SYSC 4001 Assignment 1

Part I : Concepts

Stefan Martincevic, 101295641

Sadman Sajid, 101303949

06/10/2025

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. [Student 1]

The CPU hardware contains a wire called the interrupt-request line. There are usually two wires, one is for maskable interrupts, and the other one is for nonmaskable interrupts. The maskable interrupt is used to execute critical instruction sequences which cannot be interrupted by another interrupt. Nonmaskable interrupts can be interrupted by another interrupt while executing interrupt service routine. The interrupt mechanism implements interrupts priority levels where lower priority interrupts are not masked, meaning that if a higher priority interrupt arises, the CPU is able to pause the low priority interrupt service routine to service the higher priority interrupt. These priority levels are software controlled. Another hardware component is the interrupt controller which sets an interrupt vector number to forward to the processor. The rest of the interrupt is software controlled. The CPU checks the interrupt-request line after each instruction to see if it has received an external signal. When the CPU senses an interrupt signal on the interrupt-request line, it reads the interrupt number and sends it to the interrupt handler, which then performs the interrupt service routine. The appropriate interrupt service routine is found in an array (interrupt vector) that contains the address of each interrupt service routine. The appropriate interrupt service routine address is found in the interrupt vector at the address of the interrupt number and the CPU starts executing the interrupt service routine. The interrupt service routine starts by saving the CPU state before performing the interrupt. These include register values: the program counter, stack pointer, instruction register, general-purpose registers, and flag (status) registers. This is done because the interrupt service routine may also need to modify the register values, therefore the CPU state before executing the interrupt must be explicitly saved. Then, the interrupt service routine determines the cause of the interrupt and performs the necessary processing to service the specific interrupt. Once the interrupt has been cleared, the state from before the interrupt is restored and a return_from_interrupt instruction is executed allowing the CPU to return to the execution state before the interrupt.

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 connects the program and the OS, since user applications can't execute privileged instructions directly which is where system calls allow a user to request services from the OS kernel. The user programs run in user mode with restricted privileges, system calls let the requested privileged operations through by switching into kernel mode so the OS can perform the action on their behalf. Some examples of system calls would be open(), close(), read(), and write(). System calls are related to interrupts because it essentially is a software interrupt (trap), the hardware will trigger an interrupt to the CPU and make it transfer control to the kernel for the privileged instructions.

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
Discuss in detail all the steps that the printer must carry out for each of these two items

When the driver executes check_if_printer_OK(), it first checks the printer's status to make sure whether the printer is ready.The CPU pauses until the printer indicates that it is no longer busy due to the waitOK() step. The driver sends the subsequent character to the printer's buffer by issuing a print(text[i++]) command as soon as the printer indicates that it is ready. Before continuing, the function waitPrintEnd() ensures that the print head has finished processing the character. The paper is then moved slightly by advance_motor() to allow for the proper placement of the new output at the following location, and waitMotor() makes sure the paper is fed before the cycle is repeated. These procedures ensure that each character is printed neatly and sequentially, and that output only occurs when the printer is ready.

ii. print (LF, CR)

The parameters LF and CR are ASCII control characters, which means the printer interprets these characters as a control signal and not as a printable character in the text. When the printer receives the LF command (line feed), it moves the paper to the next line on the paper. That is done by activating the printer hardware (a step motor controller and a step motor) to move paper transport and send an LF acknowledgment signal to the motor controller once the move is completed. After completion of the LF move the printer process CR character (carriage return). Similarly to LF move, the printer activates the printer hardware (a step motor controller and a step motor) and drives the printing head to the position of left margin. Once the move is completed CR acknowledgment signal is sent to the motor controller and printer is ready for printing of the next line.

