Carleton University SYSC 4001A - L3 - GROUP 23 - Assignment 2: Report

Names: Teereza Allaham (101289630) , Tamunoibuomi Okari (101287549)

Due: November 7th, 2025

https://github.com/TeerezaAllaham/SYSC4001_A2_P3

https://github.com/TeerezaAllaham/SYSC4001_A2_P2

INTRODUCTION

Operating systems manage the execution of programs, allocation of memory, and handling of system resources. Some of these operations include process creation and program execution, which are handled by system calls called FORK and EXEC. This simulation project aims to act as a simplified operating system environment, where processes are executed according to trace files, and system calls are simulated step by step.

METHODOLOGY

We created a C++ simulator that models a single CPU system handling CPU bursts, system calls, forks, and program executions from trace files. Each interrupt or system call switches to kernel mode, saves context, looks up the vector, loads the ISR address, executes, and returns with IRET, The simulator also manages memory partitions and tracks processes with PCBs, handling forks and EXEC calls. We tested it with different trace files to see how the different sequences of CPU bursts and system calls affect process scheduling and system state.

RESULTS AND ANALYSIS

Analysis of the execution of the trace file trace3.txt

*This simulation begins with the **init** process running and at time 0, it encounters a **FORK** system call, and this is what happens next.*

1. From (0-1ms): The CPU moves into a safe mode to handle the interrupt(kernel mode)
2. From (1 - 11ms): The current process's state is saved.
3. From (11-12ms): The system looks up the correct Interrupt Service Routine in the vector table for FORK.
4. From (12 - 13ms): The program counter points to the Interrupt Service Routine for FORK
5. From (13 - 23ms): The FORK routine runs and clones the init process

6. After this, the system will now have two processes running: the parent init, which goes into a waiting state, and the child init, which runs immediately.
7. The child process then executes an EXEC at 36ms, replacing its image with program 1
8. From (36 - 37ms): It switches into kernel mode to safely handle the call
9. From (37-56ms): The context is saved to remember the child's current state
10. From (56-206ms): The system looks up the correct Interrupt Service Routine in the vector table for EXEC.
11. From (56 -206ms): The new program is loaded into memory(Partition 4) replacing the child's old image
12. At 207ms, the child is now running program 1 while the parent init remains waiting
13. Program 1 then runs a CPU burst of 100ms(207 - 307ms) before a SYSCALL occurs
14. Similar ISR operations such as: Switching to kernel mode(307 - 308ms), Context saving(308 - 318ms), Vector Lookup to find the ISR(318 - 320ms), SYSCALL execution(320 - 531ms) and IRET(531 - 532ms) occur before Program 1 continues with a 50ms CPU burst(532 - 582ms)

Analysis of the execution of the trace file trace4.txt

1. From(0–1ms): CPU switches to kernel mode to handle the FORK called by init.

2.From(1–11ms): Init's state is saved.

3. From(11–12ms): FORK ISR is looked up in the vector table.

4 . From(12–13ms): Program counter set to FORK ISR.

5. From(13–30ms): FORK runs, cloning init; now two processes exist: parent init (waiting) and child init (running).

6.From(31–32ms): Child executes EXEC program1, 16, switching to kernel mode.

7.From(32–42ms): Child's context is saved.

8. From(42–219ms): EXEC ISR runs; program1 is loaded into Partition 4, replacing the child's old image.

9.From(220–248ms): FORK inside program1 creates a new child; processes: child of program1 (running), parent program1 (waiting), original init (waiting).

10.From(249–530ms): New child executes EXEC program2, 33, loading it into Partition 3.

11.From(531–583ms): CPU burst of program2 runs (53ms).

12.From(583–863ms): Parent program1 executes EXEC program2, 33, replacing its image in Partition 3.

13.From(864–917ms): Remaining CPU burst of original parent init runs (205ms), completing its workload.

Why is "break;" in line 230 important?

The break statement in line 230 is essential because it ensures the correct termination of the current process's trace once the EXEC system call has been executed. In an operating system, the EXEC call replaces the current process image with a new program, effectively discarding the old instructions and starting execution of the new program from scratch. In this simulation, after the EXEC system call loads the new program into memory and recursively calls simulate_trace() to execute that program's trace, the process has already transitioned to a completely different program context. Without the break; statement, the loop would continue reading and executing the remaining lines of the original trace file, which no longer represent the active process. This would result in inaccurate simulation behavior, such as mixing the old and new process instructions, generating incorrect system logs, and disrupting the time progression of the simulation. Therefore, the break; acts as a control mechanism that terminates the current trace loop at the exact moment the EXEC system call is completed, ensuring that the simulation correctly reflects real operating system behavior where the old program ceases to exist and the new one takes over execution.

CONCLUSION

The simulation shows how a single CPU system handles process creation, execution, and interrupts. The logs demonstrate how FORK and EXEC work, including context saving, ISR handling and memory allocation. The simulator effectively tracks process states, CPU bursts, and priority handling, giving a clear view of how multiple processes are managed.