PERSISTENCE: FAST FILE SYSTEM

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

Midterm grades
- Solutions with explanation
- Next 1 or 2 days

P6 progress?
- Test cases? → Update by end of day
- Two equal keys: does order matter? Not?

AGENDA / LEARNING OUTCOMES

How does file system represent files, directories?

What steps must reads/writes take?

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)
```

advantages:

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

FILE SYSTEM LAYOUT

FS level metadata

block size
number of inodes

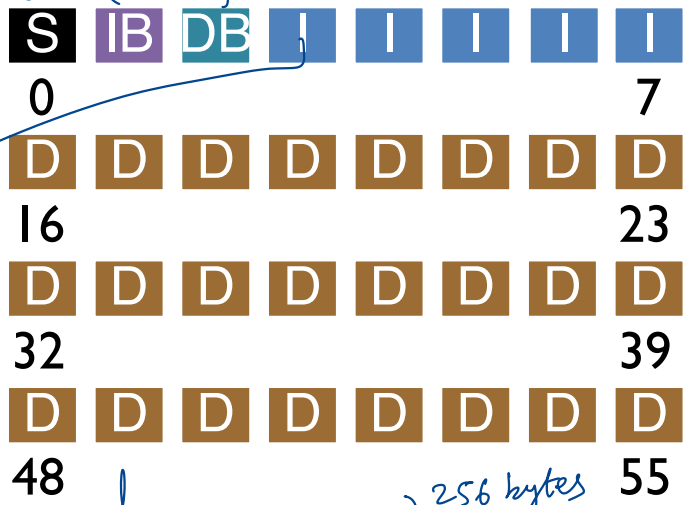
