We acknowledge and pay our respects to the Kaurna people, the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

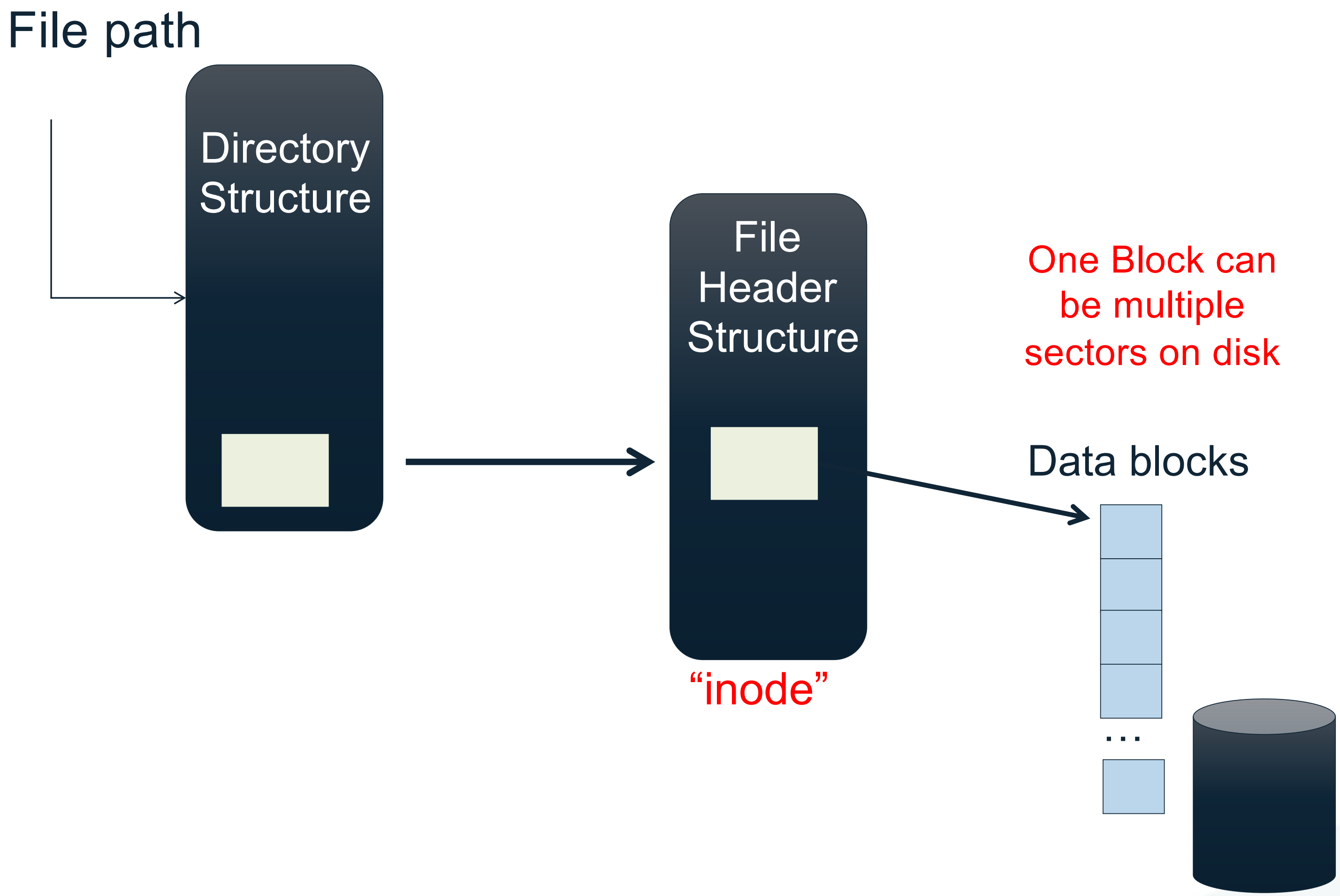
COMP SCI 3004 Operating Systems

Week 11 – Fast File System & Log-Structured File System

make history.

THE UNIVERSITY of ADELAIDE

Recall: Components of a File System

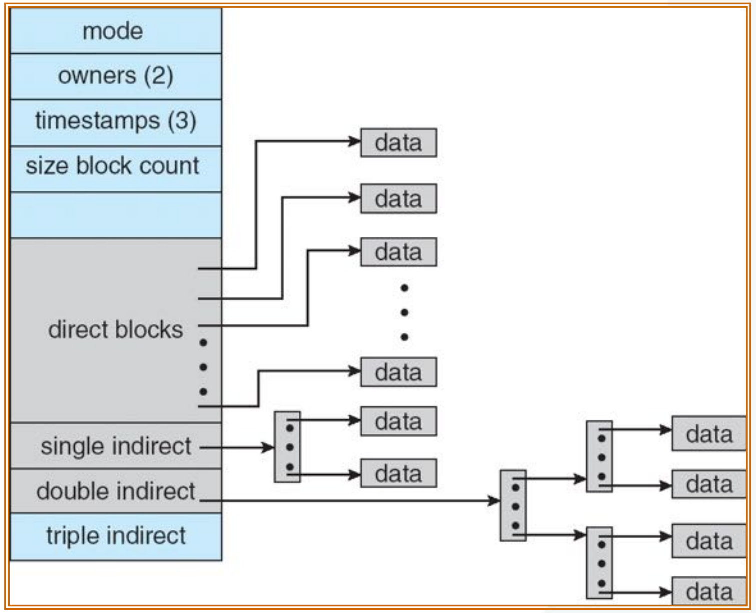


Recall: Multilevel Indexed Files

Sample file in multilevel indexed format:
12 direct ptrs, different block sizes, e.g., 1 KB or 4KB

Pros: Simple (more or less)
Files can easily expand (up to a point)
Small files particularly cheap and easy

Cons: Lots of seeks
Very large files must read many indirect block (four I/Os per block!)



File-System: Case Studies

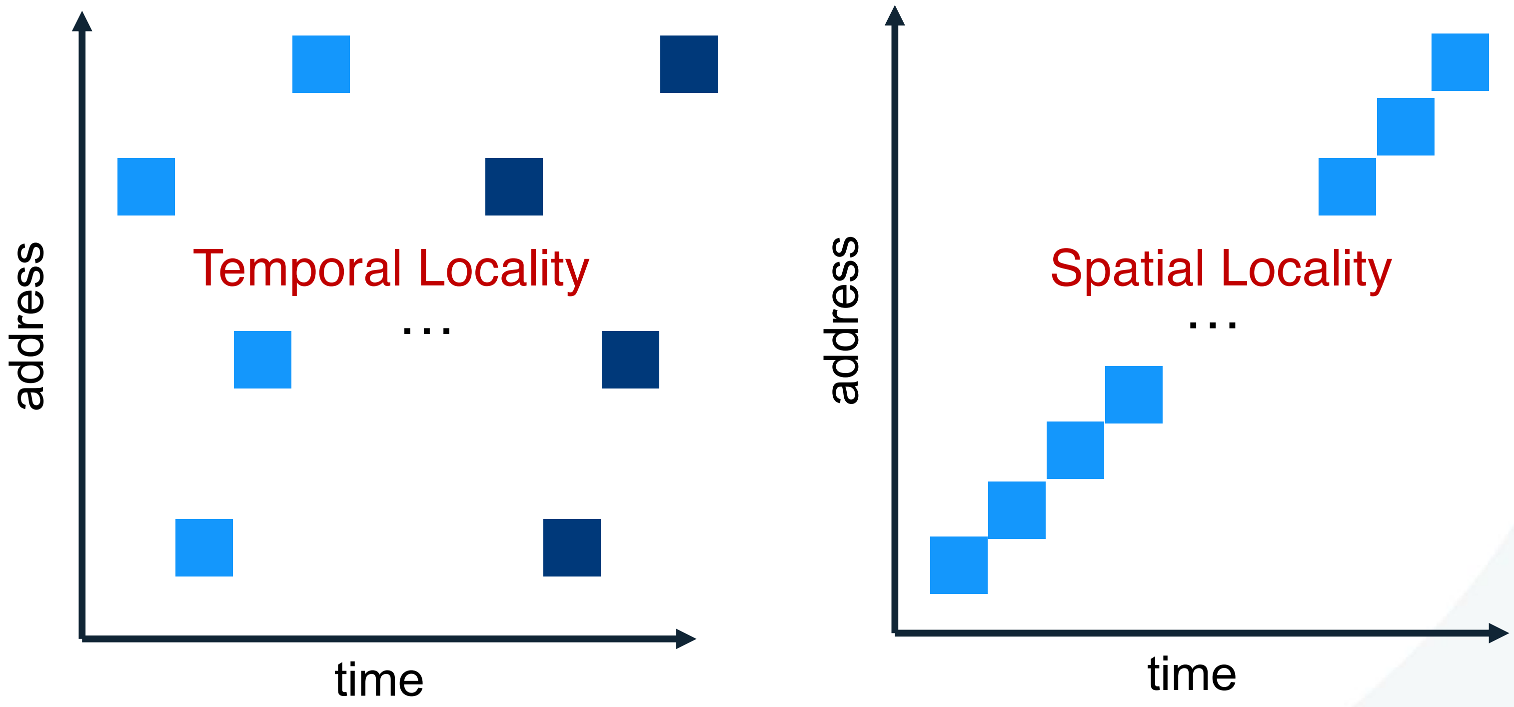
Local (this week)

- FFS: Fast File System
- LFS: Log-Structured File System

Network (next week)

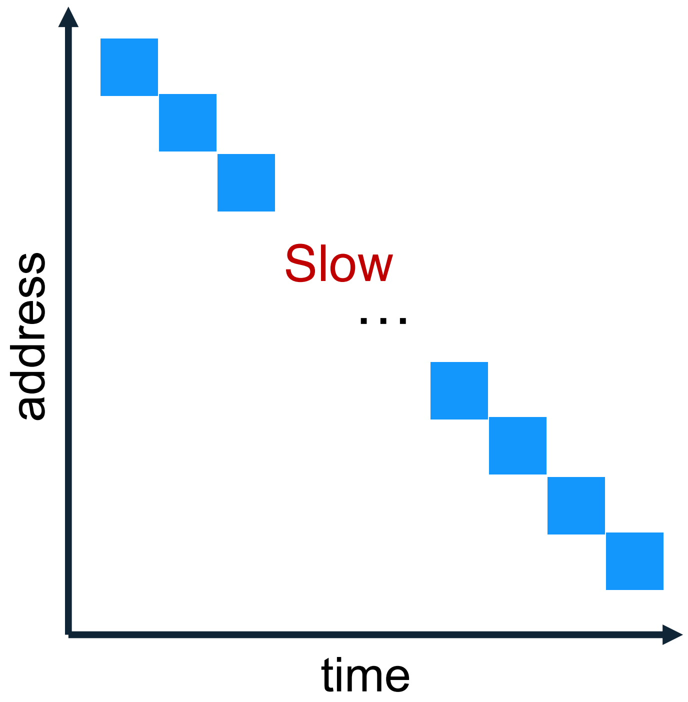
- NFS: Network File System
- AFS: Andrew File System

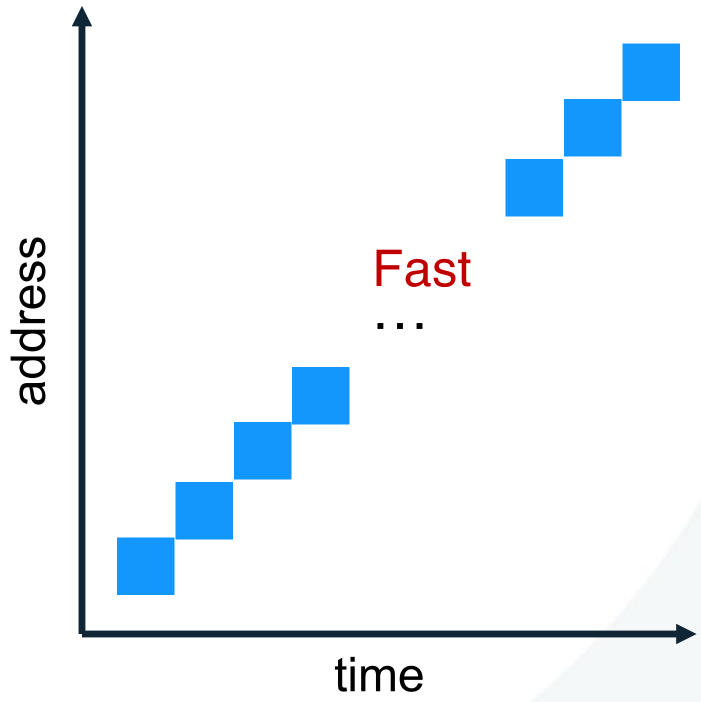
Review: Locality Types



Which type of locality is most interesting with a disk?

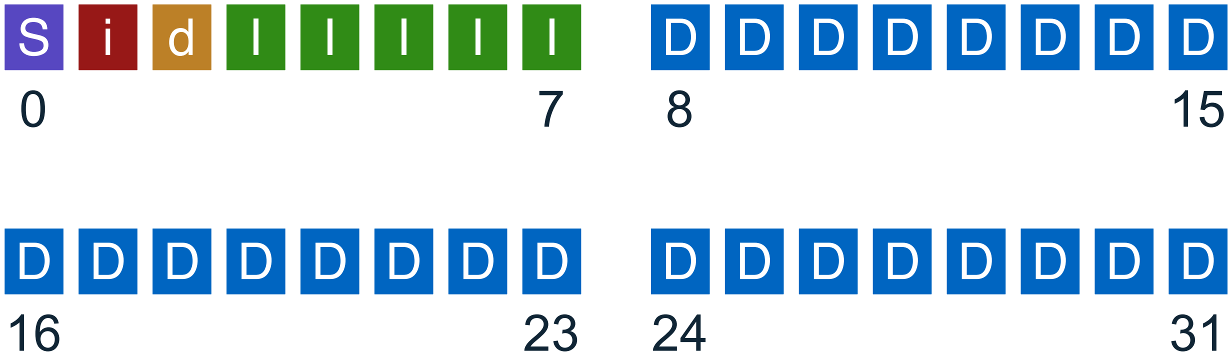
Order Matters





Implication for disk schedulers?

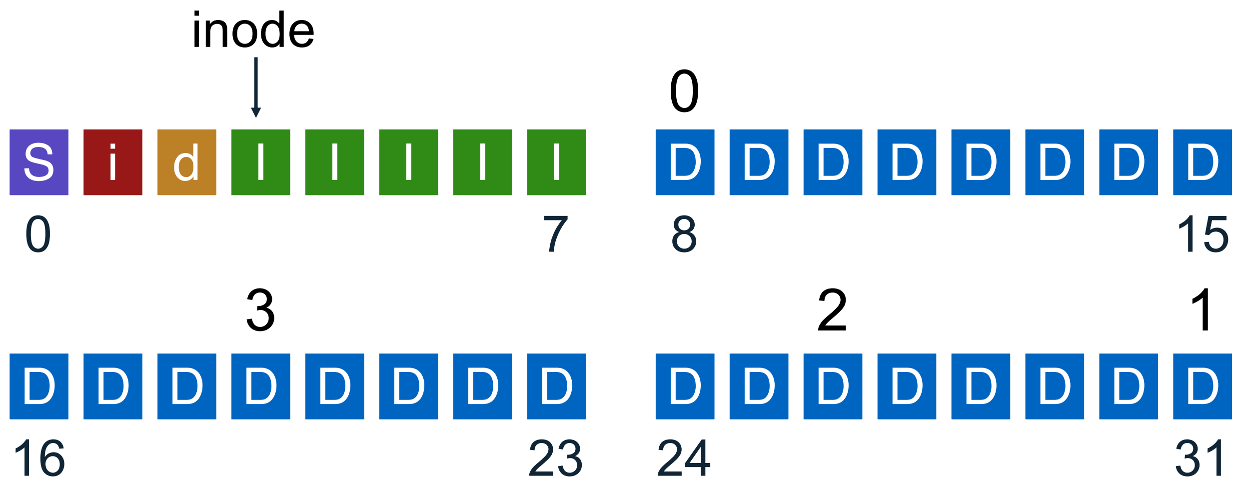
Policy: Choose Inode, Data Blocks



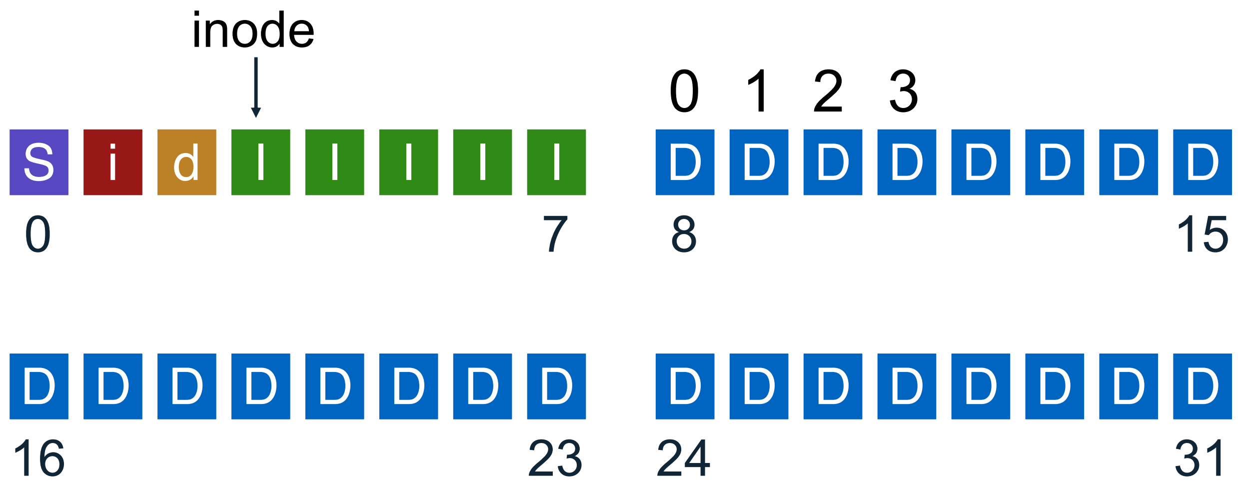
Assuming all free, which should be chosen?

THE UNIVERSITY of ADELAIDE

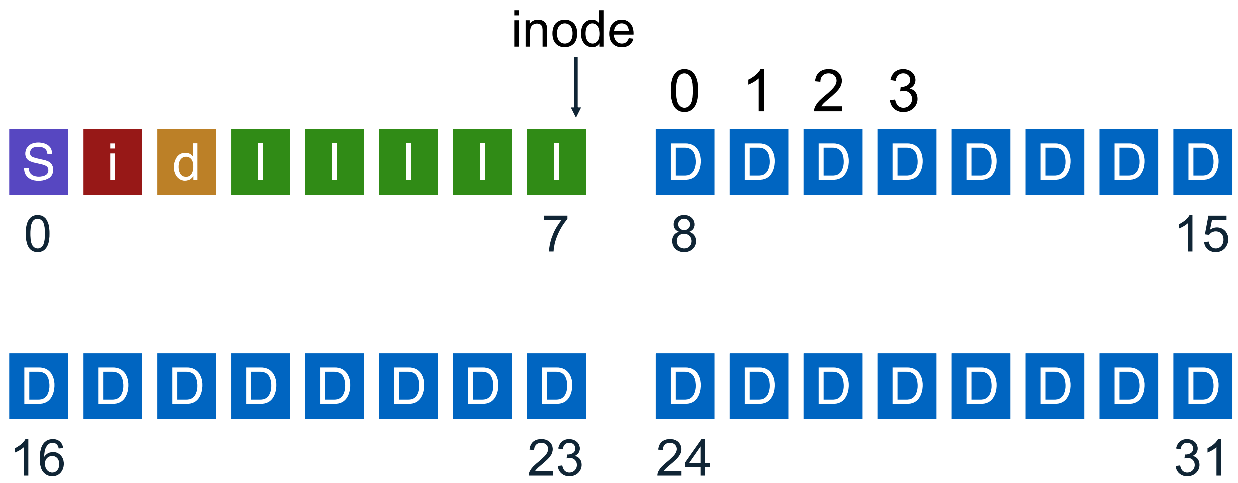
Bad File Layout



Better File Layout



Best File Layout



Can’t do this for all files ☹

Fast File System: FFS (1980’s)

System Building

Beginner's approach

1. get idea
2. build it!

"Pro approach" **measure then build**

1. identify existing state of the art
2. measure it, identify and understand problems
3. get idea (solutions often flow from deeply understanding problem)
4. build it!

THE UNIVERSITY of ADELAIDE

Measure Old FS

State of the art: original UNIX file system

Free lists are embedded in inodes, data blocks
Data blocks was 512 bytes

Measure throughput for whole sequential file reads/writes

Compare to the theoretical max, which is… disk bandwidth

Measurement 1: Aging?

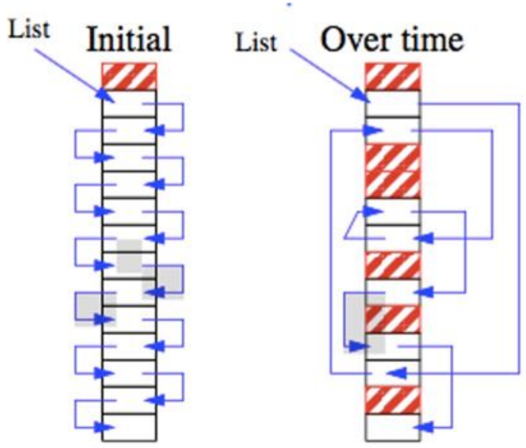
What is the performance before/after aging of OLD Unix FS?

Newly created FS: **17.5%** of disk bandwidth

A few weeks old: **3%** of disk bandwidth

Problem: FS has become fragmented over time

Free list makes contiguous chunks hard to find



Hacky Solutions:

- Occasional defrag of disk
- Keep free list sorted

Measurement 2: Block SIZE?

How does block size affect performance?
Try doubling it!

Result: Performance **more** than doubled

Why double the performance?
- Logically adjacent blocks not physically adjacent
- Only half as many seeks+rotations now required

Why **more** than double the performance?
- Smaller blocks require more indirect blocks

THE UNIVERSITY of ADELAIDE

Old FS Summary

Free list becomes scrambled → random allocations

Small blocks (512 bytes)

Blocks laid out poorly

long distance between inodes/data

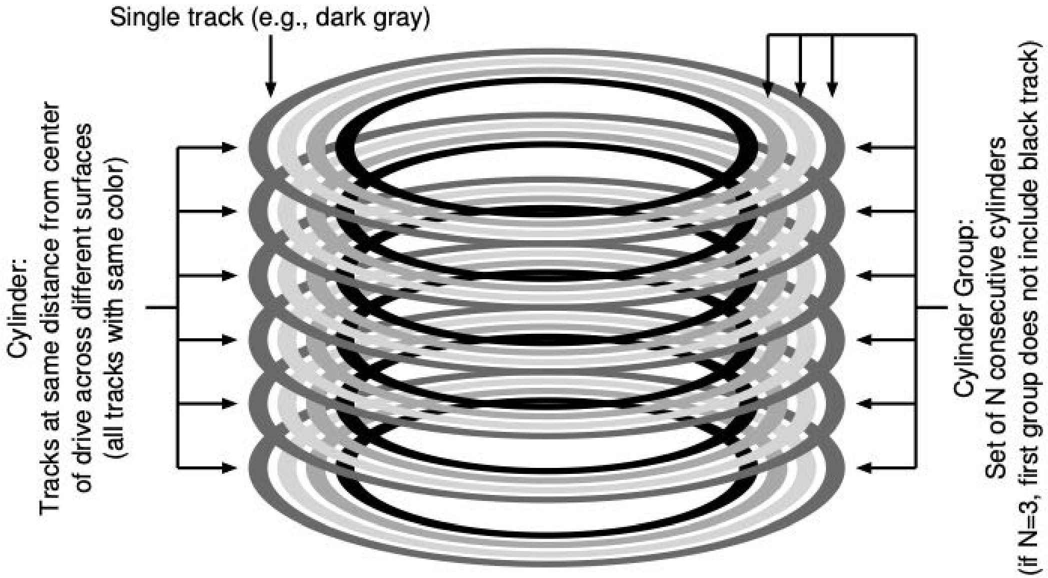
related inodes not close to one another

- Which inodes related?

