OSTEP
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

Journaling “Review”

Motivation: Redundancy

Definition: if *A* and *B* are two pieces of data, and knowing *A* eliminates some or all the values *B* could *B*, there is redundancy between *A* and *B*.

Superblock: field contains total blocks in FS.

Inode: field contains pointer to data block.

Is there redundancy between these fields? Why?

FFS Redundancy

Examples:
Dir entries AND inode table.
Dir entries AND inode link count.
Inode pointers AND data bitmap.
Data bitmap AND group descriptor.
Inode file size AND inode/indirect pointers.
…

Regaining Consistency After Crash

Solution 1: reformat disk

Solution 2: guess (fsck)

Solution 3: do fancy bookkeeping before crash

General Strategy for Crash Consistency

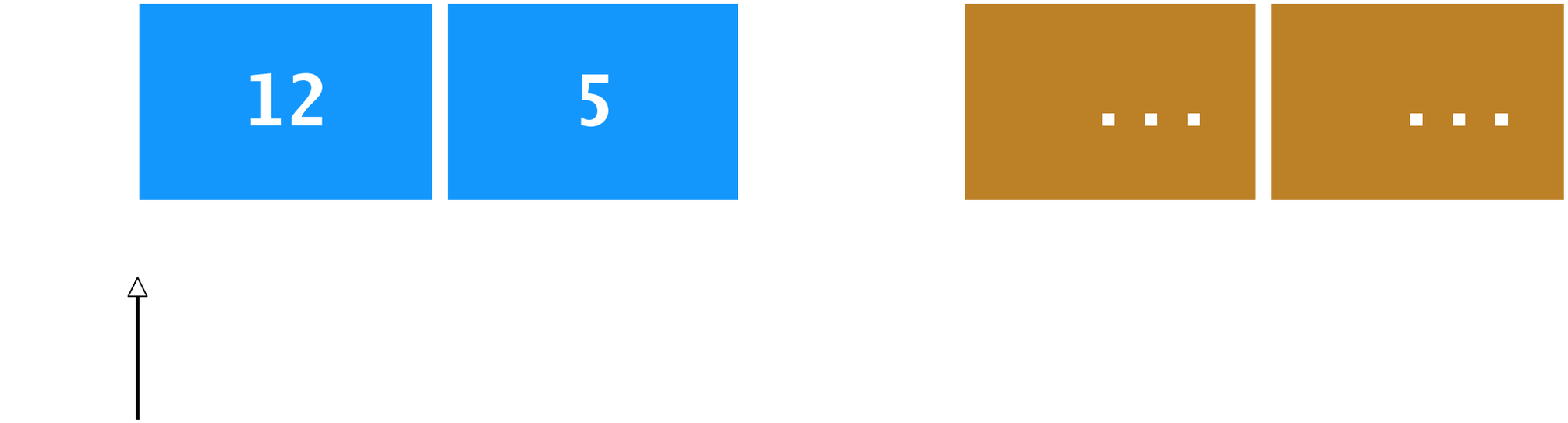
Never delete ANY old data, until ALL new data is safely on disk

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

Three techniques in file systems(2,3 are popular)

1. **journal** old info, then overwrite (new info with old info) **in place**
2. **journal** new info, then overwrite (old info with new info) **in place**
3. **copy-on-write**: write new info to new location, discard old info

1. Journal Old, Overwrite In-Place



file data

1. Journal Old, Overwrite In-Place

1. Journal Old, Overwrite In-Place

1. Journal Old, Overwrite In-Place

1. Journal Old, Overwrite In-Place

1. Journal Old, Overwrite In-Place

General Strategy for Crash Consistency

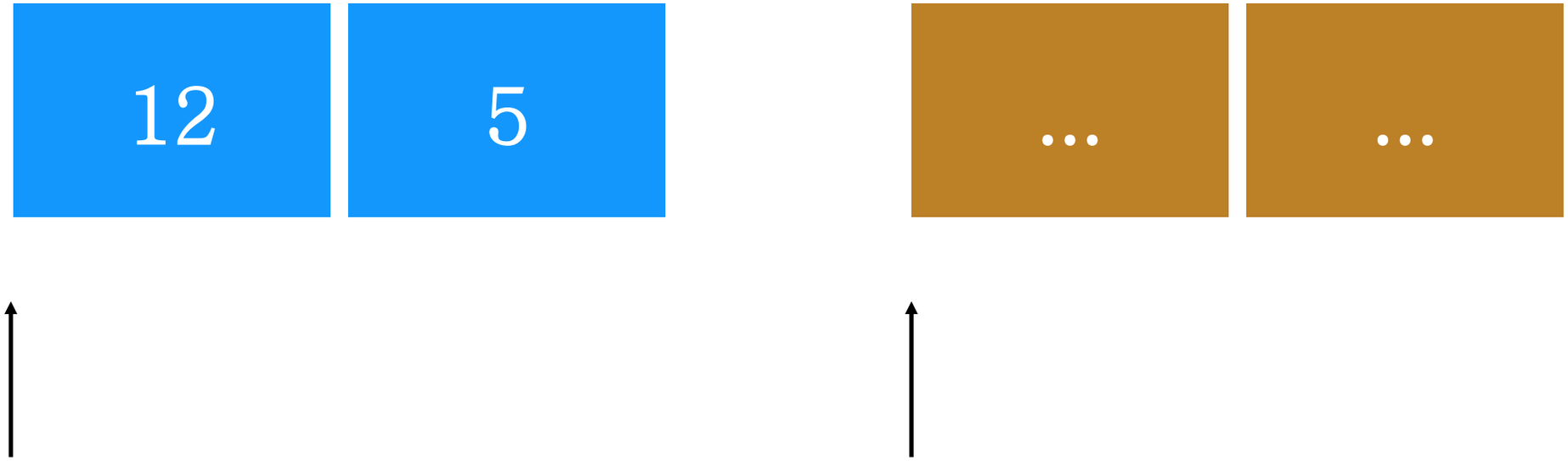
Never delete ANY old data, until ALL new data is safely on disk

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

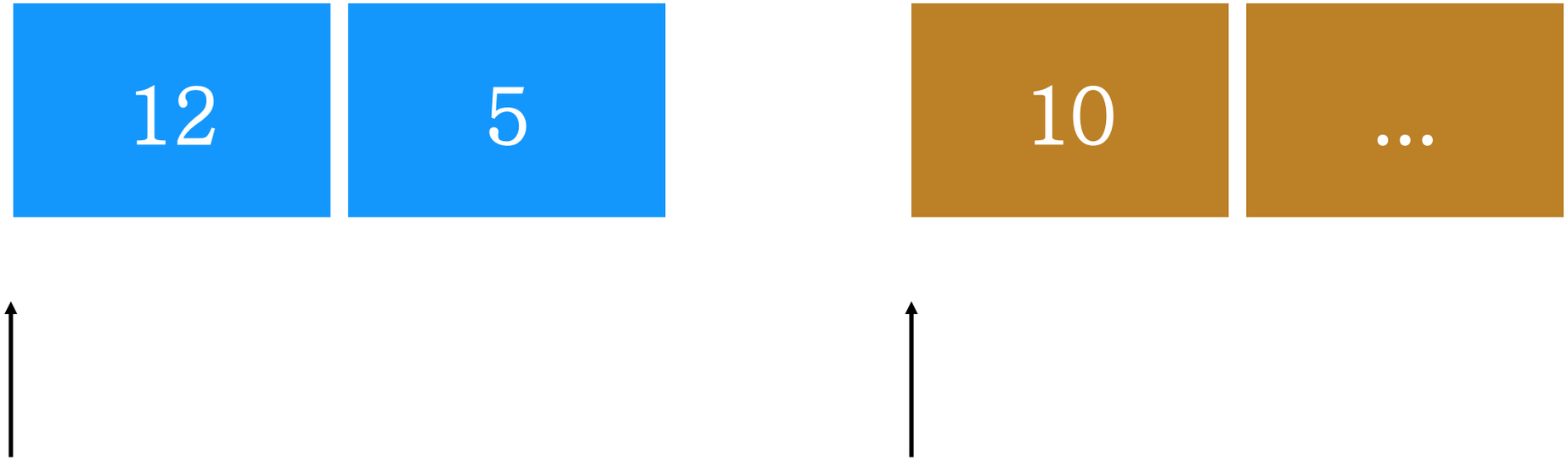
Three techniques in file systems(2,3 are popular)

1. **journal** old info, then overwrite (new info with old info) **in place**
2. **journal** new info, then overwrite (old info with new info) **in place**
3. **copy-on-write**: write new info to new location, discard old info

2. Journal New, Overwrite In-Place

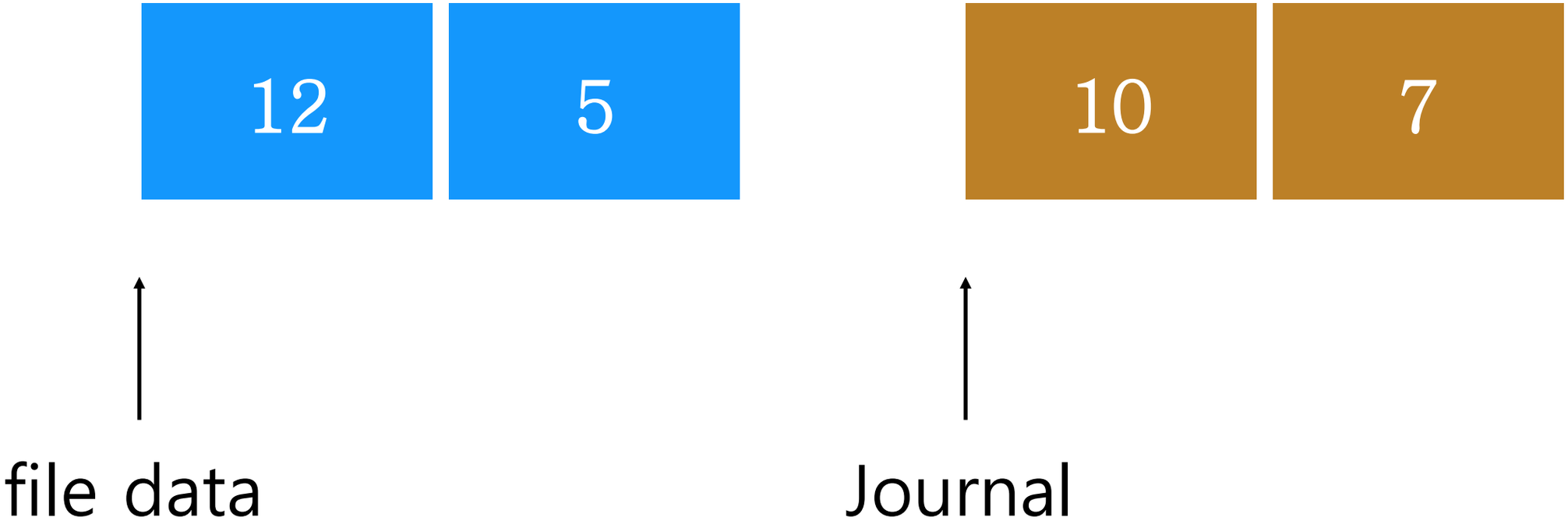


2. Journal New, Overwrite In-Place



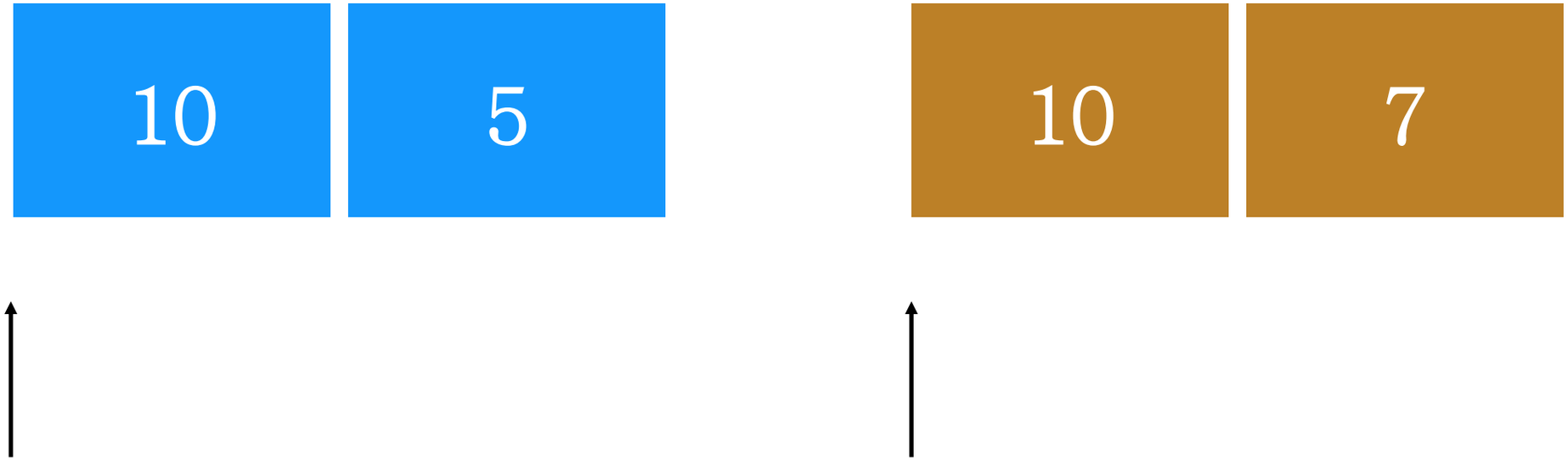
Imagine journal header describes in-place destinations

