PERSISTENCE: FILE SYSTEMS

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Midterm 2?

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

Shivaram’s OH / next week schedule

AGENDA / LEARNING OUTCOMES

How does file system represent files, directories?

What steps must reads/writes take?

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

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>
```

LINKS

Hard links: Both path names use same inode number

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

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

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

FILE API SUMMARY

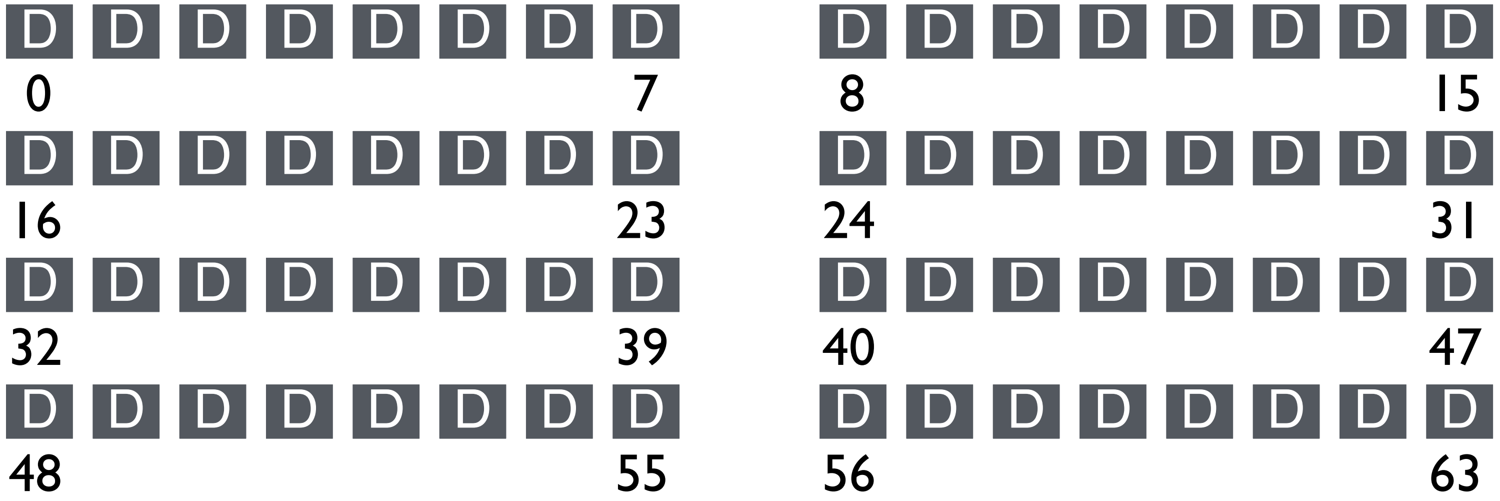
Using multiple types of name provides convenience and efficiency

Hard and soft link features provide flexibility.

