Assignment 2: Part 3 Report

SYSC 4001

November 07, 2025

Kevin Abeykoon (101301971)

Jesse Handa (101264747)

Github: https://github.com/KevinAbeykoonUniversityAccount/SYSC4001_A2_P3.git

Introduction

This report looks at how an API simulator (for an operating system) behaves when modeling things like process execution, memory allocation, and the fork and exec system calls. Using the provided code template, the simulator reads a program trace line by line and keeps track of changes in state, such as stack operations, memory updates, and system call behavior.

The main part of this analysis focuses on three required simulation tests: one for simple sequential program execution, one that uses fork to show process creation, and another that uses exec to replace a process's memory image. I also created two extra custom tests to see how the simulator performs in much longer trace files and with more forks and execs. Throughout the report, I discuss the log outputs and expected results to show how the different parts of the simulator work together. The code is based on the template provided on GitHub.

Implementation Overview

The major focus point of the simulator is the simulate_trace() function. This function recursively reads and "processes" instructions from a program. By simulating an API Simulator, we follow the major steps the OS goes through when processing a program. The steps are printed into a file called execution.txt. The system status is printed into a file called system_status.txt. Most of the code was provided in the template. The important items we programmed were the fork and exec system calls, which I'll go into more depth in the next two sections

FORK Mechanism

The fork instruction is implemented according to steps we learned in class. But first, when fork is encountered, the simulator duplicates the current process state, including its: The major steps it took were:

- Entering kernel and saving context
- Look up address in vector table and load the ISR
- Clone the PCB
    - Same program name and size
    - But gets new PID
- Call the scheduler

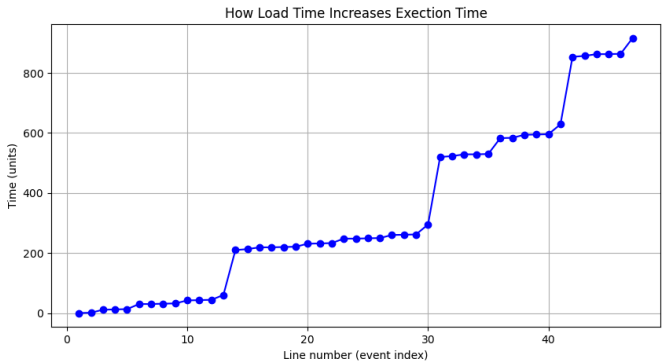
A new process control block (PCB) is created for the child using the parent's PCB and inserted into the waiting queue. Before it is inserted however, the system allocates memory for the child process.

EXEC Mechanism

Unlike the fork call, exec doesn't create a new process, it completely replaces the current one with a new program. When exec runs:

- The process's old address space is wiped out.
- The new program is loaded into memory.
- The stack and program counter are reset to start the new program.
- The process keeps the same PID (process ID)

Basically, exec makes the process "become" the new program. This is how real operating systems work. In an execution log, this looks like a sudden switch: the old program's instructions stop right away, and the new program starts running from its entry point. As the assignment instructions suggest, I made the loading time the size of the program times 15. This looks like a sudden spike, as seen in the graph below. It shows that larger files can increase the execution very quickly.



Role of the break in EXEC

The break statement in the exec if-case is really important for making sure the program works correctly. When exec runs, it should never go back to the old code, meaning continuing to run those old instructions would mess up both the simulator's logic and how a real operating system behaves. If the break wasn't there:

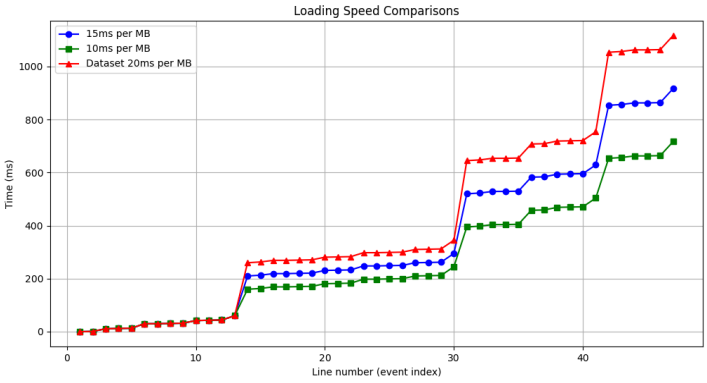
- The simulator would keep running instructions from the old program.
- The log output might get mixed up between two different programs.
- From tests, I found the partition assignments started failing
- Any child processes created before EXEC could end up with broken or inconsistent states.

By stopping the old execution path right away, the break makes sure the simulator switches cleanly to the new program and keeps the simulated memory space consistent. I actually tested this by deleting the break to see how it changes the output. The output showed that the code returned to the parent code and started to repeat itself.

Test Scenarios and Observed Behavior

Across the three required and two additional tests, a few clear patterns showed up in the simulator's behavior. When the fork instruction was introduced, the simulator created a parent that waited and a child process that continued running, which showed that the PCB was being duplicated properly. This follows what we learned about the fork function in class. Exec behaved differently, it stopped the current line of execution right away and switched over to the new program image. The logs made this obvious because the instruction stream suddenly changed.

As explored in the previous , as the number of items in the trace file increases, the output logs dramatically increase. It also increases heavily when calling sub programs (in this assignment they were called program1.txt and program2.txt). I also tested how the execution times differed when I altered the load speed. The instructions suggest setting it to 15ms per MB. I tested this by altering it to 10 and 20. The results showed that it almost follows an exponential increase as the gap starts out small and gradually grows much larger.



Conclusion

By testing both required and custom scenarios, the simulator shows that it can accurately and realistically model how processes run, how memory is managed, and how system calls work. The fork function works correctly by duplicating the PCB and submitting the child into the execution queue. The exec expectantly replaces the entire process and immediately starts running a new program.

The break in the EXEC handler turns out to be really important, it stops the program from continuing incorrectly after execution changes, keeping the simulator accurate and similar to how real operating systems behave.

We also ran additional tests to change the trace length and load speed. These tests showed us even a little change in these amounts caused a large change in the execution speeds. Overall, these tests and results show that the simulator accurately represents key OS concepts and serves as a valuable learning tool for understanding how processes are created, managed, and executed.