SYSC 4001 Operating Systems Fall 2025

Assignment 1

SYSC4001 L3 – Group 3

Gabriel Wainer

Muhammad Ali – 101291890

Gregory Horvat – 101303925

https://github.com/aliooo36/SYSC4001_A1

Part I – Concepts

I.a)

Hardware:

Device drivers are loaded into the I/O controller to translate hardware signals before they are sent to the central processing unit (CPU) through the interrupt-request line. The CPU checks this line after every instruction execution cycle. When an interrupt is detected, the CPU pauses the current execution and saves the current state in a link register or the stack. The interrupt controller provides an interrupt number, which is used as an index into the interrupt vector table to fetch the memory address of the corresponding interrupt service routine (ISR). This memory address is then loaded into the program counter (PC), and control is transferred to the software.

Software:

The ISR, written either within the operating system (OS) or a device driver, identifies the interrupt source and performs the necessary processing. Once processing is completed, the CPU restores the prior state and executes a “return-from-interrupt” instruction to return the CPU to the state before the interrupt occurred.

Overall, the device controller raises an interrupt signal via the interrupt-request line, which is then caught by the CPU and dispatched to the interrupt handler to then be cleared by the handler once execution of the ISR is done.

I.b)

A System call is an entry point that allows the user’s program to request an operation or service from the kernel. They act as the interface between the two. Examples of known system calls include read(), write(), open(), close(), etc.

System calls are related to interrupt as they are software-generated interrupts and are implemented using the interrupt mechanism. When a program issues a system call, the CPU is interrupted and transfers control to a fixed memory location with the starting address of the system call routine. An interrupt vector table is used to find the correct routine, while the hardware saves the address of the interrupted instruction. The routine then executes in kernel mode and then the state is restored, finally returning to the interrupted user program.

I.c)

i)

When check_if_printer_OK() is called, the printer goes through many checks. It first verifies it is on and responding. It then checks that the feed card is loaded and aligned with the print line, including checking for jams and misfeeds. Then, before each character, it verifies the device is not busy, that the counter is in sync with the punch card, and that no errors have occurred, such as misalignments.

ii)

For print (LF, CR):

After the normal execution of printing the 80-character long array loop is done, the printer is instructed to “Line Feed” (Move the paper up by one line) and “Carriage Return” (Move the print head to the beginning of the line). This step is entirely mechanical; the printer must ensure that the print head has been moved correctly to the start of the first line and that the paper has been moved down one line. The printer must also check that the printer is not actively processing another instruction before moving the paper and print head.

I.d)

The off-line operation in a Batch Operating System (Batch OS) involves using mechanical input/output devices that prepare and feed programs and data into the main computer system for later processing. Common examples include punch cards and magnetic tapes, which are used to store job data. These jobs are then executed sequentially in large batches by the computer and continue running until completion or manual interruption by an operator. The outputs are usually produced on off-line devices such as printers.

As discussed in class, Weaker I/O CPUs (1401) found within the punch card reader and the printer are significantly cheaper to produce and run than the main computer system which houses a much more powerful CPU (7094).

Advantages:

- Efficient CPU Utilization: Input and Output operations are handled without the CPU; this allows for focus to be placed entirely on computation of programs.

- Reduced Idle Time: Since programs are continuously being computed, The CPU is rarely idle. This reduces system downtime and maximizes throughput.

Disadvantages:

- No User Interaction: Operators/Programmers cannot directly interface with the system as it is computing, if there happen to be errors in a program the programmer must wait until their

program is executed in the large batch of other programs. This causes large turn around times in terms of debugging and error handling of programs.

- Handling & Storage: Given the input and output are handled in Punch Cards, Magnetic Tape and printed outputs, storage and handling of these items lead to additional overhead and presents potential issues in terms of damage and misplacement.

I.e)

i)

If a programmer wrote a driver card and forgot to parse the $ character in the cards read, the CPU would perform the specialized command regardless of if it was intentional or not. Keywords for commands such as FORTRAN, LOAD and RUN present throughout the card would cause the program to compile, start or stop without the intention of the programmer.

