[537] LFS

Chapters 43-44

Tyler Harter

11/17/14

File-System Case Studies

Local
- **FFS**: Fast File System
- **LFS**: Log-Structured File System

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

File-System Case Studies

Local

- **FFS**: Fast File System
- **LFS**: Log-Structured File System [today]

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

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

General Strategy

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 at same time.

General Strategy

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 at same time.

Three techniques:
1. journal old, overwrite in place
2. journal new, overwrite in place
3. write new, discard old [today]

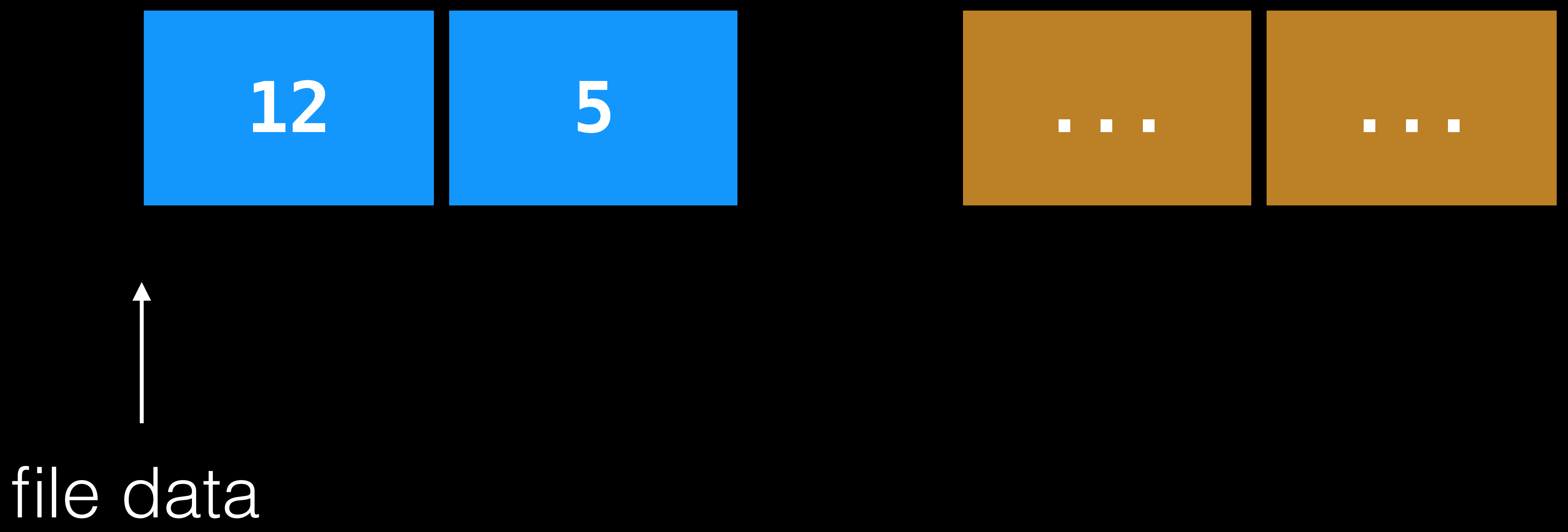
General Strategy

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

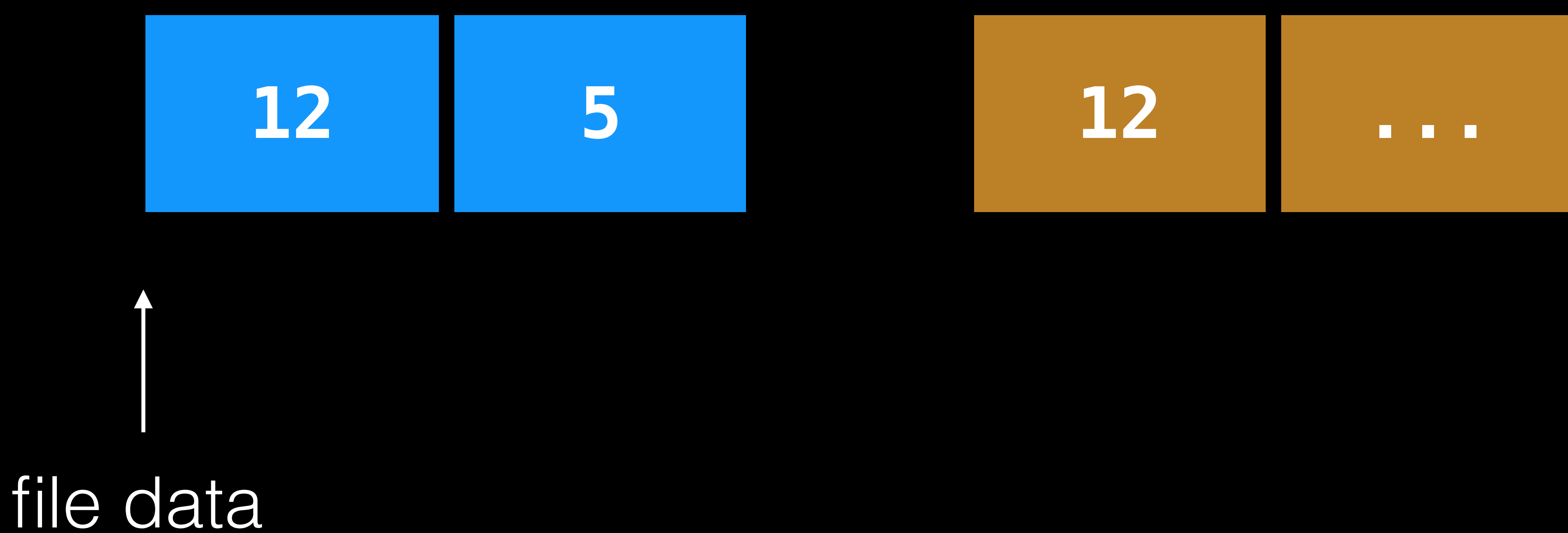
Implication: at some point in time, all old AND
all new data must be on disk at same time.

Three techniques:
1. journal old, overwrite in place
2. journal new, overwrite in place
3. write new, discard old [today]

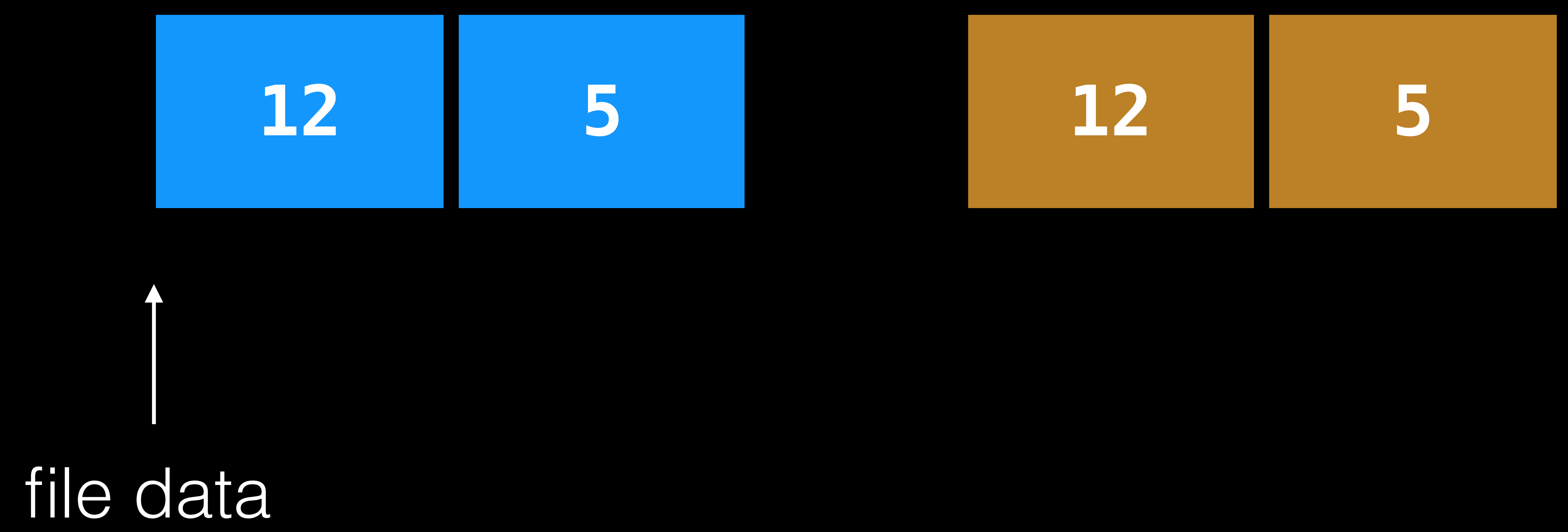
1. Journal Old, Overwrite In-Place



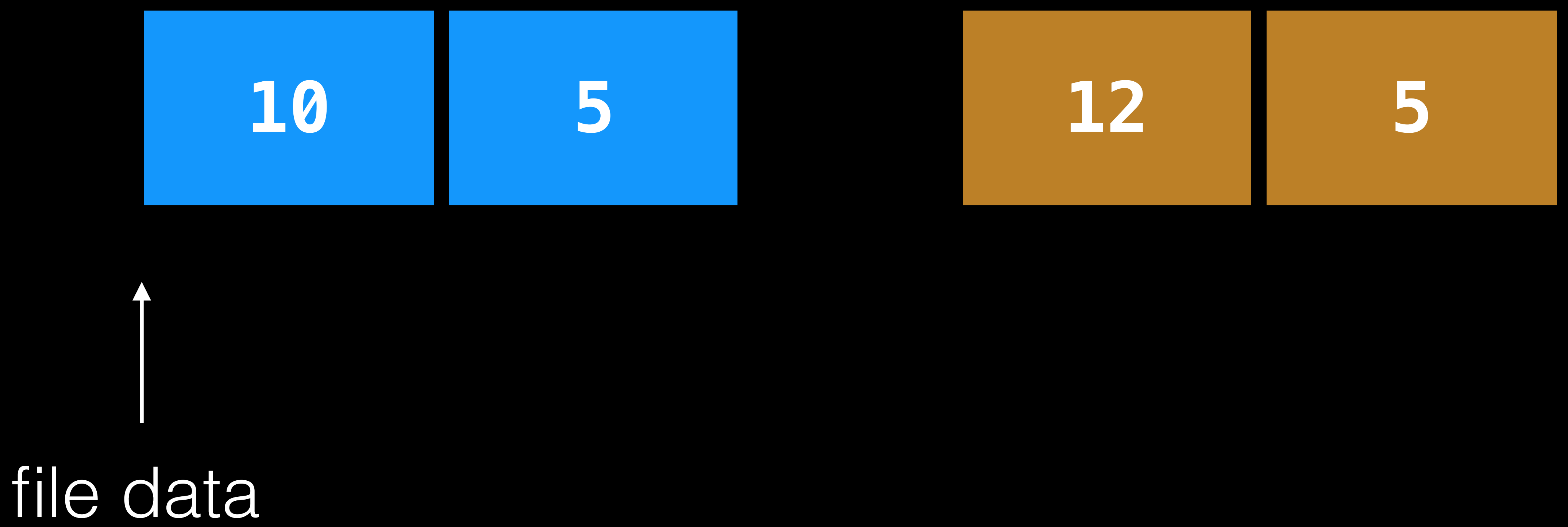
1. Journal Old, Overwrite In-Place



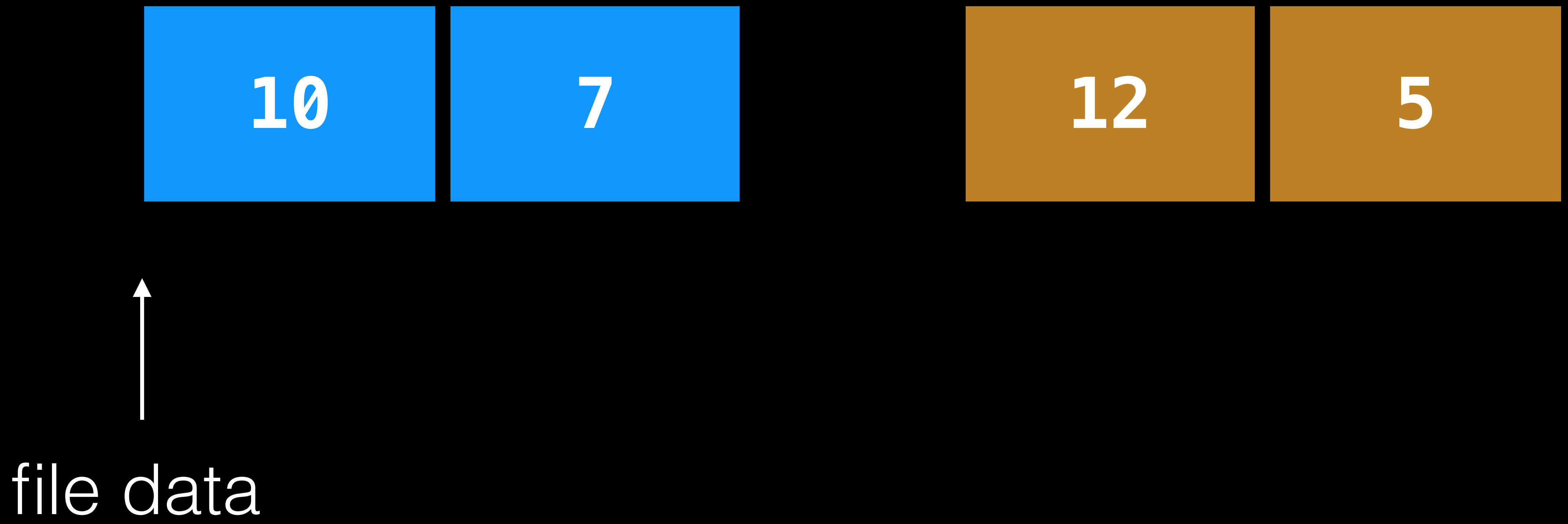
1. Journal Old, Overwrite In-Place



1. Journal Old, Overwrite In-Place



1. Journal Old, Overwrite In-Place



1. Journal Old, Overwrite In-Place

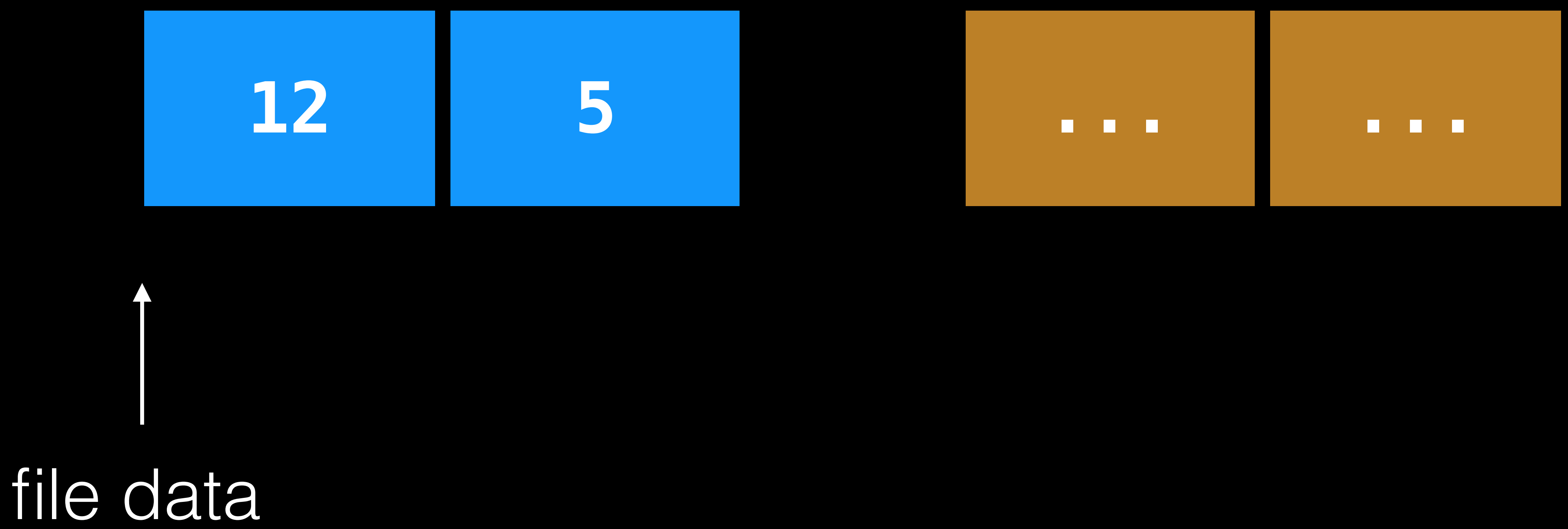
General Strategy

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

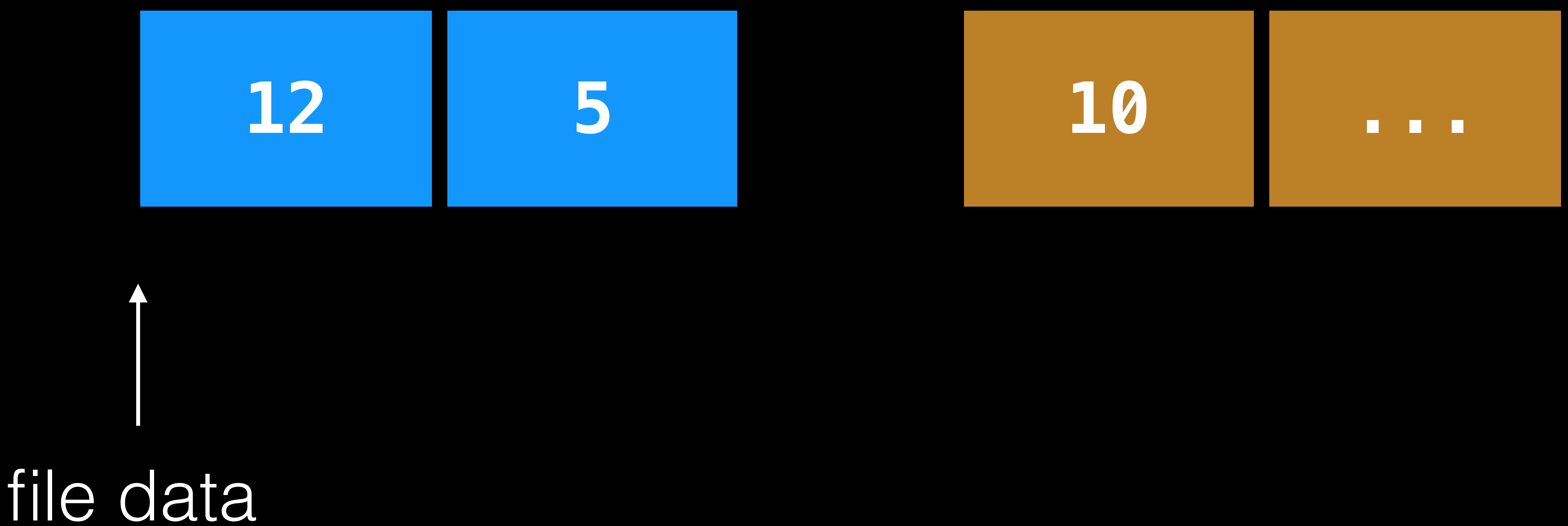
Implication: at some point in time, all old AND
all new data must be on disk at same time.

