



Group Assignment

How a modern language like Rust can be used to implement more secure, reliable, high-performance operating system

IE2032 - Secure Operating System

Student Name	Registration Number
D.S.C. Wijesuriya	IT21155802
P.G.E Janith Sandamal	IT21166860
W.N. Dilsara	IT21182600
Premakanthan. N	IT21197550

Date of submission

November 07th, 2022

Table of Contents

List of figures.....	4
1. Introduction.....	6
2. A Freestanding Rust Binary.....	7
1.1 Introduction.....	7
1.1.1 The no_std Attribute	7
1.1.2 Defining our entrypoint.....	9
1.3 Building for bare-metal.....	10
1.4 Code	11
3. A Minimal Rust Kernel.....	12
2.1 Installing Rust Nightly	12
2.2 Target Specification	12
2.3 Assembling the Parts.....	15
4. VGA Text Mode	16
3.1 Different colors in this module.	17
3.1.1 Provide color code	17
3.1.2 Provide text buffer	18
3.1.3 Create Writer.....	18
3.1.4 Code in Action	20
5. Testing.....	21
4.1 Custom Test Frameworks	21
6. CPU Exceptions	24
7. Hardware Interrupts	26
8. Introduction to Paging.....	28
7.1 Hidden Fragmentation	28
7.2 Page Tables	29
7.3 Multilevel Page Tables	30
9. Paging Implementation	32
8.1 Accessing the Page Tables.....	32
8.2 Translating Addresses	35
8.3 Utilizing OffsetPageTable	38

8.4 Creating a new Mapping	41
8.4.1 A dummy FrameAllocator	43
8.4.2 Choosing a Virtual Page	44
8.4.3 Creating the Mapping	45
8.5 Allocating Frames	47
8.5.1 A usable_frames Method	48
8.5.2 Implementing the FrameAllocator Trait	50
8.5.3 Using the BootInfoFrameAllocator	50
10. Heap Allocation	52
9.1 Box	52
9.2 Rc/Arc	52
9.3 Vec	53
9.3.1 Vec growth	53
9.3.2 Short Vecs	54
9.3.3 Longer Vecs	54
9.4 String	54
11. Allocator Designs	57
10.1 Bump Allocator	57
10.2 Linked List Allocator	58
10.3 Fixed-Size Block Allocator	59
12. Async/Await	61
11.1 async Lifetimes	61
11.2 async move	62
11.3 .awaiting on a Multithreaded Executor	63
13. Conclusion	65
References	66

List of figures

Figure 1: no_std attribute	7
Figure 2: Errors	7
Figure 3: Panic function.....	8
Figure 4: Adding dev & release	8
Figure 5: Error 2.....	8
Figure 6: Entrypoint.....	9
Figure 7: Define entry point.....	9
Figure 8: Error 3.....	9
Figure 9: Building bare-metal 1	10
Figure 10: main.rs file	11
Figure 11: Cargo.toml file.....	11
Figure 12: Target specification 1	13
Figure 13: Target specification 2	14
Figure 14: Target specification 3	14
Figure 15: Target specification 4	14
Figure 16: Target specification 5	14
Figure 17: Target specification 6	14
Figure 18: Assemble code.....	15
Figure 19: Create Rust module 1	16
Figure 20: Colors	17
Figure 21: Color code	17
Figure 22: Text buffer	18
Figure 23: Create writer 1	19
Figure 24: Create writer 2	19
Figure 25: Code in action 1.....	20
Figure 26: Code in action 2.....	20
Figure 27: cargo test running	21
Figure 28: main.rs	22
Figure 29: main.rs 2	23
Figure 30: Running tests	23
Figure 31: Test output.....	23
Figure 32: Hardware Interrupts 1.....	26
Figure 33: Hardware Interrupts 2.....	27
Figure 34: Paging example	28
Figure 35: Page table	29
Figure 36: Multiple page table 1	30
Figure 37: Multiple page table 2	31

Figure 38: Create memory module	32
Figure 39: Accessing page table 1	32
Figure 40: Accessing page table 2	33
Figure 41: Output of accessing page table	34
Figure 42: Turn the mapped frame of an entry back into a virtual address	34
Figure 43: Translating address 1	35
Figure 44: Translating address 2	36
Figure 45: Translating address 3	37
Figure 46: Output of translating address	37
Figure 47: Adding an init function to the memory module	39
Figure 48: Kernal main	40
Figure 49: Output of utilizing OffsetPageTable	40
Figure 50: Create_example_mapping function	42
Figure 51: Create EmptyFrameAllocator	43
Figure 52: Page table example	44
Figure 53: create_example_mapping	46
Figure 54: Output of running in QEMU	46
Figure 55: Map page 0xdeadbeaf000 instead of 0	47
Figure 56: Error message	47
Figure 57: Allocating frames 1	48
Figure 58: FrameAllocator trait	49
Figure 59: FrameAllocator trait 2	50
Figure 60: BootInfoFrameAllocator	51
Figure 61: Heap allocation	52
Figure 62: Bump Allocator	58
Figure 63: Linked List Allocator	58
Figure 64: Fixed-Size Block Allocator	59
Figure 65: async/await implementation 1	61
Figure 66: async Lifetimes 1	62
Figure 67: async Lifetimes 2	62
Figure 68: async move	63

1. Introduction

Rust is a well-established, practical manufacturing technology. You can keep your finger on the pulse of the machine's inner workings with this systems programming language. Data can be stored on either the stack (for static memory allocation) or the heap (dynamic memory allocation) (used for dynamic memory allocation). Here, we should highlight RAII (Resource Acquisition Is Initialization), a programming idiom typically associated with C++ but present in Rust: if an object leaves the scope of an application, the destructor for that object is invoked, and any resources it owns are released. This prevents resource leakage bugs and eliminates the need for manual intervention. In the end, it allows for more effective application of stored information within the mind's memory. With their Skylight product, Tilde rewrote some Java HTTP endpoints in Rust. They went from using 5 GB of memory to just 50 MB thanks to this change. Because Rust's garbage collector does not operate in the background, its libraries can be accessed by other languages through their respective foreign-function interfaces. For ongoing tasks, this is the best case since high performance and memory security are both guaranteed. In this scenario, rust code can be used to replace portions of software whose performance is paramount without having to redo the complete product. Rust's immediate access to hardware and memory makes it an ideal choice for integrated and bare-metal development because it's a low-level language. Rust may be used to create programs for both desktop computers and microcontrollers. Several OSes, including Redox, intermezzOS, QuiltOS, Rux, and Tock, have been developed entirely in Rust. As its original developer, Mozilla incorporates the language into its browser engines. Because of its high performance and security, scientists have begun utilizing Rust to conduct intensive data analysis. Rust is often used in fields like computational biology and machine learning because it can do things so quickly.

2. A Freestanding Rust Binary

1.1 Introduction

The kernel of an operating system must be developed from scratch without using any of the system's preexisting features. We can't use OS abstractions or specialized hardware, so we can't use threads, files, heap memory, the network, random numbers, or the standard output. It makes sense for us to try to develop our own drivers and OS.

Unfortunately, this means we can't use the vast majority of the Rust standard library, while we can still leverage many of the language's features. Some of the tools at our disposal include iterators, closures, pattern matching, options, results, string formatting, and the ownership system. These features make it possible to make a high-level kernel that is very expressive without having to worry about things like memory safety and behavior that isn't clear.

If we want to use Rust to create an OS kernel, we must be able to create a standalone executable. This kind of app is sometimes called a "freestanding executable" or a "bare metal executable."

1.1.1 The `no_std` Attribute

The standard library is used automatically by Rust, so we will turn that off. By using the `#![no_std]` attribute, we can instruct the compiler to avoid including the standard library in its output.

```
1  #![no_std]
2  fn main() {
3  }
```

Figure 1: `no_std` attribute

Here are some of the errors we encountered when constructing:

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

Figure 2: Errors

`#[panic_handler]`

When Rust encounters an error during execution, it invokes the panic handler, which then dumps the crash stack. This is part of the base set of libraries available. Since we've turned off the default library, an alternative must be provided.

```

1 use core::panic::PanicInfo;
2
3 /// Panic function.
4 #[panic_handler]
5 fn panic(_info: &PanicInfo) -> ! {
6     loop {}
7 }

```

Figure 3: Panic function

When a function is never supposed to return, it is marked with The!. One would classify this as a diverging function. A useless loop is added to the panic section of the tutorial because we aren't actually doing anything there.

eh_personality

Something pertaining to the English language. The characteristics of language items provide the compiler with additional information about the function, and it is these properties that it looks for while parsing the code. The compiler is informed by the eh_personality type that this method is responsible for implementing stack unwinding. In other words, it's the code that gets run when a function is "popped" off the function stack. Because of this, the function's allocated memory is likely released. Let's disable it by editing Cargo.toml to read as follows.

```

1 [profile.dev]
2 panic = "abort"
3
4 [profile.release]
5 panic = "abort"

```

Figure 4: Adding dev & release

After addressing both of the issues we encountered, we're now attempting to construct it:

```
error: requires `start` lang_item
```

Figure 5: Error 2

In other words, contrary to popular belief, the main function is not the very first function called in a Rust program. As a result, we are not making use of the crt0 and start linguistic constructs. Therefore, it is up to us to establish our own threshold.

1.1.2 Defining our entrypoint

The first step is to inform the compiler that we are not going to be including a runtime. Add a `#![no main]` attribute to do this. Since there is no longer a runtime to invoke the main function, it can be removed.

```
1  #![no_std]
2  #![no_main]
3
4  use core::panic::PanicInfo;
5
6  /// Panic function.
7  #[panic_handler]
8  fn panic(_info: &PanicInfo) -> ! {
9      loop {}
10 }
```

Figure 6: Entrypoint

Our entry point must be decided upon now:

```
1  #[no_mangle]
2  pub extern "C" fn _start() -> ! {
3      loop {}
4  }
```

Figure 7: Define entry point

It turns out that while compiling, the Rust compiler gives each function a new, random name. It's called "name mangling" when this happens. We're turning it off in case we ever need to call our start function by its full name.

We must tell the compiler that this function should be called using the C standard, in addition to using the `extern "C"` notation.

Also, by adding the `!`, the function will become a diverging function that must never return. (We may decide to make it a call to power off the system in the future.)

A linking error occurs when attempting to construct the current state of affairs, which is as follows:

```
error: linking with `link.exe` failed: exit code: 1561
```

Figure 8: Error 3

Rust's default behavior during the compilation process is to aim for an executable that can function in the user's current operating system. In this case, because I'm using a Windows x86_64 PC, it attempts to compile the executable file using x86_64 instructions.

The linker makes the safe assumption that we're utilizing the C runtime by default. To get around this, a bare-metal target building is being developed.

1.3 Building for bare-metal

A Target triple is a string used in Rust to specify various environments.

The string's function is to characterize the hardware setup of the device.

In order to compile for our host triple, the Rust compiler and linker make the incorrect assumption that the underlying operating system is either Linux or Windows, both of which use the C runtime by default and hence generate linker errors. Linker errors can be avoided by compiling for a separate environment in which the OS is not present. One such example is targets like thumbv7em-none-eabihf.

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

Figure 9: Building bare-metal 1

Before we can build, we have to cross-compile our executable for a bare metal target.

