PERSISTENCE: FFS

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

Make-up Points for Projects 2 + 3 (more later) – Canvas Quizzes
- Specification and Concepts/Implementation
  - Can take only one time
  - Up to 35 more points (scaled);
    - if > 70 points, no benefit
    - if received < 50 points, expect to take
- Due 1 week from when made available (Fri, Mon)

File System Structures Quiz: Lecture Content
- Due Next Tuesday

Project 6: Due next Friday
- Specification Quiz – Due yesterday

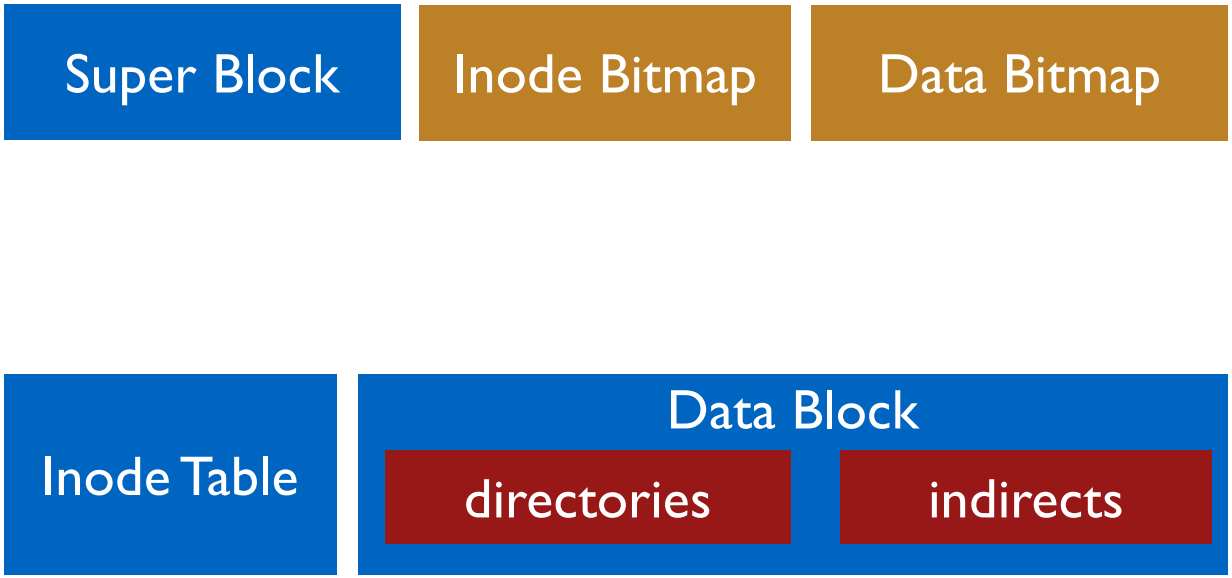
AGENDA / LEARNING OUTCOMES

How does FFS improve performance?

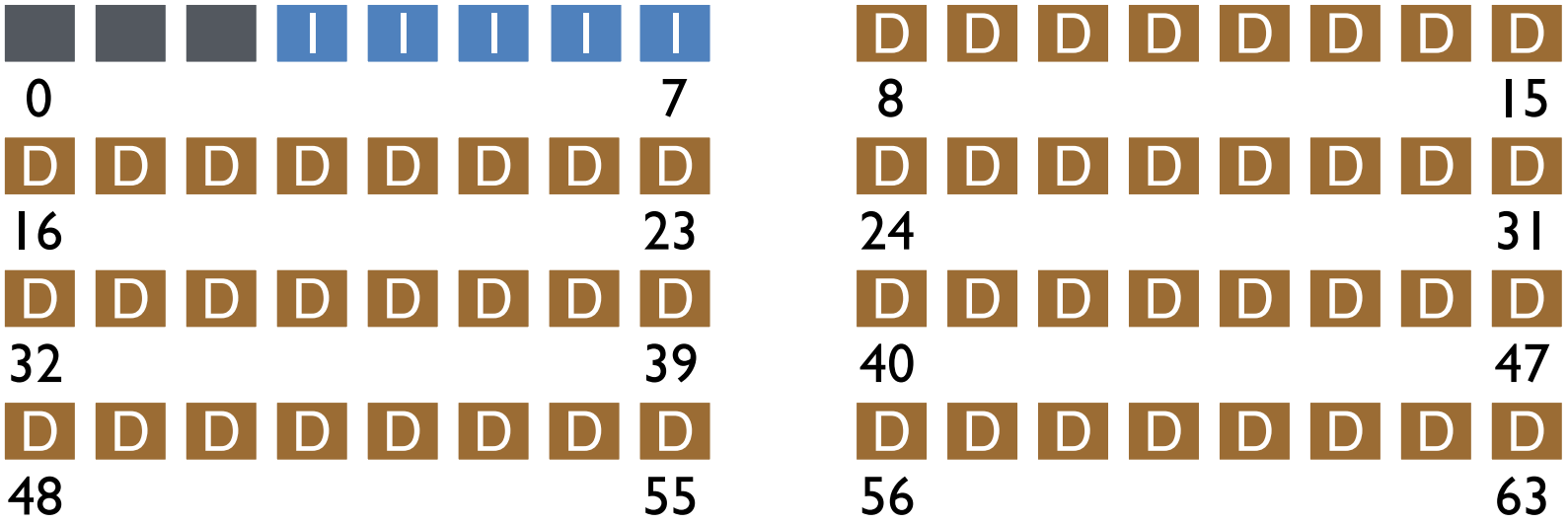
- How to improve performance of complex system?
- Why do file systems obtain worse performance over time?
- How to choose the right block size? How to avoid internal fragmentation?
- How to place related blocks close to one another on disk?

RECAP

SUMMARY


Super Block
Inode Bitmap
Data Bitmap
Inode Table
Data Block
directories
indirects

FS LAYOUT


I I I I I
0 7
D D D D D D D D
8 15
D D D D D D D D
16 23
D D D D D D D D
24 31
D D D D D D D D
32 39
D D D D D D D D
40 47
D D D D D D D D
48 55
D D D D D D D D
56 63

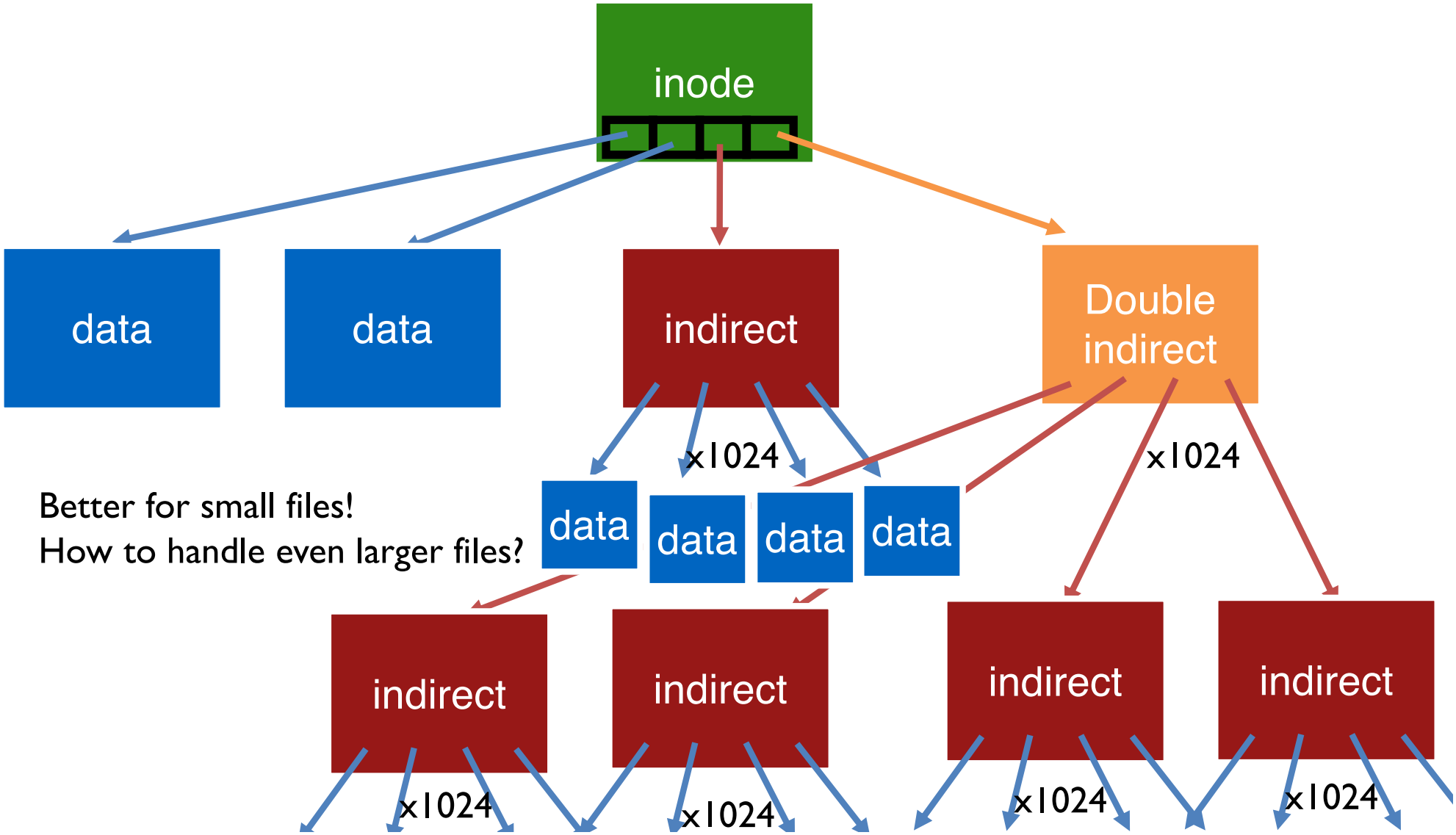
INODE

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
Data pointers[N] (N data blocks)
Indirect Pointer
Double Indirect Pointer
Triple Indirect Pointer

