PERSISTENCE: JOURNALING

Kai Mast

CS537

Fall 2022

Announcements

- No discussion tomorrow
- Schedule change: We will cover NFS on Thursday
- Please fill out course evaluations

RECAP: FILE SYSTEM ORGANIZATION

Each file has a unique identifier (index number, inode number, or i-number)

Most support **three types of files**

- *Regular files:* User-defined content, opaque to the file system
- *Directories:* Contains list of file names and their inode number
- *Symbolic Links:* Contains path of the file they point to

What about **hard links**?

- Multiple directories can point to the same file

RECAP: FILE SYSTEM API

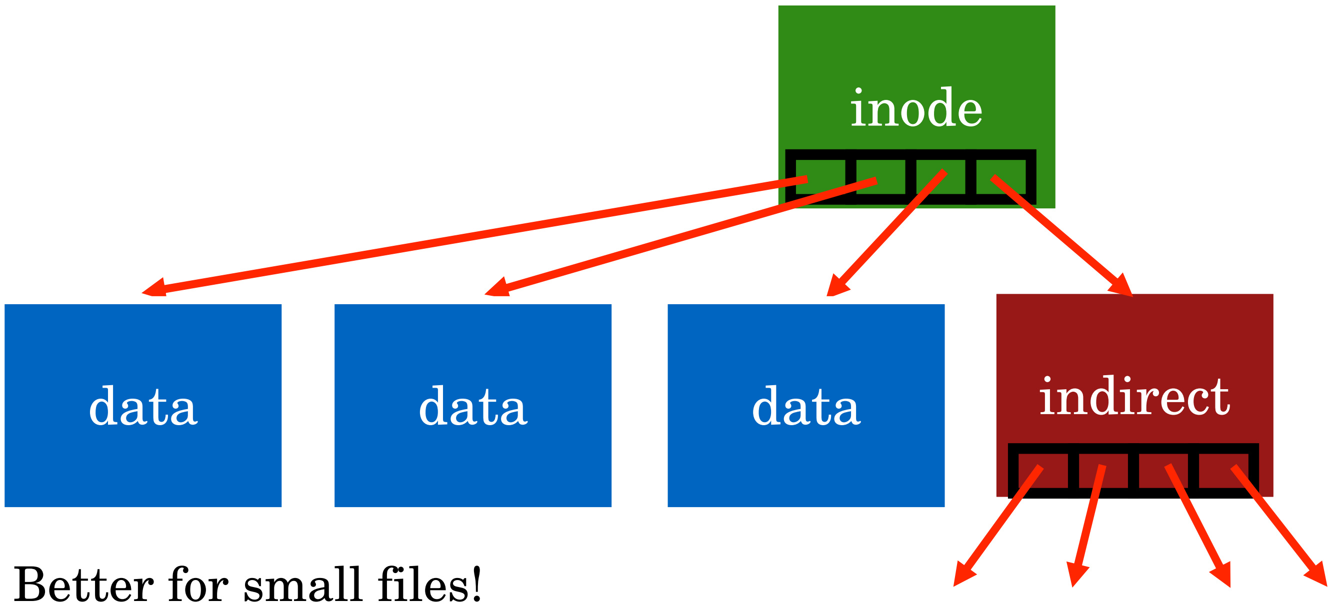
UNIX-style API

- Create new files or open existing files using `open()`
- Access file contents using `read()`/`write()`
- Create directories using `mkdir()`
- Delete files (or directories, hard links) using `unlink()`

When are files actually removed?