1.4 Code

```
1  #![no_std] // don't link the Rust standard library
2  #![no_main] // disable all Rust-level entry points
3
4  use core::panic::PanicInfo;
5
6  #[no_mangle] // don't mangle the name of this function
7  pub extern "C" fn _start() -> ! {
8      // this function is the entry point, since the linker looks for a
9      // function
10     // named `_start` by default
11     loop {}
12 }
13
14 /// This function is called on panic.
15 #[panic_handler]
16 fn panic(_info: &PanicInfo) -> ! {
17     loop {}
18 }
```

Figure 10: main.rs file

```
1  [package]
2  name = "crate_name"
3  version = "0.1.0"
4  authors = ["Author Name <author@example.com>"]
5
6  # the profile used for `cargo build`
7  [profile.dev]
8  panic = "abort" # disable stack unwinding on panic
9
10 # the profile used for `cargo build --release`
11 [profile.release]
12 panic = "abort" # disable stack unwinding on panic
```

Figure 11: Cargo.toml file

3. A Minimal Rust Kernel

We'll recall that we used cargo to construct the standalone binary and that different OSes required unique entry point names and compilation settings. This is because cargo always targets your current operating system when constructing packages. We don't want this for our kernel because it's not really practical to have a kernel that runs on top of another operating system like Windows. We'd prefer to compile for a certain platform instead.

2.1 Installing Rust Nightly

Stable, beta, and nightly are the three distinct Rust release channels. See if you can find some time to read The Rust Book, because it has a fantastic explanation of the distinctions between these channels. Rust must be installed from the nightly channel because the OS build process needs to use some experimental features that aren't stable enough for the stable channel just yet.

I recommend rustup for managing Rust installs. Parallel installation of nightly, beta, and stable compilers is supported, as is simple updating of all of them. Running rustup with the override set to nightly switches the current directory to using the nightly compiler. Alternately, you can place the nightly content in a file named rust-toolchain in the root directory of your project. To see if a nightly version has been installed, type "rustc —version" into a terminal. It's important that the -nightly suffix be included in the version number.

The nightly compiler lets us enable new features by setting "feature flags" at the beginning of our code. By including the directive `#![feature(asm)]` at the start of our main.rs file, we can enable the experimental asm! macro for inline assembly. The full unpredictability of experimental features means that they may be altered or removed entirely in subsequent versions of Rust. This is why we will only employ them in extreme circumstances.

2.2 Target Specification

The target option allows you to choose a different system to which Cargo will deliver its contents. Targeting information includes CPU architecture, vendor, OS, and application binary interface (ABI). This is illustrated by the x86_64-unknown-linux-gnu target triple, which identifies a computer with an x86_64 CPU, no obvious vendor, and Linux using the GNU application binary

interface. If you use the target triple arm-linux-androideabi, you can build an Android app in Rust. If you use the target triple wasm32-unknown-unknown, you can build a WebAssembly app.

Unfortunately, none of the available target triples are suitable for our system because it necessitates unusual configuration settings (for example, no underlying operating system). The good news is that Rust supports defining our own target in a JSON file. For instance, here is how a JSON file describing the x86_64-unknown-linux-gnu target would look:

```
1 {  
2   "llvm-target": "x86_64-unknown-linux-gnu",  
3   "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",  
4   "arch": "x86_64",  
5   "target-endian": "little",  
6   "target-pointer-width": "64",  
7   "target-c-int-width": "32",  
8   "os": "linux",  
9   "executables": true,  
10  "linker-flavor": "gcc",  
11  "pre-link-args": ["-m64"],  
12  "morestack": false  
13 }
```

Figure 12: Target specification 1

LLVM needs information from most fields in order to produce platform-specific code. The sizes of various data types like integers, floats, and pointers are specified in the data-layout field. Then there are fields like target-pointer-width that Rust uses for conditional compilation. The crate's construction is specified by the third type of field. The pre-link-args field, for instance, is used to define the parameters sent to the linker.

Our kernel is designed to run on x86_64 architectures, and as such, our target specification will look quite similar to the one shown above. Let's get started by making a file (call it what you will) named x86_64-blog_os.json that contains the following:

```
1 {  
2   "llvm-target": "x86_64-unknown-none",  
3   "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",  
4   "arch": "x86_64",  
5   "target-endian": "little",  
6   "target-pointer-width": "64",  
7   "target-c-int-width": "32",  
8   "os": "none",  
9   "executables": true  
10 }
```

Figure 13: Target specification 2

As we will be running on bare metal, we have set the OS in the `llvm-target` and the `os` field to none. We add the following build-related entries:

```
"linker-flavor": "ld.lld",  
"linker": "rust-lld",
```

Figure 14: Target specification 3

For linking our kernel, we use the cross-platform LLD linker that comes with Rust instead of the platform's default linker, which might not support Linux targets.

```
"panic-strategy": "abort",
```

Figure 15: Target specification 4

When this option is selected, the program is told to terminate immediately upon encountering a panic condition, as the target does not enable stack unwinding on panic. We can now remove the `panic = "abort"` option from our `Cargo.toml` because it serves the same purpose as this.

```
"disable-redzone": true,
```

Figure 16: Target specification 5

Since we are creating a kernel, interruptions will need to be managed at some point. A stack pointer optimization known as the "red zone" must be disabled to avoid introducing corruption into the stack; enabling it would be dangerous. If you want to know more about turning off the danger zone, please read that article instead.

```
"features": "-mmx,-sse,+soft-float",
```

Figure 17: Target specification 6

Features are toggled on and off in the features box. Disabling `mmx` and `sse` is accomplished by adding a minus sign before the feature name, while adding a plus sign to `soft-float` enables it. It's important to remember that LLVM won't be able to decipher the features string if there are spaces between the various flags.

Support for SIMD instructions, which can considerably speed up programs, is determined by the `mmx` and `sse` characteristics. While the massive SIMD registers are convenient, they cause

performance issues when used in operating system kernels. This is because, before picking up where a stopped program left off, the kernel must first reset all registers to their original values. The kernel must perform a full SIMD state save to main memory on every system call and hardware interrupt. These extra save/restore operations have a significant negative impact on performance because the SIMD state is huge (512-1600 bytes) and interrupts can occur often. To prevent this, our kernel no longer supports SIMD.

Because x86_64's floating point operations by default use SIMD registers, turning off SIMD presents a challenge. This problem was solved by adding the soft-float feature, which mimics the behavior of floating-point calculations with regular integer-based software functions.

2.3 Assembling the Parts

This is the current version of our target specification file.

```
{
  "llvm-target": "x86_64-unknown-none",
  "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
  "arch": "x86_64",
  "target-endian": "little",
  "target-pointer-width": "64",
  "target-c-int-width": "32",
  "os": "none",
  "executables": true,
  "linker-flavor": "ld.lld",
  "linker": "rust-lld",
  "panic-strategy": "abort",
  "disable-redzone": true,
  "features": "-mmx,-sse,+soft-float"
}
```

Figure 18: Assemble code

4. VGA Text Mode

The VGA text buffer is a 25 by 80 pixel array that is displayed directly on the screen. Two bytes, or 16 bits, are used to represent each character on the screen. Character attributes are represented by 8 bits, whereas code points are represented by the remaining 8 bits. In accordance with Unicode specifications, a code point will be assigned to each and every character. A, for instance, will be represented by the code point "U+0041." The Unicode family includes UTF-8, UTF-16, and others. The first byte stands in for the character itself, while the second byte specifies how that character should be displayed, with the first four bits indicating the foreground color, the next three indicating the background color, and the last bit indicating the blink attribute.

Because the VGA text buffer is memory-mapped I/O at address 0xb8000, we can read and write to it using standard memory operations.

Create Rust module and define colors

Of course, we know that the `mod` keyword is required if we want to make a new module in Rust. Let's make a module to control screen output text output.

In the `main.rs` file, add the following line to produce a binary module:

```
mod vga_buffer;
```

Figure 19: Create Rust module 1

3.1 Different colors in this module.

```
#[allow(dead_code)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u8)]
pub enum Color {
    Black = 0,
    Blue = 1,
    Green = 2,
    Cyan = 3,
    Red = 4,
    Magenta = 5,
    Brown = 6,
    LightGray = 7,
    DarkGray = 8,
    LightBlue = 9,
    LightGreen = 10,
    LightCyan = 11,
    LightRed = 12,
    Pink = 13,
    Yellow = 14,
    White = 15,
}
```

Figure 20: Colors

We've defined colors using an enum, and thanks to the `repr(u8)` feature, each color option will be stored as an `u8`. To prevent the compiler from issuing a warning for an unused variation, we've included the `#[allow(dead_code)]` attribute and derived characteristics such as `Copy`, `clone`, `Debug`, `PartialEq`, and `Eq`.

Let's get down to the nitty-gritty of setting up a color code that defines the colors of the foreground and the background.

3.1.1 Provide color code

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(transparent)]
struct ColorCode(u8);

impl ColorCode {
    fn new(foreground: Color, background: Color) -> ColorCode {
        ColorCode(((background as u8) << 4 | (foreground as u8)))
    }
}
```

Figure 21: Color code

By using the `repr(transparent)` trait, we guaranteed that both the foreground and background colors in this structure were the same.

Alright, we've given you the hexadecimal value for each color, and now it's time to give you the structure for each screen character and the text buffer itself.

3.1.2 Provide text buffer

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(C)]
struct ScreenChar {
    character: u8,
    color_code: ColorCode,
}

const BUFFER_HEIGHT: usize = 25;
const BUFFER_WIDTH: usize = 80;

#[repr(transparent)]
struct Buffer {
    chars: [[ScreenChar; BUFFER_WIDTH]; BUFFER_HEIGHT],
}
```

Figure 22: Text buffer

We have provided a structure for screen characters, and the following structure is for the text buffer.

The first line is self-explanatory, while the `repr(c)` property ensures that the struct's fields are set out in the same way as in a C struct, hence ensuring the correct field ordering. Additionally, the screen character has been represented by "character" and "color_code" in the `ScreenChar`. We used `BUFFER HEIGHT` and `BUFFER WIDTH` to represent the 2580 pixel width and height of the VGA text buffer.

Specifically, the `repr(transparent)` feature has been utilized to guarantee that the `Buffer` struct's sole field uses the same memory layout as itself. We've just included one data-related field within this structure.

3.1.3 Create Writer

Let's first establish why we need a writer, shall we? The writer, as its name implies, is responsible for displaying the value graphically. As a result, we must consider what qualities or characteristics our writer should possess.

If we want to tell the writer where to start writing, we need to tell it what to write, what color to write in, and in what column to start writing; we don't need to tell it the row number, as it will always start at the end of the line.

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

Figure 23: Create writer 1

The Writer data type is described in the code above. It has three members: `column_position`, `color_code`, and `buffer`. `usize` denotes the number of bytes in a buffer, `color_code` is the structure we just built, and `column_position` is the size of the `usize`. It's time to supply the writer's implementation now that we've established its structure.

```
pub fn write_byte(&mut self, byte: u8) {  
    let row = BUFFER_HEIGHT - 1;  
    let col = self.column_position;  
  
    let color_code = self.color_code;  
    self.buffer.chars[row][col].write(ScreenChar {  
        character: byte,  
        color_code,  
    });  
    self.column_position += 1;  
}
```

Figure 24: Create writer 2

We've added `write_byte` as the initial implementation of the writer structure. The approach discussed here is for displaying bytes on the screen. We first reduce the buffer's height by one because the writer always writes to the last line, then we get the column position and the text color, and last we set the buffer's byte and color. `chars` to output the data, and finally we increase the column position to move the pointer. Now that everything is set up, we can run our code, but first we need to give the writer some input that will be shown on the screen.

