**Sri Lanka Institute of Information Technology**

**(SLIIT)**

A picture containing text, clipart, vector graphics

Description automatically generated

How Modern Language “Rust” Can Be Used to Implement

More Secure, Reliable, High-Performance Operating System

**IE2032-Secure Operating System**

**2nd Year 1st semester**

|  |  |  |
| --- | --- | --- |
|  | Student Registration Number | Student Name |
| 01 | IT21169076 | D.A.S.M ATHUKORALA |
| 02 | IT21184994 | R.D.D.L.K JAYASINGHE |
| 03 | IT21164576 | M.A.D SANDEEPANI |
| 04 | IT21177750 | P.A.G.N PALIHAWADANA |

**Group details\_\_**

**Terms of Reference**

A report submitted in fulfilment of the requirement for the module secure operating system with module code IE2032, Sri Lanka Institute of Information Technology.

**Acknowledgement**

We would like to express our gratitude to the module lecturer and the lecture panel for guiding us through this semester and for helping us by giving examples, guidelines, reminders, and advice about the project.

Furthermore, we would like to extend our gratitude to every author, contributor and publishers of the websites, books, and journal we referred for this report.

**Introduction**

Rust is an open-source systems programming language that enables the development of operating systems that really are faster, more dependable, or more impactful while still verifying that memory is protected. These execute quickly because Rust has control over memory allocation and no runtime requirements.

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

This section covers information on "What is the operating system" and a synopsis of the programming language Rust. And talks on Rust's efficiency, dependability, and productivity. The information regarding the various features of the Rust programming language, such as memory safety, encapsulating unsafety, the strength of the Rust type system, dependency management, and contribution as a tool to implement an operating system, is then included. This concludes the discussion of Rust's intriguing new features, such as futures and async/await.

**What is operating system**

An operating system is a collection of software that enables users to execute different applications on a device and, following initial loading by a boot program, manages the entirety of application programs in a computer system. In its systems, the operating system controls the input and output devices. To interact and communicate with the devices, it uses device drivers developed by the makers of the underlying hardware. OS serves as the interface between hardware and software, sitting between the two. Through user interfaces like a Command Line Interface (CLI) or a Graphical User Interface (GUI), users can interact with its OS directly (CLI). Operating systems include a variety of software components, such as libraries, standard system functions, and APIs (Application Programming Interfaces).

The operating system (OS) likewise manages multi-tasking by allocating hardware among many executing apps. If any device has many CPUs or cores, the operating system decides which processes run and distributes them among them effectively allow that it can operate concurrently. Additionally, it is in charge of allocating memory among active apps within the system's internal memory. All computer operations are managed and controlled by the operating system, which enables the best features like "Plug and Play," which eliminates the need for drivers to use peripherals like the mouse, keyboard, and other items. Using several techniques including memory segmentation, swapping, and paging, the operating system can manage its own memory.

Some operating system provides protection from harmful viruses and files, such Windows Defender in Windows operating systems. Numerous operating systems (OS) are available in open source, including Unix/Linux, and can be easily run on a computer system without any fees. Operating systems encompass a wide range of hardware and client needs despite the fact that its basic functions are limited. Systems are ubiquitous and uncountable.

• General-purpose Operating System

• Mobile Operating System (MOS)

• Embedded Operating System

• Network Operating System (NOS)

• Real-time Operating System (RTOS)

Several operating systems can share capabilities with other operating systems, and there is not a stark difference between the many operating system types. For examples, networking features available in traditional NOS are frequently included in general-purpose OSs. A MOS may typically run multiple apps at once like other General-purpose operating systems, although an RTOS frequently integrates capabilities of an RTOS.

**About RUST**

The five-year-old programming language called Rust primarily focuses on concurrency, performance, and safety. This is a multi-paradigm programming language that is comparable to C++ in terms of syntax. Rust, a relatively new programming language that is already utilized by Mozilla, Dropbox, and Cloudflare, does not have a garbage collection when compared to other safe languages. Graydon Hoare, Dave Herman, Brendan Eich, and others from Mozilla Research created the language Rust.

