
# IE2050
# Operating Systems
# 2nd Year, 2nd Semester

IE2050 – Assignment

# Rust Language

Submitted to

Sri Lanka Institute of Information Technology

In partial fulfilment of the requirements for the

Bachelor of Science Special Honors Degree in Information Technology

10th October 2020

## Declaration

I certify that this report does not incorporate without acknowledgement, any material previously submitted for a degree or diploma in any university, and to the best of my knowledge and belief it does not contain any material previously published or written by another person, except where due reference is made in text.

Registration Number: IT19082066

Name: S D Kasthuriarachchi

**Table of Content**

**Table of Figures**

The detailed video of this report available at Google Drive. To access click the link below,

https://drive.google.com/file/d/1Jx0aPYICewFPLptCxkfRbckT4yAbXEc_/view?usp=sharing

## 1.Introduction

Rust is a programming language with multiple paradigms that focuses on performance and safety, mostly safe competition. Rust is syntactically like C++ and provides memory protection through the use of garbage collection, but instead using a borrowing framework. Rust is blazingly fast, and it can power performance-critical services, operate on embedded devices and easily integrate with other languages without a runtime or garbage collector.[1]

Rust guarantees memory-safety and thread-safety, allowing compile-time to remove several classes of bugs. Rust has excellent documentation and a friendly compiler with helpful error messages.

The installation of the Rust is straightforward; it can download via Rust online website (Figure 1.1). Then the Rust installer and version management tools will be installed.[1]

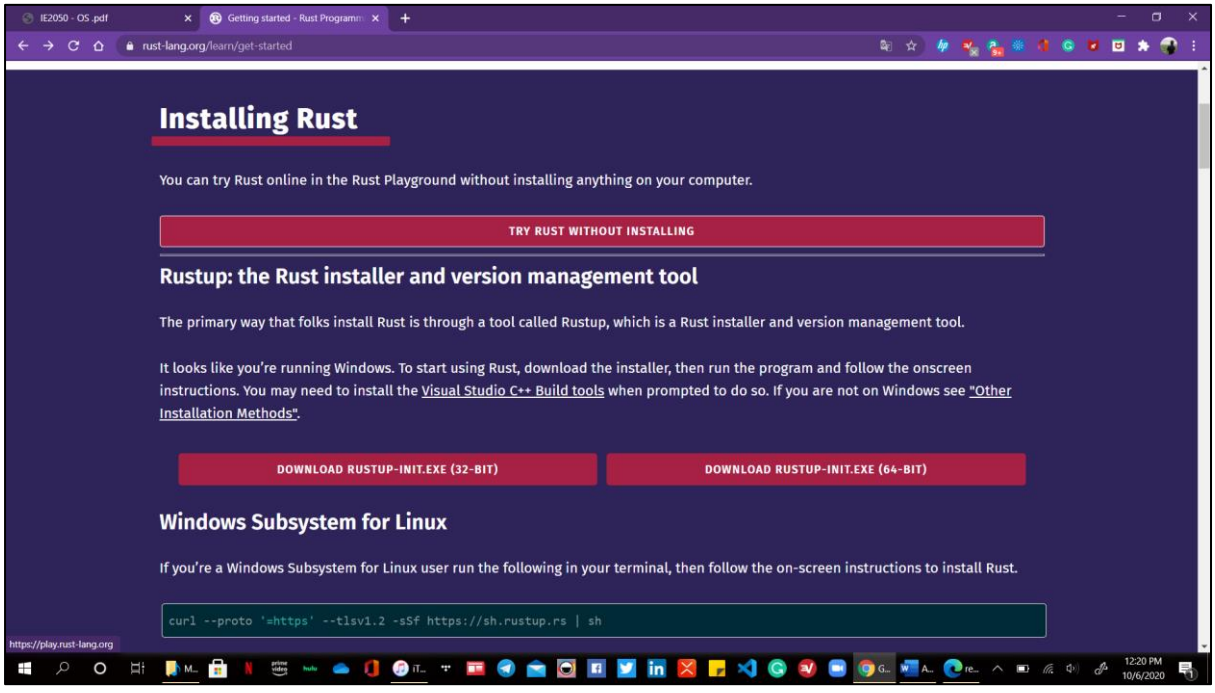


Figure 1.1: Rust Language Downloading Website

Rust is meant to be a language for highly concurrent and highly secure systems, and large-scale programming, that is, creating and maintaining limits that preserve the integrity of the extensive structure.[1]

## 2.Analysis

In here talk about, how the modern operating systems are created using rust language and its features for it.