Three techniques:
1. journal old, overwrite in place
2. journal new, overwrite in place
3. write new, discard old [today]

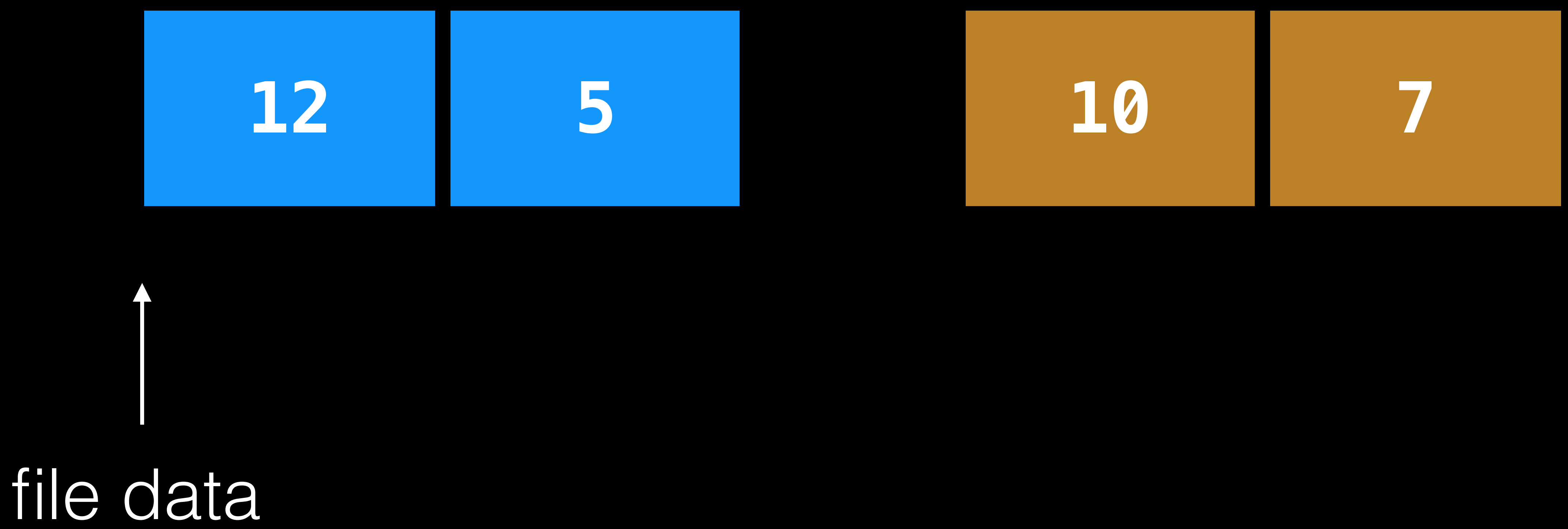
2. Journal New, Overwrite In-Place



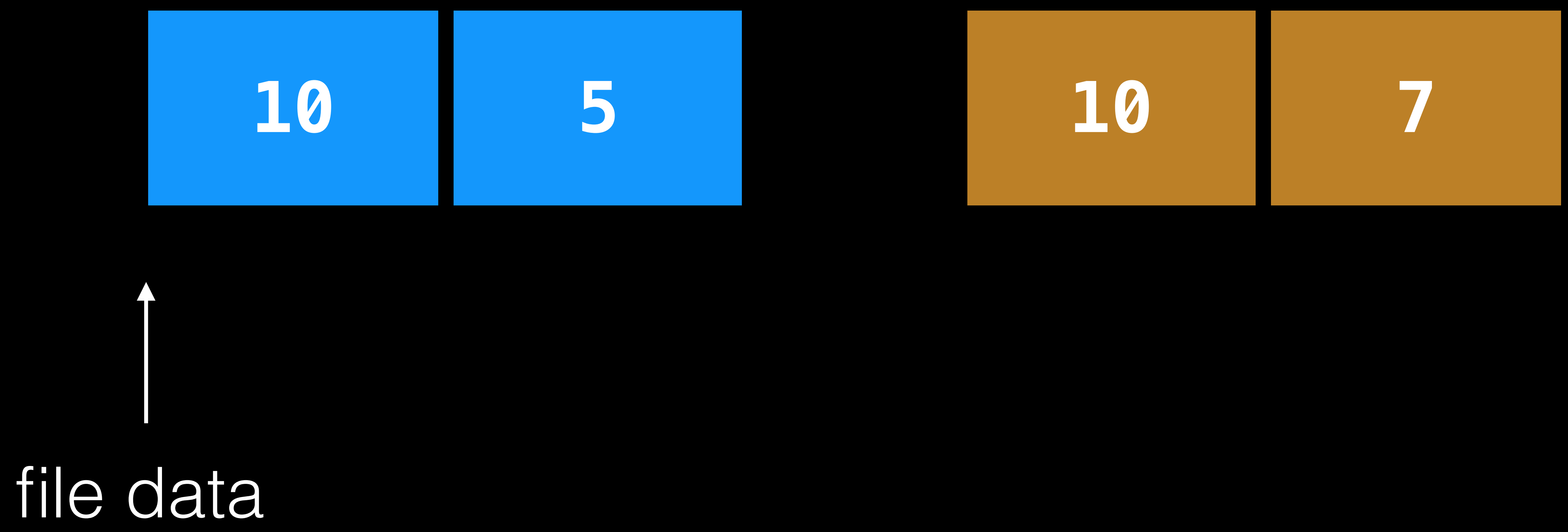
2. Journal New, Overwrite In-Place



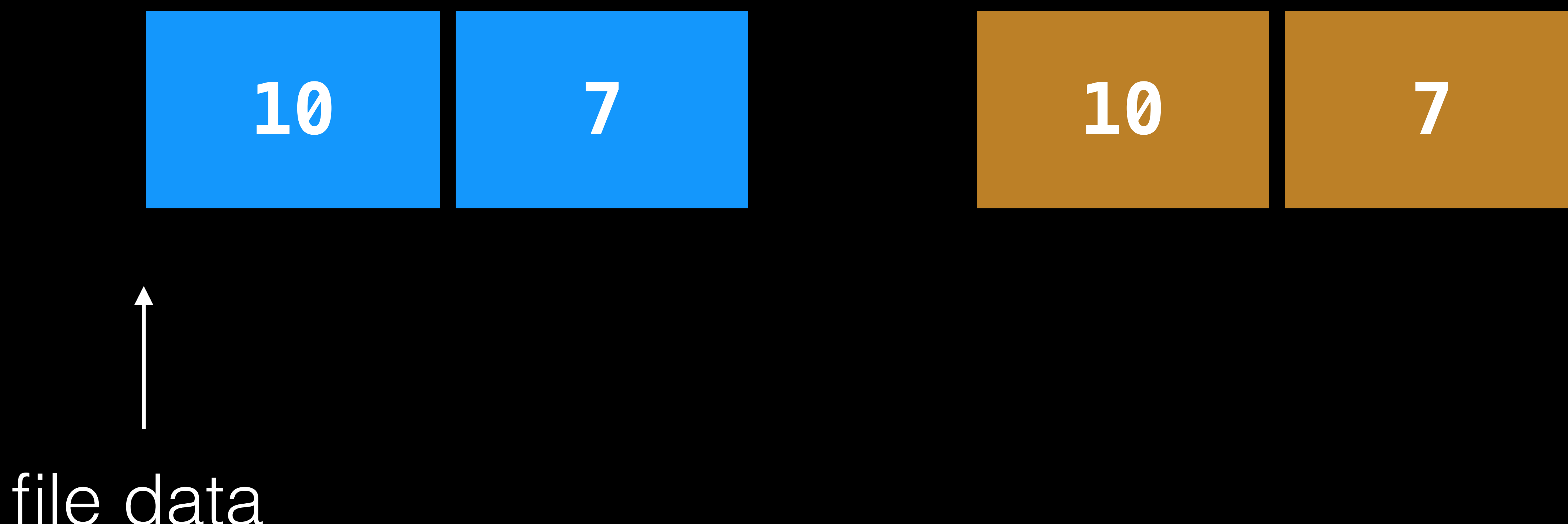
2. Journal New, Overwrite In-Place



2. Journal New, Overwrite In-Place



2. Journal New, Overwrite In-Place



2. Journal New, Overwrite In-Place

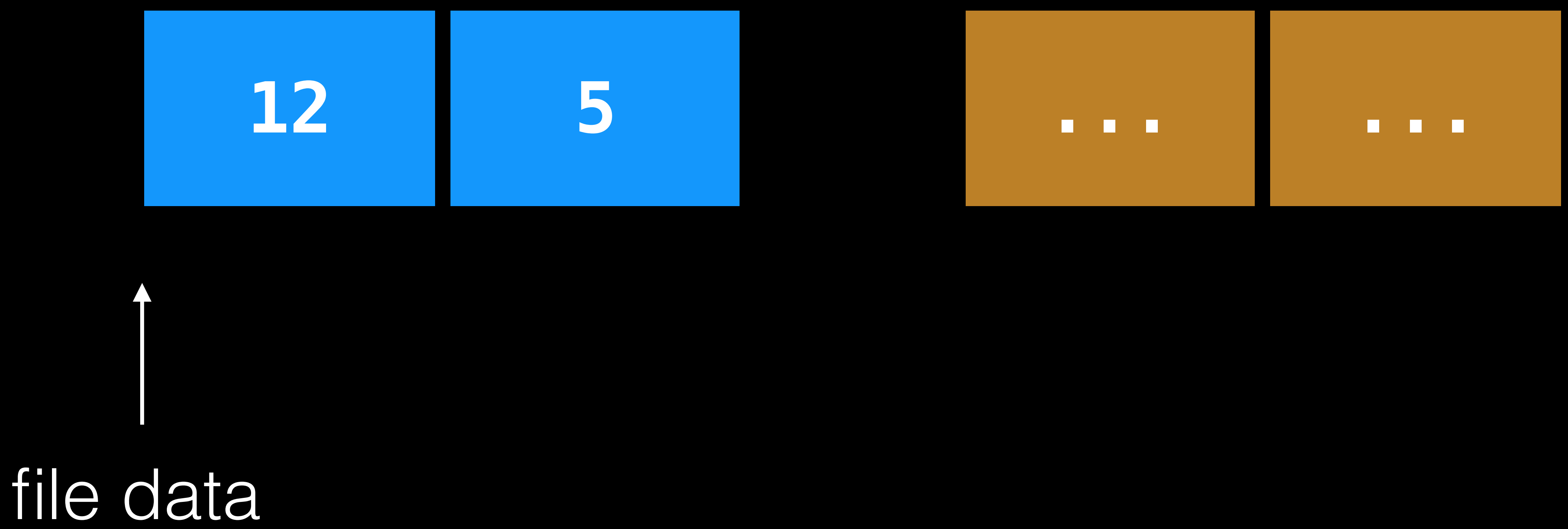
General Strategy

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

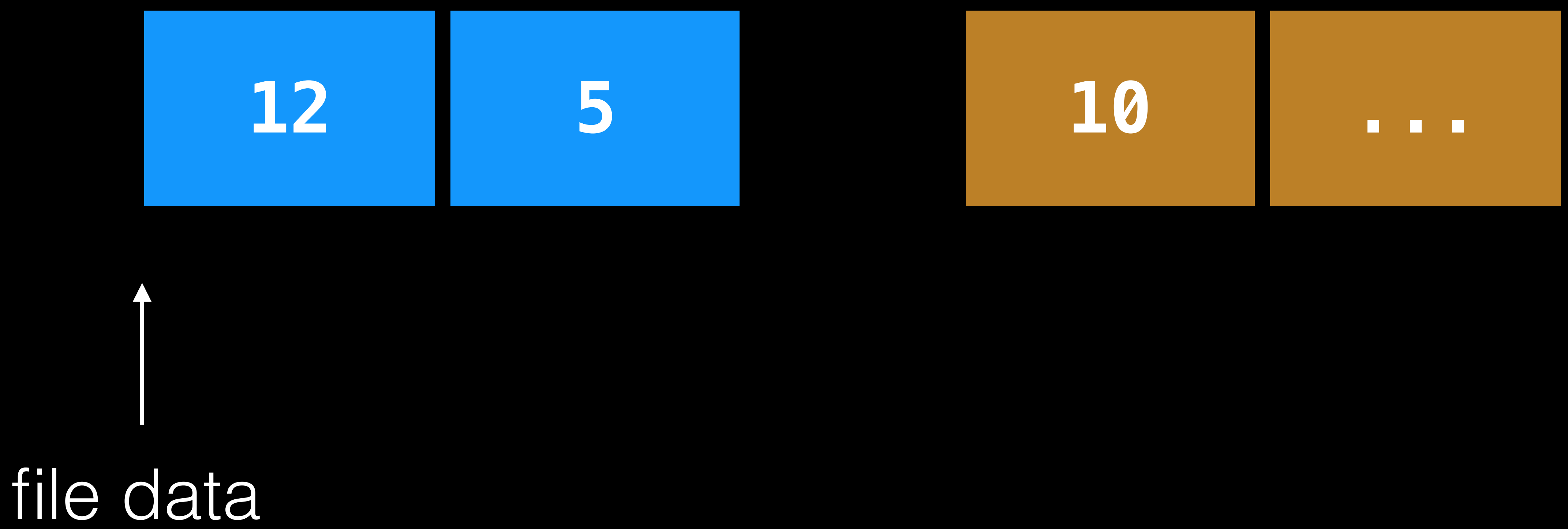
Implication: at some point in time, all old AND
all new data must be on disk at same time.

Three techniques:
1. journal old, overwrite in place
2. journal new, overwrite in place
3. write new, discard old [today]

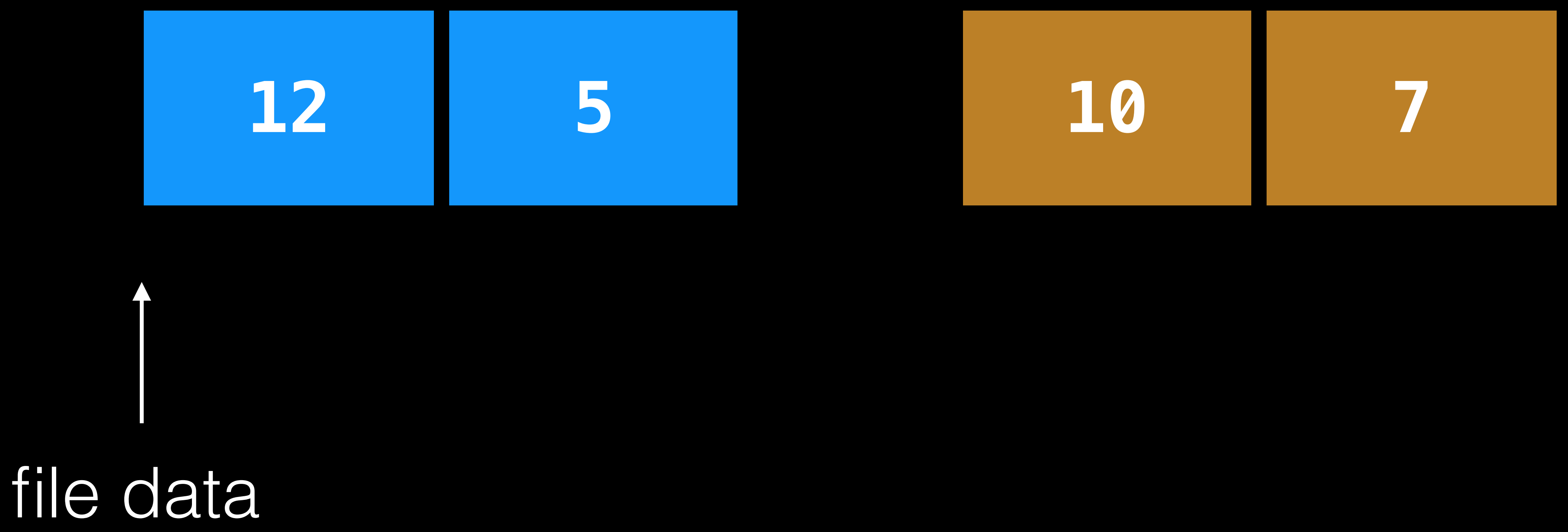
3. Write New, Discard Old



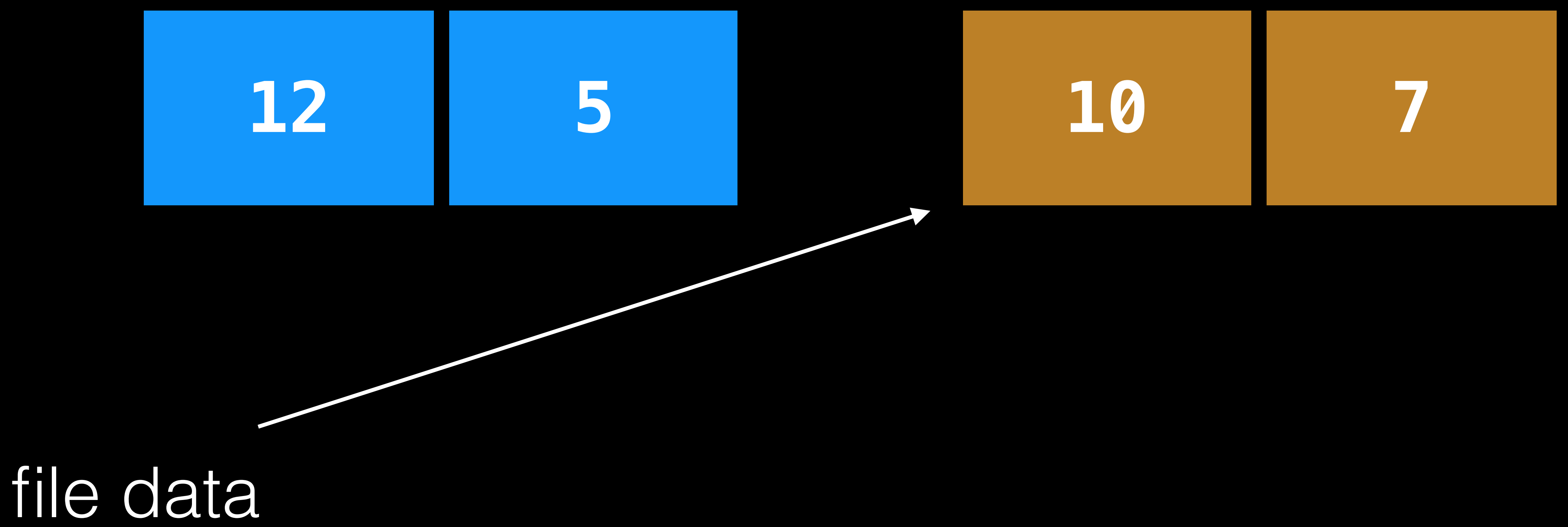
3. Write New, Discard Old



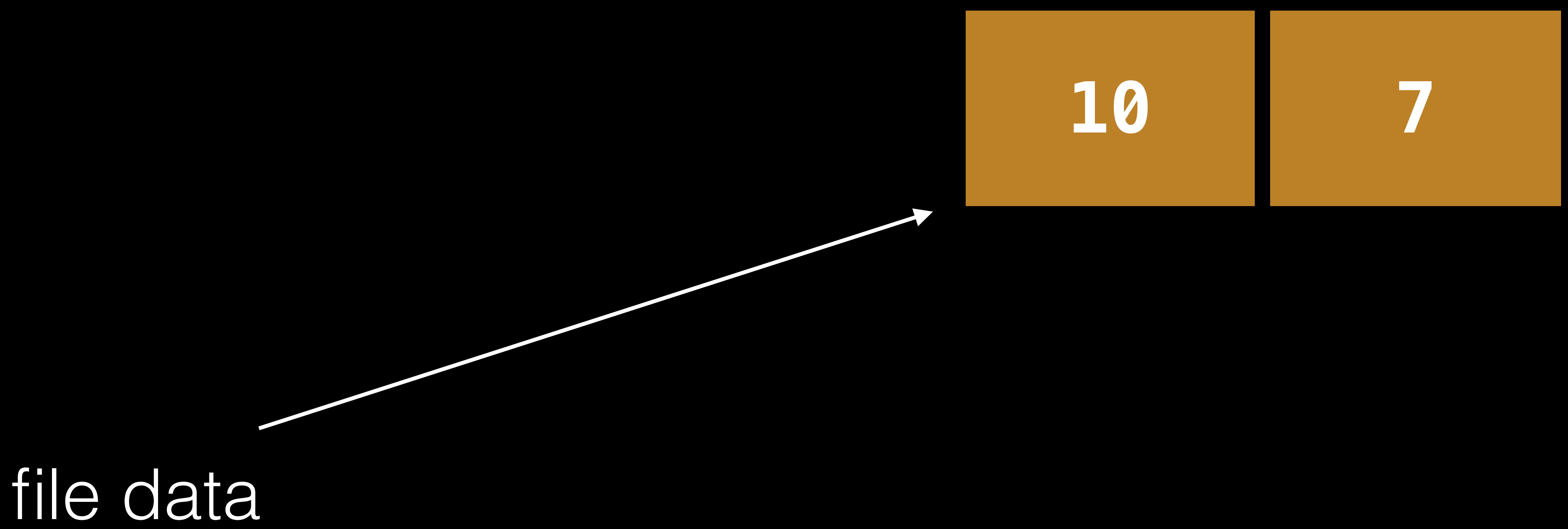
3. Write New, Discard Old



3. Write New, Discard Old



3. Write New, Discard Old



General Strategy

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 at same time.

Three techniques:
1. journal old, overwrite in place
2. journal new, overwrite in place
3. write new, discard old [today]

General Strategy

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

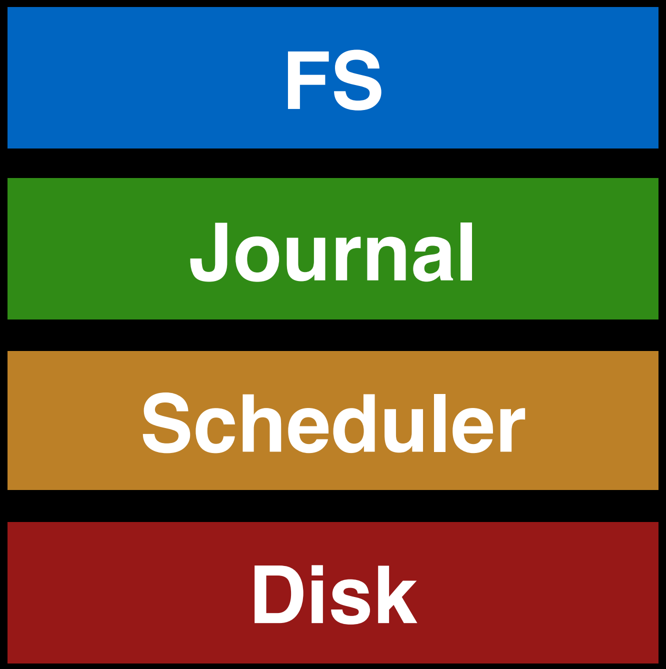
Implication: at some point in time, all old AND
all new data must be on disk at same time.

