Hello!

PERSISTENCE: FILE SYSTEMS

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Midterm 2? – This week

Project 5 – Due Nov 19th (one week from today)

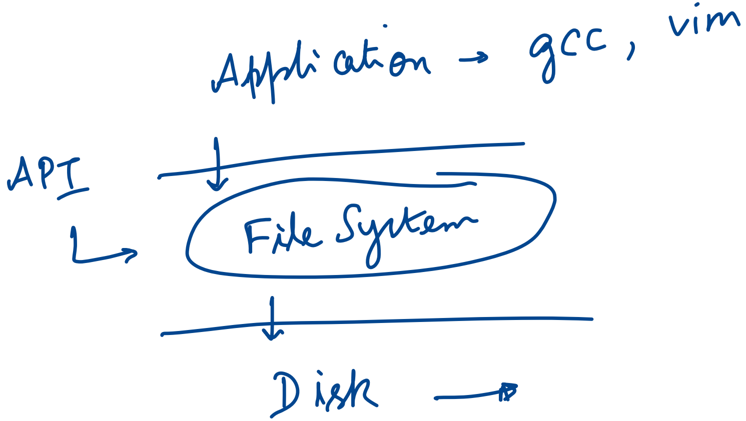
Shivaram's OH / next week schedule → Zoom office hours next week

→ Today 3-4 pm

AGENDA / LEARNING OUTCOMES

How does file system represent files, directories?

What steps must reads/writes take?



RECAP

FILE API WITH FILE DESCRIPTORS

file descriptors

```
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)
```

Impl

advantages:

- string names
- hierarchical
- traverse once
- offsets precisely defined

track

STRACE

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
prompt> strace cat foo -- prints system calls performed by program
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
hello
read(3, "", 4096) = 0
close(3) = 0
...
prompt>
```

fd for foo

6 bytes

stdout

LINKS

metadata associated with a file

Hard links: Both path names use same inode number

File does not disappear until all hard links removed; cannot link directories

short cut

Soft or symbolic links: Point to second path name; can softlink to dirs

Can cause confusing behavior: "file does not exist"!

ls /usr/lib/libpthread.so → libpthread1.5.so

FILE API SUMMARY

Using multiple types of name provides convenience and efficiency

Hard and soft link features provide flexibility.

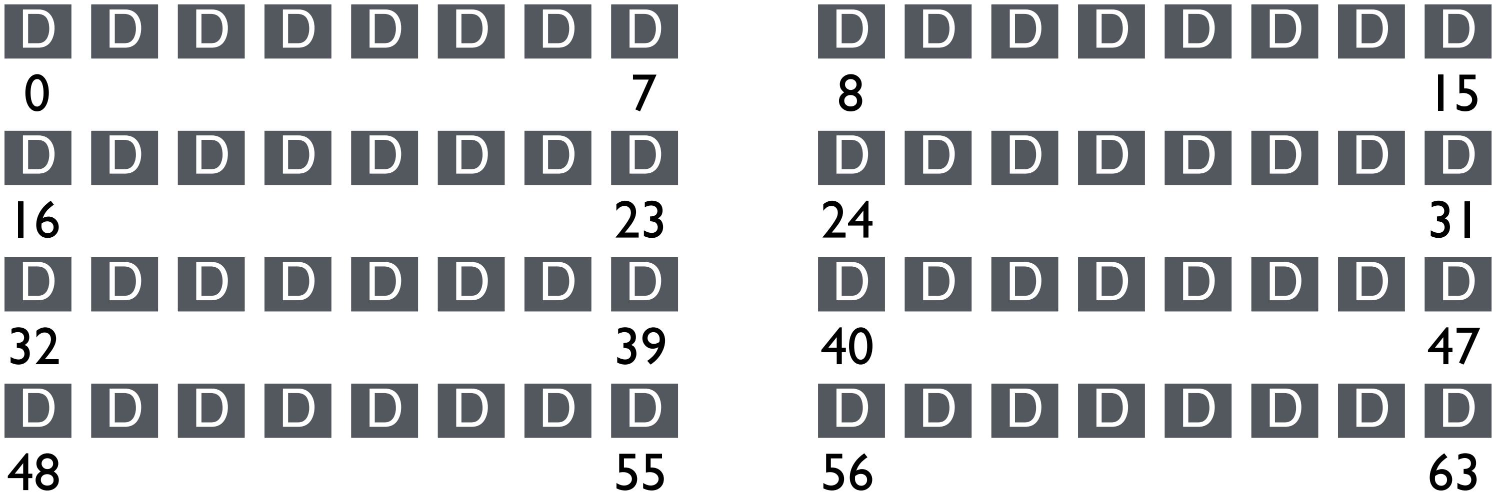
Special calls (fsync, rename) let developers communicate requirements to file system

