PERSISTENCE: FILE SYSTEMS & FFS

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

Project 4b: Due next week 4/16

Project 5: One project 9%. Updated due dates on website

Discussion this week: Review worksheet, More Q&A for 4b

AGENDA / LEARNING OUTCOMES

How does file system represent files, directories?

What steps must reads/writes take?

How does FFS improve performance?

Fast file system

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 — path that is passed to open
- hierarchical —
- traverse once — traverse
- offsets precisely defined

FILE, DIRECTORY API SUMMARY

Using multiple types of name provides convenience and efficiency

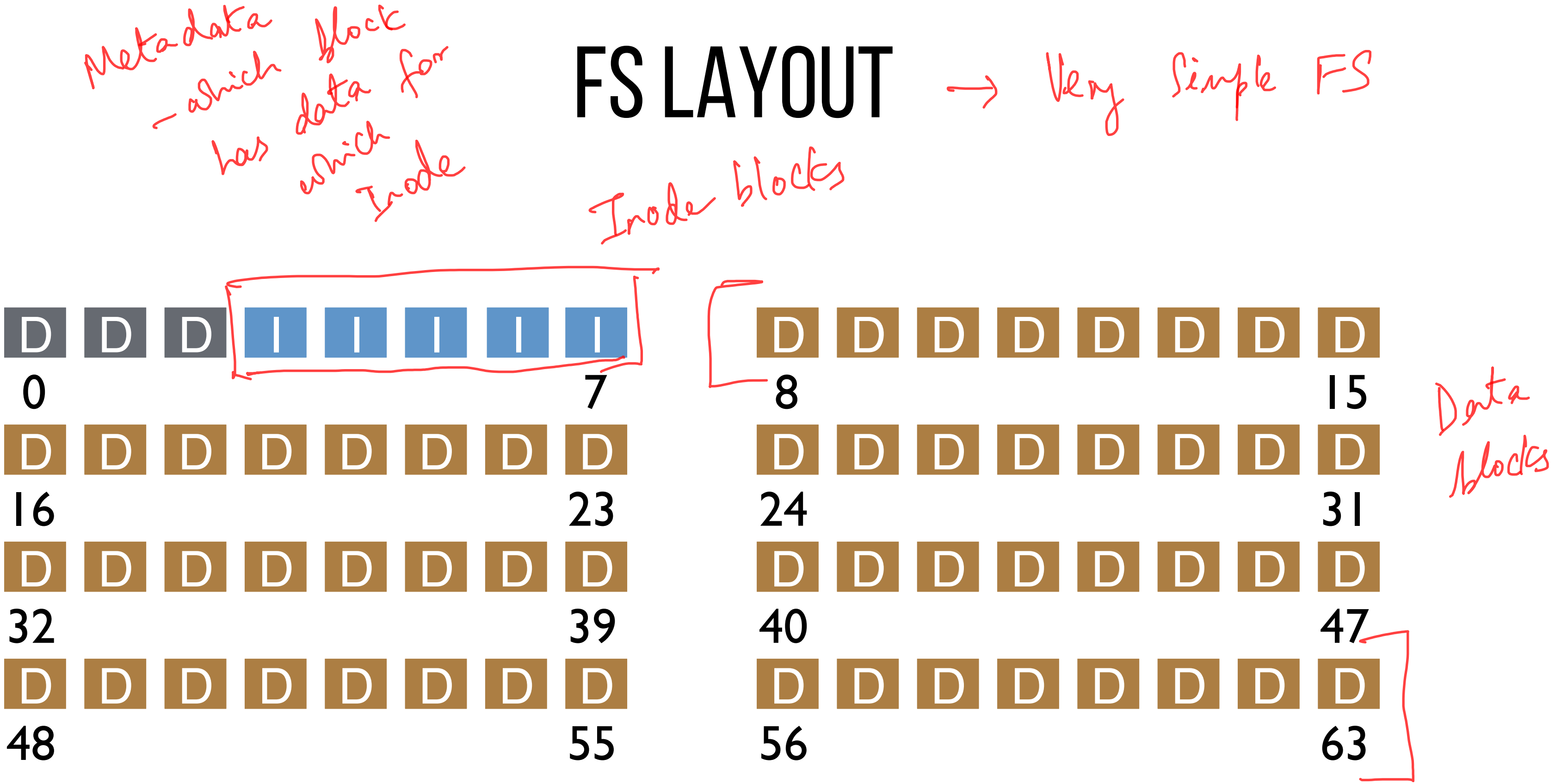
Mount and link features provide flexibility.