- Once the last reference to them is dropped
- References can be held by directories or processes

RECAP: INODES

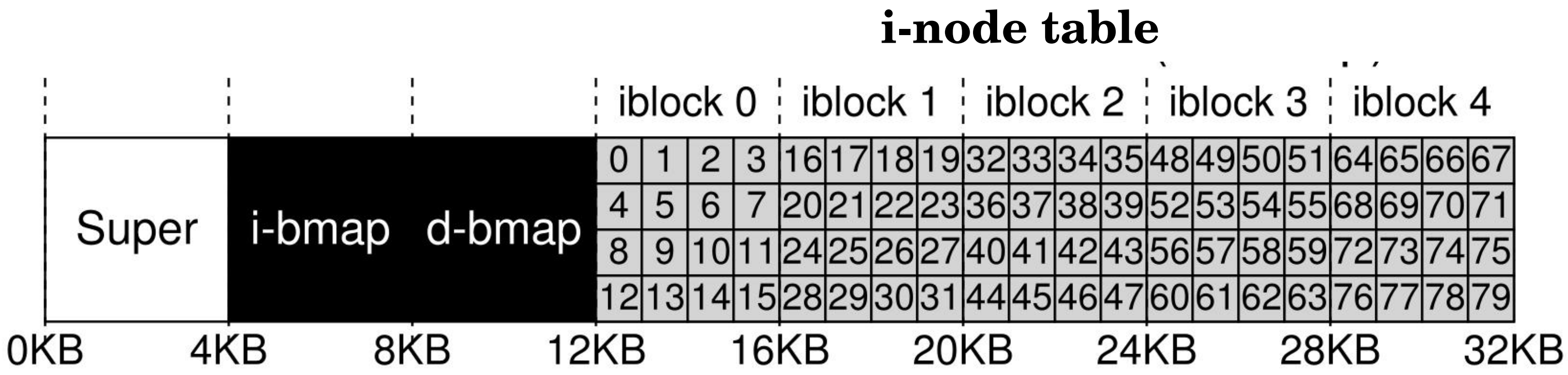


Better for small files!
How to handle even larger files?

Additional Indirection: Double indirect blocks, Triple indirect blocks...

RECAP: FILE SYSTEM ORGANIZATION

Very simple file system (vsfs) from the textbook



Superblock: Parameters of the file system (e.g., how many inodes)

i-bitmap: Which inodes are in use?

d-bitmap: Which data blocks are in use?
(Data blocks are not shown on the above figure)

FSCK AND JOURNALING

(Book Chapter 42)

DATA REDUNDANCY

Informal Definition:

- Data is stored in two or more locations
- If one location is corrupted we can (partially) recover it from the other(s)

RAID examples:

- mirrored disk (e.g., RAID 1)
- parity blocks (e.g., RAID 4)

File system examples:

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

FILE SYSTEM CONSISTENCY EXAMPLE

Superblock: One field contains total number of blocks in FS
value = N

Inode:

- field contains index of data block
- possible values? {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

- Two blocks may conflict with each other; which one is correct?

Only single sector writes are guaranteed to be atomic by disk

What can interrupt write operations?

- power loss
- kernel panic
- reboot

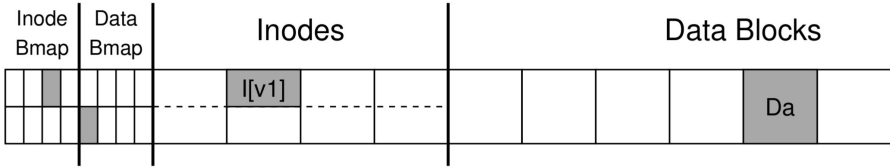
EXAMPLE

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

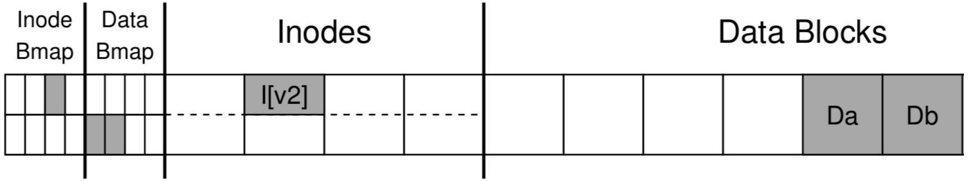
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
							write
				write			

FILE APPEND EXAMPLE

Old State



New State (if all writes succeed)



What if only some writes succeed?

