File System Implementation

04/26/2021

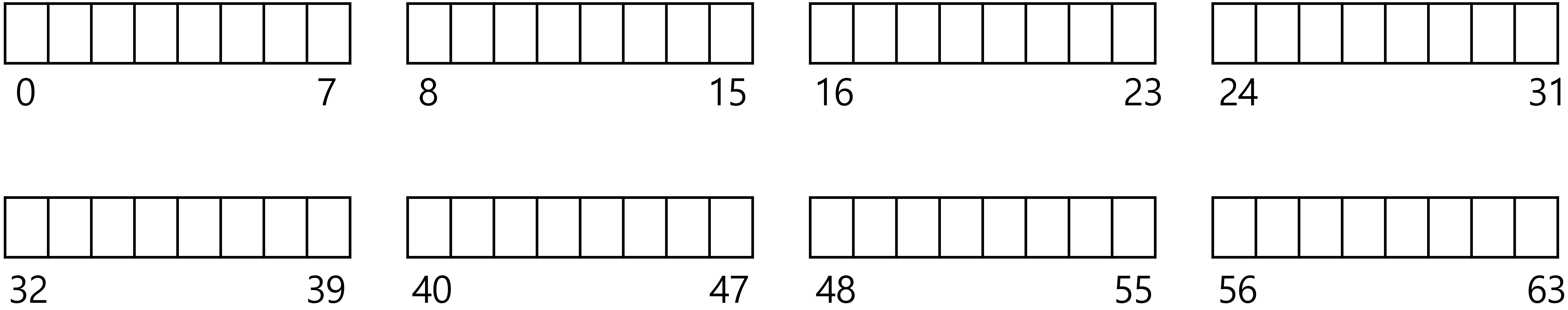
Professor Amanda Bienz

The Way to Think

- Two different aspects to implement file systems
  - Data structures:
    - 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 as open(), read(), write(), etc
    - Which structures are read during the execution of a particular system call?

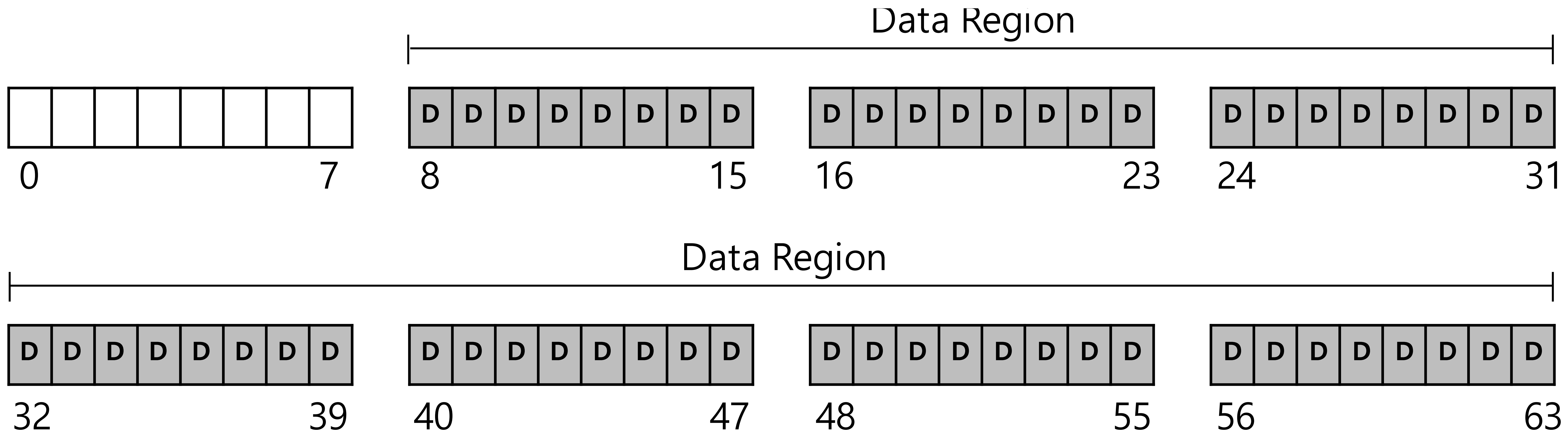
Overall Organization

- Let's develop the overall organization of the file system data structure
- Divide the disk into **blocks**
  - Block size is 4KB, blocks addressed from 0 to N-1



