Lab 2 Report

Deeksha Juneja

CprE 308

2/2/2016

Purpose

The purpose of this lab is to understand processes, creating of child processes, and executing another program through a child process.

Creating New Processes

In this section I learnt the creation of a new child process. This can be accomplished with pid_t child.

It is interesting to note that, on creating of a child process and the execution of it, the fork() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created.

Moreover, the parent process calls the fork() system call, and the operating system figures out how to create a child process that has an identical structure. This is OS dependent, but we can assume that any OS has a way of keeping track of which processes are running. Since system calls are capable of returning values just like a function, the system call returns the child PID to the parent processes, and 0 to the child process. It knows which value to return to which process, via the internal process bookkeeping.

It is actually pretty interesting to see new process being created using fork(). A new process called the child process is completely identical to parent process but can be made to do different things. Moreover, we generally want the parent to wait for the child process. Hence wait(&status) comes in handy for that. The parent suspends execution until the execution of the child is complete. Waitpid() can be used in case we want the parent to wait for a particular child as a parent might have multiple child processes.

The waitpid() function lets the calling process obtain status information about one of its child processes. If status information for two or more child processes, the order in which their status is reported is unspecified. If more than one thread is suspended in waitpid() awaiting the termination of the same process, exactly one thread returns the process status at the time of the target child process termination. The other threads return -1, which errno set to ECHILD.

We can determine if a child process has ended without waiting making the parent to wait for the child process to end by using the variable "status" as the second argument of the function waitpid(). Also WNOHANG can be used for determining the status of the process.

If waitpid() was invoked with WNOHANG set in *options*, and there are children specified by *pid* for which status is not available, waitpid() returns 0. If WNOHANG was not set, waitpid() returns the process ID of a child when the status of that child is available. Otherwise, it returns -1 and sets errno to one of the following values:

ECHILD

> The process or process group specified by *pid* does not exist or is not a child of the calling process.

EFAULT

> *stat_loc* is not a writable address.

EINTR

> The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

EINVAL

> The *options* argument is not valid.

ENOSYS

pid specifies a process group (0 or less than -1), which is not currently supported.

Exec functions

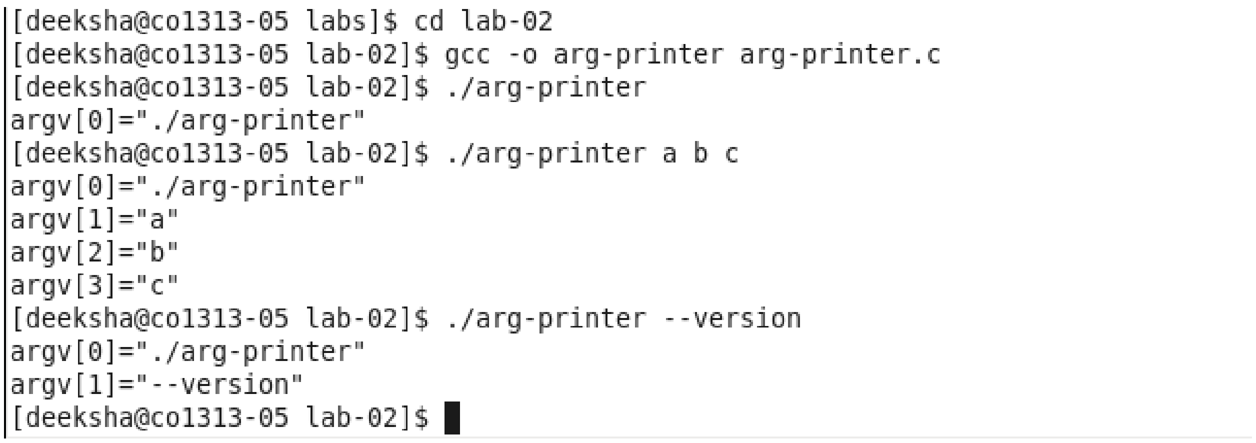
This is a family of functions which allow the process to run a different program. It is important to note that the functions of exec family including execvp() return a value only when an error has occurred. The returned value is -1 and the errno is set to indicate the error.

