Operating Systems

RAID and Filesystems

Overview

- RAID
- File Systems
- Inodes
- Storage

Disks

What if... MOARR!!!

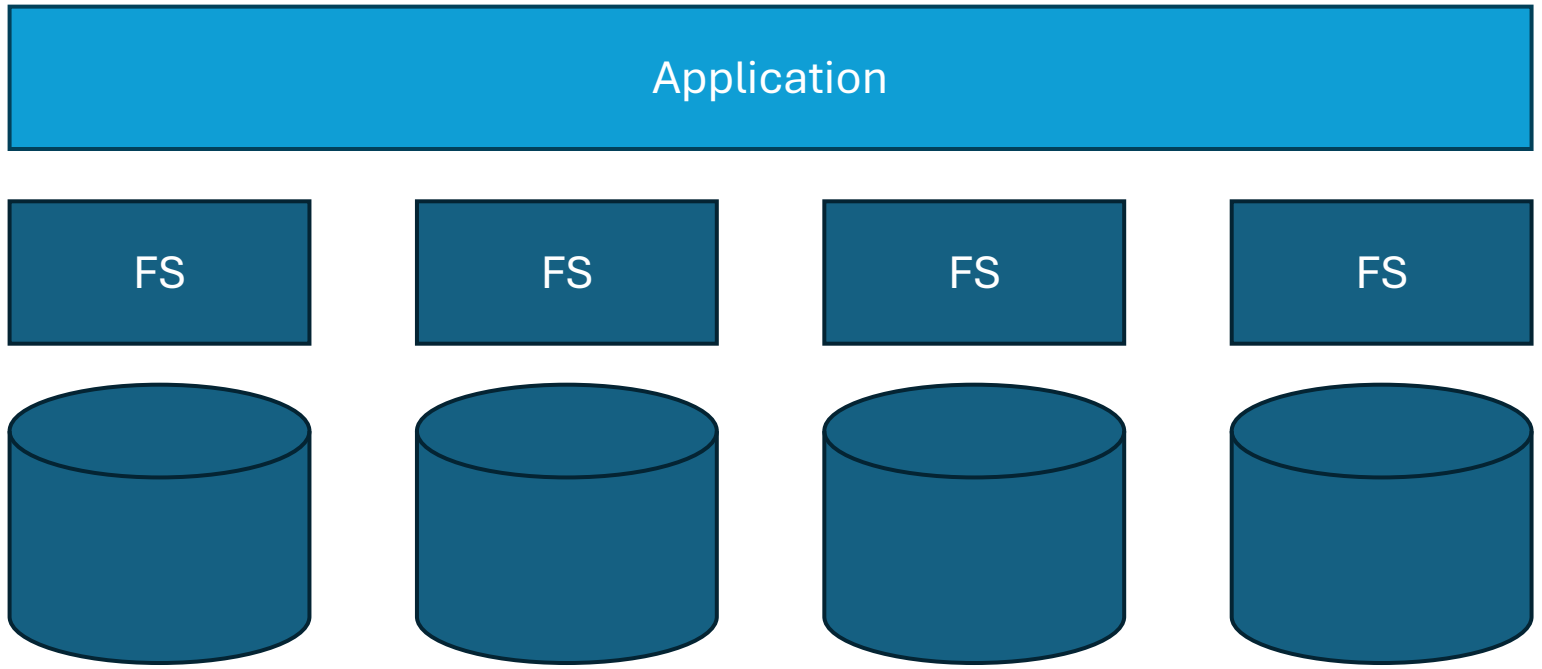
Disks

Motivation?

- Capacity
- Reliability
- Performance

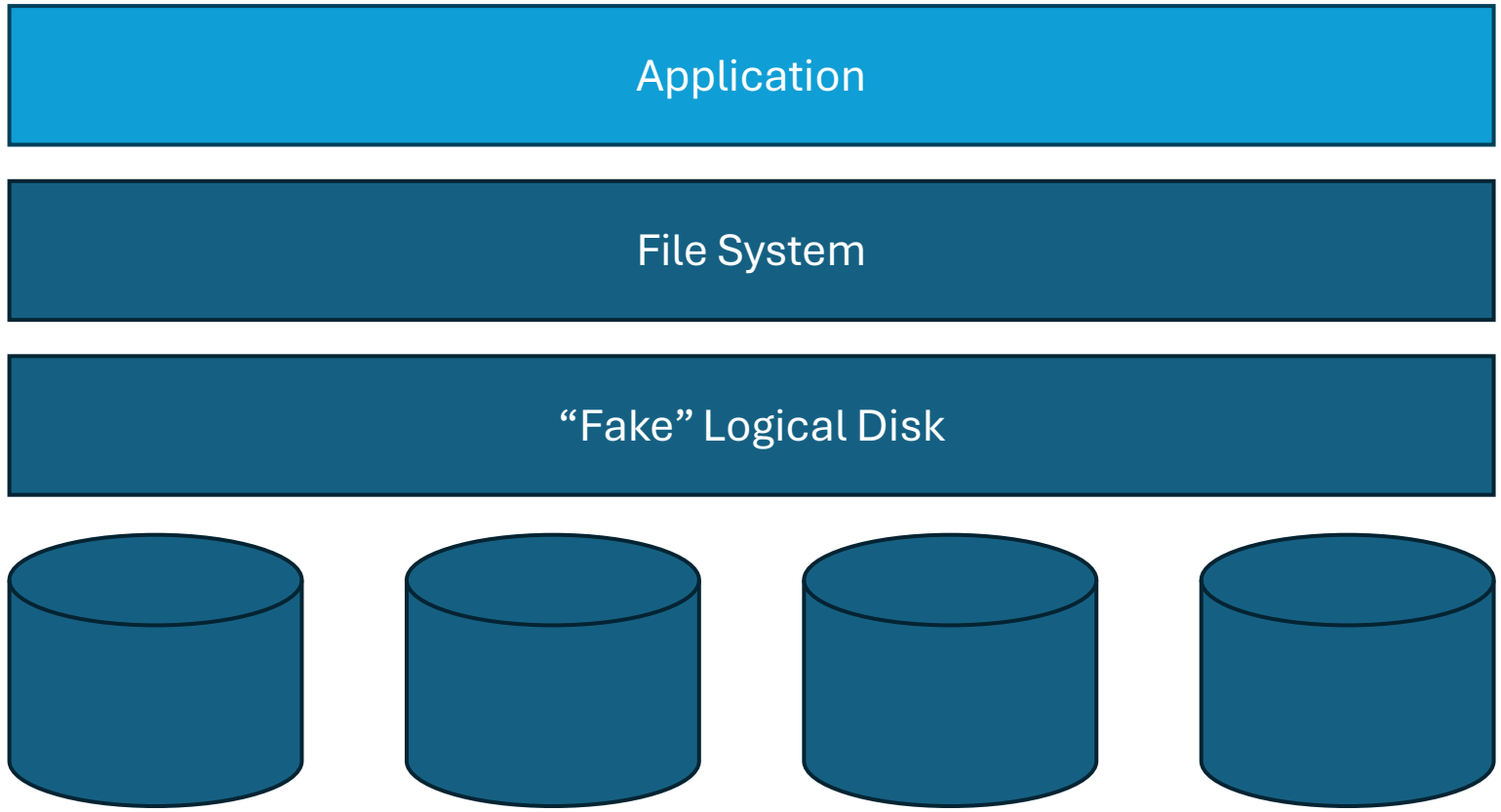
Options: JBOD

Just a bunch of disks



Options: RAID

Redundant Array of Independent Disks



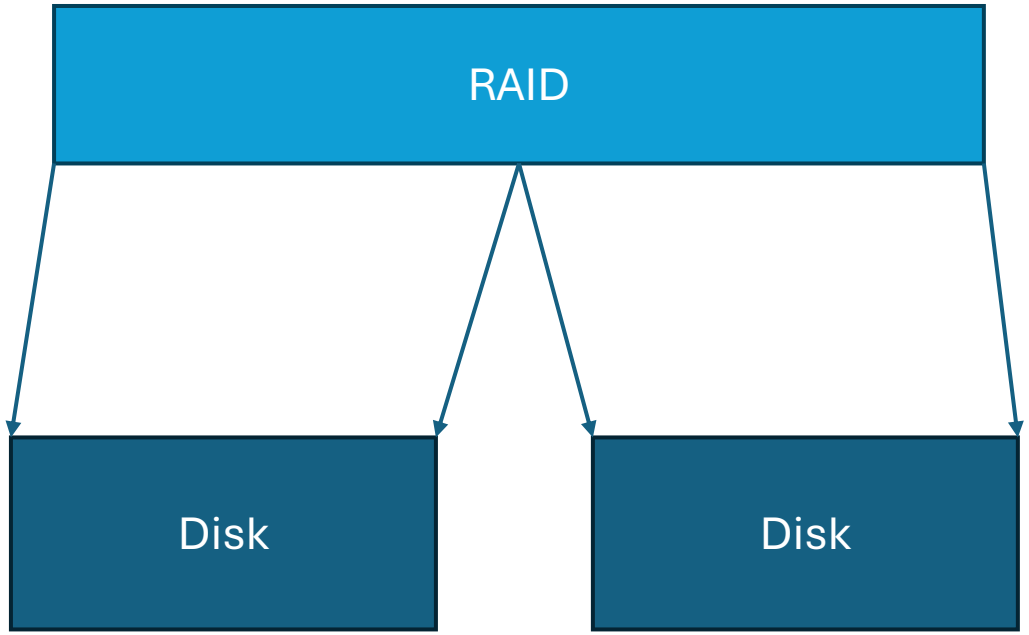
Options: RAID

Inexpensive

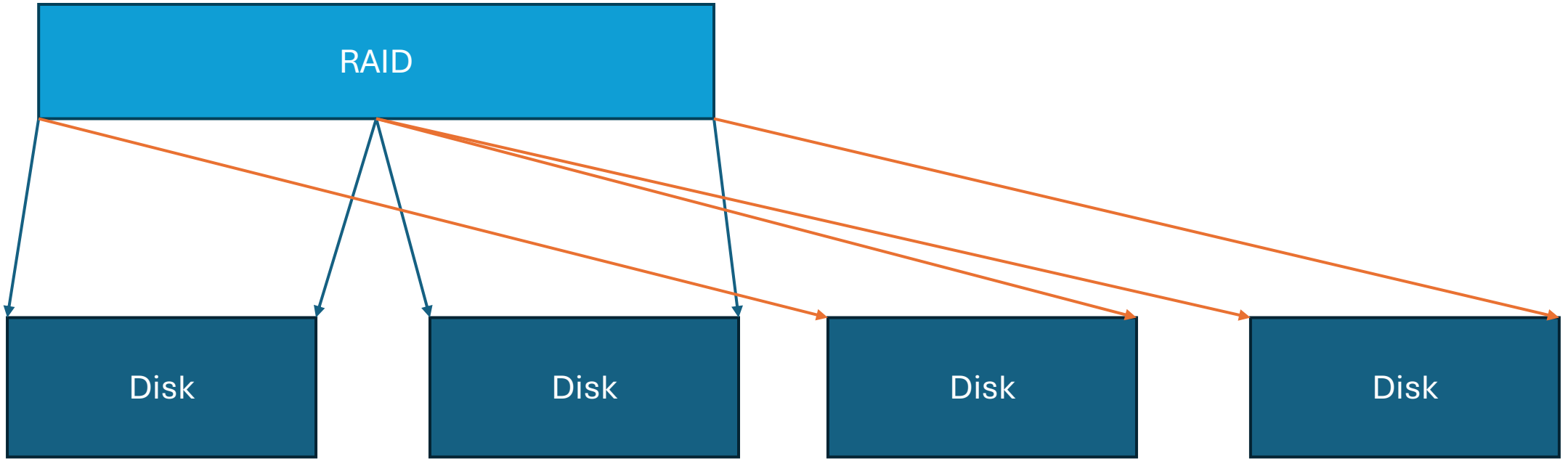
- Economy of scale!
- Big, but cheap

Strategy

- Write Software to build high-quality logical devices from small cheap devices



Options: RAID: Redundancy



Mapping

RAID

- Done with basic arithmetic…
- No indirections
- Uses a micro-controller to do all the thinking
- The OS doesn’t know…

Redunancy

Principles

- Must write to each copy
- Can read from any copy
  => speed improvement
- More duplicates
  => more space required

RAID

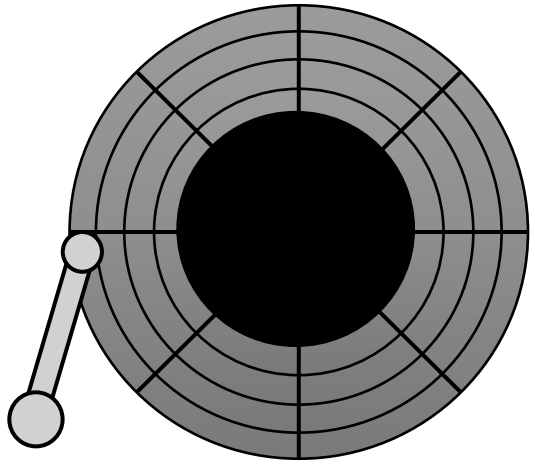
Workload

- Types of read/writes issued by applications (sequential vs random)

RAID

Workload

- **Sequential:** Lots of data, all in the same location
- **Random:** Little bits of data, all in different locations



We will assume that ‘seek time’ is relevant here…

RAID Design

Design Considerations

How to do the mapping?

How to use the extra blocks?

Different RAID levels, make different choices.