An alternative to C++ is a systems programming language called Rust. Rust's primary goal is (memory) safety, although it has now started to concentrate on performance as well, embracing the C++ strategy of zero cost abstraction. Rust excels in performance and safety, which is fantastic, but it also has other characteristics that make it far more adaptable than other systems languages.

A variety of crates (Rust libraries) are available for every need you may have, and all can be installed using the official Cargo program, making any software project much simpler thanks to the ecosystem that exists for Rust.

Programs with low-level language output and control but high-level language abstractions can be created using the Rust architecture. These characteristics make Rust the perfect programming language for both experienced C and Python programmers looking for a safer option, as well as Python programmers looking for ways to develop code that performs better without sacrificing expressiveness.

**Performance**

Rust uses the idea of zero cost abstraction to simplify the language without sacrificing efficiency, as was noted in the introduction. Monomorphization, which enables the creation of generic functions that are transformed into the required concrete type functions at compile time, is a straightforward illustration of this concept. As a result, no runtime costs are incurred. Additionally, the standard library uses this idea so that there is no need to reinvent common types like collections, which enhances performance and library interoperability. The absence of a trash collector, which provides various performance advantages, is the second noteworthy advantage of Rust.

**RUST Security**

The security of software produced in a language and the safety of that language have a lot in common. Apparently, this means that ensuring memory safety can eliminate up to about 70% of security bugs. The Microsoft Security Response Center estimates that memory exploits (i.e., buffer overflows or other memory misuses) are to blame for 70% of the bugs that they assign a common vulnerabilities and exposure (CVE) tag to. Memory safety flaws encompass a wide range of various problems. In prior work, we have discussed some of the more prevalent memory-related problems, such as data races, use after free, dereferencing a null pointer, and double free (not all consider data races to be a memory safety issue).

Any safety-related bug's primary flaw is that they all produce undefined behavior (UB), which leaves anything possible, including security flaws, open to possibility. Languages with manual memory management, like C, are where use after free problems are most prevalent. Simply freeing a memory location while it is still needed is all that is required to generate one. Accessing memory after it has been released can result in a wide range of effects, each of which will vary depending on the operating system and hardware. Since the memory could be utilized for other variables after it is freed, this could result in a security vulnerability where information could be leaked by sending the data at the freed memory location.

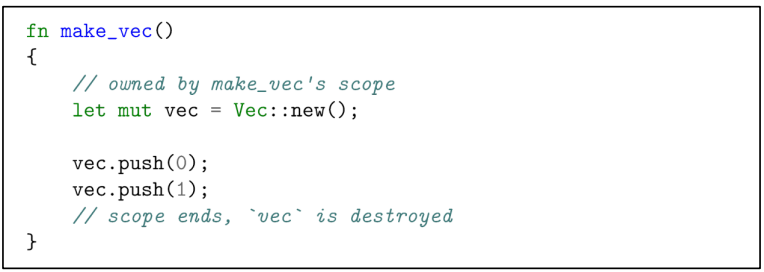
Indeed, if the operating system determines that the address is invalid after it has been freed, the application will encounter a segmentation fault, and execution will be suspended. As a result, Rust does not generally have this issue and it is far simpler to guarantee the validity of a tiny portion of code than for the full codebase. Rust does not provide manual memory management outside of the infrequently used unsafe block.

**RUST Safety**

A programming language's safety depends on its capacity to stop or catch errors like buffer overreads. Since memory management is the root of the most of these problems, modern languages have developed memory management systems, the most well-known of which being garbage collection.

Rust optimizes run time performance by using an ownership based approach to choose memory allocations and deallocations at compile time. Simply explained, memory is allocated when a variable is declared and deallocated when it is no longer in scope because the variable's scope is the memory's owner. The use after free and double free vulnerabilities are avoided by this component of the ownership system because the programmer is not required to manually manage the memory. Although the complete complexity of the ownership system is outside the purview of this study, it should be noted that memory can be given a lifetime that enables it to outlive the range of its variable.

