Sri Lanka Institute of Information Technology

Specialized in Cyber Security

Year 2, Semester 1

Group 02.01

IE2032 – Secure Operating System

Group Assignment 1

Group Details

Registration No	Name	Email
IT21170720	K.D.S.P. Jayawikrama	it21170720@my.sliit.lk
IT21176388	H.C.K. Ariyarathna	it21176388@my.sliit.lk
IT21184758	Liyanage P.P.	it21184758@my.sliit.lk
IT21180316	Samaranayake Y.C	it21180316@my.sliit.lk

Terms of Reference

A report submitted fulfils the requirement of module IE2032, Sri Lanka Institute of Information Technology.

Acknowledgment

We want to express our heartfelt gratitude to our Lecturer, Ms. Suranjini Silva for her valuable advice and support. In addition, we would like to thank our friends and elders for their support and guidance in completing this report through suggestions and comments.

Contents

Abstract 5
Introduction 6
Programming on High Performance System 7
A Freestanding Rust Binary 7
A Minimal Rust Kernel 11
VGA Text Mode 13
Testing 18
CPU Exceptions 19
Double Faults 23
Hardware Interrupt 28
Introduction to Paging 29
Paging Implementation 32
Heap Allocation 39
Allocator Designs 47
The Async/Await Pattern 51
Performance 59
Reliability 61
Productivity 62
Security 63
Conclusion 65
References 66
Contribution Of Group Members 67

Abstract

The languages of preference for high-performance computing (HPC) applications have often been Fortran and C++. However, they are both over 35 years old and do not provide much in the way of memory security or user-friendliness. A newly emerging systems language called Rust aims to be efficient while providing features like safety and usability, as well as bundling the tools that a contemporary developer requires.

We evaluate several finite difference stencil code implementations and demonstrate that idiomatically written Rust programs may be just as effective as their Fortran or C++ equivalents while providing the benefits.

Introduction

Rust is a statically and strongly typed programming language. It’s focusing on safety, speed, and concurrency. It's also compiled. And what that means is that if you write a Rust program, you compile it into a binary executable file, and that file can be run by anyone who doesn't have Rust installed. So, this differs from something like Python, or JavaScript, or Ruby, where you need a Python interpreter, for example, on your system to be able to run the code. This is different. Someone does not have to have Rust installed to run your Rust program, assuming they're on the same operating system that you were on when you compiled the program. Continuing here, Rust is known as a very fast and reliable programming language. It's very good at error handling and catching bugs, and a lot of developers use it because it is very efficient. It's a very fast language in terms of execution speed, but it's also not extremely difficult to write. The main benefit of Rust is that it has a bunch of high-level features, but it also gives you control of lower-level features and code without some of the difficulties of languages that typically do that, like C or C++. So really, that is why you would use rust. You want to make an efficient, scalable, very fast system of some sort, but you don't want to deal with the kind of the annoyance of writing code, for example, in C, yet you still want to have control of those low-level features. Again, this is a super high-level overview of the language. There are a ton of other awesome features, and I have a little list here.

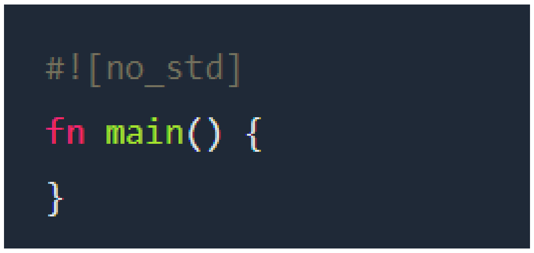
- Rust allows you to make command-line tools
- Web services
- Dev ops tools
- Embedded devices or embedded systems
- Cryptocurrency related stuff
- Bioinformatics
- Search engines
- Machine learning
- And it's used in part of the Firefox browser.

A Freestanding Rust Binary

In this section, we'll build a Rust binary that will ultimately operate on bare metal. This effectively indicates that we won't be utilizing any standard library features for Rust. In addition, because we are building an OS from scratch, we won't be utilizing any features provided by our host operating system, like threads, files, heap memory, the network, random numbers, standard output, etc.

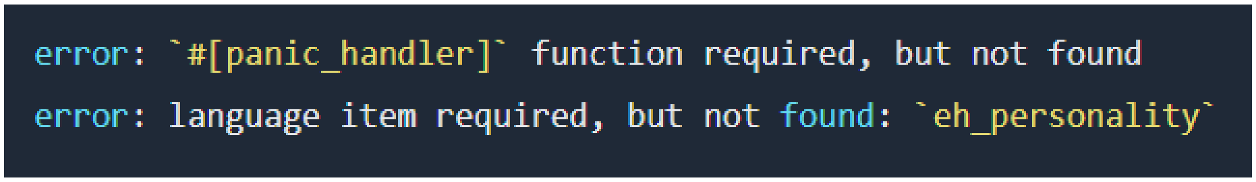
The no_std Attribute

Let's disable this as the standard library is used implicitly by Rust. The compiler is instructed not to connect the standard library using the attribute #![no std].



```rust
#![no_std]
fn main() {
}
```

We encounter the following errors when building:



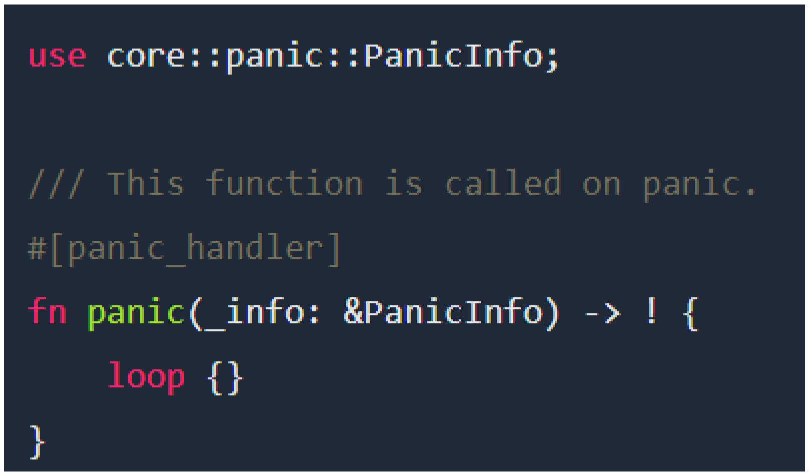
```
error: `#[panic_handler]` function required, but not found
error: language item required, but not found: `eh_personality`
```

Let's try to comprehend these 2 mistakes:

#[panic_handler]

Rust calls the panic handler when it encounters an unforeseen issue during runtime, and the handler then dumps the crash's stack trace. This can be found in the standard library.

We must offer an implementation for the standard library because we disabled it.



```rust
use core::panic::PanicInfo;

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

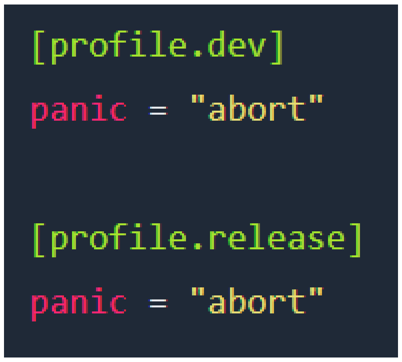
This function should never return, as indicated by the **'!'**. A diverging function is what it is known as. The guide just adds an empty loop.

eh_personality

It's a language-related item. These language item properties give the compiler additional information about the function when it parses the code.

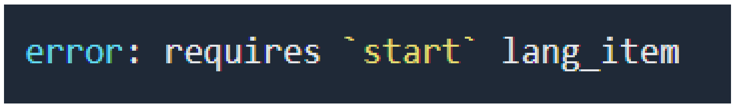
The compiler is specifically informed that this function is used to implement stack unwinding by the eh personality type. This basically refers to the program that is run when a function is called from the function stack. Therefore, it is likely that this will release the RAM that the function was utilizing.

Let's disable this by including the following code in our Cargo.toml file.



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

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

Now that we have fixed both errors, let's try to build it:



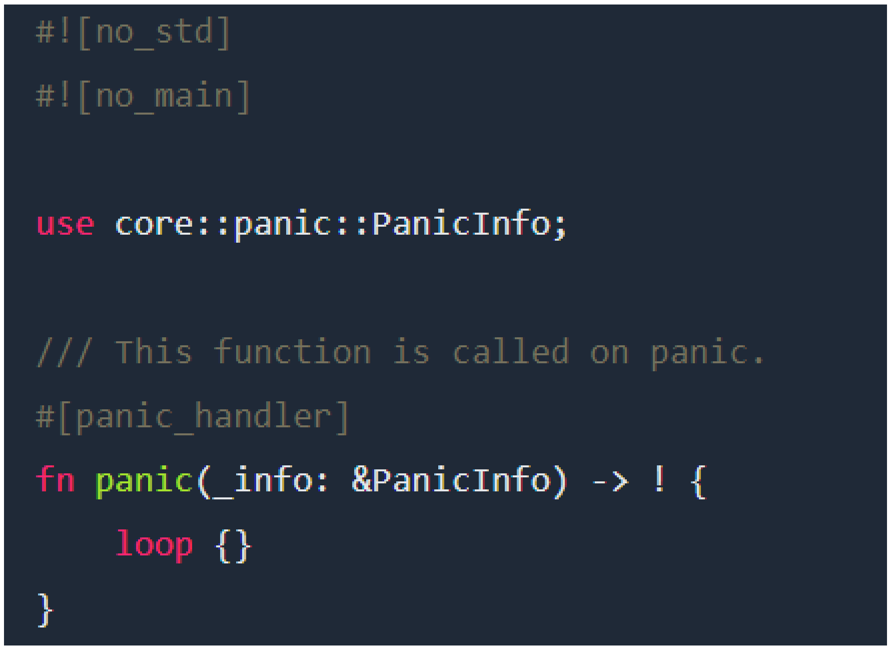
```
error: requires `start` lang_item
```

So, it turns out that in a rust program, the main function is not always called first.

Therefore, we are not using the crt0 (C Runtime 0) and start language item in our situation. Therefore, we must establish our own entry point.

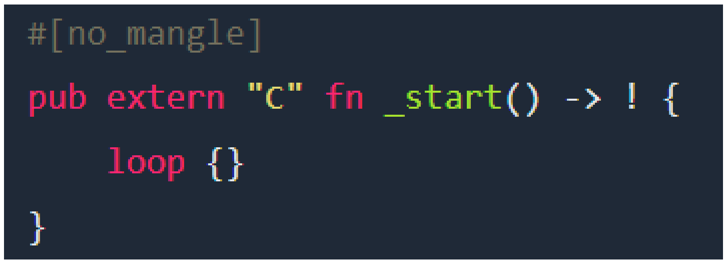
Defining our entrypoint

We can use #![no_main] to remove runtime:



```
#![no_std]
#![no_main]

use core::panic::PanicInfo;

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

We can define our entry point:

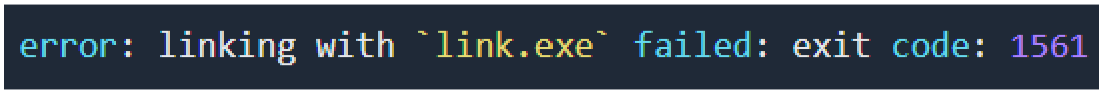


```
#[no_mangle]
pub extern "C" fn _start() -> ! {
    loop {}
}
```

Therefore, during compilation, the rust compiler replaces all the function names to new, distinct values. This process is known as name mangling. Because we eventually want to refer to our start function by name, we are preventing that.

To instruct the compiler to call this function using the C calling convention, we must also declare the function as extern "C". Furthermore, the ‘!’ will change the function into a diverging function that shouldn't return.

Then we get following error:



```
error: linking with `link.exe` failed: exit code: 1561
```

The linker (an application that produces executable code from the compiler's generated objects) presumptively uses the C runtime by default. So, develop for a bare-metal target in order to solve this.

Building for bare-metal

The target triple string is used by Rust to specify various environments. This string's objective is to describe the machine's architecture. When building for our host triple, the linker issues occur because the Rust compiler and linker presume that there is an underlying operating system, such as Linux or Windows, that utilizes the C runtime by default. In order to prevent the linker issues, we may alternatively compile for a new environment without an underlying OS.

An example:

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

As a result, the build will succeed, and our executable will be cross compiled for a bare metal target.

A Minimal Rust Kernel

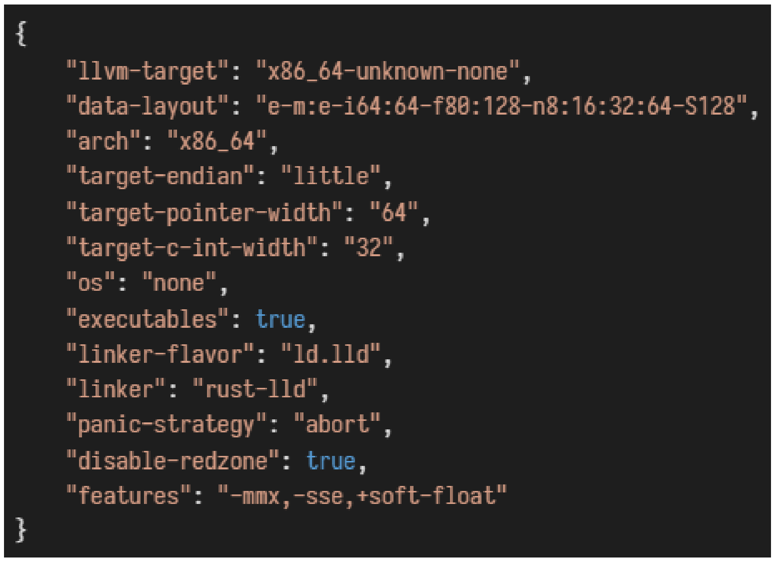
In the prior article, we created the freestanding binary using cargo, however we need various namespaces for entry and compilation settings depending on the OS. This is because cargo, which you are now using, builds by default for the host system. This is not what we want for our kernel because it makes little sense for a kernel to run on top of, say, Windows. Instead, we prefer to compile for a system with a known target.

Nightly Rust installation is necessary. The nightly compiler allows us to select among a selection of experimental features by using featured flags at the front of our project.. The experimental asm! macro, for instance, may be made available for inline assembly by adding #![feature(asm)] to the top of our main.rs. It should be noted that these experimental features are utterly unstable, meaning that future versions of Rust may modify or eliminate them without prior notice. We won't use them unless it is absolutely essential because of this.

Target specifications

The —target parameter in cargo supports a variety of target systems. A target triple is used to describe the target and includes information about the CPU architecture, the vendor, the operating system, and the ABI. Rust supports a wide range of target triples, such as wasm32-unknown-unknown for WebAssembly and arm-linux-androideabi for Android.

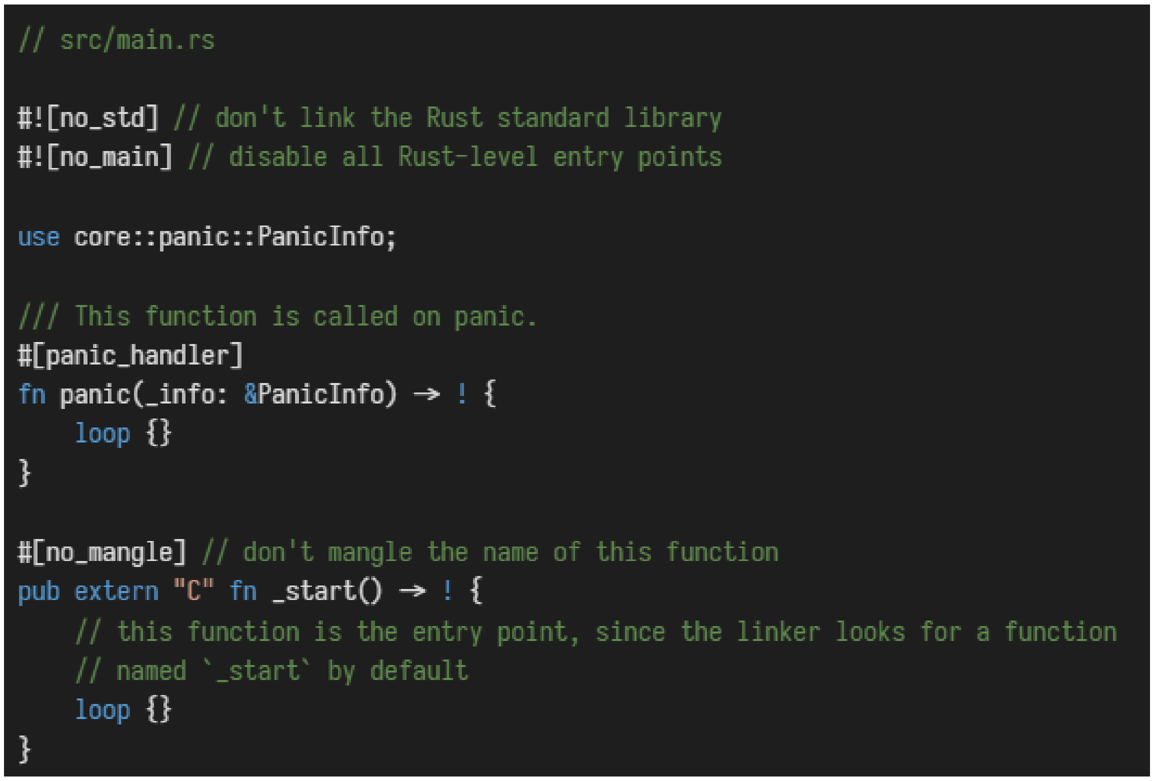
the target specification file currently seems as follows:



```json
{
    "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"
}
```

Building the kernel

We will follow Linux norms while compiling for our new target. This implies that we require the above _start entry point.



```rust
// src/main.rs

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

use core::panic::PanicInfo;

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

#[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 {}
}
```

The core library is distributed as a precompiled library together with the Rust compiler, which is a concern. Therefore, it does not apply to our custom target and is only valid for supported host triples. Recompiling the core for these targets is required before we may compile code for other targets.

Memory-Related Intrinsic

The Rust compiler assumes that all systems support a particular set of built-in functions. The compiler _builtins crate that we just recompiled provides most of these functions. However, because the system's C library typically provides these functions, several memory-related functions in that crate are not enabled by default. These include the memset, memcpy, and memcmp functions, which change all the bytes in a memory block to a specified value, copy a memory block to another, and compare two memory blocks, respectively. While none of these functions were necessary to compile our kernel at this time, they will be as soon as we add additional code to it.

