File System Implementation

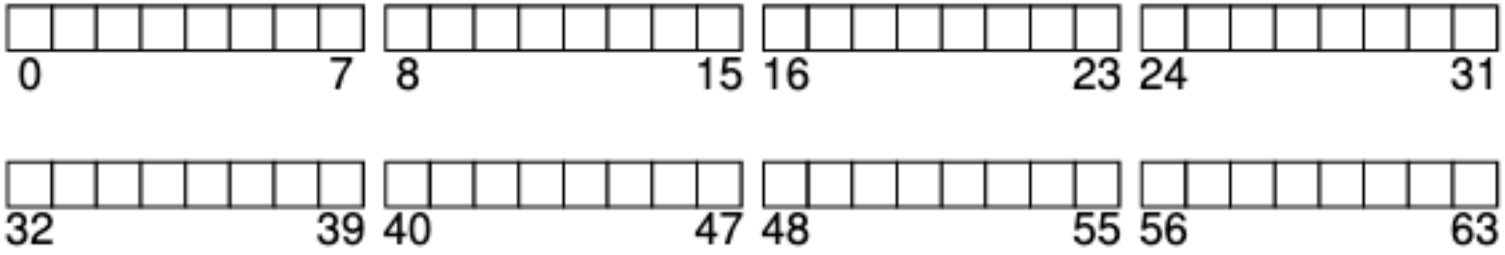
Chapter 40

Previously in CS212…

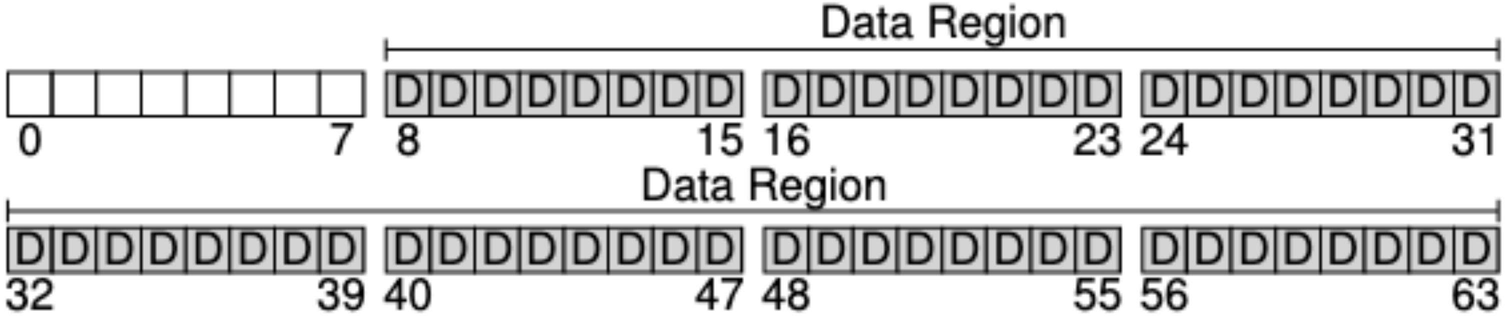
- We looked at the higher-level abstraction for our persistent storage
- Files and Directories in the file system hierarchy
- File Table
  - File descriptors
- Operations
  - open
  - read
  - write
  - close
  - lseek
- File system metadata

Basic Organization

- Chop up the storage medium into storage units of **blocks** (commonly 4 KB)



