PERSISTENCE: CRASH CONSISTENCY

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

Make-up Points for Projects 4

- Due 1 week from when made available

File System Structures Quiz: Lecture Content

- Due Today

Project 6: Due Friday (5pm or midnight)

- Discussion tomorrow for more questions

Project 7: Available immediately after Thanksgiving

- xv6 File systems: Improvements + checker?
- Will have Specification Quiz

AGENDA / LEARNING OUTCOMES

How to maintain consistency with power failures / crashes?

- What can go wrong if disk blocks are not updated consistently?
- How can file system be **checked and fixed** after crash?
- How can **journaling** be used to obtain **atomic updates**?
- How can the **performance** of journaling be improved?

DATA REDUNDANCY

Definition:

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

RAID examples:

- mirrored disk (complete redundancy)
- parity blocks (partial redundancy)

File system examples:

- **Superblock**: field contains total blocks in FS
- **Inodes**: field contains pointer to data block
- Is there redundancy across these two fields? Why or why not?

FILE SYSTEM CONSISTENCY EXAMPLE

Superblock: field contains total number of blocks in FS

value = N

Inode: field contains pointer to data block; possible values?

values in {0, 1, 2, …, N - 1}

Pointers to block N or after are invalid!

Total-blocks field has redundancy with inode pointers

WHY IS CONSISTENCY CHALLENGING?

File system may perform several disk writes to redundant blocks

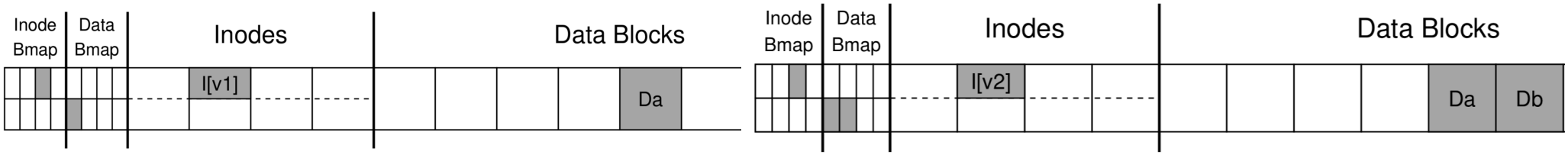
If file system is interrupted between writes, may leave data in inconsistent state
Only single sector writes are guaranteed to be atomic by disk

What can interrupt write operations?
- power loss
- kernel panic
- reboot

File system must update multiple structures: append to /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
				write			
							write

FILE APPEND EXAMPLE



Written to disk	Result
Db	
I[v2]	
B[v2]	
I[v2] + B[v2]	
I[v2] + Db	
B[v2] + Db	

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?

Is the corresponding data block allocated or free?

If any pointer to data block, the corresponding bit should be 1; else bit is 0

Read every valid inode+indirect block

FSCK CHECKS

Do superblocks match?

Do directories contain "." and ".."? Size and numblocks in inodes match?

Is the list of free blocks correct?

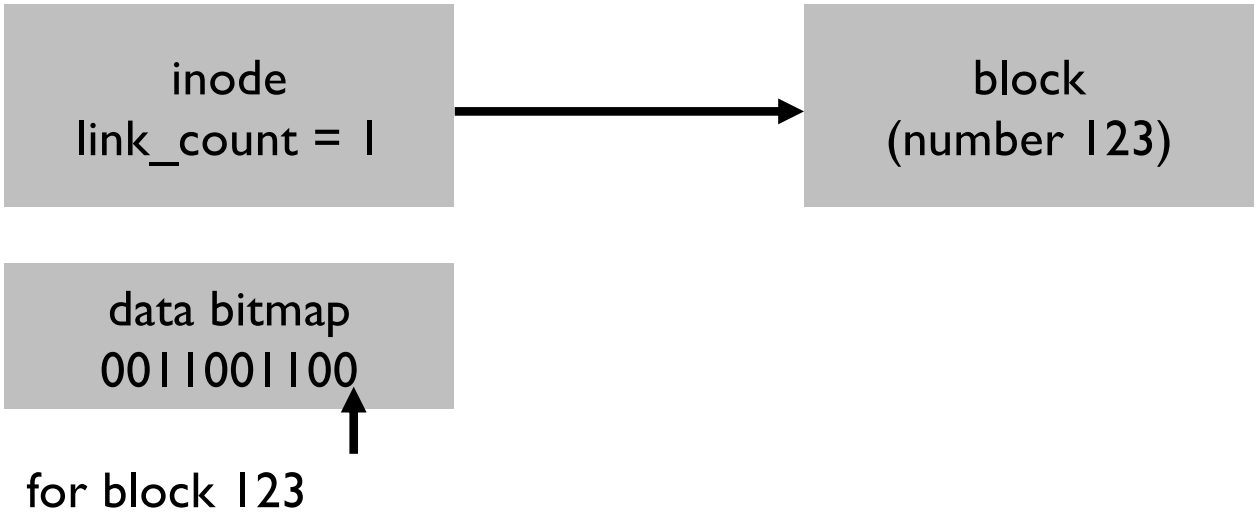
Do number of dir entries equal inode link counts?

Do different inodes ever point to same block?

Are there any bad block pointers?

…

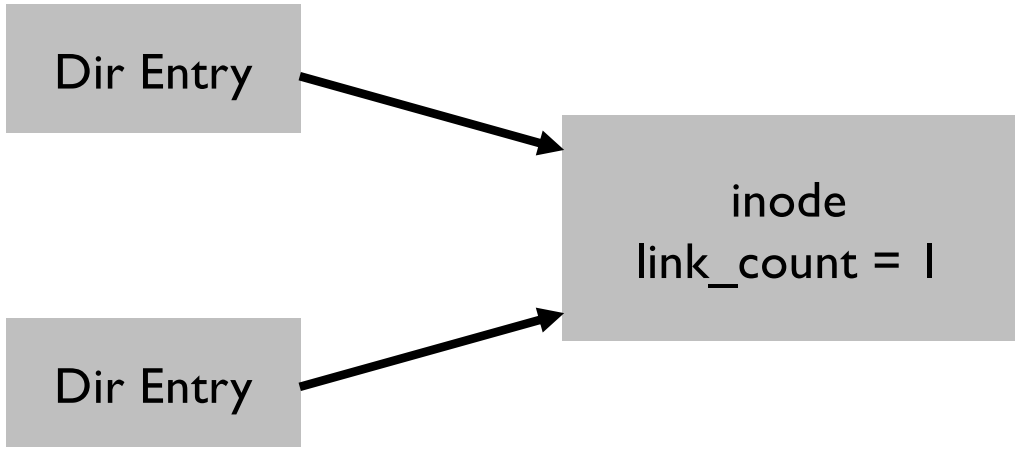
FREE BLOCKS EXAMPLE



How to fix to have consistent file system?

