# SYSC4001 Assignment 1:

# Interrupt Simulator: Part I & II

**Antoine Hickey,**

**ID: 101295764**

**Enzo Chen,**

**ID: 101306299**

**Lecture: SYSC4001A**

**Lab: SYSC4001L2**

## Part I - Concepts

a)

Below is a sequence of hardware and software steps that are involved in the full interrupt mechanism for I/O device servicing.

- The interrupt is enabled by a set of sensor(s) and/or software within the I/O device (this could be for when it's finished printing, reading, or doing some other task depending on the I/O device).

- This signal is then stored within the I/O device's internal IR (interrupt register) hardware.

- It then travels through a physical connection (copper wire, cable, port, etc.)

- To the IOP IR (Input/Output Processor Interrupt Register) hardware. This can be compared to a 1401 processor from the early stages of computers as it is a less powerful computer that is used as a middleman for communicating between I/O devices and the main CPU.

- The interrupt signal is then sent through the internal Interrupt Data bus, where it wakes up the CPU (if it was previously asleep). The signal is then stored within the CPUs IR (Interrupt Register) hardware.

- Because of the way the CPU's hardware architecture is set up, this register (IR) is checked at the end of each fetch execute cycle. Therefore, after finishing its execution, or waking up, it will recognize that an interrupt signal has been sent.

- Then, the CPU will use the data that is within the IR to find which interrupt handler to use by looking through its vector table hardware, which then points to a set of drivers stored in the OS/Monitor. This will branch to that specific device's interrupt service routine (ISR) software, before continuing with the instruction it was doing previously.

b) A system call is a request sent by a user program to ask the OS to perform tasks which are normally reserved for the OS. They are accessed via an Application Programming Interface (API).

There are six main categories of system calls:

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

Three examples of system calls are:

1. Reading from a file (File management)
2. Get current time/date (Information maintenance)
3. Allocate memory (Process control)

System calls are related to interrupts because they can create a software interrupt, known as a "trap" or "exception". The system call provides an interrupt vector, which the CPU then uses to find the system call service routine and execute it. The system call also sets the CPU state to kernel mode, by setting the mode bit accordingly. The kernel identifies which system call was invoked, then executes it, and finally gives control back to the program.This process of using a software interrupt, providing an interrupt vector, checking the vector table, and executing a service routine uses the same hardware as interrupts to execute a system call. This is different from hardware interrupts, which are triggered by hardware such as input and output devices, instead of from user software.

c)

i. To check if the printer is OK, the CPU should:

- Check if the printer is powered on and connected

- Check if busy (already printing something)

- Check if the printer is jammed

- Check if the printer has paper

- Check if the printer has ink

ii.

LF (Line Feed):

- Spin a motor to move the print head down by one line (if page) OR

- Spin a motor to move the paper up by one line (if roll)

- Check if reached bottom of paper, feed new paper (if page), or alert operator to provide more paper if none left

CR (Carriage Return):

- Spin a motor to move the print head back to the left side of the page

- Read new line (character array) from buffer

d. In batch OSes, off-line operation works by offloading the human readable input and output devices to separate processors. This makes these devices off-line, since they were not managed by the main CPU. Typically, off-line operation was implemented in the following way:

1. A weak processor, such as an IBM 1401, uses a card reader to read punch cards and load programs onto a magnetic tape.
2. An operator moves the tape from the 1401 to the strong CPU, such as an IBM 7094.
3. The 7094 CPU runs the program and outputs the data on another tape.
4. The operator moves the tape from the 7094 to another weak processor, such as a 1401.
5. This 1401 processor reads the tape and outputs the data to a printer for humans to read.

Off-line operation has some advantages and disadvantages compared to the CPU operating the card reader and printer directly. An advantage of off-line operation is that the CPU no longer has to wait for the card reader and printer, and can instead use tape drives, which are much faster. This frees up the CPU to finish a batch faster and spend more time processing, rather than waiting for I/O. This advantage can be leveraged even further by using several input and output tape processor units, so that input tapes are always ready for the CPU to process, and output tapes are printed onto paper as soon as possible. Using extra processors, readers, and printers allows the CPU to process even more, and minimizes idle and I/O processing time. The disadvantage of off-line operation lies in the use of magnetic tapes to store data. Since tapes can only be written to and read from sequentially, programs could not be added to the tape while the CPU was processing. Because of this, operators did not run individual programs, so programmers had to wait for an entire batch of programs to be processed before being able to view the result of their code.

