Sri Lanka Institute of Information Technology

# Implementing an Operating System using Rust

**Assignment Report**

IE2032- Secure Operating Systems

B.Sc. (Hons)in Information Technology Specializing in Cyber Security

Course Code: IE2032

Date: 10/20/2022

Batch: CS 2.2

Project Title: Implementing an Operating System using Rust

Group Members :

| | |
|---|---|
| Dissanayake W.P.D.B. | - IT21372308 |
| Zakey M.S.M.A. | - IT21299902 |
| Dilhara W. M. A. | - IT21299452 |
| Pemachandra T.H.R.T. | - IT21301322 |

# Terms Of Reference

The report was produced and submitted to fulfill the specifications for the IE2032 module at the Sri Lanka Institute of Information Technology.

# Acknowledgement

We would want to take this chance to express our gratitude to our Module lecturer, the SLIIT teaching staff, and the organizers for accommodating my last-minute inquiries and giving up their important time to mentor us through this assignment, which was a brand-new difficulty. We would also like to thank the lecturer for devoting long hours of his time to assist with the chosen topic. We also wish to say a big thank you to our parents for supporting us out.

# Contents

# Table of Figures

# Introduction

Rust is a static compiled language with a comprehensive type system and ownership concept that is both fast and memory economical. It may enable performance-critical services while ensuring memory and thread safety, allowing developers to debug at build time.

Furthermore, Rust offers excellent documentation and a consumer compiler with high-end tools like as integrated package management and a multi-editor with capabilities such as type inspection and auto-completion. Rust prevents all crashes, and it's worth noting that, like JavaScript, Ruby, and Python, rust is safe by default. This is far more effective than C/C++ since we can never build incorrect parallel code in rust. It is particularly quick at expressing a wide range of programming paradigms.

# 1. What is Rust and Why Rust ?

## 1.1 What is Rust

Rust is a low-level multi-paradigm programming language focused on safety and efficiency that overcomes problems that C/C++ has continued to struggle with for a long period of time, such as memory errors and developing concurrent applications [1].

Rust has three major advantages,

- The compiler provides improved memory safety.
- Concurrency is made easier because of the data ownership paradigm, which prevents data races.
- Abstractions at no expense.

## 1.2 What does rust do?

Rust, although being a low-level language, is beneficial when you need to get more out of your resources. Because it is statically typed, the type system aids in the elimination of certain types of problems during compilation. As a result, you will most likely utilize it when resources are restricted, and it is necessary that your application does not fail. High-level dynamically typed languages, such as Python and JavaScript, on the other hand, are excellent for things like fast prototypes.

Rust is not an object-oriented programming language. However, it includes certain object-oriented capabilities, such as the ability to build structs, which may have both data and related functions on that data, comparable to classes but without inheritance. In comparison to languages such as Java, Rust does not employ inheritance and instead relies on characteristics to create polymorphism [1].

## 1.3 Rust vs C++

When attempting to prevent undefined behavior in C++, developers face greater challenges. The borrow checker in Rust enables you to prevent unsafe practices by designing. This eliminates an entire class of bugs, which is crucial. Furthermore, Rust is a far more current and, in some ways, better-designed language. The sophisticated type system, in particular, will assist you even if its primary goal is not to detect memory issues, and since it is new, it can design its tooling with guiding principles in mind not having to worry about old codebases [1].

The slogan for Rust is "A language that empowers everyone to write dependable and efficient software."

While Rust began as a C++ alternative, it is evident that they are aiming higher, attempting to enable lower-level programming approachable to an increasing number of individuals who may not be capable of mastering C++ . Rust is not a replacement, but rather a language that offers up new avenues of potential, one of which we shall describe in the next chapter.

# 2. How to implement an OS using Rust

This chapter contains the steps to create an operating system in the Rust programming language.

## 2.1 A Freestanding Rust Binary

We need code that is independent of any operating system features in order to develop an operating system kernel. This implies that we won't be able to utilize heap memory, threads, files, the network, standard output, random numbers, , or any other capabilities that require OS abstractions or particular hardware.

So, the first step in building an own operating system kernel is to write a Rust executable that does not reference the standard libraries. This allows Rust programs to execute on bare metal without the need for an underlying OS. This sort of executable is sometimes referred to as "freestanding" or "bare-metal."

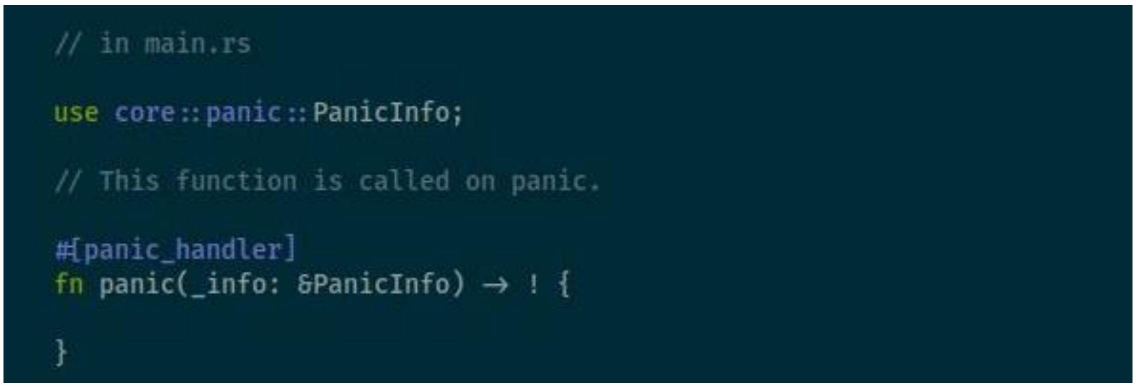
### Disabling the Standard Library

All Rust crates, by default, connect towards the standard library, which itself is dependent on the operating system for capabilities like threads, files, and networking. It also relies upon C standard library `libc`, which interacts closely with operating system services.

As a result, we should disable the standard library's automatic inclusion using the `no_std` attribute. After adding `no_std` attribute, we can’t include println since it a part of the standard library. Without the standard library, compiler is missing a `#[panic_handler]` function and the `eh_personality` language item [2].

#### Panic Implementation

The panic handler attribute specifies the function that should be called by the compiler when a panic occurs. The standard library contains its own panic handler method, but we must create it ourselves in a no std environment [2].

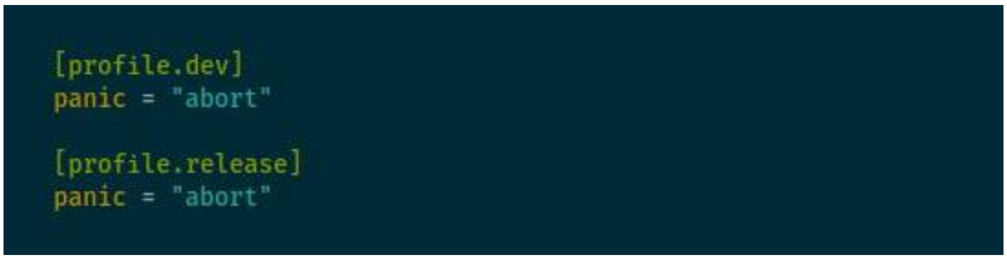


```rust
// in main.rs

use core::panic::PanicInfo;

// This function is called on panic.

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {

}
```

*Figure 2.1.1*

## The eh_personality Language Item

Language items are particular functions and types which the compiler need internally. The eh_personality language item denotes a function which used to implement stack unwinding. Rust offers the ability to abort on panic. This prevents the creation of unwinding symbol information, significantly reducing binary size. There are several areas in which we could disable unwinding. The simplest method is to include the following code in Cargo.toml [2].



```toml
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

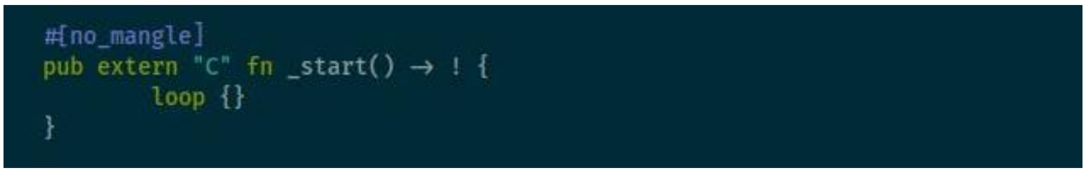
*Figure 2.1.2*

This causes the panic strategy for both the dev and release profiles to terminate. The language item eh personality will not be needed anymore.

## Defining our entry-point

In a rust program, the main function isn't the initial function that is called. The execution process begins with a runtime library. Crt0 is used by Rust. This, in turn, invokes the Rust runtime, which is a method denoted by the start language item. This, in turn, invokes the main function.  we do not use the crt0 and start language items. As a result, we must specify our own entrance point [3].

We use the #! [no main] attribute to notify the Rust compiler that we don't utilize the standard entry point chain. We no longer execute the main function. Instead, we've replaced the operating system's point of entry with our _start function [2].



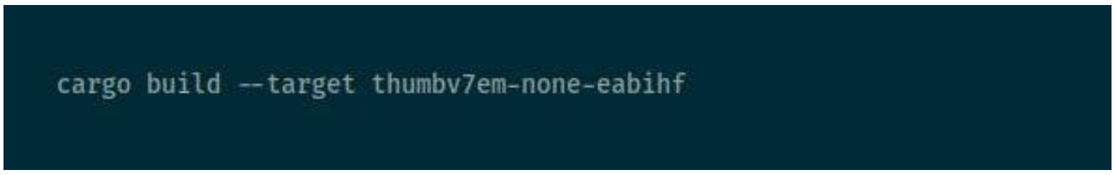
```
#[no_mangle]
pub extern "C" fn _start() → ! {
        loop {}
}
```

*Figure 2.1.3*

The ! will change the function into a diverging function that should never return. (In the future, we alter it to a command to shut down the system.)

<u>Solving linker errors by building a bare metal</u>

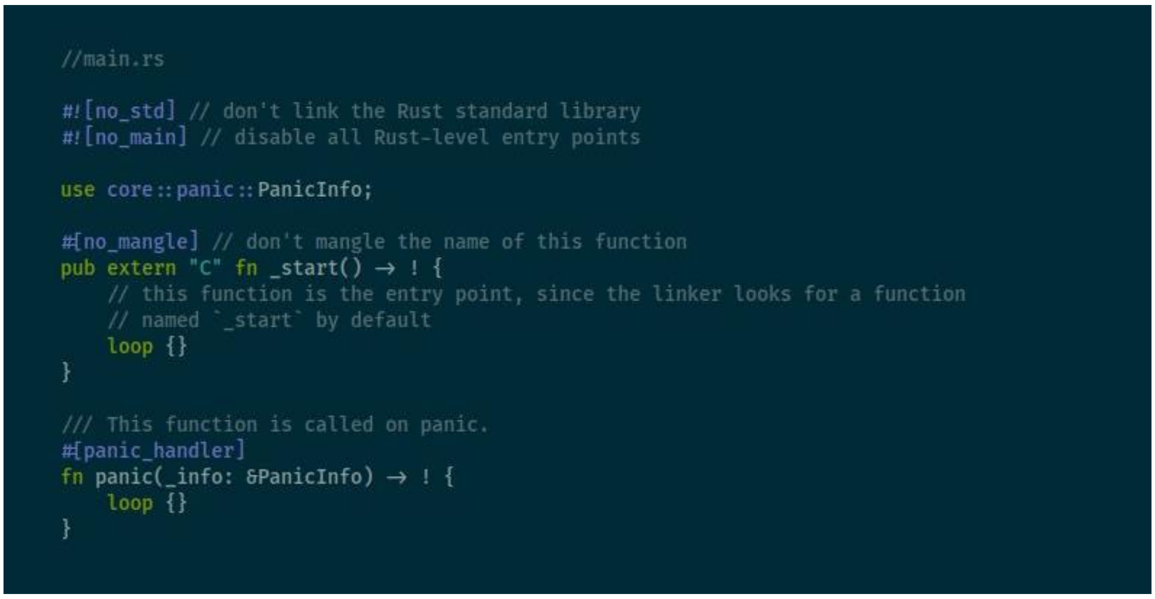
The linker is a program which converts the compiler's objects into executable code. The linker believes we are executing the C runtime(which used by an underlying OS) by default. To avoid linker error from occurring, we can create a bare-metal target. We can do it by building freestanding executable for this target using following code [3].



```
cargo build --target thumbv7em-none-eabihf
```

*Figure 2.1.4*

This is how a minimal freestanding rust binary looks.



```rust
//main.rs

#![no_std] // don't link the Rust standard library
#![no_main] // disable all Rust-level entry points

use core::panic::PanicInfo;

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() -> ! {
    // this function is the entry point, since the linker looks for a function
    // named `_start` by default
    loop {}
}

/// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

*Figure 2.1.5*

## 2.2 A Minimal Rust Kernel

In this section we will go through how to turn a freestanding binary into a minimal operating system kernel. Creating a custom target, merging our executable with a bootloader, and understanding how to print things to the screen are all part of this.

It is now time to build our own simple kernel. Our objective is to design a disk image that, when booted, writes "Hello World!" to the screen. We do this by upgrading the freestanding Rust binary from the previous chapter. We created the standalone binary using cargo, however various point of entry names and compilation settings were needed depending on the operating system. This happens because cargo always creates for the host system. We wish to compile for a particular target system.

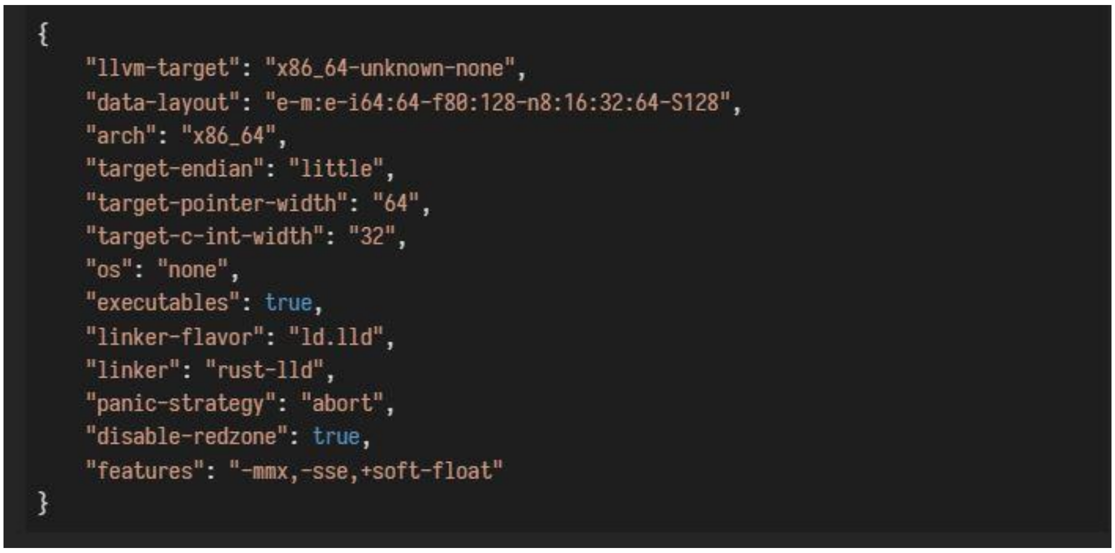
## Installing Rust Nightly

Rust has three release channels: stable, beta, and nightly. We'll require specific experimental features that are only available through the nightly channel. We can use software like rustup to manage rust installs. Using the rustup override nightly command, you may utilize a nightly compiler for the root folder.

