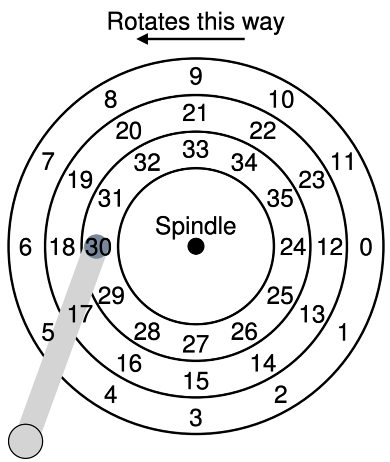
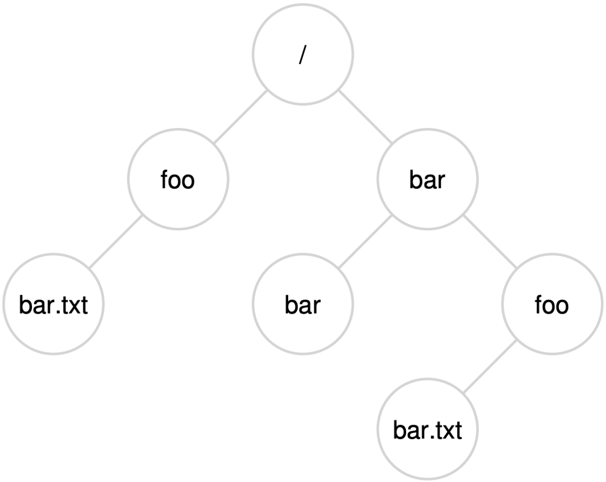
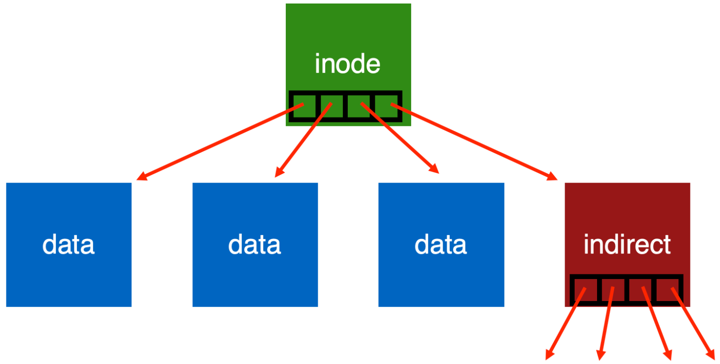
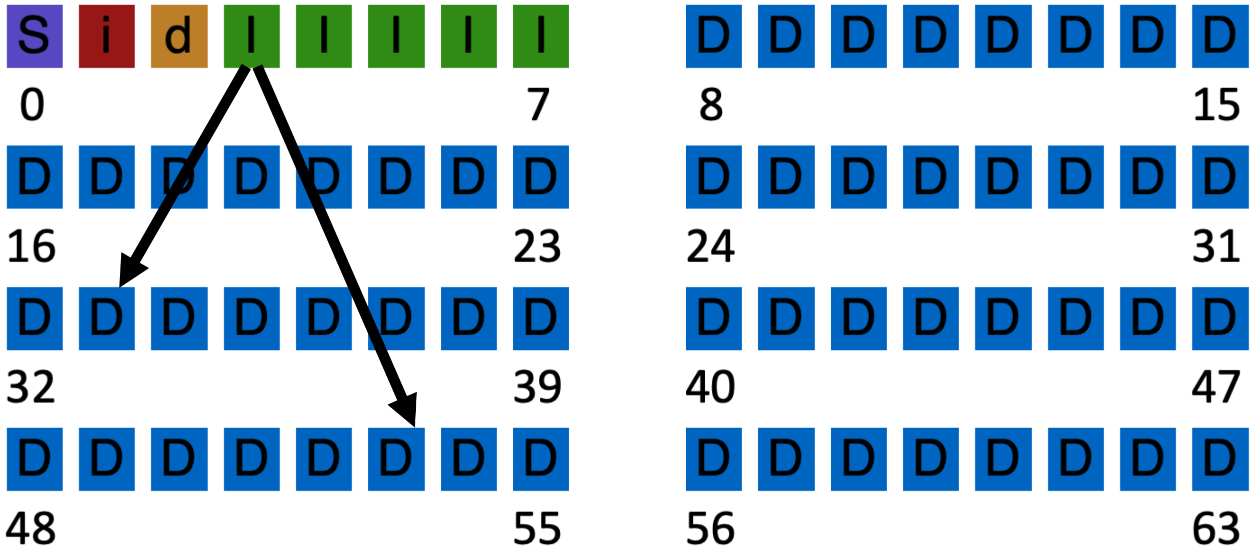
Persistence





valid	name	inode
1	.	134
1	..	35
1	foo	80
1	bar	23





create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read write					
						write
				read write		
			write			

Efficiency

How can we avoid this excessive I/O for basic ops?

Cache for:

- reads
- write buffering

Write Buffering

Why does procrastination help?

Overwrites, deletes, scheduling

Shared structs (e.g., bitmaps+dirs) often overwritten.

We decide: how much to buffer, how long to buffer…
 - tradeoff durability vs. performance

How to allocate file data to disk blocks?

Disk layout of data matters!

- Why?
- Positioning latency: disk rotation; seek
- Sequential reads are faster than random reads

Allocation Strategies

Many different approaches

- Contiguous
- Extent-based
- Linked
- File-allocation Tables
- Indexed
- Multi-level Indexed

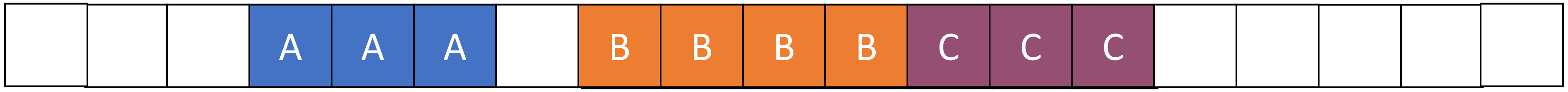
Questions

- Amount of fragmentation (internal and external)
  – free space that can't be used
- Ability to grow file over time?
- Performance of sequential accesses (contiguous layout)?
- Speed to find data blocks for random accesses?
- Wasted space for meta-data overhead (everything that isn't data)?
  - Meta-data must be stored persistently too!

Contiguous Allocation

Allocate each file to contiguous sectors on disk

- Meta-data: Starting block and size of file
- OS allocates by finding sufficient free space
  - Must predict future size of file; Should space be reserved?
- Example: IBM OS/360

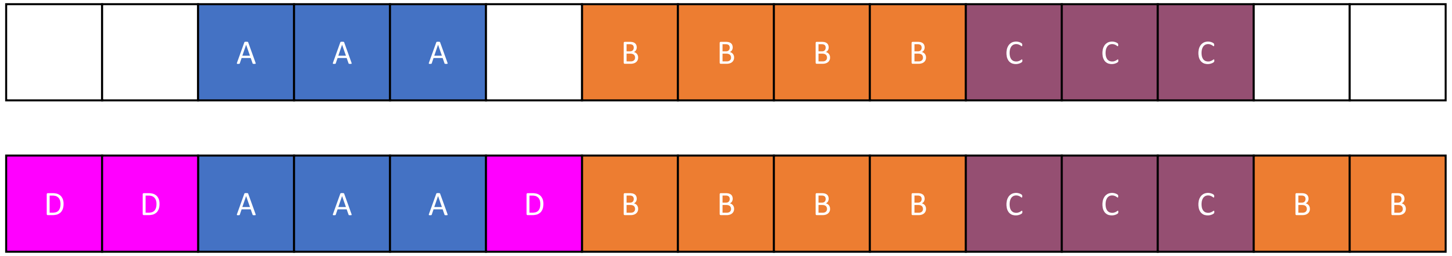


Fragmentation (internal and external)?	- Horrible external frag (needs periodic compaction)
Ability to grow file over time?	- May not be able to without moving
Seek cost for sequential accesses?	+ Excellent performance
Speed to calculate random accesses?	+ Simple calculation
Wasted space for meta-data?	+ Little overhead for meta-data

Small # of Extents

Allocate multiple contiguous regions (extents) per file

- Meta-data: Small array (2-6) designating each extent
  Each entry: starting block and size



Fragmentation (internal and external)?
- Helps external fragmentation

Ability to grow file over time?
- Can grow (until run out of extents)

Seek cost for sequential accesses?
+ Still good performance

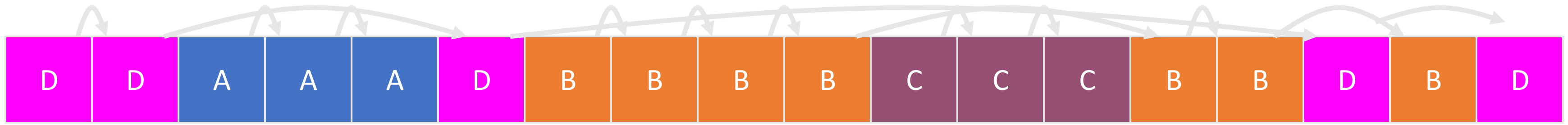
Speed to calculate random accesses?
+ Still simple calculation

Wasted space for meta-data?
+ Still small overhead for meta-data

Linked Allocation

Allocate linked-list of **fixed-sized** blocks (multiple sectors)

- Meta-data: Location of first block of file
  Each block also contains pointer to next block
- Examples: TOPS-10, Alto



Fragmentation (internal and external)? + No external frag (use any block);

Ability to grow file over time? + Can grow easily

Seek cost for sequential accesses? +/- Depends on data layout

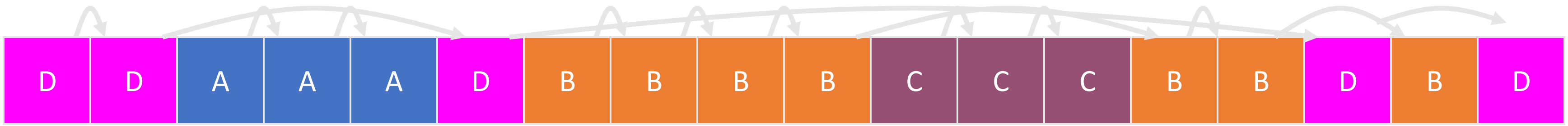
Speed to calculate random accesses? - Ridiculously poor