### 2.1 Freestanding Rust Binary

The first step in creating the kernel of the operating system is to generate a Rust executable that does not belong to the standard library. It allows the bare metal to run its Rust code without an underlying operating system.

User need code that does not rely on any operating system features to write an operating system kernel. It means that threads, files, heap memory, the network, random numbers, standard output, or any other features that require hardware or OS abstractions cannot be used. The reason for that is the user trying to write his OS and his drivers.[2]

That means much of the Rust standard library cannot be used, but there are a lot of Rust features that can use. For example, Rust has features like iterators, closures, pattern matching, option and result and string formatting.

Users need to create an executable that can be run without the underlying operating system to develop an OS kernel in Rust. It is called a “freestanding” or “bare-metal” executable.

A minimal freestanding Rust binary look like following figures,

```rust
#![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 2.1.1: src/main.rs[2]

```toml
[package]
name = "crate_name"
version = "0.1.0"
authors = ["Author Name <author@example.com>"]

# the profile used for `cargo build`
[profile.dev]
panic = "abort" # disable stack unwinding on panic

# the profile used for `cargo build --release`
[profile.release]
panic = "abort" # disable stack unwinding on panic
```

Figure 2.1.2: Cargo.toml[2]

For compile type the following code[2],

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

## 2.2 Minimal Rust Kernel

In previously user builds the freestanding Rust binary. Next part is to create the minimal kernel. To do that first need to understand the BIOS and power-on self-test (POST) in the computer.[3]

After the progress mentioned above, the user can create a minimal kernel. The main task is to create a disk image that prints a “Hello World!” when the computer booted.

Previously used cargo (Figure 2.1.2) to build the freestanding binary, as cargo makes by default for the host device. That is not something that the kernel needs, so it does not make any sense for a kernel that runs on top of Windows. Instead, the user wants to compile a target system that is specified.[3]

Installing Rust Nightly[3]

There are three release channels for Rust: stable, beta, and nightly. By using “rustup” to control Rust Installations. It helps users to install side-by-side nightly, beta, and stable compilers and makes it simple to update them.

Target Specification[3]

Via the “--target” parameter, cargo supports different target systems. A so-called target triple, which describes the CPU architecture, the vendor, the operating system, and the ABI, defines the target. Many distinct target triples are supported by Rust, including “arm-linux-androideabi” for Android or “wasm32-unknown-unknown” for WebAssembly.

In here some unique configuration parameters are needed. Luckily, Rust allows the target to be specified via a JSON file. Next is to create the target specification. It is a “.jason” file.

```
{
    "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
}
```

Figure 2.2.1: .jason File[3]

After adding the build-related entries, ".jason" file looks like the following figure.

```
{
  "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 2.2.2: .jason File after adding entries[3]

Building our Kernel[3]

It will use the Linux conventions to compile the new target. It implies that an entry point named start is needed. Next, bypassing the name of the JSON file as "--target", create the kernel for the new target.

```
> cargo build --target x86_64-blog_os.json
```

If the above command fails, the user needs to edit "build-std" option.

Printing to Screen[3]

At this point, the VGA text buffer is the easiest way to print text on the screen. It is a particular area of memory-mapped to the hardware of the VGA that holds the contents shown on the screen. To print "Hello World!", only need to know that the buffer is located at address 0xb8000 and that there are an ASCII byte and a colour byte in each character cell. The implementation appears to be following,

```rust
static HELLO: &[u8] = b"Hello World!";

#[no_mangle]
pub extern "C" fn _start() -> ! {
    let vga_buffer = 0xb8000 as *mut u8;

    for (i, &byte) in HELLO.iter().enumerate() {
        unsafe {
            *vga_buffer.offset(i as isize * 2) = byte;
            *vga_buffer.offset(i as isize * 2 + 1) = 0xb;
        }
    }

    loop {}
}
```

Figure 2.2.3: Implementation[3]

Running the Kernel[3]

After the above progress user has an executable that does something noticeable, it is time to run it. First, by linking it with a bootloader, the user needs to convert the compiled kernel into a bootable disk image. Then, in the QEMU virtual machine, the user can run the disk image or boot it on real hardware using a USB stick. After Bootup the QEMU, it looks like follows.