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

atomic writes using fsync, rename(old_name, new_name)
↳ atomicity

cp file.txt file.txt.tmp
<operate on tmp> ||
fsync file.txt.tmp
rename (file.txt.tmp, file.txt)

FS LAYOUT



→ Very Simple FS
Metadata block – which has data for which Inode
Inode blocks
Data blocks
D D D I I I I I
0 7
D D D D D D D D
8 15
16 23
24 31
32 39
40 47
48 55
56 63

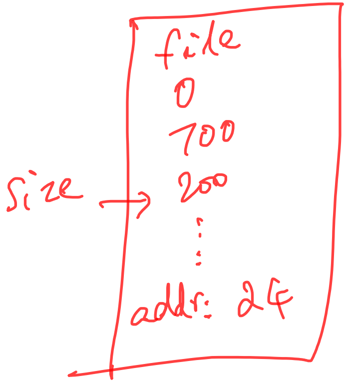
INODE

ls
stat

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

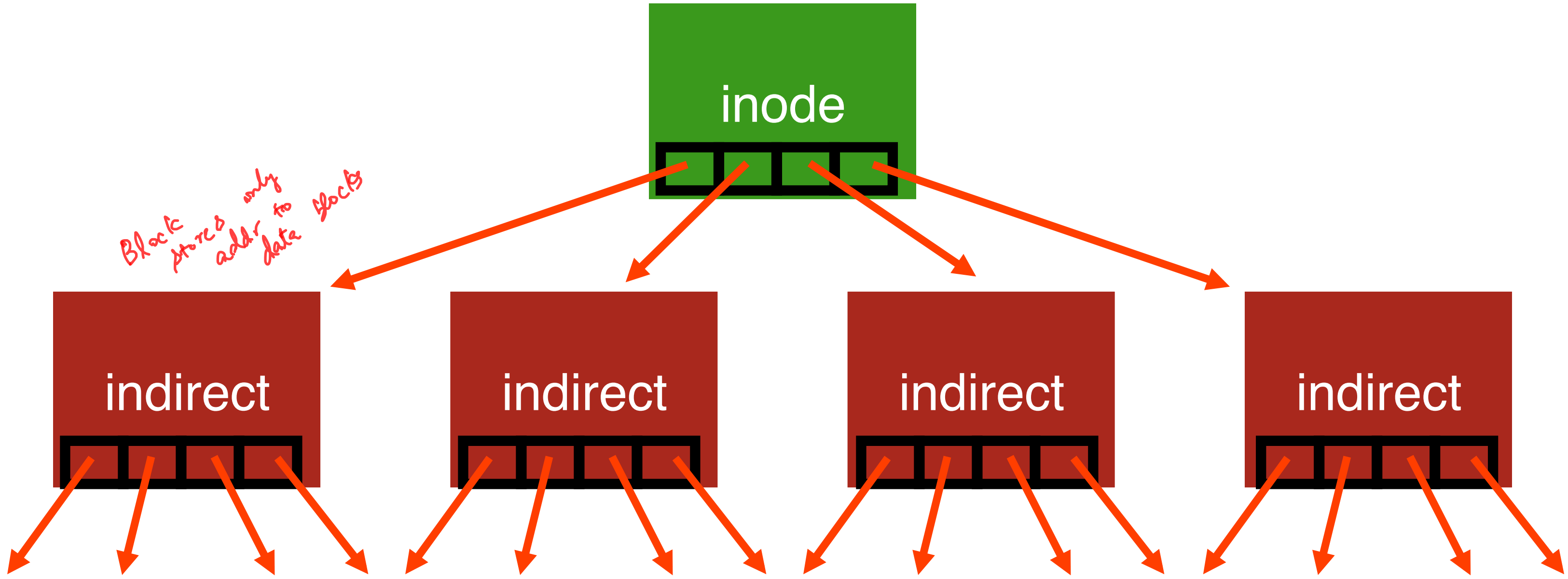
What is max file size with single level?

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



Direct pointers
↳ each block is 4 KB

Inode size is 256 byte
Each addr 4 byte
⇒ 64 addr in 1 inode
⇒ 64 × 4KB = 256 KB



Indirect blocks are stored in regular data blocks

Largest file size with 64 indirect blocks?

1 inode is 256 byte 64 ptrs to indirect block

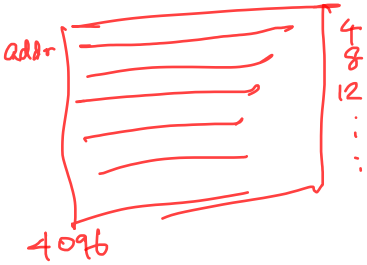
1 indirect block = 4KB , each addr 4 bytes

