Part 2: Design and Implementation of an Interrupts Simulator

SYSC 4001: Operating Systems

James Bian, Amreen Shahid
101311339, 101306199
Carleton University
Oct 6 2025

The goal of this assignment is to simulate how system calls and interrupts are handled within an operating system. The system reads a trace file that holds the activity and the duration or associated device number. From that input, an execution file is created, outlining the process of how the interrupt is handled using information from the vector table and device table. Each execution line holds the current time, the duration of the event, and the event type, which can include switching modes, looking up the vector table address, executing the ISR and returning back to the main program. The program is then tested and adjusted to see how different duration times affect the output.

The results of the changes in simulation reflected the effects seen in actual operating systems with events such as delays and system calls. Increases save/context time led to increases in run time, especially with more instances of accessing memory, or loading it.

Table 1: Resulting time of each trace file showing the effect of a varying save/restore context time

10ms (execution_11.txt)	1470, 72, CPU Burst
20ms (execution_12.txt)	1490, 72, CPU Burst
30ms (execution_13.txt)	1510, 72, CPU Burst

System calls require access to the vector table to find the ISR required; the fewer system calls the more efficient the simulation, as the program does not have to keep track of where it left off as frequently. Increasing the delay time from 10ms led to a 10 ms delay per system instruction; each system call had to save context upon giving control to Kernel mode. Meanwhile, a long ISR activity time can lead to problems if not checked.

The ISR execution taking too long can lead to the CPU getting stuck in the ISR if there is no protocol to handle multiple interrupts, if there are multiple from different I/O devices. While the ISR is being done, the computer is in Kernel mode, which means it is not running the actual program. All other interrupts that may occur soon after will also be backloaded, which leads to slowness in a real machine. This is shown by the timer and sharp increase in delay times during the simulation any time after the CPU has loaded, and before the routine returns with the IRET line.

Table 2: Resulting time of each trace file showing the effect of a varying ISR activity time

40ms (execution_18.txt)	2048, 10, CPU Burst
120ms (execution_19.txt)	2368, 10, CPU Burst
200ms (execution_20.txt)	2688, 10, CPU Burst

In the total execution time, fast context switches with short ISR's give the CPU its desired conditions, as there is little delay for the computer to deal with, meaning little to no overhead. Slow context switches, as shown in the simulation, creates delays in the exiting and entering the interrupt. Long ISR execution creates problems as overhead significantly increases, and more time ends up being used handling calls than running the actual program. The user's program would also handle interrupts much slower, even if the context switches are efficient.

In the main program when executing an interrupt or system call, the ISR runs with this line:

```
execution += std::to_string(current_time) + ", " + std::to_string(delays[duration_intr - 1]) + ", SYSCALL: run the ISR (device driver)\n";
```

While testing, it was observed that not subtracting 1 from the duration_intr, will produce an incorrect output, since it will be accessing the device number one more than intended. This is because the simulation uses zero based indexing. As this is just a simulation, no harm is done, but in real systems this small tweak in code can cause big problems.

In real systems it is not uncommon to handle multiple interrupts one after another. In some of the test cases, there are multiple interrupts and system calls one after another and the simulation handles them sequentially. This makes sure that each interrupt is fully handled before moving on to the next one even if the multiple interrupts are from the same device number.

In the current vector table, each vector is 2 bytes long. Increasing this size will change the amount of memory allocated for each vector and change the memory positions. For example, for 2 byte long vectors, vector 0 will have a position of 0x0000 and vector 1 will have a position of 0x0002. If, for instance, each vector were 4 bytes long instead, vector 0 will have a position of 0x0000 and vector 1 will have a position 0x0004. This won't exactly affect the timing of the execution however in the execution file, events like, “find vector 4 in memory position 0x0008” will change if the byte size is increased since the memory position will shift.

Overall, this simulation is a solid way of understanding how interrupts and system calls are handled and what little changes can do to affect the program. By changing duration times for the context and ISR, we observed that longer durations can lead to longer execution times and increased overhead, so it is ideal if the system has shorter duration times to run quickly and efficiently. Even small changes like indexing and byte sizes can greatly affect the way the system runs and should not be overlooked when creating these systems.