SYSC 4001 Assignment 1 – Interrupt Simulator

SYSC 4001-A

Student 1: Maxim Creanga 101298069

Student 2: Braedy Gold-Conlin

GitHub: https://github.com/Volley24/Assignment-1

Part 1 - Concepts

Student 1 —> Maxim

Student 2 —> Braedy

a) Student 1

a) [0.1 mark] Explain, in detail, the complete Interrupt mechanism, starting from an external signal until completion. Differentiate clearly what part of the process is carried out by hardware components, and what is done by software. **[Student 1]**

An interrupt works by alerting the CPU whenever something external has completed doing something.

Essentially, what happens is I/O devices or even processor related things can raise an interrupt. The exact process is as follows:

1. An external signal is applied to the processor over an interrupt request line. IRQx
2. The processor pauses whatever it was doing, and does R15→ R14, and pushes flag register(xPSR) and interrupt flag register on the stack. It then does LDR operation where it goes into the vector table, and depending on what IRQ number raised the interrupt, it will then move the value at that vector table → R15. (the vector table is located at the top of memory).
3. Then, on the next fetch-execute cycle, since R15 is the memory address of the ISR, it will load in the first instruction for the ISR.
4. In the ISR, we typically set the mask register in the processor I = 0 so that while we are processing this interrupt, no other interrupts can occur.
5. We perform the ISR, and POP the registers back, so R15 becomes what it was before entering the ISR and all the flag registers are saved, and I =1 again (so we can accept interrupts again).

b) (Student 2)

b) [0.1 marks] Explain, in detail, what a System Call is, give at least three examples of known system calls. Additionally, explain how system Calls are related to Interrupts and explain how the Interrupt hardware mechanism is used to implement System Calls. **[Student 2]**

To understand system calls, we first need to understand the difference between user and Kernel mode:

User mode: Where normal programs run. they **cannot directly access hardware or critical OS data**.

Kernel mode: This is where the OS runs; it can access hardware and manage system resources.

A system call is essentially a request from a user program to the operating system to perform an action that the program cannot do itself. For example, if you wanted to do printf("____") in C, the user program cannot access kernel level operations like writing to the terminal. So, the C standard library would switch to kernel mode, and call on the required function to do that.

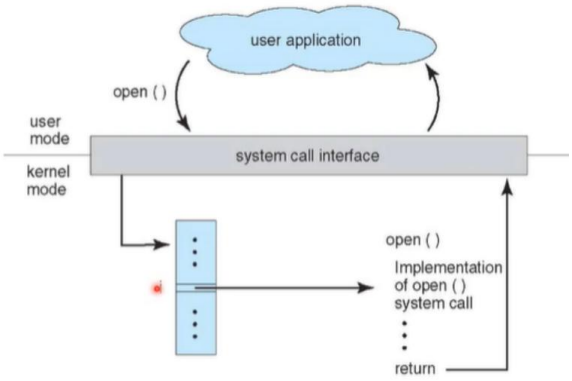
So, a **system call** is the interface between a user application and the operating system. When a user application needs certain services like reading a file, allocating memory, or communicating with hardware it requests them through system calls.

Here is an in depth description of how it works from my research:

The user program will set up certain registers to tell the kernel which system call to execute and what arguments to pass. x86 operating system uses four of its general purpose registers (out of the total 15), to do this. In detail, Eax (holds the system call number), (ebx,ecx,edx) which hold the arguments for the system call. Once you are done configuring these registers, you would do "int 0x80", which makes the system call which triggers a software interrupt. The CPU then switches from user mode to kernel mode, and the kernel

receives control and looks up which system call you requested (from eax register). The kernel then executes the service routine/ system call and once it is done, returns the result back to the user program and switches back to user mode.

