OSTEP
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?

Crash Consistency

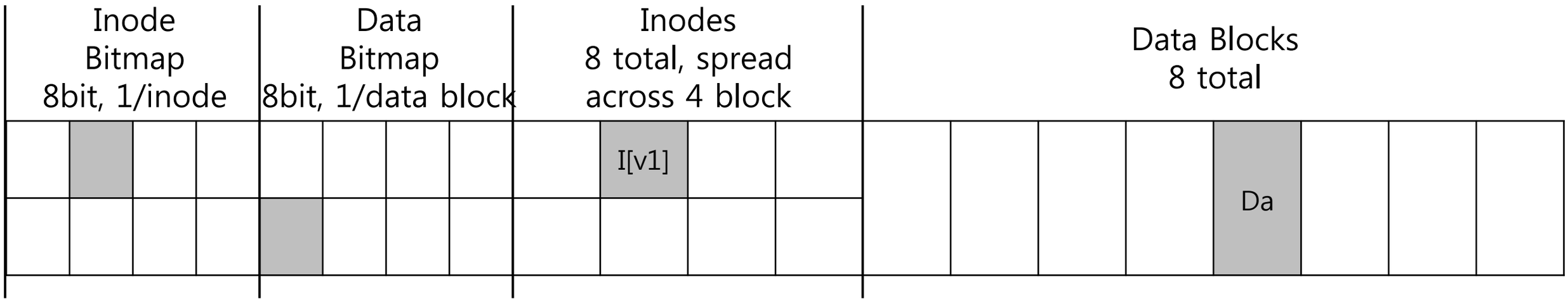
- Unlike most data structure, file system data structures must **persist**
  - They must survive over the long haul, stored on devices that retain data despite power loss.

- One major challenge faced by a file system is how to update persistent data structure despite the presence of a **power loss** or **system crash**.

- We'll begin by examining the approach taken by older file systems.
  - **fsck**(file system checker)
  - **journaling**(write-ahead logging)

A Detailed Example (1)

- Workload
  - Append of a single data block(4KB) to an existing file
  - open() → lseek() → write() → close()

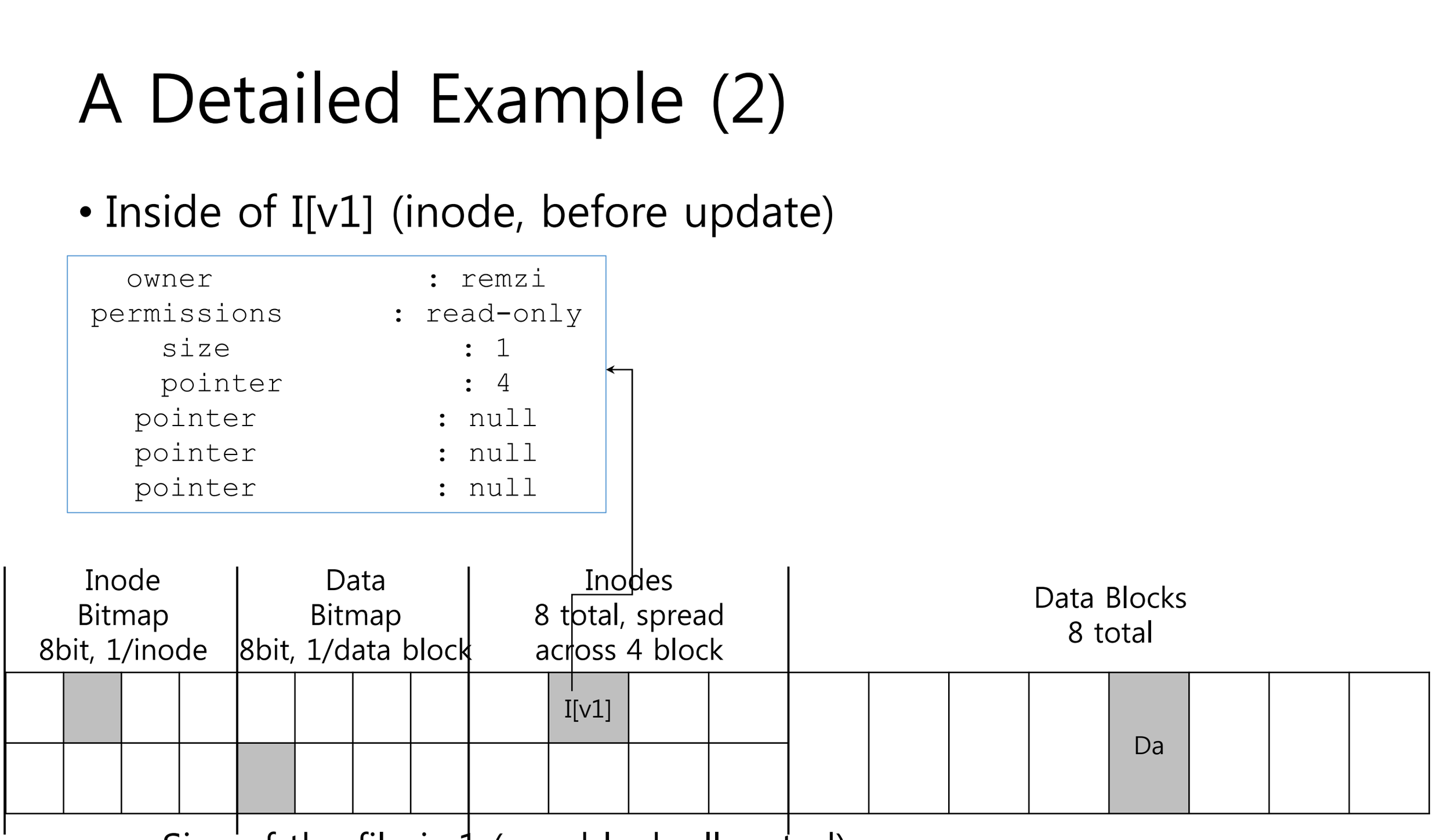


- Before appending a single data block
  - single inode is allocated (inode number 1)
  - single data block is allocated (data block 4)
  - The inode is denoted I[v1]

A Detailed Example (2)

- Inside of I[v1] (inode, before update)

```
   owner             : remzi
permissions        : read-only
    size              : 1
    pointer           : 4
   pointer           : null
   pointer           : null
   pointer           : null
```

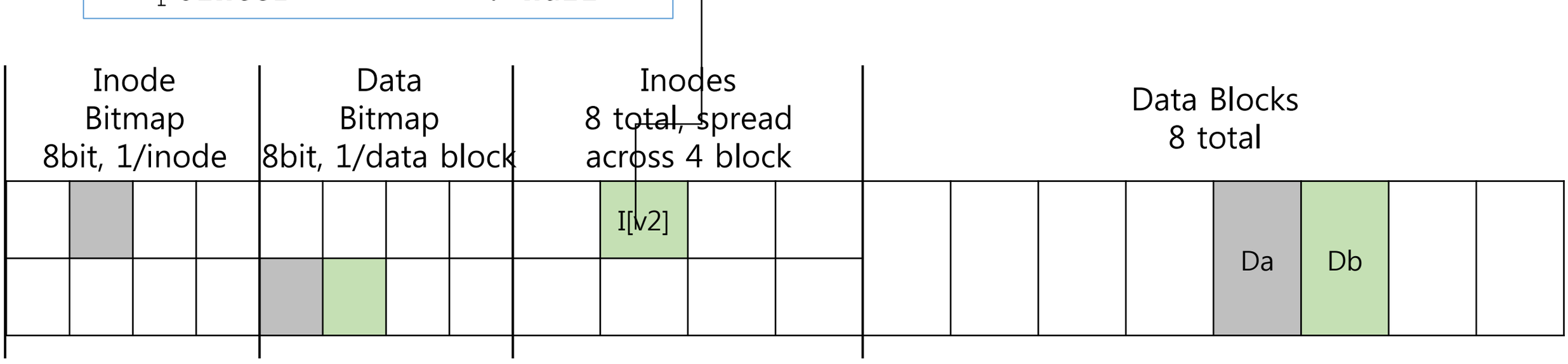


- Size of the file is 1 (one block allocated)
- First direct pointer points to block4 (Da)
- All 3 other direct pointers are set to `null`(unused)

A Detailed Example (3)

- After update

```
   owner              : remzi
 permissions        : read-only
     size               : 2
     pointer            : 4
     pointer            : 5
    pointer            : null
    pointer            : null
```



- Data bitmap is updated
- Inode is updated (I[v2])
- New data block is allocated (Db)

A Detailed Example (end)

- To achieve the transition, the system perform three separate writes to the disk.
  - One each of inode I[v2]
  - Data bitmap B[v2]
  - Data block (Db)

- These writes usually don’t happen immediately
  - dirty inode, bitmap, and new data will sit in main memory
  - **page cache** or **buffer cache**

- If a crash happens after one or two of these write have taken place, but not all three, the file system could be left in a funny state

Crash Scenario (1)

- Imagine only a single write succeeds; there are thus three possible outcomes
  1. Just the data block(Db) is written to disk
     - The data is on disk, but there is no inode
     - Thus, it is as if the write never occurred
     - This case is not a problem at all
  2. Just the updated inode(I[v2]) is written to disk
     - The inode points to the disk address (5, Db)
     - But, the Db has not yet been written there
     - We will read **garbage** data(old contents of address 5) from the disk
     - **Problem : file-system inconsistency**

Crash Scenario (2)

- Imagine only a single write succeeds; there are thus three possible outcomes (Cont.)
  3. Just the updated bitmap (B[v2]) is written to disk
     - The bitmap indicates that block 5 is allocated
     - But there is no inode that points to it
     - Thus, the file system is inconsistent again
     - **Problem : space leak,** as block 5 would never be used by the file system

Crash Scenario (3)

- There are also three more crash scenarios. In these cases, two writes succeed and the last one fails

1. The inode(I[v2]) and bitmap(B[v2]) are written to disk, but not data(Db)
   - The file system metadata is completely consistent
   - **Problem : Block 5 has garbage in it**

2. The inode(I[v2]) and the data block(Db) are written, but not the bitmap(B[v2])
   - We have the inode pointing to the correct data on disk
   - **Problem : inconsistency between the inode and the old version of the bitmap(B1)**

Crash Scenario (end)

- There are also three more crash scenarios. In these cases, two writes succeed and the last one fails (Cont.)
  3. The bitmap(B[v2]) and data block(Db) are written, but not the inode(I[v2])
     - **Problem : inconsistency between the inode and the data bitmap**
     - We have no idea which file it belongs to

The Crash Consistency Problem

- What we'd like to do ideally is move the file system from on consistent state to another **atomically**

- Unfortunately, we can't do this easily
  - The disk only commits one write at a time
  - Crashes or power loss may occur between these updates

- We call this general problem the **crash-consistency problem**

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 sys tem 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 bl
ocks

If file system is interrupted between writes, may leave data i
n 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**

Solution #1: The File System Checker

The File System Checker (1)

- The File System Checker (**`fsck`**)
  - `fsck` is a Unix tool for finding inconsistencies and repairing them.
  - `fsck` check super block, Free block, Inode state, Inode links, etc.
  - Such an approach can’t fix all problems
    - example : The file system looks consistent but the inode points to garbage data.
  - The only real goal is to make sure the file system metadata is internally consistent.

The File System Checker (2)

- Basic summary of what `fsck` does:
  - **Superblock**
    - `fsck` first checks if the superblock looks reasonable
      - Sanity checks : file system size > number of blocks allocated
    - Goal : to find suspect superblock
    - In this case, the system may decide to use an alternate copy of the superblock
  - **Free blocks**
    - `fsck` scans the inodes, indirect blocks, dobule indirect blocks,
    - The only real goal is to make sure the file system metadata is internally consistent.

The File System Checker (3)

- Basic summary of what `fsck` does: (Cont.)
  - **Inode state**
    - Each inode is checked for corruption or other problem
      - Example : type checking(regular file, directory, symbolic link, etc)
    - If there are problems with the inode fields that are not easily fixed.
      - The inode is considered suspect and cleared by `fsck`
  - **Inode Links**
    - `fsck` also verifies the link count of each allocated inode
      - To verify the link count, `fsck` scans through the entire directory tree
    - If there is a mismatch between the newly–calculated count and that found within an inode, corrective action must be taken
      - Usually by fixing the count with in the inode

The File System Checker (4)

- Basic summary of what `fsck` does: (Cont.)
  - **Inode Links** (Cont.)
    - If an allocated inode is discovered but no directory refers to it, it is moved to the lost+found directory
  - **Duplicates**
    - fsck also checks for duplicated pointers
    - Example : Two different inodes refer to the same block
      - If on inode is obviously bad, it may cleared
      - Alternately, the pointed-to block could be copied

The File System Checker (5)

- Basic summary of what `fsck` does: (Cont.)
  - **Bad blocks**
    - A check for bad block pointers is also performed while scanning through the list of all pointers
    - A pointer is considered “bad” if it obviously points to something outside its valid range
      - Example : It has an address that refers to a block greater than the partition size
        - In this case, `fsck` can’t do anything too intelligent; it just removes the pointer

The File System Checker (6)

- Basic summary of what `fsck` does: (Cont.)
  - **Directory checks**
    - `fsck` does not understand the contents of user files
      - However, directories hold specifically formatted information created by the file system itself
      - Thus, `fsck` performs additional integrity checks on the contents of each directory
    - Example
      - making sure that "." and ".." are the first entries
      - each inode referred to in a directory entry is allocated?
      - ensuring that no directory is linked to more than once in the entire hierarchy

The File System Checker (end)

- Buliding a working `fsck` requires intricate knowledge of the filesystem
- `fsck` have a bigger and fundamental problem: **too slow**
  - scanning the entire disk may take many minutes or hours
  - Performance of `fsck` became prohibitive.
    - as disk grew in capacity and RAIDs grew in popularity
- At a higher level, the basic premise of fsck seems just a tad irrational.
  - It is incredibly expensive to scan the entire disk
  - It works but is wasteful
  - Thus, as disk(and RAIDs) grew, researchers started to look for other solutions

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?

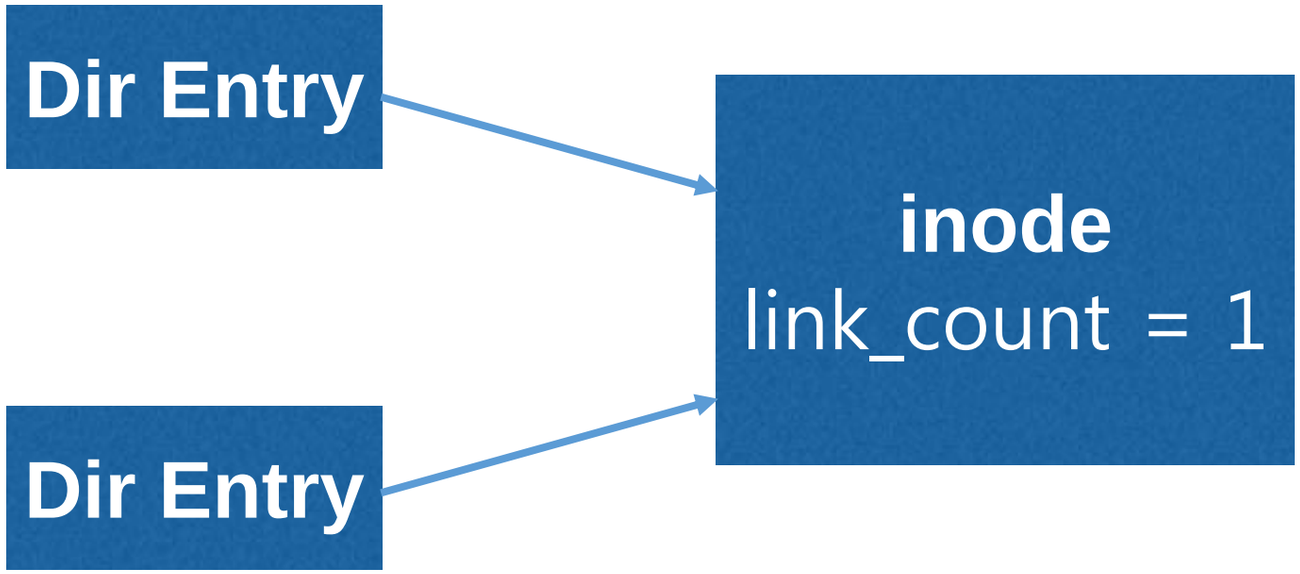
Read every valid inode+indirect block

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

Fsck Checks

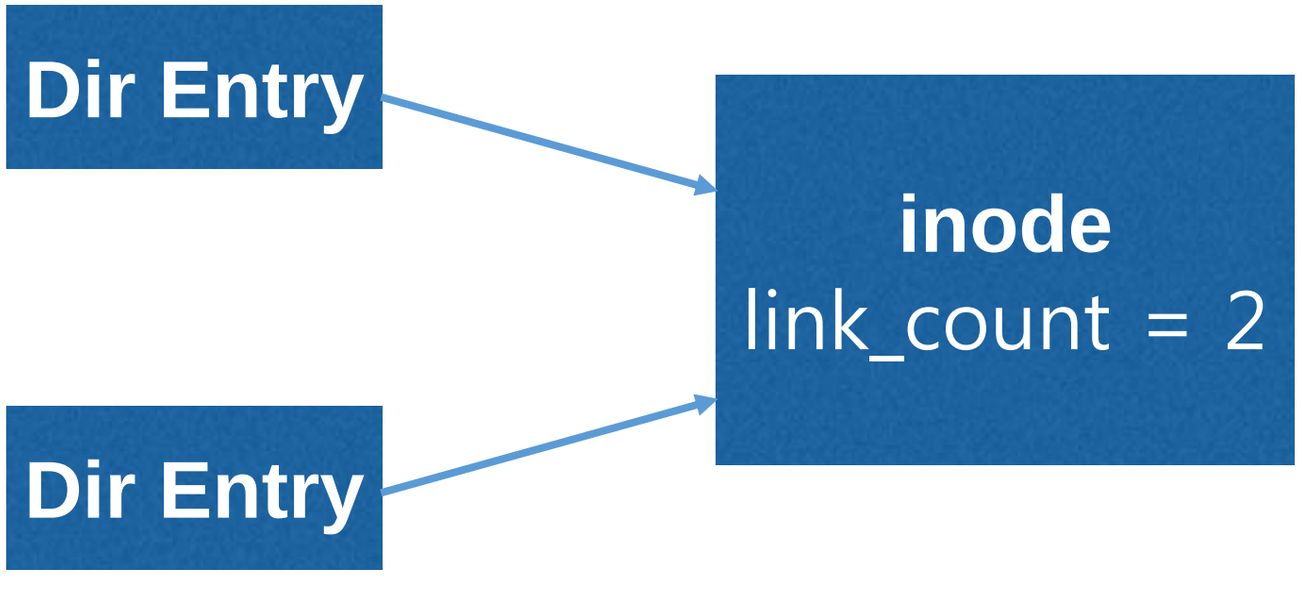
Hundreds of types of checks over different fields…
Do superblocks match?
Do directories contain “.” and “..”?
Do number of dir entries equal **inode link counts**?
Do different inodes ever point to **same block**?
…
How to solve problems?

