File System Implementation

Two aspects

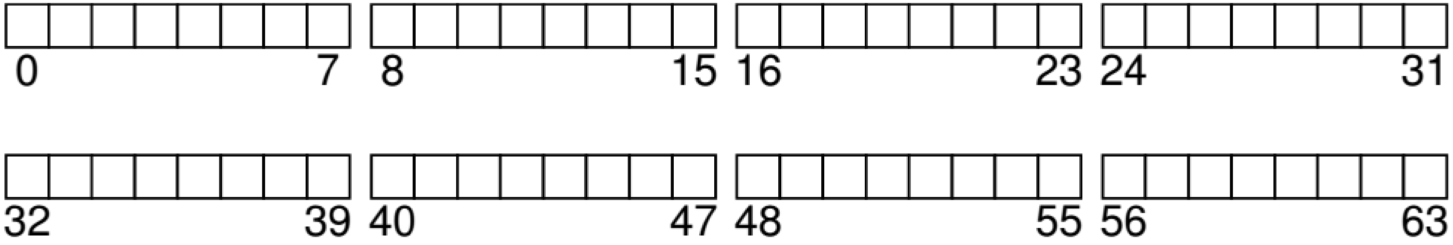
- Simplified version of a typical UNIX file system
  - Introduce some of the basic on-disk structures, access methods, and various policies that you will find in many file systems today.
  - Other file systems
    - AFS (the Andrew File System)
    - ZFS (Sun’s Zettabyte File System)
- Two aspects
  - **data structures** of the file system.
    - What types of on-disk structures are utilized by the file system to organize its data and metadata?
  - **access methods**.
    - How does it map the calls made by a process, such as open(), read(), write(), etc., onto its structures?
    - Which structures are read during the execution of a particular system call? Which are written?
    - How efficiently are all of these steps performed?

Outline

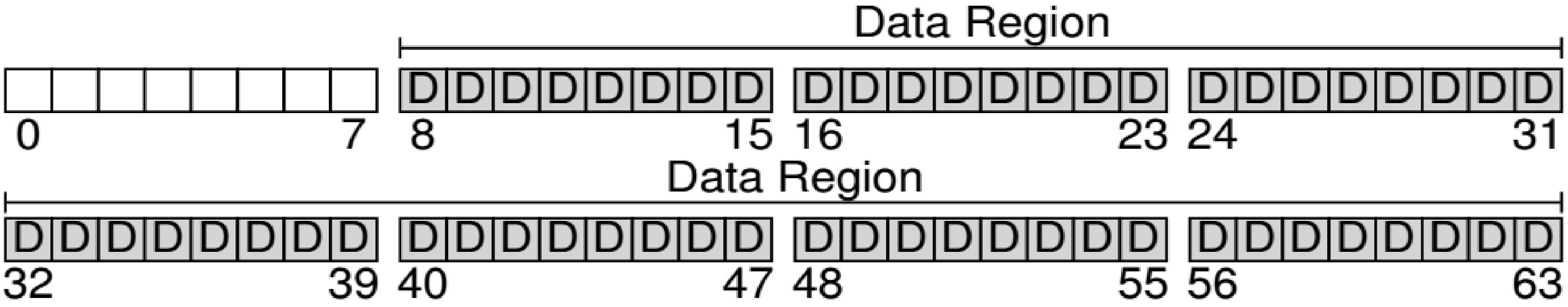
- Explain about simple file system
- i-node
- Directory organization
- Access paths