RAID Design

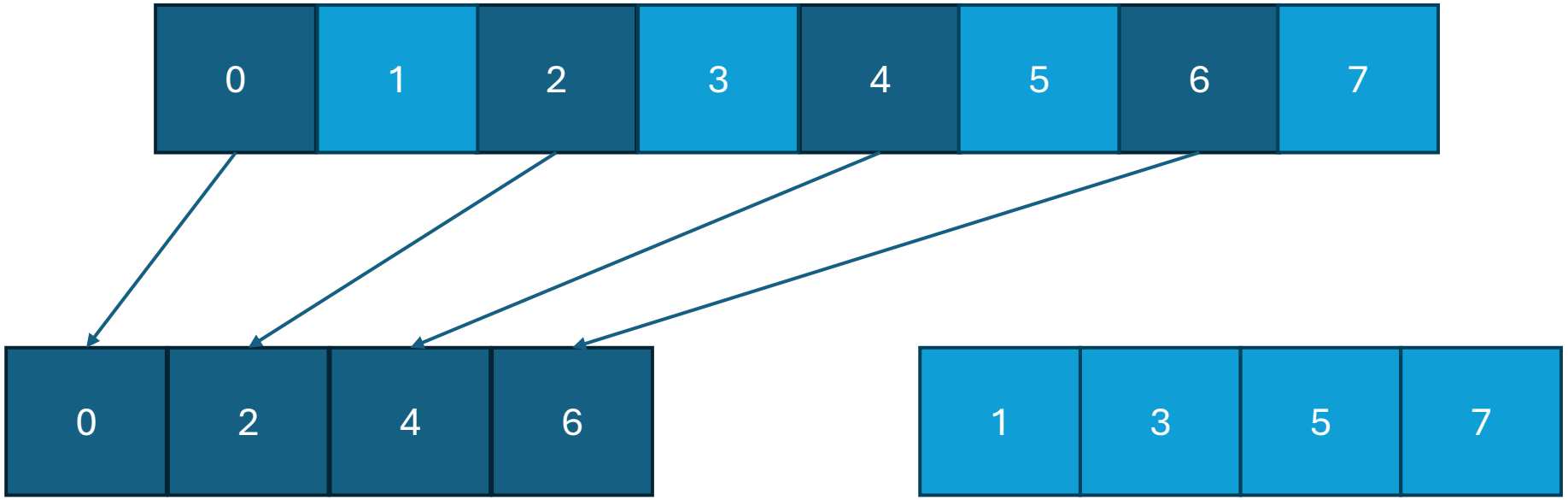
Capacity: How much space can apps use?

Reliability: How many disks can we safely lose?

Performance: How long does each workload take?

N:	Number of disks	
C:	Capacity of 1 disk	
S:	Sequential throughput of 1 disk	
R:	Random throughput of 1 disk	(S >> R)
D:	latency of one small I/O operation	

RAID-0: Striping



I/O is evenly distributed
Zero-disks may fail

RAID-0: Maths

I am looking for chunk #45?

Where is it?

RAID-0: Maths

Disk #: block# % disk_total

Offset#: block# // disk_total

45%2 = 1 (i.e.) second disk

45//2 = 22 (i.e.) 22nd block

RAID-0: Striping

Capacity: N * C

Disk Fails: 0

Latency: D

Througput: N * S

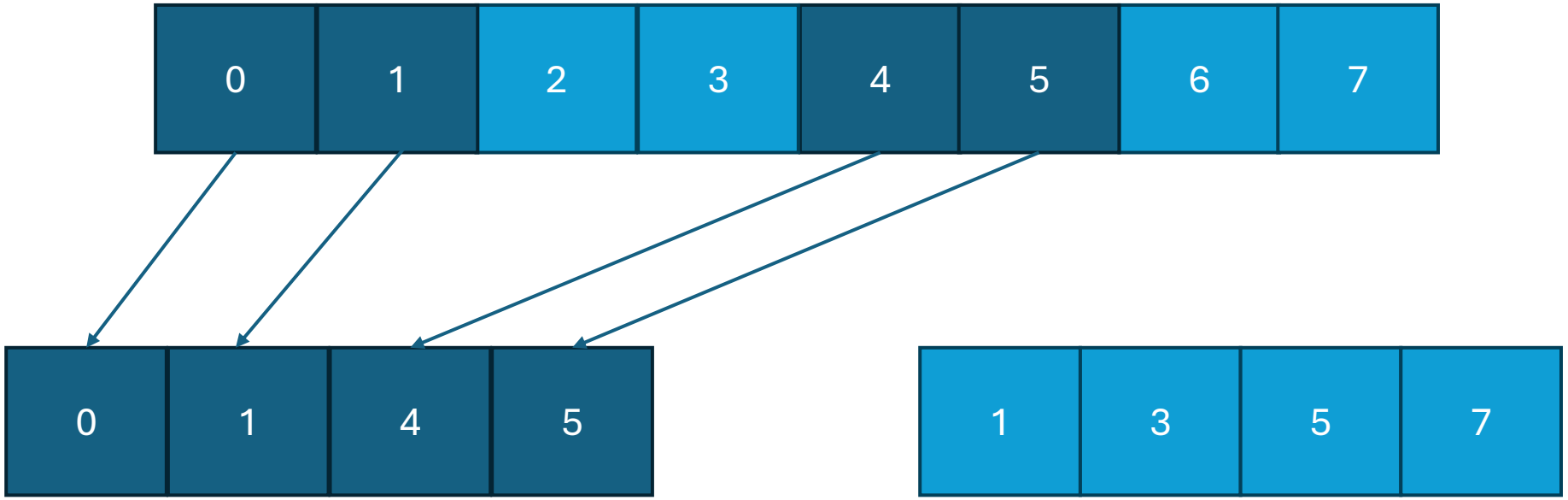
N * R

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Striping with higher disk counts

N: Number of disks
C: Capacity of 1 disk
S: Sequential throughput of 1 disk
R: Random throughput of 1 disk
D: latency of one small I/O operation

RAID-0: Chunk-Size



The chunk size can change

RAID-0 Chunk-Size

Name one advantage of a small chunk size

Name one disadvantage of a small chunk size

RAID-0 Chunk-Size

Small Chunks

- Requests **more** likely to be distributed across drives
- Positioning time bound by the maximum of multiple requests

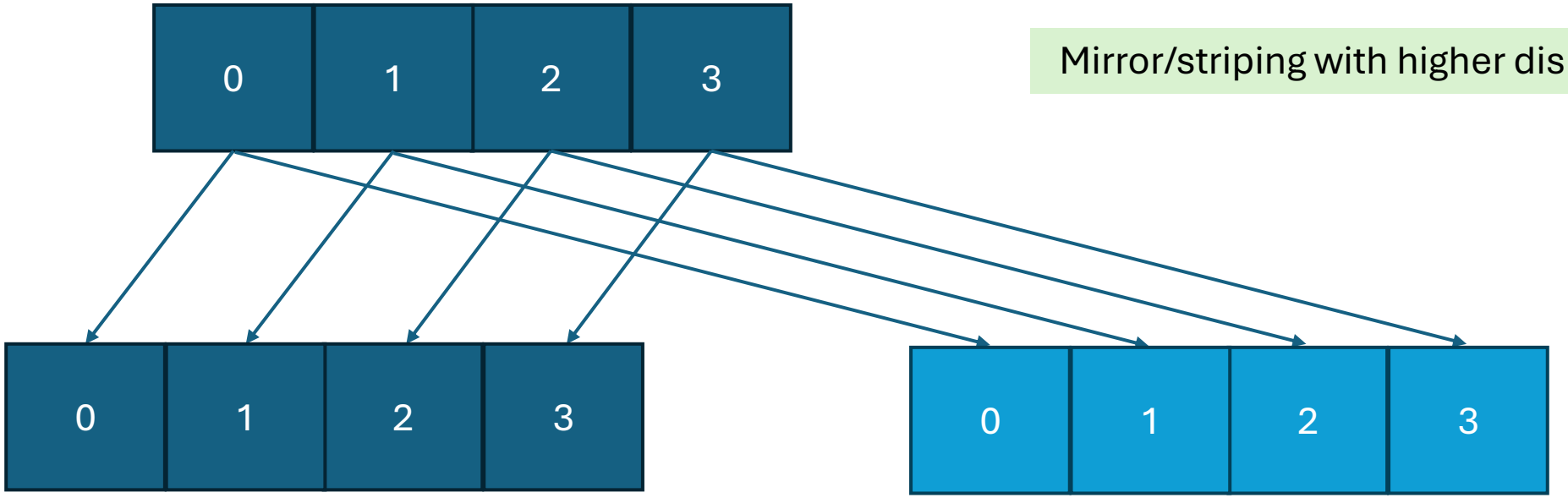
Big Chunks

- Requests **less** likely to be distributed across drives

RAID-1: Mirroring

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

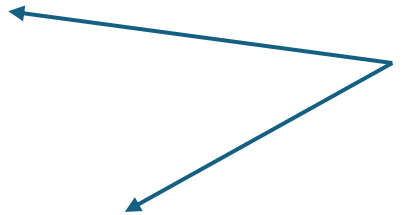
Mirror/striping with higher disk counts



Duplicate All
One disk may fail

RAID-1: Mirroring

Capacity:	C/2
Disk Fails:	1 -> N/2
Latency:	D
Througput (Random Read):	N * R
Througput (Random Write):	N/2 * R+
Througput (Sequential Read):	N/2 * S
Througput (Sequential Write):	N/2 * S+

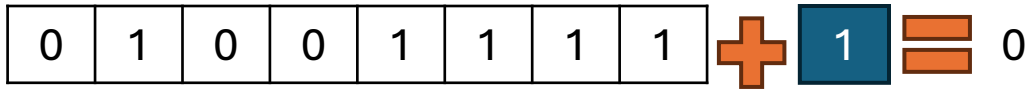


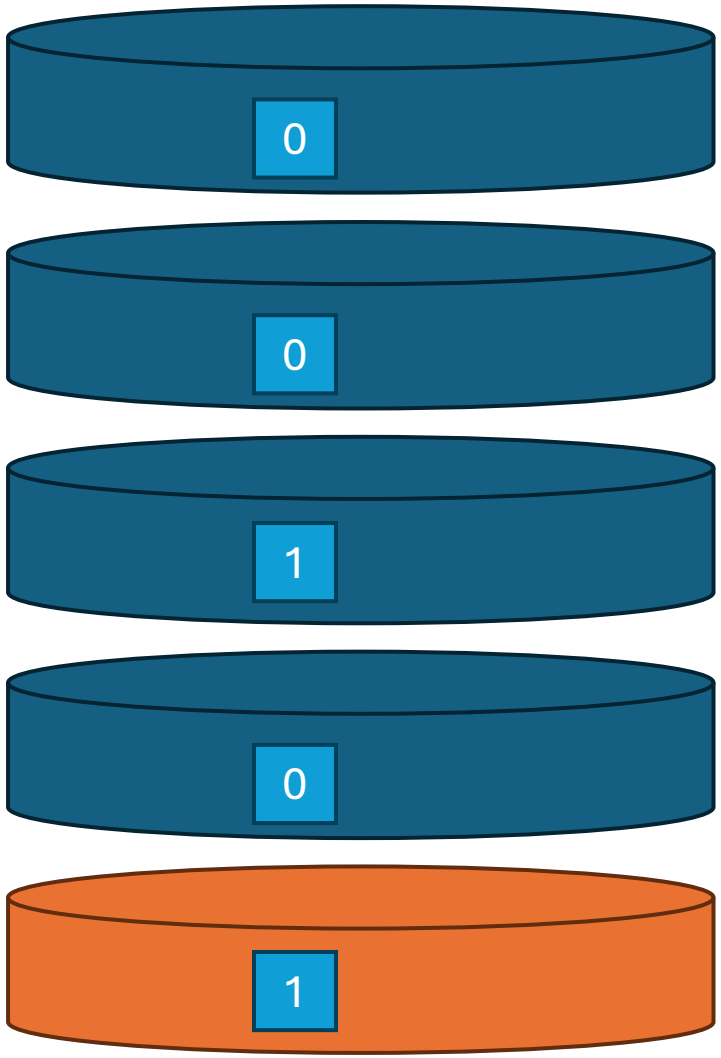
Split the reads

N:	Number of disks
C:	Capacity of 1 disk
S:	Sequential throughput of 1 disk
R:	Random throughput of 1 disk
D:	latency of one small I/O operation

