SYSC4001 Assignment 1: Part 2 **2025-10-05**

Date of submission (Group): 2025-10-05

Colin Calhoun (Student 1) - 101307947
James Noel (Student 2) - 101306496

Part II – Report: Design and Implementation of an Interrupts Simulator

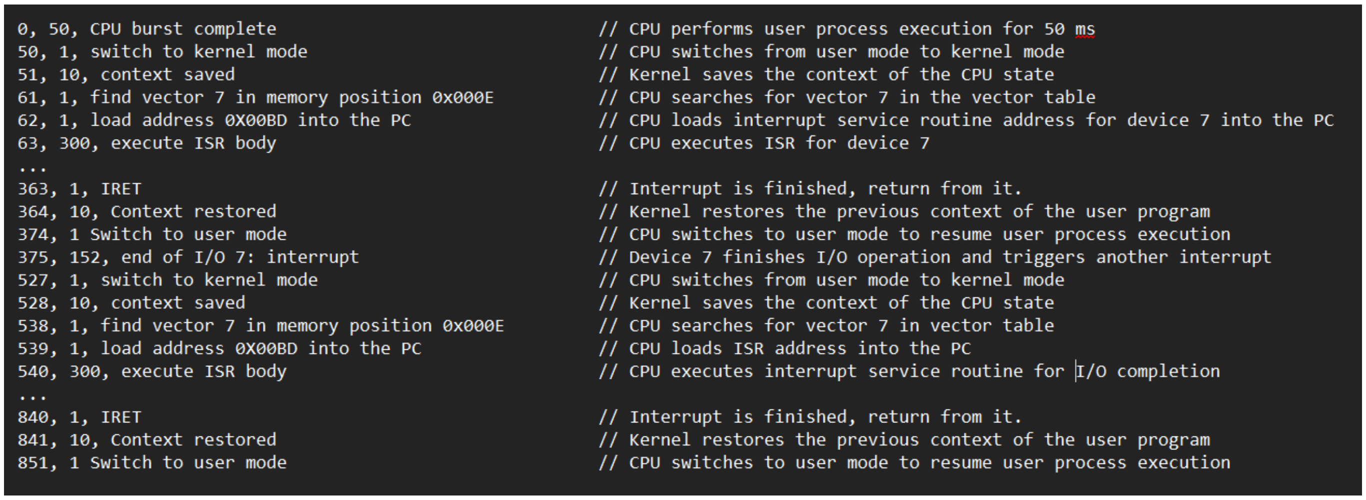
Link to project repository:
https://github.com/ColinCalhoun/SYSC4001_A1.git

The following trace file provides the interrupt simulation with instructions to execute sequentially. The simulation essentially imitates the mechanics of system calls from user programs that request services from the operating system, performing context switches in between user and kernel mode, saving user process contexts and ISR instruction execution.

CPU, 50
SYSCALL, 7
END_IO, 7

This trace segment first delegates 50 ms of the CPUs processing time to the user process by instantiating a "CPU" instruction. It then performs a system call using the "SYSCALL" instruction, which interrupts the CPU and shifts its attention to a specified ISR ad address 7. Finally, an end-interrupt instruction is called using the "END_IO" instruction which alerts the user-mode CPU and shifts its attention once more back to kernel mode once a device is finished executing.

The above trace file results in the following execution:



```
0, 50, CPU burst complete                          // CPU performs user process execution for 50 ms
50, 1, switch to kernel mode                       // CPU switches from user mode to kernel mode
51, 10, context saved                              // Kernel saves the context of the CPU state
61, 1, find vector 7 in memory position 0x000E     // CPU searches for vector 7 in the vector table
62, 1, load address 0X00BD into the PC             // CPU loads interrupt service routine address for device 7 into the PC
63, 300, execute ISR body                          // CPU executes ISR for device 7
...
363, 1, IRET                                       // Interrupt is finished, return from it.
364, 10, Context restored                          // Kernel restores the previous context of the user program
374, 1 Switch to user mode                         // CPU switches to user mode to resume user process execution
375, 152, end of I/O 7: interrupt                  // Device 7 finishes I/O operation and triggers another interrupt
527, 1, switch to kernel mode                      // CPU switches from user mode to kernel mode
528, 10, context saved                             // Kernel saves the context of the CPU state
538, 1, find vector 7 in memory position 0x000E    // CPU searches for vector 7 in vector table
539, 1, load address 0X00BD into the PC            // CPU loads ISR address into the PC
540, 300, execute ISR body                         // CPU executes interrupt service routine for I/O completion
...
840, 1, IRET                                       // Interrupt is finished, return from it.
841, 10, Context restored                          // Kernel restores the previous context of the user program
851, 1 Switch to user mode                         // CPU switches to user mode to resume user process execution
```