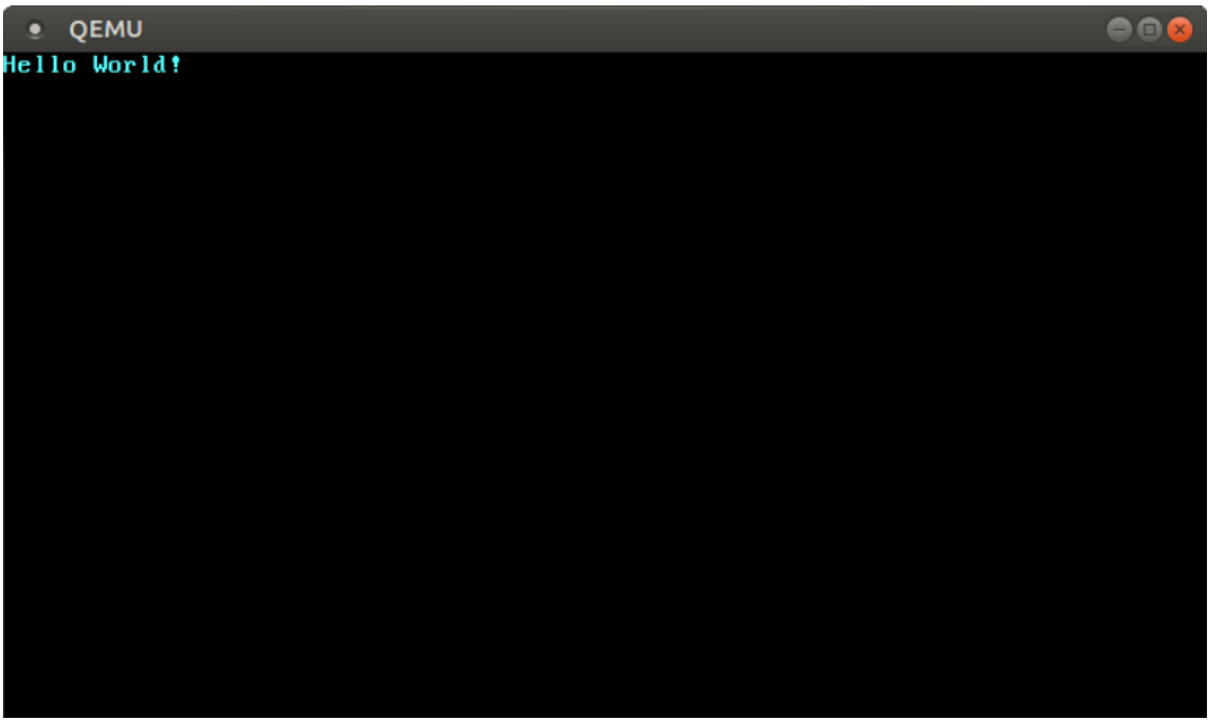


Figure 2.2.4: QEMU[3]

### 2.3 VGA Text Mode

The text mode of the VGA is a convenient way to print text on the screen. In here develop an interface, that makes it secure and easy to use by encapsulating all vulnerability in a separate module.

The VGA Text Buffer[4]

In order to print a character on a screen in VGA text mode, it must be written to the VGA hardware text buffer. A two-dimensional array of usually 25 rows and 80 columns is the VGA text buffer, which is rendered directly to the screen. Using the following format, each array entry defines a single screen character:

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

The first byte is the character to be printed in the ASCII encoding. The second byte specifies the display of the character.

The VGA text buffer is accessible at the address 0xb8000 via memory-mapped I/O. This implies that the text buffer on the VGA hardware does not access the RAM, but explicitly reads and writes to that address.