RAID-4 Strategy

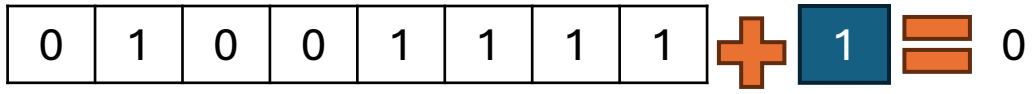
Parity disk

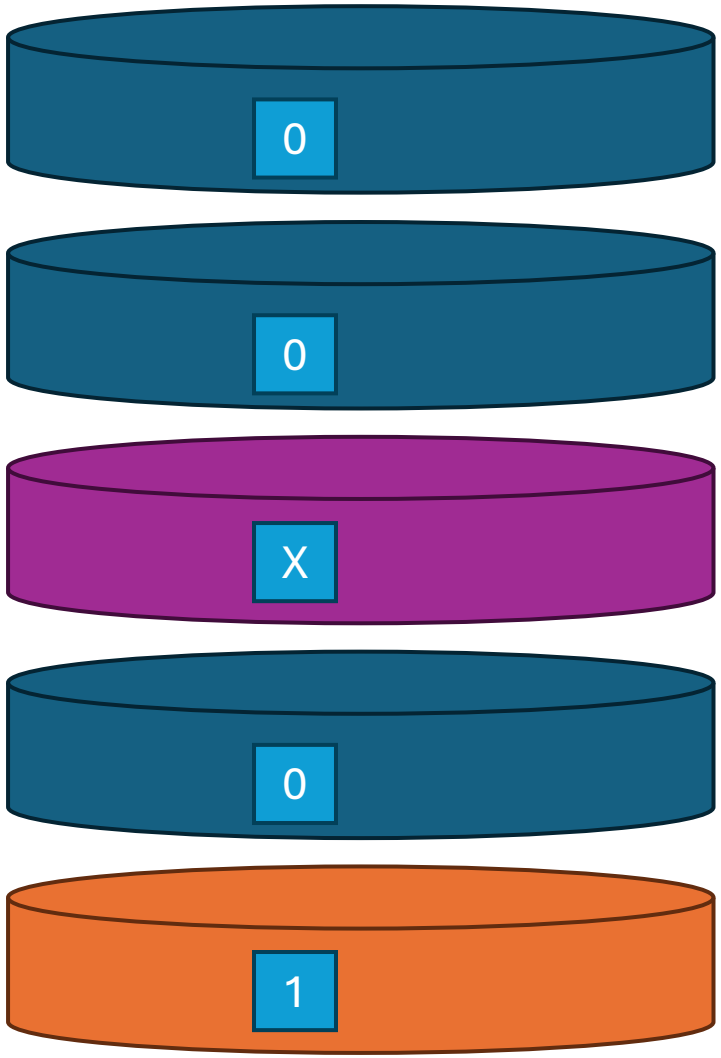




RAID-4 Strategy

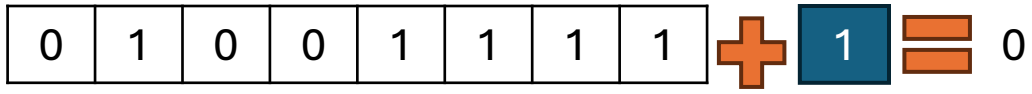
Parity disk

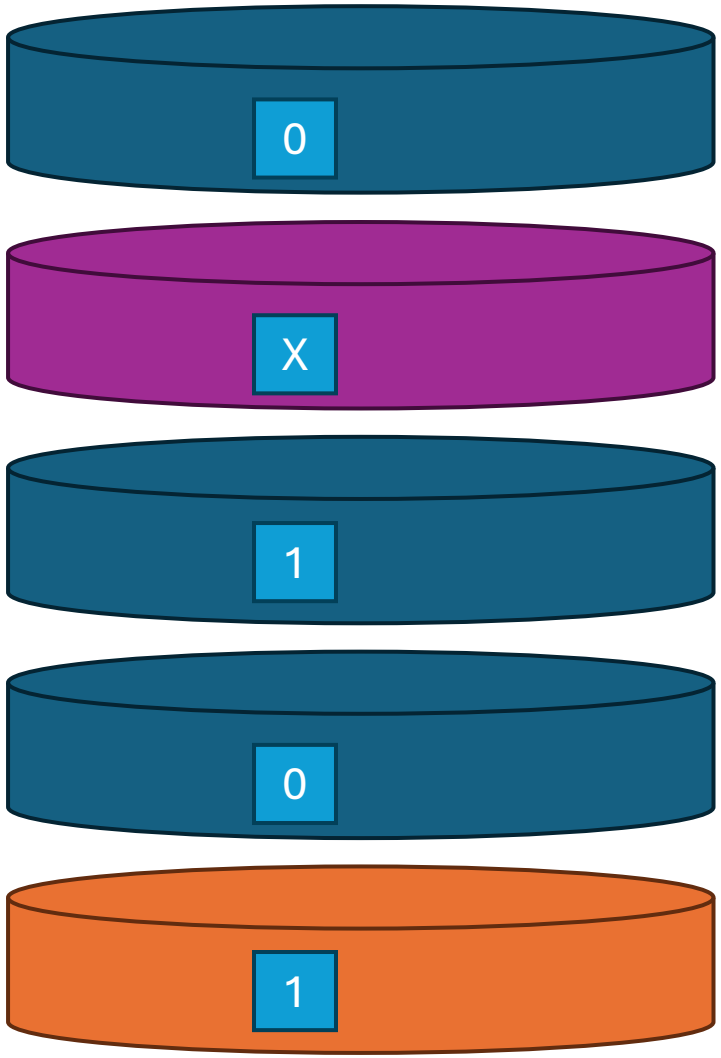




RAID-4 Strategy

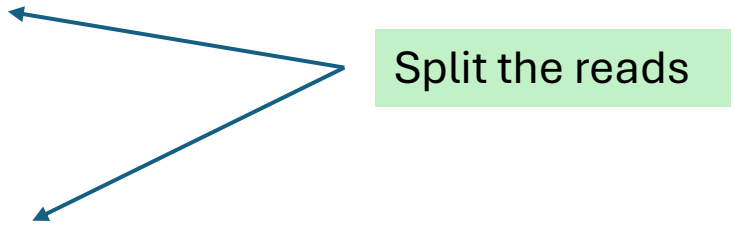
Parity disk





RAID-4: Parity

Capacity:	(N – 1) * C
Disk Fails:	1
Latency:	D
Througput (Random Read):	(N-1) * R
Througput (Random Write):	???
Througput (Sequential Read):	(N-1) * S
Througput (Sequential Write):	(N-1) * S



N:	Number of disks
C:	Capacity of 1 disk
S:	Sequential throughput of 1 disk
R:	Random throughput of 1 disk
D:	latency of one small I/O operation

Additive vs Subtractive Parity

Additive

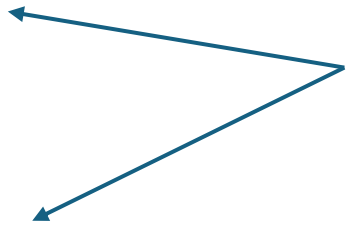
- Read all the other blocks
- Compute parity
- Write to the data block
- Write to the parity block

Subtractive

- Read the old value
- Find diff of old and new
- Write to the data block
- Write to the parity block

RAID-4: Parity

Capacity:	(N – 1) * C
Disk Fails:	1
Latency:	D (read) OR **D * 2 (write)**
Througput (Random Read):	(N-1) * R
Througput (Random Write):	R/2
Througput (Sequential Read):	(N-1) * S
Througput (Sequential Write):	(N-1) * S



Split the reads

N:	Number of disks
C:	Capacity of 1 disk
S:	Sequential throughput of 1 disk
R:	Random throughput of 1 disk
D:	latency of one small I/O operation

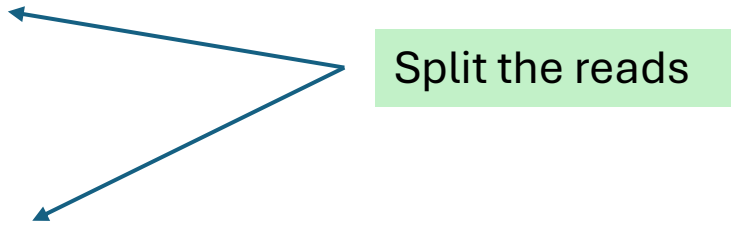
RAID-5 Strategy

As RAID 4, but spreads the parity bit across the hard-drives

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5: Rotating Parity

Capacity:	(N – 1) * C
Disk Fails:	1
Latency:	D (read) OR D * 2 (write)
Througput (Random Read):	**N** * R
Througput (Random Write):	N/4 * R
Througput (Sequential Read):	(N-1) * S
Througput (Sequential Write):	(N-1) * S



N:	Number of disks
C:	Capacity of 1 disk
S:	Sequential throughput of 1 disk
R:	Random throughput of 1 disk
D:	latency of one small I/O operation

Disks

Motivation?

- Capacity
- Reliability
- Performance

Activity

- Rate each method!

Mirroring and Power Loss

Mirroring and Power Loss

File Systems

How we manage things

Physical Storage vs Logical Storage

Physical

Issues

- Persistence

Components

- Hard Drives
- SSDs

Logical

Issues

- Structure
- Format

Components

- Files
- Directories

Files

What is a File?

- Persistent
- Modifiable
- Locatable
- Data (bytes)

What is a File System?

- A collection of files…
- The management of that collection…

Files: Locatable

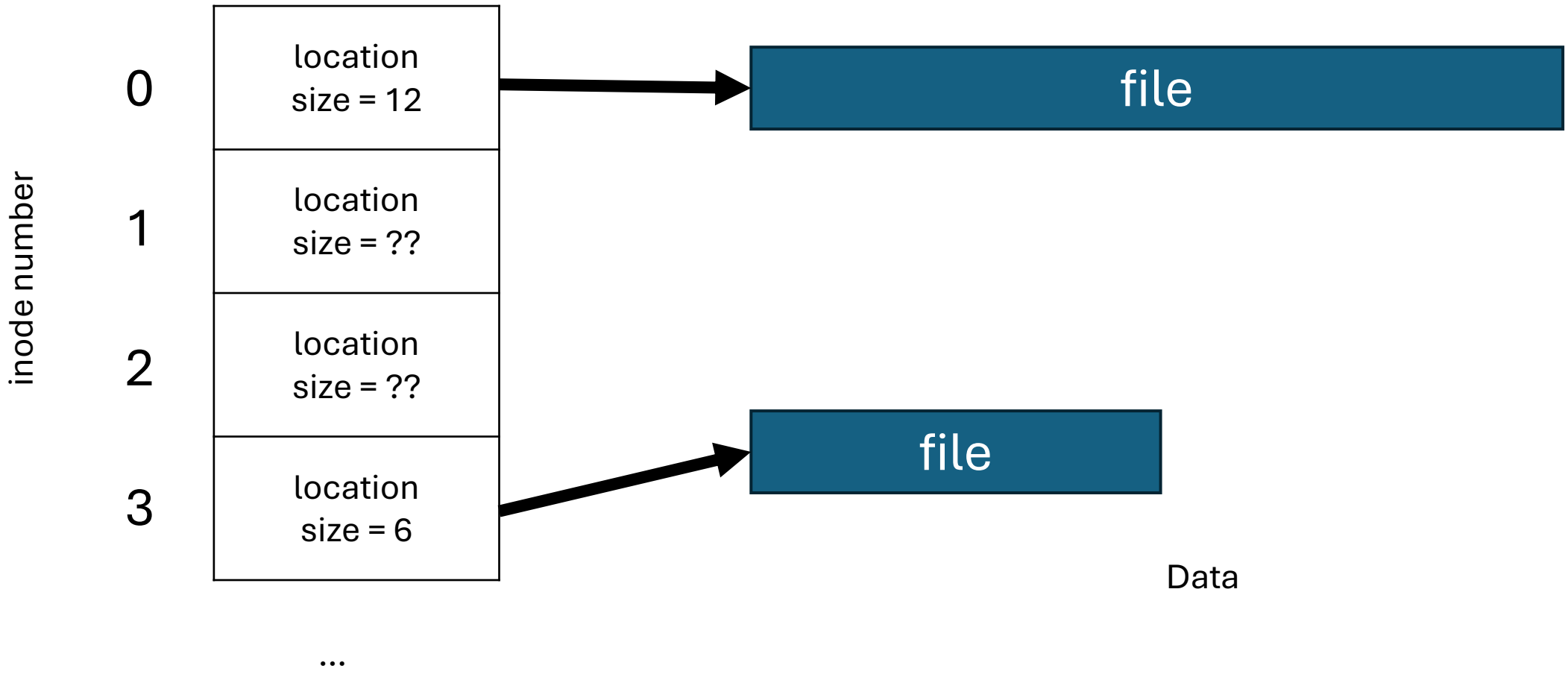
Locatable

- Every file requires a unique… number
- **i-node number**
  - Each file has exactly one
  - Each i-node number is unique
  - Can be recycled
  - Use “ls –i” to see

“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 (C, Unix)

Files: i-node


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

Files: Locatable (Humans)

Problem:

- Humans don’t ‘inode’

Facts:

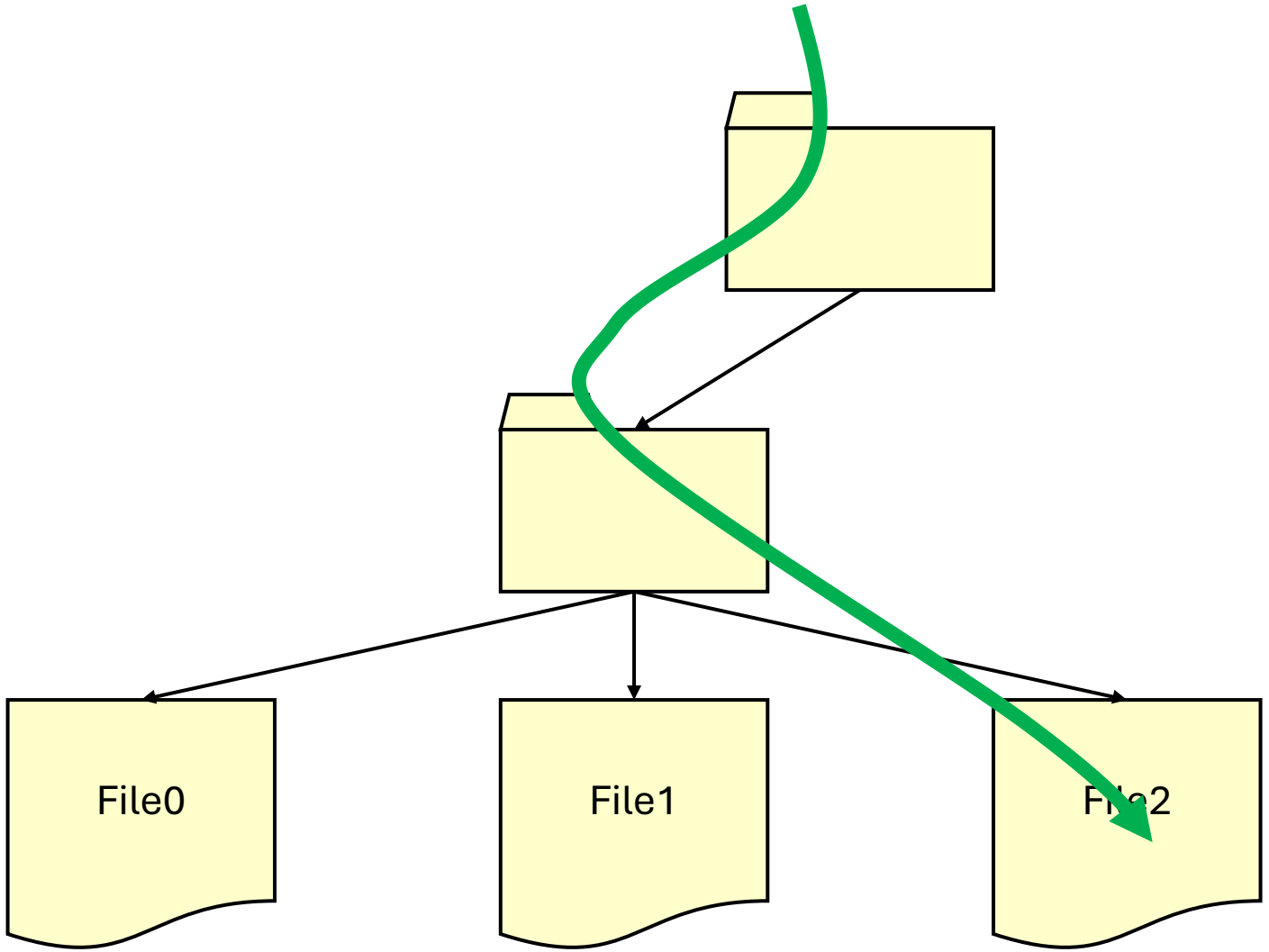
- Humans like words

Solution:

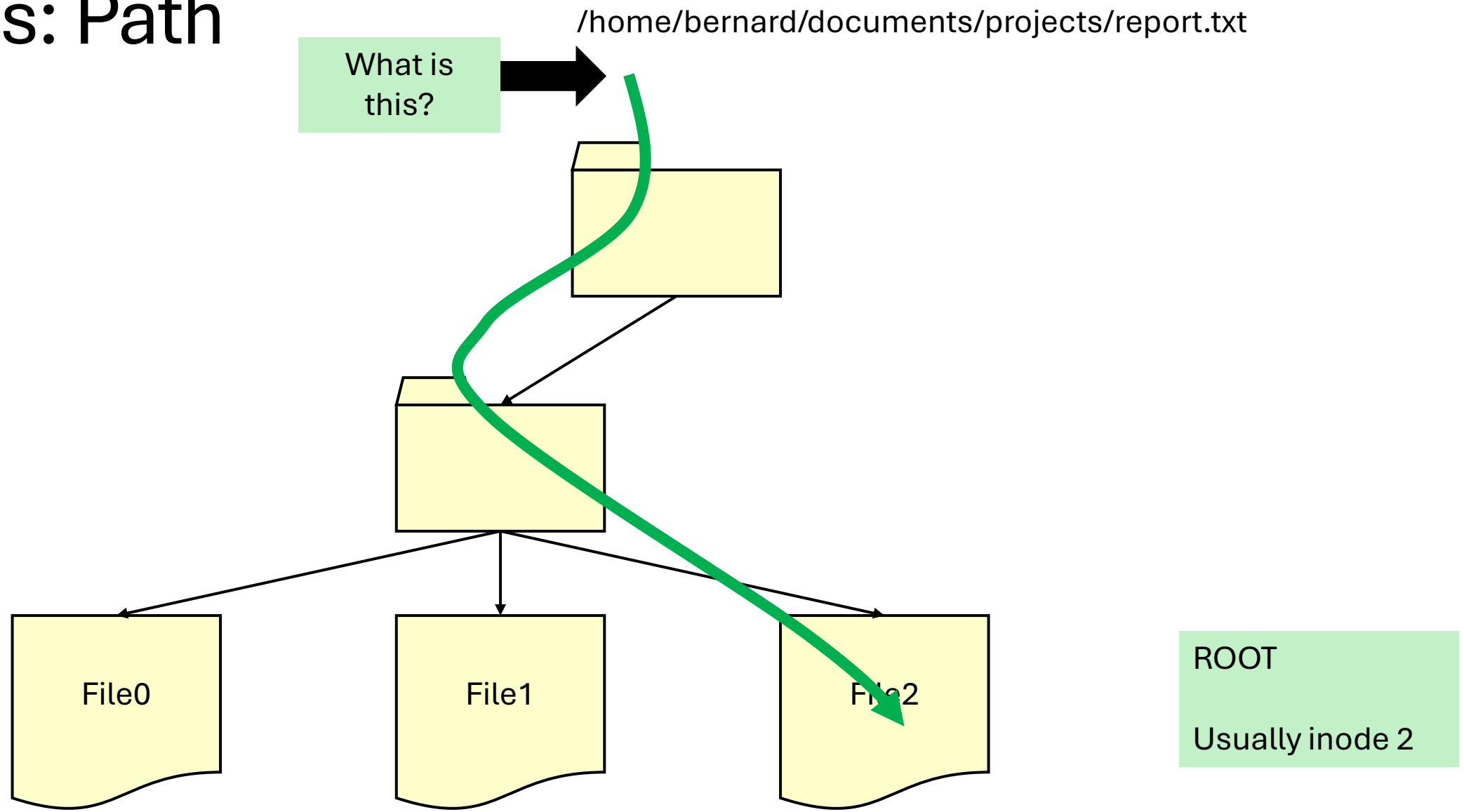
- Use words
- Map words to inodes
- Create “**DIRECTORY**”

Files: Path

/home/bernard/documents/projects/report.txt



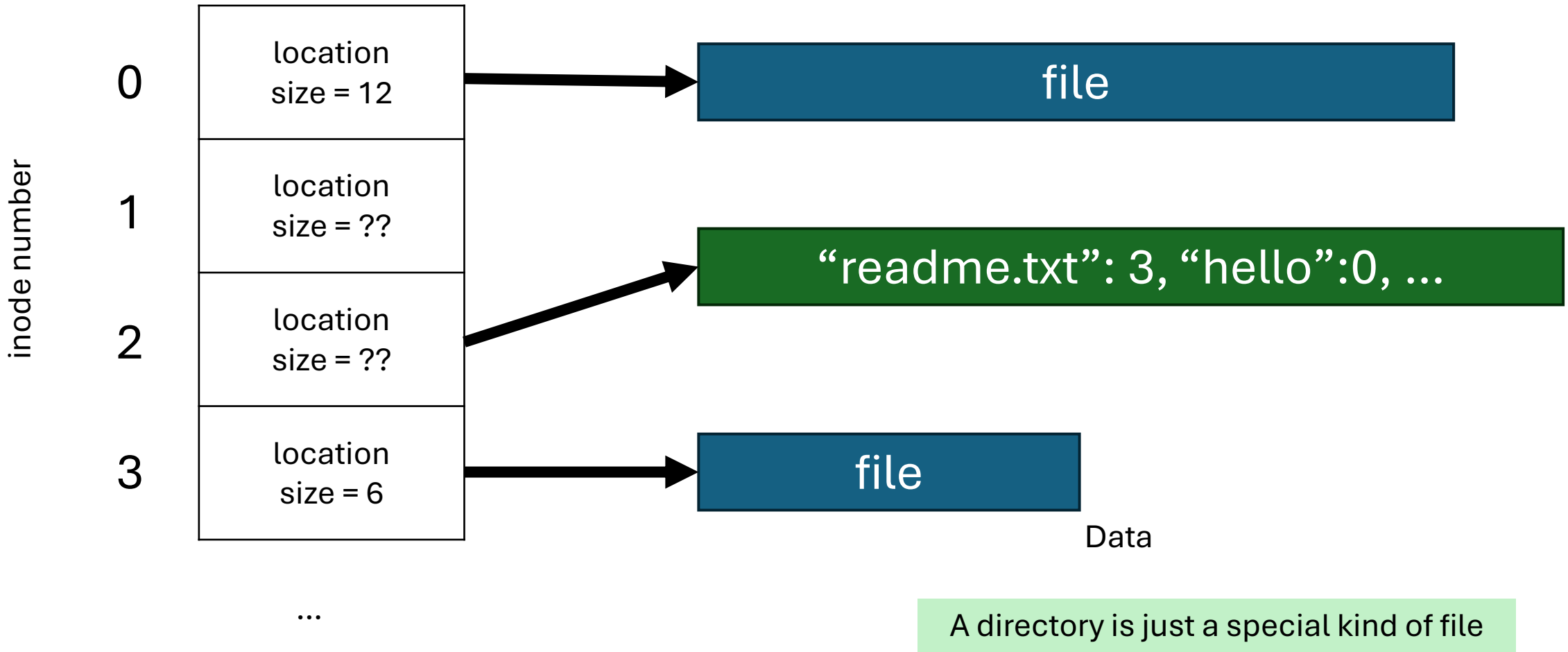
Files: Path



ROOT

Usually inode 2

Files: i-node



A directory is just a special kind of file

Directories

```
l:long
a:all (show hidden files)
```

me

my parent

`d...`

`ls -la`

```
drwxr-xr-x 4 emper emper  4096 Sep 15 13:10 .
drwxr-xr-x 3 emper emper  4096 Aug 20 17:00 ..
-rwxr-xr-x 1 emper emper 16208 Aug 27 09:03 EVEN
-rwxr-xr-x 1 emper emper 17184 Aug 29 15:25 MINISHELL
-rw-r--r-- 1 emper emper    26 Aug 19 11:31 c1
-rw-r--r-- 1 emper emper    36 Aug 19 11:31 c2
-rw-r--r-- 1 emper emper   819 Aug  4 10:42 even.c
-rw-r--r-- 1 emper emper  1589 Aug 20 16:24 even2.c
-rw-r--r-- 1 emper emper   511 Aug 27 09:02 even_ecd.c
-rw-r--r-- 1 emper emper   465 Aug 20 11:41 gold_output
-rw-r--r-- 1 emper emper    74 Aug 19 11:31 j1
-rw-r--r-- 1 emper emper   132 Aug 19 11:31 j2
-rw-r--r-- 1 emper emper    41 Aug 19 12:51 makefile
-rw-r--r-- 1 emper emper  7167 Aug 19 12:55 minishell.c
-rw-r--r-- 1 emper emper  4173 Aug 21 09:29 minishell3.c
-rw-r--r-- 1 emper emper  1895 Aug 21 09:29 minishell3.c:Zone.Identifier
-rw-r--r-- 1 emper emper  9819 Aug 20 16:59 minishell5.c
-rw-r--r-- 1 emper emper  4936 Aug 22 10:09 minishell_bw.c
-rw-r--r-- 1 emper emper  1883 Aug 22 10:09 minishell_bw.c:Zone.Identifier
-rw-r--r-- 1 emper emper  6711 Aug 20 11:18 minishell_correct.c
-rw-r--r-- 1 emper emper  3612 Aug 26 13:03 minishell_mh.c
-rw-r--r-- 1 emper emper  1821 Aug 26 13:03 minishell_mh.c:Zone.Identifier
-rwxr-xr-x 1 emper emper   463 Aug 21 11:34 quick_test.sh
drwxr-xr-x 2 emper emper  4096 Aug 19 11:16 student
drwxr-xr-x 2 emper emper  4096 Aug 19 09:25 testfiles
```

File Contents

What is inside?

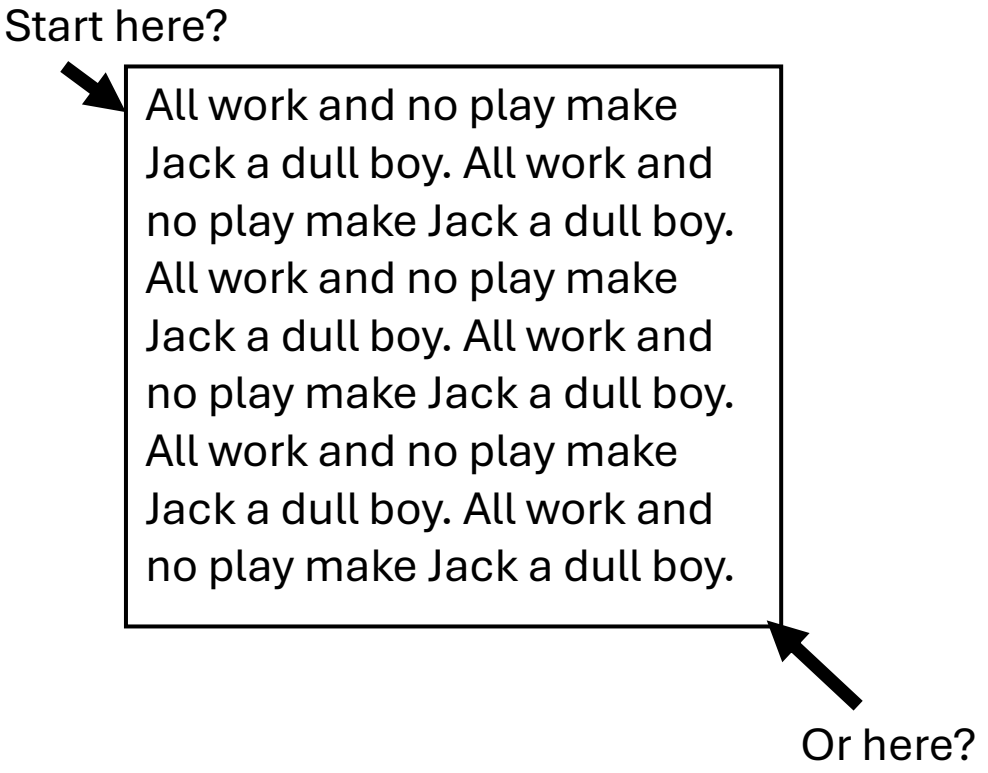
File Names: Extensions

What does a .txt file contain?

Reading and Writing

Each file has two locations:

- Start (can be used for over-writing)
- Offset (can be used for reading/appending)

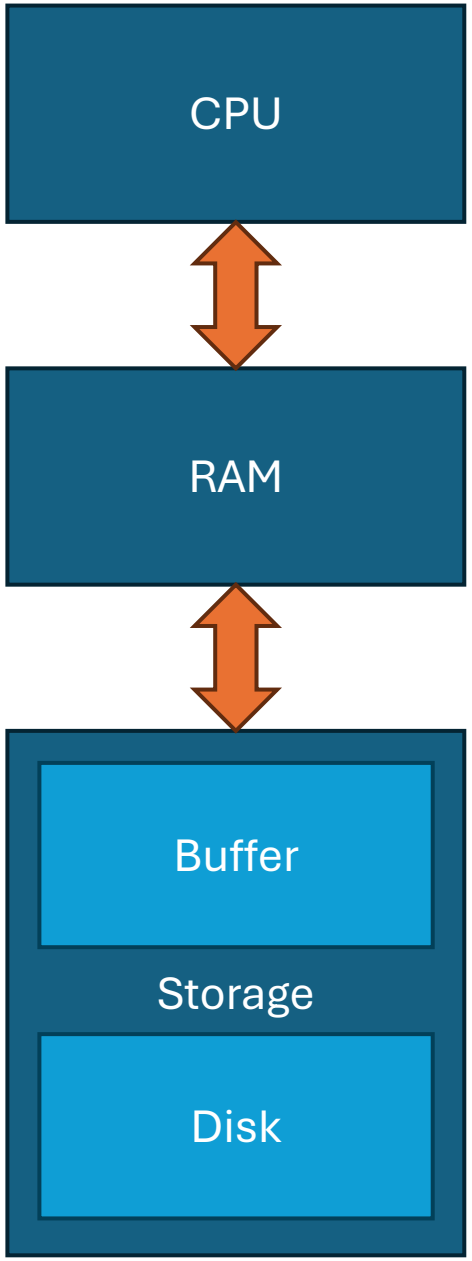


fsync

Hidden Lies

- When you call ‘write’, what is actually happening?

