SYSC 4001 Assignment 2

Atik Mahmud (101318070) & Modibedi Mosigi (101256959)

Part III

Trace File 1

At the start, only the init process is running. When it reaches the FORK 10 instruction, the CPU switches to kernel mode to handle the request. The system saves the current process state, looks up the correct service routine, and creates a new child process by copying the parent's data. After that, the parent waits, and the child starts running right away. When the fork is done, the CPU returns to user mode.

The child then runs EXEC program1 50. This again makes the CPU switch to kernel mode. The system saves the process state, finds the right service routine, and loads program1 into memory. Since the program is 10 MB, it takes 150 milliseconds to load (15 ms per MB). After loading, the system updates the process information and goes back to user mode. The child then uses the CPU for 100 milliseconds, as defined by the program.

Later, the parent (the original init process) runs EXEC program2 25 after the child finishes. This again loads a new program and switches modes. The process then calls SYSCALL 4, which runs for 250 milliseconds to handle a device task before returning to normal mode. Once a process uses EXEC, its old program is replaced, so anything that came after in the old trace is ignored.

The system snapshots confirm this. At time 24, the child is running while the parent waits. At time 247, the child has finished running program1. At time 620, the parent has replaced its old image with program2, and only program2 is active.

Trace File 2

The second simulation starts the same way. The init process performs FORK 17, creating a child. The parent waits while the child runs. The child soon does EXEC program1 16, which replaces its current program with program1. The system handles the same steps as before switching to kernel mode, saving state, and loading the new program.

Inside program1, another FORK 15 happens. The system duplicates the process again, creating a third one. Because both the parent and child parts of this fork have no special instructions, they both continue to the same next command: EXEC program2 33. This means both of them replace program1 with program2 and run it separately. That is why the log shows program2 running twice, once at time 530 and again at time 864.

After that, the original init process resumes its own work and finishes with a 205-millisecond CPU burst. The logs clearly show how processes are added and replaced: one fork makes two processes, a second fork makes three, and each EXEC changes what program each process runs.

Trace File 3

The third test begins with init doing FORK 20, which again creates a new process. The child runs first, while the parent waits. The child then performs EXEC program1 60, which loads program1 into memory. This program runs a 50-millisecond CPU burst, a SYSCALL 6, a 15-millisecond CPU burst, and finally END_IO 6.

When the SYSCALL 6 happens, the CPU moves to kernel mode and runs the device service routine. This causes a long delay while the input or output device finishes its task. Afterward, control returns to user mode, and the process continues. The END_IO 6 step again switches to kernel mode to finish the I/O operation. The large time jumps in the log show how long these device operations take before the process can continue.

Trace File 4

The system starts with a 60-time-unit CPU burst. Then it switches to kernel mode and performs a fork, creating a new process. The child loads program1, which is 5 MB in size. The system marks its memory area as used and updates its data. The process then runs a 70-unit CPU burst, handles system calls, and finishes I/O actions through the service routines. After an ENDIO, it runs another 40-unit CPU burst and then performs an EXEC program2, loading a new program. This program has more bursts and another fork that makes another child process. That child runs a 25-unit CPU burst and two system calls.

At time 86, there are two processes (the parent and the child). At time 214, program6 is running while the parent waits. At time 945, program7 becomes active, replacing older programs. At time 1057, another fork happens, creating a new program7 process, one running and one waiting.

Trace File 5

This one starts already in kernel mode with a fork. The new process loads program8 (10 MB) into memory and starts running. It performs a long 80-unit CPU burst, then handles an ENDIO interrupt. After that, it does a 30-unit CPU burst and loads program9 (15 MB). The system updates everything and starts the program. The next step is a SYSCALL that lasts 300 time units. When that finishes, the process runs a 55-unit CPU burst, forks again, and runs several smaller bursts of 15, 20, and 20 units.

Trace File 6

The system starts with a 40-unit CPU burst, then switches to kernel mode to do a fork. The child loads program10 (5 MB) into memory. Soon after, another fork happens, creating another child process that runs a 25-unit CPU burst. Then there are two system calls, each taking 211 time units. After that, the process returns to user mode and runs a 70-unit burst before loading program11 (5 MB). It then runs a 120-unit burst, an ENDIO that lasts 150 time units, and a final 30-unit burst.

At time 65, the fork creates two init processes, one running and one waiting. At time 181, program10 runs while the parent waits. At time 205, another fork creates a new program10 process that runs while the others wait. By time 868, only program11 is active.

How the Simulation Handles System Calls and Interrupts

All the tests follow the same pattern. Whenever a process calls FORK, EXEC, SYSCALL, or END_IO, the CPU moves from user mode to kernel mode. The system saves the current process state, finds the right service routine, and runs it.

- FORK makes a new process.
- EXEC loads a new program and assigns it memory.
- SYSCALL and END_IO deal with device operations.

After the service routine is done, the system may call the scheduler to decide which process runs next, then goes back to user mode.

The simulation also tracks memory and scheduling. Each EXEC reserves a new space in memory and updates which process uses it. The scheduler usually lets new child processes run first after a fork. The status tables show these changes in memory and process states over time.

Usage of break at line 204 in interrupts.cpp

After a successful EXEC, the current process no longer exists in its original form. In other words, exec() replaces the current process image with the new program and it never returns to the original code.

If we didn't use break, the function would continue iterating through the remaining lines of the old trace file, executing instructions that belong to the pre-exec context, which should be invalid now.