Result: 2% of potential performance! (and worse over time)

Problem: old FS treats disk like RAM!

Solution: a disk-aware



Primary File System Design Questions:

Where to place meta-data and data on disk?

How to use big blocks without wasting space?

Placement Technique 1: Bitmaps

Use bitmaps instead of free list
Provides better speed, with more global view

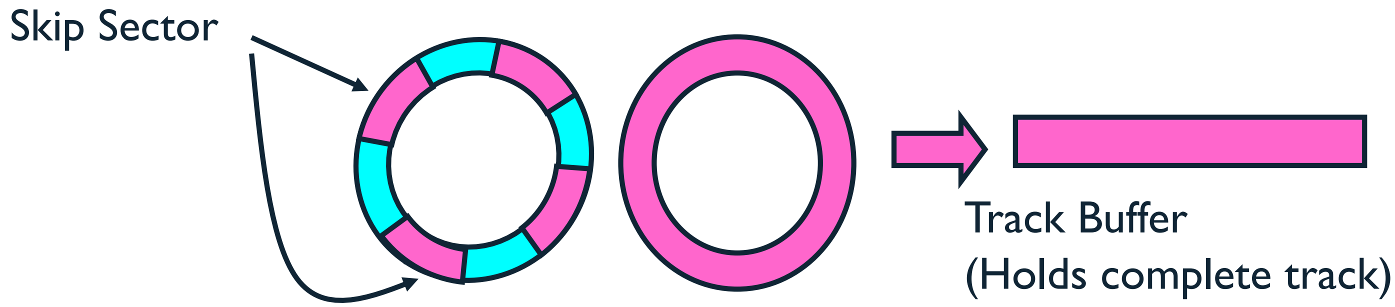
Faster to find contiguous free blocks

THE UNIVERSITY of ADELAIDE

Attack of the Rotational Delay

Problem: Missing blocks due to rotational delay

Issue: Read one block, do processing, and read next block. In the meantime, the disk has continued turning: missed the next block! Need 1 revolution/block!



Solution1: Skip sector positioning (“interleaving”)

- Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
- Can be done by OS or in modern drives by the disk controller

Solution 2: Read ahead: read next block right after first, even if application hasn’t asked for it yet

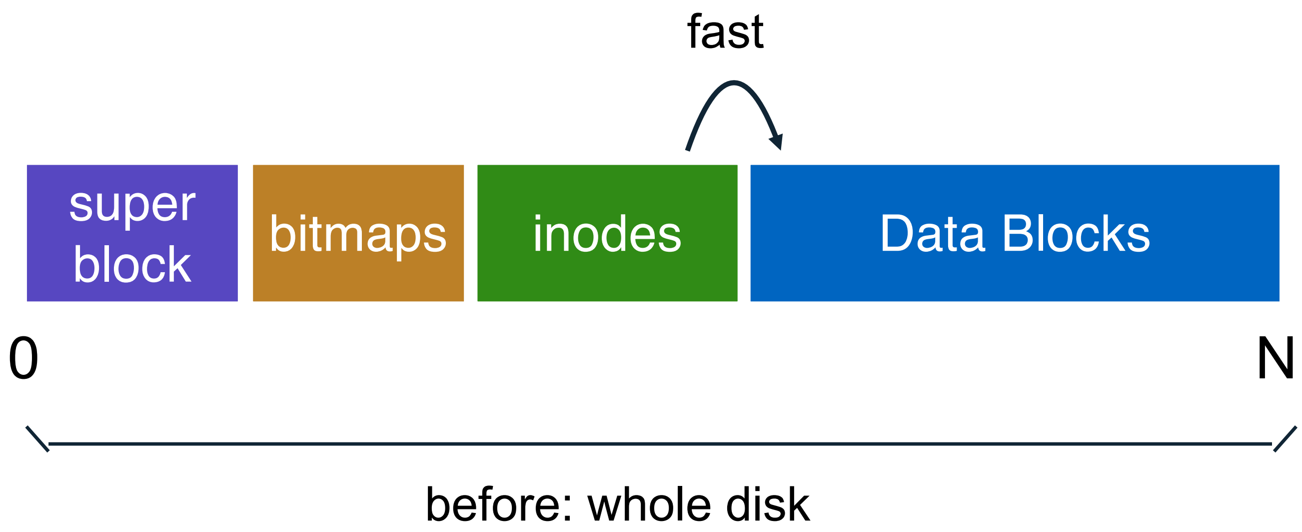
- This can be done either by OS (read ahead)
- By disk itself (track buffers) - many disk controllers have internal RAM that allows them to read a complete track

Modern disks + controllers do many things “under the covers”

Track buffers, elevator algorithms, bad block filtering

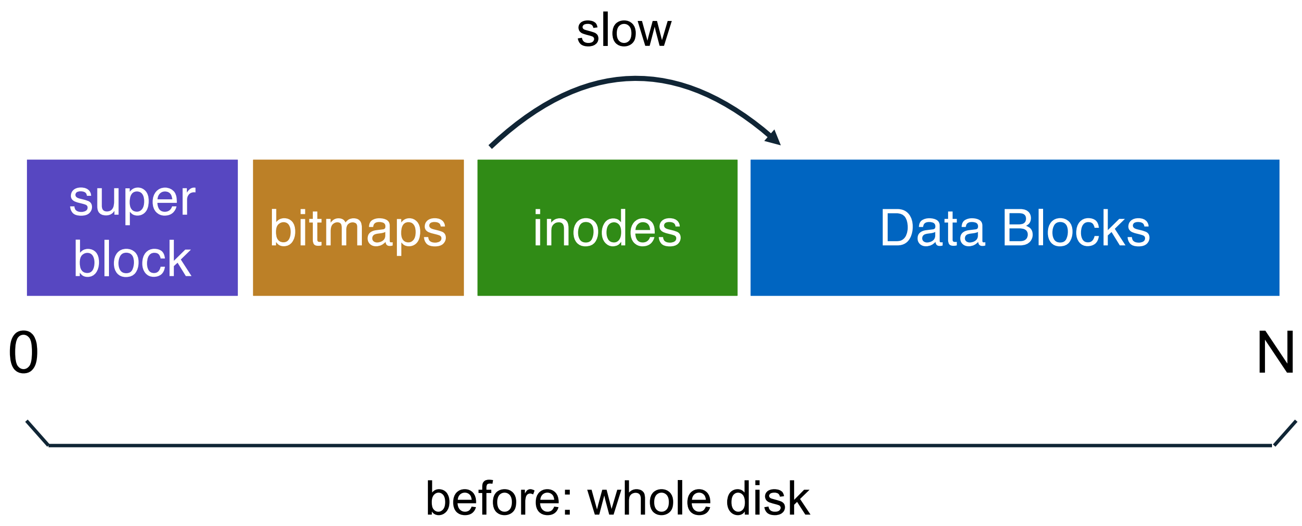
THE UNIVERSITY of ADELAIDE

Placement Technique 2: Groups



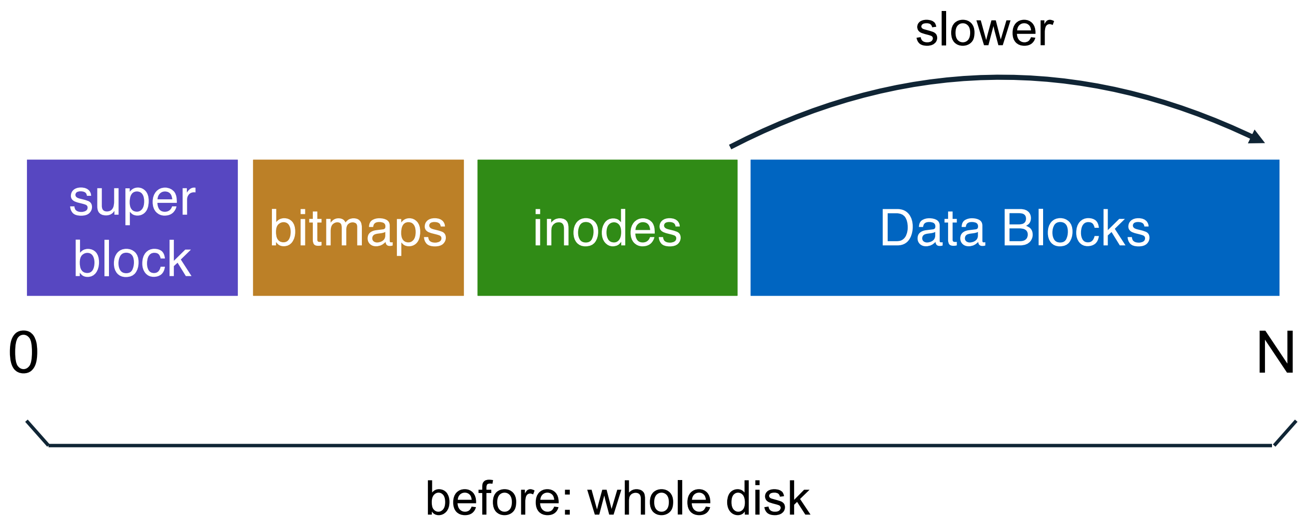
How to keep inode close to data?

Technique 2: Groups



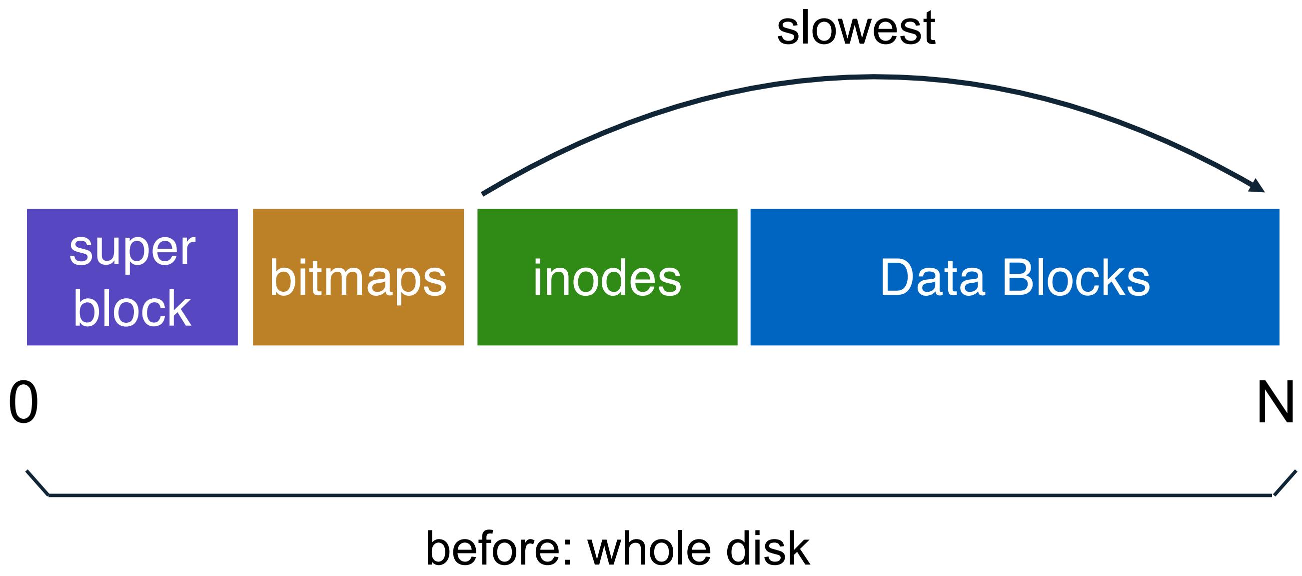
How to keep inode close to data?

Technique 2: Groups



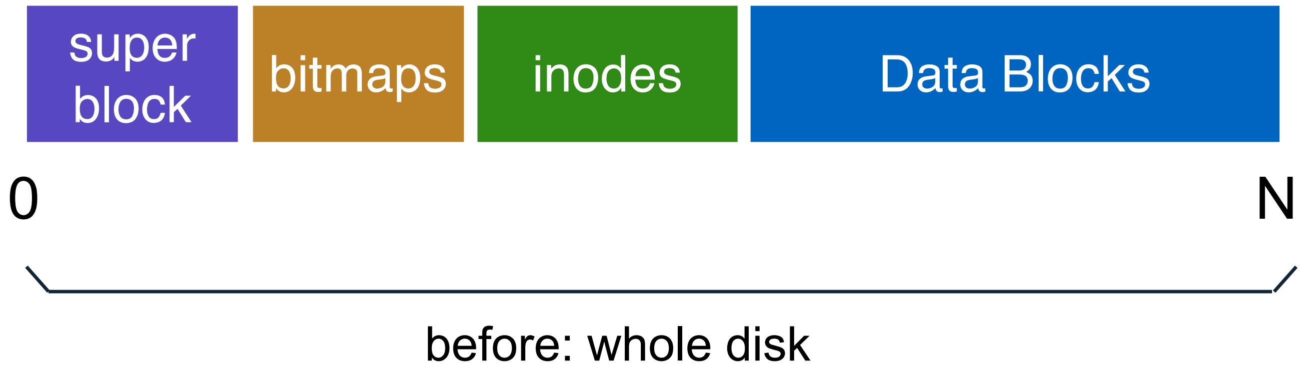
How to keep inode close to data?

Technique 2: Groups



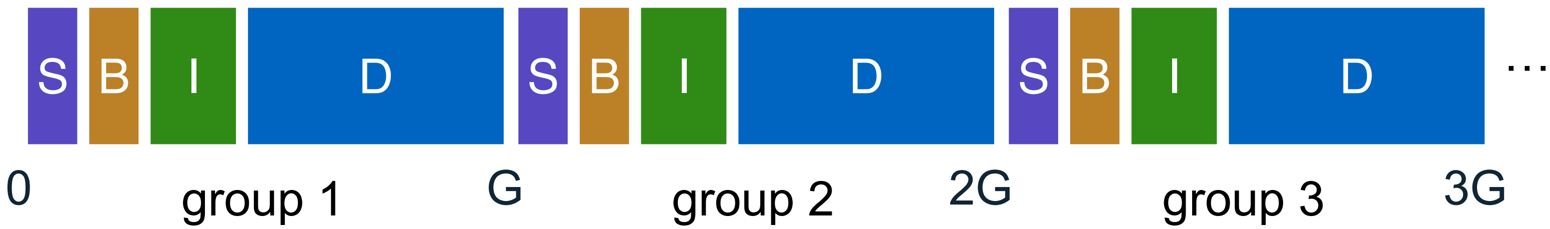
How to keep inode close to data?

Technique 2: Groups



How to keep inode close to data?

Technique 2: Groups

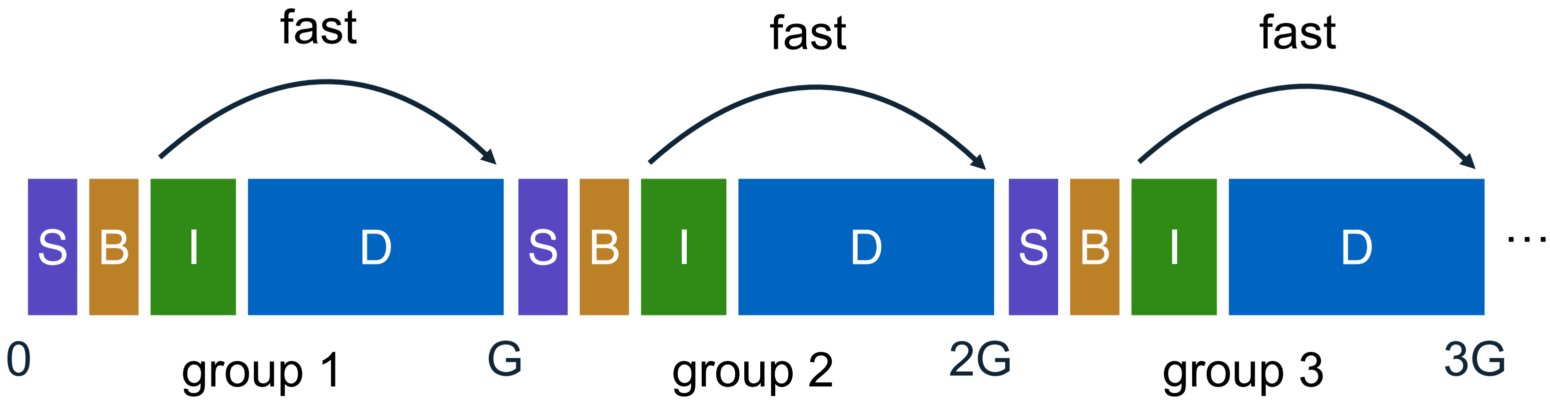


How to keep inode close to data?

Answer: Use groups across disks;
Try to place inode and data in same group

THE UNIVERSITY of ADELAIDE

Technique 2: Groups

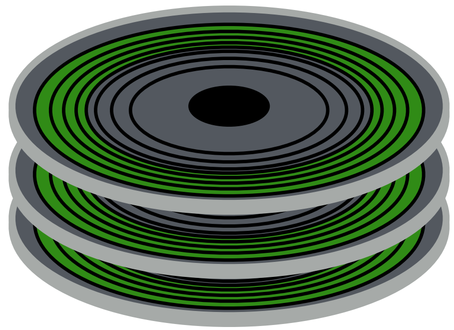


strategy: allocate inodes and data blocks in same group.

Groups

In FFS, groups were ranges of cylinders

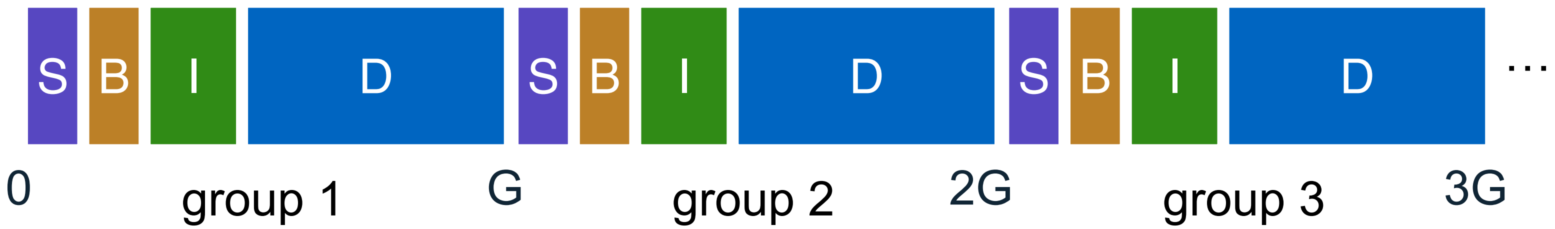
- called <u>cylinder group</u>



In ext2-4, groups are ranges of blocks

- called <u>block group</u>

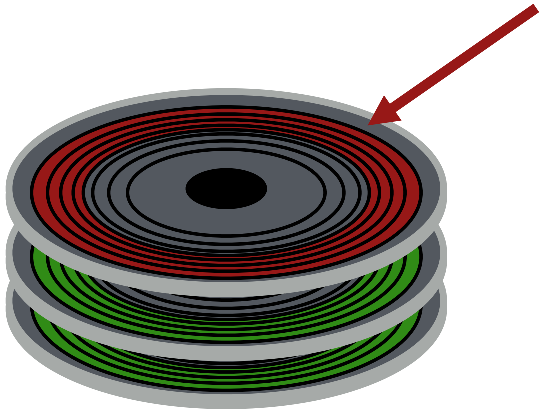
Placement Technique 3: Super Block



Is it useful to have multiple super blocks?

Yes, if some (but not all) fail.

Problem



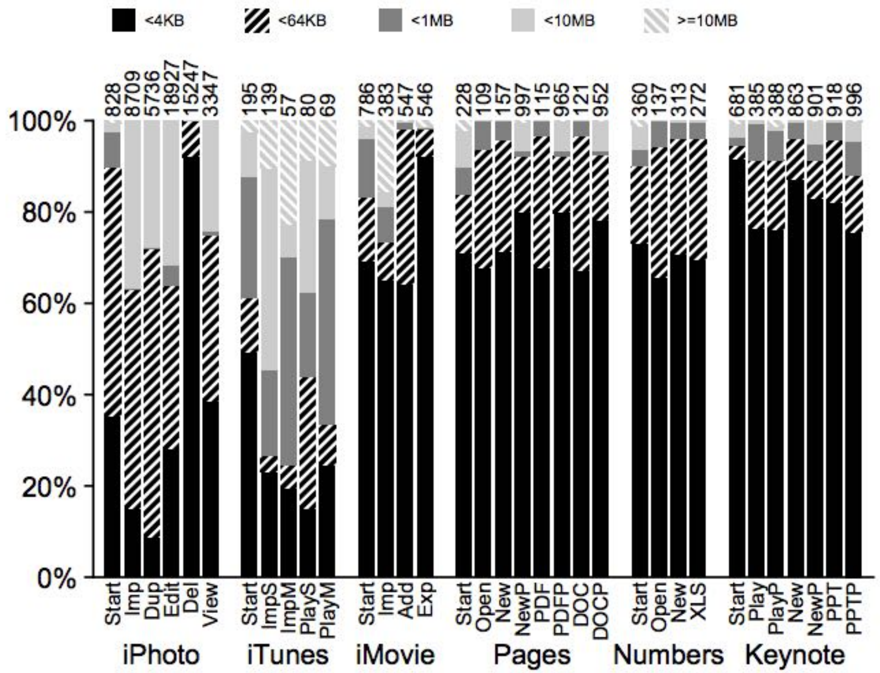
Old FS: All super-block copies are on the top platter.
Correlated failures! What if top platter dies?

Solution: for each group, store super-block at a different offset

Technique: Larger Blocks

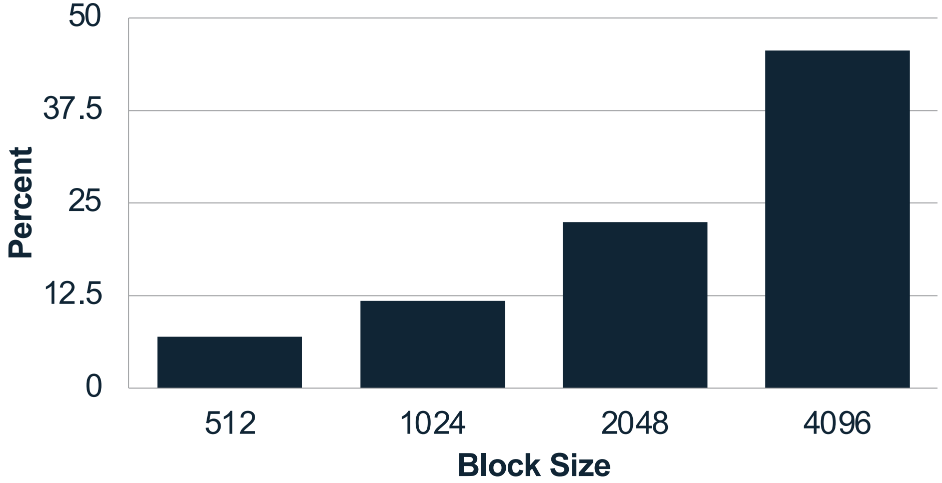
Most file are very small, even today!

