CS 470 Lab 2 Documentation

Project Goal: The goal of Lab 2 was to simulate process management with a parent process and 10 child processes, each performing a unique task while utilizing C system calls such as fork(), execvp(), and wait(). Additionally, a Makefile was created to facilitate compilation and execution.

Theoretical Knowledge: A process is an active instance of a program, and new processes are created using fork(). These new processes can replace their execution context using execvp(), allowing them to run different commands or programs. Synchronization between processes is achieved with wait(), ensuring that the parent process properly manages child execution. This lab provided hands-on experience in managing concurrency and handling process execution effectively.

Implementation Summary: The main function of the program initializes global variables for process IDs and status codes. A loop is used to create 10 child processes via fork(), with basic error handling to ensure proper process creation. Within each child process, a switch statement determines which command to execute using execvp(), such as echo, ls, date, and pwd. If execvp() fails, an error is displayed using perror(), and the process exits. The parent process waits for each child to complete using wait(), collecting their termination status.

Results and Observations:

- Each child process successfully executed its designated task.
- The parent process correctly managed process execution and termination.
- Error handling ensured that any failures in fork() or execvp() were reported.
- Process execution order was not strictly sequential due to concurrent execution.

Conclusion: This lab reinforced the importance of process management in Unix-like systems. Understanding how fork(), execvp(), and wait() interact provided deeper insight into process coordination and execution control. Future improvements could include additional synchronization mechanisms to better manage execution order and avoid unintended behavior due to concurrency.