SYSC4001
Assignment 2

Report

Eric Cui – 101237617

Aydan Eng – 101295820

Part 2 GitHub

Part 3 GitHub

Objective

This assignment is an extension of the work from assignment 1. The objective is to create an extension by implementing fork/exec functions in the simulator, as well as taking account of partition memory limits. The output aim to capture the partition change throughout the execution as well as the CPU usage from the previous assignment.

Analysis

To validate our simulator's process and memory management, six simulations were run with variations in programs instruction, trace file, and external files. From the analyzing the execution and system status files, we found that the partitions with lower sizes are more frequently occupied. This matches with our design goal to use the best fit for every processes. This also showed its usefulness in scenarios where later processes require more partition size than prior processes. If the system uses other algorithms such as first-fit, the prior processes may occupy a larger partition, which may prevent the later processes from getting the partition they need. The execution file is updated as well. With the addition of the partition, the CPU requires additional task of assigning each process with a partition, leading to a bit more overhead compared to assignment 1.

Test 1 (`execution.txt and system_status.txt`):

```
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
24, 1, IRET
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
247, 100, CPU Burst
347, 1, switch to kernel mode
348, 10, context saved
358, 1, find vector 3 in memory position 0x0006
359, 1, load address 0X042B into the PC
360, 25, Program is 15 Mb large
385, 225, loading program into memory
610, 3, marking partition as occupied
613, 6, updating PCB
619, 0, scheduler called
619, 1, IRET
620, 1, Switch to kernel mode
620, 4, context saved
620, 1, find vector 4 in memory 0X0292
620, 1, obtain ISR address
620, 39, Call device driver
620, 109, Perform device check
620, 102, Send device instruction
877, 1, IRET
```

```
+-------------------------------------------------------+
| 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: 620; current trace: EXEC program2, 25
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   0 |    program2 |               3 |   15 | running |
+-------------------------------------------------------+
```

This test validates the fundamental fork() and exec() process replacement pattern. The parent process (PID 0) forks, creating a child (PID 1). Following the child-first priority rule, the child

executes first, immediately calling exec to replace its image with program1. Upon program1's termination, the original parent process resumes and executes the exec for program2. This sequence demonstrates how exec terminates the execution flow of the calling trace file, ensuring the code after the ENDIF is never reached by either process.