FILESYSTEM DISK STRUCTURES

FS STRUCTS: EMPTY DISK

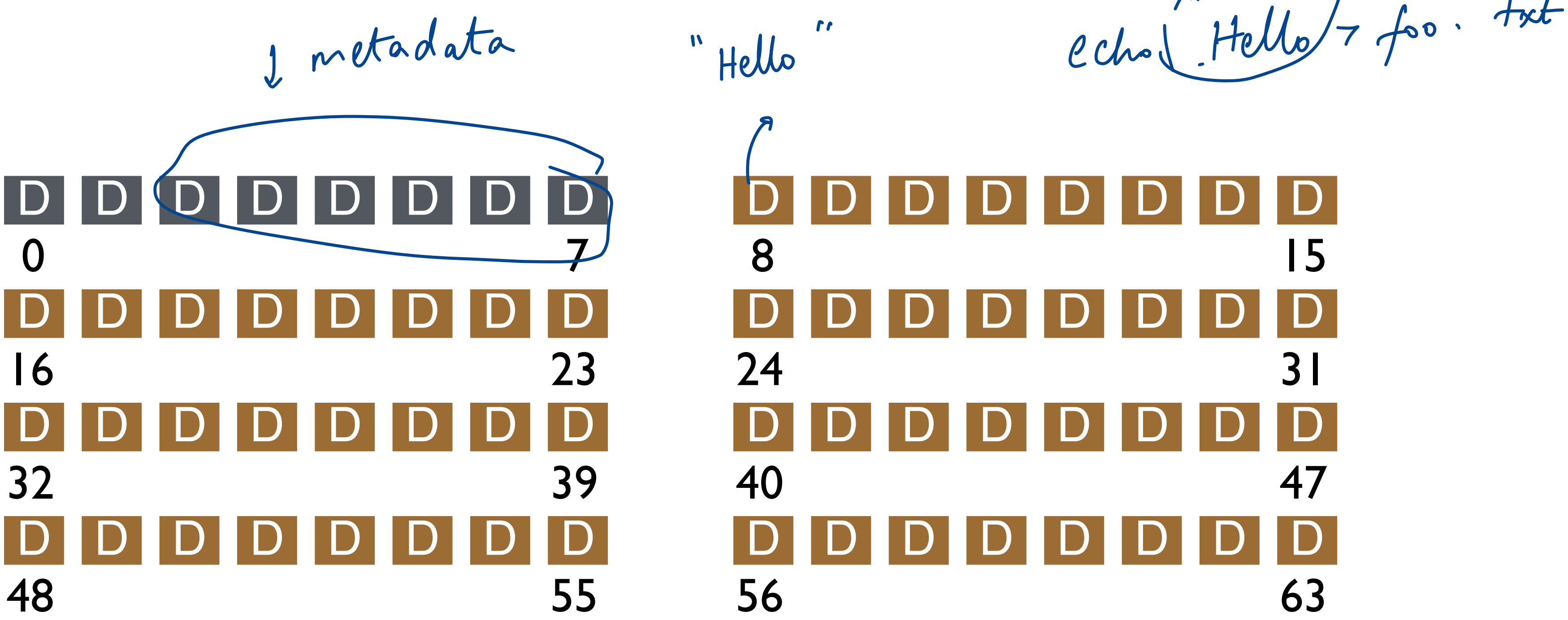
layout a filesystem on this disk

block = 4KB in size



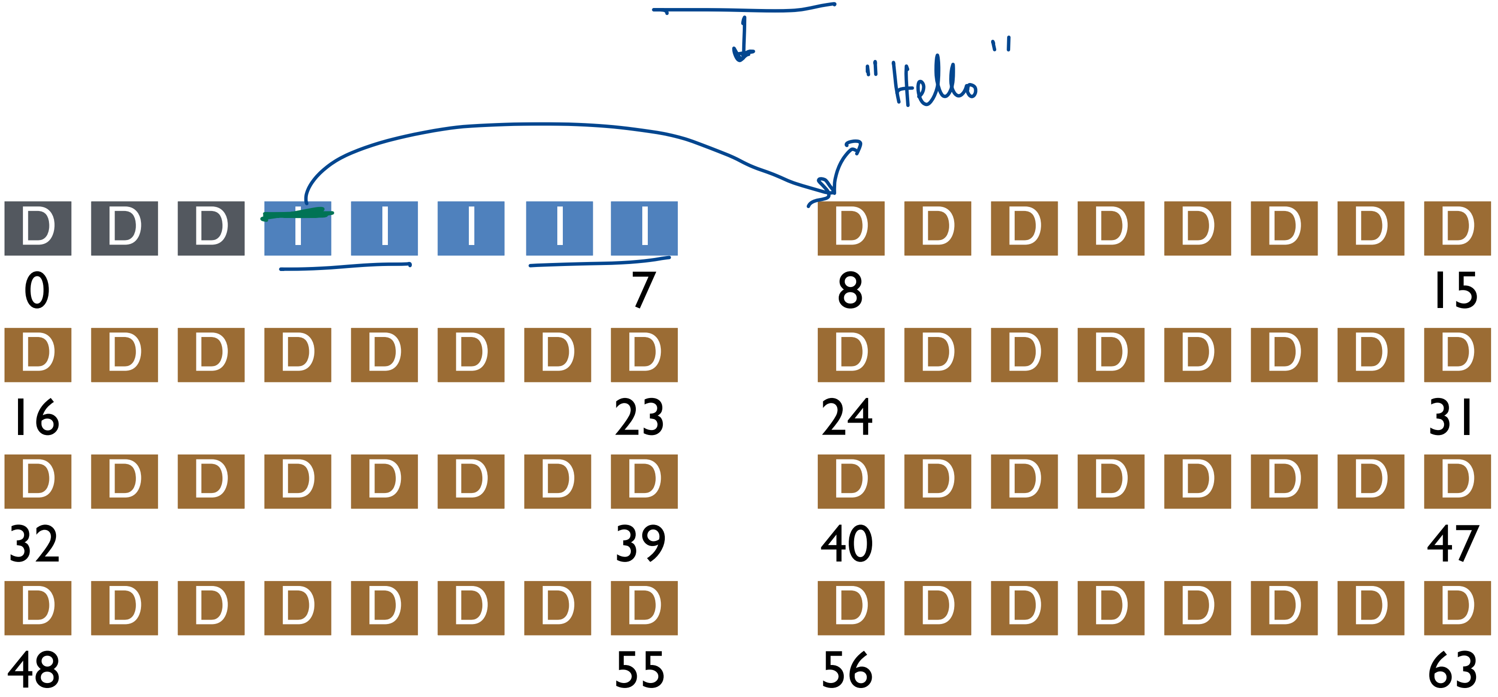
Assume each block is 4KB

FS STRUCTS: DATA BLOCKS



Simple layout → Very Simple File System

FS STRUCTS: INODE DATA POINTERS



INODE

/ foo.txt

└→ Directory

- type (file or dir?) → foo.txt file
- uid (owner)
- rwx (permissions) → user 1 can read / write etc.
- size (in bytes)
- Blocks
- time (access)
- ctime (create)
- links_count (# paths) → number of hard links
- addrs[N] (N data blocks) → links to where this file's content are present

ONE INODE BLOCK

inode 1 block

Each inode is typically 256 bytes (depends on the FS, maybe 128 bytes)

4KB disk block

16 inodes per inode block. $= \frac{4KB}{256} = 16$ inodes

5 blocks for inodes

⇒ 80 inodes in my simple FS

INODE

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