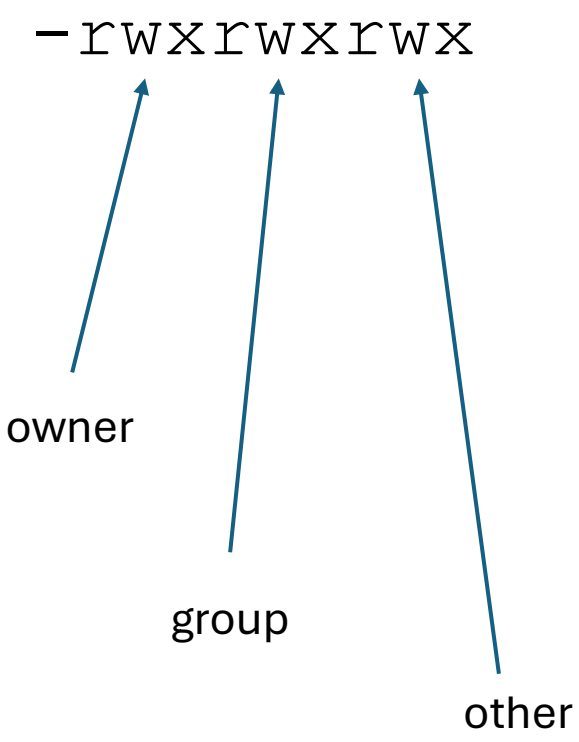
fsync forces all buffers to flush to disk (and usually) tells the disk to flush its write cache too



File Permissions

What is an isn’t allowed

Unix File Permissions




```
drwxr-xr-x 4 emper emper  4096 Sep 15 13:10 .
drwxr-xr-x 3 emper emper  4096 Aug 20 17:00 ..
-rwxr-xr-x 1 emper emper 16208 Aug 27 09:03 EVEN
-rwxr-xr-x 1 emper emper 17184 Aug 29 15:25 MINISHELL
-rw-r--r-- 1 emper emper    26 Aug 19 11:31 c1
-rw-r--r-- 1 emper emper    36 Aug 19 11:31 c2
-rw-r--r-- 1 emper emper   819 Aug  4 10:42 even.c
-rw-r--r-- 1 emper emper  1589 Aug 20 16:24 even2.c
-rw-r--r-- 1 emper emper   511 Aug 27 09:02 even_ecd.c
-rw-r--r-- 1 emper emper   465 Aug 20 11:41 gold_output
-rw-r--r-- 1 emper emper    74 Aug 19 11:31 j1
-rw-r--r-- 1 emper emper   132 Aug 19 11:31 j2
-rw-r--r-- 1 emper emper    41 Aug 19 12:51 makefile
-rw-r--r-- 1 emper emper  7167 Aug 19 12:55 minishell.c
-rw-r--r-- 1 emper emper  4173 Aug 21 09:29 minishell3.c
-rw-r--r-- 1 emper emper  1895 Aug 21 09:29 minishell3.c:Zone.Identifier
-rw-r--r-- 1 emper emper  9819 Aug 20 16:59 minishell5.c
-rw-r--r-- 1 emper emper  4936 Aug 22 10:09 minishell_bw.c
-rw-r--r-- 1 emper emper  1883 Aug 22 10:09 minishell_bw.c:Zone.Identifier
-rw-r--r-- 1 emper emper  6711 Aug 20 11:18 minishell_correct.c
-rw-r--r-- 1 emper emper  3612 Aug 26 13:03 minishell_mh.c
-rw-r--r-- 1 emper emper  1821 Aug 26 13:03 minishell_mh.c:Zone.Identifier
-rwxr-xr-x 1 emper emper   463 Aug 21 11:34 quick_test.sh
drwxr-xr-x 2 emper emper  4096 Aug 19 11:16 student
drwxr-xr-x 2 emper emper  4096 Aug 19 09:25 testfiles
```

Access Control Lists

Allow fine-grain control over permissions

Not always enabled by default

User	Rights
me	rwx
Alice	r
Bob	w

Directories

What are these?

Directory

Structure

- Varies across OS's
- Basically a file
- Store directory entries in data blocks (how long is a filename?)
  - Large directories split across blocks
  - Inode distinguishes the type
- Lists (small directories)
- B-Trees (giant directories)

Links

To the past?

Hard Link

`link(old path, new path)`

Link a new filename to an old file

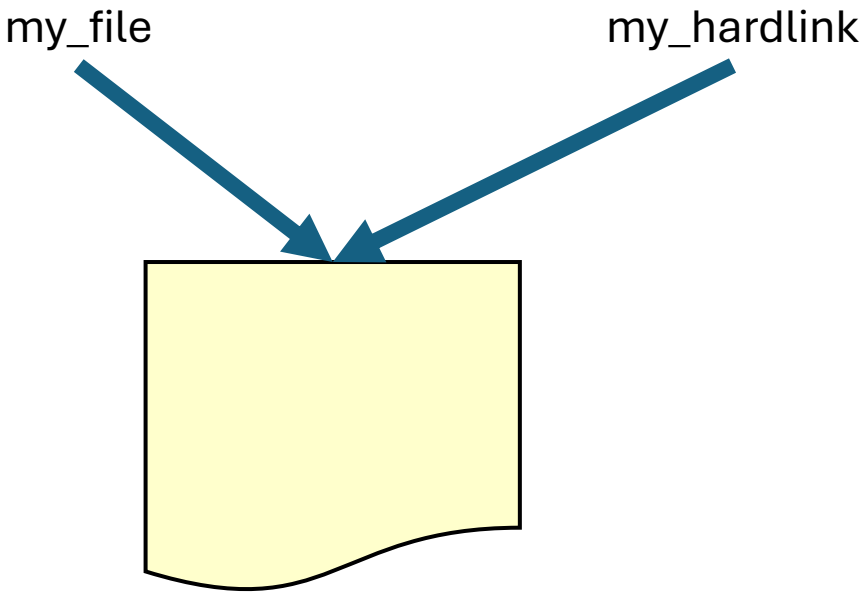
Two pointers at the same file

Create:

`ln`

Remove:

`rm`



Only works within the same filesystem

Cannot work on directories normally (dangerous)

Symbolic Links

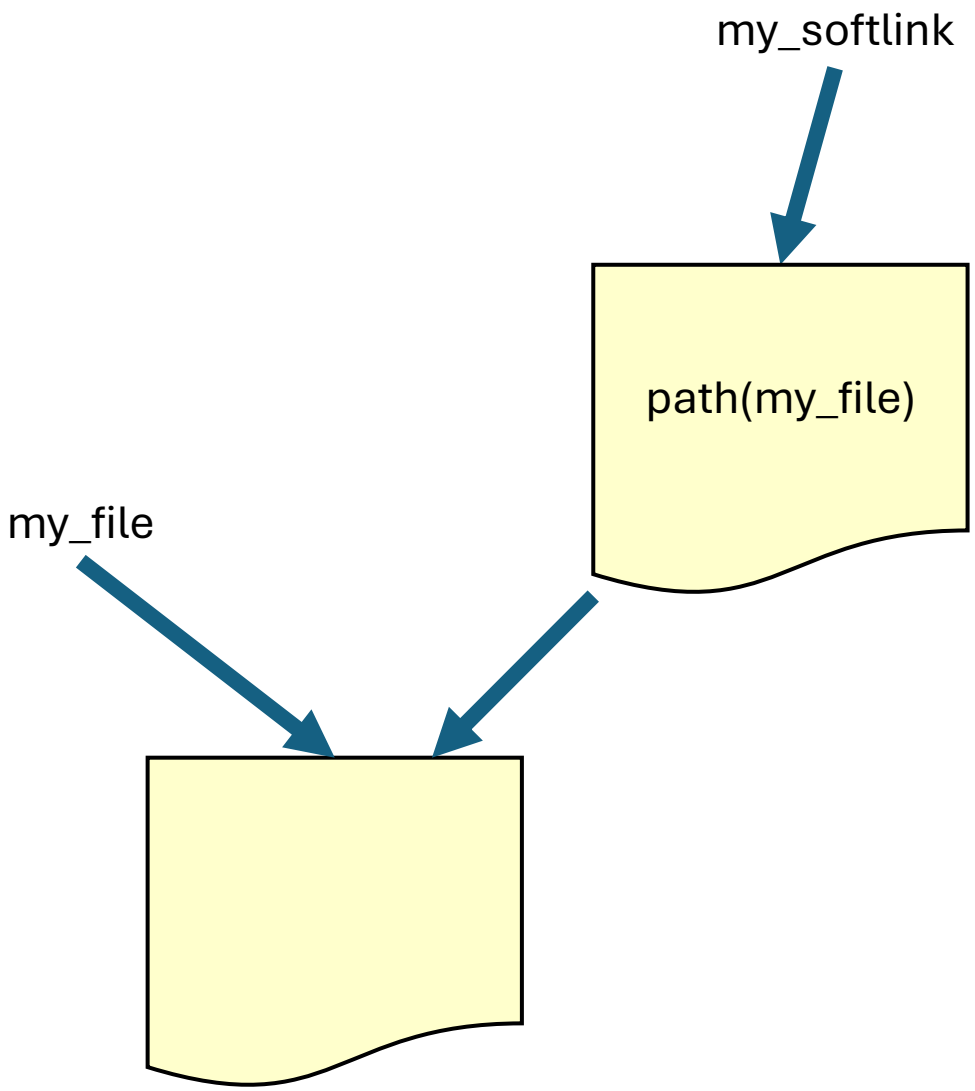
Definition:

Examples:

- ~

Usage:

```
ln -s file1 file2
```



Deletion Rules

Hard Link

```
echo hello > file1
ln file1 file2
cat file2
```

➢ “hello”

```
rm file1
cat file2
```

➢ “hello”

Soft Link

```
echo hello > file1
ln -s file1 file2
cat file2
```

➢ “hello”

```
rm file1
cat file2
```

➢ File2: No such file

File Manipulation

Some basics

Function: rename

What does this mean?

- Find names (are they legal)
- Checks permissions (may you)
- Check old and new **inodes**
- **Lock** both directories
- Unlink ‘old’ new
- Add new entry
- Remove old entry

```
rename(char * old, char * new)
```

This will need to be atomic to avoid disasters…

Function: remove

What does this mean?

- Calls ‘unlink()’
- Find the inode associated with the file
- Decrement st_nlink
  - May result in inode being ‘free’
  - Will depend on **file descriptors**
- Remove entry from parent directory

```
rm file
```

This will need to be atomic to avoid disasters…

File Descriptors

Handling Files

File Descriptors

Problem:

- We have written code which references a file
- To do so, we need to find it
  - Once we've found it… do we need to find it again?

File Descriptors

Processes and FDs

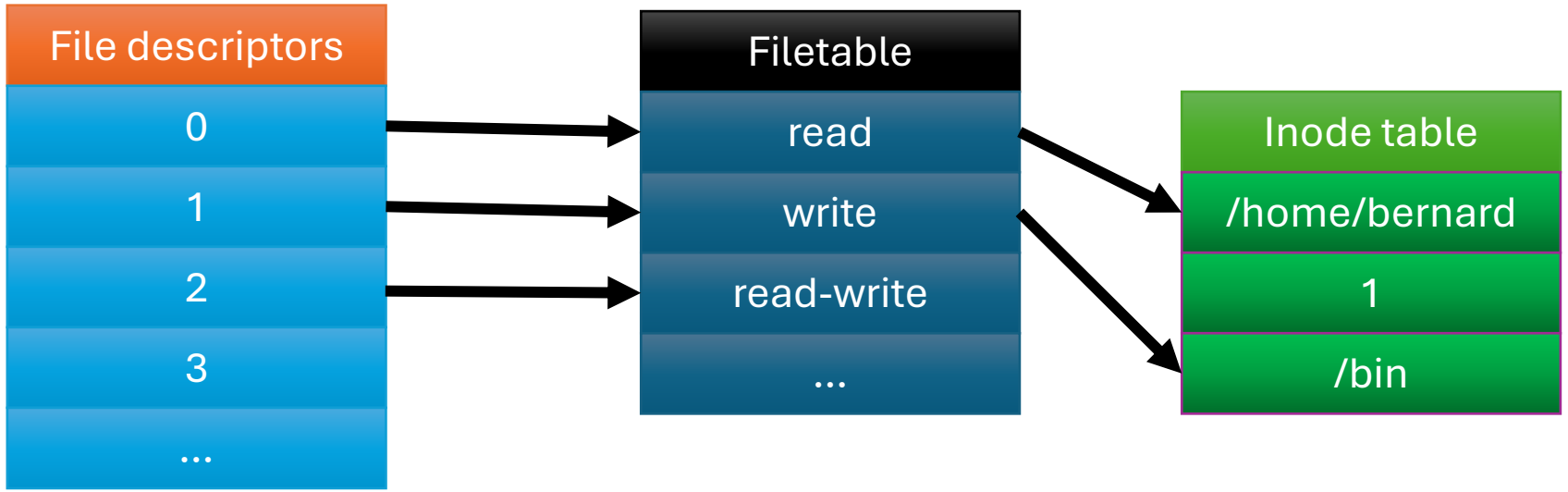
A **file descriptor** is a connection from a process to a file

- Each file descriptor is identified by a **unique** number
- Each process has its own set of **file descriptors**

File Table

- Store the mappings of processes and **file descriptors**… somewhere

File Descriptor Table



In the kernel

Stores the **modes** (read/write etc)

Where?

Important:

- **Processes** know file descriptors, the kernel knows where they lead, so if a **process** gives the kernel a **file descriptor** (i.e. I would like 0 – stdin or 205 – my_text.txt) the kernel can provide a place to write
- **Processes do not** have direct access…

File Descriptors: Defaults

0: standard in stdin

1: standard out stdout

2: standard error stderr

Next one goes here

Files API

```
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 Descriptors: strace

strace cat quick_test.sh