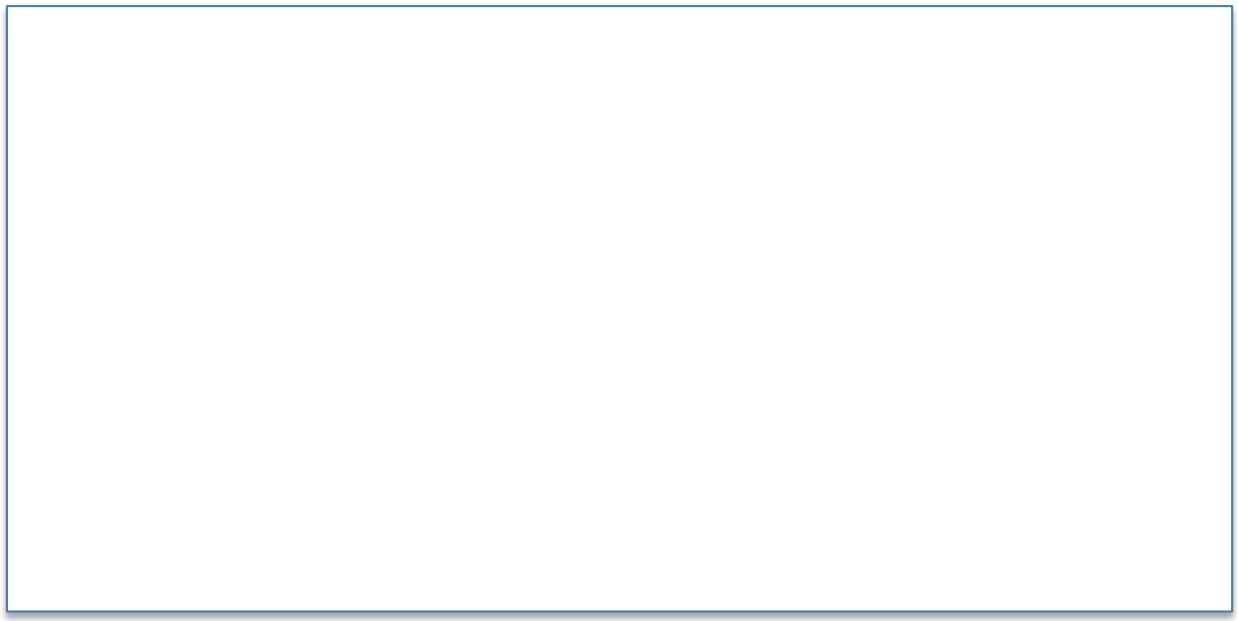
FREE BLOCKS EXAMPLE

LINK COUNT EXAMPLE

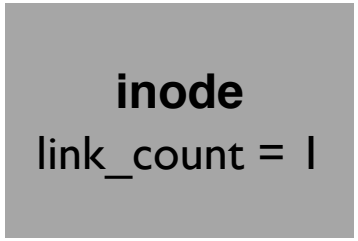


How to fix to have consistent file system?

LINK COUNT EXAMPLE

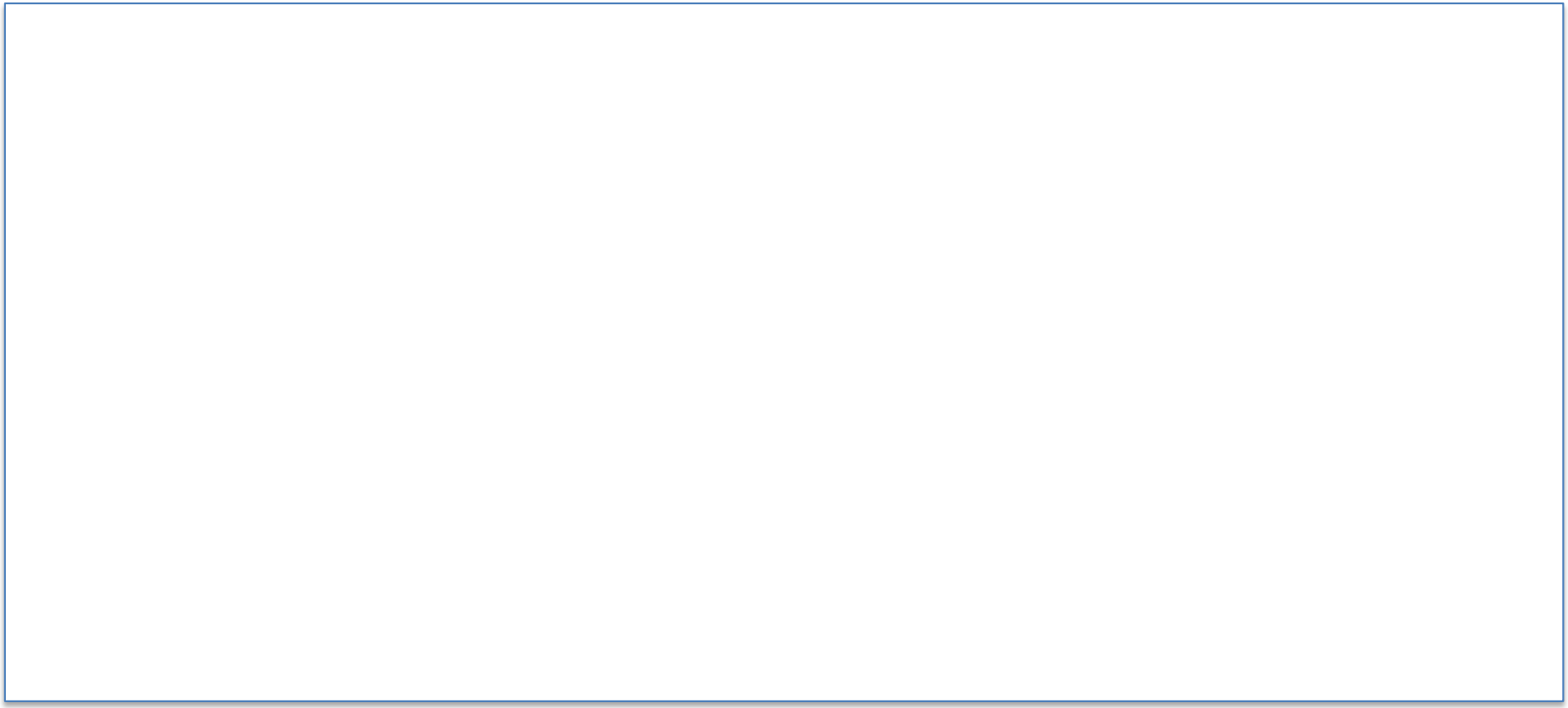


LINK COUNT (EXAMPLE 2)

