Sri Lanka Institute of Information Technology

BSc. Honors in Information Technology Specialize in Cyber Security

Secure Operating Systems-IE2032
June 2022

Group Assignment

Members of Group

INDRAJITH G.B.T.G	**IT21229220**
VIDMAKA D.H.T	**IT21219016**
GUNASEKARA W.M.M	**IT21226496**
WIJESINGHE N.T	**IT21218576**

RUST

Programming Language

Table of Contents

Introduction

RUST created as a personal project by Graydon Hoare while working at Mozilla Research in year 2006. The programming language Rust is multi-paradigm and all-purpose. Concurrency, type safety, and performance are focused on Rust. Rust enforces memory safety, which is the requirement that all references point to valid memory, without needing the usage of a garbage collector or reference counting, which are present in other memory-safe languages. Rust's borrow checker keeps track of the object lifetime and variable scope of all references in a program during compilation to enforce memory safety while avoiding concurrent data races. This language popular for systems programming and also give access to high-level features including functional programming constructs. Rust has gained attention for its development as a more recent language, and academic programming languages research has focused on it. SML, OCaml, C++, Cyclone, Haskell, and Erlang are among the languages that most heavily affected Rust. Amazon, Discord, Dropbox, Facebook (Meta), Google (Alphabet), and Microsoft are a few of the businesses that have embraced Rust since its initial stable release in January 2014.[1]

From the developer's side, Rust has gained attention for its development as a more recent language, and academic programming languages research has focused on it. Large teams of developers with varied degrees of systems programming expertise are finding RUST to be a useful tool for team collaboration. Low-level code is vulnerable to a wide range of subtle defects that, in the majority of other languages, can only be found through thorough testing and meticulous code review by experienced developers. Rust's compiler serves as a gatekeeper by forbidding the compilation of code containing these elusive defects, such as concurrency flaws. The team can concentrate on the program's logic rather than hunting for flaws by working with the compiler.[2] To establish an operating system, which plays an important role, the kernel must first be designed. For an operating system to have the same qualities as the kernel, the kernel must be well-performing, memory-efficient, and stable. To create such an operating system, the programming language should include all necessary functionality. Numerous releases have been made available because Rust is still being developed.[3]

Performance

For those who seek stability and speed in a language, there is rust programming language. We refer to both the speed at which Rust enables you to build programs and the speed at which those programs can be created. Stability is provided via the tests included into the Rust compiler through refactoring and feature additions. Contrast this with the fragile legacy code in languages lack similar tests, which developers are frequently hesitant to change. Rust works to make safe code as quick as manual code by aiming for zero-cost abstractions and higher-level features that translate into lower-level code as quickly as possible.[2]

Rust uses zero cost abstraction to streamline the language without compromising performance. Monomerization, 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 expenses are incurred.

Additionally, this concept is used in the standard library to make sure that common types like collections don't need to be recreated, resulting in better performance and library interoperability. The trash collector absence, which has numerous advantages for performance, is the second noteworthy advantage of Rust.As the garbage collector must keep track of the memory in some way (often through reference counting) in order to ascertain when memory can be released, garbage collection adds overhead at runtime. This increases both memory usage and CPU usage.

The fact that garbage 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 results show that Rust leads in CPU time for two of the three algorithms and trails C and C++ in the third test by a very small margin. For all tests, Rust is only second to C in terms of memory consumption.[4]

Reliability

Reliability of RUST can be carried out on three subtopics. They are memory safety, type safety, thread safety.

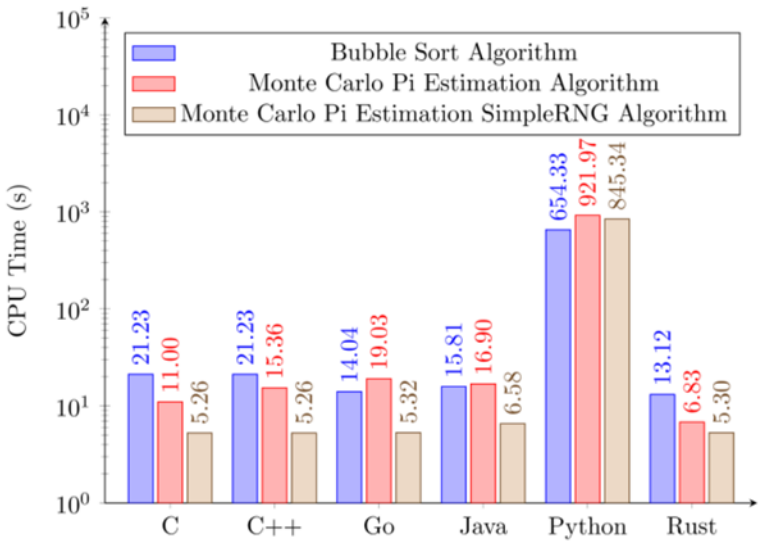


Figure 1- Average CPU time benchmark results [5]

Memory safety

Rust's unique ownership structure and the compiler's built-in borrow checking ensure memory safety at build time. Except when explicitly designated as such in an unsafe block or function, the compiler does not allow memory-unsafe code. Through the use of additional runtime checks and static compile-time analysis, Rust ensures memory safety while removing many different sorts of memory issues.[6]

At the language level, there is no idea of null. Instead, the Option enum in Rust is available and can be used to indicate whether a value is present or not. Users won't ever run into null pointer exceptions in Rust because the resulting code is null safe and a lot simpler to deal with.[6]

Rust is one of the languages with the highest memory efficiency because to its ownership and borrowing systems, which also help programmers avoid the problems associated with manual memory management and garbage collection. It is as fast and efficient with memory as C/C++, yet it has higher memory safety than garbage-collected languages like Java and Go. [6]

Type safety

When you access a variable in a type-safe language, you access it as the appropriate kind of data based on how it is stored. As a result, users are confident enough to work with data without manually verifying the data type at runtime. A language must have memory safety in order to be type safe. [6]

Rust is statically typed, and it ensures type safety through rigorous type checks at build time as well as through ensuring memory safety. Since most contemporary languages are statically typed, this is not unusual. Additionally, Rust supports some degree of dynamic typing, when necessary, by using the dyn keyword and Any type. Type safety is guaranteed by the compiler and sophisticated type inference even it has some circumstances. [6]

Thread safety

Users can allow many threads to simultaneously access or modify the same memory without having to worry about data races. To achieve this, thread synchronization, mutual exclusion locks, and message forwarding techniques are frequently utilized. Overall, languages that are type- and memory-safe also tend to be thread-safe since the best type and memory safety depend on thread safety. [6]

Rust offers common library features like channels, mutual exclusion locks, and ARC (Atomically Reference Counted) pointers, as well as a guarantee for thread safety using notions similar to those used for memory safety. Safe Rust allows for limitless read-only references to a value as well as one mutable reference at any given moment. Because of the ownership mechanism, a shared state cannot accidentally start a data race. This gives users the peace of mind to concentrate on the code and leave the compiler to handle shared data between threads. [6]

Productivity

Rust produces a lot of things. Rust also has a helpful compiler in addition to having excellent documentation. Rust also features an awesome twist tool, a constructed package manager, and intelligent multi-editor support.

