Hello!

PERSISTENCE: FSCK, JOURNALING

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Project 5 updates →

Midterm 2: Solutions, grades → Q33

Next week's schedule

→ Tue: LFS

Thu: SSD

Zoom OHs:

AGENDA / LEARNING OUTCOMES

How to check for consistency with power failures / crashes?

How to ensure consistency in filesystem design?

RECAP

open read write

FS API

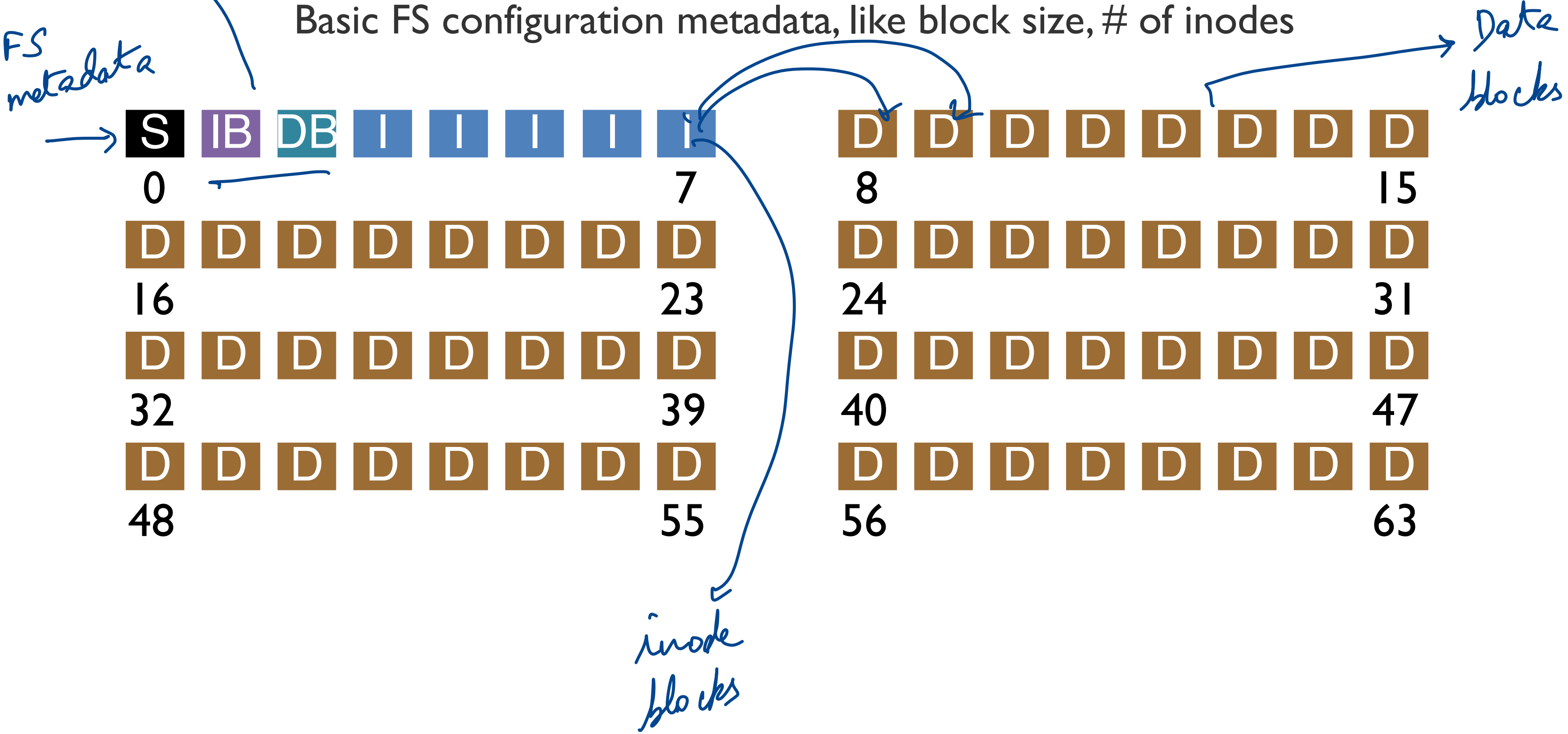
File Systems → Design

Disks

FS STRUCTS: SUPERBLOCK

free data blocks
inode

Basic FS configuration metadata, like block size, # of inodes



one high level

create /foo/bar

TIME

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		1. read				
					2. read	
			3. read			
						4. read
	5.read 6.write					
						7.write
				8.read 9.write		
			10.write			

many disk operations

FFS PLACEMENT GROUPS

tracks within this group

fast file systems

cylinder number

replicated

Key idea: Keep inode close to data

accessing blocks within the group is cheaper

Use groups across disks;

Strategy: allocate inodes and data blocks in same group.

