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

```
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);
```

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.

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

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:

```
#include <sys/types.h>
#include <sys/wait.h>
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()`:

```
#include <sys/types.h>
#include <sys/wait.h>
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

To create a process hierarchy, the first step is to `fork()` a child or miultiple child rpcoess from the very first proces `init`. The second step is to start executing different rpograms within those child processes. UNIX has a family of functions with `exec` as their prefix that can be used to invoke a binary within a program. There are many variations, but they are all just wrappers for one function: `execve()` system call.

5.3.1 Using the `execve()` System Call

The declaration of `execve()` is as follows:

```
#include <unistd.h>
int execve(const char* filename, char* const argv[],
           char* const envp[]);
```

`filename` is the program we want to execute, `argv` is the argument list to that program, and `envp` is the environment variable list. The filename is the one that we want to execute by calling `execve()`, and it has to be a binary file.

5.3.2 Wrapper Calls in `exec` Family

The following lists all the functions in `exec` family.

```
#include <unistd.h>
extern char** environ;

/* Requires a pathname */
int execl (const char* path, const char* arg, ...);
int execv (const char* path, char* const argv[]);
int execle(const char* path, const char* arg,...,
           char* const envp[]);
int execve(const char* path, char* const argv[],
           char* const envp[]);

/* Requires a filename */
int execvp(const char* file, char* const argv[]);
int execlp(const char* file, const char* arg, ...);
```

In summary, all the functions start with `exec`. The next suffix is:

- `l`: a list of arguments. That is, put your arguments as individual strings when calling these functions.
- `v`: vector/array of arguments. When you use these functions, put all arugment strings into a char*[], and only pass this array.

5.3.3 Environment Variables

In some POSIX standards, `main()` doesn't have `envp` in the argument list. In that case, we can declare an external variable like this:

```
extern char** environ;
```

Remember for both `argv` and `envp` we need `NULL` at the end of the array, so that we don't need to use another integer to record the length of the array.

The `PATH` environment variable specifies the directories to be searched to find a commnad. The default systemwide `PATH` value is specified in `/etc/profile` file and each user normally has a `PATH` value in the user's `$HOME/.profile` file. The `PATH` value in the `.profile` file either overrides the systemwide `PATH` value or adds extra directories to it.

5.3.4 Using `system()`

An alternative to `exec` functions is `system()`.

```
#include <stdlib.h>
int system(const char* command);
```

You can just write the command you'd put into the terminal as a string and pass it to `system()`. From the manpage, the `system()` library function uses `fork(2)` to create a child process that executes the shell command specified in `command` using `execl()`.

5.4 Organization of Processes

5.5 Signals

5.5.1 General Concepts of Signal

5.5.1.1 Sending & Receiving Signals

Kernel sends (delivers) a signal to a destinatin process by updating some state in the context of the destination process. It sends a signal for one of the following reasons:

- kernel has detected a system event such as a divide-by-zero (`SIGFPE`) or the termination of a child process (`SIGCHLD`)
- another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process

When a signal has been delivered to the destination process, the process has to handle it. There are three cases:

- **Ignore it.** Many signals can be and are ignored, but not all. Hardware exceptions such as divide-by-zero (`SIGFPE`) cannot be ignored successfully and some signals such as `SIGKILL` and `SIGSTOP` cannot be ignored at all
- **Catch and handle the excpetion.** The process has a functin to be executed if and when the exception occurs. The function may terminate the program gracefully or it may handle it without terminating the program
- **Let the default action apply.** Every signal has a default action. THe default may be ignore, terminate, terminate and dump core, stop and pause the program, resume a program paused earlier.

Each signal has a current "disposition" which indicates what action will be the default.

5.5.1.2 Pending & Blocked Signals

A signal is **pending** if sent but not yet received. There can be at most one pending signal of any partiular type, because signals are not queueed. What this means is, if a process has a pending signal of type `SIGINT`, then subsequent signals of type `SIGINT` that are sent to that process are discarded. A process can also **block** to receipt of certain signals. Blocked signals can be delivered, but will not be received until the signal is unblocked.

