Slides: Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

PERSISTENCE: CRASH CONSISTENCY

Questions answered in this lecture:

What benefits and complexities exist because of data **redundancy?**

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 between these two types of fields? Why or why not?

FILE SYSTEM REDUNDANCY 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

QUESTION FOR YOU…

Give 5 examples of redundancy in FFS (or files system in general)

- Dir entries AND inode table
- Dir entries AND inode link count
- Data bitmap AND inode pointers
- Data bitmap AND group descriptor
- Inode file size AND inode/indirect pointers

…

PROS AND CONS OF REDUNDANCY

Redundancy may improve:

- reliability
  - RAID-5 parity
  - Superblocks in FFS
- performance
  - RAID-1 mirroring (reads)
  - FFS group descriptor
  - FFS bitmaps

Redundancy hurts:

- capacity
- consistency
  - Redundancy implies certain combinations of values are illegal
  - Illegal combinations: inconsistency

CONSISTENCY EXAMPLES

Assumptions:

Superblock: field contains total blocks in FS.

DATA = 1024

Inode: field contains pointer to data block.

DATA in {0, 1, 2, …, 1023}

Scenario 1: Consistent or not?

Superblock: field contains total blocks in FS.

DATA = 1024

Inode: field contains pointer to data block.

DATA = 241

Consistent

Scenario 2: Consistent or not?

Superblock: field contains total blocks in FS.

DATA = 1024

node: field contains pointer to data block.

DATA = 2345

Inconsistent

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

QUESTION FOR YOU…

File system is appending to a file and must update:

- inode
- data bitmap
- data block

What happens if crash after only updating some blocks?

a) **bitmap**: lost block

b) **data**: nothing bad

c) **inode**: point to garbage (what?), **another file may use**

d) **bitmap** and **data**: lost block

e) **bitmap** and **inode**: point to garbage

f) **data** and **inode**: **another file may use**

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?

Read every valid inode+indirect block
If pointer to data block, the corresponding bit should be 1; else bit is 0

FSCK CHECKS

Hundreds of types of checks over different fields…

Do superblocks match?

Do directories contain “.” and “..”?

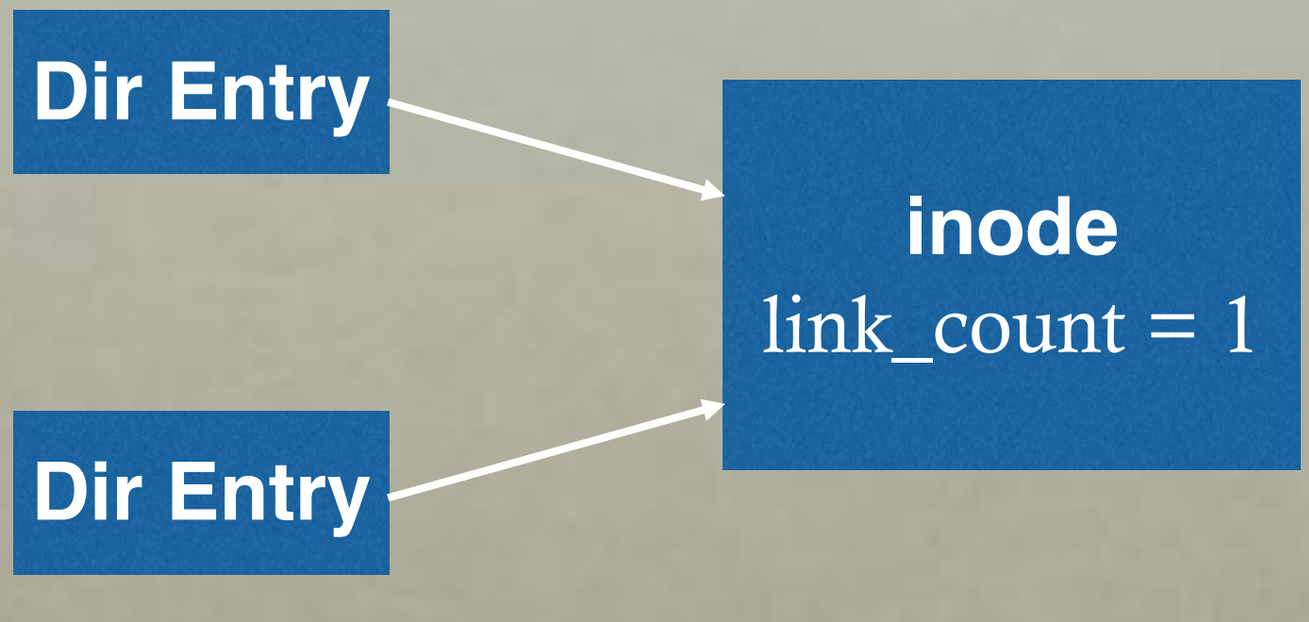
Do number of dir entries equal **inode link counts**?

Do different inodes ever point to **same block**?

…

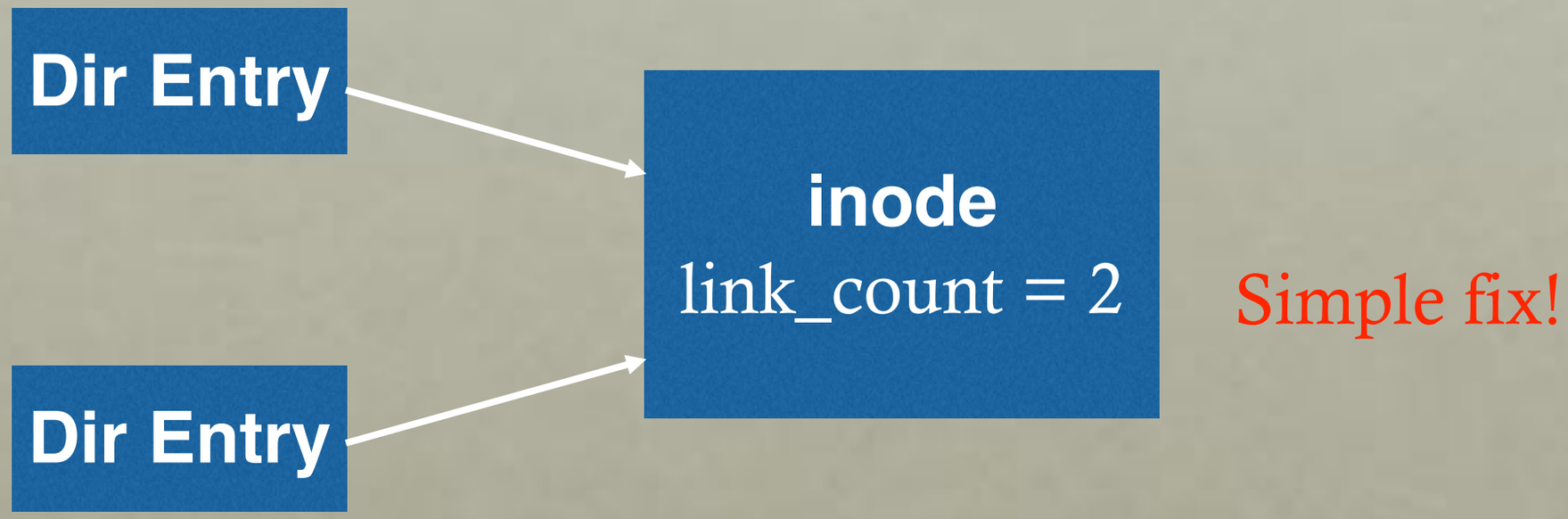
How to solve problems?

LINK COUNT (EXAMPLE 1)



How to fix to have consistent file system?

LINK COUNT (EXAMPLE 1)



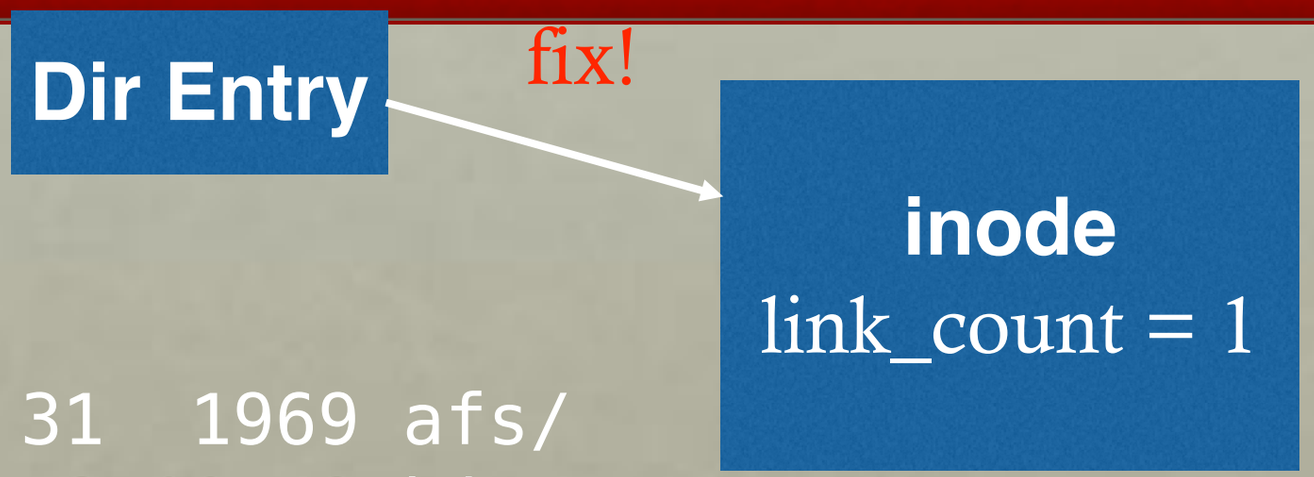
Simple fix!

LINK COUNT (EXAMPLE 2)

inode
link_count = 1

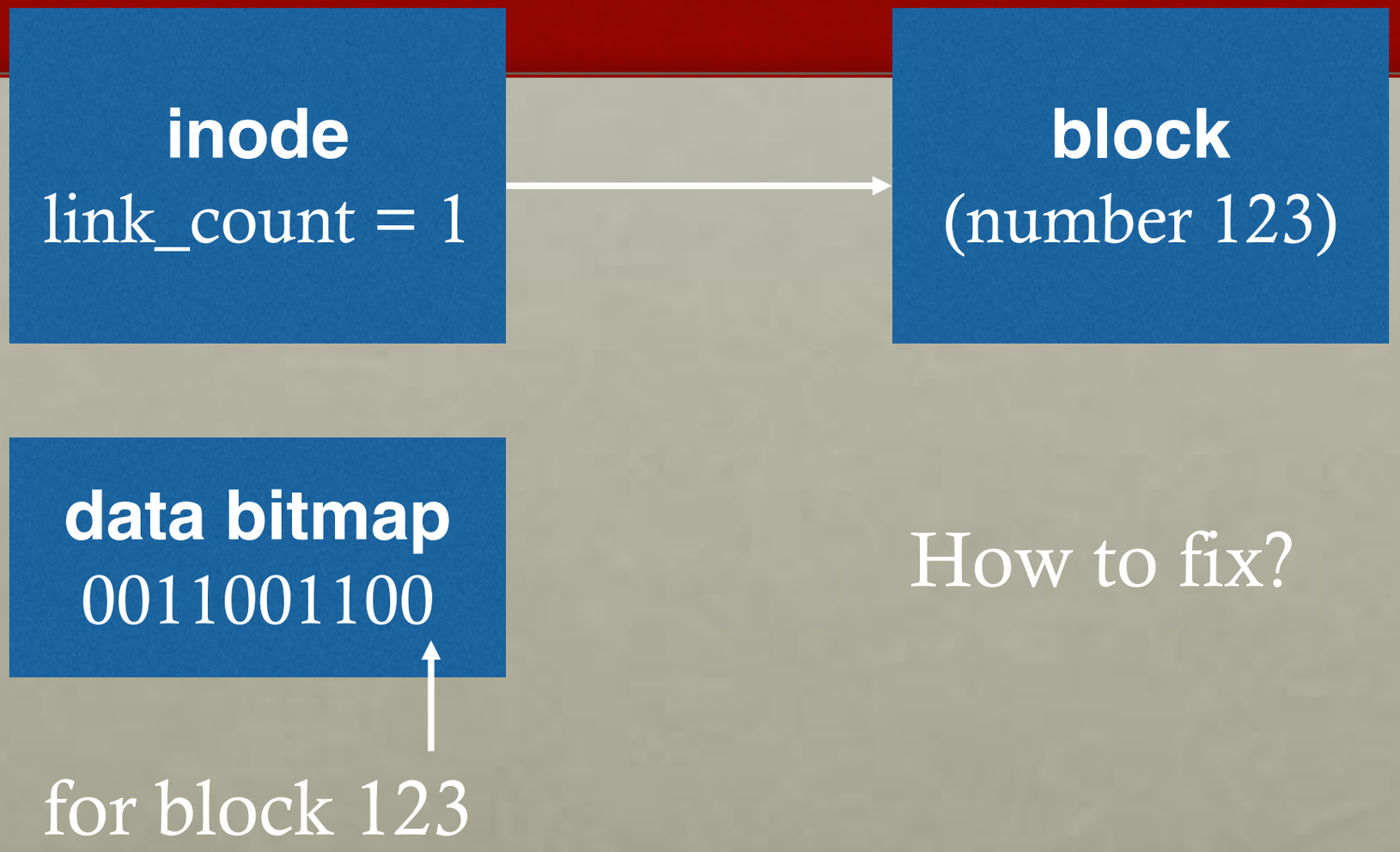
How to fix???

