File System Implementation

CMPU 334 – Operating Systems
Jason Waterman

Introduction

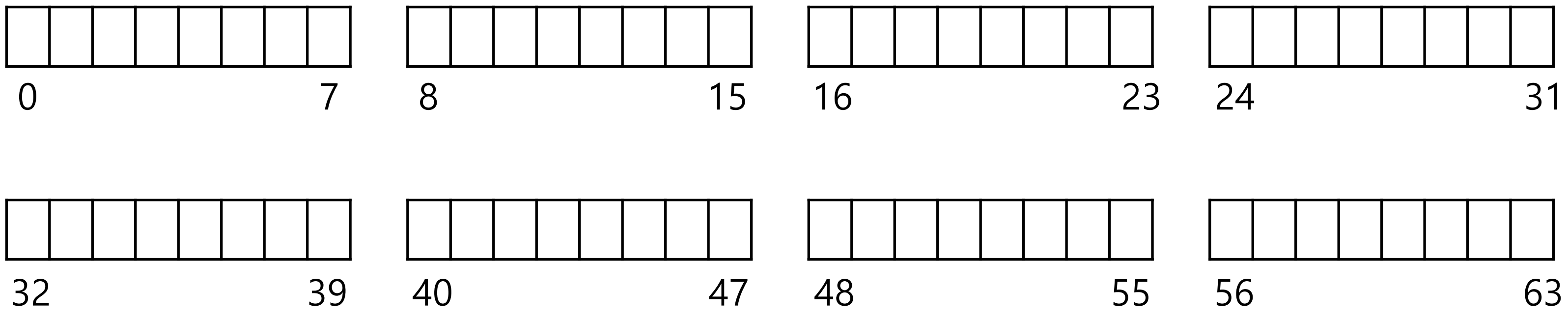
- Goals for today:
  - To build a simple file system (vsfs the Very Simple File System)
  - Simplified version of a typical UNIX file system
  - Introduces basic on-disk structures, access methods, and policies found in typical file systems

- There are two different aspects to implement file system
  - **Data structures**
    - The 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: Disk Blocks

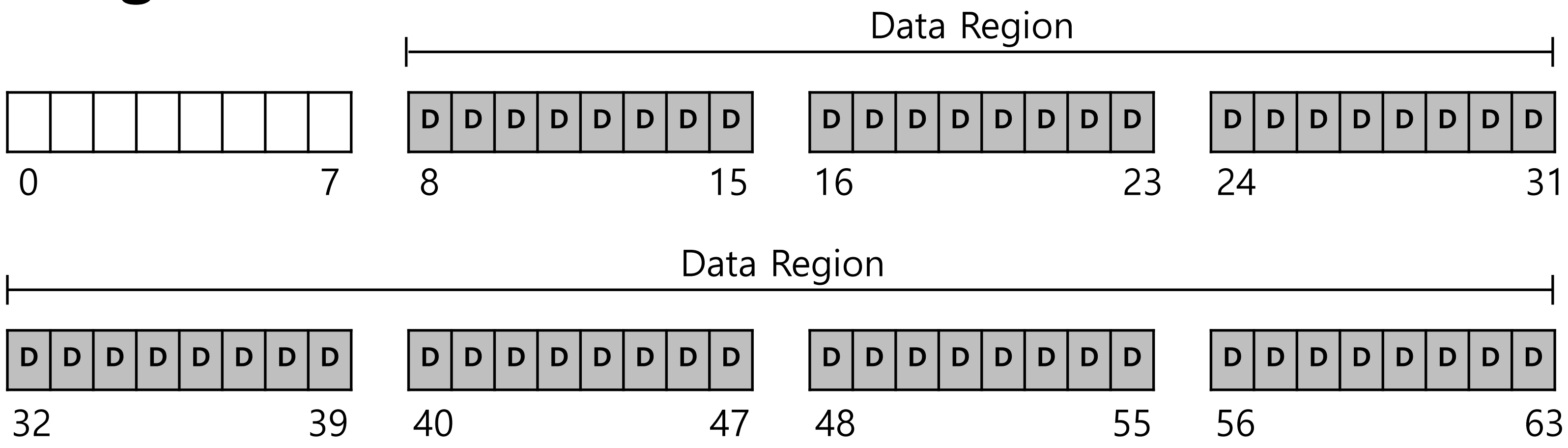
- Let's develop the overall organization of the file system data structure

