PERSISTENCE: JOURNALING, LFS

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

Project 5: Out now. Last Project!

Discussion today: Project 5

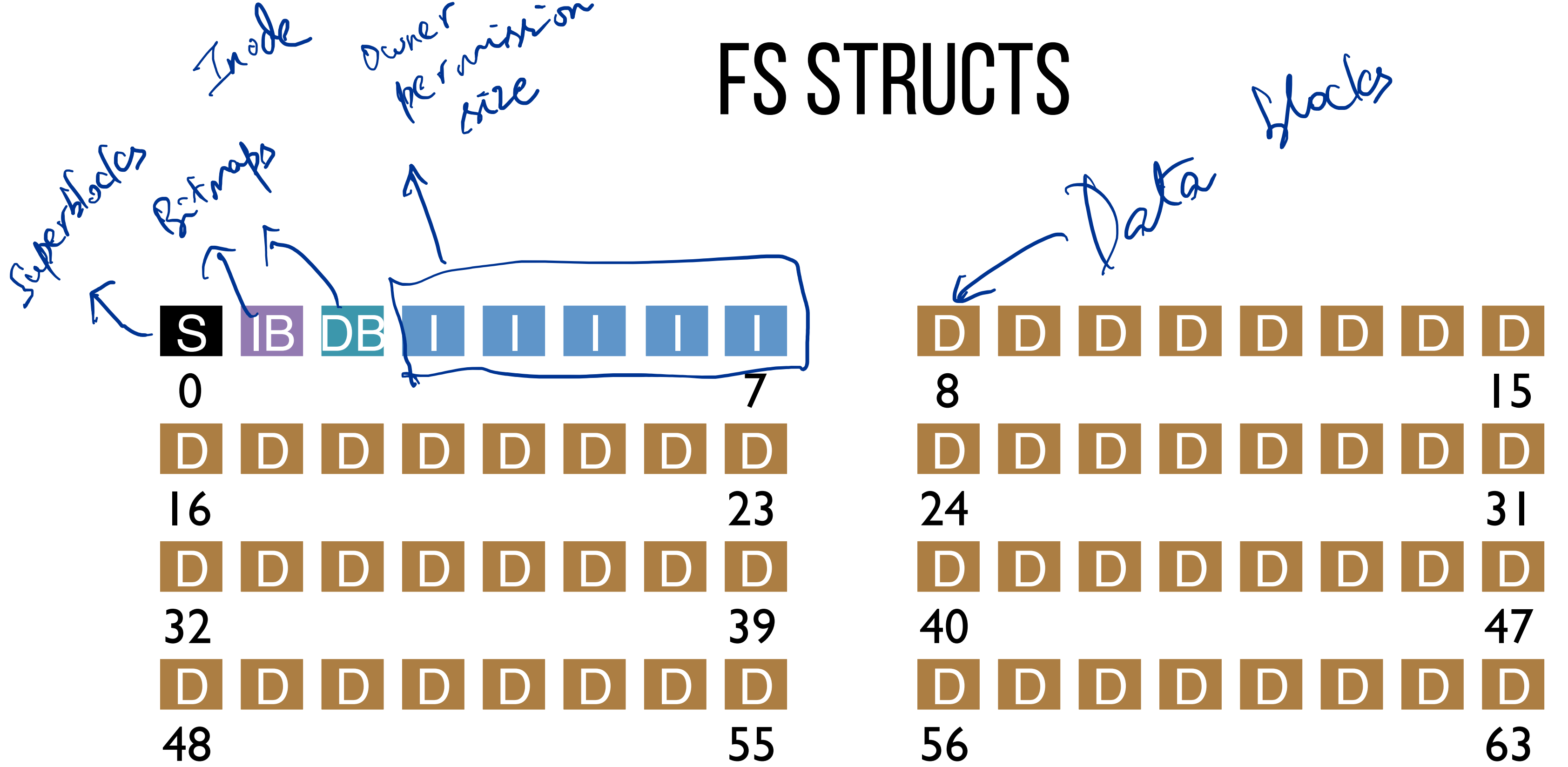
AGENDA / LEARNING OUTCOMES

How to use journaling to maintain consistency during crashes?

