龙星计划课程: 文件系统和分布式数据管理系统

Building File Systems and Distributed Data Management Systems for Performance and Reliability

Song Jiang

Department of Electrical and Computer Engineering
Wayne State University

What is the course about?

- Learn how a data center is built to provide Internet-wide scalable and secure services.
  - We are in the big data era and most Internet services rely on large volume of data.
  - The stack of a data center includes:
    - ✓ Computing platform (individual servers and their local file system)
    - ✓ Distributed system infrastructure, such as GFS, BigTable, MapReduce, and Pregel.
    - ✓ Services, such as search, advertising, email, maps, video, chat, blogger.
  - We will study how the distributed system infrastructure is built from bottom to the top.
- Get an insider's view with case studies
  - Look at design and implementation of real-life systems, mostly Google's.
- Have a taste of what the research in the CS/CE area looks like
  - Cultivate your curiosity

What is the course not about?

- This is not a tutorial about Ext3/4/BtrFS, GFS/HDFS, BigTtable/Hbase.
- We do not cover comprehensively all aspects of a file system and distributed systems.
- We do not follow every details of specific systems.

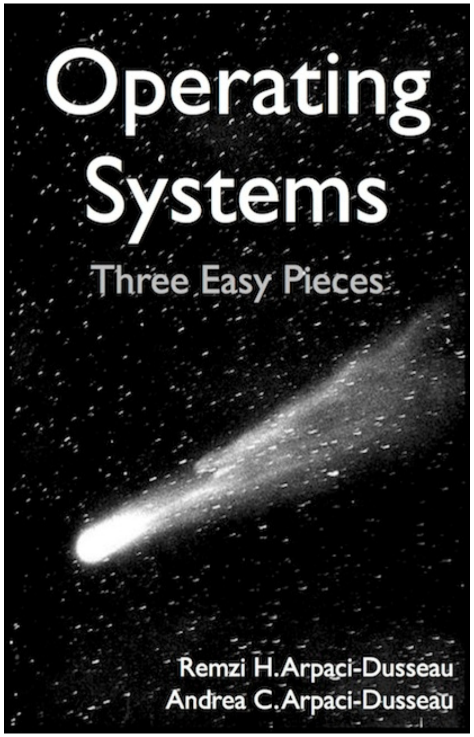
Instead, this course will focus on understanding the issues, design choices, and problem solving skills.

Why you should take the course?

- Data center is the most critical IT infrastructure of the society.
- Our digital life depends on data centers,
  - almost all Internet-based services, including cloud computing.
- The course will cover from fundamental system concepts to techniques enabling very-large-large systems.
  - Issues in a file/storage system: data replication and consistency, failure management, system reliability, scalability, availability, and efficiency.
- Research experience will go a long way for your career development.
  - Many people who program with Internet don't understand how things happen within a data center.
  - Students would be inspired to keep learning and to contribute.

Course Outline

1. File Systems
   - ❑ Files and directories
   - ❑ File system implementation
   - ❑ FSCK and journaling
   - ❑ Log-structured file system (LFS)
   - ❑ Data integrity and protection



2. Distributed File Systems and Others

3. Key-Value Data Management Systems

龙星计划课程: 文件系统和分布式数据管理系统

Building File Systems and Distributed Data Management Systems for Performance and Reliability

Lecture 1: File Systems

File-System Abstraction

What is a File?

Array of bytes.

Ranges of bytes can be read/written.

File system consists of many files.

What is a File?

Array of bytes.

Ranges of bytes can be read/written.

File system consists of many files.

Files need names so programs can choose the right one.

File Names

Three types of names:

- inode
- path
- file descriptor

File Names

Three types of names:
- inode
- path
- file descriptor

Inodes

Each file has exactly one inode number.

Inodes are unique (at a given time) within a FS.

Different file system may use the same number, numbers may be recycled after deletes.

Inodes

Each file has exactly one inode number.

Inodes are unique (at a given time) within a FS.

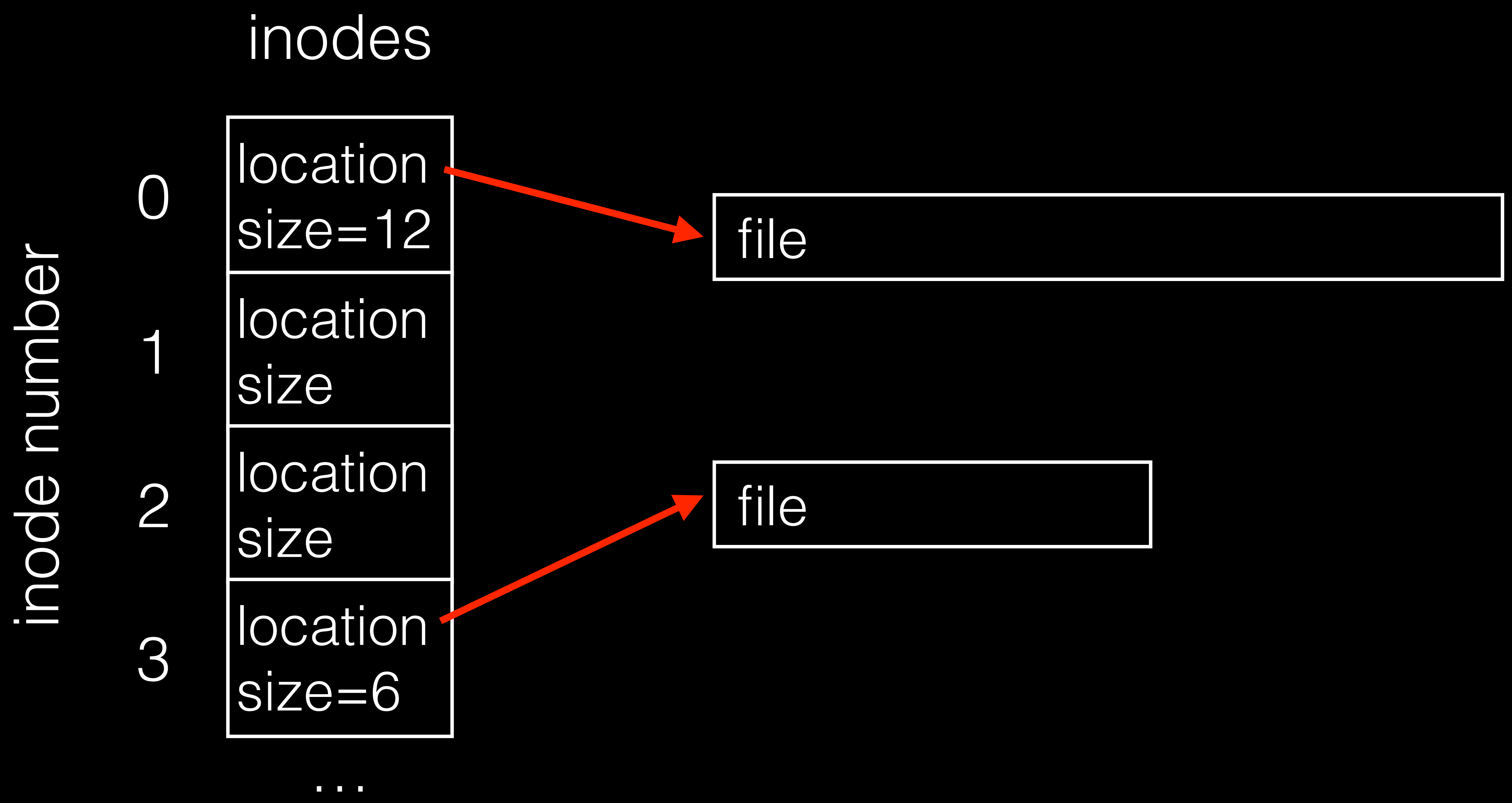
Different file system may use the same number, numbers may be recycled after deletes.

Show inodes via `stat`*.*

What does “i” stand for?

“In truth, I don't know either. It was just a term that we started to use. ‘Index’ is my best guess, because of the slightly unusual file system structure that stored the access information of files as a flat array on the disk…”

~ Dennis Ritchie


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
...
file
file

File API (attempt 1)

```
read(int inode, void *buf, size_t nbyte)

write(int inode, void *buf, size_t nbyte)

seek(int inode, off_t offset)
```

File API (attempt 1)

```
read(int inode, void *buf, size_t nbyte)

write(int inode, void *buf, size_t nbyte)

seek(int inode, off_t offset)
```

note: seek does not cause disk seek
unless followed by a read/write

File API (attempt 1)

```
read(int inode, void *buf, size_t nbyte)

write(int inode, void *buf, size_t nbyte)

seek(int inode, off_t offset)
```

Disadvantages?

File API (attempt 1)

```
read(int inode, void *buf, size_t nbyte)

write(int inode, void *buf, size_t nbyte)

seek(int inode, off_t offset)
```

Disadvantages?
- names hard to remember
- everybody has the same offset
- collisions (not hierarchical)

File API (attempt 1)

```
pread(int inode, void *buf,
      off_t offset, size_t nbyte)
pwrite(int inode, void *buf,
       off_t offset size_t nbyte)
seek(int inode, off_t offset)
```

~~seek(int inode, off_t offset)~~

Disadvantages?

- names hard to remember
- ~~everybody has the same offset~~
- collisions (not hierarchical)

File Names

Three types of names:

- inode
- path
- file descriptor

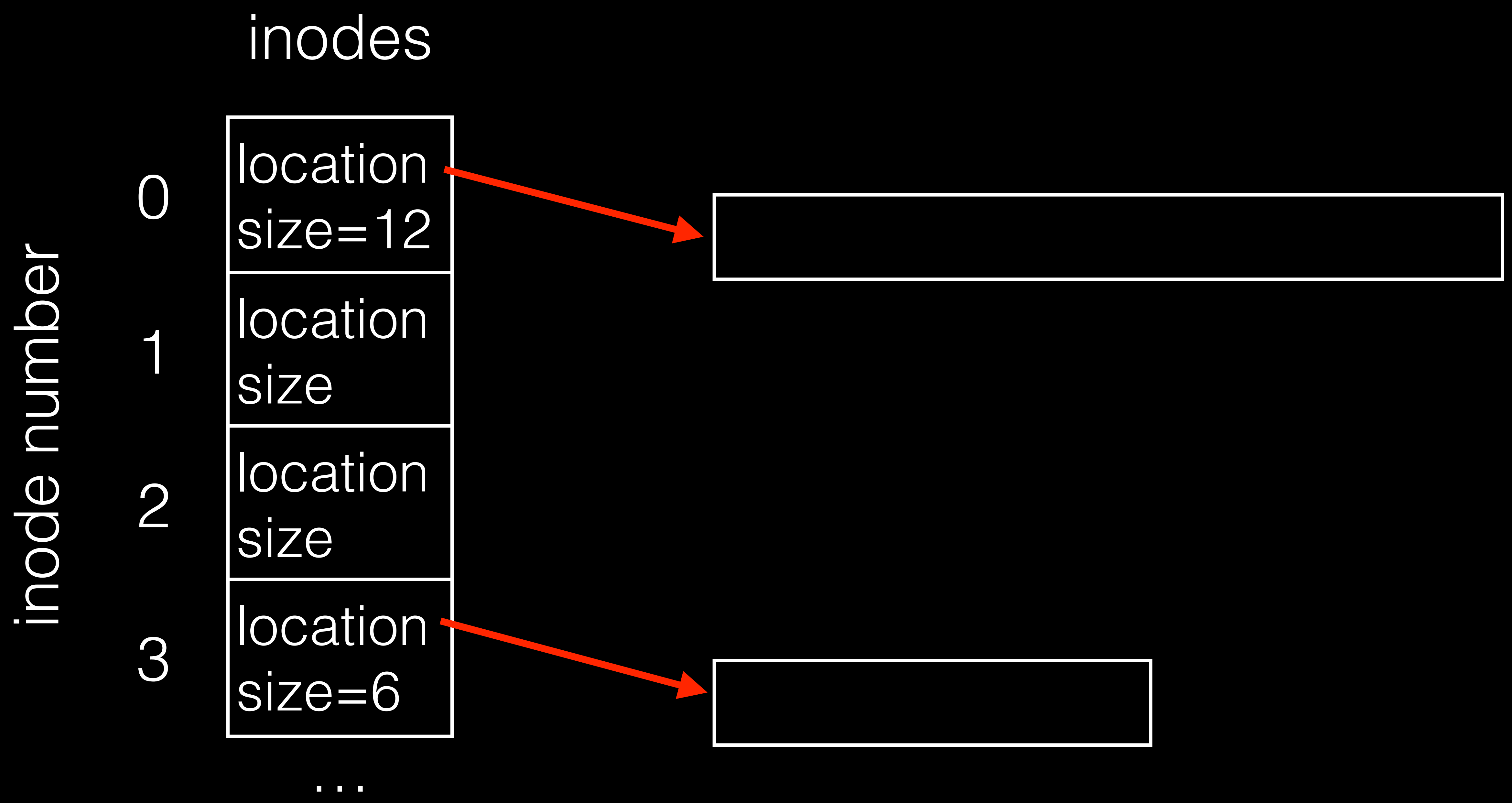
Paths

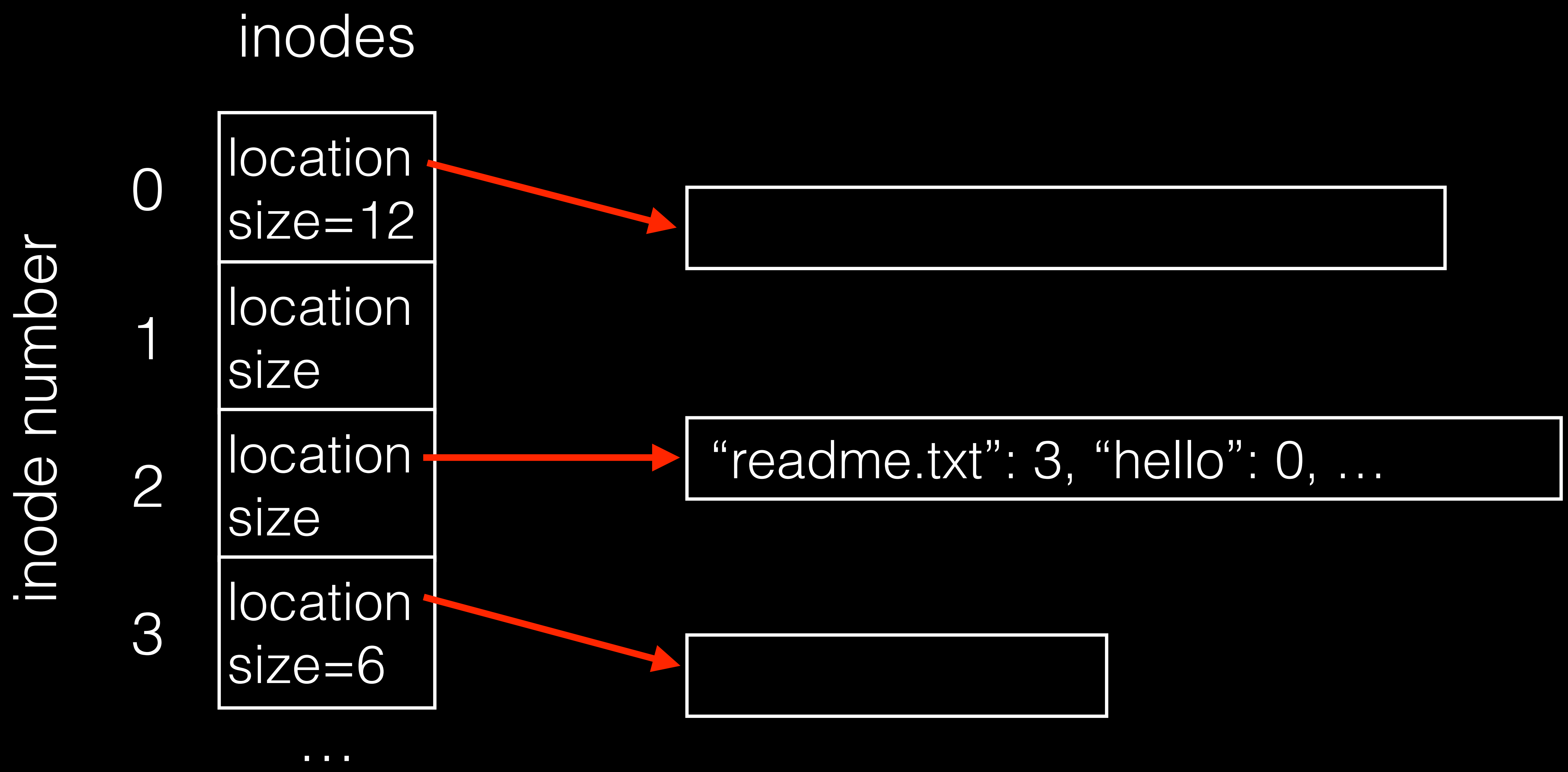
String names are friendlier than number names.

Paths

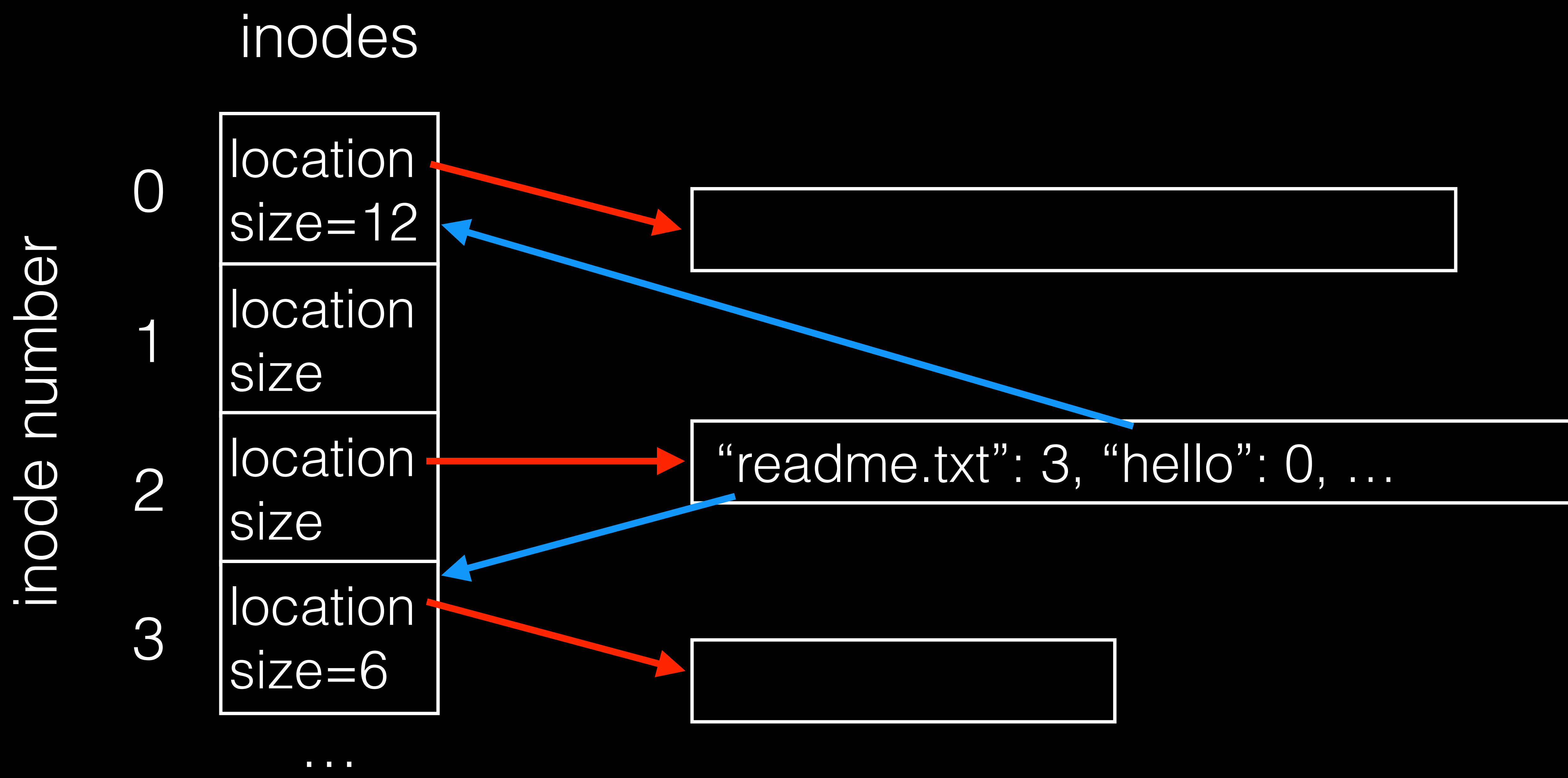
String names are friendlier than number names.

Store *path-to-inode* mappings in a predetermined "root" file (typically inode 2)


inodes
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
...



inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“readme.txt”: 3, “hello”: 0, …


inodes
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
…
“readme.txt”: 3, “hello”: 0, …

Paths

String names are friendlier than number names.

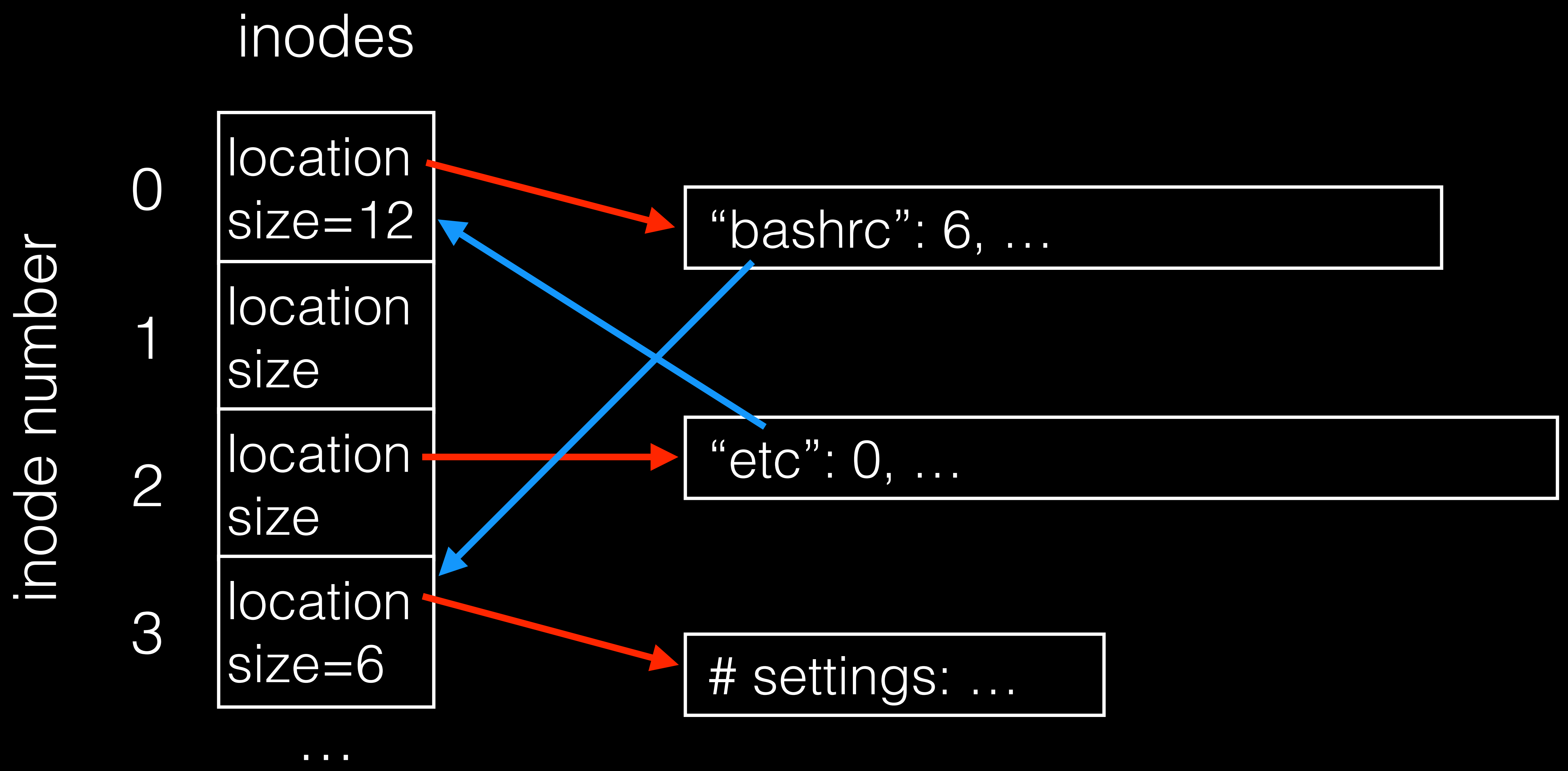
Store *path-to-inode* mappings in a predetermined "root" file (typically inode 2)

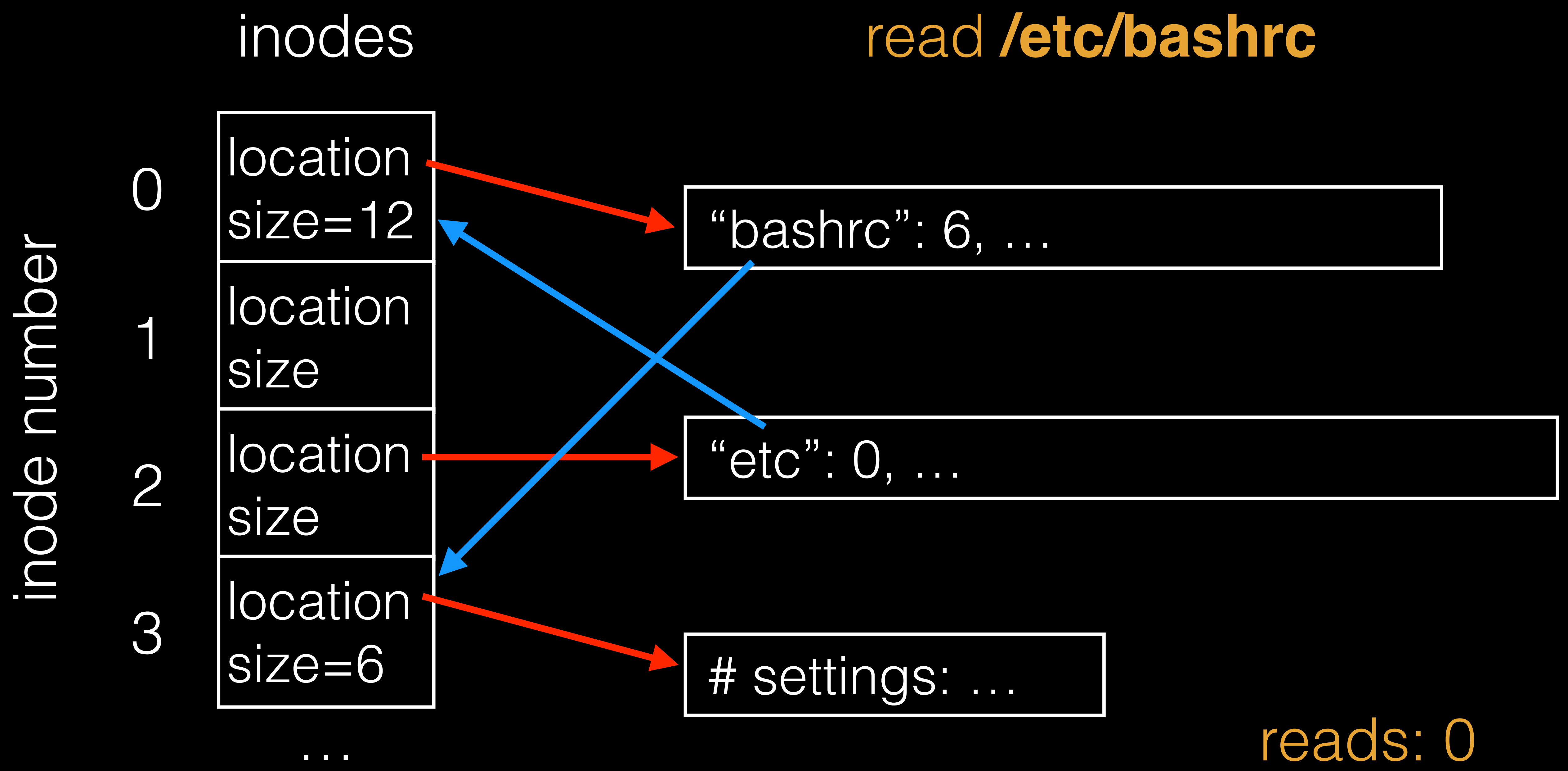
Paths

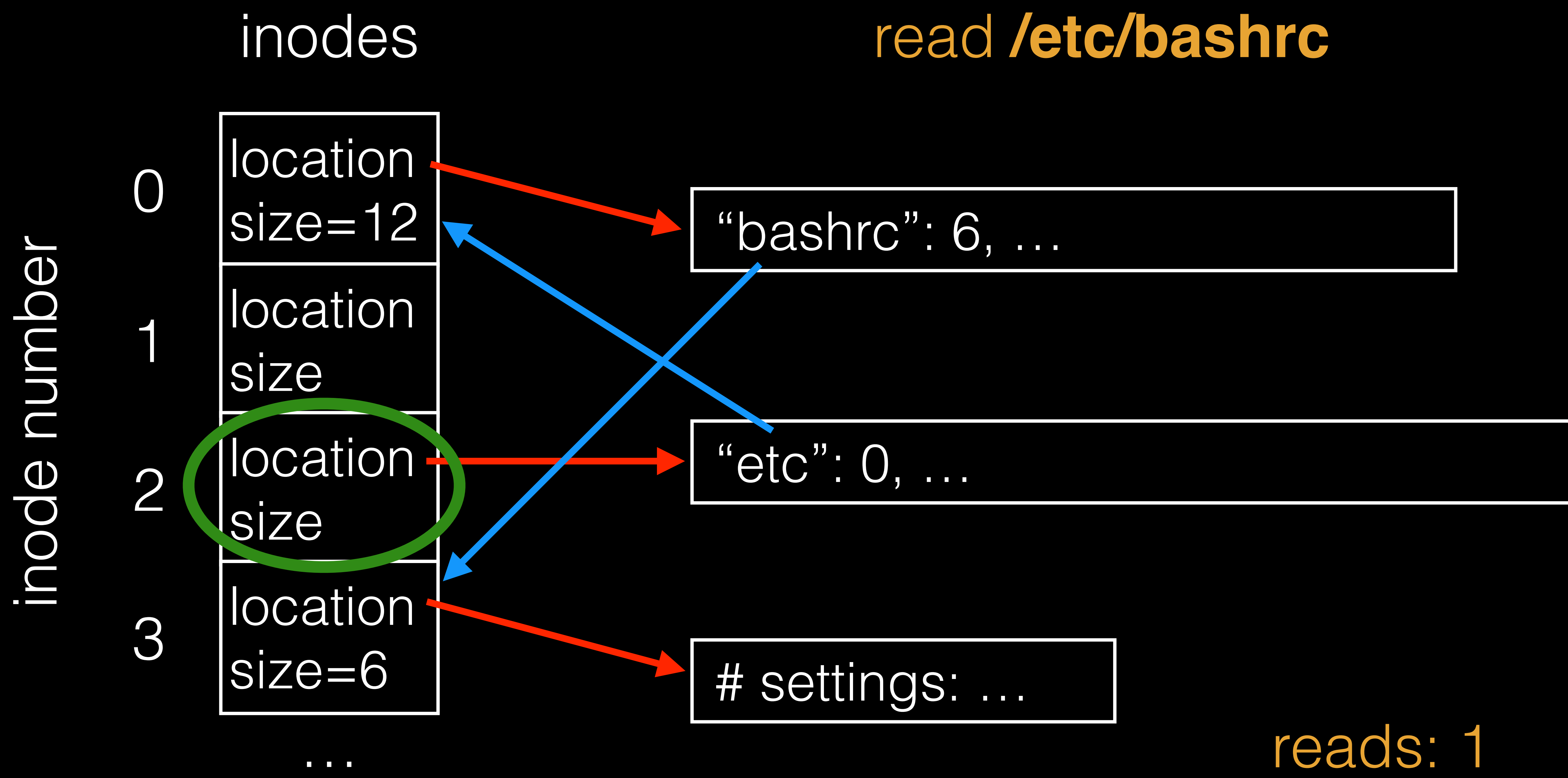
String names are friendlier than number names.

Store *path-to-inode* mappings in a predetermined "root" file (typically inode 2)

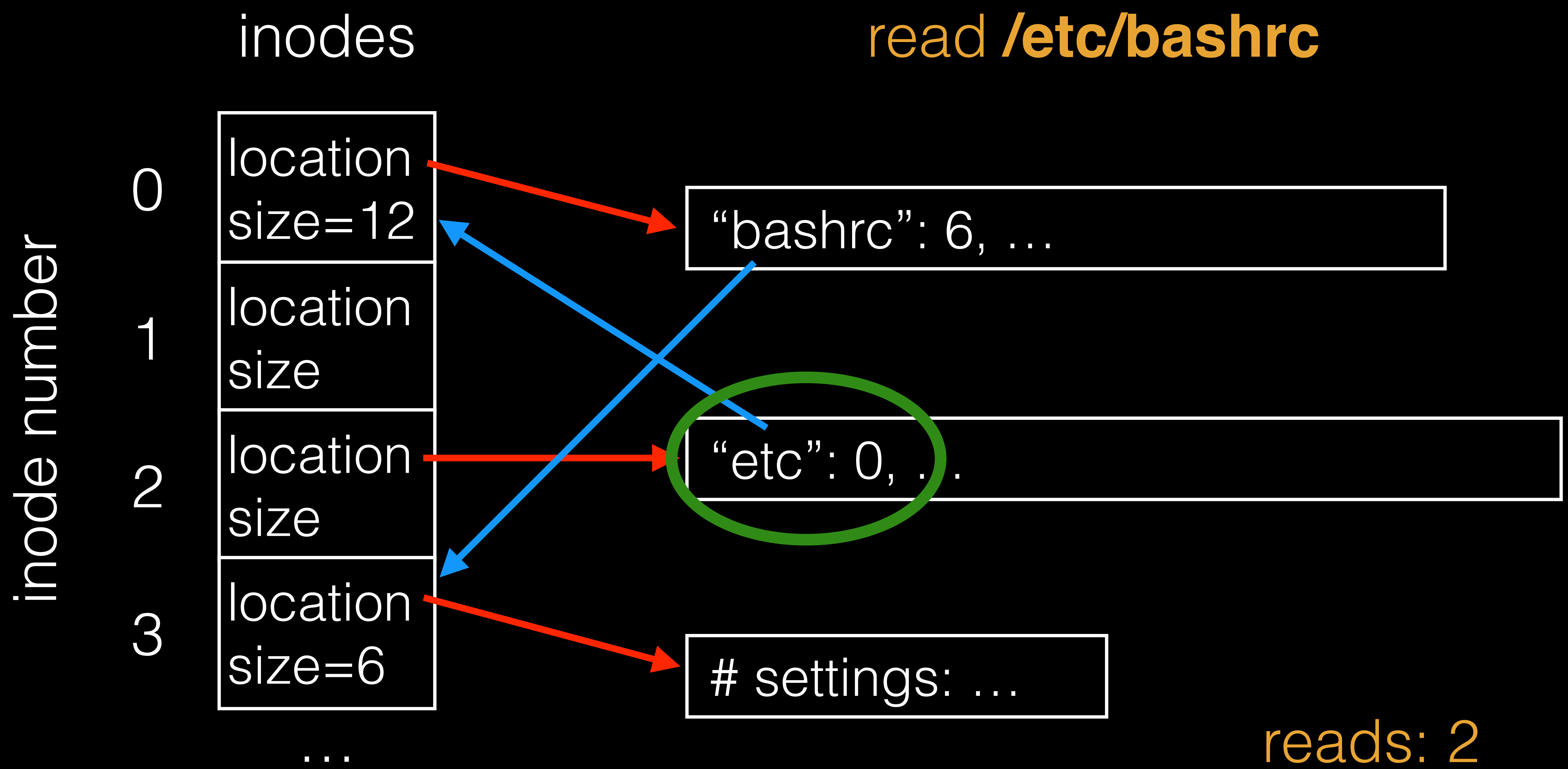
Generalize! Store path-to-inode mapping in many files. Call these special files directories.

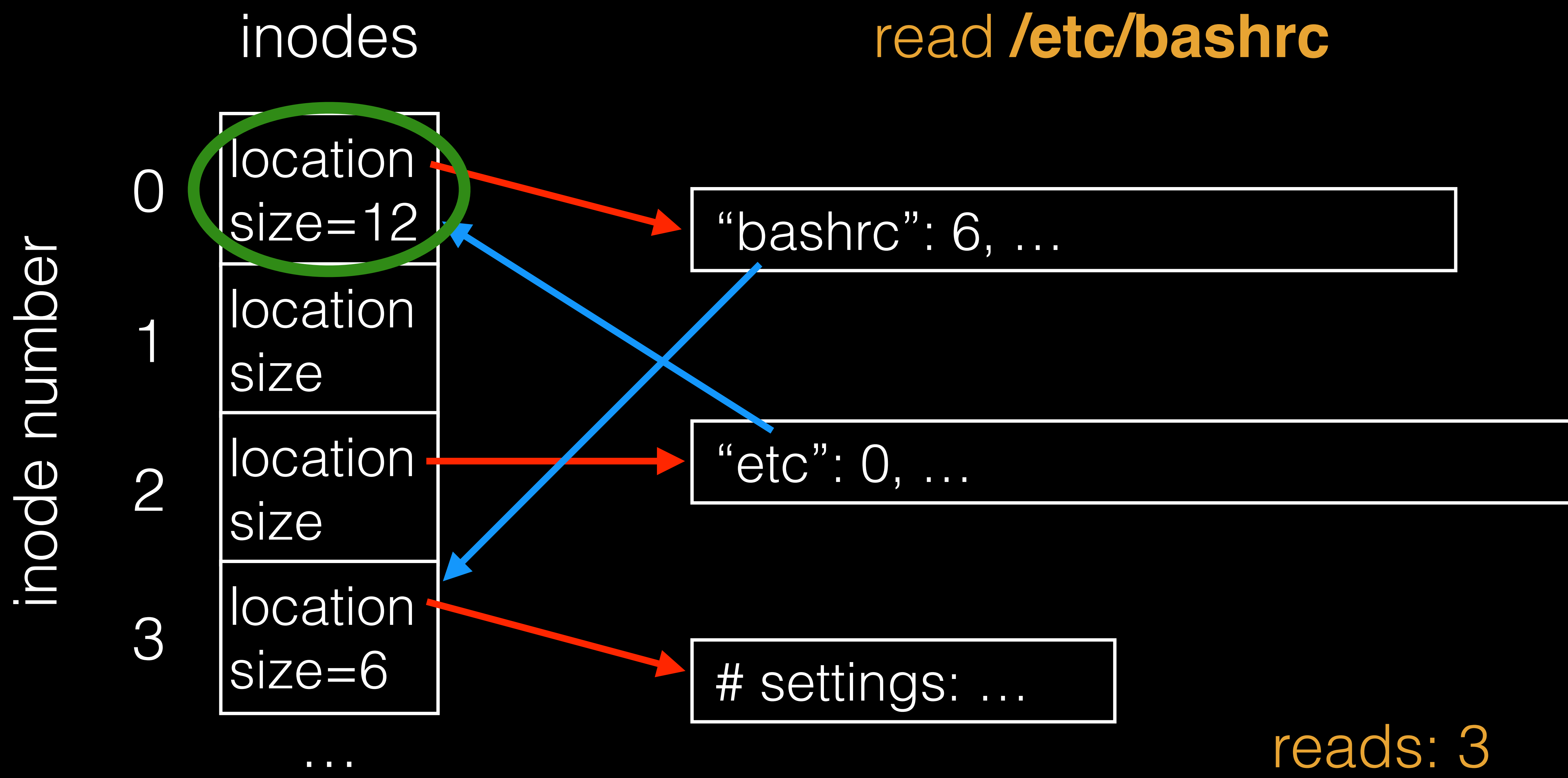

inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“bashrc”: 6, …
“etc”: 0, …
settings: …

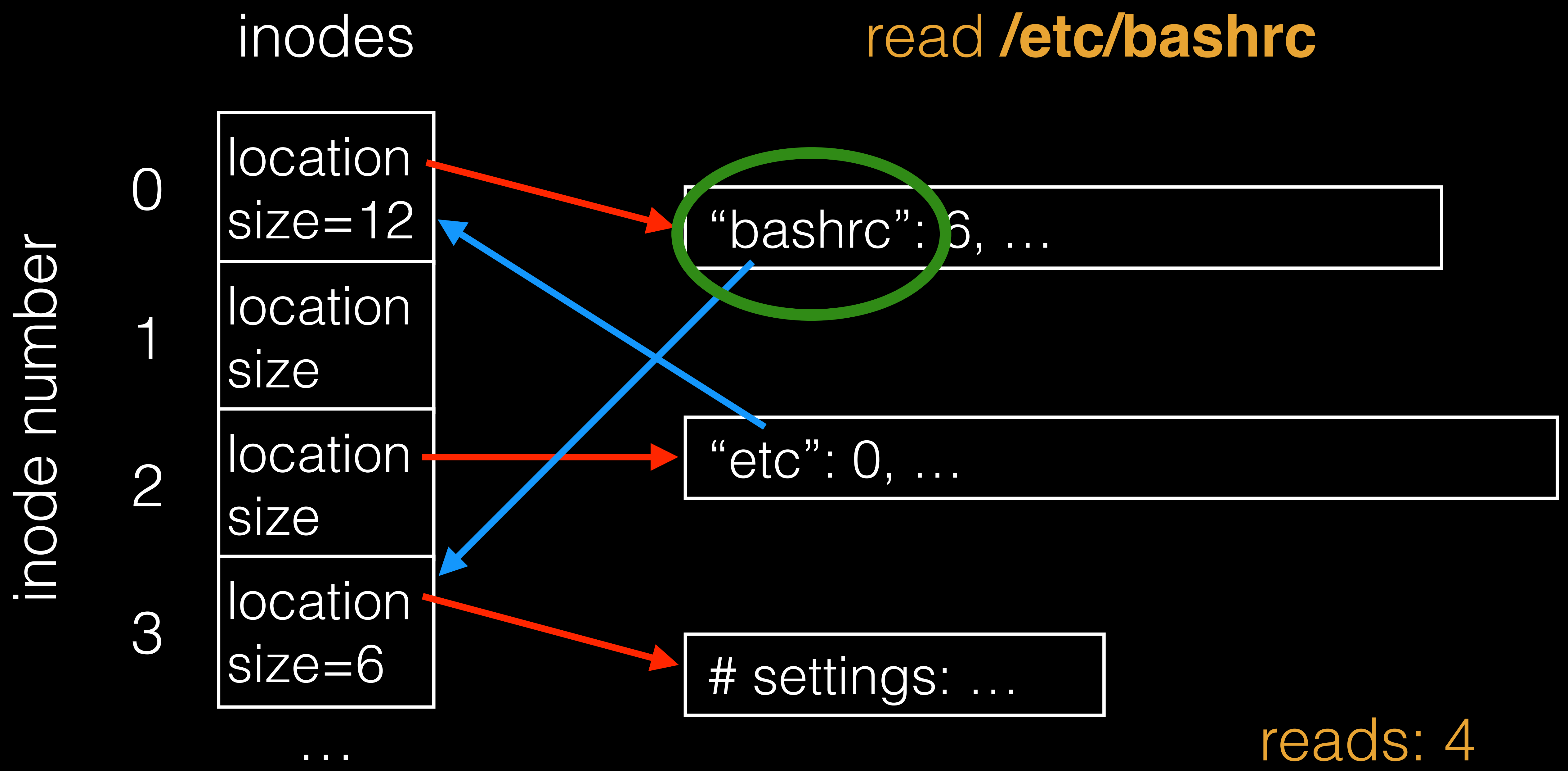

inodes
read /etc/bashrc
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
"bashrc": 6, …
"etc": 0, …
settings: …
reads: 0



inodes
read /etc/bashrc
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
"bashrc": 6, …
"etc": 0, …
settings: …
reads: 1


inodes
read /etc/bashrc
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
"bashrc": 6, …
"etc": 0, …
settings: …
reads: 2

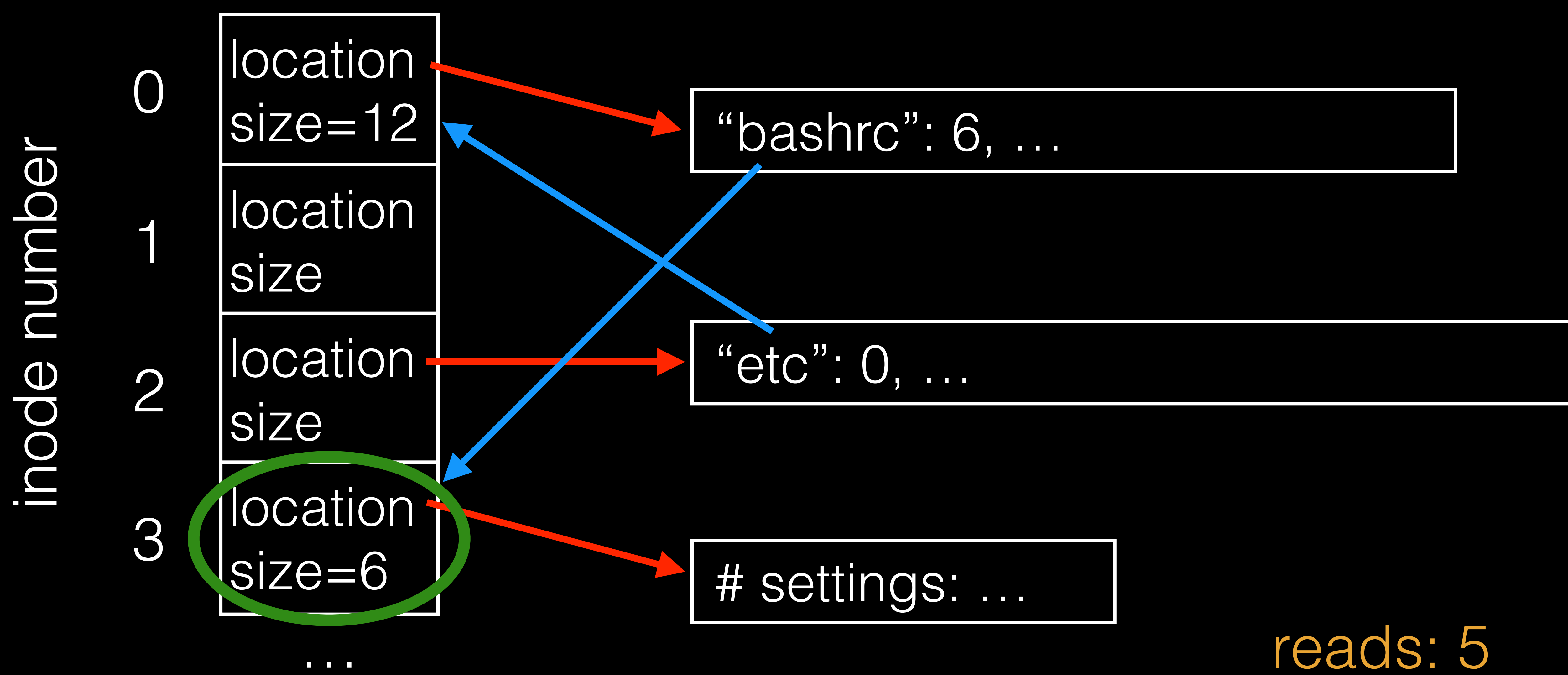

inodes
read /etc/bashrc
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
…
"bashrc": 6, …
"etc": 0, …
settings: …
reads: 3



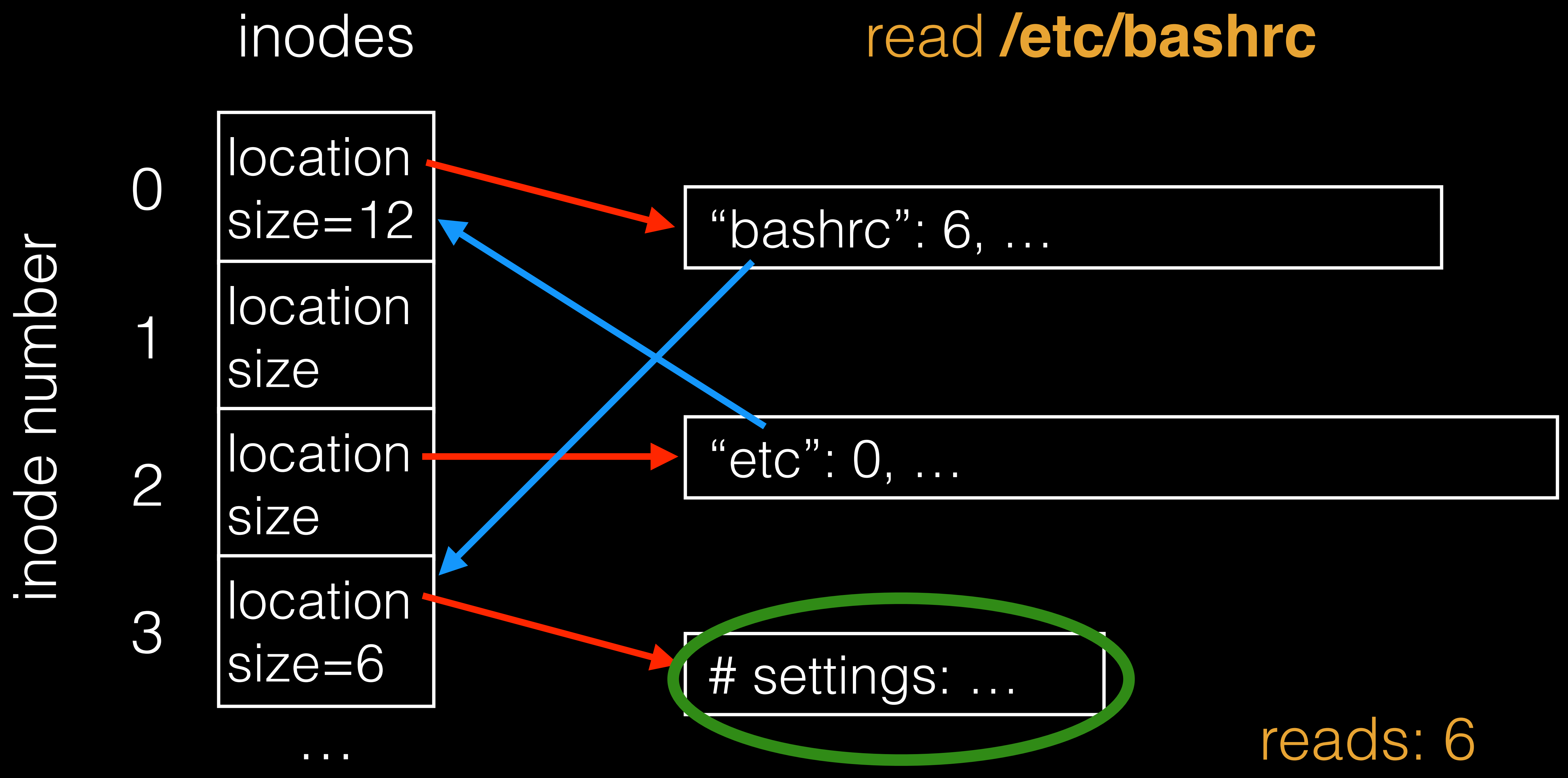
inodes
read /etc/bashrc
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“bashrc”: 6, …
“etc”: 0, …
settings: …
reads: 4

read **/etc/bashrc**

inodes



reads: 5


inodes
read /etc/bashrc
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
"bashrc": 6, …
"etc": 0, …
settings: …
reads: 6

Paths

String names are friendlier than number names.

Store *path-to-inode* mappings in a predetermined "root" file (typically inode 2)

Generalize! Store path-to-inode mapping in many files. Call these special files directories.

Paths

String names are friendlier than number names.

Store *path-to-inode* mappings in a predetermined "root" file (typically inode 2)

Generalize! Store path-to-inode mapping in many files. Call these special files directories.

Reads for getting final inode called "traversal".

Directory Calls

mkdir: create new directory

readdir: read/parse directory entries

Why no writedir?

Special Directory Entries

```
Tylers-MacBook-Pro:scratch trh$ ls -la
total 728
drwxr-xr-x  34 trh  staff    1156 Oct 19 11:41 .
drwxr-xr-x+ 59 trh  staff    2006 Oct  8 15:49 ..
-rw-r--r--@  1 trh  staff    6148 Oct 19 11:42 .DS_Store
-rw-r--r--   1 trh  staff     553 Oct  2 14:29 asdf.txt
-rw-r--r--   1 trh  staff     553 Oct  2 14:05 asdf.txt~
drwxr-xr-x   4 trh  staff     136 Jun 18 15:37 backup
…
```

File API (attempt 2)

```
pread(char *path, void *buf,
      off_t offset, size_t nbyte)

pwrite(char *path, void *buf,
       off_t offset size_t nbyte)
```

File API (attempt 2)

```
pread(char *path, void *buf,
      off_t offset, size_t nbyte)

pwrite(char *path, void *buf,
       off_t offset size_t nbyte)
```

Disadvantages?

File API (attempt 2)

```
pread(char *path, void *buf,
      off_t offset, size_t nbyte)

pwrite(char *path, void *buf,
       off_t offset size_t nbyte)
```

Disadvantages? Expensive traversal! Goal: traverse once.

File Names

Three types of names:

- inode
- path
- file descriptor

File Descriptor (fd)

Idea: do traversal once, and store inode in descriptor object. Do reads/writes via descriptor. Also remember offset.

A file-descriptor table contains pointers to file descriptors.

The integers you're used to using for file I/O are indexes into this table.

FD Table (xv6)

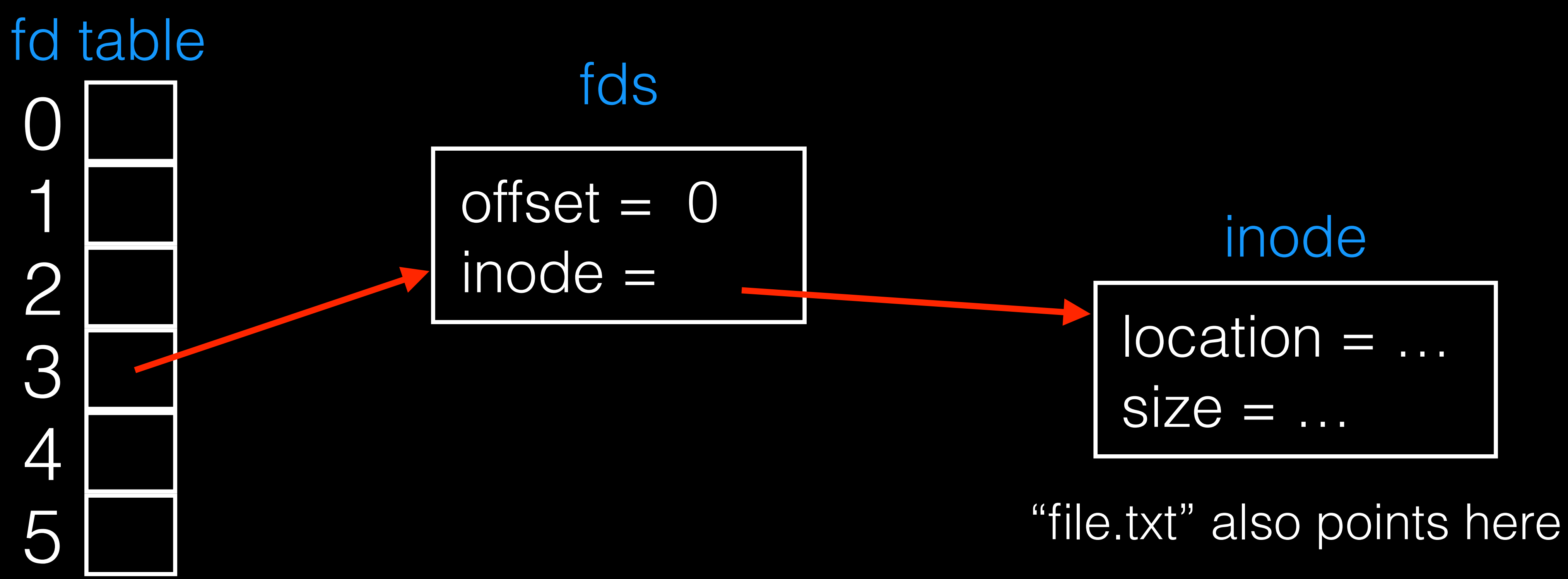
```
struct file {
  ...
  struct inode *ip;
  uint off;
};

// Per-process state
struct proc {
  ...
  struct file *ofile[NOFILE];  // Open files
  ...
}
```

Code Snippet

```
int fd1 = open(“file.txt”); // returns 3
read(fd1, buf, 12);
int fd2 = open(“file.txt”); // returns 4
int fd3 = dup(fd2);         // returns 5
```

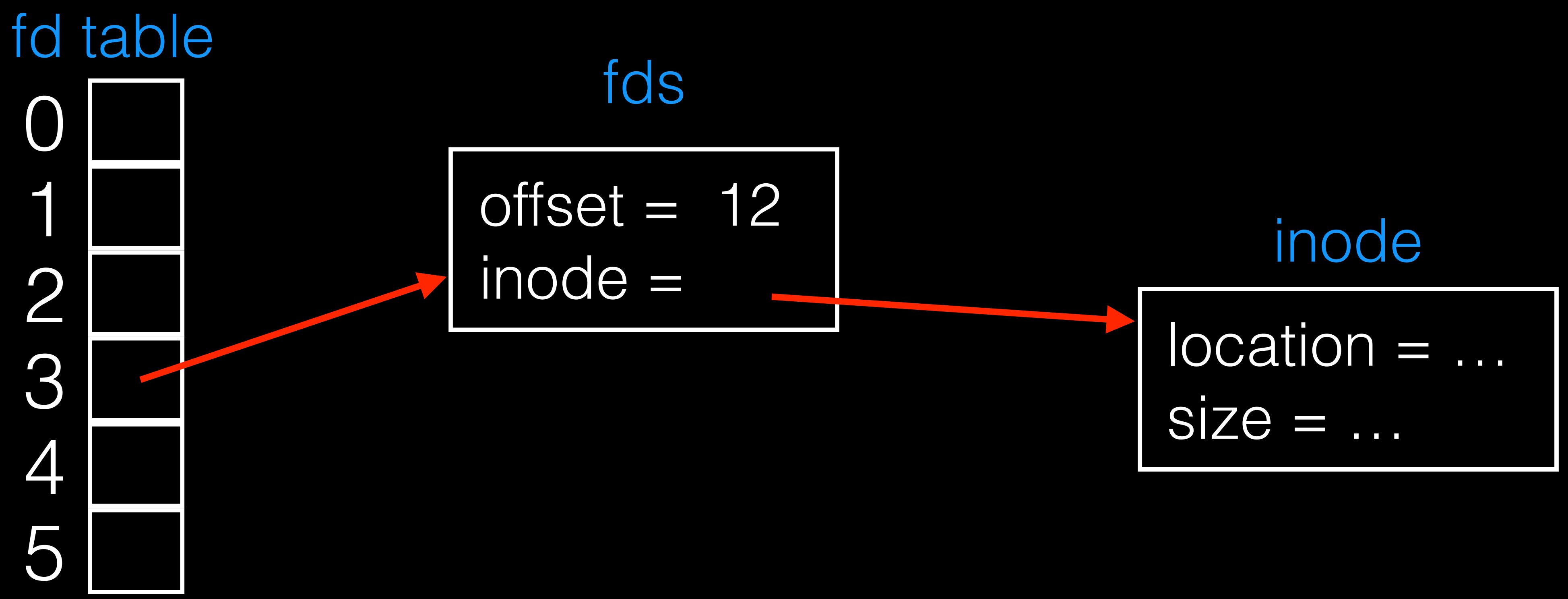
Code Snippet

```
int fd1 = open("file.txt"); // returns 3
```



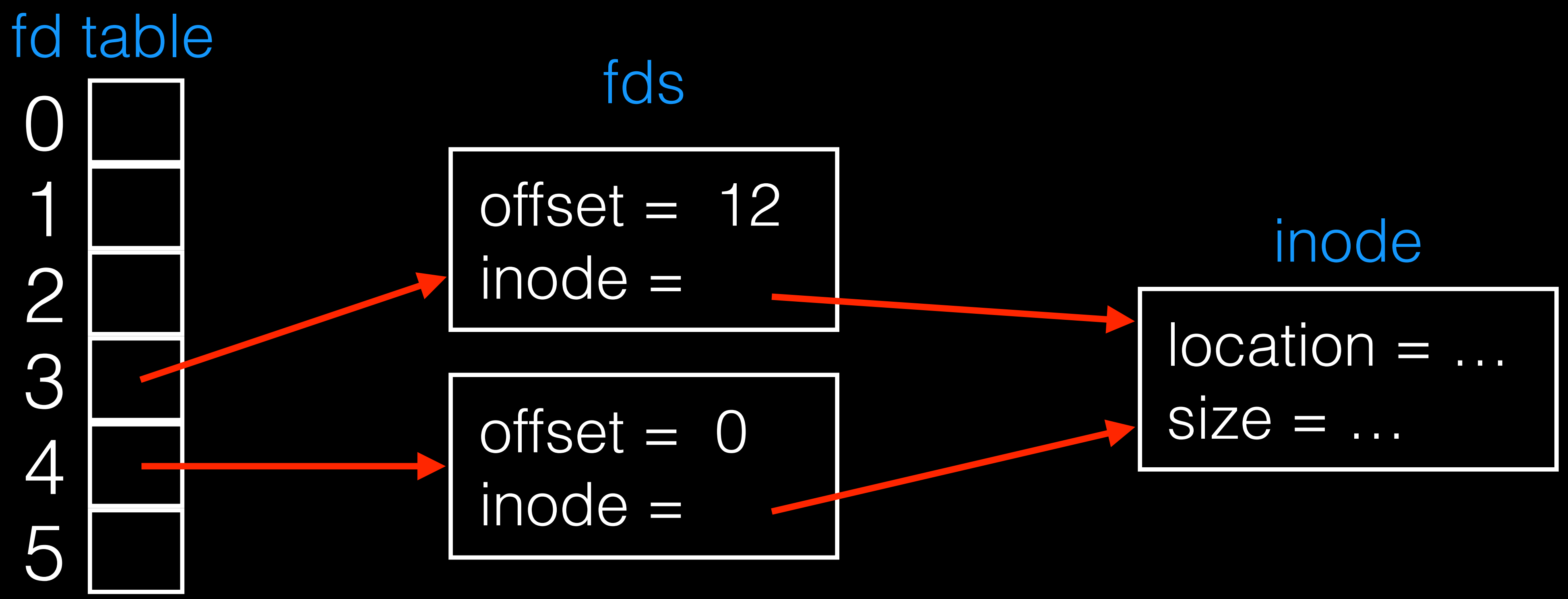
Code Snippet

```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
```



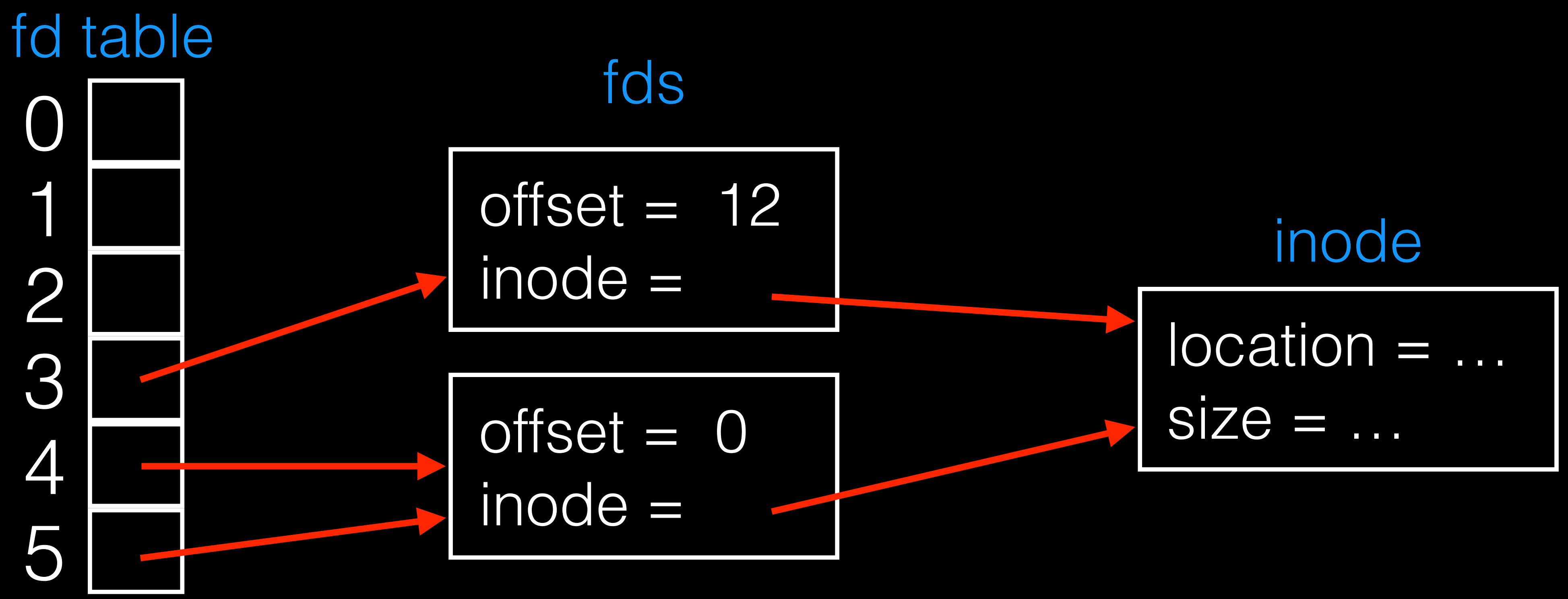
Code Snippet

```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
int fd2 = open("file.txt"); // returns 4
```



Code Snippet

```
int fd1 = open("file.txt"); // returns 3
read(fd1, buf, 12);
int fd2 = open("file.txt"); // returns 4
int fd3 = dup(fd2);         // returns 5
```



File API (attempt 3)

```
int fd = open(char *path, int flag, mode_t mode)

read(int fd, void *buf, size_t nbyte)

write(int fd, void *buf, size_t nbyte)

close(int fd)
```

File API (attempt 3)

```
int fd = open(char *path, int flag, mode_t mode)

read(int fd, void *buf, size_t nbyte)

write(int fd, void *buf, size_t nbyte)

close(int fd)
```

advantages:
- string names
- hierarchical
- traverse once
- different offsets

Deleting Files

There is no system call for deleting files!

Deleting Files

There is no system call for deleting files!

Inode (and associated file) is garbage collected when there are no references (from paths or fds).

Deleting Files

There is no system call for deleting files!

Inode (and associated file) is garbage collected when there are no references (from paths or fds).

Paths are deleted when: `unlink()` is called.

FDs are deleted when: ???

Deleting Files

There is no system call for deleting files!

Inode (and associated file) is garbage collected when there are no references (from paths or fds).

Paths are deleted when: `unlink()` is called.

FDs are deleted when: `close()`, or process quits

Deleting Directories

Directories can also be unlinked with unlink().
But only if empty!

How does “rm -rf” work?

