Section 3

Friday, April 19, 2019 1:55 PM

section3

Section 3: Syscalls and I/O

September 11-12, 2018

Contents

1 Vocabulary

- **system call** - In computing, a system call is how a program requests a service from an operating system's kernel. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling.
- **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. Consequently a process's read or write calls that reference that file-descriptor are routed to the correct place by the kernel to ultimately do something useful. Initially when your program starts you have 3 file descriptors.

File Descriptor	File
0	stdin
1	stdout
2	stderr

- **int open(const char *path, int oflags)** - 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 fildes, void *buf, size_t nbytes)** - read is a system call used to read n 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 fildes, const void *buf, size_t nbytes)** - write is a system call that is used to write data out of a buffer to the file offset position. The file offset is incremented by the number of bytes written.
- **size_t lseek(int filedes, 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`
- **int dup(int fildes)** - creates an alias for the provided file descriptor. dup always uses the smallest available file descriptor. Thus, if we called dup first thing in our program, then you could write to standard output by using file descriptor 3 (dup uses 3 because 0, 1, and 2 are already signed to stdin, stdout, stderr). You can determine the value of the new file descriptor by saving the return value from dup.
- **int dup2(int fildes, int fildes2)** - dup2 is a system call similar to dup. It duplicates one file descriptor, making them aliases, and then deleting the old file descriptor. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the old 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.

2 Problems

2.1 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
```

			of 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

2.1.1 Warmup

How do we stop the following program?

```
int main(){
  signal(SIGINT, SIG_IGN);
  while(1);
}
```

The program bypassed default sigint handling to sig_ign. We need another way to stop the process

Many options are available, one that is "keyboard ready" is ctrl \

2.1.2 Did you really want to quit?

Fill in the blanks for the following function using syscalls such that when we type Ctrl-C, the user is prompted with a message: "Do you really want to quit [y/n]? ", and if "y" is typed, the program quits. Otherwise, it continues along.

```
void sigint_handler(int sig)
{
    char c;
    printf(Ouch, you just hit Ctrl-C?. Do you really want to quit [y/n]?);
    c = getchar();
    if (c == 'y' || c == 'Y')
        ____________________________;
}
```

Raise(SIGHUP);

Alternative, use exit(0);
Does things the nromal way

```
int main() {
    ____________________________;
    ...

}
```

Signal(sigint, sigint_handler);

2.2 Files

2.2.1 Files vs File Descriptor

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

FD is how a OS handles files
A File is a HLPL (C) abstraction of a file

- (program library lelve) The fopen() function opens the file whose name is the string pointed to by path and associates a stream with it.
- (sys call level) Open returns a file descriptor for a file at a path, preparing it for subsequence r/w/seek calls

2.2.2 Quick practice with write and seek

What will the test.txt file look like after I run this program? (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);
}
```

Populate buffer w/ a's
Open a file called test.txt, for rd/wr
Write out 200 bytes from the buffer into fd
Reset the offset of fd to 0
Read out from the buffer up to 100 bytes
In your file, offset by 500 bytes (to byte 600)

'a'*200+'0'*400+'a'*100
File* has some more bells and whiwstles than a simple fd.

2.3 Dup and Dup2

2.3.1 Warmup

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("Luke, I am your...\n");
    dup2(newfd, 1);
    printf("father\n");
    exit(0);
}
```

up front !!

fd for I/O

Like, I am your...

2.3.2 Redirection: executing a process after dup2

Describe what happens, and what the output will be.

```
int
main(int argc, char **argv)
{
    int pid, status;
    int newfd;
    char *cmd[] = { "/bin/ls", "-al", "/", 0 };

    if (argc != 2) {
        fprintf(stderr, "usage: %s output_file\n", argv[0]);
        exit(1);
    }
    if ((newfd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        perror(argv[1]);    /* open failed */
        exit(1);
    }
    printf("writing output of the command %s to \"%s\"\n", cmd[0], argv[1]);
    dup2(newfd, 1);
    execvp(cmd[0], cmd);
    perror(cmd[0]);     /* execvp failed */

    exit(1);
}
```

skip

errrll

Sexy c programming :)

If we made it here, we fucked up defo-for-shure

Invokes ls and redirects stdout to a newly created file called std.out
If we fail to make the file, or we fail the ls command, we'll send exit 1 error codes.

Show me, ALVERYTHING

2.3.3 Redirecting in a new process

Modify the above code such that the result of ls -al is written to the file specified by the input argument and immediately after "all done" is printed to the terminal. (Hint: you'll need to use fork and wait.)

```c
int main(int argc, char **argv) {
    int pid, status;
    int newfd;
    char *cmd[] = { "/bin/ls", "-al", "/", 0 };
    if (argc != 2) {
        fprintf(stderr, "usage: %s output_file\n", argv[0]);
        exit(1);
    }
    if ((newfd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        perror(argv[1]); /* open failed */
        exit(1);
    }
    printf("writing output of the command %s to \"%s\"\n", cmd[0], argv[1]);
    dup2(newfd, 1);
    pid_t child = fork();
    if (child == 0) { //LUKE, YOU ARE MY CHILD
        execvp(cmd[0], cmd);
        perror(cmd[0]); /* execvp failed */
        exit(1);
    } else {
        wait(&status);
        printf("Reunited and it feeeeelss so gooood!");
    }
}
```