SYSC 4001 Operating Systems Fall 2025

Assignment 1 - L1-14

Mark Bowerman 101272081
Joshua Heinze 101272848

2025/10/03

Github Link: https://github.com/MobDude/SYSC4001_A1/tree/main

Part One - Concepts

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.**

First, a device sends an interrupt signal through a wire. This interrupt signal is sent to a register in the computer. The computer will check the enabled/disabled bit before checking interrupts to see if it is accepting interrupts. If enabled a circuit will perform a context save. This will save all the data pertaining to what is running on the cpu such as the PC, register contents, status registers, etc. It saves all of this information onto the stack. The CPU is also put into kernel mode. The context switch then pulls the data for the ISR (interrupt service routine) for the I/O device giving the interrupt. It does this by looking through the vector table for the section that is for the ISR of the device. This section has to at least contain the PC for the ISR. This will tell the CPU where to jump to for the ISR. The ISR is then run. The ISR is a software that can do any number of things based on the input. Once the ISR is complete, the status register and program counter as well as all other data that was put onto the stack is pulled off. The CPU is also put back into user mode.

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 how the Interrupt hardware mechanism is used to implement System Calls.**

A system call is how a user program requests a service that it does not have authority to do itself, and must request that the OS kernel do for it. User programs don't have the authority to directly interact with hardware or manage critical resources, so when they need to do things like open a file, write data, or start a new process, they use system calls to hand the request over to the kernel. In Linux, things like open() to open a file, write() to send data somewhere, and fork() to create a new process are all examples of system calls.

System calls are similar to interrupts and they work in a very similar fashion, although they come from software instead of hardware. When a program makes a system call, it triggers a software interrupt or trap. This forces the CPU to switch from user mode to kernel mode so the OS can take over safely. The CPU then looks up the system call in the vector table, and runs the ISR for that request. Once its done it switches back to usermode so the program can keep running.

c) **In class, we showed a simple pseudocode of an output driver for a printer. This driver included two generic statements. Discuss in detail all the steps that the printer must carry out for each of these two items.**

i) **check if the printer is OK**

When the driver asks the printer to check if the printer is ok, the printer does a series of checks before reporting its status. It checks things like that it has power, it has paper and ink, and that it is not currently jammed. If all of the conditions for running are met, it returns an OK status and the driver can continue onwards. Otherwise it would raise an error and presumably this would alert an operator or run another program to resolve the issue.

ii) **print (LF, CR)**

LF, which stands for line feed, is the process of moving the paper up to position the print head on the next line.
CR, which stands for carriage return, is the process of moving the motor of the carriage to reset the print head back to the beginning (left) of the paper.

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

Off-line operation in batch operating systems means that slow input and output are managed by Input Output Processors (IOPs) instead of the main CPU. This approach has the advantage of improved CPU throughput since the expensive main computer is not kept idle waiting for slow I/O to finish, but it has the drawback of needing additional hardware.

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?**

If a programmer forgot to check for the dollar sign in the cards, commands that start with it like $FORTRAN, $LOAD, and $RUN would simply be passed as normal input to the compiler which could cause errors and other unexpected results. To prevent this, batch operating systems protected the drivers under the kernel mode, so that programmers can not overwrite drivers with potentially erroneous design.

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?**

If an $END was in the middle of a program's execution, the OS would interpret it as a job control command, not as part of the program. It would treat it as the signal that the current job is finished and terminate the running program, and reset memory.

f) **Write examples of four privileged instructions and explain what they do and why they are privileged.**

An example of a privileged instruction would hardware commands such as advance_motor() or check_card_contacts(). These are both parts of the card reader driver. The check_card_reader() is where the device actually checks whether the contacts are touching through the holes in the punch cards or not, and inputting a 1 or 0 based on its result. The advance_motor() is the motor moving forward the head of the device to the next line of the punch card. These are privileged instructions because we do not want users to mess with hardware in the event that they make a mistake and break things not only for them, but for all other users of the machine. This generally applies to hardware I/O devices and is why drivers are stored in the kernel protected OS.

Another privilege instruction would be something like save() or delete() where we are editing a specific part of memory. We want to be very careful and consistent with these programs as a mistake or flippant use could have devastating effects when overwriting or removing parts of memory that may be in use. This is why they are privileged. By only allowing the OS to make these changes in kernel mode, we protect the system from the user incorrectly editing memory and resulting in errors.

The print text instruction is another example of a privileged instruction. It takes an input, and sends it to be printed. This instruction has to be privileged because if misused by software, it can trap the system in an infinite loop of printing lines of text and eat into other people's programs. This is because this instruction does not check if data is job language code, it will interpret it all as text to be printed.

