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
- **LFS**: Log-Structured File System

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

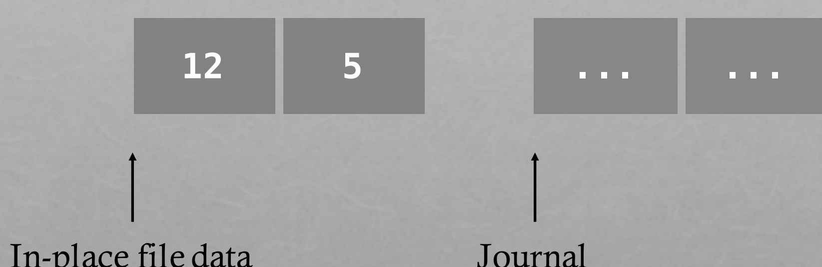
Implication:
At some point in time, all old AND all new data must be on disk

Two techniques popular in file systems:

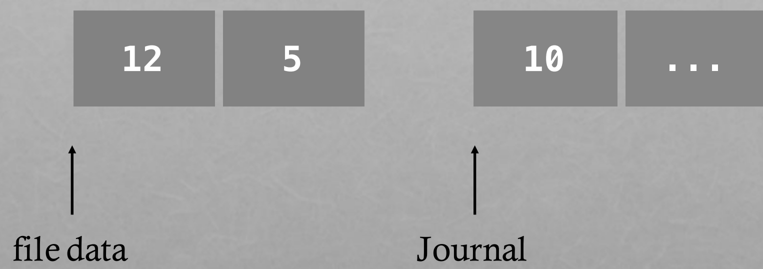
1. **journal** new info, then overwrite old info with new info **in place**

2. **copy-on-write**: write new info to new location, discard old info

Review: Journal New, Overwrite In-Place



Review: Journal New, Overwrite In-Place

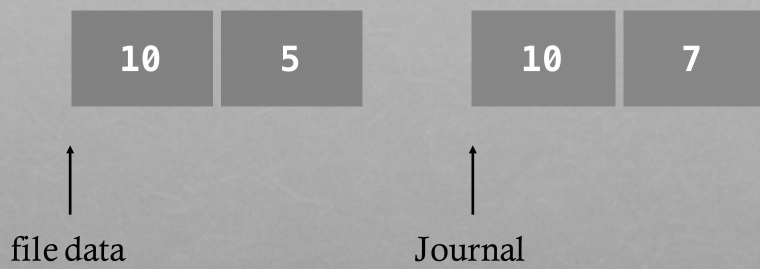


Imagine journal header describes in-place destinations

Review: Journal New, Overwrite In-Place

Imagine journal commit block designates transaction complete

Review: Journal New, Overwrite In-Place



Perform checkpoint to in-place data when transaction is complete

Review: Journal New, Overwrite In-Place

Review: Journal New, Overwrite In-Place

Clear journal commit block to show checkpoint complete

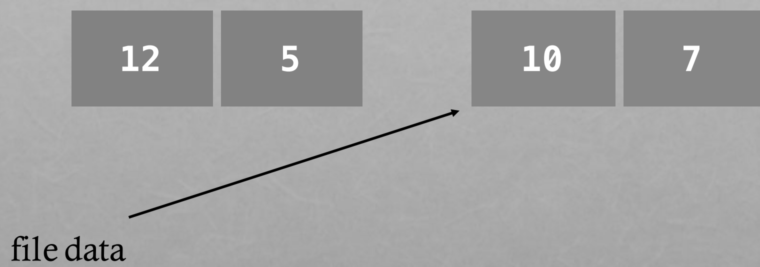
Today: Write New, Discard Old

Make a copy-on-write (COW)

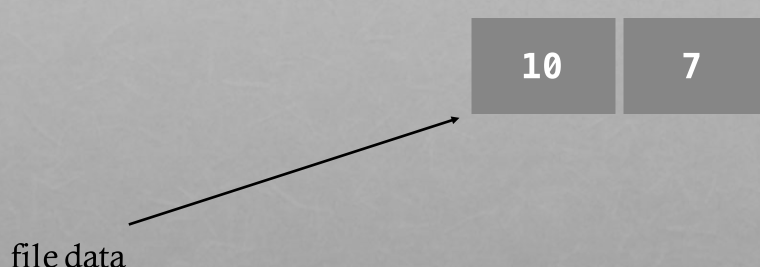
Today: Write New, Discard Old

Today: Write New, Discard Old

Today: Write New, Discard Old



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)