- Divide the disk into **blocks**
  - Block size is 4 KB
  - The blocks are addressed from `0 to N-1`
  - Example below shows a very small drive with 64 blocks



Data region in the filesystem

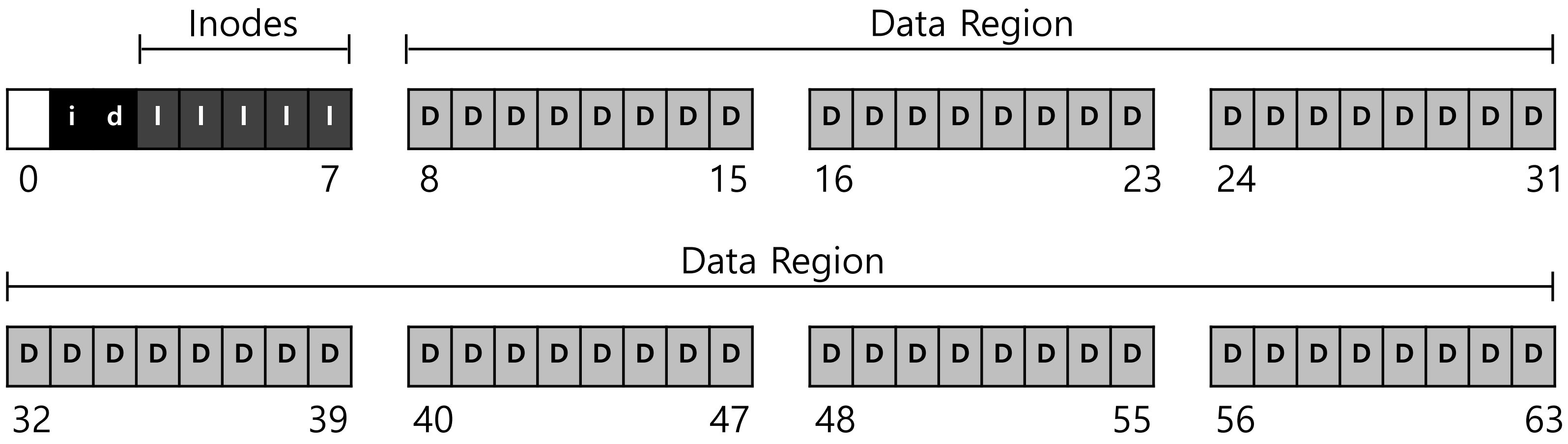
- Reserve **data region** to store user data



- File system has to track information about each file
  - We'll use **inodes** do to this