Link Count (example 1)



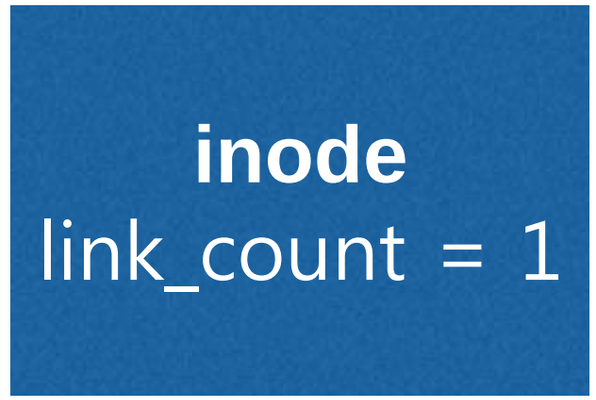
How to fix to have consistent file system?

Link Count (example 1)

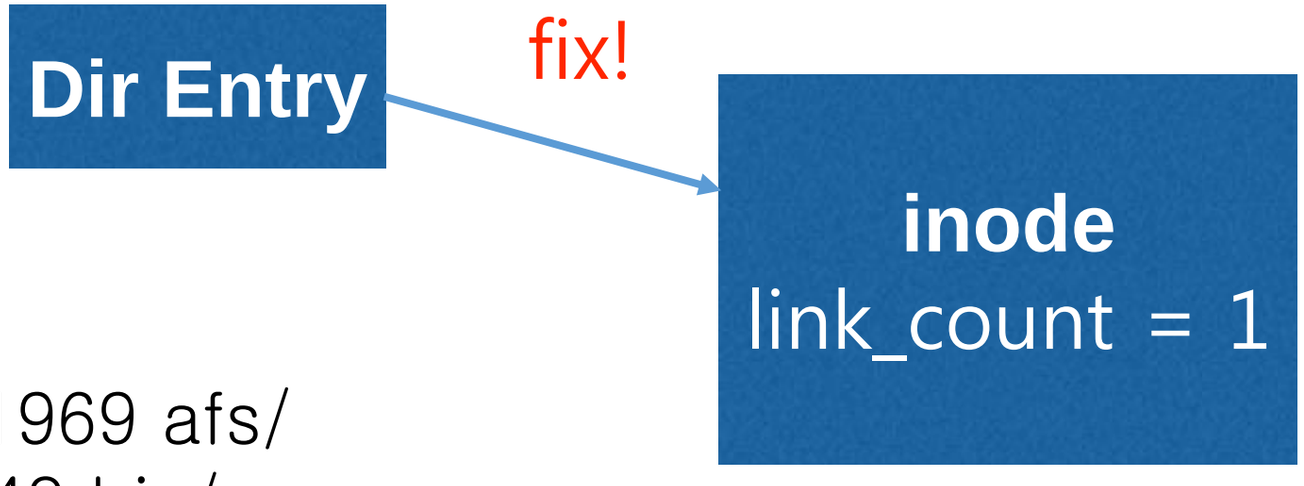


Simple fix!

Link Count (example 2)