For example, as discussed in class, a charitable run club including the word “RUN” in their punch card could unintentionally cause the system to attempt to start program execution in the middle of their data.

We prevent this error by requiring the programmer to explicitly request system calls to execute these commands. The OS should check for the $ prefix character to ensure the programmer is indeed requesting a given command.

ii)

If a card has the text “$END” at the beginning of their card, the program should be terminated by the OS even if it is not fully finished executing. The OS would then move onto the next job.

I.f)

1. Instruction to switch to kernel mode

It changes the CPU from user mode to kernel mode, giving the program access to protected resources. It is privileged as by allowing a user program to switch to kernel mode, you allow it to bypass all OS protection, which can lead to harming the system.

2. I/O Control Instructions

It performs operations to communicate with hardware devices, for example sending data to a printer. It is privileged as the access you gain directly to the hardware can interfere with the OS. They should only be safely executed by the kernel

3. Timer management:

The CPU writes values straight to the timer device to start and stop it when needed. The OS timer is crucial when it comes to ensuring correct operation of the device, scheduling, preventing programs from running too long, sleep/timeouts, and counting program execution time.

Timer management is a privileged instruction, since user interaction and interference can potentially break or stop correct scheduling for critical OS tasks.

4. Interrupt management:

The CPU can enable or disable maskable hardware interrupts and utilizes a vector interrupt table to map interrupt numbers to their correct ISRs. This allows for the correct execution of interrupts stemming from hardware signals and handled in software via drivers.

Interrupt management is a privileged operation since disabling or manually changing the vector table could prevent the OS from scheduling or executing device ISRs, potentially locking the OS or rendering hardware devices unresponsive.

I.g)

When the system reads the control cards given, the resident monitor (the control program for the Batch OS) manages a sequence of operation to load and execute a program from the tape device.

Steps:

1. Reading $LOAD TAPE1:

The Control Language Interpreter (CLI) reads the card from the input device and recognizes the $LOAD command. It then passes the request to the job sequencing component. Basically, the CLI interprets the control statements and forwards the valid commands.

2. Initiating the load operation:

The next job is to load the program from the tape device. The job sequencing layer, after receiving the command, calls the device drivers to perform the I/O operations required for loading the device. The layer controls the order in which the jobs are executed, ensuring $LOAD runs before $RUN.

3. Transfer data from the tape:

The tape driver, within the device drivers, issues commands to the tape hardware to position and begin reading the file into memory. The interrupt processing mechanism becomes active, as after each tape finishes transferring, an interrupt signals the monitor that the I/O has completed, and more data can be loaded.

4. Complete the load

After tape is read, control returns to the CLI. The monitor finishes the loading step by placing the program into the user area, where any address relocation is done, and the job is now marked as ready to run.

5. Reading $RUN

The next card $RUN is now interrupted by the CLI, which passes it to the Job Sequencing routine. The routine confirms that a program has already been loaded, and the execution can begin.

