**Hardware interrupts.**

As we know programming language like RUST always provides great efficiency with reliable security features. For example, RUST programming language only uses the trusted keyword that specified by the developers itself. By using only trusted keywords it proved the security of the OS [1].

Apart form those features an operating system must be extremely supportive for Hardware Interrupts and to memory management specially to perfect the CPU utilization and the memory of the system. From here onwards we will have a brief look about how rust programming language can be used to manage hardware interrupts, introduction and, last have a look how we can implement paging using RUST language in order to utilize the memory management of the operating system.

1. **What is known by an interrupt?**

In today almost each and all computing systems are interrupt-driven. An interrupt is a signal sent by a device connected to a computer or by a software running on the computer saying that operating system to temporarily pause what it’s doing currently and focus to the interrupt that has being send by the hardware or by software [2] . we can simply say an interrupt is a temporary suspension or cancellation of a service or a current process.

We can mainly divide interrupts into two main categories.

1. **Hardware interrupts.**
2. **Software interrupts.**

Diagram

Description automatically generated

**Figure 1.** types of interrupts

1. **Hardware interrupts.**

A hardware interrupt is an event connected to the hardware's status that can be signaled by an external hardware device. such as a request to begin I/O, a hardware failure, or something related. Hardware interrupts were designed to avoid literally wasting processor time in polling loops waiting for external events.

For example, when an I/O activity, such as reading data from a tape drive, is completed can be considered as hardware interrupts [3].

1. **Software interrupts.**

The CPU requests a software interrupt when specified instructions are executed or when certain conditions are satisfied. Each software interrupt signal is assigned to a specific interrupt handler. Software interrupts also referred as trap or exception. When the system interacts with device drivers or when a program requests OS services, software interrupts are often used.

For example, when fork() system call in Linux called it would generate a software interrupt to create a new process [4].

1. **Using Rust to manage Hardware interrupts.**

As we know operating system is an interrupt driven. Even if we create an operating system using rust language it still needs to support managing interrupts in order to increase the interrupt handling ability of the target operating system. Otherwise, it will lead to deadlock situations or race conditions.

1. **Overview of hardware interrupt handling in rust**

Interrupts allow linked hardware devices to inform the CPU. Instead of allowing the kernel to scan the keyboard for new characters on a regular basis. The process that continuously check for input is known as Polling. By using polling keyboard can notify the kernel of each keystroke.

Polling is the process by which a computer or controlling device waits for an external device to check its readiness or condition, which is frequently accomplished with low-level hardware. When a printer is connected through a parallel port, for example, the computer waits until the printer receives the next character [5]. These procedures can be as simple as reading a single bit.

circumstance in which a device is continuously checked for readiness and, if it is not, the computer returns to a different task. It also enables for shorter reaction times because the kernel can respond instantly instead of waiting at the next poll.

To pass interrupts to the CPU in efficient manner there is a segment called Interrupt controller. Role of the interrupt controller is to pass all the interrupts that generated by the hardware’s to CPU.

Diagram

Description automatically generated

**Figure 2.** Example diagram for PIC

1. **Intel 8259 PIC (Programmable interrupt controller)**

As we know all the hardware devices directly cannot be connected to the CPU. To connect to CPU, we must use interrupt controller. A programmable interrupt controller (PIC) is an integrated circuit that helps a microprocessor (or CPU) in handling interrupt requests (IRQ) from various sources (such as external I/O devices) that may occur repeatedly [6].

In nowadays most of the interrupt controllers are programmable. In rust programming language can use **intel 8259 Programmable interrupt controller** or **Advance programmable interrupt controller** in order toassign all hardware devices to PIC/APIC. Form **figure 3** can see how a real PIC looks like inside the system. In PIC each line has connected to a different hardware devices like mouse, keyboard etc.