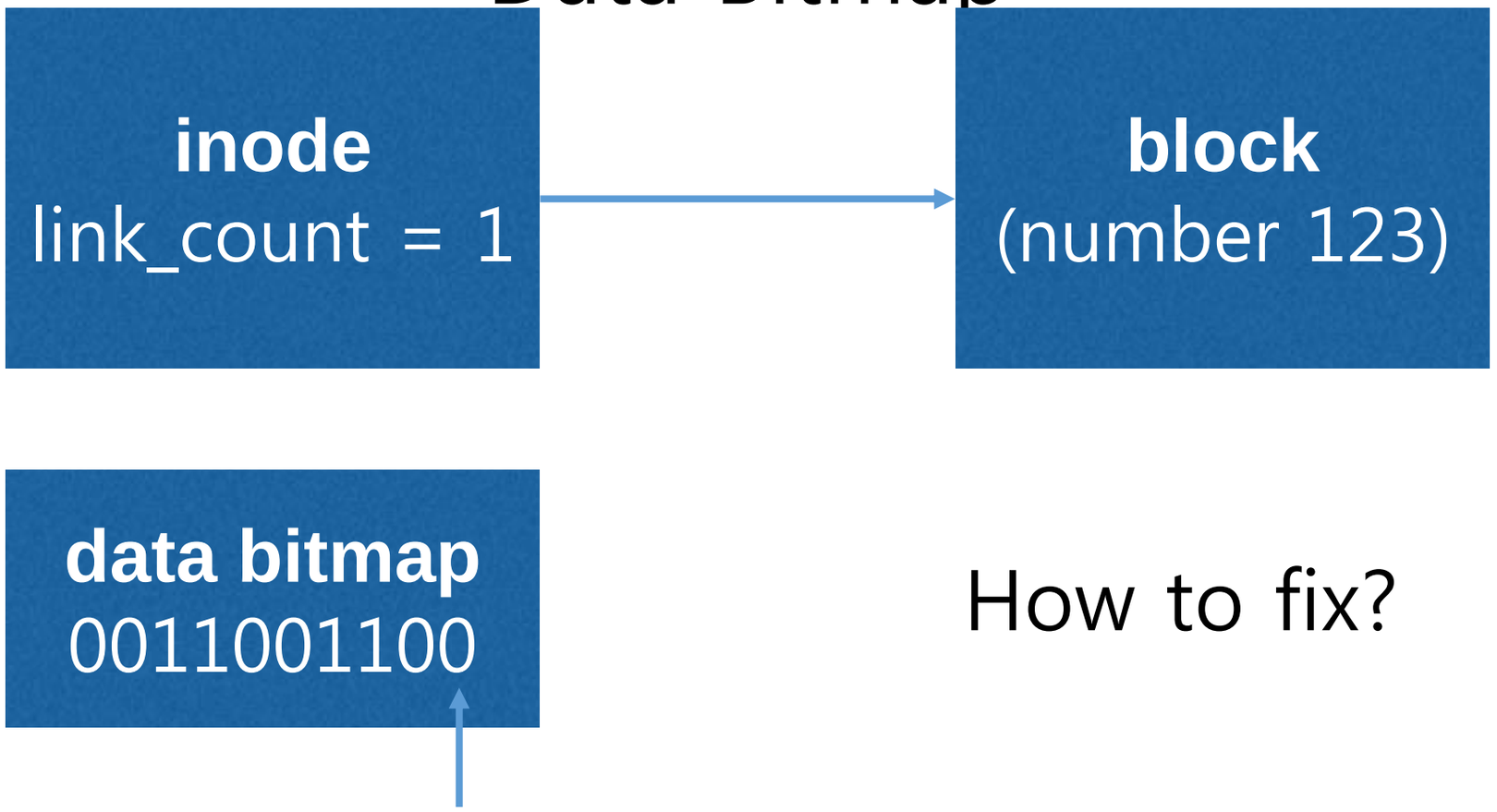


How to fix???




```
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

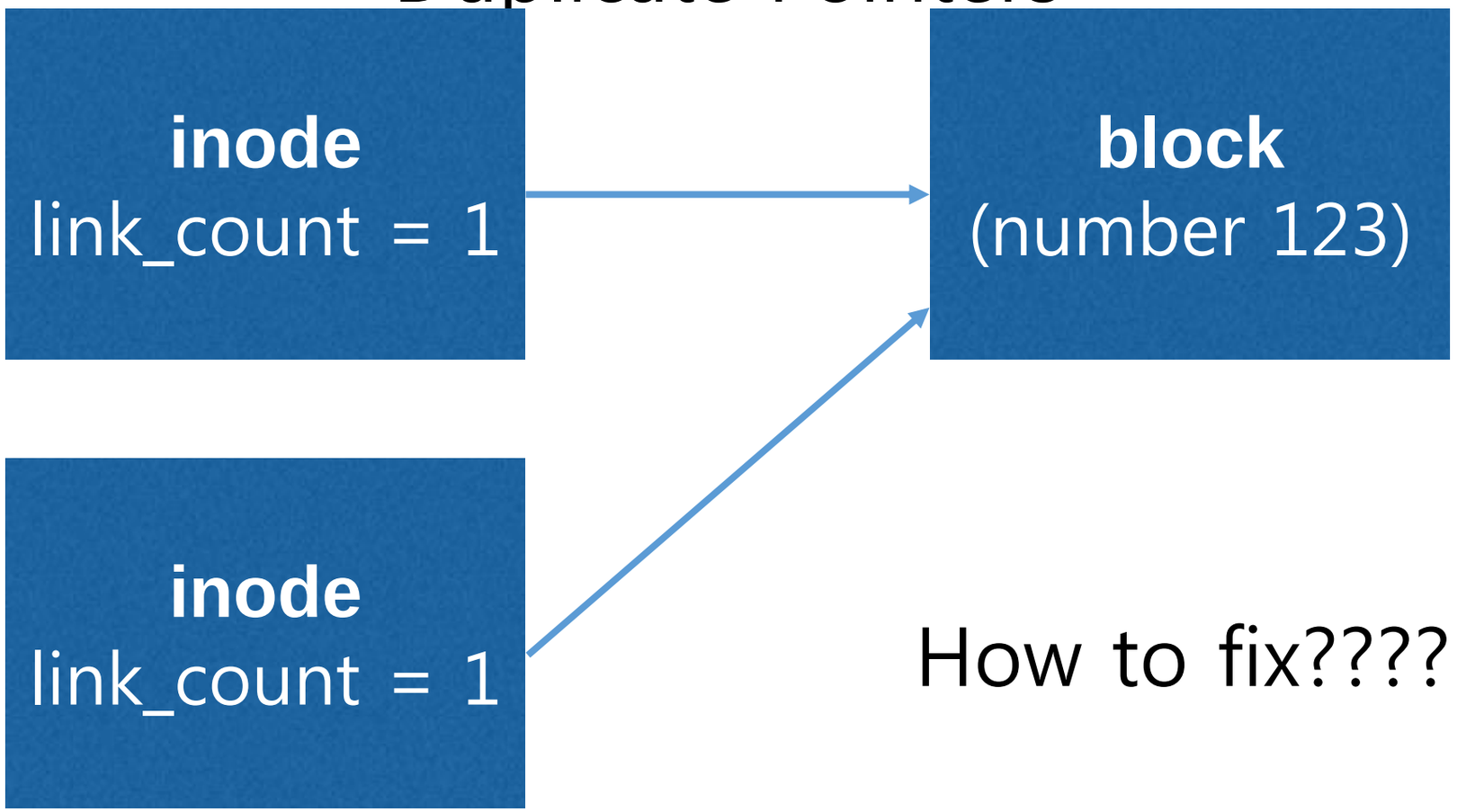
block
(number 123)

data bitmap
0011001101

Simple fix!

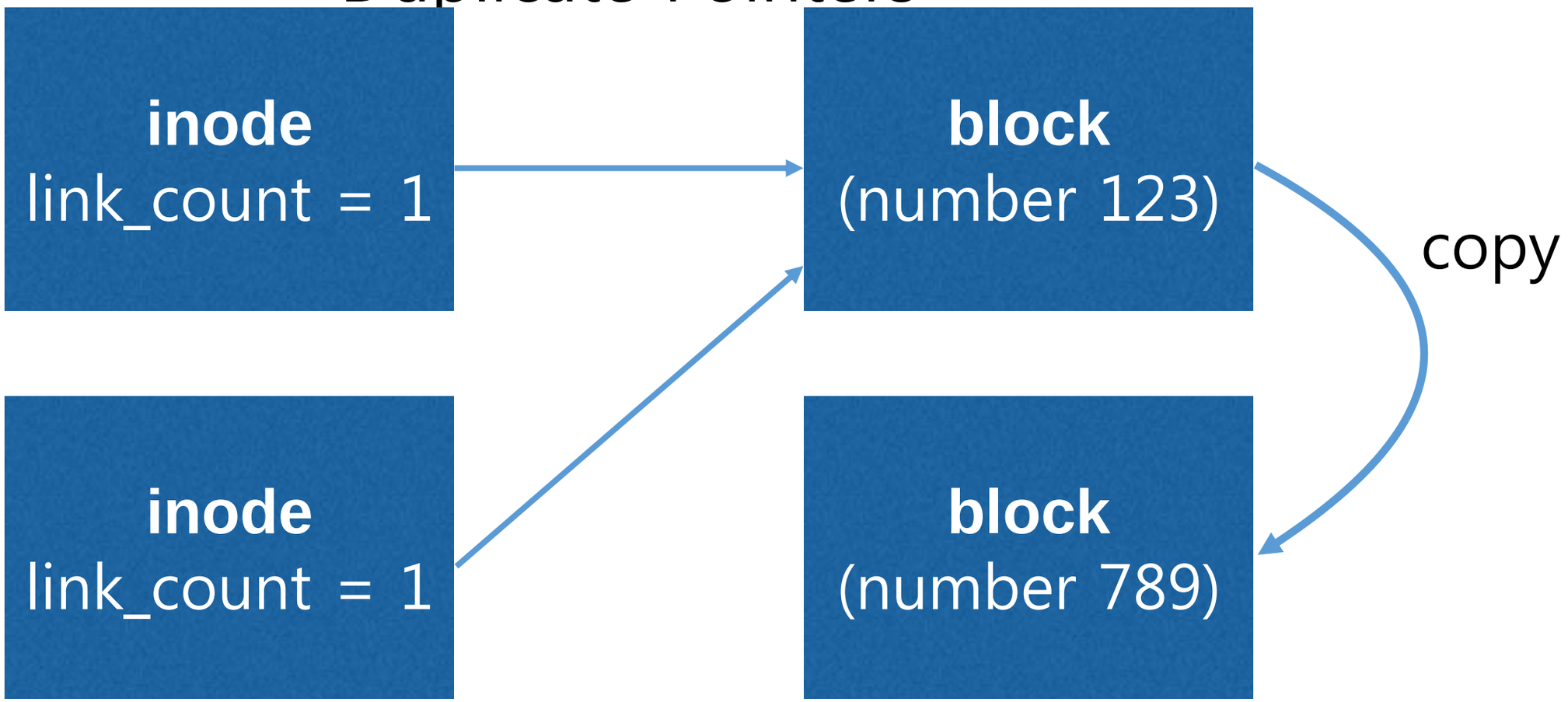
for block 123

Duplicate Pointers

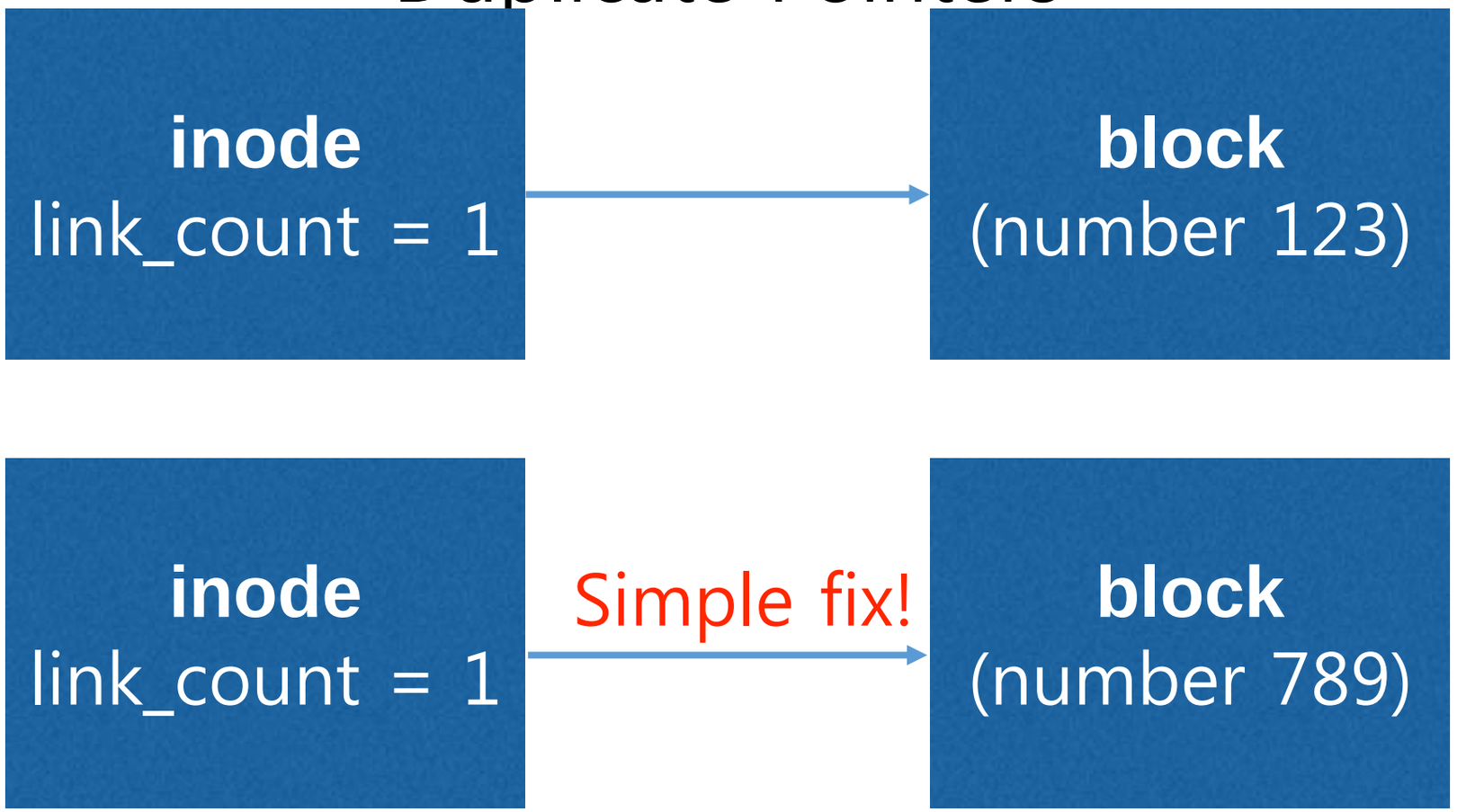


How to fix????

Duplicate Pointers

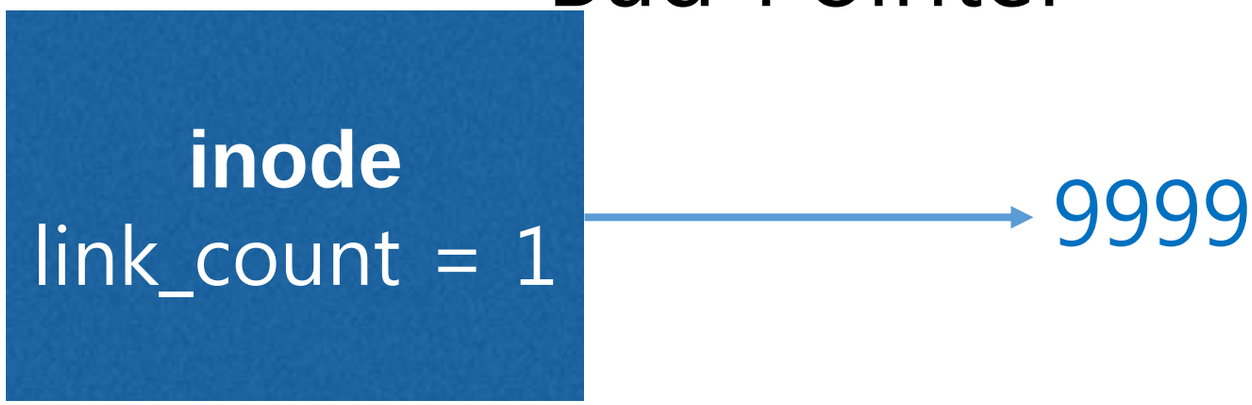


Duplicate Pointers



But is this correct?

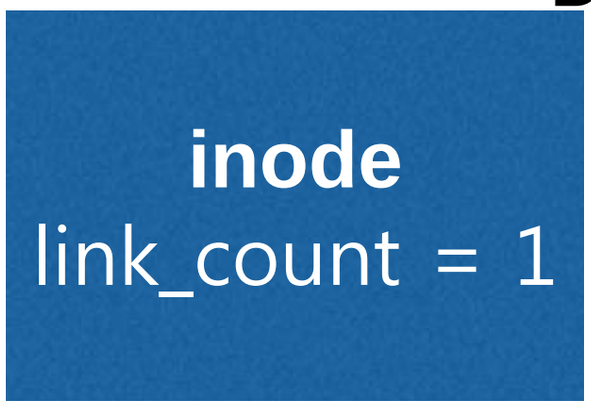
Bad Pointer



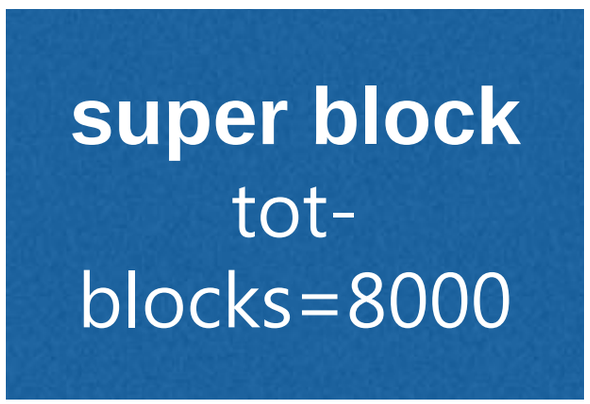
super block
tot-blocks=8000

How to fix???

Bad Pointer



Simple fix! (But is this correct?)

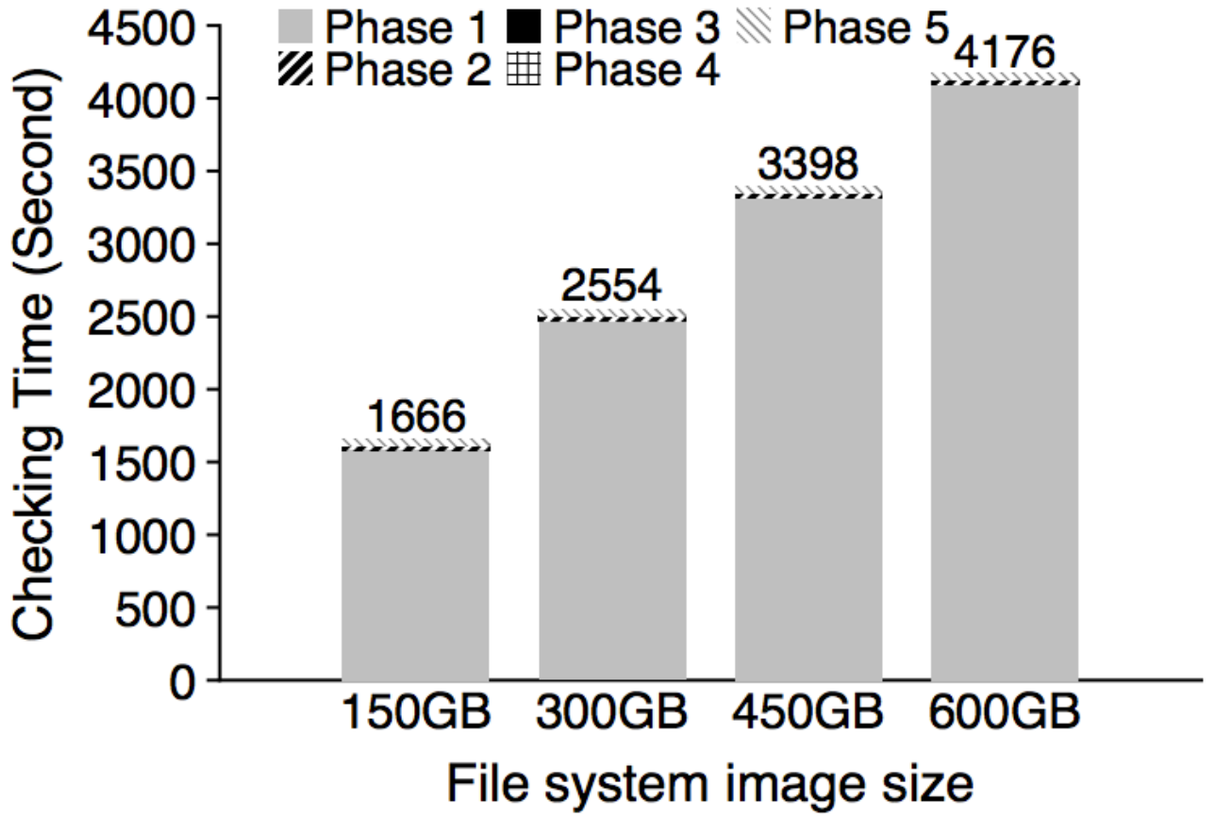


Problems with fsck

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

Solution #2: Journaling

Journaling (1)

- **Journaling** (**Write-Ahead Logging**)
  - When updating the disk, before over writing the structures in place, first write down a little note describing what you are about to do
  - Writing this note is the “write ahead” part, and we write it to a structure that we organize as a “log”
  - By writing the note to disk, you are guaranteeing that if a crash takes places during the update of the structures you are updating, you can go back and look at the note you made and try again
  - Thus, you will know exactly what to fix after a crash, instead of having to scan the entire disk

Journaling (Cont.)

- We'll describe how Linux ext3 incorporates journaling into the file system.
  - Most of the on-disk structures are identical to Linux ext2
  - The new key structure is the journal itself
  - It occupies some small amount of space within the partition or on another device

Fig.1 Ext2 File system structure

Fig.2 Ext3 File system structure

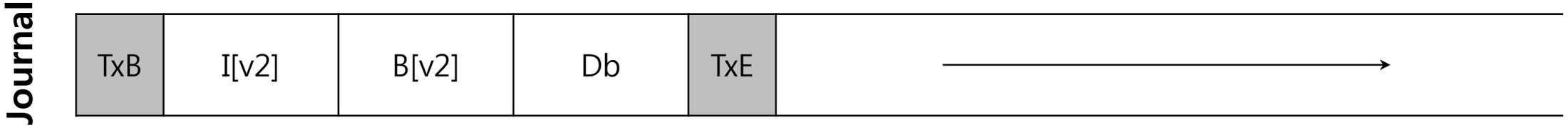
Data Journaling (1)

- Data journaling is available as a mode with the ext3 file system

- Example : our canonical update again
  - We wish to update inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk
  - Before writing them to their final disk locations, we are now first going to write them to the log(a.k.a. journal)

Data Journaling (2)

- Example : our canonical update again (Cont.)



- TxB: Transaction begin block
  - It contains some kind of **transaction identifier(TID)**
- The middle three blocks just contain the exact content of the blocks themselves
  - This is known as **physical logging**
- TxE: Transaction end block
  - Marker of the end of this transaction
  - It also contain the TID

Data Journaling (3)

- **Checkpoint**
  - Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system
  - This process is called **checkpointing**
  - Thus, to checkpoint the file system, we issue the writes I[v2], B[v2], and Db to their disk locations

Data Journaling (4)

- Our initial sequence of operations:
  1. **Journal write**
     - Write the transaction to the log and wait for these writes to complete
     - TxB, all pending data, metadata updates, TxE
  2. **Checkpoint**
     - Write the pending metadata and data updates to their final locations

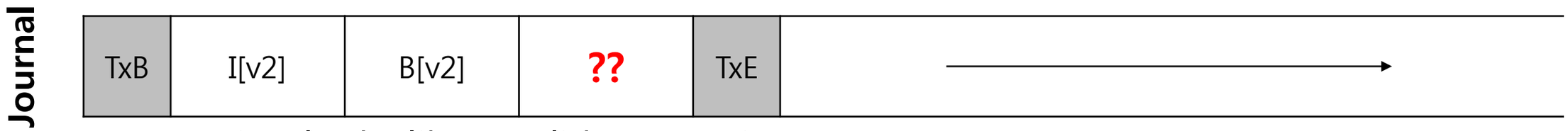
Data Journaling (5)

- When a crash occurs during the writes to the journal
  1. **Transaction each one at a time**
     - 5 transactions (TxB, I[v2], B[v2], Dnb, TxE)
     - This is slow because of waiting for each to complete

  2. **Transaction all block writes at once**
     - Five writes -> a single sequential write : Faster way
     - However, this is unsafe
       - Given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order

Data Journaling (6)

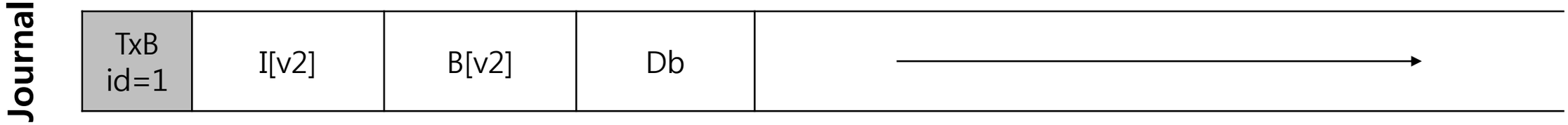
- When a crash occurs during the writes to the journal (Cont.)
  2. **Transaction all block writes at once (Cont.)**
     - Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db
     - Unfortunately, if the disk loses power between (1) and (2)



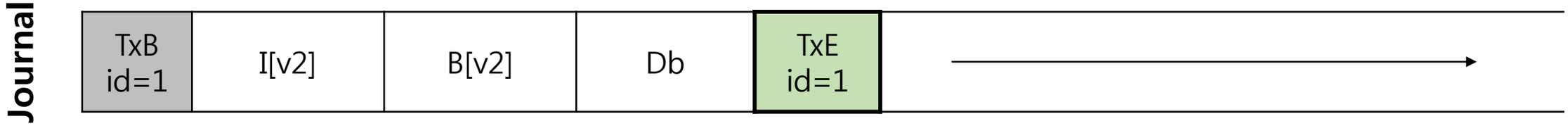
