Oct 5, 2025
Group 36
Riley Foxton - 101304022

SYSC 4001 Assignment 1

Part i)

a)

To start an interrupt an action happens in an input/output device, this could be a signal indicating readiness from the printer or a button being pushed. This sends a signal to the CPU, where it's stored in a register. If the CPU has enabled interrupts then the sequence continues, otherwise the sequence stops. The next step would be the CPU saving the program context, this means putting all the registers into the stack and incrementing the stack pointer. Then the CPU would look in the vector table to find the address of the ISR (interrupt service routine) for the activated interrupt (if there are multiple interrupts activated at the same time then the one with the highest priority would be run first). The starting address of the ISR would be put in the PC (program counter) and the ISR would be executed. After the ISR is done executing the flag that signaled the interrupt would be disabled and the program context would be retrieved from the stack and restored in the PC so that it was the same as before the interrupt. The hardware is responsible for all of this except restoration of the program and the disabling of the flag, which is software. If one interrupt is raised in the middle of running a different ISR then the first ISR may be interrupted only if the new interrupt is of a higher priority.

b)

A system call is a set of useful functions offered by the operating system for programs to use. This maintains a layer of protection between non OS programs and instructions involving hardware. There are lots of system calls and they can be responsible for file management, information maintenance, protection and more. System calls can be accessed through an API (application programming interface). Some examples of system calls are making a file, reading a file, and closing a file.

System calls operate using the same process as interrupts. Once a system call is made the program looks in a vector table for the starting address of that system call’s ISR, then it does a context switch from the main program and runs the ISR. Lastly it restores the context to before it was run and switches the program back from kernel mode into user mode and lets the user program continue running.

c)

i.

When the printer receives checkOK from the CPU it evaluates everything to do with the printer, the ink levels, if the paper is jammed, if there is paper underneath the printer head, if the motors are ok, ect. If everything that needs to be functional for printing is functional, then the printer sends a signal back to the CPU to indicate that it is ok and can continue printing, if not then it sends a signal saying something is wrong and printing cannot continue.

ii.
When the printer receives print(LF, CR) from the CPU it moves the paper down to a blank section and resets the printer head back to the beginning of the page. LF means line finish and indicates the page should be moved, the printer physically moves the paper so it does not print over the previous lines. CR , means carriage return and shifts the printer head back to the start of the page (the left hand side) so that the head does not hit the edge of the printer and print off the page.

d)
The off-line operations in batch OS work by cards that have holes punched through them (to indicate that the character where that hole is was used) being grouped together to make a program. Multiple programs would be put together by an operator and fed to a card reader, who would read the cards using contacts (touching contacts means there was a hole in that spot). Then the CPU would run the program on the cards (there were different methods throughout the years of reading cards directly, transferring the contents to tapes, and using a hard drive). Then the results would be printed by a printer and given to the programmer. Some benefits of this system was that a computer had more throughput and the programmer no longer had to write programs like the I/O drivers. Some drawbacks were that it took longer for an individual person's program to run and get feedback from, and the programmer was more restricted as they couldn't access the I/O devices directly (because of protected instructions).

