The VFS & A Reference File System

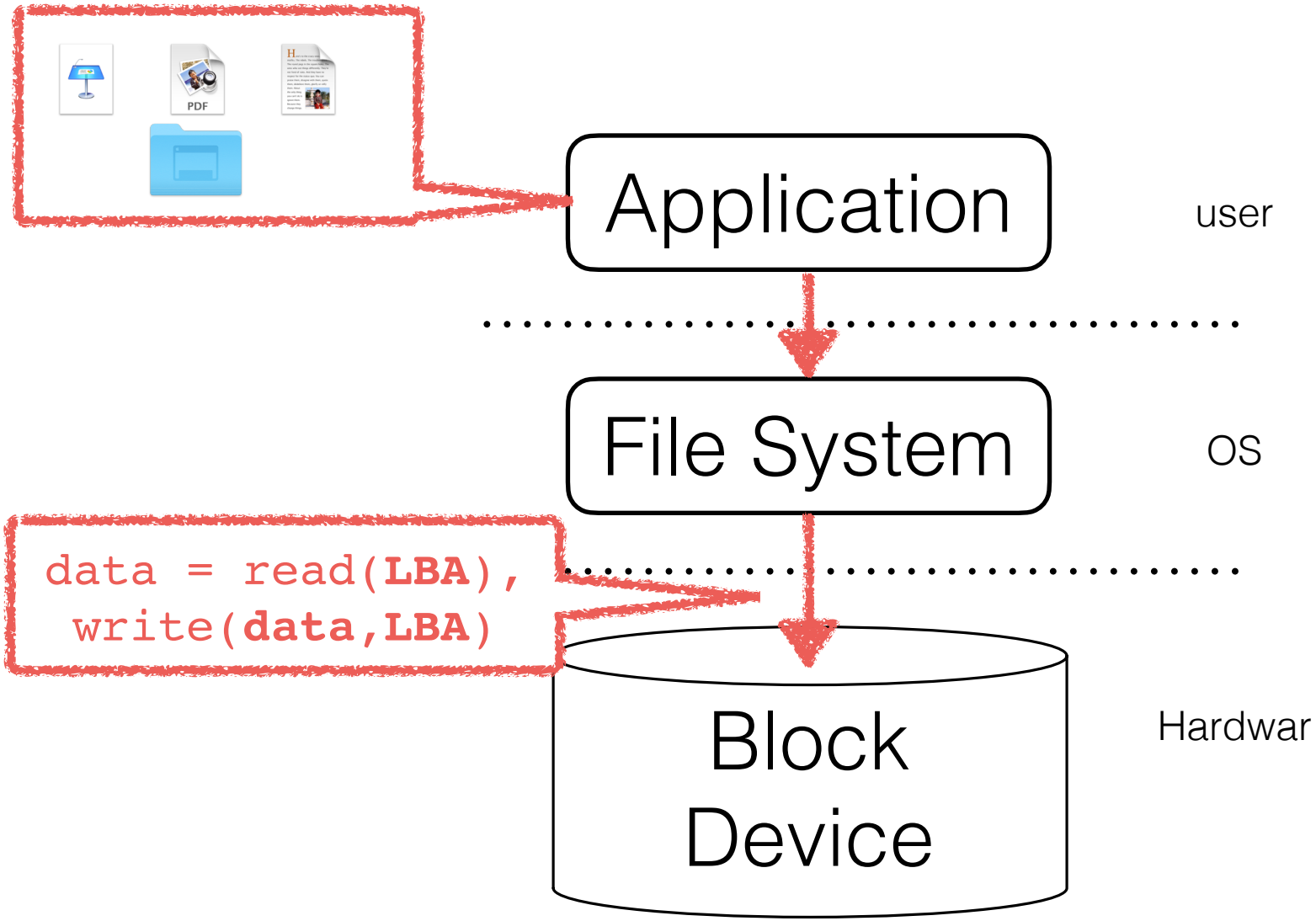
CSCI 333 :: Storage Systems
Williams College

Lecture Outline

This lecture covers the general design and structure of a reference file system ("VSFS")

- What is the role of the file system?
- What types of information does it need to manage to do its job?
- How does the file system organize data on a disk?
- How does the file system traverse this data when performing representative file system operations?

Simplified Storage Stack



The File System's Role

A file system implements the file abstraction over an array of blocks. A file system must:

