Section 2: Files, Pipes, Signals, Dup, Sockets

CS 162

February 5, 2021

Contents

1 Vocabulary

- **file descriptors** - File descriptors are an index into a file-descriptor table stored by the kernel. The kernel creates a file-descriptor in response to an open call and associates the file-descriptor with some abstraction of an underlying file-like object; be that an actual hardware device, or a file-system or something else entirely. Using file descriptors, a process's read or write calls are routed to the correct place by the kernel. When your program starts you have 3 file descriptors.

File Descriptor	File
0	stdin
1	stdout
2	stderr

- **int open(const char *path, int flags)** - open is a system call that is used to open a new file and obtain its file descriptor. Initially the offset is 0.
- **size_t read(int fd, void *buf, size_t count)** - read is a system call used to read `count` bytes of data into a buffer starting from the file offset. The file offset is incremented by the number of bytes read.
- **size_t write(int fd, const void *buf, size_t count)** - write is a system call that is used to write up to `count` bytes of data from a buffer to the file offset position. The file offset is incremented by the number of bytes written.
- **size_t lseek(int fd, off_t offset, int whence)** - lseek is a system call that allows you to move the offset of a file. There are three options for whence
  - SEEK_SET - The offset is set to `offset`.
  - SEEK_CUR - The offset is set to current_offset + `offset`
  - SEEK_END - The offset is set to the size of the file + `offset`
- **test_and_set** - An atomic operation implemented in hardware. Often used to implement locks and other synchronization primitives. In this handout, assume the following implementation.

```
int test_and_set(int *value) {
    int result = *value;
    *value = 1;
    return result;
}
```

  This is more expensive than most other instructions, and it is not preferable to repeatedly execute this instruction.

- **pipe** - A system call that can be used for interprocess communication.

  More specifically, the `pipe()` syscall creates two file descriptors, which the process can `write()` to and `read()` from. Since these file descriptors are preserved across `fork()` calls, they can be used to implement inter-process communication.

```
/* On error, pipe() returns -1. On success, it returns 0
 * and populates the given array with two file descriptors:
 *  - fildes[0] will be used to read from the data queue.
 *  - fildes[1] will be used to write to the data queue.
 *
 * Note that whether you can write to fildes[0] or read from
 * fildes[1] is undefined. */
int pipe(int fildes[2]);
```

- **int dup(int oldfd)** - creates an alias for the provided file descriptor and returns the new fd value. dup always uses the smallest available file descriptor. Thus, if we called dup first thing in our program, it would use file descriptor 3 (0, 1, and 2 are already signed to stdin, stdout, stderr). The old and new file descriptors refer to the same open file description and may be used interchangeably.
- **int dup2(int oldfd, int newfd)** - dup2 is a system call similar to dup. It duplicates the `oldfd` file descriptor, this time using `newfd` instead of the lowest available number. If newfd was open, it closed before being reused. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the file descriptor, performing the redirection in one elegant command. For example, if you wanted to redirect standard output to a file, then you would simply call dup2, providing the open file descriptor for the file as the first command and 1 (standard output) as the second command.
- **signals** - A signal is a software interrupt, a way to communicate information to a process about the state of other processes, the operating system, and the hardware. A signal is an interrupt in the sense that it can change the flow of the program when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.
- **int signal(int signum, void (*handler)(int))** - signal() is the primary system call for signal handling, which given a signal and function, will execute the function whenever the signal is delivered. This function is called the signal handler because it handles the signal.
- **SIG_IGN, SIG_DFL** Usually the second argument to signal takes a user defined handler for the signal. However, if you'd like your process to drop the signal you can use SIG_IGN. If you'd like your process to do the default behavior for the signal use SIG_DFL.
- **TCP** - Transmission Control Protocol (TCP) is a common L4 (transport layer) protocol that guarantees reliable in-order delivery. In-order delivery is accomplished through the use of sequence numbers attached to every data packet, and reliable delivery is accomplished through the use of ACKs (acknowledgements).
- **Socket** - Sockets are an abstraction of a bidirectional network I/O queue. It embodies one side of a communication channel, meaning that two must be required for a communication channel to form. The two ends of the communication channel may be local to the same machine, or they may span across different machines through the Internet. Most functions that operate on file descriptors like read() or write() work on sockets. but certain operations like lseek() do not.

