PERSISTENCE: FAST FILE SYSTEM

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

2 weeks from now

P5 release tonight, due April 23rd, 10pm

Attend the discussion section for more details!

optional project after P5?

AGENDA / LEARNING OUTCOMES

How does file system represent files, directories?

What steps must reads/writes take?

Very simple file system

How does FFS improve performance?

RECAP

FILE API WITH 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)
```

Persistence

↳ Devices

↓

Hard disks

↳ RAID

File API to access data on disks

advantages:

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

/root/test.txt

FILE API SUMMARY

Using multiple types of name provides convenience and efficiency

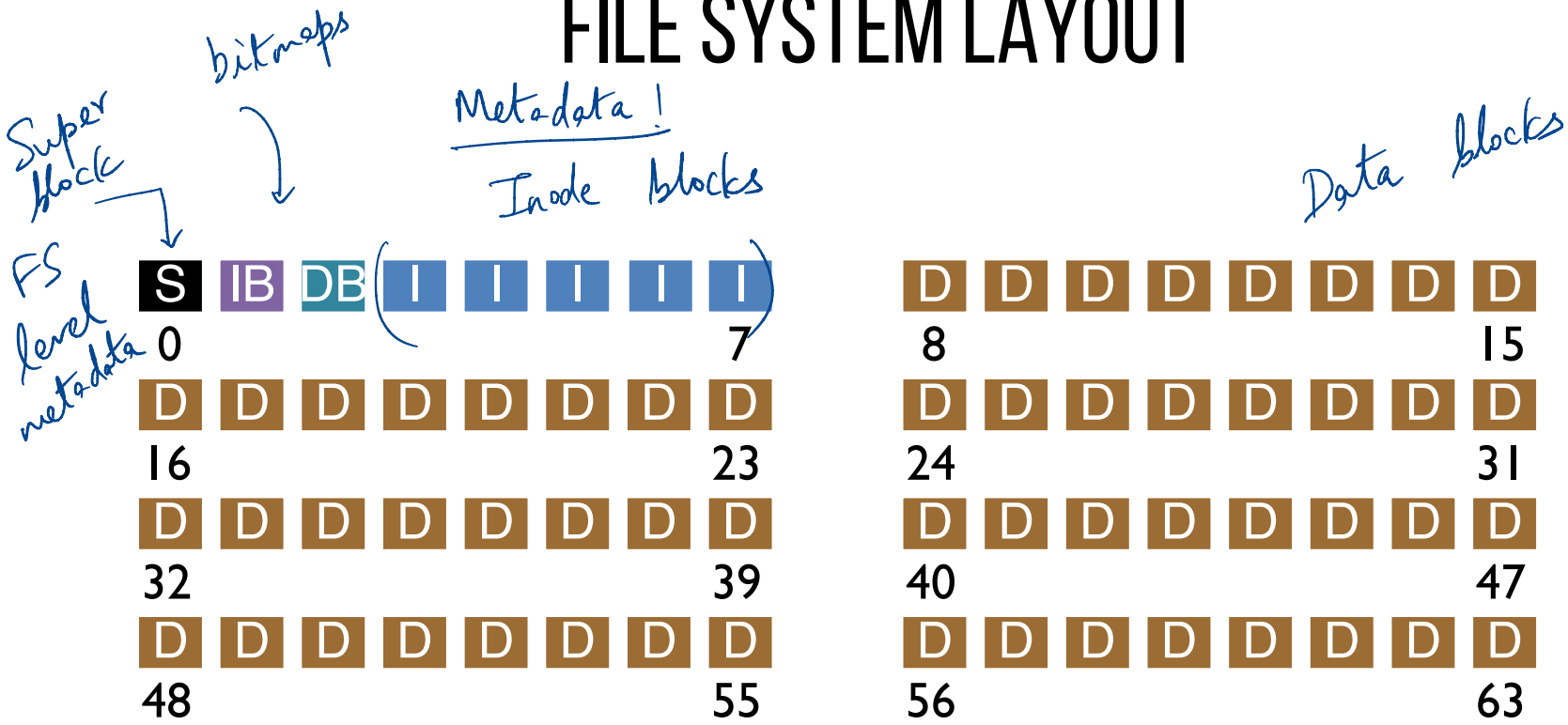
Hard and soft link features provide flexibility.

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

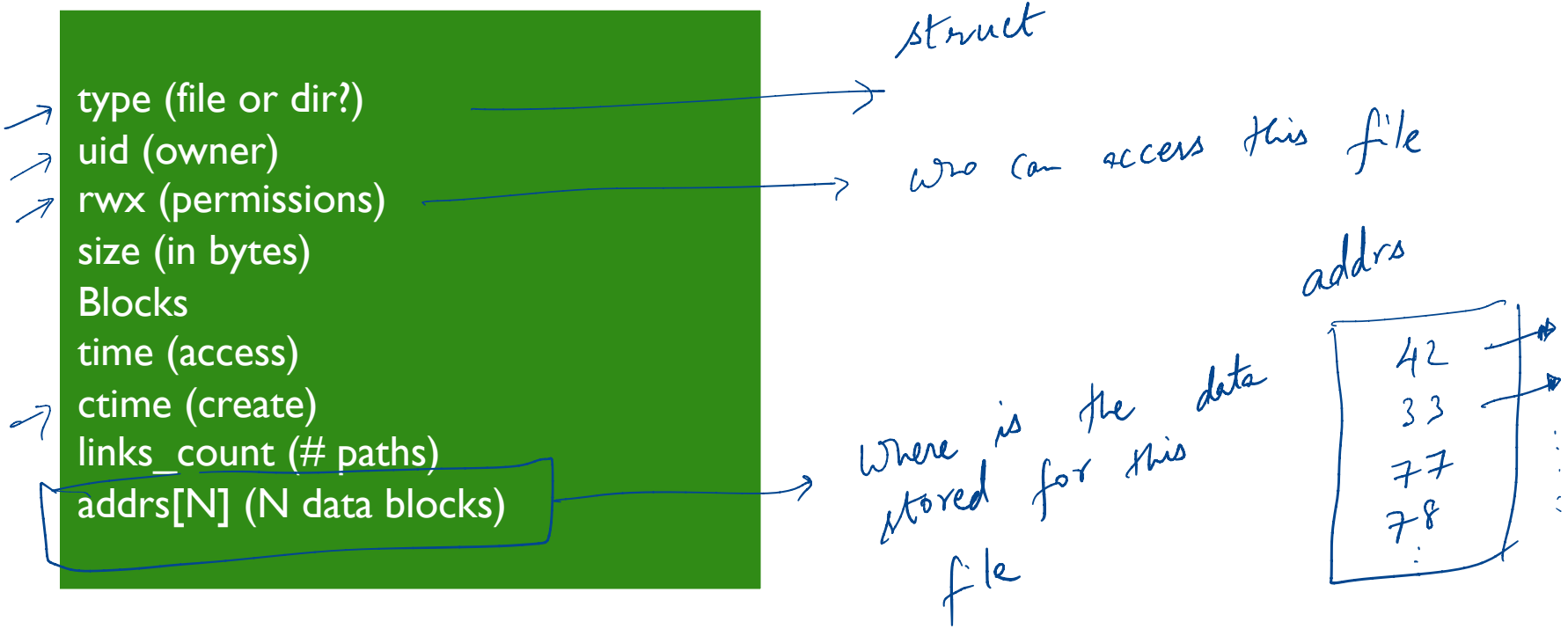
flushing data to disk

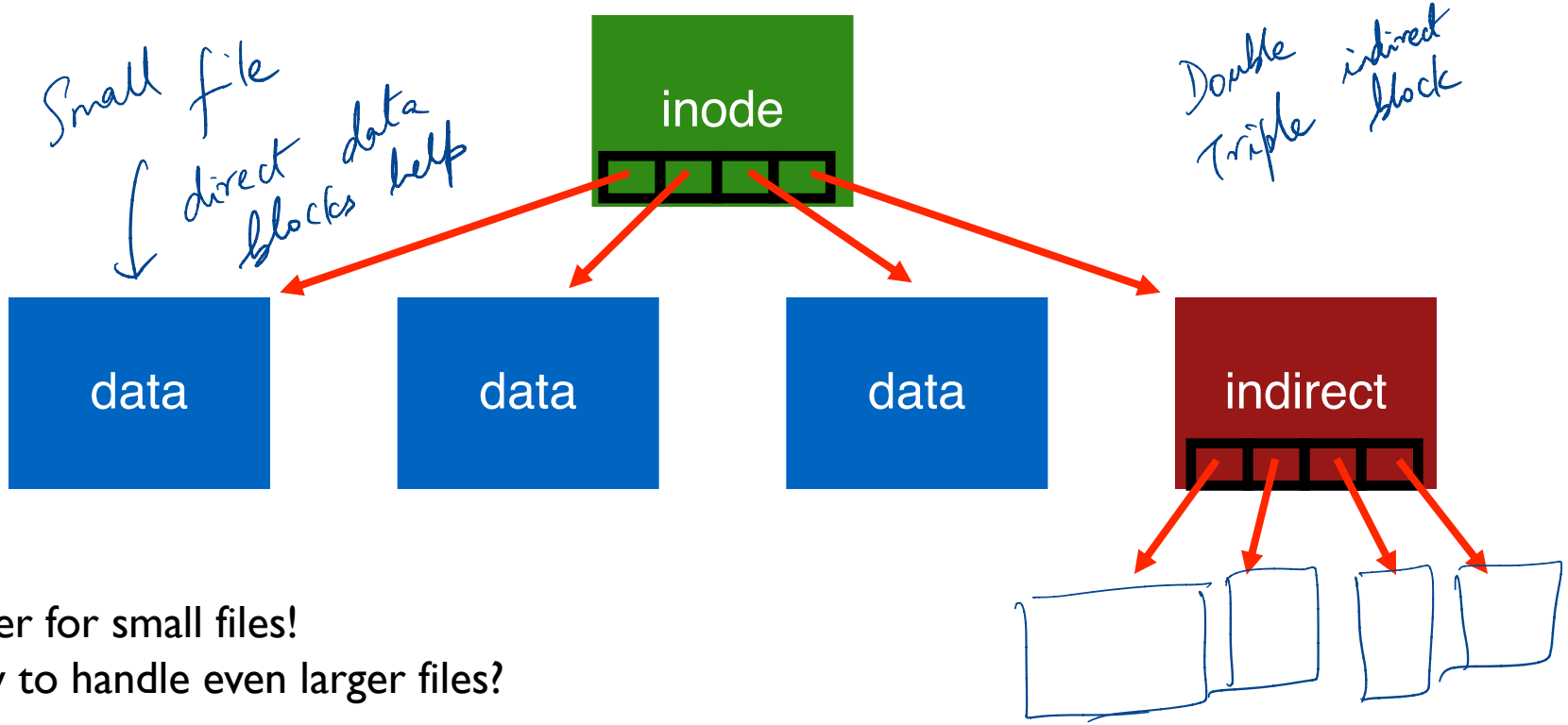
atomically moving a file from one path to another

FILE SYSTEM LAYOUT



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)
struct
who can access this file
Where is the data stored for this file
addrs
42
33
77
78



Better for small files!

How to handle even larger files?