3. Journal New, Overwrite In-Place



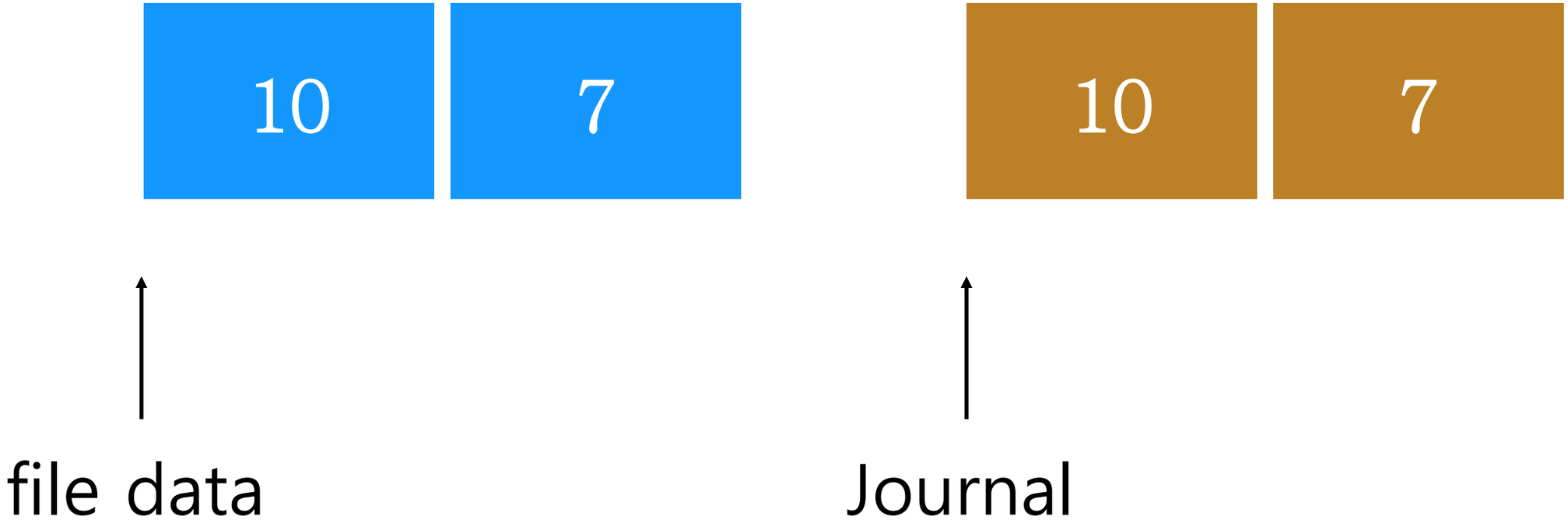
Imagine journal commit block designates transaction complete

4. Journal New, Overwrite In-Place

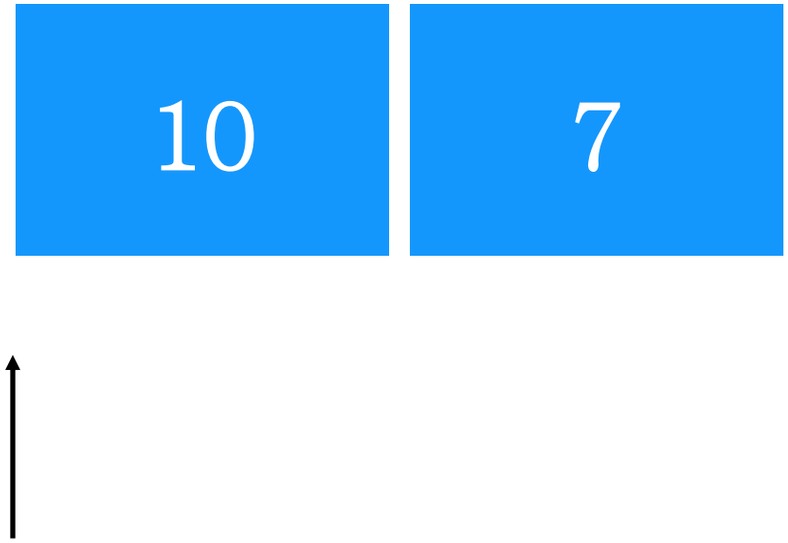


Perform checkpoint to in-place data when transaction is complete

5. Journal New, Overwrite In-Place



6. Journal New, Overwrite In-Place



Clear journal commit block to show checkpoint complete

General Strategy for Crash Consistency

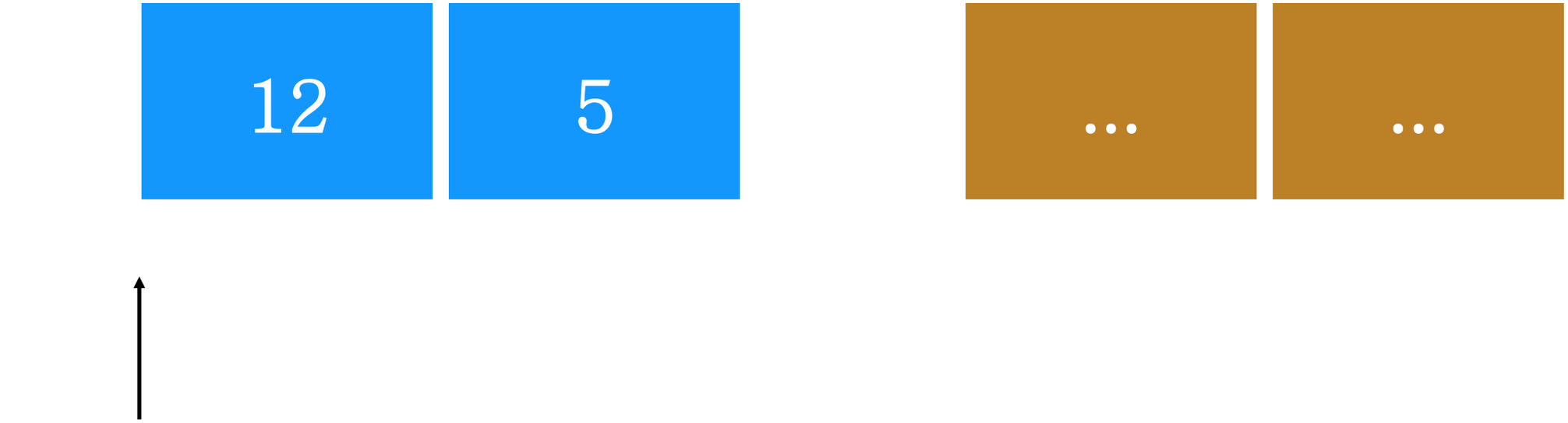
Never delete ANY old data, until ALL new data is safely on disk

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

Three techniques in file systems(2,3 are popular)

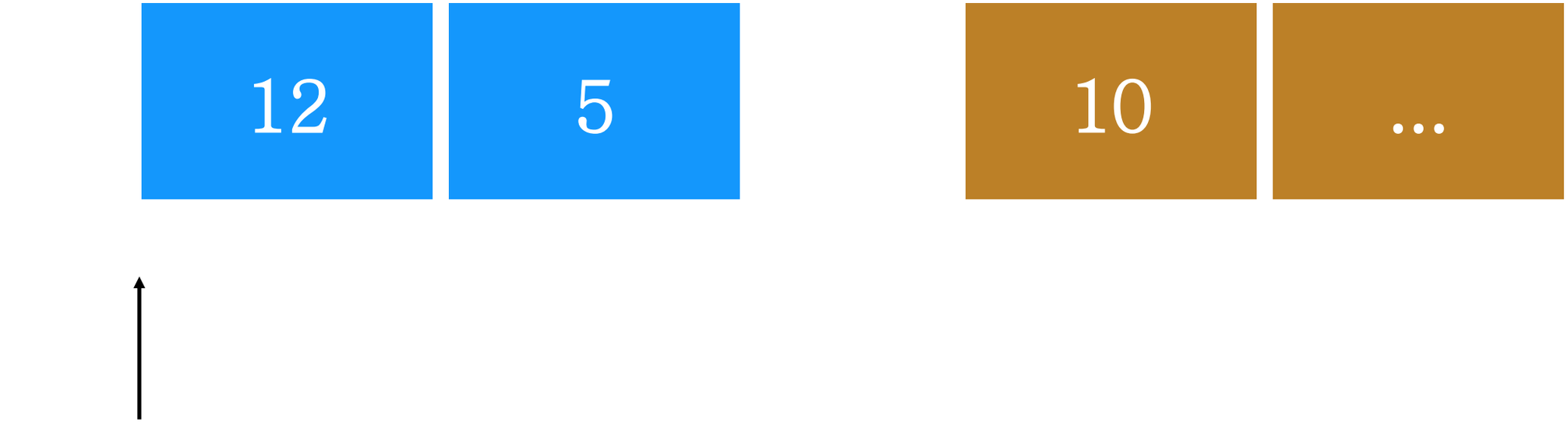
1. **journal** old info, then overwrite (new info with old info) **in place**
2. **journal** new info, then overwrite (old info with new info) **in place**
3. **copy-on-write**: write new info to new location, discard old info

3. Write New, Discard Old

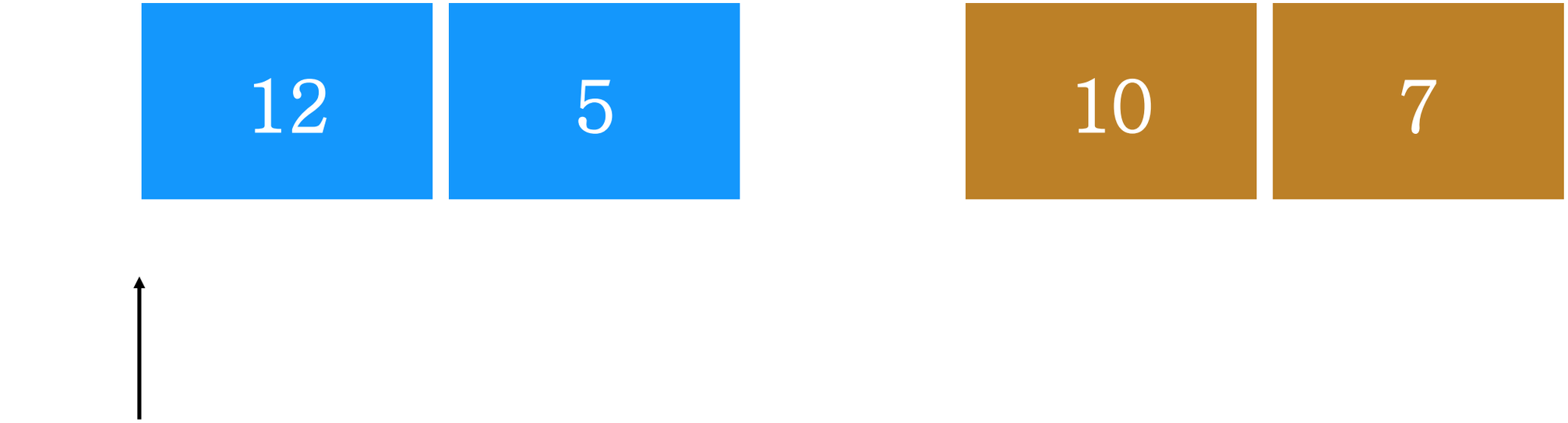


Make a copy-on-write (COW)