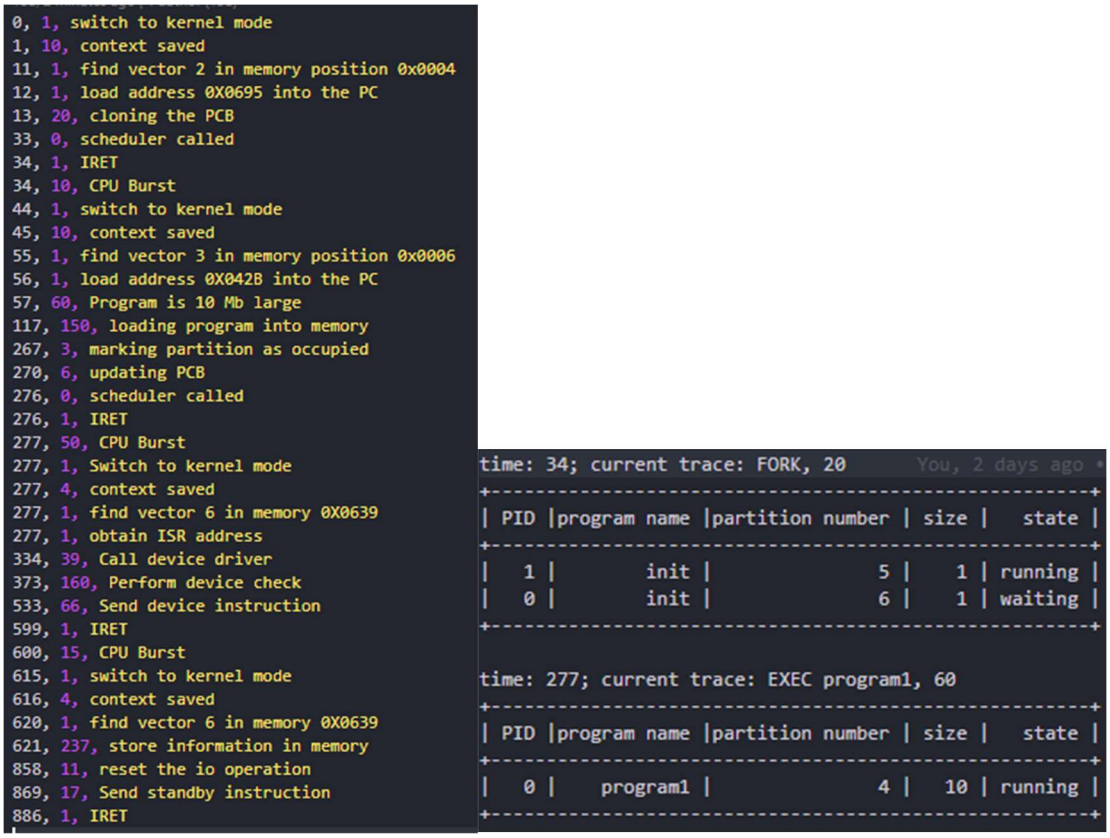
Test 2 (`execution.txt and system_status.txt`):

```
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
31, 1, IRET
31, 1, switch to kernel mode
32, 10, context saved
42, 1, find vector 3 in memory position 0x0006
43, 1, load address 0X042B into the PC
44, 16, Program is 10 Mb large
60, 150, loading program into memory
210, 3, marking partition as occupied
213, 6, updating PCB
219, 0, scheduler called
219, 1, IRET
220, 1, switch to kernel mode
221, 10, context saved
231, 1, find vector 2 in memory position 0x0004
232, 1, load address 0X0695 into the PC
233, 15, cloning the PCB
248, 0, scheduler called
249, 1, IRET
249, 1, switch to kernel mode
250, 10, context saved
260, 1, find vector 3 in memory position 0x0006
261, 1, load address 0X042B into the PC
262, 33, Program is 15 Mb large
295, 225, loading program into memory
520, 3, marking partition as occupied
523, 6, updating PCB
529, 0, scheduler called
529, 1, IRET
530, 53, CPU Burst
583, 1, switch to kernel mode
584, 10, context saved
594, 1, find vector 3 in memory position 0x0006
595, 1, load address 0X042B into the PC
596, 33, Program is 15 Mb large
629, 225, loading program into memory
854, 3, marking partition as occupied
857, 6, updating PCB
863, 0, scheduler called
863, 1, IRET
864, 53, CPU Burst
917, 205, CPU Burst
```

```
time: 31; current trace: FORK, 17
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------+

time: 220; current trace: EXEC program1, 16
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------+

time: 249; current trace: FORK, 15
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   2 |    program1 |               3 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+-------------------------------------------------+

time: 530; 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: 864; current trace: EXEC program2, 33
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------+
```

This test demonstrates nested process creation and shared code execution. The initial fork creates Child 1 (PID 1), which execs program1. Inside program1, a second fork occurs, creating Child 2 (PID 2) and Parent 1 (PID 1). Both processes then execute the instruction EXEC program2 because it appears after the ENDIF statement in program1's trace. This confirms that all code following a conditional block (ENDIF) is executed by both the parent and child processes that spawned from the most recent FORK.

Test 3 (`execution.txt and system_status.txt`):

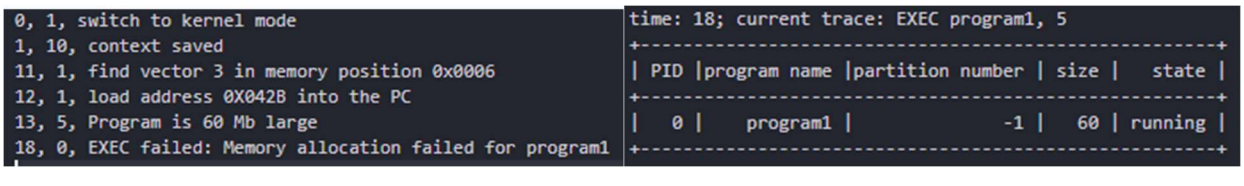


```
0, 1, switch to kernel mode
1, 10, context saved
11, 1, find vector 2 in memory position 0x0004
12, 1, load address 0X0695 into the PC
13, 20, cloning the PCB
33, 0, scheduler called
34, 1, IRET
34, 10, CPU Burst
44, 1, switch to kernel mode
45, 10, context saved
55, 1, find vector 3 in memory position 0x0006
56, 1, load address 0X042B into the PC
57, 60, Program is 10 Mb large
117, 150, loading program into memory
267, 3, marking partition as occupied
270, 6, updating PCB
276, 0, scheduler called
276, 1, IRET
277, 50, CPU Burst
277, 1, Switch to kernel mode
277, 4, context saved
277, 1, find vector 6 in memory 0X0639
277, 1, obtain ISR address
334, 39, Call device driver
373, 160, Perform device check
533, 66, Send device instruction
599, 1, IRET
600, 15, CPU Burst
615, 1, switch to kernel mode
616, 4, context saved
620, 1, find vector 6 in memory 0X0639
621, 237, store information in memory
858, 11, reset the io operation
869, 17, Send standby instruction
886, 1, IRET
```

```
time: 34; current trace: FORK, 20      You, 2 days ago
+-------------------------------------------------------+
| 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 |
+-------------------------------------------------------+
```