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

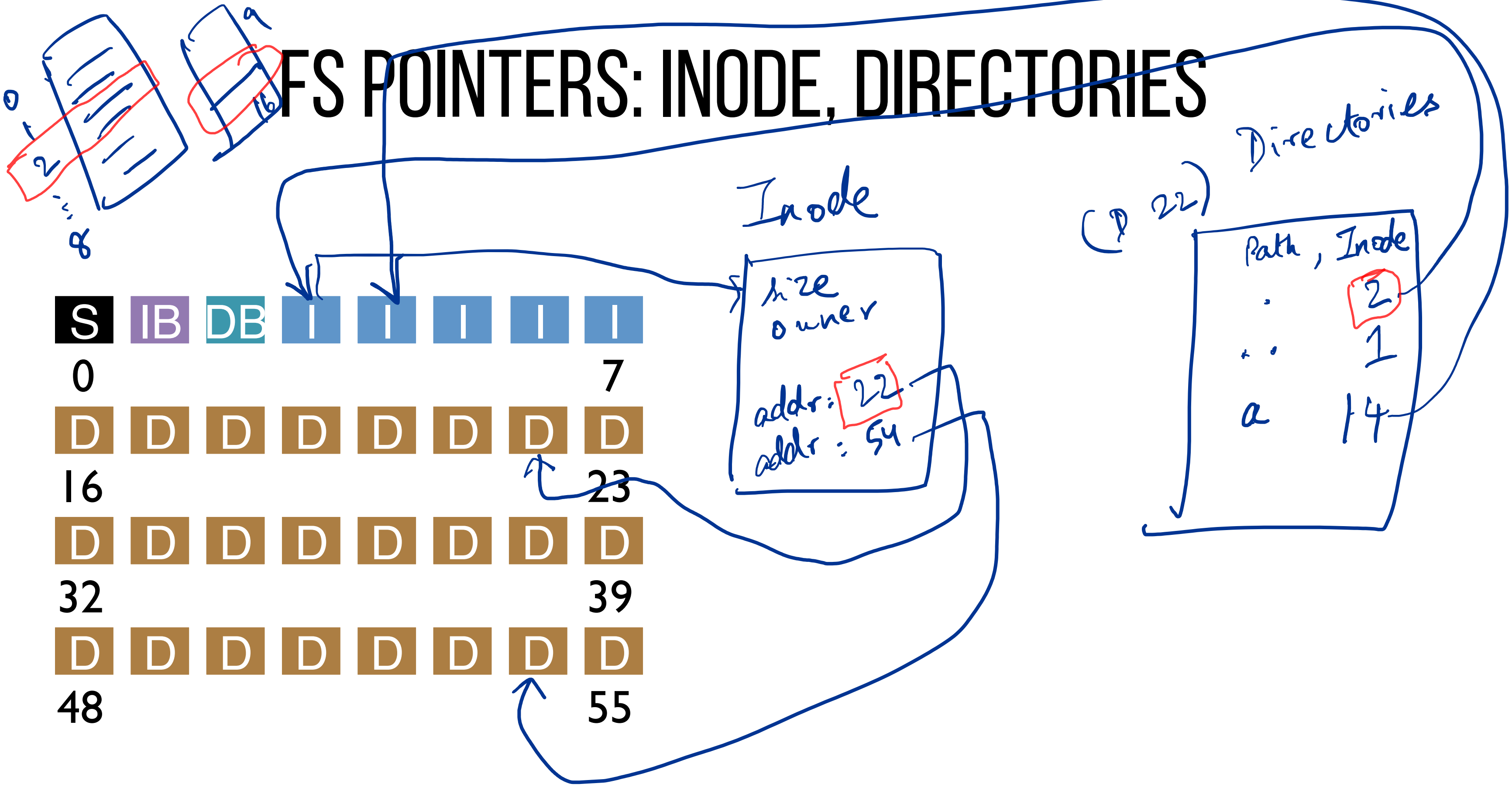
RECAP

FS STRUCTS



Inode
Owner
permission
size
Superblocks
Bitmaps
Data
Blocks
S
IB
DB
I
I
I
I
I
D
0
7
8
15
16
23
24
31
32
39
40
47
48
55
56
63

FS Pointers: Inode, Directories



FFS POLICY SUMMARY

nearby location on a disk
can avoid

File inodes: allocate in same group with dir

Dir inodes: allocate in new group with fewer used inodes than average group

First data block: allocate near inode

Other data blocks: allocate near previous block

direct

Large file data blocks: after 48KB, go to new group.

Move to another group (w/ fewer than avg blocks) every subsequent 1MB.

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 ".."?

…

bitmaps, inode

dir 1

link = 2

dir2

copy block

inode points to invalid block

BUNNY 18

https://tinyurl.com/cs537-sp19-bunny18

BUNNY 18

https://tinyurl.com/cs537-sp19-bunny18

```
Inode Bitmap : 10000000
Inode Table  : [size=1,ptr=0,type=d] [] [] [] [] [] [] []
Data Bitmap  : 10000000
Data         : [("." 0), (".." 0)] [] [] [] [] [] [] []
```

There are only eight inodes and eight data blocks; each of these is managed by a corresponding bitmap. The inode table shows the contents of each of eight inodes, with an individual inode enclosed between square brackets; in the initial state above, only inode 0 is in use. When an inode is used, its size and pointer field are updated accordingly (in this question, files can only be one block in size; hence a single inode pointer); when an inode is free, it is marked with a pair of empty brackets like these "[]". Note there are only two file types: directories (type=d) and regular files (type=r). Data blocks are either "in use" and filled with something, or "free" and marked accordingly with "[]". Directory contents are shown in data blocks as comma-separated lists of tuples like: ("name", inode number). The root inode number is zero.

(a) INITIAL STATE: state(i) as above to FINAL STATE (a):

0, 2

```
Inode Bitmap : 11000000
Inode Table  : [size=1,ptr=0,type=d] [size=0,ptr=-,type=r] [] [] [] [] [] []
Data Bitmap  : 10000000
Data         : [("." 0),(".." 0),("f" 1)] [] [] [] [] [] [] []
```

DATA

Creates a empty file

/f

creat(/f)

Operation that caused this change?