5.5.2 Reacting to Signals

5.5.2.1 Key-Binded Signals

When we are pressing `Ctrl+C`, it sends signal `SIGINT` to the process, and default action of `SIGINT` is to terminate the program. THat's why `Ctrl+C` terminates the program. There are three keybindings that can send signals to the process as shown below:

- `Ctrl+C` is `SIGINT` whose default action is to terminate
- `Ctrl+Z` is `SIGTSTP` whose default action is to stop or pause the program
- `Ctrl+\` is `SIGQUIT` whose default action is to terminate and dump core

5.5.2.2 Altering Default Actions

Default actions of most of the signals can be altered by us, except `SIGKILL` and `SIGSTOP` which cannot be caught, blocked, or ignored. This is an example of changing the default handling of signals

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <unistd.h>
```

```
5
6  void sig_handler(int sig) {
7      printf("Interrupt\n");
8  }
9
10 int main() {
11     /* installing the signal handler */
12     if (signal(SIGINT, sig_handler) == SIG_ERR)
13         printf("can't␣catch␣SIGINT\n");
14
15     /* main code...*/
16
17 }
```

In this example, the `SIGINT` is the signal to be received, which will interupt the process. When we hit `Ctrl+C` to a running program, this signal will be send to the process from the kernel and, therefore, when our main program receives the signal, the newly defined actin will be executed.

The function `signal()` is declared as follows:

```
1 #include <signal.h>
2 typedef void (*sighandler_t)(int);
3 sighandler_t signal(int signum, sighandler_t handler);
```

where `sighandler_t` is typedef'd as a function pointer, pointing to a function with an integer parameter and a `void` type. Thus, we know that our self-defined signal handler has to be defined list this:

```
1 void our_signal_handler(int sig);
```

which returns `void` and receives exactly one integer. Argument `signum` is the signal macro to which you'd like to change action, wheree `handler` is the functin name of our self-defined signal handler. Now that we have a signal handler, when we press `Ctrl+C` the process will not be terminated; instead it will trigger `sig_handler()`, and keep running.

The calling of `signal()` in the code is called installing the signal handler. This means that the function that handles the signal (i.e. signal handler) will be registered in the kernel, so that later on when the corresponding signal was received, the kernel can call the signal handler. When the signal arrives, the execution of handler is called **catching** or **handling** the signal.

5.5.3 Sending Signals

Just like other commands, there's a corresponding C function called `kill()` that we can use to send signals in our program

```c
#include <signal.h>
int kill(pid_t pid, int sig);
```

On success (at least one signal was sent), 0 is returned. On error -1 is returned, and `errno` is set to indicate the error. There is another function called `raise()` that is defiend as follows:

```c
#include <signal.h>
int raise(int sig);
```

which sends a signal to the executing process (i.e. the current process). Therefore, use of `kill()` is equivalent to

```c
raise(SIGINT);
```

6 Inter-Process Communication (IPC)

6.1 Pipes

Pipes on the command level can be used like this

```
last | grep 'reboot'
```

which connects the output of `last` command to the input of `grep`. The `'|'` is a bash operator; it causes bash to start the `last` command and the `grep` command **simultaneously**, and to direct the standard output of `last` into the standard input of `grep`. Although `'|'` is a bash oerator, it uses the lower-level, underlying pipe facility of UNIX. You can visualize the pipe mechanism as a special file or buffer that acts quite literally like a physical pipe, connecting the output of `last` to the input of `grep`.

6.1.1 Introduction

A pipe is not a file, and there is no file pointer associated with it. Conceptually, pipes are like a conveyor belt consisitng of a fixed number of logical blocks that can be filled and emptied. These types of pipes are called **unnamed pipes** because they do not exist anywhere in the file system. They do not have names. In Linux, a pipe is created with the `pipe()` function:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

The function `pipe(fd)`, given an integer array `fd` of size 2, creates a pair of file descriptors, `fd[0]` and `fd[1]`, pointing to the "read-end" and "write-end" of a pipe inode respectively. If it is successful, it returns a 0, otherwise it returns -1. The process can then write to the write-end, `fd[1]`, using the `write()` funtion, and can read from the read-end, `fd[0]`, using `read()`. The read and write-ends are opened automatically as a result of the `pipe()` call. Written data is read in **first-in-first-out (FIFO)** order.

6.1.2 Full- & Half-Duplex

Pipes exist in order to allow two different processes to communicate. Typically, a process creates a pipe, and then forks a child process. After the fork, the parent and child will each have copies of the read and write-ends of the pipe, so there will be two data channels and a total of four descriptors. This is called a **full-duplex**.

Full-duplex is only implemented on some UNIX systems. POSIX standards do not allow full-duplex; it only allows **half-duplex**, meaning the message is

unidirectional. In practice, we have to close a file descriptor in both parent and child processes. For examples, if we want the child to read from parent, we should close child's `fd[1]` and paren'ts `fd[0]`. For a half-duplex, the program strucutre usually is as follows:

```
#define WRITE_END 1
#define  READ_END 0

if ( pipe(fd) == -1) { exit(EXIT_FAILURE); }

if (fork() == 0) {
    /* child process */
    close(fd[WRITE_END]); /* close write-end */
    bytesread = read(fd[0], message, BUFSIZ);
    /* check for errors afterward of course */
} else {
    /* parent process */
    close(fd[READ_END]);
    byteswritten = write(fd[1], buffer, strlen(buffer));
}
```

There are two things to consider with the half-duplex:

1. If we read from a pipe whose write end has been close, `read()` returns 0 to indicate an end of file after all the data has been read;
2. If we write to a pipe whose read end has been closed, the signal `SIGPIPE` (or broken pipe) is generated. If we either ignore the signal or catch it and return from the signal handler, `write()` returns `-1` with `errno` set to `EPIPE`.

6.1.3 I/O Redirection

6.1.3.1 Redirecting Standard Output

The key to understanding I/O redirection in the shell is knowing that the `open()` function always chooses the lowest numbereed available file descriptor. Suppose you have entered the command:

```
ls > listing
```

The steps taken by the shell are:

1. `fork()` a new process
2. In the new process, `close()` file descriptor 1 (standard output)
3. In the new process, `open()` (with the `O_CREAT` flag) the file named `listing`
4. Let the new process `exec()` the `ls` program.

6.1.3.2 The Pipe Operator |

To write a program to simulate how the shell carries out a command such as

```
last | grep 'tty'
```

This cannot be accomplished only using `open()`, `close()`, and `pipe()`, because we need to connect one end of a pipe to a standard output for `last`, and the other end to the standard input for `grep`. Just cloning the file descriptors cannot do this. There are two functions that can be used for this purpose `dup()` and `dup2()`, and thfeir purpose is to duplicate a file descriptor. Their declaratinos are as follows:

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Given a file descriptor `oldfd`, `dup()` creates a new file descriptor that points to the same kernel file structure as the old one. The critical feature if `dup()` is that it returns the lowest numbered available file descriptor. To understand this better, let's assume we have two processes, `last` and `grep 'tty'`, where we want to feed the output of `last` to `grep`. See the following implementation in C code, where one of the processes is the parent and the other is the parent of the other (so that they can share a pipe).

src/dup1.c

```
/*** dup1.c ***/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char const *argv[]) {

    int  fd[2];

    if (argc < 3) {
        fprintf(stderr, "Usage: %s <command1> <command2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(fd) == -1) {
        perror("pipe call");
        exit(EXIT_FAILURE);
    }

    switch (fork()) {
        case -1:
            perror("fork");
            exit(EXIT_FAILURE);

```