Observation: Doubling block size for old FS over doubled performance
Why not make blocks huge?



THE UNIVERSITY of ADELAIDE

Larger Blocks



Lots of waste due to internal fragment in most blocks

Time vs. Space tradeoffs…

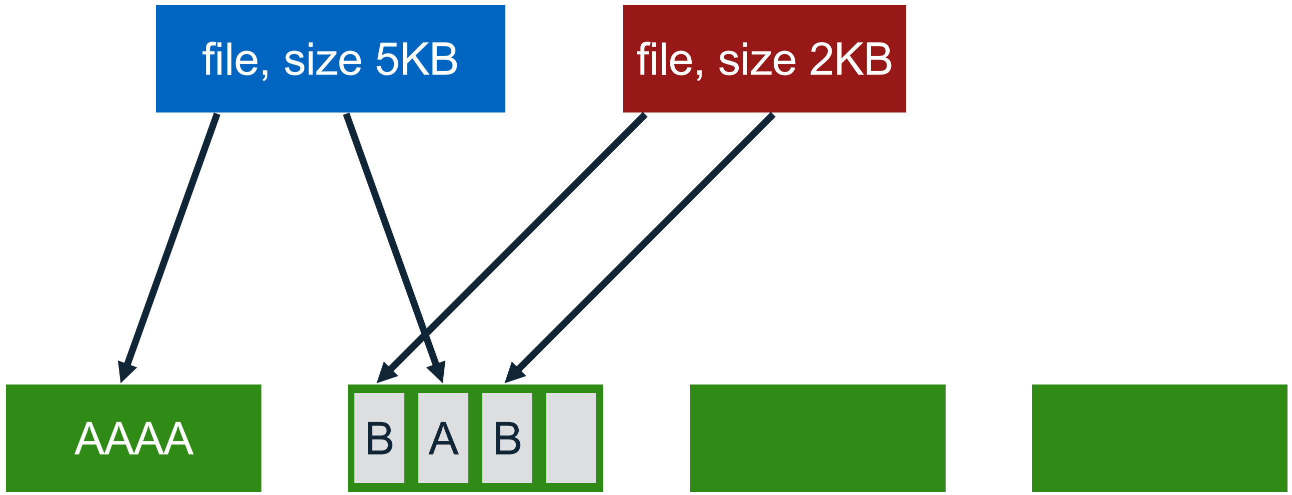
Solution: Fragments

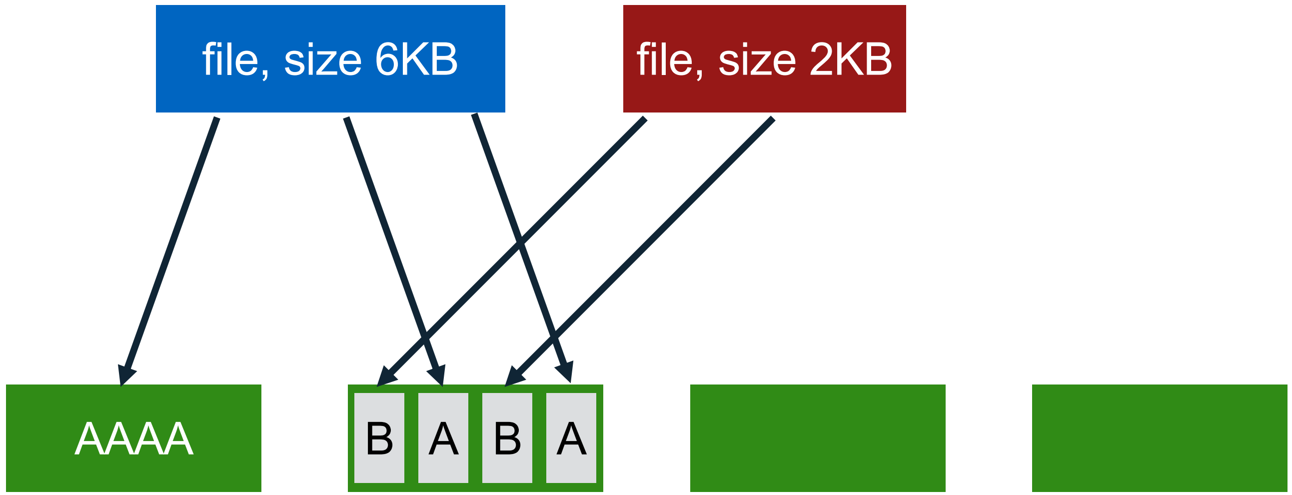
Hybrid – combine best of large blocks and best of small blocks

Use large block when file is large enough

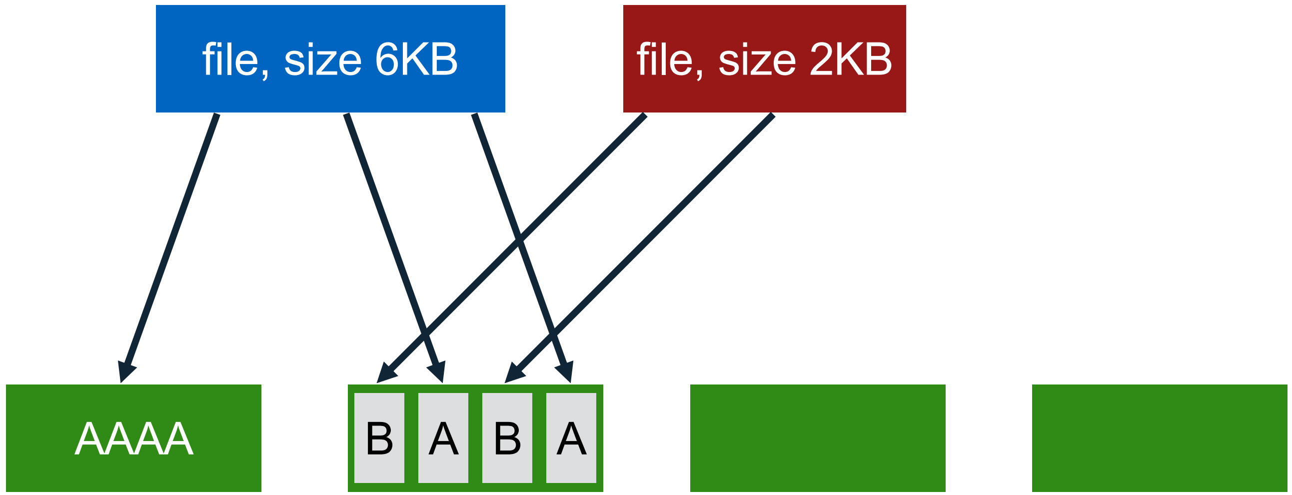
Introduce "fragment" for files that use parts of blocks

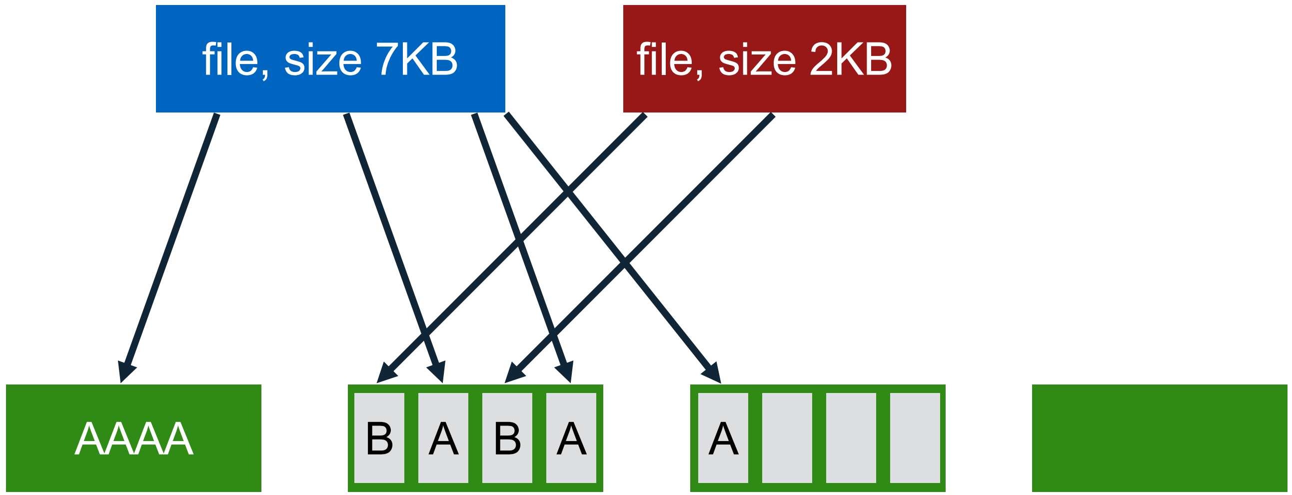
- Only tail of file uses fragments


file, size 5KB
file, size 2KB
AAAA
B
A
B



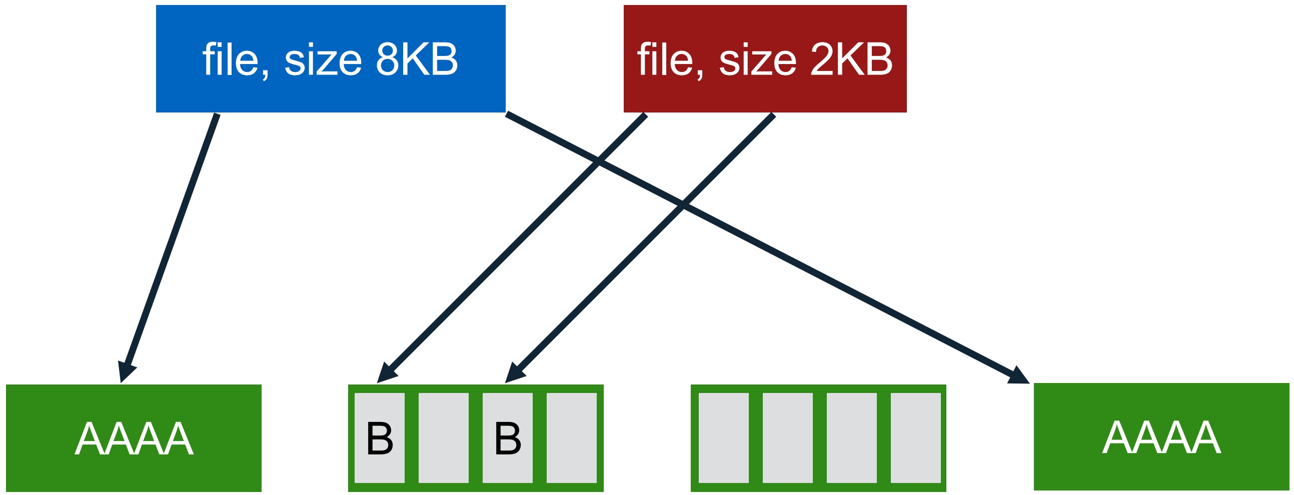
append A to first file


file, size 6KB
file, size 2KB
AAAA
B
A
B
A



append A to first file
Not allowed to use fragments across multiple blocks!

What to do instead?



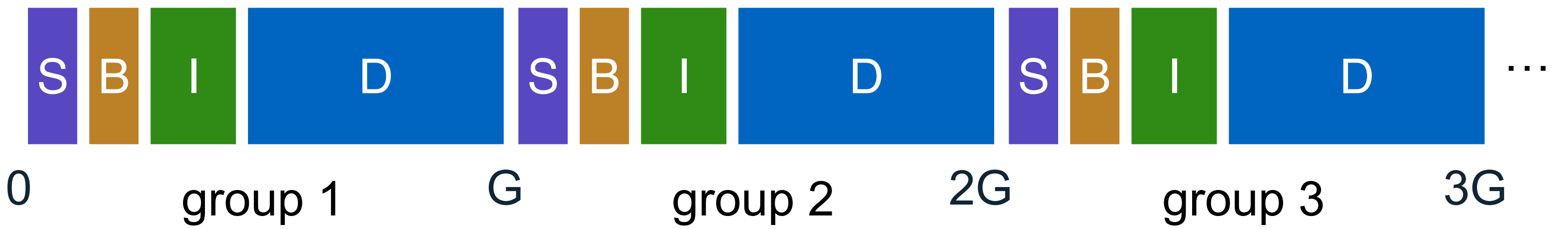
append A to first file,
copy to fragments to new block

Optimal Write Size

Writing less than a block is inefficient

Solution: new API exposes optimal write size

Smart Policy



Where should new inodes and data blocks go?

Placement Strategy

Put related pieces of data near each other.

Rules:

1. Put directory entries near directory inodes.
2. Put inodes near directory entries.
3. Put data blocks near inodes.

Problem: File system is one big tree

All directories and files have a common root.

All data in same FS is related in some way

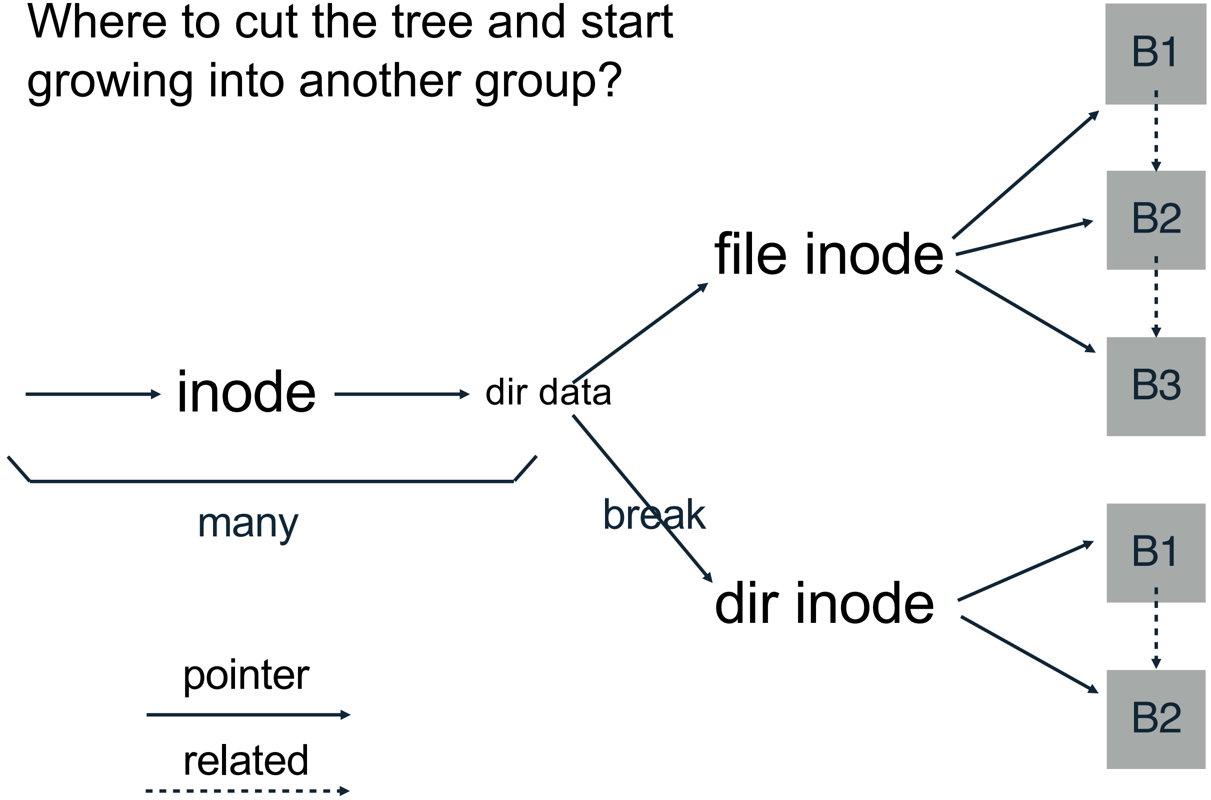
Trying to put everything near everything else doesn't make any choices!

Revised Strategy

Put more-related pieces of data near each other

Put less-related pieces of data far from each other

Where to cut the tree and start growing into another group?



FFS puts dir inodes in a new group

"`ls`" is fast on directories with many files.

Preferences

File inodes: allocate in <u>same</u> group with dir

Dir inodes: allocate in <u>new</u> group with fewer used inodes than average group

First data block: allocate near inode

Other data blocks: allocate near previous block

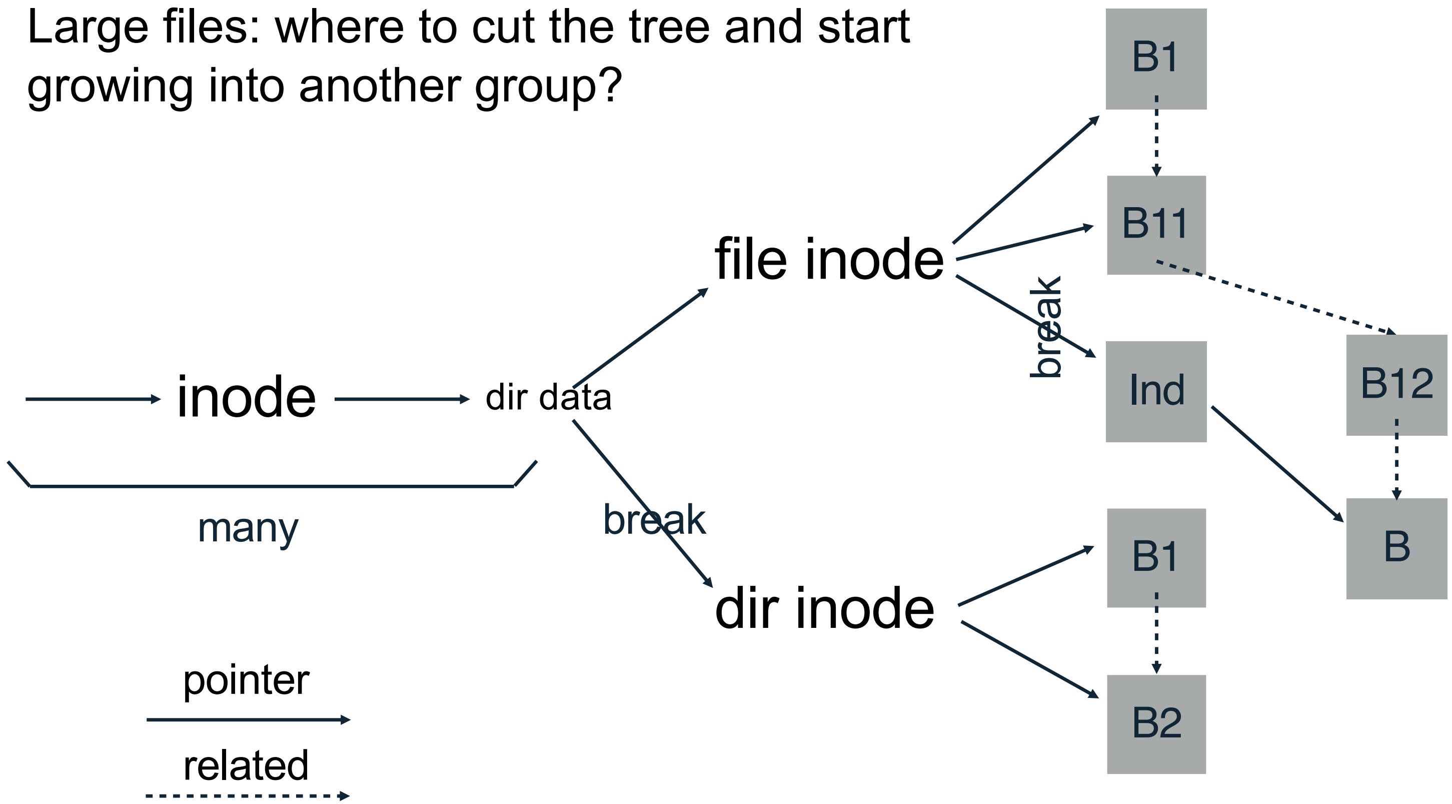
Problem: Large Files

Single large file can fill nearly all of a group

Displaces data for many small files

Better to do one seek for large file than one seek for each of many small files

Large files: where to cut the tree and start growing into another group?



Define “large” as requiring an indirect block

Starting at indirect (e.g., after 48 KB) put blocks in a new block group.

Preferences

File inodes: allocate in same group with dir

Dir inodes: allocate in new group with fewer used inodes than average group

First data block: allocate near inode

Other data blocks: allocate near previous block

Large file data blocks: after 48KB, go to new group. Move to another group (w/ fewer than avg blocks) every subsequent 1MB.

THE UNIVERSITY of ADELAIDE

Conclusion

First disk-aware file system

- Bitmaps
- Locality groups
- Rotated superblocks
- Large blocks
- Fragments
- Smart allocation policy

FFS inspired modern files systems, including ext2 and ext3

FFS also introduced several new features:

- long file names
- atomic rename
- symbolic links

THE UNIVERSITY of ADELAIDE

COMP SCI 3004 Operating Systems

FSCK & Journaling

make history.

THE UNIVERSITY of ADELAIDE

Crash Consistency

Unlike most data structures, 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 and current file systems.

- fsck (file system checker)
- Journaling (write-ahead logging)

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*

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?

THE UNIVERSITY of ADELAIDE

File System Redundancy Example

Superblock: field contains the total number of blocks in FS

DATA = N

Inode: field contains a 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

THE UNIVERSITY of ADELAIDE

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

THE UNIVERSITY of ADELAIDE

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

THE UNIVERSITY of ADELAIDE

Consistency

File system is appending to a file and must update:

- inode
- data bitmap
- data block

What happens if crash after only updating some parts of those tasks?

a) **bitmap**: lost block on disk

b) **data**: "nothing bad"

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

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

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

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

THE UNIVERSITY of ADELAIDE

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

THE UNIVERSITY of ADELAIDE

Consistency Solution #1: FSCK Checks

Hundreds of types of checks over different fields…

Do superblocks match?

Do directories contain “.” and “..”?

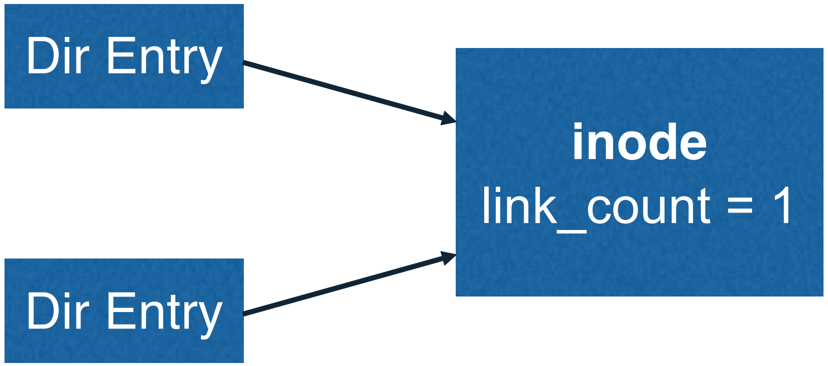
Do number of dir entries equal inode link counts?

Do different inodes ever point to same block?

…

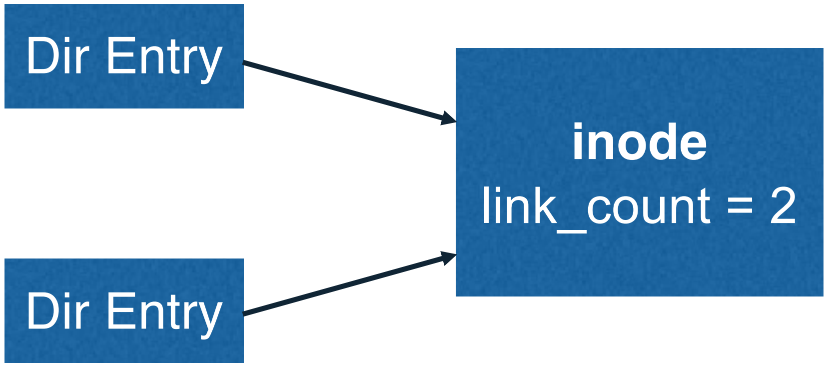
How to solve problems?