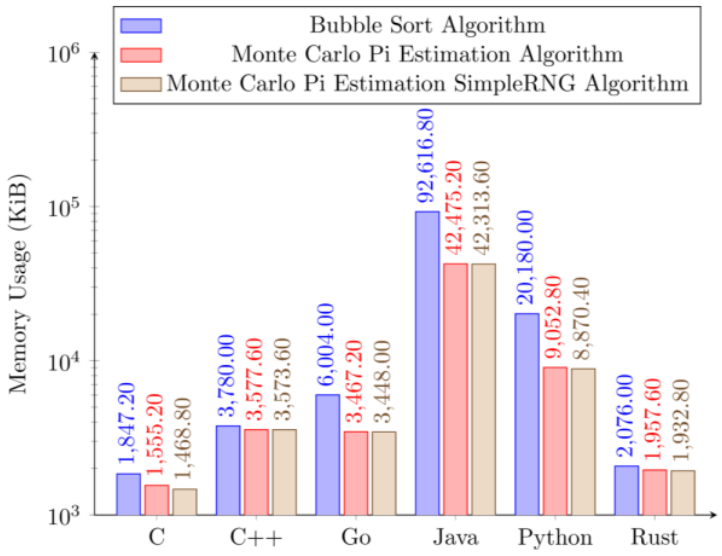


Figure 2- Average memory usage benchmark results [5]

A Freestanding Rust Binary

We require code that does not rely on any operating system capabilities in order to create an operating system kernel. This implies that none of the capabilities needing OS abstractions or particular hardware, such as threads, files, heap memory, the network, random numbers, or standard output, may be used. Which makes sense given that we're attempting to develop our own drivers and operating system.

This implies that while we are unable to access the majority of the Rust standard library, we can still make extensive use of many of its capabilities. Iterators, closures, pattern matching, option and result, string formatting, and, of course, the ownership system are just a few examples. Because of these properties, a kernel may be written in a very expressive, high-level manner without having to worry about ill-defined behavior or memory safety.

We need to construct an executable that can be executed independently of an underlying operating system in order to implement an OS kernel in Rust. Such an executable is frequently referred to as "freestanding" or "bare-metal."

Disabling the standard library

The standard library is connected by default throughout all Rust crates, and it is dependent on the system for features like as threads, files, and networking.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 property to prevent the standard library from being included automatically.

A fresh cargo application project is first created. Using the command line is the simplest method to accomplish this:

```
cargo new blog_os --bin --edition 2021
```

The —bin and —edition 2021 flags indicate that we want to build an executable binary rather than a library and that we want to use the 2018 edition of Rust for our crate, respectively.

The no_std Attribute

The built crate is presently implicitly referencing to the standard library.In order to prevent this, add the no std attribute:

```rust
// main.rs

#![no_std]

fn main() {}
```

And now you will get an error because compiler is missing a panic_handler function and a language item

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

Panic implementation

When a panic happens, the compiler should call the function specified by the panic_handler property. Although the standard library has a panic handler function, we must define it ourselves in a no std environment:

```rust
// in main.rs

use core::panic::PanicInfo;

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

The file, line, and optional panic message are all included in the PanicInfo argument. The function returns the "never" type to identify it as a diverging function because it should never return. Since there isn't much we can do with this function for now, we merely loop forever.

The eh_personality language item

Language items are unique types and functions that the compiler needs internally.

It is possible to provide customized implementations of language objects, although this should only be done in extreme cases. The cause is because language items are extremely shaky implementation details that aren't even type verified (the compiler therefore doesn't even verify if a function has the appropriate argument types). Fortunately, there is a more reliable solution to the language item problem mentioned above.

Stack unwinding is implemented via a function that is designated by the language item eh personality. In the event of a panic, Rust by default utilizes unwinding to execute all live stack variables' destructors. By doing this, all utilized memory is released and the parent thread is able to recognize the panic and carry on with execution. We don't want to utilize unwinding for our operating system because it is a difficult procedure and necessitates some OS-specific libraries (like libunwind on Linux or structured exception handling on Windows).

The start attribute

One would believe that when a program is run, the main function is the first one to be called. However, most programming languages feature a runtime system that handles tasks like garbage collection (for example, in Java) or software threads (e.g. goroutines in Go). Since this runtime needs to initialize itself, it must be called before main.

Execution begins in a C runtime library called crt0 ("C runtime zero"), which prepares the environment for a C program, in a typical Rust binary that connects the standard library. This entails building a stack and setting the arguments' register locations. The C runtime then executes the Rust runtime's starting point, signaled by the start language item. Rust only has a very modest runtime that handles a few minor tasks like installing stack overflow guards or displaying a backtrace on panic. The main function is then eventually called by the runtime.

Linker errors

An executable is created by the linker, a software that merges the produced code. Since Linux, Windows, and macOS all use distinct executable formats, each system's linker generates a unique error. The underlying reason for the problems is the same—the linker's default setting considers that our application depends on the C runtime, even though it does not.

We must instruct the linker not to include the C runtime in order to fix the issues. We may do this by either developing for a bare metal target or by providing the linker with a specific set of parameters.

A Minimal Rust Kernel

Under this topic we create a minimal rust kernel

The boot process

A computer starts running firmware code that is kept in motherboard ROM when you switch it on. The power-on self-test, RAM availability check, and pre-initialization of the CPU and hardware are all performed by this code. A bootable disk is then sought for, and the operating system kernel is then started.

The "Basic Input/Output System" (BIOS) and the more recent "Unified Extensible Firmware Interface" are the two firmware standards used on x86 processors (UEFI). Although the BIOS standard has been around since the 1980s, it is simple and extensively maintained. While UEFI is more advanced and has many more functions, it is more difficult to set up.

A minimal kernel

Now that we have a general understanding of how a computer boots, we can design our own simple kernel. Our aim is to produce a bootable disk image that prints "Hello World!" on the screen. We do this by extending the standalone Rust binary from the previous post.

Installing rust nightly

The stable, beta, and nightly release channels for Rust are available. Installing a nightly version of Rust is necessary since we will require some experimental capabilities for developing an operating system that are only accessible on the nightly channel.

With the help of so-called feature flags at the start of our file, the nightly compiler enables us to choose to use a variety of experimental features. The experimental asm! macro, for instance, may be made available for inline assembly by adding #![feature(asm)] at 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 utilize them unless it is absolutely essential because of this.

Target specification

Through the —target argument, cargo supports many target systems. A "target triple" that details the CPU architecture, the vendor, the operating system, and the ABI is used to characterize the target. The target triple x86 64-unknown-linux-gnu, for instance, refers to a machine having an x86 64 CPU, an unknown Linux distribution, and the GNU ABI. Rust supports a wide range of target triples, such as wasm32-unknown-unknown for WebAssembly and arm-linux-androideabi for Android.

Building our Kernel