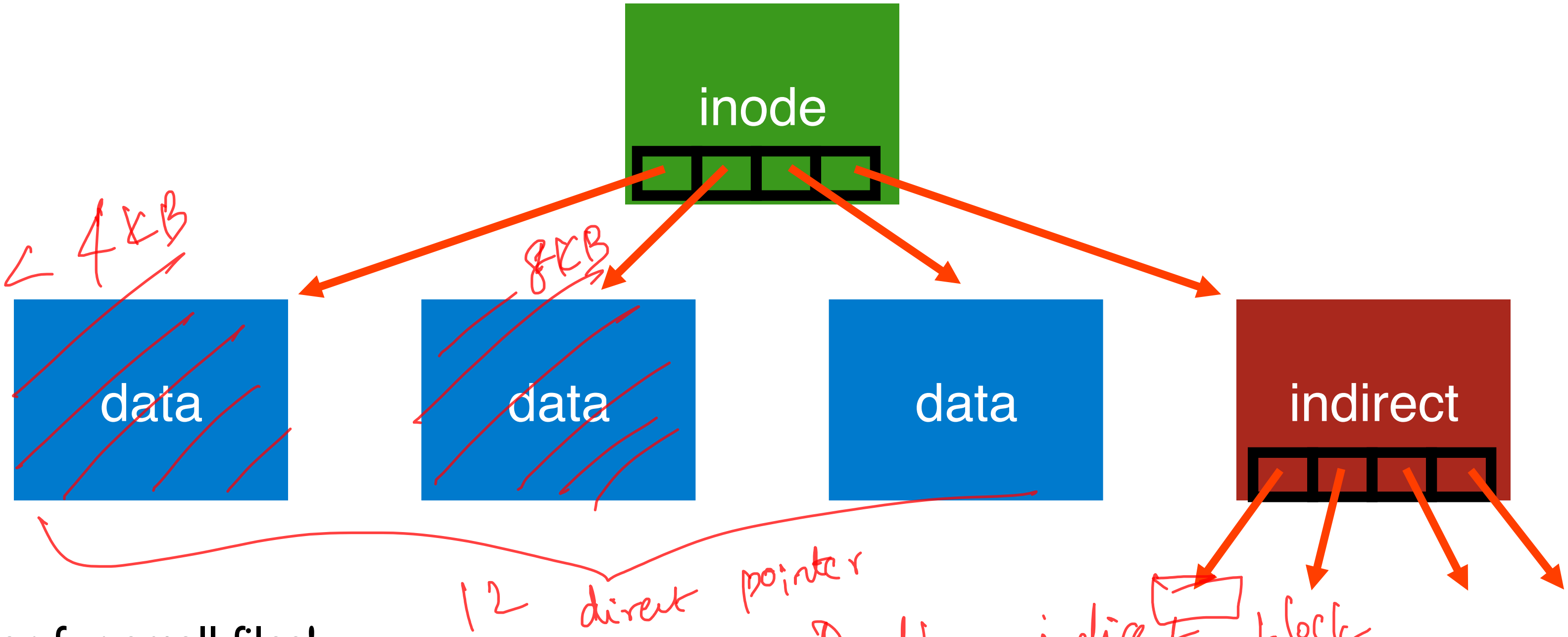
= 1024 addr in 1 indirect block

= 1024 x 4KB = 4MB

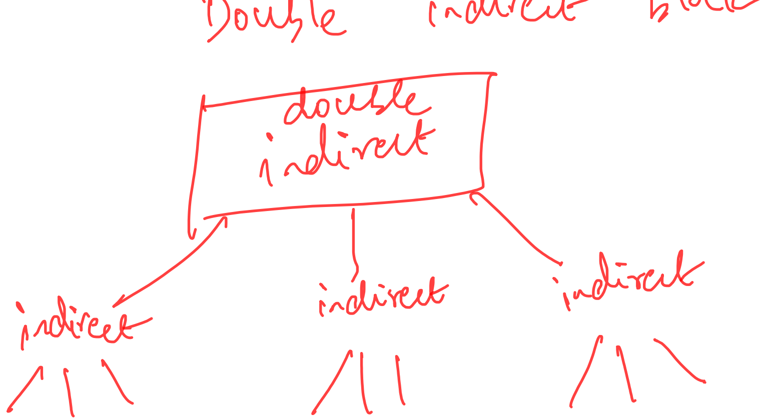
64 indirect = 64 x 4MB = 256 MB

Any Cons?





Better for small files!
How to handle even larger files?



BUNNY 15

https://tinyurl.com/cs537-sp19-bunny15

BUNNY 15

Assume 256 byte inodes (16 inodes/block).
What is the offset for inode with number 0?

12 KB

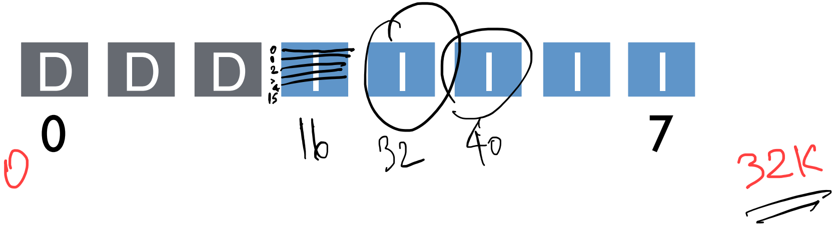
What is the offset for inode with number ~~0~~ 4?

12KB + 4 x 256 = 13KB

What is the offset for inode with number ~~0~~ 40?