SIMPLE DIRECTORY LIST EXAMPLE

Directories are stored in data blocks

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

unlink("foo")

"struct" {
name
inode_num
valid
}

FS OPERATIONS

FILE API

- open
- read
- close
- create file
- write

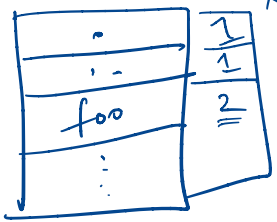
TIME

open /foo/bar (already exists)

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

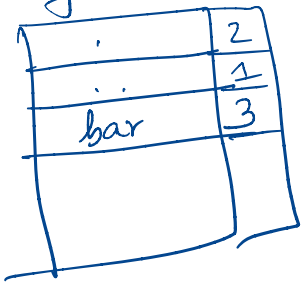
Root inode is in a well known location

/

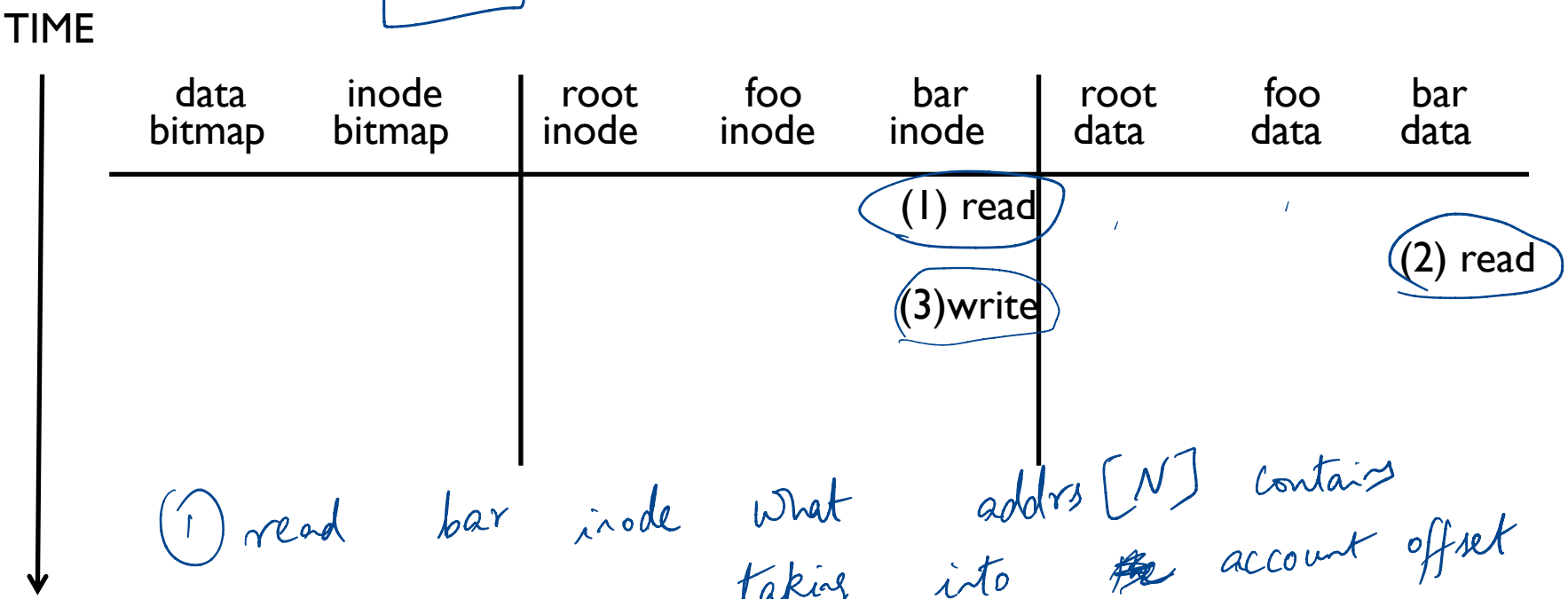


Can this user read this file?

foo



read /foo/bar – assume opened



(1) read bar inode what addrs [N] contains taking into account offset