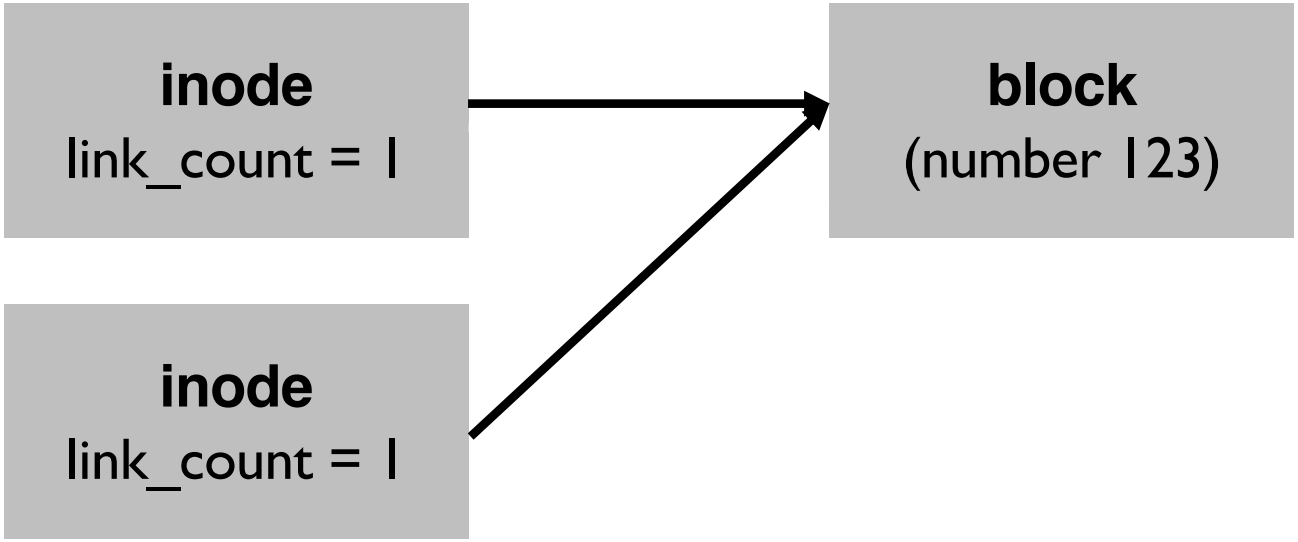


How to fix???

LINK COUNT (EXAMPLE 2)

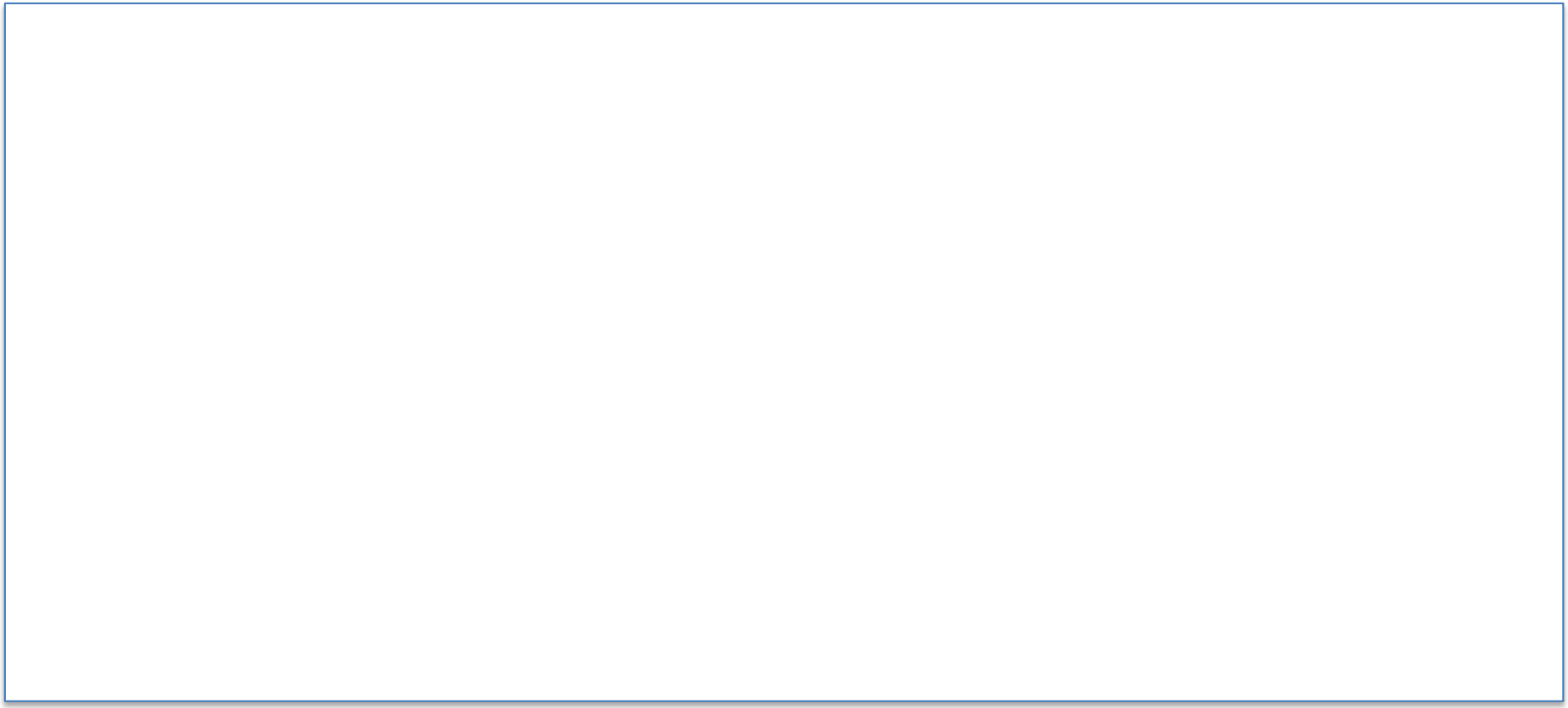


DUPLICATE POINTERS

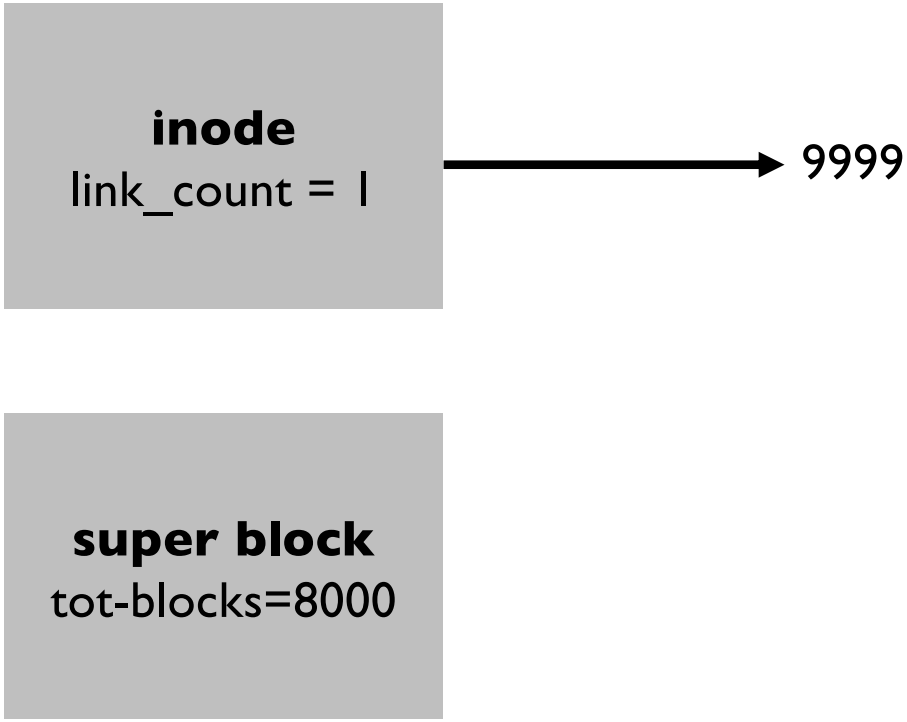


How to fix to have consistent file system?

