SYSC 4001 Assignment 1

Justin Morrison (101302852) - Student 1

Random Hygaard (101273397) - Student 2

Github: https://github.com/Justin-Morrison-github/SYSC4001_A1

Part 1

A. Interrupt System (Justin)

An interrupt is a signal to the CPU that an event has occurred. Most of the time, an interrupt comes from hardware, but they can also be triggered by software. A specific register in the C
PU stores a 1 if an event has occurred, or 0 otherwise. After every fetch-execute cycle, the CPU checks this register to see if an interrupt has occurred. If an interrupt has occurred, the CPU will switch over to kernel mode so it has sufficient permissions. The CPU will halt (interrupt) the current execution, look up the address of the ISR (Interrupt Service Routine) in the vector table based on the specific device that raised the interrupt signal. The vector table is a block in main memory, usually at the very beginning, which contains the addresses to the start of each ISR. Once retrieved from the vector table, the address of the ISR will be placed into the PC (Program Counter). Before jumping to that ISR, the CPU will save the current context of the program on the stack and save the PC value into the LR (Link Register), so when the ISR is done it knows where to return/jump to. Now the CPU will continue the normal fetch-execute cycle using the ISR address stored in the PC and will execute the ISR. This is all handled by hardware.

Here is where the software takes over. The specific software of the ISR will determine exactly what happens during its execution. There will likely be some logic based on the type of device that caused the interrupt. An keyboard ISR might check which key caused the interrupt to know what letter to print, for example. The SW will likely reset the control register in the CPU that indicates that an interrupt has occurred. This tells the CPU that the interrupt has been dealt with, and on the next fetch-execute it will not see an interrupt unless of course another interrupt occurs. At the end of the ISR an IRET (Return From Interrupt) command is executed. This command pulls the previously saved values from the stack, and loads the LR (which contains the previous PC value) into the PC. The last thing the IRET command does is return to the user mode. On the next fetch-execute cycle the CPU will jump to the instruction directly after the one where the interrupt occurred. Now the ISR has been run, the interrupt control register has been

cleared, and the CPU will continue the normal fetch-execute cycle until the next interrupt happens.

B. System calls (Random)

System calls are how user programs interact and request actions from the OS. It is not the same as an interrupt. User programs are not granted access to many critical functions the OS uses. These privileged instructions can be crucial for user programs to utilize. If a user wishes to use these they must make a system call to the OS. System calls use the same hardware and method as interrupts in order to execute the command. The system call will send a signal to the CPU causing it to stop the current fetch execute cycle. Then it will process the system call. Since the privileged instructions must be in kernel mode first the system call will send a signal to the CPU to switch to kernel mode. Next it will save the context to the stack and jump to a predetermined ISR labeled by the system call which is the implementation of the system call. After the ISR is executed it continues to follow the interrupt process and change the system back to user mode before continuing execution of the user program. Some examples of system calls may be, creating a new process, opening and closing files, and allocating and freeing memory.

C.

a. (Random) When the driver calls check_if_printer_OK() the CPU will send a signal to the printer to check its various sensors and hardware components. This includes checking if the printer is on and connected, the printer has sufficient ink and paper, there are no jams or clogs which could jeopardize the print, if the printer is currently busy with another job, or if it has run out of space on the paper to print. Once the check is finished and no issues have been found the printer will send a signal back to the CPU, likely through an interrupt. This notifies the OS that the printer is available to use. If a problem is detected it will send a different signal back to the CPU, telling it what issue has been found in the printer. This will likely be relayed back to the user so they can physically check the printer and get it back in working order.

b. (Justin) When the printer needs to go to a new line, two commands are used: LF and CR. LF(“Line Feed”) will send a signal to the printer to rotate a motor to move the paper upwards so there is blank space to print. There will likely be some register in the printer device that will reflect when the motor is done. The output driver would wait until the LF is done by either polling the control register, or waiting for the printer to raise an interrupt. Once the LF is done, now the CR ("Carriage Return”) command is run and a signal is sent to the printer to return the print head to the left side of the page to start printing a new line from left to right. A motor must be activated to move the print head back. Same as before there will likely be a register in the printer that reflects when the head is done. Polling or interrupts could be used to determine when this has occurred, after

which the printer can return to normal printing operation as there is now new blank paper to print on.

