File Systems

From blocks to something useful….

Roadmap

- Files and directories
  - CRUD operations
- How to implement them
  - Data structures
  - Access methods

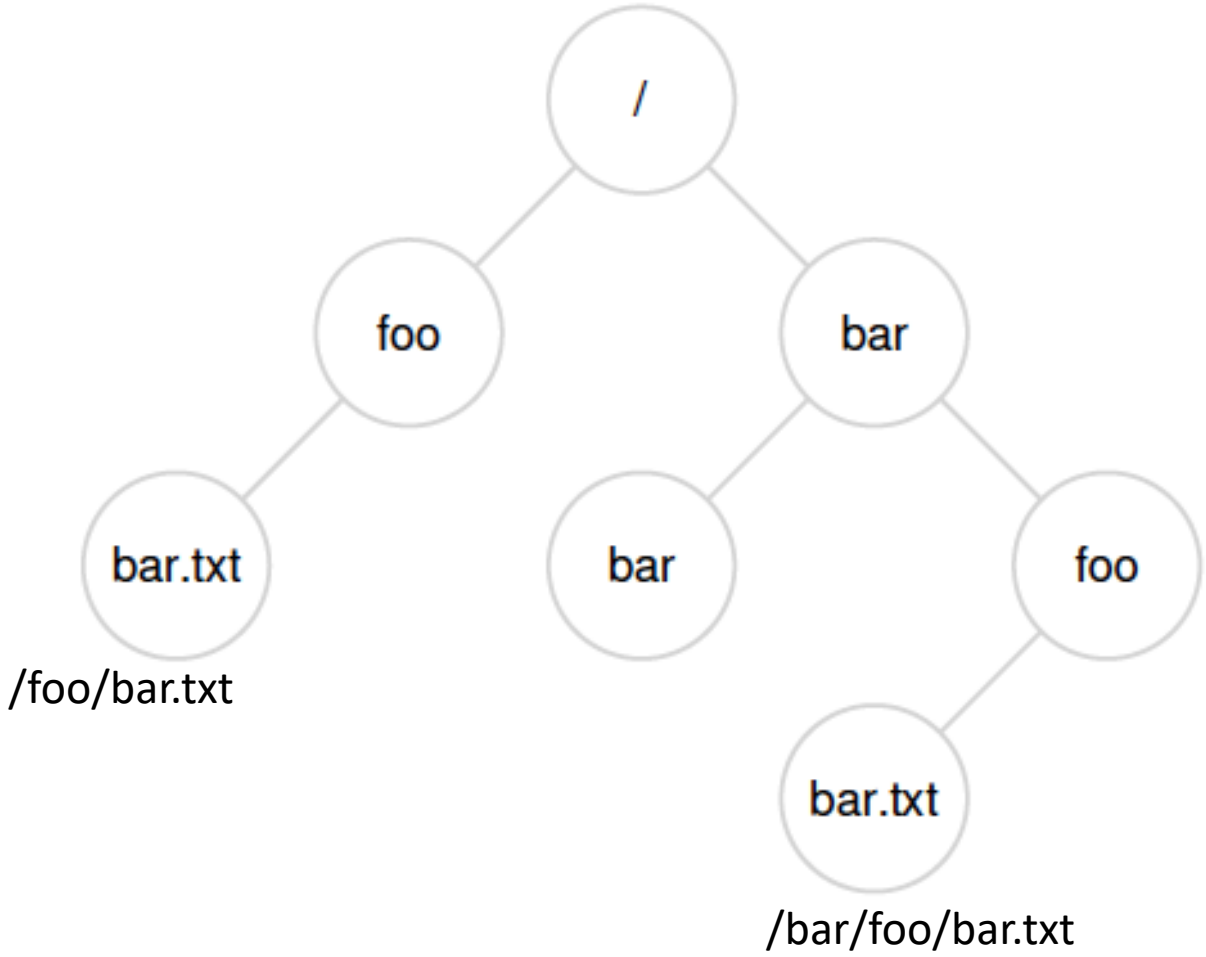
File Systems

- Disks are organized into block (sectors)
- Applications generally like to see files
  - Names, directories, permissions, well defined operations
- How do we organize the blocks on the disk to make files?
- How do we implement operations efficiently on that organization
- How do we manage the files to deal with reliability, backup, performance, security, etc?