## Target Specification

The target argument is compatible with a wide range of target systems. The target trinity describes the target's CPU architecture, vendor, operating system, and ABI. Rust supports several target triplets, including arm-Linux-androideabi for Android and wasm32-unknown-unknown for Web Assembly [4].

However, there are no feasible targets since we require certain peculiar configuration conditions for our target computer (no underlying OS). To begin, make an x86 64-blog_os.json file with the following general content: We need to update our target specification with new features,



```
{
    "llvm-target": "x86_64-unknown-none",
    "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
    "arch": "x86_64",
    "target-endian": "little",
    "target-pointer-width": "64",
    "target-c-int-width": "32",
    "os": "none",
    "executables": true,
    "linker-flavor": "ld.lld",
    "linker": "rust-lld",
    "panic-strategy": "abort",
    "disable-redzone": true,
    "features": "-mmx,-sse,+soft-float"
}
```

*Figure 2.2.1*

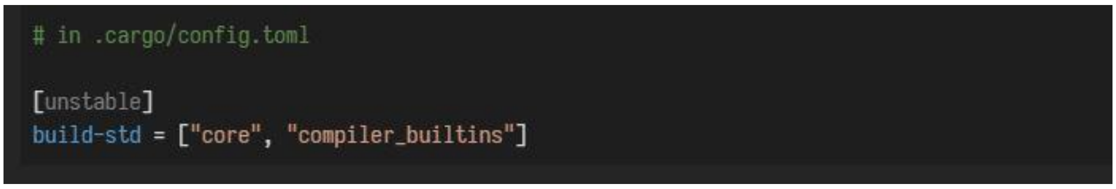
## Building Kernel

Our new target will be compiled using Linux conventions. This implies we'll need a _start entry point.

The build-std Option

That's where cargo's build-std functionality kicks in. It enables on-demand updating of core as well as other standard library crates rather to utilizing the precompiled versions that come with the Rust installation. Because this feature is currently in its initial stages and is only enabled on nightly Rust compilers, it is classed as "unstable."

To use the functionality, we should first construct a cargo configuration file called.cargo/config.toml that has the following content:



```
# in .cargo/config.toml

[unstable]
build-std = ["core", "compiler_builtins"]
```

*Figure 2.2.2*

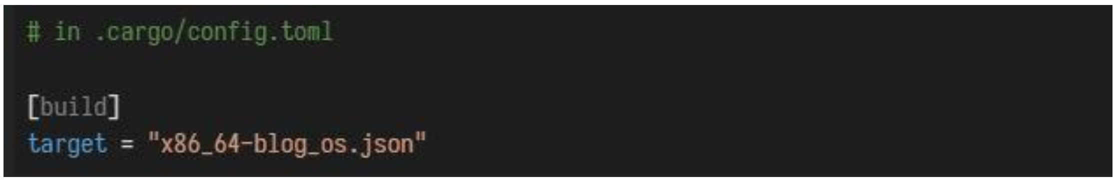
We can restart our build command after changing the unstable.build-std configuration key and downloading the rust-src component:

Memory-Related Intrinsic

Rust implies that almost all machines use the same collection of built-in methods. The majority of these functions are provided by the compiler builtins crate, that we just recompiled. However, because they are normally provided by the system's C library, certain memory-related functions in that crate are not available by default. These methods include memset, which changes all bytes in a memory block to a given value, memcpy, which transfers one memory block to another, and memcmp, that analyzes two memory blocks. Although none of those methods were necessary at the time to construct our kernel, they will be in the future [4].

Set a Default Target

We may change the default target to avoid giving the -target argument on each execution of cargo build. To do this, we include the following in our cargo configuration file,.cargo/config.toml:



```
# in .cargo/config.toml

[build]
target = "x86_64-blog_os.json"
```

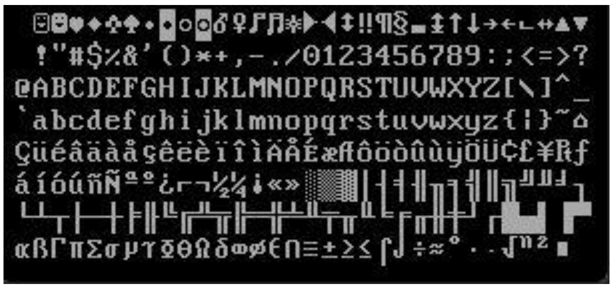
*Figure 2.2.3*

We can now use a simple cargo build to construct the kernel for a bare metal target. The _start entry point, that is going to invoked by the boot loader, is, however, still empty. It's time to put something on the screen from it.

## Printing to Screen

The VGA text buffer provides the most basic method for printing text on the screen. The contents of the screen are saved in a memory region specific to the VGA hardware. It is often made up of 25 lines, each featuring 80-character cells. Each character cell presents an ASCII character with a unique foreground and background color [4].

The screen output is as follows:



```
 ☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→←∟↔▲▼
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~⌂
ÇüéâäàåçêëèïîìÄÅÉæÆôöòûùÿÖÜ¢£¥₧ƒ
áíóúñÑªº¿⌐¬½¼¡«»░▒▓│┤╡╢╖╕╣║╗╝╜╛┐
└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀
αßΓπΣσµτΦΘΩδ∞φε∩≡±≥≤⌠⌡÷≈°∙·√ⁿ²■
```

*Figure 2.2.4*

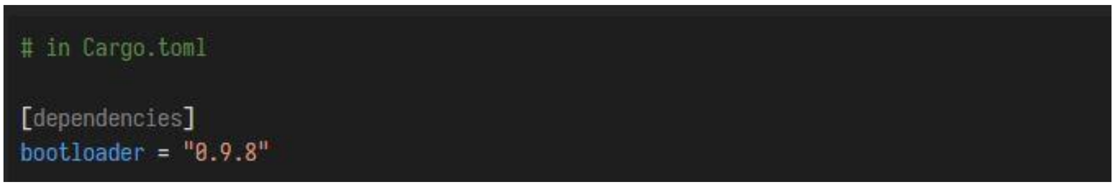
## **Running Kernel**

We can now run the executable because it accomplishes something visible. To make a bootable disk image, we should first link our built kernel to a bootloader. The disk image can then be loaded in the QEMU virtual machine or booted from a USB stick on physical hardware.

## Creating a Boot image

To create a bootable disk image, we must connect our generated kernel with a bootloader. As we learned in the booting section, the bootloader is in charge of resetting the CPU and loading our kernel.

We utilize the bootloader crate instead of writing our custom bootloader, that would be a distinct project. This crate creates a basic BIOS bootloader only using Rust and inline assembly, and no C dependencies. To utilize this to boot our kernel, we must place a dependency on it [4]:

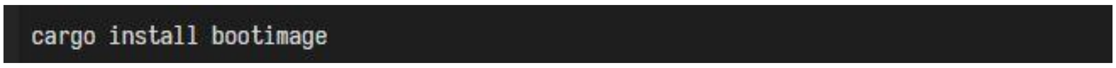


```toml
# in Cargo.toml

[dependencies]
bootloader = "0.9.8"
```

*Figure 2.2.5*

A bootable disk image will not be produced if the bootloader is included as a dependent. The problem is that we have to connect our kernel with the bootloader after compilation, although cargo does not allow post-build scripts.

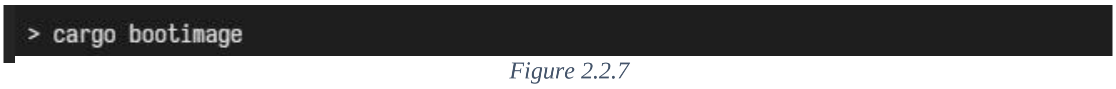
To solve this problem, we created boot image, a program that generates the kernel and bootloader prior putting them together to create a bootable disk image. To install the utility, type the following command into your terminal:



```
cargo install bootimage
```

*Figure 2.2.6*

To execute the boot image and produce the bootloader, you must possess the llvm-tools-preview rustup component installed. This may be done by executing rustup component add llvm-tools-preview. After installing the boot image and integrating the llvm-tools-preview component, we can generate a bootable disk image by running the:



```
> cargo bootimage
```

*Figure 2.2.7*

We recognize that the program uses cargo build to rebuild our kernel, so any changes you provide would be picked up immediately. The bootloader is then compiled. It is built once and cached, as well as all crate dependencies, therefore next builds will be much quicker. Finally, bootimage combines the bootloader with your kernel to generate a bootable disk image.

After executing the application, you should find a bootable disk image called bootimage-blog os.bin in your target/x86 64-blog os/debug directory. It may be executed on a virtual computer or transferred to a USB stick for execution on physical hardware [4].

## 2.3 Testing

Testing is a complicated field with varying terminology and organizational structures. The Rust community divides testing into two categories: unit tests and integration tests. Unit tests are smaller and more concentrated, focusing on one module at a time and can test private interfaces. Integration tests are completely independent of a library and utilize the code in the same manner that any other external programs would, by just utilizing the public interface and possibly exercising many modules per test [5].

### Unit Testing

The goal of unit tests is to isolate each unit of code from the rest of the code in order to rapidly identify where code is and not operating as intended. Unit tests will be placed in the **src** directory in each file that contains the code being tested. The convention is to include the test routines in a module named tests in each file and to annotate the module with **cfg (test)** .

The Tests Module and  #[cfg(test)]

The #[cfg(test)] declaration on the test’s component instructs Rust to compile and execute the test code only when cargo test is performed, not when cargo build is executed. When user merely wish to create the library, this reduces compilation time and space in the resultant generated artifact because the tests are not included. Because integration tests are stored in a separate directory, they do not require the #[cfg(test)] annotation. However, because unit tests are stored in the same files as the code, user will use #[cfg(test)] to tell the compiler not to include them in the built output [5].

### Testing Private Functions

There is disagreement in the testing profession about whether private functions should be tested properly, and some languages make testing private functions difficult or impossible. Whatever testing theory users follow, Rust's privacy rules make it possible to test private functions [5].

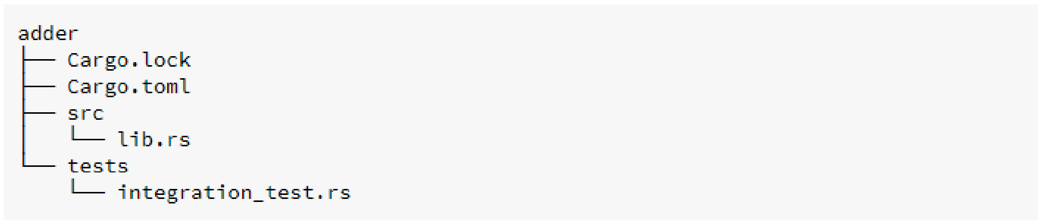
## Integration Tests

Integration tests in Rust are completely independent of the library. They utilize the library in the same manner that any other programs would, which means they can only use functions that are part of the public API of any library. Their function is to ensure that numerous sections of the library work well together. Units of code that perform successfully on their own may fail when combined, therefore test coverage of the integrated code is also critical. A tests directory is required to be able to write integration tests [5].

### The "tests" Directory

Users add a tests directory alongside src at the top level of the project directory. Cargo knows to search in this directory for integration test files. Users can then create as many test files as they like, and Cargo will compile each one as a separate crate [5].