Let’s find out with `strace`!

```c
void recursiveDelete(char* dirname) {
   char filename[FILENAME_MAX];
   DIR *dp = opendir (dirname);
   struct dirent *ep;
   while(ep = readdir (dp)) {
      snprintf(filename, FILENAME_MAX,
               "%s/%s", dirname, ep->d_name);
      if(is_dir(ep))
         recursiveDelete(filename);
      else
         unlink(filename);
   }
   unlink(dirname);
}
```

my worst bug ever

Many File Systems

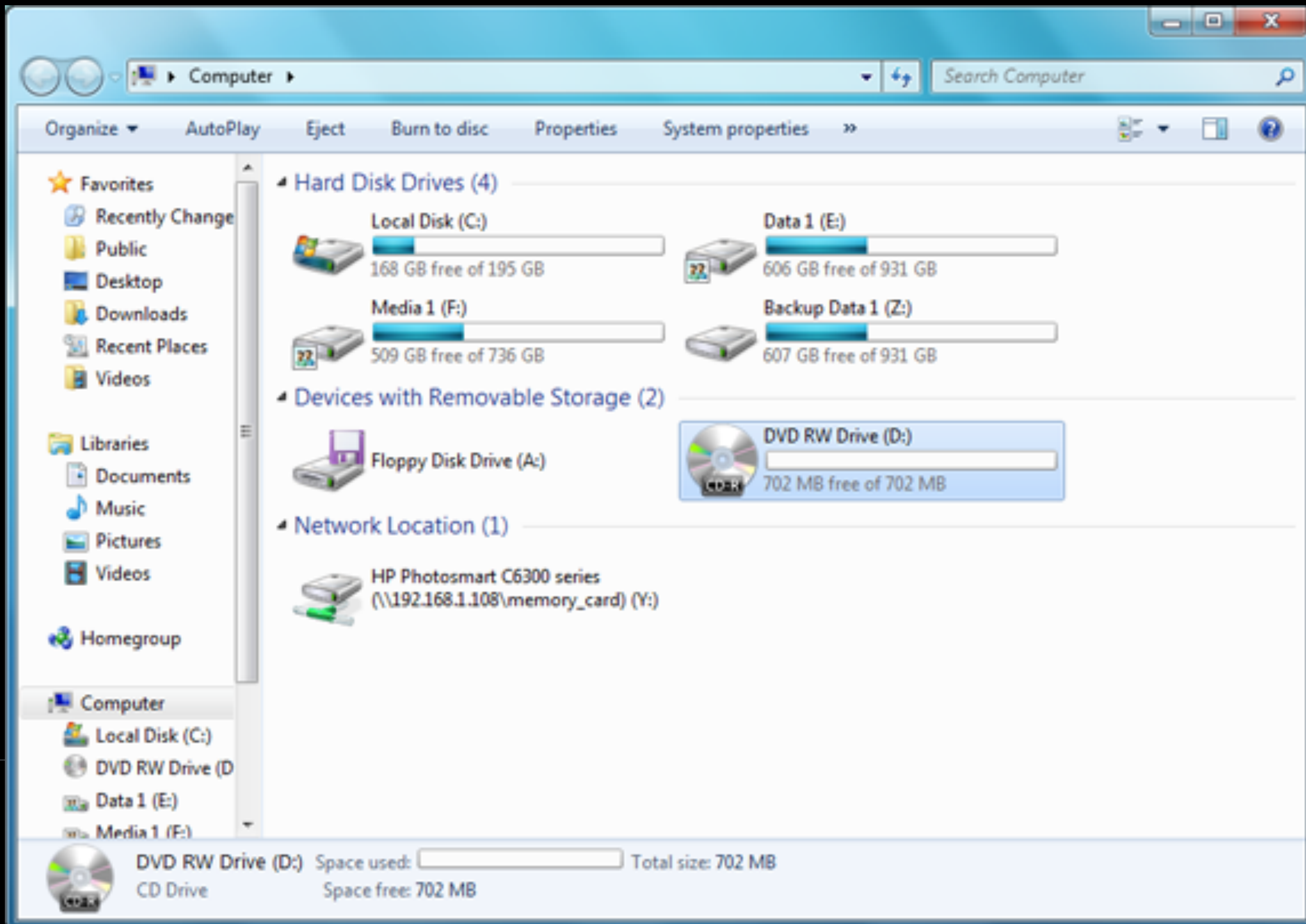
Many File Systems

Users often want to use many file systems.

For example:
- main disk
- backup disk
- AFS
- thumb drives

What is the most elegant way to support this?

Many File Systems: Approach 1



http://www.ofzenandcomputing.com/burn-files-cd-dvd-windows7/

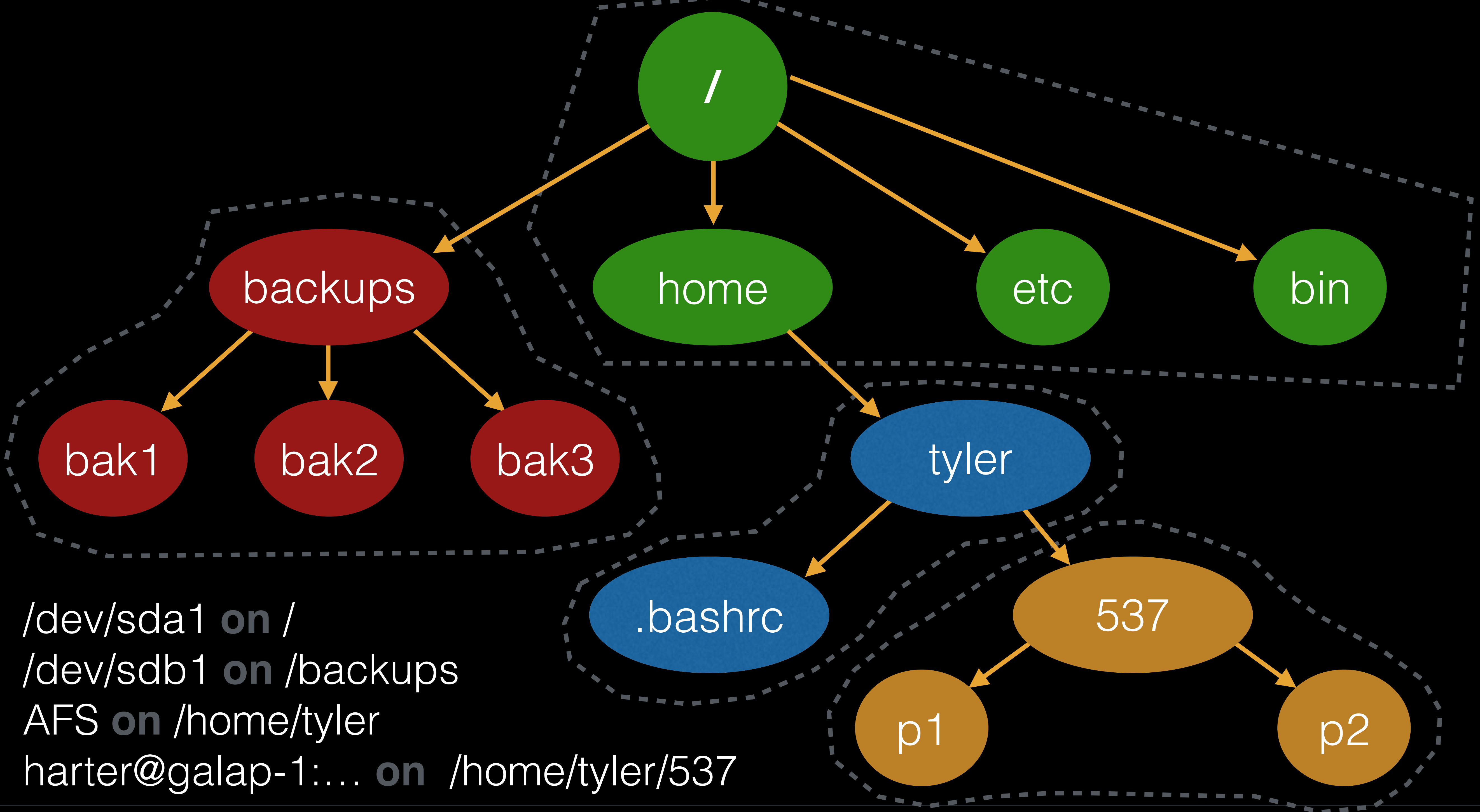
Many File Systems: Approach 2

Idea: stitch all the file systems together into a super file system!

Many File Systems: Approach 2

Idea: stitch all the file systems together into a super file system!

sh> **mount**
/dev/sda1 on / type ext4 (rw)
/dev/sdb1 on /backups type ext4 (rw)
AFS on /home/tyler type afs (rw)
harter@galap-1:~/537_projects /home/tyler/537 type sshfs (rw)



/dev/sda1 **on** /
/dev/sdb1 **on** /backups
AFS **on** /home/tyler
harter@galap-1:… **on** /home/tyler/537

Special Calls

fsync

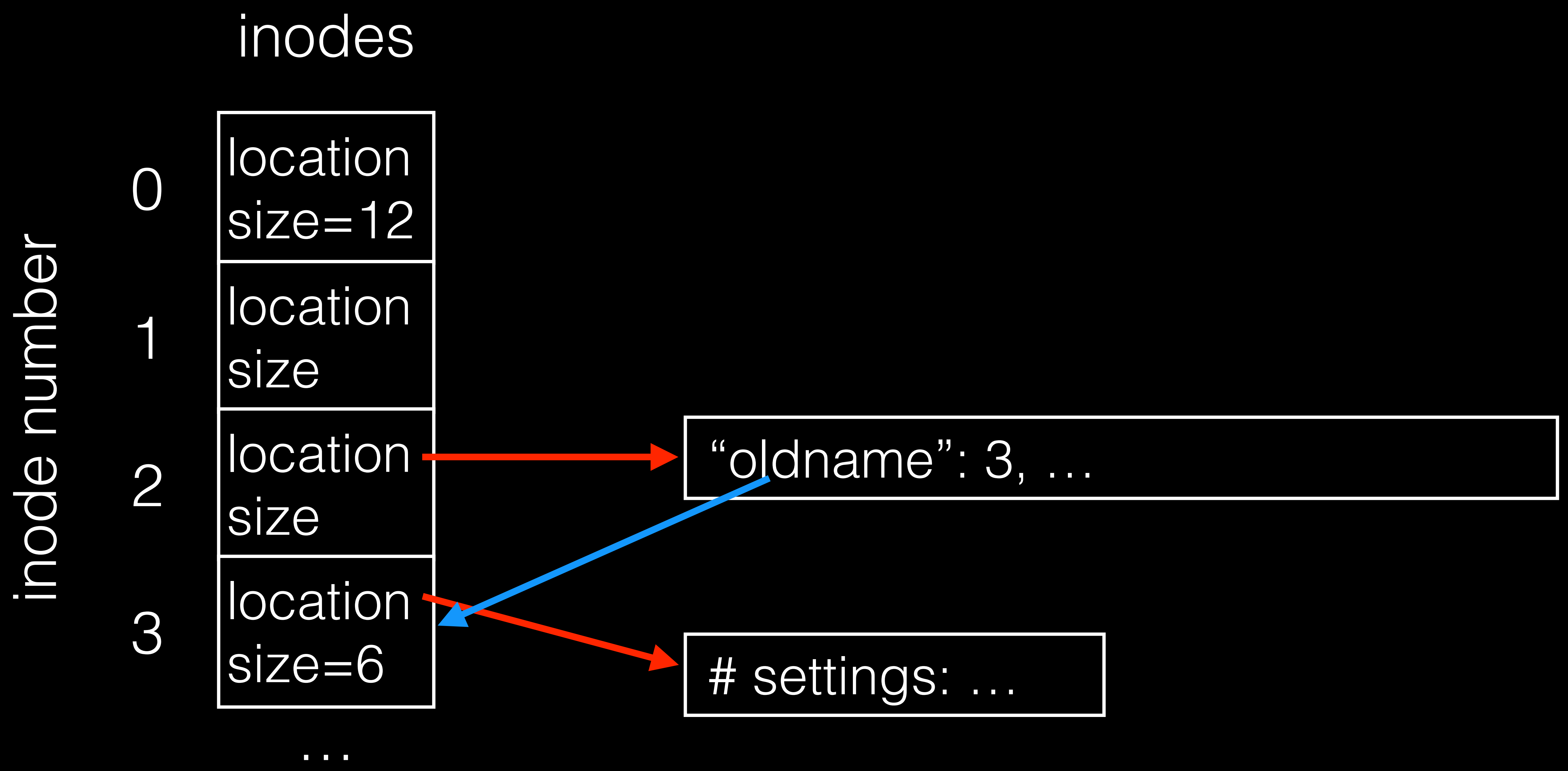
Write buffering improves performance (why?).
But what if we crash before the buffers are flushed?

fsync(int fd) forces buffers to flush to disk, and (usually) tells the disk to flush it's write cache too.

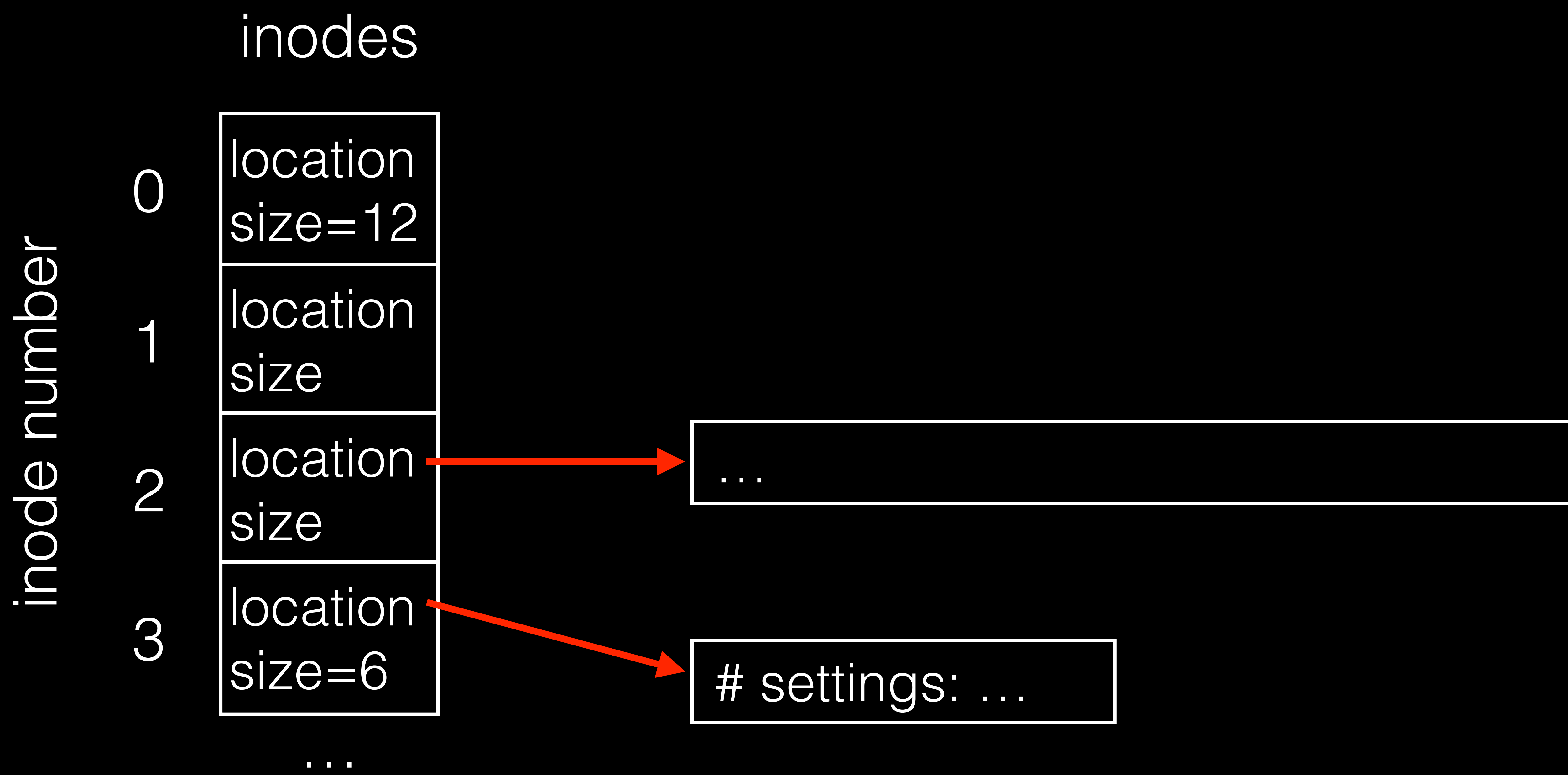
This makes data durable.

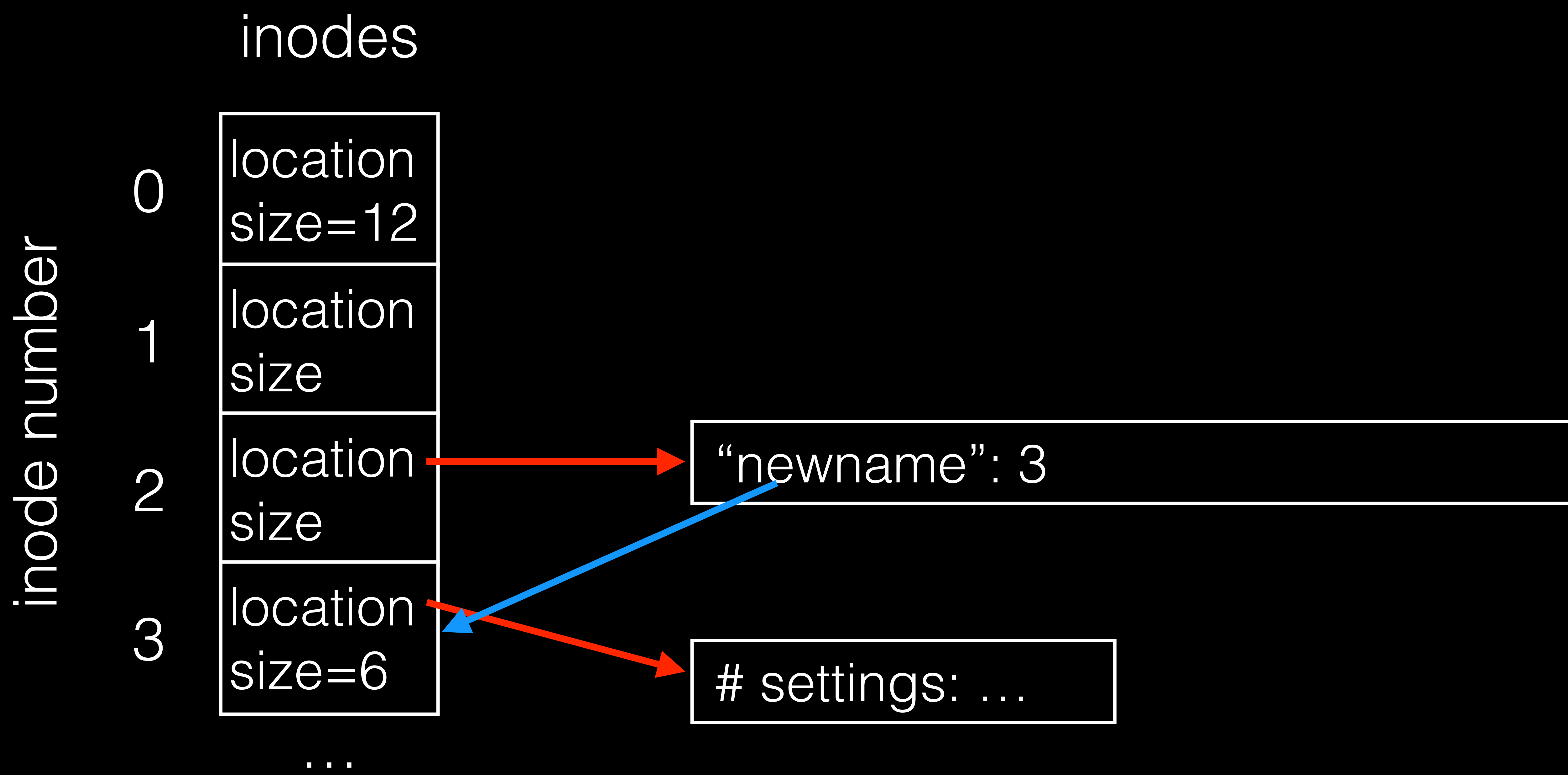
rename

rename(char *old, char *new):
- deletes an old link to a file
- creates a new link to a file



inodes
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
…
“oldname”: 3, …
settings: …


inodes
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
…
…
settings: …


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“newname”: 3
settings: …

rename

rename(char *old, char *new):
- deletes an old link to a file
- creates a new link to a file

rename

rename(char *old, char *new):
- deletes an old link to a file
- creates a new link to a file

What if we crash?

rename

rename(char *old, char *new):
 - deletes an old link to a file
 - creates a new link to a file

What if we crash?
FS does extra work to guarantee atomicity.

Atomic File Update

Say we want to update file.txt.

1. write new data to new file.txt.tmp file
2. fsync file.txt.tmp
3. rename file.txt.tmp over file.txt, replacing it

Concurrency

How can multiple processes avoid updating the same file at the same time?

Concurrency

How can multiple processes avoid updating the same file at the same time?

Normal locks don't work, as developers may have developed their programs independently.

Concurrency

How can multiple processes avoid updating the same file at the same time?

Normal locks don't work, as developers may have developed their programs independently.

Use flock(), for example:

- flock(fd, LOCK_EX)
- flock(fd, LOCK_UN)

Summary

Using multiple types of name provides
- convenience
- efficiency

Mount and link features provide flexibility.

Special calls (fsync, rename, flock) let developers communicate special requirements to FS.

Implementation

Implementation

1. On-disk structures
   - how do we represent files, directories?

2. Access methods
   - what steps must reads/writes take?

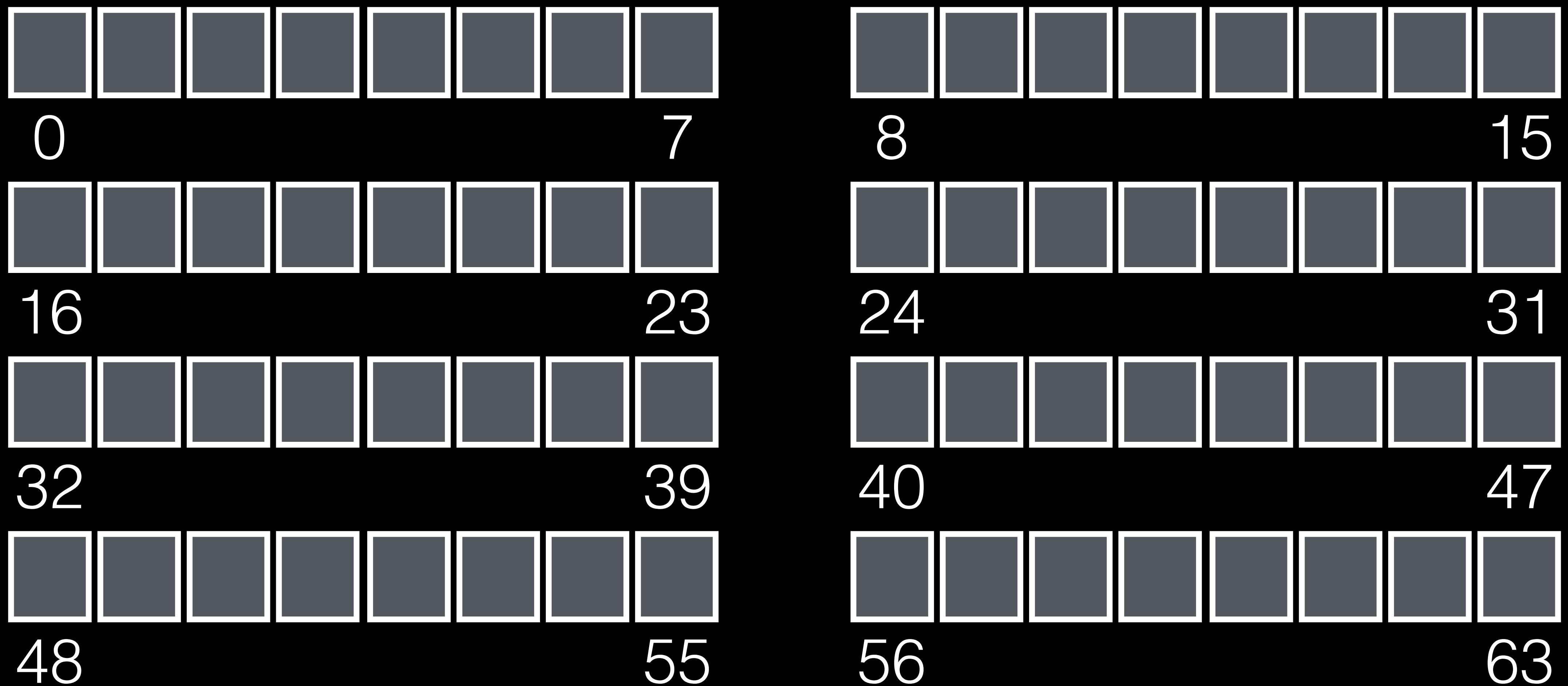
Disk Structures

Structures

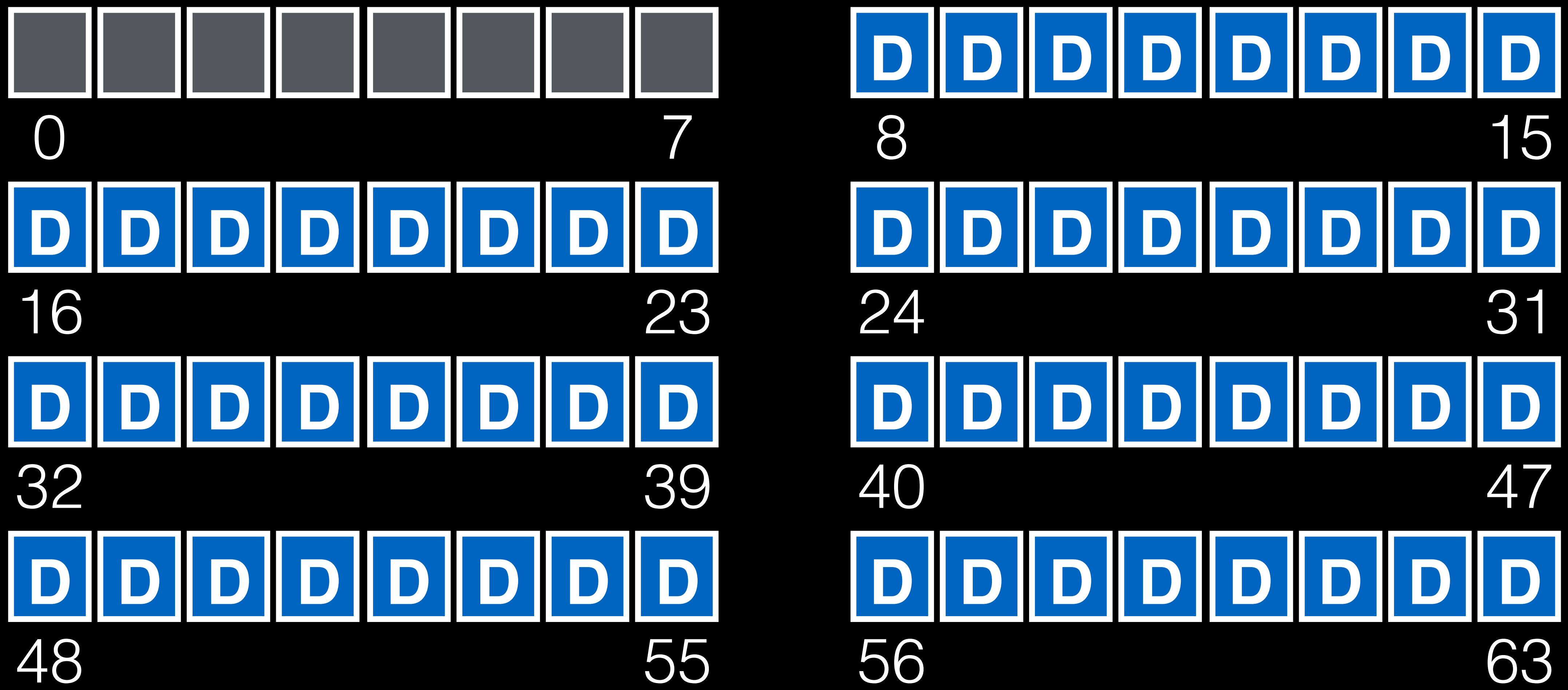
What data is likely to be read frequently?

- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

FS Structs: Empty Disk



Data Blocks



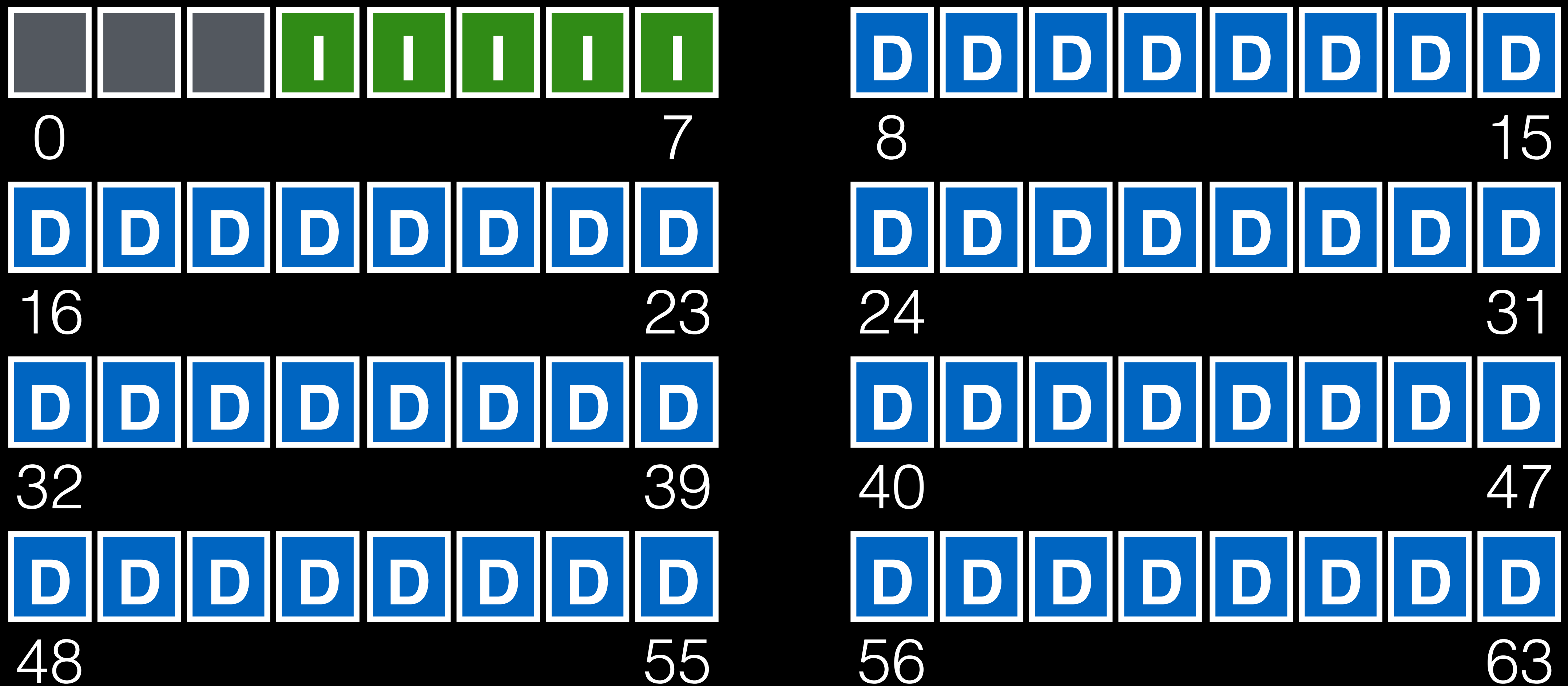
Structures

What data is likely to be read frequently?
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

Structures

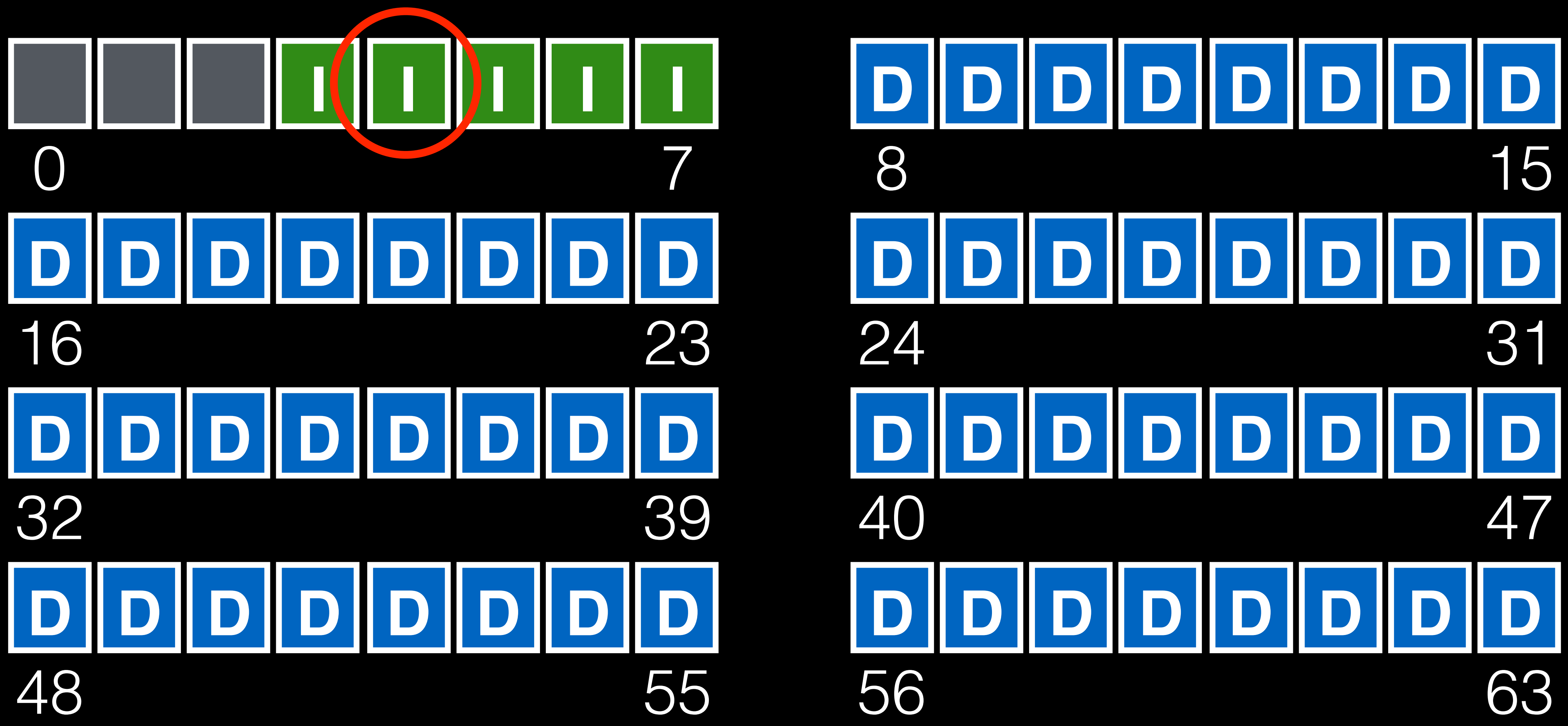
What data is likely to be read frequently?
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

Inodes



I I I I I
0
7
D D D D D D D D
8
15
D D D D D D D D
16
23
D D D D D D D D
24
31
D D D D D D D D
32
39
D D D D D D D D
40
47
D D D D D D D D
48
55
D D D D D D D D
56
63

Inodes


I I I I I
0
7
D D D D D D D D
8
15
D D D D D D D D
16
23
D D D D D D D D
24
31
D D D D D D D D
32
39
D D D D D D D D
40
47
D D D D D D D D
48
55
D D D D D D D D
56
63

Inode Block

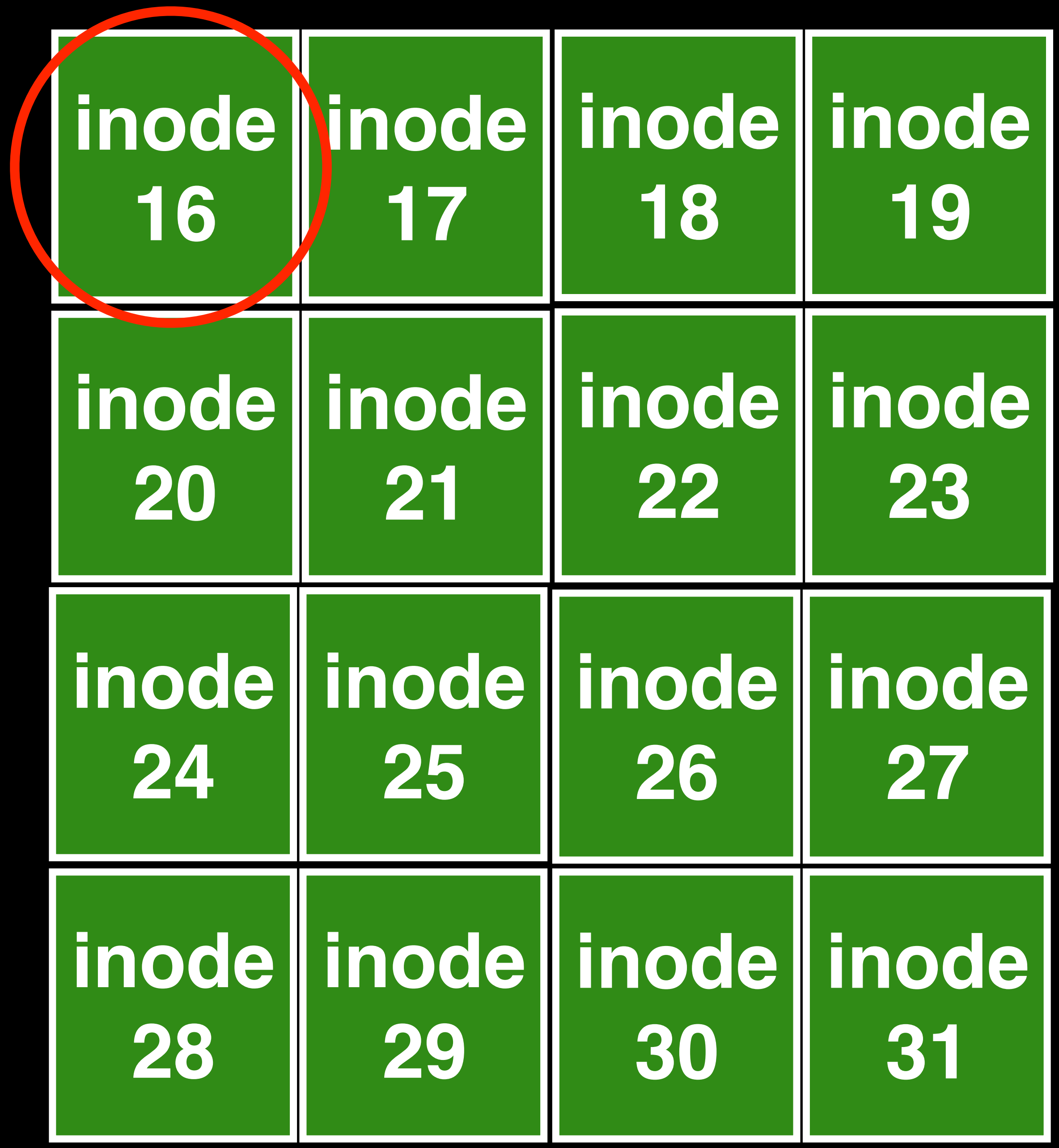
Inodes are typically 128 or 256 bytes (depends on the FS).

So 16 - 32 inodes per inode block.

Inode Block

Inodes are typically 128 or 256 bytes (depends on the FS).

So 16 - 32 inodes per inode block.



Inode

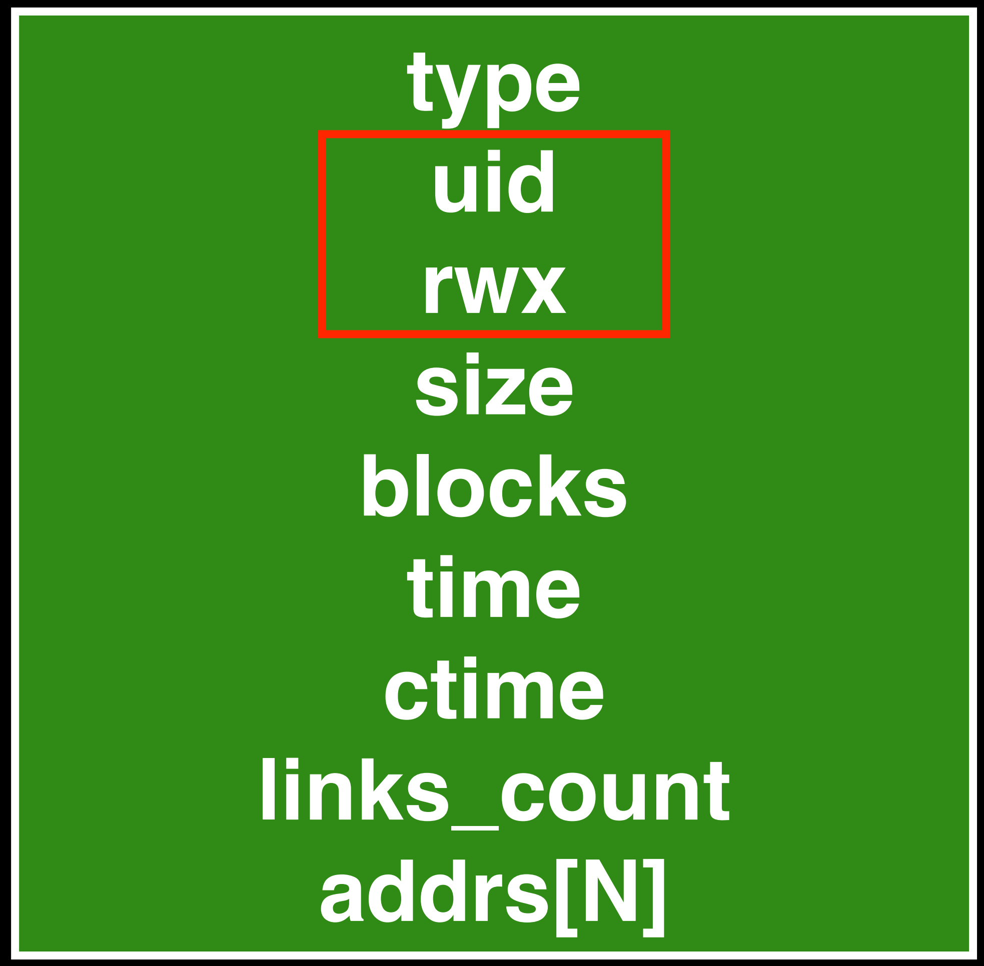
type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

file or directory?

Inode



user and permissions

Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

size in bytes and blocks

Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

access time, create time

Inode

- type
- uid
- rwx
- size
- blocks
- time
- ctime
- **links_count**
- addrs[N]

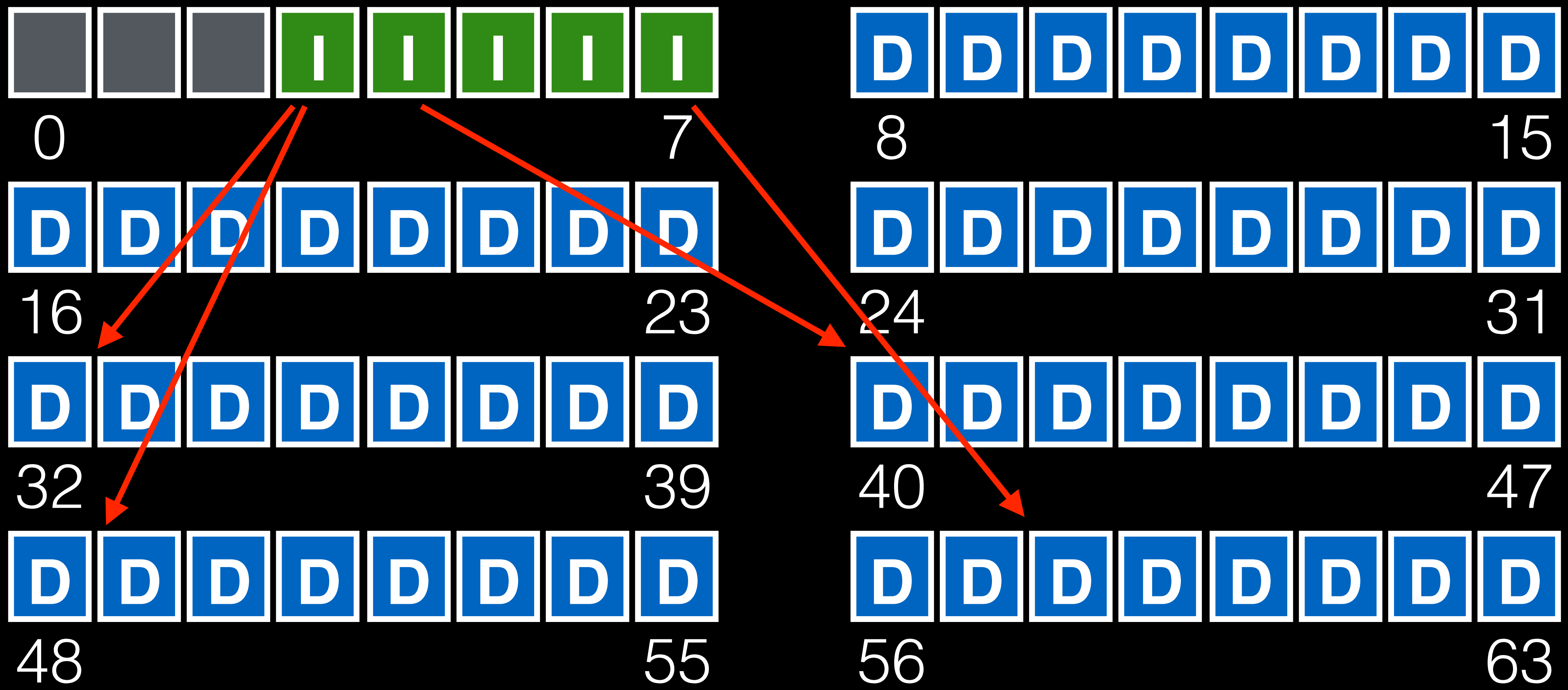
how many paths

Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

N data blocks

Inodes



Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

Assume 4-byte addrs.
What is an upper bound
on the file size?
(assume 256-byte inodes)

Inode

type
uid
rwx
size
blocks
time
ctime
links_count
addrs[N]

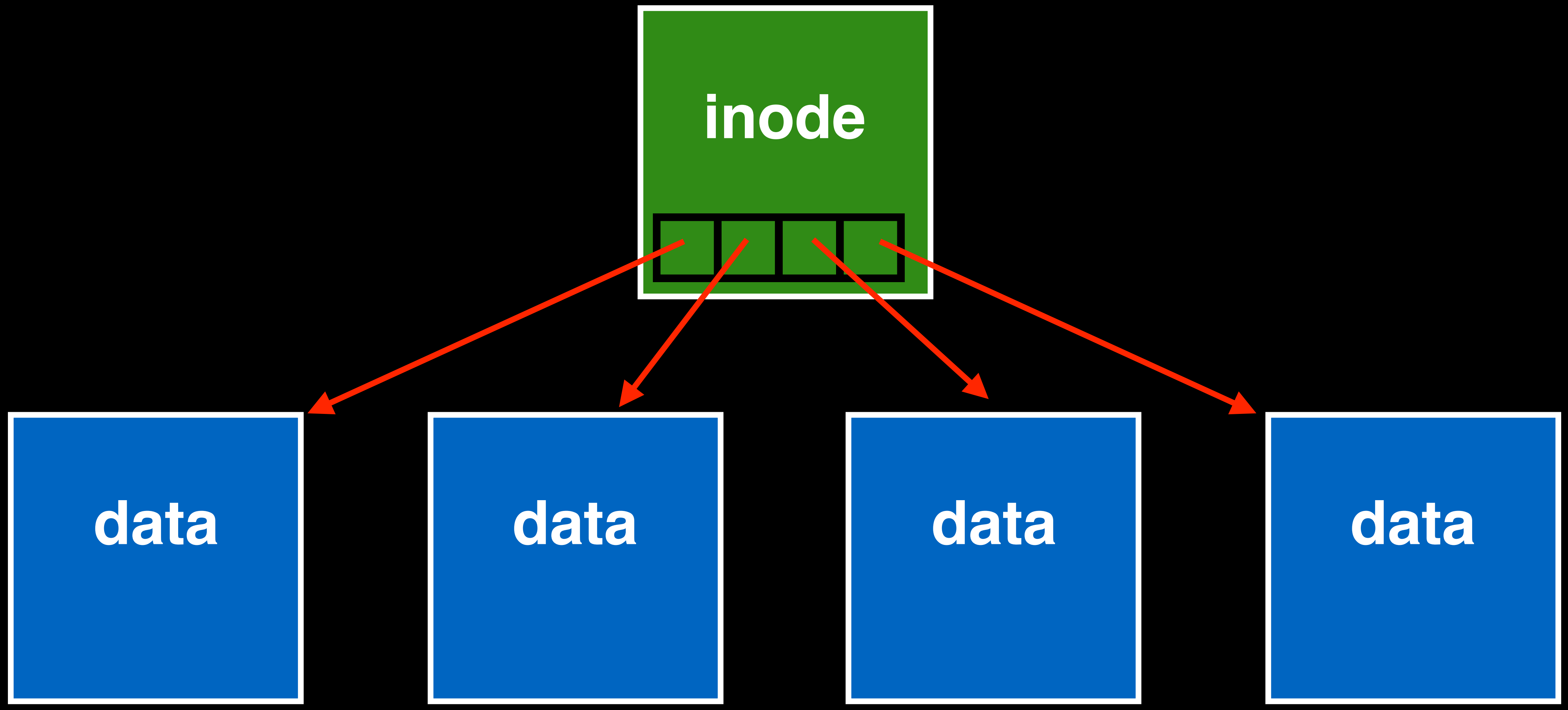
Assume 4-byte addrs.
What is an upper bound on the file size?
(assume 256-byte inodes)

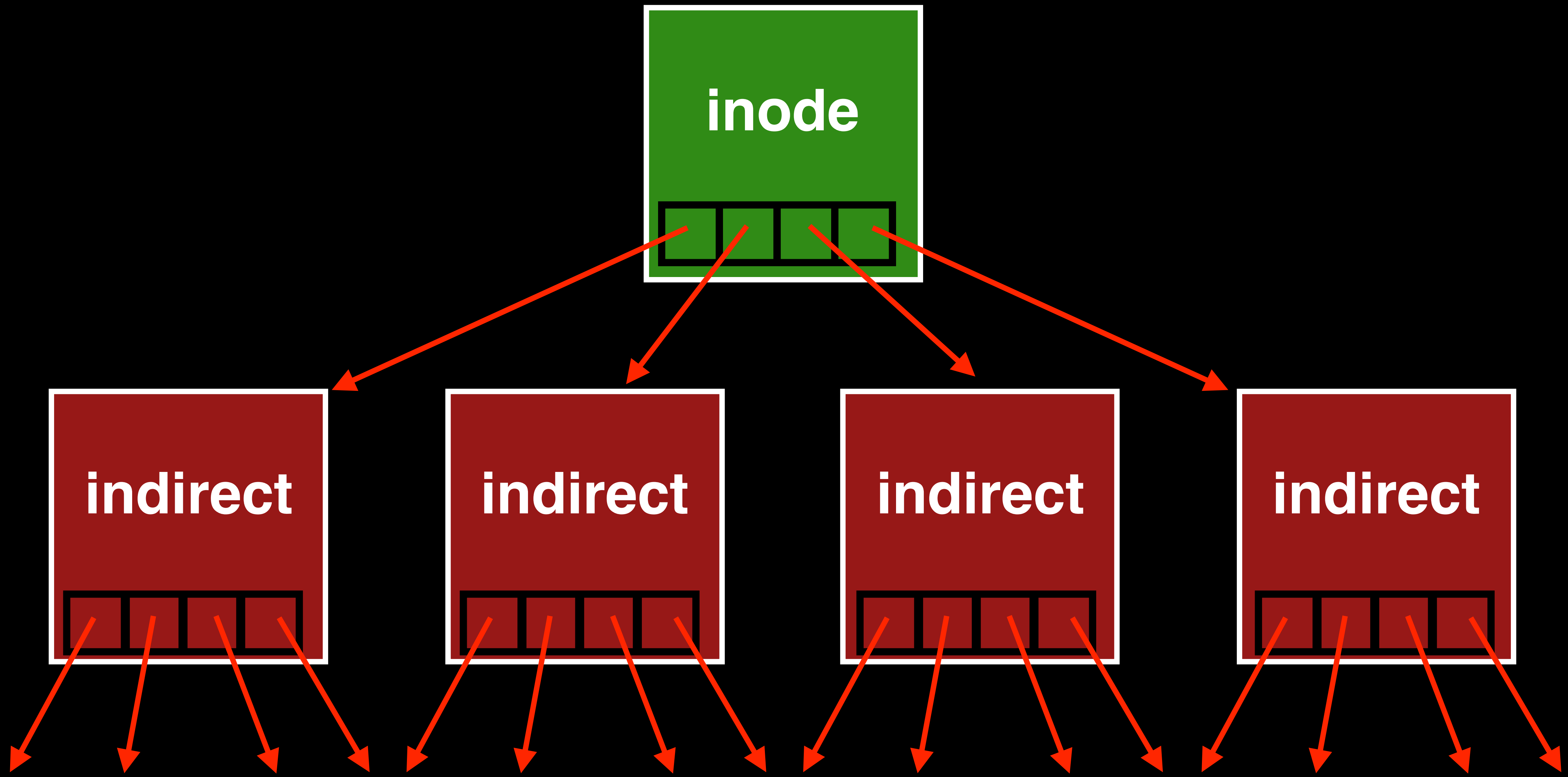
How to get larger files?

Structures

What data is likely to be read frequently?

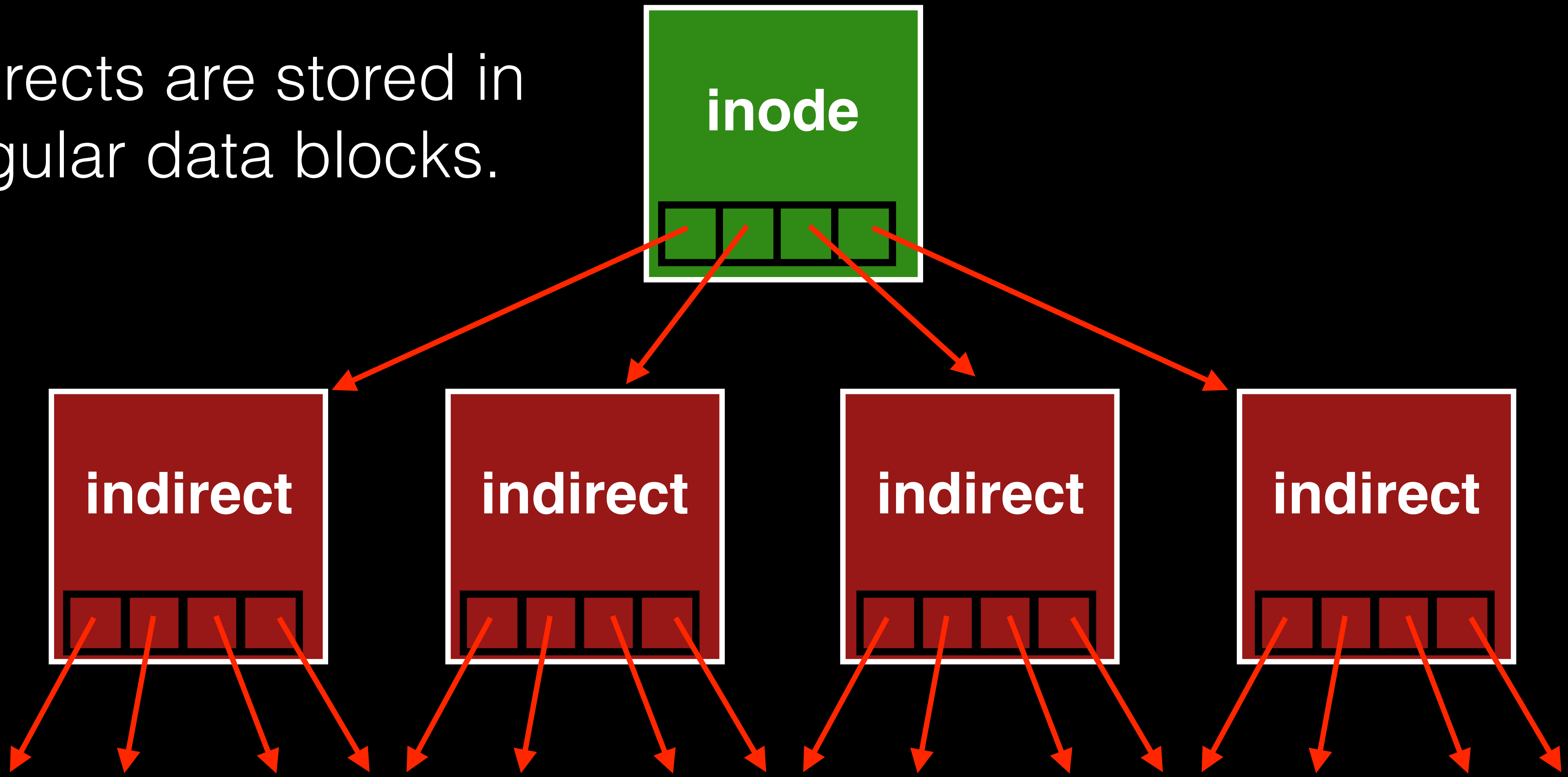
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock


inode
data
data
data
data

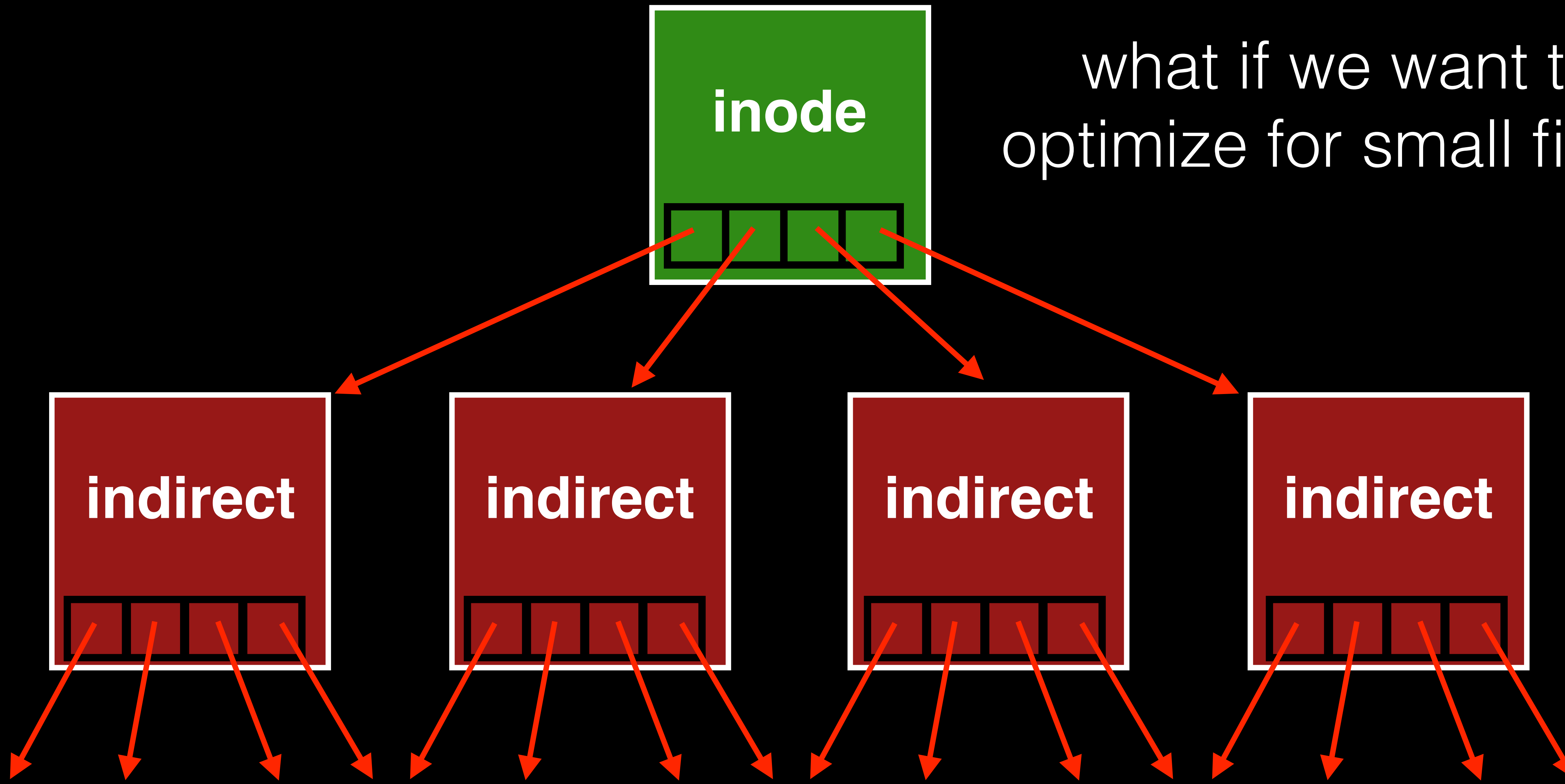


inode
indirect
indirect
indirect
indirect

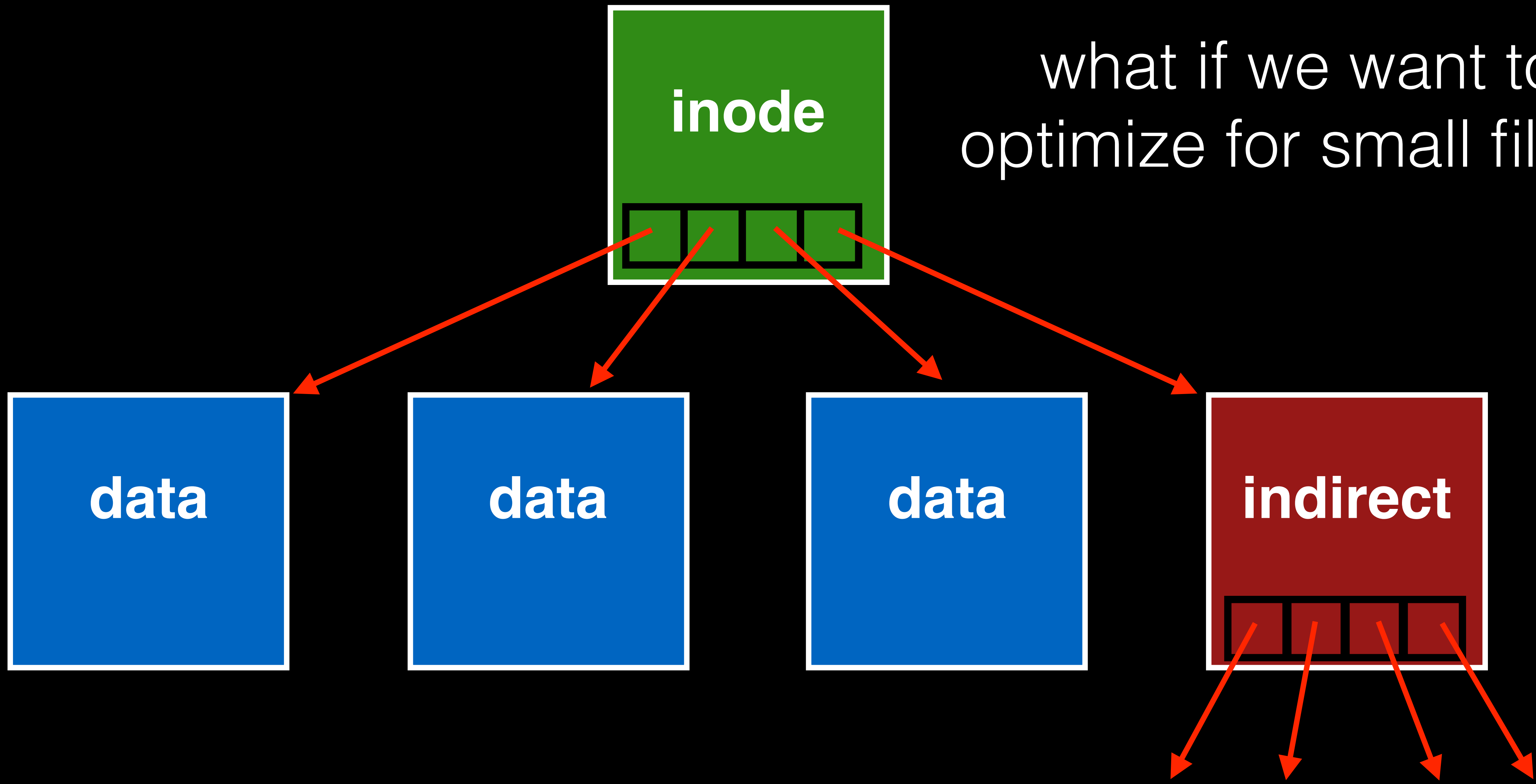
indirects are stored in regular data blocks.



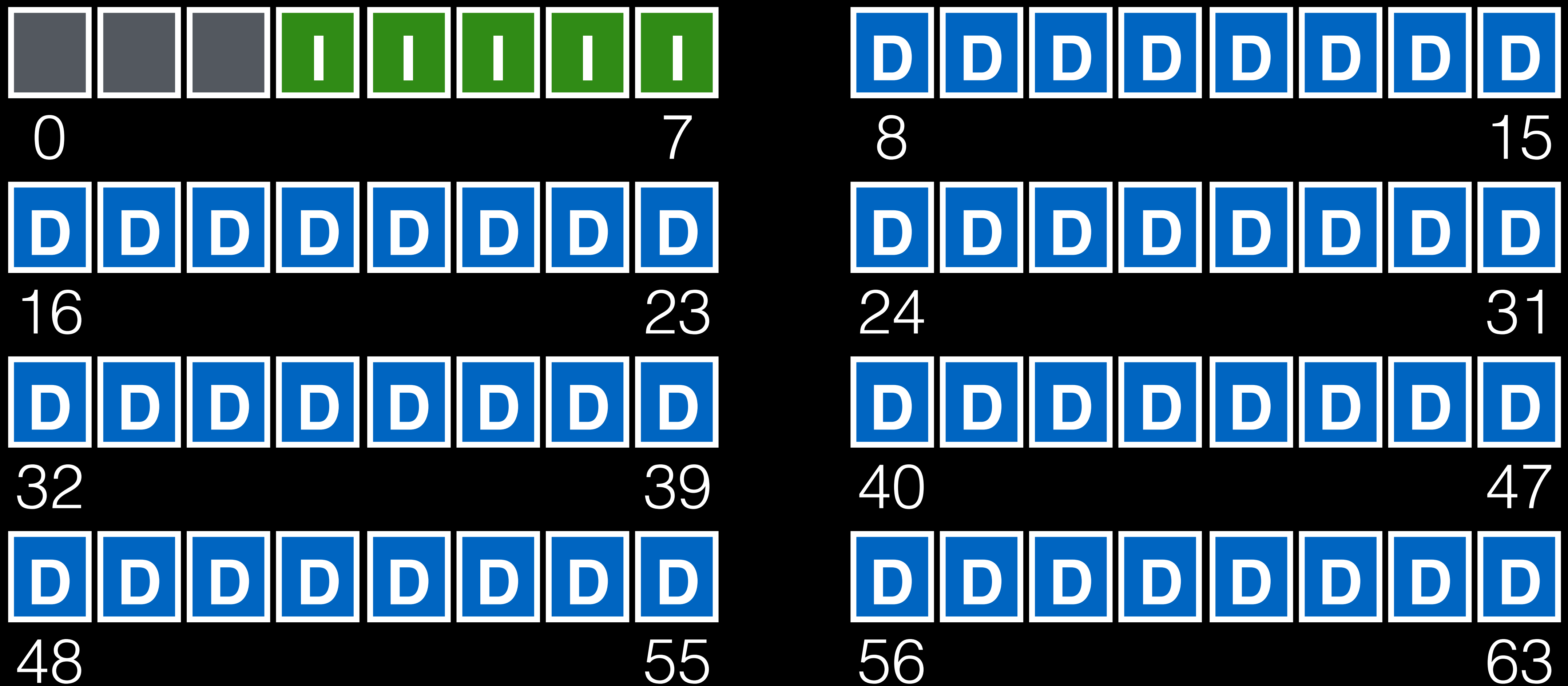
what if we want to optimize for small files?



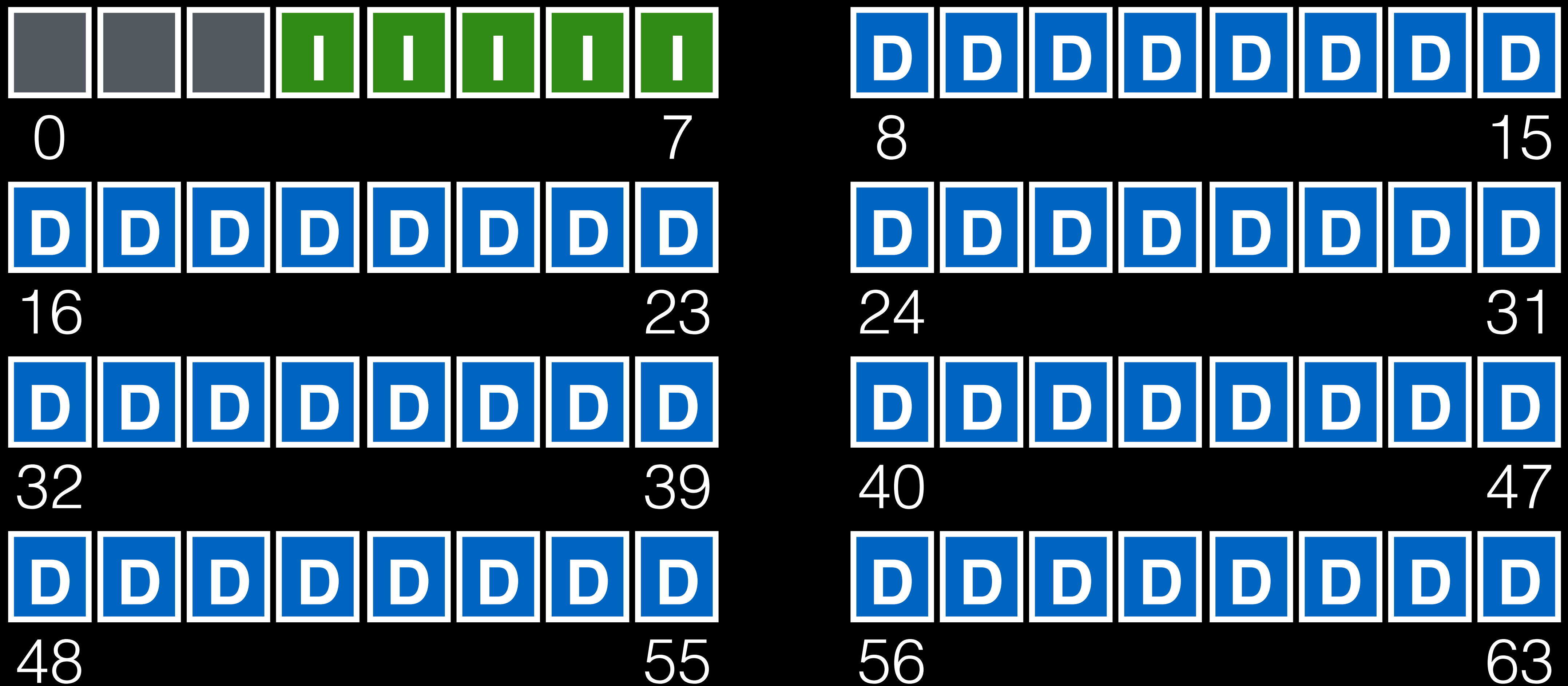
what if we want to
optimize for small files?