BUNNY 18

fsck → Cannot call FS operations?

(f) FILE SYSTEM STATE: Consistent or inconsistent? If inconsistent, how to fix?

```
Inode Bitmap  : 11100000
Inode Table   : [size=1,ptr=0,type=d] [size=1,ptr=1,type=r] [size=1,ptr=2,type=r] [] [] [] [] []
Data Bitmap   : 11100000
Data          : [(".." 0),(".." 0)] [DATA] [DATA] [] [] [] [] []
```

Not consistent.

Inode 1, 2 are not pointed to by any dir.

Put them

/lost+found/0001

Delete the inode 1, 2
data 1, 2
update bitmaps

CONSISTENCY SOLUTION #2: JOURNALING

Goals

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

Recover

A or B

X. Crash

$A \xrightarrow{op} B$

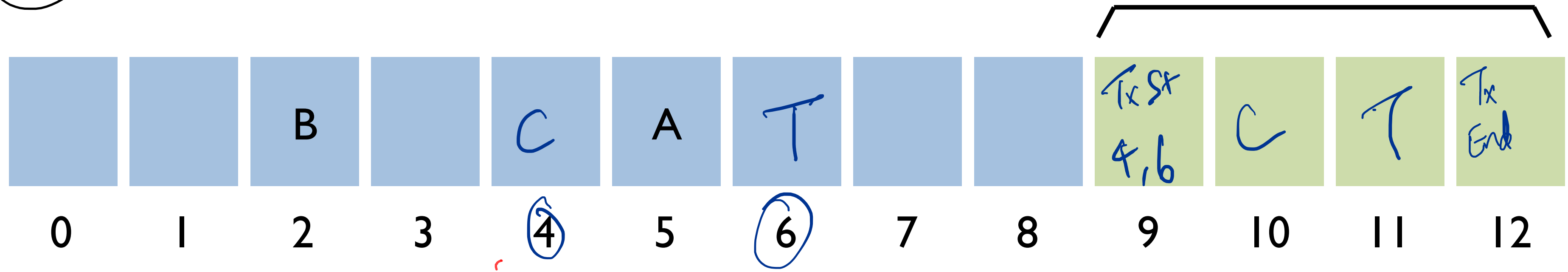
Atomicity

- 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

ORDERING FOR CONSISTENCY

A → B

Journal → log



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

Barrier

before appear

after

write order →

Correctness

Parallelism

Tx Start which blocks

4, 6

Tx End

Write to blk 4 6

Tx Valid - 0

Journal

rollback A