```
openat(AT_FDCWD, "quick_test.sh", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0755, st_size=463, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7625d43000
read(3, "echo \"## Testing CD ##\"\n(echo ec"..., 131072) = 463
write(1, "echo \"## Testing CD ##\"\n(echo ec"..., 463echo "## Testing CD ##"
```

read(3 ....

File descriptor!

File Names

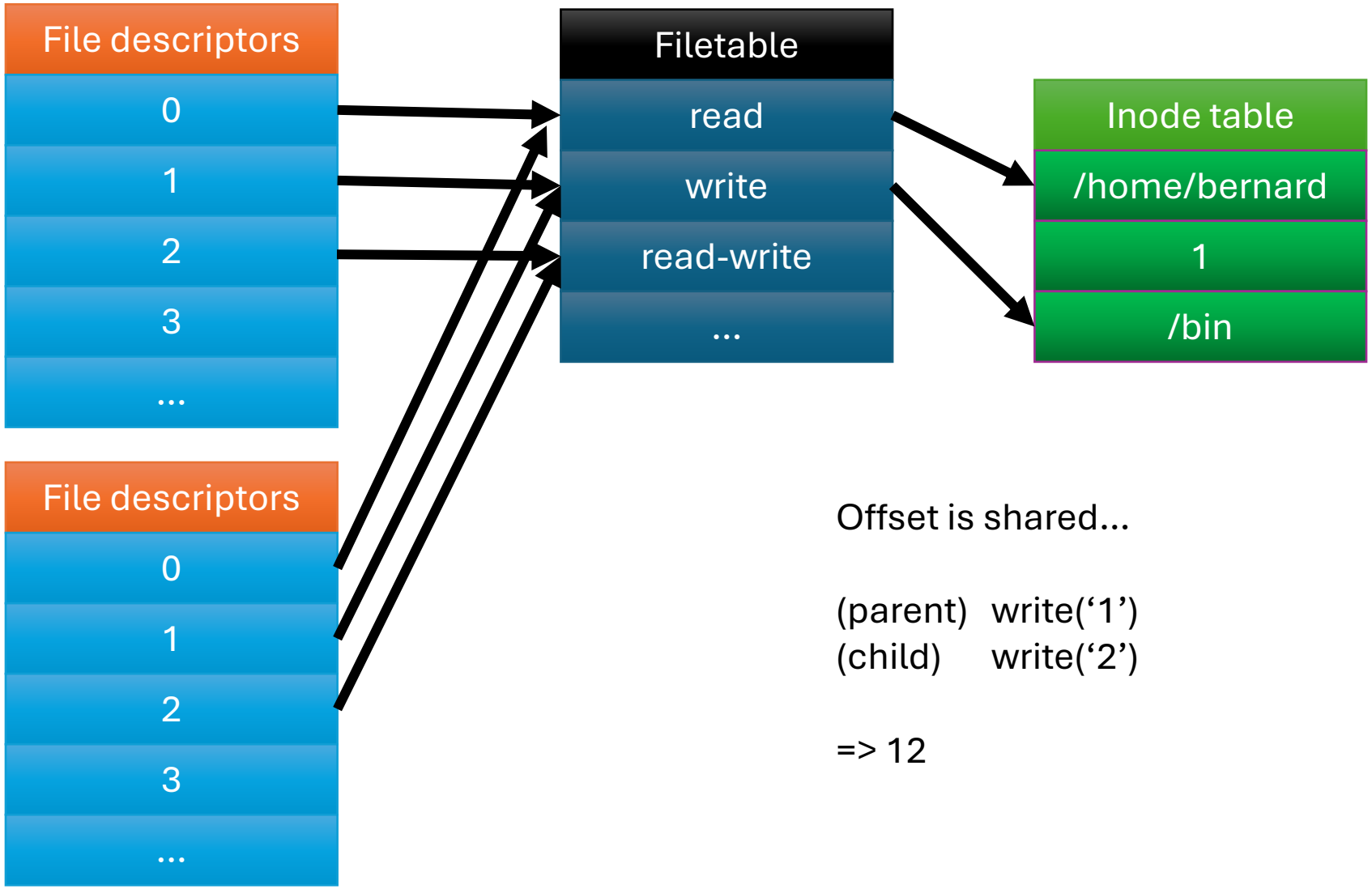
Three types of names:

- inode — a storage-centric way of talking about files
- path — a human parseable way of finding the file
- file descriptor — a process-centric way of talking about files

File Descriptors and Processes

How does this work?

Forking



Offset is shared...

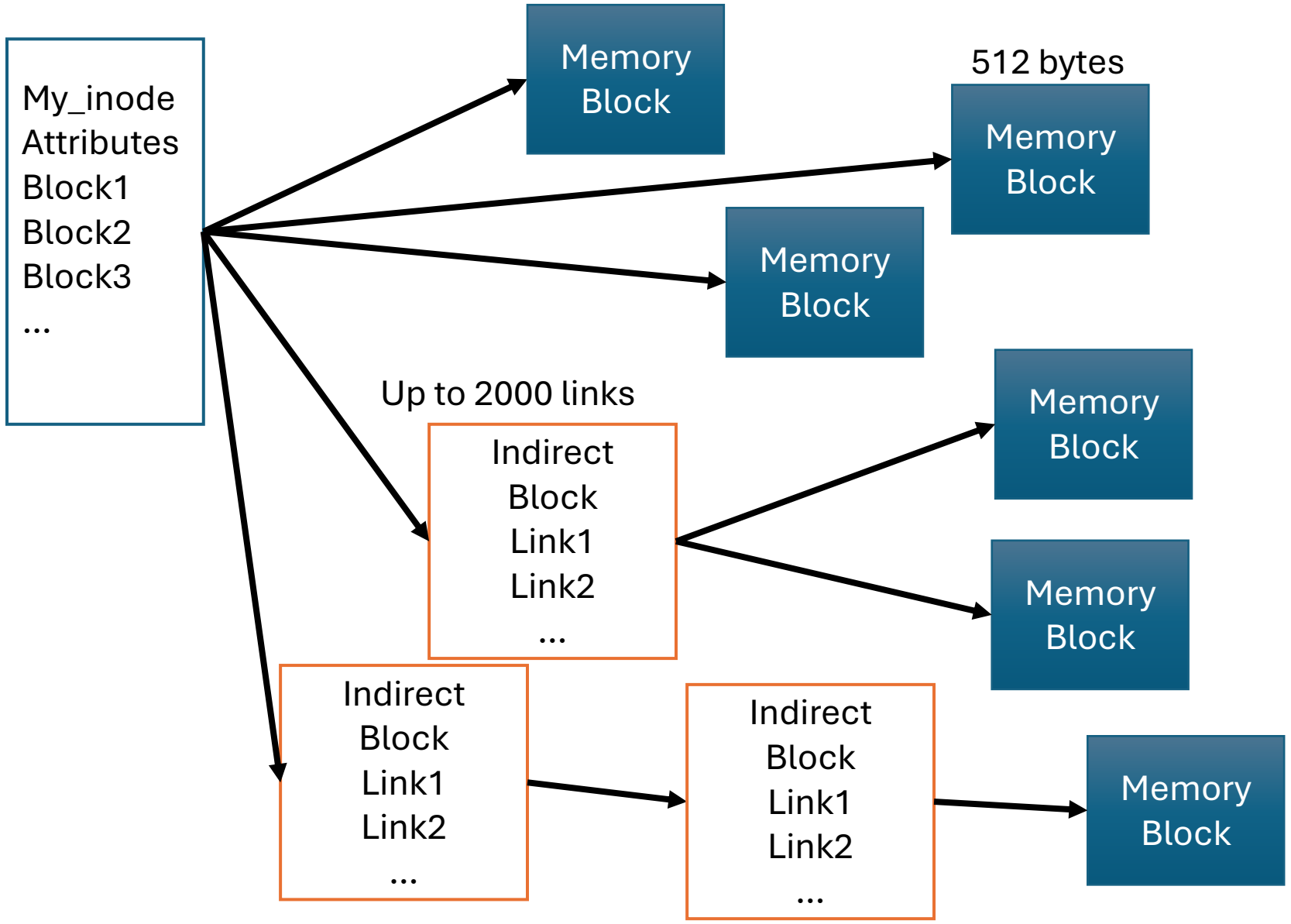
(parent) write('1')
(child) write('2')

=> 12

I-Nodes

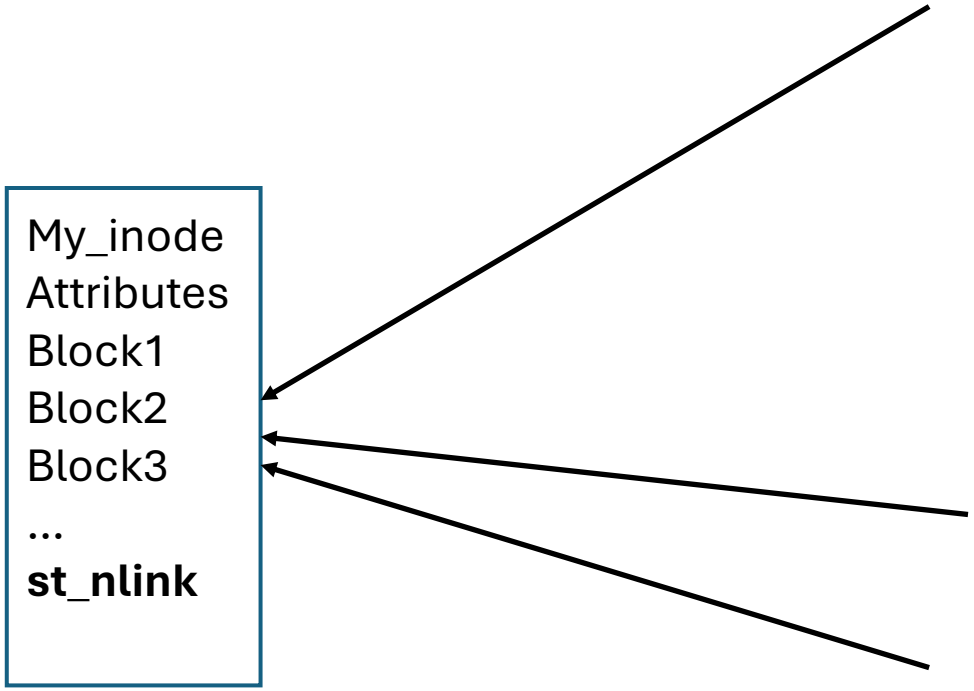
Some revision...

Inodes!


My_inode
Attributes
Block1
Block2
Block3
...
Memory Block
512 bytes
Memory Block
Memory Block
Up to 2000 links
Indirect Block
Link1
Link2
...
Memory Block
Memory Block
Indirect Block
Link1
Link2
...
Indirect Block
Link1
Link2
...
Memory Block

Inodes: Recycling

Each inode maintains a ‘is pointing’ list (allowing deletion/recycling)



unlink(): file system
close(): processes

Multiple File Systems

When there are many

Multi-File System

Main Disk

Backup

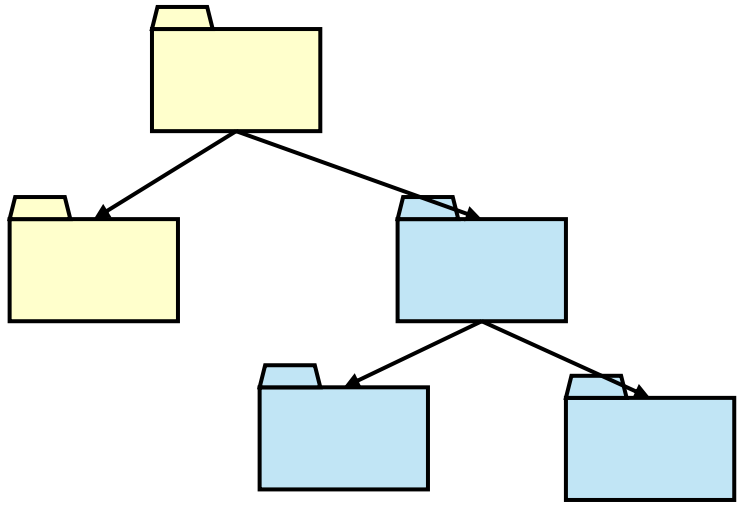
USB?

Multi-File System: Mounting

`mount files`

Mount

- Attaches a set of files/file system to the linux tree



File System Implementation

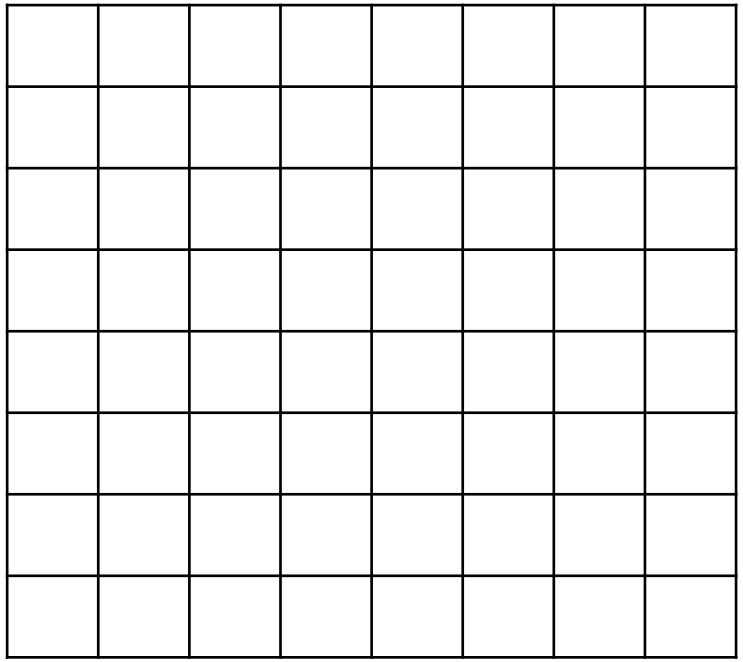
What goes there

Persistent Storage

Question #1

- Haven’t we already done this?
  - Translation Lookaside Buffers
  - Page Directories/Page Tables
  - Fragmentation

Memory and Storage aren’t exactly the same thing



Persistent Storage

Question #2

- What is the same?
  - 4KB is a convenient size
  - Fragmentation

Question #3

- What changes?
  - Speed issues are more relevant
  - Fragmentation

Storage Strategies

Approaches

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

Concerns

- Fragmentation
- Dynamic file-sizes
- Performance
  - Sequential accesses
  - Random accesses
- Meta-data overhead

Bad Idea #1: Contiguous Allocation

Description

- Allocate each file contingous sectors on the disk

Why is this good/awful?

Bad Idea #1: Contiguous Allocation

Good

- Good for reading and writing
- No metadata overhead

Awful

- How much space per file?
  - Fixed?
  - Extra for growth?
- External Fragmentation
- How to find space for a file?

Bad Idea #2: Extent

Description

- Allocate each file multiple contingous sectors on the disk

Why is this good/awful?

Bad Idea #2: Extent

Good

- Good for reading and writing
  - Worse than contiguous
- Small metadata overhead
- How much space per file?
  - Can grow!

Awful

- External Fragmentation
