SYSC 4001 Assignment 2 - report
Arshiya Moallem (101324189) - Student 1
Aveen Majeed (101306558) - Student 2

Part 3

Github link: https://github.com/arshiyamoallem/SYSC4001_A2_P3

Introduction:

This report discusses how the API simulator was used to model the fork() and exec() system calls in a Unix-like environment. The purpose of the simulator is to demonstrate how new processes are created and how different programs are loaded into memory and executed. To keep things simple, the simulator runs on a setup with one CPU, fixed memory partitions, and no preemption, meaning the child process always runs before the parent continues.

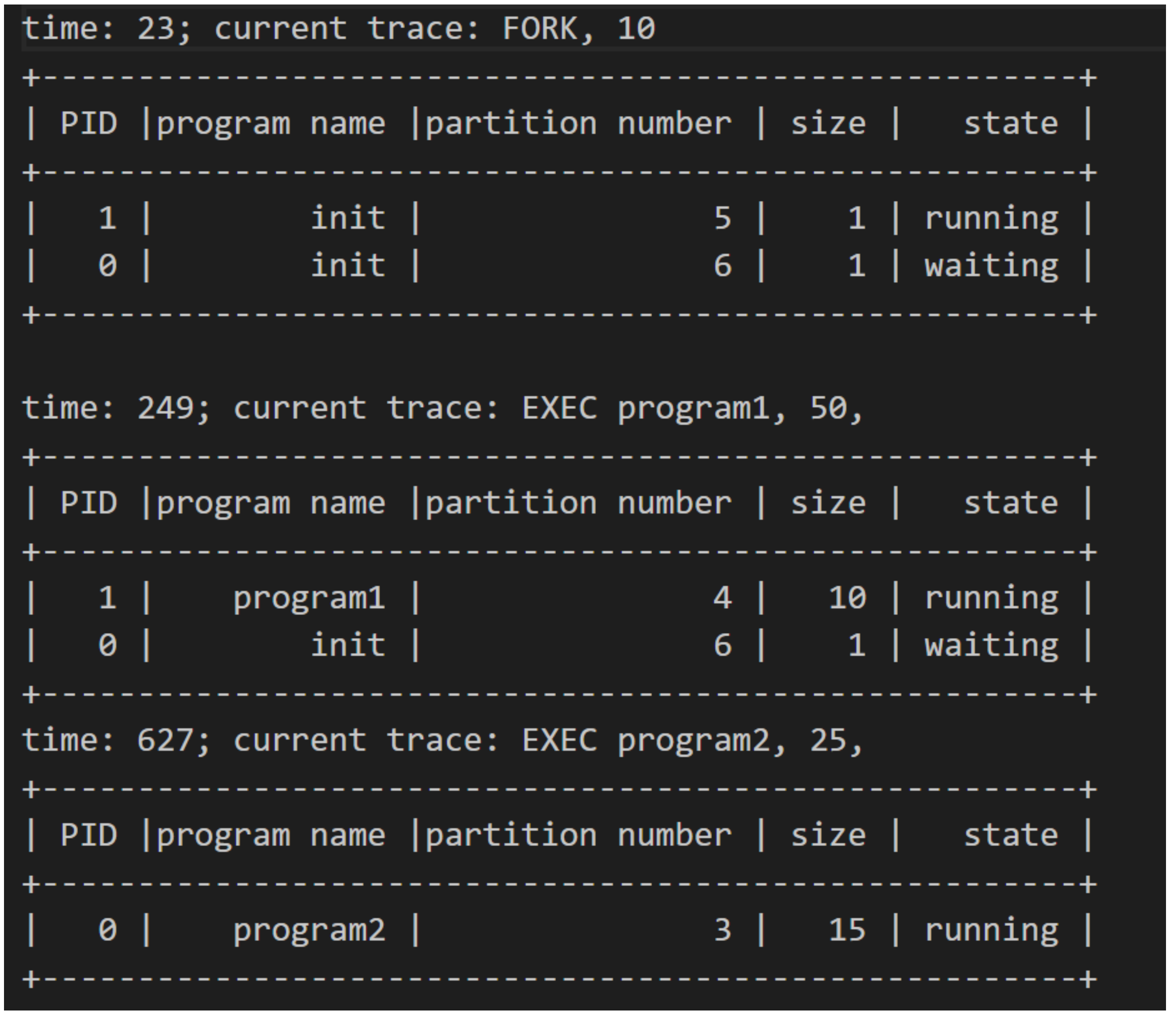
Each simulation test is based on a trace file that contains commands such as FORK, EXEC, and CPU, as well as conditionals like IF_CHILD, IF_PARENT, and ENDIF. These commands define which actions belong to the child or parent process and help simulate real system behavior. The following sections describe three different simulations that show how the system handles process creation, execution, and switching between parent and child processes.