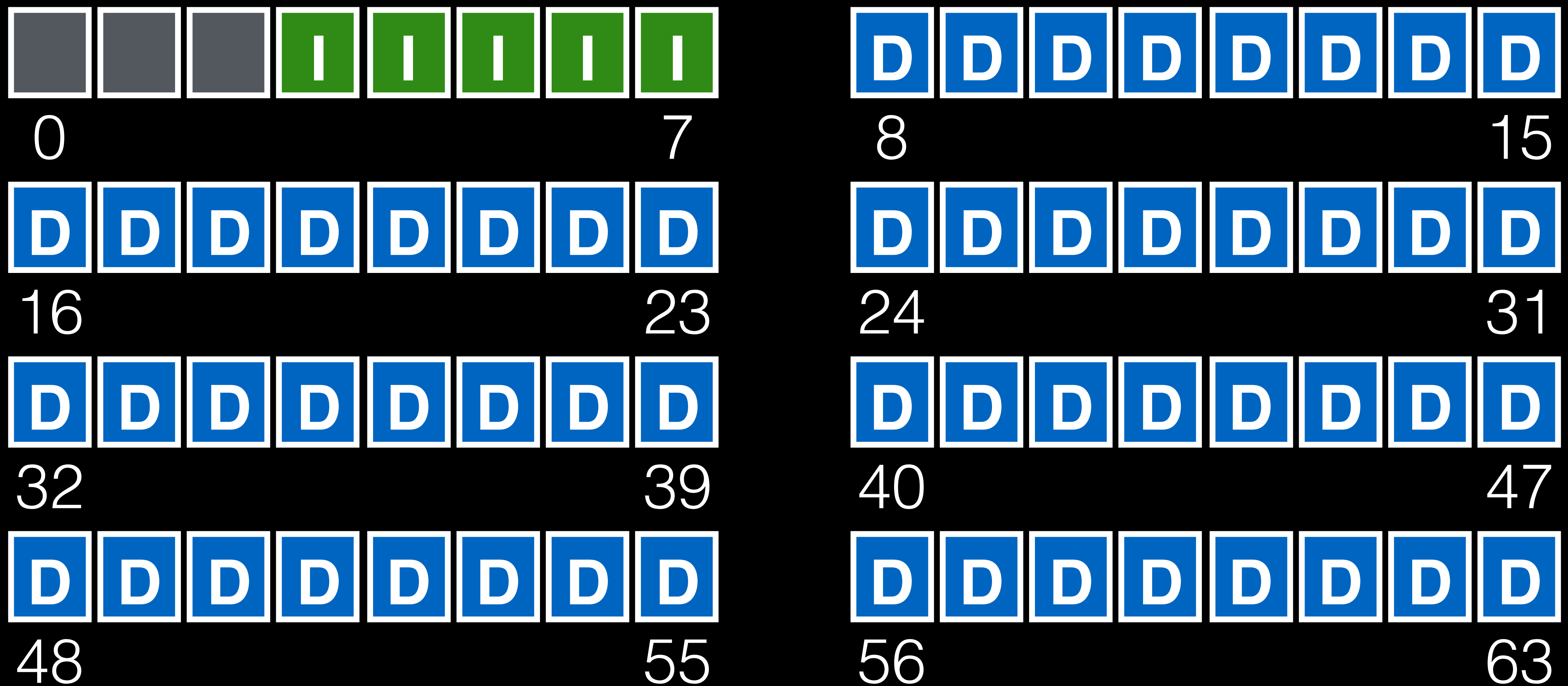
Assume 256 byte sectors. What is offset for inode with number 0?



Assume 256 byte sectors. What is offset for inode with number 4?



Assume 256 byte sectors. What is offset for inode with number 40?



Various Link Structures

Tree (usually unbalanced)
- with indirect blocks
- e.g., ext3

Extents
- store offset+size pairs
- e.g., ext4

Linked list
- each data block points to the next
- e.g., FAT

Structures

What data is likely to be read frequently?

- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

Directories

File systems vary.

Common design: just store directory entries in files.

Various formats could be used
- lists
- b-trees

Simple List Example

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

Simple List Example

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

unlink("foo")

Structures

What data is likely to be read frequently?
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

Allocation

How do we find free data blocks or free inodes?

Allocation

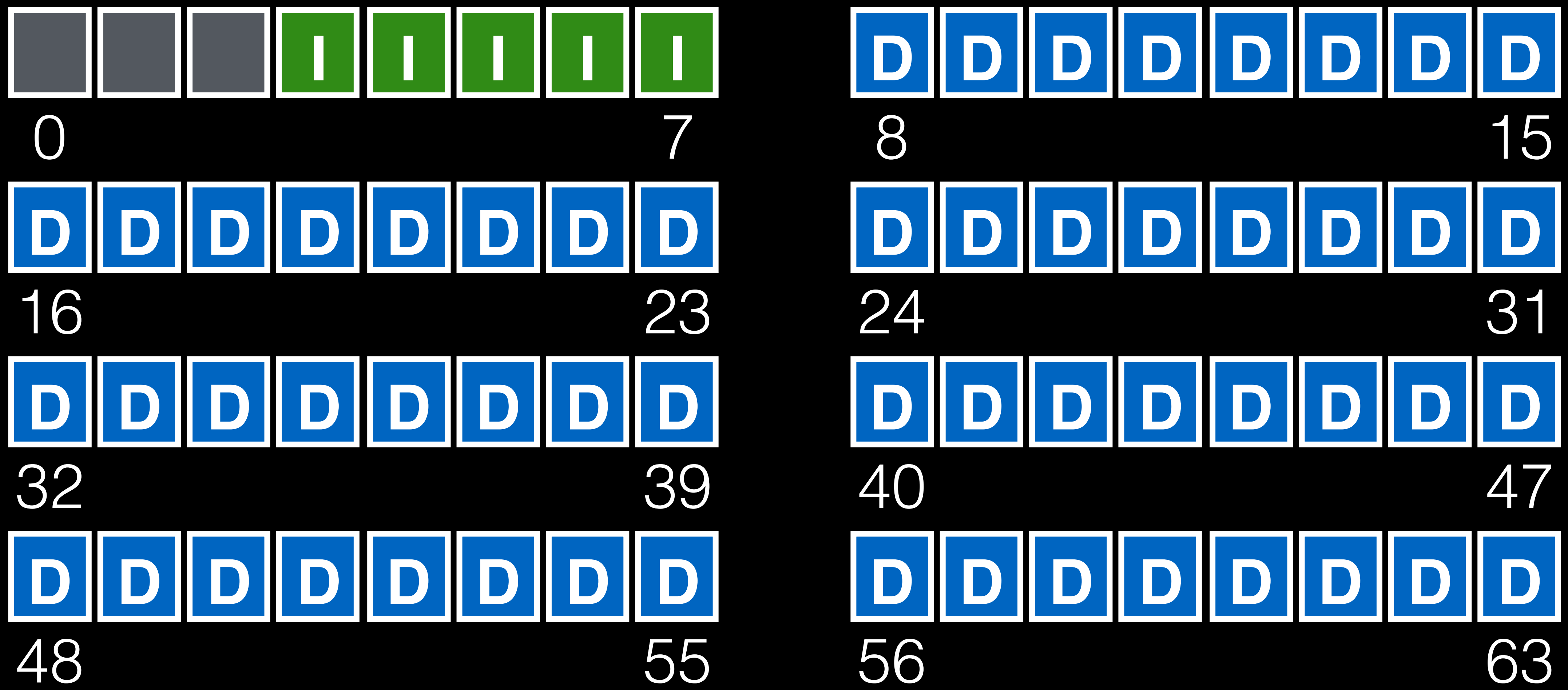
How do we find free data blocks or free inodes?

Free list.

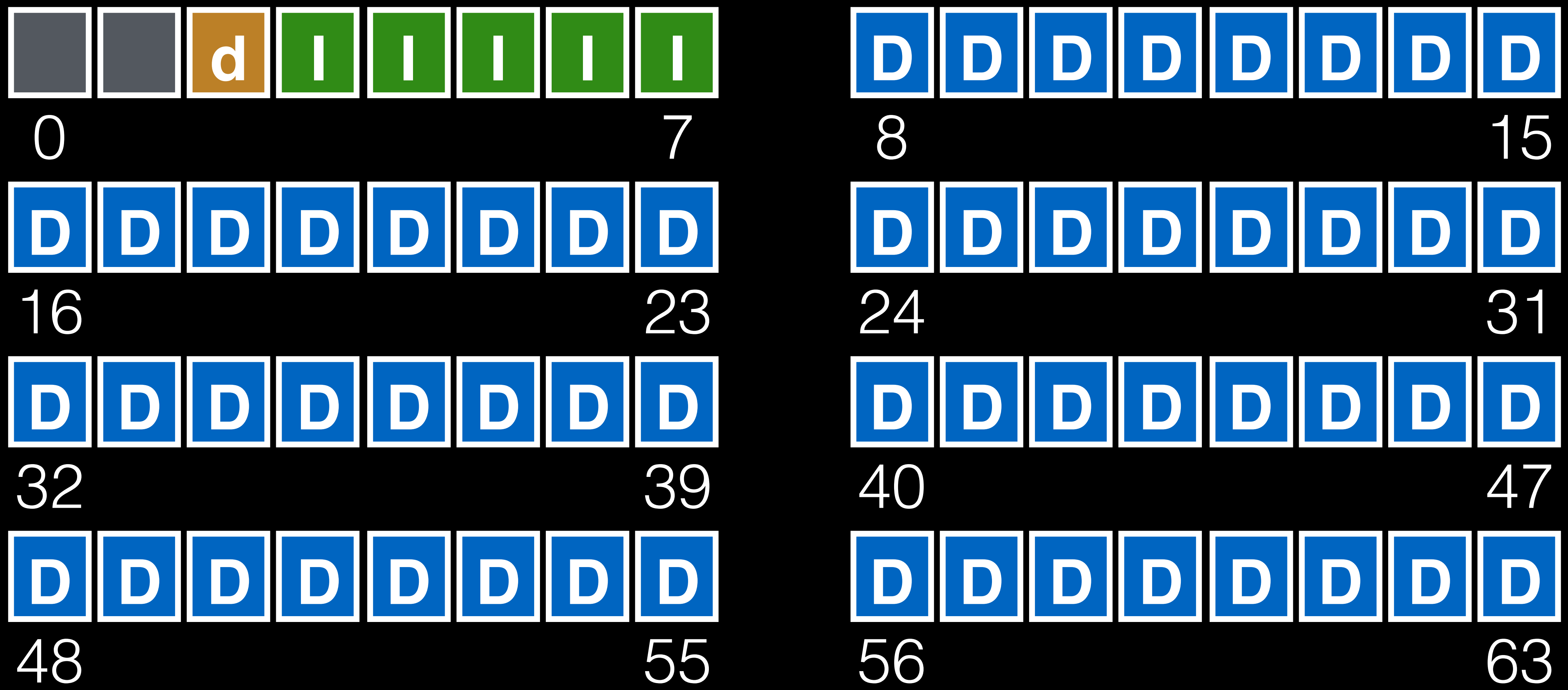
Bitmaps.

Tradeoffs?

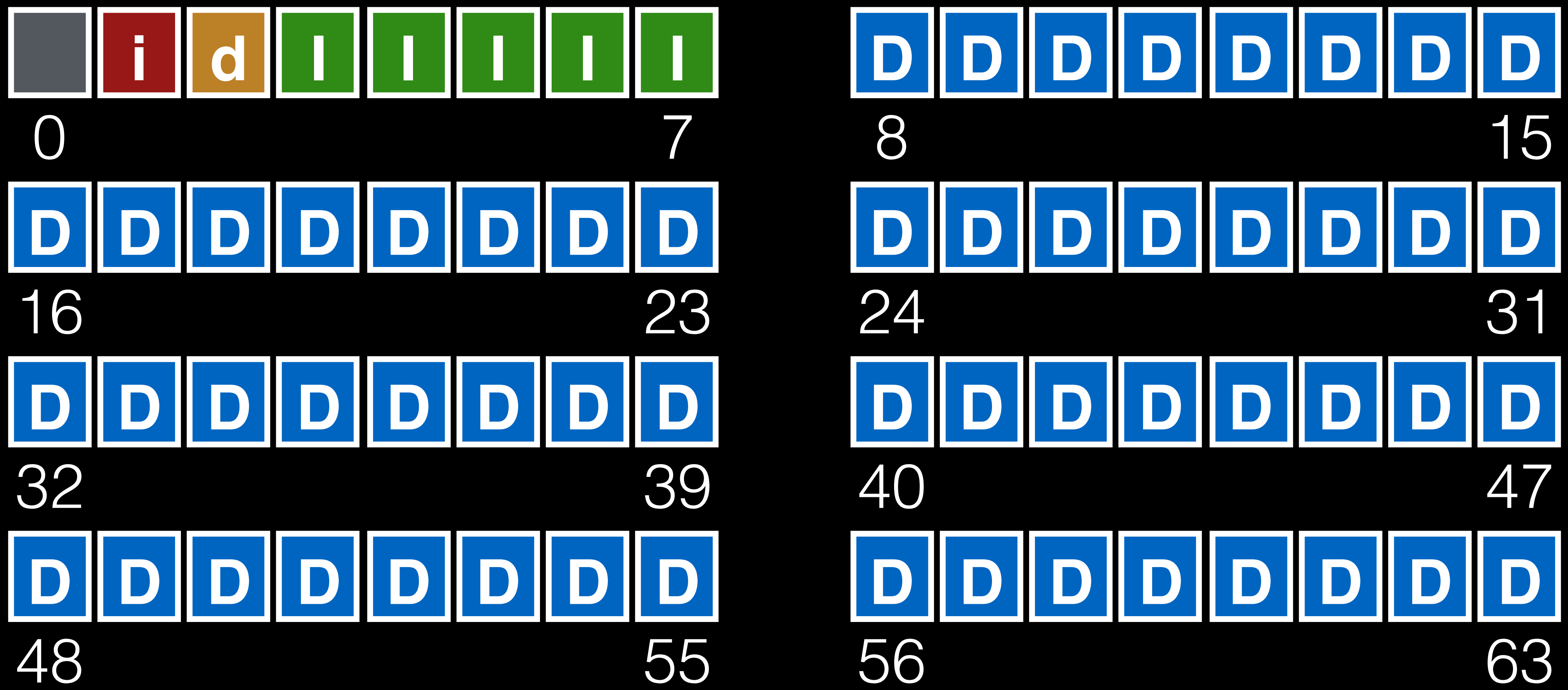
Bitmaps



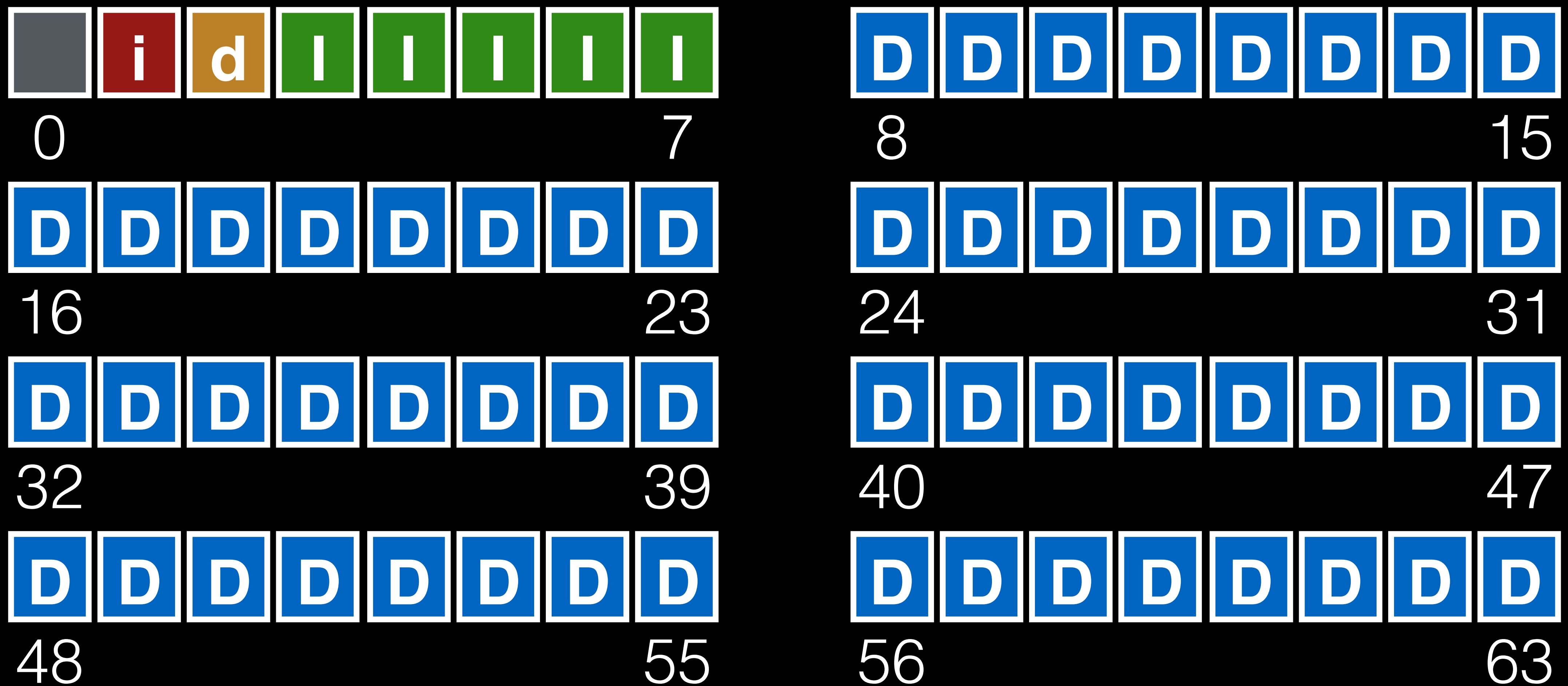
Data Bitmap



Inode Bitmap



Opportunity for Inconsistency (fsck)



Structures

What data is likely to be read frequently?
- data block
- inode table
- indirect block
- directories
- data bitmap
- inode bitmap
- superblock

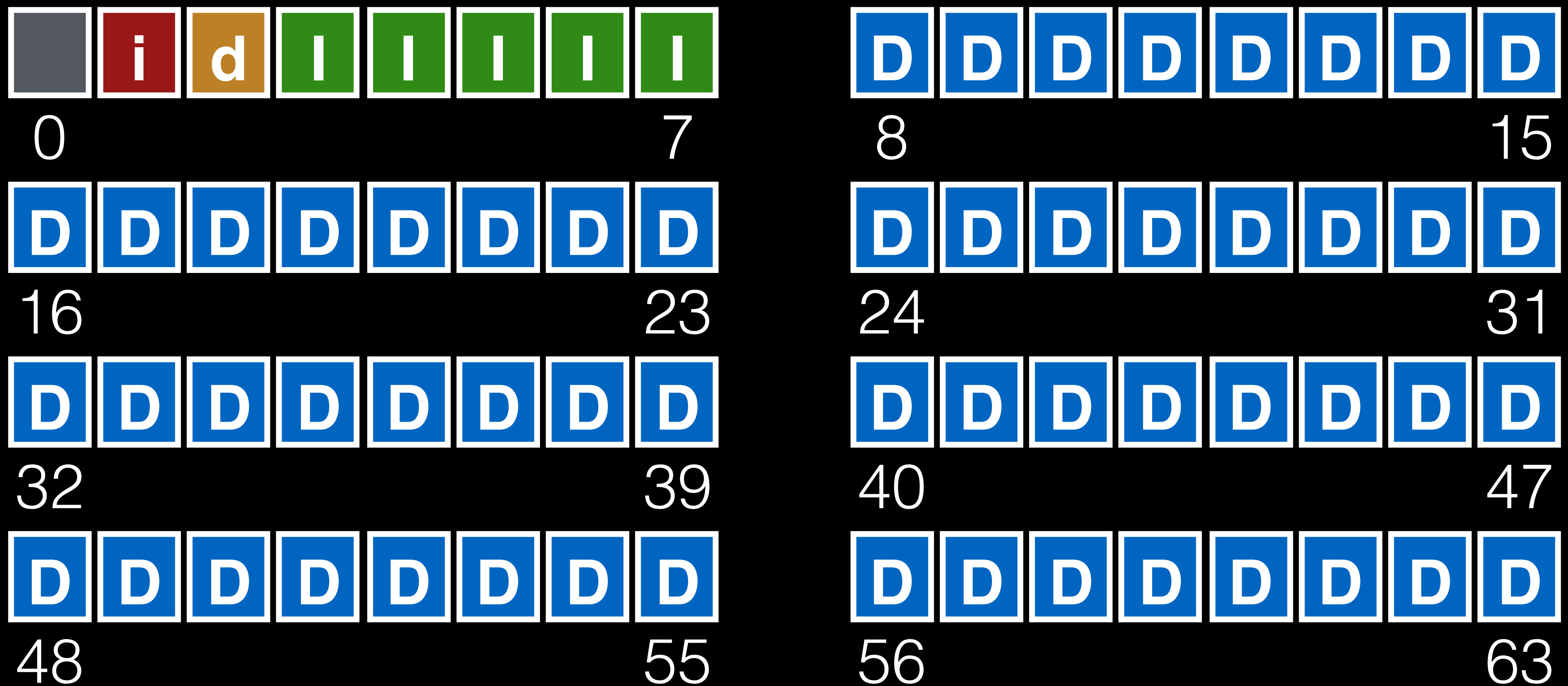
Superblock

Need to know basic FS metadata, like:

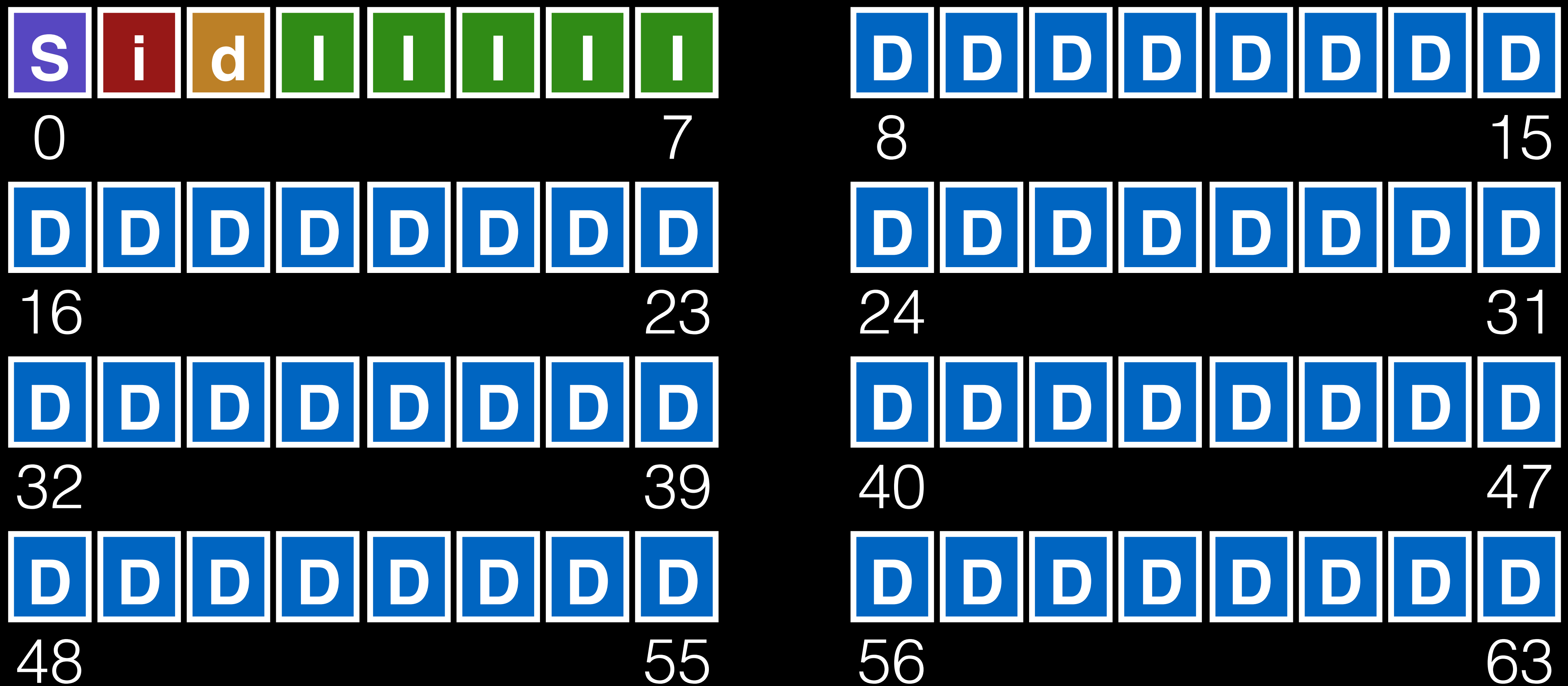
- block size
- how many inodes are there
- how much free data

Store this in a superblock

Super Block



Super Block



Structure Overview

Structures:
- superblock
- data block
- data bitmap
- inode table
- inode bitmap
- indirect block
- directories

Structure Overview

Core

Performance

Super Block

Structure Overview

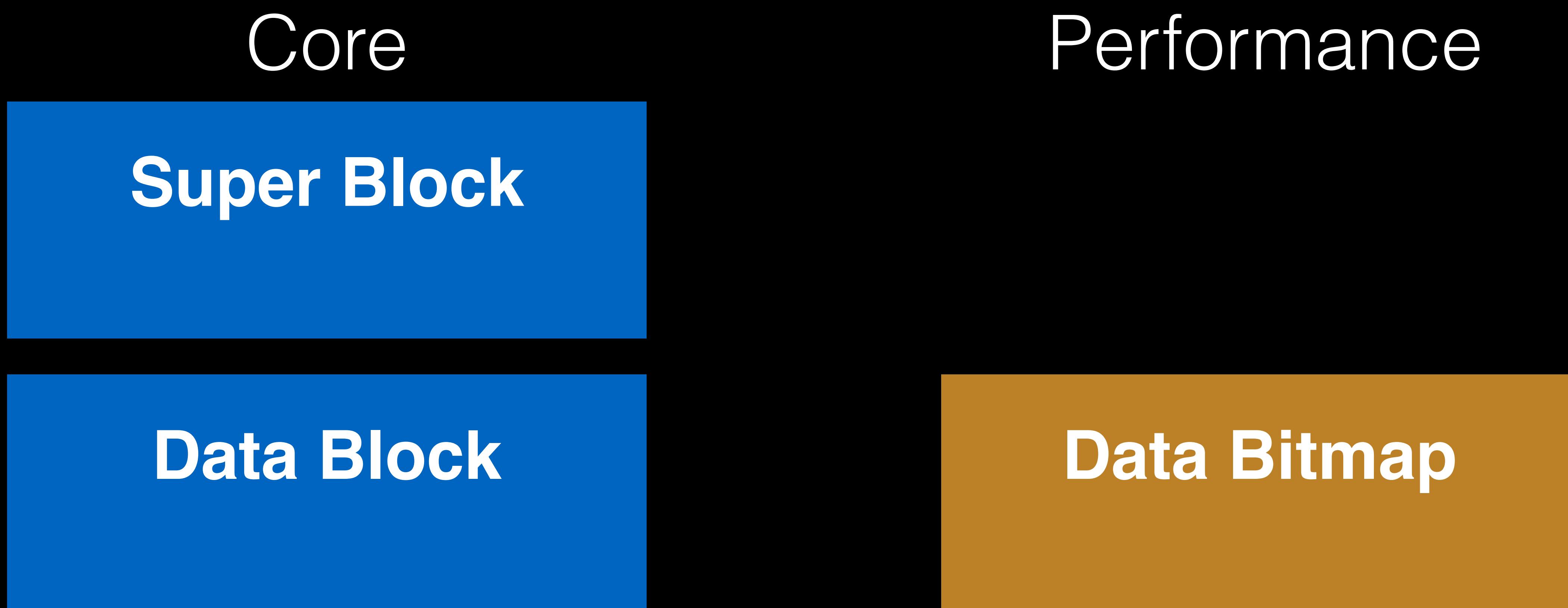
Core

Performance

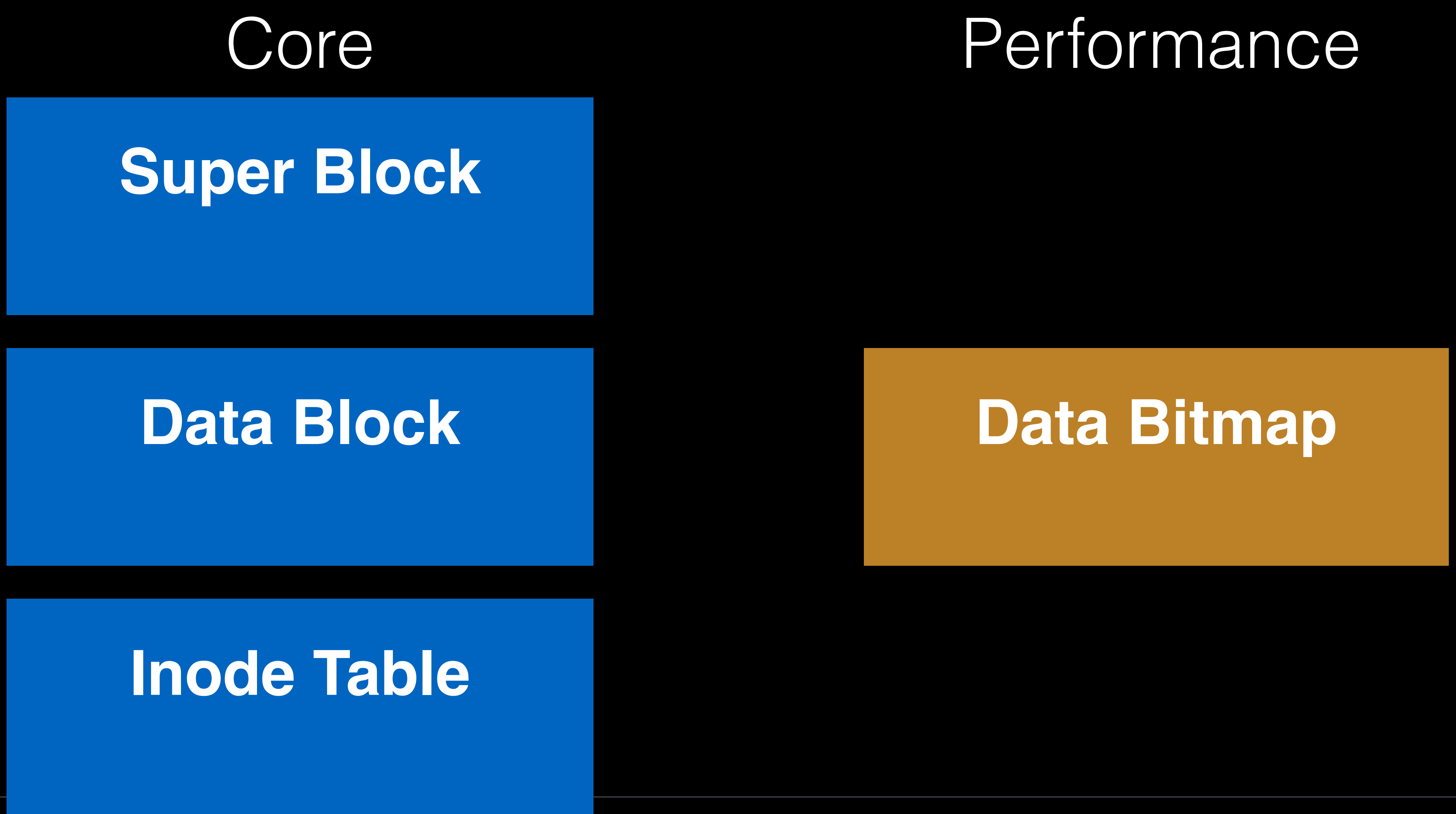
Super Block

Data Block

Structure Overview



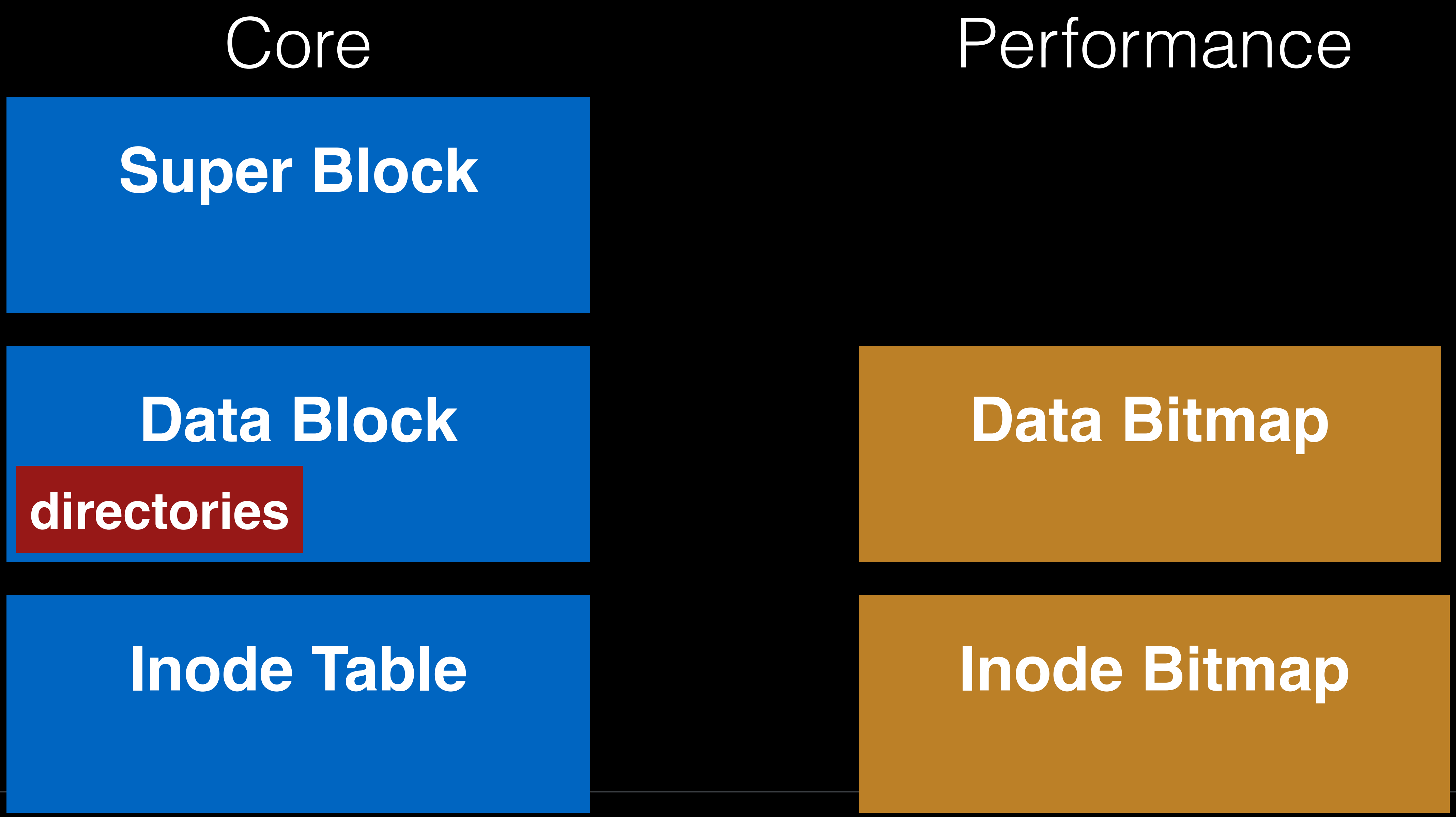
Structure Overview



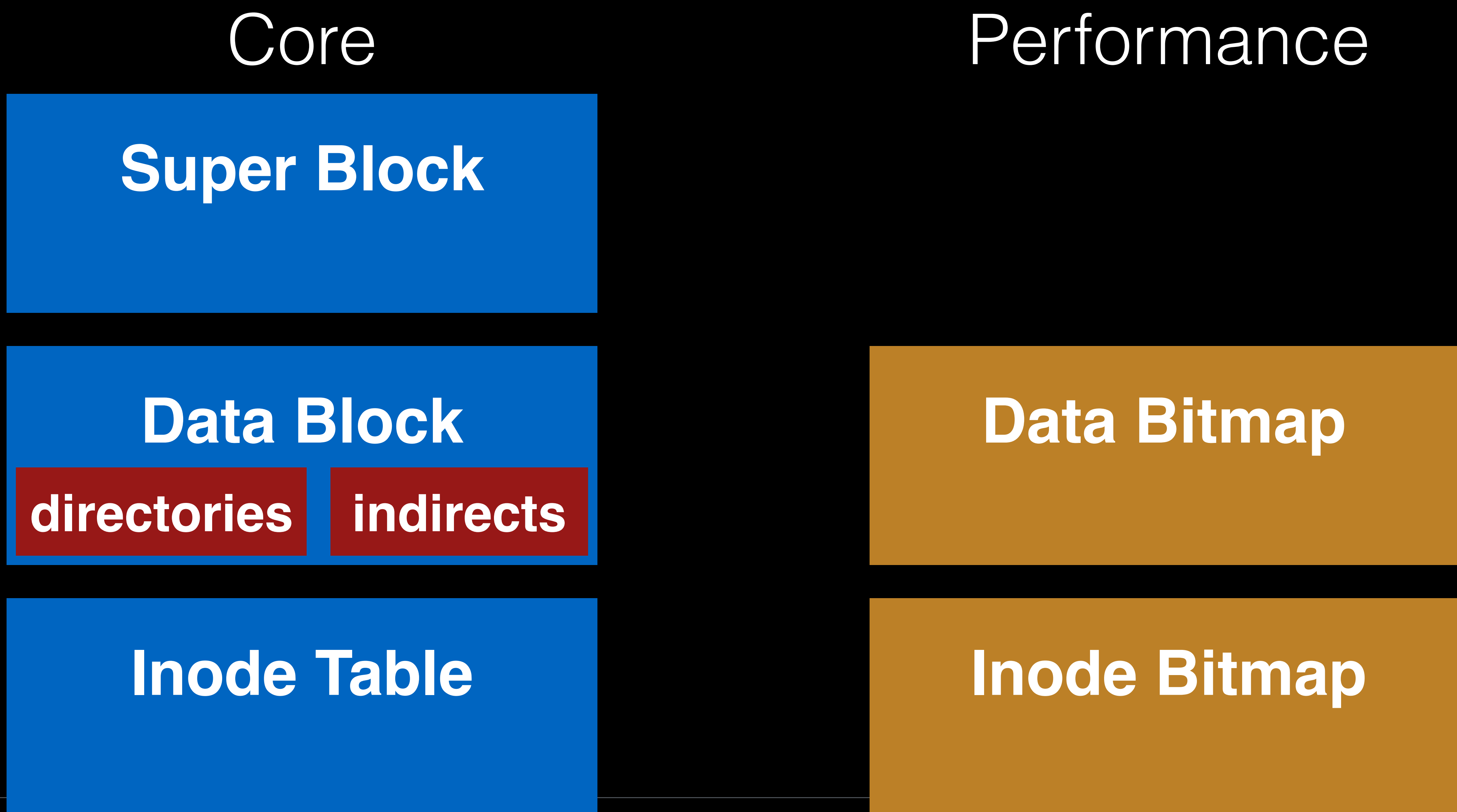
Structure Overview

Core	Performance
Super Block	
Data Block	**Data Bitmap**
Inode Table	**Inode Bitmap**

Structure Overview



Structure Overview



Operations

Operations

FS
- mkfs
- mount

File
- create
- write
- open
- read
- close

Operations

FS
- mkfs
- mount

File
- create
- write
- open
- read
- close

mkfs

Different version for each file system
(e.g., mkfs.ext4, mkfs.xfs, mkfs.btrfs, etc)

Initialize metadata (bitmaps, inode table).

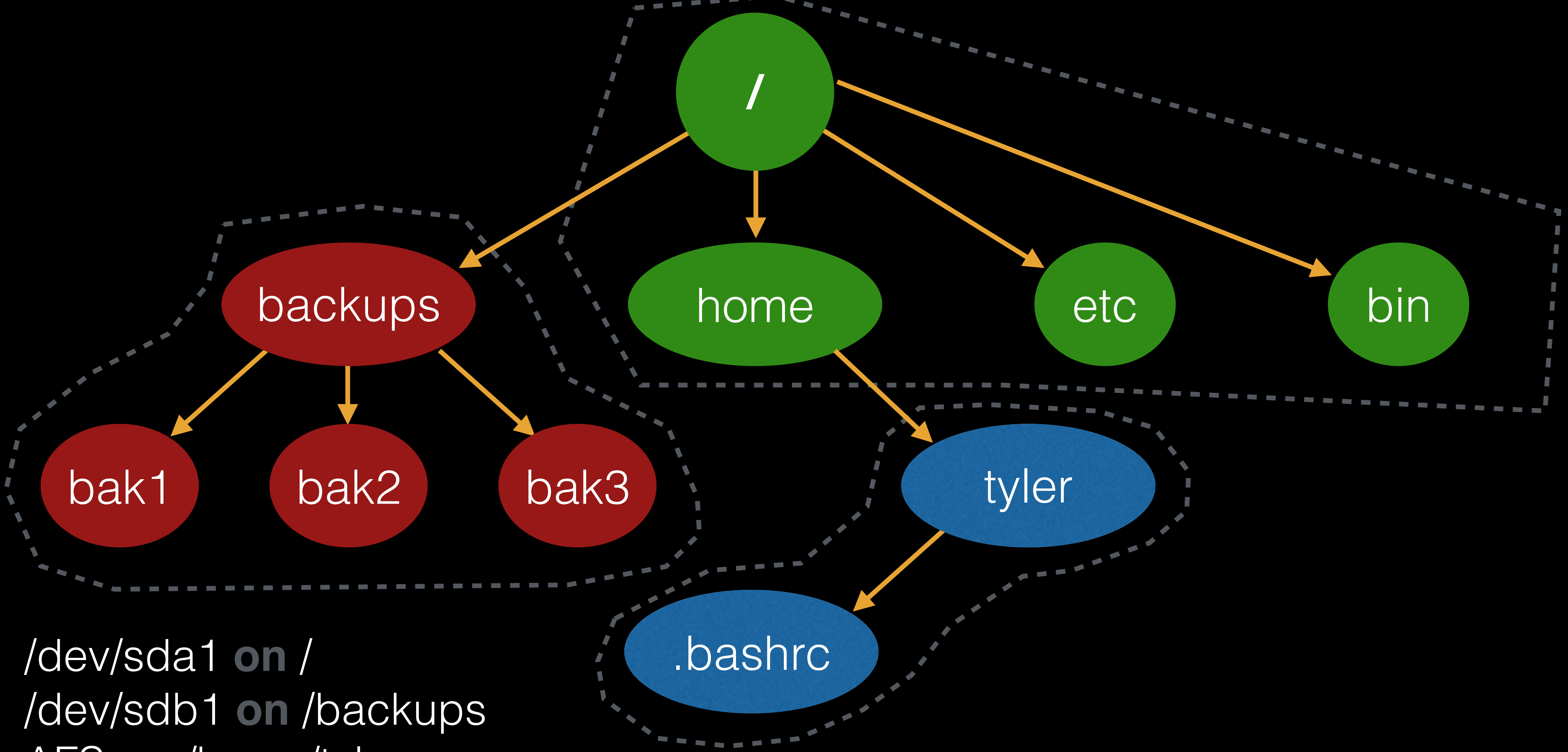
Create empty root directory.

Demo…

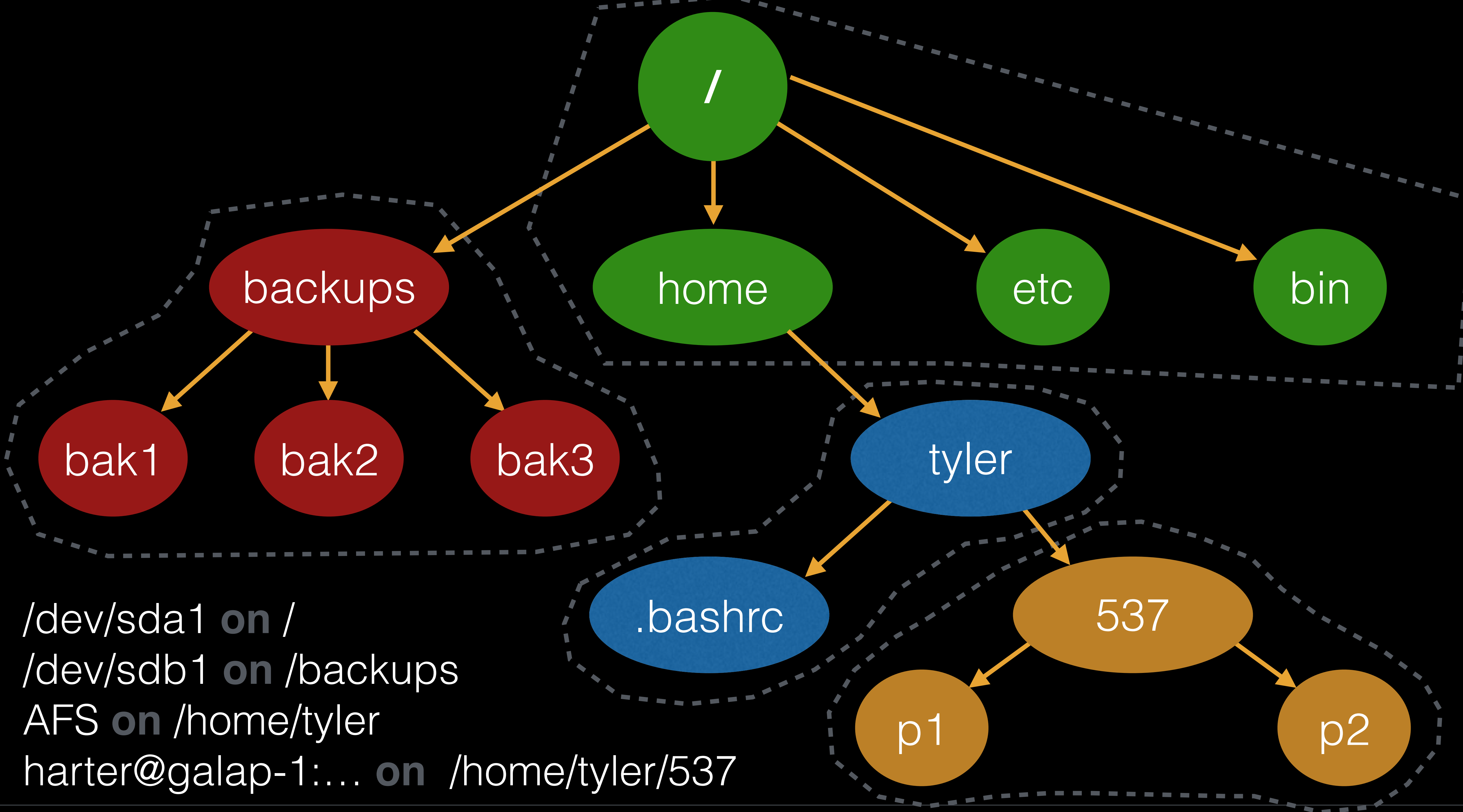
Operations

FS
- mkfs
- mount

File
- create
- write
- open
- read
- close



/dev/sda1 **on** /
/dev/sdb1 **on** /backups
AFS **on** /home/tyler



/dev/sda1 **on** /
/dev/sdb1 **on** /backups
AFS **on** /home/tyler
harter@galap-1:… **on** /home/tyler/537

mount

Add the file system to the FS tree.

Minimally requires reading superblock.

Demo…

Operations

FS

- mkfs
- mount

File

- create
- write
- open
- read
- close

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data

create /foo/bar

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

create /foo/bar

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

create /foo/bar

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

create /foo/bar

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

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		

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			

Operations

FS
- mkfs
- mount

File
- create
- write
- open
- read
- close

write to /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

write to /foo/bar

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

write to /foo/bar

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

write to /foo/bar

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

write to /foo/bar

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

write to /foo/bar

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

Operations

FS
- mkfs
- mount

File
- create
- write
- open
- read
- close

open /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

open /foo/bar

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

open /foo/bar

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

open /foo/bar

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

open /foo/bar

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

open /foo/bar

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

Operations

FS
 - mkfs
 - mount

File
 - create
 - write
 - open
 - read
 - close

read /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

read /foo/bar

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

read /foo/bar

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

Operations

FS
- mkfs
- mount

File
- create
- write
- open
- read
- close

close /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

close /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

nothing to do on disk!

Efficiency

Efficiency

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

Efficiency

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

Cache for:
- reads
- write buffering

Structures

What data is likely to be read frequently?

- superblock
- data block
- data bitmap
- inode table
- inode bitmap
- indirect block
- directories

Unified Page Cache

Instead of a dedicated file-system cache, draw pages from a common pool for FS and processes.

API change:
- read
- shrink_cache (Linux)

LRU Example

Ops	Hits	State
read 1	miss	1
read 2	miss	1,2
read 3	miss	1,2,3
read 4	miss	1,2,3,4
shrink	-	2,3,4
shrink	-	3,4
read 1	miss	1,3,4
read 2	miss	1,2,3,4
read 3	hit	1,2,3,4
read 4	hit	1,2,3,4

Write Buffering

Why does procrastination help?

Write Buffering

Why does procrastination help?

Overwrites, deletes, scheduling.

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

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…
 - tradeoffs?

Summary/Future

We've described a very simple FS.
 - basic on-disk structures
 - the basic ops

Future questions:
 - how to allocate efficiently?
 - how to handle crashes?

[537] Fast File System

Chapter 41
Tyler Harter
11/10/14

Review Basic FS

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data

create /foo/bar

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

[traverse]

create /foo/bar

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

[traverse]

create /foo/bar

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

bar does not already exist

create /foo/bar

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

[allocate inode]

create /foo/bar

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

[populate inode]

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		

[add bar to /foo]

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

append to /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

[append? yes]

append to /foo/bar

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

[allocate block]

append to /foo/bar

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

append to /foo/bar

[point to block]

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

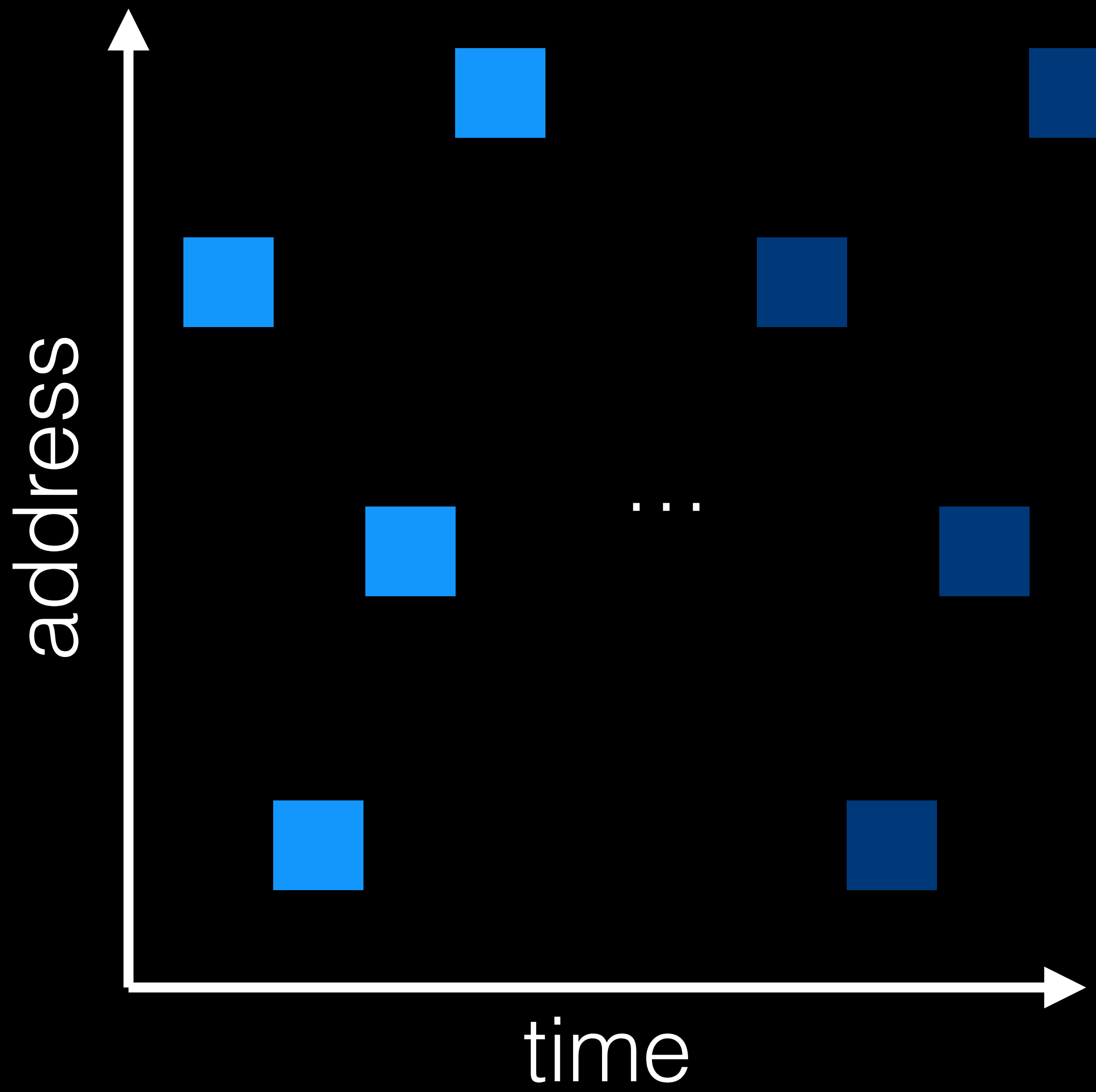
[write to block]

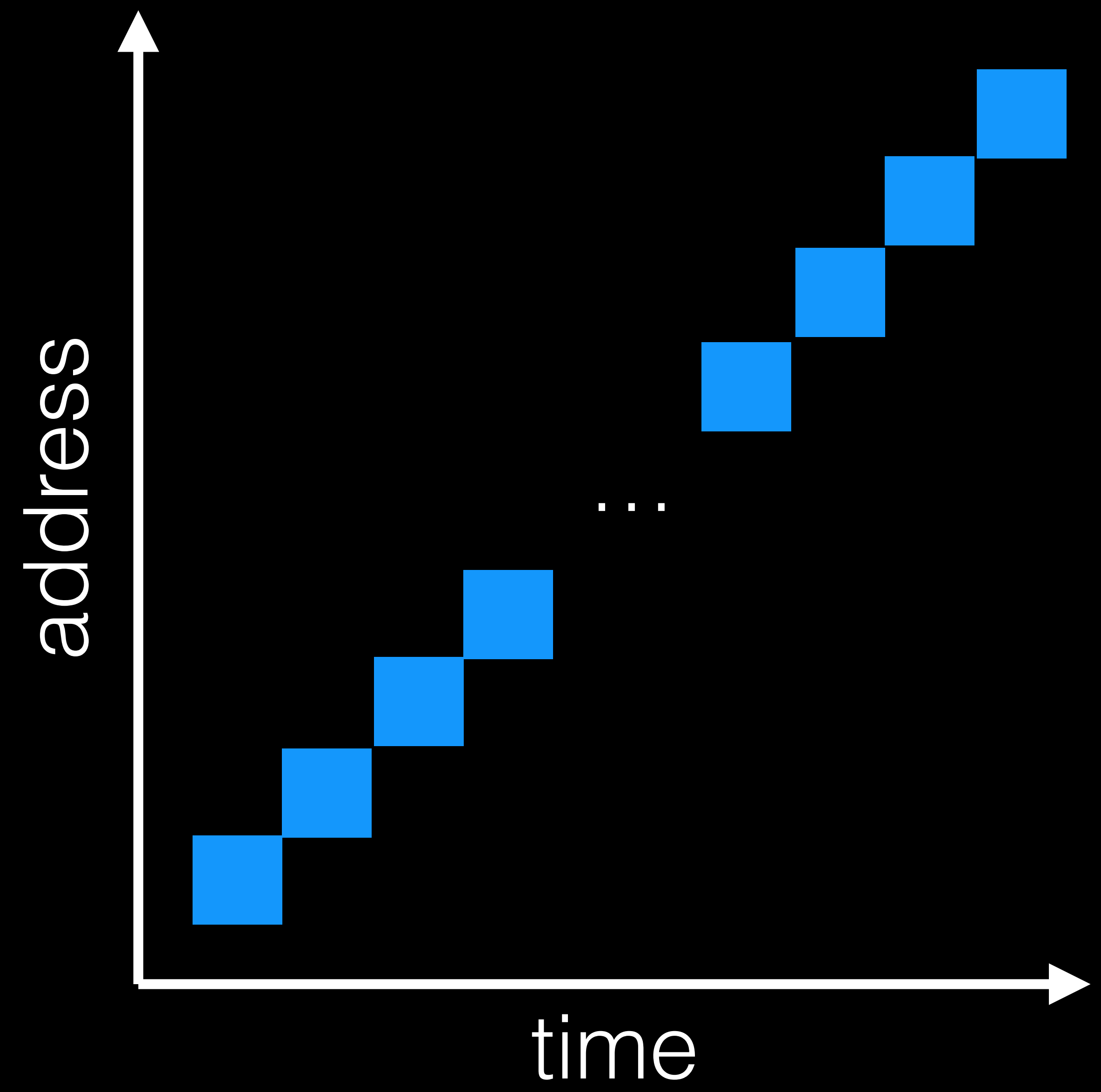
append to /foo/bar

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

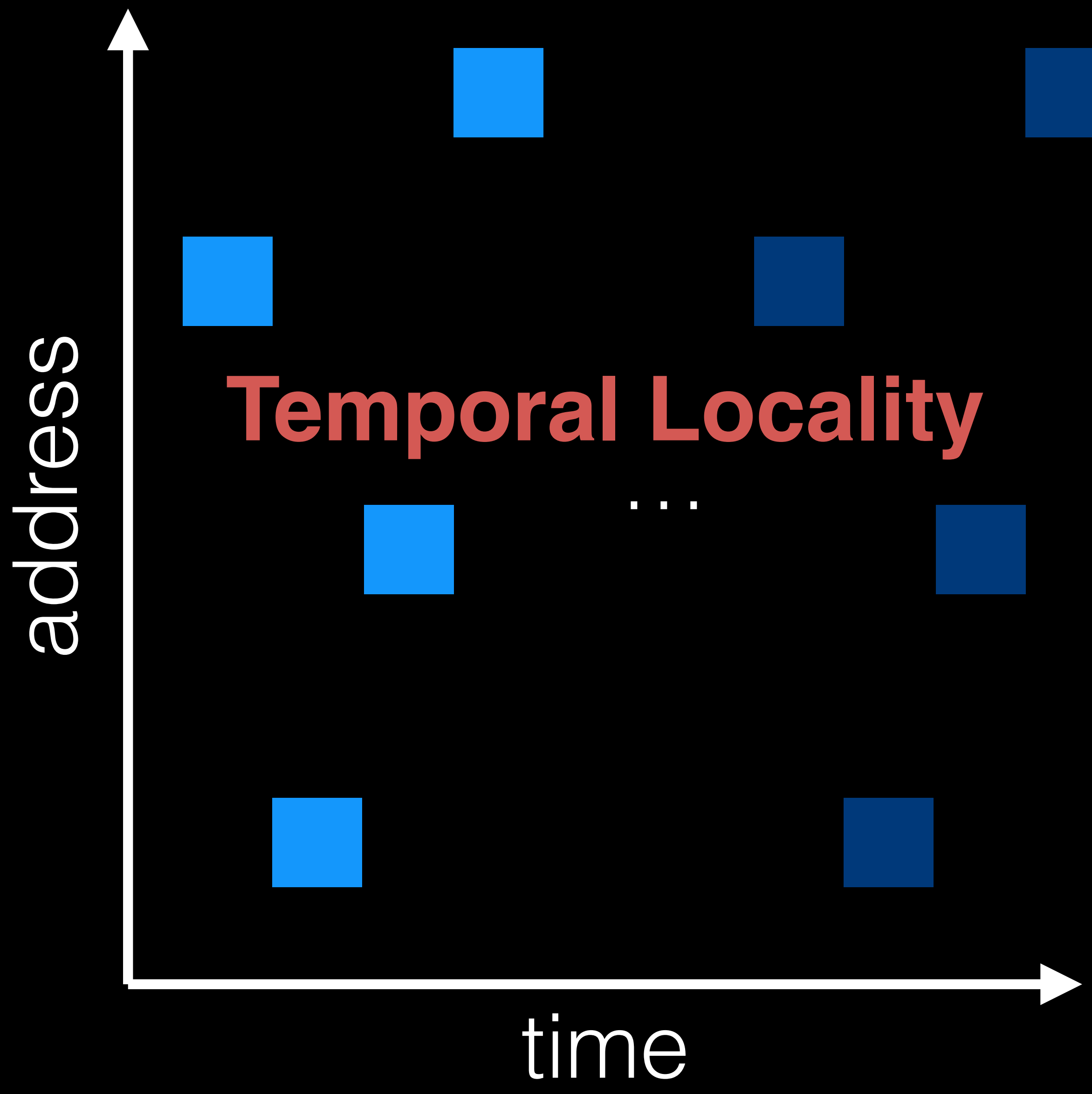
Review Locality

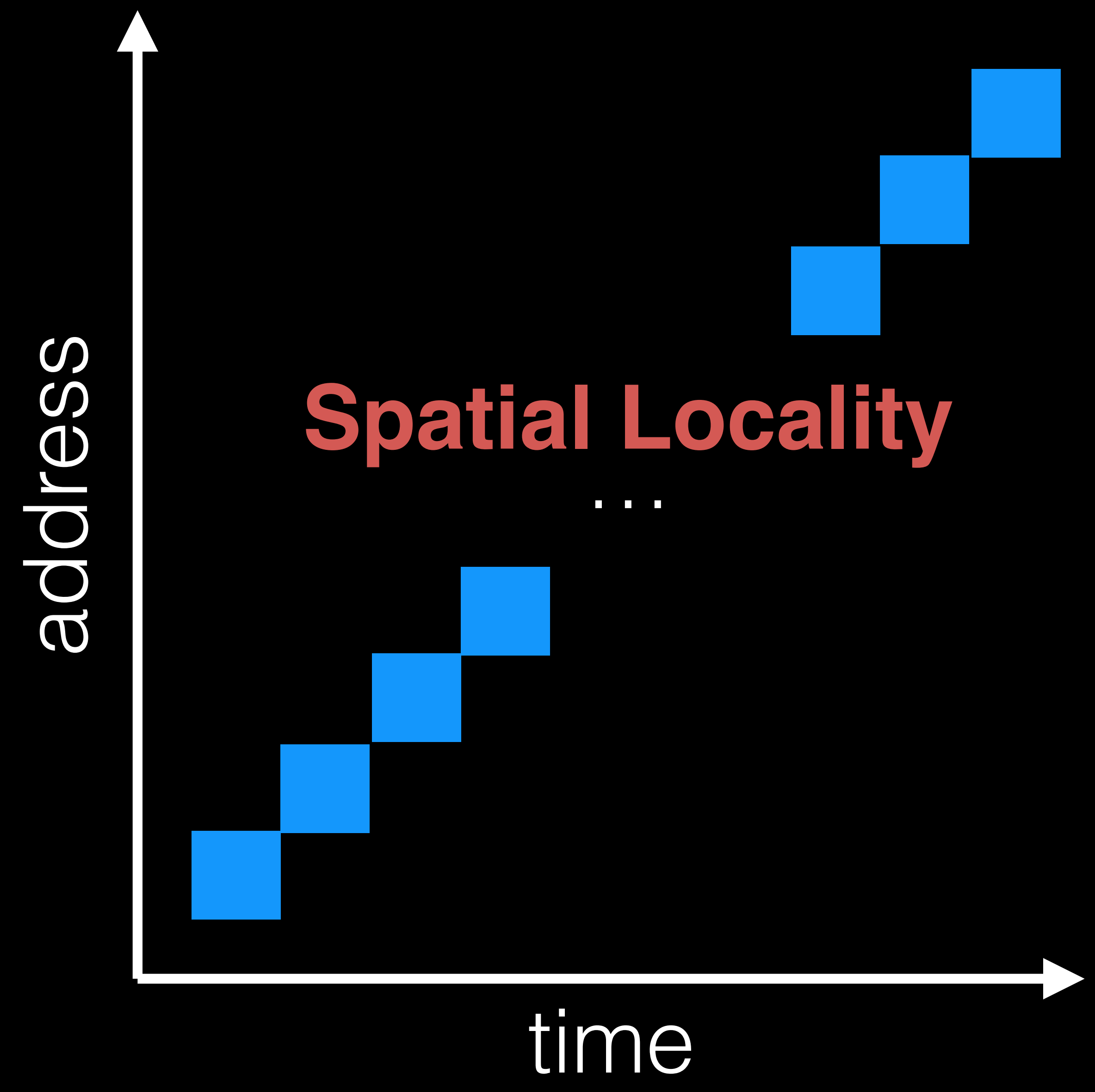
Locality Types





Locality Types



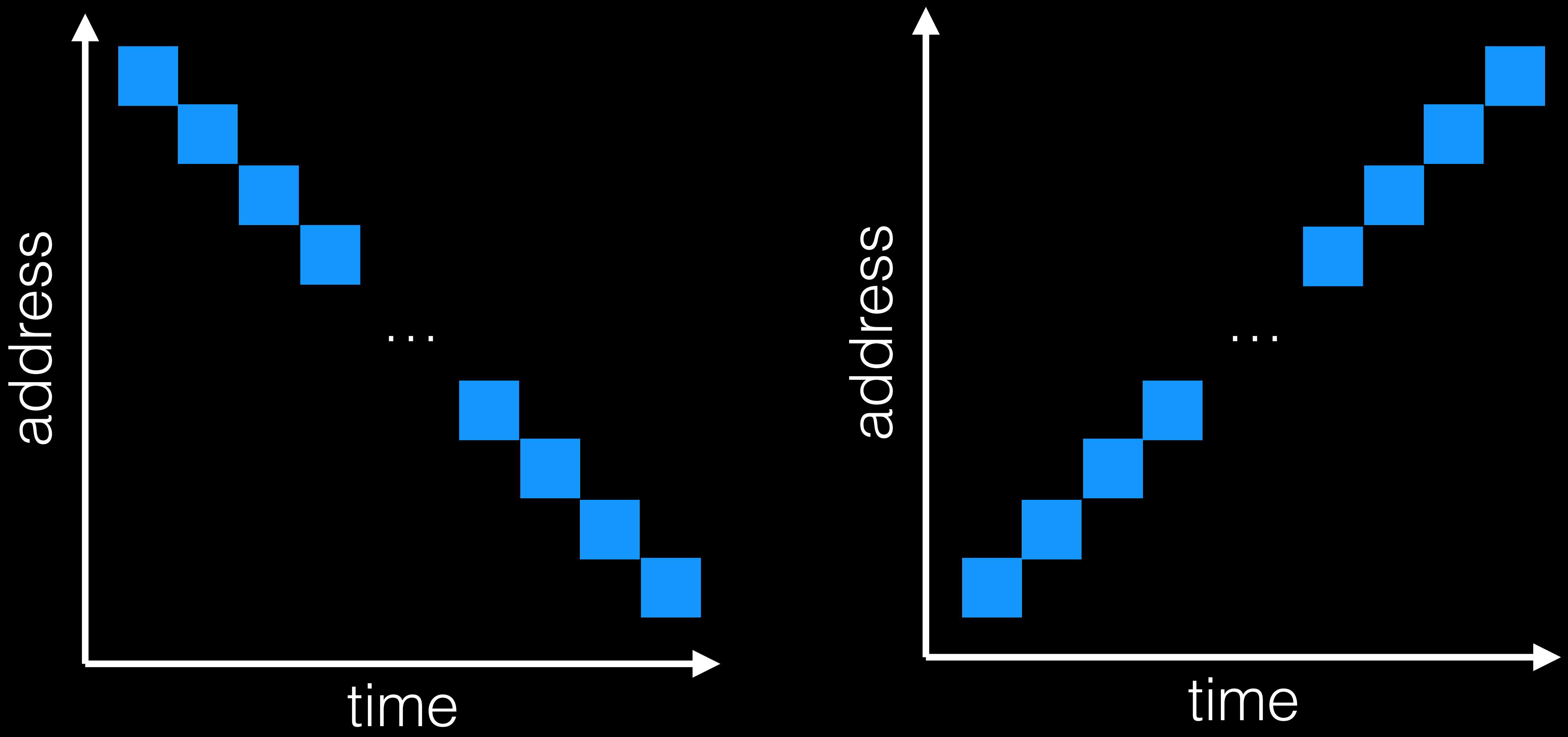


Locality Usefulness

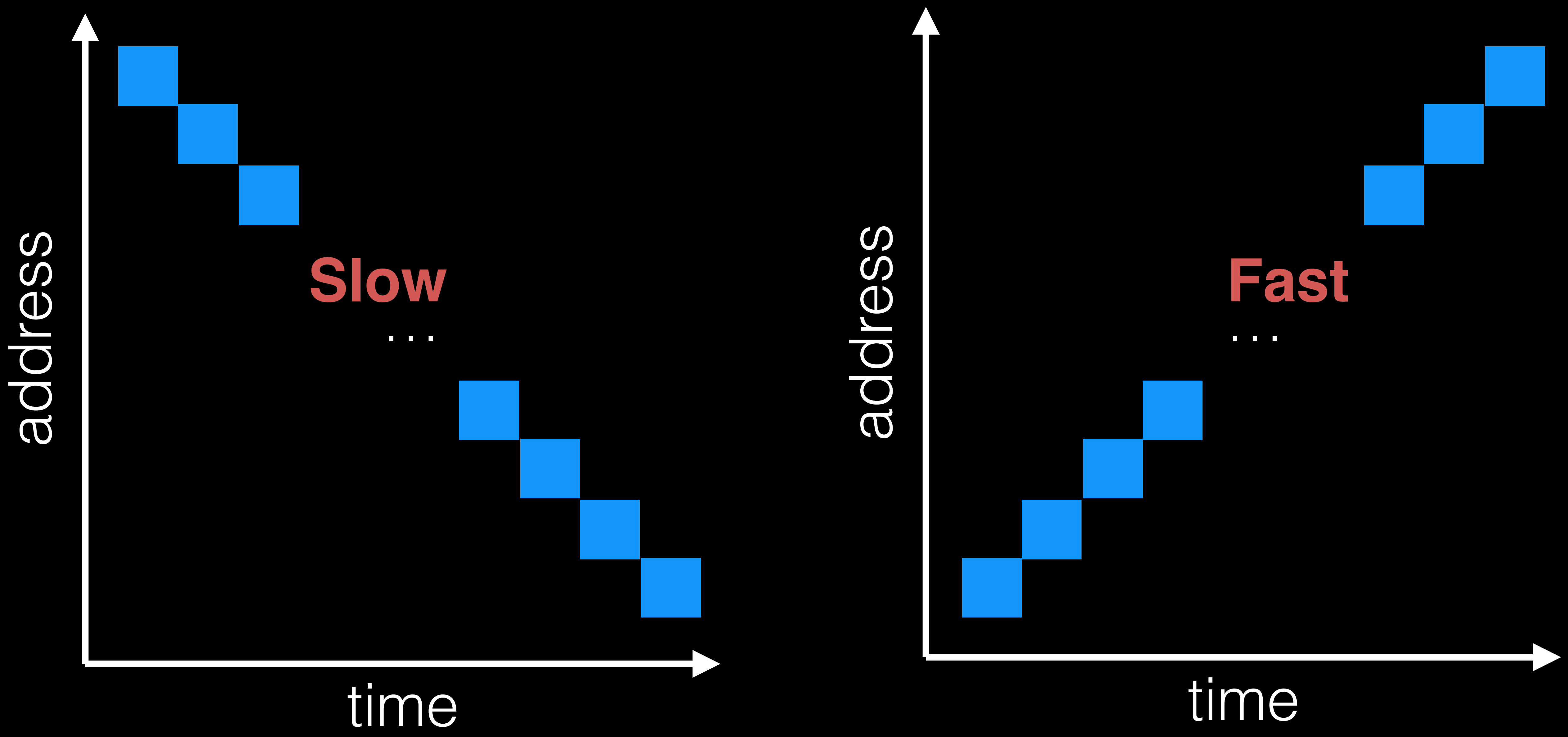
What types of locality are useful for a cache?