Link Count (example 1)



How to fix to have consistent file system?

Link Count (example 1)



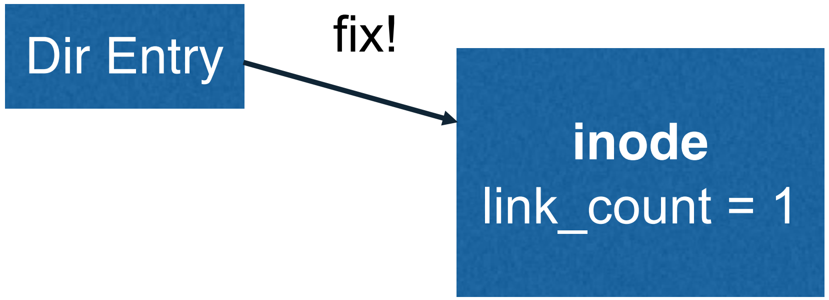
Simple fix!

Link Count (example 2)

inode
link_count = 1

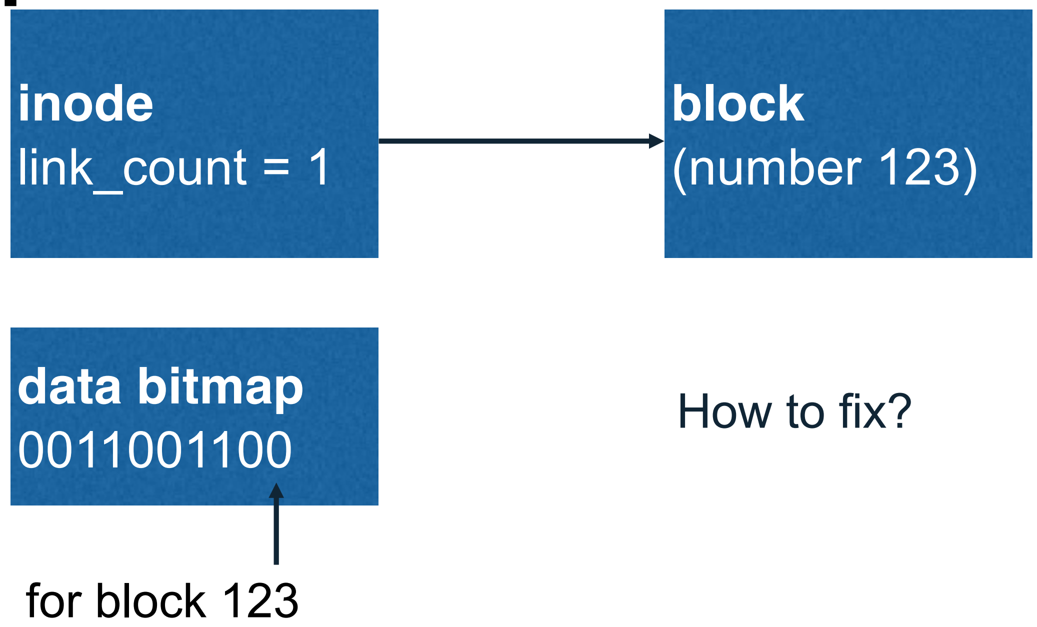
How to fix???

Link Count (example 2)



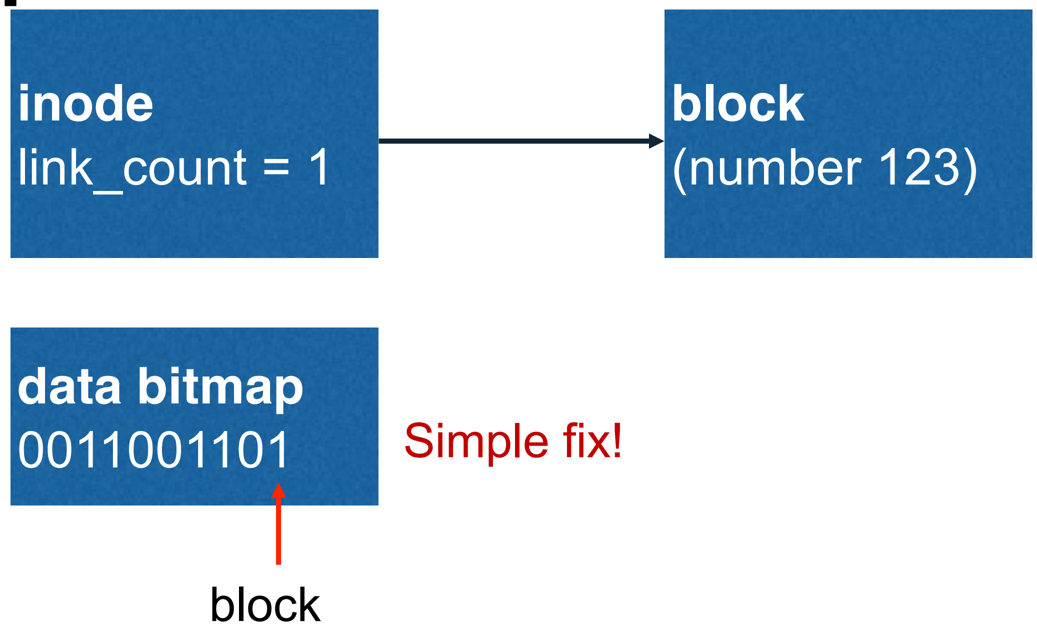

```
ls -l /
total 150
drwxr-xr-x  401 18432 Aug 31  1969 afs/
drwxr-xr-x    2 4096  Oct  3 09:42 bin/
drwxr-xr-x    5 4096  Sep  1 14:21 boot/
dr-xr-xr-x   13 4096  Oct  3 09:41 lib/
dr-xr-xr-x   10 12288 Aug  3 09:41 lib64/
drwx------    2 16384 Aug  1 10:57 lost+found/
...
```

Data Bitmap



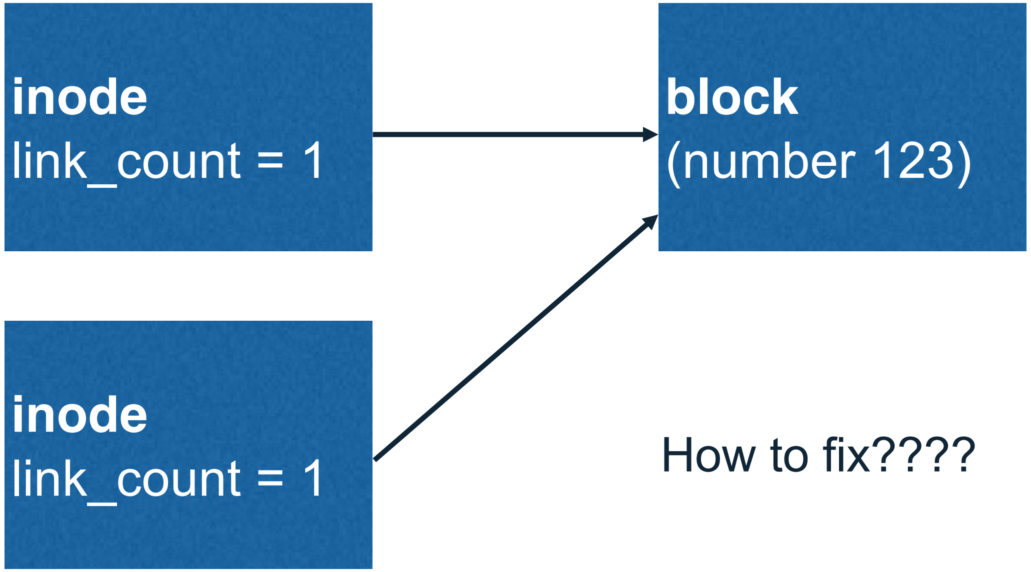
How to fix?

Data Bitmap



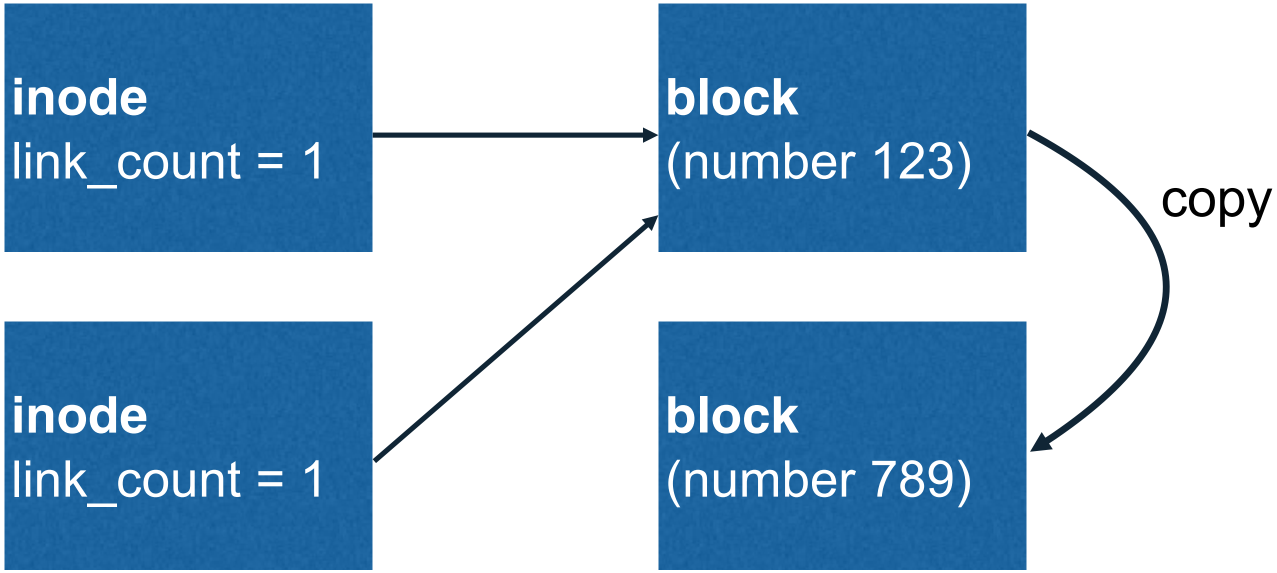
inode
link_count = 1
block
(number 123)
data bitmap
0011001101
block
Simple fix!

Duplicate Pointers

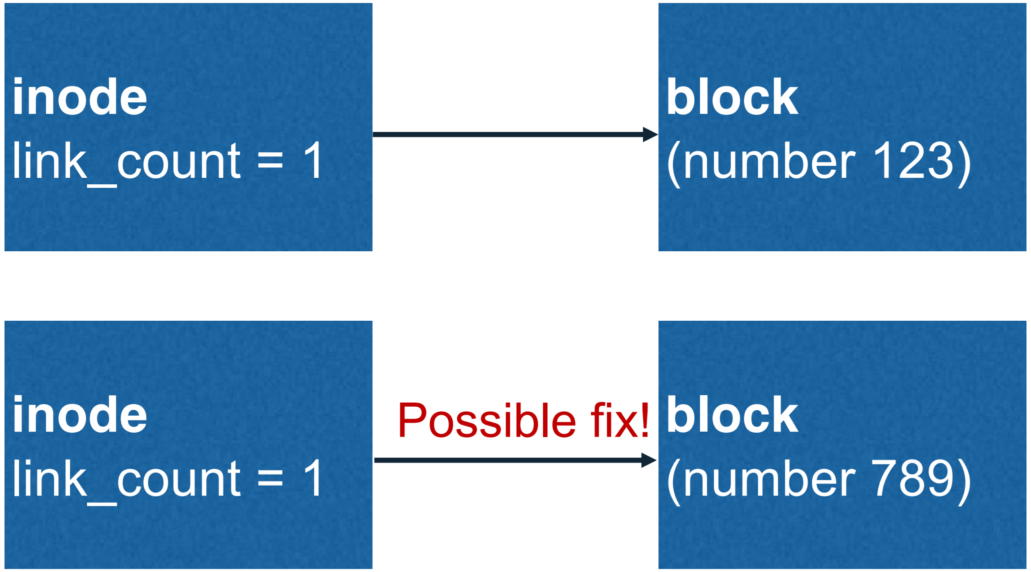


How to fix????

Duplicate Pointers



Duplicate Pointers



But is this correct?

Bad Pointer

inode
link_count = 1 → 9999

super block
tot-blocks=8000

How to fix???

Bad Pointer

inode
link_count = 1

Simple fix! (But is this correct?)

super block
tot-blocks=8000

Problems with fsck

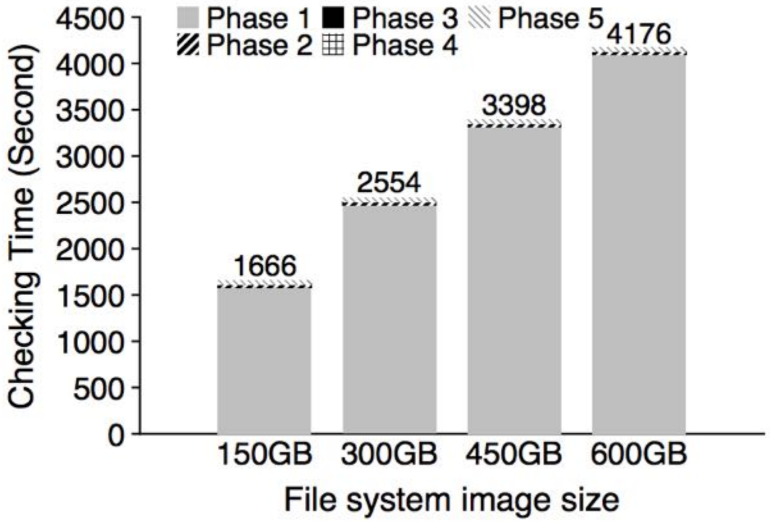
Problem 1:

Not always obvious how to fix file system image

Don’t know “correct” state, just consistent one

Easy way to get consistency: reformat disk!

Problem 2: fsck is very slow



Checking a 600GB disk takes ~70 minutes

Consistency Solution #2: Journaling

Goals

Okay to do some **recovery work** after crash, but not to read the entire disk

Don’t move file system to just any consistent state, get the **correct** state

Strategy

Atomicity

Definition of atomicity for **concurrency**

Operations in critical sections are not interrupted by operations on related critical sections

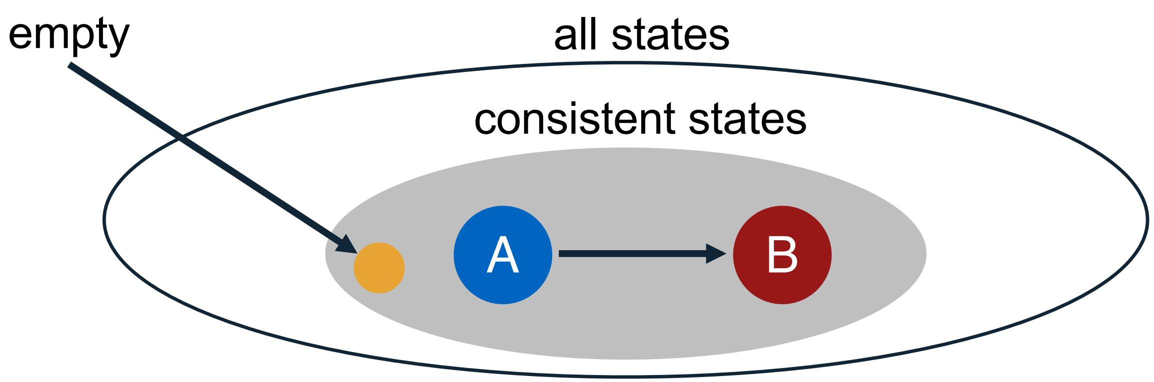
Definition of atomicity for **persistence**

- collections of writes are not interrupted by crashes;
  either (all new) or (all old) data is visible

THE UNIVERSITY of ADELAIDE

Consistency vs Correctness

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



fsck gives consistency
Atomicity gives A or B.

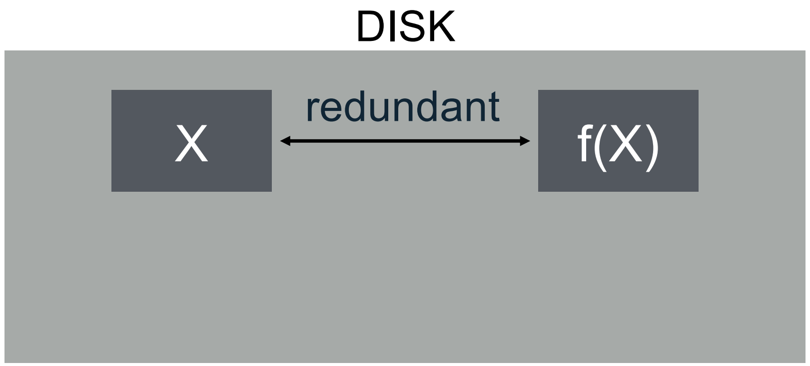
Journaling General Strategy

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

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

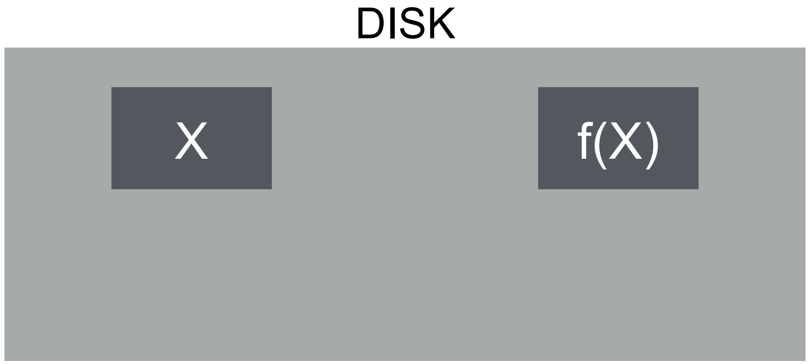
Fight Redundancy with Redundancy

Want to replace X with Y. Original:



Fight Redundancy with Redundancy

Want to replace X with Y. Original:

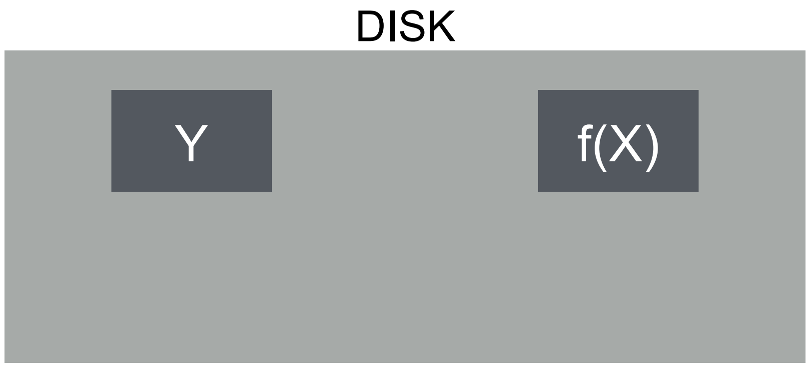


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

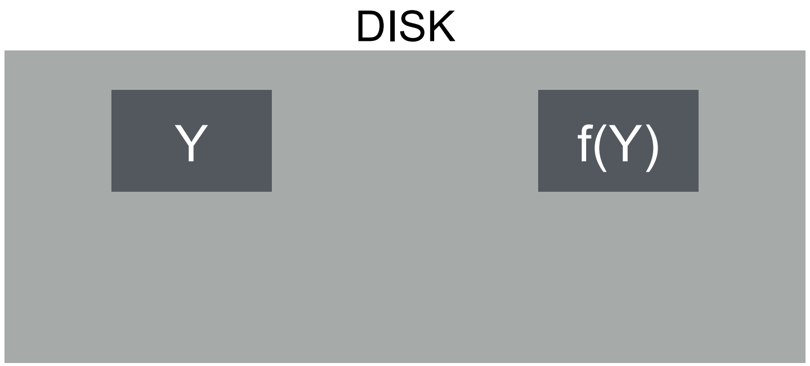


Good time to crash?

bad time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:



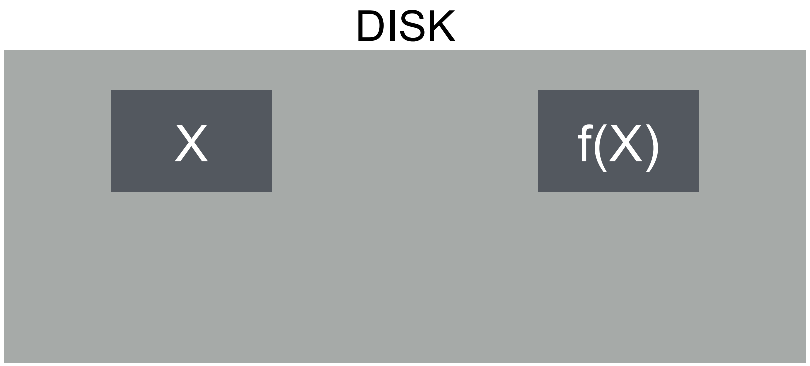
Good time to crash?

good time to crash

THE UNIVERSITY of ADELAIDE

Fight Redundancy with Redundancy

Want to replace X with Y. With journal:

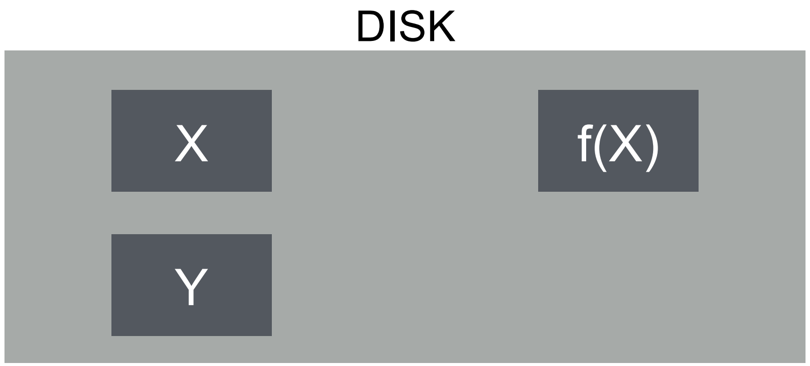


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

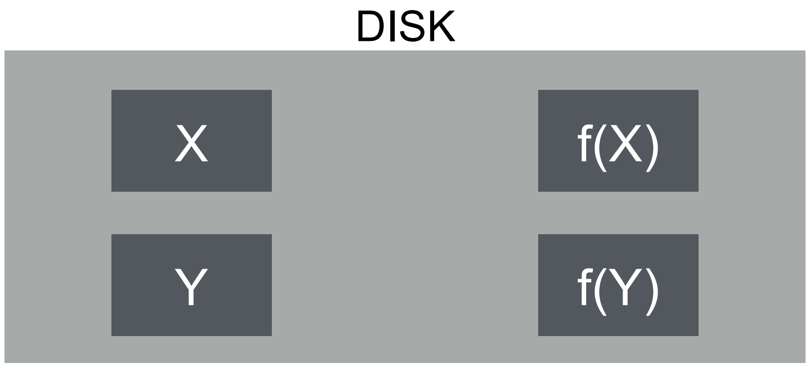
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

