SYSC 4001 Assignment 1 Part II

Student 1 Aveen Majeed 101306558
Student 2 Arshiya Moallem 101324189

Oct 6, 2025

Part 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.

Interrupts are given by a device. The device asks for help by sending a signal to the CPU (CPU acts as the brain). The CPU pauses its task and looks up the address of the appropriate Interrupt Service Routine (ISR). The operating system (OS) now runs the program found in the ISR to handle the event. The OS then restores the CPU's saved information, and the CPU goes back to completing the task. The last two events that include the operating system are done by software. The sent signal, CPU pause, and lookup of the ISR address are carried out using hardware.

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]**

A system call is a programmatic service request sent by the user via a user-level program to request a service from the operating system's kernel. This process is the stereotypical way of the computer to communicate (the interface) with the operating system, because it is impermissible for the system calls to grant any direct access from the user mode.

Three examples:

1) open(): The system call allows the user to gain access and open a file so that it can either read data or write data. Ultimately, the application provides the path to the file and flags to indicate the access/role when in access mode (e.g., read-only, write-only, etc). The kernel from here would check the permission and accessibility of the user to see if it can access the file in the request. If the access mode has those roles, it can locate the file on the disk and retrieve the file path to open the file.

2) read(): The system call is utilized to read the data/content contained in an open file. Its purpose is to read the data (the number of bytes) from the open file and store it in a buffer in memory. The kernel is then responsible for finding the right file, reading the number of bytes from the disk memory, and storing it in its own memory. The reason is that it can at least copy this data from the kernel's memory into the buffer and return the number of bytes that were read.

3) write(): The system call is utilized to write data/content into an open file. The kernel copies the data (the number of bytes) from the application buffer and stores the data on a disk. Then it returns the number of bytes written.

System calls are related to Interrupts because they are implemented using a software-generated interrupt. Interrupts send a signal to the CPU, either by software or hardware, that alerts the CPU. The CPU responds by temporarily suspending its current activity, saving its current state, and executing the Interrupt Service Routine (ISR) or interrupt handler. Once the handler or ISR is finished, the CPU restores its saved state and resumes the interrupted task. These interrupts could be an issue with the physical hardware component of the device or could be an issue caused by the software (often referred to as traps).

The Interrupt hardware mechanism is used to implement system calls as a controlled gate into the OS kernel. When a user application gets executed, a special instruction generates a software interrupt. This means that the instructions trigger the CPU to automatically switch into kernel mode, saving its predefined state and going to a system call request within the OS. From there, the handler or ISR, now in kernel mode, successfully performs the system call request. Once it is finished, the kernel executes a return instruction to the CPU. The CPU then saves its current state, switches back to user mode, and resumes the user application. This process is useful when creating a secure, safe, and controlled environment when dealing with system calls.

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

Steps to ensure the printer is ok:

1) Check the hardware condition: The controller sensors verify the physical integrity of the paper, whether there is paper, if the paper is jammed, the toner, and the cover status. If there are issues raised, the result will be encoded into the status register, and the status flag will be set to Not OK.

2) Check the software condition (Driver/OS): The driver in the CPU reads this result encoded into the status register from the I/O port. If the resulting bit is set to BUSY, the driver must wait; If the resulting bit is set to error, the driver triggers a recovery action to obtain its saved data.

If the resulting bit is set to READY, the printer is OK, meaning that the hardware status indicates no active error flags. This set of steps in the function ensures that the driver can now send the next character.

ii. print (LF, CR)

When the driver sends print(LF, CR), the printer does two things. The first is Line Feed (LF), where the roller inside the printer turns so the paper moves up by the height of one line. This makes space for printing the next row. The second is Carriage Return(CR) where the printer

moves the print head back to the start of the line on the left side. Together, LF and CR prepare the printer so it is ready to start printing on the next line.