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

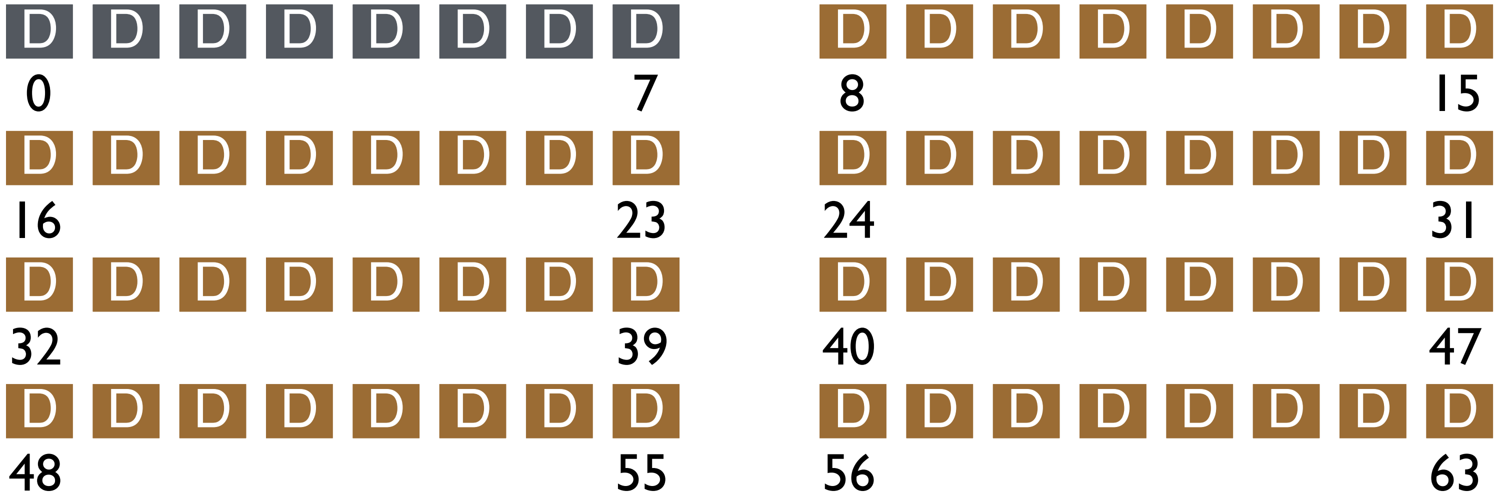
FILESYSTEM DISK STRUCTURES

FS STRUCTS: EMPTY DISK



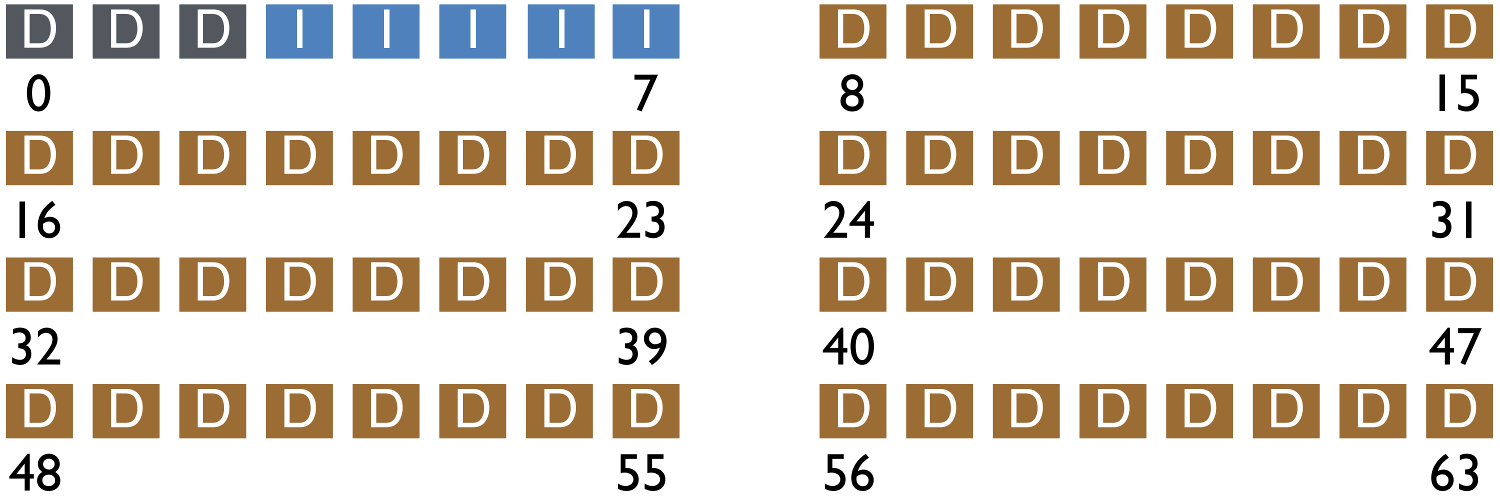
Assume each block is 4KB

FS STRUCTS: DATA BLOCKS



Simple layout → Very Simple File System