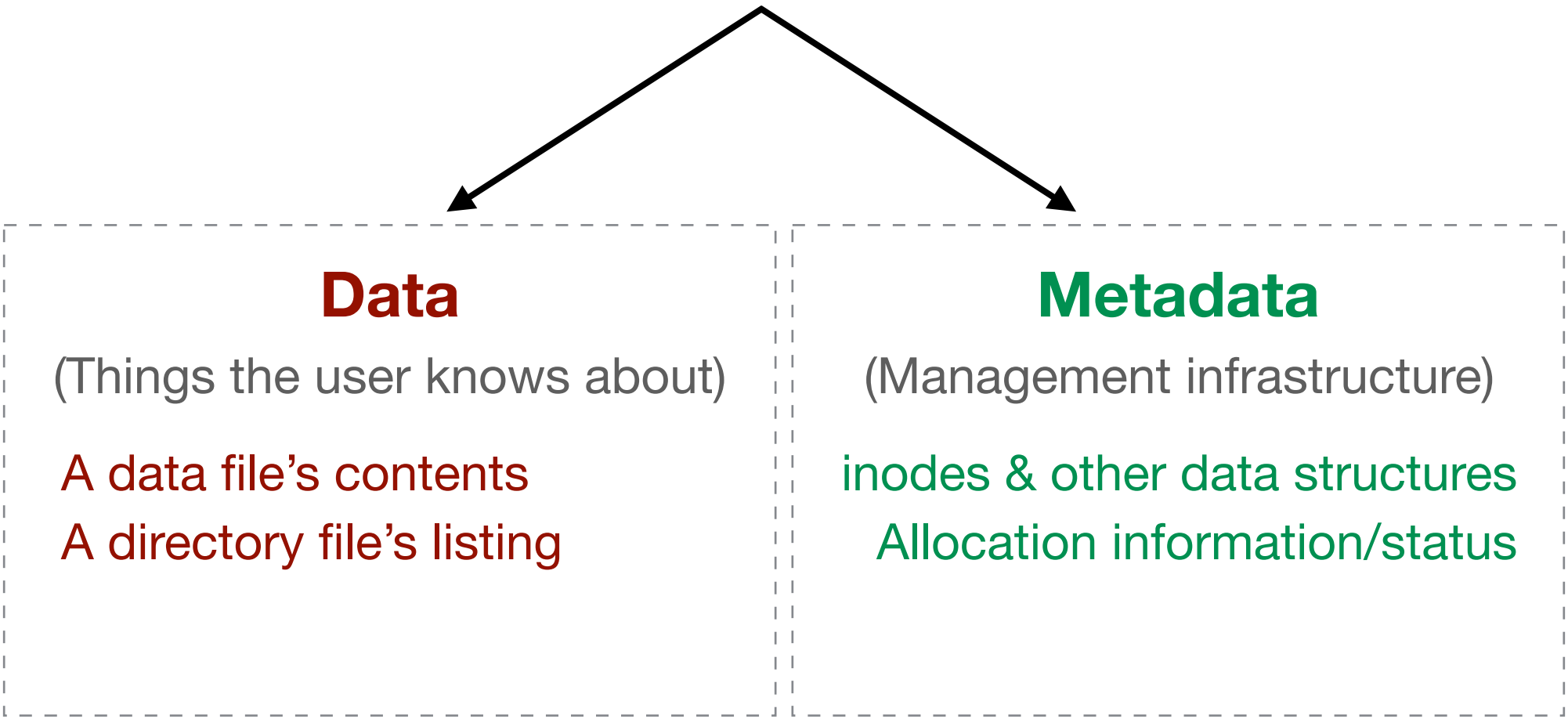
- translate application requests into block requests
- impose access controls on our data
- ensure consistency in the presence of failures

- ~~perform well~~?

Basic Organization

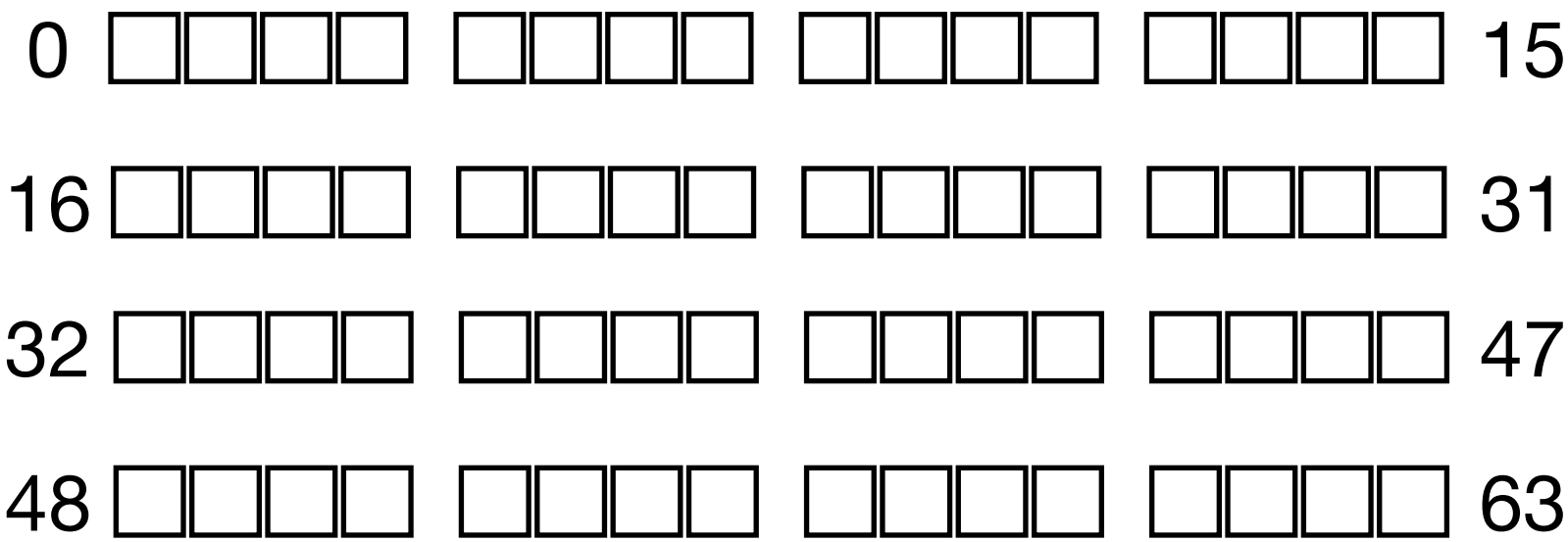
A disk is divided into a logical array of blocks

- A file system takes ownership of some partition, and imposes structure using two types of information:



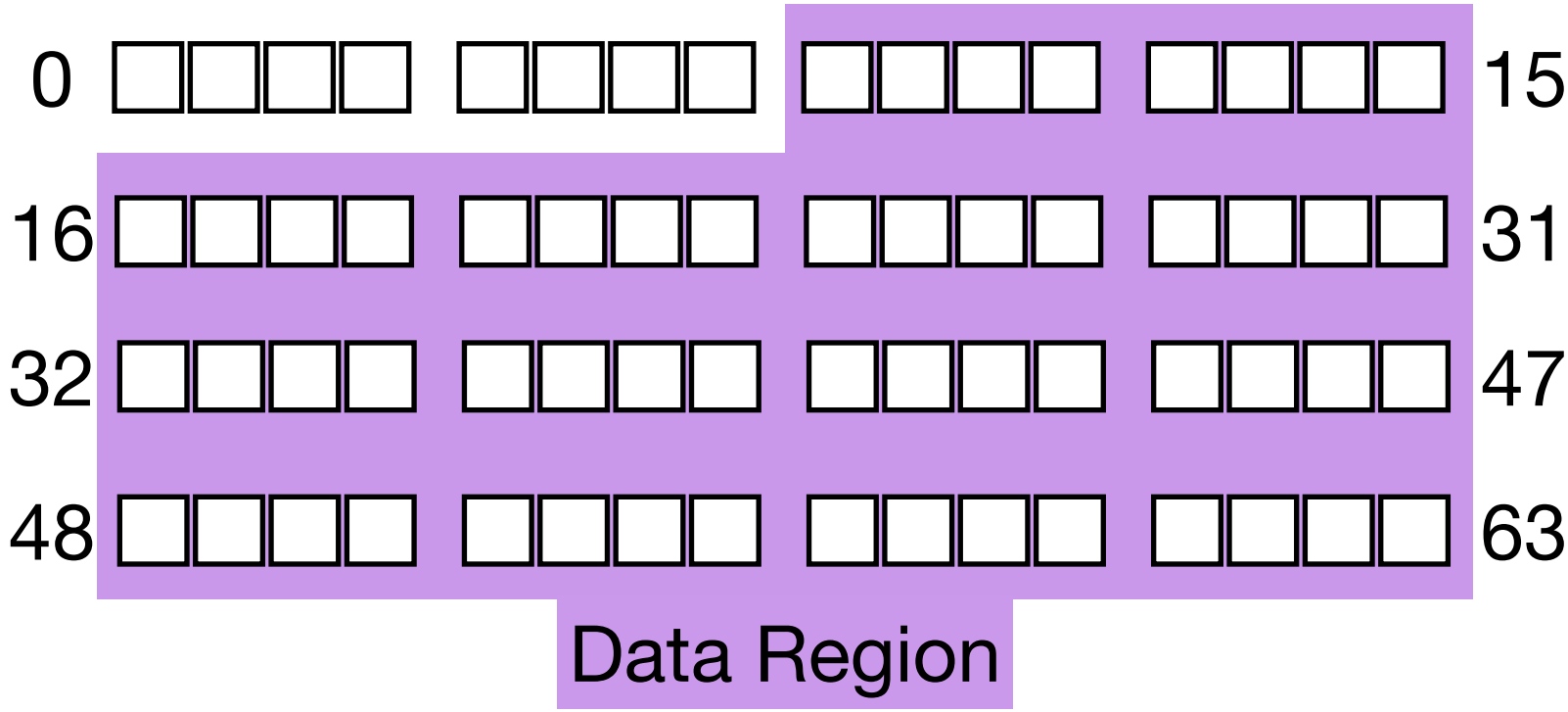
A Reference File System

Consider this picture of a 64-block disk partition. How might a file system manage these blocks?



First, let's set aside a dedicated region for storing file data.