(2) read data block address

(3) last accessed time for a file, updated after reading data

close /foo/bar

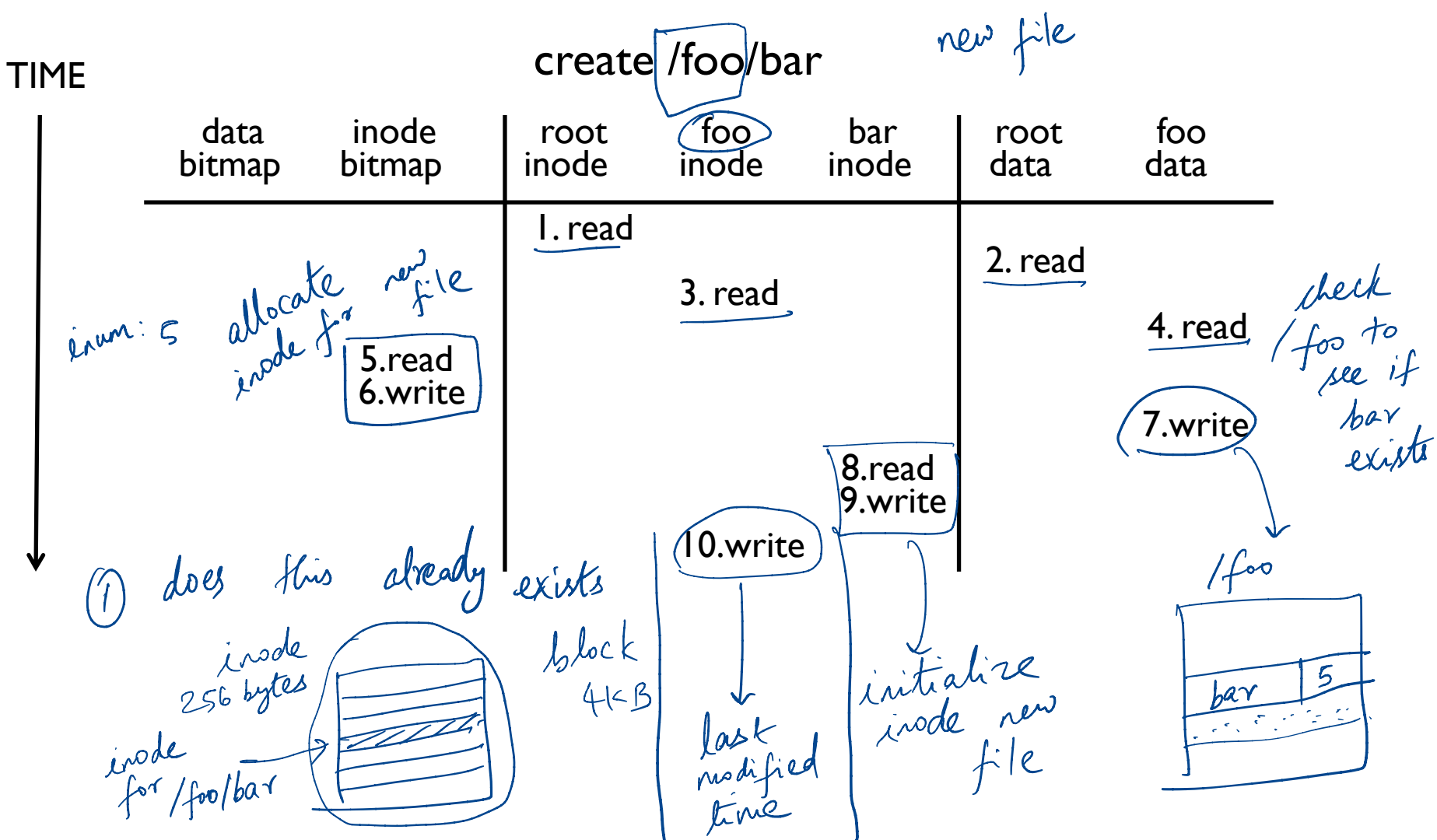
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

nothing to do on disk!

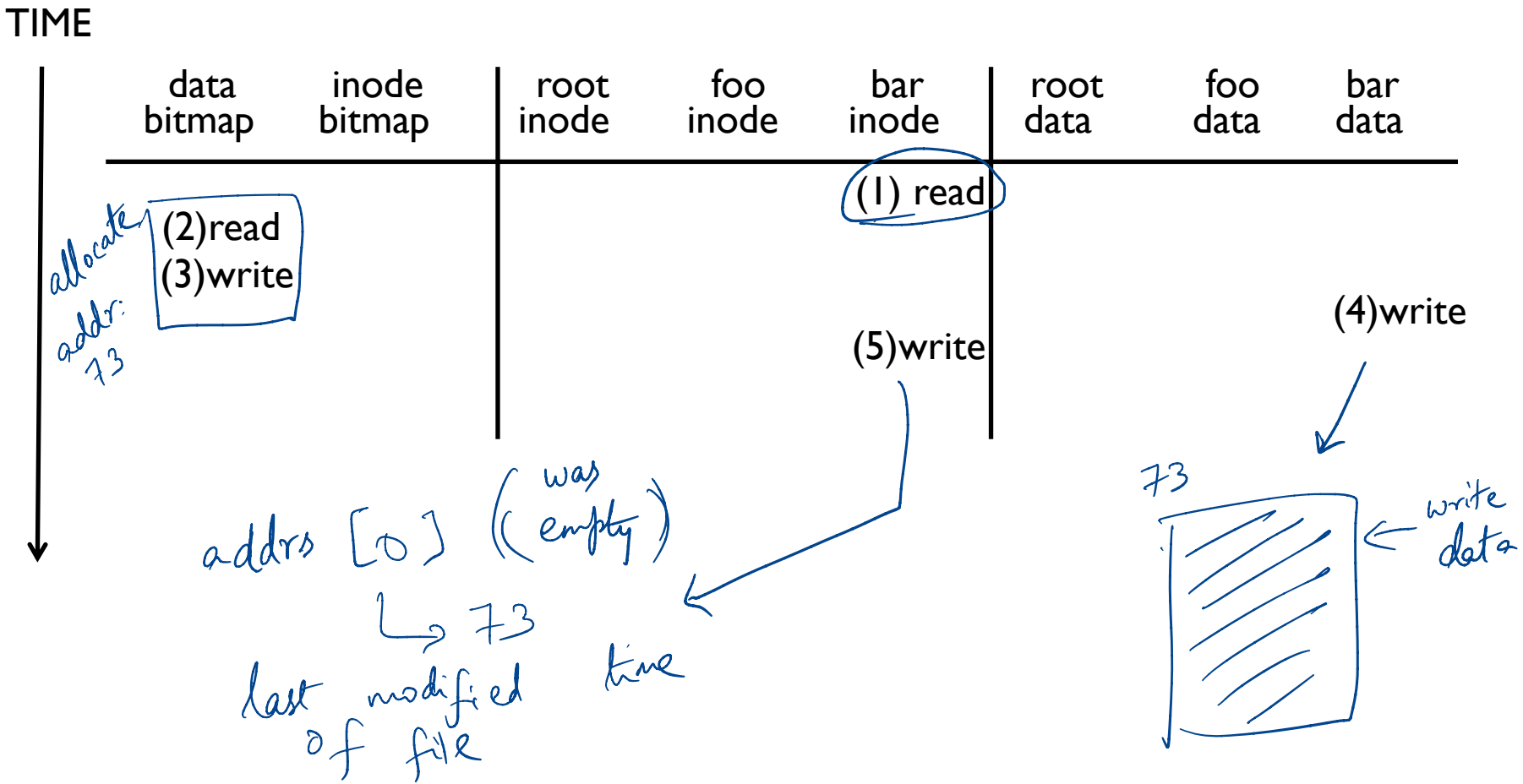
open — traverse the path

read — inode → data

close — no operations


TIME
create /foo/bar
new file
data bitmap
inode bitmap
root inode
foo inode
bar inode
root data
foo data
1. read
2. read
3. read
4. read
check /foo to see if bar exists
inum: 5
allocate inode for new file
5.read
6.write
7.write
8.read
9.write
10.write
/foo
bar
5
① does this already exists