FS STRUCTS: INODE DATA POINTERS



INODE

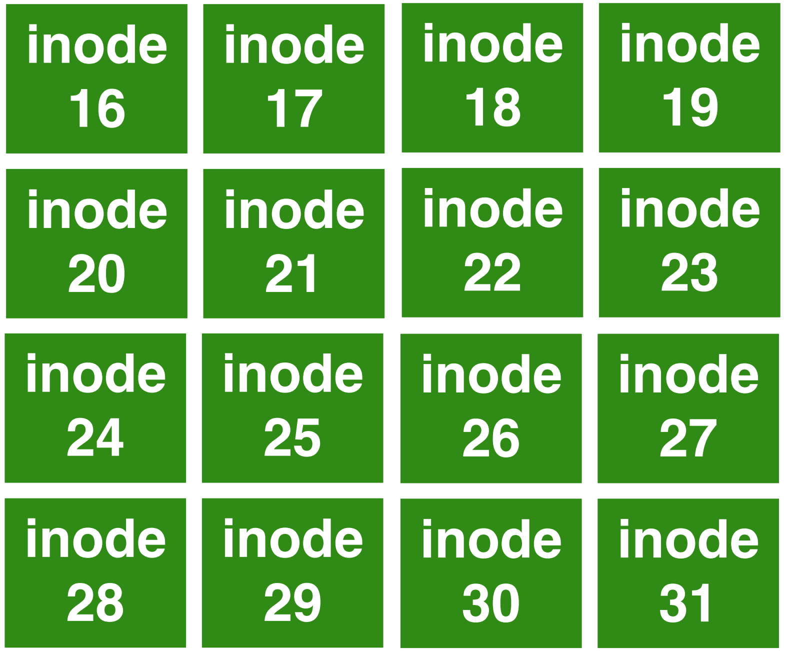
type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

ONE INODE BLOCK

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

4KB disk block

16 inodes per inode block.



INODE