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

Offline operation works by separating the input and output from the main CPU allowing for the CPU to just handle the instruction/task while the I/O is handled by separated devices. The input was handled by the first device which read the punch cards and stored the instructions onto the magnetic tape. The magnetic tape with the instructions would then be loaded into the CPU where the instruction was executed then stored in an output tape. The output tape is then loaded in the output device where it reads the tape and prints out the content. The major advantage to this would be that the CPU's only job was to compute and not focus on I/O which made tasks execute faster. Some disadvantages of this were the expense of additional hardware, the delay in error detection due to the delayed detection of input/output failures, and the decreased flexibility resulting from the inability of users to interact with their programs in real time.

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 the “$” was forgotten the system would ignore it as a command and treat it as if it were just data that’s in the program. This would result in the OS not knowing the proper instructions as the loader might not be invoked which in return will give incorrect execution. This error can be simply avoided by ensuring that “$” is parsed which will keep the control cards separate from the user data and execute the commands properly.

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?

Having $END in the middle after $RUN has already started doing the job will terminate the current job. The OS will stop the execution of the program and prepare to load for the next job that’s in the batch. The $END ensures that a program doesn’t continue past the set boundary and keeps the correct sequence of the jobs.

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).

Two examples for privileged instructions would be enabling or disabling interrupts and I/O control.The CPU's response to hardware interrupts is controlled by interrupt enable or disable instructions. While activating interrupts allows regular processing of device requests, it's occasionally necessary to disable them during crucial activities to prevent CPU disruption. It is privileged because if a user program were able to stop interrupts, it may prevent crucial system functions like timers or I/O completion, which could cause the system to freeze or take up CPU resources. I/O device control is a privileged instruction because a user program may harm the device, corrupt data, or interfere with other processes if it were able to interact directly with hardware, such as the disk or printer. For instance, data can be sent straight to an I/O interface, such a printer, using the OUT instruction. Device protection is ensured by the kernel's safe and orderly handling of I/O requests rather than allowing user programs to accomplish this.

A halt instruction is an example of a privileged instruction. When the halt instruction is executed, the CPU immediately stops fetching instructions from main memory and executing them. The CPU remains turned on in an idle state and waits to receive a signal from an interrupt. The CPU then handles the interrupt by executing the appropriate interrupt service routine. After the interrupt has been handled the CPU resumes by fetching the next instruction after the halt instruction and executing it. The halt instruction is privileged as it completely stops the CPU from executing any further instructions. If the halt instruction could be used in user mode as an unprivileged instruction, user applications would be able to trigger the instruction at any time (with malicious intent) without needing the operating system's permission (kernel mode). This could result in system instability where a user application can trigger a halt, stopping the execution of other running processes (including operating system processors) in the operating system. This could lead to unpredictable behaviour and crashes as well as stop the operating system from having the ability of controlling its own processes.

Another example of a privileged instruction is timer management instructions. CPU timers are used to schedule different processes. These tasks are scheduled by setting a timer and waiting for it to finish. When the timer has expired, an interrupt is used to perform the specified task. Timers are used to allow for fair CPU usage across all processes. In this case when the CPU receives an interrupt it allows for the operating system to regain control and move to the next program in the queue, so that each program receives the processor time it requires. Additionally, the timer is used to generate interrupts at fixed intervals to ensure system reliability to perform tasks such as refreshing buffers and checking for timeouts. A fixed timer interval allows for the CPU to maintain an accurate clock cycle used to perform instructions consistently and correctly. Timer management is a privileged instruction as giving user applications access to the CPU timer could allow untrusted applications to directly set the timer to an unwanted value or disable it. This could cause CPU hogging and breaking the CPU's ability to multitask. Furthermore, if a CPU timer could be constantly reset, then no timer interrupts would be triggered, causing a system freeze. A change of timer intervals could also alter the CPU's clock cycle, creating an unreliable system. In this case many security checks could be bypassed (which are scheduled by the timer).

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.
ii. A detailed explanation of the execution order and how the routines interact.
iii. A step-by-step breakdown of what each routine performs during its execution.