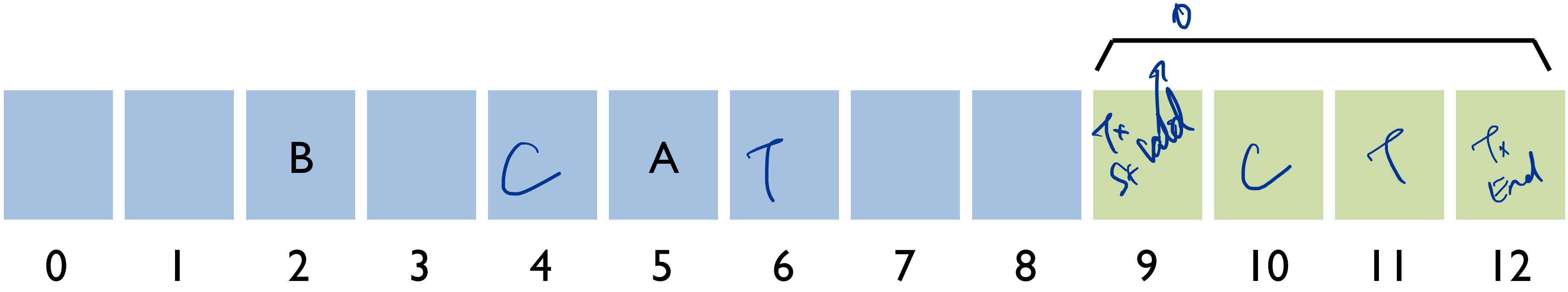
rollback A

replay B

checkpointing

Journal can be reused

ORDERING FOR CONSISTENCY



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

write order

9,10,11

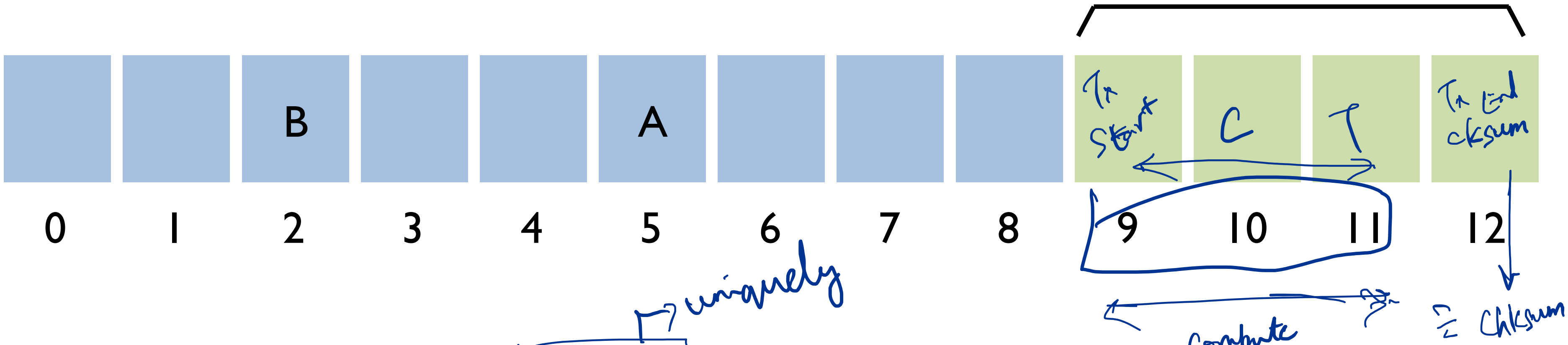
12

4,6

12

CHECKSUM OPTIMIZATION

Can we get rid of barrier between (9, 10, 11) and 12 ?



In last transaction block, store checksum of rest of transaction

SHA or hash

During recovery: If checksum does not match, treat as not valid

Compute Chk sum

write order before

9,10,11

12

4,6

12

OTHER OPTIMIZATIONS

wait for timer (5s)

Batched updates

- If two files are created, inode bitmap, inode etc. get written twice
- Mark as dirty in-memory and batch updates

Batch all ops into 1 transaction

Bitmap once

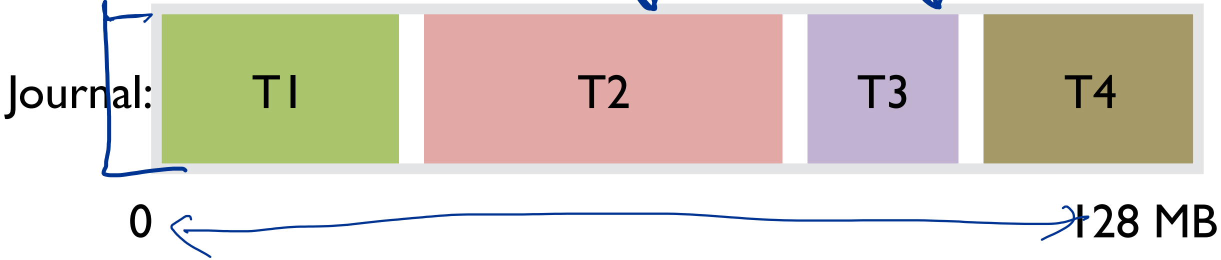
Inode once

Circular log

first

last

→ Write Txn



Linked list

Add

start

last

Checkpt

finish

head

more

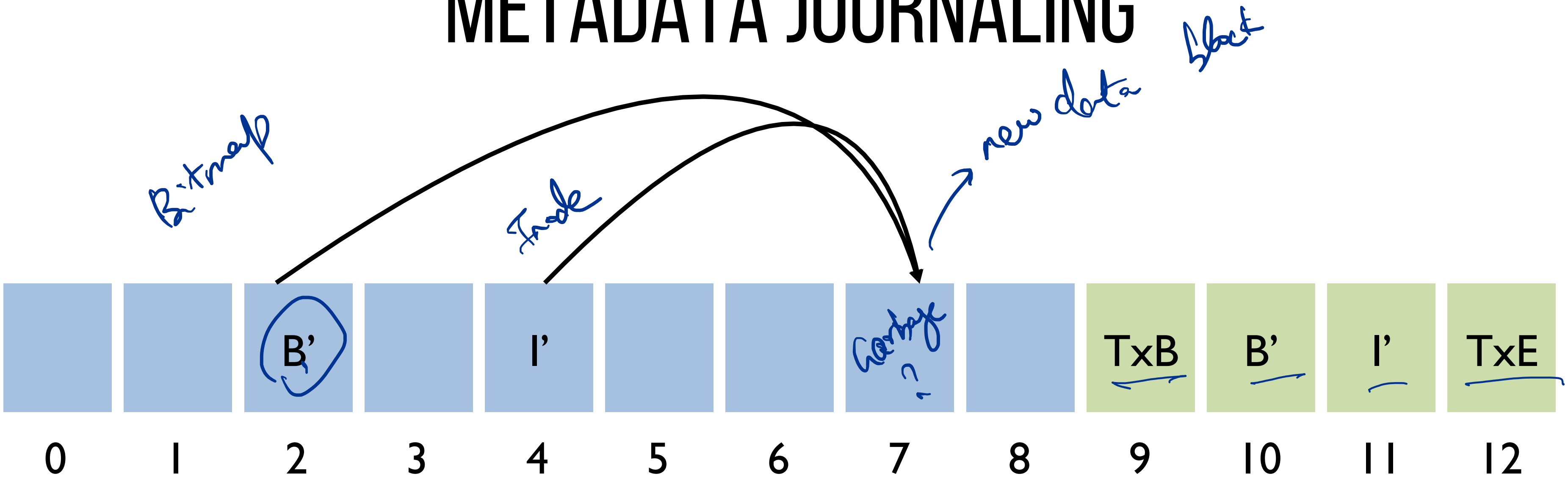
HOW TO AVOID WRITING ALL DISK BLOCKS TWICE?