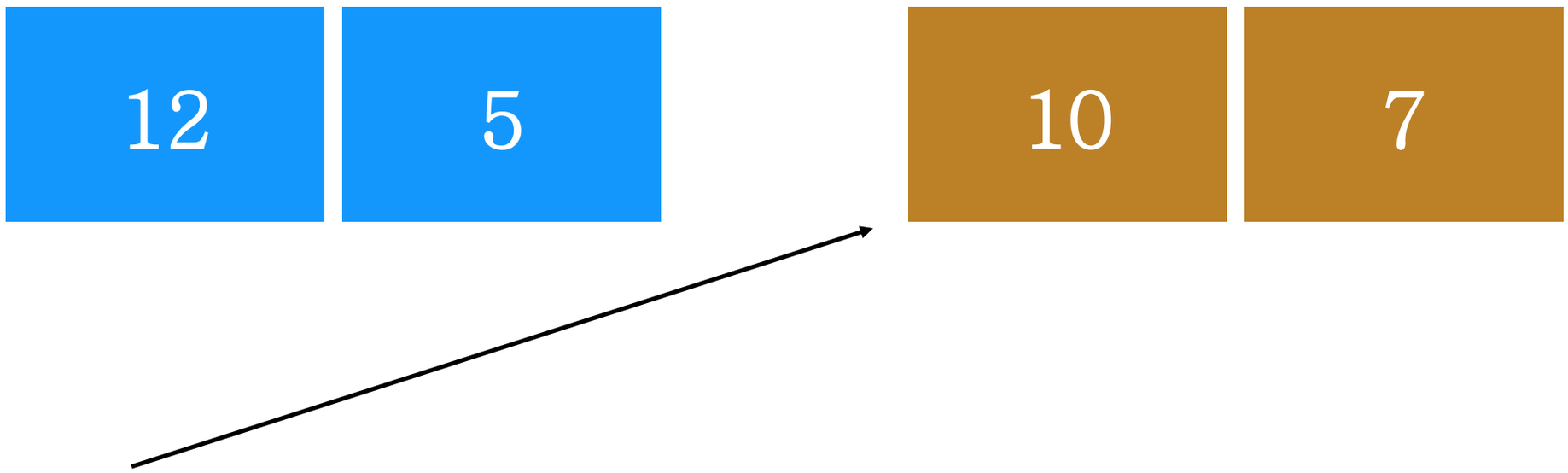
3. Write New, Discard Old



3. Write New, Discard Old

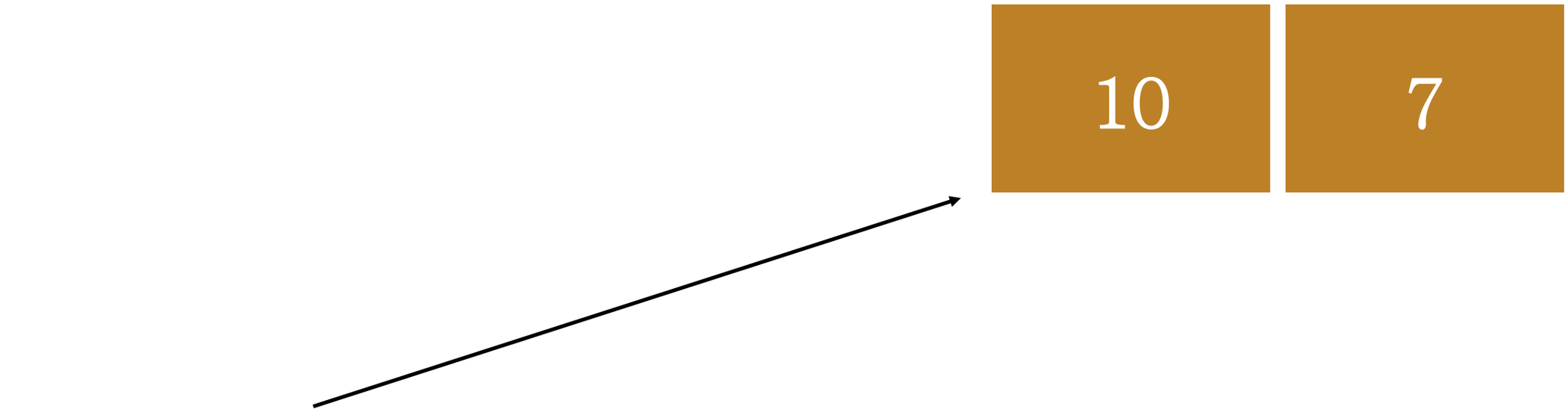


3. Write New, Discard Old



file data

3. Write New, Discard Old



Obvious advantage?

Only write new data once instead of twice

Log-Structured File System

LFS: Log-Structured File System

Different than FFS:
- optimizes allocation for writes instead of reads

Different than Journaling:
- use copy-on-write for atomicity

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

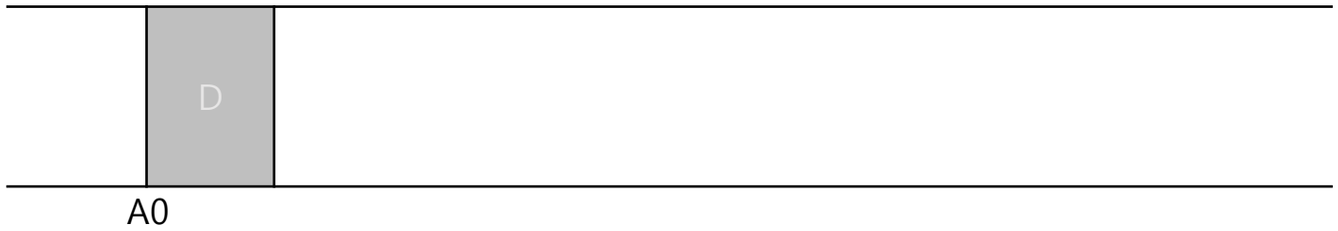
- How much is enough?
- Enough to get good sequential bandwidth from disk (MB)

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

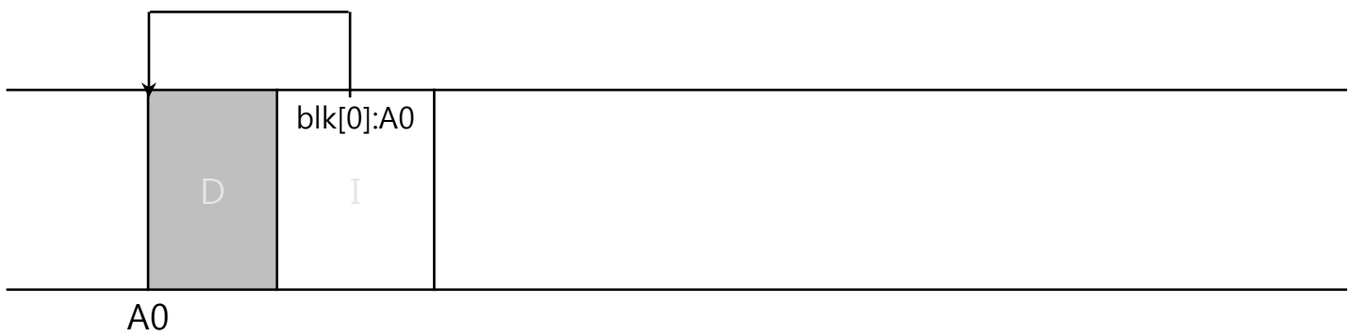
Never overwrite old info: old copies left behind

Writing to Disk Sequentially

- How do we transform all updates to file-system state into a series of sequntial writes to disk?
  - data update



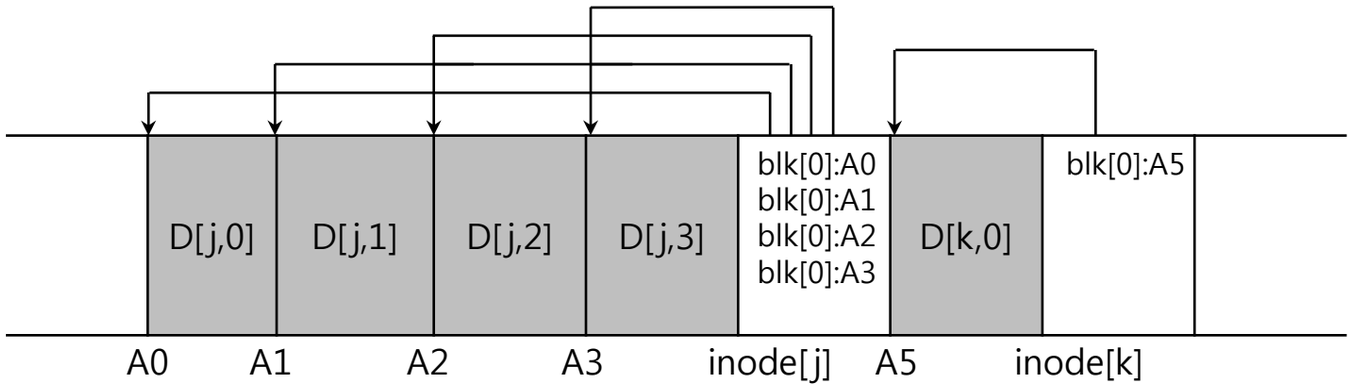
- metadata needs to be updated too. (Ex. inode)



Writing to Disk Sequentially and Effectively

- Writing single blocks sequentially does not guarantee efficient writes
  - After writing into A0, next write to A1 will be delayed by disk rotation

- Write buffering for effectiveness
  - Keeps track of updates in **memory buffer** (also called **segment**)
  - Writes them to disk all at once, when it has sufficient number of updates.



How Much to Buffer?

- Each write to disk has fixed overhead of positioning
  - Time to write out D MB

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \quad (43.1)$$

($T_{position}$: positioning time, R_{peak}: disk transfer rate)

- To amortize the cost, how much should LFS buffer before writing?
  - Effective rate of writing can be denoted as follows

$$R_{effecitve} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} \quad (43.2)$$

How Much to Buffer?

- Assume that $R_{effecitve} = F \times R_{peak}$ (F: fraction of peak rate, 0 < F < 1), then

$$R_{effecitve} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \quad (43.3)$$

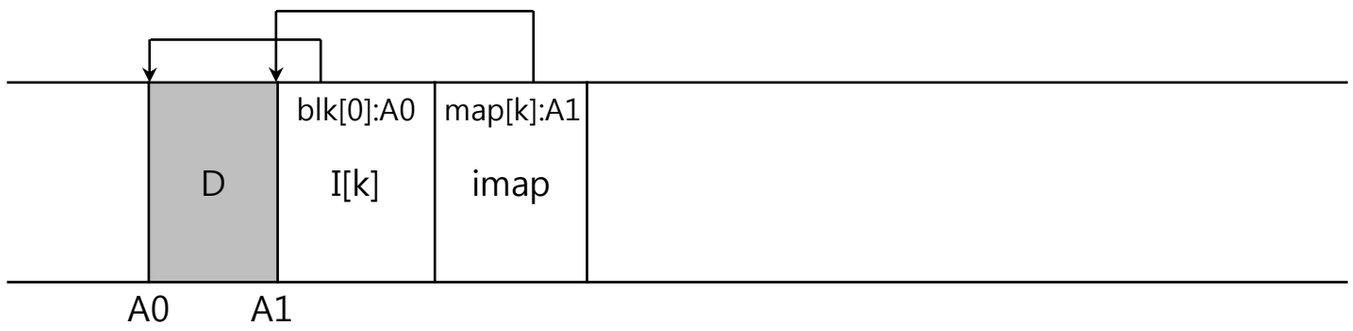
- Solve for D

$$\mathrm{D} = \frac{F}{1-F} \times R_{peak} \times T_{position} \quad (43.6)$$

- If we want F to be 0.9 when $T_{position} = 10msec$ and $R_{peak} = 100MB/s$,
  then $D = 9MB$ by the equation.
  - Segment size should be 9MB at least.

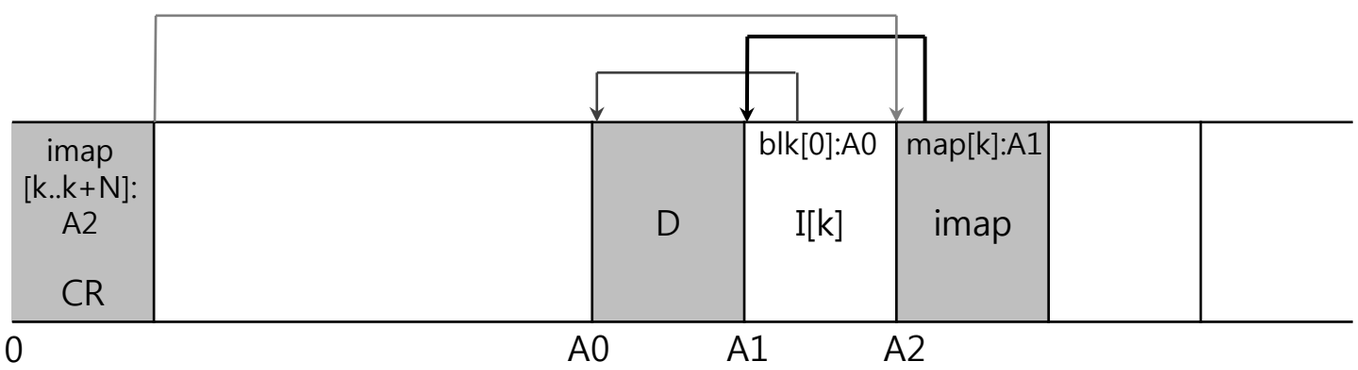
Finding Inode in LFS

- Inodes are scattered throughout the disk!
- Solution is through indirection “Inode Map” (imap)
- LFS place the chunks of the inode map right next to where it is writing all of the other new new information



The Checkpoint Region

- How to find the inode map, spread across the disk?
  - The LFS File system have fixed location on disk to begin a file lookup
- **Checkpoint Region** contains pointers to the latest of the inode map
  - Only updated periodically (ex. Every 30 seconds)
    → performance is not ill-affected

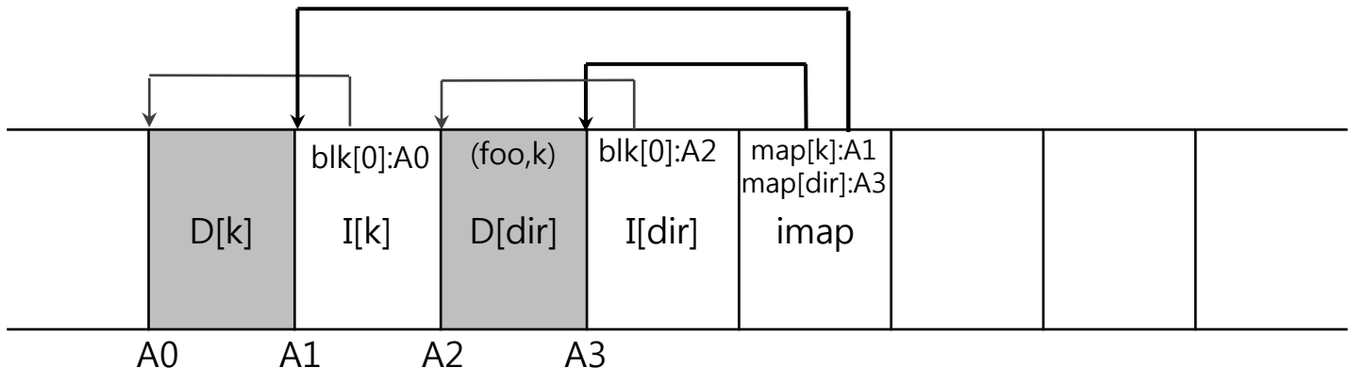


Reading a File from Disk: A Recap

- Read checkpoint region
- Read entire inode map and cache it in memory
- Read the most recent inode
- Read a block from file by using direct or indirect or doubly-indirect pointers

What About Directories?