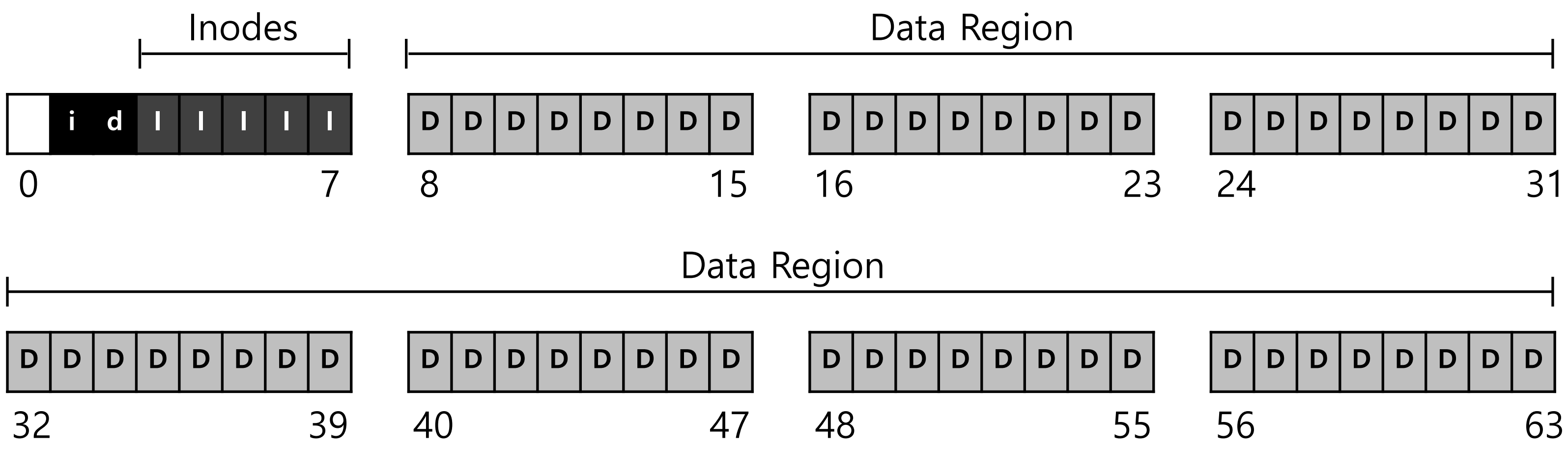
Inode table in the filesystem

- An inode tracks which data blocks comprise a file, its size, owner, and other metadata
  - One inode per file in the system
- Reserve space for **inode table**
  - An array of on-disk inodes (blocks 3 to 7 in our filesystem)
    - inode size : 256 bytes
    - 4-KB block can hold 16 inodes
    - The filesystem contains 80 inodes (maximum number of files supported in this filesystem)



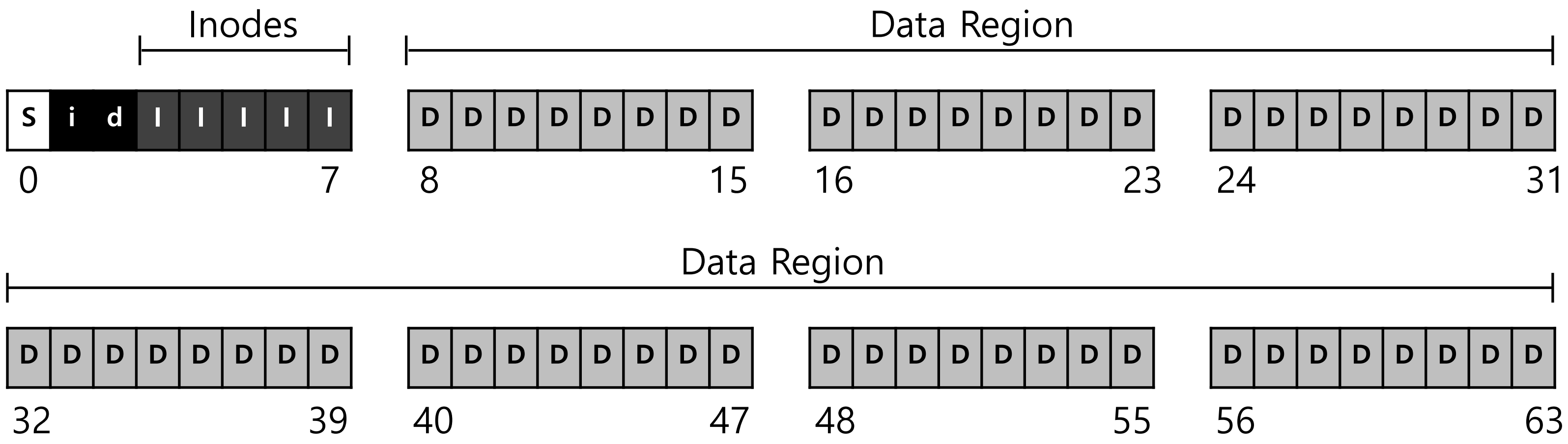
Allocation structures

- 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 for data region
  - **inode bitmap**: for inode table



Superblock

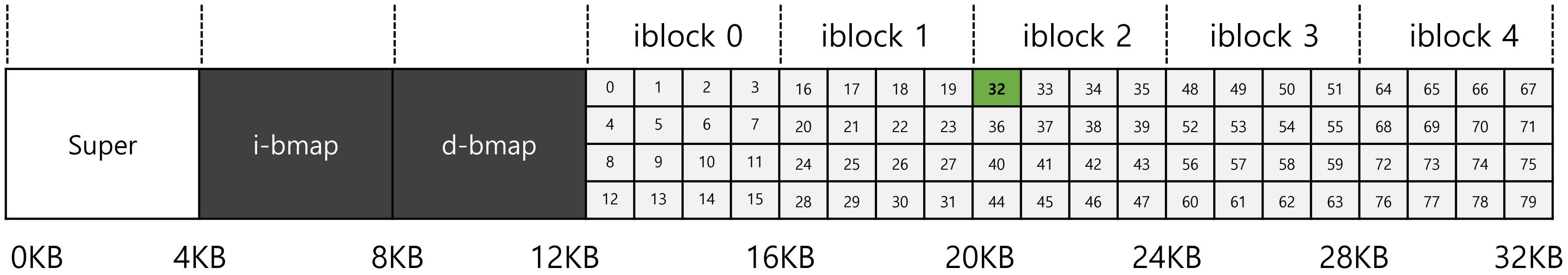
- Contains meta information for the file system
  - E.g., the number of inodes, beginning location of inode table, etc.



