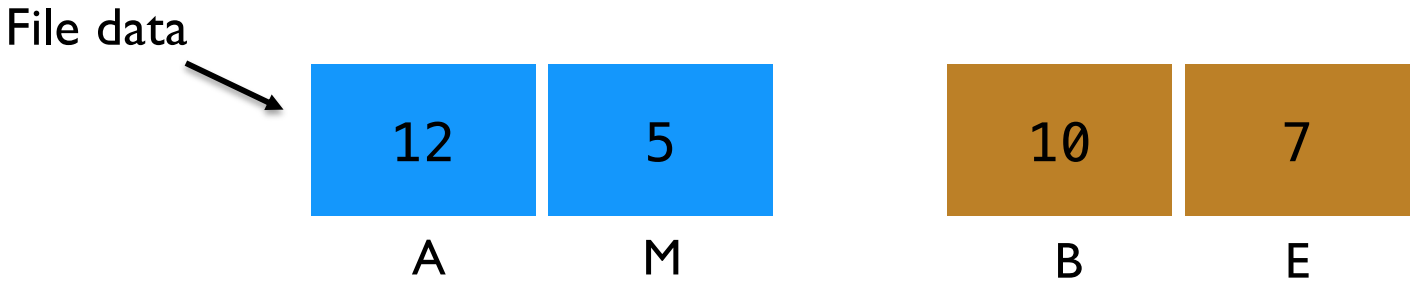
Clear journal commit block to show checkpoint complete

Especially expensive for full data-mode journaling

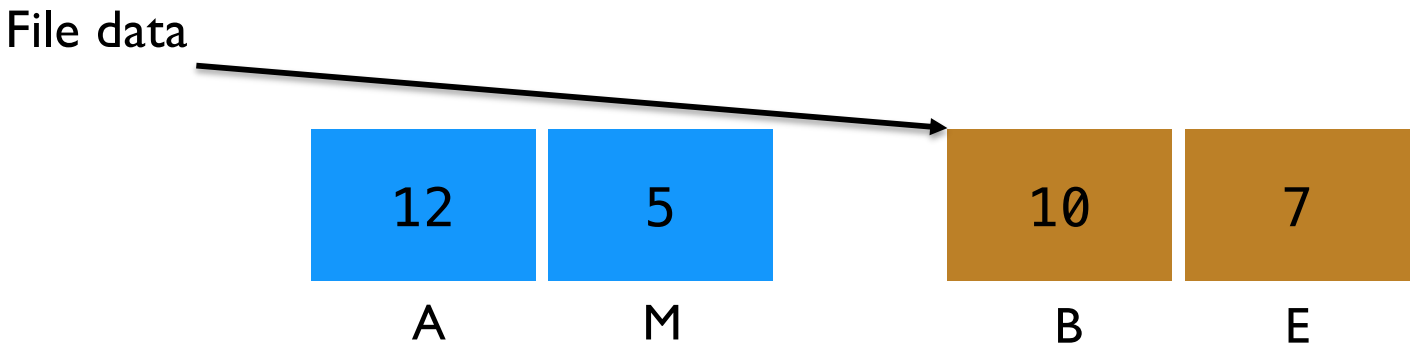
TODAY: WRITE NEW, DISCARD OLD
COPY-ON-WRITE



TODAY: WRITE NEW, DISCARD OLD
COPY-ON-WRITE



TODAY: WRITE NEW, DISCARD OLD
COPY-ON-WRITE



Obvious advantage?

Only write new data once instead of twice

LFS PERFORMANCE GOAL

Motivation:

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

Idea: use disk purely sequentially

Design for writes to use disk sequentially – why?

- Can do all writes near each other to empty space – new copy
- Works well with RAID-5 (large sequential writes)

Hard for reads – why?

- User might read files X and Y not near each other on disk
- Maybe not be too bad if disk reads are slow – why?
  - Memory sizes are growing (cache more reads)

LFS STRATEGY

File system buffers writes (metadata + data) in main memory until "enough" data

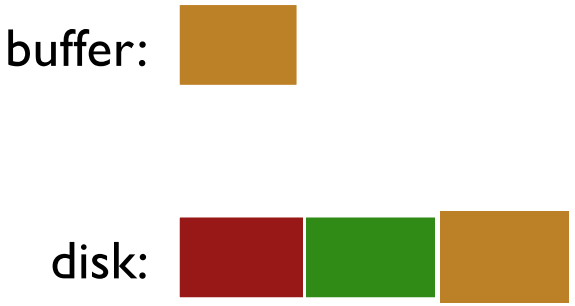
- How much is enough?
- Enough to get good sequential bandwidth from disk (MB)

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

- Segment is some contiguous regions of blocks

Never overwrite old info: old copies left behind

BUFFER WRITES IN MEMORY



WHAT ELSE IS DIFFERENT FROM FFS?

LFS builds on FFS data structures: directories, inodes, indirect blocks, data blocks

What data structures from FFS could LFS remove?

allocation structs: data bitmaps

How to do reads?

