SIENA*college*
Computer Science

Computer Science 330
Operating Systems
Siena College
Fall 2020

Lab 7: Unix Systems Programming

Due: 11:59 PM, Wednesday, October 14, 2020

Quote: UNIX system calls, reading about those can be about as interesting as reading the phone book... – George Williams, Union College Computer Science, March 12, 1991.

(so we will not learn them through a lecture, but by trying them out!)

In this lab, you will learn and/or review several aspects of Unix systems programming, focusing on those things you will need for the upcoming shell project.

You must work alone on this lab.

Learning goals:

1. To gain experience with low-level file operations in Unix
2. To learn how to execute a new program in processes created by `fork`
3. To learn about signal handling in Unix
4. To learn about pipes in Unix

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `sysprog-lab-yourgitname`, for this lab. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

Answers to written questions may be given in a PDF document, `answers.pdf`, or by writing them in your repository's `README.md` file.

Examples related to this lab are in https://github.com/SienaCSISOperatingSystems/sysprog-examples. You probably want to clone a copy of it (but don't try to push to it – you will push to the repository you create through the GitHub Classroom link).

Low-level File Operations

You have used at least some of the C standard file I/O routines defined in `stdio.h`, such as `fopen()`, `fscanf()`, `fprintf()`, and `fclose()`. These provide relatively "high-level" access to files in that you deal with data types rather than a low-level stream of bytes.

Underneath the stdio functions, you will find those low-level operations: `open()`, `close()`, `read()`, `write()`.

Error Checking and Reporting

Before we look at the use of all of these, we recall the standard error reporting mechanism.

Most Unix system calls can fail for a variety of reasons. You should always check the return value of system calls that may fail.

Read the `intro(2)` man page on noreaster and the `errno(3)` and `perror(3)` man pages on a Linux system to learn about or refresh your knowledge of the `errno` error condition and the system calls `perror(3)` and `strerror(3)` that allow you to print out (hopefully) meaningful error messages when you detect a failed system call.

? **Question 1:**
The file `/usr/include/errno.h` in FreeBSD defines all of the names for the error conditions. What is the highest-numbered error in use on noreaster, and what does it represent? (Note: `ELAST` doesn't count, as it is a constant indicating the highest-numbered error code) (1 point)

For example, consider a program that uses the low-level `open` and `close` system calls:

`perror/perror-ex.c` in `https://github.com/SienaCSISOperatingSystems/sysprog-examples`.

Compile it in your clone of that repository (it has a `Makefile`).

? **Question 2:**
What is the output when you run the program? (1 point)

? **Question 3:**
Create the file "nonexistent.txt". Now what is the output when you run the program? (1 point)

? **Question 4:**
Modify the program so you get an error condition on the `close` system call. Briefly describe your modification and give the error reported. (2 points)

With Unix system calls, there are a lot of good reasons that something can fail. It's worth your trouble to check these return conditions and print meaningful error messages.

A More Complete Example

Whereas `fopen` returns a value of type `FILE *`, the `open` call returns an `int`. This `int` has a special meaning – it is a *file descriptor*. It can subsequently be used in `read` and `write` calls, and is later passed to `close` when we are done.

There are three file descriptors that are automatically created for each process:

`0`	the standard input (`stdin`)
`1`	the standard output (`stdout`)
`2`	the standard error output (`stderr`)

Read through the man pages for these four system calls, then consider this example:

`everyother/everyother.c` in `https://github.com/SienaCSISOperatingSystems/sysprog-examples`.

? Question 5:
The `open` calls take some flags as their second parameter. What do the flags mean in the two calls in this program? Note that a "bitwise or" is used to combine them on the second call. Why? (2 points)

? Question 6:
What is the output of the program when you use the C source code for the program itself as input? (1 point)

Running a New Program – the `exec` Calls

Recall that the `fork()` system call lets you have two copies of a process – each running the same program and executing at the statement immediately following the `fork()` call.

Sometimes this is what you want, but more likely you will want to start a new process to run some new program.

To create processes that do "something else", the `fork()` is followed by one of these "exec" calls, in the child process:

`execl()` – exec a process with list of arguments

`execv()` – exec a process with args specified in an array

`execlp()` – list, but search the existing path for the program.