Here is an abstract diagram from the textbook:



c) [0.1 marks] In class, we showed a simple pseudocode of an output driver for a printer. This driver included two generic statements:
i. check if the printer is OK
ii. print (LF, CR)
Discuss in detail all the steps that the printer must carry out for each of these two items (**Student 1** should submit answer ii., and **Student 2** should submit answer i.).

```
Input parameter: text[80];
// holds the data to print
i=0;

 repeat 80 times {
        check_if_printer_OK();
        print(text[i++]);
        advance_motor();
 }
 i = 0;
 print(LF, CR);
 return();
```

i) Student 2:(Braedy): This driver is loaded into RAM by external storage (could be punch card, tape, ROM ect...). It should check that the printer has power, has paper, ink or jammed before executing the instructions below it. If the printer is not okay, it should break out of the loop and return.

ii) Student 1(Maxim): In order for the driver to execute print(LF, CR), the printer must do the following steps:

1) Read the characters present in the buffer. In this case, this would be the ‘LF’ and the ‘CR’ character, which are of course represented in binary, in either ASCII, UTF-8 or whichever character set the printer is using.

2) Interpret the characters as a special command: LF, or line feed, which is to step down the paper one vertical line height, , and CR, or carriage return, which is to move the printer head to the left.

3) After the instructions are parsed by the printer’s controller, the actual mechanical actions are executed. For the Line Feed, the controller sends instructions to the printer’s motor, controlling the flow of the paper, and moves it down by one line height, which is a constant. For the Carriage Return, the printer must send instructions to the motor controlling the printer head, and move it to the left however many times needed to reach the leftmost margin.

4) During each CR, LF, update the position of the ‘cursor’ internally, so that the controller is aware of where the printer head is. This is needed to know how much the motor should backtrack during the CR command.

5) Finally, during each CR, LF, the controller should be aware of any jamming issues that may have occurred during these operations, and send an interrupt to the CPU, with an error code.

6) If no errors have occurred, send an interrupt to the CPU letting it know that it has successfully completed.

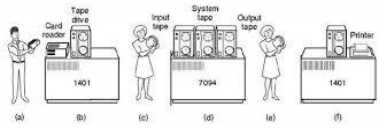
Additional notes: This assumes the computer we are dealing with has a SPOOLer (Simultaneous Peripheral Operation On-Line) and uses interrupts. In the case of an off-line system, the CPU may have to repeatedly check the buffer of the IOP responsible for the printer, to determine whether a print is done.

d) [0.4 marks] Explain briefly how the off-line operation works in batch OS. Discuss advantages and disadvantages of this approach.

Off-line operation in a batch operating system refers to performing input/output (I/O) work outside the main CPU’s processing cycle. In our lectures, we introduced this topic by using the two separate 1401’s for I/O, while the main expensive 7034 for the main computational tasksFor example, preparing a job using punch cards with FORTRAN code is an off-line

operation. The punch cards are collected and converted into magnetic tape, which is then fed to the main CPU via a buffer or a person.

Better worded→So before, we would use one main CPU and have the I/O devices as peripherals, the issue with this was that the CPU (which would cost 600$/Hr) would be waiting (polling) for the peripherals to be done. THe CPU is fast, and it takes a long time for the I/O devices to finish. The solution to this was to have two cheaper and slower CPUs that would be responsible for the I/O devices, while the main CPU would be responsible for computations.



An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

The issue with this now was that people had to move tapes between I/O and the main CPU, which is the bottle neck.

→solution: The batch OS! The batch OS where a special program called the monitor r (which has drivers in the OS section of main memory) which are responsible for loading and running and scheduling jobs.

Advantages:

1. The person working only now has to simply feed the input into the machine, and can go for lunch. The monitor will terminate jobs if they fail, and can move to the next job.
2. People are no longer required to move the cards from I/O to CPU to output.

Disadvantages:

1. More hardware needed
2. Users would have to wait until the job was processed by the CPU, then printed out by the other offline operator (printer) to find out if it worked or not.

Note: I could go into more detail on the pro's and con's if we were specified which implementation of the batch OS it was. The early version used magnetic tape as a buffer, and the loader would load it sequentially into the user program area of memory. In later

implementations, we introduced the spooler with interrupts and disc, but again that implementation has different pros and cons. So I had to be very vague about my advantages and disadvantages points above.

f) [0.2 marks] Write examples of four privileged instructions and explain what they do and why they are privileged (**each student should submit an answer for two instructions, separately, by the first deadline**).