Multiple inodes per disk block

To modify inode:

- Read old disk block containing inode
- Change inode in memory
- Write entire disk block out to disk


inode
data
data
indirect
Double indirect
x1024
x1024
Better for small files!
How to handle even larger files?
data
data
data
data
indirect
indirect
indirect
indirect
x1024
x1024
x1024
x1024

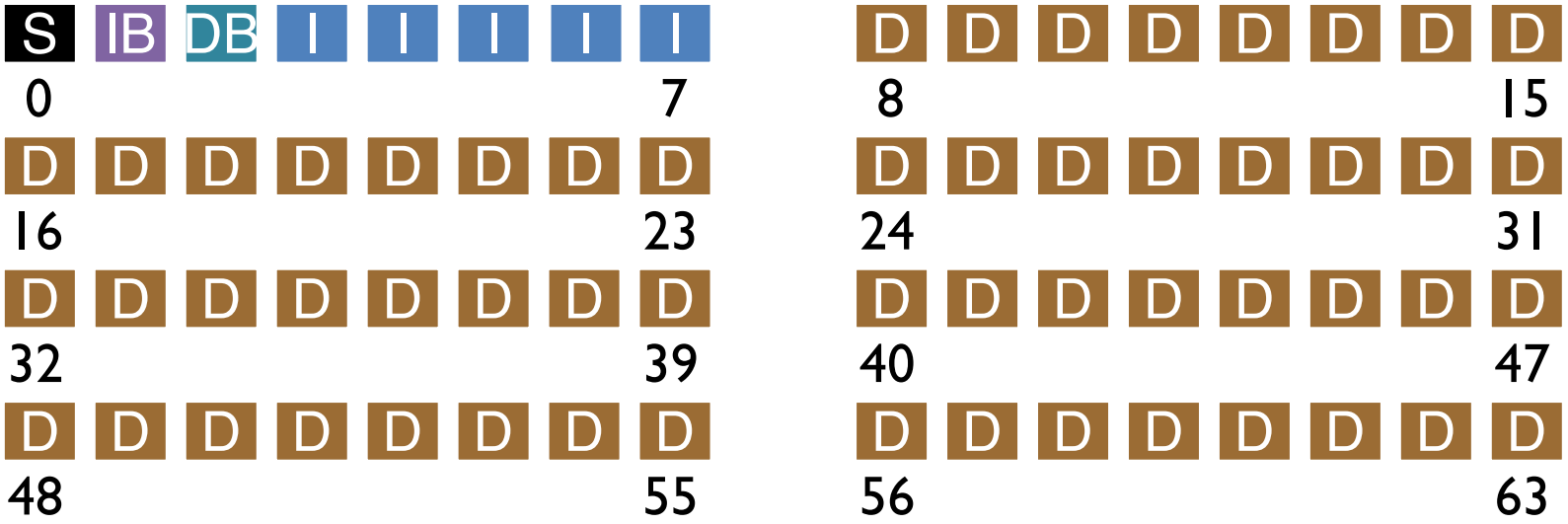
SIMPLE DIRECTORY LIST EXAMPLE

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

unlink("foo")

FS STRUCTS: SUPERBLOCK

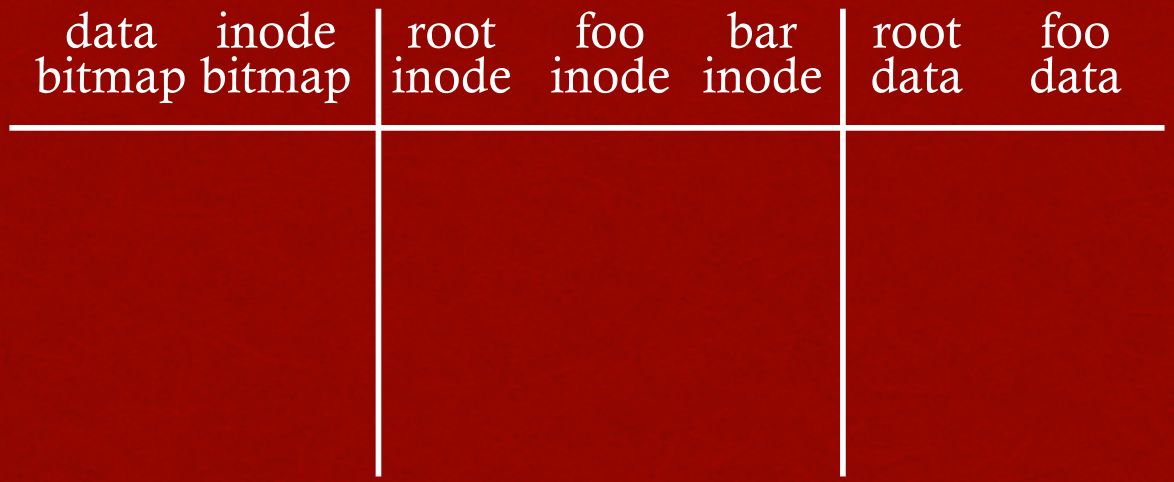
Basic FS configuration metadata, like block size, # of inodes



FS OPERATIONS

- create file
- write
- open
- read
- close

REVIEW: create /foo/bar

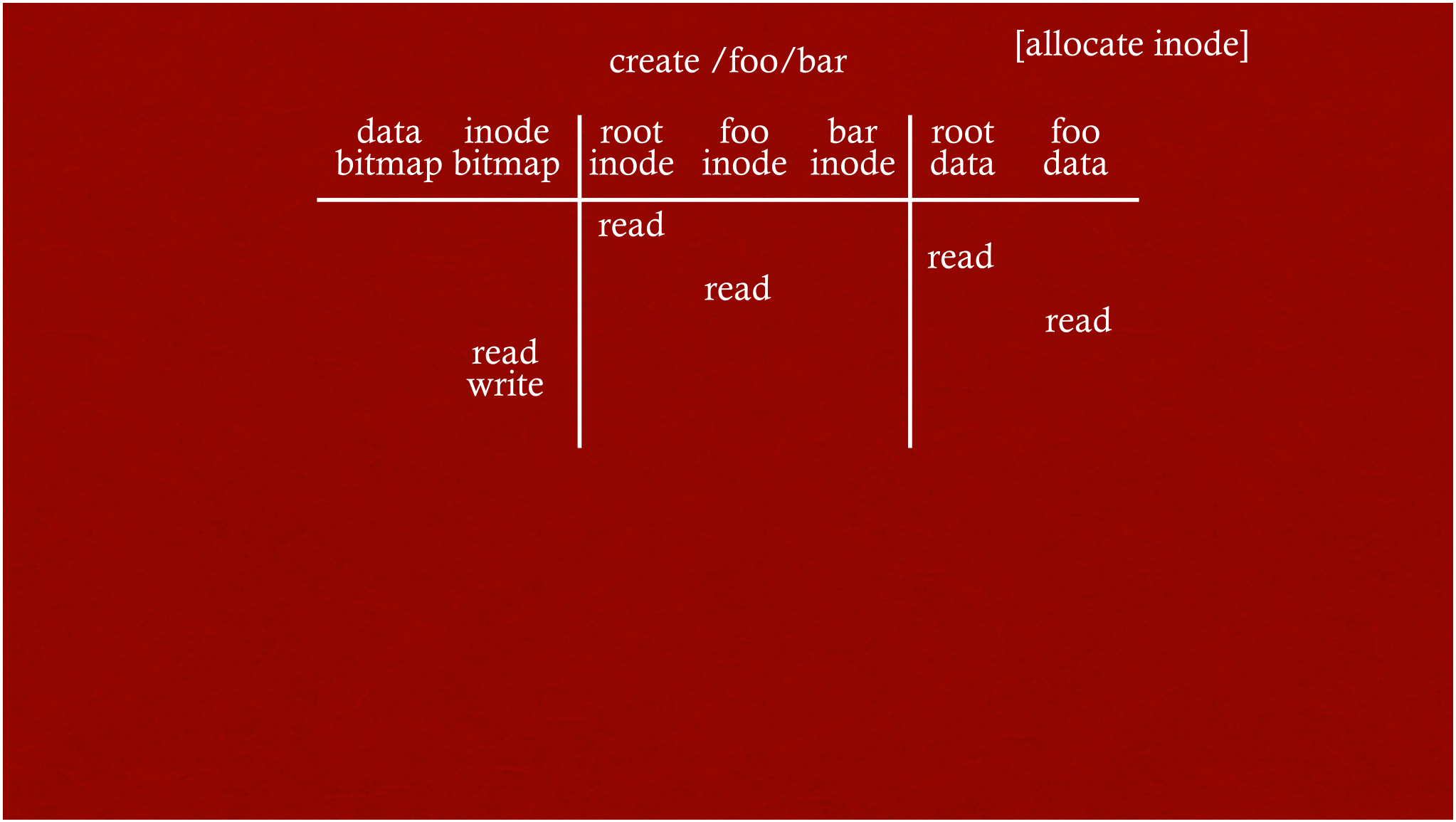


create /foo/bar

[traverse path]