Overall Organization

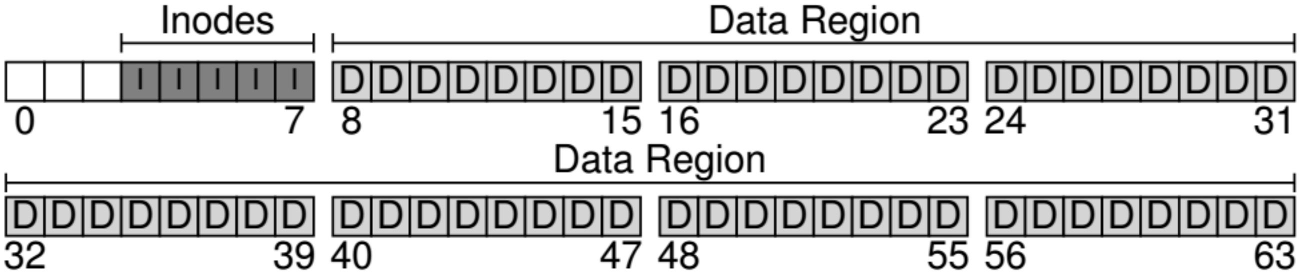
- Disk is divided into blocks
  - For example, the block size is 4k.
- Assume a small disk has 64 small blocks.



- We will allocate 56 blocks for data

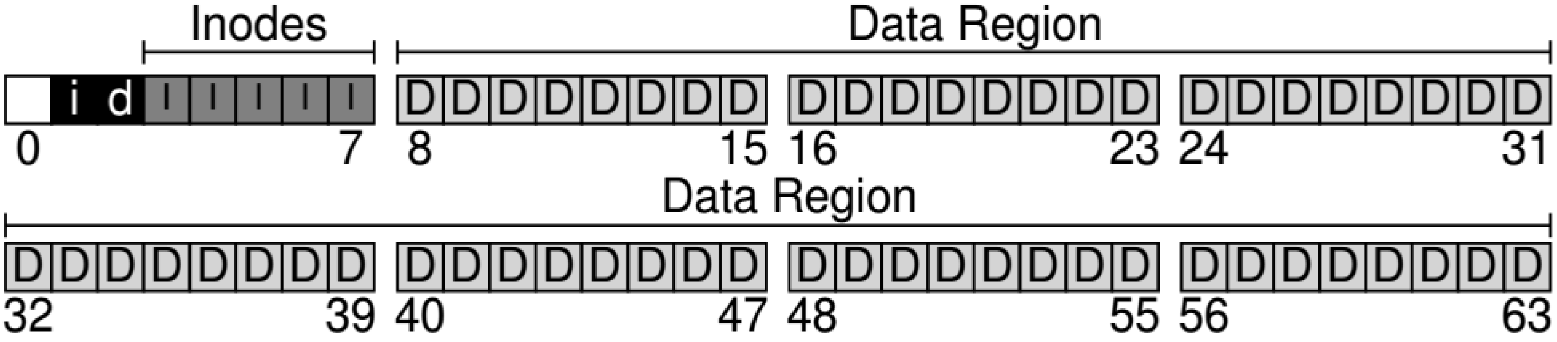

Data Region
D D D D D D D D D D D D D D D D D D D D D D D D
0
7
8
15
16
23
24
31
Data Region
D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
32
39
40
47
48
55
56
63

- File system has to track information about files
- To store this information i-nodes are used
- Five blocks are assigned to i-nodes.
- The inodes are typically not that big, for example 128 or 256 bytes.
- Assuming 256 bytes per inode, a 4-KB block can hold 16 inodes, and our file system above contains 80 total inodes.