- When mounting a file system, the OS will read the superblock first to initialize various information for the filesystem

File Organization: The inode

- Each inode is referred to by its inode number
  - The filesystem can calculate where the inode is on the disk
  - Example: calculate the location of inode number 32
    - Calculate the offset into the inode region (32 x `sizeof(inode)` (256 bytes) = 8192
    - Add start address of the inode table (12 KB) + inode region (8 KB) = 20 KB
- Disks are not byte addressable, they are disk sector addressable
  - 20 KB / 512 (sector size) = 40
- To get the sector address of the inode block:
  - `sector = (inode * sizeof(inode_t) + inodeStartAddr) / sectorsize`



File Organization: The inode (Cont.)

- The `inode` has all of the information about a file (e.g., `ext2` inode)
  - File type (regular file, directory, etc.),
  - Size, the number of blocks allocated to it
  - Protection information(who owns the file, who can access, etc.)
  - Time information
  - Etc.

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

Finding file data: direct pointers

- How does an inode know where the data blocks for the file are?
  - Direct pointers
    - An ordered list of one or more disk addresses kept inside the inode
    - Each pointer refers to one disk block that belongs to the file
  - Limitations
    - An inode has a fixed size – you have a limited number of direct pointers you can store in there
    - 8-15 direct pointers are typical → 32 Kb to 60 Kb max file size
    - How do you support larger files?

The Multi-Level Index

- To support bigger files, we use multi-level index
- **Indirect pointer** points to a block (in the data section of the disk) that contains more pointers
- Typical setup:
  - inodes have fixed number of direct pointers (e.g., 12) and a single indirect pointer
  - If a file grows large enough, an indirect block is allocated
  - The inode’s slot for an indirect pointer is set to it
    - Now have an additional 4096 / 4 (bytes per disk address) = 1024 pointers
    - Max file size: (12 + 1024) x 4 KB or 4144 KB (little over 4 MB)
- What if we need larger files?

The Multi-Level Index (Cont.)

- **Double indirect pointer** points to a block that contains indirect blocks
  - Allow file to grow with an additional $1024 \text{ x } 1024$ or 1 million 4KB blocks (4 GB)
- Even larger files?
  - **Triple indirect pointer** points to a block that contains double indirect blocks

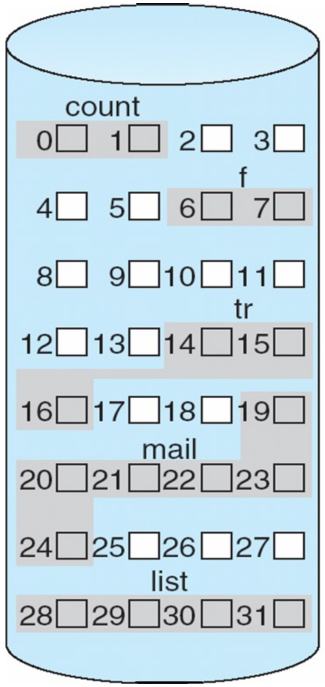
- Multi-Level Index approach to pointing to file blocks
  - Don't want the overhead of all these layers of indirection for small files
  - Example: twelve direct pointers, a single and a double indirect block
    - over 4GB in size $(12+1024+1024^2) \text{ x } 4\text{KB}$
- Many file system use a multi-level index
  - Linux EXT2, EXT3, NetApp's WAFL, Unix file system
  - Linux EXT4 use **extents** instead of simple pointers

The Multi-Level Index (Cont.)