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

Verify that bar does not already exist


create /foo/bar
[allocate inode]
data bitmap
inode bitmap
root inode
foo inode
bar inode
root data
foo data
read
read
read
read
write
read

create /foo/bar

[populate inode]

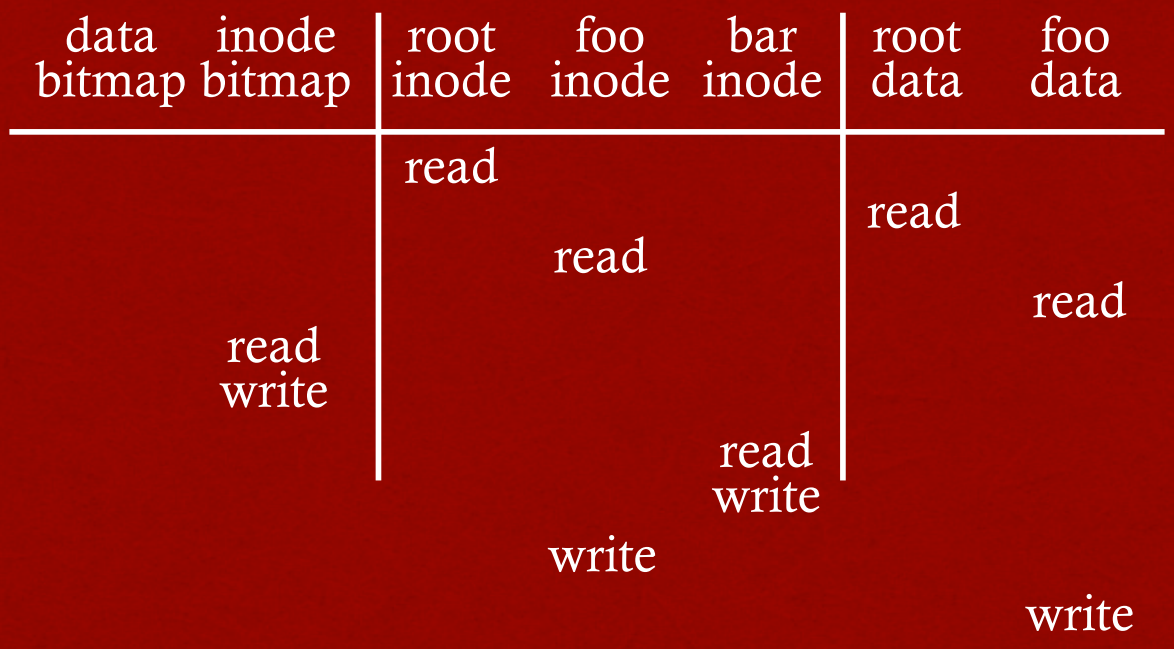
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read write					
				read write		

Why must **read** bar inode?

How to initialize inode?

create /foo/bar

[add bar to /foo]



Update inode (e.g., size) and data for directory

open /foo/bar

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

write to /foo/bar (assume file exists, is empty, 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			

No reads or writes to / or /foo

append to /foo/bar (opened already)

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

append to /foo/bar

[allocate block]

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

append to /foo/bar

[point to block]

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

append to /foo/bar [write to block]

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

What assumptions did we make for append?

Needed to allocate a new data block (appended data didn't fit in current block)

How would file system know this?

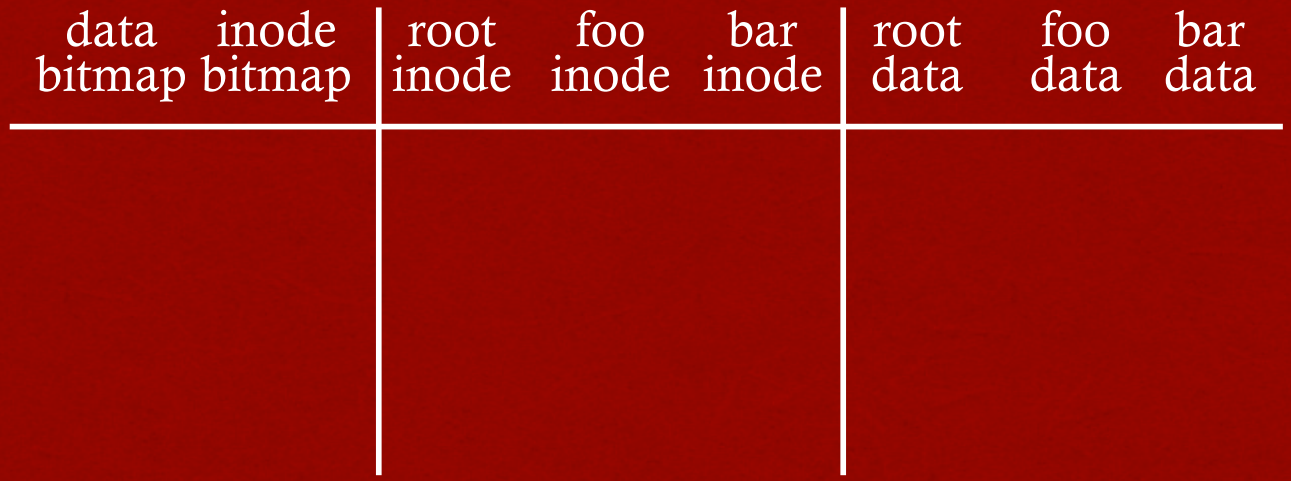
What steps would be skipped if don't need to allocate new data block?

read /foo/bar – assume opened

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

(Skipping updates to timestamps in inode which would cause another write)

close /foo/bar



nothing to do on disk!

FAST FILE SYSTEM: FFS (1980S)

SYSTEM BUILDING

Beginner's approach

1. get idea
2. build it!

Measure then build

Pro approach

1. identify existing state of the art
2. measure it, identify and understand problems
3. get idea to improve
   - solutions often flow from deeply understanding problem
4. build it!

Iterative process

MEASURE OLD FS

State of the art: Original UNIX file system

Free lists are embedded in inodes, data blocks
Data blocks are 512 byte sectors (smallest possible)

Measure throughput for whole sequential file reads/writes

Compare to theoretical max, which is? disk bandwidth

Old UNIX file system: Achieved only **2%** of potential. Why?

MEASUREMENT 1: AGING?

What is performance before/after **aging**?

Over time, files created and deleted, new files created in newly freed space

- New FS: **17.5%** of disk bandwidth
- Few weeks old: **3%** of disk bandwidth

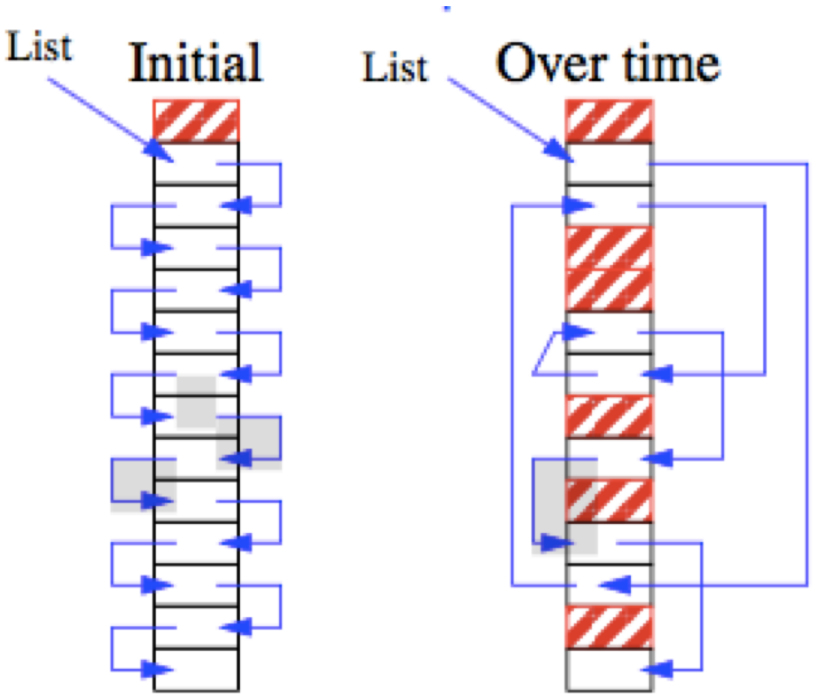
Problem: FS is becomes fragmented over time

- Free list makes contiguous chunks hard to find

MEASUREMENT 1: AGING?

