SYSC 4001 Assignment 2 — Part III

Nate Babyak — 101310590
Ozan Kaya — 101322055

November 7, 2025

1 Introduction

The objective of this part of the assignment is to design and implement a small API simulator for a single-CPU system. Building on the interrupt system developed in the previous assignment, this simulator adds support for the `fork` and `exec` system calls. It models process creation, memory allocation, and program loading using fixed memory partitions.

2 Tests

2.1 Test 1

2.1.1 Analysis of `execution.txt`

The test begins with the `init` process calling `fork`, which creates a child process. The child then executes `program1` using `exec`, loading it into the 10 MB partition. After the child completes, the parent executes `program2` in the 15 MB partition. Each system call triggers an interrupt service routine (ISR), as with the configuration implemented in the previous assignment. The rest of the trace doesn't really matter because we only care about the `fork` and `exec` calls.

2.1.2 Analysis of `system_status.txt`

At time 24, the system executes `fork`, creating a child process. The child (PID 1) inherits the parent's PCB and begins running, while the parent (PID 0) is placed in the waiting state, since the simulator gives priority to child processes. Around time 256, the child executes `program1`. Using the best-fit allocation strategy, `program1` (10 MB) is assigned to partition 4 (10 MB). At approximately time 633, the parent executes `program2`, which gets partition 3 (15 MB). At this point, `init` has been replaced by `program2`.

2.2 Test 2

2.2.1 Analysis of execution.txt

The test begins with the `init` process creating a child through `fork`. The child then executes `program1`. During its execution, `program1` itself calls `fork`, creating another child process. Once `program1` completes, its parent executes `program2`. Finally, the remaining portion of `init` resumes execution in the parent process. Both processes execute `program2`.

2.2.2 Analysis of system_status.txt

Around time 31, the system executes `fork`, creating a child process. The child (PID 1) inherits the parent's PCB and begins running, while the parent (PID 0) waits. Around time 229, the child executes `program1`, which requires 10 MB and is allocated to partition 4 (10 MB) using the best-fit strategy. At time 258, `program1` executes another `fork`, creating a new process (PID 2) that also runs `program1` and is allocated partition 3. Around time 543, the child executes `program2`, completing the sequence of nested forks and executions.

2.3 Test 3

2.3.1 Analysis of execution.txt

This test begins with the `init` process invoking `fork`, which creates a child process. The parent then executes `program1`, while the child remains unchanged. The child process runs first and completes execution before the parent resumes and runs to completion.

2.3.2 Analysis of system_status.txt

At around time 34, `init` executes `fork`, which creates a child (PID 1). The child is set to run and is allocated partition number 5 using best-fit allocation. At around time 286, the child has completed and the parent executes `program1` (10 MB), and is thus allocated partition number 4 (10 MB).

2.4 Test 4

2.4.1 Analysis of execution.txt

This test starts with a `fork`, which clones the current PCB into the process. The child process inherits the memory image of `program1`. This program also invokes `fork` and the child executes `program2`. The programs run child-first, meaning `program2` first, then `program1`, and finally `program3` is executed by `init`.

2.4.2 Analysis of system_status.txt

At around time 26, `fork` is invoked, resulting in a child process (PID 1). At around time 228, the child process executes `program1`, which occupies partition number 4. At around time 252, `program1` executes `fork` itself, creating another `program1` with PID 2. This child then executes `program2`, which occupies partition number 5, as 4 is already occupied. Finally, at

around time 1791, after the two children have completed, `init` executes `program3` (10 MB), which is allocated partition number 3 (15 MB).

2.5 Test 5

2.5.1 Analysis of `execution.txt`

First, `fork` is used to create a child (PID 1), which starts running. This child executes `program1`, which then runs to completion. The parent (PID 0) then executes `program2`, which also runs to completion.

2.5.2 Analysis of `system_status.txt`

At around time 32, `init` invokes `fork`, which creates a child (PID 1) and allocates it partition number 5. At around time 274, the child process executes `program1` (12 MB) and is thus allocated partition number 3 (15 MB). After `program1` completes, the parent process executes `program2` (18 MB), which is allocated partition number 2 (25 MB). Finally `program2` runs until completion.

3 Discussion

The line `break;` inside the `exec` block is important because when a process executes an `exec` call, it replaces its current memory image with a new program. If the loop didn't break, the simulator would keep reading and executing the old trace instruction from the previous program.

4 Conclusion

In conclusion, this part of the assignment successfully demonstrated process creation and program execution in a single-CPU system with fixed memory partitions, emphasizing the behavior of `fork` and `exec` system calls and the simulator's scheduling and memory allocation mechanisms.

Appendix

- SYSC 4001 Assignment 2 — Part II
- SYSC 4001 Assignment 2 — Part III