  - Better than contiguous
- How to find space for a file?

Bad Idea #3: Linked List

Description

- Allocate each file multiple contingous sectors on the disk

Why is this good/awful?

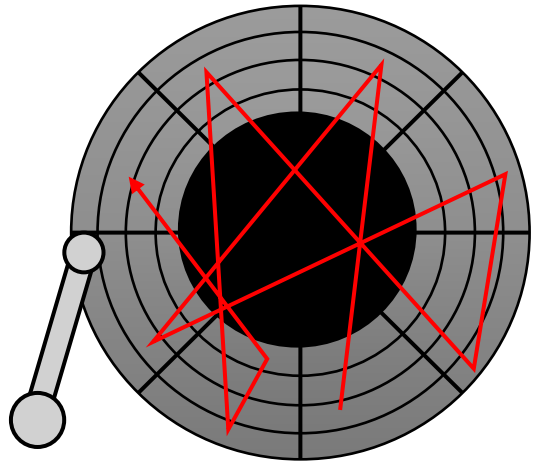
Bad Idea #3: Linked List

Good

- Small metadata overhead ☹
  - Point to the next block
- How much space per file?
  - Can grow!
- External Fragmentation
  - Easy
- Sequential reading and writing
  - OK

Awful

- Random reading and writing
  - Really, really bad



Idea #4: File-Allocation Table (FAT)

Block	Block
0	15
1	22
2	17
3	7
…	…

Description

- Create a table!
- Store all the links (simplified)

Why is this good/awful?

Idea #4: FAT

Good

- Small metadata overhead ☹
  - Only in one place – corruption
- How much space per file?
  - Can grow!
- External Fragmentation
  - Easy
- Sequential reading and writing
  - OK

Awful

- Each read/write requires an extra read
  - Can be cached to avoid really bad performance
  - Much better than a standard Linked-List
- Small metadata overhead ☹
  - Only in one place – *corruption*

Idea #5: Indexed Allocation

Description

- Create an ‘index block’ for each file
- Index block created when file created
- Index block contains all the pointer stuff

Why is this good/awful?

Idea #5: Indexed Allocation

Good

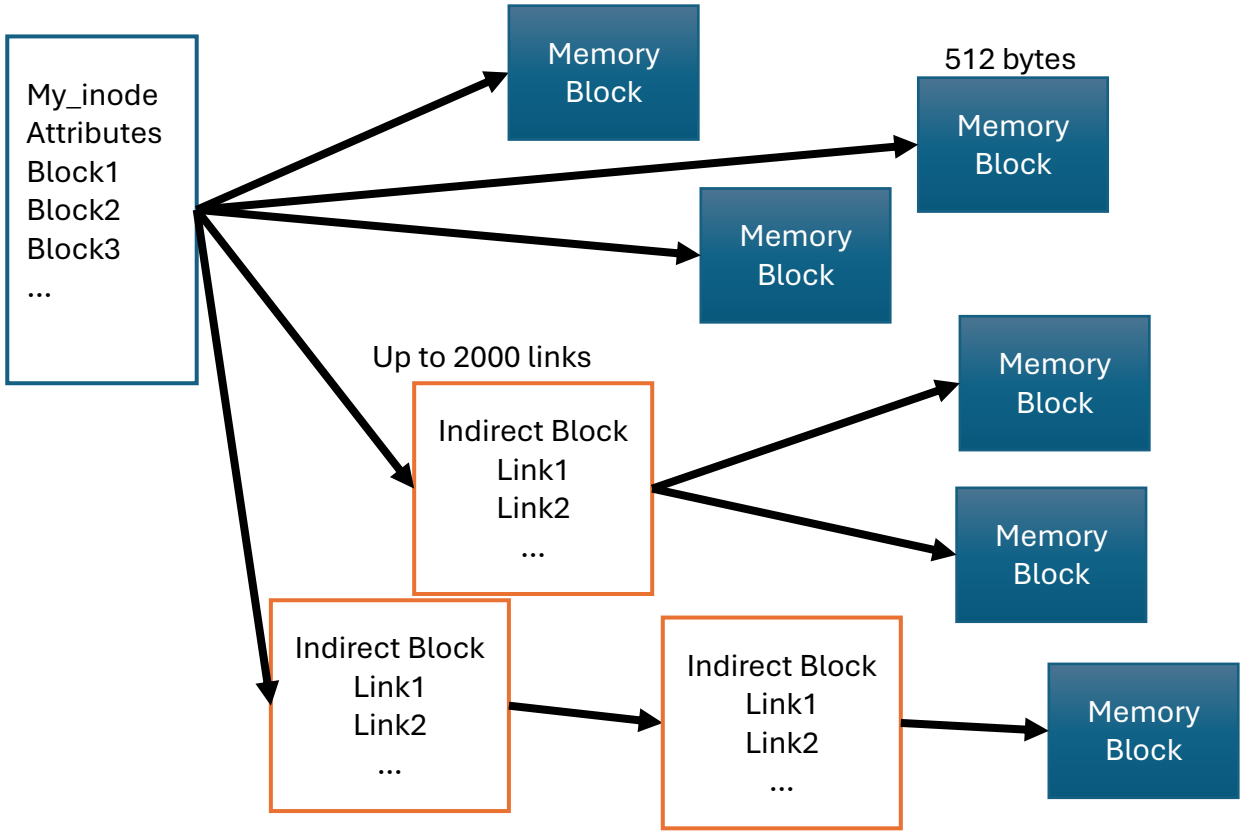
- How much space per file?
  - Can grow!
- External Fragmentation
  - Easy
- Sequential reading and writing
  - OK
- Random reading and writing
  - OK

Awful

- Large metadata overhead ☹
  - Every tiny file needs a ‘friend’
- There is a limit to file size
  - One block worth of blocks!

Idea #6: Multi-Level Indexed Allocation

inodes!



Why is this good/awful?

Idea #6: inodes

Good

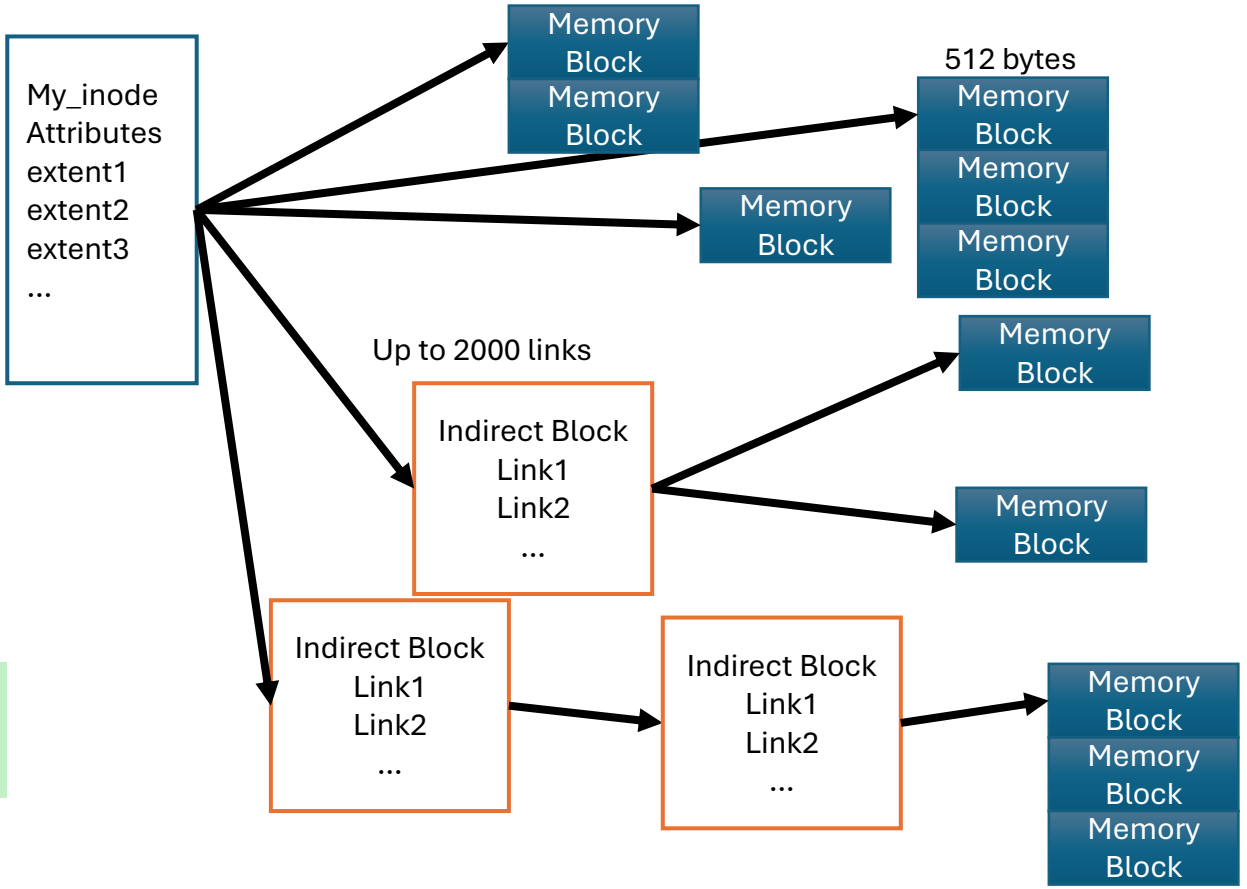
- How much space per file?
  - Can grow!
- External Fragmentation
  - Easy
- Sequential reading and writing
  - OK
- Random reading and writing
  - OK

Awful

- Large metadata overhead ☹
  - Every tiny file needs a ‘friend’
- ~~There is a limit to file size~~
  - ~~One block worth of blocks!~~
- Big files can be a bit slow
  - Triple indirection??

Idea #7: Multi-Level Indexed Extents

inodes!
ext4



Instead of storing individual blocks, store ‘contiguous extents’

Why is this good/awful?

Idea #7: Multi-Level Indexed Extents

Good

- How much space per file?
  - Can grow!
- External Fragmentation
  - Better than blocks
- Sequential reading and writing
  - Better!
- Random reading and writing
  - OK

Awful

- Large metadata overhead ☹
  - Every tiny file needs a ‘friend’
- Big files can be a bit slow
  - Triple indirection??

I-Nodes

Some new stuff

I-Nodes

Contents

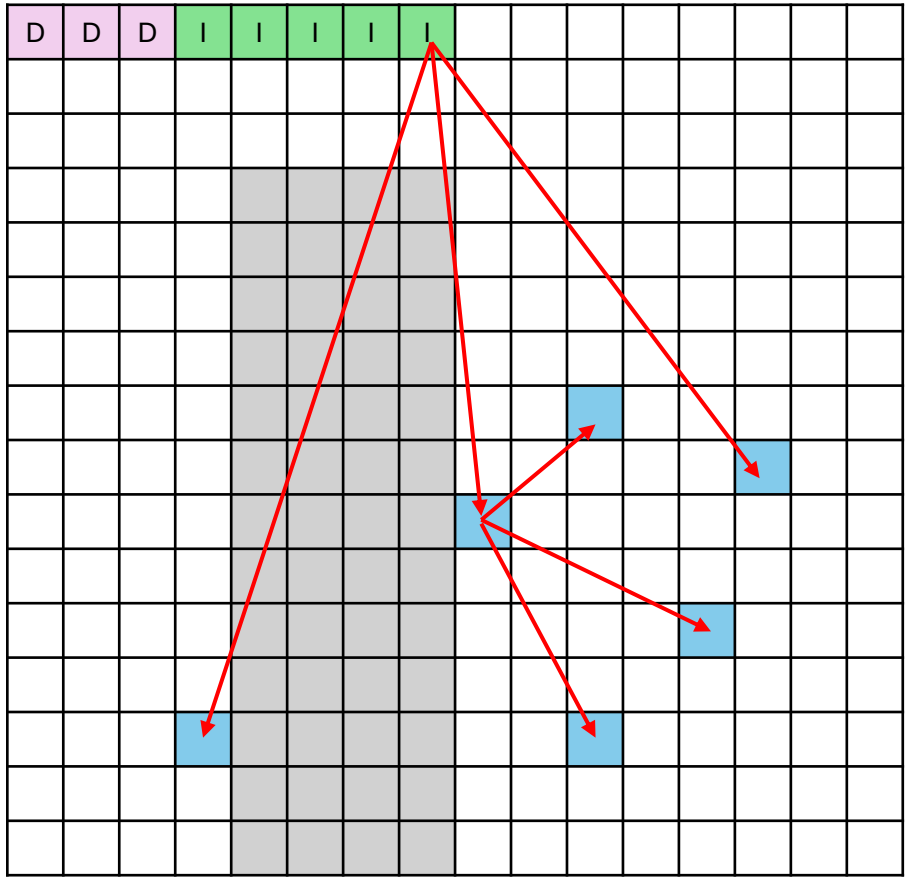
- type — file/directory/…?
- uid — owner
- rwx — permissions
- size — bytes
- blocks — total
- time
  - Access time
