

ANNOUNCEMENTS

P3 Grading: Available

- Look at runtests.log for details
- Contact TA (give login and discussion section) with questions

P4: Threads (Part a and b) available

- Due Friday 11/18 at 9pm – really Sunday 11/27 9pm (no request needed)

Exam 2: Answers with explanations available

Read as we go along!

- Chapter 41

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

PERSISTENCE: FAST FILE SYSTEM (FFS)

Questions answered in this lecture:

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?

FILE-SYSTEM CASE STUDIES

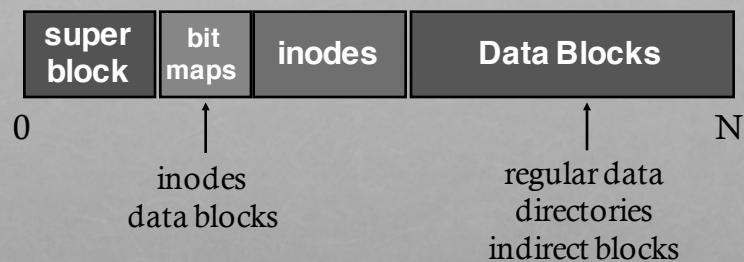
Local

- **FFS**: Fast File System
- **LFS**: Log-Structured File System

Network

- **NFS**: Network File System
- **AFS**: Andrew File System

REVIEW: BASIC LAYOUT



What is stored as a data block?

REVIEW: create /foo/bar

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

create /foo/bar

[traverse path]

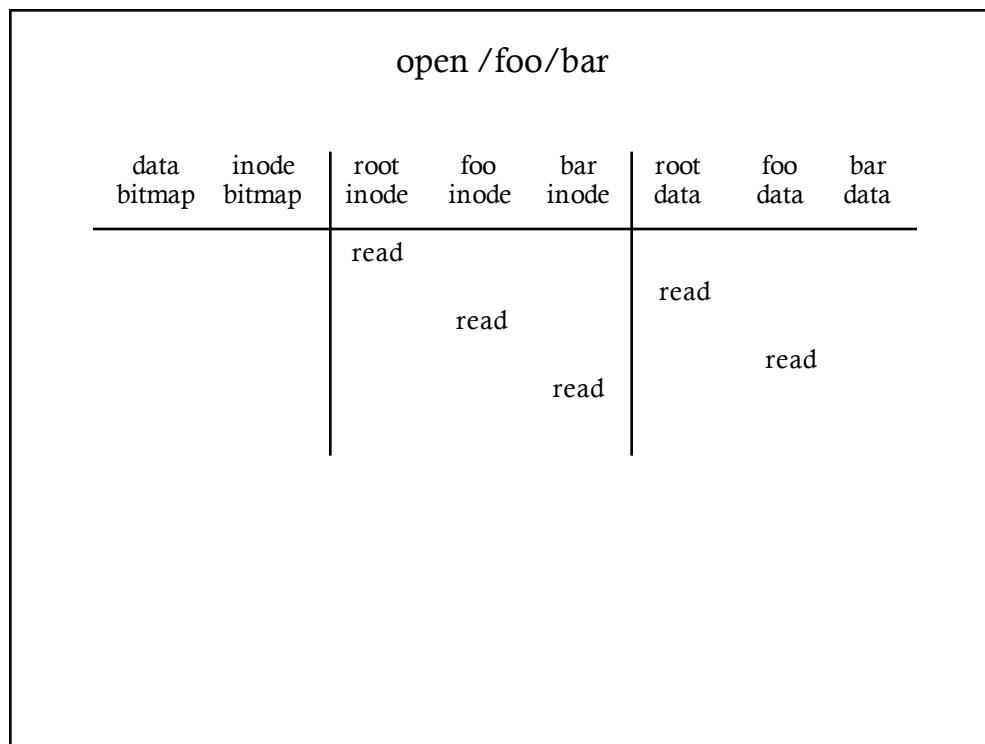
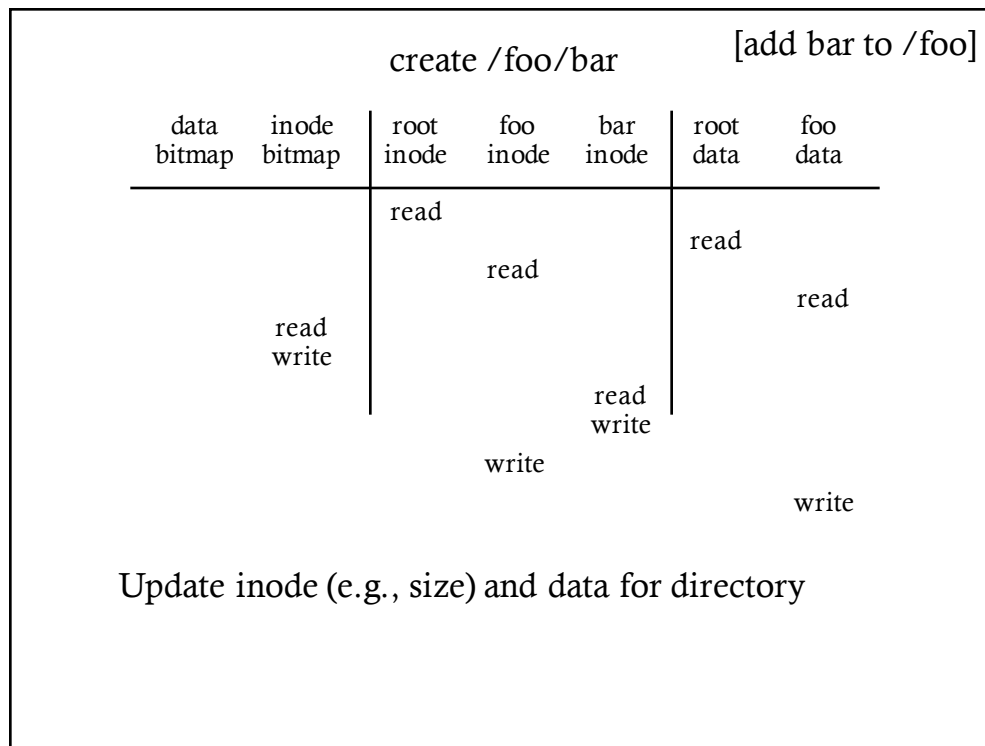
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read			read	
			read			
						read

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 write					read
				read write		

Why must **read** bar inode?
How to initialize inode?



