CS392: Systems Programming Notes

Steven DeFalco

Spring 2023

Contents

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

```
1  int openat(int dirfd, const char* pathname, int flags);
2  int openat(int dirfd, const char* pathname,
3             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()`:

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

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

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

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

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

4.6.2   Reading and Writing a File

The functions to read and write files are as follows:

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

The function `read()` attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`. THe other direction applies to `write()`. The functions return the number of bytes read or written if scucessful, 0 if it reaches the end of the file, and -1 if there is an error. It is good practice to always check the retrun value of `write()` to make sure it matches the expected number. See the following for an appropriate implementation.

```
1  char* tr = "Hello!";
2  if (write(fd, str, 6) != 6) {
3      fprintf(stderr, "File failed to write.\n");
4  }
```

4.6.3 File Reposition

For each file opened, the kernel keeps a record for current file offsets, i.e. which byte of the file we're currently at. THe `lseek()` function can b used to chnage the current location **within** a file.

```
off_t lssek(int fd, off_t offset, int whence);
```

where **off_t** is offset in bytes and `whence` can be one of three places (declared as macros):

- ○ `SEEK_SET`: relocate to the beginning of the file (offset of zero) + `offset`. Essentially just `offset`'

- ○ `SEEK_END`: relocate to the end of the file (or, the size of the file) + `offset`;

- ○ `SEEK_CUR`: relocate to its current location within the file + `offset`

File holes can be created when the content of the file is no longer contiguous due to changing the file pointer with an `lseek()` command.

4.6.4 Buffering

Since everything including hardware in UNIX is considered a file, to utilize those hardwares, we have to connect them to the file systems. SOme devices need immediate access without delay, such as RAM, keyboard, monitor, etc. These devices are called **character special devices**. Some other devices, however, woudl be better to "cache" some data first in order to avoid longer access time, and these are called **block special devices** such as the hard drive.

Recall that getting access to a file on a hard drive is very slow. Therefore, we use the idea of "cache" and create a "buffer cache" (aka "buffer") between the hard drive and the file system. Whenever we `open()` a file, the data contained in this file will be copied into the cache. This "buffer cache" is in the main memory and is a structure maintained by the kernel. During the booting of the system, the kernel will allocate a couple of buffers in a certain location of the memory. Each of the buffers includes a data area, and a buffer header to identify this buffer.

4.6.5 Streams

Whenever we want to operate on a file, we create a channel to connect our program and the file. On the low level, we know that the channel is a file descriptor created by calling `open()`. On a higher (user) level, the channel we use is an object of `FILE*`, declared in the standard C library `<stdio.h>`. This connection is called a **stream**.

Recall that to open a file using `<stdio.h>` we use `fopen()` with `FILE*`. This function does the system call `open()` for us, and associates this stream with the file descriptor. You can get the file descriptor number of a stream using the `fileno()` function. If we want to open a file using an existing file descriptor, we can use the `fdopen()` function, which will create a `FILE` struct and associate it with the file descriptor.

4.6.6   Stream Buffering

In user space, we also need buffered stream between our program and system call. This is because we want to avoid using system calls too often, so we buffer some data from our program, and only when the buffer is full do we use system call. For example: Say we want to use `fprintf()` to write a string to a file. When our program reaches the `fprintf()` line, from our perspective the operation of writing to the file is done, i.e., the string is already written to the file. This is not necessarily true, however, because this only means it's written to the **stdio buffer**, not the actual file.

C standard I/O library `stdio` provides three buffering modes, fully buffered, line buffered, and unbuffered.

4.6.6.1   Fully Buffered

This is typically used with files, such as reading from and writing to a file. The actual I/O happens in two situations:

- ○ Reading: buffer is emtpty and needs to be filled;

- ○ Writing: buffer is full and needs to be emptied;

When the stream buffer is full, the content inside the buffer will be actually written to the disk or the hard drive. This process is called **flushing**. Flushing can be automatic (such as when the buffer is full or we closed the file), but can also be manual by using `fflush()` function:

```
1  int fflush(FILE* stream;
```

4.6.6.2   Line Buffered

Line buffered will flush everything from buffer to the destination when a line is finished, i.e., when a newline character '\n' is entered. Line buffered is still buffered, and therefore, the buffer has a limit. If the buffer is full and there's still no newline character entered, everything has to be flushed so the final data might be shorter than you actually input.

4.6.6.3 Unbuffered

Unbuffered means I/O takes place immediately. One example is `stderr`, which is never buffered. This means whenever we write something to `stderr`, it'll be put into an actual file on the drive immediately

4.6.6.4 Controlling Buffer

There are some functions we can use to control the stream buffer:

```
int setvbuf(FILE* stream, char* buf, int mode, size_t size);
void setbuf(FILE* stream, char* buf);
void setbuffer(FILE* stream, char* buf, size_t size);
void setlinebuf(FILE* stream);
```

The `mode` argument `setvbuf()` can be one of the following macros:

- `_IONBF`: unbuffered;

- `_IOLBF`: line buffered;

- `_IOFBF`: fully buffered;

If you want to use `stdio` stream buffer, pass `NULL` to `buf`; otherwise you can use your own buffer by declaring a `char` array with length of `size`.

5 Processes

5.1 Introduction

When we invoke a program, the actual instance of that program becomes a **process**. A process usually starts when it's executed and ends when it returns from `main()` or calls other system calls to exit.

5.1.1 Process Image

The layout of a process is called a **process image**.Each process image is split into two poriotns; the user-addressable and the kernel-addressable. The kernel-addressable portions are all maintained in the kernel space of the memory, while the user-addressable portions the user space. Additionally, each process also has a process structure (PCB) to store all its status information.

5.1.1.1 Process structures

The kernel maintains a process structure for every running process called the **process control block (PCB**, meaning whenever we start running a program, the kernel creates an object of the structure. This structure contains all the information that the kernel needs to manage the process. In Linux, this structure is called task structure. There are many fields of which the most important is Process ID or **PID**. Each process has its own unique PID, and the kernel assigns PID to the process when it starts running.

In Linux, the PID's type is `pid_t`. If your C program needs to know its PID when running, you can use the following function:

```
1  pid_t getpid();
```

5.1.1.2 Memory Maps

There is a memory map for each process. Memory maps are used to indicate where all segments start. The variables are declared as `unsigned long` and are virtual memory addresses; they depict where each segment starts and ends.

5.1.2 The /proc/ Virtual File System

For every running process, the system also maintains a virtual file system under the directory `/proc/`. We say it's vritual because it doesn't take any physical disk space; it's in the form of a file just for the convenience of inspecting running process status.

5.1.3 Process States

For each process, there are five possible states:

- **Running (R)**: those who are actively using CPU

- **Uninterruptible Sleep (D)**: they are not doing anything and thus don't use CPU. They do not take up memory space, though. For example, when a program is waiting for user input

- **Interruptible Sleeping (S)**: it is similar to uninterruptible sleep state, except that it'll respond to signals

- **Stopped (T)**: it's similar to sleeping, but it's done manually. We can manually bring a stopped process back to running state

- **Zombie (Z)**: they are finished so they do not take any resources such as CPU and memory, but for some reason they still appear in the process table

5.1.4 System Process Hierarchy

Upon booting, Linux systems first launch a process called `init`, which is a system and service manager. This is the very first process of the entire system, and therefore it has a PID of 1. `init` then launches all other processes and becomes the ultimate parent of all other processes. Eventually, all the processes create a hierarchy in the system.

5.2 Process Control

5.2.1 Creating a Process

Whenever we invoke a program from terminal, we have actually created a process. We can create a process from our program by using `fork()`:

```
1  pid_t fork();
```

the program that calls `fork()` is called the parent process, while the program invoked by `fork()` is called the child process. A **child process** at the moment of successful call of `fork()` is just a copy of the parent - same code, same data, same everything. In the `fork()` function, the parent retruns its child's PID, while the child will return 0. Therefore, a successful `fork()` always returns twice: once in the parent and once in the child. You can use `getppid()` to get the parent process' PID:

```
1  #include <unistd.h>
2  pid_t getppid();
```

5.2.2 Parent vs. Child Processes

There are several attributes inherited by the child process from the parent, but also differences.

Properties inherited by child:

○ real user ID, real goup ID, effective user ID, effective group ID

○ supplementary group IDs

○ process group ID

○ session ID

○ controlling terminal

○ the set-uder-ID and set-group-ID flags

○ current working directory

○ root directory

○ file mode creation mask

○ signal ask and dispositions

○ the close-on-exec flag for any open file descriptors

○ environment

○ attached shared memory segments

○ memory mappings

○ resource limits

Differences between parent and child

○ the return values from `fork()` are different

○ the process IDs are different

○ the two processes have different parent process IDs

○ the child's process timer values are set to 0

○ file locks set by parent are not inherited by the child

○ pending alarms are cleared for the child

○ the set of pending signals for the child is set to the empty set

5.2.3   Avoid Zombies

To avoid zombies, the functions we want to use in the parent process are:

```
1  #include <sys/types.h>
2  #include <sys/wait.h>
3  pid_t wait(int* status);
```

When a process starts executing `wait()`, if it has no children, `wait()` returns immediatelyt witha -1. If this process has a child or multiple child processes, it will halt at the line where `wait()` is called, until one of its child processes has finihsed. Once a child process has finished, the parent process will resume executing from `wait()`, and the return value of `wait()` is the PID of that terminated child process. The parameter of `wait()` is an integer pointer.

5.2.3.1   The `waitpid()` Function

Another function we could use is `waitpid()`:

```
1  #include <sys/types.h>
2  #include <sys/wait.h>
3  pid_t waitpid(pid_t pid, int* status, int options);
```

It is very similary to `wait()`, but it's more flexible. The first argument, `pid`, can be one of the following:

- ○ `< -1`: Wait for any child process whose process ID is equal to the absolute value of PID

- ○ `-1`: As long as one of the child process terminates, the parent stops waiting for others

- ○ `0`: Wait for any child process whose process group ID is equal to that of the calling process

- ○ `> 0`: Wait for the child whose process ID is equal to the value of `pid`

Argument `options` can be one or more (by using the OR operator) of the following macros:

- ○ `WCONTINUED`: returns if a stopped child has been resumed by delivery of `SIGCONT`

- ○ `WNOHANG`: returns immediately if no child has exited

- ○ `WUNTRACED`: returns if a child has stopped

5.3   Executing Programs

6 Inter-Process Communication (IPC)

7 Threads