Key Abstractions

- Files are a sequence of bytes
  - Can read and write randomly
  - Can append to make bigger
- Files are organized into directories
  - File may live in more then one directory!
    - Hard links, and soft links

A Directory “Tree”



Storage Operations

- When describing a storage system, we need to specify the semantics of basic operations
  - Create
  - Read
  - Update
  - Delete

File Systems

- File Operations
  - Create, delete, rename, open, close,
  - seek, read, write,
  - get/set attributes
  - Need to define what shared operations mean with respect to read/write
- Directory Operations
  - create, delete, link, rename
- Files live in a “file system”
  - File system must be mounted before use
  - Mounting specifies “physical” and logical location of the data
- File operations don’t map directly to disk operations!

Accessing a file

- You must “open” a file before you can read it
  - Allows the operating system to do book-keeping and provides a convenient to use “handle” called a file descriptor or file object
    - Eg. in f = open(“foo”);
- Open can take options, for example to create the file if it doesn’t exist, to open read only, or to zero it out (truncate), or append mode
- Must “close” file when you are done with it

Create

- User interface, e.g., via GUI
- Implementation, e.g., via a C program with a call to system function open()

- fd = open("foo", “w+”);
  - Open with flags indicating how to open the file

open(*file*, *mode='r'*, *buffering=1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open for updating (reading and writing)

Binary vs Text Mode

- Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding (the default being UTF-8). 'b' appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don’t contain text.
- In text mode, the default when reading is to convert platform-specific line endings (\n on Unix, \r\n on Windows) to just \n. When writing in text mode, the default is to convert occurrences of \n back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

File descriptor

- Note open() returns file object
  - Predefined file objects: stdin 0, stdout, 1, stderr 2

Reading and Writing

- File descriptor used for subsequent read and write operations
  - f.read(25),
  - write(fd,”this is some good text”)
  - Read and write operations are “stateful” in that they read/write at the position last read/written to
- Random reads/writes require that to say where in the file you want the operation to take place, using a “seek” operation
  - f.lseek(36)
    f.write(”this is 36 bytes into the file”);

Read

- f.read(size)
  - Read from file “f” <*size*> number of bytes/characters
- f.readline*()*
- Read starts from the current offset of f
  - Initially at 0

Write

- f.write(buffer)
  - Write to file *f* <*size*> the contents stored in buffer
  - Also start from the current offset

Random read and write

- f.seek(offset, whence)
  - If whence is SEEK_SET, the offset is set to offset bytes from the beginning of file
- A *whence* value of
  - 0 measures from the beginning of the file,
  - 1 uses the current file position, and
  - 2 uses the end of the file
- whence: from where

File and directory

- When creating a file
  - Bookkeeping data structure created (inode): recording size of file, location of its blocks, etc.
  - Linking a human-readable name to the file
  - Putting the link in a directory

Files have…

- A name, which is a human readable form
  - File names live in directories
  - The directory is not part of the name!!
- Can have more the one name for the same file
  - Remember operating system knows files by their inode
  - Can “link” more then one name to the same inode number (hard link)
  - Can “link” one name to another name (soft link)

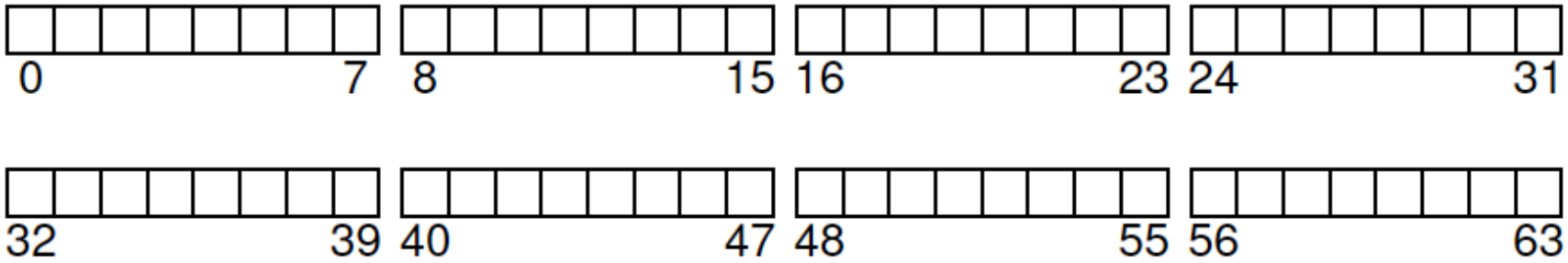
Directories

- A directory is a list of file names or directory names
  - os.getcwd()
  - os.listdir()

File systems

- How to we efficiently create files and associated operations out of a set of disk blocks?