A Rust Module[4]

First, need to create rust module to handle printing. It looks like following,

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

Build a new src / vga buffer.rs file for the content of this module. The code below goes all the way into a new module.

```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,
}
```

Figure 2.3.1: src/vga_buffer.rs file[4]

Text Buffer[4]

Now add structures as follows to represent a screen character and the text buffer,

```rust
// 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],
}
```

Figure 2.3.2: Adding structures[4]

Since the field ordering in default structures is undefined in Rust, the attribute of repr(C) is required by the user. It ensures that the fields of the structure are set out exactly as in a C structure and thus guarantees the proper field ordering.

To write to the screen, create a writer-type like following,

```
// in src/vga_buffer.rs

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

Printing[4]

Users will now use the Writer to change characters in the buffer. Create a method for writing a single ASCII byte first like following,

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

impl Writer {
    pub fn write_byte(&mut self, byte: u8) {
        match byte {
            b'\n' => self.new_line(),
            byte => {
                if self.column_position >= BUFFER_WIDTH {
                    self.new_line();
                }

                let row = BUFFER_HEIGHT - 1;
                let col = self.column_position;

                let color_code = self.color_code;
                self.buffer.chars[row][col] = ScreenChar {
                    ascii_character: byte,
                    color_code,
                };
                self.column_position += 1;
            }
        }
    }

    fn new_line(&mut self) {/* TODO */}
}
```

Figure 2.3.3: Modifying the Buffer's Characters[4]

The VGA text buffer supports only ASCII and page 437 for additional bytes of code. By default, Rust strings are UTF-8, so they might contain bytes that the VGA text buffer does not support.

## 2.4Testing

Rust can use custom test frameworks to test functions within the kernel to support them. Using the various features of QEMU and the boot image tool to monitor results from QEMU.

Testing in Rust[5]

Rust has an integrated test system that is capable of running unit tests without the need to set up anything. Only build a function that tests some of the results by assertions and add the # [test] attribute to the function header. Then the cargo test will automatically find and perform all the test functions that have been developed.

The issue is that the Rust Test System implicitly uses the built-in test library, which relies on the standard library. This ensures that the default test system for the # [no std] kernel cannot be used.

Custom Test Frameworks[5]

Rust supports replacing the default test framework with an insecure custom test frameworks feature. It works by collecting all functions annotated with the # [test case] attribute and then invoking a user-specific runner function with the test list as an argument. It, therefore, gives the execution full power over the testing process.

## 2.5 CPU Exceptions

In different erroneous cases, CPU exceptions occur, such as when accessing an invalid memory address or when dividing by zero. An interrupt table of descriptors provides handler functions for reacting to them. It would be possible to grab breakpoint exceptions at the end of the kernel and restart regular execution afterwards.[6]

An exception indicates that the current instruction is wrong. For instance, if the current directive divides by 0, the CPU issues an exception. If an exception happens, the CPU will interrupt its existing work and call an exception handler feature immediately, depending on the exception type.[6]

There are about 20 distinct CPU exception types on x86. The most critical ones are,

- Page Fault
- Invalid Opcode
- General Protection Fault
- Double Fault
- Triple Fault

Users must set up a so-called Interrupt Descriptor Table (IDT) to record and manage exceptions. The user may define a handler function for each CPU exception in this table. The hardware uses this table explicitly, so the user needs a predefined format to follow. There must be a 16-byte structure for each entry.

In the implementation, the user needs to handle CPU exceptions in the above-created kernel. It starts with creating a new src / interrupts.rsr interrupt module. Now the user can add handler functions.

After the above process user can run it. The Processor effectively invokes the breakpoint handler that prints the message.[6]

### 2.6 Double Faults

Double faults happen when the CPU does not invoke an exception handler. Prevent fatal triple faults, triggering a system reset by handling this exception. Set up an Interrupt Stack Table to capture double faults on a separate kernel stack to avoid triple faults in all situations.