Observation: Most of writes are user data (esp sequential writes)

Strategy: journal all metadata, including
superblock, bitmaps, inodes, indirects, directories

For regular data, write it back whenever convenient.

METADATA JOURNALING



transaction: append to inode I

Crash !?!

B' in 2

I' in 4 Crash

D in 7

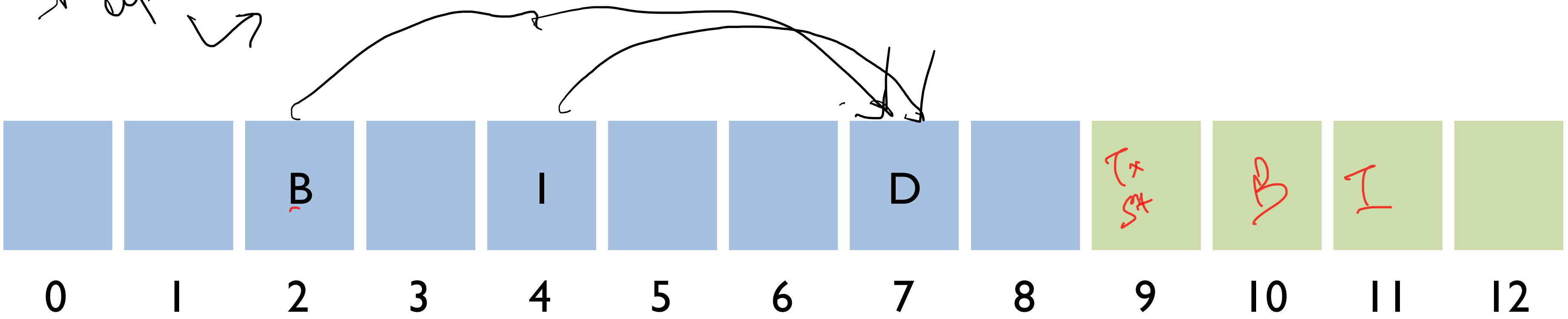
ORDERED JOURNALING

Still only journal metadata

But write data **before** the transaction!

ORDERED JOURNAL

In place updates



Write D to 7

Crash here?

Write Txn St B I to Journal

Write Txn End

Write B, I to 2, 4.

What happens if crash now?
B indicates D currently free, I does not point to D;
Lose D, but that might be acceptable

Can remove this barrier!

SUMMARY

Crash consistency: Important problem in filesystem design!

Two main approaches

FSCK: → checker on recovery

Fix file system image after crash happens

Too slow and only ensures consistency

Journaling

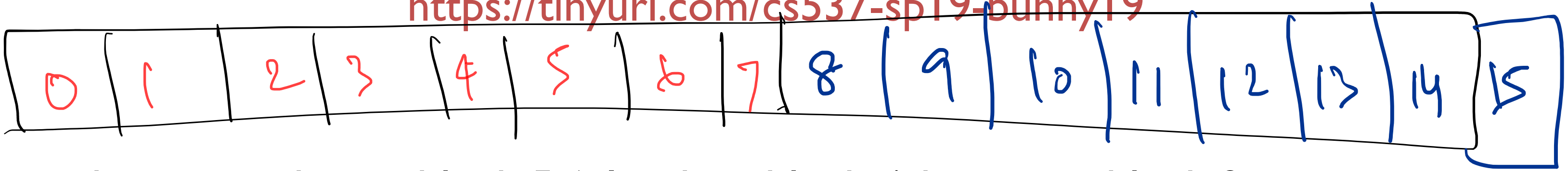
Write a transaction before in-place updates

Checksum, batching

→ Ordered journal avoids data writes

BUNNY 19: IDENTIFY THE KIND OF JOURNALING

https://tinyurl.com/cs537-sp19-bunny19



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

1

Write 5,6 → Data
Write 8, 9, 10 → Txn St entries
Barrier
Write 11
Barrier
Write 4, 2 → Only metadata

Ordered journal

2

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

Data Journaling

3

Tx End

Write 8, 9, 10, 11, 12, $\boxed{13}$
Barrier
Write 2,4,5,6 ← Checkpointing

Checksum Optimization

LOG STRUCTURED FILE SYSTEM (LFS)

LFS PERFORMANCE GOAL

Motivation:

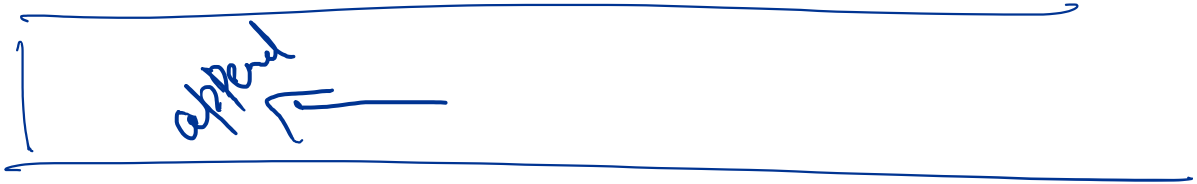
- Growing gap between sequential and random I/O performance
- RAID-5 especially bad with small random writes

