Lab 2: Processes and System Calls

January 26, 2016

1 Processes and Interfacing with the Operating System

In this lab, how to create new processes and have those processes do something useful will be explored. Do note that the man pages (specifically sections 2 and 3) will be your best friend for this lab. Some familiarity with how to open a file, read from it, and parse what was read from the file will be needed for this lab as well.

2 Creating new Processes

Recall that to create a new child process in a C program, the following lines of code are needed:

```
#include <unistd.h>
.
.
.
pid_t child;
.
.
.
child = fork();
.
.
.
```

This snippet will create a variable named `child` to store the return value of the `fork()` system call which is where all the magic happens. This system call creates a new process as a child of the current process that is a clone of the current process. This means that the child process has the same program, program counter (where in the program the process is), variables, etc. (with some exceptions). Refer to:

```
$ man 2 fork
```

for the full list of exceptions that the parent process has at the time the `fork()` call is made.

Use the fork man page to answer the following questions in your lab report

- What are the return values of `fork()`?
- How can a program determine if it is the parent or child process?

2.1 Make the new Process do Something Different

Since the `fork()` function call creates a clone of the parent process and returns different values for the parent and child processes, using these two bits of information, the child process can be made to do something that is not the same as the parent. Firstly, we can use the fact that the value returned by `fork()` is zero for the child to differentiate between which process is which. The simplest way to do this is to use an `if` statement. Using this method, we would have the following snippet of code in a program to differentiate between parent and child:

```
...
pid_t child;
child = fork();
if(child == 0){
        // Child code here
        // Cool code here
}else if(child > 0){
        // Parent code here
        // boring code here
}else{
        // fork failed, handle error here
        perror("fork");
}
...
```

This snippet will guarantee that the child process will do something cool (because the comment said so), and the parent process will do something boring (again, because the comment states as such).

2.2 Waiting on the Child Process

In the event that the above snippet is run, the parent and child process will each execute at their own pace, and either can finish execution before the other. In the event that this is not the desired behavior (e.g., almost always), there are two system calls that the parent can run that will guarantee that the parent waits for the child process to exit. The two functions are `wait` and `waitpid`. Full information on what each of these functions do and how they are used can be found at:

```
$ man 2 wait
```

To use `wait` to make the parent process wait for the child process to complete, the following snippet of code is used:

```
#include <sys/types.h>
#include <sys/wait.h>
...
int status = 0;
...
child = fork();
if(child == 0){
        //does something cool that takes a while
    return 42;
}else if(child > 0){
        wait(&status);
        printf("child process is done, status is: %d\n", status);
```

```
        return 0;
}else{
        perror("fork");
        exit(-1);
}
```

This snippet will make sure that the parent suspends execution until one of its children terminates. In the event that there are multiple children, and knowledge of a specific child process' termination is of importance, then `waitpid` should be used to tell the parent process to wait. The code snippet to do that is:

```
#include <sys/types.h>
#include <sys/wait.h>
...
int status = 0;
...
child = fork();
if(child == 0){
        //does something cool
    return 42;
}else if(child > 0){
        waitpid(child, &status, 0);
        printf("child process is done, status is: %d\n", status);
        return 0;
}else{
        perror("fork");
        exit(-1);
}
```

This will guarantee that the parent process wait for the child process with the process id stored in `child` to terminate before continuing. Note that, as shown in the man pages for `wait`, to make this snippet of code work like the previous snippet, change the value of the first argument for `waitpid` (the argument that is occupied by the variable `child`) to `-1` and the behavior of this snippet and the previous snippet would be the same. So effectively:

```
wait(&status);        // is the same as waitpid(-1, &status, 0);
```

It is highly recommended that the student read through the man page for `wait` and `waitpid` to fully understand how to use `wait` and `waitpid` and experiment with multiple `fork()` and `wait` programs until fully comfortable with what is actually happening. What is presented here is merely a quick overview of how to use the two.

In your lab report answer the following questions:

- How can you use `waitpid` to determine if a child process has ended without making the parrent wait for that child to end?
- What is the return value of `waitpid`?