Bitmaps used for allocation

inodes → metadata, access time, permissions, ptrs to data blocks

Data Blocks

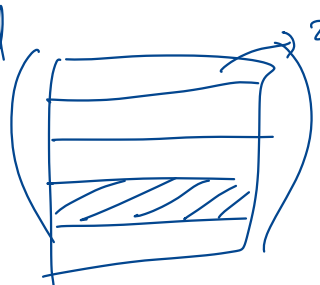
store file contents

store directories



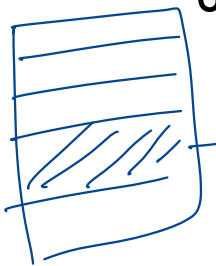
read

256 bytes

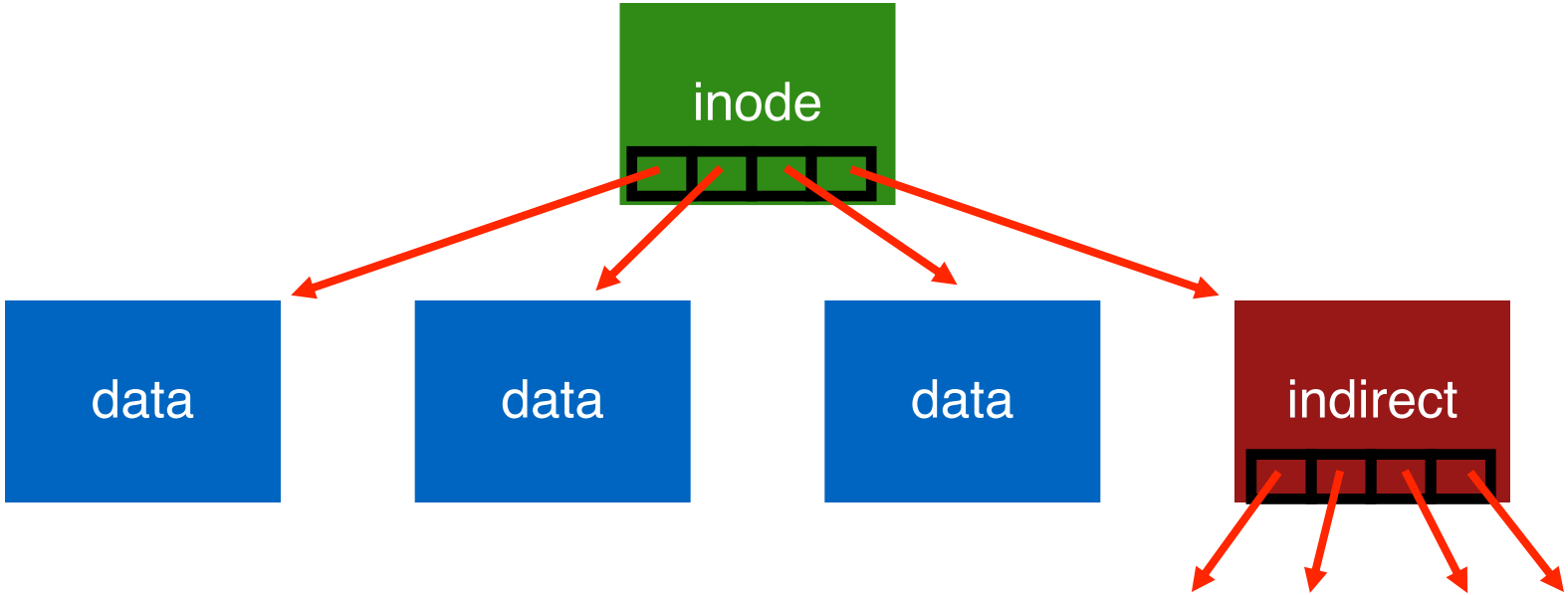


4K size block →

write



new content


inode
data
data
data
indirect

FS OPERATIONS

- open
- read
- close
- create file
- write

What does the FS need to do on disk?

TIME

open /foo/bar

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	

① Check if file exists