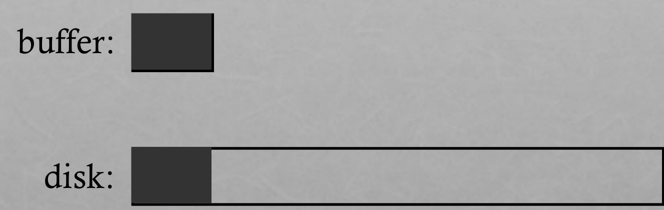
LFS Strategy

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

Never overwrite old info: old copies left behind

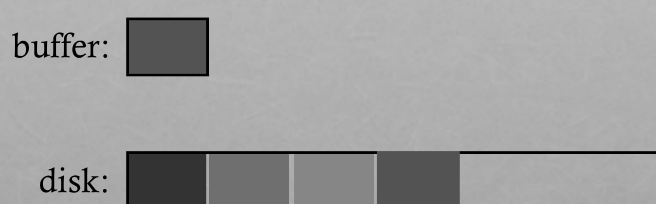
Big Picture



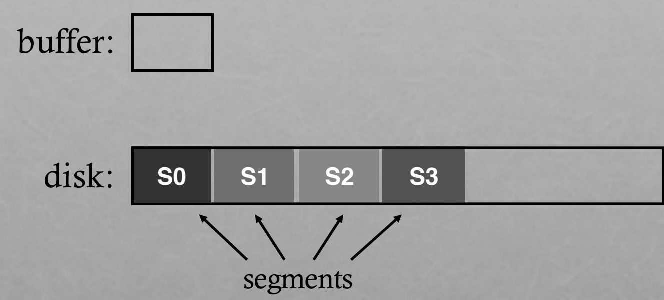
Big Picture

Big Picture

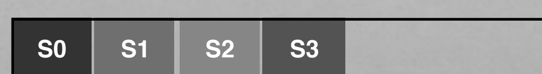
Big Picture



Big Picture



Data Structures (Attempt 1)



What data structures from FFS can LFS remove?

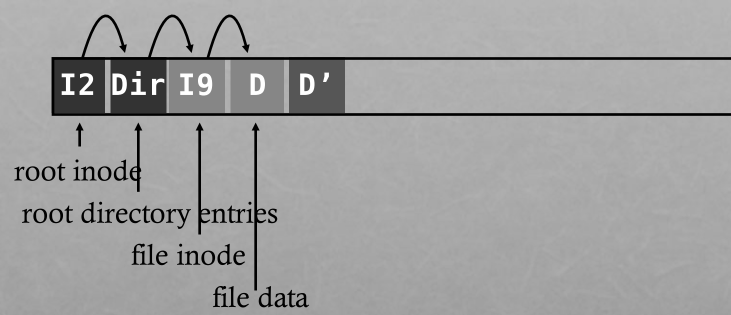
- allocation structs: data + inode bitmaps

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 number changes!!

ATTEMPT 1

Overwrite data in /file.txt



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

ATTEMPT 1

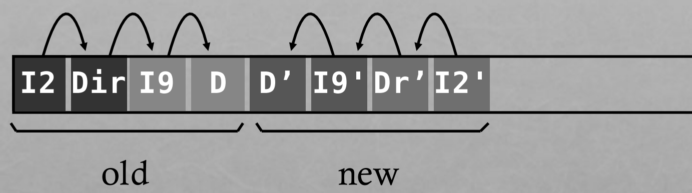
Overwrite data in /file.txt

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

NO! This would be a random write

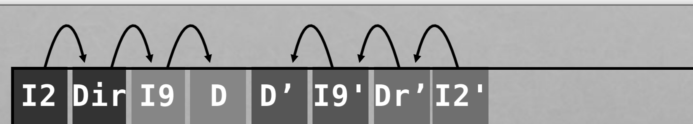
Attempt 1

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 number) changes

Solution:
  Keep inode numbers constant; don’t base name on offset

FFS found inodes with math. How now?