A basic batch operating system consists of four major components: interrupt processing, device drivers, job sequencing, and the Control Language Interpreter (CLI). Interrupt Processing handles hardware signals and ensures that the CPU and I/O devices work together smoothly. Device drivers handle the complicated nature of I/O operations, such as reading from tape, and transferring data from external devices to memory. Job sequencing determines the order in which jobs are executed, including when to load and run programs. Lastly, the Control Language Interpreter reads control cards and invokes the corresponding OS functions based on the commands given.

When the CLI reads the first card ($LOAD TAPE1:), it recognizes it as an instruction to load a program. It invokes the loader, which uses the tape Device Driver to transfer the first file from TAPE1: into the User Area of memory. While the data is being sent, Interrupt Processing manages the hardware signals that indicate I/O completion or error. Once the software is entirely loaded into memory and the Loader has completed its job, control is returned to the CLI, which can read the next card.

The CLI recognizes the next card ($RUN) as a request to execute the program that was just loaded. The CLI then transfers control to Job Sequencing, which configures the CPU environment by initializing registers, setting the program counter, and switching to user mode. At this point, the CPU starts running the user program. During execution, Device Drivers handle any I/O actions, and Interrupt Processing notifies the OS when such activities are complete. When the program finishes, control is returned to the operating system. Job Sequencing determines whether there are any additional jobs to run, and if not, the CLI searches for new control cards.