**Figure 3**. intel 8259 PIC

The 8259 has eight interrupt lines and many communication lines with the CPU. Typical systems in old days had two 8259 PIC implementations, one primary and other secondary, each linked to one of the primary's interrupts lines

**Text

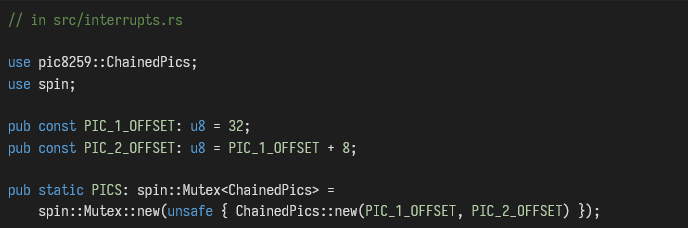
Description automatically generated**

**Figure 4**. Typical system that using 2 PIC's

The PICs' default setup is unusable because it transmits interrupt vector numbers ranging from 0 to 15 to the CPU. CPU exceptions have already taken up these addresses. Number 8 represents a double fault, for example. To resolve the overlapping problem, we must remap the PIC interrupt to separate values. The actual range doesn't important as long as it doesn't overlap with the exceptions, however 32-47 is usually picked because they are the first free integers following the 32 exception slots.

1. **PIC implementation.**

As we know default setup of 8259 PIC is not usable due to re-use of addresses that has already taken by different exceptions. So, we must manually configure a PIC by writing values to the command and data ports of the PICs [7, 8]. From **figure 5** can see how the primary/Secondary PIC redesigned to range of 32–47.

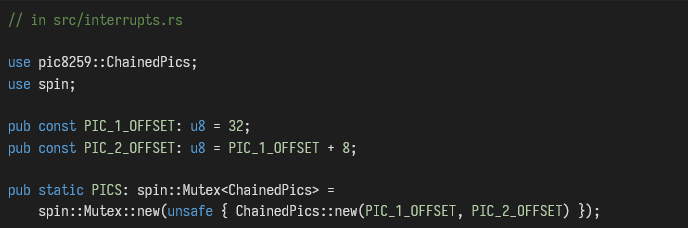


**Figure 5**. Example setting offset for primary/secondary PIC

1. **Enabling interrupts**

Until now interrupt signals not being sending to the CPU and CPU doesn’t have any idea about the interrupts and not responding to interrupt controller. Because interrupts are still disabled in CPU configuration. To enable external interrupts can use the interrupts::enable function [9].

From **figure 5** can see how the interrupts can be enabled using the Interrupts::enable function. After using relevant commands with the function specified CPU will start getting those interrupt signals generated by the PIC.



**Figure 6** enabling interrupt

With enabling the interrupts and while trying to execute the code segment it will show up some errors like double fault occurs. A double fault happens when there is a fault but the processor is unable to correctly execute the first instruction of the primary fault handler to completion, causing the processor to switch to processing the first instruction of the double-fault handler [10].

Reason of happening this double fault is because that hardware timer of the PIC enabled by default. So, timer will be creating interrupts upon enable of the interrupts. As a solution must specify a handler function to it.



Figure . handle function for timer interrupt

1. **Ending interrupts.**

Afterward following above mentioned methods and step we must have to explicitly say the PIC that interrupt was served and the system is ready to receive the next interrupt. Otherwise PIC hold the other interrupt’s thinking the system busy serving previous interrupt.

To notify the PIC that the interrupt was served successfully can use the **notify\_end\_of\_interrupt** command. This will uses the command and data ports to send the signal to the relevant controller.

As we know interrupts will occur asynchronously. Due to it asynchronous interrupts it could lead errors like deadlock situation and race conditions. In rust language it supports preventing many types of concurrency related bugs at the compile time itself. Event though exceptions like deadlock situation and race conditions could happen anytime.