```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 {}
}
```

By specifying the name of the JSON file as the —target argument, we can now create the kernel for our new target.

Running our Kernel

It's time to execute the executable now that it performs a noticeable action. Our built kernel must first be linked with a bootloader to create a bootable disk image. The disk image may then be used to boot actual hardware via a USB stick or run in the QEMU virtual machine.

Creating a boot image

We must connect our generated kernel with a bootloader in order to create a bootable disk image. The bootloader is in charge of loading our kernel and initializing the CPU, as we discovered in the section on booting.

VGA Text Mode

Printing text on the screen is easy when using the VGA text mode. By enclosing every unsafety in a distinct module under this subject, we develop an interface that makes its usage secure and straightforward. We also provide support for the formatting macros in Rust.

The VGA text buffer

A character must be written to the VGA hardware's text buffer before it can be shown on the screen in VGA text mode. The VGA text buffer is a two-dimensional array that is immediately projected to the screen. It normally has 25 rows and 80 columns. The following structure is used in each array element to describe a single screen character:

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

The ASCII character that should be printed is represented by the first byte. To be more precise, it isn't exactly ASCII; rather, it is a character set known as code page 437 with a few more characters and minor changes.

The second byte specifies the character's visual representation. The foreground color is determined by the first four bits, the background color by the next three bits, and whether the character blinks is determined by the last bit. There are the following hues available:

Number	Color	Number + Bright Bit	Bright Color
0x0	Black	0x8	Dark Gray
0x1	Blue	0x9	Light Blue
0x2	Green	0xa	Light Green
0x3	Cyan	0xb	Light Cyan
0x4	Red	0xc	Light Red
0x5	Magenta	0xd	Pink
0x6	Brown	0xe	Yellow
0x7	Light Gray	0xf	White

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

Keep in mind that not all typical RAM operations may be supported by memory-mapped devices. For instance, while reading an u64, a device could only enable byte-wise reads and return garbage. We don't need to treat the text buffer differently because it allows standard reads and writes.

Testing

The following attributes are needed to define the routines that will be used to run tests. Building the test functions and a test harness for running the tests is possible when compiling a crate in "test" mode. The test conditional compilation option is enabled when the test mode is enabled as well.

Rust methods known as tests are used to check that non-test code is operating as intended. Typically, the test function bodies carry out the following three tasks:

- Create any necessary states or data.
- Test the code by running it.
- Assert the results are what you expect.

The test attributes

In Rust, a test is essentially a function that has the test attribute applied. Add **#[test]** to the line preceding the function to turn it into a test function.

```
#[test]
fn test_the_thing() -> io::Result<()> {
    let state = setup_the_thing()?; // expected to succeed
    do_the_thing(&state)?;          // expected to succeed
    Ok(())
}
```

The ignore attribute.

The ignore attribute may also be used to annotate a function that has the test attribute. The test harness is instructed not to run that function as a test using the ignore attribute. In test mode, it will still be compiled.

It can sometimes take a long time to run a few particular tests. The ignore attribute can be used to disable these by default.

To give a reason why the test should be disregarded, the ignore attribute can optionally be specified using the *MetaNameValueStr* syntax.

```
#[test]
#[ignore = "not yet implemented"]
fn mytest() {
    // …
}
```

The should_panic attribute.

Should panic annotations may also be added to functions with the test and return values of (). The test can only pass if it truly panics thanks to the should panic attribute.

An optional input string that must be included in the panic message can be provided for the should panic attribute. The test will fail if the specified string cannot be found in the message. The string can be given using either the MetaListNameValueStr syntax with an expected field or the MetaNameValueStr syntax.

```
#[test]
#[should_panic(expected = "values don't match")]
fn mytest() {
    assert_eq!(1, 2, "values don't match");
}
```

There is one additional difference between exception handlers and simple functions besides the exception attribute: software cannot invoke exception handlers. A compilation error would be produced by the statement SysTick(); if we were to follow the previous example.[2]

CPU Exceptions

A hardware mechanism known as exceptions and interrupts allows the CPU to deal with asynchronous events and fatal errors (e.g. executing an invalid instruction). Exceptions involve exception handlers, which are subroutines run in response to the signal that caused the event, and therefore imply preemption.

Here is the example for it:

```
// Exception handler for the SysTick (System Timer) exception
#[exception]
fn SysTick() {
    // ..
}
```

There is one additional difference between exception handlers and simple functions besides the exception attribute: software cannot invoke exception handlers. A compilation error would be produced by the statement *SysTick()*; if we were to follow the previous example.

Static mut variables declared inside exception handlers are safe to use, hence this behavior is essentially intended and necessary to provide the feature.