FFS STRATEGY

within the same group

dir go into new groups

Put more-related pieces of data near each other

Put less-related pieces of data **far**

```
group inodes     data
    0 /--------- /---------
    1 acde------ accddee---
    2 bf-------- bff-------
    3 ---------- ----------
    4 ---------- ----------
    5 ---------- ----------
    6 ---------- ----------
    7 ---------- ----------
  ...
```

FS tree

```
/a/b e
/a/c
/a/d
/b/f
```

/

↳ /a

/b

2 dirs

4 files

/a/b
/a/c
/a/d
/b/f

files go with parent dir

PROBLEM: LARGE FILES

~ 80-20 behavior

small

large

most of the bytes

Single large file can fill nearly all of a group

Displaces data for many small files

```
group inodes     data
    0 /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
    1 ---------- ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ---------- ----------
    ...
```

Most files are small!

Better to do one seek for large file than one seek for each of many small files

SPLITTING LARGE FILES

same indirect block
~ 4 MB

≤ 48KB in size ← small files

inode

direct ptr ....

indirect

Data blocks

```
group inodes       data
    0 /a---------- /aaaaa----- ----------- ----------- -----------
    1 ------------ aaaaa------ ----------- ----------- -----------
    2 ------------ aaaaa------ ----------- ----------- -----------
    3 ------------ aaaaa------ ----------- ----------- -----------
    4 ------------ aaaaa------ ----------- ----------- -----------
    5 ------------ aaaaa------ ----------- ----------- -----------
    6 ------------ ----------- ----------- ----------- -----------
  ...
```

Define "large" as requiring an indirect block

Starting at indirect (e.g., after 48 KB) put blocks in a new block group.

Each chunk corresponds to one indirect block

Block size 4KB, 4 byte per address => 1024 address per indirect

1024*4KB = 4MB contiguous "chunk"

POLICY SUMMARY

parent

File inodes: allocate in same group with dir

Dir inodes: allocate in new group with fewer used inodes than average group

balance inode usage

First data block: allocate near inode

Other data blocks: allocate near previous block

Large file data blocks: after 48KB, go to new group.

Move to another group (w/ fewer than avg blocks) every subsequent 4MB.

balance data blocks usage

OTHER FFS FEATURES

FFS also introduced several new features:

- large blocks (with libc buffering / fragments)
- long file names
- atomic rename
- symbolic links

Inspired modern files systems, including ext2 and ext3

FILE SYSTEM CONSISTENCY

FILE SYSTEM CONSISTENCY EXAMPLE

Superblock: field contains total number of blocks in FS

DATA = N

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

DATA in {0, 1, 2, …, N - 1}

Pointers to block N or after are invalid!

Total-blocks field has redundancy with inode pointers

Consistency requirement

inode points to a valid DB

WHY IS CONSISTENCY CHALLENGING?

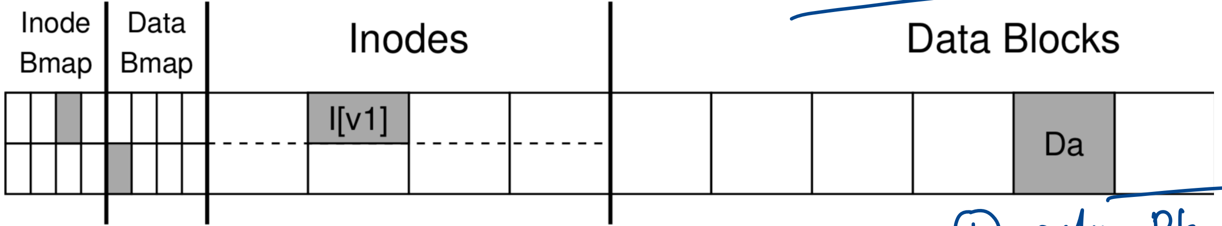
File system may perform several disk writes to redundant blocks