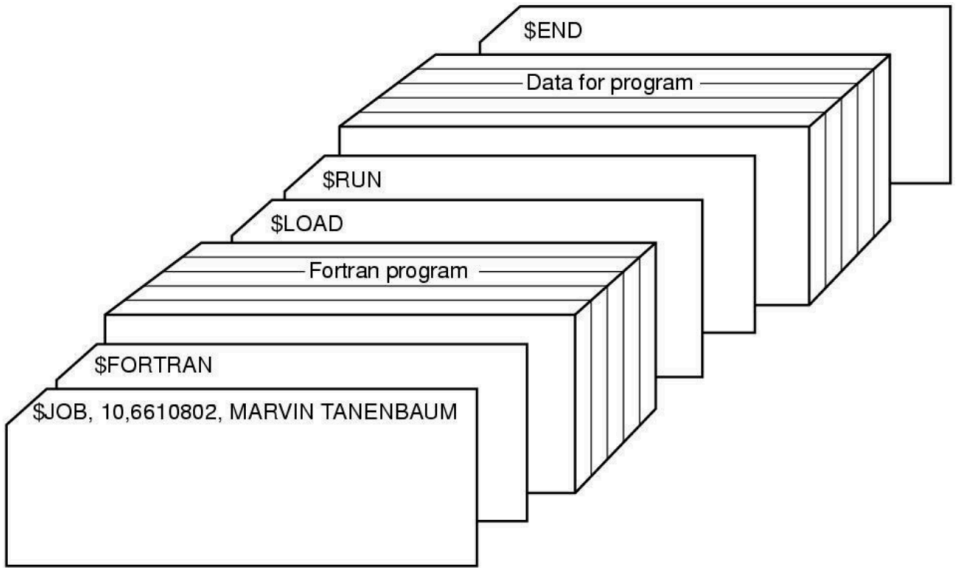
Discuss in detail all the steps that the printer must carry out for each of these two items (Student 1 should submit answer ii., and Student 2 should submit answer i.).

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

The off-line operation works in batch OS functions by overcoming the speed mismatch between fast CPU vs. slow I/O devices with multitasking (the Spooler), since the CPU has to wait to transfer the outputs into the slow I/O device when available.

off-line operation works in batch OS	
Advantage	Disadvantage
- Increased CPU Utilization: CPU spends less time waiting for the I/O device, boosting its throughput and cost-effectiveness. - Eliminates the I/O bottleneck: The CPU can process jobs from high-speed tapes since I/O is being handled by separate offline devices and avoids being slowed down by the relatively slow card readers, printers, or tape drives. - Increased Throughput: Reducing idle CPU cycles by batching jobs together on tape instead of manual intervention between them.	-Lack of Interactivity: The User has no way of interacting with their batch once submitted, making it a slow process for debugging. - Inefficient Debugging: The User has no way of knowing if their program is successful until the entire batch cycle is completed. Any errors would cause the program to fail and would only be found out at the end of the cycle. -High Cost for Hardware: Increased cost off-line devices due to the requirement of additional computers dedicated solely to I/O.

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?

If a programmer wrote a driver and forgot to parse the “$” in the cards read, the special control card would be treated as a regular input/data card and would not be able to be distinguished from the other regular input/data cards. The problem with this is that the program would not switch modes, and the control card would be fed into the compiler, creating unintended behaviour or compilation errors in the program. This batch monitor would never start the request since there is no parsing of the “$”.

To prevent this error, the batch monitor checks the first character to see whether we get “$” or not, to distinguish their roles in the program (whether it is data or control).

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?

If, in the middle of the execution of the program, we have a card that has the text “$END” at the beginning of the card, the running program would abruptly terminate. To deal with

this dilemma, the Operating System would need to add code to the script that must disable any interpreter prompt (first character in a control card, aka “$”) made right during the “$RUN” execution, treating the “$END” card as a data card rather than an interpreter prompt.

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).**

1) Set Timer

What it does: This instruction changes the system timer so the operating system knows when to take back control. The timer lets the OS decide when to switch between programs and make sure each one gets a fair amount of CPU time.

Why it is privileged:If a regular user program could change the timer, it could stop the OS from interrupting it and basically run forever. That would make the whole system freeze or become unstable. That’s why only the operating system is allowed to use this instruction.

2) Load system control registers

What it does: This instruction loads a new physical memory address into a CPU register, effectively changing the memory map of the virtual-to-physical since it points to the base of the page-table base register.

Why it is privileged: A user-level program to change its own table would grant it access to any physical memory location, meaning it could access and possibly append or overwrite the memory of other processes and the kernel memory. Therefore, the OS must solely control any and all registers between the processes.

3) Halt CPU / shutdown

What it does: This instruction stops the CPU’s execution cycle of instructions and shuts down/turns off the computer until a reset or a specific interrupt occurs.