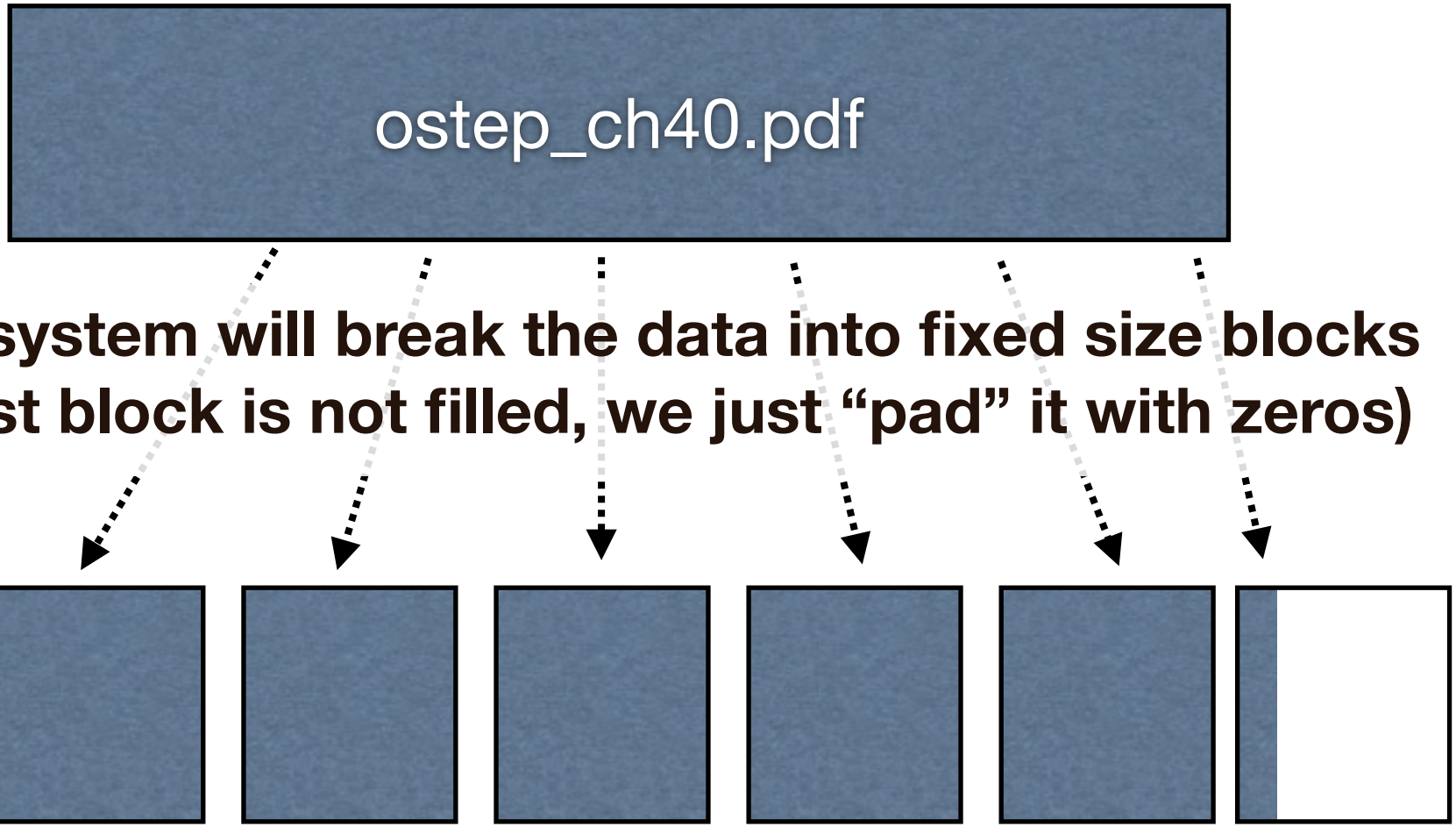
A Reference File System



Now, let's see how file data ends up in the data region.

Storing File Data in Blocks

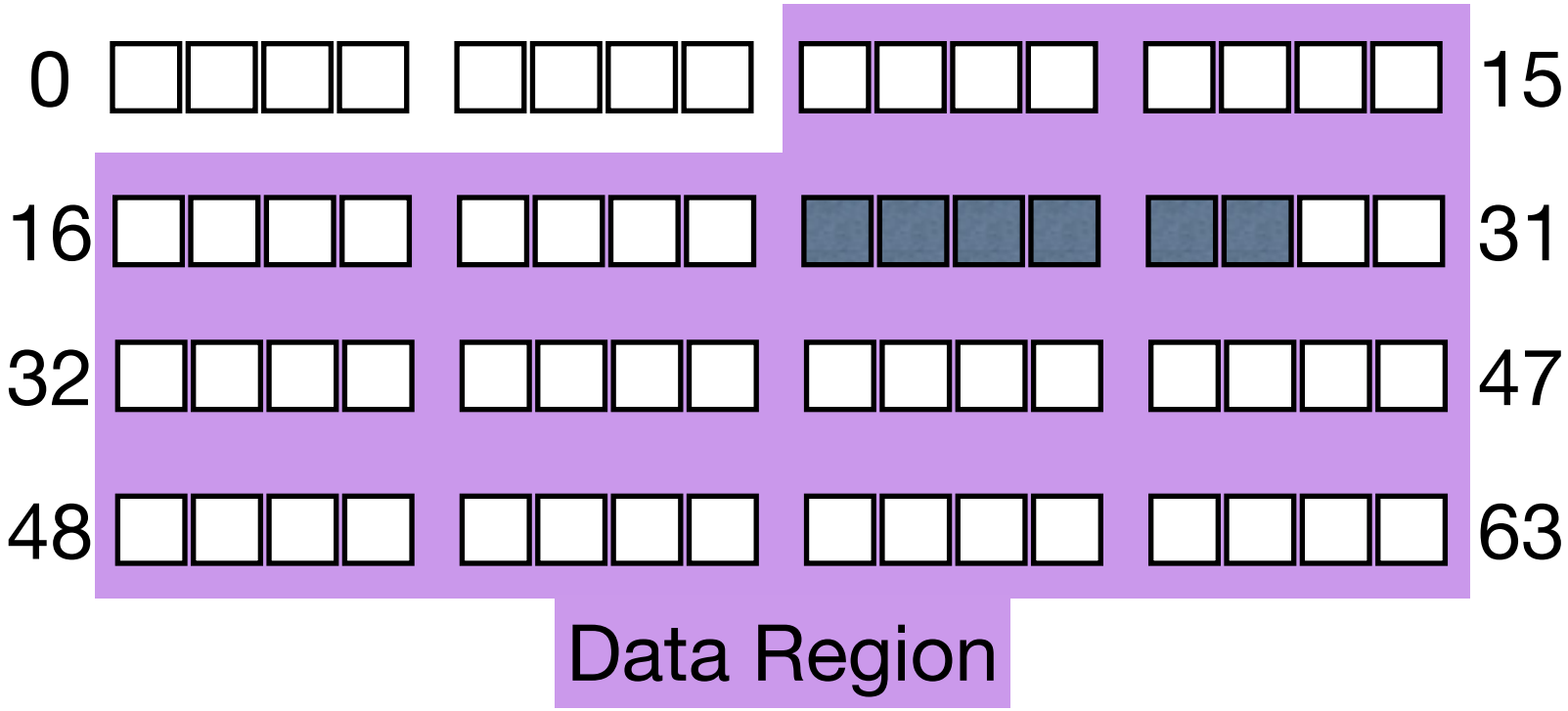
Suppose I have a large file, like OSTEP Ch40, stored in some memory region.