If file system is interrupted between writes, may leave data in inconsistent state

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

FILE APPEND EXAMPLE

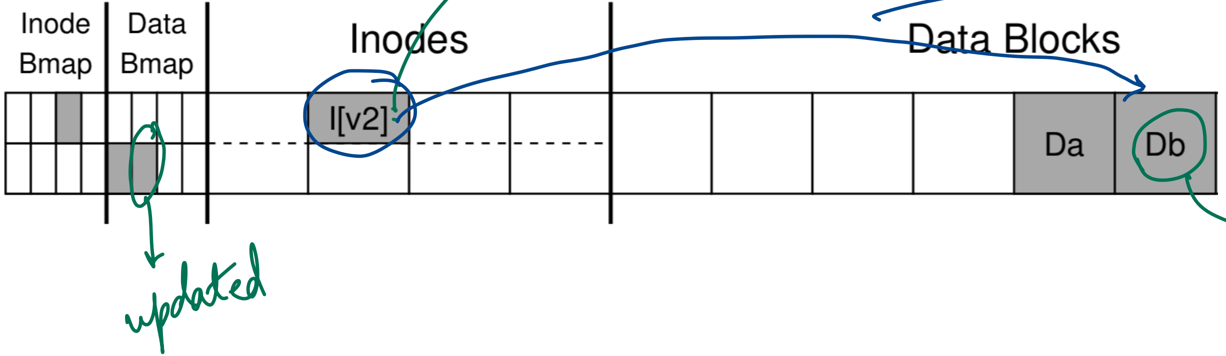
(3) only DB is on disk: → lose some disk space



→ Old state

(1) only Db is on disk: lose this block

(2) only inode is on disk: garbage data
inconsistency DB & inode



→ Target state

new data

HOW CAN FILE SYSTEM FIX INCONSISTENCIES?

Solution #1:

FSCK = file system checker

when you reboot

Strategy:

After crash, scan whole disk for contradictions and "fix" if needed

Keep file system off-line until FSCK completes

matches bitmap

all ptrs to data blocks

For example, how to tell if data bitmap block is consistent?

Read every valid inode+indirect block

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

FSCK CHECKS

Do superblocks match?

Is the list of free blocks correct? bitmap

Do number of dir entries equal inode link counts? hard links

Do different inodes ever point to same block?

Are there any bad block pointers?

Do directories contain "." and ".."?

…

FREE BLOCKS EXAMPLE

inode
link_count = 1

block
(number 123)

data bitmap
001100110~~0~~ 1

for block 123

Fix is update the bitmap
↳ consistent
"correct" ??