`execvp()` – array, but search the existing path for the program.

`execvP()` – array, but specify a search path for the program.

The man pages have details.

The related `vfork()` system call is often more appropriate when the child process will be doing an `exec()` immediately. It doesn't duplicate all of the memory for the parent process. Beware: this may cause you trouble in the shell if you use it, since the parent is usually suspended until the child `exits` or calls an `exec`.

We consider a series of example programs, all in the `exec` directory of `https://github.com/SienaCSISOperatingSystems/sysprog-examples`.

Start by looking at the `exec` program:

- This one uses `execlp`. The parameters are the program to run and its arguments.
- This is a "varargs" function call – we can send any number of parameters.

? Question 7:
What is the output of the program when you run it? (1 point)

? Question 8:
Change the program so it attempts to "exec" a program that doesn't exist. What happens then? (1 point)

Note that we can specify a program by its name only (like `"ls"`), in which case the search path is used to try to find a program to run. We can also give a full path to the program (like `"/bin/ls"`) in which case the program must be at the exact path specified.

Next, we look at a program that doesn't use any of the "exec" calls, but which will be useful as we look at further examples: `procinfo`. This one simply prints the process id and the command-line parameters (including one beyond the last).

Use the `execprocinfo` program to execute `procinfo`.

? Question 9:
What does the output tell us about the value provided in `argv[0]`? (1 point)

Next, look at `exec2`, which uses `execvp()` instead of `execlp()`. This is the "list" form rather than the "varargs" form. We pass a `NULL`-terminated array of parameters.

? Question 10:
Run the program `exec2nonull`, which first "forgets" the `NULL` in the array, then later adds it in but not immediately. Explain the results. (3 points)

Our last example program is `execwithargs`, which uses its command-line parameters to determine which program it should become (weird).

? Question 11:
Use this program to execute `procinfo`. What command line did you use? What was the output? (2 points)

? Question 12:
What would happen if we mistakenly use `argv[0]` for both parameters to the `execvp` call? (1 point)

Question 13:
Use the program to execute itself 3 times before executing some other program. What command line did you use? What was the output? (2 points)

Practice With `exec`

Practice Program:
Write a program `execlsloop.c` that loops forever (well, until you kill it) and every 5 seconds, creates a child process that executes `ls -l` and waits for that child process to finish. You may use any of the class examples as a starting point if you'd like. (5 points)

Signals

We next consider a form of interprocess communication in a Unix system known as *signals*.

Question 14:
Run `kill -l` at on both a Linux and a FreeBSD system to see the list of signals supported by each. What is the output on each system? (1 point)

We can send a signal `SIGNAL` to a process `pid` with the command

```
kill -SIGNAL pid
```

For example, if we launch a program at our Unix prompt to sleep for 60 seconds and put it into the background:

```
-> sleep 60 &
```

you should see output something like:

```
[1] 96132
```

where "96132" would be the process id of the `sleep` process you just created, and `[1]` is the job number within your Unix shell of the process.

We can then send signals to that process by using its pid or `%1` which will refer to job number 1.

For example:

```
-> kill -TERM %1
```

will send the `SIGTERM` signal to try to terminate the process. If you do this, you should see output similar to:

```
[1]+  Terminated                 sleep 60
```

Now launch another `sleep 60` process in the background. Assuming this becomes shell job 1, issue these commands:

```
-> kill -STOP %1
-> kill -CONT %1
```

and wait until the `sleep` command finishes.

Question 15:
What do these do, and what output do you see? (2 points)

Every process has *signal handlers* that are used to respond to signals sent to the process. Basically, it's a function that gets called *asynchronously* when a signal is received.

A *default signal handler* is installed when a process begins.

Two system calls are used to send and catch signals:

`signal()` replaces default handler. This lets you *trap* many signals and handle them appropriately.

Be careful not to confuse this `signal()` *with the* `signal()` *operation on semaphores!*

We consider the example programs in the `signals` directory of `https://github.com/SienaCSISOperatingSystems/sysprog-examples`.

The `sigalrm-example.c` example is compute-bound process that "wakes up" every 5 seconds to report on its progress.

