REPORT:

SIMULATION ANALYSIS OF IMPLEMENTED API SIMULATOR

PART III - DESIGN AND IMPLEMENTATION OF AN API SIMULATOR: fork/exec

Akshavi Baskaran	101302106
Liam Addie	101315124

Repository Link for part 2: *https://github.com/akshavib/SYSC4001-A2-P2.git*

Repository Link for part 3: *https://github.com/liamaddie/SYSC4001_A2_P3.git*

Lab Section: L4

Group #: 5

Due: November 7th, 2025

OBJECTIVE

This report presents an analysis of the implemented API simulator under varying simulation conditions. Key features such as variations in interrupt durations, memory partition allocation, and process creation through fork() and exec() are analyzed to understand their impact on the overall efficiency of the simulated operating system. This report examines the execution, resource utilization, and representation of context switching and process scheduling.

IMPLEMENTATION

The simulation was developed by including the interrupts.hpp header file and significantly modifying the interrupts.cpp file to implement the fork() and exec() logic. Multiple trace files (such as trace.txt) were formulated to model different process creation and execution scenarios involving parent–child interactions, conditional branching (IF_CHILD, IF_PARENT, ENDIF), and system call handling.

For each run, the final execution timeline, along with detailed logging statements, was recorded in output files (such as execution.txt), while system state snapshots which included active and waiting processes, memory partition usage, and PCB information were logged in system_status.txt.

** The program files are located in the corresponding input folders, even though they are not technically considered input files. However, all program files used are also in the main directory, named program1.txt through programn.txt, and is included under its respective section in the analysis for reference. **

1) ANALYSIS FOR MANDATORY TEST 1

Files: md_test1_in in input_files, md_test1_out in output_files, program1.txt, program2.txt

fork() and exec() are simulated in this test, ensuring that the child process executes before the parent continues (with non-preemptive child priority).

Fork Execution (0 - 23 ms)

When the program is executed, the system begins by switching to the kernel, saving the context `(log: 0, 1, switch to kernel mode and 1, 10, context saved)`, fetching the interrupt vector for the fork process `(log: 11, 1, find vector 2 ...)`, and triggering the ISR. The ISR clones the parent PCB to initialize the child process `(log: 13, 10, cloning the PCB)`, with a unique PID, but identical code and attributes. The scheduler is invoked to select the next process `(log: 23, 0, scheduler called`), which is the child, and the parent is moved to the waiting state. The child then begins to run.

In a real kernel, fork() returns twice: once in the parent (which returns the child's PID) and once in the child (which returns 0). In this simulator, the fork only "returns" once, (since the child trace runs while the parent is waiting) so the effect is represented through process state changes.

Child EXEC program1 (24 - 247 ms)

The child executes the EXEC program1, once again by switching to kernel mode, getting the vector, and memory allocation is then attempted `(log: 24, 1, switch to kernel mode, log: 25, 10, context saved, log: 35, 1, find vector 3 ..., log: 36, 1, load address ...)`. After this, a free partition is assigned: Partition 4 = 10 MB. The loader simulates copying data from secondary storage into RAM, 10 MB × 15 ms = 150 ms `(log: 37, 50, Program is 10 MB large and log: 87, 150, loading program into memory)`. After, the PCB is updated with the new program details, and the scheduler resumes the execution `(log: 246, 0, scheduler called and log: 246, 1, return from interrupt (IRET))`. The child process runs the content of program1.txt, completing its run `(log: 247, 100, CPU Burst)`. In an actual system, exec() replaces the process image and the kernel loads the program and updates the process's memory mapping. This is shown in the simulation by changing the PCB's program name and size, and changing its partition in respect to its program size.

Parent EXEC program2 (347 - 883 ms)
Once the child is done, the parent executes the EXEC program2. The init code is now program2, the ISR for EXEC loads program2 into partition 3, as it has a space of 15MB `(360, 25, Program is 15 Mb large and log: 385, 225, loading program into memory)`. The program is loaded, the PCB is updated, and then the program executes its internal trace `(SYSCALL, 4)`, producing its own interrupt log `(log: 633, 250, SYSCALL ISR, etc.)`. After this has completed `(log: 883, 1, IRET)`, both processes terminate.

In this trace file, ENDIF, 0 does not matter. After FORK, the child and parent execute different code blocks. Once they both reach ENDIF, they rejoin. However, in this case, both processes end execution before that point, so anything after ENDIF does not get reached.

2) ANALYSIS FOR MANDATORY TEST 2

Files: md_test2_in in input_files, md_test2_out in output_files, program3.txt, program4.txt

fork() and exec() are simulated in this test, showing that multiple generations of processes execute in order of creation. This test specifically uses nesting, as program1 calls program2. The trace starts with the init process forking once, followed by the child executing another program that performs a second fork and exec, leading to three active PCBs in memory before completion.

Fork Execution (0 – 30 ms)
This process follows the same detailed steps as Mandatory Test 1. The system switches to kernel mode, saves the context, fetches the interrupt vector for the fork, and triggers the ISR to clone the parent PCB, creating a new child process. The scheduler then runs the child while the parent moves to the waiting state, just as described in the first test.

Child EXEC program1 (31 – 219 ms)
The beginning of this process is the same as in Mandatory Test 1, following the same detailed sequence. The child switches to kernel mode, saves its context, gets the interrupt vector for EXEC, and triggers the ISR. The loader then loads program1 into memory, assigns a 10 MB partition, updates the PCB, and resumes execution. The behavior and order of operations are identical to Test 1.