- Transaction looks like a valid transaction.
  - Further, the file system can't look at that forth block and know it is wrong.
  - It is much worse if it happens to a critical piece of file system, such as superblock.

Data Journaling (7)

- When a crash occurs during the writes to the journal (Cont.)
  2. **Transaction all block writes at once (Cont.)**
     - To avoid this problem, the file system issues the transactional write in two steps
     - First, writes all blokes **except the TxE block** to journal



- Second, The file system issues the write of the TxE



- An important aspect of this process is the atomicity guarantee provided by the disk.
  - The disk guarantees that any 512-byte write either happen or not
  - Thus, TxE should be a single 512-byte block

Data Journaling (8)

- When a crash occurs during the writes to the journal (Cont.)
  2. **Transaction all block writes at once (Cont.)**
     - Thus, our current protocol to update the file system, with each of its three phases labeled:
       1. Journal write : write the contents of the transaction to the log
       2. **Journal commit (added) :** write the transaction commit block
       3. Checkpoint **:** write the contents of the update to their locations

Data Journaling (end)

- Recovery
  - If the crash happens **before the transactions** is written to the log
    - The pending update is **skipped**
  - If the crash happens **after the transactions** is written to the log, but **before the checkpoint**
  - **Recover** the update as follow:
    - Scan the log and lock for transactions that have committed to the disk
    - Transactions are replayed

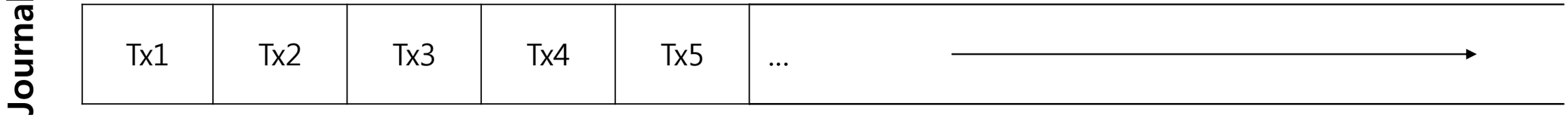
Batching Log Updates

- If we create two files in same directory, same inode, directory entry block is to the log and committed twice.

- To reduce excessive write traffic to disk, journaling manage the **global transaction**.
  - Write the content of the global transaction forced by synchronous request.
  - Write the content of the global transaction after timeout of 5 seconds.

Making The log Finite (1)

- The log is of a finite size
  - Two problems arise when the log becomes full



1. The larger the log, the longer recovery will take
   - Simpler but less critical
2. No further transactions can be committed to the disk
   - Thus making the file system “less than useful”

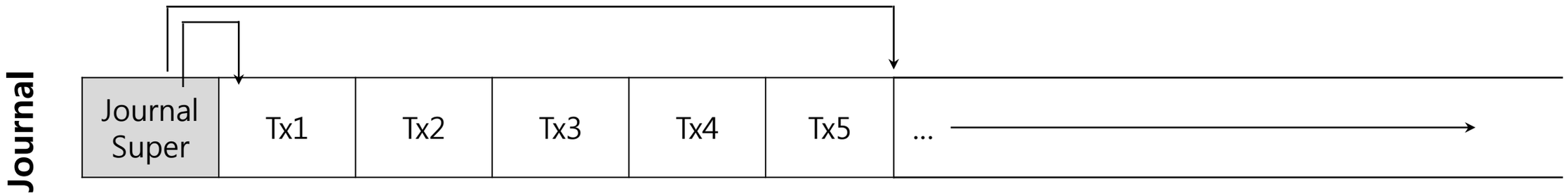
Making The log Finite (2)

- To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over
  - This is why the journal is referred to as a circular log.

- To do so, the file system must take action some time after a checkpoint
  - Specifically, once a transaction has been checkpointed, the file system should free the space

Making The log Finite (3)

- journal super block
  - Mark the oldest and newest transactions in the log.
  - The journaling system records which transactions have not been check pointed.



Making The log Finite (end)

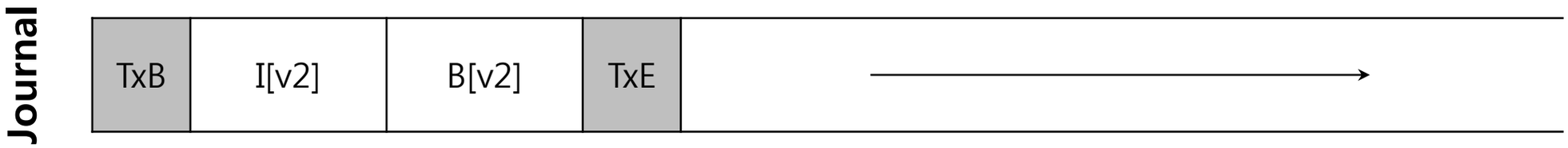
- journal super block (Cont.)
  - Thus, we add another step to our basic protocol
    1. Journal write
    2. Journal commit
    3. checkpoint
    4. **Free**
       - Some time later, mark the transaction free in the journal by updating the journal Superblock

Metadata Journaling (1)

- There is a still problem : writing every data block to disk **twice**
  - Commit to log (journal file)
  - Checkpoint to on-disk location.
- People have tried a few different things in order to speed up performance.
  - Example : A simpler form of journaling is called **ordered journaling (metadata journaling)**
    - **User data is not written to the journal**

Metadata Journaling (2)

- Thus, the following information would be written to the journal



- The data block Db, previously written to the log, would instead be written to the file system proper

Metadata Journaling (3)

