Persistence: File System Implementations

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2024

Administrivia

- Project 5 due Nov 19 @ 11:59pm
- Discussion Session go over spec, `mmap`, VA to PA in xv6.

Review File System

- File System Abstractions – Files, Directories, Directory Tree
- Refer to a file: path (relative & absolute), inode number, file descriptor
- File IO Calls: open, read, write, lseek, fsync, (fd with fork and dup)
- Command line programs: stat, rm, ls, mkdir, mkfs, mount, strace
- Concepts: soft & hard links, permission bits and ACL, owner & group

Quiz 16 File API

https://tinyurl.com/cs537-fa24-q16

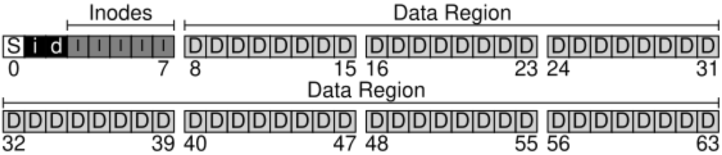
File System Implementation (Way to Think)

- Data Structures
  - What are the on-disk data structures to implement the file system?
- Access Methods
  - How does a call like `open()`, `read()`, or `write()` get mapped onto the data structures of the disk?

If you understand the data structures and access methods then you have a good mental model of the file system.

Overall Organization

A disk with 64 4-KB blocks:



Data Region (D) : Content of user's files and directories

Inodes (I) : A structure holding *metadata* for each file or directory

bitmap (d) : A bitmap of free/used data region blocks

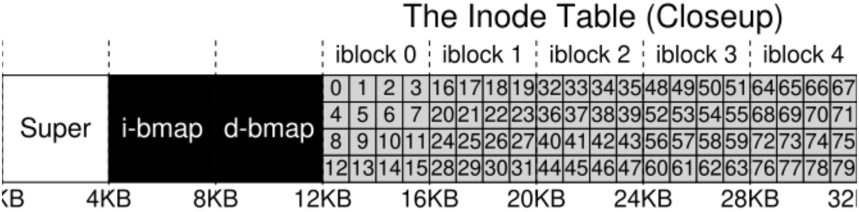
bitmap (i) : A bitmap of free/used inodes

Superblock (S) : The superblock contain information about the file system structure

Superblock and Bitmaps

The **superblock** contains information about the file system: - Number of inodes (80) and data blocks (56) - Where the inode table begins (block 3) - Magic Number indicating file system type

In **bitmaps**, each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1). - Bitmap for data blocks - Bitmap for inode table



Inodes

An **inode** contains the metadata for a file or directory:

- *type* – regular file, directory, etc.
- *size* – the number of bytes in the file
- *blocks* – number of blocks allocated to file
- *protection information* – Who owns the file and who can access it
- *time information* – last accessed time, creation time, last modified time
- *location information* – Where data blocks reside on disk

The Multi-Level Index

A **direct pointer** refers to one disk block that belongs to the file. Inodes often contain 12 direct pointers.

An **indirect pointer** refers to a block of pointers. If disk addresses are 4-bytes, a single 4KB block can hold 1024 pointers.

Max file size with 12 direct pointers and one indirect pointer is $(12 + 1024) \cdot 4K = 4144KB$.

For larger files, doubly or triply indirect pointers are used.

One finding of research on file systems is that *most files are small*.

Directory Organization

A directory has an inode with data blocks. The data blocks hold a list of (entry name, inode number) pairs.

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
0	*12*	*5*	*blah*
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

Deleting a file can leave an empty space in the middle of the directory, use inode number 0 to mark as empty.

Access Methods: Opening a File

Observe what happens when a file (e.g. `/foo/bar`) is opened, read, and then closed:

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

- Read root's inode
- Read root's data, scanning down the entries to find foo
- Read foo's inode
- Read foo's data, scanning down the entries to find bar
- Read bar's inode

Update an entry in the open file table and return the file descriptor.

Notice 5 I/O requests are needed to find bar's inode and "open" the file.

Access Methods: Reading a File

`count=read(fd,buf,4096)`

- Using the file's inode number and offset in open file table:
  - Read inode to find location of first block
  - Read data block
  - Write inode to update last access time
- Update the offset in open file table

For each block of file that is read, 3 I/O requests are performed.

Access Methods: Opening and Reading a File

	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)**

Access Methods: Writing to Disk

Writing is similar to reading:

- First, open the file
- Write changes to existing blocks
- Close file

Gets interesting when a new block must be **allocated**. This can occur with writing. Also occurs with create(). The bitmaps are consulted to find an unused entry.

	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					

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

Caching and Buffering

The file system aggressively caches important, frequently used blocks.

Read I/O can be avoided with a cache, but write traffic has to go to disk to become persistent.