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

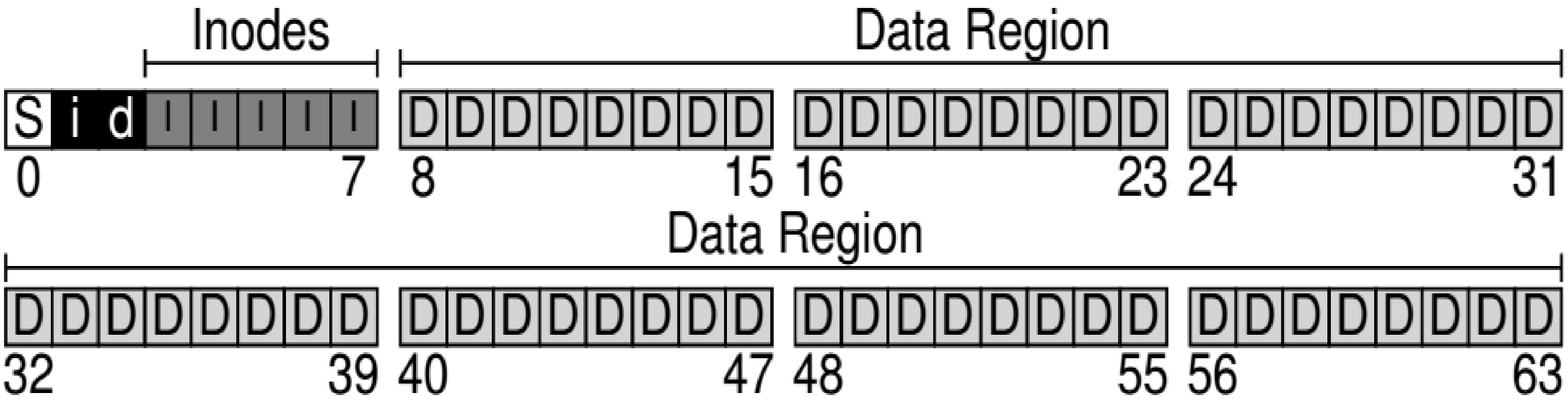
Tracking of Files: bit maps

- We have to track whether inodes or data blocks are free or allocated.
  - – Bitmaps are used
    - One for data region and one for inode region



Super Block

- The superblock contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system (80 and 56, respectively in this instance), where the inode table begins (block 3), and so forth. It will likely also include a magic number of some kind to identify the file system type (in this case, vsfs).

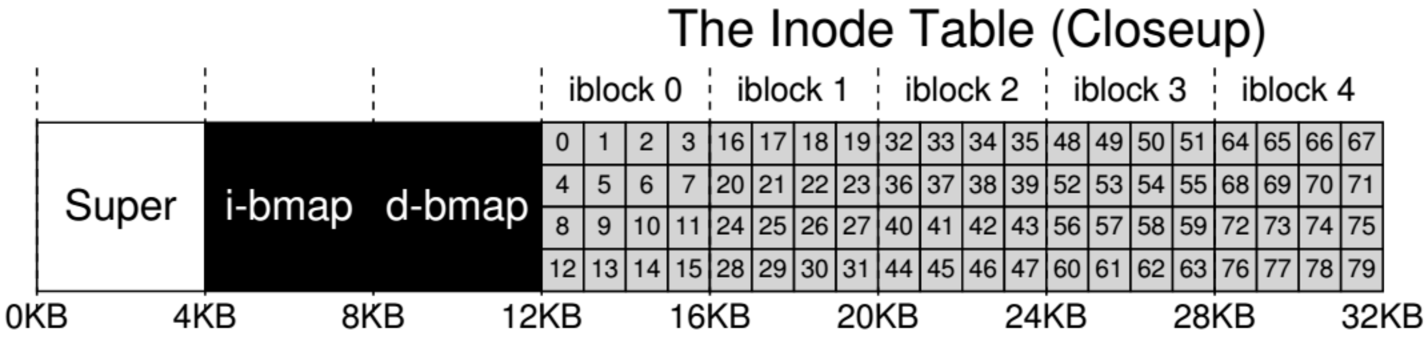


Mounting of a File System

- Reading of super block and initialize the parameters and attaching the volume to the file system tree

File Organization

- i-node is the short form of index node
- Each inode is implicitly referred to by a number (called the **inumber**), which we’ve earlier called the **low-level name** of the file.
- In vsfs (and other simple file systems), given an i-number, you should directly be able to calculate where on the disk the corresponding inode is located.



- To read inode number 32, the file system would first calculate the offset into the inode region (32 · sizeof(inode) or 8192), add it to the start address of the inode table on disk (inodeStartAddr = 12KB), and thus arrive upon the correct byte address of the desired block of inodes: 20KB.