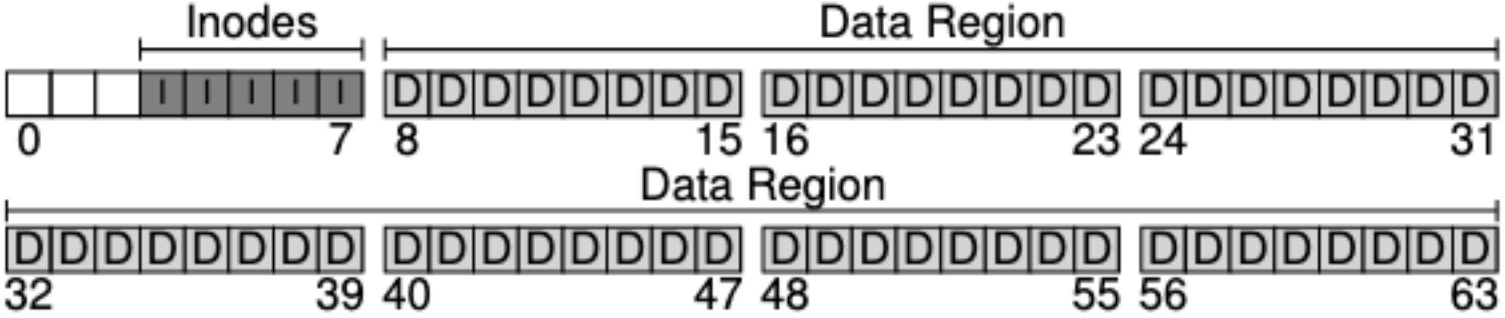
- However, we don't get to use ALL the space for storage



- Why not?

Metadata

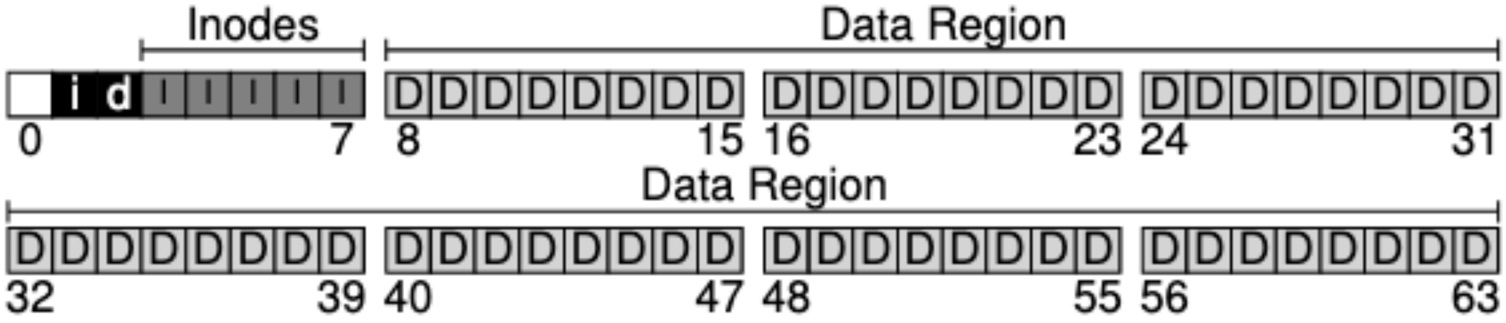
- We need to store information about files/directories stored in the data region
  - Size, owner and access permission, timestamps for access and modification, etc.
- The structure that holds this data is called an **inode**



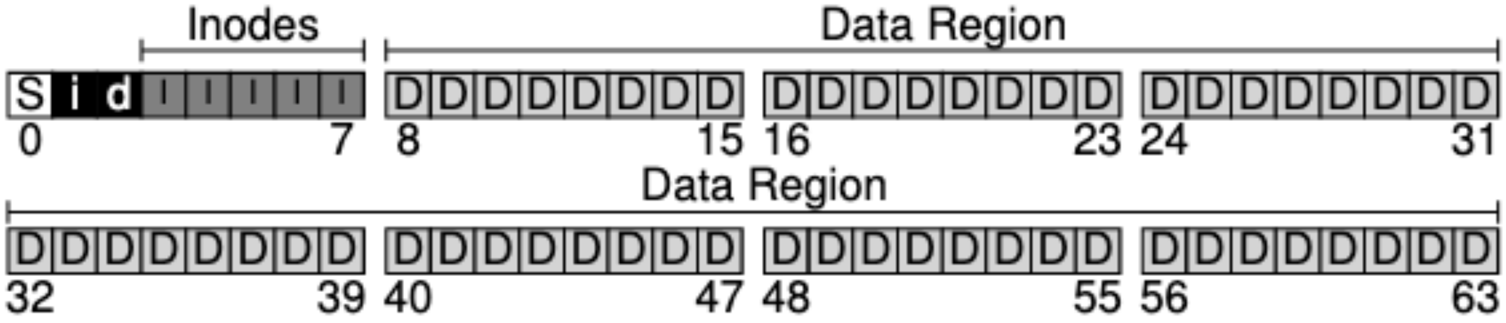
- Inodes tend to be small (128 or 256 bytes)
  - A 4-KB block could hold 16 inodes of size 256 bytes
  - The example above can hold 80 inodes (16 inodes per block * 5 blocks) which is also how many files we could store

