Process Management Simulation Report

Shangyue Zhu

Jaewon Heo

CS 470

Central Washington University

Contents

[Introduction 3](#_Toc197369215)

[Implementation Summary 4](#_Toc197369216)

[Results and Observations 5](#_Toc197369217)

[Conclusion 6](#_Toc197369218)

[Appendix 1 7](#_Toc197369219)

[Figure 1 creating child processes 4](#_Toc197368446)

[Figure 2 parent process waits 7](#_Toc197368447)

[Figure 3 program output 7](#_Toc197368448)

[Figure 4 command order 8](#_Toc197368449)

# Introduction

This program simulated and demonstrated the process creation and termination in Linux using the C language. The objective of this simulation was to create 10 unique child processes. Each child executed a different UNIX command to demonstrate uniqueness. This exercise was held by Dr Shangyue Zhu in his CS 470 as Lab assignment 2. After finishing this lab exercise, we gained knowledge of the process scheduling and parent-child relationship.

# Implementation Summary

This program was designed to create 10 child processes by using fork(). Each process executed Linux command which stored in a 2D array of char\*, which were fed into each child process’s execvp() system call sequentially.

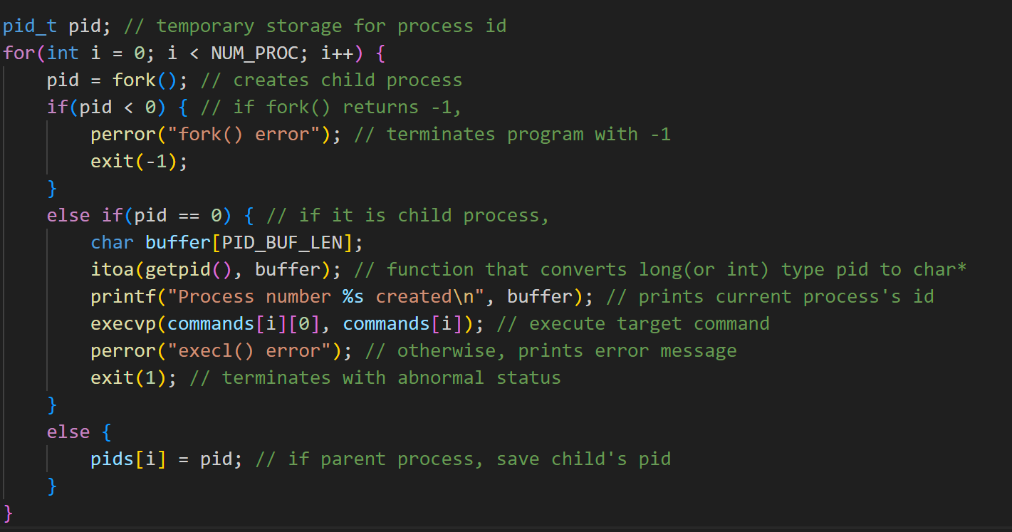
 The core logic of this program is iteration. In each iteration, the process will call fork() system call to create its child processes. Then it checks returned value to control its flow. If the fork system call returned negative integer, then the process will print error messages and be terminated. If returned value is 0, which means it is a child process created by fork(), then the process will print its process identifier and will execute Linux command. Lastly, If the fork system call returns positive integer, then the process will save the returned integer to a predefined array. The reason for that, the positive integer returned by fork() is process identifier of its child process. Thus, we can conclude that the process caught in this condition is the parent process.

Figure 1 creating child processes

After the fork procedures, the parent process will iterate over the array which stores its child processes’ identifier to wait until its child processes to be terminated (figure 2, appendix 1).

# Results and Observations

The output of the program can be found in figure 3 in Appendix 1. First, we can see that child processes are successfully created. Here, the initial assumption was that child processes would process its task right away. However, the processes were first stored into a ready queue and waited until scheduler selects, Linux commands were not executed until nine child processes were created.

When the first Linux command which was stored in the 2D array (figure 4, Appendix 1) executed, the last child process was created. Then the second child process executed its assigned command. Here, after execution of second child process, the first two child processes were terminated right away, and the rest of child processes were still in the ready queue.