Within program1, another fork() call is made. The same process occurs: the system switches to kernel mode, saves the context, and fetches the interrupt vector `(logs: 220, 1, switch to kernel mode to log: 232, 1, load address 0X0695 into the PC)`. The ISR clones the process, creating a new PCB for the child `(log: 233, 15, cloning the PCB)`. The scheduler again selects the child to run first `(log: 248, 0, scheduler called and 248, 1, return from interrupt (IRET)).`

Nested Child EXEC program2 (249 – 529 ms)
The child of program1 executes EXEC program2, switching to kernel mode and fetching the EXEC vector `(log: 249, 1, switch to kernel mode to log: 261, 1, load address 0X042B into the PC)`. The program is identified as 15 MB in size, and the loader simulates loading it into memory (15 MB × 15 ms = 225 ms; `log: 262, 33, Program is 15 MB large and log: 295, 225, loading program into memory`). Partition 3 = 15 MB is assigned, and the PCB is updated `(logs: 520, 3, marking partition as occupied and log: 523, 6, updating PCB`). The scheduler resumes the process, and the CPU begins executing the program's instructions `(log: 530, 53, CPU burst)`.

Here, the child of program1 executes program2. This occurs because after the nested fork, the scheduler prioritizes the new child process. The parent (program1) waits until the child finishes.

Parent EXEC program2 (583 – 863 ms)
After the grandchild (child of program1) completes its CPU burst `(log: 530, 53, CPU burst)`, the parent process (program1) resumes execution. The parent executes its own EXEC program2, repeating the same system call sequence: switch to kernel mode, save context, load vector, and trigger the ISR `(log: 583, 1, switch to kernel mode through and log: 595, 1, load address 0X042B into the PC)`. The loader again simulates program loading (15 MB × 15 ms = 225 ms; `log: 596, 33, Program is 15 MB large and log: 629, 225, loading program into memory`). The partition and PCB are updated, and execution continues `(log: 854, 3, marking partition as occupied, and log: 857, 6, updating PCB, 863, 1, return from interrupt (IRET))`. The process then executes its own CPU burst `(log: 864, 53, CPU burst)`.

Parent CPU Burst and Termination (917 – end)
Finally, the original parent (init) resumes execution and performs a final CPU burst of 205 ms `(log: 917, 205, CPU burst)`. Once this completes, all processes have finished their execution, and the simulation ends.

3) ANALYSIS FOR MANDATORY TEST 3

Files: md_test3_in in input_files, md_test3_out in output_files, program5.txt

This test is a complete process with fork(), exec(), SYSCALL and END_IO operations. It is very close to how a real OS would have to process, and it validates that this simulator correctly handles process creation, execution, I/O handling, and process resumption all together.

FORK Execution (0–33 ms)
The fork() sequence behaves the same as in Test 1, where the system switches to kernel mode, saves context, retrieves the interrupt vector, and clones the PCB. The parent waits while the child runs.

Parent EXEC program1 (44–276 ms)
Similar to the EXEC steps in Test 2, the parent switches to kernel mode, retrieves the EXEC vector, and loads program1 into memory. The PCB is updated and the scheduler resumes the process.

Program1 Execution and SYSCALL (277–605 ms)
The new program begins execution with a CPU burst `(log: 277, 50 CPU burst)` and then triggers a SYSCALL. This interrupt follows the same logic as in Assignment 1, which was switching to

kernel mode, saving context, and running the ISR, transferring device data, and finalizing the ISR. Execution resumes after returning from interrupt.

END_IO Handling (606–899 ms)
After a CPU burst, an END_IO interrupt occurs. The ISR runs and validates device status, before control returns to user mode `(log: 899, 1, return from interrupt`). This confirms proper coordination between device completion and process continuation.

4) ANALYSIS FOR FORK INSIDE EXEC TEST

Files: fork_exec_test_in in input_files, fork_exec_test_out in output_files, program6.txt, program7.txt

The simulation begins with an EXEC call loading a 10 MB program into memory `(logs 0–197 ms)`. After initialization, the program performs a FORK `(logs 198–226 ms)`, creating a child PCB. The child runs first, executing a short CPU burst (30 ms), followed by an EXEC of another program (15 MB), which loads into memory `(270–529 ms)` and runs `(530–570 ms)`.

Next, a SYSCALL is issued for device 5 `(logs 570–794 ms)`, simulating kernel ISR handling, data transfer, and completion checks, followed by an END_IO `(805–1029 ms)`, confirming proper interrupt return (IRET).

Overall this test validates that nested recursion works correctly, memory loading updates occur in order, and SYSCALL and END_IO ISRs execute and resume properly.

5) ANALYSIS FOR SIBLING PROCESSES TEST

Files: sibling_proc_test_in in input_files, sibling_proc_test_out in output_files, program8.txt

This test confirms the simulation of creating sibling processes. Starting with a fork, where the kernel saves context, retrieves the vector, and clones the parent PCB. The child process then runs `(25 - 79 ms)` and the child executes. The kernel switches modes, finds the vector, and loads a 10 MB program into memory `(loading time 150 ms)`. The PCB is updated, and a new memory partition is assigned before the scheduler resumes execution.

The process then performs one final CPU burst `(log: 307, 80 ms)`, completing the run. This demonstrates the correct sequencing of FORK, then CPU bursts, then EXEC, and dealing with memory management and context switching.

IMPORTANCE OF BREAK STATEMENT

Inside the EXEC branch (else if) of the main simulate_trace() loop in the interrupts.cpp file, there's a `break;` statement at the end of it. This is important because when an EXEC system call is made, the current process replaces its memory image with a new one and executes it, meaning it should no longer execute the old program's trace file (i.e., the one before EXEC was called, a.k.a., the parent). The `break` stops the outer `for` loop in simulate_trace() from continuing, ensuring that the simulator stops processing any further trace lines from the old program's trace and instead runs the new program's trace file. Otherwise, the simulator would continue to execute the trace lines from the old program, combining them with the new one's, which is wrong.