Part 3 Report: Design and Implementation of the API Simulator (fork/exec)

Overview:
This project models process management system calls, fork() and exec(), within an operating system level simulator. Each instruction in the simulator represents a CPU level sequence of events: entering kernel mode, saving context, locating the interrupt vector, updating the Program Counter (PC), executing the appropriate Interrupt Service Routine (ISR), and returning to user mode via IRET.The simulator reproduces the control flow of a simple OS kernel: process creation (fork()), process image replacement (exec()), scheduling, and memory partitioning. It was implemented in C++ using structured data types (PCB, memory_partition_t, external_file) and vector-based trace simulation.

1. System Behavior:

Every simulated instruction begins with a transition to kernel mode, followed by context saving and vector lookup. The vector table determines which ISR is triggered (CPU, SYSCALL, END_IO, etc.).In code, this logic is shown in intr_boilerplate() and simulate_trace(). Each system call follows this sequence:

Context save: intr_boilerplate()
Vector lookup: update PC to ISR address
ISR execution: modify PCB and memory state
Return to user mode: IRET

This ensures each system call realistically mirrors an OS level interrupt cycle.

2. The fork() System Call:

When fork() executes, the simulator clones the parent process, producing a new PCB with its own PID and partition. The parent process transitions to waiting, and the new child becomes running. Typical log segment:

```
24, 1, switch to kernel mode
34,10, cloning the PCB
44, 0, scheduler called
45, 1, IRET
```

And corresponding system status table:

```
time: 24; current trace: FORK, 10
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
```

```
|   1 |init          |               5 |    1 | running |
|   0 |init          |               6 |    1 | waiting |
+------------------------------------------------------+
```

This matches actual fork() semantics: the parent waits while the child continues execution independently.

3. The exec() System Call:

exec() replaces the current process's program image with a new one. The simulator frees the previous memory partition (if required), allocates a new partition based on external_files.txt, and updates the PCB's program name and size. Illustrative log sequence:

```
247, 1, switch to kernel mode
260,50, Program is 10MB large
310,150, loading program into memory
470, 6, updating PCB
477, 1, IRET
```

Status table:

```
time: 247; current trace: EXEC program1_1, 50
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |program1_1   |               4 |   10 | running |
|   0 |init         |               6 |    1 | waiting |
+------------------------------------------------------+
```

The PCB now shows program1_1 in partition 4, confirming that the child's process image was successfully replaced.

4. Extended Simulation Testing:

To evaluate robustness, a multi-fork/exec trace was executed (e.g., trace_4.txt). This verified the simulator's ability to handle nested process creation and replacement without memory conflicts. Example snippet:

```
965, 1, switch to kernel mode  // second fork
975,10, cloning PCB
985, 0, scheduler called
990, 1, IRET
1122, 1, switch to kernel mode // exec()
```

1135,564, SYSCALL ISR

Each new child process received a unique PID and partition, and memory was properly freed and re-allocated for subsequent exec calls.

5. Interpretation of Results:

Across all five simulation traces in /input_files, the output logs confirm:

- Correct kernel/user mode transitions
- Accurate PCB duplication and memory allocation during fork()
- Proper image replacement and PCB updates during exec()
- Valid scheduling and parent child synchronization (running vs waiting)
- Realistic interrupt and IRET logging within execution traces

The formatted system status tables display each process's current state and memory assignment, matching the expected behavior of a multitasking system.

Conclusion:

The final version of the simulator (Interrupts_101297993_101302793.cpp + main.cpp) faithfully models the fork() and exec() mechanisms within a simplified OS environment. It manages multiple processes, context switches, and partition based memory allocation while maintaining distinct PCB entries for running and waiting states.The test results in /output_files demonstrate accurate system behavior and validate the correctness of the implementation in the provided repository (https://github.com/Noor-e001/SYSC4001_A2_P3). Together, the code and logs confirm that the simulator achieves the learning objectives of Part 3 by accurately emulating OS process creation and execution dynamics.