Three techniques:
1. journal old, overwrite in place
2. journal new, overwrite in place -- do exercise 1 (worksheet)
3. write new, discard old [today]

File System Integration

Observation: some data (e.g., user data) is less important.

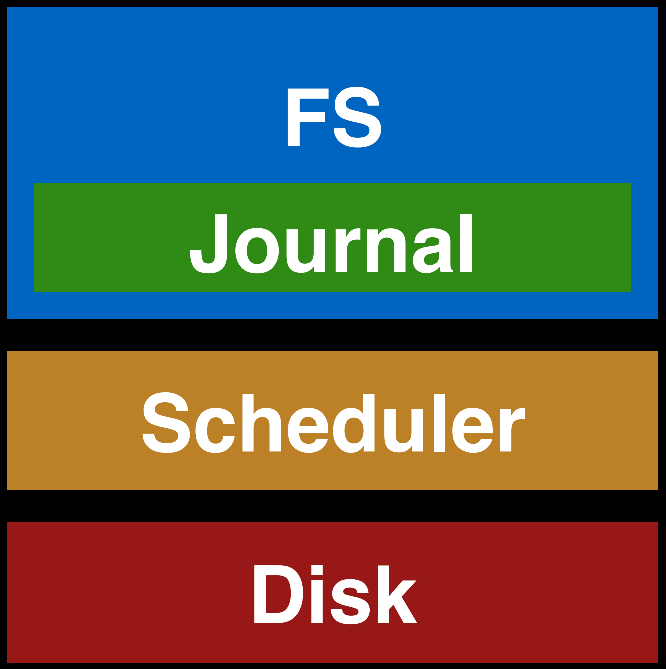
If we want to only journal FS metadata, we need tighter integration.



File System Integration

Observation: some data (e.g., user data) is less important.

If we want to only journal FS metadata, we need tighter integration.



Writeback Journal

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

For regular data, write it back whenever it's convenient. Of course, files may contain garbage.

Writeback Journal

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

For regular data, write it back whenever it's convenient. Of course, files may contain garbage.

What is the worst type of garbage we could get?

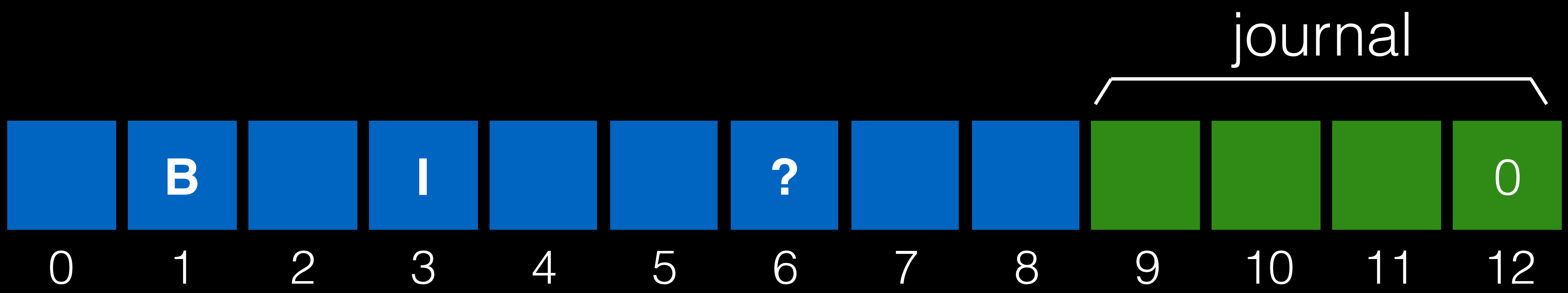
Writeback Journal

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

For regular data, write it back whenever it's convenient. Of course, files may contain garbage.

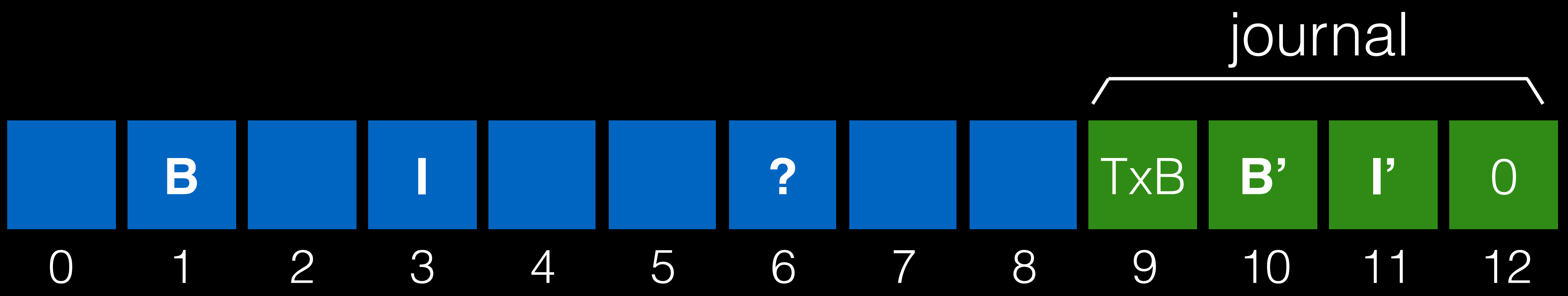
What is the worst type of garbage we could get? How to avoid?

Writeback Journal



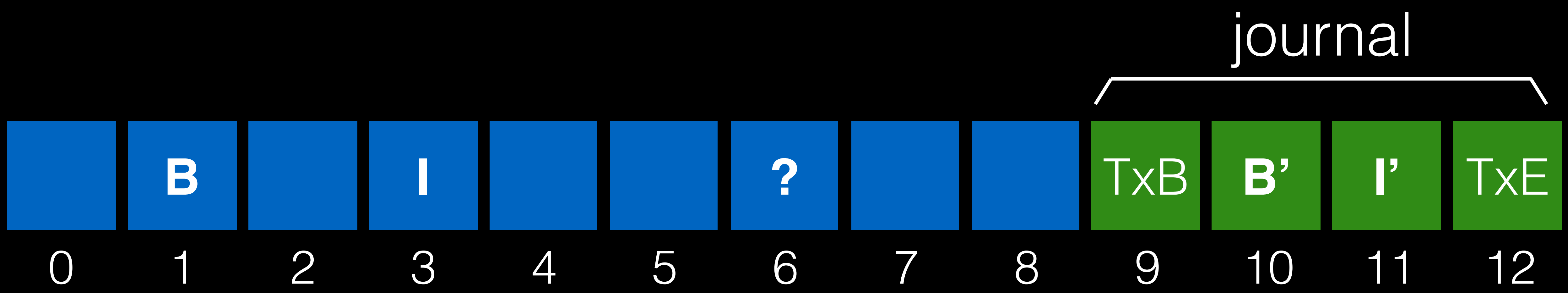
transaction: append to inode I

Writeback Journal



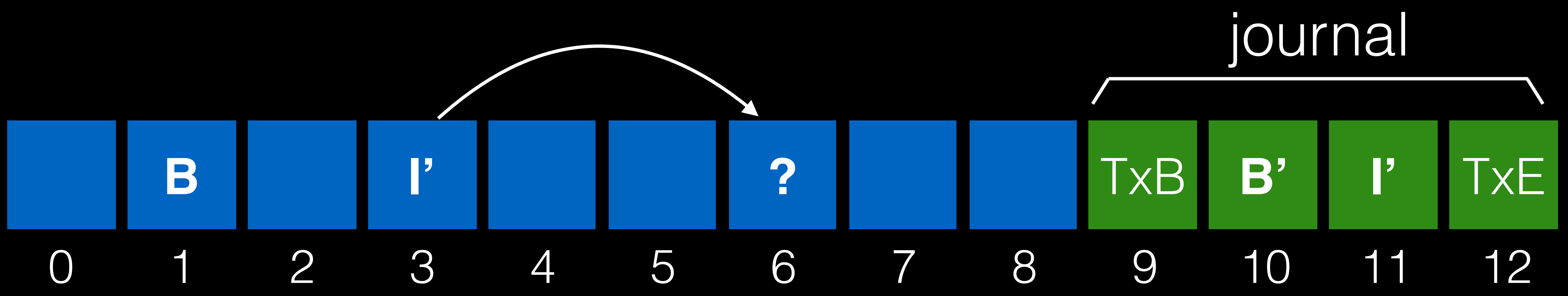
transaction: append to inode I

Writeback Journal



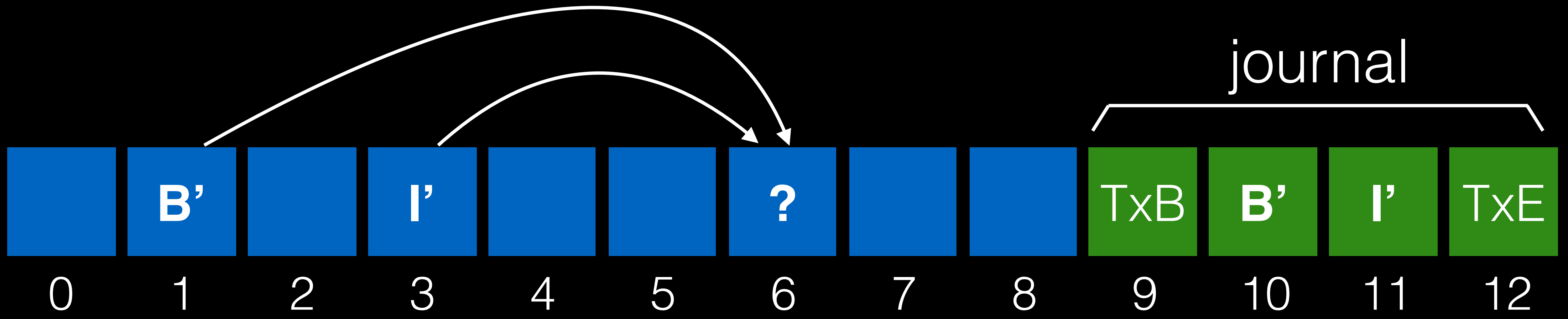
transaction: append to inode I

Writeback Journal



transaction: append to inode I

Writeback Journal



transaction: append to inode I

what if we crash now? Solutions?

Ordered Journaling

Still only journal metadata.

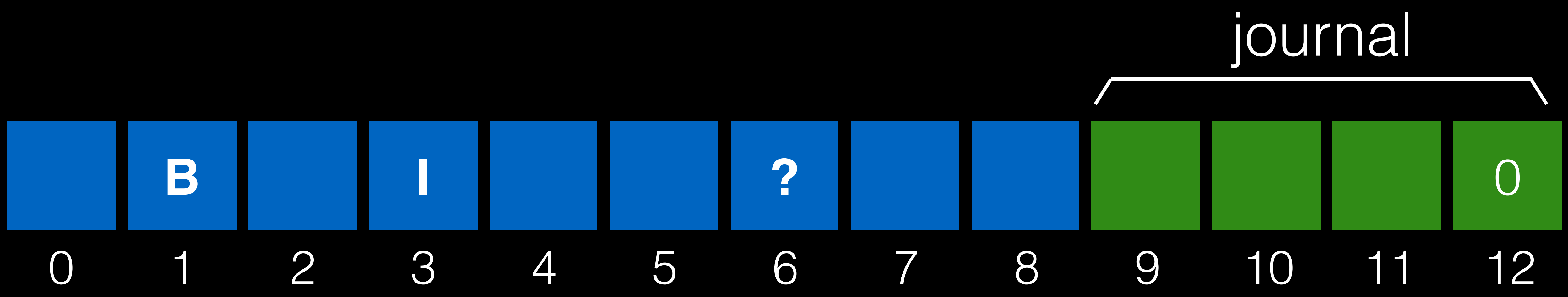
But write data before the transaction.

May still get scrambled data on update.

But appends will always be good.

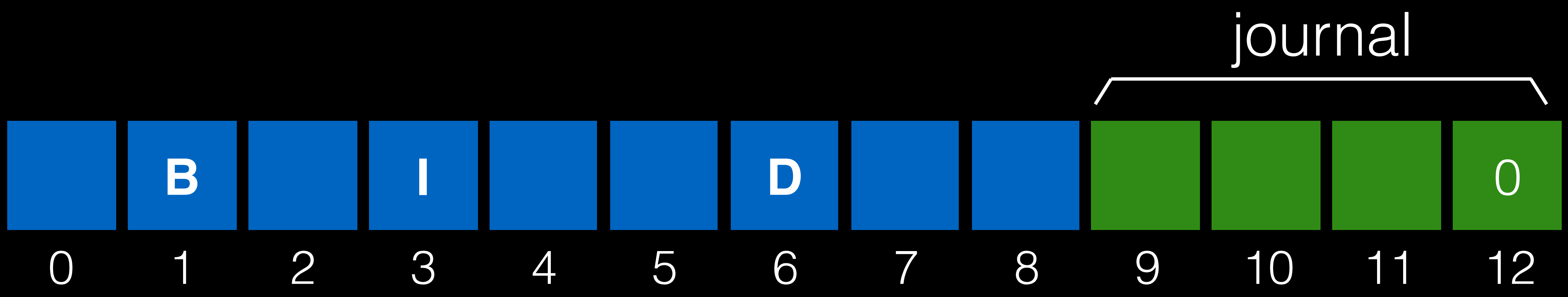
No leaks of sensitive data!

Ordered Journal



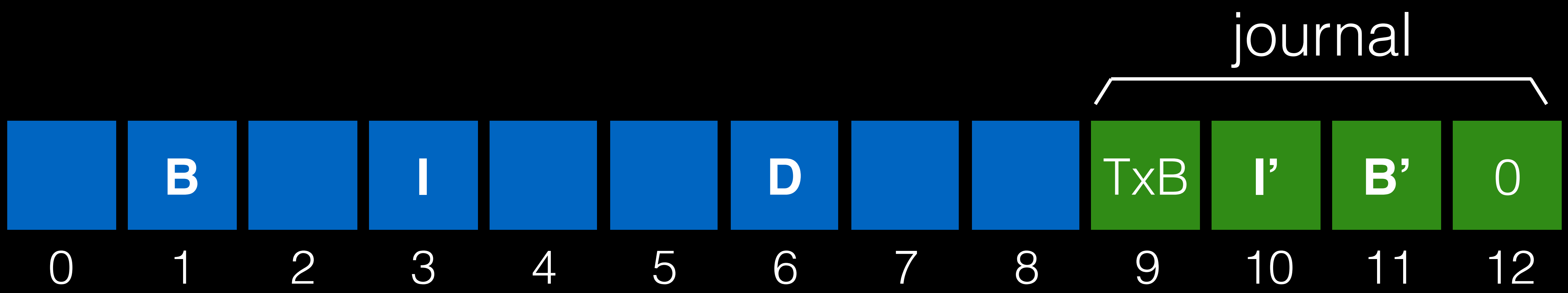
transaction: append to inode I

Ordered Journal



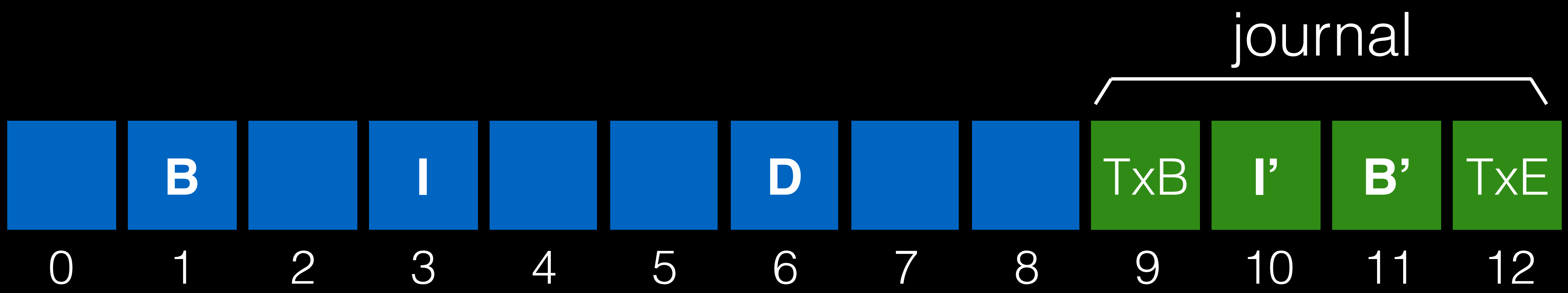
transaction: append to inode I

Ordered Journal



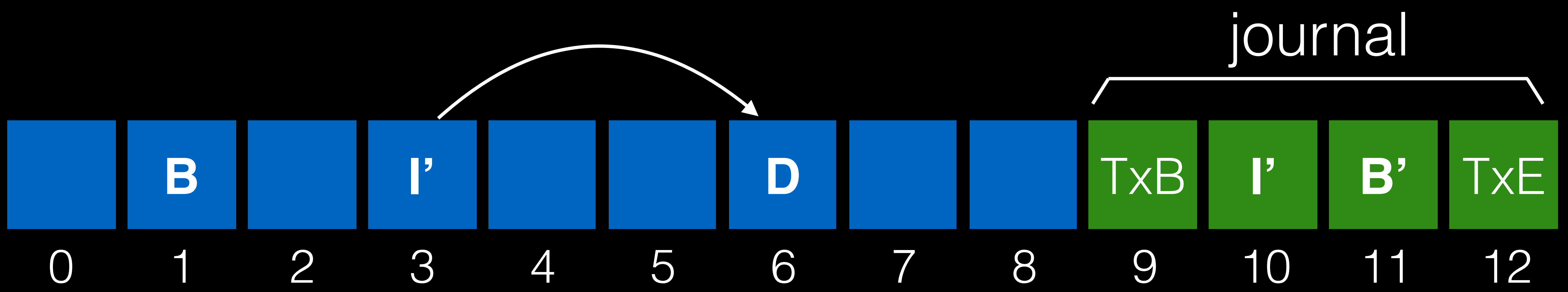
transaction: append to inode I

Ordered Journal



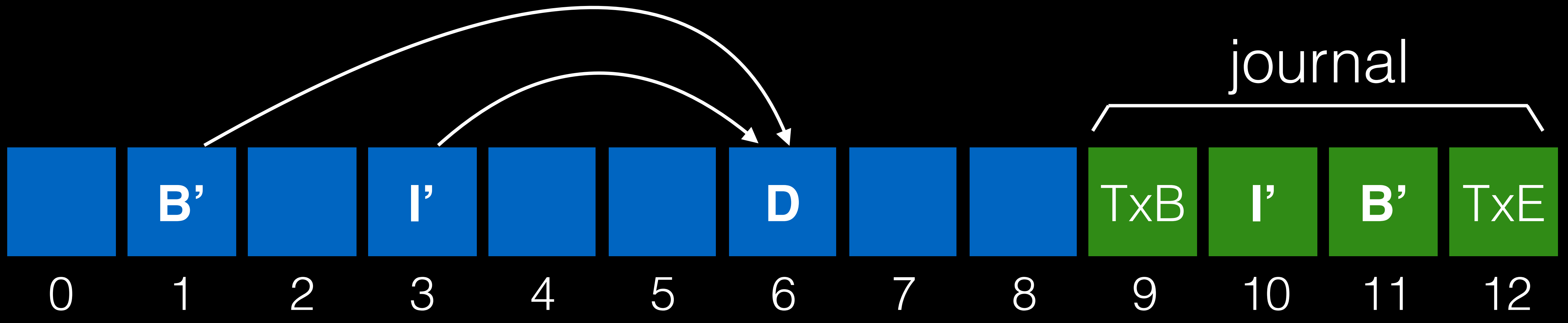
transaction: append to inode I

Ordered Journal



transaction: append to inode I

Ordered Journal



transaction: append to inode I

Log-Structured File System

Copy On Write (COW)

Do problem 2.

LFS: Log-Structured File System

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

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

Performance Goal

Ideal: use disk purely sequentially.

Performance Goal

Ideal: use disk purely sequentially.

Hard for reads -- why?

Easy for writes -- why?

Performance Goal

Ideal: use disk purely sequentially.

Hard for reads -- why?
 - user might read files X and Y not near each other

Easy for writes -- why?
 - can do all writes near each other to empty space

Observations

Memory sizes are growing (so cache more reads).

Growing gap between sequential and random I/O performance.

Existing file systems not RAID-aware (don't avoid small writes).

