SYSC4001 Assignment 2

Part 3 Report

11/7/2025

Written by Aws Ali - 101296896

https://github.com/AwsAli05/SYSC4001_A2_Part3

https://github.com/AwsAli05/SYSC4001_A2_Part2

The goal of this assignment was to simulate how an operating system handles the fork and exec system calls using interrupt routines and a simplified process control model. Each test represents a small operating system scenario where processes are created, loaded into memory, and executed while the simulator tracks each event in a detailed log. The logs show the order of kernel mode switches, context saving, vector table lookups, and the loading and execution of programs stored in external files.

The fork system call is simulated by entering kernel mode, saving the CPU context, and finding the interrupt vector that corresponds to the fork operation. The simulator then clones the Process Control Block (PCB) of the parent process to create a child. The child inherits its parent's attributes, such as the program name and size, and is allocated a partition in memory if space is available. The logs show events like *"cloning PCB"*, *"scheduler called"*, and *"IRET"*, which represent the steps the operating system takes to return control to user mode. In a real system, `fork()` creates a duplicate process, and both the parent and child continue executing from the same point. In our simulation, the child always executes first because it has higher priority, which prevents preemption and simplifies scheduling.

The exec system call replaces the process image of the currently running program with a new executable. In the simulation, the system enters kernel mode, saves context, and finds the corresponding interrupt vector for exec. The simulator then reads the external file table to determine the program's size and multiplies it by 15 ms per MB to estimate loading time. The events *"Program is 10 MB large"* and *"loading program into memory"* represent this step. The program is placed in the smallest free partition that can fit it, and the PCB is updated to reflect the new process state. If memory cannot be allocated, the simulator logs *"Error: no available memory partition"* and skips further execution for that process.

Across the tests, the system follows a consistent pattern. For example, in test 1, the init process forks a child that executes *program1* while the parent waits. After the child completes, the parent executes *program2*, which includes a system call handled through vector 4 with a 250 ms delay. The log steps *"switch to kernel mode"*, *"find vector 2"*, and *"load address 0x0695 into PC"* mirror what happens inside a real CPU when the control flow moves into an interrupt service routine. The return instruction *"IRET"* shows the transition back to user mode.

In test 2, a nested fork occurs: the init process creates a child that executes *program1*, which then forks another child that executes *program2*. This test demonstrates that each new process receives its own partition and that the scheduler respects the child-first execution rule. The *system_status.txt* file confirms that process 2 runs *program2* while process 1 and init wait. This mimics a simple ready queue where only one process can run at a time.

Test 3 verifies that CPU bursts, system calls, and end-of-I/O interrupts appear correctly in sequence. The trace shows *"SYSCALL ISR"* followed by *"ENDIO ISR"* using vector 6, reflecting how the OS responds to hardware events and returns control after completion.

In tests 4 and 5, extra operations like multiple CPU bursts and parent-side forks confirm that the simulator handles process scheduling and memory reuse properly. The marking and updating

PCB steps show that the simulator maintains partition records after every exec. Longer loader times for larger executables (15 MB → 225 ms) make the total runtime longer, which matches expectations for real I/O-bound operations.

Overall, the simulation logs provide a clear picture of what happens when a system call triggers an interrupt. Each entry corresponds to an actual event in a real operating system: switching to kernel mode, saving CPU context, identifying the interrupt vector, executing the appropriate ISR, updating the PCB, and returning to user mode. The structure of the logs shows that even though this simulator runs on simple files, it reflects real-world OS design concepts such as process creation, program loading, and context switching.

The break statement at the end of the EXEC section is very important because it prevents the program from continuing to loop through the main trace after executing the new program's trace. Once a process has replaced itself with a new program, the old code should not keep running. Without the break, the simulator would continue reading further trace lines from the parent file after the child or new program had finished, which would cause duplicate or incorrect executions. In a real operating system, the `exec` call does not return to the old code because the process's memory and instruction flow are replaced. The break ensures that the simulation stops at that point, just as the real system would.

In conclusion, this simulator accurately models the fork and exec system calls and shows the relationships between interrupts, system calls, and process scheduling. The logs explain how processes are created, loaded, and executed one at a time while the kernel updates its data structures. Each test confirms that the simulation behaves as expected, reproducing the steps a real operating system takes when handling process creation, memory allocation, and execution.