Here is a small, empty disk



Consider a disk with 64 blocks

4KB/block

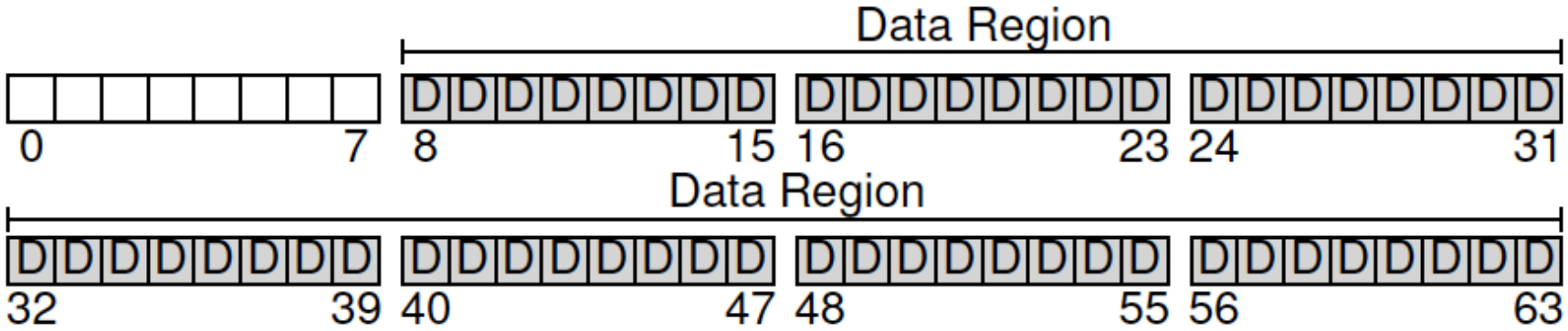
512B/sector

So there are $2^{12}/2^{9} = 2^{3} = 8$ sectors/block

Capacity of disk = 64 * 4KB = 256KB

- Can be multiple sectors to a block
- Storage could be implemented on RAID device

Where to put user data?



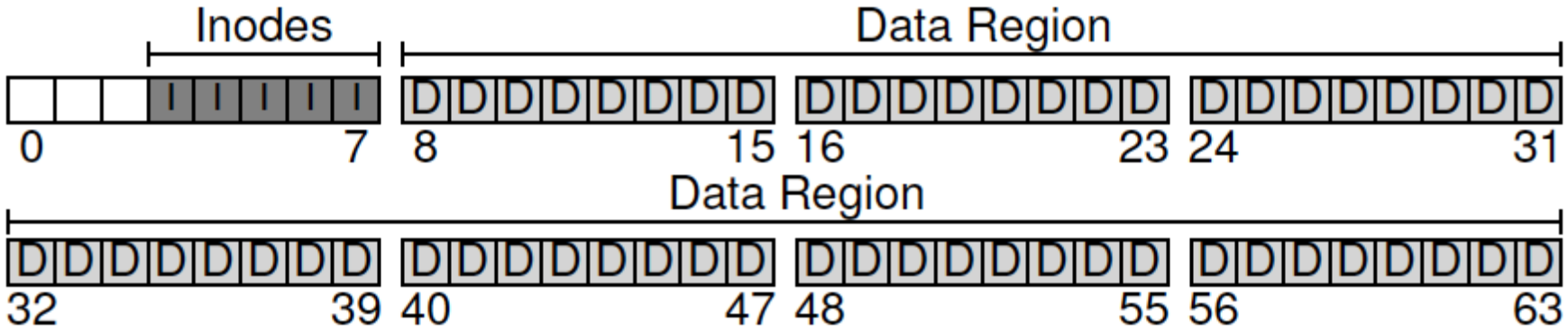
- Store data in blocks 8-63, leaving some space at the beginning to do book keeping
- This leaves us 4K*56 bytes to store data

Metadata

- For each file, file system keeps track of
  - Location of that blocks that comprise the file
  - Size of the file
  - Owner and access rights
  - Access and modify times
  - Etc. (see struct stat)

- These metadata are stored in **inode**

Storing File Metadata



- Save 5 blocks of space to store per-file metadata.
- Metadata stored in structured commonly called an inode (for index node)

How many inodes are there?