LINK COUNT (EXAMPLE 2)



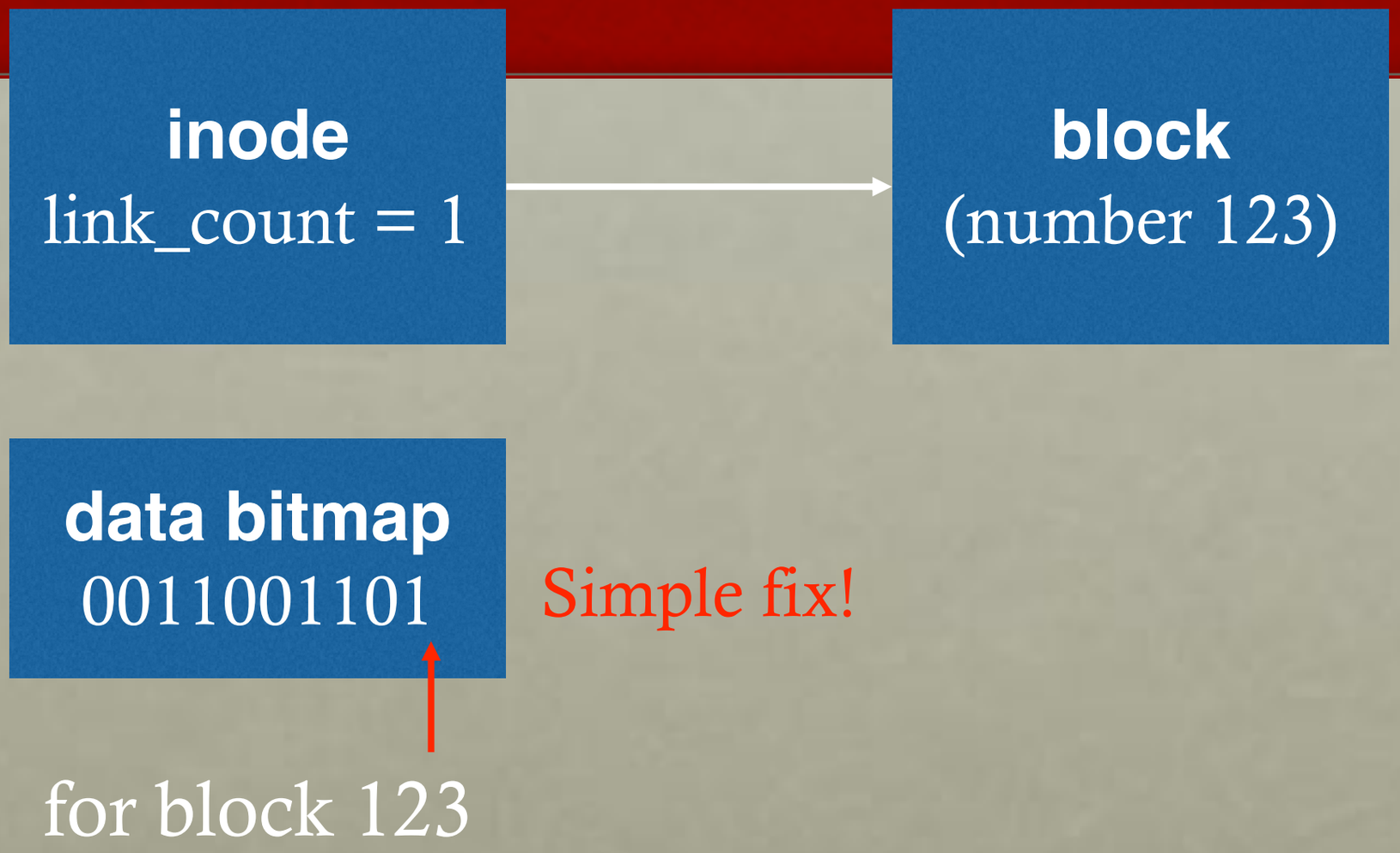

```
ls -l /
total 150
drwxr-xr-x  401 18432 Dec 31  1969 afs/
drwxr-xr-x.   2 4096  Nov  3 09:42 bin/
drwxr-xr-x.   5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.  13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.  10 12288 Nov  3 09:41 lib64/
drwx------.   2 16384 Aug  1 10:57 lost+found/
...
```

DATA BITMAP

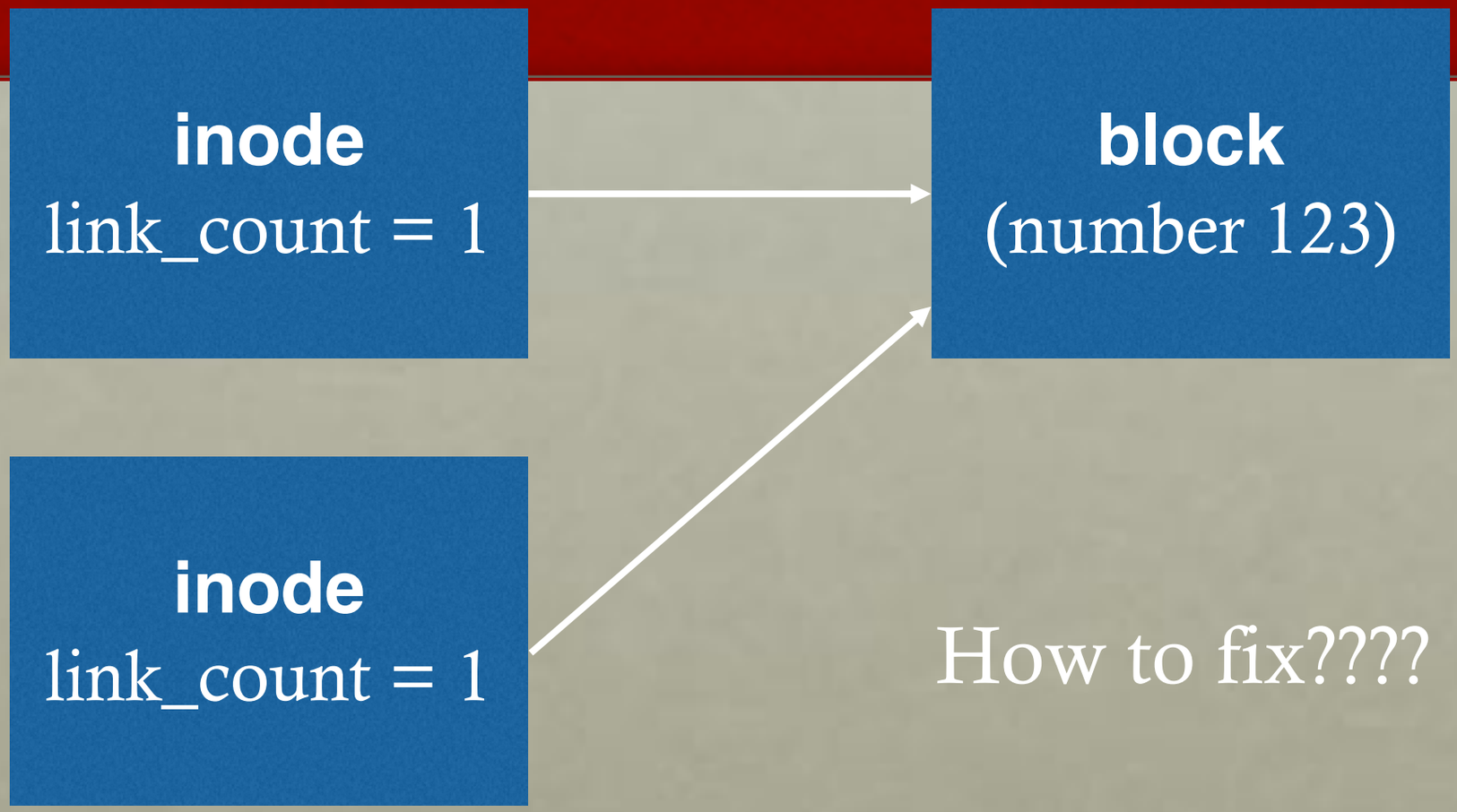


How to fix?

DATA BITMAP

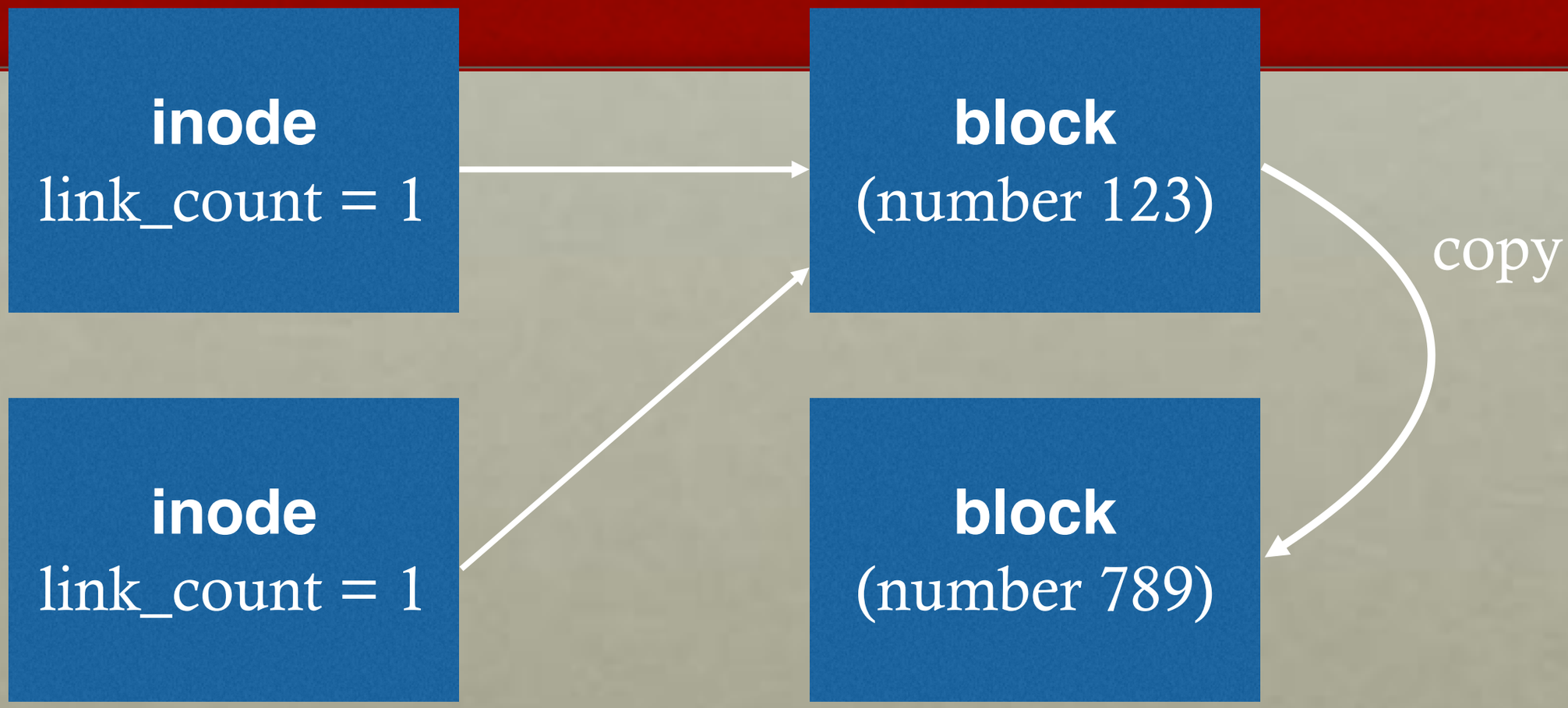

inode
link_count = 1
block
(number 123)
data bitmap
0011001101
Simple fix!
for block 123

DUPLICATE POINTERS

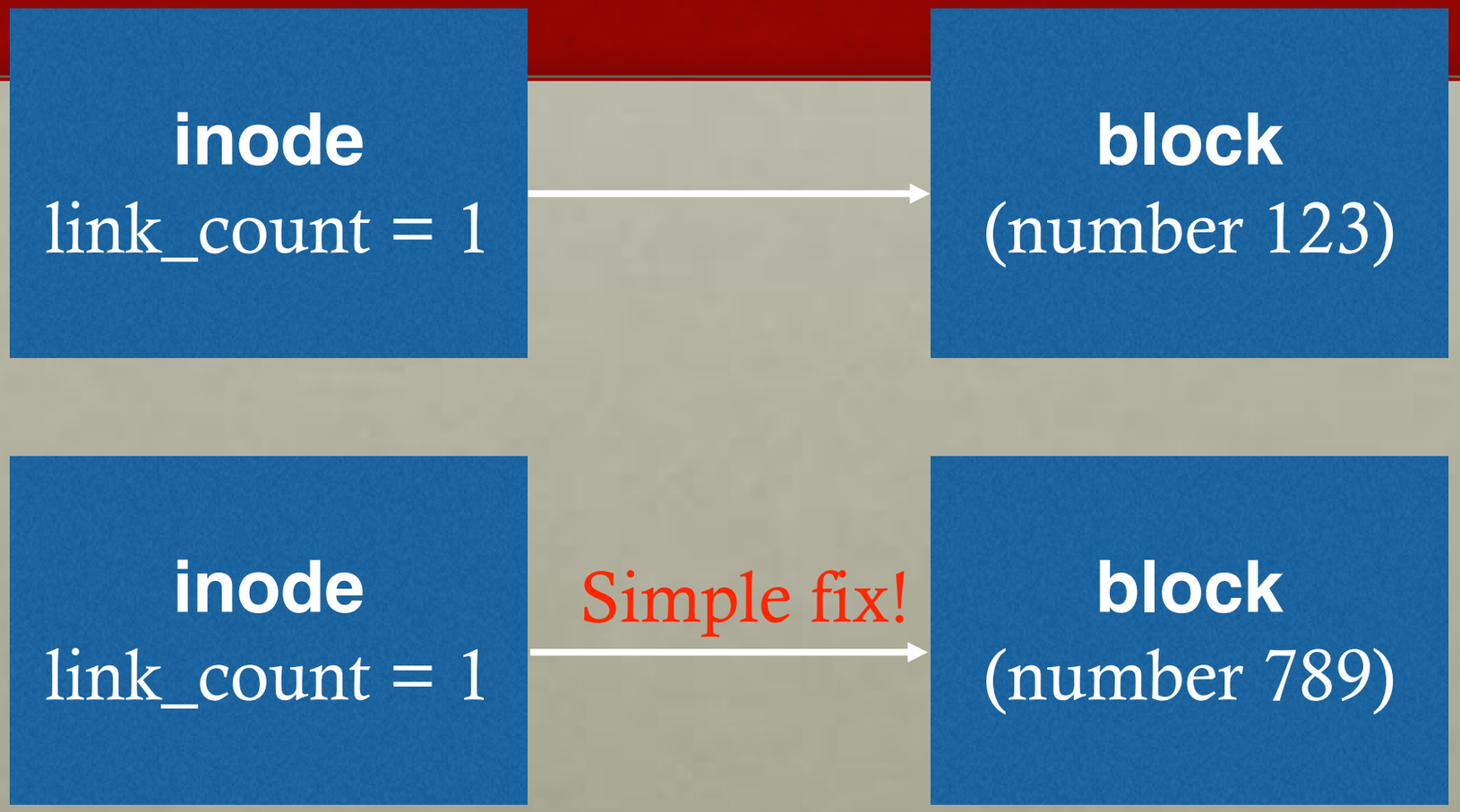


How to fix????