The below Figure 2.3.1 - Structure of a Directoryshows the structure of a directory.



```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

*Figure 2.3.1 - Structure of a Directory*

Submodules in Integration Tests

As users add additional integration tests, users may want to create extra files in the tests directory to better organize them; for example, users may arrange the test functions by the feature they're evaluating. As previously stated, each file in the tests directory is created as its own independent crate, which is beneficial for generating various scopes to more precisely mimic how end users would use the crate. This implies that files in the tests directory do not behave the same way as files in the src directory [5].

The varied functionality of tests directory files is most obvious when user have a collection of helper calls to utilize in many integration test files and attempt to combine them into a common module [5].

Integration Tests for Binary Crates

If the project is a binary crate with only a src/main.rs file and no src/lib.rs file, users can not construct integration tests in the tests directory and use statements to call functions specified in the src/main.rs file into scope. Only library crates expose functions for usage by other crates; binary crates are intended to function on their own [5].

One of the reasons why Rust projects that produce a binary have a simple src/main.rs file that calls logic in the src/lib.rs file. Using that structure, integration tests can validate the library crate's ability to provide critical functionality. If the critical feature works, the miniscule fraction of code in the src/main.rs file should also work, and that short amount of code does not need to be tested [5].

## 2.4 CPU Exceptions

CPU exceptions arise in a variety of inappropriate scenarios, such as fetching an improper memory location or dividing by zero. To respond to them, we must create an interrupt descriptor table with handler routines [6].

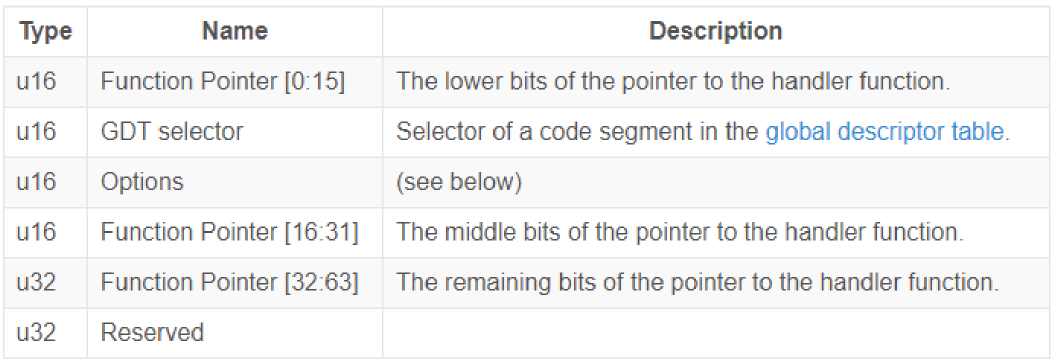
On x86, there are around 20 different forms of CPU exceptions. The most significant are:

- **Page Fault -** Unauthorized memory accesses cause a page fault. For instance, if the current instruction attempts to read from an undiscovered page or writes to a read-only page [6].
- **Invalid Opcode -** When users attempt to utilize new SSE instructions on an old CPU that does not support them, users get this error [6].
- **General Protection Fault -** This is the exception with the most diverse set of reasons. It can occur as a result of a variety of access breaches, such as attempting to execute an advantaged instruction in user-level code or writing restricted features in configuration records [6].
- **Double Fault -** When an exception occurs, the CPU attempts to invoke the appropriate handler code. If another exception exists even as the exception handler is being called, the CPU throws a double fault exception. This error also happens when no handler code for an exception is registered [6].
- **Triple Fault -** If an exception occurs while the CPU is attempting to execute the double fault handler code, a fatal triple fault is generated. Users cannot  manage or recognize a triple fault. Most CPUs respond by restarting the operating system and resetting themselves [6].

### The Interrupt Descriptor Table

To capture and handle exceptions, users must first create an Interrupt Descriptor Table (IDT). Users may define a handler function for each CPU exception in this table.

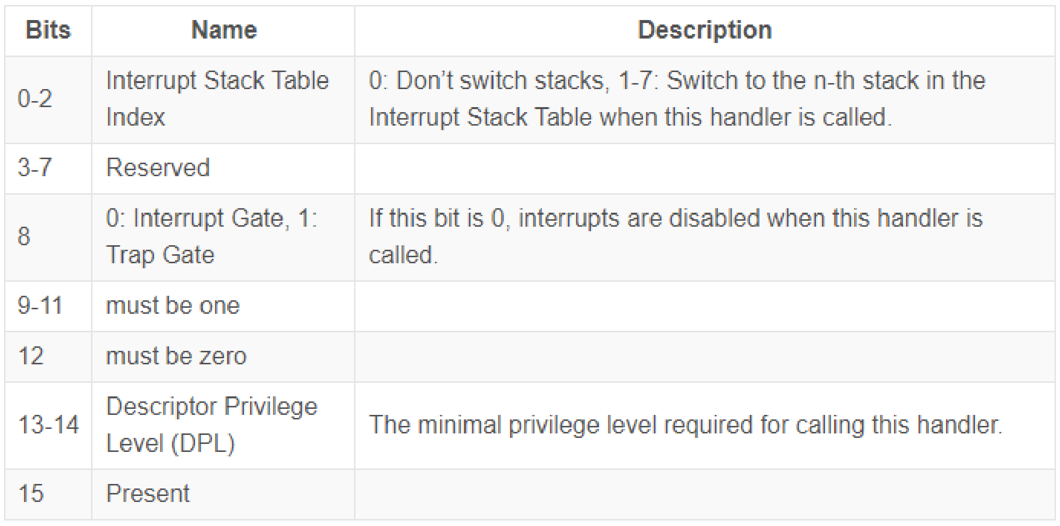
Because the hardware relies on this table, users must adhere to a certain structure. Each item must have the 16-byte structure as shown below in Figure 2.4.1.



| Type | Name | Description |
|---|---|---|
| u16 | Function Pointer [0:15] | The lower bits of the pointer to the handler function. |
| u16 | GDT selector | Selector of a code segment in the global descriptor table. |
| u16 | Options | (see below) |
| u16 | Function Pointer [16:31] | The middle bits of the pointer to the handler function. |
| u32 | Function Pointer [32:63] | The remaining bits of the pointer to the handler function. |
| u32 | Reserved | |

*Figure 2.4.1 - The Interrupt Descriptor Table*

The format of the options field is as follows in Figure 2.4.2 .



| Bits | Name | Description |
|---|---|---|
| 0-2 | Interrupt Stack Table Index | 0: Don’t switch stacks, 1-7: Switch to the n-th stack in the Interrupt Stack Table when this handler is called. |
| 3-7 | Reserved | |
| 8 | 0: Interrupt Gate, 1: Trap Gate | If this bit is 0, interrupts are disabled when this handler is called. |
| 9-11 | must be one | |
| 12 | must be zero | |
| 13-14 | Descriptor Privilege Level (DPL) | The minimal privilege level required for calling this handler. |
| 15 | Present | |

*Figure 2.4.2 - The Option Field*

Each exception has its own IDT index. The erroneous opcode exception, for example, has table index 6, whereas the page fault exception has table index 14. As a result, for each exception, the hardware may automatically load the associated IDT item. The IDT indices of all exceptions are shown in the "Vector number." column of the OSDev wiki's Exception Table [6].

When an exception occurs, the CPU does the following roughly [6]:

- Prompt some stack registers, such as the register and the RFLAGS register.
- Examine the Interrupt Descriptor Table for the matching item (IDT). When a page fault occurs, for instance, the CPU examines the 14th item.
- Verify the existence of the entry and, if not, issue a double fault.
- If the entry is an interrupt entrance, restrict hardware interrupts (bit 40 not set).
- Import the GDT selection supplied into the CS (code segment).
- Navigate to the handler function given.

## The Interrupt Calling Convention

Exceptions are analogous to procedure calls: The CPU then performs the first instruction of the called function. Following that, the CPU jumps to the return address and resumes execution of the parent function [6].

Moreover, there is a significant distinction among exceptions and function calls: a function call is intentionally prompted by a compiler-inserted call instruction, but an exception can exist at any instruction. So, need to explore at function calls in further depth to comprehend the implications of this discrepancy [6].

The specifics of a function call are specified by calling conventions. They indicate, for example, where function parameters are stored (e.g., in registers or on the stack) and how results are delivered [6]. For C functions (defined in the System V ABI), the following rules apply on x86 64 Linux:

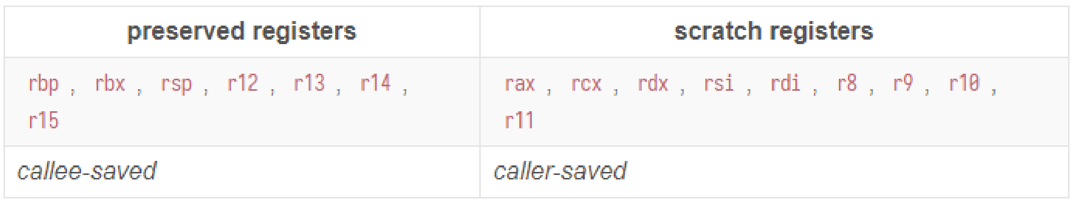
- The first six integer inputs are supplied by registers rdi, rsi, rdx, rcx, r8, and r9.
- On the stack, further parameters are supplied.
- The results are returned in rax and rdx formats.

## Preserved and Scratch Registers

The calling convention separates registers into two categories: preserved registers and scratch registers. The contents of preserved registers must not change between function calls. As a result, a called function (the "callee") may alter these registers only if it restores their original values before returning. Be a result, these registers are referred to as "callee-saved." It's usual practice to save these registers to the stack at the start of the function and reestablish them shortly before exiting [6].

A called function, on the other hand, has complete freedom to rewrite scratch registers. If the caller wishes to keep a scratch register's value across function calls, it must backup and restore it prior to the function call (e.g., by pushing it to the stack). As a result, the scratch registers are stored by the caller [6].

As seen in Figure 2.4.3 the C calling convention on x86 64 provides the following preserved and scratch registers:



| preserved registers | scratch registers |
|---|---|
| `rbp` , `rbx` , `rsp` , `r12` , `r13` , `r14` , `r15` | `rax` , `rcx` , `rdx` , `rsi` , `rdi` , `r8` , `r9` , `r10` , `r11` |
| *callee-saved* | *caller-saved* |

*Figure 2.4.3 - The Calling Convention*

Because the compiler is aware of these principles, it creates code in accordance with them. Most routines, for example, begin with a push rbp, which backups rbp on the stack (since it is a callee-saved register) [6].

## Preserving all Registers

Exceptions, unlike function calls, can occur on any instruction. In most circumstances, users will not even know if the resultant code will throw an exception at compilation time. The compiler, for example, has no way of knowing if an instruction generates a stack overflow or a page fault [6].

Users can not backup any registers before an exception occurs since users will not know when it will occur. This implies users cannot utilize a calling convention for exception handlers that relies on caller-saved registers. Instead, a calling convention that maintains all registers is required. As an example, the x86-interrupt calling convention ensures that all register values are restored to their original values upon function return [6].

## The Interrupt Stack Frame

The CPU transmits the return address before switching to the kernel function during a conventional function call (through the call instruction). The CPU releases this return address and moves to it on function return (via the ret instruction). As a result, the stack frame of a regular function call looks like this following Figure 2.4.4:

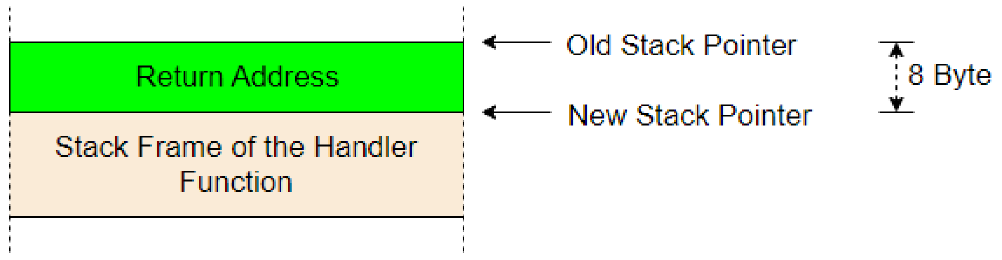


*Figure 2.4.4 - The Interrupt Stack Frame of Normal Function*

Dragging a return address, on the other hand, would not be sufficient for exception and interrupt handlers, because interrupt handlers sometimes execute in a distinct context (stack pointer, CPU flags, etc.). When an interrupt occurs, the CPU instead does the following [6]:

- **Saving the old stack pointer -** The CPU examines the contents of the stack pointer (rsp) and stack segment (ss) registers and stores them in an internal buffer.
- **Aligning the stack pointer -** Because an interrupt can occur at any instruction, the stack pointer can also have any value. However, because some CPU instructions (for

example, some SSE instructions) require the stack pointer to be positioned on a 16-byte boundary, the CPU executes this adjustment immediately after the interrupt.

- **Switching stacks (in some cases) -** When the CPU access transitions, such as when a CPU exception occurs in a user-mode application, a stack switch happens. The Interrupt Stack Table can also be used to configure stack switches for specific interruptions.
- **Pushing the old stack pointer -** The rsp and ss numbers from step 0 are pushed to the stack by the CPU. When heading back from an interrupt handler, this allows users to restore the original stack pointer.
- **Pushing and updating the** RFLAGS **register -** Various control and status bits are stored in the RFLAGS register. When an interrupt occurs, the CPU modifies certain bits and transmits the previous value.
- **Pushing the instruction pointer -** The CPU releases the instruction pointer (rip) and the code segment before proceeding to the interrupt handler procedure (cs). This is analogous to the return address push of a standard function call.
- **Pushing an error code (for some exceptions) -** The CPU sends an error code that identifies the reason of certain possible errors, including such page faults.
- **Invoking the interrupt handler -** The CPU receives the interrupt handler function's address and segment descriptor from the relevant field in the IDT. The handler is then called by placing the values into the rip and cs registers.

The below Figure 2.4.5 shows how looks like the interrupt stack frame is,

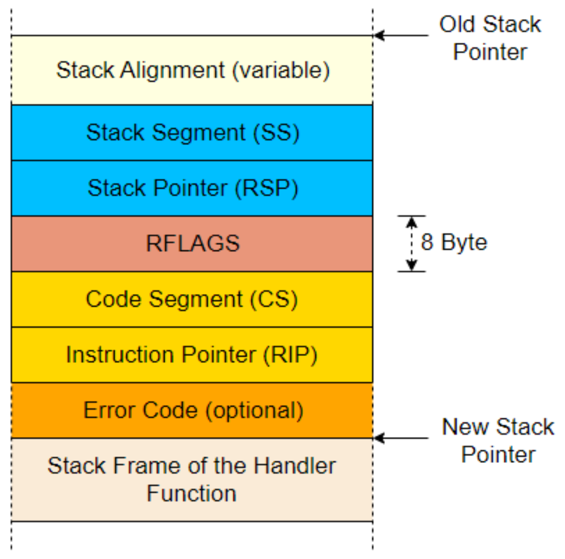


*Figure 2.4.5 - The Interrupt Stack Frame*

The InterruptStackFrame struct in the x86 64 crate represents the interrupt stack frame. It is supplied to interrupt handlers as &mut and can be used to get further information about the reason of the exception. Because only a few exceptions generate an error code, the struct lacks an error code field. These exceptions make use of the HandlerFuncWithErrCode function type, which has an extra error code parameter [6].

## 2.5 Double Faults

A double fault is a specific exception that happens when the CPU fails to call an exception handler. It happens, for example, when a page fault occurs but no page fault handler is registered in the Interrupt Descriptor Table (IDT). So it's comparable to catch-all blocks in exception-handling programming languages, such as catch(...) in C++ or catch(Exception e) in Java or C# [7].

A double fault works similarly to a standard exception. It has the vector number 8, and we may construct a standard handler function in the IDT for it. It is critical to provide a double fault handler because an unhandled double fault results in a deadly triple fault. Triple faults are not detectable, and most hardware responds with a system reset [7].

### Triggering a Double Fault

To write to the invalid address 0xdeadbeef, simply utilize unsecured. A page fault arises when the virtual address is not mapped to a physical address in the page tables. Because users did not register a page fault handler in the IDT, a double fault occurred. When users restart the kernel, users find that it goes into an infinite boot loop. The boot loop is caused by the following [7]:

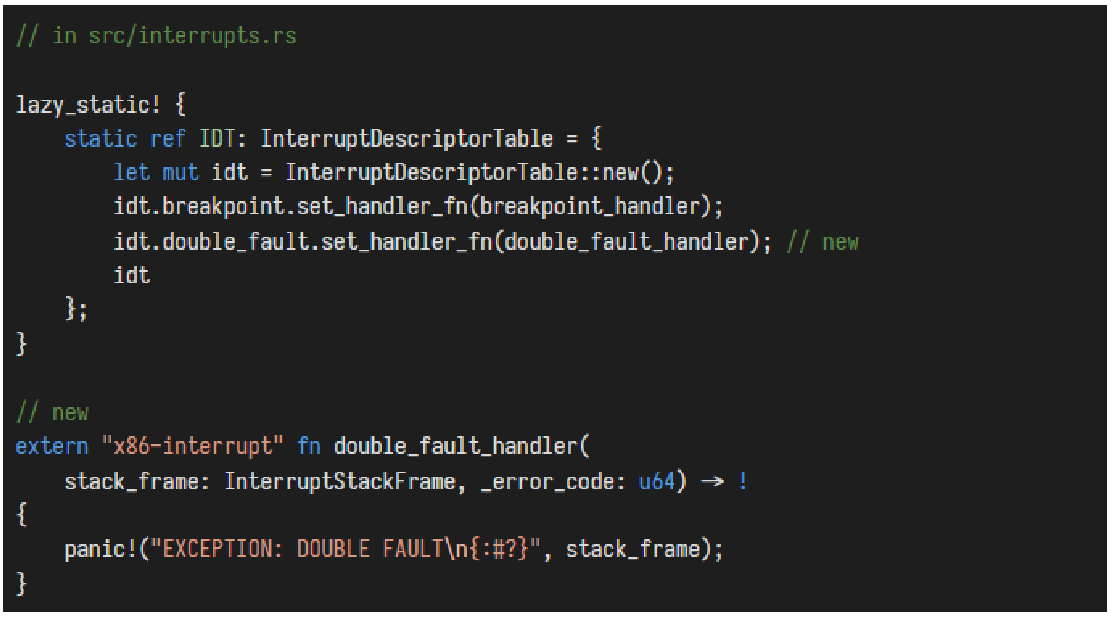
- The CPU attempts to write to 0xdeadbeef, resulting in a page fault.

- The CPU examines the associated IDT item and notices that no handler function is given. As a result, it is unable to contact the page fault handler, resulting in a double fault.
- The CPU examines the double fault handler's IDT entry, but this entrance, too, does not provide a handler function. As a result, a triple fault develops.
- A triple error is deadly. QEMU responds to it in the same way that most genuine hardware does, by performing a system reset.

To avoid the triple fault, users must either provide a handler method for page faults or a double fault handler. To avoid triple faults in all circumstances,   just begin with a double fault handler that is called for all unhandled exception types [7].

## A Double Fault Handler

A double fault is a regular exception with an error code; therefore, users can use the same handler method as previously did for the breakpoint [7]:

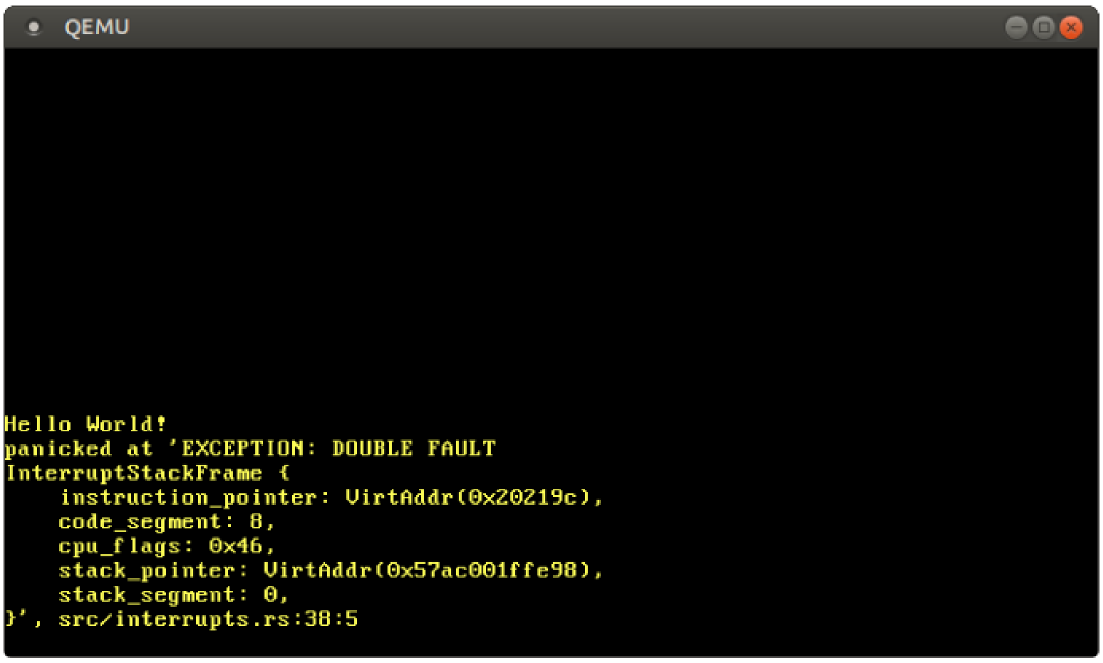