Then, the rest of the tasks were proceeded. However, the execution order was different than expected. After termination of PID 46059, execution of the Linux commands was not ordered as order of the 2D array. The order of process execution was shown, date, head, id, tail (small closing bracket), uptime, and ip. Also, the mkdir and touch command were executed in between above commands.

# Conclusion

In conclusion, this program successfully demonstrated the creation and termination of child processes in Linux operating system using fork system call and exec system call. The result showed that child processes were not executed sequentially due to the scheduler. The first and second child process started their tasks before even every child process was created. Also, the execution of the rest of child processes were not in order as they were listed in the 2D array. This shows that the scheduler would not guarantee the execution order and the process’s priority was determined by several factors such as available resource size, and I/O operations.

# Appendix 1

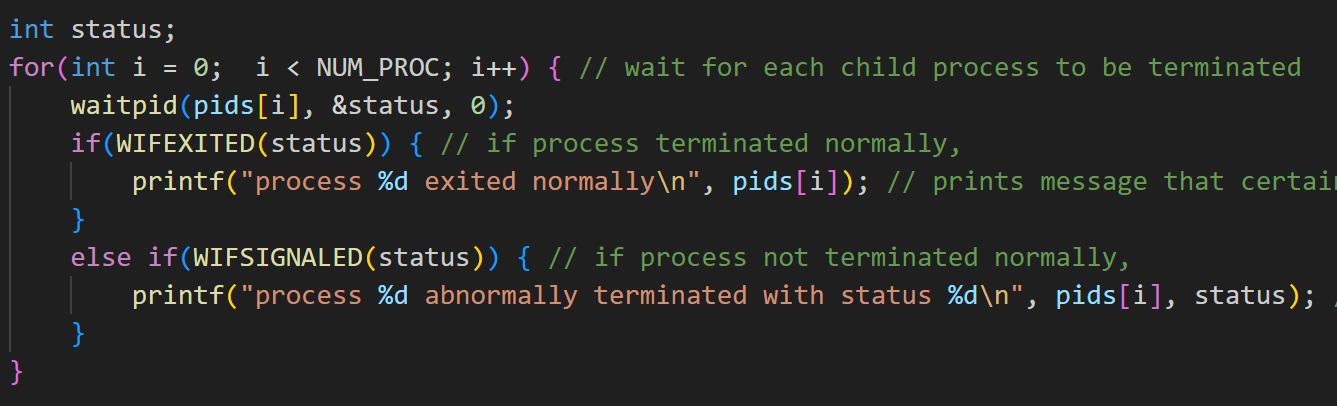


Figure 2 parent process waits

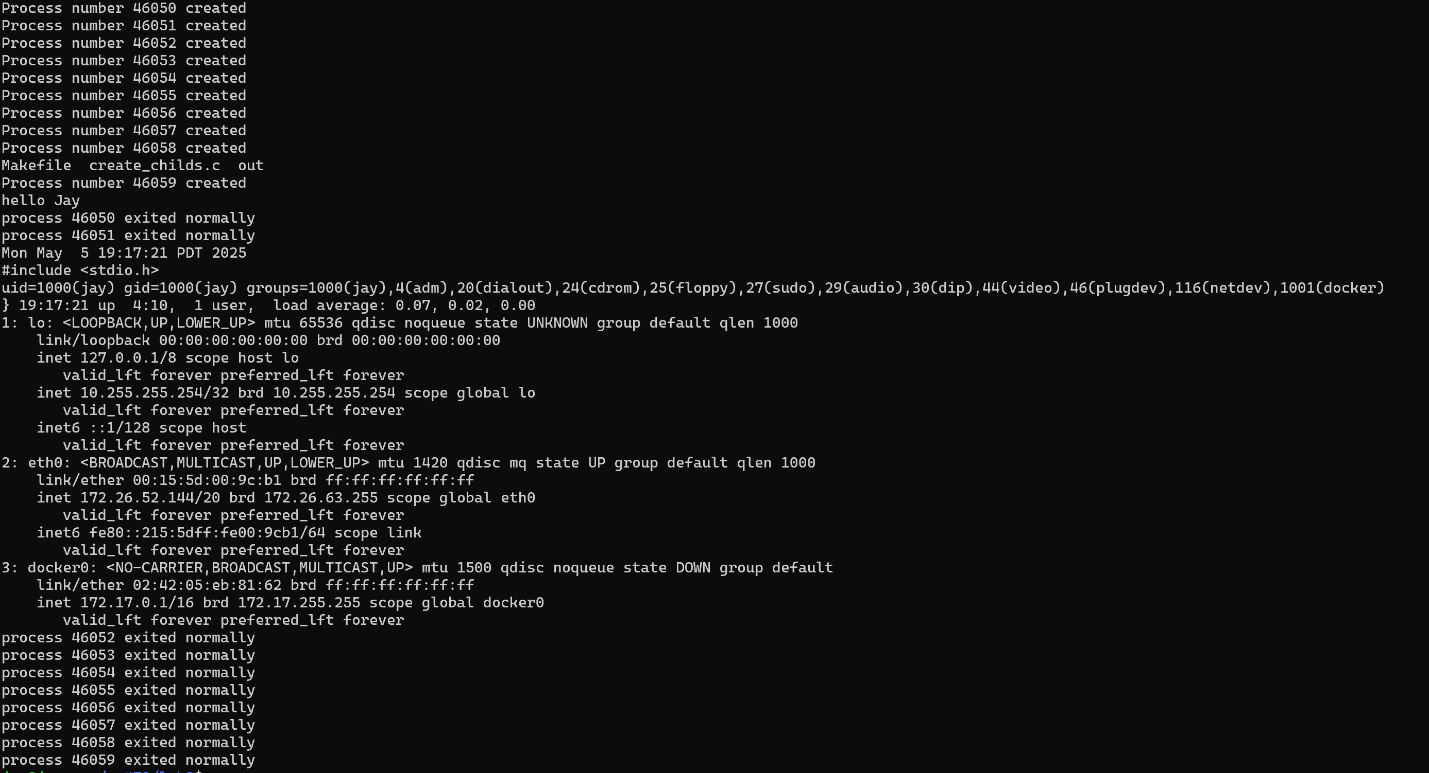


Figure 3 program output

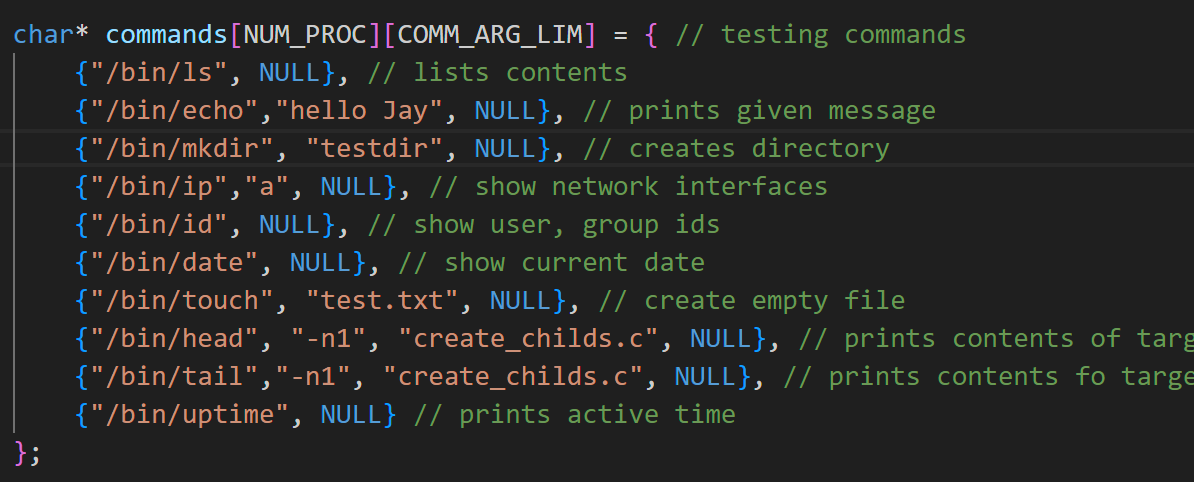


Figure 4 command order