Test 1: Basic Fork and Exec

In this test, the system started with only the init process. The trace showed a FORK at time 0, which created a child process. The child was given priority, so it ran first while the parent waited. The child then executed program1, which was 10 MB in size, and was correctly loaded into memory partition 4. After finishing program1, the parent resumed and executed program2, which was 15 MB in size and loaded into partition 3. The execution log showed proper context saving, vector lookups, and IRET returns. The system status confirmed that memory allocation worked correctly, and both processes transitioned between running and waiting states as expected. This test verified that FORK and EXEC work correctly and that process switching happens in the right order.

Test 2: Nested Forks and Shared Execution

In this test, init performed a FORK and the child executed program1. Inside program1, another FORK occurred, which created a third process. This showed that the simulator could handle multiple processes at once. The logs showed that each new process got its own PCB entry and partition, and that the system correctly saved and restored contexts during each switch. Both the child and the parent of program1 later executed program2, which showed that the same external program can be executed by multiple processes without conflict. Timing in the log also showed that the child ran first, followed by the parent. The PCB table displayed all three processes correctly with one running and two waiting. This test confirmed that the simulator supports nested forks, multiple active PCBs, and correct child–parent scheduling.

Test 3: Fork, Exec, and I/O Handling

In this test, init forked once and the child executed program1. The child's execution included CPU, SYSCALL, and END_IO instructions. The SYSCALL ISR introduced a 265 ms delay, which matched the value from the device table. The same delay occurred again for END_IO, showing that I/O handling and interrupt timing were simulated correctly. The results confirmed that the kernel switched to system mode before every interrupt and returned control to user mode afterward. The system status showed that memory partitions were correctly updated and that the I/O interrupts didn't affect other processes. This test verified that system calls and I/O events were properly synchronized and handled.

Test 4: Invalid Memory Allocation Test

This custom test was designed to check how the system behaves with invalid program sizes. The output showed that the simulator attempted to load program3 and program4, each with an impossible memory size of 4294967295 MB. The negative loading times in the output confirmed that integer overflow occurred. Although this result was unrealistic, it was useful because it showed that the simulator doesn't yet have protection for invalid or overflowed values. In a real operating system, this would trigger a memory allocation error. This test highlighted the need for input validation and size checking before allocating memory.

Test 5: Multiple Execs and System Call Combination

The final test combined several features at once: EXEC, FORK, and SYSCALL. The system first executed programA, which again had an invalid size, leading to large overflow values. After that, a fork occurred, creating a second process that later executed programB.
The child ran first, followed by the parent, and both successfully completed their system calls. The SYSCALL ISR delay of 211 ms appeared correctly in the log. However, both

programs had corrupted memory values due to the overflow issue, which was expected based on Test 4. Despite that, the process scheduling, ISR timing, and kernel transitions were all correct. The test proved that the simulator can still manage multiple interrupts and system calls even when one process has faulty data.

Overall Discussion

Across all five tests, the simulator behaved consistently and followed the correct order of execution for all system operations. The recursive handling of FORK allowed smooth switching between parent and child processes, and the memory management functions worked well in valid scenarios. The interrupt sequence (switch to kernel mode, context saved, vector lookup, load PC, ISR, IRET) was clear in every trace, showing that the system was processing interrupts correctly. The only major issue occurred in Tests 4 and 5, where invalid sizes caused unrealistic memory values. This could be fixed by adding range checks in the allocate_memory() or get_size() functions. Overall, the results confirmed that the simulation models the essential behavior of a real OS: process creation, context switching, and interrupt handling. Each test demonstrated a different aspect of system control and timing, making the project a complete simulation of low-level process management.