```rust
// in src/interrupts.rs

lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        idt.breakpoint.set_handler_fn(breakpoint_handler);
        idt.double_fault.set_handler_fn(double_fault_handler); // new
        idt
    };
}

// new
extern "x86-interrupt" fn double_fault_handler(
    stack_frame: InterruptStackFrame, _error_code: u64) -> !
{
    panic!("EXCEPTION: DOUBLE FAULT\n{:#?}", stack_frame);
}
```

*Figure 2.5.1 - A Double Fault Handler*

As illustrated in Figure 2.5.1, the handler sends a brief error message before dumping the exception stack frame. There is no purpose to report the error code of the double fault handler because it is always zero. The double fault handler differs from the breakpoint handler in that it is divergent. The x86 64 architecture does not support returning from a double fault exception [7].

When restart the kernel, users should observe the double fault handler being called as in Figure 2.5.2,



```
Hello World!
panicked at 'EXCEPTION: DOUBLE FAULT
InterruptStackFrame {
    instruction_pointer: VirtAddr(0x20219c),
    code_segment: 8,
    cpu_flags: 0x46,
    stack_pointer: VirtAddr(0x57ac001ffe98),
    stack_segment: 0,
}', src/interrupts.rs:38:5
```

*Figure 2.5.2 - Invoked Double Fault Handler*

What happened on above figure is [7],

- The CPU attempts to write to 0xdeadbeef, resulting in a page fault.
- The CPU, like previously, examines the appropriate element in the IDT and notices that no handler function is declared. As a result, a double fault arises.
- The CPU advances to the now-present double fault handler.

Because the CPU may now invoke the double fault handler, the triple fault (and the boot-loop) no longer happens [7].

## 2.6 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 [8].

Apart from 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.

### 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 [9] . 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.

- Hardware interrupts.
- Software interrupts.

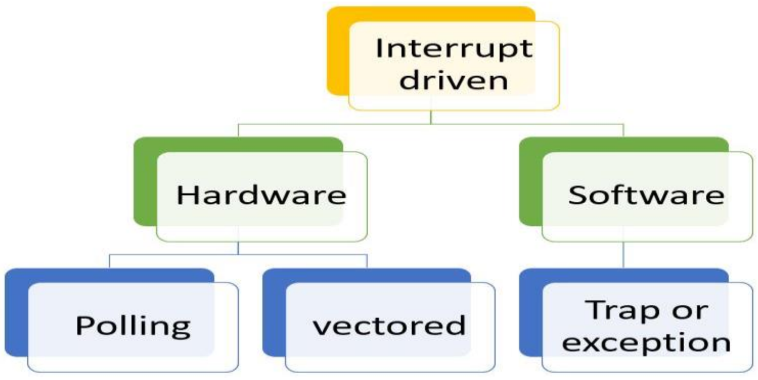


*Figure 2.6.1*

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 [10].

2. 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 [11].

## 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.

### 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 [12]. 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.

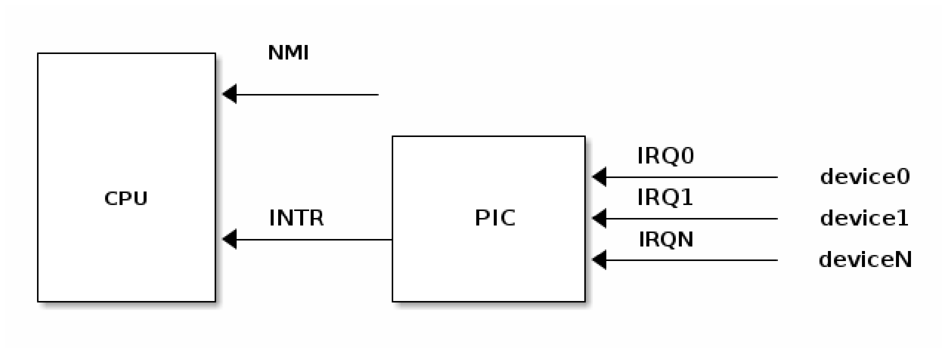


*Figure 2.6.2 - . Example diagram for PIC*

## 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 [13].

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 to assign 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.

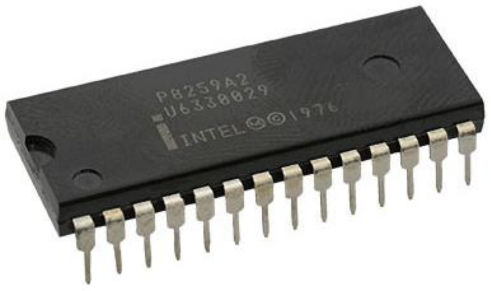
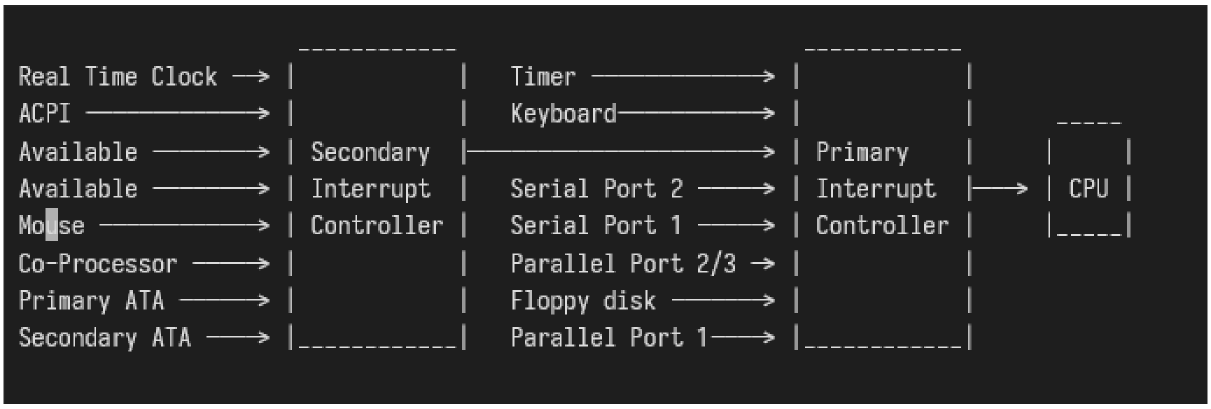


*Figure 2.6.3 - . intel 8259 PIC*

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



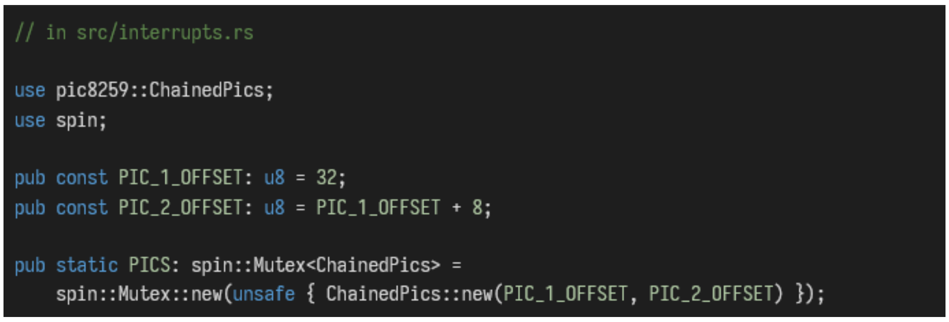
```
                     ____________                            ____________
Real Time Clock --> |            |   Timer -------------> |            |
ACPI -------------> |            |   Keyboard-----------> |            |      _____
Available --------> | Secondary  |----------------------> | Primary    |     |     |
Available --------> | Interrupt  |   Serial Port 2 -----> | Interrupt  |---> | CPU |
Mouse ------------> | Controller |   Serial Port 1 -----> | Controller |     |_____|
Co-Processor -----> |            |   Parallel Port 2/3 -> |            |
Primary ATA ------> |            |   Floppy disk -------> |            |
Secondary ATA ----> |____________|   Parallel Port 1----> |____________|
```

*Figure 2.6.4 - . Typical system that using 2 PIC'sc*

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.

## 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 [14, 15]. From Figure 2.6.5 can see how the primary/Secondary PIC redesigned  to range of 32–47.



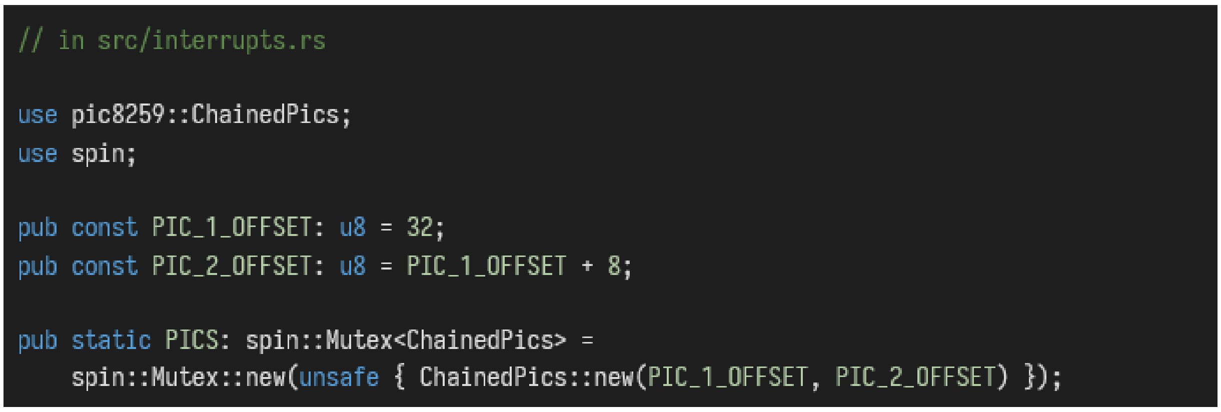
```
// in src/interrupts.rs