What types of locality are useful for a disk?

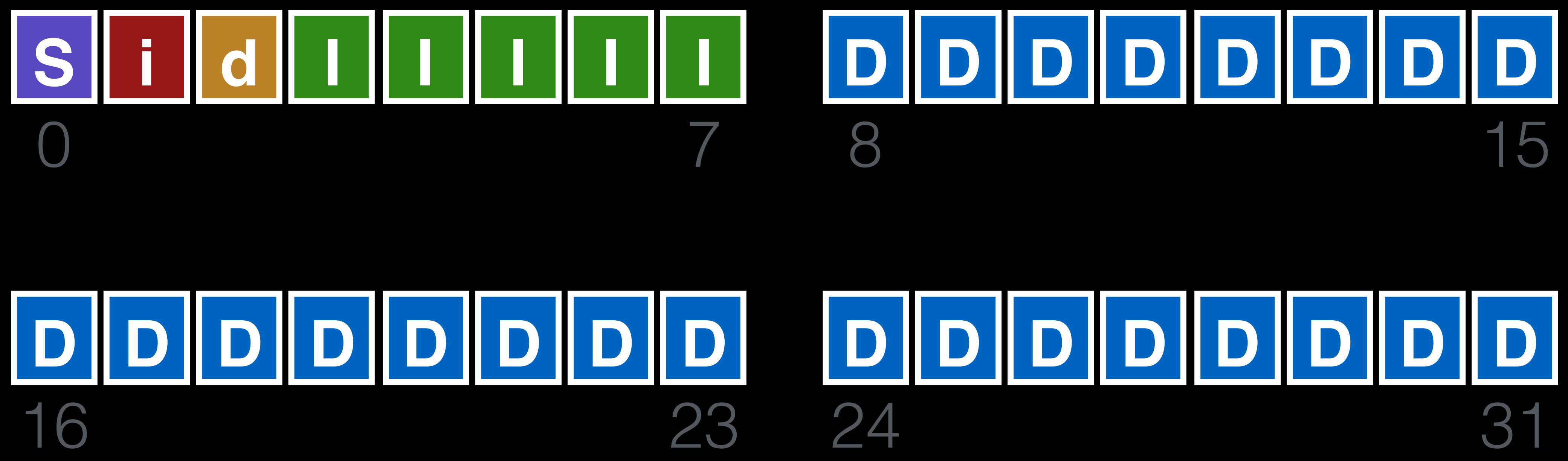
Order Matters Now



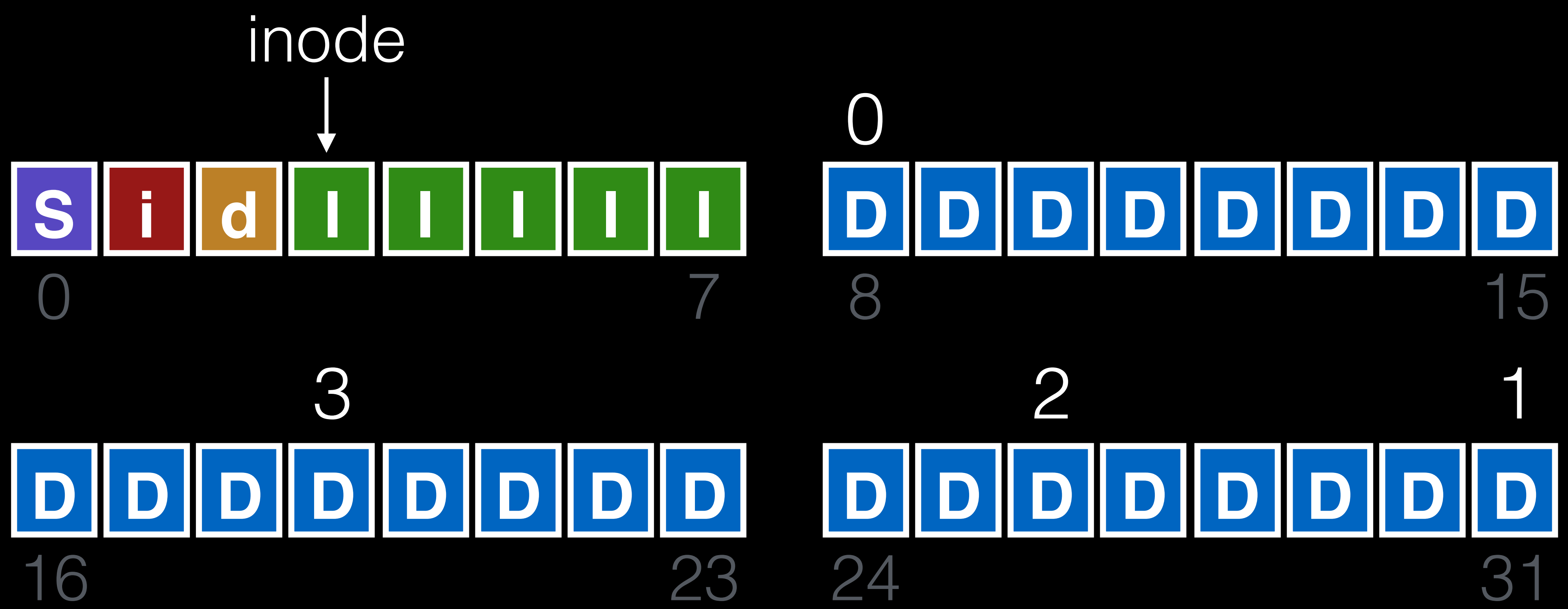
Order Matters Now



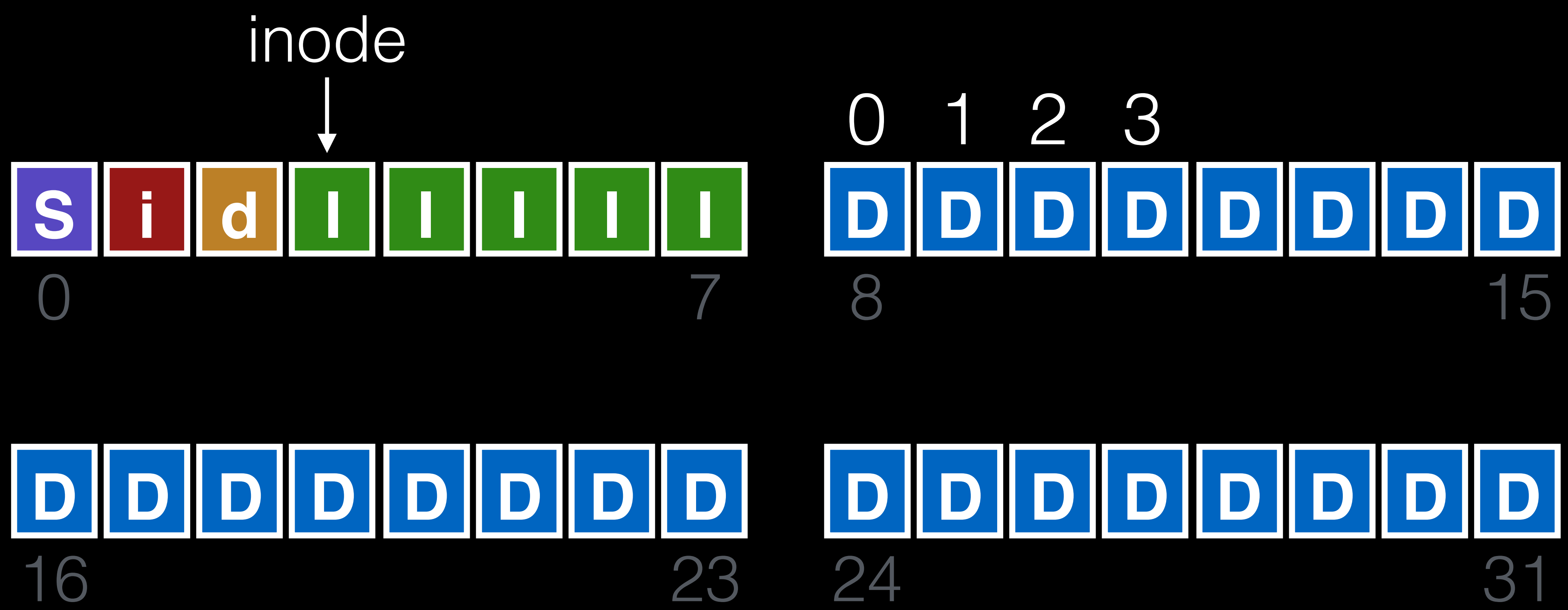
Policy: Choose Inode, Data Blocks



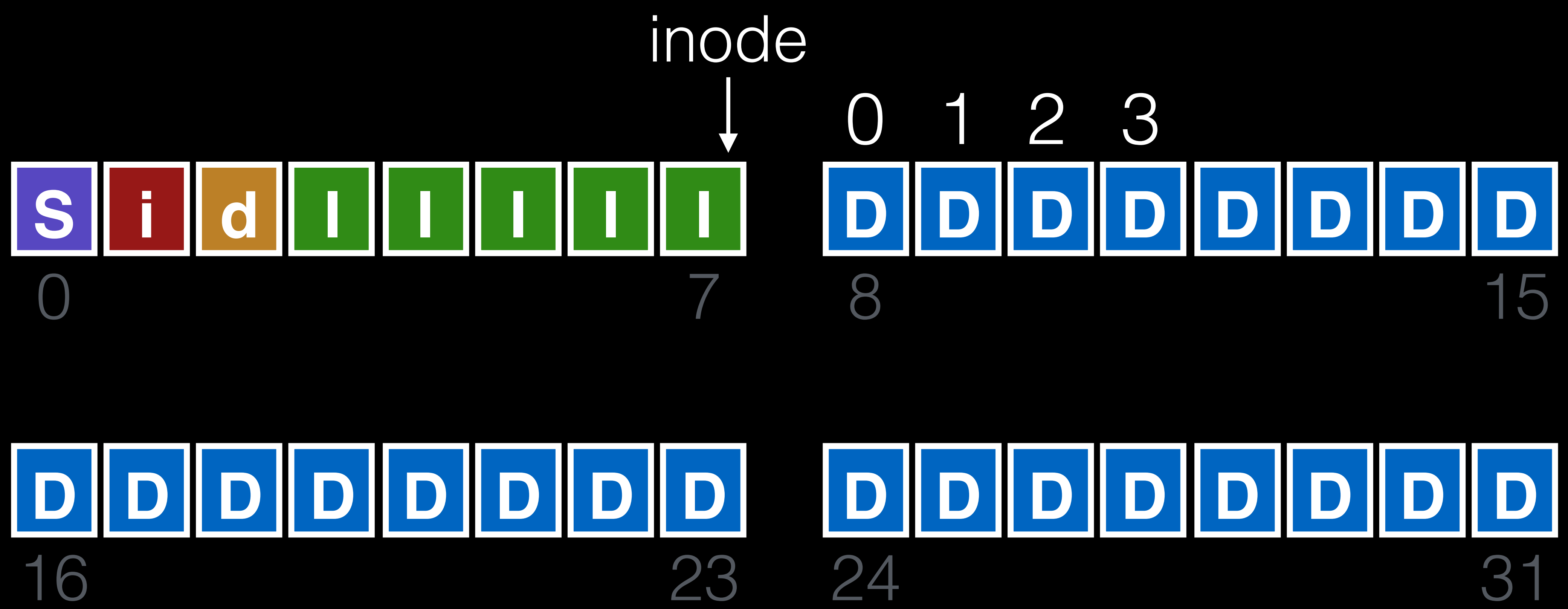
Bad File Layout



Better File Layout



Best File Layout



Fast File System

System Building

noob approach

1. get idea
2. build it!

System Building

noob approach

1. get idea
2. build it!

pro approach

1. identify state of the art
2. measure it, identify problems
3. get idea
4. build it!

System Building

noob approach

1. get idea
2. build it!

pro approach

1. identify state of the art
2. measure it, identify problems
3. get idea
4. build it!

measure then build

Old FS

State of the art: original UNIX file system.

Layout

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

Old FS

State of the art: original UNIX file system.

Old FS

State of the art: original UNIX file system.

Measure throughput for file reads/writes.

Compare to theoretical max, which is…

Old FS

State of the art: original UNIX file system.

Measure throughput for file reads/writes.

Compare to theoretical max, which is…
disk bandwidth

Old FS

State of the art: original UNIX file system.

Measure throughput for file reads/writes.

Compare to theoretical max, which is…
disk bandwidth

Old UNIX file system: only **2%** of potential. Why?

Measurement 1

What is performance before/after aging?

Measurement 1

What is performance before/after aging?

New FS: **17.5%** of disk bandwidth
Few weeks old: **3%** of disk bandwidth

Measurement 1

What is performance before/after aging?

New FS: **17.5%** of disk bandwidth
Few weeks old: **3%** of disk bandwidth

FS is probably becoming fragmented over time.

Free list makes contiguous chunks hard to find.

Measurement 1

What is performance before/after aging?

New FS: **17.5%** of disk bandwidth
Few weeks old: **3%** of disk bandwidth

hacky solution:
occasional defrag

FS is probably becoming fragmented over time.

Free list makes contiguous chunks hard to find.

Measurement 2

How does block size affect performance?
Try doubling it!

Measurement 2

How does block size affect performance?
Try doubling it!

Performance **more** than doubled.

Measurement 2

How does block size affect performance?
Try doubling it!

Performance **more** than doubled.

Measurement 2

How does block size affect performance?
Try doubling it!

Performance **more** than doubled.

Logically adjacent blocks are probably not physically adjacent.

Measurement 2

How does block size affect performance?
Try doubling it!

Performance **more** than doubled.

Logically adjacent blocks are probably not physically adjacent.

Measurement 2

How does block size affect performance?
Try doubling it!

Performance **more** than doubled.

Logically adjacent blocks are probably not physically adjacent.

Smaller blocks cause more indirect I/O.

Old FS Summary

Observations:
- long distance between inodes/data
- inodes in single dir not close to one another
- small blocks (512 bytes)
- blocks laid out poorly
- free list becomes scrambled, causes random alloc

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

Problem: old FS treats disk like RAM!

Solution: a disk-aware FS

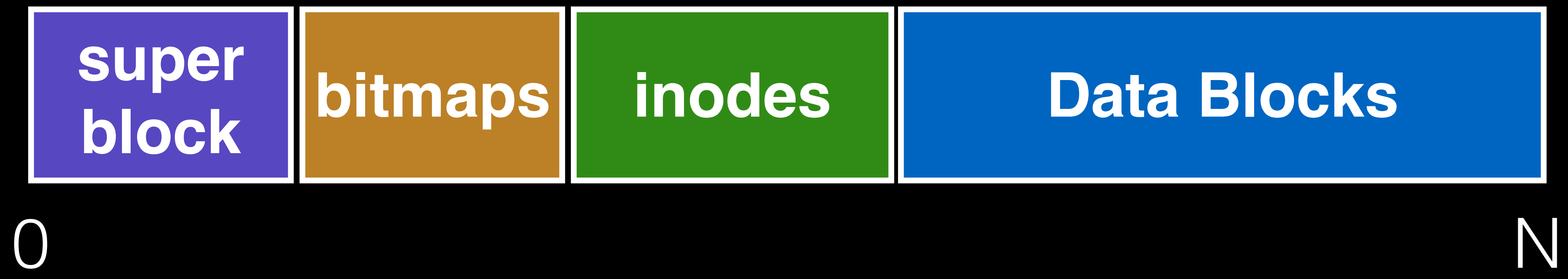
Design Questions

How to use big blocks without wasting space.

How to place data on disk.

Technique 1: Bitmaps

Technique 1: Bitmaps

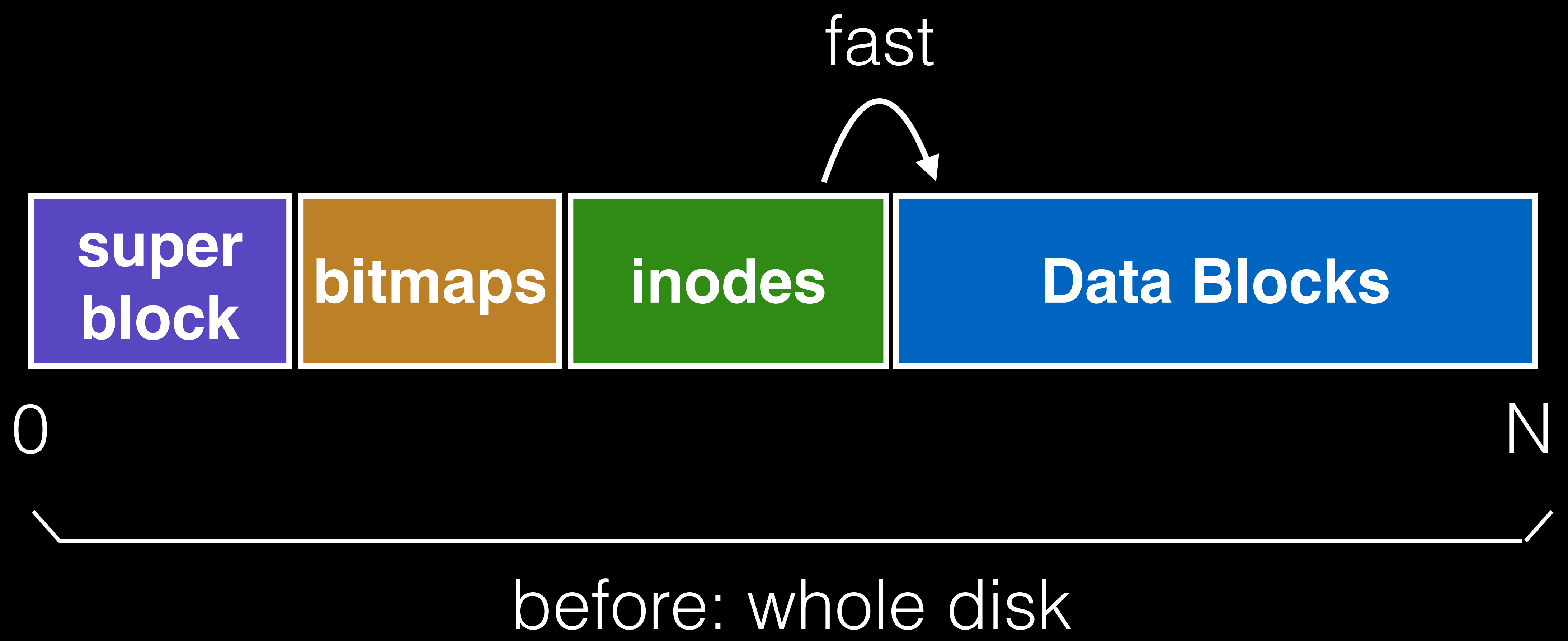


Use bitmaps instead of free list.
Provides more flexibility, with more global view.

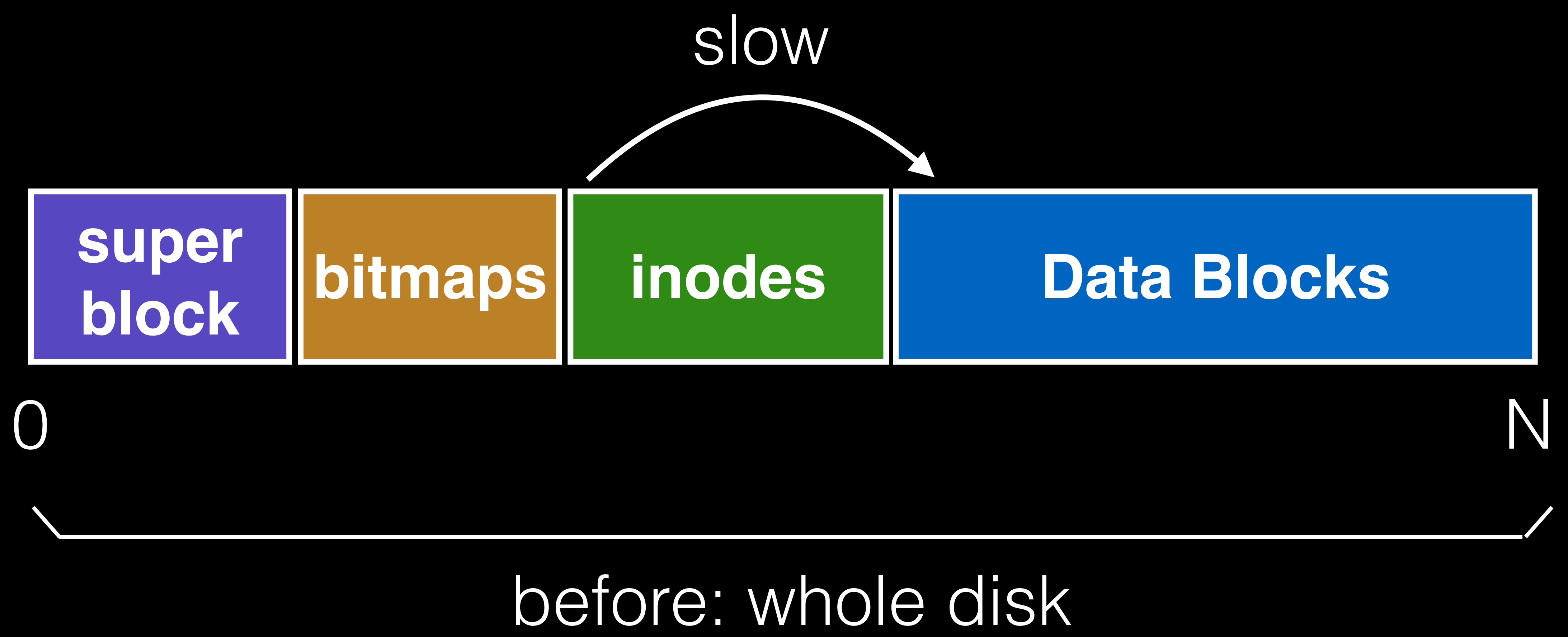
Techniques

Bitmaps

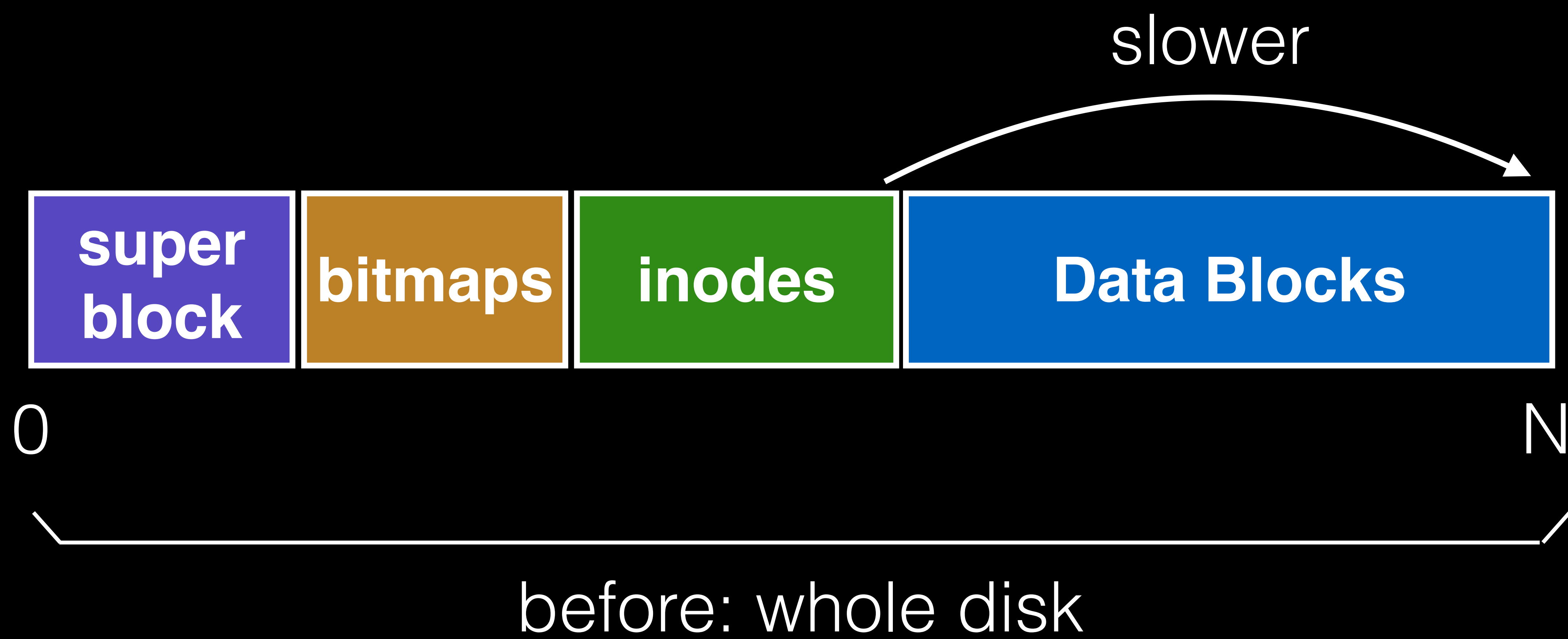
Technique 2: Groups



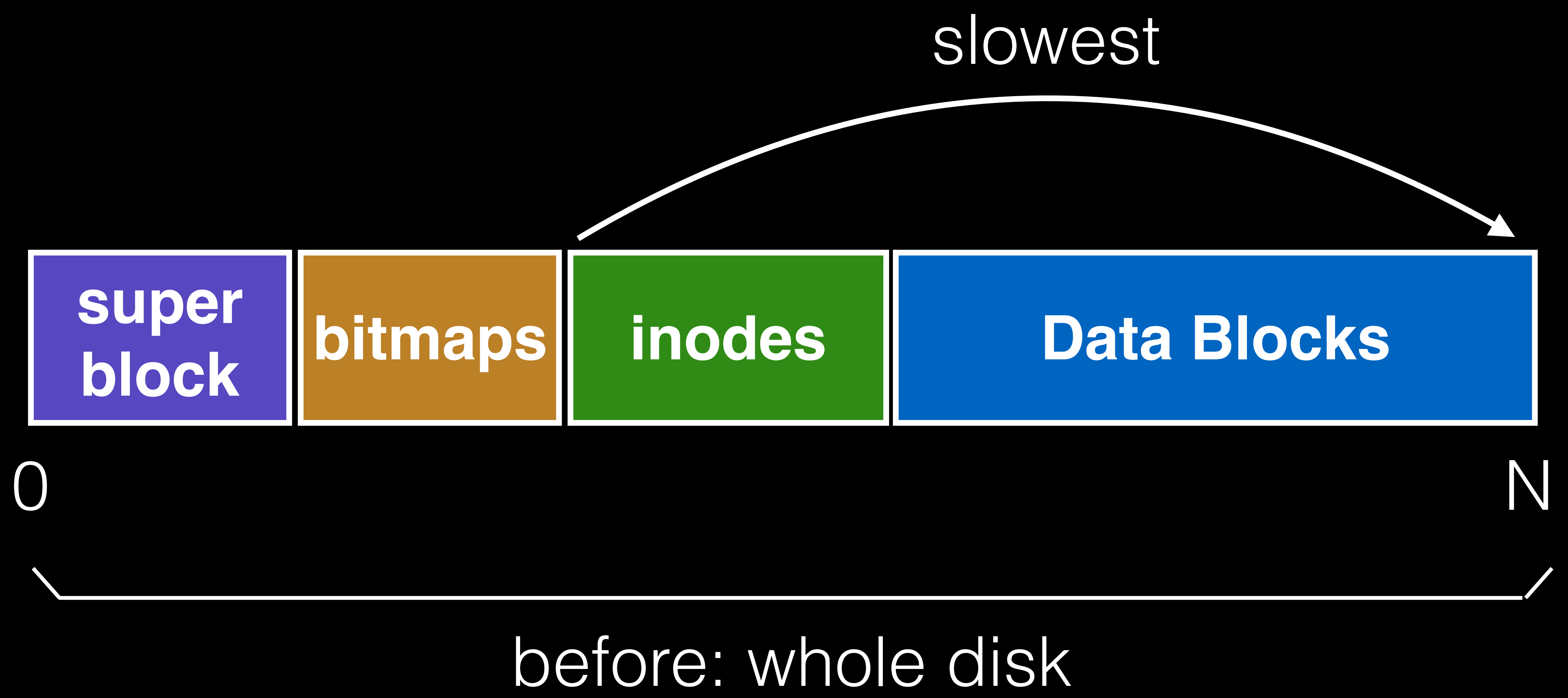
Technique 2: Groups



Technique 2: Groups

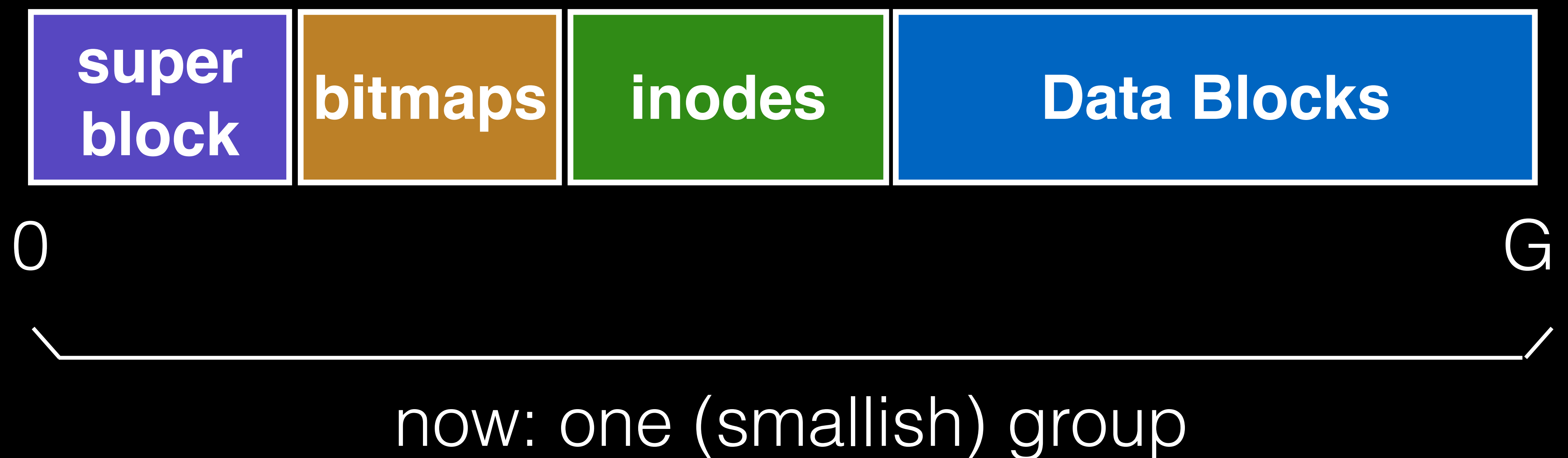


Technique 2: Groups



Technique 2: Groups

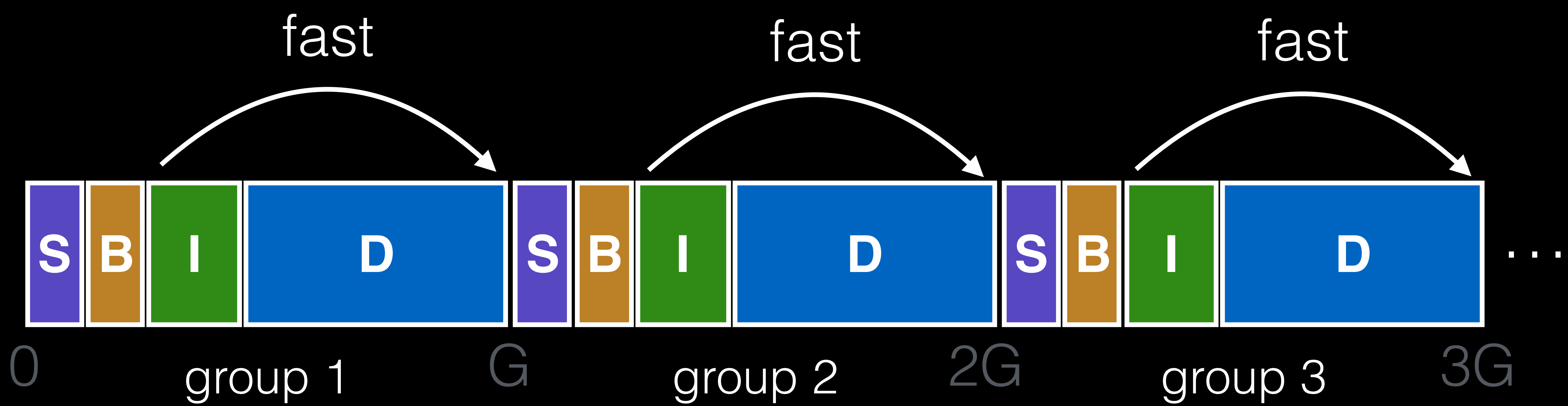
Technique 2: Groups



Technique 2: Groups

zoom out

Technique 2: Groups

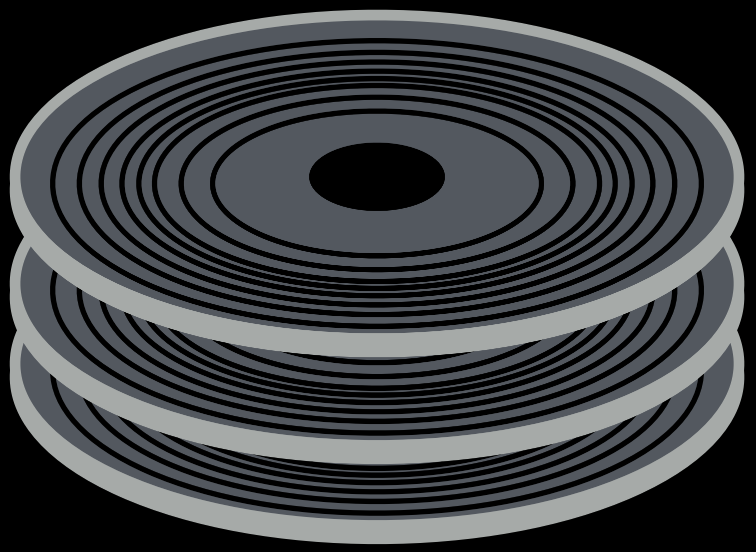


strategy: allocate inodes and data blocks in same group.

Groups

In FFS, groups were ranges of cylinders
- called cylinder group

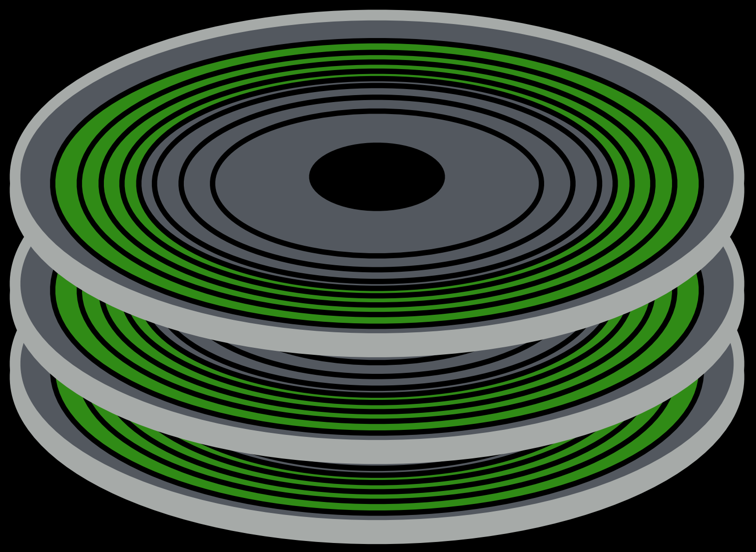
In ext2-4, groups are ranges of blocks
- called block group



Groups

In FFS, groups were ranges of cylinders
- called cylinder group

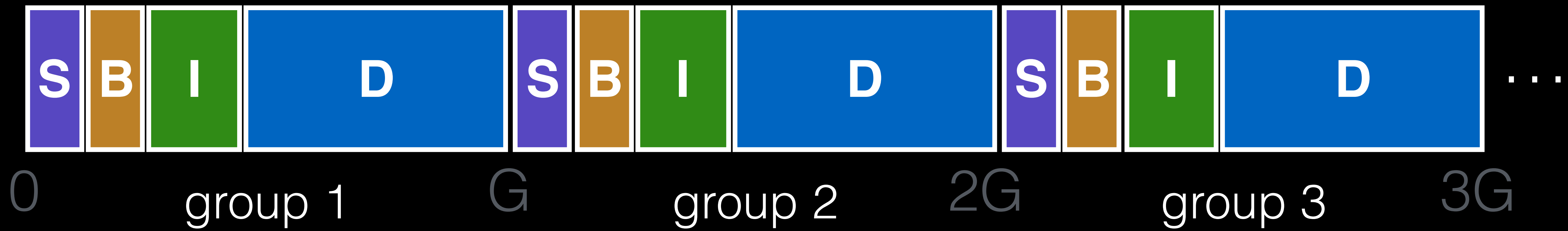
In ext2-4, groups are ranges of blocks
- called block group



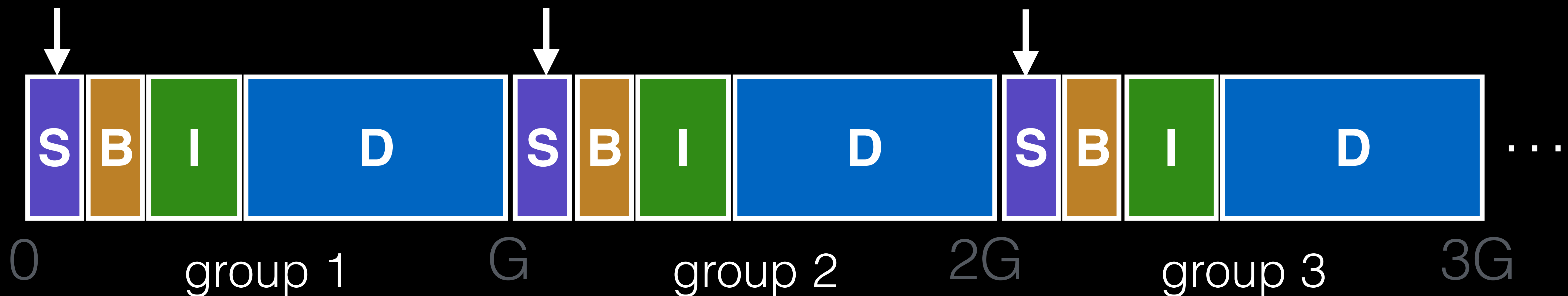
Techniques

Bitmaps
Locality groups

Technique 3: Super Rotation

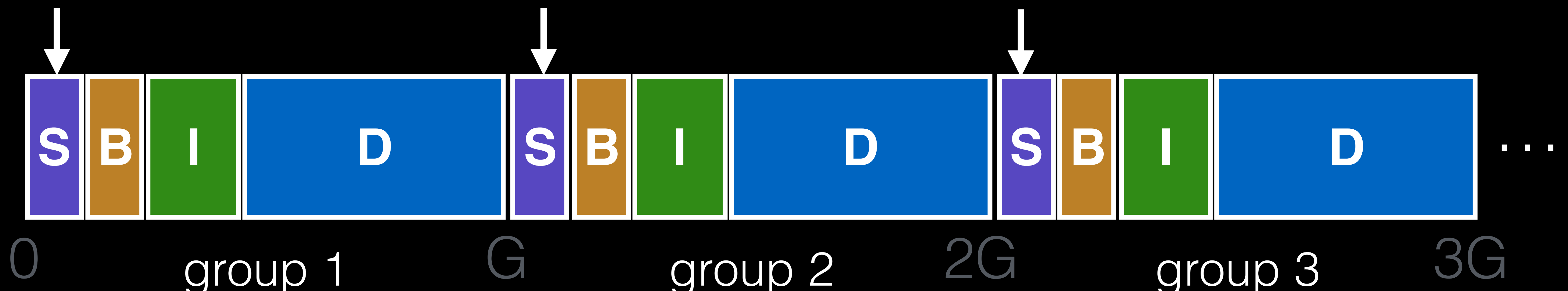


Technique 3: Super Rotation



Is it useful to have multiple super blocks?

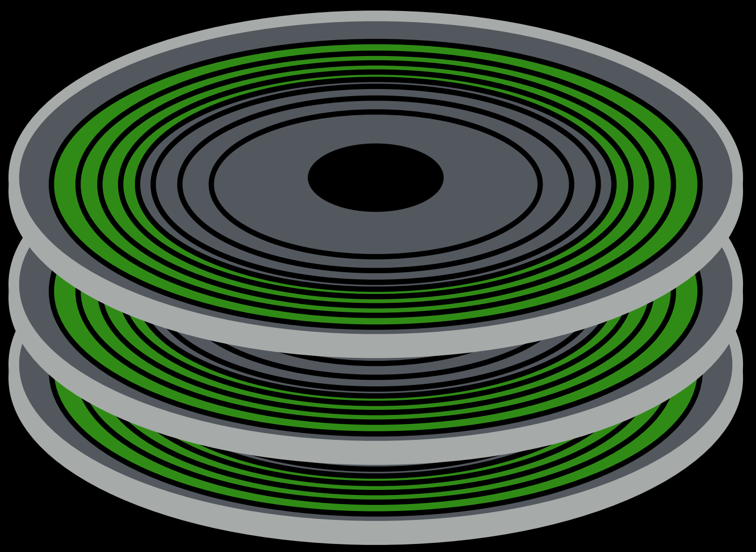
Technique 3: Super Rotation



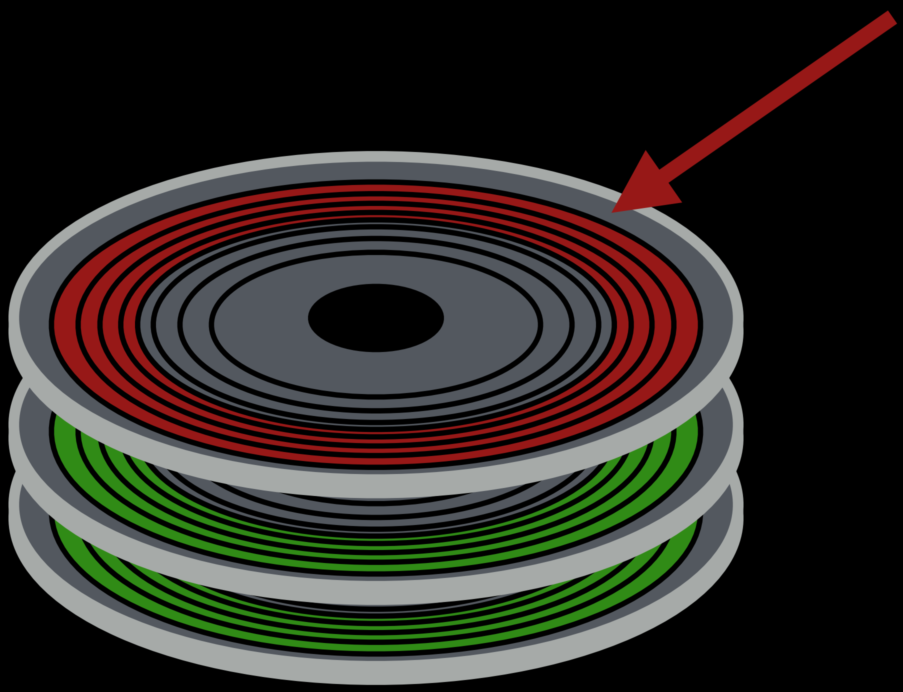
Is it useful to have multiple super blocks?

Yes, if some (but not all) fail.

Problem

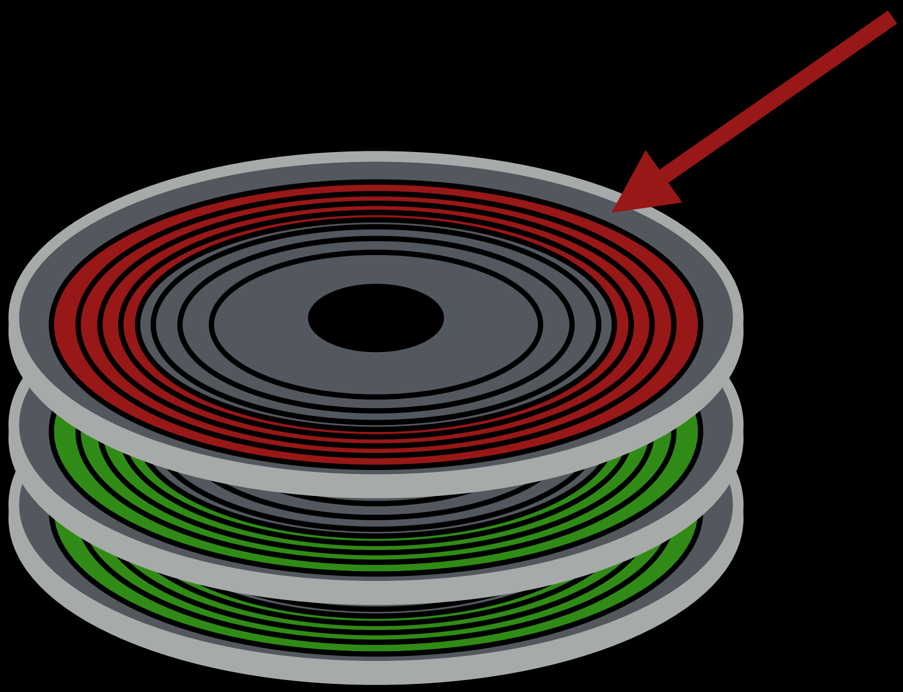


Problem



All super-block copies are on the top platter. What if it dies?

Problem



All super-block copies are on the top platter. What if it dies?

solution: for each group, store super-block at different offset.

Techniques

Bitmaps
Locality groups
Rotated super

Block Size

Doubling the block size for the old FS over doubled performance.

Strategy: choose block size so we never have to read more than two indirect blocks to find a data block (2 levels of indirection max). Want 4GB files.

How large is this?

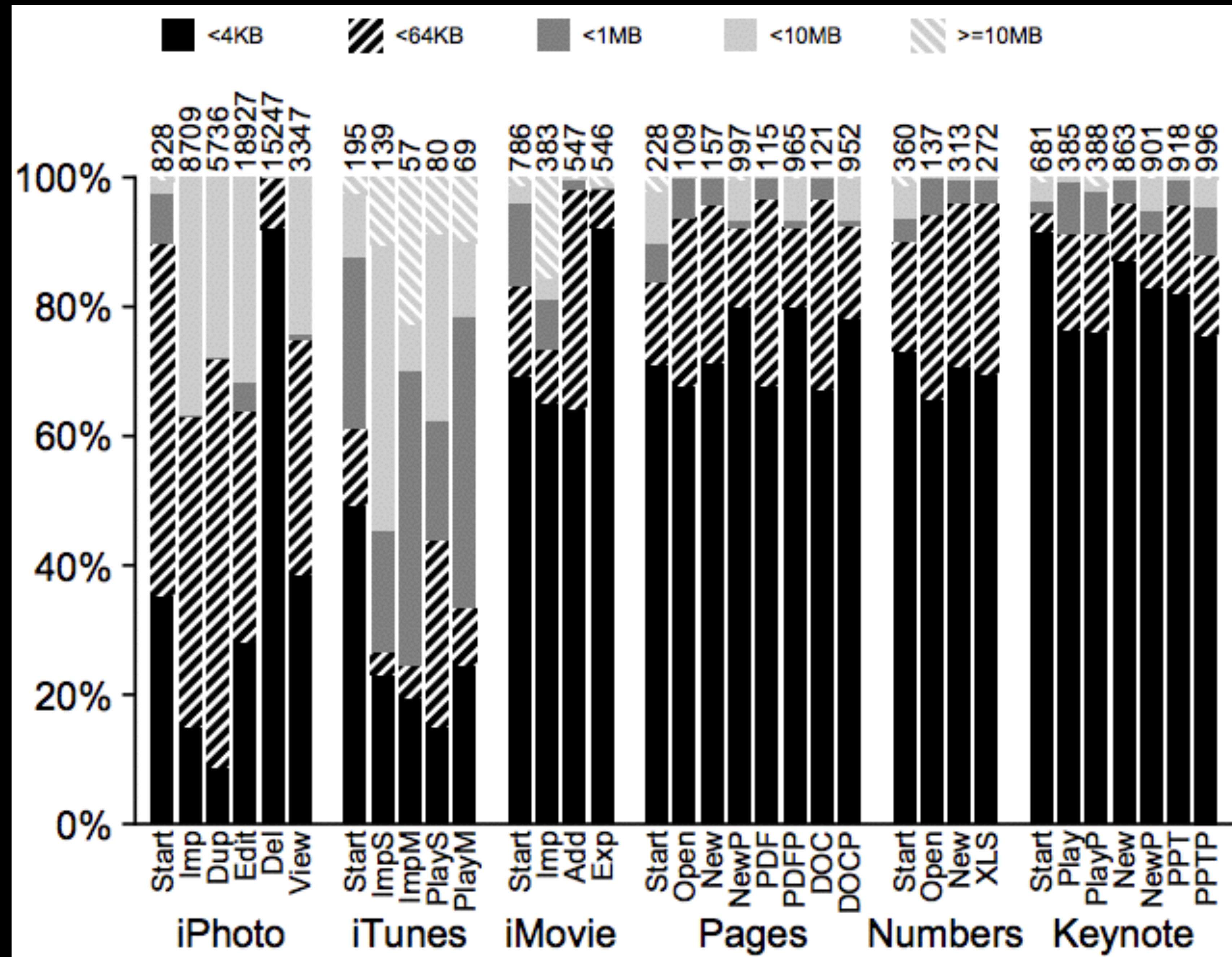
Techniques

Bitmaps
Locality groups
Rotated super
Large blocks

Large Blocks

Why not make blocks huge?

Most file are very small.

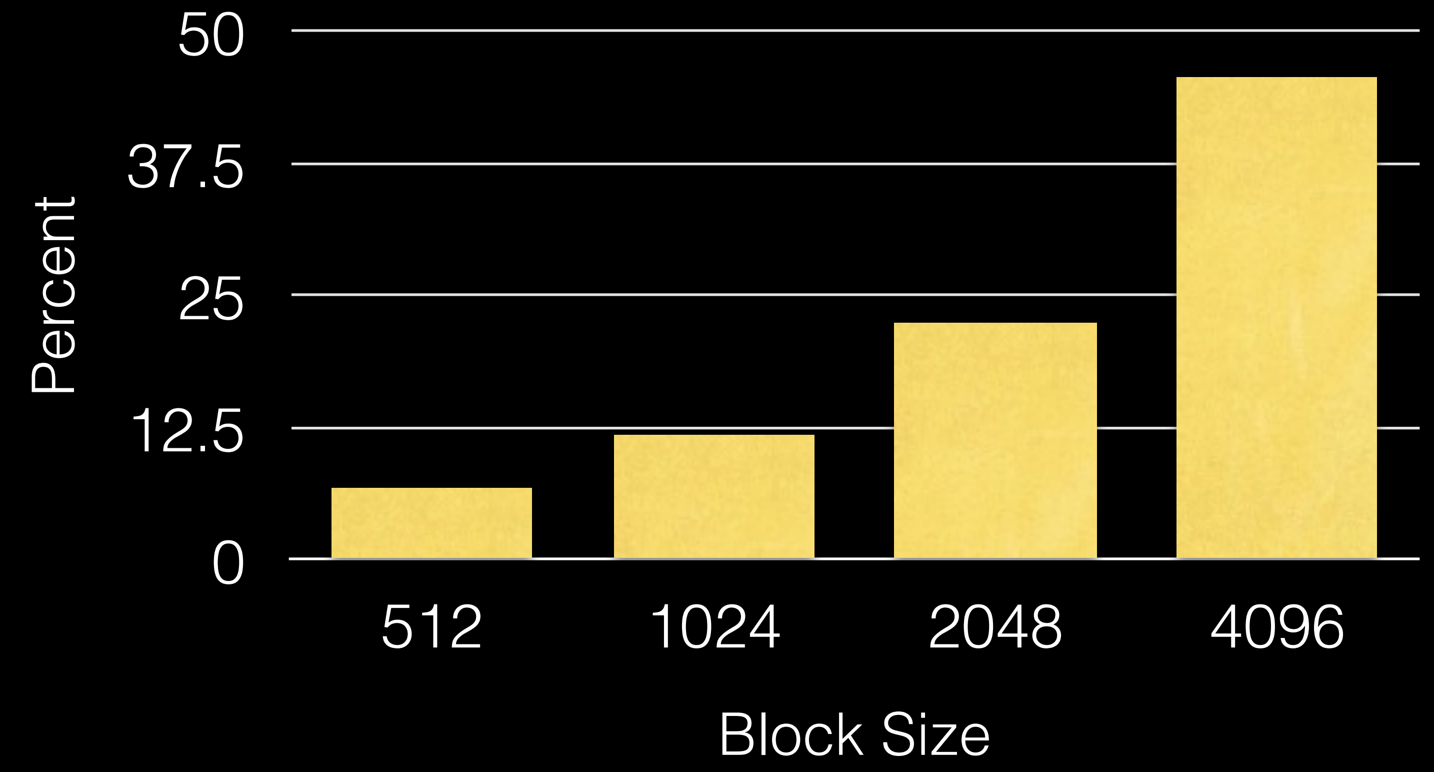


Large Blocks

Why not make blocks huge?
Lots of waste in remainder of blocks.

Large Blocks

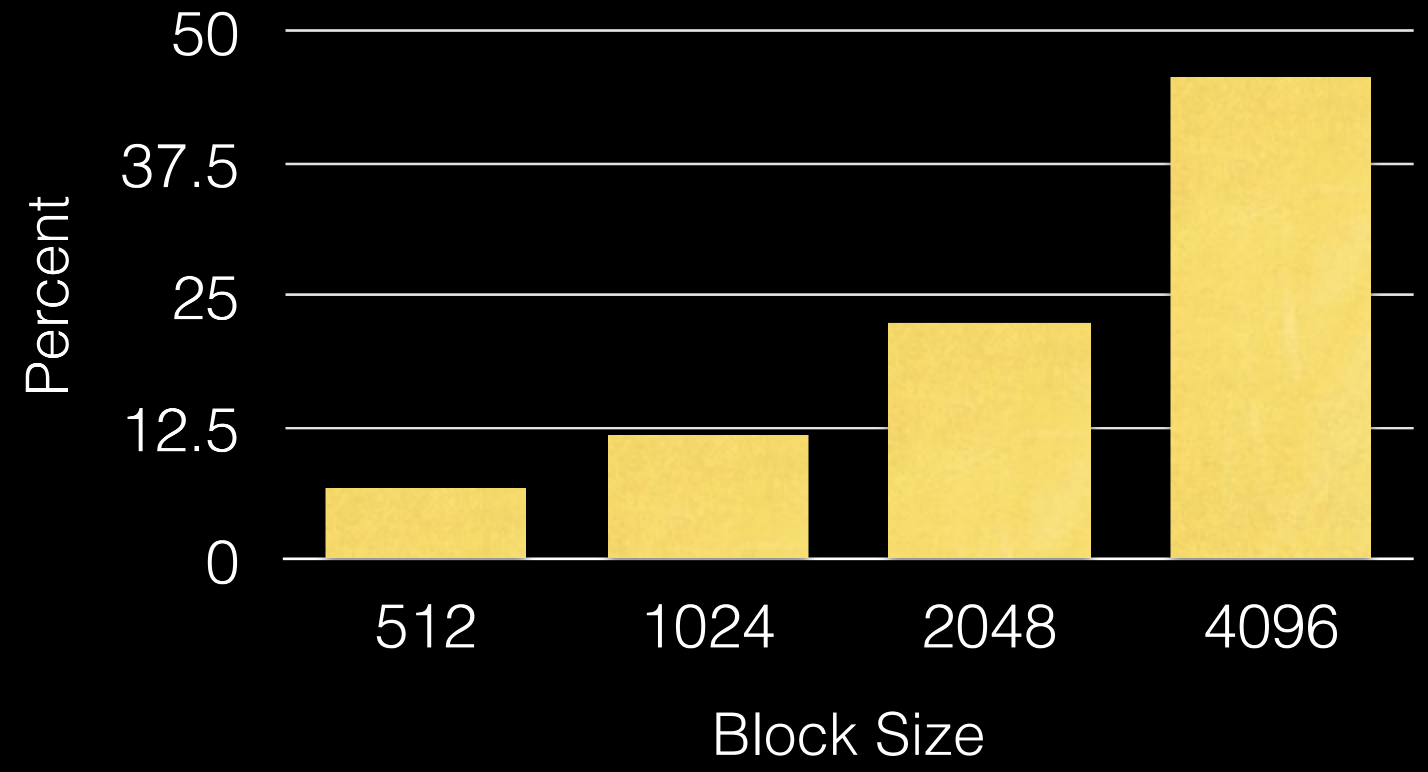
Why not make blocks huge?
Lots of waste in remainder of blocks.



Large Blocks

Why not make blocks huge?
Lots of waste in remainder of blocks.

Time vs. Space
Tradeoffs…



Solution: Fragments

Hybrid!

Introduce “fragment” for files that use parts of blocks.

Only tail of file uses fragments.

Techniques

Bitmaps
Locality groups
Rotated super
Large blocks
Fragments

Smart Policy

Where should new inodes and data blocks go?

Strategy

Put related pieces of data near each other.

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.

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.

Sound good?

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.

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.

Revised Strategy

Put more-related pieces of data near each other.

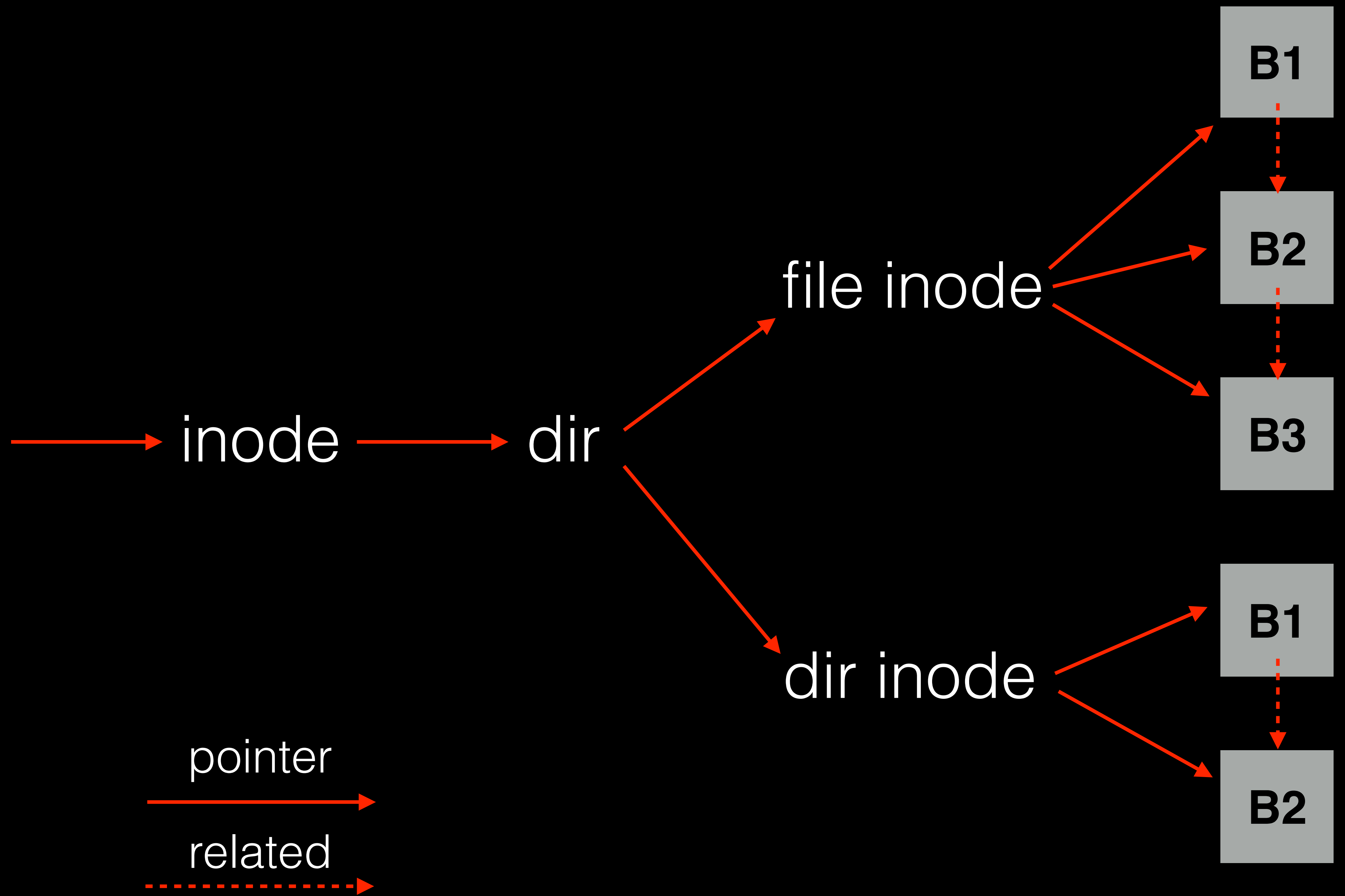
Put less-related pieces of data far from each other.

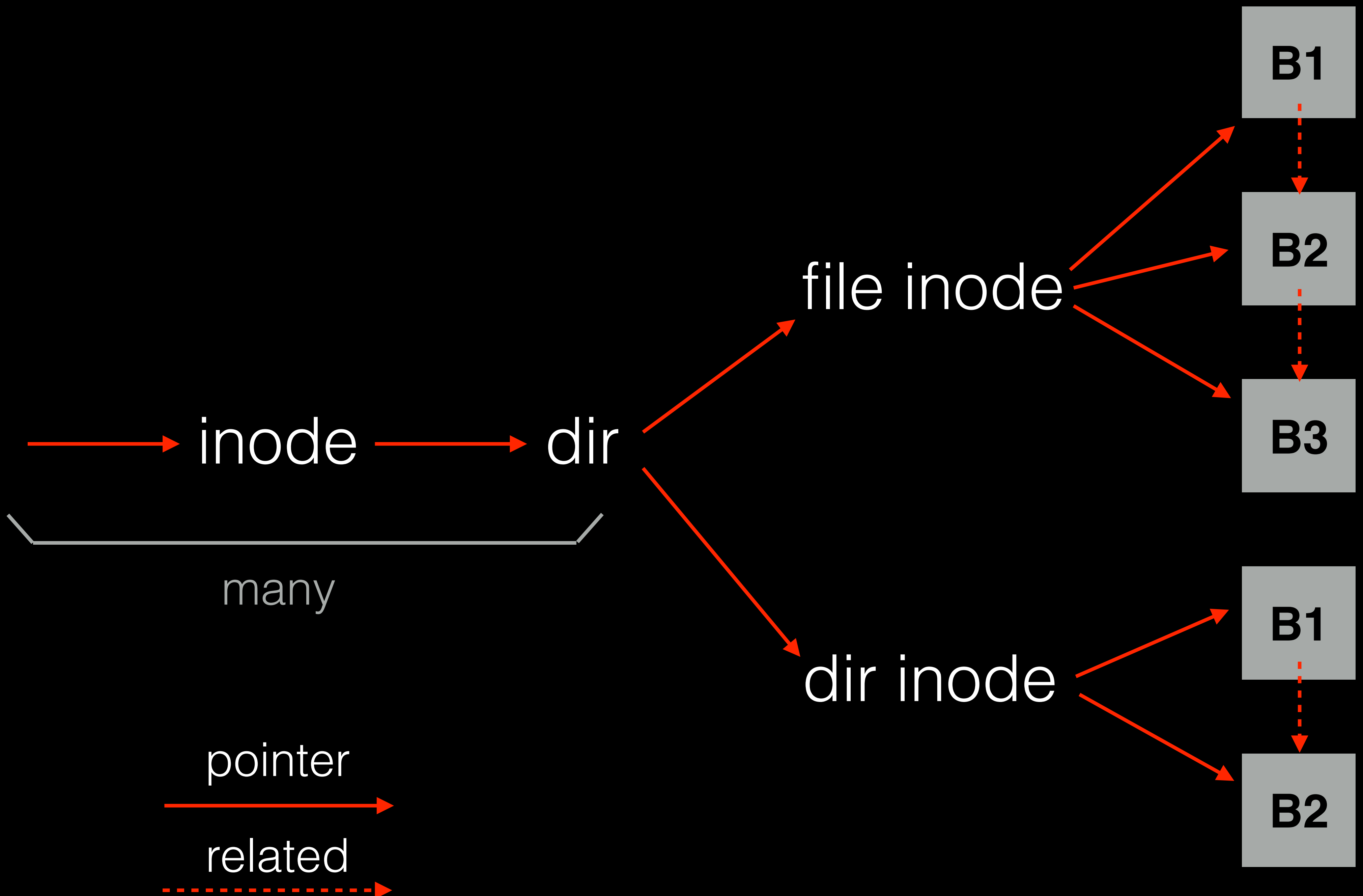
FFS developers used their best judgement.

