Assignment 2 Report

SYSC 4001: Operating Systems

Ajay Srirankan (Student 2, 101313173)

Siddarth Jain (Student 1, 101304051)

Lab Section: *SYSC 4001 L2*

GitHub Link: https://github.com/sidj21/SYSC4001_A2_P3

Objective

The goal of the assignment is to simulate the behaviour of fork and exec system calls in OS. The simulator from Assignment 1 was extended, and new test cases were added and have been analyzed in this report.

Implementation

The implementation mostly revolves around the allocation of resources and flow of control between processes. This is implemented with a **current PCB** and a **wait queue.** Each PCB is also allocated in a specific **memory partition** (in the **.hpp** file). For example, the FORK system call simply copies the current process.

```
PCB child = current; // the child becomes the “running” process.
child.PID = max_pid;
//child runs
free_memory(&child);
current = wait_queue.back(); // bring parent back to running queue.
```

Since children are given priority, the parent is pushed to the **wait queue** until the child terminates (i.e. **free_memory** is invoked on the child process). Only then, the parent process is pulled from the **wait queue** and set as the current process again.

On the other hand, EXEC replaces the **current** process entirely with a new program.

```
free_memory(&current);
current.program_name = program_name;
current.size = program_size;
allocate_memory(&current);
```

The existing **memory image** of the **current** process is wiped from the memory partitions. Its PID does not change, but its PCB is updated with a new name and program size, which must now be allocated properly.

Thus, the break statement at the end of the EXEC section ensures that in the calling process, the instructions after EXEC do not run. In the example below, CPU 100 will never be executed because the memory image of the process has been replaced by **program5**.

```
EXEC program5, 10
CPU, 100
```

Note: Traces 1-3 are the same input files as given in the lab manual. To use trace 4 and trace 5, copy their contents to the main ***trace.txt*** file.

Analysis of Trace 4

From **trace4.txt**, it is evident that the child process (forked from init) will be replaced by **program6**. Meanwhile, the parent will be processing consecutive CPU and I/O bursts.

The FORK system call is received after the CPU burst (t = 46), when it looks up the address of the FORK ISR. During the ISR, it clones the PCB as described in *Implementation*, and there is the resulting system snapshot at t = 77 when fork has done executing.

```
time: 77; current trace: FORK, 18
+-------------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------------+
```

The child process inherited everything, except an incremented PID. Since partition 6 is allocated to **init**, the child process resides in partition 5. Plus, the child process has priority, it runs everything until the **IF_PARENT** or **ENDIF** line. In this case, it needs to run **EXEC program6, 25**. The bottleneck in executing the command was the loader, as evident by the trace **115, (300), loading program into memory**. Since the child process called EXEC, its PCB is updated (*not process 0's PCB*). and it jumps to memory partition 2 due to additional requirements. As mentioned before, the **CPU, 50** after **EXEC** never gets executed since **program6** takes over.

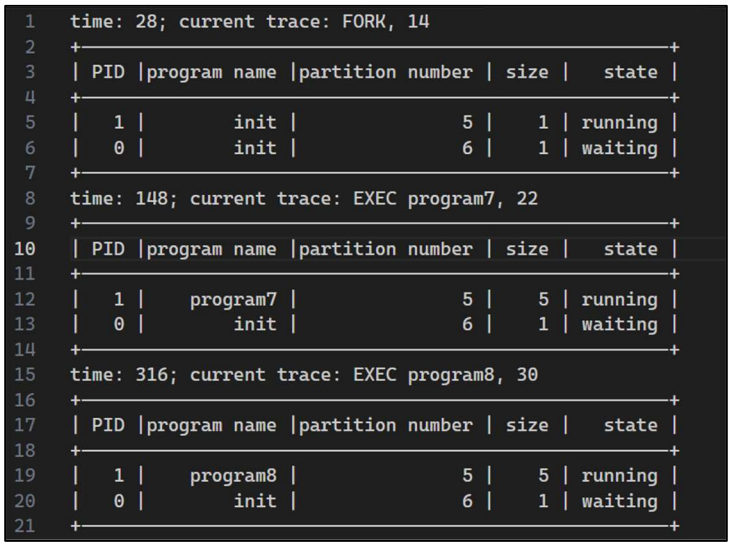
```
time: 425; current trace: EXEC program6, 25
+-------------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------------+
|   1 |    program6 |               2 |   20 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------------+
```

After EXEC, we can see **program6** running SYSCALL ISR's 5 and 7. These were implemented as recursive calls to allow the child to fully execute its own trace before returning. Once the child (*PID 1*) completes, control returns to the parent (*PID 0, init*) which executes SYSCALL ISRs 12 and 4. Since there is no duplication of the instructions, the simulator differentiated the tasks of the two processes. Plus, after the **ENDIF**, the remaining SYSCALL and CPU

burst have be executed by *PID 0 (init)* since after the recursive calls, the child has been freed.

Analysis of Trace 5

trace5.txt validates the recursive handling of EXEC calls inside programs.



```
time: 28; current trace: FORK, 14
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
time: 148; current trace: EXEC program7, 22
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program7 |               5 |    5 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
time: 316; current trace: EXEC program8, 30
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program8 |               5 |    5 | running |
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
```

We follow the same concept as *trace4.txt* until t = 148 where the forked child process is loaded with a new program. In this case, **program7.**

CPU, 40 148, 40, CPU Burst 188, 1, switch to kernel mode 189, 10, context saved 199, 1, find vector 3 in memory position 0x0006

After **program7's** CPU burst, there is another exec system call, **EXEC program8, 30**. The CPU switches context and finds the ISR for **EXEC** (memory position 3). The loader loads it within 75ms since the program is 5Mb large (15 Mb/s * 5Mb).In other words, the child that initially got forked from FORK, 14 has gone through two transitions. It has kept its PID of 1, which confirms that exec only replaces the **memory image** of the current running process.

CPU executes the CPU, 50 from **program8** as stated in the execution log **316, 50, CPU Burst.** Since the child has had priority during all its transitions, the parent instructions encoded within **IF_PARENT** never ran. But at t = 316, the child's PCB is deallocated and the parent process gets a chance to run its 95ms CPU burst.