A double fault is a notable exception in more straightforward language, which happens when the CPU fails to invoke an exception handler. For example, it happens when a page fault is triggered, but the Interrupt Descriptor Table (IDT) does not have a page fault handler recorded. However, it is close in programming languages to catch-all blocks with exceptions.

e.g. catch(...) in C++ or catch(Exception) in Java or C#[7]

A double fault behaves like a typical exception. It has the number 8 vector and can specify in the IDT a standard handler function for it. It is essential to have a double fault handler since a fatal triple fault occurs if a double fault is unhandled. It is difficult to capture triple faults, and most hardware responds with a system reset.[7]

Triggering a Double Fault[7]

To provoke a double fault, use “unsafe” keyword in invalid address 0xdeadbeef. It shows in the following figure,

```
// in src/main.rs

#[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 {}
}
```

Figure 2.6.1: Using the “unsafe” keyword[7]

A Double Fault Handler[7]

A double fault is a common exception to the error code so that the user may assign a handler feature similar to the breakpoint handler.

```
// 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: &mut InterruptStackFrame, _error_code: u64) -> !
{
    panic!("EXCEPTION: DOUBLE FAULT\n{:#?}", stack_frame);
}
```

Figure 2.6.2: Double Fault Handler[7]

A short error message is printed in this handler, and the exception stack frame is discarded. The Double Fault Handlers’ error code is always zero, so there is no need to print it. After starting the kernel, it shows double fault handler is invoked.

## 2.7 Hardware Interrupts

In here using the programmable interrupt controller to forward hardware interrupts to the CPU correctly. Add new entries to the Interrupt Descriptor Table to handle these interrupts, much as for exception handlers.

Interrupts provide a way for connected hardware devices to alert the CPU. So, the keyboard should inform the kernel of each keystroke instead of asking the kernel to periodically search the keyboard for new characters (a process called polling). This is much more powerful because when anything happens, the kernel needs to function. It also allows for quicker response times, as the kernel is able to respond instantly and not only at the next poll.[8]

It is not possible to connect all hardware devices direct to the CPU. A separate interrupt controller instead aggregates all system interrupts and then alerts the CPU as following,

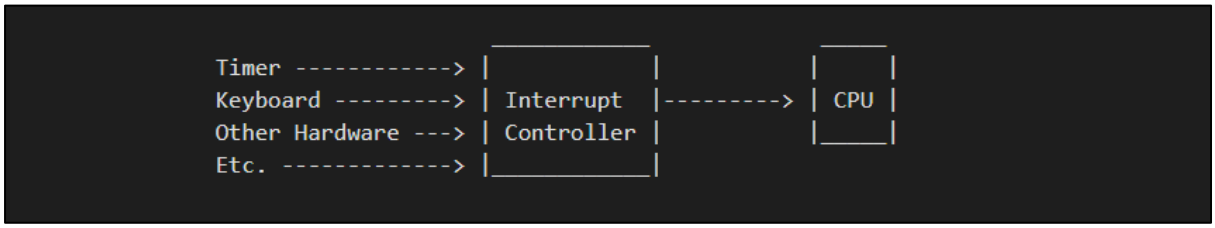


Figure 2.7.1: Separate Interrupt Controller[8]

Hardware interrupts happen asynchronously, unlike exceptions. The strict ownership model of Rust assists here because it prevents a mutable global state.

## 2.8 Introduction to Paging

The operating system uses a straightforward memory management system. This explains why memory insulation is necessary, how segmentation works, and which virtual memory is, and how memory fragmentation problems are solved by paging. The development of multilevel page tables on the x86 64 architecture is also explored.

Memory Protection[9]

One of the operating system's key tasks is to separate programs from each other. For example, a web browser should not be able to mess with a text editor. Operating systems use hardware features to accomplish this purpose, to ensure that memory areas of one process are not accessible by other processes.

Segmentation[9]

Segmentation increases the amount of memory that is addressable. The CPU automatically added this offset on each memory access, so that up to 1MiB of memory were accessible.

Virtual Memory[9]

Abstracting the memory addresses from the underlying physical storage system is the concept behind virtual memory. Addresses before the translation are called virtual and address after the translation are called physical. One significant distinction between these two types of addresses is that physical addresses are identical and often refer to the same, different location of memory. On the other hand, virtual addresses depend on the translation function.

