File and Directory in Practice

Operating Systems

Wenbo Shen

Acknowledgement: some pages are based on the slides from Yajin Zhou@ZJU

Two Key Abstractions

- File
  - A linear array of bytes, with each you can read/write
  - Has a low-level name (user does not know) - **inode number**
  - Usually OS does not know the exact type of the file
- Directory
  - Has a low-level name
  - Its content is quite specific: contains a list of user-readable name to low-level name. Like (“foo”, inode number 10)
  - Each entry either points to file or other directory

Two Key Abstractions

- File
  - External name (visible to user) must be symbolic
    - In hierarchical file system, unique external names are given as pathnames (path from root to the file)
  - Internal names (low-level names): i-node in UNIX
    - Stores information about file system object: file, directory, socket, …
    - Index into array of file descriptors/headers for a volume
- Directory: translation from external to internal name

File System Interface

- Create file

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

  - O_CREAT: create the file if not exists (not ***O_CREATE***)
  - O_WRONLY: can only write to the file
  - O_TRUNC: if the file exists, truncate it to zero bytes

The return value of **open**() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

File Descriptor

A *file descriptor* is an index in the per-process file descriptor table (in the left of the picture). Each file descriptor table entry contains a reference to a *file object*, stored in the file table (in the middle of the picture). Each file object contains a reference to an i-node, stored in the i-node table (in the right of the picture).

A file descriptor is just a number that is used to refer a file object from the user space. A file object represents an opened file. It contains things likes current read/write offset, non-blocking flag and another non-persistent state. An i-node represents a filesystem object. It contains things like file meta-information (e.g. owner and permissions) and references to data blocks.

File descriptors created by several `open()` calls for the same file path point to different file objects, but these file objects point to the same i-node. Duplicated file descriptors created by `dup2()` or `fork()` point to the same file object.

A BSD lock and an Open file description lock is associated with a file object, while a POSIX record lock is associated with an `[i-node, pid]` pair. We'll discuss it below.

One example

- Stdin, stdout, stderr

```
wenbo@parallels:~/os-course$ strace ./a.static
execve("./a.static", ["./a.static"], 0x7ffc89bb12f0 /* 68 vars */) = 0
brk(NULL)                                   = 0xc17000
brk(0xc181c0)                               = 0xc181c0
arch_prctl(ARCH_SET_FS, 0xc17880)           = 0
uname({sysname="Linux", nodename="parallels", ...}) = 0
readlink("/proc/self/exe", "/home/wenbo/os-course/a.static", 4096) = 30
brk(0xc391c0)                               = 0xc391c0
brk(0xc3a000)                               = 0xc3a000
access("/etc/ld.so.nohwcap", F_OK)          = -1 ENOENT (No such file or directory)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
write(1, "hello world!\n", 13hello world!
)           = 13
exit_group(0)                               = ?
+++ exited with 0 +++
```

The cat example

- strace cat main.c
- What is the file descriptor of main.c?

```
openat(AT_FDCWD, "main.c", O_RDONLY)    = 3
newfstatat(3, "", {st_mode=S_IFREG|0674, st_size=89, ...}, AT_EMPTY_PATH)
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
read(3, "#include <stdio.h>\n\nint main () "..., 131072) = 89
write(1, "#include <stdio.h>\n\nint main () "..., 89) = 89
read(3, "", 131072)                     = 0
munmap(0xffff969f7000, 139264)          = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

write and fsync

- write():
  - Write this data to persistent storage, at some point in the future
  - The file system, for performance reasons, will buffer such writes in memory for some time (say 5 seconds, or 30); at that later point in time, the write(s) will actually be issued to the storage device
- fsync(): forces all dirty (not yet written) data to disk

```
int fd = open ("foo", _CREAT | O_WRONLY | O_TRUNC);
assert (fd > -1);

int rc = write (fd, buffer, size);
assert (rc == size) ;

rc = fsync (fd) ;
assert (rc == 0) ;
```

Getting Information about Files

- Information is kept in **struct stat**