6. Start the program

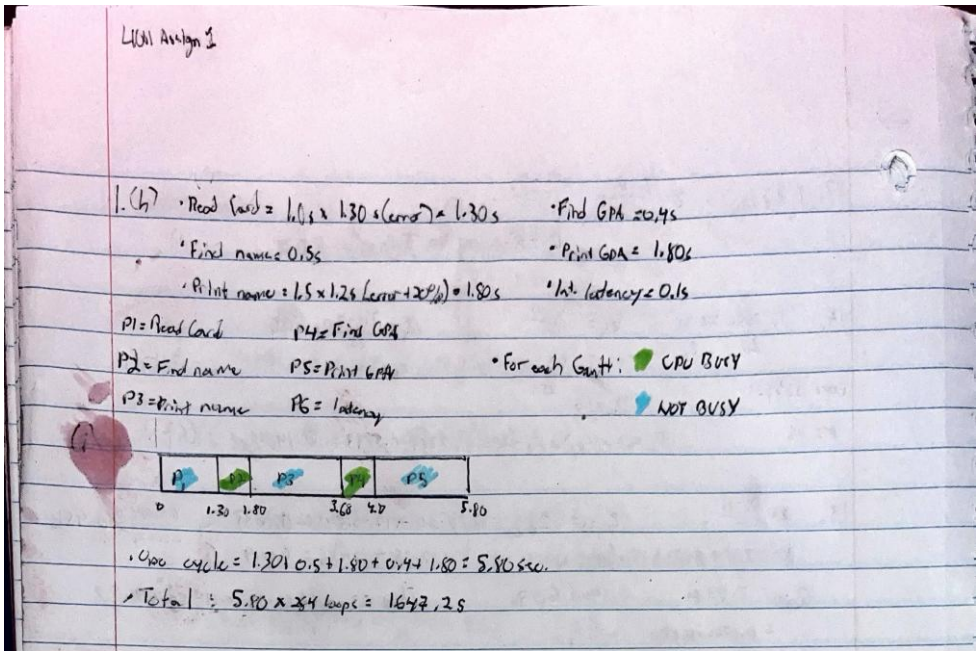
The device drivers set the program counter to the loaded program's starting address and changes the CPU from monitor mode to user mode. The monitor's interrupt processing remains active to handle any I/O requests or exceptions that occur while the program runs.

7. Execution and monitoring

The program now executes in the user area. Whenever I/O is performed, or the execution finishes, control returns to the monitor, allowing the Job Sequencing routine to continue with the next job, if any, in the deck.

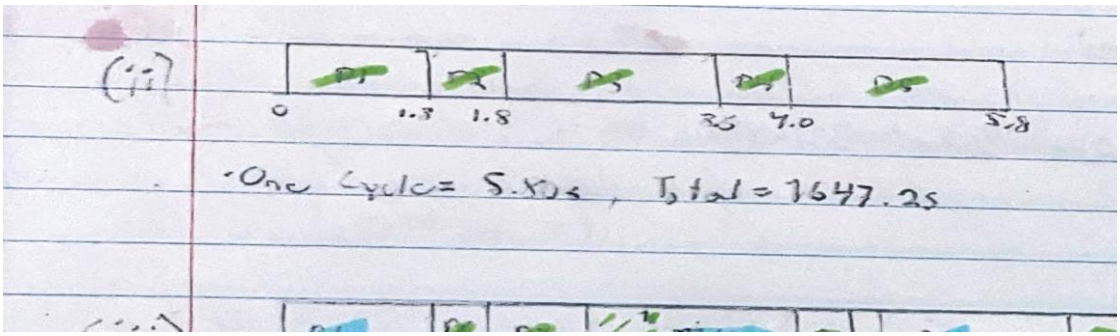
I.h)

(i) **Timed I/O:**



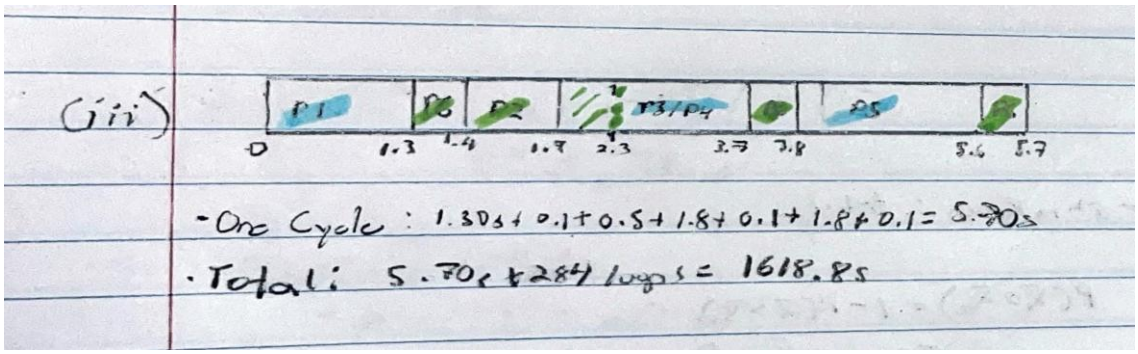
The CPU waits idly for every input or output operation to finish before proceeding. The CPU is only active for the two Find Name and Find GPA processes, for a total of 0.9 seconds. This means the CPU is idle for the rest, therefore poor CPU utilization.