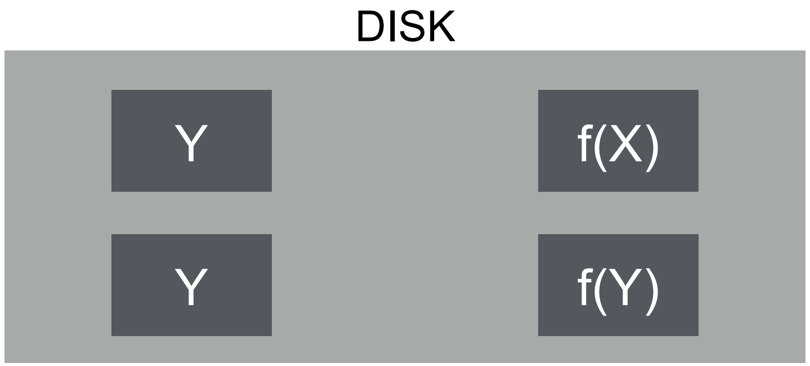
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

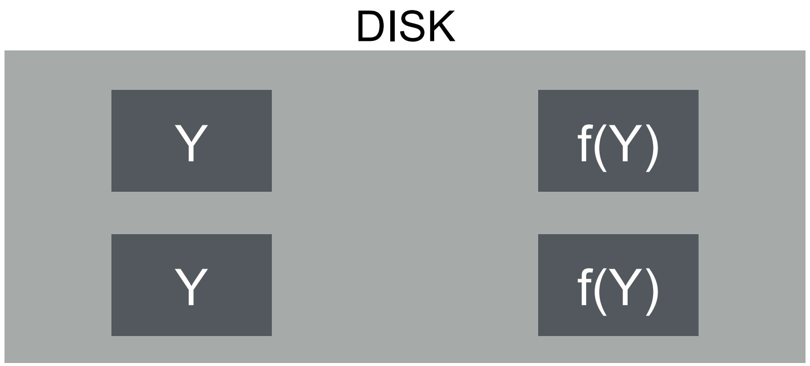
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

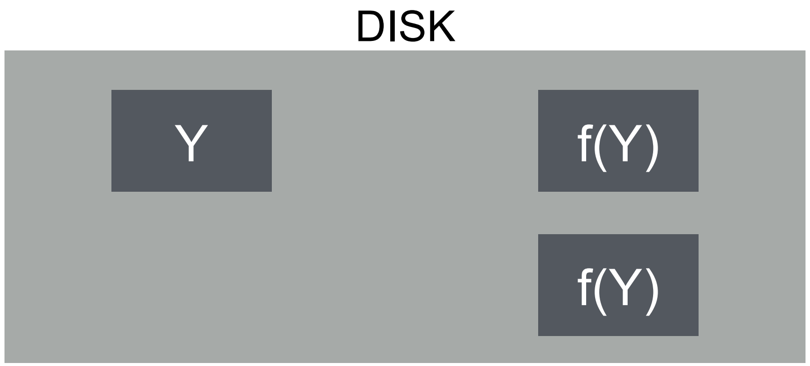
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

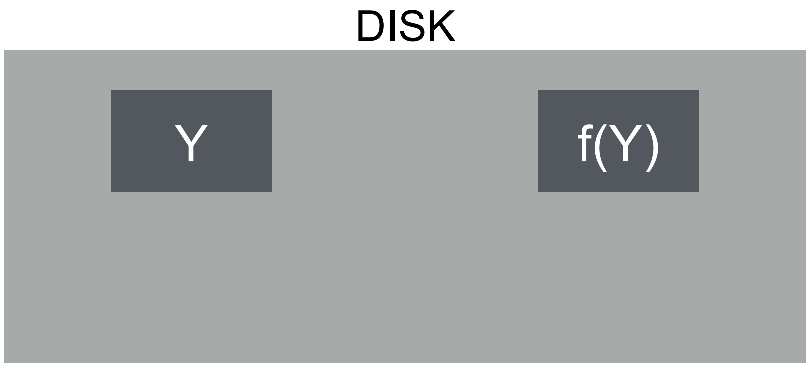
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

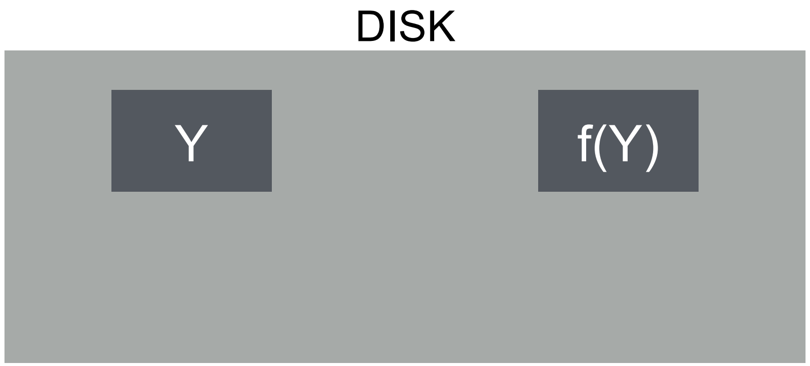
Want to replace X with Y. With journal:



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!

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”

THE UNIVERSITY of ADELAIDE

Small Journals

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

Goal:

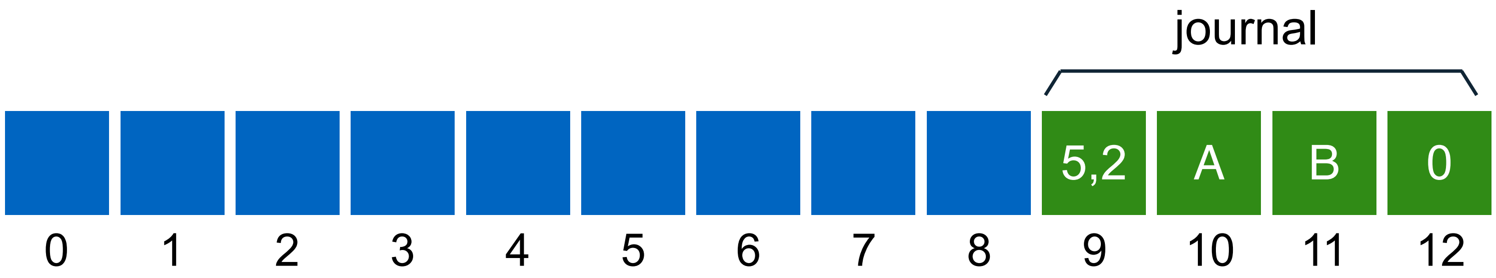
Reuse small area as backup for any block

How?

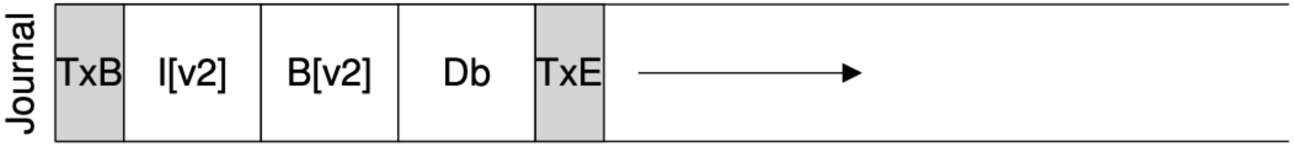
Store block numbers in a transaction header

THE UNIVERSITY of ADELAIDE

New Layout

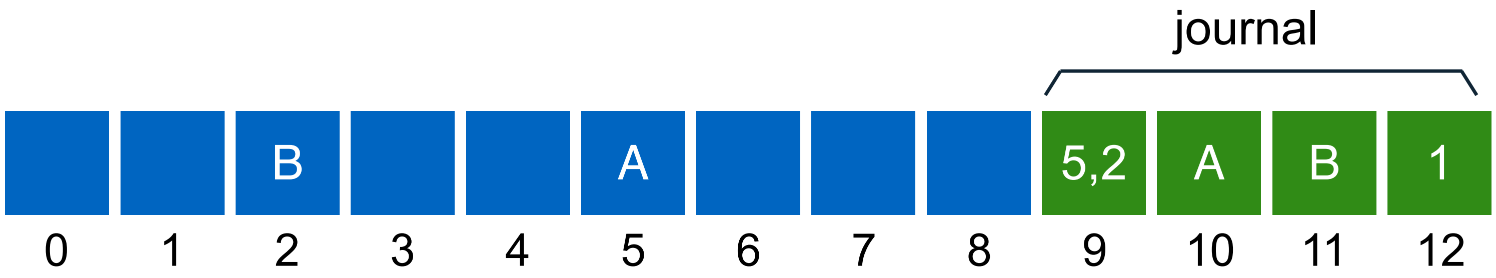


transaction: write A to **block 5**; write B to **block 2**



THE UNIVERSITY of ADELAIDE

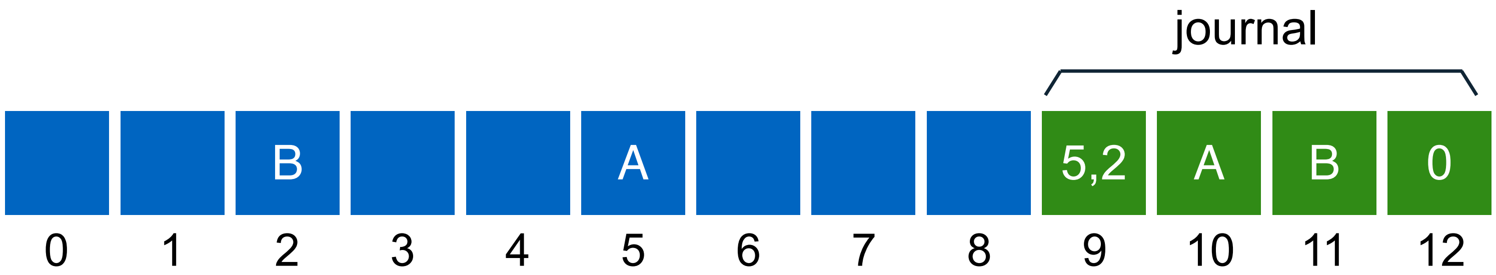
New Layout



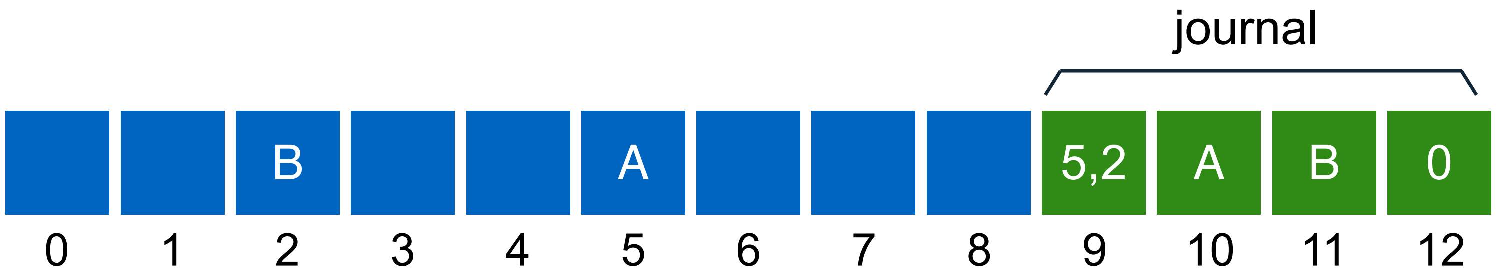
Transaction: write A to **block 5**; write B to **block 2**

Checkpoint: Writing new data to in-place locations

New Layout

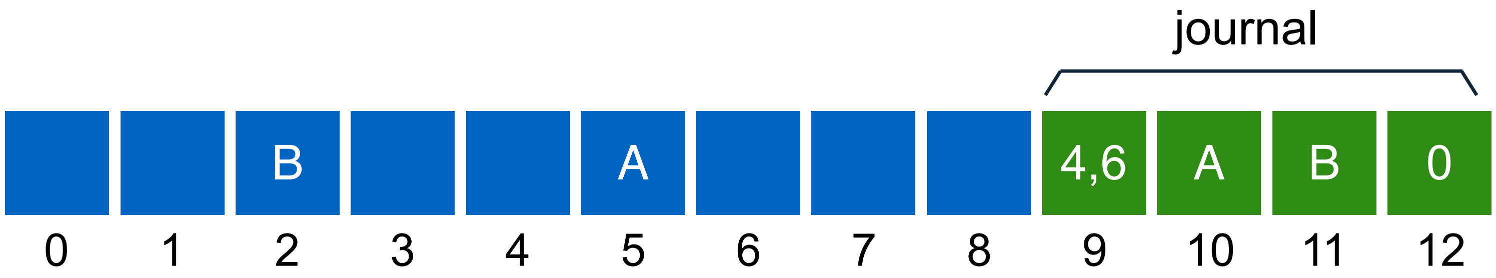


New Layout



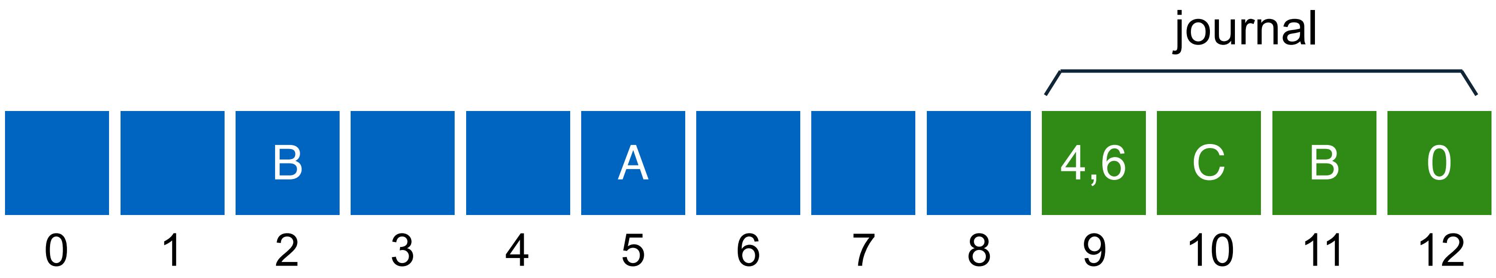
transaction: write C to **block 4**; write T to **block 6**

New Layout



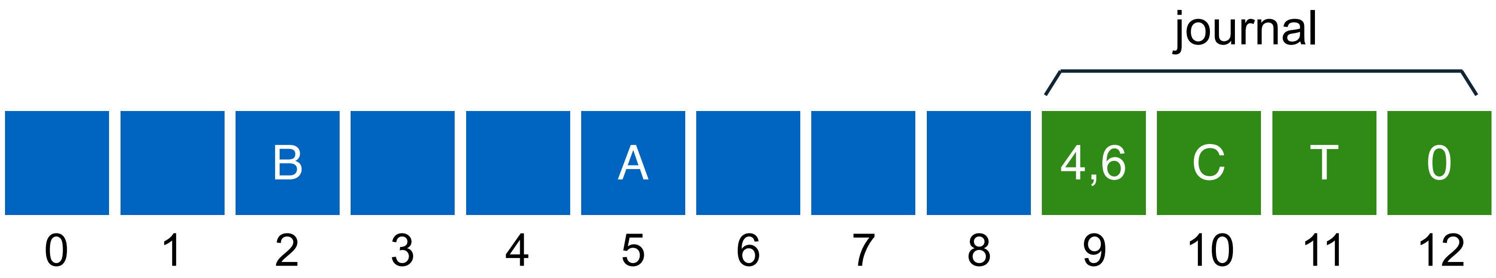
transaction: write C to **block 4**; write T to **block 6**

New Layout



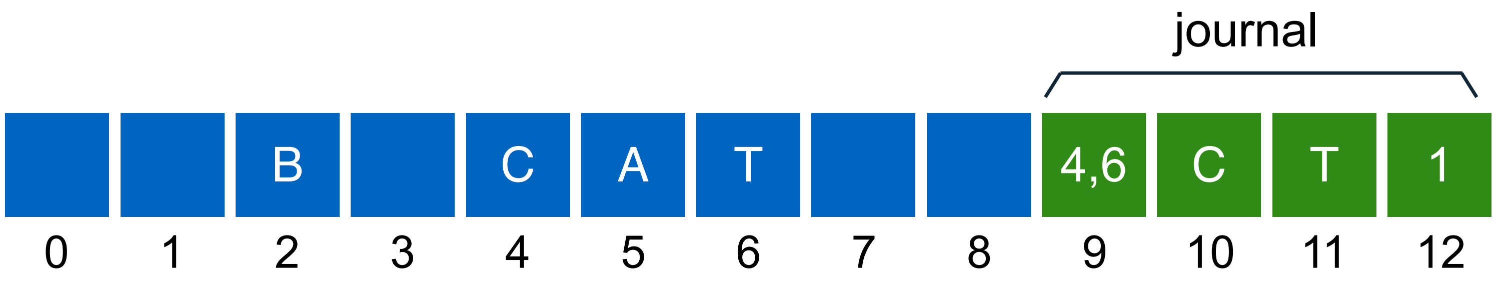
transaction: write C to **block 4**; write T to **block 6**

New Layout



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

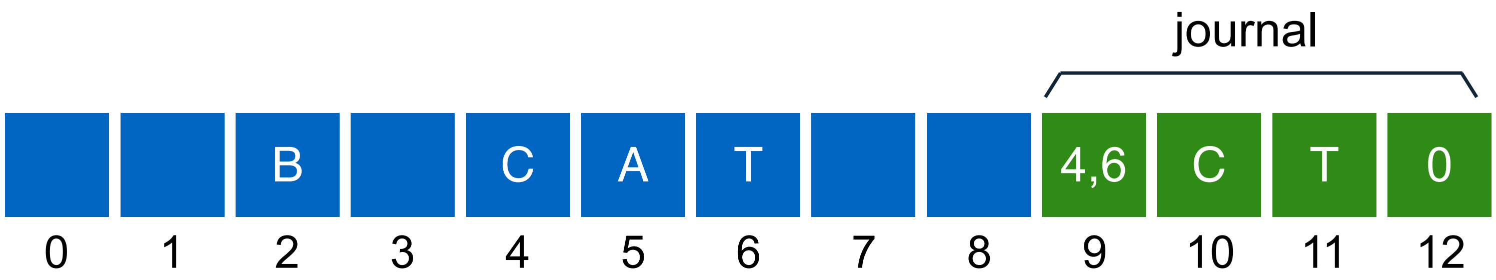
New Layout



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

Checkpoint: Writing new data to in-place locations

New Layout

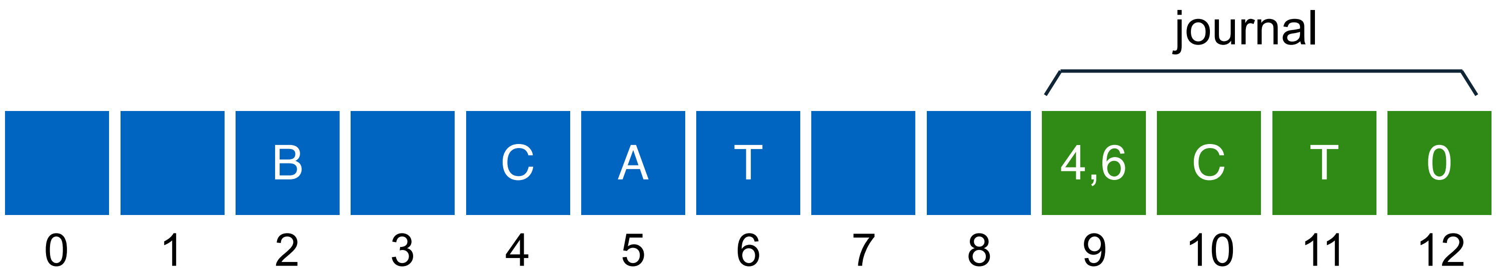


transaction: write C to **block 4**; write T to **block 6**

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**

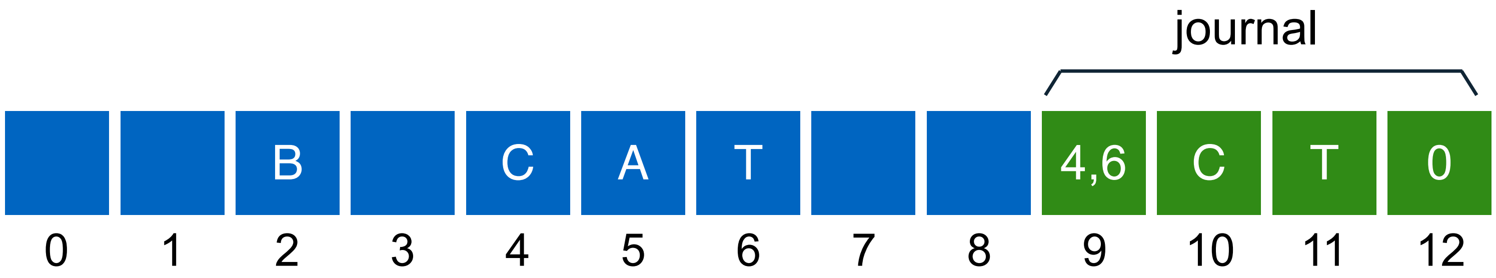
write order: 9, 10, 11, 12, 4, 6, 12

Enforcing total ordering is inefficient. Why?
Random writes

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

THE UNIVERSITY of ADELAIDE

Ordering



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

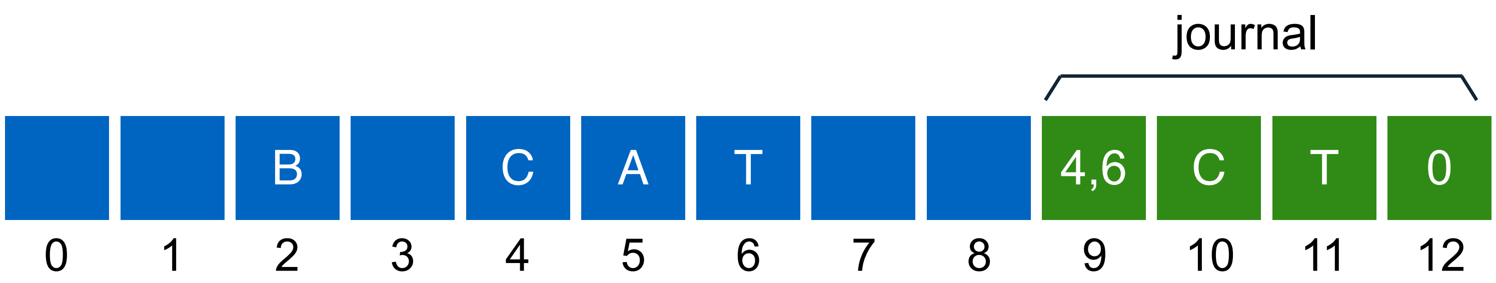
write order: 9,10,11 | 12 | 4,6 | 12

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

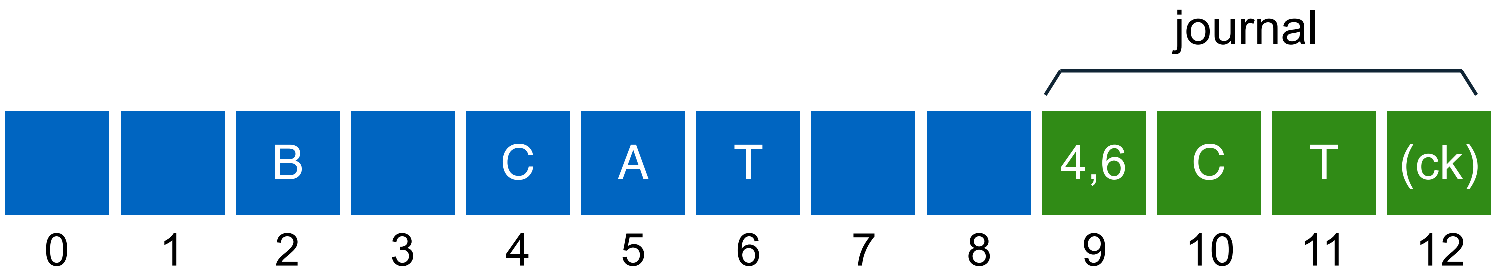
Checksum Optimization



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

Circular Buffer

Keep data also in memory until checkpointed on disk

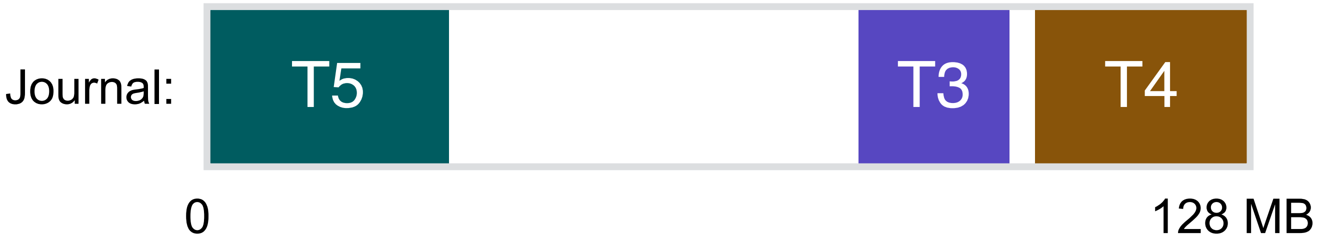
Circular Buffer

checkpoint and cleanup

Circular Buffer

transaction!