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

No reads or writes to / or /foo

append to /foo/bar (opened already)

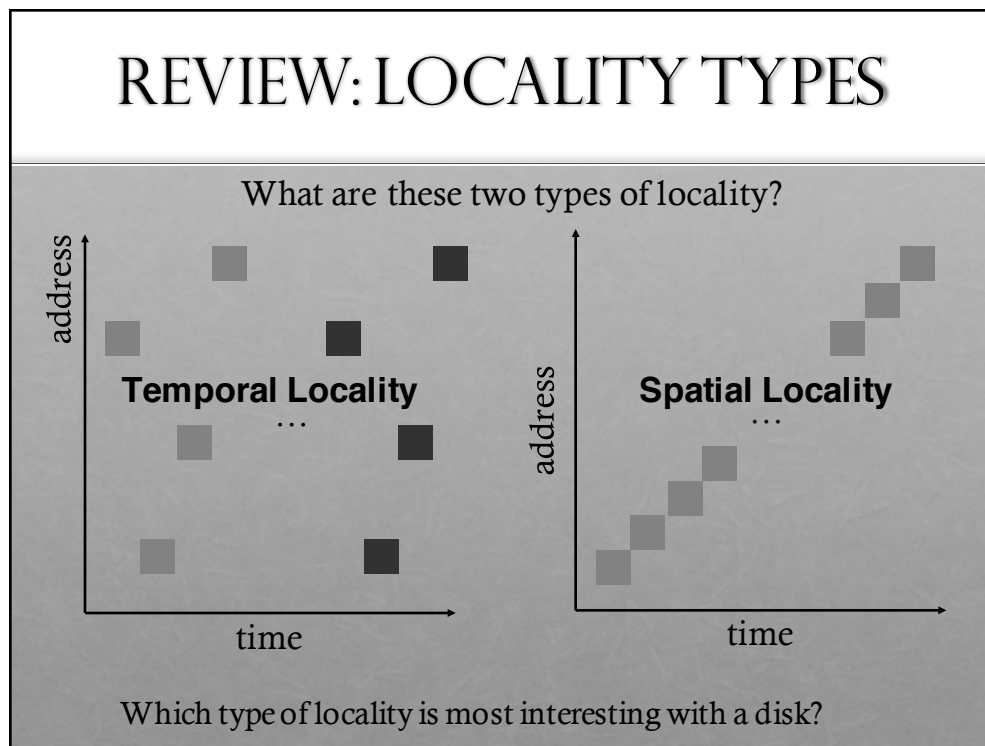
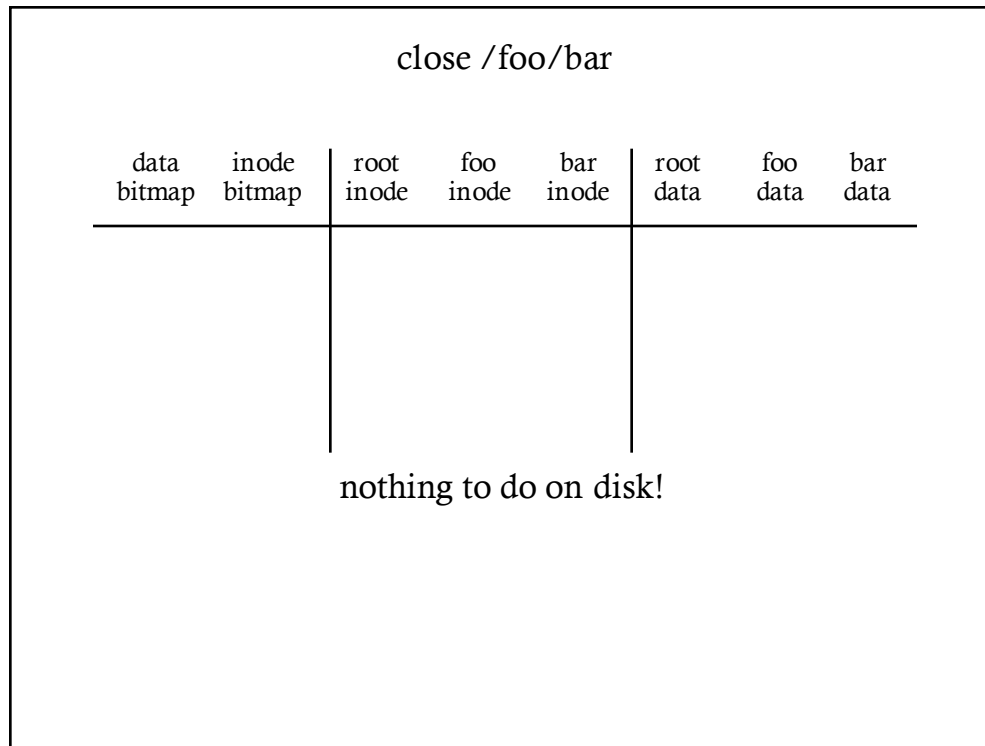
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			

append to /foo/bar					[allocate block]		
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
read write				read			