inode 256 bytes
inode for /foo/bar
block 4KB
last modified time
initialize inode new file

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



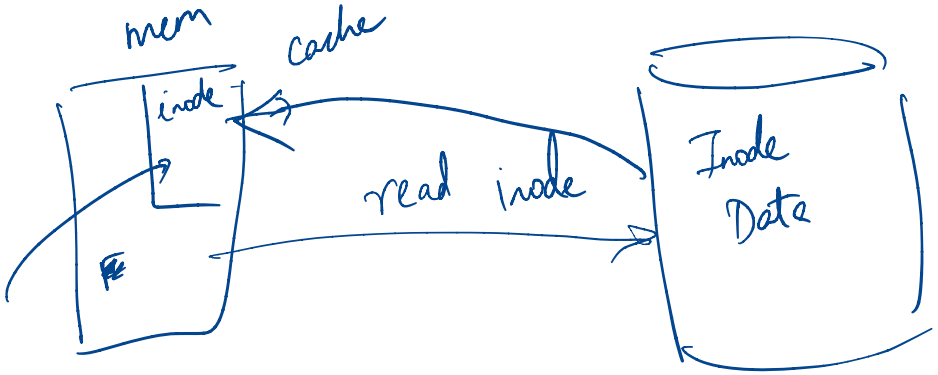
EFFICIENCY

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

Cache for:

- reads
- write buffering

Periodically or on fsync



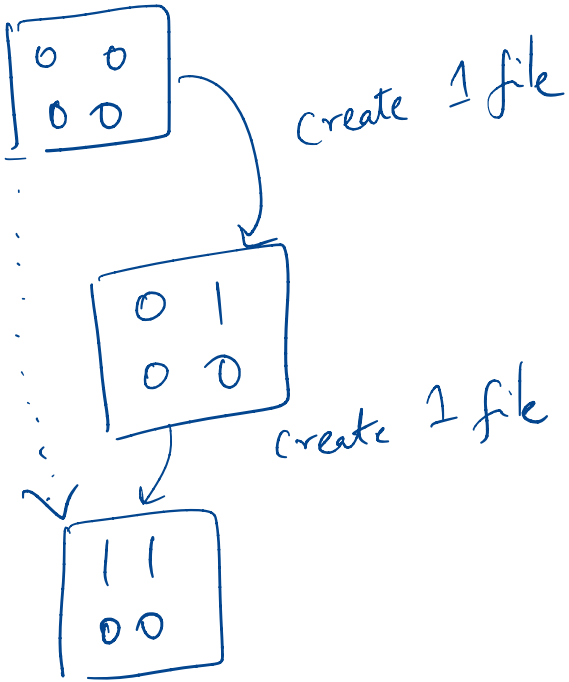
WRITE BUFFERING

Overwrites, deletes, scheduling

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

Tradeoffs: how much to buffer, how long to buffer

configurable by users



QUIZ 26

https://tinyurl.com/cs537-sp20-quiz26

a: address , r: ref. count