Data Structures (Attempt 2)

What data structures from FFS can LFS remove?

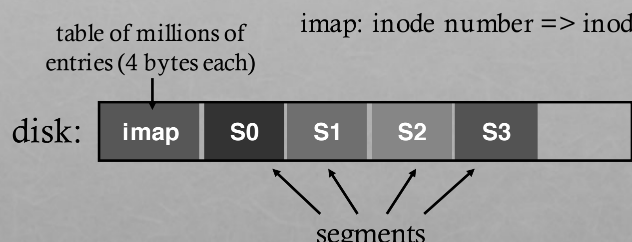
- allocation structs: data + inode bitmaps

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?

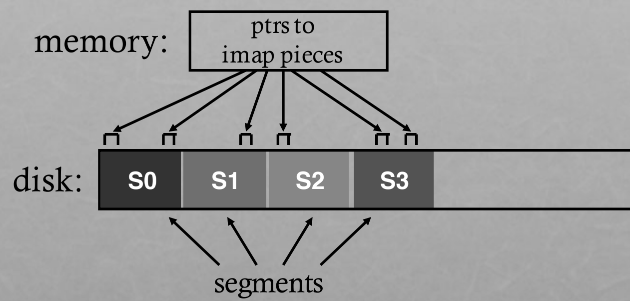
imap: inode number => inode location on disk



Where can imap be stored???? Dilemma:
1. imap too large to keep in memory
2. don't want to perform random writes for imap

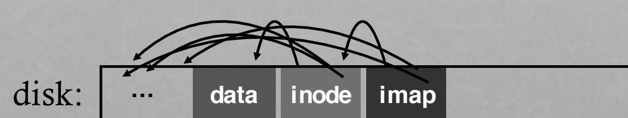
Solution:
Write imap in segments
Keep pointers to pieces of imap in memory

SOLUTION: IMAP IN SEGMENTS



Solution:
Write imap in segments
Keep pointers to pieces of imap in memory
Keep recent accesses to imap cached in memory

EXAMPLE WRITE



Solution:
Write imap in segments
Keep pointers to pieces of imap in memory
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)
	~~read~~ ~~write~~					
				~~read~~ write		
			write			
						write

Most data structures same in LFS as FFS!

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

Other Issues

Crashes

Garbage Collection

Crash Recovery

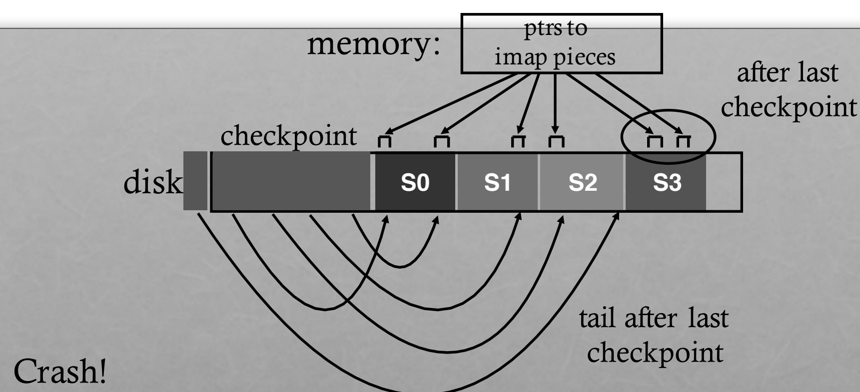
What data needs to be recovered after a crash?
- Need imap (lost in volatile memory)