Wasted space for meta-data? - Waste pointer per block

File-Allocation Table (FAT)

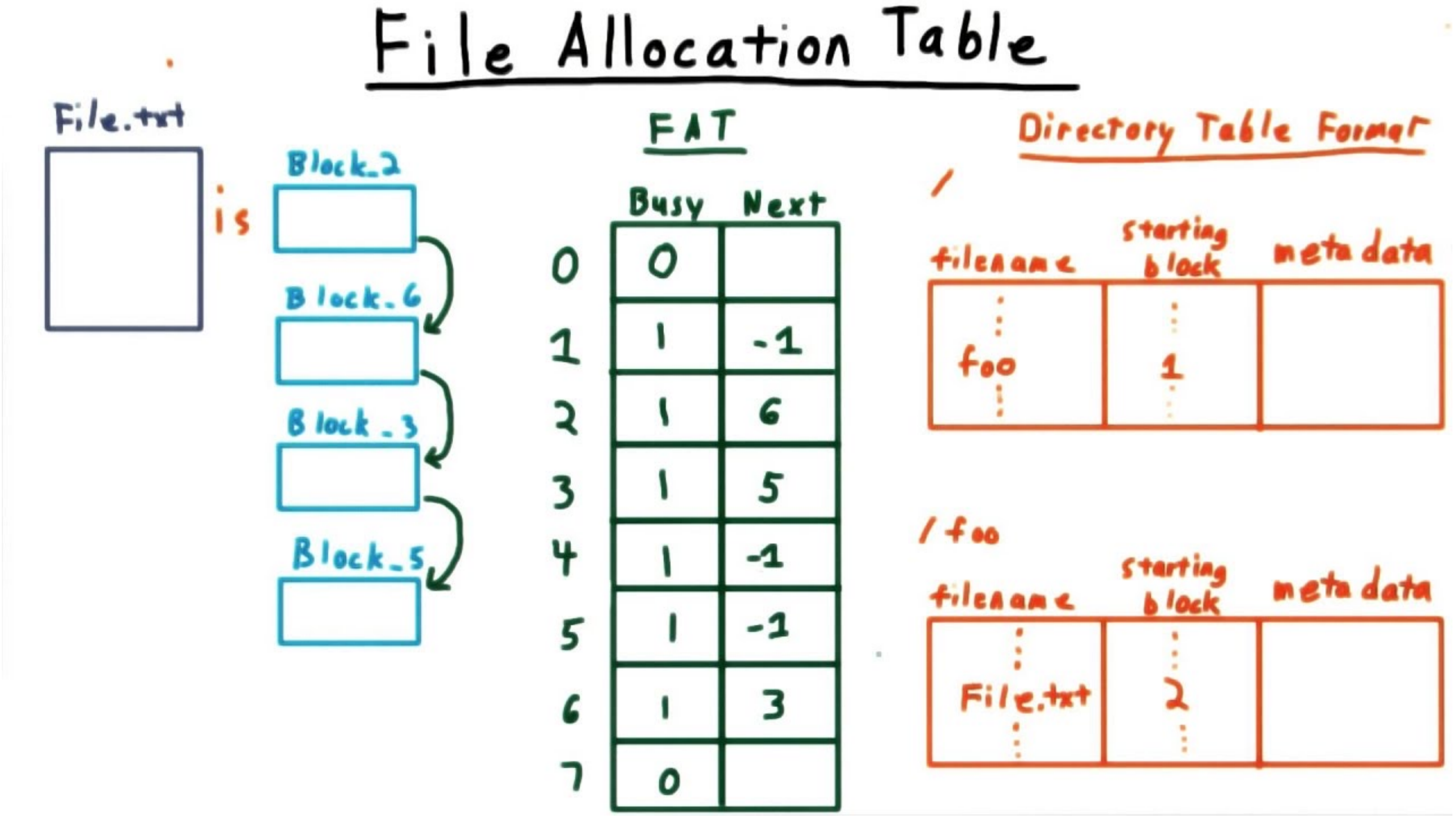
Variation of Linked allocation

- Keep linked-list information for all files in on-disk FAT table
- Meta-data: Location of first block of file
  - And, FAT table itself



Draw corresponding FAT Table?

Example of a FAT

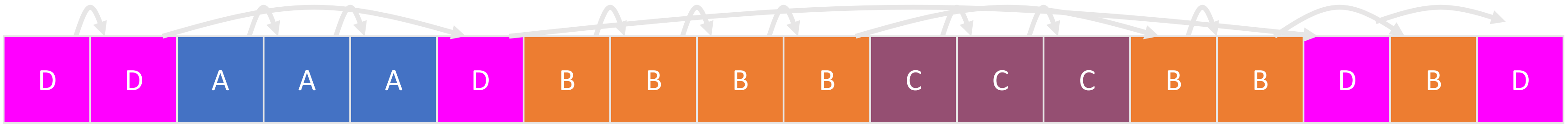


https://www.youtube.com/watch?v=mgQtlXBxH0c

File-Allocation Table (FAT)

Variation of Linked allocation

- Keep linked-list information for all files in on-disk FAT table
- Meta-data: Location of first block of file
  - And, FAT table itself



Draw corresponding FAT Table?

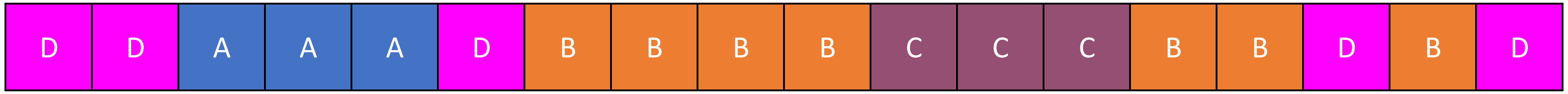
Comparison to Linked Allocation

- Same basic advantages and disadvantages
- Disadvantage: Read from two disk locations for every data read
- Optimization: Cache FAT in main memory
  - Advantage: Greatly improves random accesses
  - What portions should be cached? Scale with larger file systems?

Indexed Allocation

Allocate fixed-sized blocks for each file

- Meta-data: Fixed-sized array of block pointers
- Allocate space for pointers at file creation time



Advantages

- No external fragmentation
- Files can be easily grown up to max file size
- Supports random access

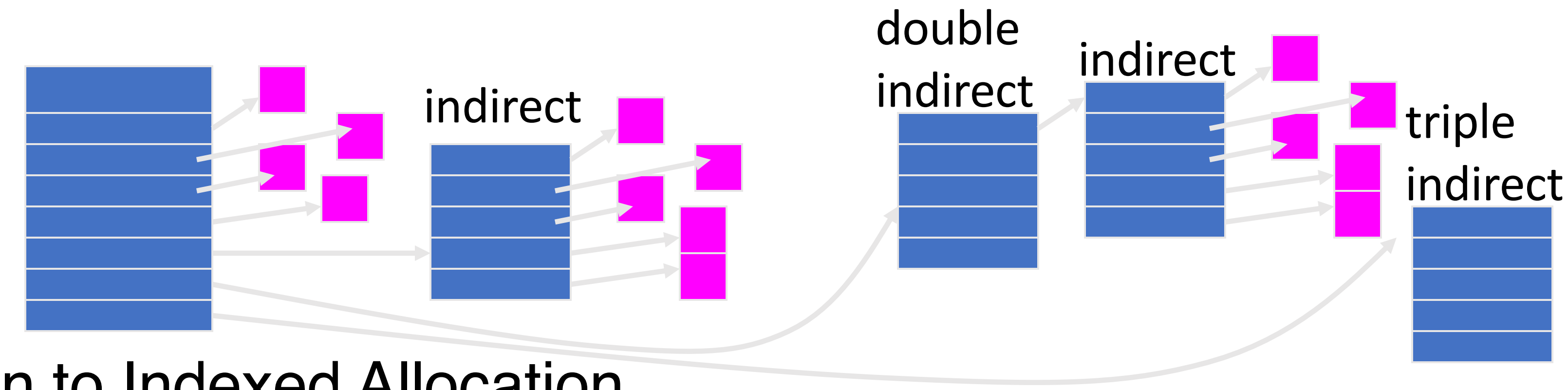
Disadvantages

- Large overhead for meta-data:
  - Wastes space for unneeded pointers (most files are small!)

Multi-Level Indexing

Variation of Indexed Allocation

- Dynamically allocate hierarchy of pointers to blocks as needed
- Meta-data: Small number of pointers allocated statically
  - Additional pointers to blocks of pointers
- Examples: UNIX FFS-based file systems, ext2, ext3



Comparison to Indexed Allocation

- Advantage: Does not waste space for unneeded pointers
  - Still fast access for small files
  - Can grow to what size?
- Disadvantage: Need to read indirect blocks of pointers to calculate addresses (extra disk read)
  - Keep indirect blocks cached in main memory

Flexible # of Extents

Modern file systems:

Dynamic multiple contiguous regions (extents) per file

- Organize extents into multi-level tree structure
  - Each leaf node: starting block and contiguous size
  - Minimizes meta-data overhead when have few extents
  - Allows growth beyond fixed number of extents

Fragmentation (internal and external)?	+ Both reasonable
Ability to grow file over time?	+ Can grow
Seek cost for sequential accesses?	+ Still good performance
Speed to calculate random accesses?	+/- Some calculations depending on size
Wasted space for meta-data?	+ Relatively small overhead

Assume Multi-Level Indexing

Simple approach

More complex file systems build from these basic data structures

Summary/Future

We've described a very simple FS.

- basic on-disk structures
- the basic ops

Future questions:

- how to handle **crashes**?

