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 examples of redundancy in FFS (or files system in general)**

- **Inode file size AND inode/indirect pointers**
  - file size can be calculated with pointers
- **Data bitmap AND inode pointers**
  - through inode pointers can know which data block is free or not
- **Data bitmap AND group descriptor**
  - (tracks free inodes and data blocks for fast location)
- **Dir entries AND inode link count**
  - traversing the entire dir entries can know the link count
- **…**

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
  - Inode: 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 , no file will access the data block
  - b) data: nothing bad, but fail to update
  - c) inode: point to garbage, another file may use
  - d) bitmap and data: lost block, no file will access the data block
  - e) bitmap and inode: point to garbage, data is not updated
  - f) data and inode: another file may use, because bitmap is not set

How can file system fix Inconsistencies?

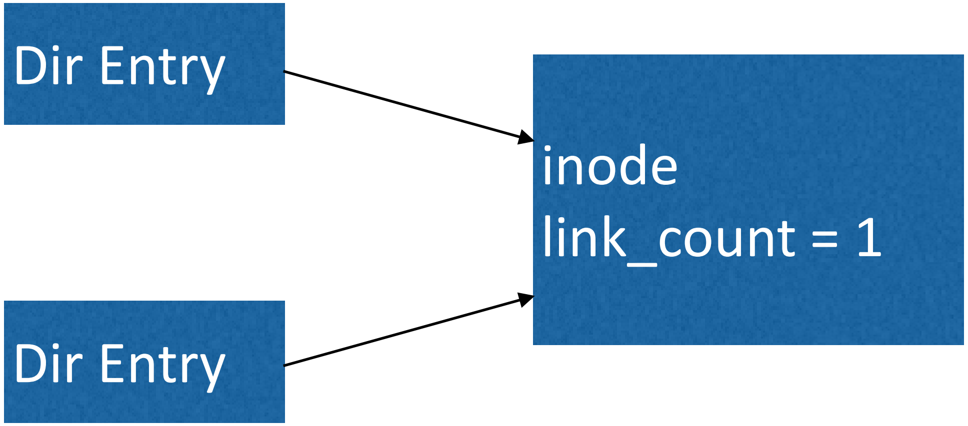
- **Solution #1:**
  - FSCK = file system checker
- **Strategy:**
  - After crash, scan whole disk for contradictions and “fix” if needed
  - Keep file system off-line until FSCK completes
- **For example, how to tell if data bitmap block is consistent?**

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

Fsck Checks

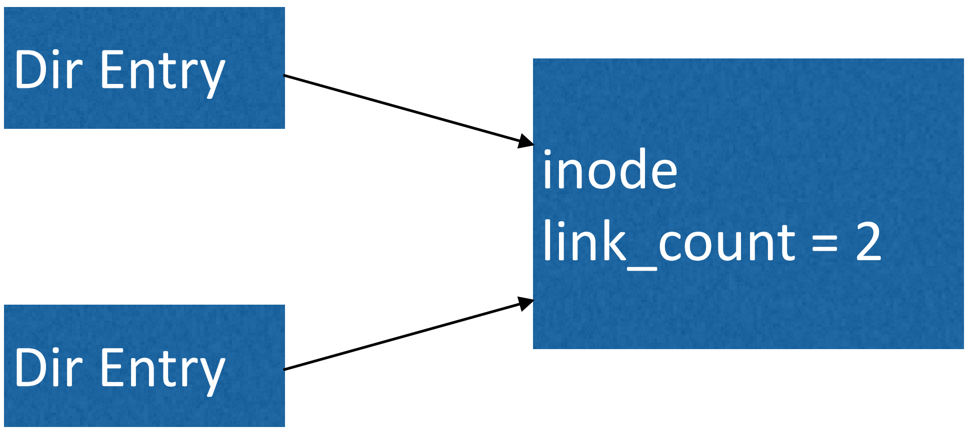
- Hundreds of types of checks over different fields…
- Do superblocks match?
- Do directories contain "." and ".."?
- Do number of dir entries equal inode link counts?
- Do different inodes ever point to same block?
- …
- How to solve problems?

Link Count (example 1)



How to fix to have consistent file system?

Link Count (example 1)



Simple fix!

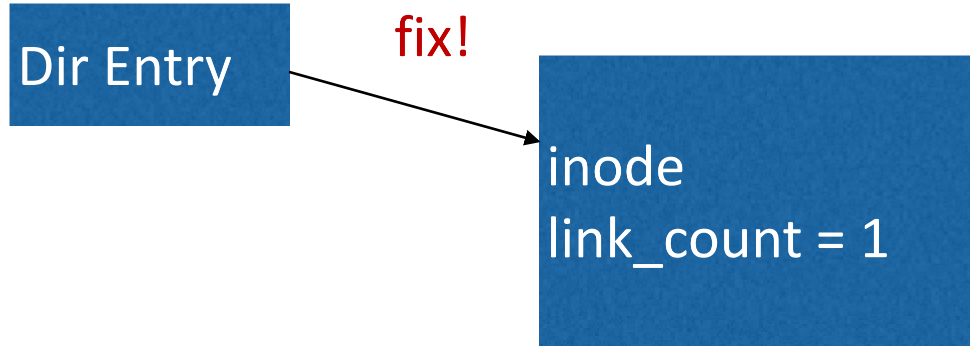
Link Count (example 2)

no dir entry point to it

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

inode
link_count = 1

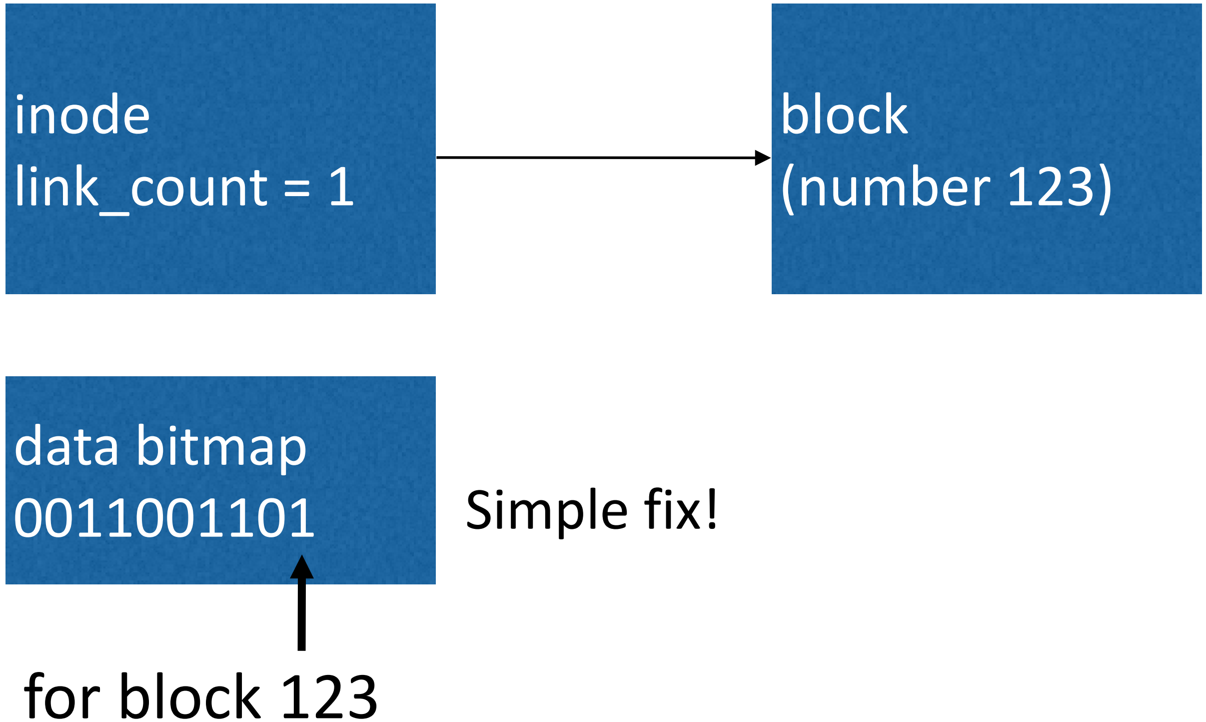
block
(number 123)

data bitmap
0011001100

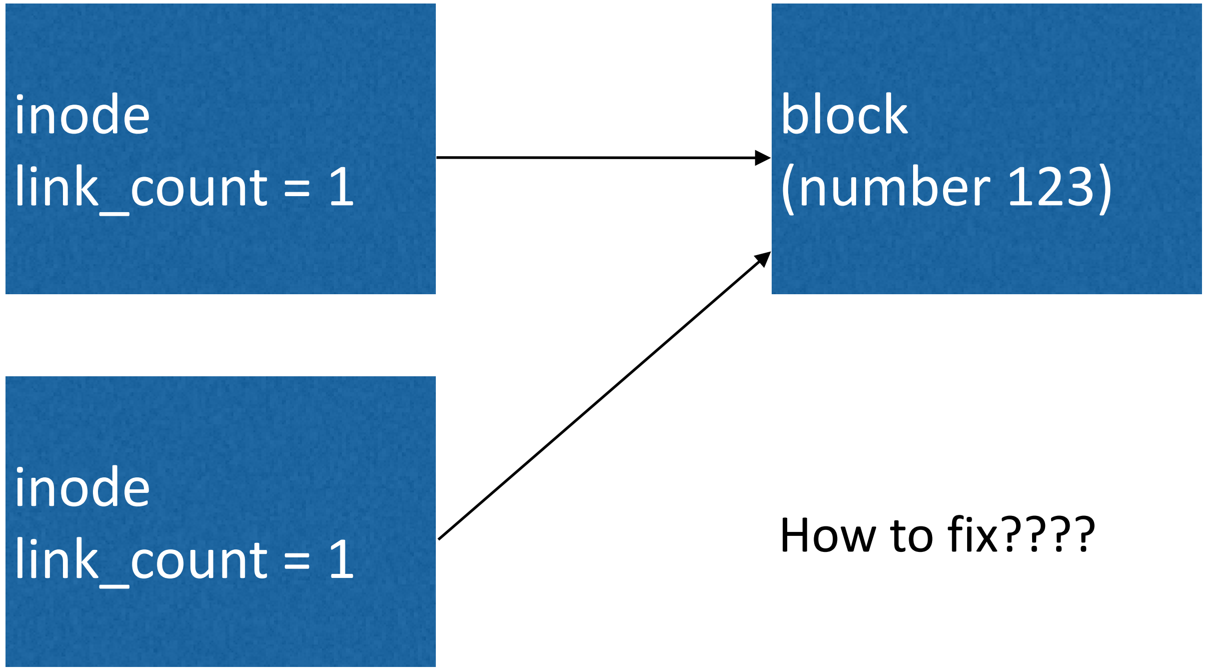
for block 123

How to fix?