(ii) Polling:



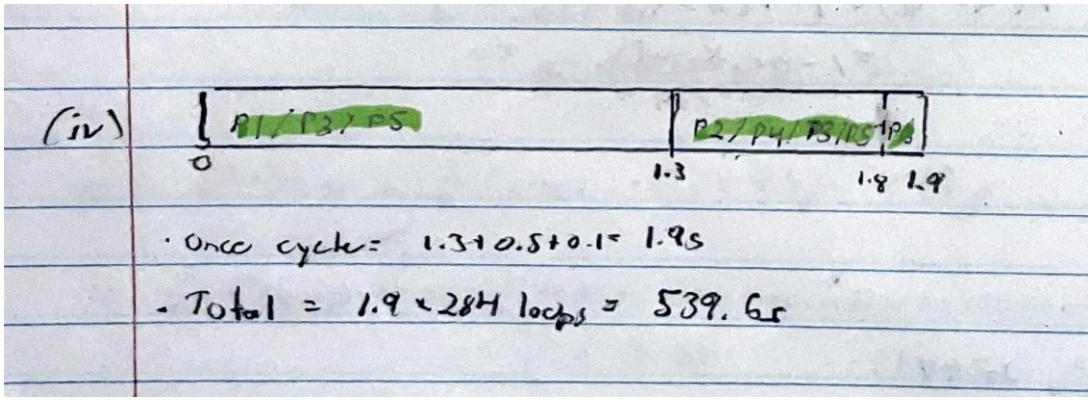
The CPU polls each device's status continuously, meaning the CPU is constantly busy. It is the same sequence and timing as the Timed I/O, so the calculations remain the same. CPU utilization reaches 100%. But efficiency is still poor as many useless polls occur during each cycle.

(iii) Interrupts:



CPU is idle during I/O, active during P2 and interrupts, and extends activity partially into the processes after P2. CPU does computation simultaneously while the first print runs but may still wait for the printer before the print GPA process. With Interrupts occurring three times, and the overlap of P3 and P4, the overall time of one cycle was 5.70s, with the total execution time for all loops totalling 1618.8s.

(iv) Interrupts + Buffering:



The addition of one input buffer and one output buffer allow for pipelining. This means much more overlap can occur in the reading, computation and printing. While the printer outputs one record, the CPU computes the next, and the input reads the next card, eliminating most idle/wait times. This also means constant work from the CPU and is the most efficient CPU utilization out of all four scenarios. The resulting time for one cycle is therefore 1.9s, with the total time for all loops being 539.6s.

Analysis of Simulation Test Cases:

This report analyzes 22 interrupt simulation test cases examining various aspects in processing performance. The tests reveal insight into context switching overhead, ISR timing, CPU scaling effects, and workload characteristics that influence overall system performance.

1. Context Switching Tests (3 tests)

Purpose: Analyze impact of context save/restore time variations

- test_context_10ms.txt (Baseline)
  This test establishes the baseline performance for efficient context switching with a total overhead of 13ms, demonstrating optimal interrupt handling. This result indicates that the 10ms save/restore context time represents a well-made, standard operating system configuration.

- test_context_20ms.txt
  Switching the context time variable to 20ms, we can see a 77% increase in interrupt overhead, totalling 23ms. This confirms the linear relationship between context switching time and interrupt latency. This shows that an increase in switching time directly contributes increased system latency, potentially affecting real-time responsiveness and throughput.

- test_context_30ms.txt
  This case revealed a substantial 154% increase in interrupt overhead, reaching 33ms, this result further confirms our findings in the previous tests, and underscores context switching optimization as high priority for performance tuning.

Overall Analysis: Context switching time has linear impact on interrupt overhead. Each 10ms increase in context time adds exactly 10ms to total interrupt latency.

2. ISR Timing Tests (4 tests)

Purpose: Analyze impact of ISR execution time variations

- test_isr_40ms.txt (Baseline)
  Establishes baseline for standard ISR execution with predictable 40ms processing time as stated in the assignment.