Printing

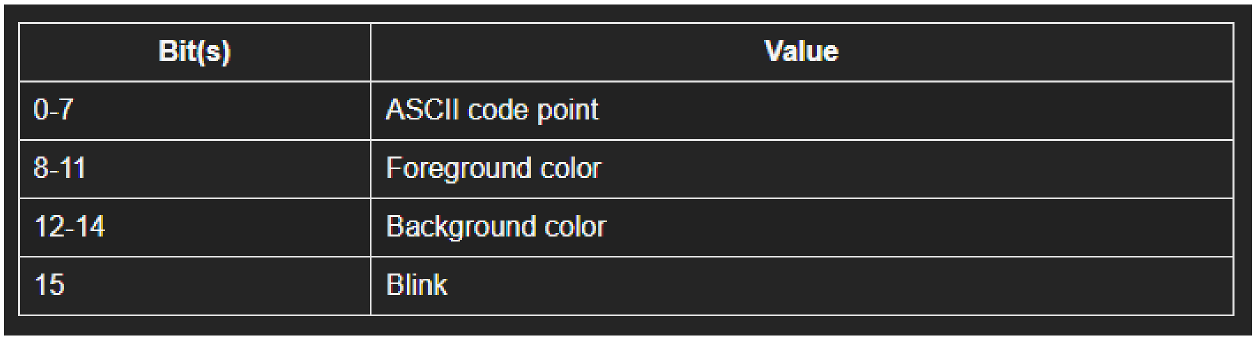
The VGA text buffer is currently the simplest method for printing text on the screen. The information shown on the screen is stored in a unique memory space that is mapped to the VGA hardware. It typically consists of 25 lines with 80 character cells per line. An ASCII character is displayed in each character cell along with certain background and foreground hues.

VGA Text Mode

When utilizing the VGA text mode, printing text on the screen is simple. In this situation, adding support for Rust's macros and enclosing any unsafety in a separate module can produce an interface that makes usage secure and uncomplicated.

The VGA Text Buffer

A character must be written to the VGA hardware's text buffer before it can be displayed on the display. The VGA text buffer is a two-dimensional array that is immediately projected to the screen. It normally has 25 rows and 80 columns. Each array entry describes one screen character..



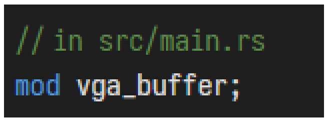
Bit(s)	Value
0-7	ASCII code point
8-11	Foreground color
12-14	Background color
15	Blink

- The first byte represents the character that should be printed in the ASCII encoding.
- The second byte defines how the character is displayed.
- The next three bits the background color, and the last bit whether the character should blink.

Memory-mapped I/O can be used to access the VGA text buffer at address 0xb8000. As a result, reads and writes to that location directly access the text buffer on the VGA hardware rather than the RAM. This indicates that we can read and write to it at that address using standard memory operations.

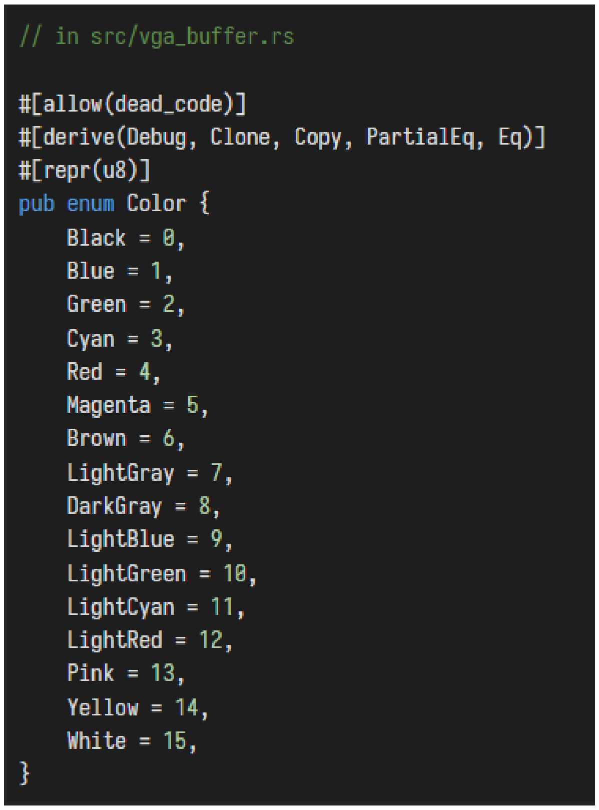
A Rust Module

we can create a Rust module to handle printing:



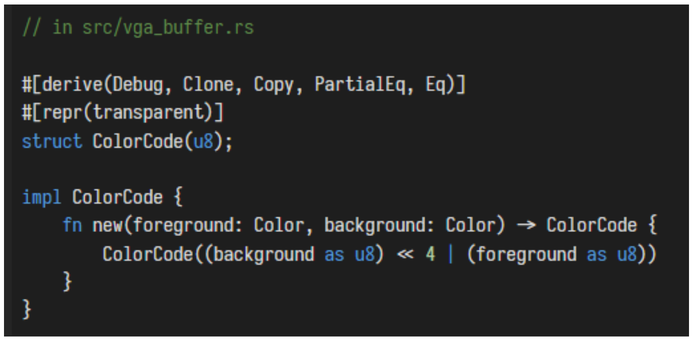
```
// in src/main.rs
mod vga_buffer;
```

First, an enum is used to represent the various colors:


```rust
// in src/vga_buffer.rs

#[allow(dead_code)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u8)]
pub enum Color {
    Black = 0,
    Blue = 1,
    Green = 2,
    Cyan = 3,
    Red = 4,
    Magenta = 5,
    Brown = 6,
    LightGray = 7,
    DarkGray = 8,
    LightBlue = 9,
    LightGreen = 10,
    LightCyan = 11,
    LightRed = 12,
    Pink = 13,
    Yellow = 14,
    White = 15,
}
```

Each enum variant is saved as an u8, according to the repr(u8) attribute. 4 bits would be enough, but there isn't an u4 type in Rust. We can use #[allow(dead_code)] attribute to disable the waring for unused variant.

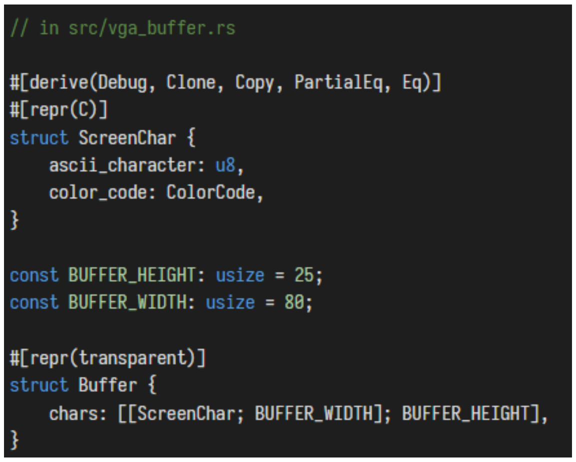
To express a complete color code that specifies foreground and background colors, we develop a newtype on top of u8:


```rust
// in src/vga_buffer.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(transparent)]
struct ColorCode(u8);

impl ColorCode {
    fn new(foreground: Color, background: Color) -> ColorCode {
        ColorCode((background as u8) << 4 | (foreground as u8))
    }
}
```

The whole color byte, which includes the foreground and background colors, is contained in the Color Code struct. We again derive the features of Copy and Debug for it. The repr(transparent) feature is used to make sure that the Color Code has the exact same data layout as an u8.

Text Buffer

We can add structure to represent a screen character and the text buffer:

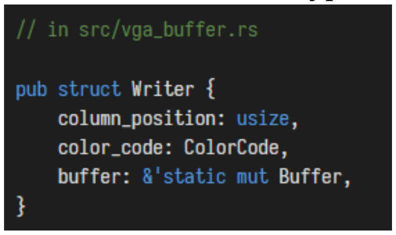


```
// in src/vga_buffer.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(C)]
struct ScreenChar {
    ascii_character: u8,
    color_code: ColorCode,
}

const BUFFER_HEIGHT: usize = 25;
const BUFFER_WIDTH: usize = 80;

#[repr(transparent)]
struct Buffer {
    chars: [[ScreenChar; BUFFER_WIDTH]; BUFFER_HEIGHT],
}
```

repr(C) = Default structs like C

repr(transparent) = same memory layout as single field (for buffer struct)

We now build a writer type to really write to screen:z

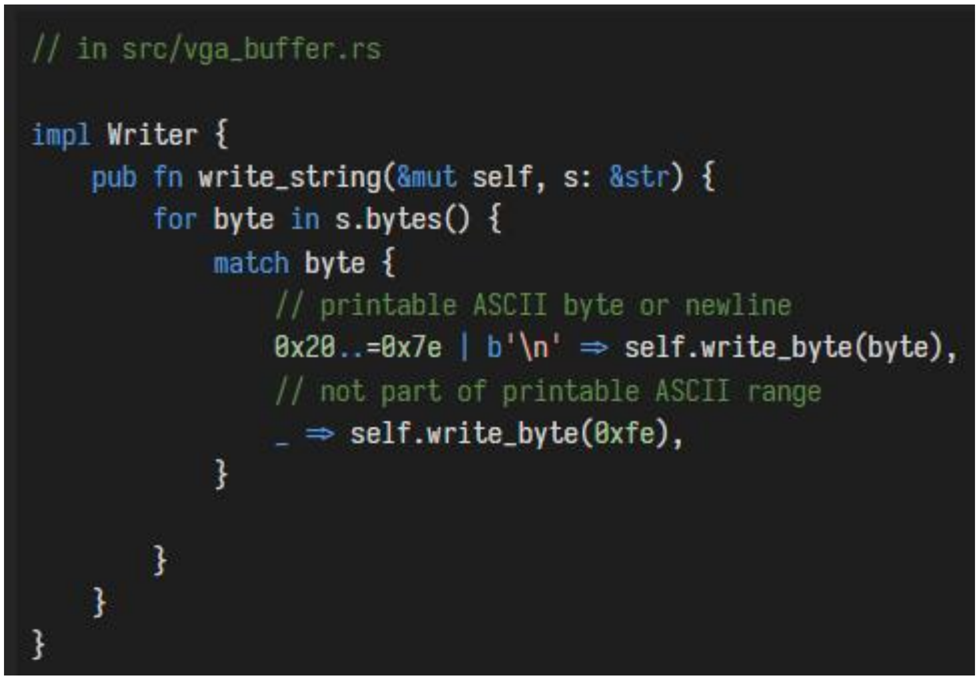


```
// in src/vga_buffer.rs

pub struct Writer {
    column_position: usize,
    color_code: ColorCode,
    buffer: &'static mut Buffer,
}
```

'static = reference is valid for whole program run time

Printing

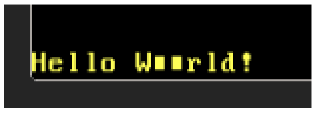
We can convert strings to bytes and print each byte individually to print entire strings:



```
// in src/vga_buffer.rs

impl Writer {
    pub fn write_string(&mut self, s: &str) {
        for byte in s.bytes() {
            match byte {
                // printable ASCII byte or newline
                0x20..=0x7e | b'\n' => self.write_byte(byte),
                // not part of printable ASCII range
                _ => self.write_byte(0xfe),
            }

        }
    }
}
```

In the VGA text buffer, only ASCII is supported. Because Rust strings are UTF-8 by default, they could include bytes that the VGA text buffer cannot handle.

We can print hello world below in the bottom left corner.



```
Hello W■■rld!
```

Testing

Rust methods known as tests are used to check if non-test code is operating as intended. Usually, the test function bodies carry out these three tasks.

1. Set up any needed data or state.
2. Run the code you want to test.
3. Assert the results are what you expect.

Let's examine the tools that Rust offers for creating tests that do these activities, including the test attribute, a few macros, and the should panic attribute.

Add #[test] to the line before a function to turn it become a test function. When you use the cargo test command to run your tests, Rust creates a test runner binary that executes the annotated functions and generates a report on whether or not each test function succeeds. With Cargo, a test module with a test function is automatically created for us whenever we create a new library project. In order to save you time every time you begin a new project; this module provides you with a template for creating your tests. You are free to include as many more test modules and test routines as you like.

Before we test any code, we'll play with the template test to see certain features of how tests function. Then, we'll create some practical tests that run our previously developed code and confirm that it behaves as intended.

Let's create a new library project called "adder" that will add two numbers:

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Unit tests may be executed using Rust's built-in test framework without any further setup. All you need to do is write a function that uses assertions to verify some outcomes, then add the #[test] property to the function header. Following that, cargo test will instantly locate and run all of your crate's test functions.

Unfortunately, no std programs like our kernel require a little more complexity. The issue is that the built-in test library, which depends on the standard library, is used implicitly by Rust's test framework. This means that for our #[no std] kernel, we cannot utilize the default test framework.

CPU Exceptions

CPU exceptions can happen while accessing an improper memory location or when dividing by zero, among other erroneous circumstances. We must build up an interrupt descriptor table with handler routines in order to respond to them. Our kernel will be able to catch breakpoint exceptions at the conclusion of this article and then restart regular execution.

An exception indicates that the current instruction is in error. For instance, the CPU produces an error if the current instruction tries to divide by Zero. Depending on the kind of exception, the CPU invokes a specified exception handler code right away when an exception occurs, interrupting its ongoing activity.

There are about 20 different CPU exception time. The most important are:

Page fault

Illegal memory accesses result in a page fault. For instance, if the current instruction tries to write to a read-only page or read from an unmapped page.

Double Faults

The CPU attempts to invoke the appropriate handler code whenever an exception occurs. The CPU raises a double fault exception if another exception happens while invoking the exception handler. When no handler function has been registered for an exception, this exception also occurs. We are explaining Furth more in the next level about the double faults.

Invalid Opcode

An error occurs when the current instruction is incorrect, for example, when we try to utilize contemporary SSE instructions on a legacy CPU that does not support them.

General Protection Fault

This is the anomaly with the most varied causes. It occurs when a user, among other access violations, tries to execute a privileged instruction in user-level code or puts reserved fields into configuration registers..

Triple Fault

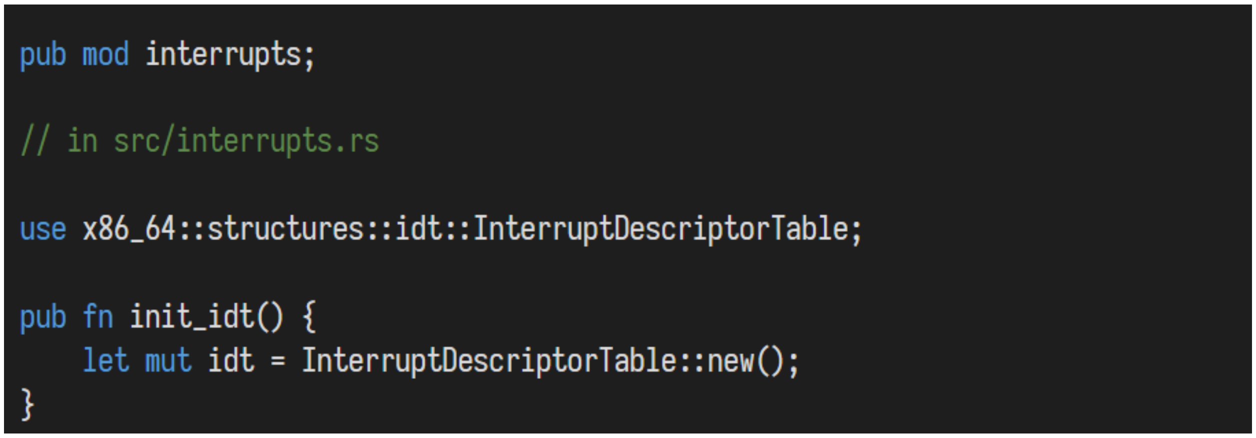
The CPU delivers a fatal triple fault if an error arises when it attempts to call the double fault handler code. A triple fault cannot be handled or caught by us. Most processors respond by restarting the operating system and themselves.

The CPU generally acts as follows when an exception arises,

1. Push a few registers, such as the instruction pointer and the REFLAGS register, onto the stack.
2. Check the Interrupt Descriptor Table for the matching item (IDT)
3. Verify if the entry is there and issue a double fault if it isn't.
4. If the entry is an interrupt gate, disable hardware interrupts.
5. Insert the Global Descriptor table selection supplied into the CS.
6. Navigate to the chosen handler function.

Implementation Of CPU Exception

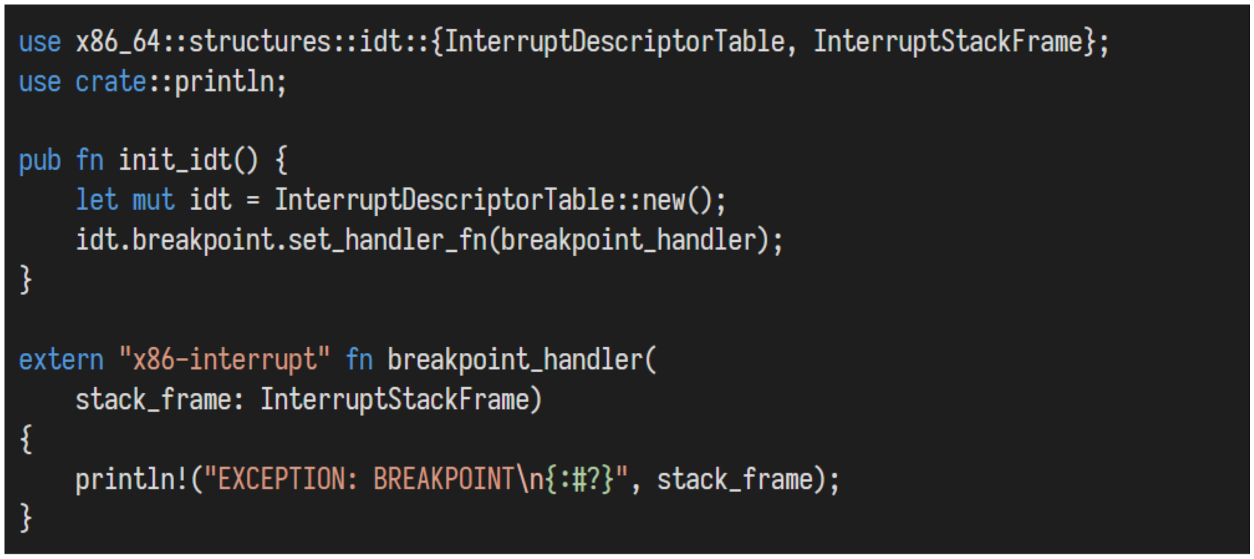
Now CPU exceptions are handled in our kernel.



```
pub mod interrupts;

// in src/interrupts.rs

use x86_64::structures::idt::InterruptDescriptorTable;

pub fn init_idt() {
    let mut idt = InterruptDescriptorTable::new();
}
```

We are now able to add handler functions. The first thing we do is provide a handler for the breakpoint exception. To test exception handling, the breakpoint exception is the ideal exception. Its sole function is to execute the breakpoint instruction int3 and halt the program for a short period of time.

