



# Persistence: File Systems and RAID

CS 571: *Operating Systems (Spring 2021)*

Lecture 10

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials by Remzi Arpacı-Dusseau.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

# File System Abstraction

# What is a File?

- File: Array of bytes
  - Ranges of bytes can be read/written
- File system (FS) consists of many files
- Files need names so programs can choose the right one

# File Names

- Three types of names (abstractions)
  - **inode** (low-level names)
  - **path** (human readable)
  - **file descriptor** (runtime state)

# Inodes

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- Numbers may be recycled after deletes

# Inodes

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- Numbers may be recycled after deletes
- Show inodes via `stat`
  - `$ stat <file or dir>`

# 'stat' Example

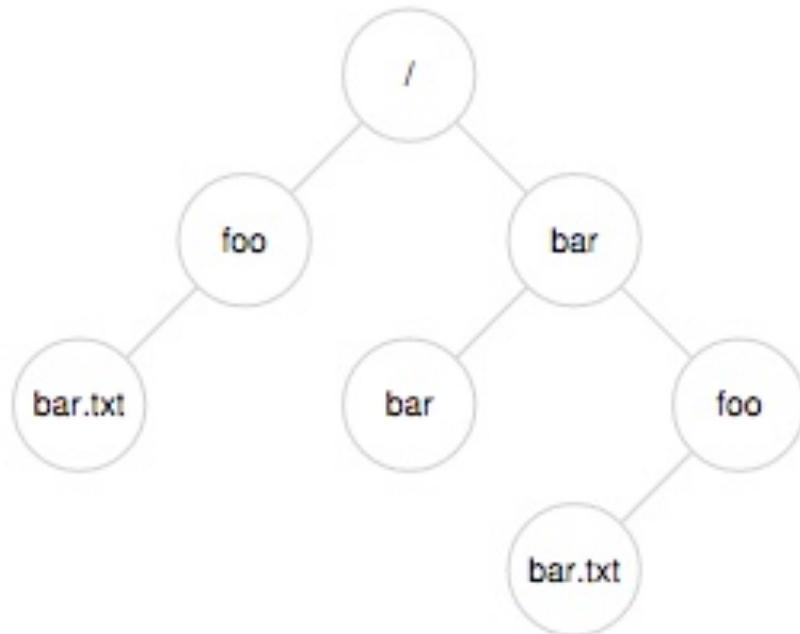
```
PROMPT>: stat test.dat
File: 'test.dat'  Size: 5      Blocks: 8      IO Block: 4096   regular
file
Device: 803h/2051d      Inode: 119341128  Links: 1
Access: (0664/-rw-rw-r--) Uid: ( 1001/      yue)  Gid: ( 1001/      yue)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2015-12-17 04:12:47.935716294 -0500
Modify: 2014-12-12 19:25:32.669625220 -0500
Change: 2014-12-12 19:25:32.669625220 -0500
Birth: -
```

# Path (multiple directories)

- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs

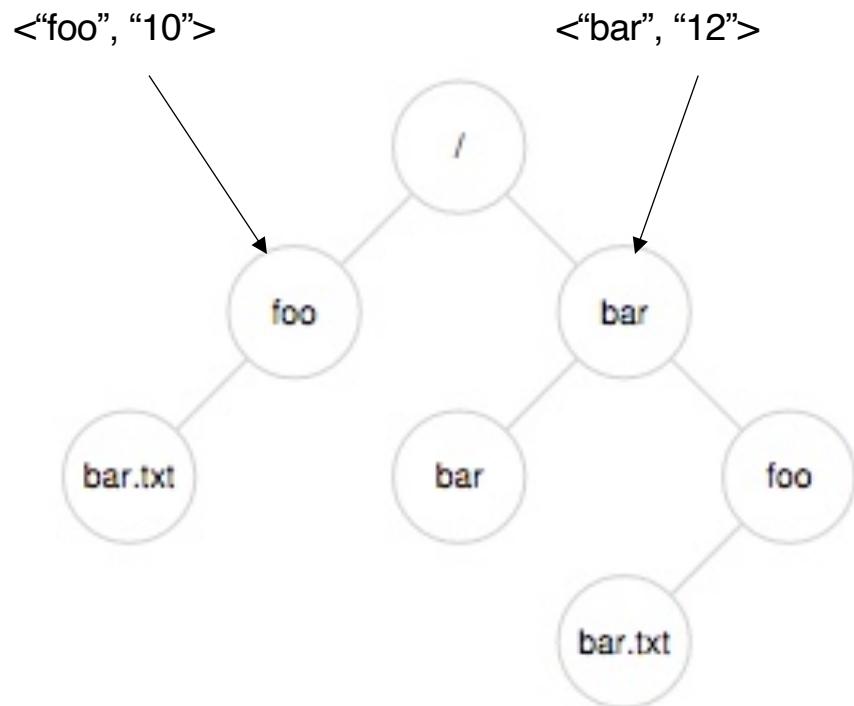
# Path (multiple directories)

- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs



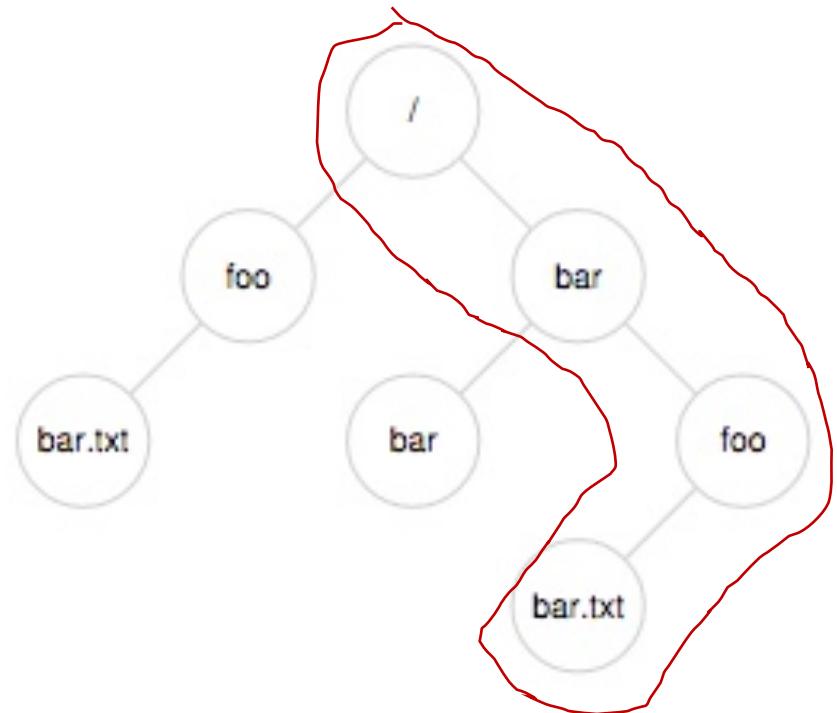
# Path (multiple directories)

- A directory is a file
  - Associated with an inode
- Contains a list of `<user-readable name, low-level name>` pairs



# Path (multiple directories)

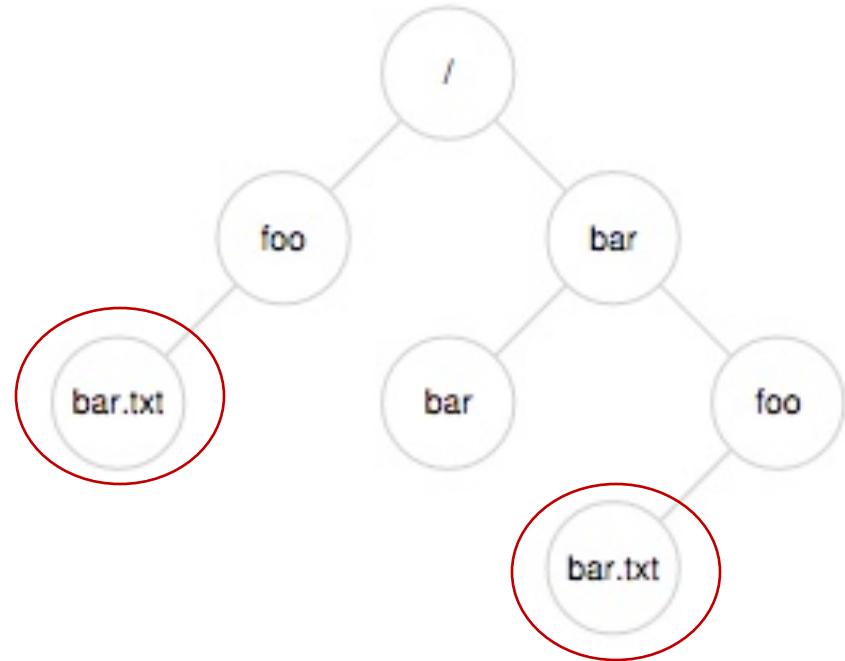
- A directory is a file
  - Associated with an inode
- Contains a list of `<user-readable name, low-level name>` pairs
- Directory tree: reads for getting final inode called **traversal**



[traverse /bar/foo/bar.txt]

# File Naming

- Directories and files can have the same name as long as they are in different locations of the file-system tree
- .txt, .c, etc.
  - Naming convention
  - In UNIX-like OS, no enforcement for extension name



# Special Directory Entries

```
prompt> ls -al
```

```
total 216
```

```
drwxr-xr-x 19 yue staff 646 Nov 23 16:28 .
```

```
drwxr-xr-x+ 40 yue staff 1360 Nov 15 01:41 ..
```

```
-rw-r--r--@ 1 yue staff 1064 Aug 29 21:48 common.h
```

```
-rwxr-xr-x 1 yue staff 9356 Aug 30 14:03 cpu
```

```
-rw-r--r--@ 1 yue staff 258 Aug 29 21:48 cpu.c
```

```
-rwxr-xr-x 1 yue staff 9348 Sep 6 12:12 cpu_bound
```

```
-rw-r--r-- 1 yue staff 245 Sep 5 13:10 cpu_bound.c
```

```
...
```

# File System Interfaces

# Creating Files

- UNIX system call: open()

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

-OR-

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

# File Descriptor (`fd`)

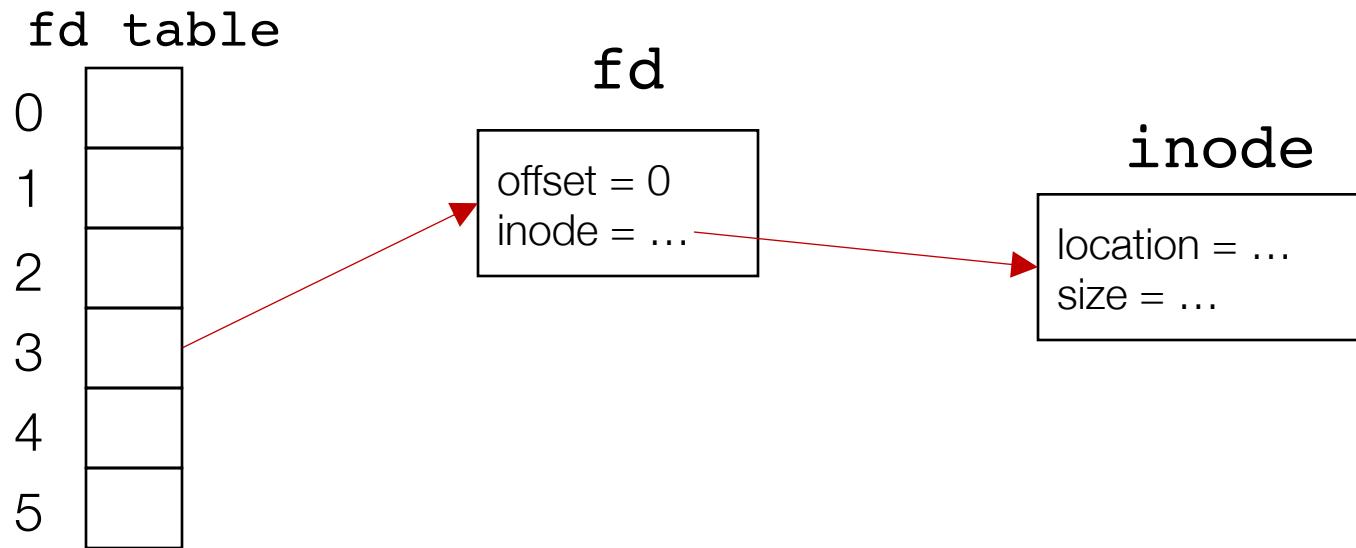
- `open()` returns a file descriptor (`fd`)
  - A `fd` is an integer
  - Private per process
- An **opaque handle** that gives caller the power to perform certain operations
- Think of a `fd` as **a pointer to an object** of the file
  - By owning such an object, you can call other “methods” to access the file

# open() Example

```
int fd1 = open("file.txt", O_CREAT); // return 3
read(fd1, buf, 8);
int fd2 = open("file.txt", O_WRONLY); // return 4
int fd3 = dup(fd2); // return 5
```

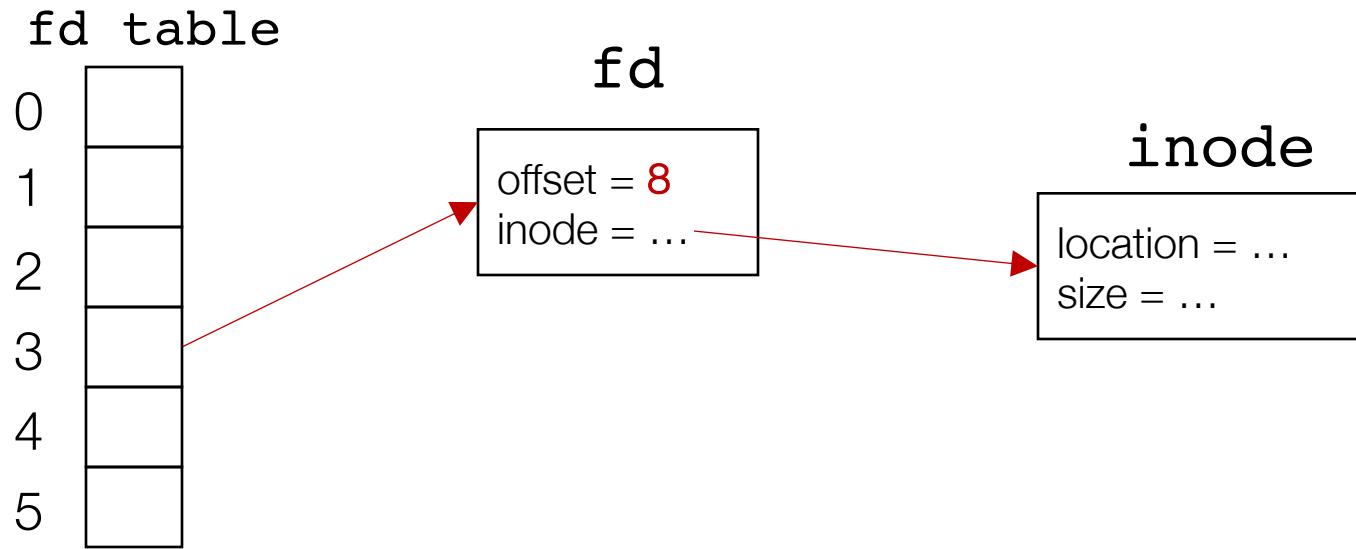
# open() Example

```
int fd1 = open("file.txt", O_CREAT); // return 3
```



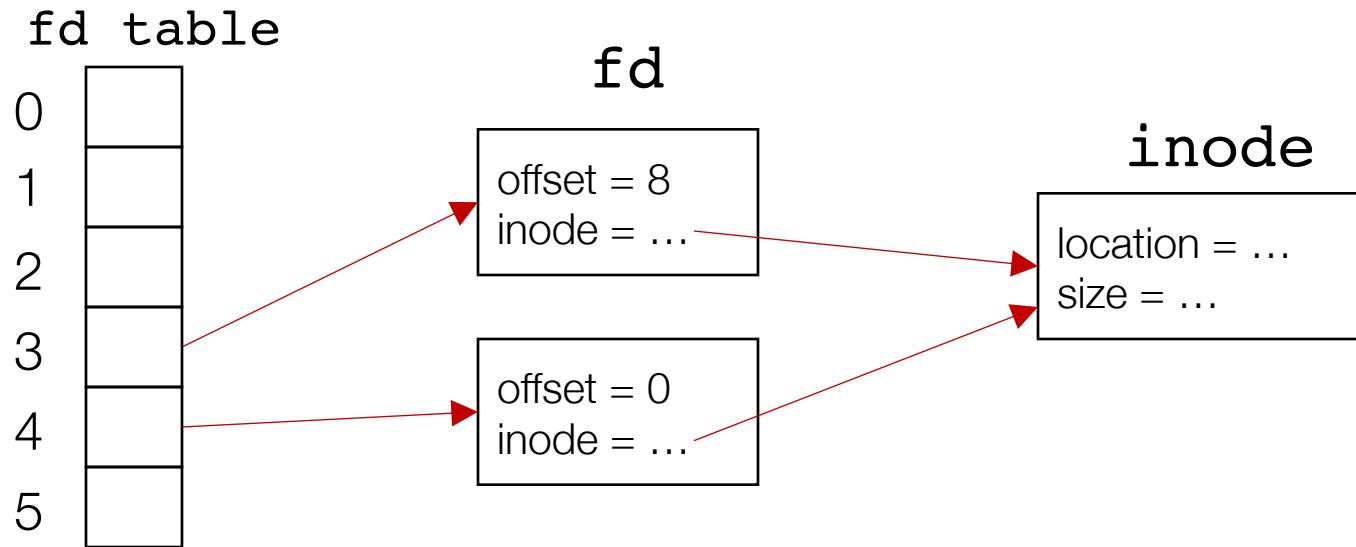
# open() Example

```
int fd1 = open("file.txt", O_CREAT); // return 3  
read(fd1, buf, 8);
```



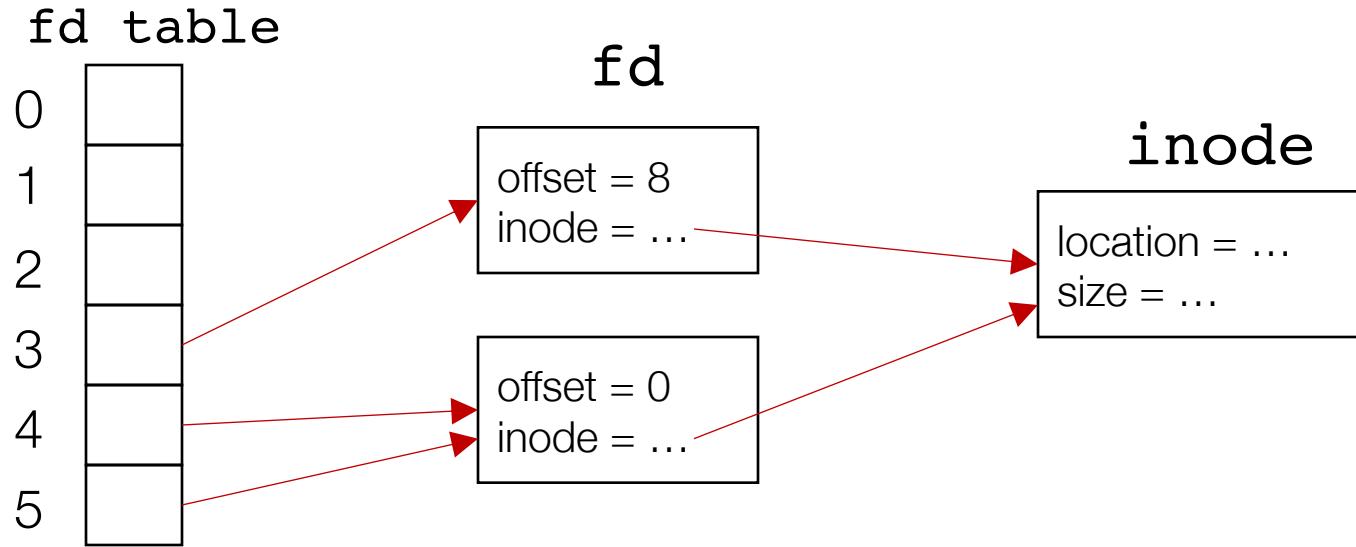
# open() Example

```
int fd1 = open("file.txt", O_CREAT); // return 3  
read(fd1, buf, 8);  
  
int fd2 = open("file.txt", O_WRONLY); // return 4
```