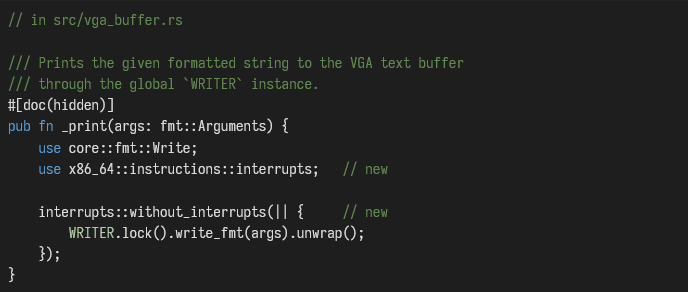
1. **Using Rust to oversee deadlocks & race conditions.**

A deadlock occurs when two computer programs that share the same resource effectively block each other from accessing the resource, causing both programs to stop working.

1. **Fixing the deadlock.**

To avoid deadlocks in system we can simply disable interrupts as long as the Mutex is locked in system. Mutex is the object that going to control the no of threads that going to access critical section in order to use the system resources. Mutex will deicides which thread going to enter the critical section it will only allow one thread at a time to enter critical section [11].

In rust language can use **Without\_interrrupts** function to ensure that the no interrupts happens while Mutex is locked.



**Figure 8.** Without\_interrrupts function example code

By disabling interrupts for short time is the solution for the deadlocks meanwhile its not the great solution either. For now, disabling interrupts while mutex locked is the considerable solution. This will also so increase the interrupt latency as well.

1. **Fixing race conditions.**

Race conditions also like deadlock situation. In here simply two threads trying to access a shared variable at the same time. For example, T1 thread accessing the shared variable and updating it value after that T2 thread also accessing the same variable and it just reading. But T2 reading the value that updated by T1.

Hence we can say due to the race condition output of the two or more thread accessing the shared variable always depend on the order of the execution of the thread.

In rust programming language data races are more likely to prevent from the start of the execution. But it won’t prevent general race conditions. Because every hardware is racy, OS is racy and other programs also racy hence preventing race conditions totally would be impossible because of this reason.

1. **Using Rust to manage power consumption.**

With this now we can have an OS that responsive to deadlocks and to race conditions. But a problem is there if we create an OS using above mentioned methods it will create an infinite loop of execution. CPU never goes to idle or sleep in this scenario.

It is very bad for OS in many ways. It will increase power consumption, decrease CPU performance, CPU will run in full speed even it doesn’t have any work to do and decreased life time etc. To make CPU efficient must put CPU in halt until next interrupt signal generated by PIC.

1. **CPU power management using halt functions.**

As we know CPU will endlessly executing on its max power. It is not good to ether CPU or to OS. To limit the CPU active time can use **hlt instruction**. **Figure 9** shows a sample code how to use the hlt instruction correctly. By using hlt instruction we only can create an energy efficient infinite loop.

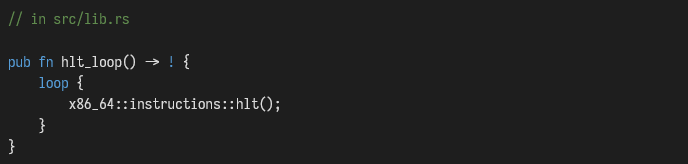


Figure . hlt instruction example

But we don’t need an infinite loop to be happen in low efficient mode unnecessarily. To stop this and to reduce CPU power consumption more can use **start panic** instructions. It wont lead CPU unnecessarily executing infinite loops. So, by using start panic instructions we can save the CPU power consumption more accurately.

Now the CPU will be able to handle interrupts that are coming from external devices. At last, need to do is adding support materials for keyboard inputs.

**Introduction to paging.**

In this section we will introduce to what is paging, memory protection mechanisms, paging on x86\_64 and in last paging implementation. Memory management is the process of managing and organizing the primary memory of a computer. It guarantees that memory space blocks are appropriately managed and allocated so that the operating system (OS), applications, and other running processes have the memory they require to perform their functions [12].

Paging also a very common technique that used in memory management of the operating system in order to improve the memory.

1. **Ways of protecting memory.**

In operating system each program reading and writing data to memory repeatedly. Imagine that in two or more programs running concurrently at the same time and they are reading and writing data to main memory from main memory. What if those programs accidently collide each other operations. It will give a huge problem.