FFS: Two-Level Allocator

Level 1: decide which group

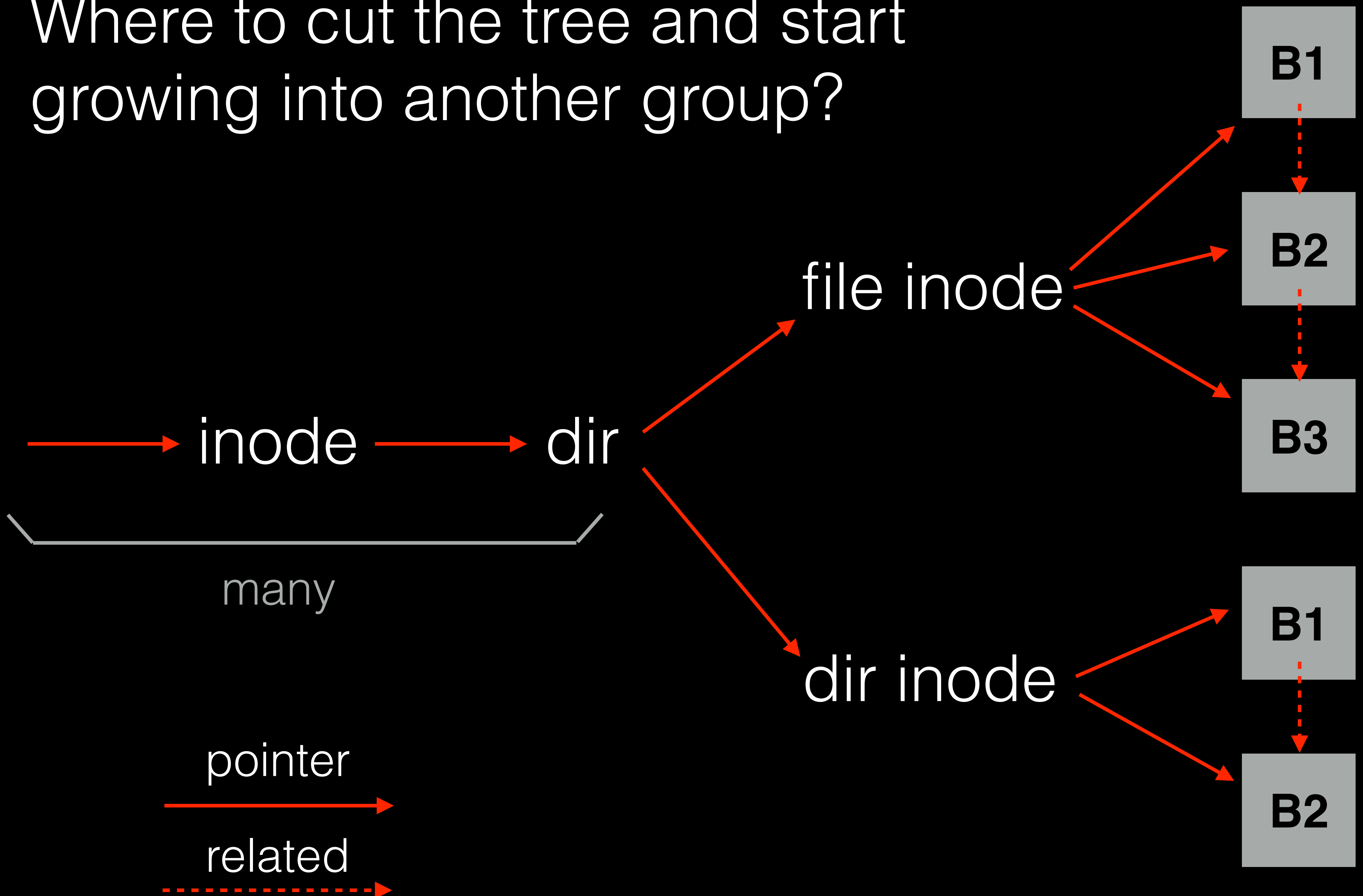
Level 2: decide where in group


inode
dir
file inode
dir inode
B1
B2
B3
B1
B2
pointer
related

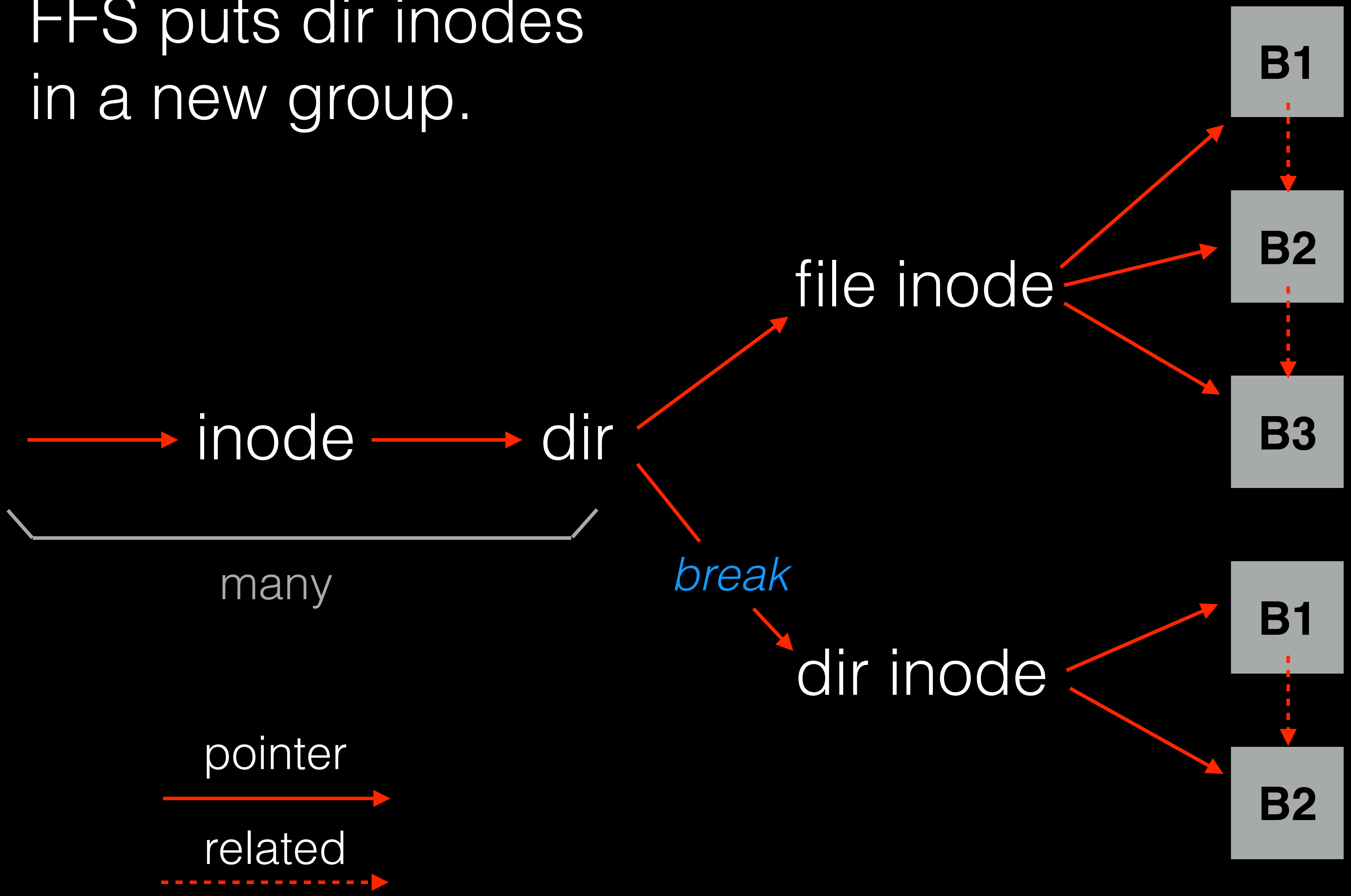


inode
dir
many
file inode
dir inode
B1
B2
B3
B1
B2
pointer
related

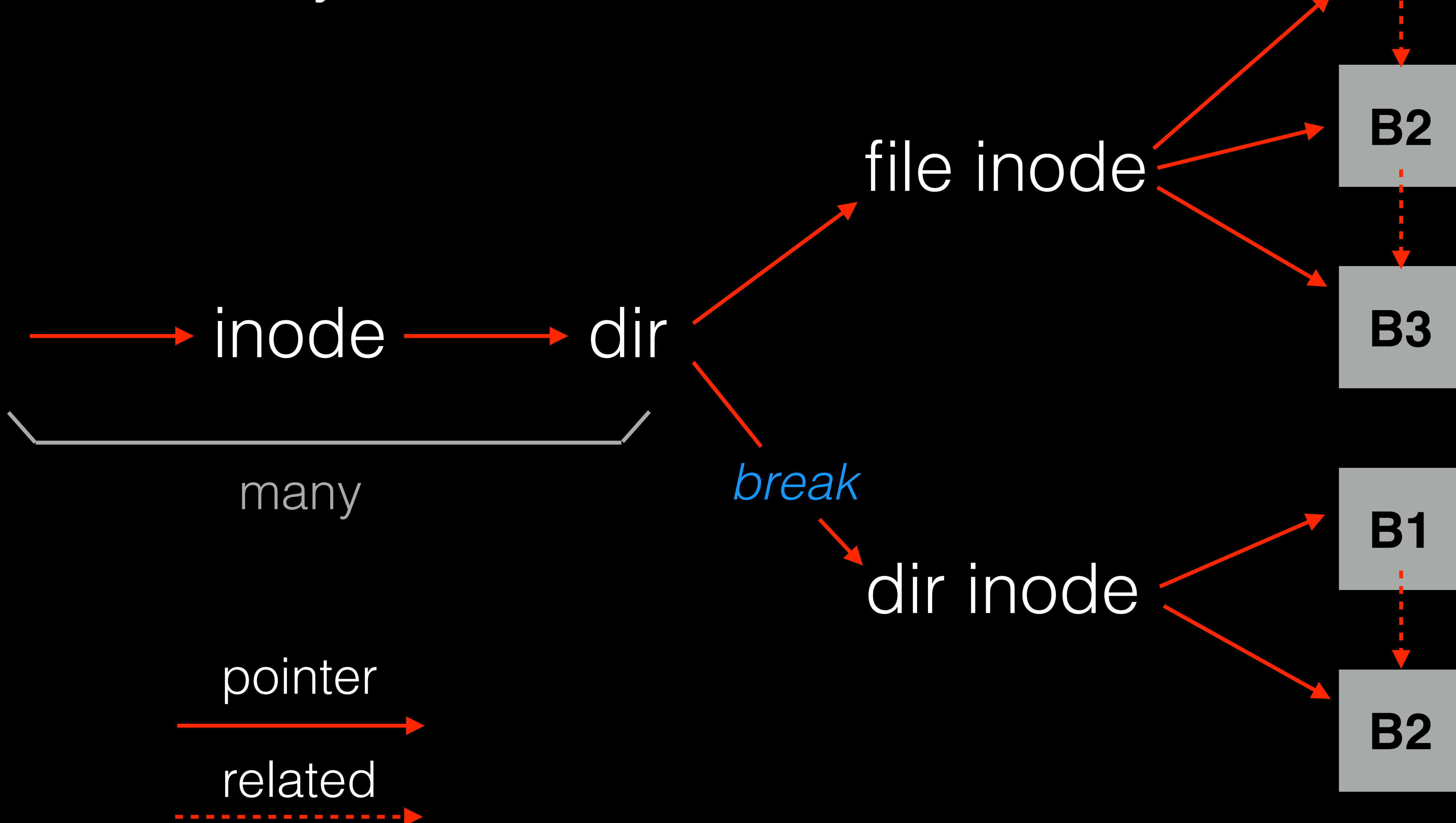
Where to cut the tree and start growing into another group?



FFS puts dir inodes in a new group.



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



Preferences

File inodes: allocate in same group with dir

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

First data block: allocate near inode

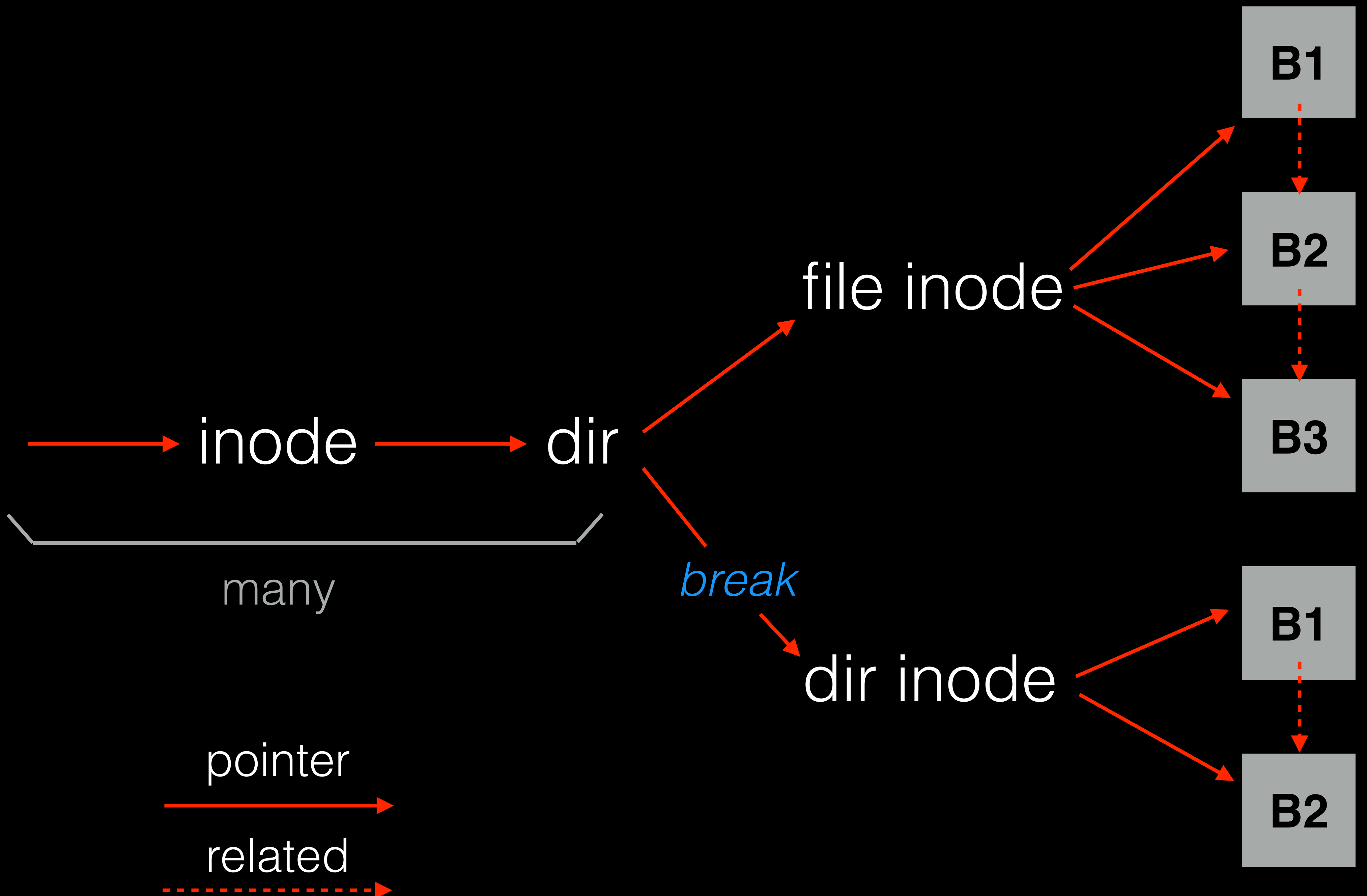
Other data blocks: allocate near previous block

Problem: Large Files

A single large file can use nearly all of a group.

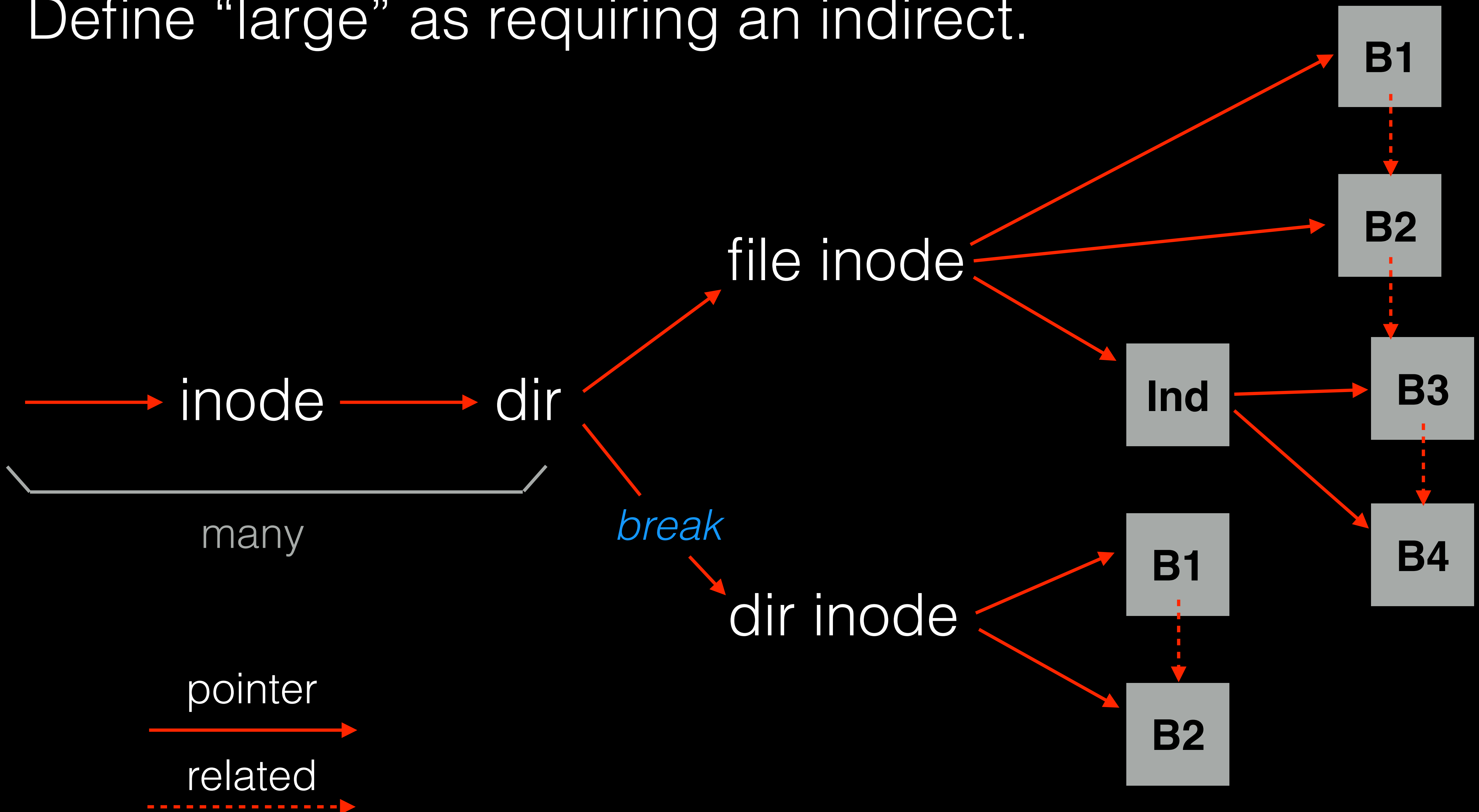
This displaces data for many small files.

It's better to do one seek for the large file than one seek for each of many small files.

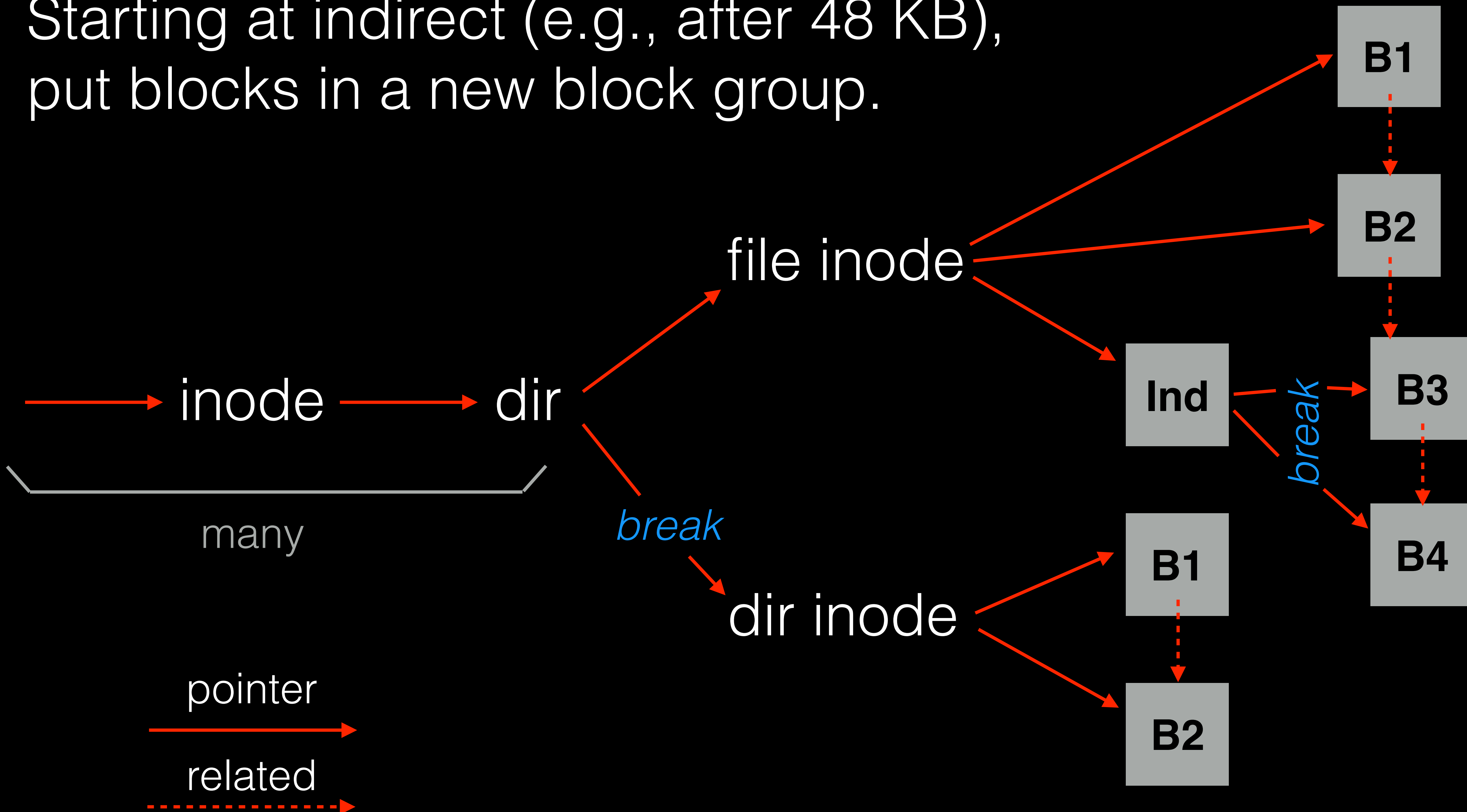


B1
B2
B3
file inode
inode
dir
many
break
dir inode
B1
B2
pointer
related

Define "large" as requiring an indirect.



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



Conclusion

First disk-aware file system.

FFS inspired modern files systems, including ext2 and ext3.

FFS also introduced several new features:
- long file names
- atomic rename
- symbolic links

Advice

All hardware is unique.

Treat disk like disk!

Treat flash like flash!

Treat random-access memory like random-access memory!

Advice

All hardware is unique.

Treat disk like disk!

Treat flash like flash!

Treat random-access memory like random-access memory!
(actually don't -- the name is a lie)

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 ".."?
…

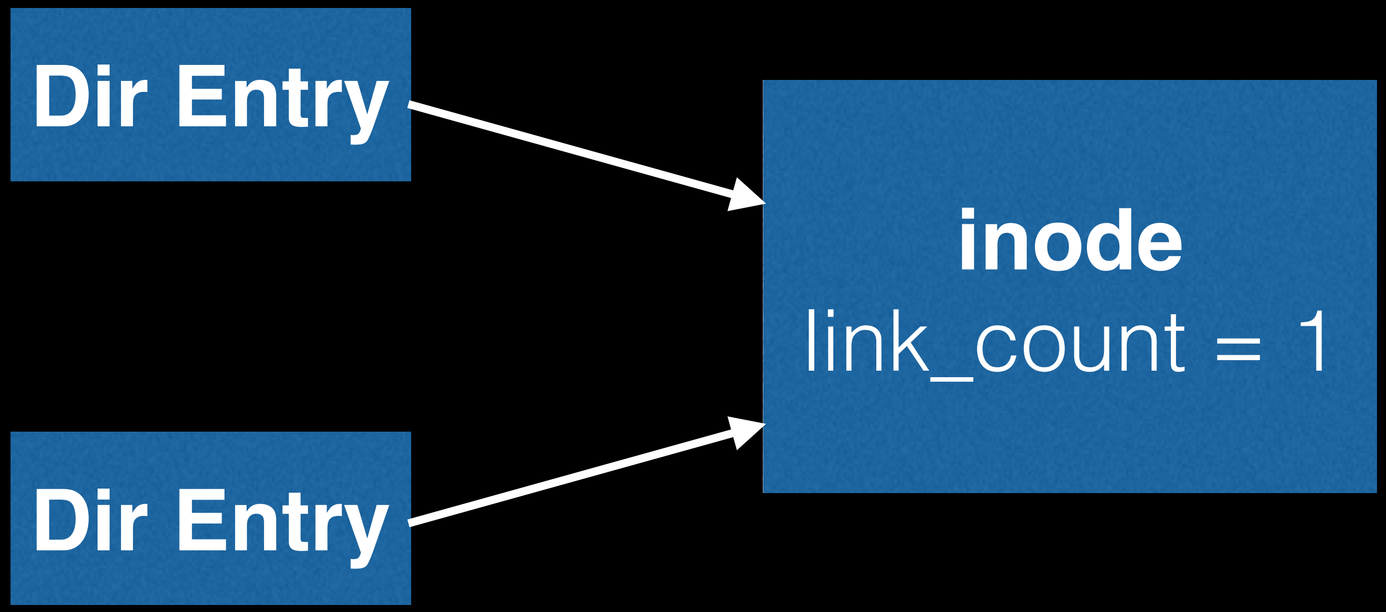
fsck

Other checks:

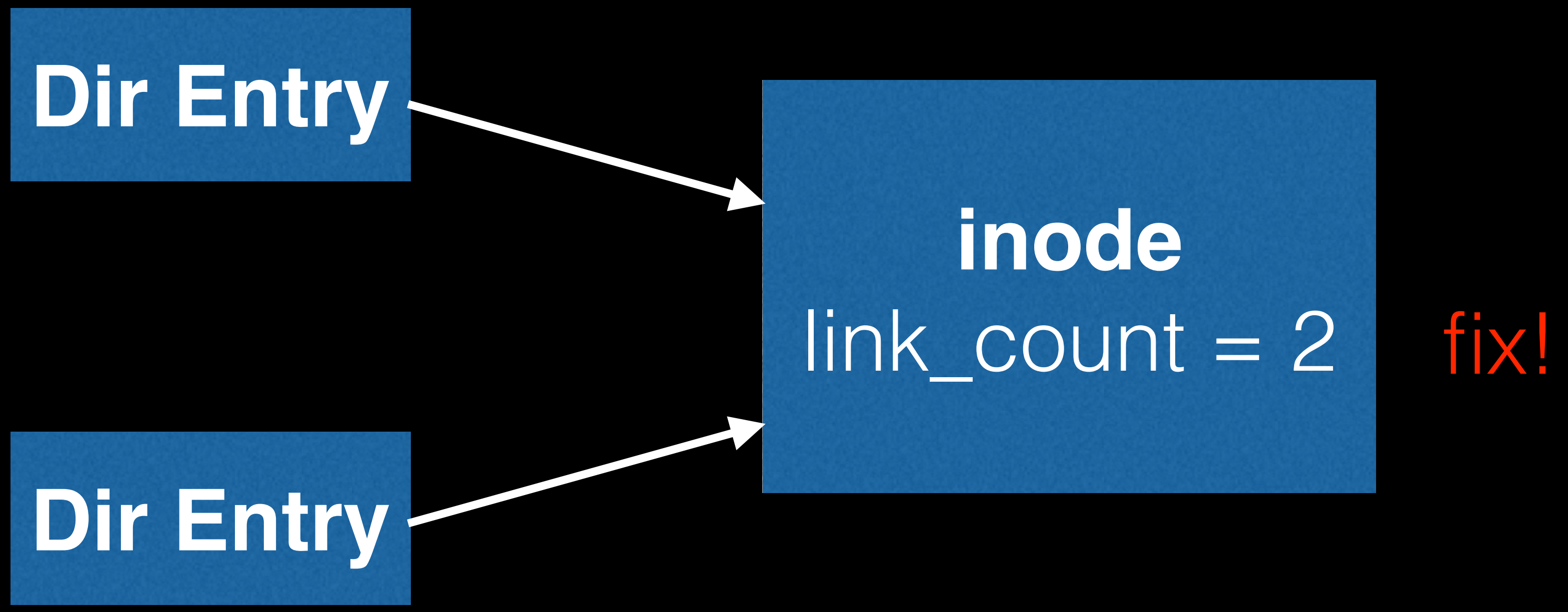
Do superblocks match?
Do number of dir entries equal inode link counts?
Do different inodes ever point to same block?
Do directories contain "." and ".."?
…

How to solve problems?

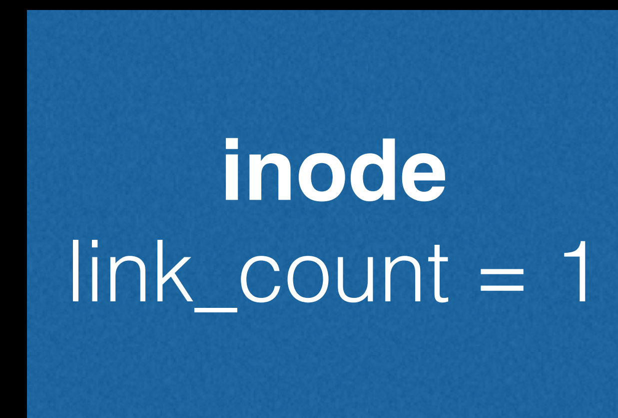
Link Count (example 1)



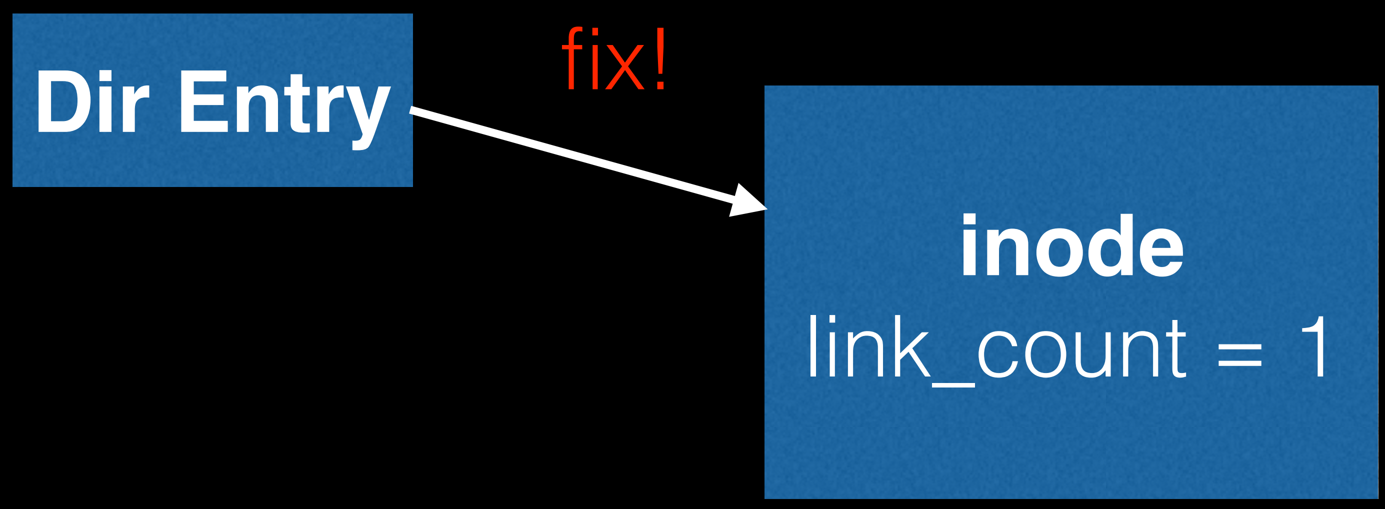
Link Count (example 1)



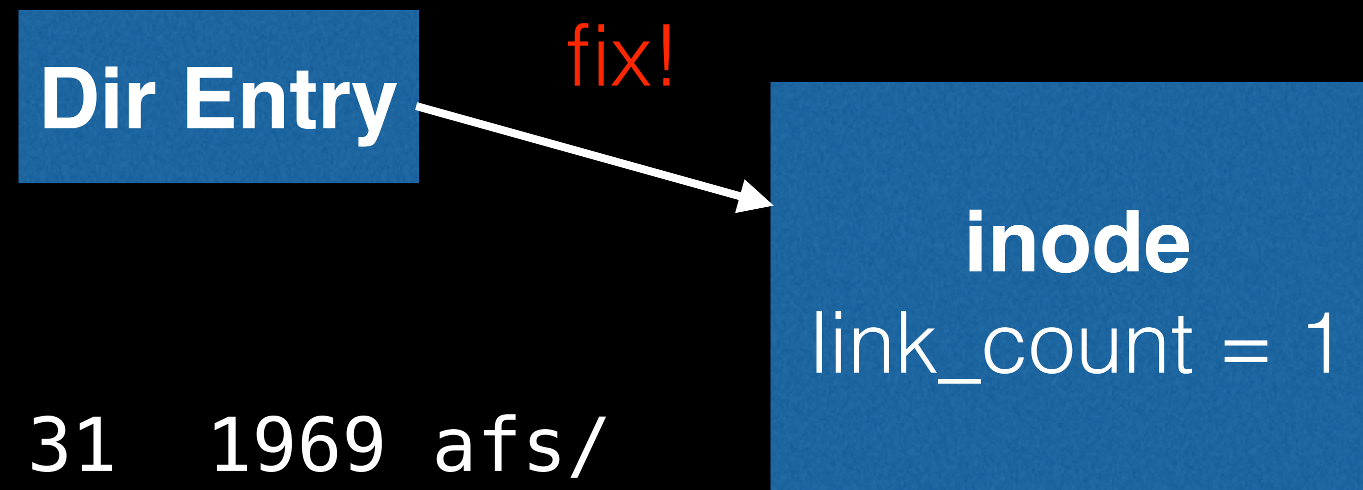
Link Count (example 2)



Link Count (example 2)

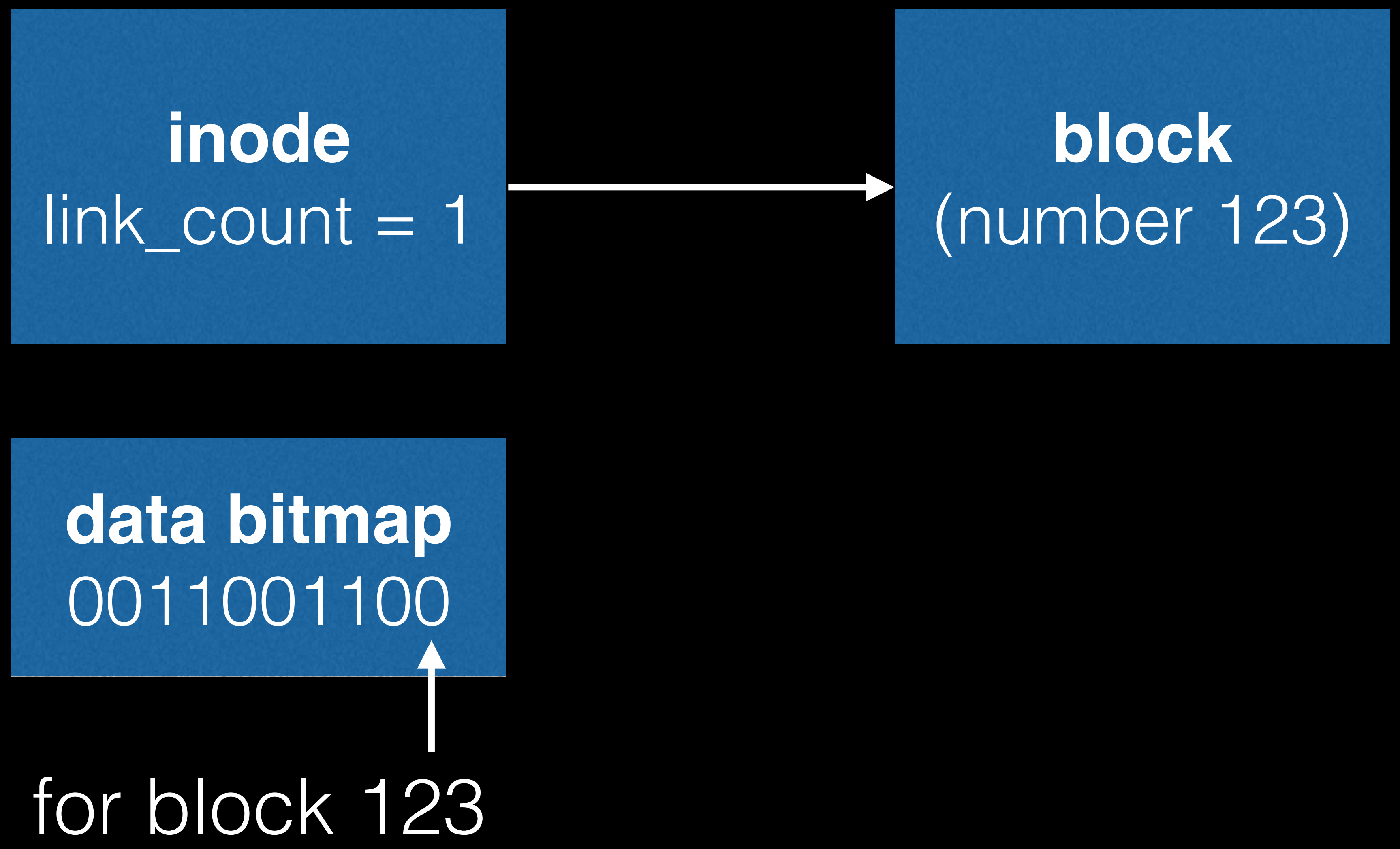


Link Count (example 2)

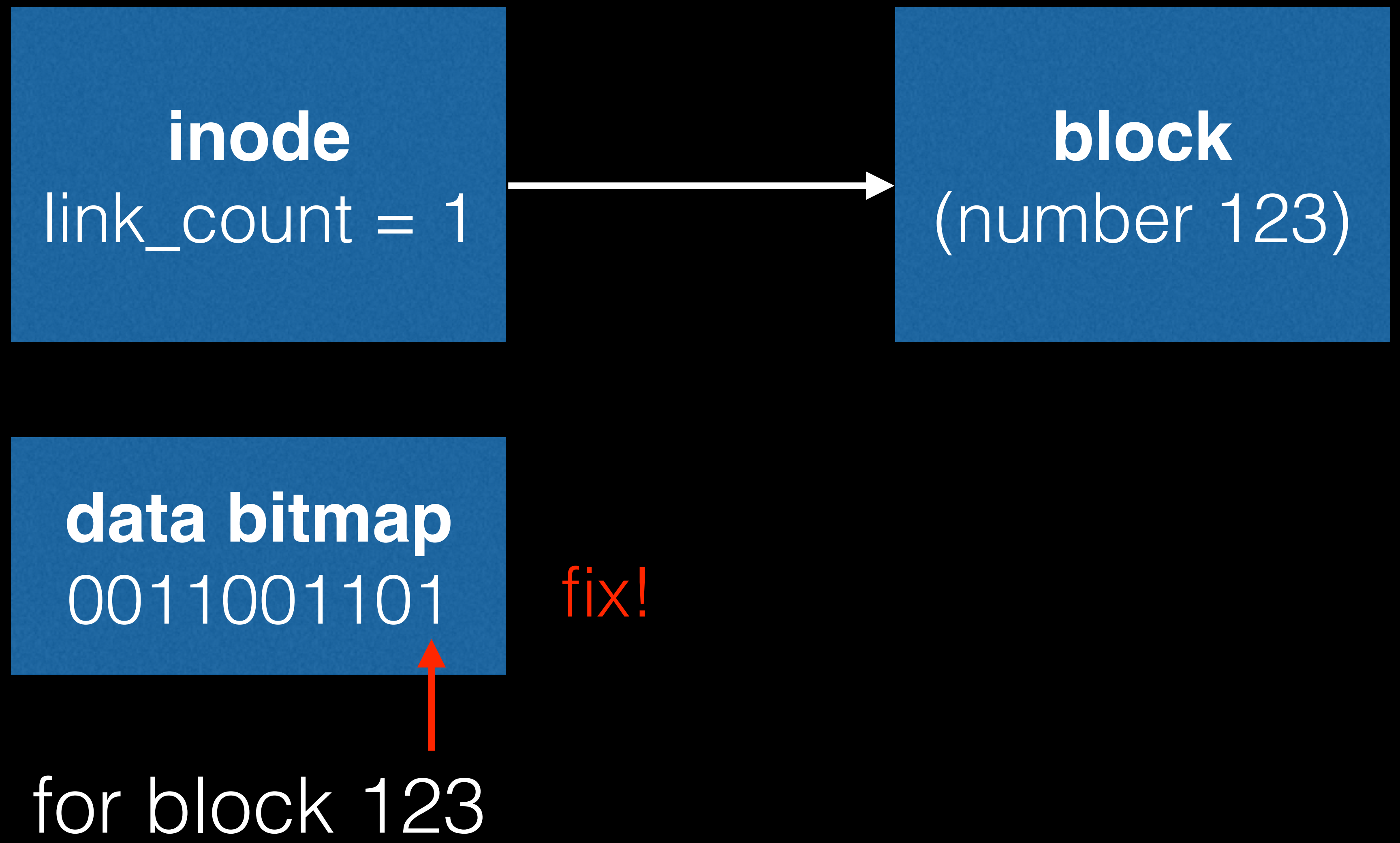



```
ls -l /
total 150
drwxr-xr-x  401 18432 Dec 31  1969 afs/
drwxr-xr-x.   2 4096  Nov  3 09:42 bin/
drwxr-xr-x.   5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.  13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.  10 12288 Nov  3 09:41 lib64/
drwx------.   2 16384 Aug  1 10:57 lost+found/
...
```

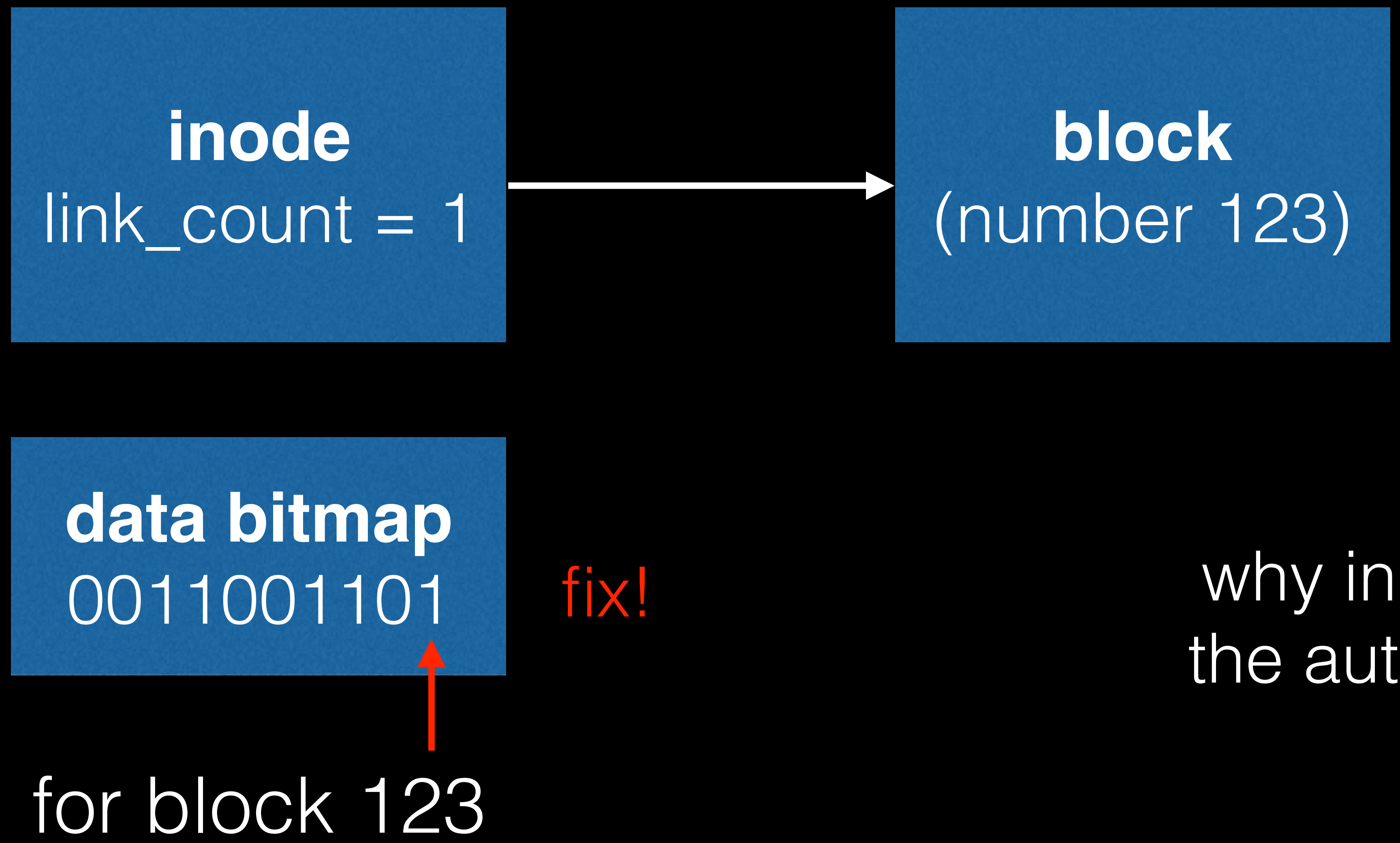
Data Bitmap


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

Data Bitmap

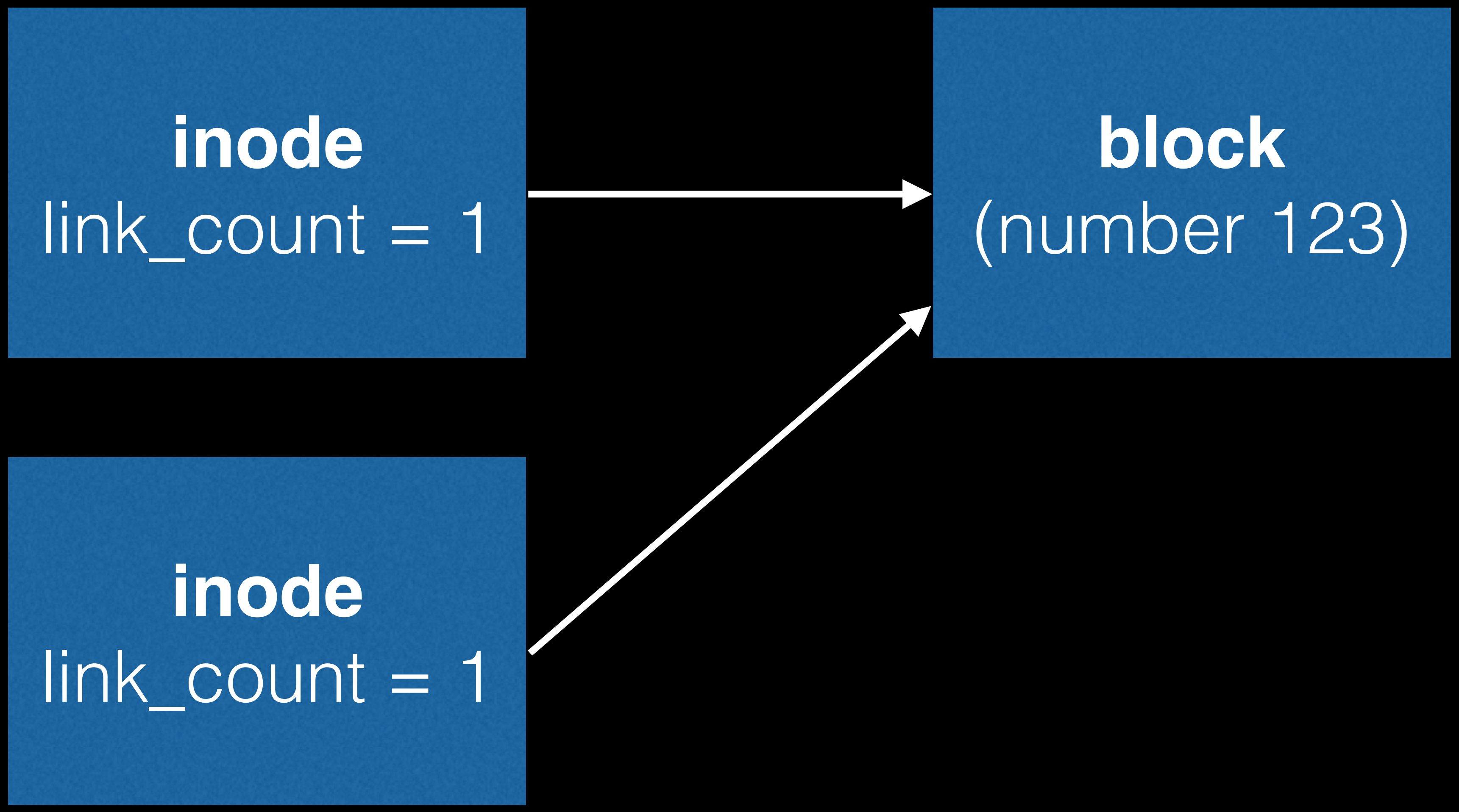

inode
link_count = 1
block
(number 123)
data bitmap
0011001101
fix!
for block 123

Data Bitmap

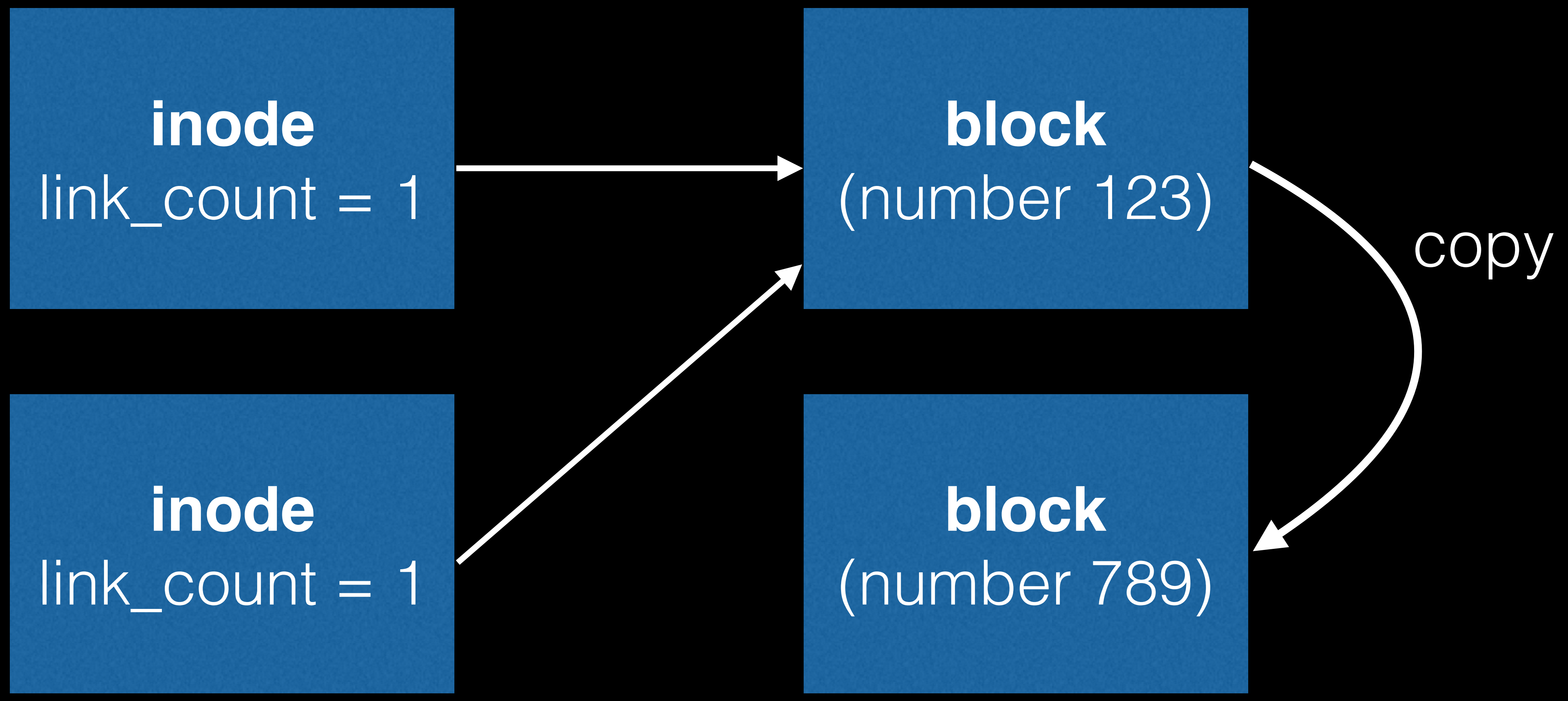


why in inode
the authority?

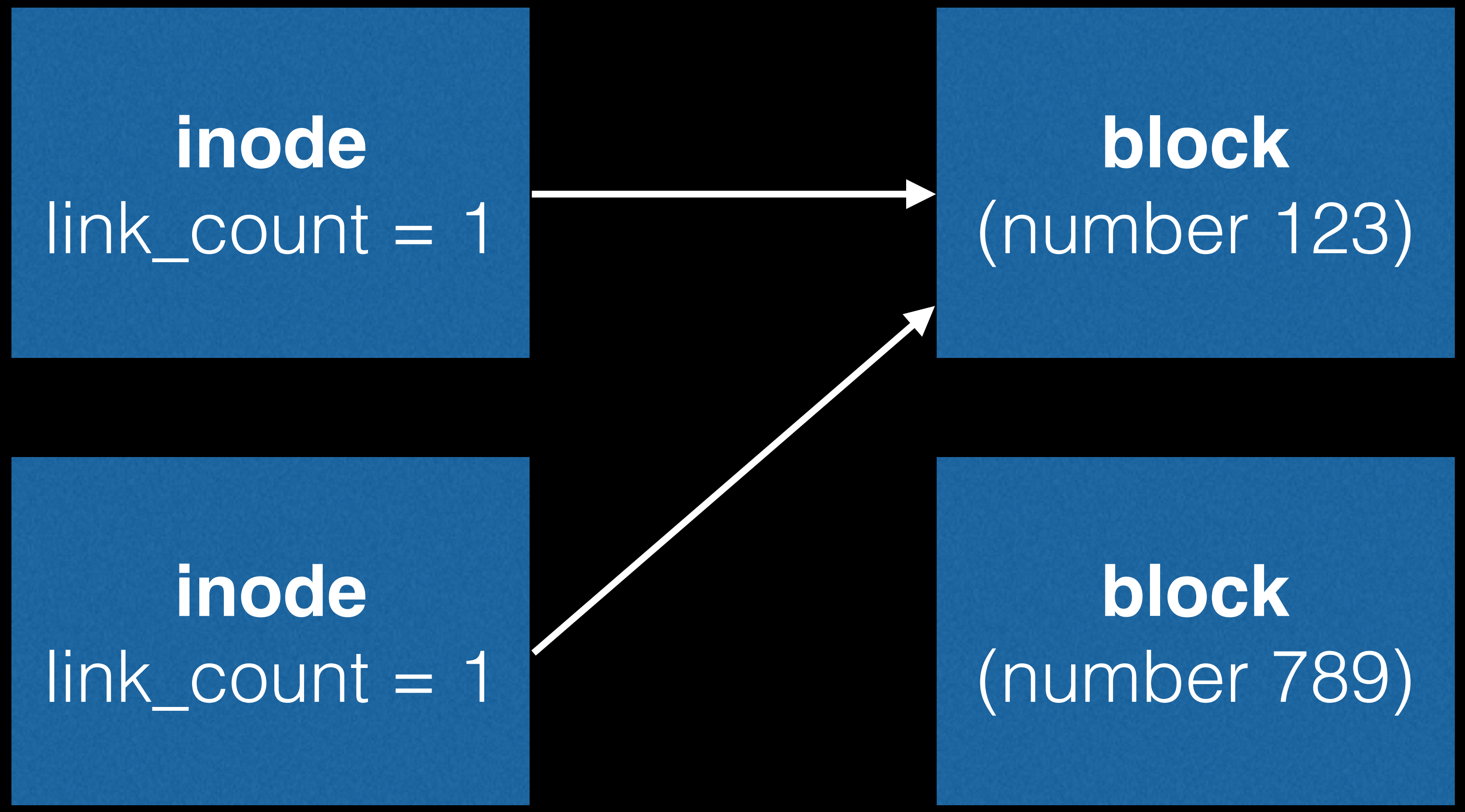
Duplicate Pointers



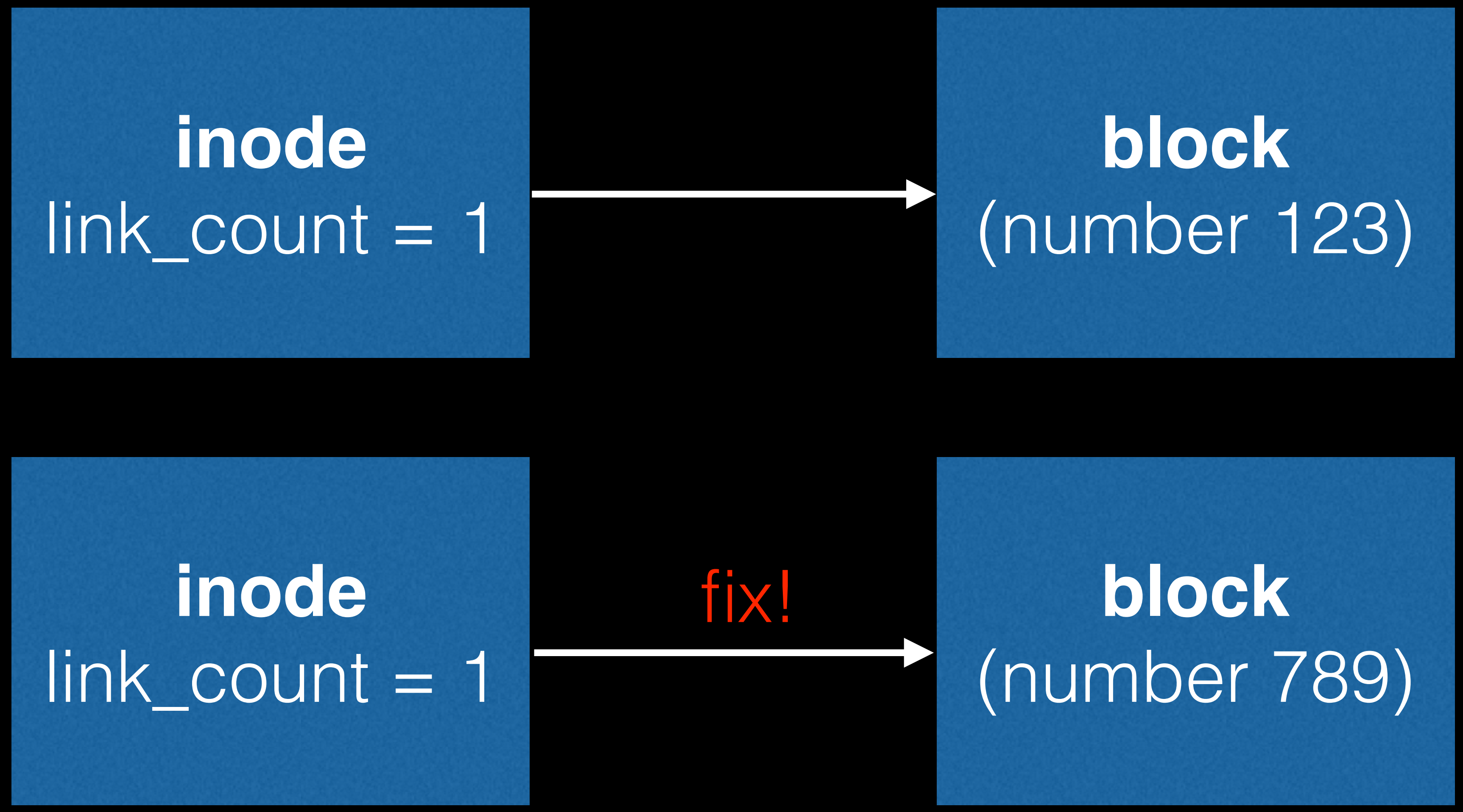
Duplicate Pointers



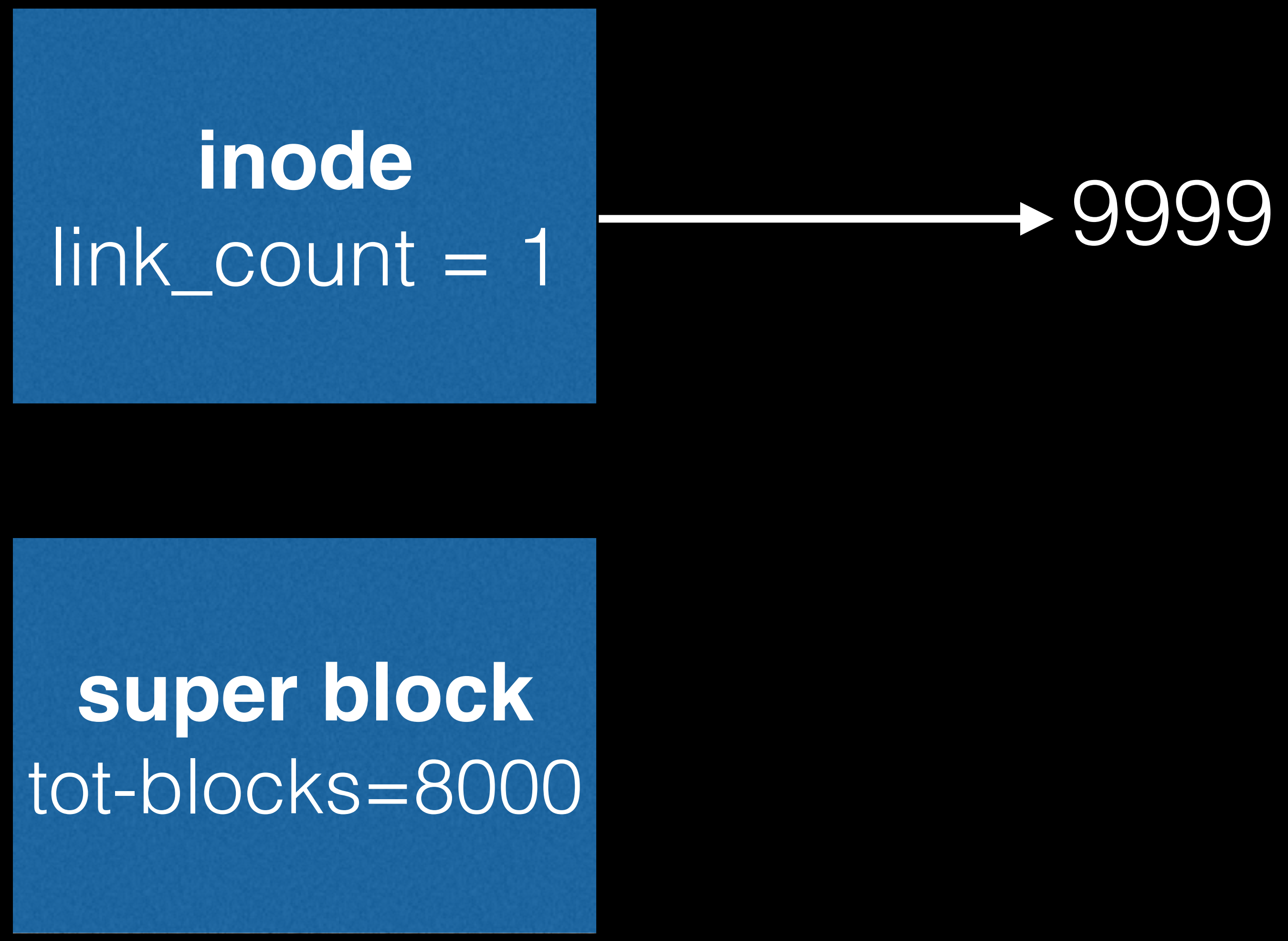
Duplicate Pointers



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.

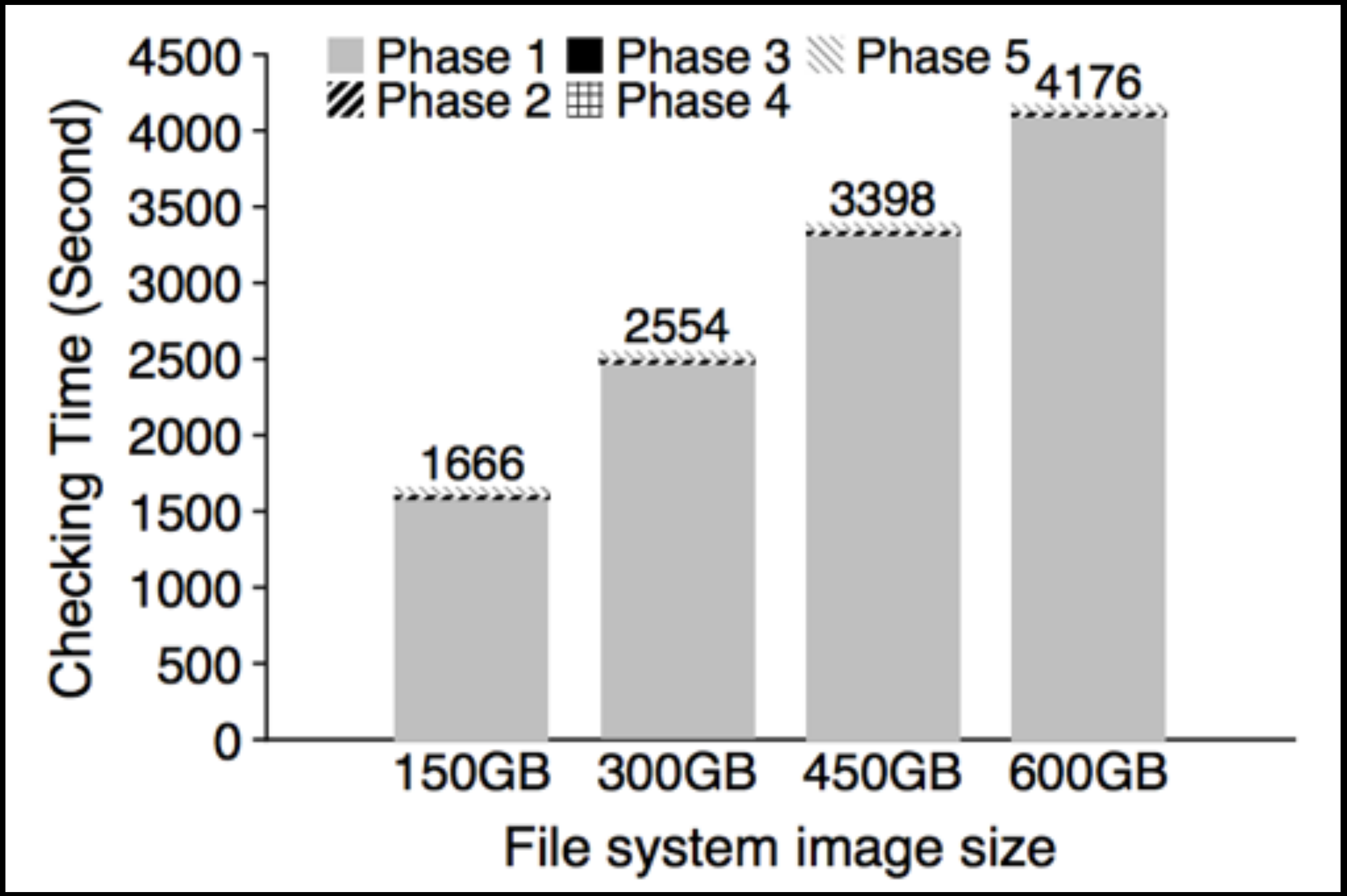
fsck

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.