Debuggers frequently employ the breakpoint exception, The breakpoint exception is thrown by the CPU when it reaches that line because the debugger replaces the appropriate instruction with the int3 instruction when the user sets a breakpoint. The debugger replaces the int3 instruction with the original instruction once more and resumes the program when the user requests to do so.



```rust
use x86_64::structures::idt::{InterruptDescriptorTable, InterruptStackFrame};
use crate::println;

pub fn init_idt() {
    let mut idt = InterruptDescriptorTable::new();
    idt.breakpoint.set_handler_fn(breakpoint_handler);
}

extern "x86-interrupt" fn breakpoint_handler(
    stack_frame: InterruptStackFrame)
{
    println!("EXCEPTION: BREAKPOINT\n{:#?}", stack_frame);
}
```

Double Faults

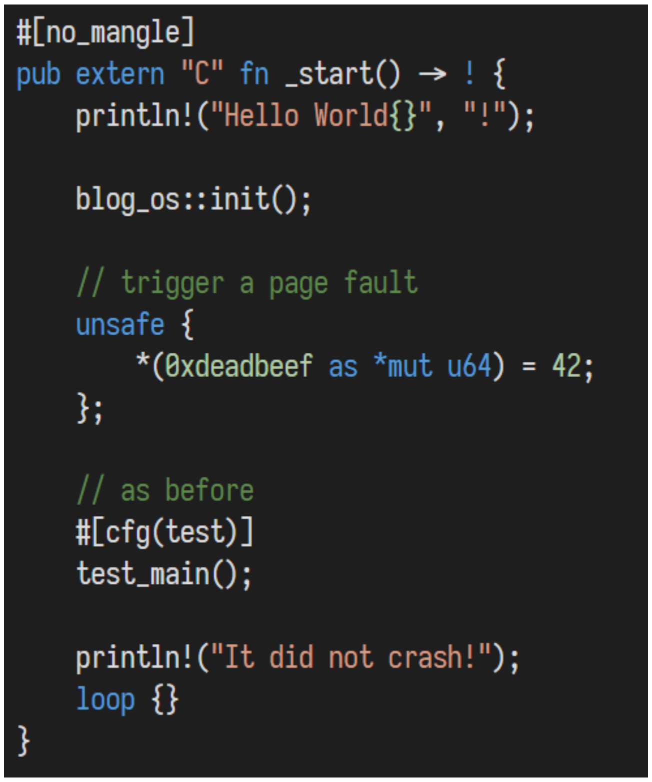
This post offers a thorough overview of the double fault exception, which arises when the CPU neglects to invoke an exception handler. By handling this error, we avoid catastrophic triple failures that call for a system reset. We further developed an Interrupt Stack Table to identify double faults on a separate kernel stack to entirely avoid triple faults.

What is the Double fault.?

A double fault, defined simply, is a special exception that happens when the CPU neglects to call an exception handler. For example, it occurs when a page fault is triggered but there is no page fault handler registered in the Interrupt Descriptor Table (IDT).

Triggering Double Fault

Let's cause an exception for which we have not yet defined a handler method to result in a double fault:



```
#[no_mangle]
pub extern "C" fn _start() -> ! {
    println!("Hello World{}", "!");

    blog_os::init();

    // trigger a page fault
    unsafe {
        *(0xdeadbeef as *mut u64) = 42;
    };

    // as before
    #[cfg(test)]
    test_main();

    println!("It did not crash!");
    loop {}
}
```

The invalid address 0xdeadbeef is written to using unsafe. A page fault happens when the virtual address in the page tables is not mapped to a physical address. A double fault happens because we don't have a page fault handler registered in our IDT (Interrupt Descriptor Table). Our kernel now enters an unending boot loop when we try to start it. The following explains the boot loop:

1) A page fault is caused when the CPU tries to write to 0xdeadbeef.
2) Upon inspecting the appropriate entry in the IDT, the CPU notices that no handler function is mentioned. Because of this, a double fault happens, and it is unable to contact the page fault handler.
3) The CPU examines the IDT item for the double fault handler, but this entry also lacks a handler function specification. A triple fault therefore happens.
4) A threefold error results in death. As with most of the genuine hardware, QEMU responds to it by performing a system reset.

So, either a page fault handler method or a double fault handler must be provided if we want to avoid this triple fault.

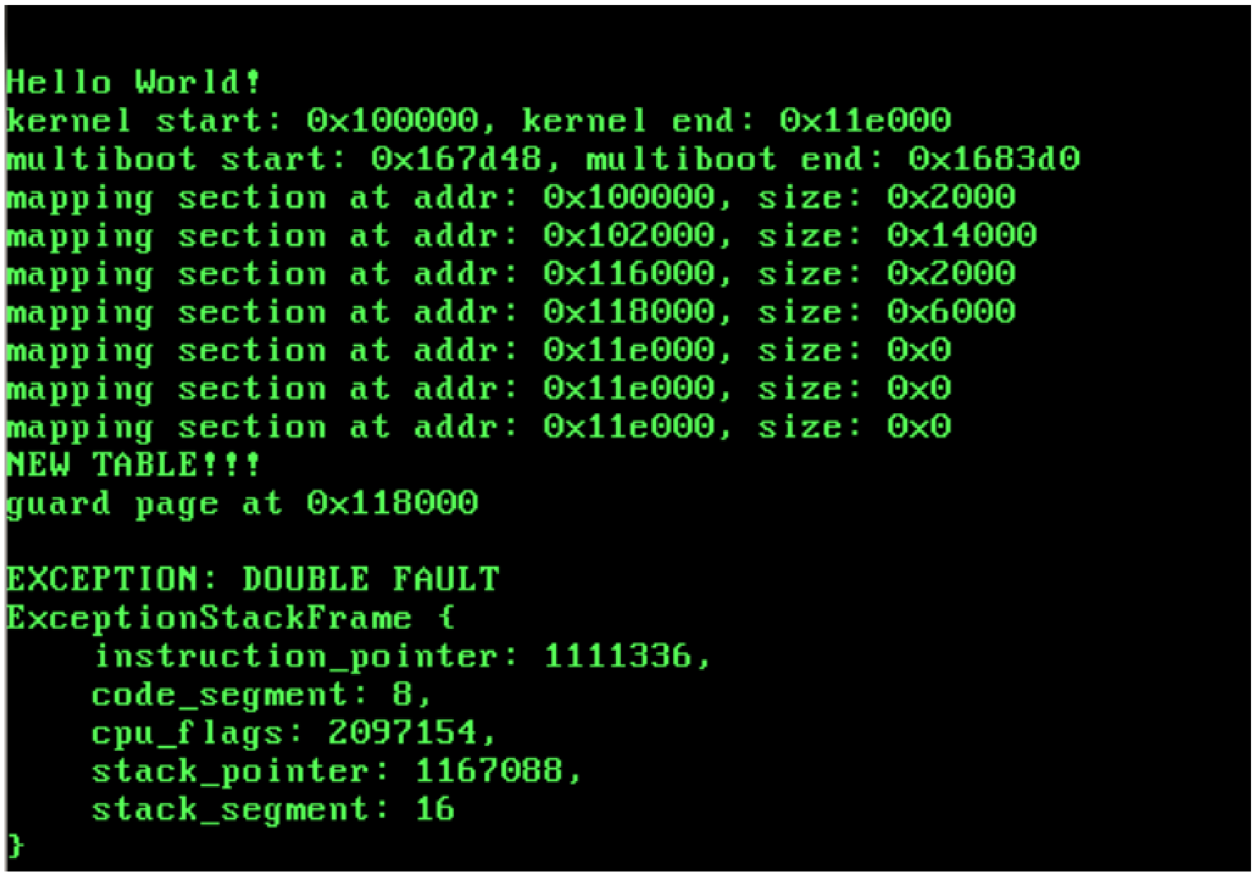
Double Fault Handler

A handler method identical to our breakpoint handler may be specified because a double fault is a common exception with an error code:

```rust
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);
}
```

The exception stack frame is dumped along with a brief error message by our handler. There is no need to report the error code of the double fault handler because it is always zero.

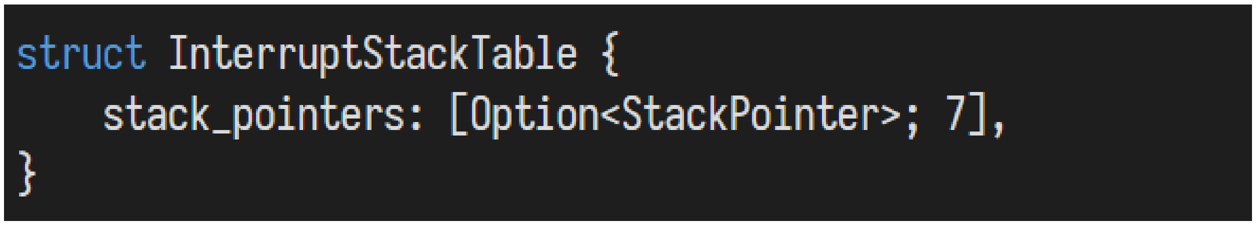
When we start our kernel now, we should see that the double fault handler is invoked:



```
Hello World!
kernel start: 0x100000, kernel end: 0x11e000
multiboot start: 0x167d48, multiboot end: 0x1683d0
mapping section at addr: 0x100000, size: 0x2000
mapping section at addr: 0x102000, size: 0x14000
mapping section at addr: 0x116000, size: 0x2000
mapping section at addr: 0x118000, size: 0x6000
mapping section at addr: 0x11e000, size: 0x0
mapping section at addr: 0x11e000, size: 0x0
mapping section at addr: 0x11e000, size: 0x0
NEW TABLE!!!
guard page at 0x118000

EXCEPTION: DOUBLE FAULT
ExceptionStackFrame {
    instruction_pointer: 1111336,
    code_segment: 8,
    cpu_flags: 2097154,
    stack_pointer: 1167088,
    stack_segment: 16
}
```

Switching stacks

When an exception occurs, the x86 64 architecture has the ability to transition to a predetermined, known-good stack. This switch can be executed prior to the CPU pushing the exception stack frame because it occurs at the hardware level. This switching mechanism is implemented as an Interrupt Stack Table. (IST)



```
struct InterruptStackTable {
    stack_pointers: [Option<StackPointer>; 7],
}
```

Through the options field in the associated IDT (Interrupt Descriptor Table) item, we can select a stack from the IST (Interrupt Stack Table) for each exception handler. For instance, we may implement our double fault handler on the first stack in the IST (Interrupt Stack Table).When a double fault happens, the CPU would then immediately switch to this stack. The triple fault would be avoided because this switch would occur before anything is pushed.

We need a means to allocate new stacks so that we can later fill an Interrupt Stack Table. As a result, we add a new [stack_allocator] submodule to our [memory] module.

```
mod stack_allocator;
```

We make a new constructor function and" StackAllocator" struct:

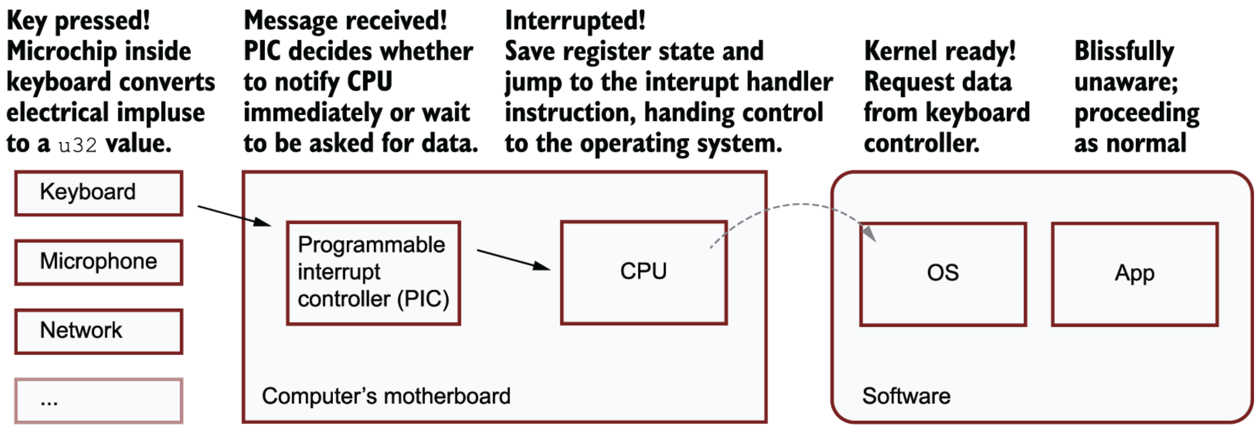
```
use memory::paging::PageIter;

pub struct StackAllocator {
    range: PageIter,
}

impl StackAllocator {
    pub fn new(page_range: PageIter) -> StackAllocator {
        StackAllocator { range: page_range }
    }
}
```

A straightforward "StackAllocator" that allots stacks from a specified range of pages is created.

Hardware Interrupt

Interrupts allow attached hardware devices to signal the CPU. Instead of allowing the kernel to inspect the keyboard for new characters on a regular basis, the keyboard can inform the kernel of each keypress. Because the kernel only needs to act when something happens, this is far more efficient. It also allows for quicker reaction times because the kernel may respond instantly rather than waiting for the next poll.

Since all the hardware is unable to communicate with the CPU directly, an intermediary called the "Interrupt Controller" is implemented. These programmable interrupt handlers can classify interruptions into priority levels and respond in accordance with those levels.


Key pressed! Microchip inside keyboard converts electrical impluse to a u32 value.
Message received! PIC decides whether to notify CPU immediately or wait to be asked for data.
Interrupted! Save register state and jump to the interupt handler instruction, handing control to the operating system.
Kernel ready! Request data from keyboard controller.
Blissfully unaware; proceeding as normal
Keyboard
Microphone
Network
...
Programmable interrupt controller (PIC)
CPU
Computer's motherboard
OS
App
Software

An asynchronous hardware interrupt should happen. As a result, they are totally independent of the code that was run and can happen at any time. So, all the potential concurrency-related issues are suddenly present in our kernel in the form of concurrency. However, Rust's tight ownership model bans a changeable global state, which prevents such issues from occurring. It also guarantees that Rust is an appropriate programming language for creating a highly dependable modern operating system.

None of these, however, guards against deadlocks, a logical lock created by threads themselves that never releases. It makes that thread in the CPU continue to run indefinitely. If the "Mutex" is locked, interrupts must be turned off in order to prevent such deadlocks.

Introduction to Paging

Here, paging is introduced which is a very common memory management method used for an operating system. It explains why memory isolation is necessary, how segmentation works, what virtual memory is, and how to solve paging memory fragmentation problems.

Memory Protection

One of the main purposes of an operating system is to segregate programs from one another. Operating systems use hardware features to prevent other processes from accessing memory sections of one process. There are several techniques depending on the hardware and OS implementation.

There are some ARM Cortex-M processors (used in embedded systems) that contain a Memory Protection Unit (MPU), which allows you to specify a limited number of memory areas with different access permissions for each area (e.g., 8) (e.g., no access, read-only, and read-write). Each memory access is checked by the MPU to determine if it is in an area with the correct access rights and if it is not, an exception is thrown as a result. The operating system changes the regions and access rights for each process as it transitions to ensure it can only access its own memory as it transitions.

Memory protection is handled by the hardware in two ways: segmentation and paging.

Segmentation

Segmentation added an offset on each memory access, so that up to 1 MiB of memory was accessible. The CPU automatically added this offset on each memory access, so that up to 1 MiB of memory was accessible. To make more than these 64 KiB accessible, additional segment registers were introduced, each containing an offset address.

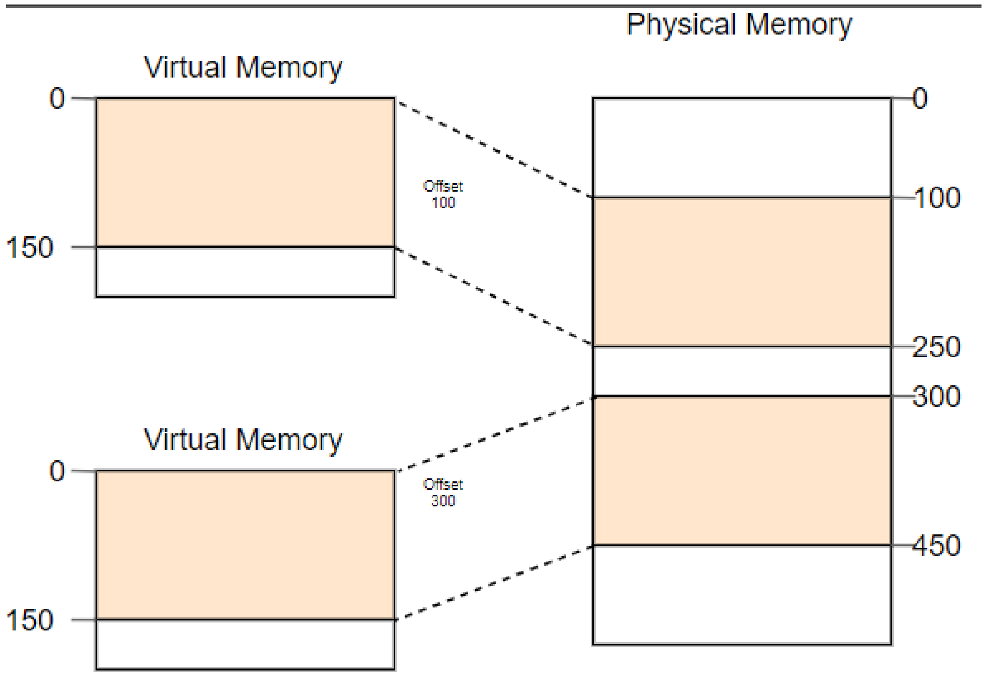
The CPU uses the segment register based on the kind of memory access. The code segment CS is used for fetching instructions, whereas the stack segment SS is utilized for stack operations (push/pop). The data segment DS or the additional segment ES are used by other instructions.