DUPLICATE POINTERS



DUPLICATE POINTERS



But is this correct?

BAD POINTER

inode
link_count = 1

9999

super block
tot-blocks=8000

How to fix???

BAD POINTER

inode
link_count = 1

Simple fix! (But is this correct?)

super block
tot-blocks=8000

PROBLEMS WITH FSCK

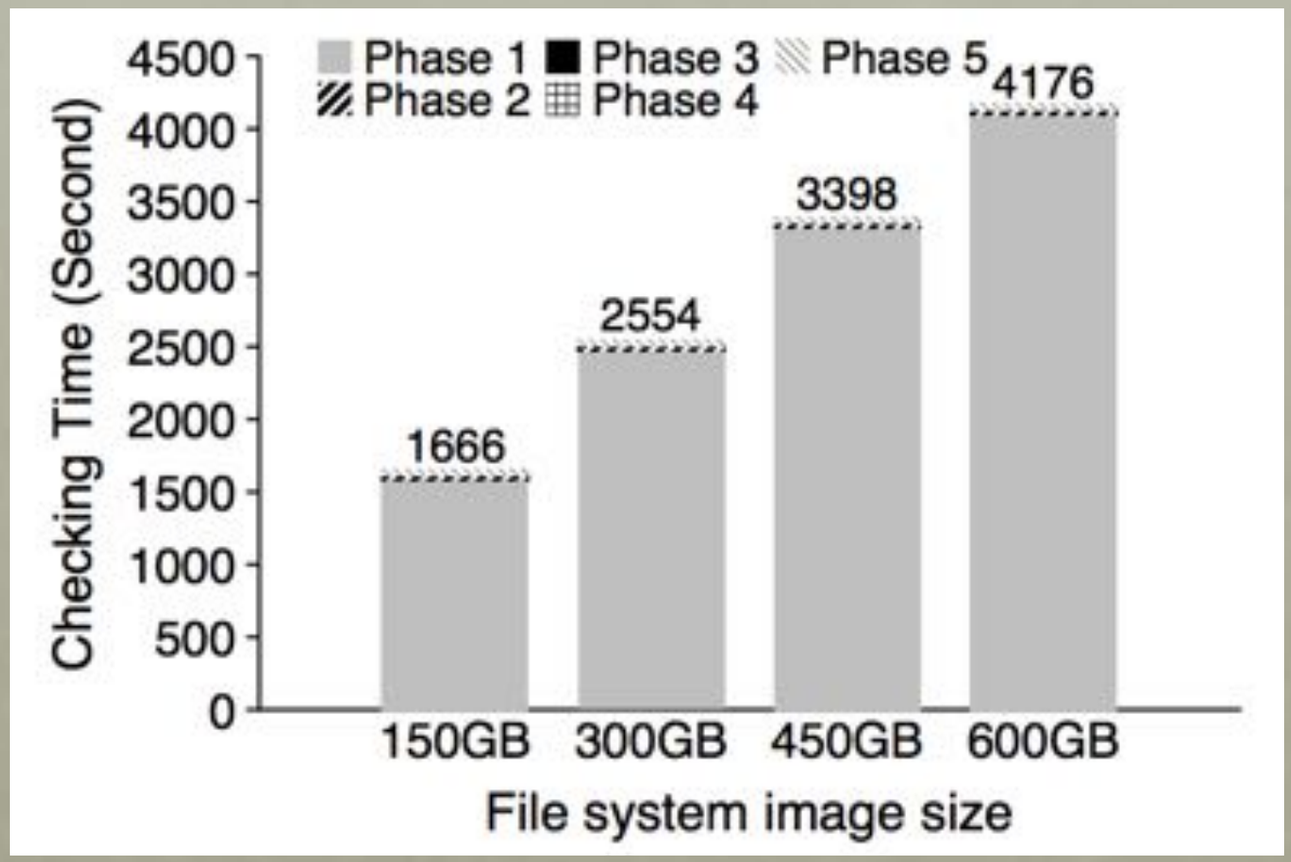
Problem 1:

- 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, EMC Corporation and University of Wisconsin—Madison; Chris Dragga,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

Consistency Solution #2: Journaling

Goals

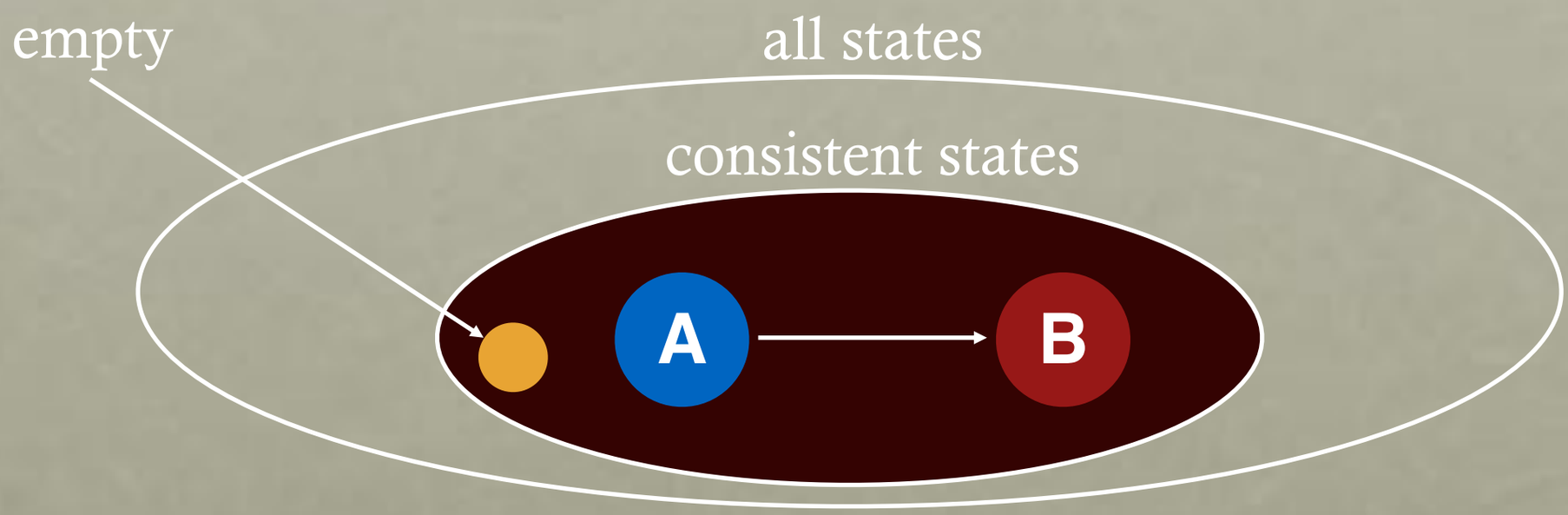
- Ok to do some **recovery work** after crash, but not to read entire disk
- Don't move file system to just any consistent state, get **correct** state

Strategy

- Atomicity
- Definintion 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 CORRECTNESS

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



fsck gives consistency
Atomicity gives A or B.

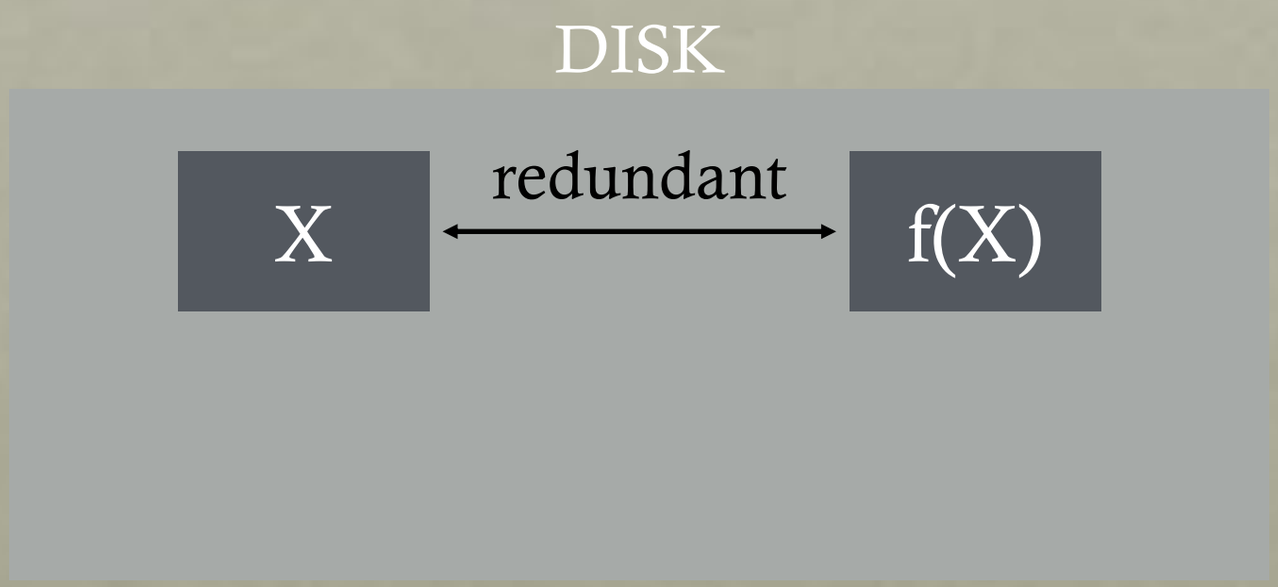
JOURNALING GENERAL STRATEGY

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

Ironically, adding redundancy to fix the problem caused by redundancy.

FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. Original:



FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. Original:

Good time to crash?

good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. Original:

Good time to crash?

bad time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

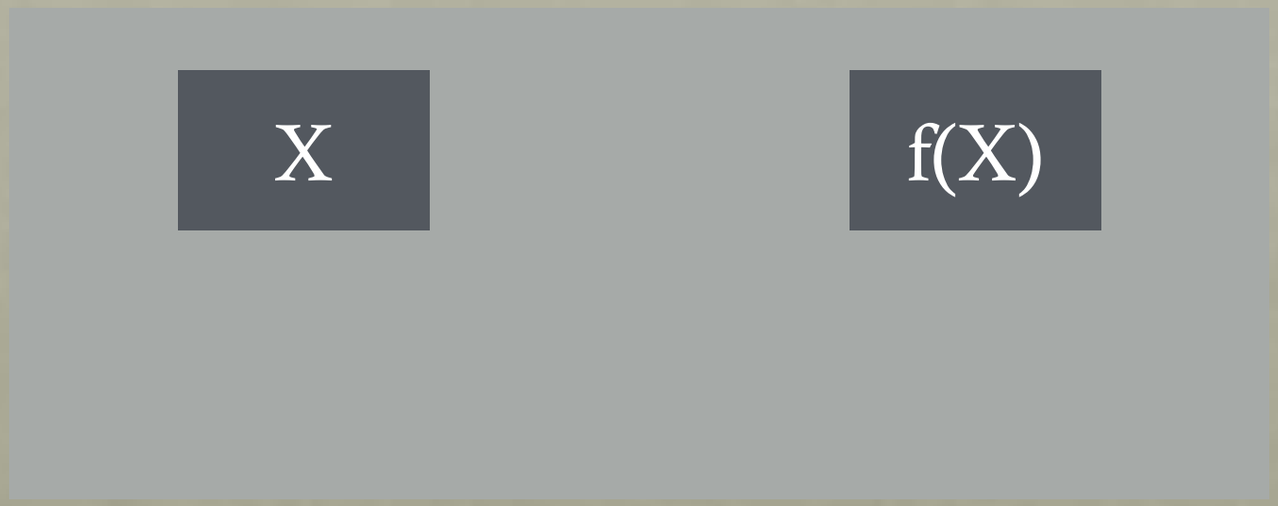
Want to replace X with Y. Original:

Good time to crash?

good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. **With journal:**

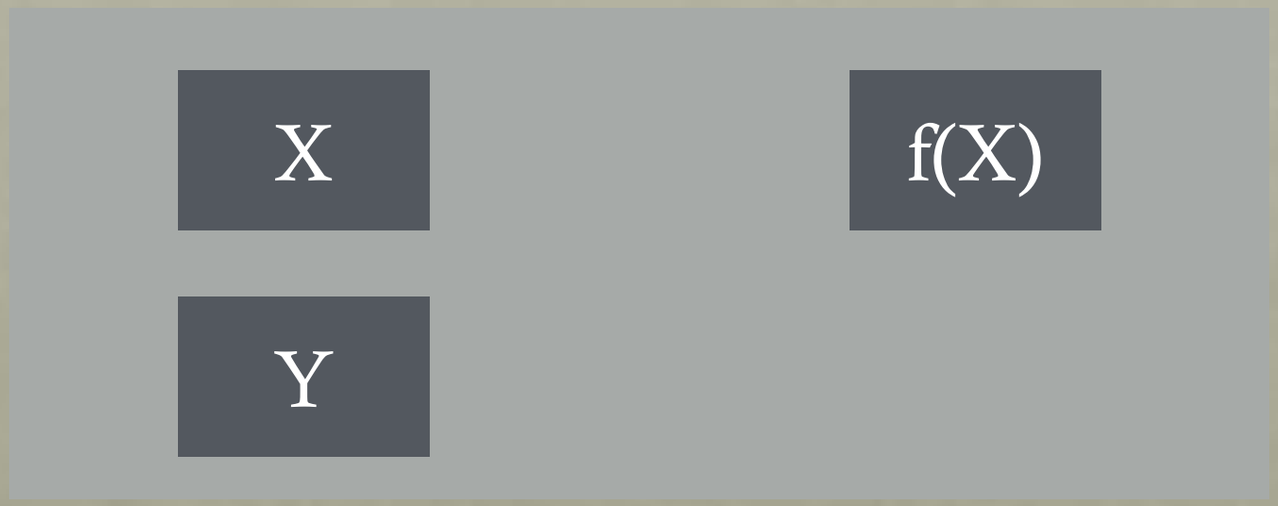


Good time to crash?

good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

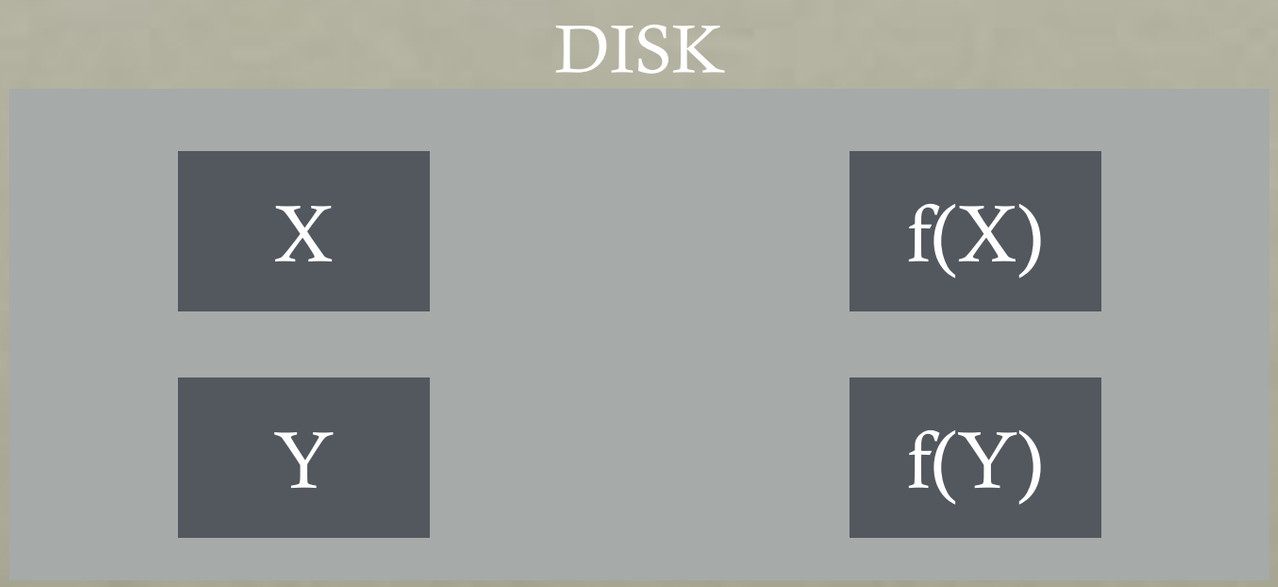
Want to replace X with Y. With journal:



good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

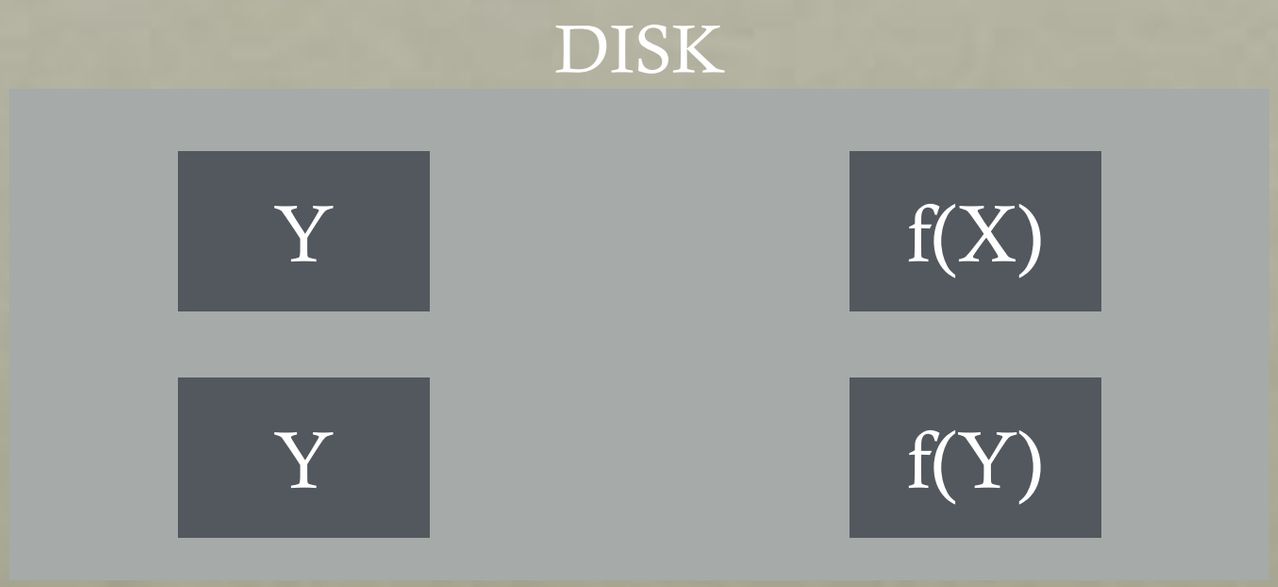
Want to replace X with Y. With journal:



good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

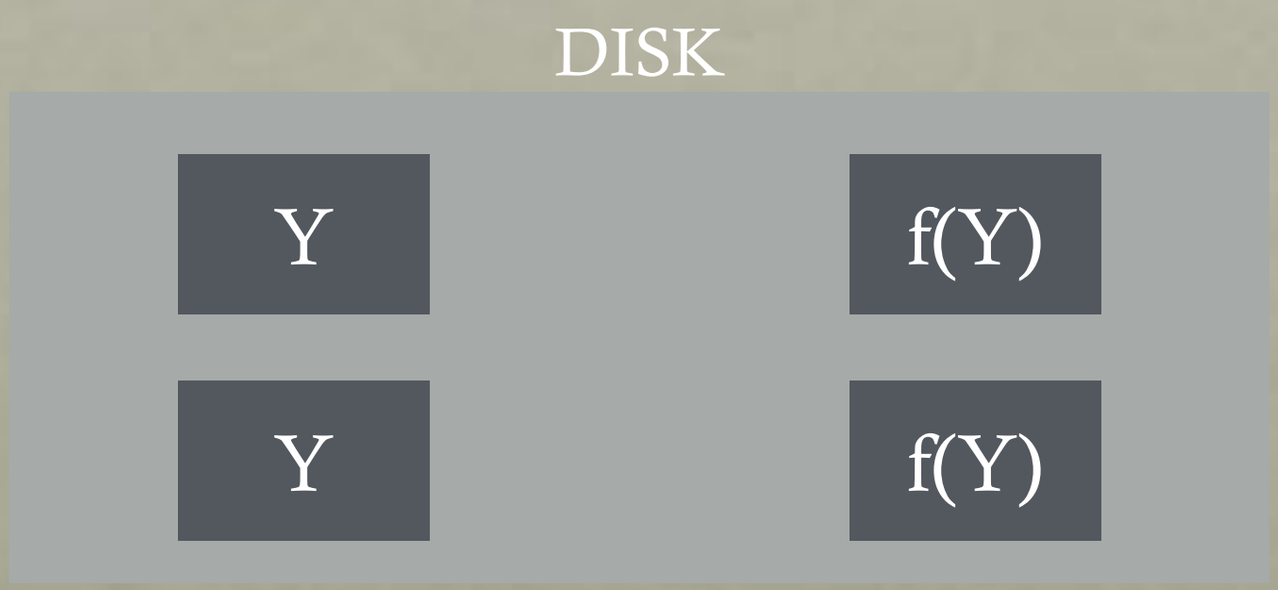
Want to replace X with Y. With journal:



good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. With journal:



good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. With journal:

good time to crash

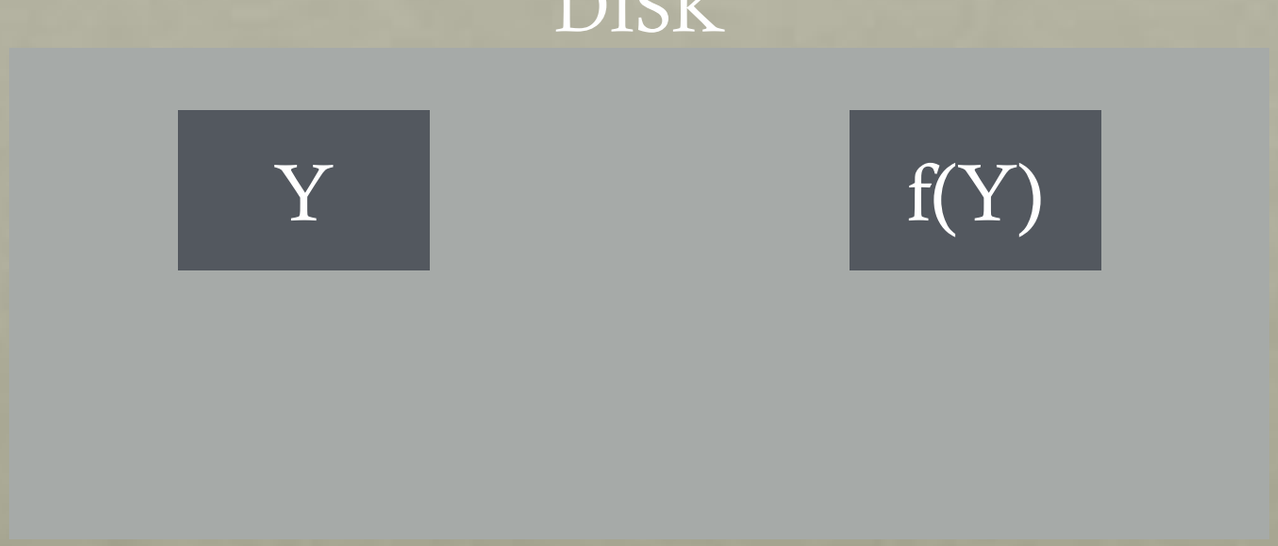
FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. With journal:

good time to crash

FIGHT REDUNDANCY WITH REDUNDANCY

Want to replace X with Y. With journal:



With journaling, it's always a good time to crash!