e)

i. If a programmer wrote a driver and forgot to parse the "$" symbols, the driver might misinterpret data that is identical to keywords such as "RUN" and "END" that were supposed to be plain text or part of a program. The driver would then execute the commands, causing unexpected and incorrect behavior. We prevent this error by preventing user programs from directly interfacing with devices, instead requiring them to use system calls to request the OS to interface with devices instead.

ii. If a card had "$END" in the middle of the program, the OS should stop the execution of the program immediately and regain control over the CPU. The CPU would then fetch the next instruction from the OS, as it begins to process the next job.

f) Below are two operations that are considered privileged

- Switch to kernel mode

This privileged instruction switches the processor from user mode to kernel mode, which allows them to modify the Operating System/Kernel/Monitor. This includes the job sequencer, drivers, JCL (Job Control Language), and Interrupt Processing. This is privileged because unrestricted access would allow user programs to corrupt the OS, potentially bypassing security measures, and giving unrestricted access to all other user programs.

- Interrupt management

The interrupt manager is used to manage when interrupts get called, as well as how they are handled depending on the device. Being able to modify the interrupt management system is privileged because malicious user programs could disable critical interrupts (like timer interrupts). If you can modify the interrupt management system (by changing the vector table), you can also redirect interrupts to malicious handlers Interrupt Servicing Routine (ISRs).

- I/O control instructions

I/O control instructions allow the CPU to interact directly with I/O devices, controlling them and exchanging data with them. All I/O control instructions are privileged because the operating system normally must verify that the instructions are valid. If this can be modified, then invalid commands can be sent to the I/O devices, having unwanted, or unpredictable results. Defining the I/O instructions as privileged prevents the user from performing these illegal I/O operations and forces the user to execute system calls for the operating system to perform the I/O which is safer.

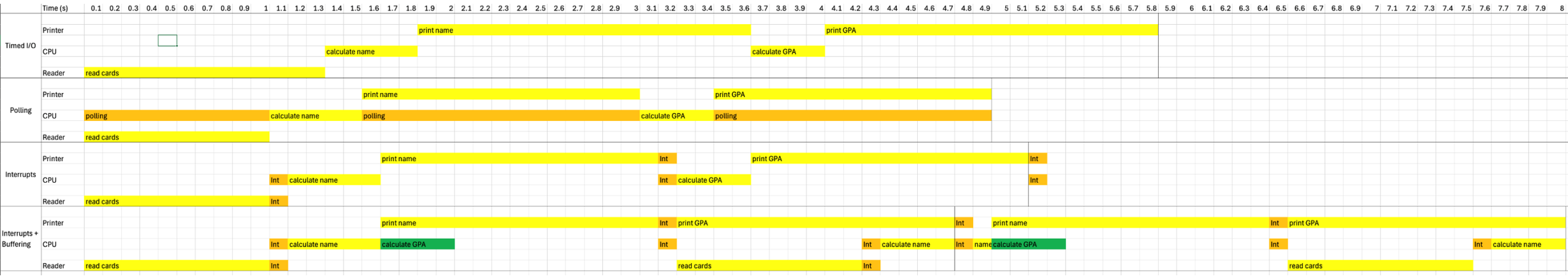
- Timer management

The timer is used to ensure a user program never takes control of the CPU and gets stuck in a failed system call or infinite loop, preventing the OS from regaining control. The timer management instructions are used to control the timer and make sure it interrupts user instructions after a count of clock cycles, to let the OS regain control of the CPU. Timer management instructions are privileged because giving the user direct control over the timer could cause a program to prevent the timer from interrupting it, preventing the OS from regaining control of the CPU through the timer and prevents the CPU from running other programs (multi-processing).

g) First, the Control Language Interpreter reads the $LOAD TAPE1: instruction, which makes it call the loader. The loader, which runs in kernel mode, uses the Device Drivers to request the magnetic tape reader's IOP to read the first file on the tape. The IOP reads data from the tape and stores it on a buffer. As the IOP's buffer fills, it sends interrupts. The CPU then runs the corresponding ISR, by using the provided vector to look up and go to the start of the ISR. The ISR instructs the CPU to move data from the IOP's buffer into main memory.