12KB + 40 x 256 = 22KB

https://tinyurl.com/cs537-sp19-bunny15



DIRECTORIES

File systems vary

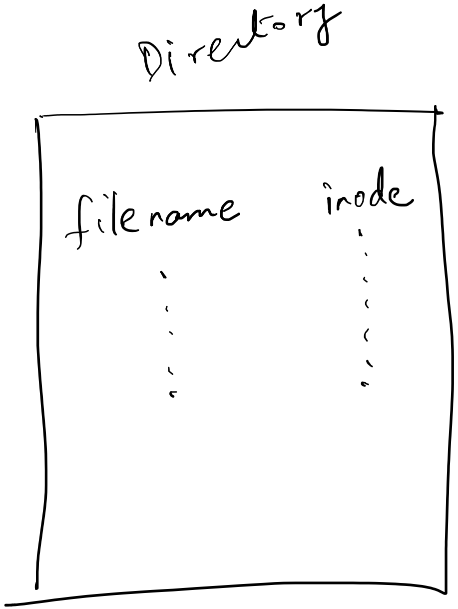
Common design:

- Store directory entries in data blocks
- Large directories just use multiple data blocks
- Use bit in inode to distinguish directories from files

type bit

Various formats could be used

- lists
- b-trees



SIMPLE DIRECTORY LIST EXAMPLE

valid	name	inode
1	.	134
1	..	35
~~1~~ 0	foo	80
1	bar	23

Special entries

→ these could also be directory

→ unlink("foo")

Create ("os")

FS STRUCTS: BITMAPS

How do we find free data blocks or free inodes?

└→ free list

list of addr which are free

Data bitmap = data structure that has 1-bit to indicate data block is used or not

1 0 1 0 . . . . . .
. . . . . 0 0 0 0
0 0 0 0 0 0 0

→ 56 bits for our layout

Inode bitmap

1 0 . . . . . .
0 0 0 1 0 0 0

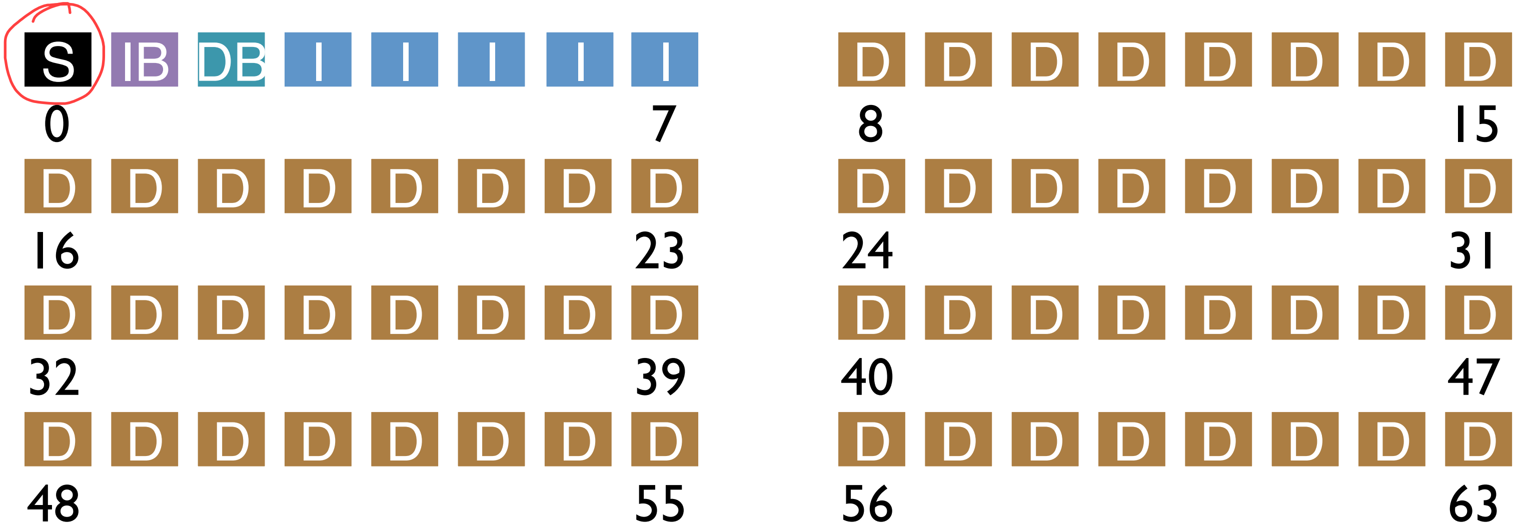
1 - bit to indicate if an I - node is used or not