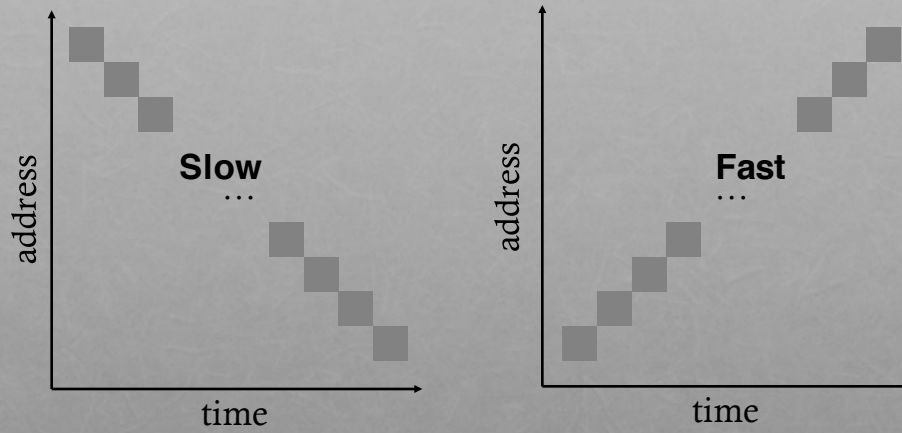
append to /foo/bar					[point to block]		
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
read write				read			
				write			

append to /foo/bar					[write to block]		
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
read write				read			
				write			
							write

read /foo/bar – assume opened							
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			read
				write			



ORDER MATTERS



Implication for OS-level disk schedulers?

FAST FILE SYSTEM:
FFS
(1980'S)

SYSTEM BUILDING

Beginner's approach

1. get idea
2. build it!

Pro approach

measure then build

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!

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

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

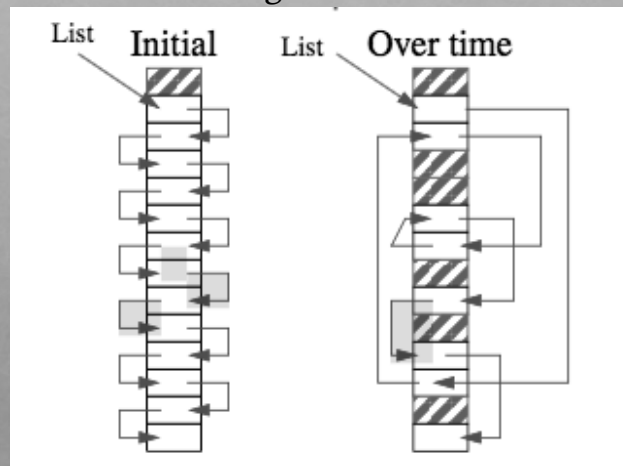
Problem: FS becomes fragmented over time

- Free list makes contiguous chunks hard to find

MEASUREMENT 1: AGING?

Problem: FS becomes fragmented over time

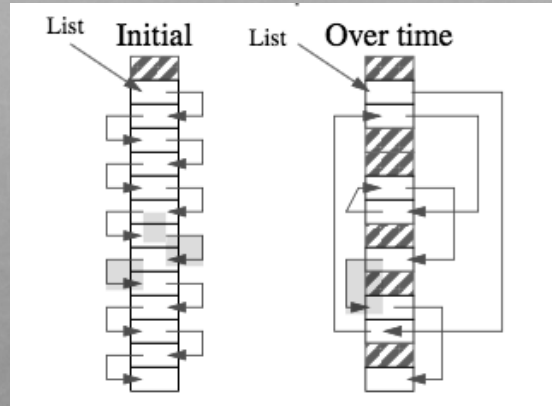
- Free list makes contiguous chunks hard to find



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 not physically adjacent
- Only half as many seeks+rotations now required

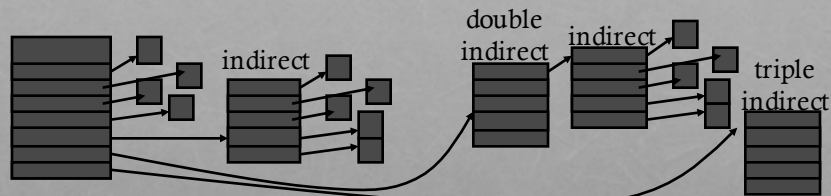
Why **more** than double the performance?

- Smaller blocks require more indirect blocks

MULTI-LEVEL INDEXING

Variation of Indexed Allocation

- 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



Comparison to Indexed Allocation

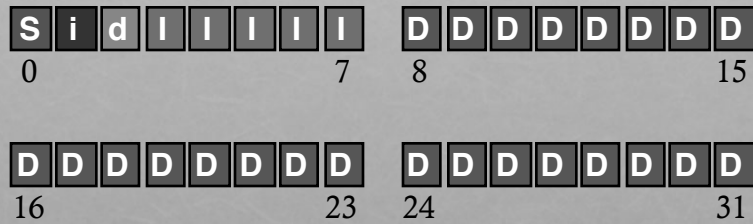
- Advantage: Does not waste space for unneeded pointers
 - Still fast access for small files
 - Can grow to what size??
- Disadvantage: Need to read indirect blocks of pointers to find addresses (extra disk read)
 - Keep indirect blocks cached in main memory (esp for sequential)