This scenario specifically validates the conditional block skip logic. The child process (PID 1) executes first, finding an empty IF_CHILD block. It then correctly skips the IF_PARENT block entirely before executing the shared CPU, 10 instruction outside the conditional structure. After the child terminates, the parent process (PID 0) resumes and executes the code within the IF_PARENT block (the EXEC program1 call), demonstrating that the parent/child processes only execute their respective sections and the shared post-ENDIF instructions.

Test 4 (`execution.txt and system_status.txt`):



```
0, 1, switch to kernel mode
1, 10, context saved
11, 1, find vector 3 in memory position 0x0006
12, 1, load address 0X042B into the PC
13, 5, Program is 60 Mb large
18, 0, EXEC failed: Memory allocation failed for program1
```

```
time: 18; current trace: EXEC program1, 5
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   0 |    program1 |              -1 |   60 | running |
+-------------------------------------------------------+
```

This test validates the memory allocation failure mechanism during an EXEC call. The running process attempts to execute program1 (60 Mb). The system successfully performs the interrupt boilerplate (0 to 13 ms) and finds the program size. However, the subsequent allocate_memory function fails because the required 60 Mb cannot fit into any available partition. Crucially, the process's memory is immediately freed at the start of EXEC. Since a new partition cannot be assigned, the simulator logs EXEC failed: Memory allocation failed for program1 and the process

terminates (as indicated by the function returning at 18 ms), demonstrating resource exhaustion as a valid termination condition.

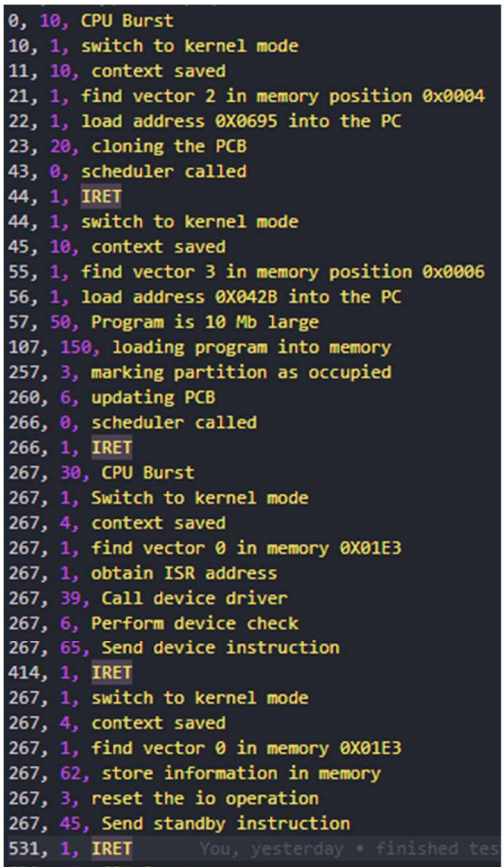
Test 5:

```
0, 10, CPU Burst
10, 1, switch to kernel mode
11, 10, context saved
21, 1, find vector 3 in memory position 0x0006
22, 1, load address 0X042B into the PC
23, 50, Program is 10 Mb large
73, 150, loading program into memory
223, 3, marking partition as occupied
226, 6, updating PCB
232, 0, scheduler called
232, 1, IRET
233, 20, CPU Burst
253, 1, switch to kernel mode
254, 10, context saved
264, 1, find vector 3 in memory position 0x0006
265, 1, load address 0X042B into the PC
266, 60, Program is 15 Mb large
326, 225, loading program into memory
551, 3, marking partition as occupied
554, 6, updating PCB
560, 0, scheduler called
560, 1, IRET
561, 30, CPU Burst
```

```
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 |
+-------------------------------------------------------+
```

This test validates successful nested EXEC calls and the dynamic memory reallocation for a single process. The process (PID 0) first executes EXEC program1, which is successfully allocated to a partition. Immediately after, program1 executes EXEC program2. This second EXEC correctly frees the 10 Mb partition and successfully allocates the process to a larger 15 Mb partition. The simulation confirms the process replacement logic: upon the completion of program2, the execution flow returns and breaks the program1 loop (abandoning CPU, 888), and the main loop is also broken (abandoning CPU, 999), showing that the exec call never returns to its calling trace file.