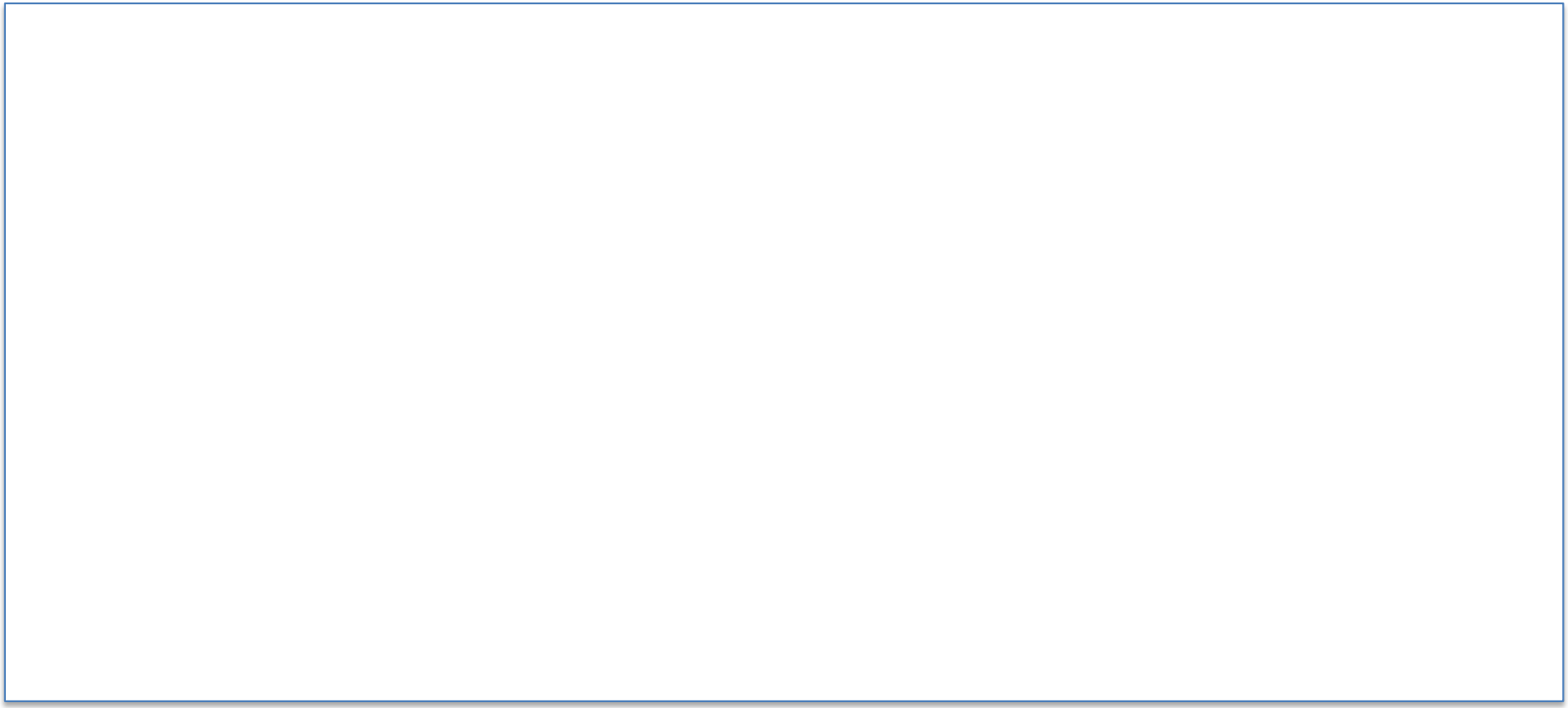
DUPLICATE POINTERS



BAD POINTER



BAD POINTER

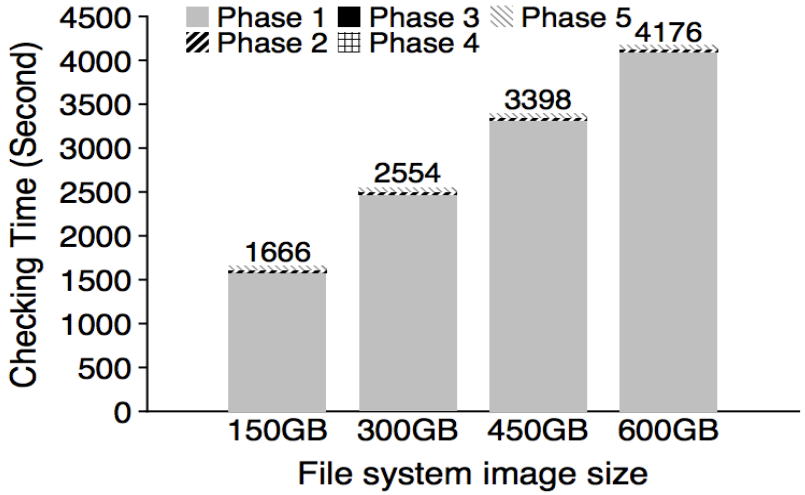


PROBLEMS WITH FSCK

Problem 1:

- Not always obvious how to fix file system image
- Don't know "correct" state, just consistent one
- Easy way to get consistency: reformat disk!

PROBLEM 2: FSCK IS VERY SLOW



Checking a 600GB disk takes ~70 minutes

ffsck: The Fast File System Checker
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

CONSISTENCY SOLUTION #2: JOURNALING

Goals

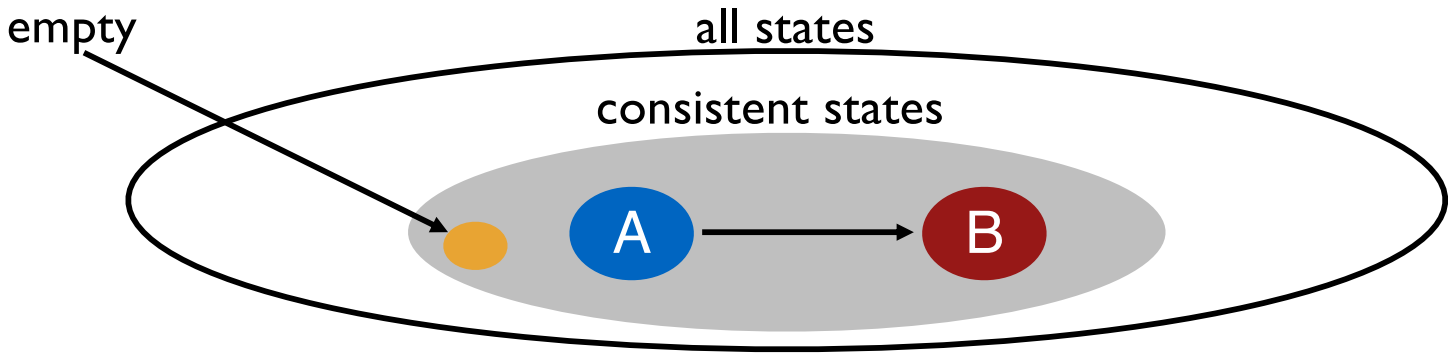
- Ok to do some **recovery work** after crash, but not to read entire disk
- Don't move file system to just any consistent state, get **correct** state

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

CONSISTENCY VS ATOMICITY

Say a set of writes moves the disk from state A to B



fsck gives consistency
Atomicity gives A or B.