```
inode bitmap   ????????
inodes         [d a:0 r:3] [d a:1 r:2] [] [] [] [] [] []
data bitmap    11000000
data           [(.,0) (..,0) (n,1)] [(.,1) (..,0)] [] [] [] [] [] []
```

mkdir ('/n')

11000000 ⟶ Two inodes are used so two bits should be 1

Create(/2)

```
inode bitmap   11000000
inodes         [d a:0 r:2] [f a:- r:1] [] [] [] [] [] []
data bitmap    10000000
data           [CORRUPT!] [] [] [] [] [] [] []
```

1 inode number

directory

[(.,0) (..,0) (2,1)]

name inode number

name	inode num
2	1

QUIZ 26

https://tinyurl.com/cs537-sp20-quiz26

```
inode bitmap 11100000
inodes       [d a:0 r:4] [d a:1 r:2] [d a:2 r:2] [] [] [] [] []
data bitmap  11100000
data         [(.,0) (..,0) (d,1) (w,2)] [????] [(.,2) (..,0)] [] [] [] [] []
```

/ → data block 1

mkdir(/d) mkdir(/w)

[(.,1) (..,0)] = /d

(.,1): 1 = parent dir

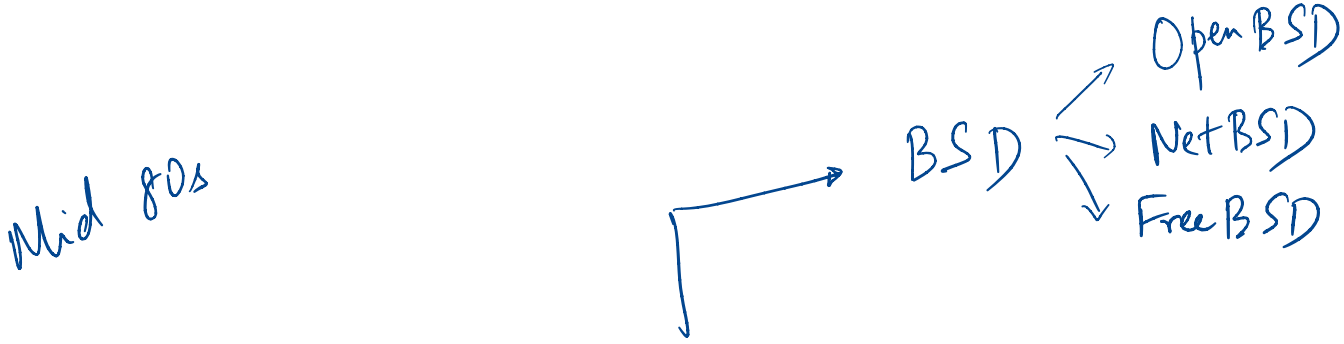
(..,0): 0 = inode number of parent dir (/)