MEASUREMENT 3: BLOCK LAYOUT

Blocks laid out poorly

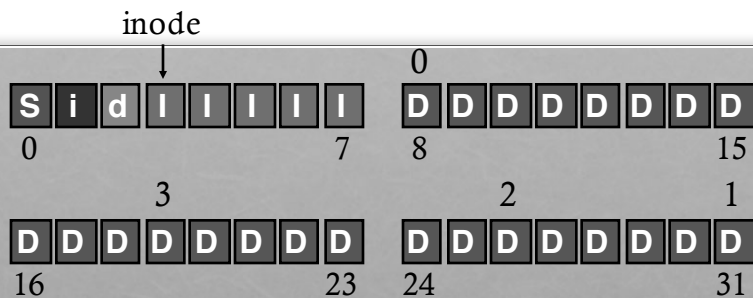
- long distance between inodes/data
- related inodes not close to one another

POLICY: CHOOSE INODE, DATA BLOCKS

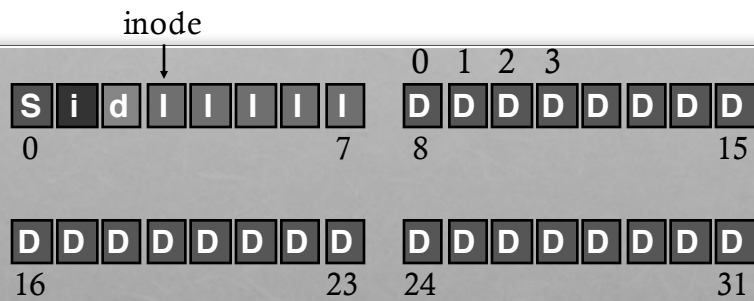


Assuming all free, which should be chosen?

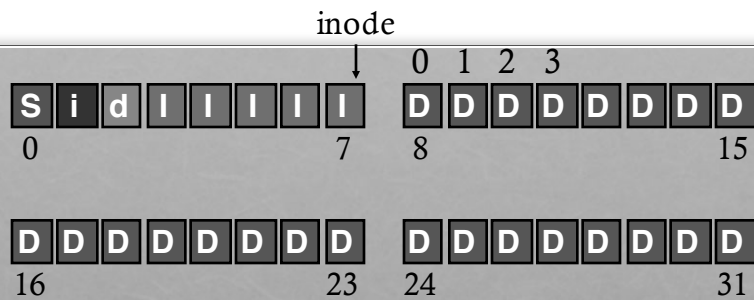
BAD FILE LAYOUT



BETTER FILE LAYOUT

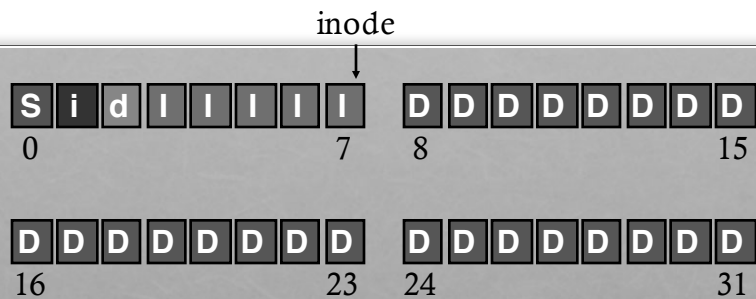


BEST FILE LAYOUT



Can't do this for all files ☹

INODE LAYOUT MATTER?



What does FS do for ls?
What does FS do for ls-l?

Inodes in same directory should be near one another

OLD FS SUMMARY

Free list becomes scrambled → random allocations

Small blocks (512 bytes)

Blocks laid out poorly

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

Result: **2%** of potential performance! (and worse over time)

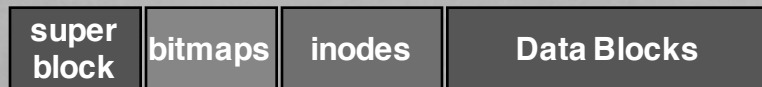
Problem: Old FS treats disk like RAM!

SOLUTION: A DISK-AWARE

Primary File System Design Questions:

- Where to place meta-data and data on disk?
- How to use big blocks without wasting space?