JOURNALING: GENERAL STRATEGY

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

Ironically, add redundancy to fix the problem caused by redundancy

1. Make a note of what needs to be written
2. After note is completely written, can update metadata and data

If crash and recover:

1. If see note is not completely written, ignore (old data still good)
2. If see note is completely written, replay to get new data

JOURNALING TERMINOLOGY

Extra blocks in note are called a "journal"

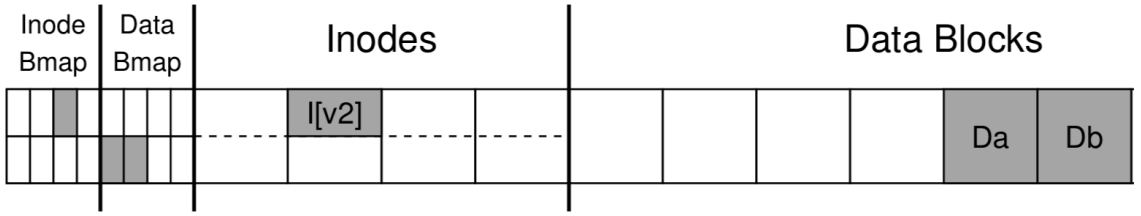
Writes to journal are "journal transaction"

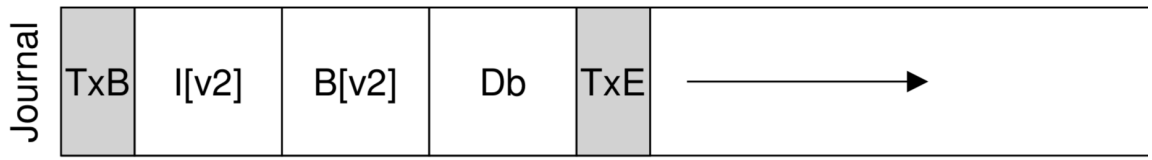
Last valid bit written is "journal commit block"