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

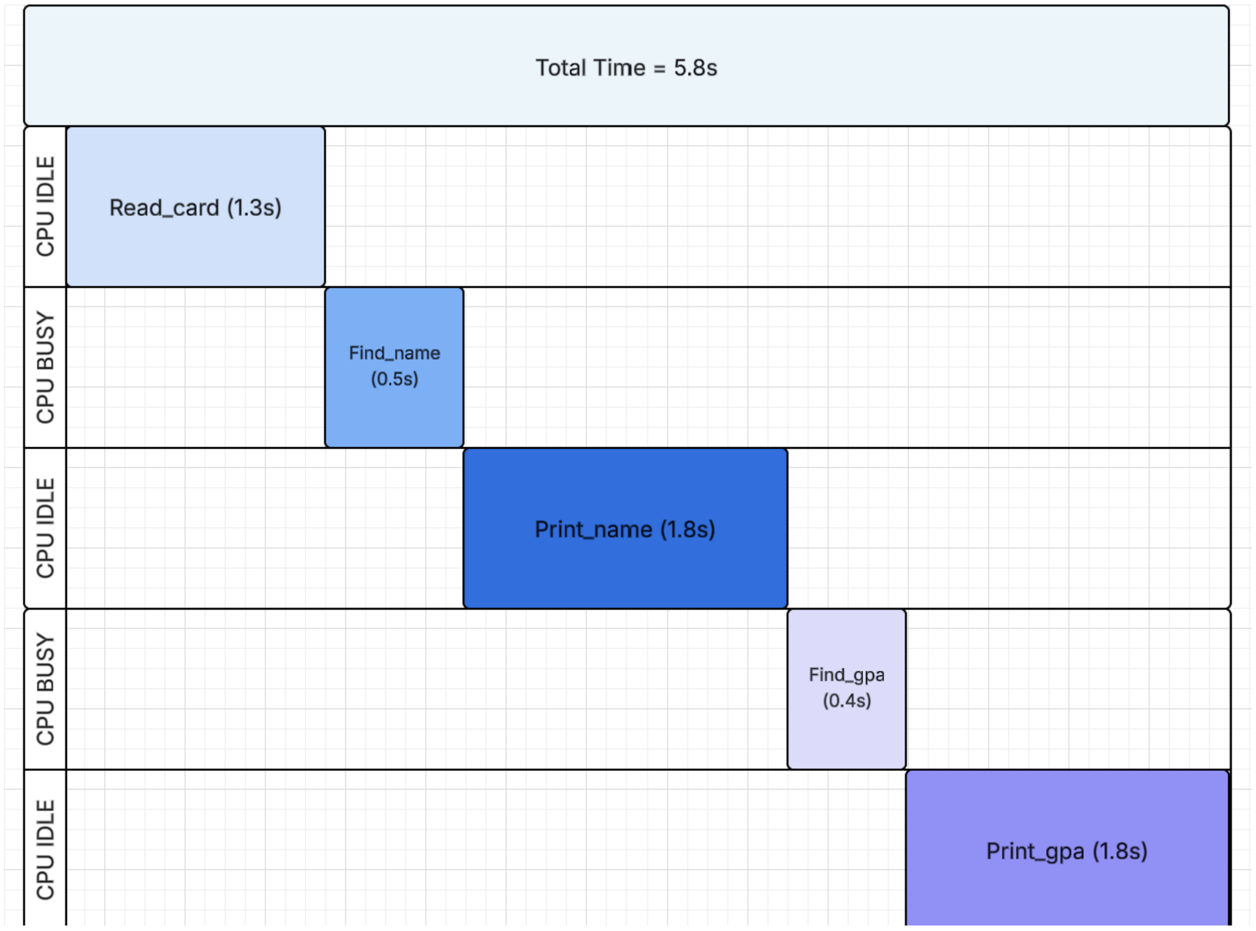

Figure 1: Gantt diagram displaying Timed I/O where the CPU waits during each input and output operation until it's complete.

Calculation for the Cycle Times

Time for 1 cycle = 1.3s + 0.5s + 1.8s + 0.4s + 1.8s

Time for 1 cycle = 5.8s

Time for all 284 cycles = 5.8s x 284

Time for all 284 cycles = 1647.2s

Time for all 284 cycles ≈ 27.45 mins

In the Timed I/O case, the CPU remains idle for all I/O operations, waiting for each device to complete before proceeding. There is no overlapping between processing and I/O, leading to inefficient CPU utilization.