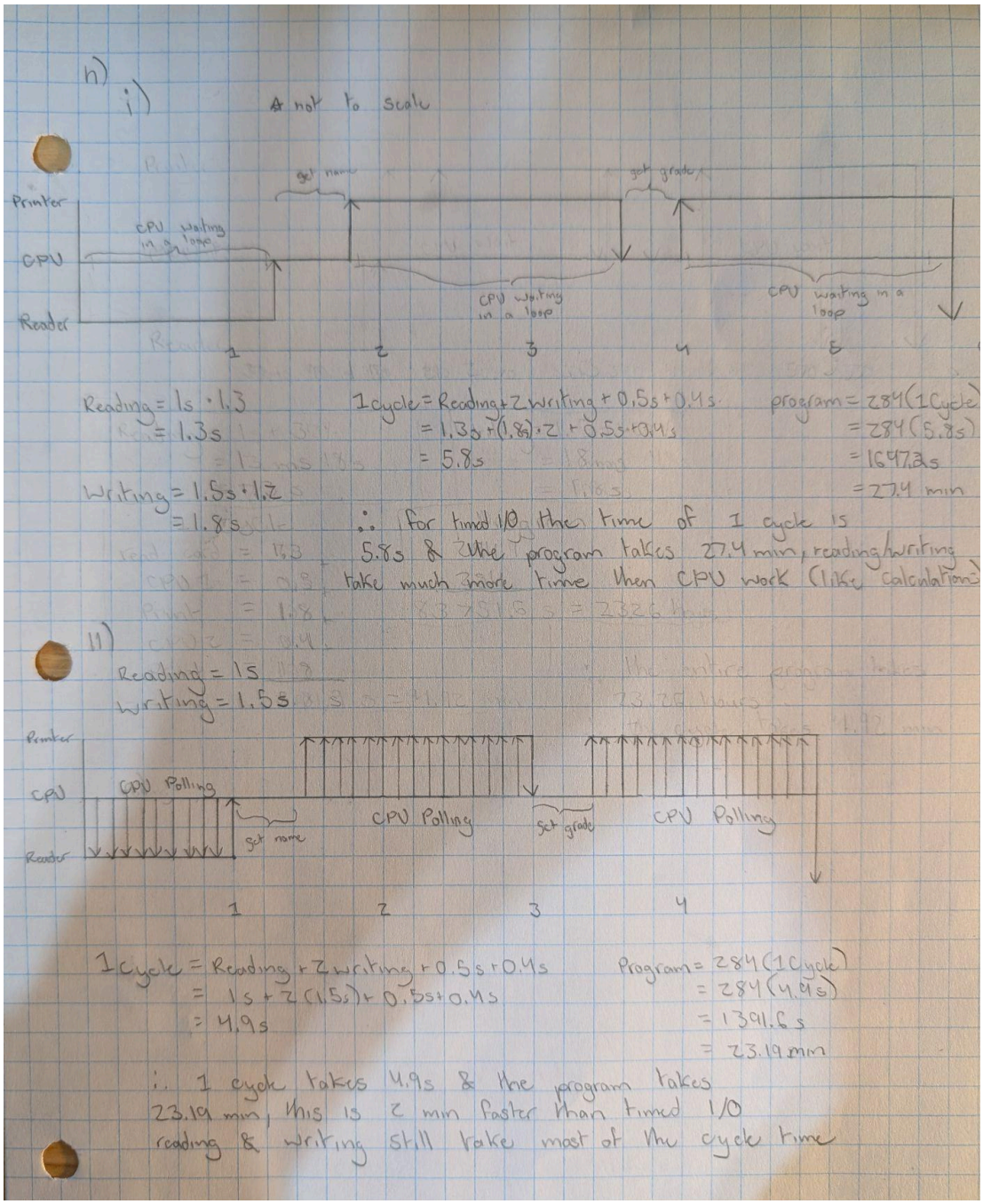
e)
i.
If the programmer forgot to account for the $ symbol in the driver then the card reader would keep running one program forever as it would interpret the $END as data for that program (since both data and $END are just holes in a card). The solution to this is not letting the programmer make the drivers for the I/O and having privileged instructions

