SYSC 4001 – Assignment 2, Part III Report

Danilo Bukvic 101297163
Oluwatobi Oluwookere 101245900

Part II link

https://github.com/tobiddaneiel/Assignment_2_SYSC_4001.git

Part III link

https://github.com/danilo-hire-me/SYSC4001_A2_P3.git

Overview

This part implements process creation (fork) and program replacement (exec) in a small simulator. The simulator uses process control blocks (PCBs), fixed memory partitions, a vector table, and an interrupt-driven flow. It writes two logs: execution.txt (a timeline of actions) and system_status.txt (snapshots of the ready/running queue). Below, each test is explained step by step using what appears in the logs and how the simulator models the real system call.

How the Simulator Models the System Calls

fork()
1 A software interrupt triggers the kernel path.
2 The simulator switches to kernel mode, saves context, and looks up the handler using the vector table.
3 The parent PCB is cloned to create the child PCB. A free partition is found for the child.
4 The scheduler chooses the child to run. The parent becomes waiting. The ISR returns to user mode.
5 The child then executes the next lines in the trace. When it finishes, control returns to the parent trace.

exec()
1 A software interrupt enters the kernel path.
2 The current program image is freed. The simulator looks up the external program and reads its size.
3 The loader time is proportional to size: 15 ms per MB in the assignment. The simulator logs loading, marking the partition, and updating the PCB.
4 The scheduler is called. The ISR returns. Execution now continues following the external program's trace.
5 The old trace does not continue. That is why the simulator stops iterating the old script after exec.

Tests

Test1

In this test, the simulator performed fork and exec behavior back-to-back multiple times within the same trace. The fork sequence shows the cloning of the PCB and the scheduler switching control. Exec replaced the running image, loaded the new program based on its size, updated memory, and continued executing the new trace. This demonstrates that the base path of fork + exec + IO continues end-to-end correctly and that the simulator kept the same PID while replacing its address space.
Log:
13, 10, cloning the PCB
23, 0, scheduler called
87, 150, loading program into memory
These lines show the fork clone step, scheduler call, and loader time proportional to program size.

Test2

Test 2 repeated fork and exec transitions with more CPU and syscall sequences afterwards. The simulator showed that the system can keep performing new exec operations after previous ones without corrupting memory allocation state. The execution log shows context switching, memory replacement, and CPU bursts after each exec call. This test confirms the simulator supports repeated sequential exec transformations on the same process.
Log:
42, 1, find vector 3 in memory position 0x0006
60, 150, loading program into memory
530, 53, CPU Burst
This shows ISR vector lookup, loader behavior again, and CPU burst continuing under new program image.

Test3

Test 3 again validates fork behaviour, the simulator clones the PCB, moves the parent to waiting, and schedules the child first. After exec occurs within the child trace, the loader steps, partition updates, and device IO all complete. This matches fork semantics: child executes first, parent resumes after recursion completes.
Log:
13, 20, cloning the PCB
33, 0, scheduler called
57, 60, Program is 10 Mb large
This demonstrates the fork clone step, scheduler context change, and exec program size evaluation.

Test4

In this test exec fails due to insufficient partition size. The simulator correctly printed failure and the test terminated without attempting to continue replacing the process image. This shows memory partition enforcement is correctly implemented and exec does not override memory rules.
Log:
13, 30, found program 'programBig' size 100 MB
43, 0, EXEC failed: no suitable partition
This demonstrates that once the simulator identified the program size, it found no partition capable of fitting the new program.

Test 5

This test validates correct parent resumption after child execution is complete. The simulator performed exec operations multiple times and still continued normal CPU bursts afterwards. This shows stable scheduler return path and memory consistency across nested exec usage.
Log:
23, 50, Program is 10 Mb large
73, 150, loading program into memory
561, 30, CPU Burst
These lines show another program replacement then normal execution resuming after that new program loads.

Status Snapshot analysis

Test 3: The system status snapshots for this test show the correct child-first fork semantics followed by a proper exec transition. Right after the fork, the table shows PID 1 as the running child and PID 0 moved to waiting, which confirms that the scheduler switches execution to the child immediately after cloning the PCB. Later when exec occurs, the system status shows the process image replaced with program1, with the size updated to 10 MB and stored into partition 4, proving that the process memory image was successfully overwritten and relocated. This behavior is consistent with Linux semantics: fork duplicates the PCB and exec replaces the running image within the same PID.

```
time: 34; current trace: FORK, 20
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+

time: 277; current trace: EXEC program1, 60
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   0 |    program1 |               4 |   10 | running |
+-------------------------------------------------------+
```

Test 4: In this test, exec was attempted with a program that was too large to fit into any available memory partition. The system status displays programBig with a partition number of -1 at the moment exec occurs, indicating that the

```
time: 43; current trace: EXEC programBig, 30
+-------------------------------------------------------+
| PID |program name |partition number | size |   state|
+-------------------------------------------------------+
|   0 |  programBig |              -1 |  100 | running|
+-------------------------------------------------------+
```

simulator detected no partition capable of holding a 100 MB program. Because exec must successfully load the program image into memory before replacing the current process address space, the failure to assign a partition means that exec cannot complete. This confirms that memory constraints are enforced by the simulator and that exec does not bypass fixed partition limits.

Test 5:In this test, the system status clearly shows that multiple exec operations can occur sequentially while maintaining memory integrity. First the process switches to program1 and is placed into partition 4, and then later switches again to program2 and is placed into partition 3. This demonstrates that the simulator correctly frees the old memory partition before allocating a new one, ensuring clean transitions between program images. It also confirms that the simulator can resume execution normally after each exec stage, showing correct process continuation after child completion and multiple sequential image replacements.

```
time: 233; current trace: EXEC program1, 50
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   0 |    program1 |               4 |   10 | running |
+-------------------------------------------------------+

time: 561; current trace: EXEC program2, 60
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   0 |    program2 |               3 |   15 | running |
+-------------------------------------------------------+
```

2. ***Only Init is in the system. Init runs this code:***

```
FORK, 17      //fork is called
IF_CHILD, 0
EXEC program1, 16  //exec is called by child
IF_PARENT,0
ENDIF, 0
CPU, 205 //CPU burst being called after the conditionals
```

Contents of program1:

```
FORK, 15
IF_CHILD, 0
IF_PARENT, 0
ENDIF, 0  //notice how nothing is happening in the conditionals
EXEC program2, 33 //which process executes this? Why?
```

Contents of program2:

```
CPU, 53
```

The child executes this exec. When fork happens, the simulator switches to run the child first. So the IF_CHILD path is executed by the child, therefore the exec(program1) occurs in the child, not the parent.

1. ***Only Init is in the system. Init runs this code:***

```
FORK,  10           //fork is called by init
IF_CHILD, 0
EXEC program1, 50 //child executes program1
IF_PARENT, 0
EXEC program2, 25   //parent executes program2
ENDIF, 0      //rest of the trace doesn't really matter (why?)
```

Because after exec, the original program image is replaced entirely, so execution can't return to these lines. exec overwrites the process code so anything after this in the trace is unreachable.

Conclusion

Across all five tests, the simulator demonstrated correct handling of fork and exec semantics: child-first execution order, memory image replacement, loader costs tied to program size, scheduler usage, and partition enforcement. The short log excerpts from each test confirm that the simulator behavior aligns with the actual OS flow described by Linux fork() and exec() system calls. Overall, the simulation behaved correctly and matched the intended specification for Part III.