ii) Polling

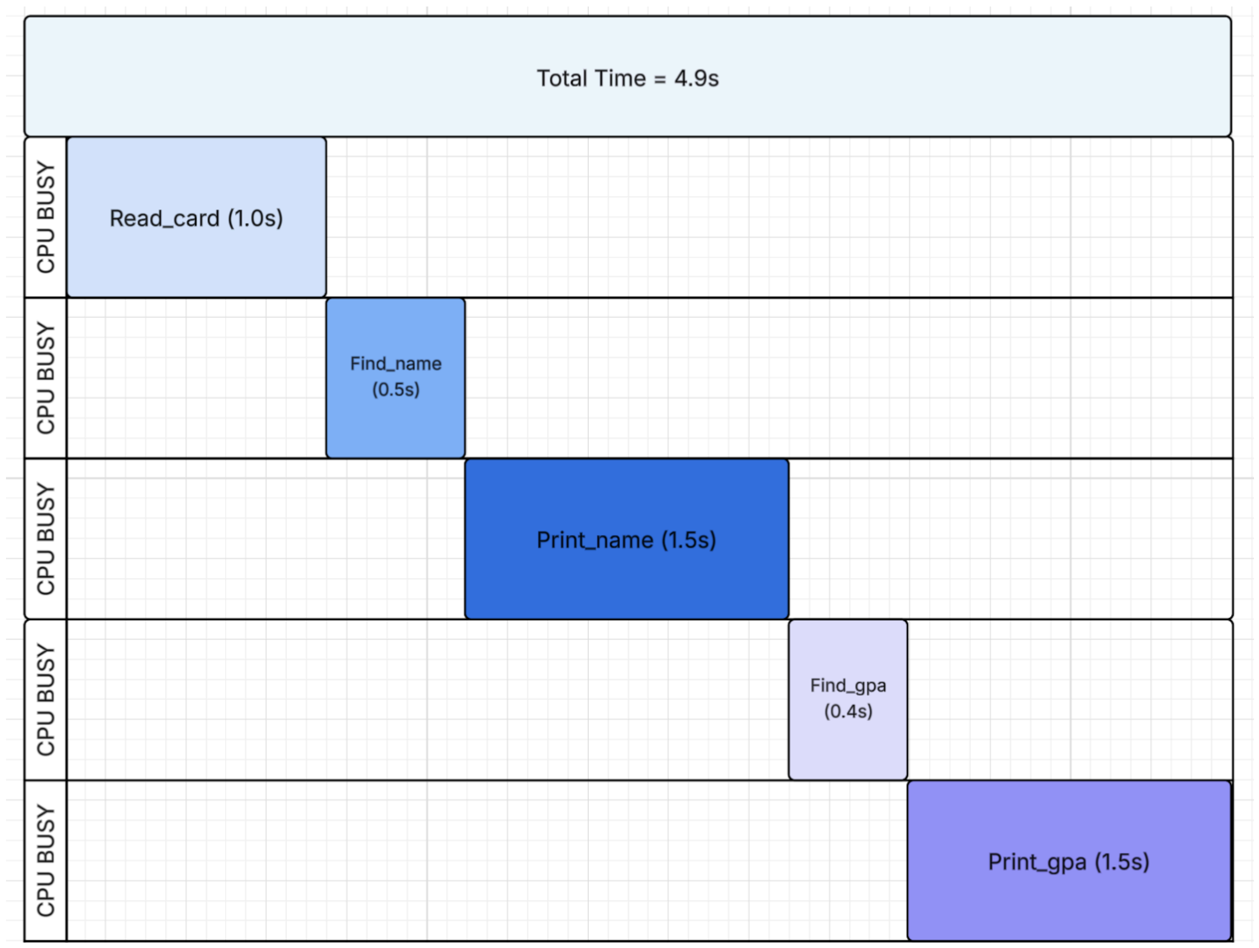

Figure 2: Gantt diagram displaying polling where the CPU checks device status while handling I/O.

Calculation for the Cycle Times

Time for 1 cycle = 1.0s + 0.5s + 1.5s + 0.4s + 1.5s
Time for 1 cycle = 4.9s

Time for all 284 cycles = 4.9s x 284
Time for all 284 cycles = 1391.6s
Time for all 284 cycles ≈ 23.19 mins

During polling, the CPU continuously checks for device flags that signal I/O completion. The CPU remains busy throughout the cycle, but does not accomplish any valuable work during polling. Polling remains inefficient in terms of CPU utilization.