The Control Language Interpreter then reads the $RUN instruction, which instructs the resident monitor to set the PC to the start of the program, sequencing the job. The CPU then starts executing the program.

h)

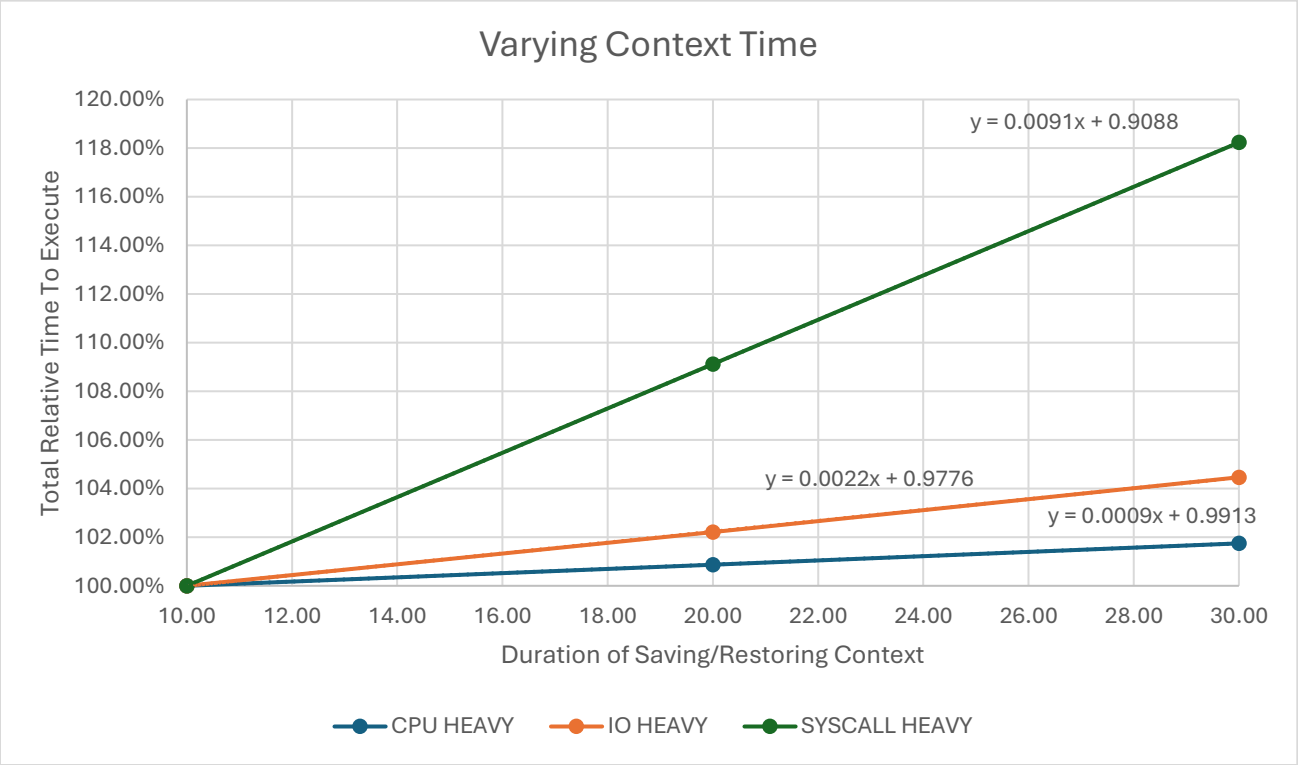



The slowest of the four cases for both one cycle and the entire execution was by far Timed I/O, which took 5.8s per cycle and 1647s total. This is because of the extra margin of error added to the I/O times and because none of the processes could be run in parallel. Interrupts were the next slowest in both categories, taking 5.1s per cycle and 1477s total. It removed the margin of error required for Timed I/O but added interrupt latencies. Polling was slightly faster than Interrupts, since it no longer had interrupt latency, taking 4.9s per cycle and 1392s total. The major downside to polling, however, is that the CPU is always active while waiting for the I/O, causing transistor degradation and wasting energy. Interrupts with buffering was the fastest in both categories, taking 4.7s per cycle and only 939s total. It was slightly faster for one cycle, since calculating GPAs and printing names could be done simultaneously. It, however, was far faster to complete the entire execution, as parallelization removed almost all the idle time from the printer, meaning the total execution time was limited almost exclusively by the printer.