RAID-4

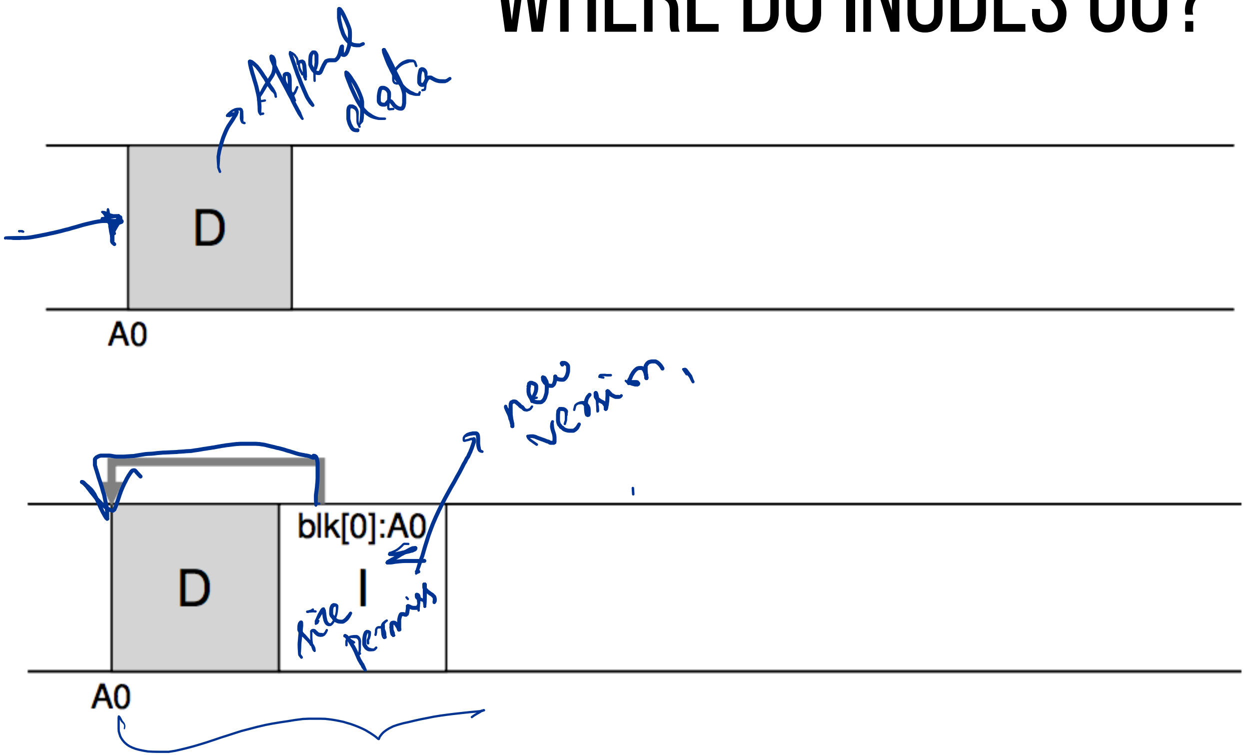
Idea: use disk purely sequentially

Design for writes to use disk sequentially – how?

log



WHERE DO INODES GO?



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)

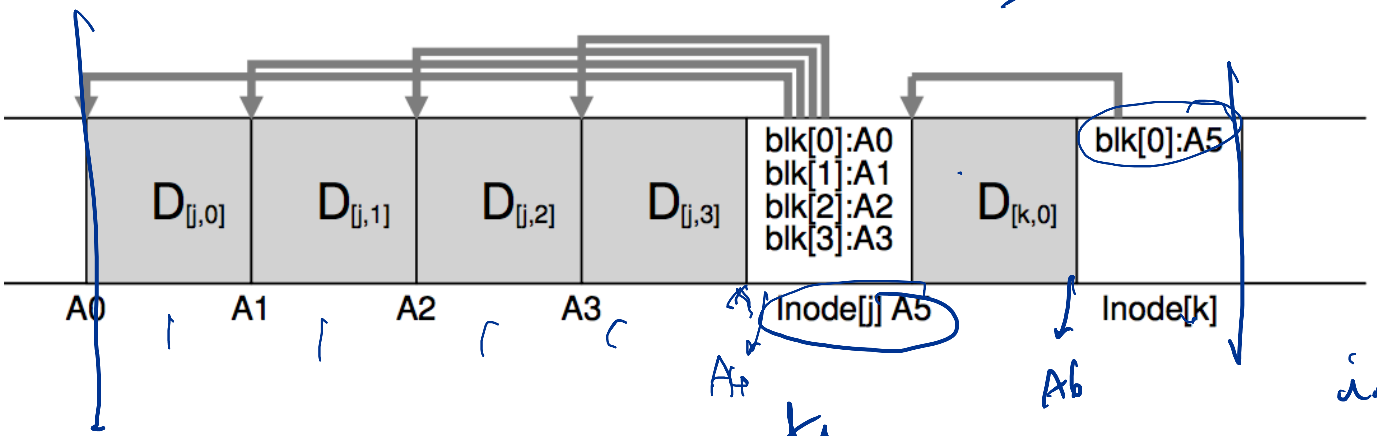
Write buffered data sequentially to new **segment** on disk