iii) Interrupts

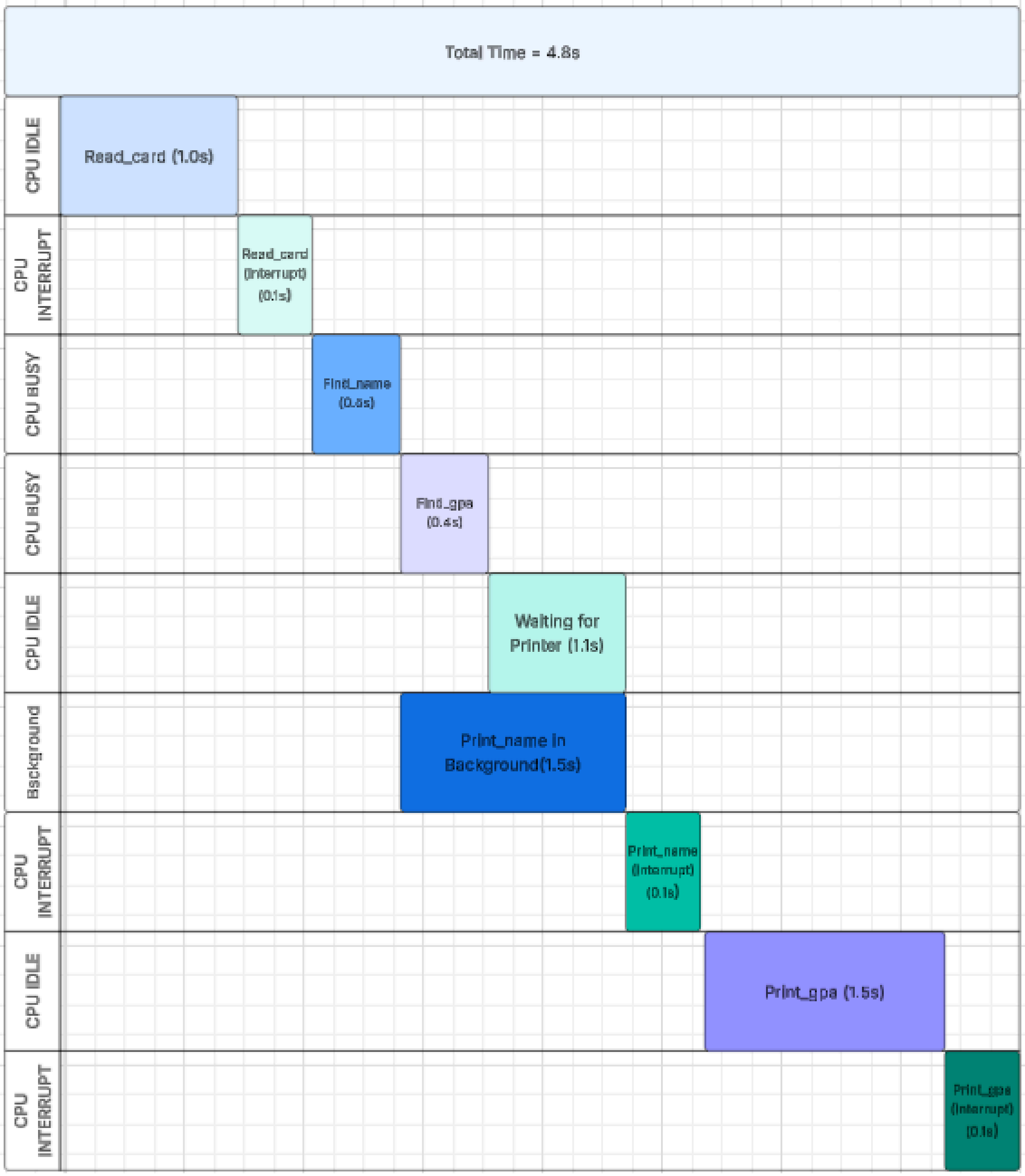

Figure 3: Gantt diagram displaying interrupts where the CPU executes user instructions while being interrupted for I/O events.

Calculation for the Cycle Times

Time for 1 cycle = 1s + 0.1s + 0.5s + 0.4s + 1.1s + 0.1s + 1.5s + 0.1s

Time for 1 cycle = 4.8s

Time for all 284 cycles = 4.8s x 284

Time for all 284 cycles = 1363.2s

Time for all 284 cycles ≈ 22.72 mins

The CPU will carry out user instructions in the interrupts case until an I/O device raises an interrupt, which will cause an ISR. As can be seen from the print_name output, this scenario permits overlap between the CPU's computation and any I/O devices. This increases efficiency, but the CPU does still need to wait for roughly 1.1 seconds since it calculates the GPA more quickly than the printer can print the student's name, thus the GPA must be printed when the printer is ready. In this scenario, CPU idle time is decreased but not entirely eliminated.