Problem: FS becomes fragmented over time

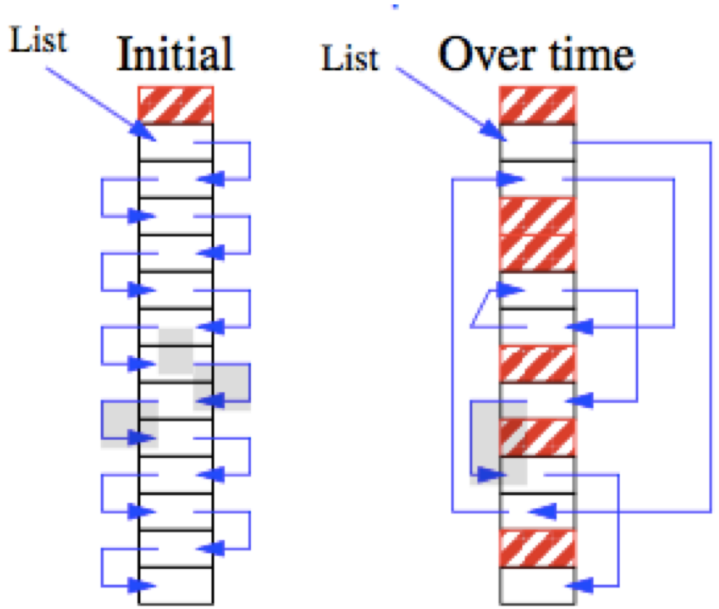
- Free list makes contiguous chunks hard to find



FREELIST SOLUTIONS?

Hacky Solutions:

- Occasionally defragment disk: move around existing files
- Keep freelist sorted



MEASUREMENT 2: BLOCK SIZE?

How does block size affect performance?
Try doubling it (from 512 bytes to 1KB)

Result: Performance **more** than doubled

Why **double** the performance?

- Logically adjacent blocks were not physically adjacent
- With half the blocks, only half as many seeks+rotations required

Why **more** than double the performance?

- Larger blocks require fewer indirect blocks

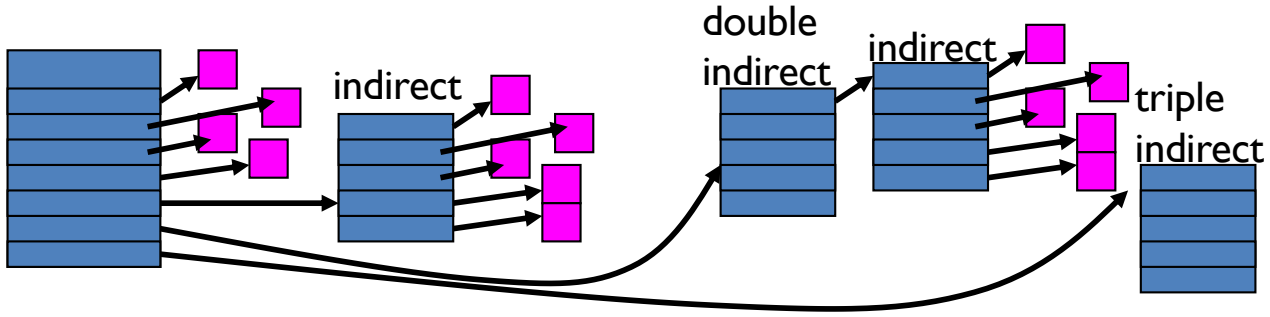
MULTI-LEVEL INDEXING

Dynamically allocate hierarchy of pointers to blocks as needed

Meta-data: Small number of pointers allocated statically

Additional pointers to blocks of pointers

Examples: UNIX FFS-based file systems, ext2, ext3



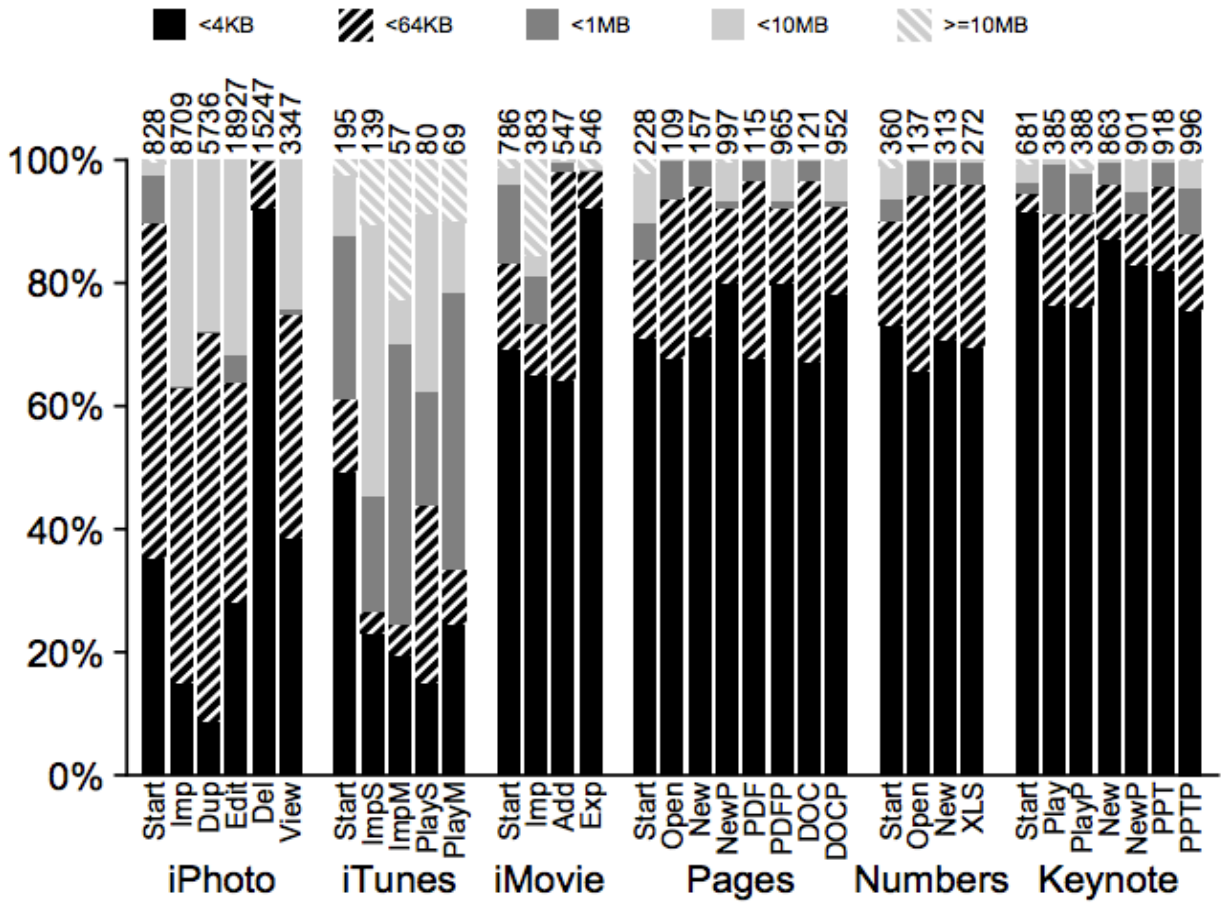
Double block size: Reach 2x data directly; Indirect blocks fit 2x as many pointers