Circular Buffer



checkpoint and cleanup

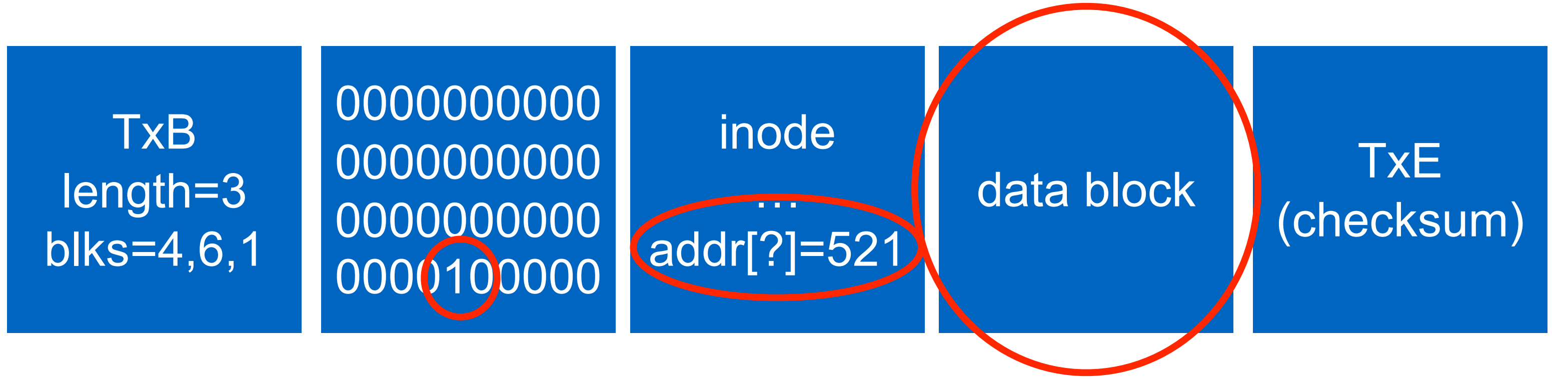
Optimizations

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

Physical Journal

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

Physical Journal



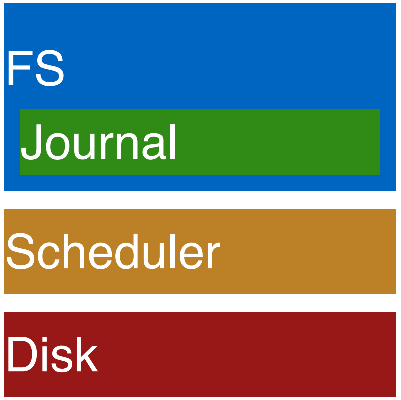
Actual changed data is much smaller!

Logical Journal

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

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

File System Integration



How to avoid writing all disk blocks Twice?

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

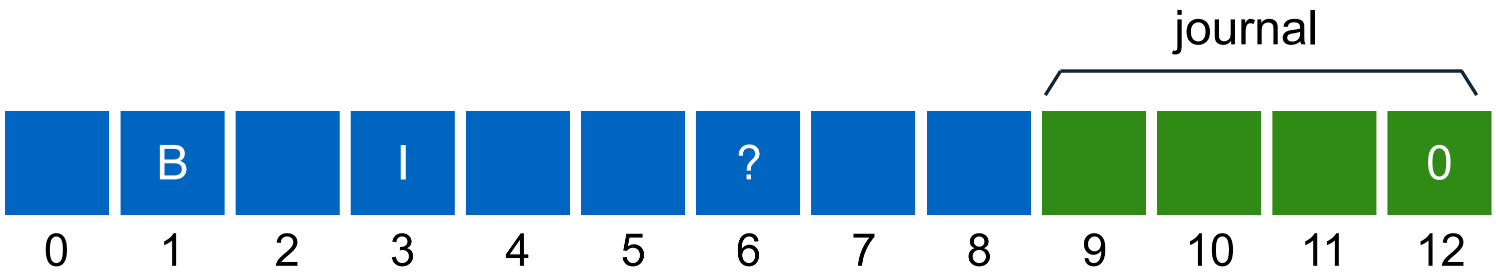
Strategy: journal all metadata, including:

superblock, bitmaps, inodes, indirects, directories

For regular data, write it back whenever convenient.

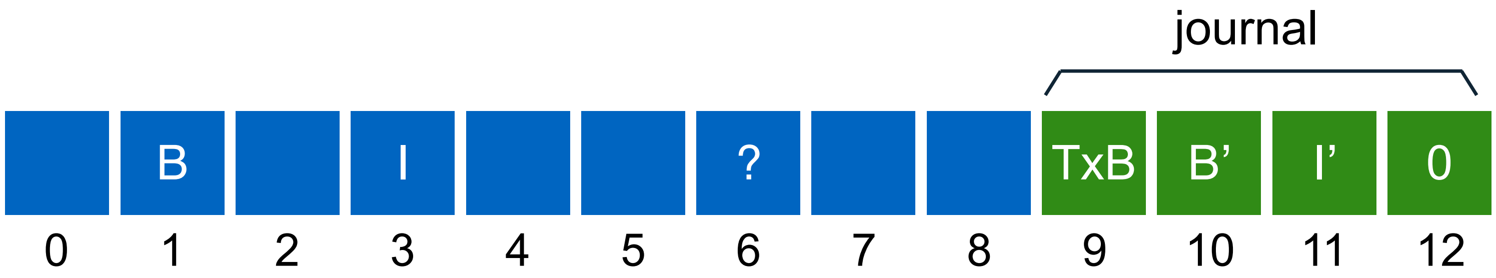
Of course, files may contain garbage.

Writeback Journal



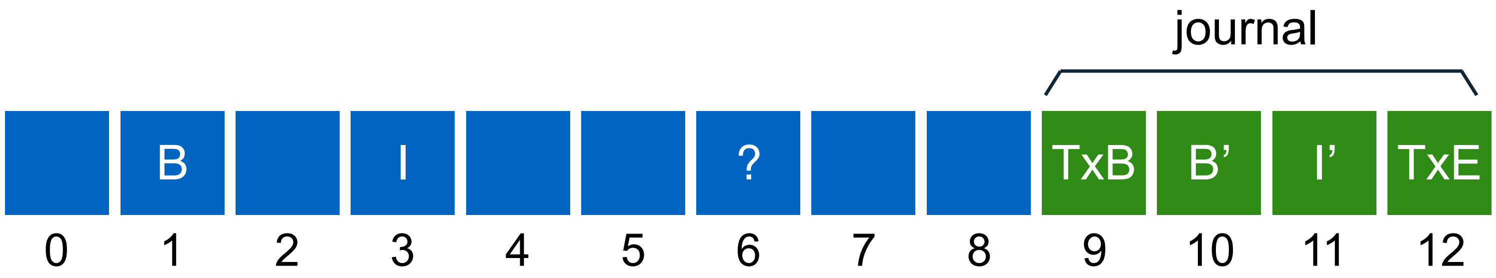
transaction: append to inode I

Writeback Journal



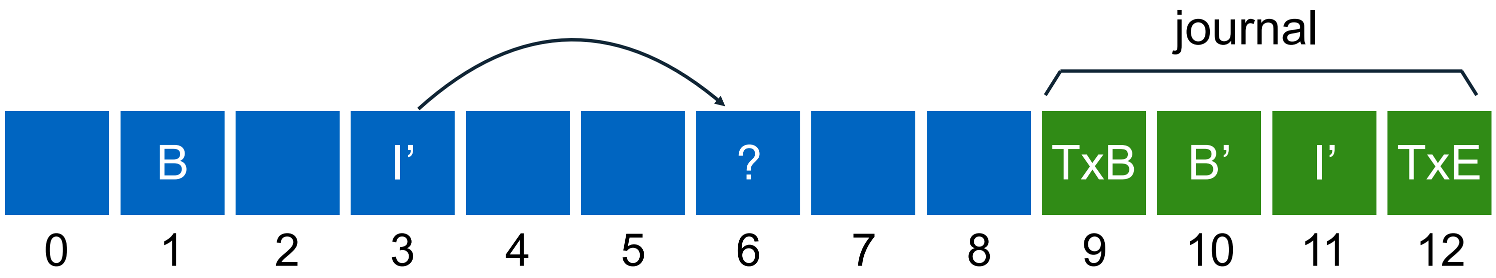
transaction: append to inode I

Writeback Journal



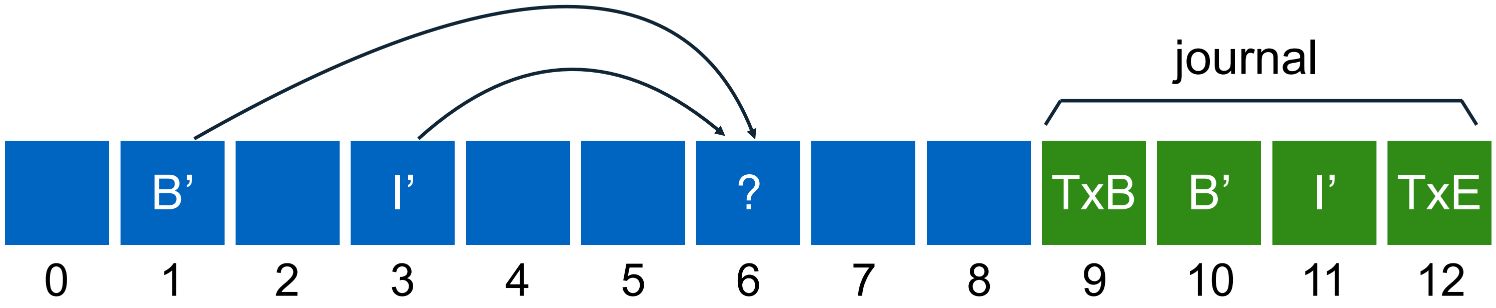
transaction: append to inode I

Writeback Journal



transaction: append to inode I

Writeback Journal



transaction: append to inode I

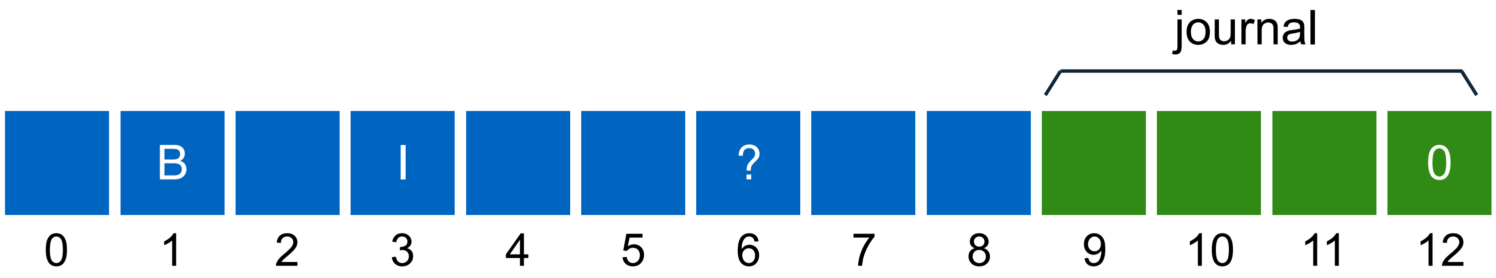
what if we crash now? Solutions?

Ordered Journaling

Still only journal metadata

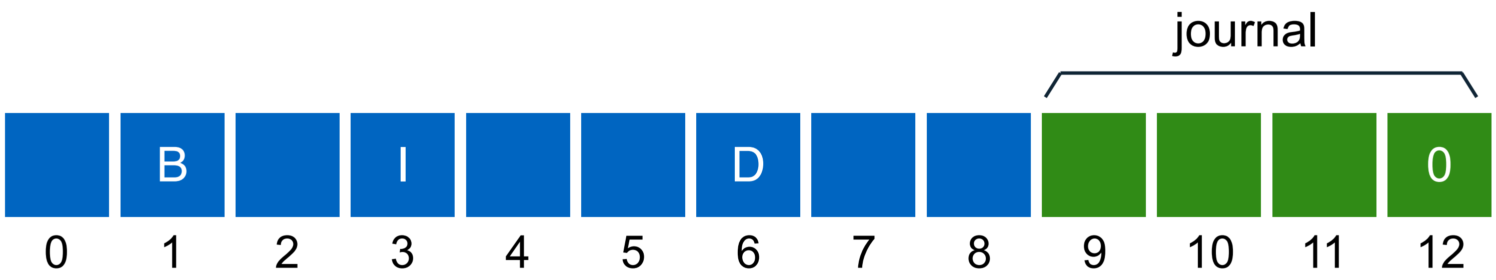
But write data before the transaction

Ordered Journal



transaction: append to inode I

Ordered Journal

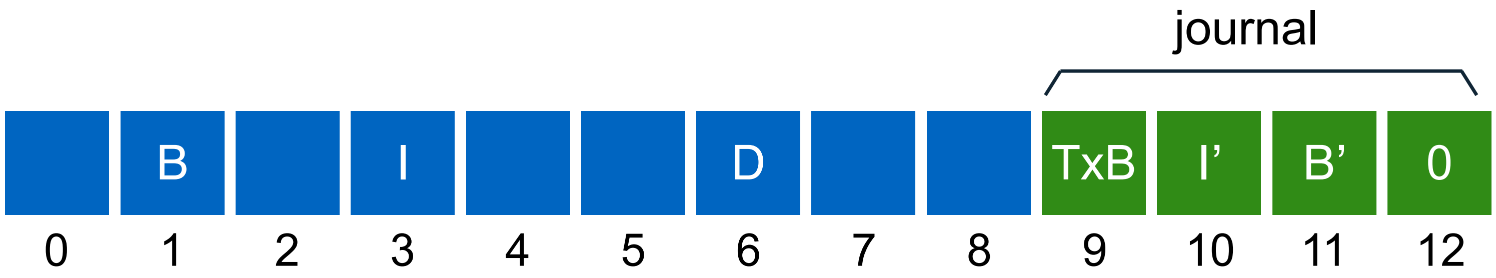


transaction: append to inode I

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

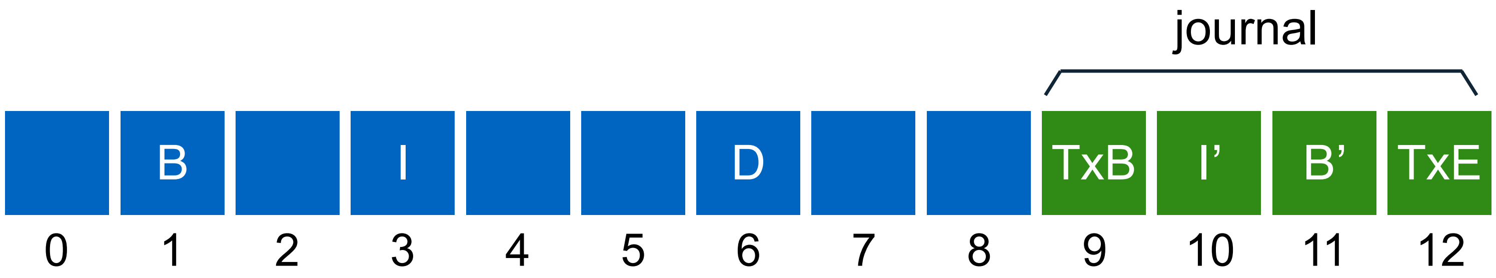
THE UNIVERSITY of ADELAIDE

Ordered Journal



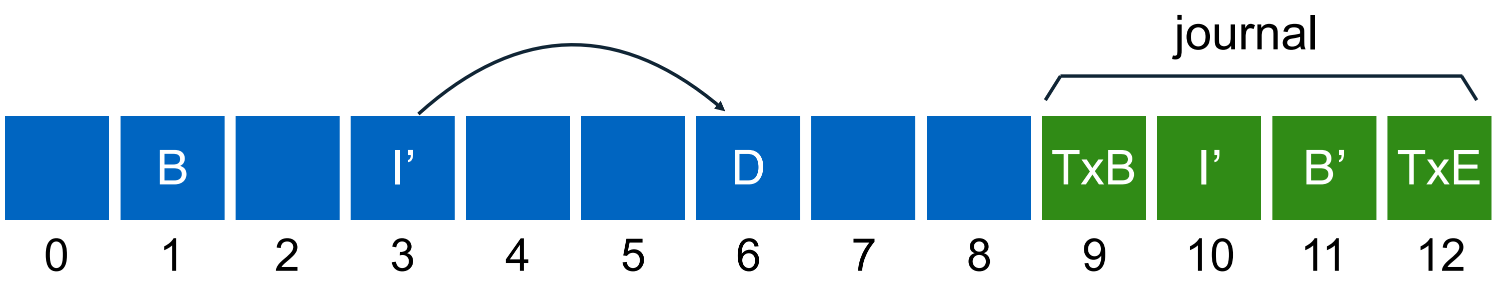
transaction: append to inode I

Ordered Journal



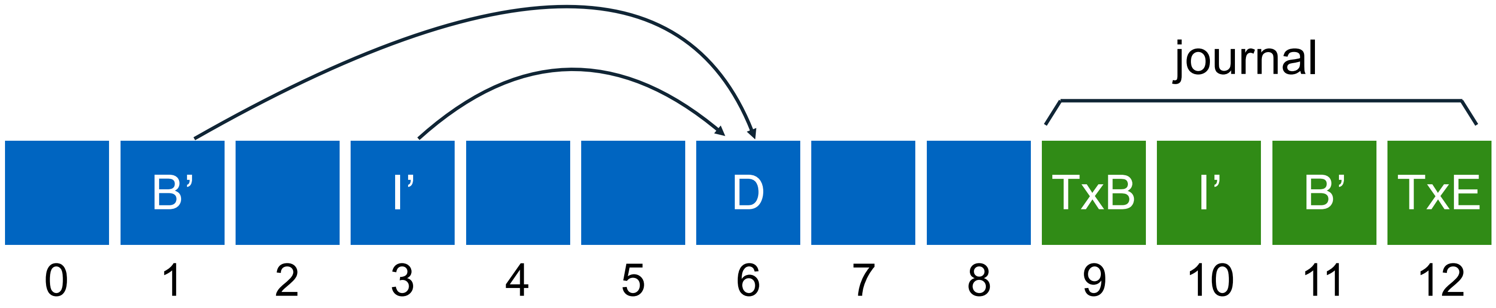
transaction: append to inode I

Ordered Journal



transaction: append to inode I

Ordered Journal



transaction: append to inode I

Conclusion

Most modern file systems use journals

ordered-mode for meta-data is popular

FSCK is still useful for weird cases

- bit flips

- FS bugs

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

COMP SCI 3004
Operating Systems

Log-structured File System (LFS)

make history.