# open() Example

```
int fd1 = open("file.txt", O_CREAT); // return 3  
read(fd1, buf, 8);  
  
int fd2 = open("file.txt", O_WRONLY); // return 4  
int fd3 = dup(fd2); // return 5
```



# UNIX File Read and Write APIs

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

-OR-

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

```
ssize_t sz = read(int fd, void *buf, size_t count);
```

```
ssize_t sz = write(int fd, void *buf, size_t count);
```

```
int ret = close(int fd);
```

# Reading and Writing Files

```
prompt> echo hello > file.txt
```

```
prompt> cat file.txt
```

```
hello
```

```
prompt>
```

# Reading and Writing Files

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)          = 3
read(3, "hello\n", 65536)          = 6
write(1, "hello\n", 6)              = 6
read(3, "", 65536)                = 0
close(3)                           = 0
...
prompt>
```

# Reading and Writing Files

Open the file with read  
only mode

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)      = 3
read(3, "hello\n", 65536)       = 6
write(1, "hello\n", 6)          = 6
read(3, "", 65536)             = 0
close(3)                        = 0
...
prompt>
```

# Reading and Writing Files

Open the file with read  
only mode

```
prompt> strace cat file.txt
```

• • •

# Read content from file

```
open("file.txt", O_RDONLY) = 3
read(3, "hello\n", 65536) = 6
write(1, "hello\n", 6) = 6
read(3, "", 65536) = 0
close(3) = 0
```

• • •

prompt>

# Reading and Writing Files

Open the file with read  
only mode

```
prompt> strace cat file.txt
```

...

open("file.txt", O_RDONLY)	= 3
read(3, "hello\n", 65536)	= 6
write(1, "hello\n", 6)	= 6
read(3, "", 65536)	= 0
close(3)	= 0
...	

prompt>

# Reading and Writing Files

Open the file with read  
only mode

```
prompt> strace cat file.txt
```

• • •

## Read content from file

```
open("file.txt", O_RDONLY) = 3
```

```
read(3, "hello\n", 65536) = 6
```

```
write(1, "hello\n", 6) = 6
```

```
read(3, "", 65536) = 0
```

`close(3)` = 0

• • •

prompt>

## Write string to std output fd 1

cat tries to read more  
but reaches EOF

# Reading and Writing Files

Open the file with read  
only mode

```
prompt> strace cat file.txt
```

Read content from file

Write string to std  
output fd 1

open("file.txt", O_RDONLY)	= 3
read(3, "hello\n", 65536)	= 6
write(1, "hello\n", 6)	= 6
read(3, "", 65536)	= 0
close(3)	= 0

cat tries to read more  
but reaches EOF

cat done with file ops  
and closes the file

```
prompt>
```

# Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

# Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

**whence**:

- If whence is SEEK\_SET, the offset is set to offset bytes
- If whence is SEEK\_CUR, the offset is set to its current location plus offset bytes
- If whence is SEEK\_END, the offset is set to the size of the file plus offset bytes

# Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

**whence**:

- If whence is SEEK\_SET, the offset is set to offset bytes
- If whence is SEEK\_CUR, the offset is set to its current location plus offset bytes
- If whence is SEEK\_END, the offset is set to the size of the file plus offset bytes

**Note:** Calling **lseek()** does not perform a disk seek!

# Writing Immediately with `fsync()`

```
int fd = fsync(int fd);
```

- `fsync(fd)` forces buffers to flush to disk, and (usually) tells the disk to flush its write cache too
  - To make the data **durable** and **persistent**
- **Write buffering** improves performance

# Renaming Files

```
prompt> mv file.txt new_name.txt
```

# Renaming Files

```
prompt> strace mv file.txt new_name.txt  
...  
rename("file.txt", "new_name.txt") = 0  
...  
prompt>
```

# Renaming Files

System call `rename()`  
atomically renames a  
file

```
prompt> strace mv file.txt new_name.txt
...
rename("file.txt", "new_name.txt") = 0
...
prompt>
```

# File Renaming Example

```
prompt> vim file.txt
```



```
int fd = open(".file.txt.swp", O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
```

Using `vim` to edit a file and then save it

# File Renaming Example

```
prompt> vim file.txt  
... vim editing session ...
```



```
int fd = open(".file.txt.swp", O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);  
write(fd, buffer, size); // write out new version of file (editing...)
```

Using `vim` to edit a file and then save it

# File Renaming Example

```
prompt> vim file.txt  
... vim editing session ...  
prompt> ..... :WQ
```



```
int fd = open(".file.txt.swp", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);  
write(fd, buffer, size); // write out new version of file  
fsync(fd); // make data durable  
close(fd); // close tmp file  
rename(".file.txt.swp", "file.txt"); // change name and replacing old file
```

Using vim to edit a file and then save it

# Deleting Files

```
prompt> rm file.txt
```

# Deleting Files

```
prompt> strace rm file.txt
...
unlink("file.txt") = 0
...
prompt>
```

# Deleting Files

System call `unlink()` is called to delete a file

```
prompt> strace rm file.txt
...
unlink("file.txt") = 0
...
prompt>
```

# Deleting Files

System call `unlink()` is called to delete a file

```
prompt> strace rm file.txt
...
unlink("file.txt") = 0
...
prompt>
```

Directories are deleted when `unlink()` is called

Q: File descriptors are deleted when ???

# Demo: Hard Links vs. Symbolic Links

# File System Implementation

# File System Implementation

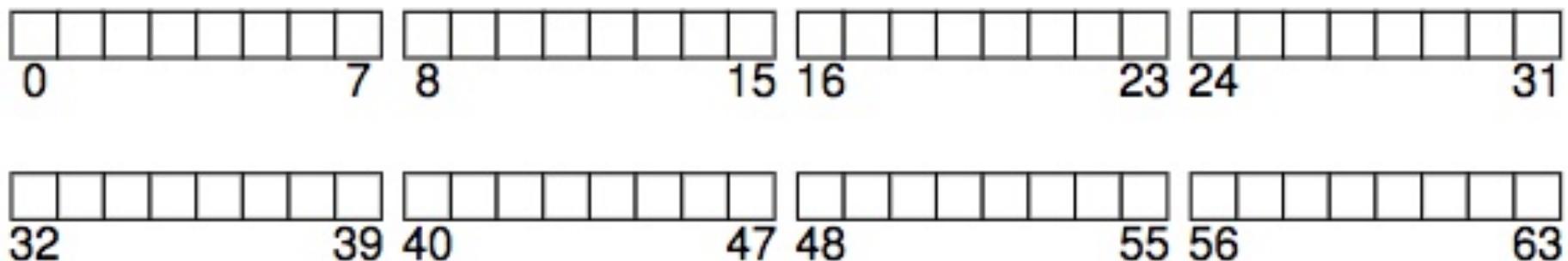
- On-disk structures
  - How do we represent files and directories?
- File system operations (internally)
  - How on-disk structures get touched when performing FS operations
- File system locality & data layout policies
  - How data layout impacts locality for on-disk FS?
- File system caching
  - How to avoid excessive I/O operations?

# On-Disk Structures

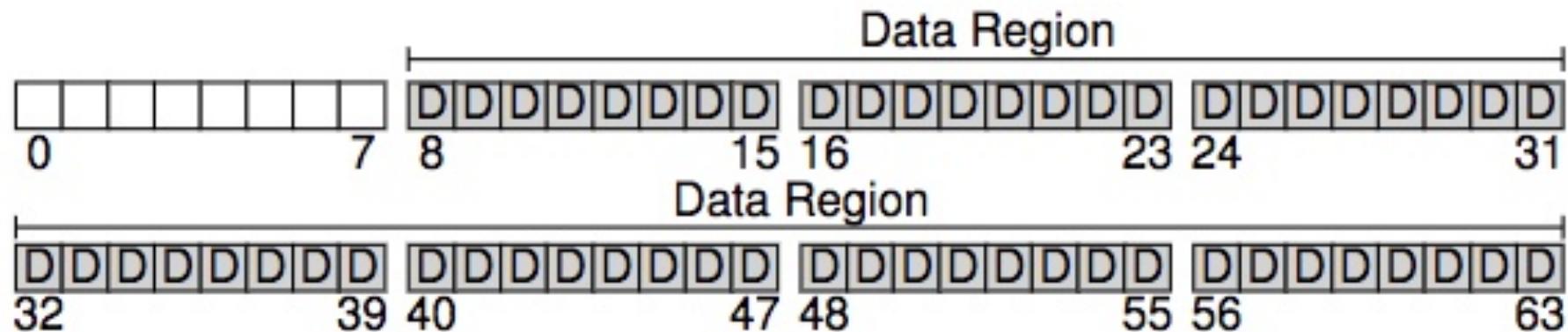
# On-Disk Structures

- Common file system structures
  - Data block
  - inode table
  - Directories
  - Data bitmap
  - inode bitmap
  - Superblock

# On-Disk Structure: Empty Disk



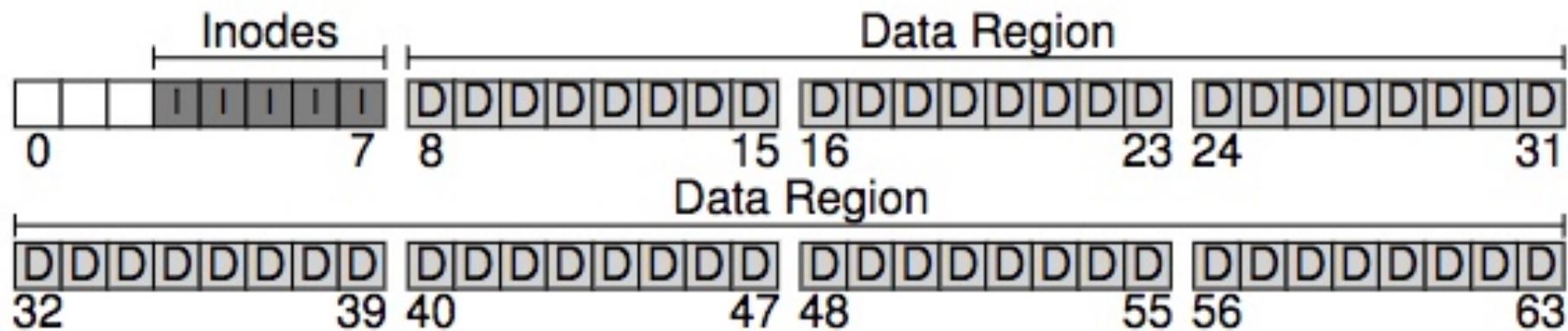
# On-Disk Structure: Data Blocks



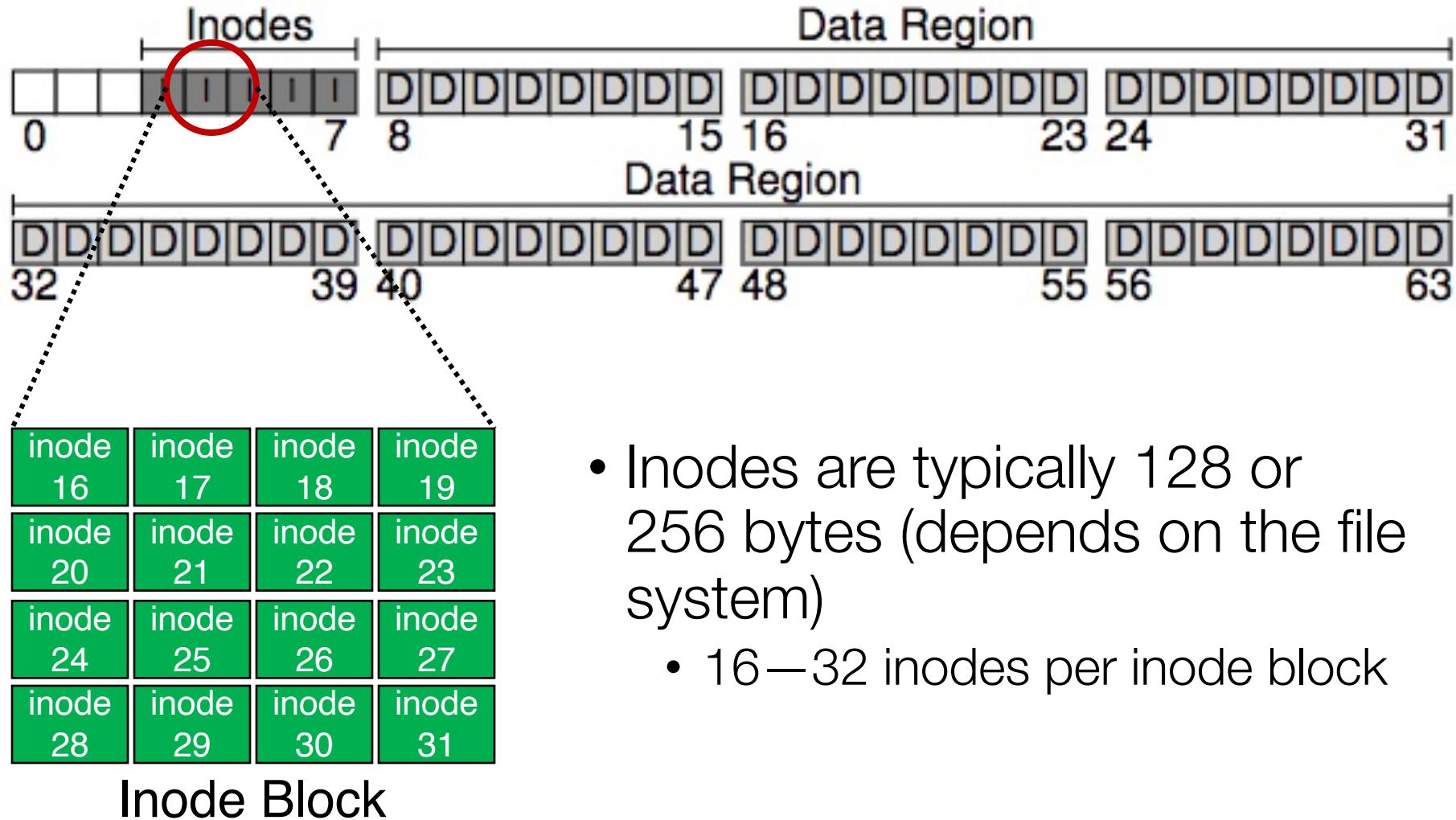
# On-Disk Structures

- Common file system structures
  - Data block
  - **inode table**
  - Directories
  - Data bitmap
  - inode bitmap
  - Superblock