use pic8259::ChainedPics;
use spin;

pub const PIC_1_OFFSET: u8 = 32;
pub const PIC_2_OFFSET: u8 = PIC_1_OFFSET + 8;

pub static PICS: spin::Mutex<ChainedPics> =
    spin::Mutex::new(unsafe { ChainedPics::new(PIC_1_OFFSET, PIC_2_OFFSET) });
```

*Figure 2.6.5 - Example setting offset for primary/secondary PIC*

## 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 [16].

From Figure 2.6.6 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.

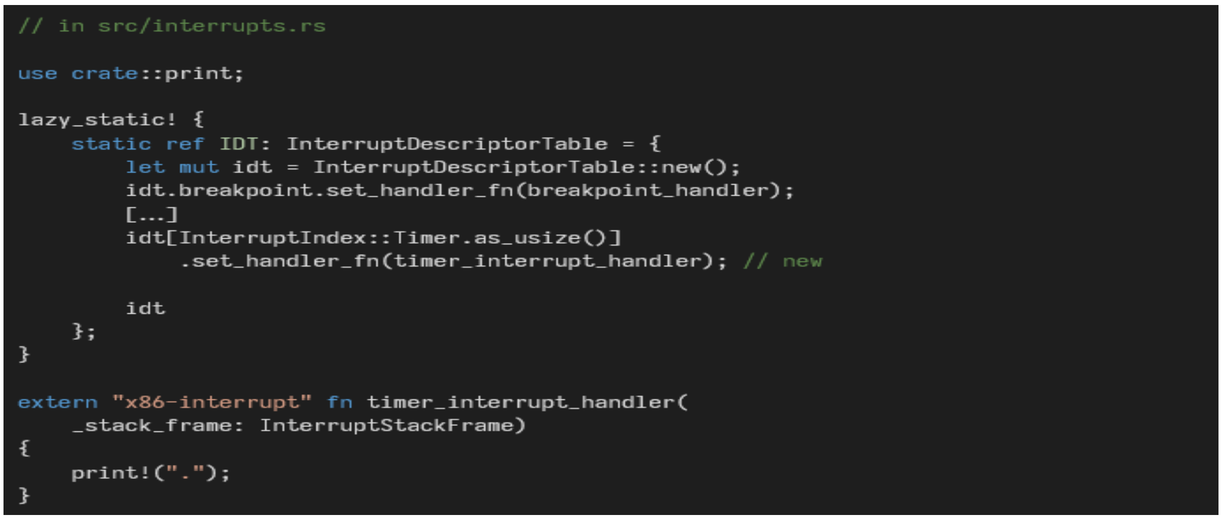


```
// in src/interrupts.rs

use pic8259::ChainedPics;
use spin;

pub const PIC_1_OFFSET: u8 = 32;
pub const PIC_2_OFFSET: u8 = PIC_1_OFFSET + 8;

pub static PICS: spin::Mutex<ChainedPics> =
    spin::Mutex::new(unsafe { ChainedPics::new(PIC_1_OFFSET, PIC_2_OFFSET) });
```

*Figure 2.6.6*

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 [17].

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.



```
// in src/interrupts.rs

use crate::print;

lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        idt.breakpoint.set_handler_fn(breakpoint_handler);
        [...]
        idt[InterruptIndex::Timer.as_usize()]
            .set_handler_fn(timer_interrupt_handler); // new

        idt
    };
}

extern "x86-interrupt" fn timer_interrupt_handler(
    _stack_frame: InterruptStackFrame)
{
    print!(".");
}
```

*Figure 2.6.7 - handle function for timer interrupt*

## 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 use 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. Even though exceptions like deadlock situation and race conditions could happen anytime.

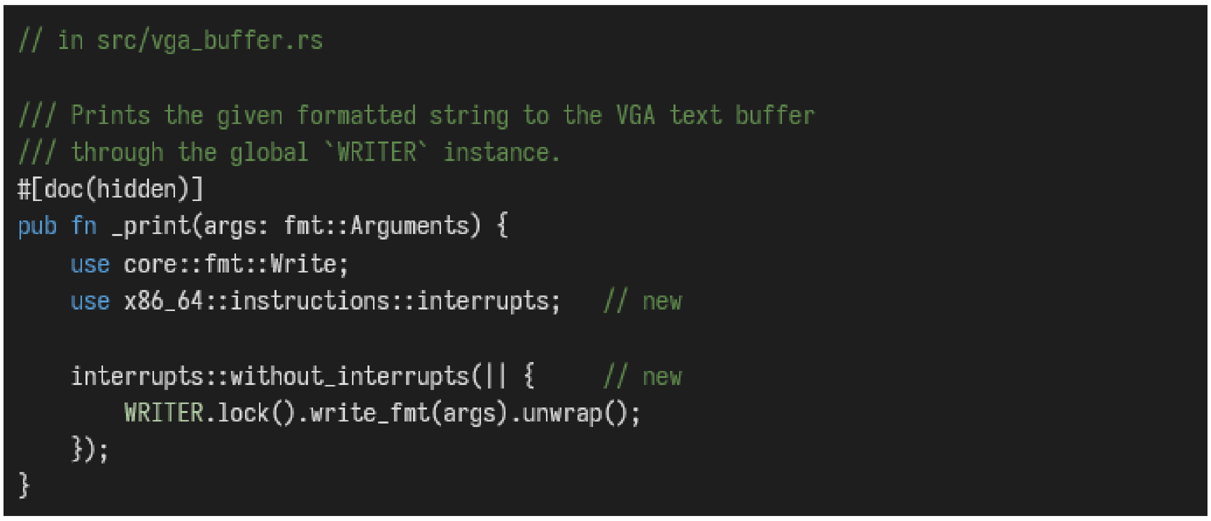
## Using Rust to oversee deadlocks & race conditions.

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.

### 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 [18].

In rust language can use **Without_interrrupts** function to ensure that the no interrupts happens while Mutex is locked.



```
// in src/vga_buffer.rs

/// Prints the given formatted string to the VGA text buffer
/// through the global `WRITER` instance.
#[doc(hidden)]
pub fn _print(args: fmt::Arguments) {
    use core::fmt::Write;
    use x86_64::instructions::interrupts;   // new

    interrupts::without_interrupts(|| {     // new
        WRITER.lock().write_fmt(args).unwrap();
    });
}
```

*Figure 2.6.8 - Without_interrrupts function example code*

By disabling interrupts for short time is the solution for the deadlocks meanwhile it’s 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.

### 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.

## 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 lifetime etc. To make CPU efficient must put CPU in halt until next interrupt signal generated by PIC.

### 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.

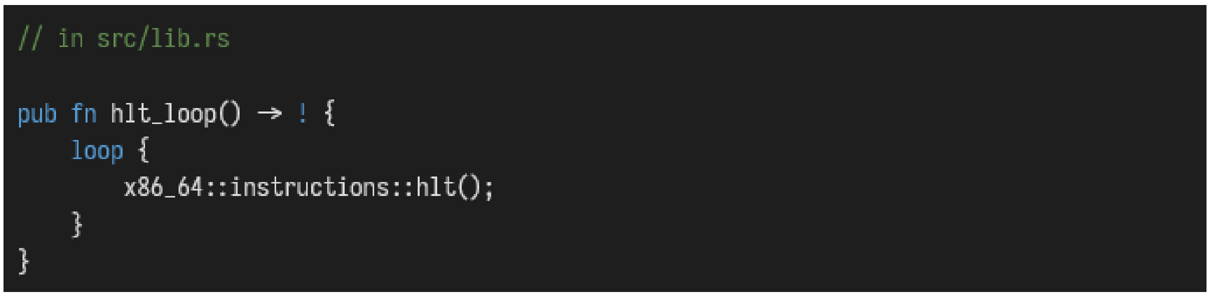

```
// in src/lib.rs

pub fn hlt_loop() -> ! {
    loop {
        x86_64::instructions::hlt();
    }
}
```

*Figure 2.6.9 - 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 won’t 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.

## 2.7 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 [19].

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

## 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.

- Segmentation.
- Paging.

### 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 [20].

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.

- 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 [21]

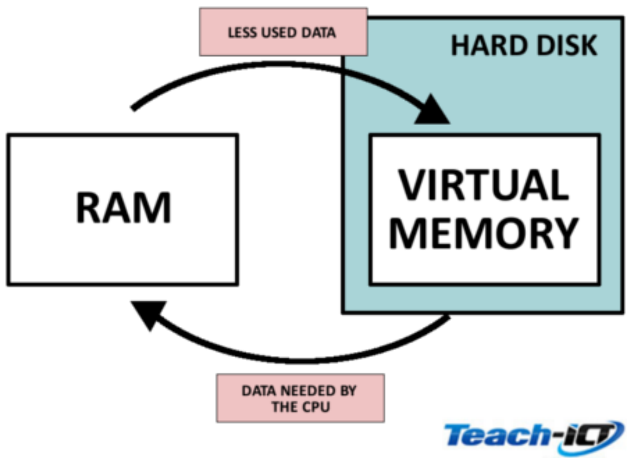


*Figure 2.7.1 - . virtual memory architecture*

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 can't be added sue to the small  block size. Because of these memory block always stays unused [22].

## Paging

The simple idea of paging is  divided 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 2.7.2 can see how the pages of virtual memory are mapped into frame in physical memory.

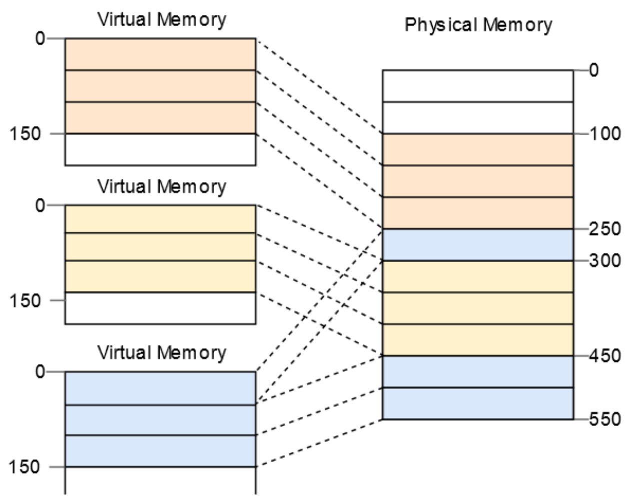


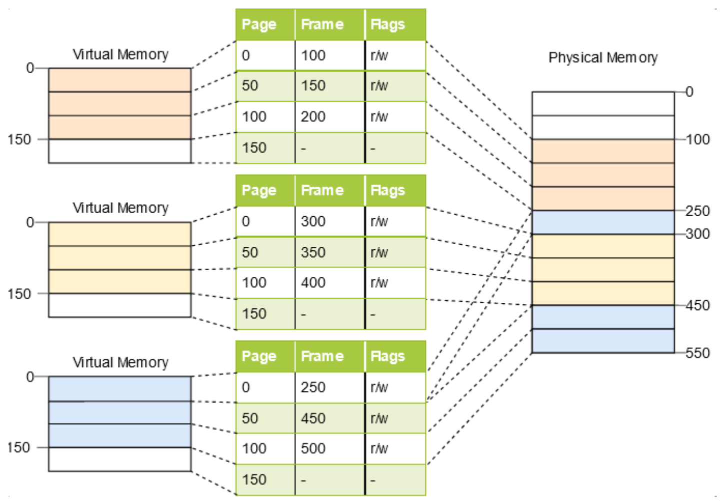
*Figure 2.7.2 - .  linked pages and frames example*

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 is known as internal fragmentation or hidden fragmentation.

- <u>Paging tables</u>

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.



| Page | Frame | Flags |
|---|---|---|
| 0 | 100 | r/w |
| 50 | 150 | r/w |
| 100 | 200 | r/w |
| 150 | - | - |

| Page | Frame | Flags |
|---|---|---|
| 0 | 300 | r/w |
| 50 | 350 | r/w |
| 100 | 400 | r/w |
| 150 | - | - |

| Page | Frame | Flags |
|---|---|---|
| 0 | 250 | r/w |
| 50 | 450 | r/w |
| 100 | 500 | r/w |
| 150 | - | - |

*Figure 2.7.3 - 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.

## 2.8 Paging Implementation

### 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.

#### 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.

- Identity mapping.
- Map the complete physical memory.
- Recursive page mapping.

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

- <u>Identity mapping</u>

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 2.8.1 can see how the actual identity mapping done [23].

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.

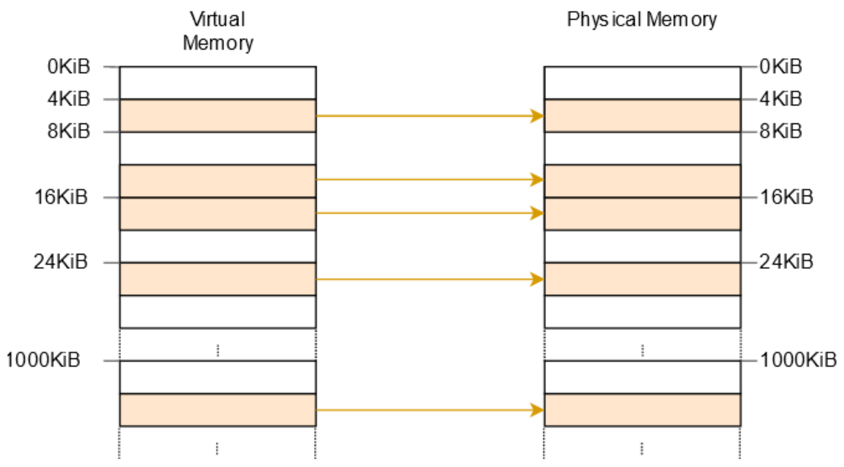


*Figure 2.8.1 - Identity mapping example*

- 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 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.

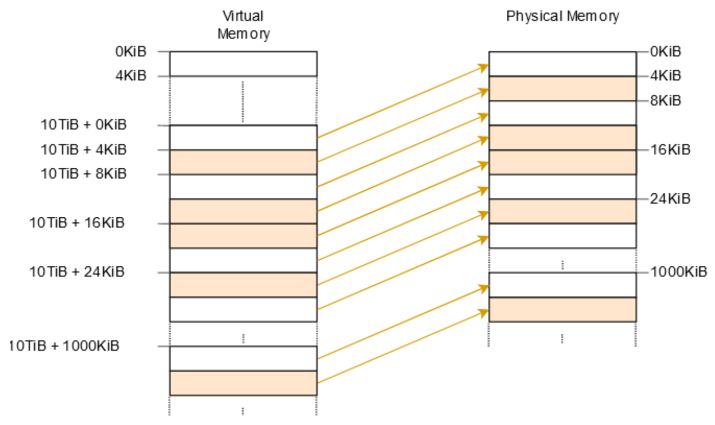
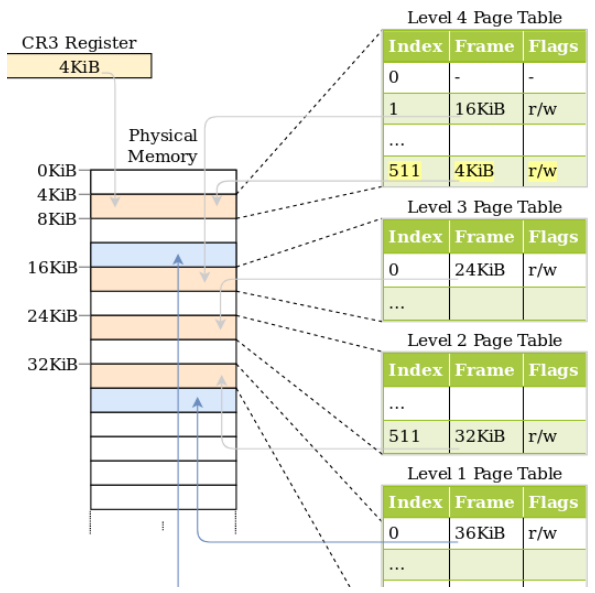


*Figure 2.8.2 - complete mapping example*

- 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.



Level 4 Page Table

| Index | Frame | Flags |
|---|---|---|
| 0 | - | - |
| 1 | 16KiB | r/w |
| ... | | |
| 511 | 4KiB | r/w |

Level 3 Page Table

| Index | Frame | Flags |
|---|---|---|
| 0 | 24KiB | r/w |
| ... | | |

Level 2 Page Table

| Index | Frame | Flags |
|---|---|---|
| ... | | |
| 511 | 32KiB | r/w |

Level 1 Page Table

| Index | Frame | Flags |
|---|---|---|
| 0 | 36KiB | r/w |
| ... | | |

*Figure 2.8.3 - Recursive mapping example*

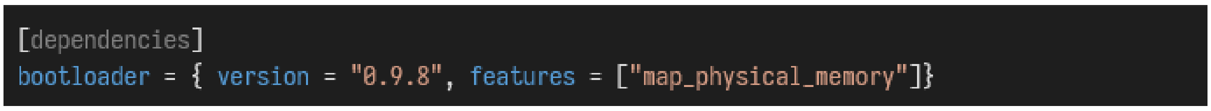
## Implementing paging using RUST

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.

### 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 [24].



