Code Report Lab 2 CS 470

Ben Howell-Burke

Introduction

Process management is a key concept in operating systems as it governs how processes are created, executed, and synchronized. This lab's objective was to simulate basic process management in a Unix-like environment using C. It focuses on the use of three system calls: fork(), exec(), and wait(). These functions are used as the backbone of multitasking in Unix and are critical for creating child processes, executing different tasks, and managing the termination and synchronization of those processes. Through this lab we will be looking at the lifecycle of processes, from creation to execution and final reporting by the parent process.

Implementation Summary

The project is implemented as a C program named Lab2_CS270.c, which creates 10 unique child processes using fork(). Each child process executes a different task using the execvp() system call. The commands executed include common Unix commands such as ls, echo, pwd, date, and ps. One command includes the requested use of echo "Hello Benjamin".

The implementation of this code is centered around a loop that:

1. Creates a new child process with fork().
2. In the child, executes a command using execvp().
3. In the parent, stores the PID for synchronization.

After all children are spawned, the parent waits for each using waitpid() and reports whether the process exited normally or was terminated by a signal. Proper error handling is included for fork(), execvp(), and waitpid().

A Makefile is provided for easy compilation using make.

Results and Observations

A. Process Creation and Management

The program successfully demonstrates the creation of multiple concurrent child processes. Each child process is independent and inherits the parent's environment. The use of execvp() replaces the child's memory with a new program, ensuring efficient resource use and separation of tasks.

All commands were executed as expected:

- echo printed strings, including a greeting with my name.
- ls, ps, pwd, and whoami ran and produced system-level output.
- All children terminated cleanly and reported an exit status of 0, indicating success.

B. Parent and Child Interaction

The parent process does not execute any child commands but controls the flow by:

- Creating each child.
- Waiting for each to complete.
- Printing status information, including PID and how the child exited.

This pattern reflects real-world parent-child synchronization in operating systems. The parent waits on each child to prevent zombie processes and logs the result for verification.

Conclusion

This lab effectively shows basic process management using fundamental Unix system calls. It shows how parent and child processes interact and how execution flow is managed in C. By using fork(), execvp(), and waitpid(), we replicated a simple multi-process environment where tasks are distributed and monitore.

This test shows how the operating system uses process to handle multi-tasking and the importance of synchronization and error handling in such systems. The project can be extended further by adding more inter-process communication or by looking at execution time for each process.