3.1.4 Code in Action

```
pub fn write_byte(&mut self, byte: u8) {
    let row = BUFFER_HEIGHT - 1;
    let col = self.column_position;

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

pub fn print_data() {
    let mut writer = Writer {
        column_position: 0,
        color_code: ColorCode::new(Color::Yellow, Color::Black),
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };

    writer.write_byte(b'K');
    writer.write_byte(b'N');
    writer.write_byte(b'O');
    writer.write_byte(b'L');
    writer.write_byte(b'D');
    writer.write_byte(b'U');
    writer.write_byte(b'S');
}
```

Figure 25: Code in action 1

This function begins by making a new `Writer` that uses the `0xb8000` address as the target for the VGA buffer. Next, we changed the type of integer `0xb8000` to raw pointer, which can be modified. Next, we make it a mutable reference by dereferentializing it (through `*`) and borrowing it again right away (through `&mut`). Since the compiler can't be sure that the raw pointer is correct, this conversion needs an `unsafe` block. Finally, a byte was passed to the `write_byte()` method. Now, from the `_start()` method, we will invoke `print_data()`.

```
#[no_mangle]
pub extern "C" fn _start() -> ! {
    vga_buffer::print_data();

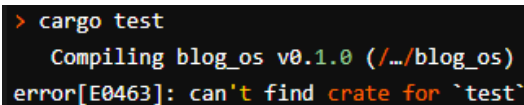
    loop {}
}
```

Figure 26: Code in action 2

5. Testing

There is no need to configure anything special to run unit tests in Rust because of the language's in-built test framework. Make a new function whose header has the `#[test]` attribute, and then use assertions to test its output. Then, your crate's built-in test routines will be discovered and run without further input from you, thanks to cargo test.

Applications that do not conform to the `no_std` standard, like our kernel, make this process more difficult. The issue is that the standard library is required for the Rust test framework because it uses the built-in test library. We can't use the standard testing infrastructure because our kernel is `#[no_std]`.



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

Figure 27: cargo test running

Due to its reliance on the standard library, the test crate is not supported on our bare metal target. The test crate can be ported to a `#[no_std]` environment, but doing so is highly unstable and necessitates hacks like rewriting the panic macro.

4.1 Custom Test Frameworks

Thankfully, the unreliable custom test frameworks feature in Rust allows you to swap out the language's standard test framework. This capability does not rely on any third-party libraries; hence it is compatible with `#[no_std]` setups. The method works by first accumulating all `#[test_case]`-annotated functions and then passing that list of tests to a runner function that the user specifies. Therefore, the implementation has complete autonomy over the testing procedure.

Disadvantages include missing support for a variety of high-level features, such as `should_panic` tests, when compared to the standard testing framework. If such functionality is required, it must be provided by the implementation. This is perfect, as our execution environment is quite unique and the standard implementations of such sophisticated features would likely not function

correctly. If you look at the `#[should_panic]` attribute, for instance, it uses stack unwinding to catch panics, which we disabled in our kernel.

This is what we change in `main.rs` to make a custom environment for testing our kernel:

```
// 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());
    for test in tests {
        test();
    }
}
```

Figure 28: main.rs

Our runner just iterates over the list of test functions, printing a little debug message between each one. `& [&dyn Fn()]` is a trait object reference slice that can be used as an argument to the `Fn()` trait. It's essentially a collection of pointers to different types that can be invoked just like functions. The `#[cfg(test)]` attribute is used to restrict its use to test runs only, as the function has no practical purpose outside of that context.

If the cargo test is run now, we should notice that it succeeds (if it doesn't, check the caveat below). Despite this, "Hello World" continues to appear instead of the test runner's message. The reason for this is that our `_start` method is still the starting point for our application. Using the `#[no_main]` attribute, we override the custom test framework's default entry point and prevent it from calling the generated main function, which in turn calls `test_runner`.

With the `reexport_test_harness_main` attribute, we can rename the produced function from `main` to something else. If we rename a function, we can then use it in 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 {}
}
```

Figure 29: main.rs 2

From our `_start` entry point, we invoke the test framework's entry function, `test main`, which we have renamed to reflect its purpose. Since the `test main` function isn't made when the program is run normally, we use conditional compilation to make sure that the call to `test main` is only made during tests.

The test runner now displays "Running 0 tests" when we call a cargo test. Developing the initial test function is next:

```
// in src/main.rs

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

Figure 30: Running tests

The following is the current output from the cargo test:

```
Hello world!
Running 1 tests
trivial assertion... [ok]
```

Figure 31: Test output

6. CPU Exceptions

When something goes wrong, the CPU throws an exception, which is a special kind of interrupt. In most cases, exceptions like page faults are not errors.

A few notable exceptions are:

Faults: These can be fixed, and the show can go on as planned.

Traps: Immediately following the execution of the trapping instruction, the results are reported.

Aborts: There has been a catastrophic misstep.

For more detailed information about an exception, some of them will push a 32-bit "error code" to the top of the stack. Prior to resuming execution, this value must be retrieved from the stack.

Page Fault:

A page fault will result when an unlawful section of memory is accessed. For instance, if the current instruction attempts to read from a page that has not been mapped or tries to write to a page that may only be read, an error will occur.

Invalid Opcode:

This exception is thrown if the currently executed instruction cannot be executed. For instance, it happens when we attempt to use modern SSE instructions on an older CPU that does not support them.

Failure of General Protection:

This is the one and only exception that has a wide variety of potential reasons. It happens when access rights have been violated in a variety of ways, such as when user-level code attempts to carry out privileged instructions or when configuration registers have reserved fields that are written to.

Double Fault:

The Central Processing Unit (CPU) will attempt to invoke the appropriate handler function whenever an exception is thrown. A double fault exception is generated by the CPU whenever there is more than one exception during the process of contacting the exception handler. This exception will also happen when there isn't a handler function set up for an exception.

Triple Fault:

A catastrophic fatal triple fault is generated by the central processing unit (CPU) if an error occurs while it is attempting to call the double fault handler code. We are unable to prevent or deal with a triple fault. Most CPUs will respond by resetting themselves, which will also restart the operating system.

7. Hardware Interrupts

Despite their differences, interrupts and exceptions are managed by the same interrupt controller and share many commonalities in operation and application. Even though the Cortex-M architecture defines what an exception is and how it should be handled, interrupts are always done in a way that is unique to the vendor (and sometimes even the chip).

The complexity of sophisticated interrupt usage arises from the fact that interrupts provide a great deal of freedom. We won't talk about those applications here, but you should still keep in mind the following:

- The order in which interrupt handlers run can be adjusted by the programmer.
- It is possible for a higher priority interrupt to stop a lower priority interrupt from running, and vice versa. This is called "interrupt nesting" or "interrupt preemption."
- Commonly, the interrupt's trigger is cleared so that the handler doesn't keep being called over and over again.

At any given point in time, the following actions will always be taken to initiate the program:

- A set of interrupts can be triggered at predetermined times by configuring the relevant peripheral(s).
- Arrange the interrupt controller's handler priority as needed.
- The interrupt controller's handler must be turned on.

Cortex-m-rt supports an interrupt attribute, analogous to exceptions, for declaring interrupt handlers. From an SVD description, svd2rust can automatically create the interrupts that are available and their place in the table of interrupt handlers.

```
1 // Interrupt handler for the Timer2 interrupt
2 #[interrupt]
3 fn TIM2() {
4     // ..
5     // Clear reason for the generated interrupt request
6 }
```

Figure 32: Hardware Interrupts 1

Just like exception handlers, interrupt handlers take the form of simple functions. Their unique calling conventions, however, prevent them from being called directly from anywhere else in

the firmware. But software interrupt requests can be made to make the operating system switch to the interrupt handler.

Static mut variables can be declared inside interrupt handlers for secure state maintenance just like they can inside exception handlers.

```
1 #[interrupt]
2 fn TIM2() {
3     static mut COUNT: u32 = 0;
4
5     // `COUNT` has type `&mut u32` and it's safe to use
6     *COUNT += 1;
7 }
```

Figure 33: Hardware Interrupts 2

8. Introduction to Paging

The concept is to segment both virtual and physical memory into chunks of uniform size. Pages are the building blocks of a computer's virtual memory, while frames are the building blocks of a computer's physical address space. By mapping each page to a different physical frame, larger memory regions can be split up into smaller pieces that don't share the same frame. When we redo the memory fragmentation example with paging instead of segmentation, the benefit becomes clear.

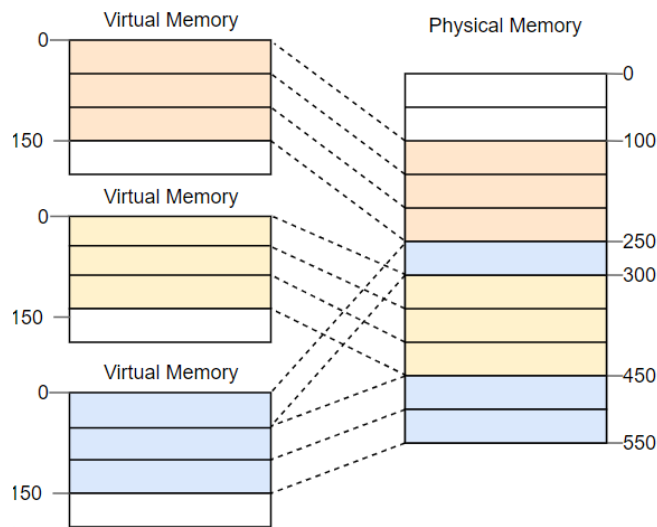


Figure 34: Paging example

Assuming a page size of 50 bytes, each of our memory regions would be spread across three pages. Since a continuous virtual memory region can be transferred to non-continuous physical frames, this method is known as page-by-page mapping. This means we can launch the third instance of the software without first defragmenting.

7.1 Hidden Fragmentation

Instead of using a small number of large sections, segmentation, paging employs a huge number of smaller regions of fixed size. No fragmentation takes place since there are no unusable little frames due to the uniformity of frame size.

If no fragmentation is apparent, then neither. Even if it seems like the pieces are coming back together, there is still "internal fragmentation" at play. When the page size is not exactly divisible by all the memory regions, internal fragmentation occurs. For this example, let's pretend the size of the program is 101 lines. Three pages of size 50 are still required, making the unnecessary space taken up by this text 49 bytes. Segmentation-induced fragmentation is distinguished from other types of fragmentation by the term "external fragmentation."

The internal fragmentation that results from segmentation is undesirable but typically preferable to the exterior fragmentation that normally happens. Even though it still wastes RAM, it eliminates the need for defragmentation and makes the level of fragmentation predictable (on average half a page per memory region).

7.2 Page Tables

We observed that every one of the millions of pages is uniquely mapped to a frame. The location of this mapping data must be preserved for future use. While paging can utilize a single segment selection register for all pages, segmentation requires a separate register for each page in use. Instead, paging uses a structure called a "page table" to keep track of mapping information.