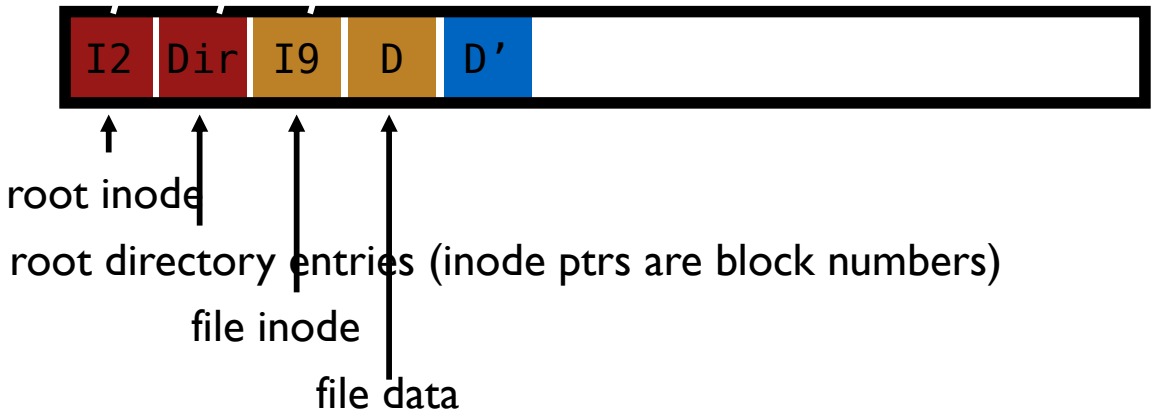
Inodes are no longer at fixed offset; how will find them?

Attempt #1: Use current offset on disk instead of table index (inode #) for name

Problem: When update inode, write to new location, inode name changes

ATTEMPT 1: INODE NUMBER IS DISK OFFSET

Overwrite data in /file.txt



How to update Inode 9 to point to new D' ???

ATTEMPT 1: INODE NUMBER IS DISK OFFSET

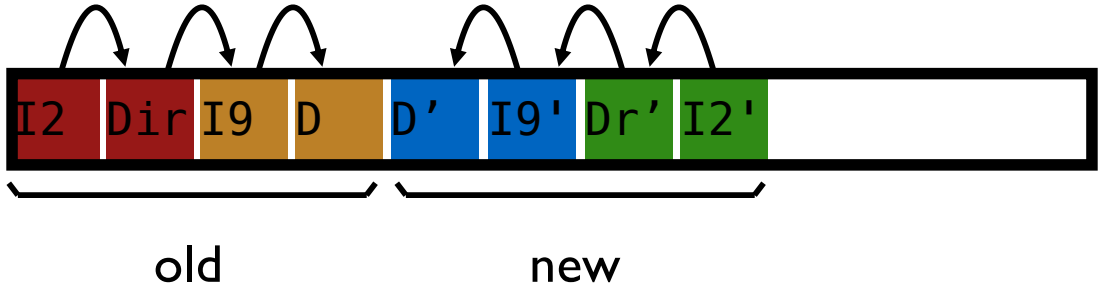
Overwrite data in /file.txt

Can LFS update Inode 9 to point to new D’?

NO! This would be a random write

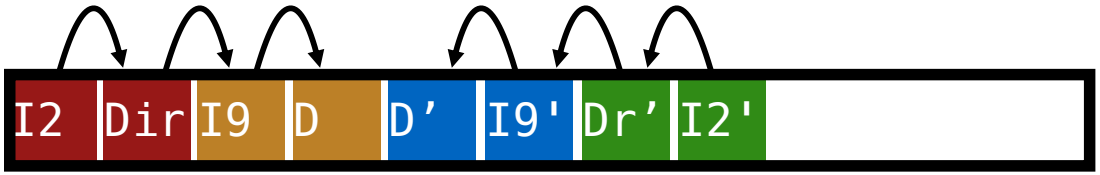
ATTEMPT 1: INODE NUMBER IS DISK OFFSET

Overwrite data in /file.txt



Problem: Must update all structures in sequential order to log

ATTEMPT 1: PROBLEM W/ INODE NUMBERS



Problem:

For every data update, must propagate updates all the way up directory tree to root

Why?

When inode copied, its location (inode name) changes

Solution:

Keep inode names constant; don't base inode name on offset

FFS found inodes with math. How in LFS?

DATA STRUCTURES (ATTEMPT 2)

What type of name is much more complicated?

Inodes are no longer at fixed offset

Use imap structure to map:

inode number => inode location on disk

WHERE TO KEEP IMAP?

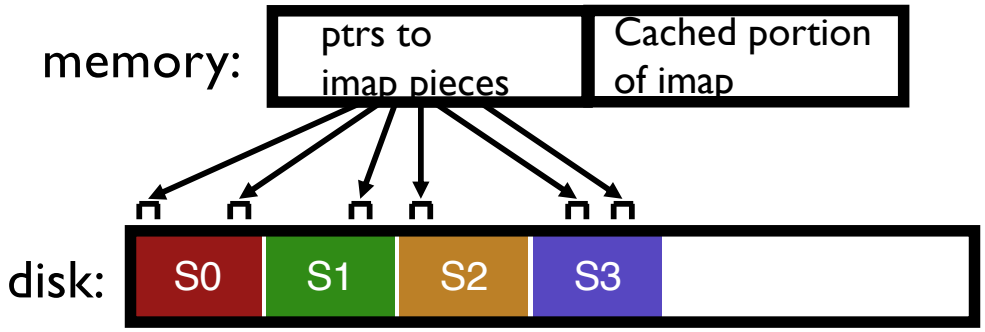
imap: inode number => inode location on disk

table of millions of entries (4 bytes each)

Where can imap be stored???? Dilemma:
1. imap too large to keep in memory (and must be persistent)
2. don't want to perform random writes for imap
   (don't want to go to beginning of disk each time)