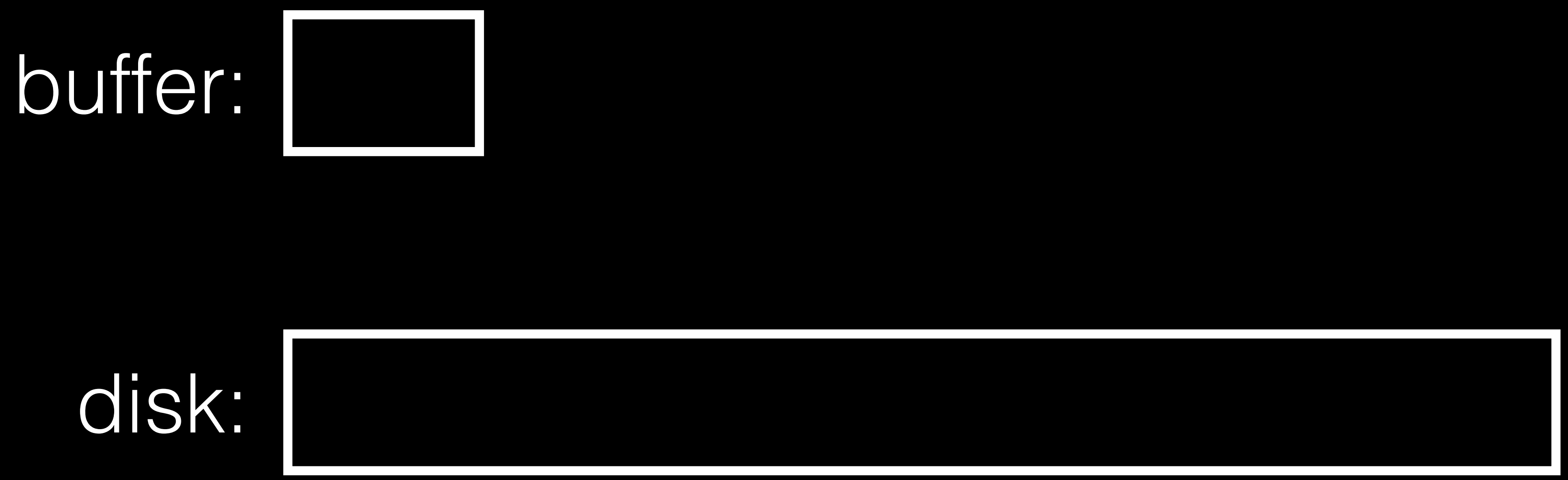
LFS Strategy

Just write all data sequentially to new segments.

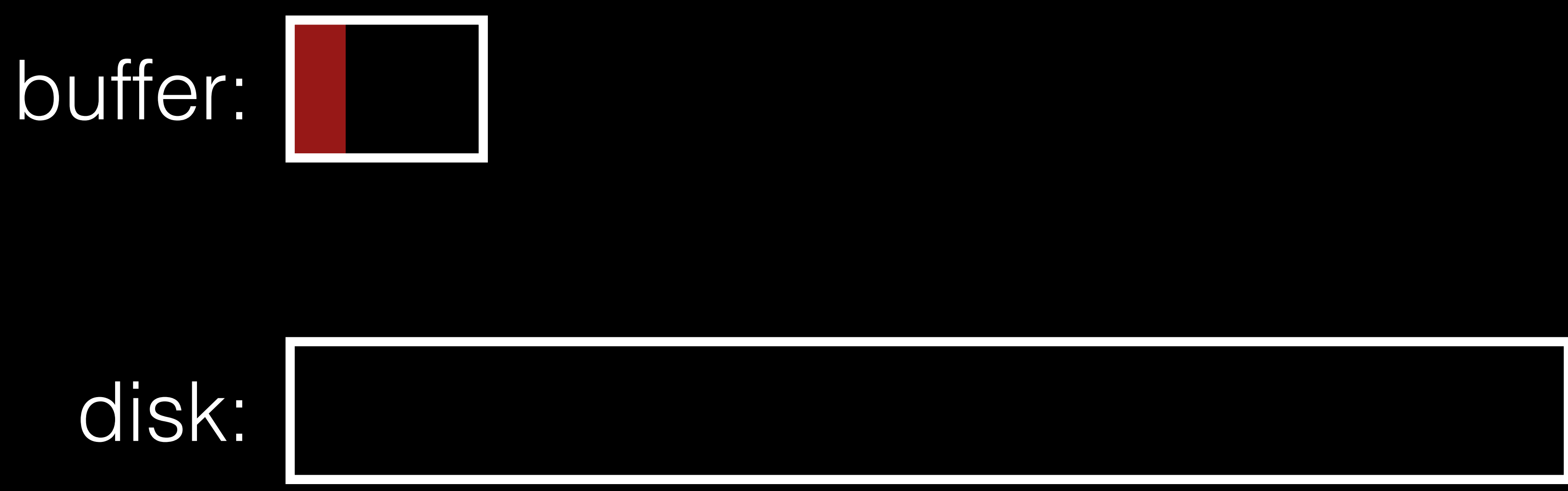
Never overwrite, even if that means we leave behind old copies.

Buffer writes until we have enough data.

Big Picture

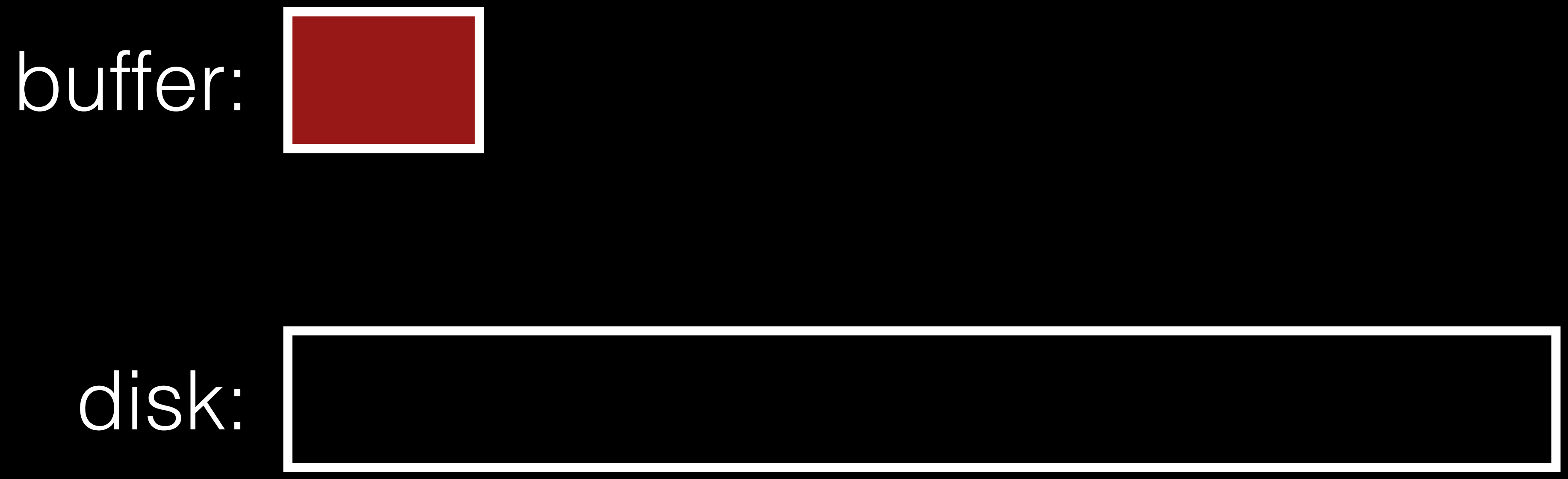


Big Picture

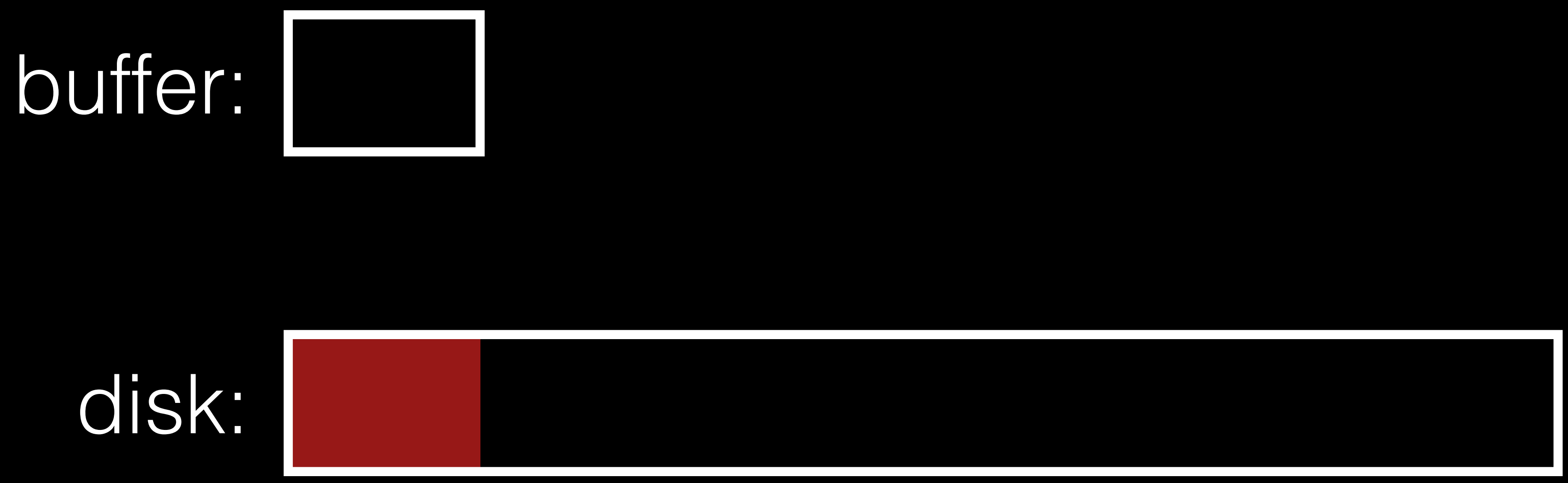


Big Picture

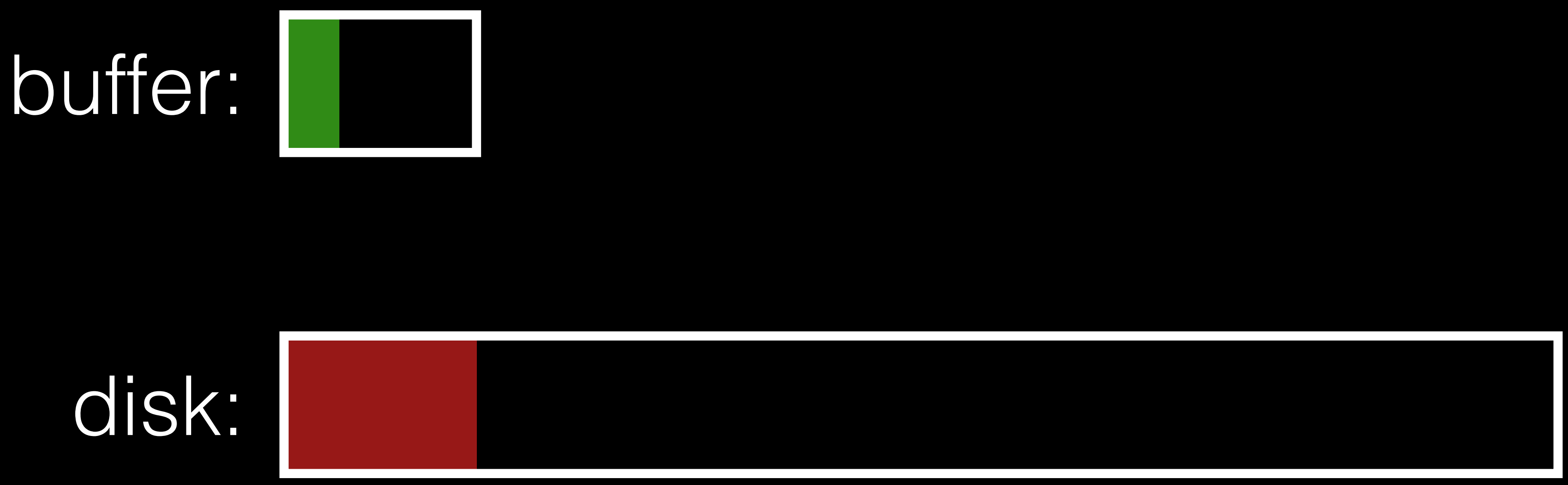
Big Picture



Big Picture

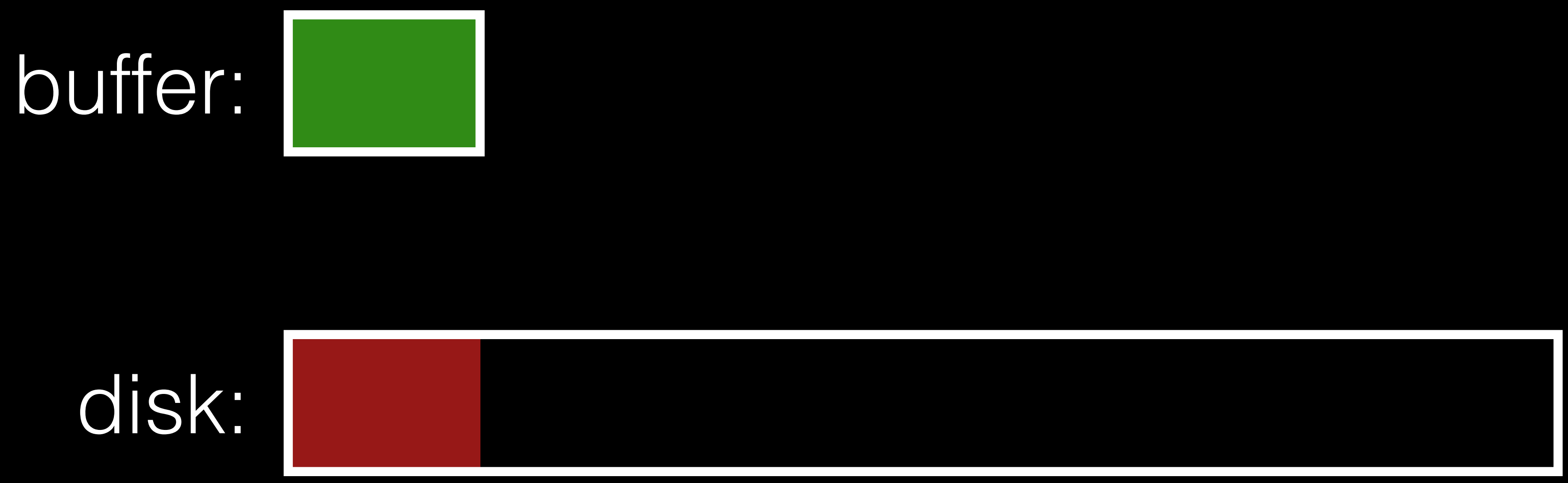


Big Picture



Big Picture

Big Picture



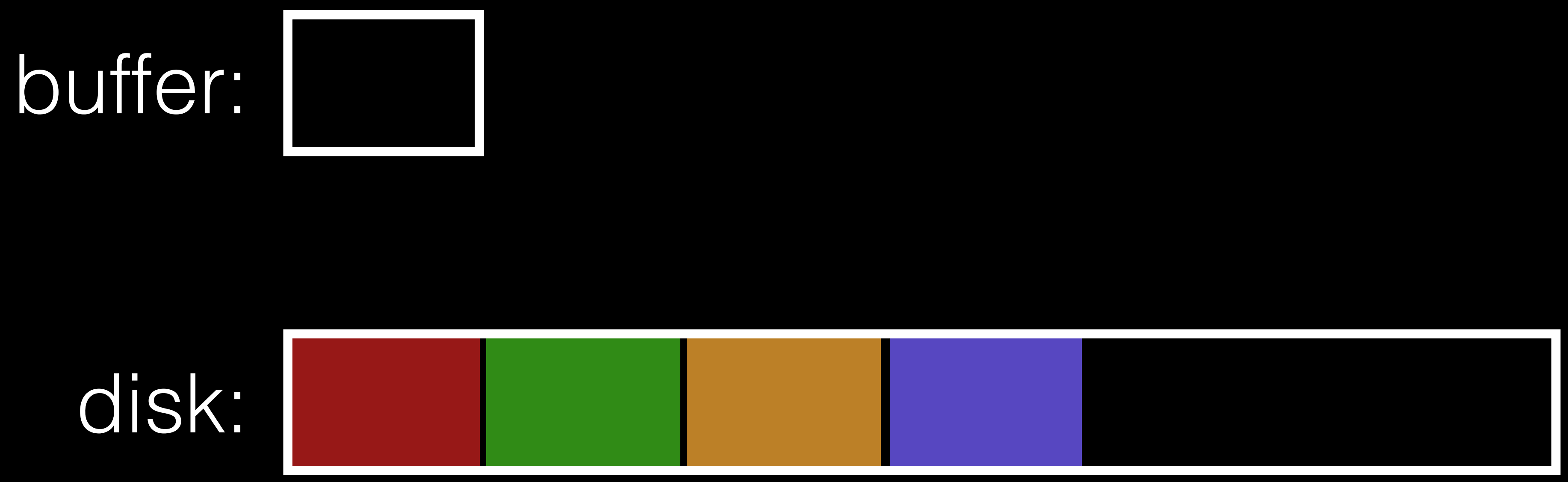
Big Picture

Big Picture

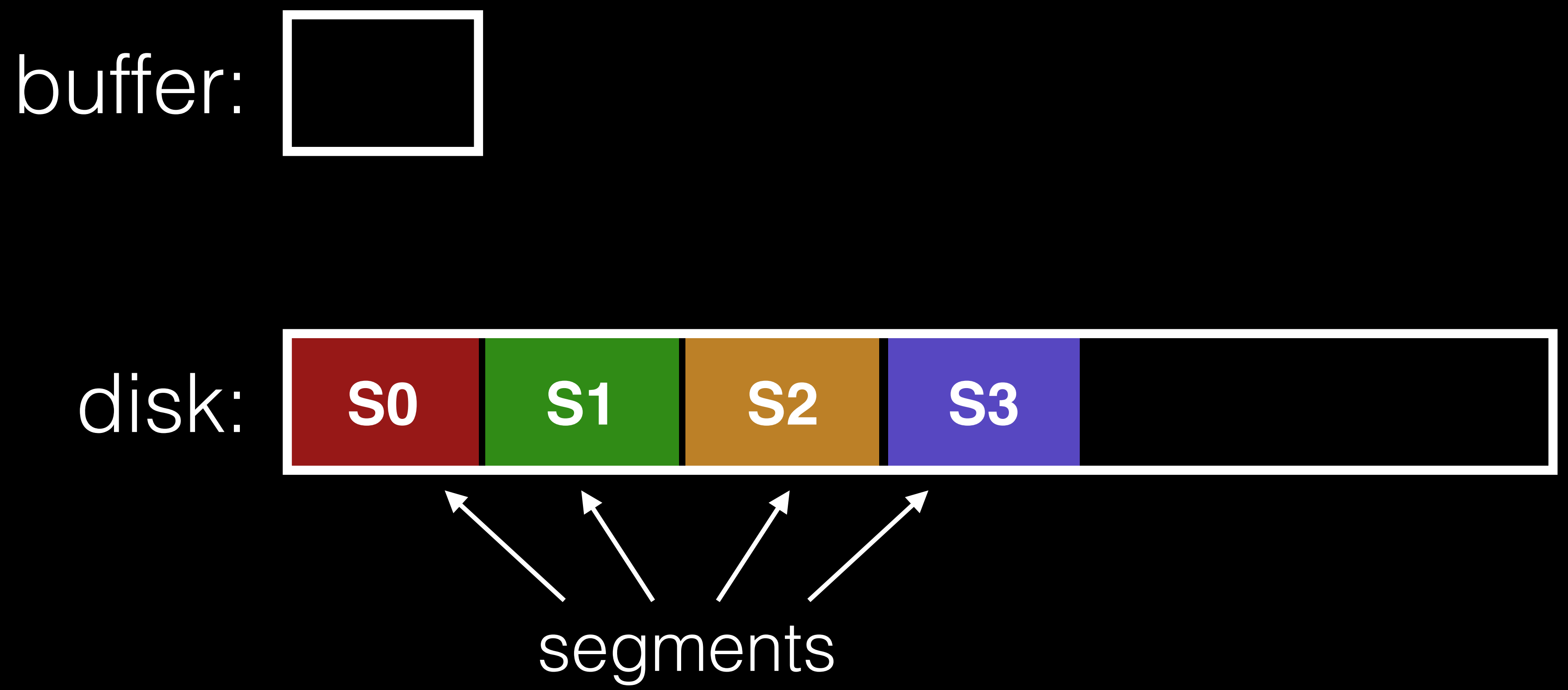
Big Picture

Big Picture

Big Picture



Big Picture



Data Structures (attempt 1)

What can we get rid of from FFS?

Data Structures (attempt 1)

What can we get rid of from FFS?