The ownership system's requirement that there only ever be one owner of memory at any given time is another truly remarkable feature. Value sharing between functions, structs, and threads becomes problematic as a result of this. In order to resolve this, a borrowing system that has distinct guidelines for shared references is used. The borrowing system's rules are quite straightforward: either one changeable reference to memory or several immutable references may exist, which means that no more than one thread may modify the memory while no other threads may read it. Rust is a fantastic option for highly concurrent software systems because the combination of these systems and their rules entirely prevents data races from happening.



Simple example of Rust’s ownership system managing memory

**Reliability**

A powerful type system, ownership model guarantee, memory safety, and thread safety are all features of Rust. This enables the removal of entire classes of defects at compilation time.

**Productivity**

Rust includes top-notch tools including an integrated package manager and build tool, a clever multi-editor that supports completion and type checks, and an auto-formatter. It also has outstanding documentation and a pleasant compiler with useful error messages.

**Memory safety of Rust**

No Invalid Memory Accesses

No buffer overflows #

Buffer overflows don't exist in the Rust programming language. In the C language, array capacity is not verified on access when comparing. So, buffer overflows are common.

No dangling pointers

In pointer languages, it is possible to create a dangling pointer—one that points to a location in memory that may have been given to someone else—by releasing some memory while still holding a pointer to that region. References will never dangle in Rust, according to the compiler's promise. The compiler will make sure that any data that is referenced will remain inside its scope up until the reference is made.

No data races

A secure Rust guarantee prevents data races. These are described as, at least two threads accessing the same memory region simultaneously, at least one of which is a write, and at least one of which is unsynchronized.

The data race has ill-defined behavior, making it challenging to execute in safe Rust. Rust's ownership mechanism frequently prevents data races. A data race cannot be carried out since it is impossible to alias a mutable reference. Internal mutability adds to the complexity of things. Fundamentally speaking, it is challenging and undesired. If the operating system, other software, and the hardware are all racy, so is the environment in which everything is running.

In so doing, it is entirely "OK" for safe Rust programs to urge deadlocking or to perform absurd tasks with poor synchronization. This program is not very nice, but Rust can only go so far. A race condition cannot break memory safety on its own in a Rust program. Race conditions can only compromise the memory's security when used in conjunction with other hazardous code.

**The Async/Await**

The Rust syntax features async/await, which allow you to watch the running thread rather than interrupt it and make progress in another program while waiting for the process to finish. Async/await works on the principle of letting programmers write what appears to be standard synchronous code, but the compiler converts it to asynchronous code. The two keywords are awaited, and the async foundation is used. The synchronous function in the signature function can be changed into an asynchronous function that returns the future by using the asynchronous keyword. In synchronous functions, the await keyword might be used to retrieve the asynchronous value of the future because the async/await keywords alone would not be helpful.

Async/ await are special pieces of rust syntax that make it possible to yield control of the current thread rather than blocking , allowing other code to make progress while waiting on an operation to complete.

There are two main ways to use async : async fn and async : async fn and async blocks. Each return a value that implements the future trait :

// `foo()` returns a type that implements `Future<Output = u8>`.

// `foo().await` will result in a value of type `u8`.

async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {

// This `async` block results in a type that implements

// `Future<Output = u8>`.

async {

let x: u8 = foo().await;

x + 5

}

}

**Double fault**

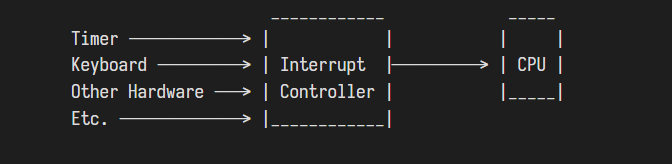
A double fault is a specific exception that happens when the CPU neglects to call an exception handler, to put it simply. It happens, for instance, when a page fault is raised but no page fault handler is registered in the interrupt descriptor table (IDT). The catch(...) or catch(Exception e) blocks in C++ or Java or C# are examples of catch-all blocks in programming languages with exceptions. A double fault functions just like any other exception. So can define a typical handler function for it in the IDT because it has the vector number. The provision of a double fault handler is crucial because, if left unattended, a double fault can result in a lethal triple fault.