Student 1

TIME MANAGEMENT (SET TIMER)

The TIMER is responsible for ensuring that each program does not exceed a certain amount of time. While this is useful for ensuring every program gets its fair share of compute power, it also solves the issues of avoiding "while (true)" loops which would essentially render the computer useful for everyone.

As such, this must be a privileged instruction as a user should not be able to set the timer value, essentially bypassing the program restrictions, potentially resulting in malicious intentions, such as executing "while (true)" and preventing the computer from advancing further

KERNEL MODE (SET MODE)

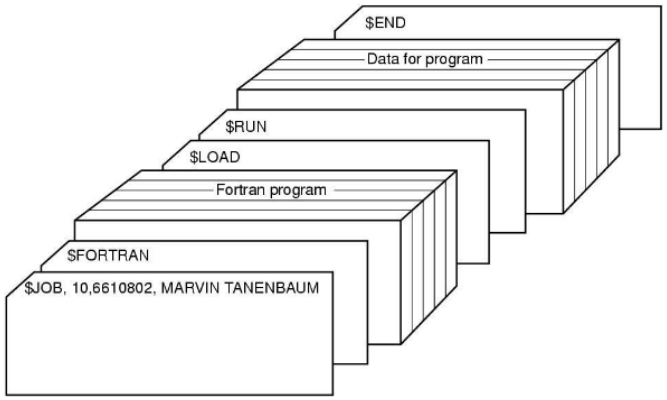
There is a buffer which holds a bit determining what the current privilege level is. If the bit is 1, that means the permissions are in program mode, or user mode. This means that the program cannot call any privileged instructions, such as modifying drivers, or directly interacting with I/O. However, if the user program can actually modify the bit responsible for determining the current permission, such as to a 0, which would be Kernel Mode. As such, the command for changing the permission level to kernel mode must obviously also be kept as a privileged instruction, as it would render privileged instructions useless otherwise. Only the operating system must have the permission required to enter or exit kernel mode.

Student 2 (me→ Braedy)

HLT: Stops the CPU from executing further instructions until a reset or external interrupt occurs. The reason this is a privileged program is because, allowing user programs to halt the CPU would freeze the entire system, affecting all processes. Only the OS should be able to do this, since it manages system resources.

IN / OUT (I/O port access): Reads from or writes to hardware I/O ports. For example, sending data to a keyboard controller or reading from a disk controller. It is privileged as it is working directly with the hardware of the system.

e) [0.4 mark] Batch Operating Systems used special cards to automate processing and to identify the jobs to be done. A new job started by using a special card that contained a command, starting with $, like:



For instance, the $FORTRAN card would indicate to start executing the FORTRAN compiler and compile the program in the cards and generate an executable. $LOAD loads the executable, and $RUN starts the execution.

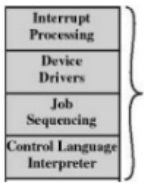
i. [0.2 marks] Explain what would happen if a programmer wrote a driver and forgot to parse the "$" in the cards read. How do we prevent that error?

ii. [0.2 marks] Explain what would happen if, in the middle of the execution of the program (i.e., after executing the program using $RUN), we have a card that has the text "$END" at the beginning of the card. What should the Operating System do in that case?

i) The $ operations tells the Fortran compiler that this is a JCL operation meaning that it will use the device's driver. For example, when you do $Job, ____... , then pass the program in and do $load, the $load uses the device drivers to load the program from the magnetic tape to the main memory (user program area). Then, when you do $RUN, it will again use the device drivers to move the PC to the start of the program and it will begin to execute. **So, if the user forgot to do the $, it would make it behave unexpectedly because RUN would be interpreted as input rather than a command.**

ii) lets say we had a bunch of jobs all queued up. If we had a $END at the beginning of the card, the program would halt execution there and, and the job sequencer would then set the PC to the next job (so essentially, all the code left on that program will be skipped over), and it will go to the next job (signified by $JOB).

g) [0.4 marks] A simple Batch OS includes the four components discussed in class:



Suppose that you have to run a program whose executable is stored in a tape. The command $LOAD TAPE1: will activate the *loader* and will load the first file found in TAPE1: into main memory (the executable is stored in the User Area of main memory). The $RUN card will start the execution of the program.

