[537] Journaling

Chapter 42

Tyler Harter

11/12/14

FFS Review

Problem 1

What structs must be updates in addition to the data block itself? [worksheet]

Problem 1

What structs must be updates in addition to the data block itself? [worksheet]

Data block bitmap.
Group descriptor.
Inode.

Fast File System

A few contributions:
- hybrid block size
- groups
- smart allocation

Fast File System

A few contributions:
- hybrid block size
- groups
- smart allocation

Problem 2

Compute waste in worksheet.

Hybrid: Blocks + Fragments

Big blocks: fast
Small blocks: space efficient

FFS split regular blocks into fragments when less than a block is needed.

Saving less than fragment size wastes space.

Appending less than block size causes copies.

New System Call

FFS gives new API to exposes block/fragment size:

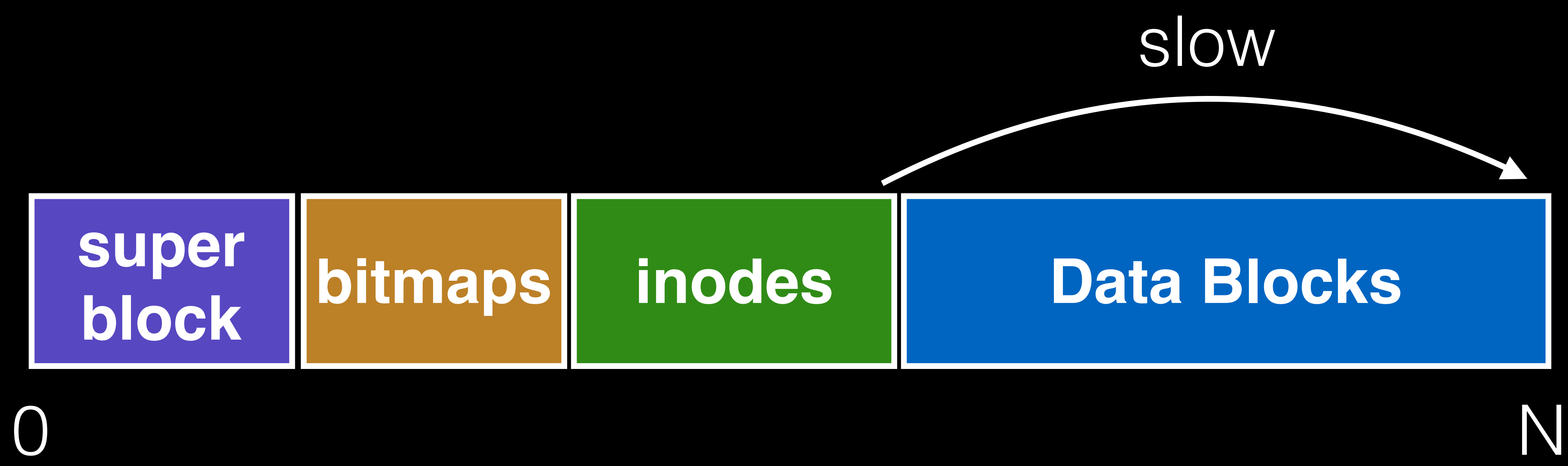
```
int fstatvfs(int fd, struct statvfs *buf);

struct statvfs {
   unsigned long  f_bsize;  // block size
   unsigned long  f_frsize; // fragment size
   ...
};
```

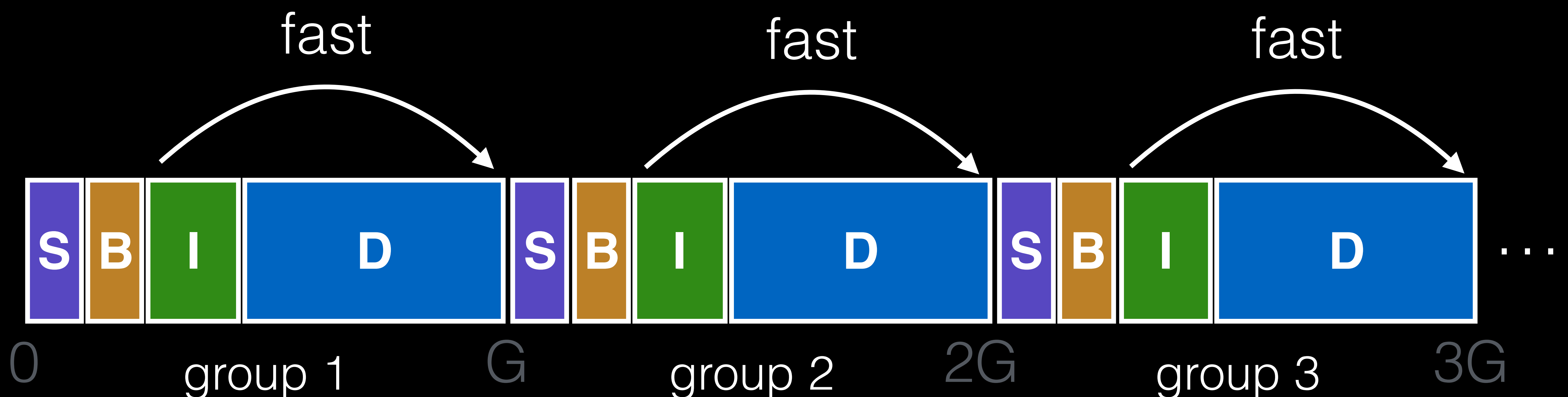
Fast File System

A few contributions:
- hybrid block size
- groups
- smart allocation

Old UNIX File System



Fast File System is Fast



With groups, each inode has data blocks near it.

Fast File System

A few contributions:
- hybrid block size
- groups
- smart allocation

Challenge

The file system is one big tree.

All directories and files have a common root.

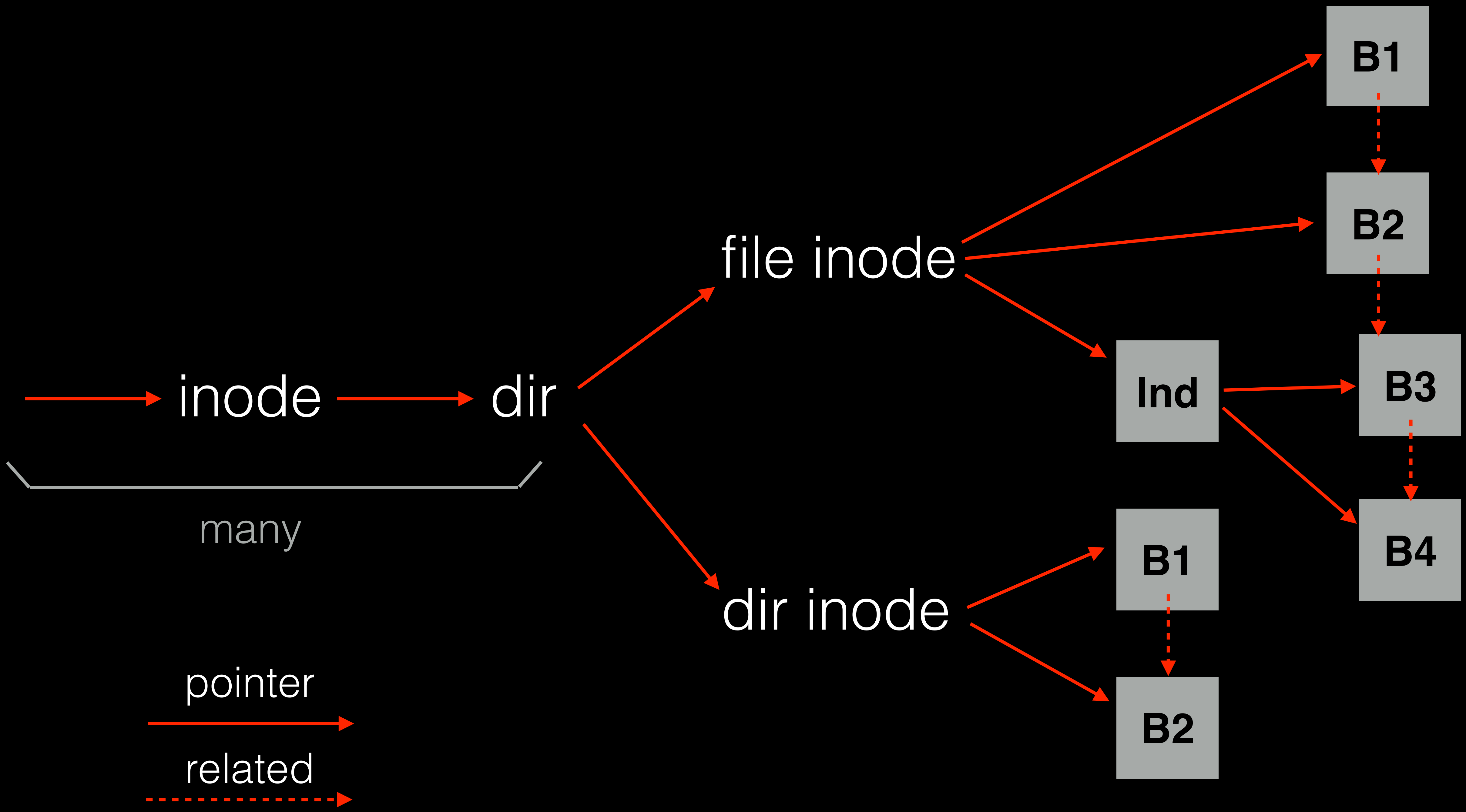
In some sense, all data in the same FS is related.

Trying to put everything near everything else will leave us with the same mess we started with.

Revised Strategy

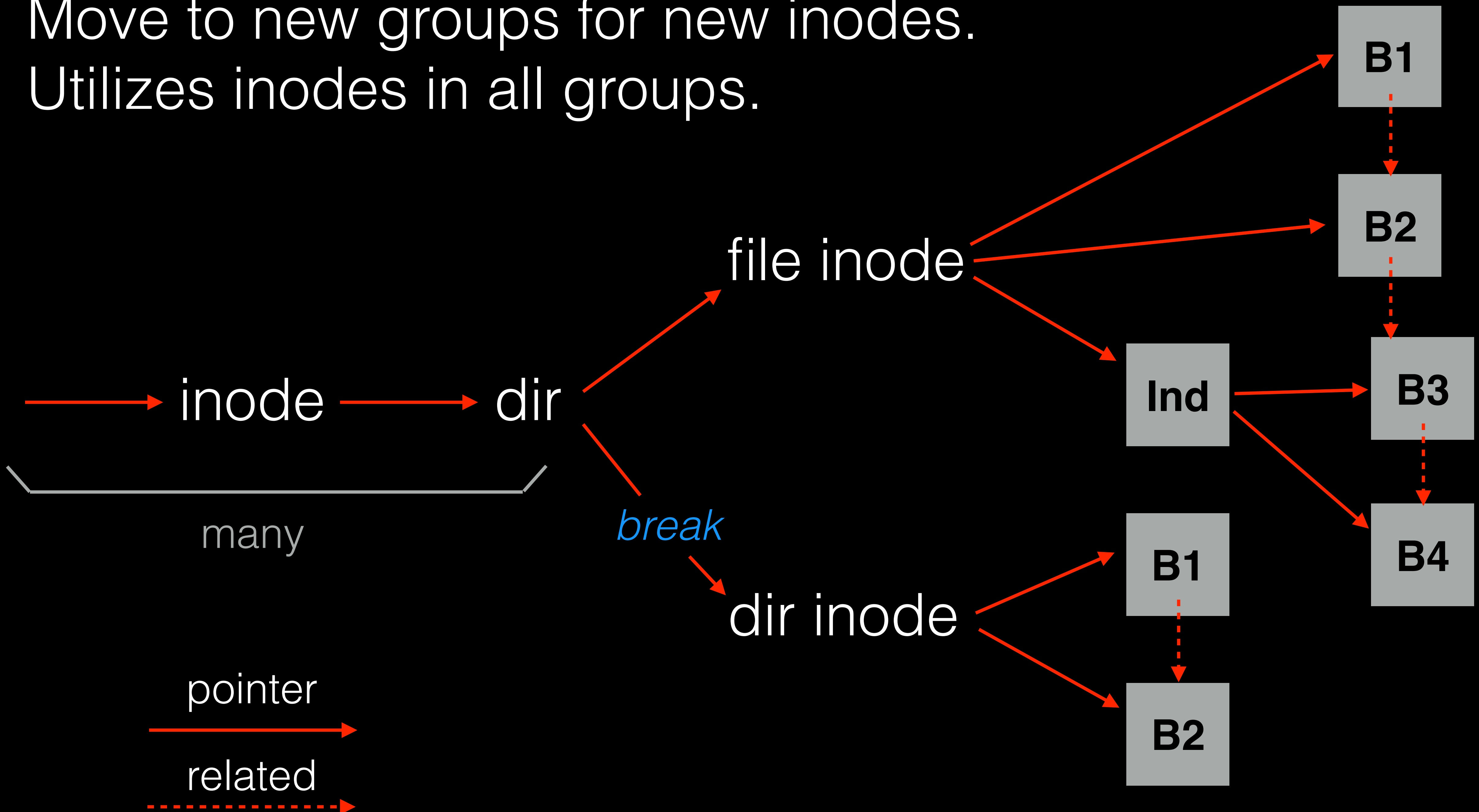
Put more-related pieces of data near each other.

Put less-related pieces of data far from each other.

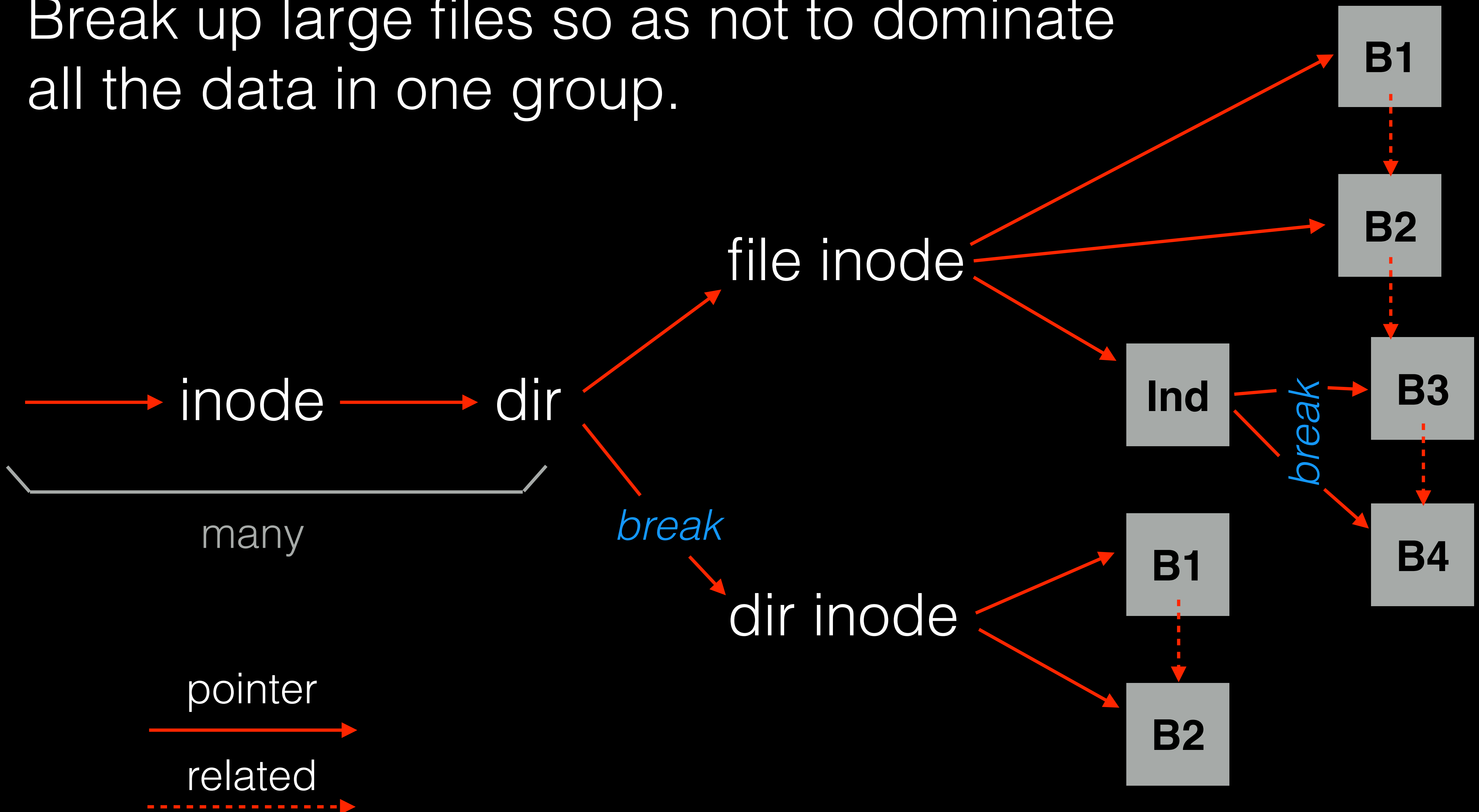


inode
dir
many
file inode
dir inode
B1
B2
Ind
B3
B4
B1
B2
pointer
related

Move to new groups for new inodes.
Utilizes inodes in all groups.



Break up large files so as not to dominate all the data in one group.



Redundancy

Redundancy

Definition: if *A* and *B* are two pieces of data, and knowing *A* eliminates some or all the values *B* could *B*, there is redundancy between *A* and *B*.

RAID examples:

- mirrored disk (complete redundancy)
- parity blocks (partial redundancy)

Subtle Example

Definition: if *A* and *B* are two pieces of data, and knowing *A* eliminates some or all the values *B* could *B*, there is redundancy between *A* and *B*.

Superblock: field contains total blocks in FS.

Inode: field contains pointer to data block.

Is there redundancy between these fields? Why?

Subtle Example

Superblock: field contains total blocks in FS.
DATA = ???

Inode: field contains pointer to data block.
DATA in {0, 1, 2, …, UINT_MAX}

Subtle Example

Superblock: field contains total blocks in FS.
DATA = N

Inode: field contains pointer to data block.
DATA in {0, 1, 2, …, UINT_MAX}

Subtle Example

Superblock: field contains total blocks in FS.
DATA = N

Inode: field contains pointer to data block.
DATA in {0, 1, 2, …, N - 1}

Pointers to block N or after are invalid!

Subtle Example

Superblock: field contains total blocks in FS.
DATA = N

Inode: field contains pointer to data block.
DATA in {0, 1, 2, …, N - 1}

Pointers to block N or after are invalid!

Total-blocks field has redundancy with inode pointers.

Problem 3

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

Problem 3

Give 5 examples of redundancy in FFS
(or files systems 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.
…

Redundancy Uses

Redundancy may improve:
- performance
- reliability

Redundancy hurts:
- capacity

Redundancy Uses

Redundancy may improve:
 - performance (e.g., FFS group descriptor)
 - reliability (e.g., RAID-5 parity)

Redundancy hurts:
 - capacity

Redundancy Challenges

Redundancy implies:
certain combinations of values are illegal.

Names for bad combinations:
- contradictions
- inconsistencies

Example

Superblock: field contains total blocks in FS.
DATA = 1024

Inode: field contains pointer to data block.
DATA in {0, 1, 2, …, 1023}

Example

Superblock: field contains total blocks in FS.
DATA = 1024

Inode: field contains pointer to data block.
DATA = 241

Consistent.

Example

Superblock: field contains total blocks in FS.
DATA = 1024

Inode: field contains pointer to data block.
DATA = 2345

Inconsistent.

Consistency Challenge

We may need to do several disk writes to redundant blocks.

We don't want to be interrupted between writes.

Consistency Challenge

We may need to do several disk writes to redundant blocks.

We don't want to be interrupted between writes.

Things that interrupt us:
- power loss
- kernel panic, reboot
- user hard reset

Problem 4

Suppose we are appending to a file, and must update the following:

- inode
- data bitmap
- data block

What happens if we crash after only updating some of these?

Partial Update

a) **bitmap**: lost block
b) **data**: nothing bad
c) **inode**: point to garbage, somebody else may use
d) **bitmap** and **data**: lost block
e) **bitmap** and **inode**: point to garbage
f) **data** and **inode**: somebody else may use

Partial Update

a) **bitmap**: lost block
b) **data**: nothing bad
c) **inode**: point to garbage, somebody else may use
d) **bitmap** and **data**: lost block
e) **bitmap** and **inode**: point to garbage
f) **data** and **inode**: somebody else may use

What is in “garbage”?

FSCK

fsck

FSCK = file system checker.

Strategy: after a crash, scan whole disk for contradictions.

fsck

FSCK = file system checker.

Strategy: after a crash, scan whole disk for contradictions.

For example, is a bitmap block correct?