Allocation Structures

- We need to also be able to keep track of which data blocks and inodes are in use



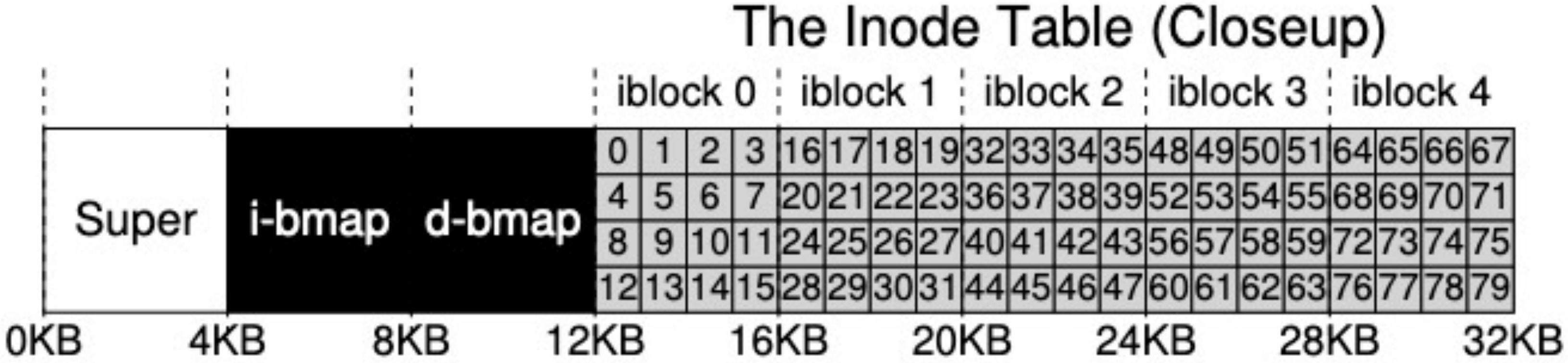
- The vsfs example uses two bitmaps (i and d) for inodes and data blocks
  - 1 is in-use, 0 is free



- The last thing we need is a **SUPERBLOCK** which can tell us details about the filesystem

Inodes

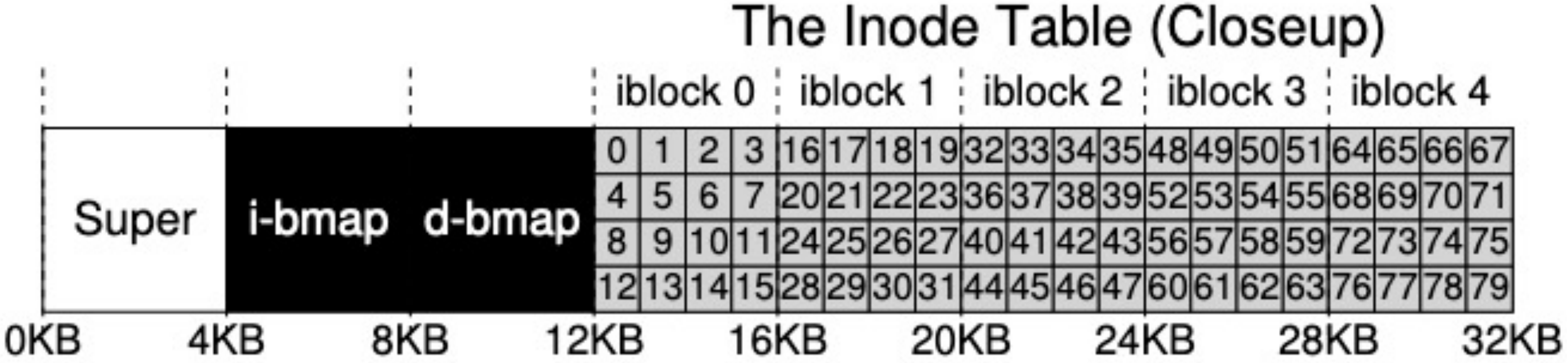
- Index nodes (inodes) are referred to by the i-number or low-level name of the file
- These allow you to locate where the data is located on disk
- Assume:
    - inode number = 52
    - inode size = 256 bytes
    - sector size = 512 bytes
- What is the byte address of the block of inodes?
    - 52 * 256 + 12KB = 25KB
- Oops, the HDD is not byte addressable…