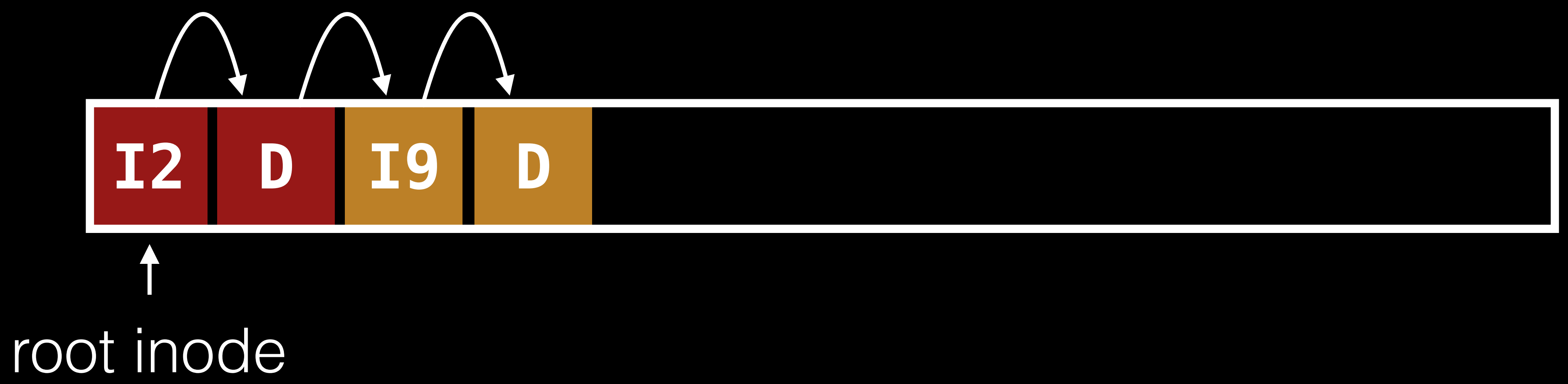
- allocation structs: data + inode bitmaps

Data Structures (attempt 1)

What can we get rid of from FFS?
- allocation structs: data + inode bitmaps

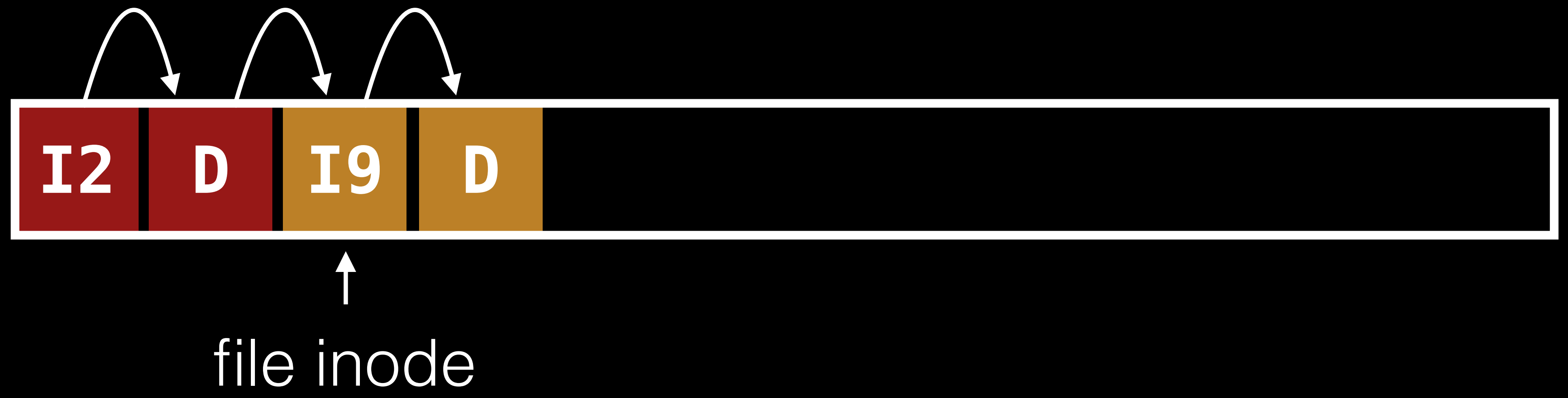
Inodes are no longer at fixed offset.
- use offset instead of table index for name.
- note: upon inode update, inode number changes.

Overwrite Data in /file.txt

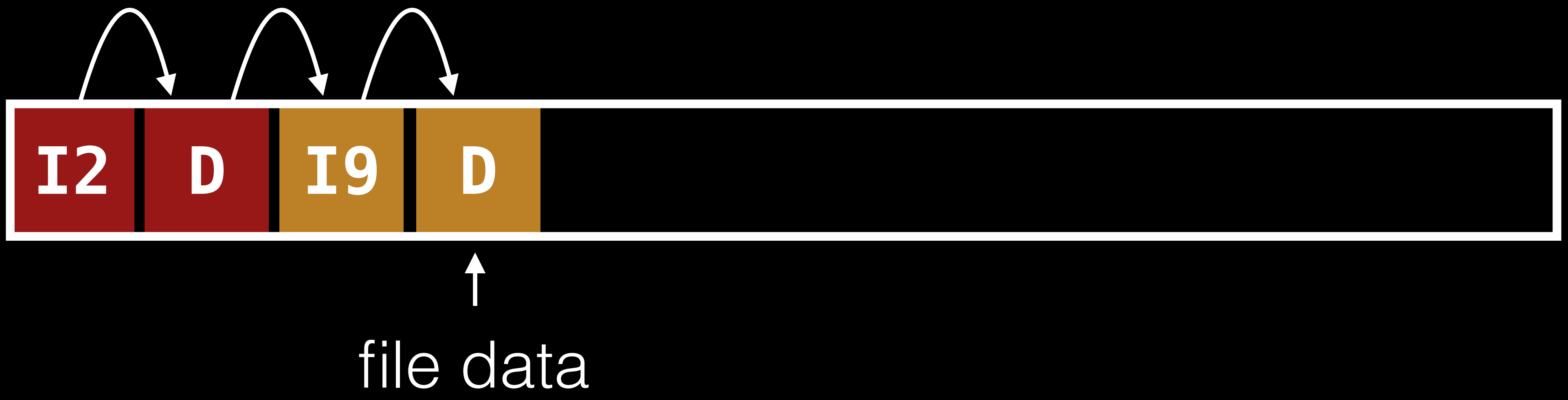


Overwrite Data in /file.txt

Overwrite Data in /file.txt



Overwrite Data in /file.txt



Overwrite Data in /file.txt

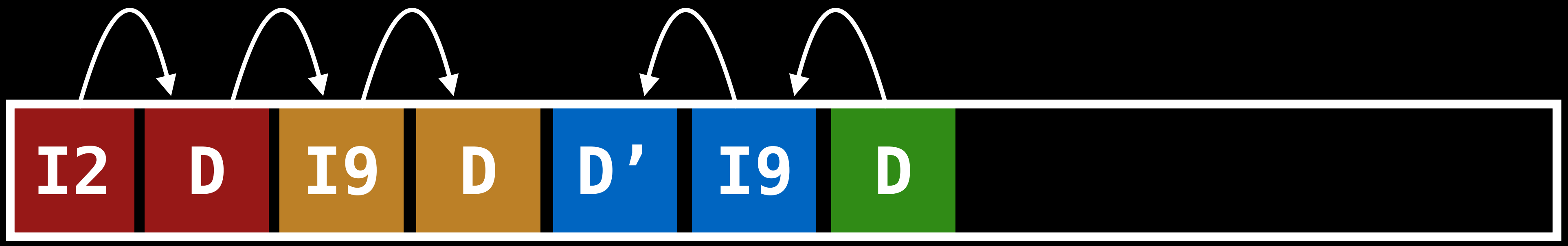
Overwrite Data in /file.txt

Overwrite Data in /file.txt

NO! This would be a random write.

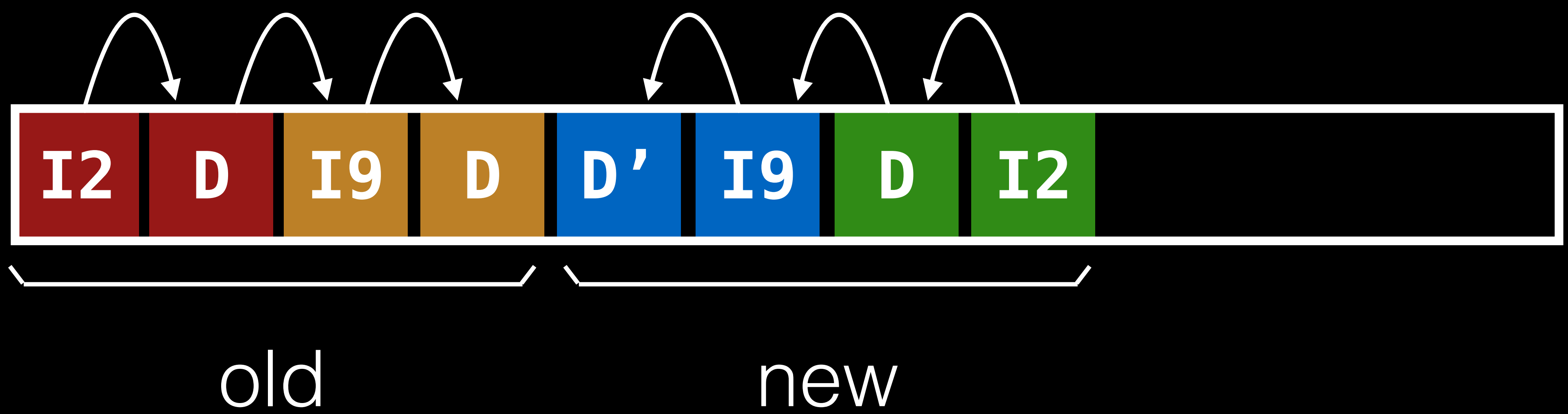
Overwrite Data in /file.txt

Overwrite Data in /file.txt



Overwrite Data in /file.txt

Overwrite Data in /file.txt



Inode Numbers

Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Inode Numbers

Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Solution: keep inode numbers constant. Don't base on offset.

Inode Numbers

Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Solution: keep inode numbers constant. Don't base on offset.

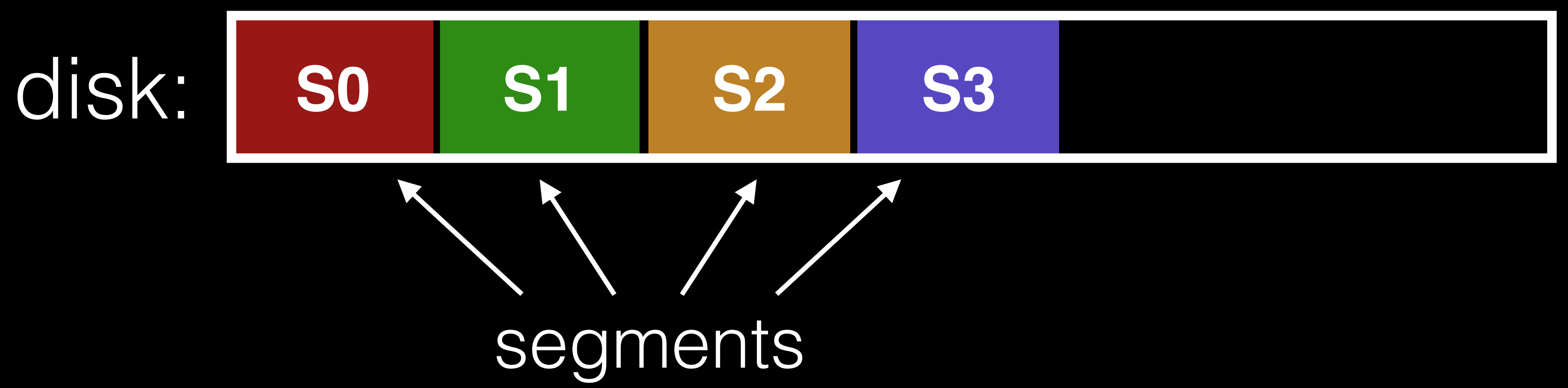
Before we found inodes with math. How now?

Data Structures (attempt 2)

What can we get rid of from FFS?
- allocation structs: data + inode bitmaps

Inodes are no longer at fixed offset.
- use imap struct to map number => inode.

imap



imap

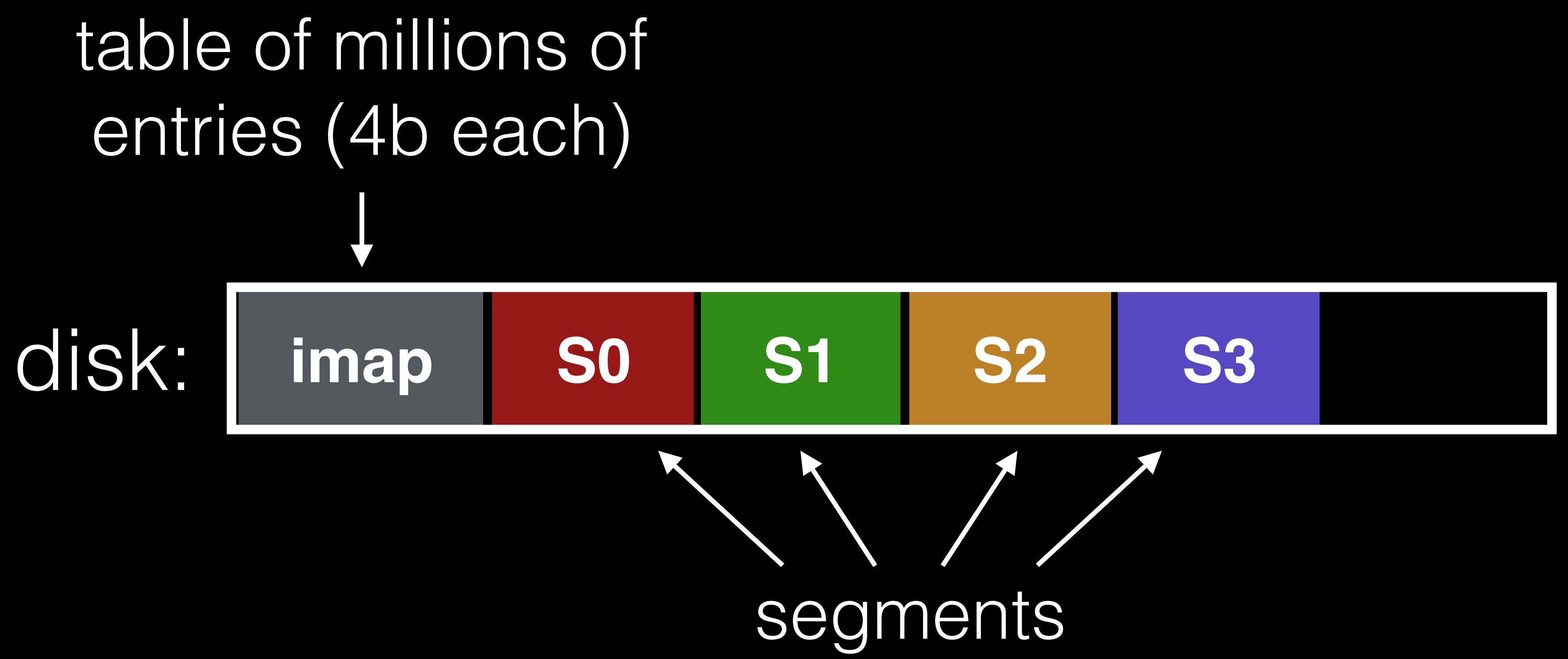
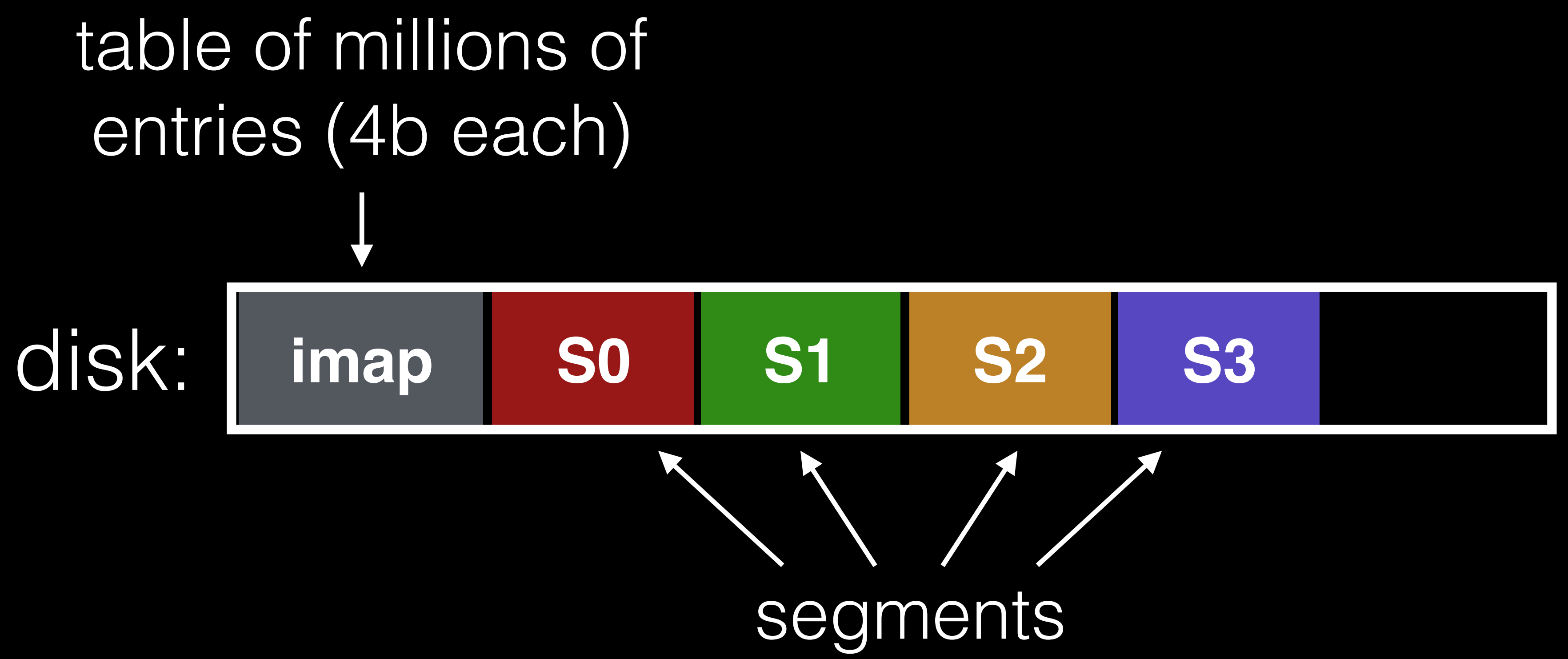


table of millions of entries (4b each)
disk:
imap
S0
S1
S2
S3
segments

imap



problem: updating imap each time makes I/O random.

Problem

Dilemma:

1. imap too big to keep in memory
2. don’t want to use random writes for imap

Attempt 3

Dilemma:

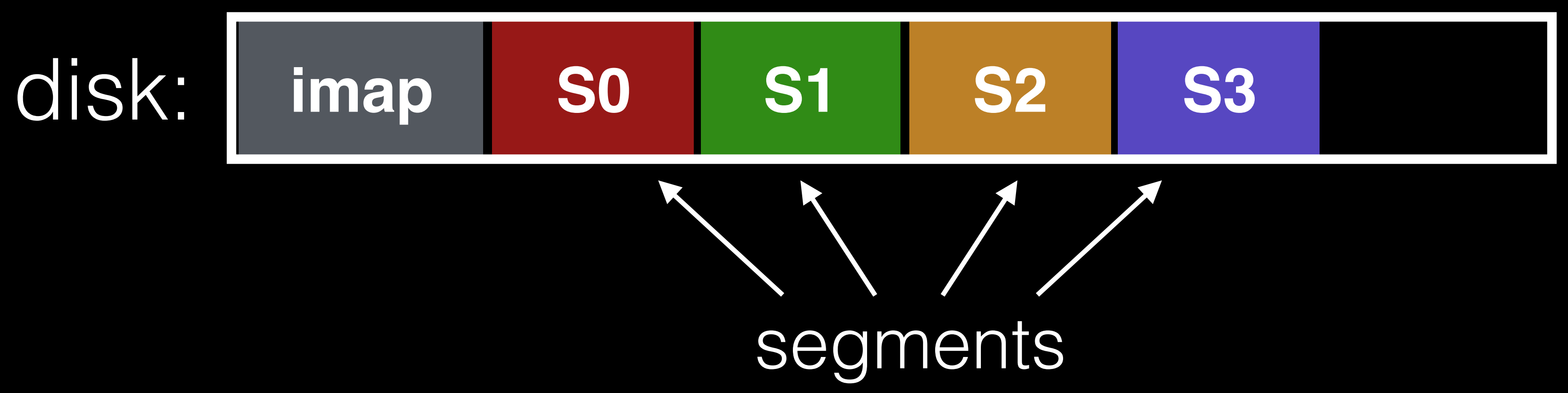
1. imap too big to keep in memory
2. don't want to use random writes for imap

Solution:

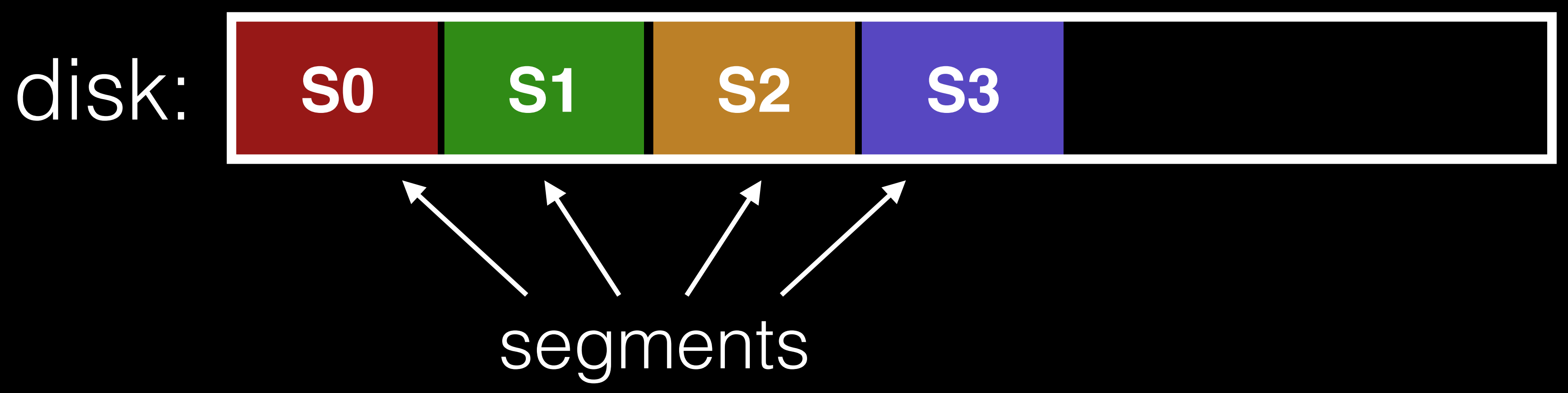
write imap in segments.

keep pointers to pieces of imap in memory.

