# Assignment 2 Analysis Report

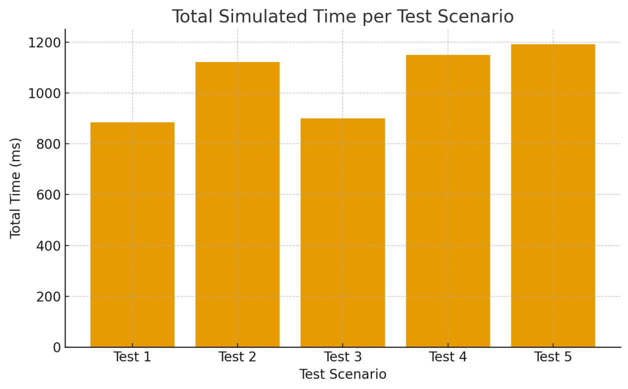
**Team members:** Manoj Kuppuswamy Thyagarajan 101166589
Saan John 101257741

**Here is the links to the project repositories:**
**Part 2:** https://github.com/KaiceADC/SYSC4001_A2_P2
**Part 3:** https://github.com/KaiceADC/SYSC4001_A2_P3

## Overall Analysis:

Across all five tests, the simulator accurately modeled fork() and exec() behavior, including process creation, execution, and interrupt handling. It correctly managed parent–child relationships, memory partitions, and mode switching during SYSCALL and ENDIO events. Even with nested forks, longer CPU bursts, and timing variations, the simulator maintained proper scheduling, PCB updates, and partition allocation, confirming its accuracy in replicating process management and interrupt-driven execution.

Here is a visual representation of the time taken for each test case:



The graph shows that the total simulated time increases with scenario complexity — Tests 1–3, which involve basic fork/exec and single-level interrupts, complete more quickly, while Tests 4 and 5 take significantly longer due to multiple forks, extended CPU bursts, and additional ISR activity, illustrating how greater process and interrupt handling overhead directly increases overall system execution time.

## General Overview of Execution Steps of Simulation:

The simulation starts by initializing system components such as memory partitions, the PCB table, and the ready queue, then loads input files including the trace, vector, device, and external file lists. It processes each instruction in the trace sequentially and handling FORK to create child processes, EXEC to load programs, CPU to simulate execution, and interrupts like `SYSCALL` or `END_IO` for context switching. Conditional blocks manage parent–child execution order. Throughout, events are logged in `execution.txt`, system states in `system_status.txt`, and the simulation ends by outputting final PCB and partition details.

**Question: What is the purpose of the “break;” in the EXEC block of code?**
The break statement in the EXEC block is basically a kill switch. After you fork and create a parent and child process, they'd normally both keep running the same code. But once a process calls exec to load a new program, you don't want it to keep going through the rest of the simulation code. The break statement stops that and it exits the current branch immediately so the process doesn't accidentally run both its own program and the other process's program. Without it, you'd get messy duplicate execution. Basically, break makes sure that when exec happens, that process is done with the simulation loop and moves on to actually run the program it loaded.

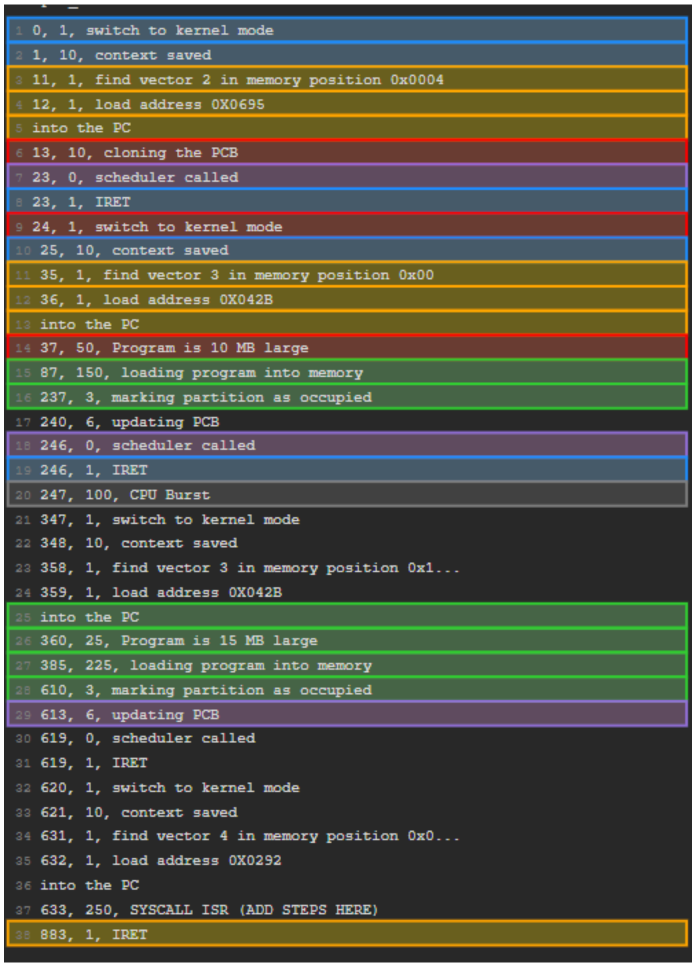
**Explain how the actual system call works:**
A system call (syscall) is a mechanism that allows a user program to request a service from the operating system’s kernel, such as reading a file, creating a process, or accessing hardware.

In this simulator, a system call (SYSCALL) is implemented as a software interrupt handled through the `handle_interrupt()` function. When a SYSCALL appears in the trace file, the simulator switches the process from user mode to kernel mode, saves its context, and locates the correct interrupt vector using `intr_boilerplate()`. It then executes the corresponding Interrupt Service Routine (ISR) with `execute_isr()`, simulating the kernel performing the requested operation. After the ISR completes, the system restores the process state and returns to user mode using `execute_iret()`. This sequence effectively mimics how real operating systems handle system calls — by temporarily transferring control from user space to the kernel, executing privileged actions, and then resuming normal execution.