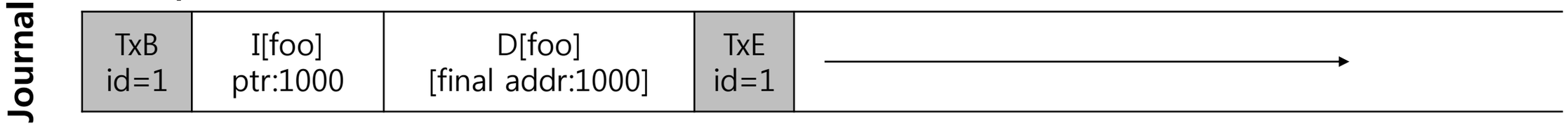
- The modification does raise an interesting question: **when should we write data blocks to disk?**
- Let's consider an example
  1. Write Data to disk after the transaction
     - Unfortunately, this approach has a problem
     - The file system is consistent but I[v2] may end up pointing to garbage data
  2. Write Data to disk before the transaction
     - It ensures the problems

Metadata Journaling (end)

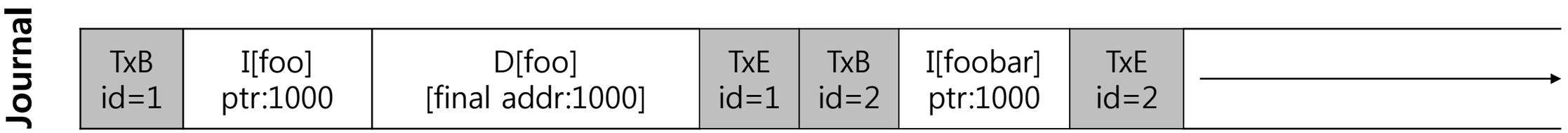
- Specifically, the protocol is as follows:
  1. **Data Write(added)**: Write data to final location
  2. **Journal metadata write(added)**: Write the begin and metadata to the log
  3. Journal commit
  4. Checkpoint metadata
  5. Free

Tricky case: Block Reuse (1)

- Some metadatas should not be replayed.
- Example

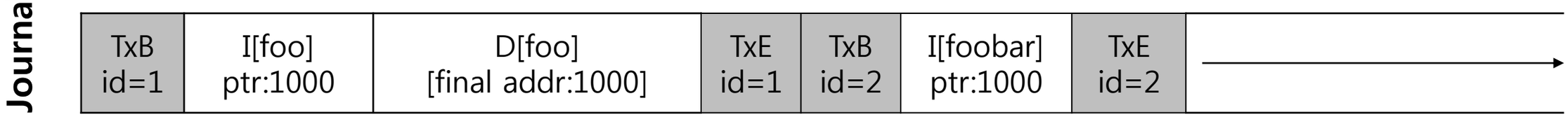


2. Directory “foo” id deleted. block 1000 is freed up for reuse
3. User Create a file “foobar”, reusing block 1000 for data



Tricky case: Block Reuse (2)

4. Now assume a crash occurs and all of this information is still in the log.



5. During replay, the recovery process replays everything in the log
   - Including the write of directory data in block 1000
6. The replay thus overwrites the user data of current file `foobar` with old directory contents

Tricky case: Block Reuse (2)

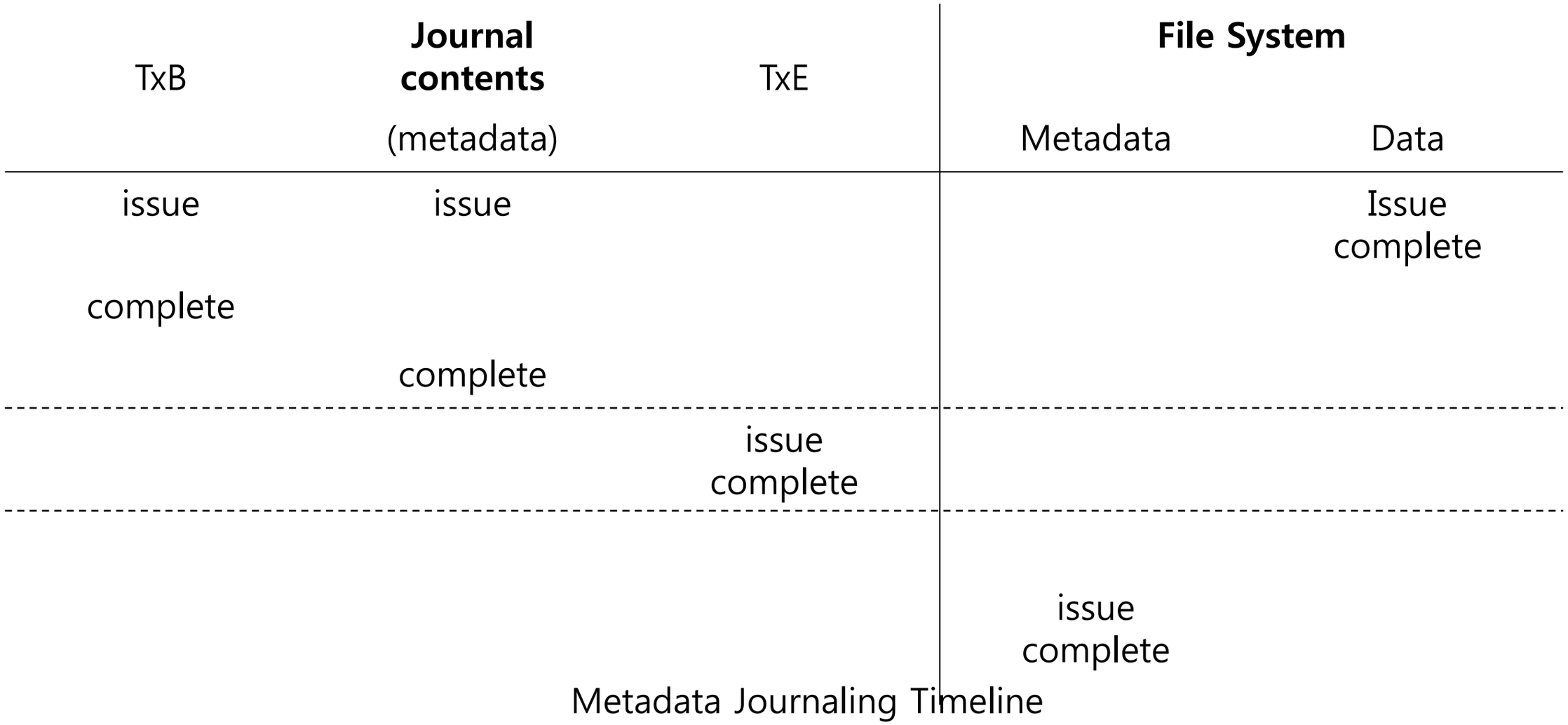
- Solution
  - What Linux ext3 does instead is to add a **new type** of record to the journal, Known as a **revoke** record
  - When replaying the journal, the system first scans for such revoke records
  - Any such revoked data is never replayed

Data Journaling Timeline

TxB	Journal contents (metadata)	Journal contents (data)	TxE	File System Metadata	File System Data
issue	issue	issue			
complete					
	complete				
		complete	issue complete		
				issue	issue complete
				complete	

Data Journaling Timeline

Metadata Journaling Timeline



Consistency Solution #2: Journaling

Goals

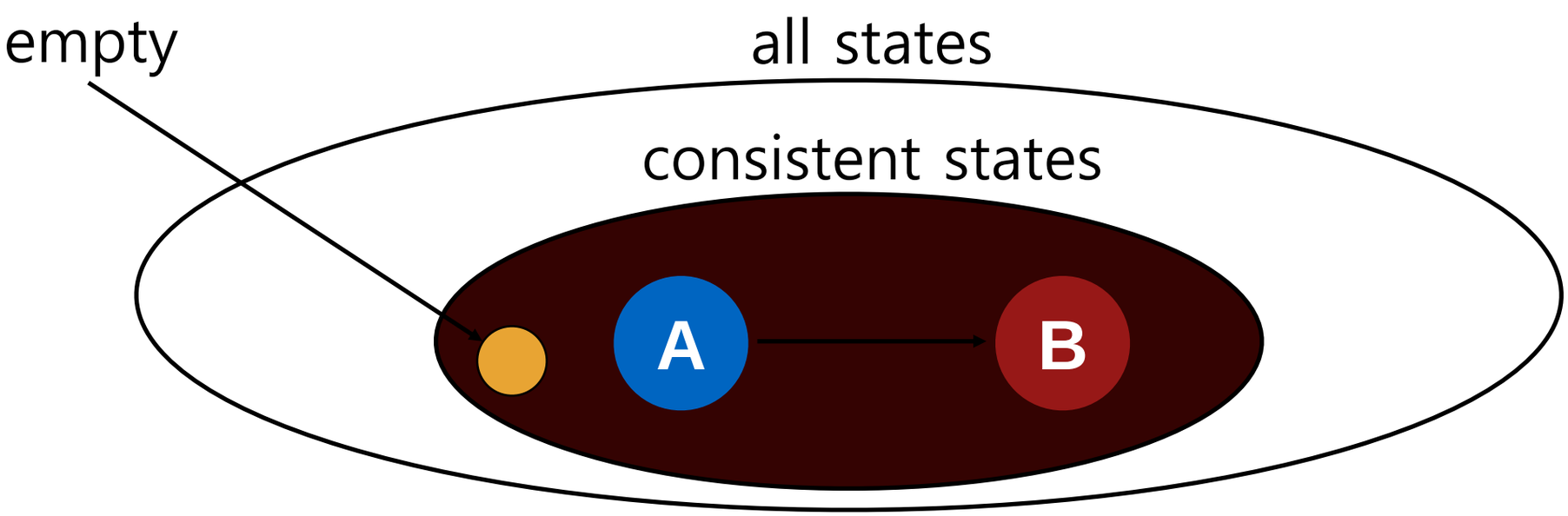
- Ok to do some **recovery work** after crash, but not to read entire disk
- Don't move file system to just any consistent state, ge t **correct** state

Strategy

- Atomicity
- Definintion of atomicity for **concurrency**
  - operations in critical sections are not interrupted by op erations 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.

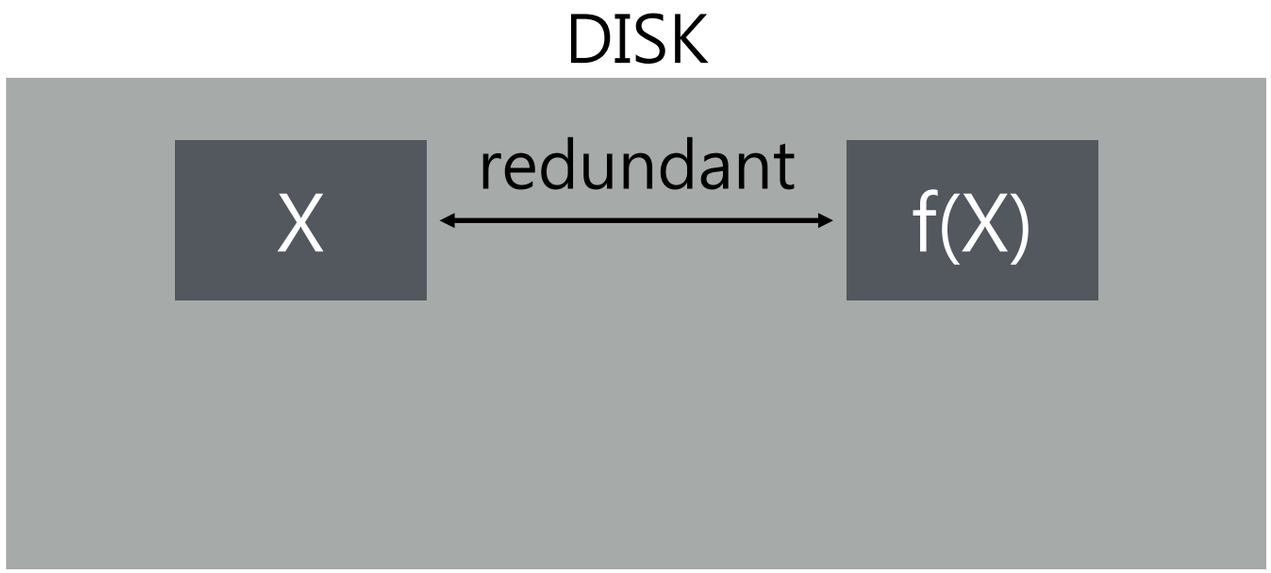
Journaling General Strategy

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

Ironically, adding redundancy to fix the problem c aused by redundancy.

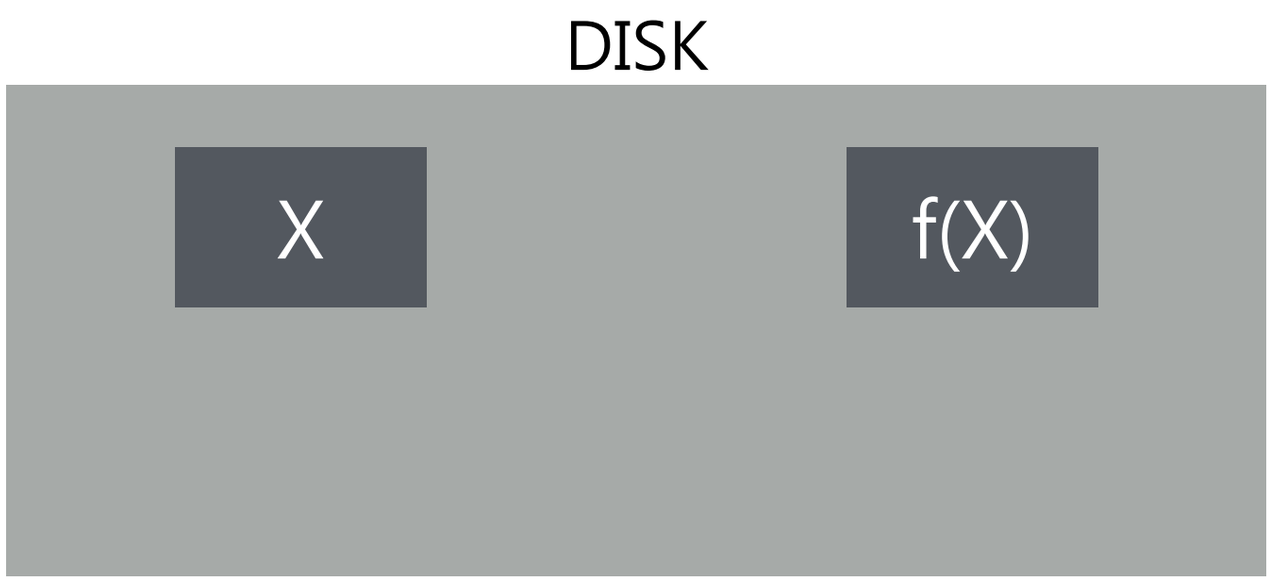
Fight Redundancy with Redundancy

Want to replace X with Y. Original:



Fight Redundancy with Redundancy

Want to replace X with Y. Original:

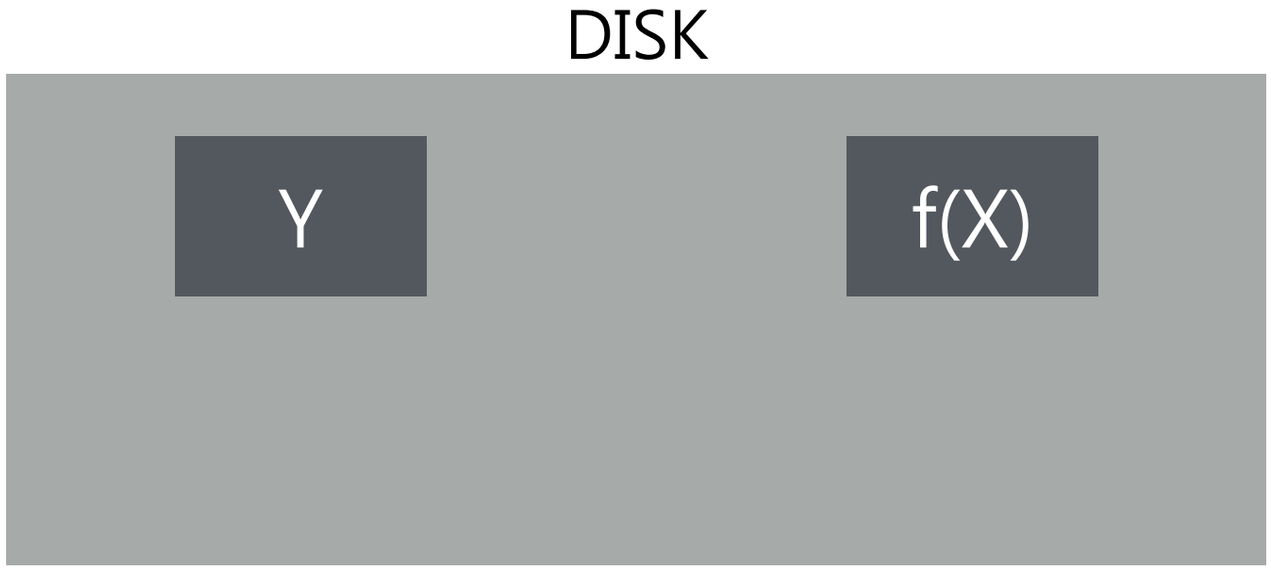


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

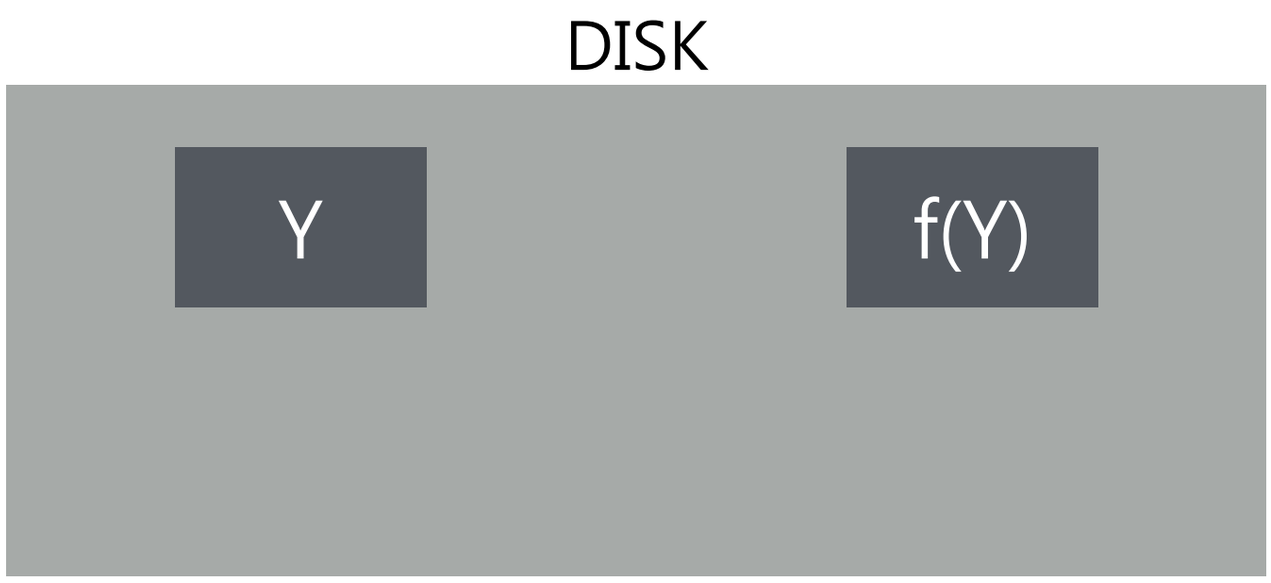


Good time to crash?

bad time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

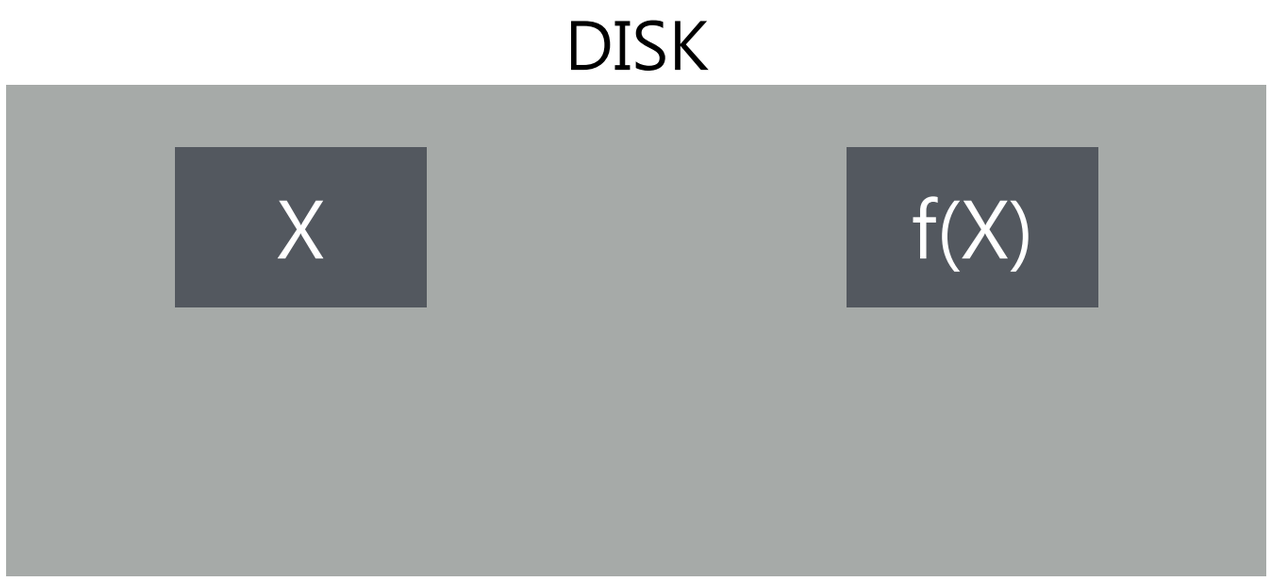


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

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

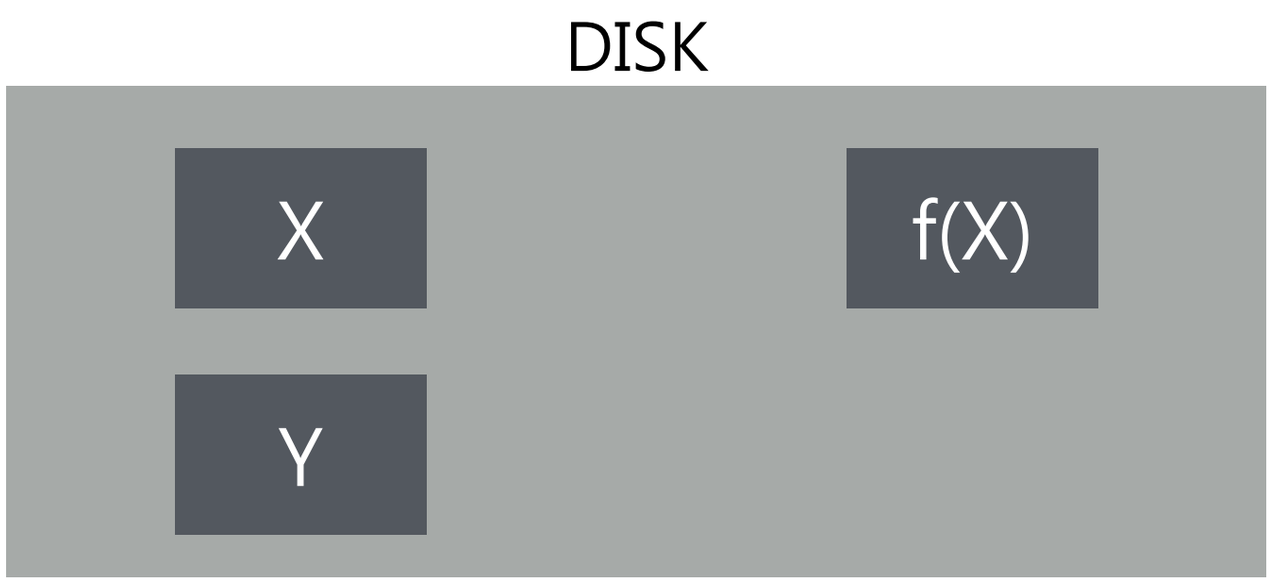


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

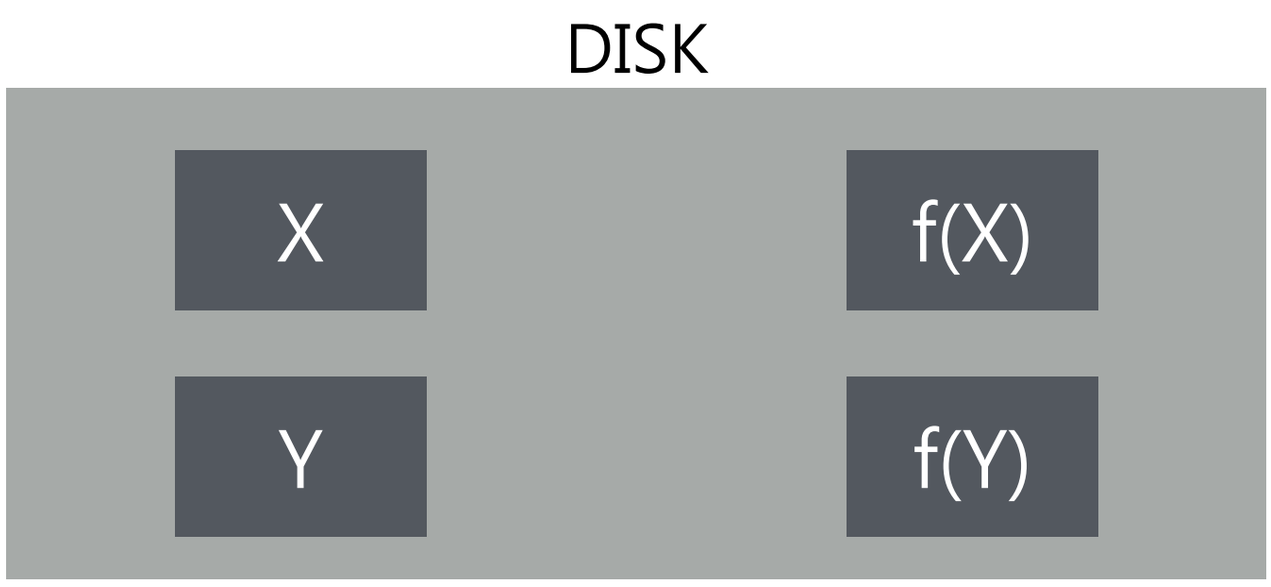
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

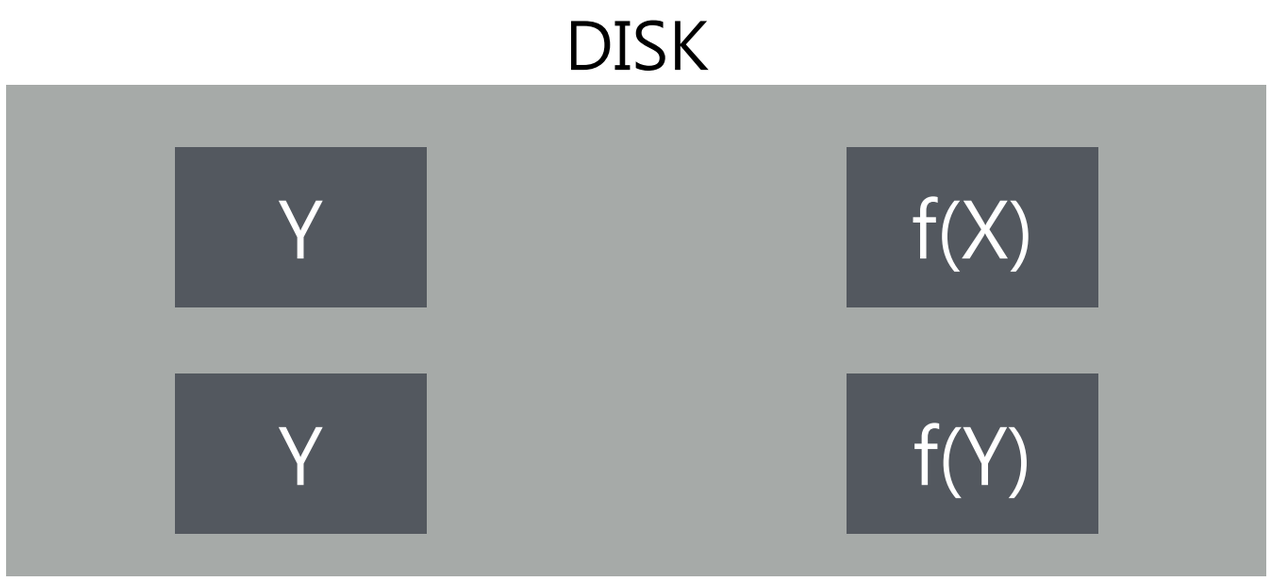
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

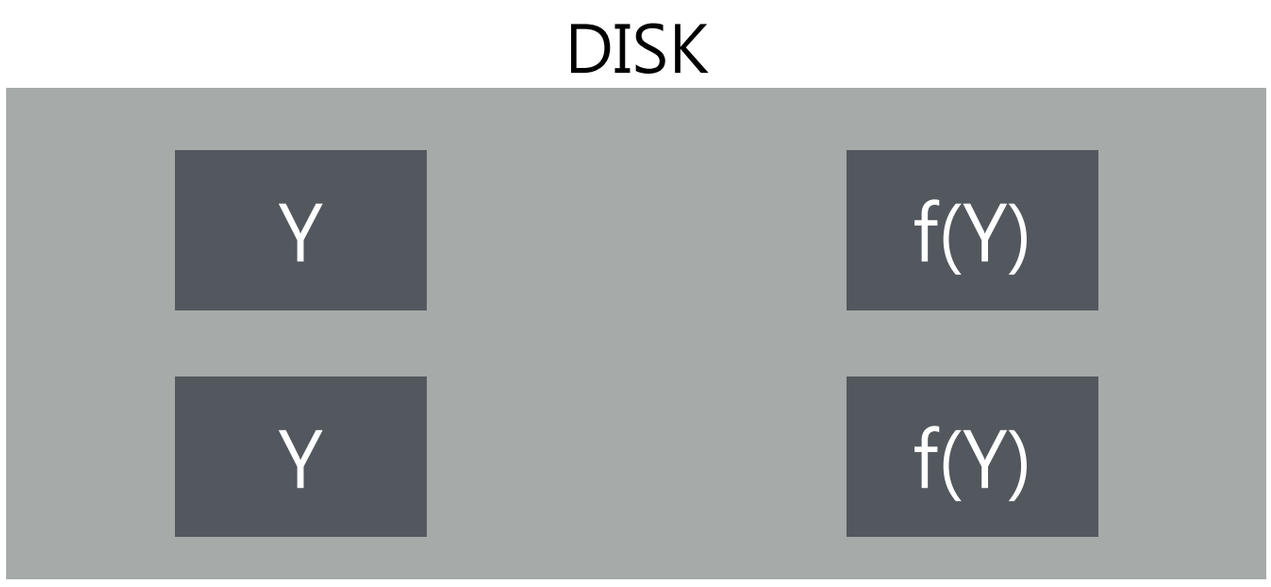
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

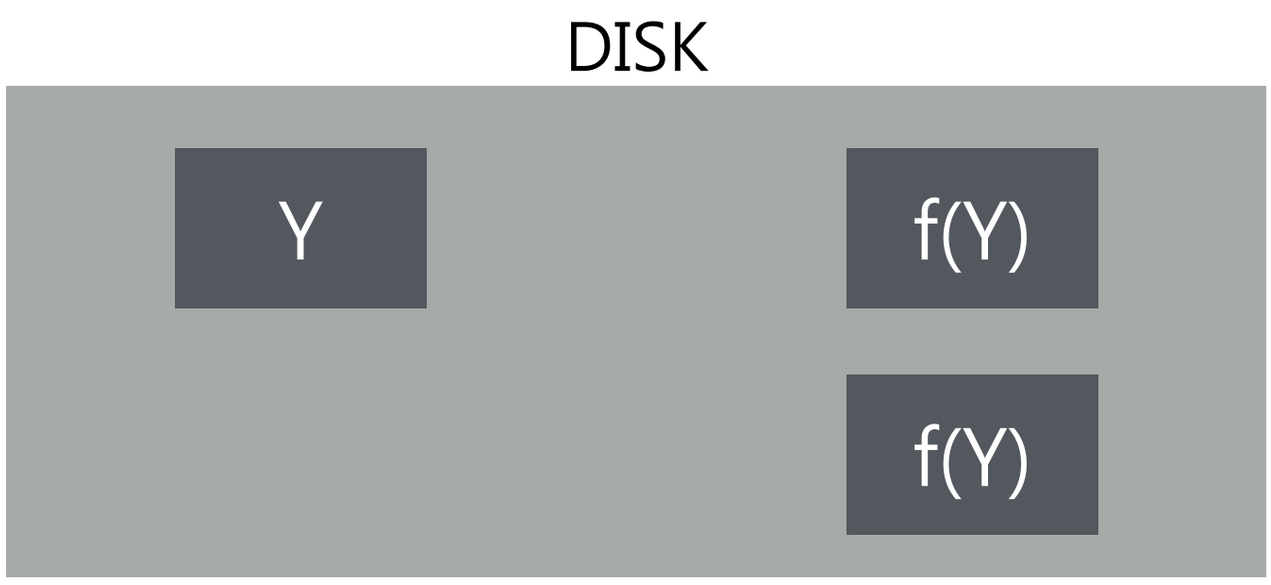
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

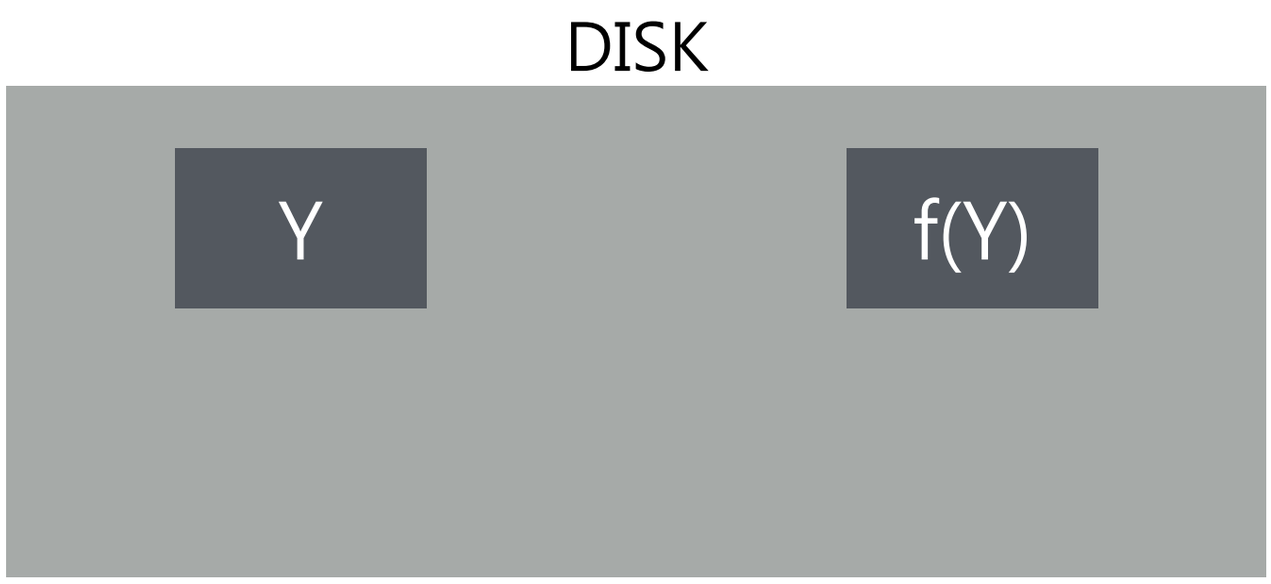
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

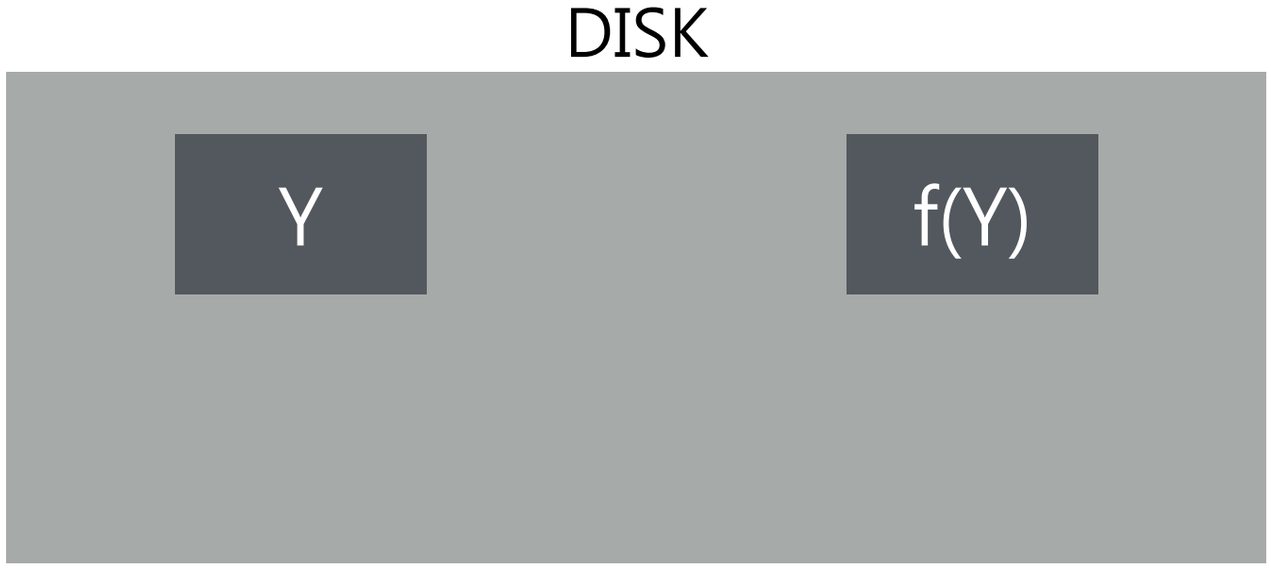
Want to replace X with Y. With journal:



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 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
     }
}
```