Assume single level (just pointers to data blocks)

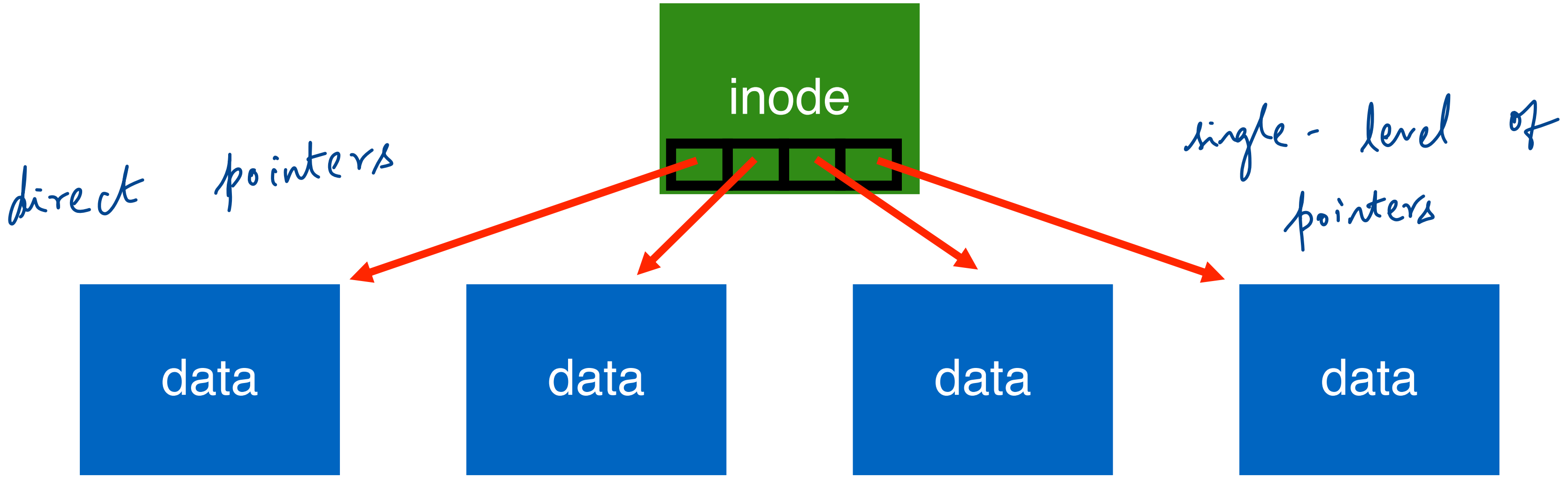
What is max file size?

Assume 256-byte inodes
(all can be used for pointers)
Assume 4-byte addrs

⇒ 64 addrs / pointers

How to get larger files?

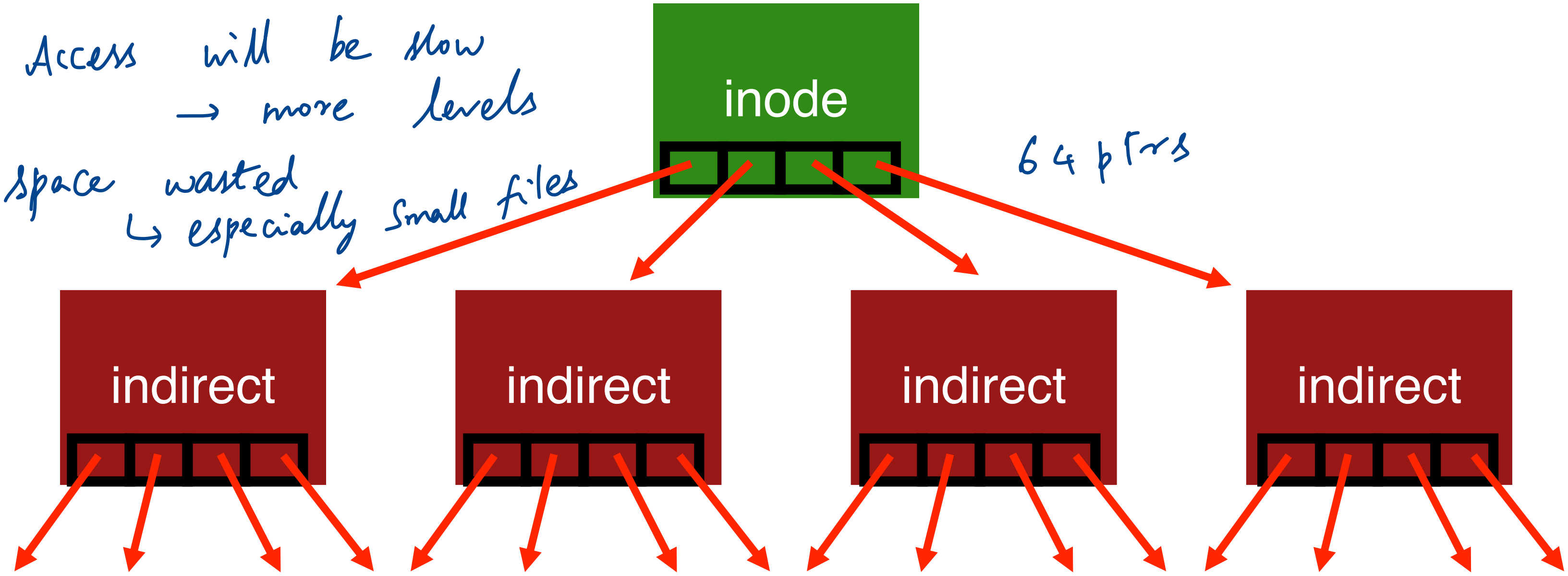
Max file size
= 64 × 4 KB = 256 KB



inode
direct pointers
single-level of pointers
data
data
data
data

① Access will be slow
→ more levels

② space wasted
↳ especially small files



Indirect blocks are stored in regular data blocks

Largest file size with 64 indirect blocks?

Any Cons?

1 indirect block

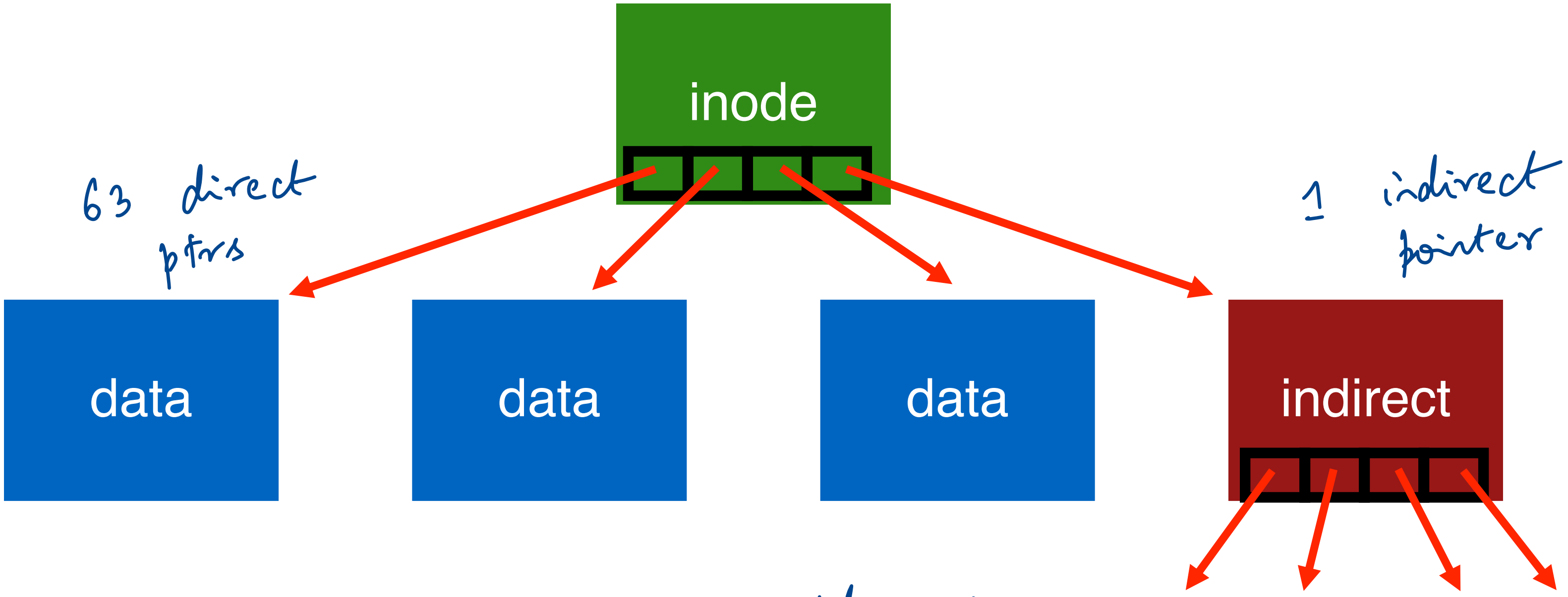
$= \frac{4KB}{4} = 1024$ address

→ addr data block = 4 bytes

4KB size

→ 1024 × 4KB = 4MB
↳ 1 indirect block

64 × 4MB = 256 MB



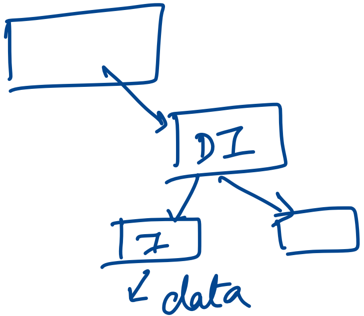
Better for small files! → directly store on data blocks

How to handle even larger files?

& 1 traversal

→ double indirect block

→ triple indirect block



DIRECTORIES