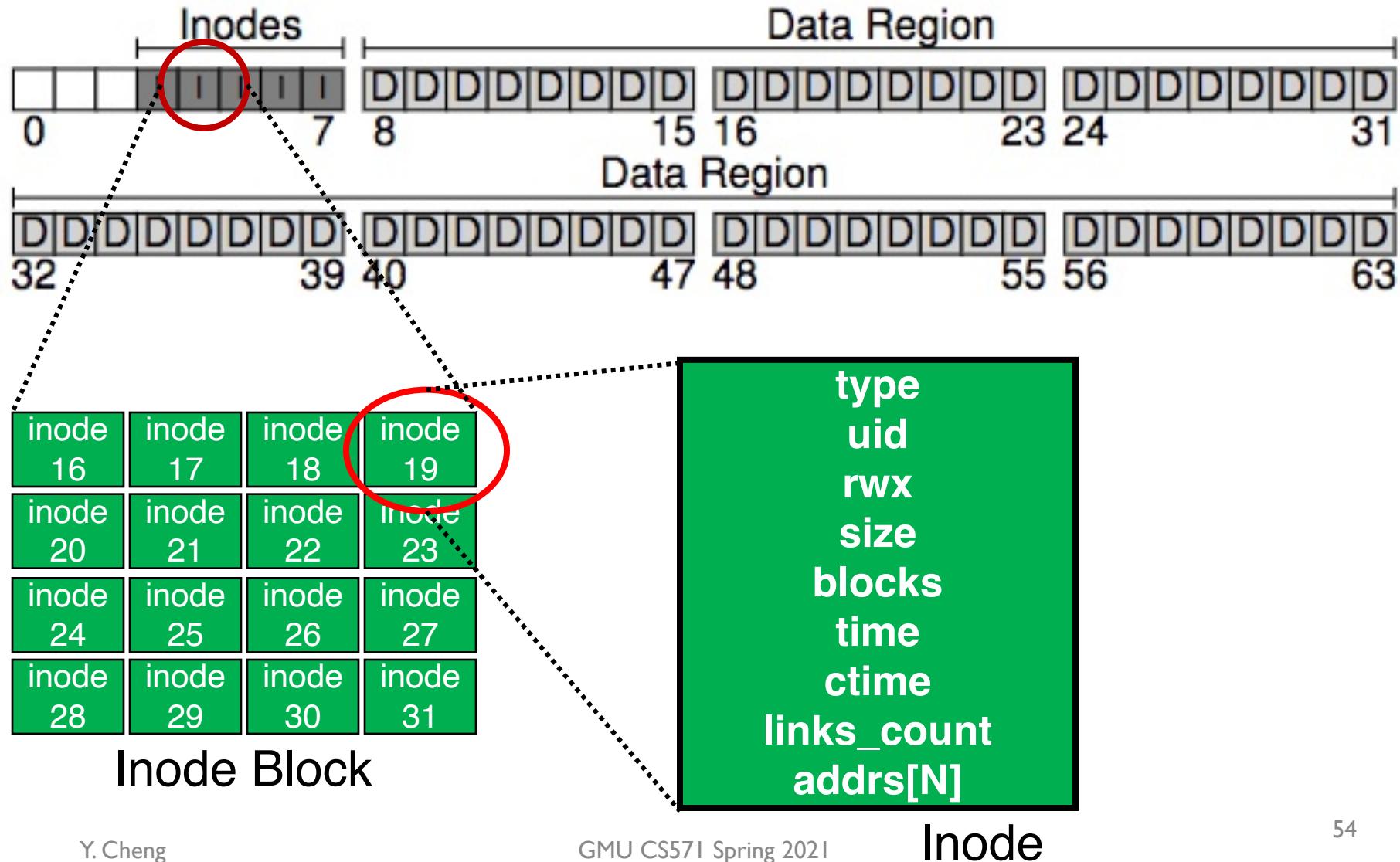
# On-Disk Structure: Inodes



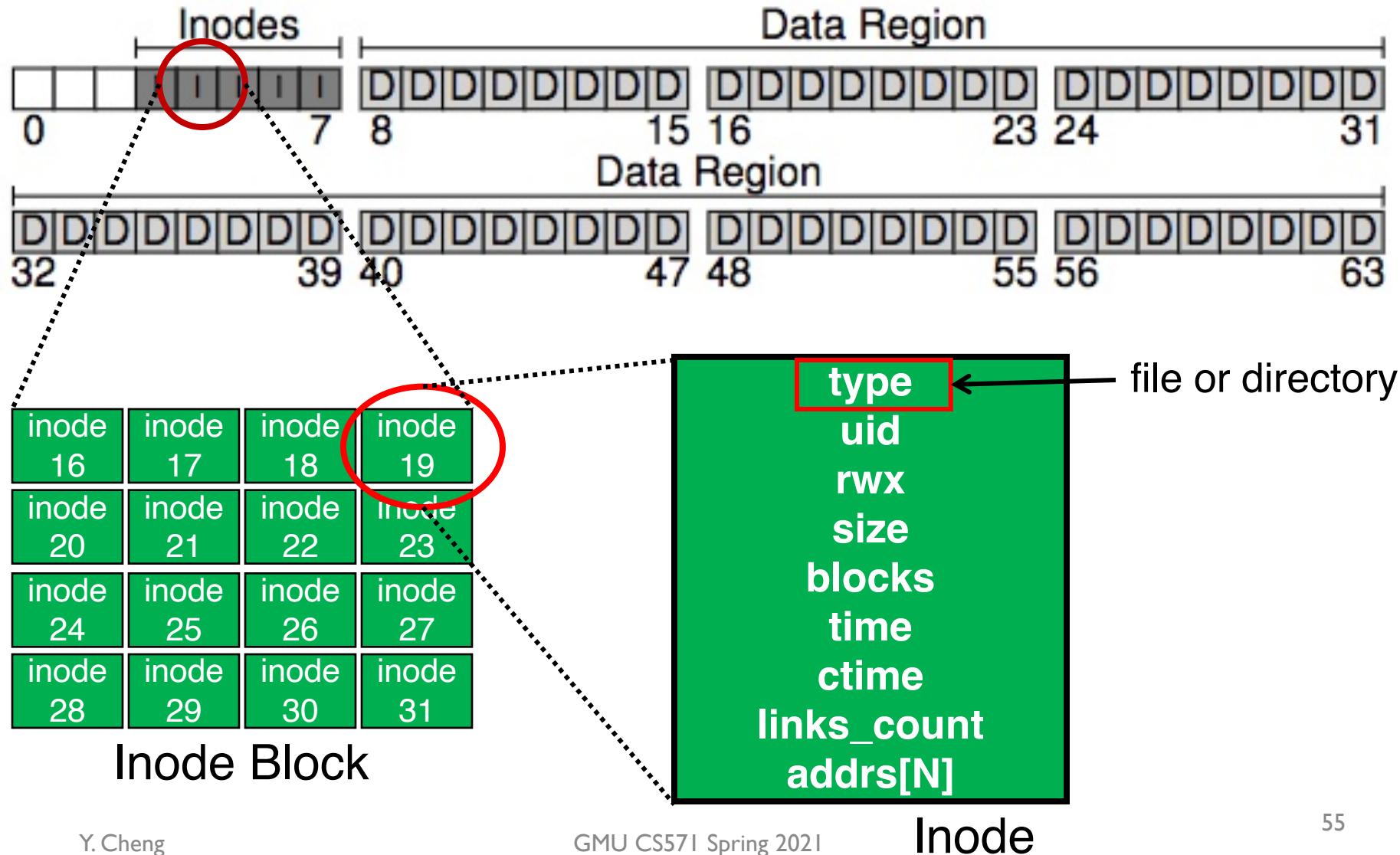
# On-Disk Structure: Inodes



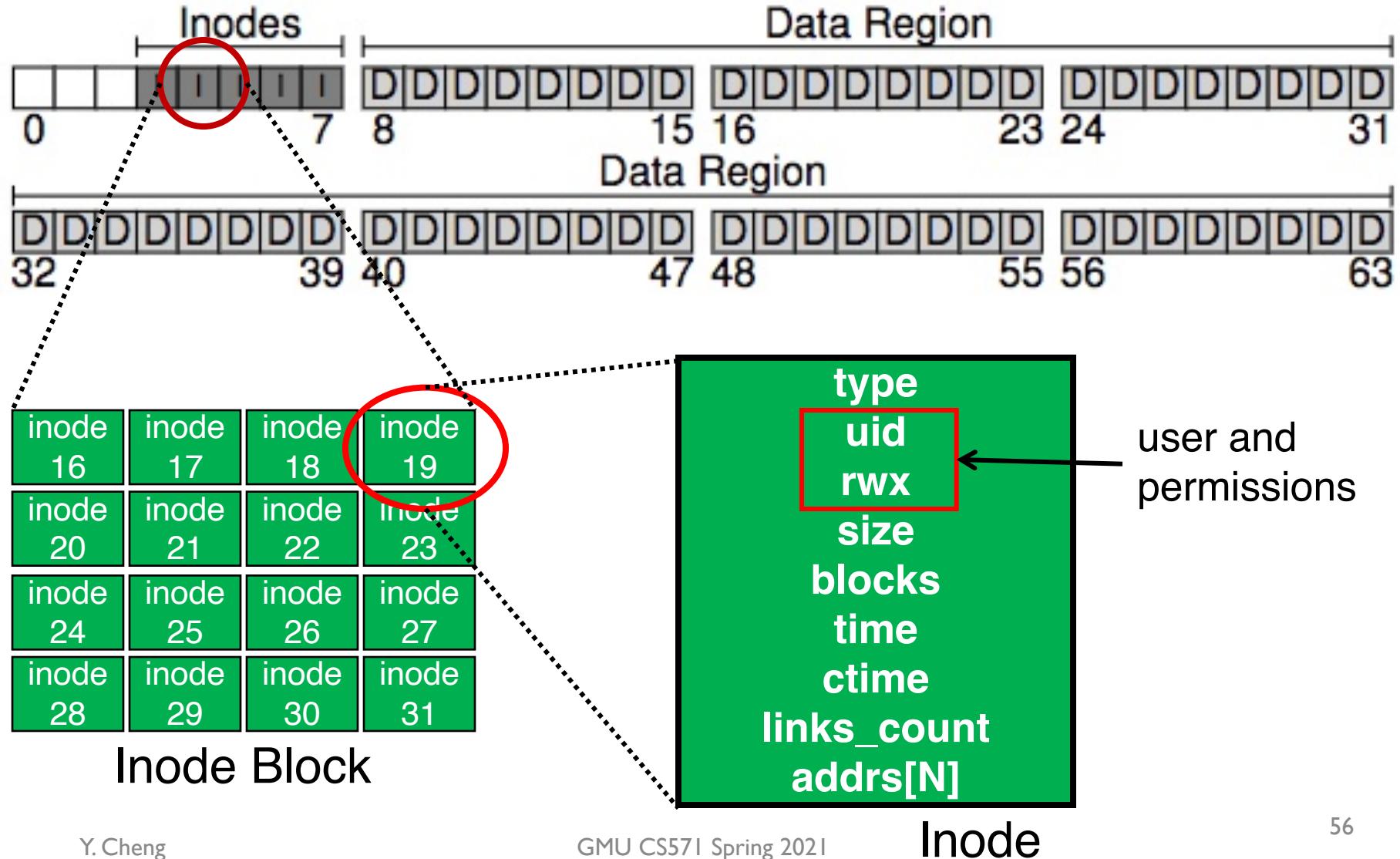
# On-Disk Structure: Inodes



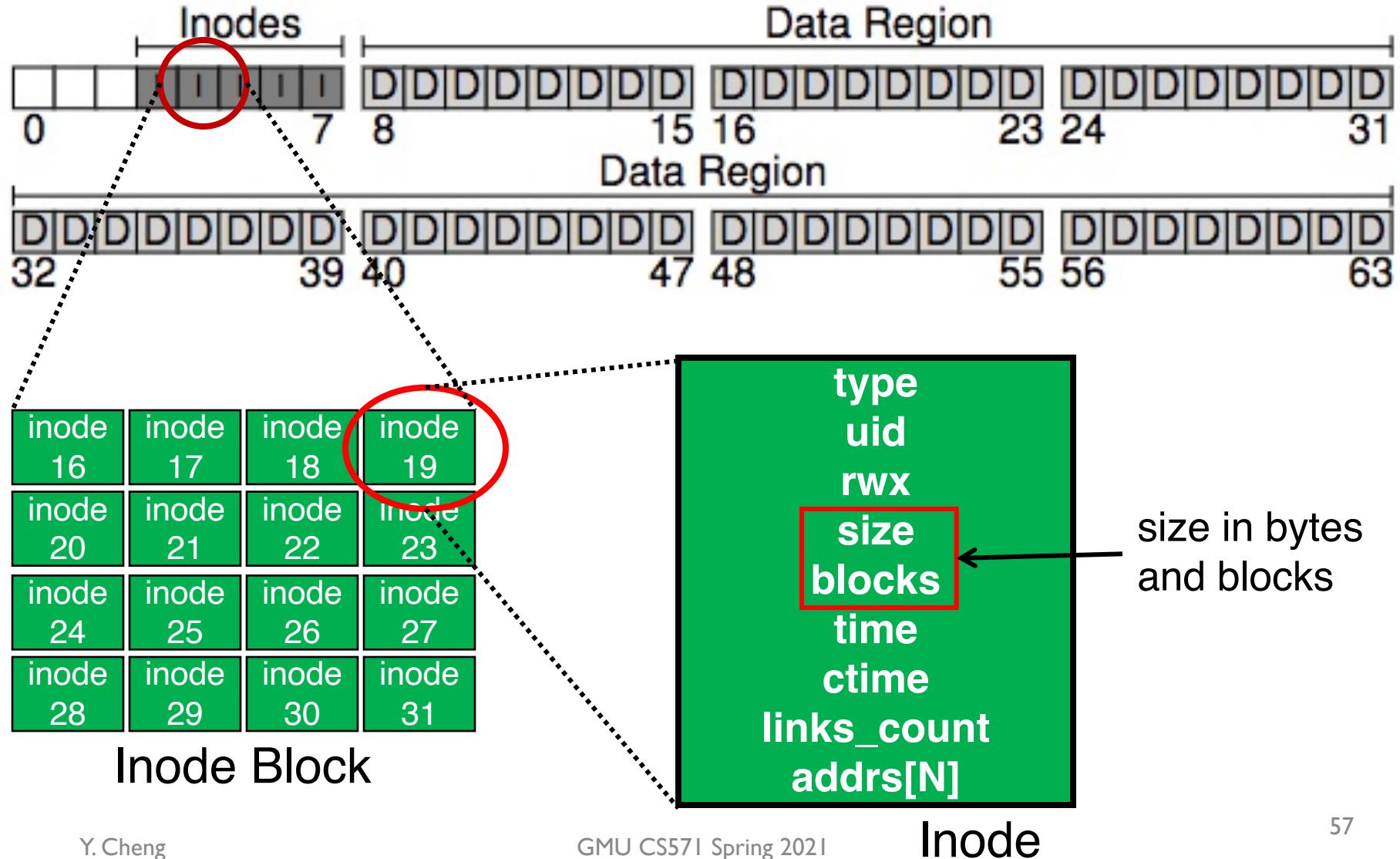
# On-Disk Structure: Inodes



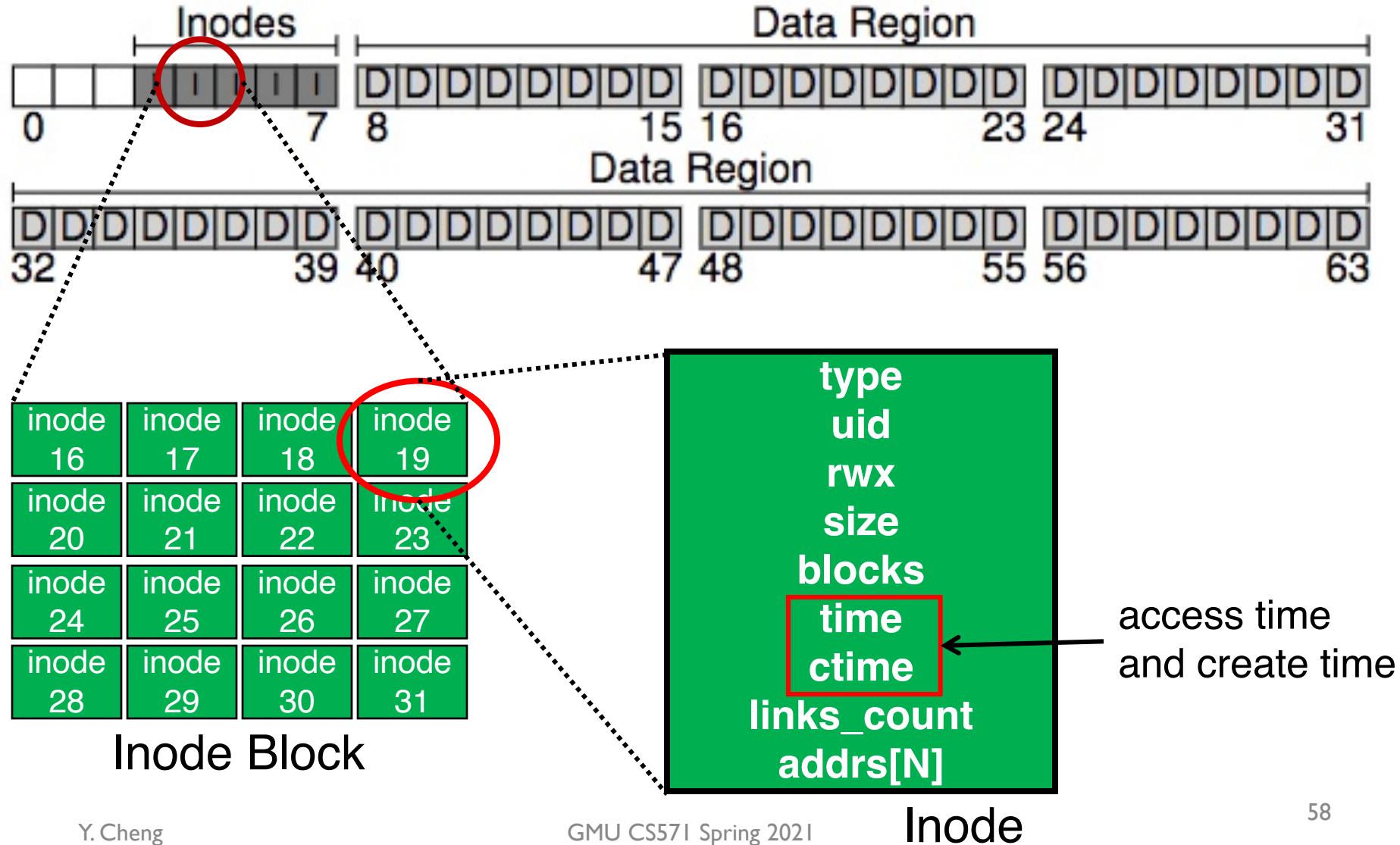
# On-Disk Structure: Inodes



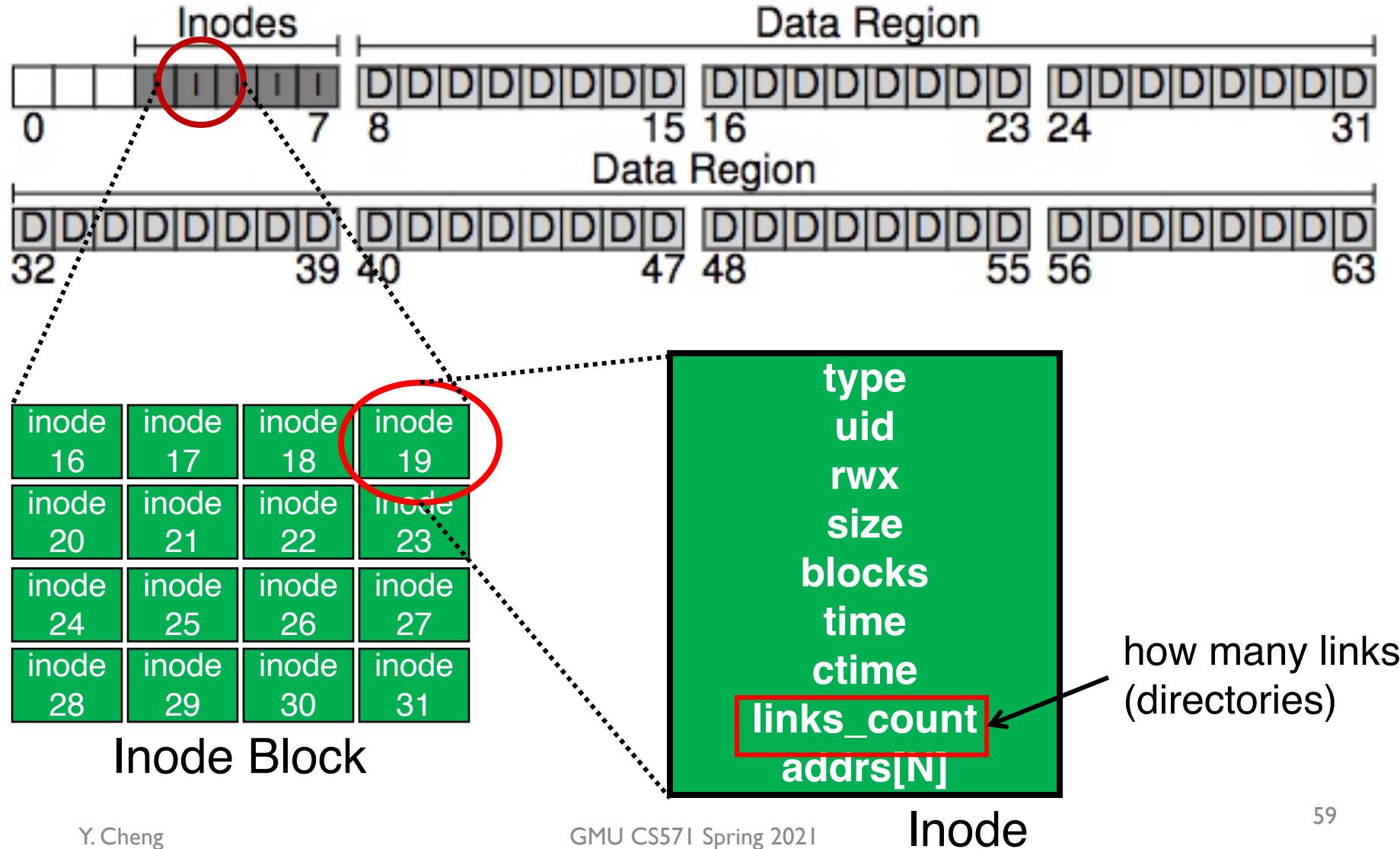
# On-Disk Structure: Inodes



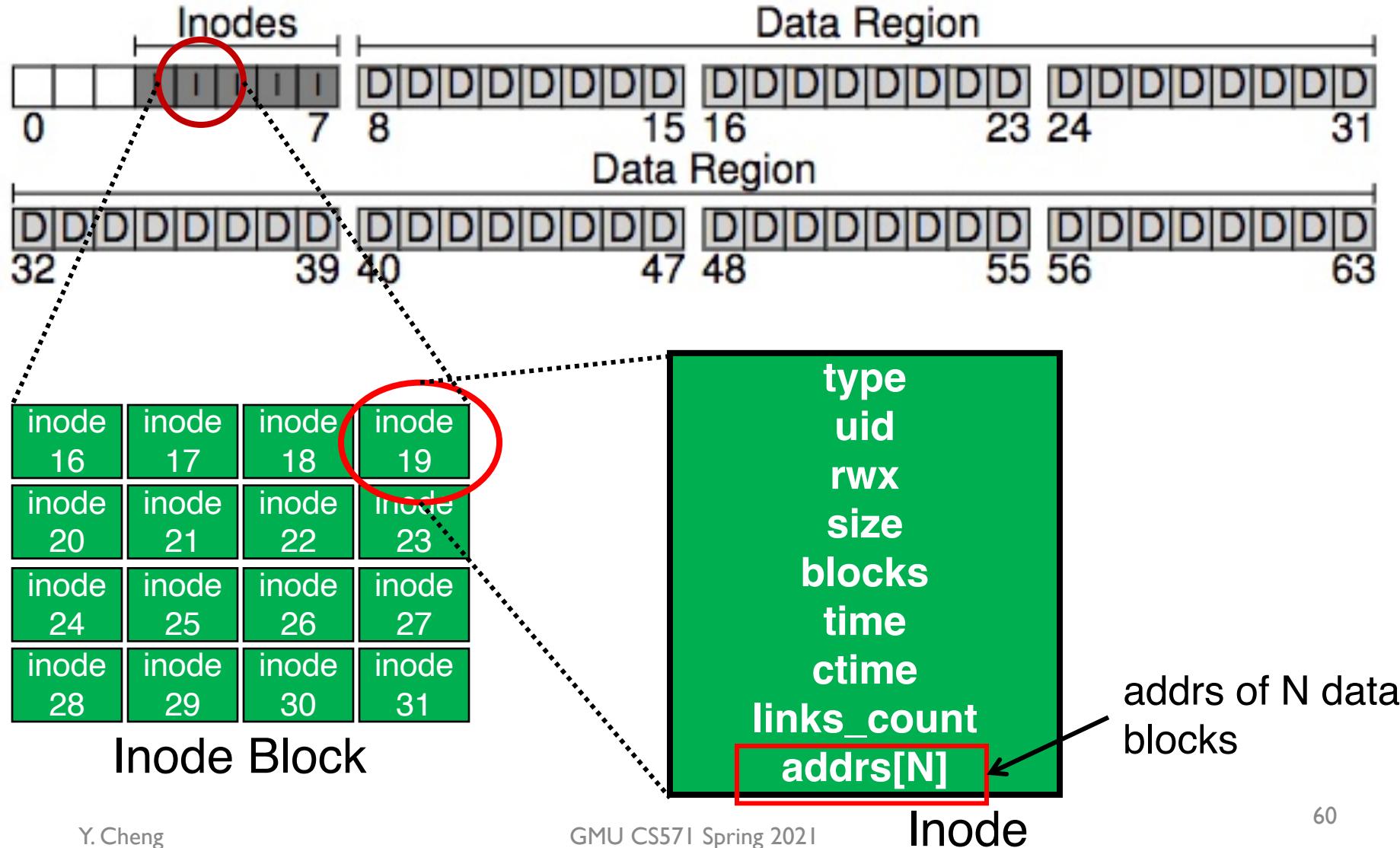
# On-Disk Structure: Inodes



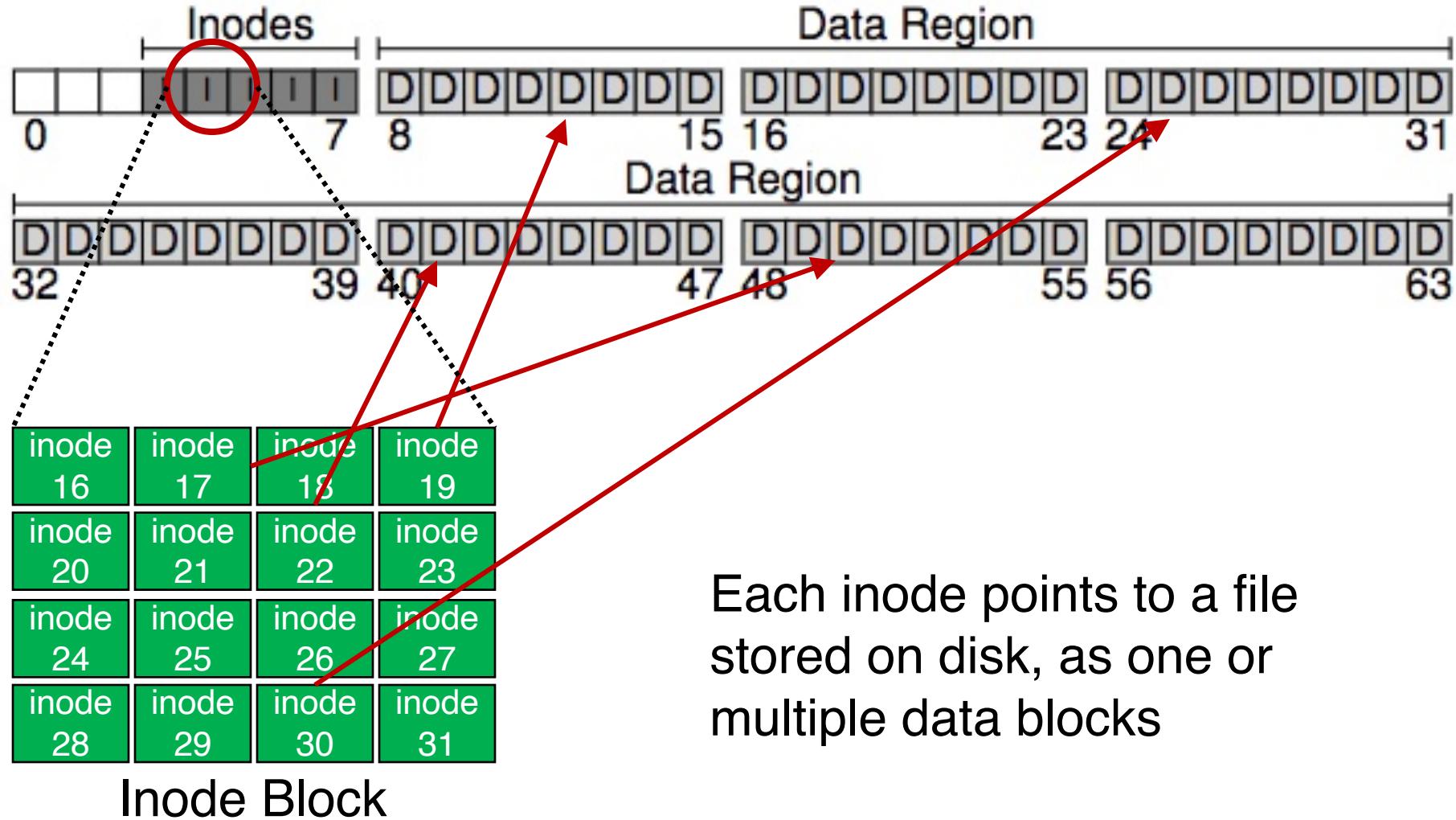
# On-Disk Structure: Inodes



# On-Disk Structure: Inodes



# On-Disk Structure: Inodes



# On-Disk Structures

- Common file system structures
  - Data block
  - Inode table
  - Directories
  - Data bitmap
  - Inode bitmap
  - Superblock

# On-Disk Structure: Directories

- Common directory design: just store directory entries in files
  - Different file systems vary
- Various data structures (formats) could be used
  - Lists
  - B-trees

# On-Disk Structures

- Common file system structures
  - Data block
  - inode table
  - Directories
  - Data bitmap
  - inode bitmap
  - Superblock