TECHNIQUE: LARGER BLOCKS

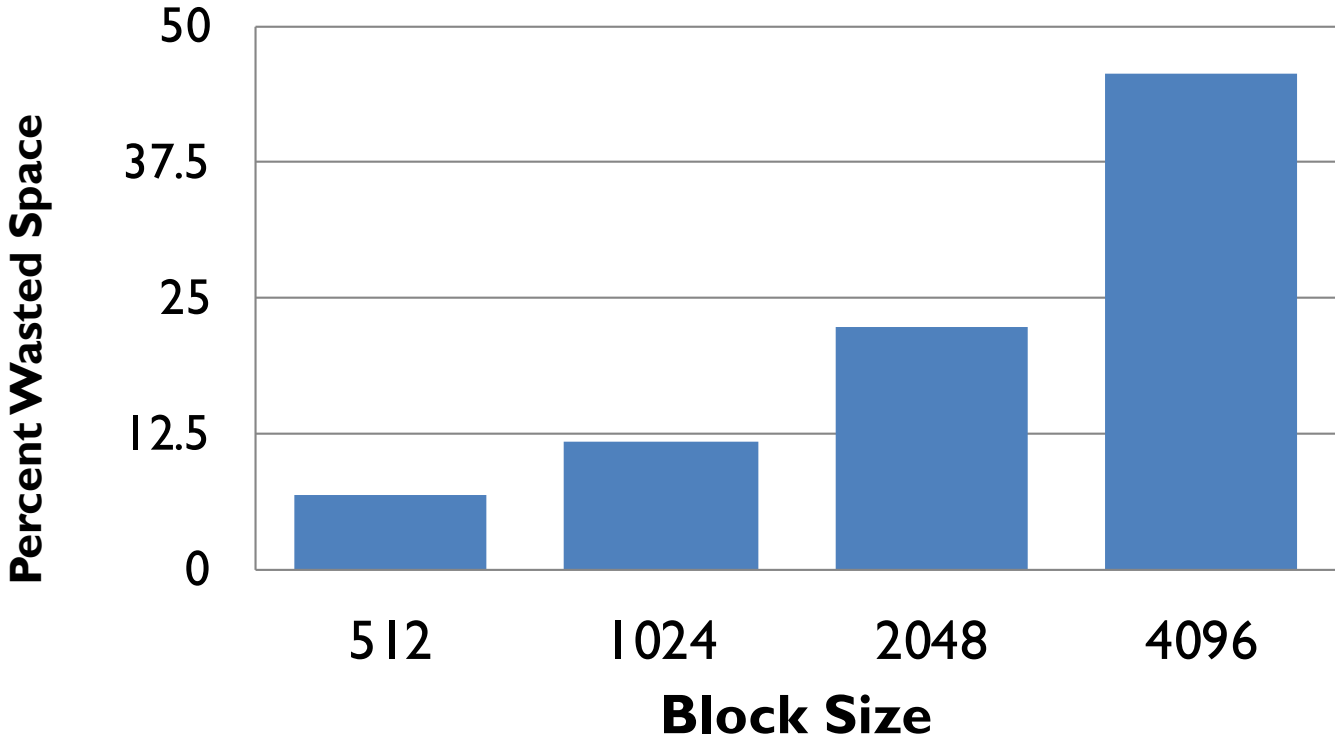
Observation: Doubling block size for old FS over doubled performance

Why not make blocks huge?

Most file are very small, even today!



IMPACT OF LARGER BLOCKS



Lots of waste due to internal fragment in most blocks

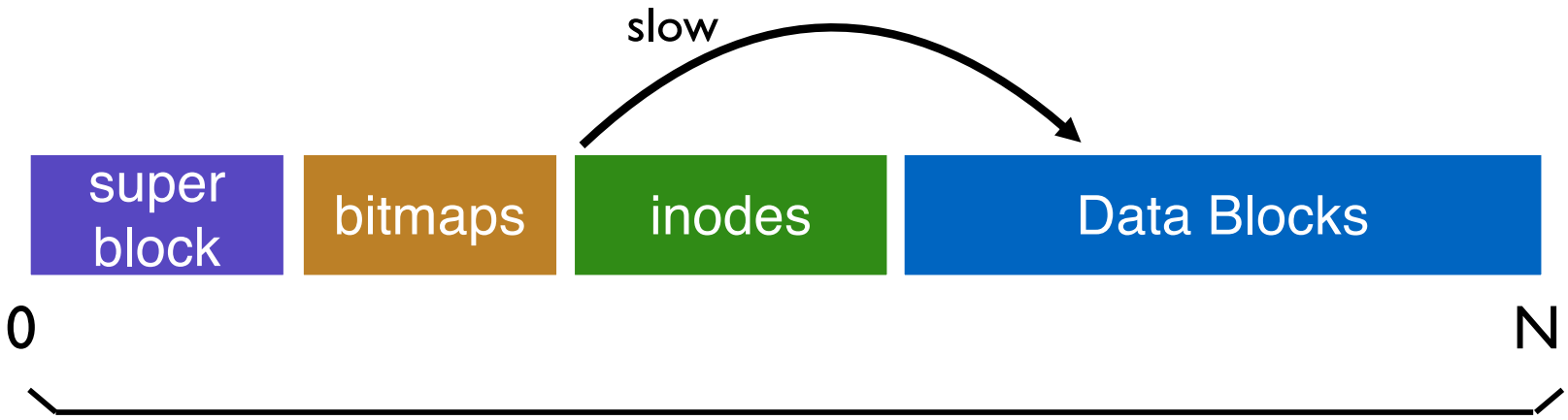
Time vs. Space tradeoffs…

MEASUREMENT 3: BLOCK LAYOUT

Blocks laid out poorly

- long distance between inodes / data
- related inodes not close to one another

FILE LAYOUT IMPORTANCE



Layout is not disk-aware!

OLD FS SUMMARY

1. Free list becomes scrambled → random allocations
2. Small blocks (512 bytes)
3. Blocks laid out poorly
   - long distance between inodes/data
   - related inodes not close to one another

Result: **2%** of potential performance!

Problem: Old FS treats disk like RAM!

DISK-AWARE FILE SYSTEM

How to make the disk use more efficient?

Where to place meta-data and data on disk?

PLACEMENT TECHNIQUE 1: BITMAPS

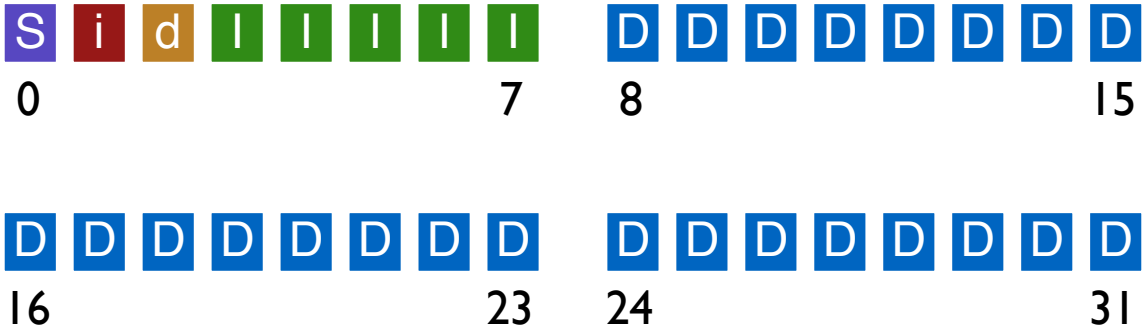
Use bitmaps instead of free list
Bitmaps provides better speed, with more global view

Fast to update: Just flip single bit to change state from free -> allocated or back
No sorting needed
Fast to do lookups: Reading one bitmap block gives state of many data blocks
Quickly find contiguous free blocks