When the CPU operates in this mode, the segment descriptors carry an index into a local or global descriptor database, which provides the segment size and access rights in addition to an offset address. The operating system can isolate processes by loading separate/local tables for each one. Memory accesses are thus limited to the process's own memory locations. This was later altered with the addition of the protected mode.

Virtual Memory

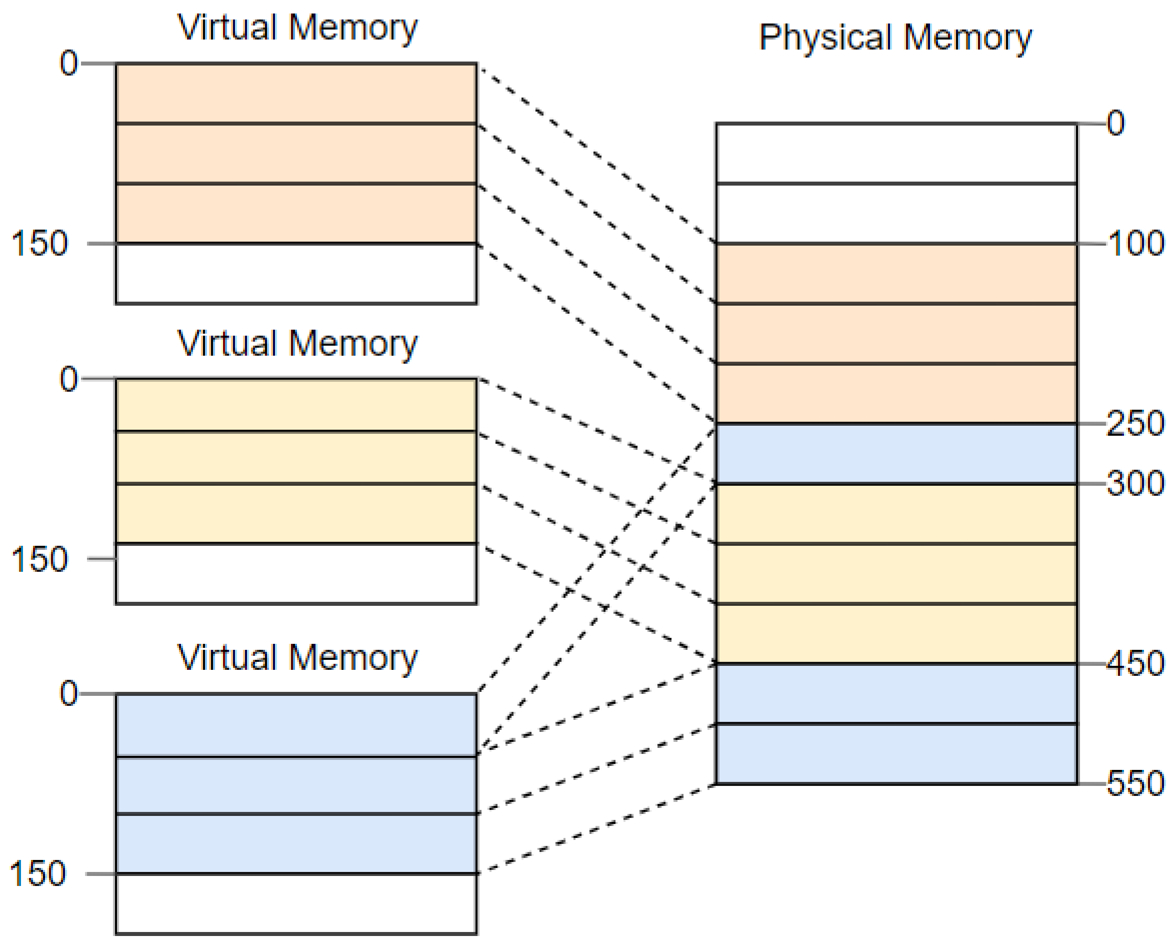
Virtual memory is a method of separating memory addresses from the underlying physical storage device. Instead of accessing the storage device directly, a translation step is done first. The translation step adds the offset address of the active segment to the active section for segmentation.

Physical addresses are one-of-a-kind and always point to the same specific memory region. The translation function, on the other hand, determines virtual addresses. When running the same program twice in parallel, this characteristic comes in handy. When various translation functions are used, same virtual addresses might lead to distinct physical addresses.



Paging

Pages are virtual memory space blocks, whereas frames are actual addresses. Because each page may be independently mapped to a frame, bigger memory areas can be divided among non-continuous physical frames.

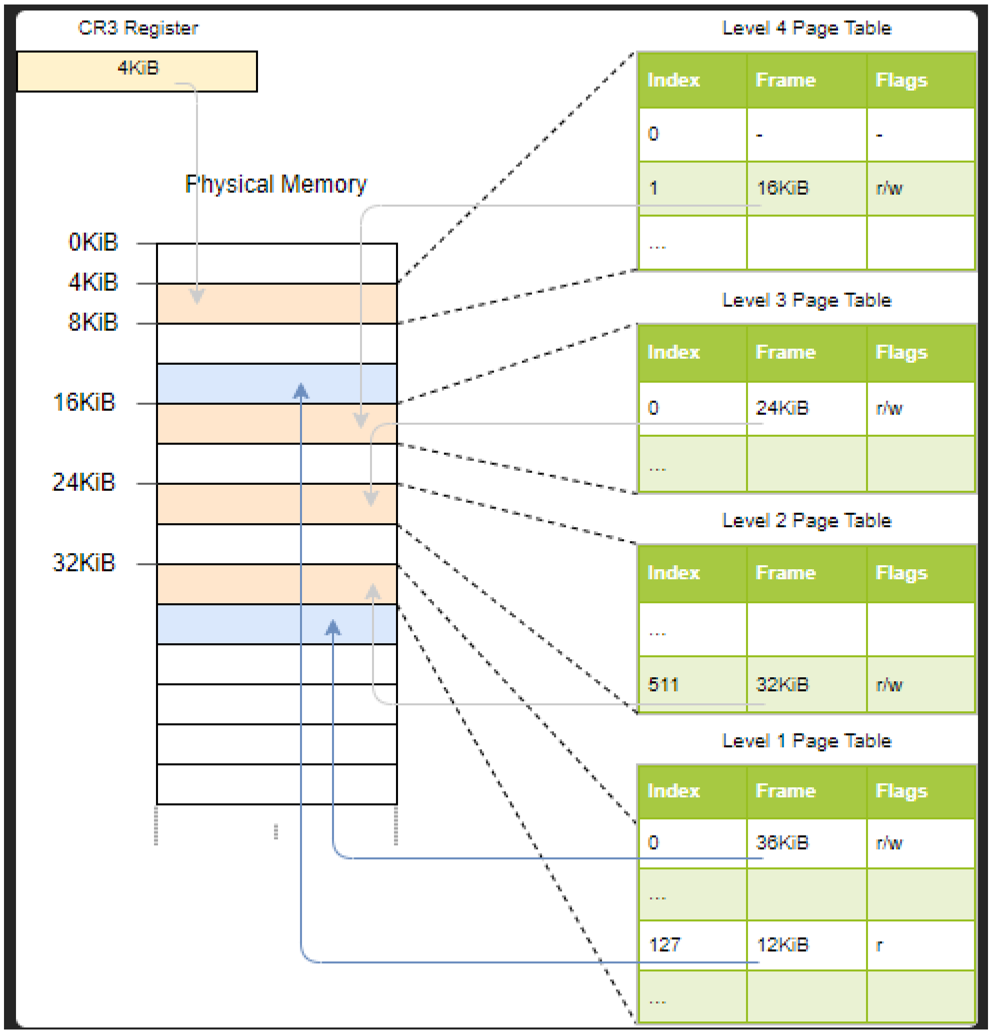


Because each page is separately mapped to a frame, a continuous virtual memory area can be transferred to non-continuous physical frames. This enables us to launch the third instance of the program without first conducting any defragmentation. With a page size of 50 bytes in our example, each of our memory regions is split into three pages.

Paging Implementation

This explores different approaches for the kernel to access page table frames. There are advantages and disadvantages to each approach. In order to implement the approach, support from the bootloader will be required. So, we'll configure it first.

Accessing Page Tables



Level 4 Page Table

Index	Frame	Flags
0	-	-
1	16KiB	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
...		
127	12KiB	r
...		

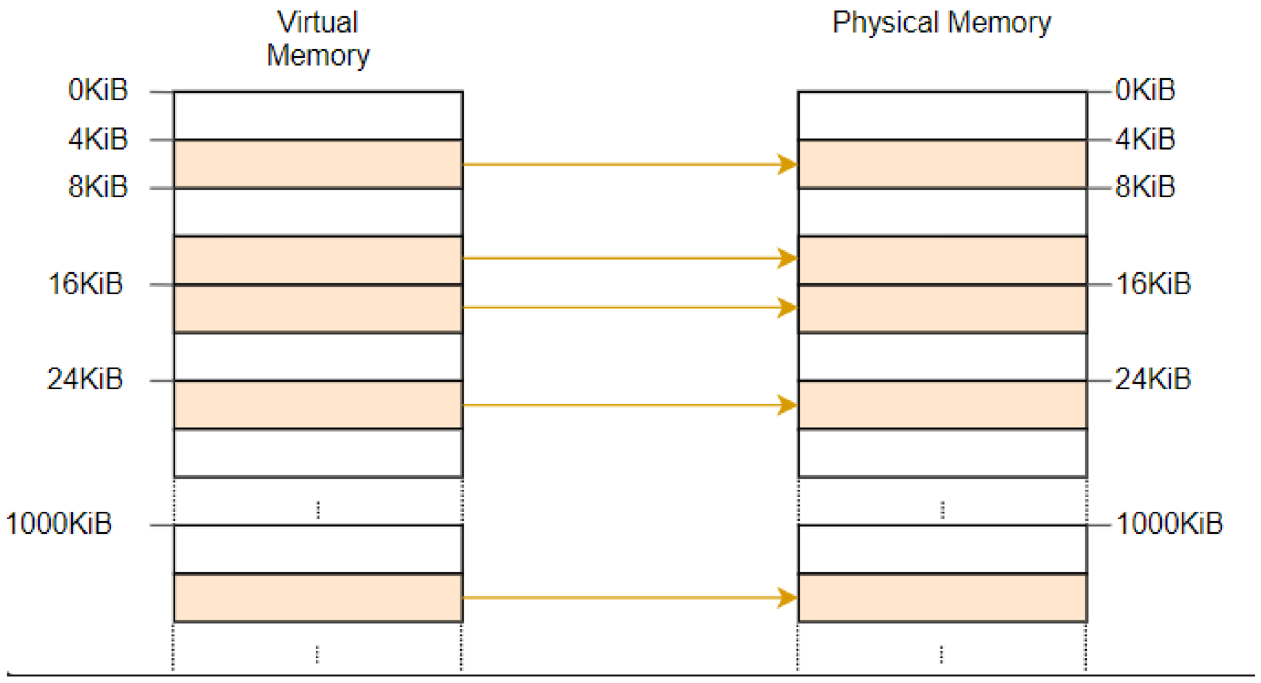
The crucial thing to note here is that each page item contains the physical location of the following table. This eliminates the need to translate these addresses as well, which would be inefficient and may potentially result in infinite translation loops.

We can't directly access physical addresses from our kernel because it also runs on top of virtual addresses. The level 4-page table is stored at address 4 KiB, however when we access address 4

KiB, we access the virtual address 4 KiB instead of the physical address. We can only reach the 4 KiB physical address through a virtual address that corresponds to it.

We must thus map some virtual pages to page table frames to access them. These mappings may be made in a variety of methods, and they all let us access any page table frame.

Identity Mapping



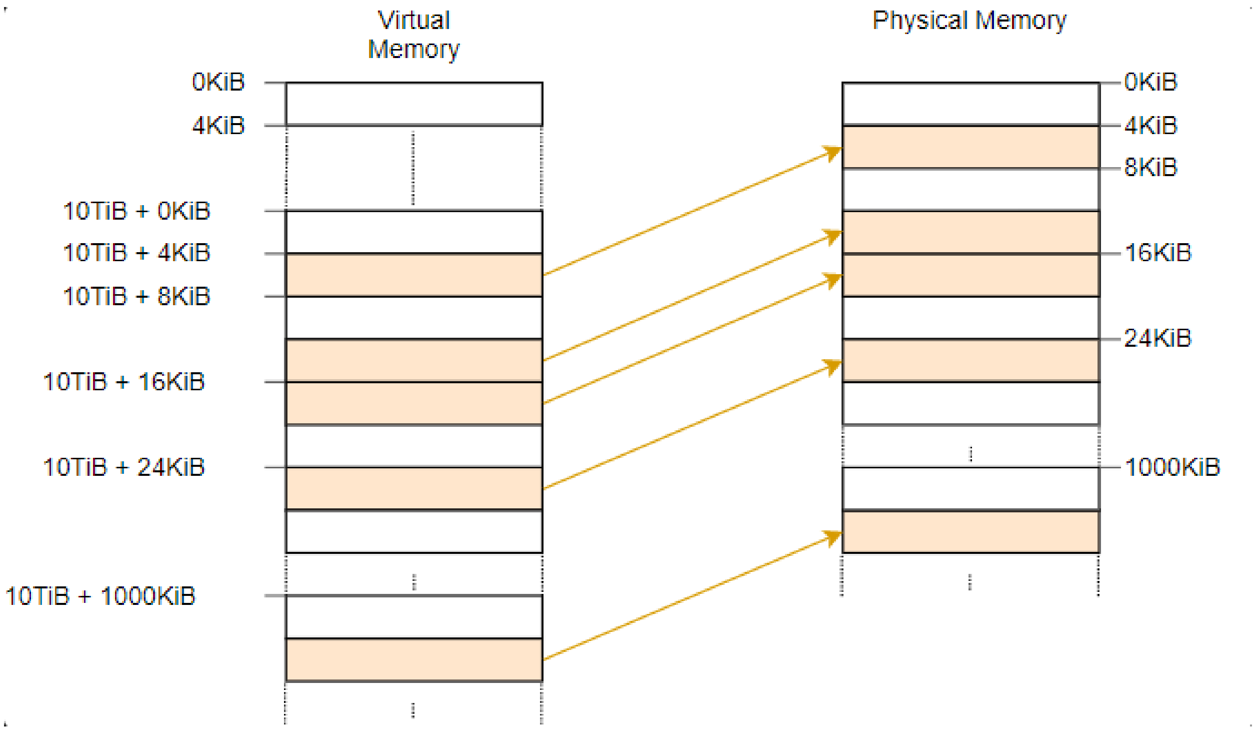
So that we may readily access the page tables of all levels starting with the CR3 register, the physical addresses of page tables are likewise valid virtual addresses.

It complicates the search for continuous memory areas with higher sizes and clogs the virtual address space. Think about how we would wish to, for instance, memory-map a file in the aforementioned image by creating a virtual memory area of 1000 KiB in size. Since the already-mapped page is 1004 KiB, we can't begin the area there since it would overlap. As a result, we must continue searching until we locate a sufficiently big unmapped area, such as at 1008 KiB. A segmentation problem with fragmentation is present here.

We need to locate physical frames with matching pages that aren't currently being used, which makes it considerably more challenging to generate new page tables. For illustration, that we designated the virtual 1000 KiB memory area beginning at 1008 KiB for our memory-mapped file. Any frame with a physical address between 1000 KiB and 2008 KiB is no longer usable because we can't identify map it.

Map at a Fixed Offset

We can utilize a different memory location for page table mappings to prevent the issue of clogging the virtual address space. Therefore, we map page table frames at a defined offset in the virtual address space rather than identity mapping them. The offset may, for instance, be 10 TiB:

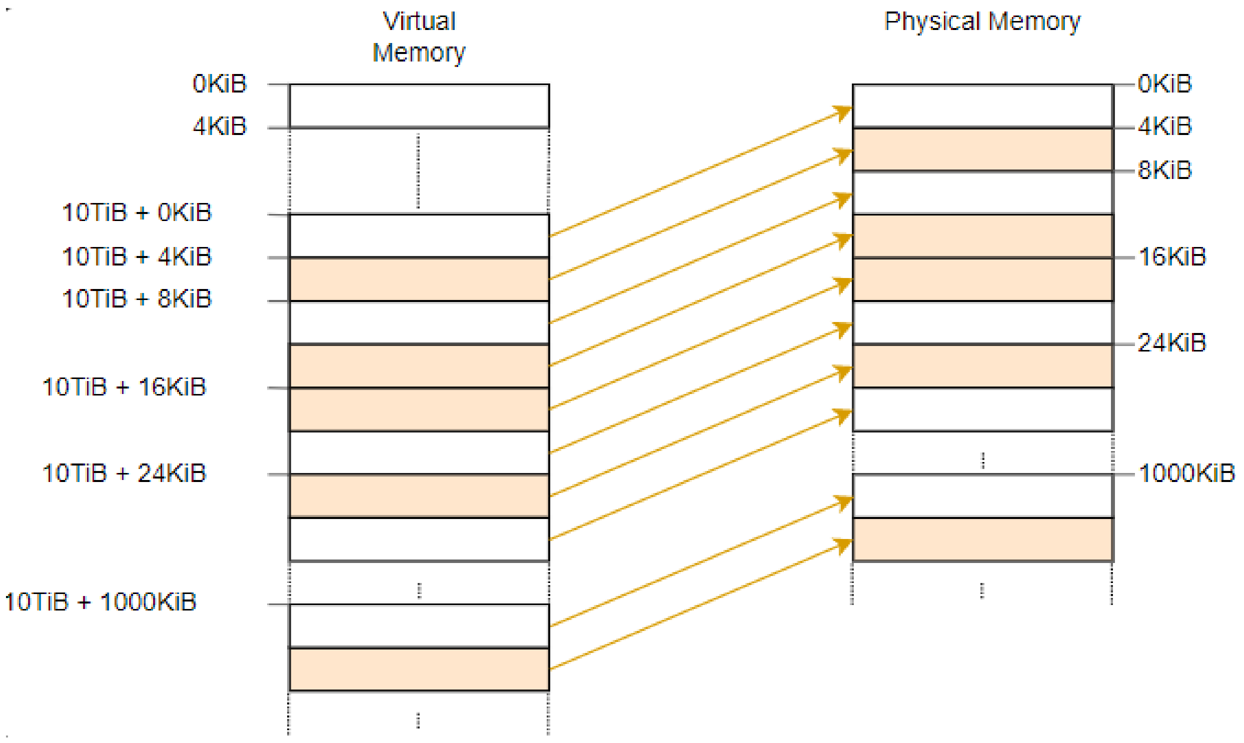


We prevent the collision issues with the identity mapping by limiting the use of virtual memory for page table mappings to the range 10 TiB. (10 TiB + physical memory size). Only when the virtual address space exceeds the physical memory capacity is it feasible to reserve such a sizable portion of the virtual address space. On x86 64, this is not an issue because the 48-bit address space is 256 TiB in size.