```
#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;

    // `COUNT` has transformed to type `&mut u32` and it's safe to use
    *COUNT += 1;
}
```

The default exception handler.

The default exception handler for a particular exception is actually replaced by the exception attribute. The *DefaultHandler* function, which by default handles the following exceptions, will take care of them if you don't override the handler for a specific exception:

```
fn DefaultHandler() {
    loop {}
}
```

The hard fault handler.

An unusual exception is the *HardFault* one. When the program enters an invalid state, this exception is thrown, and its handler cannot continue because doing so can lead to unexpected behavior. Additionally, to aid in debugging, the runtime crate performs some work before invoking the user-defined HardFault handler.[2]

Double Faults

A double fault, defined simply, is a specific exception that happens when the CPU neglects to call an exception handler. When a page fault is triggered, for instance, but no page fault handler is registered in the interrupt descriptor table, it happens (IDT). As a result, it is somewhat comparable to catch-all blocks in programming languages with exceptions, such as catch(...) in C++ or catch (Exception e) in Java or C#.

A double fault acts like any other exception would. Its vector number is 8, and the IDT allows us to construct a standard handler function for it. It is crucial to include a double fault handler because, if left unattended, a double fault can result in a fatal triple fault. Triple faults are impossible to detect, and most hardware responds by resetting the system.

A double fault handler.

We may give a handler method identical to our breakpoint handler for a double fault as it is a common exception with an error code:

```
// in src/interrupts.rs

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

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

Our handler dumps the exception stack frame along with a brief error message. There is no need to print the double fault handler's error code because it is always zero. The double fault handler differs from the breakpoint handler in that it is divergent. The x86 64 architecture forbids returning from a double fault exception, which is the cause.

We terminate QEMU with a success exit code once the double fault handler is called, indicating that the test was successful.

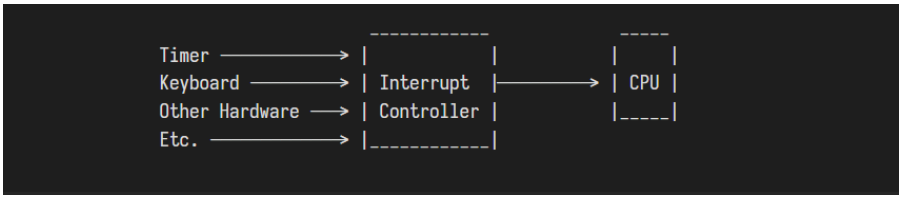
Summary.

In this article, we learnt what a double fault is and when it happens. We implemented an integration test and a straightforward double fault handler that outputs an error message.

In order for stack overflow exceptions to be handled, we also enabled the hardware-supported stack switching on double fault exceptions. We discovered the task state segment (TSS), containing interrupt stack table (IST), and global descriptor table (GDT), which were utilized for segmentation on earlier architectures, when putting it into practice.[2]

Hardware Interrupts

The CPU can be notified by associated hardware devices using interrupts. So, the keyboard may inform the kernel of each keypress rather than allowing the kernel regularly to inspect the keyboard for new characters (a process known as polling). This is substantially more efficient because the kernel only needs to act when anything happens. Not only that but also it allows faster reaction times because the kernel can respond instantly and not only at the next poll.



It is not possible to connect all hardware devices directly to the CPU. Therefore, a separate interrupt controller gathers the interrupts from all the devices and then alerts CPU. Most interrupt controllers may be programmed; thus, they offer a range of interrupt priority levels. To guarantee precise timekeeping, for example, this allows providing timer interrupts a greater priority than keyboard interrupts.

Hardware interrupts take place asynchronously, unlike exceptions. It means these hardware's are fully independent from the running code and it can occur at any time. So, all the potential concurrency-related issues suddenly appear in our kernel in the shape of a type of concurrency. Our situation is made easier by the rigid ownership concept of Rust, which precludes changing global state.

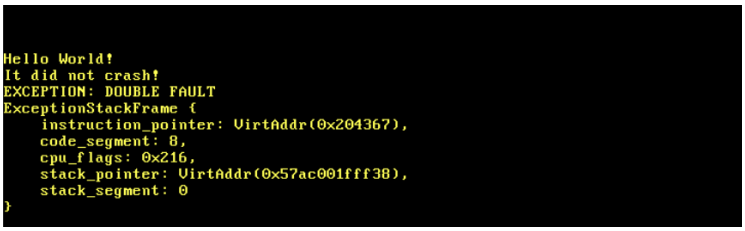
Enabling Interrupts

The x86 64 crate's `interrupts:enable` method uses the unique `sti` instruction ("set interrupts") to activate external interrupts. When we attempt cargo run now, we notice a double fault,

```
// in src/lib.rs

pub fn init() {
    gdt::init();
    interrupts::init_idt();
    unsafe { interrupts::PICS.lock().initialize() };
    x86_64::instructions::interrupts::enable();     // new
}
```

The hardware timer gets enable by default. Therefore, we are receiving timer interrupts as soon as we enable interrupts. This causes the double faults in hardware interrupts.



Handling Timer Interrupts

According to the figure 2, we can see that the timer uses line 0 of the primary PIC. In other words, it enters the CPU as interrupt 32 (0 + offset 32). We save index 32 in an InterruptIndex enum rather to hardcoding it.

```
// in src/interrupts.rs

#[derive(Debug, Clone, Copy)]
#[repr(u8)]
pub enum InterruptIndex {
    Timer = PIC_1_OFFSET,
}

impl InterruptIndex {
    fn as_u8(self) -> u8 {
        self as u8
    }

    fn as_usize(self) -> usize {
        usize::from(self.as_u8())
    }
}
```

The signature of our timer interrupt handler is the same as that of our exception and external interrupt handlers. Due to the IndexMut trait being implemented in the InterruptDescriptorTable struct, we may access specific elements using array indexing syntax. We print a dot on the screen each time a timer tick happens in our timer interrupt handler.

End of Interrupts

It is dangerous to utilize the function notify end_of _interrupt. The PIC anticipates an explicit "end of interrupt" signal from our interrupt handler, which is the main cause of the issue. The system is now prepared to receive the next interrupt, according to this signal, which informs the controller that the interrupt has been handled.[2]

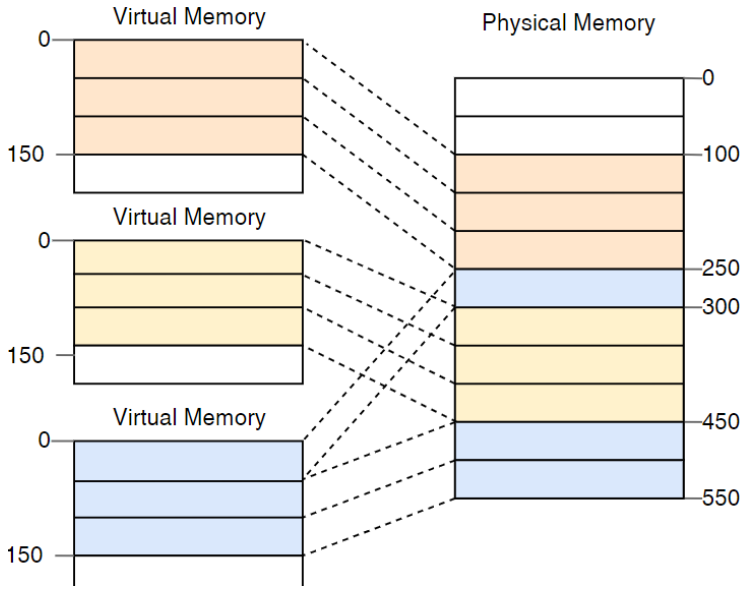
```
// in src/interrupts.rs

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

    unsafe {
        PICS.lock()
            .notify_end_of_interrupt(InterruptIndex::Timer.as_u8());
    }
}
```

