Sri Lanka Institute of Information technology

Year 2, Semester 1

Secure Operating Systems -IE2032

Implement more secure, reliable, high-performance operating system Using Rust

Submitted by:

Student Registration Number	Student Name
DE ZOYSA A.S.	IT21167096
NILUPUL S.A.	IT21167478
MADURANGA D.B.W.N.	IT21170270
BANDARA K.M.N.M	IT21163418

Table of Contents

INTRODUCTION

Rust is a static compiled language with a rich type system and a memory- and time-efficient ownership notion. While guaranteeing memory and thread safety, it may power performance-critical services, enabling developers to troubleshoot at build time. Additionally, Rust has a multi-editor with features like type inspection and auto-completion, a user-friendly compiler with complex features like integrated package management, and great documentation. Rust prevents all crashes, and it's important to note that, like Python, Ruby, and JavaScript, Rust is secure by default. Due to the fact that we cannot create flawed parallel code and thus Rust can never have a bug, this is far more powerful than C/C++. It can convey a huge variety of programming ideas very quickly.

But why create a new programming language when there are already so many options accessible, including Python, C++, and Java? The second programming language is less secure, but it provides you a lot more control over the hardware you're using, including the ability to optimize it and convert directly to assembly code.

Rust therefore provides us with all of the security levels and controls that are available.

Although some C programming is utilized, the majority of it is done in Rust, which implies that all of the common compilers and libraries are written in the language. The main project in Rust is called "servo," which aims to create a layout engine that runs entirely in parallel, like a gecko in Firefox or WebKit in Safari. The layout engine was developed by Servo to render HTML top to bottom.

A Standalone Rust Binary

This report will show you how to create a Rust binary that can run without the need for an operating system. In order for the standard library to function, the operating system must provide threads, files, and networking. Every Rust creates links to the standard library by default. A pattern matching system, iterators, closures, string formatting, and pattern matching are all standard aspects of Rust. Because of these properties, high-level kernels with high expressiveness may be created. The Cargo. toml file contains information about the inventor, the semantic version number, the name of the crate, and a list of dependents.

After using cargo build to construct your crate, you may start the blog of binary from the target/debug directory. Add the no std property to our function to see if we can prevent this from happening. For the language item, this exhibits the personality attribute of stack unwinding. When a panic occurs, Rust invokes the destructor of each current stack variable. Because all previously consumed memory has now been freed, the parent thread may become aware of the panic and continue working

A Rust binary will often begin functioning using the crt0 runtime library (sometimes known as "C runtime zero") ("C runtime zero"). In order to keep the inputs in the correct registers, a stack must be constructed. The C runtime starts the start language item, which is a representation of the Rust runtime. We must create our own entry point since our lone executable does not have access to it.

The return type is a diverging function that is not allowed to return. This is essential since no function other than the operating system or bootloader contacts the entry point. There are two methods to overcome this problem: provide linker options or write code for bare metal targets. Using the —target argument, we may cross-compile our executable for a bare metal target machine. Because the target machine lacked an operating system, the linker did not attempt to link the C runtime, demonstrating that our build was successful.

In this manner, we will build the kernel for our operating system. The following is a simple freestanding Rust binary: Because the Rust standard library looks for functions with the name "_start" by default, using main.rs / Connecting is not recommended.

When the _start function is called, this binary requires the establishment of a stack, among other things. Without extra inputs, such a binary cannot be used correctly on the host machine.

A Simple Rust Kernel

Rust is highly recommended for managing Rust installations since it allows you to install nightly, beta, and stable compilers at the same time. By including feature flags at the beginning of our file, we may enable the experimental features in the nightly version of Rust. Because our kernel also supports x86 64 CPUs, our goal requirements will be the same. The majority of the fields are required for LLVM to generate code for that platform. The widths of various integer, floating point, and pointer types are listed in the data-layout field.

The linker arguments are stored in the pre-link-args field. Large SIMD registers may cause performance issues since the OS kernel must reset all registers to their original states before continuing a halted application. To avoid this, we disable SIMD in our kernel (but not in the programs that run on top of it) and simulate all floating point operations using software functions based on standard integers (but not for apps running on top of it). Instead of relying on precompiled versions supplied with the Rust installation, the core and other standard library crates may be recompiled as needed using the cargo's build-std capabilities. This feature is labeled as "unstable" due to its ongoing development and the fact that nightly Rust compilers are the only ones that can utilize it.

According to the Rust compiler, each system has a set of built-in functions. Programs that change the values of all bytes in a memory block include memset, memcpy, and memcmp. Because we cannot link to the operating system's C library, we need a new way to transmit these functions to the Rust compiler. We may enable them by modifying the cargo build-std-features option to ["compiler-builtins-mem,"]. We can now use a simple cargo build to produce our kernel for a bare metal target.

The contents of the VGA text buffer may now be shown on the screen. All that is required to print "Hello World" is the buffer address 0xb8000 and an awareness that each character cell consists of an ASCII byte and a color byte. The example code below generates a raw pointer from the integer 0xb8000. Walking over each byte of the static HELLO byte string, In addition, the enumerate method is used to construct a running variable called I. Using the offset method, the for loop publishes the string byte and the corresponding color byte.

Text Mode in VGA

A character can only be shown on the screen in VGA text mode after it has been written to the hardware's text buffer. We created the VGA buffer module to represent the many colors used by Rust for color coding.

To assign a number to each color, a C-style enum is utilized. The type is made printable and equivalent to C by using the #[allow(dead code) tag, which also allows the type to have copy semantics. We can now write characters to the VGA text buffer using Rust's Writer and Buffer. The Writer uses the repr(C) capability to ensure that its fields are in the same order as in a C struct. When a line is full (or on), it is pushed up and is on, and it always writes to the last line while keeping track of where it is in the last row.

Before beginning to write a byte, the Writer ensures that the line is empty. In this case, it replaces the Buffer's current ScreenChar. During the second match instance, more bytes are shown on the screen. We can show whole phrases by converting words to bytes and publishing each byte separately. In our write string function, we test the acceptability of nonprintable characters by writing the strings "Hello!" and "Wörld!"

To display the results, use the print method of our _start function in src/main.rs: [No Mangle]. Instead of reading from the Buffer, we write to it. The compiler is completely oblivious that some characters are being shown incorrectly on the screen and that we are really accessing VGA buffer memory (rather than regular RAM). As a result, some jobs may be deemed pointless and should be avoided.

To avoid this poor optimization, these writes must be designated as volatile. Indicates to the compiler that the writer should not be optimized. Now that we can print a variety of types, including integers and floats, it is time to start using Rust's built-in write and print macros. Core::fmt:: must be used to help them. This trait's write str method, which has a different type but otherwise looks like our write string function, is the only one that is required.

The following section discusses how to deal with newlines and characters who no longer fit inside the line. To begin at the beginning, each letter must be shifted one line ahead (the top line is eliminated) (the top line is removed). To do this, we provide an implementation of the Writer:/ new line function in the src/VGA buffer file. However, trying to build one causes a slew of problems owing to the way statics are presently handled. Allow the sluggish static to commence!

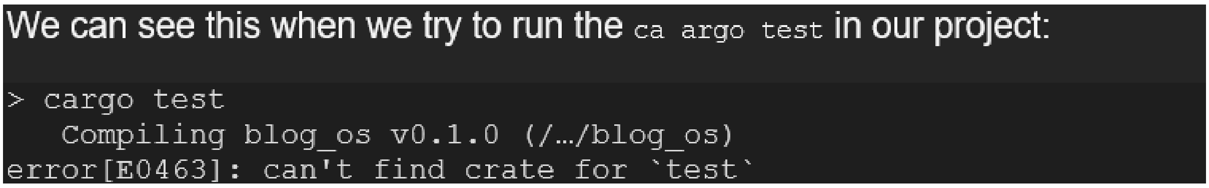
A macro is used to define a static that will not be computed during the build process. Instead, when initially used, the static progressively initializes itself, enabling for the construction of complicated code. Despite our best efforts, statics cannot utilize RefCell or UnsafeCell because they are not Sync, despite our efforts to connect an immutable static to a cell type that allows inner mutability. Now that we have a global writer, we can construct a macro that applies to the whole codebase. A macro is characterized by one or more rules, much like the standard library's "print! macro" function.

The format args macro invokes the io module's _print function. The $crate variable ensures that the macro operates outside of the std crate by extending to std when used in other crates. The application crate: buffer VGA:: Because the macros are imported into the crate's root namespace as a consequence, the println import command is ineffective.

Rust Testing

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

Unfortunately, no standardized programs like our kernel require a little more complexity. The issue is that the built-in test library, which depends on the standard library, is used implicitly by Rust's test framework. Because of this, we are unable to use the standard test framework with our [no std] kernel.



We can see this when we try to run the ca argo test in our project:

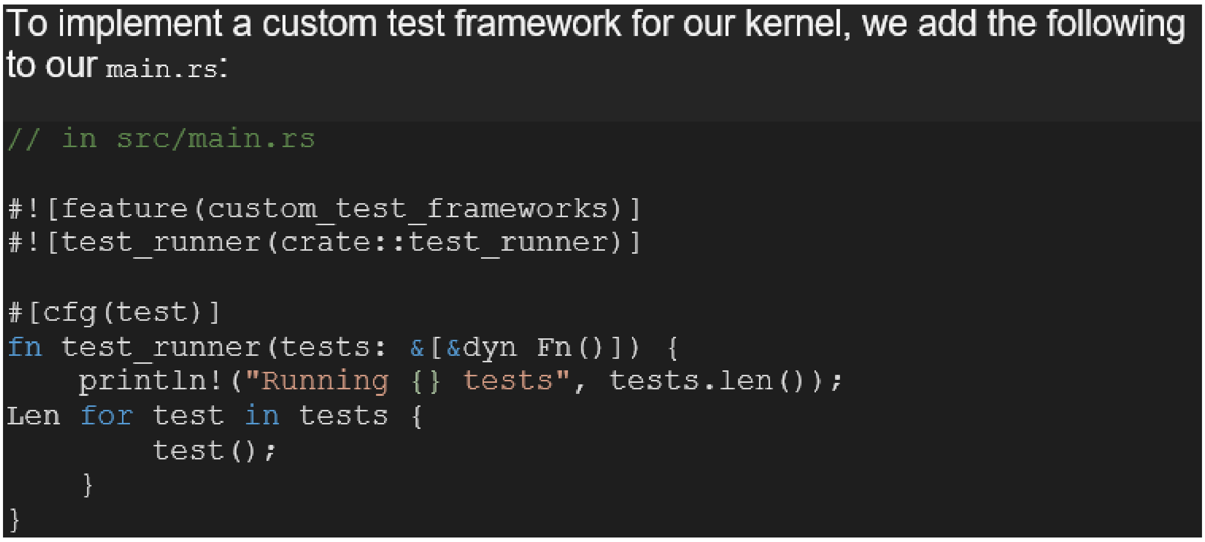
```
> cargo test
   Compiling blog_os v0.1.0 (/…/blog_os)
error[E0463]: can't find crate for `test`
```

The test crate is unavailable for our bare metal target because it depends on the standard library. It is possible to convert the test crate to a #[no std] context, but doing so is quite unstable and calls for certain hacks, such as redefining the panic macro.