```
26          /* Child process */
27          case 0:
28
29
30          /* Parent process */
31          default:
32
33      }
34      return 0;
35  }
```

An **atomic operation** refers to an operation that executes as a single, inseparable, and uninterruptable unit. For example `open()` is atomic, because until the function has finished, no other operations can interrupt. Non-atomic operatins can lead to **race conditions**.

There are two major flaws with the code in `dup1.c` above...

- When the parent process executes the `grep` command, its entire process image will be replaced by the program of `grep`. Thus, it cannot reap its child process and will risk making the child process a zombie
- There are two steps: close `stdout` and then `dup()` the write end of the pipe. There is small window of time between closing standard output and duplicating the write-end of the pipe. During this time, the child could be interruipted by a signal whose handler might make the descriptor closed unavailable. In this case, the descriptor returned by `dup()` will not be the one that was just closed.

To resolve the first flaw, we can just create two child proceses, one for each program, so the parent process can wait for both of them.

The second flaw is the reason that `dup2()` exists/is used. `dup2(fd1, fd2)` will duplicate `fd1` and `fd2`, closing `fd2` if necessary, as a **single atomic operations**. In other words, if `fd2` is open, it will close it, and make `fd2` point to the same file structure to which `fd1` pointed.

Making the change from `dup()` to `dup2()` will yield the improved code below:

src/dup2.c

```
1   /*** dup2.c ***/
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <stdlib.h>
5   #include <fcntl.h>
6
7   int main(int argc, char const *argv[]) {
8
9       int     fd[2];
10      int     i;
11      pid_t   child1, child2;
```

```
12
13      if (argc < 3) {
14          fprintf(stderr, "Usage: %s <command1> <command2>\n", argv[0]);
15          exit(EXIT_FAILURE);
16      }
17
18      if (pipe(fd) == -1) {
19          perror("pipe call");
20          exit(EXIT_FAILURE);
21      }
22
23      switch (child1 = fork()) {
24          case -1:
25              perror("fork");
26              exit(EXIT_FAILURE);
27
28          /* Child process */
29          case 0:
30
31
32          /* Parent process */
33          default:
34              switch (child2 = fork()) {
35                  case -1:
36                      perror("fork");
37                      exit(EXIT_FAILURE);
38                  case 0:
39
40                  default:
41
42                      return 0;
43              }
44      }
45      return 0;
46  }
```

6.1.4 The `popen()` Library Function

To create an IPC betwen the parent and child processes, there are several steps:

1. generate a pipe
2. fork a child process
3. duplicate file descriptors
4. execute a new program in order to redirect the input or output of that program to the parent

This sequence is so common that there is a pair of functions `popen()` and `pclose()` to streamline this procedure:

```
1   #include <stdio.h>
```

```
FILE* popen(const char* command, const char* type);
int   pclose(FILE* stream);
```

The `popen()` function creates a pipe, forks a new process to execute the shell `/bin/sh`, and passes the command to that shell to be executed by it. `popen()` expects the second argument to be either `"r"` or `"w"`:

- `"r"`: the process invoking it will be returned a `FILE` pointer to the read-end of the pipe, and the write-end will be attached to the `stdout` of the command
- `"w"`: the process invoking it will be returned a `FILE` pointer to the write-end of the pipe, and the read-end will be attached to the `stdin` of the command. The output stream is fully buffered.

See the code below which works like the above examples, but uses `popen()` and `pclose()` this time.

src/dup3.c

```c
/*** dup3.c ***/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define FAILED_POPEN \
        fprintf(stderr, "popen() failed\n"); \
        exit(EXIT_FAILURE);

int main(int argc, char const *argv[]) {

    int    nbytes;
    FILE*  fin;    /* read-end of the pipe */
    FILE*  fout;   /* write-end of the pipe */
    char   buffer[PIPE_BUF];

    if (argc < 3) {
        fprintf(stderr, "Usage: %s <command1> <command2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((fin = popen(argv[1], "r")) == NULL) {
        FAILED_POPEN;
    }

    if ((fout = popen(argv[2], "w")) == NULL) {
        FAILED_POPEN;
    }

    while ((nbytes = read(fileno(fin), buffer, PIPE_BUF)) > 0) {
        write(fileno(fout), buffer, nbytes);
    }
    pclose(fin);
```

```
36      pclose(fout);
37
38      return 0;
39  }
```

6.1.5 Pipe Capacity

Like any other I/O operations, pipes also have a capacity, meaning you can only read/write a limited amount of data if you want to make sure the operation is atomic. In Linux, this is declared as a macro `PIPE_BUF`, which has a value of 4,096 bytes.

7 Threads