Introduction

The following assignment simulates process creation for interrupt handling using recursive calls to model FORK, EXEC, SYSCALL, and END_IO operations. The goal is to trace how the operating system switches between user and kernel modes, manages PCB tables, allocates memory, and executes multiple child processes. A total of five test scenarios were conducted, three given plus two additional custom-designed tests.

Tests

Test 1 – Basic Fork + Exec:

[code given in assignment]

Output Behaviour:
The init process forks a child, the child executes program1 (a long CPU burst), and the parent executes program2, which triggers a SYSCALL. The Output shows the correct sequence of switch to kernel mode, cloning PCB, scheduler called, and IRET. Two PCBs appear in the system state file (parent + child).

Test 2 - Nested Exec

[code given in assignment]

Output Behaviour:
The recursive call in simulate_trace() successfully executed program1.txt and program2.txt, creating three total forks and four execs. The output matches the provided sample from the assignment manual.

Test 3 - Return from Child (Scheduler Validation)

[code given in assignment]

Output Behaviour:

The objective was to verify that when a child process finishes execution, the scheduler correctly switches control back to the parent. As a result, the IRET instruction appears immediately after each child terminates, followed by the parent's resumed CPU burst. PCB entries show terminated processes removed and parents transitioned from "waiting" to "running".

Custom Test 4 - Syscall and END_IO

trace.txt:

FORK, 15
IF_CHILD, 0
EXEC program1, 40
IF_PARENT, 0
EXEC program2, 20
ENDIF, 0

program1.txt:

CPU, 30
SYSCALL, 6
END_IO, 6
CPU, 20

program2.txt:

CPU, 25

The purpose of this test was to evaluate ISR handling, interrupt vector lookup, and delay processing from device_table.txt. The behaviour is that SYSCALL triggers an interrupt using the vector table, then the ISR executes with delay from the corresponding device, then END_IO signals I/O completion and calls the scheduler. As seen in execution test-4.txt, the output confirmed interrupt service routines are correctly invoked and the kernel returns to user mode afterward.

Custom Test 5 - Multiple Exec and Memory Reallocation

trace.txt:

FORK, 10
IF_CHILD, 0
EXEC program1, 30
IF_PARENT, 0
EXEC program2, 20
EXEC program1, 30
ENDIF, 0

program1.txt:

CPU, 50

program2.txt:

CPU, 50

The purpose of this test was to verify multiple sequential EXECs, ensuring proper partition freeing and reallocation. The behaviour is that each EXEC frees the previous program's memory (using free_memory()), then reassigns a new partition. Then, the execution trace shows multiple "marking partition as occupied" and "updating PCB" entries. Our output, as seen in "execution test-5.txt", confirms sequential EXECs, PCBs update correctly, and no duplicate allocations. The system_status test-5.txt shows alternating "running" and "waiting" states across reloaded processes.

Analysis

Test #	Key Feature	Expected Behaviour	Observed Behaviour	Result
1	Basic FORK & EXEC	2 PCBs, forked child	Match	PASS
2	Nested EXEC	Recursive exec, 4 total execs	Match	PASS
3	Scheduler Return	Parent resumes after child ends	Match	PASS
4	SYSCALL / END_IO	ISR triggered, delay added	Match	PASS
5	Multiple EXEC	Partition freed/reassigned	Match	PASS

The five tests evaluate every major component of the interrupt simulator: process creation, execution replacement, I/O handling, and scheduling. Tests 1-3 show predictable growth and reduction of the PCB table. While tests 4 and 5 show time values in the execution traces increase linearly with no overlap, indicating proper synchronization between kernel and user modes.

Each EXEC call freed and reallocated partitions without overlap, confirming zero internal fragmentation, and PCB updates always reflected the correct state transitions from running to waiting to terminated.

All together, these tests demonstrate that the system behaves consistently and correctly.

Explanation for “Why is this important?”

In interrupts.cpp, the line: “*break; //Why is this important?*” appears immediately after the recursive EXEC call. Without the break statement, after finishing the recursive simulation of the new program, the parent function would continue executing the remaining lines of the old trace file, causing the parent and child traces to behave incorrectly. The break statement ensures the current process is fully replaced by the new program and the previous trace is not re-executed.