Assignment 1 Group Report
SYSC 4001 A, L4

Justin Kim
101298662

Dorian Bansoodeb
101309988

Monday, October 6th, 2025

Part I

A. 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.

The interrupt signal begins with an external device such as an I/O. It generates a 5v pulse. If the CPU has interrupts enabled, the CPU can accept the interrupt, else the signal is ignored.

If the interrupt is accepted, the CPU switches from user mode to kernel mode, to allow the execution of privileged instructions.
The CPU will then save the current context such as the program counter, and registers. This saving of state and the jump to service routine are handled by the hardware. The program counter and processor status are pushed onto the stack at memory address 0.
The CPU then determines which device caused the interrupt by going down the interrupt vector table. The vector table provides the address for the correct Interrupt Service Routine (ISR) to initiate.
The CPU will then load the ISR address into the program counter. The execution jumps to the ISR.
Software:
The Interrupt Service Routine is written in the OS. It handles:

- Saving additional context such as registers
- Handling the event such as I/O completion
- If preemptive scheduling is used, move the interrupted process back to the ready queue, allowing the scheduler to pick another process.

At the end of the ISR, the instruction Interrupt Return is executed, which restores the saved program counter, restores the status and switches the mode back from kernel to user. Furthermore, it restores the saved general registers and stack pointers from the PCB. If preemption occurred, the scheduler loads a new process's context.

B. 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 the Interrupt hardware mechanism is used to implement System Calls.

A system call is a programmatic request made by a user's application to the operating system (OS) kernel. When a call is invoked, the CPU transitions from user mode to kernel mode, since privileged commands cannot be executed in user mode. The OS implements a system call interface, which acts as the intermediary between applications and the kernel, ensuring the user's program cannot directly interact with the computer's hardware or bypass the protections enforced by the kernel. In practice, programmers rarely invoke system calls; instead, they rely on standard application programming interfaces (API) and library functions. Three known system calls are open, read and write (please note, that the system call names are generic and different OS have different function names). These calls belong to the file management category, which provides controlled access to files. Specifically, open creates a file and establishes a reference in memory, reads and retrieves data from that file and copies it into a buffer in user space, and write takes the data from a buffer in user space and copies it into a file.

Hardware interrupts are signals from devices (usually) that cause the CPU to terminate the current execution and run an interrupt service routine (ISR). To accomplish this, the CPU has an implemented interrupt handler that validates the interrupt. The handler ensures the interrupt is enabled, has priority, performs a context change, saves the current state of the user's program and locates the ISR through the vector table. A system call uses the same mechanism but is instead initiated by a software interrupt otherwise know as a trap. This causes the same sort of interrupt context change into the kernel. The key distinction is that hardware interrupts are triggered by external events, while traps are explicit request by the user's program

C. i. In class, we showed a simple pseudo of an output driver for a printer. This driver included a generic statement: ”check if the printer is OK”. Discuss in detail all the steps that the printer must carry out.

When the command ”check if printer is OK” is executed, it is referring to a diagnostic routine for the printer's hardware before accepting data. These steps will vary from printer to printer, but as we discussed in class, this routine contains:

- Power on?: without power the printer can't run any instructions
- Paper available?: without paper there is no where to write
- Motors working/Ink check?: ensuring the mechanical components are functional
- Empty buffer?: the CPU ensures the buffer (of one character or one line) can receive more data

After completing all checks, if the printer passes, it will generate a ”READY” flag (name is generic) and will begin receiving data to proceed with the printing job

C.ii In class, we showed a simple pseudo of an output driver for a printer. This driver included a generic statement: “print (LF,CR)”. Discuss in detail all the steps that the printer must carry out.

In the printer driver, there is the line print(LF,CR). LF stands for Line Feed, and CR Stands for Carriage return. Together, they act similarly to a \n in code, or resetting a typewriter to the next line. The printer recognizes the system call print(), which is a formatting command. The printer's controller will set the appropriate signals for the mechanical components of the system.
The CR brings the head for printing back to the starting point, to the left. LF engages the motor to advance the paper vertically. The combination of using LF & CR prepare the printer for the next line of text to be printed. Once the motions are complete, the printer's sensors confirm the alignment, and signals that the printer is ready for the next line of data.