Read every valid inode+indirect. If an inode points to a block, the corresponding bit should be 1

fsck

Other checks:

Do superblocks match?
Do number of dir entries equal inode link counts?
Do different inodes ever point to same block?
Do directories contain "." and ".."?
…

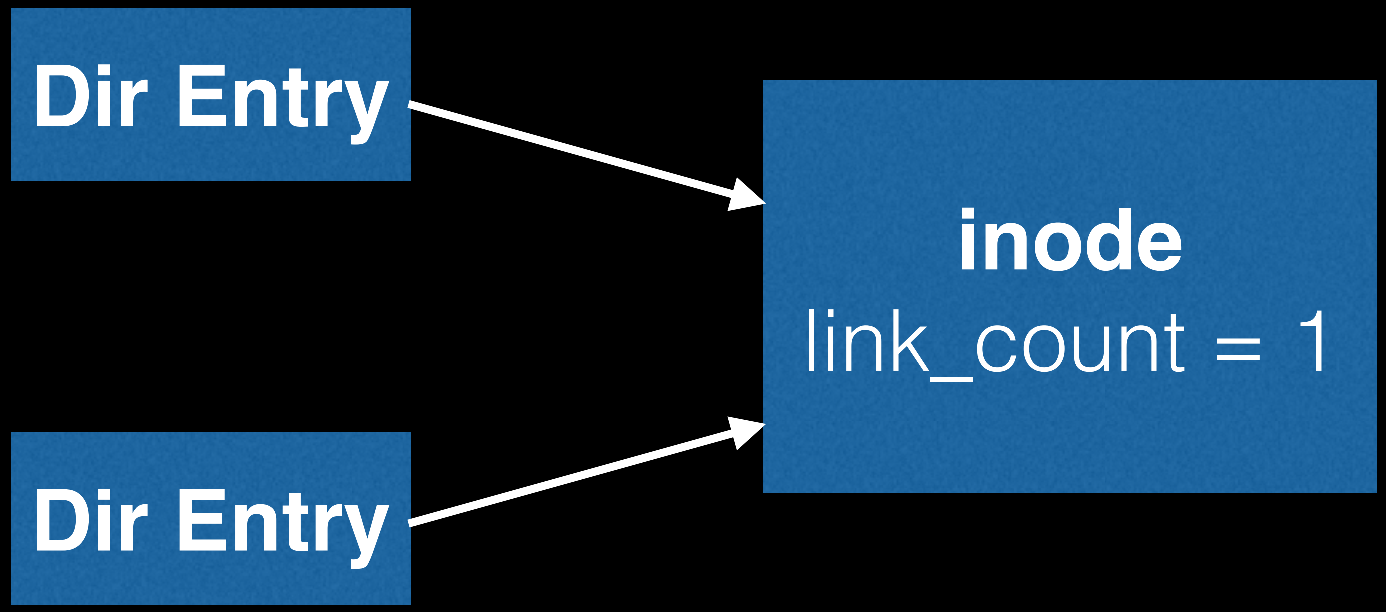
fsck

Other checks:

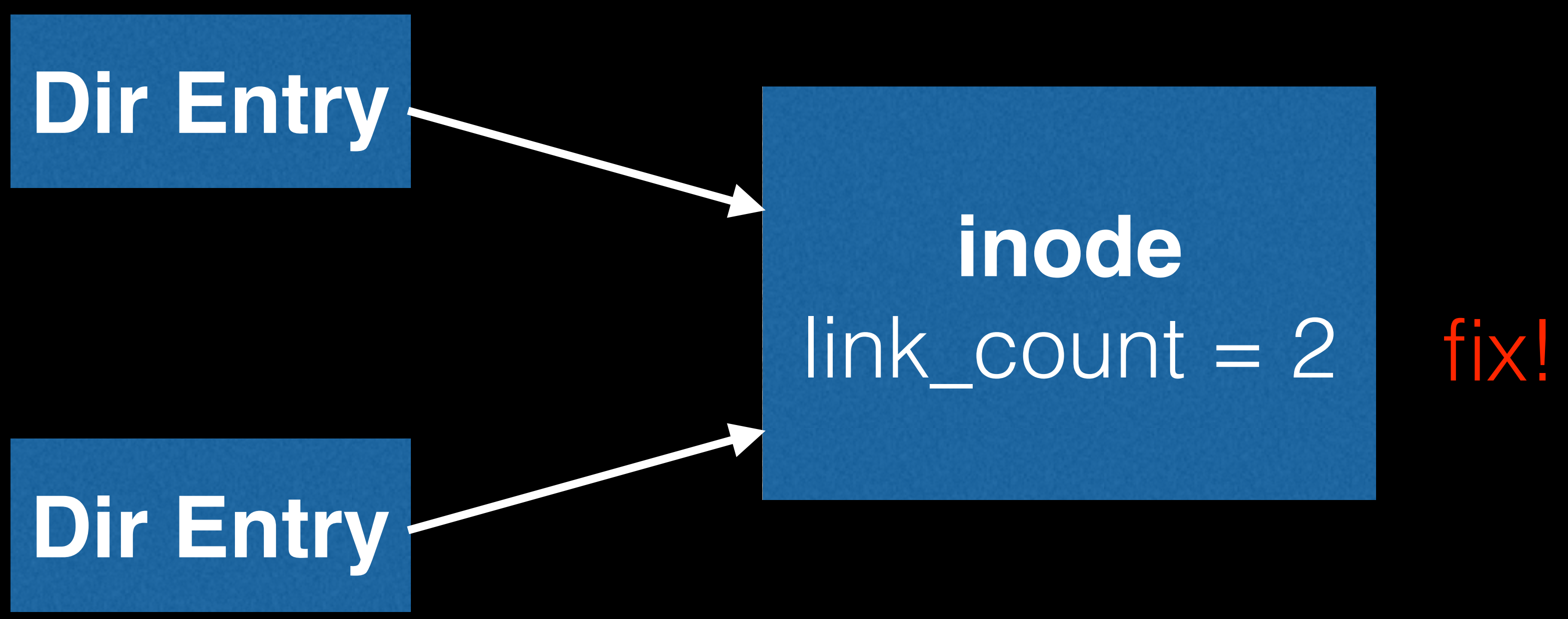
Do superblocks match?
Do number of dir entries equal inode link counts?
Do different inodes ever point to same block?
Do directories contain “.” and “..”?
…

How to solve problems?

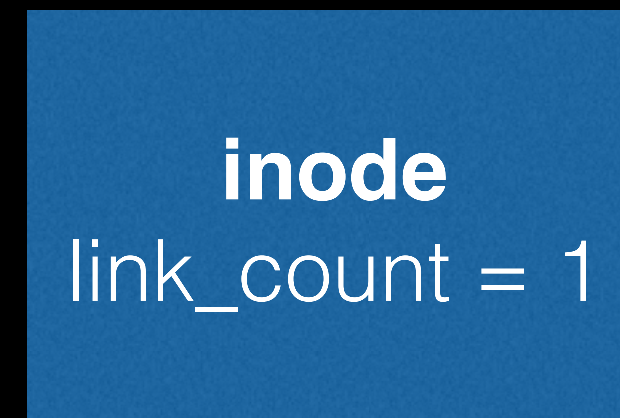
Link Count (example 1)



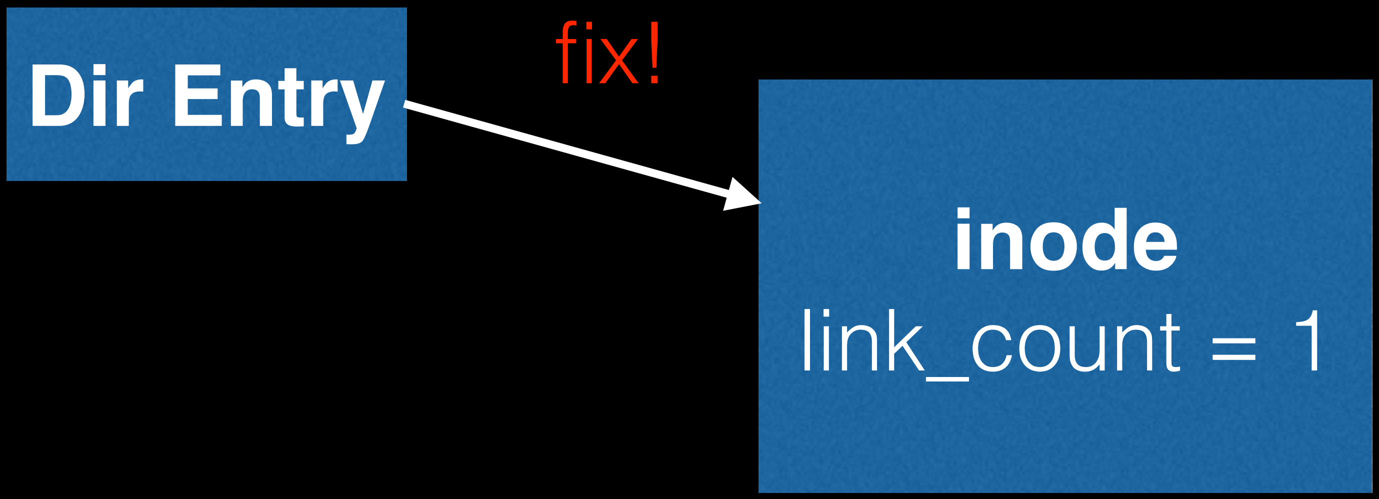
Link Count (example 1)



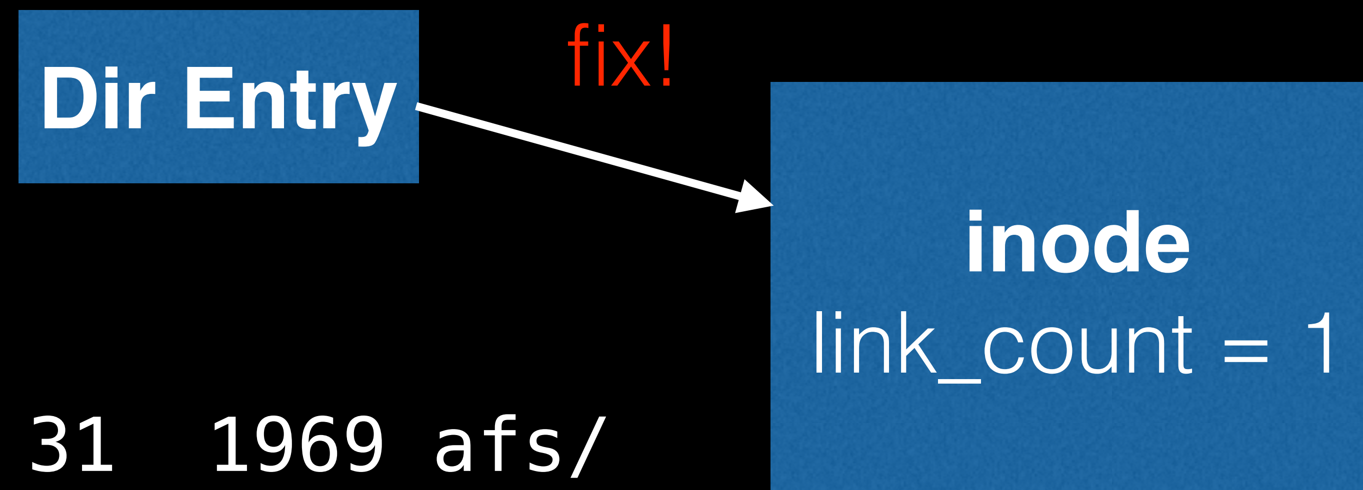
Link Count (example 2)



Link Count (example 2)

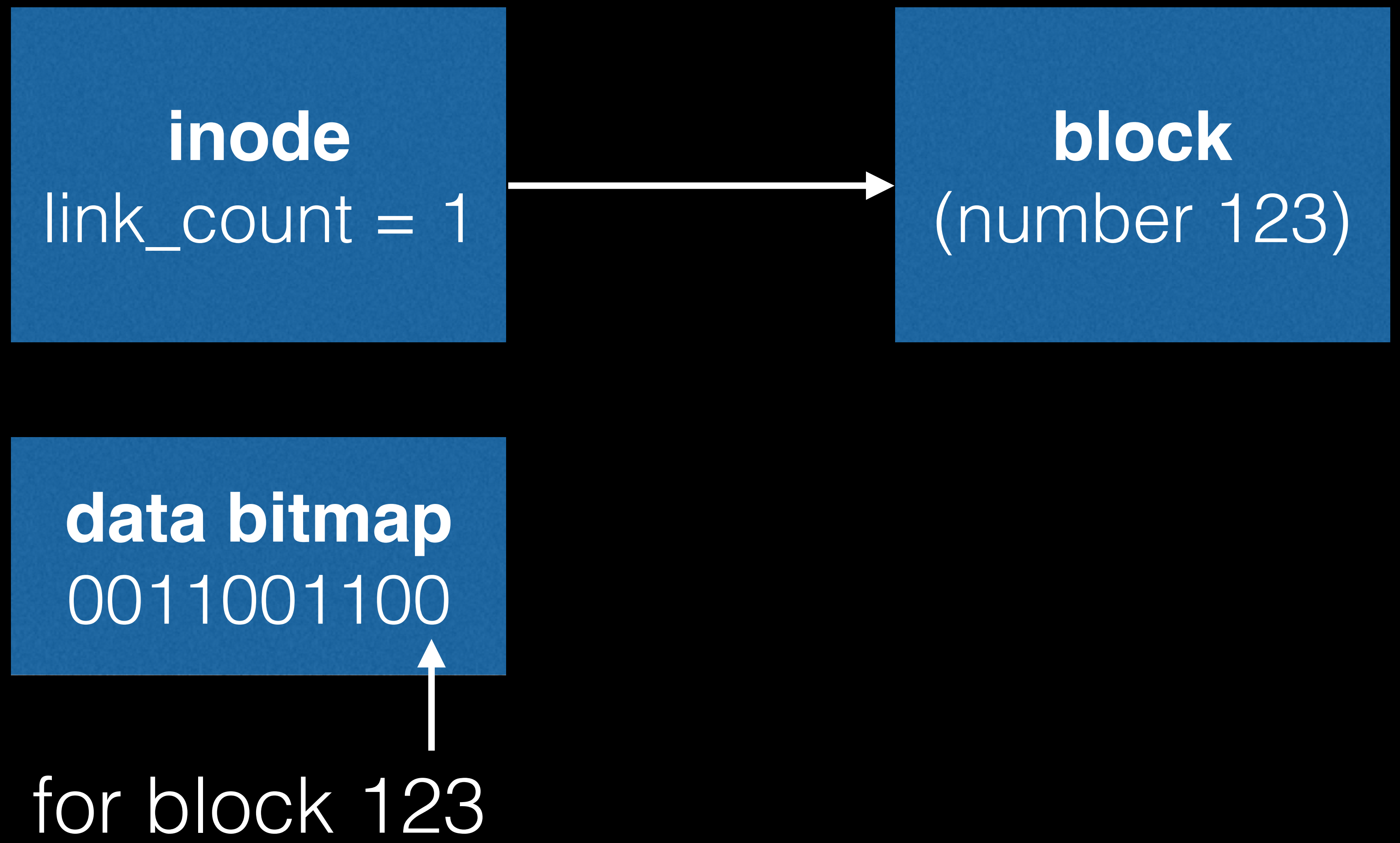


Link Count (example 2)

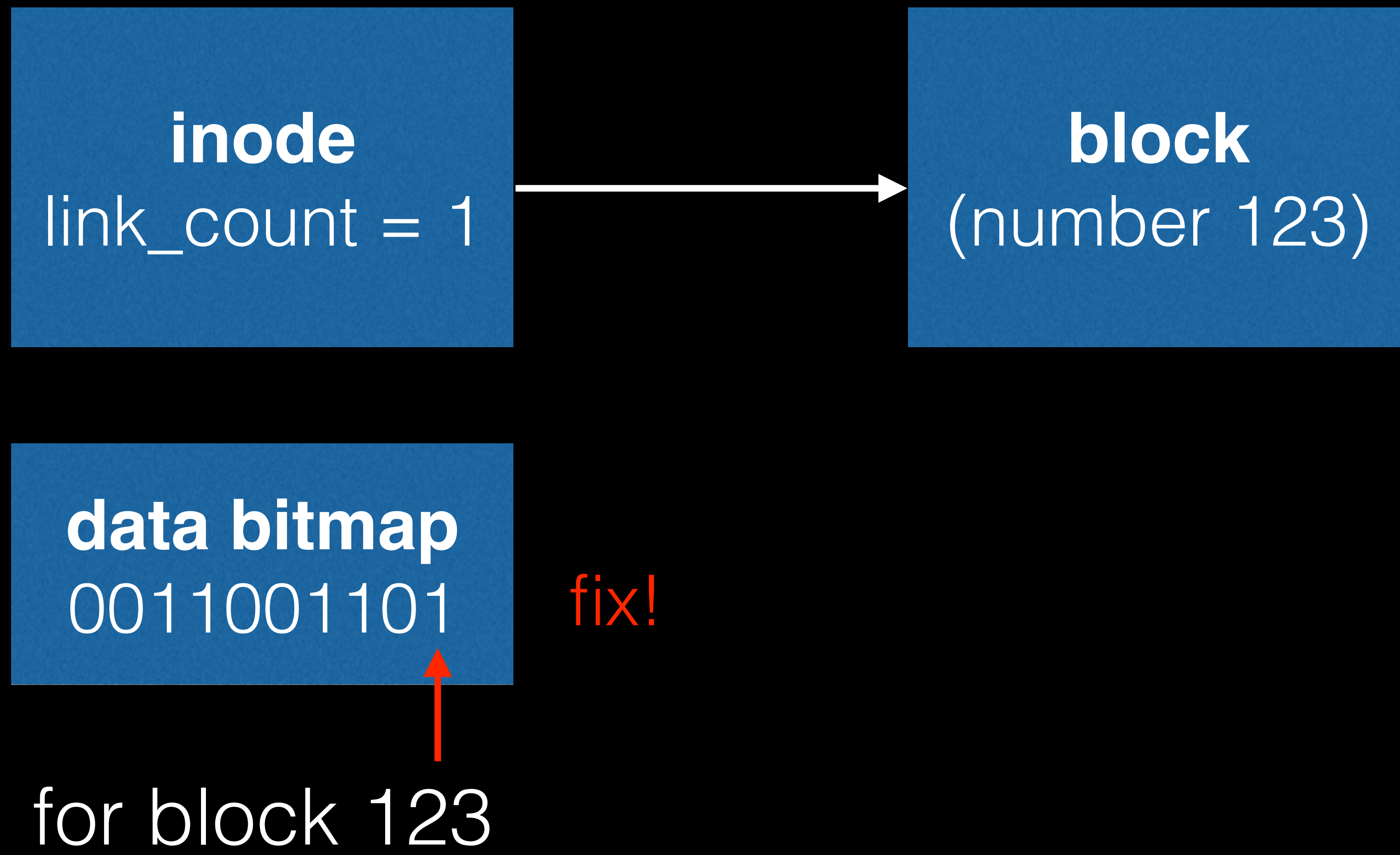



```
ls -l /
total 150
drwxr-xr-x  401 18432 Dec 31  1969 afs/
drwxr-xr-x.   2 4096  Nov  3 09:42 bin/
drwxr-xr-x.   5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.  13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.  10 12288 Nov  3 09:41 lib64/
drwx------.   2 16384 Aug  1 10:57 lost+found/
...
```