```c
struct stat {
        unsigned long   st_dev;         /* Device.  */
        unsigned long   st_ino;         /* File serial number.  */
        unsigned int    st_mode;        /* File mode.  */
        unsigned int    st_nlink;       /* Link count.  */
        unsigned int    st_uid;         /* User ID of the file's owner.  */
        unsigned int    st_gid;         /* Group ID of the file's group. */
        unsigned long   st_rdev;        /* Device number, if device.  */
        unsigned long   __pad1;
        long            st_size;        /* Size of file, in bytes.  */
        int             st_blksize;     /* Optimal block size for I/O.  */
        int             __pad2;
        long            st_blocks;      /* Number 512-byte blocks allocated. */
        long            st_atime;       /* Time of last access.  */
        unsigned long   st_atime_nsec;
        long            st_mtime;       /* Time of last modification.  */
        unsigned long   st_mtime_nsec;
        long            st_ctime;       /* Time of last status change.  */
        unsigned long   st_ctime_nsec;
        unsigned int    __unused4;
        unsigned int    __unused5;
};
```

```
os@os:~/temp$ stat foo
  File: 'foo'
  Size: 6               Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d      Inode: 1328649     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      os)   Gid: ( 1000/      os)
Access: 2018-12-19 00:24:02.448286431 +0800
Modify: 2018-12-19 00:24:01.316296543 +0800
Change: 2018-12-19 00:24:01.316296543 +0800
 Birth: -
```

Removing Files

- Why we just “remove” or “delete” the file, but using “unlinkat”?

```
parallels@ubuntu-linux-22-04-02-desktop:~/os-course$ strace rm tmp
execve("/usr/bin/rm", ["rm", "tmp"], 0xffffc5c6f3a8 /* 54 vars */) = 0
brk(NULL)                               = 0xaaaae84c7000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=63887, ...}, AT_EMPTY_PATH)
mmap(NULL, 63887, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffff9bb95000
close(3)                                = 0
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
newfstatat(AT_FDCWD, "tmp", {st_mode=S_IFREG|0664, st_size=89, ...}, AT_SYML
faccessat2(AT_FDCWD, "tmp", W_OK, AT_EACCESS) = 0
unlinkat(AT_FDCWD, "tmp", 0)            = 0
lseek(0, 0, SEEK_CUR)                   = -1 ESPIPE (Illegal seek)
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Hard link vs soft link

- Link
  - A file may be known by more than one name in one or more directories. Such multiple names are known as links.
  - The two kinds of links are also known as hard links and soft links.
- Hard link
  - A hard link is a directory entry that associates with a file
  - The file name "." in a directory is a hard link to the directory itself
  - The file name ".." is a hard link to the parent directory
- Soft link (a.k.a., Symbolic link or symlink)
  - A symbolic link is a file containing the path name of another file
  - Soft links, unlike hard links, may point to directories and may cross file-system boundaries

Hard link

- Directory is a file
- A hard link is a directory entry

```
struct ext2_dir_entry {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __le16  name_len;               /* Name length */
        char    name[];                 /* File name, up to EXT2_NAME_LEN */
};
```

Soft link

- A soft link is a file containing the path name of another file

Hard link

- link() system call: system call takes two arguments, an old pathname and a new one
  - Another way to refer to the same file
  - Same inode number

```
parallels@ubuntu:~/os-course$ echo hello > file1
parallels@ubuntu:~/os-course$ cat file1
hello
parallels@ubuntu:~/os-course$ ln file1 file2
parallels@ubuntu:~/os-course$ ls -l file*
-rw-rw-r-- 2 parallels parallels 6 Dec 17 21:02 file1
-rw-rw-r-- 2 parallels parallels 6 Dec 17 21:02 file2
parallels@ubuntu:~/os-course$ cat file2
hello
parallels@ubuntu:~/os-course$ ls -i file1 file2
3670573 file1  3670573 file2
```