Crash Consistency

Questions answered:

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 <u>redundancy</u> between *A* and *B*

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

Pros and CONs of Redundancy

Redundancy may improve:

- reliability
  - Superblocks in FFS
- performance
  - bitmaps

But Redundancy could hurt!

- capacity
- consistency
  - Redundancy implies certain combinations of values are (possibly) 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

Scenario 2: Consistent or not?

Superblock: field contains total blocks in FS.
DATA = 1024
node: field contains pointer to data block.
DATA = 2345

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

Bad things that can happen: inconsistency, garbage data, data loss,

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 & data

b) **data**: Data loss, but otherwise OK

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

d) **bitmap** and **data**: lost block & data (nothing can reach it)

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

f) **data** and **inode**: **another file may use (from bitmap)**

How can file system fix Inconsistencies?

Solution #1:

- FSCK = file system checker

Strategy:

- After crash, scan whole disk for contradictions and “fix” if needed
- Keep file system off-line until FSCK completes

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

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

Fsck Checks

Hundreds of types of checks over different fields…

Do superblocks match?

Do directories contain “.” and “..”?

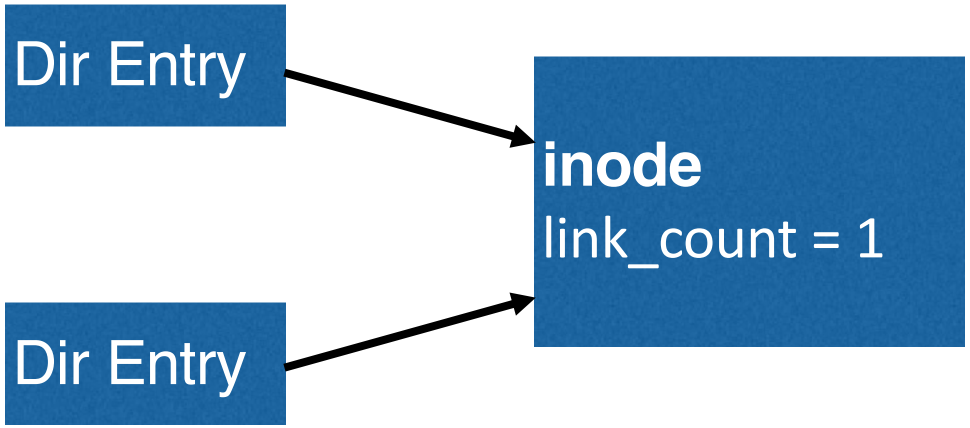
Do number of dir entries equal **inode link counts**?

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

…

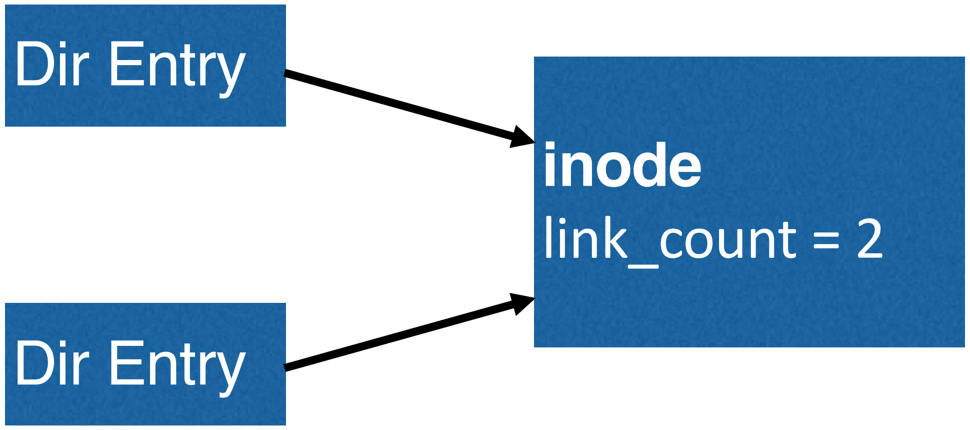
How to solve problems?

Link Count (example 1)



How to fix to have consistent file system?

Link Count (example 1)



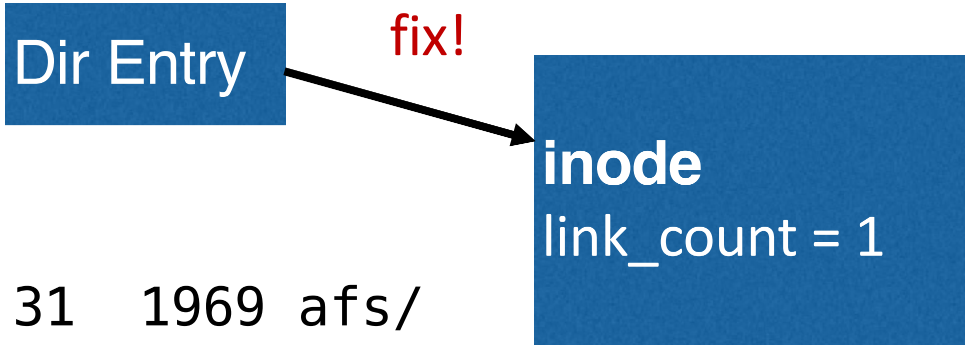
Simple fix!

Link Count (example 2)

inode
link_count = 1

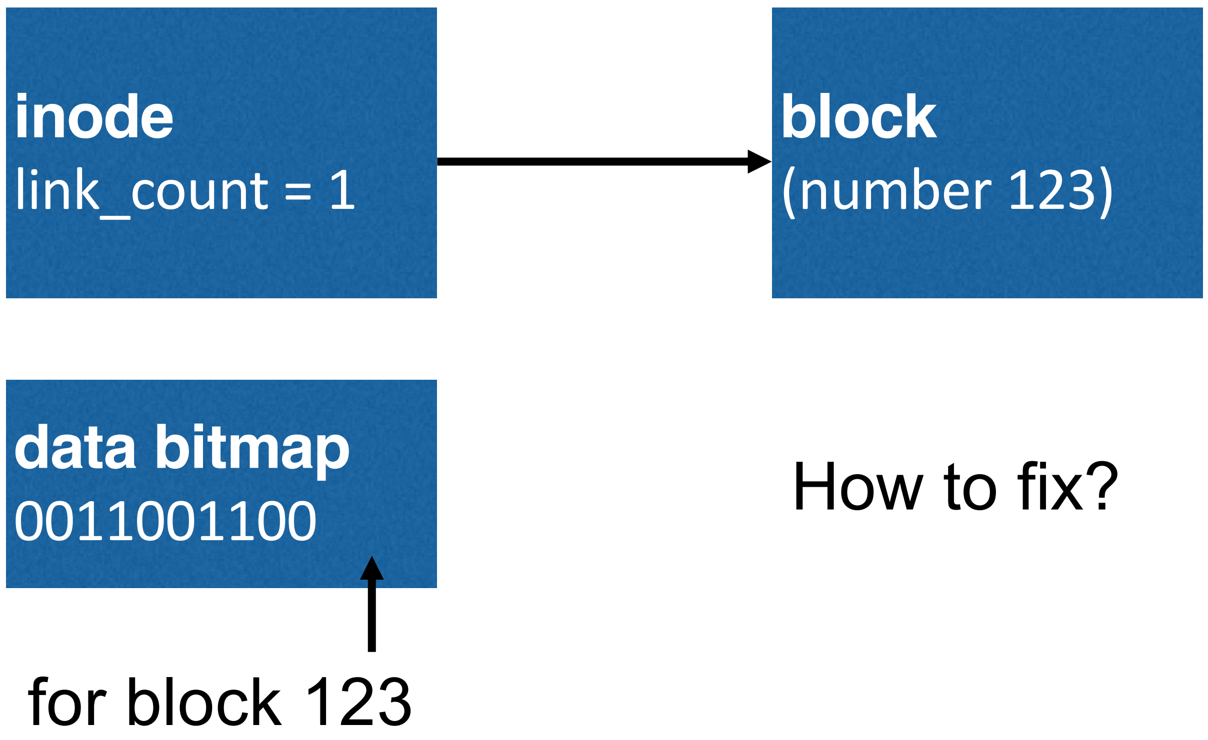
How to fix???

Link Count (example 2)



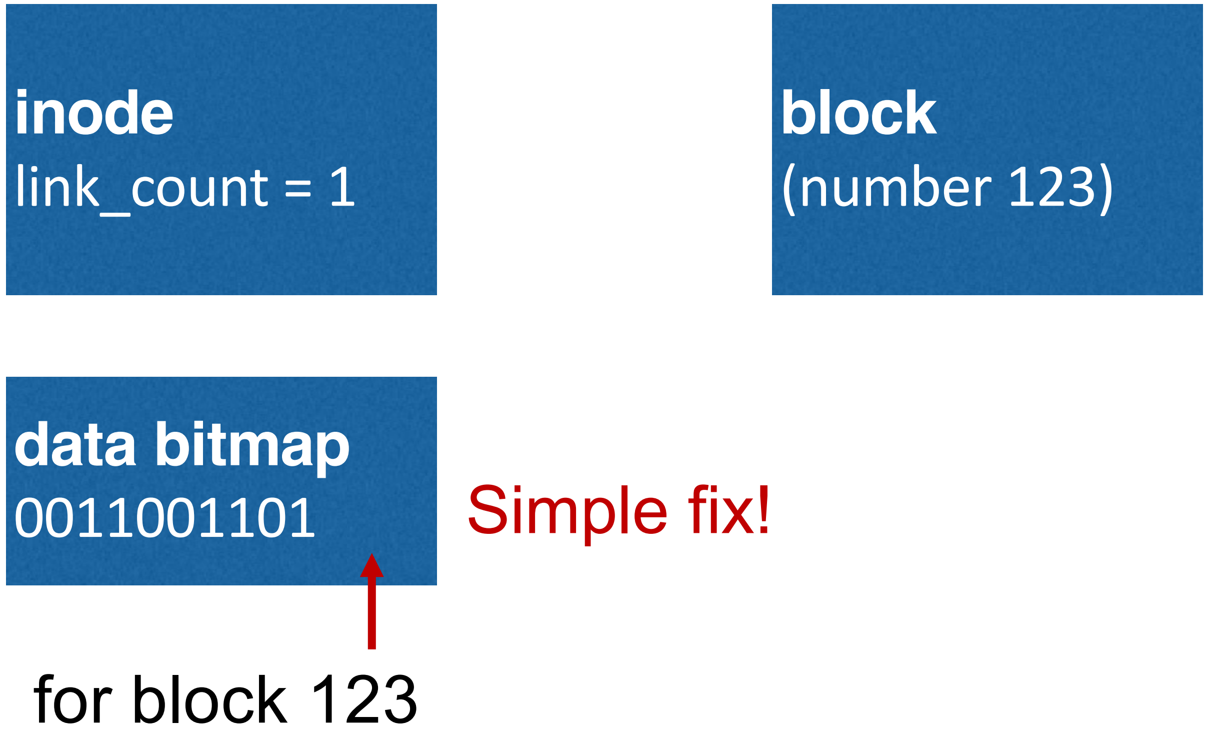

```
ls -l /
total 150
drwxr-xr-x  401 18432 Dec 31  1969 afs/
drwxr-xr-x.   2 4096  Nov  3 09:42 bin/
drwxr-xr-x.   5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.  13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.  10 12288 Nov  3 09:41 lib64/
drwx------.   2 16384 Aug  1 10:57 lost+found/
...
```