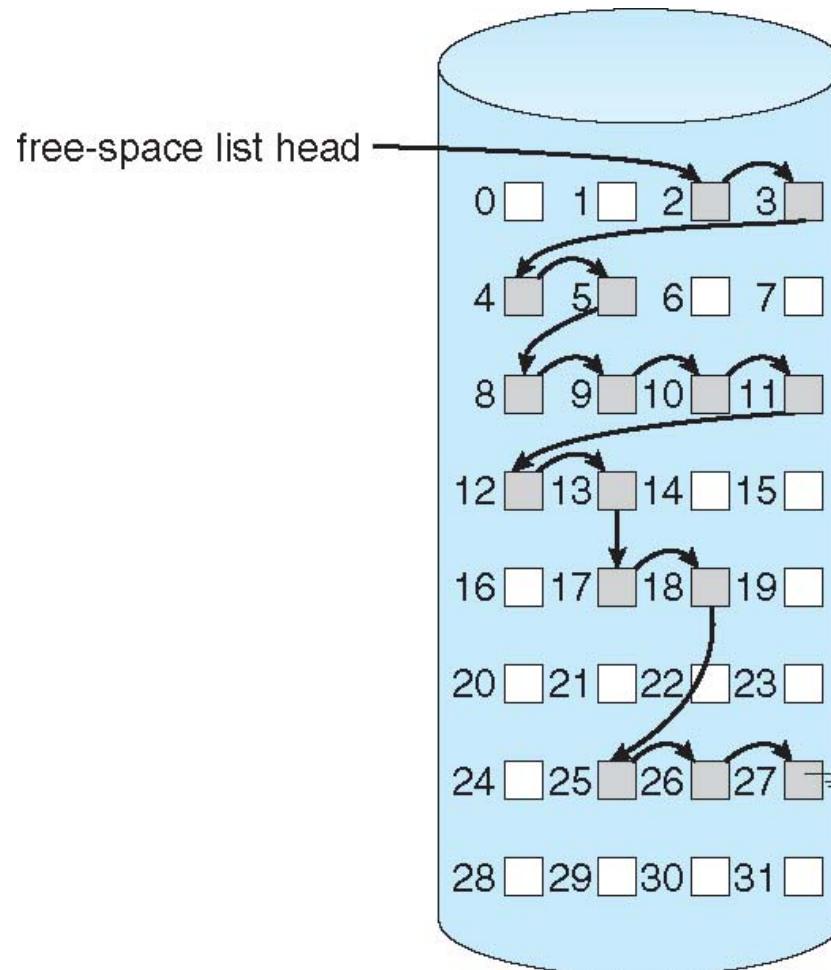
# Allocation

- How does file system find free data blocks or free inodes?

# Allocation

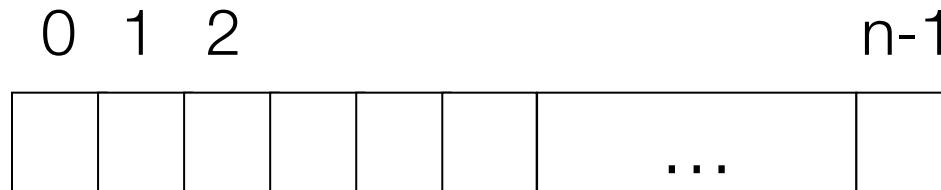
- How does file system find free data blocks or free inodes?
  - Free list
  - Bitmaps
- What are the tradeoffs?

# Free List



# Bitmap

Each bit of the bitmap is used to indicate whether the corresponding object/block is **free** (0) or **in-use** (1)

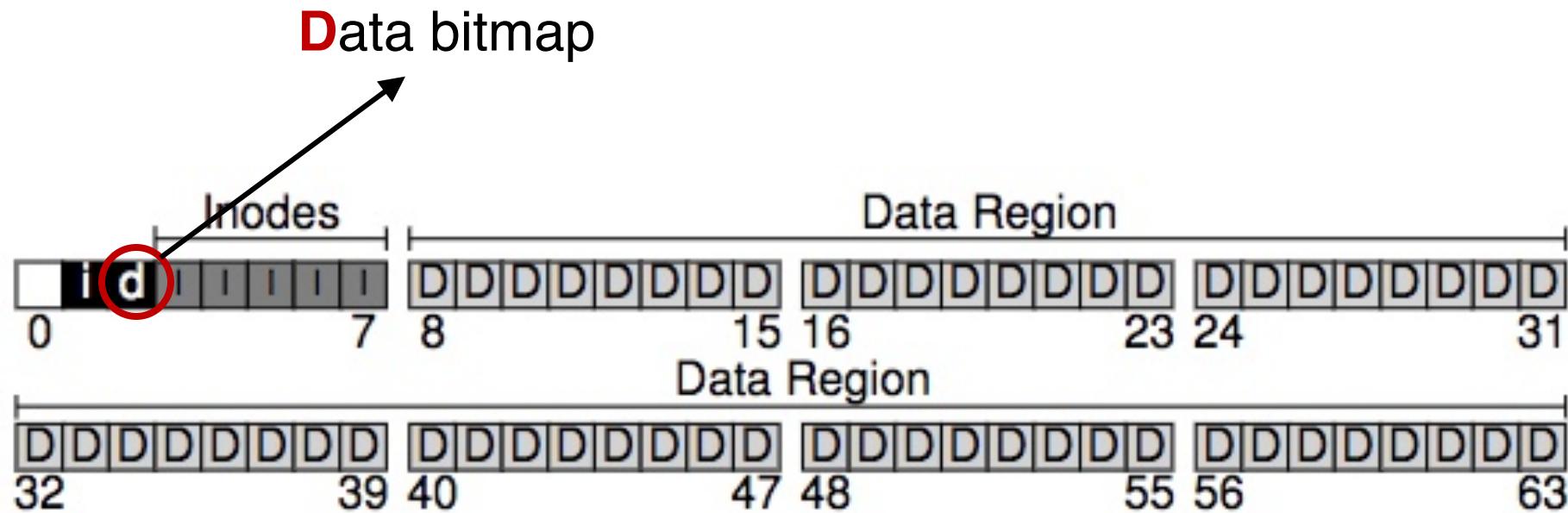


$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{object}[i] \text{ in use} \\ 0 \Rightarrow \text{object}[i] \text{ free} \end{cases}$$

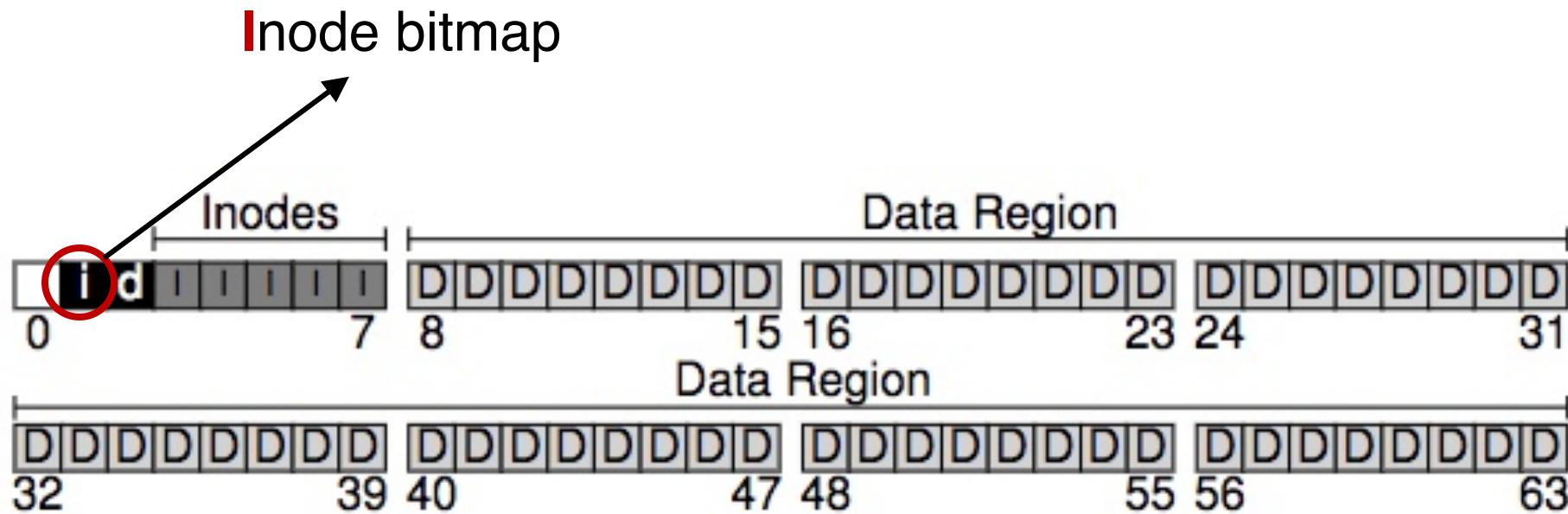
# Allocation

- How does file system find free data blocks or free inodes?
  - Free list
  - Bitmaps
- What are the tradeoffs?
  - Free list: Cannot get contiguous space easily
  - Bitmap: Easy to allocate contiguous space for files

# On-Disk Structure: Data Bitmaps



# On-Disk Structure: Inode Bitmaps



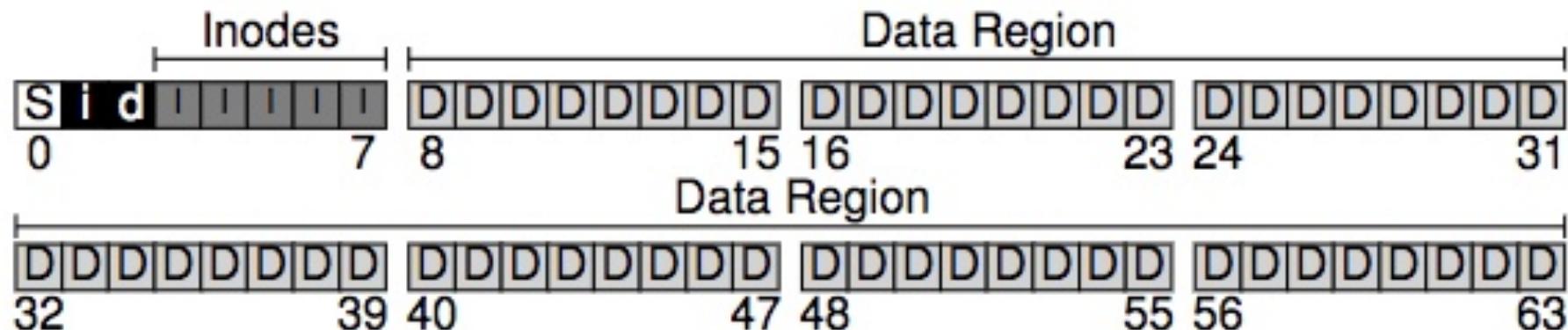
# On-Disk Structures

- Common file system structures
  - Data block
  - Inode table
  - Directories
  - Data bitmap
  - Inode bitmap
  - **Superblock**