LINK COUNT EXAMPLE

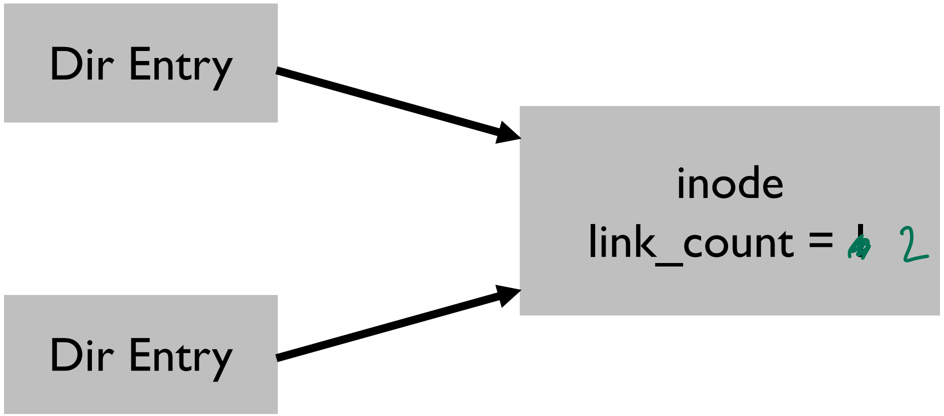
name inode

foo 25

name inode

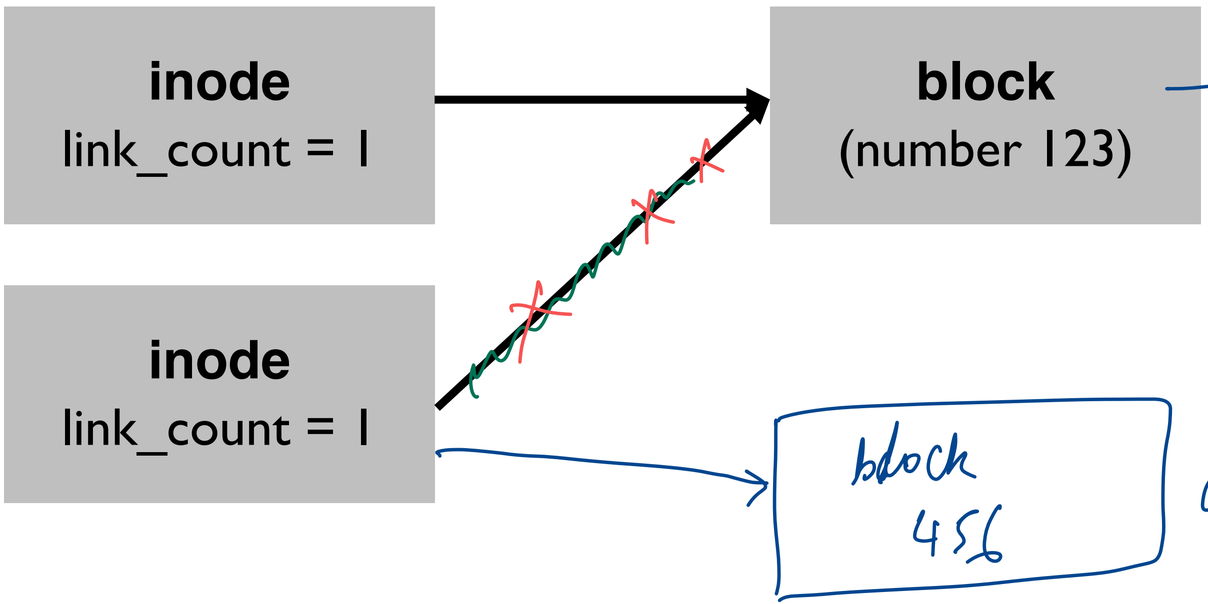
bar 25

hard link



→ makes it consistent

DUPLICATE POINTERS



a block can only belong to 1 file

Copy contents 123 to 456

BAD POINTER

inode
link_count = 1

→ ~~9999~~

ptr is pointing to invalid block

super block
tot-blocks=8000

fix
↳ clear this pointer

Consistent

QUIZ 17

inode size = 256 bytes
block size = 4KB

Offset for inode with number 0 (in kB)? 12 KB

Offset for inode with number 4 (in kB)? 12 KB + 4 × 256

= 13 KB

data block

dir

ref count

```
inode bitmap   10000000
inodes         [d a:0 r:2] [] [] [] [] [] [] []
data bitmap    10000000
data           [(.,0) (..,0)] [] [] [] [] [] [] []
```

empty

```
inode bitmap 11100000
inodes       [d a:0 r:4] [d a:1 r:2] [d a:2 r:2] [] [] [] [] []
data bitmap  11100000
data         [(.,0) (..,0) (d,1) (w,2)] [????] [(.,2) (..,0)] [] [] [] [] []
```

what are the operations ?

2 new things in root dir

they are directories

(.,1) (..,0)

```
inode bitmap   10000000
inodes         [d a:0 r:2] [] [] [] [] [] [] []
data bitmap    10000000
data           [(.,0) (..,0)] [] [] [] [] [] [] []
```

empty

```
inode bitmap   11000000
inodes         [d a:0 r:2] [f a:1 r:2] [] [] [] [] [] []
data bitmap    ????????
data           [(.,0) (..,0) (c,1) (m,1)] [foofoofoo] [] [] [] [] [] []
```