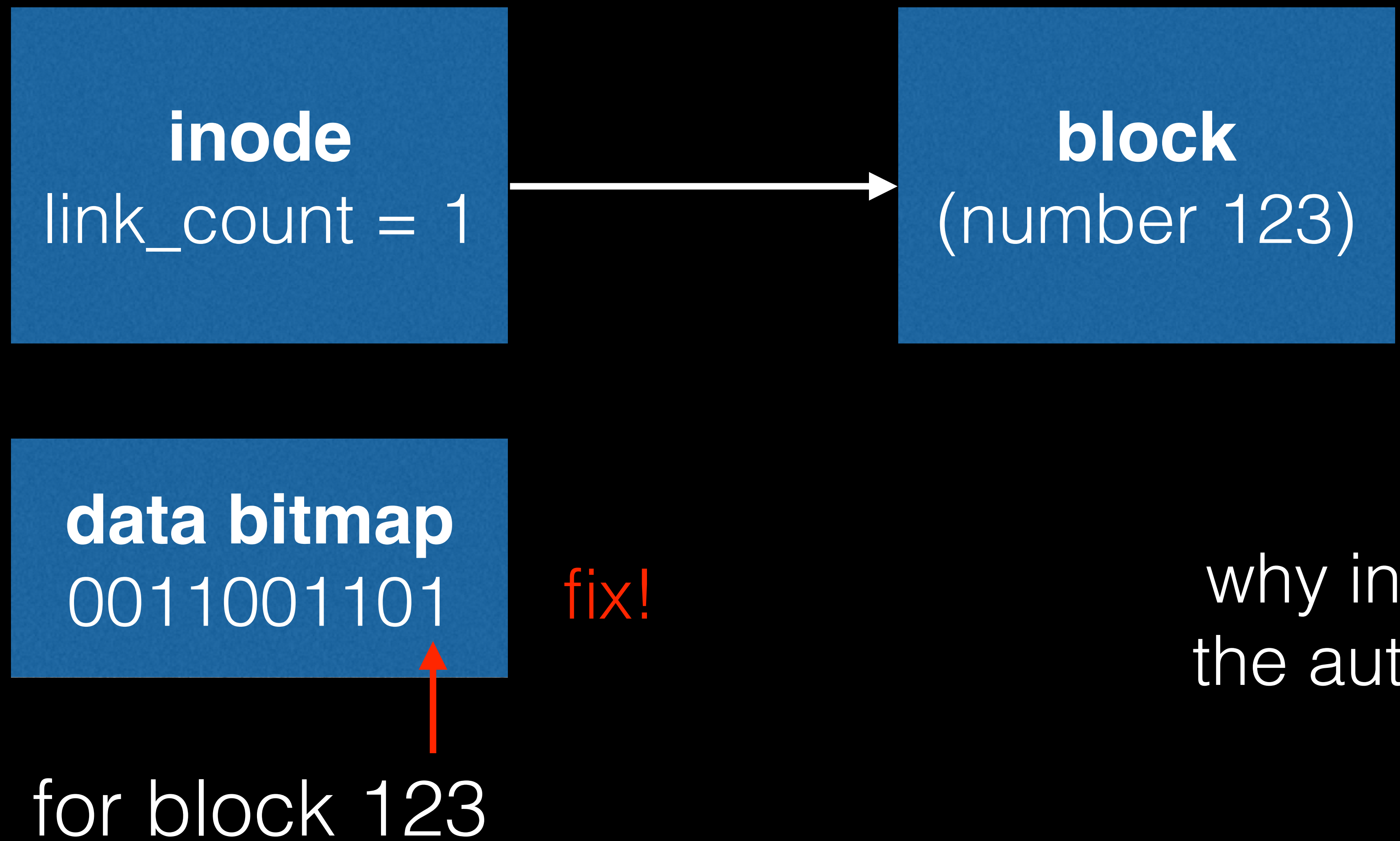
Data Bitmap


inode
link_count = 1
block
(number 123)
data bitmap
0011001100
for block 123

Data Bitmap

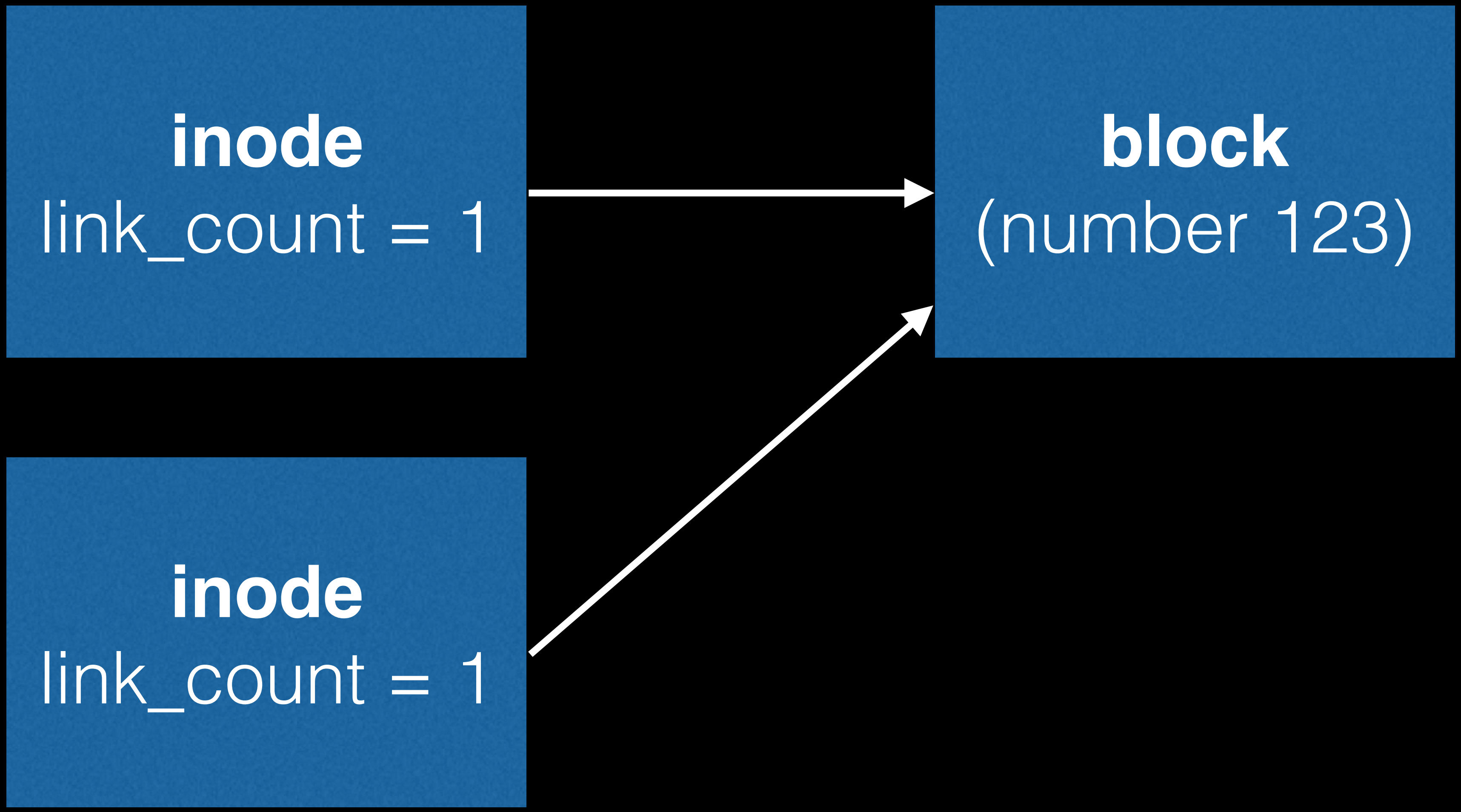

inode
link_count = 1
block
(number 123)
data bitmap
0011001101
fix!
for block 123

Data Bitmap

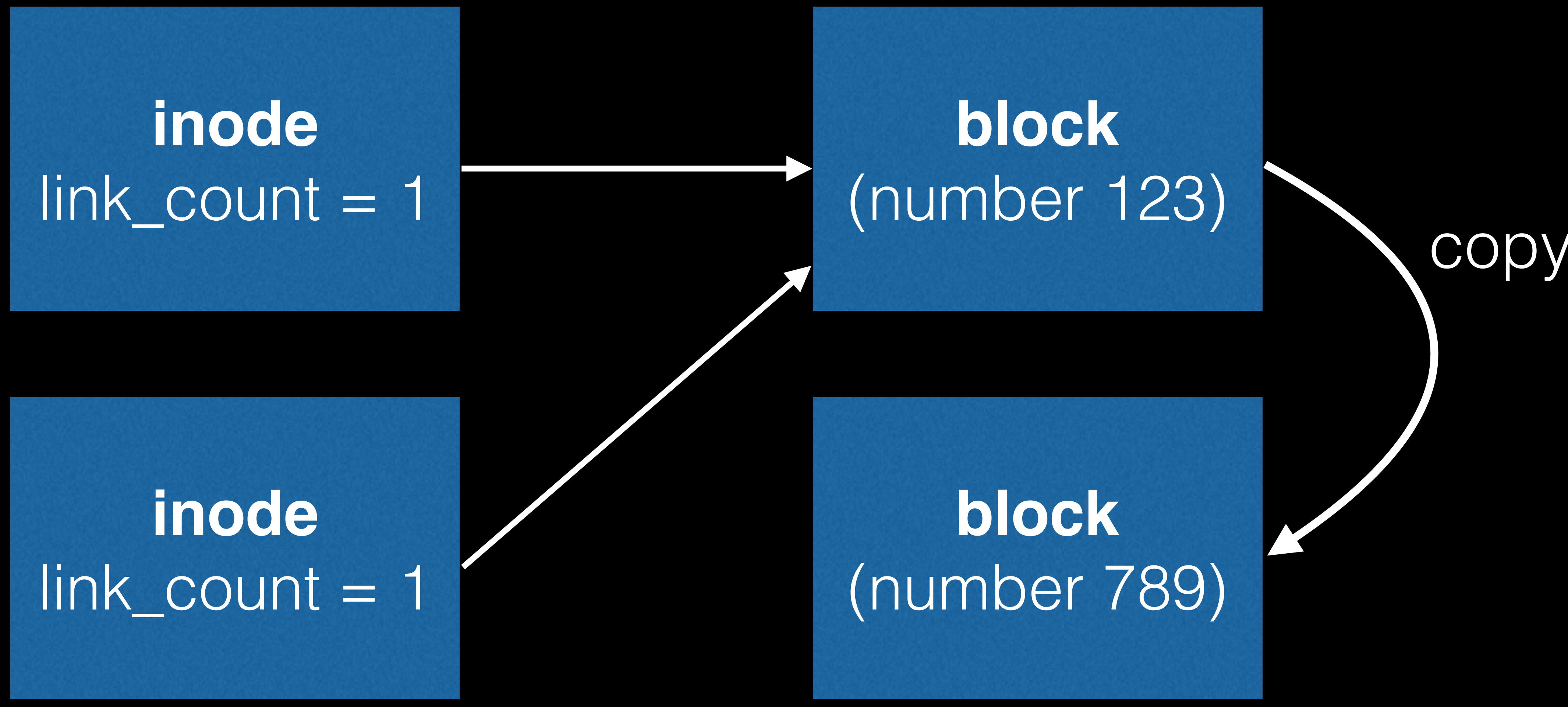


why in inode
the authority?

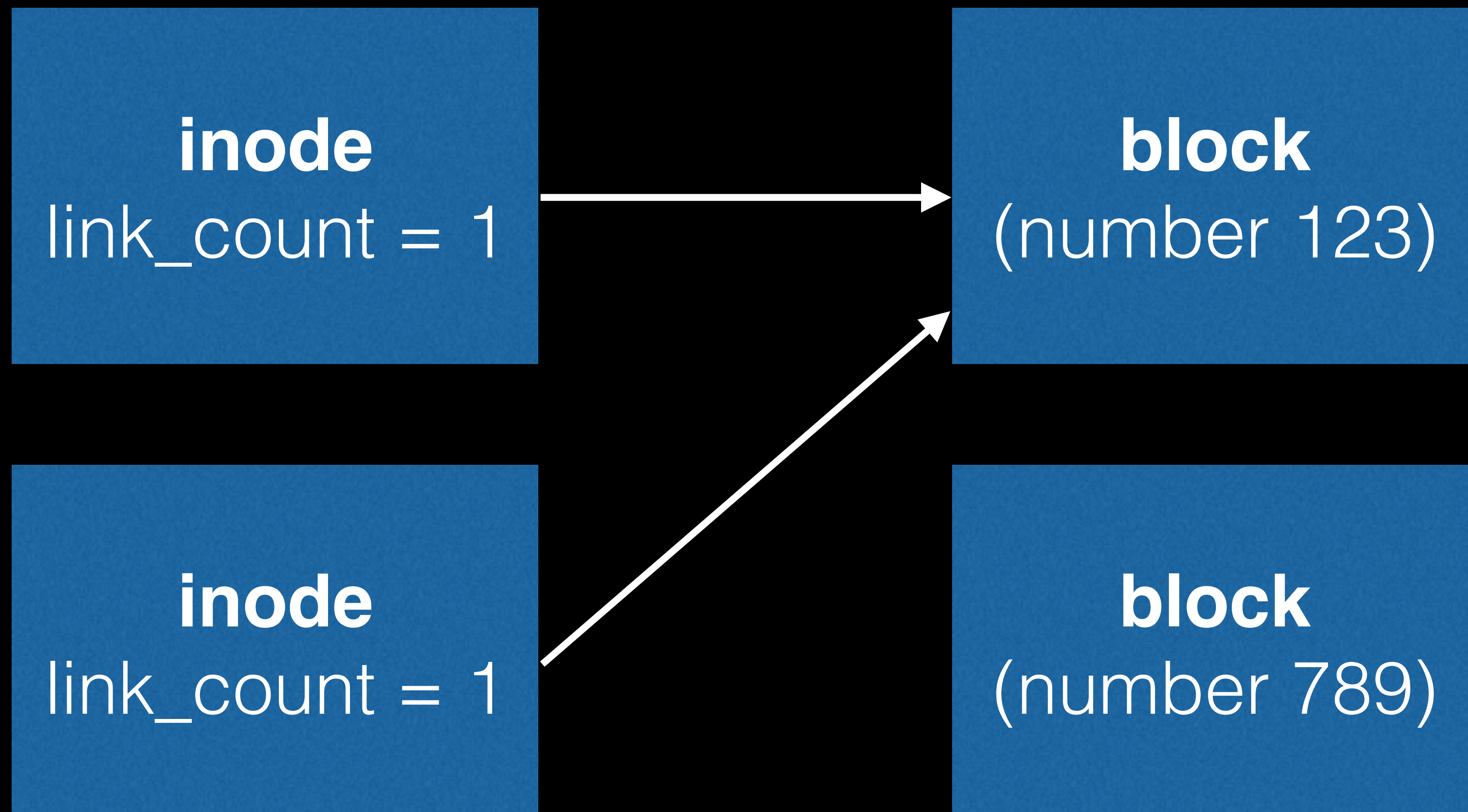
Duplicate Pointers



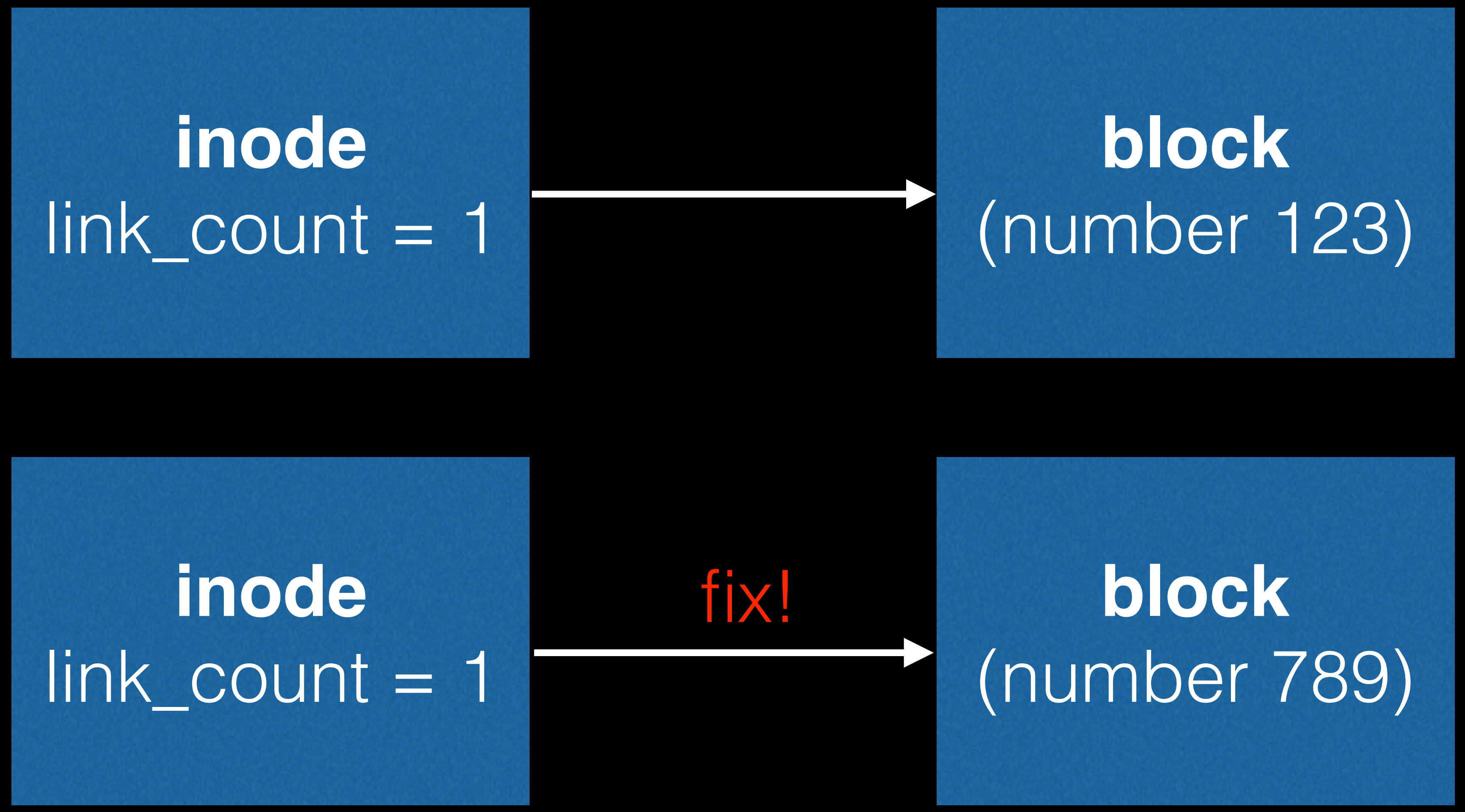
Duplicate Pointers



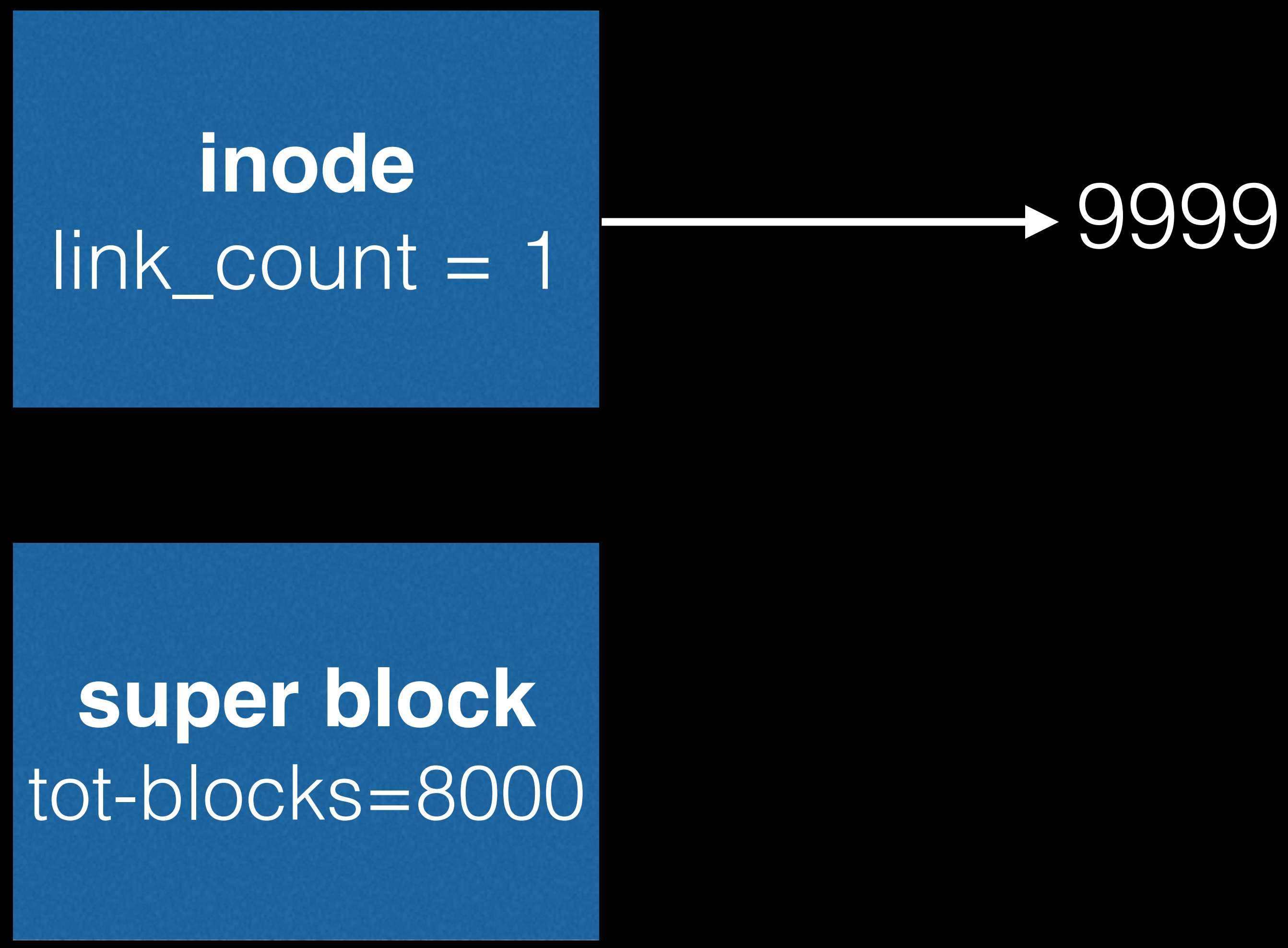
Duplicate Pointers



Duplicate Pointers



Bad Pointer



Bad Pointer

inode
link_count = 1

fix!

super block
tot-blocks=8000

fsck

It's not always obvious how to patch the file system back together.

We don't know the "correct" state, just a consistent one.