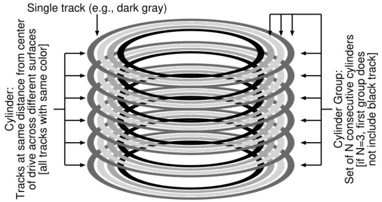
Write buffering has performance benefits: - Can **batch** some updates, reducing the number of I/O requests - Can use **scheduling** to optimize the ordering of the requests - Some I/Os can be **avoided** entirely, if a file is created and then deleted.

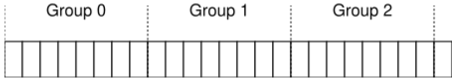
Modern FS buffer writes in memory anywhere from 5 to 30 seconds causing a trade-off between performance and data loss.

Can use `fsync()` to force writing to disk.

Fast File System Idea

- Organize file system structures and allocation policies to be **disk aware**
- Divided disk into collection of **cylinder groups**
- Modern file systems organize drive into similar **block groups** (consecutive portion of disk's address space)





- FFS includes all structures of a file system **within each group**

Per-Group Data Structures

- per-group **super-block** (needed to mount the file system, if one copy corrupt can us other copies)
- per-group **inode bitmap** and **data bitmap**
- per-group **data blocks**

- since all structures are per-group, they are close together on disk (less seek time)

Allocating Files and Directories

Keep related stuff together, keep unrelated stuff far apart.

Placement of Directories

- Find cylinder group with low number of allocated directories (to balance directories across groups) and high number of free inodes (to subsequently be able to allocate a bunch of files)
  - Put the directory data here
  - Put the directory inode here

Placement of Files

- Allocate a file's data blocks in same group as its inode
- Place all files in same directory in group with directory

Example Layout

Directories:

/

/a

/b

Files:

/a/c

/a/d

/a/e

/b/f

group	inodes	data
0	/---------	/---------
1	acde------	accddee---
2	bf--------	bff-------
3	----------	----------
4	----------	----------
5	----------	----------
6	----------	----------
7	----------	----------

...

Common Sense suggests files in a directory are often accessed together

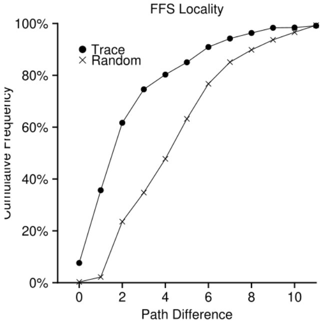
FFS will improve performance because (1) inode and data are together and (2) namespace-based locality

Measuring File Opening Locality

Analyzing the SEER workload trace of opening files:

Path Difference Metric measures how far up directory tree to find *common ancestor*:

- Same file – 0
- Another file, same directory – 1
- Another file, parent directory – 2
- Etc.



- Compared to randomly reordering file openings
- 7% were to same file
- 40% were to same directory

Large File Exception

A large file (e.g. 30 data blocks) would entirely fill most of the data blocks in a group, leaving little room for other files in the directory to be placed in the same group

```
group    inodes      data
0        /a--------  /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
1        ----------  ---------- ---------- ---------- ----------
2        ----------  ---------- ---------- ---------- ----------
...
```

The large file exception (here set to 5 blocks) spreads the file across groups:

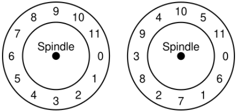
```
group    inodes      data
0        /a--------  /aaaaa---- ---------- ---------- ----------
1        ----------  aaaaa----- ---------- ---------- ----------
2        ----------  aaaaa----- ---------- ---------- ----------
3        ----------  aaaaa----- ---------- ---------- ----------
4        ----------  aaaaa----- ---------- ---------- ----------
5        ----------  aaaaa----- ---------- ---------- ----------
6        ----------  ---------- ---------- ---------- ----------
...
```

Large File Exception (cont.)

- Slows access to large files, but if chunk of a file in a group is large enough, this seeking will be **amortized**.
- FFS used 12 direct block pointers in inode (48KB) placed in group with inode
- Each indirect block pointer (4MB) pointed to block of pointers in different group, along with the data pointed to by those pointers.

Other FFS Innovations

- Introduction of **sub-blocks** (512-bytes) until file needs 4KB, then copy sub-blocks to a full block
  - Causes more I/O for each sub-block
  - Modified `libc` to buffer and do I/O in 4KB chunks
- Used skip-layout (called **parameterization**) so sequential I/O requests arrive before head rotates past them



  - Modern disks cache the entire track in an internal **track buffer**
- Added **long file names**
- Added **symbolic links**

Summary

- Metadata information is stored in a structure called an inode
- Directories are just specific type of file that store name -> inode-number mappings
- Bitmaps are used to record used/unusued information about the inode table and data blocks
- Understand for each I/O system call the series of I/O requests made to the file system
- FFS – divide the disk into **groups**, treat each group like a mini disk, large file exception.