10. File Systems

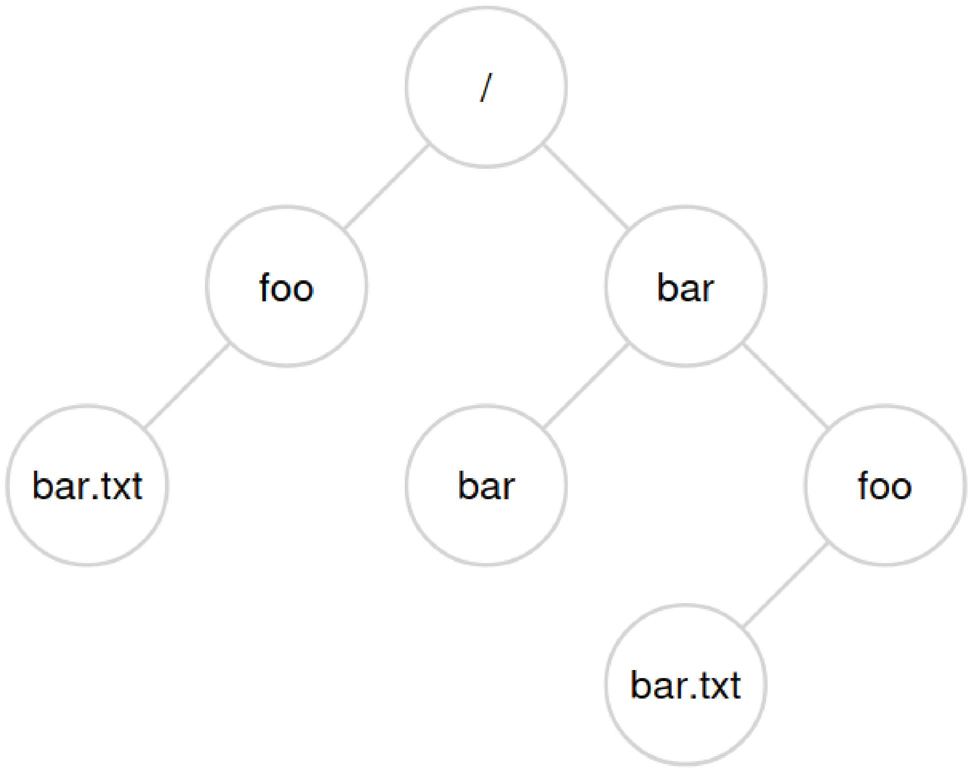
inode, open(), read(), write(), close(), STDIN/STDOUT/STDERR, file descriptor, fsync, metadata, strace, link/unlink, mkdir(), opendir(), readdir(), closedir(), rmdir(), hard link, symbolic link, permission bits (rwx), SetUID, SetGID, sticky bit, chmod(), chown(), mkfs, mount, inode/data bitmap, metadata, superblock, single/double/triple indirect pointers/addressing, extents, EXT, page cache, sleuthkit, fsck, journalling, idempotent

> Now we need one more virtualisation: **Persistent storage**

10.1 Files and Directories

Two key abstractions: **File** and **Directory**

file: a linear array of bytes, and it has some kind of *low-level*: **inode number**. **directory**: contains a list of (`user-readable-name`, `inode-number`) pairs. It also has an inode number.



In Unix-based systems, the **root-directory** starts at `/`

10.1.1 API

open:

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
S_IRUSR|S_IWUSR);
```

- `O_CREAT`: create the file if it doesn't exist
- `O_WRONLY`: open for writing
- `O_TRUNC`: truncate the file to zero length
- `S_IRUSR`: owner has read permission
- `S_IWUSR`: owner has write permission

10.1.2 File Descriptors

This returns a file descriptor `fd`:

- used to access the file
- *private* to the process
- stored in a per-process structure (proc)

There are three reserved file descriptors:

- `0`: standard input
- `1`: standard output
- `2`: standard error

cat:

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
hello
read(3, "", 4096) = 0
close(3) = 0
...
prompt>
```

10.1.3 Sync

A call to `write` doesn't actually write to the disk immediately. It writes to the **page cache**. To force the write to the disk, use `fsync`:

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

10.1.4 Metadata

We can use the `stat` tool for metadata

```
prompt> echo hello > file
prompt> stat file
File: ‘file’
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

To remove a file, use `unlink`:

```
prompt> strace rm foo
...
unlink("foo") = 0
...
```

10.1.5 Directories

We cannot write to a directory, because it's considered metadata.