The resulting page tables would look like this for the scenario we just discussed:

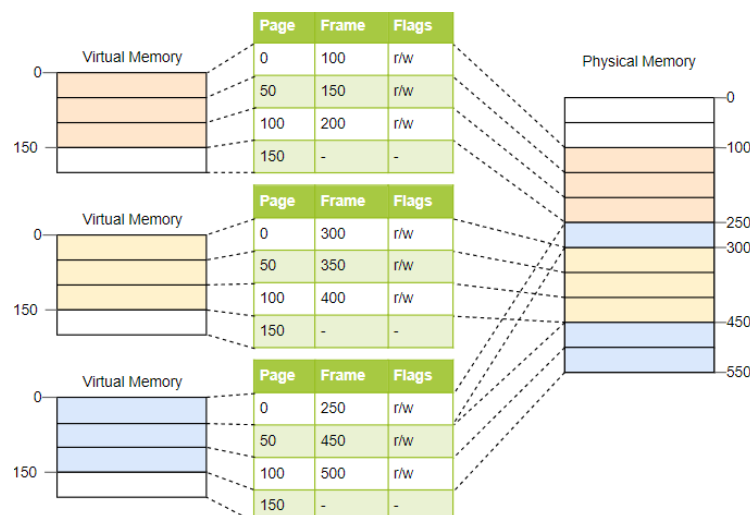


Figure 35: Page table

It's clear that each process runs its own copy of the page table. A specific register in the CPU keeps track of the active table and can be accessed with the appropriate pointer. This x86 register is labeled as CR3. Before a new instance of a program can be run, the operating system must put a pointer to the right page table into this register.

At the time of each memory access, the CPU retrieves the table pointer from the register and consults it to find the mapped frame corresponding to the page being visited. This processing takes place entirely in hardware and is completely transparent to the application. Many CPU designs keep track of the previous translations' outcomes in a dedicated cache to speed up the operation.

Flags fields in page table entries may also be used to hold information like access rights, albeit this depends on the underlying architecture. The "r/w" flag enables reading and writing in the preceding example.

7.3 Multilevel Page Tables

When dealing with bigger address spaces, the straightforward page tables we've been looking at thus far become inefficient. Consider a program that uses the following virtual addresses (0, 1_000_000, 1_000_050, and 1_000_100; the underscore "_" is used to separate thousands):

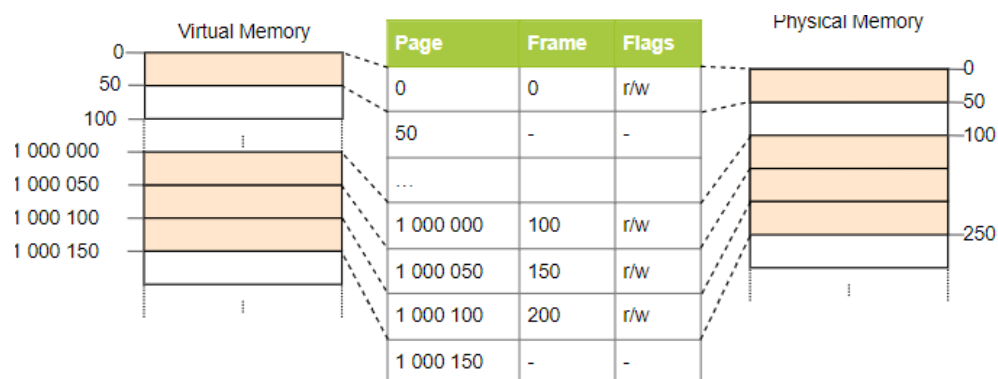


Figure 36: Multiple page table 1

Although only 4 frames are required physically, the page table contains over a million records. If we left out the empty entries, the computer wouldn't be able to skip forward to the right one during translation. Using a two-level page table is one way to cut down on unnecessary memory usage.

The goal is to use separate page tables for each address range. Address areas are mapped to their corresponding page tables (level 1) in a separate table termed the "level 2 page table."

An illustration is the best way to clarify this point. Let's say we assign a size of 10 000 bytes to each level 1 page table. This hypothetical mapping would require the following tables to exist:

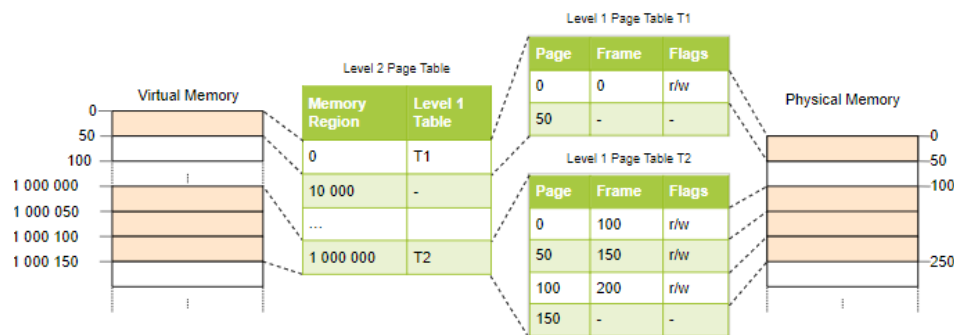


Figure 37: Multiple page table 2

Because page 0 occupies the first 10 000 bytes, it uses the first entry in the level 2 page table. This entry refers to the table T1 on page 1 of level 1, which identifies frame 0 as being referenced by page 0. All three of the pages labeled 1_000_000, 1_000_050, and 1_000_100 reside in the 100th 10 000 byte region and hence employ the 100th item in the level 2 page table. This entry refers to a unique page table T2 on level 1, which assigns frames 100, 150, and 200 to the aforementioned pages. Keep in mind that the region offset is not a part of the page address in level 1 tables. For instance, on page 1_000_050, there is only the number 50 recorded.

Even though we've reduced the number of blank rows in the level 2 table from 1,000,000 to 100, we still have a lot of space there. To save this much money, we don't have to generate level 1 page tables for the unmapped memory areas between 10_000 and 1_000_000.

Two-level page tables can easily be expanded to include three, four, or more layers of data. Following this, the highest-level table in the hierarchy is referenced by the page table register, which in turn references the next-lowest level table in the hierarchy, and so on. Once the mapping is complete, the level 1 page table will reference it. A multilayer or hierarchical page table describes the overarching concept here. With our newfound knowledge of paging and multiple page tables, we can investigate the x86_64 architecture's paging implementation.

9. Paging Implementation

Incorporating our page table code is the next logical step after gaining access to physical memory. To begin, we will inspect the live page tables that our kernel is using right now. Our next action is to design a translation function that gives us back the corresponding physical address for a given virtual one. Finally, we'll try to make changes to the page tables to generate a new mapping.

Before starting to write our program, we create a new memory module:

```
// in src/lib.rs  
  
pub mod memory;
```

Figure 38: Create memory module

A blank src/memory.rs file is made for the module.

8.1 Accessing the Page Tables

From there, we can move on by implementing the active level 4 table method, which will return a pointer to the table containing the currently active level 4 pages.

```
// in src/memory.rs  
  
use x86_64::{  
    structures::paging::PageTable,  
    VirtAddr,  
};  
  
/// Returns a mutable reference to the active level 4 table.  
///  
/// This function is unsafe because the caller must guarantee that the  
/// complete physical memory is mapped to virtual memory at the passed  
/// `physical_memory_offset`. Also, this function must be only called once  
/// to avoid aliasing `&mut` references (which is undefined behavior).  
pub unsafe fn active_level_4_table(physical_memory_offset: VirtAddr)  
    -> &'static mut PageTable  
{  
    use x86_64::registers::control::Cr3;  
  
    let (level_4_table_frame, _) = Cr3::read();  
  
    let phys = level_4_table_frame.start_address();  
    let virt = physical_memory_offset + phys.as_u64();  
    let page_table_ptr: *mut PageTable = virt.as_mut_ptr();  
  
    &mut *page_table_ptr // unsafe  
}
```

Figure 39: Accessing page table 1

First, we access the CR3 register for the physical frame of the level 4 table that is currently active. The logical location of the page table frame is calculated by taking its physical start address, casting it as an u64, and appending the result to the physical memory offset. Finally, we unsafely build a mut PageTable reference by converting the virtual address to a *mut PageTable raw pointer using the as_mut_ptr method. Since we'll be altering the page tables later on, we'll use a &mut reference instead of a regular & reference.

Due to the way unsafe functions are handled in Rust, we can avoid using an unsafe block in this case. Since we might not realize we just introduced a potentially dangerous operation in the lines before, this makes our code riskier. Moreover, it makes it harder to distinguish potentially dangerous activities from generally safe ones. An RFC has been created to modify this action.

It is now possible to use this function to print the entries of the data included in the level-four table:

```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    use blog_os::memory::active_level_4_table;
    use x86_64::VirtAddr;

    println!("Hello World{}", "!");
    blog_os::init();

    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let l4_table = unsafe { active_level_4_table(phys_mem_offset) };

    for (i, entry) in l4_table.iter().enumerate() {
        if !entry.is_unused() {
            println!("L4 Entry {}: {:?}", i, entry);
        }
    }

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

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

Figure 40: Accessing page table 2

In order to use the active_level_4_table function, we must first cast the physical memory offset member of the BootInfo struct to a VirtAddr. The entries in the page table are then iterated over

using the `iter` function, and the `enumerate` combinator is used to append an index to each item in the table. Due to screen real estate limitations, we only print entries that contain data.

The output looks like this when we execute it:

```
Hello World?
L4 Entry 0: PageTableEntry { addr: PhysAddr(0x2000), flags: PRESENT : WRITABLE :
  ACCESSED }
L4 Entry 2: PageTableEntry { addr: PhysAddr(0x441000), flags: PRESENT : WRITABLE
  : ACCESSED : DIRTY }
L4 Entry 3: PageTableEntry { addr: PhysAddr(0x449000), flags: PRESENT : WRITABLE
  : ACCESSED : DIRTY }
L4 Entry 3: PageTableEntry { addr: PhysAddr(0x445000), flags: PRESENT : WRITABL
  E : ACCESSED : DIRTY }
It did not crash?
.....
```

Figure 41: Output of accessing page table

Multiple non-blank items are displayed, each of which corresponds to a unique table on level 3. This is due to the fact that the kernel's code, kernel stack, physical memory mapping, and boot information all need their own dedicated regions of memory.

To learn more about page tables, we need to turn the mapped frame of an entry back into a virtual address. Here's how to do that:

```
// in the `for` loop in src/main.rs

use x86_64::structures::paging::PageTable;

if !entry.is_unused() {
    println!("L4 Entry {}: {:?}", i, entry);

    // get the physical address from the entry and convert it
    let phys = entry.frame().unwrap().start_address();
    let virt = phys.as_u64() + boot_info.physical_memory_offset;
    let ptr = VirtAddr::new(virt).as_mut_ptr();
    let l3_table: &PageTable = unsafe { &*ptr };

    // print non-empty entries of the level 3 table
    for (i, entry) in l3_table.iter().enumerate() {
        if !entry.is_unused() {
            println!("  L3 Entry {}: {:?}", i, entry);
        }
    }
}
```

Figure 42: Turn the mapped frame of an entry back into a virtual address

If we want to examine the level 2 and level 1 tables, we need to repeat the steps we just took to examine the level 3 entries. The complete code is not shown since, as you might expect, it quickly becomes unmanageably long and complex.

It's interesting to learn more about how the CPU translates by exploring the page tables by hand. While this is useful in some cases, most of the time we only want to know which physical address corresponds to a given virtual one, so let's make a function for that.

8.2 Translating Addresses

One must go through the four levels of the page table until they reach the mapped frame in order to convert a virtual address to a physical one. Let's write a program that translates this:

```
// in src/memory.rs

use x86_64::PhysAddr;