- Directory structure of LFS is basically identical to classic UNIX file systems.
  - Directory is a file which data blocks consist of directory information

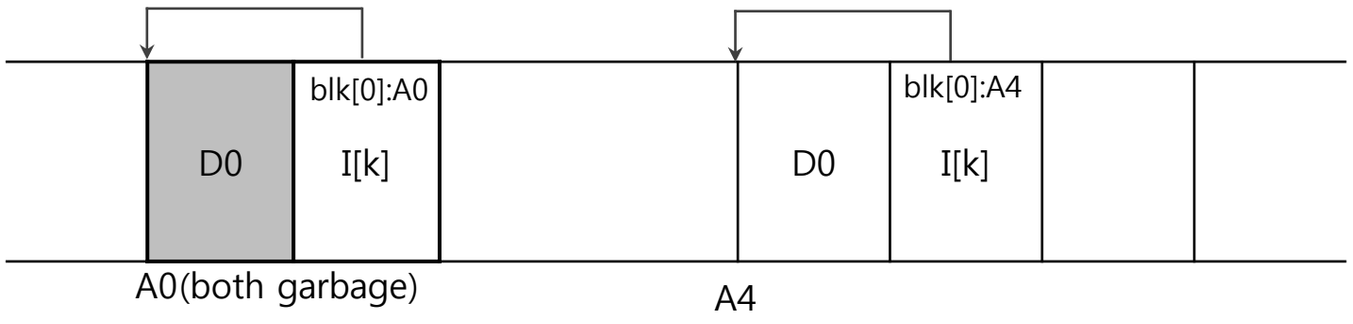


Garbage Collection

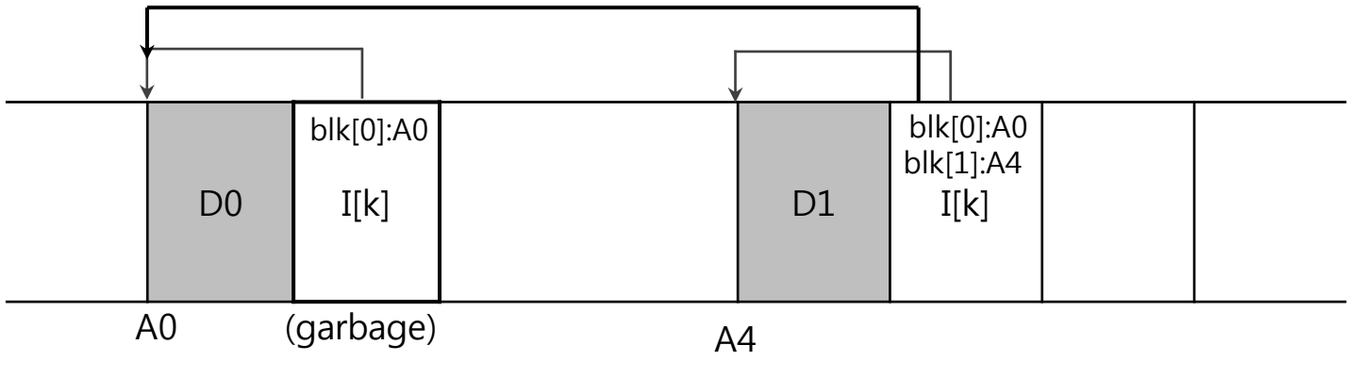
- LFS keeps writing newer version of file to new locations.
- Thus, LFS leaves the older versions of file structures all over the disk, call as garbage.

Examples: Garbage

- For a file with a singe data block
  - Overwrite the data block: both old data block and inode become garbage



- Append a block to that original file k: old inode becomes garbage

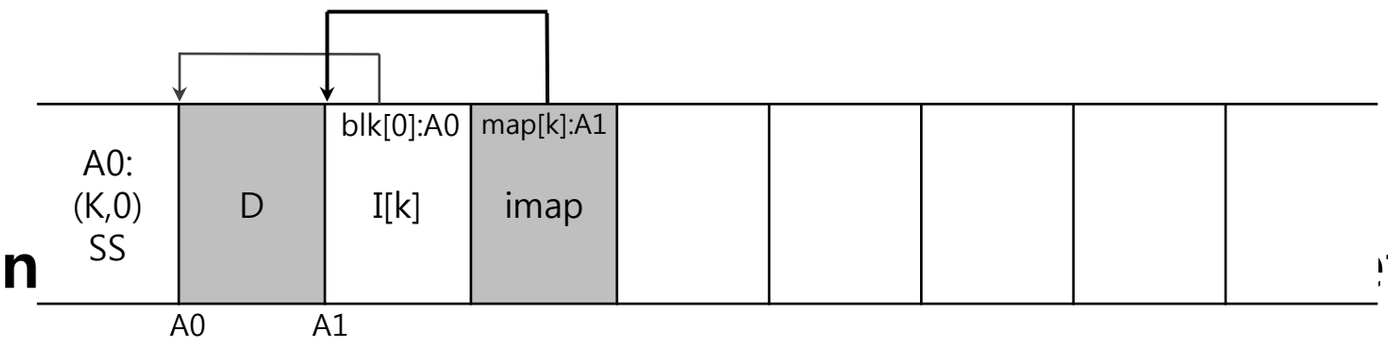


Handling older versions of inodes and data blocks

- One possibility: **Versioning file system**
  - keep the older versions around
  - Users can restore old file versions

- LFS approach: **Garbage Collection**
  - Keep only the latest live version and periodically clean old dead versions
  - Segment-by-segment basis
    - Block-by-block basis cleaner eventually make free holes in random location
      → Writes can not be sequential anymore

Determining Block Liveness

- **Segment summary block (SS)**
  - Located in each segment
  - Inode number and offset for each data block are recorded
- Determining Liveness
  - The block is live if the latest inode indicates the block
- **Version** 

termining

Which Blocks to Clean, and When?

- When to clean
  - Periodically
  - During idle time
  - When the disk is full
- Which blocks to clean
  - Segregate hot/cold segments
    - Hot segment: frequently over-written
      → more blocks are getting over-written if we wait a long time before cleaning
    - Cold segment: relatively stable
      → May have a few dead blocks, but the other blocks are stable
  - Clean cold segment sooner and hot segment later

Crash Recovery and the Log

- Log organization in LFS
  - CR points to a head and tail segment
  - Each segment points to next segment
- LFS can easily recover by simply reading latest valid CR
  - The latest consistent snapshot may be quite old
- To ensuring atomicity of CR update
  - Keep two CRs
  - CR update protocol: timestamp → CR → timestamp
- Roll forward
  - Start from end of the log (pointed by the lastest CR)
  - Read next segments and adopt any valid updates to the file system

Big Picture

buffer: 

disk: 

Big Picture

buffer: 

disk: 

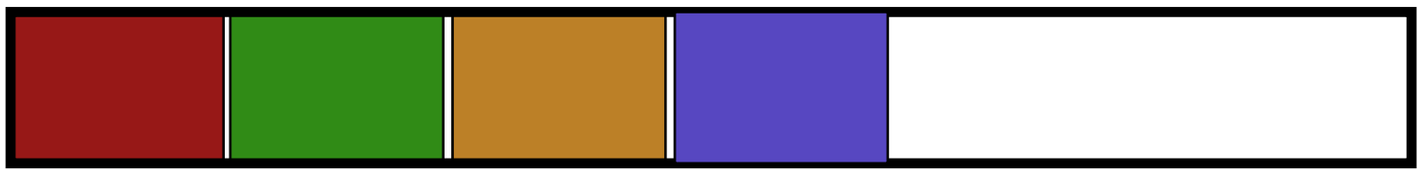
Big Picture

buffer: 

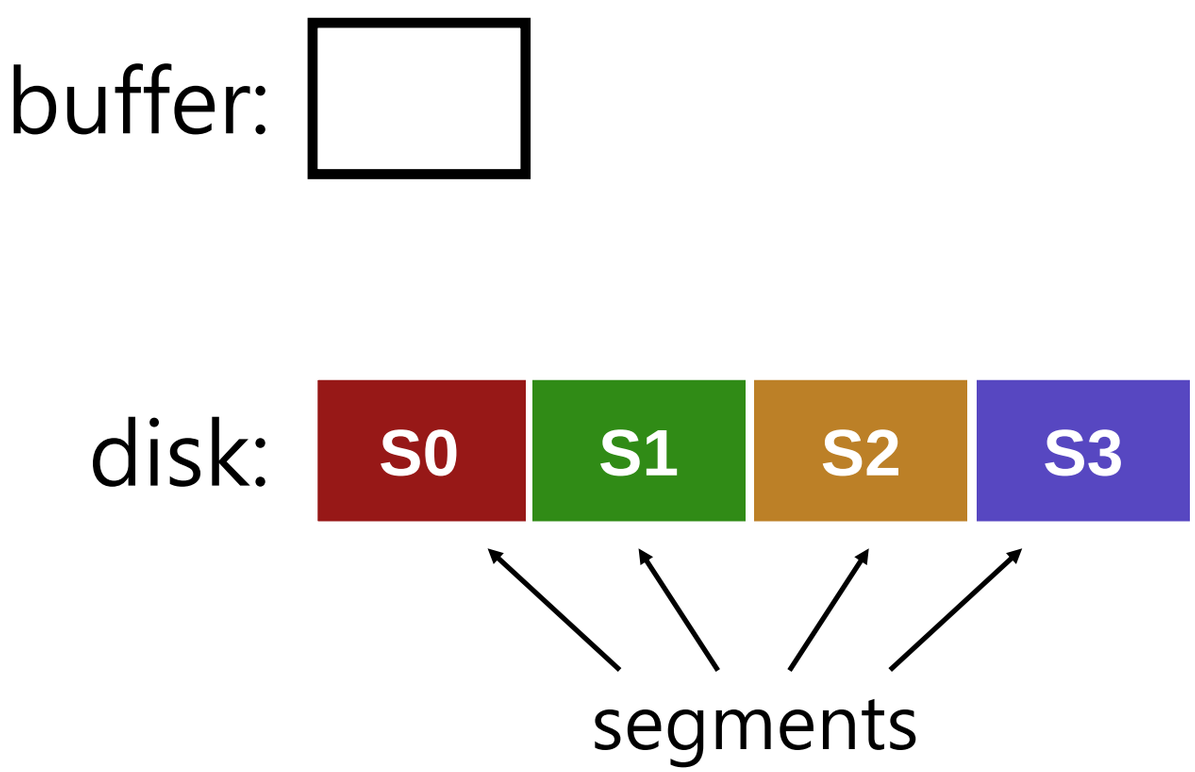
disk: 

Big Picture

buffer: 

disk: 

Big Picture



Data Structures (attempt 1)

What data structures from FFS can LFS remove?