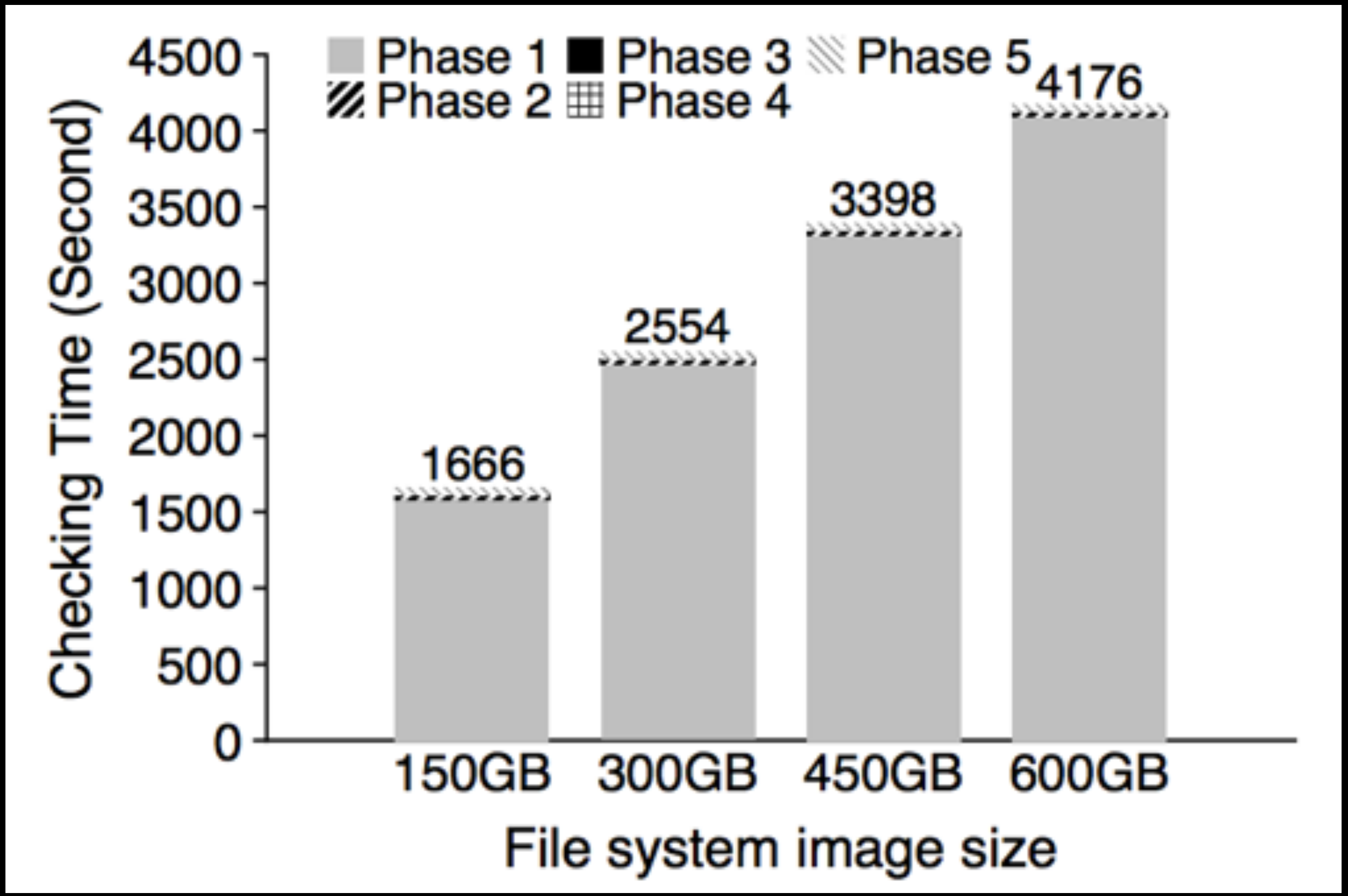
fsck

It’s not always obvious how to patch the file system back together.

We don’t know the “correct” state, just a consistent one.

Easy way to get consistency: reformat disk!

fsuck 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

Journaling

Goals

It’s ok to do some recovery work after crash, but not to read entire disk.

Don’t just get to a consistent state, get to a “correct” state.

Goals

It’s ok to do some recovery work after crash, but not to read entire disk.

Don’t just get to a consistent state, get to a “correct” state.

Strategy: atomicity.

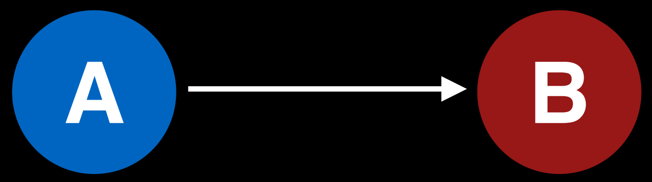
Atomicity

Concurrency definition:
operations in critical sections are not interrupted by operations on other critical sections.

Persistence definition:
collections of writes are not interrupted by crashes. Get all new or all old data.

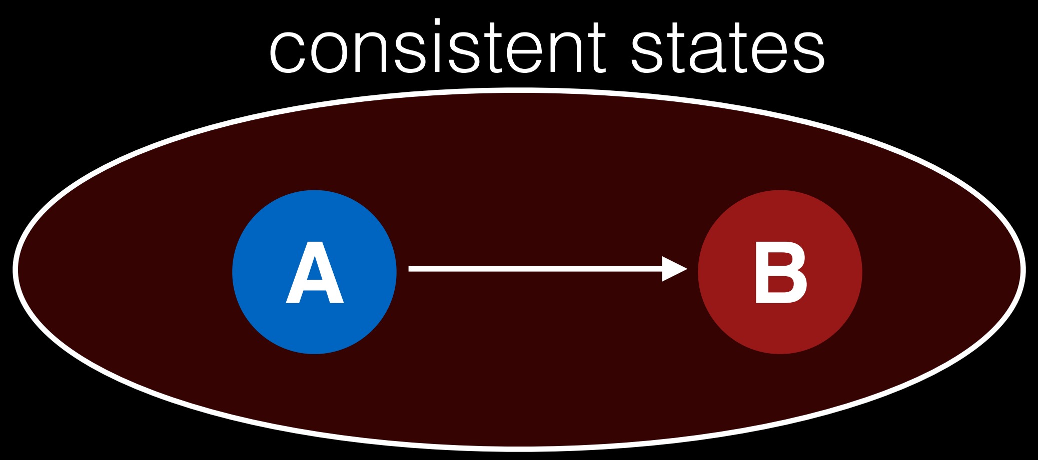
Consistency vs Correctness

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



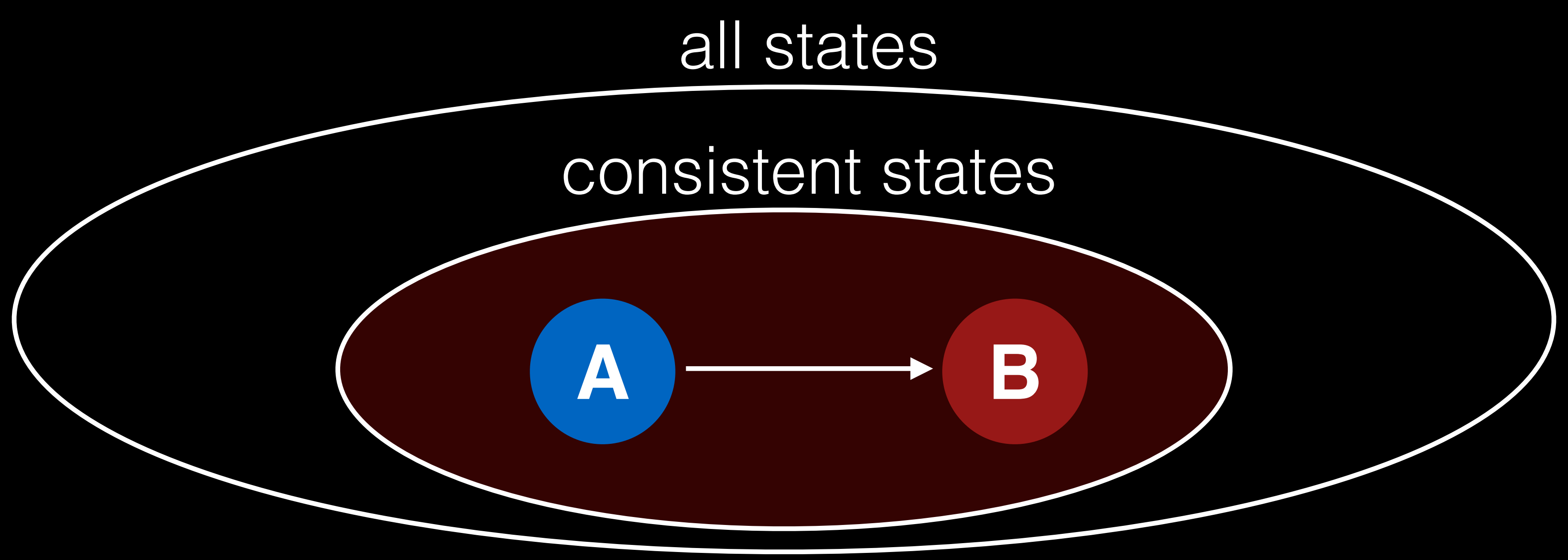
Consistency vs Correctness

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



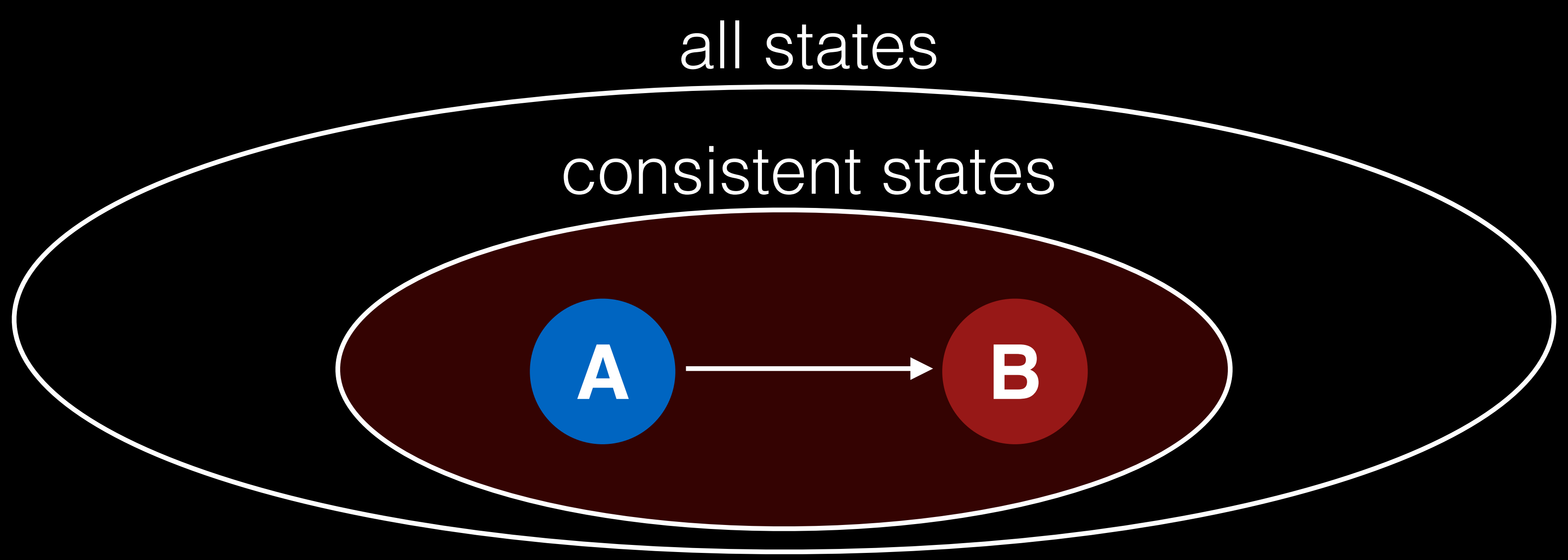
Consistency vs Correctness

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



Consistency vs Correctness

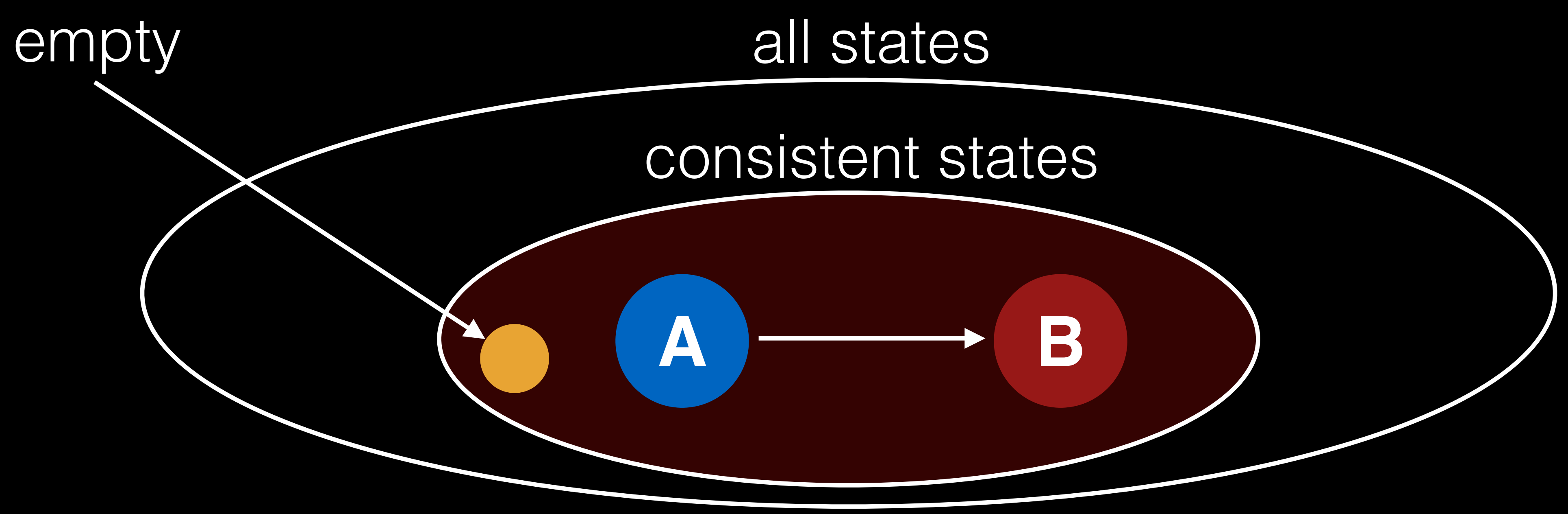
Say a set of writes moves the disk from state A to B.



fsck gives consistency. Atomicity gives us A or B.

Consistency vs Correctness

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



fsck gives consistency. Atomicity gives us A or B.

General Strategy

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

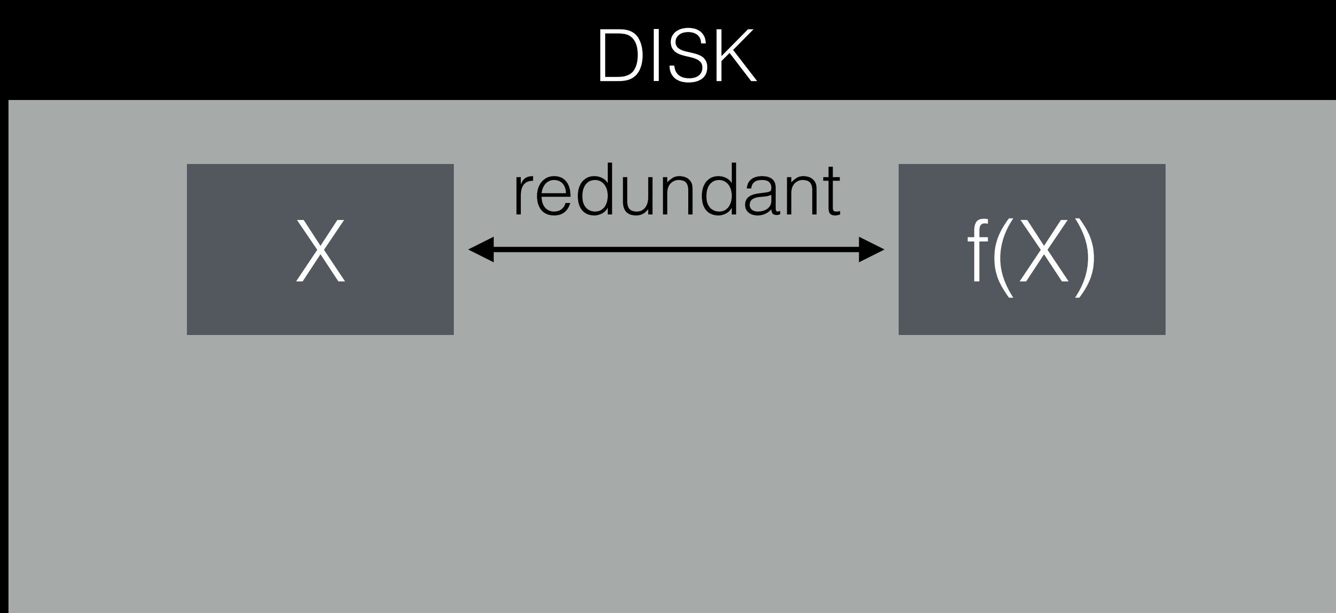
General Strategy

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

Ironically, this means we're adding redundancy to fix the problem caused by redundancy.

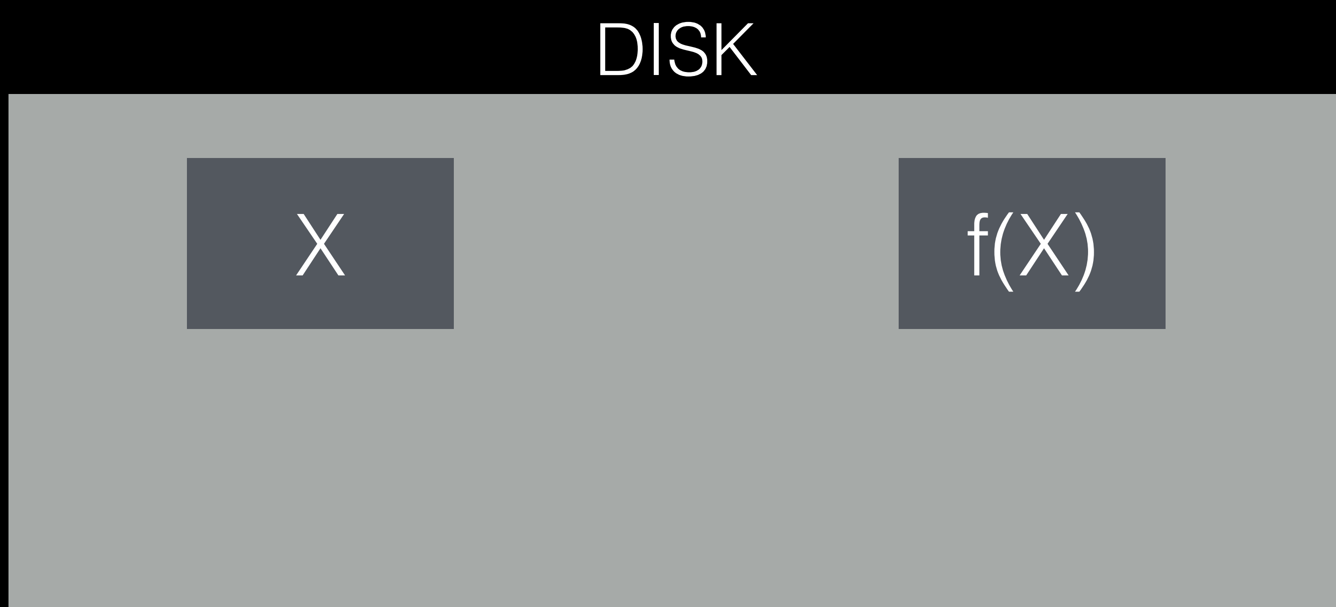
Fight Redundancy with Redundancy

Want to replace X with Y. Original:



Fight Redundancy with Redundancy

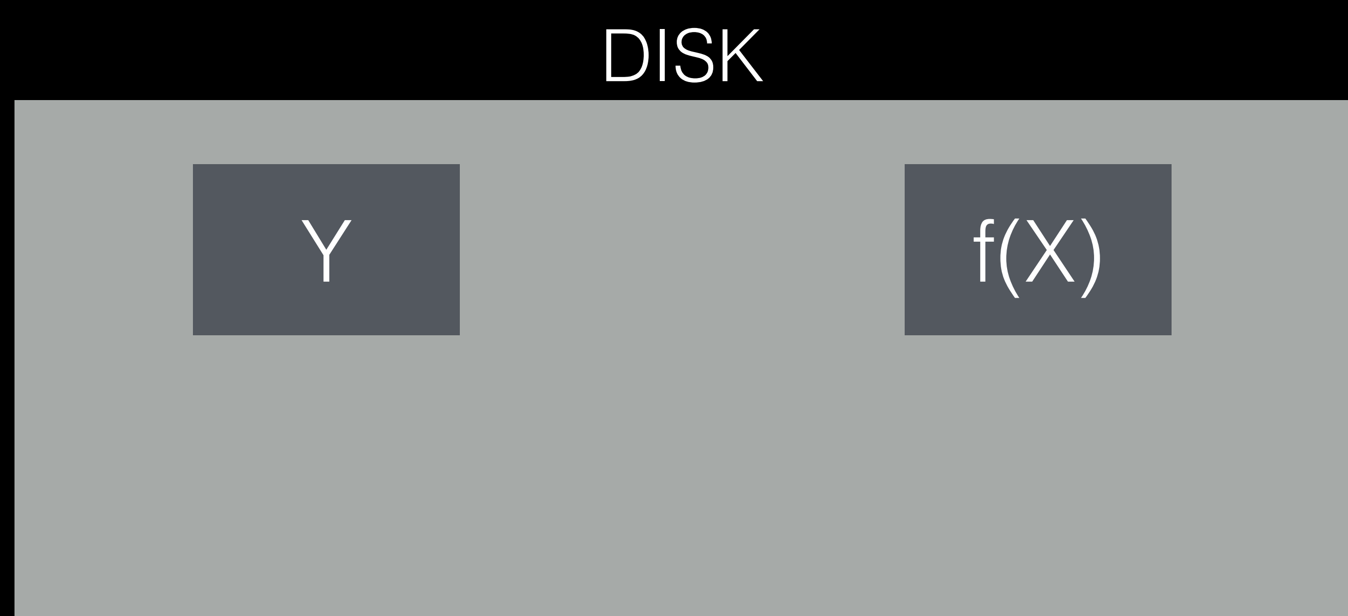
Want to replace X with Y. Original:



good time to crash

Fight Redundancy with Redundancy

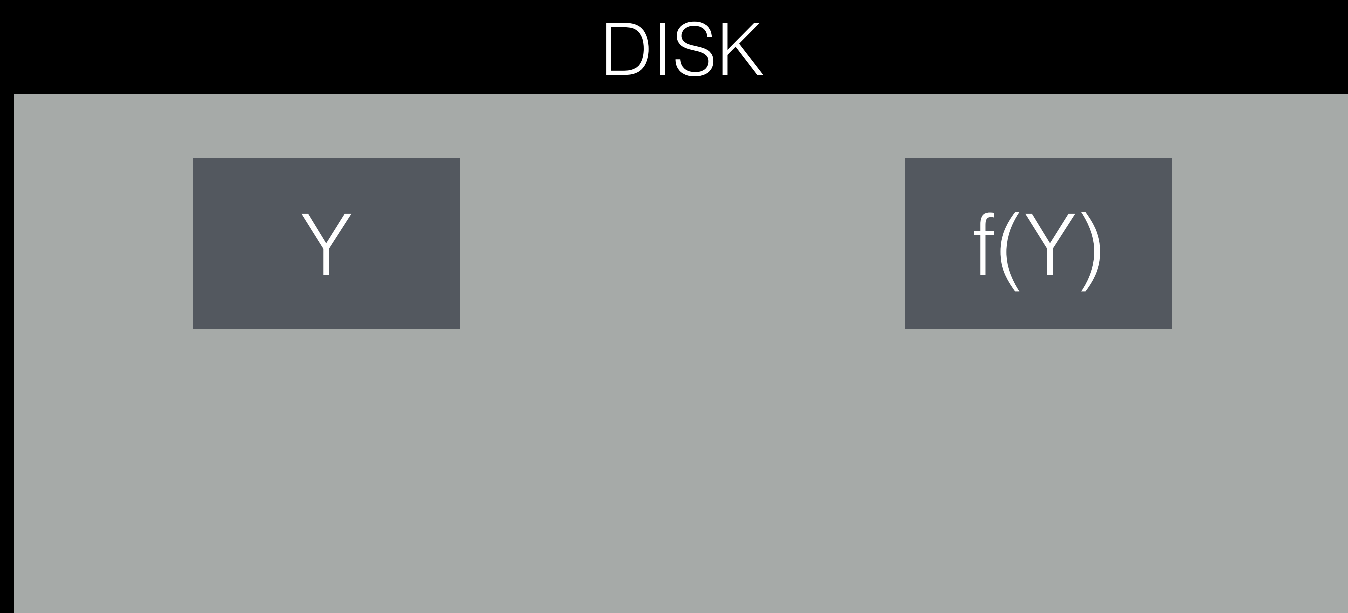
Want to replace X with Y. Original:



bad time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:



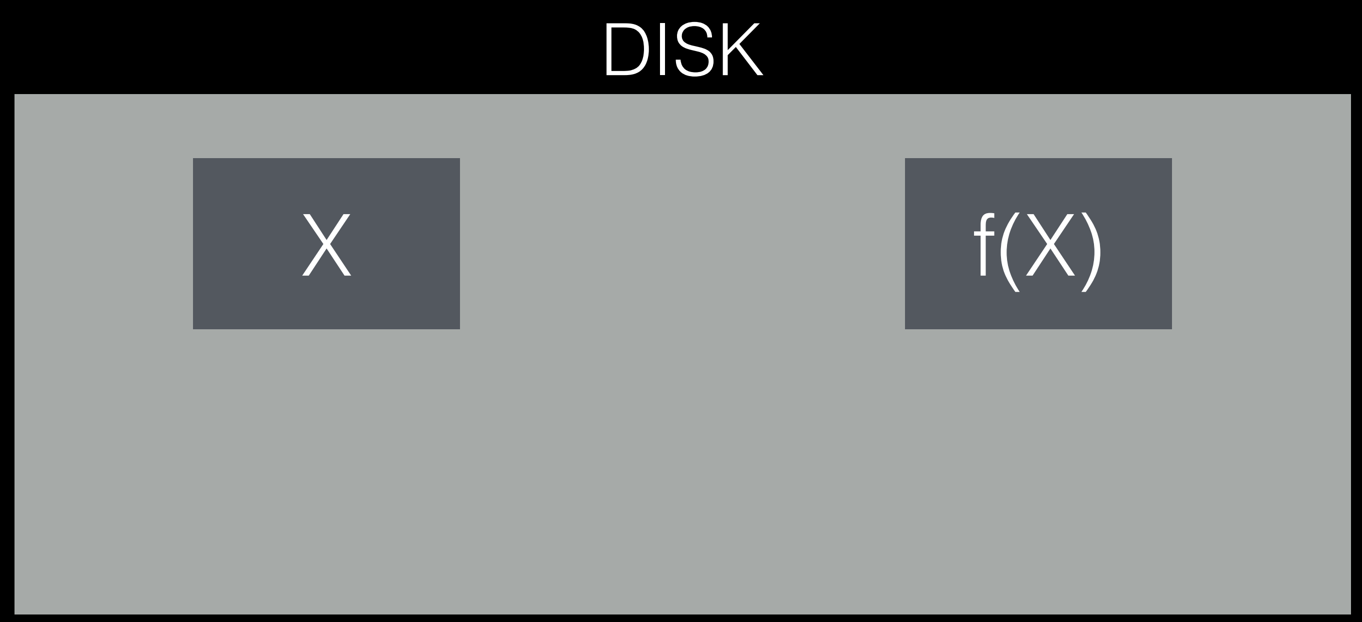
good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y.

Fight Redundancy with Redundancy

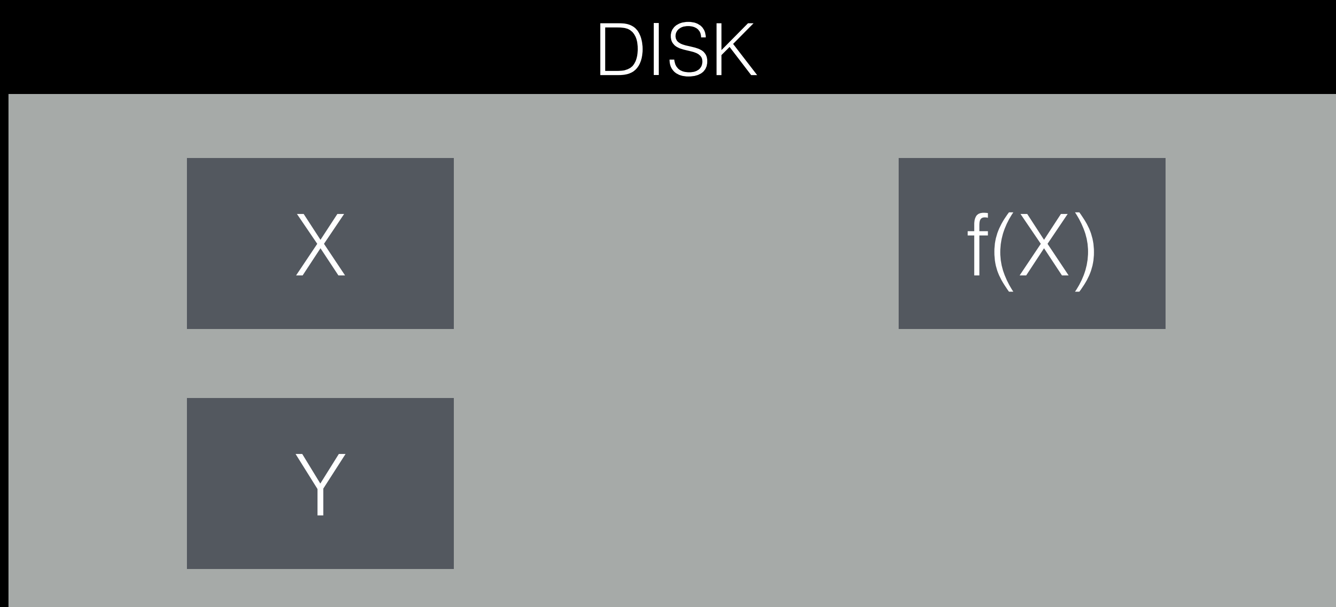
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

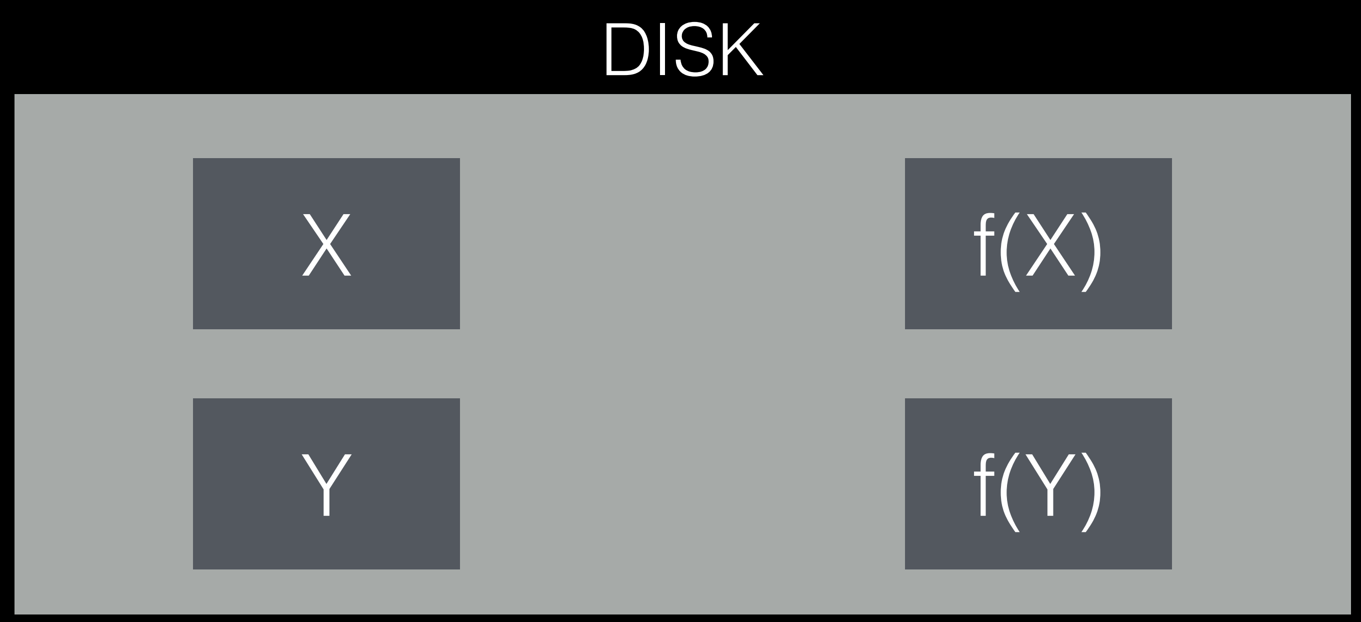
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

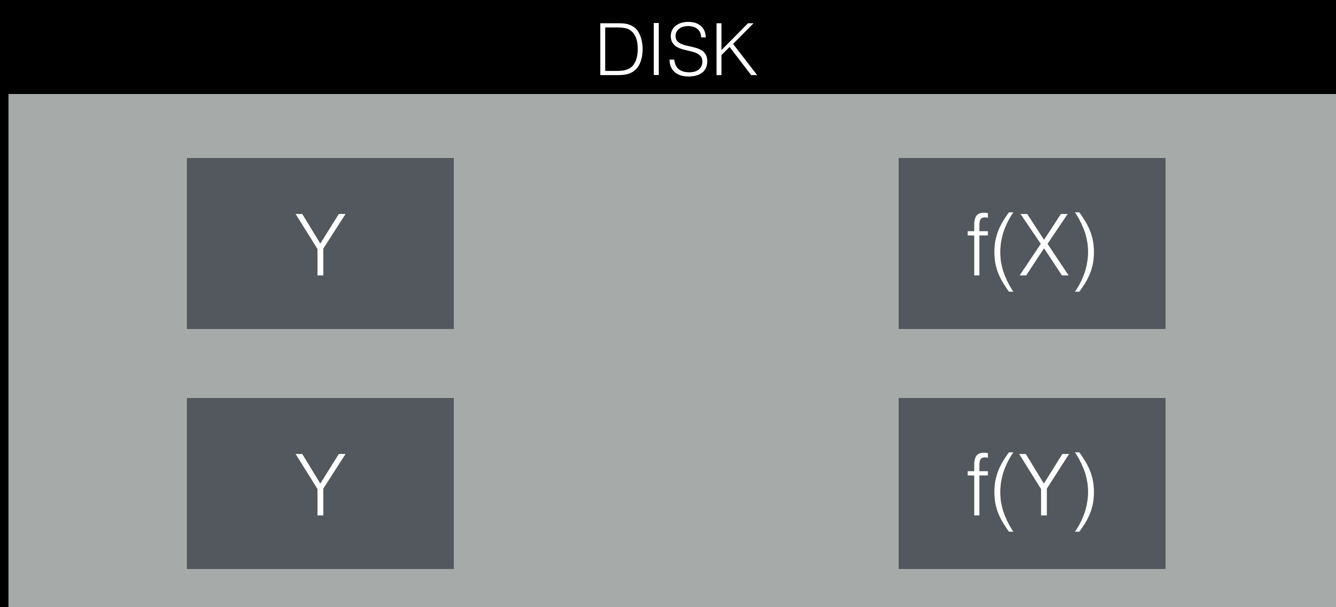
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

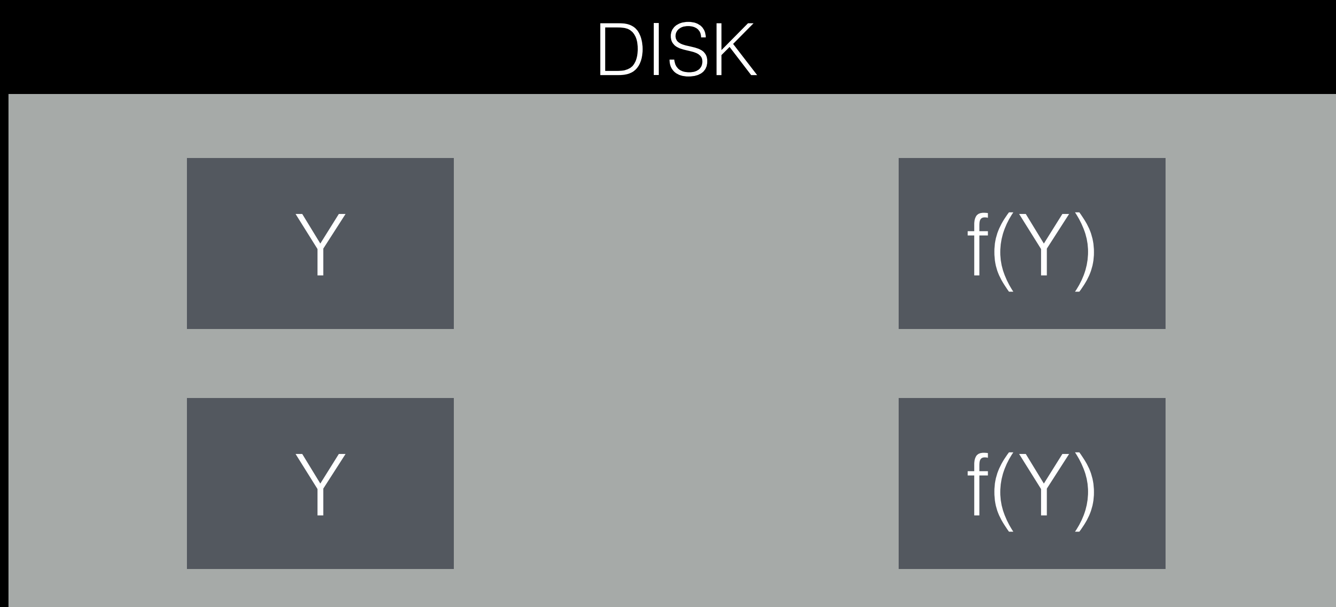
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

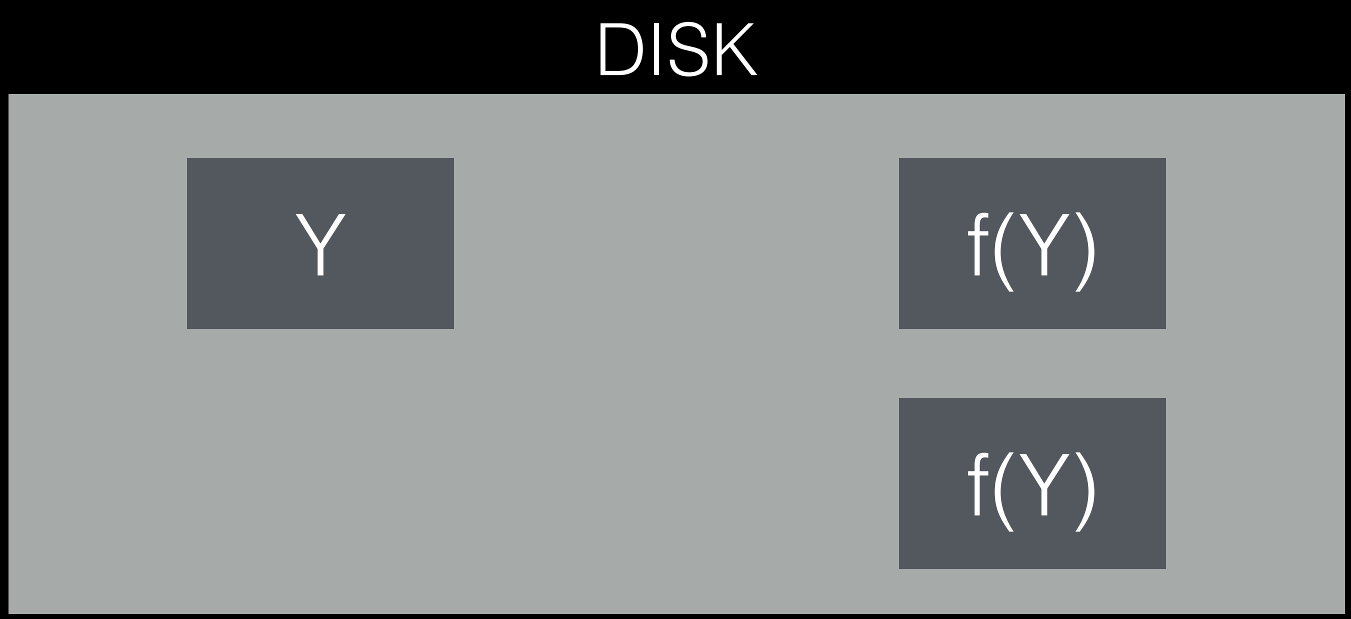
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

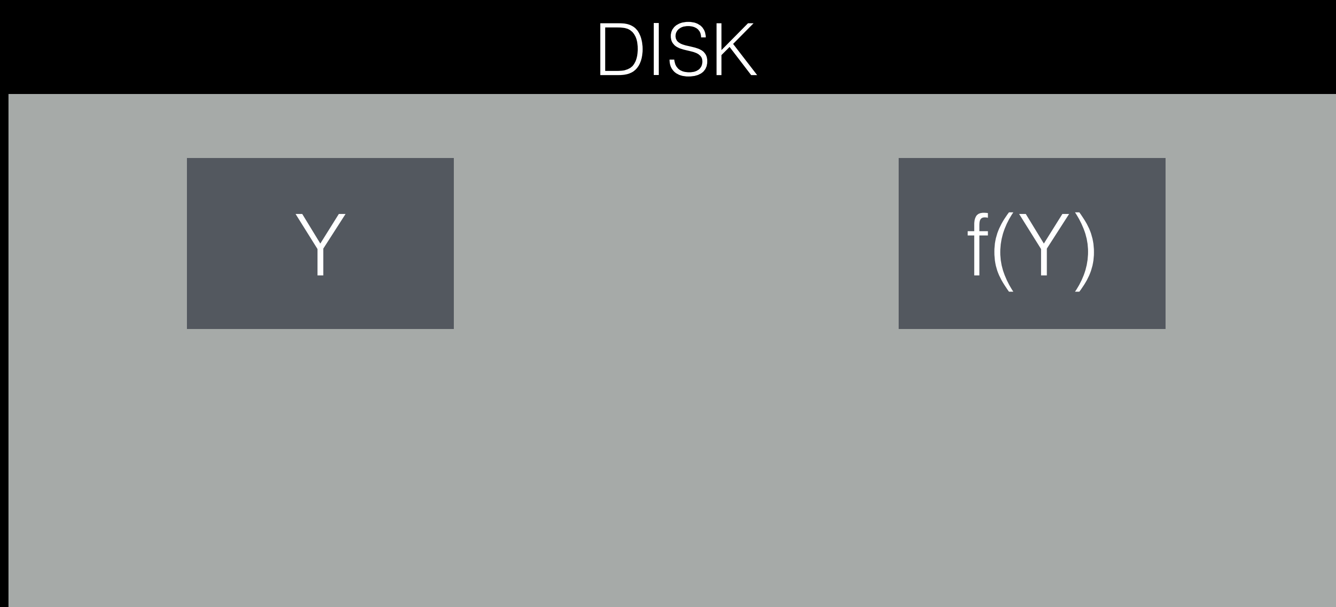
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