Data region in file system

- Reserve **data region** to store user

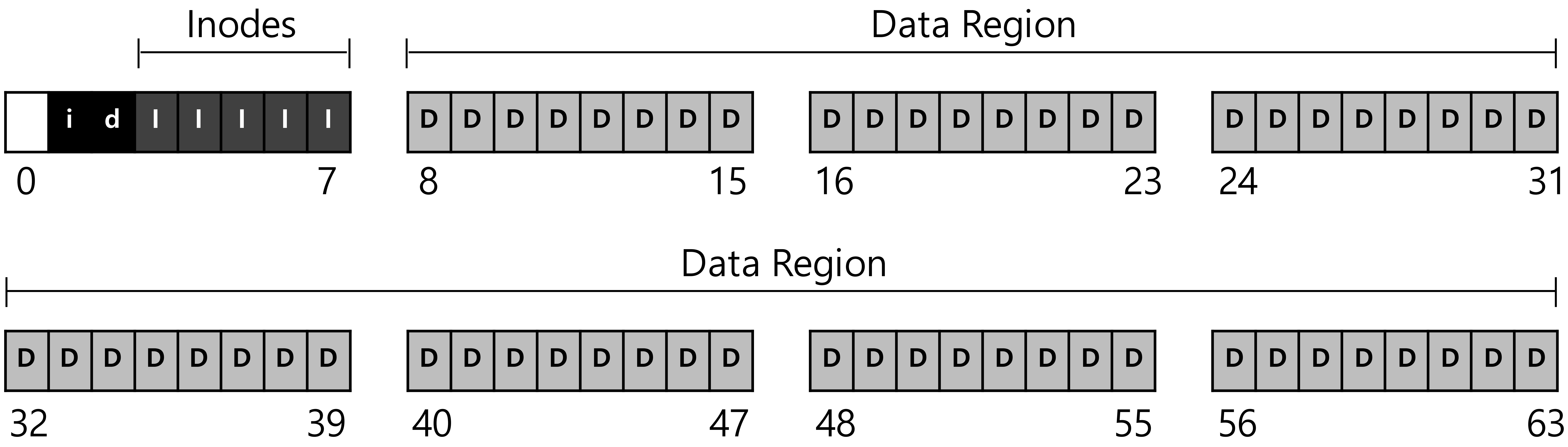


- File system has to track which data block comprise a file, the size of the file, its owner, etc.

How we store these inodes in file system?

