Recap

File System Implementation – Data Structures

- Data blocks (fixed size)

- Metadata: Superblock, inode bitmap, data bitmap, inodes
  - Data block Indexes in inodes: direct pointers; single/double/triple indirect pointers

Example Reading a File

How to read from the file /foo/bar? First, fd=open("/foo/bar").

1. inode number for root directory is well known; read "root inode" and get the location for "root data".
2. Read "root data" to get inode number for "/foo".
3. Read "foo inode" to get location of "foo data".
4. Read "foo data" to get inode number for "/foo/bar".
5. Read "bar inode" and bring the metadata of /foo/bar to main memory.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Example Reading a File

How to read from the file /foo/bar?
Second, read(fd,...).

1. Read "bar inode" to get the location of the first data block of /foo/bar, i.e., "bar data [0]".
2. Read "bar data [0]".
3. Update last access time at "bar inode".
4. Read "bar inode" to get the location of the second block of /foo/bar, i.e., "bar data[1]".
5. Read "bar data [1]".
6. Update last access time at "bar inode".
7. ... ...

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**							
						read				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

Example: Creating and Writing File

How to create file /foo/bar?

1. Read "root inode" (i.e., inode for "/") to get location of "/".
2. Read data block of "/" (i.e., "root data") to get inode number of "/foo".
3. Read "foo inode" to get location of "/foo".
4. Read "/foo" (i.e., "foo data"); read inode bitmap to find an empty inode, denoted as x, and update the bitmap; add a pair ("bar", x) to "foo data".
5. Read "bar inode" (i.e., inode with number x) and initialize it.
6. Update last access time at "foo inode".

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					
write()					read					
	read									
	write									
									write	
					write					
write()					read					
	read									
	write									
										write
					write					

Example: Creating and Writing File

How to write to new file /foo/bar?

1. Read "bar inode".
2. Read data bitmap to find an empty data block x and update the bitmap.
3. Write to data block x (i.e., "bar data [0]").
4. Add block x to "bar inode".
5. Update the last access time at "bar inode"
6. Read "bar inode".
7. Read data bitmap to find an empty data block y and update the bitmap.
8. Write to data block y (i.e., "bar data [1]").
9. Add block y to "bar inode".
10. … …

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)			**read**							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					**read**					
	read									
	write									
								write		
					write					
write()					**read**					
	read									
	write									
									write	
					write					
write()					**read**					
	read									
	write									
										write
					write					

Basic Performance Improvements

Caching – holds popular blocks to decrease number of times blocks are read from disk

Write buffering - batch multiple updates into a smaller set of I/O operations

Fast File System

Based on Ch. 41

Slides revised from
Matthew Tancreti
Iowa State University

Faster File System

Original UNIX file system had very poor performance (used only 2% of disk bandwidth)

The most significant physical limitation is the difference between random and sequential latencies

Optimizations need to take the disk into account

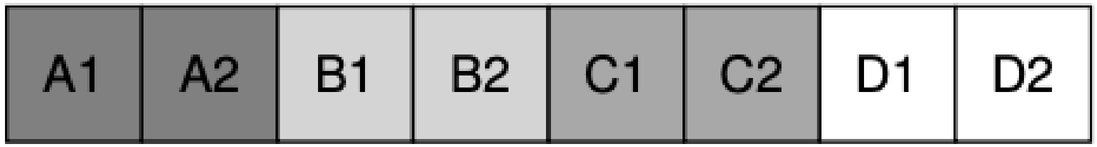
How to improve file system performance?

Consequence of Fragmentation

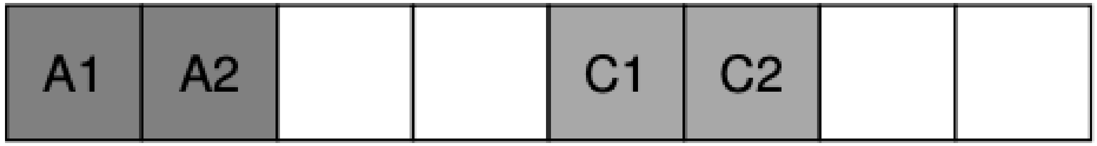
Block based allocation means files can become spread out over the disk

Best performance is when files are written to contiguous memory

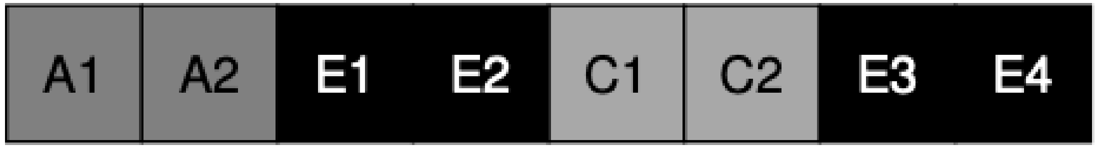
1. Assume files A, B, C and D are stored on disk