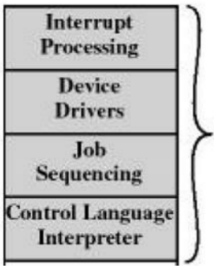
Why it is privileged: If a user-level program had the capacity to shut down the computer, it could lead to data loss or filesystem corruption, accidentally shut down the computer, and cause a denial-of-service for the user. The capability to shut down the computer is a fundamental privilege of the OS

4) Interrupt Management

What it does: This instruction turns hardware interrupts on or off. The OS uses it to control when interrupts happen so that important operations can finish without being disturbed.

Why it is privileged: If a user program could control interrupts, it could block the system from handling important signals like input, output, or timing events. That would make the OS stop responding and could crash the whole system. For that reason, only the operating system can perform interrupt management.

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.

1. Referencing the figure, this process begins with Control Language Interpreter. This part interprets the control cards and reads the $LOAD TAPE1: and $RUN. It reads the instructions from the job deck and decides what to do next.
2. Next is Job sequencing. This component is in charge of managing the order of when jobs are executed. Its main jobs include ensuring a new job begins automatically after the last one finished and coordinating transitions between different operating system routines.
3. Device Drivers handle communication between the operating system and devices. The tape drive driver transfers the executable program from TAPE1 into main memory.

4. Interrupt processing handles all interrupts that occur while a program is being loaded or executed. It allows the operating system to stop the running process, process the interrupt and then resume again after.

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

When the system processes the job deck that includes the two control cards `$LOAD TAPE1:` and `$RUN`, here's what happens step by step:

1. The Control Language Interpreter first reads the command $LOAD TAPE1 that asks the system to load a program from this tape device.
2. The device driver is called and reads the executable file from tape1 and loads it into the user area of main memory.
3. After the loading is done, job sequencing updates the job list and says the program is ready to run.
4. The Control Language Interpreter now reads the next card $RUN which tells the system to start running the program that was loaded.
5. Then the job sequencing routine changes control from the operating system to the user program by setting up the CPU and program counter.
6. Lastly, while the program is running, if any I/O requests or system events happen, the Interrupt processing handles them before returning back to the user program.

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

1. Control Language Interpreter: This system starts by reading the given command cards. The control Language Interpreter recognizes this as a request to load the program, so it calls the loader routine to start the process.
2. Device Drivers: The tape driver is activated, it reads the executable file on Tape1 and transfers it into the user area of main memory where programs are stored.
3. Job Sequencing: After the program is fully loaded, it updates the queue to say that this job is ready. It prepares the system for the next command, so the next program can be executed right away.
4. Control Language interpreter (again): Now it reads the second command card $RUN. It recognizes this command as an instruction to start executing the program that was just loaded into memory.
5. Job Sequencing: Job Sequencing now takes control and starts the program by setting up the CPU registers and program counter. Control is then handed over from the operating system to the user program.
6. Interrupt Processing: While the program is running, any I/O requests, errors, or system events are handled through interrupt processing. The system temporarily pauses the program to deal with the interrupt, then returns control back to the program to continue running normally.

h) [0.3 marks] Consider the following program:

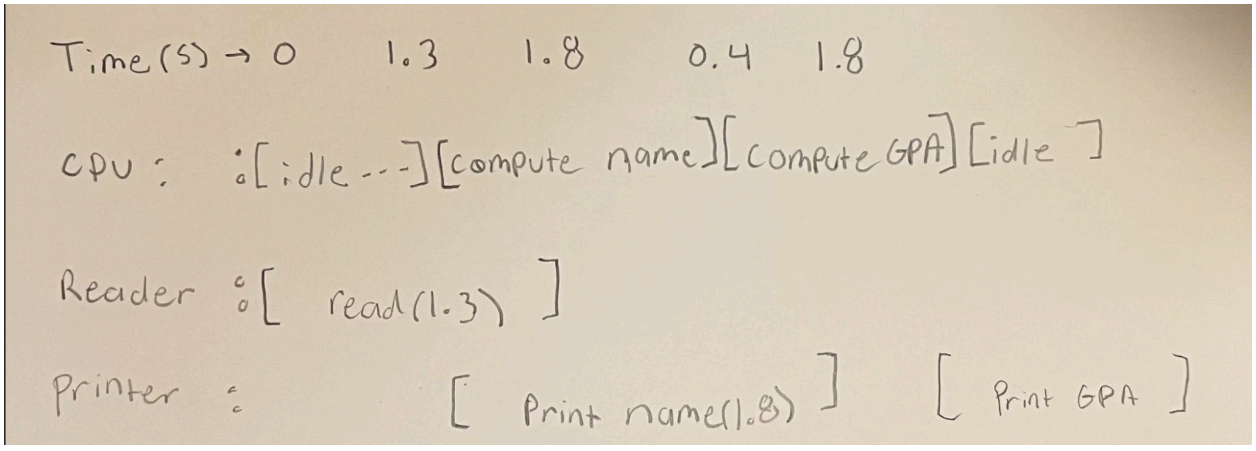
```
Loop 284 times {
        x = read_card();
        name = find_student_Last_Name (x); // 0.5s
        print(name, printer);
        GPA = find_student_marks_and_average(x); // 0.4s
        print(GPA, printer);
}
```

