Chapter 39: Interlude: Files and Directories

A persistent-storage device, such as a classic **hard disk drive** or a more modern **solid-state storage device**, stores information permanently. Unlike memory, whose contents are lost when there is a power loss, a persistent-storage device keeps such data intact.

**39.1 Files And Directories**

A **file** is simply a linear array of bytes, each of which you can read or write. Each file has a **low-level name**, usually a number. For historical reasons, the low-level name of a file is often referred to as its **inode number**.

In most systems, the OS does not know much about the structure of the file (e.g., whether it is a picture, or a text file, or C code). The responsibility of the file system is simply to store data persistently on disk and make sure it is the same.

A **directory**, like a file, also has a low-level name or **an inode number**, but its contents are quite specific as it contains a list of (user-readable name, low-level name) pairs. Each entry in a directory refers to either files or other directories. By placing directories within other directories, users can build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

The directory hierarchy starts at a **root directory** (referred to as /) and uses some kind of **separator** to name subsequent sub-directories until the desired file or directory is named (e.g. /foo/bar.txt is an **absolute pathname**). The part after the period sign in file name is usually the **type** of the file (txt, c, jpg). However, this is usually just a **convention**: there is usually no enforcement that the data contained in a file named main.c is indeed C source code.

Names are important in systems as the first step to accessing any resource is being able to name it.

**39.2 The File System Interface**

We will discover the mysterious call known as unlink().

**39.3 Creating Files**

To create a file, we simply use the open system call with O\_CREAT flag. For example,



There are other flags as well. O\_WRONLY as ensures that the file can only be written to and O\_TRUNC is to truncates it to a size of zero bytes (removing existing content).

Open() returns a **file descriptor**. A file descriptor is just an integer, private per process, and is used in UNIX systems to access files.

Once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. In this way, a file descriptor is a **capability** or the ability to perform certain operations.

file descriptors are managed by the operating system on a per-process basis. This means simple structure (e.g., an array) is kept in the proc structure on UNIX systems:

A picture containing text

Description automatically generated

A simple array (with a maximum of NOFILE open files) tracks which files are opened on a per-process basis. Each entry of the array is actually just a pointer to a struct file.

**39.4 Reading And Writing Files**

We can use **cat** to read a file. To trace its behavior, we use a linux tool called strace

Text

Description automatically generated

The first thing cat does is to open the file for reading. Notice that the file is only opened for reading. The second flag indicates that the 64-bit offset be used. The open call returns a file descriptor.

Each running process already has three files open, standard input and standard error and they are represented by 0, 1 and 2. Thus, when we open another file, it is likely to be 3. Then, the cat uses the read() system call to read some bytes from a file. The read system call takes the file descriptor, the buffer where the result of read() will be placed and the size of the buffer. The call to read() returns successfully as well, here returning the number of bytes it read.

As we mentioned above, this descriptor 1 is known as the standard output, and thus is used to write the word “hello” to the screen as the program cat is meant to do.

The cat program then tries to read more from the file, but since there are no bytes left in the file, the read() returns 0 and the program knows that this means it has read the entire file. Thus, the program calls close() to indicate that it is done with the file “foo”, passing in the corresponding file descriptor. The file is thus closed, and the reading of it thus complete.

The write() system call is similar.

**39.5 Reading And Writing, But Not Sequentially**

The above read and write is for sequential execution. How can we deal with **random** offsets within the document. TO do so, we will use the lseek() system call.



The first argument is the file descriptor. The second one is the positions the **file offset** to a particular location within the file. The third one determines how the seek is performed.

Text

Description automatically generated

The offset is kept in that struct file we saw earlier, as referenced from the struct proc.

Text, letter

Description automatically generated

In this structure, the OS can use this to determine whether the opened file is readable or writable.

These file structures represent all of the currently opened files in the system. Together, they are sometimes referred to as the **open file table**. The xv6 kernel just keeps these as an array as well, with one lock per entry, as shown here:

Text

Description automatically generated

Trace example:

Text

Description automatically generated

In the last read, the return code is 0 because the process has read the file entirely. A process can also open the same file twice:

Table

Description automatically generated

Two file descriptors will be allocated and each refers to a different entry in the open file table.

If we use lseek() to reposition the current offset before reading:

Text

Description automatically generated

**39.6 Shared File Table Entries: fork() And dup()**

There are a few interesting cases where an entry in the open file table is **shared**. One of those cases occurs when a parent process creates a child process with fork().

Text

Description automatically generated

As we can see, as the child change the offset via lseek(), the offset of the parent also change.

Diagram, engineering drawing

Description automatically generated

The above figure shows the relationships that connect each process’s private descriptor array, the shared open file table entry, and the reference from it to the underlying file-system inode. Note that we finally make use of the reference count here. When a file table entry is shared, its reference count is incremented. Only when both processes close the file (or exit) will the entry be removed.

Other cases occur with the system call dup() (also dup2() and dup3()). The dup() call allows a process to create a new file descriptor that refers to the same underlying open file as an existing descriptor. For example:

Text

Description automatically generated

Dup() call is useful when writing a UNIX shell and performing operations like output redirection.

**39.7 Writing Immediately With fsync()**

Some applications require something more than eventual guarantee. For example, in a database management system (DBMS), development of a correct recovery protocol requires the ability to force writes to disk from time to time.

To do this, we need fsync(int fd). When a process calls fsync() for a particular file descriptor, the file system responds by forcing all **dirty** (i.e., not yet written) data to disk, for the file referred to by the specified file descriptor. The fsync() routine returns once all of these writes are complete.