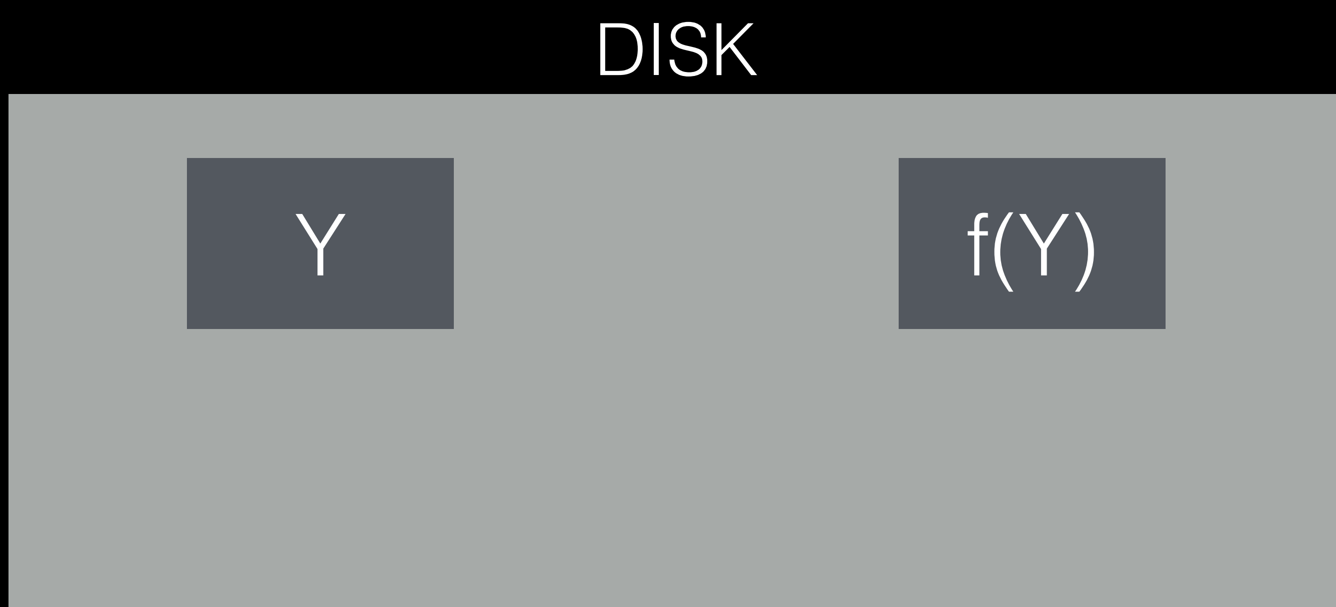
Want to replace X with Y. With journal:



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!

Problem 5

Write an algorithm for a simple case of atomic block update.

Problem 5

Write an algorithm for a simple case of atomic block update. Bad example:

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

Problem 5

Write an algorithm for a simple case of atomic block update. Bad example:

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

don't crash here!