Explain what will happen when you have the two cards below in your deck, one after the other:

$LOAD TAPE1:

$RUN

You must provide a detailed analysis of the execution sequence triggered by the two cards, clearly identifying the routines illustrated in the figure above. Your explanation should specify which routines are executed, the order in which they occur, the timing of each, and their respective functions—step by step. In your response, include the following:

i. A clear identification and description of the routines involved, with direct reference to the figure.

ii. A detailed explanation of the execution order and how the routines interact.

iii. A step-by-step breakdown of what each routine performs during its execution.

i)

Job sequencer: Knows where the jobs are in the magnetic tape.

Job control block: Reads the next card and determines if it's a control card ($LOAD, $RUN, etc.) or program data.

Loader: Reads the executable from tape (TAPE1:) and loads it into the User Area of main memory

Device drivers: Used for card reader and printer to operate (in kernel side of OP). Additionally, the loader also uses it to read the magnetic tape and put in memory.

(ii and iii) → I explained both here.

1. **Job Sequence-** First component to act when the deck is processed. It also Knows the location of jobs on the magnetic tape and feeds cards to the Job Control Block (JCB).

2. **Job Control Block (JCB)**

- Reads the $LOAD TAPE1: card from the buffer.

- Identifies it as a control card.

- Signals Loader to execute.

3. **Loader**

- Uses device drivers to access TAPE1

- Reads the executable from tape and loads it into the User Area of main memory.

- Updates the job table (in JCB) to mark the job as loaded.

4. **JCB reads next card $RUN**

- Recognizes it as a control card.

- Calls the CPU Dispatcher / Execution routine to start the program.

5. **CPU Execution / Monitor**

- CPU executes the instructions in the User Area.

- Any I/O requests (printer, etc...,) are handled by device drivers.

- Monitor tracks program status and handles interrupts/exceptions

Summary of interaction between them:

Job Sequencer → JCB → Loader → Device Drivers → RAM (User Area) → CPU Execution.

h) [0.3 marks] Consider the following program:

```
Loop 284 times {
        x = read_card();
        name = find_student_Last_Name (x);  // 0.5s
        print(name, printer);
        GPA = find_student_marks_and_average(x); // 0.4s
        print(GPA, printer);
}
```

Reading a card takes 1 second, printing anything takes 1.5 seconds. When using basic timing I/O, we add an error of 30% for card reading and 20% for printing. Interrupt latency is 0.1 seconds.

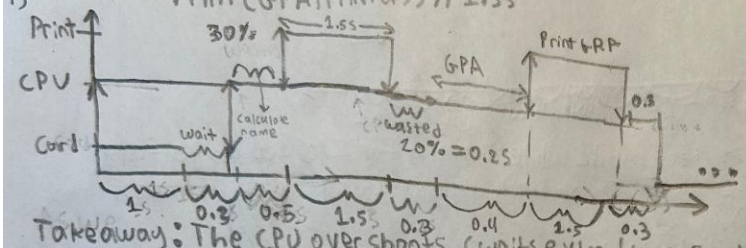
For each of the following cases: create a Gantt diagram which includes all actions described above as well as the times when the CPU is busy/not busy, calculate the time for one cycle and the time for entire program execution, and finally briefly discuss the results obtained.

i) Timed I/O
ii) Polling
iii) Interrupts
iv) Interrupts + Buffering (Consider the buffer is big enough to hold one input or one output)

Name: Braedy Gold Conlin, 101305254

```
loop 284 times {
    x = read_card();  // 1s
    name = find_student_last_name(x);  // 0.5s
    Print(name, Printer);  // 1.5s
    GPA = find_student_marks_and_average(x);  // 0.4s
    Print(GPA, Printer);  // 1.5s
```

i)

