

Sri Lanka Institute of Information Technology

**How Modern Language “Rust” Can Be Used to**

**Implement More Secure, Reliable, High Performance Operating System**

**Individual Assignment**

IE2032 – Secure Operating System

Submitted by:

**Contents**

[Introduction 3](#_Toc118584742)

[What is an Operating System? 4](#_Toc118584743)

[About Rust 4](#_Toc118584744)

[Powerful Type System 5](#_Toc118584745)

[Mutex 5](#_Toc118584746)

[Page Table Methods 6](#_Toc118584747)

[Easy Dependency Management 6](#_Toc118584748)

[Using an Allocator Crate 6](#_Toc118584749)

[Exciting New Features 7](#_Toc118584750)

[Futures 7](#_Toc118584751)

[Future Combinators 7](#_Toc118584752)

[The Async/Await 7](#_Toc118584753)

[References 8](#_Toc118584754)

# Introduction

Rust is an open-source systems programming language that empowering to implement memory safety, speed, more reliable and efficient operating systems. These runs fast because Rust has no runtime requirement and control over memory allocation.

This assignment is about “How the modern language “Rust” can be used to implement more secure, reliable, high performance operating system”.

This includes about “What is the operating system” and a brief introduction about the programming language Rust. And describes about the performance, reliability, productivity of Rust. Next this includes the information about the different aspects of the language Rust, such as memory safety, encapsulating unsafety, the power of the type system of Rust, dependency management, contribution as a tool to implement a operating system. At last this describes about the exciting new features of Rust, such as futures, async/await.

# What is an Operating System?

An operating system is the set of software that allow users to run various applications on a device, and after being initially loaded into the computer system by a boot program, manage entire application programs in a computer system. Operating System manages input and output devices in its systems. It utilizes device drivers written by hardware creators to communicate and interact with their devices. OS locate in between the applications and the hardware, as the interface between the two. Users can directly interact with its OS through user interfaces such as a Graphical User Interface (GUI) or a Command Line Interface (CLI). A lot of software things like libraries, common system services and APIs (Application Programming Interfaces) contained in Operating systems.

OS moreover handles multi-tasking, designating hardware among different running programs. The Operating System controls which processes run, and it distributes them between different CPUs if you have a device with multiple CPUs or core, allowing multiple processes run in parallel. It moreover oversees the internal memory of the system, apportioning memory between running applications. OS controls and manipulate all computer functions and allows the best features such as “Plug and Play”, it means no need any drivers for make use of devices like as mouse, keyboard and more. Operating System employments various strategies such as memory segmentation, swapping and paging and it can oversee own memory with utilizing those methods. OS can also handle all types of interrupts and it performs all types of scheduling methods such as Priority Scheduling, First Come- First Served, Round Robin and Shortest Job First Scheduling for scheduling all the processes in CPU for execution.

Some Operating Systems furnishes the protection from dangerous virus and files such as Windows defender in Windows Operating Systems. Various OS are accessible in open source such as Unix/Linux and OS can be run effortlessly on Computer System without any cost. In spite of the fundamental roles of an operating system are ubiquitous, incalculable operating systems that serve a wide extend of hardware and client needs.

* General-purpose Operating System
* Mobile Operating System (MOS)
* Embedded Operating System
* Network Operating System (NOS)
* Real-time Operating System (RTOS)

The contrast between operating system types are not supreme, and few operating systems can share features of others. For example, General-purpose OSs routinely incorporate the networking capabilities found in conventional NOS. Similarly, an Embedded Operating System commonly incorporates properties of an RTOS, while a MOS can still ordinarily run various app at the same time like other General-purpose operating systems.

# About Rust

Rust is a 5 years old programming language that mainly focused on safety, performance, concurrency. This is a multi-paradigm programming language and syntactically similar to C++. Rust does not have a garbage collection when comparing to other safe languages, Rust is a relatively young programming language and already used by Mozilla, Dropbox, Cloudflare. The language Rust was designed by Graydon Hoare, Dave Herman, Brendan Eich, and others at Mozilla research.

Rust architecture allows to build programs that have low-level language output and control, but with high-level language abstractions. These properties make Rust ideal for programmers who have experience in languages such as C and are looking for a safer alternative, as well s those from languages such as Python who are looking for ways to write code that performs better without compromising expressiveness.

### Performance

Rust is fast and memory-efficient without a runtime and garbage collector. It can control critical services, and easily integrate with other languages.

### Reliability

Rust has a powerful type system and ownership model guarantee, memory-safety and threadsafety. This allows to eliminate classes of bugs during the compile time.

### Productivity

Rust has excellent documentation, a friendly compiler with helpful error messages, and top-notch tools such as integrated package manager and build tool, smart multi-editor that support for completion and type checks, auto-formatter.

# Powerful Type System

## Mutex

Mutex is an abbreviation for mutual exclusion, in case of mutex, only one thread can access data at a given time. Rust ensures that the mutex is locked before accessing the data. The thread must first signal that it wants access by asking for the mutex lock. The lock is a data structure that forms part of a mutex that keeps track of who currently has exclusive access to the data. The mutex is defined therefor as protecting the data it keeps through the locking mechanism.

There are two rules to remember in mutex. First one is before using the data there should be an attempt to acquire the lock. Second one is when it is done with the data that the mutex holds, it is need to unlock the data. So that other threads can get the lock.

Mutex management can be extremely tricky to get right. In Rust systems an ownership rules, it is impossible to have incorrect locking and unlocking.