This strategy still has the drawback of necessitating the creation of a new mapping each time a new page table is created. Additionally, it does not permit accessing page tables of different address spaces, which would be helpful for developing a new process.

Map the Complete Physical Memory

By mapping the entire physical memory rather than just page table frames, we can address these issues.

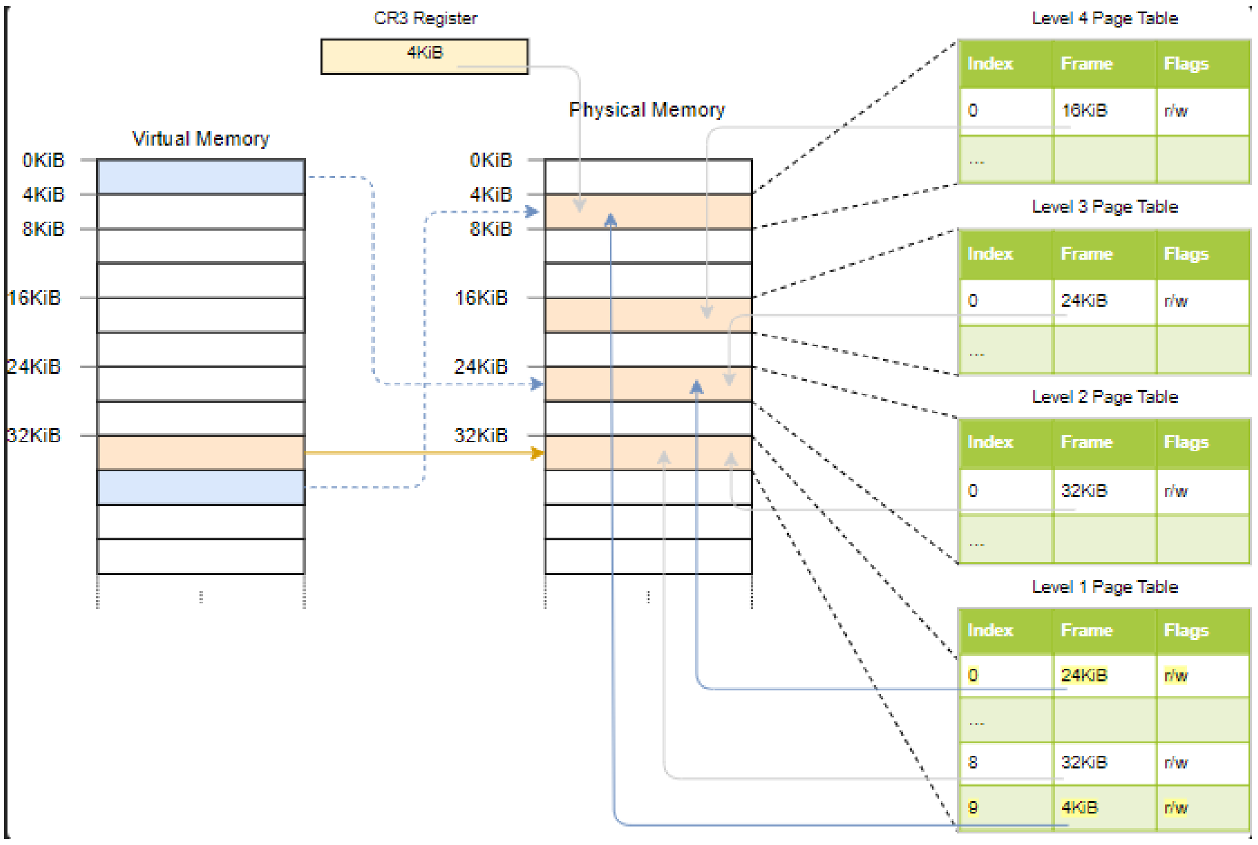


This strategy enables our kernel to access any physical memory, including the page table frames of different address spaces. Except for the absence of unmapped pages, the reserved virtual memory range is the same size as previously.

The drawback of this strategy is the requirement for additional page tables to store the physical memory mapping. Due to the necessity for storage, these page tables use some physical memory, which might be problematic for systems with limited memory.

Temporary Mapping

We could only temporarily map the page table frames on systems with extremely little physical memory in order to access them. There is just one identity-mapped level 1 table that is required in order to construct the temporary mappings.



The virtual address space's initial 2 MiB are under the authority of the level 1 table in this figure. This is due to the fact that it may be reached by beginning at the CR3 register and moving on to the 0th entry in the level 4, level 3, and level 2 page tables. The level 1 table itself is identified by the item with index 8 that connects the virtual page at address 32 KiB to the physical frame at location 32 KiB. The horizontal arrow at 32 KiB in the picture represents this identity-mapping.

Our kernel may produce up to 511 temporary mappings by writing to the identity-mapped level 1 database (512 minus the entry required for the identity mapping). The kernel produced two temporary mappings in the aforementioned illustration.

- The dashed arrow denotes the temporary mapping of the virtual page at 0 KiB to the physical frame of the level 2-page table formed by mapping the 0th item of the level 1 table to the frame with address 24 KiB.
- The dashed arrow denotes the temporary mapping of the virtual page at 36 KiB to the physical frame of the level 4-page table formed by mapping the 9th item of the level 1 table to the frame with address 4 KiB.

Because the mappings are created using the same 512 virtual pages, this method only needs 4 KiB of physical memory. Its disadvantage is that it is quite laborious, particularly given that a new mapping can need for changes at many table levels.

Recursive Paging

Recursive paging is an intriguing method that demonstrates the power of a single mapping in a page table. It is a suitable option for first paging tests since it is reasonably simple to build and only needs a single recursive entry in the setup process.

But there are some drawbacks as well.

- A lot of virtual memory is taken up by it (512 GiB). With a 48-bit address space, this is not a major issue, but it might result in less-than-ideal cache performance.
- Only the address space that is active right now may be easily accessed. Changes to the recursive entry still allow access to other address spaces but switching back requires a temporary mapping.

Heap Allocation

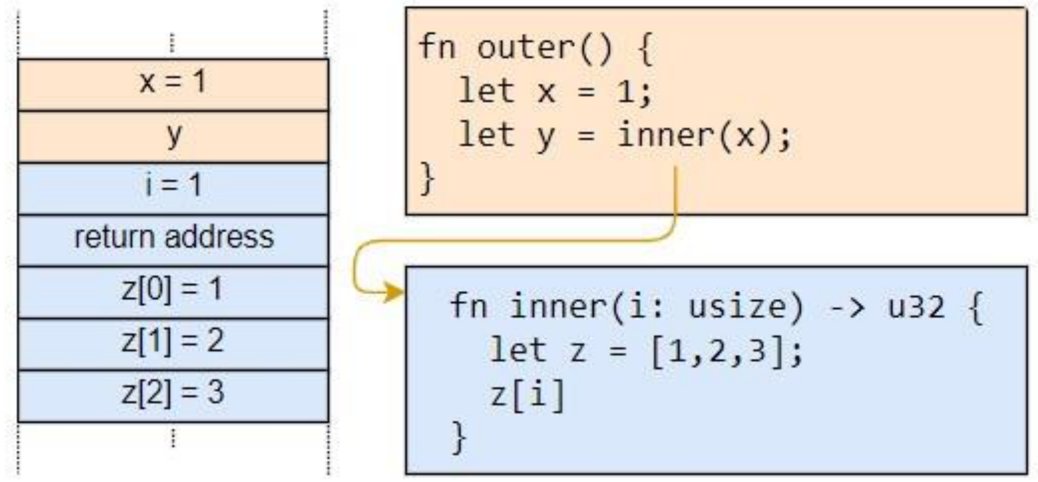
The management of memory allocation is done through a process known as heap allocation. Heap facilitates the control of dynamic memory allocation. Similar to stack allocation, dynamic data objects and data structures can likewise be created using heap allocation.

Local and Static variables

Our kernel currently uses local variables and static variables, both of which are different types of variables. On the call stack, local variables are kept and are only active until the function they are associated with returns. Static variables are always preserved for the whole duration of the program and are kept in a fixed position in memory.

Local Variables

The call stack, a stack data structure that allows for push and pop operations, is where local variables are kept. The compiler pushes the parameters, return address, and local variables of the called function on each entry of a function.

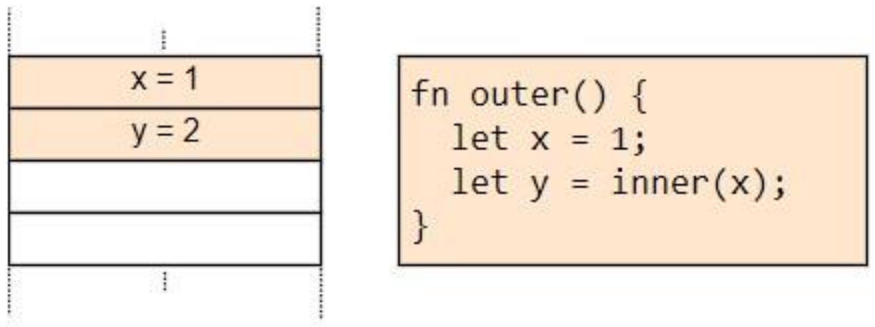


```
fn outer() {
  let x = 1;
  let y = inner(x);
}
```

```
fn inner(i: usize) -> u32 {
  let z = [1,2,3];
  z[i]
}
```

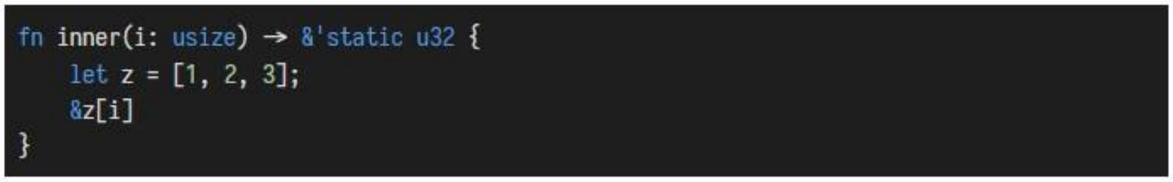
The call stack after the outer function called the inner function is displayed in the example above. The local variables of outer first are visible in the call stack. The parameter 1 and the function's return address were pushed on the inner call. Then inner received control and pushed its local variables.

Only the local variables of the outer function are left once the inner function returns since its call stack component is once more popped:



```
fn outer() {
  let x = 1;
  let y = inner(x);
}
```

We can see that the inner local variables are only valid while the function is running. When we attempt to return a reference to a local variable, for instance, the Rust compiler will enforce these lifetimes and throw an error if we use a value for too long:

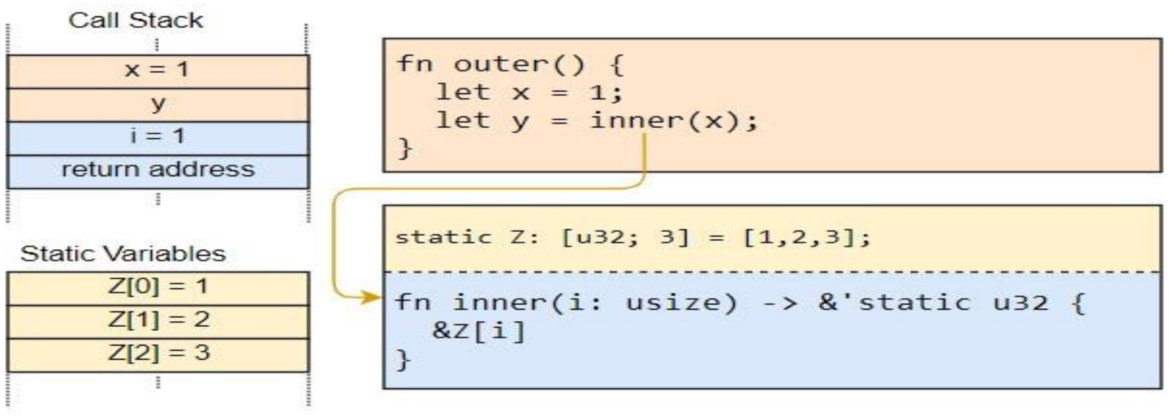


```
fn inner(i: usize) -> &'static u32 {
    let z = [1, 2, 3];
    &z[i]
}
```

There are situations where we want a variable to live longer than the function, even if in this example returning a reference makes little sense. The need to use a static variable to increase the lifetime occurred when we attempted to load an interrupt descriptor table in our kernel.

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 referenced from local variables because they last the entire duration of the program:



```
fn outer() {
  let x = 1;
  let y = inner(x);
}
```

```
static Z: [u32; 3] = [1,2,3];
```

```
fn inner(i: usize) -> &'static u32 {
  &Z[i]
}
```

In the example above, the call stack is cleared when the inner function returns. The &Z[1] reference remains valid after the return because the static variables are stored in a different memory region that is never deleted.

Static variables also benefit from the fact that their position is known at the time of compilation, eliminating the need for references when accessing them. We made use of the characteristic in our print ln macro: The internal use of a static Writer eliminates the requirement for a &mut Writer reference, which is extremely helpful in exception handlers where we don't have access to any more variables.

The fact that static variables are read-only by default is a significant downside of this feature. Rust imposes this because, for instance, if two threads updated a static variable concurrently, a data race might result. A static variable can only be changed by enclosing it in a Mutex type, which guarantees that only one &mut reference exists at all times. For our static VGA buffer Writer, we have already used a Mutex.

Dynamic Memory

The majority of use cases are already made possible by the combined power of local and static variables. We did see that each has some drawbacks, though:

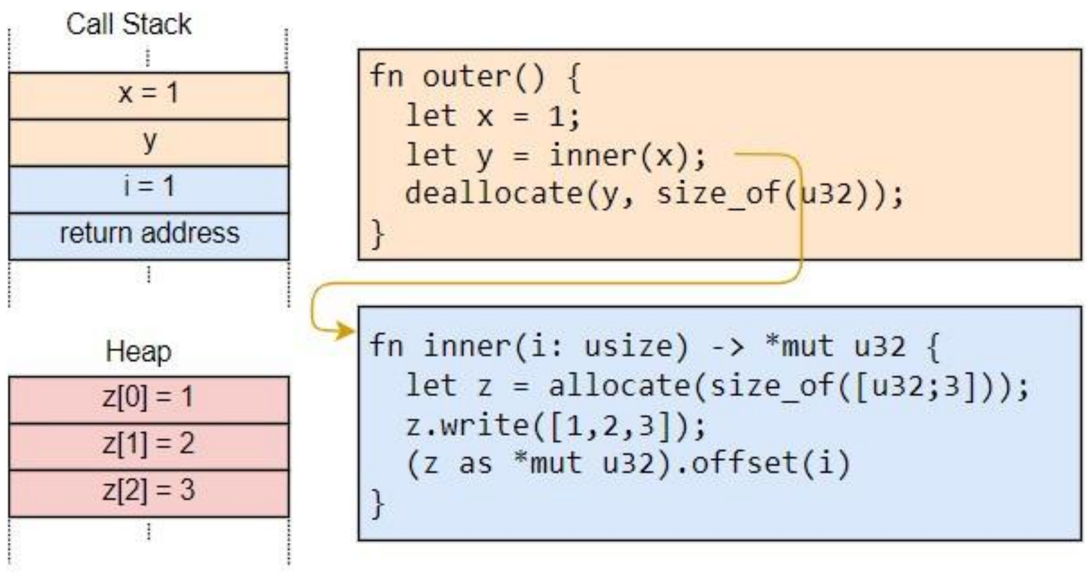
- Local variables only persist till the end of the function or block in which they are defined. This is the case because they exist on the call stack and are eliminated when the function they were residing in returns.

- There is no method to recover and reuse the memory used by static variables when they are no longer required because they always exist for the whole duration of the program. They must be protected with a Mutex when we want to edit them because they are accessible from all functions and have ambiguous ownership semantics.

The fixed size of local and static variables is another drawback. Therefore, they are unable to store a collection that expands as new elements are added. (Rust ideas for unsized values that would let dynamically sized local variables only function in a narrow range of circumstances.)

Programming languages frequently support the heap, a third memory area for storing variables, to get around these limitations. Through two methods called allocate and deallocate, the heap provides dynamic memory allocation at runtime. The way it operates is as follows: A free memory space of the provided size is returned by the allocate function and can be used to store a variable. When the deallocate function is used with a reference to the variable, the variable remains in existence until it is released.

Let’s go through an example:

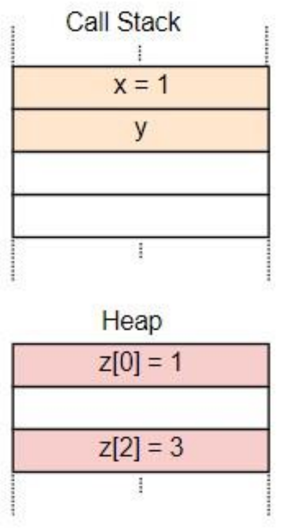


```
fn outer() {
  let x = 1;
  let y = inner(x);
  deallocate(y, size_of(u32));
}
```

```
fn inner(i: usize) -> *mut u32 {
  let z = allocate(size_of([u32;3]));
  z.write([1,2,3]);
  (z as *mut u32).offset(i)
}
```

Here, the inner function stores z in heap memory as opposed to static variables. A memory block of the necessary size is initially allocated, and this process then returns a *mut u32 raw pointer. After that, the array [1,2,3] is written to it using the ptr::write method. In the final step, it calculates a pointer to the i-th element using the offset function and then returns it. (Note that for the sake of brevity, we left out several necessary casts and unsafe blocks in this example code.)

