SYSC 4001 Assignment 2 (III)

Faraz Aleboyeh 101311227
Rehma Muzammil 101268686
Repo link part 2: https://github.com/rehmaaaa/SYSC4001_A2_P2
Repo link part 3: https://github.com/farazaleboyeh/SYSC4001_A2_P3

Simulator Description

The interrupts.cpp file implements a simplified operating system simulator that models how system calls like fork() and exec() work using a recursive, trace-based design. It uses helper functions and data structures from interrupts.hpp, including PCBs, memory management, and parsing tools. The helper function append_system_status() records snapshots of the system state, such as process details, memory partitions, and the wait queue after key events. The main engine, simulate_trace(), reads a list of trace instructions and simulates each by updating time, logging events, and performing the required actions. “CPU” lines represent CPU bursts, while “SYSCALL” and “END_IO” simulate interrupt routines that execute an ISR and end with an “IRET.” The FORK section is the most complex: it identifies which trace lines belong to the child process (between IF_CHILD and IF_PARENT or ENDIF), creates a new PCB, allocates memory, and recursively runs the child’s trace before freeing memory and returning to the parent. The EXEC section replaces the current process image by finding the requested program, allocating space for it, and recursively running its trace, then freeing memory and ending the current flow, mimicking how exec() replaces a process in a real OS. The simulation produces two logs: one for the event timeline (execution.txt) and one for system snapshots (system_status.txt). Finally, main() acts as the driver by loading input files, creates and allocates memory for the “init” process, runs the full simulation from time zero, and writes the final outputs. Overall, the file combines recursion, timing, and interrupt simulation to realistically model process creation, execution, and termination as required by the assignment.

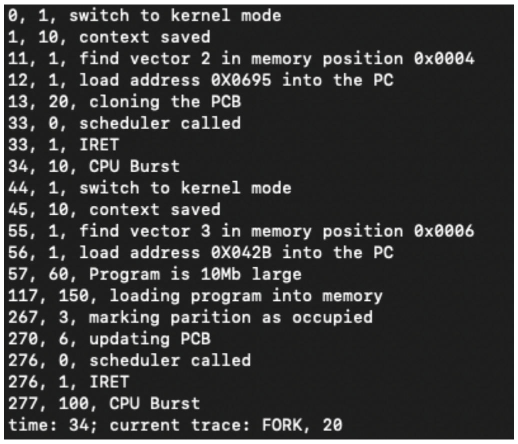
The FORK section of the simulator models how a process creates a child and how both interact in a controlled execution flow. When FORK is encountered, the code triggers a simulated interrupt using vector number 2 and logs the event. It then scans ahead in the trace file to collect all lines between IF_CHILD and IF_PARENT or ENDIF, isolating the child’s instructions while marking where the parent should later resume. A new PCB is created for the child with a unique PID, and the simulator attempts to allocate memory for it, recording whether the allocation succeeds or fails. The child’s trace is executed recursively using simulate_trace(), ensuring the child process runs to completion before returning to the parent, consistent with the assignment’s no-preemption rule. Afterward, the child’s memory is freed, its termination is logged, and a system snapshot is appended. This snippet is relevant because it captures the core logic that makes the simulator behave like an operating system handling fork(), accurately managing process duplication, execution order, and memory allocation within a recursive simulation framework.

The EXEC system call handling section of the simulator models how a process replaces its current program image with a new one. When the simulator encounters an EXEC command, it performs an interrupt routine, locates the requested program in the external files list, and allocates a suitable memory partition for it. It then loads the corresponding program trace file (for example, “program1.txt”) and recursively calls simulate_trace() to execute that new program. Once the child program finishes, the simulator frees the allocated partition and logs the process termination. The key line break; at the end ensures that after executing the EXEC sequence, control does not return to the parent trace mirroring how, in a real system, exec() completely replaces the process’s previous image rather than returning to the old code. This detail is important for the report because it highlights the conceptual accuracy of the simulation: it correctly reproduces the behavior of exec() by halting the old program’s flow once the new executable is loaded and running.