/ < file > created

Proper root dir ✓

/c → same inode
/m →

hard links ✓

✓ Contents of /c (and /m) is foofoofoo

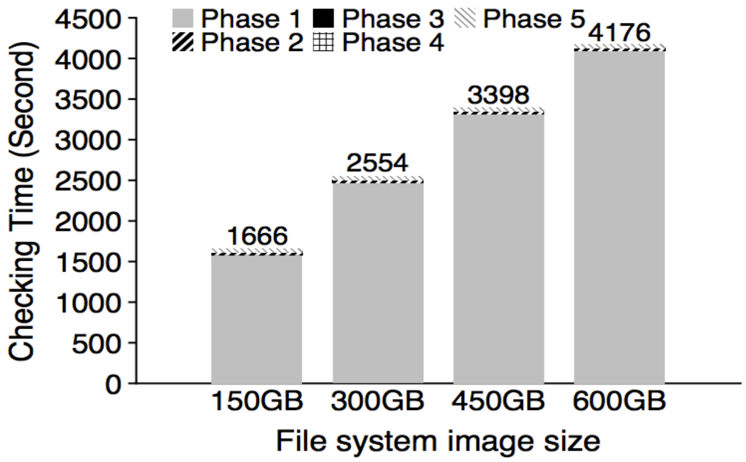
PROBLEMS WITH FSCK

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:

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

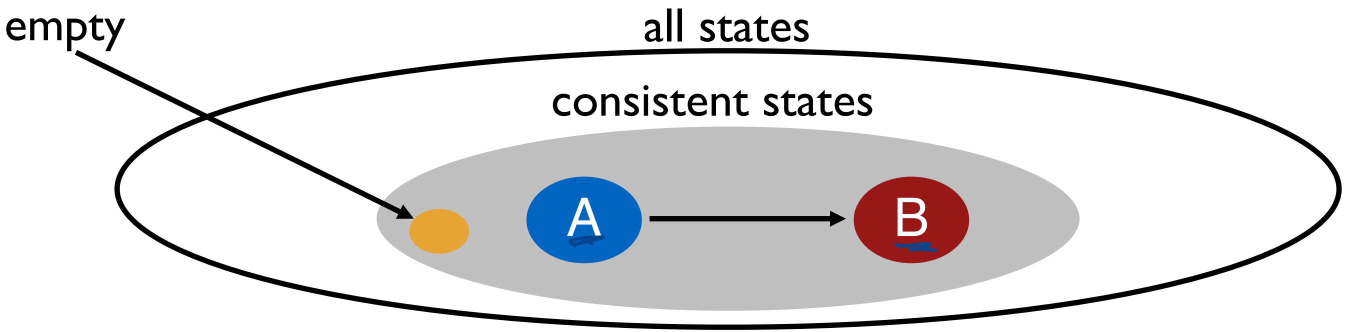
Atomicity

- 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

database

CONSISTENCY VS ATOMICITY

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



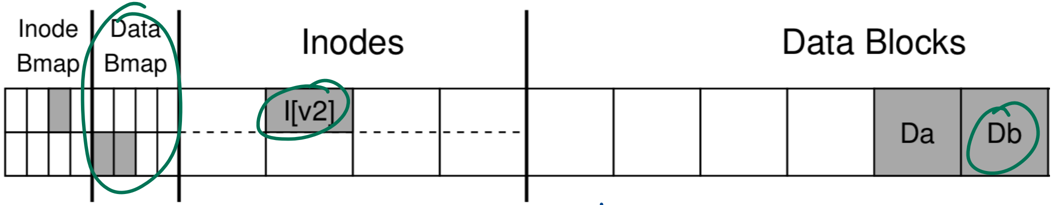
fsck gives consistency
Atomicity gives A or B.