The Inode Table (Closeup)

0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79

Inode Sector Iblock and Address Calculation

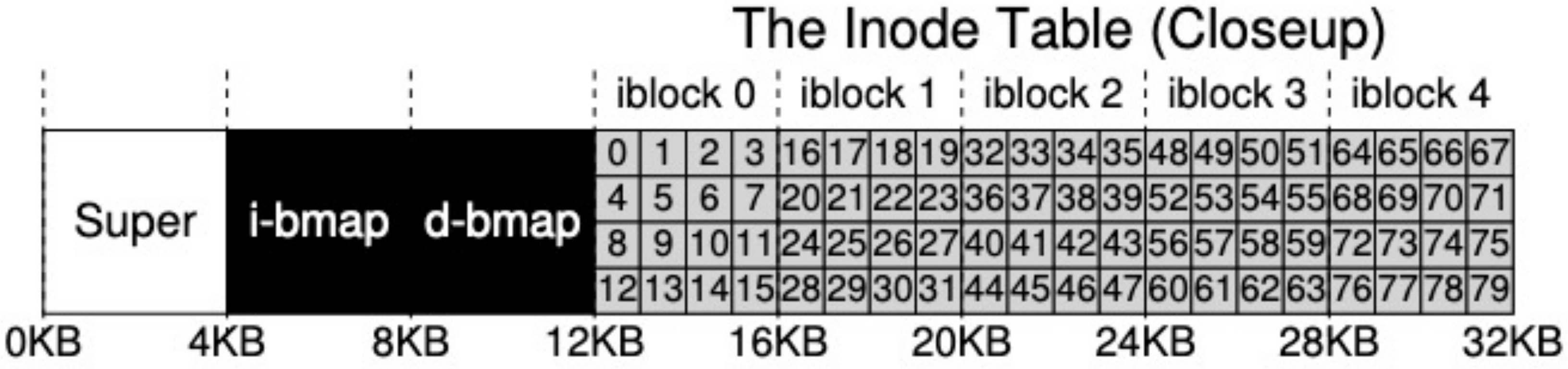
- Recall:
  - block size = 4096 bytes (4KB)
  - inode size = 256 bytes
    - 16 inodes per block
  - sector size = 512 bytes
    - 8 sectors per block


The Inode Table (Closeup)

- iblock = (i-number * inode size) / block size
- sector address = ((iblock * block size) + inode table start address) / sector size
- What’s the sector address of i-numbers 0, 32, 33, and 53?
  - 24, 40, 40, 50

Inode Sector Address Calculation Shortcut

- Recall:
  - block size = 4096 bytes (4KB)
  - inode size = 256 bytes
    - 16 inodes per block
  - sector size = 512 bytes
    - 8 sectors per block



- sector address = ((i-number * inode size) + inode table start) / sector size
  - Still works, I promise