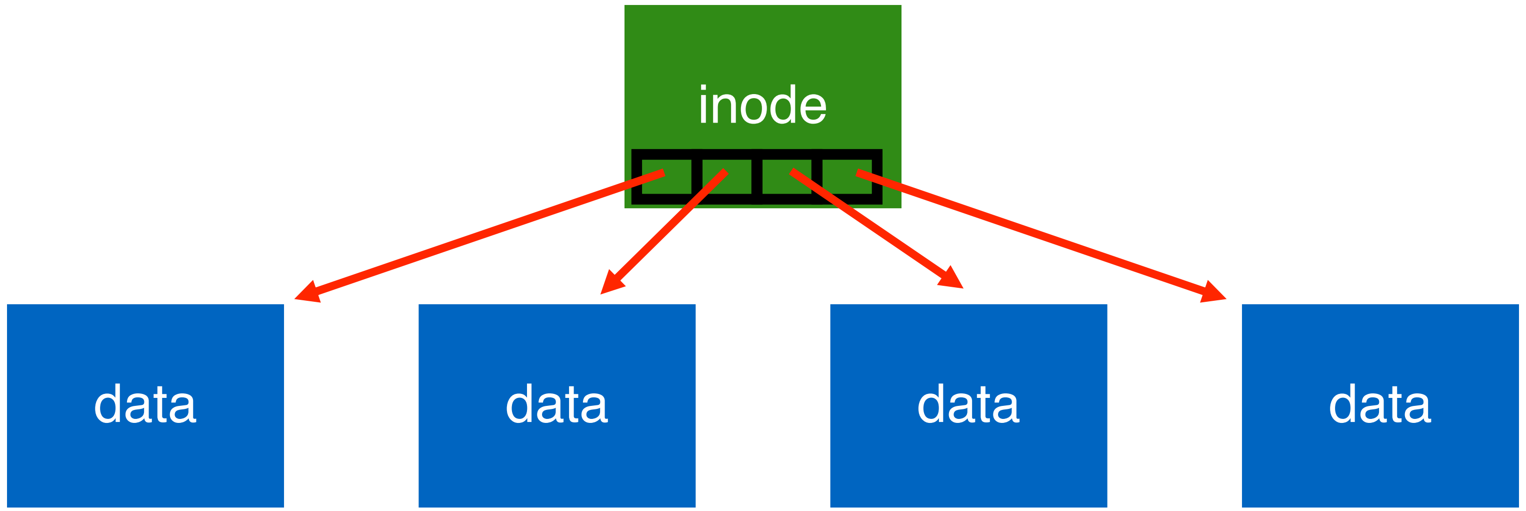
type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

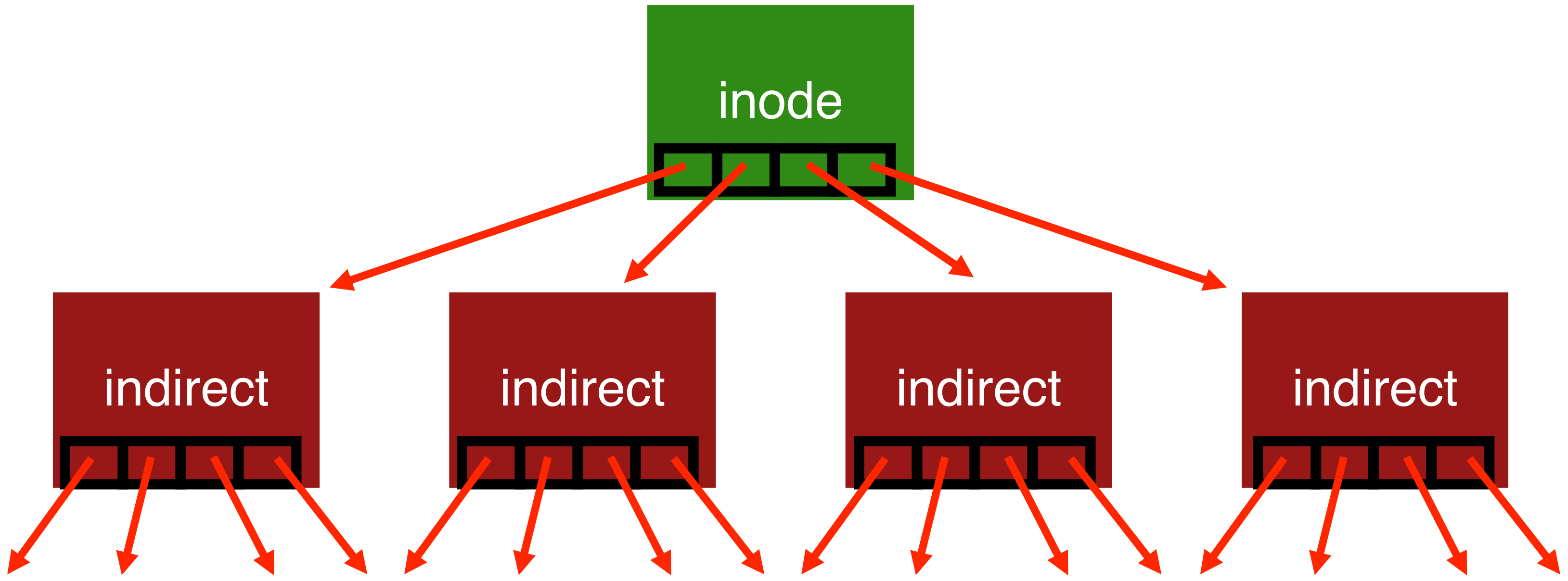
Assume single level (just pointers to data blocks)

What is max file size?

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

How to get larger files?

