

# ANNOUNCEMENTS

P4 graded: In Learn@UW; email 537-help@cs if problems

P5: Available - File systems

- Can work on both parts with project partner
- Watch videos; discussion section
- Part a : file system checker NOT in xv6 code base

Read as we go along!

- Chapter 43

UNIVERSITY of WISCONSIN-MADISON  
Computer Sciences Department

CS 537  
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau  
Remzi H. Arpaci-Dusseau

## PERSISTENCE: LOG-STRUCTURED FS (LFS)

### Questions answered in this lecture:

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?

How to **garbage collect** dead information?

## 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

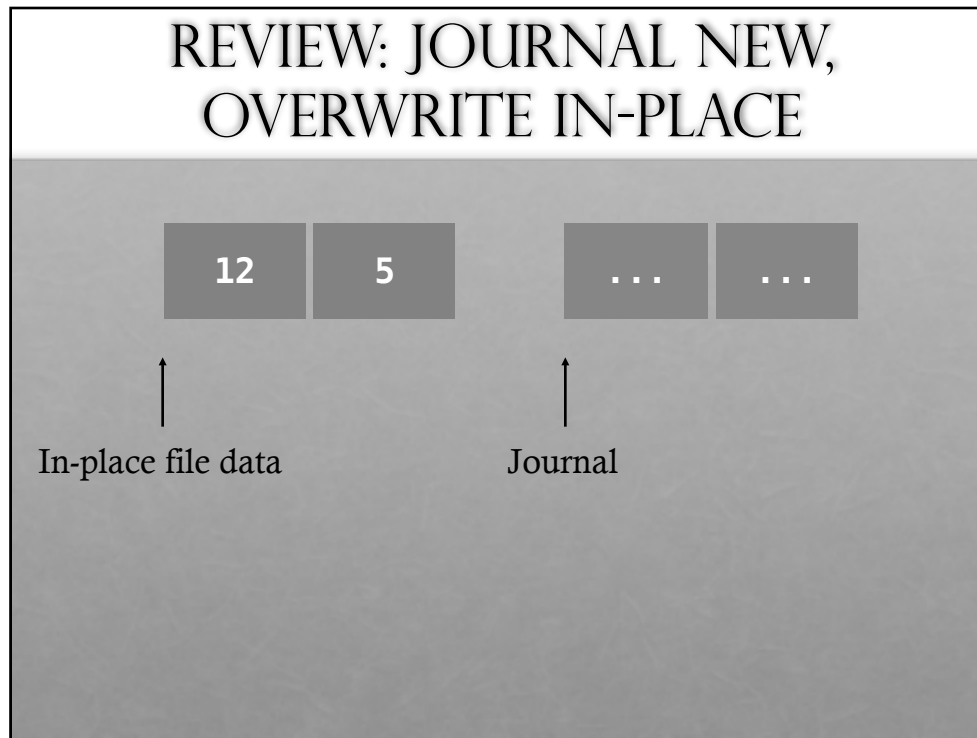
Implication:

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

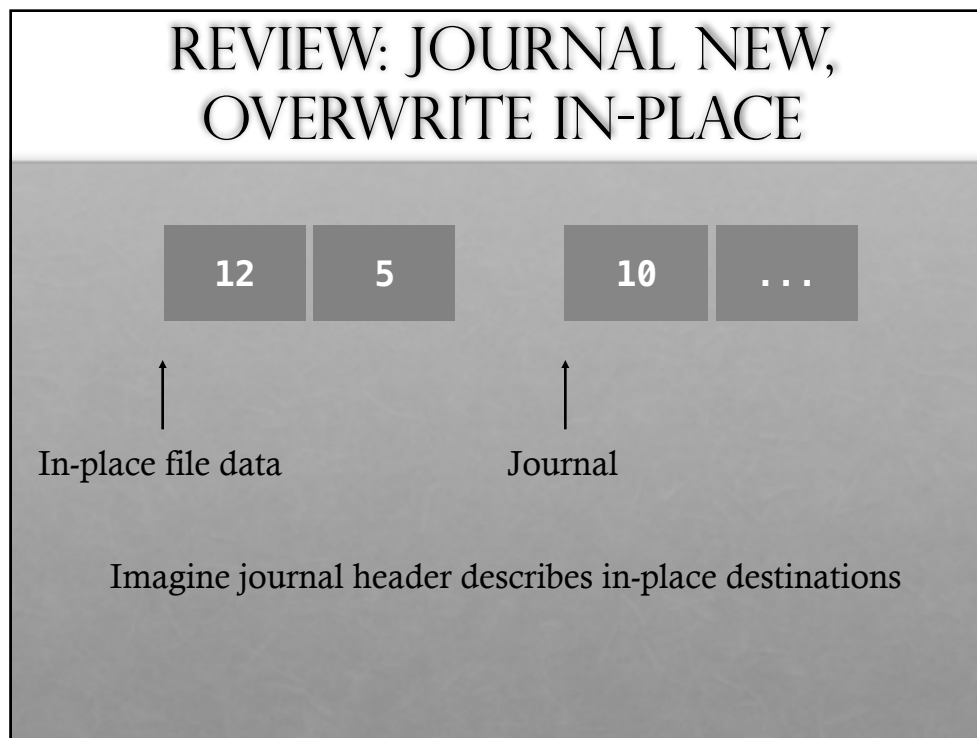
Two techniques popular in file systems:

