Assignment 2 - Report

SYSC 4001A (L3)
Student 1: Eshal Kashif (101297950)
Student 2: Emma Wong (101297761)

Part III - Design and Implementation of an API Simulator: fork/exec

GitHub Repo: https://github.com/EmmaWong8/SYSC4001_A2_P3

Introduction

This simulator models the behavior of fork() and exec() using interrupt-driven logic. Each system call triggers its respective ISR (vector 2 = fork, vector 3 = exec), while random delays (1–10 ms) emulate kernel activities, and loader time equals 15 ms × program size (MB). The logs execution.txt and system_status.txt record all events and PCB/memory snapshots. All tests demonstrate correct PCB creation, scheduling (child-first policy), and process image replacement.

Test Scenario 1 – Fork + Dual Exec (init → program1 → program2)

Description: The test combines fork() and exec(). The initial process (init) calls fork(), and the resulting child and parent processes each execute a different external program. The child branch (IF_CHILD) calls EXEC program1, while the parent branch (IF_PARENT) calls EXEC program2. This test checks that the simulator correctly clones the PCB on fork(), prioritizes the child, and performs two separate exec() operations (one in the child and one in the parent).

Execution Analysis:
At time 0 the FORK ISR (vector 2) clones the PCB and schedules the child.
At 24 ms, the child executes EXEC program1; the log shows vector 3, a 10 MB size lookup, 150 ms load time, partition marking, and PCB update.
After its CPU burst, the parent resumes and runs EXEC program2, loading 15 MB (225 ms).
The final SYSCALL (vector 4) confirms normal ISR handling post-exec.

System Status: Logs show the child running (program1) while the parent waits, then the parent switching to (program2). This validates correct process duplication, scheduling, and image replacement.

Test Scenario 2 – Nested Fork and Exec

Description: This test examines nested process creation and multiple exec() calls. The initial process (init) calls fork(), creating a child that executes EXEC program3. Inside program3, another fork() occurs, producing a new child (PID 2). Both the parent and child inside program3 later execute EXEC program4. This scenario tests correct handling of nested forks, child-first scheduling, and independent program replacement for each process.

The first FORK ISR clones init's PCB, assigns a new PID, and schedules the child first. The child executes EXEC program3 (10 MB → 150 ms load, PCB update). Inside program3, another FORK creates a third process; the new child runs first, executing EXEC program4 (15 MB → 225 ms) and its CPU burst. The parent then runs its own EXEC program4, and finally, the original init resumes to finish its CPU burst. The logs confirm correct recursive scheduling, with each child completing before its parent resumes.

System Status: Snapshots show three PCBs after the second fork, with distinct partitions and states. The child of program3 becomes program4 first, followed by the parent doing the same, confirming proper process cloning, execution order, and image replacement.

Test Scenario 3 – Fork and Exec with I/O Operations

Description: This test combines process creation and program replacement with I/O handling. The init process calls fork(), creating a child that runs first, while the parent later executes EXEC program5. program5 includes CPU bursts, a SYSCALL, and an END_IO event, testing how the simulator handles I/O interrupts alongside exec.

Execution Analysis: The FORK ISR (vector 2) clones the PCB and schedules the child first, placing the parent in the wait queue. After the child completes, the parent executes EXEC program5, triggering vector 3 for size lookup (10 MB), loader time (150 ms), and PCB update before returning to user mode. The log then shows SYSCALL and END_IO ISRs, where the simulator switches to kernel mode, saves context, performs the I/O operation, and updates the ready queue before resuming normal execution.

System Status: The first snapshot shows both parent and child after the fork, confirming proper PCB cloning and scheduling. The second snapshot shows only one active PCB (program5), indicating the exec successfully replaced the parent's image. The presence of SYSCALL and END_IO confirms that normal interrupt-driven I/O functions correctly with exec transitions.

Test Scenario 4 – Sequential Exec Calls (Program6 → Program7)

Description: This test evaluates how multiple sequential exec() calls are handled in a single process. The init process performs a short CPU burst, executes EXEC program6, and then program6 immediately executes EXEC program7. This scenario demonstrates how the simulator replaces a process image multiple times while maintaining the same PID, confirming that no new processes are created during repeated exec calls.

Execution Analysis: The log begins with a 20 ms CPU burst, followed by the EXEC ISR (vector 3) for program6. The simulator looks up the program size (10 MB), loads it into memory for 150 ms, updates the PCB, and returns to user mode. program6 then executes a CPU burst and

calls another EXEC program7, where the same ISR sequence repeats (15 MB → 225 ms loader). The consistent use of vector 3 and sequential IRET lines shows that the same process repeatedly re-enters kernel mode, replacing its memory image each time.

System Status: After each exec, the PCB shows the same PID but updated program name and partition number, confirming that exec() reuses the same process slot instead of spawning a new one. This validates proper process image replacement, memory reallocation, and state transitions across consecutive exec calls.

Test Scenario 5 – Simple Fork

Description: This test isolates fork() behavior. The init process runs briefly, calls fork(), and then the child and parent execute different CPU bursts under IF_CHILD and IF_PARENT. It verifies correct PCB cloning, child-first scheduling, and independent execution of both branches.

Execution Analysis: The FORK ISR (vector 2) logs kernel entry, PCB cloning, and scheduler activation. The child runs first, executing its 30 ms CPU burst, followed by the shared 20 ms section. The parent then resumes, performing its 40 ms burst and the same 20 ms section. The sequence confirms both processes follow their own control paths and complete correctly after ENDIF.

System Status: The snapshot shows two PCBs: child running, parent waiting, proving successful process duplication and non-preemptive child-first scheduling.

Test Scenario 6 – Exec → Fork → Child Exec

Description: This test checks that a process created by exec() can fork and that the child can independently execute another program. init runs EXEC program8, which later forks, allowing the child to run EXEC program9 while the parent continues normal CPU work.

Execution Analysis: EXEC program8 loads a 10 MB program (150 ms), updates the PCB, and resumes in user mode. Inside program8, a FORK clones the PCB; the child runs first, executes a CPU burst, then performs EXEC program9 (15 MB → 225 ms loader) and its 60 ms CPU burst. Afterward, the parent resumes and completes its 40 ms CPU section.

System Status: The logs show init becoming program8, then splitting into two PCBs. The child later changes to program9 while the parent remains program8, confirming correct exec-after-fork behavior and independent process states.