Transaction is committed

Checkpoint: Writing to in-place metadata and data after commit

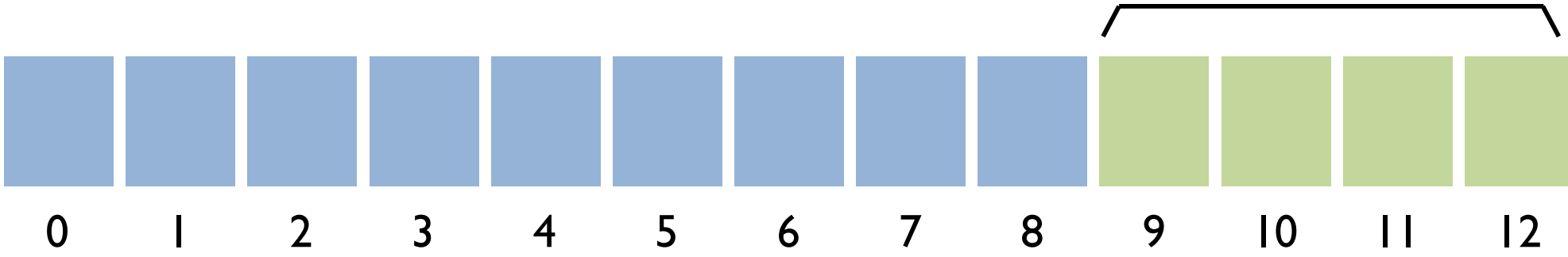
JOURNAL LAYOUT





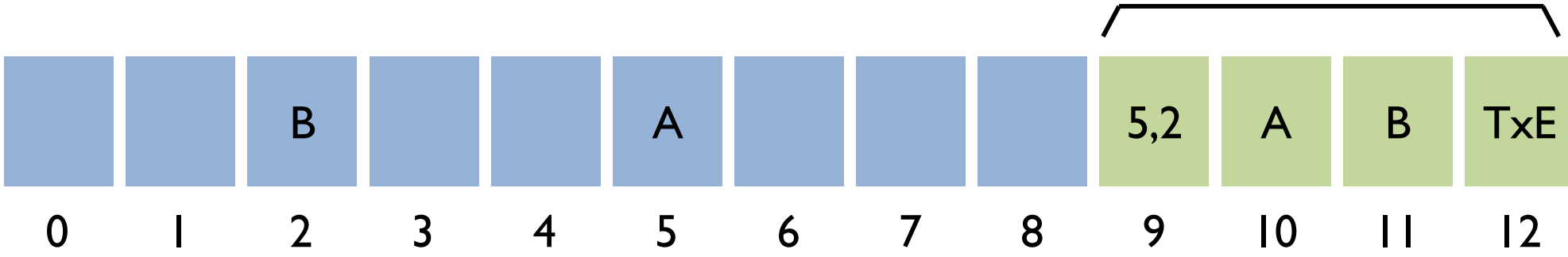
Transaction

JOURNAL WRITE AND CHECKPOINTS



transaction: write A to block 5; write B to block 2

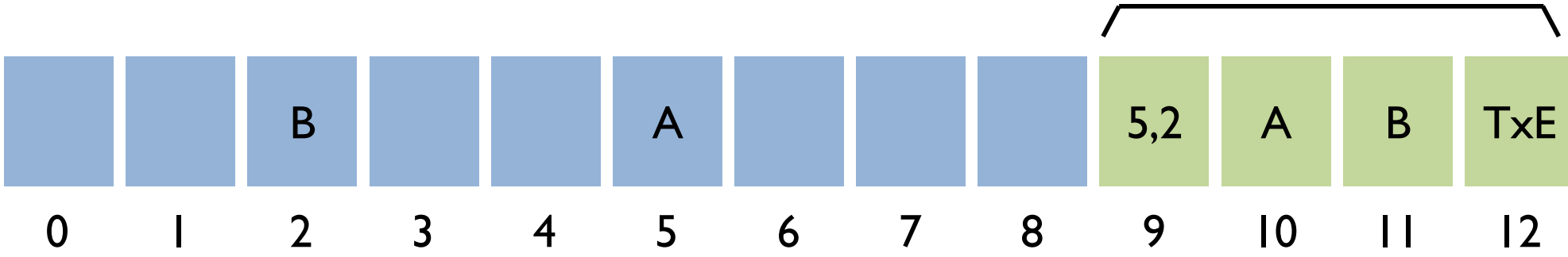
JOURNAL WRITE AND CHECKPOINTS



Journal descriptor block: What is in this transaction?
TxE: Journal commit

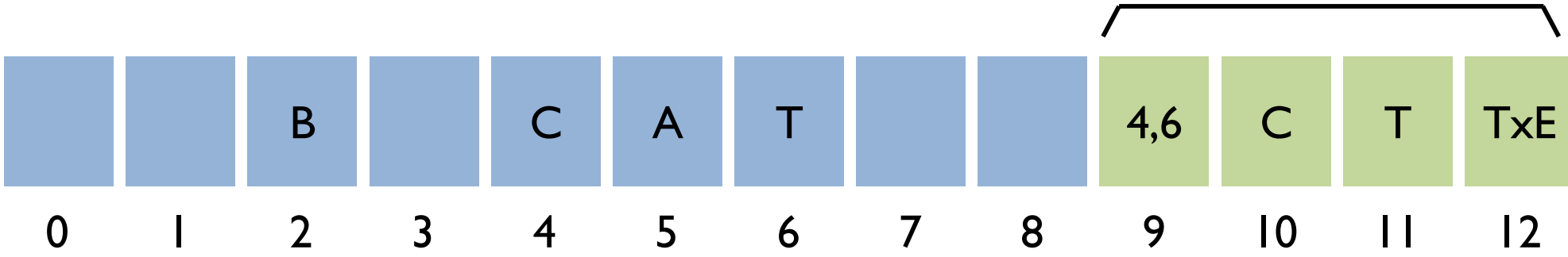
Checkpoint: Writing new data to in-place locations

JOURNAL REUSE AND CHECKPOINTS



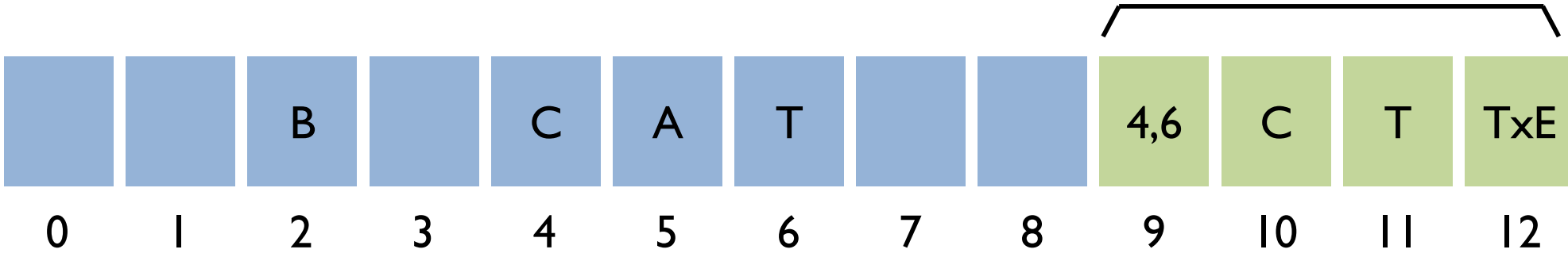
After first transaction checkpoints, can begin next one
Next transaction: write C to block 4; write T to block 6

JOURNAL REUSE AND CHECKPOINTS



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

ORDERING FOR CONSISTENCY

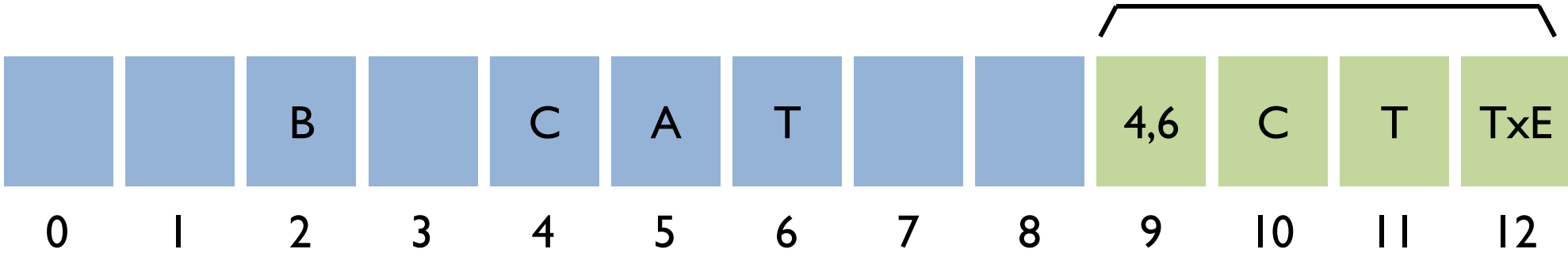


What operations can proceed in parallel and which must be strictly ordered?

Strict ordering is expensive:
must flush from memory to disk
tell disk not to reorder
tell disk can't cache, must persist to final media

writes: 9,10,11, 12, 4, 6,12

ORDERING FOR CONSISTENCY

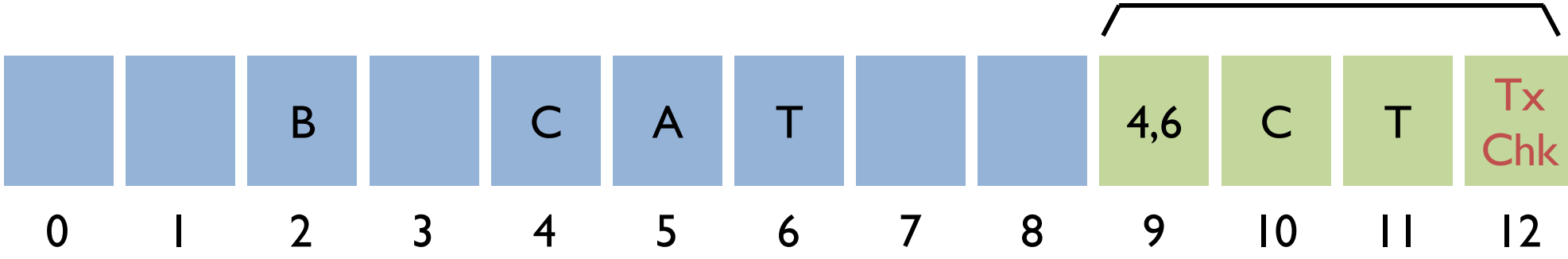


transaction: write C to block 4; write T to block 6

Barriers

CHECKSUM OPTIMIZATION

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

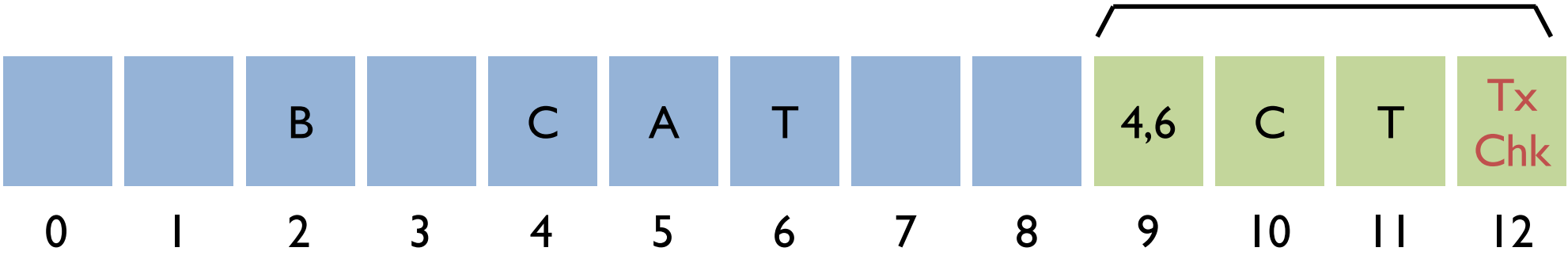


write order: 9,10,11,12 | 4,6 | 12

In last transaction block, store checksum of rest of transaction
(Calculate over blocks 9, 10, 11)

How does recovery change?