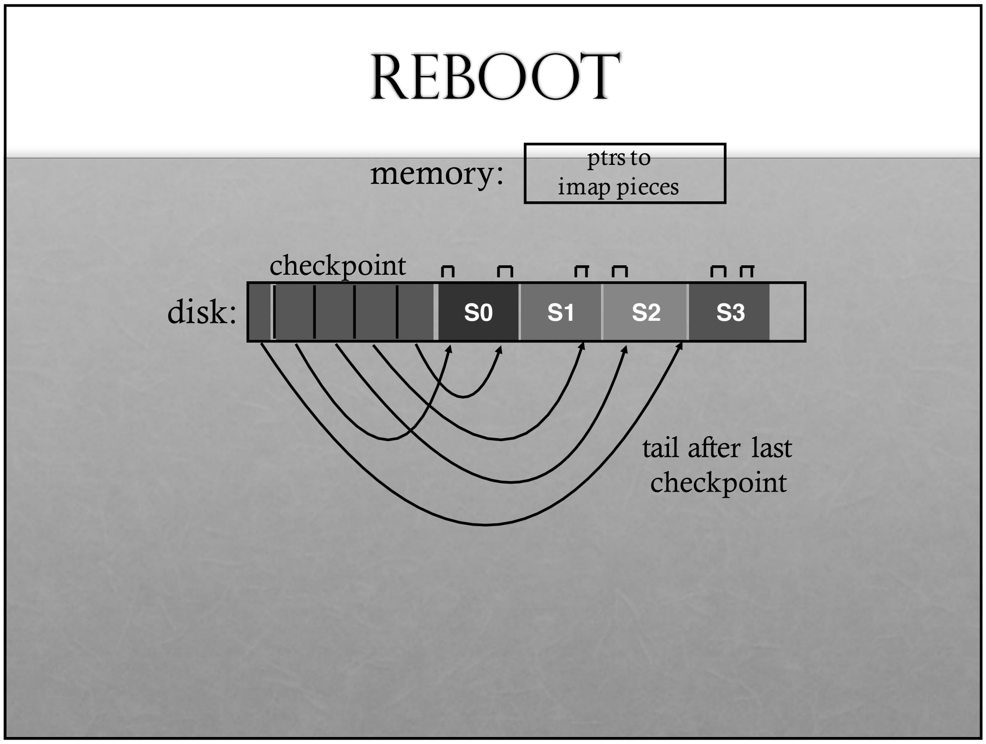
Naive approach?
- **Scan** entire log to reconstruct pointers to imap pieces. Slow!

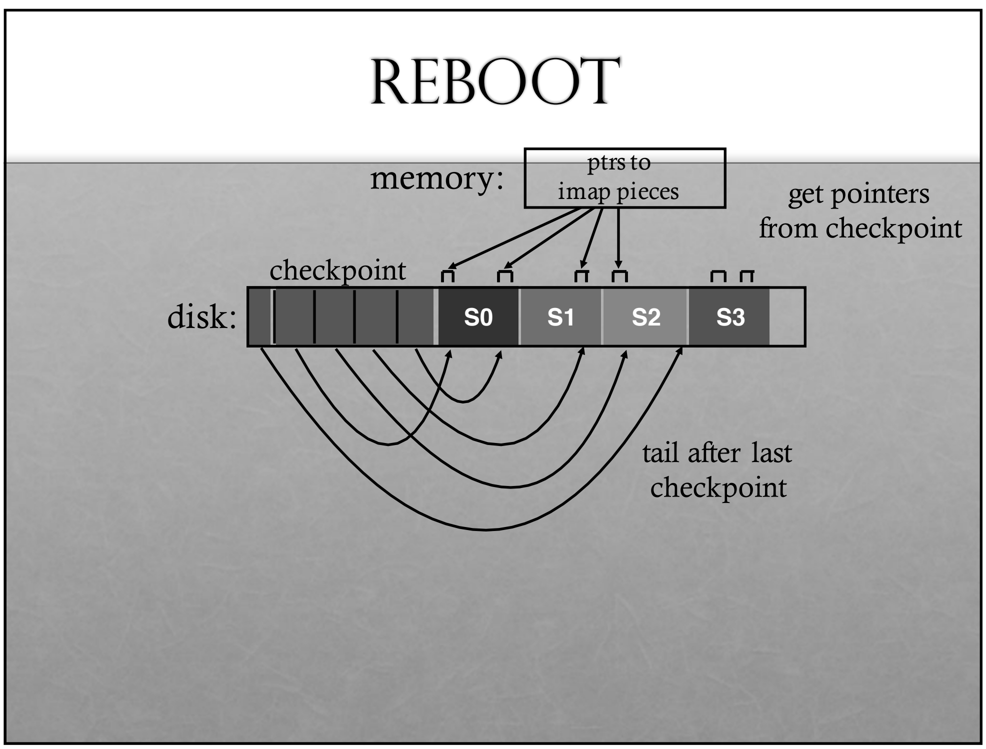
Better approach?
- Occasionally **checkpoint** to known on-disk location the pointers to imap pieces