→ 80 bits for our layout

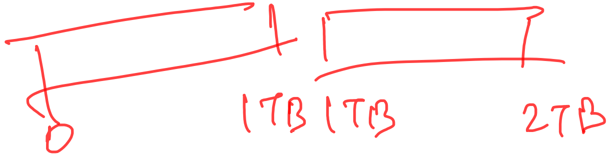
FS STRUCTS: SUPERBLOCK

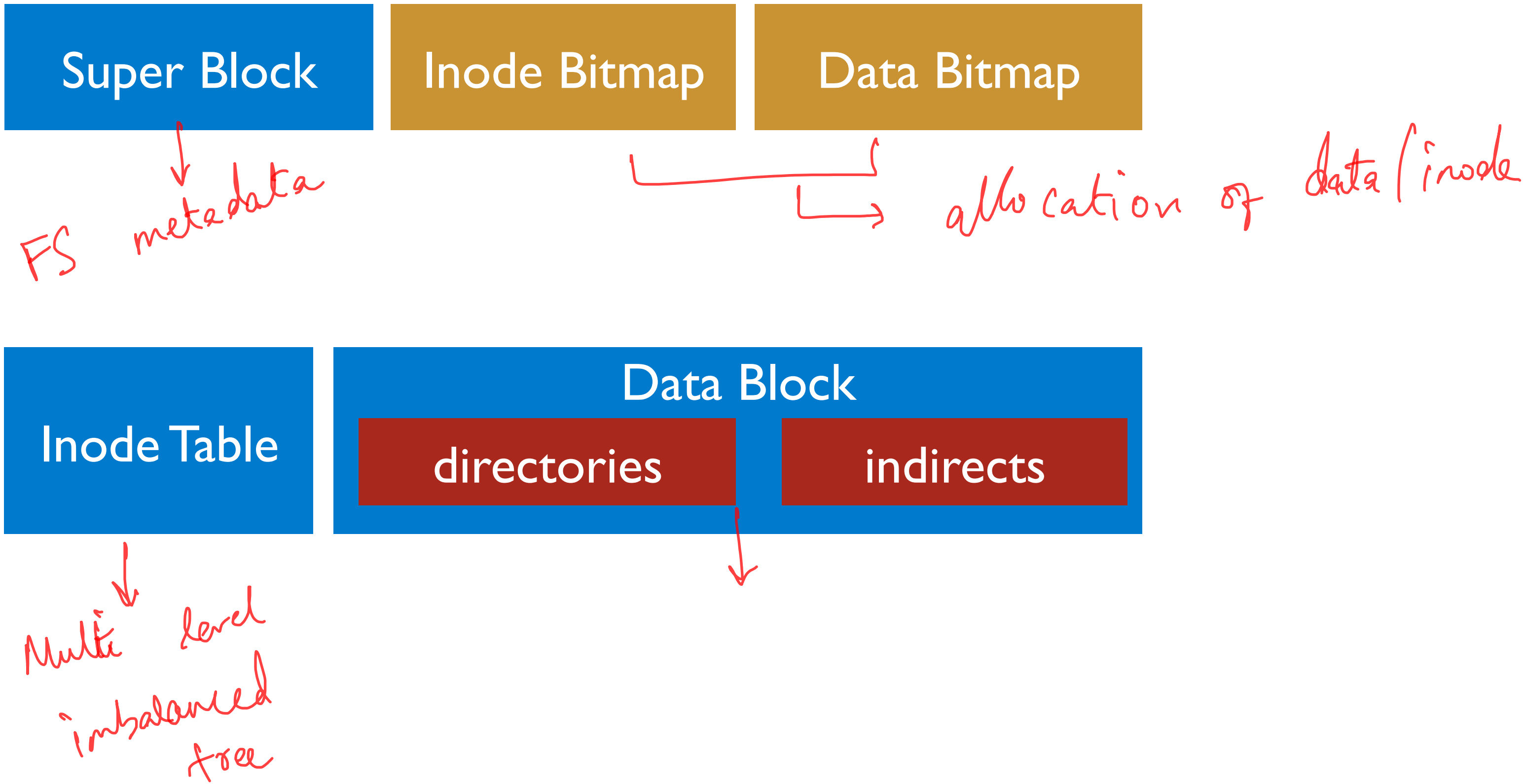
"magic string"

Basic FS configuration metadata, like block size, # of inodes



SUMMARY


0
1TB 1TB
2TB


Super Block
Inode Bitmap
Data Bitmap
FS metadata
allocation of data/inode
Inode Table
Data Block
directories
indirects
Multi level imbalanced tree

FS OPERATIONS

- create file
- write
- open
- read
- close

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		
			write			

Why must **read** for bar inode?

Check /foo/bar already exists
allocate
<i>
Traverse
bar
bar : <i>
update last modified time
10 I/O for creating a single file
Initialize
owner
type
permission
read
init
write

open /foo/bar

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

traversing

5 read for opening a file

check permission if user can open file

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

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

allocate data block

which data block 1

last modified timestamp

write to data block

read /foo/bar – assume opened

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

last accessed time stored in I-node that need to be updated close /foo/bar

→ garbage collect fd data str. in memory

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

nothing to do on disk!

EFFICIENCY

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

Cache for:

- reads → commonly read data or I-node in memory
- write buffering

"Buffer cache" → Pages of memory used to cache data & I-nodes

WRITE BUFFERING

write (fd, "hello")

Overwrites, deletes, scheduling

fsync (fd) → OS buffer write

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

Tradeoffs: how much to buffer, how long to buffer

1. write to file
   delete file || OS needs to do no I/O

2. write (fd, 0, "hello") ; write (fd, 0, "world") || Only needs to send second write to disk