② Check if user has permissions

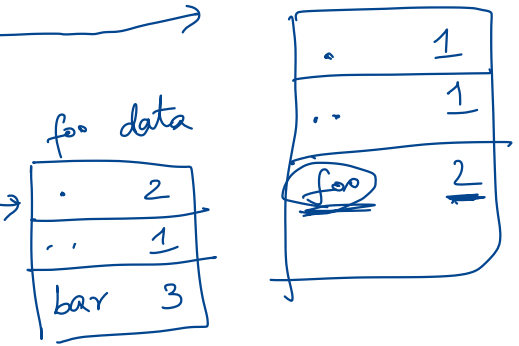
① read root inode

③ read foo inode

⑤ read bar inode

check permissions

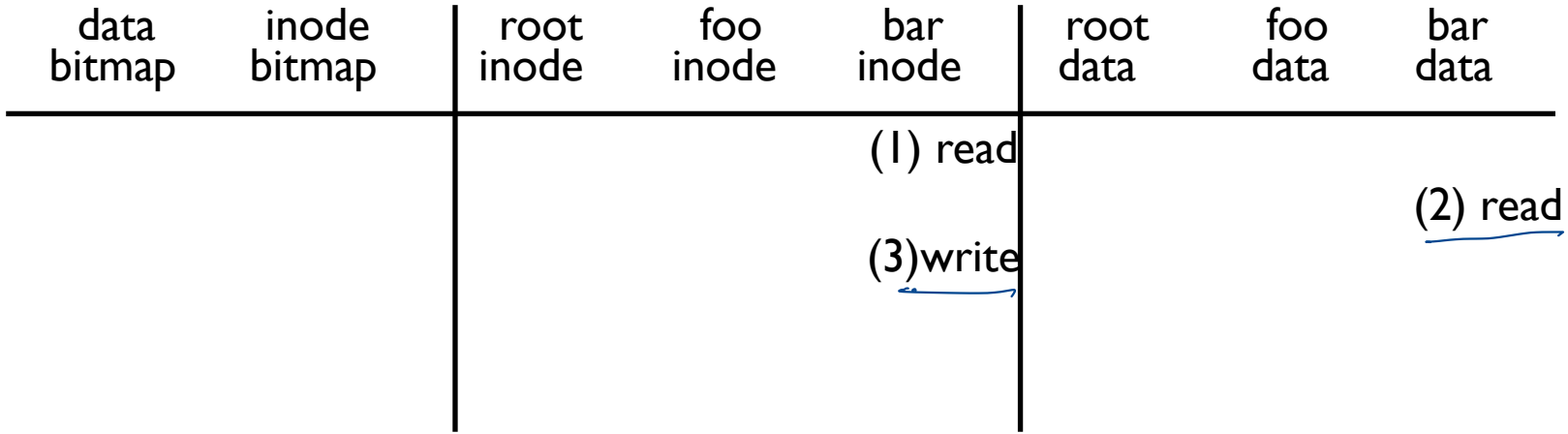
② root data block



read /foo/bar – assume opened

→ OS has a FD (offset) for this file

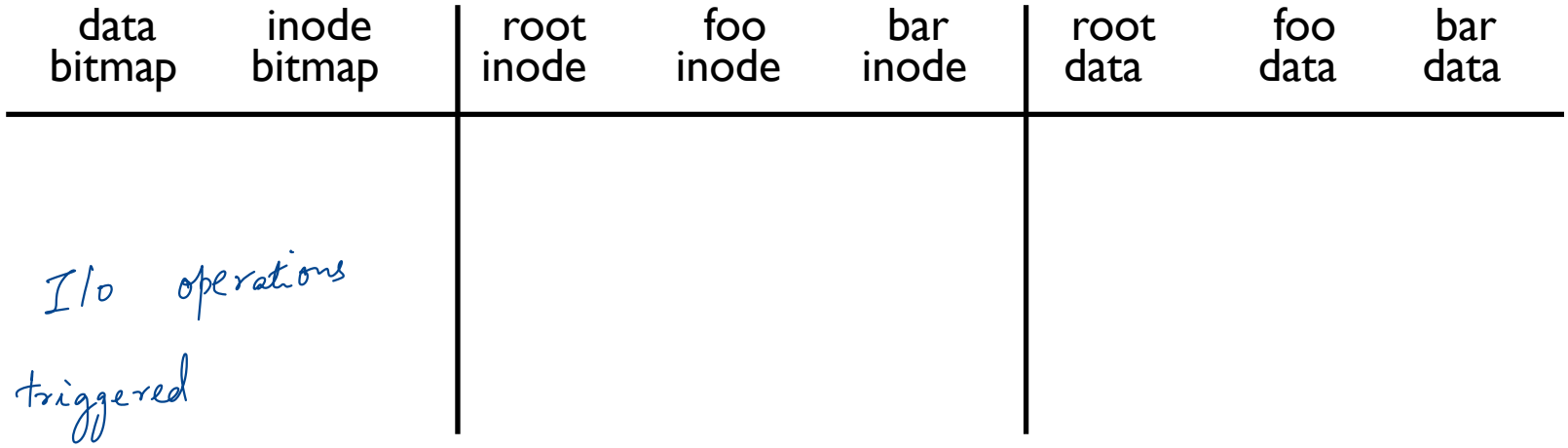
TIME



read : start offset, read N bytes

1. read inode : figure out which data blocks to read
2. read data blocks calculated in prev step
3. Update the last accessed time for this file

close /foo/bar

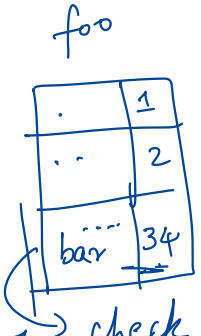


nothing to do on disk!