PLACEMENT TECHNIQUE 1: BITMAPS

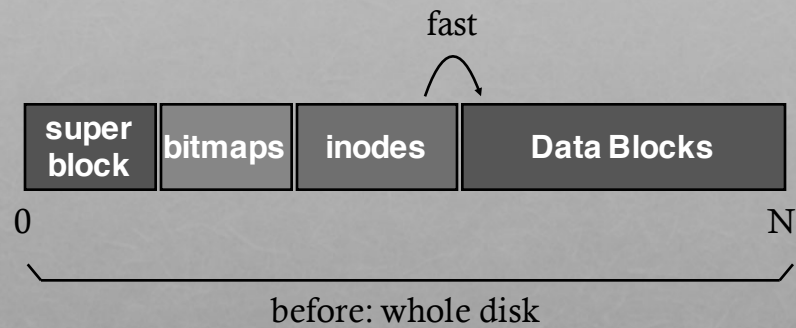


0 N

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

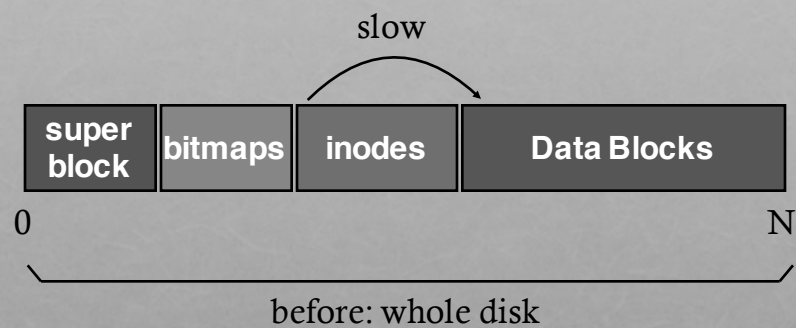
Faster to find contiguous free blocks

PLACEMENT TECHNIQUE 2: GROUPS



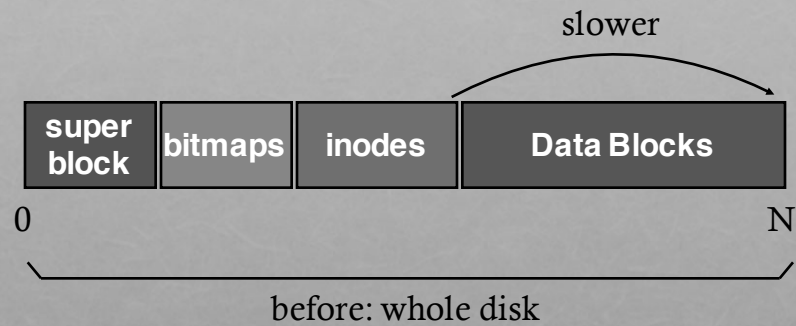
How to keep inode close to data?

TECHNIQUE 2: GROUPS



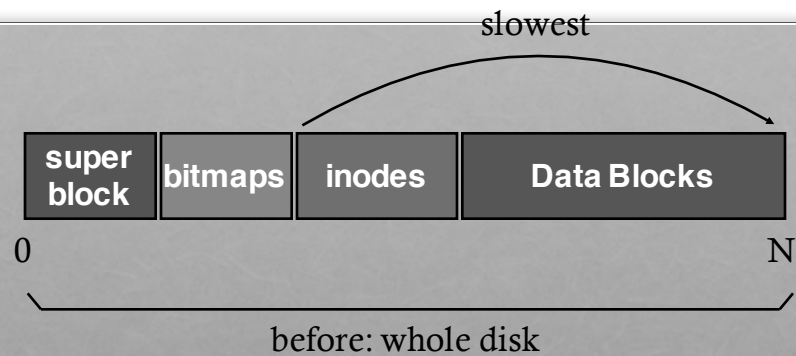
How to keep inode close to data?

TECHNIQUE 2: GROUPS



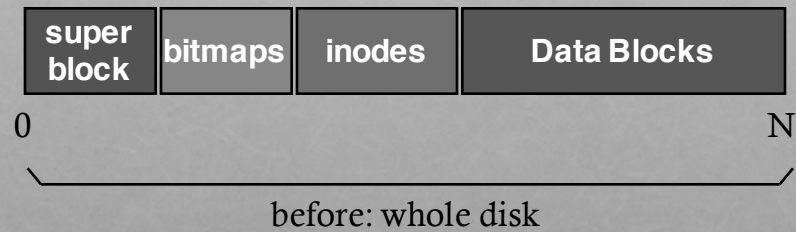
How to keep inode close to data?

TECHNIQUE 2: GROUPS



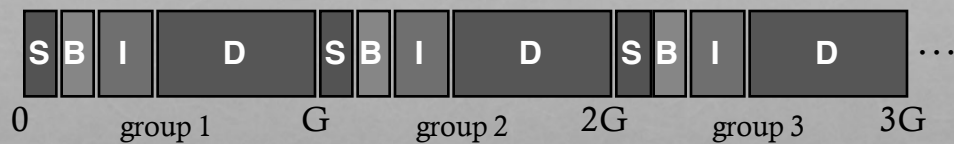
How to keep inode close to data?

TECHNIQUE 2: GROUPS



How to keep inode close to data?

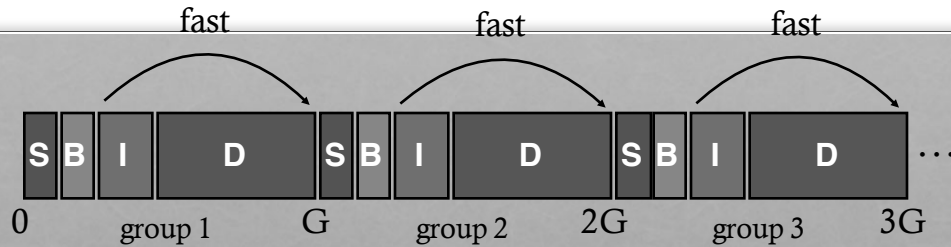
TECHNIQUE 2: GROUPS



How to keep inode close to data?

Answer: Use **groups** across disks;
Try to place inode and data in same group

TECHNIQUE 2: GROUPS



strategy: allocate inodes and data blocks in same group

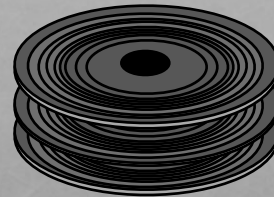
GROUPS

In FFS, groups were ranges of cylinders

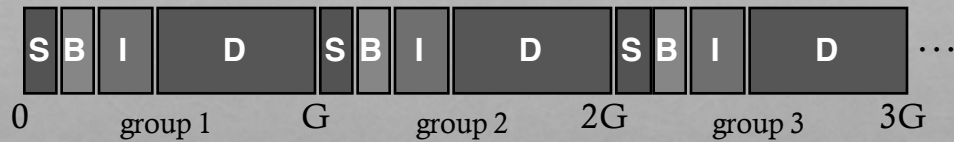
- called cylinder group

In ext2-4, groups are ranges of blocks

- called block group

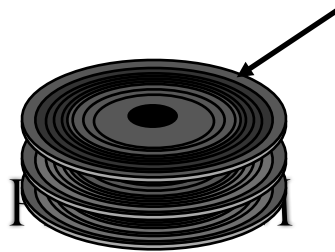


PLACEMENT TECHNIQUE 3: SUPER ROTATION



Is it useful to have multiple super blocks?