QUESTION FOR YOU…

Develop algorithm to atomically update two blocks:
Write 10 to block 0; write 5 to block 1

Assume these are only blocks in file system…

Time	Block 0	Block 1	extra	extra	extra	
1	12	3	0	0	0	
2	12	5	0	0	0	don't crash here!
3	10	5	0	0	0	

Wrong algorithm leads to inconsistent states
(non-atomic updates)

INITIAL SOLUTION: JOURNAL NEW DATA

Time	Block 0	Block 1	0’	1’	valid
1	12	3	0	0	0
2	12	3	10	0	0
3	12	3	10	5	0
4	12	3	10	5	1
5	10	3	10	5	1
6	10	5	10	5	1
7	10	5	10	5	0

Crash here?
→ Old data

Crash here?
→New data

Note: Understand behavior if crash after each write…

Usage Scenario: Block 0 stores Alice’s bank account; Block 1 stores Bob’s bank account; transfer $2 from Alice to Bob

```
void update_accounts(int cash1, int cash2) {
        write(cash1 to block 2) // Alice backup
        write(cash2 to block 3) // Bob backup
        write(1 to block 4)     // backup is safe
        write(cash1 to block 0) // Alice
        write(cash2 to block 1) // Bob
        write(0 to block 4)     // discard backup
}
```

```
void recovery() {
        if(read(block 4) == 1) {
                write(read(block 2) to block 0) // restore Alice
                write(read(block 3) to block 1) // restore Bob
                write(0 to block 4)             // discard backup
        }
}
```

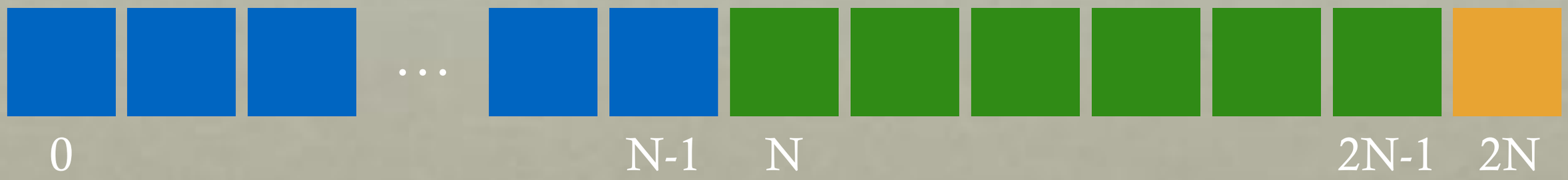
TERMINOLOGY

Extra blocks are called a “journal”

The writes to the jounral are a “journal transaction”

The last valid bit written is a “journal commit block”

PROBLEM WITH INITIAL APPROACH: JOURNAL SIZE



Disadvantages?

- slightly < half of disk space is usable
- transactions copy all the data (1/2 bandwidth!)

FIX #1: SMALL JOURNALS

Still need to first write all new data elsewhere before overwriting new data

Goal:

- Reuse small area as backup for any block

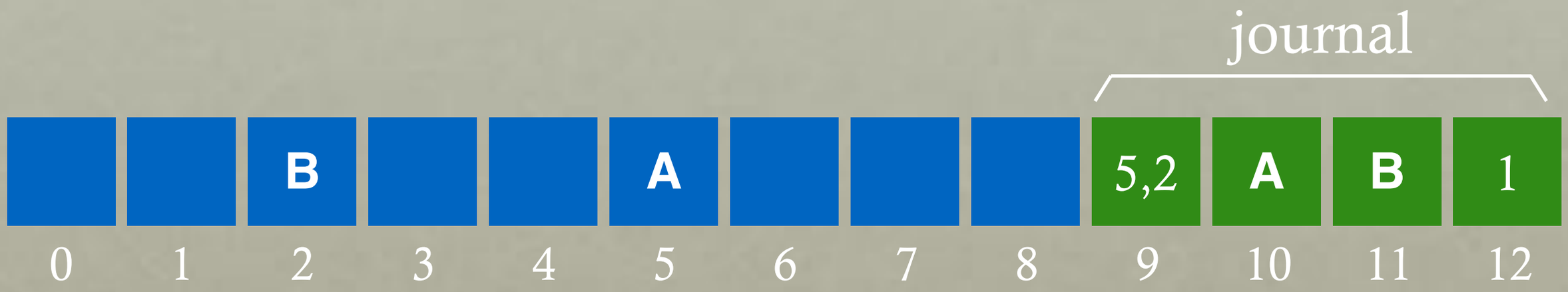
How?

- Store block numbers in a transaction header

NEW LAYOUT

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

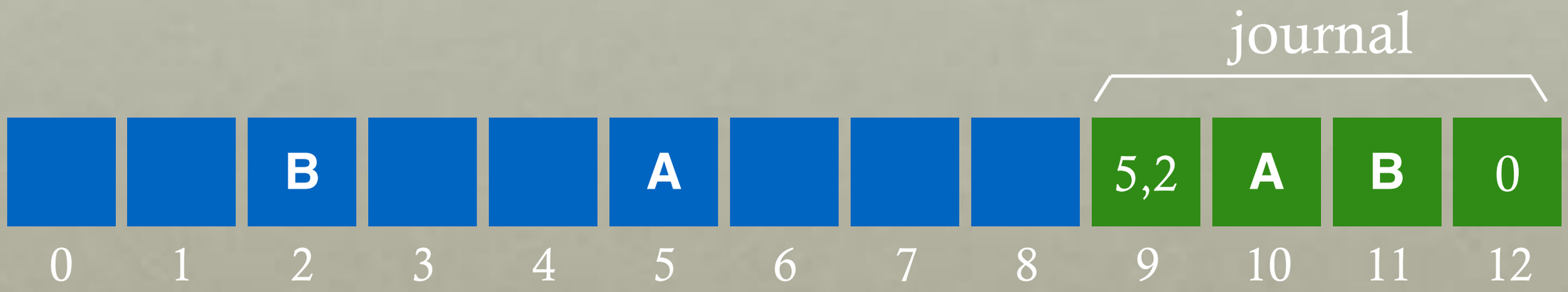
NEW LAYOUT



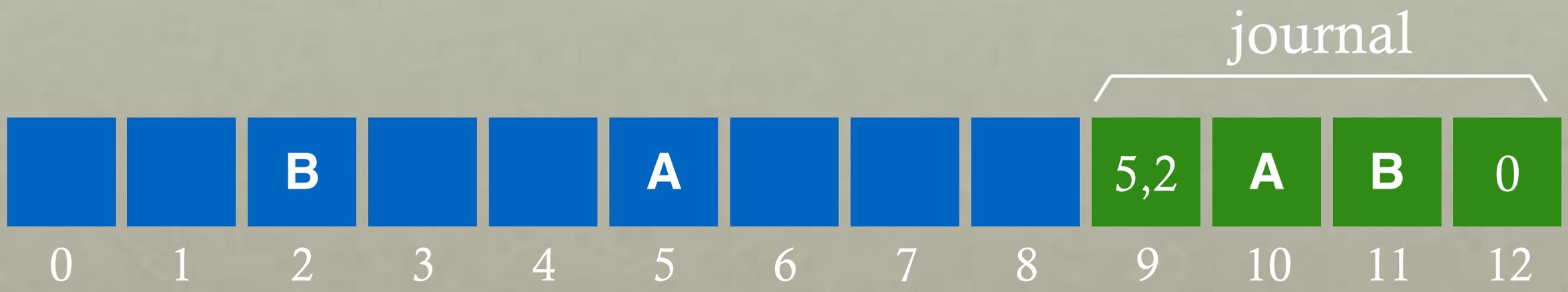
transaction: write A to block 5; write B to block 2

Checkpoint: Writing new data to in-place locations

NEW LAYOUT

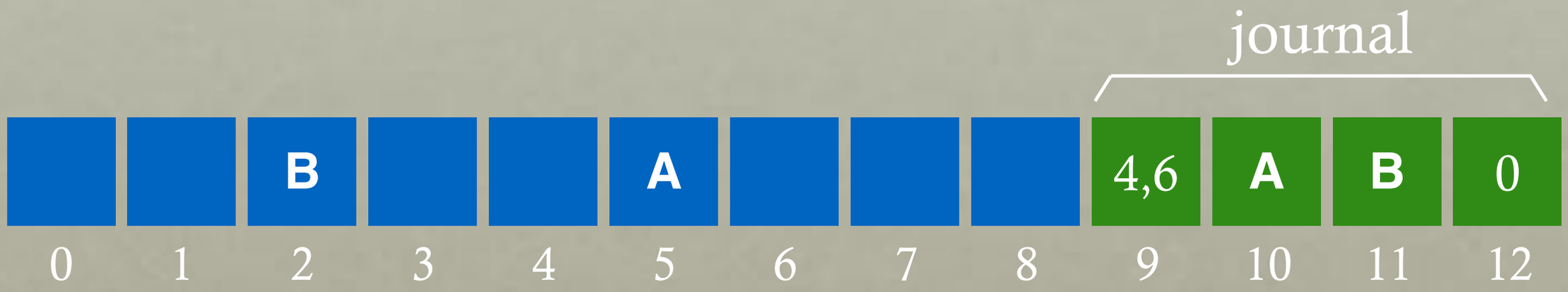


NEW LAYOUT



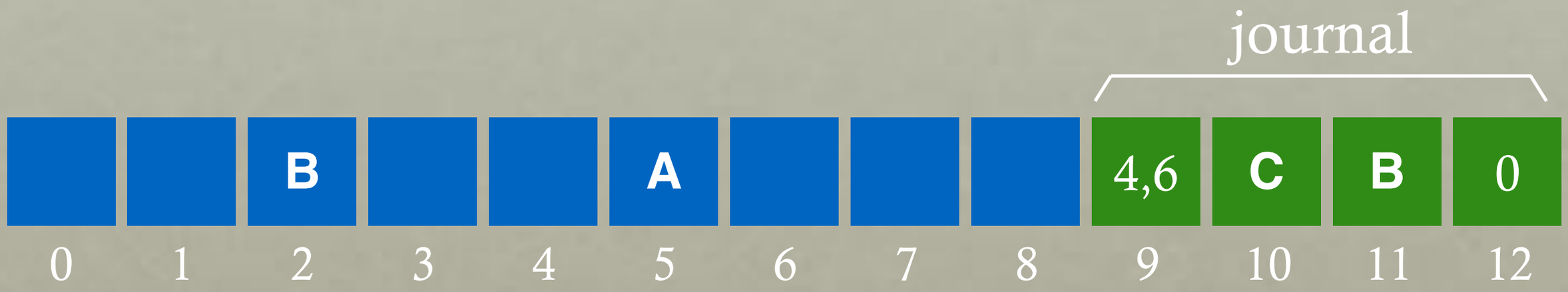
transaction: write C to block 4; write T to block 6

NEW LAYOUT



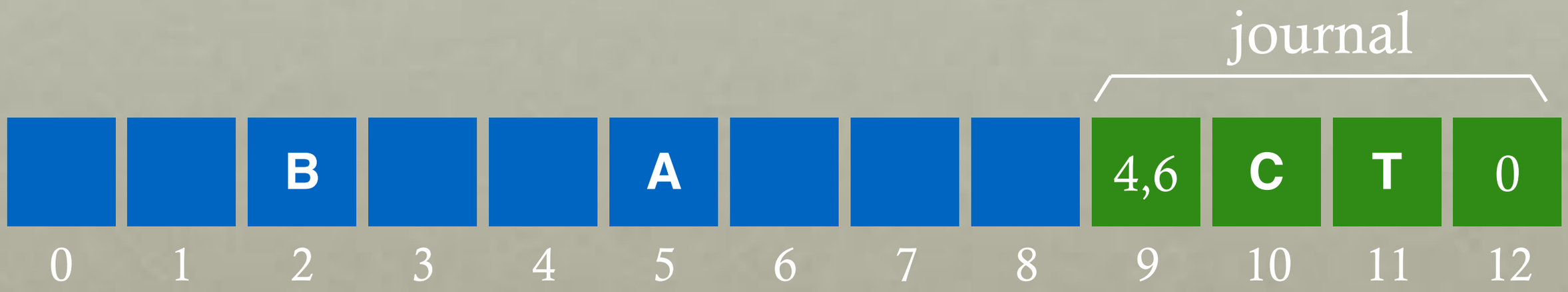
transaction: write C to block 4; write T to block 6

NEW LAYOUT



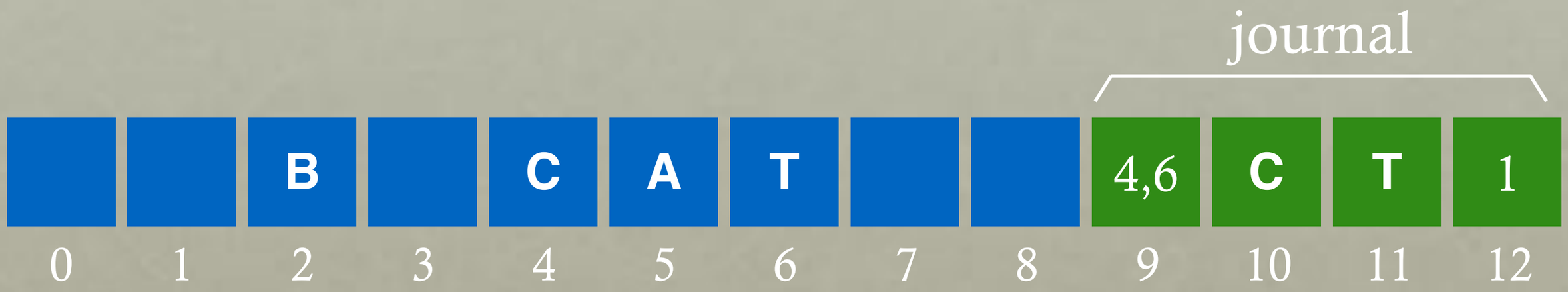
transaction: write C to block 4; write T to block 6