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)

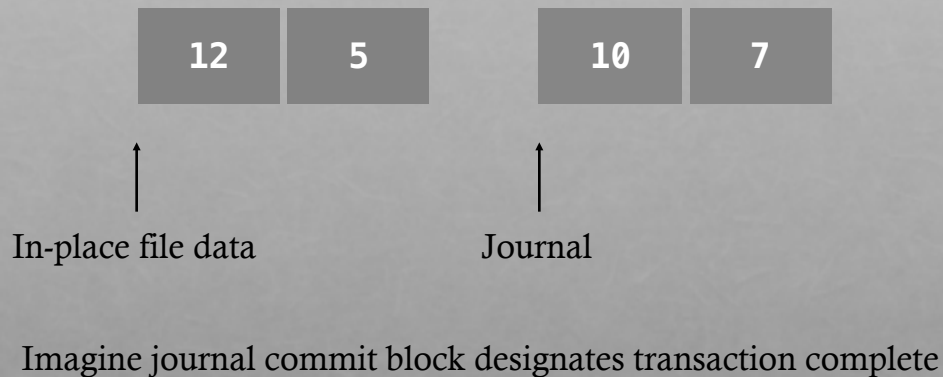
## REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE



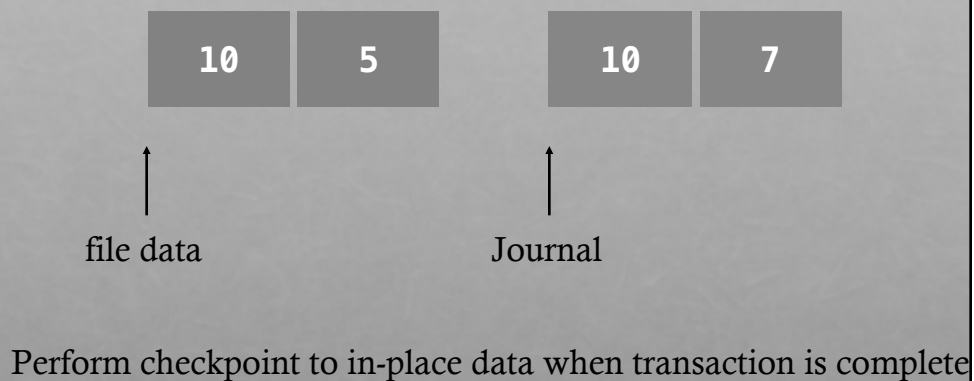
## REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE



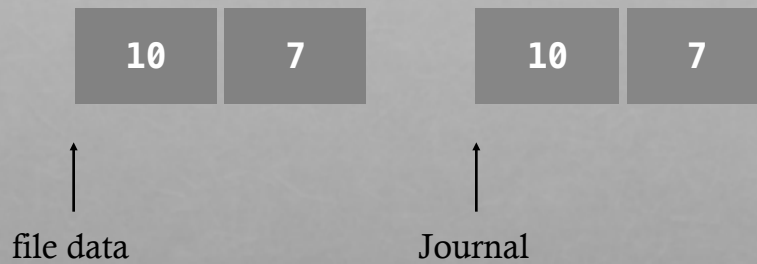
## REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE



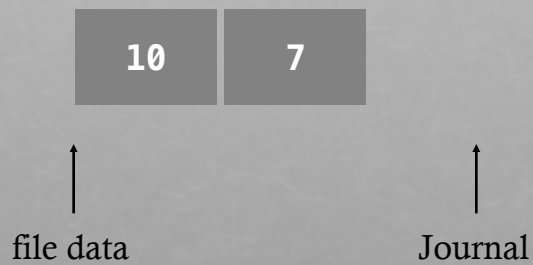
## REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE



## REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE



## REVIEW: JOURNAL NEW, OVERWRITE IN-PLACE

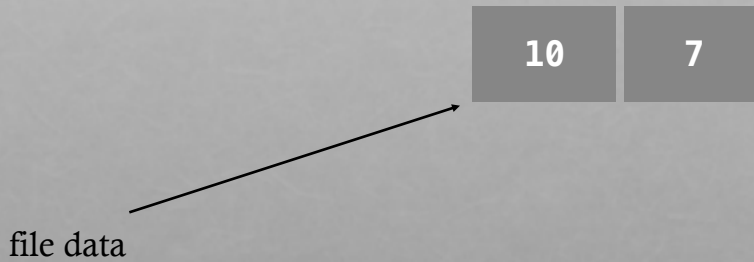


Clear journal commit block to show checkpoint complete





## TODAY: WRITE NEW, DISCARD OLD



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

Easy 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 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

- Segment is some contiguous region of blocks


Never overwrite old info: old copies left behind

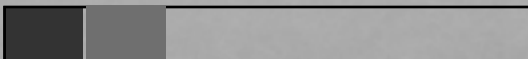
## BIG PICTURE

buffer: 


disk: 

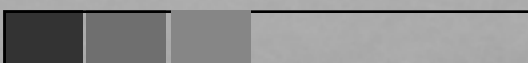
## BIG PICTURE

buffer: 


disk: 

## BIG PICTURE

buffer: 


disk: 

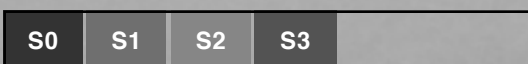
## BIG PICTURE

buffer: 

disk: 

## BIG PICTURE

buffer: 

disk:   
segments

# DATA STRUCTURES (ATTEMPT 1)



What data structures from FFS can LFS remove?

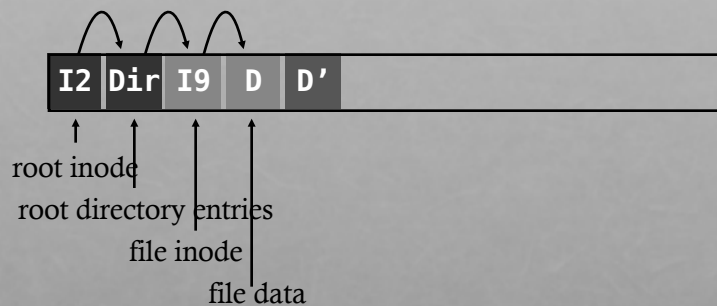
- allocation structs: data + inode bitmaps

What type of name is much more complicated?

- Inodes are no longer at fixed offset
- Use **current offset on disk** instead of table index for name
- Note: when update inode, inode name changes!!

## ATTEMPT 1

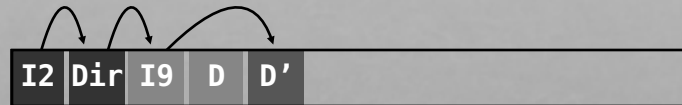
Overwrite data in /file.txt



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

# ATTEMPT 1

Overwrite data in /file.txt

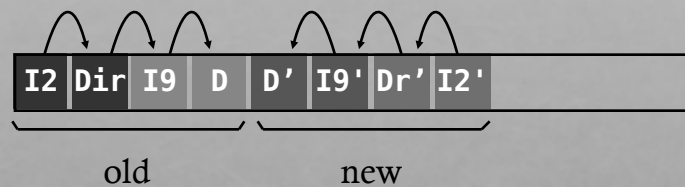


Can LFS update Inode 9 to point to new D'?

NO! This would be a random write

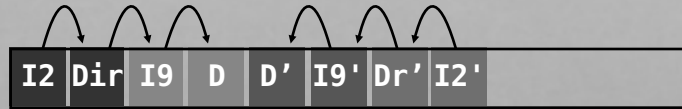
# ATTEMPT 1

Overwrite data in /file.txt



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)

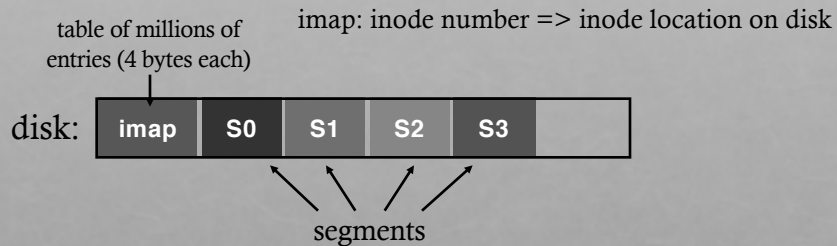
What data structures from FFS can LFS remove?