2 Files

2.1 Files vs File Descriptor

What's the difference between `fopen` and `open`?

```
fopen is implemented in libc whereas open is a syscall. fopen will use open
in it's implementation. fopen will return a FILE * and open will return an int.
The FILE * object allows you to call utility methods from
stdio.h like fscanf. Also the FILE * object comes with some library
level buffering of writes.


--------------
|  libc      |
-------------
| syscall    |
--------------
```

2.2 Quick practice with write and seek

What will the test.txt file look like after I run this program? For simplicity assume read() and write() do not return short. (Hint: if you write at an offset past the end of file, the bytes inbetween the end of the file and the offset will be set to 0.)

```
int main() {
    char buffer[200];
    memset(buffer, 'a', 200);
    int fd = open("test.txt", O_CREAT|O_RDWR);
    write(fd, buffer, 200);
    lseek(fd, 0, SEEK_SET);
    read(fd, buffer, 100);
    lseek(fd, 500, SEEK_CUR);
    write(fd, buffer, 100);
}
```

```
The first write gives us 200 bytes of a. Then we seek to the offset 0
and read 100 bytes to get to offset 100. Then we seek to offset
100 + 500 to offset 600. Then we write 100 more bytes of a.

At then end we will have a from 0-200, 0 from 200-600, and a from 600-700
```

2.3 Reading and Writing with File Pointers vs. Descriptors

Write a utility function, **void copy(const char *src, const char *dest)**, that simply copies the file contents from src and places it in dest. You can assume both files are already created. Also assume that the src file is at most 100 bytes long. First, use the file pointer library to implement this. Fill in the code given below:

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  FILE* read_file = fopen(________, ____);
  int buf_size = fread(__________, ____, __________, __________);
  fclose(read_file);

  FILE* write_file = fopen(________, ____);
  fwrite(__________, ____, __________, __________);
  fclose(write_file);
}
```

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  FILE* read_file = fopen(src, "r");
  int buf_size = fread(buffer, 1, sizeof(buffer), read_file);
  fclose(read_file);

  FILE* write_file = fopen(dest, "w");
  fwrite(buffer, 1, buf_size, write_file);
  fclose(write_file);
}
```

Next, use file descriptors to implement the same thing.

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  int read_fd = open(________, ________);
  int bytes_read = 0;
  int buf_size = 0;

  while ((bytes_read = read(__________, __________, __________)) > 0) {
    _____________________
  }
  close(read_fd);

  int bytes_written = 0;
  int write_fd = open(________, ________);
  while (_____________________) {
    __________ += write(__________, __________, __________);
  }
  close(write_fd);
}
```

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  int read_fd = open(src, O_RDONLY);
  int bytes_read = 0;
  int buf_size = 0;

  while ((bytes_read = read(read_fd, &buffer[buf_size], sizeof(buffer) - buf_size)) > 0) {
    buf_size += bytes_read;
  }
  close(read_fd);

  int bytes_written = 0;
  int write_fd = open(dest, O_WRONLY);
  while (bytes_written < buf_size) {
    bytes_written += write(write_fd, &buffer[bytes_written], buf_size - bytes_written);
  }
  close(write_fd);
}
```

Compare the file pointer implementation to the file descriptor implementation. In the file descriptor implementation, why does **read** and **write** need to be called in a loop?

```
Read and write need to be called in a loop because there is no guarantee
that both functions will actually process the specified number of bytes
(they can return less bytes read / written). However, this functionality
is already handled in the file pointer library, so a single call to
fread and fwrite would suffice.
```

3 Pipes

3.1 Basic Pipes

In the following code we use a pipe to communicate data between 2 file descriptors.

```
int main() {
    int fds[2];
    pipe(fds);
    int rfd = fds[0];
    int wfd = fds[1];
    char *str = "hello world";
    size_t bytes_written = 0;
    size_t total = 0;
    while (bytes_written = write(wfd, &str[total], strlen(&str[total]) + 1)) {
        total += bytes_written;
        if (str[total - 1] == '\0') break;
    }
    close(wfd);
    char *read_str = malloc(strlen(str) + 1);
    total = 0;
    size_t bytes_read;
    while (bytes_read = read(rfd, &read_str[total], 50)) {
        total += bytes_read;
    }
    printf("%s", read_str);
    return 0;
}
```