WRITE BUFFERING OPTIMIZATIONS



Batched updates

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

Delay checkpoints

- Note: after journal write, there is no rush to checkpoint
- If system crashes, still have persistent copy of written data!
- Journaling is sequential (fast!), checkpointing is random (slow!)
- Solution? Delay checkpointing for some time

CIRCULAR LOG

Difficulty: need to reuse journal space

Solution: keep many transactions for un-checkpointed data

Must wait until transaction T1 is checkpointed before it can be freed and reused for next transaction, T5

HOW TO AVOID WRITING ALL DISK BLOCKS TWICE?

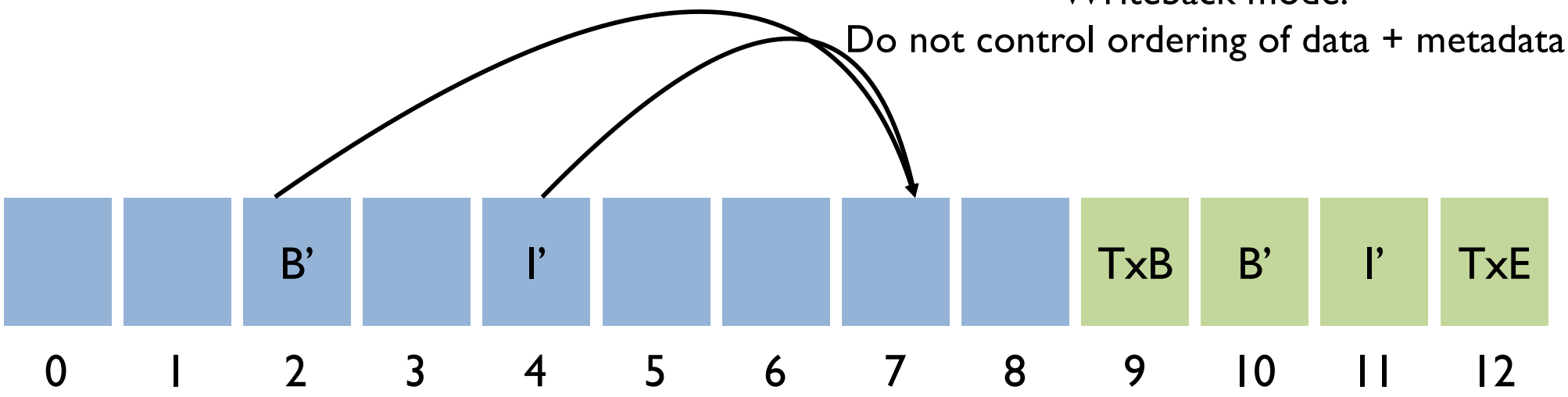
Observation:
Most of writes are user data (esp sequential writes)
If user data isn't consistent, file system still operates correctly

Strategy: Journal all metadata, including
superblock, bitmaps, inodes, indirects, directories
- Guarantees metadata is consistent if crash occurs
- Won't lead or re-allocate blocks to multiple files

For regular user data, write it back whenever convenient
- Implication: Files may contain garbage (partial old, partial new) if crash and recover
- Application needs to deal with this if it cares

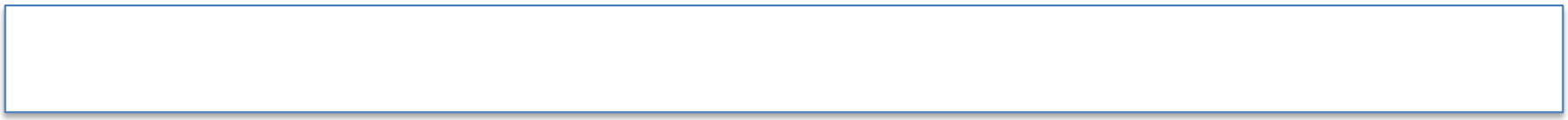
METADATA JOURNALING

Writeback mode:
Do not control ordering of data + metadata



transaction: append to inode I
Ensures B' and I' are updated atomically

Crash !?!

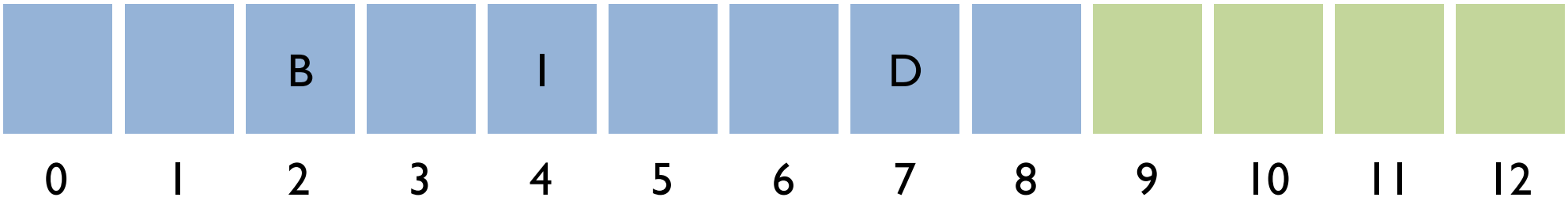


ORDERED JOURNALING

Still only journal metadata

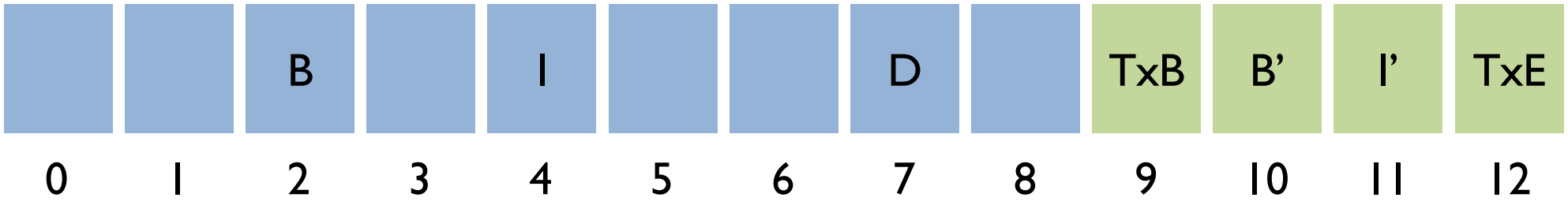
But write data **before** the transaction!