File systems vary

Common design:

Store directory entries in data blocks

Large directories just use multiple data blocks

Use bit in inode to distinguish directories from files

Various formats could be used

- lists
- b-trees

struct dir_entry {
valid
name
inode
}

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

cur dir

parent dir

just store a list of dir_entry inside data block

FS STRUCTS: BITMAPS

How do we find free data blocks or free inodes?

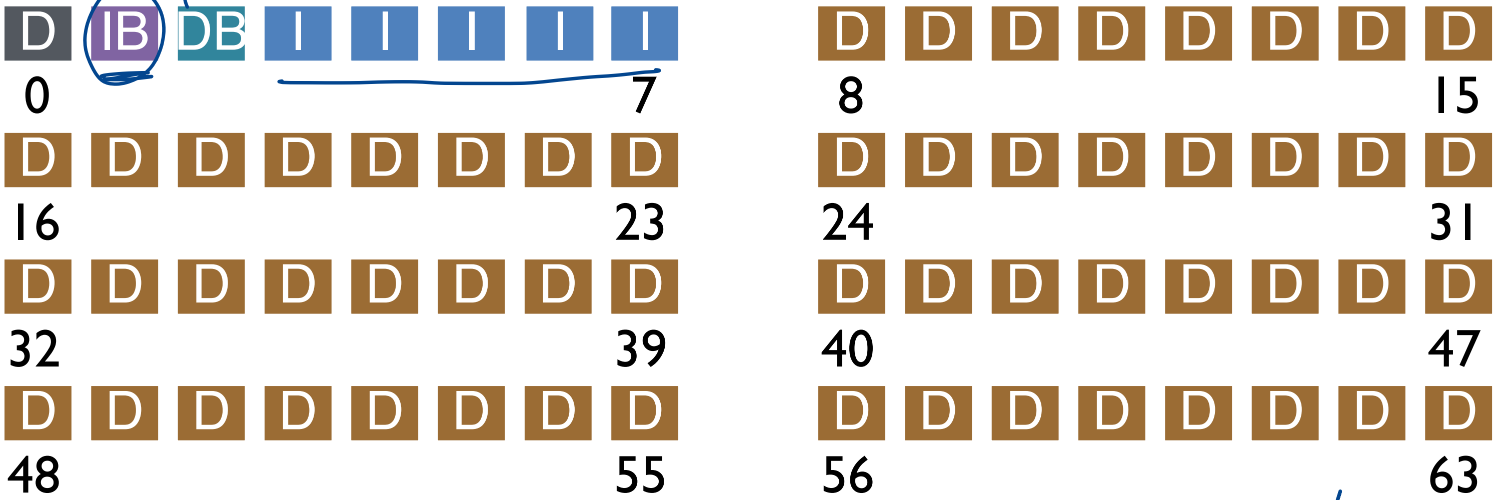
metadata

→ allocate a DB

80 bits

64 bits

80 inodes



Bitmap : 1 bit to indicate used or not

⇒ inode 3 used ? bit 3 in IB

SUPERBLOCK

Need to know basic FS configuration metadata, like:

- block size
- # of inodes

Store this in superblock

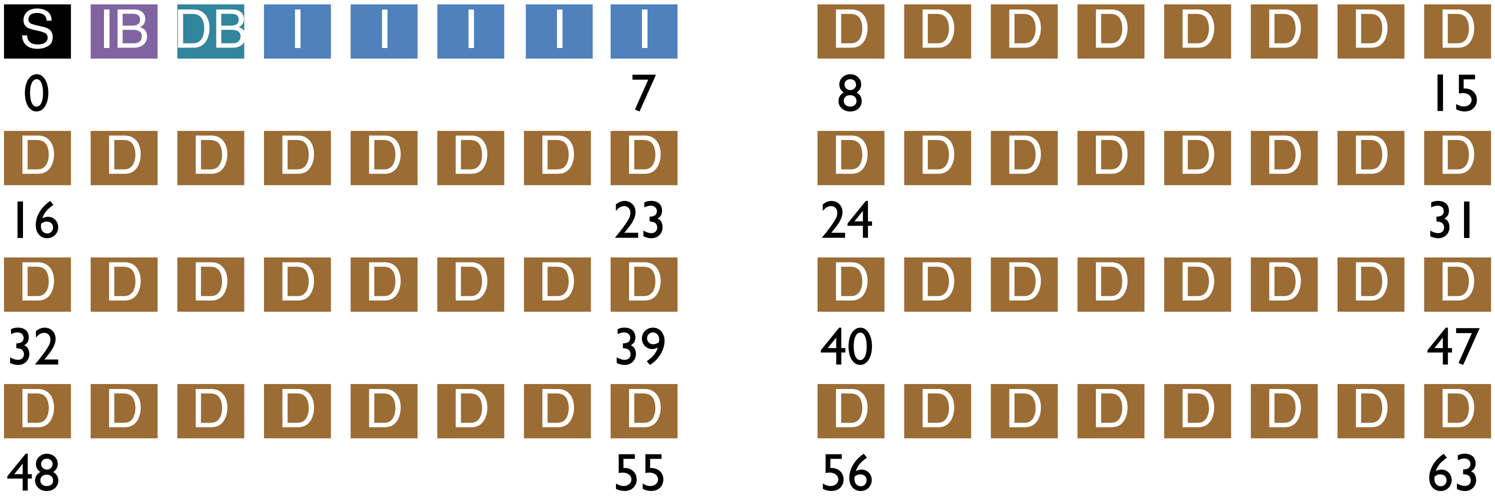
→ 1st block on disk

read the SB when you mount a FS

↳ block size = 4KB

→ 80 inode, 256 bytes inode

FS STRUCTS: SUPERBLOCK



QUIZ 16

The following command is executed:

```
echo "Hello World" > foo.txt
```

What produces

```
openat(AT_FDCWD, "foo.txt", O_RDONLY)     = 3
read(3, "Hello World\n", 131072)          = 12
write(1, "Hello World\n", 12)             = 12
read(3, "", 131072)                       = 0
close(3)                                  = 0
close(1)                                  = 0
```

cat foo.txt

QUIZ 16

```
openat(AT_FDCWD, "foo.txt", O_RDONLY|O_NOCTTY) = 3
read(3, "Hello World\n", 98304)          = 12
read(3, "", 98304)                       = 0
close(3)                                 = 0
close(1)                                 = 0
```

→ grep "Goodbye" foo.txt

```
openat(AT_FDCWD, "/proc/filesystems", O_RDONLY|O_CLOEXEC) = 3
close(3)                                 = 0
statx(AT_FDCWD, "foo.txt", ...) = 0
write(1, "-rw-rw-r-- 1 oliphant oliphant 1"..., 55) = 55
close(1)
```

→ output ls produces

```
int fd1 = open("file.txt"); // returns 12
int fd2 = open("file.txt"); // returns 13
read(fd1, buf, 16);
int fd3 = dup(fd2);         // returns 14
read(fd2, buf, 16);
lseek(fd1, 100, SEEK_SET);
```

offset for fd 1 → 100

FS OPERATIONS

- open
- read
- close
- create file
- write

TIME

open /foo/bar → fd to the calling process

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
		(1) read					
					(2) read		
			(3)read				
						(4)read	
				(5)read			

① file exists → first read '/'
↳ inode → data block
↳ read data block
↳ inode /foo

② permission to read this files
↳ read inode for /foo/bar

read /foo/bar – assume opened → fd

TIME ↓

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				(1) read			
							(2) read
				(3)write			