- allocation structs: data + inode bitmaps

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?



Where can imap be stored???? Dilemma:

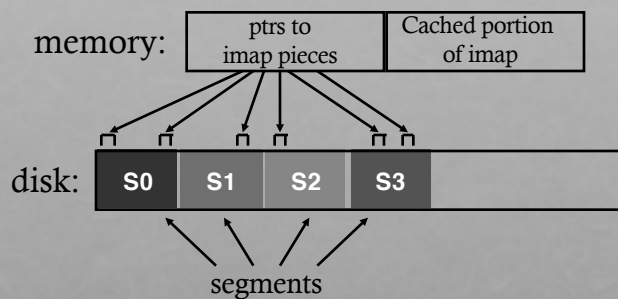
1. imap too large to keep in memory
2. don't want to perform random writes for imap

Solution:

Write imap in segments

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

## SOLUTION: IMAP IN SEGMENTS



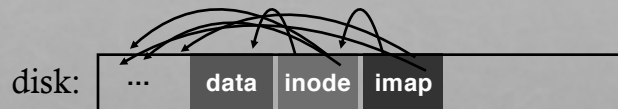
Solution:

Write imap in segments

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

Keep recent accesses to imap cached in memory

## EXAMPLE WRITE



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

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		(read)			(read)	
			(read)			(read)
	<del>read</del> write					
				<del>read</del> write		
			write			
						write

Most data structures same in LFS as FFS!

Use imap in memory to find location of root and foo inodes  
Update imap on disk with new locations for foo and bar inodes



## 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!

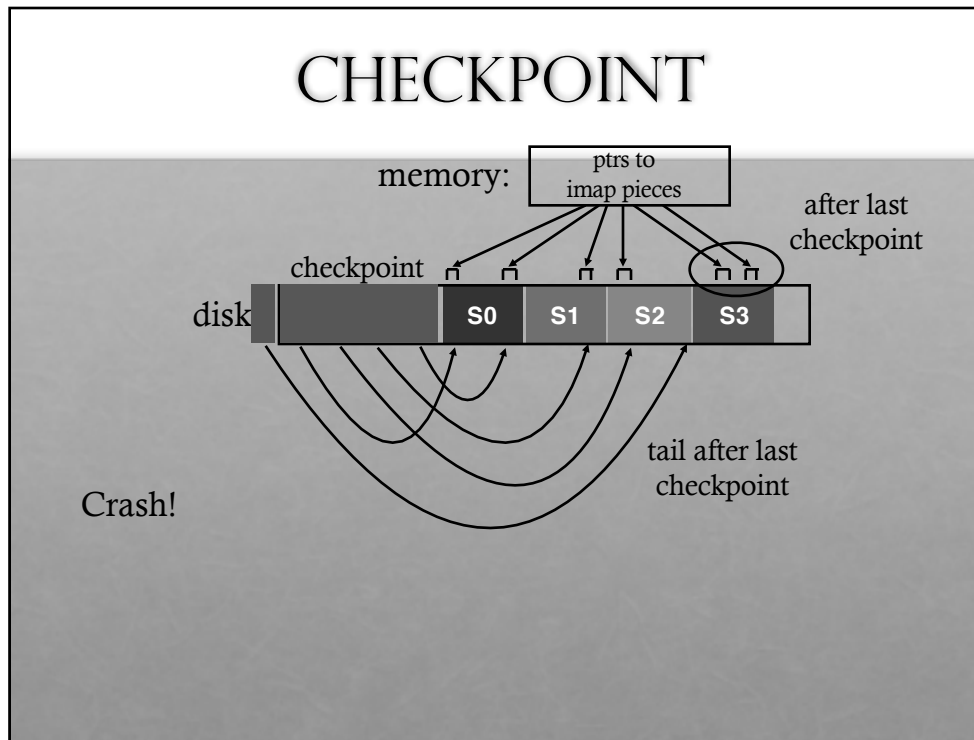
Better approach?