What would the code above print out?

hello world

3.2 Pipe and Fork

Now, we use pipes in order for 2 processes to share data between each other.

```
int main() {
    int fds[2];
    pipe(fds);
    pid_t child_pid = fork();
    size_t total = 0;
    char *str = "hello world";
    int rfd = fds[0];
    int wfd = fds[1];
    if (child_pid == 0) {
        size_t bytes_written = 0;
        while ((bytes_written = write(wfd, &str[total], strlen(&str[total]) + 1))) {
            total += bytes_written;
            if (str[total - 1] == '\0') break;
        }
    } else {
        close(wfd);
        char *read_buf = malloc(strlen(str) + 1);
        total = 0;
        size_t bytes_read;
        while ((bytes_read = read(rfd, &read_buf[total], 50))) {
            total += bytes_read;
        }
        printf("%s\n", read_buf);
    }
    return 0;
}
```

What would the code above print out?

hello world

(files are automatically closed on exit so we can omit `close(wfd)` at the end of the child process)

4 Signals

The following is a list of standard Linux signals:

Signal	Value	Action	Comment
SIGHUP	1	Terminate	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Terminate	Interrupt from keyboard (Ctrl - c)
SIGQUIT	3	Core Dump	Quit from keyboard (Ctrl - \\)
SIGILL	4	Core Dump	Illegal Instruction
SIGABRT	6	Core Dump	Abort signal from abort(3)
SIGFPE	8	Core Dump	Floating point exception
SIGKILL	9	Terminate	Kill signal
SIGSEGV	11	Core Dump	Invalid memory reference
SIGPIPE	13	Terminate	Broken pipe: write to pipe with no readers
SIGALRM	14	Terminate	Timer signal from alarm(2)
SIGTERM	15	Terminate	Termination signal
SIGUSR1	30,10,16	Terminate	User-defined signal 1
SIGUSR2	31,12,17	Terminate	User-defined signal 2
SIGCHLD	20,17,18	Ignore	Child stopped or terminated
SIGCONT	19,18,25	Continue	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

4.1 Signal Handlers

Assume you are running this program from a Bash shell. List all the ways you can cause this program to exit using signals.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void sigint_handler(int sig) {
    if (sig == SIGINT || sig == SIGQUIT) {
        exit(1);
    }
}
void sigint_handler_2(int sig) {
    if (sig == SIGINT) {
        signal(SIGINT, sigint_handler);
    }
}
int main() {
    signal(SIGINT, sigint_handler_2);
    signal(SIGQUIT, sigint_handler);
    while (1) {
        printf("Sleeping for a second (U.U)\n");
        sleep(1);
    }
}
```

- Enter Ctrl+C twice
- Enter Ctrl+\once
- Enter Ctrl+C, then Ctrl+\

5 Dup and Dup2

What does C print in the following code?

```
int main(int argc, char **argv)
{
    int pid, status;
    int newfd;
    if ((newfd = open("output_file.txt", O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        exit(1);
    }
    printf("The last digit of pi is...");
    fflush(stdout);
    dup2(newfd, 1);
    printf("five\n");
    exit(0);
}
```

This prints "The last digit of pi is..." to standard output. Unfortunately, "five" gets written to the output_file.txt and our joke is left incomplete.

6 Socket Programming

6.1 Multi-threaded Echo Server

Write a server implementation that uses multiple threads in a single process. Each connection is handled in its own thread, and threads should be allowed to handle connections concurrently. For simplicity assume `read()` and `write()` do not return short.