Aside from the fork() and exec() logic, the rest of the file provides the essential framework that makes the simulator operate as a realistic, event-driven system. The helper function append_system_status() is responsible for recording the system’s condition after each major event, including the current time, the running process, memory partition states, and any waiting processes. Within the simulate_trace() function, several other activity types are processed to simulate basic operating system behavior. When a “CPU” instruction appears in the trace, it represents a CPU burst, the simulator logs this event in the execution timeline and advances the simulated clock by the burst duration. For “SYSCALL” and “END_IO” lines, the simulator uses intr_boilerplate() to model how the CPU switches to kernel mode, saves context, executes the interrupt service routine, waits for the delay associated with the device, and then returns to user mode with an “IRET” instruction. Each of these events is logged in the execution string and followed by a call to append_system_status() to capture a new snapshot of the system. The main() function manages the entire simulation flow. It begins by loading all input data, such as the vector table, device delays, and external program files, using parse_args(). It then initializes the first process, “init,” assigns it memory using allocate_memory(), and reads the main trace.txt file into a vector of lines. The simulation starts at time zero when simulate_trace() is called, and when it finishes, the results are saved to execution.txt and system_status.txt. Together, these components handle timekeeping, interrupt simulation, memory management, and file I/O, forming the core structure that supports the more advanced fork() and exec() operations within the simulator.

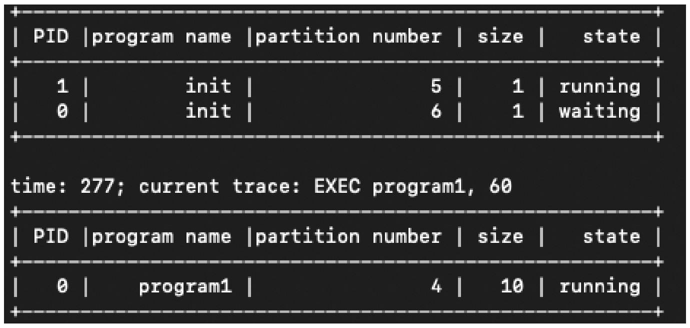
Simulation Test Analysis

The input files provide all data needed for the simulator to run. The trace file defines the main process flow, while sample program files contain traces for programs executed through EXEC. The vector table maps interrupt numbers to ISR addresses, the device table sets I/O delays, and the external files list gives program names and sizes for memory allocation. The simulator produces two outputs: execution.txt, which logs all events and timings, and system_status.txt, which records system snapshots showing process states and memory use after each major event.



```
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
33, 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 10Mb large
117, 150, loading program into memory
267, 3, marking parition as occupied
270, 6, updating PCB
276, 0, scheduler called
276, 1, IRET
277, 100, CPU Burst
time: 34; current trace: FORK, 20
```

Figure 1a: A screenshot of the execution.txt file after running the first (assignment-given) test case.



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

Figure 1b: A screenshot of the system_status.txt file after running the first (assignment-given) test case.

The execution.txt (Figure 1a) and system_status.txt (FIgure 1b) screenshots show the simulator handling both FORK and EXEC operations. In execution.txt, the trace begins at time 0 with the init process switching to kernel mode, saving its context, and performing a FORK. The PCB is cloned at time 13, creating a child process (PID 1), and after returning from the interrupt (IRET at time 33), the child begins executing. A CPU burst follows, confirming that the child is now running while the parent waits. At time 44, the child triggers an EXEC system call to replace its current program image. The simulator correctly identifies vector 3, loads the ISR address, determines the program size (10 MB), and simulates loading the program into memory over 150 ms, then marks the partition as occupied and updates the PCB. The scheduler is called, and execution resumes with a CPU burst from time 277 onward, indicating that the new program is running successfully. Correspondingly, the system_status.txt output confirms this behavior: before the EXEC, PID 1 (init) is running in partition 5 while the parent (init, PID 0) waits in partition 6. After the EXEC, the running process (PID 0) now represents program1 occupying partition 4, showing that the old init image has been replaced. The snapshots and timing in both logs verify that the simulator correctly transitions between states, handles memory allocation and replacement, and maintains proper parent/child synchronization.