TIME

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
	5.read 6.write	1. read	3. read	8.read 9.write	2. read	4. read
			10.write			7.write



modification time

1. Check if a file with same name exists → Perform traversal and read directory entry for foo
2. Allocate inode for file: read IB & write IB
3. Initialize inode → set permissions who created, what time

read inode block
write inode block

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

offset based → (start, length)

TIME ↓

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				(1) read			
(2)read (3)write → allocates new data block							(4)write
				(5)write			

① Read inode to figure out whether I need new data block

② Allocate DB

③ Write to data block

④ Update inode to contain ptr to new data block and metadata

EFFICIENCY

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

Cache for:

- reads
- write buffering

Buffer cache → Commonly used inodes / might be in the cache

"LRU"

WRITE BUFFERING

temporary files

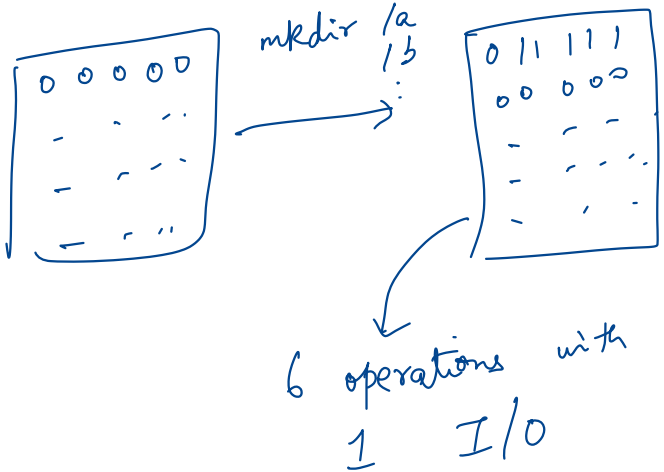
Overwrites, deletes, scheduling

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

Tradeoffs: how much to buffer, how long to buffer

Crash consistency

Inode Bitmap
Data Bitmap
→ Every create only changes 1 bit



QUIZ 26

https://tinyurl.com/cs537-sp23-quiz26

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

directory, ref count, data block

11000000

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

second entry

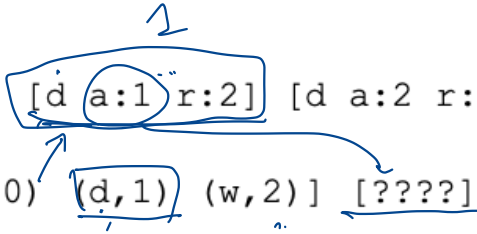
.	0
..	0
z	1

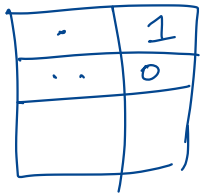
Creat ("/z")

QUIZ 26

https://tinyurl.com/cs537-sp23-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)] [] [] [] [] []
```





/
/d
/w

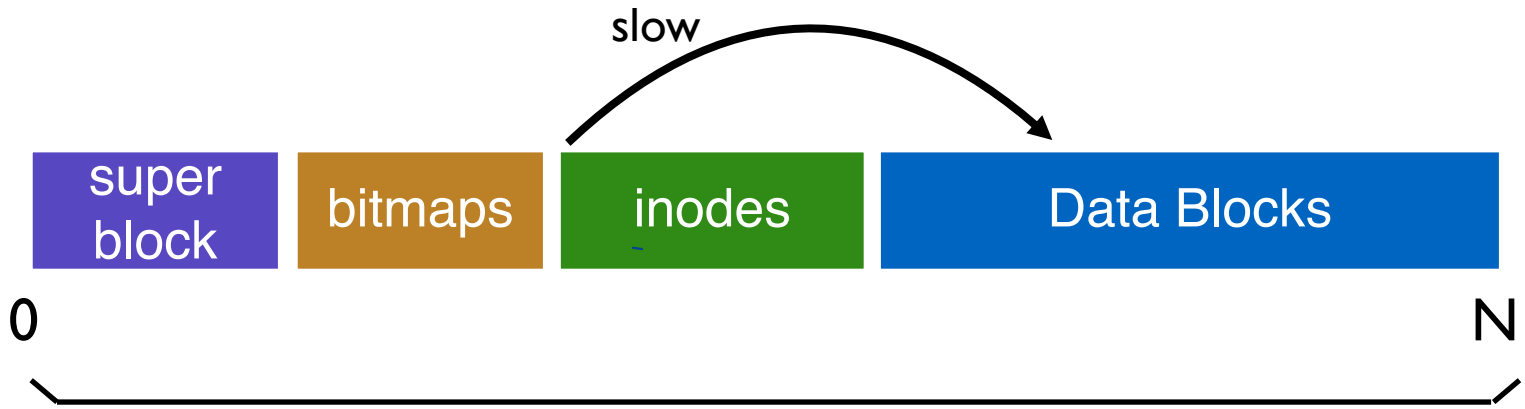
mkdir (/d)
mkdir (/w)

```
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. Both /c and /m are links to same file
3. Proper root directory
4. First block has foofoofoo

FAST FILE SYSTEM

FILE LAYOUT IMPORTANCE



Layout is not disk-aware!

DISK-AWARE FILE SYSTEM

How to make the disk use more efficient?