```
#define BUF_SIZE 1024

struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int rv = getaddrinfo(NULL, port, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}

void *serve_client(void *client_socket_arg) {
    int client_socket = (int)client_socket_arg;
    char buf[BUF_SIZE];
    ssize_t n;

    while ((n = read(client_socket, buf, BUF_SIZE)) > 0) {
        buf[n] = '\0';
        printf("Client Sent: %s\n", buf);

        if (write(client_socket, buf, n) == -1) {
            close(client_socket);
            pthread_exit(NULL);
        }
    }

    close(client_socket);
    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <port>\n", argv[0]);
        return 1;
    }
```

```
        struct addrinfo *server = setup_address(argv[1]);
        if (server == NULL) {
            return 1;
        }
        int server_socket = socket(server->ai_family,
                                server->ai_socktype, server->ai_protocol);
        if (server_socket == -1) {
            return 1;
        }
        if (bind(server_socket, server->ai_addr,
             server->ai_addrlen) == -1) {
                return 1;
        }
        if (listen(server_socket, 1) == -1) {
            return 1;
        }

        while (1) {
            int connection_socket = accept(server_socket, NULL, NULL);
            if (connection_socket == -1) {
                perror("accept");
                pthread_exit(NULL);
            }

            pthread_t handler_thread;
            int err = pthread_create(&handler_thread, NULL,
                    serve_client, (void *)connection_socket);
            if (err != 0) {
                printf("pthread_create: %s\n", strerror(err));
                pthread_exit(NULL);
            }
            pthread_detach(handler_thread);
        }
        pthread_exit(NULL);
    }
```

7 Shared Data

This problem is designed to help you with implementing the wait syscall in your project. For this problem, we're going to implement a wait function that allows one thread (the main thread) to wait for another thread to finish writing some data.

We're going to assume we don't have access to `pthread_join` for this problem. Instead, we're going to use synchronization primitives (locks and semaphores).

We need to design a struct for sharing information between the two threads. We also need to implement three functions to initialize the shared struct and synchronize the 2 threads. `initialize_shared_data` will initialize our shared struct. `wait_for_data` (called by the main thread) will block until the data is available. `save_data` (called by the child thread) will write 162 to the struct. Another requirement is that the shared data needs to be freed once it is no longer in use.

Here we have already designed a possible struct for sharing information:

```
typedef struct shared_data {
    sem_t semaphore;
    pthread_mutex_t lock;
    int ref_cnt;
    int data;
} shared_data_t;
```

For each member in `shared_data` above, describe its purpose.

- The semaphore allows to perform the waiting itself. A waiting thread could down the semaphore where the thread that saves the data can up the semaphore.
- The lock allows to have mutual exclusion on members in the struct that can be modified.
- The first integer allows to perform **reference counting**. It is an indicator for how many threads still hold a reference to this struct. Once it reaches 0, we can safely deallocate it from memory.
- The second integer is the data itself that is shared.

For the following questions, refer to the following `main` function so that it prints "Parent: Data is 162"

```
int main() {
    void *shared_data = malloc(sizeof(shared_data_t));
    initialize_shared_data(shared_data);
    pthread_t tid;
    int error = pthread_create(&tid, NULL, &save_data, shared_data);
    int data = wait_for_data(shared_data);
    printf("Parent: Data is %d\n", data);
    return 0;
}
```

For `initialize_shared_data`, write out the pseudo-code needed to initialize all members of `shared_data`.

```
void initialize_shared_data(void *shared_pg) {
    shared_data_t *shared_data = (shared_data_t *) shared_pg;
    // init semaphore to 0
    // init lock
    // init ref_cnt to 2
    // init data to a "invalid" number (i.e. -1)
}
```

For `wait_for_data`, write out the pseudo-code needed for a thread to wait for another thread to write data to `shared_data`. Remember to deallocate the `shared_data` if it is not needed anymore.

```
int wait_for_data(void *shared_pg) {
    shared_data_t *shared_data = (shared_data_t *) shared_pg;
    // down the semaphore to wait
    // read the data
    // acquire the lock
    // decrement reference count
    // release the lock
    // free struct if ref_cnt is 0 (i.e. set a flag in the critical section)
}
```

For `save_data`, write out the pseudo-code needed for a thread to save data and signal to waiting threads. Remember to deallocate the `shared_data` if it is not needed anymore.

```
void *save_data(void *shared_pg) {
    shared_data_t *shared_data = (shared_data_t *) shared_pg;
    // set data to 162
    // up the semaphore to signal to a waiting thread
    // acquire the lock
    // decrement reference count
    // release the lock
    // free struct if ref_cnt is 0 (i.e. set a flag in the critical section)
}
```