- test_isr_80ms.txt
  This test shows that doubling the ISR execution time directly affects SYSCALL duration while END_IO remains device dependent.

- test_isr_120ms.txt
  The simulation test indicates that tripling the ISR execution time creates noticeable system call delays, for establishing the correlation between ISR complexity and system responsiveness.

- test_isr_200ms.txt
  This test reveals that extremely long ISR execution times create substantial delays in system service availability. The extreme case highlights the importance of implementing interrupt prioritization mechanisms.

Overall Analysis: ISR execution time directly impacts system call performance but doesn't affect hardware interrupt timing (END_IO uses device delays).

3. CPU Speed Tests (3 tests)

Purpose: Analyze CPU speed scaling effects

- test_cpu_normal.txt
  This test provides the baseline metrics (100ms) for CPU-bound task completion times under standard operating conditions.

- test_cpu_fast.txt
  This simulation demonstrates a 50% reduction in CPU execution time when processor speed doubles, while I/O and interrupt overhead remain unchanged Amdahl's law. The result clearly shows how I/O bound operations limit the maximum speedup in a computing system.

- test_cpu_slow.txt
  This specific simulation reveals that quarter-speed processor operation results in four times longer the CPU execution, being 400ms. This demonstrates the CPU-bound limitation scenario where I/O and interrupt overhead become negligible by comparison.

Overall Analysis: CPU speed scaling affects only CPU activities. I/O and interrupt overhead remain constant, demonstrating Amdahl's Law - speedup limited by non-CPU components.

4. **Address Size Tests (2 tests)**

Purpose: Analyze vector table addressing differences

- test_addr_2bytes.txt
  This simulation confirms that compact 2-byte vector addressing provides optimal memory usage for table vector storage without impacting execution timing performance.

- test_addr_4bytes.txt
  This test demonstrates that an expanded 4-byte addressing doubles memory usage for vector tables but still maintains execution timing when compared to the 2-byte system.

Overall Analysis: Vector size affects memory layout but not execution timing. 4-byte addresses use 2x more memory but provide larger address space for ISRs.

5. Workload Pattern Tests (3 tests)

Purpose: Analyze different workload characteristics

- test_workload_cpu.txt
  This simulation demonstrates CPU-bound workloads that maintain minimal interrupt overhead (approx. 10%), resulting in maximized CPU utilization with minimal kernel transitions.

- test_workload_io_balanced.txt
  The test represents an ideal general-purpose scenario with equal CPU and I/O distribution, showing well-balanced resource utilization across components.

- test_workload_io_intensive.txt
  This simulation reveals I/O-bound workloads experiencing high interrupt overhead of 80% due to frequent I/O operations. This results in kernel mode dominance with continuous context switching.

Overall Analysis: Different workload patterns show varying interrupt overhead percentages. I/O-intensive workloads spend more time in kernel mode handling interrupts.

6. **Device Performance Tests (6+ tests)**

Purpose: Analyze device timing impacts and edge cases

- test_edge_fast_devices.txt

The simulation demonstrates how strategic device selection with fast I/O can minimize I/O wait times, significantly reducing overall execution time, as we can see in the times listed below:
Devices: 1 (100ms), 15 (68ms), 7 (152ms)

- test_mixed_device_speeds.txt
  This test examines device environments where varying performance characteristics create execution bottlenecks, particularly with slow peripheral devices.

- test_burst_io.txt
  This simulation analyzes concentrated I/O activity patterns that generate temporary high interrupt loads. This tests system capability to handle peak demand. High interrupt frequency can be seen during the I/O bursts.

- test_variable_bursts.txt
  The simulation evaluates system behaviour under dynamic workloads with unpredictable I/O patterns. The testing validated the adaptive capabilities of interrupt handling mechanisms.

- test_long_running.txt
  This test validates system stability and consistent interrupt handling over extended operational periods.

- test_edge_highfreq.txt & test_edge_slow_devices.txt
  These two simulations examine the extreme operating conditions that validate the strong capabilities of the interrupt handling system under stress.