The `setitimer(2)` system call is used to set a "timer" which will cause a `SIGALRM` signal to be sent to the process at some time in the future (in this case, every 5 seconds).

Question 16:
What line sets up the signal handler for `SIGALRM`? What function acts as the signal handler for `SIGALRM`? (1 point)

We can ignore a signal completely by setting its handler to `SIG_IGN`, and restore the default handler with `SIG_DFL`.

Consider this enhanced example: `sigalrm-example2.c`

Question 17:
Which signals are handled by the signal handling function in this example? Which ones are ignored completely? (2 points)

A process can also send signals with `kill()`. Don't let the name fool you, you can send any signal with `kill()`, not just `SIGKILL`.

Note that `SIGTERM`'s handler sends the process a `SIGINT`.

? Question 18:
What happens when you send each of these signals to your running program using the `kill` command from the command line? `SIGALRM`, `SIGINT`, `SIGTERM`, `SIGSTOP`, `SIGCONT`, `SIGUSR1`, and `SIGKILL`. Try each out and paste in your output for each. (3 points)

Final note about signals: `SIGCHLD` will be useful for your shell projects. This gets sent to a process's parent when the process terminates.

Pipes

Processes may wish to send data streams to each other. Unix *pipes* are one way to achieve this. You've almost certainly used Unix pipes at the command line. You can also use them in programs.

An unnamed pipe can be created using the

```
int pipe(int fd[]);
```

system call. `fd` is an array of two `int` values. These are file descriptors, very similar to the file descriptors used for file I/O using `open()`, `read()`, and `write()`.

`fd[0]` is the "read end" and `fd[1]` is the "write end". 0 return means success. -1 means failure.

`read()` and `write()` again operate only on basic streams of bytes – any structure must be added.

We consider the example programs in the `pipes` directory of `https://github.com/SienaCSISOperati`
`sysprog-examples`.

`pipe1.c` is an example of communication between two processes, a parent and its child created by `fork()`, communicating via an unnamed pipe.

? Question 19:
What is the output when you run this program? (1 point)

This required that the values of `fd` are shared between the parent and child processes. This is fine when you create your pipe just before a `fork()`, but what if we have two processes already in existence that wish to communicate through a pipe?

We can create a *named pipe* with `mkfifo` (command or system call).

`pipe2.c` augments our simple example using a named pipe.

? Question 20:
Run this program without creating the pipe `"testpipe"`. What happens? (1 point)

Question 21:
Create the named pipe using `mkfifo`. What is the output of the command `ls -l testpipe` after you do this? (1 point)

Question 22:
Now run the program again with the named pipe in place. What is the output? (1 point)

`pipeprocs.c` is an example that's a little more interesting: two independent processes communicate through a pipe.

Question 23:
Run two instances of this program in two different windows, one to read, one to write. What is the output from each program? Does it matter which order you create the processes? (2 points)

Duplicating file descriptors

We can use the `dup2()` system call to "reroute" input or output from one file descriptor to another file descriptor. This is how your I/O redirection and pipes will work in the shell.

Back in the `exec` example set, see and try `execredir.c`.

Question 24:
If you run the program with a parameter `"outfile"`, what ends up in `outfile`? Why? (1 point)

Note that we don't close the file here and in fact are not given an opportunity to do so since we lose control once the `execlp` call occurs.

We have seen that you can also obtain file descriptors from `open()`, `pipe()`. The fd's at the ends of a pipe can be passed to `dup2()` as well – this will be useful in the shell – set the output of one process to be the input of another through a pipe.

Submission

Commit and push!

Grading

This assignment will be graded out of 40 points.

Feature	Value	Score
Question 1	1	
Question 2	1	
Question 3	1	
Question 4	2	
Question 5	2	
Question 6	1	
Question 7	1	
Question 8	1	
Question 9	1	
Question 10	3	
Question 11	2	
Question 12	1	
Question 13	2	
`execlsloop.c` program	5	
Question 14	1	
Question 15	2	
Question 16	1	
Question 17	2	
Question 18	3	
Question 19	1	
Question 20	1	
Question 21	1	
Question 22	1	
Question 23	2	
Question 24	1	
Total	40	