2. B and D are deleted leaving two gaps

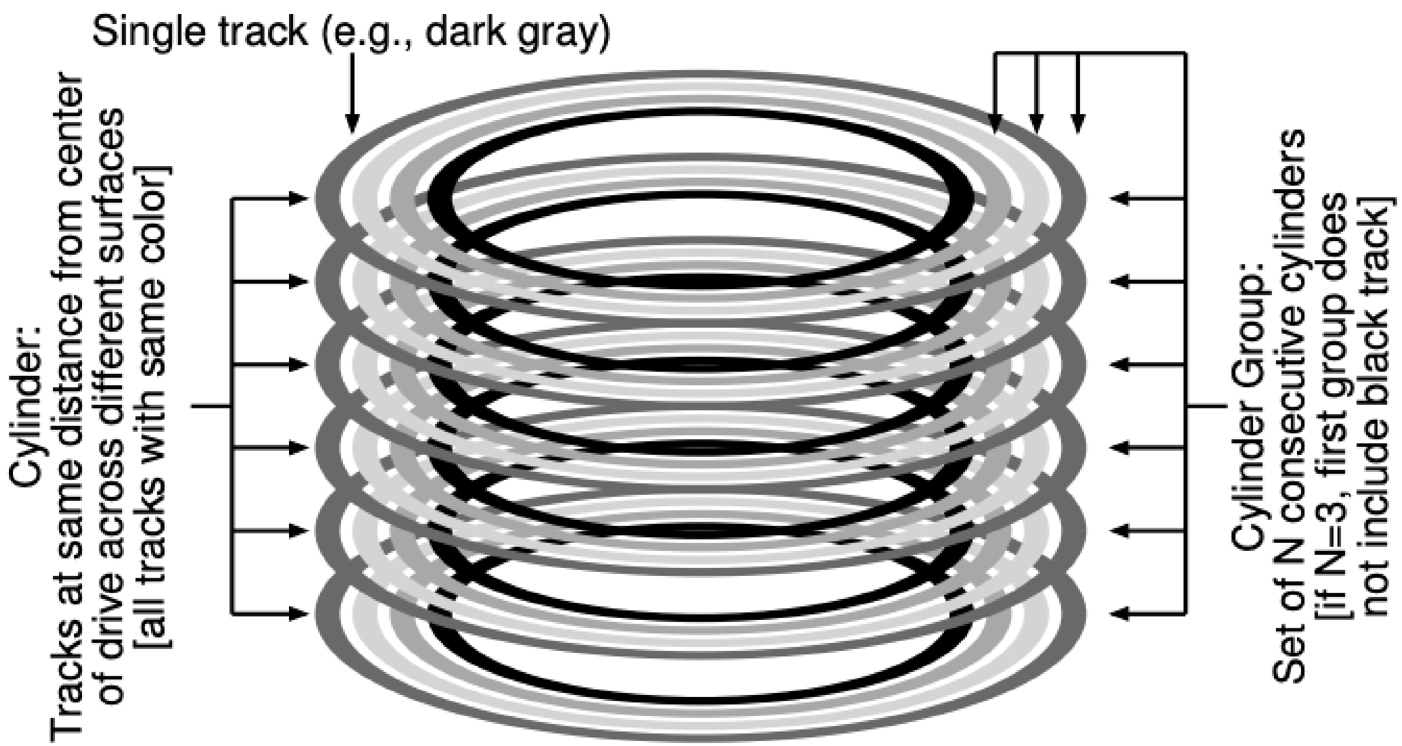


3. Blocks of E are spread out over the disk



Cylinder Groups

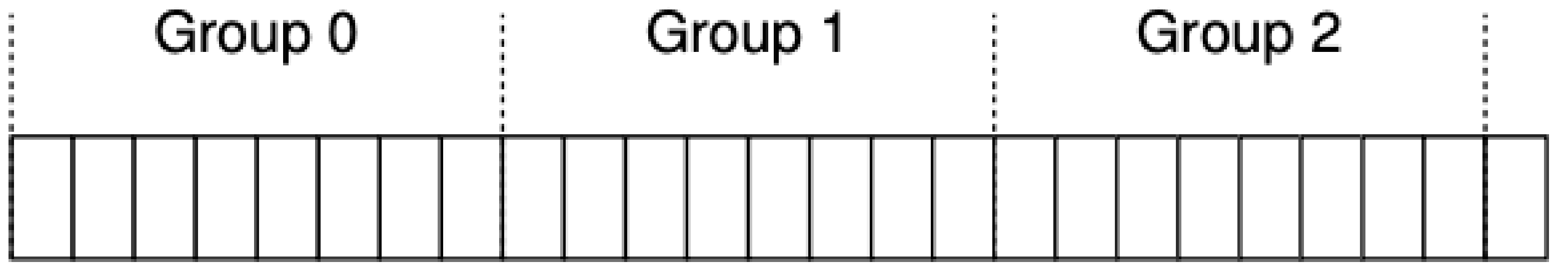
A **cylinder** is a set of tracks near to each other on the drive