Written to disk	Result
Db	Lost data (nothing bad)
I[v2]	Point to garbage; another file could use data block
B[v2]	Space leak; block marked allocated
I[v2] + B[v2]	Point to garbage data
I[v2] + Db	Another file could use same data block
B[v2] + Db	Space leak; Inode doesn't point to the block

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 repaired

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 ".."? Point to correct inode numbers?

`size` and `numblocks` in inodes match?

Is the bitmap of free blocks correct?

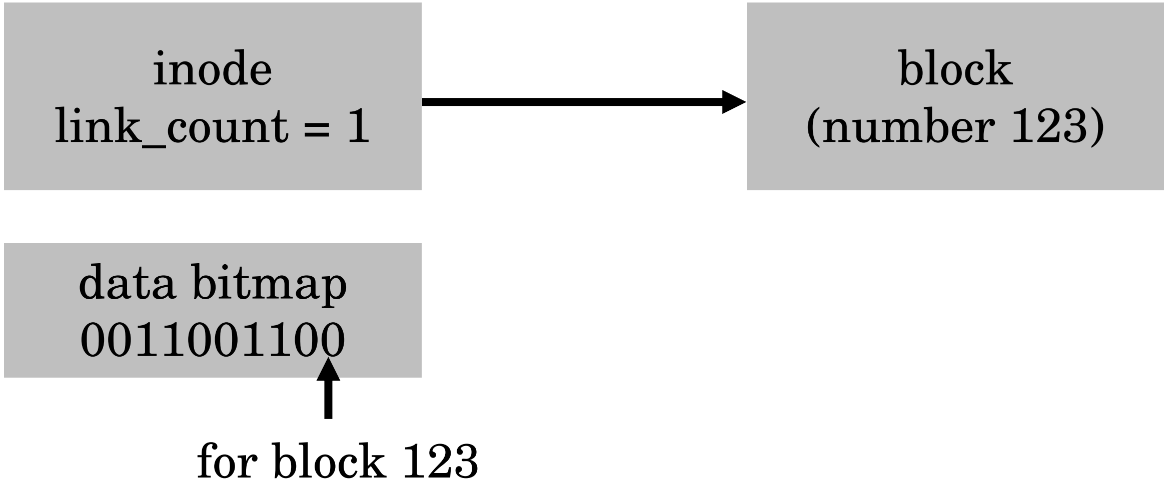
Do number of dir entries equal inode link counts?

Do different inodes ever point to same block?

Are there any bad block pointers?

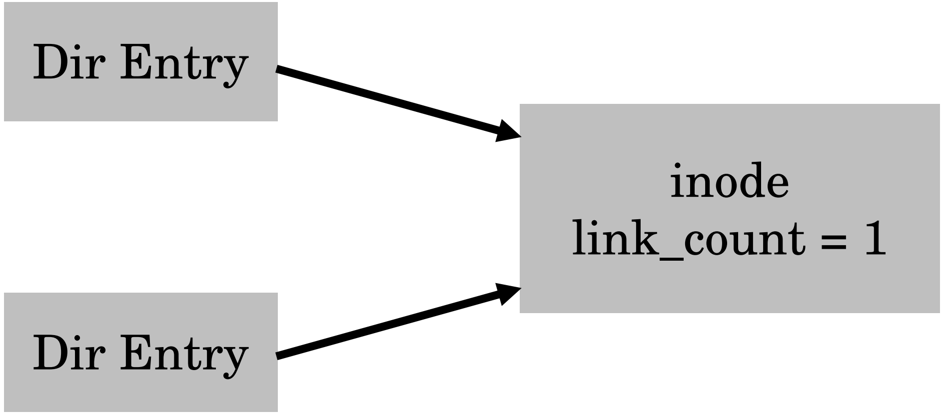
…

FREE BLOCKS EXAMPLE



How do we fix this to have a consistent file system?
Set bit referencing block 123 to 1