Data Bitmap

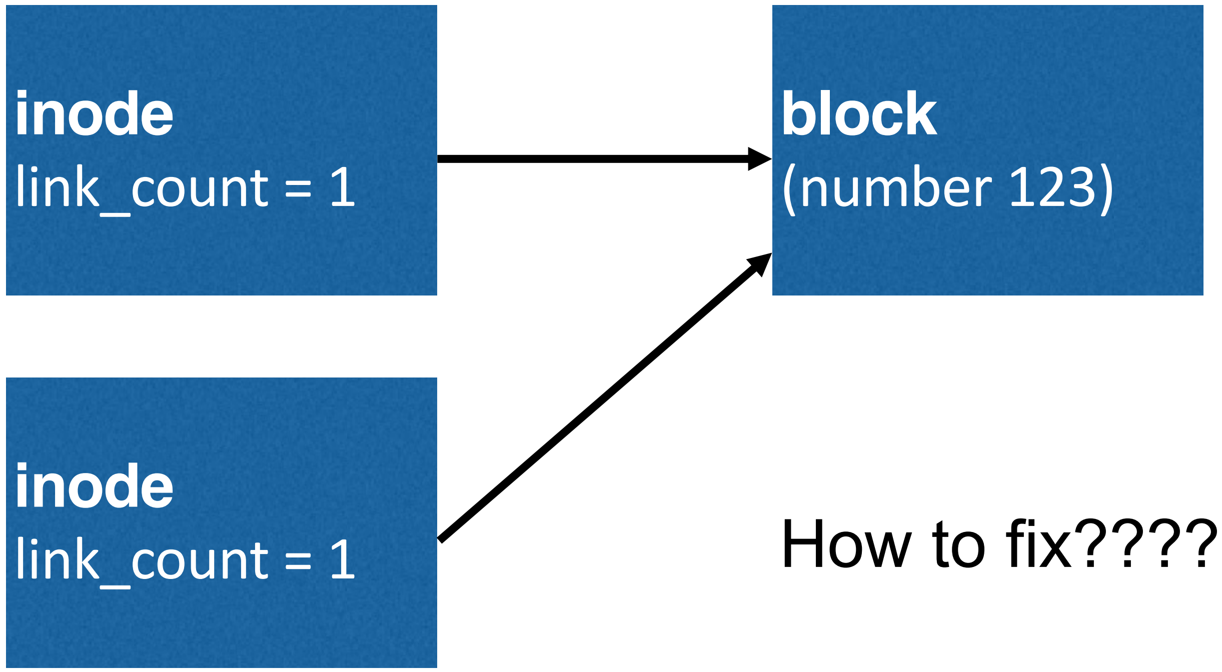


How to fix?

Data Bitmap

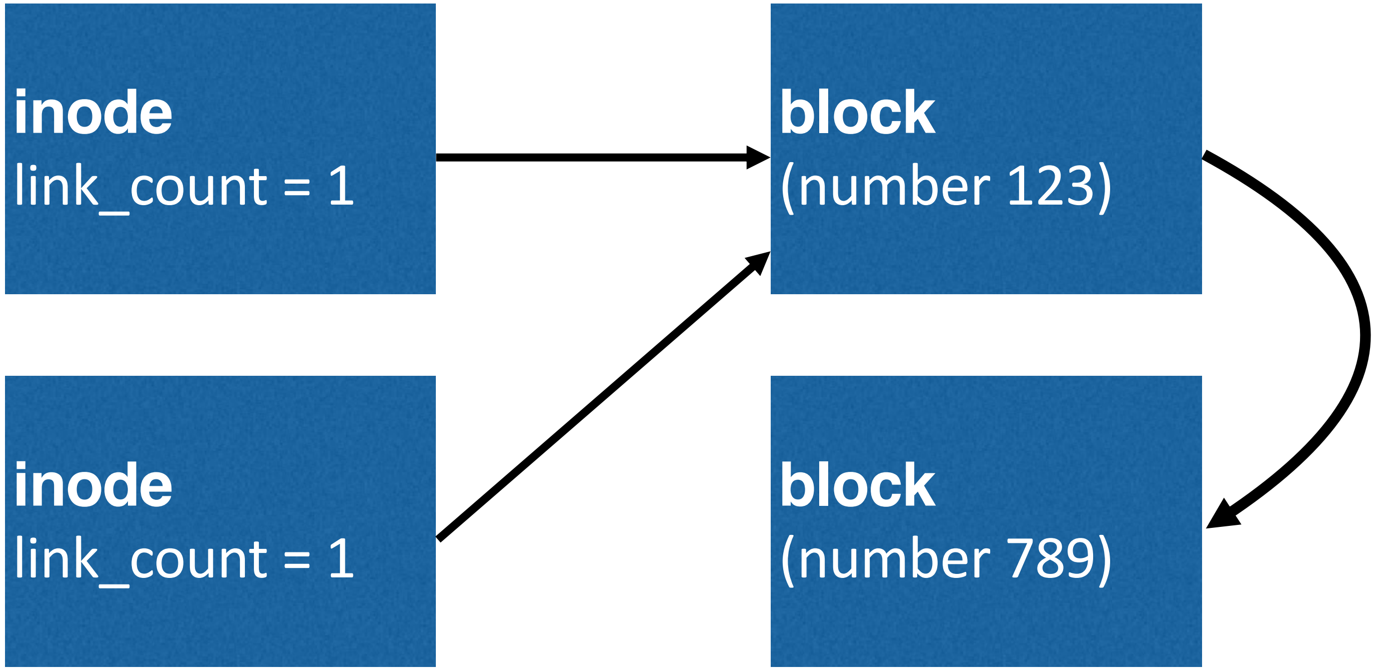

inode
link_count = 1
block
(number 123)
data bitmap
0011001101
Simple fix!
for block 123

Duplicate Pointers

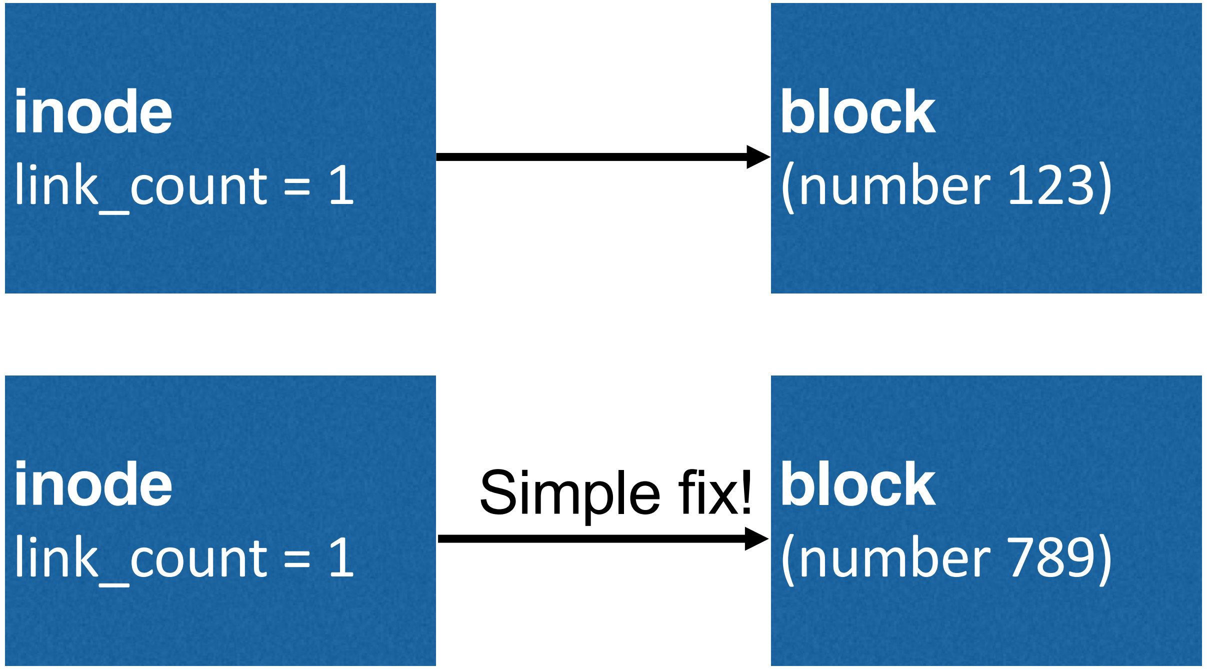


How to fix????

