

Assignment 2: Process Management and Memory Allocation Analysis Report

Esli Emmanuel Konate 101322259 and Nitish Grover 101324174

Carleton University

SYSC 4001 : Operating Systems

## **Introduction**

Modern operating systems make use of process management mechanisms in order to handle execution of programs, allocation of resources, and process hierarchies efficiently. It is important to understand the performance aspects of creation of processes using (FORK) and replacement of processes using (EXEC) for optimization of systems. This report analyzes the performance impact of implementing FORK and EXEC system calls in an interrupt-driven simulator. It focuses mainly on how allocation occurs and how to manage/order processes.

The main objective of the analysis is to measure how multiple processes, especially the combination of FORK and EXEC operations can affect the overall execution time of the system. We conducted experiments with multiple test scenarios to provide insights into the exchanges between overhead of creation of processes, memory fragmentation, and system throughput.

## **HOW**

The simulator builds up on Assignment 1 interrupt handler with process management options.

Process creation using FORK, creates a child process with PCB cloning, it has ~10-20ms variable. In addition, Process replacement using (EXEC), replaces the process image with a new program with variable ISR time. There is also memory management, which in this case is a fixed partition scheme with smallest-fit allocation. Now to order processes (Process hierarchy), relationships between child and parent are made through wait queue management. To add, the program loads with 15ms per MB loading time from the external storage. Finally, there is context switch with 10ms overhead which comes from the first assignment.

Following that, we created 5 different test files to analyze various process patterns. First, it was trace.txt, which has a full FORK/EXEC pattern with both child and parent executing new programs. In addition, trace2.txt has Child EXEC with parent CPU carrying on. Then we have trace3.txt, which has parent EXEC after the child is done with simple CPU burst. There is also trace4.txt, which has simple FORK with CPU bursts only and no EXEC. Finally, there is trace5.txt with FORK/EXEC with various program sizes (program 3 and program 4).

In addition to the variations in trace, we also analyzed the following parameters. The first one is FORK overhead timing, which ranges from 10ms to 20ms for PCB cloning. Then we also have EXEC ISR duration, which is variable from 16ms to 60ms depending on the trace. In addition, there are also memory allocation patterns with tracking of partition usage and also fragmentation. Finally, process order depth with parent/child relationships which are only single level here.

## **Execution Analysis**

Test Case 1: trace.txt

This test shows the lifecycle of FORK/EXEC. At 0ms, init meets FORK and switches to kernel mode for 1ms and saves the context for 10ms. Then it looks up vector 2 for 1ms and loads the ISR address for 1ms. Afterwards, it clones the PCB for 10ms. At 23ms, the scheduler is then called and IRET returns the control. This creates two processes, PID 1 which is the child running inside partition 5 and PID 0, the parent waiting in partition 6.

At 24ms, the child meets EXEC program1, which starts another kernel mode switch and ISR execution for 50ms. The system finds out program1 is 10MB and loads it in partition 4 for 150ms ( $10 \text{ MB} \times 15\text{ms/MB} = 150\text{ms}$ ). It then marks the partition as busy for 3ms and updates the PCB for 6ms. Arriving at 257ms, the child has now been replaced with program1 in partition 4. The child executes program1's trace (50ms CPU + SYSCALL + 15ms CPU + END\_IO) and it ends at 880ms.

The parent then continues and meets EXEC program2. It will do the same EXEC steps with a 15MB program (25ms ISR + 225ms loading). The parent is then replaced with program2 in partition 3 by time 1168ms. The parent then executes the program2's 53ms CPU burst and finishes at time 1221ms. The parent has waited 880ms for the child to finish and it shows how the child has priority here.

**Results****Process Creation and Replacement Influence**

Test File	FORK Overhead (ms)	EXEC Operatons	Total Time (ms)	System Overhead (%)
trace.txt	10	2 (child + parent)	1221	42.3
trace2.txt	17	1 (child only)	1058	38.8
trace3.txt	20	1 (parent only)	910	35.2
trace4.txt	12	0	136	26.5
trace5.txt	15	2 (different sizes)	1709	44.6

The table perfectly shows that tests with two EXEC operations (trace.txt and trace5.txt) have the highest system overhead at 42-45%, while the simple FORK without EXEC (trace4.txt) has the lowest at 26.5%. This demonstrates that EXEC operations contribute a lot more overhead than FORK alone.