The file system will break the data into fixed size blocks (if the last block is not filled, we just "pad" it with zeros)

Those blocks are stored somewhere in the data region

A Reference File System



Logically Organizing Data

Beyond storing blocks, an FS also needs to:

- Track which blocks in the data region are allocated and which are free
- Associate allocated blocks with a specific file
- Associate regions within a file to particular blocks
- Ensure only authorized users can access data

FS Metadata structures are needed to index our user data and impose order.

Tracking Allocations

There are many ways to track which blocks are in use and which are free:

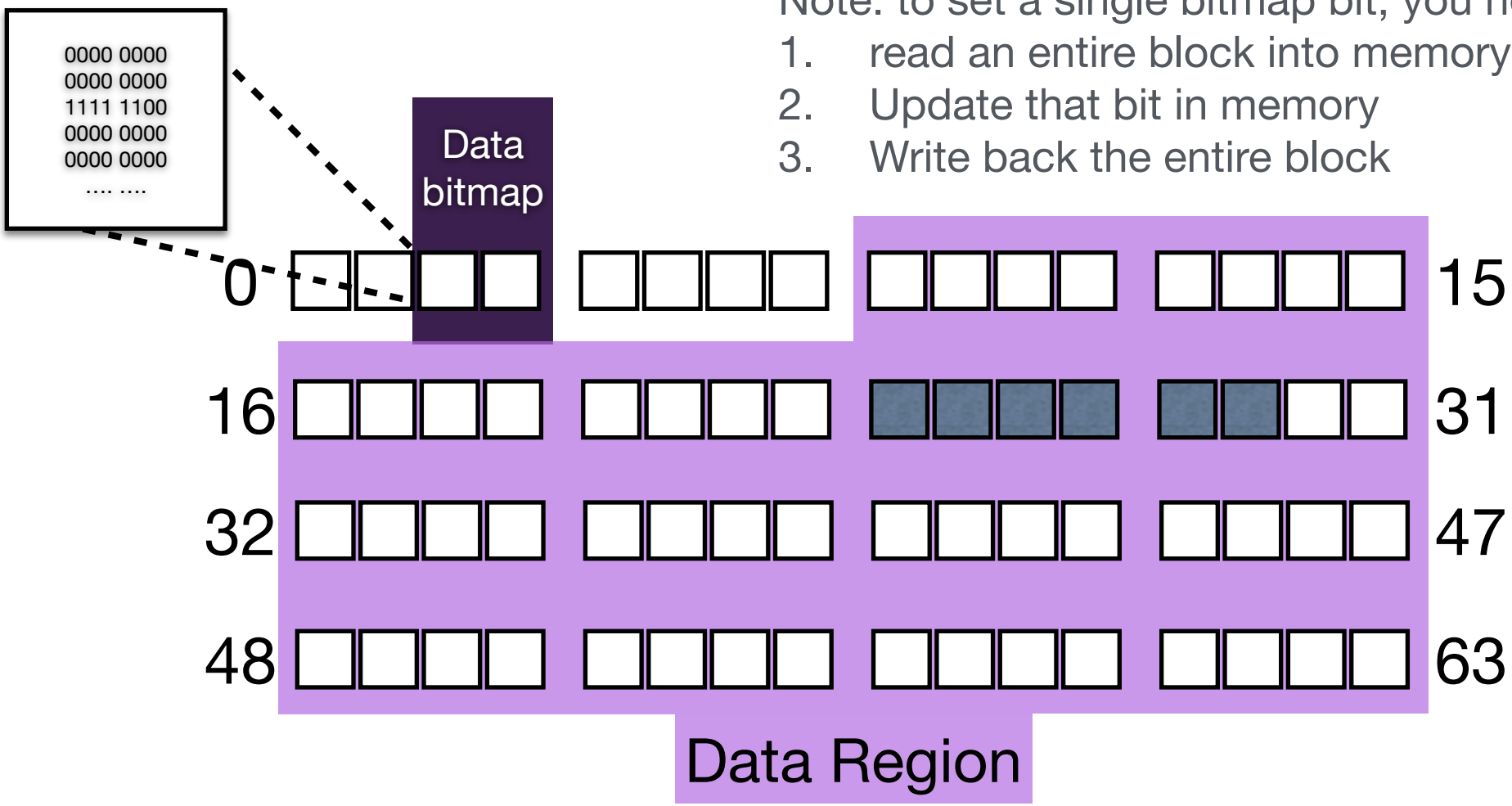
- Create a "bitmap" with one bit per block
  - 0 = free, 1 = allocated
  - Efficient when we want to allocate/free blocks one at a time
- Use "extents" to track free blocks
  - (start,end) pairs that describe a range on contiguous blocks
  - Efficient when we allocate large regions & have good locality
- Use a linked list of free blocks
  - Costs of (not) sorting?
- Use more advanced data structures, like trees
  - Can be combined with the techniques above (trees of extents?)