Duplicate Pointers

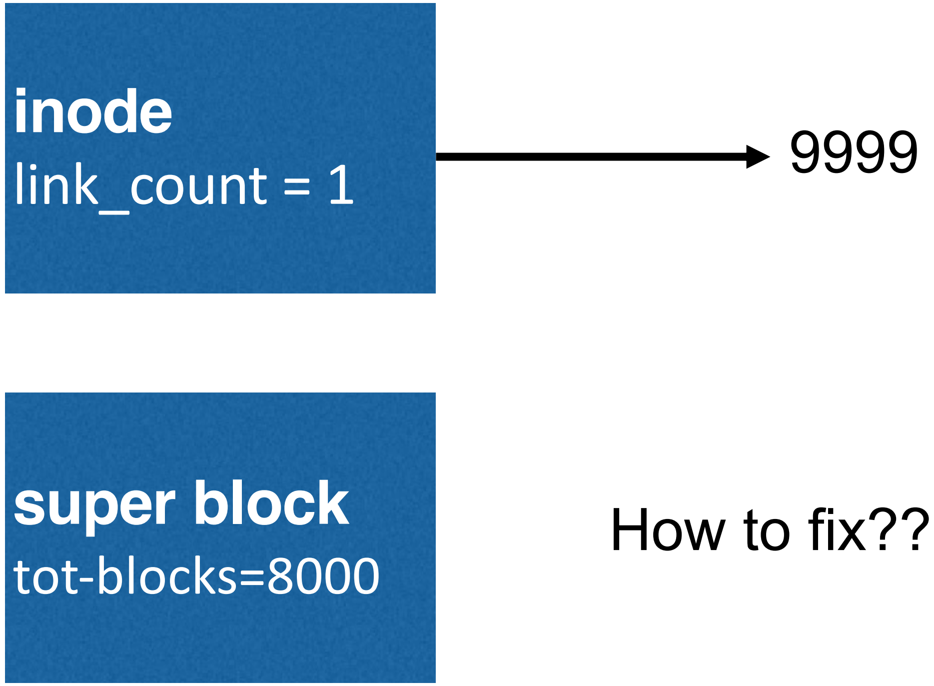


Duplicate Pointers



But is this correct?

Bad Pointer



How to fix???

Bad Pointer

inode
link_count = 1

Simple fix! (But is this correct?)

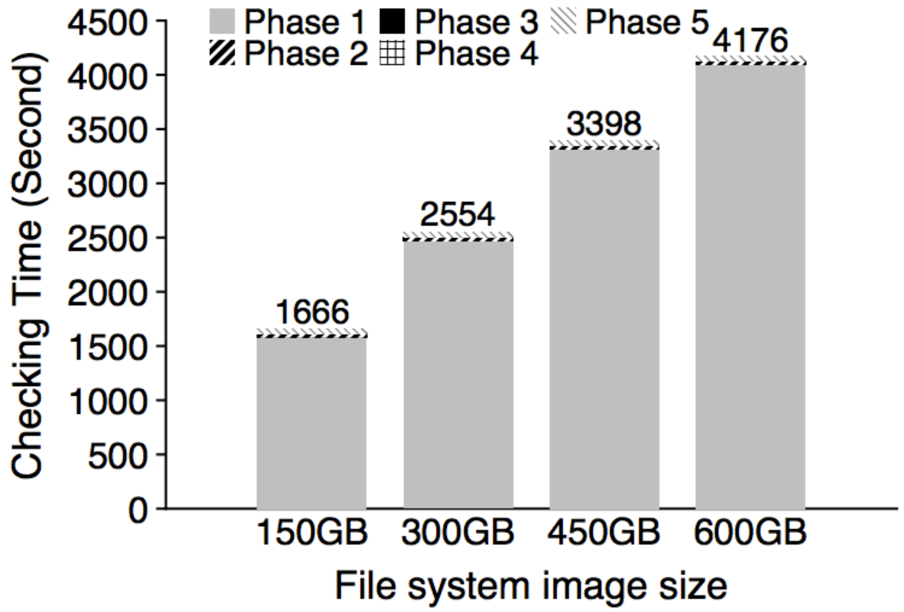
super block
tot-blocks=8000

Problems with fsck

Problem 1:

- Not always obvious how to fix file system image
- Don’t know “correct” state, just a 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

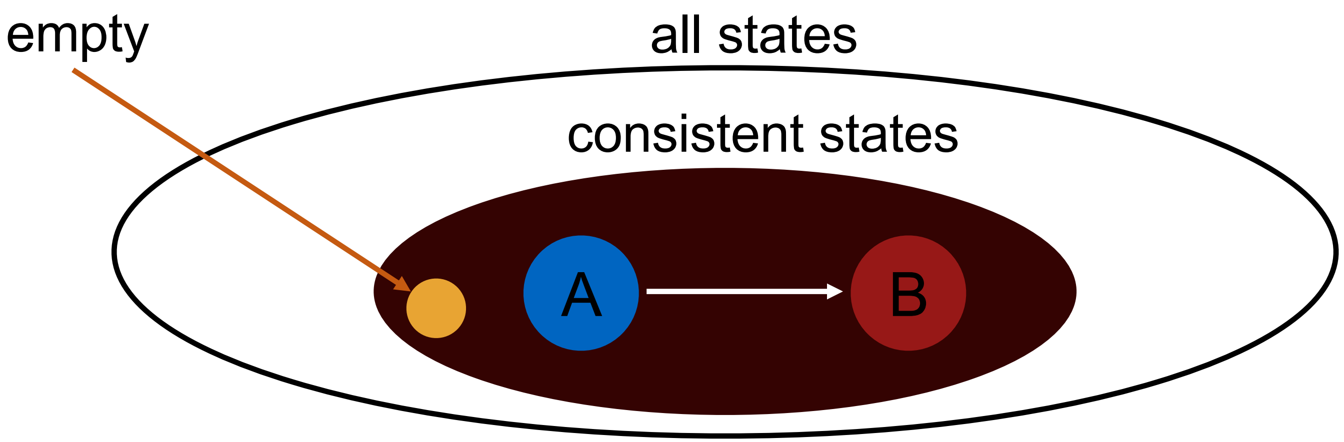
- Ok to do some **recovery work** after crash, but not to read entire disk
- Don't move file system to just any consistent state, get **correct** state (in most cases)

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.

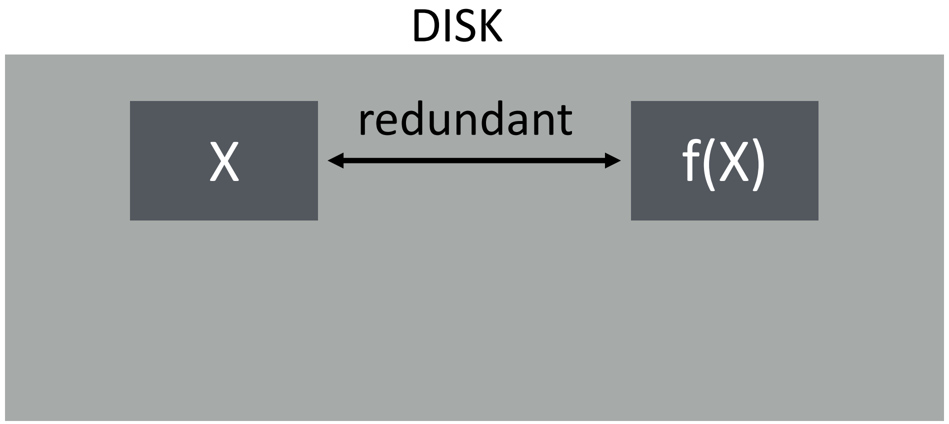
Journaling: General Strategy

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

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

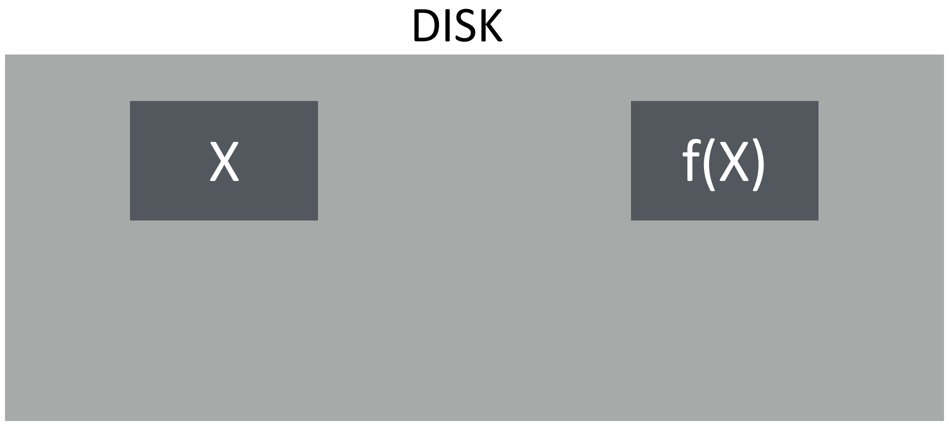
Fight Redundancy with Redundancy

Want to replace X with Y. Original:



Fight Redundancy with Redundancy

Want to replace X with Y. Original:

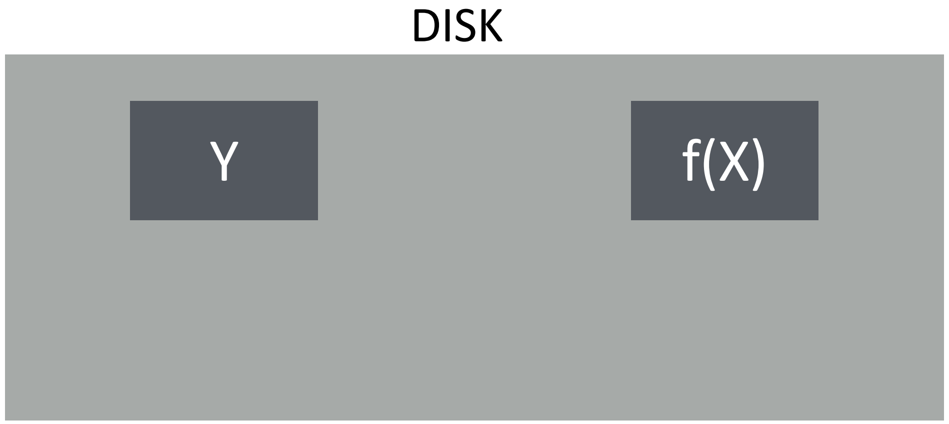


Good time to crash?