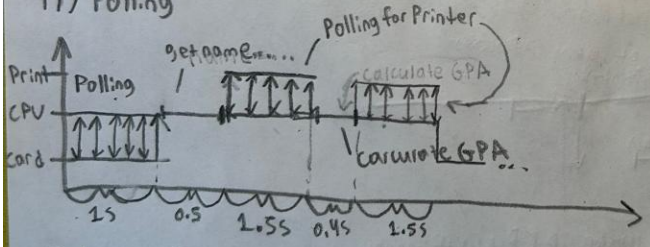


Takeaway: The CPU overshoots (waits extra time) for the printer and card readers to finish. Printer = 20%, reader = 30%, which is 0.3, 0.3 s.
→ wasted time can be seen in diagram.

total for 1 cycle = 1 + 0.3 + 0.5 + 1.5 + 0.3 + 0.4 + 1.8 = 5.8

284 cycles → 1647.02

ii) Polling

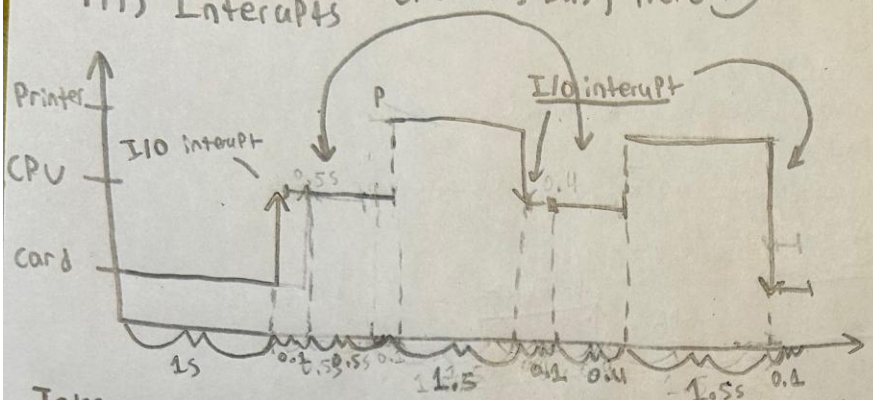


Takeaway: The CPU is either Polling or computing. we are wasting valueable time of CPU Polling. (money!!!).

1 cycle = 1 + 0.5 + 1.5 + 0.4 + 1.5 = 4.9

284 cycles = 1391.6

iii) Interupts CPU only busy here !!



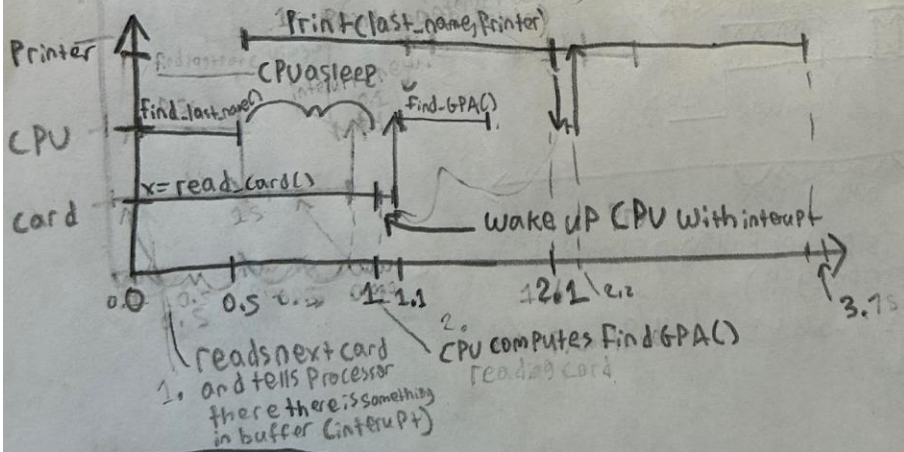
Take away → CPU is in sleep mode while waiting for I/O devices to finish (in which they wake the Processor back up, and it will continue to Process the next instruction).