It shows in the following figure,

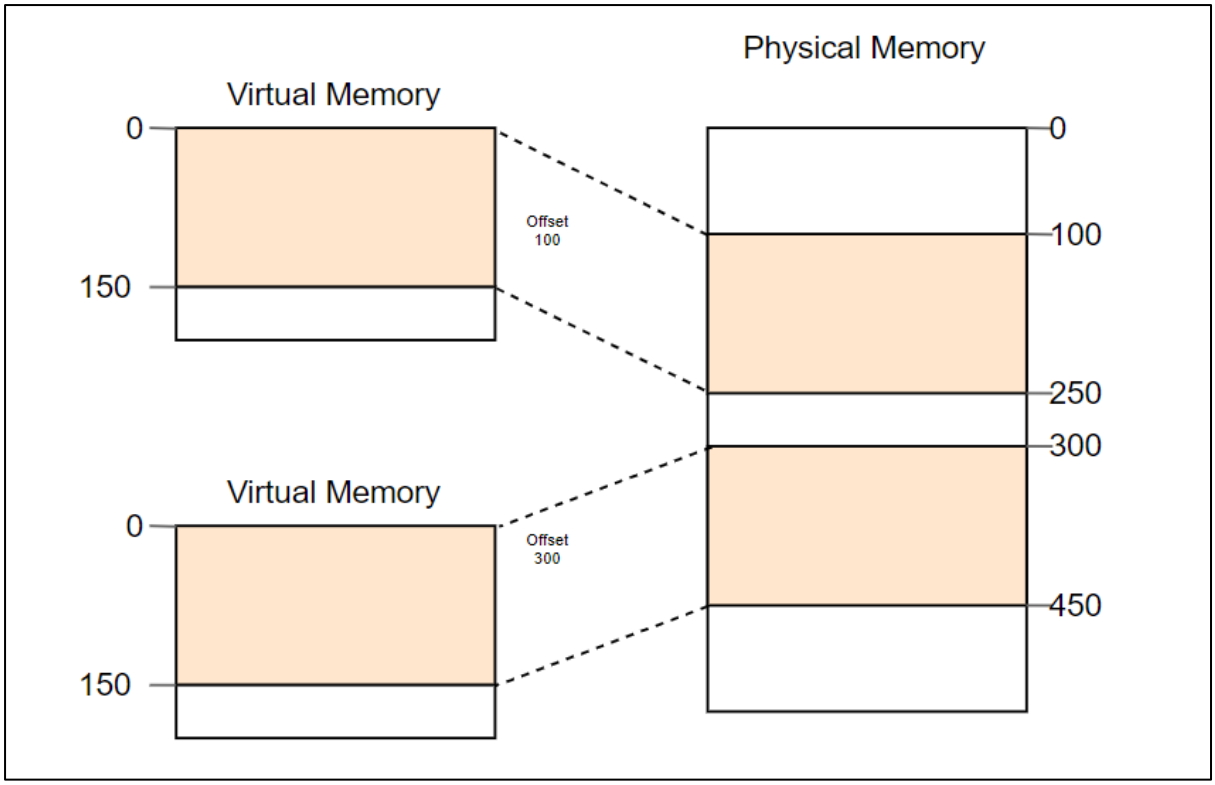


Figure 2.8.1: Physical and Virtual Memory[9]

Fragmentation[9]

Segmentation is very powerful because of the difference between virtual and physical addresses. It does have the issue of fragmentation, however. One way to combat this fragmentation is to interrupt execution, pull together the used parts of the memory, update the translation, and then restart execution.

Paging[9]

The principle is to break the space of virtual and physical memory into small, fixed-size blocks. The virtual memory space blocks are called pages, and the actual address space blocks are called frames. It is possible to map each page individually to a frame, allowing wider memory regions to be separated into non-continuous physical frames. This allows the user to start the program's third instance without previously performing any defragmentation.