→ chunk or a large buffer of data appended to the log

Never overwrite old info: old copies left behind

BUFFERED WRITES

four data blocks in 1 data block to k



segments

imap

j → A4

k → A6

WHAT ELSE IS DIFFERENT FROM FFS?

read /f

What data structures has LFS removed?

allocation structs: data + inode bitmaps

Traverse look up inode for /

data /

inode /f

data /f

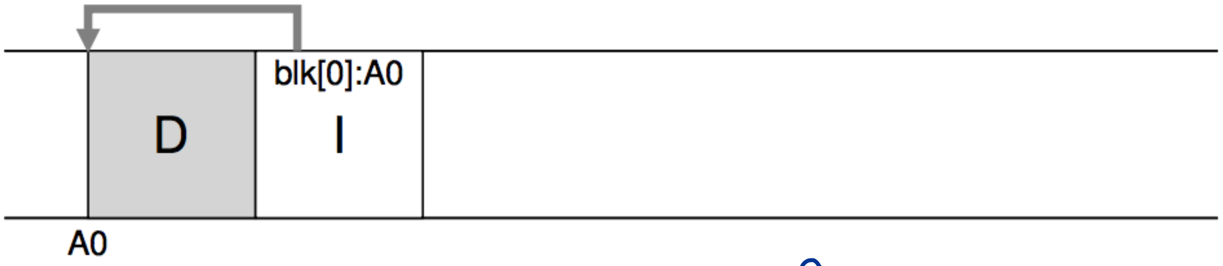
How to do reads?

Inodes are no longer at fixed offset

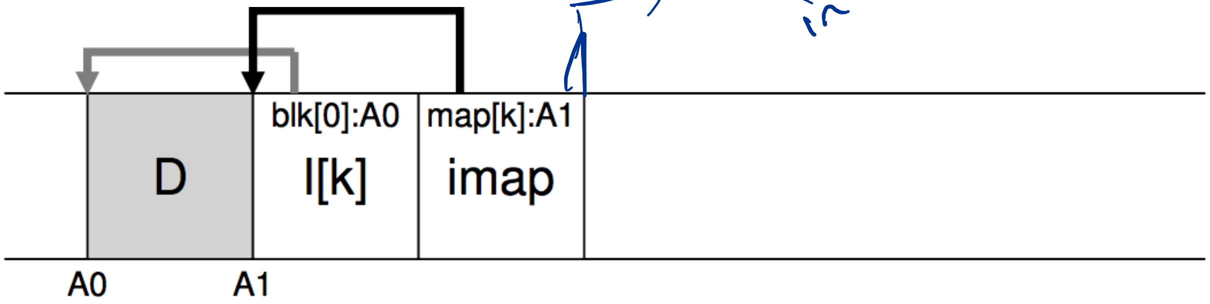
Use imap structure to map: inode map

inode number => inode location on disk

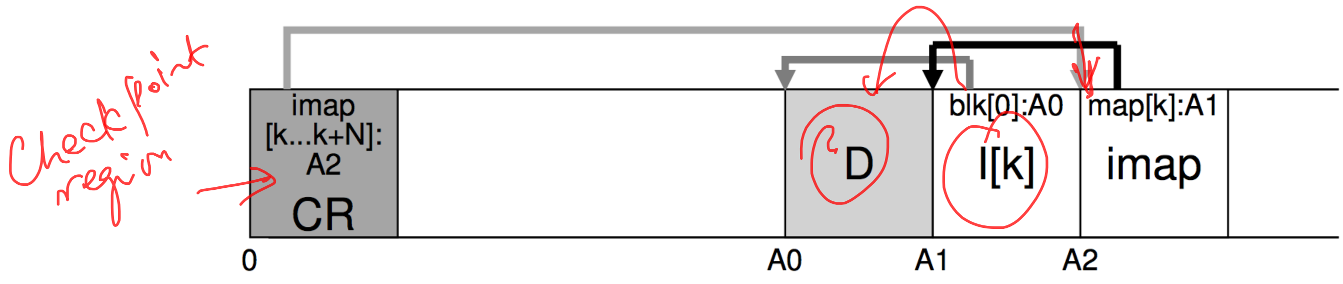
IMAP EXPLAINED



parts of imap in the log



READING IN LFS



memory

superblock

imap

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

list of imap ptrs

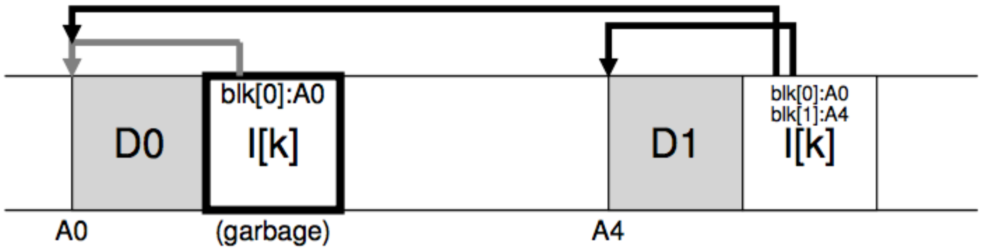
lookup inode for / → A1

read A1 addr: A0

read A0

Similar to FFS

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)