Data Bitmap

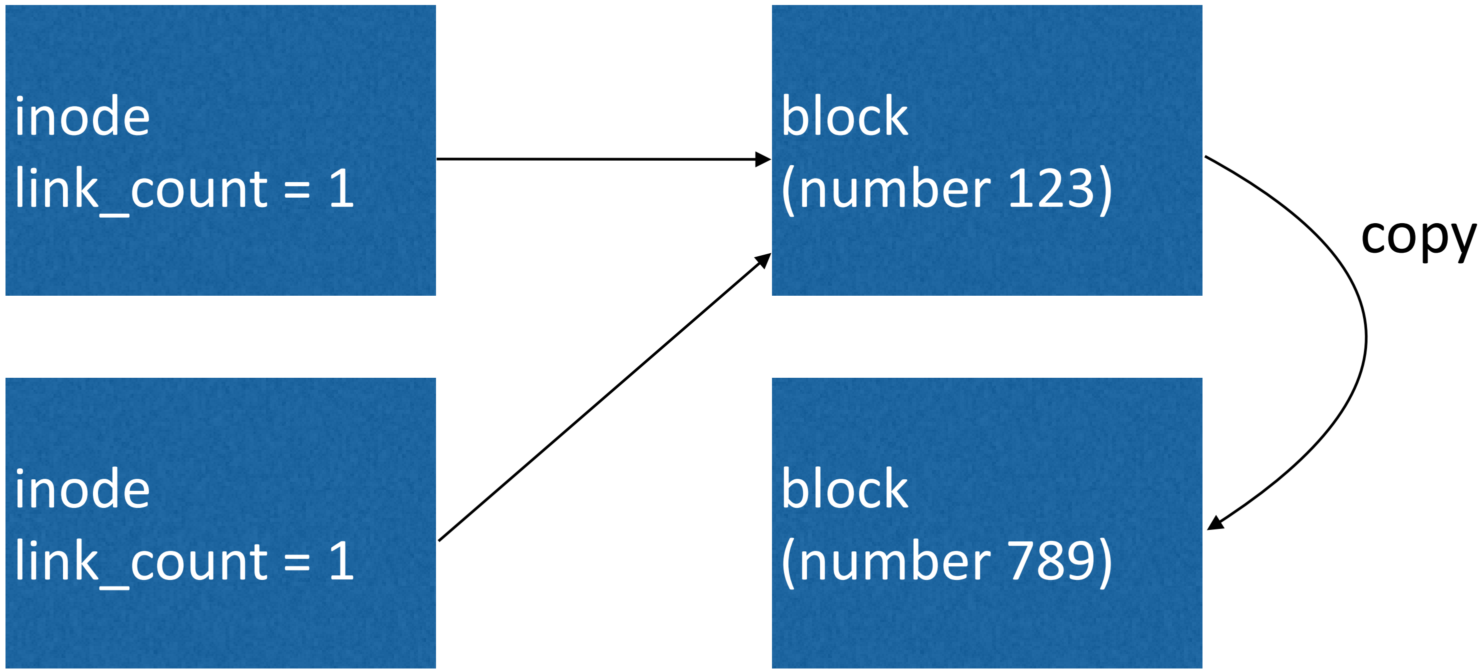

inode
link_count = 1
block
(number 123)
data bitmap
0011001101
Simple fix!
for block 123

Duplicate Pointers

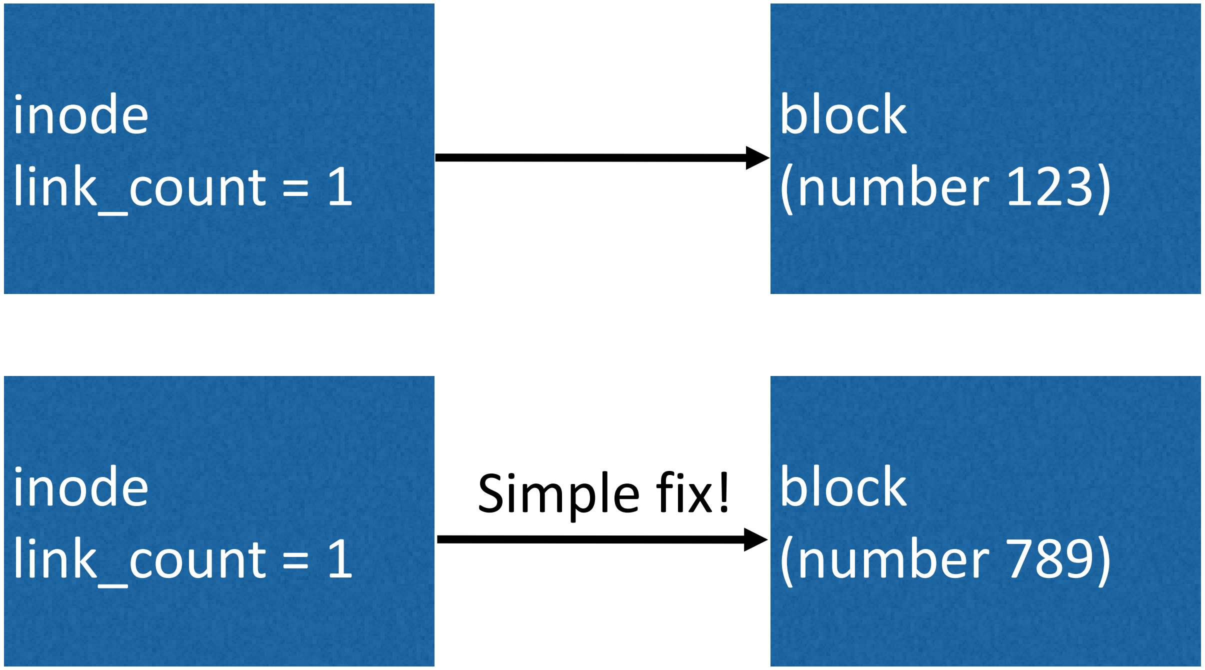


How to fix????

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

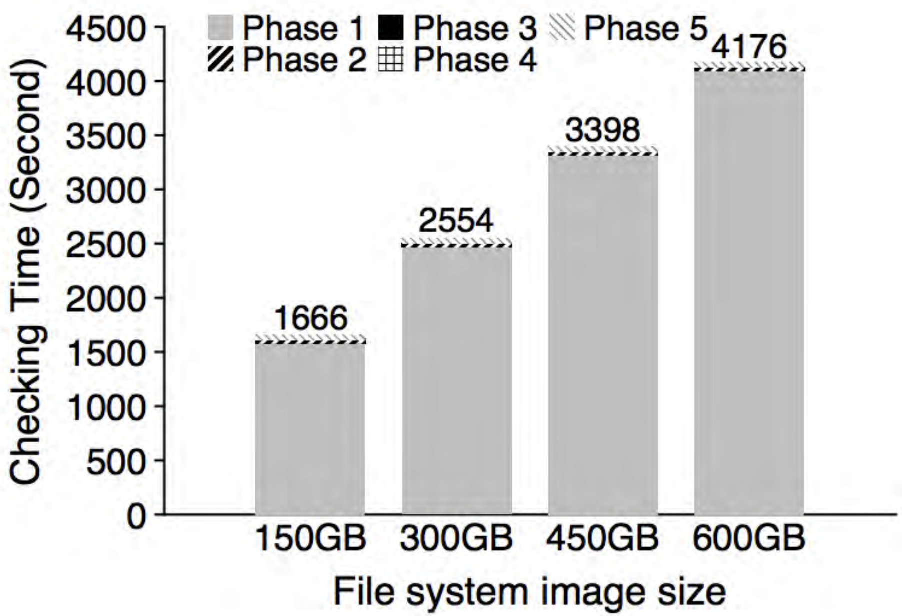
Remove the pointer
Simple fix! (But is this correct?)

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

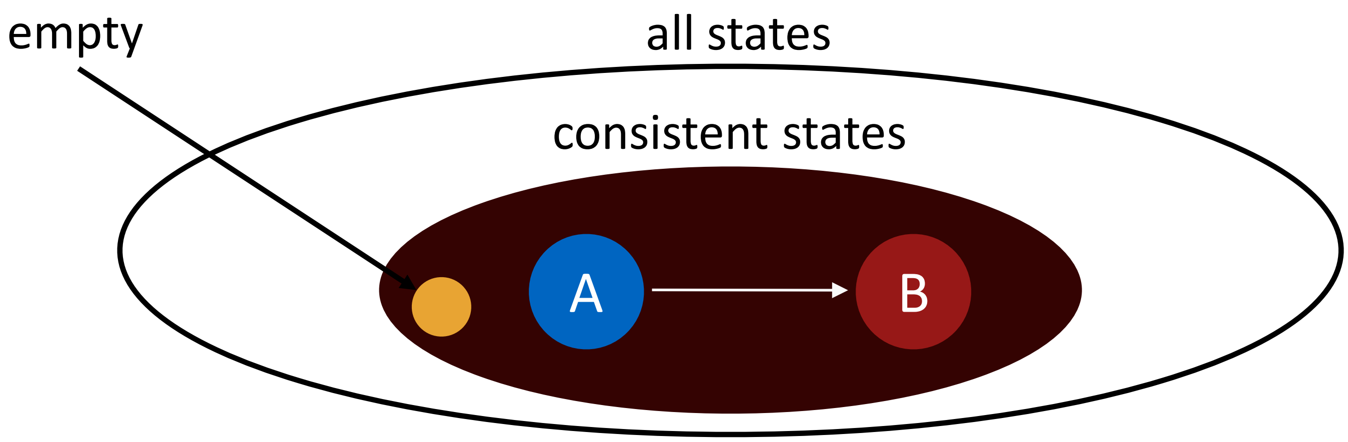
Ao Ma, EMC Corporation and University of Wisconsin—Madison; Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

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
  - 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 Correctness

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



fsck gives consistency
Atomicity gives A or B.