- 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 system remain -50% full
  - Directories are typically small
    - Most have 20 or fewer entries

Contiguous Allocation

- An allocation method refers to how disk blocks are allocated for files:
- Contiguous allocation – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

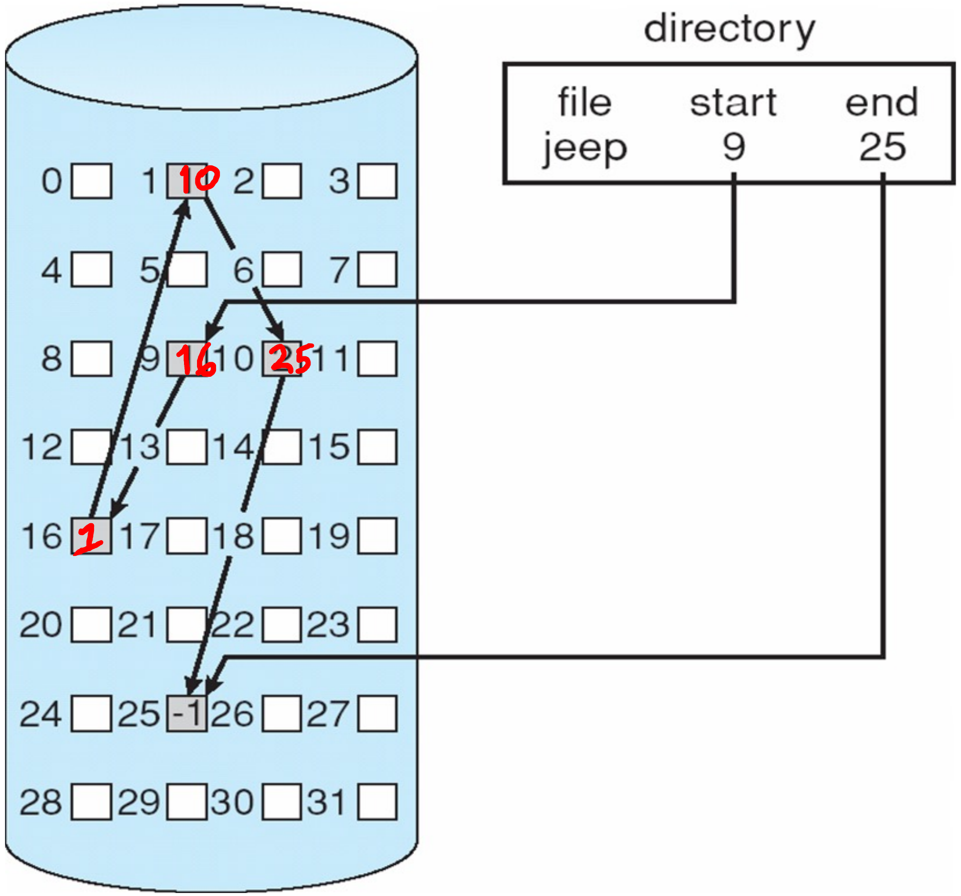
- Many newer file systems (i.e., ext4) use a modified contiguous allocation scheme