D. Explain briefly how the off-line operation works in batch OS. Discuss advantages and disadvantages of this approach.

In batch OS, off-line operations mean that the input and output (I/O) operations were done and handled separately from the main CPU. There are 3 phases in the process.

1. The Input Phase: Programmers brought a deck of punchcards or punchtape to be read in and read onto a magnetic tape.
2. The Processing Phase: The main CPU read the tape, processed and calculated the information, and finally placed the output onto another roll of magnetic tape.
3. Output Phase: The magnetic tape is transferred to a third system, where the magnetic tape is read and the results are printed.

Advantages:

- Maximized CPU utilization: The CPU can focus on only processing information, while the cheaper CPUs perform the input and output processes.
- Higher Throughput: Many jobs can be placed onto one roll of magnetic tape as a batch to increase the main CPUs productivity.
- Reduced setup time: Operators had to perform less setting up, and loading and unloading jobs could be done automatically instead of manually managing each one.

Disadvantages:

- Long turnaround time: Users had to wait for their batch of cards to be processed off-line before seeing results as the whole batch needs to be processed and then outputted together.
- No user interaction: When the batch starts, it runs automatically, making debugging impossible while running.

E. 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 $. 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. 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?
The $ sign in batch OS signifies that this is the start of a control card like $JOB, $LOAD and tells the system this card contains a command, not user programming data. The parser needs to recognize the $ symbol to treat it accordingly.

If a programmer wrote a driver and forgot to parse the $ symbol, the card would be treated as a part of the code, as normal user programming data instead of a control instruction. If this were to happen, the system would not start or end jobs correctly and would cause crashes, incorrect job sequencing, or overwriting another user's job.

To prevent this type of error, executing these commands are a privilege of the OS, not the user, only accessible within Kernel mode. Only kernel mode drivers can access hardware. The user performs a system call to request kernel services.

ii. 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?
The $END card is a control card meant for the job scheduler, not for the running program itself. Therefore, if a card with the text $END is in the middle of the program during execution, the OS will treat the card as a JCL control card. This OS will immediately terminate the current job, which stops the execution of the current program whether the tasks are completed or not. The OS will now move on to the next job in the batch or halt processing until new instructions are provided.

It would be ideal if the OS could recognize that it is currently running a process, therefore treating any of the input cards as data for the program. In this scenario, the $END control card is treated as input data for the program rather than as a command to terminate the job. This means the OS should only interpret control cards as commands when it is actively looking for job control instructions.

F. Write examples of two privileged instructions and explain what they do and why they are privileged
Dorian
The two privileged instructions I will explain are Interrupt handling & mode.

Interrupt handling saves the CPU state, program counter and registers, and transfers control to the ISR given the Vector Table. The OS determines which interrupt occurred and executes the correct handler. This instruction is privileged as the OS should control how the interrupts are being handled, otherwise user programs could have control, and corrupt the system state, or programs of other users.

The mode is when a CPU can distinguish between user mode and kernel mode. System calls are made to switch from one mode to the other. The kernel mode is useful for executing privileged actions such as opening or reading files. This instruction is privileged as it prevents user programs to directly access privileged instructions, but now have to request access through the OS.

Justin
Privileged instructions are CPU instructions that can only be executed in kernel mode.They typically perform operations that require direct access to the hardware. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute it. One example of privileged instructions are I/O control instructions. These instructions directly control I/O devices such as disks or printers. They are privileged because it guarantees the security of the system when accessing the hardware. A user’s program must make a system call to request a I/O operation. The OS then validates the user’s request and executes the privileged instructions on the program’s behalf. Another known privileged instructions are timer management instructions. The timer ensures fair CPU resource allocation and scheduling. Without these instructions being privileged, a user’s program could stop, disable or indefinitely increase the timer value

G. A simple Batch OS includes the four components discussed in class: Interrupt handler, device drivers, JS and CLI.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.
i.

ii. 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.
While the control language interpreter (CPU) is continuously running and reads the first card $LOAD TAPE1:

1. CLI identifies the $LOAD as a control card. Immediately calling the loader routine and passing the device identifier.
2. Immediately after the CLI passes control to the loader, it prepares the tape hardware in order to perform data transfer.
3. After the loader passes control to the tape driver, it issues commands to the tape hardware to initiate IO operations.
4. Because the tape driver takes a long time (not milliseconds or microseconds) physically transferring the IO's, the tape driver will return control to the CLI after issuing its hardware driver commands.
5. The CLI will now wait for the tape driver to finish.
6. Once the tape driver has completed its task, it generates a hardware interrupt that will transfer control to the interrupt handler.
7. The interrupt handler invokes the interrupt processing mechanism that will save the CPU current state and perform a context change.
8. The interrupt handler will conclude that the interrupt came from the tape driver which signals its task are complete. It then restores control to the CLI

As the CLI in now probing again, it will interpret the second card $RUN

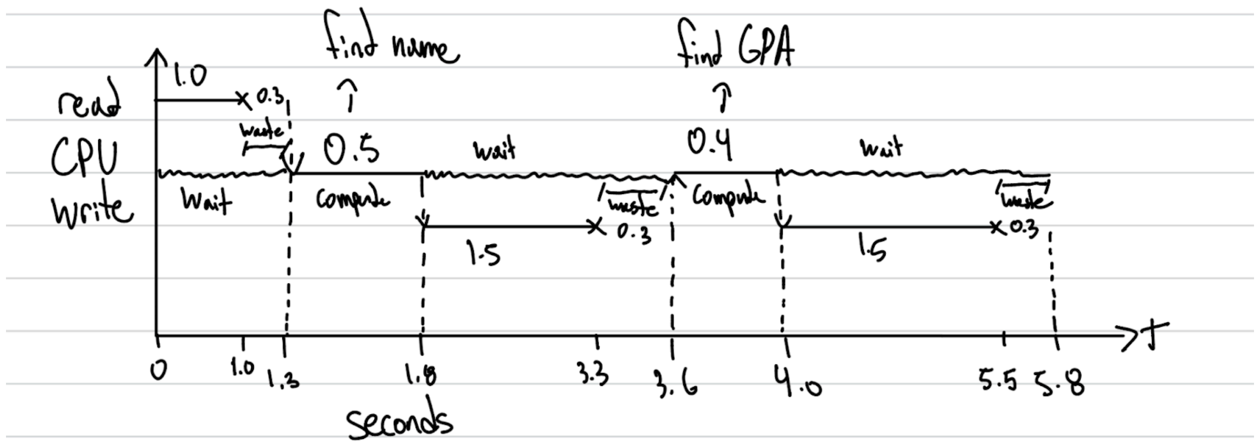
1. This transfers control to the job sequencer (JS) Its function is to manage job flow.
2. Since the program has been successfully loaded into the user area by the previous steps, the JS now sets up the execution environment. Like the interrupt processing mechanism, it will save the CPU's current state and perform a context change
3. The user program can now execute all of its instructions. This is the main reason to run these two cards.
4. When the user program terminates, it must return to kernel mode. This is once again done by the JS
5. It then returns control to the CLI so the OS can process the next card in the deck

H. Consider the following program provided in Assignment 1 Part II.
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.