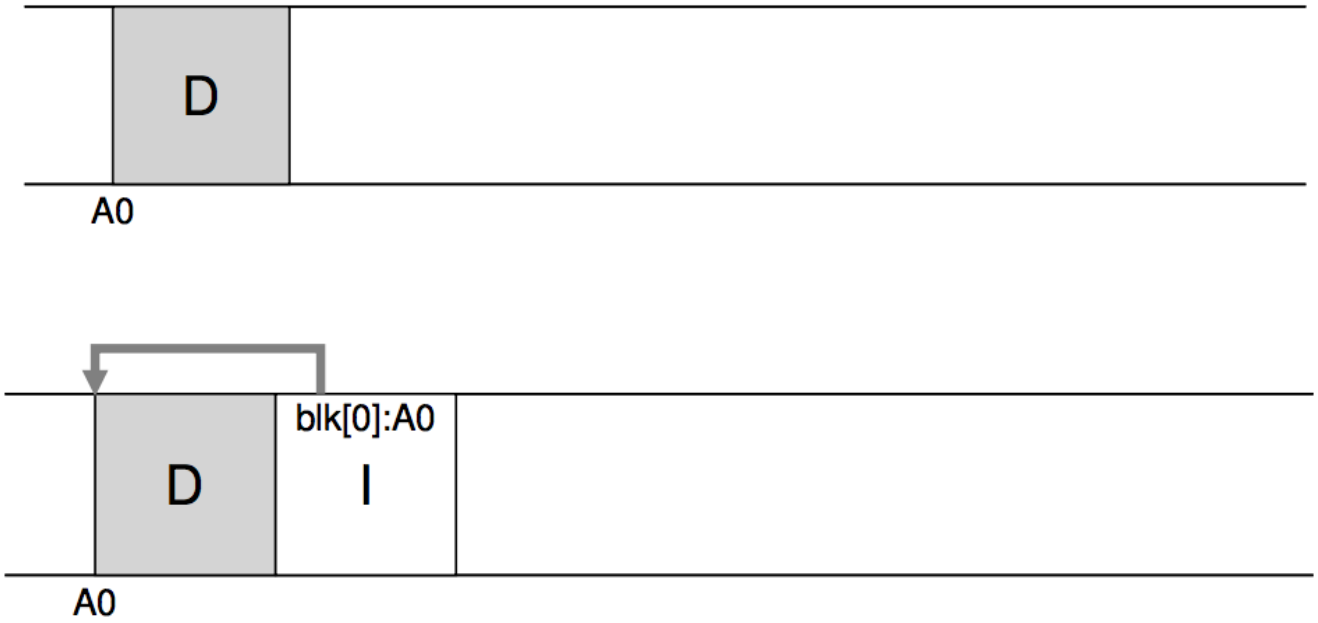
Solution:
Write imap in segments
Keep pointers to pieces of imap in memory (crash? fix this later!)

SOLUTION: IMAP IN SEGMENTS



Solution:
Write imap in segments
Keep pointers to pieces of imap in memory (crash? fix this later!)
Keep recent accesses to imap cached in memory

WHERE DO INODES GO?

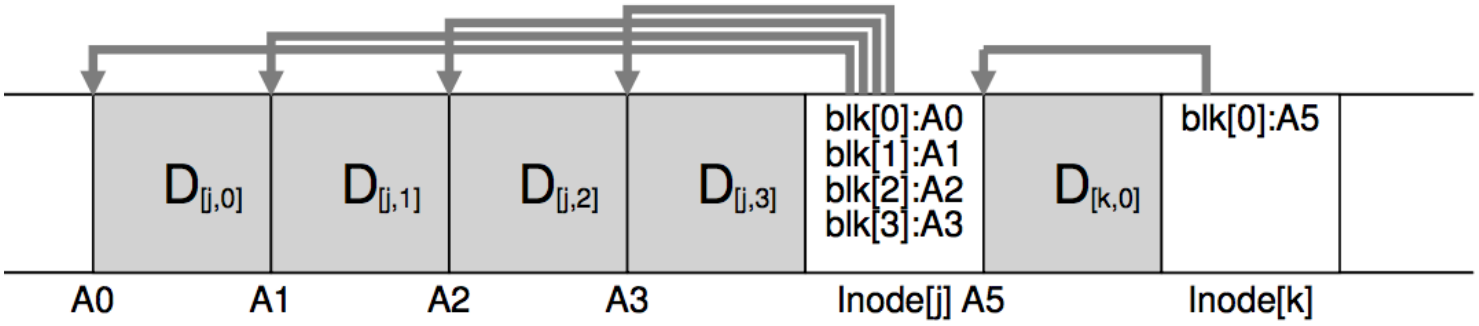


Write out new copy of data, even if already allocated and just "overwriting" block of file with new data

(Would have been a simple data overwrite in FFS)

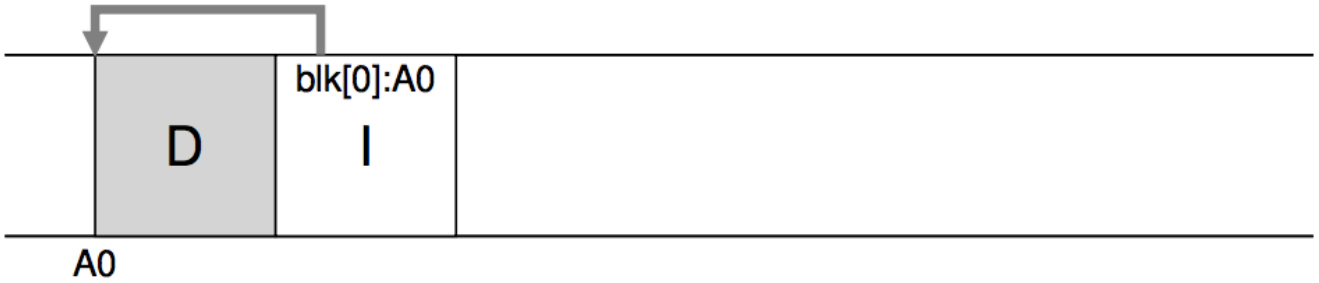
Will need to change pointers in inode if write data to new location