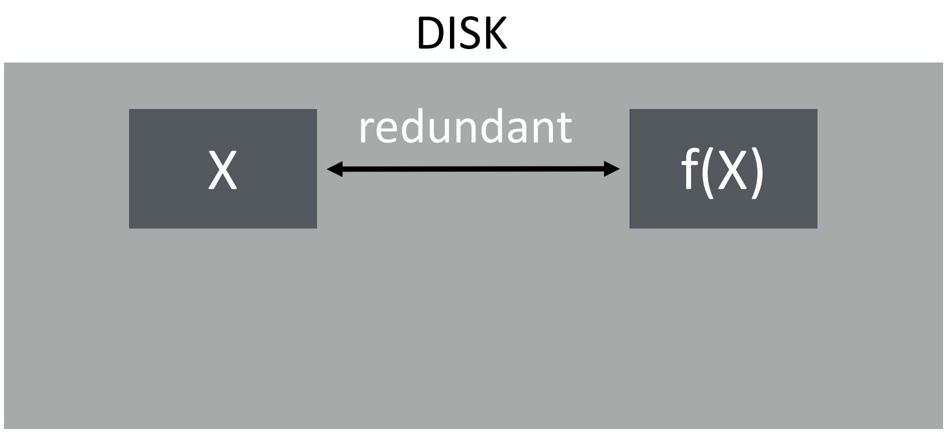
Journaling General Strategy

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

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

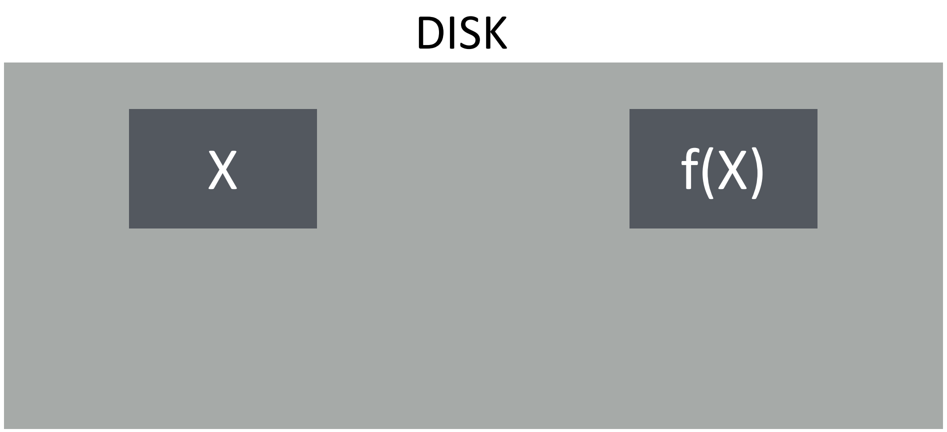
Fight Redundancy with Redundancy

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:

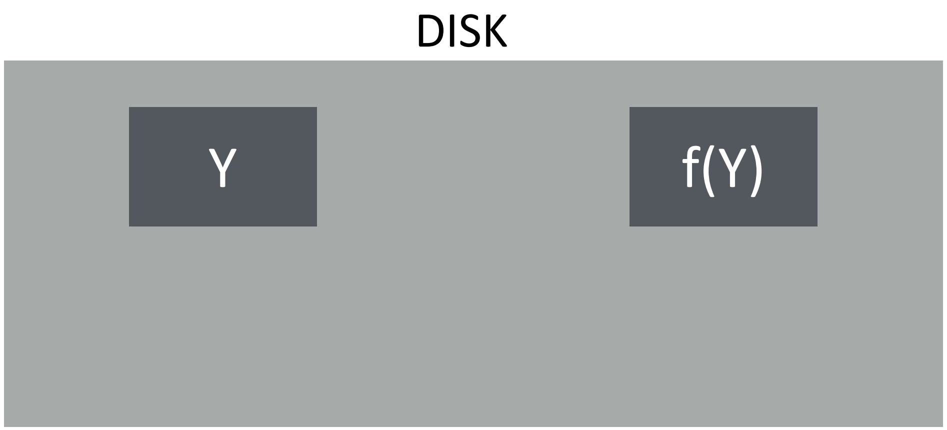
DISK

Good time to crash?

bad time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

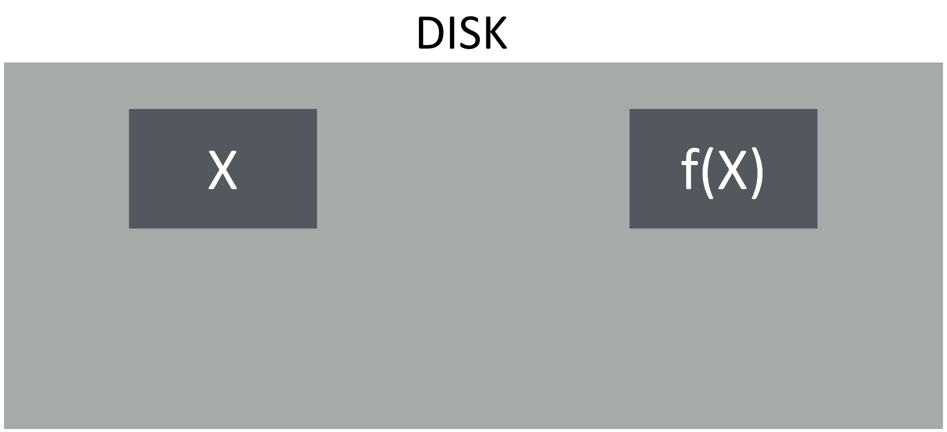


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

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

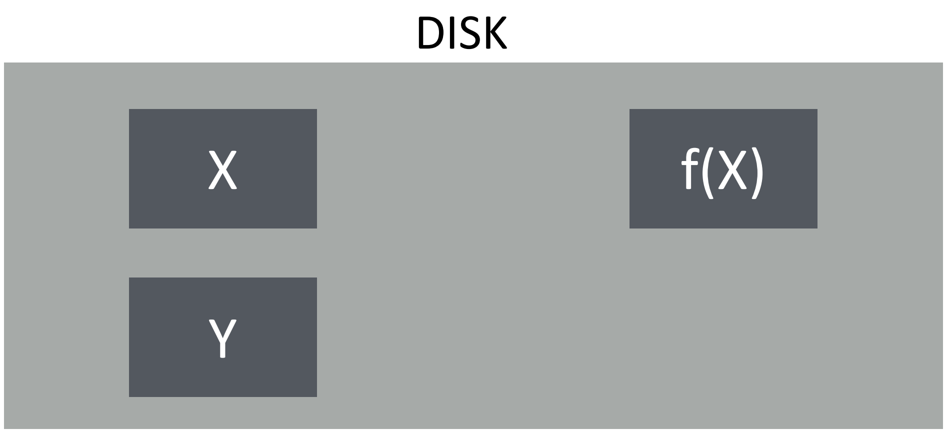


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

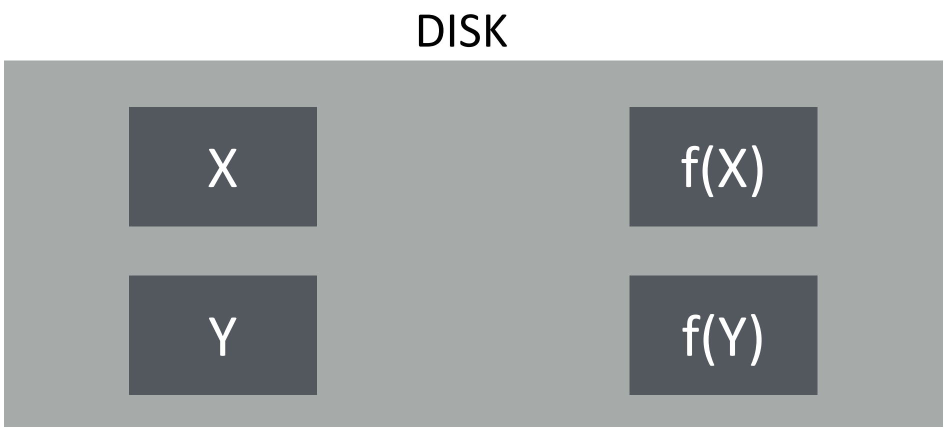
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

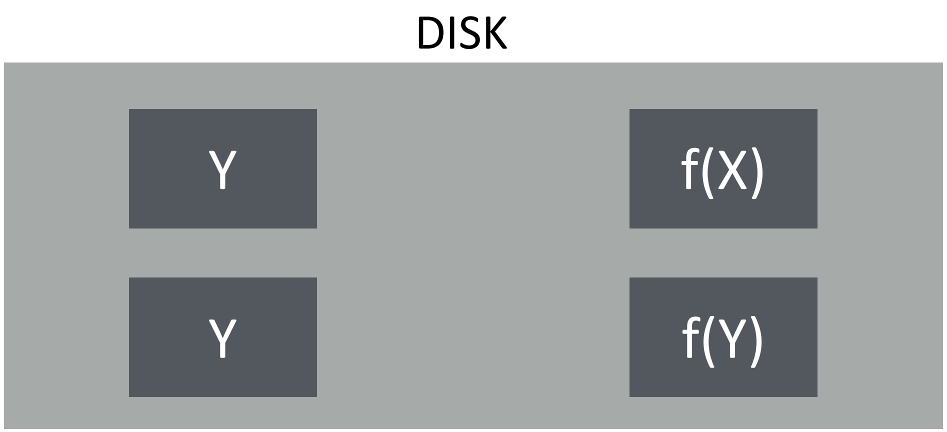
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

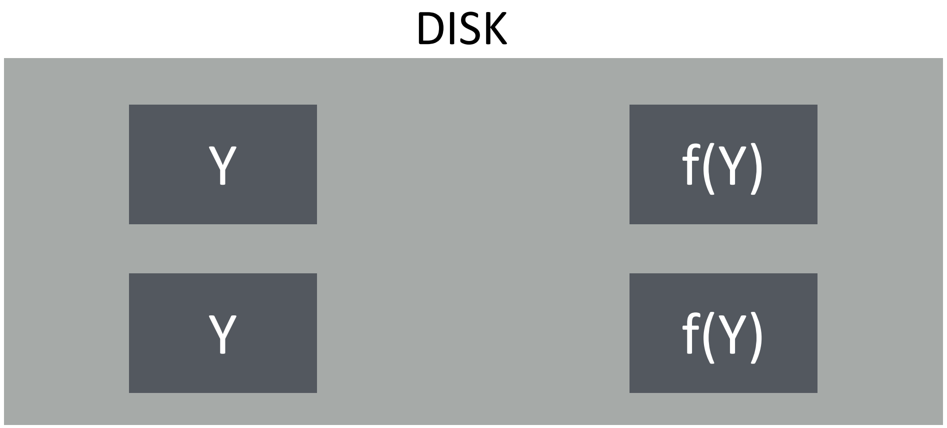
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