The memory that has been deliberately allocated lives there until it is released by calling deallocate. Therefore, even after inner returned and its portion of the call stack was deleted, the returned pointer is still valid. When compared to static memory, employing heap memory has the advantage that it can be reused after it has been released. This is accomplished by using the deallocate call in outer. Following that call, the scenario is as follows:



```
fn outer() {
  let x = 1;
  let y = inner(x);
  deallocate(y, size_of(u32));
}
```

As we can see, the z[1] slot is once again free and available for the subsequent allocation call. But because we never deallocate z[0] and z[2], we also observe that they are never released. Such a flaw is known as a memory leak, and it frequently results in excessive memory usage by programs (just imagine what happens when we call inner repeatedly in a loop). Although this may appear unpleasant, dynamic allocation can lead to much more harmful forms of errors.

Common Errors

There are two typical bug categories with more serious repercussions, aside from memory leaks, which are regrettable but do not leave the software open to attackers:

- A use-after-free vulnerability occurs when we unintentionally continue to use a variable after running deallocate on it. A defect like this results in undefinable behavior and is frequently used by attackers to execute arbitrary code.
- We have a double-free vulnerability when we unintentionally free a variable twice. This presents a concern since it might release a different allocation that had been placed there following the initial deallocate request. As a result, it can result in another use-after-free vulnerability.

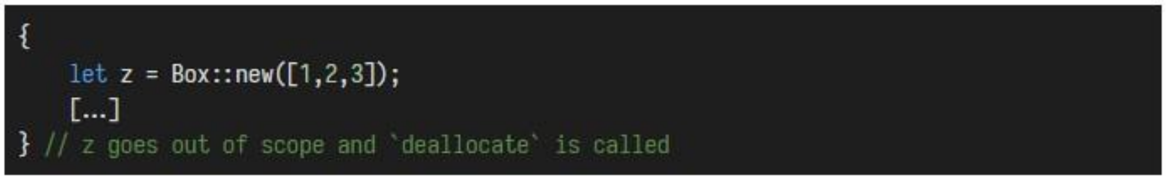
Given the widespread knowledge of these vulnerabilities, one could assume that people have already learnt how to prevent them. However, these flaws are still frequently discovered. Take, for instance, the Linux use-after-free vulnerability of 2019 that permitted arbitrary code execution. Use-after-free linux "current year" online searches will almost certainly always produce results. This demonstrates that even the finest programmers occasionally struggle to manage dynamic memory in intricate projects.

Many programming languages, like Java and Python, automatically manage dynamic memory by employing a method known as garbage collection in order to avoid these problems. The concept is that deallocate is never manually called by the programmer. Instead, unwanted heap variables are periodically found and automatically deallocated while the program is halted. As a result, the aforementioned vulnerabilities cannot exist. The normal scan's performance overhead and the sometimes-lengthy rest intervals are the disadvantages.

Rust approaches the issue differently. It makes use of a concept called ownership to ensure the accuracy of dynamic memory operations at build time. As a result, there is no performance overhead and the aforementioned vulnerabilities can be avoided without the need of trash collection. Another benefit of this strategy is that, like with C or C++, the programmer still has fine-grained control over the utilization of dynamic memory.

Allocations in Rust

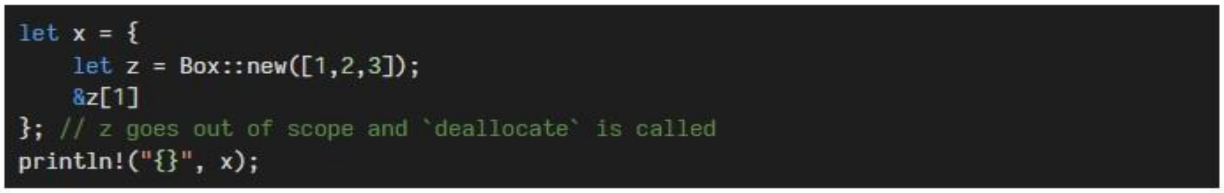
The Rust standard library offers abstraction types that implicitly call allocation and deallocate rather than allowing the programmer to individually invoke these functions. Box, an abstraction for a heap-allocated value, is the most crucial type. It offers a constructor function called Box: new that 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:



```
{
    let z = Box::new([1,2,3]);
    [...]
} // z goes out of scope and `deallocate` is called
```

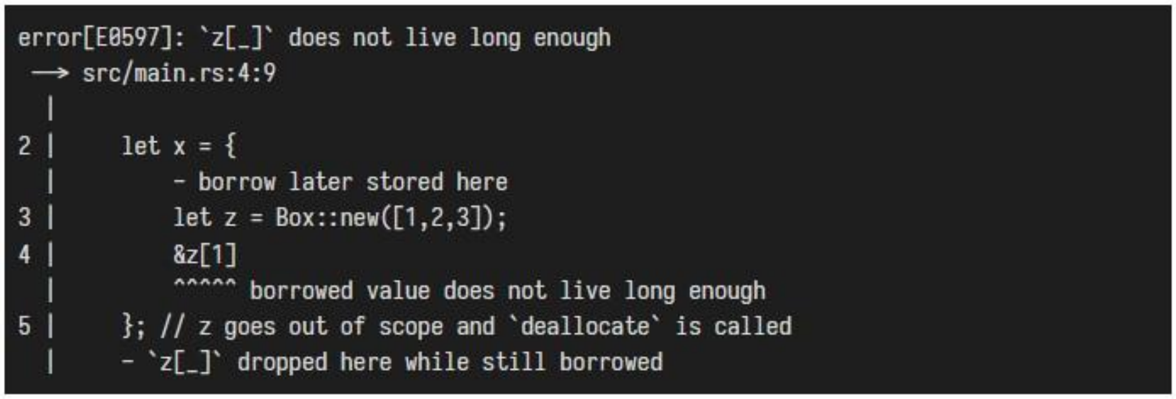
This pattern's oddly named commencement of resource acquisition (RAII). Its implementation of the std::unique ptr abstraction type, a related abstraction type, is where it first appeared in C++.

Since programmers can continue to hang on to references after the Box is out of scope and the accompanying heap memory slot is deallocated, such a type alone is insufficient to prevent all use-after-free issues.



```
let x = {
    let z = Box::new([1,2,3]);
    &z[1]
}; // z goes out of scope and `deallocate` is called
println!("{}", x);
```

Here's where Rust's ownership enters the picture. Each reference is given an abstract lifetime, or the range of time during which it is valid. Since the x reference in the aforementioned example is derived from the z array, it is rendered invalid when z is no longer in use. Running the aforementioned example on a playground demonstrates that the Rust compiler does, in fact, throw an error.



```
error[E0597]: `z[_]` does not live long enough
 --> src/main.rs:4:9
  |
2 |     let x = {
  |         - borrow later stored here
3 |         let z = Box::new([1,2,3]);
4 |         &z[1]
  |         ^^^^^ borrowed value does not live long enough
5 |     }; // z goes out of scope and `deallocate` is called
  |     - `z[_]` dropped here while still borrowed
```

At first, the language can be a little perplexing. Since taking a reference to a value is analogous to borrowing in real life, it is referred to as borrowing the value: A thing is temporarily in your possession, but you must return it at some point without damaging it. The Rust compiler ensures that there will never be a use-after-free situation by verifying that all borrowing has ended before an object is destroyed.

The ownership mechanism in Rust goes considerably farther, offering complete memory safety in addition to eliminating use-after-free problems, just as garbage collected languages like Java or Python. It also ensures thread safety, making it even safer in multi-threaded programs than those languages. Most crucially, there is no runtime overhead compared to manually implemented memory management in C because all these tests take place at compile time.

Use Cases

Now that we are familiar with the fundamentals of dynamic memory allocation in Rust, when should we use it? Without dynamic memory allocation, our kernel has advanced quite a bit; why do we suddenly require it?

First, because we must search for a spare slot on the heap before every allocation, dynamic memory allocation always entails some performance overhead. Local variables are therefore often preferred, especially in performance-critical kernel code. Dynamic memory allocation, however, may be the best option in some circumstances.

As a general rule, variables with a dynamic lifetime or a variable size require dynamic memory. Rc is the most significant type with a dynamic lifetime; it keeps track of all references to its wrapped value and deal locates it after all of those references have exited its scope. Vec, String, and other collection types that expand dynamically when new members are added are examples of types with a changeable size. When these types reach their capacity, they allocate more memory, copy all of the elements over, and then deallocate the previous allocation.
When developing multitasking in future posts, we will mostly require the collection types for our kernel to hold a list of active jobs, for example.

Allocator Designs

In this article, we'll go over how to build heap allocators from scratch. Different allocator designs, such as bump allocation, linked list allocation, and fixed-size block allocation, are shown and discussed. We will produce a fundamental implementation that can be applied to our kernel for each of the three designs.

Introduction

We improved our kernel's fundamental heap allocation support in the prior post. In order to do that, we added a new memory space to the page tables and managed it with the linked list allocator crate. While we now have a functional heap, we did not attempt to understand how the allocator crate operates and instead let it to do the bulk of the work.

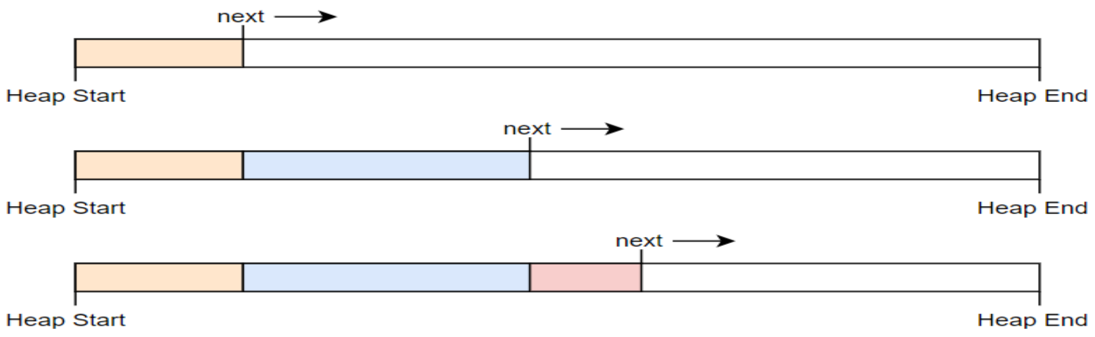
Instead of depending on an existing allocator crate, we will demonstrate in this post how to build our own heap allocator from scratch. We will go over various allocator designs, such as a straightforward bump allocator and a straightforward fixed-size block allocator, and use this information to create an allocator that performs better (than the linked list allocator crate).

Bump Allocator

A bump allocator is the most basic type of allocator (also known as stack allocator). It merely maintains track of the total number of bytes and allocations while allocating memory linearly. Because it has a serious restriction—it can only release full memory at once—it is only usable in extremely restricted use situations.

Idea

A bump allocator works by increasing (or "bumping") a next variable that corresponds to the beginning of the empty memory in order to linearly allocate memory. The start address of the heap is equal to next at the beginning. Next is always increased by the allocation size during each allocation such that it always denotes the boundary between used and unused memory:



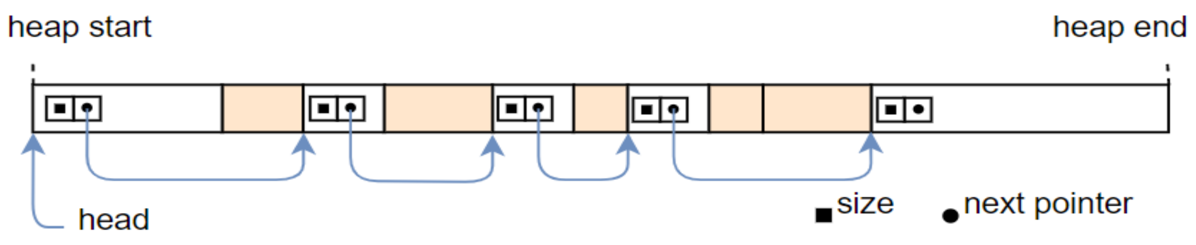
The next pointer never distributes the same memory region twice since it only flows in one direction. When the heap is full, no additional memory may be allocated, which causes an out-of-memory error on the subsequent allocation.

The allocation counter used to create a bump allocator is frequently incremented by 1 on each call to allow and decremented by 1 on each call to deadlock. All allocations on the heap have been deallocated when the allocation counter approaches zero. In this scenario, the next pointer can be reset to the heap's start address, making the entire heap memory once again available for allocations.

Linked List Allocator

When developing allocators, it's a standard practice to use the free memory areas as backing store to keep track of any number of available spaces. This takes advantage of the fact that the regions are still backed by a physical frame and mapped to a virtual address, but the stored data is no longer required. We can maintain track of an infinite number of liberated areas without using additional memory if we store the information about the freed region in the region itself.

The most typical way to implement it is to create a single linked list in the freed memory, where each node corresponds to a freed memory area:



The size of the memory region and a pointer to the following unused memory region are the two fields that each list node has. This method allows us to keep track of all unused regions, regardless of their quantity, by just maintaining a pointer to the first one (referred to as the head). The final data structure is frequently referred to as a free list.

The linked list allocator crate employs this method, as you could infer from the name. Pool allocators are another name for this type of allocator.

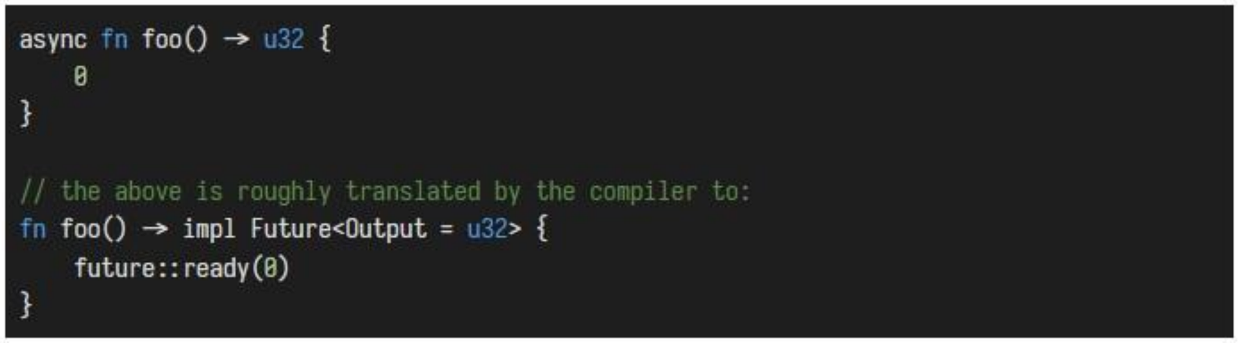
Fixed-Size Block Allocator

Following, we offer an allocator design that satisfies allocation demands by using fixed-size memory blocks. Due to internal fragmentation, the allocator frequently provides blocks that are bigger than what is required for allocations. However, compared to the linked list allocator, it significantly reduces the time needed to discover a suitable block, improving allocation performance.

The Async/Await Pattern

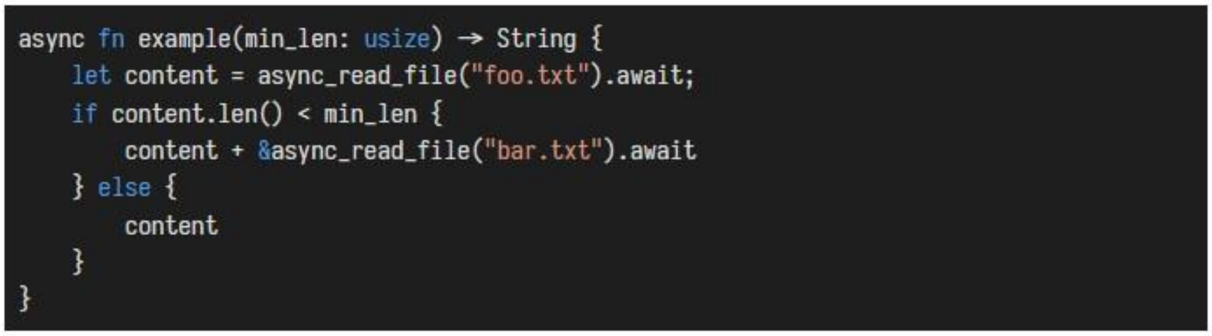
With async/await, the programmer can create code that appears to be typical synchronous code but is really generated as asynchronous code by the compiler. It operates using the two keywords await and async.

In a function signature, the async keyword can be used to transform a synchronous.



```rust
async fn foo() -> u32 {
    0
}

// the above is roughly translated by the compiler to:
fn foo() -> impl Future<Output = u32> {
    future::ready(0)
}
```

This keyword wouldn't be very helpful by itself. However, the await keyword inside of async functions can be utilized to get the asynchronous value of a future:

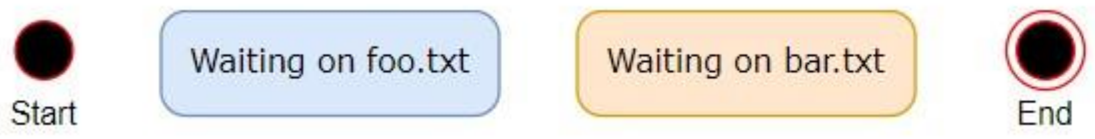


```rust
async fn example(min_len: usize) -> String {
    let content = async_read_file("foo.txt").await;
    if content.len() < min_len {
        content + &async_read_file("bar.txt").await
    } else {
        content
    }
}
```

The example function from above that uses combinator functions is directly translated into this function. We don't require closures or Either types to access the value of a future using the.await operator. As a result, we are able to write our code just how we normally write synchronous code, except that this code is still asynchronous.