3. Sequence of write

FAST FILE SYSTEM

1980's

Berkeley Software Distribution

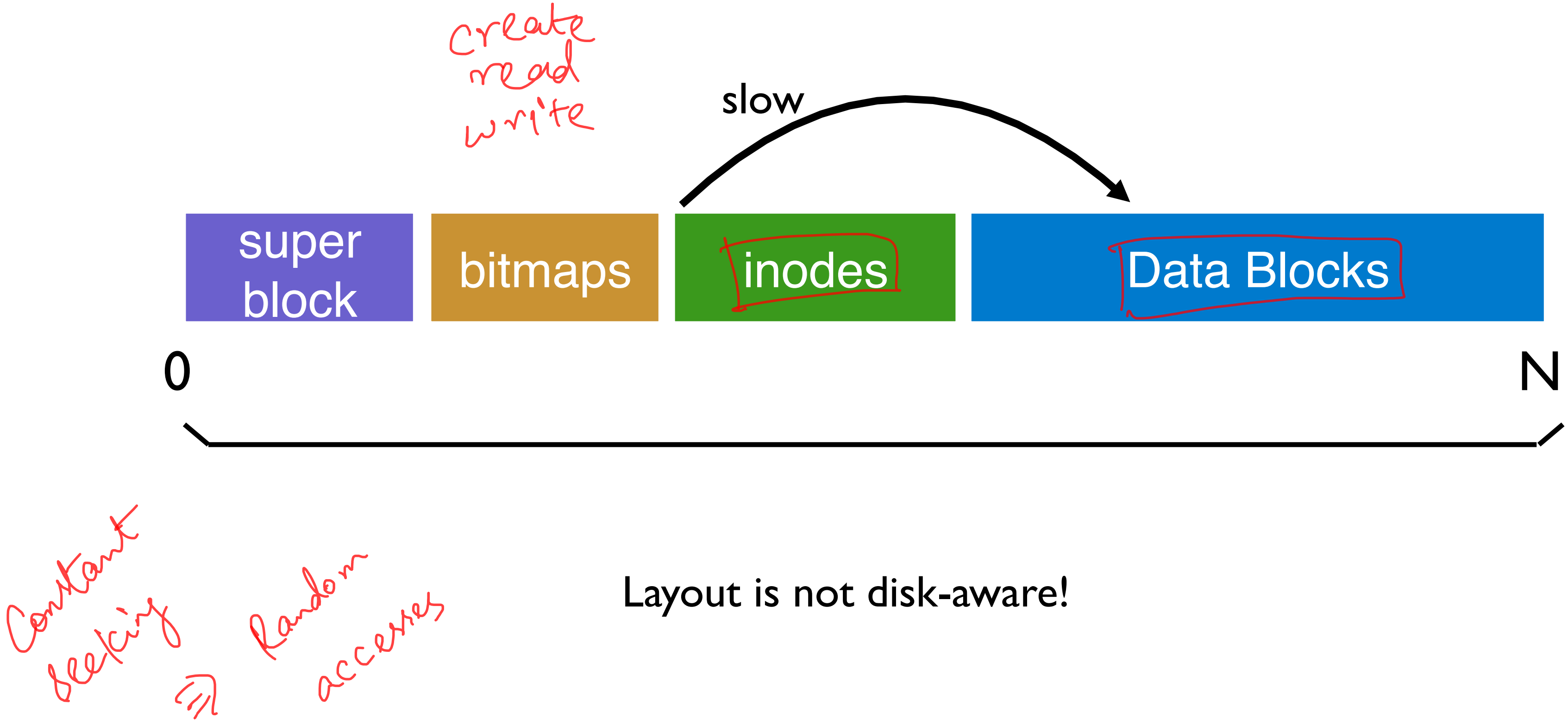
BSD ≡ based on UNIX

Open

Free

Net

FILE LAYOUT IMPORTANCE



Layout is not disk-aware!