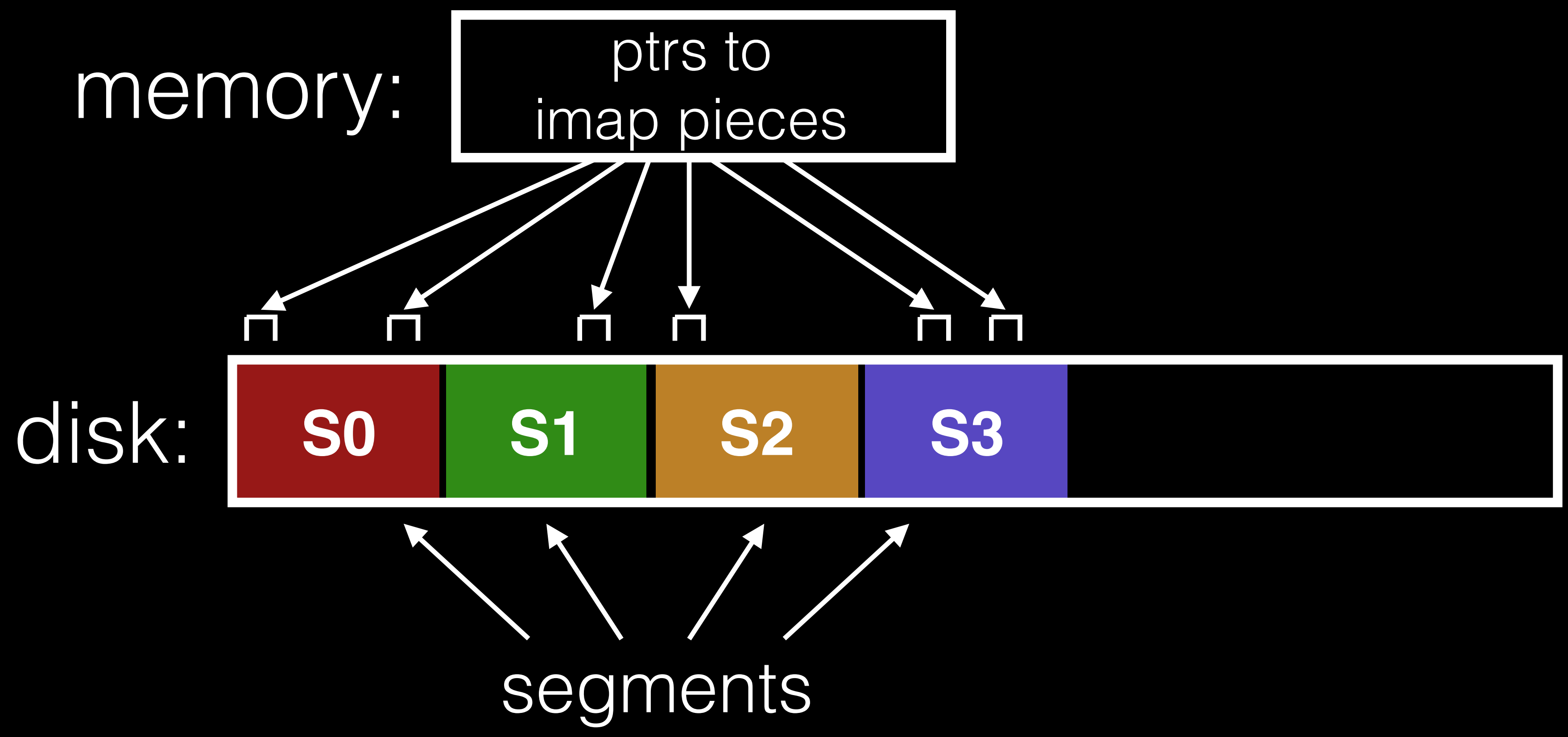
imap



imap



imap

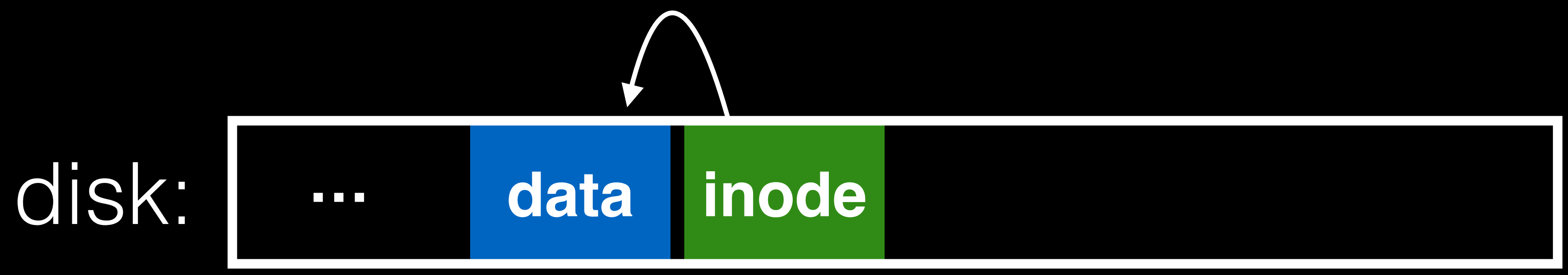


Example Write

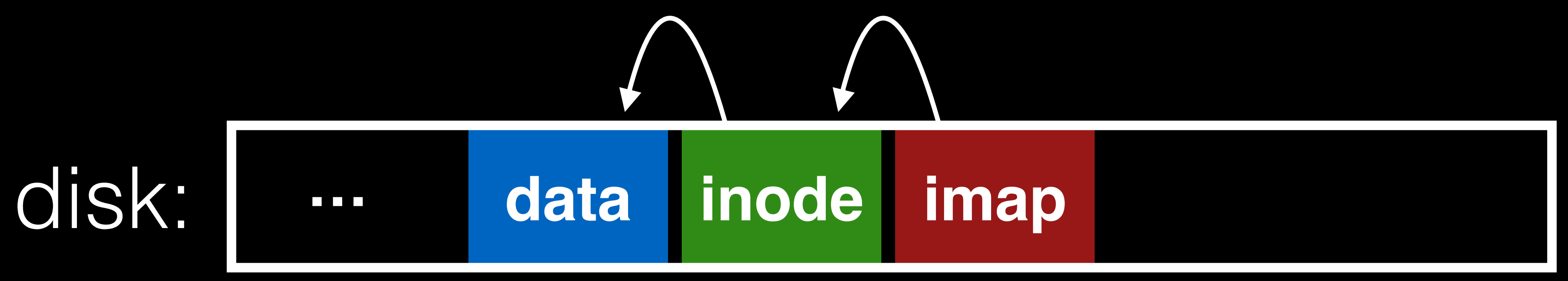
disk: ...

Example Write

Example Write



Example Write



Other Issues

Crashes

Garbage Collection

Crash Recovery

Naive approach: scan entire log to reconstruct pointers to imap pieces. Slow!

Crash Recovery

Naive approach: scan entire log to reconstruct pointers to imap pieces. Slow!

Better approach: occasionally checkpoint the pointers to imap pieces on disk.

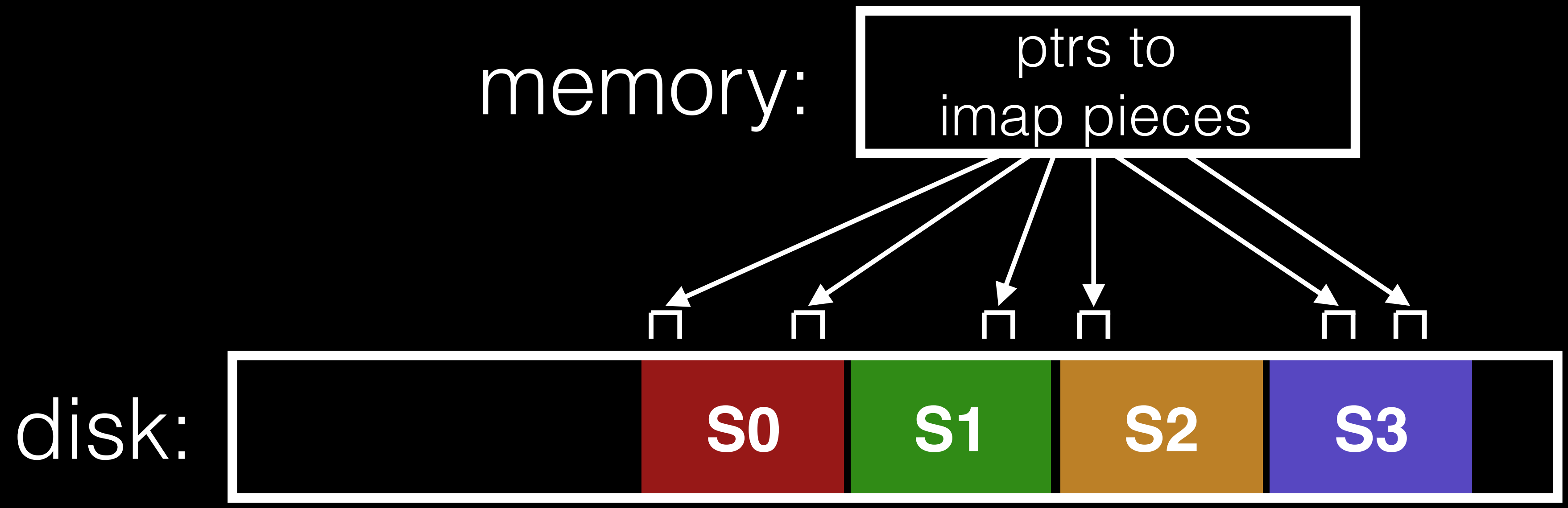
Crash Recovery

Naive approach: scan entire log to reconstruct pointers to imap pieces. Slow!

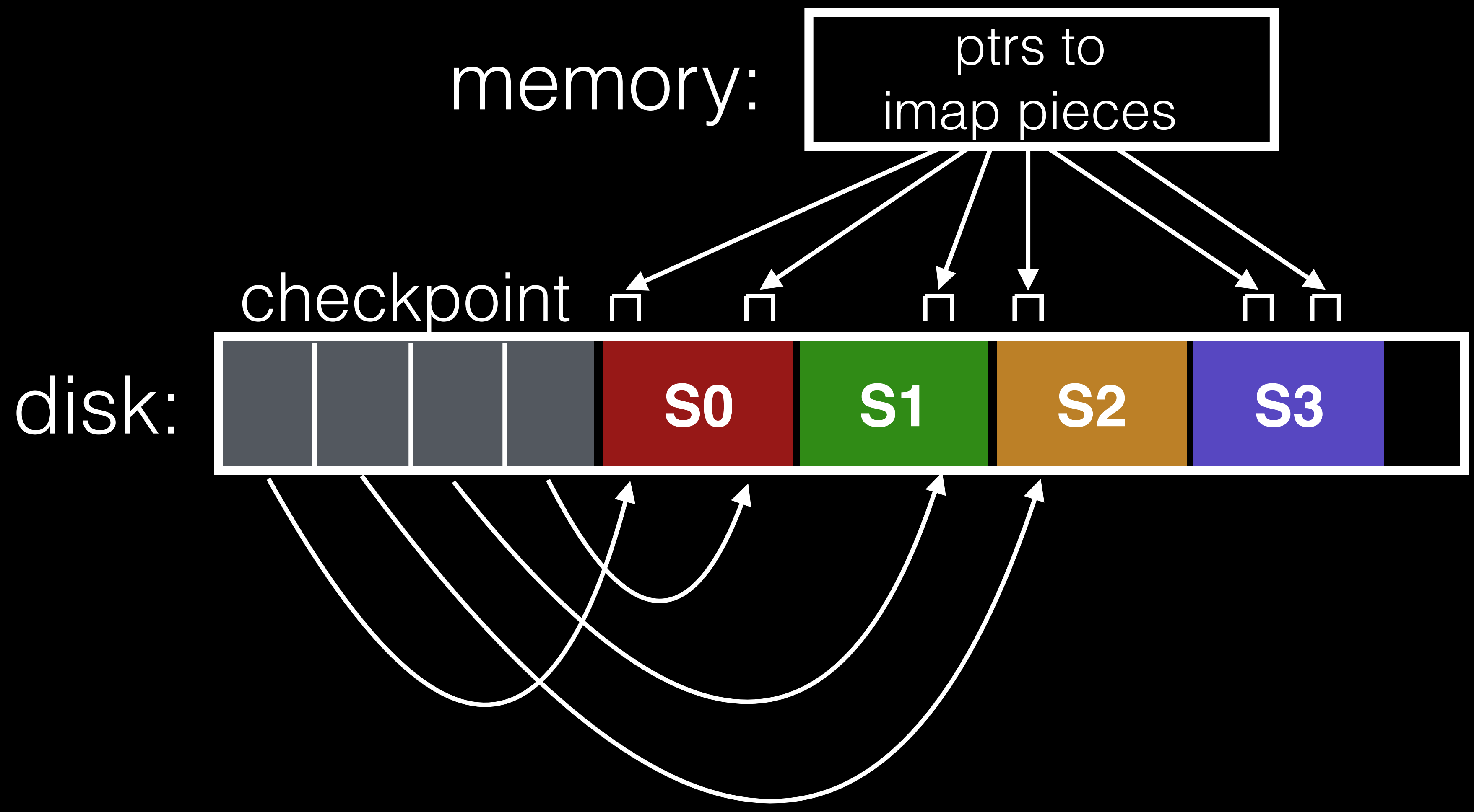
Better approach: occasionally checkpoint the pointers to imap pieces on disk.

Checkpoint often: random I/O.
Checkpoint rarely: recovery takes longer.
Example: checkpoint every 30s