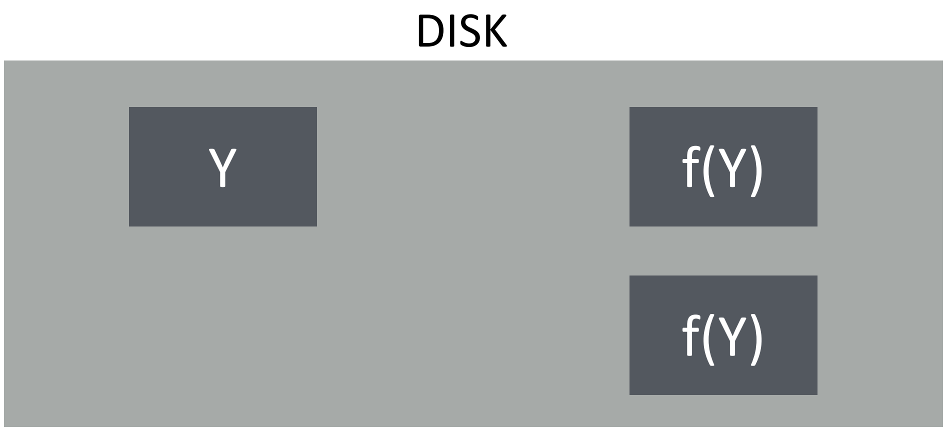
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

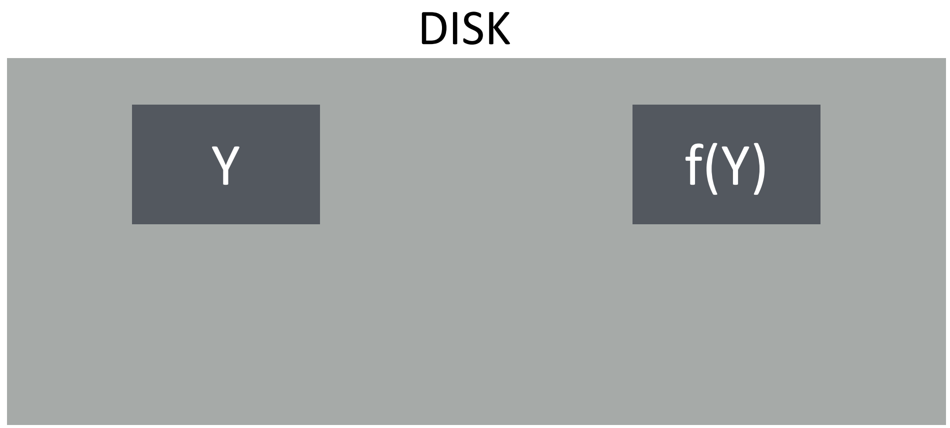
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

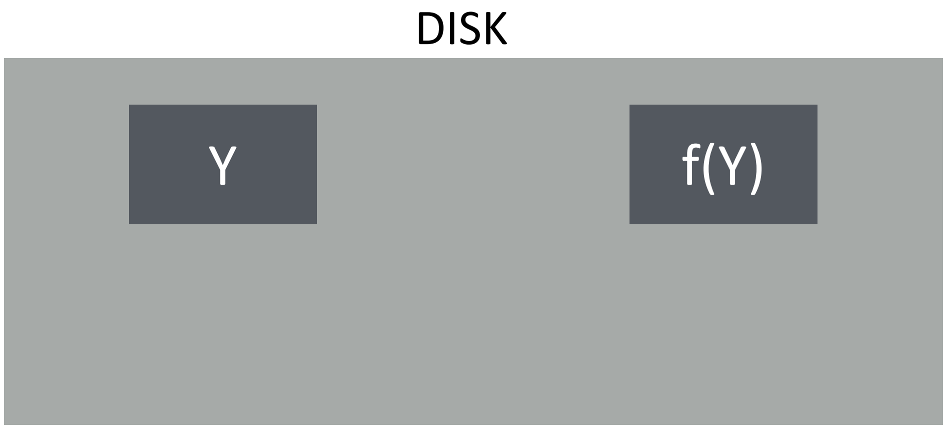
Want to replace X with Y. With journal:



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 inconsistency 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