NEW LAYOUT



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

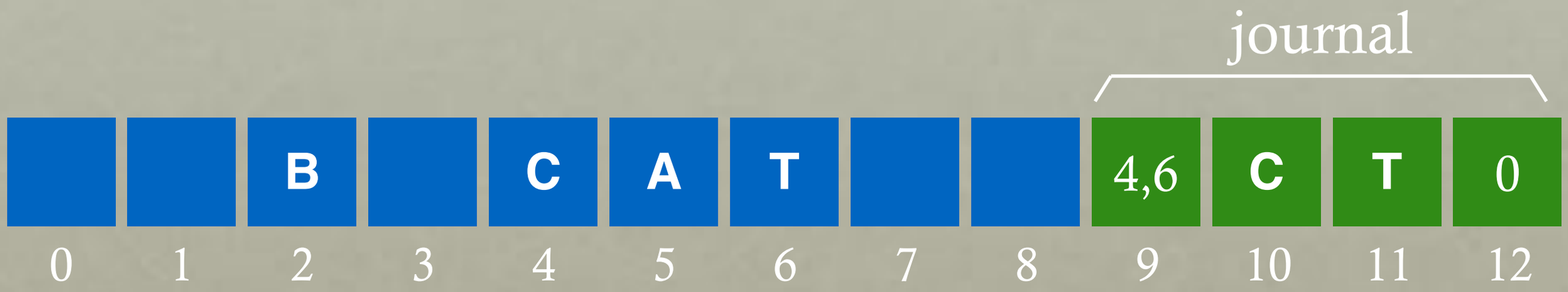
NEW LAYOUT



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

Checkpoint: Writing new data to in-place locations

NEW LAYOUT

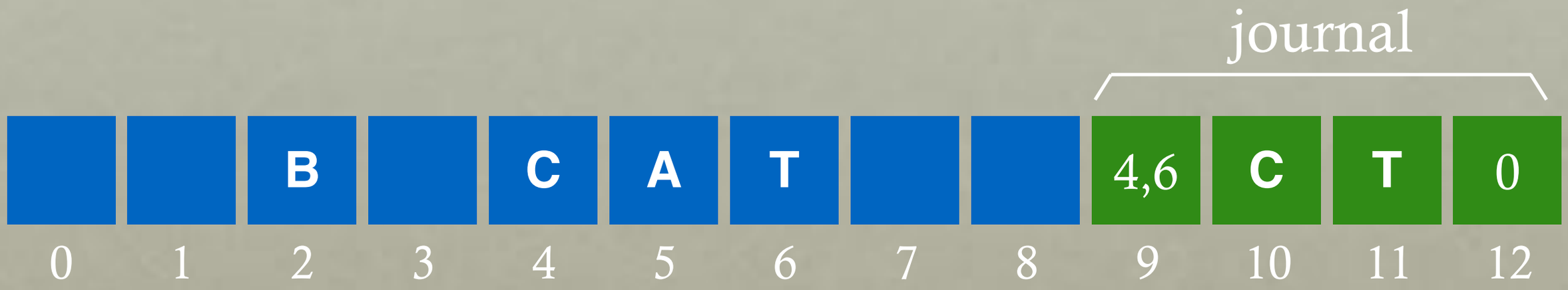


transaction: write C to block 4; write T to block 6

OPTIMIZATIONS

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal

CORRECTNESS DEPENDS ON ORDERING



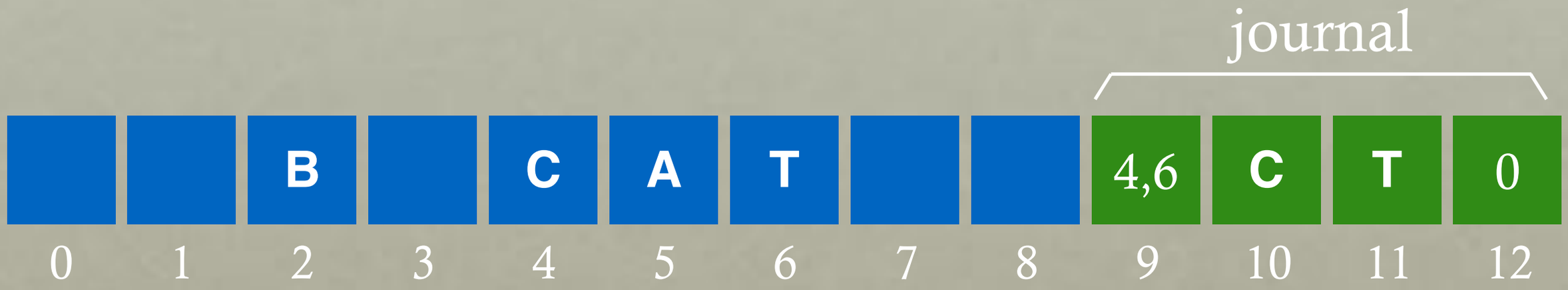
transaction: write C to block 4; write T to block 6

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

Enforcing total ordering is inefficient. Why?
Random writes

Instead: Use barriers w/ disk cache flush at key points (when??)

ORDERING



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

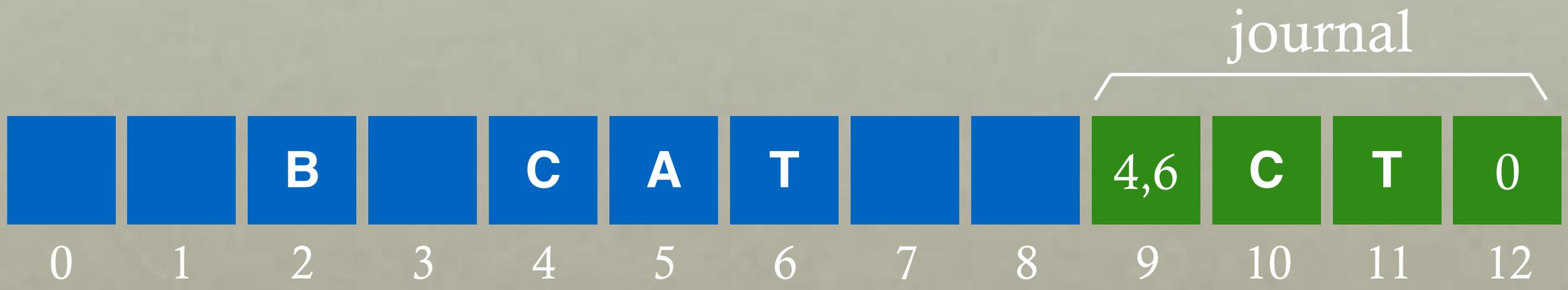
write order: 9,10,11 | 12 | 4,6 | 12

Use barriers at key points in time:
1) Before journal commit, ensure journal transaction entries complete
2) Before checkpoint, ensure journal commit complete
3) Before free journal, ensure in-place updates complete

OPTIMIZATIONS

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal

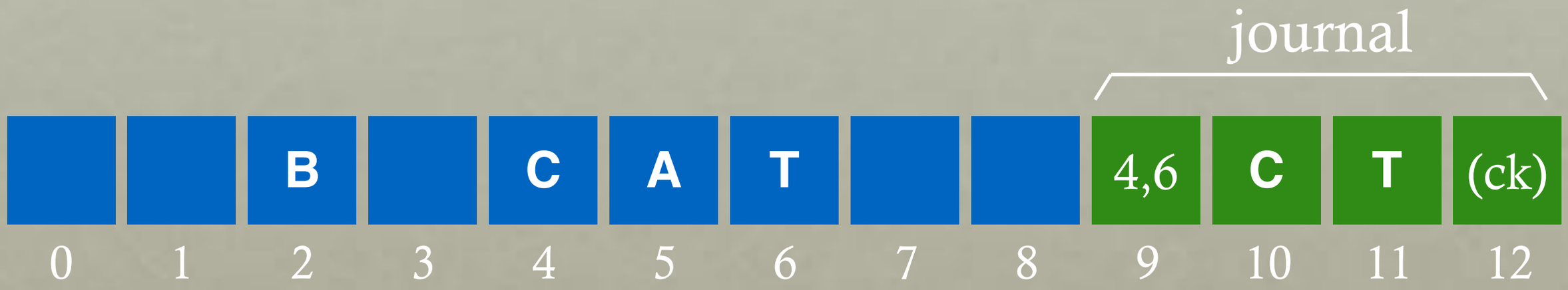
CHECKSUM OPTIMIZATION



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

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

CHECKSUM OPTIMIZATION



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

In last transaction block, store checksum of rest of transaction

12 = Cksum(9, 10, 11)

During recovery:

If checksum does not match transaction, treat as not valid

OPTIMIZATIONS

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal

WRITE BUFFERING OPTIMIZATION

Note: after journal write, there is no rush to checkpoint

- If system crashes, still have persistent copy of written data!

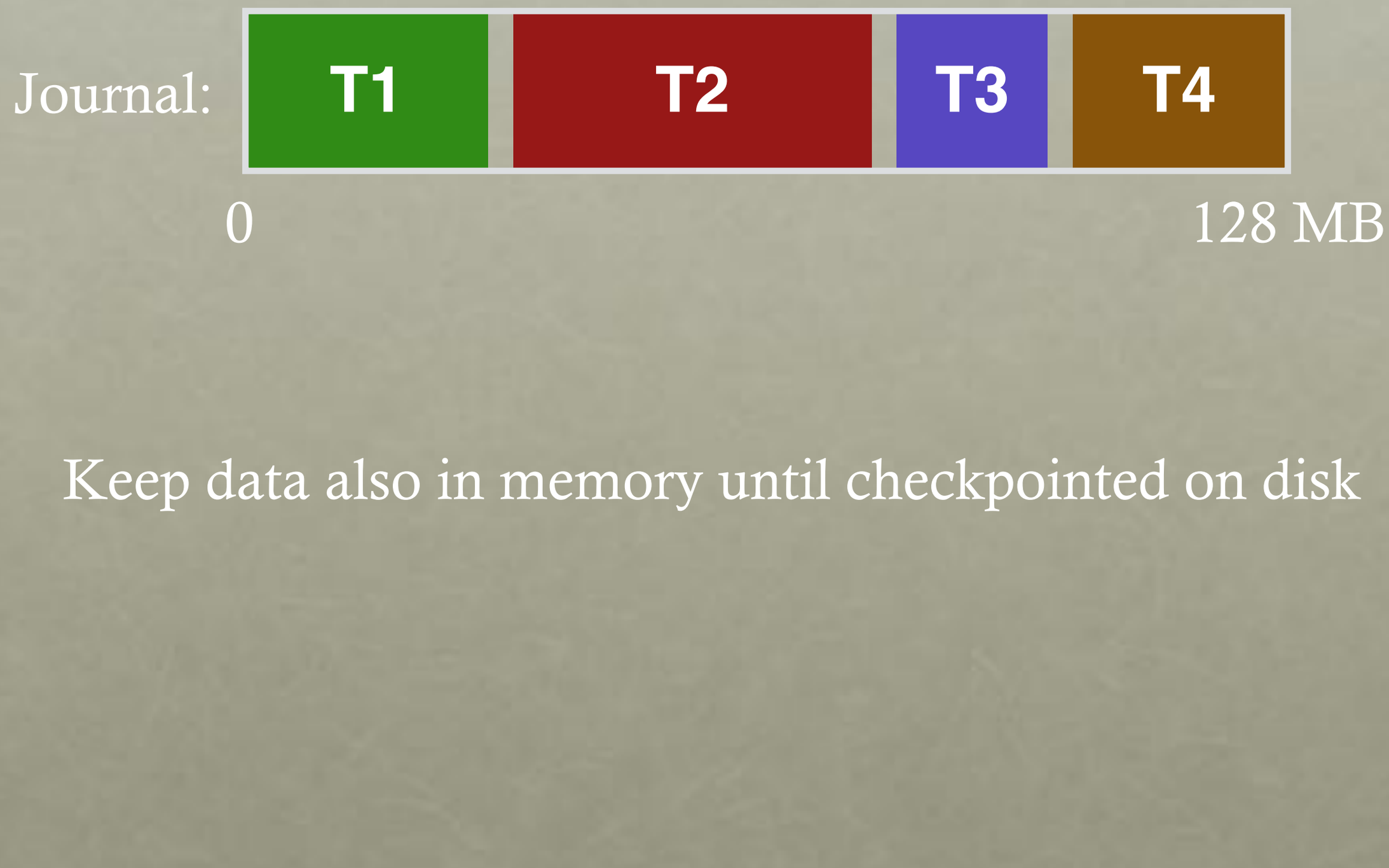
Journaling is sequential, checkpointing is random

Solution? Delay checkpointing for some time

Difficulty: need to reuse journal space

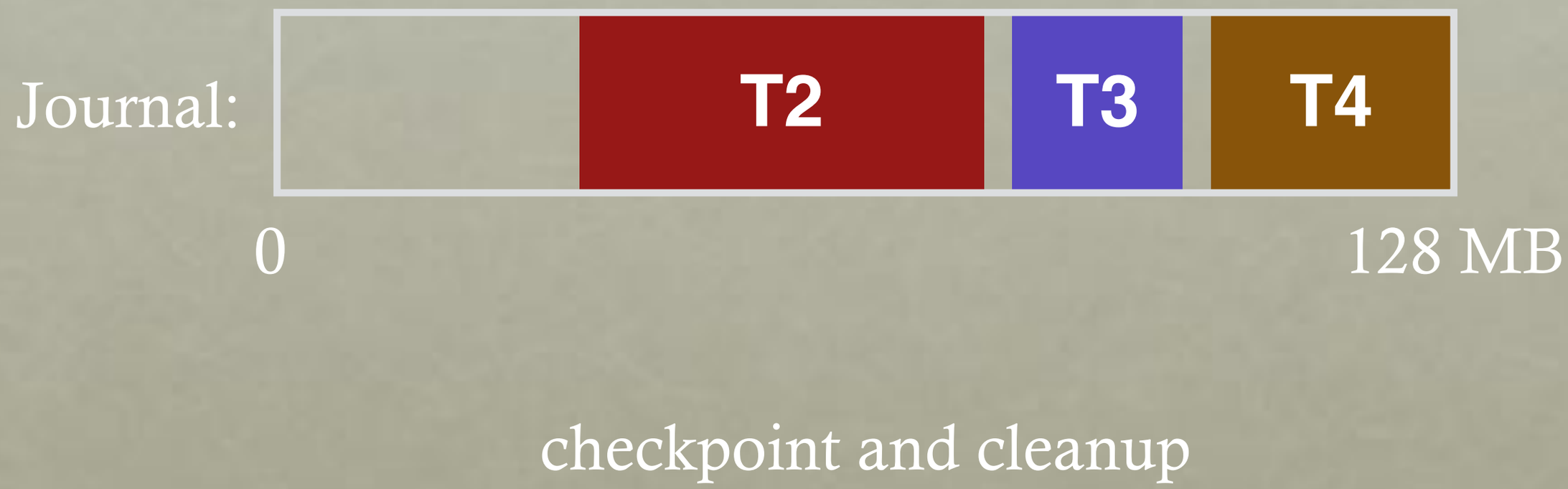
Solution: keep many transactions for un-checkpointed data