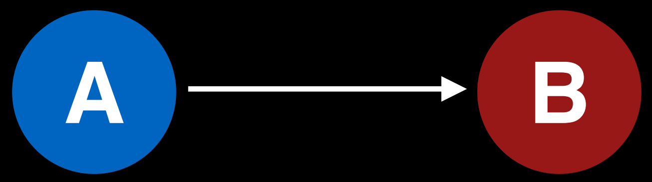
Atomicity

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.

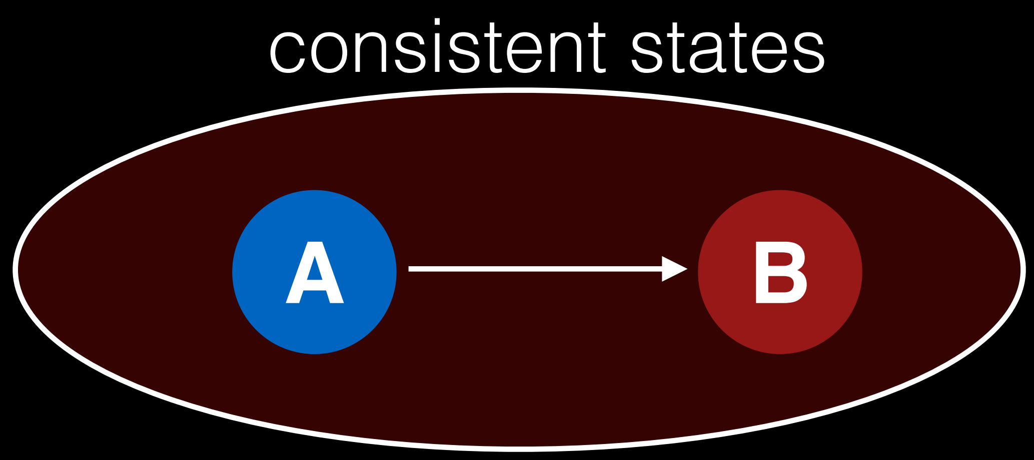
Consistency vs Correctness

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



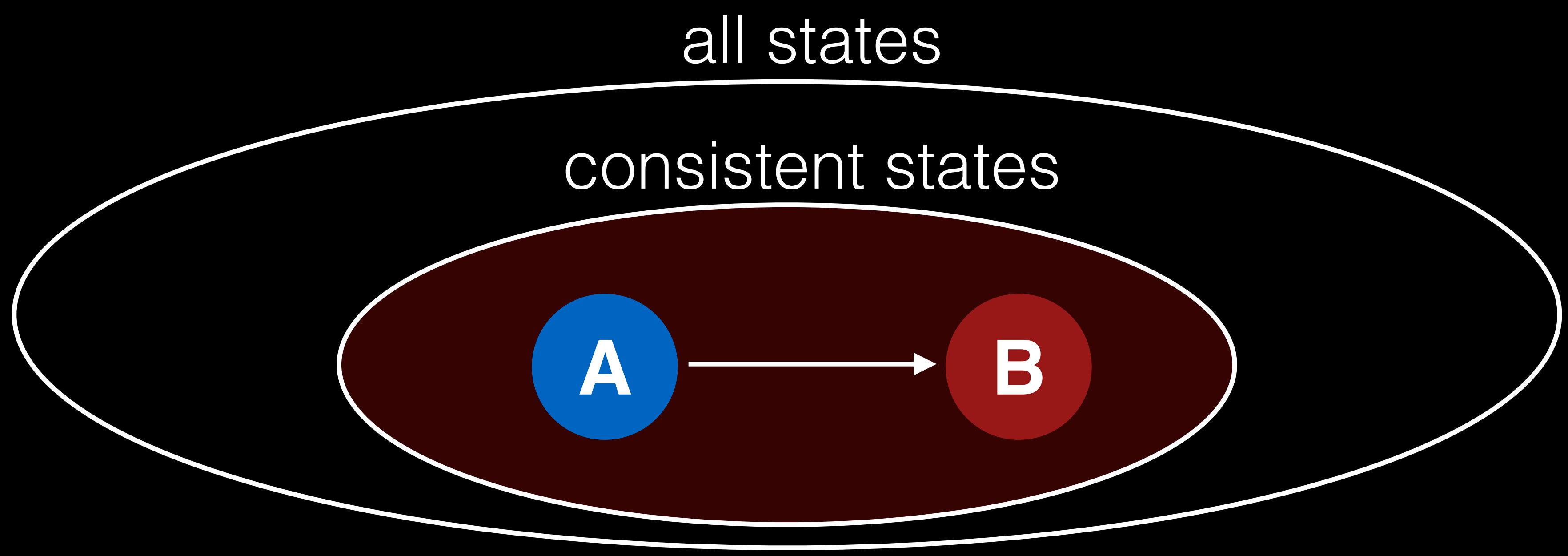
Consistency vs Correctness

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



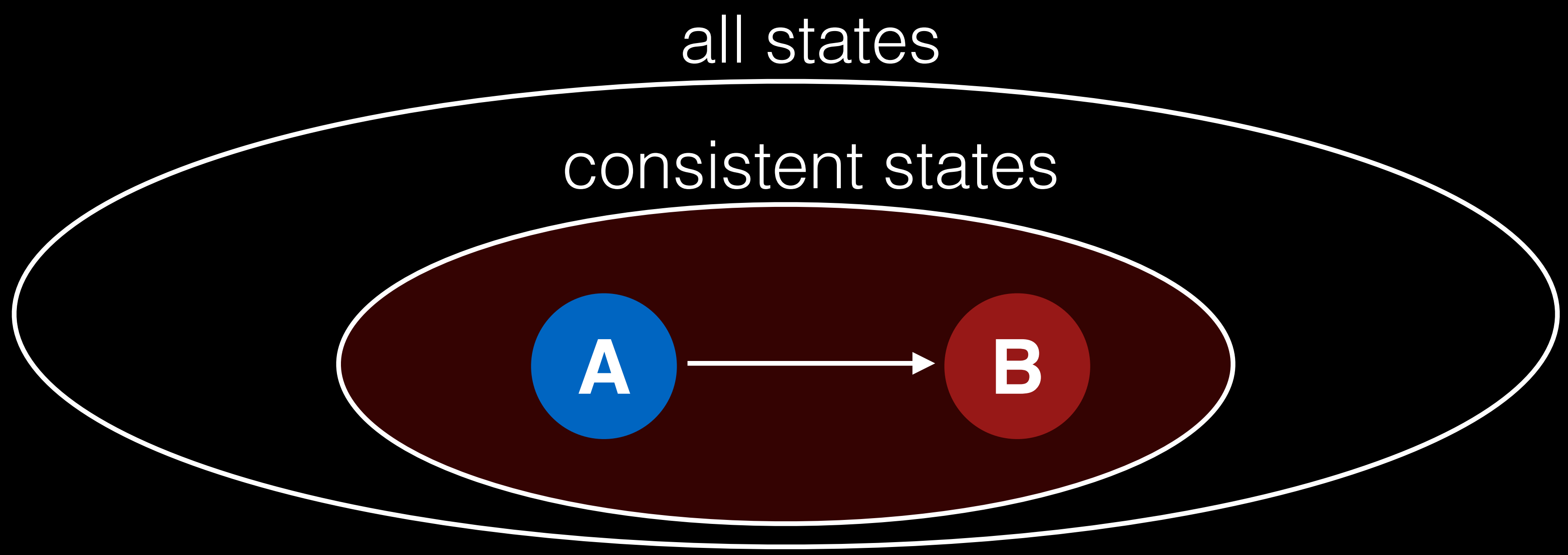
Consistency vs Correctness

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



Consistency vs Correctness

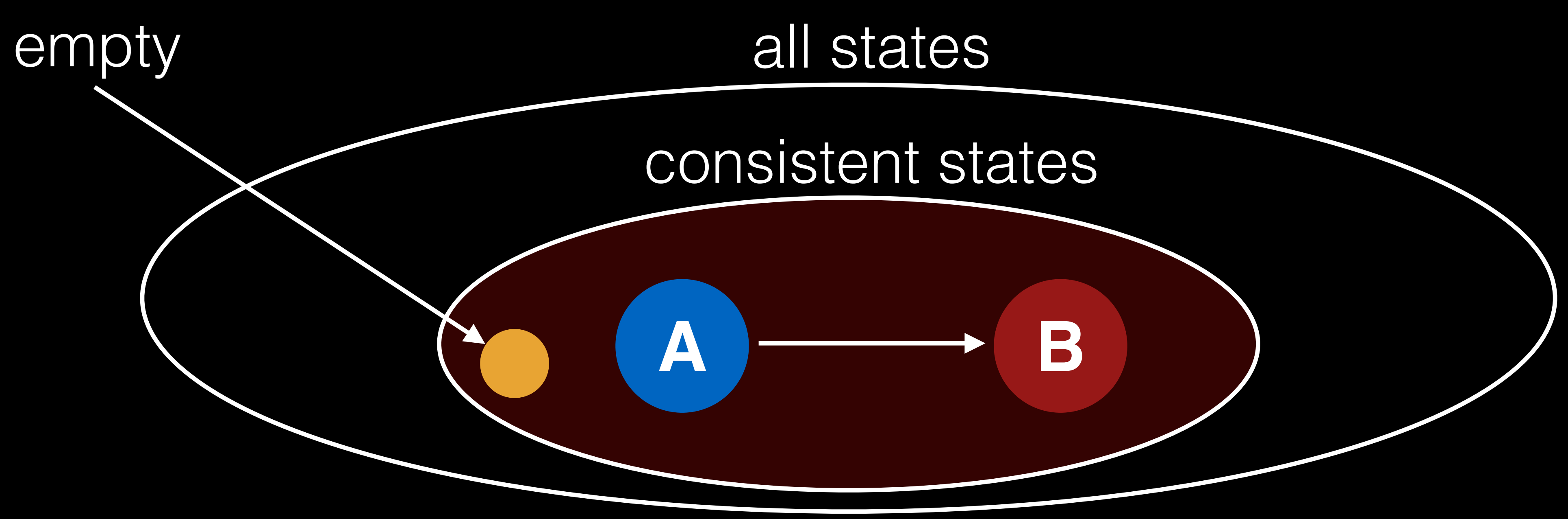
Say a set of writes moves the disk from state A to B.



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.

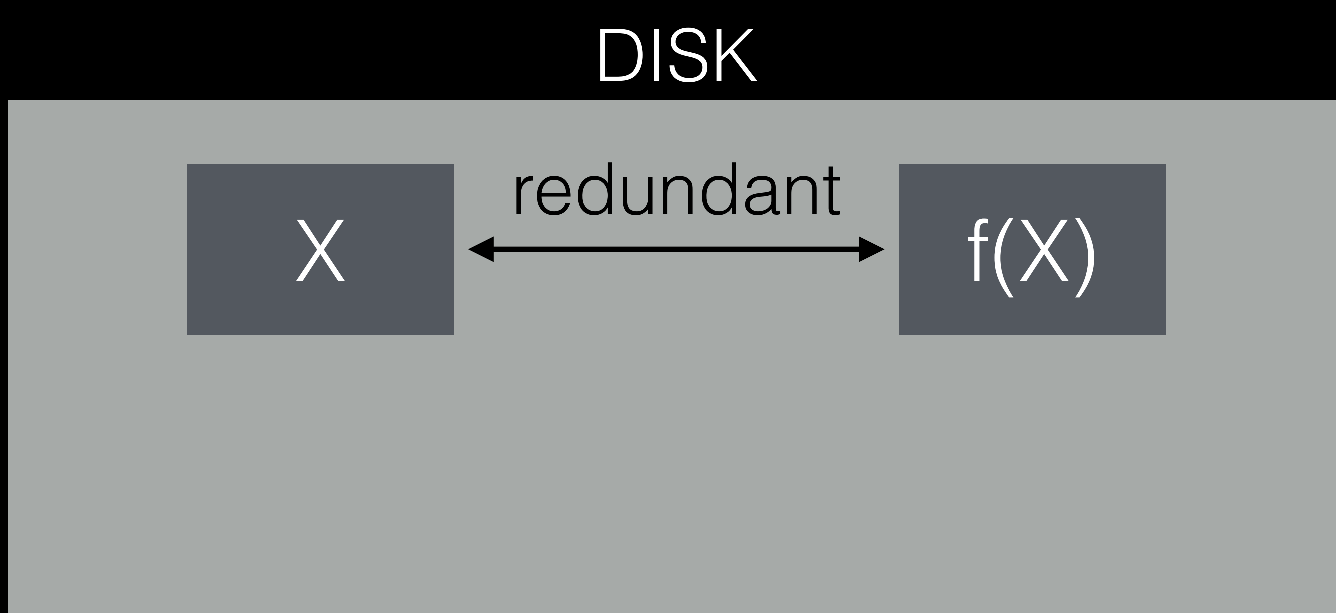
General Strategy

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.

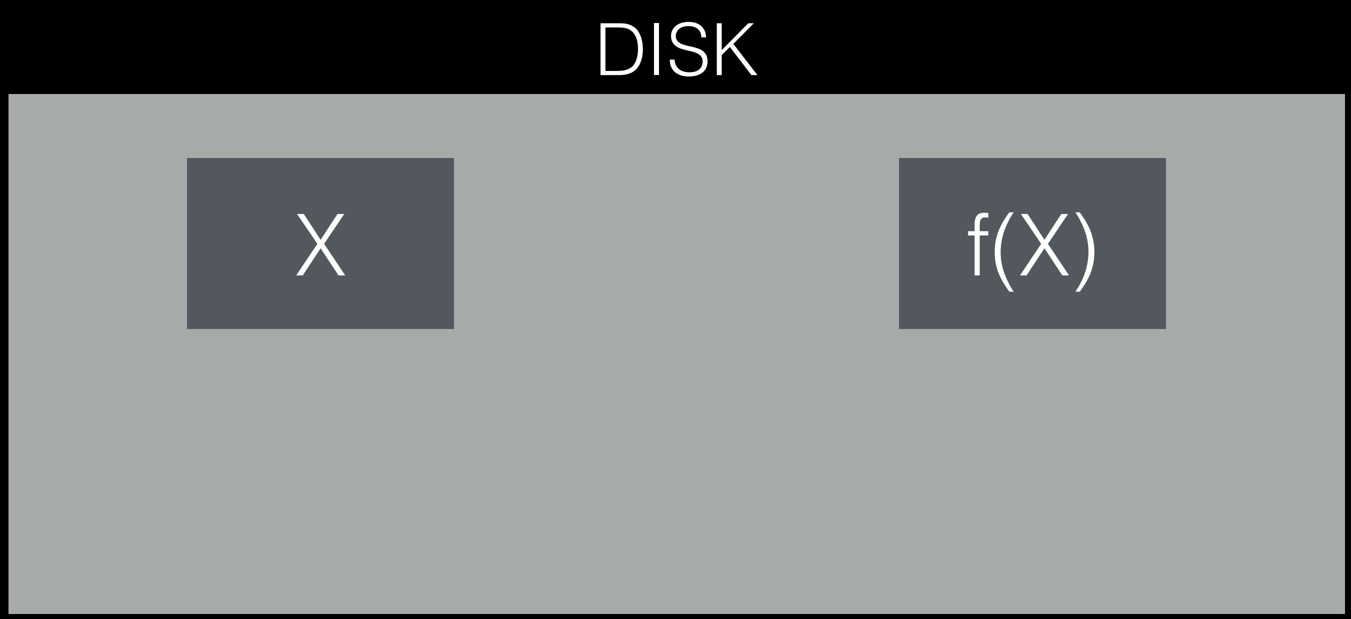
Fight Redundancy with Redundancy

Want to replace X with Y. Original:



Fight Redundancy with Redundancy

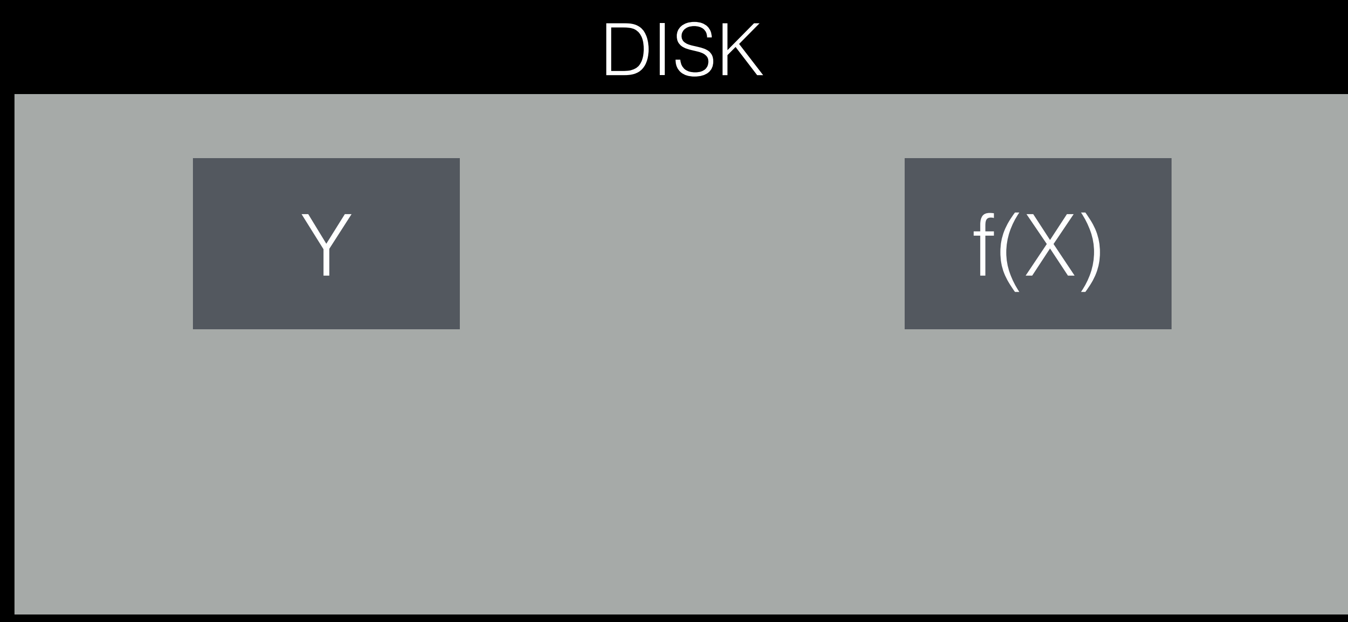
Want to replace X with Y. Original:



good time to crash

Fight Redundancy with Redundancy

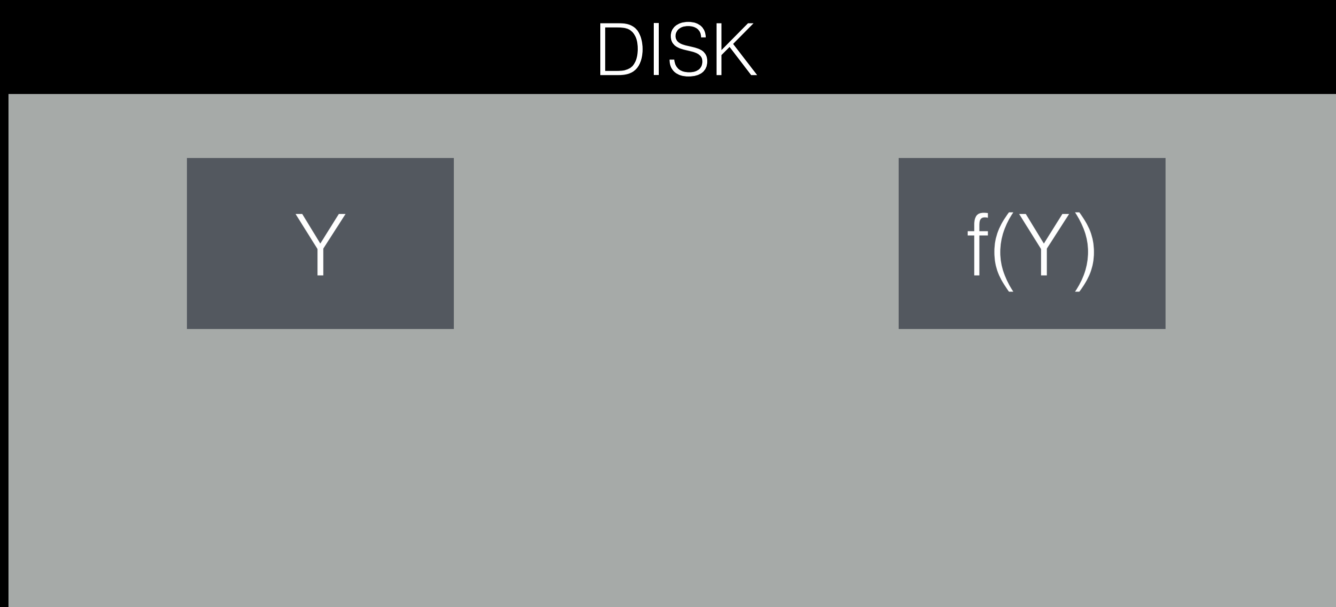
Want to replace X with Y. Original:



bad time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:



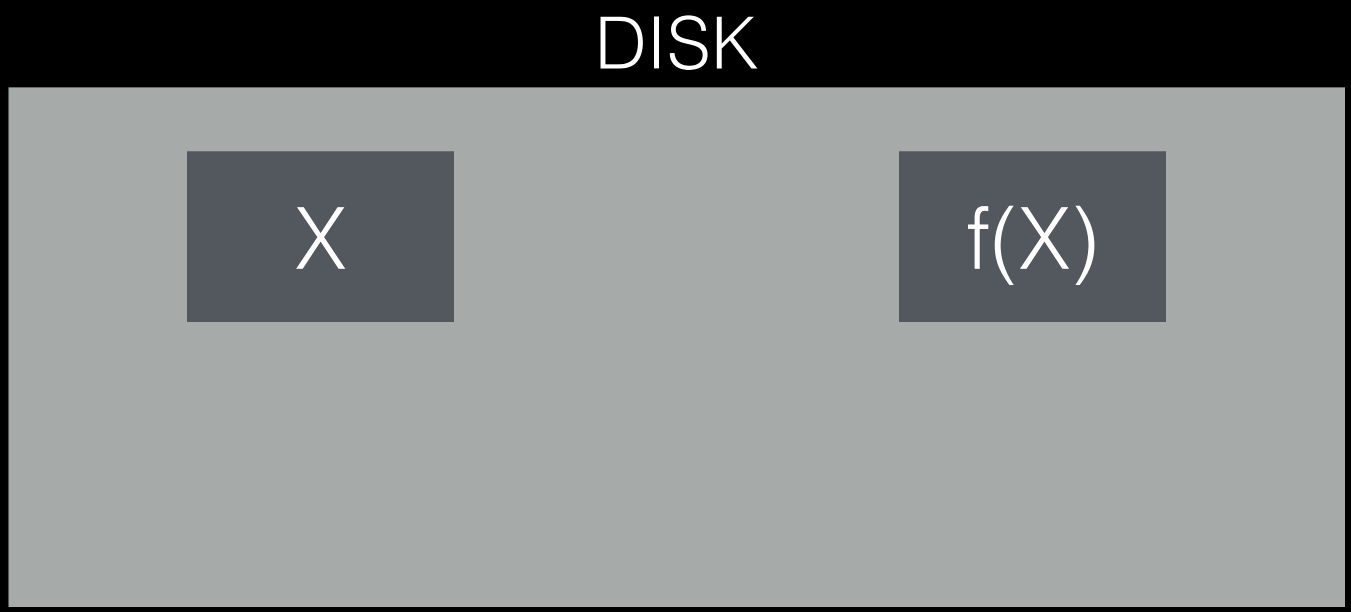
good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y.

Fight Redundancy with Redundancy

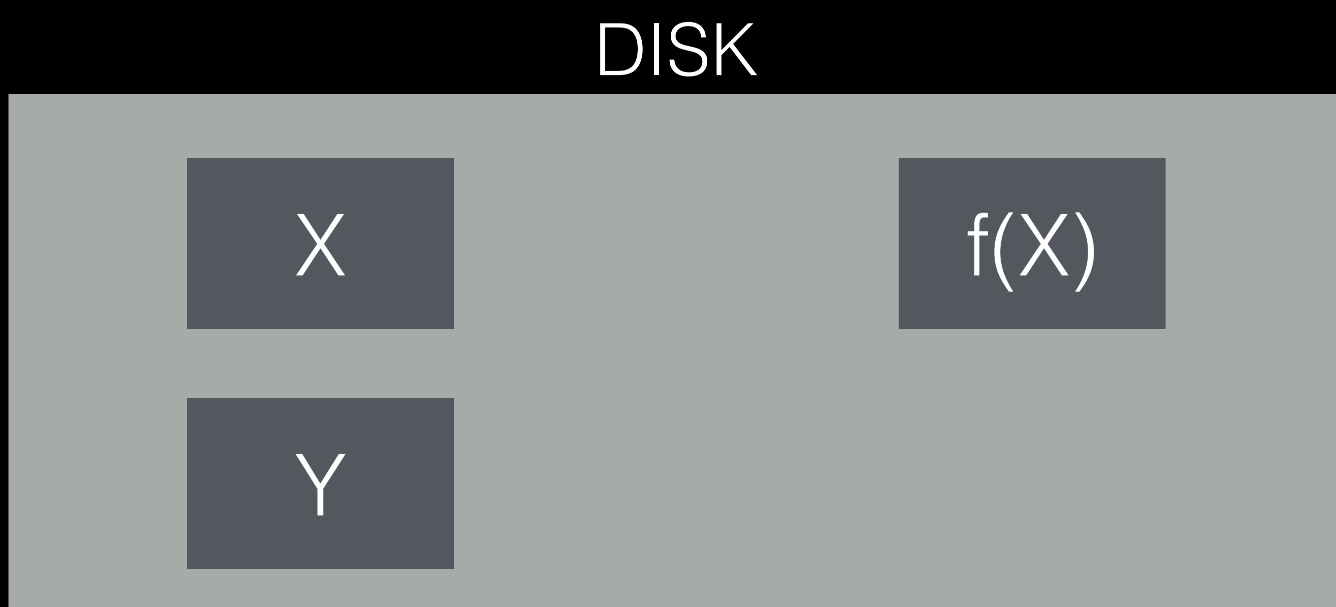
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

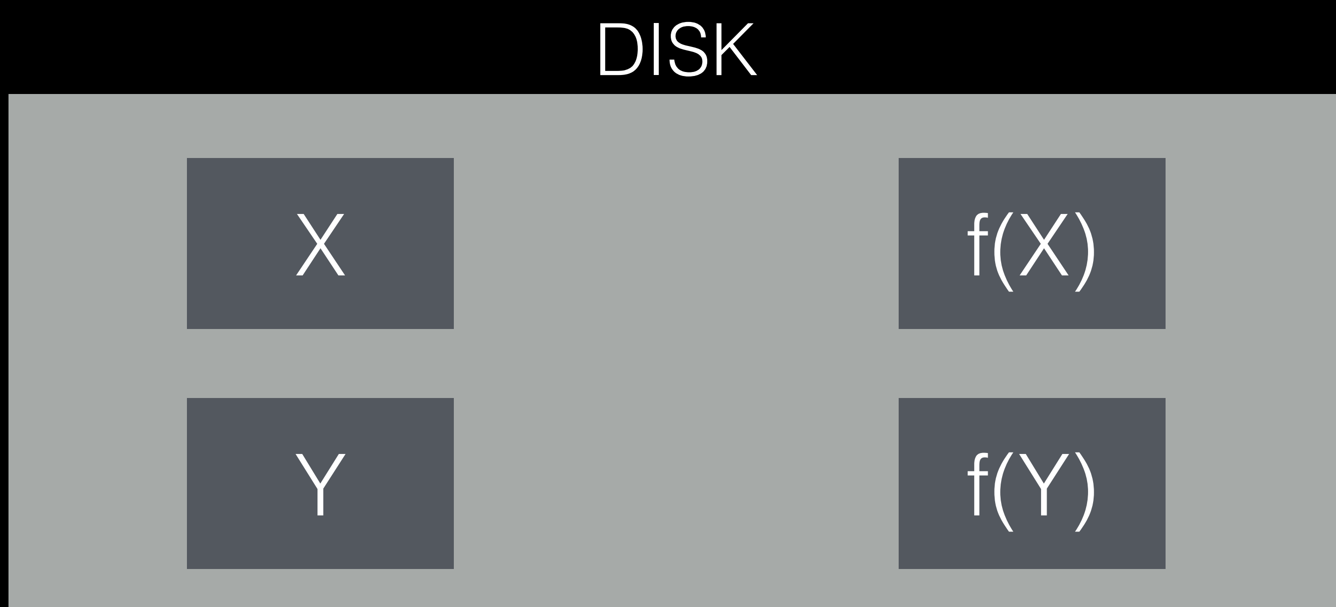
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

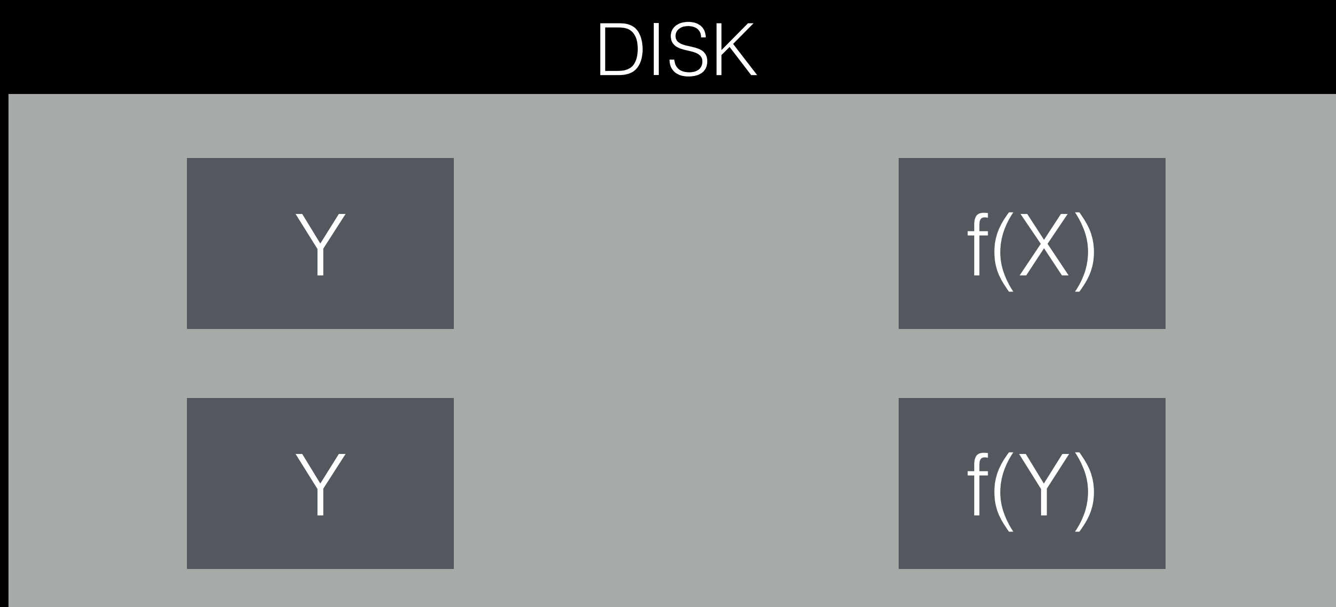
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

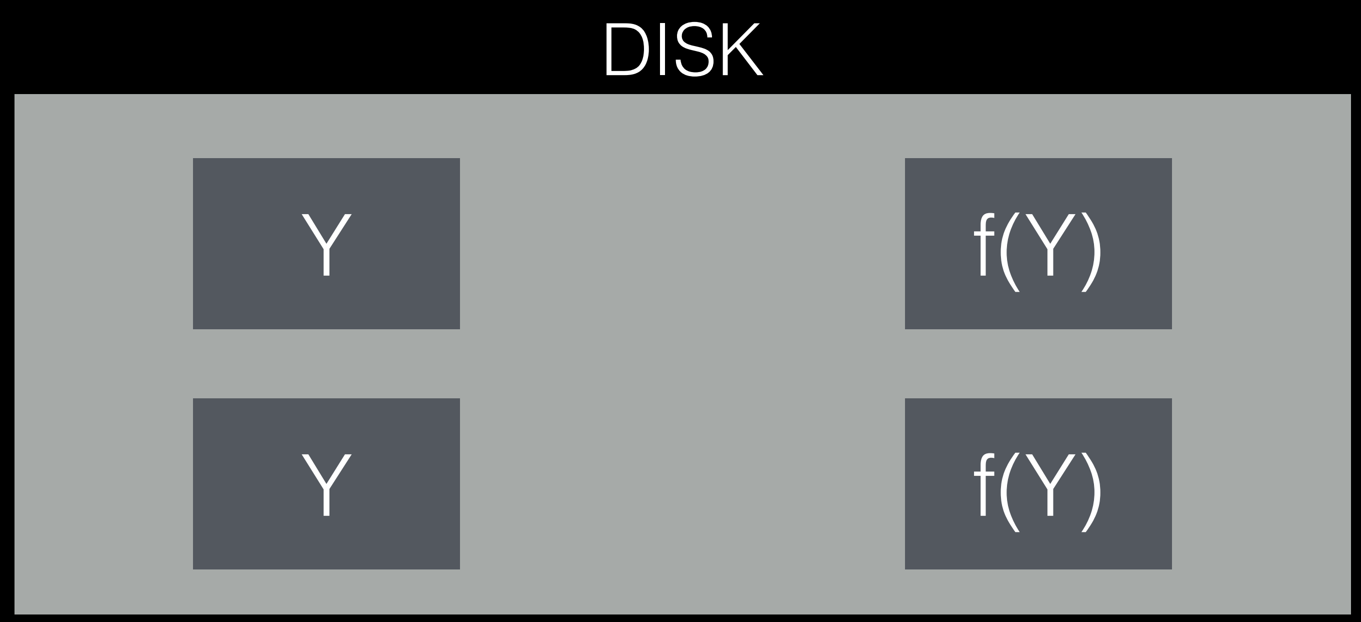
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

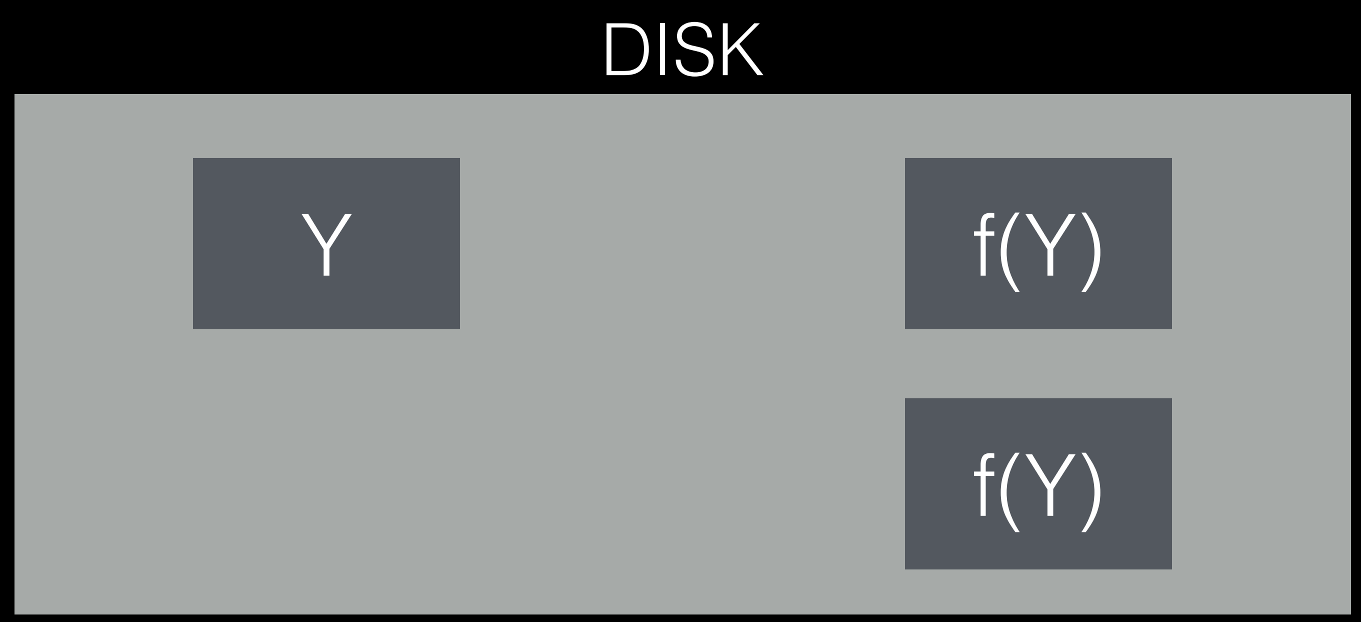
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

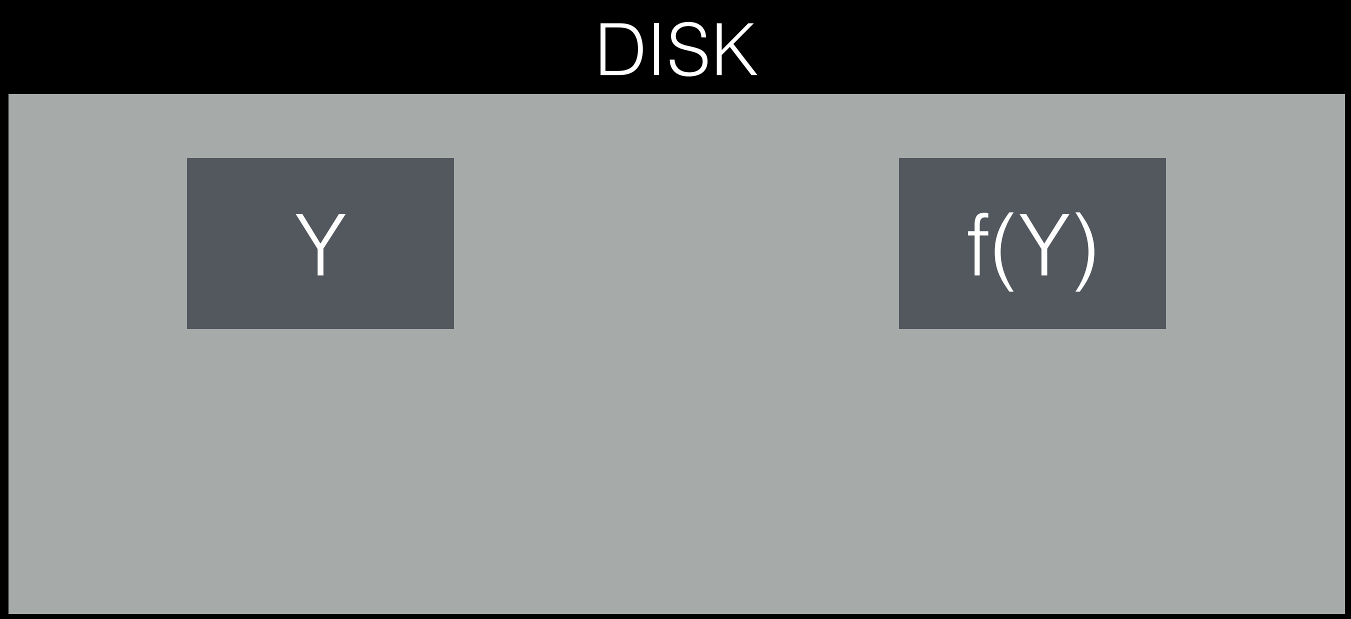
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

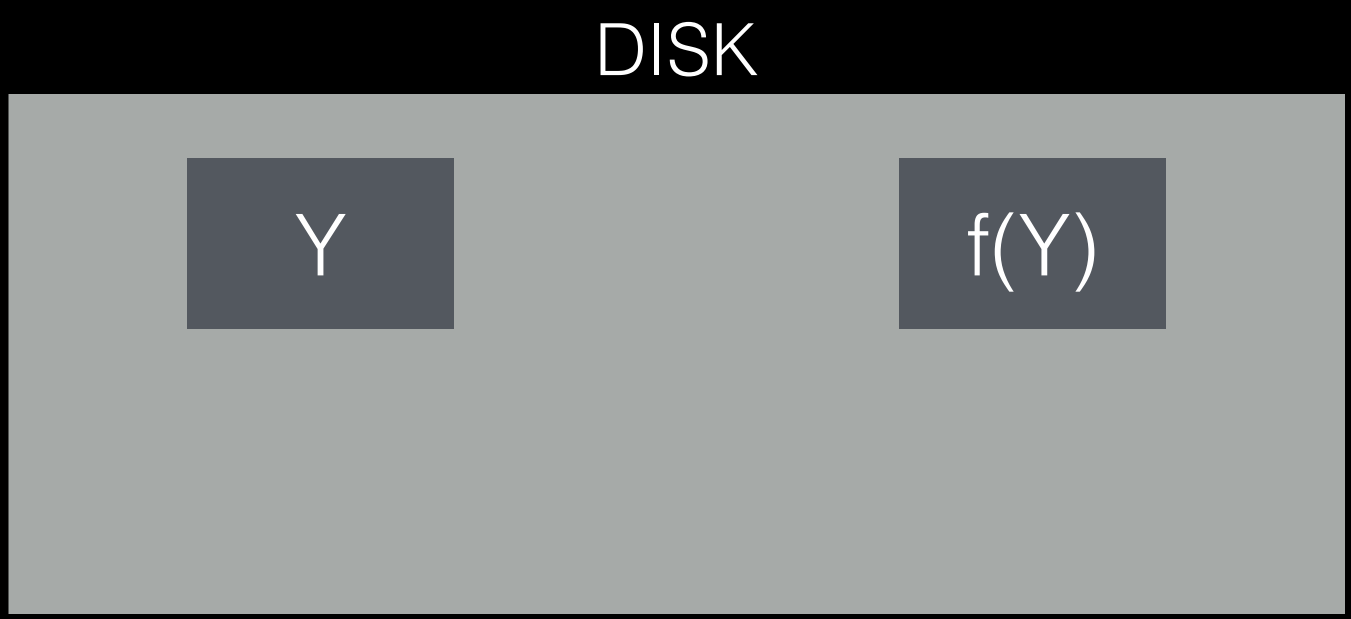
Want to replace X with Y. With journal:



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".

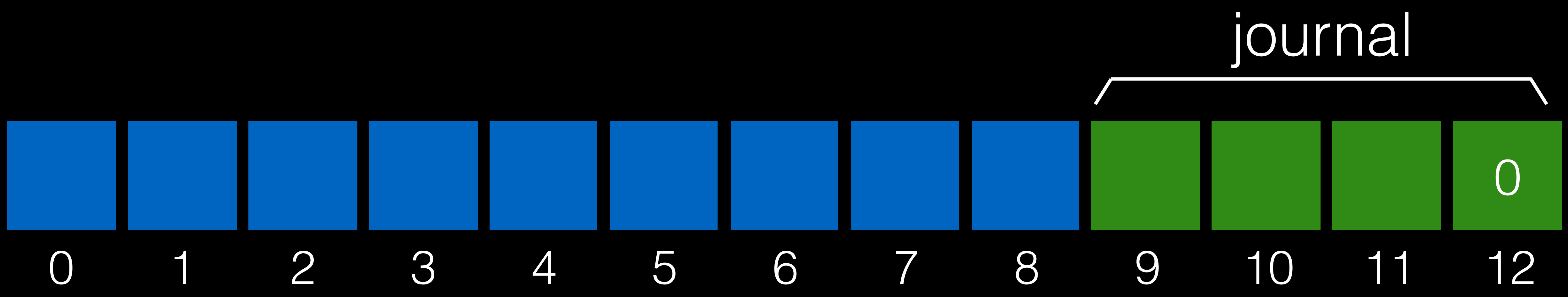
The writes to it are a "journal transaction".

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

File systems typically write new data to the journal.

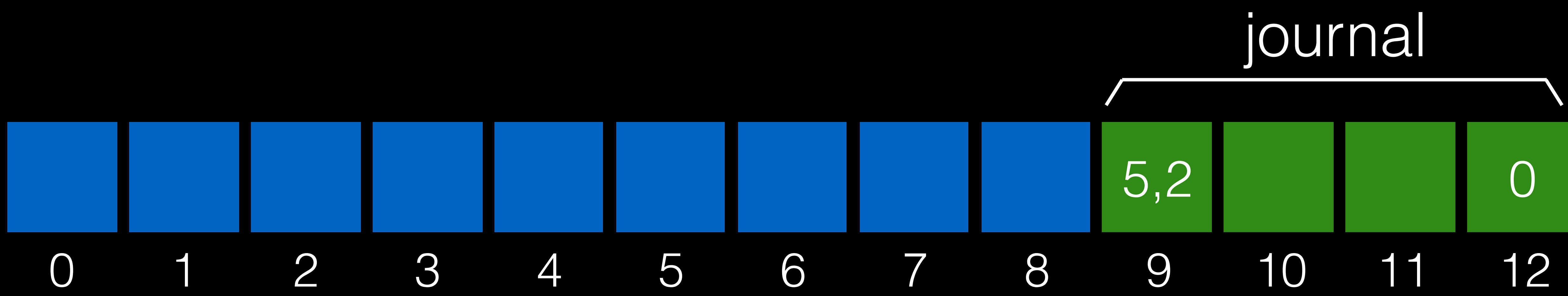
New Layout

New Layout



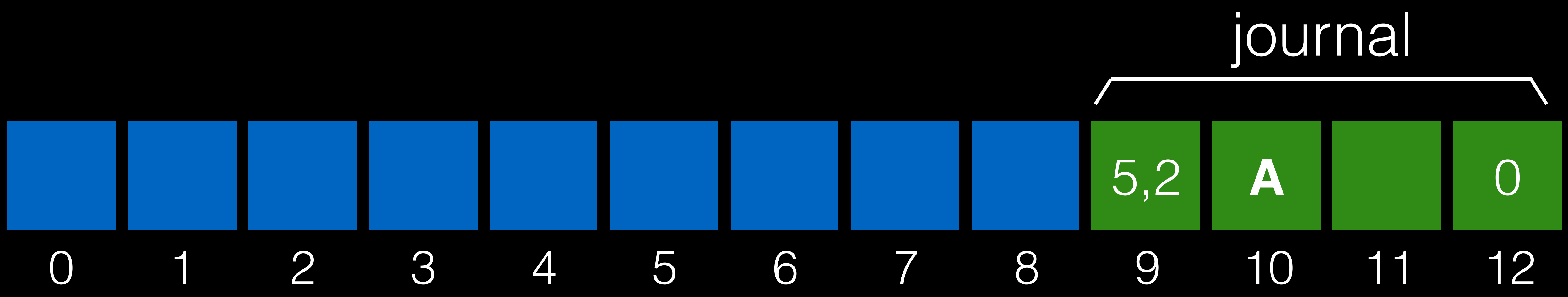
transaction: write A to block 5; write B to block 2

New Layout



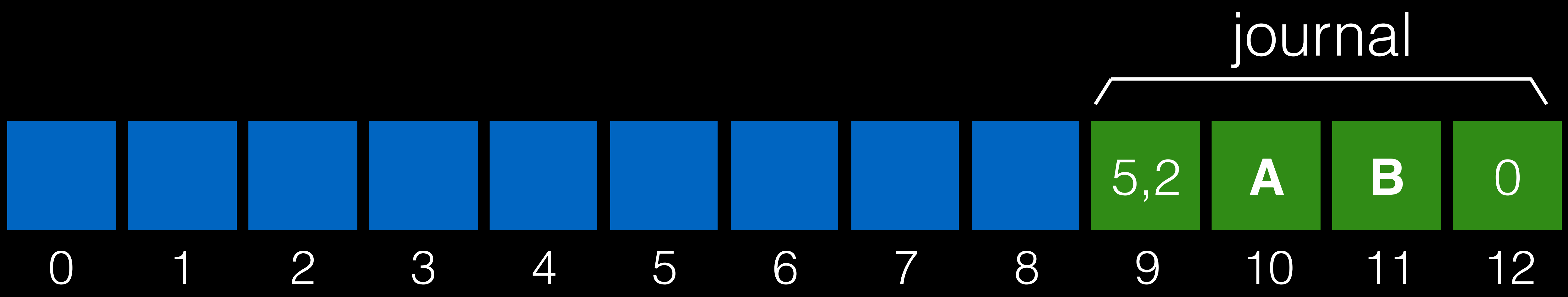
transaction: write A to block 5; write B to block 2

New Layout



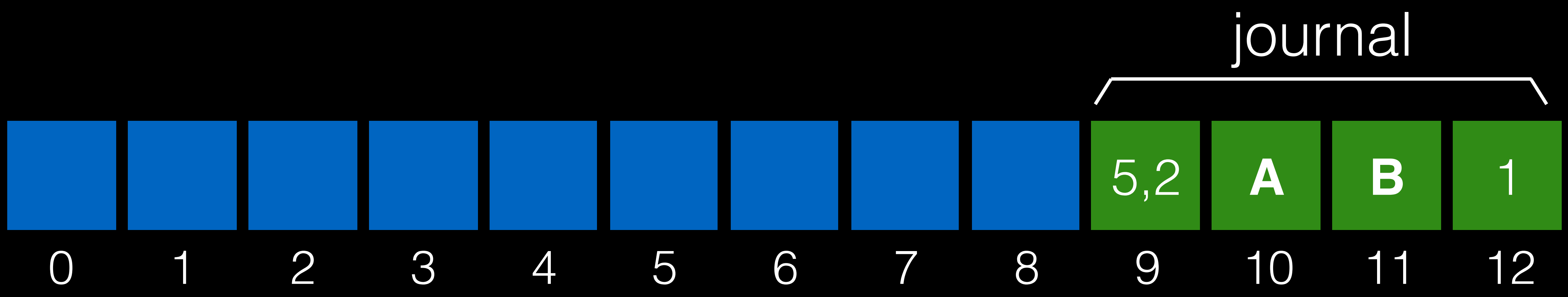
transaction: write A to block 5; write B to block 2

New Layout



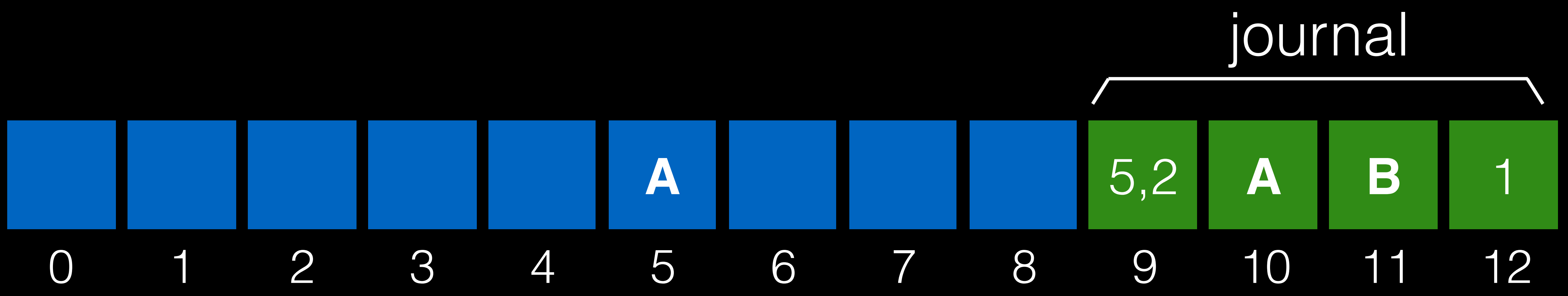
transaction: write A to block 5; write B to block 2

New Layout



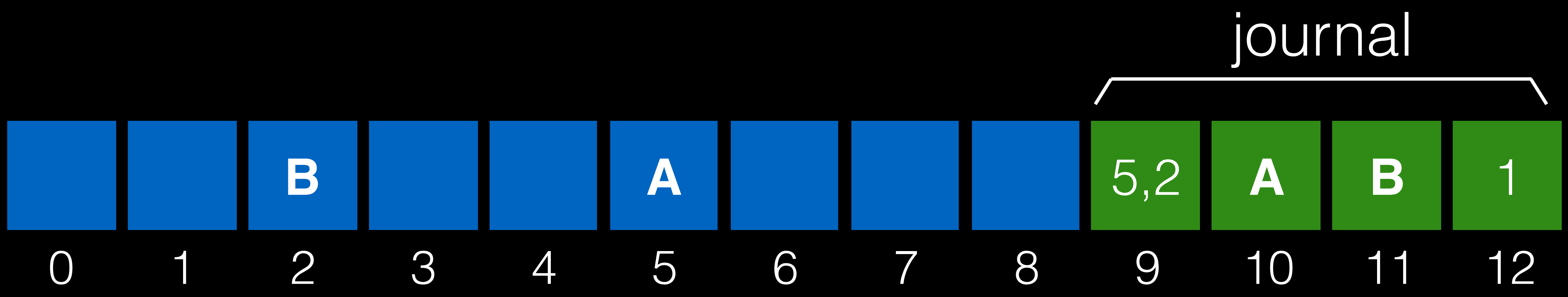
transaction: write A to block 5; write B to block 2

New Layout



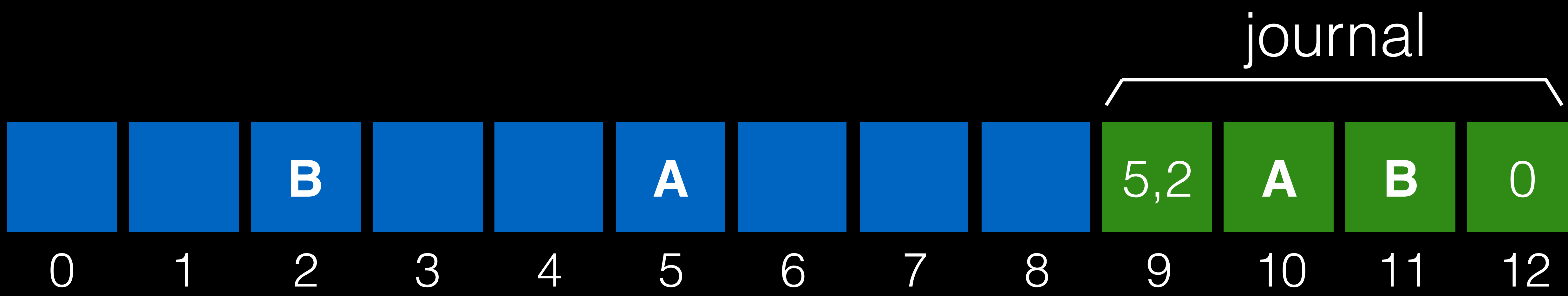
transaction: write A to block 5; write B to block 2

New Layout

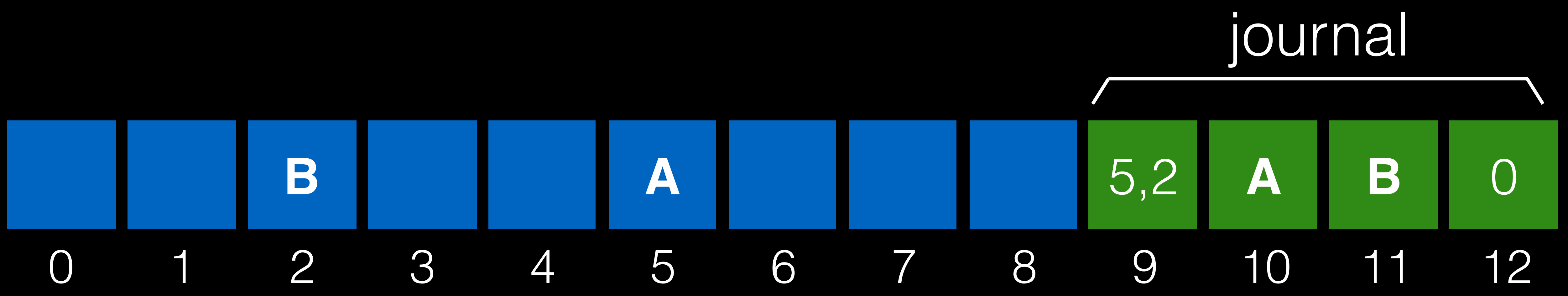


transaction: write A to block 5; write B to block 2

New Layout

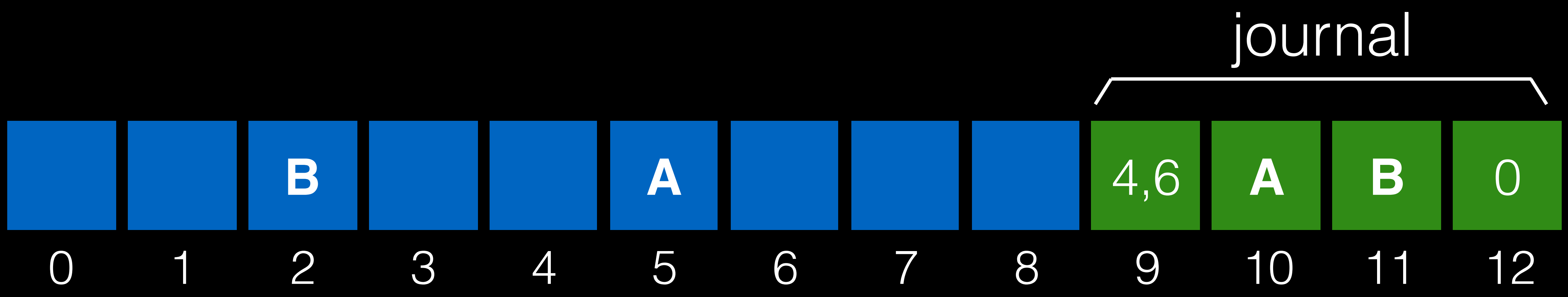


New Layout



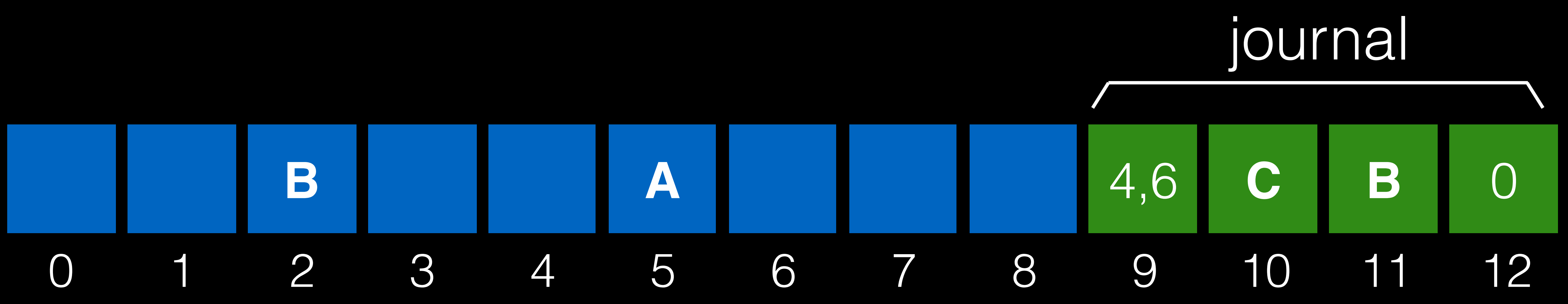
transaction: write C to block 4; write T to block 6

New Layout



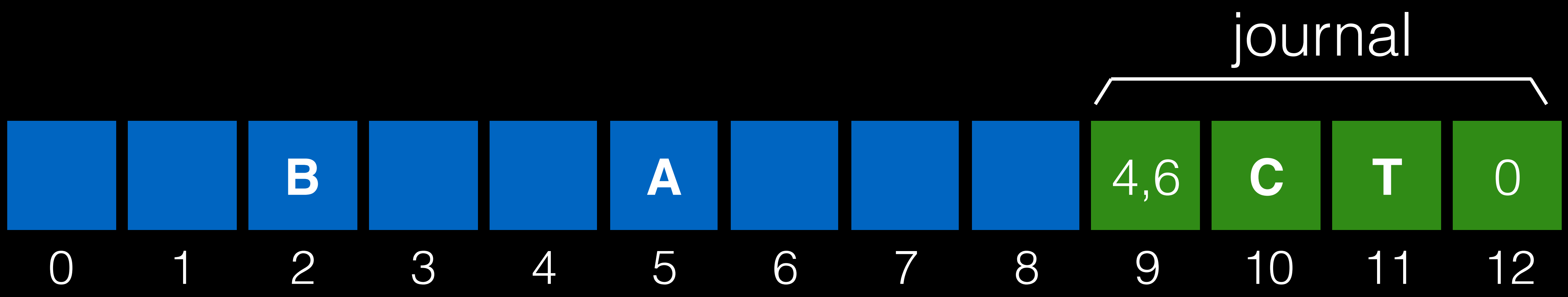
transaction: write C to block 4; write T to block 6

New Layout



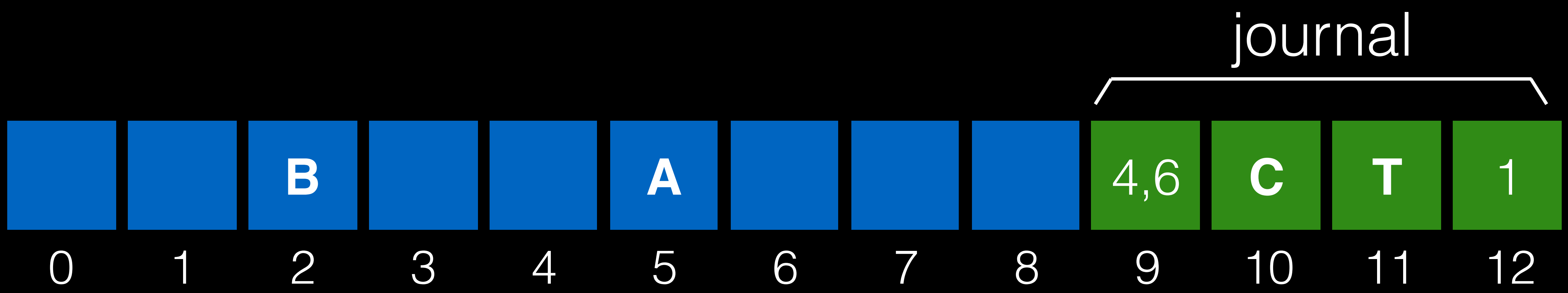
transaction: write C to block 4; write T to block 6

New Layout



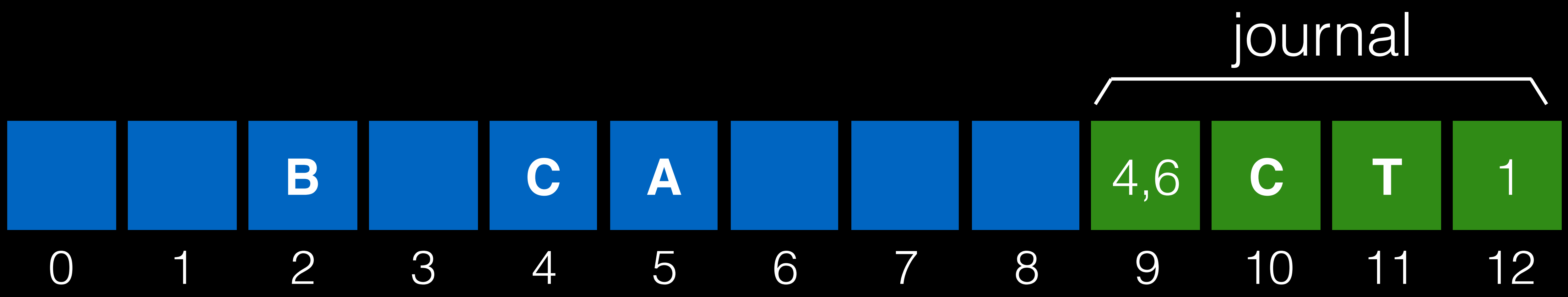
transaction: write C to block 4; write T to block 6

New Layout



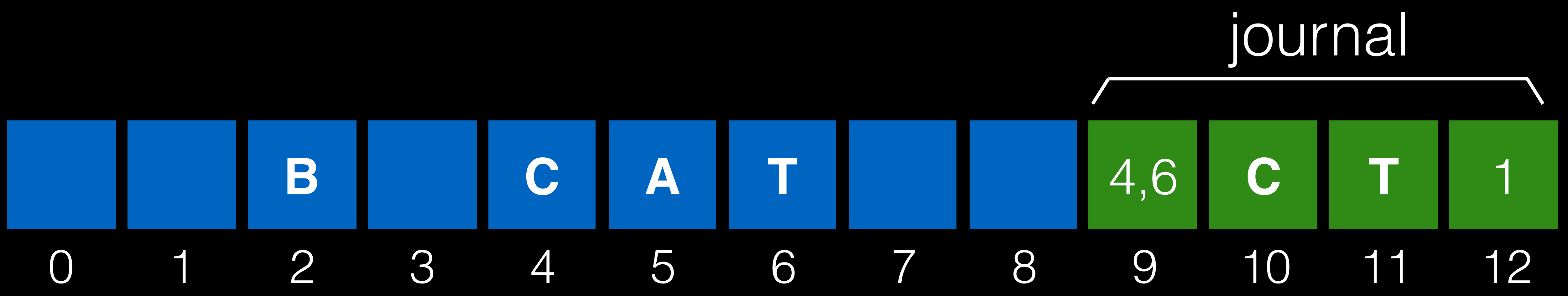
transaction: write C to block 4; write T to block 6

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

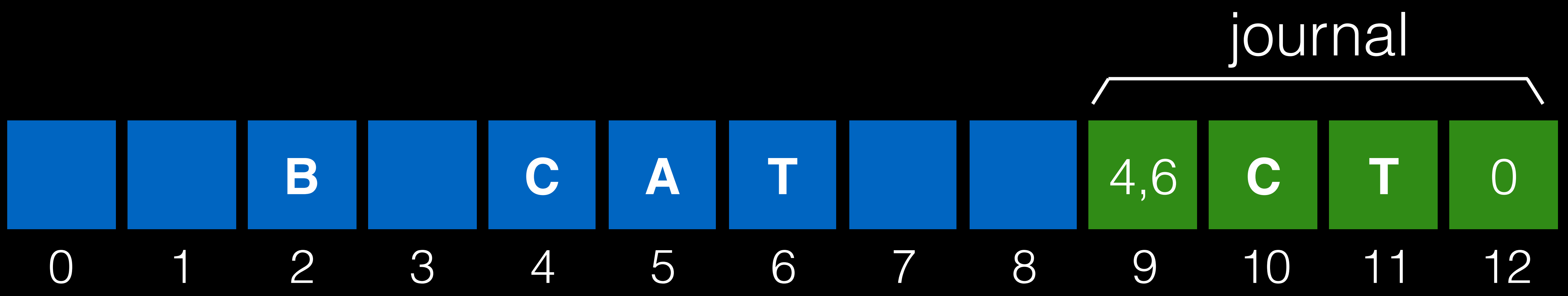
New Layout

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.

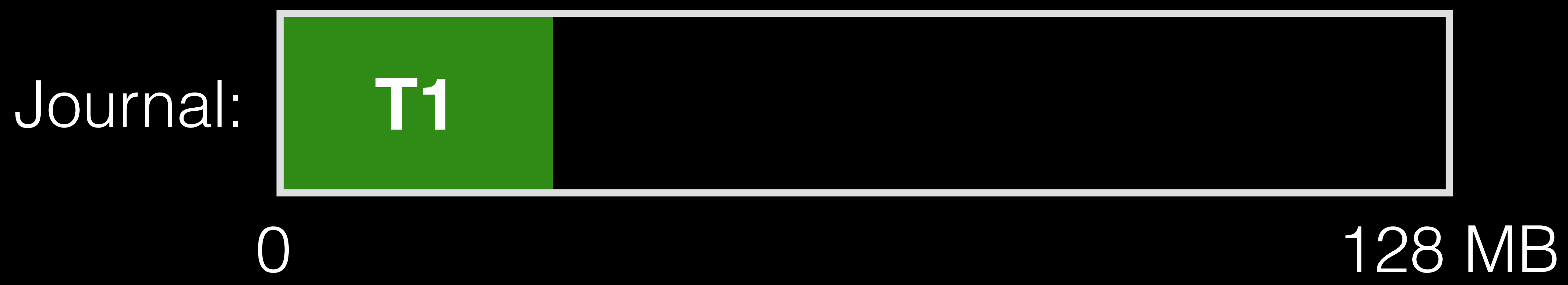
Solution: keep many transactions for un-checkpointed data.