THE UNIVERSITY of ADELAIDE

File-System Case Studies

Local

- FFS: Fast File System

- LFS: Log-Structured File System

Network

- NFS: Network File System

- AFS: Andrew File System

THE UNIVERSITY of ADELAIDE

General Strategy for Crash Consistency

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

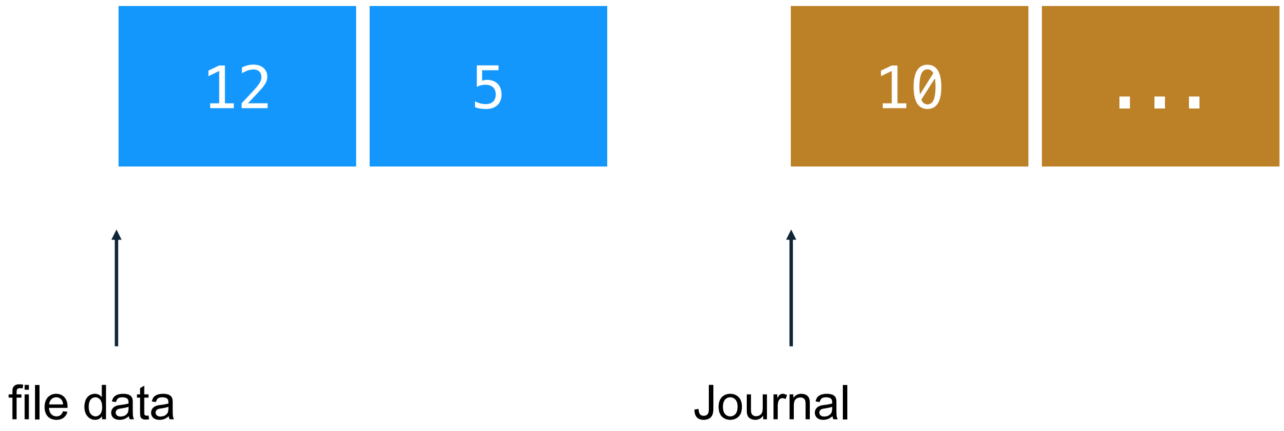
Implication:
At some point in time, all old AND all new data must be on disk

Two techniques popular in file systems:

1. **journal new info, then overwrite old info with new info in place**
2. **copy-on-write: write new info to new location, discard old info**

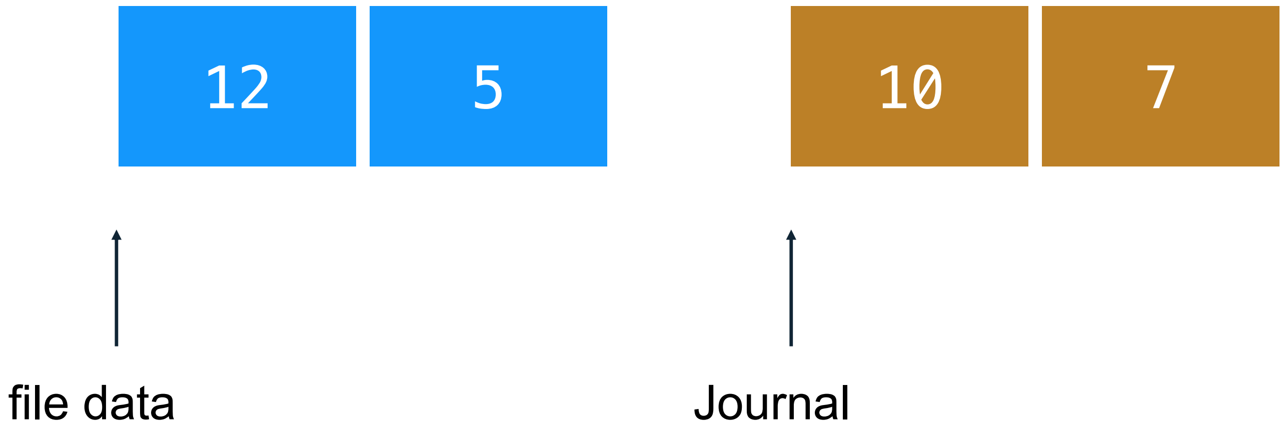
Journal New, Overwrite In-Place

Journal New, Overwrite In-Place



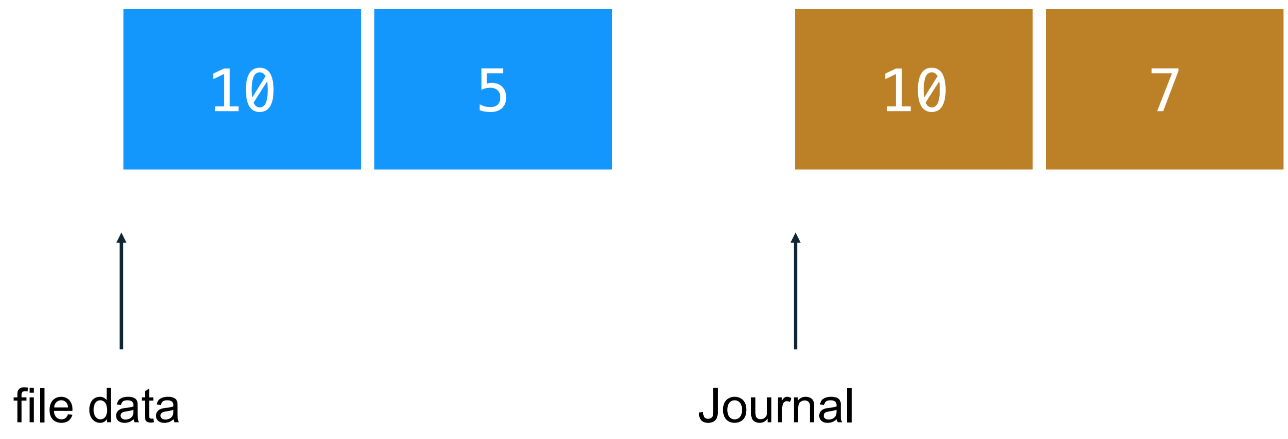
Imagine journal header describes in-place destinations

Journal New, Overwrite In-Place



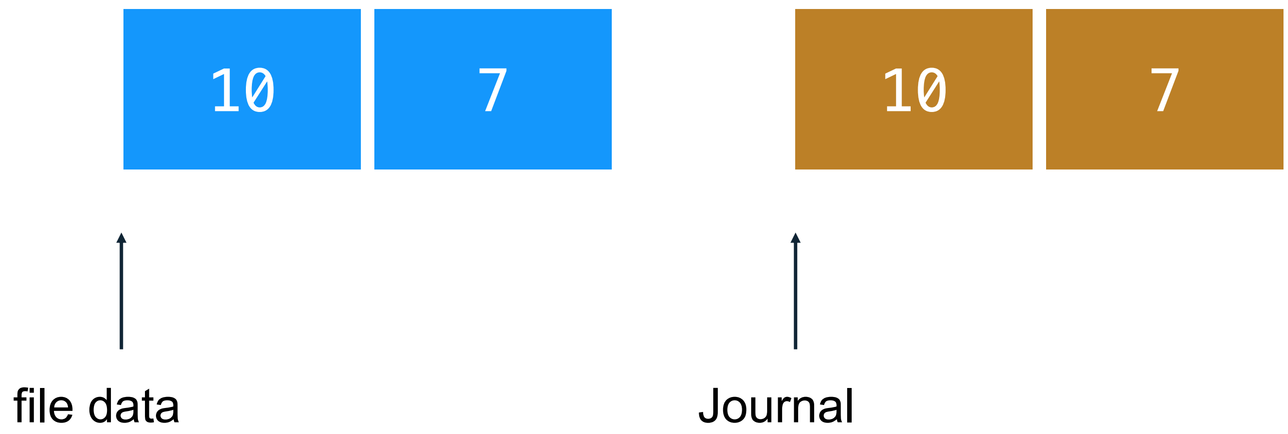
Imagine journal commit block designates transaction complete

Journal New, Overwrite In-Place

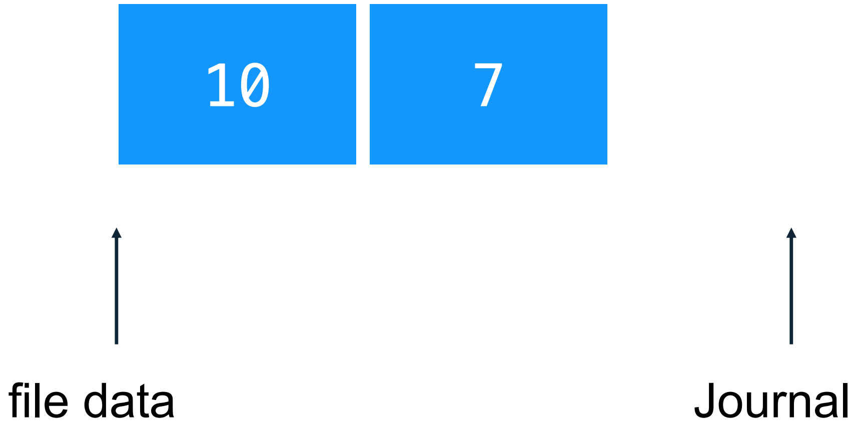


Perform checkpoint to in-place data when transaction is complete

Journal New, Overwrite In-Place



Journal New, Overwrite In-Place



Clear journal commit block to show checkpoint complete

Write New, Discard Old

file data

Make a copy-on-write (COW)

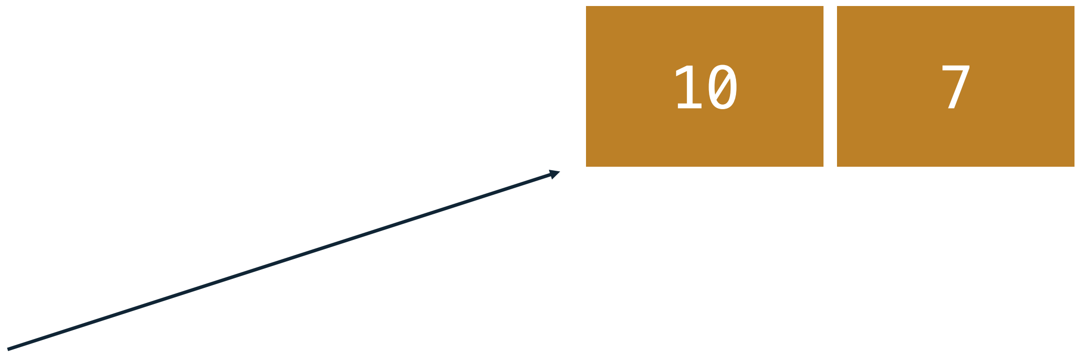
Write New, Discard Old

Write New, Discard Old

Write New, Discard Old

file data

Write New, Discard Old



file data

Obvious advantage?

Only write new data once instead of twice

LFS Performance Goal

Motivation:

Growing gap between sequential and random I/O performance

RAID-5 especially bad with small random writes

Idea: use disk purely sequentially

Easy for writes to use disk sequentially – why?

Can do all writes near each other to empty space – new copy

Works well with RAID-5 (large sequential writes)

Hard for reads – why?

User might read files X and Y not near each other on disk

Maybe not be too bad if disk reads are slow – why?

- Memory sizes are growing (cache more reads)

LFS Strategy

File system buffers writes in main memory until “enough” data

How much is enough?

Enough to get good sequential bandwidth from disk (MB)

Write buffered data sequentially to new segment on disk

Never overwrite old info: old copies left behind

THE UNIVERSITY of ADELAIDE

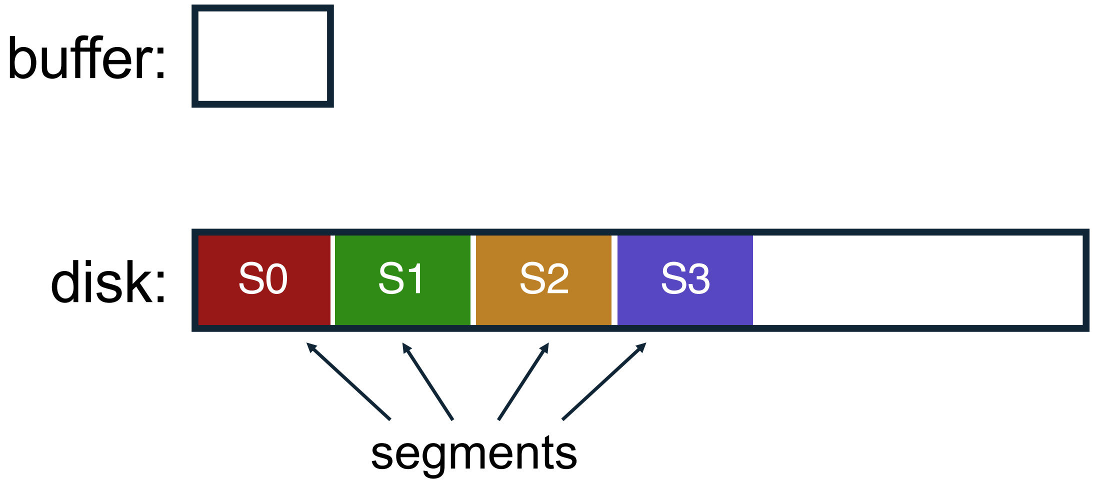
Big Picture

Big Picture

Big Picture

Big Picture

Big Picture



Data Structures (attempt 1)

What data structures from FFS can LFS remove?

allocation structs: data + inode bitmaps

What type of name is much more complicated?

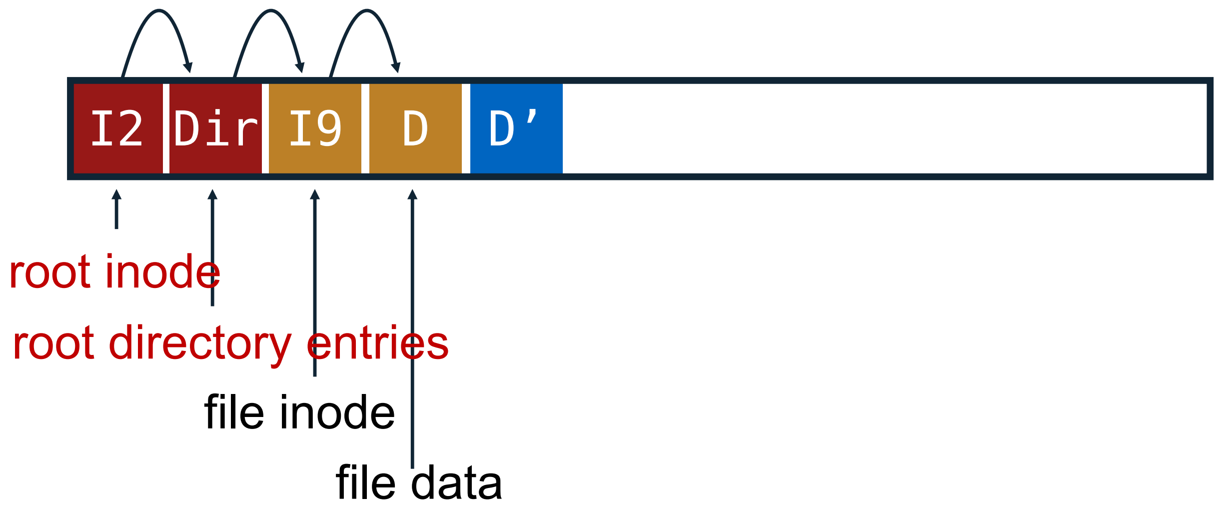
Inodes are no longer at fixed offset

Use **current offset on disk** instead of table index for name

Note: when update inode, inode number changes!!

Attempt 1

Overwrite data in /file.txt



How to update Inode 9 to point to new D' ???

THE UNIVERSITY of ADELAIDE

Attempt 1

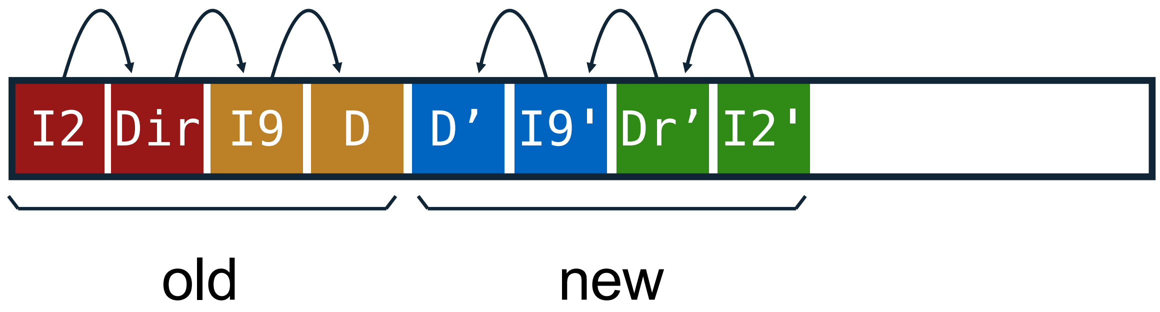
Overwrite data in /file.txt

Can LFS update Inode 9 to point to new D'?

NO! This would be a random write

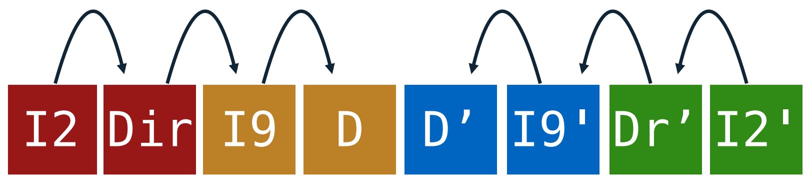
Attempt 1

Overwrite data in /file.txt



Must update all structures in sequential order to log

Attempt 1: Problem w/ Inode Numbers



Problem:

For every data update, must propagate updates all the way up directory tree to root

Why?

When inode copied, its location (inode number) changes

Solution:

Keep inode numbers constant; don’t base name on offset

FFS found inodes with math. How now?

Data Structures (attempt 2)

What data structures from FFS can LFS remove?

- allocation structs: data + inode bitmaps

What type of name is much more complicated?

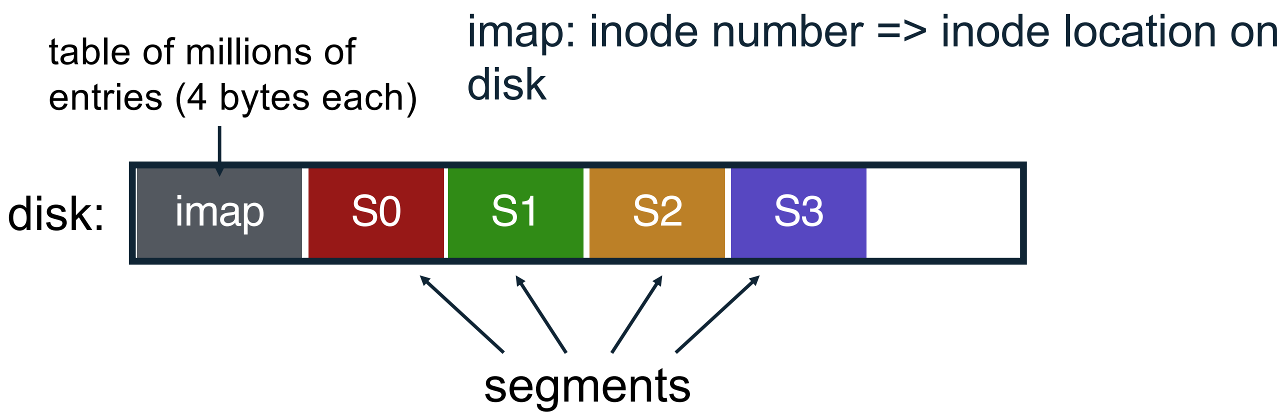
- Inodes are no longer at fixed offset

  Use imap structure to map:

  inode number => inode location on disk

THE UNIVERSITY of ADELAIDE

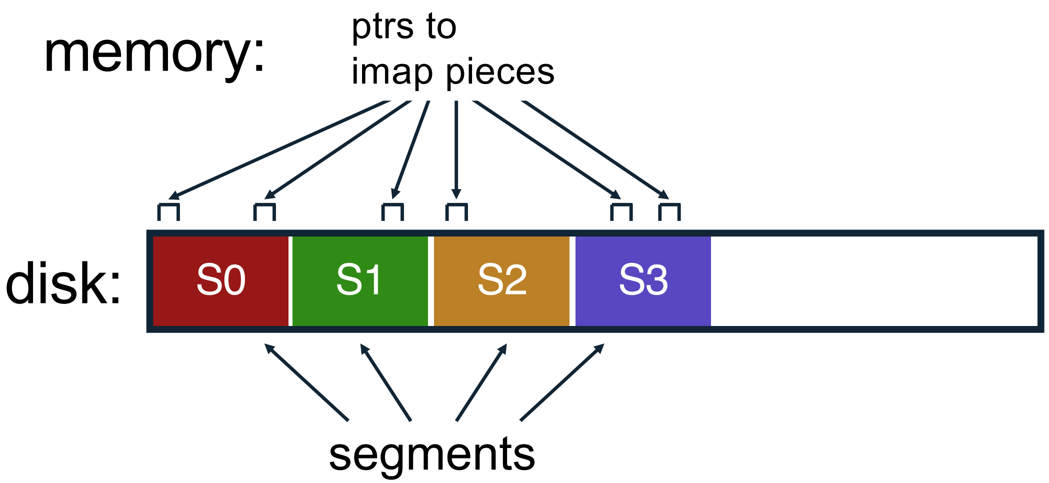
Where to keep Imap?



Where can imap be stored???? Dilemma:
1. imap too large to keep in memory
2. don't want to perform random writes for imap

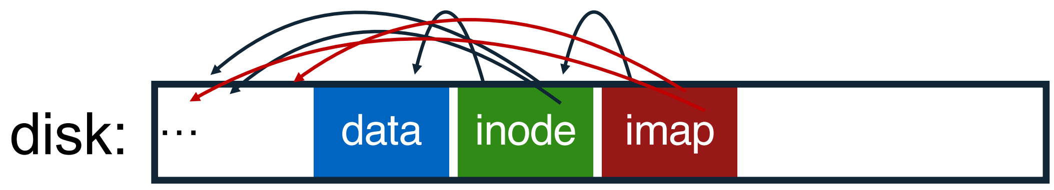
Solution:
Write imap in segments
Keep pointers to pieces of imap in memory

Solution: Imap in Segments



Solution:
Write imap in segments
Keep pointers to pieces of imap in memory
Keep recent accesses to imap cached in memory

Example Write



Solution:
Write imap in segments
Keep pointers to pieces of imap in memory
Keep recent accesses to imap cached in memory

create /foo/bar

~~data bitmap~~	~~inode bitmap~~	root inode	foo inode	bar inode	root data	foo data
		(read)				
					(read)	
			(read)			
						(read)
	~~read~~ ~~write~~					
				~~read~~ write		
			write			
						write

Most data structures same in LFS as FFS!

Use imap to find location of root and foo inodes
Update imap with new locations for foo and bar inodes

Other Issues

Crashes

Garbage Collection

Crash Recovery

What data needs to be recovered after a crash?

Need imap (lost in volatile memory)

Naive approach?

Scan entire log to reconstruct pointers to imap pieces. Slow!

Better approach?

Occasionally **checkpoint** to known on-disk location the pointers to imap pieces

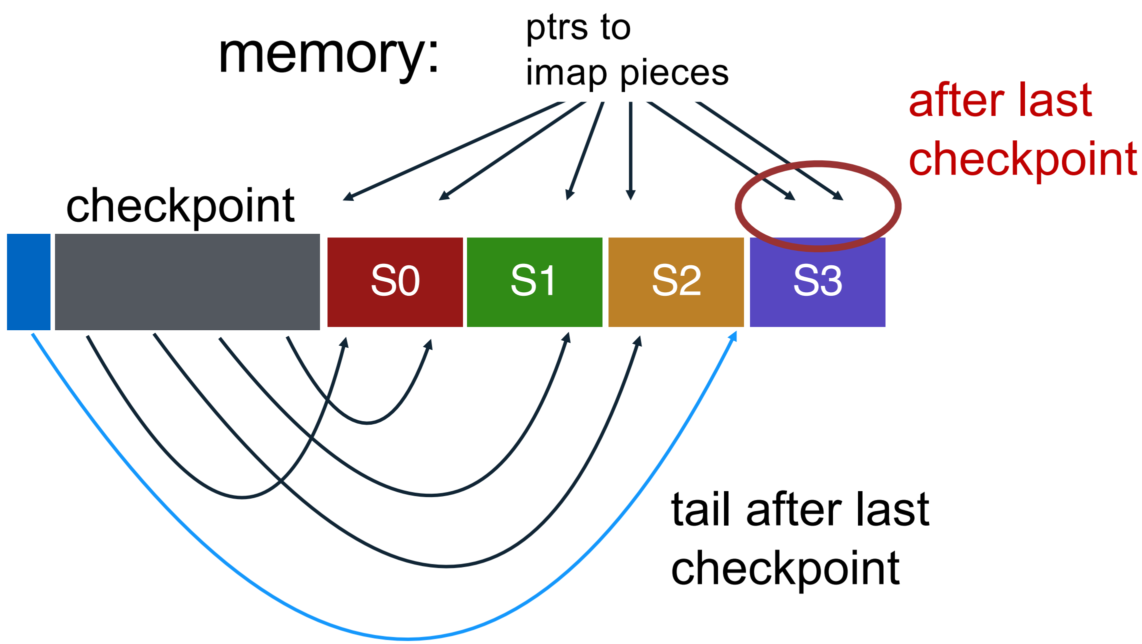
How often to checkpoint?