The CPU performs a burst by executing a user process for 50 ms. An interrupt from device 7 alerts the CPU and forces it to switch from user mode to kernel mode. The kernel saves the CPU's current state so it can be restored after the interrupt is serviced. The CPU then locates vector 7 in the interrupt vector table, loads the address of device 7's ISR, and executes it before returning to user mode to continue the user process and await I/O completion. Once device 7 has finished its operation, another interrupt is signaled to the CPU, which switches to kernel mode again. The kernel saves the CPU's state, the CPU locates vector 7, loads the ISR address into the program counter, and executes it to handle the I/O completion. Finally, when the interrupt service routine is finished, the kernel restores the CPU's user state, and the CPU switches back to user mode to resume executing the user process

Change the value of the save/restore context time from 10, to 20, to 30ms:

Context save/restore time (ms)	**Total execution time (ms)**
10	331
20	371
30	411

Vary the ISR activity time from between 40 and 200 (context save/restore time = 10ms):

ISR time (ms)	**Total execution time (ms)**
40	331
50	351
60	371
70	391
80	411
90	431
100	451
110	471
120	491
130	511
140	531

150	551
160	571
170	591
180	611
190	631
200	651

Change the value of the save/restore context time from 10, to 20, to 30ms. What do you observe?

For the trace segment above, when the save/restore context time is changed by increments of 10ms the total execution time of all three trace instructions increases by 40ms. This is because that program requires the context to be saved twice and restored twice, so by adding 10ms for each of those instructions takes 4 x 10ms extra. The 10ms trace program took a total of 331ms, the 20ms trace program took a total of 371ms, and the 30ms trace program took a total of 411ms.

Therefore, for a trace program with n save/restore context time executions will take n x 10ms longer each time you increase their execution time, assuming that you use 10ms intervals.

Vary the ISR activity time from between 40 and 200, what happens when the ISR execution takes too long?

Varying the ISR activity time between 40 ms and 200 ms will significantly increase the total process execution time, especially as it approaches 200 ms. This is because an ISR is executed each time the instructions SYSCALL or END_IO are issued, which can drastically consume the CPU's processing time, leaving less room for regular CPU bursts—which means potentially crucial CPU operations to be neglected within time constraints. If the ISR execution takes too long, the CPU will have to dedicate much more of its time to handling ISRs, causing the process execution time to extend beyond expectations and resulting in multiple device ISRs backing up, waiting to be serviced.

For the trace segment above, the total processing time increased by the extra execution time (i.e. isr_time from 40ms to 50ms) multiplied by the number of ISR instruction executions—which was two. Which means: 10 ms x 2 = 20 ms longer than before. Therefore, a trace program with *n* ISR instructions will take ($n \times \Delta t$) ms longer each time you increase their execution time, where Δt is the increase in ISR time from its predecessor.

How does the difference in speed of these steps affect the overall execution time of the process?

The difference in the speed of these steps affects the total execution time of the process by forcing all instructions to compete for CPU time. If ISR execution times increase, the CPU spends most of its time in kernel mode handling interrupts instead of performing operations or computations at the user level. Slower I/O processing times for devices can cause multiple ISRs to back up in a waiting queue, further restricting the CPU from executing user-level tasks and leaving the user process to wait idly.

What happens if we have addresses of 4 bytes instead of 2?

If both the vector table and device table have 4 bytes instead of 2 bytes, the memory's capacity would expand for more data, however the execution time of the process would be much longer due to longer address processing. The CPU would have to both access and save memory addresses containing twice the amount of data.

What if we have a faster CPU?

Although the CPU can execute instructions, save and restore process contexts, and switch between kernel and user mode more quickly, I/O operations are asynchronous and can occur at any moment, increasing the unpredictability of latency and process sequencing. For instance, the CPU may finish executing an ISR and return to user mode before the device has completed its operation, which could require an additional context switch when the I/O completes. If the CPU were slower, it might still be in kernel mode when the I/O finishes, avoiding that extra context switch. The simulator runs synchronously however, which means an interrupt termination instruction must be issued to alert the CPU while in user mode. So in this case the CPU will always switch to user mode after executing an ISR and will be interrupted by END_IO via the trace file; thus, the sequence of execution in the simulation is deterministic rather than subject to real-time variability.