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

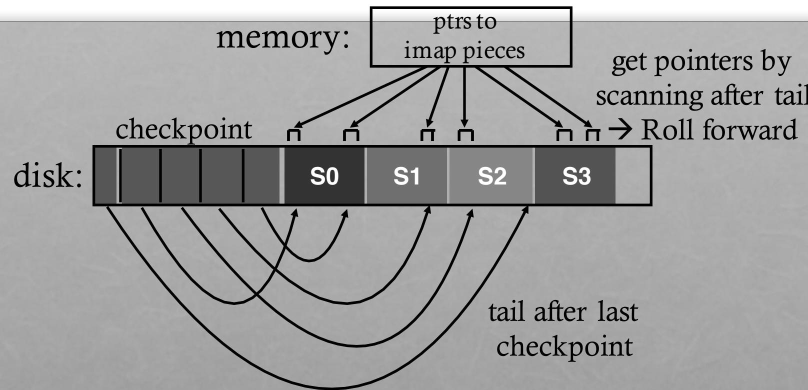
Checkpoint




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



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

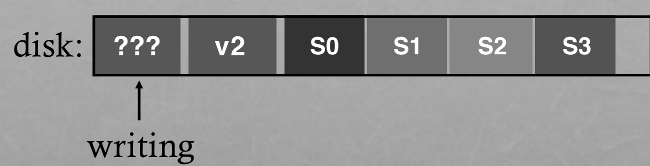
What if crash <u>during</u> checkpoint?

Checkpoint Strategy

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint



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

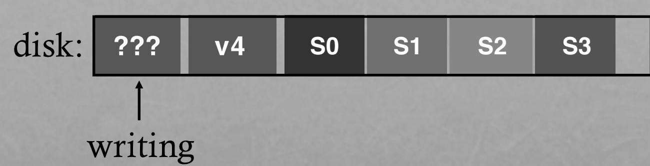
Use checksum/timestamps to identify newest checkpoint

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

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:

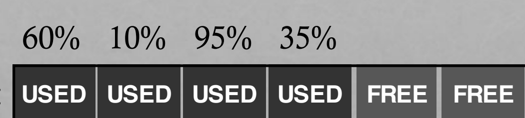
1. When no more references (any file system)
2. After newer copy is created (COW file system)

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

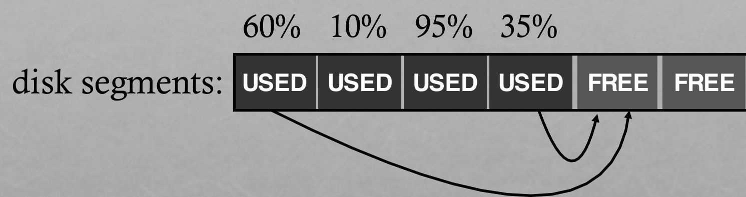
- Want future overwites to be to sequential areas
- Tricky, since segments are usually partly valid

Garbage Collection

disk segments:



Garbage Collection

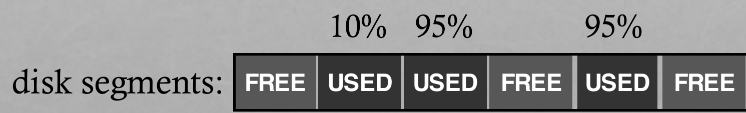


Garbage Collection

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



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 is pointed to
- Fast!

Is a data block the latest version?