/// Translates the given virtual address to the mapped physical address, or
/// `None` if the address is not mapped.
///
/// This function is unsafe because the caller must guarantee that the
/// complete physical memory is mapped to virtual memory at the passed
/// `physical_memory_offset`.
pub unsafe fn translate_addr(addr: VirtAddr, physical_memory_offset: VirtAddr)
    -> Option<PhysAddr>
{
    translate_addr_inner(addr, physical_memory_offset)
}
```

Figure 43: Translating address 1

To limit the function's unsafe impact, we redirect it to the secure `translate_addr_inner`. As was mentioned before, the entirety of an unsafe field in Rust is considered an unsafe block. Each dangerous action is re-expressed as a call to a private safe function.

The actual code is implemented within the private inner function:

```

// in src/memory.rs

/// Private function that is called by 'translate_addr'.
///
/// This function is safe to limit the scope of 'unsafe' because Rust treats
/// the whole body of unsafe functions as an unsafe block. This function must
/// only be reachable through 'unsafe fn' from outside of this module.
fn translate_addr_inner(addr: VirtAddr, physical_memory_offset: VirtAddr)
    -> Option<PhysAddr>
{
    use x86_64::structures::paging::page_table::FrameError;
    use x86_64::registers::control::Cr3;

    // read the active level 4 frame from the CR3 register
    let (level_4_table_frame, _) = Cr3::read();

    let table_indexes = [
        addr.p4_index(), addr.p3_index(), addr.p2_index(), addr.p1_index()
    ];
    let mut frame = level_4_table_frame;

    // traverse the multi-level page table
    for &index in &table_indexes {
        // convert the frame into a page table reference
        let virt = physical_memory_offset + frame.start_address().as_u64();
        let table_ptr: *const PageTable = virt.as_ptr();
        let table = unsafe {&*table_ptr};

        // read the page table entry and update 'frame'
        let entry = &table[index];
        frame = match entry.frame() {
            Ok(frame) => frame,
            Err(FrameError::FrameNotPresent) => return None,
            Err(FrameError::HugeFrame) => panic!("huge pages not supported"),
        };
    }
}

```

Figure 44: Translating address 2

We reread the level 4 frame from the CR3 register rather than reuse our `active_level_4` table method. Our reasoning behind doing this is that it makes implementing this prototype much easier. We will quickly devise a more satisfactory alternative.

Already built into the `VirtAddr` struct are routines for calculating indexes into the four-level page tables. In order to iteratively access the page tables, we keep the indexes in a tiny array. The last visited frame is stored outside the loop and used to derive the precise location. While iterating, the frame refers to frames in the page table; after the last iteration, having followed the level 1 entry, it refers to the mapped frame.

In the loop itself, we employ the `physical_memory_offset` once more to translate the frame into a page table reference. After getting the current page table entry, we can use the `PageTableEntry::frame` function to get the mapped frame. We will return `None` if the entry cannot

be found in any frames. At this time, we start to get worried if the entry maps to a really large page, say 2 MiB or 1 GiB.

Let's put our translated address function to the test:

```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    // new import
    use blog_os::memory::translate_addr;

    [-] // hello world and blog_os::init

    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);

    let addresses = [
        // the identity-mapped vga buffer page
        0xb8000,
        // some code page
        0x201008,
        // some stack page
        0x0100_0020_1a10,
        // virtual address mapped to physical address 0
        boot_info.physical_memory_offset,
    ];

    for &address in &addresses {
        let virt = VirtAddr::new(address);
        let phys = unsafe { translate_addr(virt, phys_mem_offset) };
        println!("{:?} -> {:?}", virt, phys);
    }

    [-] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

Figure 45: Translating address 3

The following is what comes out when we run it:

```
Hello World!
VirtAddr(0xb8000) -> Some(PhysAddr(0xb8000))
VirtAddr(0x201008) -> Some(PhysAddr(0x401008))
VirtAddr(0x010000201a10) -> Some(PhysAddr(0x279a10))
panicked at 'huge pages not supported', src/memory.rs:71:43
.....
```

Figure 46: Output of translating address

The identity-mapped address "0xb8000" maps to the same physical location as the original address. Depending on how the bootloader initially mapped our kernel, the code page and the stack page may map to different physical addresses. The last 12 bits are always preserved after translation,

which makes sense given that they represent the page offset and are not involved in the actual translation itself.

Adding the "physical memory offset" allows access to any physical address. Hence, the address translated from the "physical memory offset" should refer to physical address 0. Unfortunately, we can't finish the translation right now because the mapping makes good use of pages that are very large.

8.3 Utilizing OffsetPageTable

Operating system kernels frequently perform the work of translating virtual to physical addresses, and the `x86_64` crate provides an abstraction for this process. This method does not require us to add enormous page support to our own, so we will utilize it from here on out.

At its core, the abstraction is based on two things that all mapping functions for page tables have in common:

- The `Mapper` trait offers page-related operations regardless of the page size. Some of these operations are `map` to, which adds a new mapping to the page table, and `translate` page, which converts a page to a frame of the same size.
- Inheriting the `Translate` trait allows you to access functions like `translate_addr` and `translate` that are compatible with a wide range of page widths.

The traits do not supply any implementation, merely the interface definition. There are now three types in the `x86_64` crate that implement the attributes in accordance with their respective needs. Assuming that all of the physical memory is mapped to the virtual address space at some offset, the `OffsetPageTable` type is used. The `MappedPageTable` is somewhat more forgiving in its requirements, needing merely that each page table frame be mapped to the virtual address space at a computably addressable location. For good measure, you can use the `RecursivePageTable` type to get to page table frames in recursive page tables.

The `OffsetPageTable` data type is suitable for our needs since the bootloader places a map of all physical memory at the virtual address defined by the physical memory offset variable. We do this by adding an `init` function to the memory module:

```

use x86_64::structures::paging::OffsetPageTable;

/// Initialize a new OffsetPageTable.
///
/// This function is unsafe because the caller must guarantee that the
/// complete physical memory is mapped to virtual memory at the passed
/// `physical_memory_offset`. Also, this function must be only called once
/// to avoid aliasing `&mut` references (which is undefined behavior).
pub unsafe fn init(physical_memory_offset: VirtAddr) -> OffsetPageTable<'static>
{
    let level_4_table = active_level_4_table(physical_memory_offset);
    OffsetPageTable::new(level_4_table, physical_memory_offset)
}

// make private
unsafe fn active_level_4_table(physical_memory_offset: VirtAddr)
    -> &'static mut PageTable
{...}

```

Figure 47: Adding an init function to the memory module

As an output, the function creates a new 'static lifetime `OffsetPageTable` object using the provided `physical_memory_offset` as an input. So long as our kernel is active, the instance will continue to function correctly. As the first step in the function body, we make a call to the `active_level_4_table` to obtain a sharable pointer to the level 4 page table. This pointer is subsequently passed into the `OffsetPageTable::new` function. The new function takes the `physical_memory_offset` variable as its second parameter, specifying the virtual address at which the physical memory mapping begins.

Since several invocations of the `active_level_4_table` can result in aliased mutable references and hence undefined behavior, this method should henceforth only be used from the `init` function. Therefore, we remove the function's `pub` specifier, making it private.

Instead of writing our own `memory::translate_addr` function, we can now just utilize `Translate::translate_addr`. There are only a few lines in `kernel_main` that need to be updated:

```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    // new: different imports
    use blog_os::memory;
    use x86_64::{structures::paging::Translate, VirtAddr};

    [...] // hello world and blog_os::init

    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    // new: initialize a mapper
    let mapper = unsafe { memory::init(phys_mem_offset) };

    let addresses = [...]; // same as before

    for &address in &addresses {
        let virt = VirtAddr::new(address);
        // new: use the `mapper.translate_addr` method
        let phys = mapper.translate_addr(virt);
        println!("{:?} -> {:?}", virt, phys);
    }

    [...] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

Figure 48: Kernal main

For use of the `translate_addr` method, we must first import the `Translate` trait.

The translation results from the most recent version are the same as before, and the translation of very long pages now works as it should.

```
hello World!
VirtAddr(0xb8000) -> Some(PhysAddr(0xb8000))
VirtAddr(0x201008) -> Some(PhysAddr(0x401008))
VirtAddr(0x10000201a10) -> Some(PhysAddr(0x279a10))
VirtAddr(0x18000000000) -> Some(PhysAddr(0x0))
It did not crash!
```

Figure 49: Output of utilizing `OffsetPageTable`

The translations of `0xb8000` and the code and stack addresses remain unchanged, as predicted, when compared to our own translation function. We can also see that `physical_memory_offset = 0x0` shows how the virtual addresses map to the physical addresses.

The MappedPageTable type's translation feature makes developing support for very large pages unnecessary. Besides the functionalities of the current page, we may also use others, such as `map_to`, which we shall do in the following section.

`Memory::translate_addr` and `memory::translate_addr_inner` are now obsolete and can be removed.

8.4 Creating a new Mapping

To this point, we have only viewed the page tables for information. Let's make a new mapping for a page that hasn't been mapped before.

Since we'll be relying on the `map_to` method of the Mapper trait, it's important that we understand how it works before we proceed with the implementation. According to the documentation, it accepts four arguments, including the page to map, the frame to map it to, a set of flags for the page table entry, and a `frame_allocator`. To map the specified page, it may be necessary to generate additional page tables, and these in turn require spare memory, making the frame allocator an indispensable tool.

A `create_example_mapping` Function

The first thing we did was to make a new `create_example_mapping` function that converts a virtual page into the 0xb8000 physical frame of the VGA text buffer. That particular snapshot was selected so that we could quickly verify proper mapping creation: We can now test if writing to the remapped page results in output by simply writing to it.

The `create_example_mapping` function looks like this:

```

// in src/memory.rs

use x86_64::{
    PhysAddr,
    structures::paging::{Page, PhysFrame, Mapper, Size4KiB, FrameAllocator}
};

/// Creates an example mapping for the given page to frame `0xb8000`.
pub fn create_example_mapping(
    page: Page,
    mapper: &mut OffsetPageTable,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
) {
    use x86_64::structures::paging::PageTableFlags as Flags;

    let frame = PhysFrame::containing_address(PhysAddr::new(0xb8000));
    let flags = Flags::PRESENT | Flags::WRITABLE;

    let map_to_result = unsafe {
        // FIXME: this is not safe, we do it only for testing
        mapper.map_to(page, frame, flags, frame_allocator)
    };
    map_to_result.expect("map_to failed").flush();
}

```

Figure 50: Create_example_mapping function

The function also requires a `frame_allocator` and a mutable reference to an instance of `OffsetPageTable` in addition to the page to be mapped. Using the `impl Trait` syntax, the `frame_allocator` parameter is generic over all types that implement the `FrameAllocator` trait. Compatible with both small 4 KiB pages and massive 2 MiB/1 GiB pages, this characteristic is generic over the `PageSize` trait. The desired size of the mapping is only 4 KiB, thus we restrict the size of the mapping by changing the generic parameter to `Size4KiB`.

The caller of `map_to` must check to make sure the frame is not in use, making it an insecure operation. This is because undefined behavior may occur if two different `&mut` references link to the same physical memory address, which could happen if the same frame is mapped twice. We disregard the necessary condition by recycling the frame from the VGA text buffer, which has already been mapped. It's fine, though, because the `create_example_mapping` method is just a testing mechanism that will be taken out after this article. Putting a `FIXME` note on the line served as a constant reminder of the danger.

The map to method requires not just the page and the unused_frame, but also a reference to the frame_allocator, which will be discussed in a moment. To enable writing to the mapped page, we set the WRITABLE flag and the PRESENT flag. The PRESENT flag is necessary for all valid entries. Please refer to the Page Table Format portion of the preceding post for a complete list of all flags.

Since map_to's return value is a Result and the function is potentially error prone, it is always returned. This is merely some sample code, so we can afford to use anticipate to panic on any unexpected errors. With a successful call, the function returns a MapperFlush object, whose flush method can be used to remove the newly mapped page from the TLB. Similar to Result, this type makes advantage of the #[must_use] attribute to signal an error if it is not used.

8.4.1 A dummy FrameAllocator

In order to use create_example_mapping, a type that conforms to the FrameAllocator trait must be constructed. As said before, if map_to requires additional page tables, it is up to the characteristic to allocate the necessary frames for them.

Let's take the easy route and pretend we don't have to make any new page tables to begin with. An appropriate frame allocator would be one that always returns None in this scenario. In order to verify the accuracy of our mapping function, we design an EmptyFrameAllocator like the one below.