State Machine Transformation

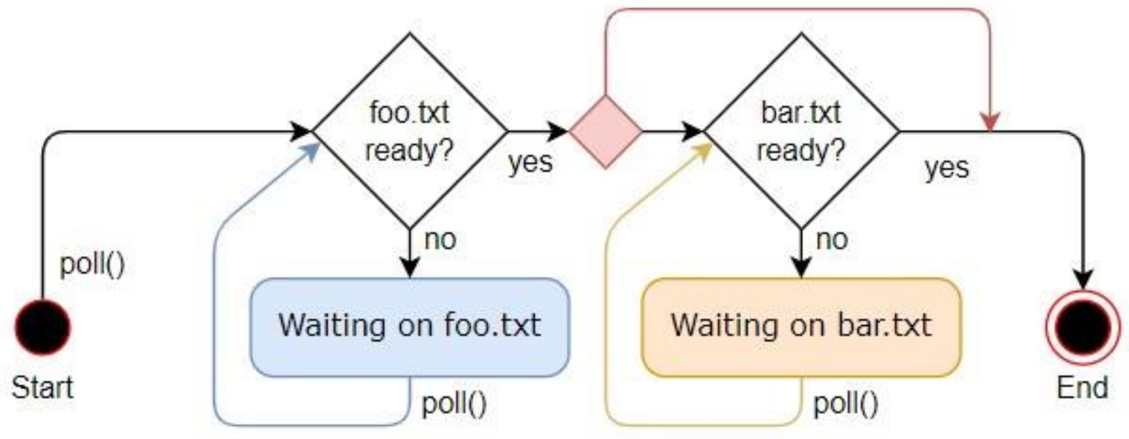
The async function's body is secretly transformed by the compiler into a state machine, with each.await call standing in for a different state. The compiler builds a state machine with the four states listed below for the aforementioned sample function:



Each stage corresponds to a separate function pause.

The function is shown in its "Start" and "Finish" states at the start and end of operation. The function's current state of "Waiting on foo.txt" indicates that it is awaiting the first async read file result. A similar pause point is represented by the "Waiting on bar.txt" state, when the function is awaiting the second async read file result.

Each poll that calls a potential state change by the state machine implements the Future trait:



The design employs diamond shapes to show alternate routes and arrows to depict state transitions. For instance, if the foo.txt file is not available, the path denoted by "no" is chosen, and the "Waiting on foo.txt" state is attained.

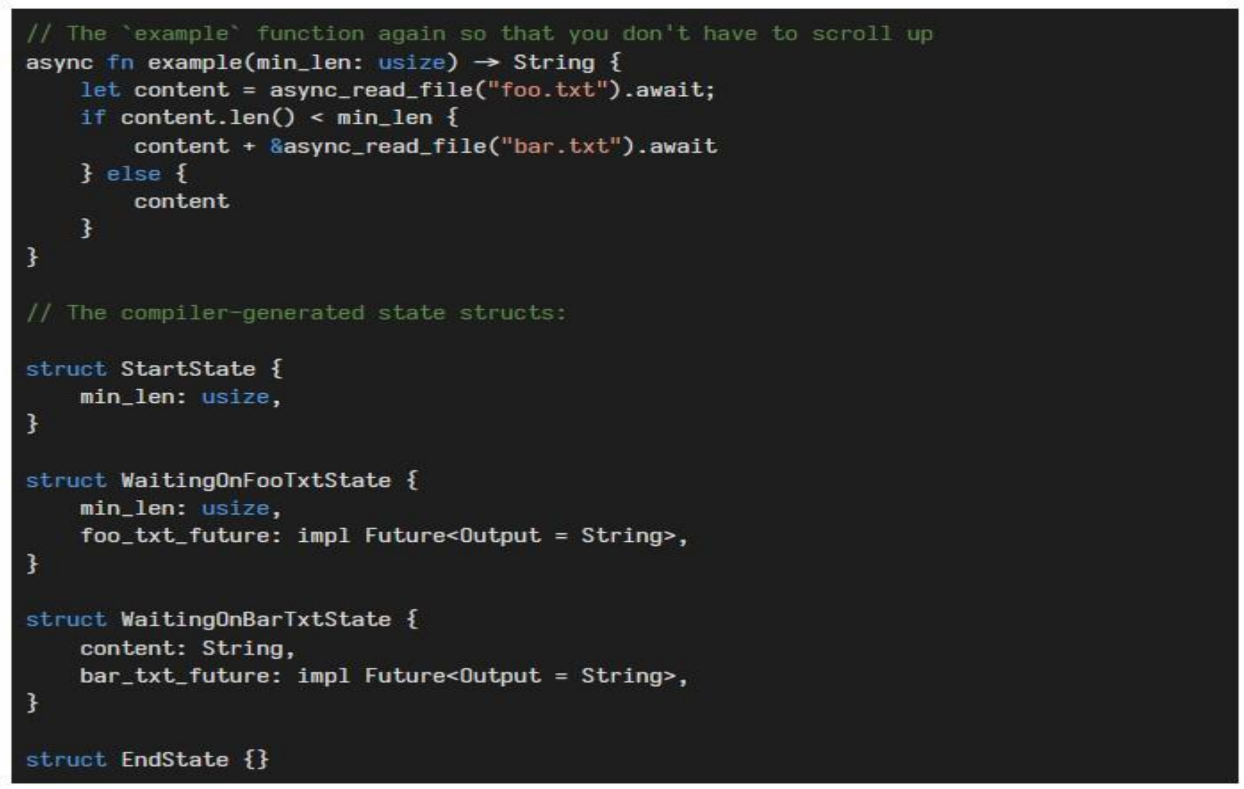
If not, the "yes" route is chosen. The if content is represented by the little red diamond without a caption. Branch of the example code where len() > 100.

As can be seen, the function is started by the initial poll call and allowed to run until it reaches an unprepared future. The function can continue to execute until the "End" state, at which point it returns its result wrapped in Poll::Ready if all futures on the path are ready. If not, the state machine returns Poll::Pending and enters a waiting state. The state machine then restarts from the most recent waiting state and tries the most recent operation on the subsequent poll call.

Saving State

The state machine must internally keep track of the current state in order to be able to proceed from the previous waiting state. Additionally, it needs to preserve all the variables it requires to keep running after the subsequent poll call. The compiler can actually excel in this situation: It can automatically create structs with the precise variables required since it understands which variables are used when.

For the function mentioned above as an example, the compiler creates structs that look like the ones below:



```rust
// The `example` function again so that you don't have to scroll up
async fn example(min_len: usize) -> String {
    let content = async_read_file("foo.txt").await;
    if content.len() < min_len {
        content + &async_read_file("bar.txt").await
    } else {
        content
    }
}

// The compiler-generated state structs:

struct StartState {
    min_len: usize,
}

struct WaitingOnFooTxtState {
    min_len: usize,
    foo_txt_future: impl Future<Output = String>,
}

struct WaitingOnBarTxtState {
    content: String,
    bar_txt_future: impl Future<Output = String>,
}

struct EndState {}
```

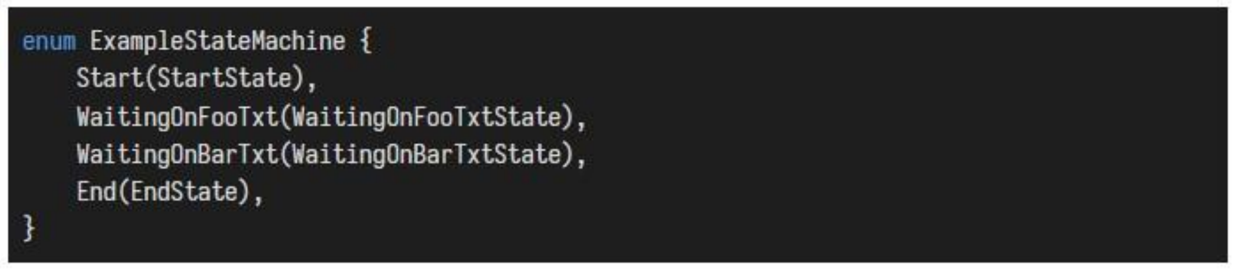
The min len option needs to be saved in the "start" and "Waiting on foo.txt" states for comparison with content.len afterwards (). Additionally, a foo txt future, which is a representation of the future returned by the async read file operation, is stored in the "Waiting on foo.txt" state. This future must be kept in order to be polled once more when the state machine begins.

The content variable for the later string concatenation, when bar.txt is prepared, is present in the "Waiting on bar.txt" stage. A bar txt future that reflects the bar.txt load that is now in progress is also stored. The min len variable is not present in the struct since it is no longer required following the content.len() comparison. No variables are kept in the "end" state because the function has already finished running.

Remember that this is merely an illustration of the type of code that the compiler might produce. The names of the structs and the arrangement of their fields may vary depending on the implementation.

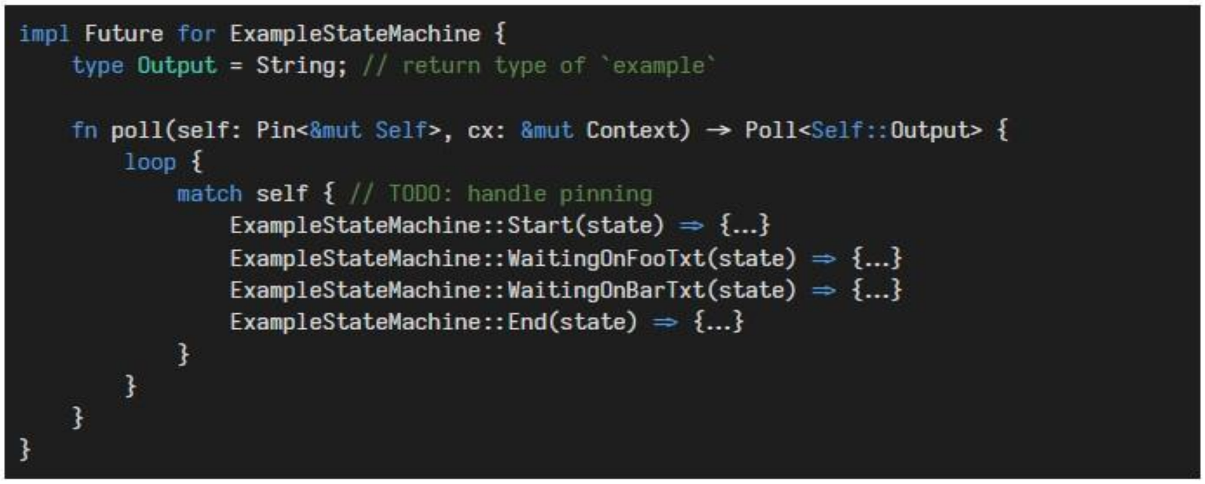
The Full State Machine Type

Even though the precise compiler-produced code is an implementation detail, seeing what the resultant state machine for the example function would look like aids with comprehension. The structs representing the various states and include the necessary variables have already been defined. We can merge them into an enum and build a state machine on top of it:



```
enum ExampleStateMachine {
    Start(StartState),
    WaitingOnFooTxt(WaitingOnFooTxtState),
    WaitingOnBarTxt(WaitingOnBarTxtState),
    End(EndState),
}
```

For each state, we define a unique enum version and include the relevant state struct as a field in each variant. The compiler creates an implementation of the Future trait based on the following example function to implement the state transitions:

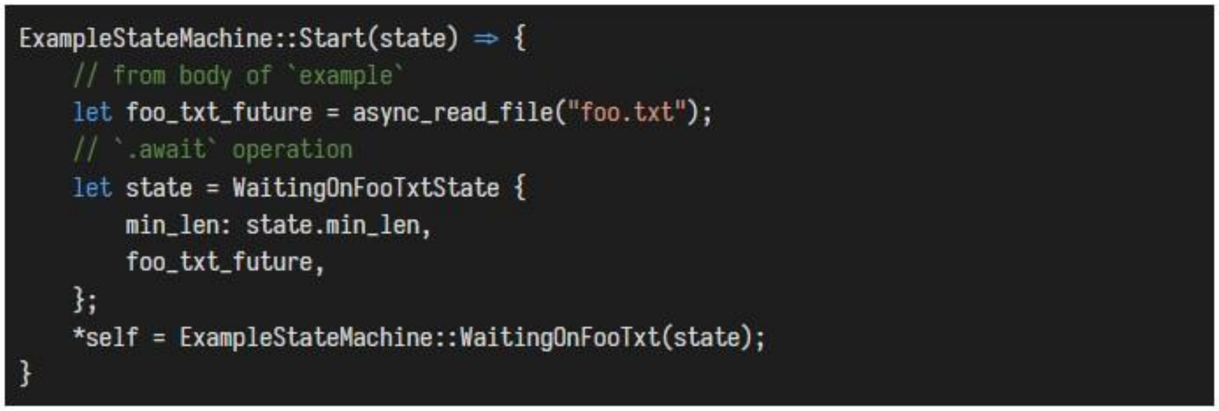


```
impl Future for ExampleStateMachine {
    type Output = String; // return type of `example`

    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output> {
        loop {
            match self { // TODO: handle pinning
                ExampleStateMachine::Start(state) => {...}
                ExampleStateMachine::WaitingOnFooTxt(state) => {...}
                ExampleStateMachine::WaitingOnBarTxt(state) => {...}
                ExampleStateMachine::End(state) => {...}
            }
        }
    }
}
```

Because String is the return type of the example function, it is the Output type of the future. We utilize a match statement inside of a loop on the current state to build the poll function. The goal is to use an explicit return when transitioning to the following state for as long as feasible. Poll:: When we are unable to go on, pending.

We merely display simplified code and ignore pinning, ownership, lifetimes, etc. because of convenience. As a result, this code as well as the ones that follow should only be used in a mock-up setting. Naturally, the actual compiler-generated code manages everything accurately, albeit maybe in a different way.

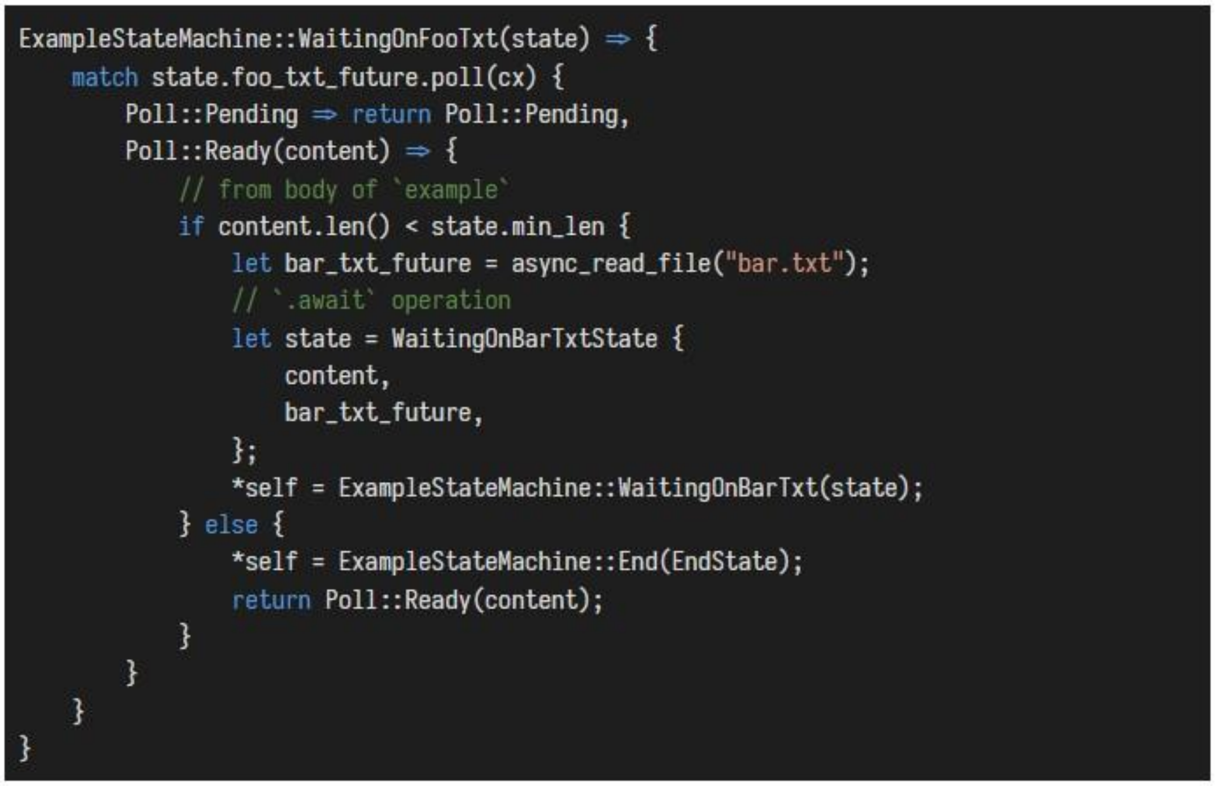
We show the code for each match arm separately to keep the code excerpts brief. Starting with the initial state



```
ExampleStateMachine::Start(state) => {
    // from body of `example`
    let foo_txt_future = async_read_file("foo.txt");
    // `.await` operation
    let state = WaitingOnFooTxtState {
        min_len: state.min_len,
        foo_txt_future,
    };
    *self = ExampleStateMachine::WaitingOnFooTxt(state);
}
```

When the function is just getting started, the state machine is in the Start state. In this instance, we run the entire sample function's body of code up until the first.await. We construct the WaitingOnFooTxtState struct and set the state of the self state machine to WaitingOnFooTxt in order to handle the.await action.

The WaitingOnFooTxt arm is the next to be executed because the match self... statement is run in a loop:



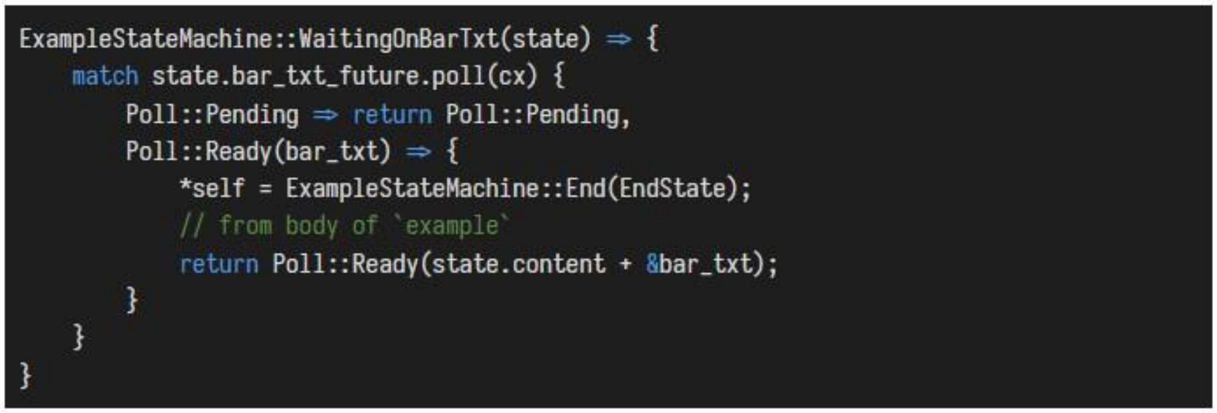
```rust
ExampleStateMachine::WaitingOnFooTxt(state) => {
    match state.foo_txt_future.poll(cx) {
        Poll::Pending => return Poll::Pending,
        Poll::Ready(content) => {
            // from body of `example`
            if content.len() < state.min_len {
                let bar_txt_future = async_read_file("bar.txt");
                // `.await` operation
                let state = WaitingOnBarTxtState {
                    content,
                    bar_txt_future,
                };
                *self = ExampleStateMachine::WaitingOnBarTxt(state);
            } else {
                *self = ExampleStateMachine::End(EndState);
                return Poll::Ready(content);
            }
        }
    }
}
```

We first call the poll function of the foo txt future in this match arm. If it is not prepared, the loop is ended and Poll::Pending is returned. The next poll call on the state machine will enter the same match arm and try polling the foo txt future again because self remains in the WaitingOnFooTxt state in this situation.

