Lab 2 Report

1. Introduction

The goal of this project is to explore processes and their management through a C program that creates a parent process and fifteen child processes under it. Each of the processes execute a different Bash command. The program also records the number of processes that exit normally, normally with a nonzero exit code, and signal terminated to abort. Processes are critical to operating systems because they allow the system to direct the computer's hardware resources, avoid conflicting processes, and to track the processes and their status.

2. Implementation Summary

The program contains an array of possible commands, counters for exit status and signal terminations, and constant for the maximum number of children processes (fifteen) as its main variables that are repeatedly used. The program first creates a parent process, then it proceeds with creating the fifteen children processes. The first four processes are used to demonstrate specific commands, failed commands, and abnormal exits. The remaining commands are random Bash commands from the array of possible commands. The program increments the exit code counters and presents those counters in a summary. The fork() function is used to duplicate the current process which starts with the parent process. The execvp() function is used to place the memory the child copied from the duplicate created by fork() with the program for the Bash command. The waitpid() function tells the parent process to wait until a child has finished its process, it also collects status info like the exit code when a child process ends.

3. Results and Observations

When fork() duplicates the parent process with a system call, the memory state of the parent is copied to the duplicate. The execvp() is used to replace that memory state with the memory required to execute the specified Bash command that is found through the system's PATH. The process IDs are managed through an array of the children PIDs that the parent process can track. The creation order for processes is in the same order as fork() is called by the parent process, meaning they will be in chronological order. However, termination order is based on the order in which processes finish their tasks, which can be variable depending on the task they were doing. The waitpid() function requests the status for how the process was terminated from the kernel which reports the exit code. A normal exit occurs if the task was completed successfully. A signal termination occurs when a fatal error happens in the process or it is terminated externally (e.g. the "kill" command is used in the terminal).

4. Conclusion

This project demonstrated how to use system calls in C to create a parent process, multiple children, replace the duplicated memory in the children to execute commands, and how to track exit codes reported by the kernel. These functions show how process management is done within a Unix based operating system.