Matthew C. Hansen

4 / 21 / 2025

CS470 – Shangyue Zhu

Lab 2 – Report

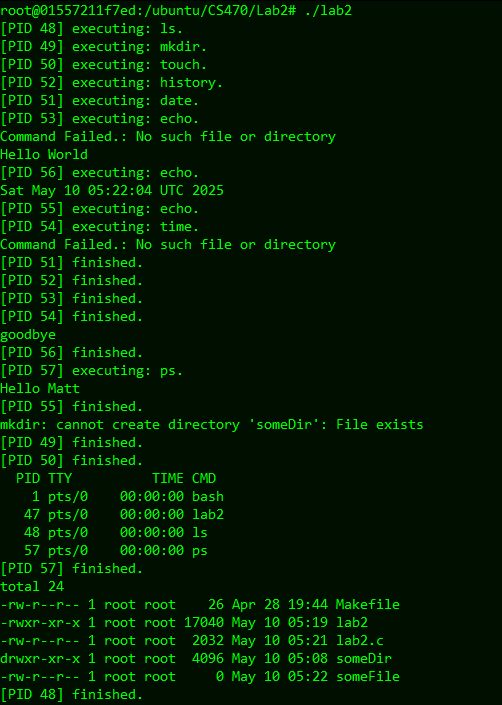
Lab 2 Report

**GitHub Link**: <https://github.com/ArwenTC/cs480_lab2>

# Introduction

Lab 2 was focused on forking processes. The main goal was to create 10 forked processes, each running a different Linux command, using the exec function family to replace the current process. To implement this, I mostly implemented the program the same way that was shown in class by Professor Zhu.

# Implementation

To implement this lab, the program uses a list of Linux command stored in an array. We then enter a for loop which forks the program into new process. Upon a successful fork, the program uses execvp to replace the current process with a new process (running a Linux command and its arguments). If the fork is not successful, or the execvp command returns an error, we exit the current execution, so the for loop is exited and the current process is terminated. If we were to continue the loop, and fork as the child process, processes would begin to multiply. To summarize, the program enters a loop, creates a child process that runs the current Linux command in the command list, and exits upon completion (exiting is typically handled inside the execvp command, unless the program runs into an error). The parent process itself never runs any of the Linux commands, but rather acts as a dispatch for a series of child processes, and lets those processes run in the background. After creating all 10 processes, the parent program then waits, given an exit status variable for each process to finish, until wait() returns an exit status of zero, meaning there are no more child processes.

# Results and Observations

To explain how the process forking works, we can view the results of the program. The results show that all the processes are created, and all the processes are finished. When an execvp command fails “Command Failed” is printed using perror, and the error is shown. We can see since process forks, each process runs concurrently, so some commands are executing while the parent process is still dispatching processes. Once all the processes are created, the parent process moves to waiting for the processes to finish and reports each process as it ends.

To sum this up, the parent process dispatches a series of child processes which go off and execute Linux commands concurrently on their own time. Once the parent finishes the for-loop that dispatches processes, it waits for the processes to report back that they have finished, and prints out which process finishes each time it enters the while loop, until it exits the while loop when there are no processes left to finish. We can think of it like a temporary task force sending out agents to do various missions. The task force has a list of missions that it needs to get done, in this case the set of commands, and an agent for every mission. The task force dispatches / delegates missions to agents, and waits for them to finish their missions. Once the missions of the agents are completed, the task force is resolved, since all the work it set out to do is finished.

# Conclusion

Fork() when used correctly, creates a series of processes that can run various tasks concurrently, and this lab is an example of how this can be used to do so. Communication between processes is not entirely necessary, however in most cases, some communication of when a task has been completed may be helpful for program execution.