- Recall that disks are not byte addressable, but rather consist of a large number of addressable sectors, usually 512 bytes. Thus, to fetch the block of inodes that contains inode 32, the file system would issue a read to sector 20×1024/ 512 , or 40, to fetch the desired inode block. More generally, the sector address iaddr of the inode block can be calculated as follows:

```
blk = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

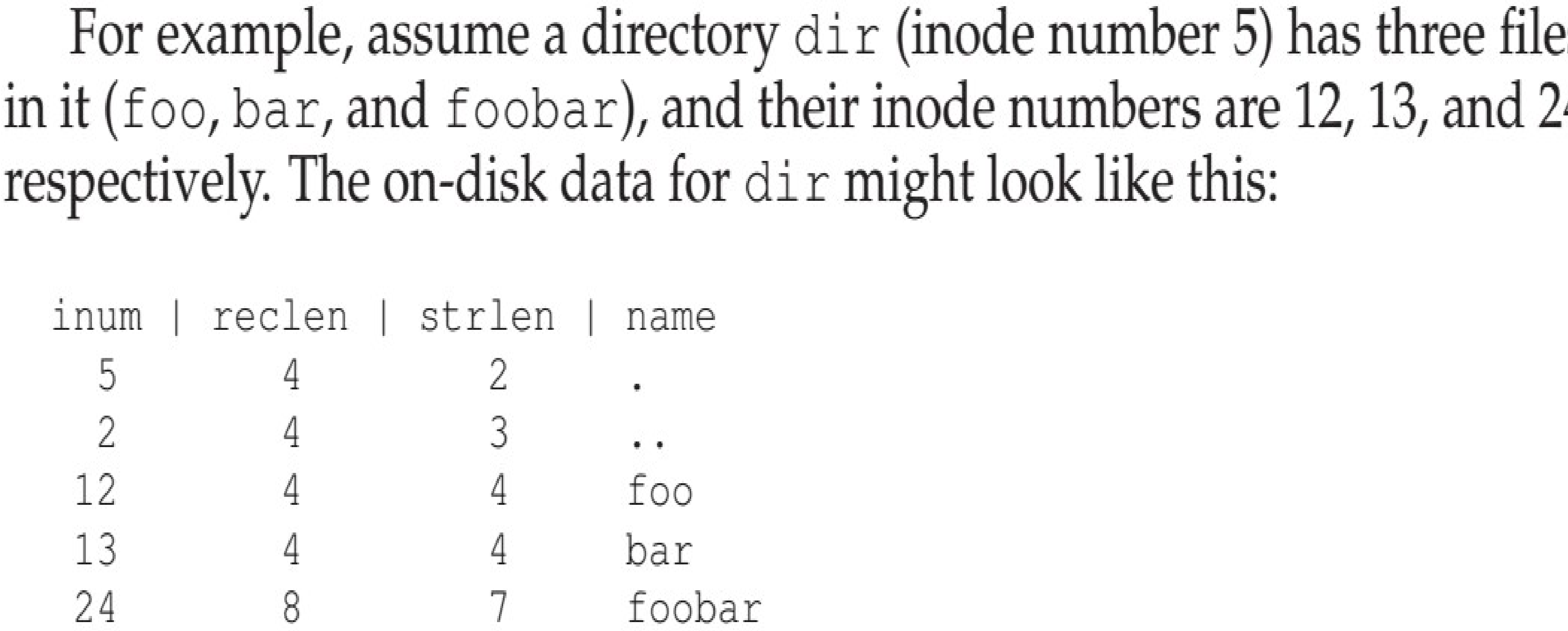
Figure 40.1: **Simplified Ext2 Inode**

Supporting Big files

- Multi-level index
- Instead of pointing to data block, i-node points to block of pointers.
- Notion of extents are used
  - An extent is simply a disk pointer plus a length (in blocks); thus, instead of requiring a pointer for every block of a file, all one needs is a pointer and a length
    to specify the on-disk location of a file.
- Link based approaches

Directory Organization

- A directory has list of pairs (entry name and i-node number)