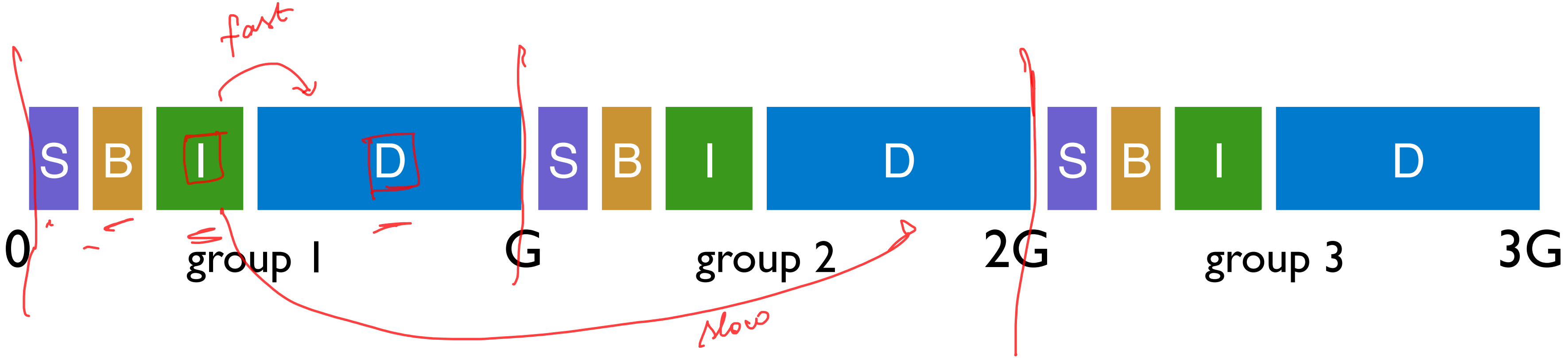
DISK-AWARE FILE SYSTEM

Given the same API

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

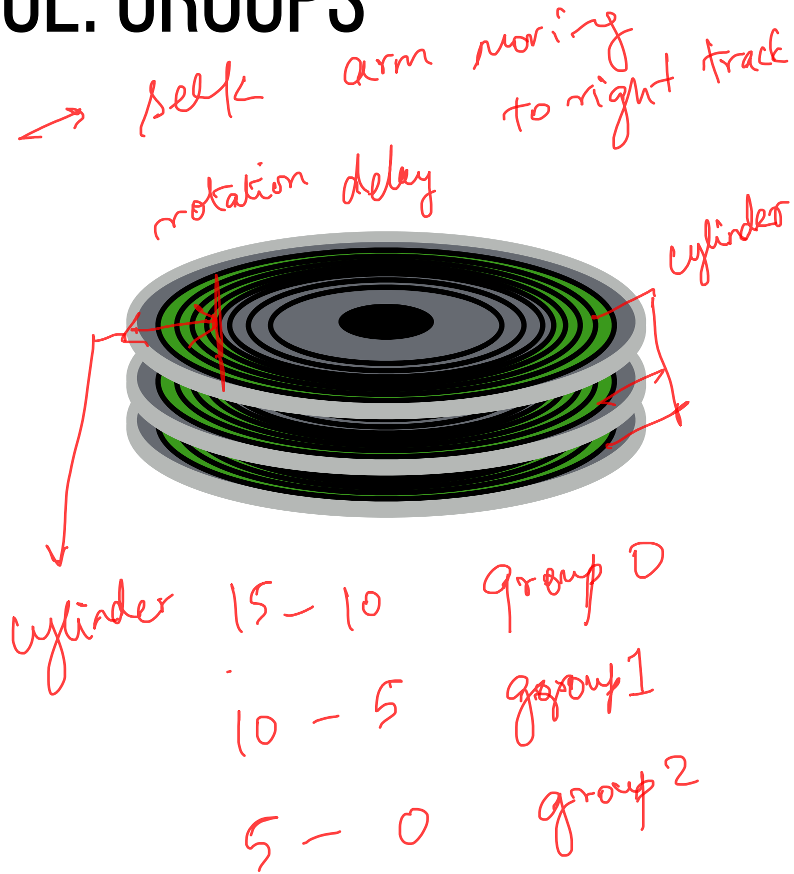
Use groups across disks;

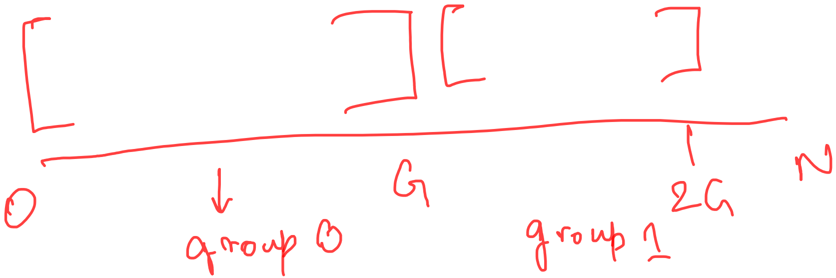
Strategy: allocate inodes and data blocks in same group.

PLACEMENT TECHNIQUE: GROUPS

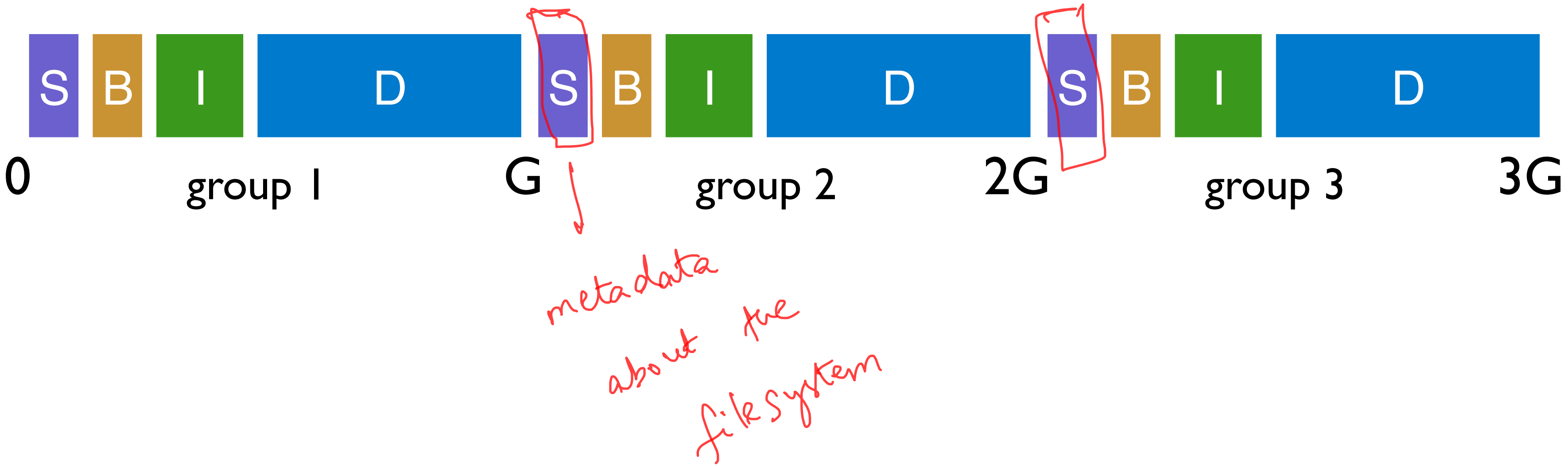
In FFS, groups were ranges of cylinders
called cylinder group