Journal Old Data

Time	Block 0: Alice	Block 1: Bob	extra	extra	extra
1	12	3	0	0	0
2	12	3	12	0	0
3	12	3	12	3	0
4	12	3	12	3	1
5	10	3	12	3	1
6	10	5	12	3	1
7	10	5	12	3	0

Terminology

Extra blocks are called a “journal”

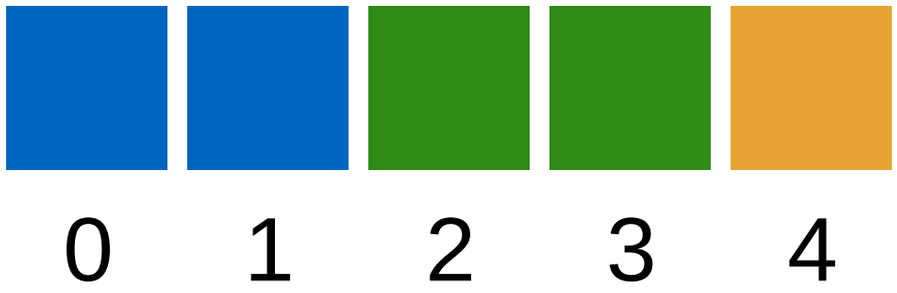
The writes to the jounral are a “journal transaction”

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

File systems typically write new data to the journal.

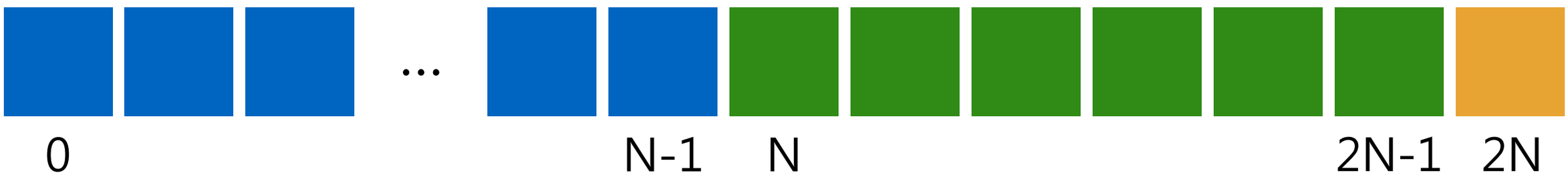
Small Disk

What if we want to use a larger disk?



Big disk

What if we want to use a larger disk?



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

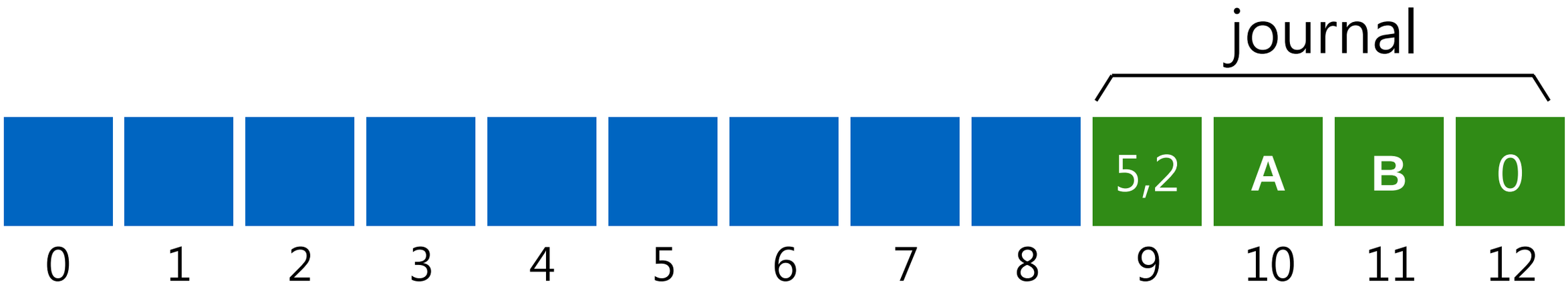
Goal:

- Reuse small area as backup for any block

How?

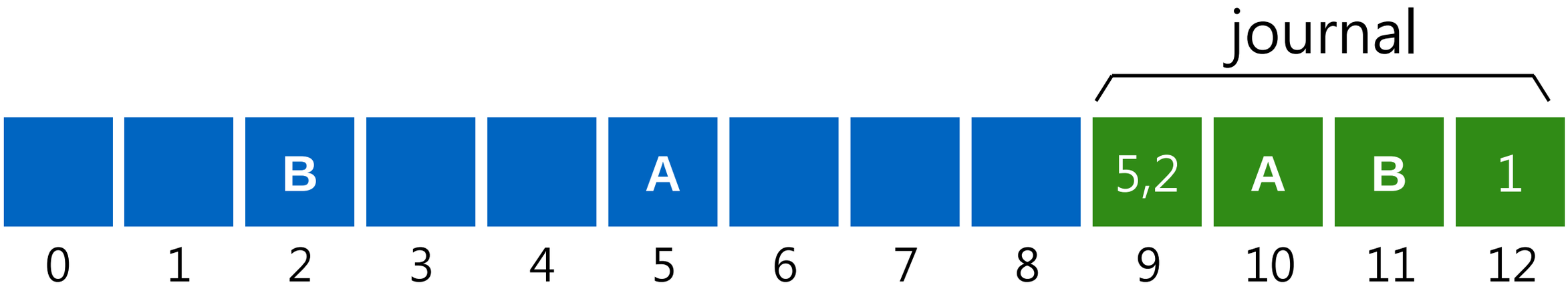
- Store block numbers in a transaction header

New Layout



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

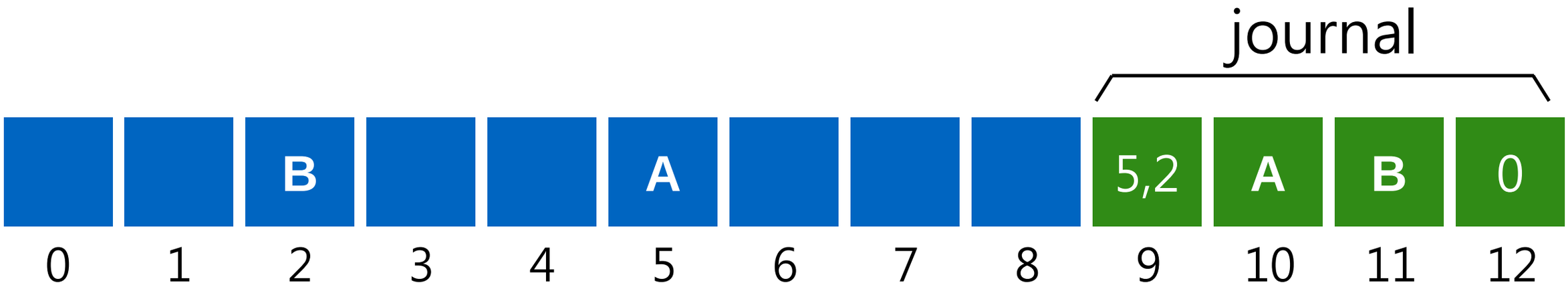
New Layout



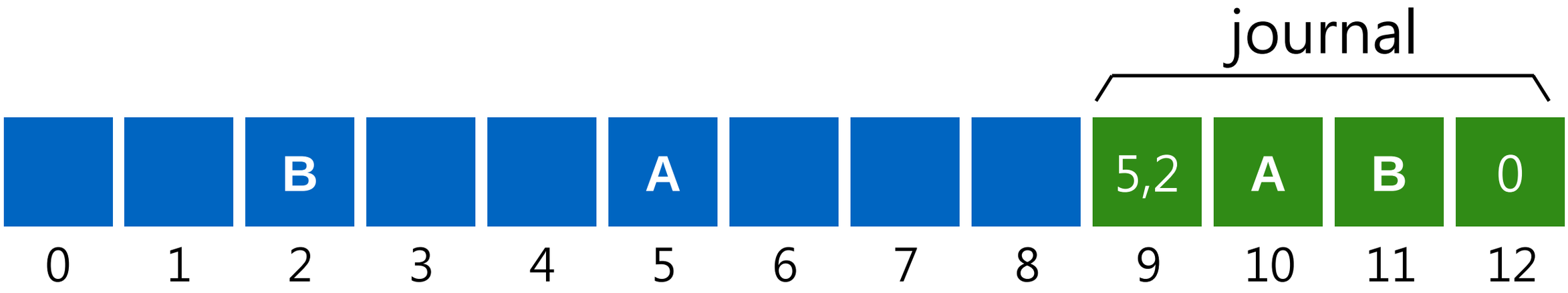
transaction: write A to block 5; write B to block 2

Checkpoint: Writing new data to in-place locations

New Layout

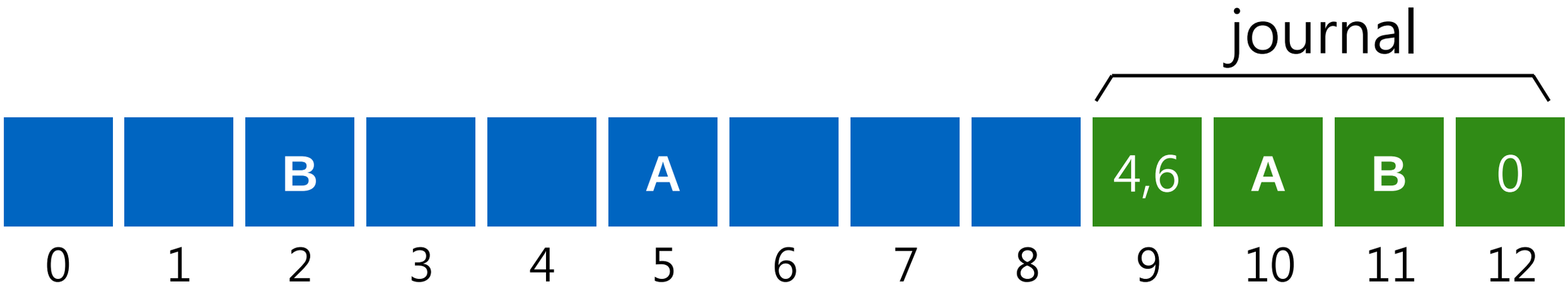


New Layout



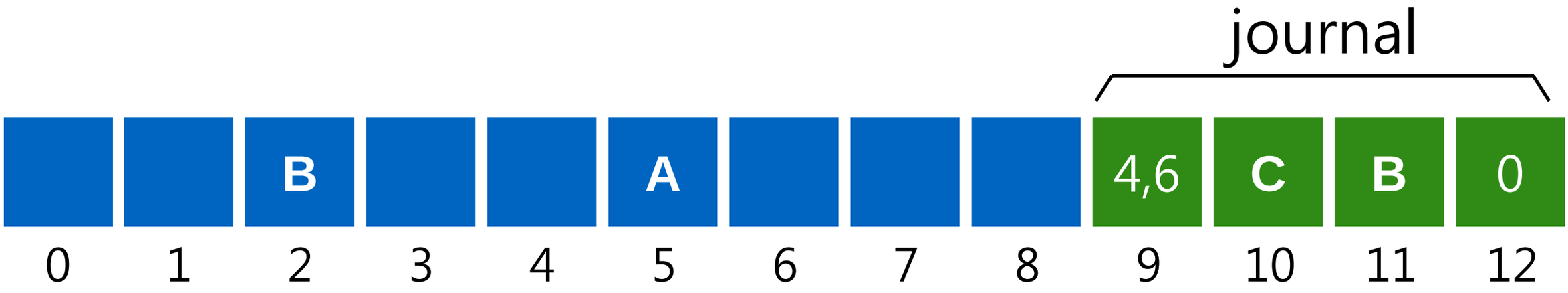
transaction: write C to block 4; write T to block 6

New Layout



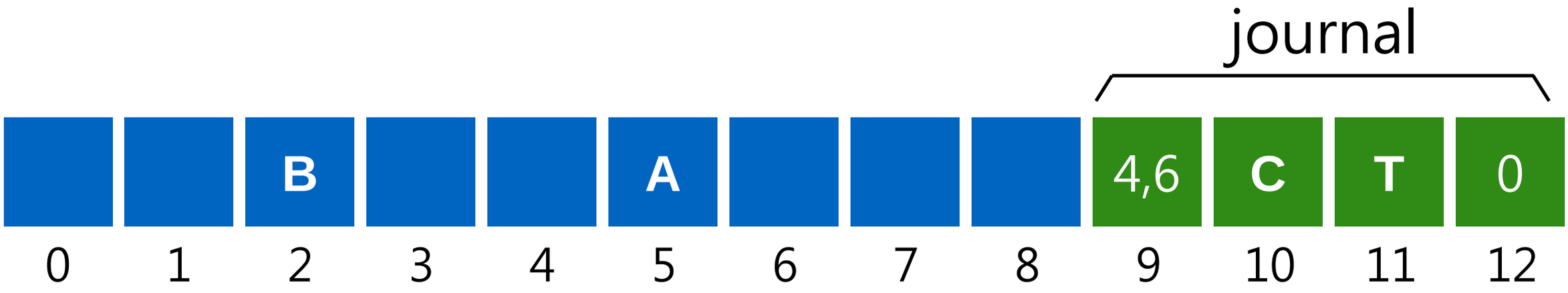
transaction: write C to block 4; write T to block 6

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!