As we see on of the main task of an operating system is to separate programs from each other. By that OS can confirm that hardware’s and memory areas that accessed by other programs are not interfered. In x86 based systems hardware’s supports two different ways of implementing memory protection.

1. **Segmentation.**
2. **Paging.**
3. **Segmentation.**

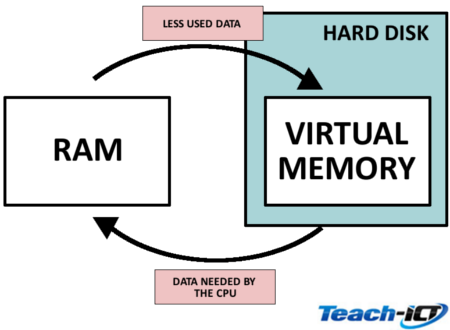
Segmentations was originally introduced in 1978 to increase the capacity of the memory space that can accessed. The functional behind the segmentation is it will divide the memory to variable size parts and each part named as a segment [13].

For example there are segments called code segment (CS) to store codes of the programs, Stack segment (SS) for stack operations, Data segment (DS / ES) to store other instructions and FS , GS segments that can be used to any purpose. There is a widely used technique that based on segmentation is virtual memory.

1. **Virtual memory.**

Virtual memory is an operating system feature that uses hardware and software to substitute for physical memory shortages. It moves data pages from random access memory (RAM) to disk storage. Virtual memory operates via a temporary process known as swapping, which combines RAM with hard disk space.

RAM is the physical memory on a computer that stores operating system data, running programs, and open documents. When RAM is insufficient, virtual memory can relocate data from RAM to a place known as a paging file. This procedure frees up RAM so that a computer can perform the work [14]



**Figure 10**. virtual memory architecture

In virtualization on problem that always occur is fragmentation. Fragmentation referred to as when a process is loaded from memory and when trying to assign a block to that process it cant be added sue to the small block size. Because of these memory block always stays unused [15].

1. **Paging.**

The simple idea of paging is divide both virtual memory and physical memory into small fixed sized blocks. The block of virtual memory is called as **page** and the block of physical address are called as **frame**. The ability to map each page to a frame separately allows for the partitioning of larger memory regions among non-continuous physical frames.

If we review the fragmented memory space example, but this time utilize paging rather than segmentation, the benefit of this becomes noticeable. From **figure 11** can see how the pages of virtual memory are mapped into frame in physical memory.

Diagram, table

Description automatically generated

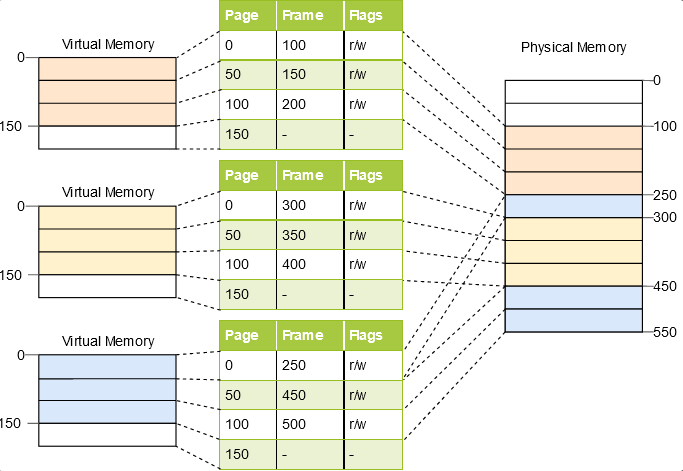
**Figure 11**. linked pages and frames example

In paging there is no chance of happening a fragmentation. Because in paging it uses same size of blocks. Since every block has a same size there a no frames that are too small and not being able to use. But hidden fragmentation possible even in this paging.

For example, imagine that there is a program with a size of 101. It would still require three pages of size 50, taking up an extra 49 bytes. This type of situations are known as internal fragmentation or hidden fragmentation.