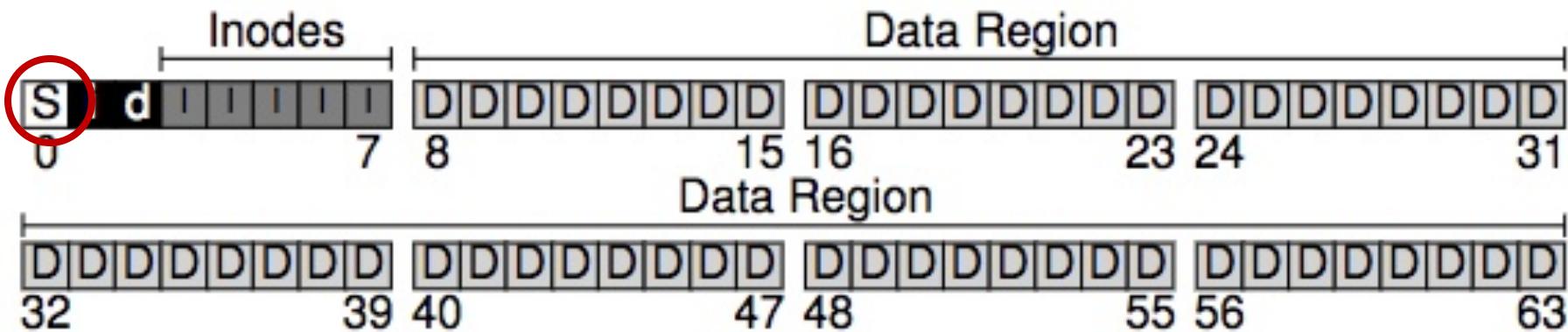
# On-Disk Structure: Superblock

- Need to know basic file system configuration and runtime status, such as:
  - Block size
  - How many inodes are there
  - How much free space
- Store all these **metadata** info in a superblock

# On-Disk Structure: Superblock

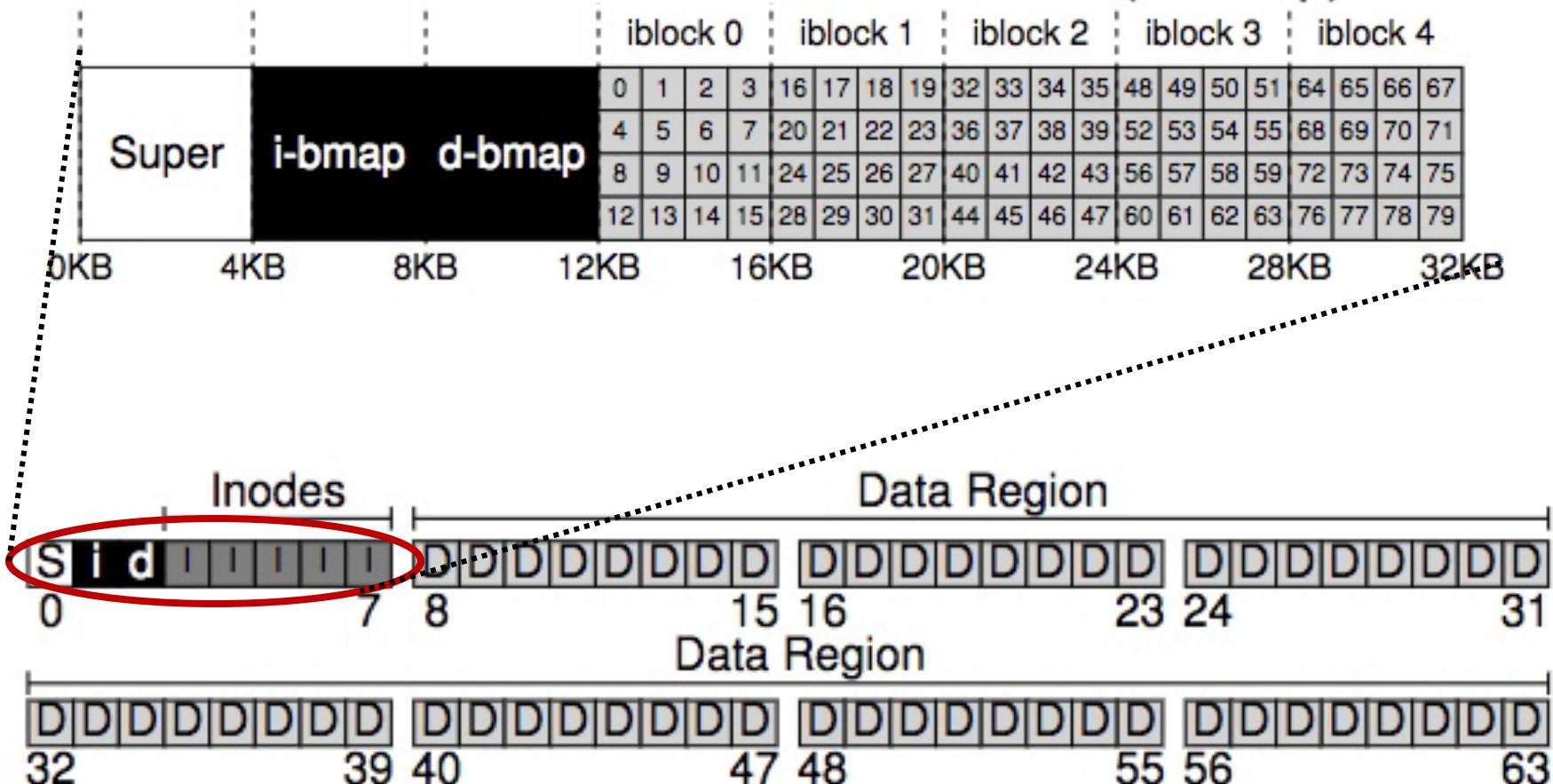


# On-Disk Structure: Superblock



# On-Disk Structure Overview

## The Inode Table (Closeup)



# File System Operations

# Basic File System Operations

create /foo/bar

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

# Basic File System Operations

create /foo/bar

[traverse]

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

# Basic File System Operations

create /foo/bar

[traverse]

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

# Basic File System Operations

create /foo/bar

[traverse]

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

foo inode: we have permission  
foo data: bar doesn't exist

# Basic File System Operations

create /foo/bar

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

# Basic File System Operations

# create /foo/bar

## [allocate inode]

# Basic File System Operations

create /foo/bar

[populate inode]

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

# Basic File System Operations

create /foo/bar

[add bar to /foo]

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

# Basic File System Operations

write to /foo/bar

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

# Basic File System Operations

write to /foo/bar

[block full? yes]

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

# Basic File System Operations

write to /foo/bar

[allocate block]

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

read  
write

# Basic File System Operations

write to /foo/bar

[point to block]



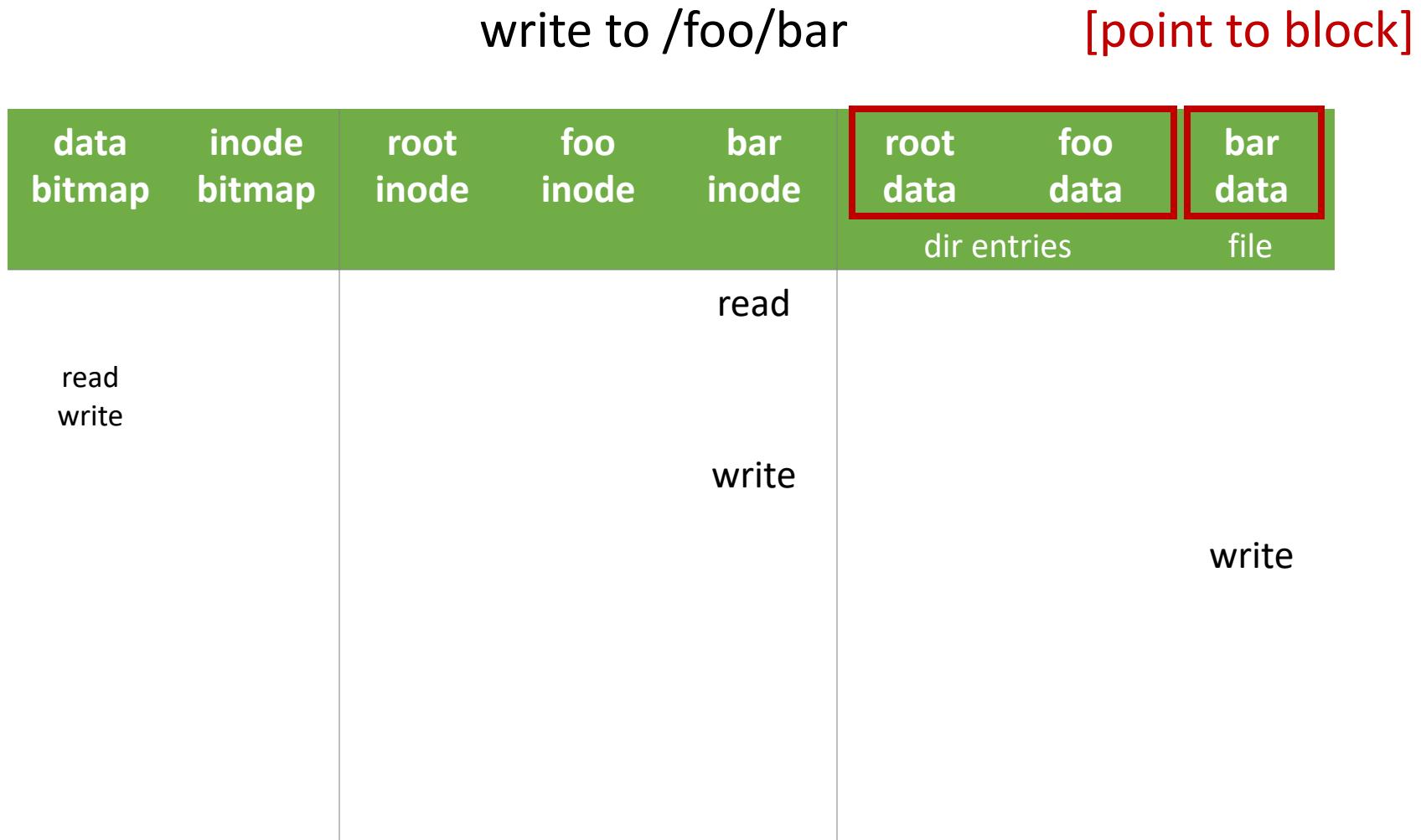
# Basic File System Operations

write to /foo/bar

[point to block]



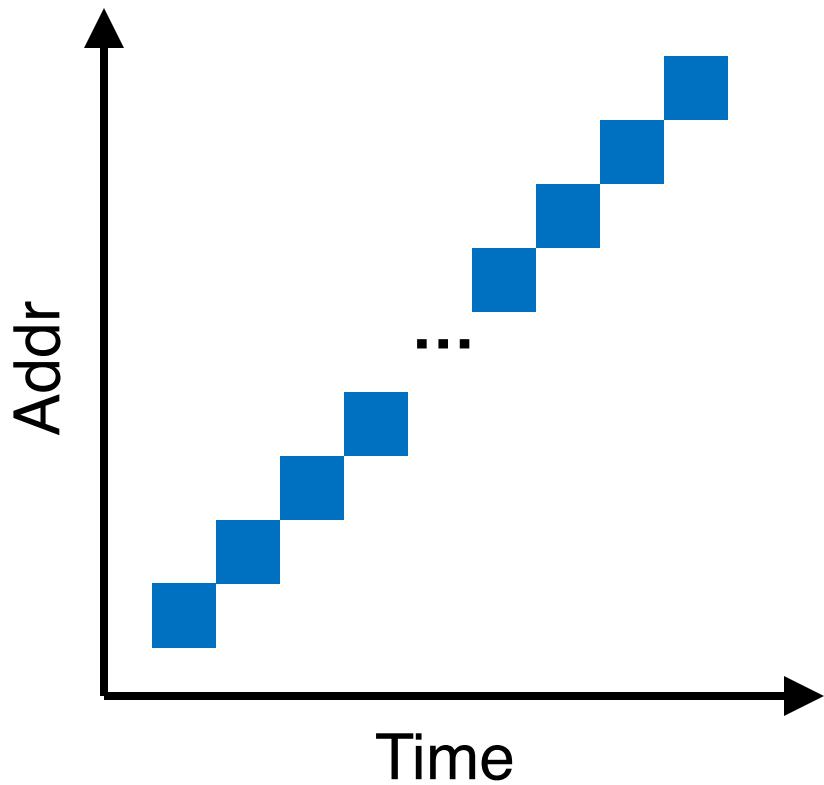
# Basic File System Operations



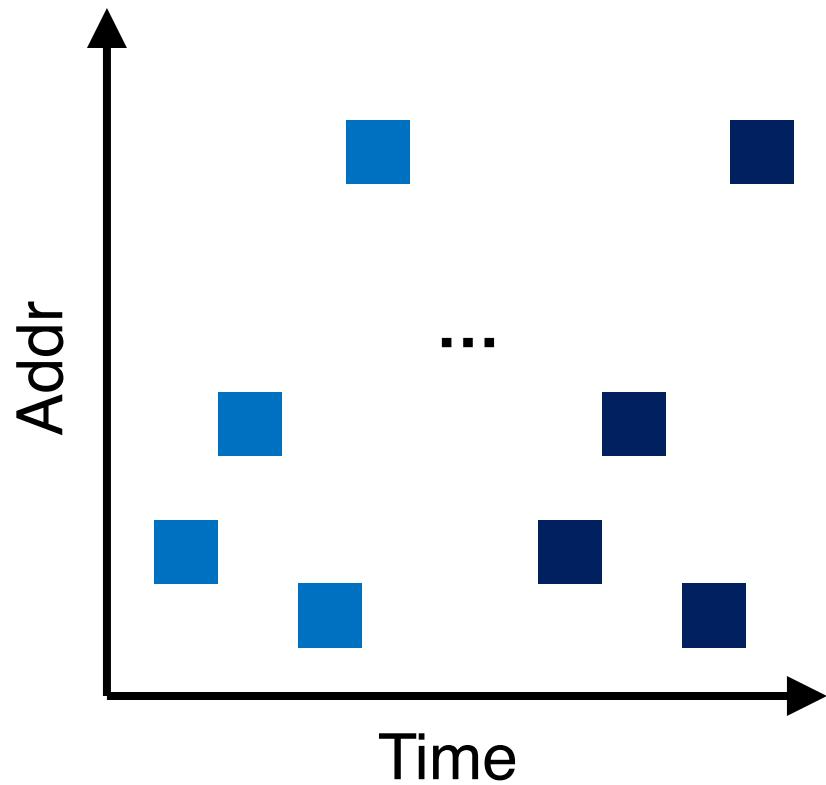
# Locality & Data Layout

# Review: Locality Types

Workload A

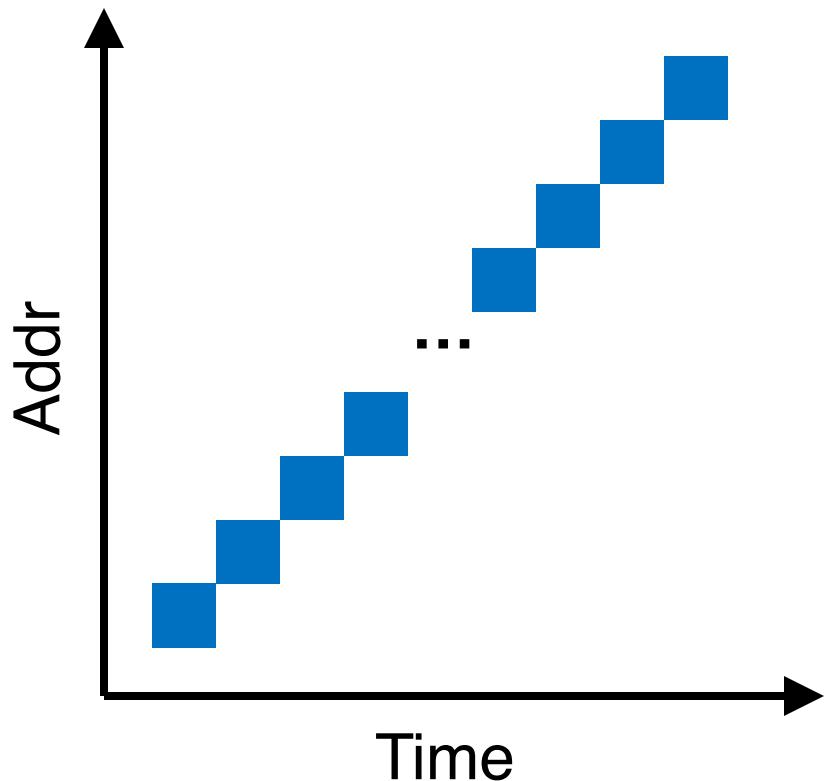


Workload B



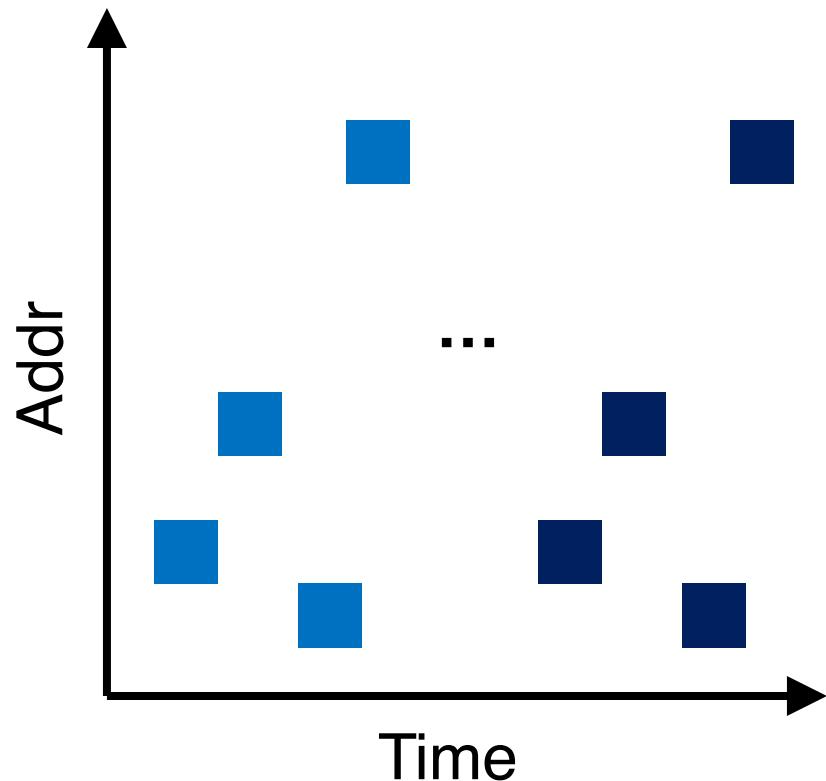
# Review: Locality Types

Workload A



**Spatial Locality**

Workload B



**Temporal Locality**

# Locality Usefulness in the Context of Disk-based File Systems

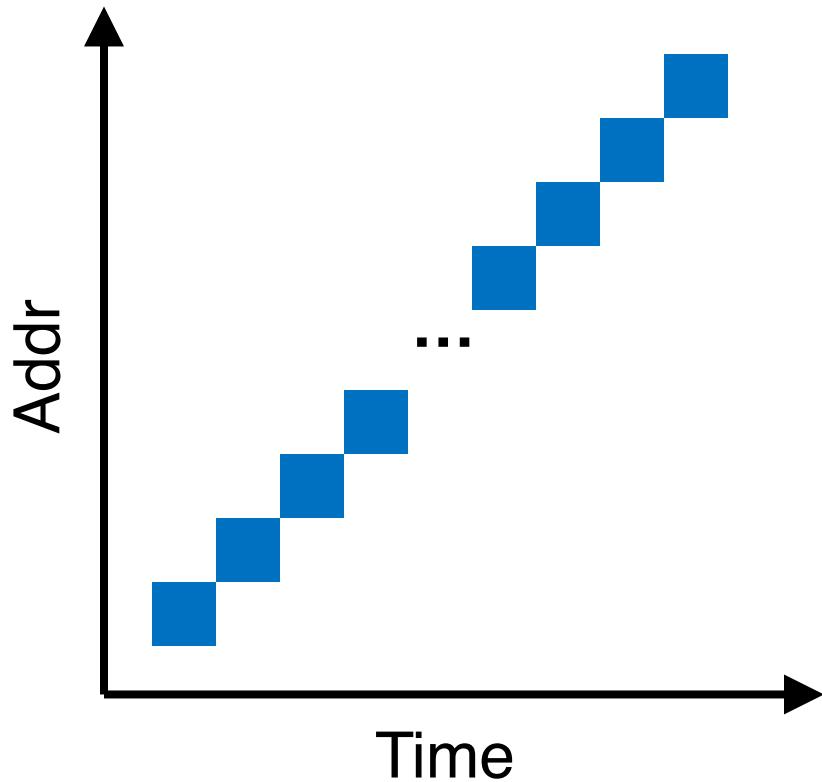
- What types of locality are useful for a *cache*?
- What types of locality are useful for a disk?