Yes, good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

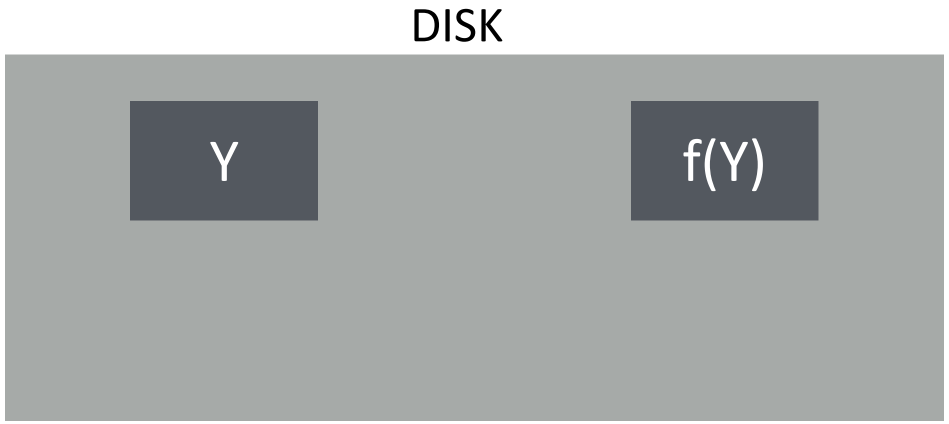


Good time to crash?

bad time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

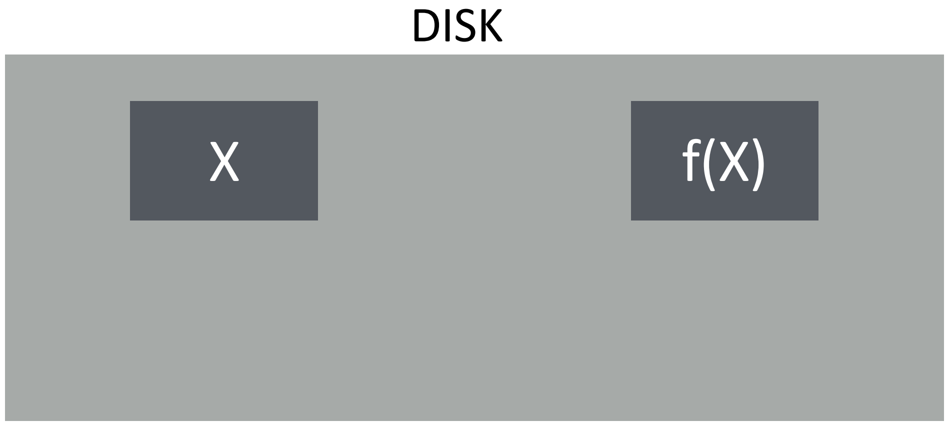


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

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

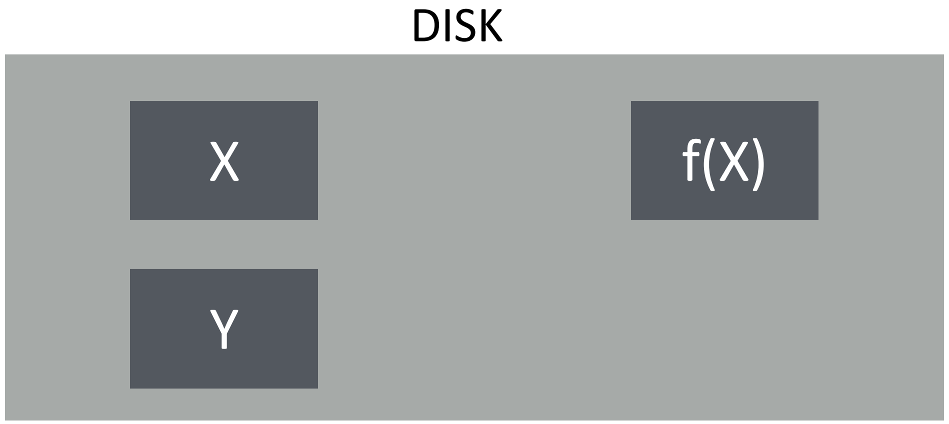


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

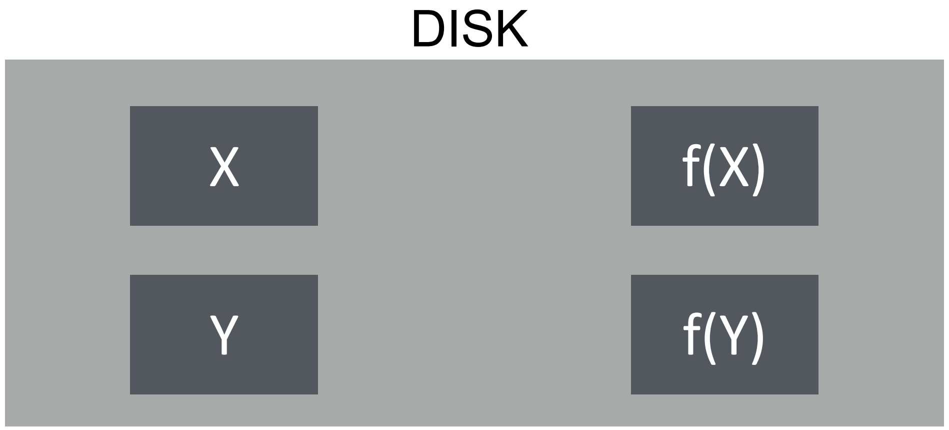
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

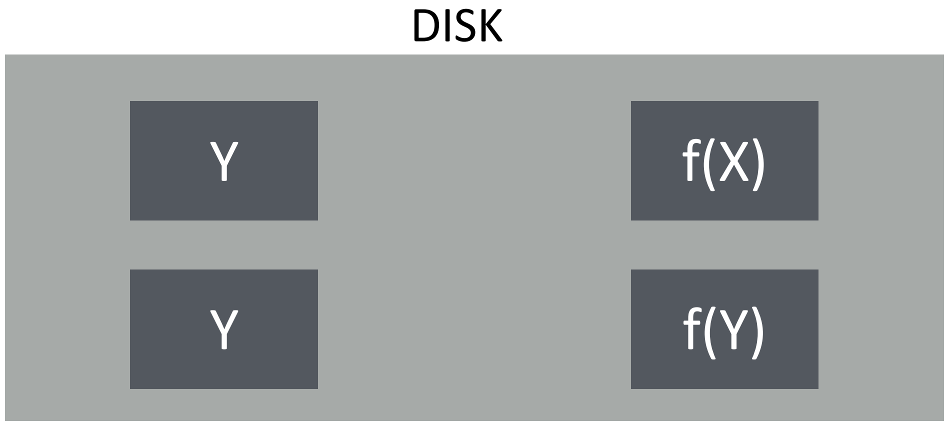
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

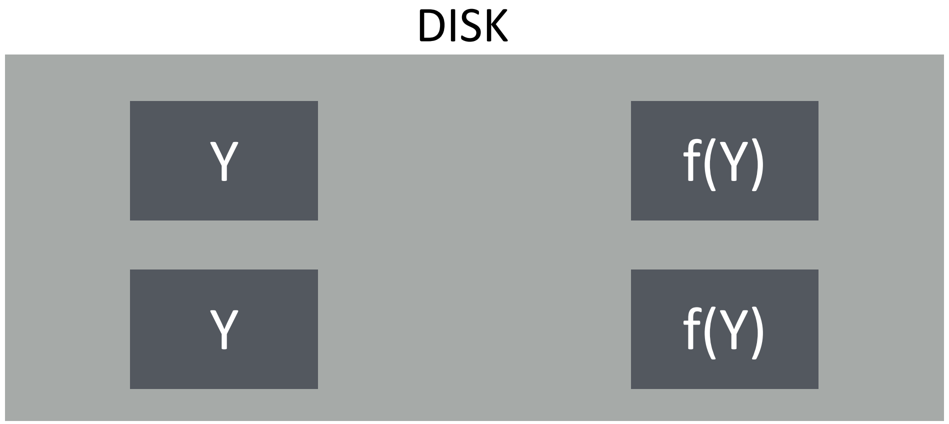
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

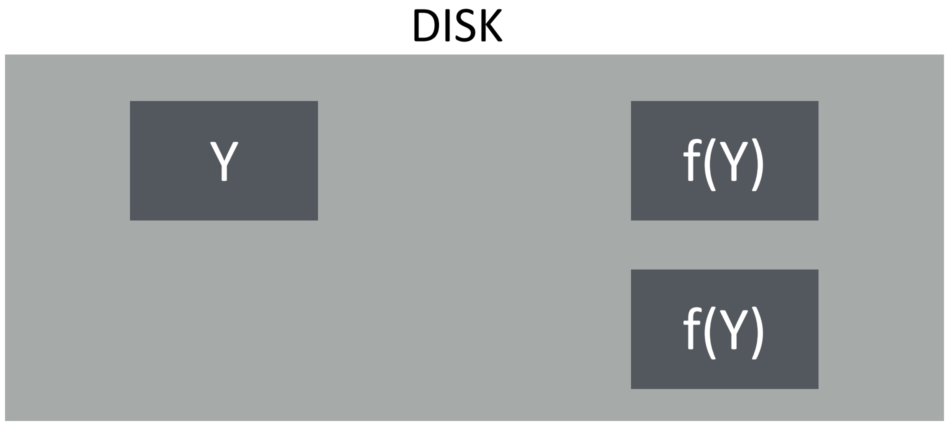
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