- File systems treat directories as special type of file.



For example, assume a directory `dir` (inode number 5) has three file in it (`foo`, `bar`, and `foobar`), and their inode numbers are 12, 13, and 2 respectively. The on-disk data for `dir` might look like this:

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

Access Paths: Reading and Writing

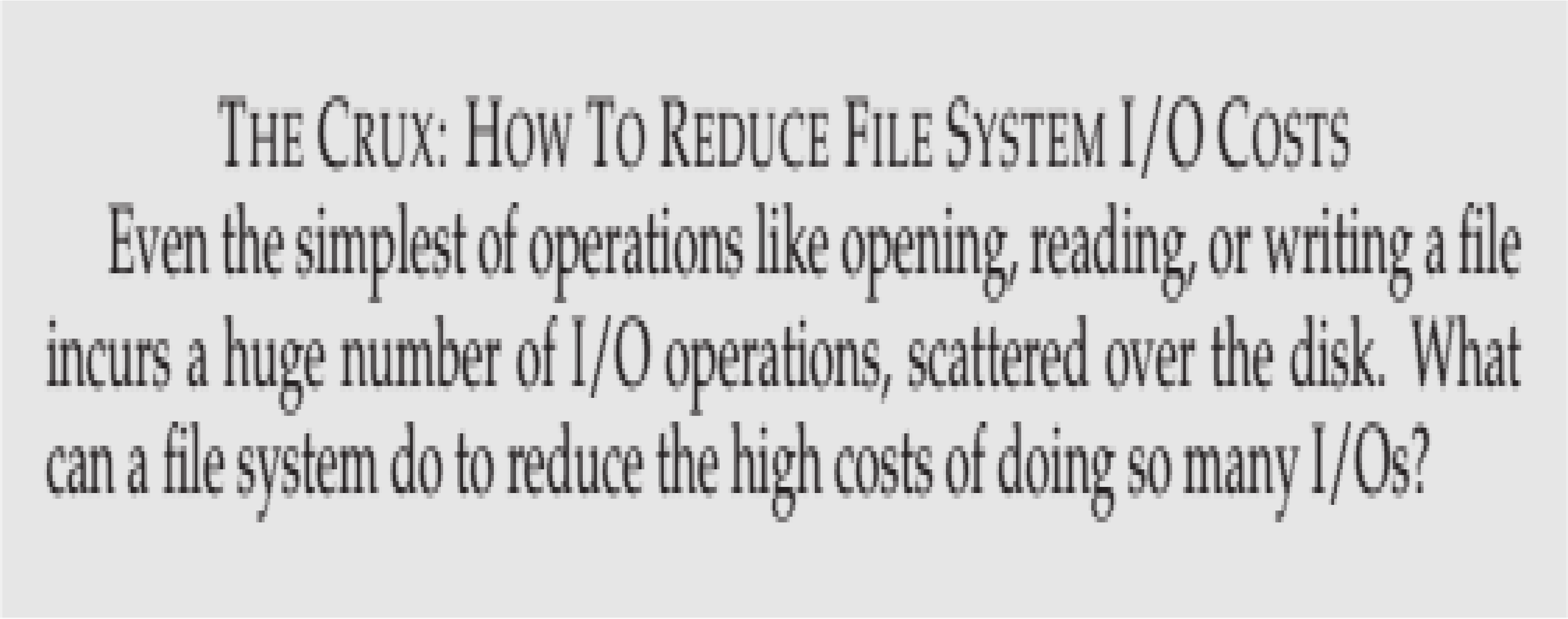
- Reading and writing require several I/Os

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

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

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

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**



The Crux: How To Reduce File System I/O Costs

Even the simplest of operations like opening, reading, or writing a file incurs a huge number of I/O operations, scattered over the disk. What can a file system do to reduce the high costs of doing so many I/Os?

Techniques: Caching and buffering

- Use caching or DRAM to cache important blocks