Yes, if some (but not all) fail.



Old FS: All super-block copies are on the top platter
Correlated failures! What if top platter dies?

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

TECHNIQUE 4: BLOCK SIZE

Observation: Doubling the block size for the old FS over doubled performance.

Strategy: choose block size so never read more than two indirect blocks (i.e., double indirect) to reach data block.

With 4KB block size, how large of a file can they support?

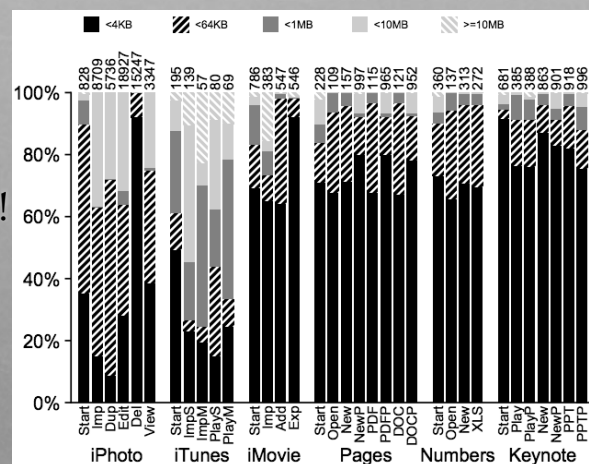
$$(\text{Blocksize} / 4 \text{ bytes}) * (\text{Blocksize} / 4 \text{ bytes}) * \text{Blocksize} = 4 \text{ GB}$$

$$\text{Blocksize}^3 = 256 \text{ MB}$$

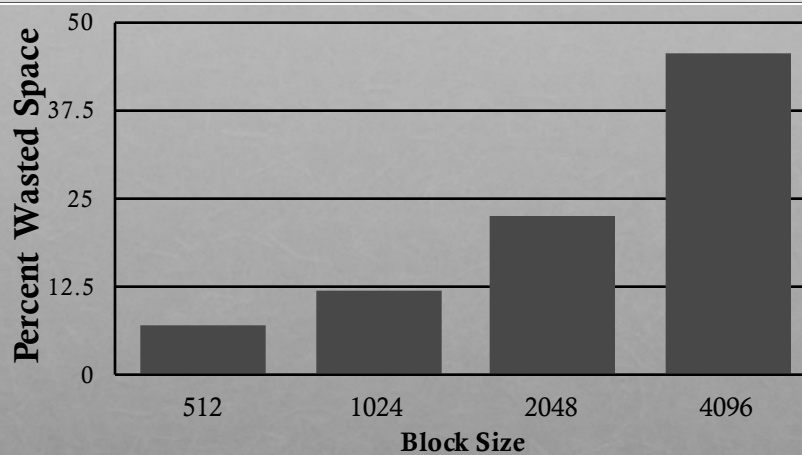
TECHNIQUE: LARGER BLOCKS

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...

