Diego Trevino
2/3/26
CS 470
Lab 2 Documentation

Introduction

Process management is an important part of an operating system because it controls how programs are created, runned, and finished. The operating system needs to be able to start new processes, allow them to execute programs, and keep track of when they end. This is necessary for multitasking, sharing system resources, and making sure the system runs correctly.

Implementation Summary

In this program, the main process acts as the parent and creates 15 child processes using fork() inside a loop. Each time a child process is created, the parent stores the child's PID in an array in the same order they were created. Inside each child process, the program prints the child index, its PID, and the command it is about to execute. The child then uses execvp() to replace itself with a Linux command such as ls, date, or echo. Two child processes are set up to run invalid commands so that execvp() fails and exits with a non-zero exit code. Two other child processes use abort() to intentionally terminate by a signal. After all child processes are created, the parent waits for them to finish using waitpid() in the same order they were created.

Results and Observations

Although the child processes are created in a specific order, they do not always finish in that same order. Some commands execute very quickly, while others take longer, which causes differences in completion order. Even when child processes finish out of order, the parent process still waits for them in creation order because it calls waitpid() using the stored PID array from the first child to the last.

To check how each child process ended, the parent examines the status value returned by waitpid(). If WIFEXITED(status) is true, the child exited normally and the exit code is obtained using WEXITSTATUS(status). If WIFSIGNALED(status) is true, the child was terminated by a signal and the signal number is obtained using WTERMSIG(status). In the program output, the invalid commands result in a non-zero exit code, while the processes that call abort() are terminated by a signal.

Conclusion

This project helped me better understand how Unix process management works using fork(), execvp(), and waitpid(). I learned how new processes are created, how a process can be replaced with a different program, and how a parent process can wait for and interpret the results of child processes. The project also showed the difference between process creation order and process termination order, which is an important concept when working with multiple processes.