ORDERED JOURNAL



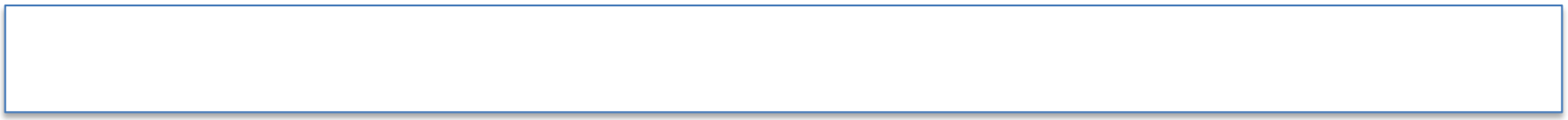
What happens if crash now? (before update B and I)
B indicates D currently free, I does not point to D;
Lose D, but that might be acceptable

ORDERED JOURNALING

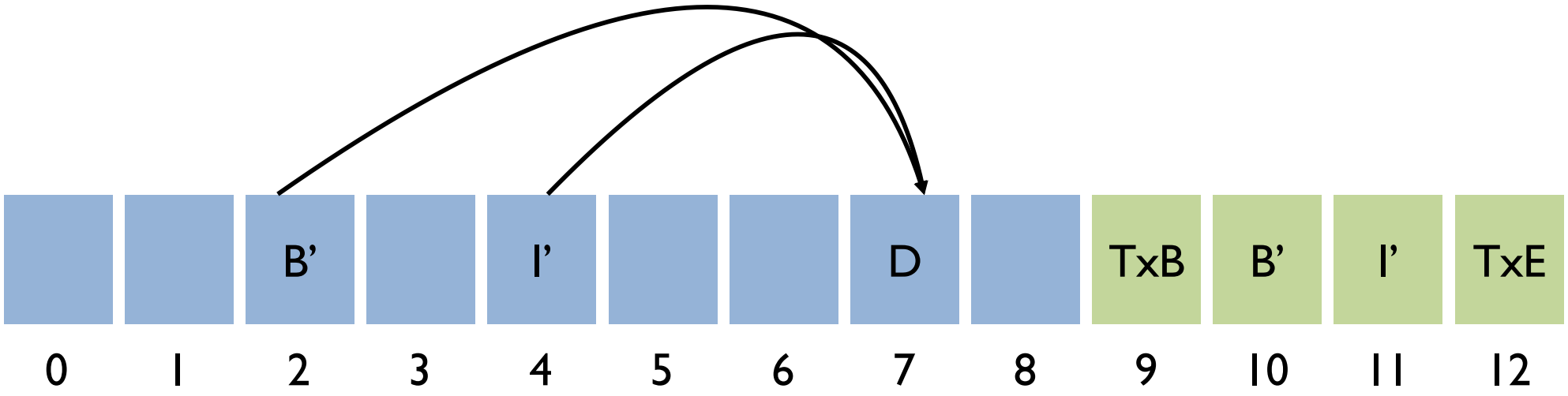


transaction: append to inode I

Crash !?!



ORDERED JOURNALING



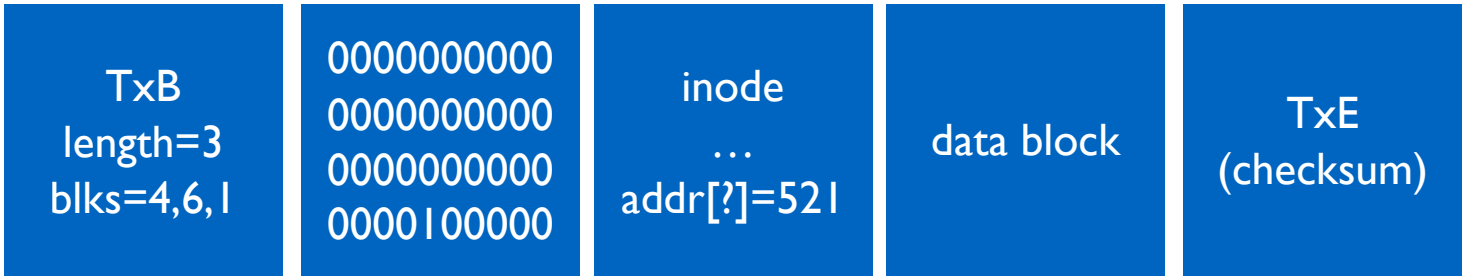
transaction: append to inode I
Ensures B' and I' are updated atomically AFTER D is on disk

Crash !?!

Everything is all good; if didn't clear TxE, will replay transaction, extra work but no problems

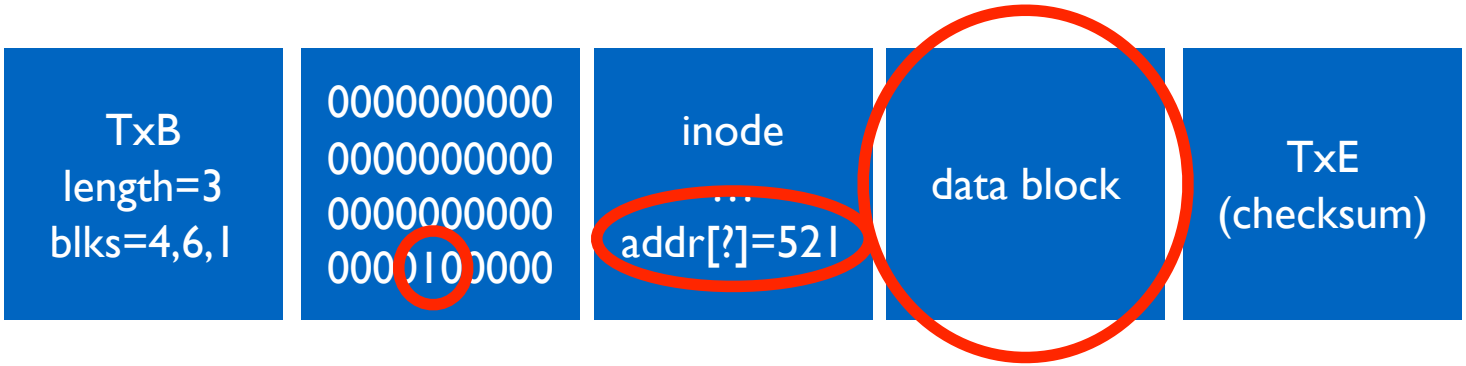
OTHER JOURNALING APPROACHES...

PHYSICAL JOURNAL



Write out lots of information, but how much was really needed?

PHYSICAL JOURNAL



Actual changed data is much smaller!

LOGICAL JOURNAL

Logical journals record **changes** to bytes, not contents of new blocks

On recovery:
Need to read existing contents of in-place data and (re-)apply changes

SUMMARY

Crash consistency: Important problem in file system design!

Two main approaches

FSCK

- Fix file system image after crash happens
- Slow and only ensures consistency
- Still useful to fix problems (bit flips, FS bugs)

Journaling

- Write a transaction before in-place updates
- Checksum, batching, ordered journal optimizations
- Used by most modern file systems
  - Ordered-mode for meta-data is very popular (default mode)

Some file systems don't use journals, but still write new data before deleting old

- Copy-on-write file systems next