Graphical user interface, text, application, chat or text message

Description automatically generated

**Hardware interrupts**

The CPU can be informed by associated hardware devices using interrupts. So the keyboard may inform the kernel of each keypress rather than allowing the kernel periodically inspect the keyboard for new characters (a process known as polling). Because the kernel only needs to react when something occurs, this is far more efficient. Additionally, because the kernel can respond right now rather than just at the subsequent poll, reaction times are sped up. It is impossible to connect all gear directly to the CPU. As an alternative, a different interrupt controller gathers all interrupts from all devices and then alerts the CPU:



The majority of interrupt controllers may be programmed, thus they support a range of interrupt priority levels. To ensure precise timekeeping, for instance, this enables giving timer interrupts a greater priority than keyboard interrupts. Hardware interrupts take place asynchronously, unlike exceptions. They can therefore happen at any time and are totally independent of the code that was actually executed. Consequently, all of the potential concurrency-related issues suddenly appear in our kernel in the shape of a type of concurrency. Because it bans changeable global state, Rust's rigid ownership model is helpful in this situation. However, as we will see later in this piece, deadlocks are still possible.

**introduction to Paging**

introduces paging, a widely used method of managing memory that we will also implement in our operating system. It defines virtual memory, shows how segmentation works, explains why memory isolation is necessary, and describes how paging addresses memory fragmentation issues. Additionally, it investigates the x86 64 architecture's multilevel page table arrangement.

Program isolation is one of the major functions of an operating system. Segmentation and paging are the two methods of memory protection that the x86 hardware enables.

*Paging*

virtual and physical memory space divide into small ,fixed size blocks and the blocks of the virtual memory space are called by pages. the blocks of the physical address space are called as frames. It is possible to divide bigger memory regions across non-continuous physical frames since each page can be individually mapped to a frame.

*page tables*

For each active memory sector, segmentation utilizes a separate segment selection register; paging is not feasible due to the vastly greater number of pages than registers. The mapping information

instead kept in a table structure called a page table by paging. We can utilize a two-level page table to reduce the memory that is wasted .The concept is to use different page tables for different address zones .The two-level page table concept can be expanded to three, four, or even more levels .The page table registration then points to the top-level table, which points to the table below it, which points to the table below that, and so forth. Next, the mapped frame is referenced by the level 1 page table. A multilevel or hierarchical page table is the term used to describe the concept in general.

This substitutes code for documentation to represent contracts. There should be a physical frame assigned to each page. Here, the frame settings and the page size should line up.These events all take place during compilation, not during execution. The mapping in the active page table is created in Rust using the method Mapper::map to.

**paging implementation**

page implementation is how to make our kernel support paging. It initially examines various methods for making the physical page table frames available to the kernel and considers each one's benefits and shortcomings .After that, a function for translating addresses and one for creating new mappings are implemented.

We will configure the bootloader first because we need its support in order to put the strategy to use. Then, in order to convert virtual to physical addresses, we will put into practice a function that iterates through the page table hierarchy. Finally, we discover how to find unused memory frames for new page tables and how to add new mappings to the page tables.

First, let's look at the currently running page tables that our kernel runs on. In the second step, we create a translation function that returns the physical address to which a given virtual address is mapped. As a final step, we try to modify the page tables to create a new mapping.

We used the Mapped Page Table type of the x86 64 crate after manually traversing the page tables to construct a translation function. On top of the memory map that the bootloader gave, we learnt how to build the appropriate Frame Allocator and how to add additional mappings to the page table.