Journal New Data

Time	Block 0: Alice	Block 1: Bob	extra	extra	extra
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

```
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

The extra blocks we use are called a “journal”.

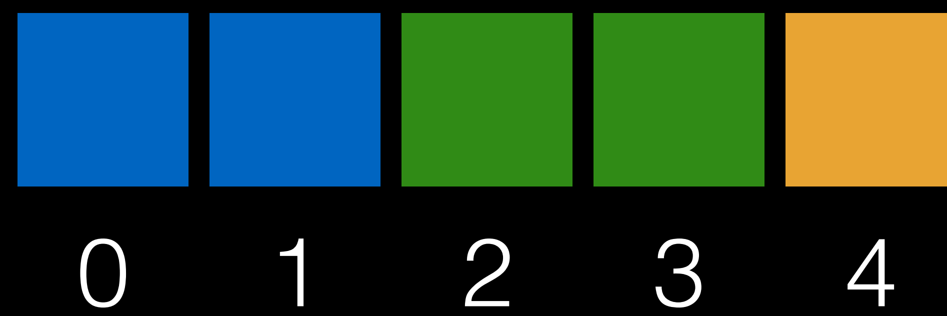
The writes to it are a “journal transaction”.

The last block where we write the valid bit is called a “journal commit block”.

File systems typically write new data to the journal.

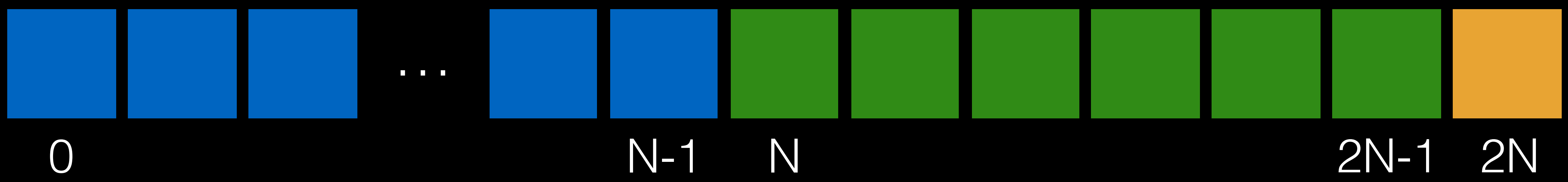
Small Disk

What if we want to use a larger disk?



Big Disk

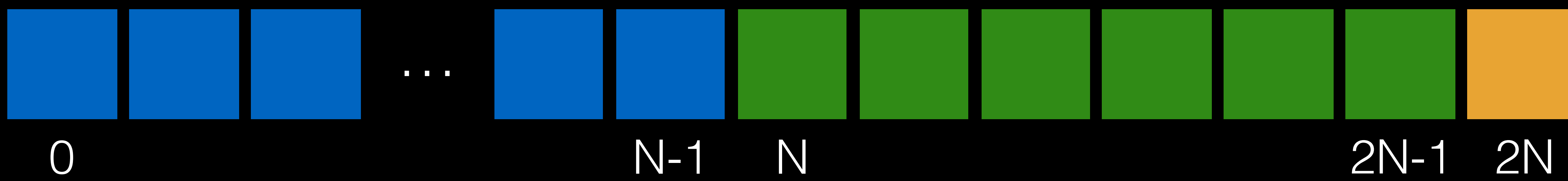
What if we want to use a larger disk?



Disadvantages?

Big Disk

What if we want to use a larger disk?



Disadvantages?

- slightly <half of spaces is usable
- transactions copy all the data

Small Journals

Still need to write all the new data elsewhere first.

Nice if we could use a small area for journalling, but it could be used as backup for any blocks.

How?

Small Journals

Still need to write all the new data elsewhere first.

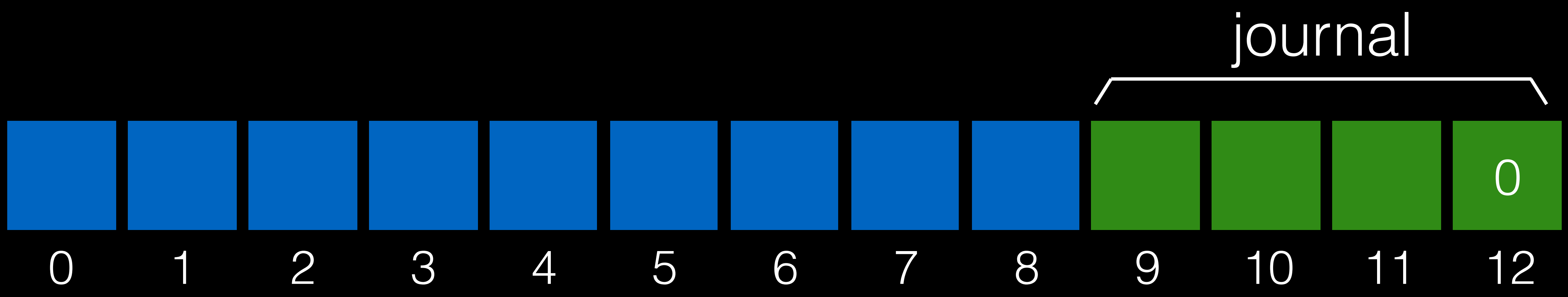
Nice if we could use a small area for journalling, but it could be used as backup for any blocks.

How?

Store block numbers in a transaction header.

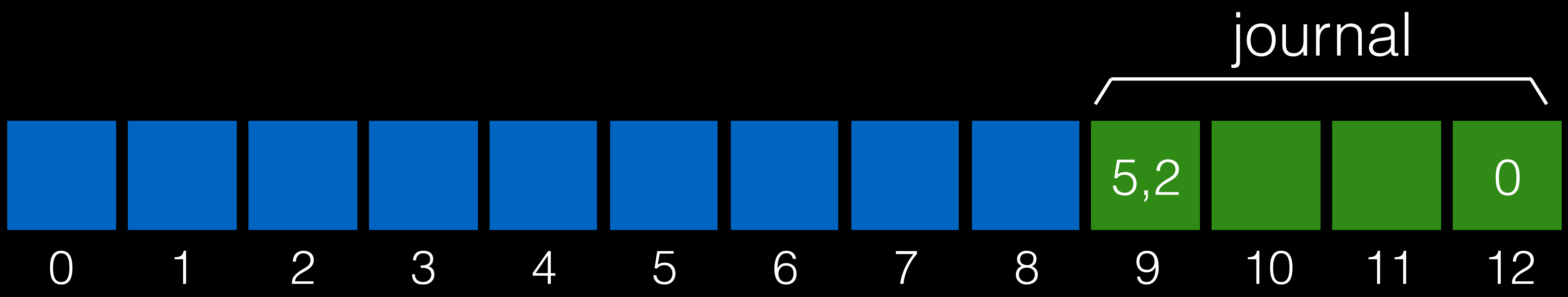
New Layout

New Layout



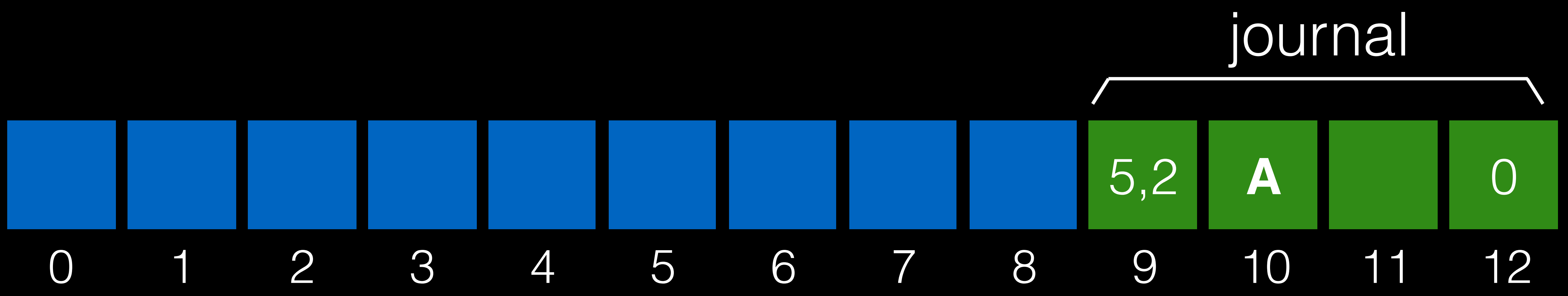
transaction: write A to block 5; write B to block 2

New Layout



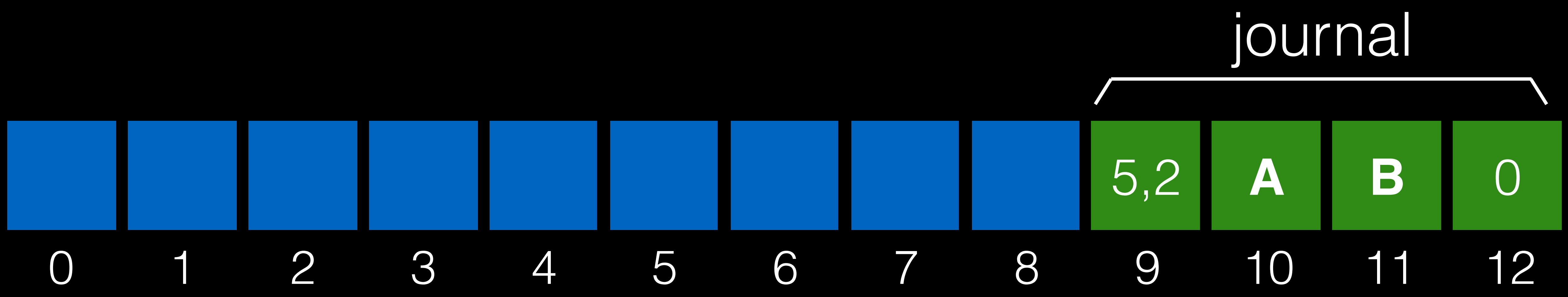
transaction: write A to block 5; write B to block 2

New Layout



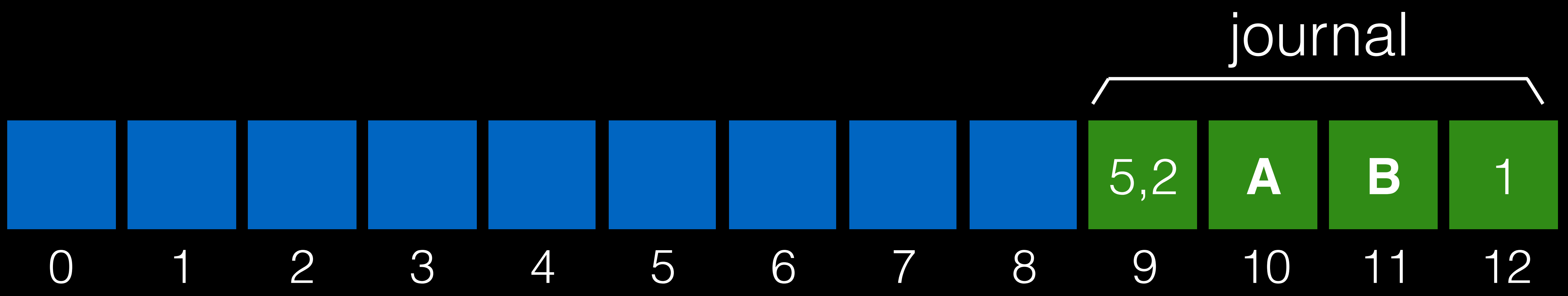
transaction: write A to block 5; write B to block 2

New Layout



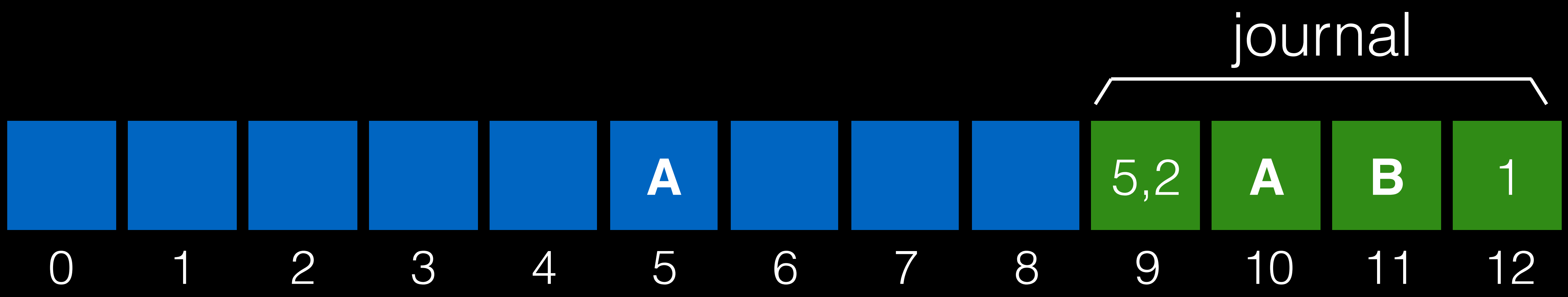
transaction: write A to block 5; write B to block 2

New Layout



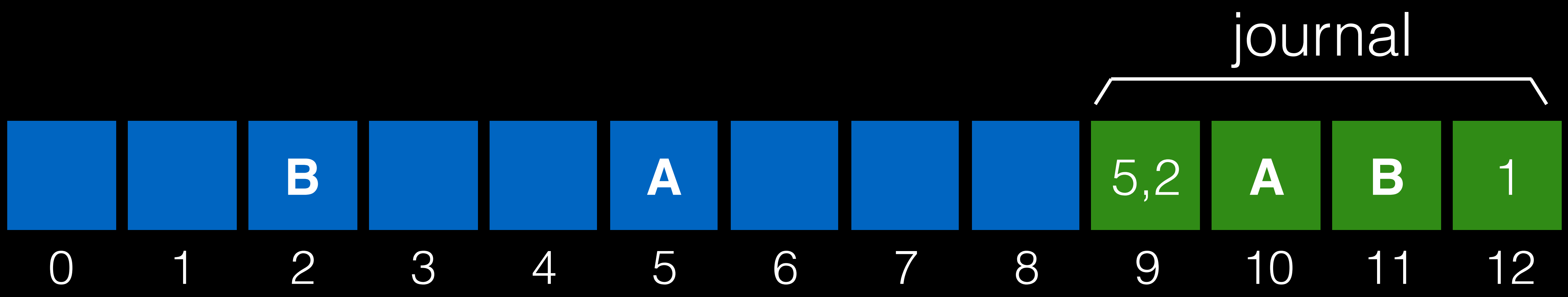
transaction: write A to block 5; write B to block 2

New Layout



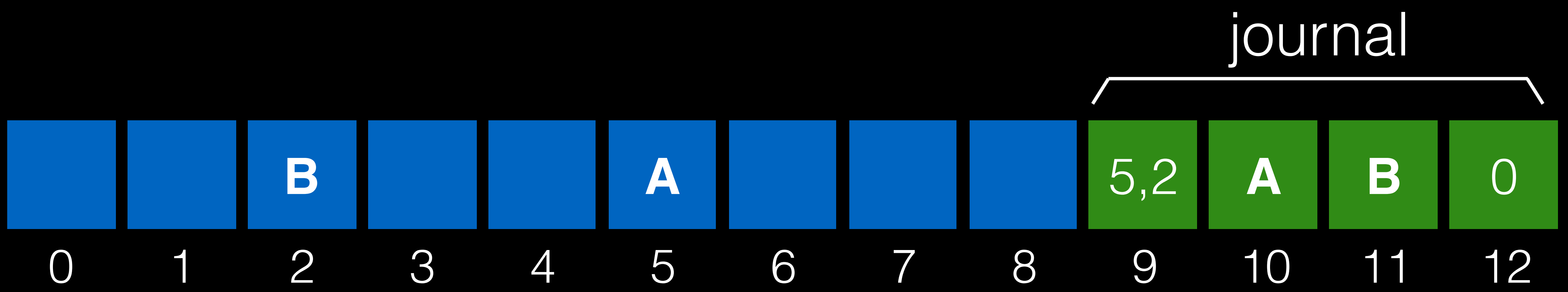
transaction: write A to block 5; write B to block 2

New Layout

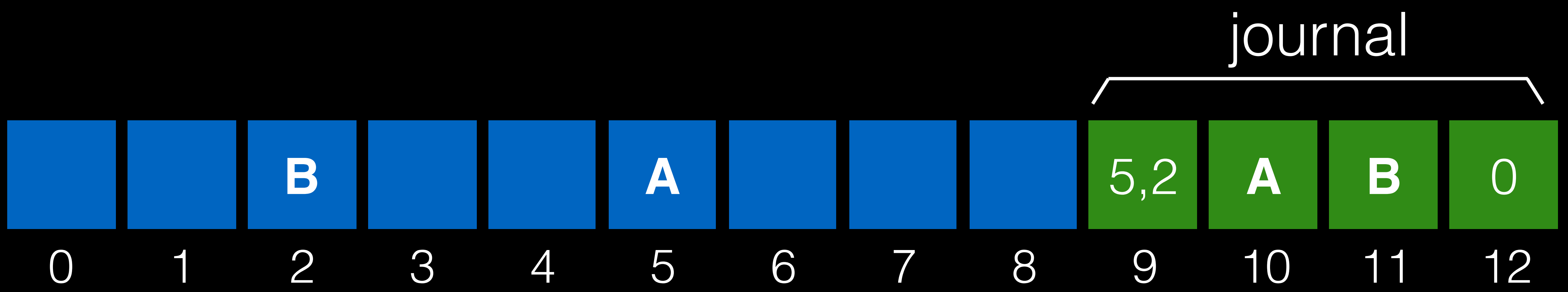


transaction: write A to block 5; write B to block 2

New Layout

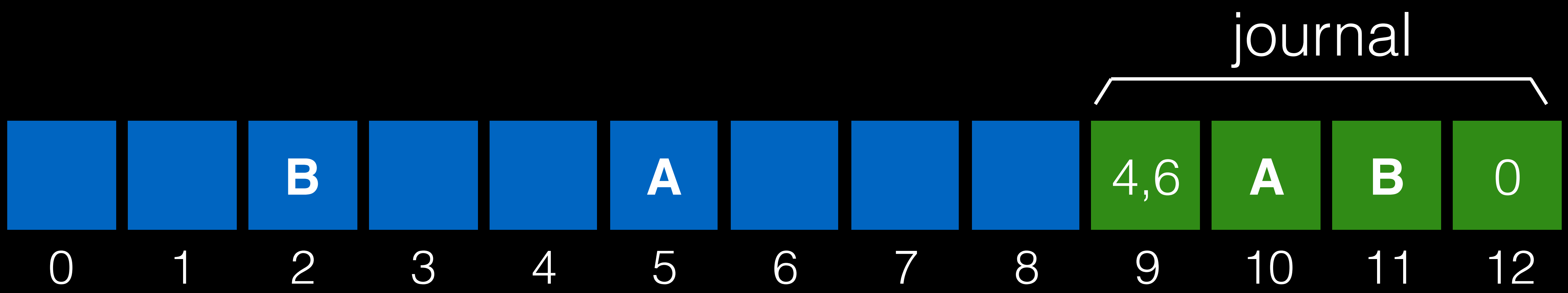


New Layout



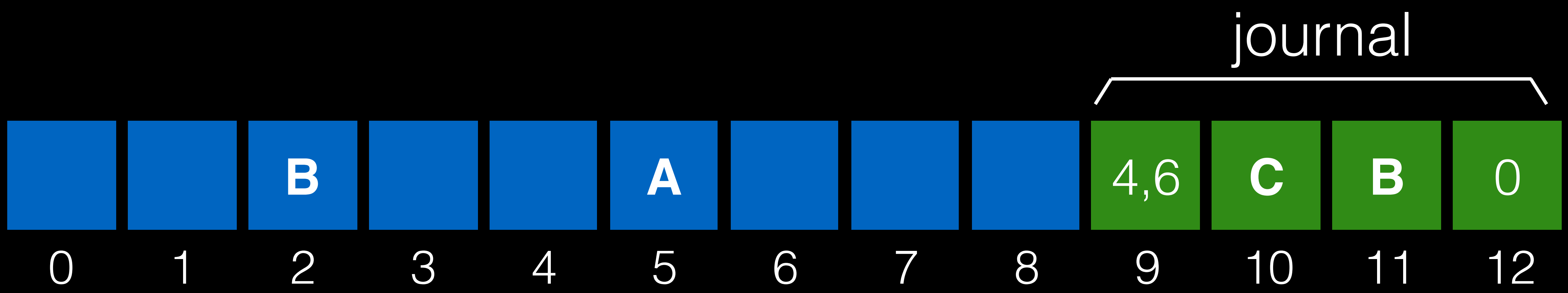
transaction: write C to block 4; write T to block 6

New Layout



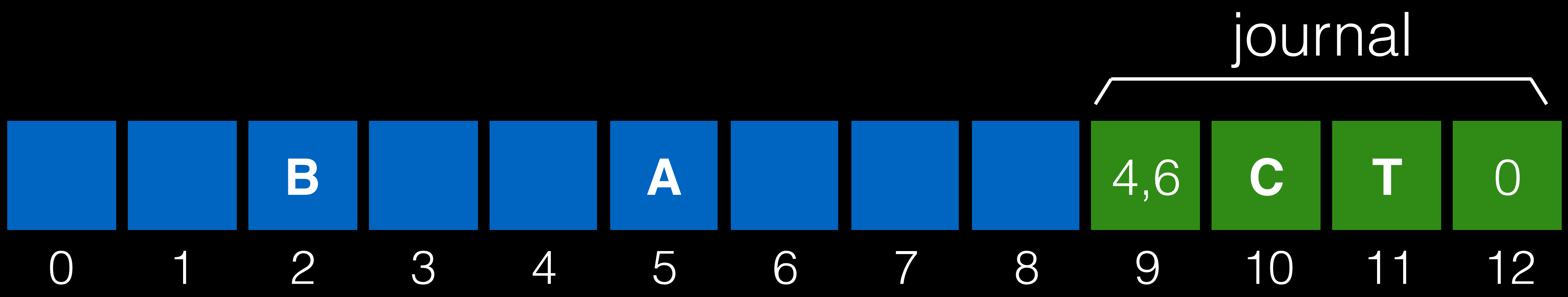
transaction: write C to block 4; write T to block 6

New Layout



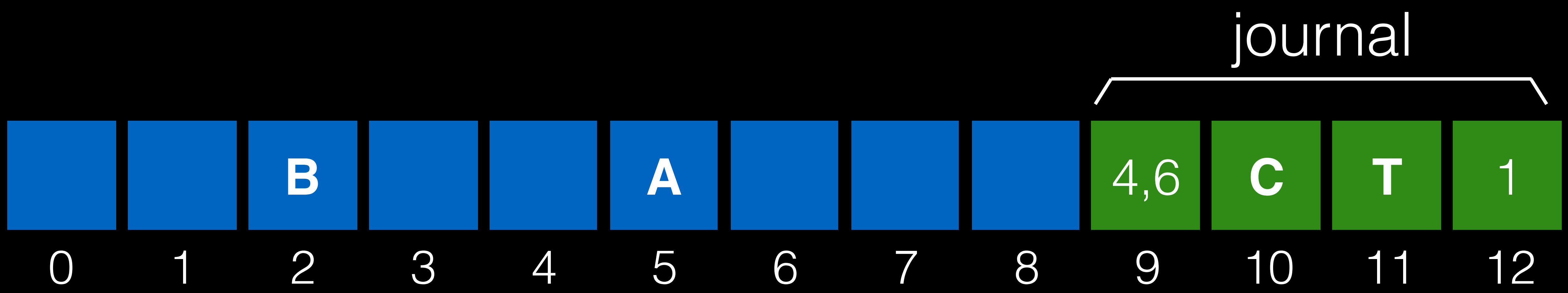
transaction: write C to block 4; write T to block 6

New Layout



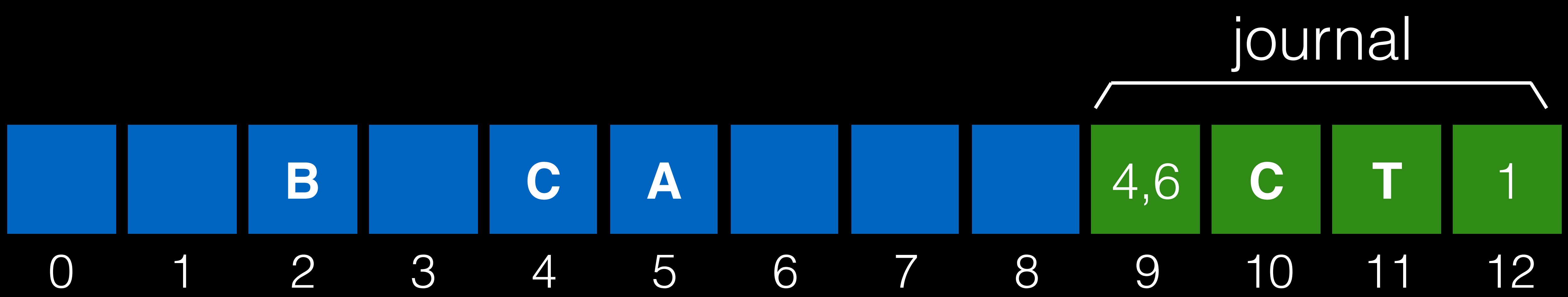
transaction: write C to block 4; write T to block 6

New Layout



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

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

Optimizations

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

Ordering

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

Ordering

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

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

Ordering

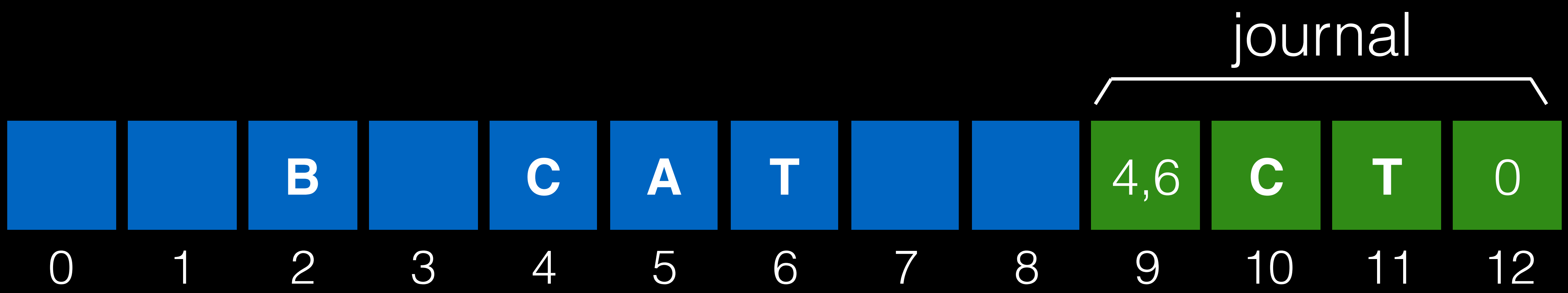
Enforcing total ordering is inefficient. Why?

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

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

Ordering

Use barriers at key points in time. Barrier does cache flush.



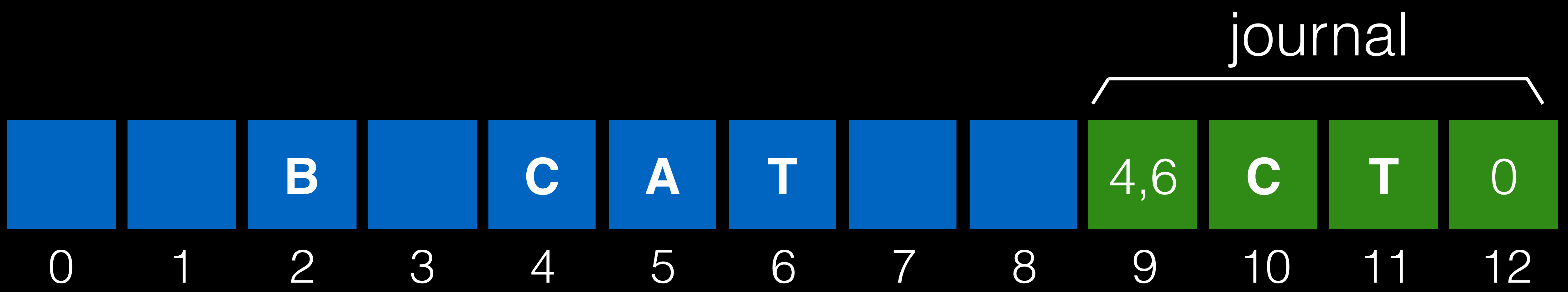
transaction: write C to block 4; write T to block 6

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

Optimizations

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

Checksum



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

Checksum

In last transaction block, store checksum of rest of transaction.

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

Optimizations

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

Write Buffering

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

Journaling is sequential, checkpointing is random.