FS STRUCTS: BITMAPS

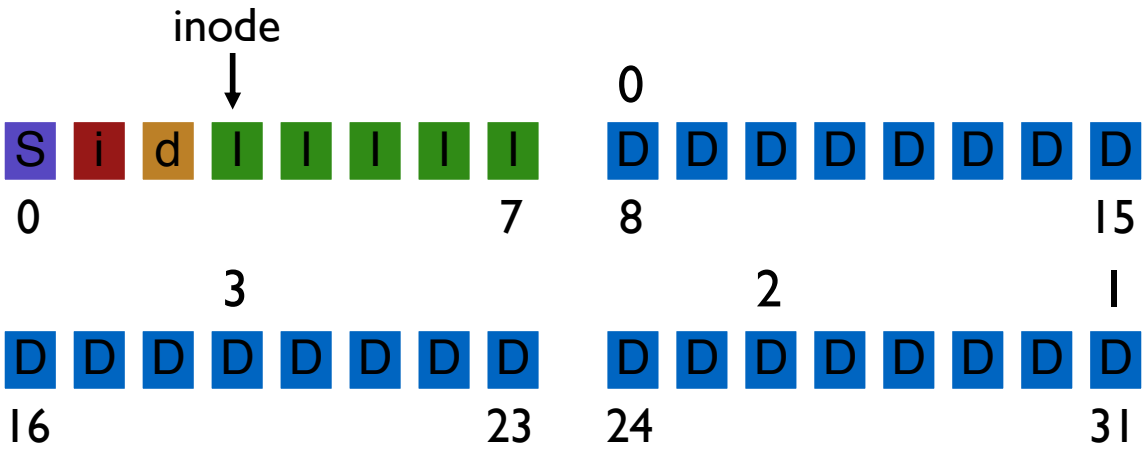
Separate Data bitmap and inode bitmap

TECHNIQUE 2: LAYOUT POLICY

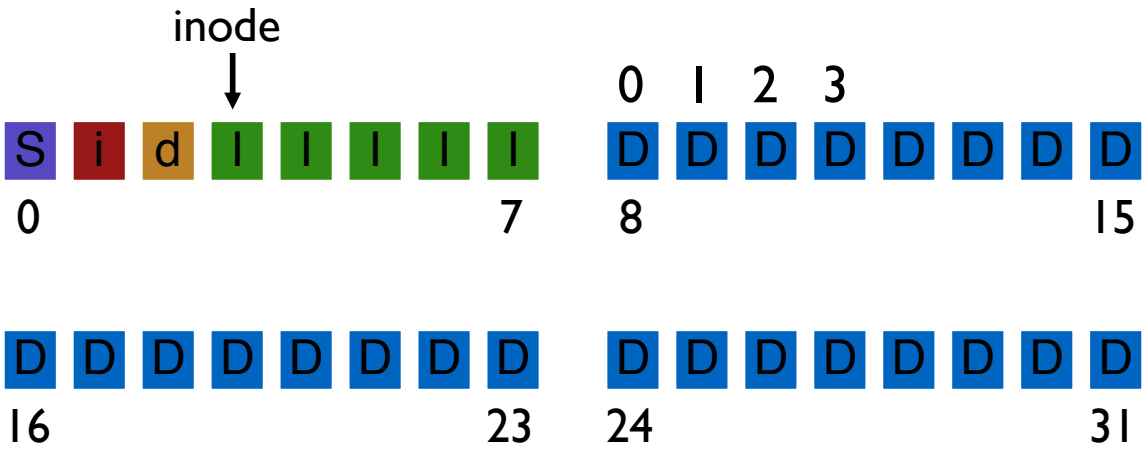


Assuming all free, which should be chosen?

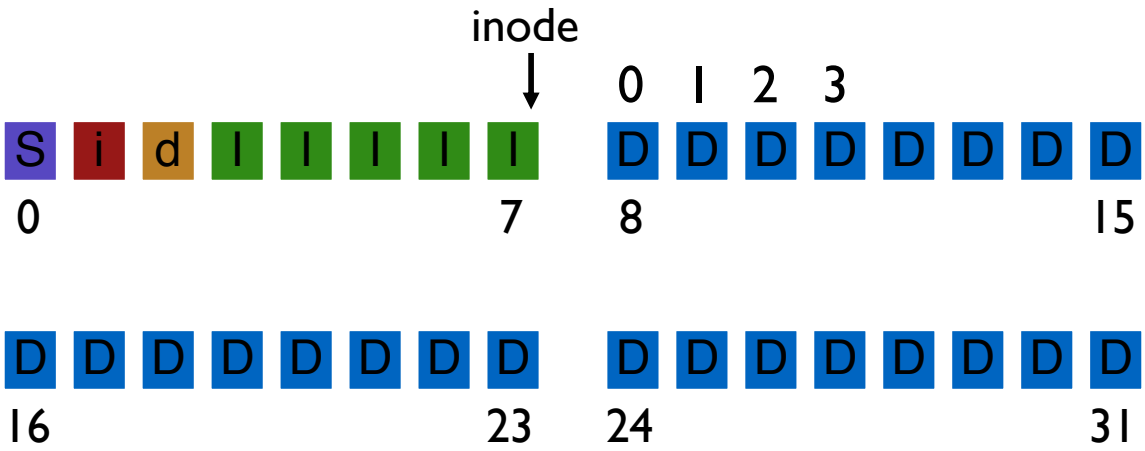
BAD FILE LAYOUT



BETTER FILE LAYOUT

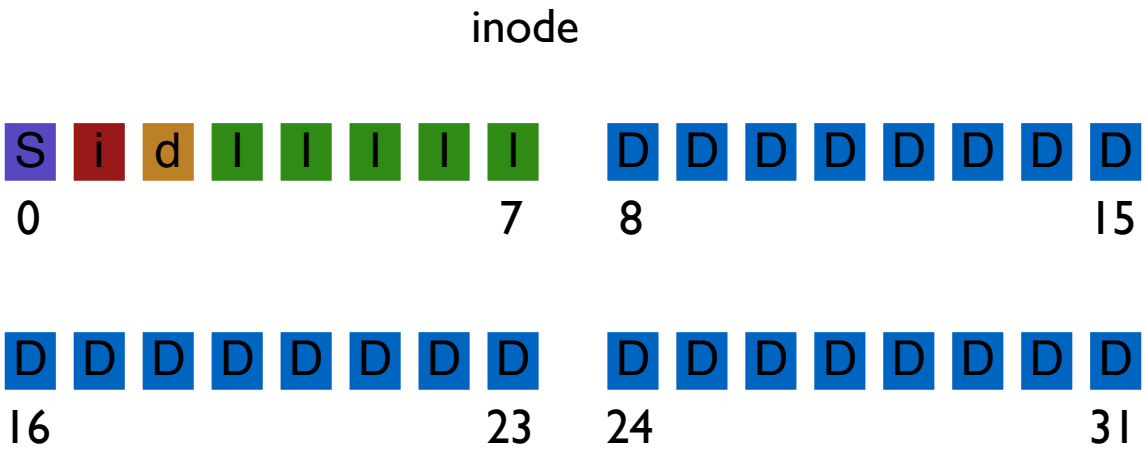


BEST FILE LAYOUT



Can't do this for all files ☹

INODE LAYOUT MATTER?



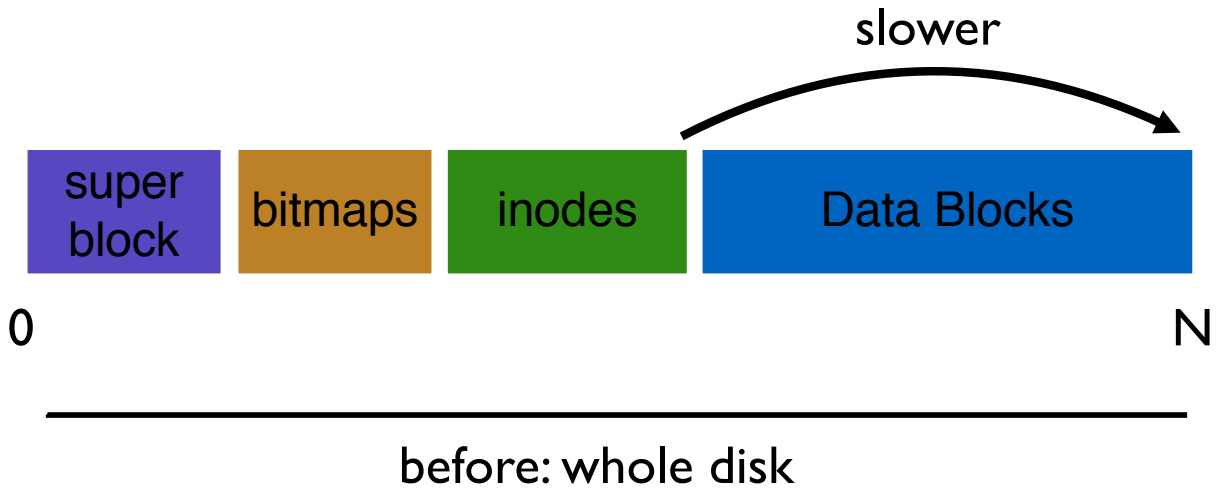
What does FS do for ls? Read directory data for file names

What does FS do for ls –l? Read inodes of all files to show size and other metadata

Keep data near its inode

Inodes in same directory should be near one another

TECHNIQUE 2: GROUPS



How to keep inode close to data?

PLACEMENT TECHNIQUE: GROUPS

Key idea: Keep inode close to data

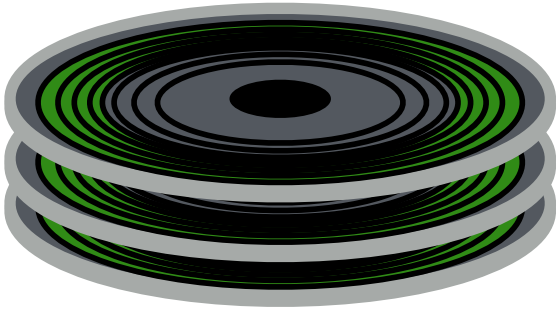
Use groups across disks;

Strategy: allocate inodes and related data blocks in same group

PLACEMENT TECHNIQUE: GROUPS

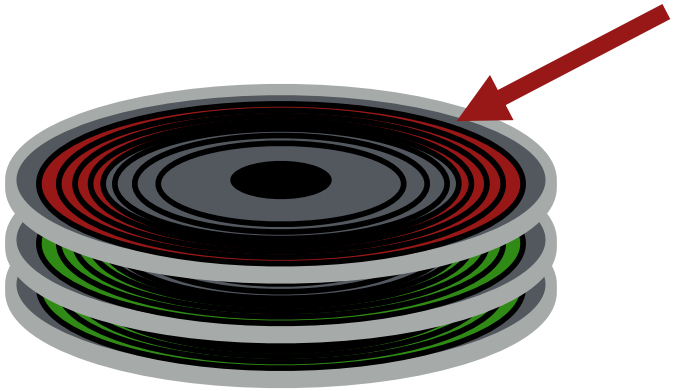
In FFS, groups were ranges of cylinders
    called cylinder group

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



REPLICATED SUPER BLOCKS

Is it useful to have multiple 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?