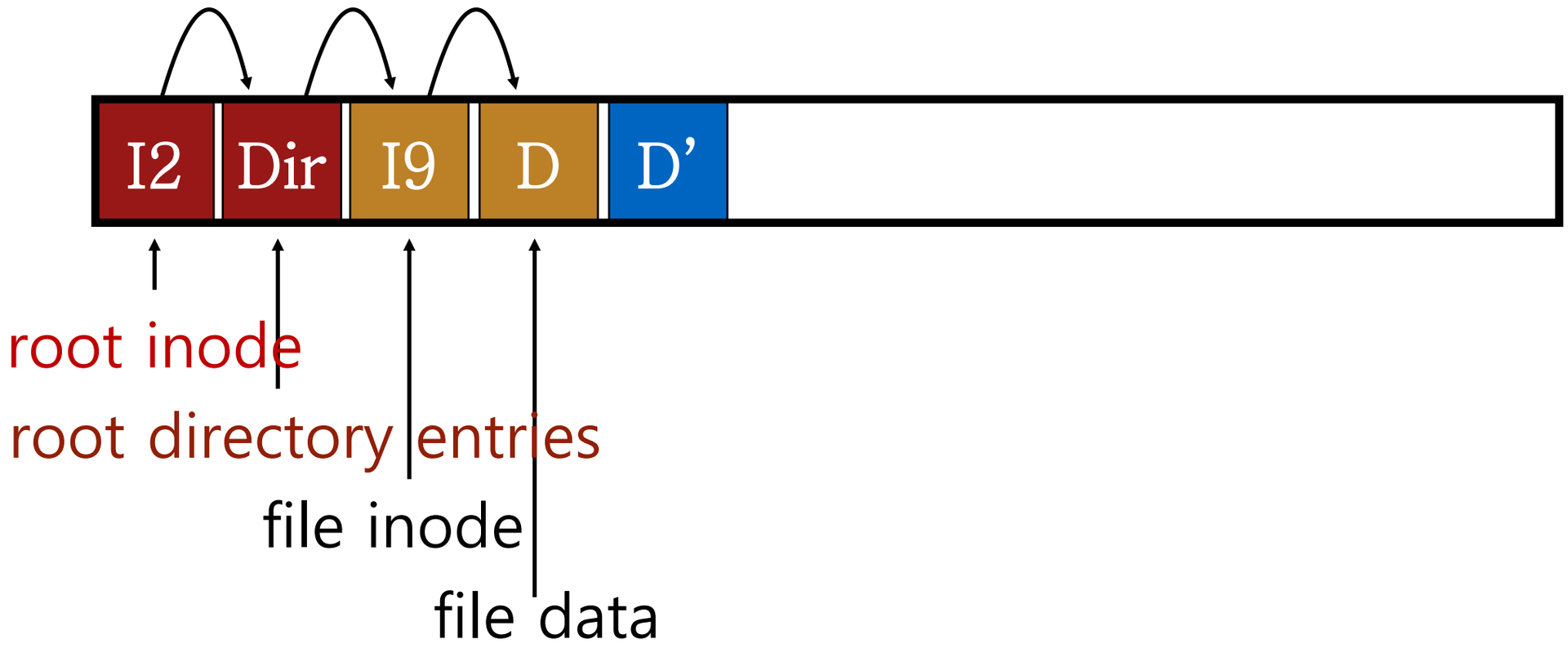
- allocation structs: data + inode bitmaps

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

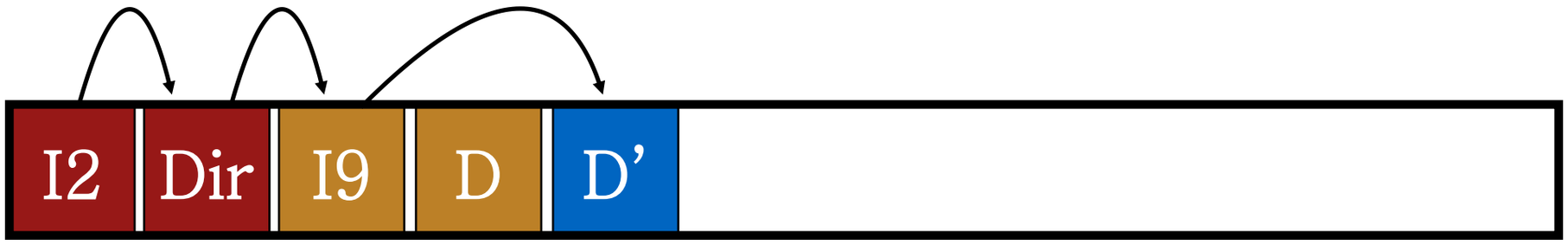
Overwrite data in /file.txt



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

Attempt 1

Overwrite data in /file.txt

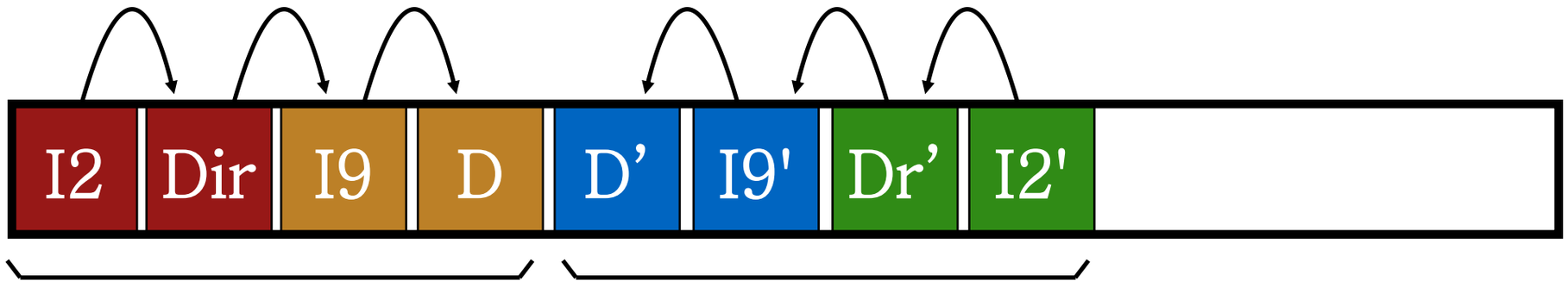


Can LFS update Inode 9 to point to new D’?

NO! This would be a random write

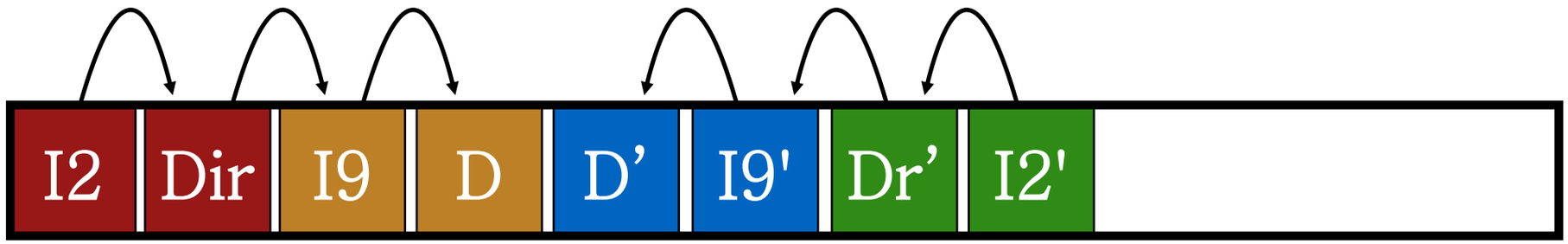
Attempt 1

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)

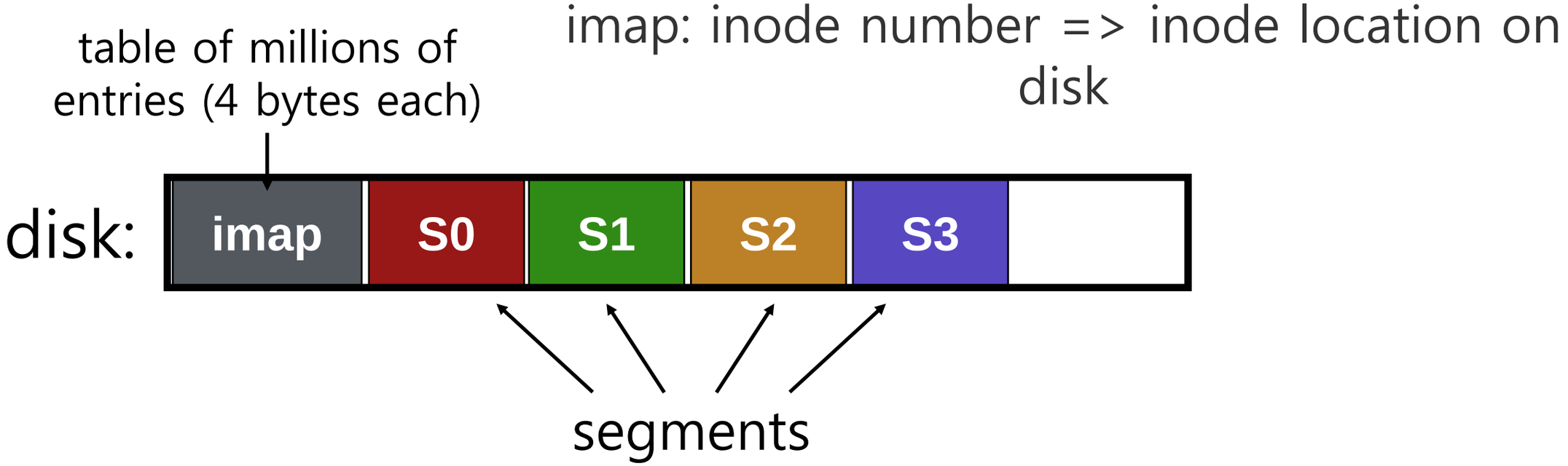
What data structures from FFS can LFS remove?

- 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

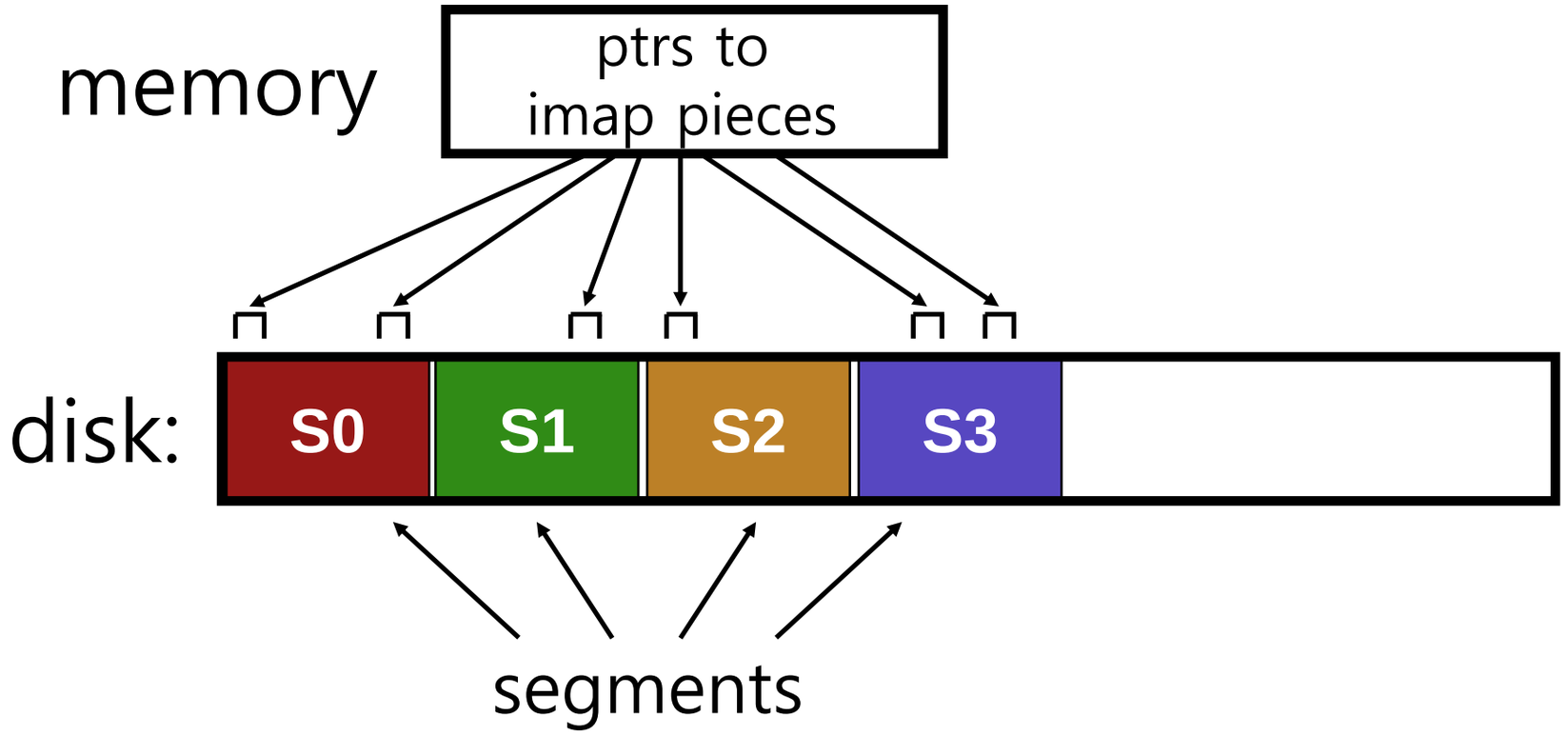
Where to keep Imap?



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

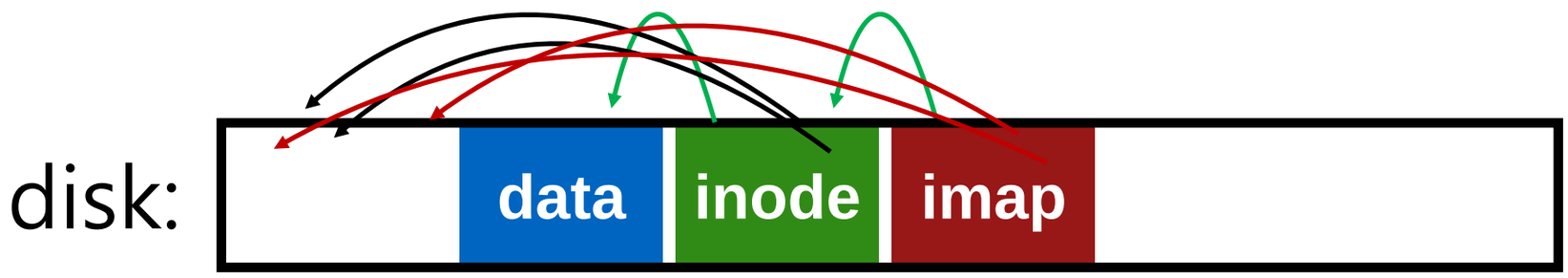
Solution:
Write imap in segments
Keep pointers to pieces of imap in memory

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!