CIRCULAR BUFFER



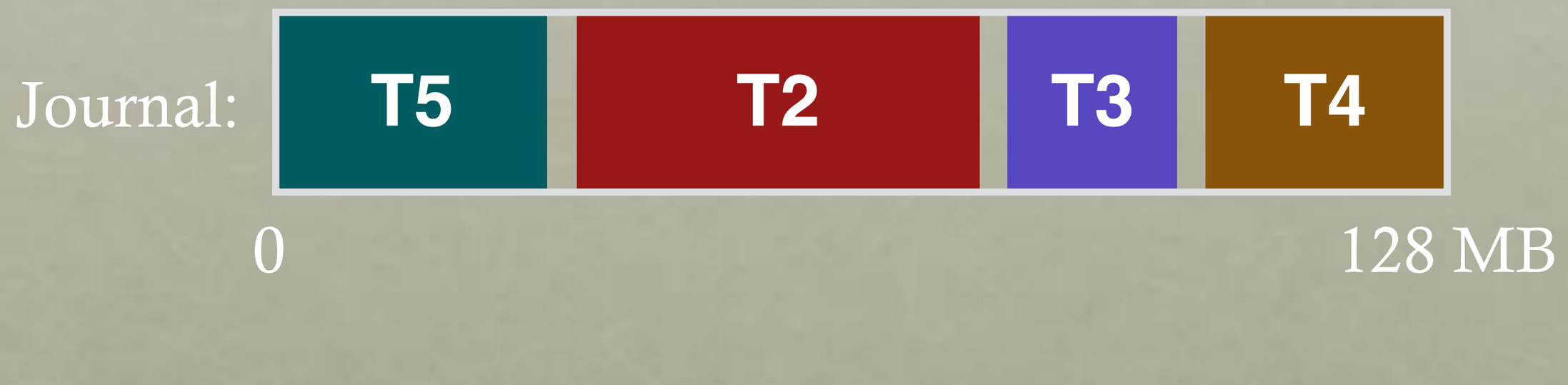
Keep data also in memory until checkpointed on disk

CIRCULAR BUFFER



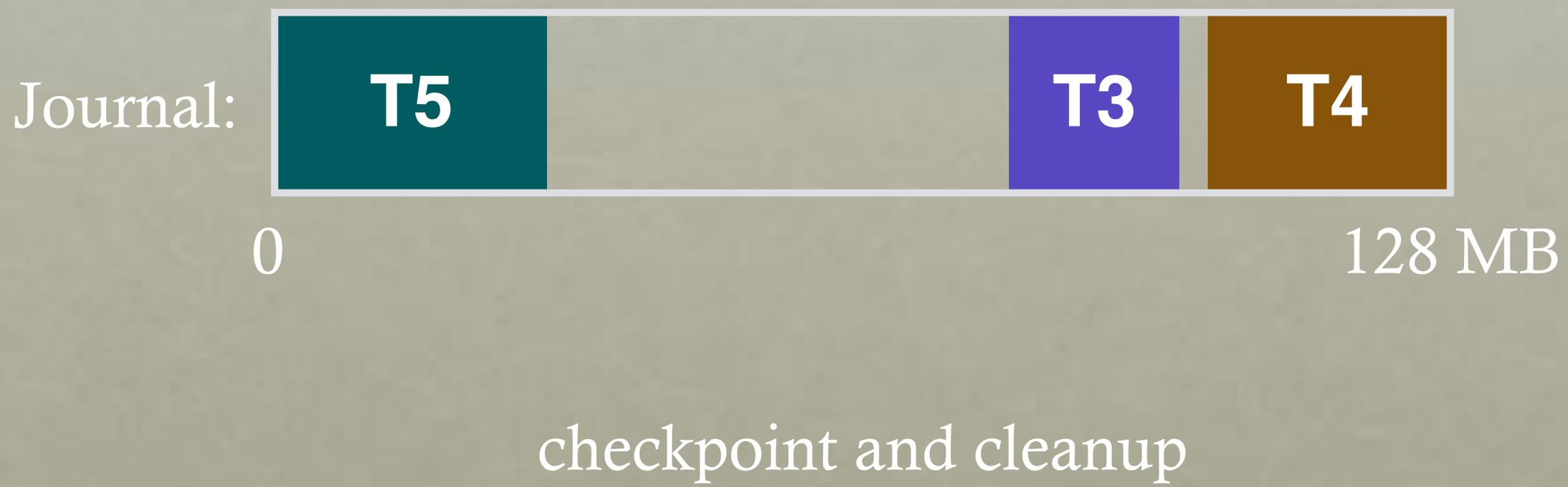
checkpoint and cleanup

CIRCULAR BUFFER



transaction!

CIRCULAR BUFFER



checkpoint and cleanup

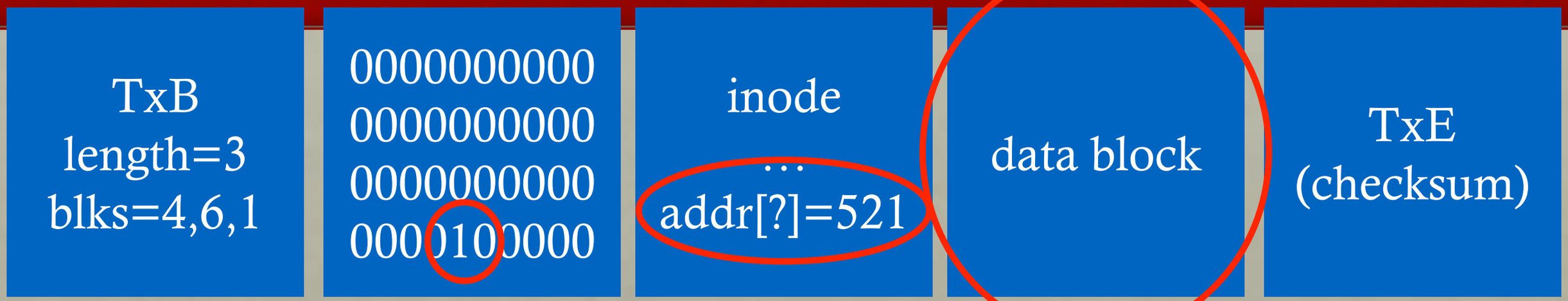
OPTIMIZATIONS

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal

PHYSICAL JOURNAL

TxB length=3 blks=4,6,1	0000000000 0000000000 0000000000 0000100000	inode … addr[?]=521	data block	TxE (checksum)

PHYSICAL JOURNAL



Actual changed data is much smaller!

LOGICAL JOURNAL

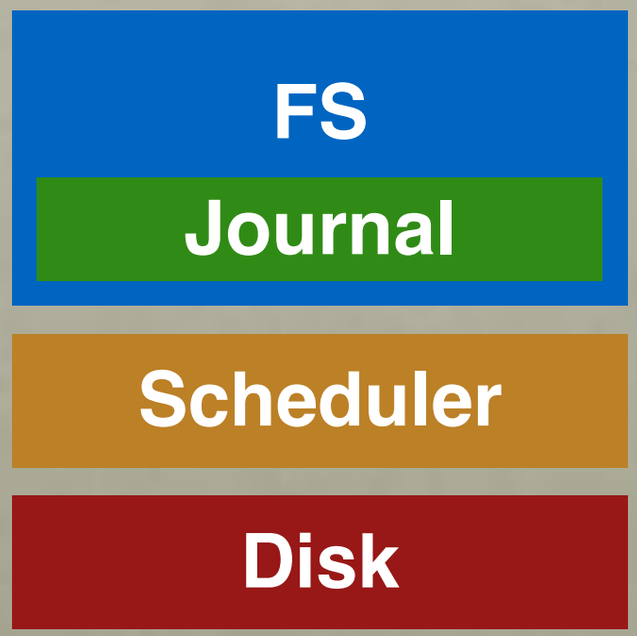
Logical journals record changes to bytes, not contents of new blocks

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

OPTIMIZATIONS

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal

FILE SYSTEM INTEGRATION



HOW TO AVOID WRITING ALL DISK BLOCKS TWICE?

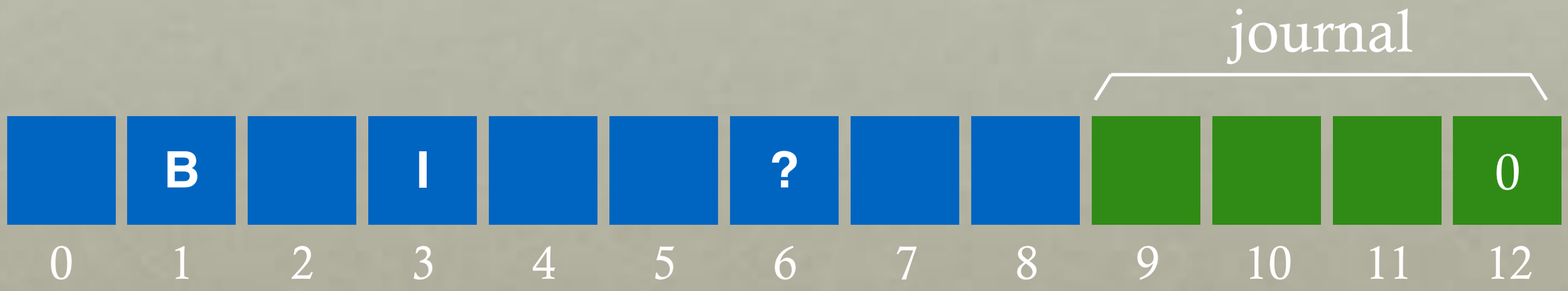
Observation: some blocks (e.g., user data) are less important

Strategy: journal all metadata, including:

superblock, bitmaps, inodes, indirects, directories

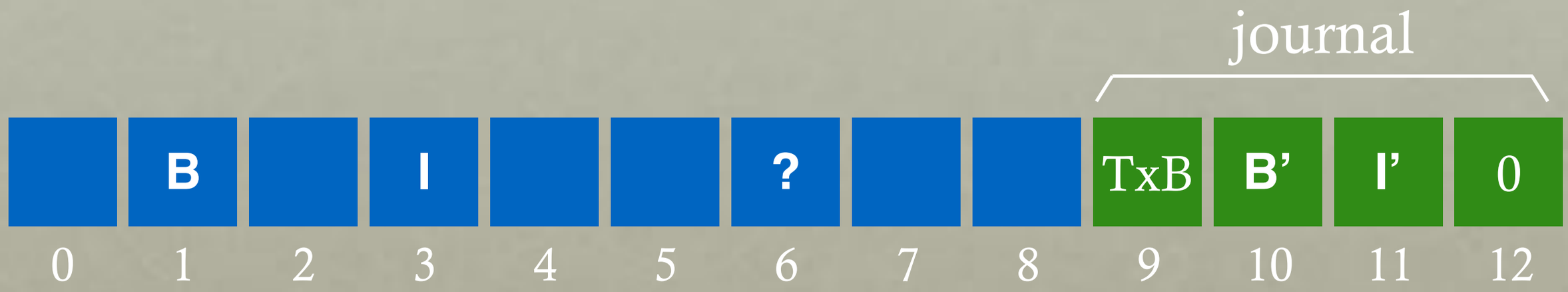
For regular data, write it back whenever convenient.
Of course, files may contain garbage.