We'll show the simplest approach: a bitmap to manage our allocations

A Reference File System

Note: to set a single bitmap bit, you need to:

1. read an entire block into memory
2. Update that bit in memory
3. Write back the entire block



The Inode

Inodes: persistent metadata about a single file

- Permissions & ownership
- Size (logical and physical)
- Locations of the file's contents
- Link count,
- …

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**

Managing Inodes

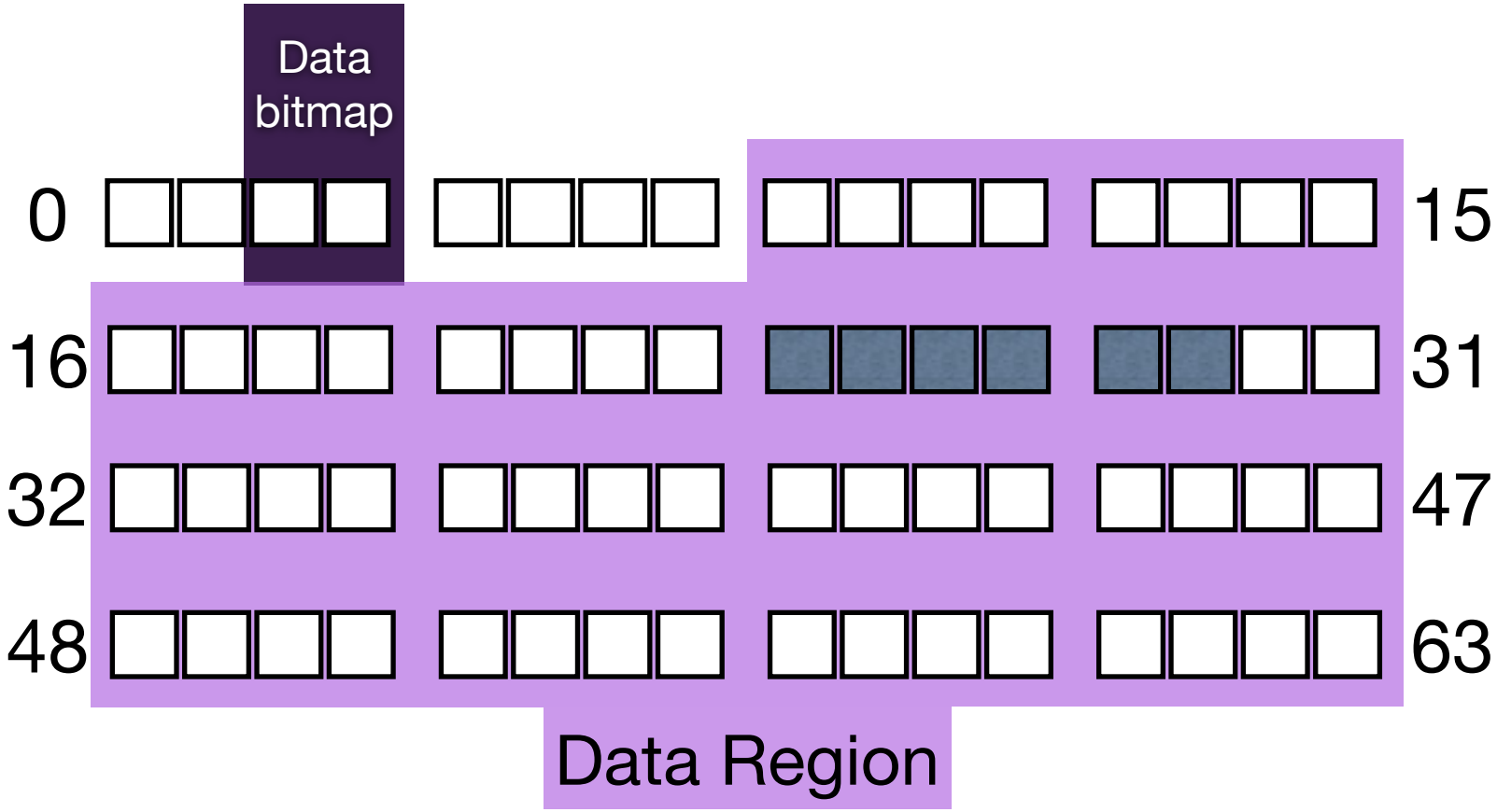
Where do we store our inodes?

- They aren't data, so we don't want to mix them in the data region
- They are small, so they don't take up a whole disk block