D. A stack of cards will be placed into an auto-loader unit with a smaller cheaper CPU. This computer's purpose is to read the contents of the punch cards and copy them to a reel of magnetic tape. Magnetic tape has more data density which can be read faster. This reel will hold many jobs alongside their data to be continuously fed to the expensive fast CPU. After the CPU has executed the jobs and computed the data, the reel will be moved to a similar small, cheap CPU to have its contents read and printed out. With the ability to load many jobs onto a single reel of magnetic tape and executed back to back, the downtime of the expensive CPU is minimized. This is very beneficial to the owner of the machine, who wants to get as much value as possible from the expensive CPU. The downsides are that since jobs are put into a batch, for the individual programmer and user it takes longer to have their program be run through the CPU. Another downside is that a human operator still needs to load cards and move reels between the three machines, losing some processing time.

E.

a. If the driver does not properly parse the "$" or any other identifiers it will cause the system to continuously read the instructions as plain data. This will for example cause the card reader to interpret the $END card as data, and will not end the job. Then it will call to read the next card, which will be the $JOB card of the next job, but it will continue reading it as data. This will cause the output of the original program to be useless and the other program was never run properly. The reader will continue to read until there are no more cards to read, skipping the entire stack of jobs and wasting CPU resources. A solution to this problem is to have some sort of JCL grammar checker program to ensure that the driver follows the correct JCL commands to ensure proper execution.
b. If, in the middle of execution, a card was read in with "$END" text, the OS would terminate the program execution since it would think it is the same as the $END JCL command. Then it would read cards until the next $JOB card, which signals the start of the next job. The OS would then continue operation by running this newly read job.

F. One example of a privileged instruction is halt. This instruction stops the processor for executing until an interrupt is received. This is an important instruction that must be privileged so that any accidental or malicious actions cannot be made to stop the CPU from executing. If the halt command could be used by any program it could drastically slow execution times from misuse.

Another privileged instruction is MOV CR. This is loading and storing control registers. MOV itself is not privileged if not accessing certain registers. Control registers are important registers that can modify CPU behaviour. This includes things such as the status register and enabling and disabling registers. If this instruction is not privileged,

any software can change these and modify the CPU behaviour. This means possibly overwriting important flags or stored addresses, leading to critical errors and losing data.

Switching the OS's operation mode (user vs kernel) is an example of a privileged instruction. This is the instruction that switches the bit(s) in a register in the CPU which determines whether the process running is on behalf of the OS or the user. This is a privileged instruction because the user should not be able to change what operation mode the OS is in as it may allow the user to execute malicious instructions. The OS should be the only entity able to change the mode.

Any I/O operation is a privileged instruction. These instructions interface with I/O peripheral devices to read or write data. The reason they are privileged is that a malicious user program could completely mess up other programs if it had direct access to an I/O device. For example, if a user program overwrote data in the hard-drive that was being used by another program, it could cause the other program to error or produce unknown/useless results. To prevent this, I/O access is only allowed through system calls, which switches the mode to kernel mode, and allows the OS to control what I/O is being done.

G. Batch OS

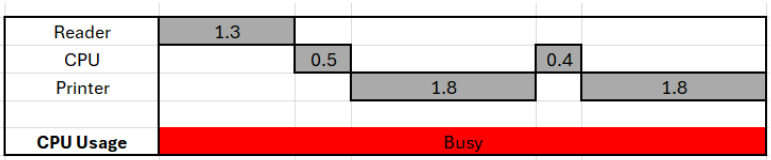
When the $LOAD: TAPE1 card is read the control language interpreter will detect the $ and parse that the instruction is a system call. It will then be executed accordingly and it will send a system call to the CPU. This is because instructions $LOAD requires privileged instructions. The TAPE1 argument points to the memory location where TAPE1 is stored on physical memory, in this case on tape or the input spool. When a system call first happens the OS is still operating in user mode. The first thing to happen is the system changes to kernel mode. Then the system call will have a specific number which the CPU will reference to find the corresponding device driver for the reader in the OS. This driver is an unchanging method which holds the implementation of the system call. In this case it will run the loader driver stored in the device drivers portion of the OS which loads the program pointed to by the argument into the main memory. After the program is fully loaded the driver finishes execution and in order to pass control back to the user process, reverts the OS to user mode. Then the PC will jump back to the instruction after the system call, the context will be popped off the stack and switched back then execution is continued normally. The next instruction is the $RUN command. Once again the control language interpreter will parse the $ and identify the command as a system call. This goes through the same process as $LOAD and begins running the program that was just loaded. This routine will set the PC to the first instruction of the program that $RUN is attempting to run. The CPU will then continue the fetch execute cycle until the end of the program, denoted here with $END. This signals to the job sequencer that the current job has finished and should expect the next one.