There are many functions in exec family. Two of them are execl() and execlp(). The difference is that the execlp() uses the environment path variable to search for the executable file named to execute. But execl() requires an absolute or relative file path to be prepended to the filename of the executable if it isnot in the current working directory.

Example Programs

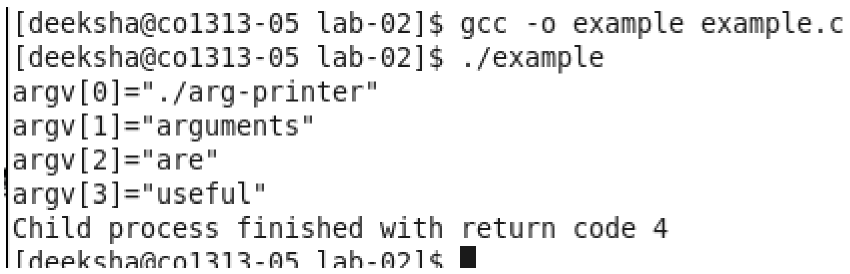
In the first example program, I learnt that the first element of the argv array points to the name of the program. Hence argv[0] points to the ”program name” string.

The output of the sample code 1 can be found here:



```
[deeksha@co1313-05 labs]$ cd lab-02
[deeksha@co1313-05 lab-02]$ gcc -o arg-printer arg-printer.c
[deeksha@co1313-05 lab-02]$ ./arg-printer
argv[0]="./arg-printer"
[deeksha@co1313-05 lab-02]$ ./arg-printer a b c
argv[0]="./arg-printer"
argv[1]="a"
argv[2]="b"
argv[3]="c"
[deeksha@co1313-05 lab-02]$ ./arg-printer --version
argv[0]="./arg-printer"
argv[1]="--version"
[deeksha@co1313-05 lab-02]$ █
```

The output of the same code 2 can be found here:


```
[deeksha@co1313-05 lab-02]$ gcc -o example example.c
[deeksha@co1313-05 lab-02]$ ./example
argv[0]="./arg-printer"
argv[1]="arguments"
argv[2]="are"
argv[3]="useful"
Child process finished with return code 4
[deeksha@co1313-05 lab-02]$
```

Finalprog.c

I used the examples provided as a skeleton for my code. I was limited by my knowledge of C and hence, I wasn't able to complete the full extra credit part. Though, I was able to complete one of the extra credits which required opening multiple files.

I made two char arrays and hard coded them with the extensions and the corresponding programs in the same index. After that I made a for loop inside which all my forks were being created. I am creating multiple child processes because I wanted to open multiple files.

Then, I made a nested for loop and what that helped me do was go over all the possible file extensions and validate if the extension inputed by the user is correct or not. Moreover, to make the code a little modular, I am using a helper function called ends_with to make sure that the file opens with the correct program.

Given that I have never used exec functions before, it took me a while to figure out execlp function because I was trying to use execvp and that was a little buggy. Execvp was just opening all the files with the same program which was not something I wanted. Hence, I used execlp.

The extra credit part was harder to implement. I did not have all my code in the upper for loop which I made later when I decided to the extra credit. But the mistake I made was that, I tried to make a lot of smaller loops instead of making one big loop. Later, I realized that wont work and hence made the big loop around everything.

My Learning

Overall, I can say that I have learnt a lot from this lab. Given that I am not taking the class but just the lab, it is a little harder for me figure out everything. But I am enjoying the process of it. I especially liked the exec functions and thought they were very useful. This lab has actually made me see the usefulness of C because I am mainly a java programmer. I tried a lot to implement the extra credit part 2. I tried various different methods with execvp and execlp. I also tried to use for loops and arrays but in the end I realized that I need to make multiple child processes. Hence, I made multiple child processes and put everything in a giant for loop. It is probably not the most effective method but it seems to work well.

Outcome

My program creates multiple child processes and opens multiple files via the input from command line argument. It opens all the 6 different required file types as well. The file names that can be run are abc.png, abc.mp3, abc.txt, abc.doc, abc.odt, abc.pdf. The user can put in any file though.