Referencing Data Blocks via Pointers

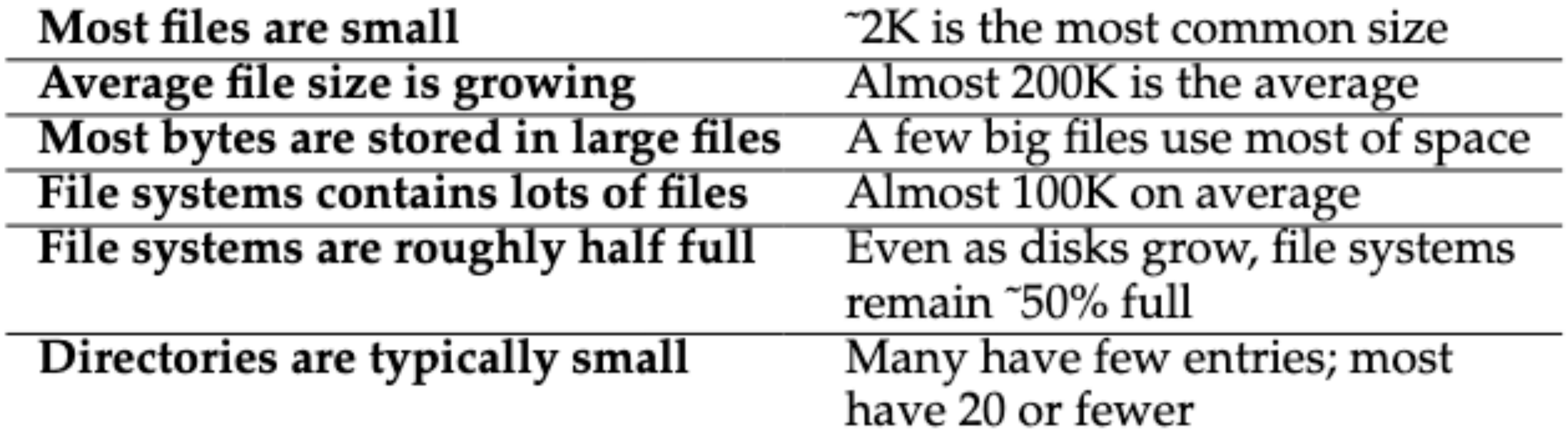
- Each inode could have a set of **direct pointers** that stores the disk block addresses for the file
- What happens for large files?
  - Any file larger than block size * num of direct pointers is too big!
- We can work around this by having an **indirect pointer** that points to a block on disk that contains even more pointers to disk blocks
- We can combine the two solutions to have a set of direct pointers and indirect pointers
  - With 12 direct pointers, 1 indirect pointer, 4-byte addresses, and 4 KB pages we can store files as large as (12 + 1024) * 4 KB or 4,144 KB (4 MB)

Multi-level Indexing

- We can continue the process of using indirect pointers for double or even triple indirect pointers
- In a double indirect pointer, we reference a block that contains pointers to indirect blocks
  - Those indirect blocks in turn contain the actual block addressed on disk
- With a double indirect pointer, we can achieve 1024^2 * 4KB or ~4GB files

Why have a set of direct pointers at all?

- Performing the extra steps of indirection to associate all the necessary block of data for a file isn’t exactly efficient
- We are optimizing for the “typical” case