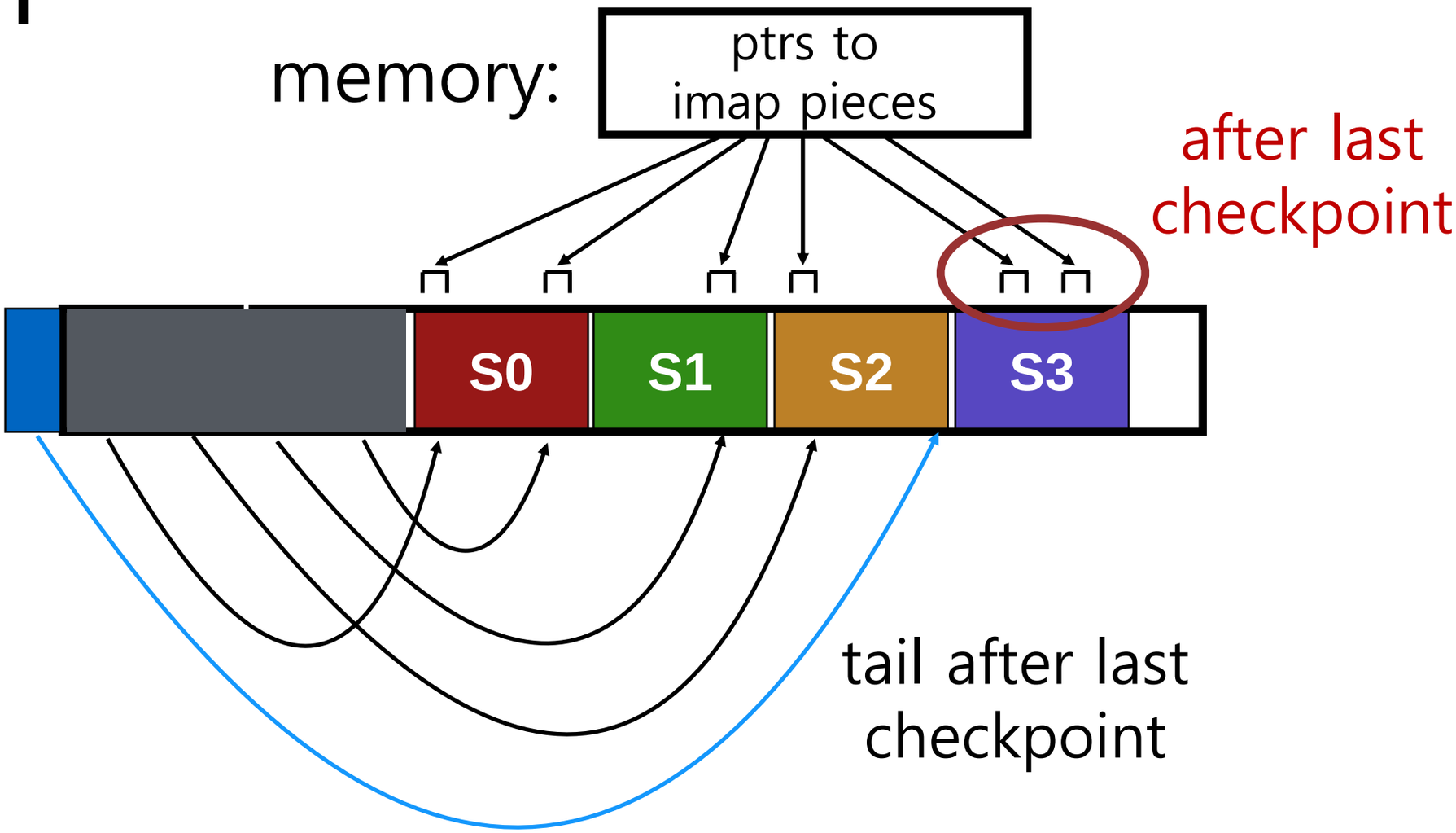
Better approach?

- Occasionally **checkpoint** to known on-disk location the pointer s to imap pieces

How often to checkpoint?

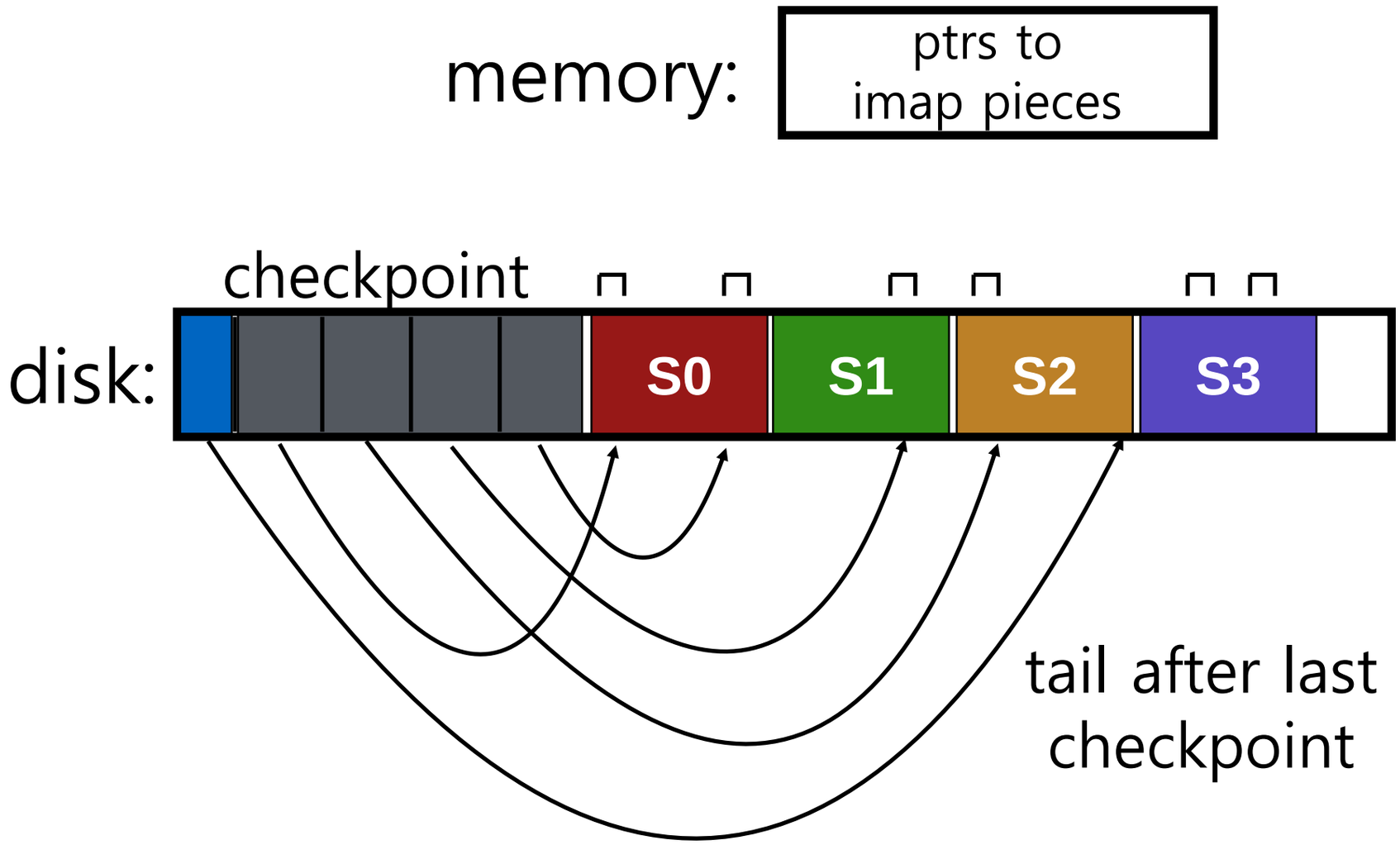
- Checkpoint often: random I/O
- Checkpoint rarely: lose more data, recovery takes longer
- Example: checkpoint every 30 secs

Checkpoint

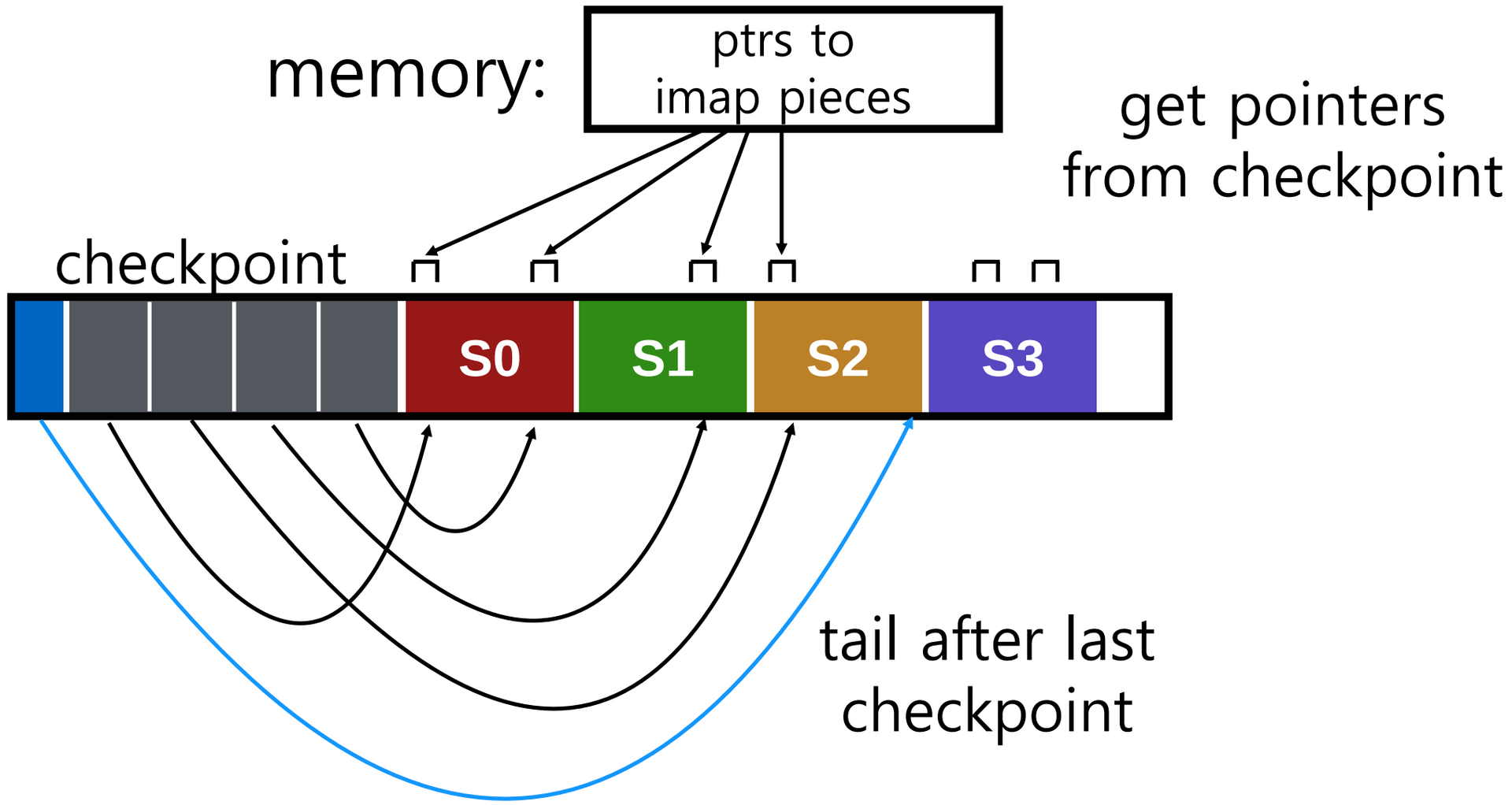


Crash!