LINK COUNT EXAMPLE 1



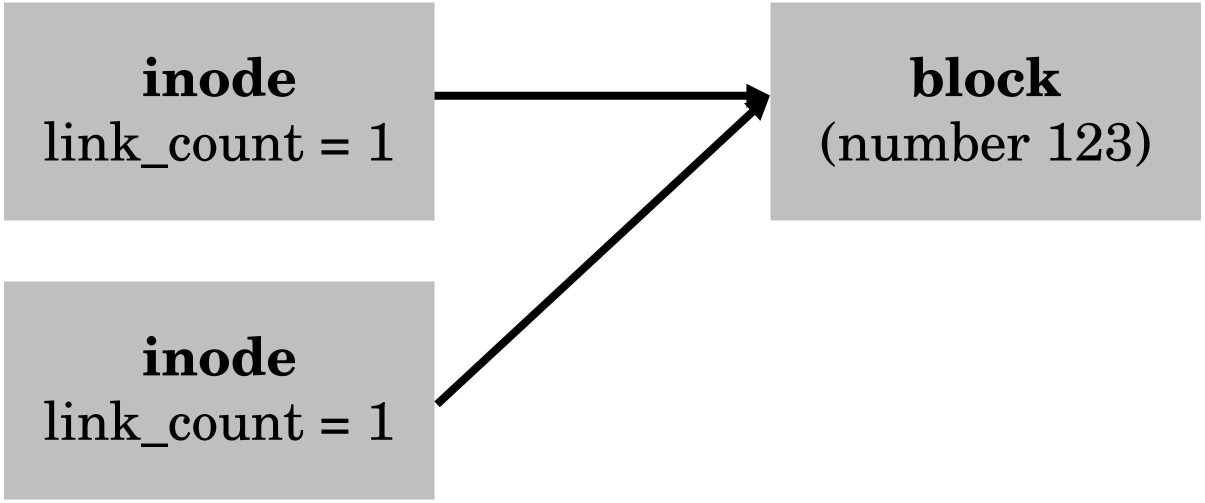
How do we fix this to have a consistent file system?
Update `link_count` in inode

LINK COUNT (EXAMPLE 2)

inode
link_count = 1

How do we fix this to have a consistent file system?
Create reference to inode in special directory, e.g., `/lost+found` on Linux

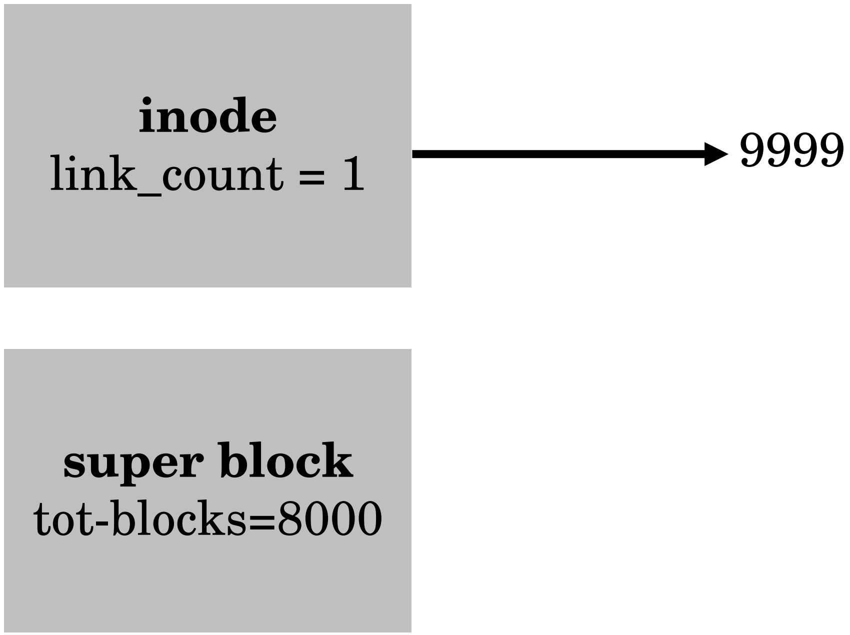
DUPLICATE POINTERS



How do we fix this to have a consistent file system?