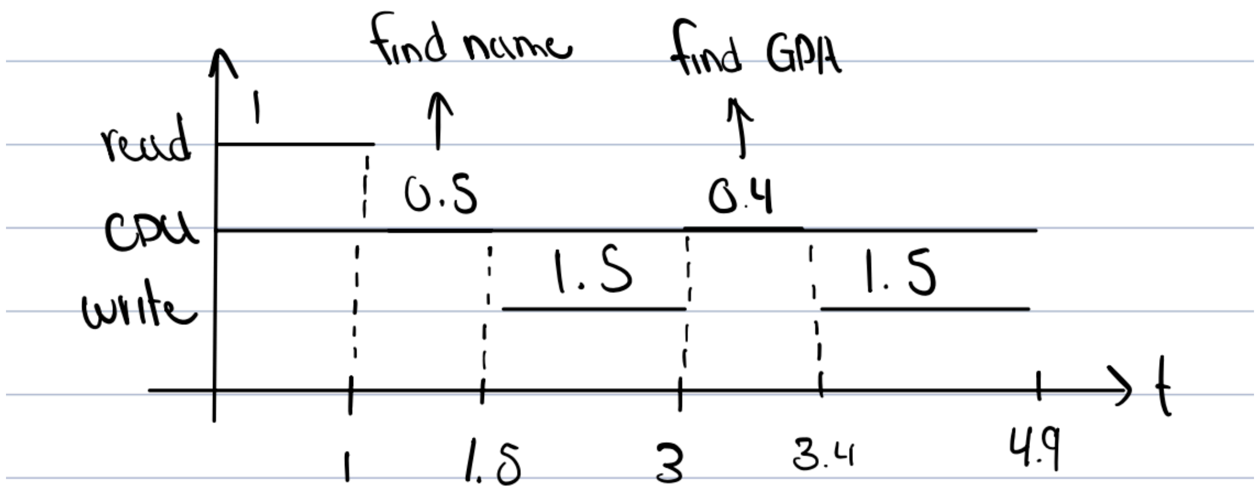
For each of the following cases: create a Gantt diagram which includes all actions described above as well as the times when the CPU is busy/not busy, calculate the time

for one cycle and the time for entire program execution, and finally briefly discuss the results obtained.

