Operating System (OS)
CS232

Persistence: File system implementation

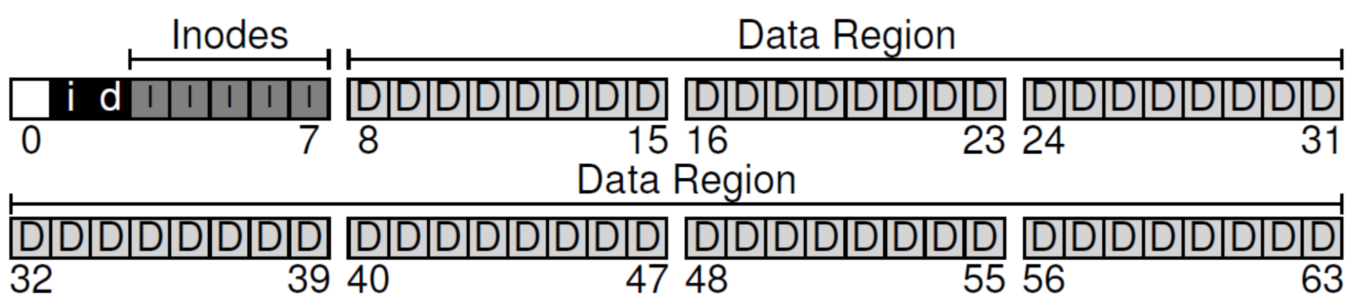
Dr. Muhammad Mobeen Movania

Outlines

- How allocations are tracked?
- How are file systems implemented
- What is the inode structure
- How inode blocks and data blocks are stored when a file object is made
- What happens when a file is read or written to
- Performance considerations
- Summary

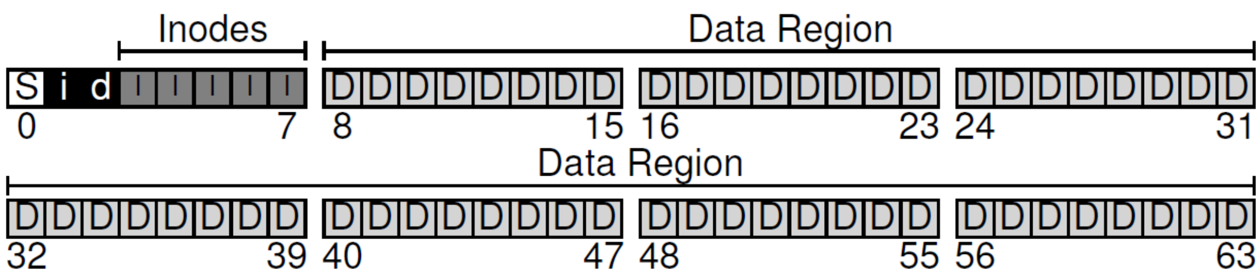
Keeping track of allocation

- We need a way to keep track of which blocks and which inodes are in use.
- We can have a free list.
- Here for simplicity we use a bitmap array. Each bit corresponds to a block/inode and a value of 0 or 1 indicates if it's free or not:



Keeping track of allocation … contd.

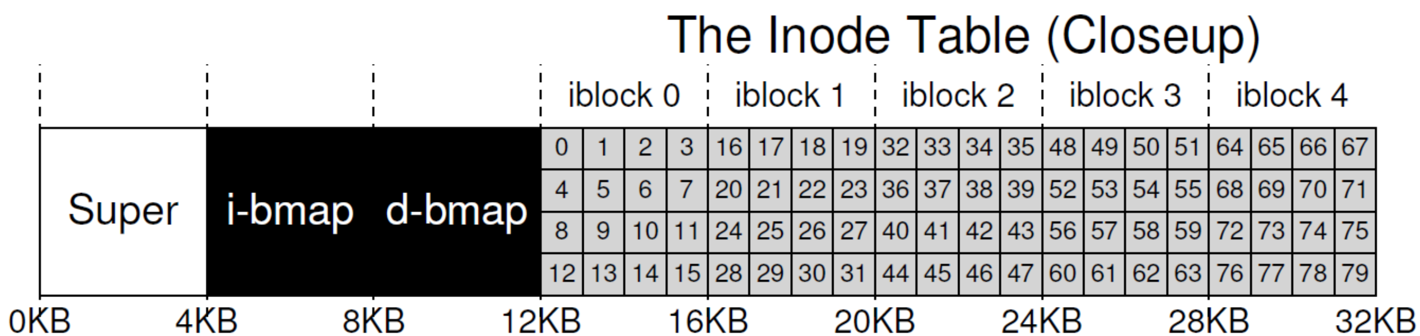
- A Super Block contains information about the file system:
  - Which type of file system being used
  - How many inode and data blocks
  - Where the inode table begins