```
[dependencies]
bootloader = { version = "0.9.8", features = ["map_physical_memory"]}
```

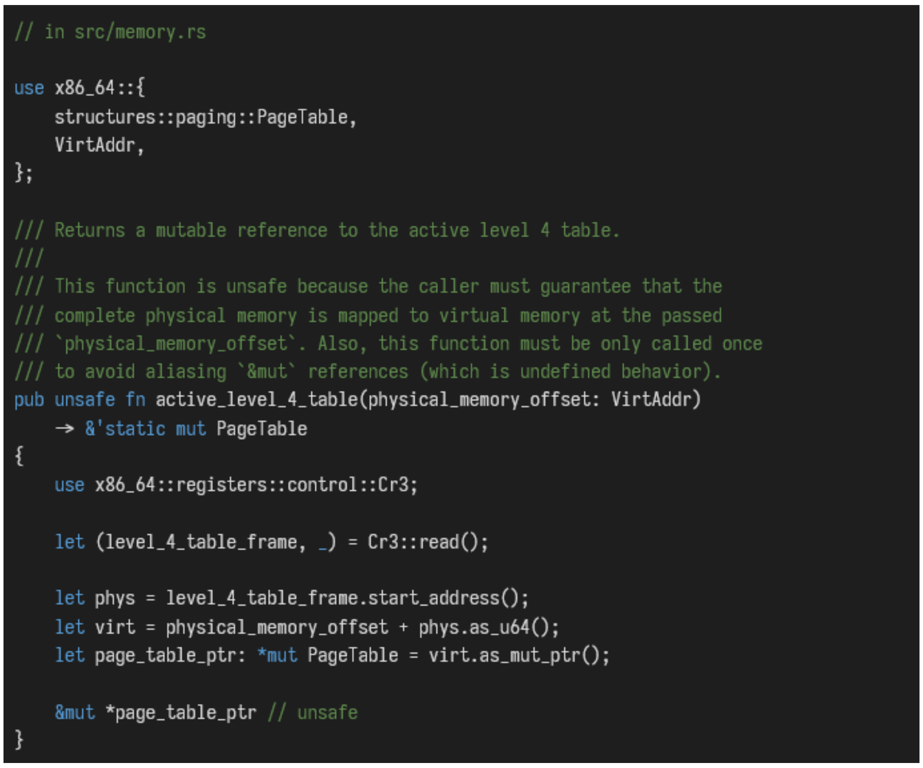
*Figure 2.8.4 - enabling bootloader function example*

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

### 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.



```rust
// in src/memory.rs

use x86_64::{
    structures::paging::PageTable,
    VirtAddr,
};

/// Returns a mutable reference to the active level 4 table.
///
/// This function is unsafe because the caller must guarantee that the
/// complete physical memory is mapped to virtual memory at the passed
/// `physical_memory_offset`. Also, this function must be only called once
/// to avoid aliasing `&mut` references (which is undefined behavior).
pub unsafe fn active_level_4_table(physical_memory_offset: VirtAddr)
    -> &'static mut PageTable
{
    use x86_64::registers::control::Cr3;

    let (level_4_table_frame, _) = Cr3::read();

    let phys = level_4_table_frame.start_address();
    let virt = physical_memory_offset + phys.as_u64();
    let page_table_ptr: *mut PageTable = virt.as_mut_ptr();

    &mut *page_table_ptr // unsafe
}
```

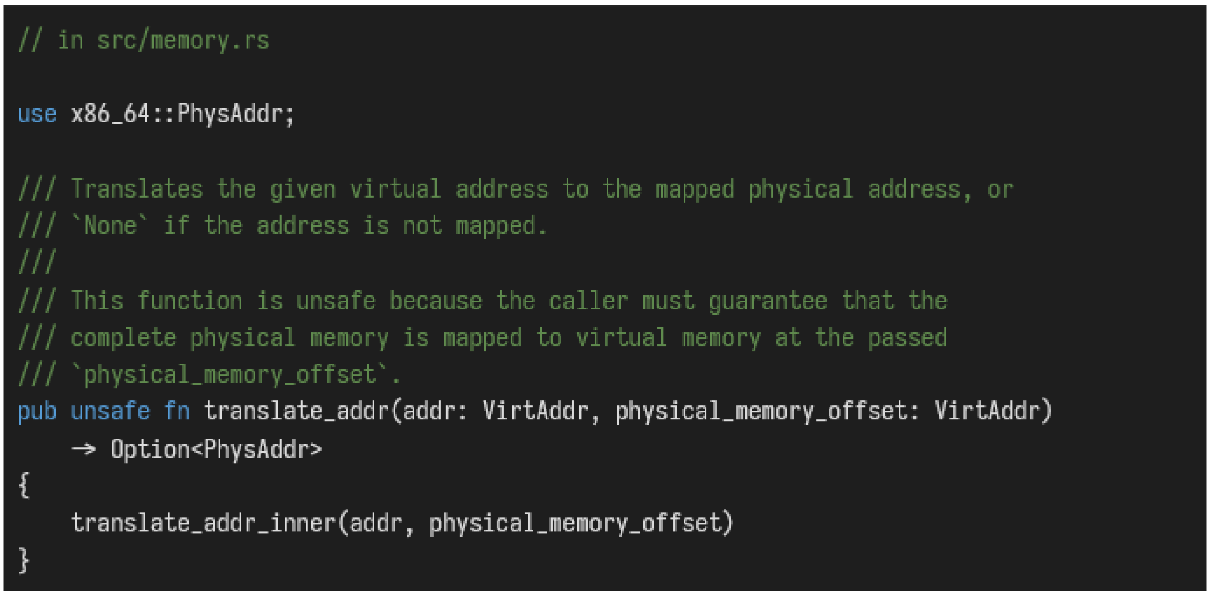
*Figure 2.8.5 - active_level_4_table example code*

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.

### Translating addresses

Afterward creating all above functions, we must 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.



```
// in src/memory.rs

use x86_64::PhysAddr;

/// Translates the given virtual address to the mapped physical address, or
/// `None` if the address is not mapped.
///
/// This function is unsafe because the caller must guarantee that the
/// complete physical memory is mapped to virtual memory at the passed
/// `physical_memory_offset`.
pub unsafe fn translate_addr(addr: VirtAddr, physical_memory_offset: VirtAddr)
    -> Option<PhysAddr>
{
    translate_addr_inner(addr, physical_memory_offset)
}
```

*Figure 2.8.6 - 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.

## 2.9 Heap Allocation

The most adaptable allocation strategy is heap allocation. Memory can be allocated and deallocated at any moment, when it is required, based on the needs of the user. Memory is dynamically allocated to variables using heap allocation, which then claims the memory back when the variables are no longer in use [25].

In kernel there are two types of variables, and they are Local variables and Static Variables

## Local variables

Stack data structure (call stack) is which supports push and pop operations that the local variables are stored. Local variables, return address and the parameters of the called function of each function entry were pushed by the compiler:

The call stack is displayed in the example above after the outer function has called the inner function. The local variables of outer first are visible in the call stack. The parameter 1 and the function's return address were pushed on the inner call. Then the inner gets the control which pushed its local variables. When the inner function returns, its section of the call stack is popped, leaving only the outer function's local variables:

## Static variables

Static variables are kept in a dedicated area of memory that is not part of the stack. The linker assigns this memory location at compile time, and the executable is coded using this information. Statics have a static lifetime and can always be linked from local variables because they last the entire duration of the program:

In the example above, the call stack is cleared when the inner function returns. The [25] reference remains valid after the return because the static variables are stored in a different memory region that is never deleted.

## Dynamic Memory

Because of these static and local variables' limitations like:

- Both variables having a fixed size and the dynamically growing elements cannot store as a collection
- Local variables are live on the call stack and when the surrounding function returns, they were destroyed
- Static variables are living until e=the end of the program so when the memory no longer needed there's no way to reuse or reclaim

So, because of these limitations programming languages uses heap for storing variables as a third memory. The allocate and deallocate functions of the heap enable dynamic memory allocation at runtime. A free memory space of the given size is returned by the allocate function which can be used to hold a variable. When the deallocate, function is used with a reference to the variable, the variable remains in existence until it is released [26].

## Common Errors

There are two typical bug kinds with more serious repercussions, aside from memory leaks, which are regrettable but don't leave the software open to attackers.

- Use-after-free vulnerability – after calling deallocate on a variable accidentally continue to use it
- Double-free vulnerability – accidentally freeing a variable twice

Rust makes use of a concept called ownership to ensure the accuracy of dynamic memory operations at build time. As a result, there isn't any performance overhead and the above-mentioned vulnerabilities can be avoided without the need of trash collection. Another benefit of this strategy is that, like with C or C++, the programmer retains fine-grained control over the utilization of dynamic memory.

## Allocations in Rust

The Rust standard library offers abstraction types which implicitly call allocation and deallocate rather than allowing the programmer to individually invoke these functions. Box, the abstraction for a heap-allocated value, is the most crucial type. It offers a constructor function called Box:new that accepts a value, calls allocate with the size of the value, and then transfers the value to the heap's newly created slot. The Box type implements the Drop trait to call deallocate when it exits scope in order to release the heap memory once more.

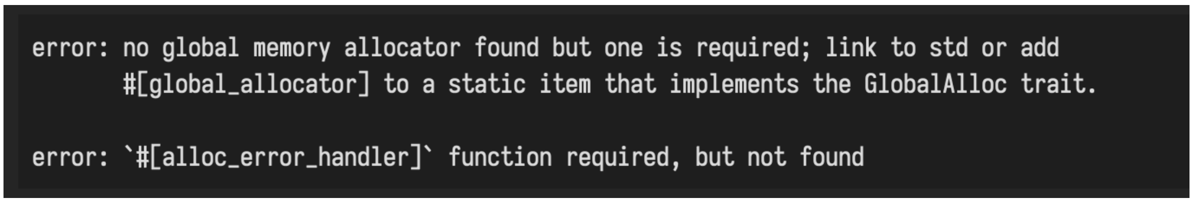
So, the rust ownership assigns lifetime abstracts for the references. Also, like garbage collected languages such as Python or Java do, they also provide total memory safety, preventing use-after-free problems. It also ensures thread safety, making it even safer in multi-threaded programs than those languages. Most crucially, there isn't any runtime overhead compared to manually implemented memory management in C because all these tests take place at compile time [26].

## The Allocator Interface

Adding a dependence on the built-in alloc crate is the first step in the heap allocator implementation process. This is a subset of a standard library which also includes the collection and allocation types, much like the core crate.

We don't have to change the Cargo.toml, in contrast to typical dependencies. The alloc crate is included in the standard library with the Rust compiler, therefore the compiler is already aware of the crate. We tell the compiler to try to include it by adding the extern crate declaration.

Then the compiler includes the alloc crate in our kernel. When trying to compile this occurs two errors



```
error: no global memory allocator found but one is required; link to std or add
       #[global_allocator] to a static item that implements the GlobalAlloc trait.

error: `#[alloc_error_handler]` function required, but not found
```

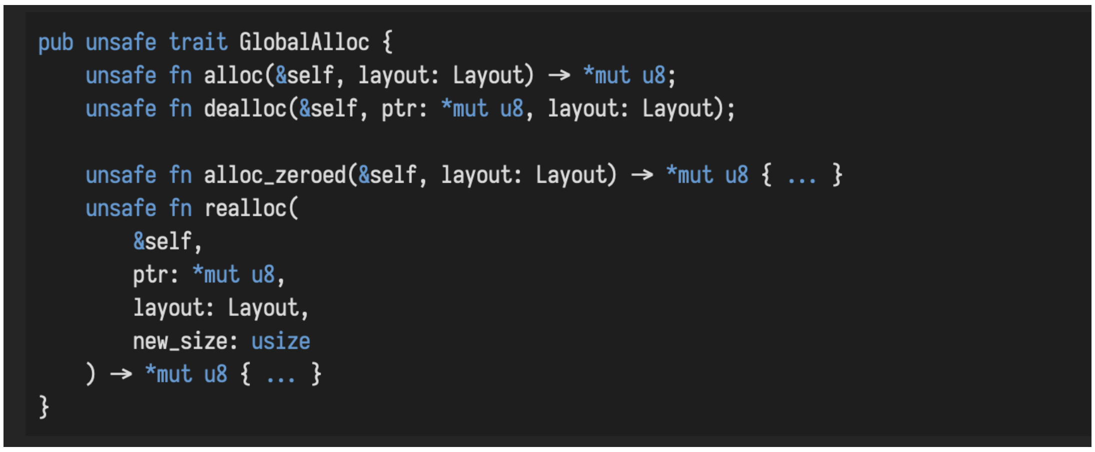
*Figure 2.9.1*

The heap allocator, an object that offers the allocation and deallocate functions, is required by the alloc crate, which leads to the first problem. The GlobalAlloc trait, which is referenced in the error message, in Rust describes heap allocators. The #[global allocator] attribute should be assigned to a static variable which implements the GlobalAlloc trait in order to define the heap allocator again for crate.

The second error happens when calls to allocate fail, most typically when no more memory is available. The #[alloc error handler] method ensures that our software can respond to this situation.

## The GlobalAlloc Trait

The GlobalAlloc trait specifies the functionalities that must be provided by a heap allocator. The trait is unique in that it is practically never explicitly utilized by the programmer. When you use the collection and allocation types of alloc, the compiler may dynamically include the required calls towards the trait methods [27].