Hard link

- link() system call: system call takes two arguments, an old pathname and a new one
  - Even an empty directory has two entries
    - Which two?

```
parallels@ubuntu:~/os-course$ mkdir dir1
parallels@ubuntu:~/os-course$ ls -al dir1
total 8
drwxrwxr-x  2 parallels parallels 4096 Dec 17 21:05 .
drwxrwxr-x 20 parallels parallels 4096 Dec 17 21:05 ..
parallels@ubuntu:~/os-course$ stat dir1
  File: dir1
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 802h/2050d      Inode: 3803181     Links: 2
Access: (0775/drwxrwxr-x)  Uid: ( 1000/parallels)   Gid: ( 1000/parallels)
Access: 2023-12-17 21:05:59.720000277 +0800
Modify: 2023-12-17 21:05:53.220000274 +0800
Change: 2023-12-17 21:05:53.220000274 +0800
 Birth: 2023-12-17 21:05:53.220000274 +0800
```

Why removing a file calls unlink

- Create a file
  - making a structure (the inode) that will track virtually all relevant information about the file
  - **linking** a human-readable name to that file (or inode), and putting that link into a directory
- To remove a file, we **unlink** it

```
unlink() deletes a name from the filesystem.  If that name was the last
link to a file and no processes have the file open, the file is deleted
and the space it was using is made available for reuse.
```

Reference count

- rm: unlink the file, and check the reference count of the **inode**

```
parallels@ubuntu:~/os-course$ ll file1
-rw-rw-r-- 1 parallels parallels 6 Dec 17 21:02 file1
parallels@ubuntu:~/os-course$ ln file1 file2
parallels@ubuntu:~/os-course$ ll file1
-rw-rw-r-- 2 parallels parallels 6 Dec 17 21:02 file1
parallels@ubuntu:~/os-course$ ln file1 file3
parallels@ubuntu:~/os-course$ ll file1
-rw-rw-r-- 3 parallels parallels 6 Dec 17 21:02 file1
parallels@ubuntu:~/os-course$ ll file3
-rw-rw-r-- 3 parallels parallels 6 Dec 17 21:02 file3
parallels@ubuntu:~/os-course$ rm file3
parallels@ubuntu:~/os-course$ ll file1
-rw-rw-r-- 2 parallels parallels 6 Dec 17 21:02 file1
parallels@ubuntu:~/os-course$ rm file2
parallels@ubuntu:~/os-course$ ll file1
-rw-rw-r-- 1 parallels parallels 6 Dec 17 21:02 file1
```

Soft links

- Hard links do not increase
- Different inode
- rm the target file, the link will be invalided

```
parallels@ubuntu:~/os-course$ cat file1
hello
parallels@ubuntu:~/os-course$ ln -s file1 file2
parallels@ubuntu:~/os-course$ ll file*
-rw-rw-r-- 1 parallels parallels 6 Dec 17 21:02 file1
lrwxrwxrwx 1 parallels parallels 5 Dec 17 21:14 file2 -> file1
parallels@ubuntu:~/os-course$ ls -i file*
3670573 file1  3670575 file2
parallels@ubuntu:~/os-course$ rm file1
parallels@ubuntu:~/os-course$ ll file*
lrwxrwxrwx 1 parallels parallels 5 Dec 17 21:14 file2 -> file1
```