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

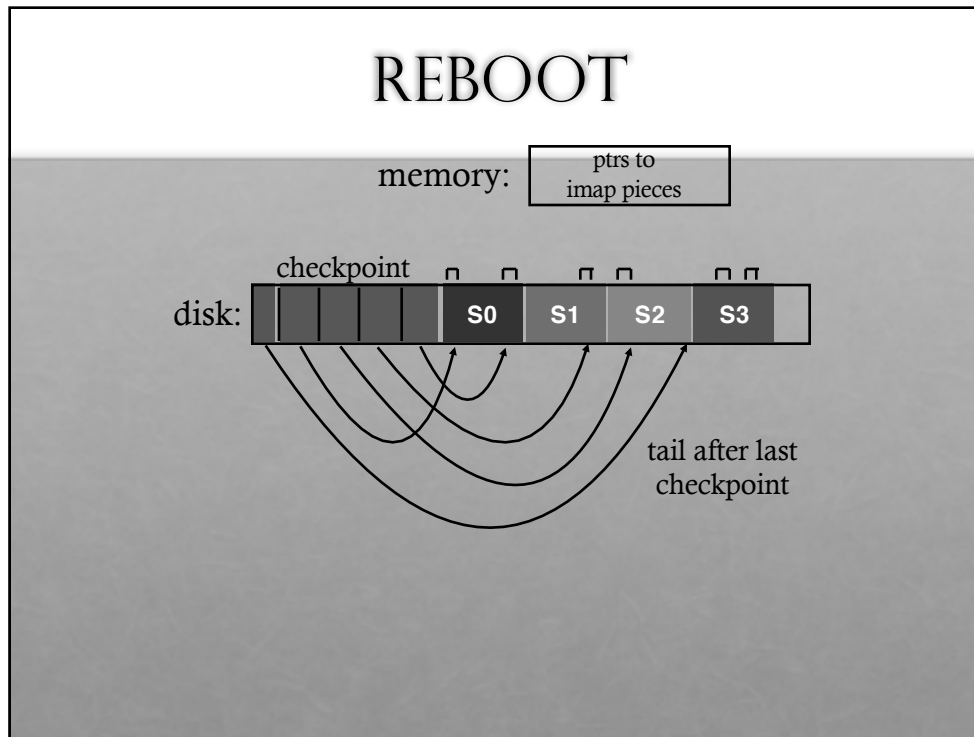
How often to checkpoint?

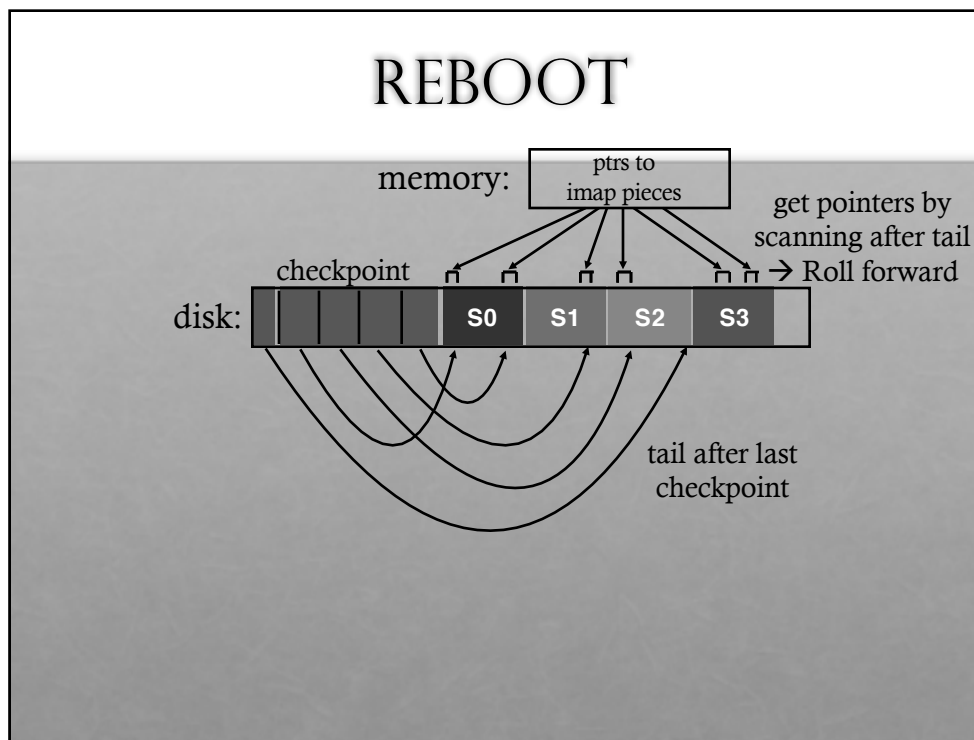
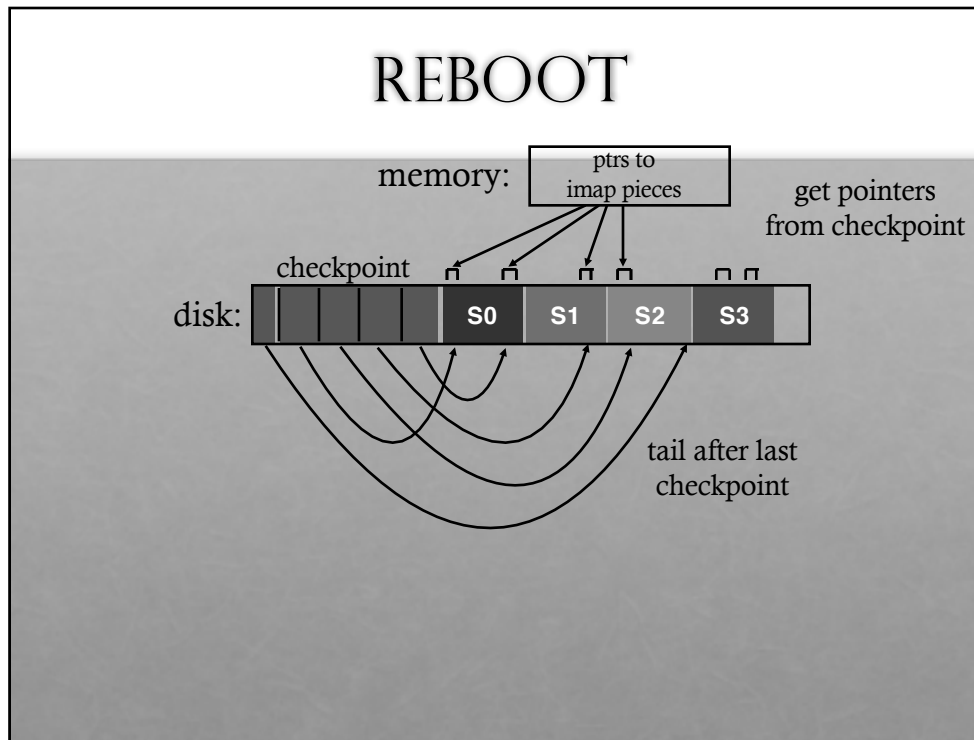
- Checkpoint often: random I/O
- Checkpoint rarely: lose more data, recovery takes longer
- Example: checkpoint every 30 secs