# Locality Usefulness in the Context of Disk-based File Systems

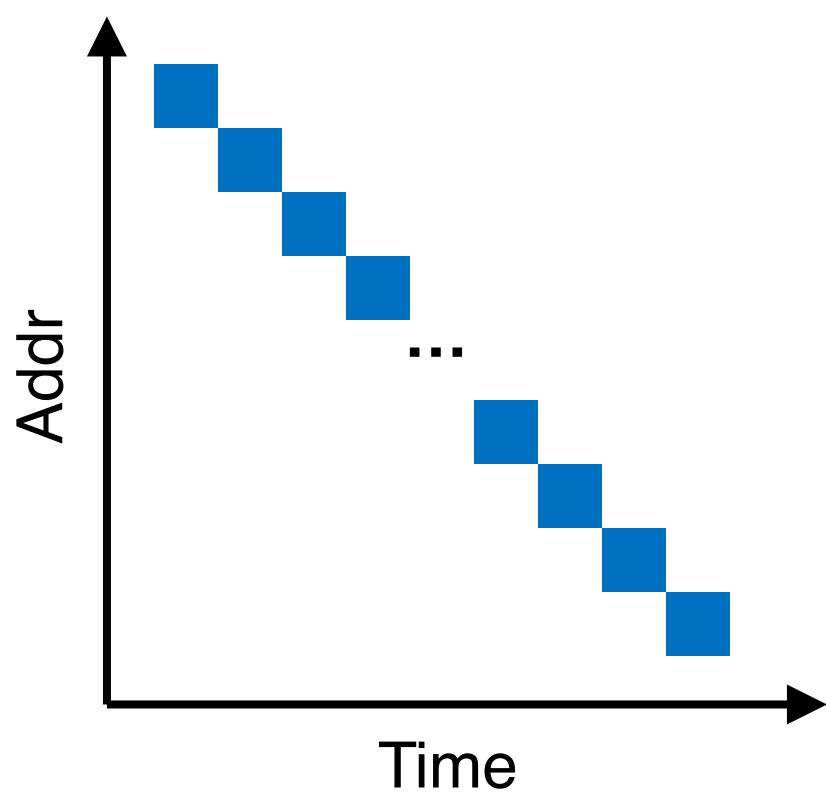
- What types of locality are useful for a **cache**?
  - Possibly, both spatial & temporal locality
- What types of locality are useful for a **disk**?
  - Spatial locality, since a disk sucks in random I/Os but can provide reasonably good sequential performance

# Order Matters Now for FS on Disk

Workload A

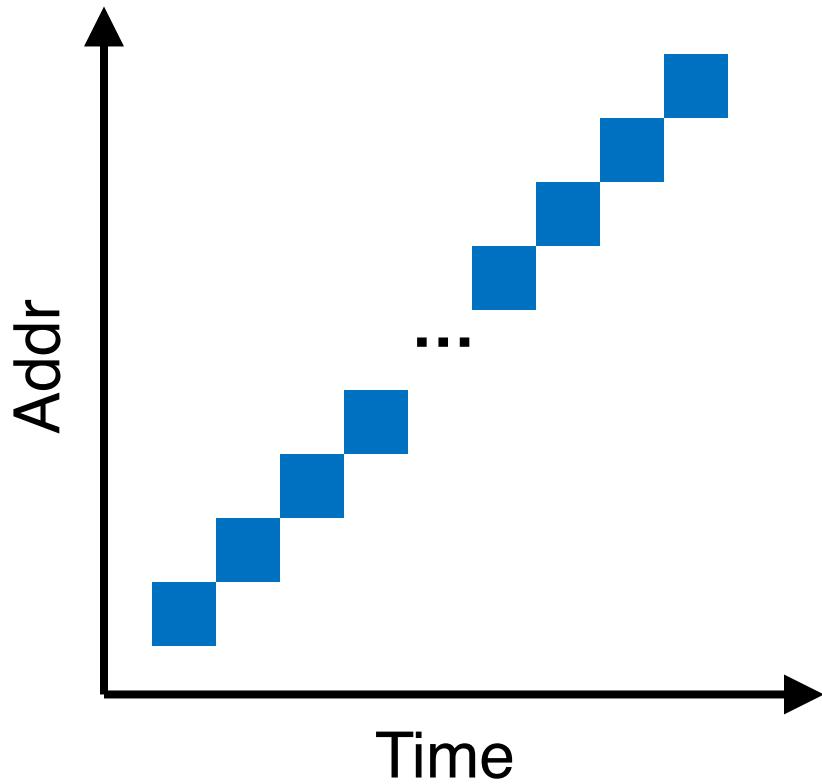


Workload B



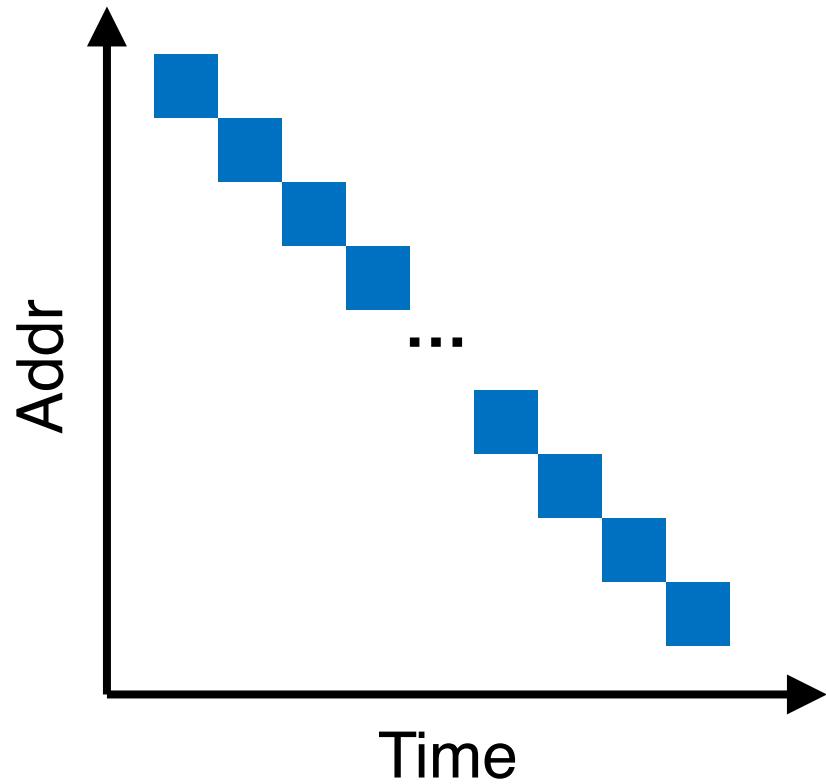
# Order Matters Now for FS on Disk

Workload A



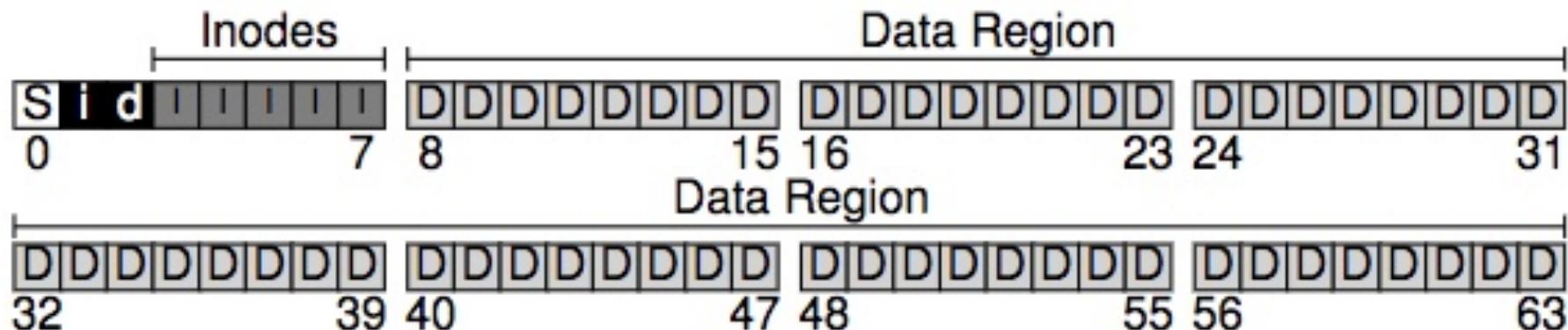
**Fast**

Workload B

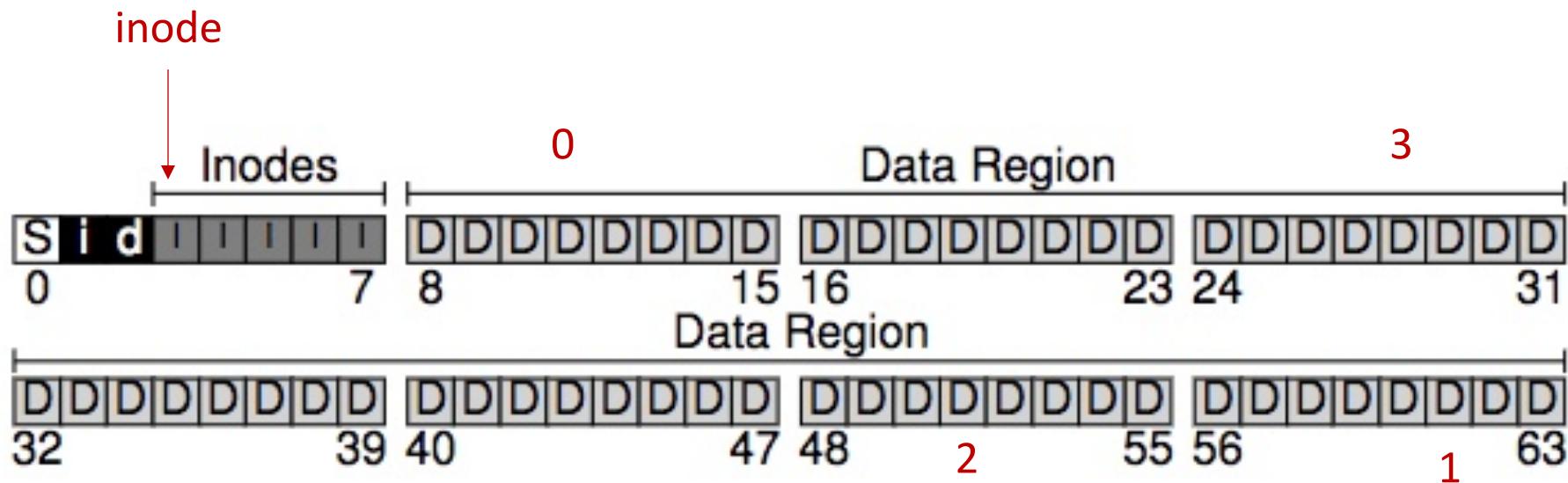


**Slow**

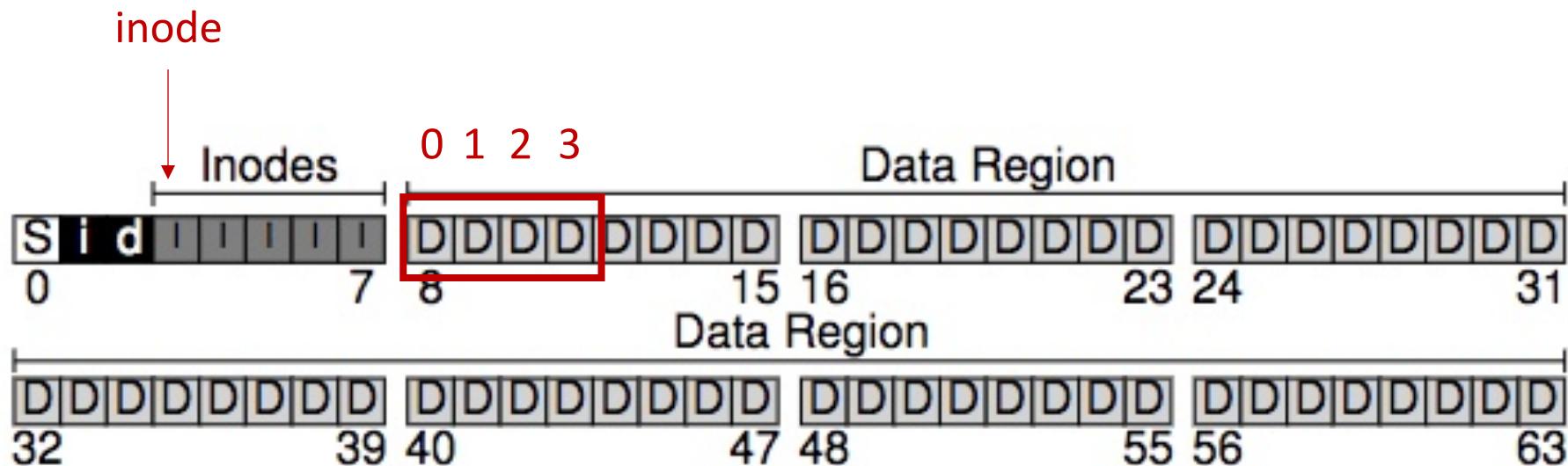
# Policy: Choose Inode, Data Blocks



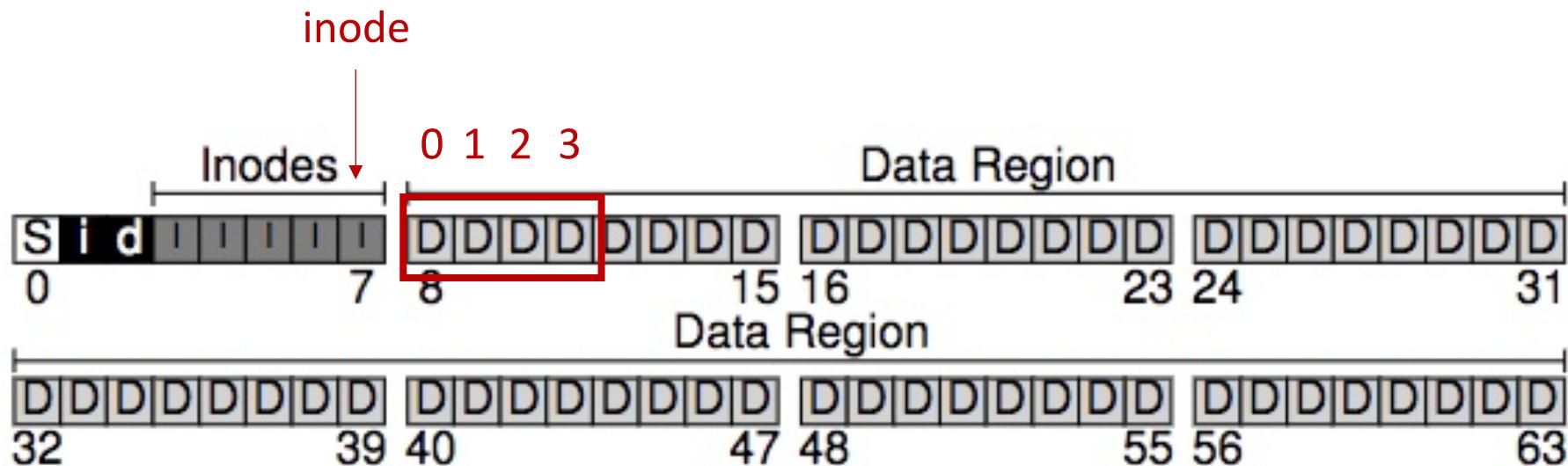
# Bad File Layout



# Better File Layout



# Best File Layout



# Properties of A Single Disk

- A single disk is slow
  - Kind of Okay sequential I/O performance
  - Really bad for random I/O

# Properties of A Single Disk

- A single disk is slow
  - Kind of Okay sequential I/O performance
  - Really bad for random I/O
- The storage capacity of a single disk is limited

# Properties of A Single Disk

- A single disk is slow
  - Kind of Okay sequential I/O performance
  - Really bad for random I/O
- The storage capacity of a single disk is limited
- A single disk is not reliable

# RAID: Redundant Array of Inexpensive Disks

# Wish List for a Disk

- Wish it to be **faster**
  - I/O is always the performance bottleneck

# Wish List for a Disk

- Wish it to be **faster**
  - I/O is always the performance bottleneck
- Wish it to be **larger**
  - More and more data needs to be stored

# Wish List for a Disk

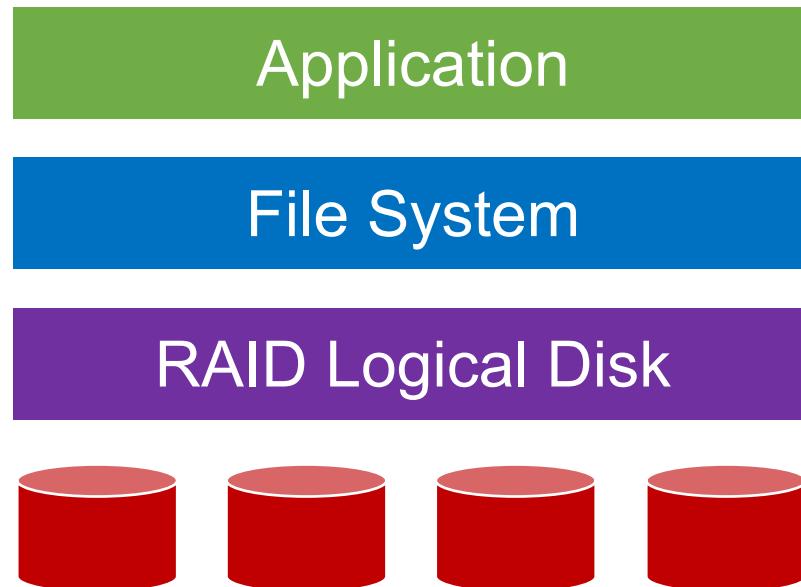
- Wish it to be **faster**
  - I/O is always the performance bottleneck
- Wish it to be **larger**
  - More and more data needs to be stored
- Wish it to be **more reliable**
  - We don't want our valuable data to be gone