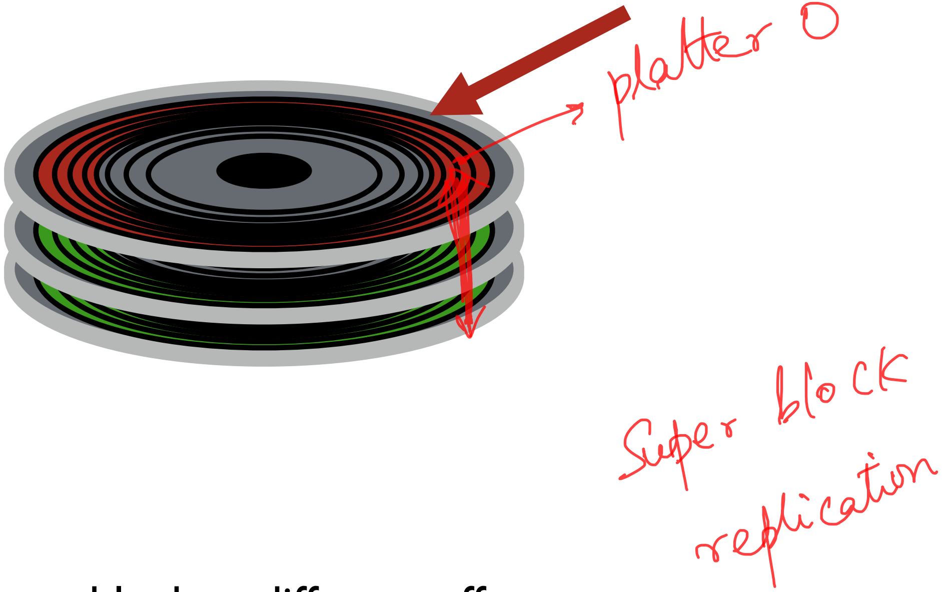
In ext2, ext3, ext4 groups are ranges of blocks
called block group





REPLICATED SUPER BLOCKS





top platter damage?

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

SMART POLICY

create /a dir

↳ create /a/b file

Where should new inodes and data blocks go?

PLACEMENT STRATEGY

Put related pieces of data near each other.

Rules:

1. Put directory entries near directory inodes.
2. Put inodes near directory entries. file inodes
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**

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

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

Trace of operation

Create /a
/a/e

mkdir /a

mkdir /b

Create /a/c
/a/d
/b/f

Create /b/g

Compiling all .c
into a binary

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

/a used up all data
/b different group
/c

```
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”

BUNNY 16

https://tinyurl.com/cs537-sp19-bunny16

BUNNY 16

Assume that the average positioning time (i.e., seek and rotation) = 10 ms.
Assume that disk transfers data at 100 MB/s.

If FFS large file chunk size is 4MB, what is the effective throughput we are getting?

What is the effective throughput with 8MB chunk size?

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.

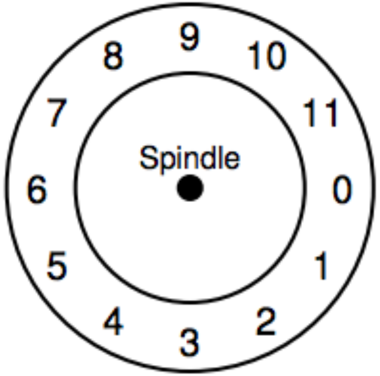
Move to another group (w/ fewer than avg blocks) every subsequent 1MB.

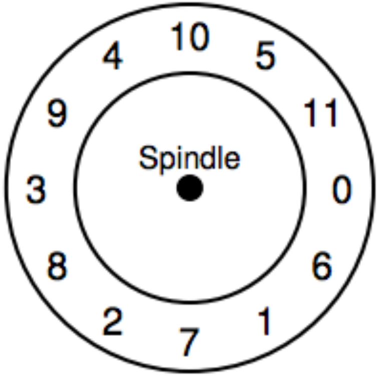
OTHER FFS FEATURES

FFS also introduced several new features:

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

FFS: SECTOR PLACEMENT





Similar to track skew in disks chapter

Modern disks: Disk cache

FFS SUMMARY

First disk-aware file system

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

Inspired modern files systems, including ext2 and ext3

OTHER TAKEAWAYS

All hardware is unique

Treat disk like disk!

Treat flash like flash!

Treat random-access memory like random-access memory!

NEXT STEPS

Next class: How to provide consistency despite failures?

Discussion today: Worksheet with problems, Q&A for project 4b