- 256 bytes/inode
- 4KB/block, 5 blocks

=> 16 inodes/block (4K/256 = $2^{12}/2^{8}$)

=> 5 blocks, 5 * 16 = 80 inodes

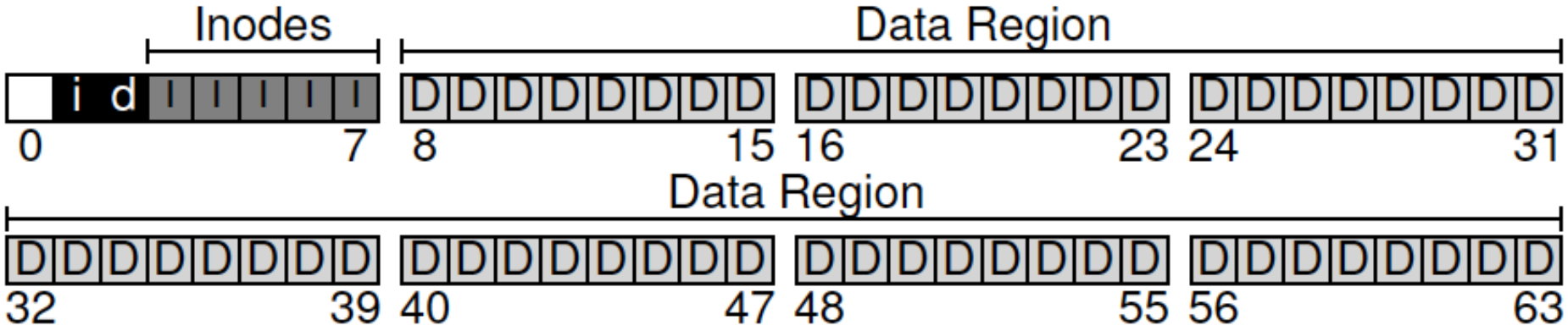
=> File system can store at most 80 files

Free space management using bitmaps

- Bitmap: a vector of bits
  - 0 for free, 1 for in-use

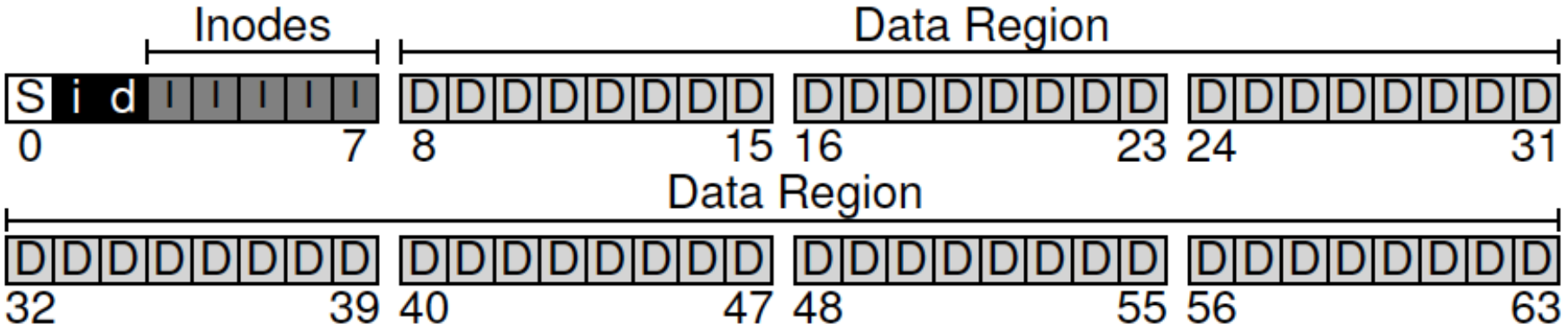
- Inode bitmap (imap)
  - keep track of which inodes in the inode table are available
- Data bitmap (dmap)
  - Keep track of which blocks in data region are available

Tracking free space



- Simple bitmap to track allocated blocks in inode and data region
  - Set bit to 1 if corresponding slot is used – 0101 means that we have used two inodes/blocks
  - Need 80 bits for inodes, and 56 bits for data blocks, so 4K block is somewhat wasteful