WRITEBACK JOURNAL



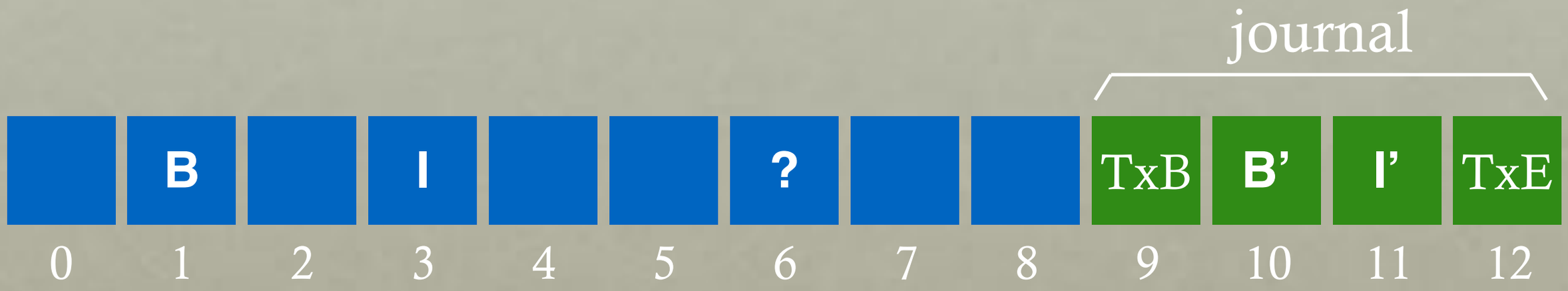
transaction: append to inode I

WRITEBACK JOURNAL



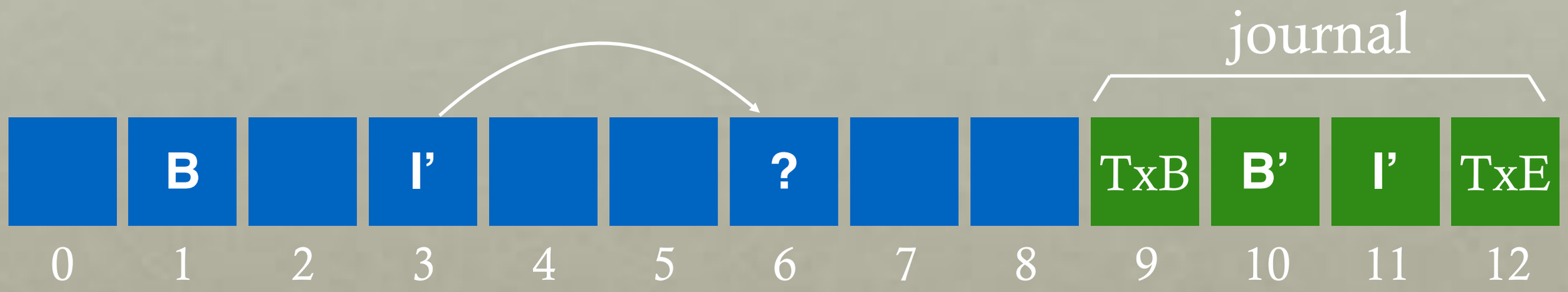
transaction: append to inode I

WRITEBACK JOURNAL



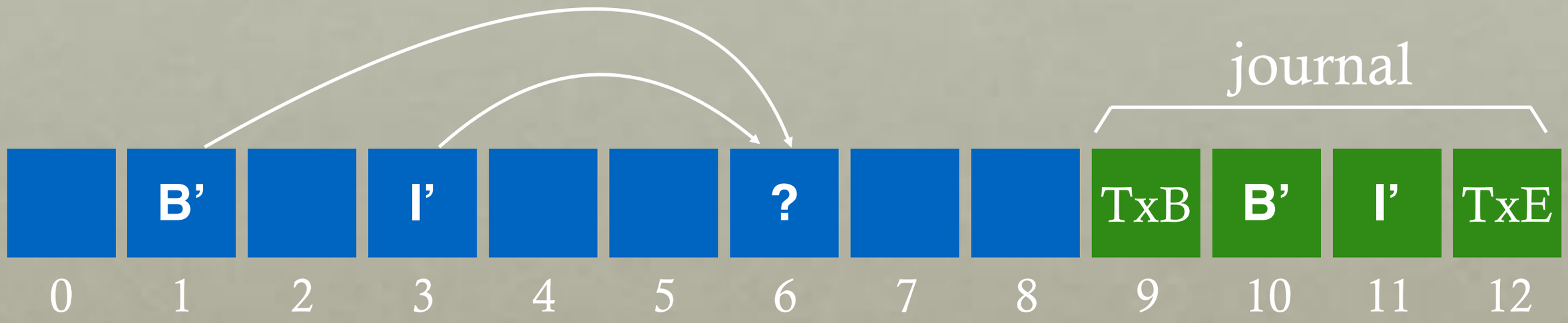
transaction: append to inode I

WRITEBACK JOURNAL



transaction: append to inode I

WRITEBACK JOURNAL



transaction: append to inode I

what if we crash now? Solutions?

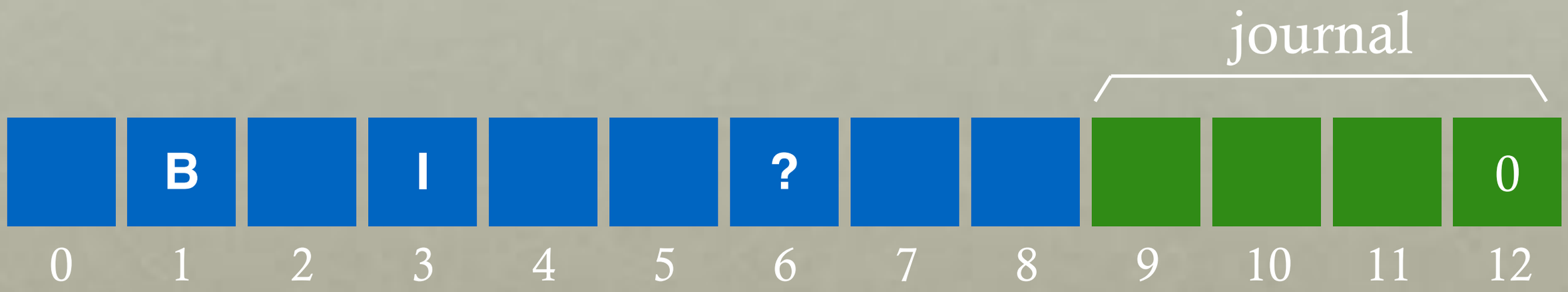
ORDERED JOURNALING

Still only journal metadata

But write data **before** the transaction

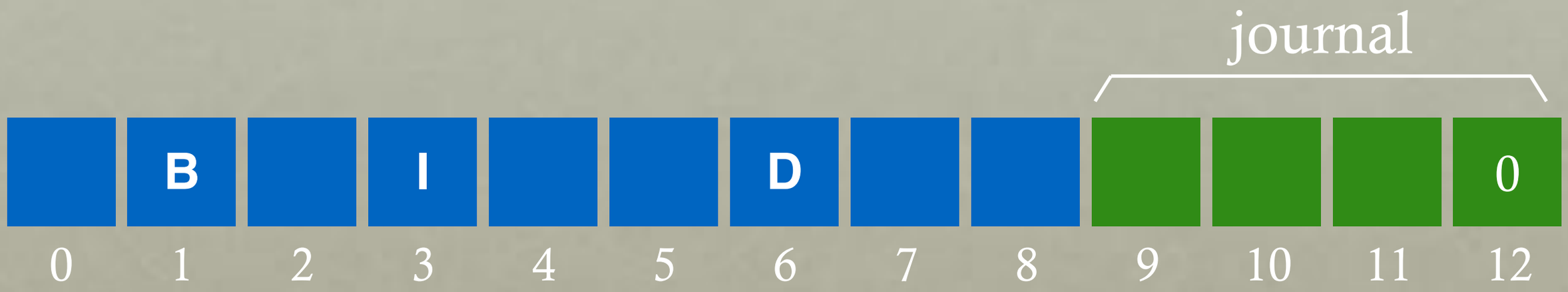
No leaks of sensitive data!

ORDERED JOURNAL



transaction: append to inode I

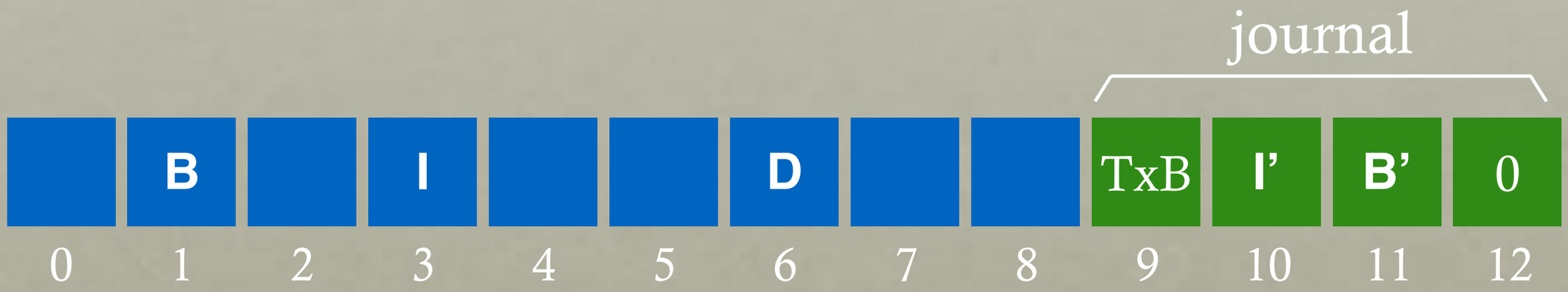
ORDERED JOURNAL



transaction: append to inode I

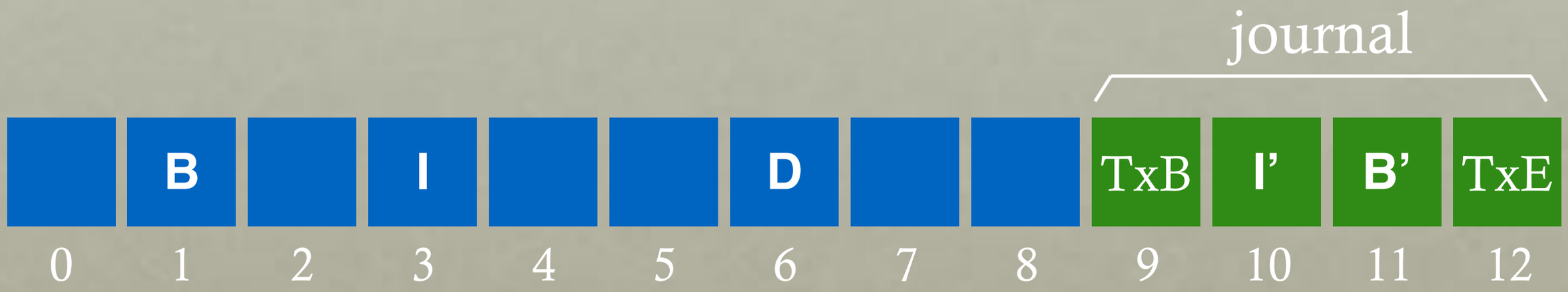
What happens if crash now?
B indicates D currently free, I does not point to D;
Lose D, but that might be acceptable

ORDERED JOURNAL



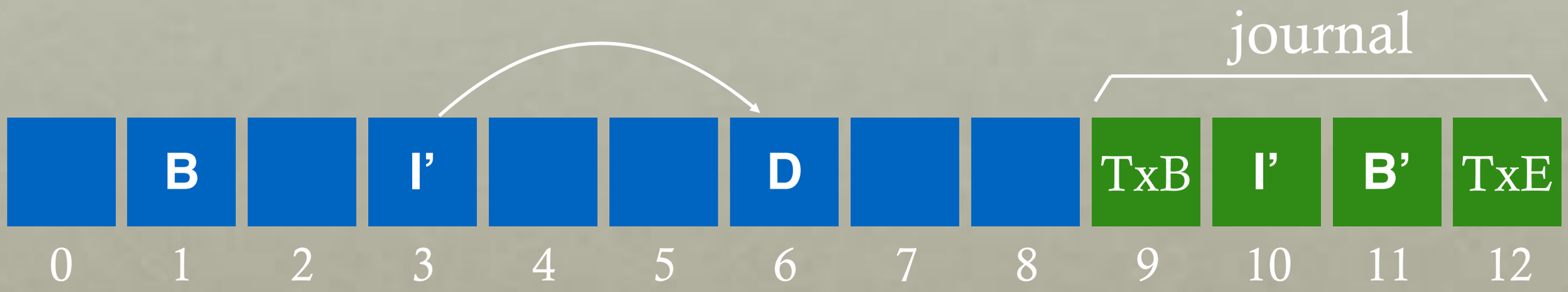
transaction: append to inode I

ORDERED JOURNAL



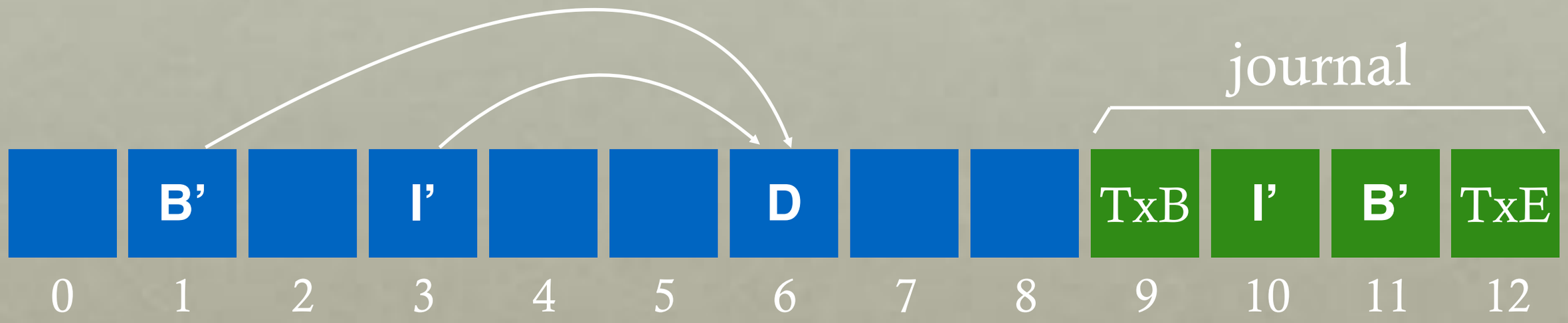
transaction: append to inode I

ORDERED JOURNAL



transaction: append to inode I

ORDERED JOURNAL



transaction: append to inode I

CONCLUSION

Most modern file systems use journals

- ordered-mode for meta-data is popular

FSCK is still useful for weird cases

- bit flips

- FS bugs

Some file systems don't use journals, but still (usually) write new data before deleting old (copy-on-write file systems)