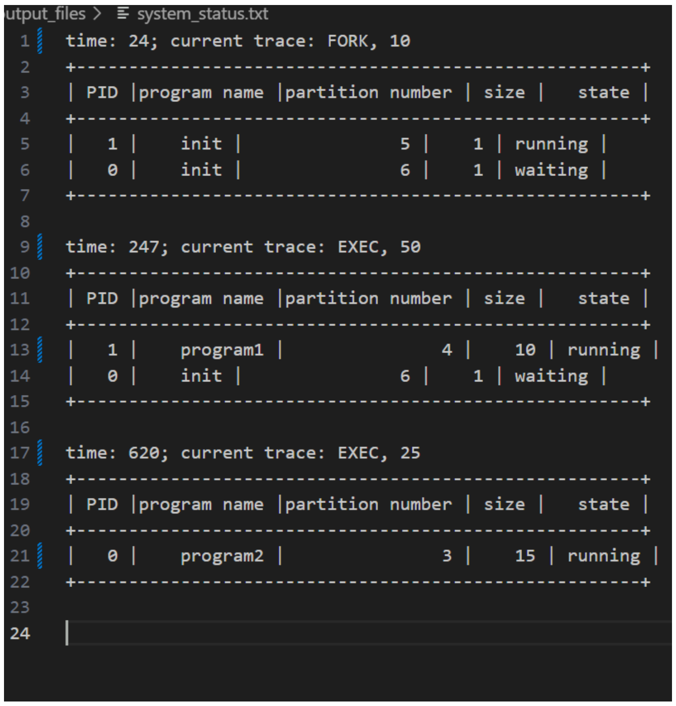
**Sample detailed explanation of output - Test 1**

| **Trace1.txt** | **Program1.txt** | **Program2.txt** |
|---|---|---|
| FORK, 10<br>IF_CHILD, 0<br>EXEC program1, 50<br>IF_PARENT, 0<br>EXEC program2, 25 | CPU, 100 | SYSCALL, 4 |

Trace1.txt is the control file that orchestrates everything. It forks a process (creating a child and parent), then branches based on which one you are. The child executes program1 for 50 time units, the parent executes program2 for 25 time units, and everything terminates at the end.
Program1.txt and Program2.txt are the actual programs being executed. Program1 just does straight CPU work for 100 time units. Program2 makes a system call that takes 4 time units.
The numbers after the operations (10, 0, 50, 25, 100, 4) are the time durations or identifiers for each operation



```
0, 1, switch to kernel mode
1, 10, context saved
11, 1, find vector 2 in memory position 0x0004
12, 1, load address 0X0695
into the PC
13, 10, cloning the PCB
23, 0, scheduler called
23, 1, IRET
24, 1, switch to kernel mode
25, 10, context saved
35, 1, find vector 3 in memory position 0x00
36, 1, load address 0X042B
into the PC
37, 50, Program is 10 MB large
87, 150, loading program into memory
237, 3, marking partition as occupied
240, 6, updating PCB
246, 0, scheduler called
246, 1, IRET
247, 100, CPU Burst
347, 1, switch to kernel mode
348, 10, context saved
358, 1, find vector 3 in memory position 0x1...
359, 1, load address 0X042B
into the PC
360, 25, Program is 15 MB large
385, 225, loading program into memory
610, 3, marking partition as occupied
613, 6, updating PCB
619, 0, scheduler called
619, 1, IRET
620, 1, switch to kernel mode
621, 10, context saved
631, 1, find vector 4 in memory position 0x0...
632, 1, load address 0X0292
into the PC
633, 250, SYSCALL ISR (ADD STEPS HERE)
883, 1, IRET
```



utput_files > system_status.txt

```
time: 24; current trace: FORK, 10
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    init |                5 |     1 | running |
|   0 |    init |                6 |     1 | waiting |
+------------------------------------------------------+

time: 247; current trace: EXEC, 50
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program1 |               4 |    10 | running |
|   0 |    init |                6 |     1 | waiting |
+------------------------------------------------------+

time: 620; current trace: EXEC, 25
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   0 |    program2 |               3 |    15 | running |
+------------------------------------------------------+
```

**How the OS Scheduler Actually Works Here:** The OS is juggling multiple processes program1, program2, and init and switching between them and managing memory. At time 24, a FORK operation creates two init processes: PID 1 (running in partition 5) and PID 0 (waiting in partition 6), establishing the parent child relationship [FORK/EXEC]. When something important happens, the system switches to kernel mode and saves the process context (registers, program counter) in the Process Control Block so it can resume later [Context Saving & Restoration]. The OS uses an Interrupt Vector Table to route different interrupt types to the right handler. For example, Vector 2 handles FORK operations, and the CPU jumps to addresses like 0x0695 to execute the appropriate kernel code [Interrupt handling]. When you fork, the system clones the entire PCB to create a new process [FORK/EXEC]. After processing an event, the scheduler decides which process runs next based on the scheduling policy [Scheduling Decisions]. Program 1 (10 MB) loads into partition 4 and runs from time 50 to 247, while Program 2 (15 MB) loads into partition 3 and starts at time 620 [Memory Operations]. At time 247, there's a 100-unit CPU burst with no interruptions [CPU Bursts & Execution]. The IRET instruction restores your saved context and switches back to user mode so execution continues where it left off.[Context Saving & Restoration]. At time 633, a SYSCALL ISR is invoked when a process needs the kernel to handle a system call [Interrupt Handling].