Test 6:



```
0, 10, CPU Burst
10, 1, switch to kernel mode
11, 10, context saved
21, 1, find vector 2 in memory position 0x0004
22, 1, load address 0X0695 into the PC
23, 20, cloning the PCB
43, 0, scheduler called
44, 1, IRET
44, 1, switch to kernel mode
45, 10, context saved
55, 1, find vector 3 in memory position 0x0006
56, 1, load address 0X042B into the PC
57, 50, Program is 10 Mb large
107, 150, loading program into memory
257, 3, marking partition as occupied
260, 6, updating PCB
266, 0, scheduler called
266, 1, IRET
267, 30, CPU Burst
267, 1, Switch to kernel mode
267, 4, context saved
267, 1, find vector 0 in memory 0X01E3
267, 1, obtain ISR address
267, 39, Call device driver
267, 6, Perform device check
267, 65, Send device instruction
414, 1, IRET
267, 1, switch to kernel mode
267, 4, context saved
267, 1, find vector 0 in memory 0X01E3
267, 62, store information in memory
267, 3, reset the io operation
267, 45, Send standby instruction
531, 1, IRET       You, yesterday • finished tes
532, 15, CPU Burst
0, 1, Switch to kernel mode
0, 4, context saved
0, 1, find vector 8 in memory 0X06EF
0, 1, obtain ISR address
0, 167, Call device driver
0, 729, Perform device check
0, 104, Send device instruction
1554, 1, IRET
0, 1, switch to kernel mode
0, 4, context saved
0, 1, find vector 8 in memory 0X06EF
0, 476, store information in memory
0, 13, reset the io operation
0, 511, Send standby instruction
2561, 1, IRET
```

```
time: 44; current trace: FORK, 20
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------+

time: 267; current trace: EXEC program1, 50
+-------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------+
```

This complex test validates the concurrent execution of a child's program replacement and a parent's I/O block. The initial fork creates a child (PID 1), which immediately executes EXEC program1. The parent (PID 0) is logically placed in a waiting state on the wait_queue. The child executes program1's trace, which includes its own internal I/O operations (SYSCALL/END_IO). Upon the child's completion, the parent resumes its trace, executes the SYSCALL/END_IO block, and then terminates. The system status logs confirm the correct state transitions: PID 0 moves from running to waiting during the child's execution and PID 1 transitions from the initial init program to program1, successfully demonstrating a real-world scenario where the child executes a new application while the parent remains operational for its own tasks.

Discussion

//rest of the trace doesn't really matter (why?)

When a program runs an exec() function, the program runs the respected process while immediately ending the current process. In the example code where this question is asked, the process is forked, where each instance of the fork is accepted by one of the if statements. Both instances resulting in an exec function, which replace itself with another processes, therefore it is impossible for the PCB to reach the line where the comment is on.

//which process executes this? Why?

The content of program 1 has two if statements before the EXEC function. However, none of the if statements have any exec function to interrupt the process. Therefore, both the parent and the child will execute program 2.

`break; // why is this important? (answer in report)`

The break; statement is important because it simulates the core, defining behavior of the exec system call. In a real operating system, when a process calls exec, its current program is completely replaced by a new program. If the exec call is successful, the original program's code stops executing forever, and the new program starts running. The exec call never returns to the original code. Without the break; After Program2 finished, the for loop would just continue to the next line of Program1's trace file (i.e., the instruction after EXEC). This would be incorrect, as it would be like exec was just a function call.

Conclusion

This assignment successfully extended the system simulator to model essential OS process and memory management concepts, specifically the fork() and exec() system calls. By implementing a fixed-partition memory scheme with a Best Fit strategy, the simulator accurately demonstrated how processes are created, replaced, and allocated resources. Analysis of the simulation logs confirmed that the memory strategy minimized fragmentation, while also quantifying the inherent CPU overhead introduced by kernel operations like PCB manipulation and program loading, thereby providing a verifiable model of real-world operating system resource management.