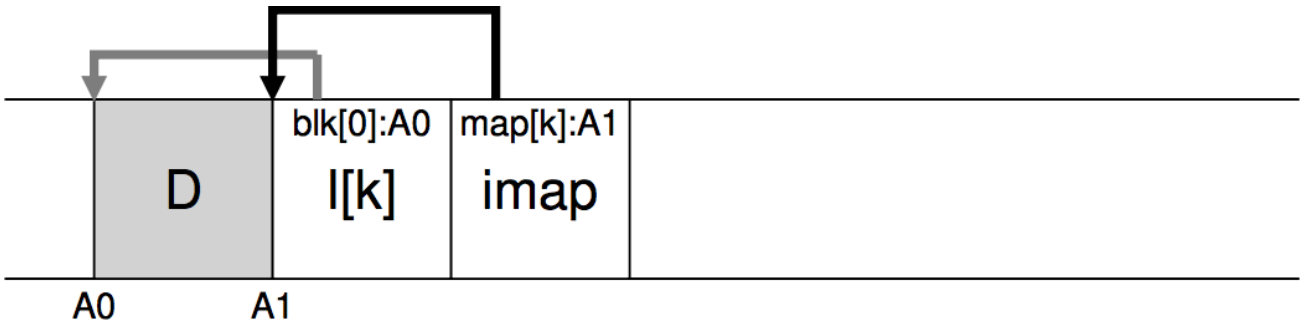
BUFFERED WRITES TO SEGMENT



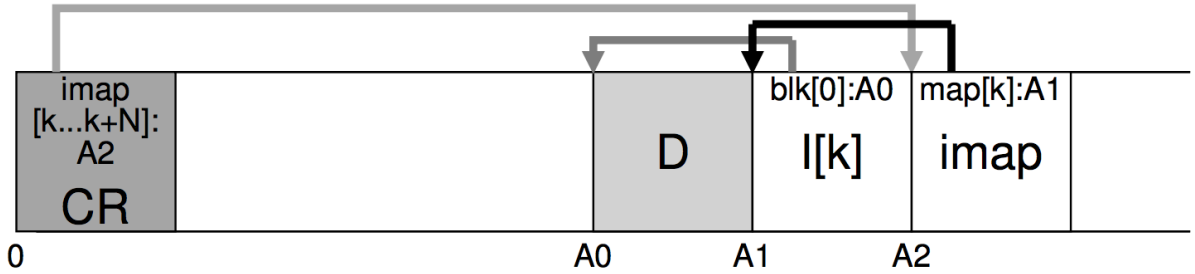
If some data in file has not changed, those data blocks will not be re-written
Pointers in inode may point to old data in other segments

IMAP EXPLAINED





READING IN LFS



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

OTHER ISSUES

Crashes

Garbage Collection

CRASH RECOVERY

What data needs to be recovered after a crash?

- Need imap (lost in volatile memory)

Naive approach?

- **Scan** entire disk to reconstruct imap pieces. Slow!

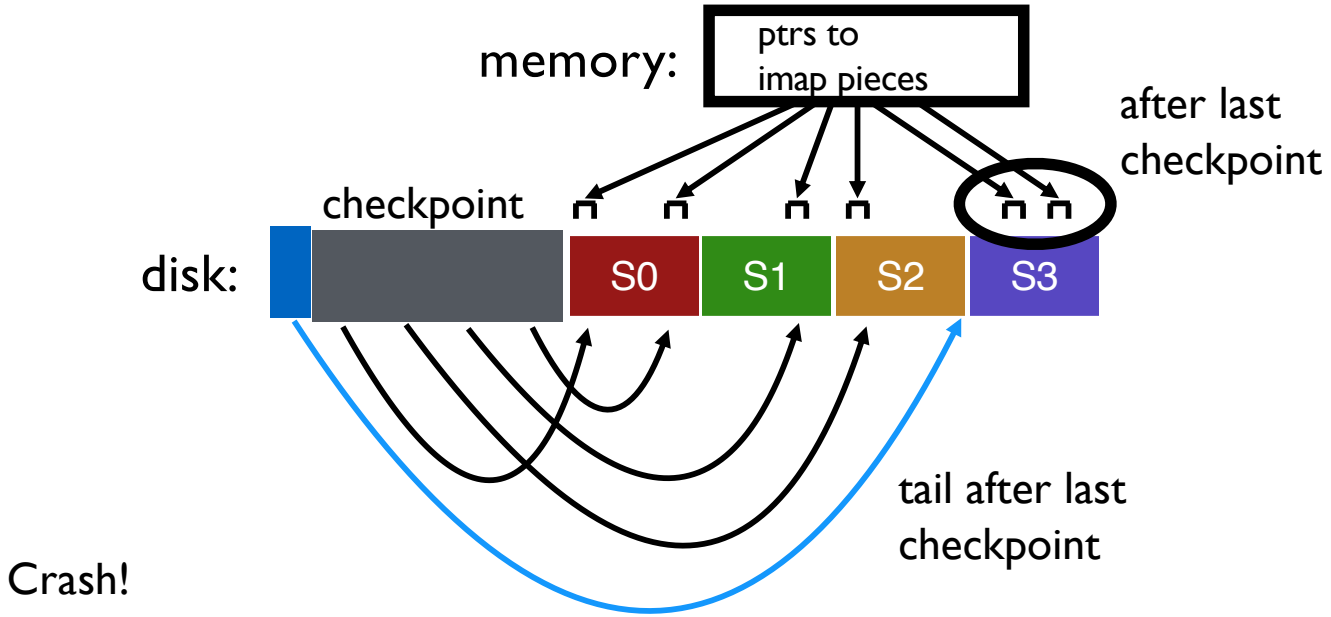
Better approach?

