SYSC 4001 Assignment 2 Report

Group Members: Aryan Kumar Singh (101299776), Eren Unal (101187793)

GitHub Repos: https://github.com/CryptidFiles/SYSC4001_A2_P2
https://github.com/CryptidFiles/SYSC4001_A2_P3

Mandatory Test 1

In this simulation, the system begins with only the init process. The trace starts with a fork system call. The call switches the CPU to the Kernel Mode and saves the current context. It also locates the interrupt vector and loads the address of the fork interrupt service routine (ISR). After that, the Kernel copies the parent's Process Control Block (PCB) to create a new child process with a new Process ID (PID). The kernel allocates the memory for the new child process. The parent is moved to the waiting queue. After the system returns to the user mode, the child process is scheduled to run first.

The child process executes exec program1. After this, the CPU switches to Kernel Mode. The exec retrieves the program size (10 MB) and loads the program into memory at 15 ms per MB. It marks the memory partition as occupied and updates the PCB. After that, the process returns to user mode as program1. The child process executes the contents of program1, performing a 50 ms CPU burst, a system call and an end_io interrupt. When the child process is done, the system resumes the parent process.

The parent executes exec program2 and follows the same approach as the child process. The Kernel loads the program into memory and updates the PCB. After that, the system returns to the user mode. The parent is now replaced by program2. It executes its program consisting of a CPU burst. After both programs are finished, only program2 remains active in the system.

The comment "rest of the trace doesn't really matter" refers to the effects of the exec call. In both real operating systems and in this simulation, exec replaces the current process image entirely. This means no instructions from the original trace are no longer executing. The simulator does this by executing the new program's trace and then breaking out of the loop after exec. The simulation models fork and exec. The child runs program1 and the parent runs program2. After both replacements, only program2 remains active.

Mandatory Test 2

The system begins with init. The trace begins with a fork that switches to Kernel Mode. It saves the context and searches for vector 2. After that, it enters a fork handler where the PCB is cloned for 17 ms. The scheduler then selects the child to run and returns with IRET. The child executes exec program3 causes the system to enter the Kernel Mode via vector 3. We can see the program in the logs as "Program is 10 Mb large". The child process is replaced by program3. Exec replaces the current process image entirely, partition marked occupied and PCB is updated. After this, execution continues with program3's trace. Inside program3, there is

another fork with empty child and parent conditionals. Because of that block is empty, the child's second fork has no trace to run and does not execute exec call. Parent of this second fork executes exec program4. Exec replaces the running program3 process with program4. The kernel loads the new process image, marks the partition as occupied and updates the PCB. After these it returns via IRET. Finally, because the init never executed its own exec call, the trace remains active after the IRET.

Exec program4 is executed by the parent of the second fork. The exec program4 line is outside of the if_child/if_parent blocks and the simulator only gives the new child the statements inside if_child. Because of those blocks being empty, the inner child has no work and the execution continues on the parent's path and reaches program4, which replaces the parent.

Mandatory Test 3

The system begins with only the init process. When fork is executed, the CPU switches to Kernel Mode. It saves the context, searches the fork vector and copies the PCB to create a new child process. This work takes 20 ms. After that, the scheduler runs and the system returns to user mode. The system status at time 34 shows the child process running and the parent process waiting. The child process executes a short 10 ms CPU burst before control returns to the parent process. The parent executes exec program5. The system goes into Kernel Mode, loads the vector for exec and identifies program5 as 10 MB in size. It loads the program5 into the memory, marks the partition as occupied and updates the PCB. After doing all these, the system returns to the user mode. At time 277, the status confirms that the parent has been replaced by program5.

After program5 runs, it follows its own trace. First, it does a 50 ms CPU burst and a SYSCALL. After that, it performs a 15 ms CPU burst and an end_io interrupt by the ISR. Each interrupt correctly shows vector searching, ISR execution and return with IRET. The remaining instruction after endif executes because only the parent process executed an exec call which means the init trace still continues after the conditional block.

Overall, test 3 demonstrates a simple fork where the child process executes some work and the parent process executes an exec that replaces itself with program5. The final state shows only program 5 running. This confirms that the parent process's image was successfully replaced while the child process is done after completing its CPU burst.

Created Test 1

The utilization of three programs creates a sophisticated process hierarchy that demonstrates intricate parent-child relationships. When the main process executes FORK operations followed by EXEC calls to different programs, it generates a process tree with multiple levels of inheritance. What makes this behavior particularly unique is how child processes can themselves become parents, spawning grandchildren processes that maintain their own

execution contexts. This creates a cascading effect where process attributes like PPID (Parent Process ID) form chains that reflect the entire execution history. The simulator must carefully manage these relationships, ensuring that when a parent process is waiting in the queue, its children can execute independently while maintaining proper lineage tracking. This hierarchical structure becomes especially complex when the same program appears at different levels of the process tree, each instance carrying its own unique execution state and memory allocation.

The multi-program execution creates complex temporal patterns that challenge the scheduler's effectiveness. Unlike single-program execution where timing is relatively linear, the introduction of multiple programs creates branching timelines where processes execute in parallel, converge, and diverge again. The unique behavior here is how execution time accumulates differently across various process branches—some processes might complete quickly while others spawn lengthy execution chains.

Created Test 2

This simplified test case creates a focused and deliberate memory allocation failure by contrasting two carefully sized programs. programA at 35MB is strategically designed to be unallocatable within the available memory partitions (largest is 40MB but already occupied by init), while too_large at 100MB represents an extreme case that dramatically exceeds system capabilities. The intentional error condition emerges from the sequential execution pattern—the system first attempts to load programA, which should fail due to the memory allocation strategy searching from smallest to largest partitions, none of which can accommodate 35MB when considering the existing memory state. This initial failure tests the simulator's ability to handle "borderline" memory cases where a program technically fits within the largest partition's raw size but cannot be allocated due to the specific search algorithm and existing memory state. The subsequent attempt to load too_large then tests the system's response to a clearly impossible memory request, creating a scenario where the error handling mechanisms must distinguish between different types of allocation failures.