Checkpoint often: random I/O

Checkpoint rarely: lose more data, recovery takes longer

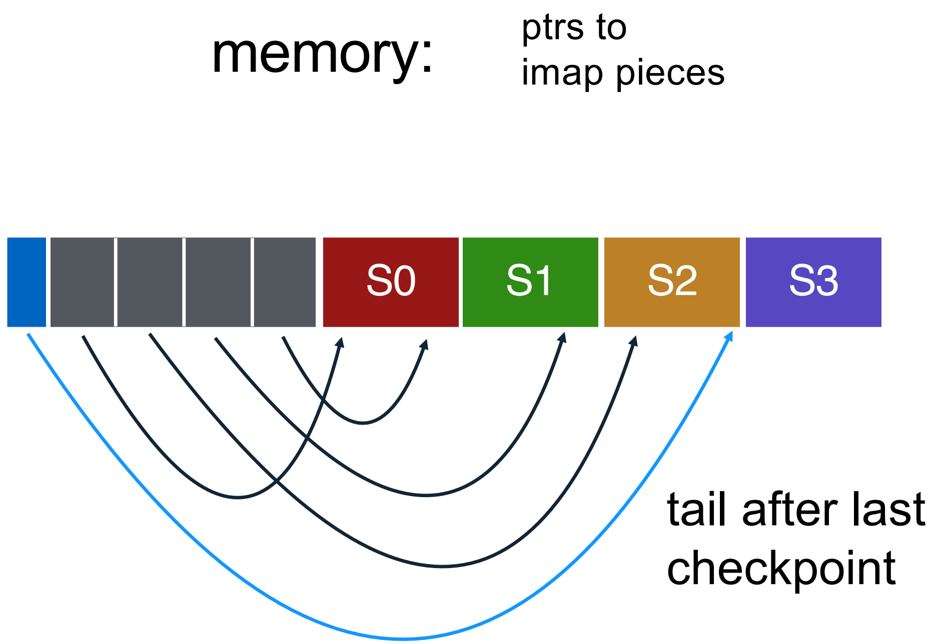
Example: checkpoint every 30 secs

Checkpoint

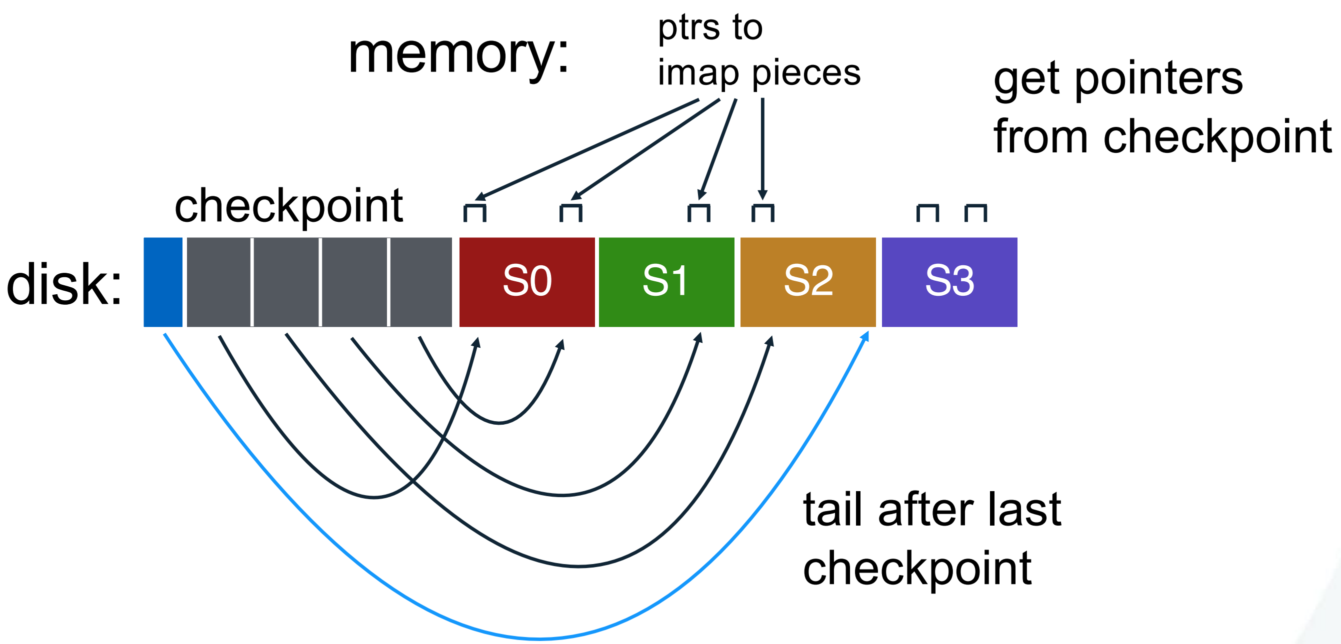


Crash!

Reboot

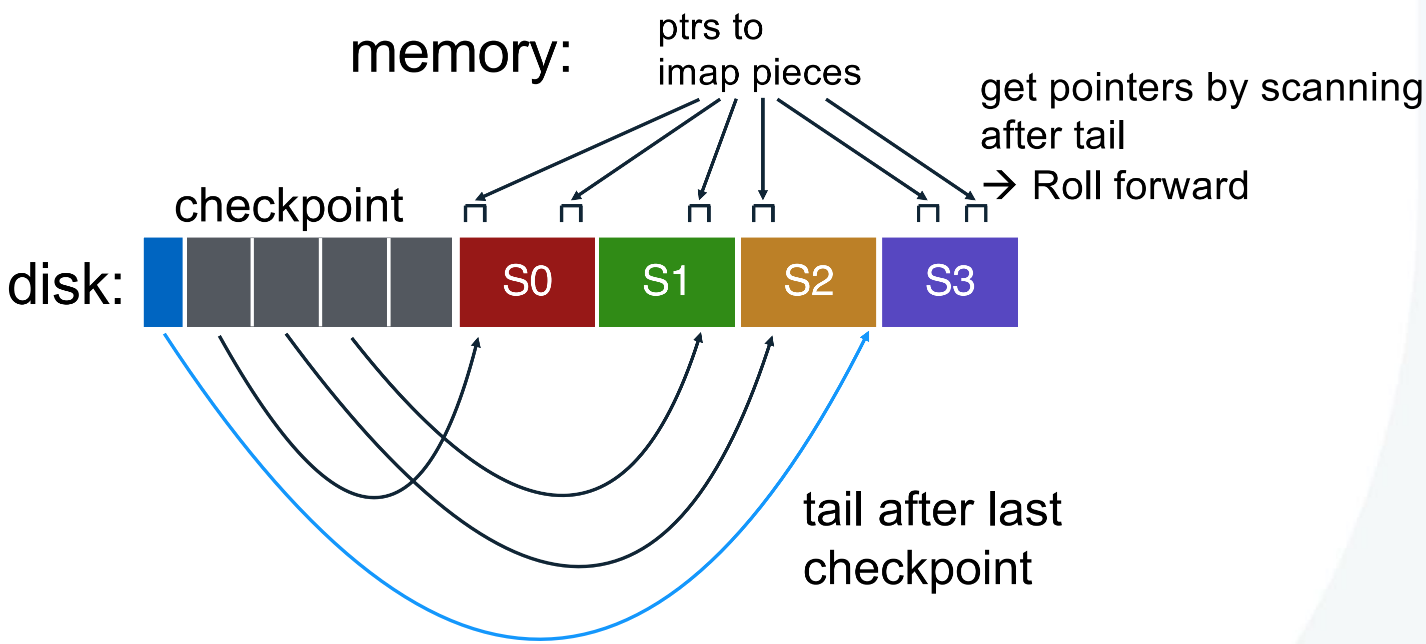

memory:
ptrs to
imap pieces
S0
S1
S2
S3
tail after last
checkpoint

Reboot



memory:
ptrs to
imap pieces
get pointers
from checkpoint
checkpoint
disk:
S0
S1
S2
S3
tail after last
checkpoint

Reboot



memory:
ptrs to
imap pieces
get pointers by scanning
after tail
→ Roll forward
checkpoint
disk:
S0
S1
S2
S3
tail after last
checkpoint

Checkpoint Summary

Checkpoint occasionally (e.g., every 30s)

Upon recovery:

- read checkpoint to find most imap pointers and segment tail
- find rest of imap pointers by reading past tail

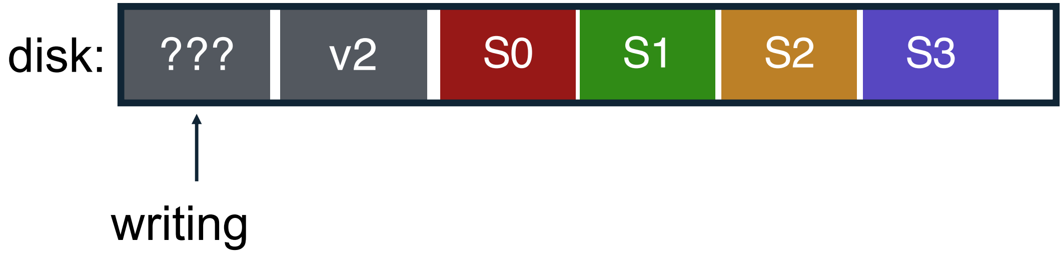
What if crash <u>during</u> checkpoint?

Checkpoint Strategy

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint



Checkpoint Strategy

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

disk:

Checkpoint Strategy

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

Checkpoint Strategy

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

disk:

Checkpoint Strategy

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

disk:

writing

Checkpoint Strategy

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint

disk: 

Other Issues

~~Crashes~~

Garbage Collection

What to do with old data?

Old versions of files -> garbage

Approach 1: garbage is a feature!

Keep old versions in case user wants to revert files later

Versioning file systems

Example: Dropbox

Approach 2: garbage collection…

Garbage Collection

Need to reclaim space:

1. When no more references (any file system)

2. After newer copy is created (COW file system)

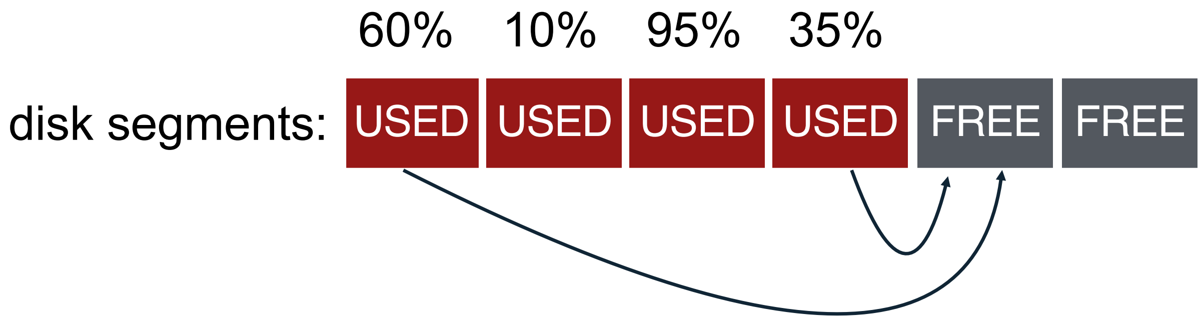
LFS reclaims segments (not individual inodes and data blocks)

- Want future overwites to be to sequential areas
- Tricky, since segments are usually partly valid

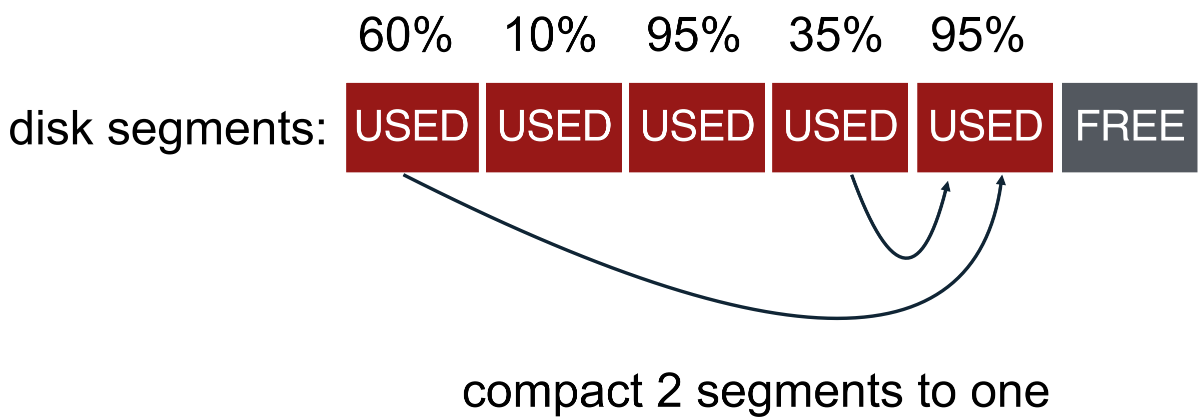
Garbage Collection

	60%	10%	95%	35%		
disk segments:	USED	USED	USED	USED	FREE	FREE

Garbage Collection

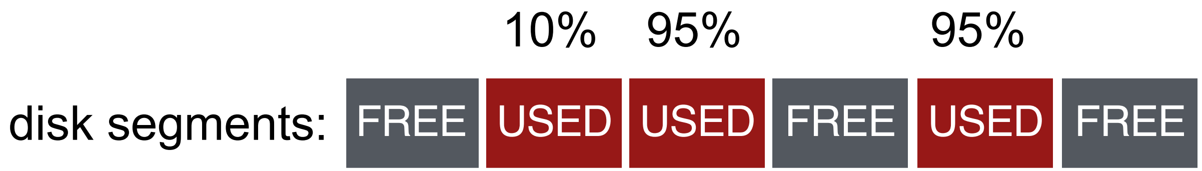


Garbage Collection



When move data blocks, copy new inode to point to it
When move inode, update imap to point to it

Garbage Collection



release input segments

Garbage Collection

General operation:

Pick M segments, compact into N (where N < M).

Mechanism:

How does LFS know whether data in segments is valid?

Policy:

Which segments to compact?

THE UNIVERSITY of ADELAIDE

Garbage Collection Mechanism

Is an inode the latest version?

- Check imap to see if this inode is pointed to
- Fast!

Is a data block the latest version?

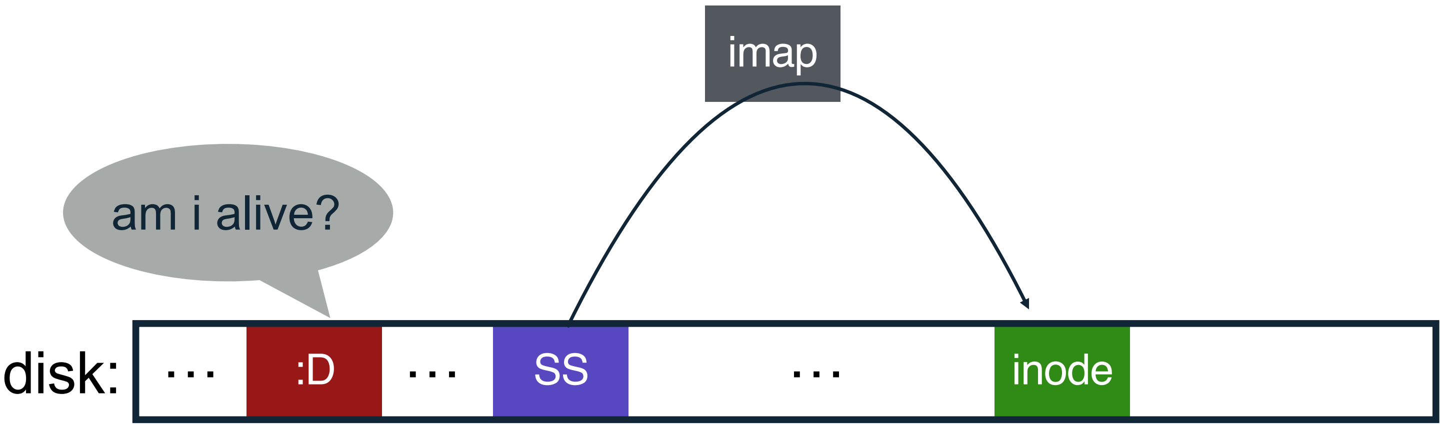
- Scan ALL inodes to see if any point to this data
- Very slow!

How to track information more efficiently?

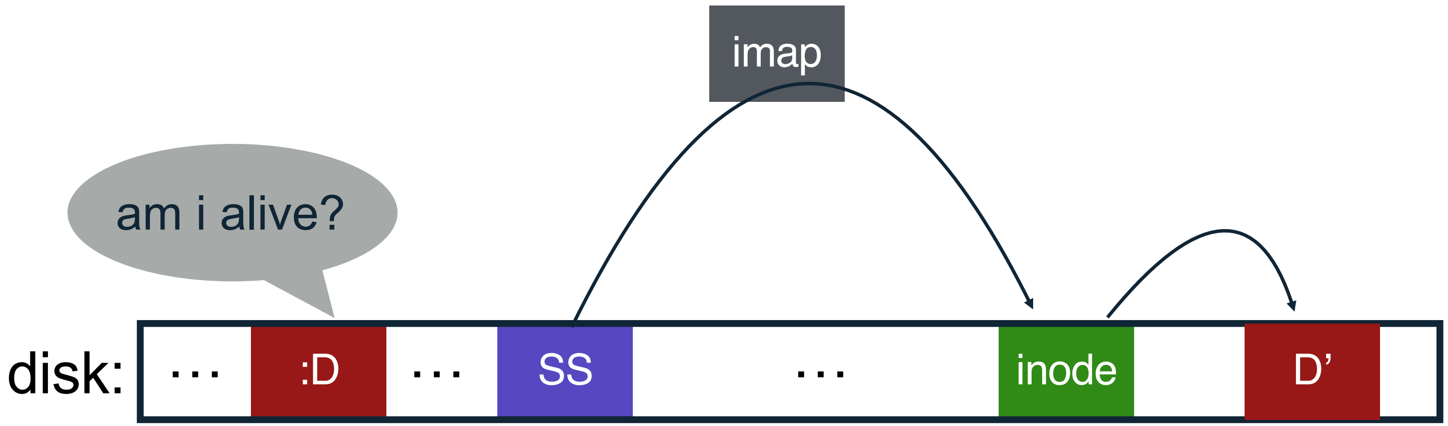
- **Segment summary** lists inode and data offset corresponding to each data block in segment (reverse pointers)

Block Liveness

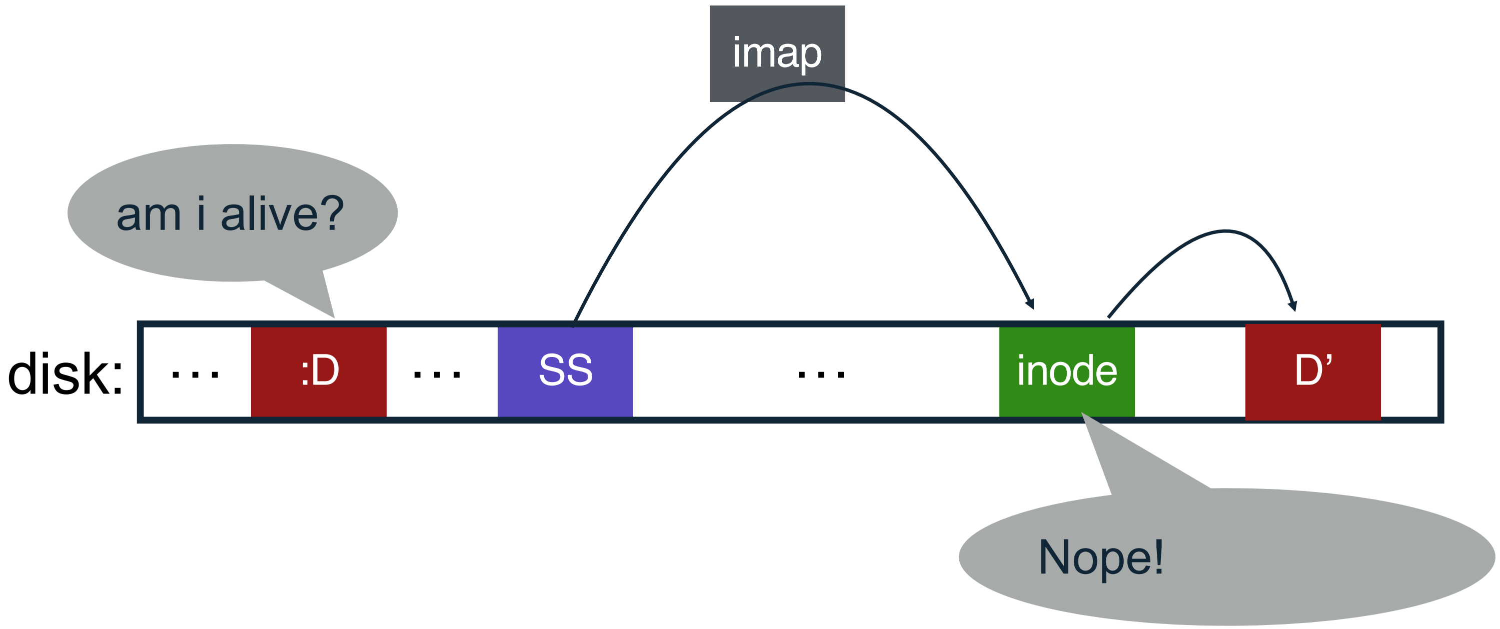
Block Liveness



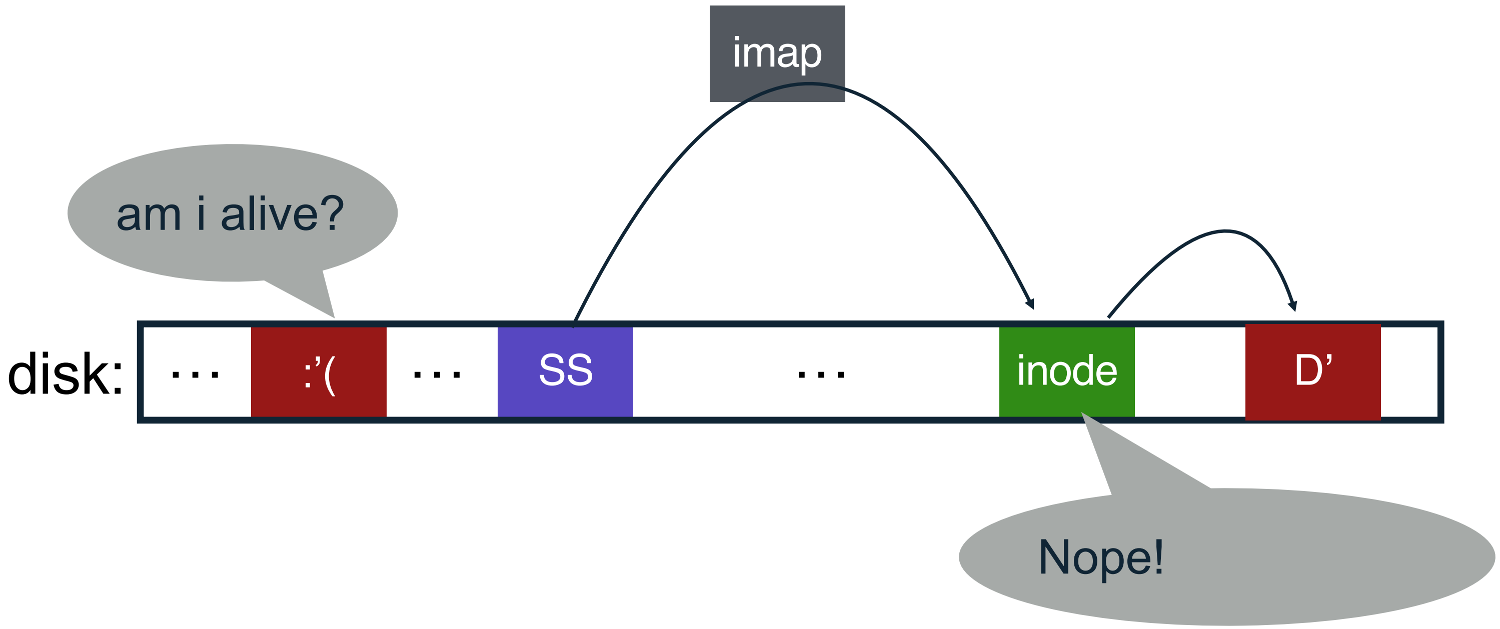
Block Liveness



Block Liveness



Block Liveness



Garbage Collection

General operation:

Pick M segments, compact into N (where N < M).

Mechanism:

How does LFS know whether data in segments is valid? [segment summary]

Policy:

Which segments to compact?

- clean most empty first
- clean coldest (ones undergoing least change)
- more complex heuristics…

THE UNIVERSITY of ADELAIDE

Conclusion

Journaling:
Put final location of data wherever file system chooses (usually in a place optimized for future reads)

LFS:
Puts data where it's fastest to write
(assume future reads cached in memory)

Other COW file systems: WAFL, ZFS, btrfs