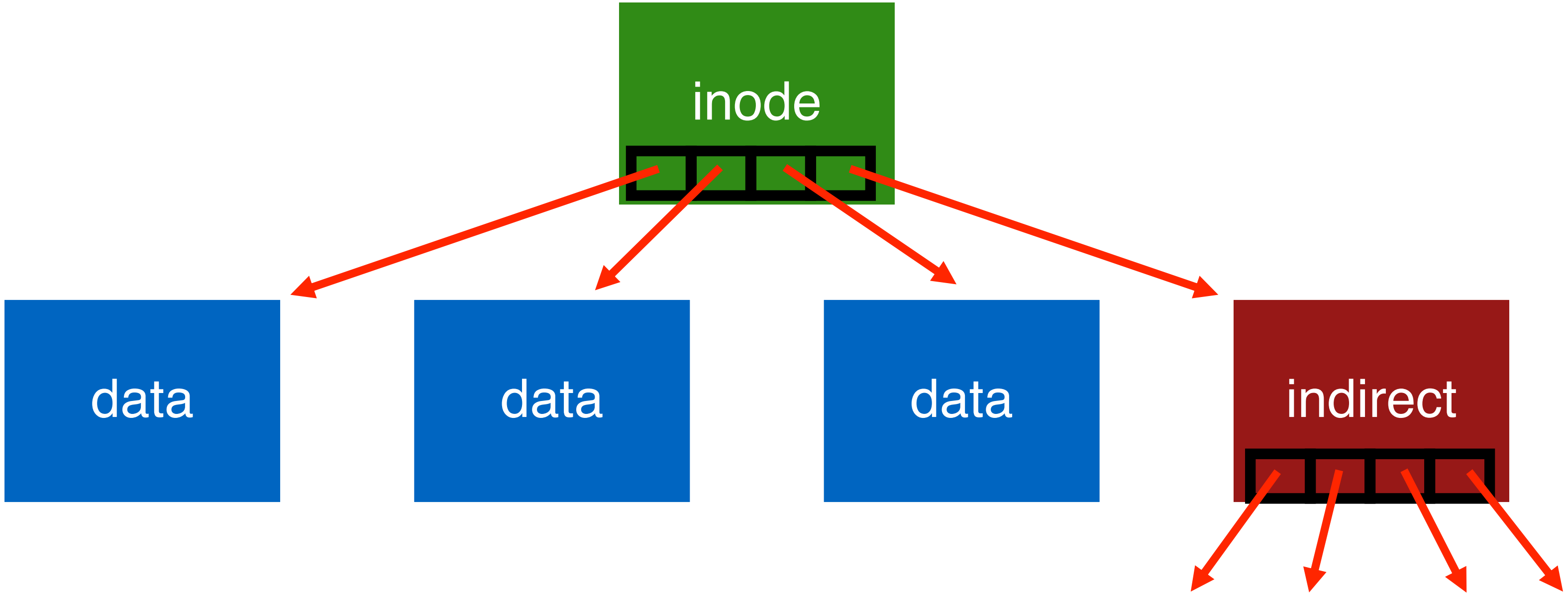

inode
data
data
data
data



Indirect blocks are stored in regular data blocks

Largest file size with 64 indirect blocks?

Any Cons?



Better for small files!

How to handle even larger files?

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

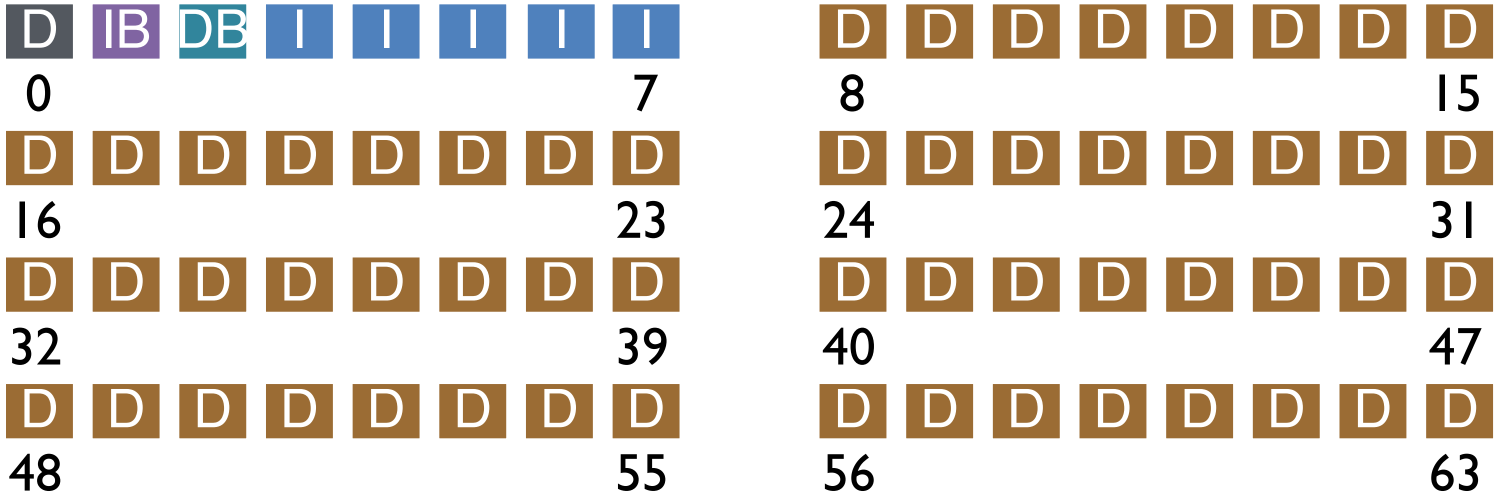
valid	name	inode
1	.	134
1	..	35
1	foo	80
1	bar	23

Various formats could be used

- lists
- b-trees

FS STRUCTS: BITMAPS

How do we find free data blocks or free inodes?



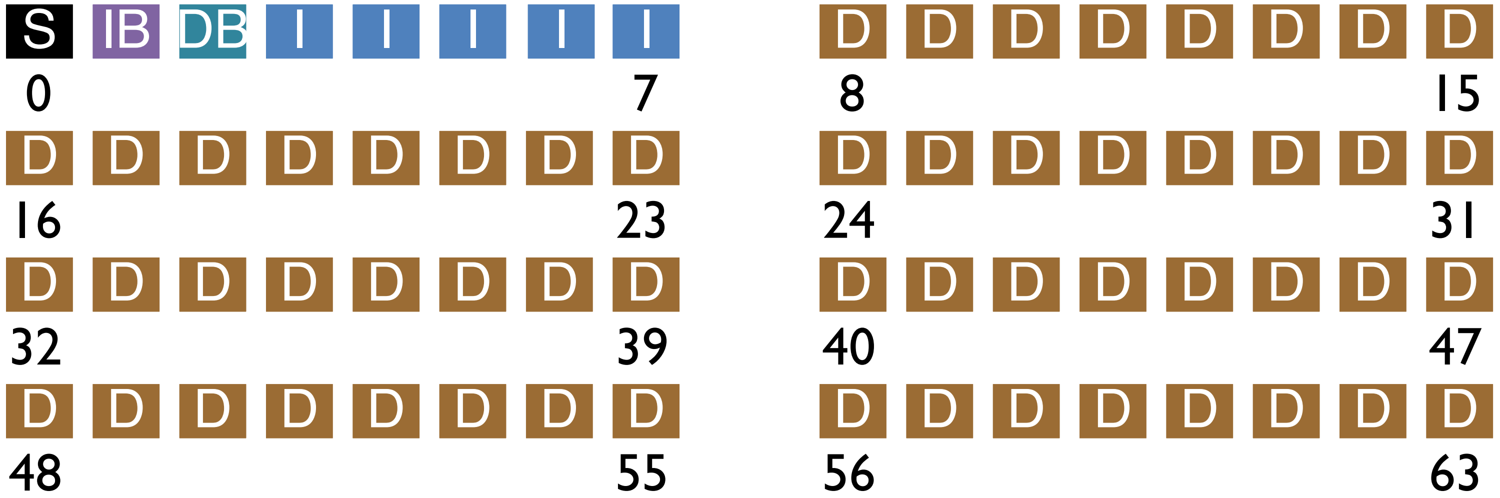
SUPERBLOCK

Need to know basic FS configuration metadata, like:

- block size
- # of inodes

Store this in superblock

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
```

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
```

```
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)
```

```
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);
```

FS OPERATIONS

- open
- read
- close
- create file
- write

open /foo/bar

TIME

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			

read /foo/bar – assume opened

TIME

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

close /foo/bar

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

nothing to do on disk!