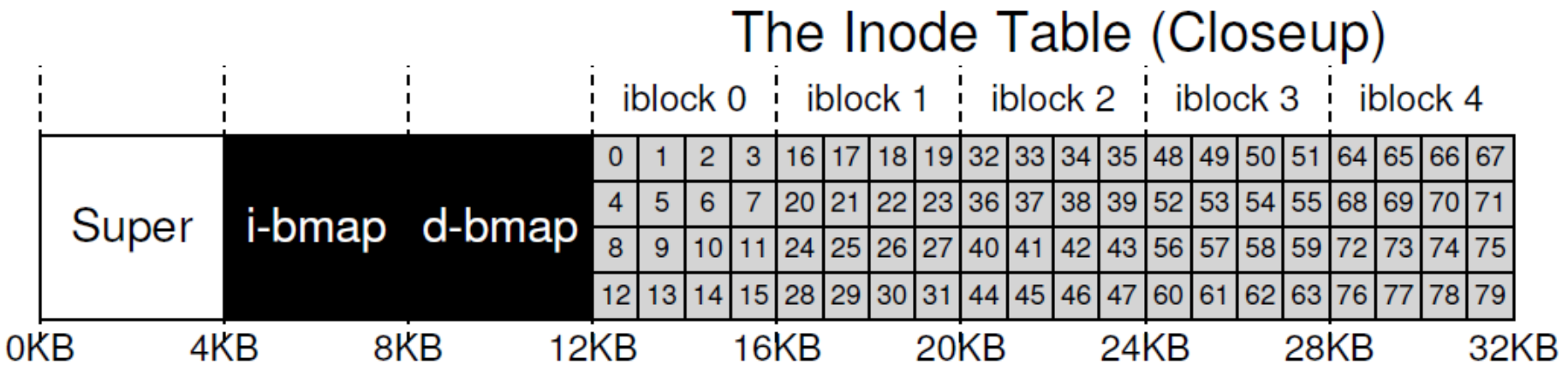
The SuperBlock!!



Need place to store information about the file system, such as size of partition, number of inode blocks, etc.

Inode (index node)

- Each is identified by a number
  - Low-level number of file name: inumber
- Can figure out location of inode from inumber

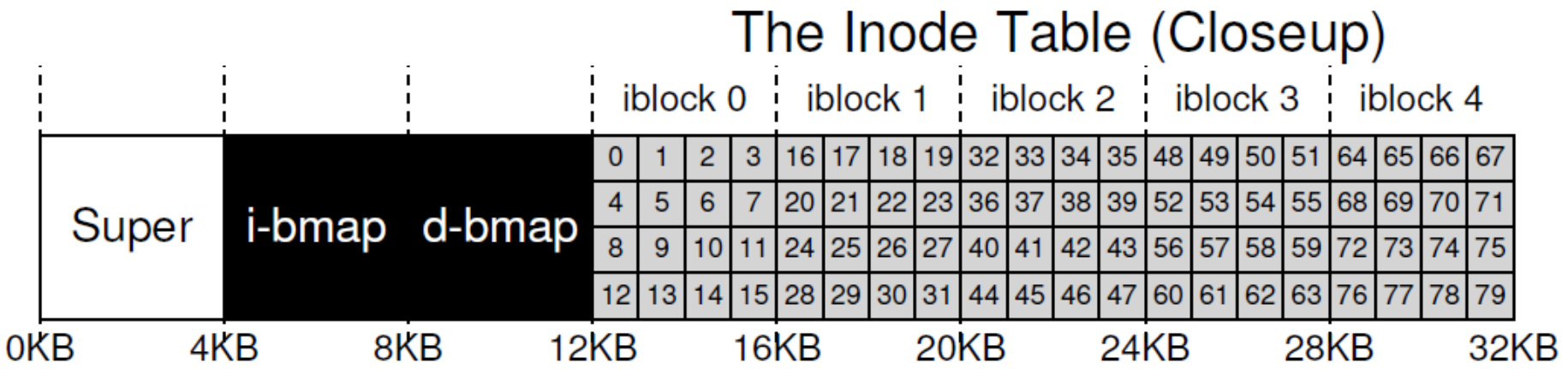


inumber => location

- inumber = 32

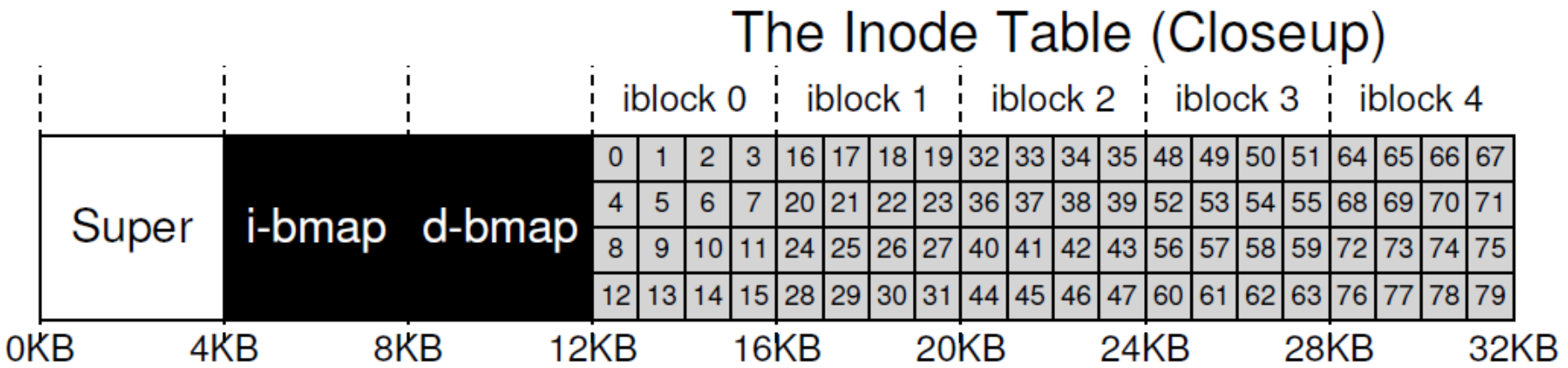
  => address: offset in bytes from the beginning