# CHECKPOINT



# REBOOT





## 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

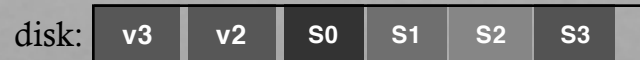


## CHECKPOINT STRATEGY

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint



## CHECKPOINT STRATEGY

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

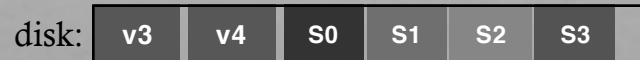


## CHECKPOINT STRATEGY

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

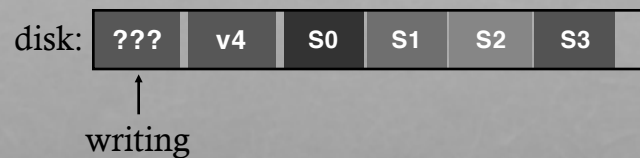


## CHECKPOINT STRATEGY

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint



## CHECKPOINT STRATEGY

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

disk: 

v5	v4	S0	S1	S2	S3	
----	----	----	----	----	----	--

## 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 overwrites 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

60% 10% 95% 35% 95%

disk segments: **USED** **USED** **USED** **USED** **USED** **FREE**

compact 2 segments to one

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

# GARBAGE COLLECTION

disk segments: 

10%	95%	95%
FREE	USED	USED
FREE	USED	FREE

release input segments

# 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 location is pointed to
- Fast!

Is a data block the latest version?