1. **Paging tables.**

As we know each and every page of virtual memory must be mapped with a frame of physical memory. This mapping should be stored. To store those details paging uses a structure called **page table**. It includes details like page number, Frame number and readable / writable ability of the page.



**Figure 12**.paging table example

But in this table there is a major problem. Imagine that the program has 5 virtual pages. But it only needs 5 physical frames also page table and page table have over millions of entries. We can’t delete those extra entries manually. To as a solution can use **Two-Level page table**.

In two level page table also, there will be extra entries. but more less number.

**Paging implementation.**

1. **Introduction to paging implementation.**

In most of case kernel of the operating system runs on virtual address. But if we run the kernel on the virtual address we will be having some issues like couldn’t access the paging table. Because kernel is running on virtual address and paging table stored in physical memory.

Although there are some methods that can used to access the paging table form virtual address. But having kernel run on virtual address gives some advantages like improved safety, responsive to page fault are some of them.

1. **Accessing page tables.**

To implements page table accessing methods we need the support of bootloader. So, we must first configure the bootloader and then must configure the functions to convert virtual address to physical address.

we are unable to directly access physical addresses from kernel because of it is also running on virtual address. For instance, when we access address 4 KiB, we really access the level 4 page table's virtual address 4 KiB rather than its physical address.

Only a virtual address that corresponds to the physical address 4 KiB can be used to access it. We must therefore map some virtual pages to page table frames to access them. These mappings can be made in a variety of methods, and they all let us access any page table frame.

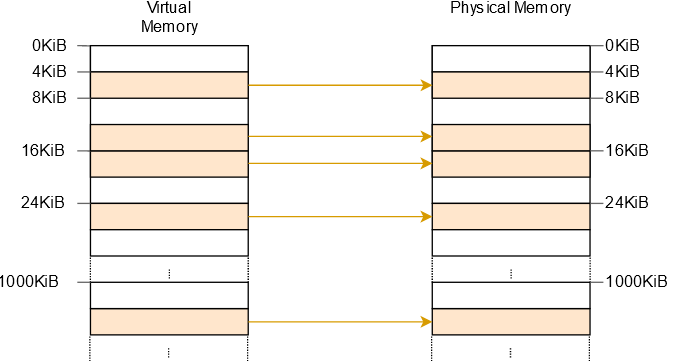
1. **Identity mapping.**
2. **Map the complete physical memory.**
3. **Recursive page mapping.**

Are the some of widely used methods to convert virtual address to physical address itself

1. **Identity mapping.**

Identity Mapping also referred as 1:1 Paging. It is a Paging table mapping option where a percentage of virtual addresses are mapped to physical addresses That has the same value. For an example 0xb8000 is 0xb8000 if identity paging is enabled and the region is identity mapped. From **Figure 13** can see how the actual identity mapping done [16].

By using this identity mapping method we can get the pages in protected mode like it allows us to set up paging without any issues.

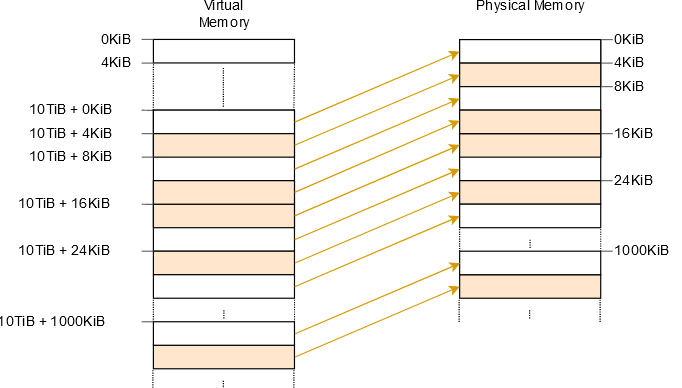


**Figure 13**. identity mapping example

1. **Map the complete physical memory.**

We can avoid lots of problems by mapping the whole physical memory to virtual addresses. This method allows our kernel to access physical memory including page table frames of other address spaces and also there are no unmapped pages left in this method all pages are mapped to frame itself.