Simulation 1 Basic Fork and Exec:

The simulation starts with the init process called FORK. This creates a child process, and because the child has higher priority, it runs first. The child executes EXEC program1, 50, which replaces its code with program1. The simulator looks up the program in the external files list, loads it into memory, and updates the PCB and partition table.

After the child finishes, the parent resumes and runs the code under IF_PARENT. It executes EXEC program2, 25, loading program2 in the same way. By the end, both the parent and child have run different programs, and the system log shows their states changing correctly.

This simulation shows the basic fork and exec process: a process creates a child, the child runs first, and then the parent continues with its own program.



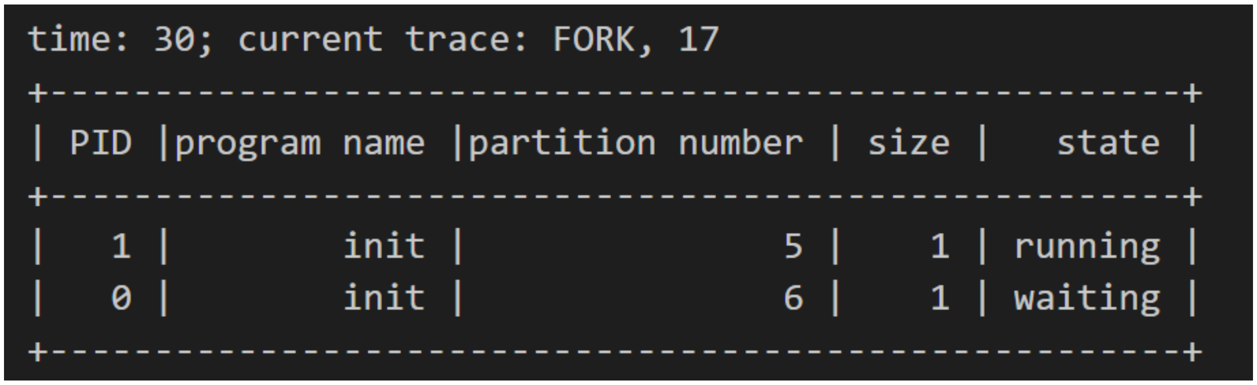
```
time: 23; current trace: FORK, 10
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+

time: 249; current trace: EXEC program1, 50,
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
time: 627; current trace: EXEC program2, 25,
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   0 |    program2 |               3 |   15 | running |
+------------------------------------------------------+
```

Simulation 2 Nested Fork within a Child Program:

This test begins with another fork, creating a new child process. The child immediately runs EXEC program3, 16, which loads program3 into memory. The simulator saves the context, finds the correct ISR for the exec call, and updates the PCB after the program loads.

When the child finishes running, the parent continues. The last command, CPU, 205, means the parent does a long CPU burst after all the child actions are done.

This test shows that the simulator correctly handles separate child and parent actions, making sure the child runs fully before the parent starts again.



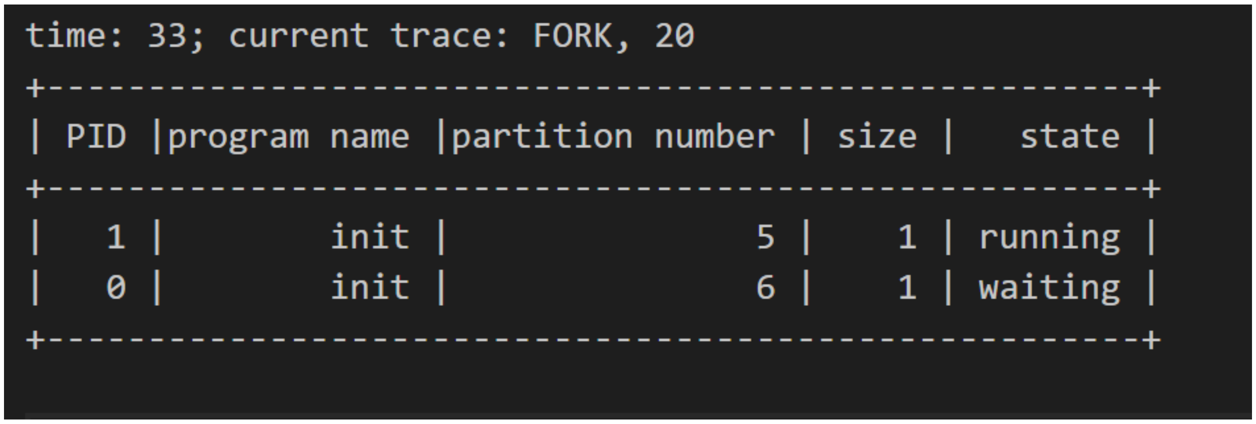
```
time: 30; current trace: FORK, 17
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
```

Simulation 3 System Calls and I/O Interrupts:

In this simulation, FORK again creates a child, but both condition sections are empty except for the parent's EXEC program5, 60. This means the child process doesn't do much, it just finishes quickly.