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

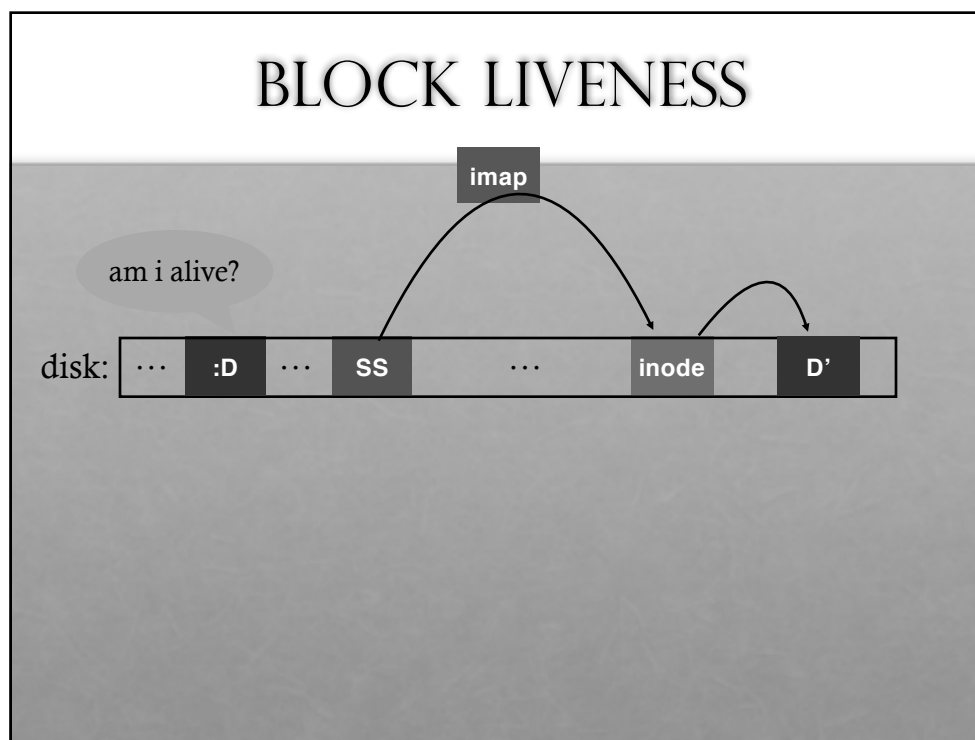
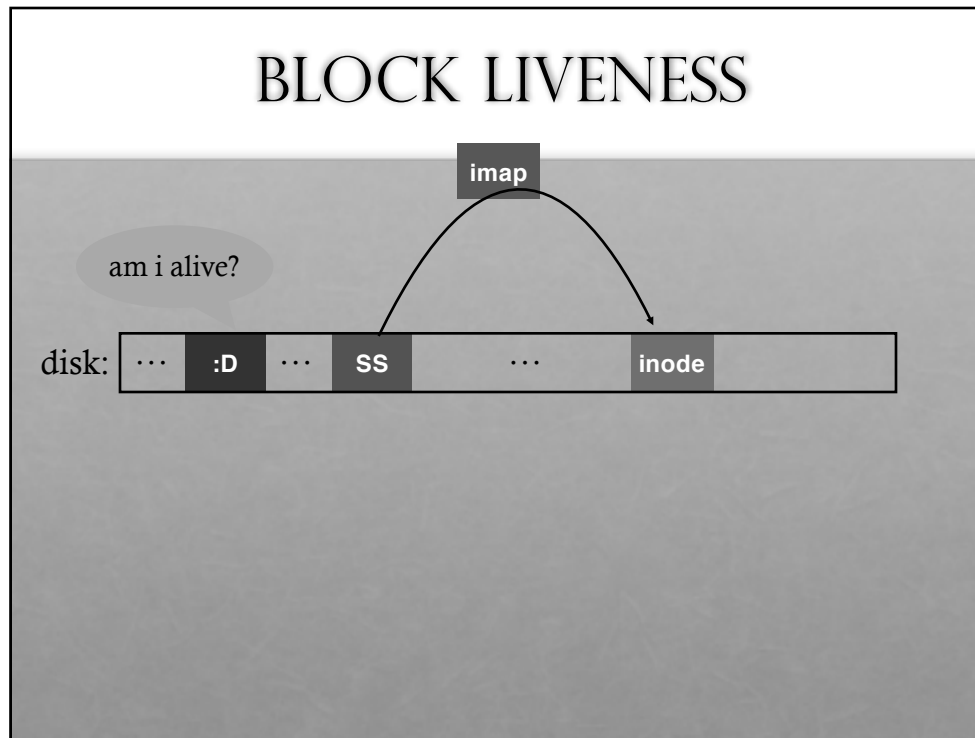
How to track information more efficiently?