- Occasionally **checkpoint** to known on-disk location pointers to imap pieces

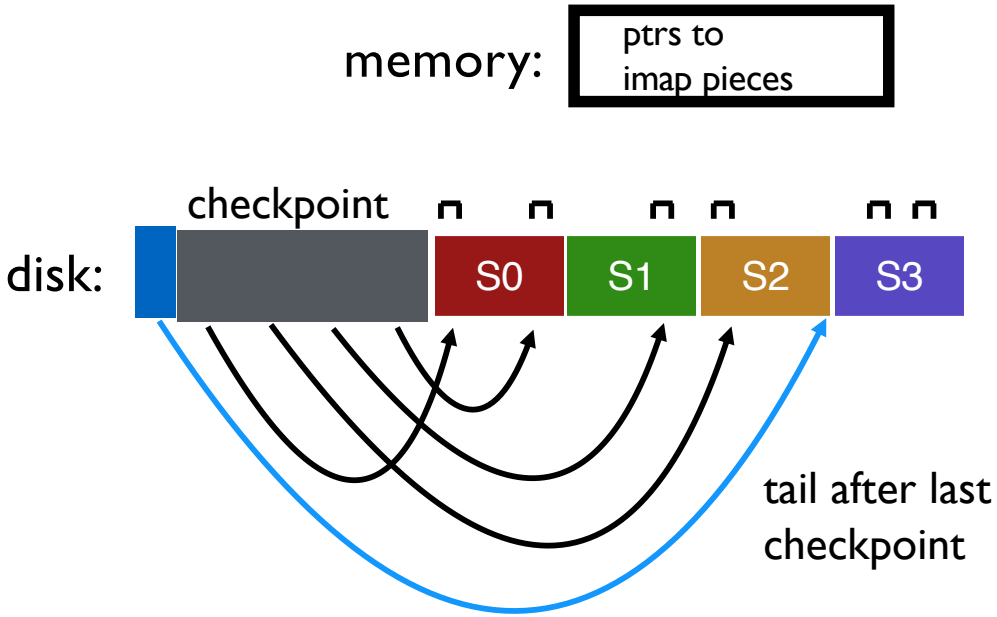
How often to checkpoint?

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

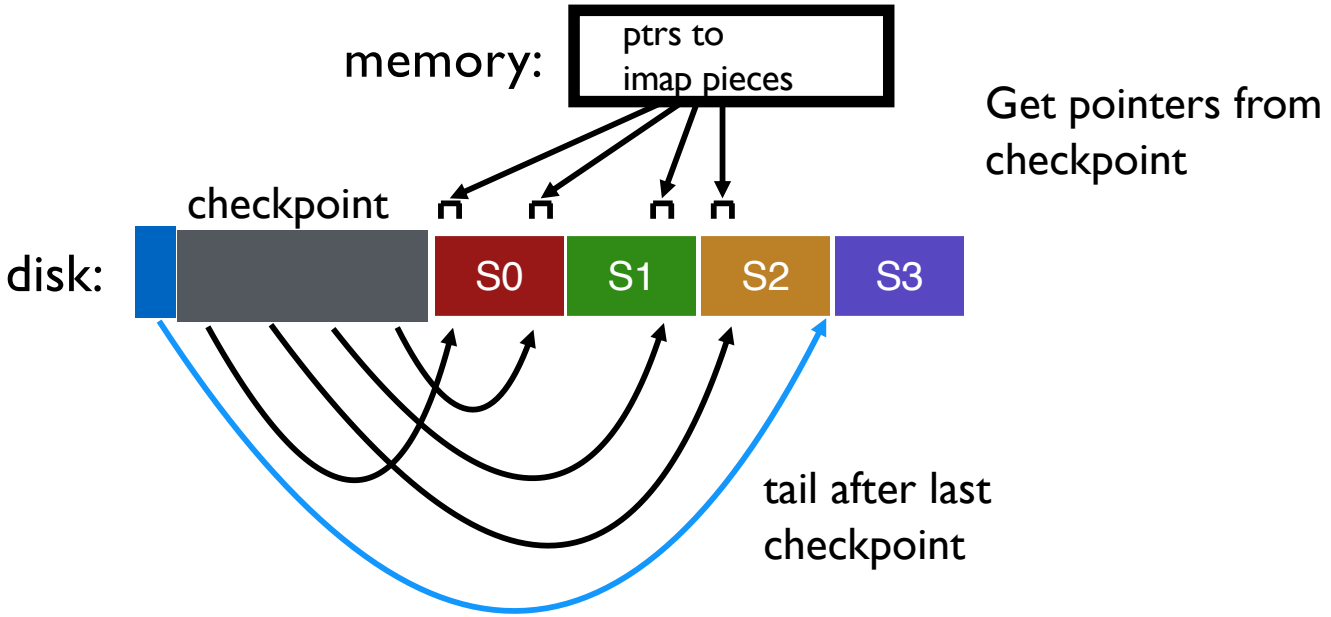
CHECKPOINT


memory:
ptrs to
imap pieces
after last
checkpoint
checkpoint
disk:
S0
S1
S2
S3
tail after last
checkpoint
Crash!