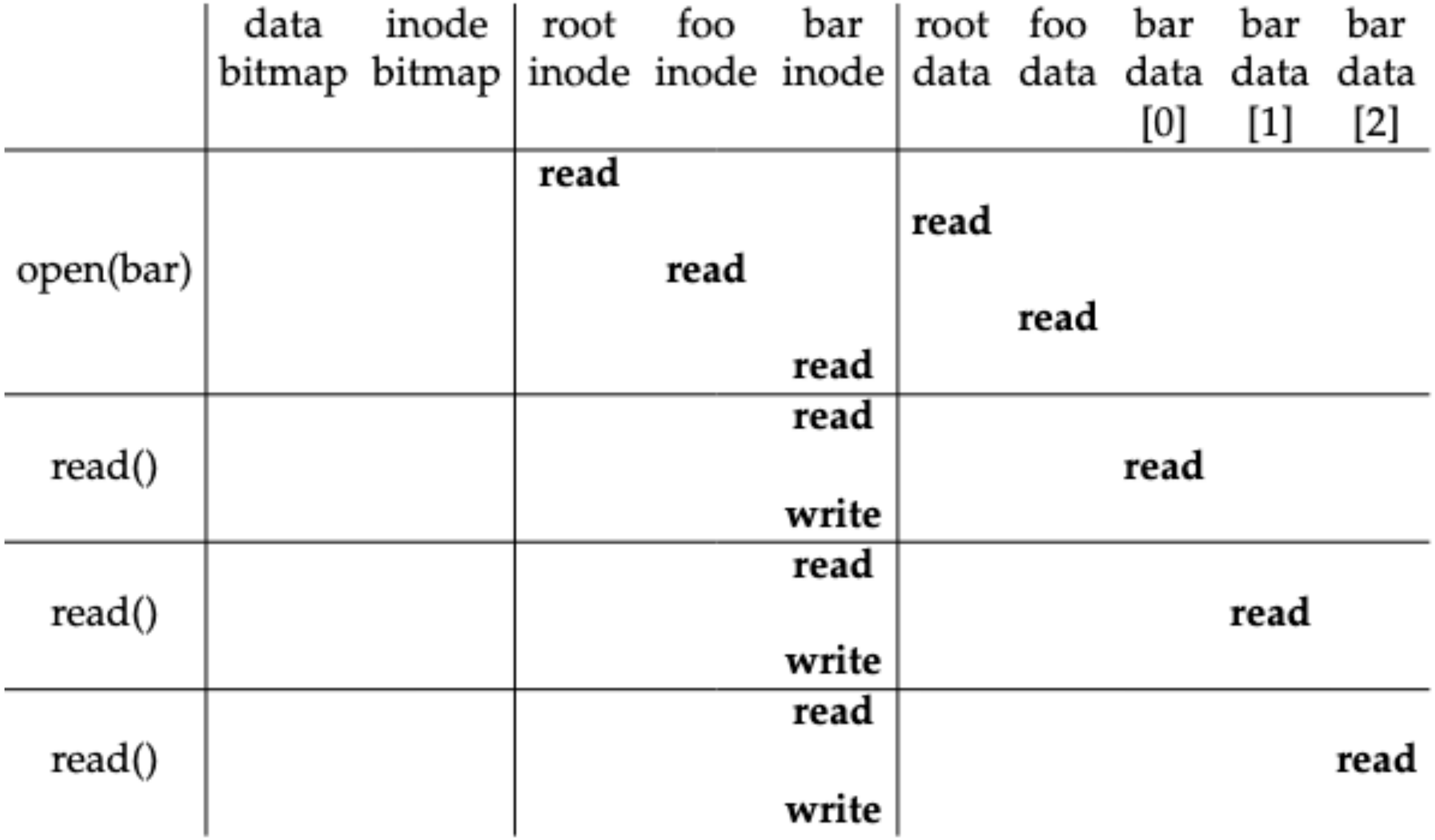


Most files are small	~2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

- If we can reference all the blocks we need with a small set of direct pointers, this is more efficient

Access Path for Reading

- Reading File @ /foo/bar
  - /
  - foo
  - bar (the file to read)
- What's with the writing?
  - Last accessed metadata update