# Only One Disk?

- Sometimes we want many disks
  - For higher performance
  - For larger capacity
  - For better reliability
- **Challenge:** Most file systems work on only one disk

# Solution: RAID

RAID: Redundant Array of Inexpensive Disks

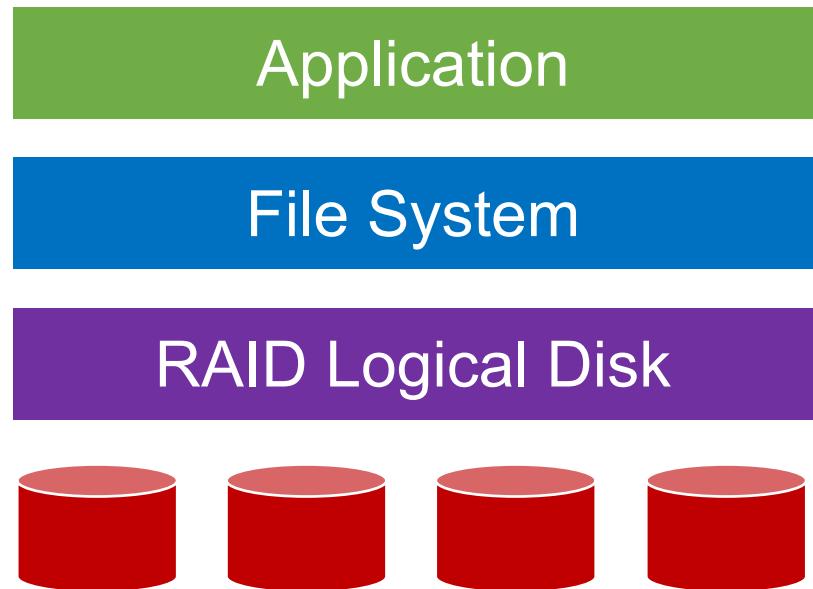


Build a logical disk from many physical disks

# Solution: RAID

RAID: Redundant Array of Inexpensive Disks

- RAID is
- Transparent
  - Deployable



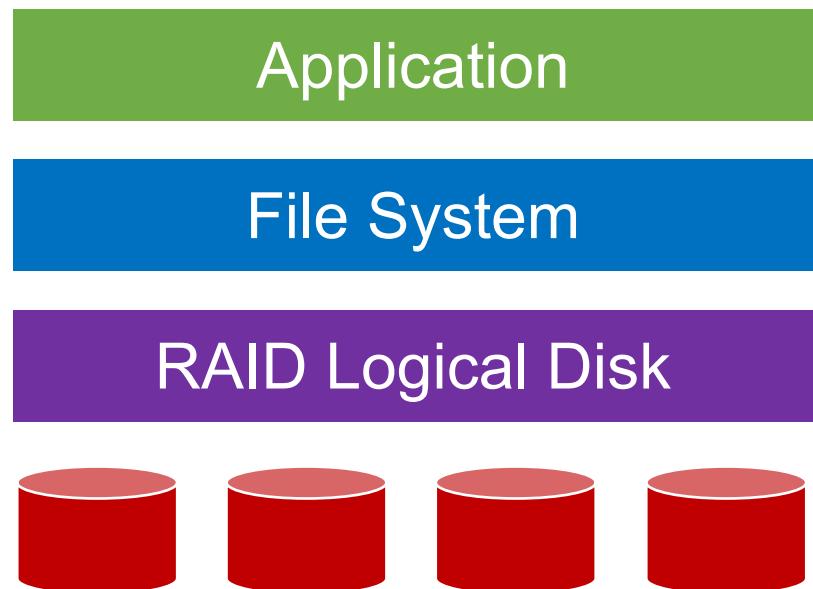
Build a logical disk from many physical disks

# Solution: RAID

RAID: Redundant Array of Inexpensive Disks

RAID is

- Transparent
- Deployable



Logical disks gives

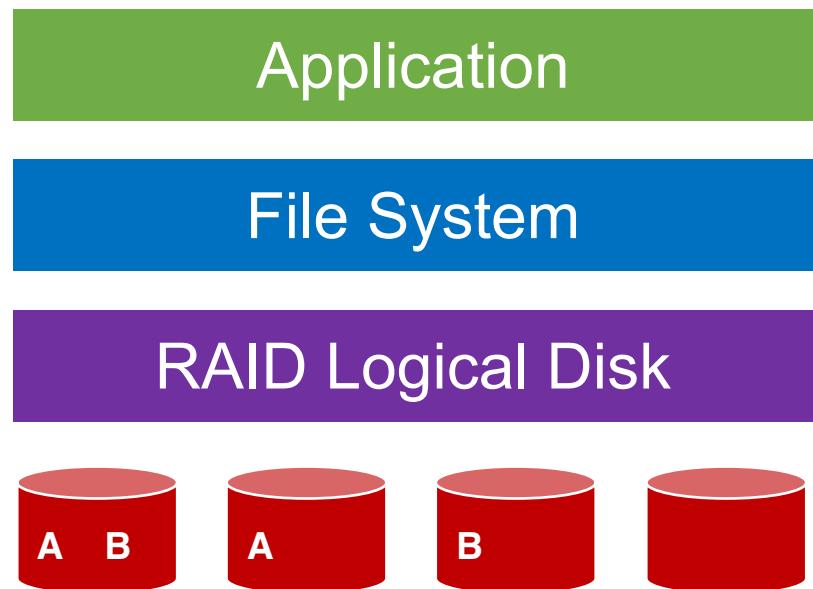
- Performance
- Capacity
- Reliability

Build a logical disk from many physical disks

# Solution: RAID

RAID: Redundant Array of Inexpensive Disks

- RAID is
- Transparent
  - Deployable



- Logical disks gives
- Performance
  - Capacity
  - Reliability

Build a logical disk from many physical disks

# Why Inexpensive Disks?

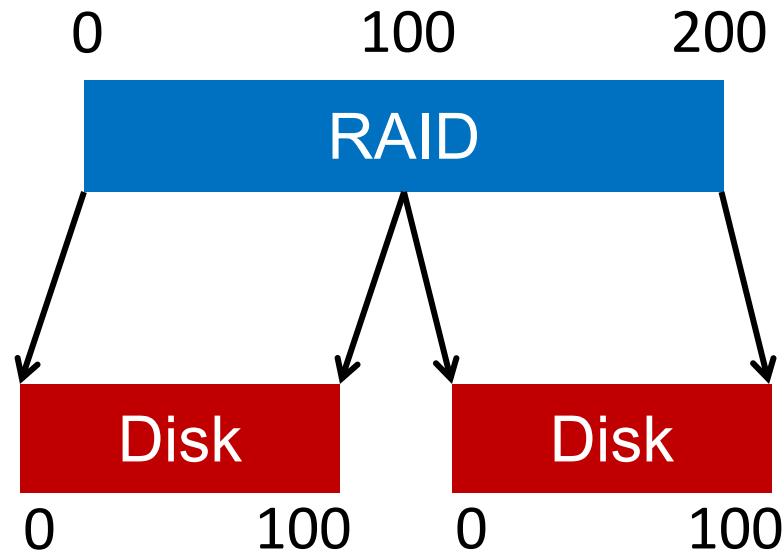
- Economies of scale! Cheap disks are popular.
- You can often get **many commodity** hardware components for the same price as a **few expensive** components

# Why Inexpensive Disks?

- Economies of scale! Cheap disks are popular.
- You can often get **many commodity** hardware components for the same price as a **few expensive** components
- Strategy: Write software to **build high-quality logical devices from many cheap devices**
  - Tradeoff: To compensate poor properties of cheap devices

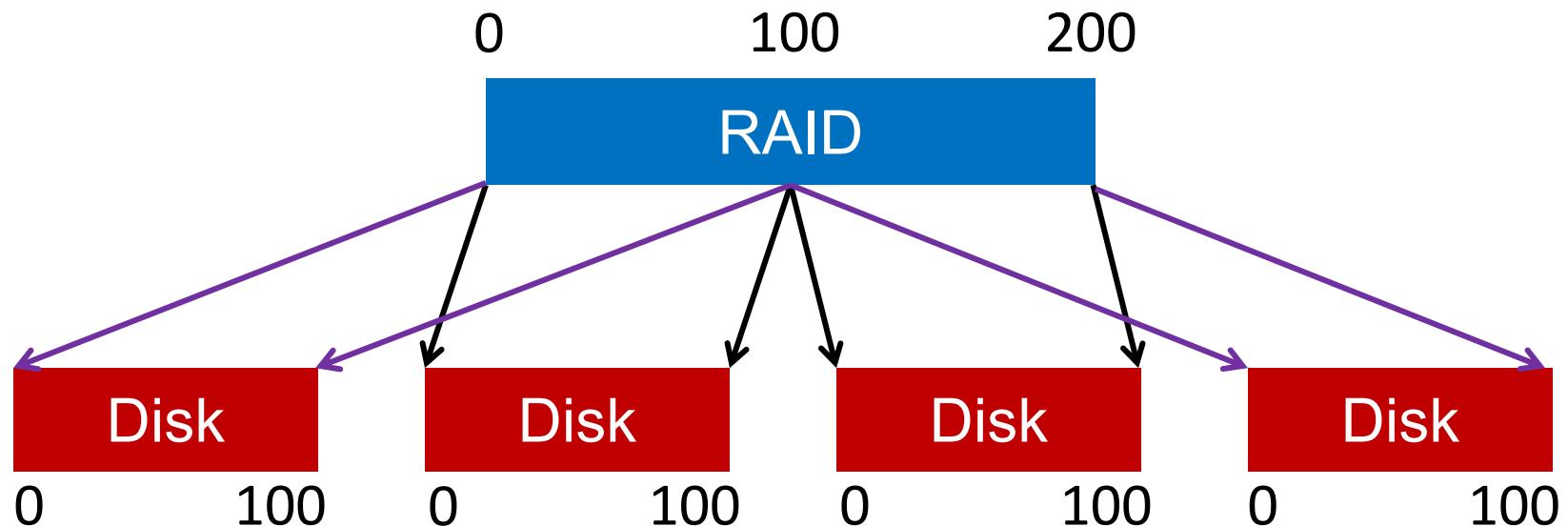
# General Strategy

Build fast and large disks from smaller ones



# General Strategy

Build fast and large disks from smaller ones  
Add more disks for **reliability++!**



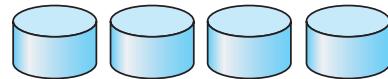
# RAID Metrics

- Performance
  - How long does each workload take?
- Capacity
  - How much space can apps use?
- Reliability
  - How many disks can we safely lose?

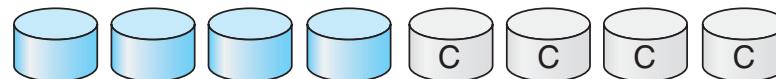
# RAID Metrics

- Performance
  - How long does each workload take?
- Capacity
  - How much space can apps use?
- Reliability
  - How many disks can we safely lose?
  - Assume **fail-stop** model!

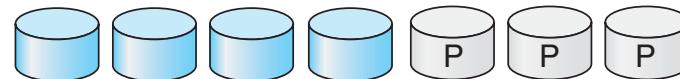
# RAID Levels



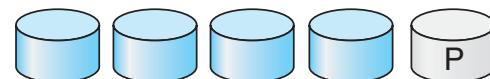
(a) RAID 0: non-redundant striping.



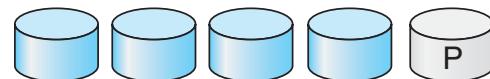
(b) RAID 1: mirrored disks.



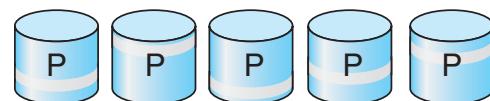
(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.

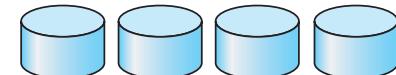


(e) RAID 4: block-interleaved parity.

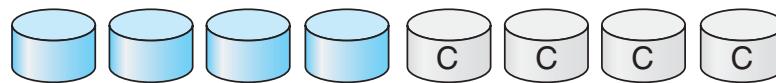


(f) RAID 5: block-interleaved distributed parity.

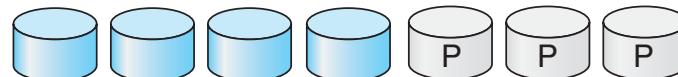
# RAID Level 0



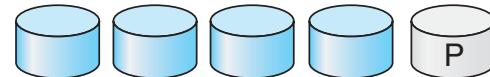
(a) RAID 0: non-redundant striping.



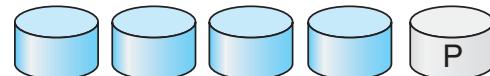
(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



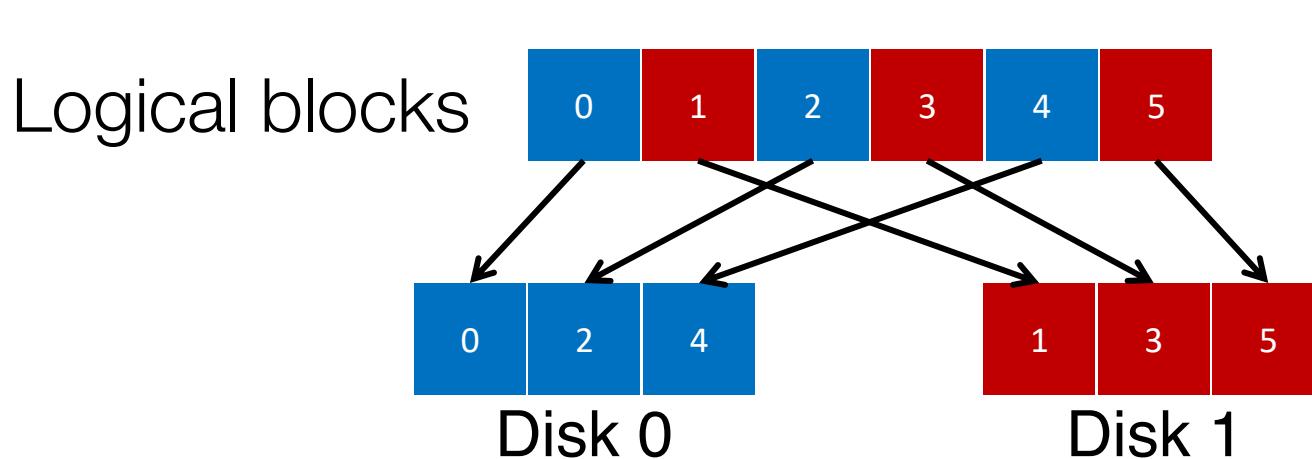
(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.

# RAID-0: Striping

- No redundancy
- Serves as **upper bound** for
  - Performance
  - Capacity



# 4 Disks

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

# 4 Disks

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

# How to Map?

- Given logical address A:
  - Disk = ...
  - Offset = ...

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

# How to Map?

- Given logical address A:
  - $\text{Disk} = A \% \text{ disk\_count}$
  - $\text{Offset} = A / \text{ disk\_count}$

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

# Mapping Example: Find Block 13

- Given logical address 13:

- $\text{Disk} = 13 \% 4 = 1$
- $\text{Offset} = 13 / 4 = 3$

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

# Chunk Size = 1

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

# Chunk Size = 1

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

# Chunk Size = 2

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

# Chunk Size = 1

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

In all following examples, we assume chunk size of 1

# Chunk Size = 2

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

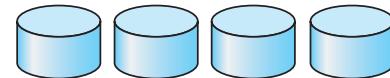
# RAID-0 Analysis

1. What is capacity?
2. How many disks can fail?
3. Throughput?
4. Latency?

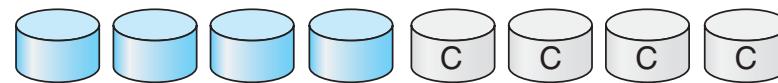
# RAID-0 Analysis

1. What is capacity?  $N * C$
2. How many disks can fail? 0
3. Throughput?  $N * S$  and  $N * R$
4. Latency? D

# RAID Level 1



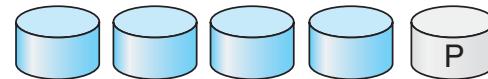
(a) RAID 0: non-redundant striping.



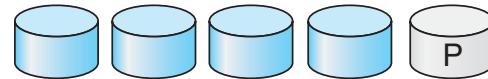
(b) RAID 1: mirrored disks.



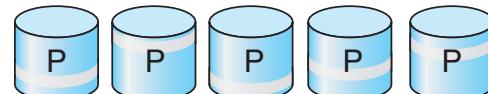
(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



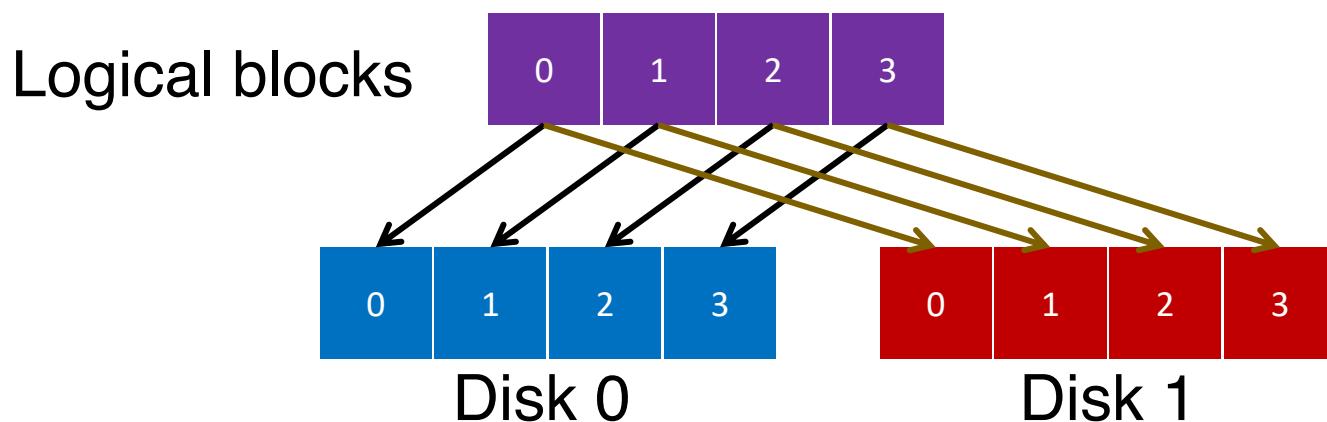
(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.

# RAID-1: Mirroring

- RAID-1 keeps two copies of each block



# Assumption

- Assume disks are **fail-stop**
  - Two states
    - They work or they don't
  - We know when they don't work

# 4 Disks

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

# 4 Disks

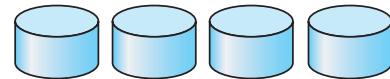
Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

**How many disks can fail?**

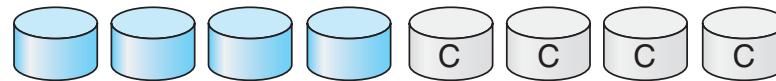
# RAID-1 Analysis

1. What is capacity?  $N/2 * C$
2. How many disks can fail?  $1$  or maybe  $N / 2$
3. Throughput?
  - Seq read:  $N/2 * S$
  - Seq write:  $N/2 * S$
  - Rand read:  $N * R$
  - Rand write:  $N/2 * R$
4. Latency?  $D$

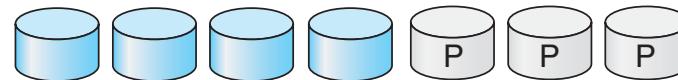
# RAID Level 4



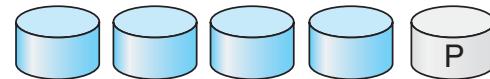
(a) RAID 0: non-redundant striping.