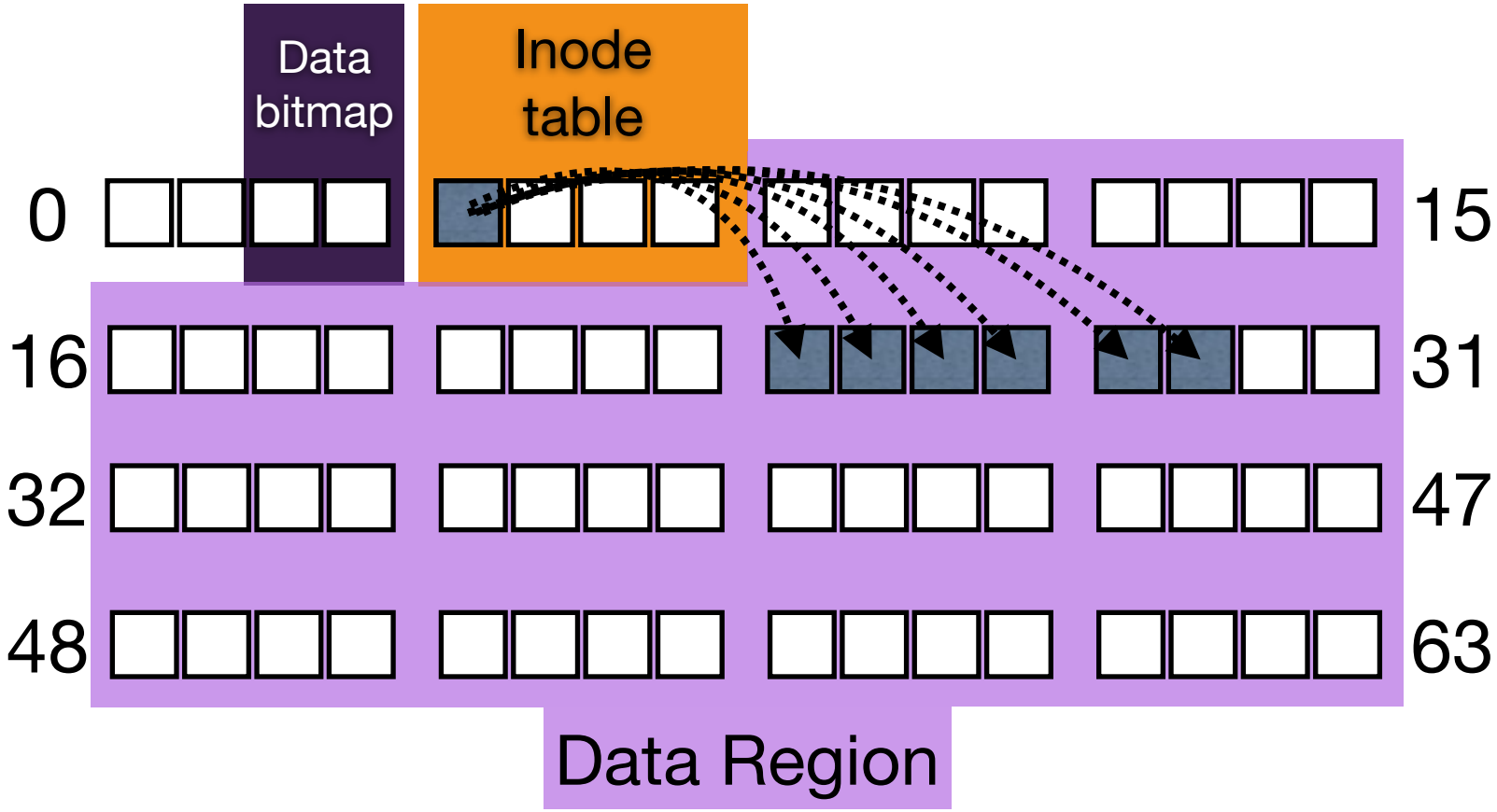
We'll create an inode table

- If we know the inode size, the block size, and the start of the inode table, we can use math to:
  - Find the disk block where an inode resides
  - Find the offset within the disk block

A Reference File System



A Reference File System



Managing Metadata

Now we need some additional metadata to manage our metadata

- How do we know which inodes are allocated/free?
  - Another bitmap!
- How do we know where our structures are located? We need to know:
  - Bitmap locations
  - Inode locations (inode table start)
  - Number of inodes
  - Data region start
  - Number of data blocks

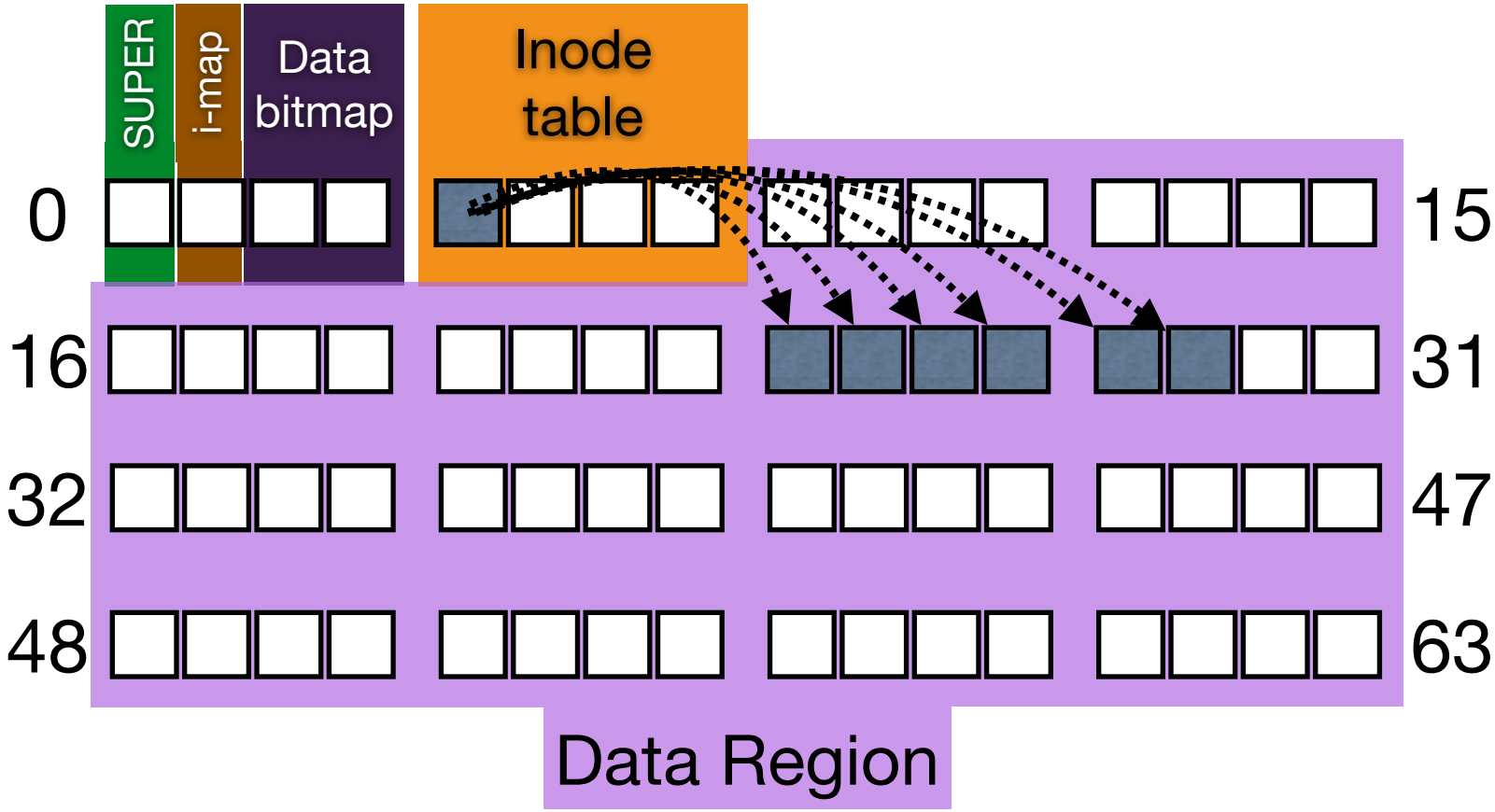
The superblock stores information about the whole file system