Text, letter

Description automatically generated

The code opens the file foo, writes a single chunk of data to it, and then calls fsync() to ensure the writes are forced immediately to disk.

**39.8 Renaming Files**

The command mv can be used to rename a file:



With strace, we know that mv uses the system call rename, which is an **atomic** call with respect to system crashes, i.e. if the system crashes during the renaming, the file will either be named the old name or the new name, and no odd in-between state can arise.

**39.9 Getting Information About Files**

Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing. We generally call such data about files **metadata**. To view it, we can use stat() or fstat(). These calls take a pathname or file descriptor to a file and fill in the following structure:

Text, letter

Description automatically generated

When we call these system calls, the output is shown as follows:

Text, letter

Description automatically generated

Each file system usually keeps this type of information in a structure called an **inode.**

**39.10 Removing Files**

To do this, we use command rm. The interesting thing is that rm uses a system call called unlink(). To understand this, we need to understand directory.

**39.11 Making Directories**

To create a directory, we use mkdir().

When such a directory is created, it is considered “empty”, although it does have a bare minimum of contents. Specifically, an empty directory has two entries: one entry that refers to itself, and one entry that refers to its parent. The former is referred to as “.” and the latter is “..”

**39.12 Reading Directories**

We can do this using ls. However, we can write our own program to iterate over every directory entry and print out the name and the inode number:

Text, letter

Description automatically generated

The dirent struct is shown as follows:

Graphical user interface, text

Description automatically generated

**39.13 Deleting Directories**

We can delete it with rmdir()

**39.14 Hard Links**

The link() system call takes two arguments, an old pathname and a new one. When we call link (ln) a new file to an older file, we create another way to refer to the same file:

Text

Description automatically generated

The way link() works is that it simply creates another name in the directory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file. The file is not copied in any way. Rather, you now just have two human-readable names (file and file2) that both refer to the same file. We can even see this in the directory itself, by printing out the inode number of each file:

Text

Description automatically generated with medium confidence

When we create a file, we are really doing two things. First, we are making a structure (the inode) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth. Second, we are linking a human-readable name to that file, and putting that link into a directory.

To remove a file from the file system, we call unlink(). In the example above, we could for example remove the file named file, and still access the file without difficulty:

A picture containing text, orange

Description automatically generated

The reason this works is because when the file system unlinks file, it checks a reference count within the inode number. This reference count (sometimes called the link count) allows the file system to track how many different file names have been linked to this particular inode. When unlink() is called, it removes the “link” between the human-readable name (the file that is being deleted) to the given inode number, and decrements the reference count; only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly “delete” the file.

**39.15 Symbolic Links**

There is one other type of link that is really useful, and it is called a **symbolic link** or sometimes a **soft link**. Hard links are somewhat limited: you can’t create one to a directory (for fear that you will create a cycle in the directory tree); you can’t hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc.

To create such a link, you can use the same program ln, but with the -s flag. For example:

Text

Description automatically generated

The differences are:

1. A symbolic link is actually a file itself, of a different type. When we call stat, it will show this:

Text

Description automatically generated

1. Because of the way symbolic links are created, they leave the possibility for what is known as a **dangling reference**, i.e. removing the original file cause the link point to a pathname no longer exists.

**39.16 Permission Bits And Access Control Lists**

The file system also presents a virtual view of a disk, transforming it from a bunch of raw blocks into much more user-friendly files and directories, as described within this chapter. However, we need a more comprehensive set of mechanisms for enabling various degrees of sharing are usually present within file systems.

The first form of such mechanisms is the classic UNIX **permission bits**. To see permissions for a file foo.txt, just type:

****

The first “-” indicates that this is a regular file. The next characters, rw-r--r--, s determine exactly who can access it and how. The permissions consist of three groupings: what the owner of the file can do to it, what someone in a group can do to the file, and finally, what anyone (sometimes referred to as other) can do. The first three characters of the output of ls show that the file is both readable and writable by the owner (rw-), and only readable by members of the group wheel and also by anyone else in the system (r-- followed by r--).

We can change these permissions using chmod command to change the **file mode**. To remove the ability for anyone except the owner to access the file, you could type:



This command enables the readable bit (4) and writable bit (2) for the owner (OR’ing them together yields the 6 above), but set the group and other permission bits to 0 and 0, respectively, thus setting the permissions to rw-------.

For directories, the execute bit behaves a bit differently. Specifically, it enables a user (or group, or everyone) to do things like change directories (i.e., cd) into the given directory, and, in combination with the writable bit, create files therein.

Beyond permission bits, we also have **access control list (ACL)** per directory. Access control lists are a more general and powerful way to represent exactly who can access a given resource. In a file system, this enables a user to create a very specific list of who can and cannot read a set of files, in contrast to the somewhat limited owner/group/everyone model of permissions bits described above.

**39.17 Making And Mounting A File System**

how to assemble a full directory tree from many underlying file systems? This task is accomplished via first making file systems, and then mounting them to make their contents accessible.

To make a file system, most file systems provide a tool, usually referred to as mkfs (make fs).

The idea is as follows: give the tool, as input, a device and a file system type, and it simply writes an empty file system, starting with a root directory, onto that disk partition.

However, once such a file system is created, it needs to be made accessible within the uniform file-system tree. This task is achieved via the mount program.

What mount does is take an existing directory as a target mount point and essentially paste a new file system onto the directory tree at that point. For example:



This would move all files and directories hierarchy of /dev/sda1/ to /home/users/.