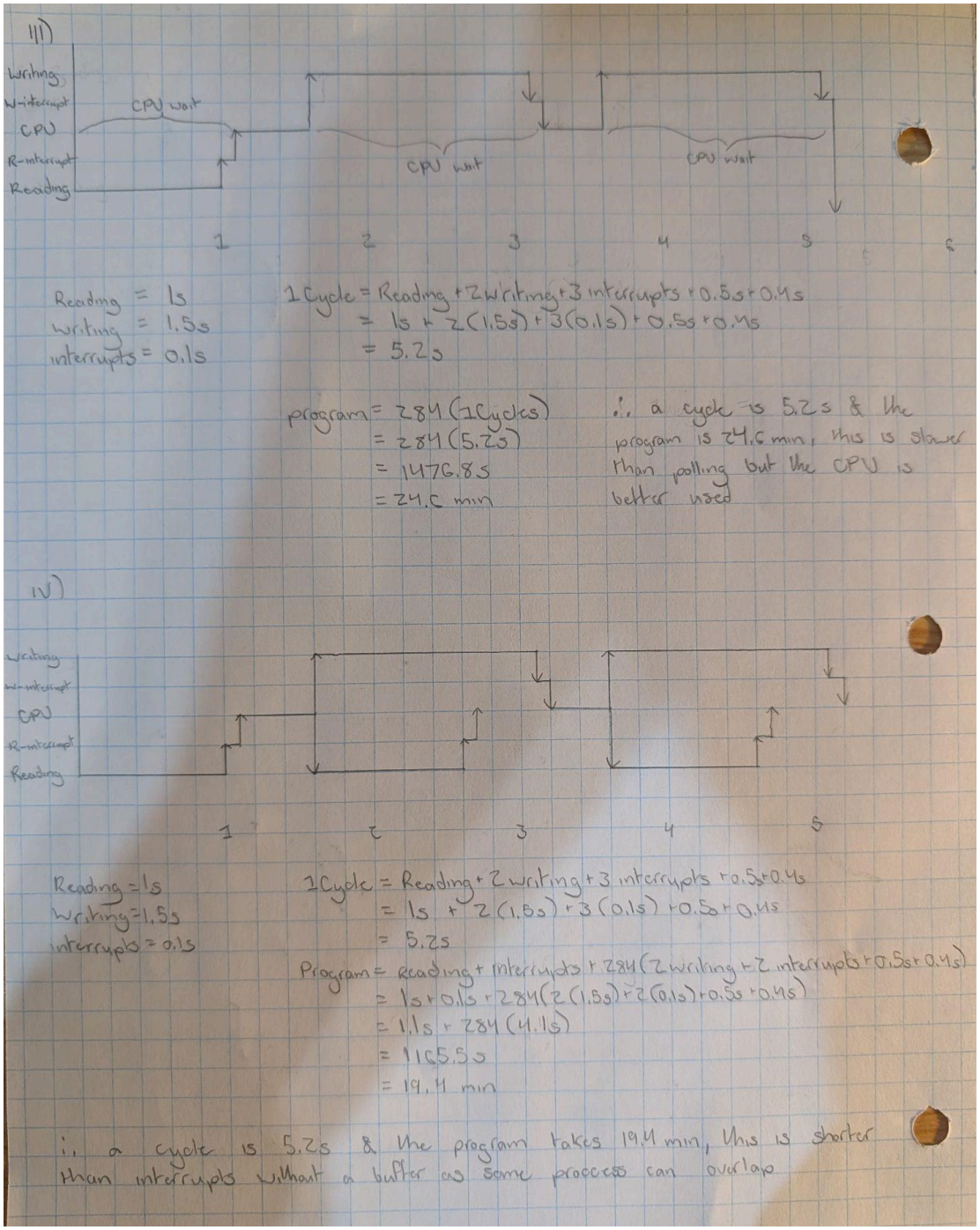
ii.
If the operating system reads $END unexpectedly then it should stop processing, continue to signal to the reader to read and look for the next job. It would know when the next job had started by the $JOB instruction, after it found the next job the OS should continue as normal.

f)
4 examples of privileged instructions are check_if_printer_ok, advance_motor, check_contacts and load_next_card. check_if_printer_ok evaluates multiple things associated with the printer (such as ink levels and paper jams) to see if it is fit for printing. The advance_motor function moves the motor further to the right of the page to print out text in lines. Check_contacts checks the contacts on a reader and is how the reader determines the symbols punched out (the contacts touching means there is a hole where the contacts are). Load_next_card loads the next card into the reader. These instructions are privileged because the user program may misuse these functions in a way that sabotages user programs or wastes resources (either accidentally or deliberately). For example a program may forget to move the motor, preventing printing in lines, or over advance the paper through a printer, wasting material.

g)
First the $LOAD card will be read by the card reader and the control language interpreter will recognize that this is an instruction and not part of the regular user program. Then the loader uses the I/O port to get the one line of executable code from the hard drive at a time and move it to main memory. It does this until the user program is completely in the main memory. Then once the $RUN card is read the PC is set to the start of the user program and exits from kernel mode.

h)





The Gantt Diagrams are useful for visualizing that even though interrupts take more time than polling, interrupts save lots of strain on the CPU. Both options take less time than timed I/O.