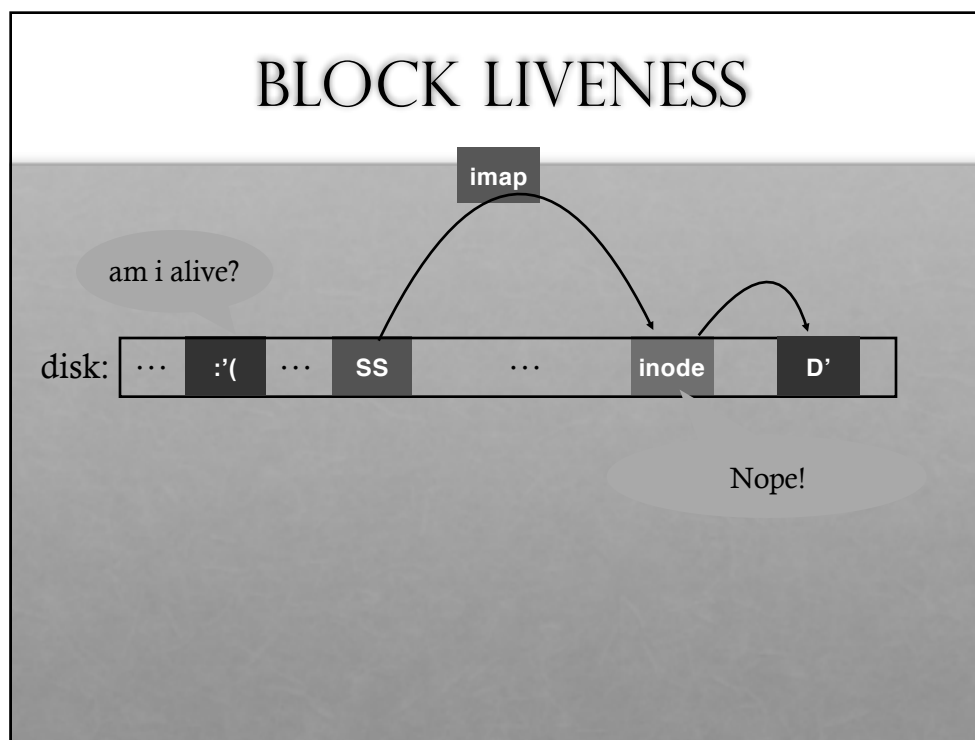
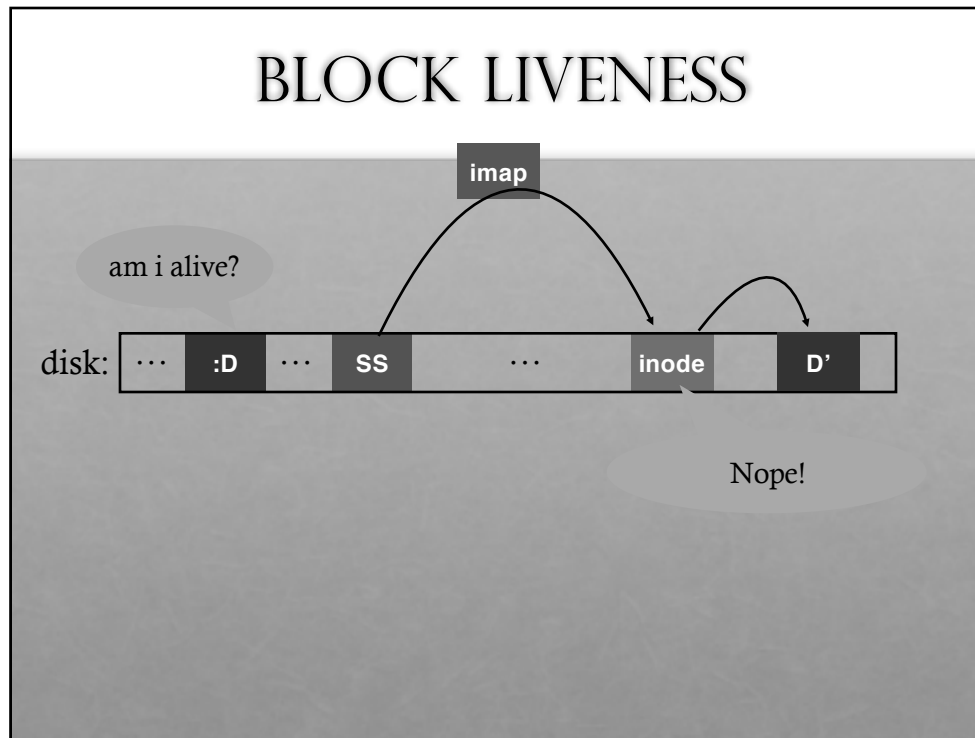
- **Segment summary** lists inode and data offset corresponding to each data block in segment (reverse pointers)

## BLOCK LIVENESS

am i alive?

disk: ... :D ... SS ...





# GARBAGE COLLECTION

**General operation:**

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

**Mechanism:**

How does LFS know whether data in segments is valid?  
[segment summary]

**Policy:**

Which segments to compact?

- clean most empty first
- clean coldest (segments changing least; wait longer for others)
- more complex heuristics...

# CONCLUSION

**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