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

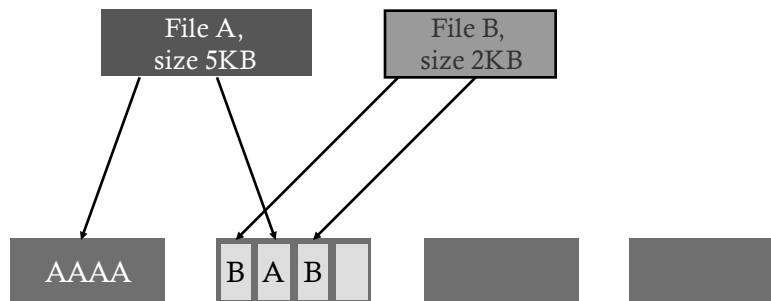
Fragment size = 1024

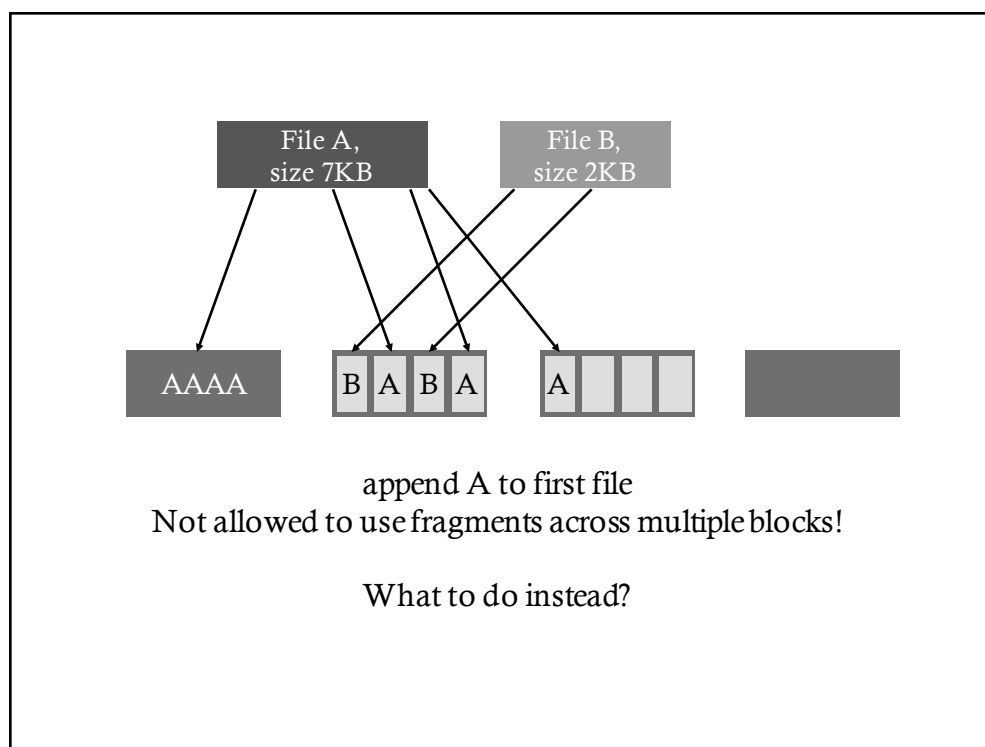
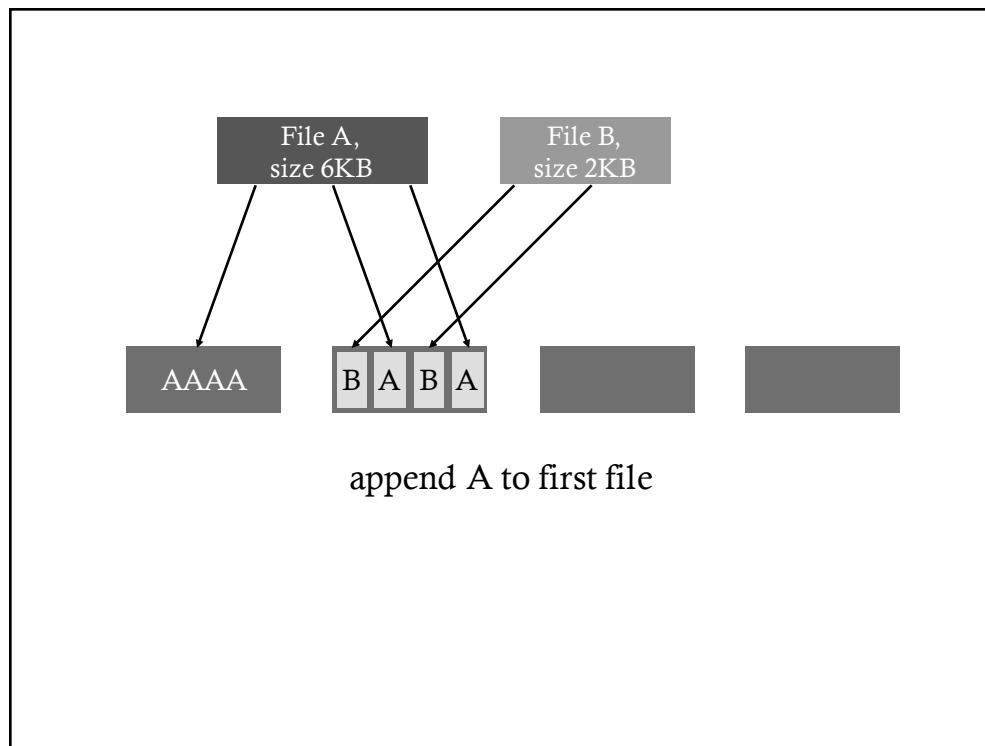
bitmap: 0000 0000 1111 0010
 blk1 blk2 blk3 blk4

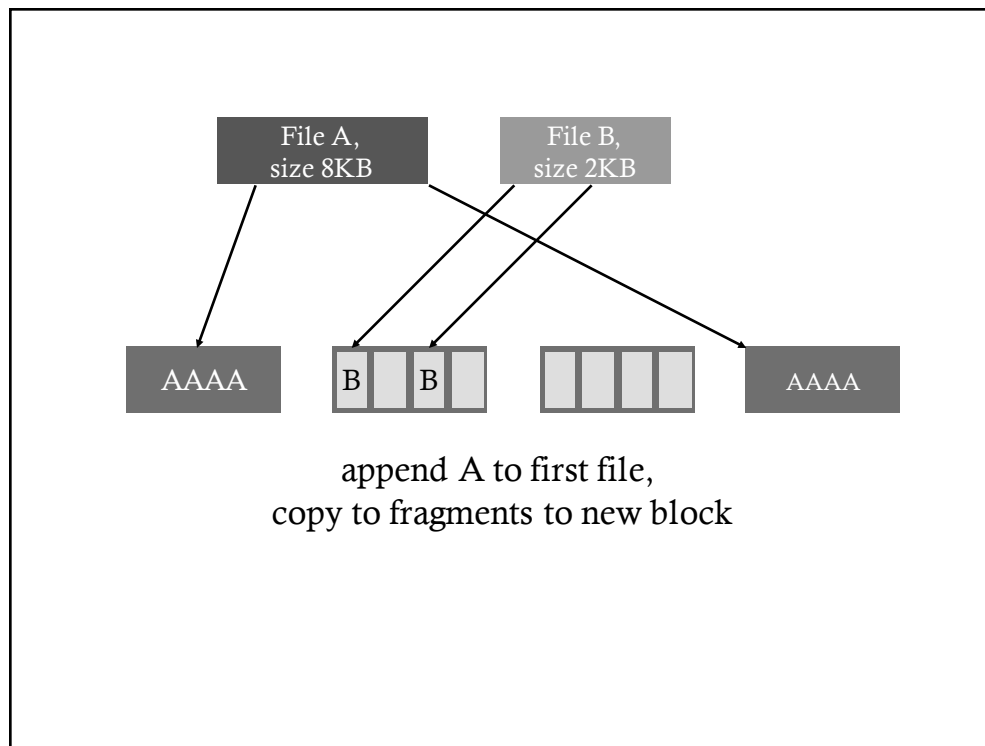
Whether addr refers to block or fragment is inferred by file offset

What about when files grow?