Note: → This is slightly longer than Polling, but at least it saves money

1 cycle = 1 + 0.1 + 0.5 + 1.5 + 0.1 + 0.4 + 1.5 + 0.1 ≈ 5.2

284 cycles = 1476.8

iiii) Interupts with buffering



1 cycle = 3.7s

284 = 1050.8s

Part 2 - Simulation

Introduction

In the second part of this assignment, we were tasked with creating a simulation of a simple CPU which executes commands from a trace file. This simple system only includes three instructions: CPU, SYSCALL, and END/IO. The goal is to simulate the CPU doing some CPU bursts, then to make a system call, by using the ISR and the vector table to lookup the device id's address. Then, the ISR would either an I/O device, which would then simulate some device processing by looking up a device wait time, or a CPU system call. In the simulation part of the assignment, we are asked to vary the context switch time and the ISR activity time, and observe its effects on the program execution.

In order to accurately simulate as many cases as possible, three separate traces files were used to try and simulate three types of program (2 of which were seen in class, 1 being an intermediate step): CPU-bound programs, I/O-bound programs, and a mixed program. These are reflected in the trace as a program which does more CPU bursts, one that does more system calls to I/O devices (that take longer to run), and a mixed program which does a bit of both.

Varying the Context Switch Time

In the simulation part of the program, the first variable to vary is the context switch time, in milliseconds. Since the three trace files will have different global execution time, it will be inaccurate to simply compare their values with the context switch time. Instead, we will be comparing a percentage of the context switch time to the total execution time. This, along with any other non I/O device or CPU burst is known as "Overhead percentage", and measure what percentage of the program is wasted on overhead. We expect there to be more overhead in programs with more system calls, which is reflected subtly in the following chart. The effect may appear small, but that is only due to the small trace files provided. For example, the I/O bound trace file has 6 system calls, the mixed has 3, and the cpu bound has 2. Of course, on a much larger program, with for-loops, the effect WILL be more pronounced. However, we can still see the **slopes** of the overhead % grow as we move towards trace files that utilize more system files. Additionally, the overhead values themselves are larger the more I/O bound a program gets. We can thus conclude that increasing the context switch time will affect programs that make use of I/O system calls more than ones that calculate more.

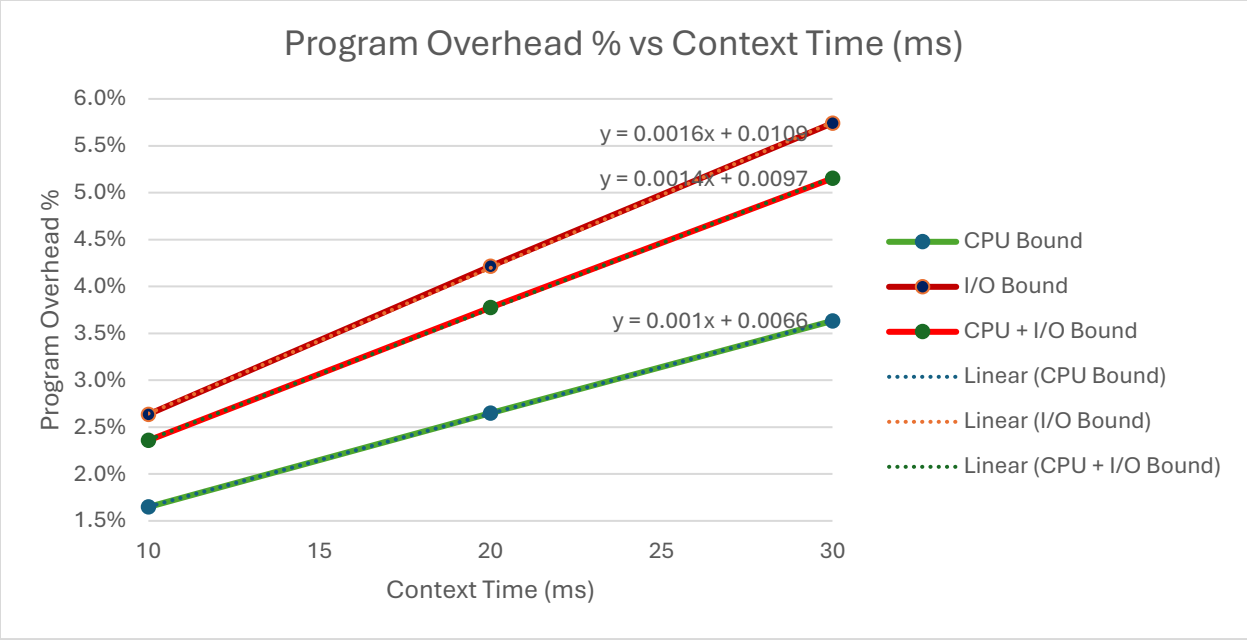


Figure 1: A chart demonstrating the overhead % slope between three trace files, using 10ms, 20ms and a 30m context switch time.