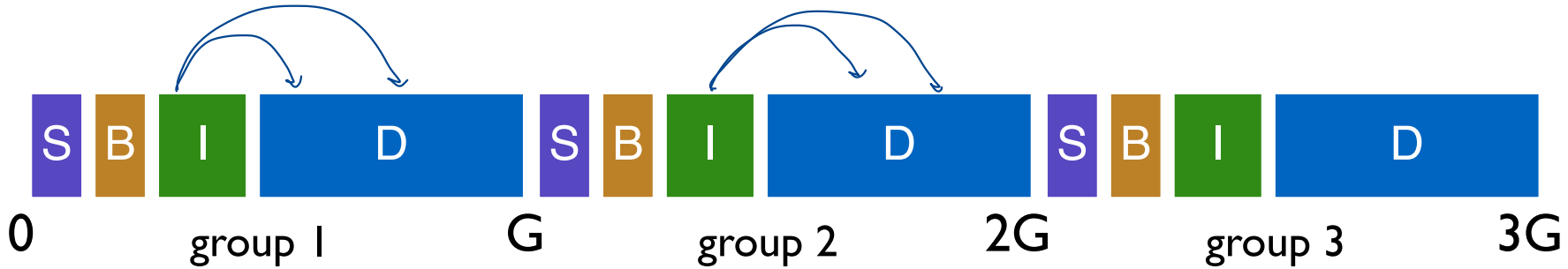
Where to place meta-data and data on disk?

inode blocks

data blocks

FFS → Fast File System

PLACEMENT TECHNIQUE: GROUPS



Key idea: Keep inode close to data

Use groups across disks;

Strategy: allocate inodes and data blocks in same group.

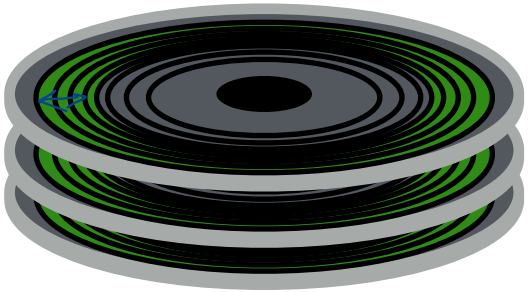
for files/directories

PLACEMENT TECHNIQUE: GROUPS

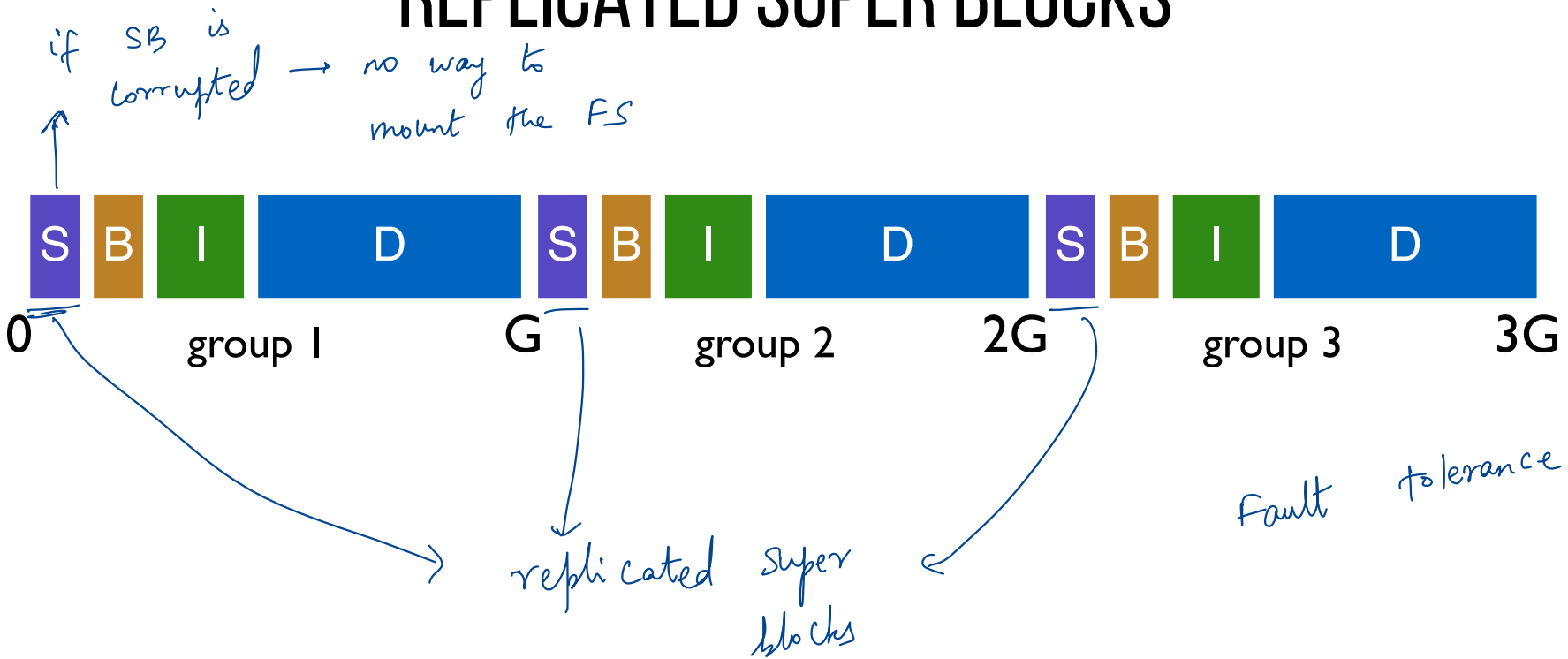
In FFS, groups were ranges of cylinders
called cylinder group