### 2.9 Paging Implementation

It first examines various strategies to make the kernel available to the physical page table frames and addresses their respective benefits and disadvantages. In order to construct a new mapping, it then implements an address translation function and functionality.[10]

The bootloader requires help to execute the solution, so configure it first. Accessing Page Tables look like the following figure,

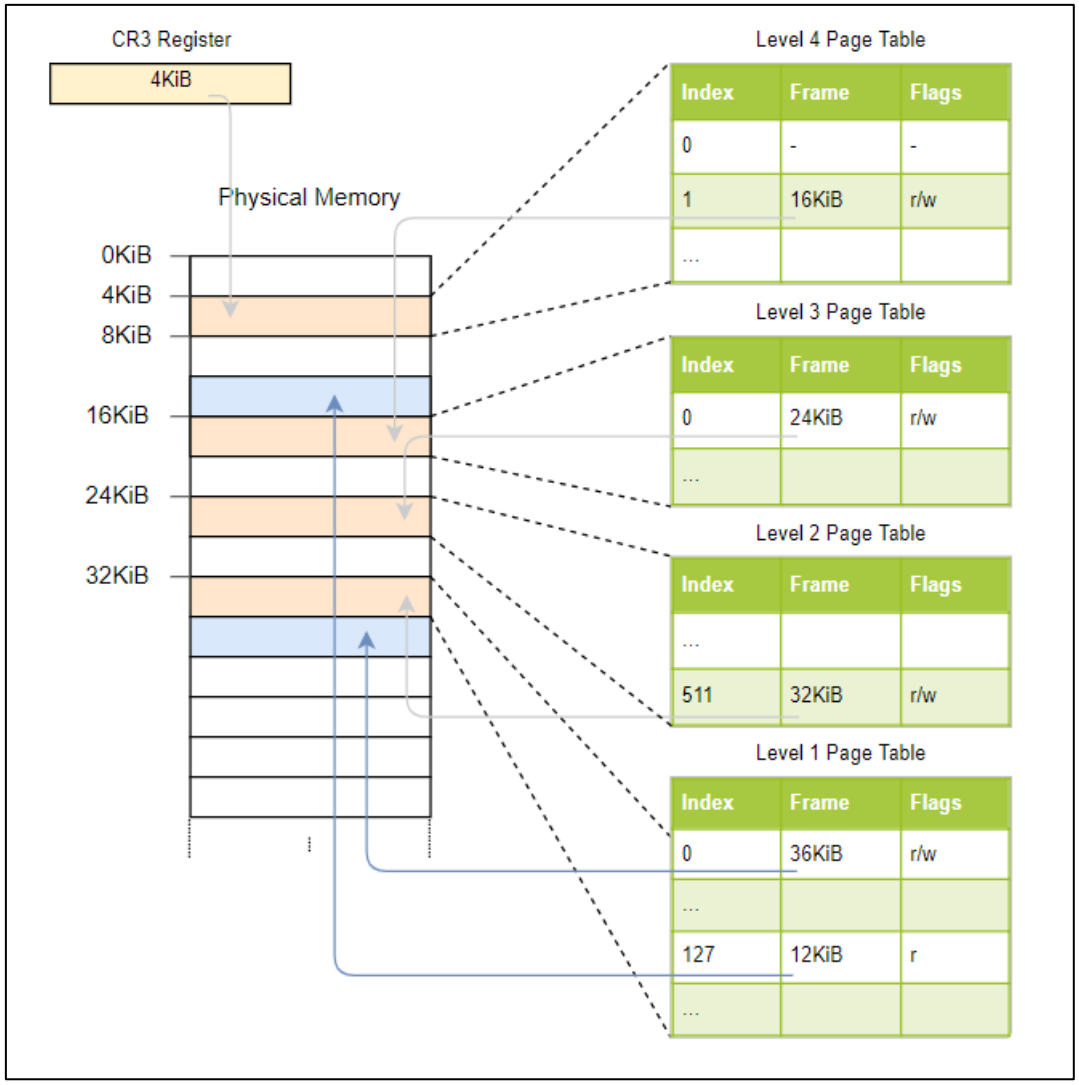


Figure 2.9.1: Accessing Page Table[10]

Finally, after getting access to physical memory user can start to implement the page table code.

### 2.10 Heap Allocation

First, it introduces dynamic memory and demonstrates how borrowing checker avoids common allocation errors. It then implements Rust's virtual allocation interface, generates a region of heap memory, and sets up an allocator crate.

The Rust standard library provides abstraction types which implicitly call these functions instead of allowing the programmer to allocate and deallocate manually. The most significant type is the "box", which is a heap-allocated value abstraction. It provides a "box::new" builder functionality that takes a value, assigns calls to the value size, and then transfers the value to the newly assigned slot on the heap.[11]

The code will look like following,

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

This is where ownership comes in for Rust. It assigns each reference an abstract lifetime, which is the range in which the reference is right.