Must copy fragments to new block if no room to grow







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

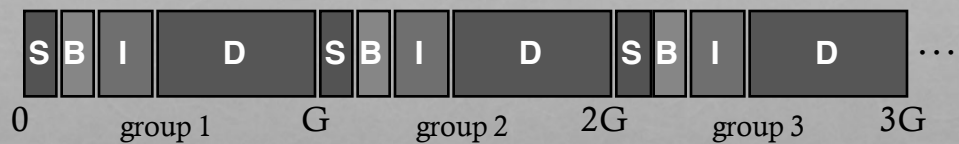
Solution:

New API exposes optimal write size

BREAK

- What do you plan to do for Thanksgiving?
- What do you wish you were doing for Thanksgiving?

SMART POLICY



Where should new inodes and data blocks go?

PLACEMENT STRATEGY

Put related pieces of data near each other

Rules:

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

Sound good?

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!

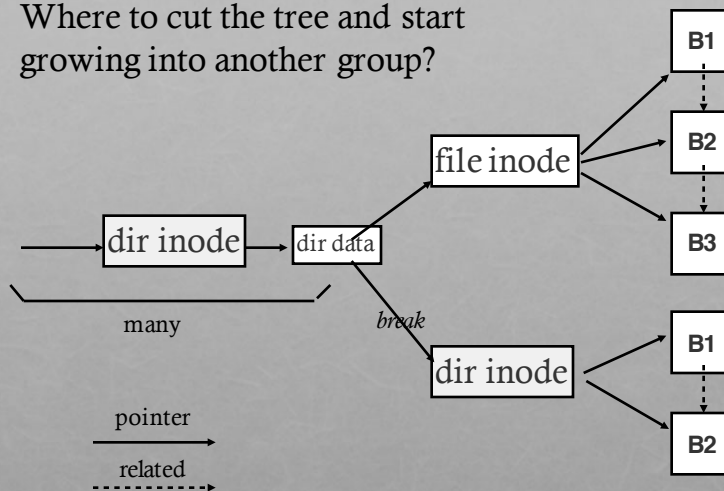
REVISED STRATEGY

Put more-related pieces of data near each other

Put less-related pieces of data **far** from each other

FFS developers used their best judgement

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**

PREFERENCES

File inodes:

- Allocate in same group with directory data and inode

Dir inodes:

- Allocate in new group
- Which new group to pick?
- One with more free inodes or more data blocks than average group?
 - Pick with fewer used inodes than average group

First data block (file and directory):

- Allocate near inode

Other data blocks:

- Allocate near previous block

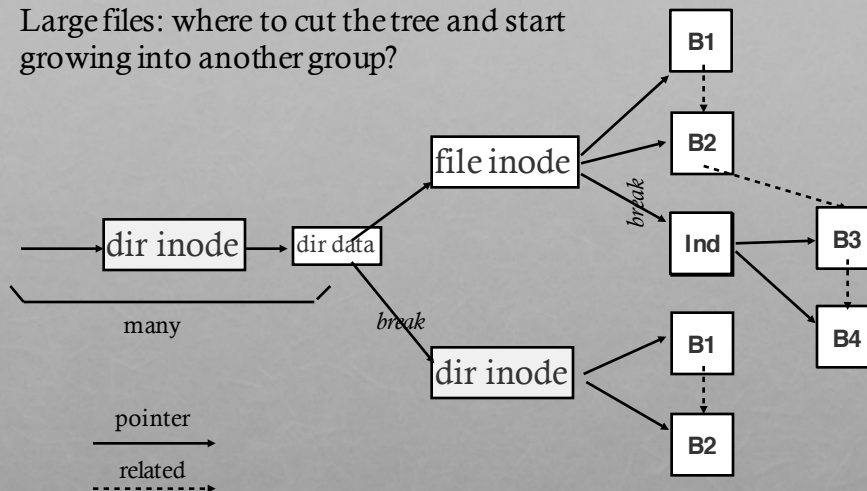
PROBLEM: LARGE FILES

Single large file can fill nearly all of a group

Displaces data for many small files

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

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

PREFERENCES

File inodes:

- Allocate in same group with directory data and inode

Dir inodes:

- Allocate in new group
- Pick with fewer used inodes than average group

First data block (file and directory):

- Allocate near inode

Other data blocks:

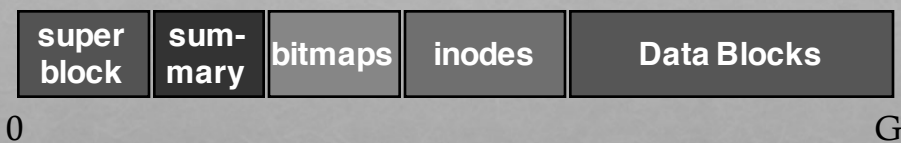
- Allocate near previous block

Large file data blocks:

- after 48KB, go to new group. Move every subsequent 1MB.
- How to pick new group?
 - Pick with fewer used data blocks than average group

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

FFS CONCLUSION

First disk-aware file system

- Bitmaps
- Locality groups
- Rotated superblocks
- Large blocks with Fragments
- Smart allocation policy

FFS inspired modern files systems, including ext2 and ext3

FFS also introduced several new features:

- long file names
- atomic rename
- symbolic links

ADVICE

All hardware is unique

Treat disk like disk!

Treat flash like flash!

Treat random-access memory like random-access memory!