Finally, modifying memory is a privileged instruction. Modifying memory changes data in memory. If a program had direct access to modify memory wherever they wanted, they could mess with other programs' code or the OS. This could cause the whole system to break, so memory modification is always controlled by the OS.

g) **A simple Batch OS includes the four components discussed in class, Interrupt Processing, Device drivers, Job Sequencing, and Control Language Interpreter.**

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.**
The Control Language Interpreter CLI reads job control cards from the card reader and parses $LOAD, $RUN, etc. Then it passes requests to the Job Sequencing routine for execution.
The Job Sequencer manages jobs in memory. It is the main control routine of the batch system and is responsible for managing the User Area in main memory. It also initiates the execution of user programs.
The Device Driver directly controls and communicates with hardware devices like tape drives, card readers, and printers. It receives I/O requests from the Job Sequencer and needs to manage data transfer between the device and main memory, then report back to the Job sequencer when the device is done or there is an error.
Interrupt Processing handles interrupts for I/O completion, device errors, or timer expirations. It detects when a hardware device raises an interrupt signal, saves the CPU's current state, and calls the correct ISR. After the routine it restores the CPU state and returns control to the original program before the interrupt.

ii) **A detailed explanation of the execution order and how the routines interact.**
CLI validates the #LOAD TAPE1 command and calls the Job sequencing/loader to load from TAPE1. Then the Job sequencer allocates the user area in memory, and asks the device driver to read the tape. The device driver sends commands to the tape hardware to locate the first file and begin data transfer. Interrupt Processing runs as tape blocks arrive and the hardware raises interrupt, invoking the driver's ISR which sends the block into memory. When the file is fully transferred the loader sets the job state to loaded or ready and returns control to the CLI. The CLI now reads $RUN and tells Job sequencing to start the job. The

job sequencer performs a context switch and transfers control to the user's program. During program execution, device drivers and interrupt processing handle any I/O the program performs. When the program ends, Interrupt processing notifies the job sequencer which returns control to the CLI.

iii) **A step-by-step breakdown of what each routine performs during its execution**
The $LOAD TAPE1: card is interpreted by the Control Language Interpreter which invokes the Job Sequencer/Loader. The Loader asks the Tape Device Driver to fetch the first file on TAPE1:. The driver starts the tape transfer and uses interrupts to report each completed block. The Interrupt Processing routine sends to the driver’s ISR which updates buffers and wakes the loader. When the file is fully transferred and relocated into the user area, the loader updates the job table and returns to the CLI. The CLI then sees $RUN, calls the Job sequencer to perform privileged setup and transfers control to the program. During the program's execution, device drivers and interrupts continue to handle I/O and termination events, and the Job Sequencer performs cleanup when the programs end.

h) **Consider the following program:**

```
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
}
```

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

Time (s)	1.3	0.5	1.8	0.5	1.8
Reader	Read card to x				Read card to x (2)
CPU	Loop for 1.3 seconds	name = find_student_last_name(x)	Loop for 1.8 seconds	GPA = find_student_marks_and_average(x)	Loop for 1.8 seconds
Printer			print (name, printer)		print(GPA, printer)

CPU Usage: 100% of the time
One Cycle: 5.9 seconds
Full Program Execution: 1307.7 seconds
Timed I/O results in a lot of wasted time as the CPU waits for a set amount of time to let the printer and card reader work. This time is set to be longer than the printer and card reader needs, resulting in wasted time. In this case, each time the reader or printer is used, the CPU wastes an extra 0.3 seconds where nothing is being done. Despite the CPU being used 100% of the time, most of that time it is not computing anything and is just looping to wait for the card reader and printer. The good thing is that while the last print is happening, the program can tell the reader to start reading the next card to x as it doesn't need the previous x.

ii) Polling

Time (s)	1	0.5	1.5	0.5	1.5
Reader	Read card to x				Read card to x (2)
CPU	Ask if reader is done	name = find_student_last_name(x)	Ask if printer is done	GPA = find_student_marks_and_average(x)	Ask if reader is done
Printer			print (name, printer)		print(GPA, printer)

CPU Usage: 100% of the time
One Cycle: 5 seconds
Full Program Execution: 1137 seconds
Polling removes wasted time by having the CPU constantly check if the I/O device has finished its job. This allows the CPU to start working as soon as the I/O device it is waiting for is finished. There is still a lot of wasted time however. This is because the CPU can still do nothing while the reader or printer are working.

iii) Interrupts

Time (s)	1	0.1	0.5	1.5	0.1	0.5	1.5	0.1
Reader	Read card to x	Interrupt					Read card to x (2)	
CPU	Sleep		name = find_student_last_name(x)	Sleep	interrupt	GPA = find_student_marks_and_average(x)	Sleep	interrupt
Printer				print (name, printer)			print(GPA, printer)	

CPU Usage: Only when processing program
One Cycle: 5.3 seconds
Full Program Execution: 1506.2 seconds
Interrupts on their own actually add to the amount of time it takes a program to execute. The benefit of it is that the CPU can sleep while the reader and printer are working. This causes the CPU to last longer which is beneficial as it is by far the most expensive part of the computer.