Circular Buffer

Journal:

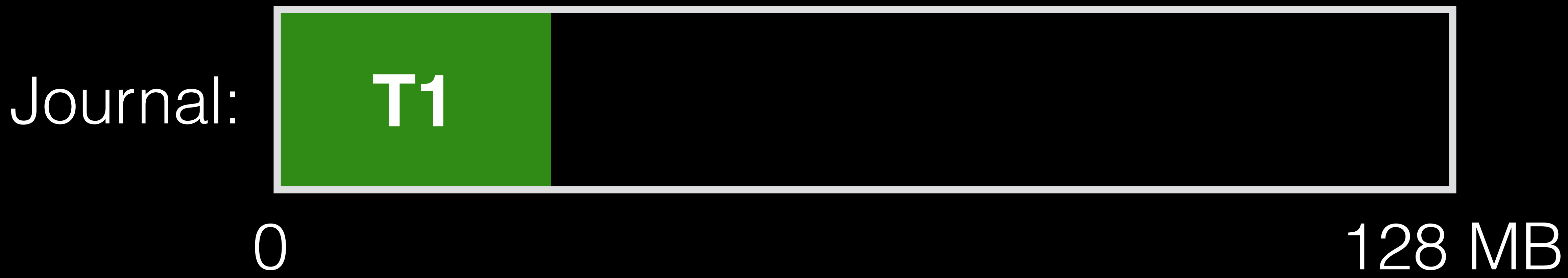
0 128 MB

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

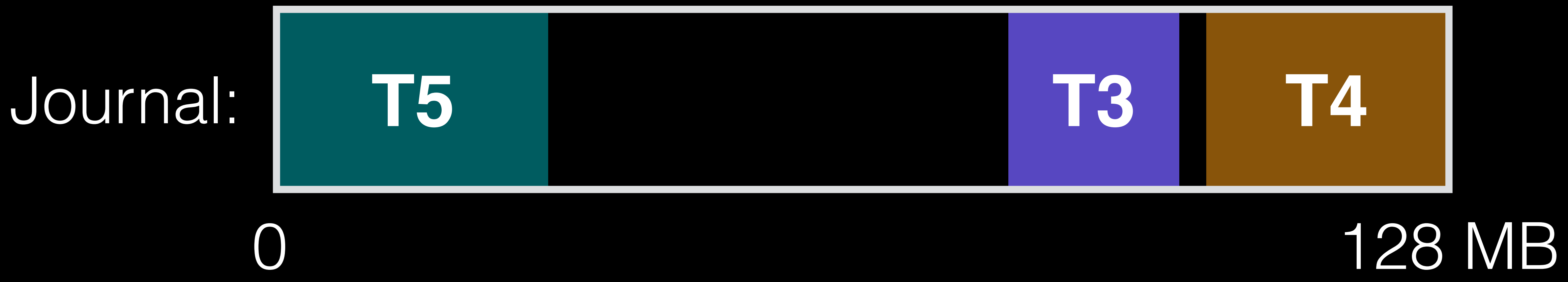
Circular Buffer

Circular Buffer

transaction!

Circular Buffer

Circular Buffer

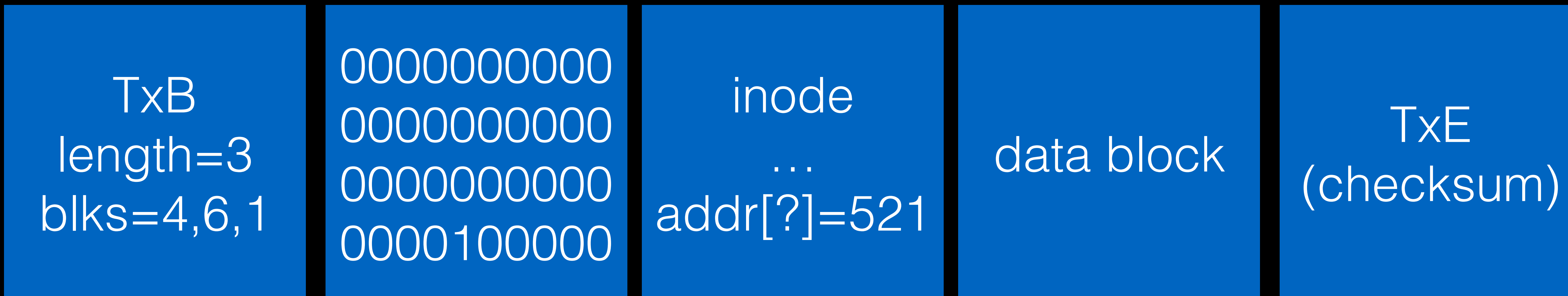


checkpoint and cleanup

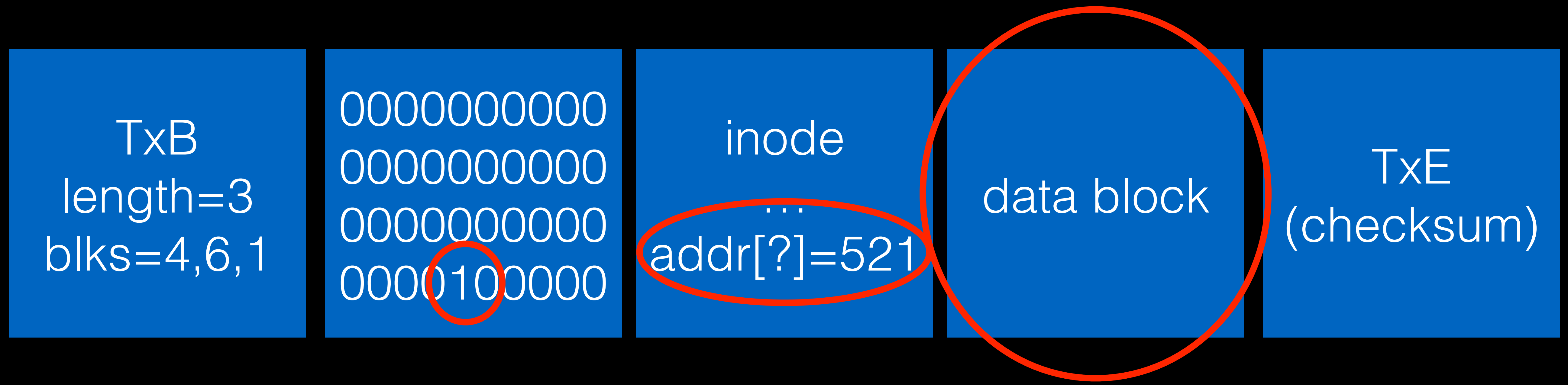
Optimizations

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

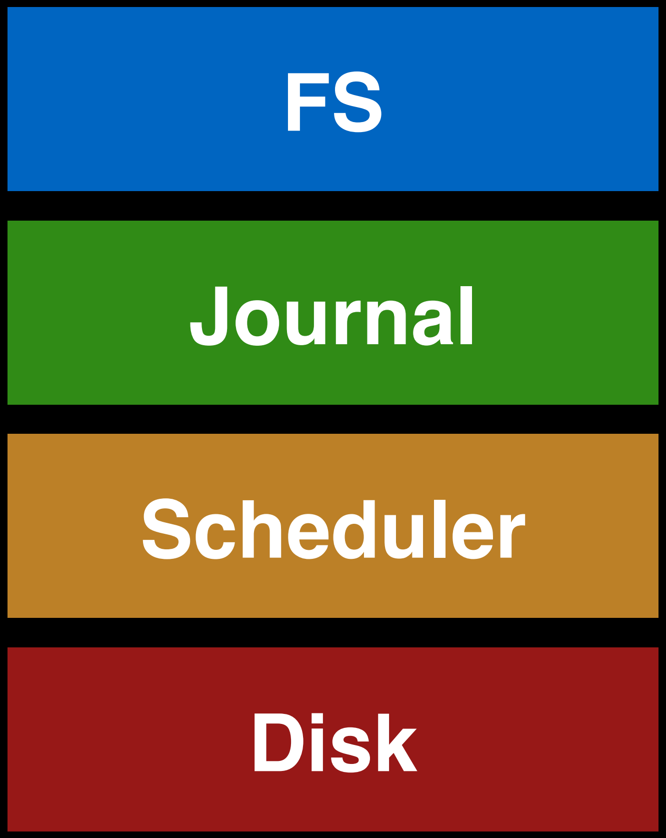
1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal

File System Integration

How should FS use journal?

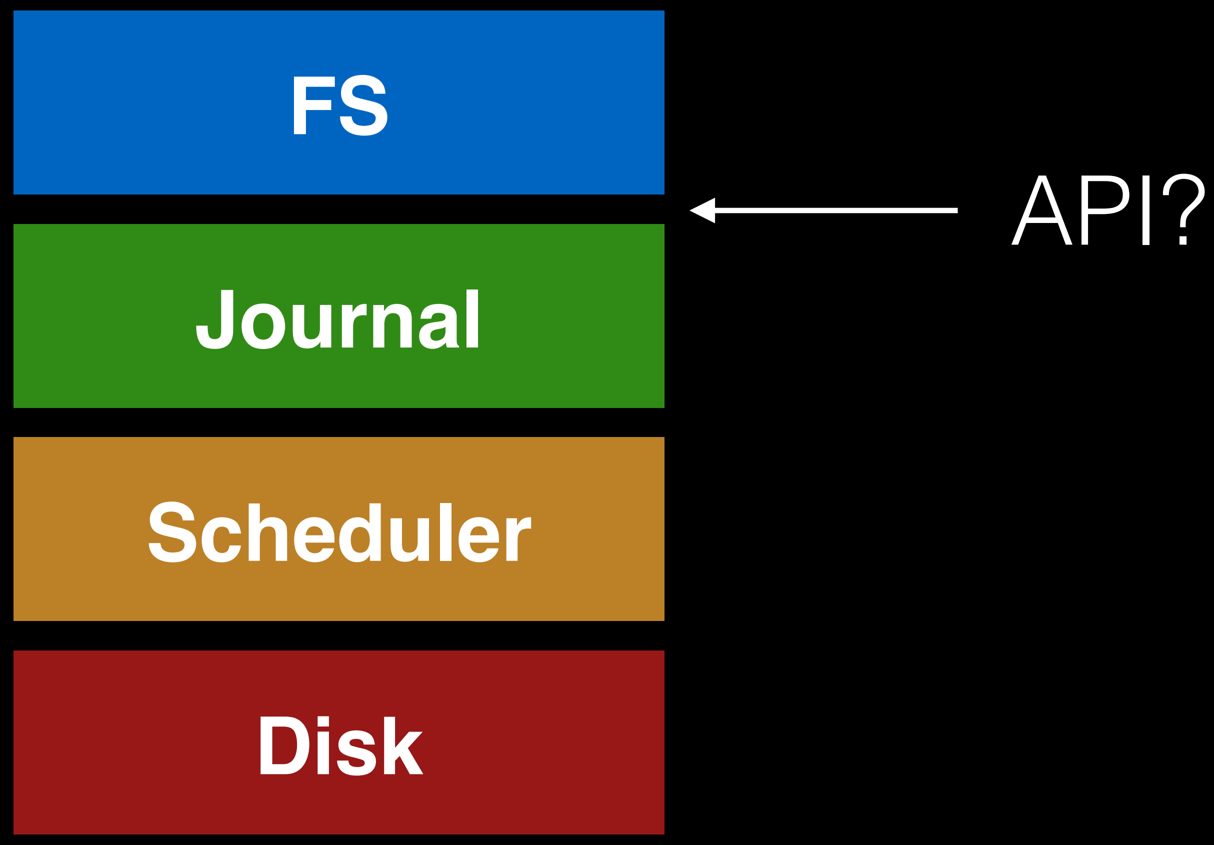
File System Integration

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

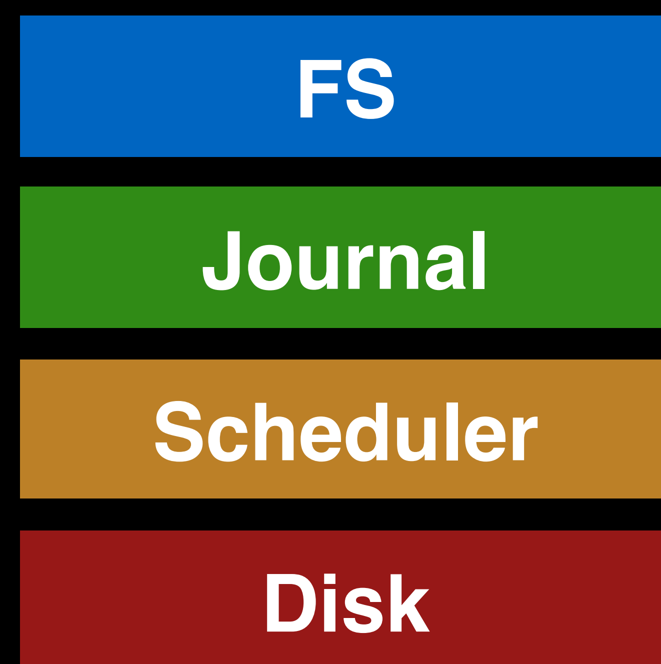
```
h = getHandle();
writeBlk(h, blknum, data);
finishHandle(h);
```

Blocks in the same handle must be written atomically.

File System Integration

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

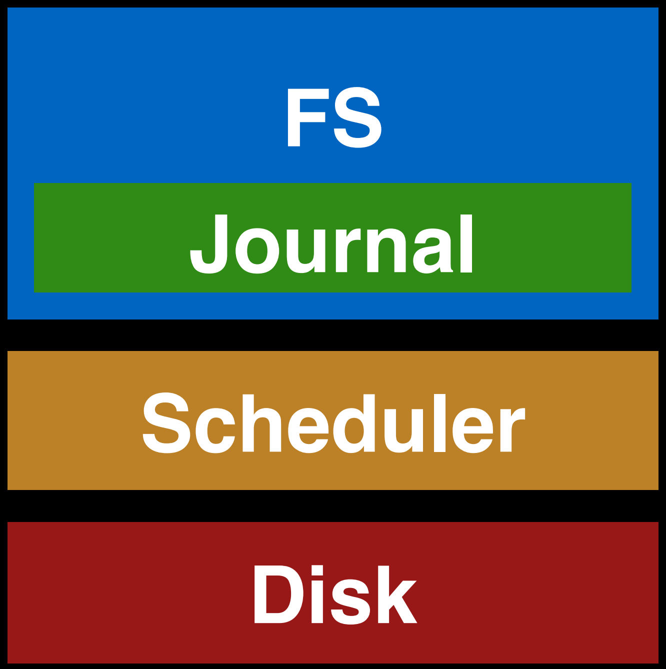
If we want to only journal FS metadata, we need tighter integration.



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?

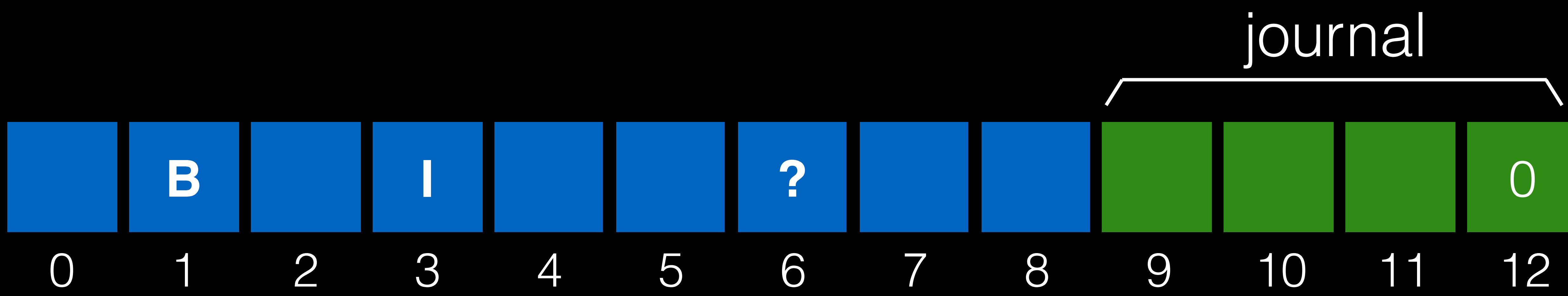
Writeback Journal

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.

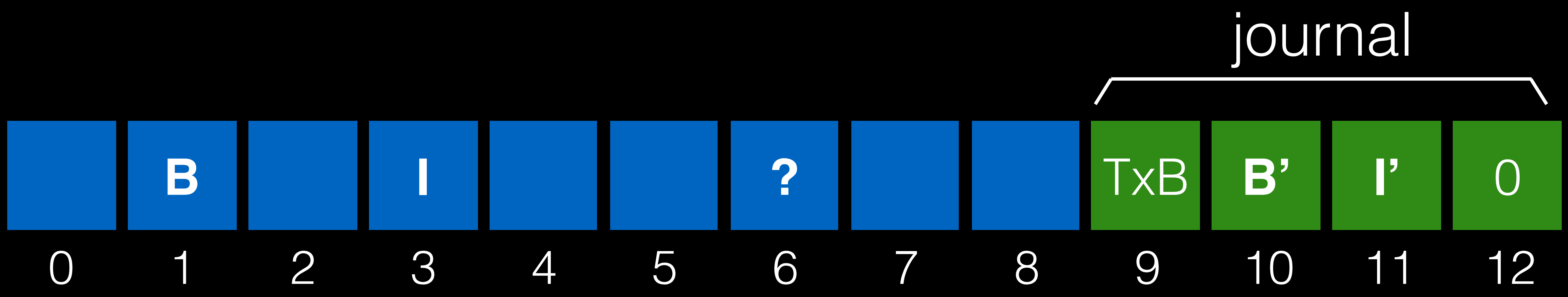
What is the worst type of garbage we could get? How to avoid?

Writeback Journal



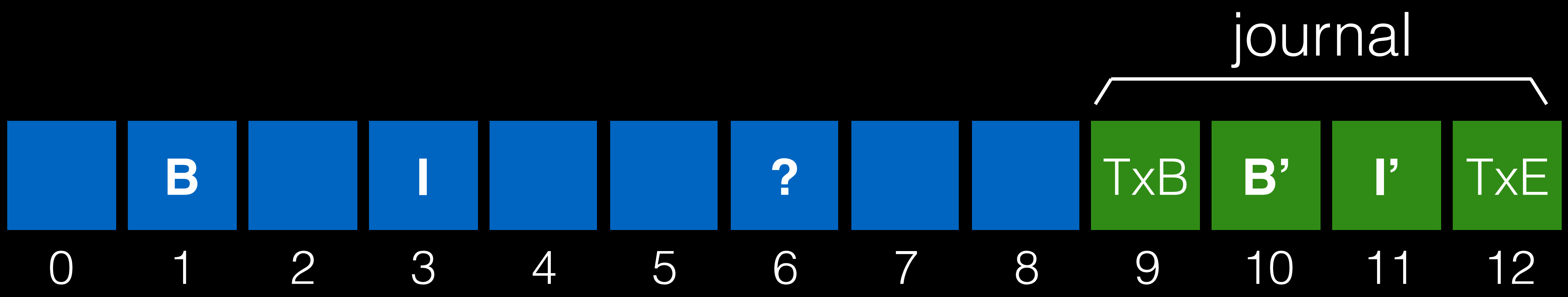
transaction: append to inode I

Writeback Journal



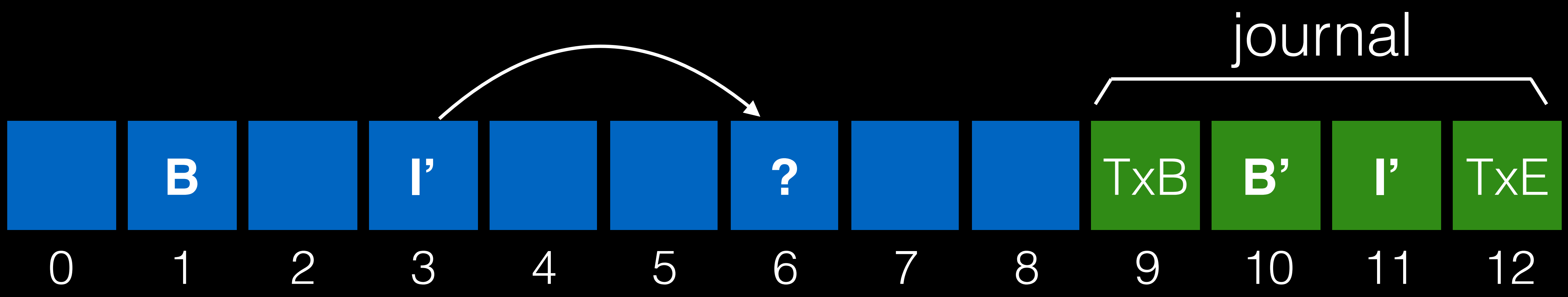
transaction: append to inode I

Writeback Journal



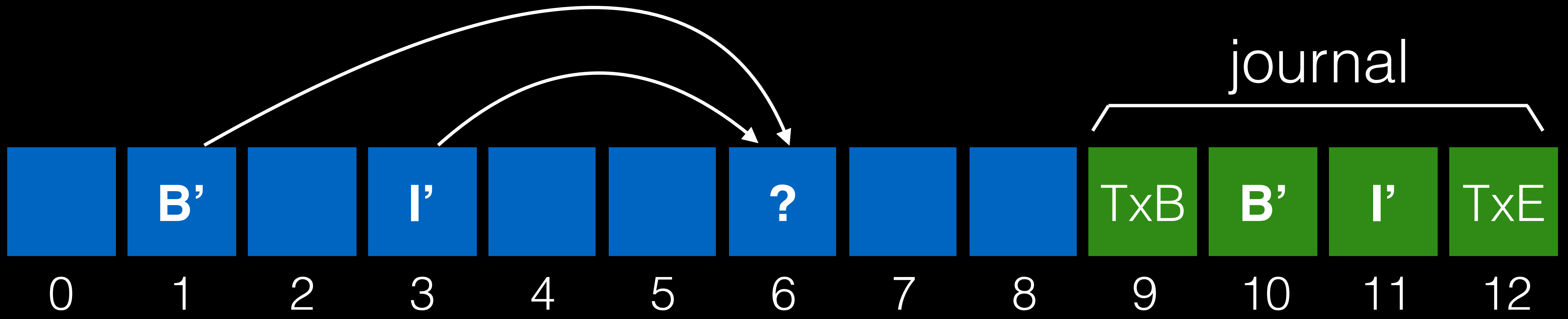
transaction: append to inode I

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.

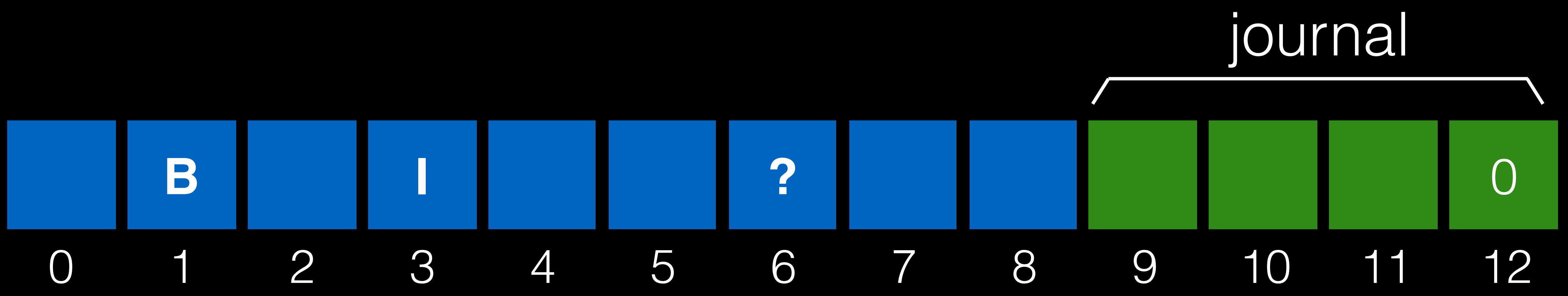
But write data before the transaction.

May still get scrambled data on update.

But appends will always be good.

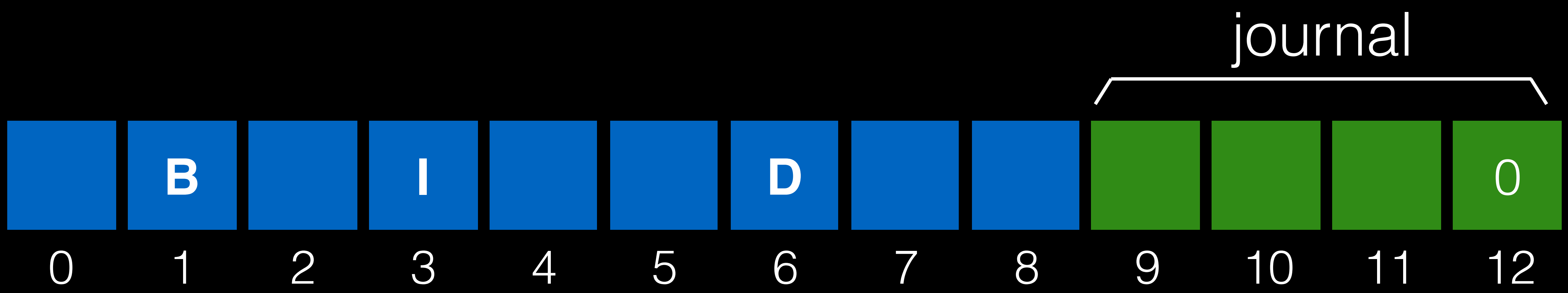
No leaks of sensitive data!

Ordered Journal



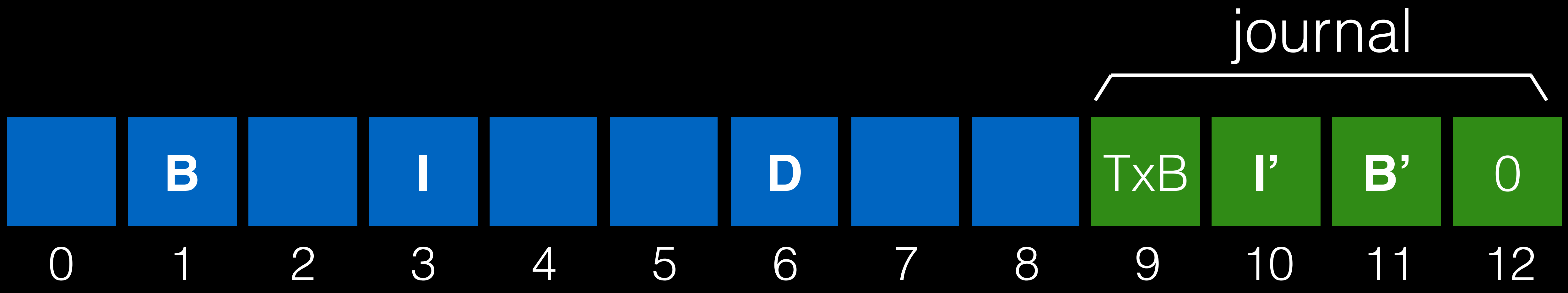
transaction: append to inode I

Ordered Journal



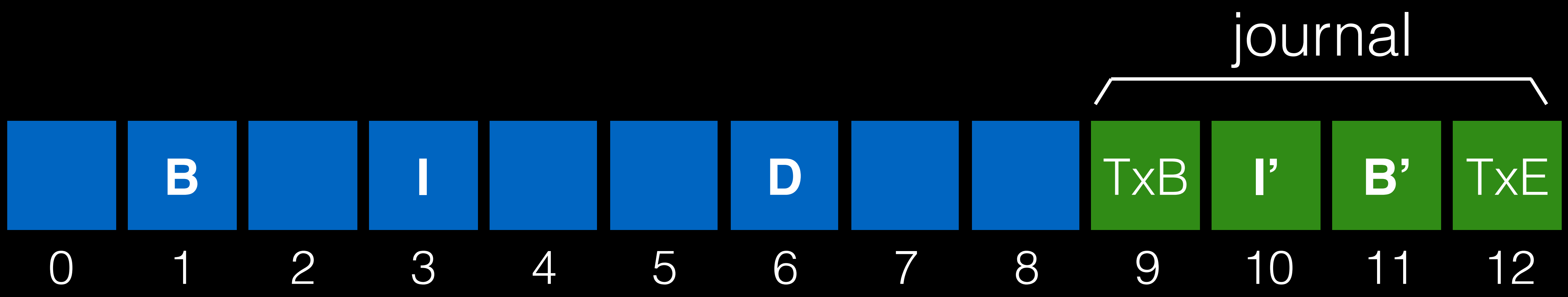
transaction: append to inode I

Ordered Journal



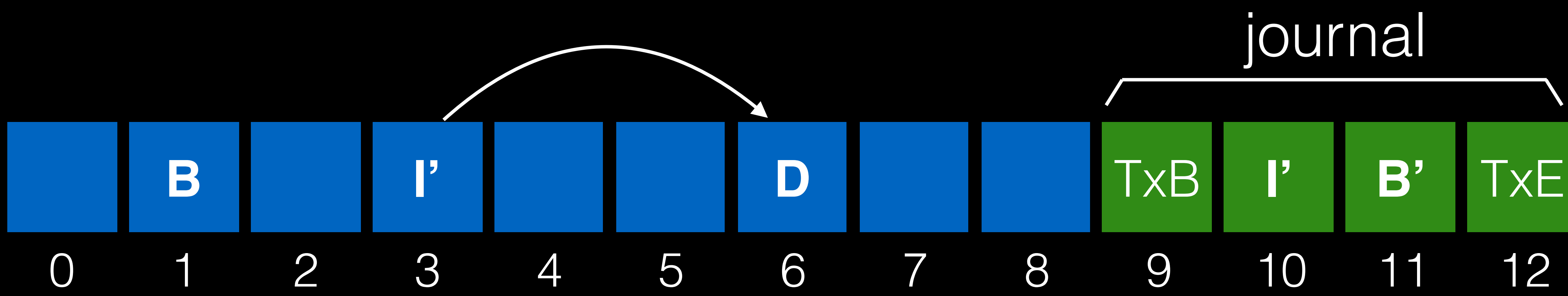
transaction: append to inode I

Ordered Journal



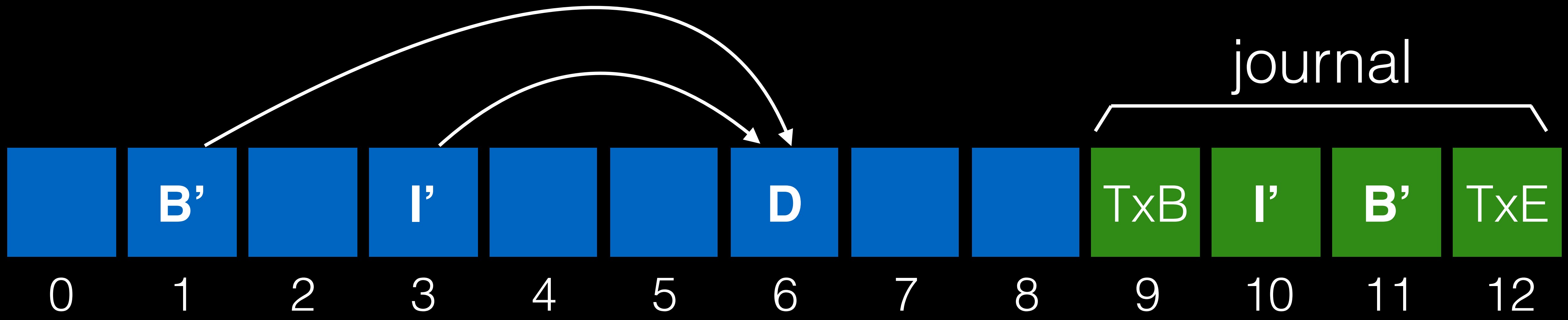
transaction: append to inode I

Ordered Journal



transaction: append to inode I

Ordered Journal



transaction: append to inode I

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.

Log-Structured File System

LFS: Log-Structured File System

Different than FFS:
- optimizes allocation for writes instead of reads

Different than Journaling:
- use copy-on-write for atomicity

Performance Goal

Ideal: use disk purely sequentially.

Performance Goal

Ideal: use disk purely sequentially.

Hard for reads -- why?

Easy for writes -- why?

Performance Goal

Ideal: use disk purely sequentially.

Hard for reads -- why?
- user might read files X and Y not near each other

Easy for writes -- why?
- can do all writes near each other to empty space

Observations

Memory sizes are growing (so cache more reads).

Growing gap between sequential and random I/O performance.

Existing file systems not RAID-aware (don't avoid small writes).

LFS Strategy

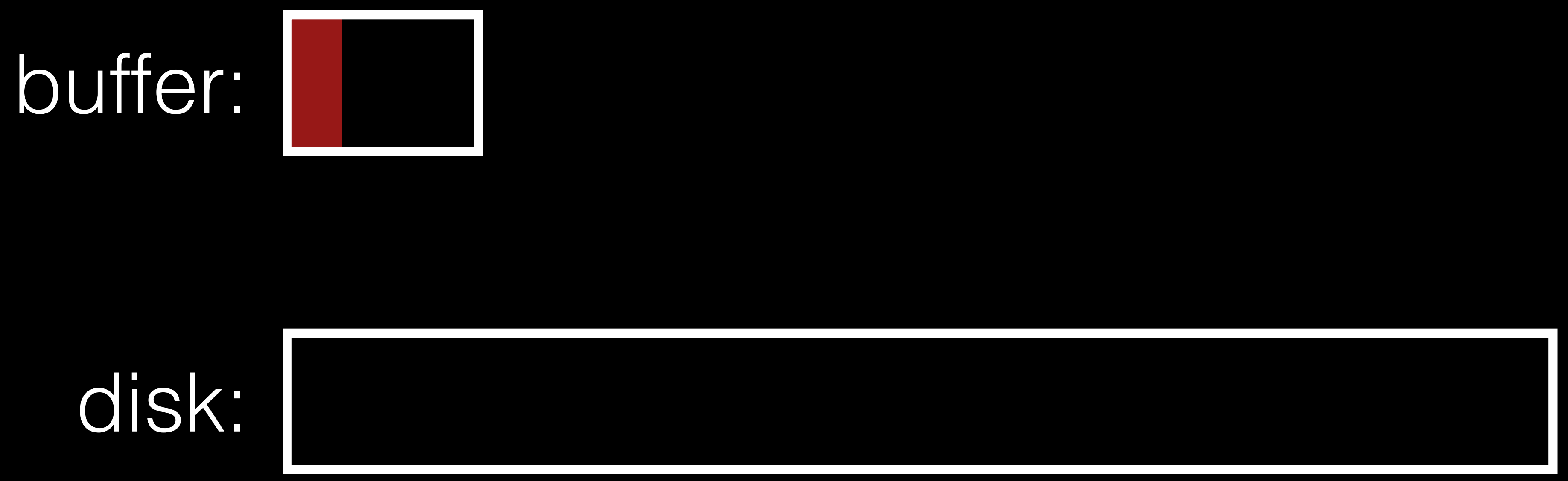
Just write all data sequentially to new segments.

Never overwrite, even if that means we leave behind old copies.

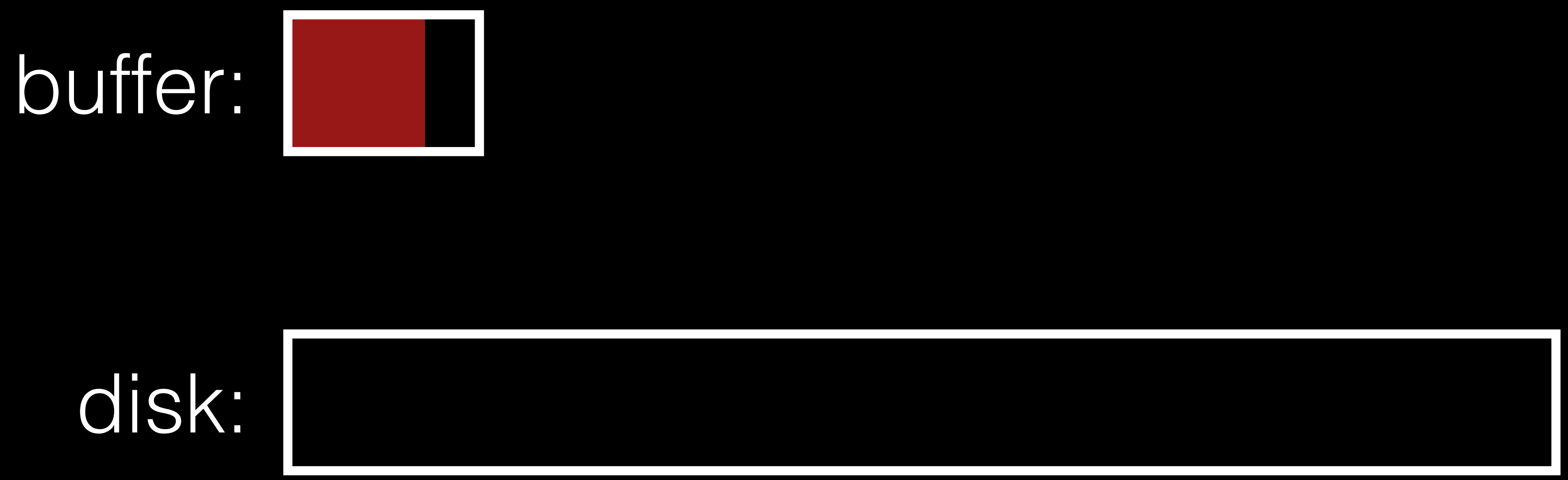
Buffer writes until we have enough data.

Big Picture

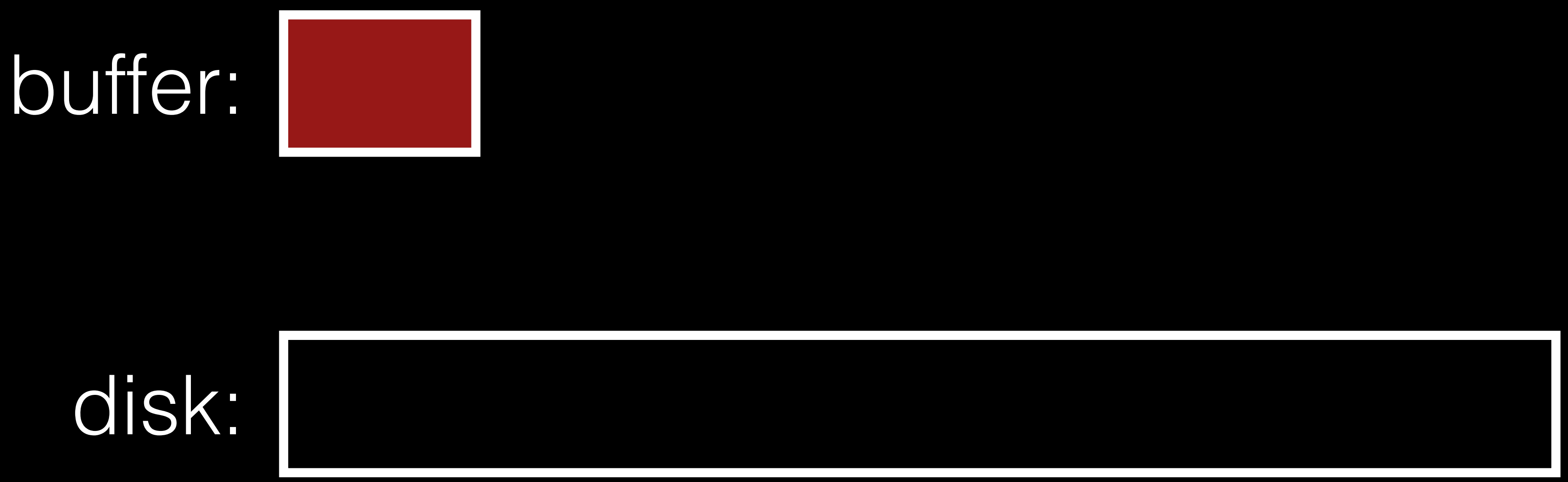
Big Picture



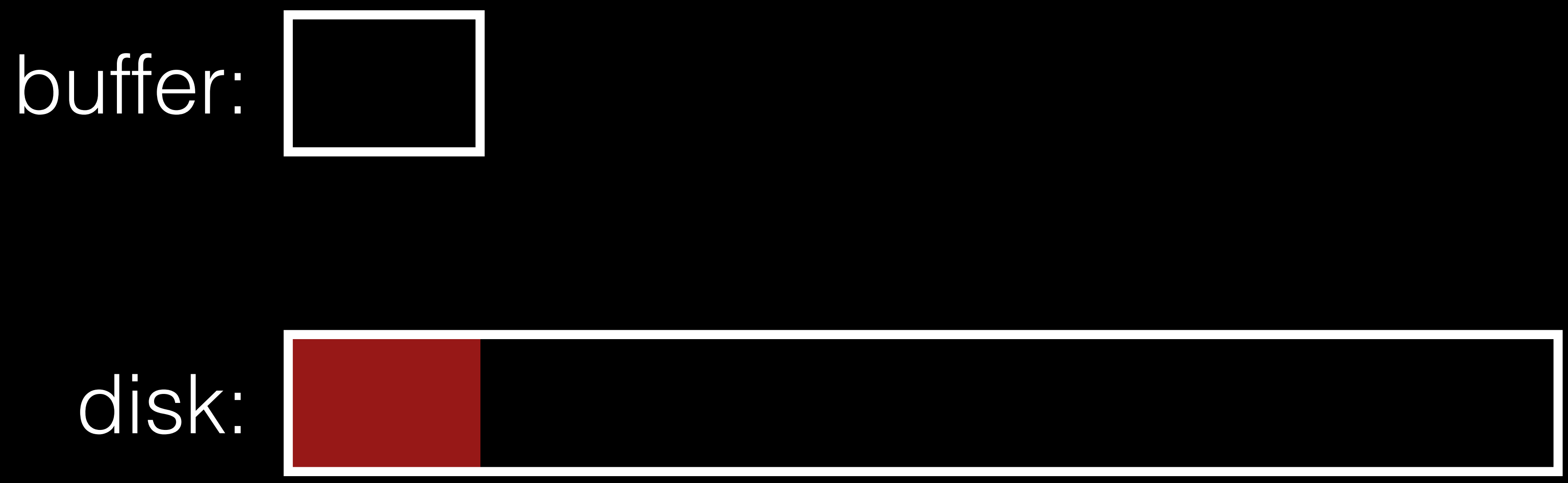
Big Picture



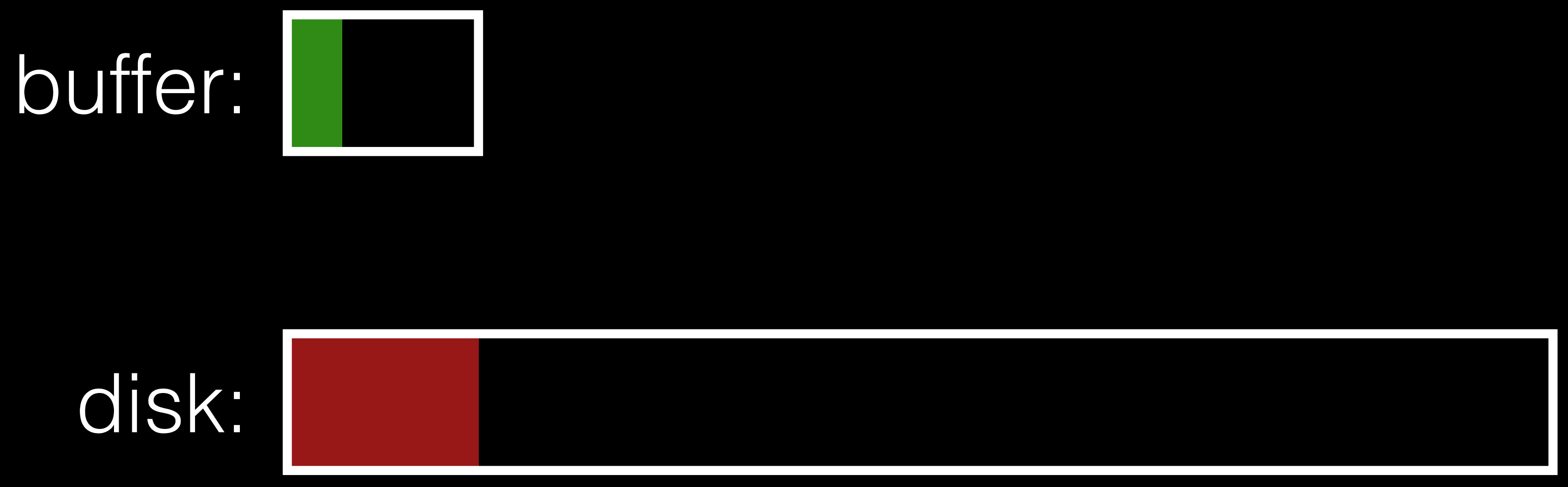
Big Picture



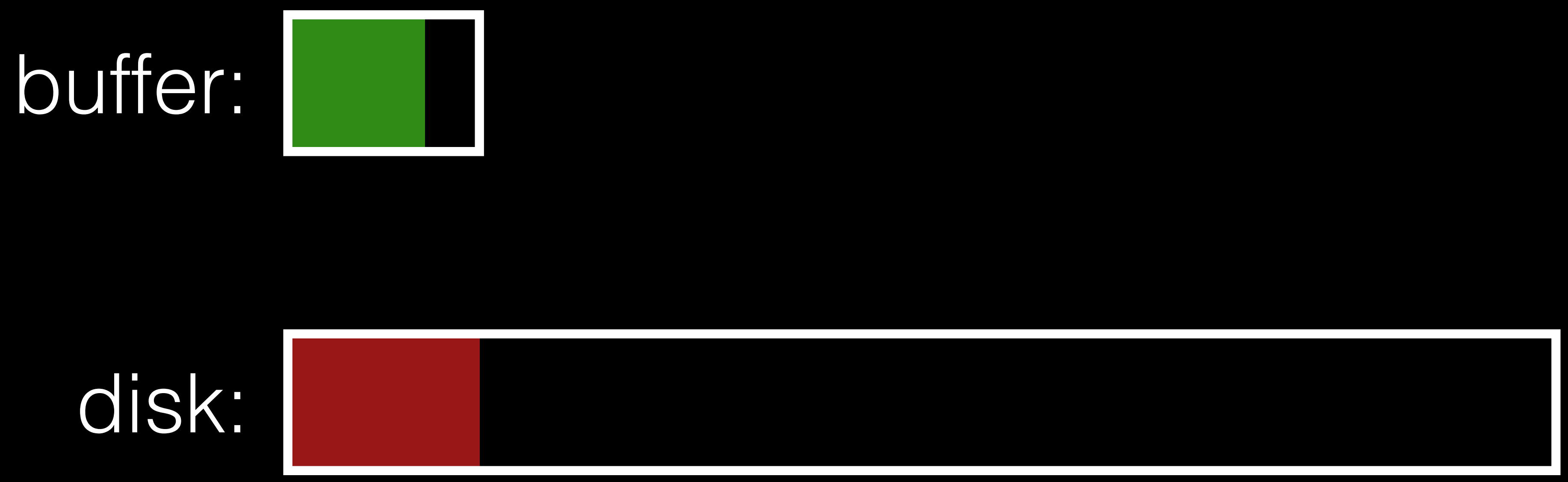
Big Picture



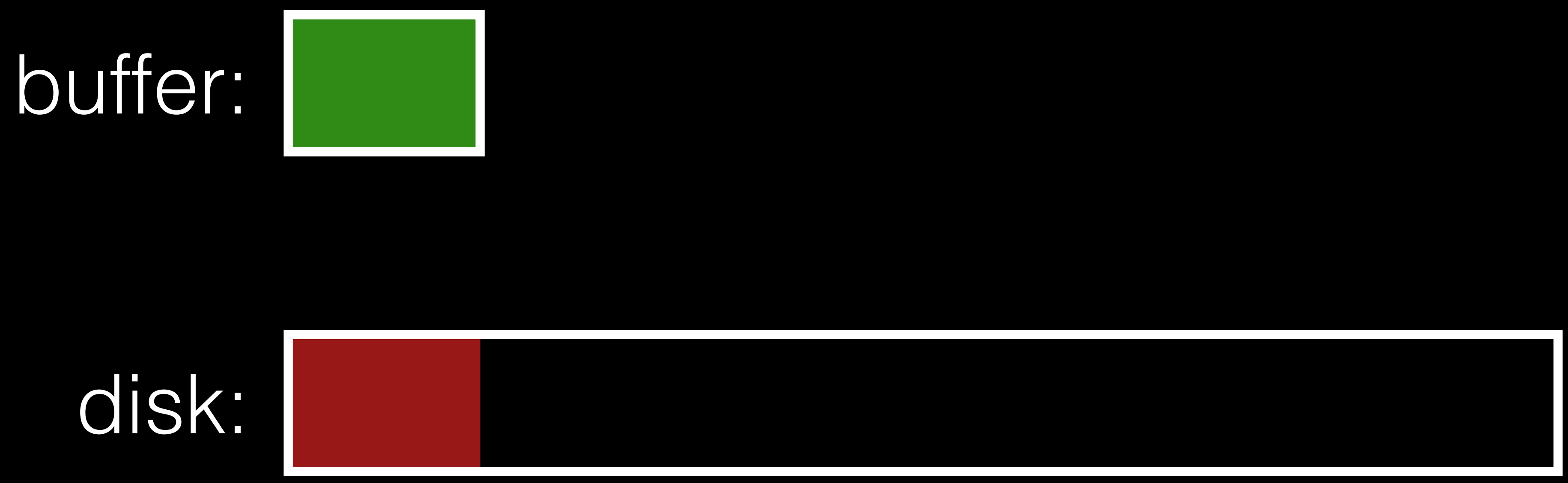
Big Picture



Big Picture



Big Picture



Big Picture

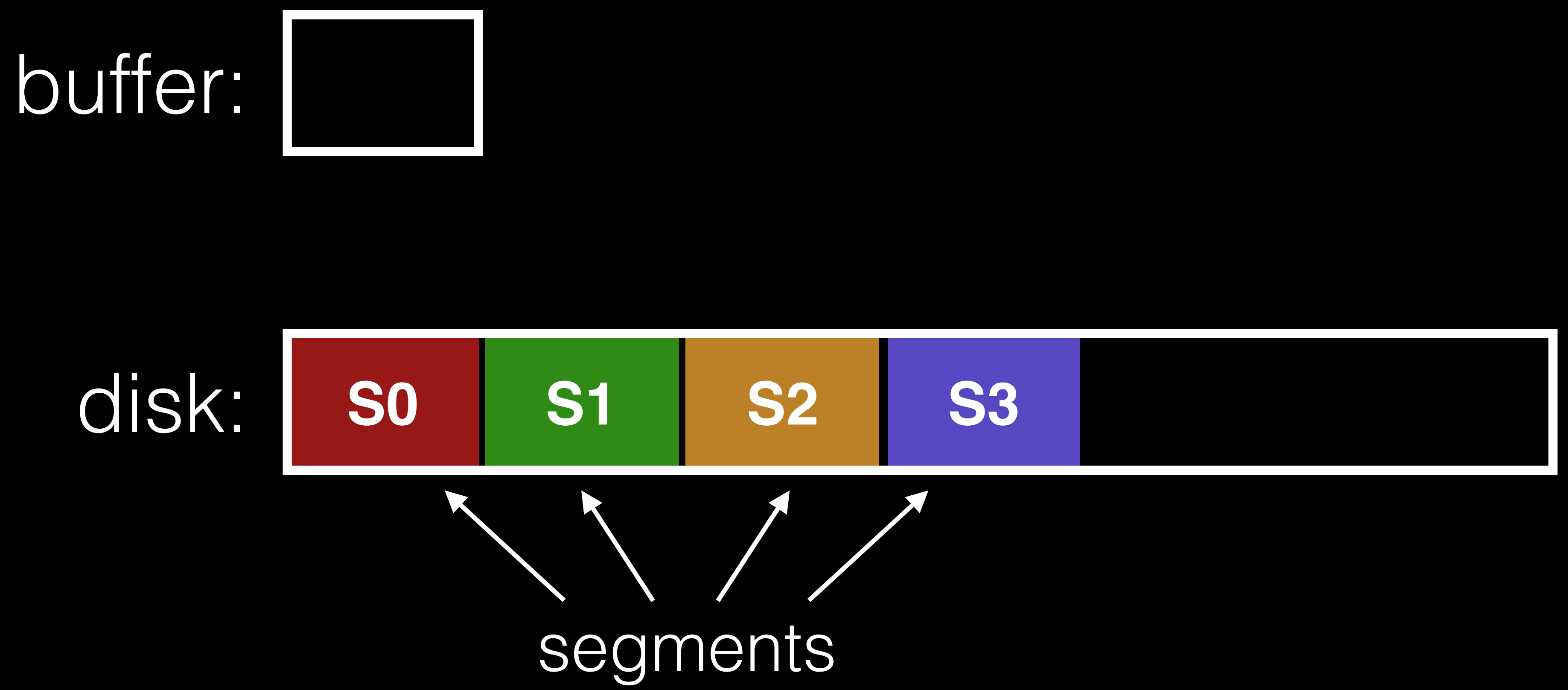
Big Picture

Big Picture

Big Picture

Big Picture

Big Picture



Data Structures (attempt 1)

What can we get rid of from FFS?

Data Structures (attempt 1)

What can we get rid of from FFS?

- allocation structs: data + inode bitmaps

Data Structures (attempt 1)

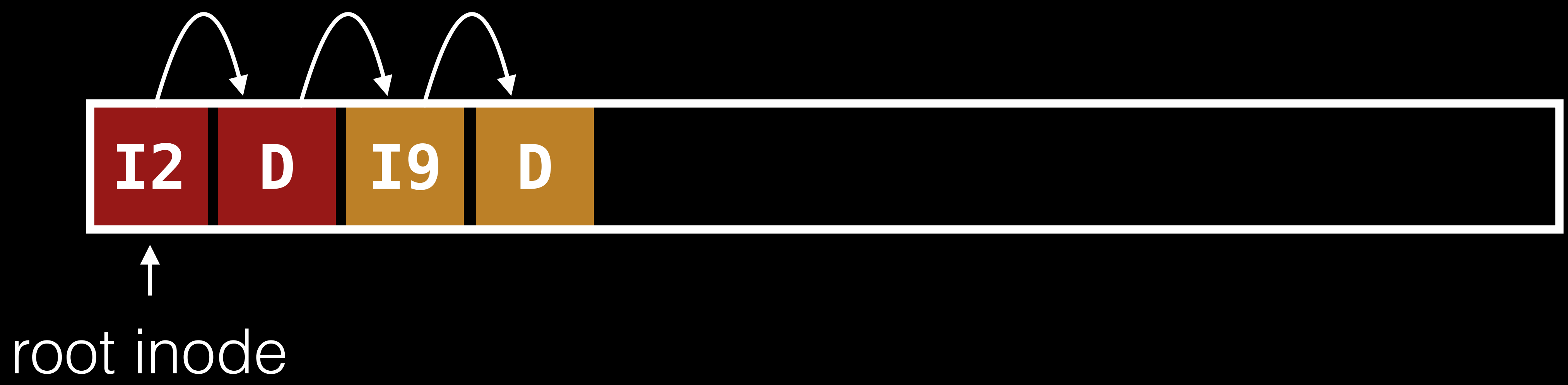
What can we get rid of from FFS?

- allocation structs: data + inode bitmaps

Inodes are no longer at fixed offset.

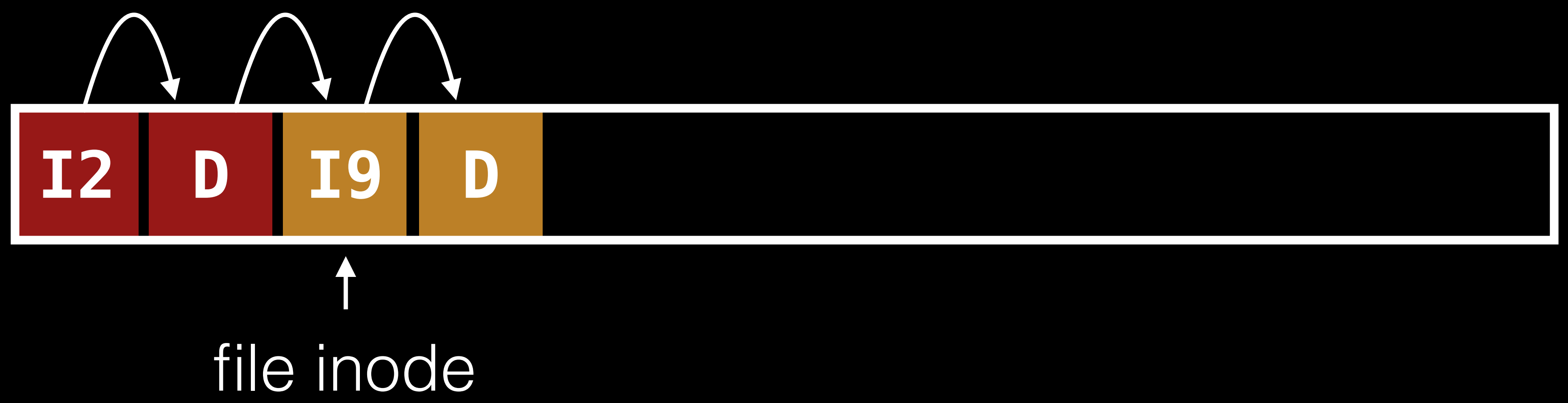
- use offset instead of table index for name.
- note: upon inode update, inode number changes.

Overwrite Data in /file.txt

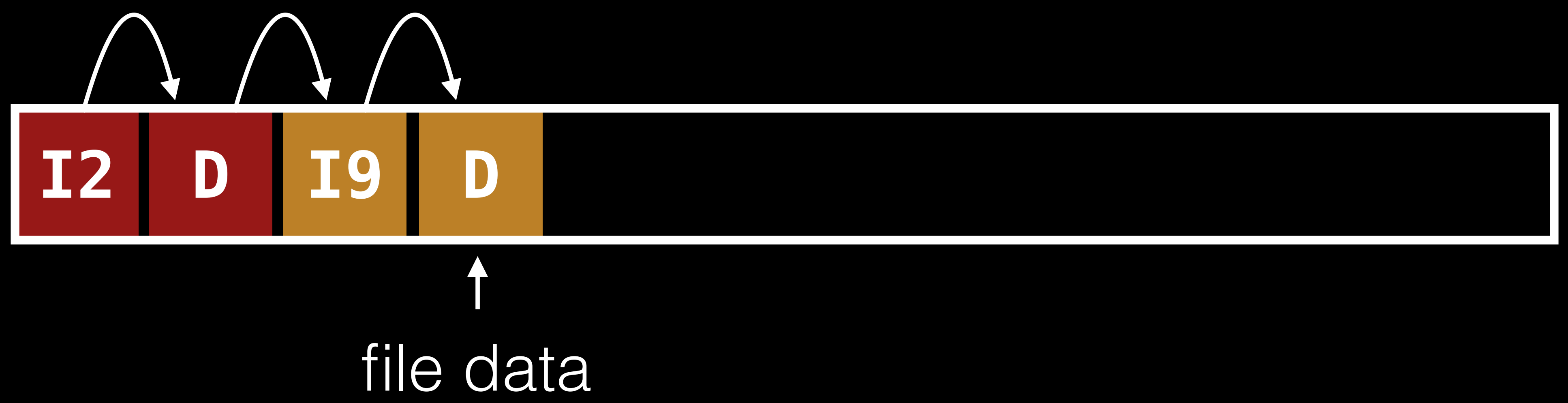


Overwrite Data in /file.txt

Overwrite Data in /file.txt



Overwrite Data in /file.txt



Overwrite Data in /file.txt

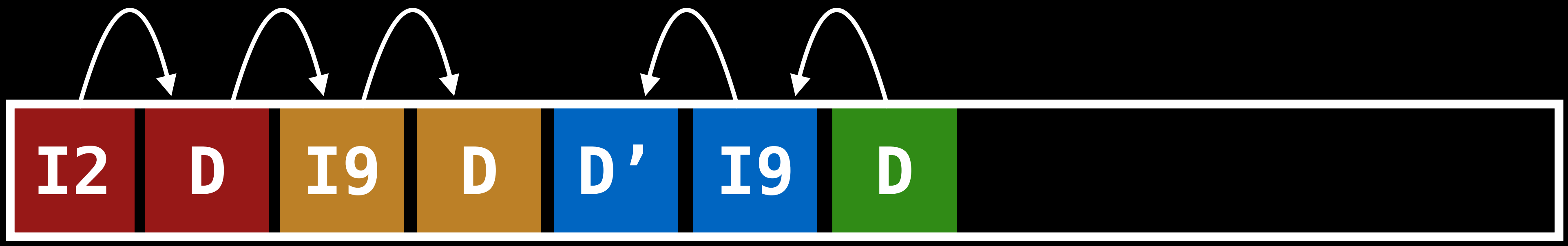
Overwrite Data in /file.txt

Overwrite Data in /file.txt

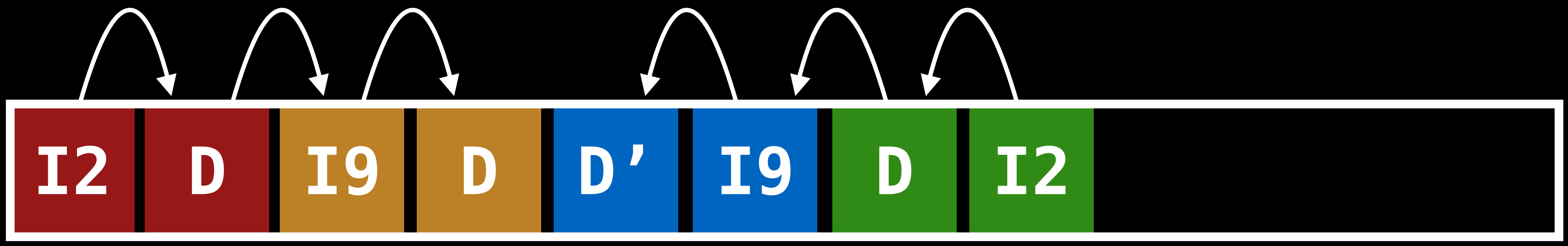
NO! This would be a random write.

Overwrite Data in /file.txt

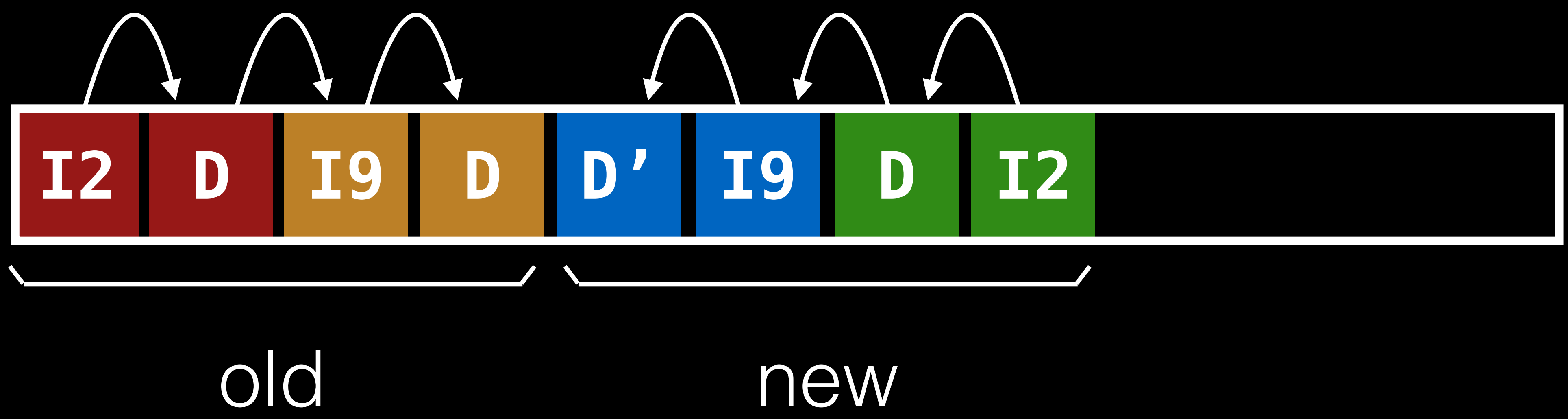
Overwrite Data in /file.txt



Overwrite Data in /file.txt



Overwrite Data in /file.txt



Inode Numbers

Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Inode Numbers

Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Solution: keep inode numbers constant. Don’t base on offset.

Inode Numbers

Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Solution: keep inode numbers constant. Don't base on offset.

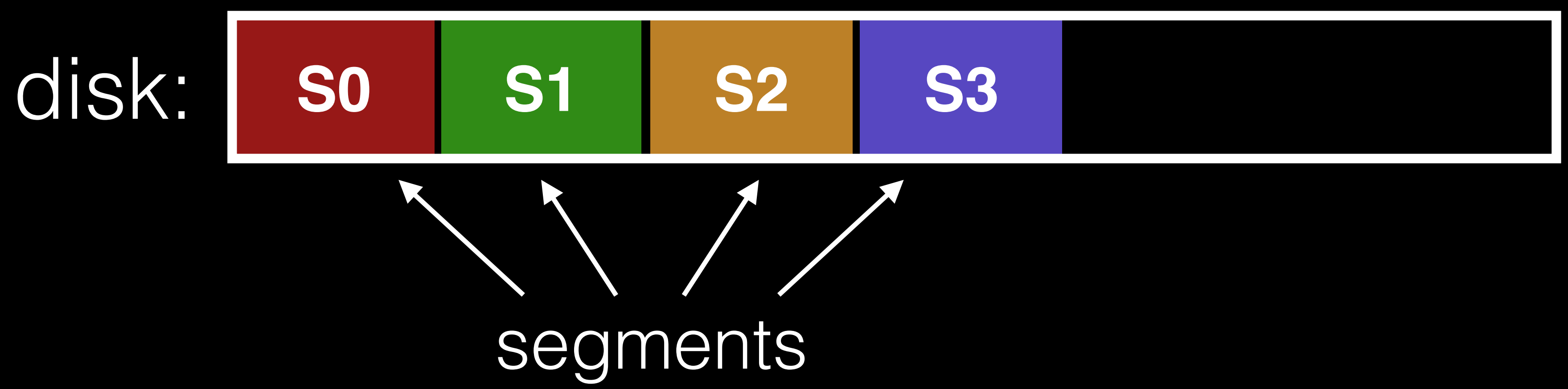
Before we found inodes with math. How now?

Data Structures (attempt 2)

What can we get rid of from FFS?
- allocation structs: data + inode bitmaps

Inodes are no longer at fixed offset.
- use imap struct to map number => inode.

imap



imap

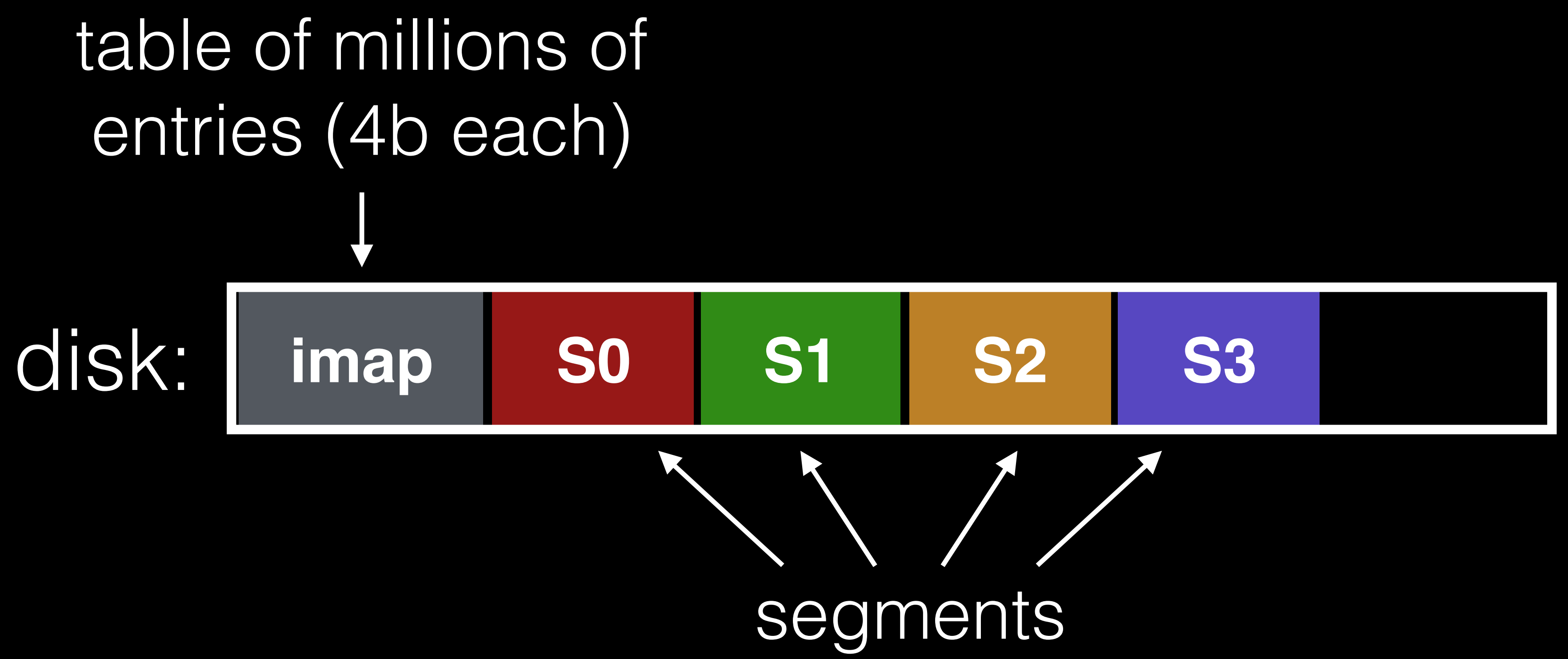
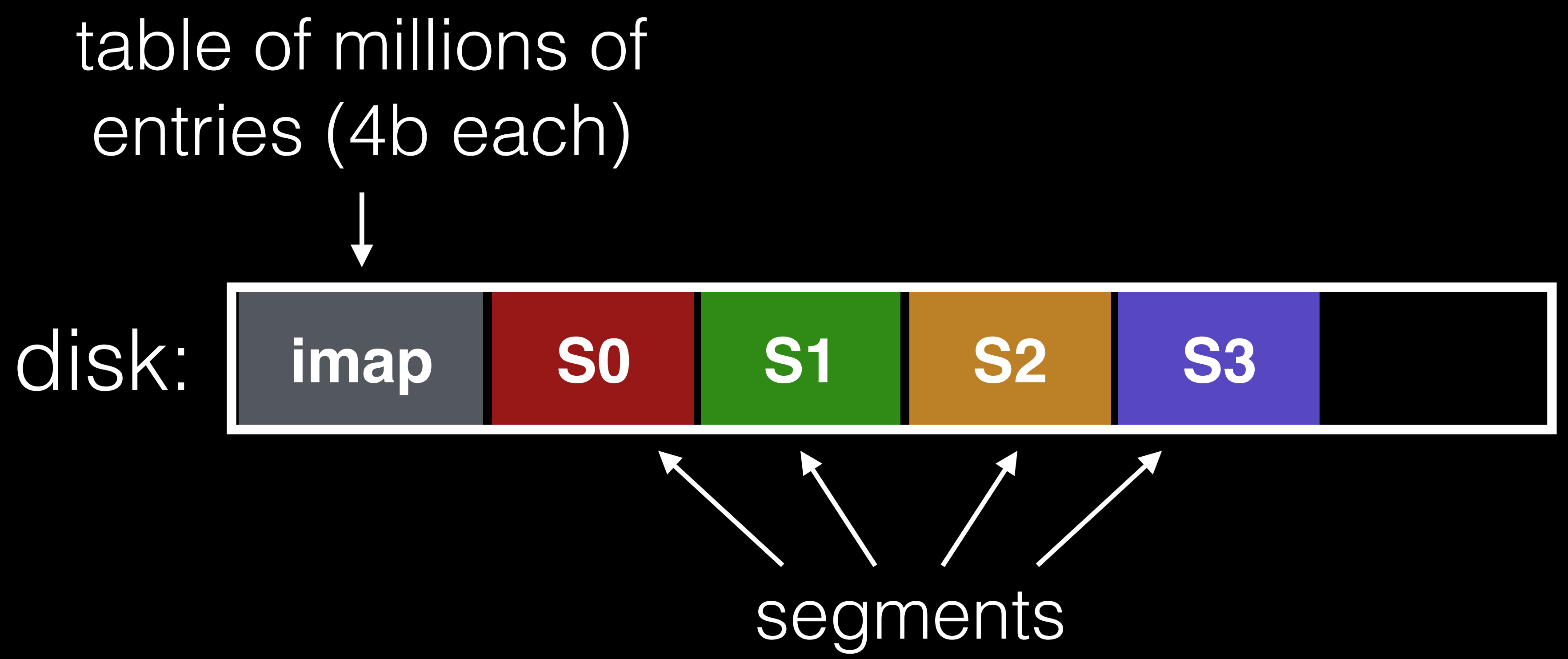


table of millions of entries (4b each)
disk:
imap
S0
S1
S2
S3
segments

imap



problem: updating imap each time makes I/O random.