  => which sector?



inumber => location of inode

- Address: 12K + 32 * 256 = 20K
- Sector #: 20K/512 = 40
  - more generally
  - ⌊(inodeStartAddress + inumber ∗ inode size)/sector size ⌋



inode => location of data blocks

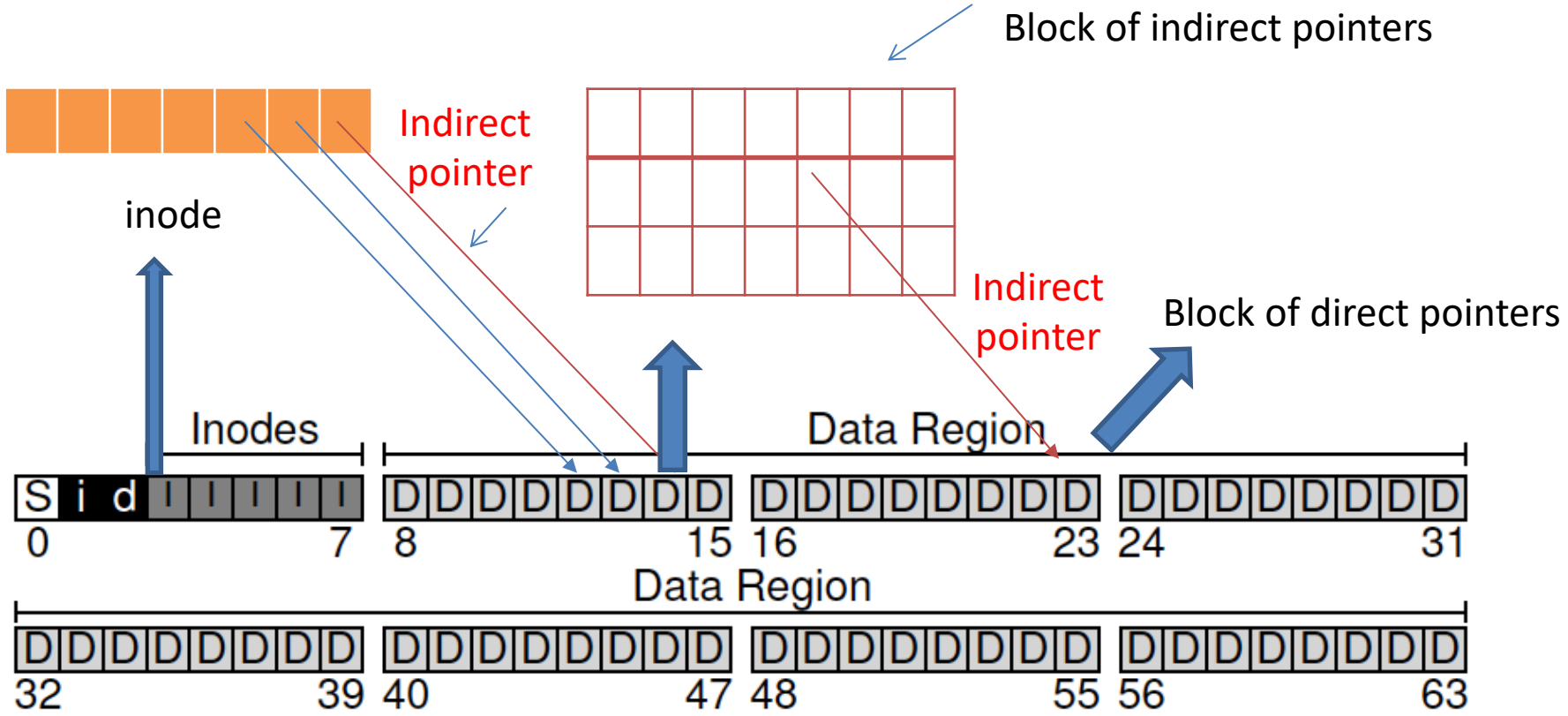
- A number of direct pointers
  - E.g., 8, each points to a block
  - Enough for 8*4K = 32K size of file

