SIENA*college*
Computer Science

Computer Science 330
Operating Systems
Siena College
Fall 2022

Lab 6: Unix Systems Programming

Due: 11:59 PM, Tuesday, November 8, 2022

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 about them through a lecture, but by trying them out)

In this lab, you will learn and/or review some of the aspects of Unix systems programming that you will need for the upcoming shell project.

You must work individually 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`

Getting Set Up

In Canvas, you will find a 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 documenty committed and pushed to your repository (give the name in the `README.md` file), by writing them in a readable (reasonably nicely formatted, not all one big line of text) GitHub Markdown form in your repository's `README.md` file, or by linking to a shared document containing your answers from your `README.md` file.

Examples related to this lab are in

`https://github.com/SienaCSISOperatingSystems/sysprog-examples`

You should clone a copy of it outside of the repository you create and clone for this lab.

Low-level File Operations

You have used at least some of the C standard file I/O (stdio) 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 formatted data (hence the '`f`' in '`printf`') rather than a low-level stream of bytes.

If we were to look at the implementations of those stdio functions, you would find these low-level operations: `open()`, `close()`, `read()`, `write()`. Our programs can call those directly, if we want.

Error Checking and Reporting

Before we look at those, we look at the standard error reporting mechanism. We have seen this in some of our examples, but have not looked closely at it.

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 variable 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`)

Note that `stdin`, `stdout` and `stderr` are the corresponding `FILE *` pointers that are created

automatically by stdio.

Read through the man pages for these four system calls.

Question 5: In what section of the manual are these found? (1 point)

Now consider this example:

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

Question 6: 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 7: Describe precisely what happens with the `read` and `write` calls in the main loop of this program. What is read or written, where does the data get stored/come from, how does the call know how much data to read or write? (4 points)

Question 8: What is the output of the program when you use the C source code for the program itself as input? (1 point)

Run the program with the C source code for the program as its input and use the filename `eo.txt` for the output. Copy the file `eo.txt` into your repository. Be sure to follow all of the steps needed to have this file in your repository on GitHub. (2 points)

Running a New Program – the `exec` Calls

Recall that the `fork()` system call creates an almost-exact copy of a process – each running the same program and executing at the statement immediately following the `fork()` call.

Question 9: Review: how do you tell if your program is running in the original (parent) process or in the new (child) process? (1 point)

Sometimes this behavior is exactly what you want, but in many cases when you create a new process, you will want to run a different program in that process.

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 (the '`v`' means use an argument "vector")

`execlp()` – like `execl`, but use the search `PATH` to find the program

`execvp()` – like `execv`, but use the search `PATH` to find the program

`execvP()` – like `execv`, but specify a search path to find 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 `exit`s 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.c` program:

- This one uses `execlp`. The parameters are the program to run and its arguments.
- Programs can take any number of parameters, so the `execlp` function needs to be able to take any number of parameters.
- C supports this with *variable length argument lists*, functions to which any number of parameters can be passed – see `stdarg(3)`.
- To write your own such functions, you can use the functions in that man page.
- The list of parameters must end with a `NULL` so the function knows when to stop looking for more parameters.
- Here, we are just calling such a function. The main thing we need to make sure we have that `NULL` terminator for the argument list.

For each program you are asked to run in this part of the lab, run both on noreaster and on your Linux VM. When the output differs, include outputs and/or note any differences in your response, as appropriate.

Question 10: What is the output of the program when you run it? (1 point)

Question 11: 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).

Run the `procinfo` program a few times, giving it first no command-line parameters, then a few different combinations of command-line parameters.

Question 12: Briefly summarize what the output of `procinfo` tells you. (1 point)

Next, take a look at the `execprocinfo` program. It executes `procinfo` by replacing itself in the running process with an instance of `procinfo`.

Question 13: Which line of code accomplishes this program replacement in the process? (1 point)

Run the `execprocinfo` program.

Question 14: What does the output tell us about the value provided in `argv[0]`? (2 points)

Question 15: What happens if you remove the parameter `NULL` from the call to `execlp`? Why? (2 points)

Next, look at `exec2.c`, 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 16: Where are the command-line parameters to the program to be executed with `execlp` specified? (1 point)

Question 17: What happens if you remove the `NULL` from the definition of `myargv`? Why? (1 point)

The program `exec2nonull.c` also "forgets" the `NULL` in the array, then later adds it in (but not immediately, and tries putting it in a couple different places).

Question 18: Run the program `exec2nonull` and explain the results. (4 points)

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

Question 19: Use this program to execute `procinfo`. What command line did you use? What was the output? (2 points)

Question 20: What would happen if we mistakenly use `argv[0]` for both parameters to the `execvp` call? (1 point)

Question 21: Use the program to execute itself 3 times before executing some other program. What command line did you use? What was the output? (3 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. (10 points)

Submission

Commit and push!

Grading

This assignment will be graded out of 45 points.

Feature	Value	Score
Question 1	1	
Question 2	1	
Question 3	1	
Question 4	2	
Question 5	1	
Question 6	2	
Question 7	4	
Question 8	1	
`eo.txt` file	2	
Question 9	1	
Question 10	1	
Question 11	1	
Question 12	1	
Question 13	1	
Question 14	1	
Question 15	2	
Question 16	1	
Question 17	1	
Question 18	4	
Question 19	2	
Question 20	1	
Question 21	3	
`execlsloop.c` program	10	
Total	45	