PLACEMENT STRATEGY

Put related pieces of data near each other (i.e., in same group)

Rules:

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

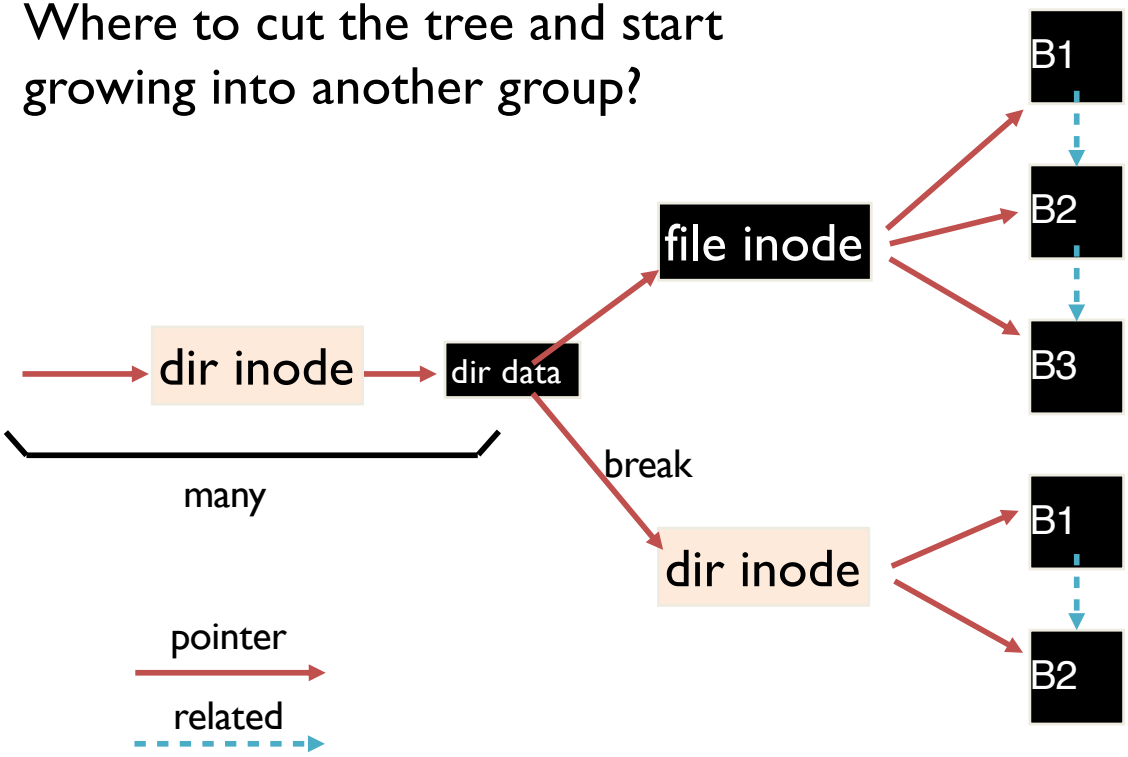
Problem: File system is one big tree

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!

Where to cut the tree and start growing into another group?



FFS puts dir inodes in a new group

"`ls -l`" is fast on directories with many **files**

POLICY SUMMARY

File inodes: allocate in same group with dir

Dir inodes: allocate in new group

Which group to pick?

One with with fewer used inodes than average group

First data block: allocate near inode (same group)

Other data blocks: allocate near previous block (in case had to move groups)

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 one large file than
one seek for each of many small files