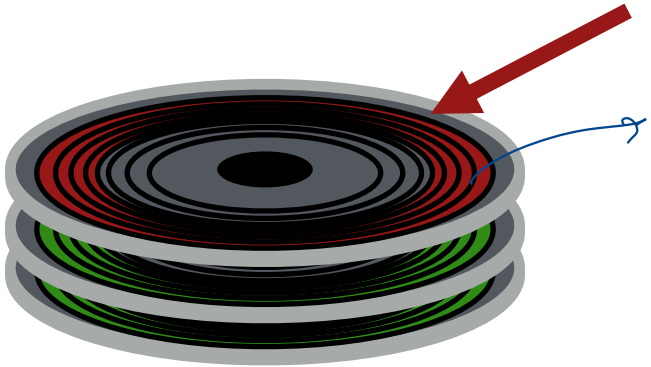
In ext2, ext3, ext4 groups are ranges of blocks
called block group

inodes and data blocks would be in nearby cylinders



REPLICATED SUPER BLOCKS





top platter damage?

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

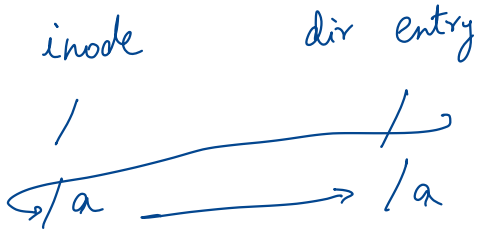
to overcome all replicated SBs becoming unavailable

SMART POLICY

Policy: Where should new inodes and data blocks go?

for files and directories

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.

