Part III - Report and Part II Descriptions

Student 1: Wenxuan Han 101256669
Student 2: Tony Yao 101298080
2025-11-07
Github Repository Links:
https://github.com/toast23/SYSC4001_A2_P2 https://github.com/toast23/SYSCC4001_A2_P3

Part 2: Descriptions of FORK and EXEC

FORK:
The fork system call can be invoked to create a concurrent child process to an existing parent process. If the system call is accepted, the kernel will create a new process by duplicating the entirety of the parent's PCB information (PC, state, files, memory limits, etc.) and assigning a separate PID and address space. Both processes run independently and continue running from the previous line (which is fork). In order to distinguish the two processes, the parent process running fork() will return the child's PID while the child running fork() will return 0.

EXEC:
In the view of the operating system, exec is used to replace the current process with a new one from an executable file. And it basically replaces the entire content of the current process with a new program, then loads the program into the process space and runs it from the entry point. From the coding part in the above picture, the function of exec () basically loads the content of process two from another executable file and replaces the current process with the new process 2 into the current executable file. In this case, process 2 starts to launch while we are using the command of./process2 to execute process 2 from another executable file. And also, process 2 is a child process of process one while we also check that the value of PID is 0. Thus, the exec () function is often used in sequence to get a new program running as a child of a current process.

Test Cases:

Test Case 1 (Mandatory)

Trace Analysis:

```
FORK, 10       // parent process (init) calls fork
IF_CHILD, 0            // if child process do the following
EXEC program1, 50   // free child process memory and replace with program1
IF_PARENT, 0           // if parent process do the following
EXEC_program2, 25  // free parent process memory and replace with program2
ENDIF, 0  // note the rest of the trace after this line does not matter as the parent process
```

memory would be completely replaced with program 2 at this point.

Program 1 Explanation:

CPU, 100 // activate CPU burst

Program 2 Explanation:

SYSCALL, 4 // call SYSCALL 4

FORK, EXEC, AND SYSCALL are all privileged operations meaning they will undergo the following base steps:

1. Switch to kernel mode
2. Save context
3. Find vector x in vector table
4. Load address into PC
5. Run ISR
6. Call scheduler
7. IRET

The extra steps in fork and exec can be observed in the execution.txt document under output_files:

- 13, 10, cloning the PCB // fork includes the step of cloning the PCB info from parent to child and assigning a new PID + address space → see line 5 output_files > test1 > execution.txt
- 37, 50, Program is 10 Mb large // exec finds the size of the program by accessing the program data in external_files.txt
- 87, 150, loading program into memory // load program into memory (loading time is 15 ms per Mb
- 237, 3, marking partition as occupied
- 240, 6, updating PCB

system_status.txt describes the states of the processes after the execution of fork or exe:

```
time: 24 current trace: FORK, 10
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------+
time: 247 current trace: EXEC program1, 50
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------+
time: 645 current trace: EXEC program2, 25
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   0 |    program2 |               3 |   15 | running |
+-------------------------------------------------+
```

The table reveals that fork gives spriority to child processes as after the parent executes fork, it is moved to the waiting queue while the child process runs. The effects of exec are observed as well in the second output of EXEC program1, 50 where the child's process is replaced with program 1. Note that the PID remains the same meaning it's the same process with different instructions. This is evident with the change in partition number from 5 to 4 as the partitions are ordered from largest to smallest (1-5). The new program has a size of 10 which causes the malloc function to move the process to partition 4 for more space.

Test Case 2 (Mandatory)

Trace Analysis:

```
FORK, 17       // most of activity consistent with test case 1
IF_CHILD, 0
EXEC program1, 16
IF_PARENT 0
ENDIF, 0
CPU, 205       // CPU burst called after conditionals
```

Program 1:

```
FORK, 15
IF_CHILD 0
IF_PARENT 0
ENDIF, 0
EXEC program2, 33            // program2→CPU, 53
```

Test case 2 is similar to test case 1 with differences in the conditional statements. From the parent view, the rest of the trace does matter in this case as it does not call exec which allows it to retain its memory. Test case 2 deviates in conditionals even more in program1 where the child creates its own child from fork and then nothing happens in any of the conditionals. Both child 1 and child 1's child retain memory and proceed to run the rest of the trace. Therefore, both child 1 and child 2 (child 1's child) execute the line EXEC program2, 33.