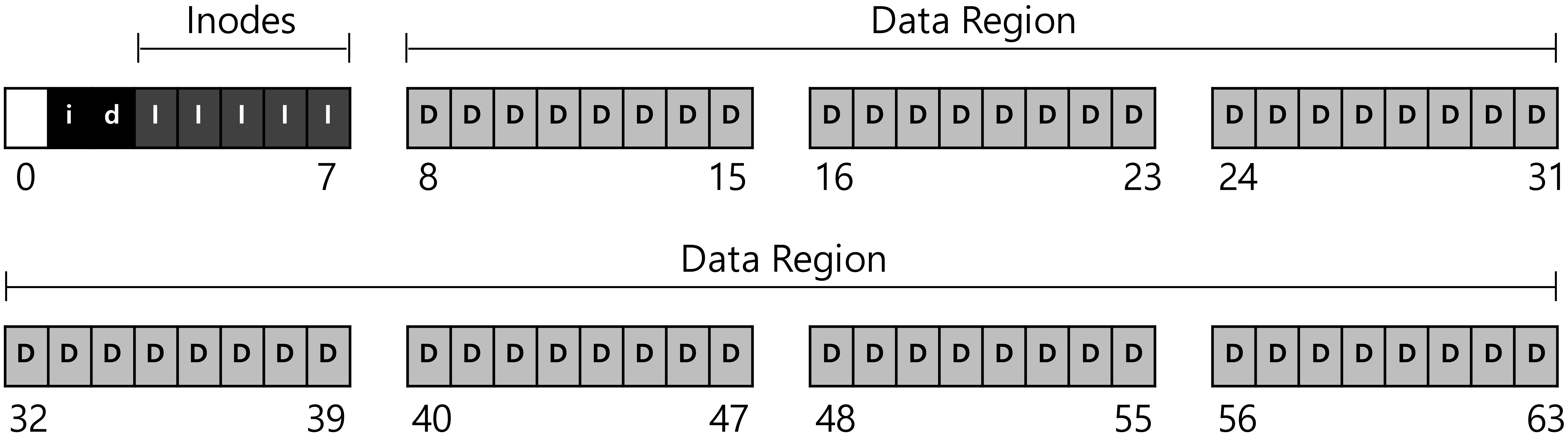
Inode table in file system

- Reserve some space for **inode table**
  - This holds an array of on-disk inodes
  - Ex) inode tables: 3 ~ 7, inode size: 256 bytes
    - 4KB block can hold 16 inodes
    - The filesystem contains 80 inodes (max number of files)



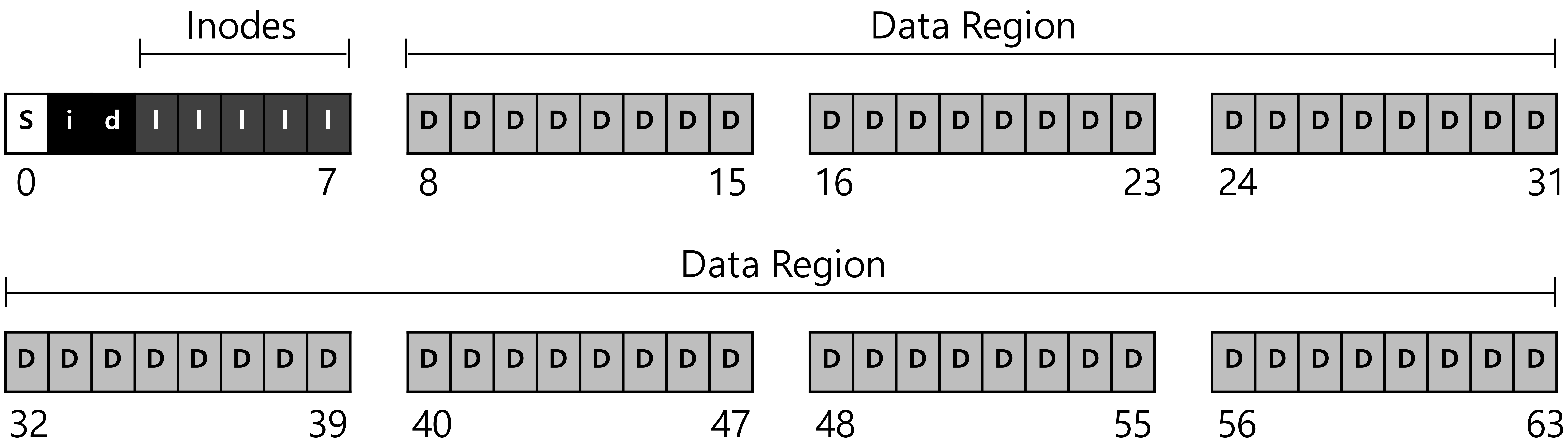
Allocation Structures

- This is to track whether inodes or data blocks are free or allocated
- Use **bitmap**, each bit indicates free(0) or in-use(1)
  - **Data bitmap:** for data region
  - **Inode bitmap:** for inode table