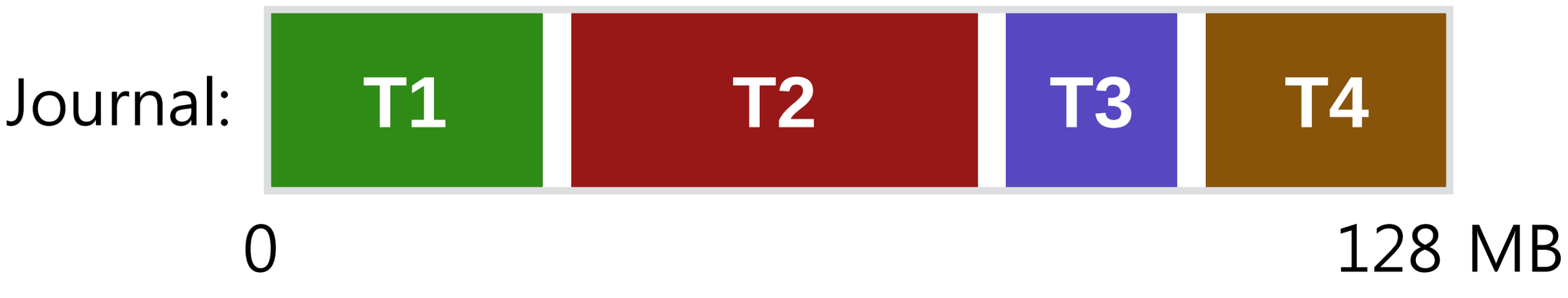
Journaling is sequential, checkpointing is random

Solution? Delay checkpointing for some time

Difficulty: need to reuse journal space

Solution: keep many transactions for un-checkpointed data

Circular Buffer

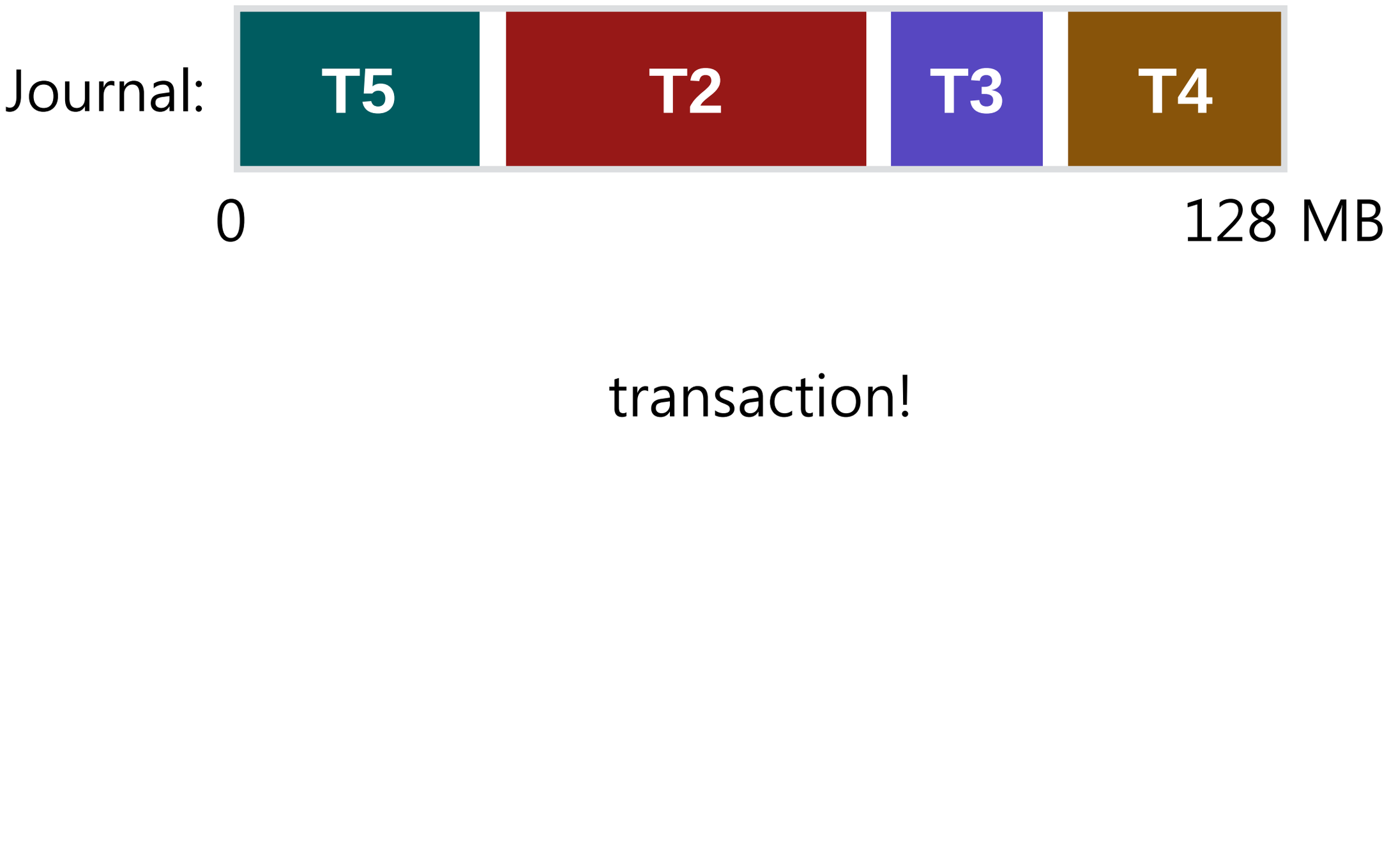


Keep data also in memory until checkpointed on disk

Circular Buffer

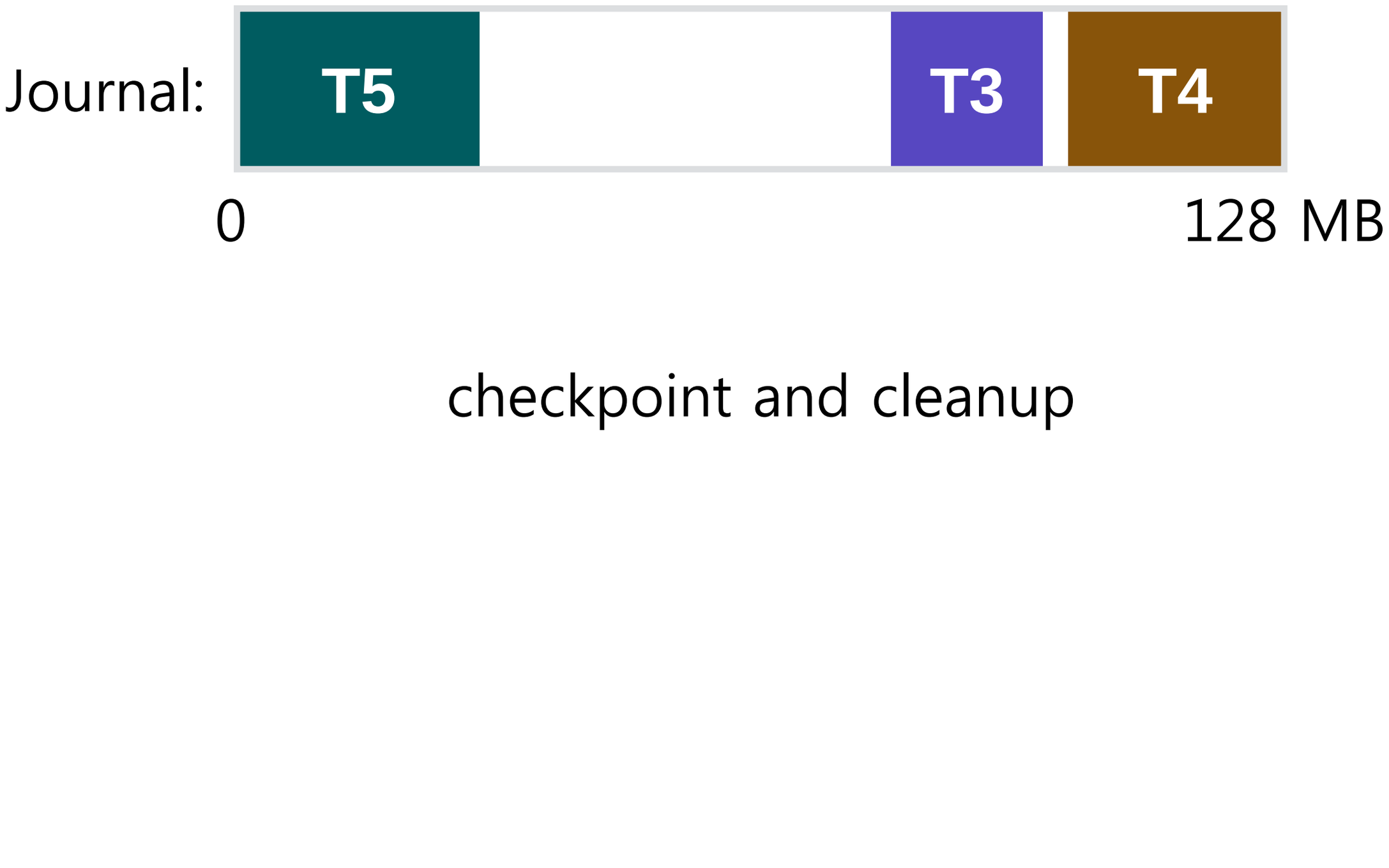
checkpoint and cleanup

Circular Buffer



transaction!

Circular Buffer

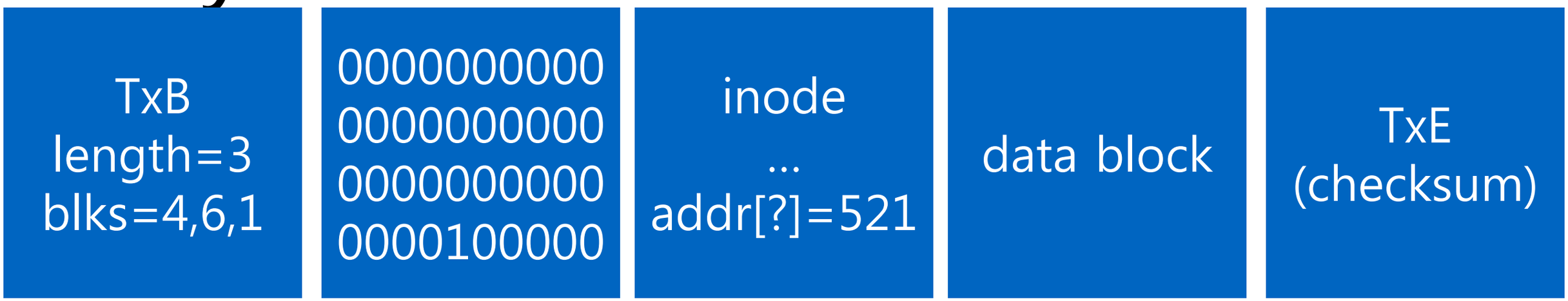


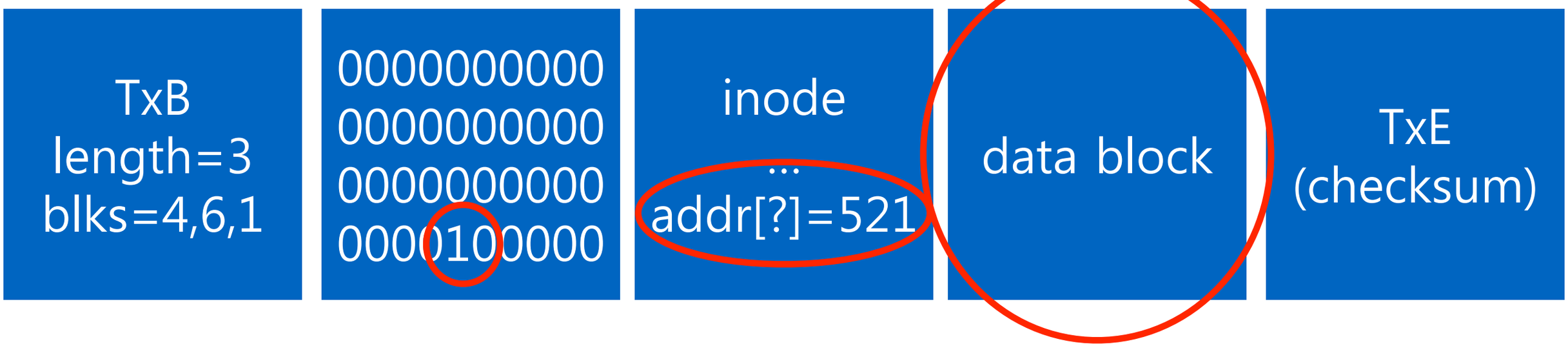
checkpoint and cleanup

Optimizations

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

Physical Journal





Actual changed data is much smaller!

Logical Journal

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

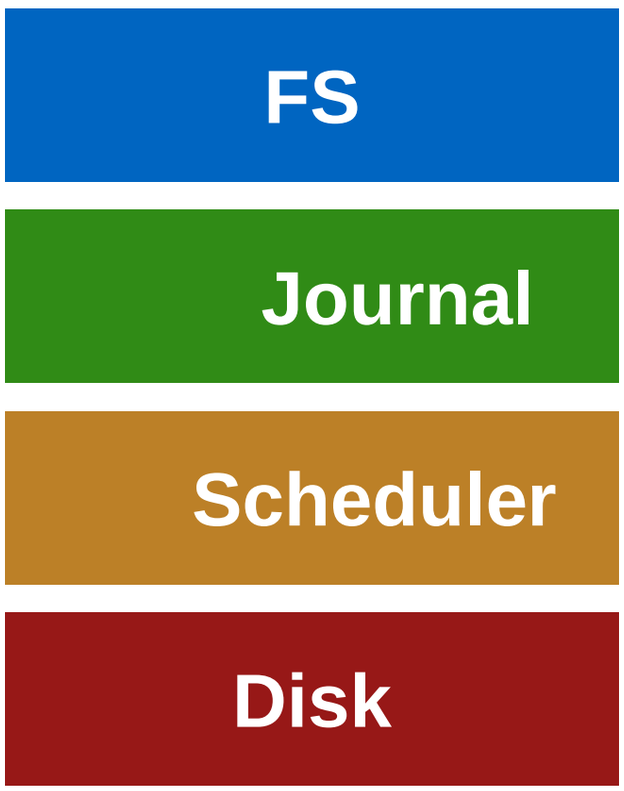
On recovery:
Need to read existing contents of in-place data and (re-)apply changes

Optimizations

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

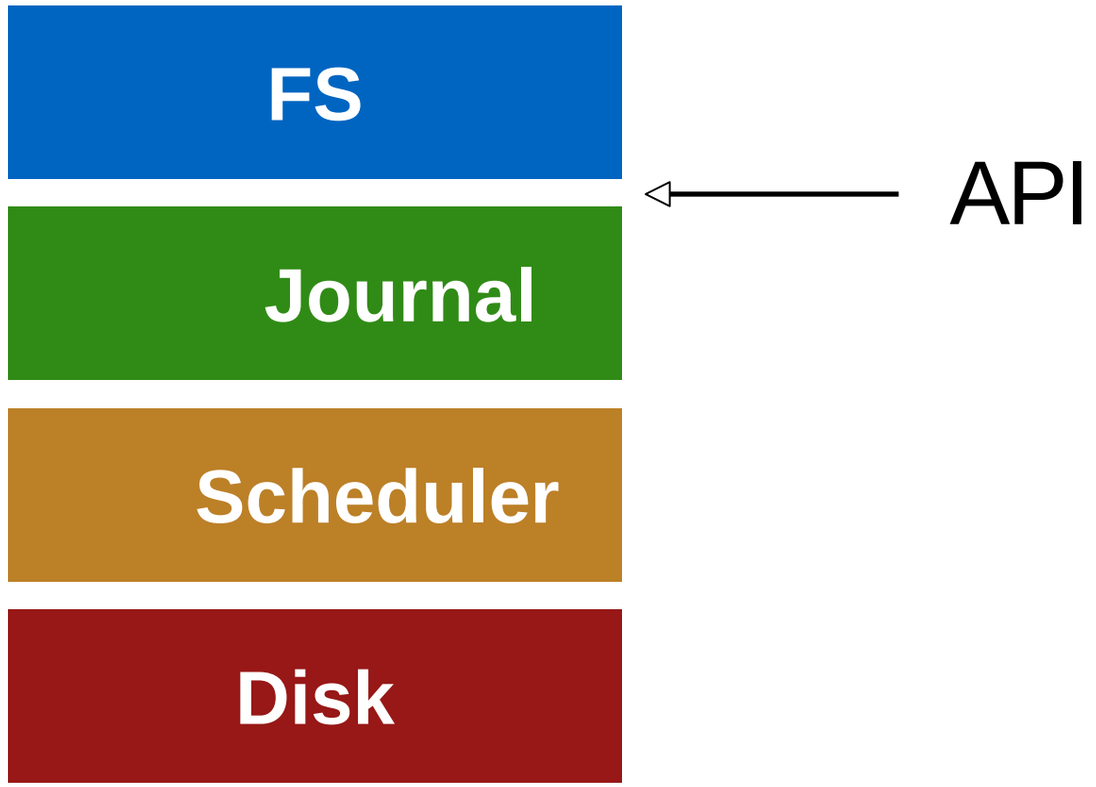
File System Integration

- How should FS use journal? Option 1:



File System Integration

- How should FS use journal? Option 1:



Journal API

With RAID we built a fast, reliable logical disk.

Can we build an atomic disk with the same API?

Journal API

With RAID we built a fast, reliable logical disk.

Can we build an atomic disk with the same API?

Standard block calls:

writeBlk()

readBlk()

flush()

Journal API

With RAID we built a fast, reliable logical disk.

Can we build an atomic disk with the same API?

Standard block calls:

writeBlk()
readBlk()
flush()

which calls must be atomic?

Handle API

```
h = getHandle();
writeBlk(h, blknum, data);
finishHandle(h);
```

Handle API

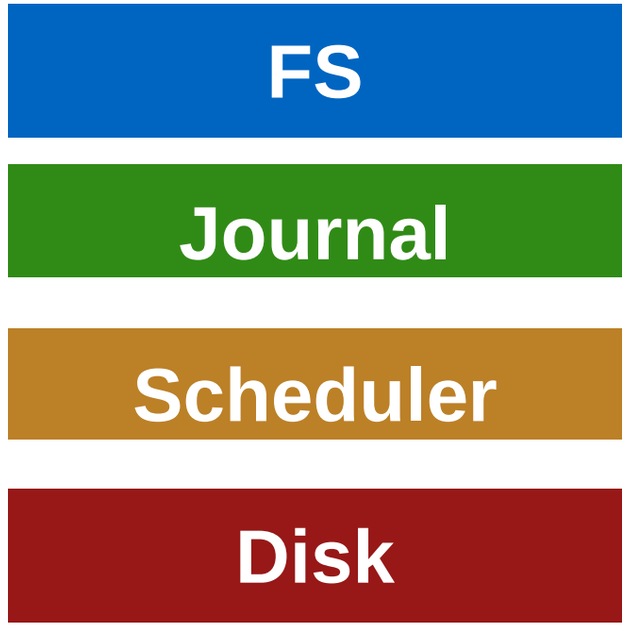
```
h = getHandle();
writeBlk(h, blknum, data);
finishHandle(h);
```

Blocks in the same handle must be written atomically.

File System Integration

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

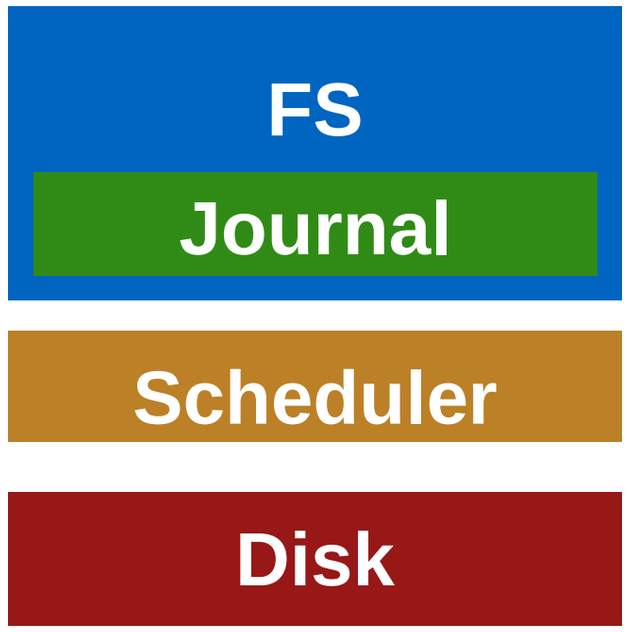
If we want to only journal FS metadata, we need tighter integration.