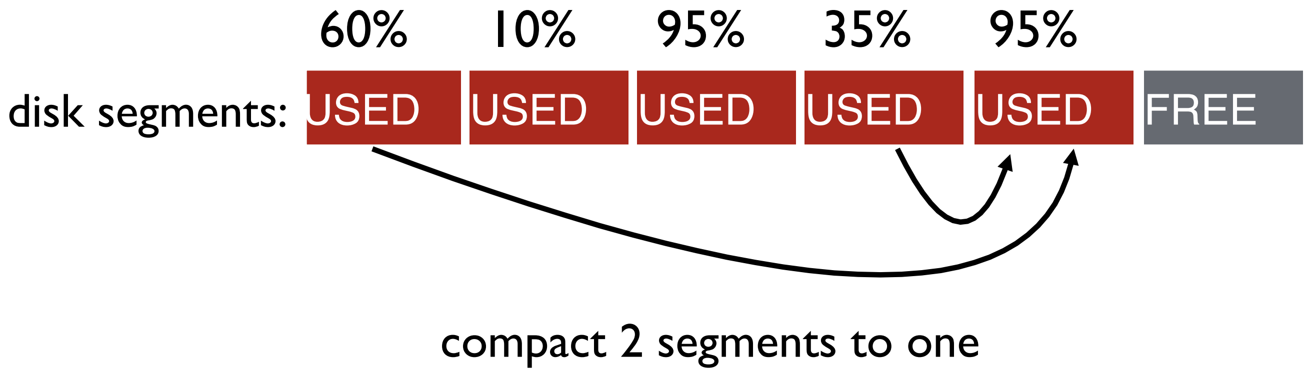
LFS reclaims **segments** (not individual inodes and data blocks)

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

GARBAGE COLLECTION

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

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 MECHANISM

Is an inode the latest version?

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

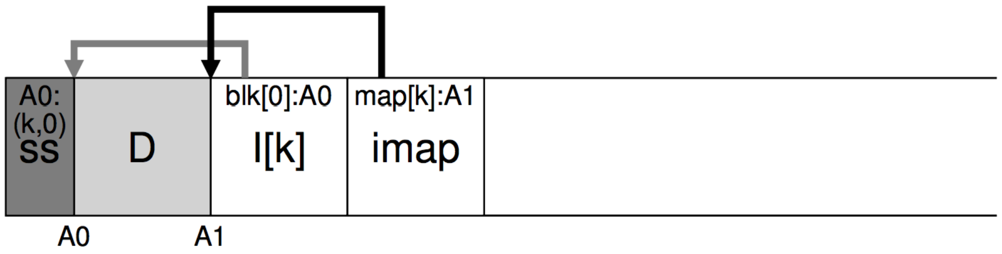
Is a data block the latest version?

- 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




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

inode = Read(imap[N]);

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

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)

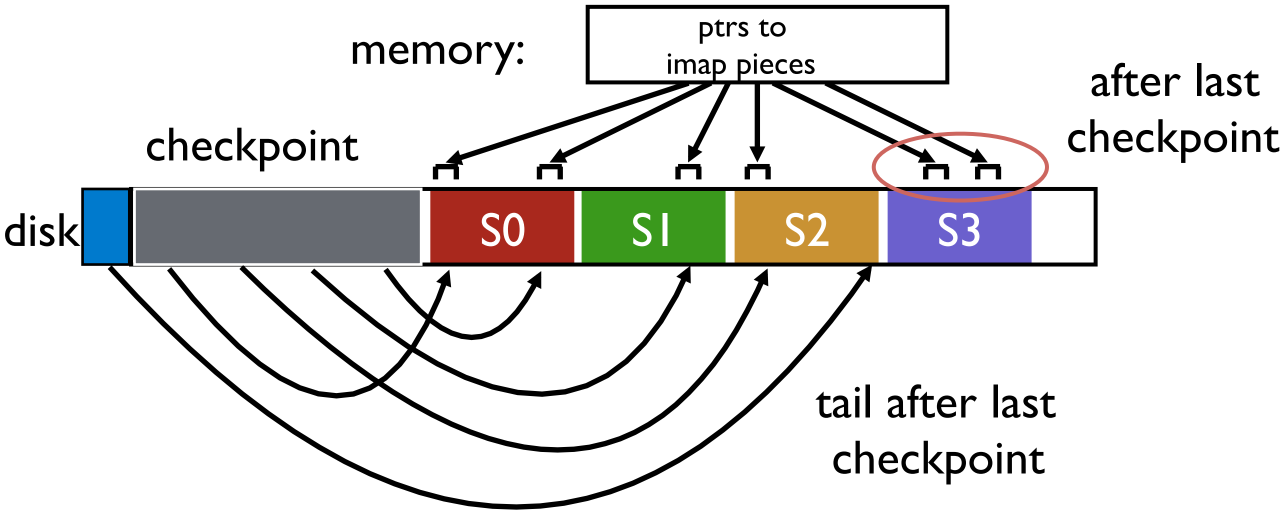
Better approach?

- 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 <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: Distributed systems

Project 5 is out!

Discussion: Project 5 walkthrough