Superblock

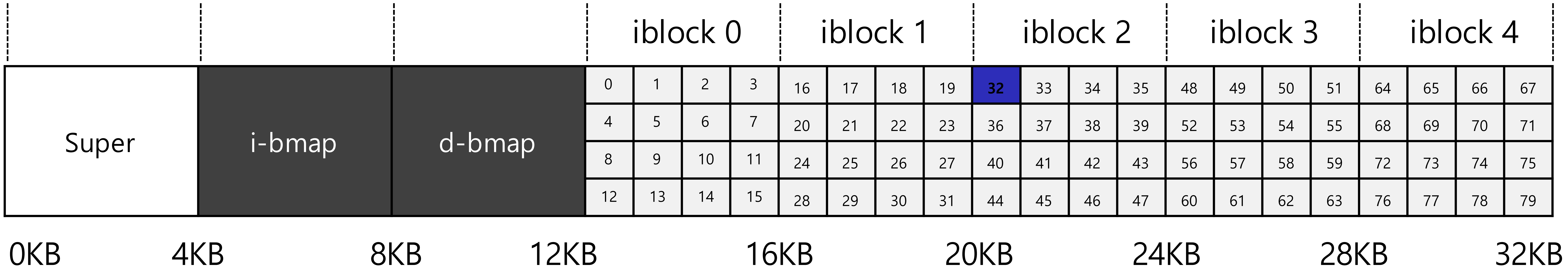
- Super block contains this **information** for **particular file system**
  - Ex) The number of inodes, begin location of node table



- Thus, when mounting a filesystem, OS will read the superblock first, to initialize various information

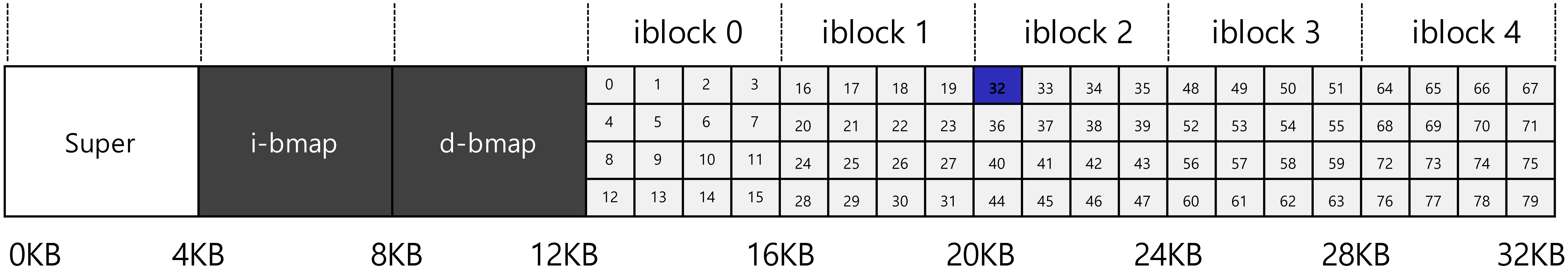
File Organization: The node

- Each inode is referred to by node number
  - By node number, filesystem calculates where the node is on the disk
  - Ex) inode number : 32
    - Calculate the offset into the node region (32*sizeof(inode))
      (32*256) = 8192
    - Add start address of node table (12KB) + node region (8KB) = 20KB



File Organization : The inode (Cont.)

- Disks are not byte addressable, sector addressable
- Disks consist of a large number of addressable sectors (512 bytes)
  - Ex) Fetch the block of inode (inode number: 32)
    - Sector address **iaddr** of the inode block:
      - **blk : (inumber * sizeof(inode)) / blocksize**
      - **Sector : ((blk * block size) + inodeStartAddr) / sector size**



File Organization : The inode (Cont.)

- inode have all of the information about a file
  - File type (regular file, directory, etc.)
  - Size, the number of blocks allocated to it
  - Protection information (who owns the file, who can access it, etc)
  - Time information
  - Etc.

File Organization : The inode (Cont.)