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

Time (s)	0.5	0.5	0.1	1.5	0.1	1.5	0.1
Reader	Read card to x (loop x)		Interrupt			Read card to x (loop x + 1)	
CPU	name = find_student_last_name(x) (loop x - 1)	GPA = find_student_marks_and_average(x) (loop x - 1)		Sleep	Interrupt	Sleep	Interrupt
Printer				print (name, printer) (loop x)		print(GPA, printer) (loop x)	

CPU Usage: Whenever there is data to be processed
One Cycle: 4.3 seconds
Full Program Execution: 938.2 seconds
Interrupts with buffering reduces execution time by a lot by allowing the CPU to process previously stored values while the printer and reader are working. This makes it so the reader, CPU, and printer can all be working at the same time. The only bottlenecks are the time it takes for the reader and printer to work on I/O bound processes and the time it takes for the CPU to process the data on CPU bound processes.

Part Two - Design and Implementation of an Interrupts Simulator

The objective of this section is to build a small simulator of an interrupt system, which could be used for performance analysis of different parts of the interrupt process. This simulator will also be used in Assignment 2 and 3.

All test executions can be found in the github in the executions folder. Test case executions are named execution_trace_x-y.txt where x denotes the variable configurations and y denotes the trace used.

Variable configurations:

1. save/restore context time 10ms, ISR activity time 40ms
2. save/restore context time 20ms, ISR activity time 40ms
3. save/restore context time 30ms, ISR activity time 40ms
4. save/restore context time 10ms, ISR activity time 100ms
5. save/restore context time 10ms, ISR activity time 200ms

Trace 1		
Save/Restore Context Time	ISR Activity Time	Total time
10	40	3802
20	40	3902
30	40	4002
10	100	4702
10	200	6202

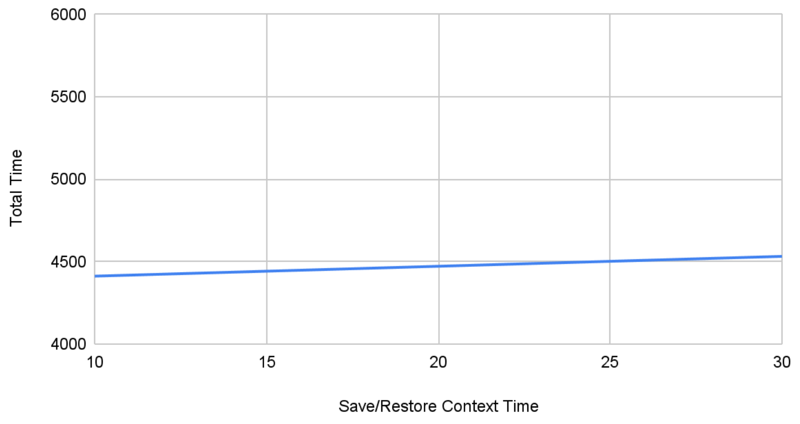
Trace 2		
context_save_time	ISR_time	Total ime
10	40	4412
20	40	4472
30	40	4532
10	100	4952
10	200	5852

Trace 3		
context_save_time	ISR_time	Total time
10	40	32090
20	40	32750
30	40	33410
10	100	38030
10	200	47930

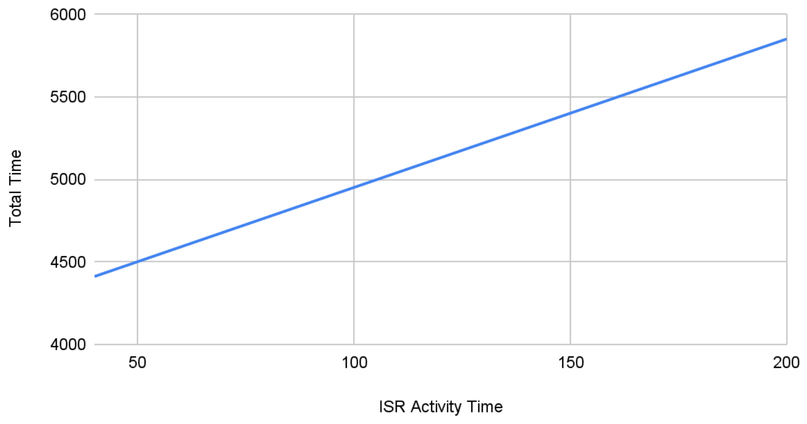
Trace 4		
context_save_time	ISR_time	Total time
10	40	37607
20	40	38287
30	40	38967
10	100	43727
10	200	53927

Trace 5		
context_save_time	ISR_time	Total time
10	40	32124
20	40	32844
30	40	33564
10	100	38604
10	200	49404

Total Time vs. Save/Restor Context Time



Total Time vs. ISR Activity Time



Based on the test cases, we can see that increasing the save/restore context time and the ISR activity time increases the total execution time on a linear scale. The slope of the scale is the same for both. ISR time tends to affect total execution time much more because it takes much longer and varies by much wider margins.

In the case of having addresses of 4 bytes instead of 2, the number of possible addresses would drastically increase. This has a side effect of every memory reference, interrupt vector fetch and I/O access having 4 bytes instead of just 2. This would increase the context save/restore time because more memory has to be read/written for each operation. By extension the total execution time will also increase slightly for every step that involves addresses.
If the CPU were faster, every CPU burst would take less time and thus a higher percentage of the computer's runtime would be spent on interrupts. This would make CPU bound processes run faster, but would have little to no effect on I/O bound processes.