## Page Table Methods

This represents contracts in code instead of documentation. For each page there should be an allocated physical frame. In this, page size of the page and the frame parameters should match. All these things happen at compile time not at run time. In Rust, the method *Mapper::map\_to* is used to create the mapping in the active page table.

# Easy Dependency Management

## Using an Allocator Crate

In Rust crate is a compilation unit. Whenever *rustc some\_file.rs* is called*, some\_file.rs* is viewed as a crate file. If some *file.rs* has mod declaration in it, the contents of the module files will be placed in positions where mod declarations are contained in the crate file before running the compiler over it. In other words, the modules are not compiled individually, only the crates are complied.

A crate can be compiled into a binary or a library. *rustc* can generate a binary from a crate. This action can be overridden by transferring the *–crate-type* flag to *lib*.

The *linked\_list\_allocator* crate is a simple allocator crate for *no\_std* applications. Its name comes from the fact that it uses a linked list data structure to keep track of the memory regions.

To use the crate, initially it needs to be added a dependency on it in *Cargo.toml.* cargo is the official Rust package management tool. It has a lot of useful features to boost code quality and developer speed.

Then should replace the dummy allocator with the allocator provided by the crate.

The struct is called *LockedHeap* because it uses the form *spinning\_top::Spinlock* for synchronization. This is necessary since multiple threads can access the static *ALLOCATOR* at the same time. When using a spinlock or a mutex, it is needed to be careful not to inadvertently trigger a deadlock. This ensures that the people are not permitted to perform any allocations in interrupt handlers, since run at a random time and might interrupt the in-progress allocation.

Setting the *LockedHeap* to be a global allocator is not enough. The explanation is that use an *empty* constructor function, which generates an allocator without any backup memory. As the dummy allocator it returns an *alloc* error. To address this, after constructing a heap, it is needed to initialize the allocator.

# Exciting New Features

## Futures

A future is a value that may not yet be available. This may be, for example, an integer that is being calculated by another task or a file that is being downloaded from the network. Instead of waiting until the value is available, futures would make it possible to continue the execution until the value is required.

Futures are represented by the *Future* trait in Rust. The *poll* method is used to check whether the value is already available or not and returns a *poll enum*. When the value is available, that means if the file was completely read from the disk, it is returned wrapped in the *ready* variant. Otherwise, another variant will be returned, which indicates to the caller that the value is not yet available.

It needs actively wait for the future by calling the poll in a loop. The arguments are omitted because *poll* do not matter in this situation. Although this approach works, it is inefficient since hold the processor busy before the values available.

Block the current thread before future becomes available, might be the more efficient way. This is possible when there is a thread, because of that this is not support for the kernel. The systems where blocking is possible, it is not needed because it transforms asynchronous tasks into synchronous task again.

## Future Combinators

The alternative to waiting is to use future combinations. These are the methods such as *map* that allow for chaining and combining future together. Without waiting for the future, these combinators return the future themselves, that applies the mapping operation.

Keep the operations asynchronous is the benefit of future combinators. This approach can lead to very high performance in combination with asynchronous I/O interfaces. These combinators are implemented as normal struct with trait implementation.

## The Async/Await

Async/await are special pieces of Rust syntax that make it possible to monitor the current thread rather than interrupt it, allowing another code to make progress while waiting for the process to be completed.

The concept behind async/await is to let the programmer write the code that looks like a regular synchronous code, but the compiler turns it into an asynchronous code. It functions the basis of *async* and *await* the two keywords. The asynchronous keyword can be used in signature function to transform the synchronous function into an asynchronous function that returns the future. Async/await keywords alone would not be useful and in synchronous functions, the *await* keyword might be used to recover the asynchronous value of the future.

# References

* Rust Programming Language, available at : <https://www.rust-lang.org/>

* Why You Should Use Rust in 2020, available at : <https://serokell.io/blog/rust-guide>

* Traits: Defining Shared Behavior - The Rust Programming Language, available at :

<https://doc.rust-lang.org/book/ch10-02-traits.html?search=map_>

* What is Ownership? - The Rust Programming Language, available at : [https://doc.rustlang.org/book/ch04-01-what-is-ownership.html?search=encapsulating%20unsa](https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html?search=encapsulating%20unsa)

* A Freestanding Rust Binary | Writing an OS in Rust, available at : [https://os.philopp.com/freestanding-rust-binary/#introduction](https://os.phil-opp.com/freestanding-rust-binary/#introduction)

* VGA Text Mode | Writing an OS in Rust, available at : [https://os.phil-opp.com/vga-textmode/#summary](https://os.phil-opp.com/vga-text-mode/#summary)

* Testing | Writing an OS in Rust, available at : <https://os.phil-opp.com/testing/>

* A Minimal Rust Kernel | Writing an OS in Rust, available at : [https://os.philopp.com/minimal-rust-kernel/](https://os.phil-opp.com/minimal-rust-kernel/)

* CPU Exceptions | Writing an OS in Rust, available at : [https://os.phil-opp.com/cpuexceptions/](https://os.phil-opp.com/cpu-exceptions/)

* Introduction to Paging | Writing an OS in Rust, available at : [https://os.philopp.com/paging-introduction/](https://os.phil-opp.com/paging-introduction/)

* Async/Await | Writing an OS in Rust, available at : <https://os.phil-opp.com/async-await/>

* Hardware Interrupts | Writing an OS in Rust, available at : [https://os.philopp.com/hardware-interrupts/](https://os.phil-opp.com/hardware-interrupts/)

* Allocator Designs | Writing an OS in Rust, available at : [https://os.philopp.com/allocator-designs/](https://os.phil-opp.com/allocator-designs/)