```
prompt> strace mkdir foo
...
mkdir("foo", 0777) = 0
...
prompt>
```

10.1.6 Links

Hard Links

> A hard link is a directory entry that points to the same inode as another directory entry.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Symbolic Links

> A symbolic link is a file that contains the name of another file.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

10.1.7 Access Control

```
prompt> ls -l foo.txt
-rw-r--r-- 1 remzi wheel 0 Aug 24 16:29 foo.txt
```

`r`: read, `w`: write, `x`: execute

- first three: owner (`rw-`)
- second three: group (`r--`)
- third three: everyone else (`r--`)

SetUID: when a file is executed, it runs with the permissions of the file's owner.

SetGID: when a file is executed, it runs with the permissions of the file's group.

Stick bit: only the owner of the file can delete it.

10.1.8 `mkfs` and `mount`

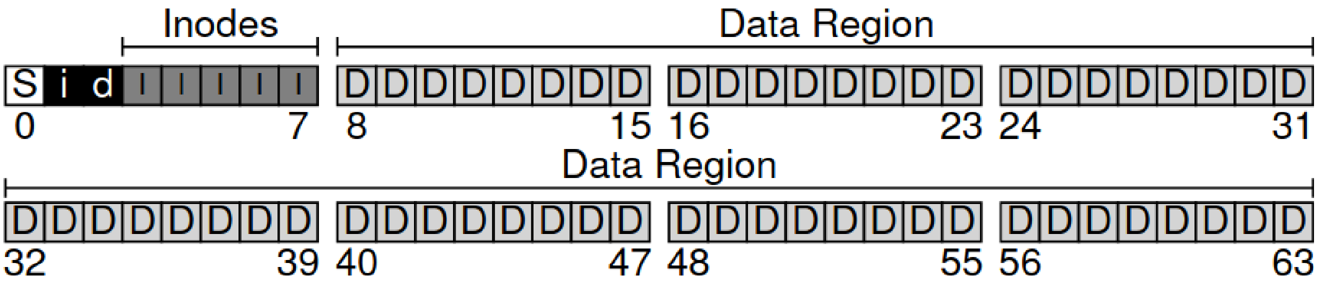
`mkfs`: create a file system on a disk `mount`: attach a file system to the directory tree

ASIDE: KEY FILE SYSTEM TERMS

- A **file** is an array of bytes which can be created, read, written, and deleted. It has a low-level name (i.e., a number) that refers to it uniquely. The low-level name is often called an **i-number**.
- A **directory** is a collection of tuples, each of which contains a human-readable name and low-level name to which it maps. Each entry refers either to another directory or to a file. Each directory also has a low-level name (i-number) itself. A directory always has two special entries: the `.` entry, which refers to itself, and the `..` entry, which refers to its parent.
- A **directory tree** or **directory hierarchy** organizes all files and directories into a large tree, starting at the **root**.
- To access a file, a process must use a system call (usually, `open()`) to request permission from the operating system. If permission is granted, the OS returns a **file descriptor**, which can then be used for read or write access, as permissions and intent allow.
- Each file descriptor is a private, per-process entity, which refers to an entry in the **open file table**. The entry therein tracks which file this access refers to, the **current offset** of the file (i.e., which part of the file the next read or write will access), and other relevant information.
- Calls to `read()` and `write()` naturally update the current offset; otherwise, processes can use `lseek()` to change its value, enabling random access to different parts of the file.
- To force updates to persistent media, a process must use `fsync()` or related calls. However, doing so correctly while maintaining high performance is challenging [P+14], so think carefully when doing so.
- To have multiple human-readable names in the file system refer to the same underlying file, use **hard links** or **symbolic links**. Each is useful in different circumstances, so consider their strengths and weaknesses before usage. And remember, deleting a file is just performing that one last `unlink()` of it from the directory hierarchy.
- Most file systems have mechanisms to enable and disable sharing. A rudimentary form of such controls are provided by **permissions bits**; more sophisticated **access control lists** allow for more precise control over exactly who can access and manipulate information.

10.2 Implementing a File System

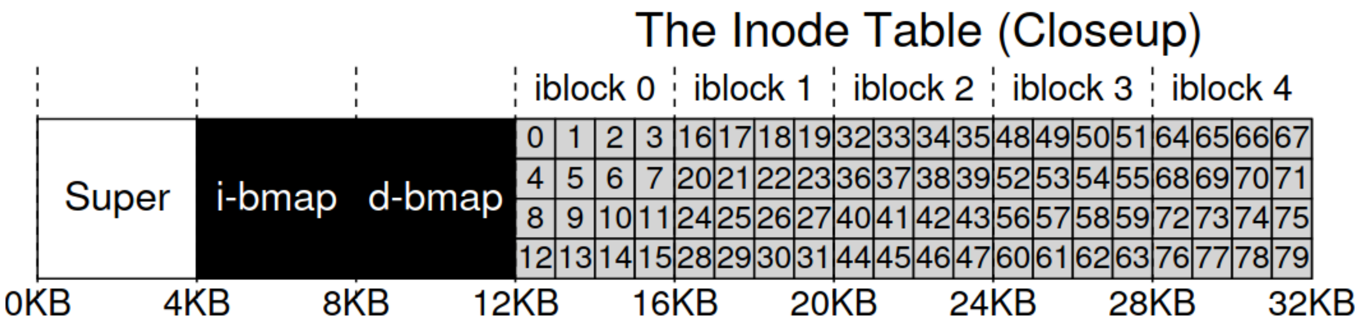
10.2.1 A file system



- **Data region**: user data stored here
- **Inode region**: metadata about files (**which data blocks comprise the file**, permissions, size, etc..)
- **Superblock**: contains metadata about the file system

10.2.2 Addresses

Inode table



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**

disk pointers: refer to blocks on disk belonging to the file **inode pointers**: refer to blocks in the inode table (to support bigger file sizes!)

^ Alternatively, we can use **extents**: a disk pointer + a length (in blocks)

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 contain 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

Figure 40.2: **File System Measurement Summary**

10.2.3 Directories