This is evident in system_status output where child 1 and 2 (PID 1 and 2) change name to program2:

```
time: 581 current trace: EXEC program2, 33
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   2 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+-------------------------------------------------------+
time: 932 current trace: EXEC program2, 33
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+
```

Test Case 3 (Mandatory)

Trace Analysis:

FORK, 20
IF_CHILD, 0
IF_PARENT, 0 // conditional for parent meaning child retains memory and runs post-ENDIF trace (CPU burst) whereas parent runs program 1
EXEC program1, 60
ENDIF, 0
CPU, 10

Program 1: // CPU/IO process
CPU, 50
SYSCALL, 6
CPU, 15
END_IO, 6

Test case 3 uses a different conditional path where the child process continues running the remaining trace whereas the parent process is replaced with program 1.

Test Case 4 (Error testing: FORK)

Trace Analysis:

FORK, 10
FORK, 10
FORK, 10
FORK, 10
FORK, 10

Test case 4 explores the event of partition overload. This is observed in the system_status output:

```
time: 96 current trace: FORK, 10
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |        init |               2 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
|   0 |        init |               6 |    1 | waiting |
|   0 |        init |               6 |    1 | waiting |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+
time: 120 current trace: FORK, 10
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |        init |               1 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
|   0 |        init |               6 |    1 | waiting |
|   0 |        init |               6 |    1 | waiting |
|   0 |        init |               6 |    1 | waiting |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+
*****fork ERROR! Memory allocation failed!*****
```

As each fork line is executed, we already start to see errors with multiple processes sharing the same PID and partition number. The system fails after a fork at 120 ms because all partitions are in use. This is because the number of processes is doubled each time a new fork is called → create child, copy info, and run from PC. Therefore, even though we only called fork 5 times, the amount of processes is doubled each call resulting in 32 processes (2^5) which is well over partition capacity.

Test Case 5 (Error testing: EXEC)

Trace Analysis:

SYSCALL, 2
EXEC missing_program, 2

Test case 5 looks at the system response when trying to replace process memory with an unregistered program (not listed in external_files):

```
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, 150, SYSCALL ISR
163, 30, transfer data from device to memory
193, 20, check for errors
213, 1, IRET
214, 1, switch to kernel mode
215, 10, context saved
225, 1, find vector 3 in memory position 0x0006
226, 1, load address 0X042B into the PC
227, 50, Program is 4294967295 Mb large
*****exec ERROR! Memory allocation failed!*****
```

The SYSCALL runs first as normal and then the parent process tries to run EXEC missing_program. It is able to run the base steps of privileged instructions like switching mode, saving context, and driving PC but ends in a memory allocation error after getting program size. Interestingly, the system is able to get the program size as it yields an extremely large but familiar value. This can be attributed to the default value initialized in the get_size() function:

```cpp
// Searches the external_files table and returns the size of the program
unsigned int get_size(std::string name, std::vector<external_file> external_files) {
    int size = -1;

    for (auto file : external_files) {
        if(file.program_name == name){
            size = file.size;
            break;
        }
    }

    return size;
}
```

Size is set to -1 by default and only changes once the program name is identified in external_files. However, since we have an unregistered program name, size is never reassigned and returns as -1.

The 2's complement is used to represent negative numbers which is done by performing bitwise NOT and adding 1 to the end of the value. Most modern systems allocate 32 bits to int (therefore this result could vary with system models!) meaning in order to represent -1, all 32 bits must be set to 1. Note that the get_size function returns an unsigned integer causing the compiler to take the value (1111 1111 1111 1111 1111 1111 1111 1111) literally without converting from the 2's complement which yields the max possible value of (2^32) - 1 = 4,294,967,296.