Custom Test Frameworks

Fortunately, Rust has an unstable custom test framework capability, which enables replacing the built-in test framework. Because this functionality doesn't need any additional libraries, it is compatible with [no std] environments. It operates by assembling every function that has a #[test case] attribute and then calling a user-specified runner function with the list of tests as an argument. Consequently, it gives the implementation complete control over the testing procedure.

The drawback in comparison to the default test framework is the absence of many sophisticated capabilities, including panic tests. Instead, if such functionalities are required, it is up to the implementation to do so. This is perfect for us because we operate in a very unique environment where the default implementations of such sophisticated features are likely to fail. For instance, we disabled stack unwinding for our kernel to prevent panics from being caught by the #[should panic] property.



To implement a custom test framework for our kernel, we add the following to our `main.rs`:

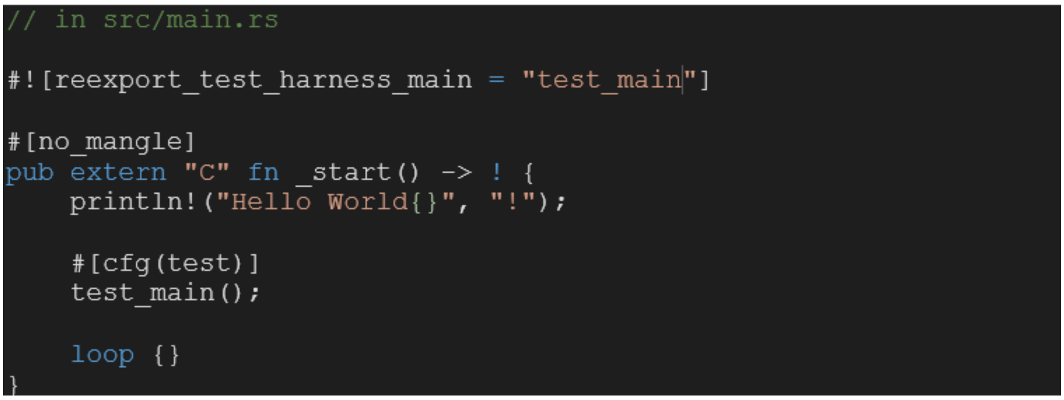
```
// in src/main.rs

#![feature(custom_test_frameworks)]
#![test_runner(crate::test_runner)]

#[cfg(test)]
fn test_runner(tests: &[&dyn Fn()]) {
    println!("Running {} tests", tests.len());
Len for test in tests {
        test();
    }
}
```

Our runner just calls each test function on the list after printing a brief debug message. The parameter type is a portion of the trait's object references. In essence, it is a list of kinds of references that can be called similarly to functions. We utilize the attribute to only include the function for test runs because it is worthless for non-test runs.

Now that we have updated the cargo test, we can see that it now succeeds or fails as shown in the notice below.

However, the message from our test runner is still displayed instead of our "Hello World" message. The cause is that the entry point is still our start f action. Because we use the attribute and offer our entry, the main function that the custom test frameworks feature produces and calls the test runner is ignored.

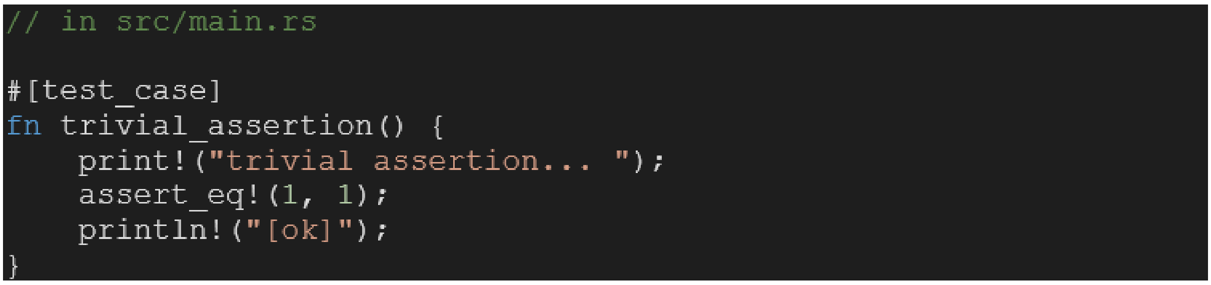
To correct this, we must first use the reexport test harness main attribute to rename the produced function to anything other than main. The renamed function can then be called from our _start function.



```
// in src/main.rs

#![reexport_test_harness_main = "test_main"]

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

    #[cfg(test)]
    test_main();

    loop {}
}
```

We call test main, the entrance function for the test framework, from our _start entry point. Since the function is not generated on a regularly add the call to test main in test contexts by using conditional compilation.

When we run a cargo test, our test runner displays the message "Running 0 tests" on the screen. Now that we are prepared, let's write the first test function.



```rust
// in src/main.rs

#[test_case]
fn trivial_assertion() {
    print!("trivial assertion... ");
    assert_eq!(1, 1);
    println!("[ok]");
}
```

A reference to the trivial assertion function is now present in the test slice that was supplied to our test runner method. We can see that the test was called and passed from the trite assertion...

[ok] output on the screen.

Our test runner returns to the test main method after running the tests, and the d test main then returns to our _start entry point function. Because the entry point function cannot return, we enter an unending loop at the end of _start. This is a concern because we want the cargo test to end when all tests have been completed

CPU Exceptions

CPU exceptions occur in various erroneous situations, for example, memory address or when dividing by zero. To react to them, we have to set up an interrupt descriptor table that provides handler functions. Finally, y, the kernel will be able to catch the breakpoint and resume execution afterward.

20 different forms of CPU exceptions.

- Page fault
  A page fault happens when memory is accessed illegally.
- Invalid opcode:
  When the current instruction is incorrect, this exception occurs.
- General protection fault
  The anomaly with the broadest range of causes is this one. It occurs, among other access violations, when a user tries to execute a privileged instruction in user-level code or writes reserved fields into configuration registers.
- Double fault
  The CPU attempts to call the appropriate handler code whenever an exception occurs. The CPU raises a double fault exception if another exception happens while calling the exception handler. When no handler function has been registered for an exception, this exception also occurs.
- Triple fault
  When attempting to call the double fault handler code, if a mistake occurs, the CPU issues a fatal triple fault. We are unable to handle or detect a triple fault. Most CPUs re-start both their operating systems and they in response

Double flutes

If the CPU is unable to launch an exception handler, a double fault is a unique exception that happens. 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++ and catches tch(Exception e) in Java or C#.

A double fault acts like any other exception would. It can define a typical handler function for it in the IDT because it has the vector number 8 as well. 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.

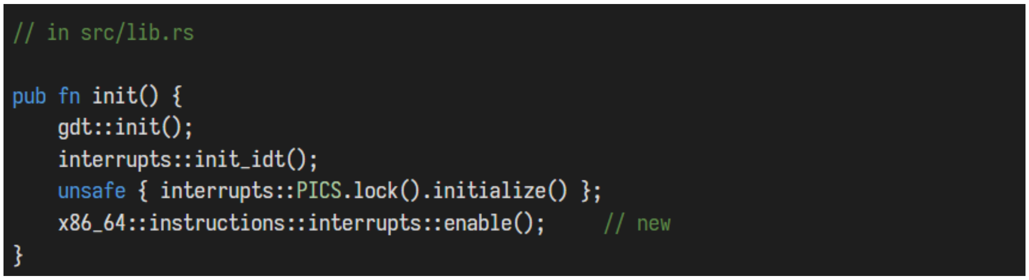
We can give a handler method that is identical to our breakpoint handle because a double fault is a typical exception with an error code. A guard page is a unique memory page located at the base of a stack that allows for the detection of stack overflows. A stack overflow results in a page fault because the bootloader creates a guard page for our kernel stack. The CPU attempts to put the interrupt stack frame in the stack when a page fault occurs by searching for the page fault handler in IDT. system reboot result from a third-page fault that happens because the stack pointer is still pointing to the guard page.

Three segments were utilized for segmentation on earlier architectures: the task state segment (TSS), the enclosed interrupt stack table (IST), and the global descriptor table (GDT). Add an integration test to test the new get module and verify that the double fault handler is called appropriately on a stack overflow to ensure that the double fault handler is called, it is intended to cause a double fault in the test method.

Hardware Interrupts

Interrupts provide a way to notify the CPU (Central Processing Unit) of attached hardware devices. So instead of letting the kernel periodically check the keyboard for new characters, the kernel can be notified of each keystroke on that keyboard. This method is more efficient because the kernel needs to act to notify the CPU (Central Processing Unit) when something happens. The kernel is able to react instantly, so it provides a faster response time. Not all hardware devices connect directly to the CPU (Central Processing Unit). Instead, a separate interrupt controller collects interrupts from all devices and feeds them to the Central Processing Unit. This means that many interrupt controllers can be programmed to support different priority levels for interrupts. Hardware interrupts to take place asynchronously, unlike exceptions. They can therefore happen at any time and are totally independent of the code that is being run. So, all of the potential concurrency-related issues suddenly appear in our kernel in the shape of a type of concurrency. Because it bans a changeable global state, Rust's rigid ownership model is helpful in this situation.