- Duplicate block so each node has points to a different one
- This probably means one inode will point to corrupted data

BAD POINTER



How do we fix this to have a consistent file system?

Remove reference to invalid data block

PROBLEMS WITH FSCK

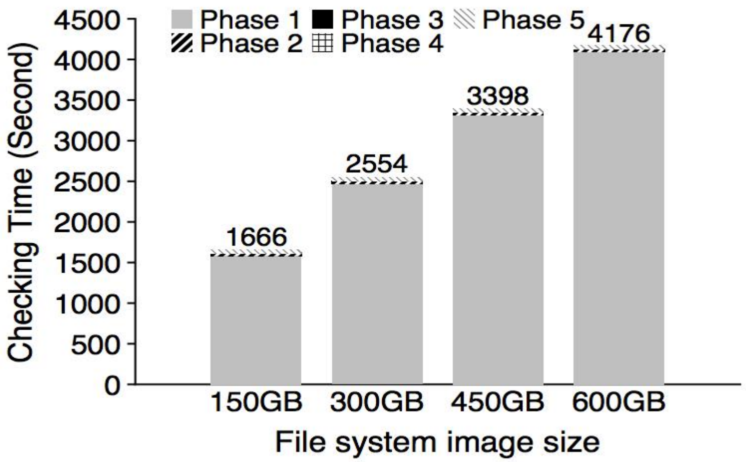
Problem 1:

- Cannot always know how to fix file system image; we might not have enough information

- No way of knowing the “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

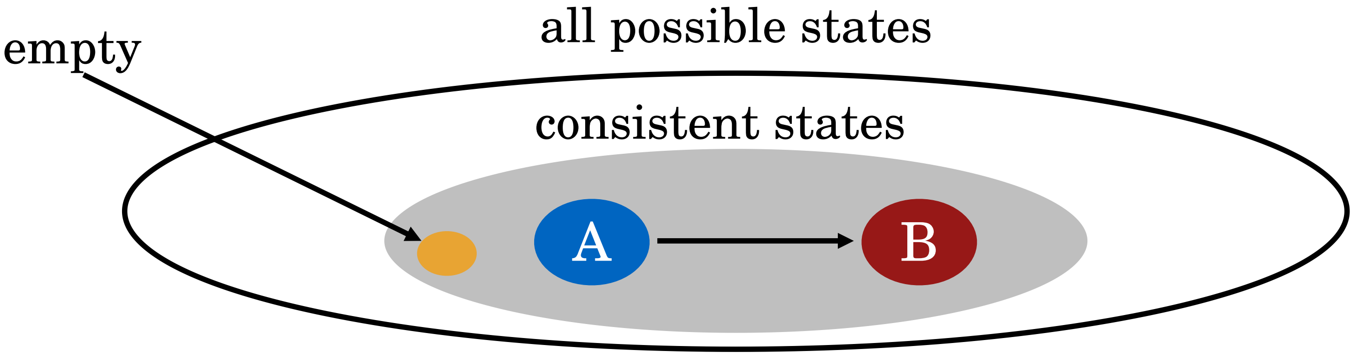
- Do some **recovery work** after crash, but don't read entire disk
- Don't move file system to just any consistent state, get **correct** state

Atomicity

- Definition of atomicity for **concurrency:**
  - Operations in critical sections do not overlap with 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
  - Disks only guarantee writes of individual sectors to be atomic

CONSISTENCY VS ATOMICITY

Set of writes moves the disk from state A to B



Atomicity gives A or B

fsck only gives consistency

JOURNALING: GENERAL STRATEGY

Don't delete any old info until all new info is safely on disk

(Ironically, we need to 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, update file metadata and data
3. Remove note

If crash and recover:

1. If a note is not completely written, ignore note (old data still good)
2. If a note is completely written, *replay* it to recover data