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

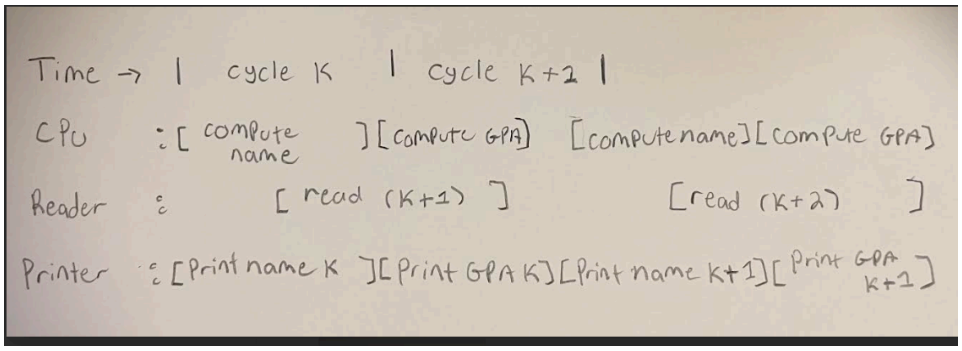
In timed I/O, the CPU starts an input or output operation and then waits until the operation finishes before doing anything else. There is no overlap between CPU and I/O. With the added timing errors, reading takes 1.3 seconds and printing takes 1.8 seconds. One full cycle will take 1.3 + 0.5 + 1.8 + 0.4 + 1.8 = 5.8 seconds.
Since the loop repeats 284 times, the total time is 284 × 5.8 = 1647.2 seconds. The CPU is only active during the two compute operations (0.5 + 0.4 = 0.9 seconds) and idle for 4.9 seconds each cycle, meaning it is only about 15% utilized.
To summarize the CPU uses a lot of time waiting and total execution time is very high.



ii) Polling

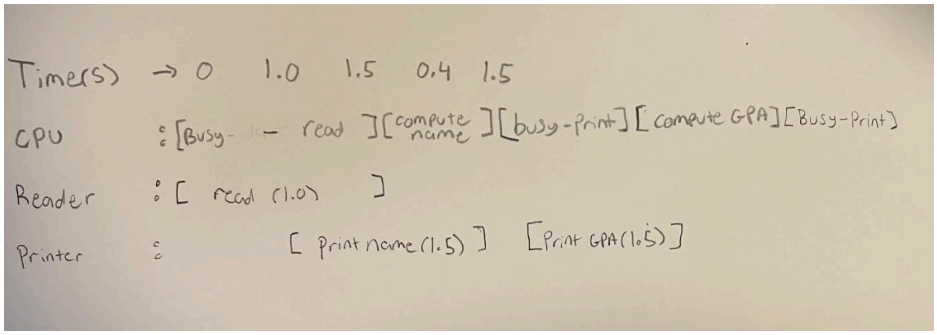
When using polling the CPU continuously checks the device status until it is ready, so there is still no overlap between I/O and computation. This makes it similar to timed I/O except that the

CPU is busy waiting instead of idle. The total cycle time is 1.0 + 0.5 + 1.5 + 0.4 + 1.5 = 4.9 seconds, and the total program time is 284 × 4.9 = 1391.6 seconds. Although the CPU is technically active during I/O, it is not doing useful work so efficiency remains low. Even though polling removes the idle time, it does not improve total speed since the CPU wastes time checking device status repeatedly.