File System Integration

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

If we want to only journal FS metadata, we need tighter integration.



How to avoid writing all disk blocks Twice?

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

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

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

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

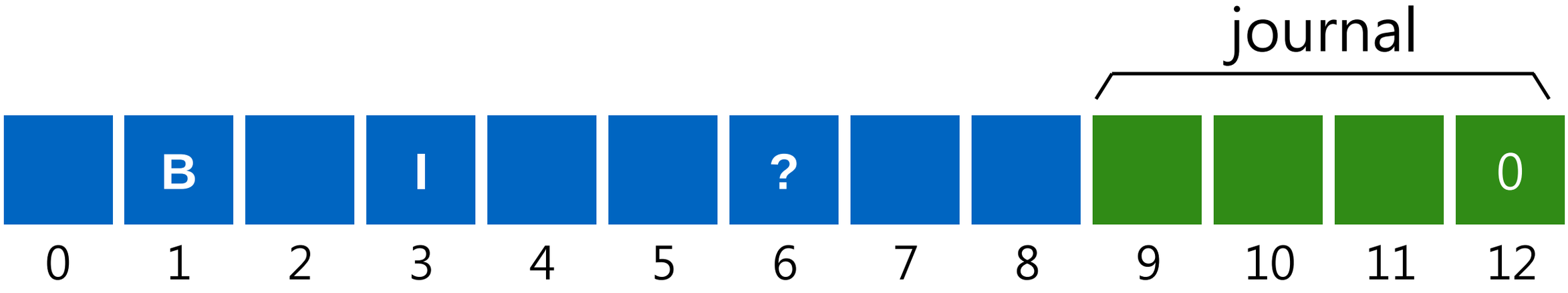
Writeback Journal

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

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

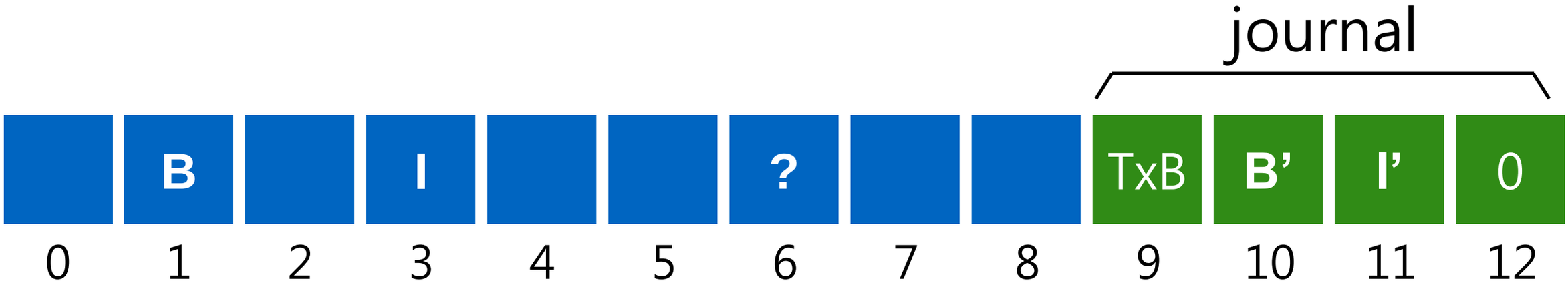
What is the worst type of garbage we could get? How to avoid?

Writeback Journal



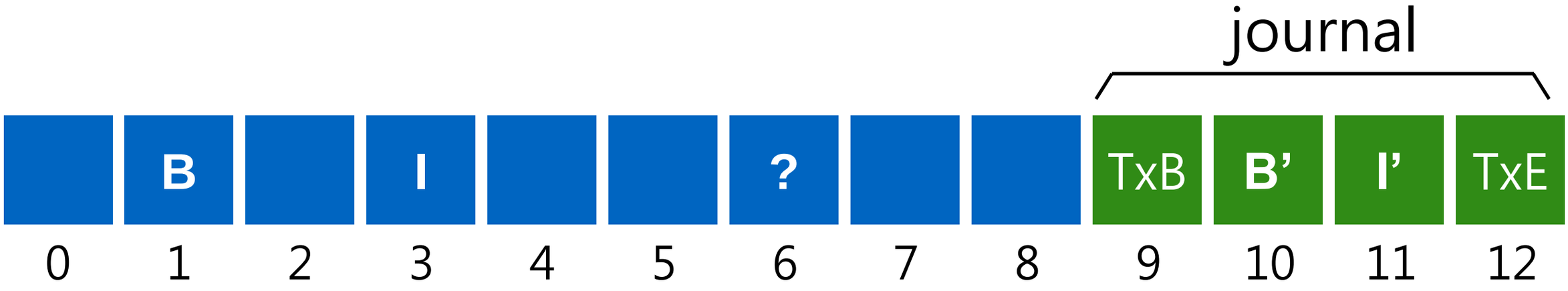
transaction: append to inode I

Writeback Journal



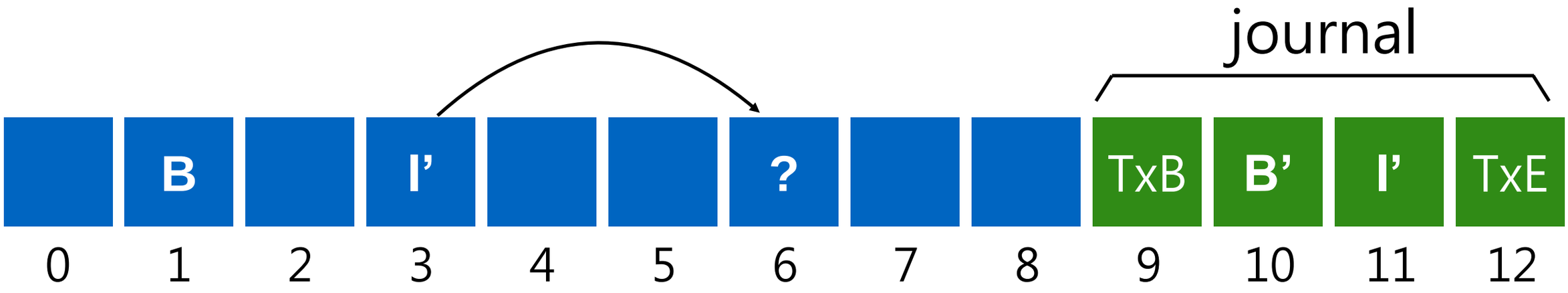
transaction: append to inode I

Writeback Journal



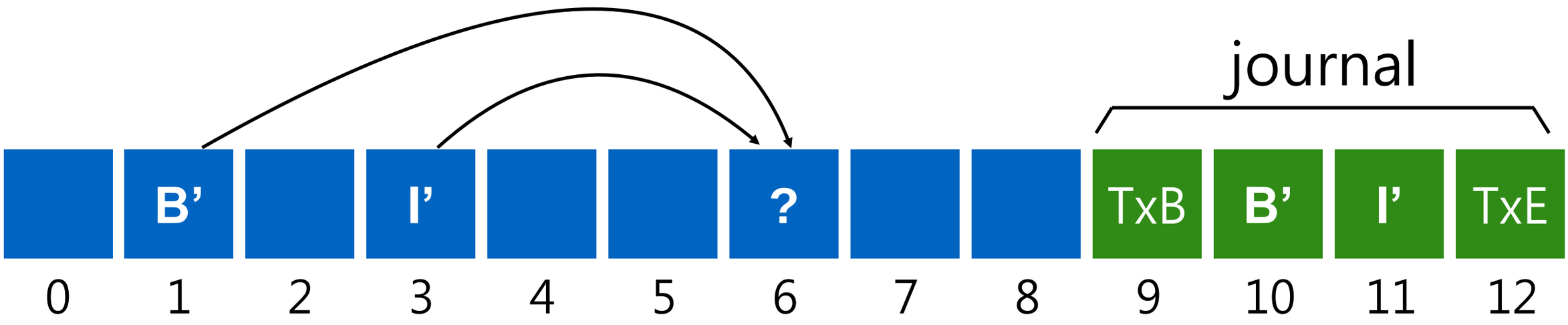
transaction: append to inode I

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

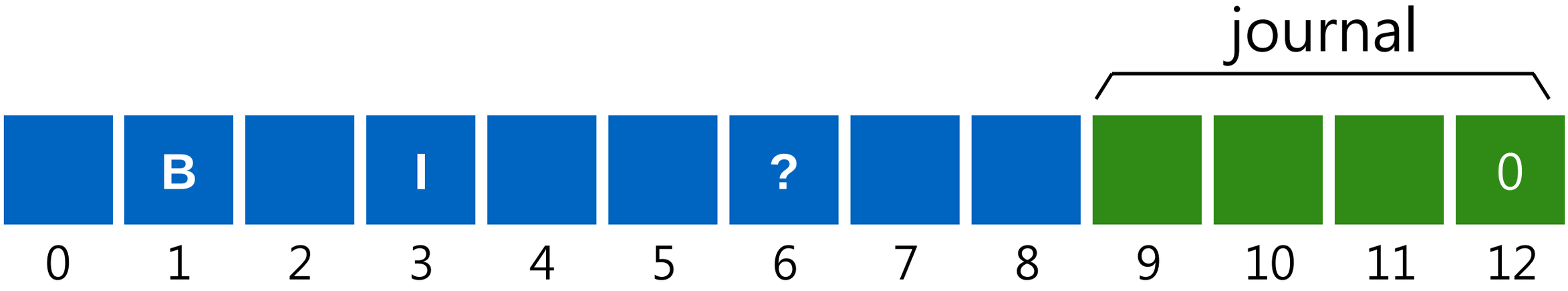
But write data **before** the transaction

May still get scrambled data on update.

But appends will always be good.

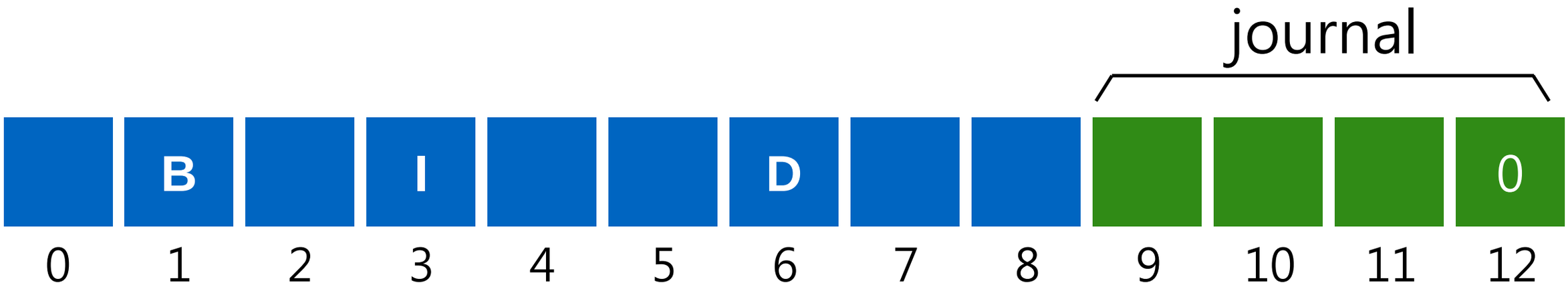
No leaks of sensitive data!

Ordered Journal



transaction: append to inode I

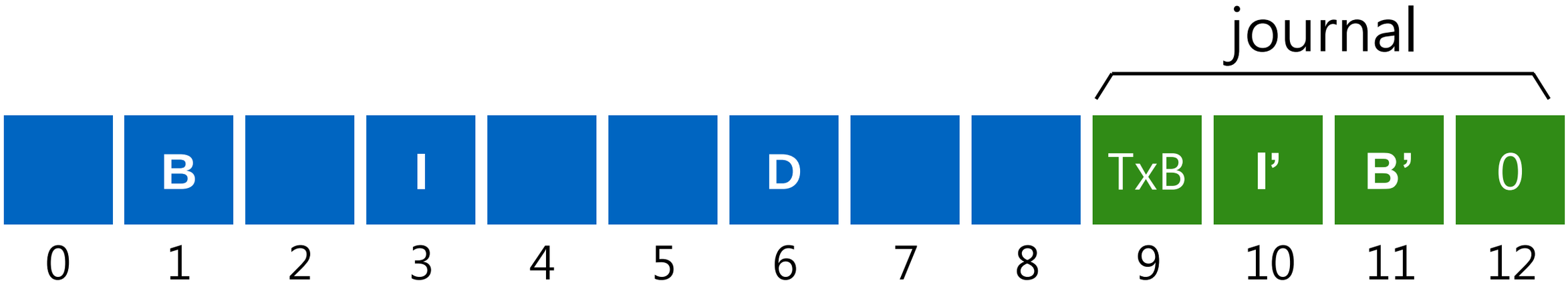
Ordered Journal



transaction: append to inode I

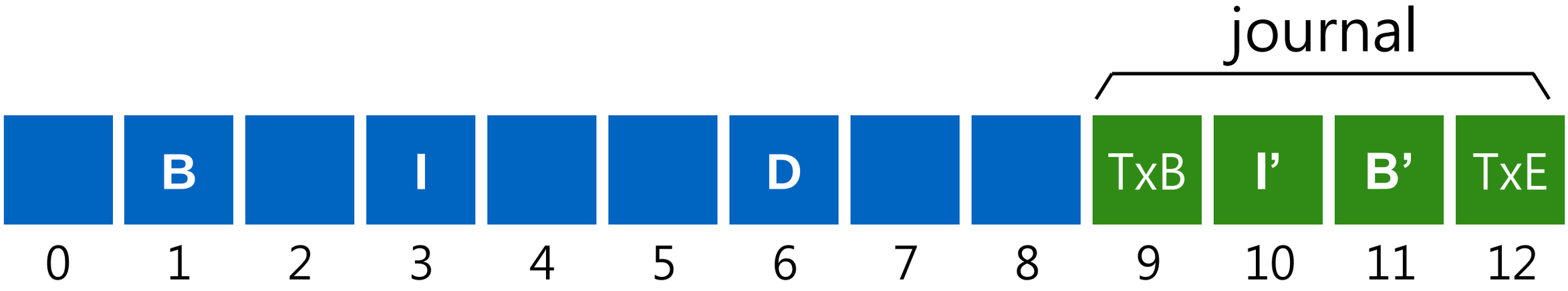
What happens if crash now?
B indicates D currently free, I does not point to D;
Lose D, but that might be acceptable

Ordered Journal



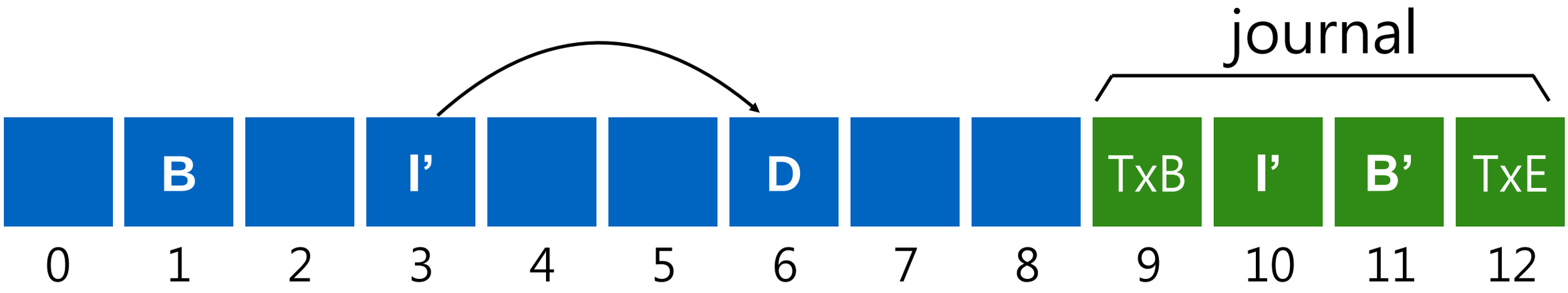
transaction: append to inode I

Ordered Journal



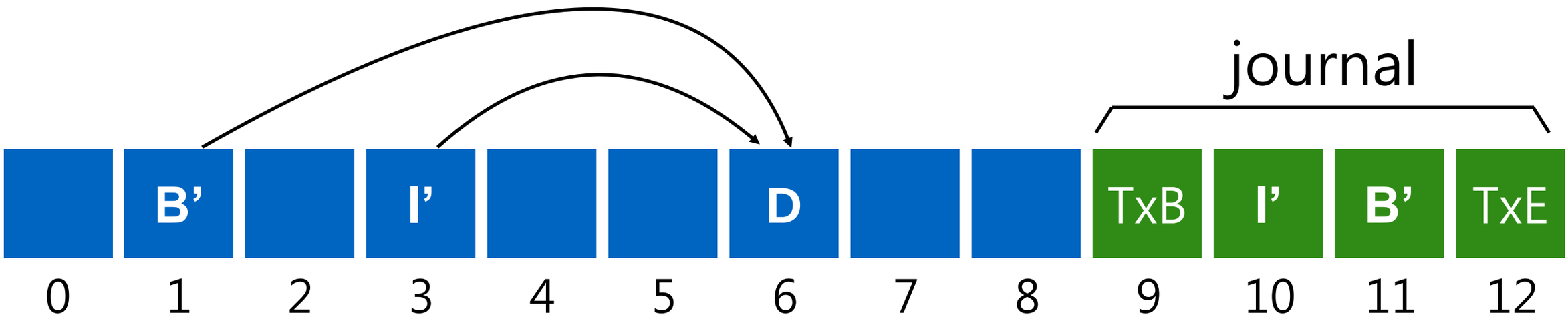
transaction: append to inode I

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) writ
e new data before deleting old (copy-on-write file systems)