JOURNAL LAYOUT

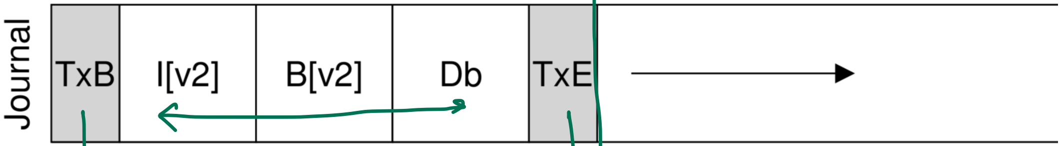
① fail before journal write
↳ old state

② replay writes from journal

Before you update the disk



persist



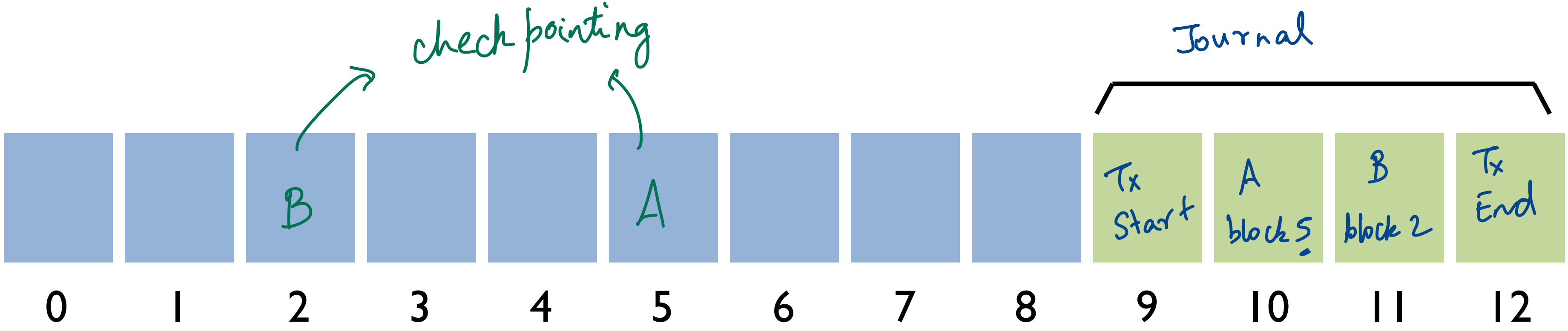
Tx Begin

end

Transaction

↳ write down what you are updating in a Journal

JOURNAL WRITE AND CHECKPOINTS

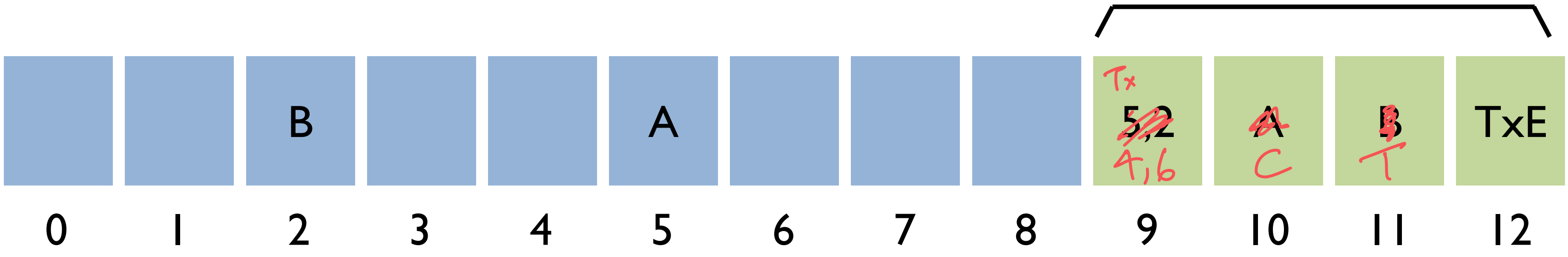


transaction: write A to block 5; write B to block 2

Checkpoint: Writing new data to in-place locations

After checkpointing entries clear journal

JOURNAL REUSE AND CHECKPOINTS



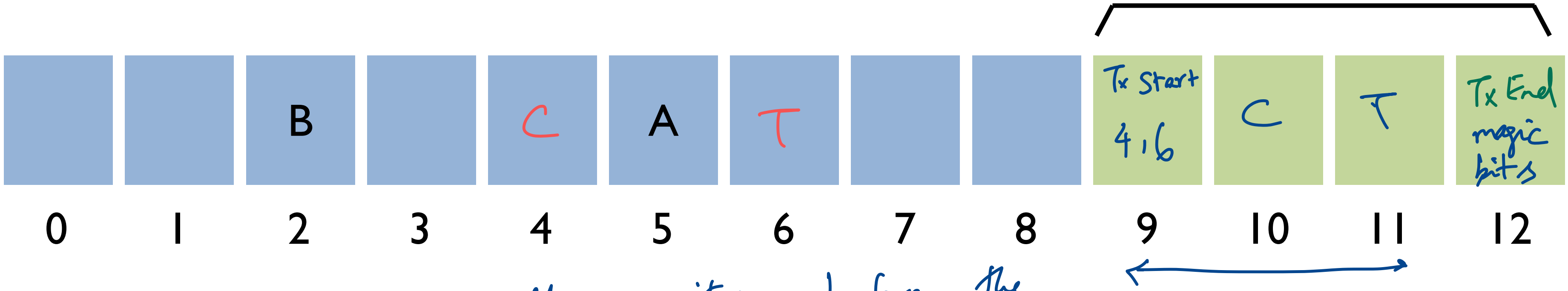
transaction: write A to block 5; write B to block 2