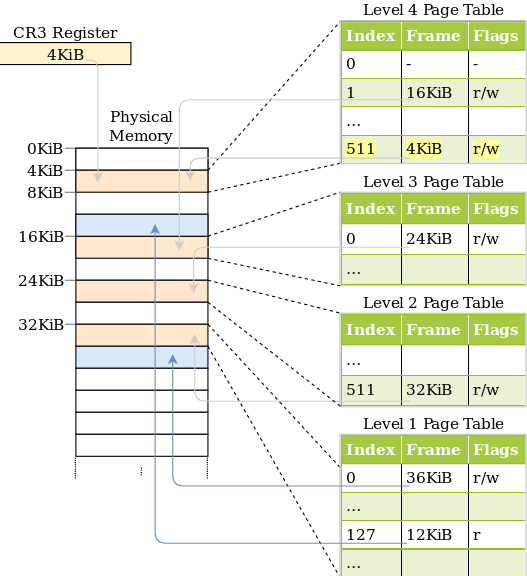
The one disadvantage of this complete physical memory map method is it need some additional page tables to store the mapping of the physical memory.



**Figure 14.** complete mapping example

1. **Recursive page table mapping.**

This approach doesn’t need any additional tables only need to do is map the page table recursively. Simply the idea is to map entry of level 4 page table to the level 4 table itself and so on. This method is the most effective one that can used to convert virtual address to physical address.



**Figure 15**. Recursive mapping example

1. **Implementing paging using RUST.**

As mentioned earlier we need the support of the bootloader to access the page table from virtual address. It means we need to map entry of the level 4 table recursively. Without doing this we can’t access it.

1. **Bootloader support.**

This is the segment that we going to configure to get the page table that the kernel runs on. Because only bootloader has the access to the page tables and by using it can create any type of mapping that we need.

To enable bootloader support simply we can use the **map\_Physical\_memory** function to the bootloader and then we are done. Can create any type of mapping that we need. To communicate with the virtual address range to our kernel the bootloader will pass the boot information structure to the kernel [17].



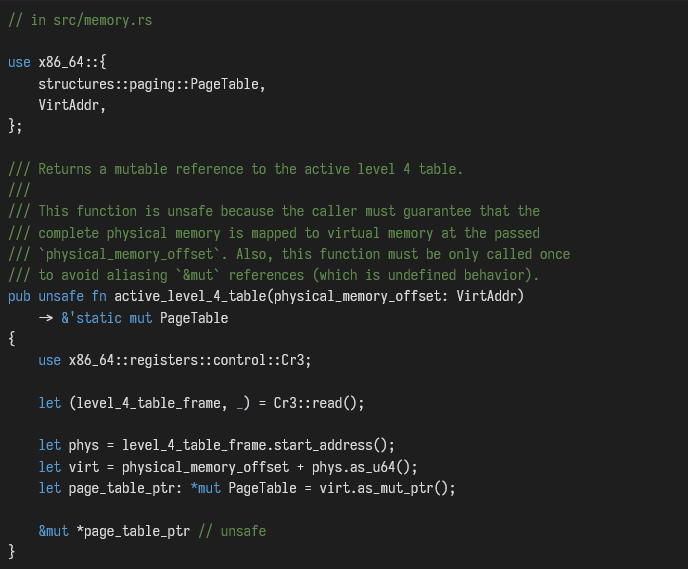
**Figure 16.** enabling bootloader function example

Now we have the access to the physical memory. Then we must do is start implementing the page table code to do mapping.

1. **Accessing the page tables.**

In previously we couldn’t be able to retrieve the details of the page table due to the kernel runed on virtual address. Now we solved that problem by enabling the bootloader function. But still now also we can’t read the address from physical address itself. We only created the link between virtual address and physical address.