Trace Fiile + Context Switch	ISR activity time	TOTAL RUNTIME	Overhead Time	Overhead %
cpu_10	40	1942	32	0.016478
cpu_20	40	1962	52	0.026504
cpu_30	40	1982	72	0.036327
io_10	40	3642	96	0.026359
io_20	40	3702	156	0.042139
io_30	40	3762	216	0.057416
mixed_10	40	2036	48	0.023576
mixed_20	40	2066	78	0.037754
mixed_30	40	2096	108	0.051527

Figure 2: The table used to generate the chart, showcasing how the different traces have different total execution times, different overhead times, and their raw overhead % values.

Varying the ISR activity time

Just like with the context switch time, the ISR activity time represents the overhead time, in milliseconds, that the CPU takes to actually call the ISR. Then, after that is completed, if the ISR is a I/O device, the simulated wait time is fetched from the device table, and printed to the execution output. However, if we increase the time it takes for the ISR to execute, this

means that all system calls will take longer. Therefore, we should observe a similar effect than we've seen in the first experiment, where traces utilizing more system calls will have see a greater system call processing slope, or, in other words, increasing the ISR activity time will increase the overhead % more in the I/O bound programs then the CPU-bound ones:

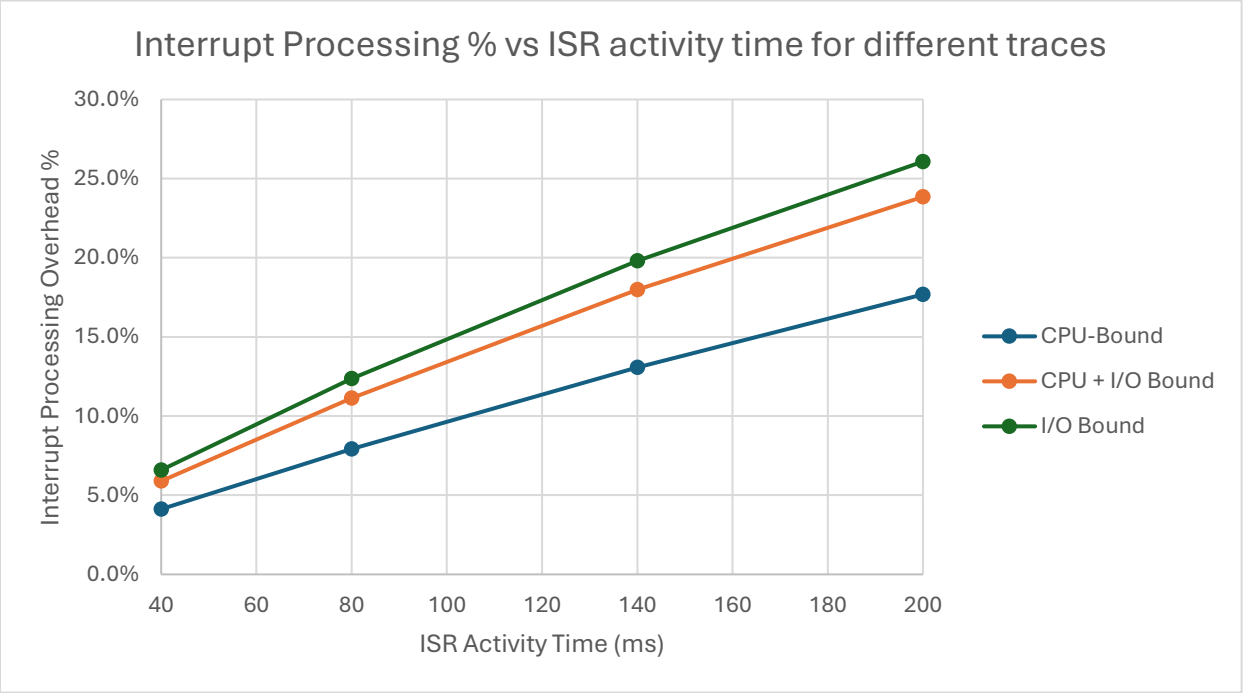
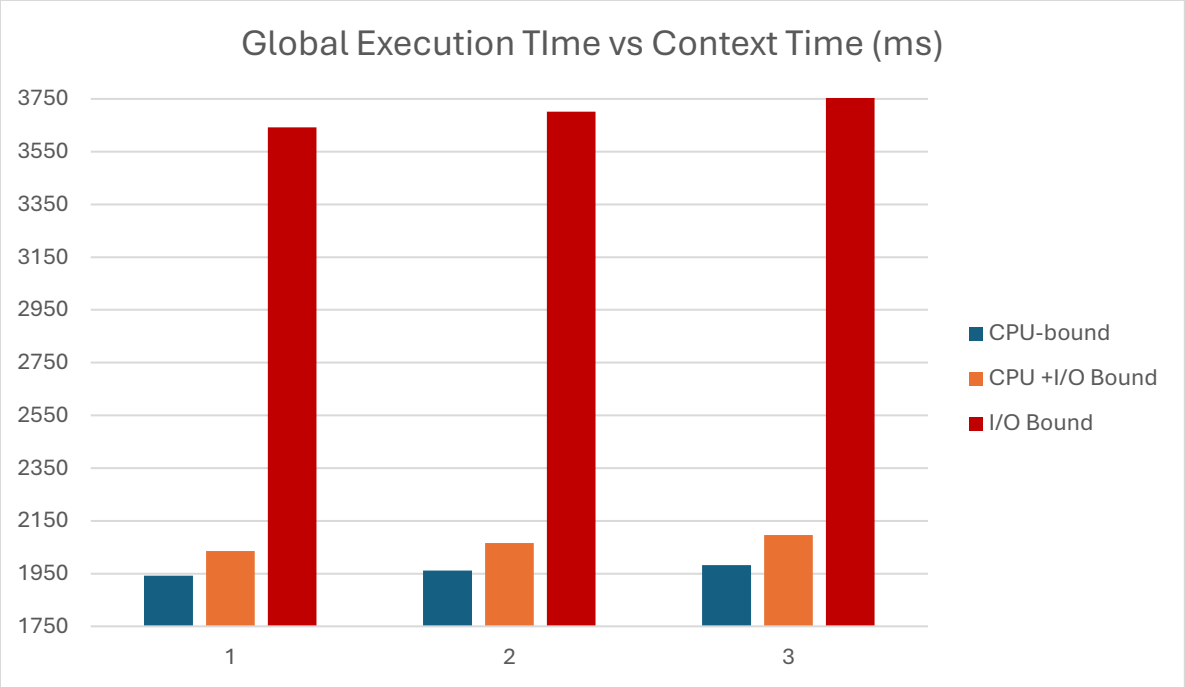


Figure 3: This chart showcases how the traces with more system calls which are more I/O bound have a much higher interrupt processing overhead, as they use more system calls.

Now, increasing the time it takes for ISR's to execute can be problematic. Whenever an ISR is running, no other ISR can be executed, which can create big queues if ISR's take too long to execute. This increases the latency of the entire system, become an expensive operation to do, when it should have been a quick, easy, and inexpensive method of doing a simulated software interruption.

Since this system is fully synchronous and there is not (yet) any low-level parallelism implemented, increasing any of the steps will increase the total execution time, and will do so at a linear rate compared to the amount of system calls. As such, this is why I/O bound applications suffer more from these increases in overheads, as the CPU is being less useful.


Global Execution TIme vs Context Time (ms)
3750
3550
3350
3150
2950
2750
2550
2350
2150
1950
1750
1
2
3
CPU-bound
CPU +I/O Bound
I/O Bound