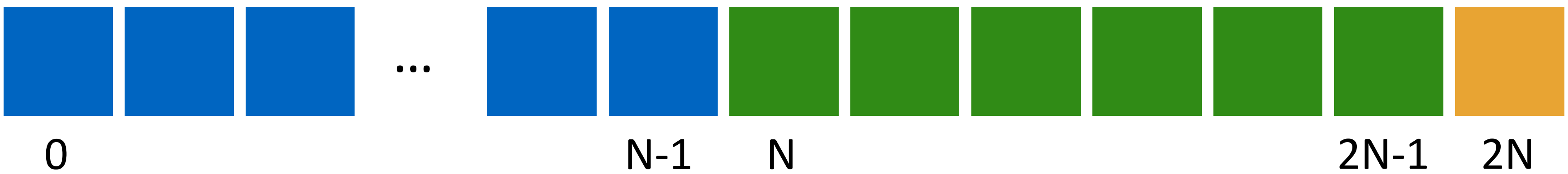
```
void update_accounts(int cash1, int cash2) {
        write(cash1 to block 2) // Alice backup
        write(cash2 to block 3) // Bob backup
        write(1 to block 4)     // backup is safe
        write(cash1 to block 0) // Alice
        write(cash2 to block 1) // Bob
        write(0 to block 4)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

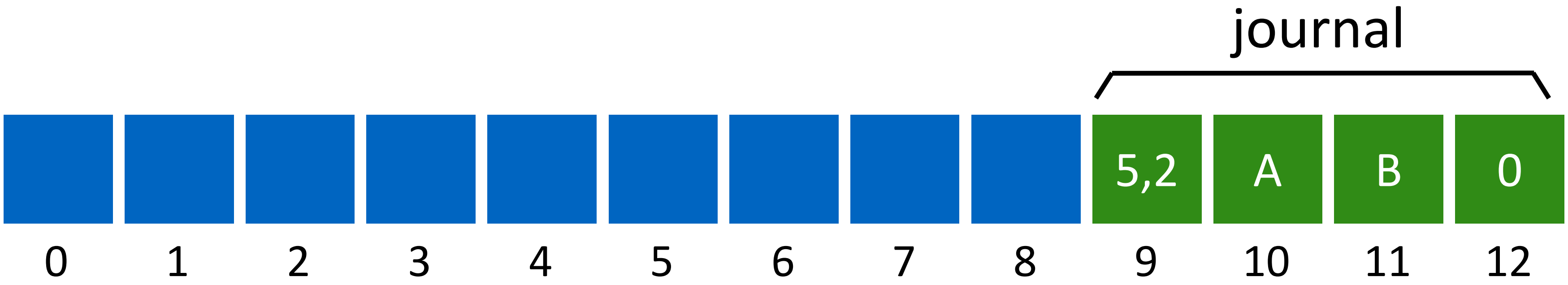
Fix #1: Small Journals

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

- **Goal:**
  - Reuse small area as backup for any block

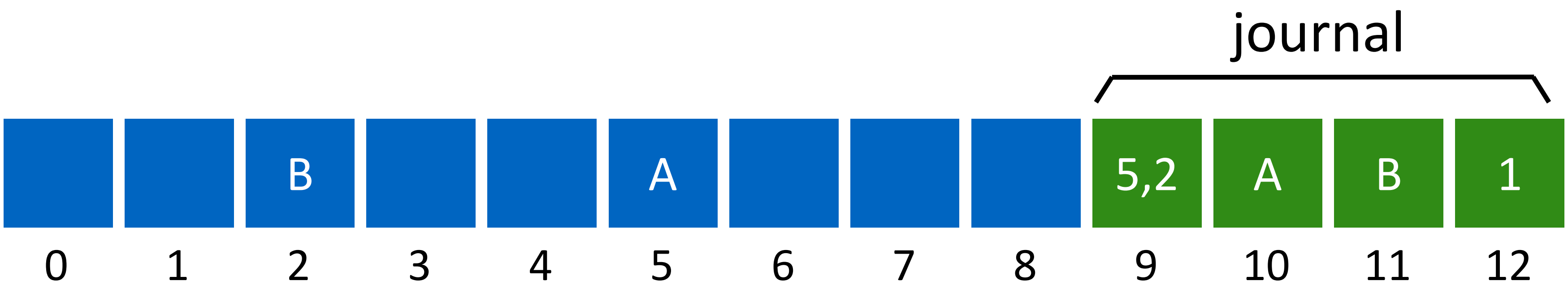
- **How?**
  - Store block numbers in a transaction header

New Layout



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

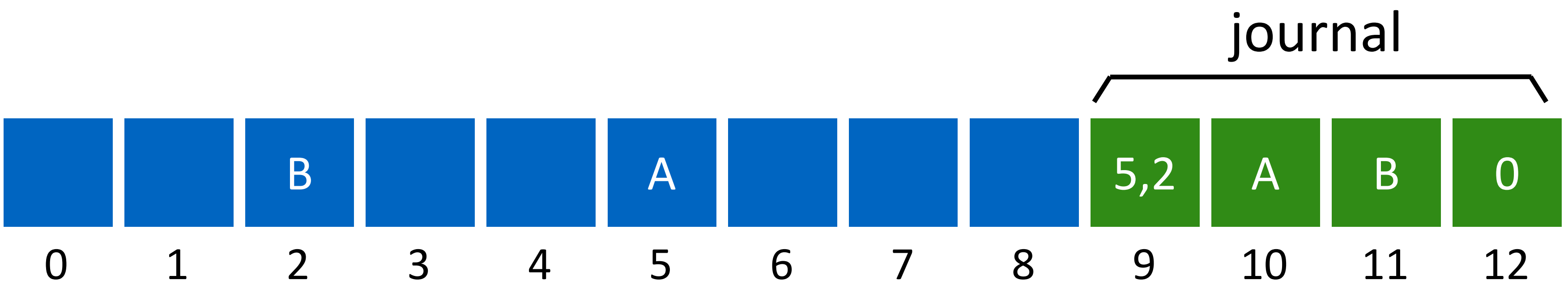
New Layout



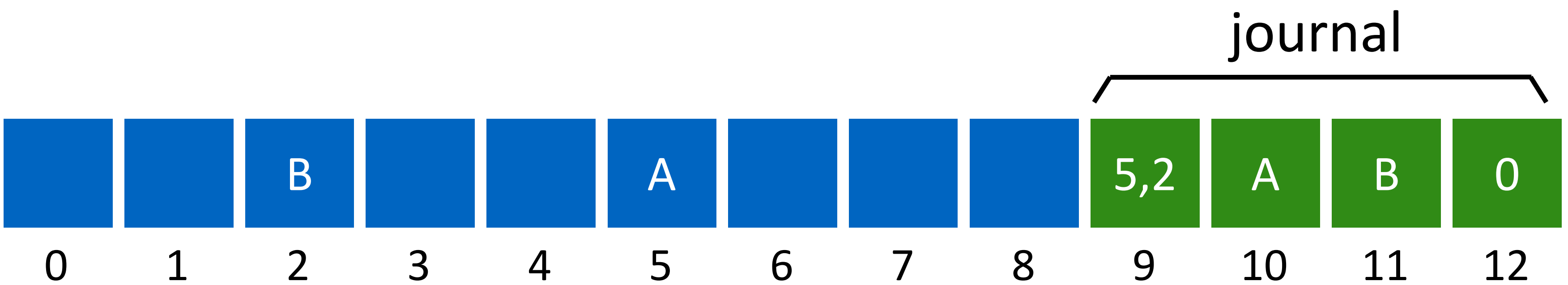
transaction: write A to block 5; write B to block 2

Checkpoint: Writing new data to in-place locations
Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called checkpointing.

New Layout

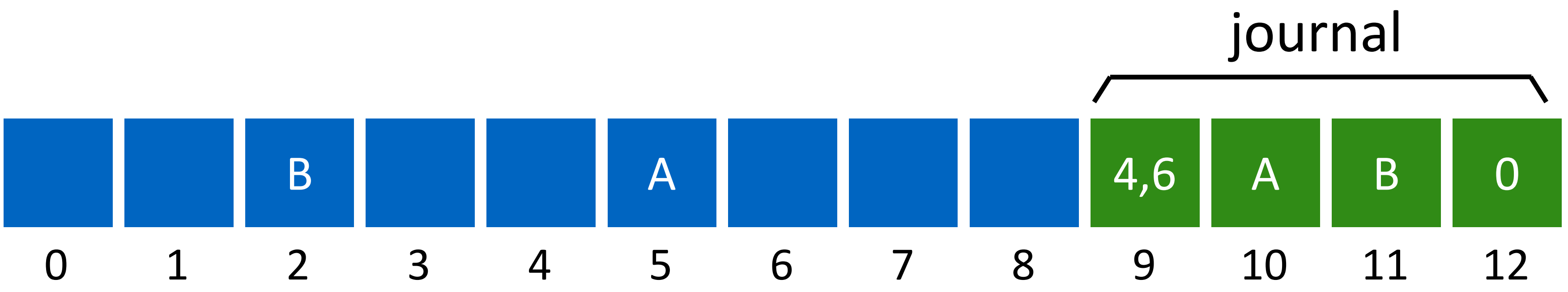


New Layout



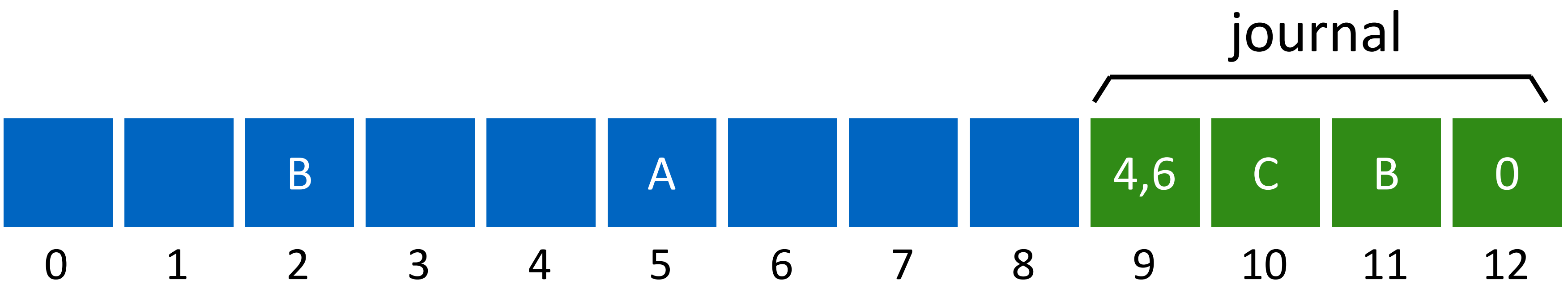
transaction: write C to block 4; write T to block 6

New Layout



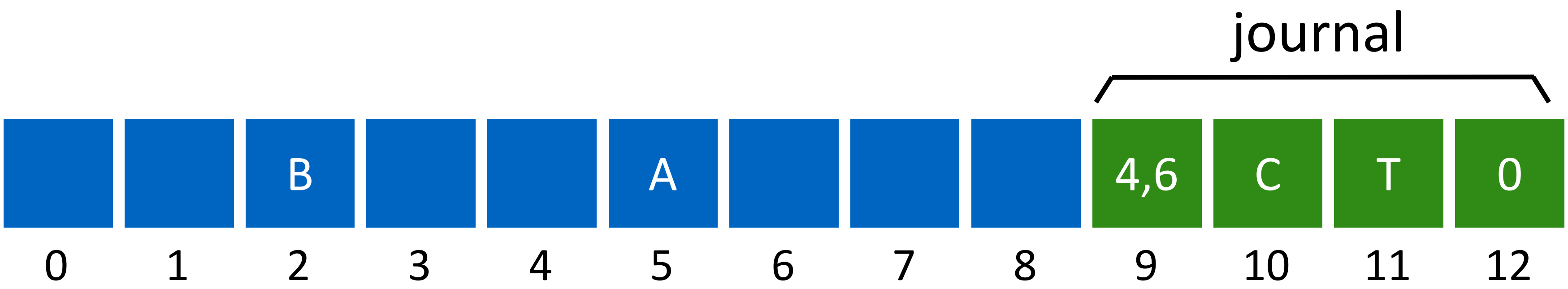
transaction: write C to block 4; write T to block 6

New Layout



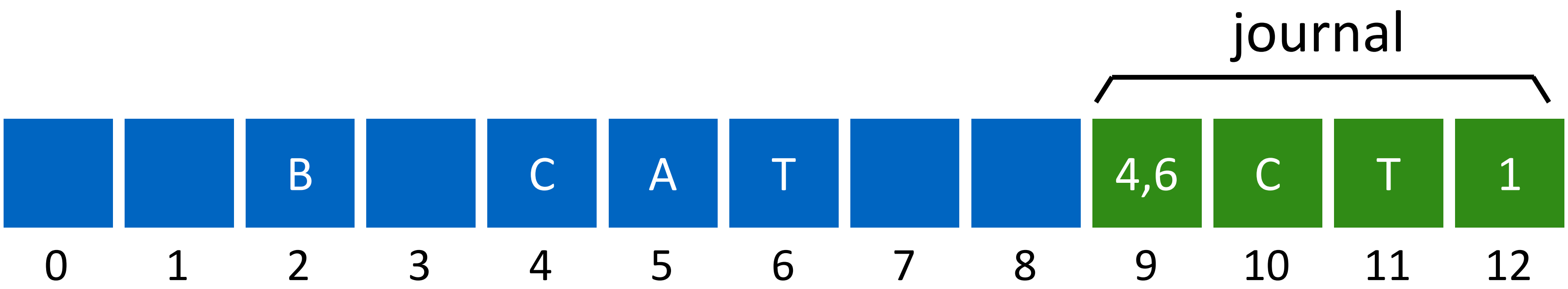
transaction: write C to block 4; write T to block 6