Solution? Delay checkpointing for some time.

Write Buffering

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

Journaling is sequential, checkpointing is random.

Solution? Delay checkpointing for some time.

Difficulty: need to reuse journal space.

Write Buffering

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

Journaling is sequential, checkpointing is random.

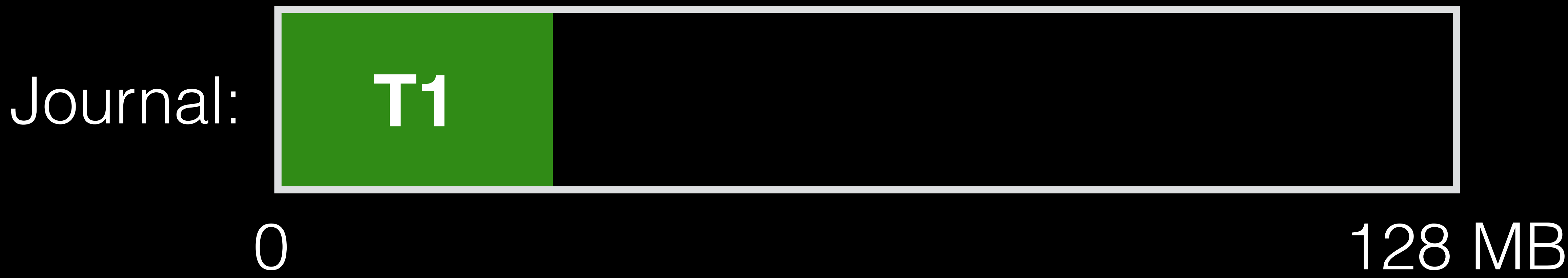
Solution? Delay checkpointing for some time.

Difficulty: need to reuse journal space.

Solution: keep many transactions for un-checkpointed data.

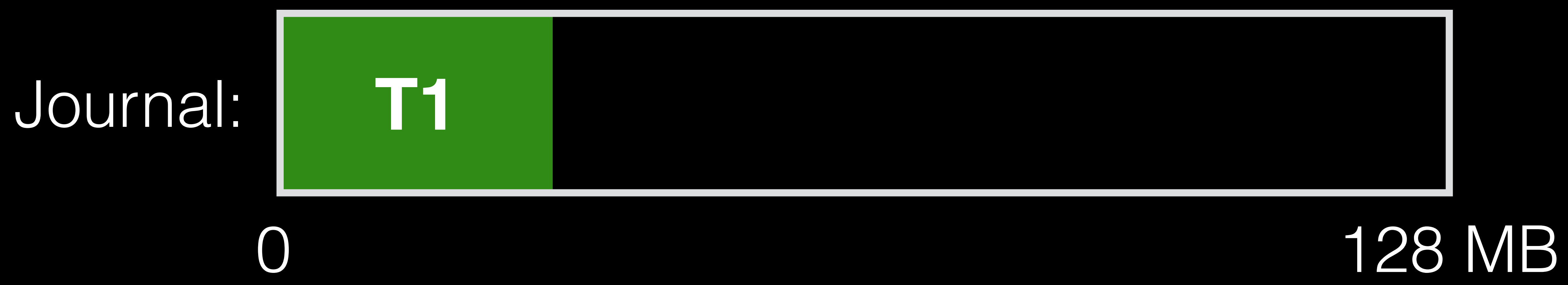
Circular Buffer

Circular Buffer



transaction!

Circular Buffer



Circular Buffer

transaction!

Circular Buffer

Circular Buffer

transaction!

Circular Buffer

Circular Buffer

transaction!

Circular Buffer

Circular Buffer

checkpoint and cleanup

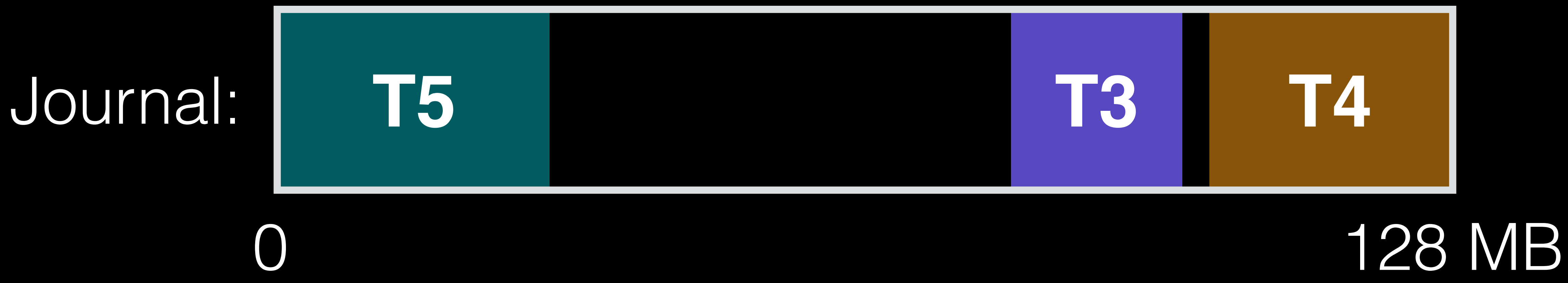
Circular Buffer

Circular Buffer

transaction!

Circular Buffer

Circular Buffer



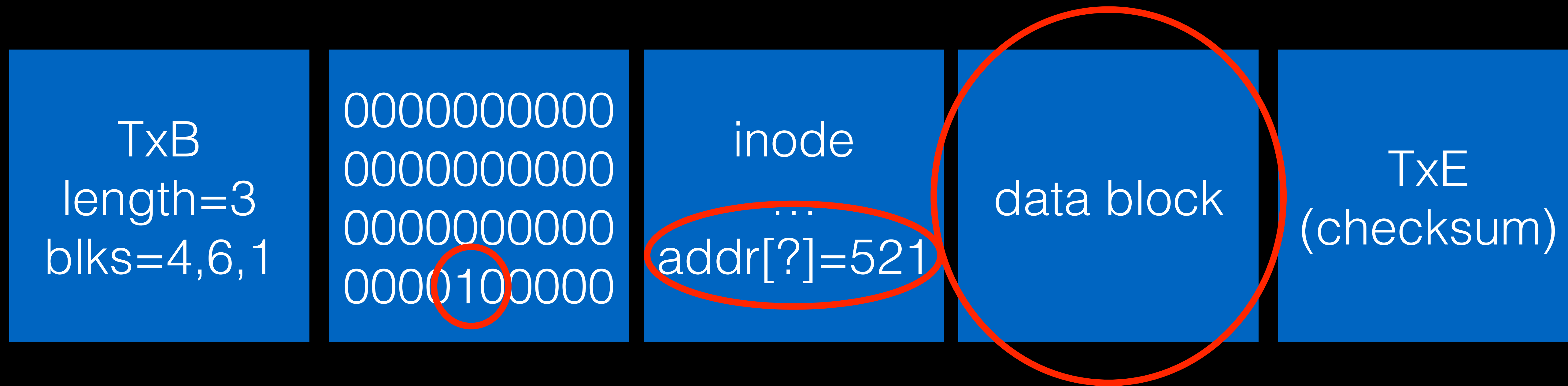
checkpoint and cleanup

Optimizations

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

Physical Journal

Physical Journal



Changes

Logical Journal

Logical journals record changes to bytes, not changes to blocks.

Optimizations

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

File System Integration

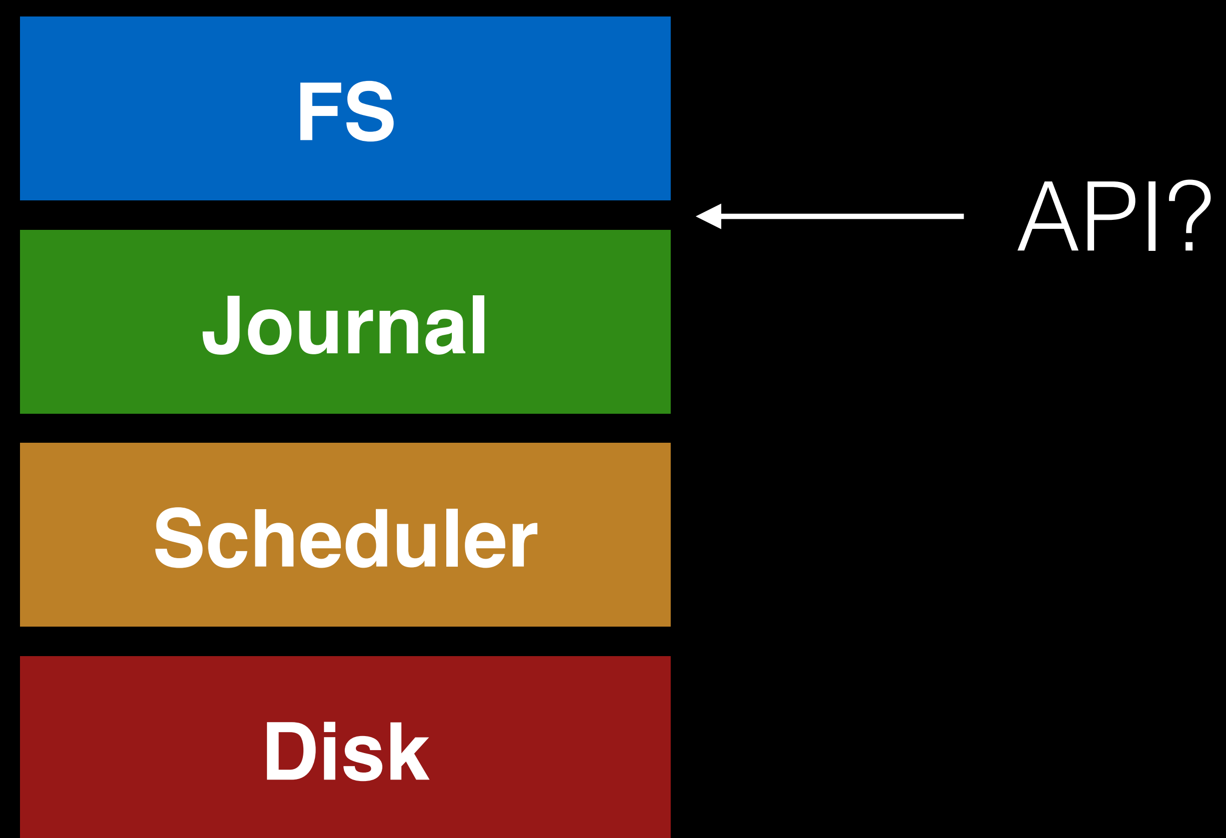
How should FS use journal?

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() ← which calls must be atomic?

readBlk()

flush()

Handle API

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

Handle API

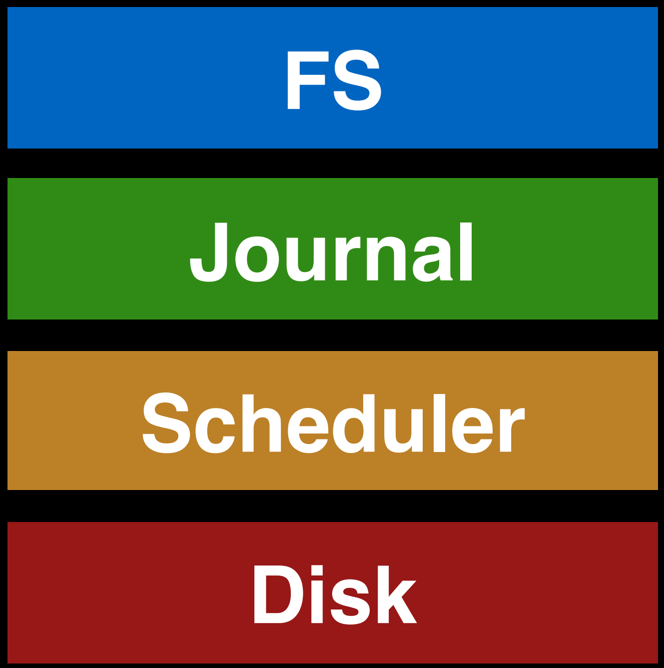
```
h = getHandle();
writeBlk(h, blknum, data);
finishHandle(h);
```

Blocks in the same handle must be written atomically.

File System Integration

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

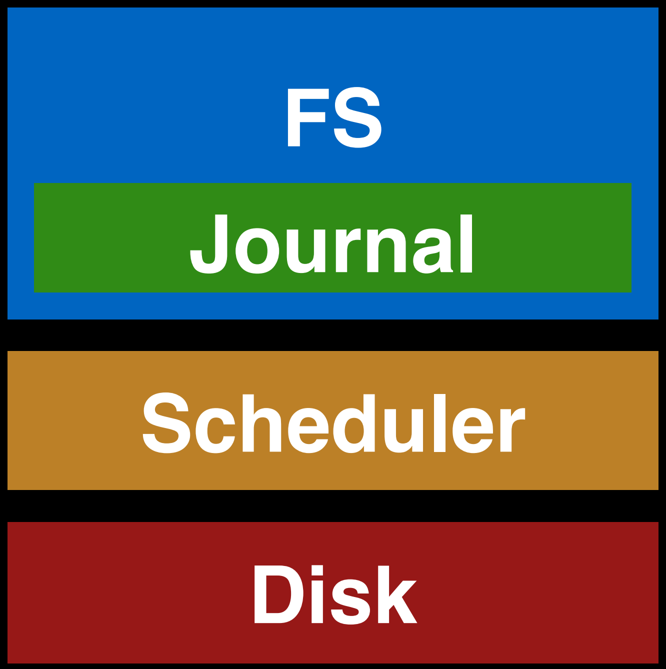
If we want to only journal FS metadata, we need tighter integration.



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.



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?

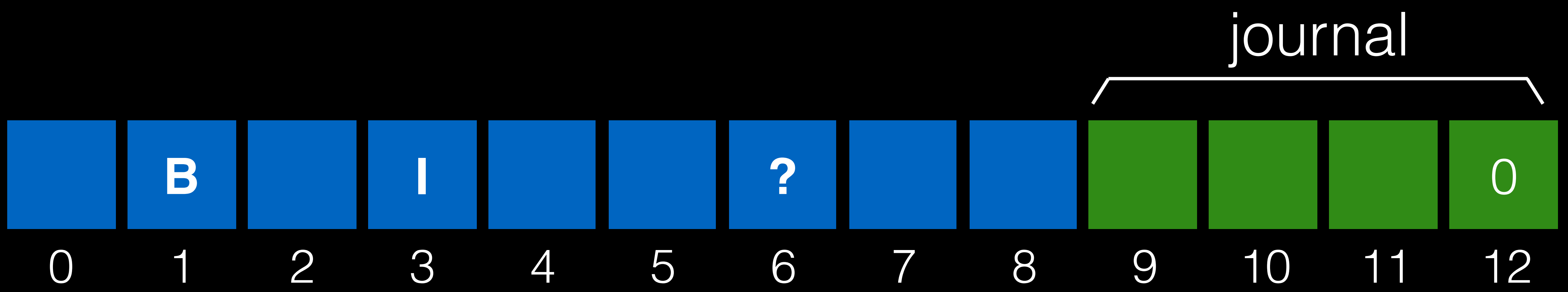
Writeback Journal

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.

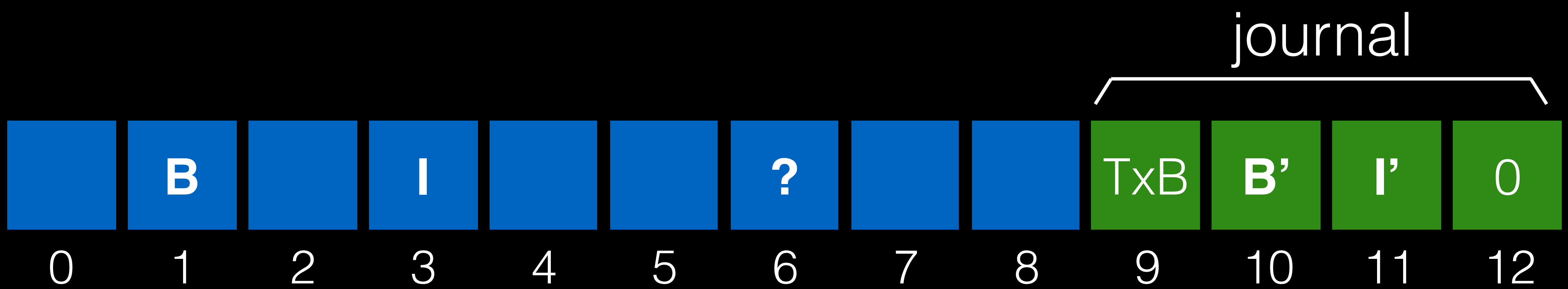
What is the worst type of garbage we could get? How to avoid?

Writeback Journal



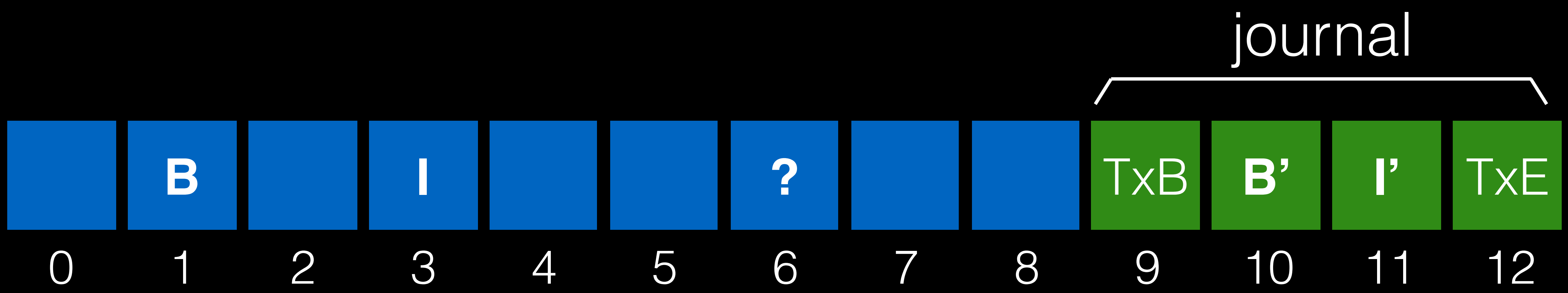
transaction: append to inode I

Writeback Journal



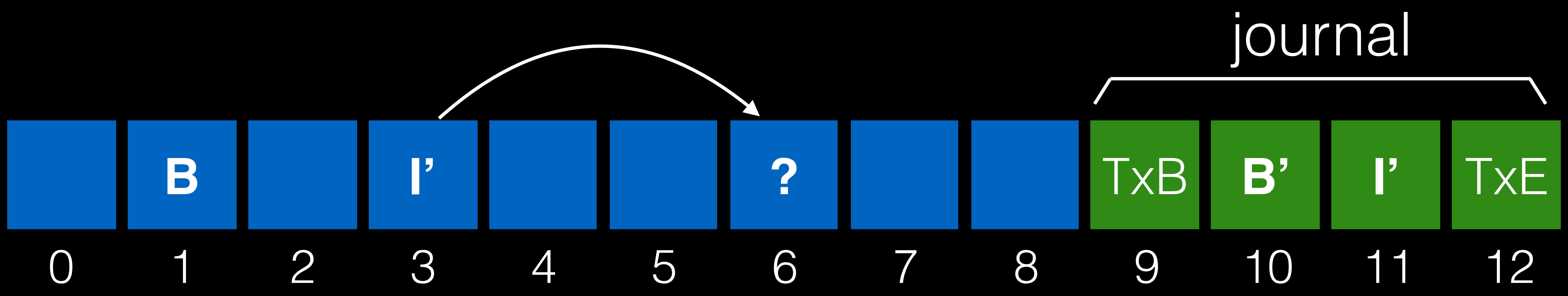
transaction: append to inode I

Writeback Journal



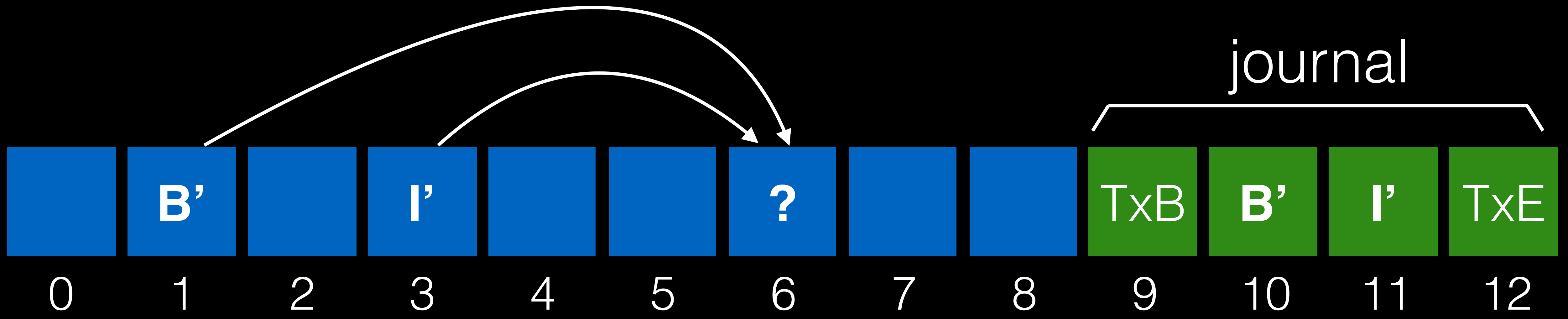
transaction: append to inode I

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.

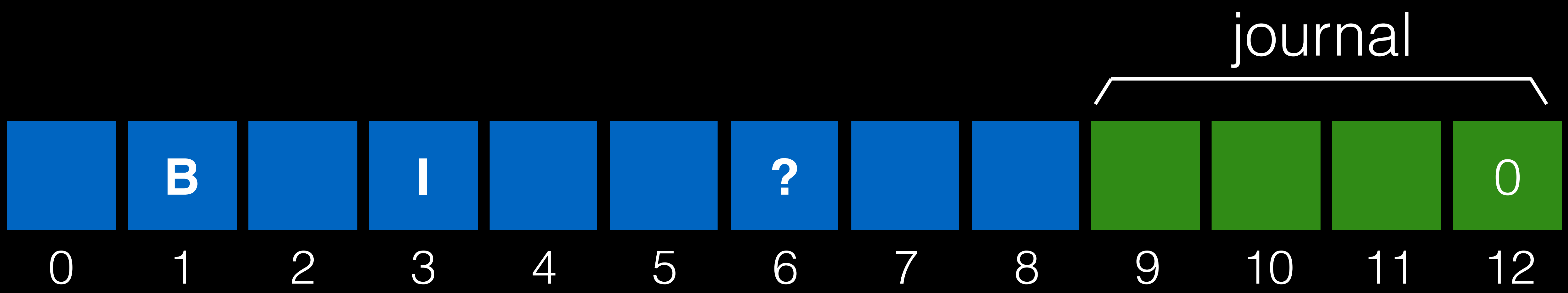
But write data before the transaction.

May still get scrambled data on update.

But appends will always be good.

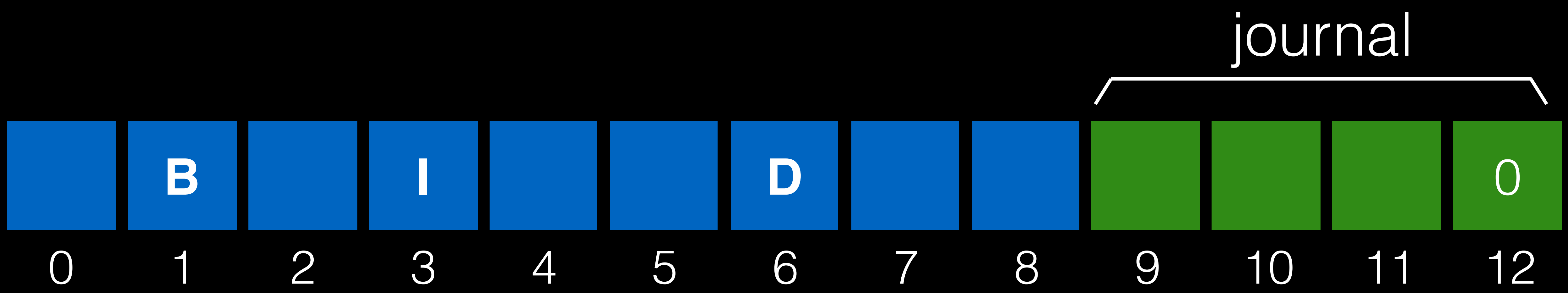
No leaks of sensitive data!

Ordered Journal



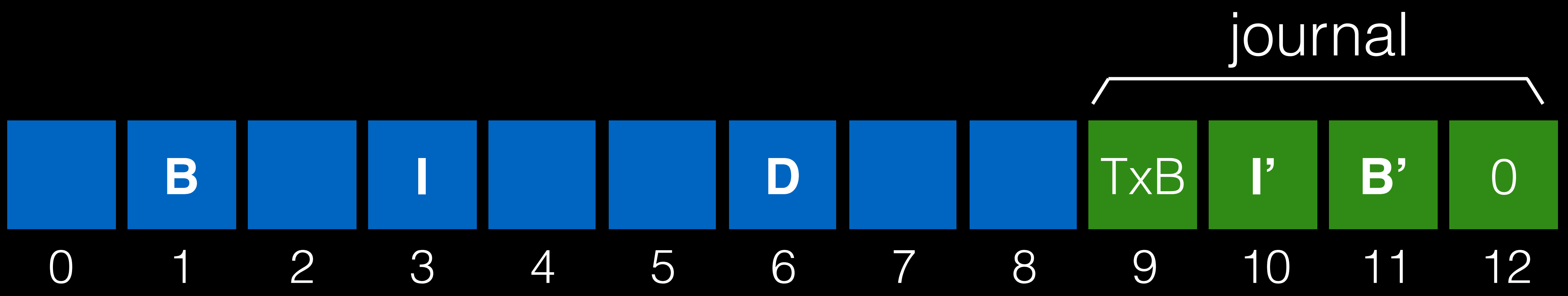
transaction: append to inode I

Ordered Journal



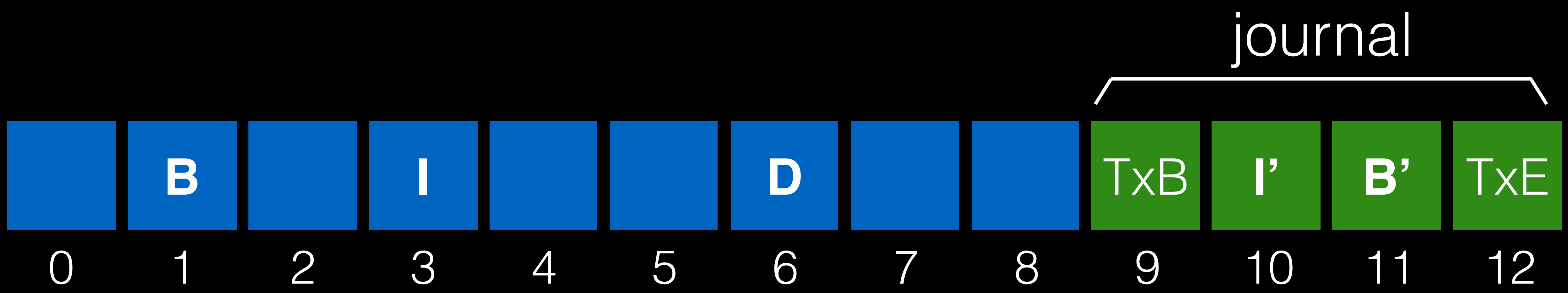
transaction: append to inode I

Ordered Journal



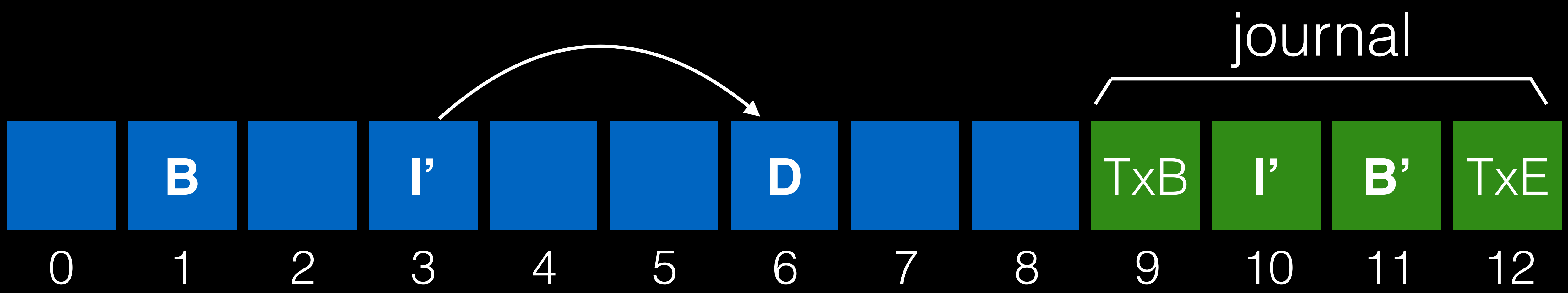
transaction: append to inode I

Ordered Journal



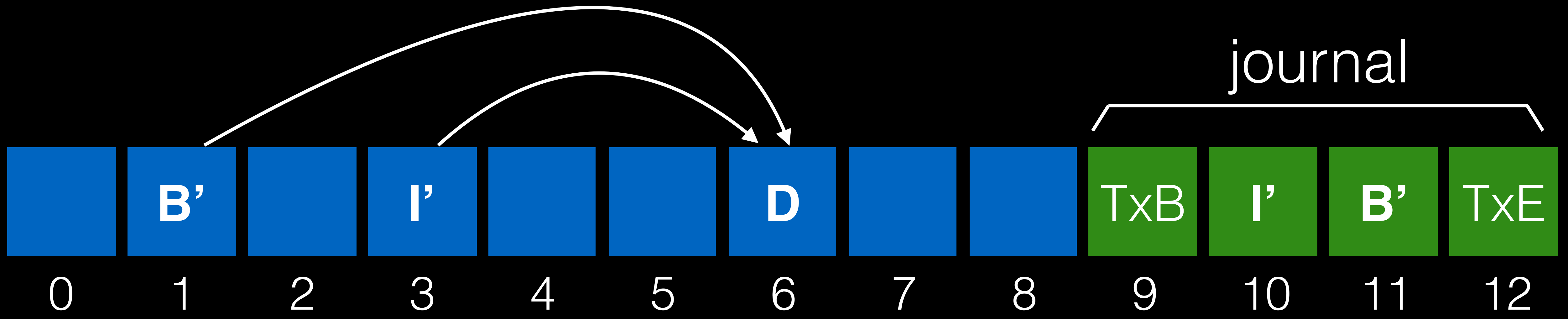
transaction: append to inode I

Ordered Journal



transaction: append to inode I

Ordered Journal



transaction: append to inode I

Announcements

Exam this Friday
- 7-9pm, CHEM 1351 (same as last time)
- 1 sheet notes
- Chapters 30 to 41 (inclusive)

Review today
- 7-9pm, room CS 1221. Bring questions.

No regular **discussion** this week.

Office hours
- 1pm today, in office
- 2:30 - 3:45pm tomorrow, in lab

Conclusion

Most modern file systems use journals.

FSCK is still useful for weird cases
- bit flips
- FS bugs

Some file systems don't use journals, but they still (usually) must write new data before deleting old.