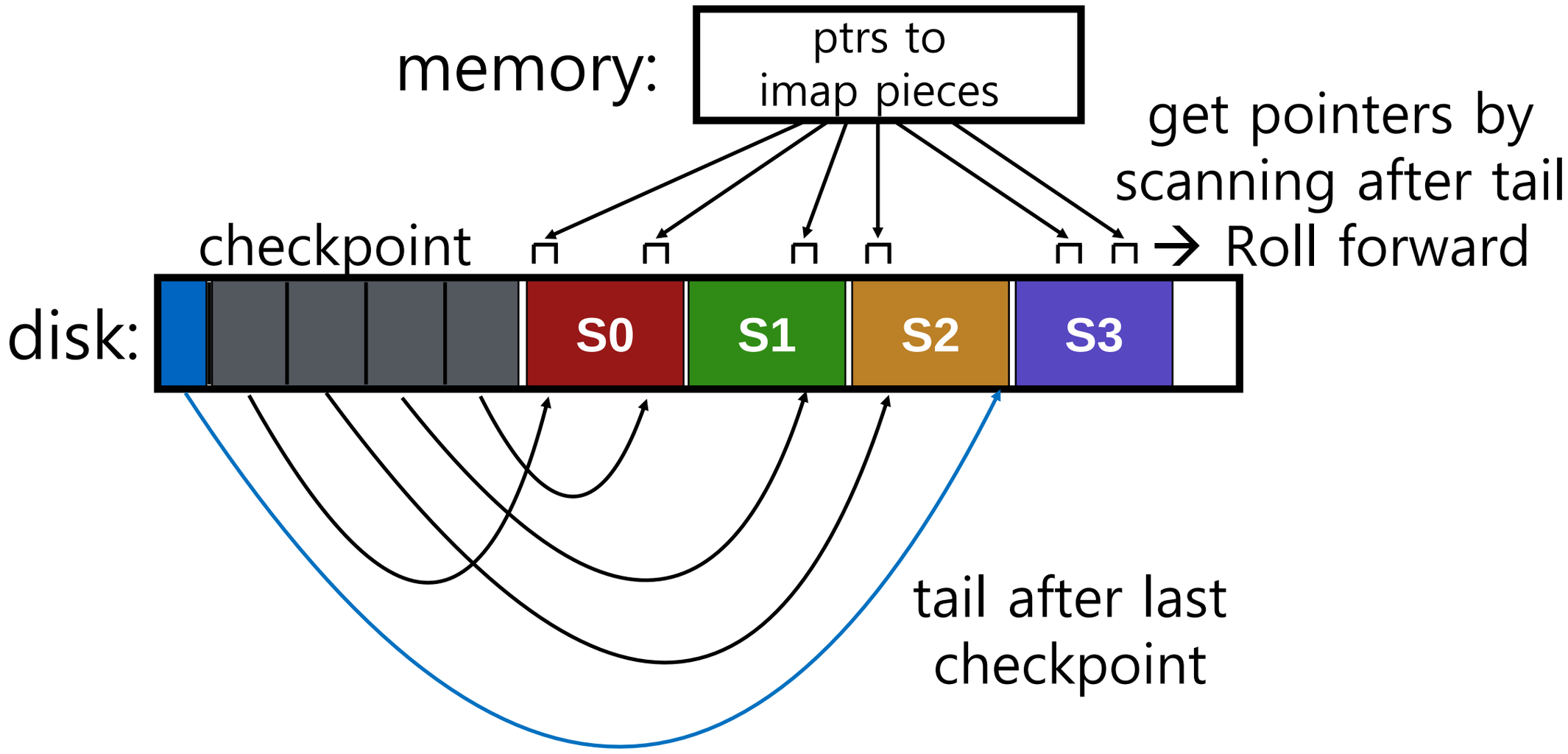
Reboot


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
S0
S1
S2
S3
tail after last
checkpoint

Reboot


memory:
ptrs to imap pieces
get pointers by scanning after tail → Roll forward
checkpoint
disk:
S0
S1
S2
S3
tail after last checkpoint

Checkpoint Summary

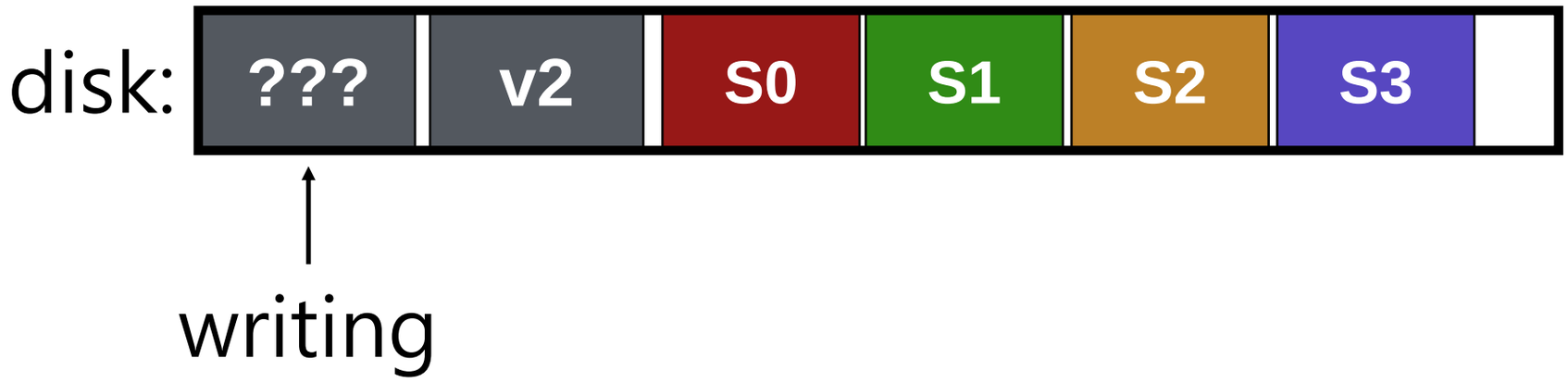
Checkpoint occasionally (e.g., every 30s)

Upon recovery:
- read checkpoint to find most imap pointers and s egment 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

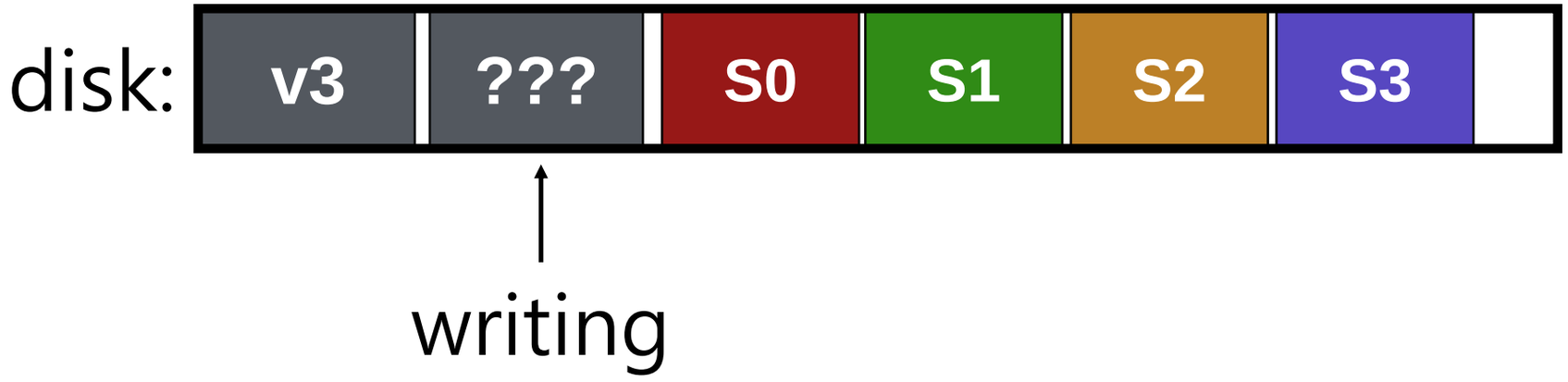
Use checksum/timestamps to identify newest checkpoint

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

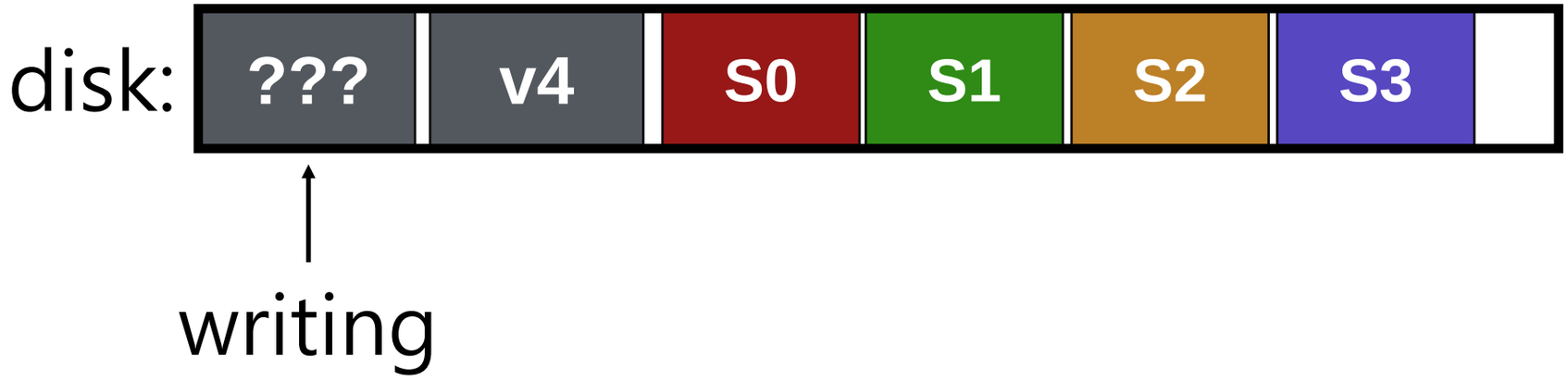
Use checksum/timestamps to identify newest checkpoint

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

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…

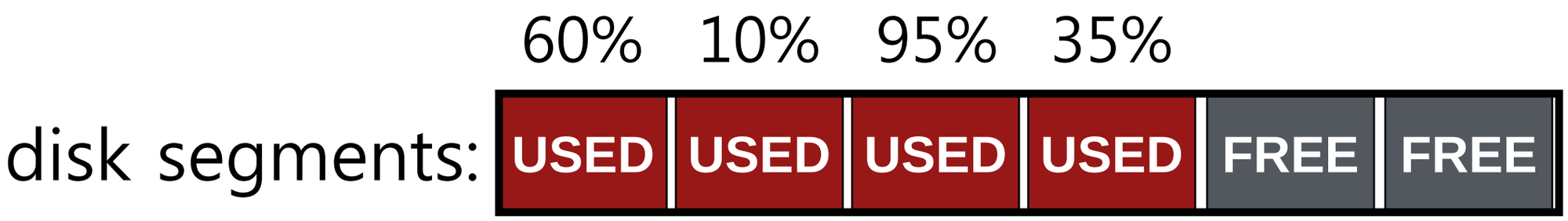
Garbage Collection

Need to reclaim space:
1. When no more references (any file system)
2. After newer copy is created (COW file system)

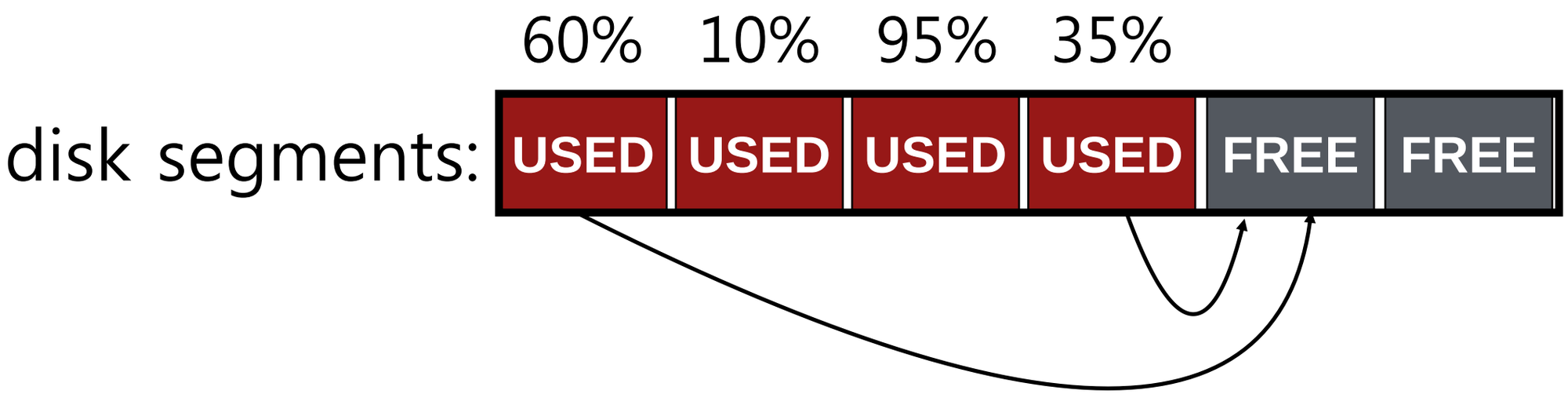
LFS reclaims **segments** (not individual inodes and d ata blocks)

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

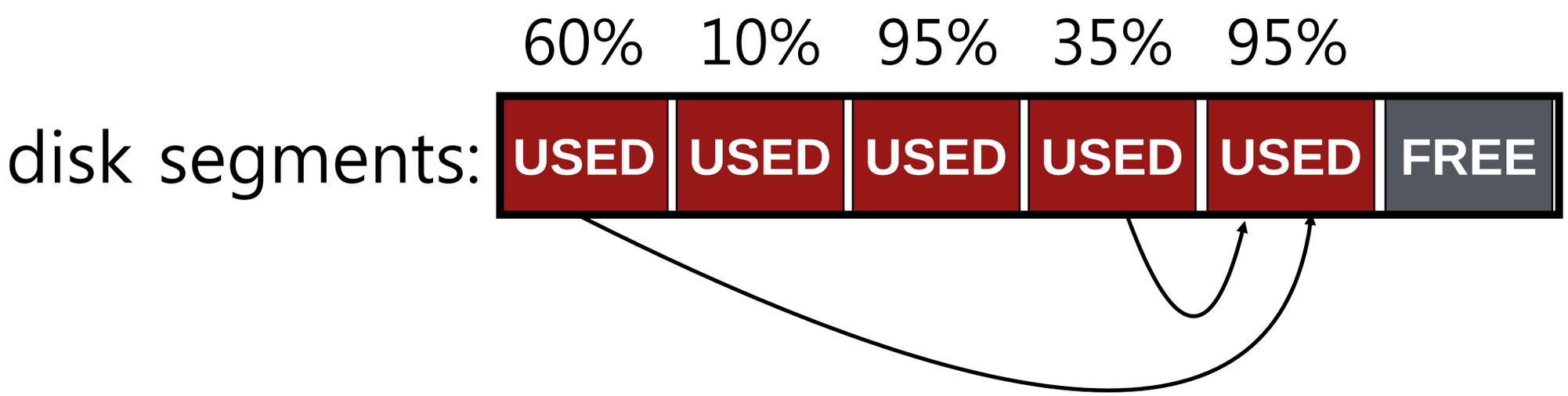
Garbage Collection



Garbage Collection



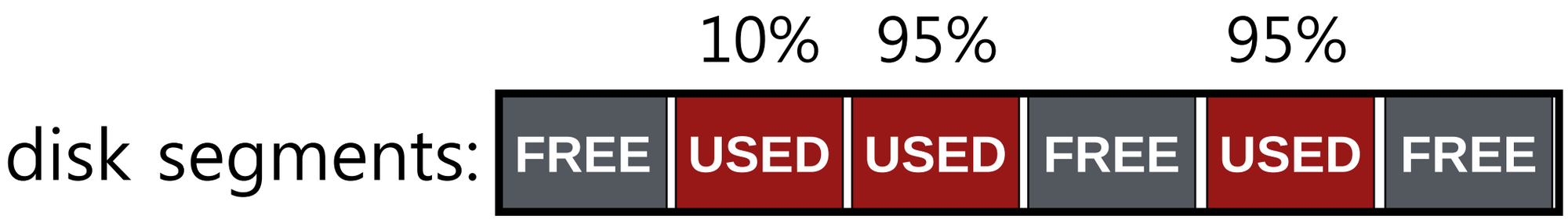
Garbage Collection



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?