Enabling interrupts means nothing happens because interrupts are still disabled in the Central Processing Unit (CPU) configuration. The idea here is that the Central Processing Unit (CPU) does not listen to the interrupt controller at all. Therefore, no obstacle can reach the Central Processing Unit (CPU). Therefore, Interrupts should be enabled. After enabling interrupts there is a double error.



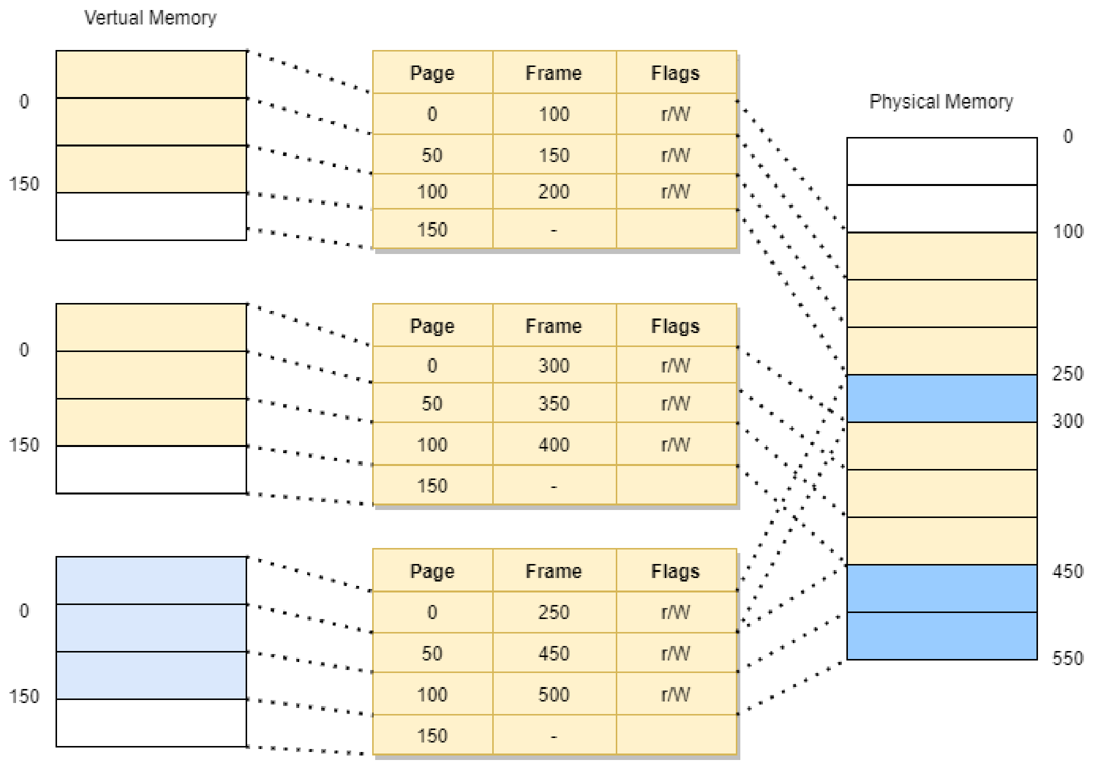
```
// in src/lib.rs

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

This binary error is caused by the hardware timer or the Intel 8253 being enabled. So as soon as we enable interrupts we start receiving timer interrupts. A timer interrupt handler should be implemented there. Also, we need to be careful to use the correct interrupt vector numbers at the end of the interrupt, otherwise our important, Central Processing Unit (CPU) An unsent interrupt may accidentally delete or break our system. This is why functions are unsafe.

Introduction to Paging

Paging divides the virtual memory in the binary storage into equal parts and allocates it properly for main memory management. Its idea is to divide the memory in the virtual space and physical memory space into small and fixed size parts and use them correctly to manage the main memory. Parts of virtual space are called pages and parts of physical space are called frames. And here the information related to the exchange of operations between main memory (physical memory) and virtual memory is maintained in the page table.

Millions of pages in virtual memory must be individually allocated, i.e., mapped, to each frame in physical space. Memory segmentation uses a segment selector register for each active memory region. Rrava cannot be paginated when there are more pages than frames. So, pagination uses a page table to store mapping information.



Page	Frame	Flags
0	100	r/W
50	150	r/W
100	200	r/W
150	-	

Page	Frame	Flags
0	300	r/W
50	350	r/W
100	400	r/W
150	-	

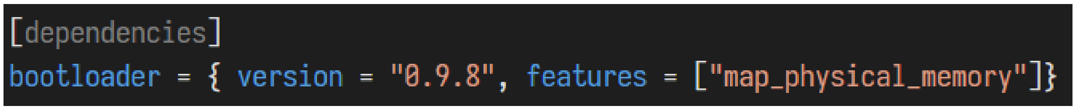
Page	Frame	Flags
0	250	r/W
50	450	r/W
100	500	r/W
150	-	

Therefore, during paging, the memory in the binary storage is divided into parts and used for main memory management through a page table. That is the process of applying the pages of the virtual space to the frames of the physical space.

Paging Implementation

A kernel runs on virtual addresses. Instead of modifying physical memory, pages of illegal memory accesses should be excluded. Since the kernel's page tables are stored in physical memory, the kernel runs on physical addresses, so the page tables cannot be accessed. The kernel has a different approach to accessing page table frames. To run an entry, support from the bootloader will be required. So a bootloader needs to be configured first. Then, a node must be created and implemented that traverses the page table hierarchy to translate virtual to physical addresses. Finally, how to create new maps in page tables and find unused memory frames to create new page tables.

This means that this requires the support of the bootloader, which creates the page tables that the kernel runs on. Since the bootloader has access to the page table, it can be used for any mapping we need. Executing this bootloader gives access to two options. map_physical_memory maps the entire physical memory into the physical address space. Thus kernel has access to all physical memory and map is followed by full physical memory access. The second entry is, Recursive_page_table. This recursively maps an access to the bootloader level 4 page tables. This allows the kernel to access the page tables as they are stored in the recursive page table section.



```
[dependencies]
bootloader = { version = "0.9.8", features = ["map_physical_memory"]}
```

map_physical_memory When this feature is enabled, the bootloader maps the entire physical memory to some somewhat unused virtual address range and to communicate the virtual address range to our kernel, the bootloader passes a boot information structure.

When physical memory is accessed, the page table code must be executed at the end. First, the kernel selects the currently active pages, creates a translation function that returns the physical address to which a given virtual address is mapped, and modifies the page tables to create a new mapping. Before doing that, a new memory module must be created.

Heap Allocation

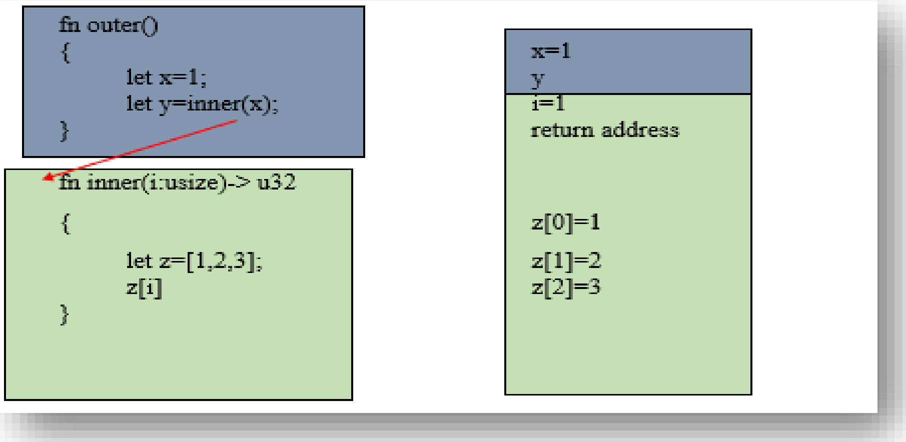
It starts off by providing a brief overview of dynamic memory and demonstrating how the borrow checker guards against typical allocation mistakes. A heap memory region is then created, along with an allocator crate, and the Rust language's fundamental allocation interface is implemented. Our kernel will have access to all of the built-in alloc crate's allocation and collection types by the end of this post.

Variables that are local and static

Static variables and local variables are both currently used by our kernel. Local variables are stored in the call stack, sometimes referred to as a stack data structure that records information about the currently running subroutines of a computer program. The execution stack, program stack, control stack, run-time stack, or machine stack are all terms used to describe this kind of stack in addition to "the stack". Although a call stack performs a variety of related tasks, its principal purpose is to keep track of the place to which each running subroutine should transfer control once it is finished. This function is only active while the function it is surrounded by is still active.

- **Local variables**

The call stack, a stack data structure that permits push and pop operations, serves as a storage location for local variables. The compiler pushes the caller function's local variables, return address, and parameters on each function entry:

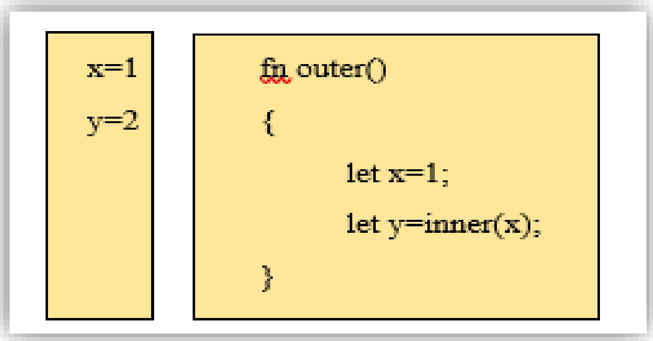


```
fn outer()
{
        let x=1;
        let y=inner(x);
}
```

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

In the example above, the call stack that followed the outer function calling the inner function is visible. The call stack makes the outer first local variables accessible. The inner call was pushed with the parameter 1 and the function's return address. As soon as inner gained control, it pushed its local variables.

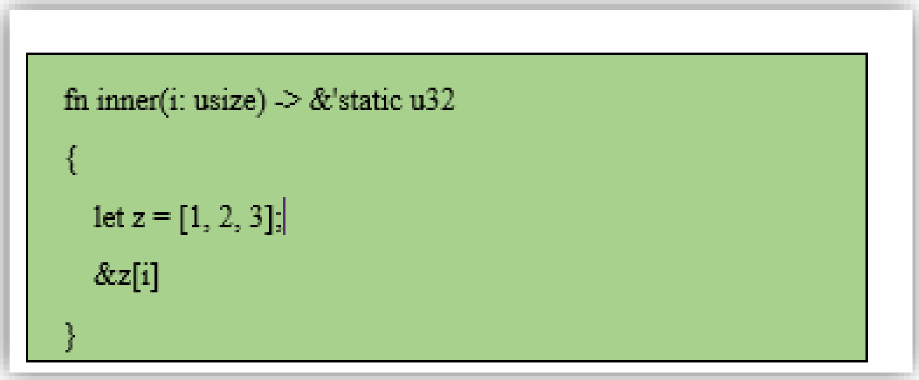
Only the local variables of outer are left after the return of the inner function, which causes its portion of the call stack to pop once more:



```
x=1
y=2
```

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

when the inner local variables are only valid while the function is running. The Rust compiler enforces particular lifetimes and produces an error when a value is utilized for too long, for as when attempting to return a reference to a local variable:

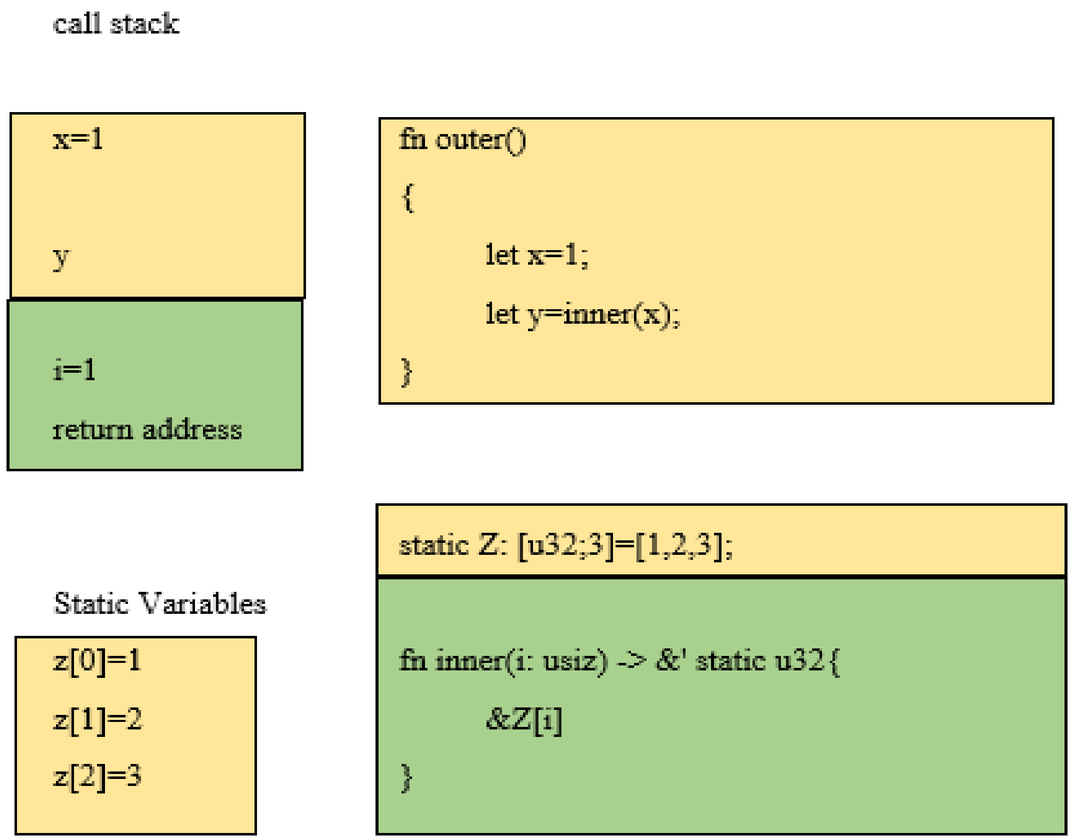


```
fn inner(i: usize) -> &'static u32
{
   let z = [1, 2, 3];
   &z[i]
}
```

Even though returning a reference in this case wouldn't make much sense, there are some cases when we want a variable to outlive the method.

- **Static Variables**

Static variables are kept in a dedicated area of memory that is not part of the stack. The linker assigns this memory address at compilation time, and the executable is written using this information. Statics have a "static lifespan" and may always be referenced from local variables since they last the whole duration of the program:



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

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

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

In the example before, when the inner function returns, the call stack is removed. Due to the fact that the static variables are kept in a distinct memory area that is never erased after the return, the reference to &Z[1] is still valid.

In addition, the position of static variables is known at the time of compilation, which eliminates the requirement for a reference when referencing them. They also offer a "static writer" in addition to this. We used that attribute for our println macro: Utilizing a static Writer internally in exception handlers avoids the need for a &mut Writer reference and makes the macro much simpler to utilize when we don't have access to any additional variables.

Static variables have the important drawback of being read-only by default as a result of this characteristic. Because a data race may happen if, for instance, two threads updated a static variable simultaneously, Rust enforces this rule.

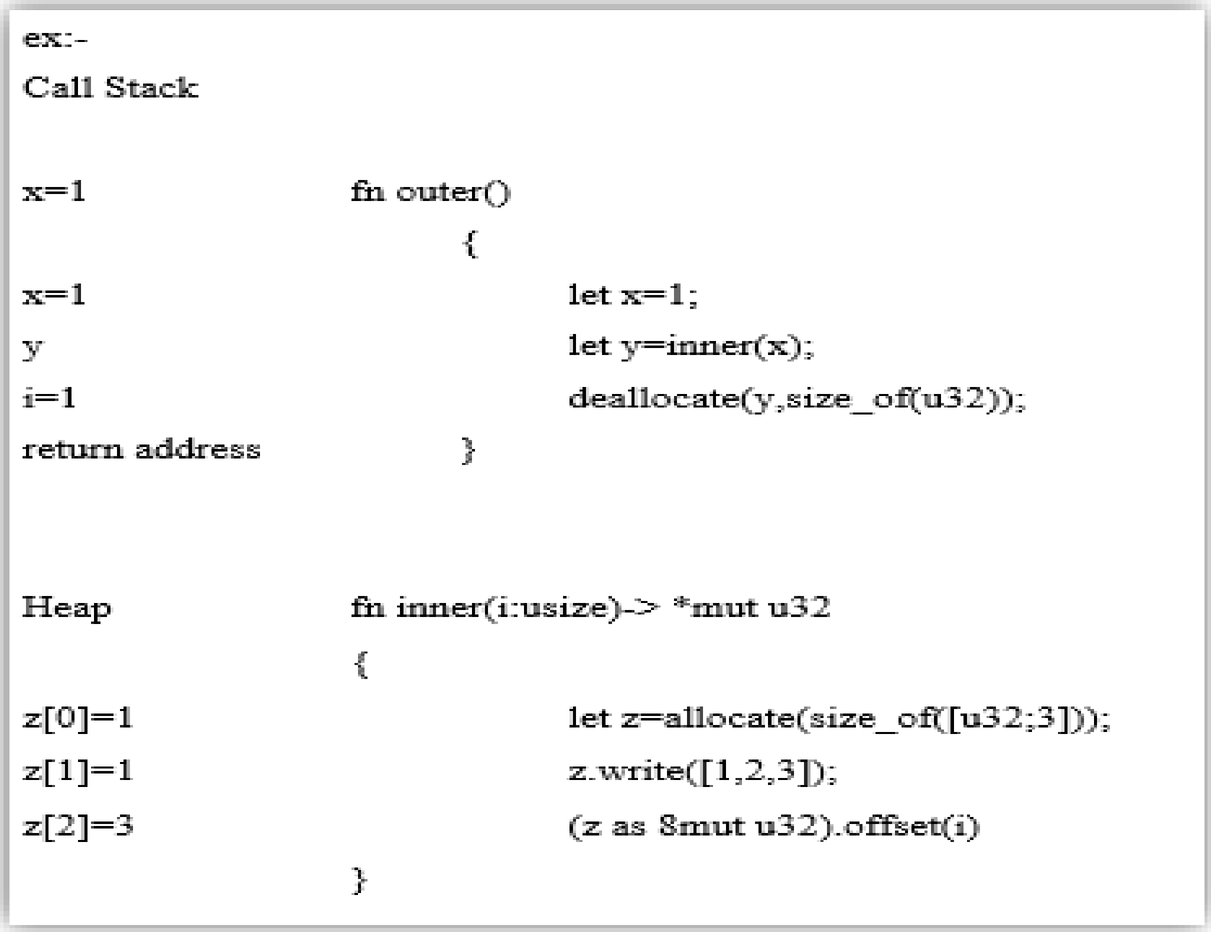
Static variables may only be changed by enclosing them in a Mutex. A static variable may only be changed by enclosing it in a Mutex type, which guarantees that only one &mut reference exists at all times. For our static VGA buffer Writer, we have already utilized a Mutex.

Dynamic Memory

The majority of use cases are already made possible by the combined power of local and static variables. We did see that each had certain drawbacks, though. Local variables are only valid for the duration of the function or block they are in. This is the case because they exist on the call stack and are eliminated when the function they were residing in returns.There is no method to recover and reuse the memory used by static variables when they are no longer required because they always exist for the whole duration of the program. Additionally, they are available from all functions and have ambiguous ownership semantics, so when we wish to edit them, we must secure them using a Mutex(https://docs.rs/spin/0.5.2/spin/struct.Mutex.html).

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

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