Two different aspects of FS

- VFS data structure
  - inode
- In-memory data structure
  - ext2_inode_info
  - Contain both ext2_inode and inode
- On-disk data structure
  - ext2_inode

```
struct inode {
        umode_t                 i_mode;
        unsigned short          i_opflags;
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;

#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif

        const struct inode_operations   *i_op;
        struct super_block      *i_sb;
        struct address_space    *i_mapping;

#ifdef CONFIG_SECURITY
        void                    *i_security;
#endif
```

```
struct ext2_inode_info {
        __le32  i_data[15];
        __u32   i_flags;
        __u32   i_faddr;
        __u8    i_frag_no;
        __u8    i_frag_size;
        __u16   i_state;
        __u32   i_file_acl;
        __u32   i_dir_acl;
        __u32   i_dtime;

        struct mutex truncate_mutex;
        struct inode    vfs_inode;
        struct list_head i_orphan;      /* unlinked but op
#ifdef CONFIG_QUOTA
        struct dquot *i_dquot[MAXQUOTAS];
#endif
};
```

Two different aspects of FS

- (on-disk) Structure about FS
  - vs in-memory structure about FS
- Access methods
  - how does the FS maps the calls (open/read/write) onto its structures

An Example of FS Organization

- Suppose we have a serial of blocks
  - Block size: 4KB
  - 64 blocks

Data Region

- We reserve some blocks for data
  - 56 of 64 blocks

Inodes

- Inode table: contains inodes
- 5 of 64 blocks are reserved for inodes
  - Suppose inodes are 256 bytes, 4 KB block can hold 16 inodes, then 5 blocks -> 80 inodes -> 80 files (directories)

D: data block
I: inode table

Bitmap

- Suppose we use bitmap to manage the free space
  - One bitmap for free inodes
  - One bitmap for free data region

D: Data block
I: inode
i: inode bitmap
d: data region bitmap

Superblock

- Superblock
  - Contains information about this file system: how many inodes/data blocks, where the inode table begins, where the data region begins, and **a magic number**

D: Data block
I: inode
i: inode bitmap
d: data region bitmap
S: superblock

Inode address calculation

- Suppose inodes are 256 bytes, 4 KB block can hold 16 inodes, then 5 blocks -> 80 inodes -> 80 files (directories)
- Each inode is identified by a number(inumber)
- To read inode number No. 32
  - 32 * sizeof(inode) = 8KB
  - Address: 4k(super block) + 8k(BITMAP) + 8k = 20KB

Linux ext2_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

Figure 40.1: **Simplified Ext2 Inode**

Multi-Level Index in inode

- To support bigger file, we need multi-level index for the nodes

Directory Organization

- Suppose a dir (inode number 5) has 3 files: foo, bar, foobar
  - inode number, name;
  - strlen: length of the name
  - reclen: length of the name plus **left over space**
    - For reuse the entry purpose

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

```
struct ext2_dir_entry {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __le16  name_len;               /* Name length */
        char    name[];                 /* File name, up to EXT2_NAME_LEN */
};
```

Directory Implementation

- Directory is a special file
  - Store the mapping from file name to its inode

Read /foo/bar

- Suppose foo is a directory, bar is a file in it
- / + foo/ + bar

	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 dir entry of foo				
				read						
							read ← Read dir entry of bar			
					read					
read()					**read**					
								read		
					write	← Update access time				
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

Write to Disk: /foo/bar

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
			read							
						read				
				read						
							read	← Find bar not exist		
create (/foo/bar)		**read**								
		write								
							write	← Write dir entry of bar to foo		
					write					
				write	← Update metadata, such as count, access time					
					read					
write()	**read**									
	write									
								write		
					write					
					read					
write()	**read**									
	write									
									write	
					write					
					read					
write()	**read**									
	write									
										write
					write					

Caching and Buffering

- Without caching, each file open would require two reads for each level of the directory
  - One for the inode, and one for data
- Early system allocate a **fixed-size** cache to hold popular blocks
- Morden systems use a **unified page cache** for both virtual memory pages and file system pages
- Write buffering: does not write to disk immediately, instead sync to disk for like 5 - 30 seconds
- Database: direct IO with raw data

Takeaway

- File interfaces
  - External name
  - Internal name (low-level name): inode
- Directory
  - Translate from external name to internal name
- Hard link vs soft link
- On-disk layout of FS