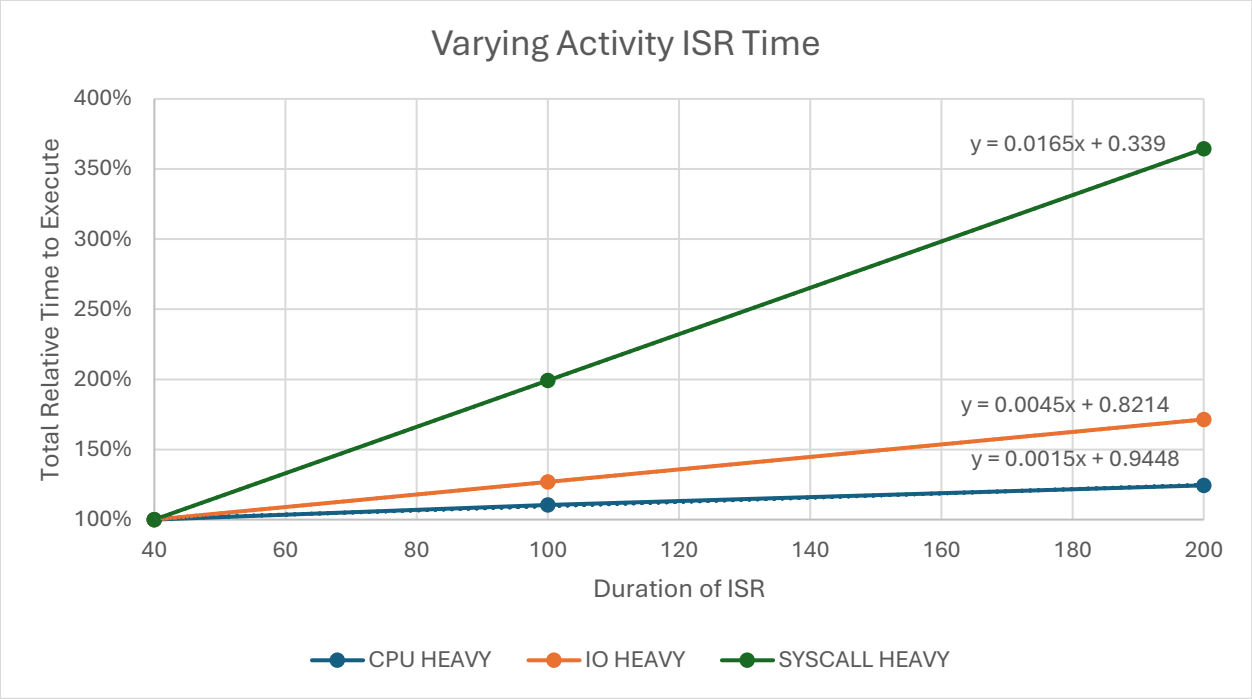
## Part II - Design and Implementation of an Interrupts Simulator