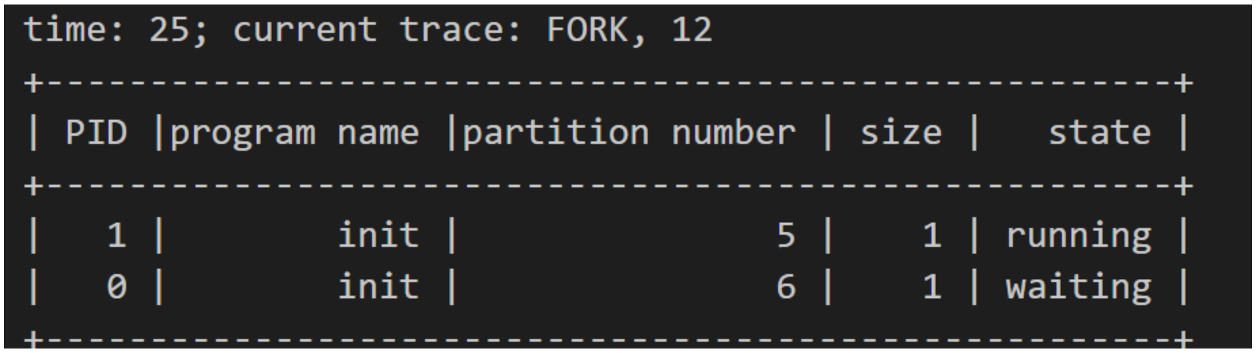
Once the child ends, the parent executes the EXEC call, which loads program5 into a free partition. The simulator calculates how long it takes to load based on the program's size and updates the PCB. After that, the parent runs the last CPU, 10 commands to finish.

This case shows how the simulator handles simple or empty child branches, and that it still correctly resumes the parent when the child is done.



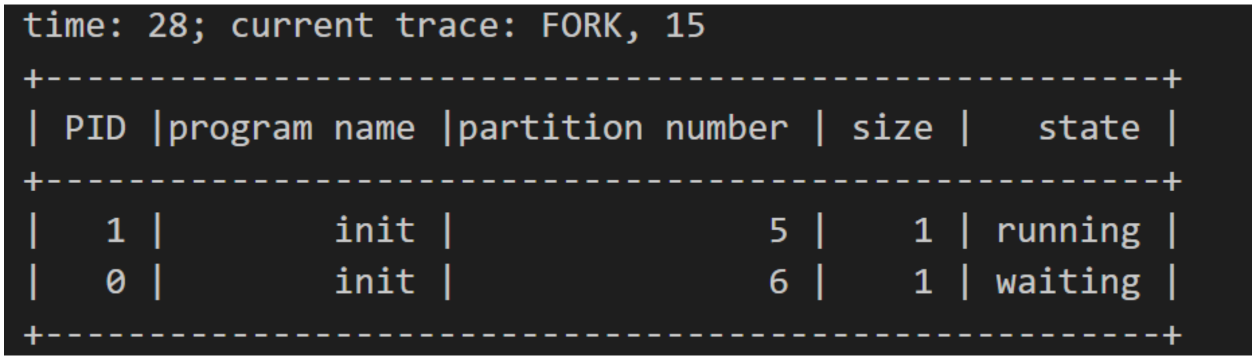
```
time: 33; current trace: FORK, 20
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
```

Simulation 4: Fork with Multiple Nested Interrupts



```
time: 25; current trace: FORK, 12
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |                5 |    1 | running |
|   0 |        init |                6 |    1 | waiting |
+------------------------------------------------------+
```

Simulation 5: Fork with CPU Burst and Interrupt Handling



```
time: 28; current trace: FORK, 15
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |                5 |    1 | running |
|   0 |        init |                6 |    1 | waiting |
+------------------------------------------------------+
```

Conclusion

These simulations show how the simulator correctly mimics how fork() and exec() work in Unix systems. Each FORK creates a new process, and each EXEC replaces a process's program in memory. The system keeps track of which partition is used, updates the PCB, and always runs the child before the parent.

Overall, the simulator behaves as expected: it creates processes, loads new programs, and shows how parent and child processes interact in a simple and clear way. It's a good way to understand how process creation and execution work inside an operating system.