create /foo/bar

TIME

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

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

TIME

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

EFFICIENCY

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

Cache for:

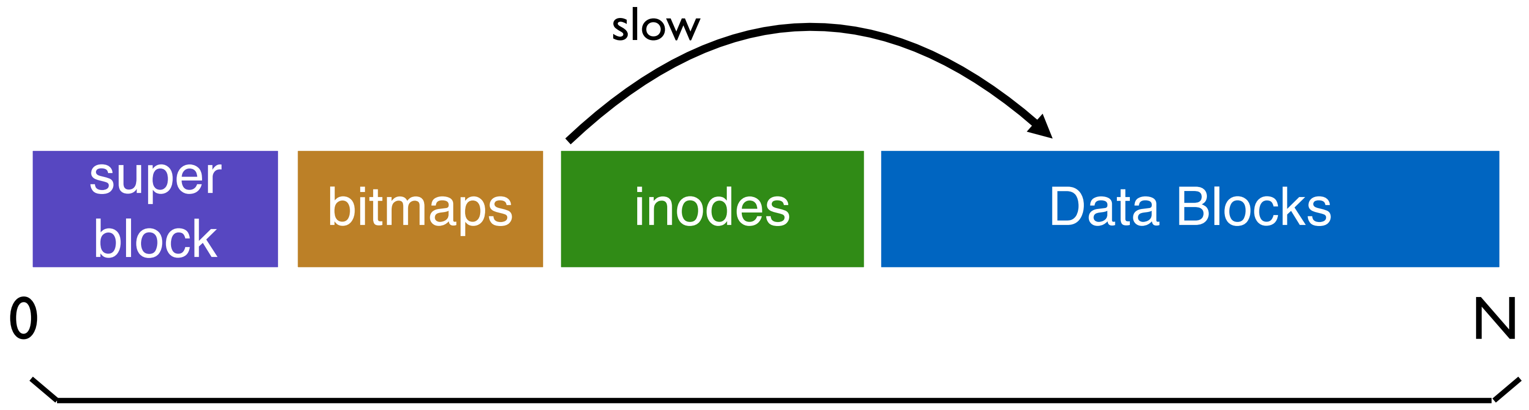
- reads
- write buffering

Overwrites, deletes, scheduling

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

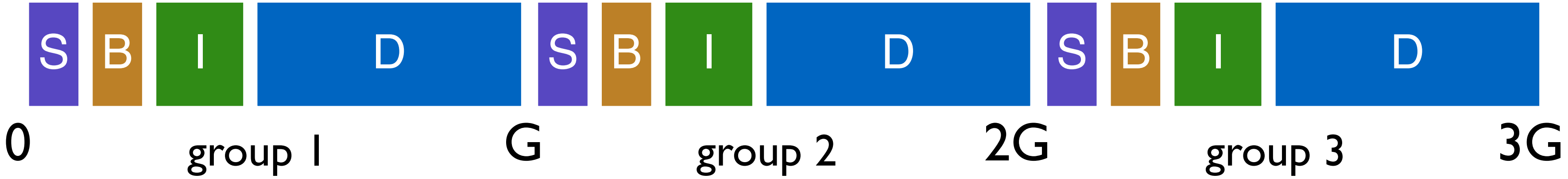
Tradeoffs: how much to buffer, how long to buffe

FFS: FILE LAYOUT IMPORTANCE

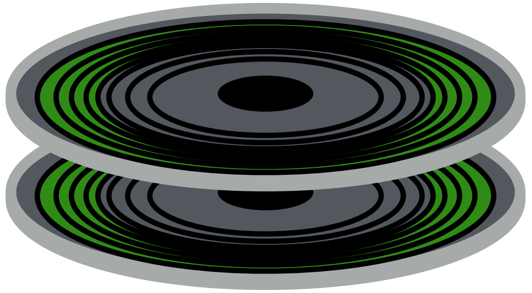


Layout is not disk-aware!