H. Timing Diagram

i.) Timed I/O

In Timed I/O the CPU waits a set amount of time for I/O tasks to finish. A error range is needed to ensure that events occur on time, which makes each I/O operation take more time. During this time the CPU cannot do anything else because it is waiting until the time has passed, likely in a while loop.

Main Program with Timed I/O	Time (s)	Error	Time w Error
Loop 284 times:			
x = read_card();	1	30%	1.3
name = find_student_Last_Name (x);	0.5		0.5
print(name, printer)	1.5	20%	1.8
GPA = find_student_marks_and_average(x);	0.4		0.4
print(GPA, printer);	1.5	20%	1.8

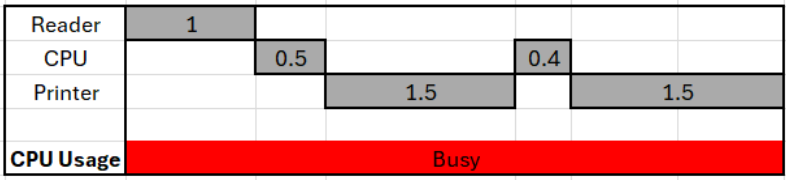


Time for 1 cycle	5.8 s
Time for program	1647.2 s
Time CPU Busy	5.8 s

ii.) Polling

Polling is definitely an improvement over Timed I/O because now the I/O devices themselves tell the CPU when they are done by setting a value in the polling register. This allows tasks to not need any extra buffer of time to wait for execution and overall execution time is improved. However, similar to Timed I/O, the CPU is busy and can't do anything else because it has to keep asking the I/O device "Are you done?".

Main Program with Polling	Time (s)
Loop 284 times:	
x = read_card();	1
name = find_student_Last_Name (x);	0.5
print(name, printer)	1.5
GPA = find_student_marks_and_average(x);	0.4
print(GPA, printer);	1.5



Time for 1 c	4.9 s
Time for prc	1391.6 s
Time CPU B	4.9 s

iii.) Interrupts

Interrupts allow the CPU to be free while the I/O devices perform their slow tasks. However, there is a time overhead with the interrupt process. Having to go fetch and execute the ISR takes time, and in this example, interrupts are actually slower than polling. But in an application where multiple tasks could be done by the CPU while waiting for I/O this would be much more efficient.

Main Program with Interupts	Time (s)
Loop 284 times:	
x = read_card();	1
name = find_student_Last_Name (x);	0.5
print(name, printer)	1.5
GPA = find_student_marks_and_average(x);	0.4
print(GPA, printer);	1.5
Interupt	0.1

Reader	1										
CPU		0.1	0.5				0.1	0.4			0.1
Printer				1.5					1.5		
CPU Usage	Free	Busy	Free			Busy	Free			Busy	

Time for 1 cycle	5.1	s
Time for program	1448.4	s
Time CPU Busy	1.2	s

iv.) Interrupts and Buffering

Interrupts alone are already CPU efficient, but introducing buffers allows CPU tasks to be done at the same time as I/O tasks. Of the examples seen, this has the lowest time of execution, and the CPU is mostly free to do other tasks if necessary. There still is overhead with ISR execution, but this architecture allows for much greater gains than the others.