Introduction to Paging

Virtual memory and physical memory space can be divided in to small, fixed sized blocks. These virtual memory boxes are called pages. Physical address spaces are called frames. Each page can be mapped into frame separately. It makes possible to split larger memory regions across non-continuous physical frames. If we review the fragmented memory space example, but utilize paging rather than segmentation this time, the benefit of this becomes apparent.



Program session has its own page table on x86. In a particular CPU register, a reference to the presently active table is saved. The CPU reads the table pointer from this register with each memory access. The software running is solely handled by hardware so it is fully unaware of this.

Hidden Fragmentation

Paging employs several tiny, fixed-sized memory sections as opposed to a few big, variable-sized regions, as opposed to segmentation. Because every frame is the same size, there are no frames that are too tiny to be utilized, and so no fragmentation occurs. Because not every memory area is an exact multiple of the page size, internal fragmentation happens.

Page Tables

Millions of pages are uniquely matched to a frame. In contrast to paging, segmentation employs a separate segment selection register for each active memory region. Instead, paging stores mapping information in a table structure called a page table.

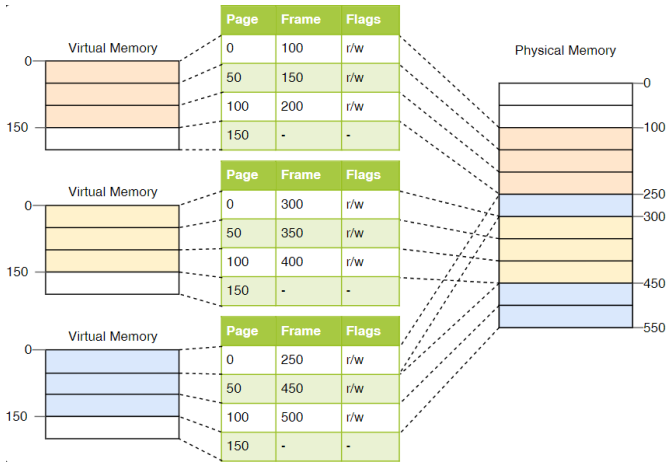
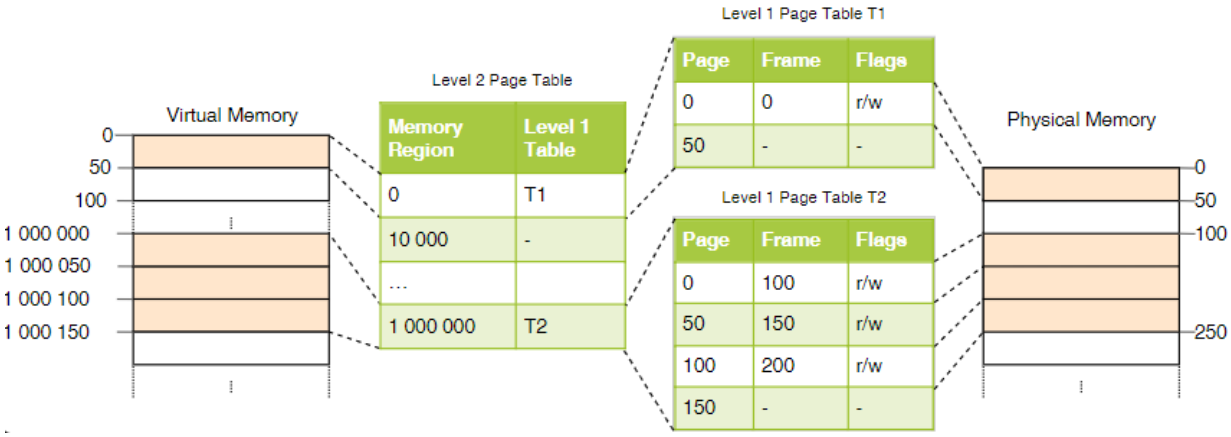


Figure 3-how pegging table look like

Multilevel Page Tables

It includes the mapping between (level 1) page tables and address areas. if we eliminated the empty entries the CPU cannot be able to move quickly to the proper entry in translation. We can use a two-level page table to reduce wastage memory.

this can be discussed by an example. Let's assume that each level 1 page table is responsible for a region of size 10_000.



In this illustration, the level 2 table has 100 empty entries, which is significantly fewer than the previous million empty items. These savings are possible because we don't have to build level 1-page tables for the unmapped frame. It is possible to add three, four, or even more levels to the two-level page ables idea.[2]

Paging Implementation

Every memory address that we use in our kernel was a virtual address. Accessing the VGA buffer at address 0xb8000 was only feasible because the bootloader identity mapped the memory page, implying that it mapped virtual page 0xb8000 to physical frame 0xb8000.

Kernel gets relatively safe by paging. Because every memory access that is outside of boundaries results in a page fault exception rather than writing to arbitrary physical memory. The bootloader even configures access rights for each page, ensuring that only code pages are executable and data pages are readable.