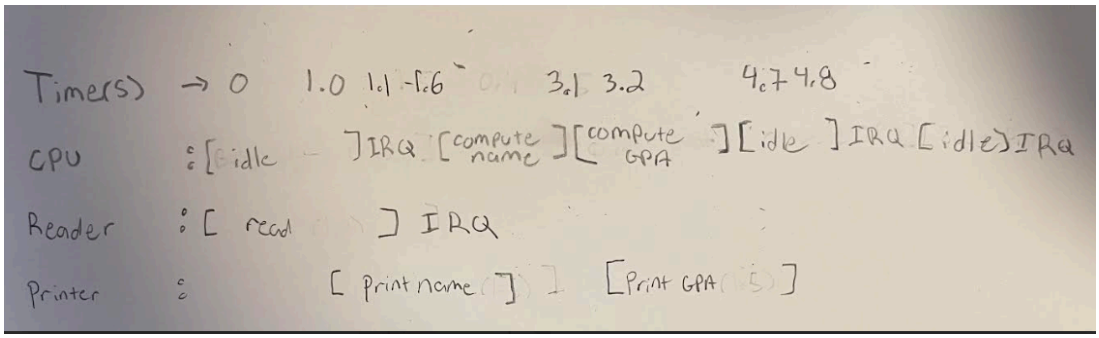
iii) Interrupts

When using interrupts, the CPU initiates an I/O operation and continues with other tasks while waiting for an interrupt signal from the device. When the device finishes, the interrupt handler runs briefly (taking 0.1 seconds). This allows overlap between computation and I/O. The CPU waits for the read to complete before computing, then performs computations while printing happens in the background. Each print and read generates an interrupt. The total time per cycle is about 4.8 seconds (including 0.3 seconds of interrupt service time). Therefore, the total time is 284 × 4.8 = 1363.2 seconds. With interrupts, CPU utilization increases significantly because the processor performs work while I/O is ongoing instead of waiting.



iv) Interrupts + Buffering (Consider the buffer is big enough to hold one input or one output)

When interrupts are combined with buffering, the CPU, printer, and reader can work almost all together. The buffer temporarily holds data so that while one record is printing, the next record can already be read into memory. The slowest device (the printer) limits the overall speed. Each print operation takes 1.5 seconds, and since there are two prints per record, the printer determines the overall cycle time of around 3.0 seconds. Interrupts (0.1 seconds each) are handled in the background and do not affect total cycle time. Therefore, the total time is about 284 × 3.0 = 852 seconds. With buffering, input, output, and computation overlap, and the CPU rarely waits. The printer becomes the bottleneck, so the total time per record depends on print time rather than processing or reading


Time(s) → 0 1.0 1.1 1.6 3.1 3.2 4.7 4.8
CPU : [idle] IRQ [compute name] [compute GPA] [idle] IRQ [idle] IRQ
Reader : [read] IRQ
Printer : [Print name] [Print GPA]

Part 2: report.pdf

We will submit test cases in BrightSpace; the TAs will mark these cases first. You must include your own test cases as well. The program must be written in C++ (or C, they are mostly backwards compatible).

You must run numerous simulation tests and discuss the influence of the different steps of the interrupt process:

- Change the value of the save/restore context time from 10, to 20, to 30ms. What do you observe?
When I change the value of the save/restore from 10 to 20 and then again to 30, the total execution time grows in a linear way with the number of interrupts in the trace. If a trace has N interrupts then going from 10 to 20 adds another 10XN ms to the completion time. Nothing else changes the interrupt path.

- Vary the ISR activity time from between 40 and 200, what happens when the ISR execution takes too long?
Increasing the ISR time always slows END_IO by the same amount as the increase, because END_IO time is the ISR time and device delay combined. So if I raise the ISR from 40 to 200 every END_IO gets an extra 160ms slower. For SYSCall, after it runs ISR it does up to 40ms of transfer and the device spends any leftover time checking for errors. So the effective SYSCALL depends on the device. In my runs, device5(68ms) jumped from 8-ms at ISR = 40 to 250ms at ISR 200. The total runtime grows fast if the trace has a lot of them while END_IO always pays the full increase.

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

The total time is just the sum of three things: the cpu bursts in the trace, the OS works around each interrupt, and the device works. if i change the speed of a step, i only grow the part it belongs to. When I raise save/restore context from 10 to 20 to 30 ms, the finish time goes up by 10 ms times the number of interrupts since every interrupt pays that once. When I increase the ISR time, every END_IO slows by exactly that increase, and syscall slows on small/medium devices until it basically acts like ISR + 40 ms. On really big devices the device delay dominates so changing ISR barely changes the total time. cpu bursts don’t change unless i change the numbers in the trace. So overall: more interrupts make the run sensitive to context time, lots of END_IO or medium-delay devices make it sensitive to ISR time, and heavy device delays hide ISR changes.

GITHUB URL_LINK: https://github.com/arshiyamoallem/-SYSC4001_A1