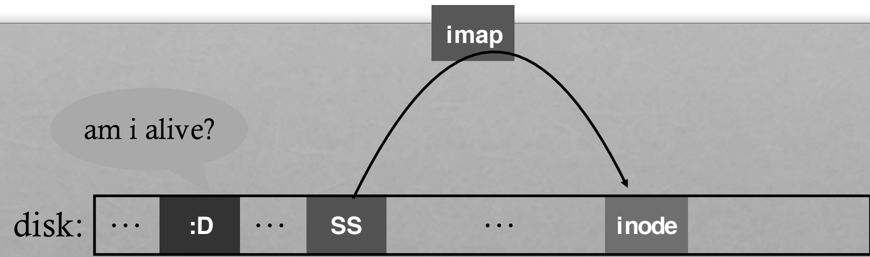
- Scan ALL inodes to see if any point to this data
- Very slow!

How to track information more efficiently?

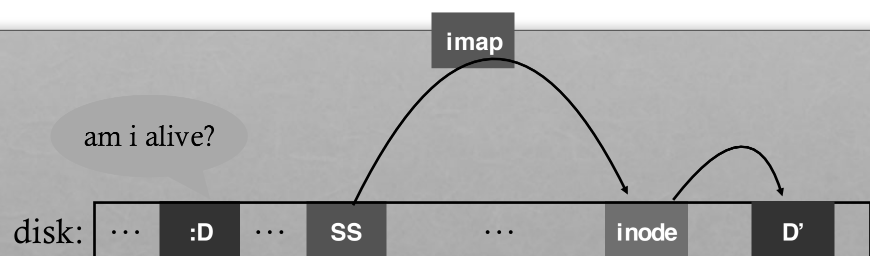
- **Segment summary** lists inode and data offset corresponding to each data block in segment (reverse pointers)

Block Liveness

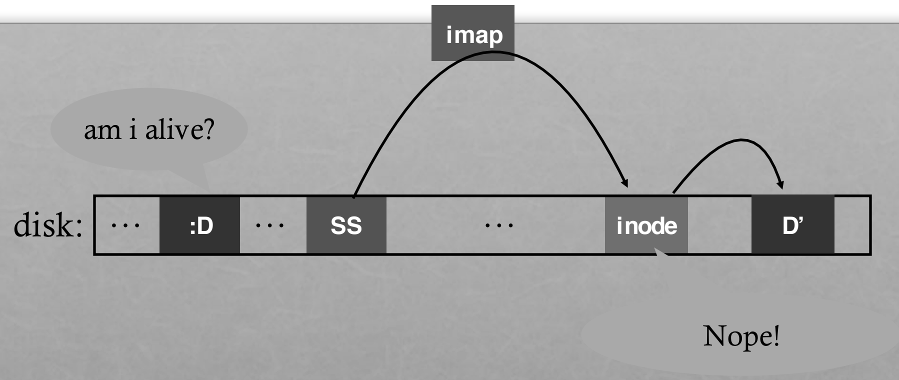
Block Liveness



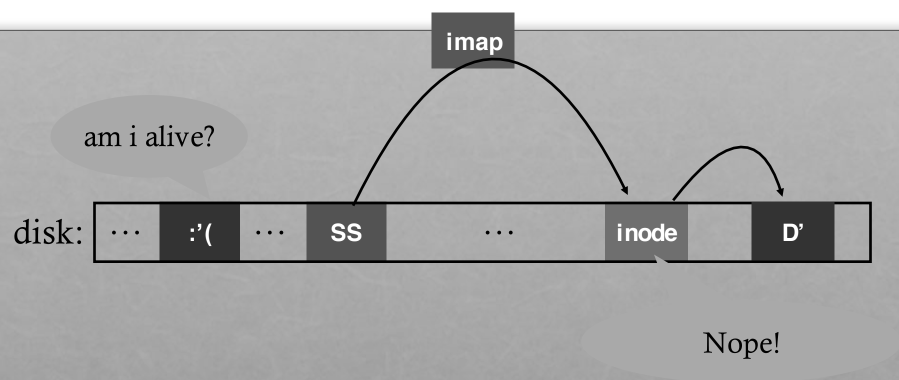
Block Liveness



Block Liveness



Block Liveness



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 (ones undergoing least change)
- 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