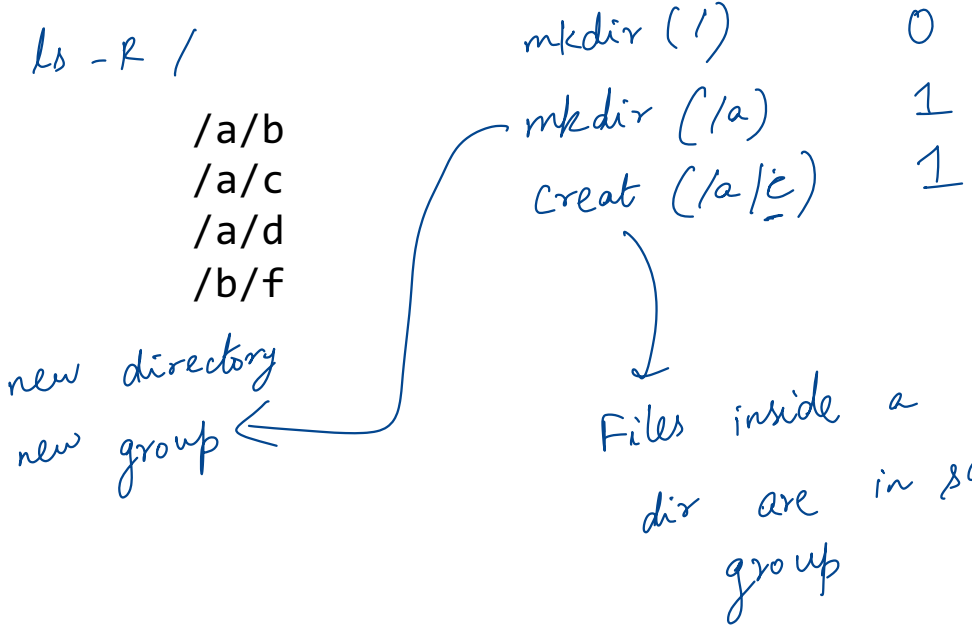
All data in same FS is related in some way

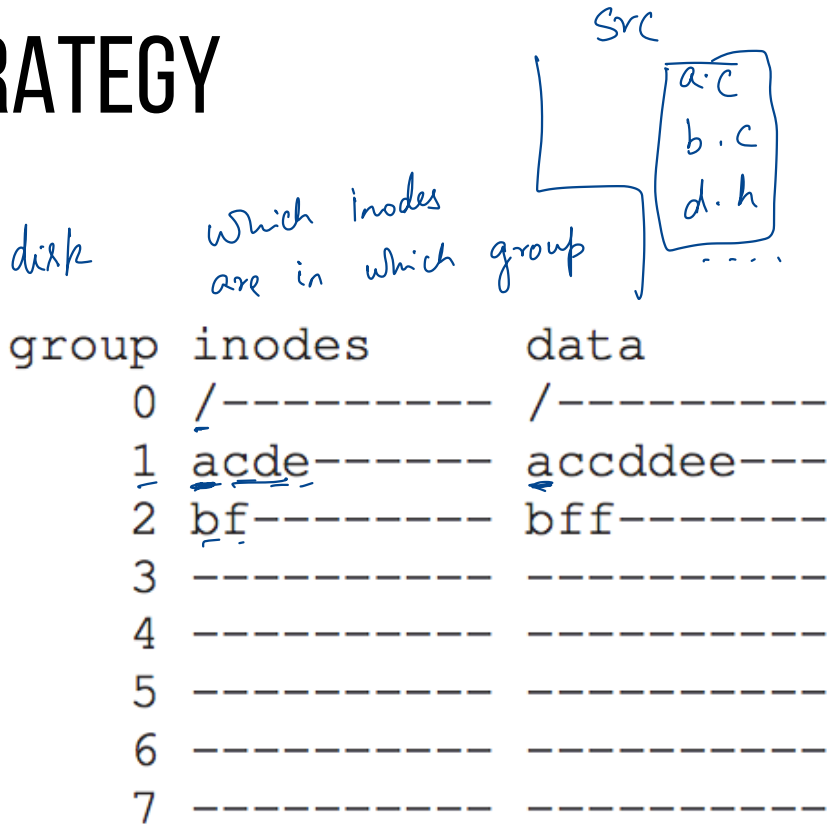
Trying to put everything near everything else doesn't make any choices!

REVISED STRATEGY

Put "more-related" pieces of data near each other

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





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

data in same grp as inode

Other data blocks: allocate near previous block

Most files are small

Most space is used by large files

PROBLEM: LARGE FILES

Big file → large number of data blocks

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!

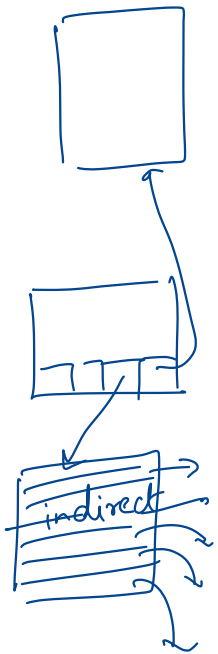
→ Better to do one seek for large file than one seek for each of many small files →

how compilers might access lots of small files

SPLITTING LARGE FILES

save space which can be used by small 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 1MB.

OTHER FFS FEATURES

FFS also introduced several new features:

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

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

Next class: Filesystem consistency