```
// in src/memory.rs

/// A FrameAllocator that always returns `None`.
pub struct EmptyFrameAllocator;

unsafe impl FrameAllocator<Size4KiB> for EmptyFrameAllocator {
    fn allocate_frame(&mut self) -> Option<PhysFrame> {
        None
    }
}
```

Figure 51: Create EmptyFrameAllocator

There is a risk involved in implementing the FrameAllocator because the implementer must ensure that only unused frames are returned. If two virtual pages are mapped to the same physical frame,

for instance, unclear behavior may result. This is not an issue because our EmptyFrameAllocator always returns None.

8.4.2 Choosing a Virtual Page

The create_example_mapping function can now take advantage of our newly minted basic frame allocator. On the other hand, the allocator always returns None, thus this will only work if no more page table frames are required to make the mapping. Let's look at an example to see if it clarifies the necessity for additional page table frames or not:

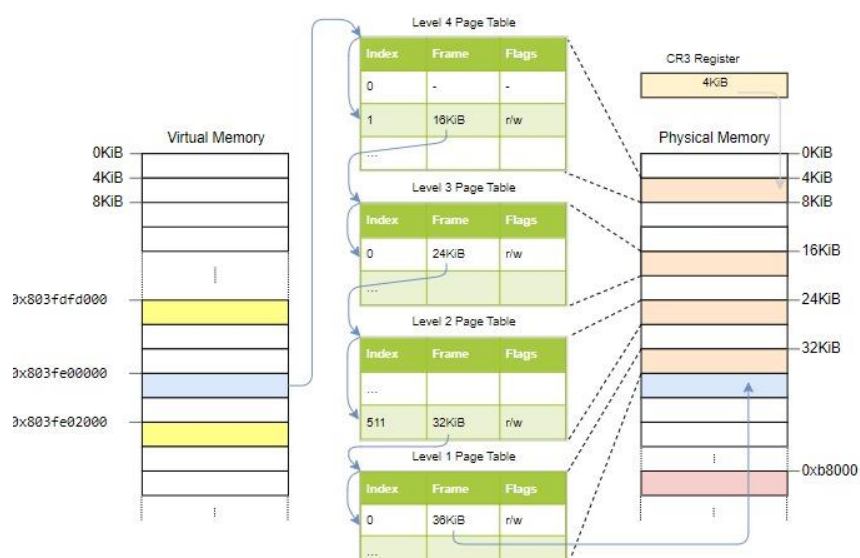


Figure 52: Page table example

Page tables are depicted in the middle of the diagram between the virtual address space on the left and the physical address space on the right. The dotted lines represent the locations of the physical memory frames where the page tables are kept. One mapped page, located at the blue-highlighted location of 0x803fe00000 in the virtual address space. The CPU uses a 4-level page table to locate the frame at location 36 KiB and translate the page there.

The VGA text buffer's actual physical frame is depicted in red on the picture as well. Our create_example_mapping method will be used to map a new virtual page to this frame. Considering that our EmptyFrameAllocator invariably returns None, we'd like to design the mapping in such a way that we never have to request extra frames from it. The answer will be found on the particular "virtual page" we end up picking for the mapping.

The figure depicts two potential locations in the virtual address space, both of which are highlighted in yellow. Address 0x803fd000 contains one page, which is 3 bytes before the mapped page. Indexes on the fourth and third levels of the table are identical to those on the blue page, while those on the second and first levels are different. Because of the unique index into the level 2 table, a different set of data from level 1 is being used for this particular page. If we used that page for our example mapping, we'd have to establish a new level 1 table which would use up a physical frame that was otherwise available. The second candidate page, located at 0x803fe000, however, avoids this issue by sharing the blue page's level 1 page table. Because of this, we have all the necessary page tables in place.

In conclusion, the challenge of making a new mapping is contingent on the specifics of the virtual page we're trying to map. In the simplest scenario, we only need to add a single entry to the page's level 1 table. In the worst-case scenario, the page is located in a part of memory for which no existing level 3 page table exists; in this situation, we must first generate new level 3, level 2, and level 1 page tables.

We need to select a page for which all page tables already exist before we can use our `create_example_mapping` function using the `EmptyFrameAllocator`. The fact that the bootloader stores its own code in the first megabyte of the virtual address space allows us to quickly locate such a page. This indicates that all pages in this area have a valid level 1 table. Consequently, any free page in this area of memory, such as the page at address 0, is available for use in our demonstration mapping. By leaving it unmapped, the bootloader ensures that any attempt to access a null pointer will result in a page fault.

8.4.3 Creating the Mapping

Now that we have everything we need to run `create_example_mapping`, let's alter `kernel_main` to map the page at virtual address 0. We should be able to write to the screen through the VGA text buffer after we have mapped the page to the frame. These are the specifics of the implementation:

```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    use blog_os::memory;
    use x86_64::{structures::paging::Page, VirtAddr}; // new import

    [-] // hello world and blog_os::init

    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = memory::EmptyFrameAllocator;

    // map an unused page
    let page = Page::containing_address(VirtAddr::new(0));
    memory::create_example_mapping(page, &mut mapper, &mut frame_allocator);

    // write the string 'New!' to the screen through the new mapping
    let page_ptr: *mut u64 = page.start_address().as_mut_ptr();
    unsafe { page_ptr.offset(400).write_volatile(0x_f021_f077_f065_f04e)};

    [-] // test_main(), "it did not crash" printing, and hlt_loop()
}
```

Figure 53: create_example_mapping

We begin by using `create_example_mapping` with a mutable reference to the `mapper` and the `frame_allocator` instances in order to construct the mapping for the page at address 0. This should allow the output of any text sent to the VGA text buffer frame on the screen.

After that, we cast the page as a raw pointer and store the value at offset 400. Because the next `println` will cause the top line of the VGA buffer to be relocated directly off the screen, we avoid writing to the beginning of the page. To depict the string "New!" on a white backdrop, we use the hexadecimal code `0x_f021_f077_f065_f04e`. We use the `write_volatile` method because, as we saw in the "VGA Text Mode" topic, writing to the VGA buffer must be treated as volatile.

The following is the result of running it in QEMU:

```
Hello World!
It did nit crash!
.....
```

Figure 54: Output of running in QEMU

Since we were able to build a new mapping in the page tables, we can see "New!" displayed on the screen after writing to page 0.

The existence of the level 1 table in charge of the page at address 0 was essential to the success of creating that mapping. Since the `map_to` method allocates frames using the `EmptyFrameAllocator` to generate new page tables, it fails when we attempt to map a page for which no level 1 table yet exists. This is evident when we try to map page `0xdeadbeaf000` instead of 0:

```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    [-]
    let page = Page::containing_address(VirtAddr::new(0xdeadbeaf000));
    [-]
}
```

Figure 55: Map page 0xdeadbeaf000 instead of 0

The following error message appears when we try to start it:

```
panicked at 'map_to failed: FrameAllocationFailed', /.../result.rs:999:5
```

Figure 56: Error message

We need a suitable `FrameAllocator` to map pages without a level 1 page table. How can we find out which frames are currently being wasted and how much actual memory is accessible?

8.5 Allocating Frames

We need a good frame allocator to generate new page tables. The `memory_map` included in the `BootInfo` struct given along by the bootloader is used for this purpose.

```

// in src/memory.rs

use bootloader::bootinfo::MemoryMap;

/// A FrameAllocator that returns usable frames from the bootloader's memory map.
pub struct BootInfoFrameAllocator {
    memory_map: &'static MemoryMap,
    next: usize,
}

impl BootInfoFrameAllocator {
    /// Create a FrameAllocator from the passed memory map.
    ///
    /// This function is unsafe because the caller must guarantee that the passed
    /// memory map is valid. The main requirement is that all frames that are marked
    /// as `USABLE` in it are really unused.
    pub unsafe fn init(memory_map: &'static MemoryMap) -> Self {
        BootInfoFrameAllocator {
            memory_map,
            next: 0,
        }
    }
}

```

Figure 57: Allocating frames 1

The struct's two members are the next field, which stores the frame number that the allocator should return, and a static reference to the memory map given by the bootloader.

The memory map is supplied by the BIOS/UEFI firmware, as we discussed in the Boot Information section. It's only accessible very early in boot, so the bootloader has already called the appropriate routines before we even need them. Each section of memory is represented by a `MemoryRegion` struct in the memory map, which includes the section's beginning address, its size, and its status (used, reserved, etc.).

The `init` function loads the specified memory map into a `BootInfoFrameAllocator`. To prevent accidentally returning the same frame twice, the next field is set to 0 and incremented with each allocation. In order to ensure that the caller provides sufficient guarantees, our `init` method must be `unsafe` since we have no idea if the memory map's accessible frames have already been allocated.

8.5.1 A `usable_frames` Method

The memory map is transformed into an iterator of available frames using an auxiliary method that we create before implementing the `FrameAllocator` trait.


```

// in src/memory.rs

use bootloader::bootinfo::MemoryRegionType;

impl BootInfoFrameAllocator {
    /// Returns an iterator over the usable frames specified in the memory map.
    fn usable_frames(&self) -> impl Iterator<Item = PhysFrame> {
        // get usable regions from memory map
        let regions = self.memory_map.iter();
        let usable_regions = regions
            .filter(|r| r.region_type == MemoryRegionType::Usable);
        // map each region to its address range
        let addr_ranges = usable_regions
            .map(|r| r.range.start_addr()..r.range.end_addr());
        // transform to an iterator of frame start addresses
        let frame_addresses = addr_ranges.flat_map(|r| r.step_by(4096));
        // create `PhysFrame` types from the start addresses
        frame_addresses.map(|addr| PhysFrame::containing_address(PhysAddr::new(addr)))
    }
}

```

Figure 58: FrameAllocator trait

- To begin, an iterator of MemoryRegions is created from the memory map via a call to the iter method.
- Then, we apply the filter technique to avoid any off-limits areas. Frames that are used by our kernel (code, data, or stack) or to store the boot information are already tagged as InUse or equivalent by the time the bootloader updates the memory map for all the mappings it produces. This ensures that the Usable frames are not being used by another application.
- The iterator of memory regions is then converted to an iterator of address ranges with the help of the map combinator and Rust's range syntax.
- As a next step, we use flat map to convert the ranges of addresses into an iterator of frame start addresses, selecting one at random every 4096 iterations with step by. The beginning of each frame's address can be calculated using the page size of 4096 bytes (= 4 KiB). Because the bootloader page-aligns all addressable memory regions, we don't need to worry about rounding or alignment. An IteratorItem = u64> is obtained rather than an IteratorItem = IteratorItem = u64>> when flat map is used in place of map.
- Then, an Iterator <Item = PhysFrame> is built using the converted start addresses.

The impl Trait feature is used for the function's return type. Then, without naming the concrete return type, we can say that we return some type that implements the Iterator trait with item type PhysFrame. As the concrete type relies on unnameable closure types, this is crucial information.

8.5.2 Implementing the FrameAllocator Trait

The FrameAllocator trait can now be used:

```
// in src/memory.rs

unsafe impl FrameAllocator<Size4KiB> for BootInfoFrameAllocator {
    fn allocate_frame(&mut self) -> Option<PhysFrame> {
        let frame = self.usable_frames().nth(self.next);
        self.next += 1;
        frame
    }
}
```

Figure 59: FrameAllocator trait 2

To begin, we use the memory map's usable frames method to obtain an iterator of useable frames. After that, we use `Iterator::nth` to retrieve the frame at index `self.next` (skipping the frames at `(self.next - 1)`). To ensure that the next frame is returned on the next request, we increment `self.next` by one before returning that frame.

Unfortunately, every time an allocation is made, the `usable_frame` allocator must be recreated, making this design less than ideal. It is preferable to keep the iterator in a dedicated field of the struct. Then we could simply call `next` after each allocation, eliminating the necessity for the `nth` procedure. The issue with this method is that `impl Trait` types cannot be stored in struct fields at the present time. When completely developed, named existential types have a chance of making this a reality.

8.5.3 Using the BootInfoFrameAllocator

Rather than passing an `EmptyFrameAllocator` object in `kernel_main`, we may now pass a `BootInfoFrameAllocator` instance.