```
ex:-
Call Stack

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


Heap                fn inner(i:usize)-> *mut u32
                    {
z[0]=1                          let z=allocate(size_of([u32;3]));
z[1]=1                          z.write([1,2,3]);
z[2]=3                          (z as 8mut u32).offset(i)
                    }
```

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

Common Errors

Other than memory leaks, which are unfortunate but do not expose the software to attackers, there are two common bug categories with more catastrophic consequences:

When we accidentally continue to use a variable after performing deallocate on it, we have a vulnerability known as use-after-free. This kind of flaw causes undefinable behavior and is commonly utilized by attackers to execute arbitrary code.

We have a double-free vulnerability when we unintentionally free a variable twice. This is a concern since it may release a separate allocation that had been placed there following the initial deallocate request. As a result, it can result in another use-after-free vulnerability.

Given the widespread knowledge of these vulnerabilities, one could assume that individuals have already learnt how to prevent them. However, these flaws are still often discovered. For instance, in 2019, the Linux use-after-free flaw allowed for arbitrary code execution. Online searches using the term "current year" and the use-after-free linux operating system will nearly always return results. This illustrates how difficult it may be for even the best programmers to control dynamic memory in complex projects.

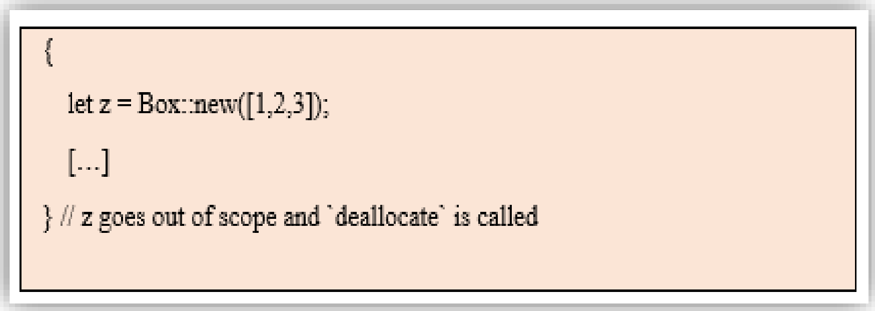
Many programming languages, including Java and Python, automatically manage dynamic memory using a method known as garbage collection. Garbage collection (GC), a sort of automated memory management, is used in computer science. The garbage collector tries to reclaim memory that the program allocated but is no longer referenced; this memory is referred to as trash).The concept is that deallocate is never manually called by the programmer. Instead, unwanted heap variables are periodically found and automatically deallocated while the application is halted. As a result, the aforementioned vulnerabilities cannot exist. The normal scan's performance overhead and the sometimes lengthy rest intervals are the disadvantages.

Rust approaches the issue differently. It makes use of a notion called ownership to ensure the accuracy of dynamic memory operations at build time. As a result, there is no performance overhead and the aforementioned vulnerabilities may be avoided without the need of trash collection. Another advantage of this approach is that, similar to C or C++, the programmer may keep precise control over how dynamic memory is used.

Allocations in Rust

The Rust standard library offers abstraction types that implicitly call allocation and deallocate rather than allowing the programmer to individually invoke these operations.Box, an abstraction for a heap-allocated item, is the most crucial type.

The Box type implements the Drop trait to use deallocate after it departs scope in order to free up heap memory once more: It provides a constructor method called Box::new that receives a value, calls allocate with the size of the value, and then moves the value to the heap's newly formed slot.



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

Use Cases

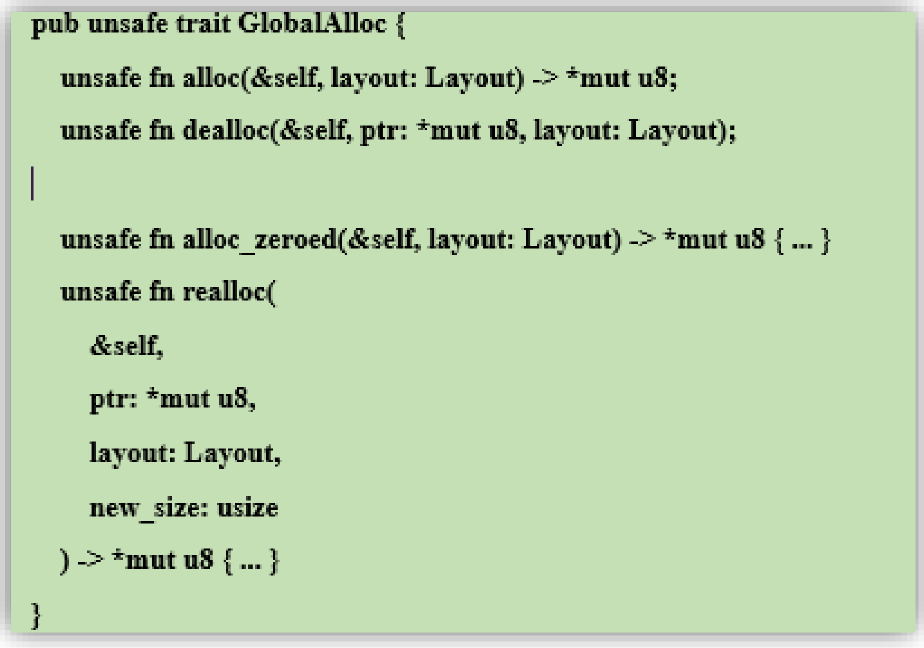
In general, local variables are preferred, especially in performance-critical kernel code. Dynamic memory allocation, however, may be the best option in some circumstances. Since we must locate a vacant spot on the heap for each allocation, dynamic memory allocation always entails some performance cost.

Variables with a dynamic lifespan or a changeable size require dynamic memory. Rc, Vec, and other collection types are examples of those that grow dynamically when new members are added. These types allocate more memory as they fill up, copy all of the items over, and then deallocate the previous allocation.

The GlobalAlloc Trait

It specifies the features a heap allocator must offer. The characteristic is unique since the programmer seldom ever directly uses it. Instead, when utilizing the allocation and collection types of alloc, the compiler will automatically add the necessary calls to the trait functions.

It is important to examine the declaration of the trait as we must implement it for each of our allocator types:



```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
|
    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 { ... }
    unsafe fn realloc(
        &self,
        ptr: *mut u8,
        layout: Layout,
        new_size: usize
    ) -> *mut u8 { ... }
}
```

It defines the two necessary methods alloc and dealloc, which are the same as the allocation and deallocate procedures we used in our examples:

A Layout object that specifies the ideal size and alignment for the memory allocation is sent as an input to the alloc function.It gives back a raw pointer to the memory block's initial byte.The alloc function delivers a null pointer to indicate an allocation error rather than an explicit error value.The opposite, dealloc, is in charge of liberating a memory block once more.The pointer that alloc returned and the layout that was utilized for the allocation are the two arguments that are given to it.

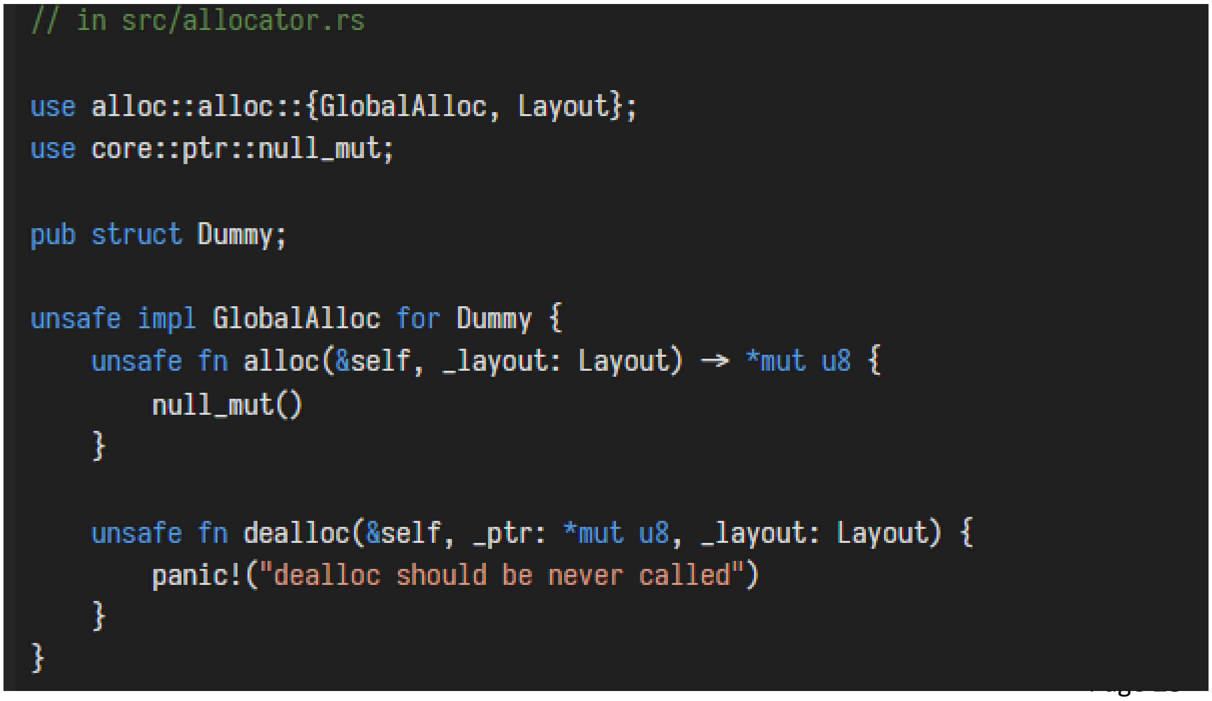
A Dummy Allocator

new Dummy allocator module creation:

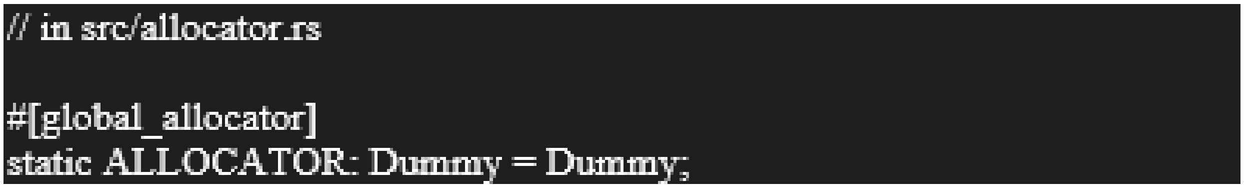
// in src/lib.rs

pub mod allocator;

The struct is created as a zero-sized type as it has no fields. We always return the null pointer from alloc, which is equivalent to an allocation error, as was previously noted. There should never be a call to dealloc since the allocator never returns any memory. We panic using the dealloc approach as a result. We don't need to supply implementations for the alloc zeroed and realloc functions because they already have default implementations.