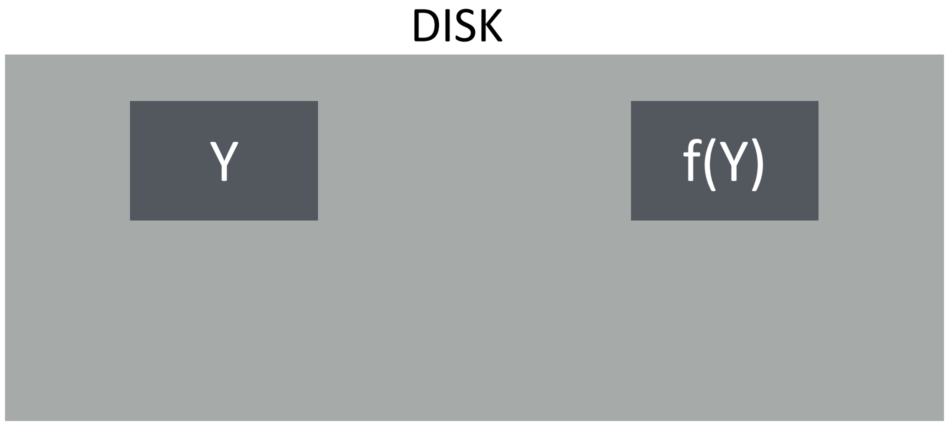
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

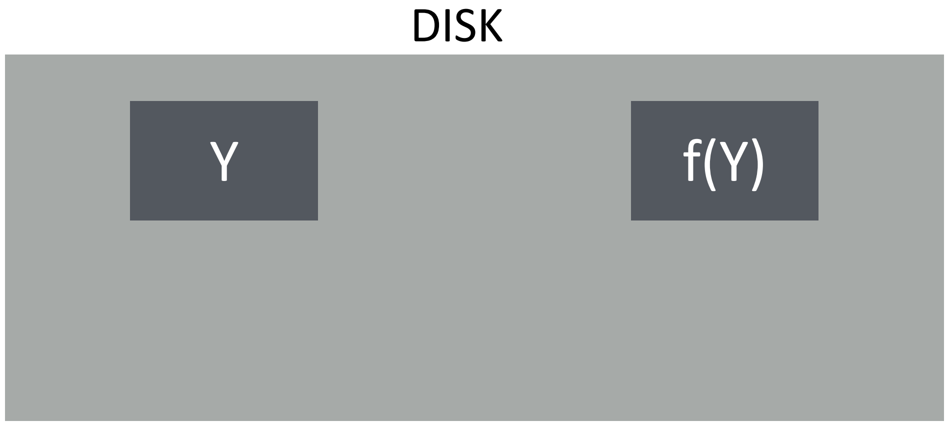
Want to replace X with Y. With journal:



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…

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

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	J:2	J:3	J:valid:4
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 journal are a “journal transaction”

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

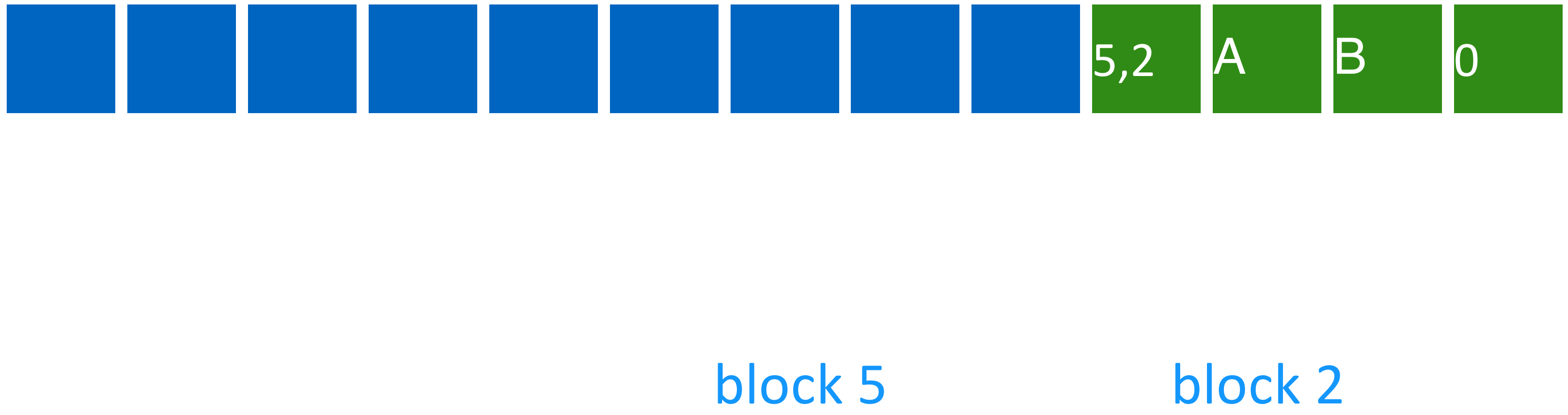
Still need to first write all new data elsewhere before overwriting new data

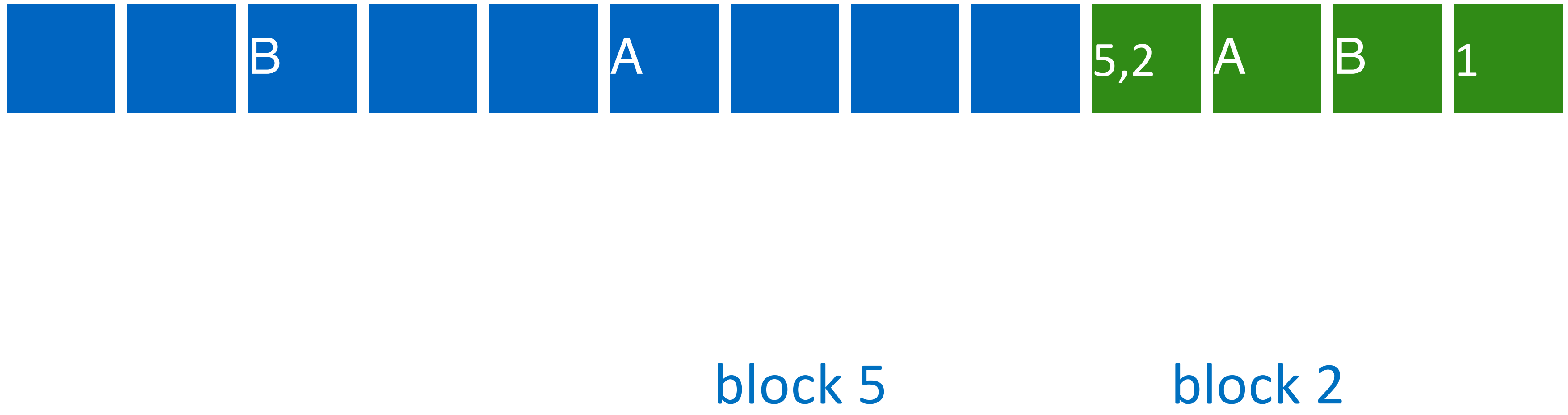
Goal:

- Reuse small area as backup for any block

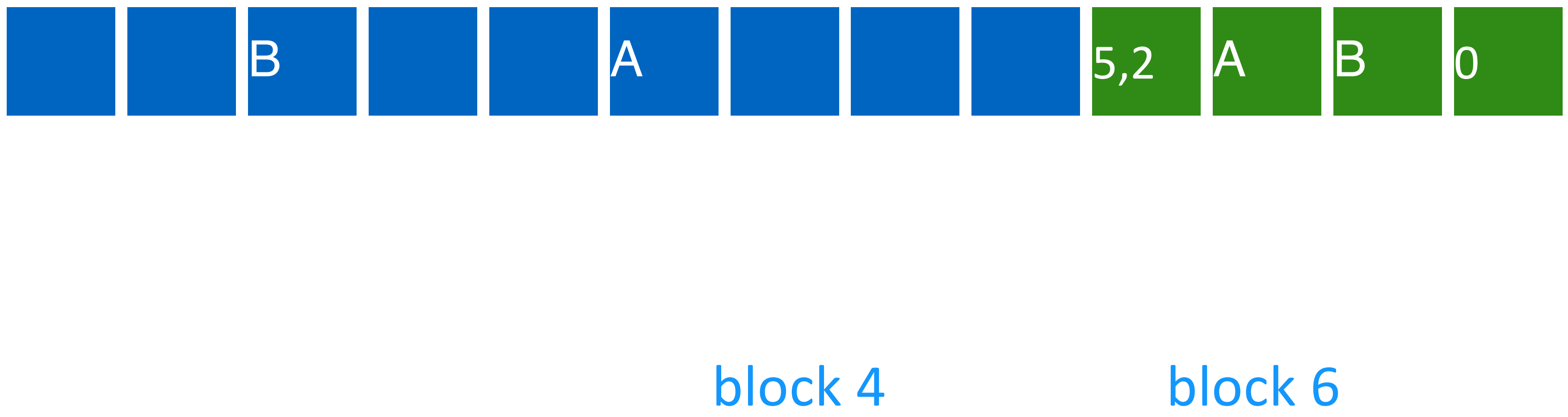
How?