Journaling Terminology

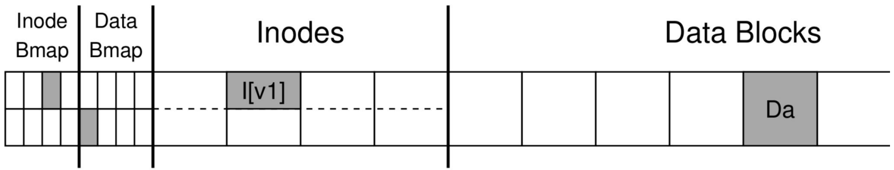
Journal: Blocks designated to store notes

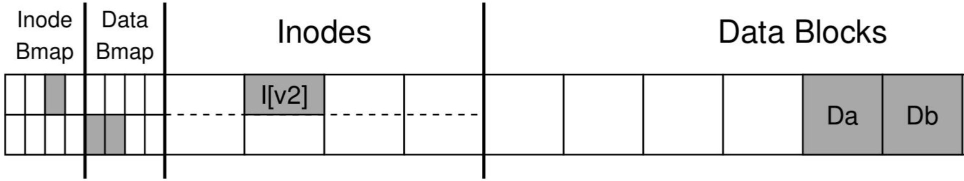
(Journal) Transaction:

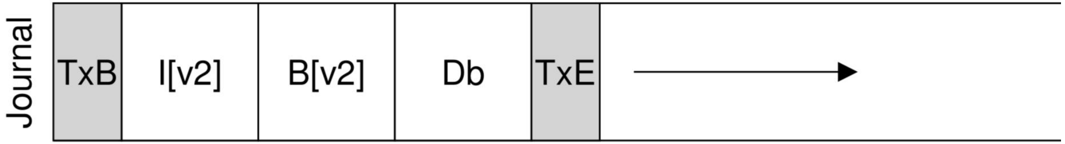
- Set of writes that “belong together” (should execute atomically)
- Last part of a transaction is its *commit block*
    - Transaction is considered *committed* after this block is written

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

DISK AND JOURNAL LAYOUT



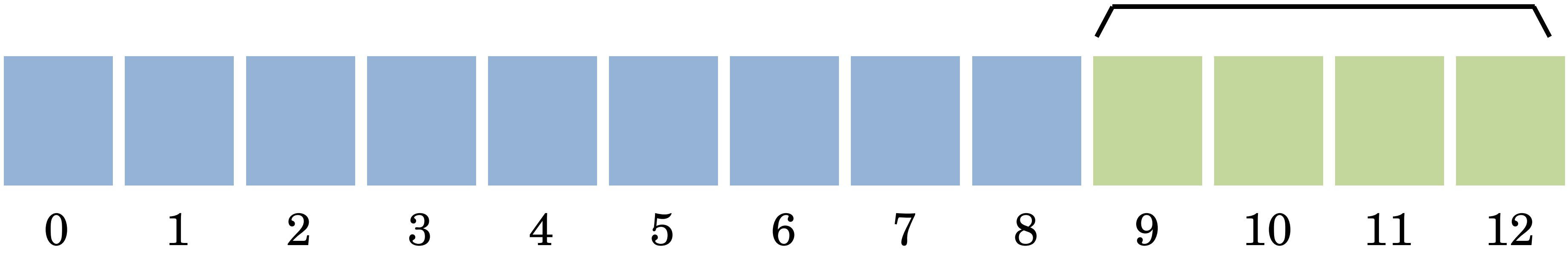




Transaction

JOURNAL WRITE AND CHECKPOINTS

Goal: write A to block 5; write B to block 2; make atomic!