Part ii)

The results of the simulation and testing can be found in Table 1.

Test File	Context Time	CPU speed	Program time (ms)	Program time (min)
Trace 5	10 ms	double speed	25824	43.04
		regular speed	27795	46.33
		half speed	31701	52.84
	20 ms	double speed	26544	44.24
		regular speed	28515	47.53
		half speed	32421	54.04
	30 ms	double speed	27264	45.44
		regular speed	29235	48.73
		half speed	33141	55.24
Trace 1	10 ms	double speed	2849	4.75
		regular speed	3200	5.33
		half speed	3898	6.50
	20 ms	double speed	2949	4.92
		regular speed	3300	5.50
		half speed	3998	6.66
	30 ms	double speed	3049	5.08
		regular speed	3400	5.67
		half speed	4098	6.83
Custom Test Case	10 ms	double speed	1572	2.62
		regular speed	1680	2.80
		half speed	1896	3.16
	20 ms	double speed	1612	2.69
		regular speed	1720	2.87
		half speed	1936	3.23
	30 ms	double speed	1652	2.75
		regular speed	1760	2.93
		half speed	1976	3.29

Table 1: A summary of the length of the program for test files, context time refers to the context save/restore time, CPU speed refers to how the time of the CPU was adjusted, program time is the overall time of the program.

The two variables that were experimented with were the context save/restore time and the CPU time. The three files used to test this were trace 5 (which was given) because it was a long program, trace 1 (which was given) because it was a medium sized program and a custom test case. The custom test case was designed to be the average of the program time for a single cycle. The CPU times were averaged from every CPU time in trace 5 and the context times were averaged from the times in the device table.

To find the effect of context speed times on the overall program time we can find the differences in the increase of the program time, shown in Table 2.

Test File	Context Speed Difference	Program time increase for CPU Regular speed	Program time increase for CPU Double speed	Program time increase for CPU Half speed
Trace 5	20 ms - 10 ms	1.20 min	1.20 min	1.20 min
	30 ms - 10 ms	1.20 min	1.20 min	1.20 min
Trace 1	20 ms - 10 ms	0.17 min	0.17 min	0.17 min
	30 ms - 20 ms	0.17 min	0.17 min	0.17 min
Custom Test Case	20 ms - 10 ms	0.07 min	0.07 min	0.07 min
	30 ms -20 ms	0.07 min	0.07 min	0.07 min

Table 2: Differences in program speeds caused by context times, i.e. in trace 5 for the increase of 10 to 20 ms in the context time the program time increased by 1.2 min for each CPU speed.

Table 2 demonstrates that changing the context save time has small increases on the overall program time. To find how the CPU speed changes affect the program we can look in Table 3, which shows the differences in program time when decreasing the CPU speed.

Test File	CPU Speed Difference	Program time increase for 10 ms Context time	Program time increase for 20 ms Context time	Program time increase for 30 ms Context time
Trace 5	Regular - Double	3.29 min	3.29 min	3.29 min
	Half - Regular	6.51 min	6.51 min	6.51 min
Trace 1	Regular - Double	0.59 min	0.59 min	0.59 min
	Half - Regular	1.16 min	1.16 min	1.16 min
Custom Test Case	Regular - Double	0.18 min	0.18 min	0.18 min
	Half - Regular	0.36 min	0.36 min	0.36 min

Table 3: Differences in program speeds caused by CPU times i.e. in trace 1 for the decrease of double speed to regular speed the program increased by 0.59 min for each context time.

Table 3 demonstrates much bigger time changes than the changes in Table 2. Therefore it can be concluded that the CPU speed has a greater impact on the run time of this program than the context time, this is to be expected as the context time makes up a much smaller part of the program.

If another test program was given then the time it would take for that program to run can be estimated by using the custom test case. This would be accomplished by multiplying the number of cycles in the new program by the time it takes for one cycle of the custom case at a context time of 30 ms and a halved CPU speed. This is because that is the time that one cycle of the average program would be the slowest and it is unlikely the new program would be slower than that.

GitHub link https://github.com/rileyfoxton/SYSC4001_A1