Checkpoint: Writing new data to in-place locations

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

ORDERING FOR CONSISTENCY

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



sync, barrier → all writes before the barrier are persisted when it returns

write order

dont replay → 9,10,11

12

after this replay transaction

4,6

Barriers

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

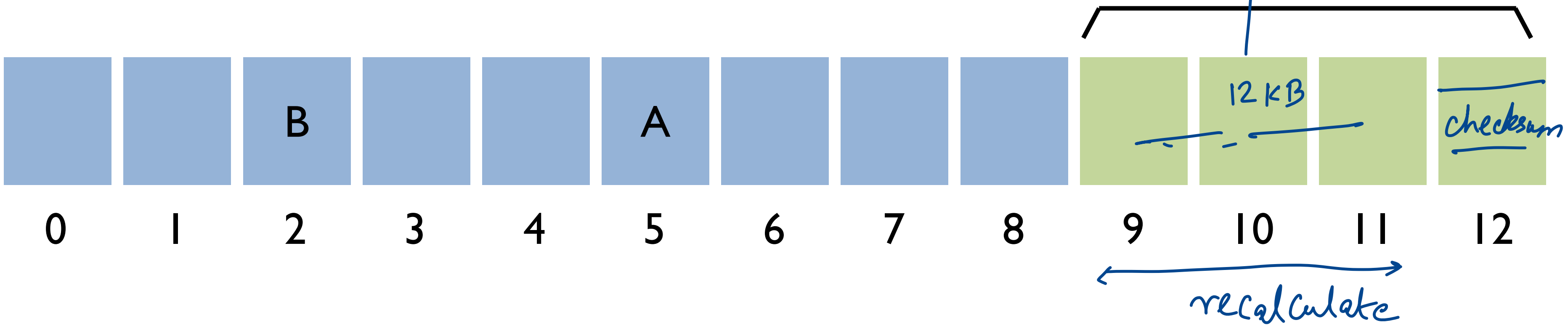
CHECKSUM OPTIMIZATION

md5

SHA

256 bytes

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



In last transaction block, store checksum of rest of transaction

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

write order before

9,10,11

12

4,6

12

write order after

9, 10, 11, 12

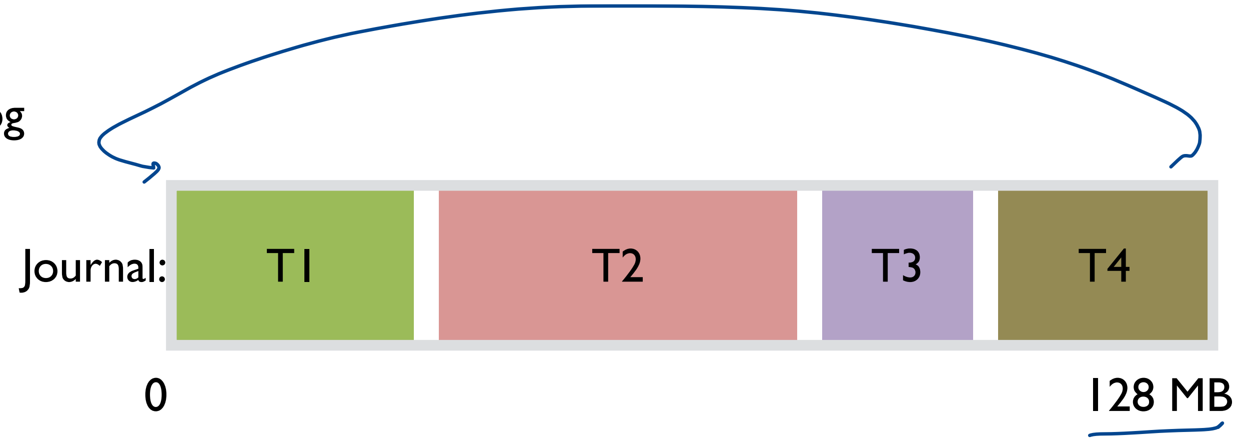
4, 6

OTHER OPTIMIZATIONS

Batched updates

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

Circular log



HOW TO AVOID WRITING ALL DISK BLOCKS TWICE?