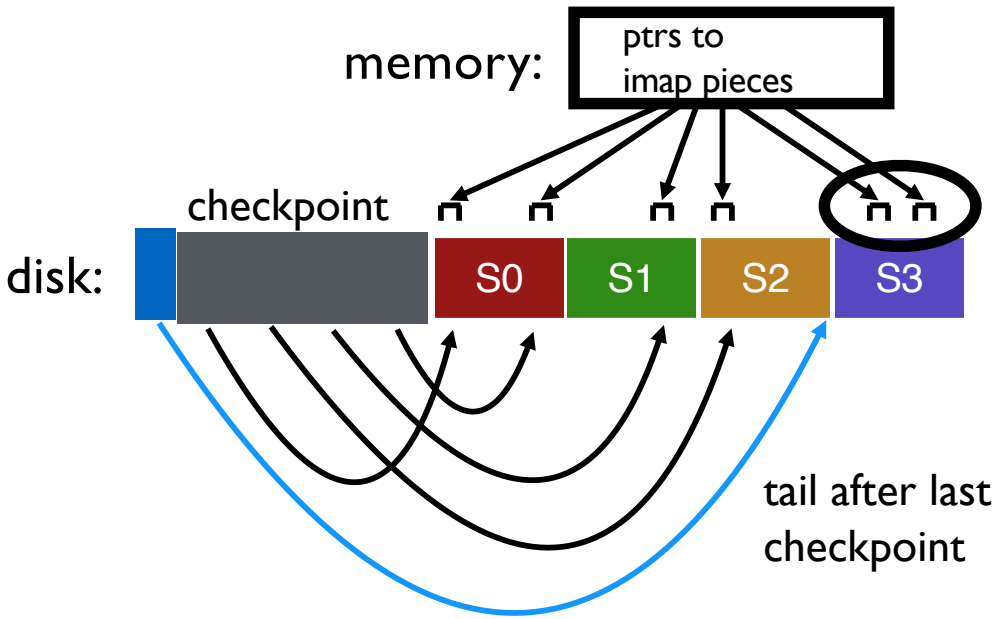
REBOOT


memory:
ptrs to
imap pieces
checkpoint
disk:
S0
S1
S2
S3
tail after last
checkpoint

REBOOT


memory:
ptrs to
imap pieces
Get pointers from
checkpoint
checkpoint
disk:
S0
S1
S2
S3
tail after last
checkpoint

REBOOT



Scan segments after tail looking for imap pieces;
→ roll forward

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

OTHER ISSUES

~~Crashes~~

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

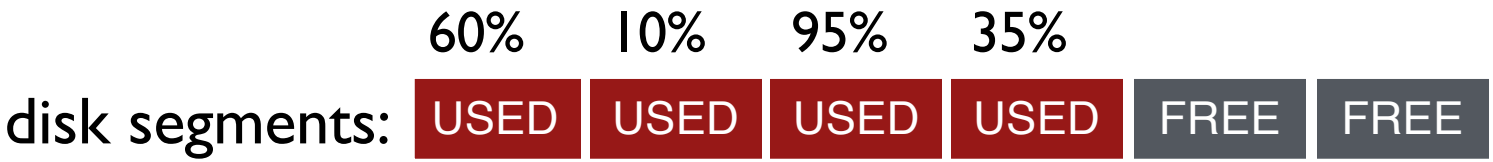
Need to reclaim space:

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

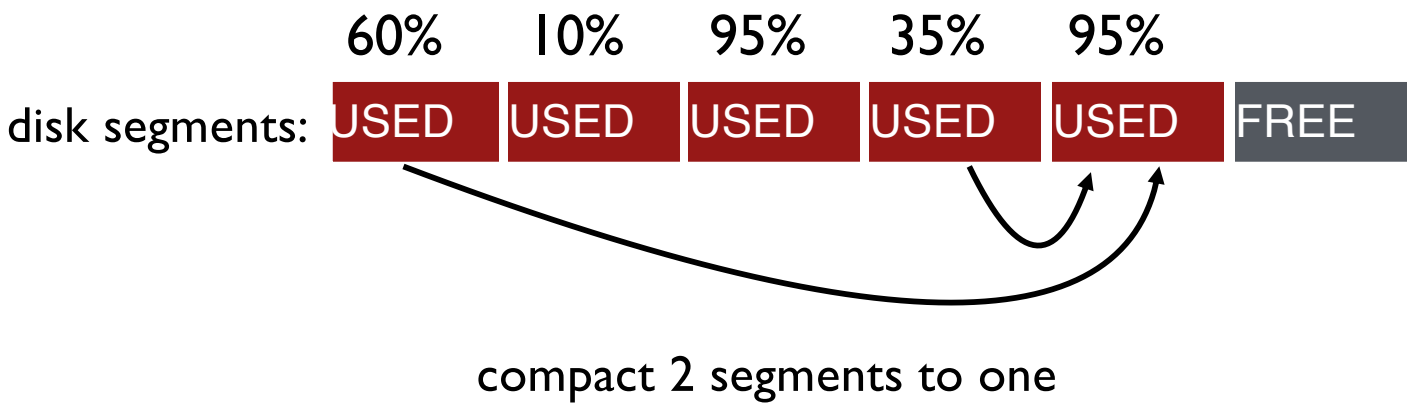
LFS reclaims **segments** (not individual inodes and data blocks)

- 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).

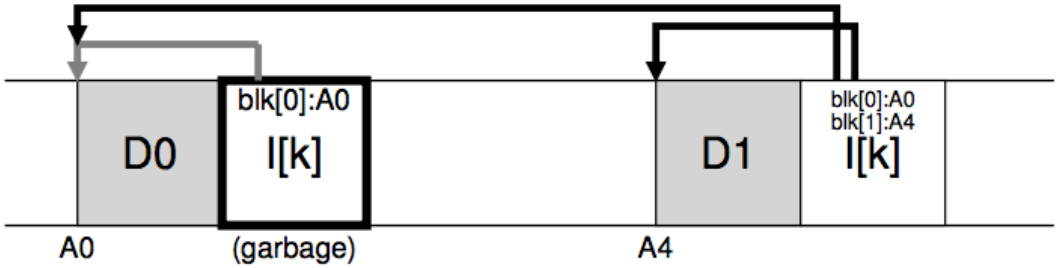
Mechanism:

How does LFS know whether data in segments is valid?