read: given offset & number of bytes

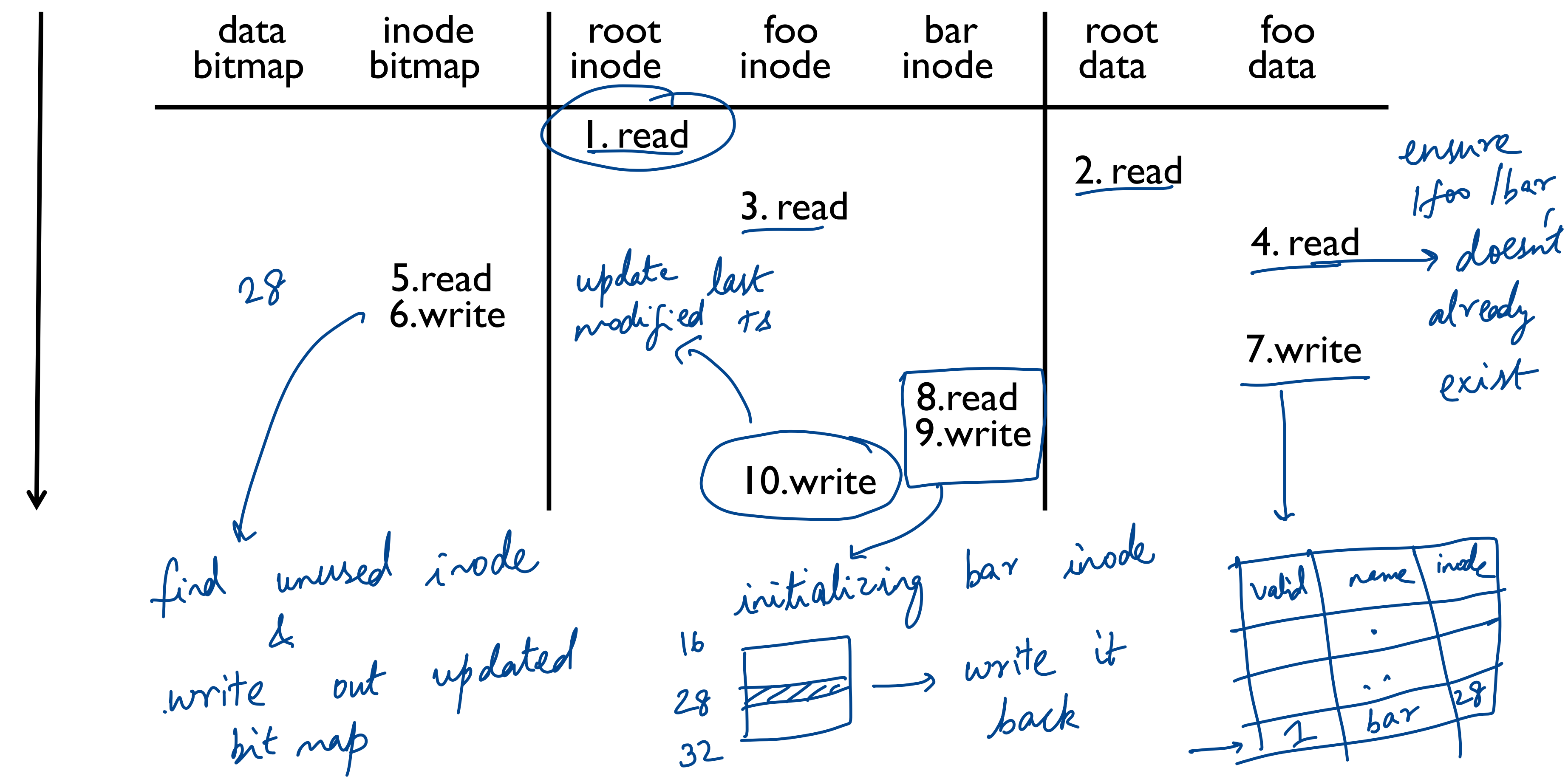
1. which data block has offset → offset 4097 → second data block
2. read block
3. update last accessed ts.

close /foo/bar

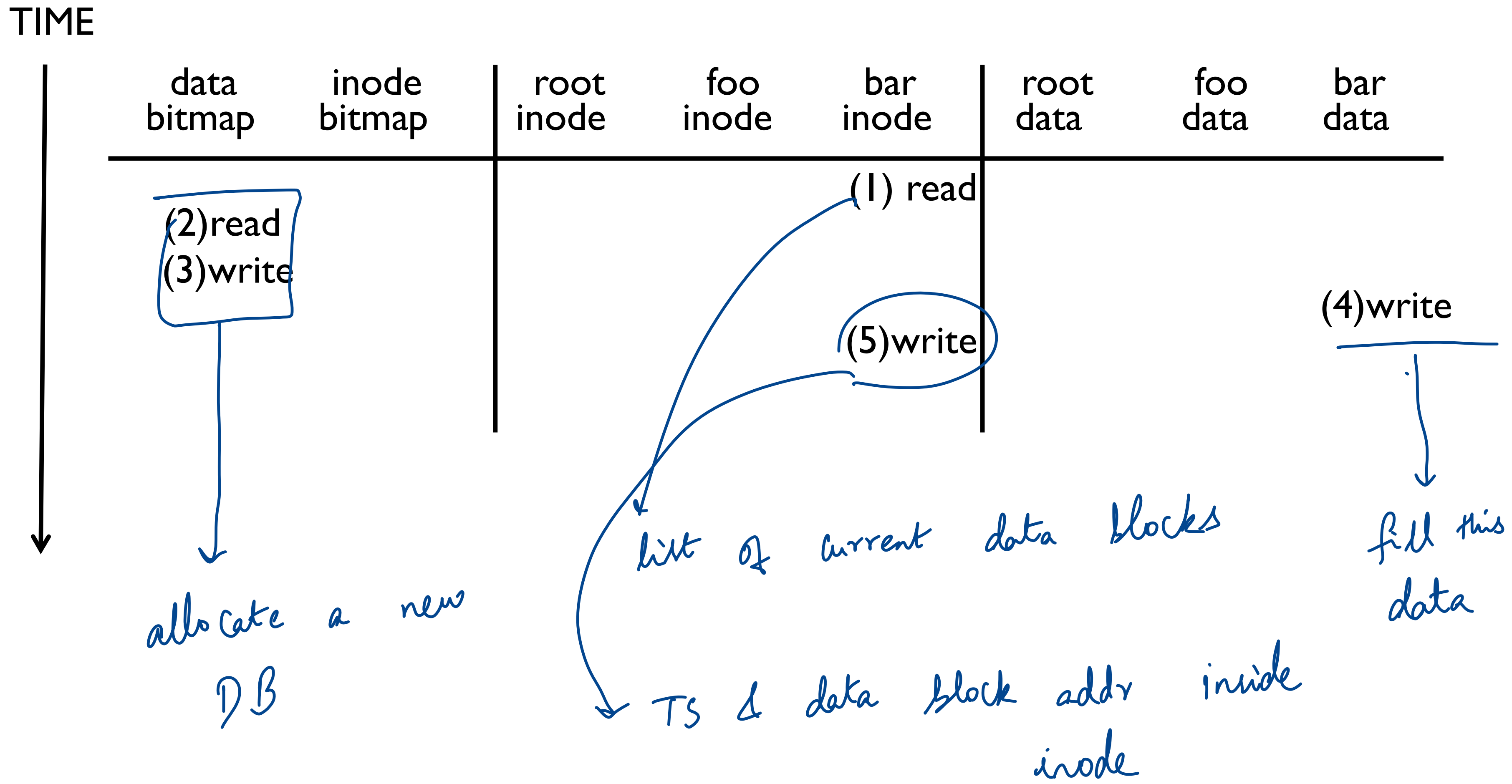
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

nothing to do on disk!

create /foo/bar



write to /foo/bar (assume file exists and has been opened)



EFFICIENCY

Crashes with writes buffered!?

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

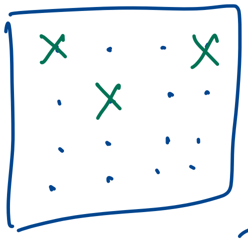
Cache for:

- reads
- write buffering → delay when writes are made to a FS

Overwrites, deletes, scheduling → create a file ⇒ update inode bitmap

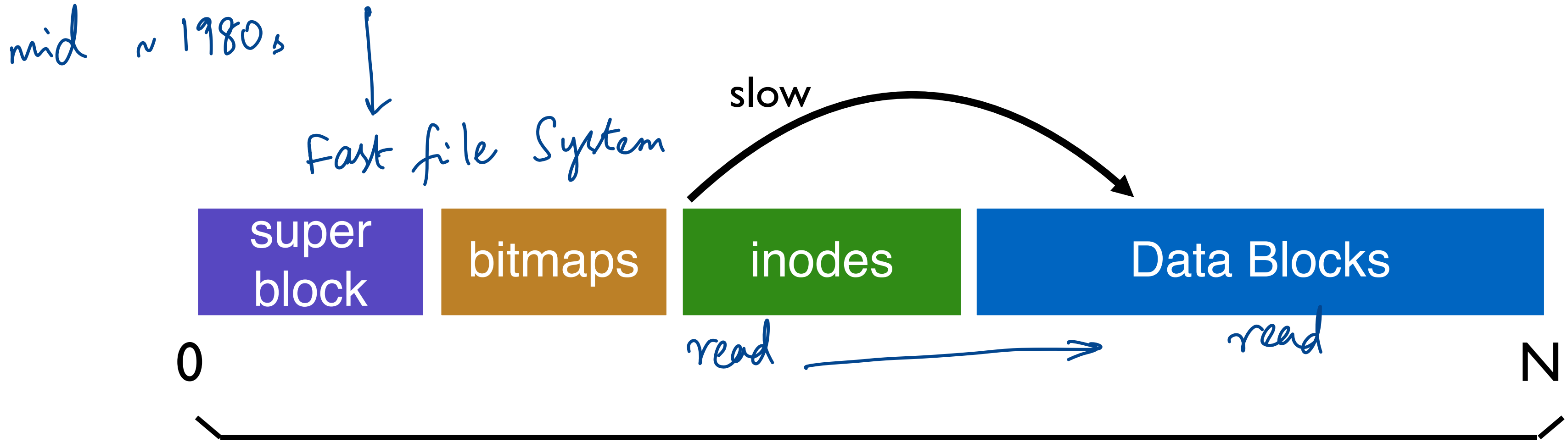
Shared structs (e.g., bitmaps+dirs) often overwritten.

Tradeoffs: how much to buffer, how long to buffe



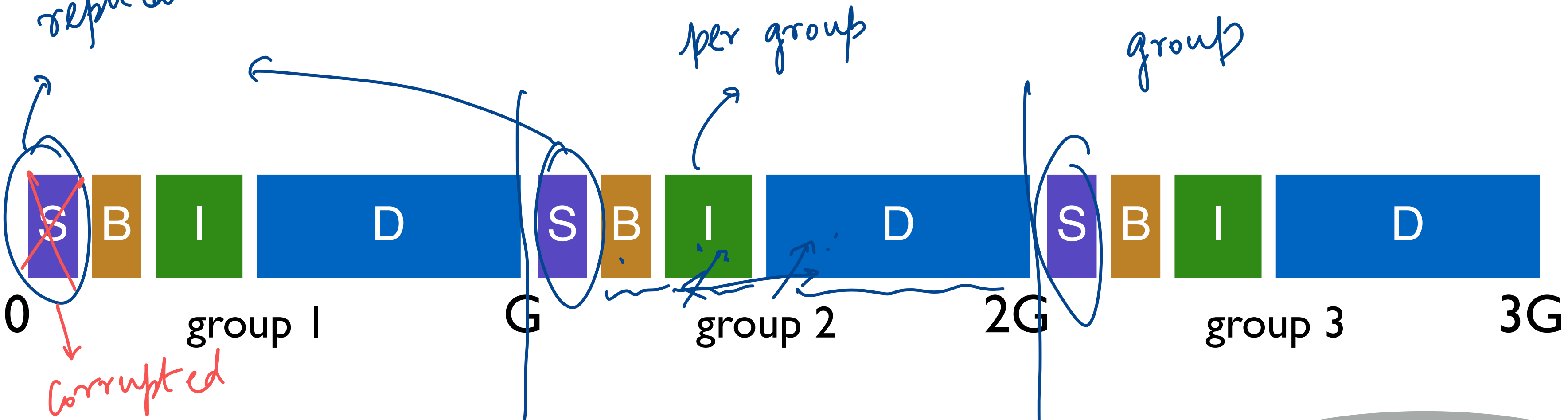
→ 1 disk write for 3 bits

FFS: FILE LAYOUT IMPORTANCE



Layout is not disk-aware!

PLACEMENT TECHNIQUE: GROUPS

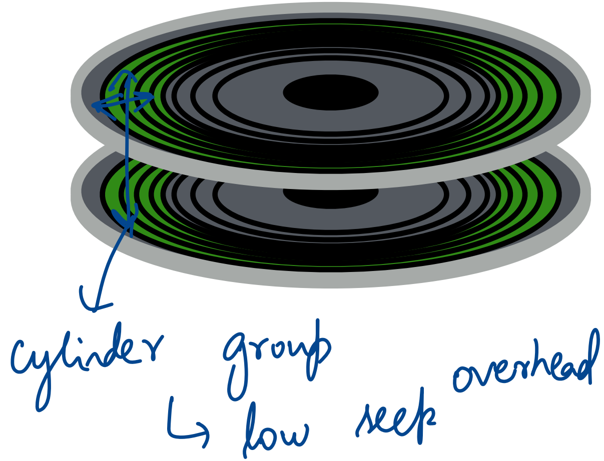


Key idea: Keep inode close to data

Use groups across disks

Strategy: allocate inodes and data blocks in same group.

Replicated superblocks



REPLICATED SUPER BLOCKS

PLACEMENT STRATEGY

Put related pieces of data near each other.

Rules:

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

Problem: File system is one big tree

All directories and files have a common root.

All data in same FS is related in some way

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

REVISED STRATEGY

Put more-related pieces of data near each other

Put less-related pieces of data **far**