- Information we need to "bootstrap" our file system (data listed above)
- Often contains other miscellaneous parameters, if tunable
  - Block size (a multiple of sector size)
  - Encryption keys
  - Compression formats
  - ...

Since the superblock is so important, it is often treated specially

- Stored at a known location (first block of partition?)
- Often replicated elsewhere for safety

A Reference File System



Creating a Hierarchy

What about directories?

Beyond file data, we also need infrastructure to:

- Traverse through our namespace to find a target file
  - We start traversing at `/`
  - We must confirm that we have sufficient rights to access each component along the path
    - To access `/foo/bar/file.txt`, we need permissions to access: `/`, `foo`, `bar`, and `file.txt`
- Add, remove, or delete children from our directories

Where is our directory structure kept?

Directories: Data or Metadata?

Users can:

- Create a directory (`mkdir`)
- Query directory continents (`readdir`)

Users cannot:

- Directly modify a directory
  - Directory contents are modified ***indirectly*** by creating/deleting children

What do directories need?

- Permissions
  - To enforce access and "traversal" through namespace
- Size, link counts, access times, …
  - They need everything we store in an inode!

We'll create explicit directory files with their own inodes and data blocks.

Directories: Data or Metadata?

Example Directory Format

- Inode points to a directory's data blocks
- Data blocks store a "table" of name, inode records
  - How to format the table?
    - Are all names the same length?
      - No
    - Are all inode numbers the same length?
      - Yes

inum	strlen	Name
7	2	.
5	3	..
55	4	foo
56	30	variable_lengths_are_hard.txt

Putting It All Together

Let's walk through sample operations, and see how those operations translate into accesses to our reference file system's data structures

Access Patterns: Creating a file, then writing 3 blocks

	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)										
write()										
write()										
write()										

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

Access Patterns: Creating a File, then writing three blocks

	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**								1. Read / inode to find location of directory data
						read					2. Read / data to find inode number for child foo
				read							3. Read foo inode to find location of directory data
							read				4. Read foo data to confirm that bar does not exist
		read									5. Read inode bitmap to find a free inode bit
		write									6. Write inode bitmap to mark inode as allocated
							write				7. Write foo data to include new child bar
					read						8. Read bar inode block (bar's inode is small part)
					write						9. Update bar's inode and write back modified block
				write							10. Update foo's inode (modification time, size, links)
write()					**read**						1. Read block containing bar's inode. Is there existing data?
	read										2. Allocate a new bit for data block (read containing block, modify single bit, write back whole block)
	write										
								write			3. Write data to newly allocated data block
					write						4. Update bar's inode, and write back the containing block
write()					**read**						
	read										
	write										
									write		(Repeat steps above)
					write						
write()					**read**						
	read										
	write										
										write	(Repeat steps above)
					write						

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

Access Patterns: open("/foo/bar"), read three blocks

	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()										

Access Patterns: open("/foo/bar"), read()

	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					

1. Read / inode to find location of directory data
2. Read / data to find inode number for child foo
3. Read foo inode to find location of directory data
4. Read foo data to find bar inode number
5. Read bar inode to check permissions & open
6. Read bar inode to identify location of first block
7. Read bar's first data block
8. Update bar's inode with access time

(Repeat 6-8 for each read)

Summary

File Systems impose order on our disks.

In our reference FS:

- A user's file data is divided into fixed-size blocks
  - Directory files have data too!
    - Well-defined format that internal FS functions read/write to satisfy user requests
- File system metadata indexes that user data
  - Inodes keep track of which blocks correspond to which file and how
  - Bitmaps are one way to track allocation status
  - The superblock summarizes all of the metadata, needed to "bootstrap" the file system

We've so far described a "***working***" file system design. In future units, we'll think about how to design ***efficient*** file systems.