Page Faults

To see page faults, first we must register it in our IDT. So we can see page fault exception without double fault.

```
// in src/interrupts.rs

lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();

        [...]

        idt.page_fault.set_handler_fn(page_fault_handler); // new

        idt
    };
}

use x86_64::structures::idt::PageFaultErrorCode;
use crate::hlt_loop;

extern "x86-interrupt" fn page_fault_handler(
    stack_frame: InterruptStackFrame,
    error_code: PageFaultErrorCode,
) {
    use x86_64::registers::control::Cr2;

    println!("EXCEPTION: PAGE FAULT");
    println!("Accessed Address: {:?}", Cr2::read());
    println!("Error Code: {:?}", error_code);
    println!("{:#?}", stack_frame);
    hlt_loop();
}
```

The PageFaultErrorCode class offers extra details regarding the kind of memory access that resulted in the page fault, such as whether a read or write operation was to blame. We read and output it using the x86 64 crate's Cr2::read method. We should resolve the page fault in order to continue execution. So we should enter hlt_loop at the end.

```
// in src/main.rs

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

    blog_os::init();

    // new
    let ptr = 0xdeadbeaf as *mut u32;
    unsafe { *ptr = 42; }

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

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

It is obvious from the error code's CAUSED_BY_WRITE that a write operation was attempted at the time the problem occurred. Reading from this URL works, however writing from it results in a page fault. The absence of the PROTECTION_VIOLATION flag indicates the absence of the target page.

We notice that the current instruction pointer is 0x2031b2. That shows this location refers to a code page. Reading from this location succeeds but writing results in a page fault because code pages are bootloader-mapped read-only. Try this by altering the pointer from 0xdeadbeaf to 0x2031b2.

```
// Note: The actual address might be different for you. Use the address that
// your page fault handler reports.
let ptr = 0x2031b2 as *mut u32;

// read from a code page
unsafe { let x = *ptr; }
println!("read worked");

// write to a code page
unsafe { *ptr = 42; }
println!("write worked");
```

Page Table Accessing

The x86 64 Cr3::read function reads the CR3 register to return the presently active level 4 page table. PhysFrame and Cr3Flags type are returned as a tuple. Because we are only concerned with the frame, we disregard the second member of the tuple.

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

    blog_os::init();

    use x86_64::registers::control::Cr3;

    let (level_4_page_table, _) = Cr3::read();
    println!("Level 4 page table at: {:?}", level_4_page_table.start_address());

    [...] // test_main(), println(...), and hlt_loop()
}
```

Below picture shows how the output looks like.

```
Level 4 page table at: PhysAddr(0x1000)
```

The current active level 4-page table is in physical memory at address 0x1000, as indicated by the PhysAddr wrapper type. When paging is enabled, it is not feasible to access physical memory directly. A virtual page that is mapped to the physical frame at address 0x1000 is the sole means to access the table.[2]

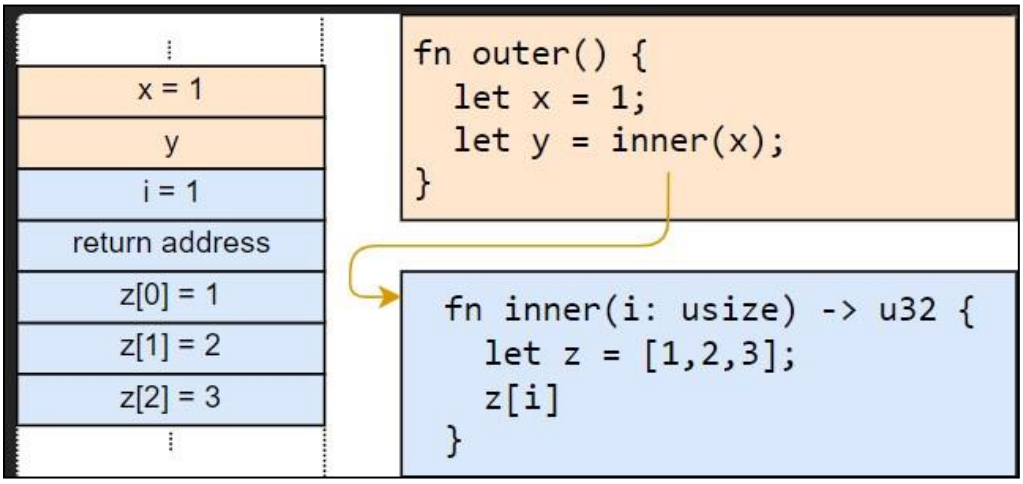
Heap Allocation

The specifics vary depending on the allocator being used, but each deallocation and allocation often includes obtaining a global lock, doing some complex data structure modification, and potentially making a system call.

Smaller allocations are not always less expensive than larger allocations. Because avoiding them can significantly increase performance, it is important to understand which Rust data structures and functions result in allocations.

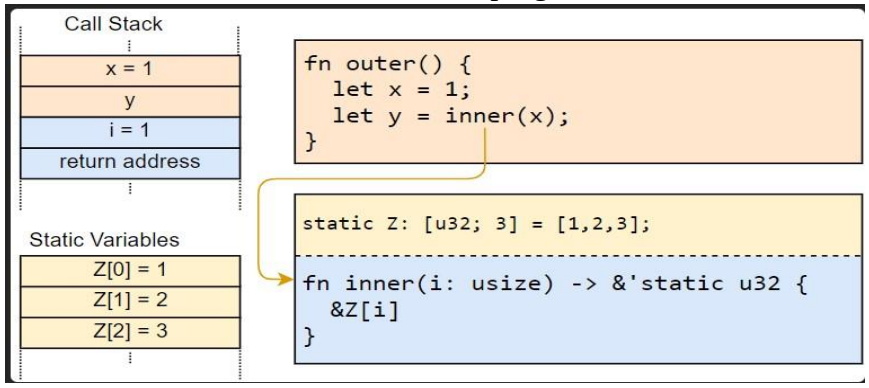
The kernel stores data in two different sorts of variables, such as Static Variables and local variables.

Local variables: The compiler pushes the caller function's local variables, return address, and parameters on each function entry.



The call stack for each thread contains local variables. The push and pop actions are supported by the stack data structure called call stack. Values that are specific to the surrounding function are stored in local variables. When the function returns, its data will have vanished.

Static variables: Statics have a "static lifetime," which means they are always available for reference from local variables for the duration of the program.



Static variables are associated with a particular procedure. The stack is not where these values are kept; they are kept at a fixed location in memory. Until the procedure is over, they will be kept there. At the compile time, static variable locations are specified.

An important flaw exists in both abovementioned variable types. Variable sizes are fixed at the time of compilation. Thus, as time and usage pass, the variables cannot increase further. "Heap" is applied to get around that.

Some programming languages support the third memory area known as the heap. Through the functions "allocate" and "deallocate," it provides dynamic memory allocation. Manually carrying this out is not advised. So, the term "Waste collection," which may be found in Python and Java, is introduced as an automated operation. Performance overhead results from it being a running process as well.

As it eliminates the need for a garbage collector and boosts efficiency, Rust's ownership model further verifies the accuracy of the dynamic memory operations at compile time. Programmers still have access to memory, just like in C or C++ languages.