https://github.com/hicked/SYSC4001_A1

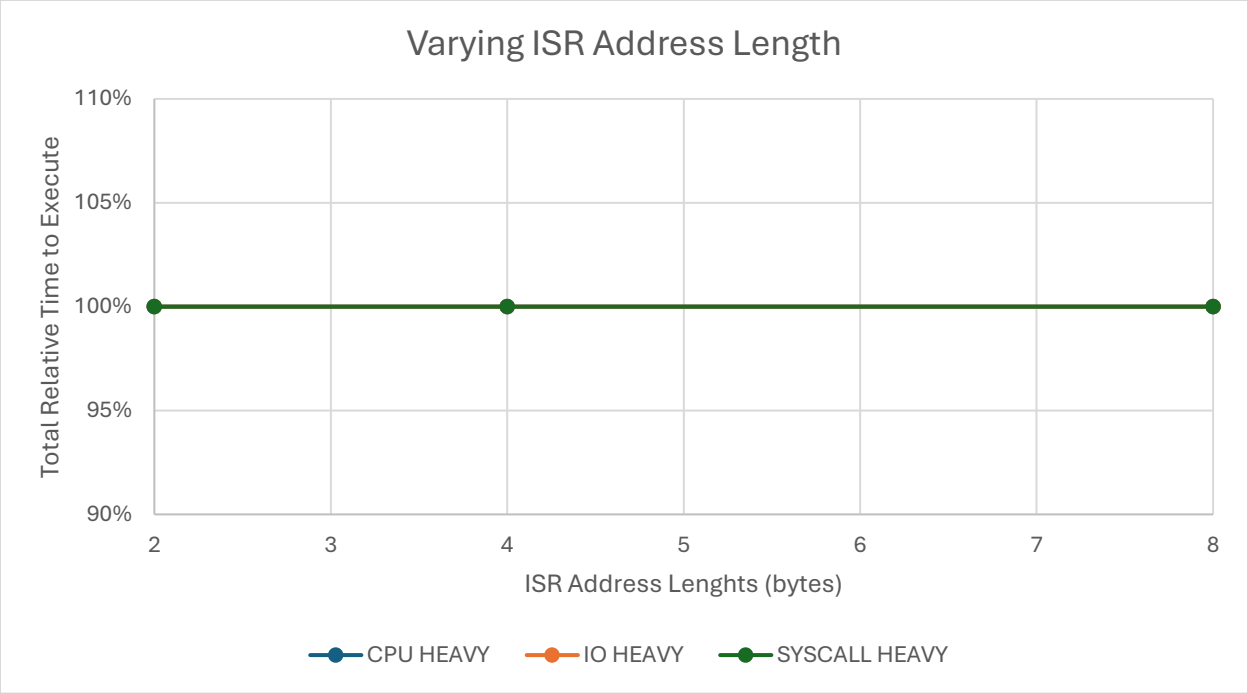
To understand how this simulated environment will work with different parameters (such as varying context time, ISR time, and ISR address size), we decided to create three different trace files: trace_cpu_heavy.txt, trace_io_heavy.txt, and trace_syscall_heavy.txt. This allows us to understand how the system will act not only under the different parameters, but also different load scenarios. Additionally, we used base cases to enable us to find a relative percentage increase of execution times instead of raw or absolute execution time. This is since if some trace files inherently take longer than others, then they will always take the longest.




Our analysis of varying context save/restore times shows that system call-heavy workloads experienced the most significant performance drop. While both system call-heavy and I/O-heavy traces undergo the same number of context switches, the relative impact is very different. This is because software interrupts complete quickly, making context switching a larger fraction of total time the trace file takes to execute. In contrast, I/O-heavy workloads include long device wait times that make the context switching a relatively small component of total execution time.




Analyzing varying ISR execution times reveals that system call-heavy workloads experienced the most dramatic performance impact, with execution time increasing at a rate higher rate (looking at the sloped). This is due to system calls using the vector table, and therefore ISR to complete. I/O-heavy workloads also showed increased completion time since hardware interrupts also require ISRs, but the relative impact is reduced by the device wait times that dominate I/O calls. CPU-heavy workloads remained essentially flat, showing that ISR time variations will not impact very CPU intensive tasks due to them being infrequent. This also increased the total execution time the most since, while context save/restores are more frequent (twice per ISR call – one save, one restore), the ISR time were increased far more (as shown by the x-axis). This means that since context switching gets called twice per ISR, it may be more important to optimise it over the ISRs themselves which only run once (depending on how long they are of course).




Looking at varying ISR address lengths, this graph shows that all workload types remain completely flat at 100% relative execution time (i.e. relative to the initial base cases), indicating that address size has no impact on interrupt processing performance. However, the ISR address length does have some impart on system capacity: ISR address lengths fundamentally determines how many ISRs can be supported. With 2-byte addresses, the system is limited to a smaller number of interrupts in the vector table, while 4-byte or 8-byte addresses enable the system to have more ISRs. Note that this will mean that the space reserved for the OS gets increased, and thus there will be less space for other data (programs, files, etc.).

Finally, reducing CPU time will make all operations run by the CPU faster (this could be implemented in the form of a clock speed). Currently, there is a Macro in the interrupt header file for CPU speed, but it is not being used and thus would not do anything if it were to be changed. If we wanted to change the CPU speed in this lab iteration, we would have to manually go in every trace file and lower the CPU times. This would essentially reduce all total execution times. The more CPU activities that are present, the bigger the difference. Thus, CPU bound processes and trace files would be reduced a great deal.