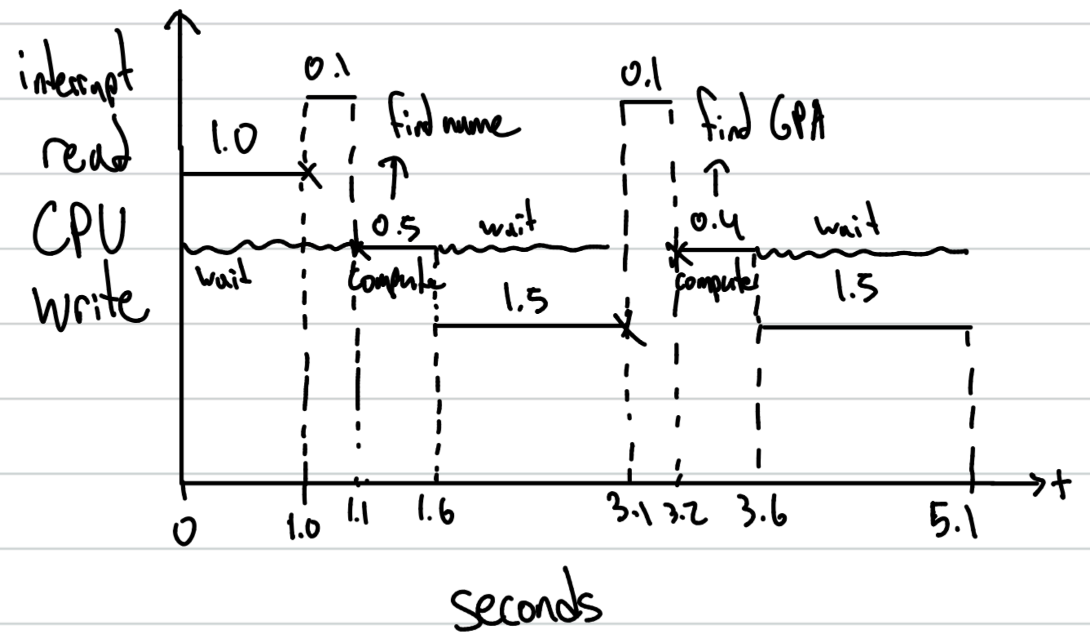
i. Timer



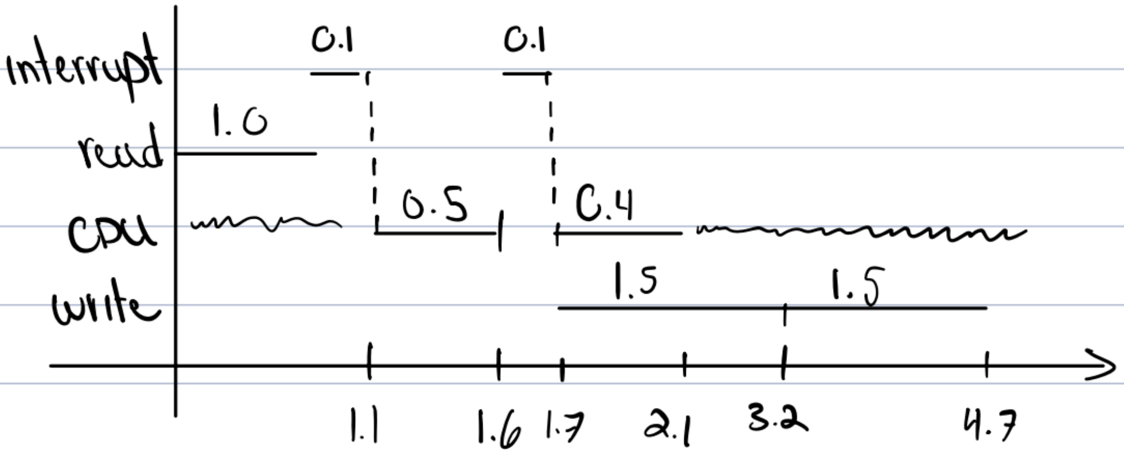
ii. Polling



iii. Interrupts



iv. Interrupts + Buffering



From the Results Obtained, the order of time it took to complete 1 cycle from most amount of time to least is as follows: Timer (5.8 seconds), Polling(4.9 seconds), Interrupts(5.1 seconds), Interrupts + Buffering (4.7 seconds)). Each case has their pros and cons. The combination of interrupts + Buffering provides the highest CPU efficiency by allowing the processor to continue computations while input and output operations occur simultaneously.

Part II

Github Repository Link:https://github.com/Dorianbansoodeb/SYSC4001_A1

Objective: The goal of this assignment is to implement a simulator in C++ that models how the interrupt handling process works in a batch operating system. The simulator tracks input/output operations, CPU activity, and all events involved in the interrupt cycle. The system's goal is to track and analyze the performance by changing different parameters such as ISR execution duration, if a faster/slower CPU was used and context switch time.

Methodology: The interrupt handling simulator was implemented using the provided files in C++ in the repository (interrupts.cpp + interrupts.hpp). The program reads a trace.txt file line by line to simulate the events in Batch OS such as CPU running, and I/O events.

Each line in the trace file contains an activity (CPU, SYSCALL, END_IO), which are all treated differently and a number, which is treated differently based on the activity. The corresponding actions from each line are logged in an output file (execution.txt for example).

The simulation is able to model the stages of the interrupt process by keeping track and incrementing a running time clock (now_ms), keeping track of time as more actions are performed. Some of these actions are:

- Switching to kernel mode (1ms)
- Vector and ISR Lookup (1ms)
- Data Transfer (40ms)
- IRET (1ms)

For each interrupt event, the simulator retrieves the timing and vector information from the device_table.txt and vector_table.txt files. These files contain the delays and ISR addresses to be searched up. The results are saved line by line for each step, into an output file (execution.txt for example). A line would contain entries, separated by commas showing the timestamp, duration of the event, and the description of the event. (0, 51, CPU burst) for example.

This simulation allows us to observe how changing timings of different parameters affects the overall performance of the execution time and system .

Simulation #	Change	Effect
1	Base case	
2	Base case	
3	Base case	
4	Base case	
5	Base case	
6	Context change 30ms	CPU waits for 60ms each int. cycle
7	ISR cost 100ms	Increases program time
8	ISR cost 150ms	Increases program time
9	Base case	
10	Base case	
11	Base case	
12	Base case	
13	Base case	
14	Base case	
15	Base case	
16	Base case	
17	Base case	
18	Context change 20ms	CPU waits for 40ms each int. cycle
19	ISR cost 200ms	Increases program time
20	Context change 10ms	CPU waits for 20ms each int. cycle