```
inode bitmap  11000000
inodes        [d a:0 r:2] [f a:1 r:2] [] [] [] [] [] []
data bitmap   ????????
data          [(.,0) (..,0) (c,1) (m,1)] [foofoofoo] [] [] [] [] [] []
```

(1) c & m refer to same inode ✓ hard link

(2) First block has foofoofoo

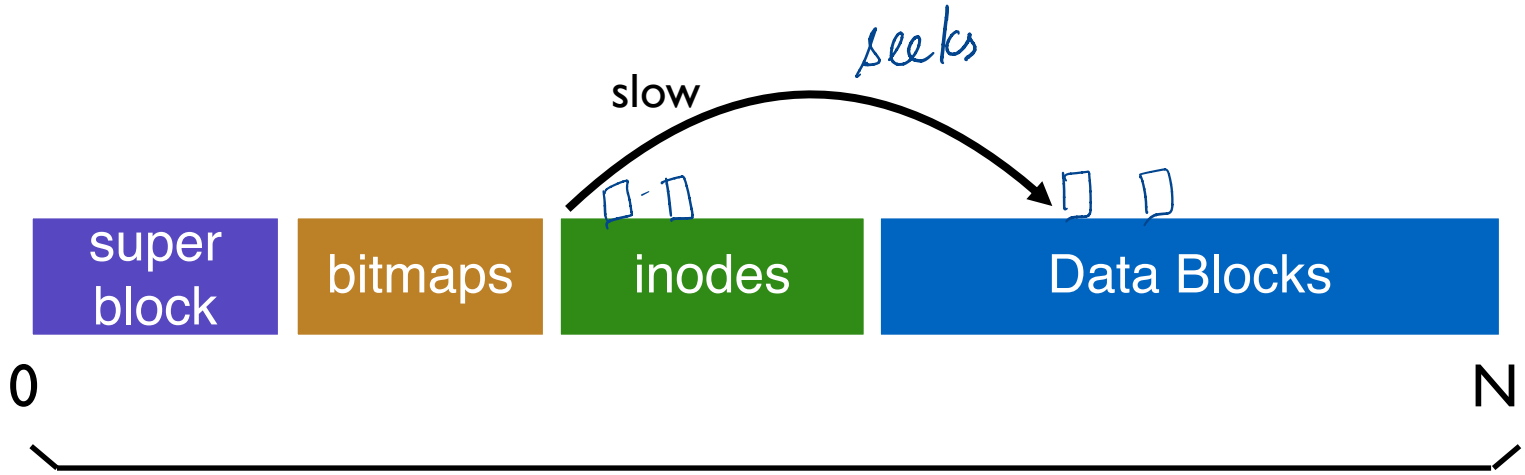
(3) Proper root dir ✓



FAST FILE SYSTEM

FFS

FILE LAYOUT IMPORTANCE



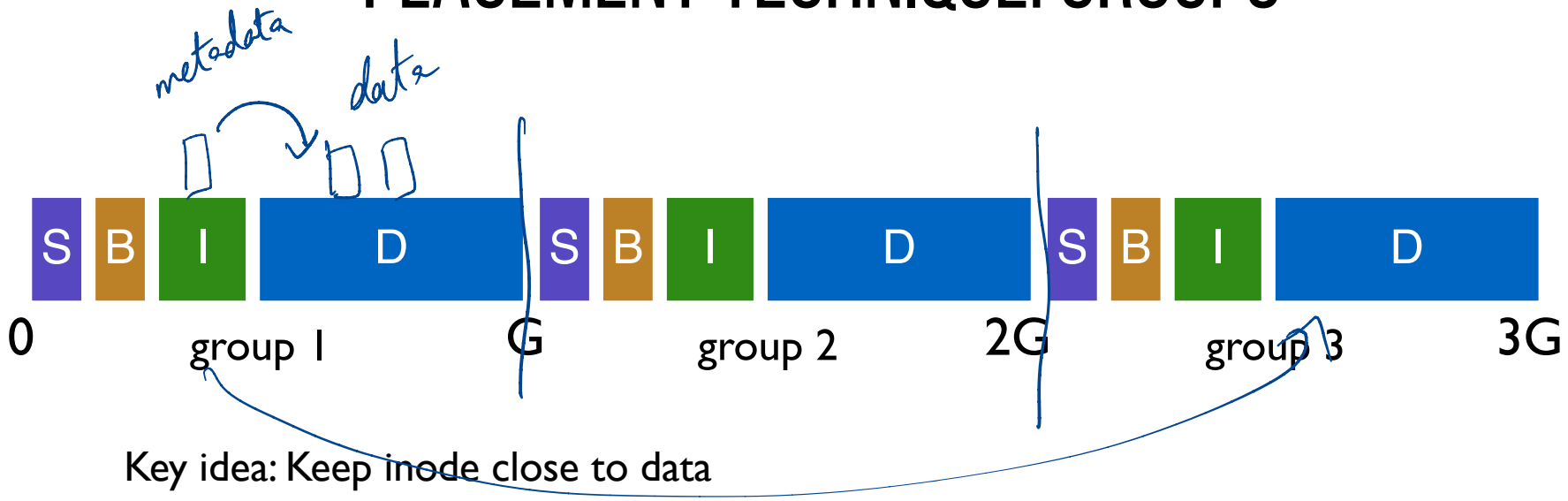
Layout is not disk-aware!

DISK-AWARE FILE SYSTEM

How to make the disk use more efficient?

Where to place meta-data and data on disk?

PLACEMENT TECHNIQUE: GROUPS



Key idea: Keep inode close to data

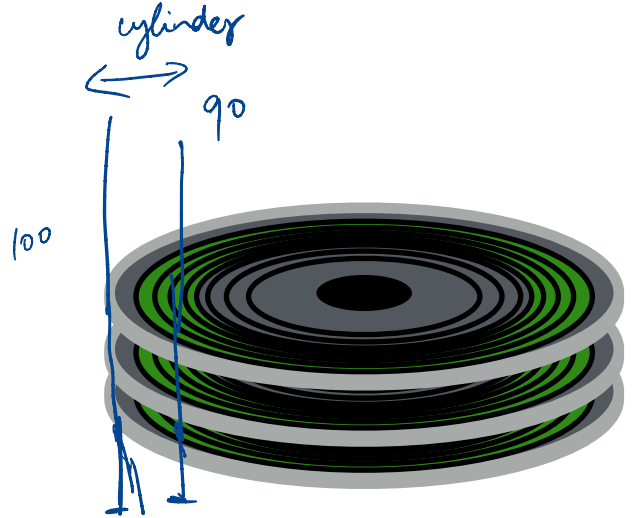
Use groups across disks;

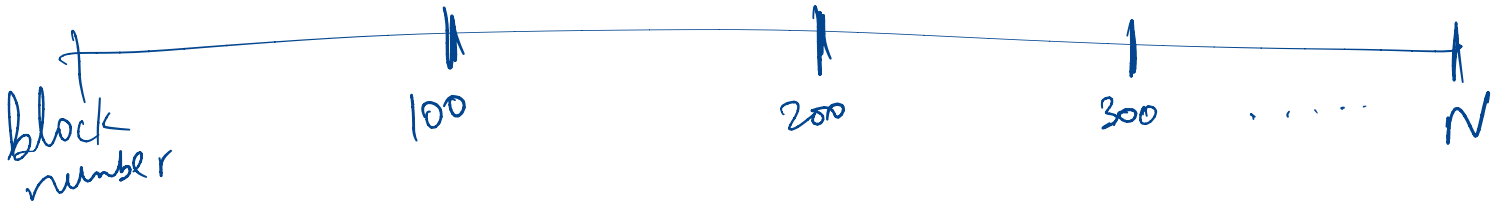
Strategy: allocate inodes and data blocks in same group.

PLACEMENT TECHNIQUE: GROUPS

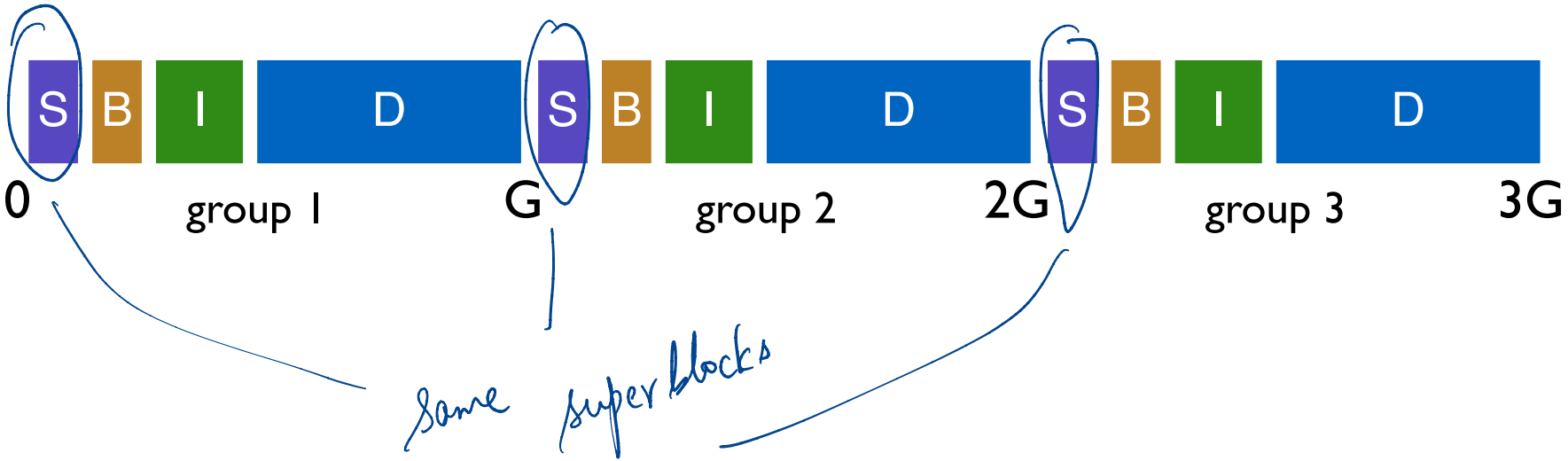
In FFS, groups were ranges of cylinders
called cylinder group

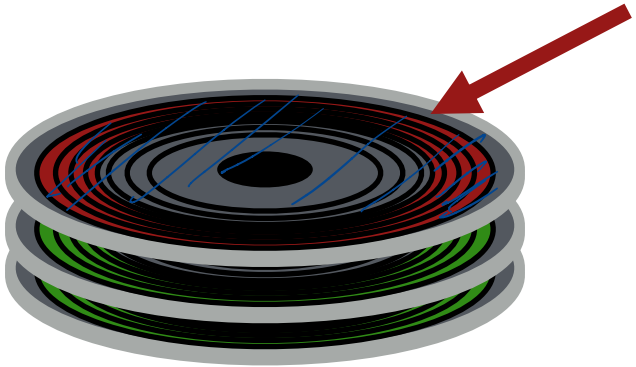
In ext2, ext3, ext4 groups are ranges of blocks
called block group ≡ SSTF





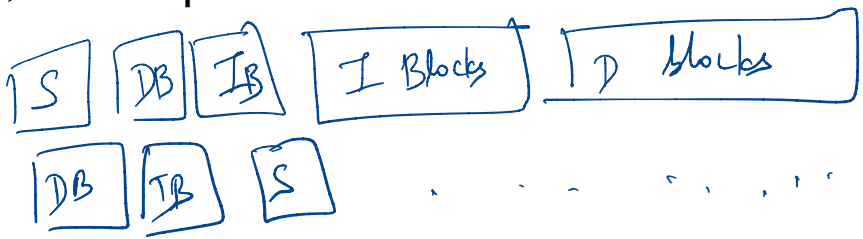
REPLICATED SUPER BLOCKS





top platter damage?

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



SMART POLICY

Where should new inodes and data blocks go?

Policy

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. for files

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

Compiling code

src/a.c
/a.h
/b.c

Put more-related pieces of data near each other

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

ls -R /

/a/e
/a/c
/a/d
/b/f

	Group
mkdir (/)	0
mkdir (/a)	1
Create (/a/e)	1
Create (/a/c)	1
⋮	⋮
mkdir (/b)	2