Policy:

Which segments to compact?

GARBAGE COLLECTION



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?

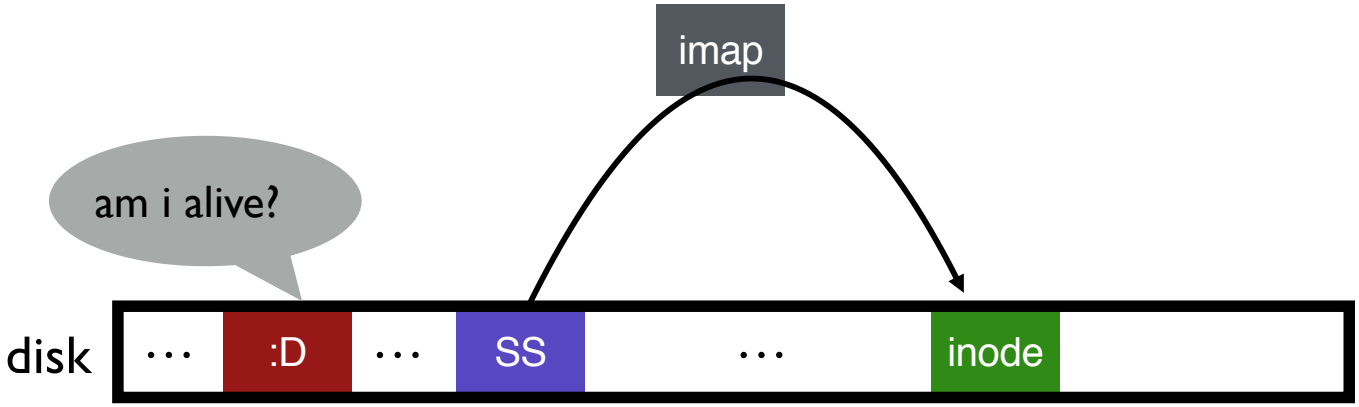
- Scan ALL inodes to see if any point to this data
- Very slow!

How to track information more efficiently?

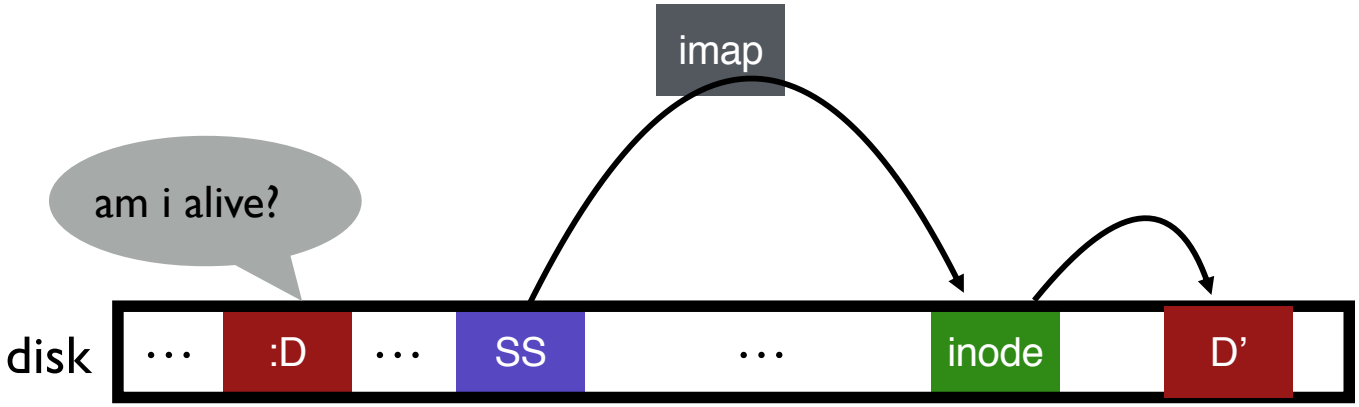
- **Segment summary** lists inode and data offset corresponding to each data block in segment (reverse pointers)