**Heap Allocation**

This post adds support for heap allocation to our kernel. First, it gives an introduction to dynamic memory and shows how the borrow checker prevents common allocation errors.

It then implements the basic allocation interface of Rust, creates a heap memory region, and sets up an allocator crate. At the end of this post, all the allocation and collection types

of the built-in allocation crate will be available to our kernel

We currently use two types of variables in our kernel: local variables and static variables. Local variables are stored on the call stack and are only valid until the surrounding function returns. Static variables are stored at a fixed memory location and always live for the complete lifetime of the program.

*local variable*

Local variables are stored on the call stack ,which is a stack data structure that supports push and pop operations. On each function entry, the parameters, the return address, and the

local variables of the called function are pushed by the compiler.

*Static Variables*

Static variables are stored at a fixed memory location separate from the stack. This memory location is assigned at com pile time by the linker and encoded in the executable. Statics live for the complete runtime of the program, so they have the 'static lifetime and can always be referenced from local variables

Rust enforces this because a data race would occur if, e.g.,

two threads modified a static variable at the same time. The only way to modify a static variable is to encapsulate it in a Mutex type, which ensures that only a single &mut reference exists at any point in time. We already used a Mutex for our static VGA buffer Writer

*Dynamic Memory*

programming languages often support a third memory region for storing variables called the heap. The heap supports dynamic memory allocation at runtime through two functions called allocate and deallocate. It works in the following way The allocate function returns a free chunk of memory of the specified size that can be used to store a variable. This variable then lives until it is freed by calling the deallocate function with a reference to the variable.

Rust takes a different approach to the problem:

It uses a concept called ownership that is able to check the correctness of dynamic memory operations at compile time. Thus, no garbage collection is needed to avoid the mentioned vulnerabilities ,which means that there is no performance overhead. Another advantage of this approach is that the programmer still has fine-grained control over the use of dynamic memory, just like with C or C++.

*Allocations in Rust*

manually call allocate and deallocate, the Rust standard library provides abstraction types that call these functions implicitly. The most important type is Box, which is an abstraction for a heap-allocated value. It provides a Box::new constructor function that takes a value, calls allocate with the size of the value, and then moves the value to the newly allocated slot on the heap. To free the heap memory again, the Box type implements the Drop trait to call deallocate when it goes out of scope:

Rust’s ownership system goes even further, preventing not only use-after-free bugs but

also providing complete memory safety, as garbage collected languages like Java or Python do. Additionally, it guarantees thread safety and is thus even safer than those languages in multi-threaded code. And most importantly, all these checks happen at compile time, so there is no runtime overhead compared to hand-written memory management in C.

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 Locked Heap 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 Locked Heap 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 allocation error. To address this, after constructing a heap, it is needed to initialize the allocator

**A Freestanding Rust Binary**

It is necessary to write code that is independent of any operating system features in order to create an operating system kernel. As a result, we are unable to use any functionality that depend on OS abstractions or particular hardware, including threads, files, heap memory, the network, random numbers, and standard output. Which makes sense given that we're attempting to create our own operating system and drivers. As a result, while we are unable to use the majority of the Rust standard library, there are several Rust features that we can make use of. We can use string formatting, iterators, closures, pattern matching, option and result, and, of course, the ownership system.

An executable that can be executed without an underlying operating system is required in order to build an OS kernel in Rust. These executables are frequently referred to as "freestanding" or "bare-metal" executables.

*Disabling the standard library*

The standard library, which depends on the operating system for capabilities like threads, files, and networking, is linked by default in all Rust crates. Additionally, it depends on libc , a component of the C standard library that works closely with OS services. We cannot utilize any OS-dependent libraries because our goal is to create an operating system from scratch. Therefore, we must use the no std attribute to prevent the standard library from being included automatically.

