【OS】Day43(2)

Class	Operating System: Three Easy Pieces
Date	@February 27, 2022

【Ch39】Interlude: Files and Directories

39.1 Files and Directories

Two key abstractions have developed over time in the virtualization of storage. The first is file. A file is simply a linear array of bytes, each of which we can read or write.

Each file has some kind of low-level name, usually a number of some kind. 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 such data persistently on disk and make sure that when we request the data again, we get what we put there in the first place.

The second abstraction is that of a directory. A directory also has a low-level name(i.e. an inode number), but its contents are quite specific: 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 are able to build an arbitrary directory tree(or directory hierarchy), under which all files and directories are stored.

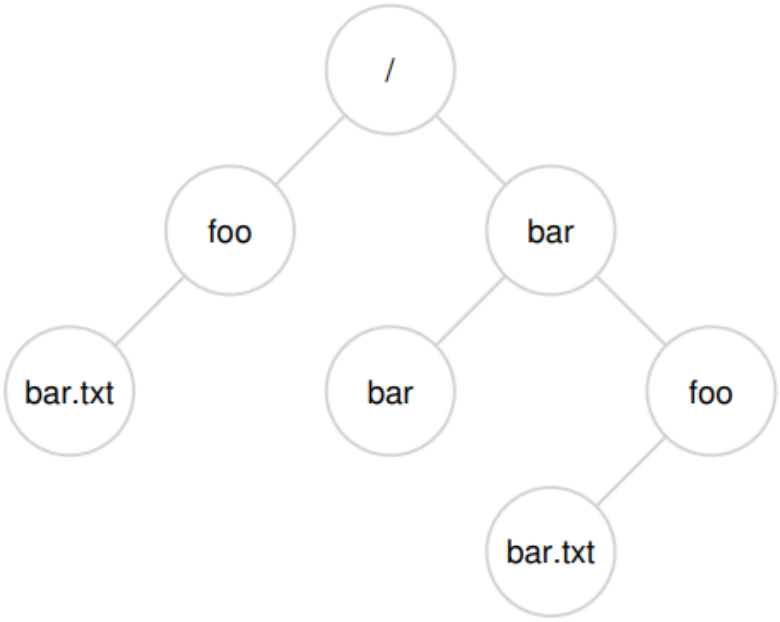

Figure 39.1: **An Example Directory Tree**

The directory hierarchy starts at a root directory(in UNIX-based systems, the root directory is imply referred to as `/`) and uses some kind of separator to name subsequent sub-directories until the desired file or directory is named.

For example, if a user created a directory foo in the root directory `/`, and then created a file `bar.txt` in the directory foo, we could refer to the file by its absolute pathname.

Directories and files can have the same name as long as they are in different locations in the file-system tree.

We may also notice that the file name in this example often has two parts: `bar` and `txt`, separated by a period. The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the type of the file(e.g. whether it is C code or an image or a music file)

39.3 Creating Files

We'll start with creating a file. This can be accomplished with the open system call; by calling `open()` and passing it the `O_CREAT` flag, a program can create a new file.

Here is some example code to create a file called "foo" in the current working directory:

```
int fd = open("foo", OCREAT|O_WRONLY|O_TRUNC, S_ITUSR|S_IWUSR);
```

`open()` takes a number of different flags. In this example, the second parameter creates the file(`O_CREAT`) if it does not exist, ensures that the file can only be written to(`O_WRONLY`), and if it already exists, truncates it to a size of zero bytes thus removing any existing content(`O_TRUNC`).

The third parameter specifies permissions, making the file readable and writable by the owner.

One important aspect of `open()` is what it returns: a file descriptor. A file descriptor is just an integer, private per process, and is used in UNIX system to access files. Thus, once a file is opened, we use the file descriptor to read or write the file, assuming we have permission to do so. In this way, a file descriptor is a capability.

Another way to think of a file descriptor is a pointer to an object of type file; once we have such an object, we can all other "methods" to access the file, like `read()` and `write()`.

As stated above, file descriptors are managed by the operating system on a per-process basis. This means some kind of simple structure is kept in the proc structure on UNIX systems.

```
struct proc {
  struct file *ofile[NOFILE]; // Open files
};
```

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, which will be used to track information about the file being read or written.

39.4 Reading and Writing Files

Let's start by reading an existing file. If we were typing at a command line, we might just use the program `cat` to dump the contents of the file to the screen.

```
prompt> echo hello > foo
prompt> cat foo
hello
```

In this code snippet, we redirect the output of the program echo to the file foo, which then contains the work "hello" in it. We then use cat to see the contents of the file.

But how does the cat program access the file foo?

We'll use an incredibly useful tool to trace the system calls made by a program. On Linux, the tool is called `strace`. What `strace` does is trace every system call made by a program while it runs and dump the trace to the screen for us to see.

Here is an example of using `strace` to figure out what cat is doing:

```
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)
...
```

The first thing that cat does is open the file for reading.

1. First, the file is only opened for reading(not writing), as indicated by the `O_RDONLY` flag.
2. Second, the 64-bit offset by used(`O_LARGEFILE`)
3. Third, the call to `open()` succeeds and returns a file descriptor, which ahs the value of 3.

Why does the first call to `open()` returns 3?

Each running process already has three files open, standard input(which the process can read to receive input), standard output(which the process can write to in order to

dump information to the screen), and standard error(which the process can write error message to). These are represented by file descriptors 0, 1, and 2 respectively.

After the open succeeds, `cat` uses the `read()` system call to repeatedly read some bytes from a file.

1. The first argument to `read()` is the file descriptor, thus telling the file system which file to read. The file descriptor enables the operating system to know which file a particular read refers to.
2. The second argument points to a buffer where the result of the `read()` will be placed. In the system-call trace above, `strace` shows the result of the read in this spot(”hello”).
3. The third argument is the size of the buffer, which in this case is 4KB.
4. The call to `read()` returns successfully as well, here returning the number of bytes it reads(the size of the buffer)

At this point, we see another interesting result of the `strace`: a single call to the `write()` system call, to the file descriptor 1. This descriptor 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.

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.