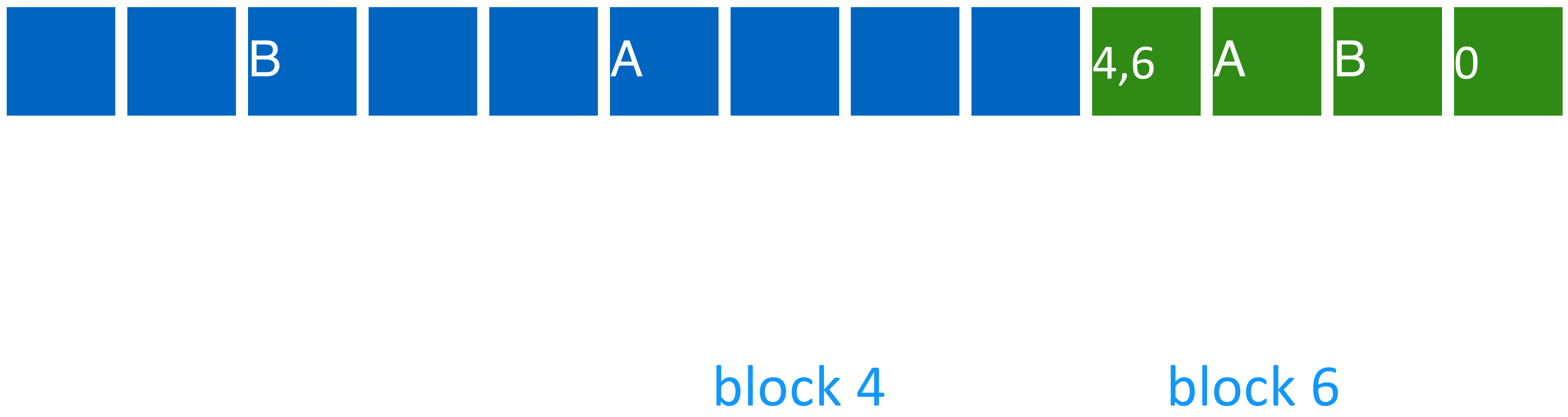
- Store block numbers in a transaction header

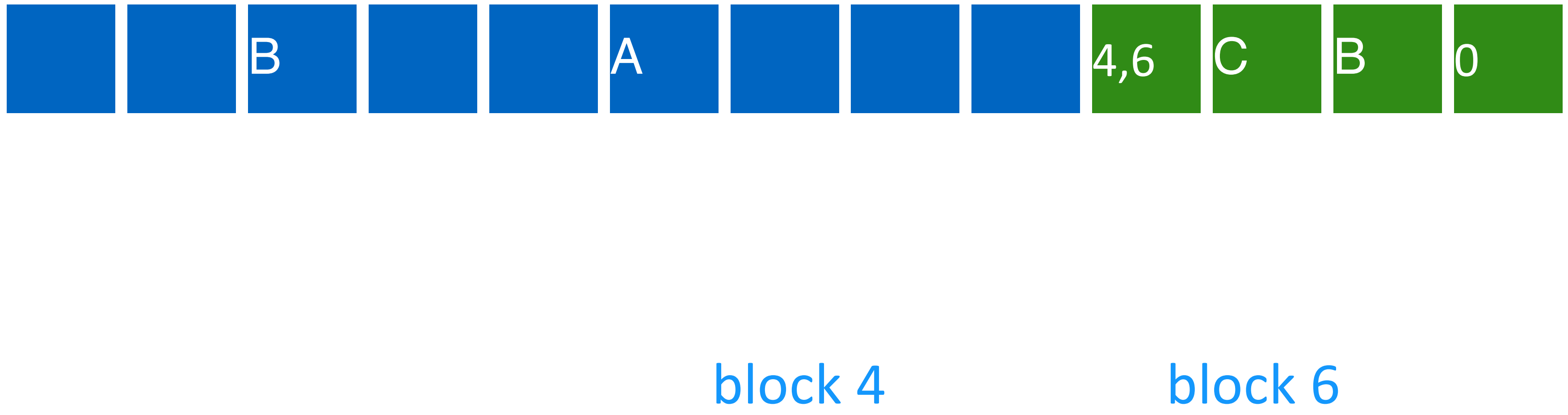

5,2
A
B
0
block 5
block 2


B
A
5,2
A
B
1
block 5
block 2

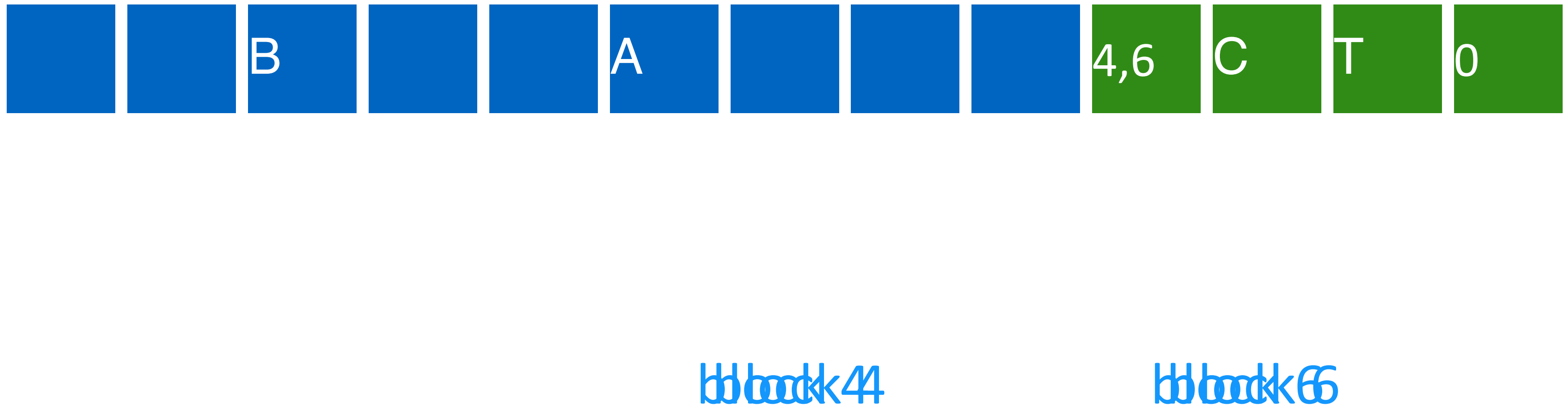

B
A
5,2
A
B
0


B
A
5,2
A
B
0
block 4
block 6


B
A
4,6
A
B
0
block 4
block 6



B
A
4,6
C
B
0
block 4
block 6


B
A
4,6
C
T
0
block 4
block 6


B
C
A
T
4,6
C
T
1
block 4
block 6


B
C
A
T
4,6
C
T
0
block 4
block 6

1. Reuse small area for journal
2. Barriers - (fsync)
3. Checksums
4. Circular journal
5. Logical journal

block 4 block 6

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

Enforcing total ordering is inefficient. Why?

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

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

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

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

12 = Cksum(9, 10, 11)

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

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


T1
T2
T3
T4

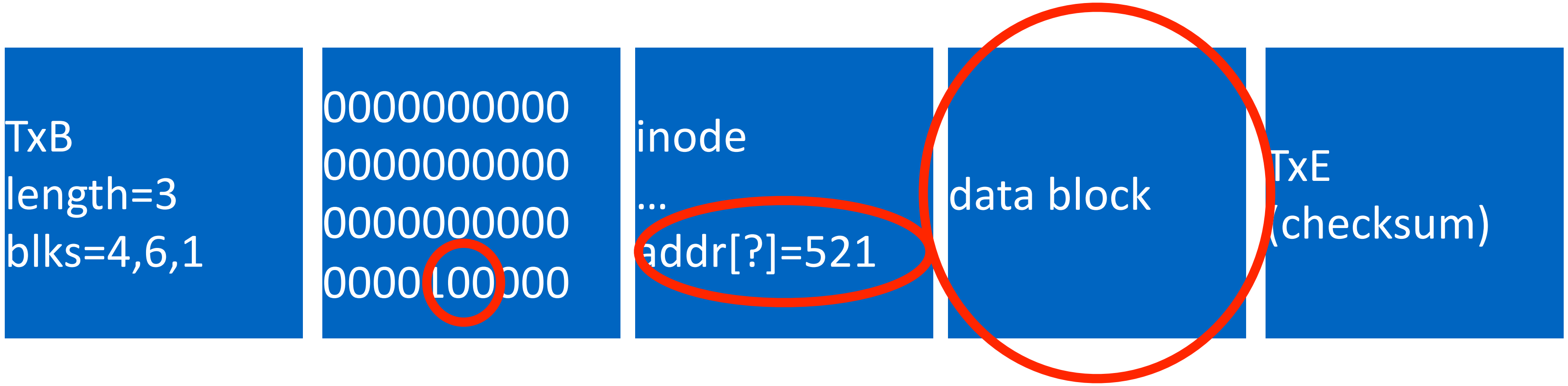

T2
T3
T4


T5
T2
T3
T4


T5
T3
T4

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

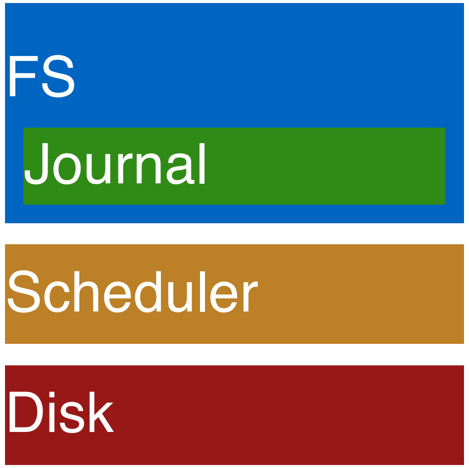
TxB length=3 blks=4,6,1	0000000000 0000000000 0000000000 0000100000	inode … addr[?]=521	data block	TxE (checksum)


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

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


FS
Journal
Scheduler
Disk

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.


B
I
?
0


B
I
?
TxB
B’
I’
0


B
I
?
TxB
B’
I’
TxE


B
I’
?
TxB
B’
I’
TxE


B’
I’
?
TxB
B’
I’
TxE

Still only journal metadata

But write data **before** the transaction

No leaks of sensitive data!

	B		I			?						0


B
I
D
0


B
I
D
TxB
I’
B’
0


B
I
D
TxB
I’
B’
TxE


B
I’
D
TxB
I’
B’
TxE


B’
I’
D
TxB
I’
B’
TxE

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)