Problem

Dilemma:

1. imap too big to keep in memory
2. don’t want to use random writes for imap

Attempt 3

Dilemma:

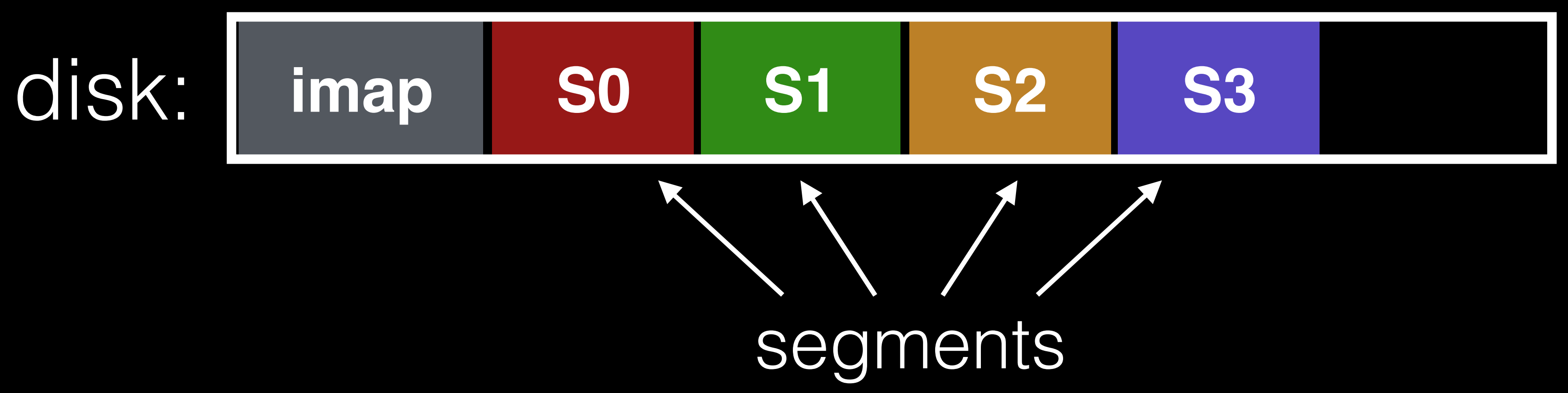
1. imap too big to keep in memory
2. don't want to use random writes for imap

Solution:

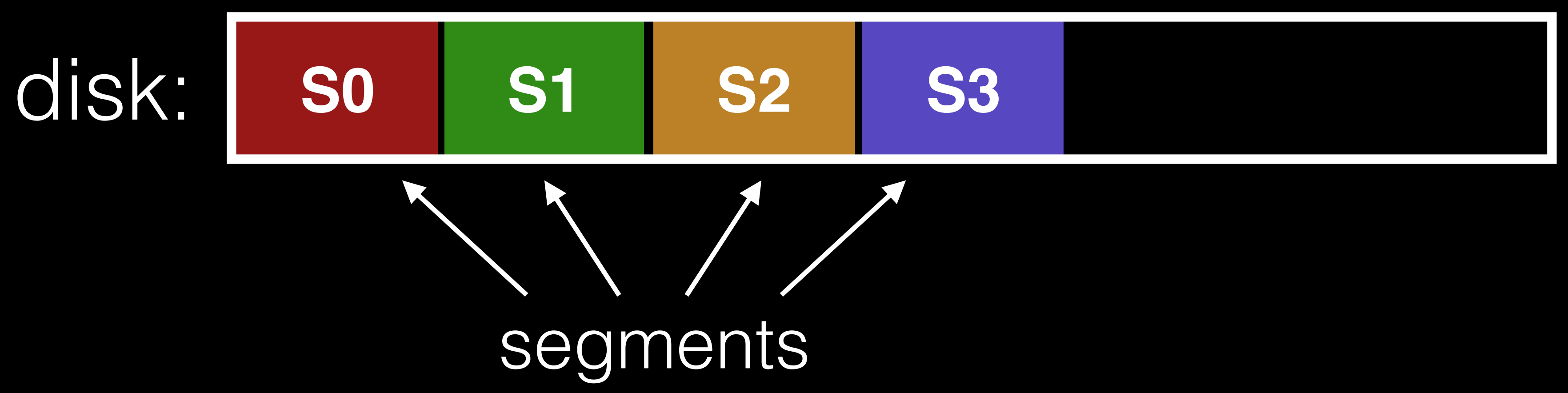
write imap in segments.

keep pointers to pieces of imap in memory.

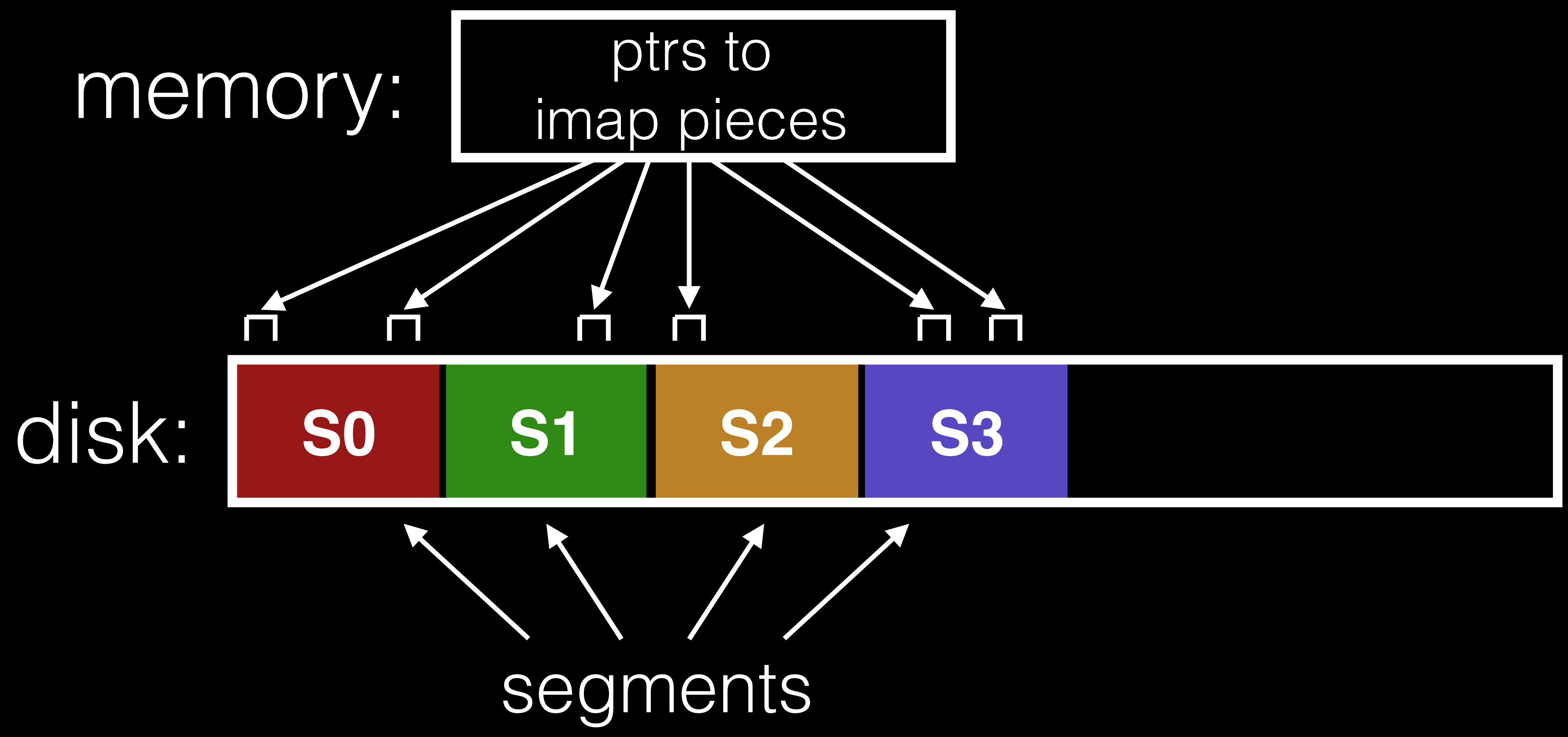
imap



imap



imap

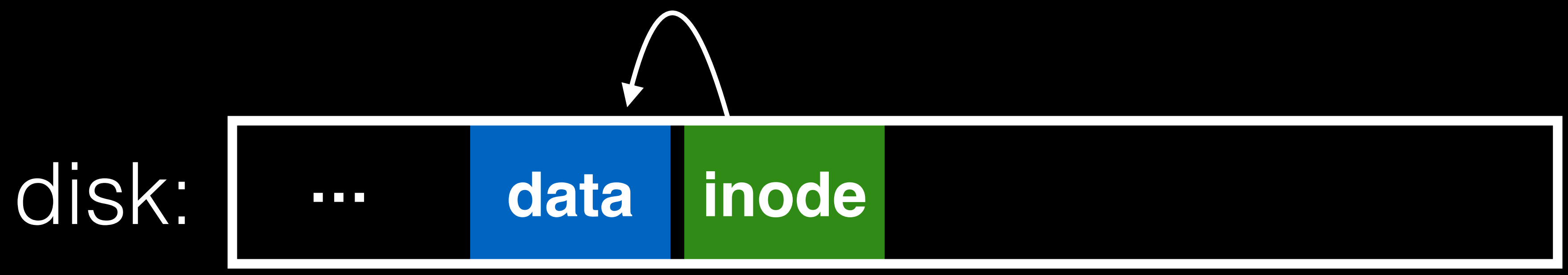


Example Write

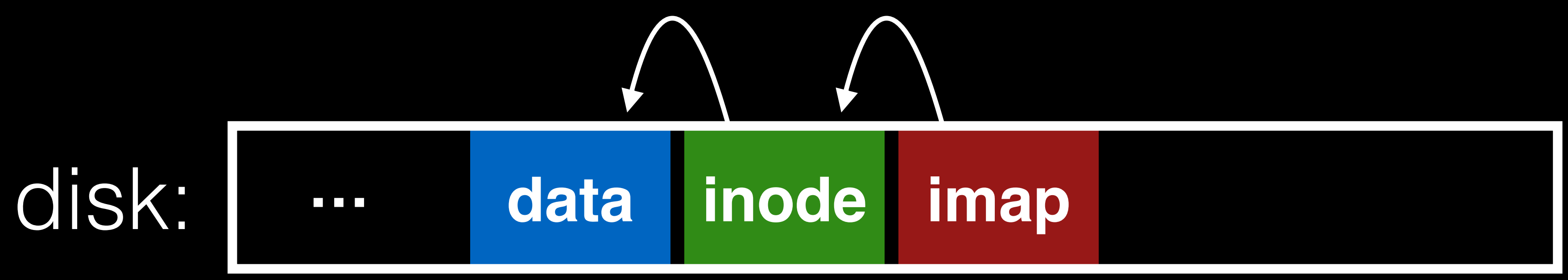
disk: | ... |

Example Write

Example Write



Example Write



Other Issues

Crashes

Garbage Collection

Crash Recovery

Naive approach: scan entire log to reconstruct pointers to imap pieces. Slow!

Crash Recovery

Naive approach: scan entire log to reconstruct pointers to imap pieces. Slow!

Better approach: occasionally checkpoint the pointers to imap pieces on disk.

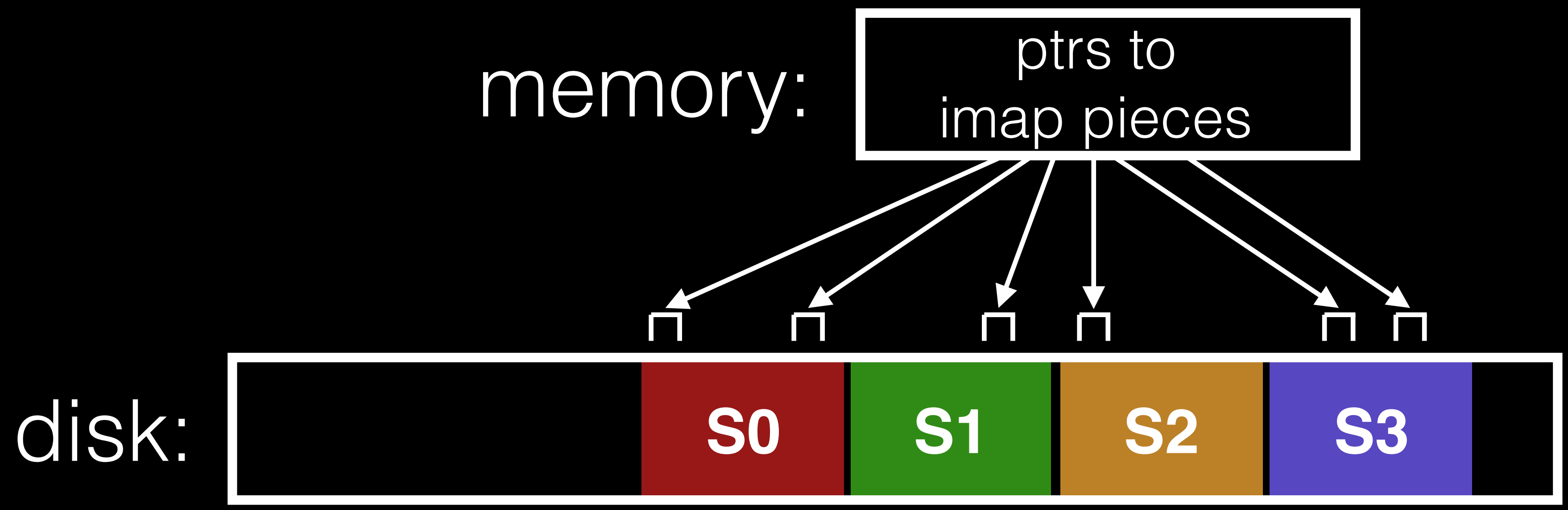
Crash Recovery

Naive approach: scan entire log to reconstruct pointers to imap pieces. Slow!

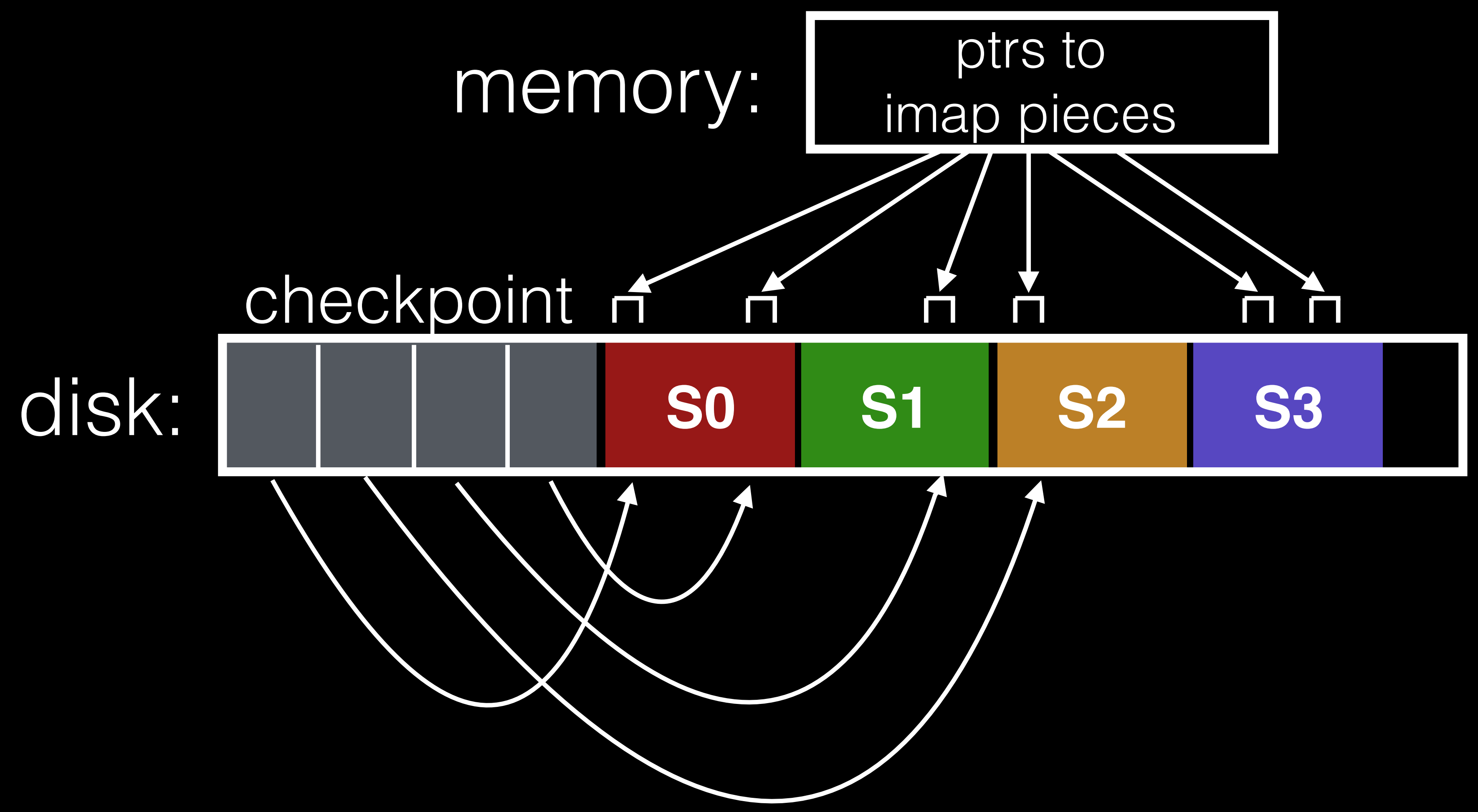
Better approach: occasionally checkpoint the pointers to imap pieces on disk.

Checkpoint often: random I/O.
Checkpoint rarely: recovery takes longer.
Example: checkpoint every 30s

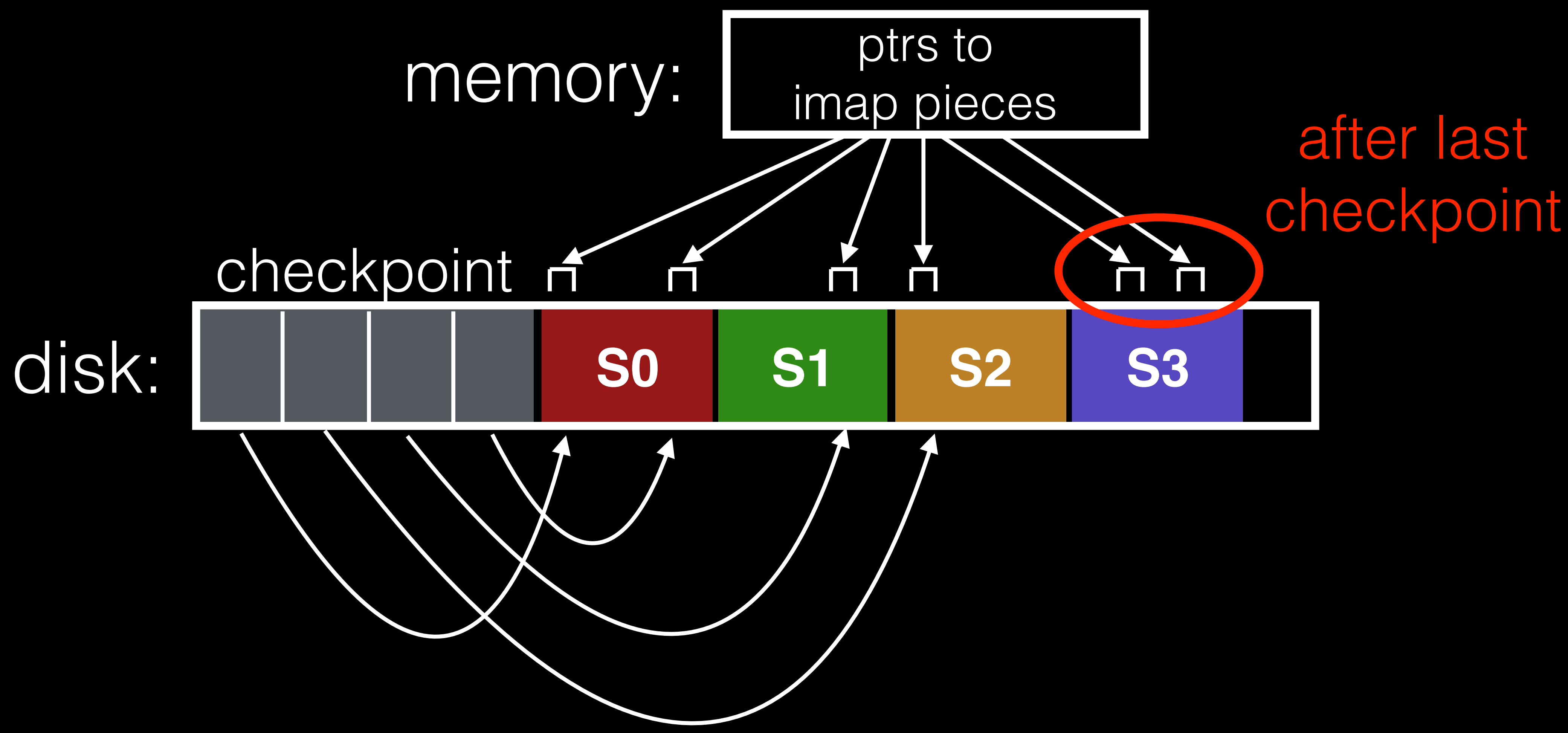
Checkpoint



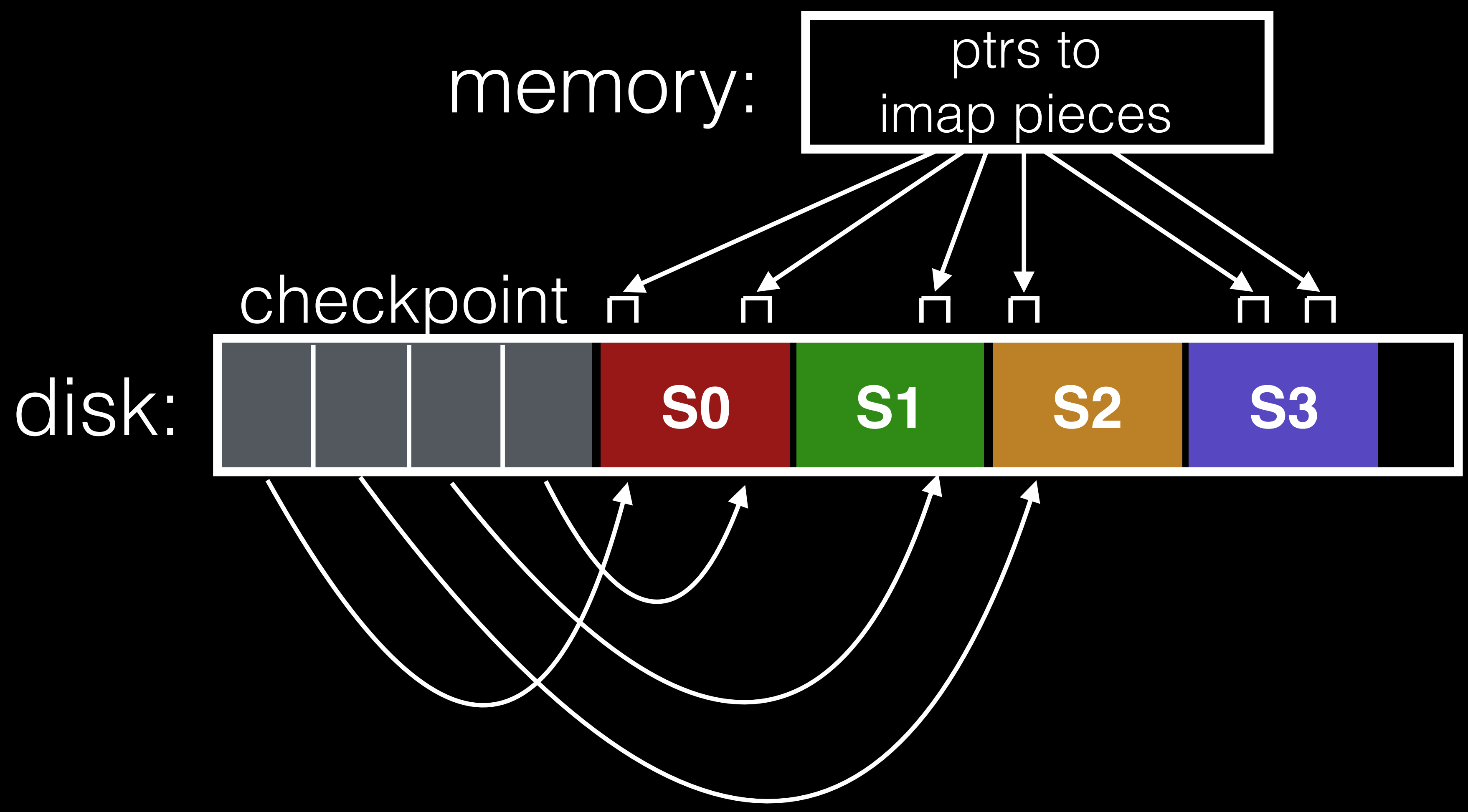
Checkpoint



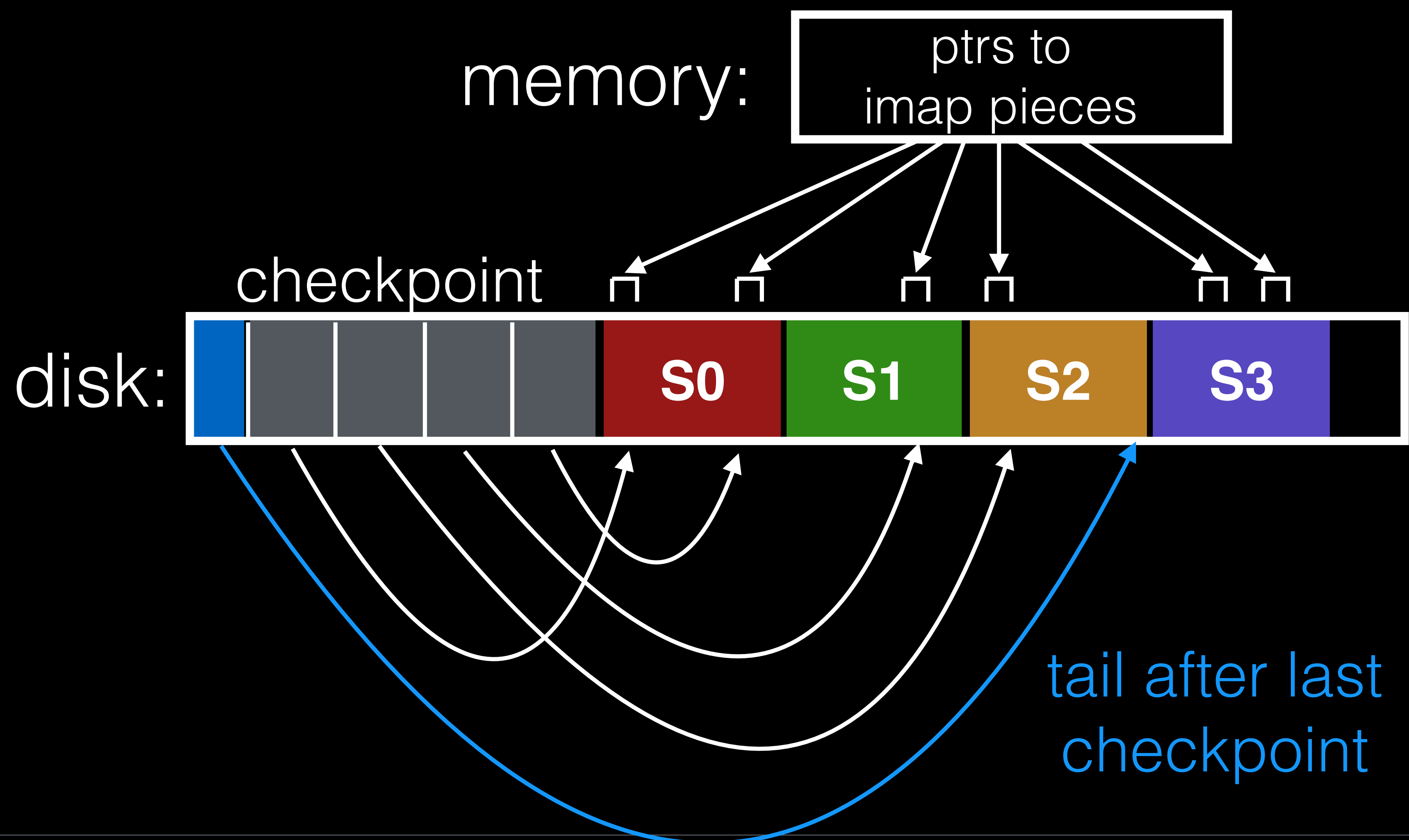
Checkpoint


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

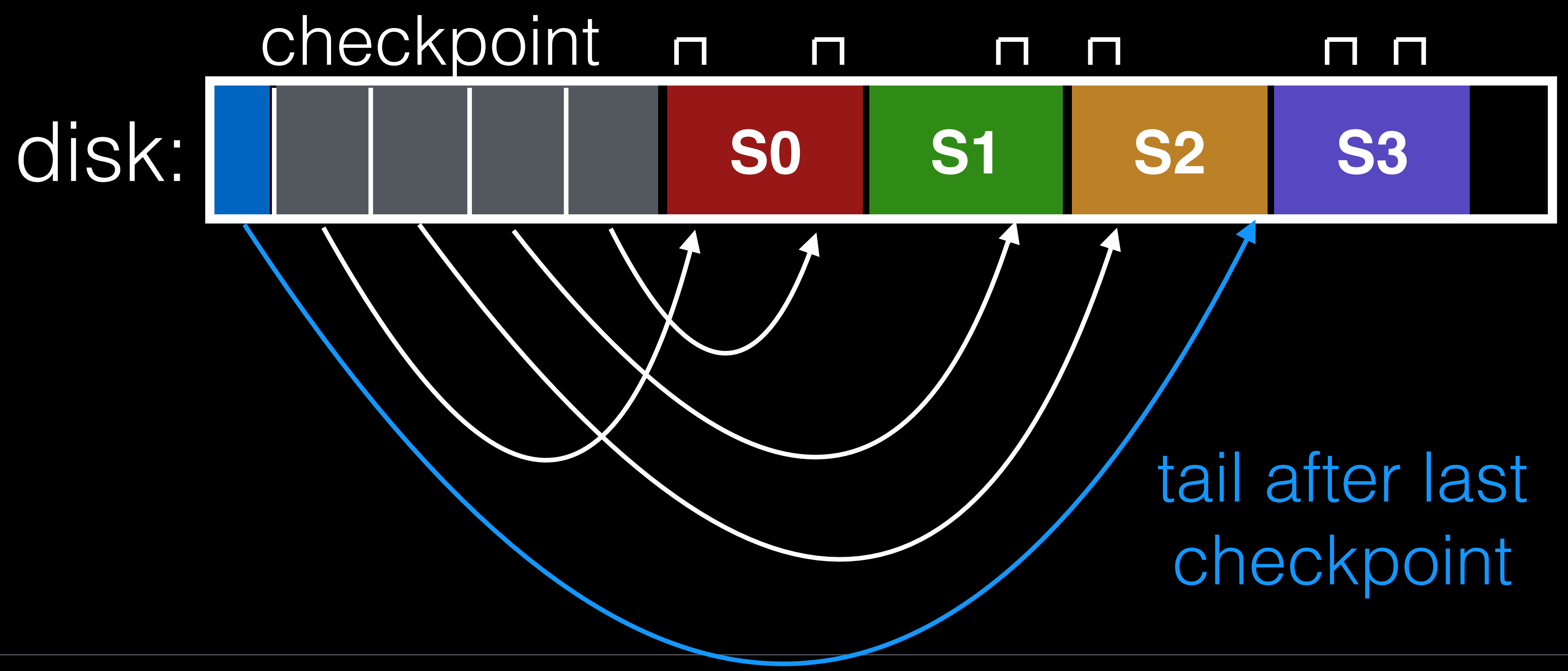
Checkpoint



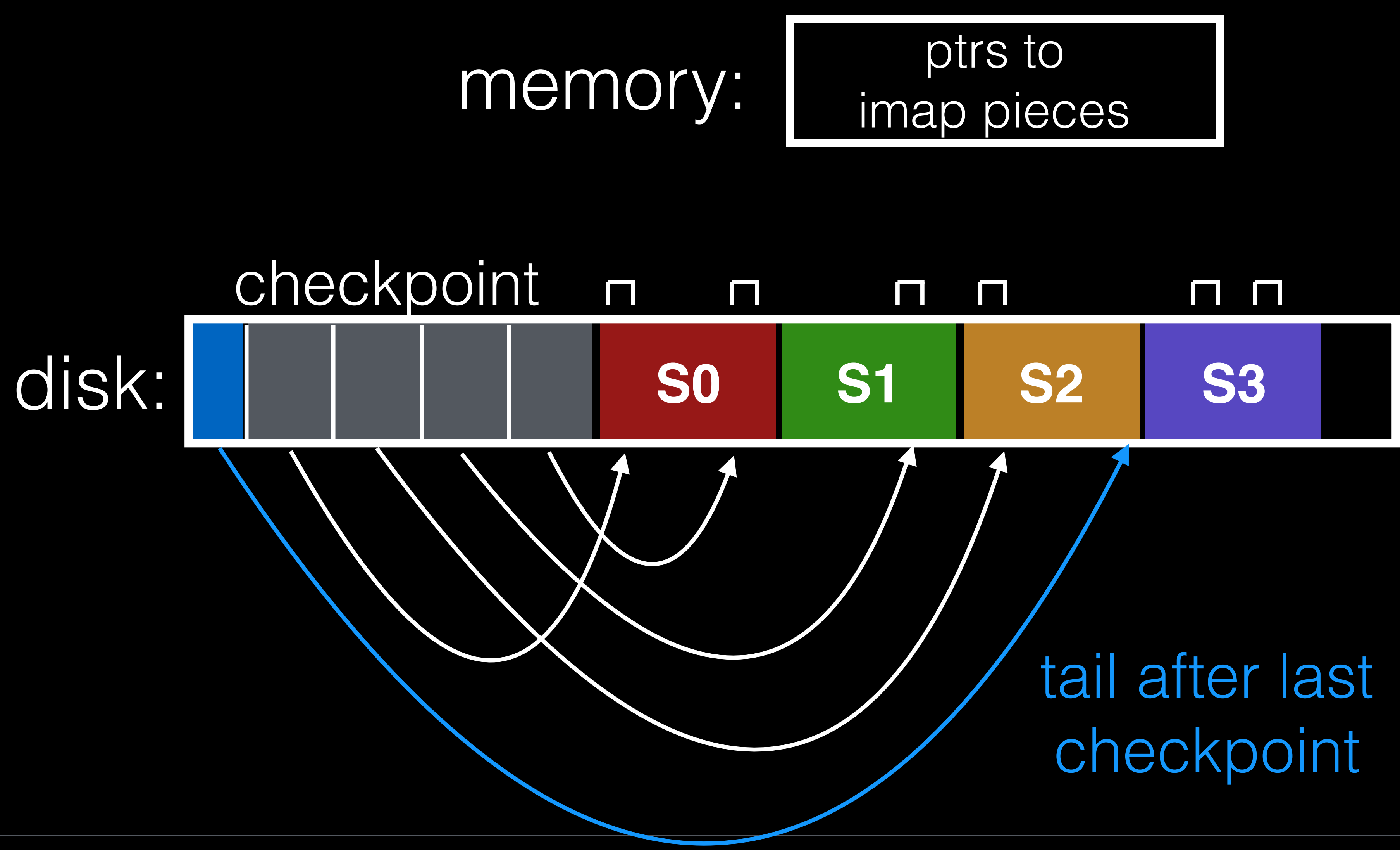
Checkpoint


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

Crash!

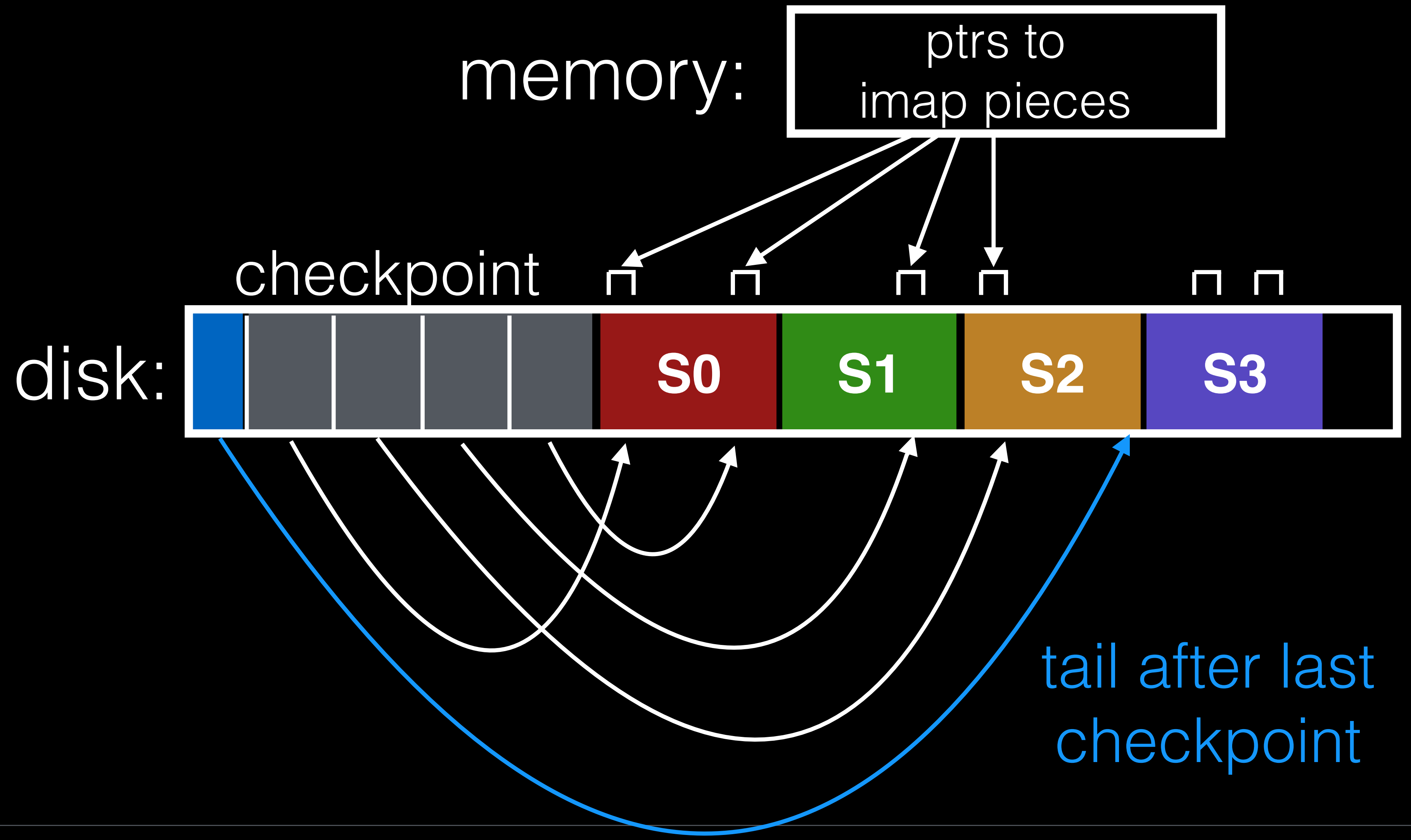


Reboot


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

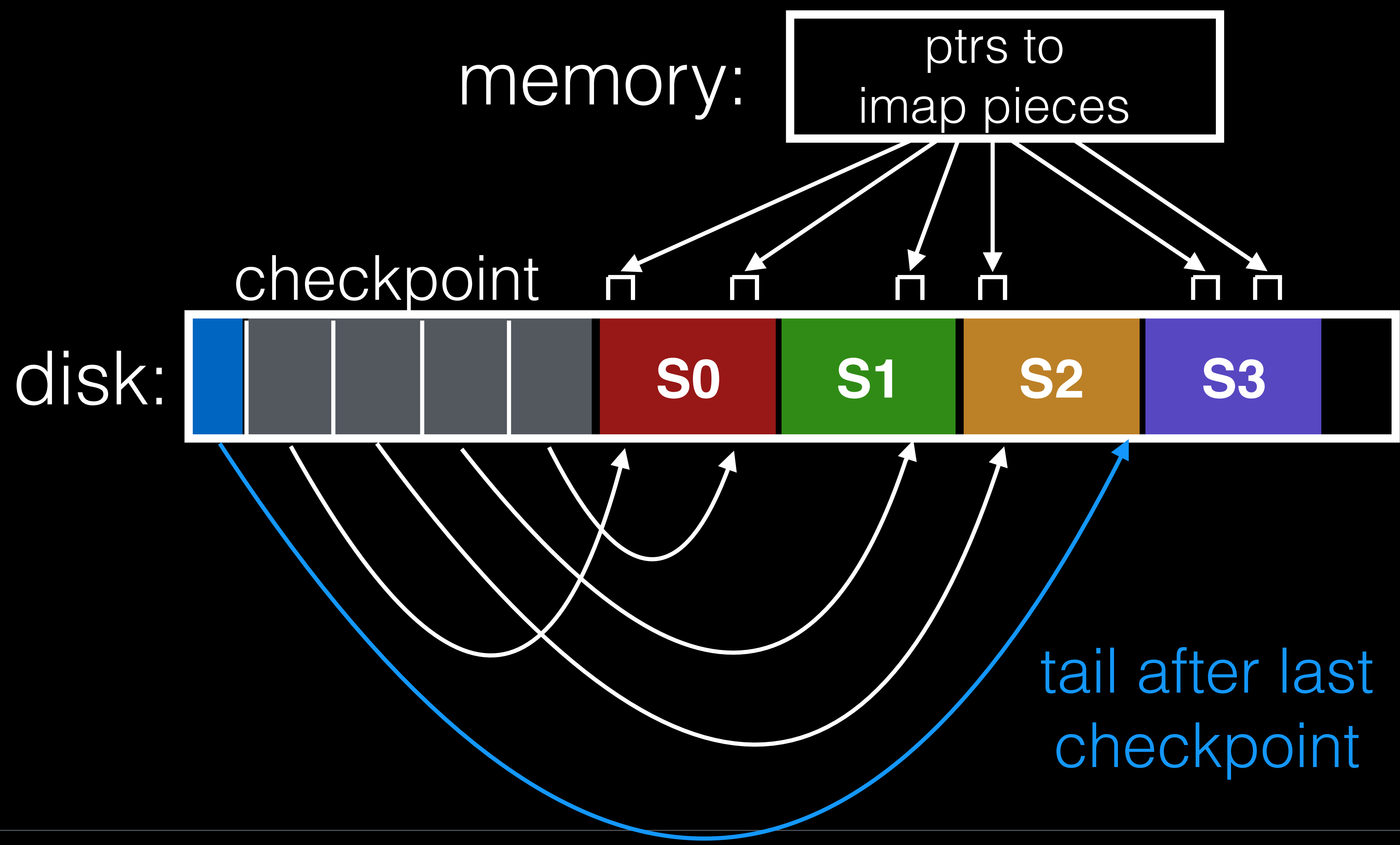
Reboot

get pointers
from checkpoint



Reboot

get pointers
by scanning
after tail.



Checkpoint Overview

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

Upon recovery:
- read checkpoint to get most pointers and tail
- get rest of pointers by reading past tail

Checkpoint Overview

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

Upon recovery:
- read checkpoint to get most pointers and tail
- get rest of pointers by reading past tail

What if we crash during checkpoint?

Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

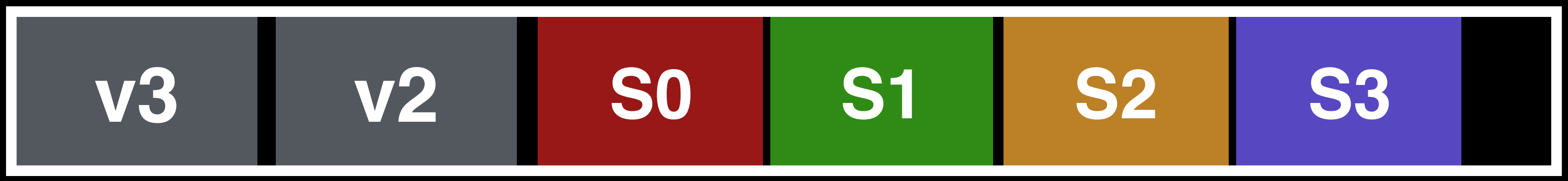
Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Checkpoint Strategy

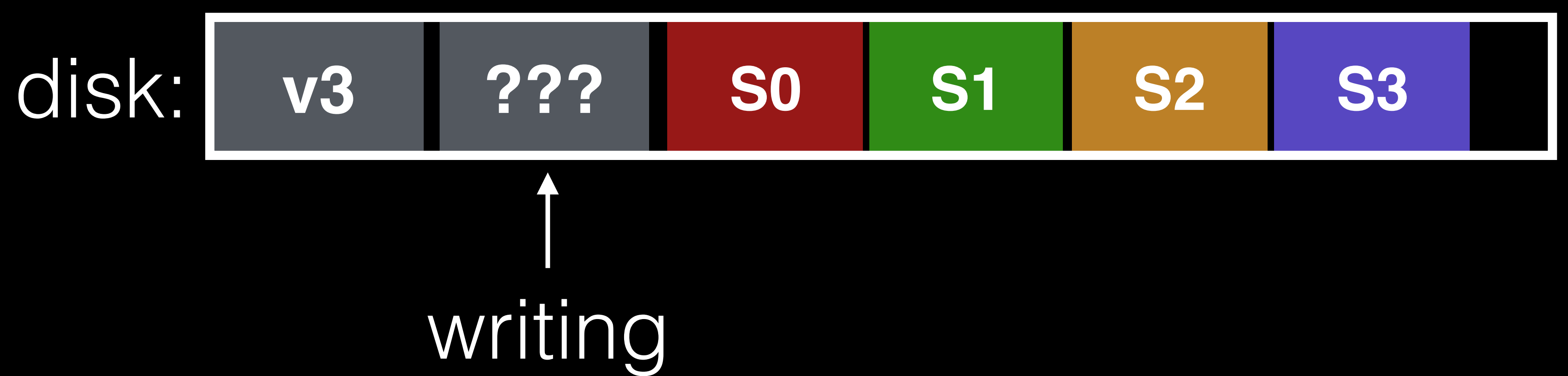
Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

disk:



Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.



Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Checkpoint Strategy

Have two checkpoints.
Only overwrite one at a time.
Use checksum/timestamps to identify newest.

Other Issues

- Crashes
- Garbage Collection

Versioning File Systems

Motto: garbage is a feature!

Versioning File Systems

Motto: garbage is a feature!

Keep old versions in case the user wants to revert files later.

Like Dropbox.

Garbage Collection

Need to reclaim space:
1. when no more references (any file system)
2. after a newer copy is created (COW file system)

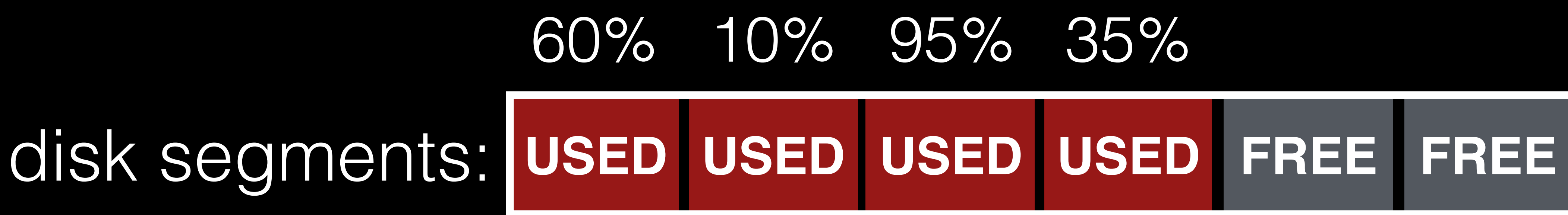
We want to reclaim segments.
 - tricky, as segments are usually partly valid

Garbage Collection

disk segments:

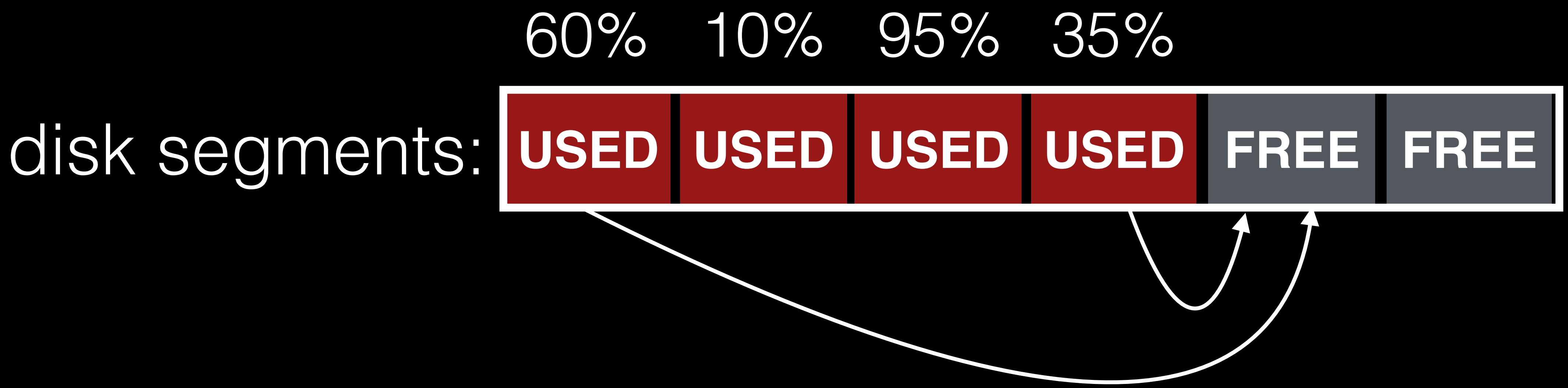
Garbage Collection

how much data is good in each?

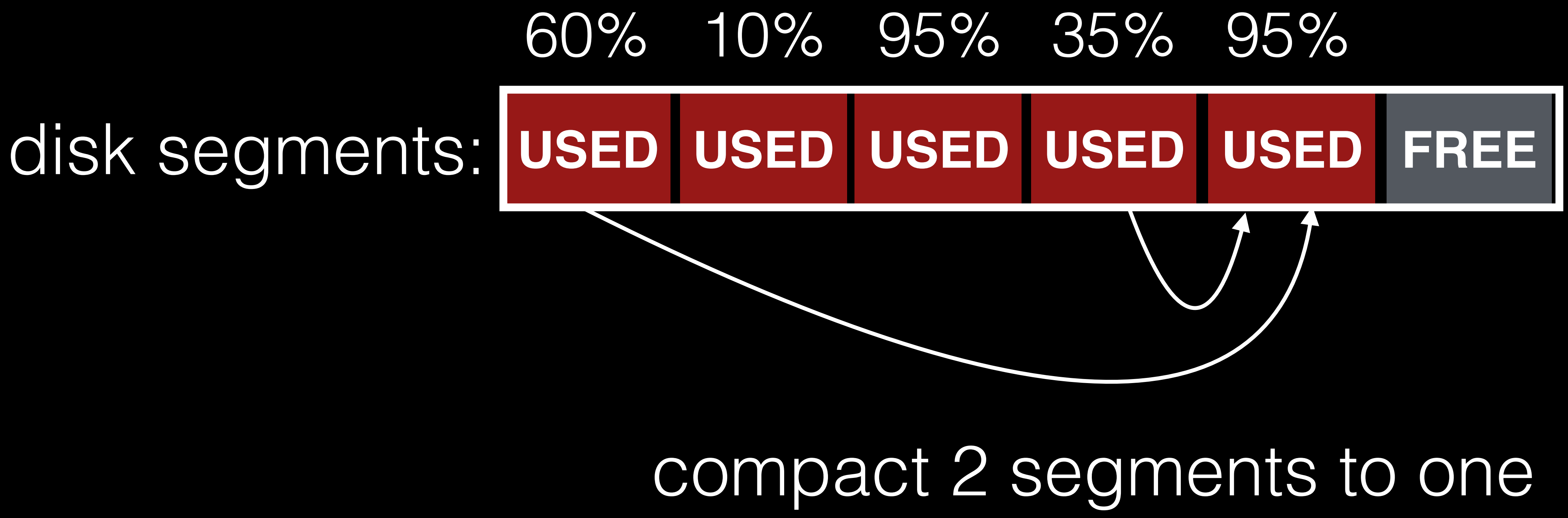


Garbage Collection

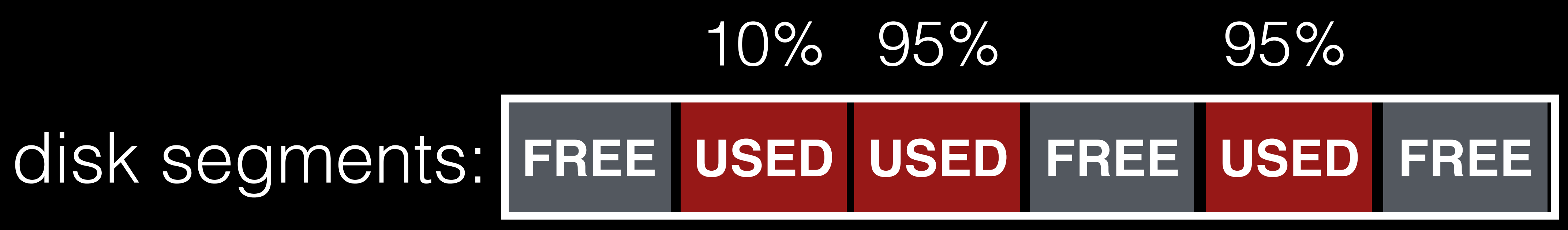
Garbage Collection



Garbage Collection



Garbage Collection



release input segments

Garbage Collection

General operation:
pick **M** segments, compact into **N** (where **N** < **M**).

Mechanism: how do we know whether data in segments is valid?

Policy: which segments to compact?

Mechanism

Is an inode the latest version?
Check imap to see if it is pointed to (fast).

Is a data block the latest version?
Scan ALL inodes to see if it is pointed to (very slow).

Mechanism

Is an inode the latest version?
Check imap to see if it is pointed to (fast).

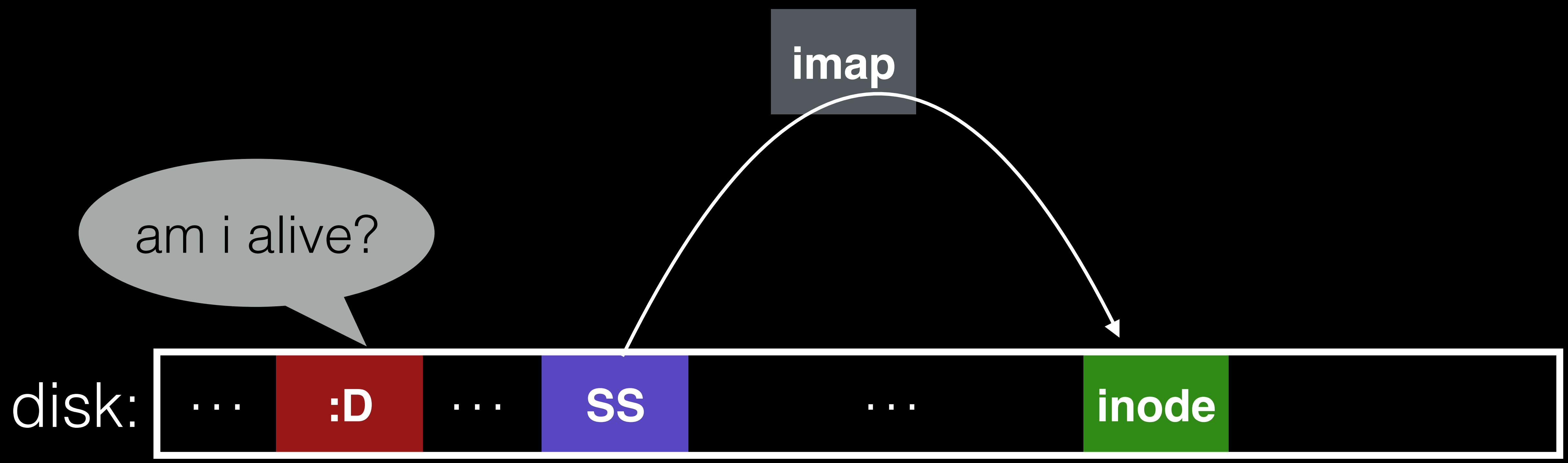
Is a data block the latest version?
Scan ALL inodes to see if it is pointed to (very slow).

Solution: segment summary that lists inode corresponding to each data block.

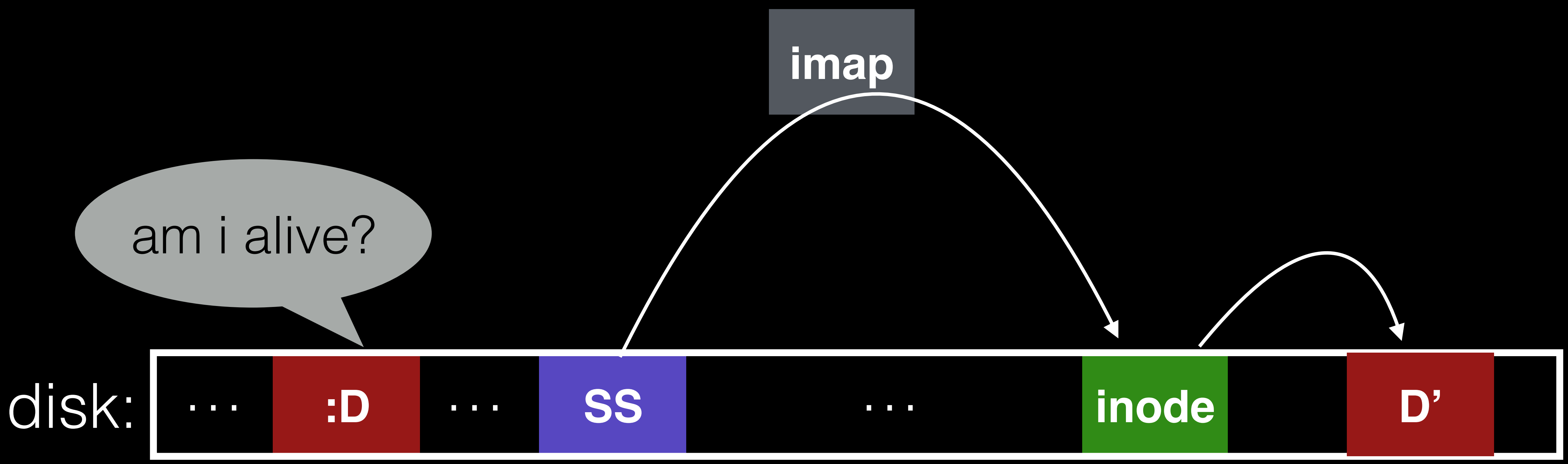
Block Liveness

Block Liveness

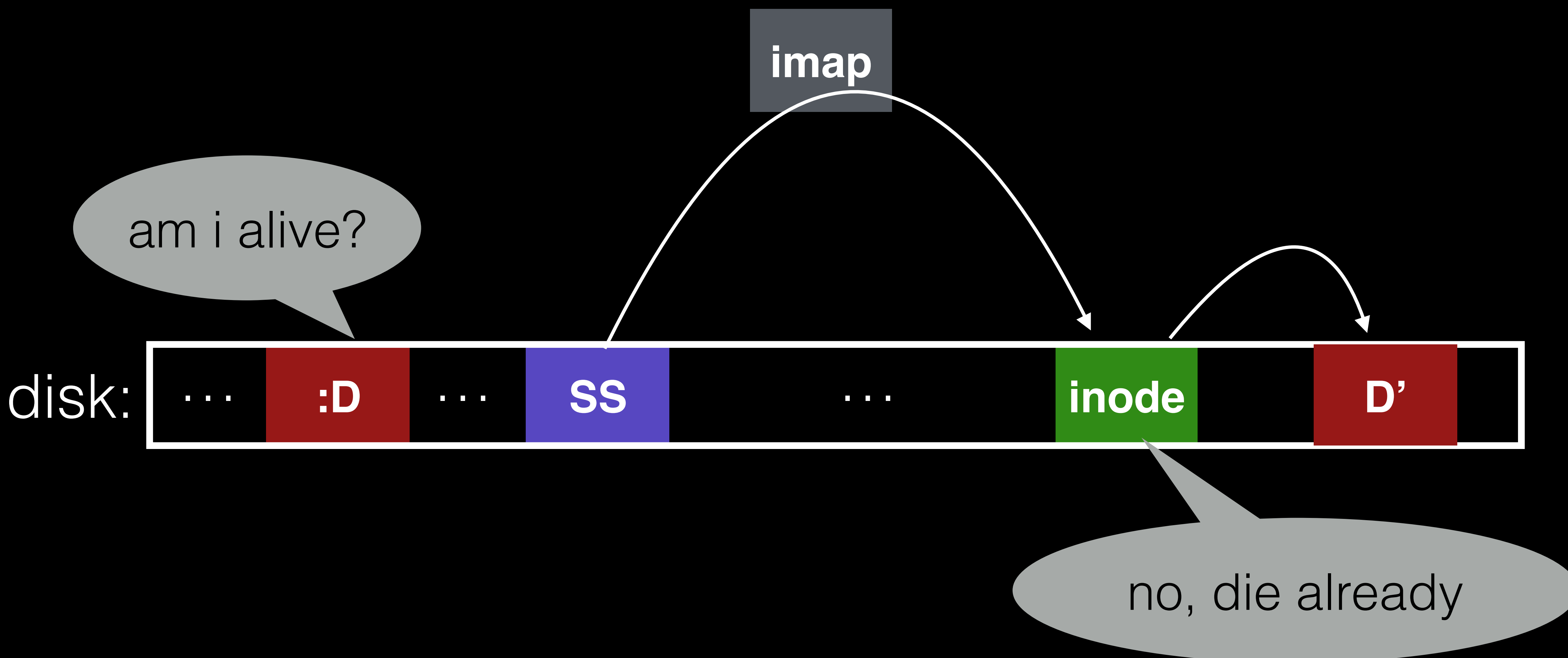
Block Liveness



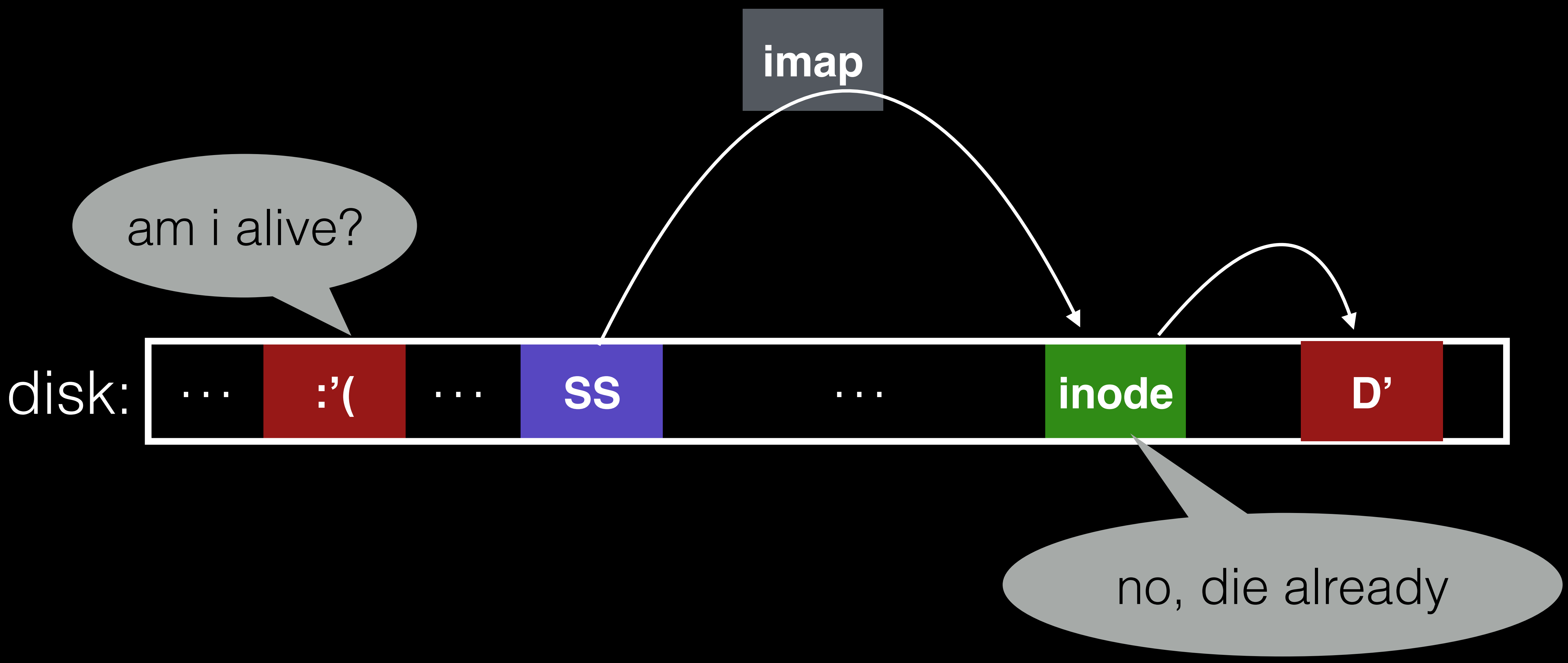
Block Liveness



Block Liveness



Block Liveness



Garbage Collection

General operation:
pick **M** segments, compact into **N** (where **N** $<$ **M**).

Mechanism: how do we know whether data in segments is valid?

Policy: which segments to compact?

Garbage Collection

General operation:
pick **M** segments, compact into **N** (where **N** < **M**).

Mechanism: how do we know whether data in segments is valid? [segment summary]

Policy: which segments to compact?

Policy

Many possible:

clean most empty first
clean coldest
more complex heuristics…

Conclusion

Journaling: let's us put data wherever we like. Usually in a place optimized for future reads.

LFS: puts data where it's fastest to write.

Other COW file systems: WAFL, ZFS, btrfs.