3 Making a Process Run Another Program

Recall that to make a process run another program, the following lines of code are needed:

```
#include <unistd.h>
.
.
.
execve(programExecutable, argArray, envArray);
.
.
.
```

With the above snippet, the process should run the program `programExecutable` with the arguments `argArray` and environments `envArray`. For more information about the `exec` family of function calls, how they work, which one to choose (there is more than just `execve`), what causes them to have errors, etc., please consult the man pages at:

```
$ man 3 exec
$ man 2 execve
```

Please answer the following question in your lab report - What is the return value of `execvp`? - What is the difference between `execl` and `execlp`?

4 Tying it Together

With the information presented in the previous sections, this opens up the possibilities for the 'do something cool' part of the child process explored earlier. Suppose that a certain function is desired to be performed by the child process; however, implementing it into the child process would not be possible for some reason or feasible. Suppose further that this specific function has already been implemented by someone else, but it is only available in the form of a binary (i.e., executable) file. The student can choose to either forego performing that specific function, or the student can use the 'exec' family of function calls. Since the former choice is not very interesting, the later choice will be explored. The following snippet of code will do just that:

```
...
int status = 0;
...
pid_t child;
child = fork();
if(child == 0){
        //create and populate argArray here if not somewhere before
        ...
        execv(specificFunction, argArray);
        perror("Child process running specificFunction encountered an error");
        return -1;
}else if(child > 0){
        wait(&status);
        return 0;
}else{
        perror("fork");
        exit(-1);
}
```

In this code snippet, the child process will attempt to run the program `specificFunction` with the arguments `argArray`. If there was an error with the `execv` call, then the `perror` and return lines will be executed.

5 Example programs

Now that you have seen several code snippits it is time for a concreate example. First will be a program that prints out all of its passed in command line argumens and the second will be a program that forks and then spans the child process.

5.1 Command Argument Printer

Examine the `arg-printer.c` file in the labs folder. This program prints out all arguments passed to it on the command line and then returns the number of arguments that it recieved. Compile the file by running the following command:

```
$ gcc -o arg-printer arg-printer.c
```

Test running the program on the command line using different arguments to understand how arguments are passed to main. In your lab report give the output to the following tests. Also, what do you notice about the zeroth element of the `argv` array?

```
$ ./arg-printer
$ ./arg-printer a b c
$ ./arg-printer --version
```

5.2 Fork and Exec example

Next look at the `example.c` file in the lab folder. This file calls the `arg-printer` program and passes in three arguments to it. Compile it and test it with the following commands. Include the output in your lab report.

```
$ gcc -o example example.c
$ ./example
```

6 Tasks for this lab

With information on how to create a child process, and make this child process run a different program, the task for the student for this lab is to create a program that when given the path to a file, it would open the file with the appropriate software to view the contents of the file (e.g., Microsoft `.doc` files and its derivatives, or Open/LibreOffice `.odt` files should be opened with LibreOffice). The specification of the program is as follows:

1. The program should take one argument, and that argument should be a file that is not a compiled executable.
2. If there is no argument, or the file that is passed in does not have a program that can open it for viewing, a usage message should be printed and the program exits.
3. In the event that an error occurred, there should be an error message associated with it (so use `perror`), and the program should exit; there should be no segmentation faults.
4. The program will only have to handle files of the following types and open with the mentioned programs:
5. x.doc -> libreoffice
6. x.odt -> libreoffice
7. x.png -> eog
8. x.txt -> gedit

9. x.pdf -> evince
10. x.mp3 -> vlc
11. The parent process should wait on the child process(the process that opens the file with the correct software) to terminate and print out the termination status.
12. **Extra credit** will be given for adding additional file types through a runtime configuration file.

13. **Extra credit** will be given for opening multiple files at the same time and printing the exit status of each program separately.

Example usage:

```
$ anyopen lab2.pdf
```

7 License

This lab write up and all accompany materials are distributed under the MIT License. For more information, read the accompaning LICENSE file distributed with the source code.