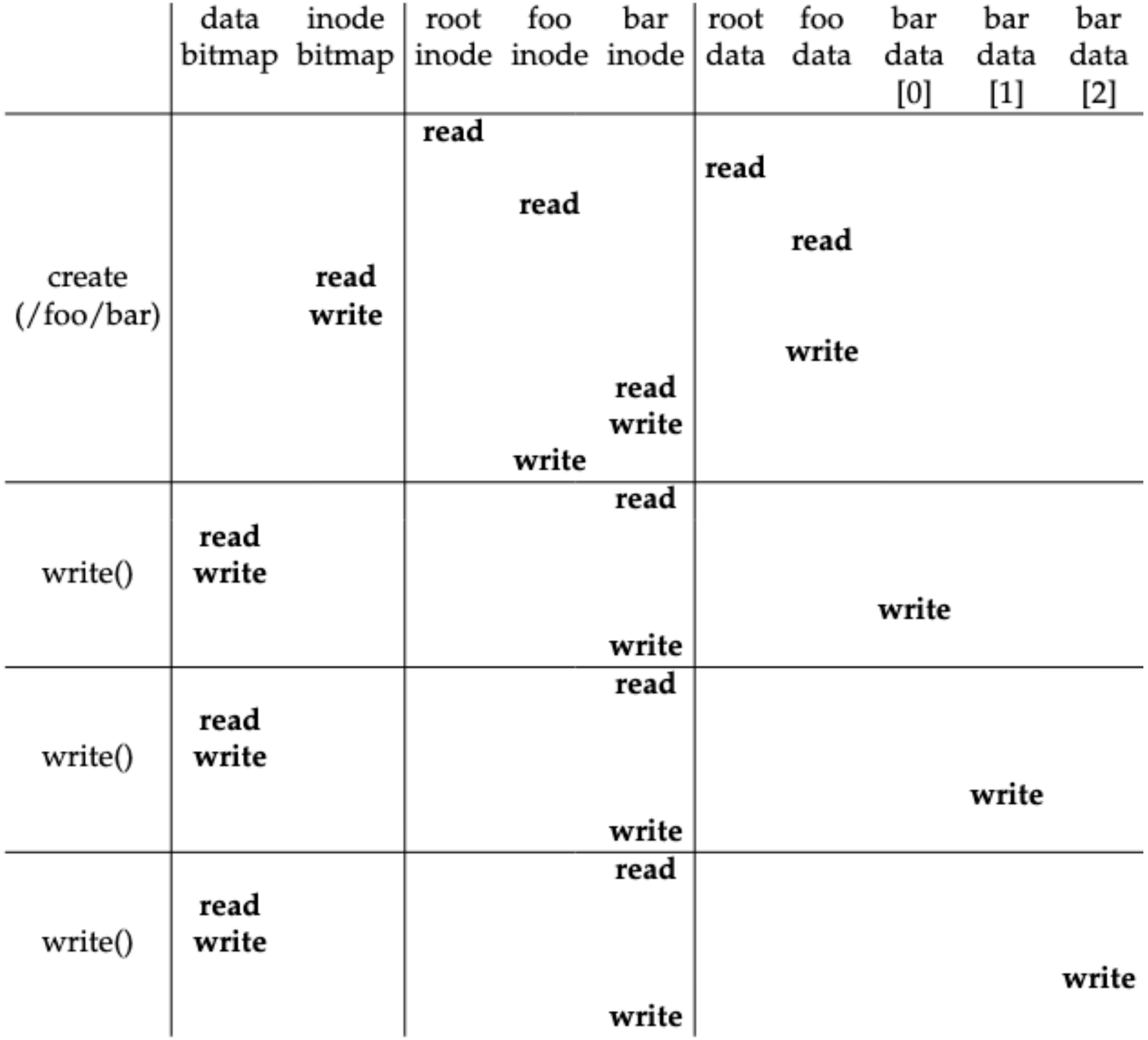
	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**							
						read				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
		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

Access Path for Writing

- Writing new file @ /foo/bar
  - /
  - foo
  - bar (the file to created)

- Need to update bitmaps

- Why the write to foo inode?
  - Directory’s hold data too!
  - As more files are added the directory information grows and takes up more space
  - The inode references the space the directory uses



	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					

Reducing File System Read I/O Costs

- Aggressive Caching with RAM!
    - static partitioning
        - Fixed-sized cache – Fair, easier to implement, but perhaps wasteful
    - dynamic partitioning
        - Unified page cache – Better utilization, flexible, perhaps unfair, difficult to implement
- Use something like the LRU (or other) strategies to save important data in memory
- While initial reads might incur a cost, subsequent reads may be able to be read from RAM cache which is MUCH faster

Reducing File System Write I/O Costs

- Caching has less of an impact here as the writing must still be done
- Here we can use write buffering to delay writes
  - Hold the data to be written in RAM and write it out later
- Why?
  - We can batch jobs together that may need to update similar structures (bitmaps, directories, etc.)
  - Can allow for better I/O scheduling
  - Some operations can be avoided completely
    - Create a file, and then delete it soon after
- Writes can be buffered between 5 and 30 seconds on most systems

Wait…RAM isn’t persistent

- Yup…buffering can mitigate file system I/O performance impacts, but if the power goes abruptly…so too goes your data
- For general purpose computing, probably fine
- A significant problem for critical systems like databases
  - May force writes to disk with fsync, direct I/O, or raw disk interface
- Durability / Performance Trade-off

Next Time

- We investigate ways to improve file system performance