Managing the heap is done differently in Rust. To manage the heap, it employs the "Borrow Checking" technique. It is essentially the same as borrowing something with the understanding that you would return it after you're done using it without damaging it. The Borrow Checker in Rust does the same task as ensuring that all borrowing terminates before an object is destroyed. This ensures that there will never be a use-after-free scenario, according to Rust's compiler.[7]

Allocator Designs

Let will demonstrate how to build a custom heap allocation from scratch rather than depending on an already-built allocator crate. In order to create an allocator with higher speed, we will cover various allocator designs, such as a straightforward bump allocator, linked list allocator, and a straightforward fixed-size block allocator.

An allocator's function is to monitor the heap memory that is currently available. When using alloc, it must return any unused memory and track memory that has been released by dealloc so that it can be used once more. More significantly, this should never distribute memory that is already being used by another process because doing so would result in unpredictable behavior. There are other additional design objectives in addition to accuracy. For instance, the allocator should minimize memory fragmentation and efficiently use the available memory. It should also scale to any number of processors and perform effectively for concurrent workloads. Even the memory structure in relation to CPU caches could be optimized for maximum speed to enhance cache locality and prevent incorrect shared.

Below, we outline three potential kernel allocator architectures and examine the strengths and flaws.

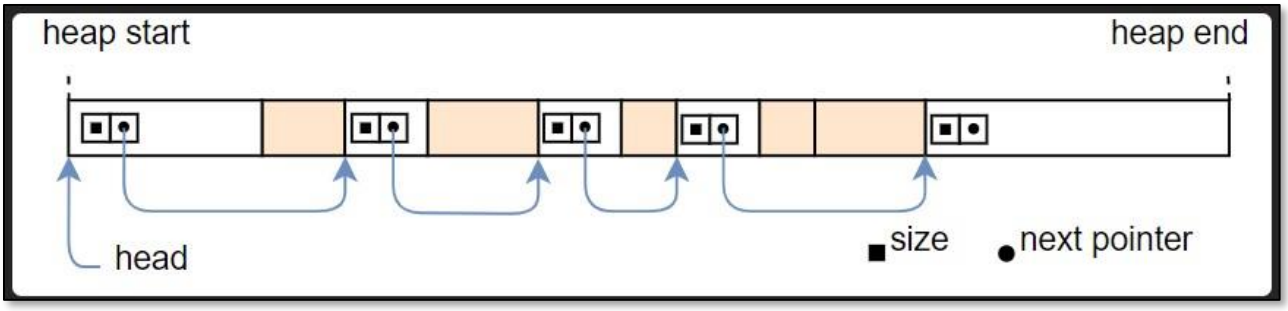
Bump Allocator/Stack Allocator

A bumped allocator is the simplest allocator design. It just maintains track of the number of allocations and the linear allocation of memory, not the total number of allocations. Since it has a significant constraint and can only release full memory at once, it is only helpful in extremely particular use scenarios. A bump allocator's primary restriction is that it cannot reuse deallocated memory once all allocation has already been released. Because of this, memory reuse can be avoided with just one large allocation.

Linked List Allocator

Using the open memory spaces itself as backing store is a typical technique used when developing allocators to keep a record of any amount of available memory spaces. The stored data is no longer required, but the regions are still allocated to a virtual address as well as supported by a physical frame.

Construction of a single queue in the released memory, for each node representing a freed memory sector, is the most typical implementation method:



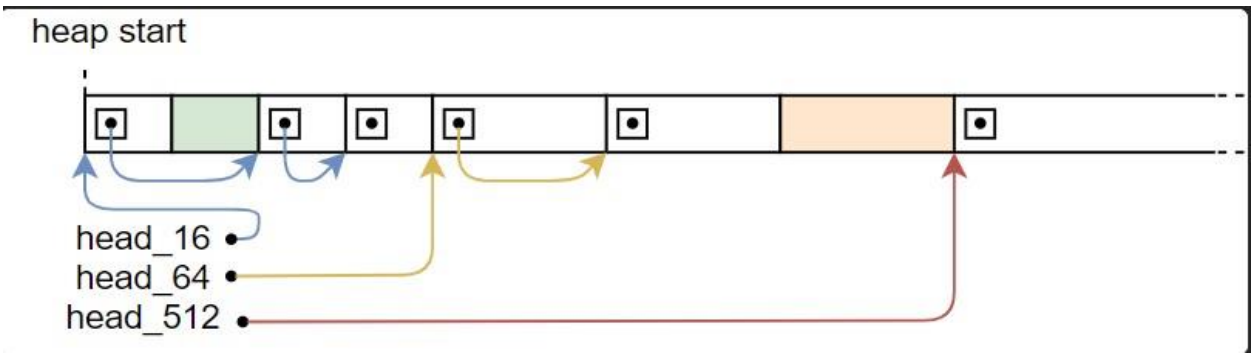
The amount of the memory region and a pointer to the subsequent unoccupied memory region are both fields found in each list node. No matter how many unused areas there are, we only require a pointer to the first one (known as the head). Usually referred to as a free list, the resulting data structure.

The bump allocator is quite quick and can be tailored to just require just few assembly processes. The issue is that, in some cases, an allocation request may have to traverse the entire linked list before it identifies an appropriate block. Performance varies greatly between programs since the list length is based on the quantity of unused memory blocks.

Fixed-size block Allocator

The following is how a fixed-size block allocator works, we provide a limited amount of block sizes and round up each allocation to the following block size rather than allocating exactly the amount of RAM that was requested. A 4-byte allocation would result in a 16-byte unit, a 48-byte allocation in a 64-byte block, as well as a 128 byte allocation in a 512-byte block, for instance, when using blocks of 16, 64, & 512 bytes.

We maintain a linked list in the unused memory, similar to the linked list allocator, to keep track of the available memory. Instead of employing a single array with various block sizes, we instead make a distinct list for each size class. Consequently, each list only keeps blocks that of a specific size.



[2]

The fixed-size block allocator design can also be found in a variety of forms. The slab allocator and the buddy allocator, two prevalent instances that are also present in well-known kernels like Linux.

This merge procedure has the benefit of reducing external fragmentation, allowing for the reuse of small, freed blocks for large allocations. Additionally, it doesn't employ a fallback allocator, making the performance more reliable. The only conceivable block sizes are power-of-2, which has the largest problem of potentially wasting a lot of RAMS owing to internal fragmentation.