BLOCK LIVENESS

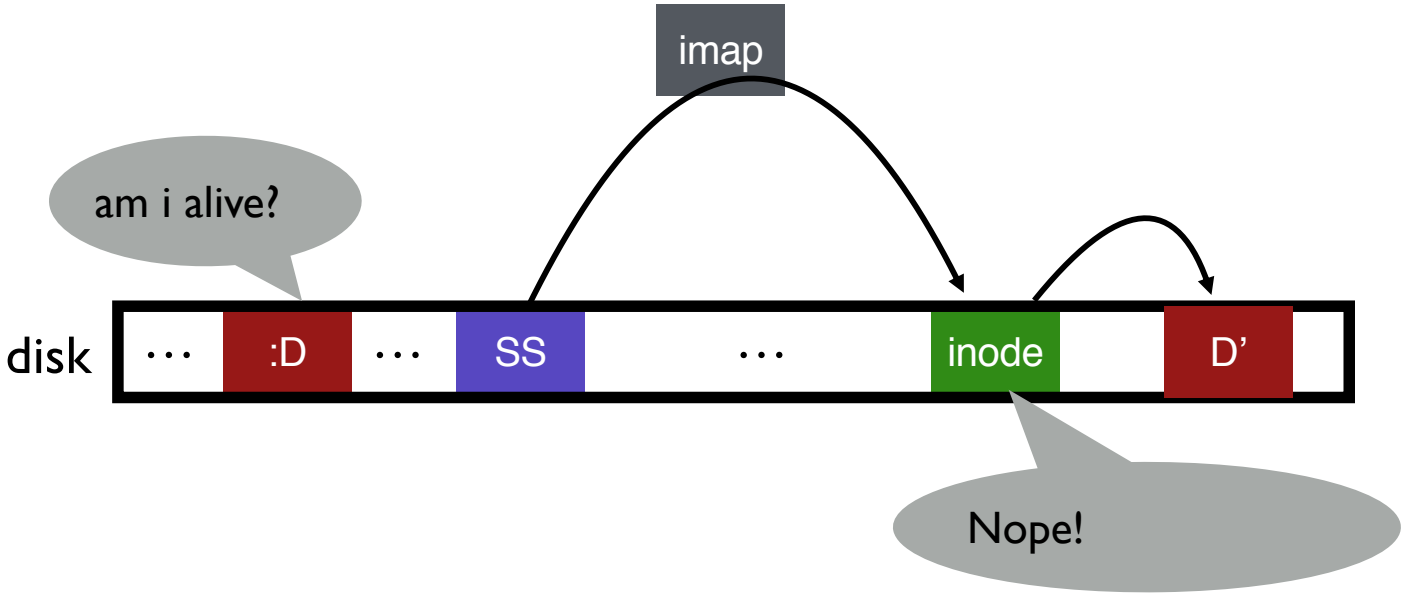
BLOCK LIVENESS



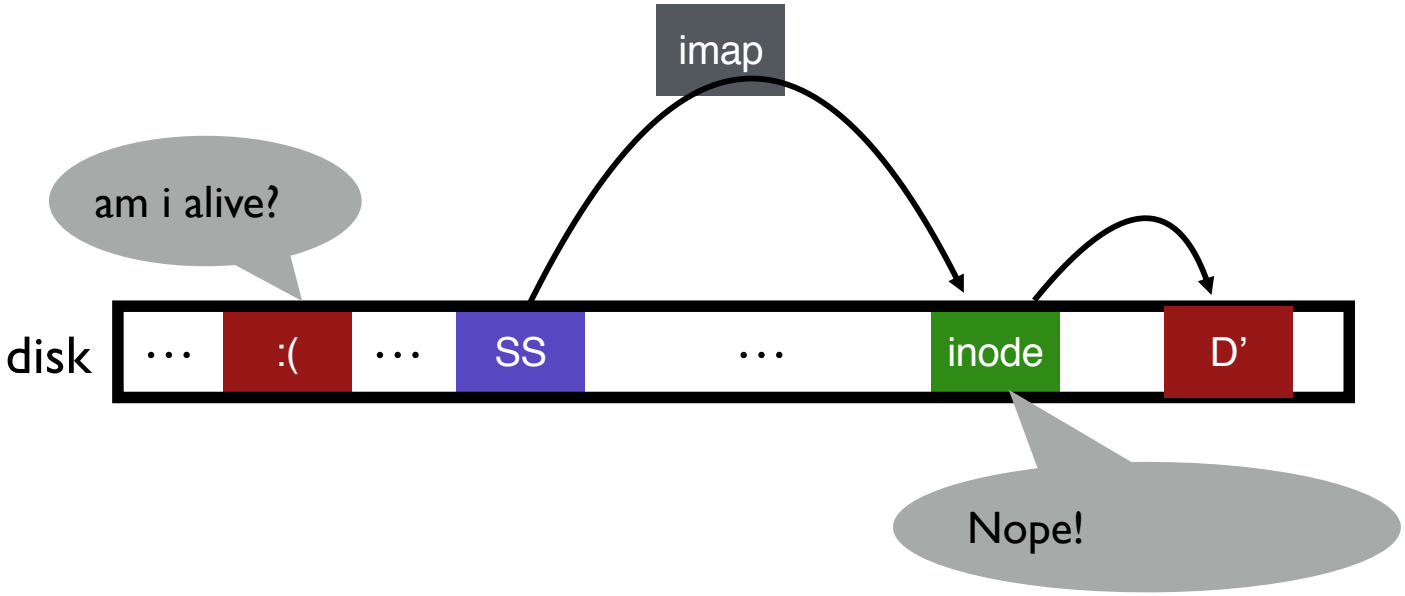
BLOCK LIVENESS



BLOCK LIVENESS



BLOCK LIVENESS



Don’t have to copy this data block to new segment; just discard

GARBAGE COLLECTION

General operation:

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

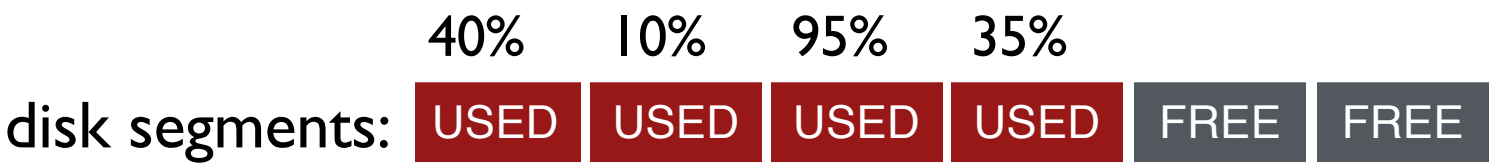
Mechanism:

Use segment summary, imap to determine liveness

Policy:

Which segments to compact?

- clean most empty first
- clean coldest (ones undergoing least change)



WHICH STYLE GIVES BETTER PERFORMANCE?

Journaling:

Put final location of data wherever file system chooses (logical locality)

(usually in a place optimized for future reads)

LFS (Other COW file systems: WAFL, ZFS, btrfs)

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

(temporal locality)

Some workloads have better read performance with LFS, some with Journaling

Which?

LFS has garbage collection costs which journaling does not…

NEXT STEPS

Enjoy Thanksgiving!

SSDs/Flash + Distributed Systems when you return