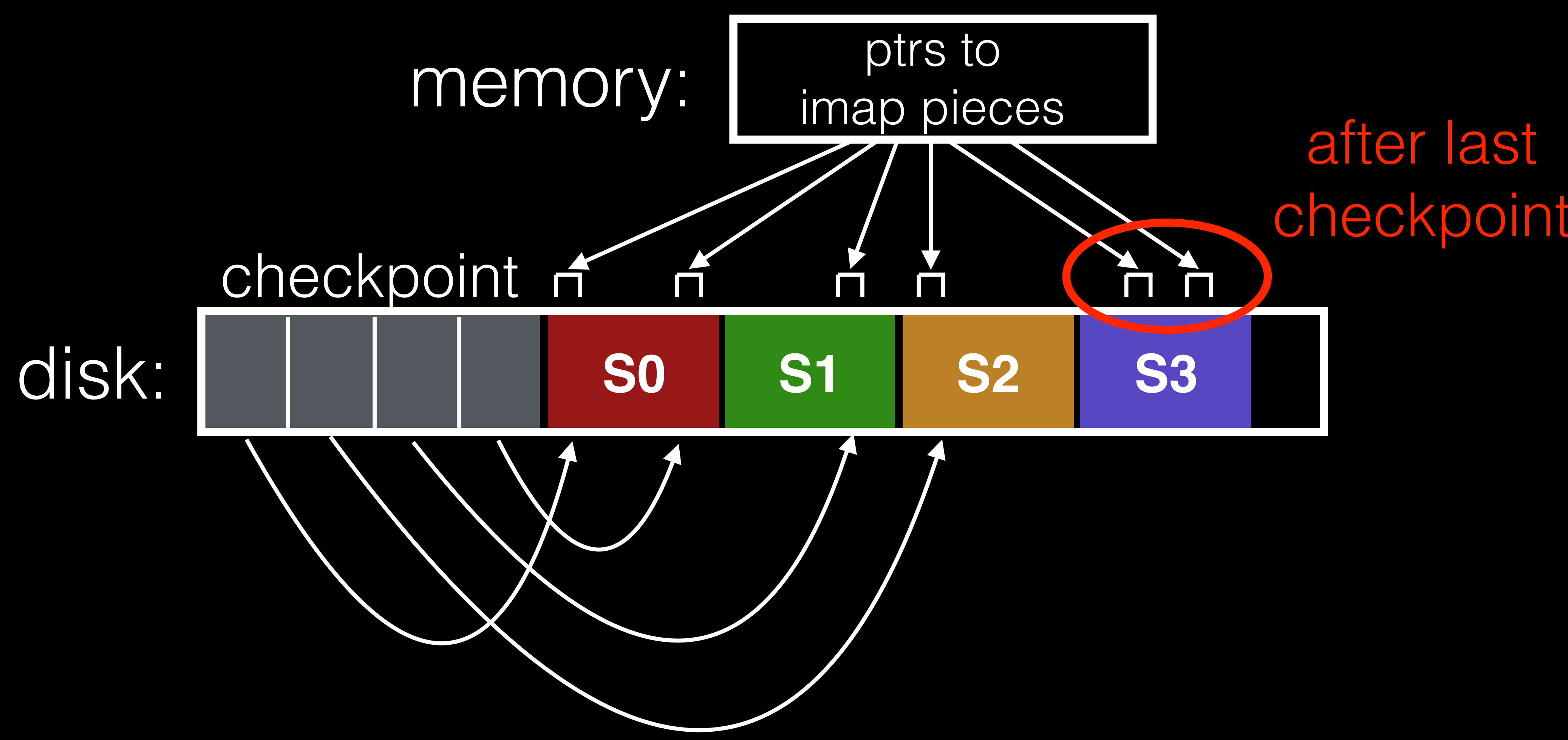
Checkpoint



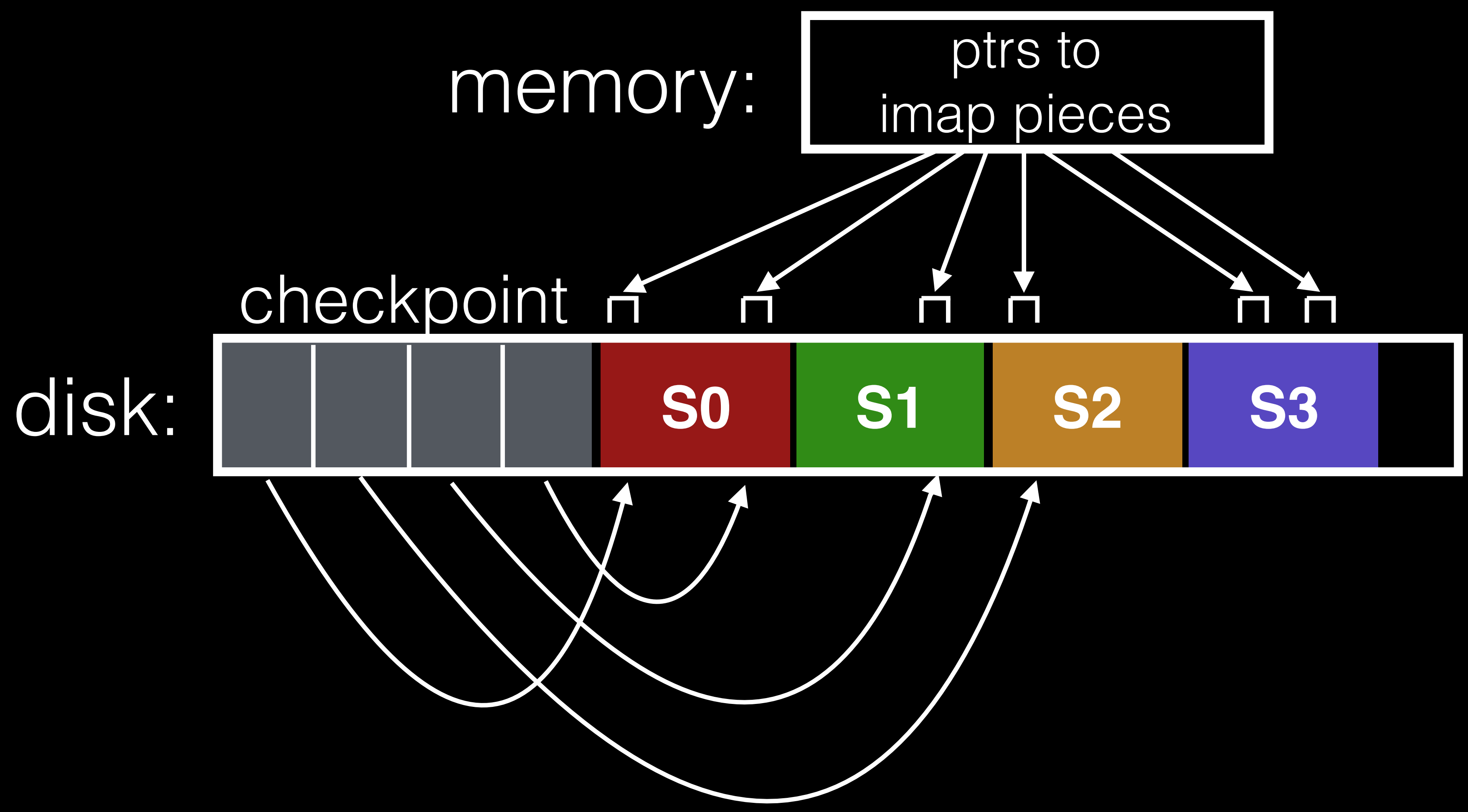
Checkpoint



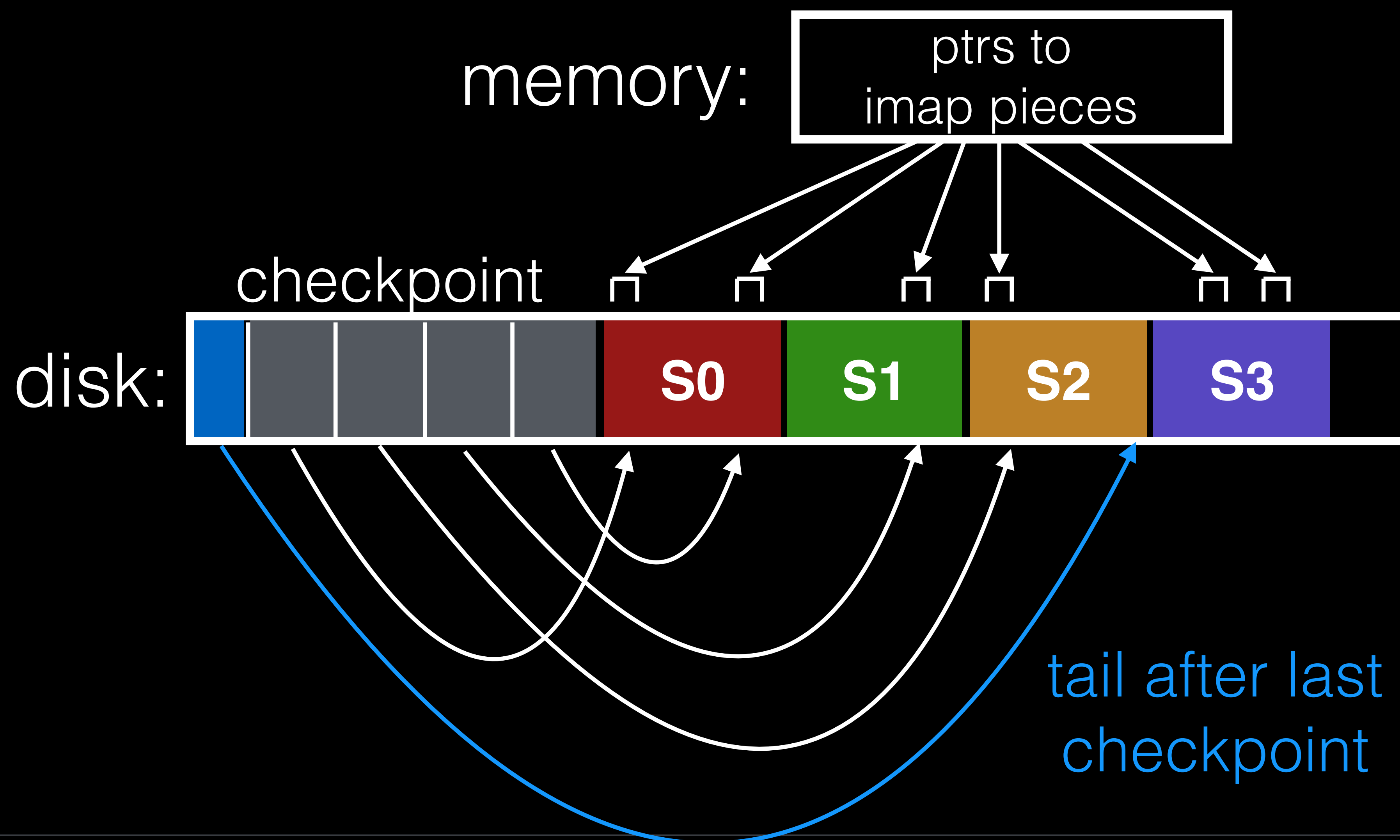
Checkpoint


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

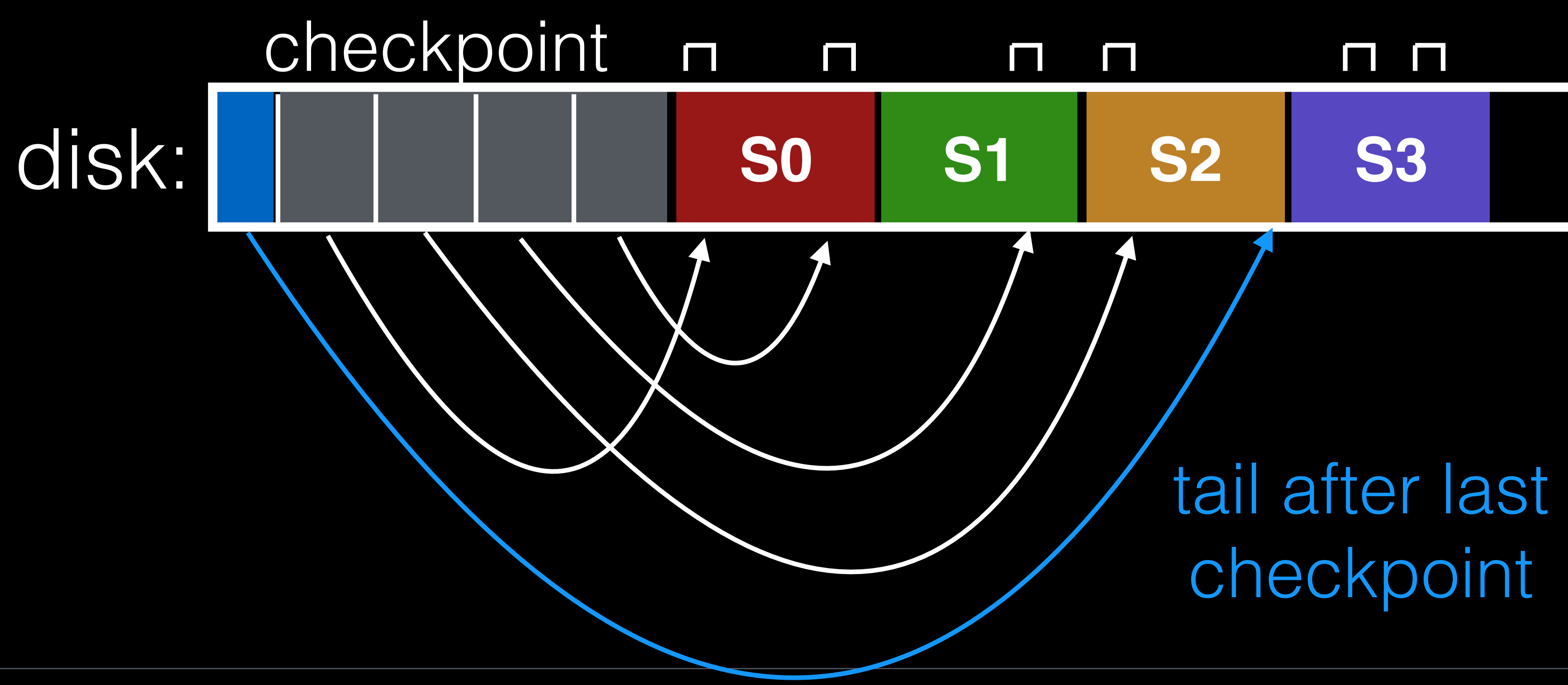
Checkpoint



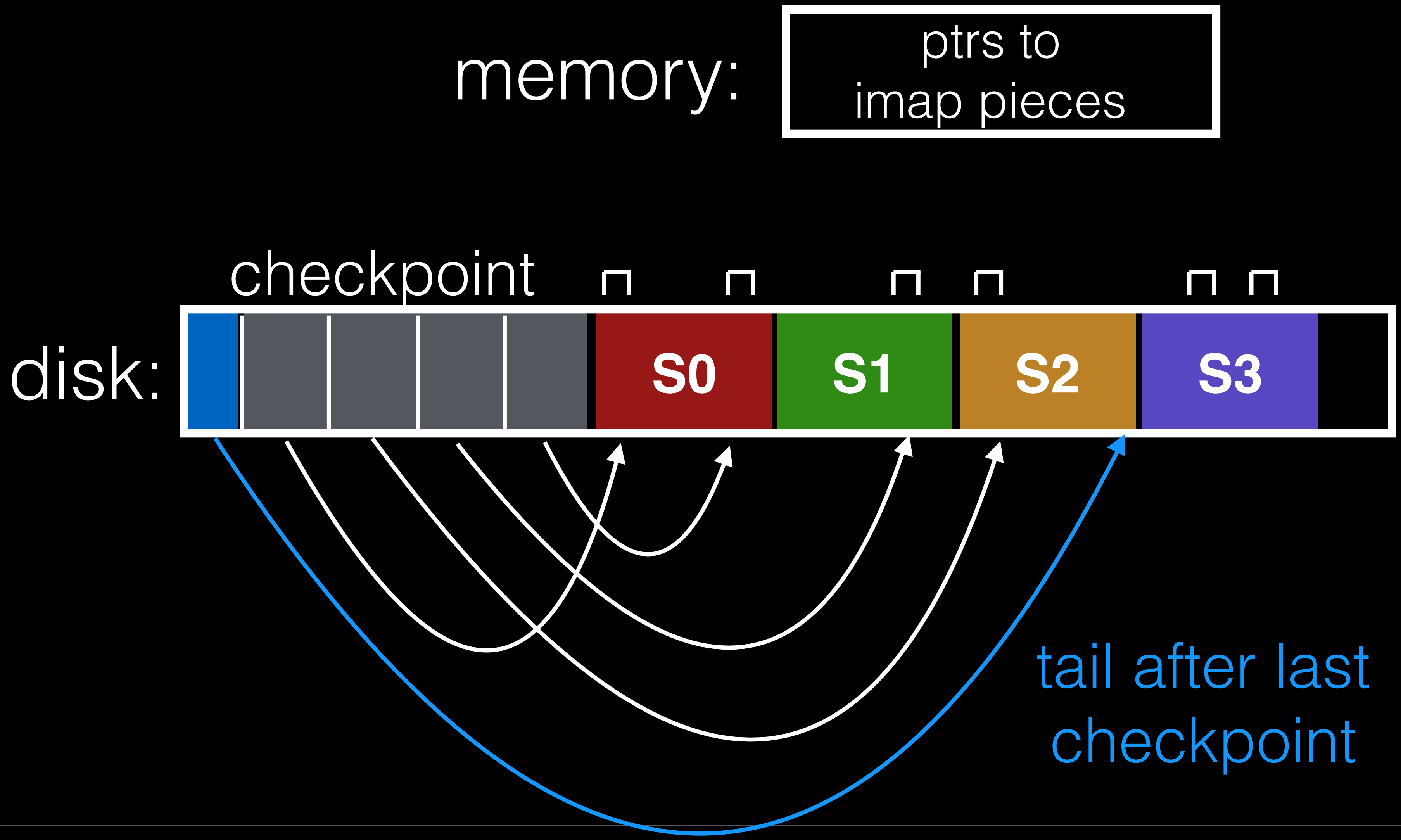
Checkpoint


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

Crash!

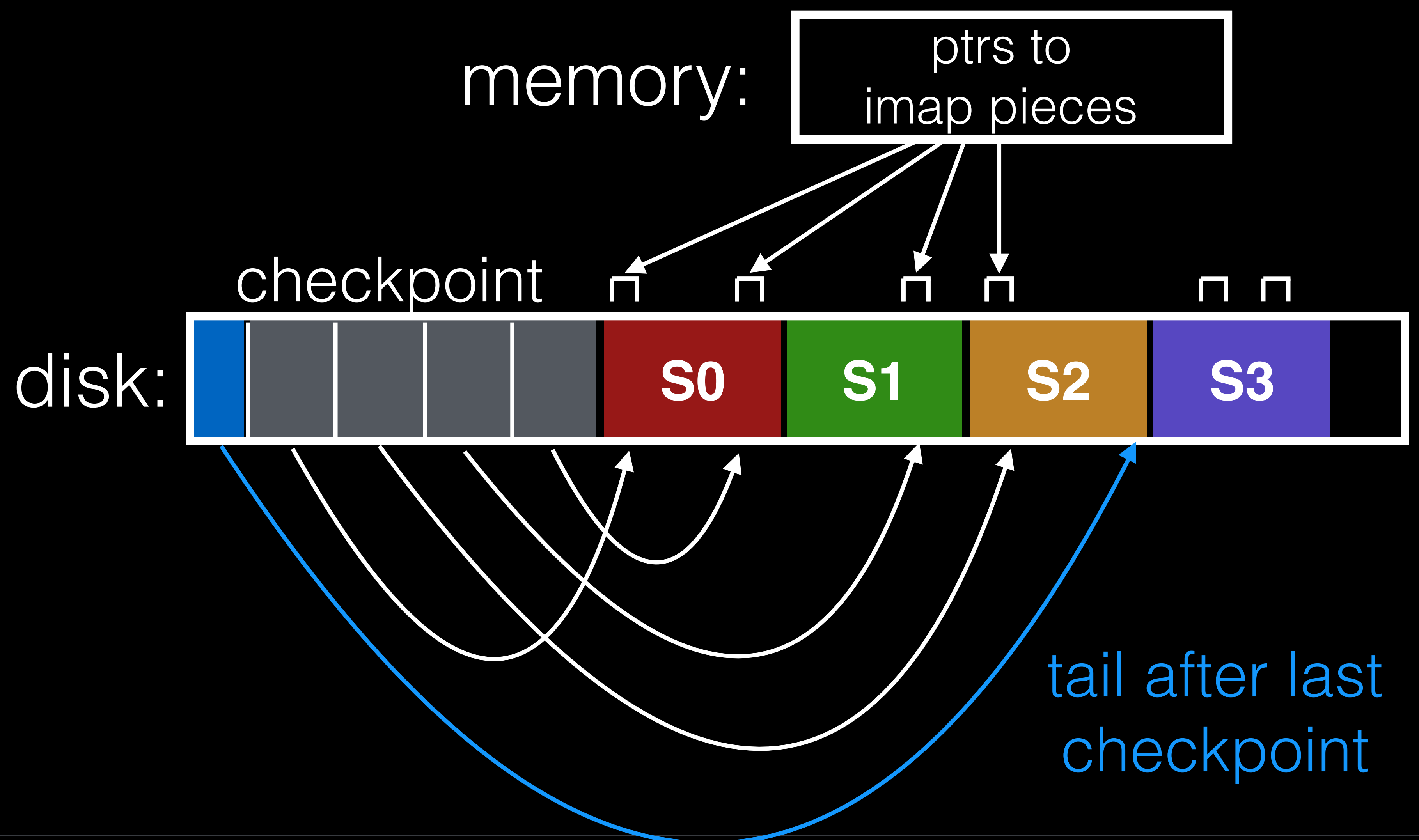


Reboot


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

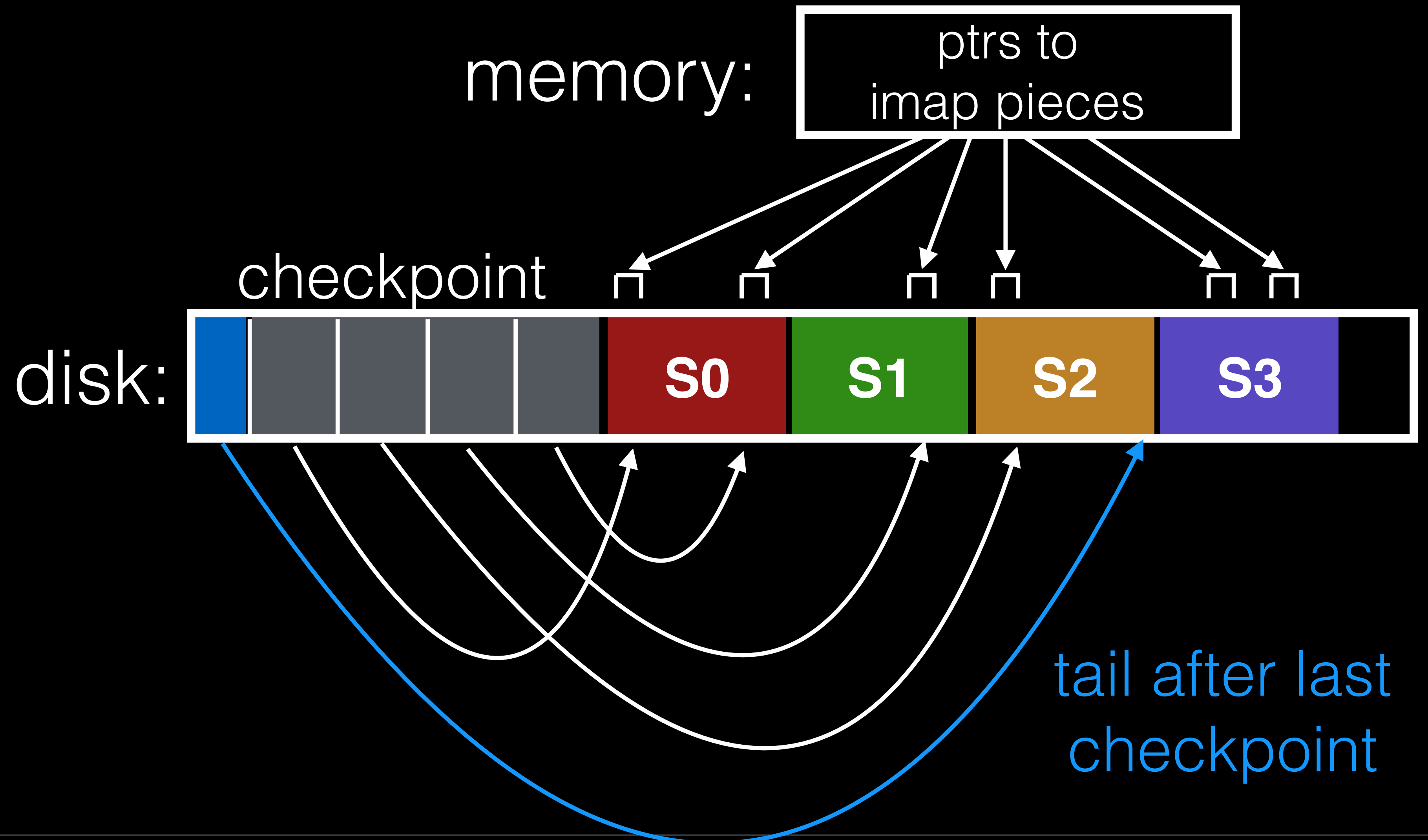
Reboot

get pointers from checkpoint



Reboot

get pointers by scanning after tail.



Checkpoint Overview

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

Upon recovery:
- read checkpoint to get most pointers and tail
- get rest of pointers by reading past tail

Checkpoint Overview

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

Upon recovery:
- read checkpoint to get most pointers and tail
- get rest of pointers by reading past tail

What if we crash during checkpoint?

Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Checkpoint Strategy

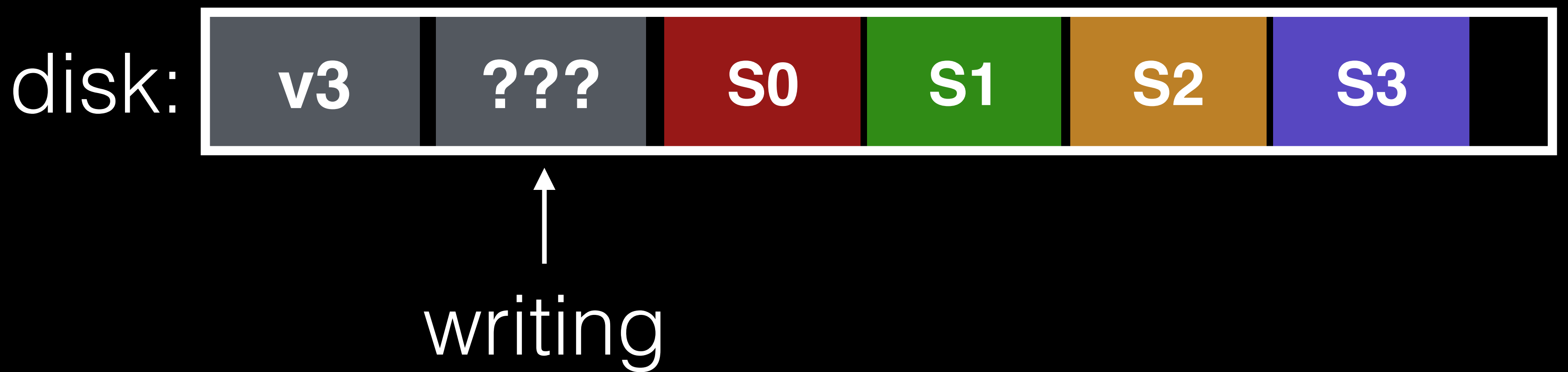
Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.



Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

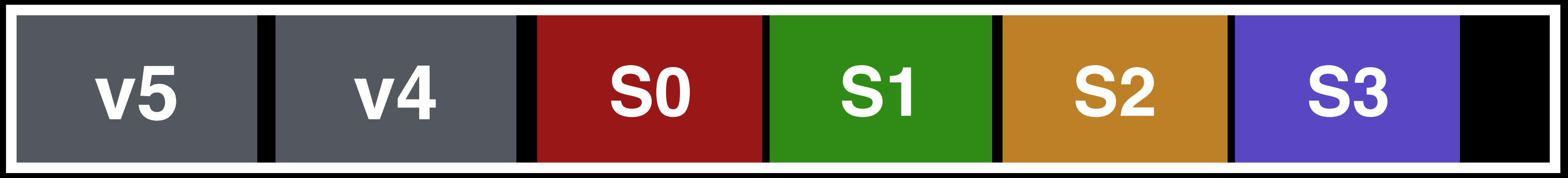
Checkpoint Strategy

Have two checkpoints.

Only overwrite one at a time.

Use checksum/timestamps to identify newest.

disk:



Other Issues

Crashes

Garbage Collection

Versioning File Systems

Motto: garbage is a feature!

Versioning File Systems

Motto: garbage is a feature!

Keep old versions in case the user wants to revert files later.

Like Dropbox.

Garbage Collection

Need to reclaim space:

1. when no more references (any file system)
2. after a newer copy is created (COW file system)

We want to reclaim segments.

- tricky, as segments are usually partly valid

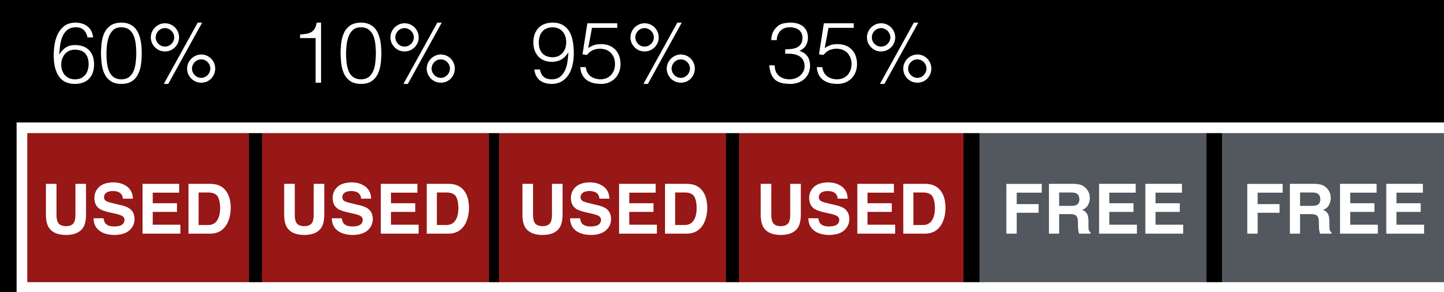
Garbage Collection

disk segments: 

Garbage Collection

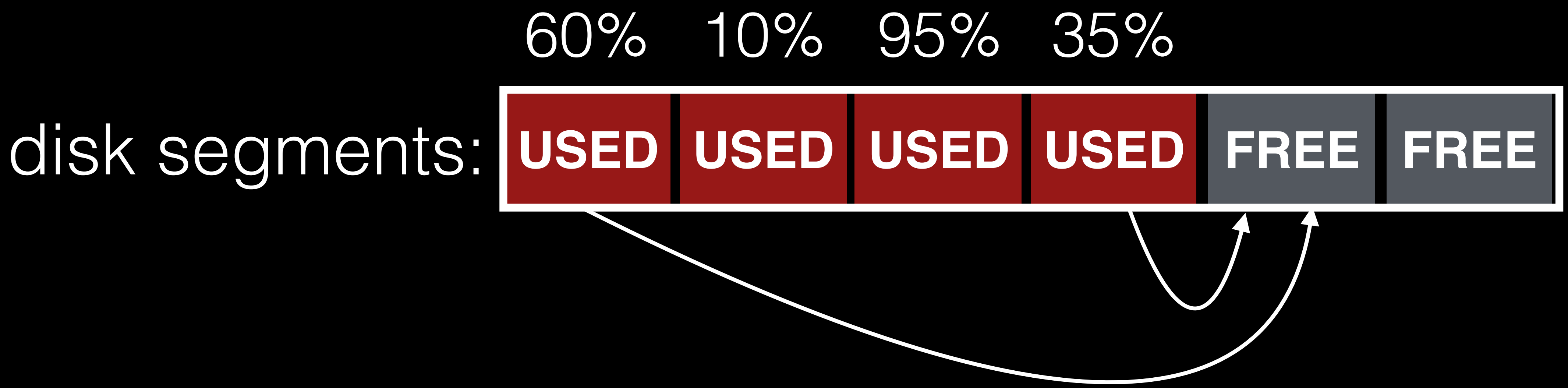
how much data is good in each?

disk segments:

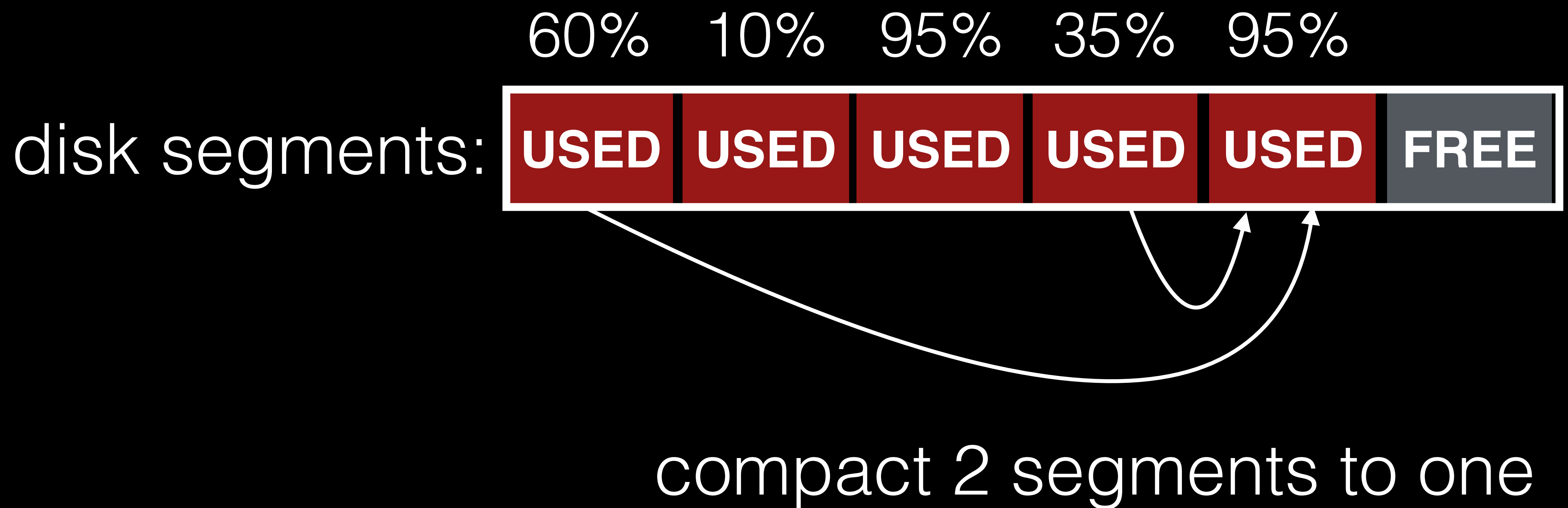


Garbage Collection

Garbage Collection



Garbage Collection



Garbage Collection

disk segments:

release input segments

Garbage Collection

General operation:
pick **M** segments, compact into **N** (where **N** < **M**).

Mechanism: how do we know whether data in segments is valid?

Policy: which segments to compact?

Mechanism

Is an inode the latest version?
Check imap to see if it is pointed to (fast).

Is a data block the latest version?
Scan ALL inodes to see if it is pointed to (very slow).

Mechanism

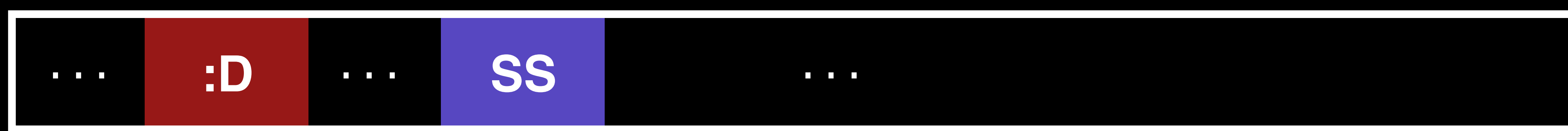
Is an inode the latest version?
Check imap to see if it is pointed to (fast).

Is a data block the latest version?
Scan ALL inodes to see if it is pointed to (very slow).

Solution: segment summary that lists inode corresponding to each data block.

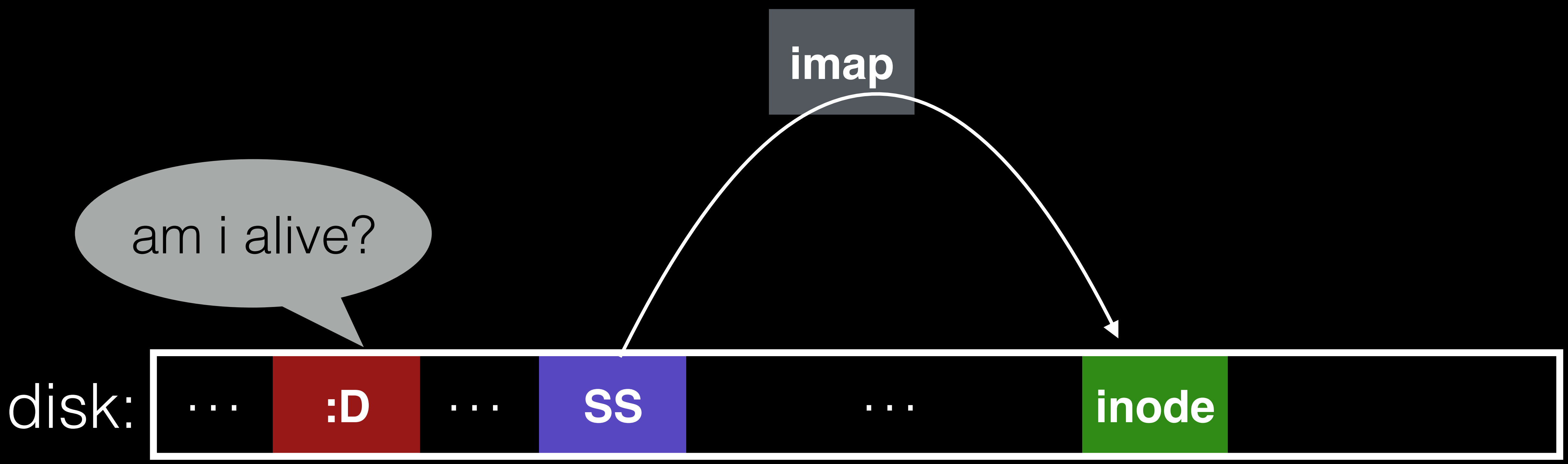
Block Liveness

disk:

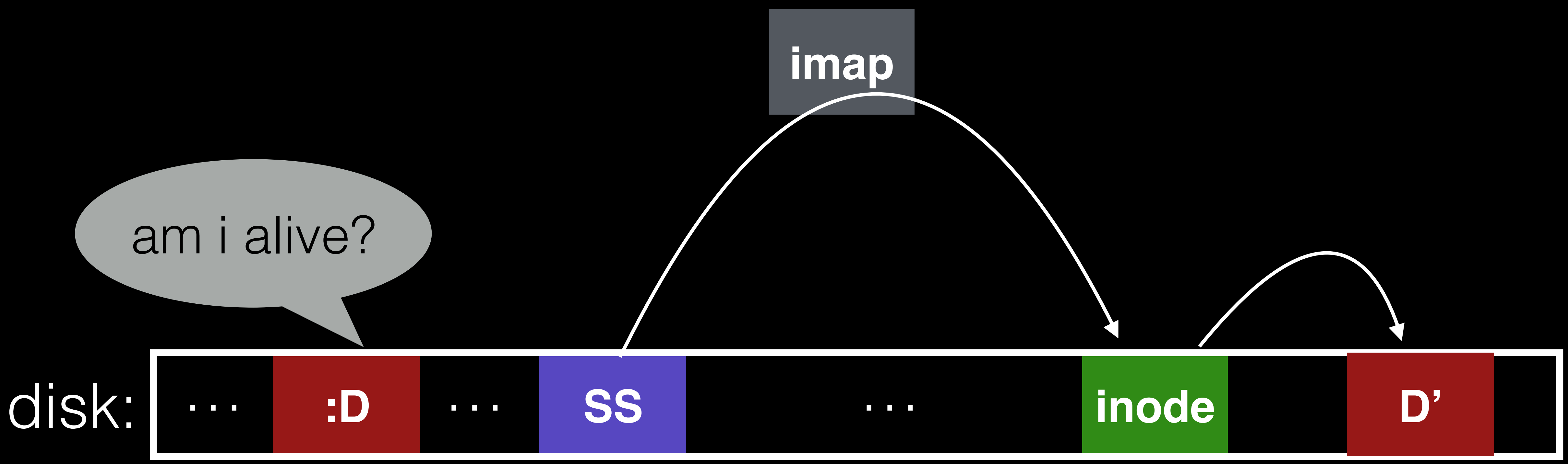


Block Liveness

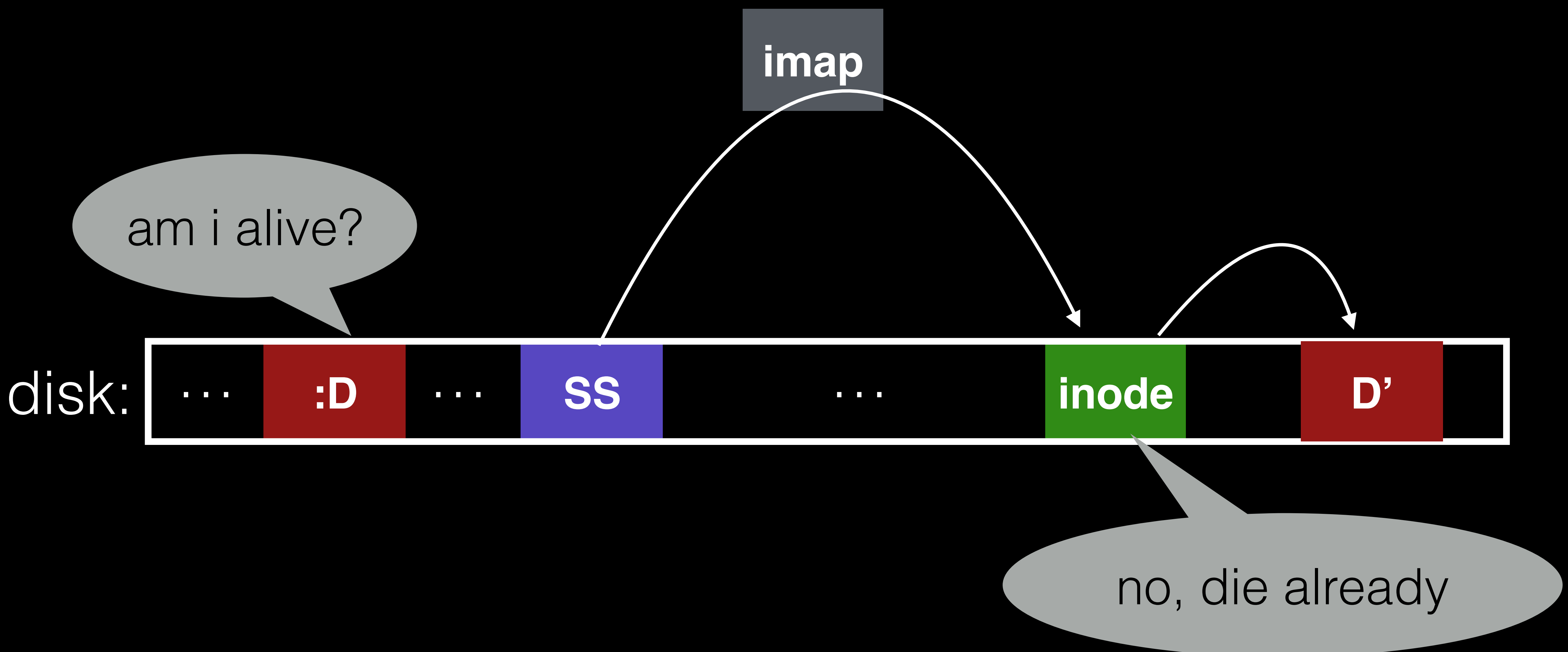
Block Liveness



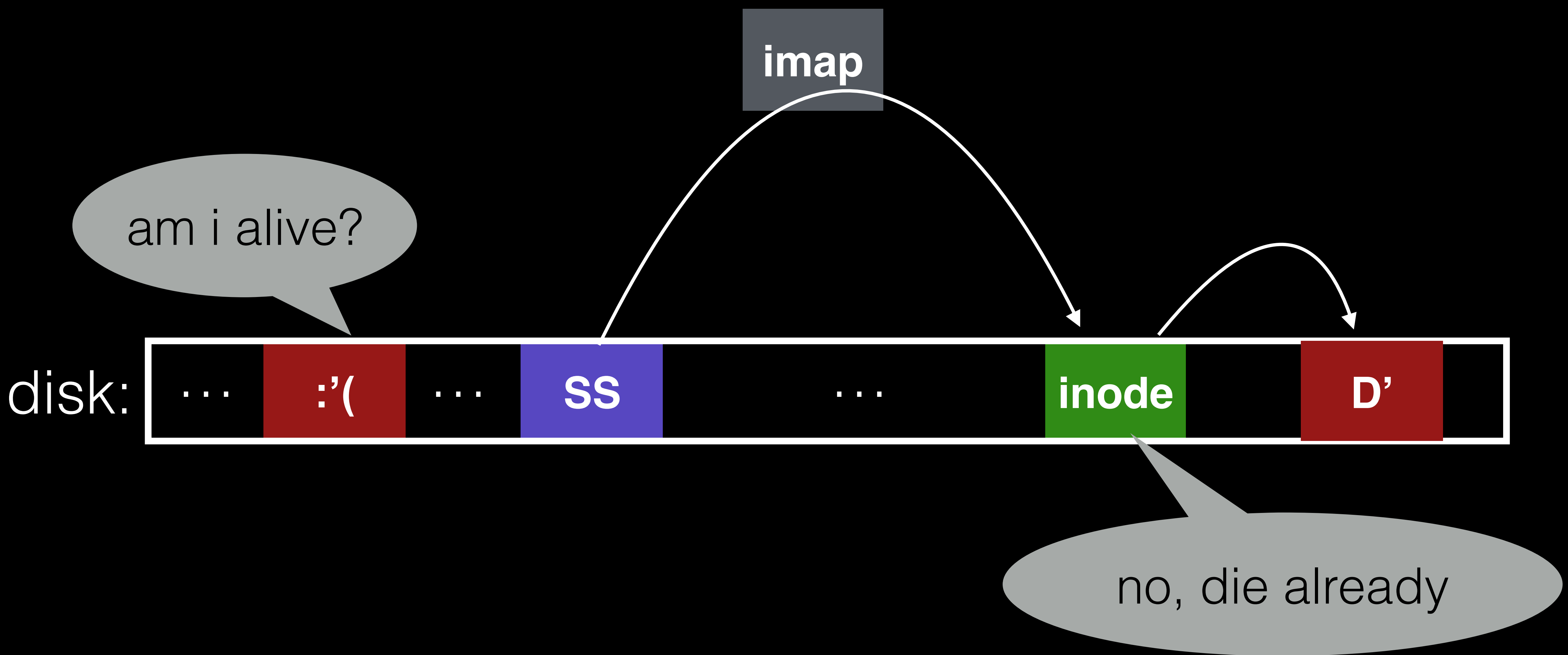
Block Liveness



Block Liveness



Block Liveness



Garbage Collection

General operation:
pick **M** segments, compact into **N** (where **N** < **M**).

Mechanism: how do we know whether data in segments is valid?

Policy: which segments to compact?

Garbage Collection

General operation:
pick **M** segments, compact into **N** (where **N** < **M**).

Mechanism: how do we know whether data in segments is valid? [segment summary]

Policy: which segments to compact?

Policy

Many possible:

clean most empty first
clean coldest
more complex heuristics…

Conclusion

Journaling: let's us put data wherever we like.
Usually in a place optimized for future reads.

LFS: puts data where it's fastest to write.

Other COW file systems: WAFL, ZFS, btrfs.

Announcements

Thursday **discussion**
- review midterm 2.

Office hours
- today, after class, in lab