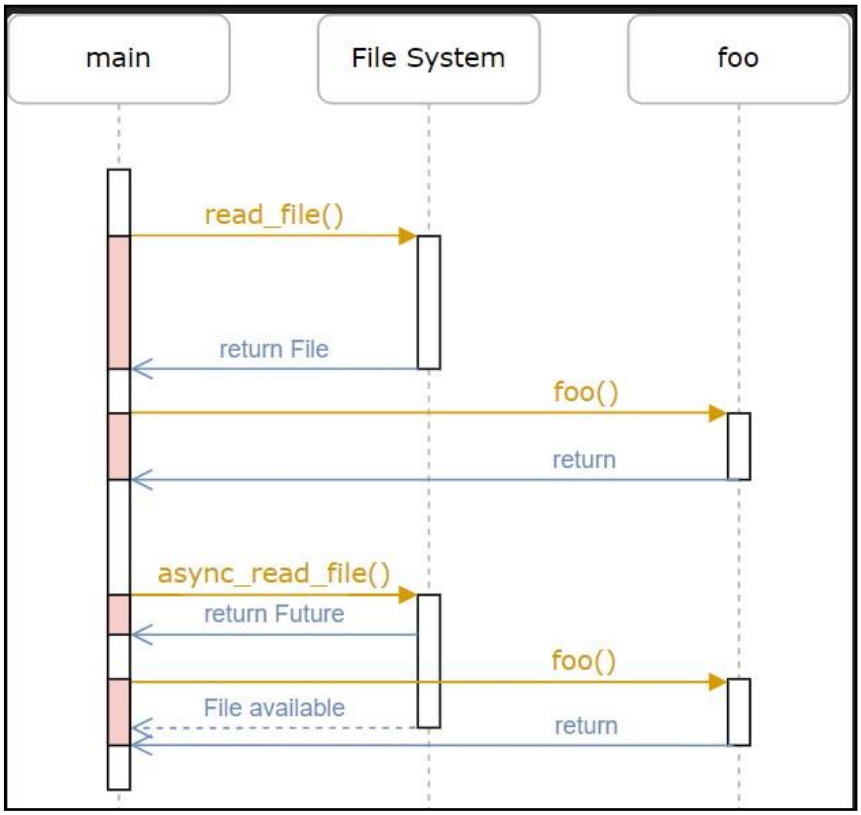
Async/Await

Async/await, a feature of the Rust language, offers first-rate assistance for collaborative multitasking. Before analyzing what async/await is, it is necessary to comprehend how futures and asynchronous programming in Rust function.

Futures:

A future denotes a value that may not be accessible right away. Futures allow us the option of continuing execution till the value is required rather than waiting until it is available.

The greatest way to explain the idea of futures is with a below example:[9]



Async fn and async blocks are the two primary applications of async. Each gives a result that reflects 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
    }
}
```

Futures, including async bodies, are inactive until they are executed. Waiting is the most typical method of running a Future. A Future will attempt to be executed to completion when the function. Await is invoked on it.

Async lifetime:

As opposed to conventional functions, async fns that accept references or other non-static parameters return a Future that is constrained by the lifetimes of the arguments:

The future delivered by an async function must therefore be. Awaited while its non-static inputs are still valid. This is not a problem when the function is called frequently, and the future is. awaiting right away. This might be a problem, though, if the future is stored or transferred to another job or thread.

Async move:

The move keyword is supported in async blocks and closures, similarly to regular closures. An async move block will acquire ownership of variables it references, enabling it to survive beyond the current scope but preventing it from sharing them with the other code:

Awaiting on multi-threaded executor:

Each and every variable used in async bodies must be capable to transit between threads since a Future may move between threads when utilizing a multithreaded Future executor. A switching to a new thread may happen because of await.

This means that it is risky to utilize types like Rc, or references to types which don't define the Sync trait if they don't also implement the Send trait.

Holding a standard non-futures-aware lock across an.await is not a smart idea as it may cause the thread pool to lock up: one job could take out a lock, Await, and yield to the executor, which would allow another task to try to pull the lock and result in a deadlock. Just use Mutex in futures contracts to prevent this.

References

[1] Wikipedia

[2]The RUST programming Language book https://doc.rust-lang.org/book/ch00-00-introduction.html#:~:text=Rust%20is%20for%20people%20who%20crave%20speed%20and%20stability%20in,through%20feature%20additions%20and%20refactoring.

[3] How Rust can be used to Implement a better Operating System https://medium.com/sliit-foss/how-rust-can-be-used-to-implement-a-better-operating-system-fe1d4327156c

[4] Rust: The Programming Language for Safety and Performance by William Bugden1 and Ayman Alahmar*,1

[5] Bugden, William and Alahmar, Ayman. "The Safety and Performance of Prominent Programming Languages." International Journal of Software Engineering and Knowledge Engineering (2022).

[6] Why Safe Programming Matters and Why a Language Like Rust Matters https://developer.okta.com/blog/2022/03/18/programming-security-and-why-rust

[7] "The Rust Programming Language," [Online]. Available: https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/the-stack-and-the-heap.html#:~:text=Memory%20management&text=The%20stack%20is%20very%20fast,size%2C%20and%20is%20globally%20accessible..

[8] P. Oppermann, "Writing an OS in Rust," 20 January 2020. [Online]. Available: https://os.phil-opp.com/allocator-designs/#linked-list-allocator.

[9] P. Oppermann, "Writing an OS in Rust," 27 March 2020. [Online]. Available: https://os.phil-opp.com/async-await/#async-await-in-rust.

Conclusion

When it comes to developing a high-performance concurrent system with a small resource footprint, the programming languages available are limited. Languages with interpreters frequently perform poorly in settings with lots of concurrent users and few resources. For these use situations, system programming languages are the best options.

When there are no resource restrictions and high concurrency is not required or can be done via alternative methods, such as event loops, interpreter-based languages might be preferable.

The pinnacle of systems programming languages is C++. But there's a reason C and C++ are one of the languages people fear using the most, according to StackOverflow surveys. It might be challenging for programmers to transition to C/C++ from other high-level languages. There's a important learning curve. There are roughly 74 different build systems and a patchwork of 28 different package managers, the most of which are tied to addressing an unified platform or environment and useless elsewhere. After more than 30 years of evolution, young programmers are inundated with information.

Contrarily, Rust is more approachable, has a big user base, has no significant technical debt, and nonetheless delivers comparable performance. Simpler concurrency and memory safety are only extra advantages.