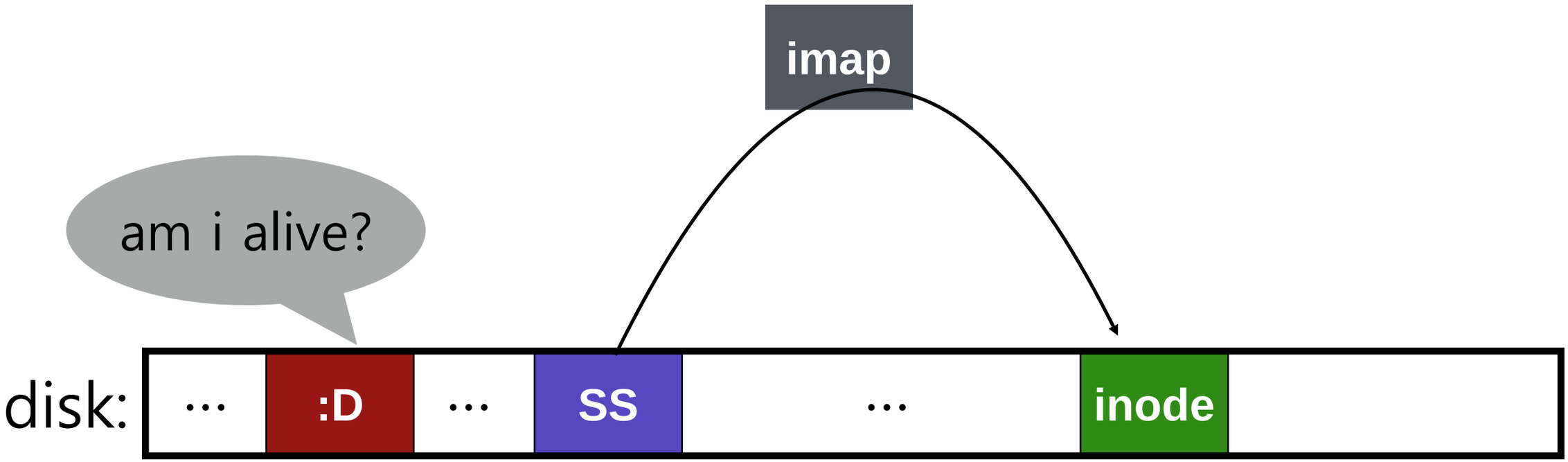
- Scan ALL inodes to see if any point to this data
- Very slow!

How to track information more efficiently?

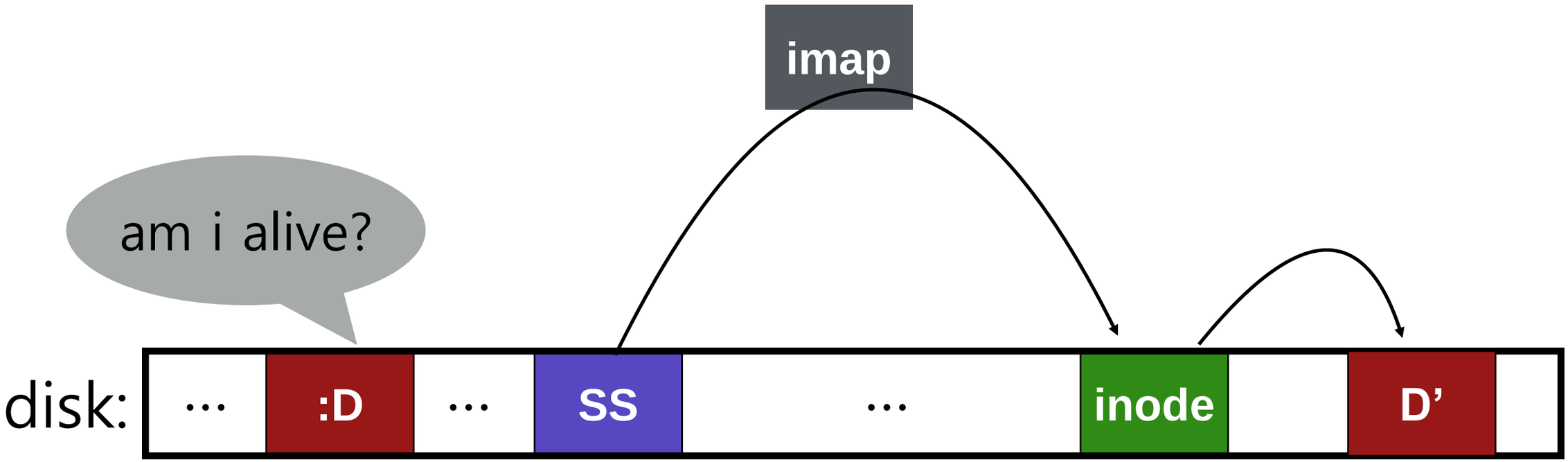
- **Segment summary** lists inode and data offset corresponding to each data block in segment (reverse pointers)

Block Liveness

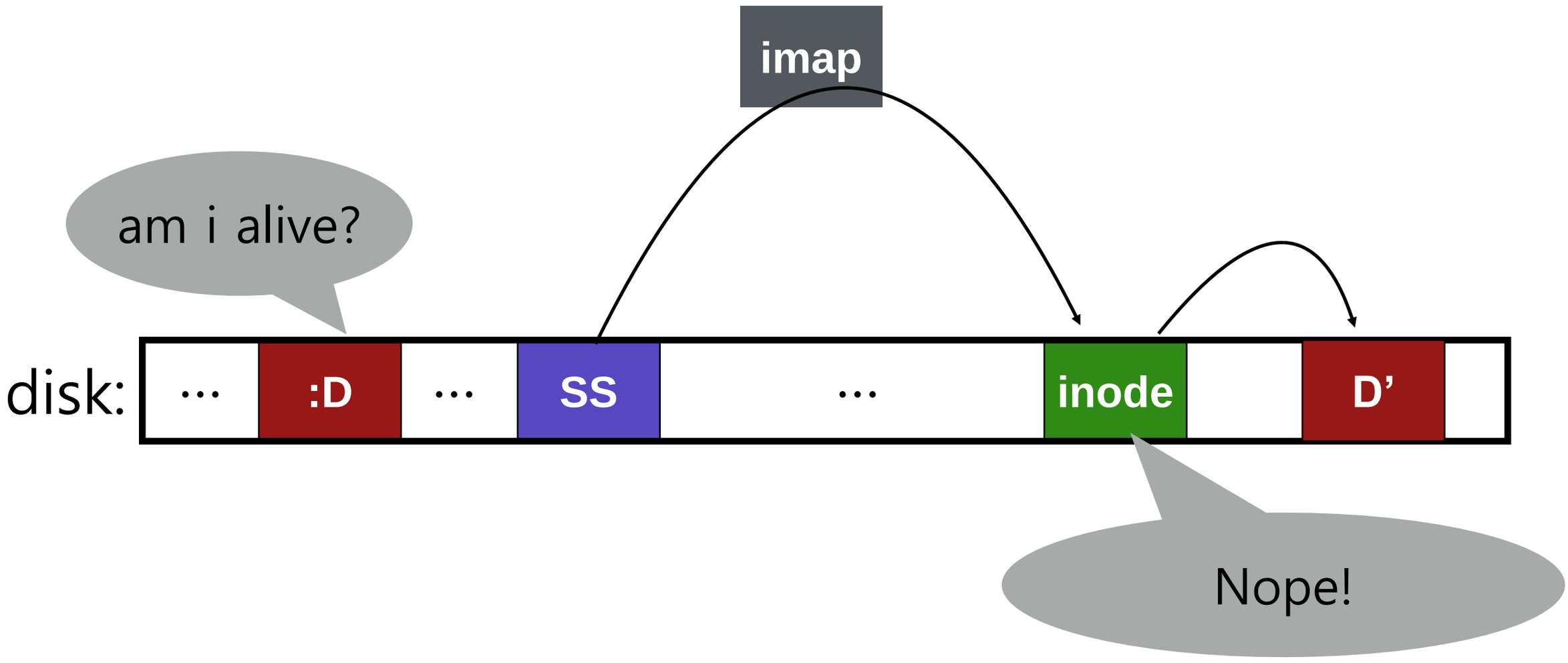
Block Liveness



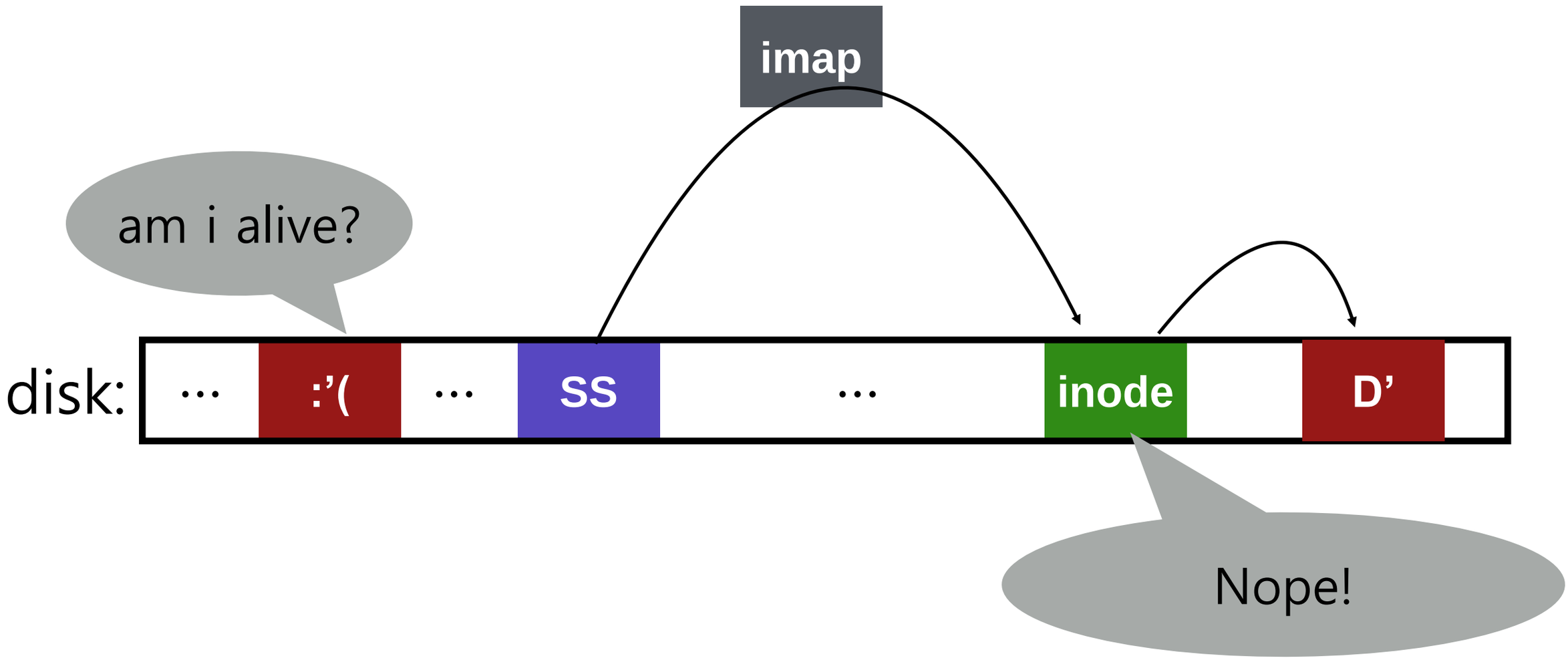
Block Liveness



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 v
alid? [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 choo ses (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