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

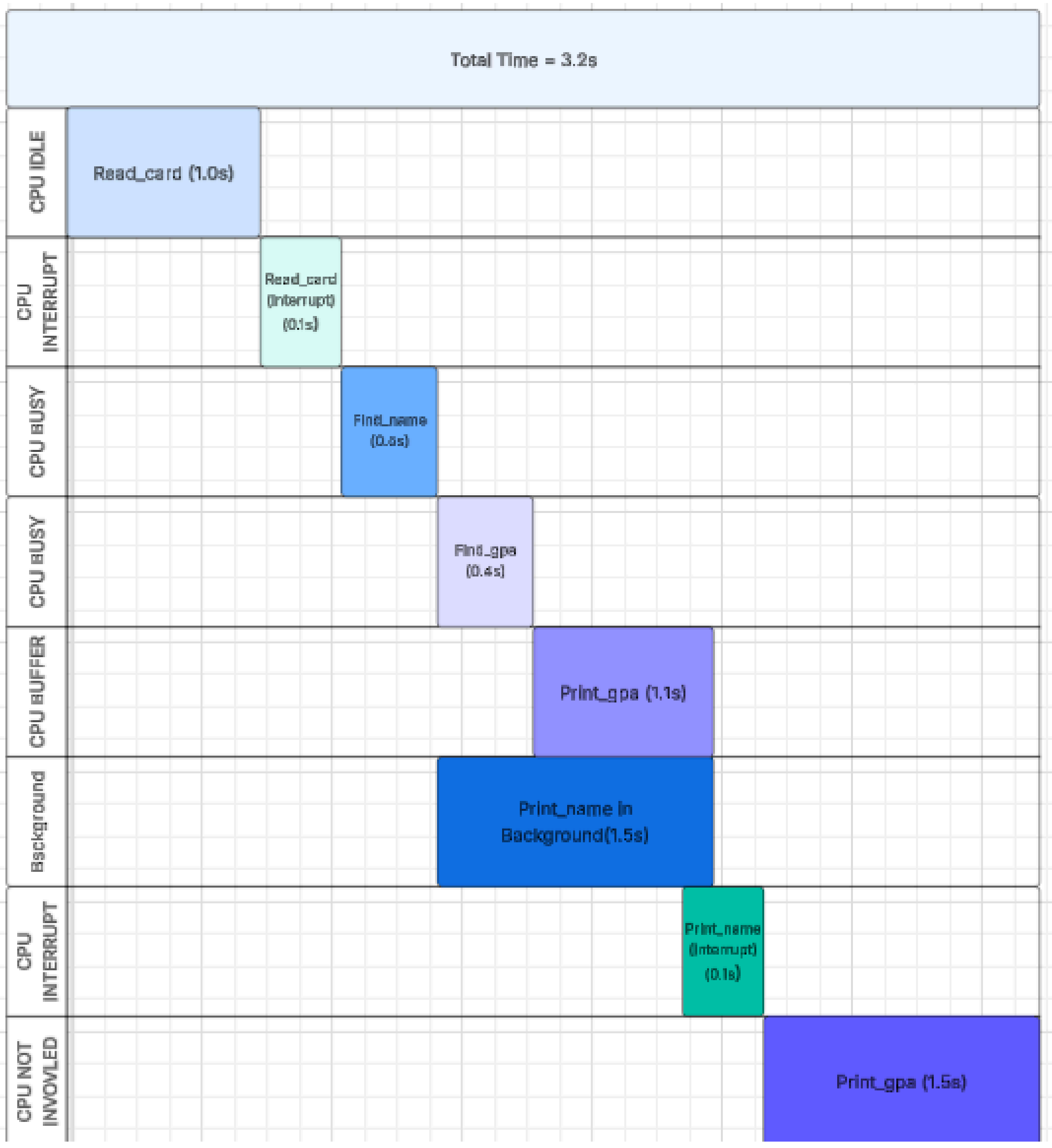

Figure 4: Gantt diagram displaying interrupts and buffering where the CPU processes data while I/O is managed through buffers.

Calculation for the Cycle Times

Time for 1 cycle = 1s + 0.1s + 0.5s + 0.4s + 1.1s + 0.1s
Time for 1 cycle = 3.2s

Time for all 284 cycles = 3.2s x 284
Time for all 284 cycles = 908.8s
Time for all 284 cycles ≈ 15.15 mins

When buffering and interrupts are used, data is stored in buffers and the CPU and I/O operate simultaneously. To maintain system efficiency, the CPU can execute one block as the next is being transferred. The CPU can proceed to the next card while the buffer and printer complete the task in the background, thus the 1.5 seconds required for "Print_GPA" does not stall it. Because of this overlap, interrupts with buffering are the most effective out of the 4 options.