- Has a slot for indirect pointer
  - Point to a data block storing direct/indirect pointers
  - Assume 4 bytes for disk address, so 1024 pointers/block
  - Now file can have (8 + 1024) blocks or 4,128KB

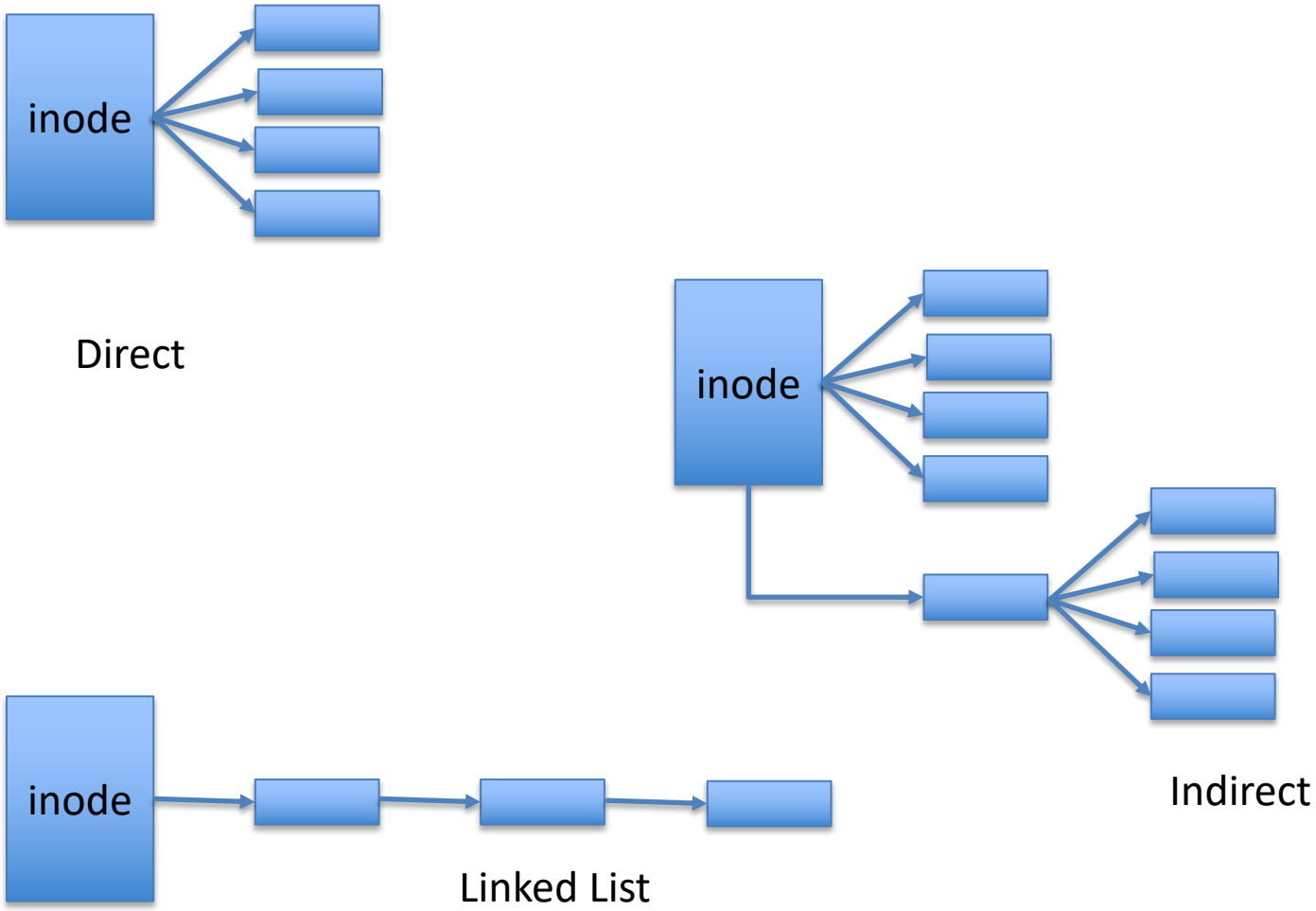
Multi-level index

- Pointers may be organized into multiple levels
  - Direct pointers & indirect pointer
    - Inode (pointer1, pointer2, …, indirect pointer)
    - Indirect pointer -> a block of direct pointers
  - Double indirect pointers
    - Inode (pointer1, pointer2, …, indirect pointer)
    - Indirect pointer -> a block of indirect pointers instead
      -> each points to a block of direct pointers
  - Triple indirect pointers
    - Indirect pointer -> a block of indirect pointers
      -> each points to a block of indirect pointers
      -> each points to a block of direct pointers