```
/a/b
/a/c
/a/d
/b/f
```

```
group inodes     data
    0 /--------- /---------
    1 acde------ accddee---
    2 bf-------- bff-------
    3 ---------- ----------
    4 ---------- ----------
    5 ---------- ----------
    6 ---------- ----------
    7 ---------- ----------
  ...
```

POLICY SUMMARY

File inodes: allocate in same group with dir

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

First data block: allocate near inode

Other data blocks: allocate near previous block

PROBLEM: LARGE FILES

Single large file can fill nearly all of a group

Displaces data for many small files

```
group inodes     data
    0 /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
    1 ---------- ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ---------- ----------
    ...
```

Most files are small!

Better to do one seek for large file than

one seek for each of many small files

SPLITTING LARGE FILES

```
group inodes      data
    0 /a--------- /aaaaa---- ---------- ---------- ----------
    1 ----------- aaaaa----- ---------- ---------- ----------
    2 ----------- aaaaa----- ---------- ---------- ----------
    3 ----------- aaaaa----- ---------- ---------- ----------
    4 ----------- aaaaa----- ---------- ---------- ----------
    5 ----------- aaaaa----- ---------- ---------- ----------
    6 ----------- ---------- ---------- ---------- ----------
  ...
```

Define "large" as requiring an indirect block

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

Each chunk corresponds to one indirect block
Block size 4KB, 4 byte per address => 1024 address per indirect
1024*4KB = 4MB contiguous "chunk"

POLICY SUMMARY

File inodes: allocate in same group with dir

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

First data block: allocate near inode

Other data blocks: allocate near previous block

Large file data blocks: after 48KB, go to new group.

Move to another group (w/ fewer than avg blocks) every subsequent 4MB.

NEXT STEPS

Next class: Journalling