- When **<u>mounting</u>** a file system, an OS first reads the Super Block to initialize its structures.

The Inode

- The most important data structure in a file system
- Each inode is associated a number which serves as index in inode table:



- Given the inode number, the start of inode table, we can calculate the address of that inode location on disk.

What’s in an inode?

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

Where’s the data?

- Each inode must contain info about where the file’s data is stored:
  - It can keep direct pointers
  - It can keep an indirect pointer
  - It can keep a double indirect pointer
  - … etc.
  - This scheme is refered to multi-level index.

Typical file system characteristics

- This tree of pointers is quite imbalanced.

Most files are small	˜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 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

- The data block of a directory would consist of *(string, number)* pairs:

(plus some info)

```
inum | reclen | strlen | name
  5      12        2     .
  2      12        3     ..
 12      12        4     foo
 13      12        4     bar
 24      36       28     foobar_is_a_pretty_longname
```

Directories are stored as files with a field in their inode indicating that it’s not a regular file.

Free space management

- In this case it’s simple: we are using bitmaps.

- Earlier systems kept a linked list of free blocks

- Many modern system use elaborate structures i.e. B-Trees.

How it works: reading a file

- We want to read /foo/bar (a 12KB file)

```
open("/foo/bar", O_RDONLY)
```

  - Traverse the path
    - Read / inode (we need its inode number)
    - Find its data block pointer and read inode of foo from within
    - Read foo inode
    - Find its data block pointer and read inode or bar from within
    - Read bar inode
    - Open would return a file descriptor associated with this inode

How it works: reading a file ... Contd.

`read()`ate the first data block of bar from its inode (unless lseek() called)
  - Read the block
    - My update the associated inode access time, file descriptor seek offset, etc.

- Close()
  - Deallocate the file descriptor, etc.

How it works: reading a file … Contd.2

	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					

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

How it works: writing a file

- Open() is same as read. (diff. permissions)

`write()`

  - Has to allocate a new block. Access the data and inode bitmaps:
    - Read data bitmap. Update it for the new block
    - Write data bitmap.
    - Read bar inode, update it with new data block ptr
    - Write data block
    - Write bar inode

- Create()
  - Has to update inode bitmaps as well:
    - Read inode bitmap table, update it
    - Write inode bitmap table
    - Write inode itself
    - Write to data block of directory to create the file entry there
    - Read, update and write directory inode entry (meta data)
    - Now imagine if the the directory data block was full and needed to expand?

I/O Requests

- Create causes

  10 I/Os

- Each write causes

  5 I/Os!

	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					

Performance

- How many I/Os for:

/1/2/3/ ... /100/file.txt

- Fixed cache: Some systems use a fixed size cache in memory to store popular blocks! i.e. 10% of memory
- Dynamic caching
- Write buffering:
  - Batches writes together to reduce number of I/Os
  - With multiple writes together, it gets the opportunity to schedule the I/Os
  - Unnecessary writes get avoided
  - Buffer writes from 15-30 seconds. Tradeoff??

Summary

- We saw how
  - OS implements file systems
  - directories are specific type of file with different inode structure storing name and an inode number mapping
  - File systems store bitmaps to track of free inodes or data blocks