Double Indirect Pointer



Alternative ways to link blocks to inode



Why does indirect work well? File System Characteristics

- 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 systems remain ~50% full
- Directories are typically small
  - Many have few entries; most have 20 or fewer

Directories

- Just a list of file names

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

- Record length used to enable reuse from deleted entry
- Directory stored as a file

Record length vs string length

- String length = # of characters in file name + 1 (for \0: end of string)
- Record length >= string length
  - Due to entry reuse

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

Reusing directory entries

- If file is deleted or a name is unlinked
  - recall the "rm" and "unlink" commands
  - File is finally deleted when its last hard link is removed

- inumber in its directory entry set to 0 (reserved for empty entry)
  - So we know it can be reused

Storing a directory

- As a special file with inode + data block

- inode:
  - file type: directory (instead of regular file)
  - pointer to block in data region storing directory entries

Open for read

- fd = open("/foo/bar", O_RDONLY)

Open for read

- fd = open("/foo/bar", O_RDONLY)
- Need to find inode of file from its path “/foo/bar“

1. Find inode and content of root (usually root's inumber = 2)
2. Look for “foo” in the directory -> foo’s inumber -> foo’s inode
3. Read inode and content of foo directory
4. Look for bar in the directory -> bar’s inumber -> bar’s inode
5. Permission check + allocate file descriptor

Cost of open()

- Need 5 reads of inode/data block

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

Reading the file

- read(fd, buffer, size)
  - Note fd is maintained in per-process open-file table
  - Table translates fd -> inumber of file

File-open table per process

File descriptor	File name	Inumber	Position offset	…
3	"/foo/bar"	32382	0	
4	"/foo/more"	48482	512	…

Reading the file

- read(fd, buffer, size)

1. Consult bar's inode to locate its 1st block
2. Read the block
3. Update inode with newest file access time
4. Update open-file table with new offset
5. Continue steps 2, 3, 4 until done
6. Deallocate file descriptor

Cost of read()

- read inode, read data block, write inode

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

Writing a file: allocate block and inode

- Need to allocate a block
  - Read bmap to find free block
  - Write (actually update) bmap
  - Write the block
- Need to allocate an inode (if new file)
  - Read imap to find free inode
  - Write (update) imap
  - Write the inode

Writing a file: update directory

- Also need to update the directory
  - Update its data block: add inumber & name for new file
  - Update directory inode (book-keeping)
  - Might need to grow directory (add data block)

Example: Open for write “/foo/bar”

- int fd = open(“/foo/bar”, O_WRONLY)
  - Assume bar is a new file under foo

Example: open “/foo/bar” for write

- int fd = open(“/foo/bar”, O_WRONLY)

1. Read root inode (inumber = 2)
2. Read root content block => foo's inumber
3. Read foo inode
4. Read foo content block, check if bar exists

Example: open “/foo/bar” for write

5. Read imap, to find free inode for bar
6. Update imap, setting 1 for allocated inode
7. Write bar inode
8. Update foo content block (why?)
9. Update foo inode (why?)

Example: open “/foo/bar” for write

- int fd = open(“/foo/bar”, O_WRONLY)
- Need 9 I/O's

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

Example: writing to "/foo/bar"

1. Read inode of bar (by looking up its inumber in the file-open table)
2. Allocate new data block
   - Read and write bmap
3. Write to data block of bar
4. Update bar inode with new modified time, …

Example: write “/foo/bar”

- 5 I/O's for each write() of 4K (a block)

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

Caching and buffering

- Caching for read
  - First read slow, subsequent ones speed up
  - Cache popular ones, e.g., determined via LRU strategy
- Buffering for delayed write
  - Batching (two updates to the same imap)
  - Scheduling (reordering for better performance)
  - Avoiding writes (if file created, then quickly deleted)
- Problem: update may be lost when system crashes

Side note: file systems that you know

- NTFS
  - New technology file system, Microsoft proprietary
- FAT
  - File allocation table
  - FAT 16, 32, …
  - 32 bits = # of sectors a file can occupy
    512B/sector => 2TB limit on file size
    4KB/sector => 16TB limit
- Ext4
  - fourth extended file system, common on Linux

Next up?

- We look at how we can share files across multiple computers