```
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
```

10.2.4 Access Path

Reading to 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)**

When the open sys call is made, the OS must **traverse** the pathname to find the file.

Writing to 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]
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)**

^ More complicated.

The cost of I/O is high, therefore we need to seek a more performant solution.

10.2.5 Caching

> Use system memory (DRAM) to cache important blocks

Early file systems introduced a **fixed-size cache** to hold popular blocks.

- The **Least Recently Used (LRU)** was used
- Would use 10% of memory
- *Static partioning*

But this is wasteful, and we can use **page cache** instead.

- *Dynamic partitioning*

10.3 Crash Management

> **crash-consistency problem** - the file system must be able to recover from a crash

If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent state**.

Solution #1: fsck

Steps:

- Are the inodes that are marked as used present in directories?
- Are the data blocks that are marked as used present in inodes?
- A bunch of small checks: e.g. are all values sensible (within range)?

Problem: ***They are too slow***

Solution #2: Journaling

> Before modifying the file system, write a log entry to a **journal** about what you're going to do.

Review Questions (10)

1. In an EXT-filesystem, how many inodes does a file have?
    - One
2. What is a file descriptor?
    - A handle to a file - private per process (integer)
3. Why can a file system that is NOT a journalling file system be damaged if the computer crashes?
    - It can be left in an inconsistent state
4. Describe some advantages and disadvantages of large and small block size i file systems.
    - Large:- Data can be located more efficiently, but more internal fragmentation
    - Small:- Inverse of the above
5. How will the performance be perceived if the operating system uses write-through caching when writing to a memory stick, compared to writing to the same memory stick without using cache? What about reading?
    - Writing will be affected (cuz more steps are required), reading will be faster
6. What is the maximum file size we can have in a file system based on inodes and double-indirect addressing when we assume 32-bit disk block addresses and disk block size of 8KB?

Block len: 8KB = 8192b = 2^13

Block addr: 32-bit = 4B = 2^2

addr/block = 2^13/2^2 = 2^11

total size = 2^11 * 2^11 * 2^13 = 2^35 = 32GB

7. Assume a file system that uses a bitmap to keep track of free/used disk blocks. The file system is located on a 4GB disk partition and uses a block size of 4KB. Calculate the size of the bitmap. -

Disk size = 4GB = 2^2* 2^30 = 2^{32} b

Block size = 4KB = 2^2* 2^10 = 2^{12} b

bitmap size = 2^{32} / 2^{12} = 2^{20}

bitmap size KB = 2^{20} / 2^{2} = 2^{17} B = 128KB

8. Explain in as much detail as you can what each command in the command line `ls -tr | tail -n 1 | xargs tail -n 2` does based on the following example:

- `ls -tr`: list files in reverse order of modification time
- `tail -n 1`: get the last line of the output
- `xargs tail -n 2`: get the last two lines of the file

9. `find . -name "*.pdf" | wc -l` to print every pdf file in your disk..