PLACEMENT TECHNIQUE: GROUPS



Key idea: Keep inode close to data

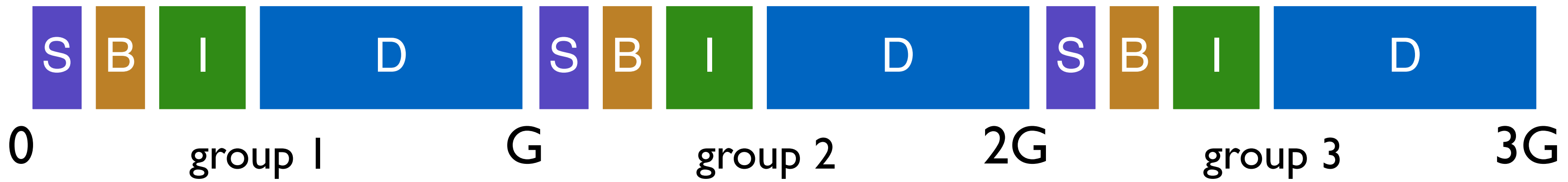


Use groups across disks

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

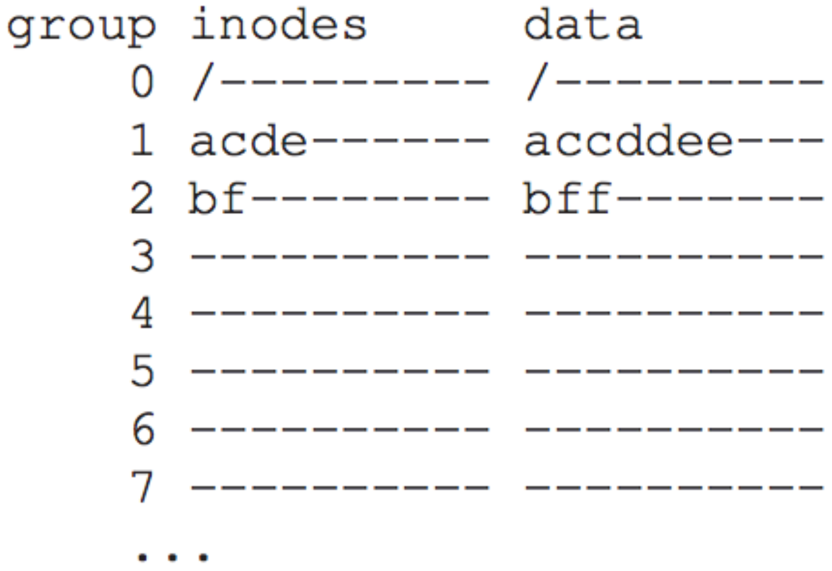
Trying to put everything near everything else doesn’t make any choices!

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

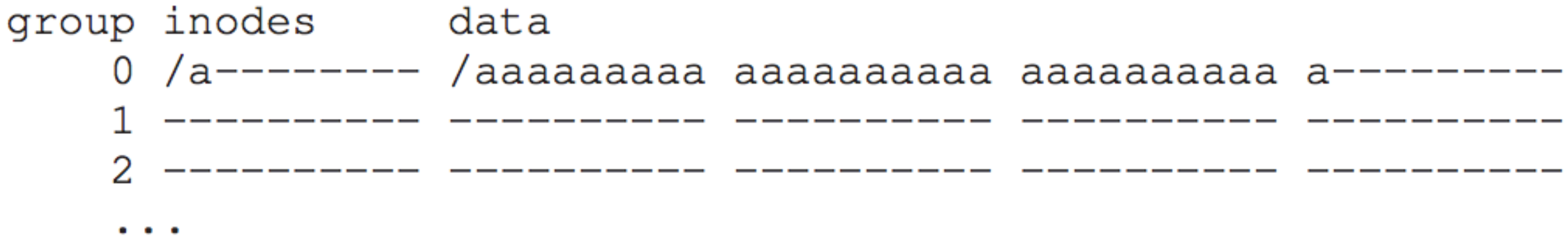
First data block: allocate near inode

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