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

parent

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

load balancing

First data block: allocate near inode

Other data blocks: allocate near previous block

PROBLEM: LARGE FILES

Most files are small

Most of the space is used by large file

Single large file can fill nearly all of a group

Displaces data for many small files

create /b

create /c

then we might not have space in this group

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

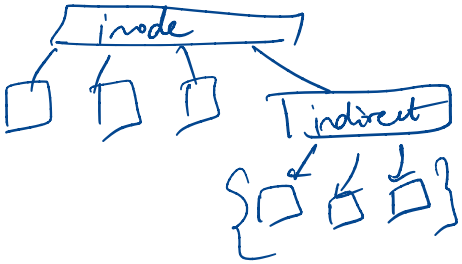
bbbbbcc

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

Amortization

1 seek
↳ read 4MB contiguously

SPLITTING LARGE FILES




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

4MB
4MB

12 direct
4 KB

Define "large" as requiring an indirect block

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

4KB block
addr 4 bytes

Each chunk corresponds to one indirect block

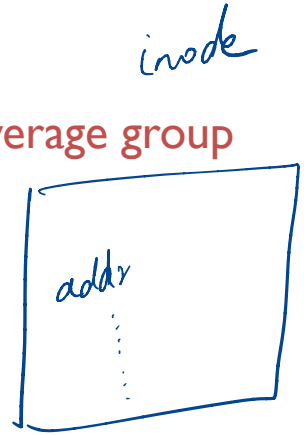
Block size 4KB, 4 byte per address => 1024 address per indirect

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 1MB.

OTHER FFS FEATURES

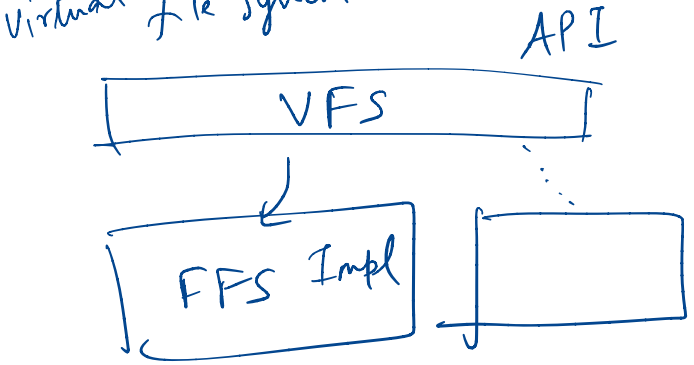
FFS also introduced several new features:

- large blocks (with libc buffering / fragments)
- long file names
- atomic rename
- symbolic links

usability

standardize

virtual file system



FFS SUMMARY

First disk-aware file system

- Bitmaps
- Locality groups
- Rotated superblocks
- Smart allocation policy

Inspired modern files systems, including ext2 and ext3

NEXT STEPS

P5 will be released later today
Details in the discussion section

Next class: Filesystem consistency