Observation: Most of writes are user data (esp sequential writes)

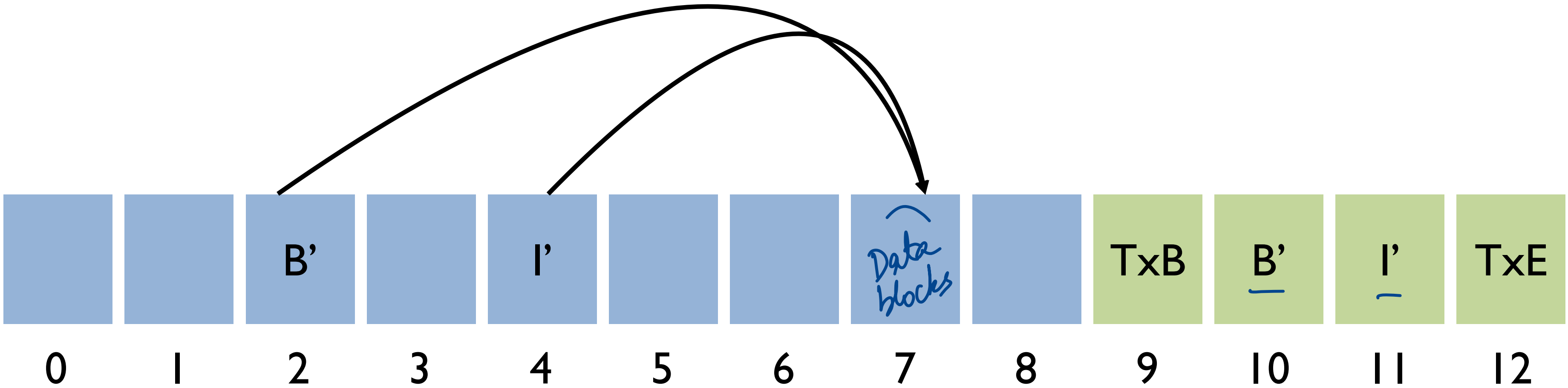
Strategy: journal all metadata, including
superblock, bitmaps, inodes, indirects, directories

For regular data, write it back whenever convenient.

Tx
↳ metadata
bitmaps
inodes

data

METADATA JOURNALING

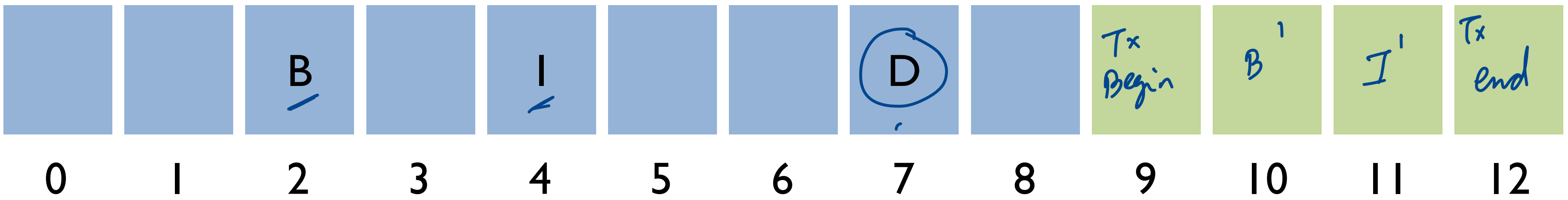


transaction: append to inode I

Crash !?!

ORDERED JOURNALING

Still only journal metadata. But write data **before** the transaction!



What happens if crash in between?

order → data goes first

data block is on disk

barrier

write order

7

9, 10, 11

12

2, 4

SUMMARY

Crash consistency: Important problem in filesystem design!

Two main approaches

FSCK:

- Fix file system image after crash happens
- Too slow and only ensures consistency

Journaling

- Write a transaction before in-place updates
- Checksum, batching, ordered journal optimizations

NEXT STEPS

Next class: How to create a file system optimized for writes