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?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	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
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

The EXT2 Inode

The Multi-Level Index

- To support bigger files, we use multi-level index
- **Indirect pointer** points to a block that contains more pointers
  - Inodes have fixed number of direct pointers (12) and a single indirect pointer
  - If a file grows large enough, an indirect block is allocated, inode's slot for an indirect pointer is set to point to it
    - (12 + 1024) * 4K = 4144KB

The Multi-Level Index (Cont.)

- **Direct indirect pointer** points to a block that contains indirect blocks
  - Allow file to grow with an additional 1024x1024 or 1million 4KB blocks
- **Triple indirect pointer** points to a block that contains double indirect blocks
- Multi-level index approach to pointing to file blocks
  - Ex) 12 direct pointers, a single and double indirect block
    - Over 4GB in size (12+1024+1024^2)*4KB
- Many file systems use a multi-level index
  - Linux EXT2, EXT3, NetApp's WAFL, Unix file system
  - Linux EXT4 use **extents** instead of simple pointers

The Multi-Level Index (Cont.)

Most files are small	Roughly 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 the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file system remain -50% full
Directories are typically small	Many have few entries; most have 20 or fewer

File System Measurement Summary

Directory Organization

- Directory contains a list of (entry name, inode number) pairs
- Each directory has two extra files "." for current directory and ".." for parent directory
  - For example, dir has three files (foo, bar, foobar)

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

on-disk for dir

Free Space Management

- File system track which inode and data block are free or not
- In order to manage free space, we have two simple bitmaps
  - When file is newly created, its allocated an inode by searching the inode bitmap and update on-disk bitmap
  - Pre-allocation policy is commonly used for allocating contiguous blocks

Access Paths: Reading a File From Disk

- Issue an open ("/foo/bar", O_RDONLY)
  - Traverse pathname and thus locate desired inode
  - Begin at the root (/)… in most Unix file systems, root inode number is 2
  - Filesystem reads in the block that contains inode number 2
  - Look inside of it to find pointer to data blocks (contents of root)
  - By reading in one or more directory data blocks, it will find "foo"
  - Traverse recursively the path name until the desired inode ("bar")
  - Check final permissions, allocate a file descriptor for this process and return file descriptor to user

Access Paths: Reading a File From Disk (Cont.)

- **Issue read() to read from the file**
  - Read in first block of the file, consulting inode to find the location of such a block
    - Update the inode with a new last accessed time
    - Update in-memory open file table for file descriptor, file offset
- **When file is closed:**
  - File descriptor should be deallocated, but for now, that is all the file system really needs to do. No disk I/Os take place.

Access Paths: Reading a File From Disk (Cont.)

	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					

File Read Timeline (Time Increasing Downward)

Access Paths: Writing to Disk

- Issue **write()** to update the file with new contents
- File may allocate a block (unless the block is being overwritten)
  - Need to update data block, data bitmap
  - It generates five I/Os:
    - One to read the data bitmap
    - One to write the bitmap (to reflect new state to disk)
    - Two more to read and then write the inode
    - One to write the actual block itself
  - To create file, it also allocates space for directory, causing high I/O traffic

Access Paths: Writing to Disk (Cont.)

	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					

File Creation Timeline (Time Increasing Downward)

Caching and Buffering

- Reading and writing files are expensive, incurring many I/Os
  - For example, long pathname (/1/2/3/…/100/file.txt)
    - One to read the inode of the directory and at least one to read its data
    - Literally perform hundreds of reads just to open the file
- In order to reduce I/O traffic, file systems aggressively use system memory (DRAM) to cache
  - Early file system use fixed-size cache to hold popular blocks
    - Static partitioning of memory can be wasteful
  - Modern systems use dynamic partitioning approach, unified page cache
- Read I/O can be avoided by large cache

Caching and Buffering (Cont.)

- Write traffic has to go to disk for persistent. Thus, cache does not reduce write I/Os
- File system use write buffering for write performance benefits
  - Delaying writes (file system batch some updates into a smaller set of I/Os)
  - By buffering a number of writes in memory, the file system can then schedule the subsequent I/Os
  - By avoiding writes
- Some applications force flush data to disk by calling fsync() or direct I/O