  - Create Time
- st_nlinks
- address[N] — (N data blocks)

Inodes!

We need:

- Superblock: Filesystem metadata
- Block bitmap: Which blocks are free?
- Inode bitmap: Which inodes are free?
- Inode table: Array of inodes for files
- Data blocks: The actual data

This is distributed into block groups



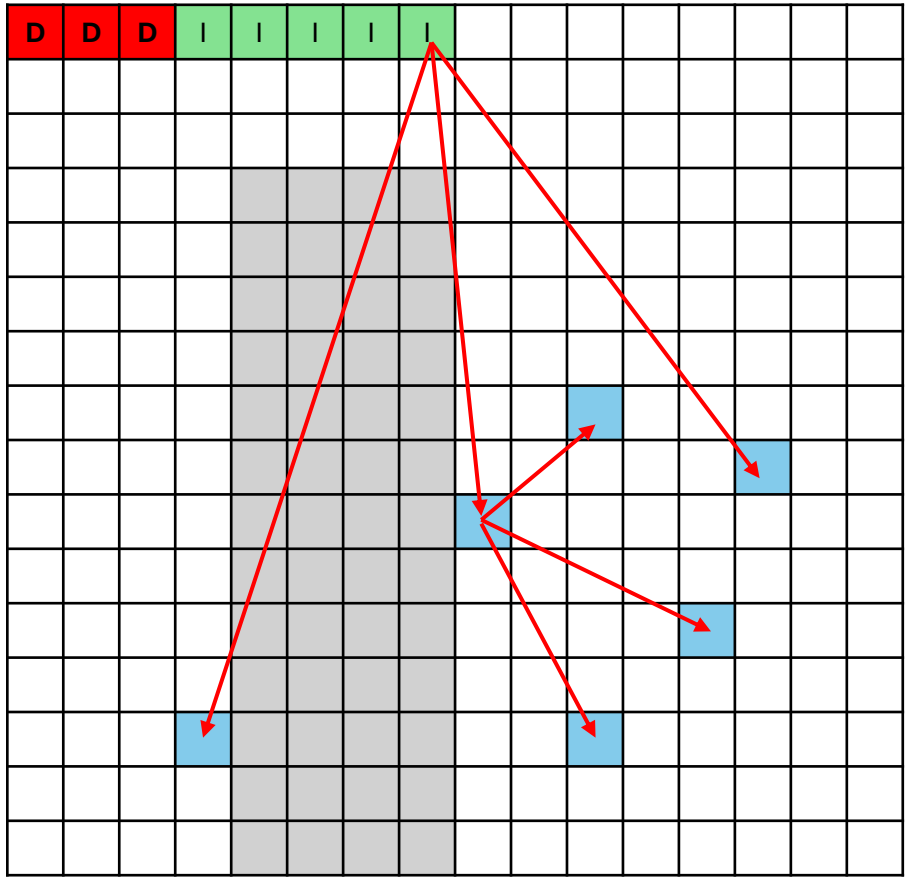
Filesystem Metadata

Basic Stuff

- Block size? Multiblock?
- # of inodes?

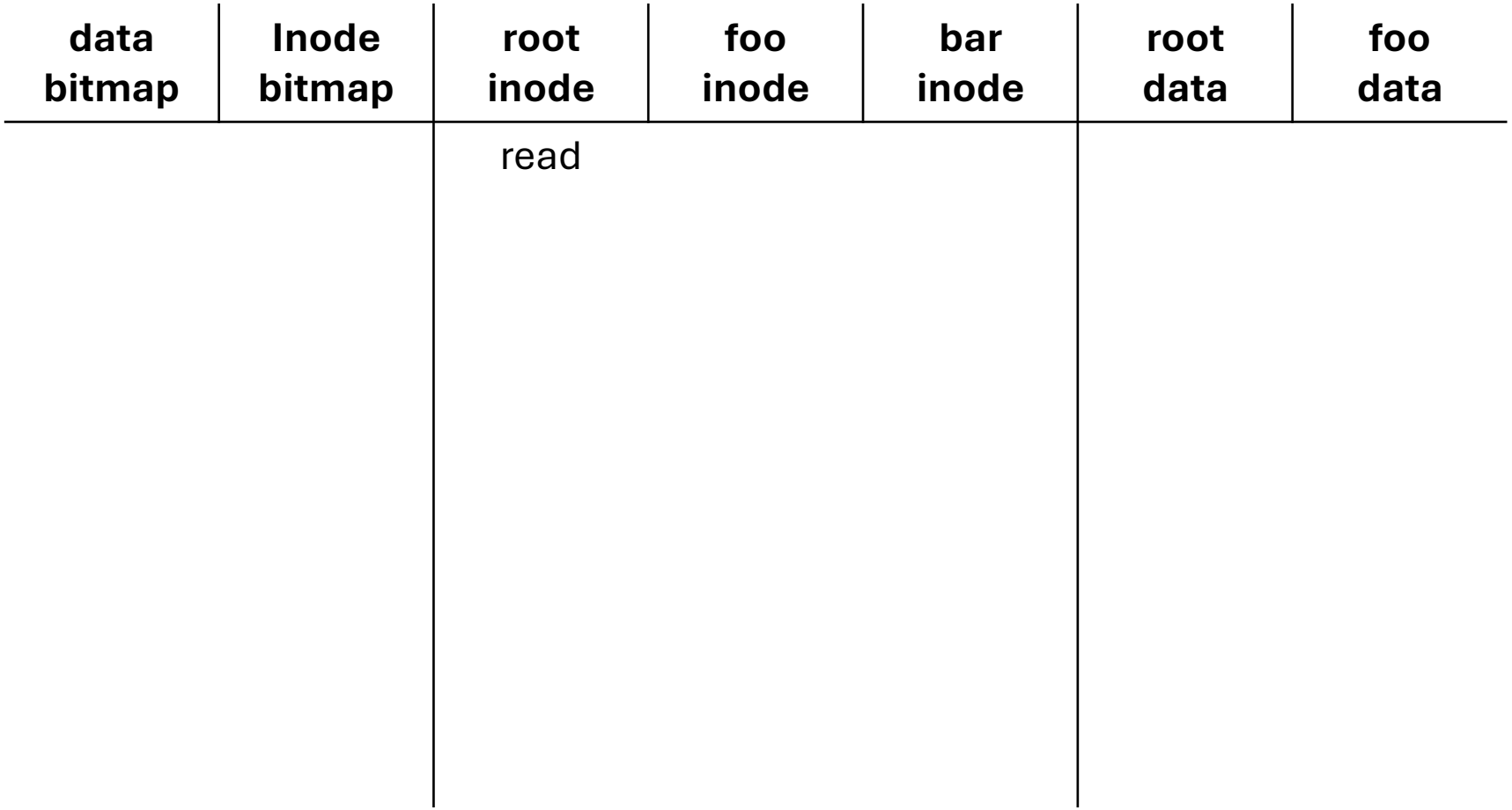
How do we find free data/inodes?

- Free list/bitmap
  - Could allow inconsistency…
  - Power outages



Start at root

Example: create /foo/bar



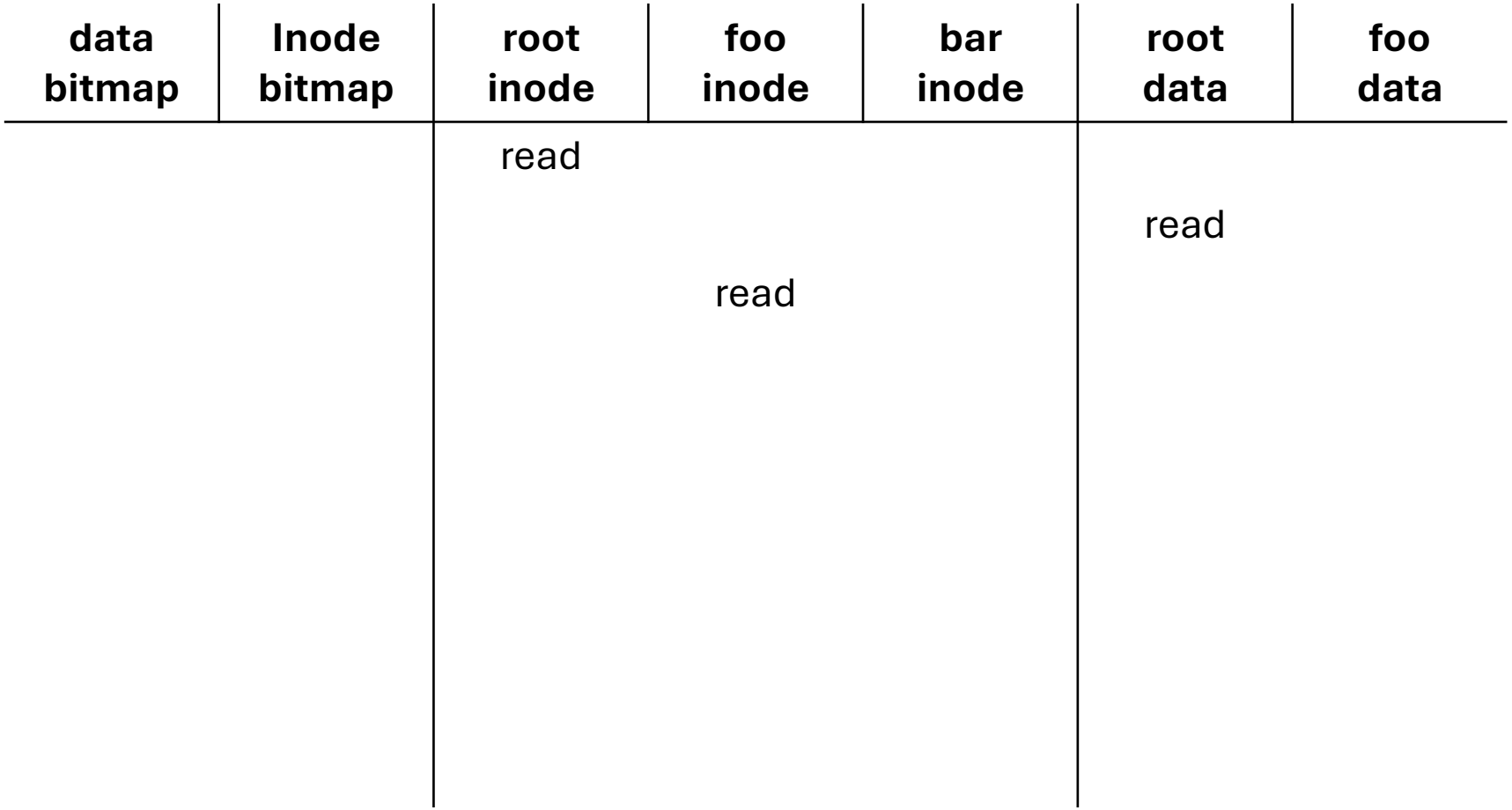
Example: create /foo/bar

Read root content (directory)

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

Find foo inode

Example: create /foo/bar



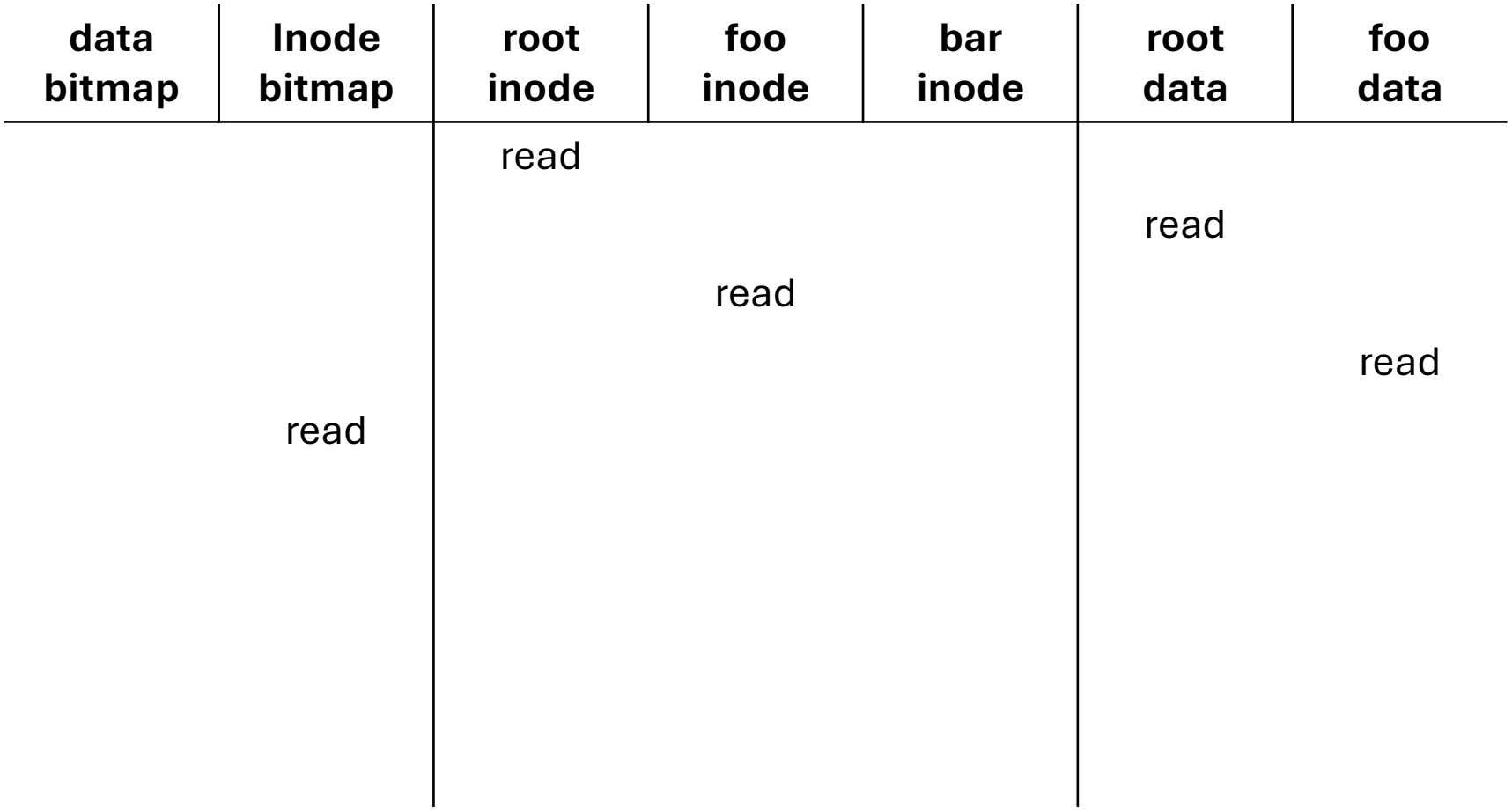
Example: create /foo/bar

Read foo content (directory)

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

We need a new inode

Example: create /foo/bar



We need a new inode

Example: create /foo/bar

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

Example: create /foo/bar

foo needs to know about this new entry

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

bar needs some love

Example: 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		

bar needs some love

Example: 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		

Example: create /foo/bar

foo has a new friend
st_nlink

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

Example: open /foo/bar

Once we get to bar, we have what we need

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

Example: write /foo/bar

Starting with bar
(after open)

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

Example: write /foo/bar

We're going to update the actual blocks being used

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

Example: write /foo/bar

Then actually write the new data

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

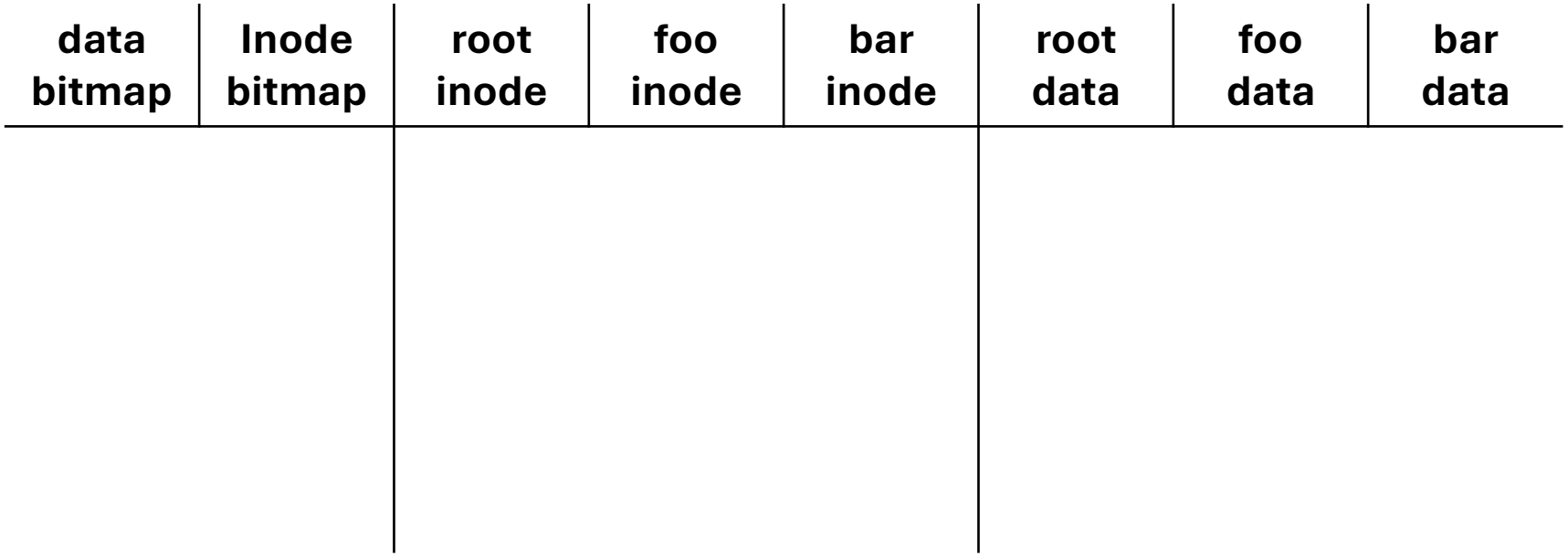
Example: write /foo/bar

Then we update the inode with the new details

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

No disk

Example: close /foo/bar



Optimisation

Caching

- Smart page replacement avoid reloading data

Read

- **Read-ahead/Prefetching**: Read the next block for fun

Reliability

- **Journaling:** Write sequentially to journal, then apply writes (crash recovery)

Writes

- **Write-back cache:** Queue writes in batches where possible.
- **Write Coalescing:** Turn many writes into one big sequential write
- **Locality:** Put inodes/blocks close together where possible
- **Pre-allocation:** Reserve contiguous blocks when creating/extending a file (avoiding fragmentation)

Summary

- RAID
- File Systems
- Inodes
- Storage

Questions?