The ownership scheme of Rust goes even further and not only prevents use-after-free bugs but offers full memory protection like languages collected from garbage such as Java or Python do.

Besides, in multi-threaded code, it ensures thread protection and is therefore even safer than those languages. Furthermore, most importantly, at compile-time, all these checks occur, so compared to handwritten memory management in C, there is no runtime overhead.[11]

### 2.11 Allocator Designs

An allocator must control the available heap memory. On “alloc” calls, it needs to return unused memory and retain track of memory released by “dealloc” so that it can be reused again. Most importantly, since this will trigger undefined actions, a memory that is already in use somewhere else must never be handed out.[12]

There are also secondary design objectives, apart from correctness. The allocator, for example, can use the available memory efficiently and keep fragmentation minimal.

There are three possible kernel allocator designs; they have their advantages and drawbacks. They are[12],

- Bump Allocator - Which distributes memory linearly by increasing the next single pointer.
- Linked List Allocator - To build a linked list, the so-called free list, that uses the released memory blocks themselves.
- Fixed-Size Block Allocator - To satisfy allocation demands, it uses fixed-size memory blocks.

Using any of the above methods user can create Allocators. It is also important to note that there is a specific workload for each kernel implementation, so there is no “best” allocator design that suits all cases.

### 2.12 Async/Await

Async/Await is a feature of Rust. In the context of async / await, the Rust language offers first-class support for cooperative multitasking.

The principle behind async / await is to allow the programmer to write code that looks like regular synchronous code but is translated by the compiler into asynchronous code. It centred on the two keywords, “async” and “await.”[13]

In a function signature, the “async” keyword can be used to transform a synchronous function into an asynchronous function which returns a future function as following.

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

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

Figure 2.12.1: Using the “async” keyword[13]

However, the wait keyword can be used within asynchronous functions to retrieve the asynchronous meaning of a future feature as following.

```
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
    }
}
```

Figure 2.12.2: Using the “await” keyword[13]

## 3.References


- [1] Rust [ online ] Available https://www.rust-lang.org/ [accessed date 07/10/2020]
- [2] Writing an OS in Rust. A Freestanding Rust Binary [ online ] Available https://os.phil-opp.com/freestanding-rust-binary/ [accessed date 07/10/2020]
- [3] Writing an OS in Rust. A Minimal Rust Kernel [ online ] Available https://os.phil-opp.com/minimal-rust-kernel/ [accessed date 08/10/2020]
- [4] Writing an OS in Rust. VGA Text Mode [ online ] Available https://os.phil-opp.com/vga-text-mode/ [accessed date 08/10/2020]
- [5] Writing an OS in Rust. Testing [ online ] Available https://os.phil-opp.com/testing/ [accessed date 08/10/2020]
- [6] Writing an OS in Rust. CPU Exceptions [ online ] Available https://os.phil-opp.com/cpu-exceptions/ [accessed date 08/10/2020]
- [7] Writing an OS in Rust. Double Faults [ online ] Available https://os.phil-opp.com/double-fault-exceptions/ [accessed date 09/10/2020]
- [8] Writing an OS in Rust. Hardware Interrupts [ online ] Available https://os.phil-opp.com/hardware-interrupts/ [accessed date 09/10/2020]
- [9] Writing an OS in Rust. Introduction to Paging [ online ] Available https://os.phil-opp.com/paging-introduction/ [accessed date 10/10/2020]
- [10] Writing an OS in Rust. Paging Implementation [ online ] Available https://os.phil-opp.com/paging-implementation/ [accessed date 10/10/2020]
- [11] Writing an OS in Rust. Heap Allocation [ online ] Available https://os.phil-opp.com/heap-allocation/ [accessed date 10/10/2020]
- [12] Writing an OS in Rust. Allocator Designs [ online ] Available https://os.phil-opp.com/allocator-designs/ [accessed date 10/10/2020]
- [13] Writing an OS in Rust. Async/Await [ online ] Available https://os.phil-opp.com/async-await/ [accessed date 10/10/2020]