New Layout



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

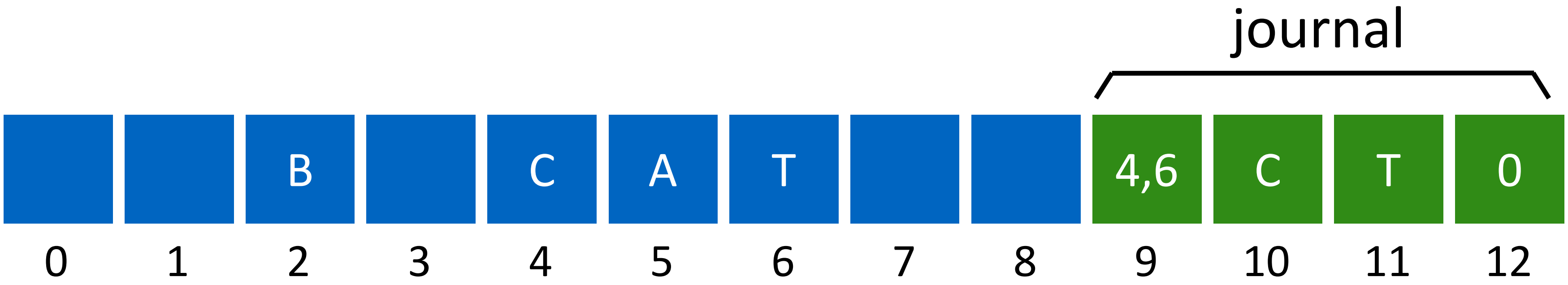
New Layout



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

Checkpoint: Writing new data to in-place locations

New Layout

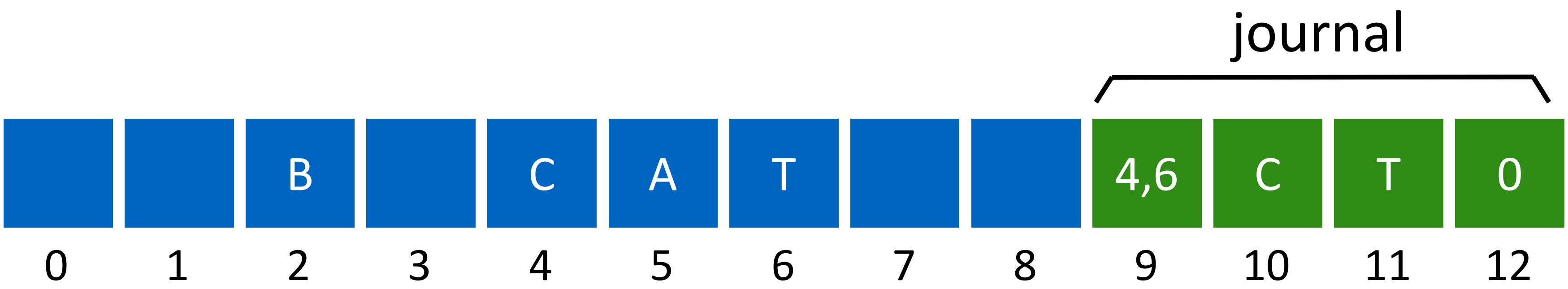


transaction: write C to block 4; write T to block 6

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

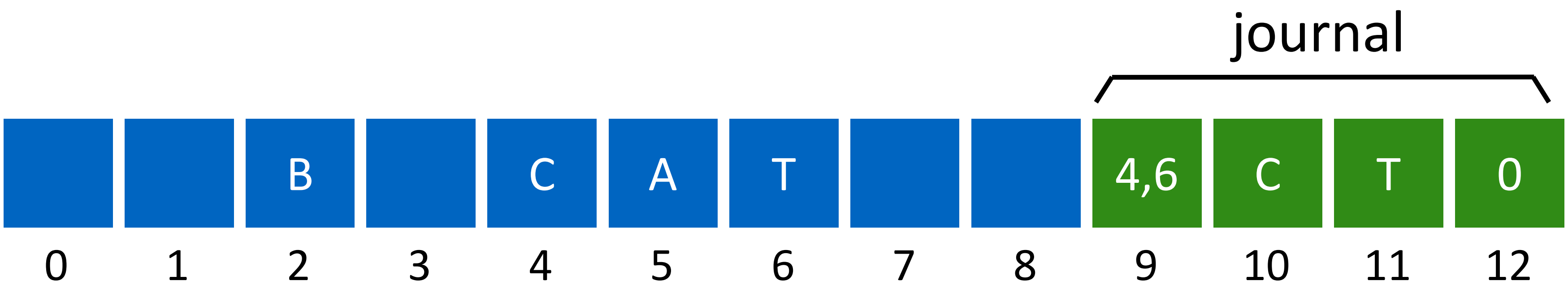
Need ordering to guarantee correctness!

But enforcing total ordering is inefficient. Why?

Random writes (especially with multiple blocks)

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

Ordering



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

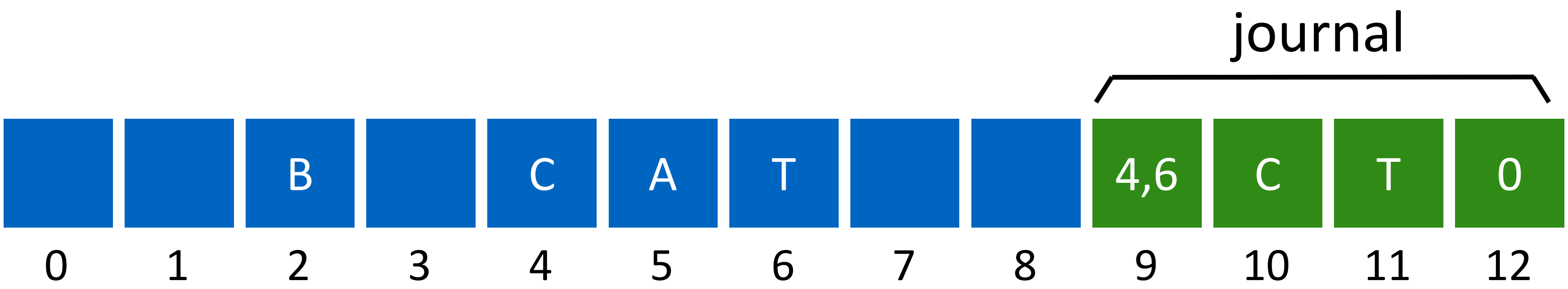
write order: 9,10,11 | 12 | 4,6 | 12

- 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

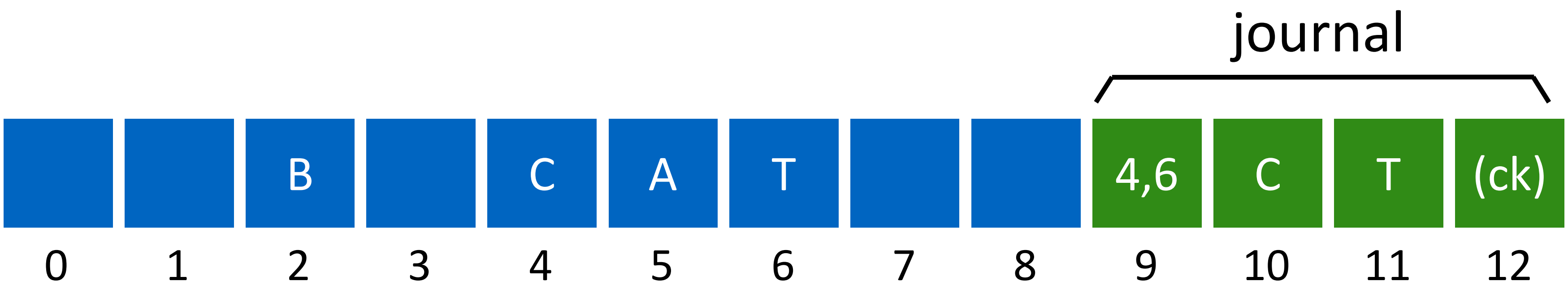
Checksum Optimization



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

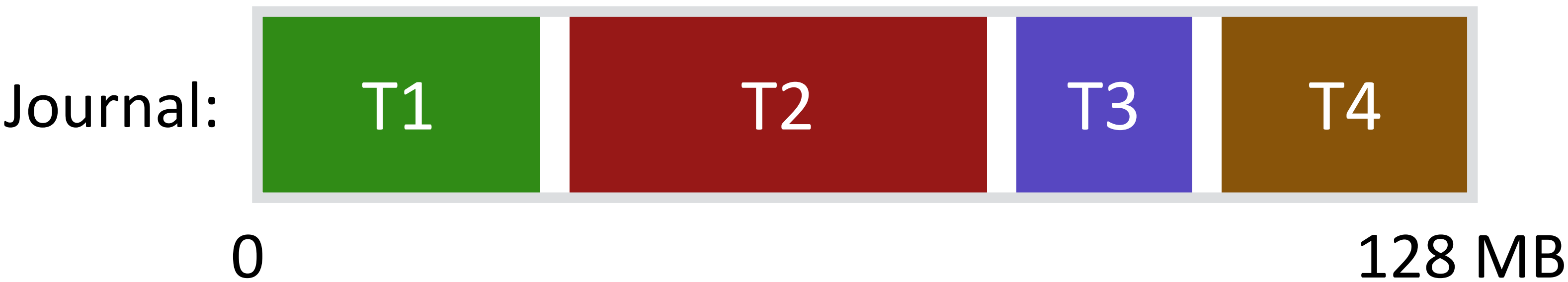
Optimizations

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

Write Buffering Optimization

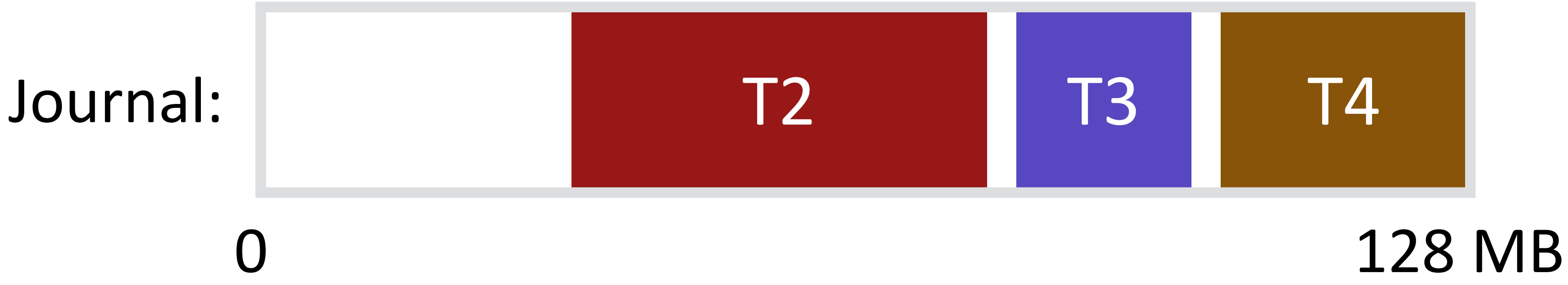
- **Note: after journal write, there is no rush to checkpoint**
  - If system crashes, still have persistent copy of written data!
- **Journaling is sequential, checkpointing is random**
- **Solution? Delay checkpointing for some time**
