Assignment 2

Sysc 4001A: API Simulator Tests & Analysis
Authors: Aditi Nahar and Yuvraj Bains

Introduction

This simulator reproduces how an operating system handles interrupts and system calls such as FORK, EXEC, SYSCALL, and END_IO. Each trace line represents a kernel-mode transition, a context save, an interrupt vector lookup, or a CPU burst. Our goal was to trace the complete lifecycle of parent and child processes and observe how process control blocks (PCBs) change after each event.

The program outputs two logs:

- execution.txt – chronological ISR and CPU-burst trace,
- system_status.txt – snapshot of process control blocks (PCBs) showing PID, partition, and state.

The execution log represents time-ordered CPU activity, while the system-status log shows which processes occupy each memory partition at a given time.

Simulator Design

All execution is done by the recursive function:

```
simulate_trace(std::vector<std::string> trace_file, int time, ...);
```

which reads each line of a trace file and performs actions like FORK, EXEC, or CPU. Every interrupt follows the same pattern: the system switches to kernel mode, saves the context, looks up the vector address, loads the ISR, waits for the delay from the device table, updates the PCB, and returns to user mode with IRET.

PROCESS MANAGEMENT FLOW

FORK: When a fork happens, the system copies the current process to create a child. The parent is moved to the waiting queue, and the child runs first. The simulator collects the child's instructions between IF_CHILD and IF_PARENT, runs them recursively, and frees the child's memory once it finishes so the partition can be reused.

EXEC: Exec replaces the current program with a new one. The simulator loads the new file, calculates the load time (size × 15), marks its partition as occupied, updates the PCB, and runs the new program. The break at the end is needed because exec never returns in a real system. Without it, the parent's old code would still run and cause duplicate or incorrect results.

SYSCALL and END_IO: A syscall happens when a program asks the system to perform an I/O task. The process pauses while the operation runs. When the I/O finishes, an END_IO interrupt resumes it, after a short delay. This simulates how real systems handle asynchronous I/O, where the CPU can keep working on other tasks while waiting for a device to finish.

TESTS & ANALYSIS

Test 1: Basic Fork and Exec

Init begins alone in the system and performs a fork. The fork creates a new child PCB while the parent is moved to the waiting queue. The child runs first and executes program1, which performs a CPU burst of 100. After the child finishes, the parent resumes and executes program2, which performs a system call (SYSCALL 4).

The execution log shows the normal kernel steps for fork and exec. During exec, the program is replaced and loaded into memory based on its size. The syscall shows an interrupt service routine (ISR) and a matching delay from the device table.

```
0, 1, switch to kernel mode
13, 10, cloning the PCB
247, 100, CPU Burst
347, 1, switch to kernel mode
633, 250, SYSCALL ISR
883, 1, IRET
```

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

The test verifies correct forking and the replacement of the process image. The child finishes first, then the parent runs program2 and performs a system call.

Test 2: Nested Fork and Exec

Init forks a child, which executes program1. Inside program1, another fork is called, creating a new child process. The new child of program1 executes program2, while the parent of program1 waits. When the new child finishes, the parent of program1 resumes and executes the same program2 again. After both are done, the original init process performs its final CPU burst.

The logs show three PCBs existing at the same time: the new child, the parent of program1, and init. Each exec section shows the program size being loaded, partitions being updated, and scheduler activity between process switches.

```
0, 1, switch to kernel mode
13, 17, cloning the PCB
220, 1, switch to kernel mode
249, 1, switch to kernel mode
530, 53, CPU Burst
864, 53, CPU Burst
```

```
time: 530; current trace: EXEC program2, 33
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   2 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+-------------------------------------------------------+
```

This test confirms that the simulator correctly handles multiple levels of forks and executes processes recursively without losing memory references or overwriting PCBs.

Test 3: Fork with I/O

Init forks and the child executes program1, which contains a CPU burst, a syscall, an end_io event, and another CPU burst. The logs show that the syscall triggers a switch to kernel mode, context saving, and a SYSCALL ISR with a delay of 250 ms. Later, the end_io interrupt resumes the process after the delay and continues normal execution.

```
277, 50, CPU Burst
327, 1, switch to kernel mode
605, 1, IRET
606, 15, CPU Burst
621, 1, switch to kernel mode
899, 1, IRET
```

```
time: 530; current trace: EXEC program2, 33
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   2 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+-------------------------------------------------------+
```

This test checks asynchronous I/O handling. When the syscall occurs, the process pauses and the CPU can be used by other processes. When the end_io happens, the scheduler brings the waiting process back. The delays match the device table values, showing that the simulation models I/O behavior correctly.

Test 4: Self-Made Fork–Exec Chain

Trace Files: trace(test4-selfmade).txt, program1(Test4).txt, program2(Test4).txt

In this test, init forks and the child executes program1. Inside program1, another fork occurs, creating one more process. Both the child and parent from program1 execute program2. Finally, init resumes and completes its own CPU burst at the end.

The logs show multiple exec sequences where the program sizes are loaded (10 MB and 15 MB), partitions are marked as occupied, and PCBs are updated. Two exec program2 entries appear in the log at different times, confirming that both parent and child of program1 executed it. The final CPU burst belongs to the original init process.

```
13, 10, cloning the PCB
205, 1, switch to kernel mode
225, 1, switch to kernel mode
485, 25, CPU Burst
770, 25, CPU Burst
795, 40, CPU Burst
```

```
time: 225; current trace: FORK, 6
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   2 |    program1 |               3 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+-------------------------------------------------------+
```

This test proves that the simulator handles sequential forks and execs correctly. The system frees the child's partition before resuming the parent, allowing the same partition to be reused. The scheduler consistently gives the child priority after each fork, which matches real process scheduling in an operating system.

Test 5: Self-Made Fork with I/O and Exec Chain

Trace Files: trace(test5-selfmade).txt, program1(Test5).txt, program2(Test5).txt

Init performs a fork, and the child executes program1. Inside program1, there are CPU bursts, a syscall, an end_io event, and an exec of program2. The parent continues its CPU burst after the child finishes.

The execution log shows the syscall interrupt pausing the process, followed by an IRET instruction when it resumes. After the end_io event, the process continues execution and then replaces its image using exec program2. The loader lines show the 15 MB load and the updated PCB state. When program2 finishes, the parent process performs its final CPU burst.

```
200, 10, CPU Burst
223, 250, SYSCALL ISR
474, 5, CPU Burst
492, 250, ENDIO ISR
997, 15, CPU Burst
1012, 20, CPU Burst
```

```
time: 200; current trace: EXEC program1, 8
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+
```

This test demonstrates the complete chain of process creation, I/O handling, and image replacement. It confirms that the simulator correctly manages context switches, I/O delays, memory freeing, and scheduling. The order of events matches child-first scheduling after each fork, and the results validate correct interaction between system calls, exec, and scheduler behavior.

SUMMARY

Across all five tests, the simulator behaves as a simplified model of an operating system. The logs confirm correct fork handling, where the child always runs before the parent. The exec replacement accurately loads new programs and stops the old code using the break statement. Syscall and end_io correctly suspend and resume processes using device delays. Memory management works as intended, with partitions being freed and reused between child and parent processes. The results together show that the simulator correctly shows core process control operations and I/O management.

ASSOCIATED LINKS

part 2: https://github.com/yuvraajbains/SYSC4001_A2_P2
part 3: https://github.com/aditinahar2005/SYSC4001_A2_P3