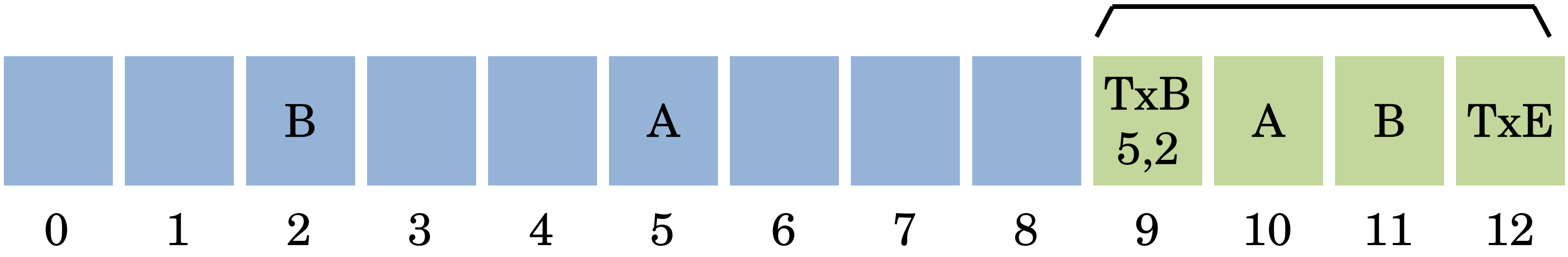


TxB ("Begin Transaction"): Holds unique id and blocks affected
TxE ("End Transaction"): Indicates transaction has committed

Checkpoint: Write new data to in-place locations

What happens if crash occurs after each write?
After and replay journal, what data will exist on disk?

JOURNAL REUSE AND CHECKPOINTS



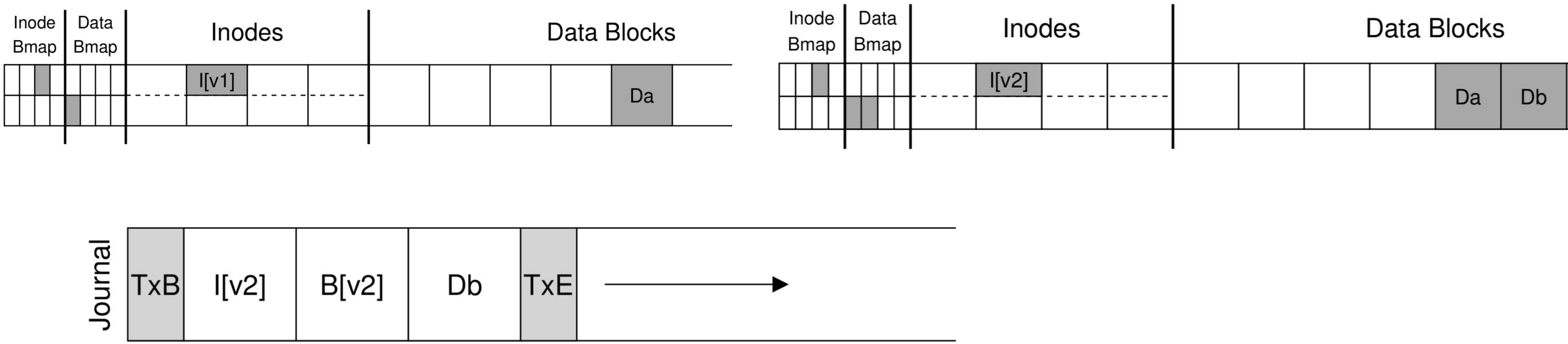
After a transaction checkpoints, file system can overwrite it

- Update TxE to indicate this

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

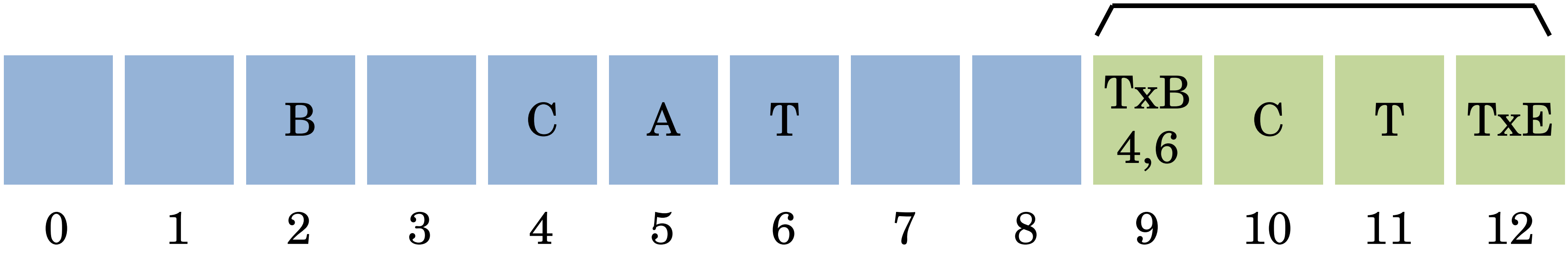
- Will overwrite journal entries in blocks 9 to 12