```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);

    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 { ... }
    unsafe fn realloc(
        &self,
        ptr: *mut u8,
        layout: Layout,
        new_size: usize
    ) -> *mut u8 { ... }
}
```

*Figure 2.9.2*

This defines alloc and dealloc as the required methods and the alloc_zeroed and realloc two methods are additionally defied by the trait with default implementation.

## A DummyAllocator

Because the struct does not require any fields, we build it as a zero-sized type. As previously stated, alloc always returns a null pointer, indicating an allocation mistake. Because the allocator never returns memory, a call to dealloc shouldn't ever be made. As a result, in the dealloc method, we simply panic. We are not required to provide implementations for such alloc zeroed and realloc procedures because they contain default implementations.

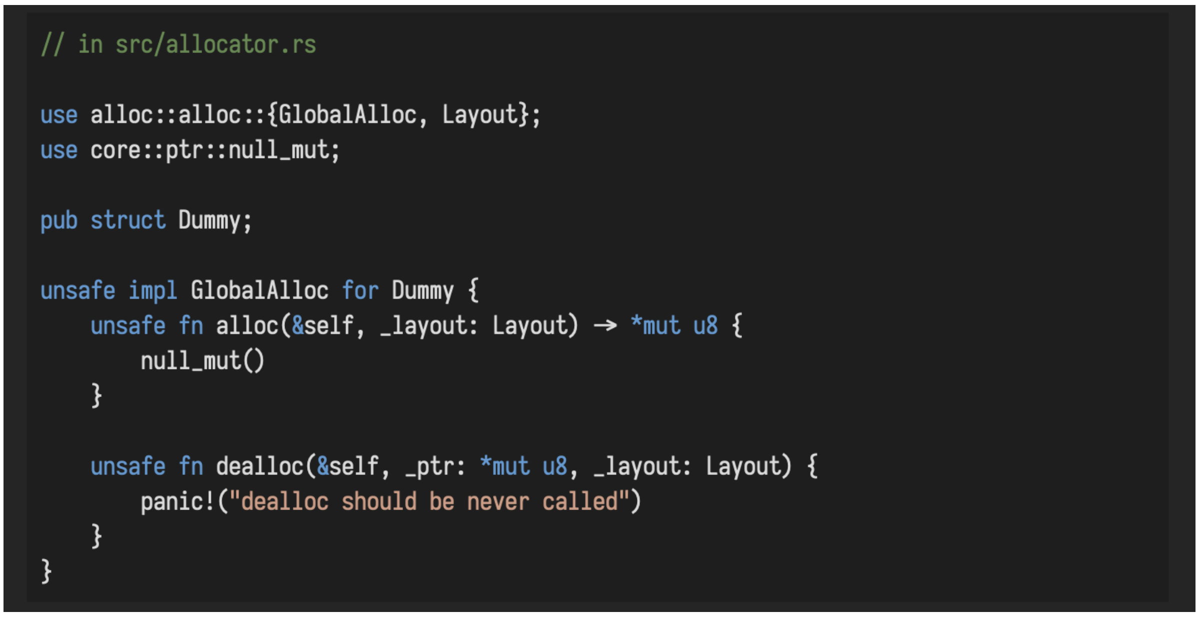

```rust
// in src/allocator.rs

use alloc::alloc::{GlobalAlloc, Layout};
use core::ptr::null_mut;

pub struct Dummy;

unsafe impl GlobalAlloc for Dummy {
    unsafe fn alloc(&self, _layout: Layout) -> *mut u8 {
        null_mut()
    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        panic!("dealloc should be never called")
    }
}
```

*Figure 2.9.3*

## The #[global_allocator] Attribute

The #[global allocator] parameter specifies which allocator instance the Rust compiler should use for the global heap allocator. The characteristic applies exclusively to statics that implement the GlobalAlloc trait. Let us make a Dummy allocator instance the global allocator:

## The #[alloc_error_handler] Attribute

The alloc function, as we discovered while studying the GlobalAlloc trait, can communicate an allocation mistake by returning a null pointer. The issue is how the Rust runtime should handle quite an allocation failure. The #[alloc error handler] attribute comes into play here. It defines a function that is invoked when there is an allocation error, like how our panic handler is called when there is a panic.

## Creating a Kernel Heap

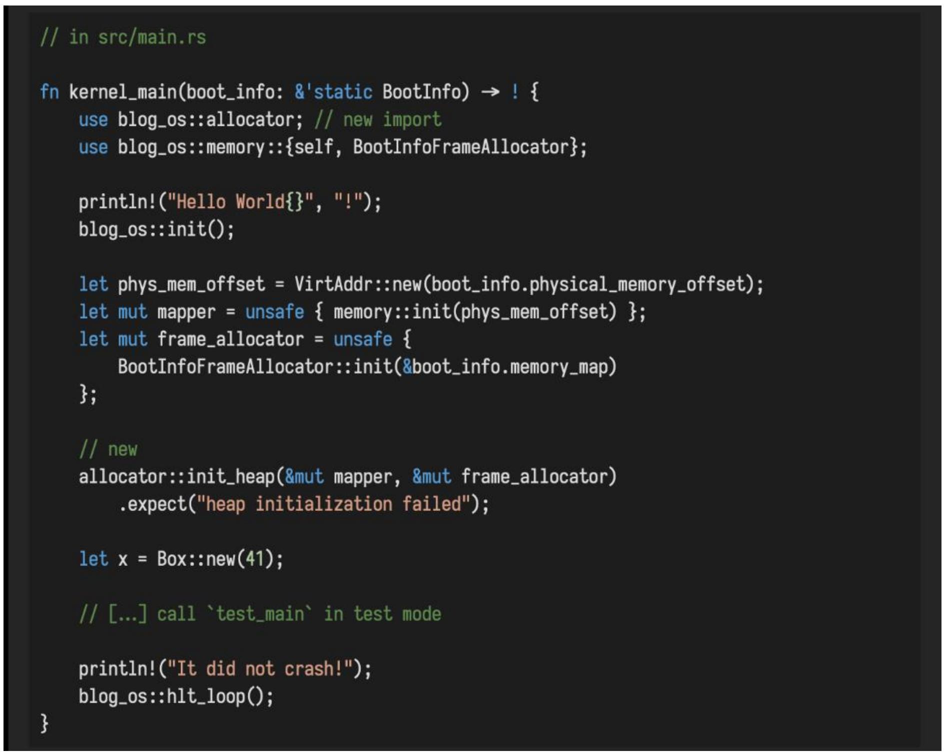
The very first step is to create a heap virtual memory region. We can utilize any virtual address range we want, as much as it is not already in use for another memory region. To make it easier to identify a heap pointer later, let's describe it as the memory beginning at address 0x 4444 4444 0000:

The function accepts mutable pointers to Mapper and FrameAllocator instances, both of which are restricted to 4 KiB pages by the Size4KiB generic parameter. The function returns a Result as the success variation as well as a MapToError even as failure variant, based on the error type provided by the Mapper:map to method. Because the map to method is the primary source of mistakes in this function, repeating the failure type makes reasonable.

The implementation is two parts:

- Creating the page range
- Mapping the pages

Calling the function from the kernel_main is the final step.



```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) → ! {
    use blog_os::allocator; // new import
    use blog_os::memory::{self, BootInfoFrameAllocator};

    println!("Hello World{}", "!");
    blog_os::init();

    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = unsafe {
        BootInfoFrameAllocator::init(&boot_info.memory_map)
    };

    // new
    allocator::init_heap(&mut mapper, &mut frame_allocator)
        .expect("heap initialization failed");

    let x = Box::new(41);

    // [...] call `test_main` in test mode

    println!("It did not crash!");
    blog_os::hlt_loop();
}
```

*Figure 2.9.4*

## 2.10 Allocator Designs

Previously added basic support for heap allocations to our kernel by creating a new memory region in the page tables and manage the memory by using linked_list_allocator crate. In allocator designs creating own heap allocator from scratch and discuss different allocator designs and these help to improve performance.

### Design Goals

An allocator's task is to control the heap memory that is accessible. It must return unused memory on alloc calls and maintain track of memory freed by dealloc in order for it to be reused. epesially , it should never allocate memory that is currently in use someplace else, as this would result in erratic behavior and there are many secondary designs goal. Those requirements would make finding good allocators difficult [26].

Let’s see three of possible kernel allocator designs:

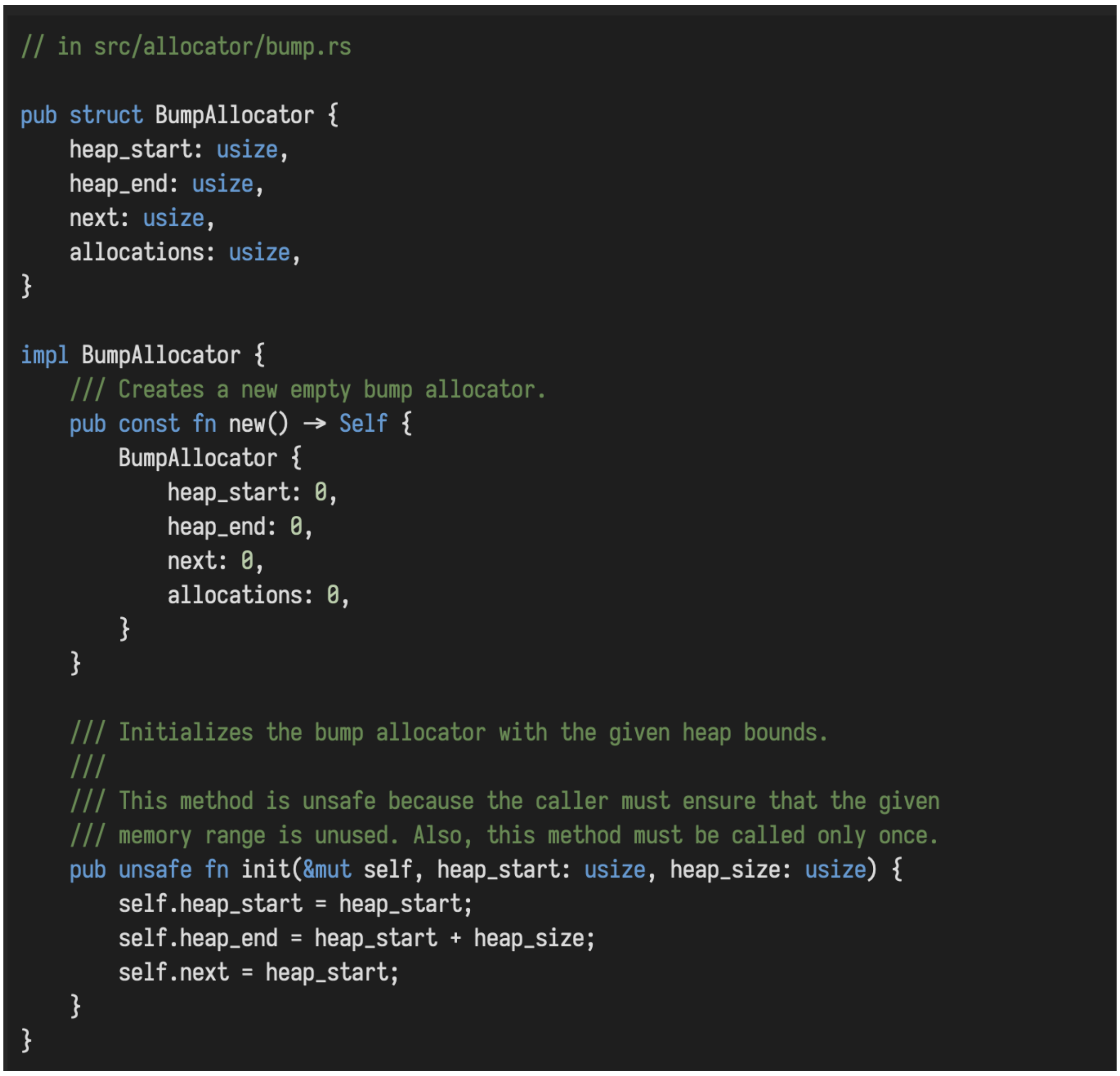
### Bump Allocator

This initializes in a linear fashion and merely maintains track of the amount of granted bytes and allocations. The concept behind a bump allocator is to allocate memory linearly by increasing ("bumping") a next variable that links to the beginning of the unused memory. At the start, next will be equal to the heap's start address.

#### Implementation

The heap start and heap end fields keep track of the heap memory region's lower and higher bounds. The caller must guarantee that these addresses are correct, else the allocator will return invalid memory. As a result, the init function must be hazardous to call.

For keep this interface identical towards the allocator supplied by the linked list allocator crate, we elected to construct a separate init method rather than executing the initialization straight in new. This allows the allocators to be altered without requiring any extra code changes.