- **Difficulty: need to reuse journal space**
- **Solution: keep many transactions for un-checkpointed data in a circular buffer**

Circular Buffer



Keep data also in memory until checkpointed on disk

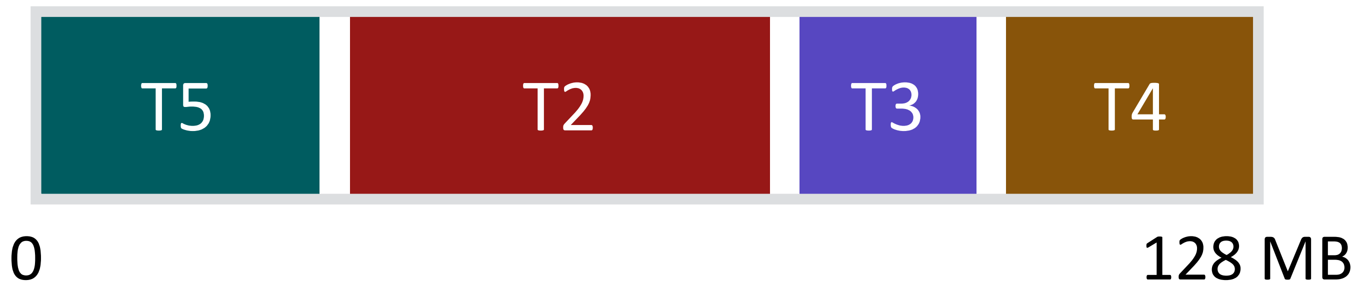
Circular Buffer



checkpoint and cleanup

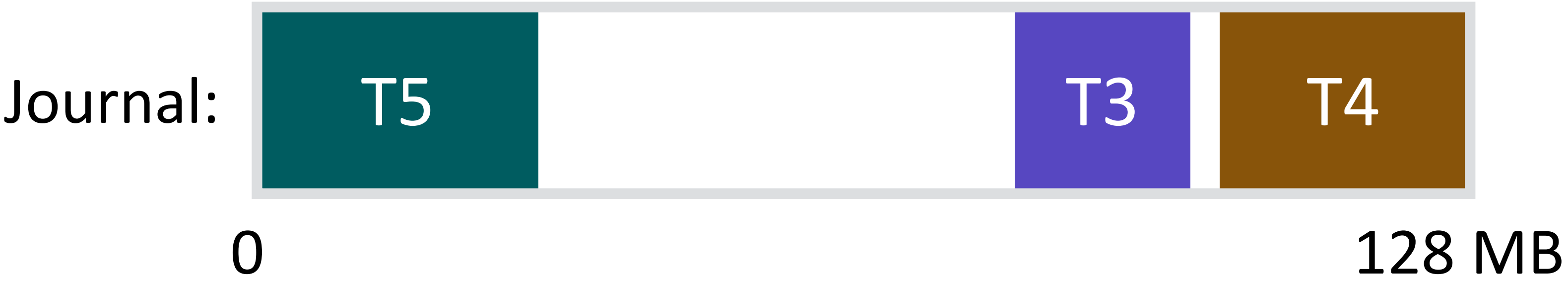
Circular Buffer

Journal:



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

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

Actual changed data is much smaller!

Logical Journal

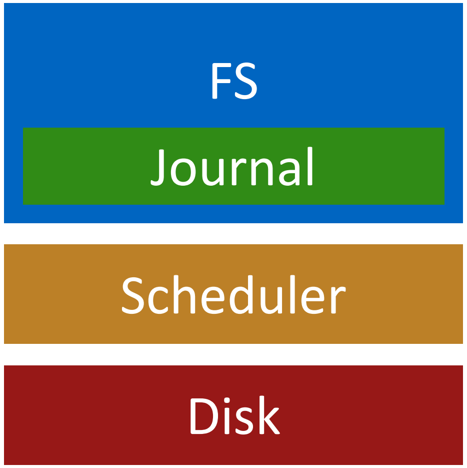
TxB length=1	list of changes	TxE (checksum)

- 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?

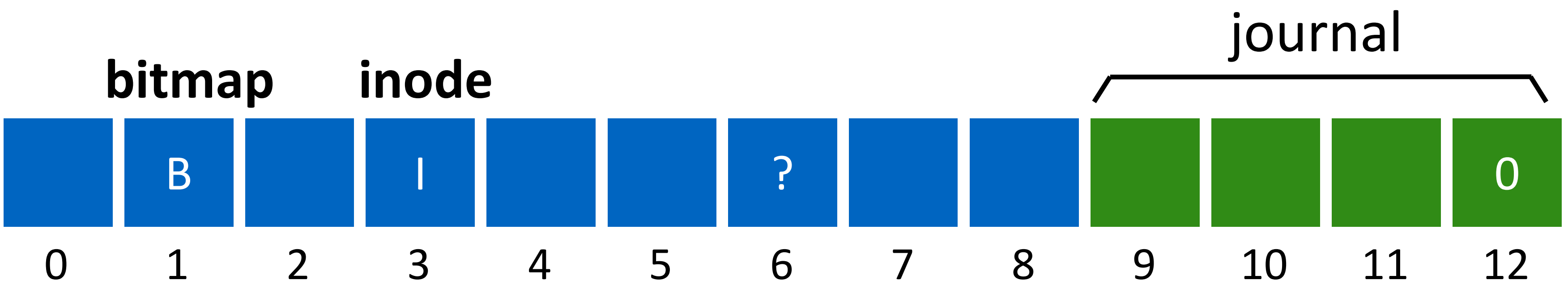
- **Write twice: data + journal**
  - half of peak bandwidth for sequential write
- **Observation: some blocks (e.g., user data) are less important**

- **Strategy: journal all metadata, including:**

  superblock, bitmaps, inodes, indirects, directories

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

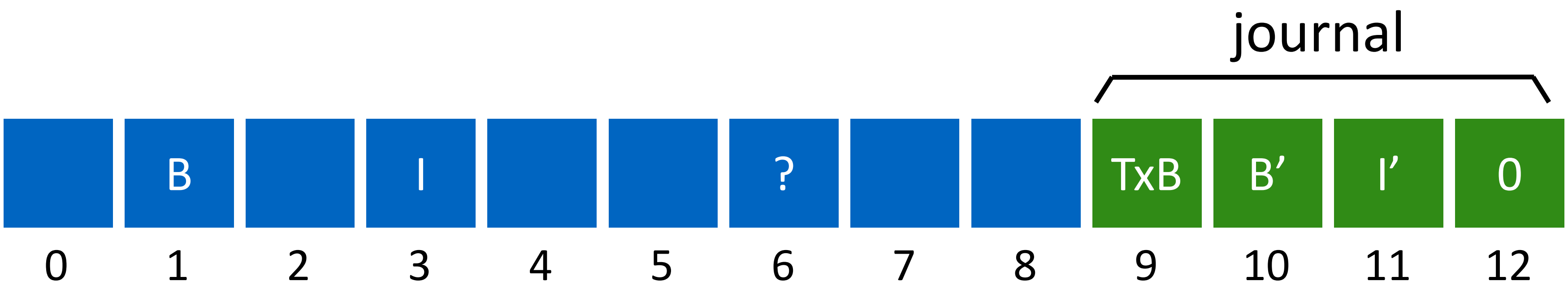
Writeback Journal



transaction: append to inode I

B: bitmap, I: inode

Writeback Journal

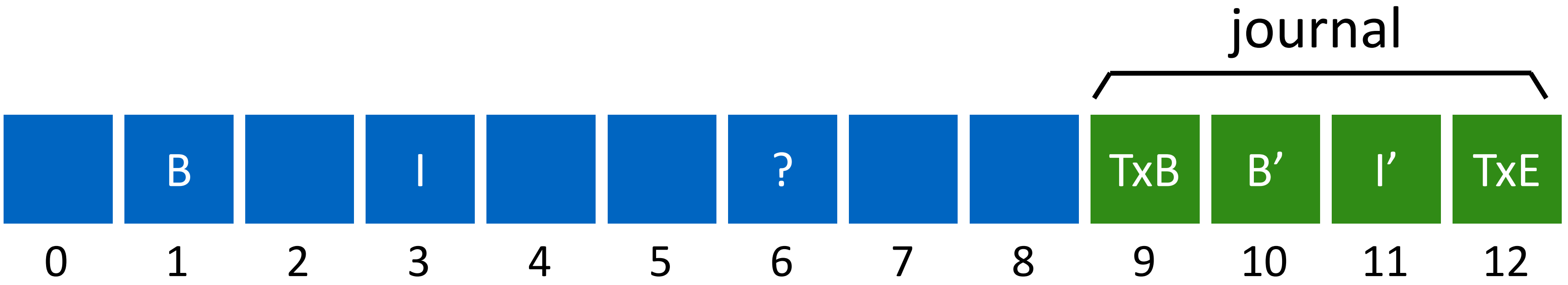


transaction: append to inode I

TxB: transaction begin

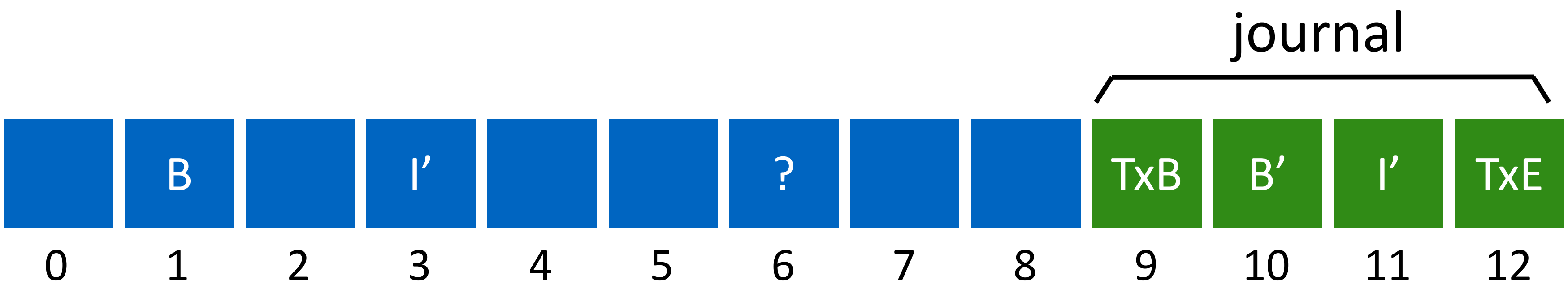
TxE: transaction end

Writeback Journal



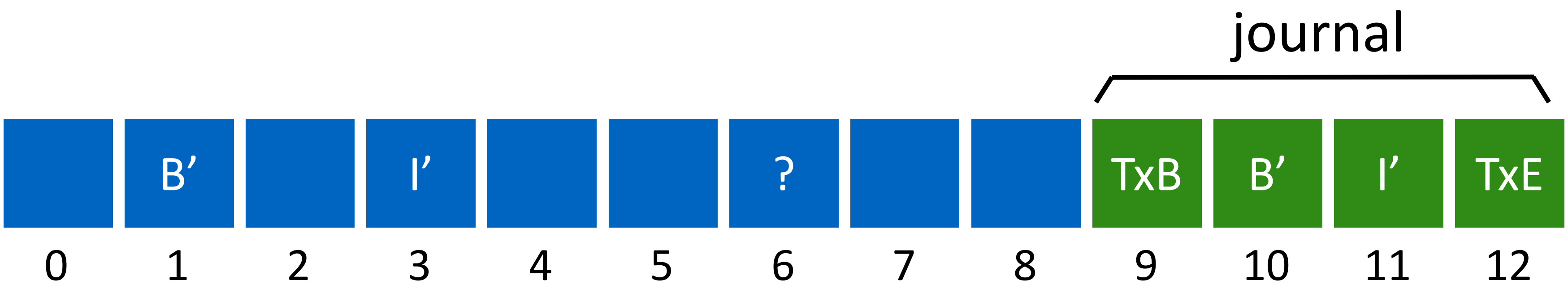
transaction: append to inode I