As soon as the foo txt future is prepared, we give the outcome to the content variable and go on running the example function's code: The bar.txt file is read asynchronously if content.len() is smaller than the min len value specified in the state struct. The.await operation is once more translated into a state change, this time into the WaitingOnBarTxt state. Due to the fact that the match is being run inside of a loop, the execution then immediately switches to the match arm for the new state, where the bar txt future is polled.

If we take the else branch, we must stop here. there is an await operation. When the function is complete, content returned is Poll::Ready-wrapped. Additionally, we switch the present state to the end state.

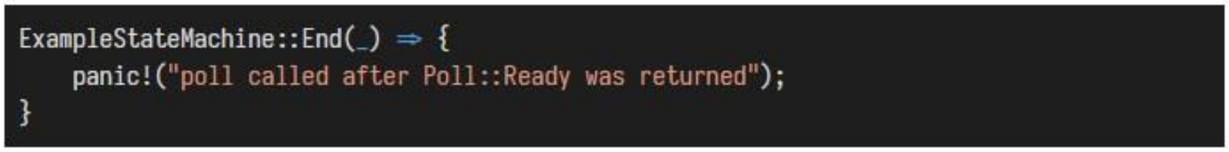
The WaitingOnBarTxt state's code appears as follows:



```
ExampleStateMachine::WaitingOnBarTxt(state) => {
    match state.bar_txt_future.poll(cx) {
        Poll::Pending => return Poll::Pending,
        Poll::Ready(bar_txt) => {
            *self = ExampleStateMachine::End(EndState);
            // from body of `example`
            return Poll::Ready(state.content + &bar_txt);
        }
    }
}
```

We start by polling the bar txt future, just like in the WaitingOnFooTxt state. We stop the loop and return Poll::Pending if it is still pending. If not, we can execute the final step of the sample function, which is to concatenate the content variable and the result from the future. We return the result wrapped in Poll::Ready after updating the state machine to the End state.

Lastly, the code for the End state appears as follows:

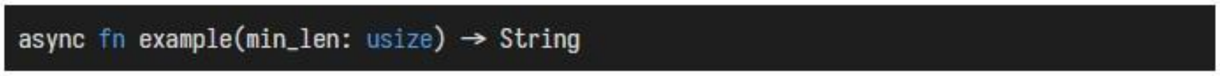


```
ExampleStateMachine::End(_) => {
    panic!("poll called after Poll::Ready was returned");
}
```

After futures have returned, polls shouldn't be conducted again. Poll:: When a poll is called while we are already in the End state, we become panicked.

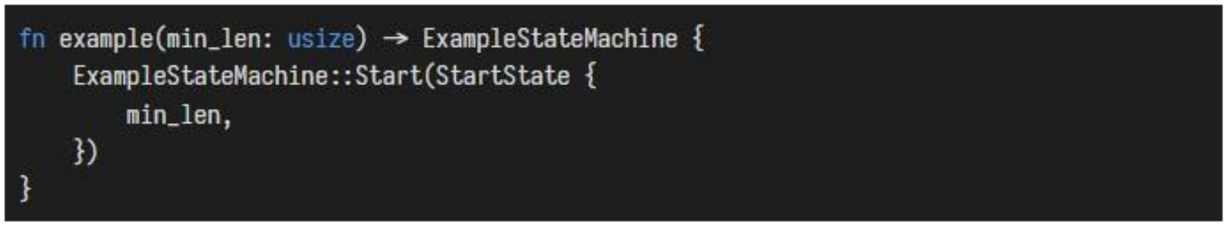
Now that we know what the Future trait implementation of the compiler-generated state machine might look like. The compiler produces code differently in practice. (If it's of interest, the implementation currently relies on generators; nevertheless, this is merely an implementation detail.)

The produced code for the example function itself completes the picture. Keep in mind how the function header was defined:



```
async fn example(min_len: usize) -> String
```

The state machine has now finished implementing the entire function body, so all that is left for the function to do is initialize it and return it. This could result in code that looks like this:



```rust
fn example(min_len: usize) → ExampleStateMachine {
    ExampleStateMachine::Start(StartState {
        min_len,
    })
}
```

Since the function now explicitly returns an ExampleStateMachine type that implements the Future trait, the async modifier has been removed from the function. The state machine is built in the Start state, as expected, and the min len parameter is used to initialize the matching state struct.

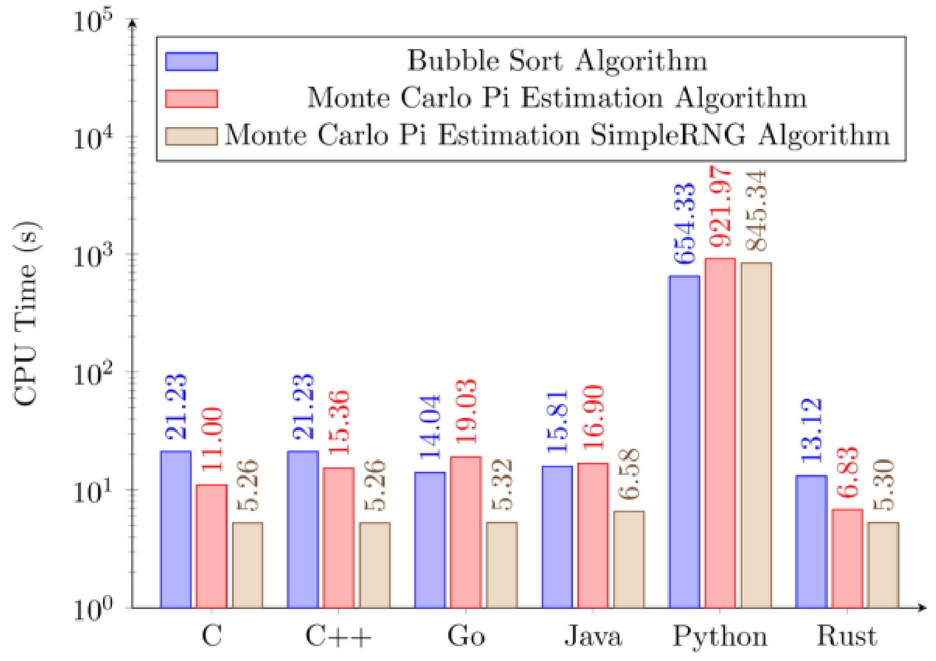
Keep in mind that this function does not initiate the state machine's execution. Futures in Rust are fundamentally designed so that they do nothing prior to being polled for the first time.

Performance

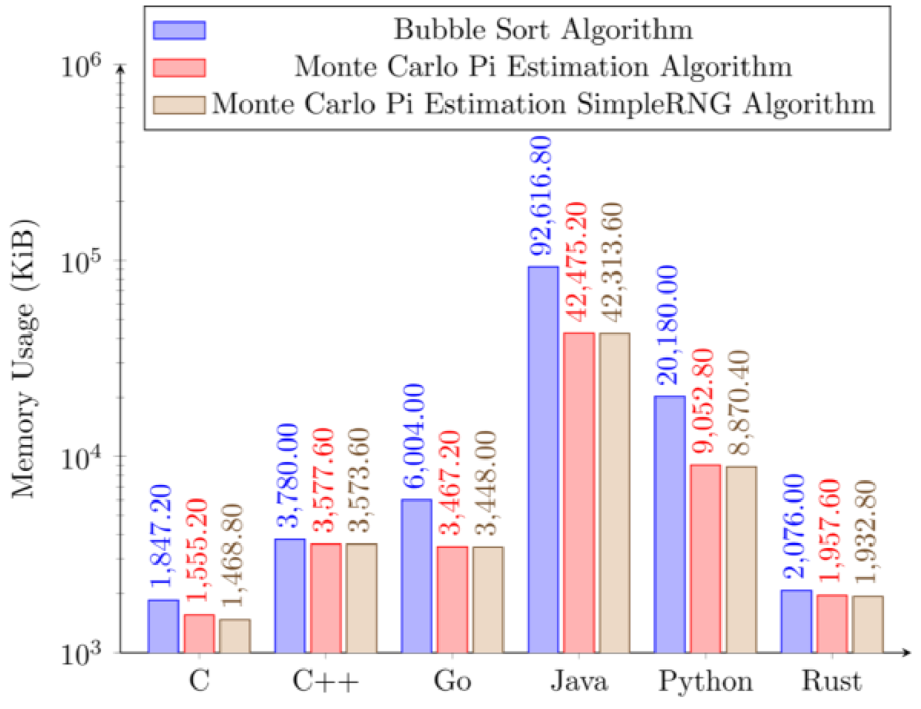
Rust uses the idea of zero cost abstraction, as was described in the introduction, to make the language simpler without sacrificing efficiency. Monomorphization is a straightforward illustration of this concept; it enables the creation of generic functions that are transformed into the required concrete type functions at compile time, incurring no runtime overhead. Additionally, this idea is used in the standard library to ensure that common types like collections don't need to be recreated, resulting in improved library speed and compatibility.

The absence of a trash collector, which has numerous advantages for performance, is the second noteworthy advantage of Rust. Runtime overhead is added by garbage collection since it needs to keep track of the memory in some way (often through reference counting) in order to know when memory can be released. This raises both memory utilization and CPU usage, which is crucial for embedded devices with less memory (important for performance critical software). The fact that trash collection is more difficult to regulate, causing unexpected execution pauses while it is operating and/or removing unwanted resources, is another significant concern with it.

The figures below, which show experimental performance findings of several benchmarks for six languages, including Rust, for two algorithms, are based on information from the chart below. For a more accurate comparison than the second test, which used the language's standard RNG (Random Number Generator), the third test employed a standard RNG implemented for each language because the Monte Carlo Pi estimation was based on an RNG. The results show that Rust leads in CPU time for two of the three algorithms and trails only marginally in the third test to C and C++. In terms of memory utilization, Rust trails only C across all tests.



1. **Average CPU time benchmark results**



2. **Average memory usage benchmark results**

Reliability

If you're a programmer, you already know how challenging it is to keep your code memory secure and prevent errors from appearing just because you handled memory incorrectly. Rust, on the other hand, is a language that provides us with dependability while writing codes in accordance with a few guidelines that are incorporated into the language's design.

- Problems with other popular languages like C, C++, or java for a variety of reasons:
- Creating a Null variable, and then accessing it assuming it’s not Null.
- Changes of a variable’s value which are beyond our control.
- Overflowing.

We can declare a variable first by a specified type, just like we did in databases, when we know that it may be Null/None or when we want a variable to occasionally have no value at all. This is done in Rust by giving the variable the keyword "Choice." It means that when you use the "Choice" keyword to define a variable, you are informing the compiler that the variable can hold the value of None. As of this point, the compiler requires you to handle and check the None value anytime you examine that variable in an "if" blockader another place comparable. A variable cannot have the value of nothing if it is defined normally.

Rust also offers a solution to the opposite issue of being changed without being acknowledged and utilizing a variable under the belief that it has the most recent valid value. And the brilliant response to that is as follows: By default, none of the variables can be changed. It must be defined as a mutable variable if you want to be able to alter it afterwards. Otherwise, you won't be able to give this variable any new values in the future. You can pass a variable to one modifier in Rust and then return it. Obviously once you give something to someone else, you are unable to use it again until you get it back.

Productivity

Rust produces a lot of things. In addition to having excellent documentation and a helpful compiler, Rust also. Additionally, Rust features an excellent build tool, a built-in package management, and clever multi-editor support.

Rust uses the idea of zero cost abstraction to make things simpler. Without hindering performance, the language. Monomorphization is a straightforward illustration of this concept (For each distinct instantiation, multiple monomorphic functions are substituted for polymorphic functions during the compile-time process of programming languages.); it enables the creation of generic functions that are transformed into the required concrete type functions at compile time, incurring no runtime overhead. Additionally, this idea is used in the standard library to make sure that common types like collections don't need to be recreated, resulting in improved library speed and compatibility.

The absence of a trash collector, which has several advantages for performance, is the second noteworthy advantage of Rust. Runtime overhead is added by garbage collection since it has to keep track of the memory in some way (often by reference counting) in order to know when memory may be released. This raises both memory utilization and CPU usage, which is crucial for embedded systems with less memory (important for performance critical software). Another significant issue with garbage collection is that it is more difficult to manage, which can result in unanticipated delays in execution while it is running and/or freeing up resources that are no longer needed.

Although Rust's syntax is similar to C++'s, it is faster and more memory-safe. Due to the prohibition of null and dangling pointers, Rust is a more advanced system-level language in terms of better memory management.

Security

The security of software produced in a language and the safety of that language have a lot in common. The Microsoft Security Response Center believes that memory exploits are to blame for 70% of the issues they categorize as common vulnerabilities and exposure (CVE). Obviously, this indicates that preventing memory exploits is essential. Memory safety can get rid of up to 70% of security flaws.

Memory safety flaws encompass a wide range of various problems. . Any safety-related bug's fundamental drawback is that they all produce undefined behavior (UB), which leaves anything possible to happen, including security flaws.

In languages like C that use manual memory management, use after free issues are most prevalent.All that is required to generate one is to temporarily release a memory place that is still required. There are a variety of outcomes that can occur when memory is accessed after it has been released, and each will rely on the operating system and hardware. Sending data to the location of released memory might result in a security risk since the memory could then be utilized for other variables after it has been freed. However, the application will encounter a segmentation fault and execution will be suspended if the operating system determines that the address is incorrect after it has been released. Rust does not generally have this issue since it forbids manual memory management outside of the infrequently used unsafe block, and it is considerably simpler to guarantee the accuracy of a tiny portion of code than for the whole codebase.

When trying to dereference a null pointer, null pointer dereferencing happens. This error has been dubbed "the billion-dollar mistake" since it typically leaves systems in an undefinable state rather than producing an outright failure, which can result in any variety of issues. When a software tries to copy data from one memory region to another, the source is a null pointer. Whatever is copied—whether it be just all zeros or random bits from address 0—will have side effects that propagate throughout the rest of the execution, perhaps leading to security flaws and unpredictable behavior that is hard to analyze. Rust resolves this issue by creating an alternative type called Option to replace all null values. Represent values that might be uninitialized. This increases the level of security and safety and since it's frequently essential to define a variable before using it, the language's ergonomics initialization.

Due to the possibility that the address being attempted to be released a second time may already be in use for another variable, double frees can result in issues similar to usage after frees. The second possibility is that the address isn't being used again and the behavior isn't known. As previously indicated, UB can result in security flaws, but it is challenging to foresee how as all UB depends on the particular compiler, operating system, and hardware.

Anything other than C++ might be seen to be more secure. Better than C++ is pretty much everything. The truth is that many issues that C++ has never addressed are being thoroughly studied by Rust. Because C++ was always attempting to be compatible with C, its capabilities were severely limited. That was C++'s primary mistake. Setting boundaries and then enforcing them is the fundamental tenet of security. The fundamental concept of C and C++ presume that programmers may just breach boundaries whenever they feel like it. C and C++ are weak on limits. The 'trust the coder' mentality has led to inadequate security (and many other problems). Trust the programmer was naive in the past.

Conclusion

We have demonstrated that Rust is a viable language option for HPC applications, both in terms of its features, which are on par with or better than those of Fortran and C++, and in terms of its performance. We think the additional safety assurances the language offers are quite helpful, and we hypothesize that they would make it possible to execute more aggressive optimizations, which would improve runtime speed.

Rust's support for GPU programming, however, is not fully realized, as we have discovered. Although of widespread interest to the Rust community, we believe that achieving GPGPU programming while upholding Rust's safety guarantees would be challenging. We have a hard time imagining how this will alter in the future. Rust may still be used for host code, and we don't notice any issues with calling GPU kernels created in other languages from Rust.

We see promise in Rust's hygienic macro systems to produce safe and efficient code in addition to better error signals. We may see employing procedural macros or compiler plugins to construct an OpenMP-like, directive-driven parallelization paradigm in the current Rust release. However, the scientific community doesn't appear to be aware of the benefits that Rust offers at the moment, hence the creation of features for scientific computing in Rust is going very slowly. We hope that this thesis will serve as a starting point for the community to explore programming in Rust, express interest in, and contribute to the Rust ecosystem and project.

References

https://medium.com/sliit-foss/how-rust-can-be-used-to-implement-a-better-operating-system-fe1d4327156c

https://medium.com/sliit-foss/how-rust-can-be-used-to-implement-a-better-operating-system-part-2-5b47fc33674

https://en.wikipedia.org/wiki/Rust_(programming_language)

https://www.rust-lang.org/

https://medium.com/@hessam_kk/how-rust-programming-language-provides-reliable-coding-85b2c423611b

https://codilime.com/glossary/rust/

https://dev.to/pancy/boost-productivity-with-rust-anf

https://os.phil-opp.com/

Contribution Of Group Members

Student Name	IT Number	Contribution
K.D.S.P. Jayawikrama	IT21170720	• Testing • CPU Exceptions • Double Faults • productivity • security
H.C.K. Ariyarathna	IT21176388	• Abstract • Introduction • Hardware Interrupts • Introduction to Paging • Paging Implementation
Liyanage P.P.	IT21184758	• A Freestanding Rust Binary • A Minimal Rust Kernel • VGA Text Mode • performance • Reliability
Samaranayake Y.C	IT21180316	• Heap Allocation • Allocator Designs • Async/Await • conclusion