CS392: Systems Programming Notes

Steven DeFalco

Spring 2023

Contents

1 Introduction to Linux

2 Bash Scripts

3 C Programming Language

4 File Systems and File I/O

4.1 File

4.2 UNIX File System

4.3 File Related Structures and Operations

4.4 Directories Related Structures and Operations

4.5 File Descriptors

File descriptors are non-negative intergers that are assigned to keep track of every file that is currently opened by a process. Each process maintains a table of file descriptors; think of this table as an array, where the indices are file descirptors and each element of the array is an object of the `fd` struct:

```
struct fd {
    struct file* file;
    unsigned int flags;
};
```

There are a lot of fields defined in `struct file`, but the most relevant ones are shown below:

```
struct file {
    ...
    struct inode* f_inode;
    unsigned int f_flags;
    loff_t f_pos;
    ...
}
```

The `f_flags` are the file flags, such as `O_RDONLY`, `O_NONBLOCK`, `O_SYNC`. The `f_pos` indicates the current reading or writing position (offset). Its type, `loff_t`, is a 64-bit value. You can list all the files in `/dev/` to see all the device files.

4.6 I/O System Calls

All ways to open/write/read/close a file are just wrappers that eventually call the lowest level system functions. These functions deal with file descriptors directly. The following are some of these system functions:

- `open()` and `close()`: to open or close a file

- creat(): to create a file
- read() and write(): to read or write a file
- lseek(): to seek a position in a file

4.6.1 Opening, Closing, and Creating Files

The prototype of open() is as follows:

```
#include <fcntl.h>
int open(const char* pathname, int flags);
int open(const char* pathname, int flags, mode_t mode);
```

which returns a file descriptor. This function is ued to open a file, regardless of its type: regular, directory, block, character, or socket.

The flags specifies the mode of opening, and it has to have one of the following macros:

- O_RDONLY: read only
- O_WRONLY: write only
- O_RDWR: read and write

There are more macros as well. For example, if O_CREAT is specified, the function will create a new file if the pathname doesn't exist. If O_APPEND is used, the function will append content to the end of the file. To combine some of these macros, we can use the *or* operator like this:

```
int fd = open(”test”, O_WRONLY | O_CREAT | O_APPEND);
```

Creating a file is also similar

```
#include <fcntl.h>
int creat(const char* pathname, mode_t mode);
```

where mode is the same as above. The following two statements are equivalent and represent how we can combine macros to customize the open() system call.

```
int fd = open(”test”, O_WRONLY | O_CREAT | O_TRUNC |
                      O_APPEND);
int fd = creat(”test”, S_IRWXU);
```

because `O_TRUNC` flag will remove everything in the file `test` if it exists already.

In addition to those mentioned, we can also open a file using the following functions

```
int openat(int dirfd, const char* pathname, int flags);
int openat(int dirfd, const char* pathname,
           int flags, mode_t mode);
```

where `dirfd` is a file descriptor of a directory, and pathname is the **relative path** under that directory. For example, to open a file at the absolute path of
`usr`
`sh`
`myfile`, you can use `open()`:

```
int fd = open(”/usr/sh/myfile”, O_RDWR);
```

but you can also open a directory first, and use `openat()`:

```
int dirfd = open(”/usr/sh/”, O_RDWR);
int fd = openat(dirfd, ”myfile”, O_RDWR);
```

To **close** a file, you simply need the file descriptor:

```
int close(int fd);
```

4.6.2 Reading and Writing a File

The functions to read and write files are as follows:

```
#include<unistd.h>
ssize_t read(int fd, void* buf, size_t count);
ssize_t write(int fd, const void* buf, size_t count);
```

5 Processes

6 Inter-Process Communication (IPC)

7 Threads