Part 3: Design and Implementation of an API Simulator

SYSC 4001: Operating Systems

Group Member 1: Amreen Shahid (101306199)
Group Member 2: Samy Kaddour (101258499)

Carleton University
Nov 7th, 2025

Introduction

In this assignment, we implemented a small API simulator extending off of assignment 1 by adding new functions like fork() an exec(). The simulation tracks the execution and the system status to see how parent and child processes are created and communicated by logging each step of the way. In order to test this simulation we created a few trace files containing the input, as well as programs to process. In an external file we set the size of each program. A detailed explanation of test 1 and test 2 is provided below. Test 3, 4, and 5 follow the same principle.

TEST 1: Basic Example

trace1.txt	**program1.txt**	**program4.txt**
FORK, 10 IF_CHILD, 0 EXEC program1, 50 IF_PARENT, 0 EXEC program2, 25 ENDIF, 0	CPU, 100	SYSCALL, 4

Test 1 begins with a fork operation, which creates a new child process by copying the attributes of its parent process. Since the fork execution can only be done in kernel mode, the system switches modes and saves the current context. It will then load the address of the fork routine in the PC and the child's program control block (PCB) will be cloned. Then the schedule is called bringing it back to user mode:

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

time: 24; current trace: FORK init, 10

```
+-----------------------------------------------------+
| PID |program name |partition number | size |   state |
+-----------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+-----------------------------------------------------
```

Next, the child process will run program 1 and while that is happening, the parent process will wait for the child to finish running:

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

Once the child process is finished running, the system will switch back to kernel mode where the parent process will run program 2 for the given duration. The program will be loaded into memory and the PCB will be updated. The scheduler is called and goes back to user mode:

```
24, 1, switch to kernel mode
25, 10, context saved
35, 1, find vector 3 in memory position 0x0006
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
```

```
time: 620; current trace: EXEC program2, 25
```

```
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |    program2 |               3 |   15 | running |
|   0 |       init |               6 |    1 | waiting |
+-------------------------------------------------+
```

This test demonstrates the use of IF_CHILD and IF_PARENT to separate which process will run each program.

TEST 2: “Granchild” Process

trace2.txt	**program 3.txt**	**program4.txt**
FORK, 17 IF_CHILD, 0 EXEC program3, 16 IF_PARENT,0 ENDIF, 0 CPU, 205	FORK, 15 IF_CHILD, 0 IF_PARENT, 0 ENDIF, 0 EXEC program4, 33	CPU, 53

In this test a child process is created in the same way as test 1 by switching to kernel mode, saving the context and then loading the address of the fork routine into the PC. The PCB of the child process is then cloned and the scheduler is called returning to user mode:

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, 17, cloning the PCB
30, 0, scheduler called
30, 1, IRET

time: 31; current trace: FORK init, 17

```
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |       init |               5 |    1 | running |
|   0 |       init |               6 |    1 | waiting |
```

The child process will run program 3 for the given duration while the parent process is waiting:

time: 220; current trace: EXEC program3, 16

PID	program name	partition number	size	state
1	program3	4	10	running
0	init	6	1	waiting

The child process will perform another fork creating a “grandchild” process *(shown in program3.txt),* following the same procedure as any fork instruction:

time: 249; current trace: FORK program3, 15

PID	program name	partition number	size	state
2	program3	3	10	running
0	init	6	1	waiting
1	program3	4	10	waiting

The grandchild will execute program 4 where the child process will be waiting for its completion:

time: 530; current trace: EXEC program4, 33

PID	program name	partition number	size	state
2	program4	3	15	running
0	init	6	1	waiting
1	program3	4	10	waiting

time: 864; current trace: EXEC program4, 33

```
+-------------------------------------------------+
| PID |program name |partition number | size | state |
+-------------------------------------------------+
| 2 | program4 | 3 | 15 | running |
| 0 | init | 6 | 1 | waiting |
| 1 | program3 | 4 | 10 | waiting |
+-------------------------------------------------+
```

This test is especially helpful since it handles multiple generations of processes rather than just one child and a parent.