**Allocator designs**

*Design goals*

The management of the heap memory that is accessible is the duty of an allocator. When allocating memory, it must return any unused space and maintain track of memory released by deallocating calls so that it can be utilized once more. The most crucial need is that it never distribute memory that is already in use somewhere else because doing so would result in unpredictable behavior. In addition to accuracy, there are other additional design objectives. For instance, the allocator should minimize fragmentation and efficiently use the memory that is currently available. Additionally, it ought to scale to any number of processors and function effectively for concurrent workloads. It could even adjust the memory layout in relation to the CPU caches for maximum speed in order to enhance cache locality and prevent false sharing.

*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.

**A minimal rust kernel**

*Installing rust nightly*

The stable, beta, and nightly release channels for Rust are available. Spend a moment reading The Rust Book to learn more about how these channels differ from one another. Installing a nightly version of Rust is necessary since we will need some experimental capabilities for developing an operating system that are only accessible on the nightly channel. I strongly suggest using rust up to handle Rust installations. It makes it simple to upgrade them and lets you install nightly, beta, and stable compilers side by side. Run rust up override set nightly to use a nightly compiler for the current directory with rust up. As an alternative, you can place a file named rust-toolchain in the root directory of the project with the content nightly.

In placing so-called feature flags at the front of our code, the nightly compiler enables us to select a number of experimental features. By adding #![feature(asm)] at the top of our main.rs, we could, for instance, enable the experimental asm ! macro for inline assembly. Due to their extreme instability, these experimental features may be changed or eliminated in future Rust versions without prior notice.

**VGA Text Mode**

The VGA text mode is a simple way to print text to the screen. To print a character to the screen in VGA text mode, it must be written to the text buffer of the VGA hardware. The VGA text buffer is a two-dimensional array with typically 25 rows and 80 columns, which is directly rendered to the screen. Each array entry describes a single screen character through the following format:

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

\*Not actually ASCII, but a character set named code page 437 with some additional characters and slight modifications.

The VGA text buffer is accessible via [memory mapped I/O](https://en.wikipedia.org/wiki/Memory-mapped_I/O) to the address 0xb8000. That reads and writes to that address but don’t access the RAM but directly access the text buffer, so we can read and write through normal memory operations to that address. Memory-mapped hardware may not support all normal RAM operations.

**Testing**

Tests are Rust functions that verify that the non-tested code is functioning in the expected manner. The bodies of test functions typically perform some setup, run the code we want to test and assert whether the result is exactly what we expected. The simplest way to do tests in Rust is by using the test configuration and test macro. A simple test has mentioned below.

Text

Description automatically generated

Apparently the #[test] macro before the function indicates its testing function. So, the test runner knows it must run this as a part of the tests. If something fails, there are other ways of making a test fail besides panicking.

One of these ways is the assert macro. Assert takes a Boolean parameter if its true it does nothing, but if its false it panics. To check functions that should panic under certain circumstances, use attribute #[should\_panic]. This accepts optional parameter “excepted=” with the text of the panic message.

**CPU Exceptions**

CPU exceptions occur in various situations like when accessing an invalid memory address or when dividing by 0. To react those exceptions, we have to set up an interrupt descriptor table that provides handler functions. At the end, the kernel of our CPU will be able to catch breakpoint exceptions and resume normal execution after. An exception indicates that something is wrong with the current instruction. Like, the dividing by 0 problem the CPU interrupts its work and immediately calls a specific exception handler function depending on the exception type.

There are 20 different CPU exception types. The most important ones are:

* Page fault:  occurs on illegal memory accesses.
* Invalid opcode: occurs when the current instruction is invalid.
* General protection fault: occurs on various kinds of access violations.

Double fault: when and exception occurs, the CPU tries to call the according handler function. If another exception occurs while calling the exception handler, the CPU raises a double fault exception. This occurs when There’s no handler function registered for an exception.

Triple fault: occurs when cpu tries to call the double fault handler function, and it issues a triple fault.

The interrupt descriptor table is something we have in order to catch and handle exceptions. In this table we can specify a handler function for each CPU exception.