SPLITTING LARGE FILES

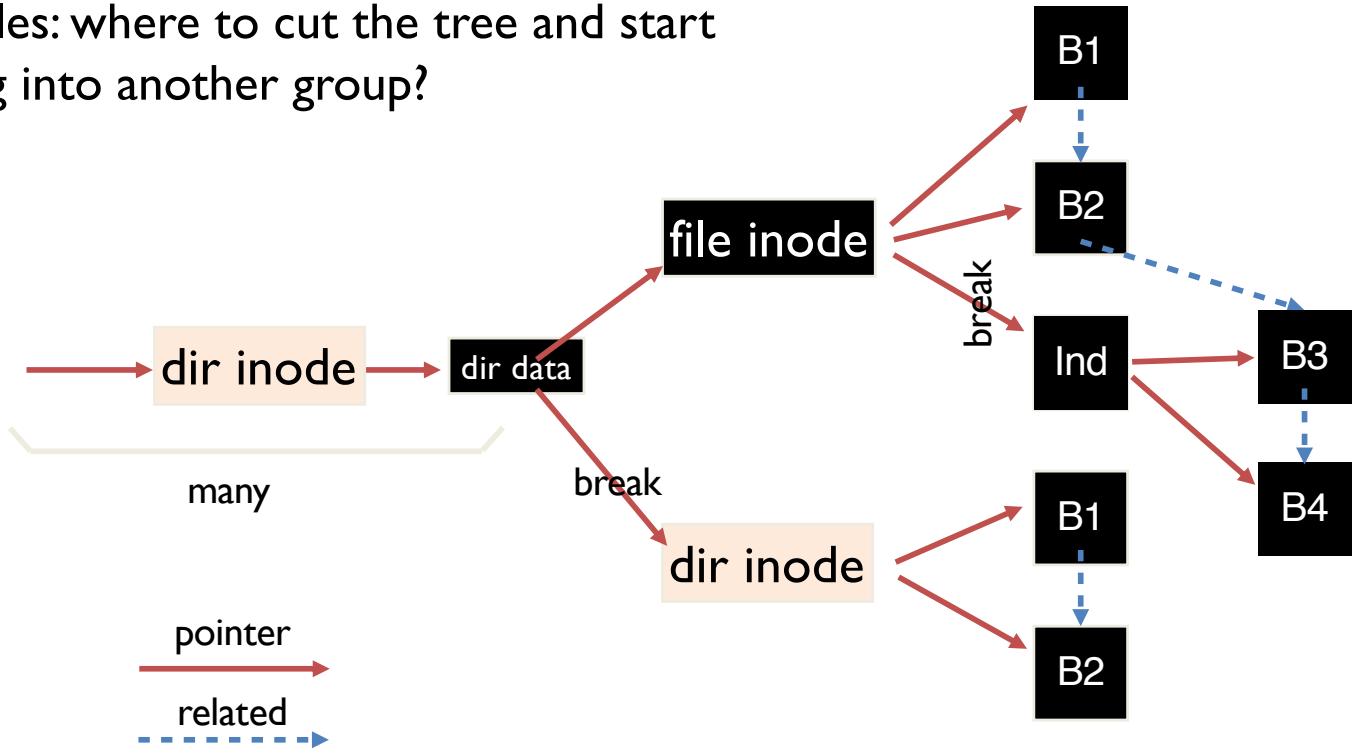
```
group inodes     data
    0 /a-------- /aaaaa---- ---------- ---------- ----------
    1 ---------- aaaaa----- ---------- ---------- ----------
    2 ---------- aaaaa----- ---------- ---------- ----------
    3 ---------- aaaaa----- ---------- ---------- ----------
    4 ---------- aaaaa----- ----------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

Define "large" as requiring an indirect block

Starting at indirect (e.g., after first 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"

Large files: where to cut the tree and start growing into another group?



Define "large" as requiring an indirect block

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

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 chunk (e.g., 1MB or 4MB).

GROUP DESCRIPTOR (AKA SUMMARY BLOCK)

How does file system know which new group to pick?

Tracks number of free inodes and data blocks

Something else to update on writes

OTHER FFS FEATURES

FFS also introduced several new features:

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

LARGE OR SMALL BLOCKS

- Large blocks:
  - Improve performance
  - Waste space (most files are small)
- Small blocks
  - Don't waste as much space
  - Bad performance (more seeks)
- How to get best of both worlds? Hybrid solution?

SOLUTION: FRAGMENTS

Hybrid Solution

- Combine best of large blocks and best of small blocks

Use large block when file is large enough

Introduce “fragment” for files that use parts of blocks

- Only tail of file uses fragments

FRAGMENT EXAMPLE

Block size = 4096

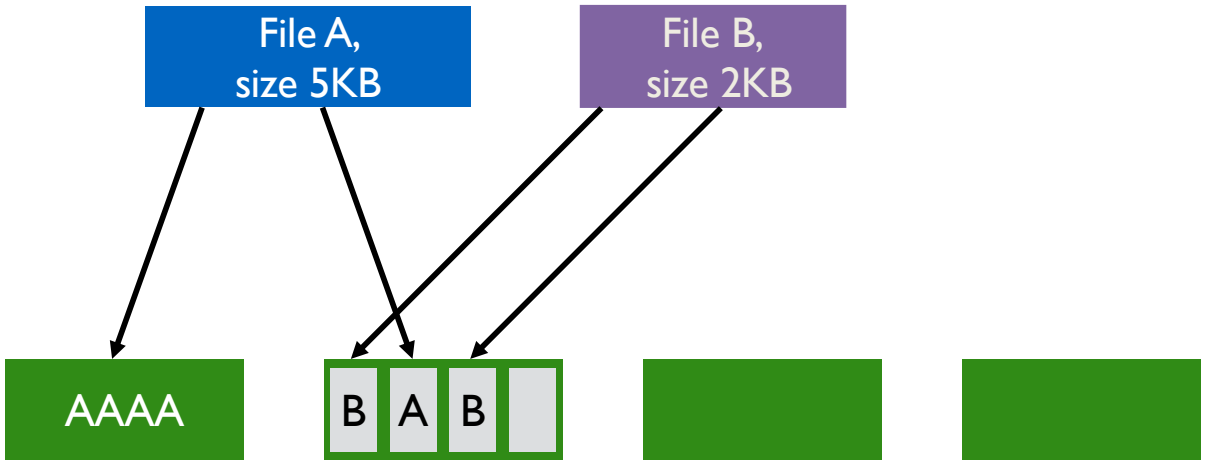
Fragment size = 1024

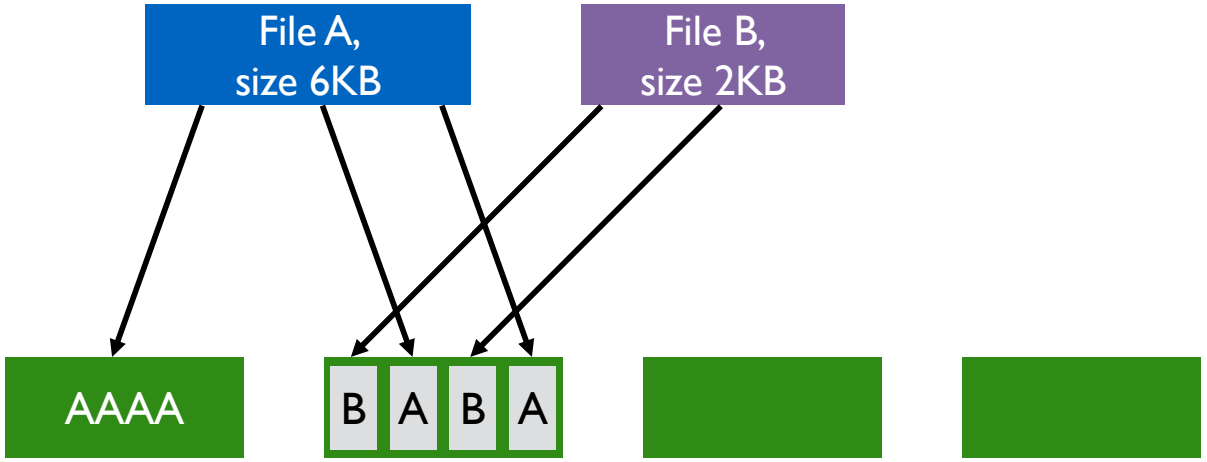
bitmap:	0000	0000	1111	0010
	blk1	blk2	blk3	blk4

Whether addr refers to block or fragment is inferred by file offset

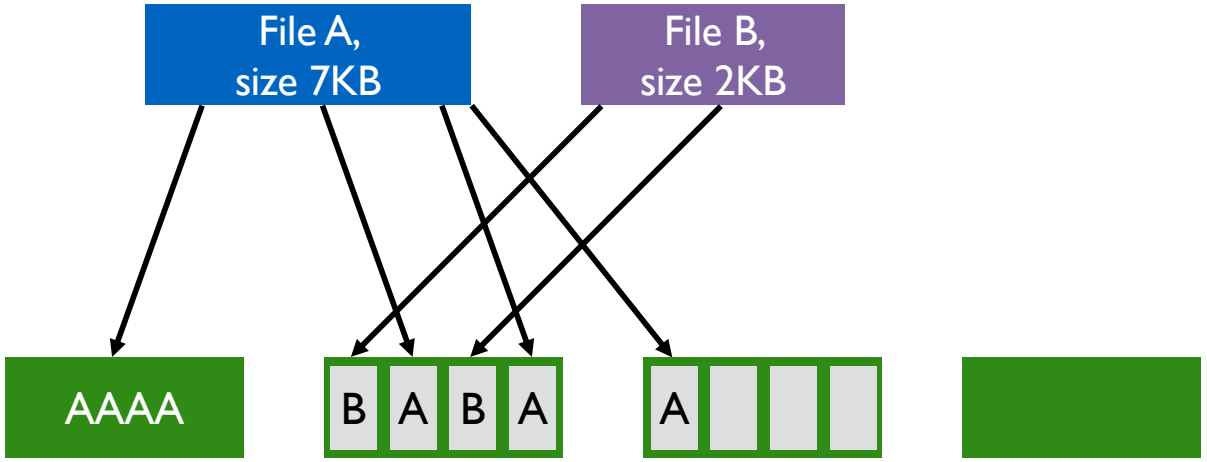
What about when files grow?

Must copy fragments to new block if no room to grow


File A,
size 5KB
File B,
size 2KB
AAAA
B
A
B



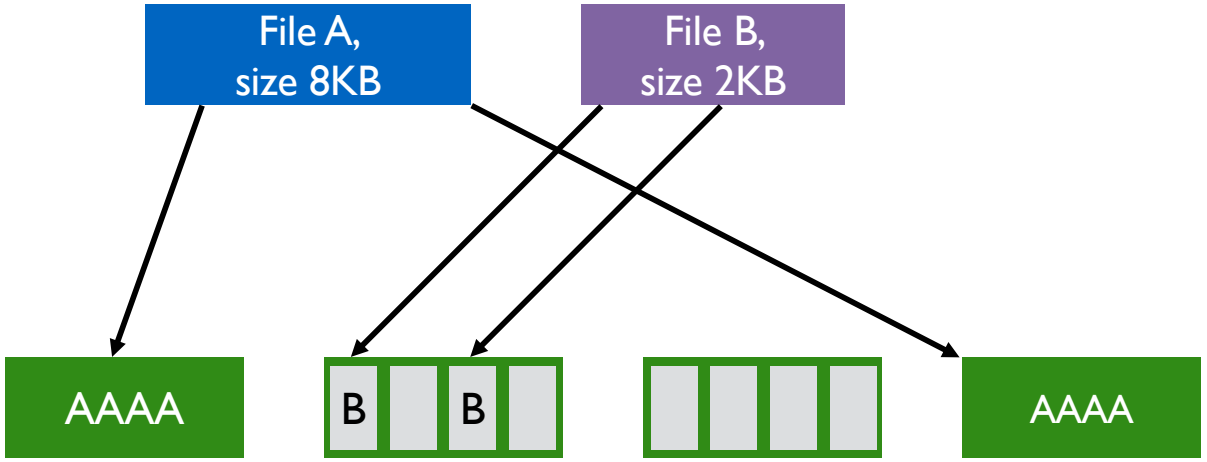
append A to first file



append A to first file

Not allowed to use fragments across multiple blocks!

What to do instead?



append A to first file,
copy to fragments to new block

OPTIMAL WRITE SIZE

Writing less than a full block is inefficient

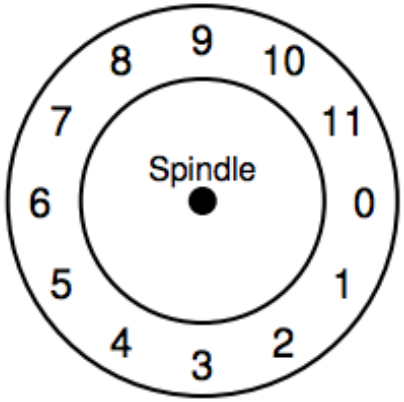
- Potentially requires moving fragments
- Small writes don't get full sequential bandwidth of disk

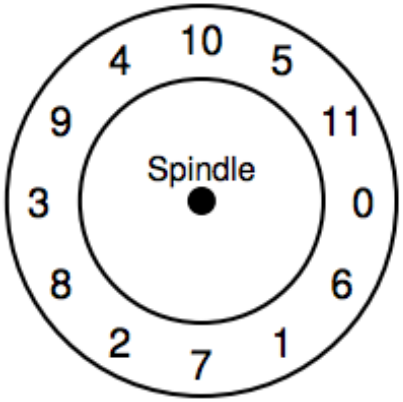
Solution:

New API exposes optimal write size

Library enables caching writes in memory until large enough size

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?