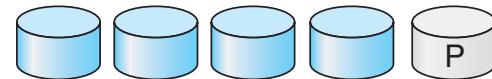
(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.

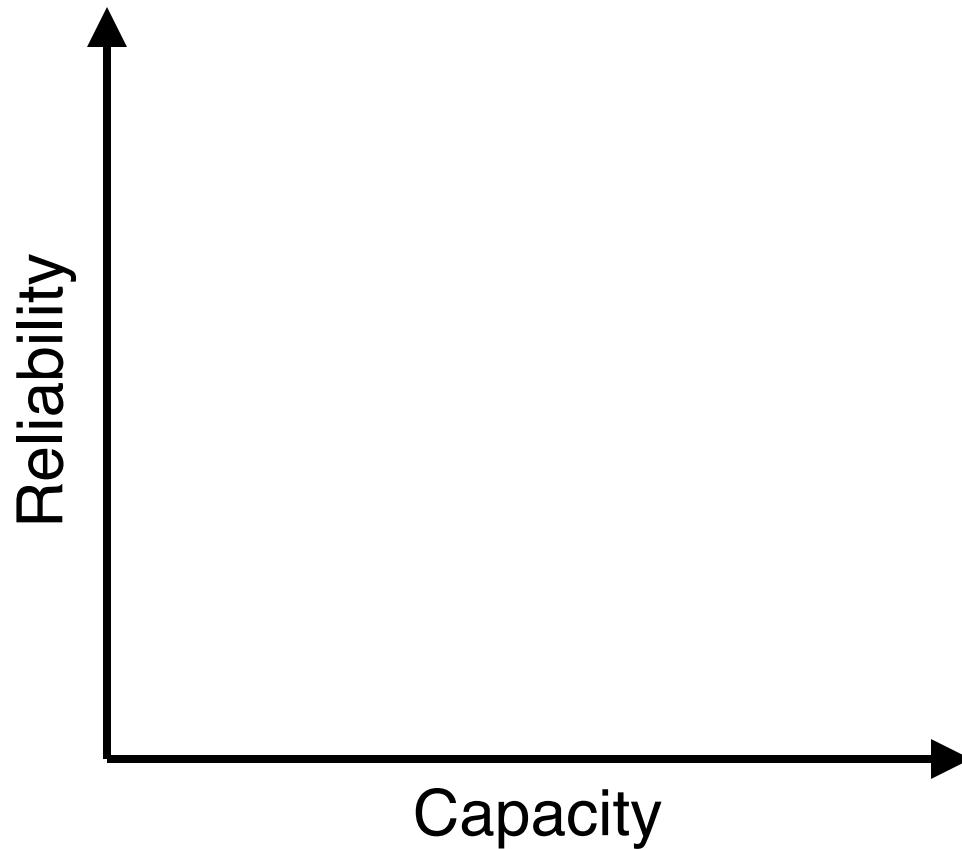


(e) RAID 4: block-interleaved parity.

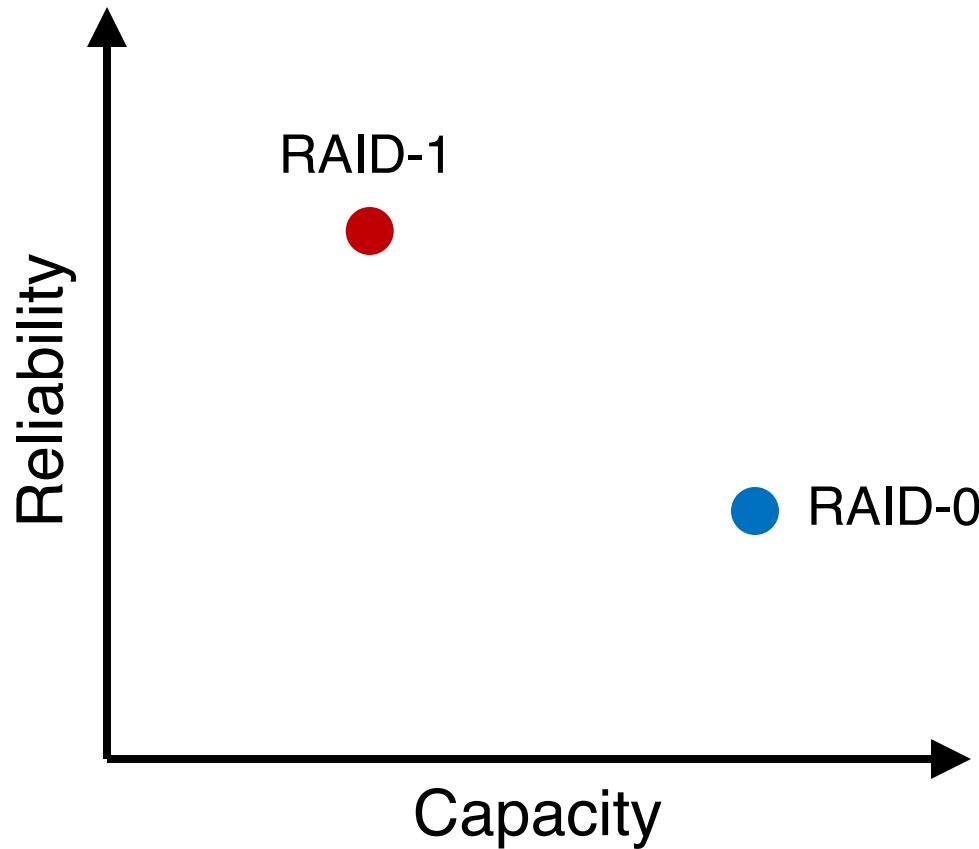


(f) RAID 5: block-interleaved distributed parity.

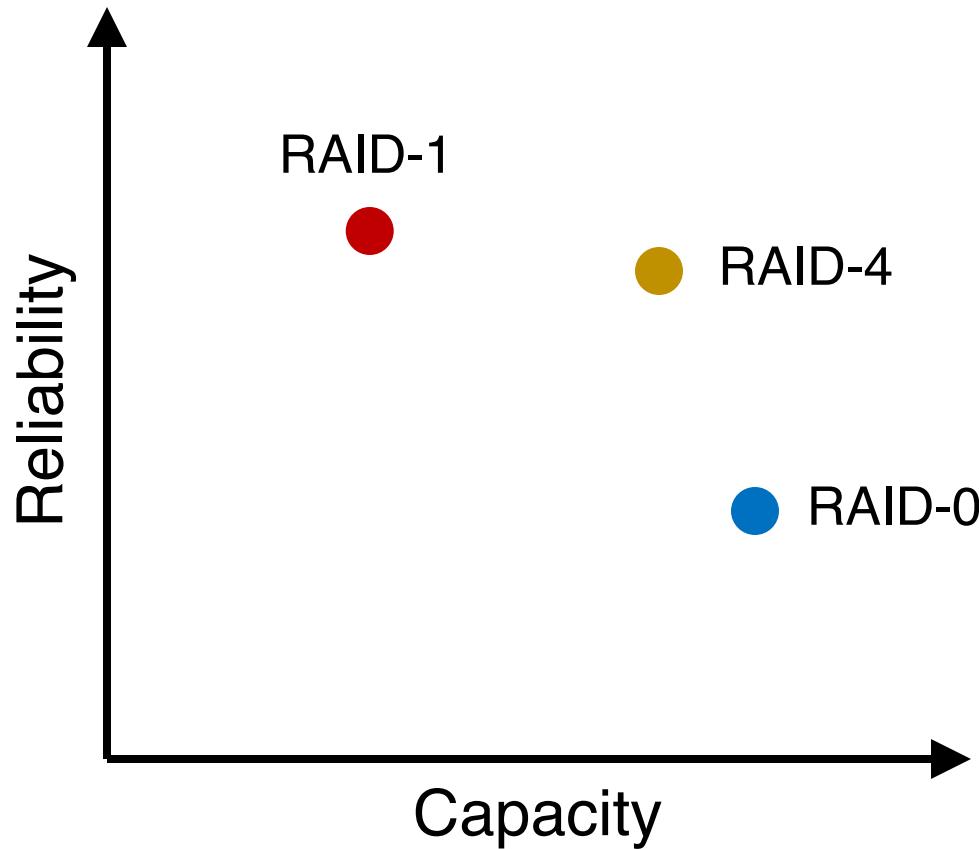
# RAID-4



# RAID-4



# RAID-4



# RAID-4: Strategy

- Use **parity** disk
- In algebra, if an **equation** has  $N$  variables, and  $N-1$  are known, you can also solve for the unknown
- Treat the sectors/blocks across disks in a stripe as an equation

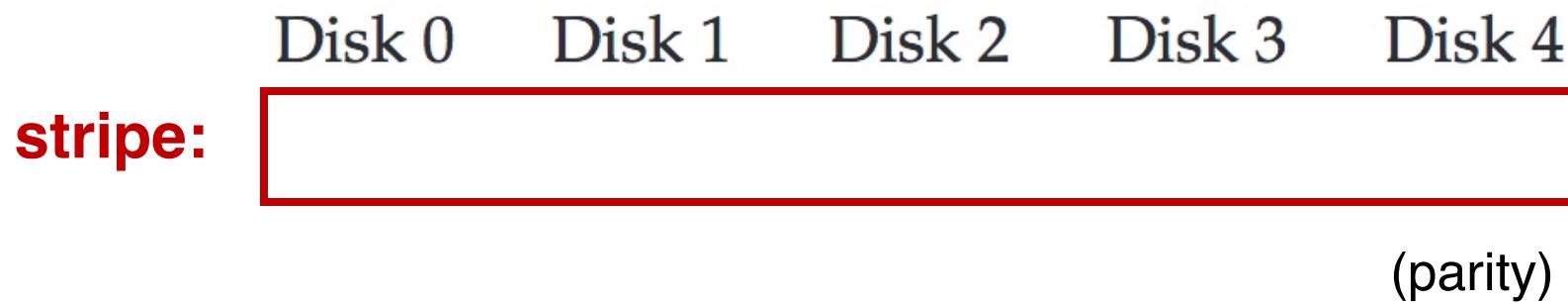
# RAID-4: Strategy

- Use **parity** disk
- In algebra, if an **equation** has  $N$  variables, and  $N-1$  are known, you can also solve for the unknown
- Treat the sectors/blocks across disks in a stripe as an equation
- A **failed disk** is like an unknown **in that equation**

# 5 Disks

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

# Example



# Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
<b>stripe:</b>	4	3	0	2	

(parity)

# Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
stripe:	4	3	0	2	9
					(parity)

# Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
stripe:	X	3	0	2	9
					(parity)

# Example

	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
stripe:	4	3	0	2	9
					(parity)

# Parity Function: XOR Example

C0	C1	C2	C3	P
0	0	1	1	$\text{XOR}(0,0,1,1) = 0$
0	1	0	0	$\text{XOR}(0,1,0,0) = 1$

# Parity Function: XOR Example

C0	C1	C2	C3	P
0	0	1	1	$\text{XOR}(0,0,1,1) = 0$
0	1	0	0	$\text{XOR}(0,1,0,0) = 1$

XOR function:

- $P = 0$ : The number of 1 in a stripe must be an even number
- $P = 1$ : The number of 1 in a stripe must be an odd number

# Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	00	10	11	10	11
	10	01	00	01	10

XOR function:

- $P = 0$ : The number of 1 in a stripe must be an even number
- $P = 1$ : The number of 1 in a stripe must be an odd number

# Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	X	10	11	10	11
	10	01	00	01	10

XOR function:

- $P = 0$ : The number of 1 in a stripe must be an even number
- $P = 1$ : The number of 1 in a stripe must be an odd number

# Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	X	10	11	10	11
	10	01	00	01	10

$$\text{Block0} = \text{XOR}(10, 11, 10, 11) = 00$$

XOR function:

- $P = 0$ : The number of 1 in a stripe must be an even number
- $P = 1$ : The number of 1 in a stripe must be an odd number

# Parity Function: XOR Example

	Block0	Block1	Block2	Block3	Parity
stripe:	00	10	11	10	11
	10	01	00	01	10

$$\text{Block0} = \text{XOR}(10, 11, 10, 11) = \mathbf{00}$$

XOR function:

- $P = 0$ : The number of 1 in a stripe must be an even number
- $P = 1$ : The number of 1 in a stripe must be an odd number

# RAID-4 Analysis

1. What is capacity?  $(N-1) * C$

2. How many disks can fail? 1

3. Throughput?

- Seq read:  $(N-1) * S$
- Seq write:  $(N-1) * S$
- Rand read:  $(N-1) * R$
- Rand write:  $R/2$

4. Latency? D, 2D

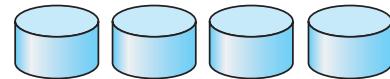
# RAID-4 Analysis: Random Write

Random write to 4, 13, and respective parity blocks

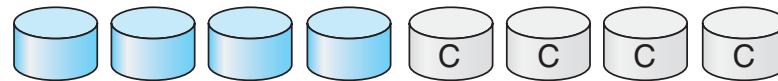
Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

**Small write problem** (for parity-based RAIDs):  
Parity disk serializes all random writes; and each **logical I/O** generates two **physical I/Os (one read and one write for parity P1)**

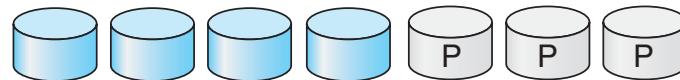
# RAID Level 5



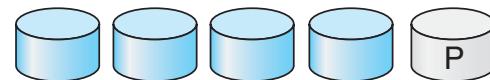
(a) RAID 0: non-redundant striping.



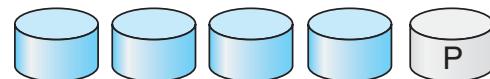
(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



# RAID-5: Rotating Parity

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 works almost identically to RAID-4, except that it rotates the parity block across drives

# RAID-5 Analysis

1. What is capacity?  $(N-1) * C$

2. How many disks can fail? 1

3. Throughput?

- Seq read:  $(N-1) * S$
- Seq write:  $(N-1) * S$
- Rand read:  $N * R$
- Rand write: ???

4. Latency? D, 2D

# RAID-5: Random Write

Write	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

Random write to Block 10 on Disk 0

# RAID-5: Random Write

1. Read

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

Random write to Block 10 on Disk 0

1. Read Block 10

# RAID-5: Random Write

1. Read		2. Read		
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

Random write to Block 10 on Disk 0

1. Read Block 10
2. Read the Parity P2

# RAID-5: Random Write

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

Random write to Block 10 on Disk 0

1. Read Block 10
2. Read the Parity P2
3. Write new data in Block 10

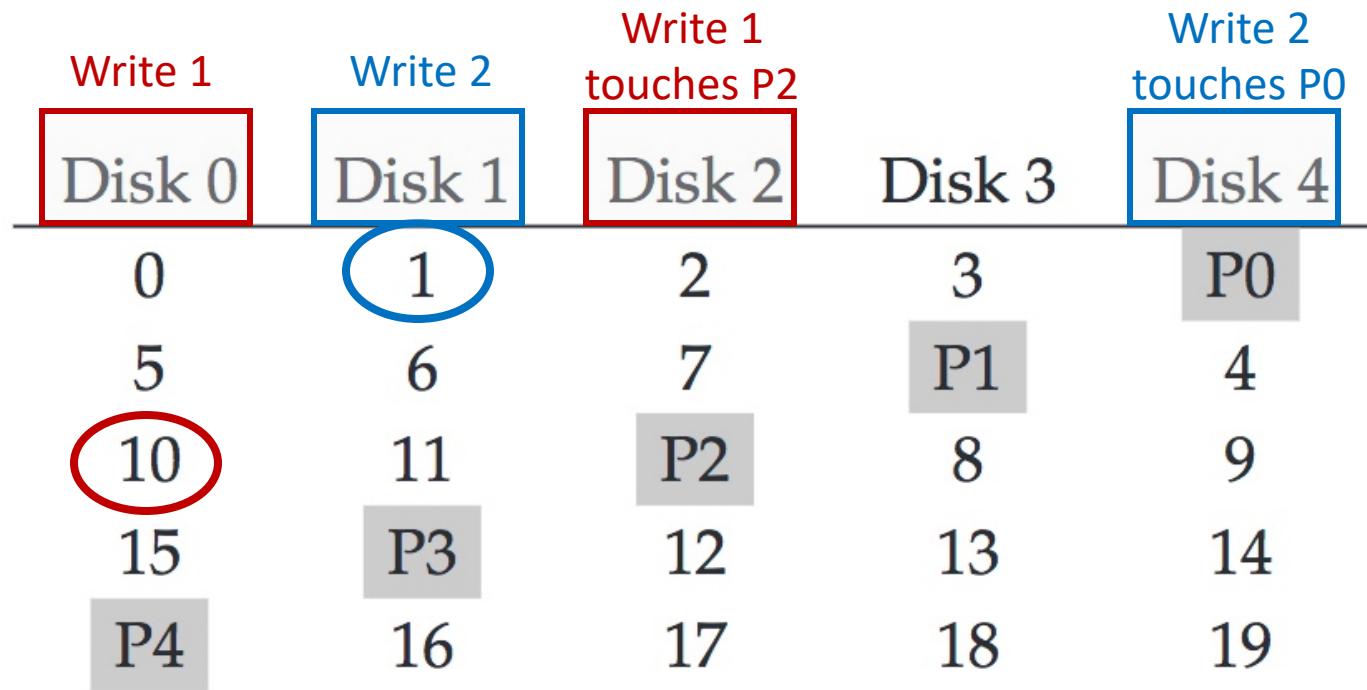
# RAID-5: Random Write

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

Random write to Block 10 on Disk 0

1. Read Block 10
2. Read the Parity P2
3. Write new data in Block 10
4. Write new parity P2

# RAID-5: Random Write



Performance reasoning

Generally, for a large number of random read/write requests, RAID-5 will be able to keep all disks busy: thus  $N * R$

Each random (RAID-5) writes generates 4 physical I/O operations:  
thus  $N * R / 4$

# RAID-5 Analysis

1. What is capacity?  $(N-1) * C$

2. How many disks can fail? 1

3. Throughput?

- Seq read:  $(N-1) * S$
- Seq write:  $(N-1) * S$
- Rand read:  $N * R$
- Rand write:  $N * R/4$

4. Latency? D, 2D

# Summary: All RAID's

	Reliability	Capacity
RAID-0	0	$C * N$
RAID-1	1 or $N/2$	$C * N/2$
RAID-4	1	$N-1$
RAID-5	1	$N-1$

# Summary: All RAID's

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	$N * S$	$N * S$	$N * R$	$N * R$
RAID-1	$N/2 * S$	$N/2 * S$	$N * R$	$N/2 * R$
RAID-4	$(N-1) * S$	$(N-1) * S$	$(N-1) * R$	$R/2$
RAID-5	$(N-1) * S$	$(N-1) * S$	$N * R$	$N/4 * R$

# Please Read the Textbook!

Please do read the textbook chapter “RAID” to gain a deeper understanding of the various analyses covered in lecture.