Now we all have to do is create a function that returns all the references of active level 4 page table. To do that we can use **active\_level\_4\_table** function.



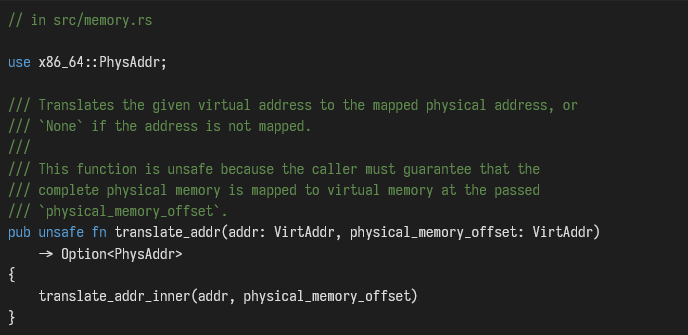
**Figure 17**. active\_level\_4\_table example code

As shown in figure 17 first need to read the physical frame of the active level 4 table from the register. We simply do is take its physical start address and convert it to a u64 and finally need to add it to the physical\_memory\_offset to get the virtual address according to mapping of the frame in the page table.

1. **Translating addresses.**

Afterward creating all above functions we have to do is configure the translator to translate virtual address to a physical address. To do that we need to do is create a function that translate level 4-page table until it matches the correct mapped frame.

To do this translation we can use a function called **translate\_addr\_inner**. this function will start reading physical address of corresponded mapped frame and will convert it to a physical address.



**Figure 18**. translate address function example

But the problem is this translate\_addr function only support for small size of page tables. To allow huge page tables addresses to be converted we can use offset page table as a solution.

So far we have done with the paging implementation. By following above mentioned steps can create successful link between virtual and physical address to read the memory form virtual address.

# References

|  |  |
| --- | --- |
| [1] | "chormium," [Online]. Available: https://www.chromium.org/Home. |
| [2] | "Techtarget," [Online]. Available: https://www.techtarget.com/whatis/definition/interrupt. |
| [3] | "geeks for geeks," [Online]. Available: https://www.geeksforgeeks.org/difference-between-hardware-interrupt-and-software-interrupt/. |
| [4] | "quora," [Online]. Available: https://www.quora.com/What-is-an-example-of-software-interrupt. |
| [5] | "wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Polling\_(computer\_science). |
| [6] | "wikipeedia," [Online]. Available: https://en.wikipedia.org/wiki/Programmable\_interrupt\_controller. |
| [7] | "OSDEV," [Online]. Available: https://wiki.osdev.org/8259\_PIC. |
| [8] | "docs.rs," [Online]. Available: https://docs.rs/crate/pic8259/0.10.1/source/src/lib.rs. |
| [9] | "docs.rs," [Online]. Available: https://docs.rs/x86\_64/0.9.6/x86\_64/instructions/interrupts/fn.enable\_interrupts\_and\_hlt.html. |
| [10] | "carnegie mellon," [Online]. Available: https://www.cs.cmu.edu/~410/doc/triple.html. |
| [11] | "tutorial sport," [Online]. Available: https://www.tutorialspoint.com/mutex-vs-semaphore. |
| [12] | "tech target," [Online]. Available: https://www.techtarget.com/whatis/definition/memory-management. |
| [13] | "javapoint," [Online]. Available: https://www.javatpoint.com/os-segmentation. |
| [14] | "techno monitor," [Online]. Available: https://techmonitor.ai/what-is/what-is-virtual-memory-4929986. |
| [15] | "byjus.com," [Online]. Available: https://byjus.com/gate/fragmentation-in-os-notes/. |
| [16] | "osdev," [Online]. Available: https://wiki.osdev.org/Identity\_Paging. |
| [17] | "ionos," [Online]. Available: https://www.ionos.com/digitalguide/server/configuration/what-is-a-bootloader/. |
| [18] | "quora," [Online]. Available: https://www.quora.com/What-is-an-example-of-software-interrupt. |