Block Groups

Because most hard drives don't provide enough information to choose cylinder groups, most file systems are organized by block groups

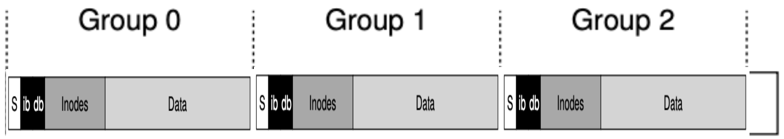
Block groups are consecutive portions of the disk's address space



The Berkeley Fast File System (FFS)

Principle: *keep related stuff together*

A single block group (file system has many)



Each block group has a copy of superblock (S), inode blocks and data blocks

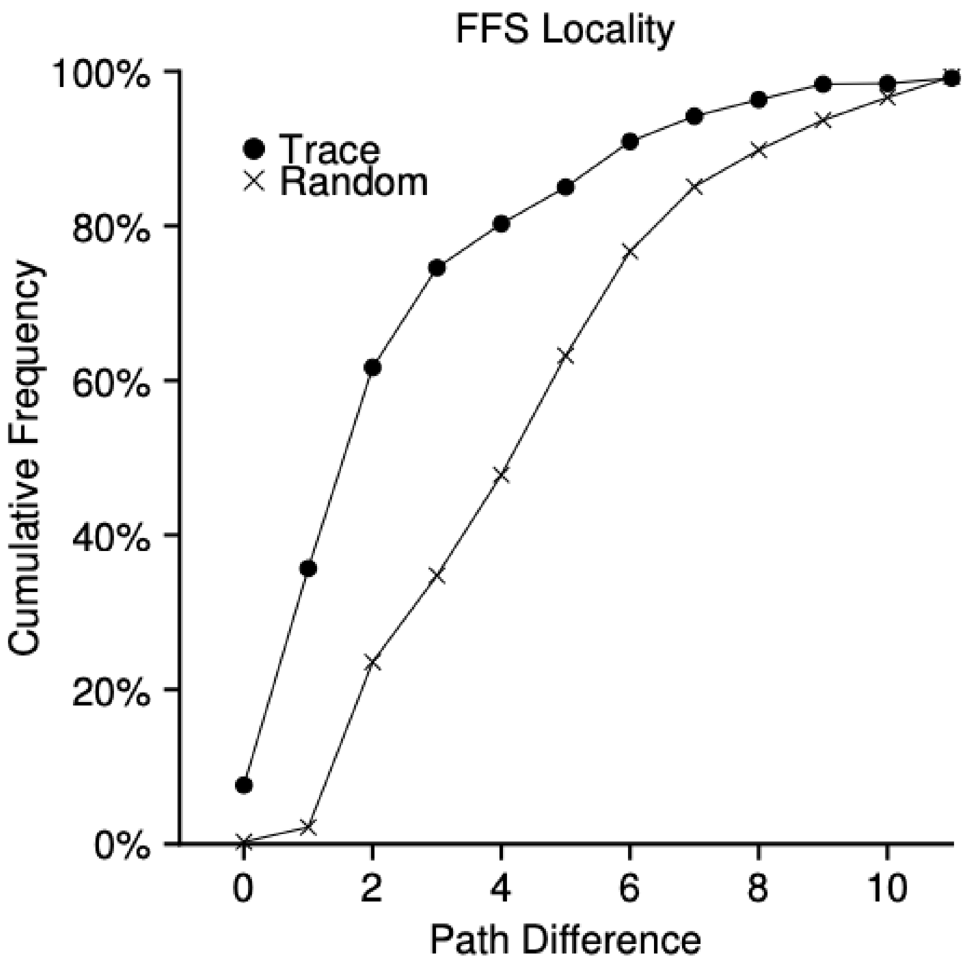
Two heuristics to improve performance:

- Try to allocate data blocks for a file in the same block group as the files inode
- Try to locate files that are together in a directory in the same block group

Path Locality

FFS heuristics are based on another form of locality

Path Locality – consecutive file accesses are likely to be to file paths that are near to each other



Large File Exception

Large file will completely fill block group, preventing files in same directory being in same group

Heuristic

- After blocks are allocated into the first block group (e.g., the 12 direct pointers), FFS places the next "large chunk" (e.g., those pointed to by the first indirect block) in another block group

Sub-Block

Block Size is 4KB, but most of small files have size < 2K. Internal Fragmentation!

For small files, allocate smaller blocks (512KB) called sub-blocks.

Writing small blocks is inefficient (more frequent positioning); for better efficiency, buffer writing till large block(s) of data to write.