- Extent-based file systems allocate disk blocks in extents

- An extent is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

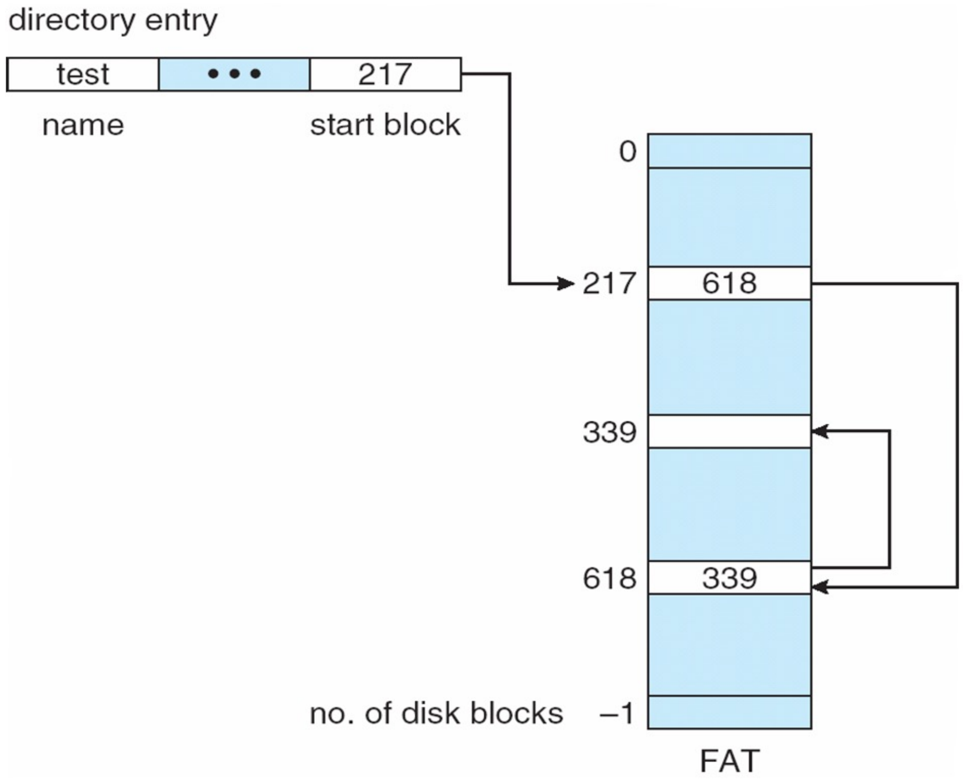
Allocation Methods - Linked

- Linked allocation – each file a linked list of blocks
  - File ends at null pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks



Allocation Methods – Linked (Cont.)

- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Like a linked list, but faster on disk and cacheable
  - New block allocation simple
- In this example, the file `test` is located at the following blocks:
  - `217, 618, 339`



Directory Organization

- Directory contains a list of (entry name, inode number) pairs
- Each directory has two extra files
  - . “dot” for current directory
  - .. “dot-dot” for parent directory
- For example, `dir` has three files (`foo, bar, foobar_is_long`)
  - Record length: total number of an entry plus any leftover space

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	24	14	foobar_is_long

On-disk data for `dir`

Free Space Management

- The filesystem tracks which inodes and data blocks are free or in use
- In order to manage free space, we have two simple bitmaps
  - One for inode blocks and one for data blocks
  - 10100100001111110101101101111111101101010110111101010101
- When file is newly created, it allocates an inode by searching the inode bitmap and updates on-disk bitmap
  - Pre-allocation policy is commonly used to allocate contiguous blocks

```
prompt> echo hello > foo
prompt> stat foo
  File: 'foo'
  Size: 6          Blocks: 8          IO Block: 1048576 regular file
Device: 35h/53d Inode: 1048872     Links: 1
Access: (0600/-rw-------)  Uid: (  328/jawaterman)   Gid: (   84/ faculty)
Access: 2019-11-19 19:50:21.340626373 -0500
Modify: 2019-11-19 19:50:21.340626373 -0500
Change: 2019-11-19 19:50:21.340626373 -0500
```

Access Paths: Opening a File From Disk

- Issue an `open("/foo/bar", O_RDONLY)`
  - Traverse the pathname and thus locate the desired inode
  - Begin at the root of the file system (/)
    - In most Unix file systems, the root inode number is 2
      - Aside: why start with inode 2?
        - Inode 0 is used to represent "null inode"
        - inode 1 has been historically used to keep track of any bad blocks in the system
  - Filesystem reads in the block that contains inode number 2
  - Look inside of it to find pointer to data blocks (contents of the root dir)
  - By reading in one or more directory data blocks, It will find "foo" directory
  - Traverse recursively the path name until the desired inode ("bar")
  - Check file permissions, allocate a file descriptor for this process and return the file descriptor to user
  - Whew!

Access Paths: Reading a File From Disk

- Issue `read()` to read from the file
  - Read in the first block of the file, consulting the 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, the file offset

- When file is closed
  - File descriptor should be deallocated
  - No disk I/O takes place

Access Paths: Reading `/foo/bar` Timeline

	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
    - Generates five I/Os:
      - one to read the data bitmap to find a free data block
      - one to write the bitmap (to reflect its 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 `/foo/bar` Timeline

	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 read of 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 systems used fixed-size cache to hold popular blocks
    - Static partitioning of memory can be wasteful
  - Modem systems use dynamic partitioning approach, unified page cache
    - Shared with virtual memory paging
- Read I/O can be avoided by a large cache

Caching and Buffering (Cont.)

- Write traffic must go to the disk for persistence
  - Disk cache does not reduce write I/Os
- The file system uses write buffering for write performance benefits
  - Delaying writes (file system batches 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 delaying writes, you may be able to avoiding them
    - E.g., creating a temp file that is quickly deleted
- Some applications force flush data to disk by calling `fsync()` or by using direct I/O calls that bypass the cache

Summary

- We looked at the basic machinery required for a filesystem
  - Need to store metadata information for each file
    - Inode
  - A directory is just a file that stores inode to name mappings
    - inodes are for computers, names are for humans
  - Also need to keep track of allocated inodes and data blocks
- Next time:
  - Look at performance
  - How can we make the filesystem faster