Writeback Journal



transaction: append to inode I

Writeback Journal



transaction: append to inode I

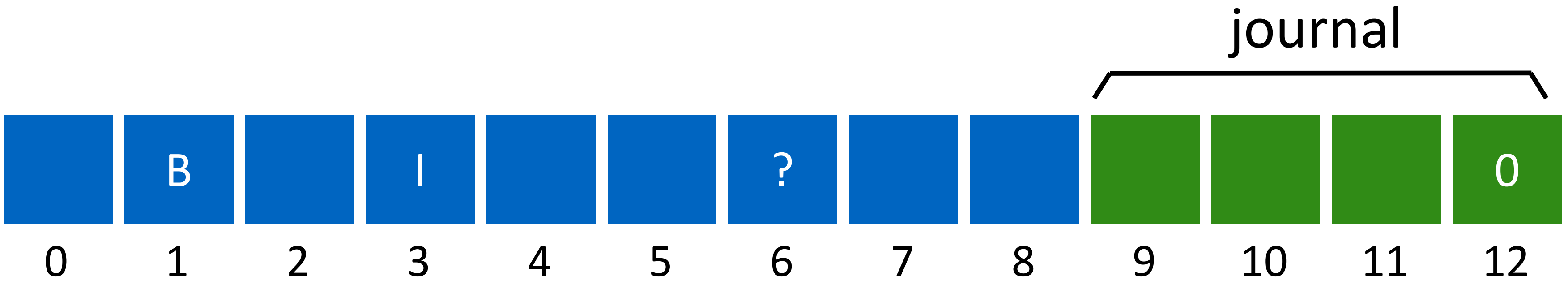
What if we crash now? Solutions?

I points to garbage data in 6;
cannot recover as not being journaled

Ordered Journaling

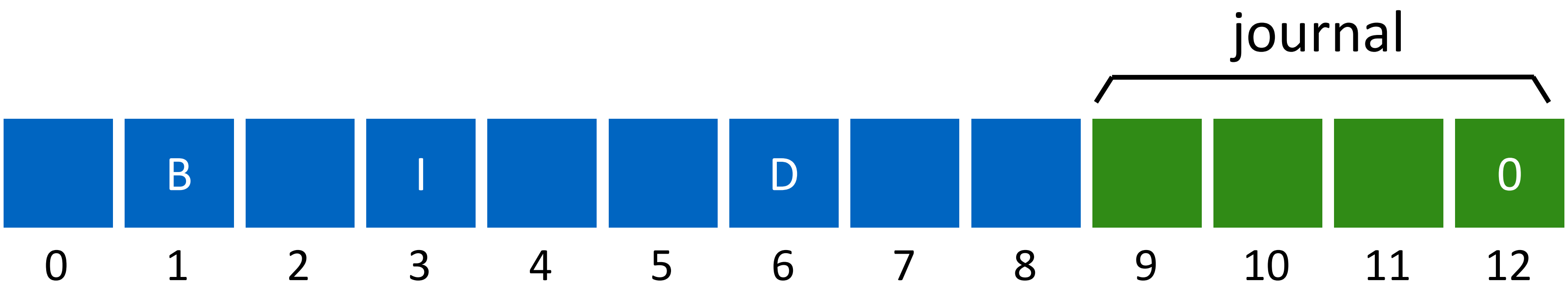
- Still only journal metadata
- But write data before the transaction
- No leaks of sensitive data!

Ordered Journal



transaction: append to inode I

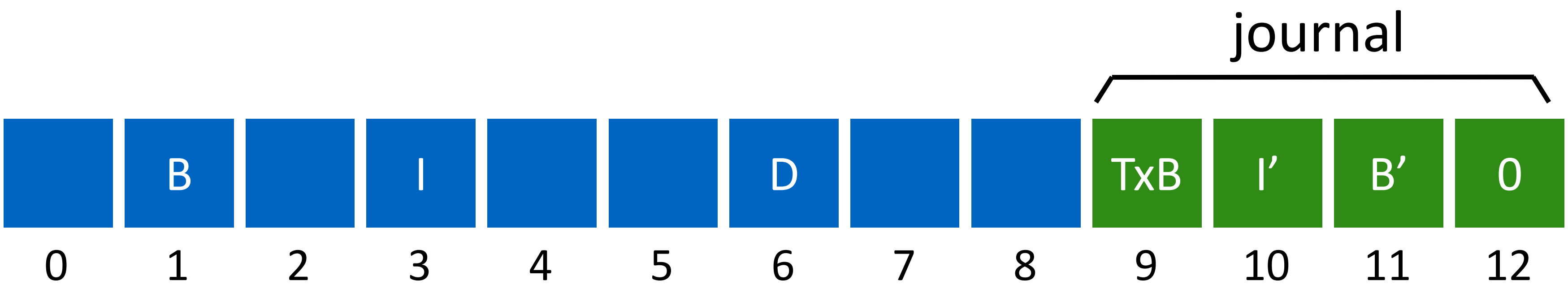
Ordered Journal



transaction: append to inode I

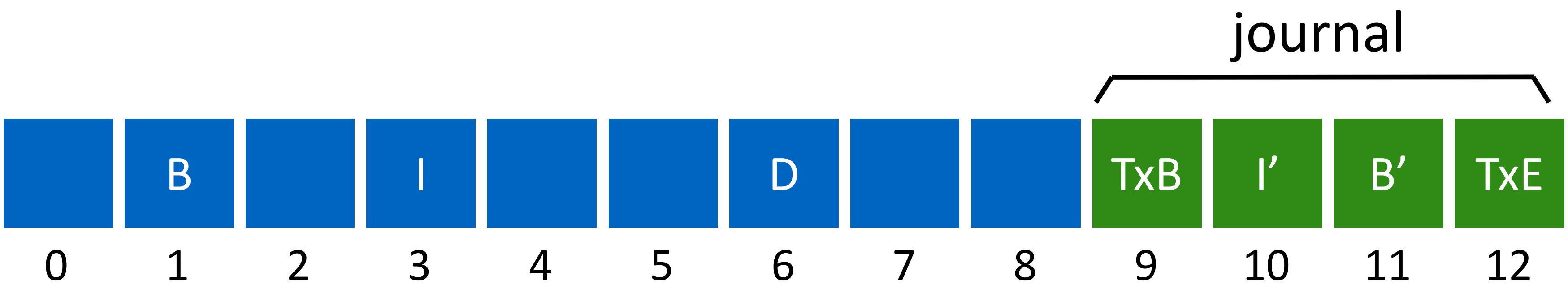
- What happens if crash now?
  - B (bitmap) indicates D currently free
  - I (inode) does not point to D;
  - Lose D, but that might be acceptable

Ordered Journal



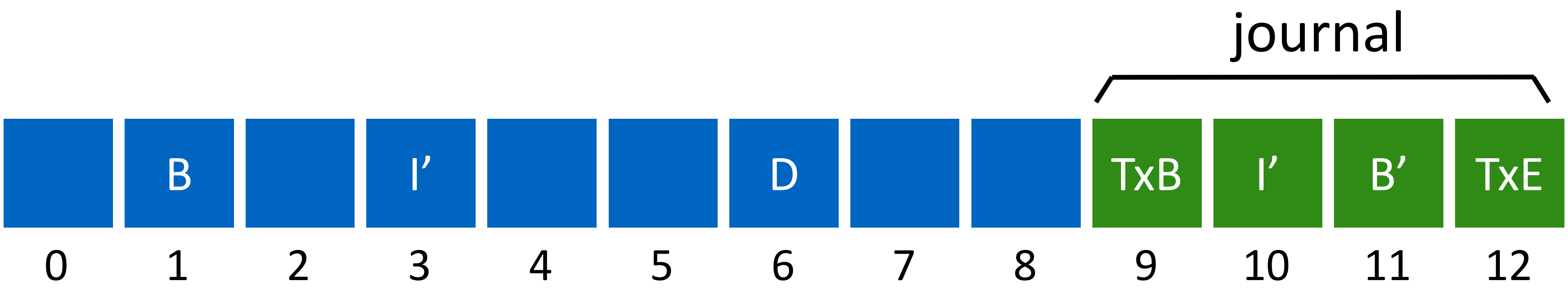
transaction: append to inode I

Ordered Journal



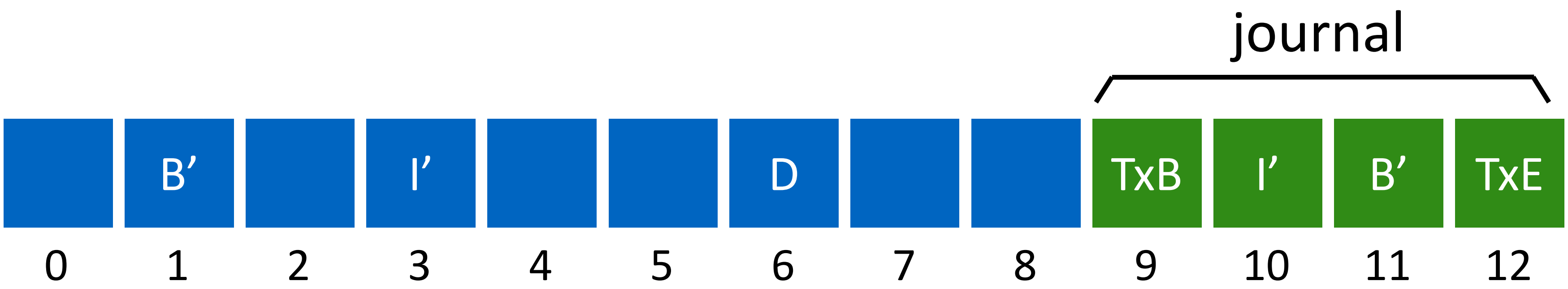
transaction: append to inode I

Ordered Journal



transaction: append to inode I

Ordered Journal



transaction: append to inode I

Ordered Journaling

- **Steps:**
  1. Data write
  2. Journal metadata write
  3. Journal commit
  4. Checkpoint metadata
  5. Free the transaction in journal

1 & 2 can issue write concurrently

- **3 modes in Linux Ext3**
  - Data
  - Ordered
  - Unordered (data can be written at any time)

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)**