Lab2 Project Report

CS370 – Operating Systems

Professor Zhu

Author: Joshua Summers

Table of Contents

Introduction

This program will create ten child processes to handle one of the ten commands and their arguments. This will allow for testing child processes running inside a parent process and evaluating how to check their status. Output from the child and parent processes will be sent to the terminal (stdout).

Implementation Summary

This program was implemented in C code. The program leverages the following shared libraries:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
```

These libraries give us the ability to call system libraries to get access to hardware resources and the exec(), fork(), and wait() libraries. Using these commands, we can create child processes within a parent process. Using a for loop, we can manage the child process creation, and along with a while loop, we can manage the parent in a wait status.

A two-dimensional character pointer array stores the command and the arguments to be sent using execvp(); this makes sending the pointer to the execvp() function simple. A for loop is used to iterate for the number of child processes to create; inside each loop, a new child is made with the fork() function. If the resulting fork function process ID (PID) is non-negative, then no error was received; otherwise, handle the error.

Once we have created the new child process, we can load it with commands using the execvp() function. We pass the name of the execution program, and then the command and argument pointer array. We then check to see if any errors were found with the running of execvp, and handle them; otherwise, we report success and continue with all child processes. Once all the child processes have started, the parent process gets into a while loop and stays there until all the child processes are complete. The parent process will report that it is complete once completion notification from each child is received.

Results and Observations

There are many command differences between exec() calls and terminal calls. For example, the following commands are not available as a system command:

- history
- export
- pipe command and redirection
- globbing (wildcard expansion)

To use these, you must either manually implement your logic or handle them separately with other system commands—as with pipe commands or redirection. This requires more thinking when designing your program and parsing output: some commands are better done with the system() function instead of exec() functions; alternatively, you could do most of this parsing directly in your program.

There was no communication between the children in this program: only tracking on behalf of the parent is done. This simplifies the communication structure as no Inter-Process Communication (IPC) is required; the child processes run independently, and only the parent process waits and tracks the status of the children. Once the status of the children left waiting is zero, the parent can continue, but in this case, it simply exits as all the children are done, and that was its only task.

Conclusions

System calls are very different than terminal interactions and should be limited to mostly direct usage while abstracting any logic into your code. Parsing and analysis are made easier in your program since all the tools of the bash terminal are not present in the system Application Programming Interface (API).

Once the child processes are created, there are a couple of ways to manage them based on the desired communication between processes, but in our case, we relied simply on wait() since we have no care of the order in which they are completed. It is important to understand that without any communication between the children or the child and the parent, the order in which the children run will be pseudo-random: the kernel will analyze the processes and make scheduling decisions based on that analysis.

Error handling in C is very different than other languages as it lacks a try-and-catch structure; instead, it has the perror(), assert(), and exit() structure for most basic use cases. With child processes, it is easy to detect the error as the process ID (PID) will be a negative number—typically a negative one. It is important to note that some system calls have non-fatal errors that do not cause an unhandled exception and will be output to the terminal.