```rust
// in src/allocator/bump.rs

pub struct BumpAllocator {
    heap_start: usize,
    heap_end: usize,
    next: usize,
    allocations: usize,
}

impl BumpAllocator {
    /// Creates a new empty bump allocator.
    pub const fn new() -> Self {
        BumpAllocator {
            heap_start: 0,
            heap_end: 0,
            next: 0,
            allocations: 0,
        }
    }

    /// Initializes the bump allocator with the given heap bounds.
    ///
    /// This method is unsafe because the caller must ensure that the given
    /// memory range is unused. Also, this method must be called only once.
    pub unsafe fn init(&mut self, heap_start: usize, heap_size: usize) {
        self.heap_start = heap_start;
        self.heap_end = heap_start + heap_size;
        self.next = heap_start;
    }
}
```

*Figure 2.10.1*

- Implementing GlobalAlloc - Because no limits checks or alignment modifications are performed, this approach is not yet safe. Because there are some errors with alloc and dealloc methods. The problem is bump allocator’s essential principles are the updating next on every allocation.

- GlobalAlloc and Mutability. – The #[global allocator] attribute is added to a static that defines the GlobalAlloc trait. Because static variables in Rust are immutable, there isn't any way to call a function that takes &mut self on the static allocator. As a result, all GlobalAlloc methods only accept an immutable &self-reference.

- A Locked Wrapper Type. - For the bump allocator, implementing the GlobalAlloc trait with the aid of the spin::Mutex wrapper type. The idea is to implement the trait for the wrapped spin::MutexBumpAllocator> type rather than the BumpAllocator directly. But this doesn’t word because trait implementations for types declared in other crates are not permitted by the Rust compiler

- Implementation for Locked<BumpAllocator> - h this is using for implement GlobalAlloc for the bump allocator.

Performance- The main advantage with bump allocation is its speed. In contrast to other allocator designs (see below), which must actively search for a suitable memory block and conduct different accounting operations on alloc and dealloc, a bump allocator can really be optimized to only a few assembly instructions. As a result, bump allocators are useful for optimizing allocation performance, such as when developing a virtual DOM library [26].

## Linked List Allocator

When implementing allocators, a typical approach for keeping a record of an arbitrary number of available memory areas is to use these areas as backing storage. This capitalizes on the fact that the regions are all still mapped to a virtual address and supported by a physical frame, however the stored information is no longer required. We can maintain track of an infinite number of released regions by storing information about the freed region within the region itself.

### Implementation

In implementation the process is creating a own simple LinkedListAllocator type for approach for keep track of freed memory regions.

Performance - In comparison to the bump allocator, the linked list allocator is significantly better suited as a general-purpose allocator, owing to its ability to reuse freed memory. It does, however, have certain downsides. Some are caused solely by the basic implementation, however there are also inherent flaws in the allocator concept itself.

In this category, the linked list allocator performs significantly worse. The issue is that an allocation query may need to traverse the entire linked list till the finds an appropriate block. But performance is isn't issue in this case [26].

## Fixed-Size Block Allocator

Rather of allocating exactly the desired amount of memory, we establish a tiny number of blocks and round up each allocation until the next block size. For example, with block sizes of 16, 64, and 512 bytes, a 4-byte allocation would result in a 16-byte block, a 48-byte allocation in a 64-byte block, and a 128-byte allocation in a 512-byte block.

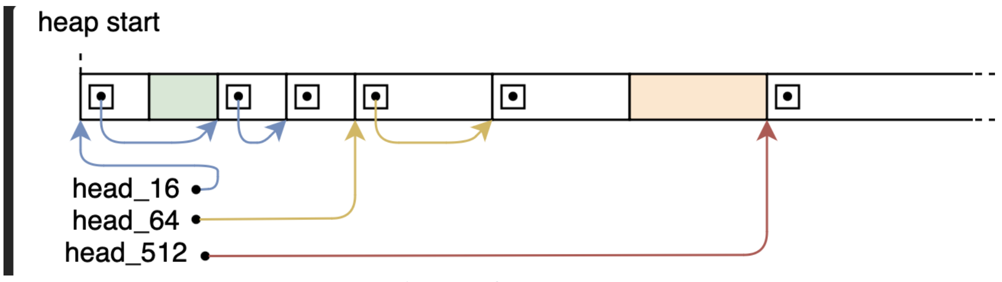


*Figure 2.10.2*

There are three head pointers head_16 , head_64 and head_512

**Block Sizes and Wasted Memory**

Rounding up based on the block sizes , a lot of memory can be lost. In the worst-case half of allocation size and in the average cases quarter of the allocation size of memory waste limit can be limit.

### Fallback Allocator

Given the rarity of large allocations (>2 KB), particularly in operating system kernels, it may make sense to use a specialized allocator for these allocations. Because only a few allocations of such magnitude are planned, the linked list would remain short and the (de)allocations would remain quite fast.

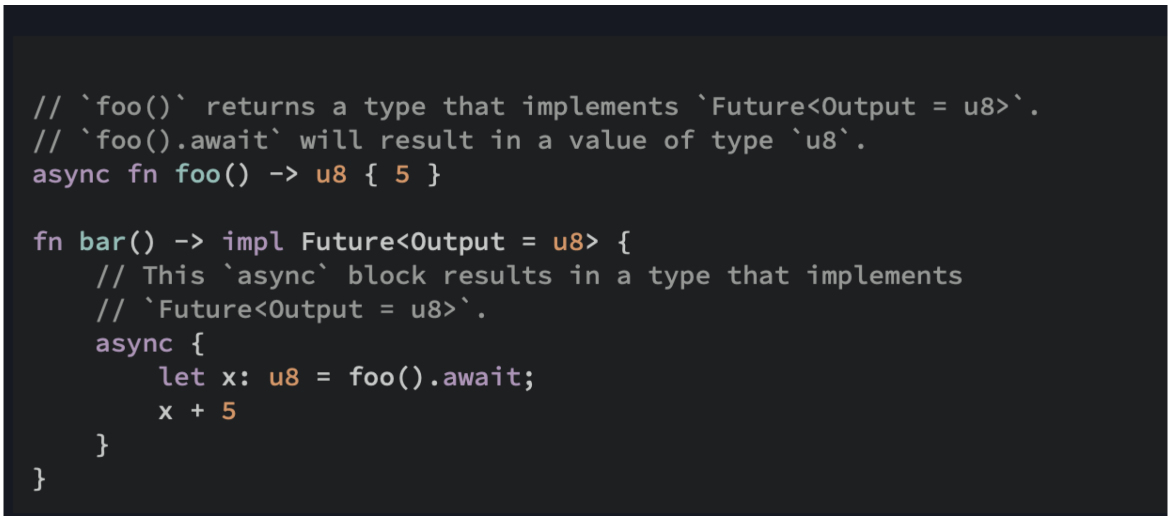
### Creating new Blocks

When fulfilling all allocation requests some point, the linked block size becomes empty. At this stage, there are two options for creating new unused blocks of a specified size to satisfy an allocation request:

## 2.11 Async/Await

Async/await are Rust language extensions that allow you to cede control of the current thread instead of blocking, allowing other code to run while waiting for an operation to complete.

Async can be used in two ways: async fn and async blocks. Each one returns a result that satisfies the Future trait:



```rust
// `foo()` returns a type that implements `Future<Output = u8>`.
// `foo().await` will result in a value of type `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type that implements
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

*Figure 2.11.1*

As we learned in the first chapter, async bodies and other futures are inactive until they are run. The most frequent method for running a Future is to await it. When await is used on a Future, it attempts to complete it. If the Future is blocked, control of the current thread is relinquished. When more progress can be achieved, the executor can grab up the Future and begin running it, allowing the await to resolve.

## async Lifetimes

This means that an async fn's future must be awaited while its non-'static arguments are still valid. This is not an issue in the common scenario of awaiting the future right after running the function (as in foo(&x).await). This may be an issue if you are storing the future or passing it to another job or thread.

One typical solution for converting an async fn with references-as-arguments into a static future is to wrap the arguments in an async block.

## async move

As with conventional closures, async blocks and closures support the move keyword. An async move block will assume ownership of the variables it references, allowing it to outlive the current scope while denying other programs the opportunity to share those variables:

## Awaiting on a Multithreaded Executor

Because a Future can move between threads when utilizing a multithreaded Future executor, all variables used in async bodies must be able to do the same. await has the possibility to switch to a new thread.

This means that using Rc, & RefCell, or any other types that do not implement the Send trait, including references to types that do not implement the Sync trait, is not safe.

(Caveat: these types can be used as long as they are not in scope during a call to await.)

Similarly, holding a standard non-futures-aware lock across an await may cause the thread pool to lock up: one job may take out a lock, await, and surrender to the executor, allowing another task to attempt to take the lock and produce a deadlock. To avoid this, use the Mutex from futures: lock instead of the Mutex from std::sync [28].

### Futures

A future reflects a resource which is not yet accessible. This might be an integer calculated by another process or a file obtained from the network. Instead of waiting until the value is ready, futures allow you to continue execution until the value is required.

### Working with Futures

Another more accurate method would be to pause the current thread until the future becomes accessible. This is,  obviously, only feasible if there are threads, therefore this method is not suitable for the kernel, at certainly still not. But on systems that enable blocking, it is frequently undesirable since it converts an asynchronous operation into a synchronous activity, so limiting the potential performance gains.

# Conclusion

Rust is a static compiled language with a comprehensive type system and provenance concept that is both fast and memory economical. It may be used to operate performance-critical services while ensuring memory and thread safety, giving developers the ability to analyze at build time. Additionally, Rust offers excellent documentation and a user-friendly compiler with high-end tools like as embedded package management and a multi-editor with capabilities such as type checking and auto-completion. The Rust community is rather effective, and newer models and technologies to improve Rust are constantly published. Rust is predicted to be popular in the next years due to its ability and reputation for building safe systems. Because of its security, efficiency, and productivity, the development community will continue to value Rust.

# References

[1] G. Dreimanis, "Introduction to Rust," 19 August 2020. [Online]. Available: https://serokell.io/blog/rust-guide#data-ownership-model. [Accessed 4 November 2022].

[2] P. Oppermann, "A Freestanding Rust Binary," 10 February 2018. [Online]. Available: https://os.phil-opp.com/freestanding-rust-binary/. [Accessed 4 November 2022].

[3] U. Bindal, "A Freestanding Rust Binary," 8 December 2021. [Online]. Available: https://usethesource.hashnode.dev/a-freestanding-rust-binary. [Accessed 4 November 2022].

[4] P. Oppermann, "A Minimal Rust Kernel," 10 February 2018. [Online]. Available: https://os.phil-opp.com/minimal-rust-kernel/. [Accessed 6 November 2022].

[5] "Test Organization," [Online]. Available: https://doc.rust-lang.org/book/ch11-03-test-organization.html. [Accessed 29 October 2022].

[6] P. Oppermann, "CPU Exceptions," 17 January 2018. [Online]. Available: https://os.phil-opp.com/cpu-exceptions/. [Accessed 1 November 2022].

[7] P. Oppermann, "Double Faults," 18 January 2018. [Online]. Available: https://os.phil-opp.com/double-fault-exceptions/. [Accessed 1 November 2022].

[8] "chormium," [Online]. Available: https://www.chromium.org/Home.

[9] "Techtarget," [Online]. Available: https://www.techtarget.com/whatis/definition/interrupt.

[10] "geeks for geeks," [Online]. Available: https://www.geeksforgeeks.org/difference-between-hardware-interrupt-and-software-interrupt/.

[11] "quora," [Online]. Available: https://www.quora.com/What-is-an-example-of-software-interrupt.

[12] "wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Polling_(computer_science).

[13] "wikipeedia," [Online]. Available: https://en.wikipedia.org/wiki/Programmable_interrupt_controller.

[14] "OSDEV," [Online]. Available: https://wiki.osdev.org/8259_PIC.

[15] "docs.rs," [Online]. Available: https://docs.rs/crate/pic8259/0.10.1/source/src/lib.rs.

[16] "docs.rs," [Online]. Available: https://docs.rs/x86_64/0.9.6/x86_64/instructions/interrupts/fn.enable_interrupts_and_hlt.html.

[17] "carnegie mellon," [Online]. Available: https://www.cs.cmu.edu/~410/doc/triple.html.

[18] "tutorial sport," [Online]. Available: https://www.tutorialspoint.com/mutex-vs-semaphore.

[19] "tech target," [Online]. Available: https://www.techtarget.com/whatis/definition/memory-management.

[20] "javapoint," [Online]. Available: https://www.javatpoint.com/os-segmentation.

[21] "techno monitor," [Online]. Available: https://techmonitor.ai/what-is/what-is-virtual-memory-4929986.

[22] "byjus.com," [Online]. Available: https://byjus.com/gate/fragmentation-in-os-notes/.

[23] "osdev," [Online]. Available: https://wiki.osdev.org/Identity_Paging.

[24] "ionos," [Online]. Available: https://www.ionos.com/digitalguide/server/configuration/what-is-a-bootloader/.

[25] Ginni, 8 Nov 2021. [Online]. Available: https://www.tutorialspoint.com/what-is-heap-allocation.

[26] 2020. [Online]. Available: https://os.phil-opp.com/async-await/.

[27] "rust," 2020. [Online]. Available: https://doc.rust-lang.org/std/alloc/trait.Allocator.html.

[28] https://os.phil-opp.com/async-await/. [Online].

## Individual Contribution

| | Student Name | Works done |
|---|---|---|
| 1 | Dissanayake W.P.D.B (Leader) | • Made Assumptions.<br>• Made the introduction.<br>• Wrote "What is rust and why rust" chapter .<br>• Wrote about freestanding rust binary<br>• Wrote about minimal rust kernel<br>• Created the finalized Report. |
| 2 | Zakey M.S.M. A | • Made Assumptions.<br>• Wrote about hardware interrupts.<br>• Wrote introduction to paging<br>• Wrote about paging implementation<br>• Checked the finalized document |
| 3 | Dilhara W. M. A. | • Made Assumptions.<br>• Wrote about heap allocation.<br>• Wrote about allocator designs.<br>• Wrote about async/ await section.<br>• Checked the finalized document. |
| 4 | Pemachandra T.H.R.T. | • Made Assumptions.<br>• Wrote about testing.<br>• Wrote about CPU exceptions.<br>• Wrote about double faults.<br>• Wrote the conclusion<br>• Checked the finalized document |