```rust
// in src/allocator.rs

use alloc::alloc::{GlobalAlloc, Layout};
use core::ptr::null_mut;

pub struct Dummy;

unsafe impl GlobalAlloc for Dummy {
    unsafe fn alloc(&self, _layout: Layout) -> *mut u8 {
        null_mut()
    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        panic!("dealloc should be never called")
    }
}
```



```
// in src/allocator.rs

#[global_allocator]
static ALLOCATOR: Dummy = Dummy;
```

The Dummy allocator is a zero-sized type, thus there is no need to add any fields in the initialization line.

Allocator Designs

It describes the steps involved in creating heap allocators from scratch. There are several allocator concepts that are demonstrated and discussed, including bump allocation, linked list allocation, and fixed-size block allocation.

We will show how to create our own heap allocator from scratch rather than relying on an allocator crate that already exists. We'll examine several allocator concepts, such as a basic bump allocator and a straightforward fixed-size block allocator, in order to develop an allocator that performs better than the linked list allocator crate.

Desiging goals

The management of the heap memory that is accessible is the duty of an allocator. In order to allow for future reuse, it must track memory released by dealloc and return any unused space upon an alloc call. Most importantly, it must never distribute memory that is already in use elsewhere since doing so would lead to unexpected behavior.

In addition to accuracy, there are other more design objectives. For instance, the allocator should minimize fragmentation and efficiently use the memory that is currently available. It should also be scalable to any processor count and capable of handling several tasks at once. Even better performance might be achieved by optimizing the memory layout in respect to CPU caches to improve cache locality and avoid false sharing.

Bump Allocator

It simply maintains track of the total amount 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 bump allocator's goal is to linearly allocate memory by growing (or "bumping") a next variable that corresponds to the beginning of the free memory. Since the next pointer only advances in one direction, it never distributes the same memory region more than once. When the heap is full, no further memory may be allocated; as a result, the subsequent allocation encounters an out-of-memory error.

An allocation counter that increases by 1 on each call to alloc and decreases by 1 on each call to dealloc is frequently used to create a bump allocator. Once the allocation counter reaches 0, the heap has been completely deallocated. When this occurs, it is possible to reset the next pointer to the heap's start address, reopening the whole heap's memory for allocations.

Implementation

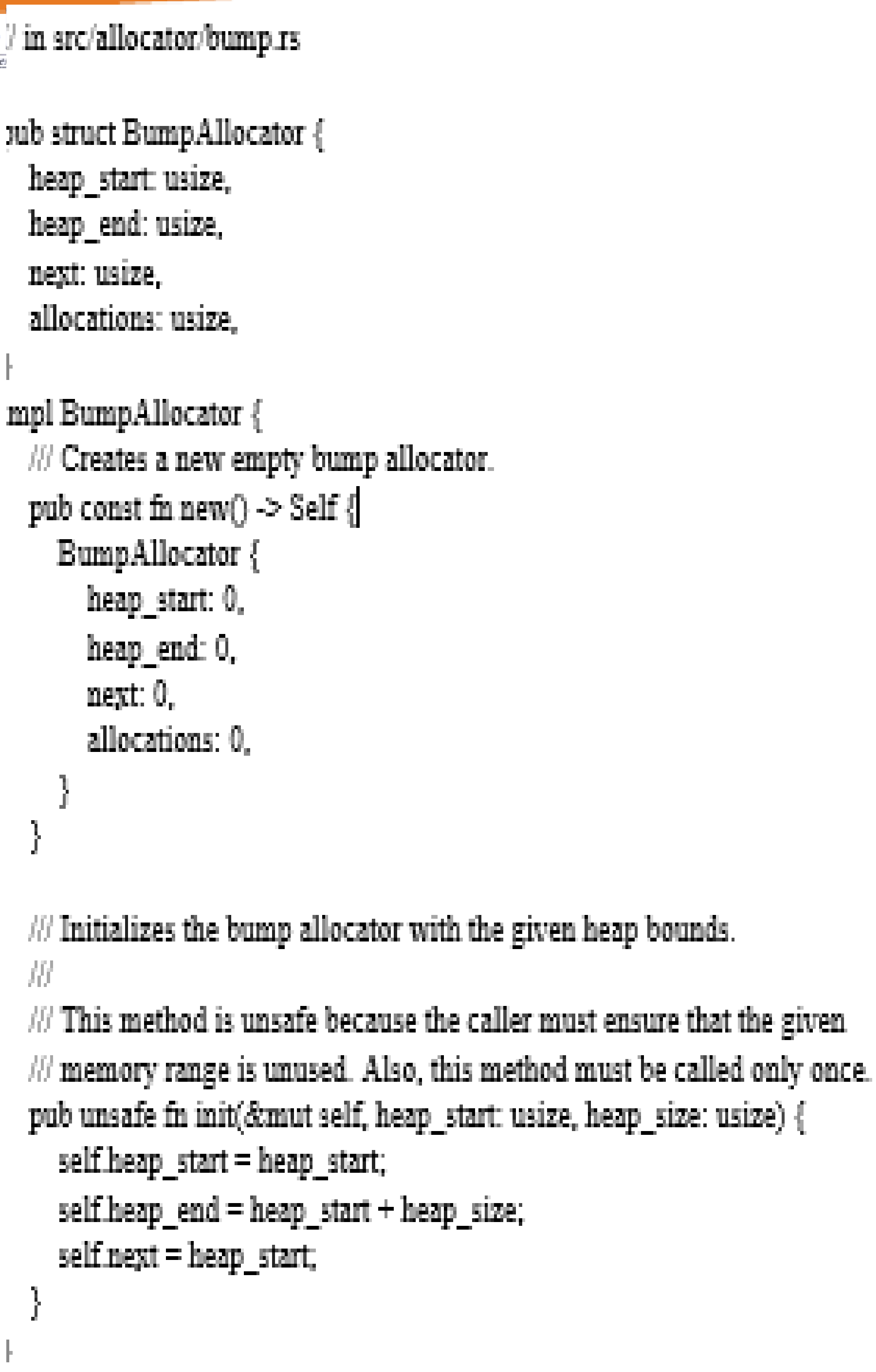
The bottom and upper limits of the heap memory area are tracked by the values heap start and heap end. If these addresses are not correct, the allocator will return RAM that is not what was requested. This necessitates making calling the init method dangerous.

The purpose of the next field is to always point to the first byte of the heap that is not in use or the start address of the following allocation. The heap's start setting in the init method is heap start since it is initially empty. This field will "bump" up with each allocation, preventing the return of the same memory space.

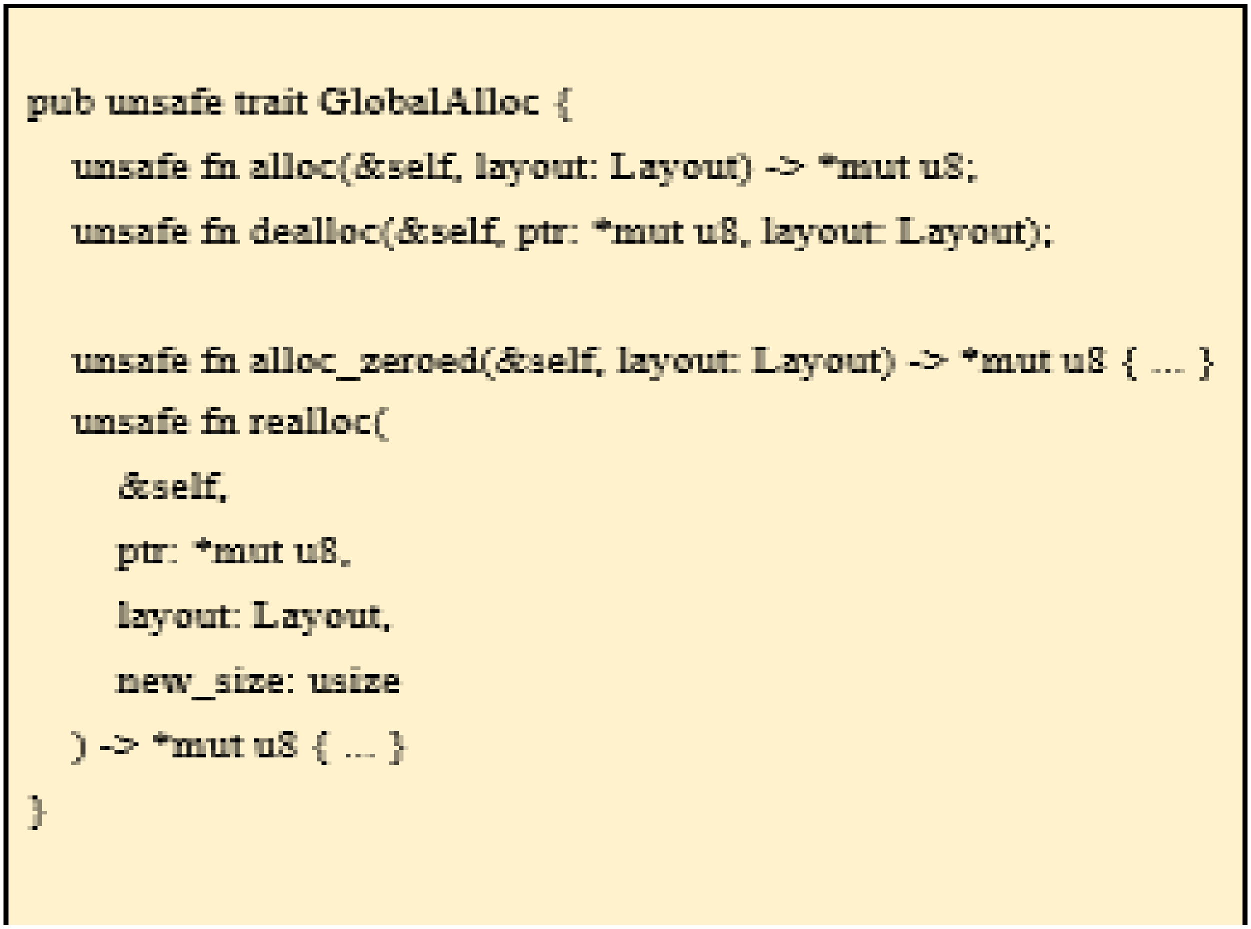
When the final allocation has been released, the allocations field's function is to reset the allocator. It's a simple counter for the current allocations. It begins at zero.

To maintain the interface's similarity to the allocator given by the linked list allocator crate, we decided to develop a separate in it function rather than doing the initialization directly in new. In this manner, altering the allocators is possible without further adjusting the code.

The bump allocator's content contains the following information:



```
/ in src/allocator/bump.rs

pub struct BumpAllocator {
    heap_start: usize,
    heap_end: usize,
    next: usize,
    allocations: usize,
}

impl BumpAllocator {
    /// Creates a new empty bump allocator.
    pub const fn new() -> Self {
        BumpAllocator {
            heap_start: 0,
            heap_end: 0,
            next: 0,
            allocations: 0,
        }
    }

    /// Initializes the bump allocator with the given heap bounds.
    ///
    /// This method is unsafe because the caller must ensure that the given
    /// memory range is unused. Also, this method must be called only once.
    pub unsafe fn init(&mut self, heap_start: usize, heap_size: usize) {
        self.heap_start = heap_start;
        self.heap_end = heap_start + heap_size;
        self.next = heap_start;
    }
}
```



```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);

    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 { ... }
    unsafe fn realloc(
        &self,
        ptr: *mut u8,
        layout: Layout,
        new_size: usize
    ) -> *mut u8 { ... }
}
```

The GlobalAlloc trait, which has the following definition, must be implemented by all heap allocators.

the rationale behind using &self arguments in the GlobalAlloc trait methods definitions By adding the #[global allocator] attribute to a static that implements the GlobalAlloc trait, the global heap allocator is defined. A method that takes &mut self on the static allocator cannot be called because static variables are immutable in Rust. Due to this, just an immutable &self reference is required by all of GlobalAlloc's methods.

Thankfully, a &self reference may be converted into a &mut self reference. A spin::Mutex spinlock can be used to surround the allocator and offer synchronized internal mutability. This type has a lock function which guarantees mutual exclusion is order to correctly transform a &self reference to a &mut self reference.

A Locked Wrapper Type

With the help of the spin::Mutex wrapper type, we are able to implement the GlobalAlloc trait for our bump allocator. Instead of using the BumpAllocator directly, the trait should be implemented for the wrapped spin::MutexBumpAllocator> type:

unsafe impl GlobalAlloc for spin::Mutex<BumpAllocator> {…}

Implementation for Locked<BumpAllocator>

We can use the Locked type to generate GlobalAlloc for our bump allocator as it is provided in our own crate, unlike spin::Mutex. The overall implementation appears to be as follows

```
// in src/allocator/bump.rs

use super::{align_up, Locked};
use alloc::alloc::{GlobalAlloc, Layout};
use core::ptr;

unsafe impl GlobalAlloc for Locked<BumpAllocator> {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        let mut bump = self.lock(); // get a mutable reference

        let alloc_start = align_up(bump.next, layout.align());
        let alloc_end = match alloc_start.checked_add(layout.size()) {
            Some(end) => end,
            None => return ptr::null_mut(),
        };

        if alloc_end > bump.heap_end {
            ptr::null_mut() // out of memory
        } else {
            bump.next = alloc_end;
            bump.allocations += 1;
            alloc_start as *mut u8
        }
    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        let mut bump = self.lock(); // get a mutable reference

        bump.allocations -= 1;
        if bump.allocations == 0 {
            bump.next = bump.heap_start;
        }
    }
}
```

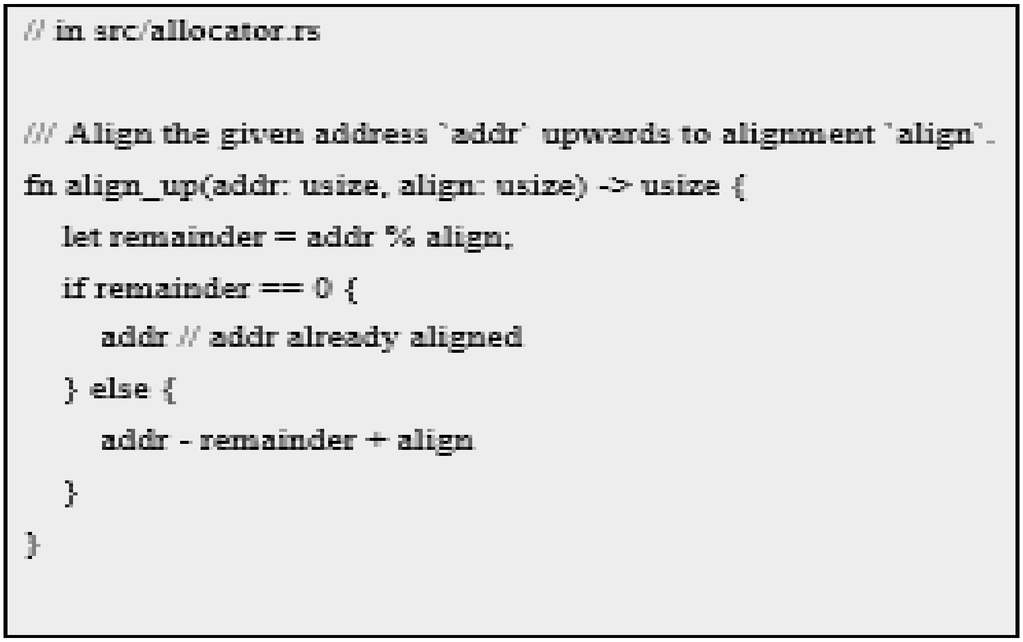
In both the alloc and dealloc stages, the Mutex::lock method must be invoked through the inner field in order to obtain a mutable reference to the wrapped allocator type. In multithreaded situations, no data race may happen since the instance is locked until the method's conclusion.

As contrast to the preceding prototype, the alloc implementation follows alignment criteria and performs a bounds check to ensure that allocations stay within the heap memory region. The following address is first rounded up to the alignment determined by the Layout parameter. In a minute, the align up function's source code will be shown. The end address of the allocation is then obtained by adding the required allocation size to alloc start. With high allocations, we employ the checked add function to prevent integer overflow. If there is an overflow or if the end address of the allocation as a result is bigger than the end address of the heap, we return a null pointer to signify that there is not enough memory. If not, we update the subsequent address and, as before, add one to the allocations counter. The alloc start address is then transformed into a *mut u8 pointer before being returned.

The provided pointer and Layout parameters are disregarded by the dealloc function. Instead, it only brings down the counter for allocations. When the number hits zero once more, all allocations have been released once more. In this case, it changes the following address back to the heap start address in order to regain access to the whole heap memory.

Address positioning

We may include the align up function in the parent allocator module since it is sufficiently broad. Typical implementation seems like follows:



```
// in src/allocator.rs

/// Align the given address `addr` upwards to alignment `align`.
fn align_up(addr: usize, align: usize) -> usize {
    let remainder = addr % align;
    if remainder == 0 {
        addr // addr already aligned
    } else {
        addr - remainder + align
    }
}
```

Using it

To use the bump allocator instead of the linked list allocator crate, the ALLOCATOR static in allocator must be changed. rs:

// in src/allocator.rs

use bump::BumpAllocator;

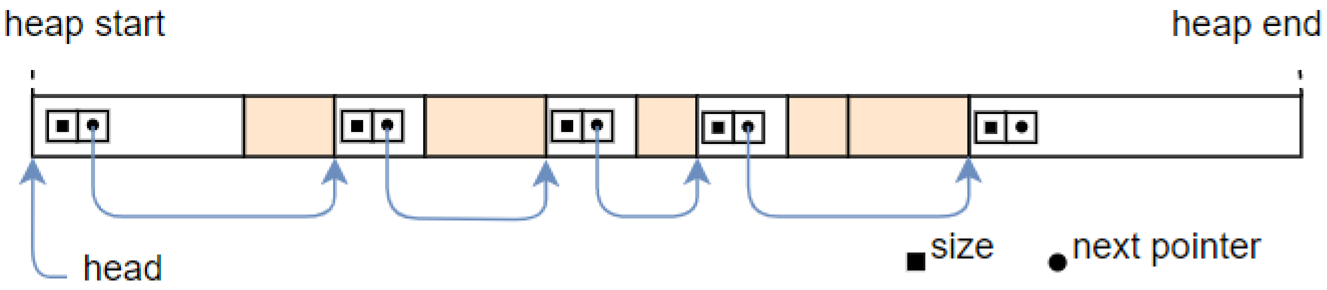
#[global_allocator]

static ALLOCATOR: Locked<BumpAllocator> = Locked::new(BumpAllocator::new());

Linked List Allocator

When developing allocators, it's a standard practice to use the free memory regions as backup store to keep track of any number of available spaces. This takes use of the fact that the regions are still supported by a physical frame and mapped to a virtual address, but the stored data is no longer required. If the data about the freed zone is stored in the region itself, we can keep track of an endless number of liberated areas without utilizing additional memory.

The most common approach to implement it is to make a single linked list in the space freed up, where each node represents a space in the memory that has been released:

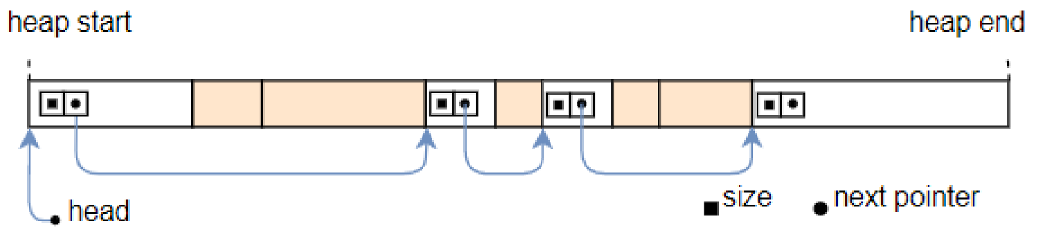
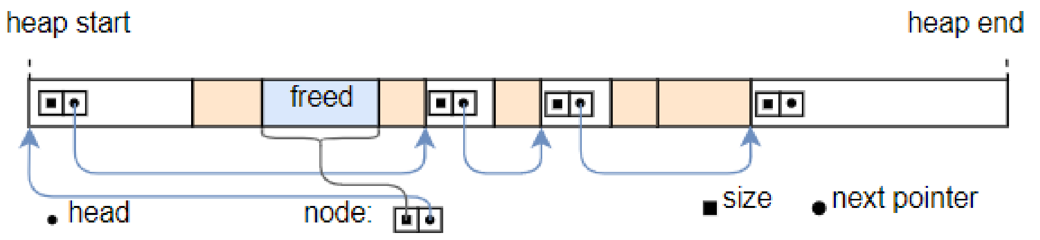
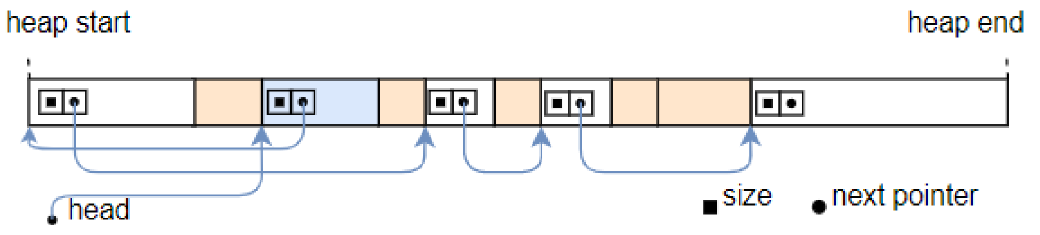


Each list node contains two fields: a reference to the next unused memory area and the size of the memory region. With this approach, we can keep track of all unused regions, regardless of their number, by just keeping track of the first one (referred to as the head). A free list is a common name for the ultimate data structure.

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

Add Free Region Technique

The add free region function on the linked list provides the required push action. In addition to being the primary method in our version of dealloc, this function is currently only utilized in init. Remember that the dealloc process is called each time a memory area is allocated and freed again. To enable tracking, we wish to move this portion of the freed memory to the linked list.

Step	Operations
0	
1	 heap start heap end freed head node: size next pointer
2	

The process puts the argument at the front of the list that provides the location and dimensions of a memory space. The region's size and alignment are initially checked to see if they are suitable for storing a ListNode. The following steps are used to construct the node and add it to the list:

The heap's condition is displayed in Step 0 prior to the call to add free region. Step 1 of the technique is when the memory region shown in the image as freed is called. Following the preliminary tests, the procedure adds a new node with the size of the released region to its stack. The head pointer is then set to None by setting the node's next pointer to the current head pointer using the Option::take function.

Step 2 uses the write method to place the newly created node at the beginning of the memory area that has been freed up. Then the head pointer is pointed at the new node. A little disarray can be seen in the resulting pointer structure since the released area is always placed at the front of the list, but if we follow the pointers, we can see that every free region is still accessible from the head pointer.

Variations

There are many different variations for the fixed-size block allocator idea as well. In well-known kernels like Linux, the slab allocator and the buddy allocator are two famous instances. In the sections that follow, we give a quick summary of these two designs.

- ### Slab Allocator

  A slab allocator works by using block sizes that exactly match certain kernel type combinations. In this manner, no memory is wasted and allocations of such types perfectly match a block size. To further boost efficiency, it may occasionally be possible to preinitialize type instances in free blocks.

  Combining slab allocation with different allocators is common. To further divide a block that has already been formed in order to conserve memory, for example, it can be used in combination with a fixed-size block allocator. On top of a single massive allocation, an object pool pattern is frequently implemented using this technique.

- ### Buddy Allocator

  The buddy allocator concept employs a binary tree data structure together with power-of-2 block sizes in place of a linked list to handle freed blocks. It divides a larger-sized block into two halves when a new block of a specific size is needed, resulting in the creation of two child nodes in the tree. When a block is released once more, the tree's neighbor block is examined. The two blocks are rejoined to create a block that is double the size if the neighbor is likewise free.

This merging procedure has the benefit of reducing external fragmentation, allowing for the reuse of tiny freed blocks for big allocations. Additionally, since no backup allocator is employed, performance is more predictable. The only conceivable block sizes are power-of-2, which has the largest problem of potentially wasting a lot of RAM owing to internal fragmentation. To further divide an allocated block into several smaller blocks, buddy allocators are frequently used in conjunction with slab allocators.

Async/Await

The Rust language's async/await concept provides world-class aid for cooperative multitasking. It is best to provide a little example to illustrate the concept of the future. A future represents a value that is not now available. This might be a file retrieved from the network or a number computed by another task. The poll function requires both selves: Pin&mut Self> and cx: &mut Context as inputs.

Despite being anchored to its memory location, the Self-value behaves similarly to a standard self-reference. The final parameter's objective is to pass a Waker object to the task. The task may use this waker to notify when it (or a part of it) is finished, such as when the file has been loaded from the disk. Future combinators, similar to the Iterator trait's methods, are techniques like a map that allow chaining and combining futures. Instead of waiting for the future, these combinators produce a future that uses the mapping operation to poll.

The future combinators' development of the Rust-type system allows developers to write programs that are both fast and efficient. They give high-level combinator functions such as a map and then, which may be used to play with the result using various closures. Because they are implemented as regular structs, the Rust compiler may over-optimize them. The Async/await pattern, which has been added support in Rust, allows programmers to write synchronous code that the Rust compiler translates to asynchronous code. We may use the await operator to access the value of a future without needing closures or either kind.

Its operation is based on the concepts of async and await. The compiler turns the function body into a state machine in the background with each. await call representing a separate state. The state machine implements the Future trait by making every Poll call a possible state transition for the aforementioned example function. It can automatically generate structs with the same variables needed since it knows when each variable is used. The structs that represent the different states and include the required variables are already defined.

The "Waiting on.foo.txt" step contains the content variable for the upcoming string concatenation that will occur when bar.txt is produced. No variables are preserved in the "end" state since the function has already completed its execution. The code generated by the compiler is just presented as an example here. To keep things simple, we don't deal with pinning, ownership, longevity, and so on.

Even if it may work differently, the compiler-generated code handles everything flawlessly. The poll function is accomplished by using a match statement on the current state inside a loop. We begin by executing the poll function of the foo txt future. If it is not ready, the loop is terminated, and Poll::Pendering is returned. On the next poll call, the state machine will try to poll the bar txt-future, which will enter the same match arm.

The bar txt future is polled first. If content.len() returns a number less than the min len value preserved in the state struct; the bar.txt file is read asynchronously. When the future is ready, we give the result to the content variable to continue processing the example function's logic. The.await action is then translated into a state change, this time to WaitingOnBarTxt. Furthermore, before returning the Poll's result, we change the current state to the End state: Signal for ready.

The state machine transformation stores the local variables at each pause point in a struct. Consider the function async fn pin example(), which generates a little array with the elements 1, 2, and 3 and stores it in an element variable. The converted number is then asynchronously written to the foo.txt file. The number referring to that element is then returned. Even if the last array member is now at memory address 0x1002c, when we move this struct from one memory address to another, the reference stays at 0x1001c.

Because the pointer was left dangling, the next poll call will behave erratically. Using memory allocation, we may try to create a self-referential struct. By storing pointers as offsets from the struct's beginning, we avoid the need to update pointers. Because the offset stays constant while the struct is moved, no field modifications are required. The main advantage of this method is that it may be used at the type system level without incurring additional runtime costs.

We create the SelfReferential struct, which is a simple object with just one pointer field. Before allocating this struct on the heap, we first allocate a null reference to it. The memory location of the heap-allocated struct is then determined and stored in a ptr variable. The struct becomes self-referential once the ptr variable is allocated to the self ptr field. The pinning API solves the &mut T conundrum by using the Unpin marker-trait and the Pin wrapper type.

The objective of these types is to gate any Pin methods that may be used to gain &mut references to the wrapped item. As a result, they may always depend on internal references. Both problems occur due to the Pin> type's lack of support for the DerefMut trait. This is exactly what we intended since updating a field does not cause the whole struct to move. To change the value using Pin::as mut, we must use the unsafe obtain unchecked mut method, which acts on a Pin&mut T> rather than a Pin>. This will enable us to restart the startup process.

We may halt this activity by selecting Pin> instead of Unpin. The Future::poll method pins using a Pin&mut Self> rather than the normal &mut self. This ensures that futures are not transferred from memory between poll calls. Because stack pinning is more challenging to perform effectively, I recommend always using Box for this process. The futures crate may be used to safely create a future combinator function that uses stack pinning.

An executor's role is to allow the spawning of futures as separate tasks, usually via some form of spawn mechanism. The capacity to conduct activities concurrently and keep the CPU busy is a significant advantage of handling all futures in one area. Because querying futures is time-consuming, Rust executors instead utilize the waker API. To balance the burden, they create a thread pool that, if there is enough work, may utilize all cores, and they use tactics such as work stealing. There are other executor implementations intended particularly for embedded devices that reduce memory overhead and latency.

Futures and async/await are two ways to implement the cooperative multitasking paradigm. Each future that is assigned to the executor is effectively a cooperative job. Futures restore control to the CPU core rather than completing an explicit yield instruction—Poll::Pending (or, in the end, Poll::Ready) (or Poll::Ready at the end).

Conclusion

Due to its accessibility and the rising need for the services it provides, it is clear why Rust's popularity is rising and won't be declining any time soon.

The Rust community is quite active, and new versions and advancements in Rust technology are frequently produced. Due to its capability and reputation for creating safe systems, Rust is anticipated to be popular in the next years.

Rust's security, effectiveness, and productivity will keep the development community interested in it (i.e., its capacity to assist developers in producing performant code quicker).

References

- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjNm8zHtpz7AhWayHMBHZOuAOYQFnoECAoQAQ&url=https%3A%2F%2Fwww.rust-lang.org%2F&usg=AOvVaw0Eo-jPMg1YR89Z4OceiP9Q
- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjNm8zHtpz7AhWayHMBHZOuAOYQFnoECEUQAQ&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FRust_(programming_language)&usg=AOvVaw1r2kaOrJGusz6PKrpinH2F
- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjLoNrptpz7AhU34nMBHdlXDeoQFnoECAkQAQ&url=https%3A%2F%2Fwww.tutorialspoint.com%2Frust%2Findex.htm&usg=AOvVaw0bO7LzmJ8Qe7AZXuXujv1T

Individual Contribution

Number	Registration Number	Name	Individual Contribution
1	DE ZOYSA A.S.	IT21167096	• Heap Allocation • Allocator Design • Async/Await • Conclusion
2	NILUPUL S.A.	IT21167478	• Hardware Interruptus • Introduction to Paging • Paging Implementation
3	MADURANGA D.B.W.N.	IT21170270	• Rust Testing • CPU Exceptions • Double Faults
4	BANDARA K.M.N.M	IT21163418	• Introduction • A Standalone Rust Binary • A Simple Rust Kernel • Text Mode in VGA