```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    use blog_os::memory::BootInfoFrameAllocator;
    [-]
    let mut frame_allocator = unsafe {
        BootInfoFrameAllocator::init(&boot_info.memory_map)
    };
    [-]
}
```

Figure 60: BootInfoFrameAllocator

The mapping is complete thanks to the boot info frame allocator, and the black-and-white "New!" appears once more. Without the user's knowledge, the `map_to` method generates the necessary backend page tables like follows:

- Allocate a free frame using the provided frame allocator.
- Set the frame size to 0 to make a blank table on the next page.
- Transfer the data from the more advanced table into that context.
- Move on to the succeeding tier of tables.

The `create_example_mapping` function we've written is merely an example; we can now write functions to generate new mappings for any given page. In upcoming articles, this information will be critical for memory allocation and multithreading.

For the same reasons stated above, we must now remove the `create_example_mapping` method entirely should it be unintentionally called, resulting in ambiguous behavior.

10. Heap Allocation

Allocating resources to a heap might be somewhat pricey. Each allocation (and deallocation) often includes gaining a global lock, doing some non-trivial data structure manipulation, and sometimes executing a system call; the specifics depend on the allocator being used. It's not always the case that a smaller allocation will cost less than a larger one. Realizing which Rust data structures and activities result in allocations is important because doing without them can have a significant impact on performance.

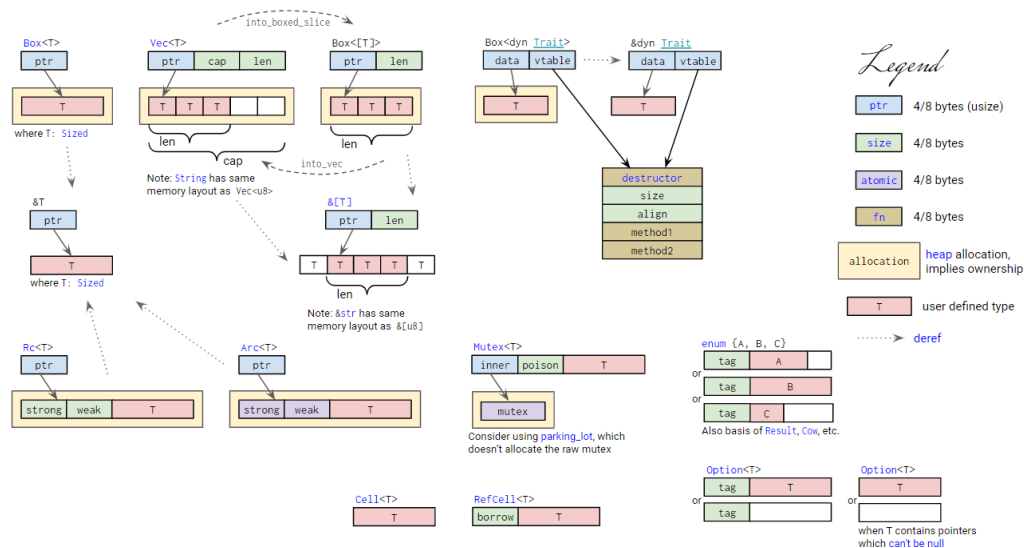


Figure 61: Heap allocation

9.1 Box

The most elementary heap-allocated data type is `Box`. Heap-allocated `T` values are referred to as `Box<T>` values. When trying to reduce the size of a type, it may be worthwhile to box one or more fields in a struct or enum. But apart from that, `Box` is simple and has limited optimization potential.

9.2 Rc/Arc

A value in the heap is accompanied by two reference counts in `Rc/Arc`, making them identical to `Box`. They facilitate value sharing, which can cut down on unnecessary memory consumption. You can increase allocation rates by allocating values to the heap that wouldn't normally be heap-allocated if they are used for variables that don't get shared very often.

When used on a Rc/Arc value, invoking clone does not trigger an allocation like it does with Box. The only thing it does is add one to the total number of references.

9.3 Vec

Vec is a heap-allocated data type with lots of room for optimizing memory utilization and decreasing allocation frequency. To do so successfully, one needs to get familiar with the storage practices governing its individual components. A Vec is composed of three words: a length or capacity measurement plus an index. If both the capacity and the element size are nonzero, then the pointer will point to heap-allocated memory; otherwise, it will not.

Even if the Vec itself is not heap-allocated, its elements (if present and of non-zero size) always are. The memory utilized to store non-zero-sized items may be greater than necessary, providing room for even more pieces in the future. Currently stored objects make up the "length," whereas the maximum number of items that can be stored without rearranging is the "capacity." When the vector grows too large for its current heap allocation, its elements will be copied into a new, larger heap allocation, and the old allocation will be freed up.

9.3.1 Vec growth

No heap allocation is needed when a new, empty Vec is generated using standard methods (`vec![]` or `Vec::new` or `Vec::default`), when the length and capacity are both zero. The Vec will periodically reallocate itself if you keep piling elements onto its tail. The method of expansion is left unstated, although at present it appears to be a quasi-doubling scheme that yields the following numbers of nodes: 0, 4, 8, 16, 32, 64, etc. (It bypasses allocations by going straight from 0 to 4, rather than via 1 and 2, because of this.) It's true that reallocations happen less frequently as a vector expands, but it's also true that the amount of surplus capacity that may be wasted grows at an exponential rate.

While this approach is normal for growable data structures and makes sense in most circumstances, it is often preferable to predict the likely length of a vector in advance. Using `eprintln!` to print the vector length at a hot vector allocation site (such as a hot `Vec::push` call) and then post-processing (such as using counts) to establish the length distribution is worthwhile. Allocation site

optimization will look different depending on factors like the quantity and length distribution of vectors.

9.3.2 Short Vecs

The `SmallVec` type, found in the `smallvec` crate, is useful if you need to store a large number of very short vectors. Drop-in replacement for `Vec`, `SmallVec[T; N]>` uses in-memory storage for up to `N` elements, moving to a heap allocation for larger vectors. (It is also important to remember to use `smallvec![]` literals in favor of `vec![]` literals.)

Even though proper use of `SmallVec` consistently lowers the allocation rate, it does not ensure better performance. For typical operations, it is slightly slower than `Vec` due to the constant need to determine if the components are heap-allocated. The copying of `SmallVec` values is further slowed down if `N` is large or if `T` is large, because the `SmallVec[T; N]>` itself can be larger than `VecT>`. The success of any optimization should be verified through comparison to industry standards. The `arrayvec` crate's `ArrayVec` is preferable to `SmallVec` if you have a large number of short vectors and can exactly determine their maximum length. Since it is not necessary, it is slightly faster than using heap allocation as a fallback.

9.3.3 Longer Vecs

`Vec::with capacity`, `Vec::reserve`, and `Vec::reserve exact` allow you to reserve a particular capacity of a vector given the minimum and exact sizes, respectively. For capacities of 4, 8, 16, and 32, these routines can automatically set aside enough space for a vector to hold at least 20 elements, instead of requiring the user to do four allocations (which is what happens by default when pushing things one at a time). Also, the aforementioned operations allow you to avoid needlessly allocating extra space while working with vectors when you know their maximum length. `Vec::shrink to fit` is another option for reducing unnecessary space utilization. It may result in a reallocation.

9.4 String

Bytes allocated to the heap make up a "String." A string is extremely similar to `Vec<u8>` in its representation and operations. String methods like "String::with capacity" are analogous to the growth and capacity-related `Vec` methods that are more common in the programming world. It's

easy to see the parallels between the SmallVec type and the SmallString type from the "smallstr" crate.

The "smartstring" crate's String type is a drop-in replacement for String that eliminates the need for heap allocations for strings shorter than three words. That includes all strings of 23 or fewer ASCII characters, and on 64-bit platforms, strings that are less than 24 bytes in length. Take into account that there is an allocation performed by the format! macro because it generates a String. You can avoid allocating this string if you use a string literal instead of a format! call. In some cases, the "lazy format" crate and/or the "std::format args" package may be useful here.

Tables for hashing

The HashSet and HashMap structures are two types of hash tables. Allocating keys and values to a single continuous heap allocation that is reallocated as needed makes their representation and operations similar to Vec. For example, HashSet::with_capacity is the HashSet/HashMap analogue to the Vec method's grow capacity.

Cow

Borrowed data, like a &str, may be primarily read-only yet occasionally require updating. It would be inefficient to constantly duplicate the data. The Cow data type, which supports both borrowing and ownership, can be used instead for "clone-on-write" semantics.

To begin with a borrowed value x, you should typically Cow::Borrowed(x) wrap it. For non-mutating operations, Cow provides direct access thanks to its implementation of Deref.

Cow::to_mut will clone an owned value if necessary to gain a mutable reference to it if mutation is requested.

clone

If a value contains heap-allocated memory, calling clone on it usually results in more allocations.

to_owned

Many common types have implementations of ToOwned::to_owned. It typically produces heap

allocations because it converts borrowed data into owned data, via cloning. For instance, you can use it to turn a `&str` into a `String`.

11. Allocator Designs

Memory allocation is the job of an allocator. Memory that is not used must be returned after alloc operations, and freed memory must be tracked after being released by dealloc. Memory that is already in use must never be made available to other threads, as this could lead to unpredictable results.

There are many more objectives besides "correctness" that can be pursued during the design process. Memory fragmentation and efficiency in allocation are two examples. In addition, it needs to be scalable to an arbitrary number of processors and perform well with applications running in parallel. By making the memory structure work better with the CPU caches, you can improve cache locality and stop false sharing, which are two of the biggest performance slowdowns.

10.1 Bump Allocator

A bump allocator is the simplest form of allocator architecture (also known as a stack allocator). It employs a linear method of memory allocation and stores simply the total number of allocations and the number of bytes that have been allocated. Due to the fact that it can only free up all memory at once, it is only used in very rare cases.

With a bump allocator, memory is allocated in a linear fashion by incrementing (or "bumping") a "next" variable that indicates the beginning of the free space. The initial value of "next" is the same as the heap's beginning address. Since the "next" pointer always advances in one direction, it ensures that the same chunk of memory is never given away twice by incrementing by the allocation size with each allocation. When the heap is full, the "next" allocation fails with an "out-of-memory" error because no more memory can be given.

An allocation counter is commonly used in implementations of a bump allocator, with increments of 1 for alloc calls and decrements of 1 for dealloc calls. All allocations on the heap have been deallocated when the allocation counter approaches zero. To make all of the heap's memory available for allocation once more, simply change the "next" pointer back to the heap's initial address in this situation.

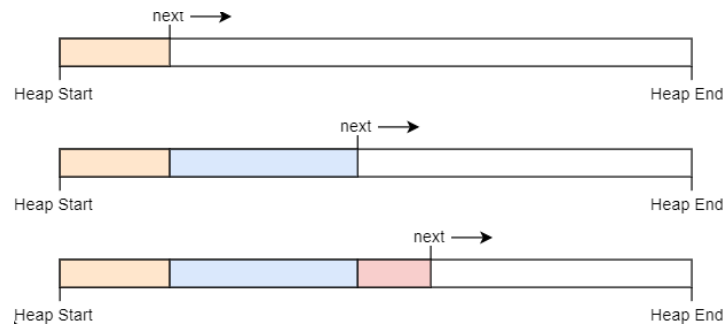


Figure 62: Bump Allocator

Since the "next" pointer may only advance forward, it ensures that no two consecutive memory locations are ever distributed. If the heap is full, allocating any more memory will fail with an out-of-memory error on the "next" allocation.

Allocation counters are commonly used in implementations of bump allocators, with the counter being incremented by 1 on every alloc call and decremented by 1 on every dealloc call. At the point where the allocation counter reads 0, all heap allocations have been deallocated. To make all of the heap's memory available for allocations once more, we can simply move the "next" reference back to the heap's initial address.

10.2 Linked List Allocator

When implementing allocators, it is common practice to use the free memory regions themselves as a backing store. The regions are still associated with a virtual address and supported by a physical frame, but the data stored there is no longer required. If we store information about each freed area in the area itself, we don't need any more memory to keep track of an infinite number of freed areas. Building a single linked list with each node being a freed memory region is the most common way to implement released memory:

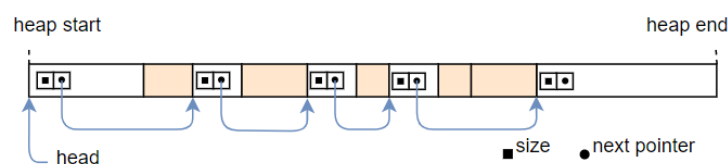


Figure 63: Linked List Allocator

The list node stores two pieces of information: the size of the memory region and a pointer to the next available memory space. To keep track of all unused regions, regardless of their number, we

simply need a pointer to the first unused area (called head) with this strategy. A free list is a common name for the ensuing data structure. The linked list allocator crate employs this method, as you could have guessed from the name. Because of the way they share out resources, these allocators are sometimes called "pool allocators."

10.3 Fixed-Size Block Allocator

The concept of a fixed-size block allocator is as follows: We establish a minimal number of block sizes and round up each allocation to the next block size rather than allocating exactly the amount of RAM that is requested. The allocation of 4 bytes would yield a 16-byte block, 48 bytes would yield a 64-byte block, and 128 bytes would yield a 512-byte block, respectively, assuming the corresponding block sizes. To keep track of the unused memory, we build a linked list in the unused memory just like the linked list allocator does. Instead of utilizing a single list where blocks of varying sizes are aggregated together, we generate individual lists for each size category. Then, only blocks of that size would be stored in each list. Three distinct linked lists in memory would be created if the block sizes were 16, 64, and 512, respectively.

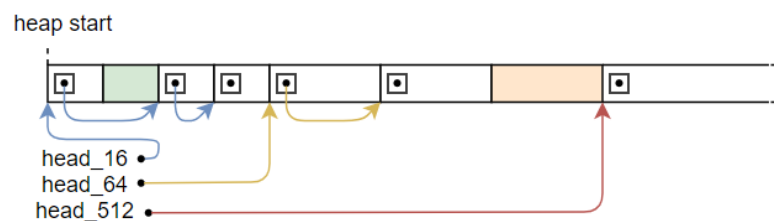


Figure 64: Fixed-Size Block Allocator

Instead of a single head pointer, we have three—head 16, head 64, and head 512—that each refer to the beginning of the first free block of their respective sizes. The size of each given node in a list is always the same. The head 16-pointer's linked list, for instance, consists entirely of 16-byte chunks. Because the size of the list is indicated by the node's reference to the head pointer, we can skip keeping size information in each node.

In a list, all of the items can be considered for an allocation request with the same weight because they are of the same size. Therefore, the following procedures for performing an allocation will work quite well:

Round the requested allocation up to the next even block size. As an illustration, if 12 bytes were requested as part of an allocation, we might opt for a block size of 16. In order to get the list's head pointer, we can use something like head 16 for a block size of 16. Take the list and return it to you without the first block.

In particular, we no longer have to traverse the entire list in order to return the first entry. So, it's clear that this allocator is a significant improvement over the linked-list allocator.

12. Async/Await

To avoid blocking and allow other code to continue running while waiting for an action to finish, "async" and ".await" are specialized parts of Rust syntax.

For the most part, you'll be using the "async: async fn" or "async" blocks to accomplish your asynchronous goals. All of them produce a value that has the "Future" property:

```
1
2 // `foo()` returns a type that implements `Future<Output = u8>`.
3 // `foo().await` will result in a value of type `u8`.
4 async fn foo() -> u8 { 2 }
5
6 fn bar() -> impl Future<Output = u8> {
7     // This `async` block results in a type that implements
8     // `Future<Output = u8>`.
9     async {
10         let x: u8 = foo().await;
11         x + 2
12     }
13 }
```

Figure 65: async/.await implementation 1

There is no activity in "async" bodies or other futures until they are run. The ".await" operator is the most frequent method of executing a "Future." Whenever a "Future" object has the ".await" method invoked on it, it will attempt to execute the object until it finishes. In the event that the "Future" is blocked, it will release control of the current thread. As soon as more progress is possible, the executor will pick up the "Future" and continue running; this will cause the ".await" to complete.

11.1 async Lifetimes

"async fn"s that accept references or other non-"static" parameters return a "Future" whose lifetime is limited by the lifespan of the arguments, as opposed to the standard function's infinite lifetime.

```

1 // This function:
2 async fn foo(x: &u8) -> u8 { *x }
3
4 // Is equivalent to this function:
5 fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output = u8> + 'a {
6     async move { *x }
7 }

```

Figure 66: async Lifetimes 1

Because of this, the future that is returned by an "async fn" must be ".awaited" while its non-"static" arguments are still active. This isn't a problem when ".awaiting" the future right after invoking the function (like in "foo(&x).await"), as is commonly done. On the other hand, this may become a problem if the future were to be saved or passed on to another process or thread.

An often-used workaround for making an "async fn" with references-as-arguments into a "static" future is to enclose the call to the "async fn" within an async block, along with the arguments.

```

1 fn bad() -> impl Future<Output = u8> {
2     let x = 5;
3     borrow_x(&x) // ERROR: `x` does not live long enough
4 }
5
6 fn good() -> impl Future<Output = u8> {
7     async {
8         let x = 5;
9         borrow_x(&x).await
10     }
11 }

```

Figure 67: async Lifetimes 2

By moving the argument into the "async" block, we extend its lifetime to match that of the "Future" returned from the call to "good".

11.2 async move

The move keyword can be used in "async" blocks and closures just as it can in regular closures. To ensure its own survival beyond the present scope, an "async move" block will make a copy of the variables it uses and will not enable their use in any other blocks of code.

```

1  /// `async` block:
2  ///
3  /// Multiple different `async` blocks can access the same local variable
4  /// so long as they're executed within the variable's scope
5  async fn blocks() {
6      let my_string = "foo".to_string();
7
8      let future_one = async {
9          // ...
10         println!("{my_string}");
11     };
12
13     let future_two = async {
14         // ...
15         println!("{my_string}");
16     };
17
18     // Run both futures to completion, printing "foo" twice:
19     let ((), ()) = futures::join!(future_one, future_two);
20 }
21
22 /// `async move` block:
23 ///
24 /// Only one `async move` block can access the same captured variable, since
25 /// captures are moved into the `Future` generated by the `async move` block.
26 /// However, this allows the `Future` to outlive the original scope of the
27 /// variable:
28 fn move_block() -> impl Future<Output = ()> {
29     let my_string = "foo".to_string();
30     async move {
31         // ...
32         println!("{my_string}");
33     }
34 }

```

Figure 68: *async move*

11.3 .awaiting on a Multithreaded Executor

It's important to remember that in a multithreaded "Future" executor, a "Future" may transition between threads, which means that variables used in "async" bodies may need to be passed to a different thread.

This also applies to references to types that don't implement the "Sync" trait, so you shouldn't use "Rc," "&RefCell," or any other type that doesn't implement the "Send" trait.

(With the proviso that they are out of scope during a ".await" invocation, these types can be used.)

To the same extent, it's not a good idea to hold a traditional non-futures-aware lock across a ".await," as this can cause the threadpool to deadlock: one task could acquire a lock, ".await," and yield to the executor, allowing another task to attempt to acquire the lock. The solution is to use the "Mutex" function from futures::lock instead of the one from std::sync. Holding a regular, non-futures-aware lock over a ".await" is a bad idea for the same reason that it's bad for the threadpool to freeze up when two tasks try to acquire the same lock at the same time. Instead of using std::sync, you should use "Mutex" from Futures::lock because it stops this problem from happening.

13. Conclusion

Rust is a programming language that boasts excellent performance, excellent tools, and an active community behind it that is constantly striving to enhance the language. A Freestanding Rust Binary, A Minimal Rust Kernel, VGA Text Mode, Testing, CPU Exceptions, Hardware Interrupts, Introduction to Paging, Paging Implementation, Heap Allocation, Allocator Designs, and Async/Await were discussed here. Rust is a good option to consider, particularly if you require a solution that places a larger emphasis on safety than C or C++ but do not want to sacrifice speed in the process. The Rust community is highly active, and on a regular basis, new versions of Rust and technology designed to improve it are made available. The software developer community will continue to value Rust as a result of its safety, effectiveness, and efficiency (i.e., its ability to help programmers produce more performant code in a shorter amount of time). Rust is likely to stay popular over the next few years because it can create secure systems and has a good reputation for doing so.

References

- 1) *Rust Programming Language* n.d., Rust Programming Language, viewed 23 October 2022, <<https://www.rust-lang.org/>>.
- 2) *Rust Basics - GeeksforGeeks* 2021, GeeksforGeeks, viewed 23 October 2022, <<https://www.geeksforgeeks.org/rust-basics/>>.
- 3) Oppermann, P 2022, *Writing an OS in Rust*, Writing an OS in Rust, viewed 23 October 2022, <<https://os.phil-opp.com/>>.
- 4) *The Rust Programming Language - The Rust Programming Language* n.d., The Rust Programming Language - The Rust Programming Language, viewed 23 October 2022, <<https://doc.rust-lang.org/book/>>.
- 5) *Getting started - Rust Programming Language* n.d., Getting started - Rust Programming Language, viewed 23 October 2022, <<https://www.rust-lang.org/learn/get-started>>.