### Memory Allocation Analysis

Test File	Partitions Used	Memory Utilization (MB)	Internal Fragmentation (MB)
trace.txt	P6 -> P5 -> P4 -> P3	28/100	12
trace2.txt	P6 -> P5 -> P4	13/100	2
trace3.txt	P6 -> P5 -> P4	13/100	2
trace4.txt	P6 -> P5	3/100	1
trace5.txt	P6 -> P5 -> P3 -> P5	22/100	6

The smallest-fit allocation makes use of partitions 5 and 6 for init processes (1MB), other larger programs make use of partitions 3 and 4. Memory utilization is between 3% to 28%, with a large amount of unused capacity in partitions 1 and 2 (40MB and 25 MB)

### Process Order Performance

Test Pattern	Parent wait Time (ms)	Child Execution Time (ms)	Context Switches
Child EXEC -> Parent EXEC	856	856	4
Child EXEC -> Parent CPU	822	822	3
Child CPU -> Parent EXEC	10	10	2
Simple FORK	50	50	1
Two Different EXEC	873	873	4

Parent wait time is directly linked with the complexity of a child execution. Tests which have child EXEC operations force the parents to wait for nearly 800ms or more, while the simple CPU bursts only delay the parents for between 10-50ms

## **Analysis**

After going through these tests, we found out four important aspects about process management overhead.

First, EXEC operations take control of execution time compared to FORK. While FORK adds a number between 24-34ms overhead (interrupt handling + the PCB cloning + the scheduler), EXEC also adds overhead based on the program size. So, base ISR (16-60ms) + program size + (15ms x MB) loading + 9ms memory operations. Therefore, a 15 MB program, EXEC adds nearly 250ms total overhead against FORK's which is nearly 30ms.

Second, the smallest-fit allocation creates memory usage patterns which are predictable but also not the most optimal. Small init processes, for example 1MB can occupy consistently the smallest partitions (P5, P6), and this allows to keep the larger partitions for bigger programs. But, this can lead to nearly 72-97% of memory which is not utilized through all the tests, with partitions 1 and 2 it gives a combination of 65 MB which is never used.

Third, Process order can strongly impact total execution time. The fact that the child runs first means that the parent processes will have delays which are equal to their children's entire execution time. In trace.txt, the parent waits 856ms while the child completes, this doubles the sequential execution time. This shows why real systems use preemption in scheduling rather than just running to completion, which is not ideal.

Finally, The break statement after EXEC is important. Without it, Test 1 would just incorrectly continue executing the original trace after process replacement, which would add nearly 205ms of bad CPU burst time. Test 5 would also be the same in terms of executing the wrong instructions, and this would lead to corruption of the process order/hierarchy. The 10-20% execution time difference really shows how important it is to have good process replacement implementation.

### **How does FORK/EXEC overhead compare to interrupt handling overhead?**

If we compare the results of A02 with the results of A01 in terms of interrupt handling. It shows that process management adds an extra part of overhead. In A01, we found out that interrupt handling could potentially add 5 to 15 % overhead through context switch. In A02, FORK/EXEC operations add 26 to 45% overhead which is almost the triple of the impacts.

In addition, overheads are also greatly different. Interrupt handling in A01, had fixed overhead per interrupt which were, context switch + ISR + IRET. But in A02, the overhead varies based on the type of operation and the size of the program.

So, for jobs with frequent process creation like shell scripts, FORK/EXEC overhead take over performance, which is a little bit like I/O delays which were dominant in A01. A system which runs like 10 FORK/EXEC operations per second would spend between 300-500ms just on process management.

## **Conclusion**

This analysis measure the performance impacts of FORK and EXEC system calls in our extended interrupt simulator. We found out that EXEC operations can contribute 2 to 3 times more overhead than FORK operations, with program loading time which increases in a linear way with the size (like 15ms/MB). The smallest fit memory allocation also gives somewhat of a behavior which can be predicted but unfortunately, it is also not optimal. In order to solve this, the use of dynamic allocation would be great.

In addition, process order with children running before the parents doubles the execution time for parent processes. This confirms why real systems use preemption. To add, the importance of having a break statement after EXEC shows how process replacement must end the original execution path.

Our results show that even though interrupt handling in A01 adds good overhead from nearly 5 to 15%, process management operations can go up to nearly 26 to 45% of execution time. Just like A01 showed that I/O speed is the main bottleneck in the system, A02 showed how process creation and replacement are important factors to take into account.

KEY to the repository : <https://github.com/J3Sli3/SYSC4001>