REAL SYSTEMS



- Batch or group Transactions
  - For performance, many operations are placed in single transaction
- Journal is large; treat as circular buffer
- Checkpoint periodically

ORDERING FOR CONSISTENCY



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

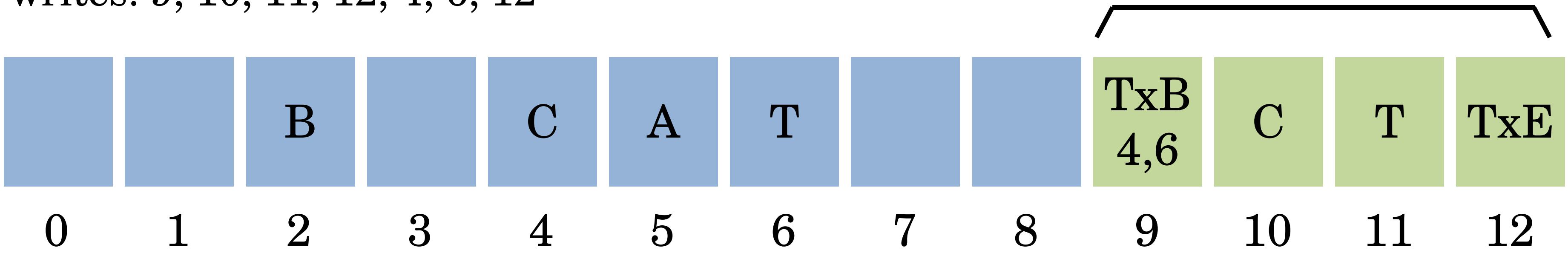
Strict ordering is expensive:

- 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

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



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

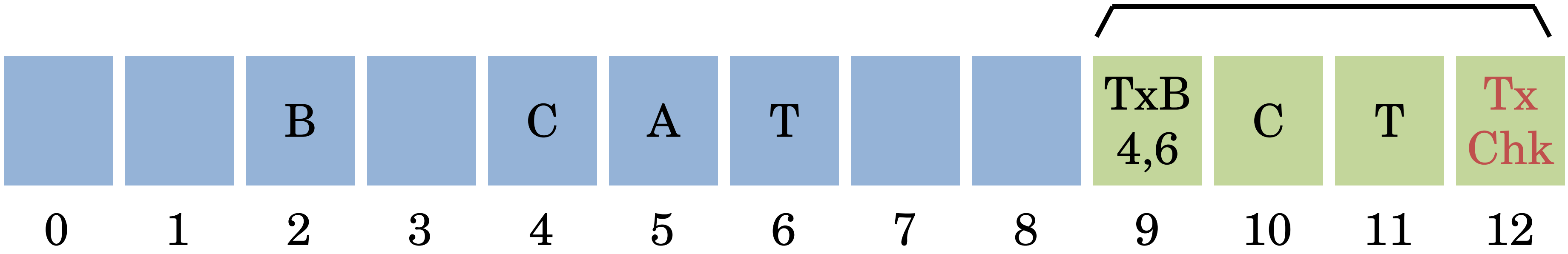
Barriers

1) Before journal commit, ensure journal transaction entries complete
2) Before checkpoint, ensure journal commit complete
3) Before free journal, ensure checkpoint (in-place updates) complete

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

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?
During recovery: If checksum does not match, treat as not valid