Main Program with Interupts and Buffers	Time (s)
Loop 284 times:	
x = read_card();	1
name = find_student_Last_Name (x);	0.5
print(name, printer)	1.5
GPA = find_student_marks_and_average(x);	0.4
print(GPA, printer);	1.5
Interupt	0.1

Reader	1										
CPU		0.1	0.5	0.4			0.1				0.1
Printer					1.5				1.5		
CPU Usage	Free	Busy	Free	Busy	Free		Busy	Free			Busy

Time for 1 cycle	4.8	s
Time for program	1363.2	s
Time CPU Busy	0.7	s

Task 2 Report

For our simulation tests we changed various parameters in the code and ran the same 7 trace files for each case. Then we made a python script to figure out the average percent of CPU usage, IO, and overhead, and also compare the total run time of the programs. All of the execution and trace files can be found in Github. We chose to analyze the average to simplify things.

The default program contains the following parameter values:

- Data Transfer Rate = 40
- Context Switch Time = 10
- ISR Execute Time = 40
- IRET Return Time = 1
- VECTOR_SIZE = 2

Parameter	Value	Avg CPU %	Avg IO %	Avg Overhead %	Total Execution Time (ms)
Default	N/A	12.1996	82.191	5.6094	9423
DATA_TRANSFER_RATE	100	11.4302	83.3142	5.2556	10057
	200	10.3431	84.9012	4.7557	11114
CONTEXT_SAVE_TIME	20	11.9011	78.476	9.6229	9726
	30	11.413	75.2573	13.3297	10142
ISR_EXECUTE_TIME	100	10.7521	84.3041	4.9438	10691
	200	8.9769	86.8955	4.1276	12805
IRET_RETURN_TIME	10	11.9582	80.5641	7.4777	9613
	30	11.4543	77.1697	11.376	10036
VECTOR_SIZE	4	12.1996	82.191	5.6094	9423

First, the DATA_TRANSFER_RATE determines an internal ISR activity time to simulate differing complexities of ISRs. This operation occurs at every I/O SYSCALL that requires data to be read from or written to the I/O device. Since the reading or writing occurs within the ISR, increasing this parameter increases the length of each SYSCALL ISR execution and increases the IO percent.

Next, looking at the results of changing CONTEXT_SAVE_TIME to be longer. This operation happens twice every time a context switch is needed, which in our example is the SYSCALL

and END_IO ISRs. So it is very costly to increase its time. This was considered part of the overhead, so predictably a longer CONTEXT_SAVE_TIME leads to much more overhead time and a longer program. This means that the time the CPU and IO were using were a smaller portion of the program.

ISR_EXECUTE_TIME is used to simulate the length of time it takes for the ISR to finish its task, such as reading information from an external device. Increasing ISR_EXECUTE_TIME had very similar effects to increasing the DATA_TRANSFER_RATE. Each time either a SYSCALL or an ENDIO command is read it must execute the corresponding ISR, if this ISR takes longer then so will the whole program. The effects of increasing this over DATA_TRANSFER_RATE were greater. This is because the DATA_TRANSFER_RATE only matters when a SYSCALL is executed, while the ISR must also be executed each ENDIO command. With the same delay between the two by increasing ISR_EXECUTE_TIME the CPU usage decreases alongside the overhead, while the share of time IO takes increases.

IRET_RETURN_TIME simulates the Interrupt Return instruction which occurs at the end of each interrupt to return to the next instruction before the interrupt occurred. In our implementation, we split the restore context and user switch into separate things, so it doesn't make a lot of sense why IRET would vary in time, but it is an interesting test case. Since it is such a low duration event, making it even 10 times longer barely had an effect on the total run time, but it did increase the overhead percentage.

The value of VECTOR_SIZE has no impact on the performance and usage of the CPU, however it did change the execution.txt file. This is because this constant declares what the size of each instruction should be. This means by doubling the vector size the expected address of the first line of the ISR is exactly twice the default. This is an important constant to have correct so that the interrupt hardware can always find the correct instruction. This would have a memory cost as now each vector is twice as large, so you would need twice as much memory to store the same vector table.

All tests were done with values greater than the default parameter values, but the trends we discussed should follow if values less than the default ones were used. For example the CPU% increasing and IO% decreasing if DATA_TRANSFER_RATE was lowered, or memory addresses being halved if VECTOR_SIZE is halved.