The guide to ChronOS

Betty Liu u2061245

February 11, 2024

Contents

Chapter 1

Introduction to ChronOS

All code available here `https://github.com/acyanbird/chronos_labs/` Please see different branch for each stage.

Writing a lab assignment to create an operating system using Rust as the programming language:

Lab Assignment: Building an Operating System with Rust

Objective: In this lab, you will learn to create a simple operating system using the Rust programming language. Operating systems are complex pieces of software that manage hardware resources and provide services to other software applications. This lab will introduce you to the basics of operating system development, focusing on the foundational components.

.

Chapter 2

Lab 0 - Introduction to Rust Programming languoage

2.1 Objectives

Since this project uses Rust as the programming language, and there are no specific university courses for it, this lab will introduce you to some fundamental syntax to help you get started. However, to master this language, you'll need to continue learning as you go. Towards the end, we'll share extra resources for self-study.

2.2 Install Rust

Rustup is used to install and managed Rust. You can check if your machine is already install Rust by typing rustc --version in your console.

If not, for installation on Unix-like machine (e.g. MacOS, Linux) input this in terminal

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

For windows users install

```
https://static.rust-lang.org/rustup/dist/i686-pc-windows-gnu/
rustup-init.exe
```

2.2.1 Integrated Development Environment

I highly recommend using an IDE for development. Currently, there are not many IDEs that support Rust. Here, I recommend Visual Studio Code + rust-analyzer entension or RustRover.

2.3 Hello World

We will use cargo to create the basic framework for the project. Cargo is Rust's build system and package manager and it should be installed by rustup. You could check it by

```
cargo --version
```

Create project by

```
cargo new hello
cd hello
```

It will also generate an empty git repository for version control, and a cargo.toml that procvie project basic information and dependency. We could ignore it right now. The file structure is like:

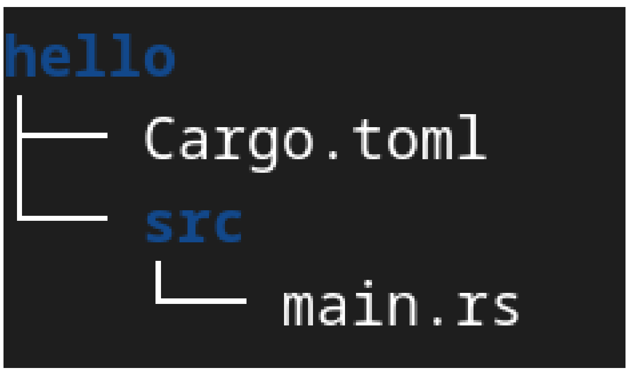


Figure 2.1: project structure

Cargo.toml: This is the configuration file for your Rust project. It contains metadata about the project, such as its name, version, dependencies, and other settings.

src/directory: This directory is where you put your source code files. It contains your project's main code. You will often have one or more Rust source files (.rs) in this directory.

main.rs: This is the primary entry point for your Rust application. It typically contains the main function, which is the starting point of your program.

The main.rs created by cargo is a simple program that would print Hello, world! It is similar to C. To run project use **cargo run** command in terminal.

```
fn main() {
    println!("Hello, world!");
}
```

2.4 Variables

Variables:

- Variables in Rust are declared using the let keyword.
- By default, variables in Rust are immutable, which means their values cannot be changed once assigned. To make a variable mutable, you use the mut keyword.
- You can reassign values to mutable variables, but their types must remain the same.

```
// Declare an immutable integer variable
let x = 10;
// Declare a mutable integer variable
let mut y = 20;
// Reassign a value to the mutable variable
y = 30;
```

Constants:

- Constants in Rust are declared using the const keyword.
- Constants must have an explicitly defined type and must have a fixed, compile-time determined value.
- Conventionally, constants are named using all uppercase letters and underscores to separate words.

```
// Declare an integer constant
const MAX_VALUE: i32 = 100;
// Declare a string constant
const GREETING: &str = "Hello, Rust!";
```

2.5 Attributes

Attributes in Rust are metadata applied to modules, crates, functions, structs, or other items. They can instruct the compiler to perform specific tasks or apply certain properties to the item they annotate. Attributes can be divided into two main categories: Inner Attributes and Outer Attributes.

- Outer Attributes (#[outer_attribute]): Applied to the item that follows them. They are used to set attributes or give instructions related to the item directly below them.
- Inner Attributes (#![inner_attribute]): Applied to the item they are contained within. They are often found at the beginning of source files or modules to configure or set options for the scope they reside in.

2.6 Unsafe Rust

Tell the compiler I know what I'm doing!

The unsafe keyword allows you to bypass the language's usual safety checks and guarantees. It's used when you need to perform operations that the Rust compiler can't prove to be safe at compile-time, such as accessing raw pointers, dereferencing them, or making changes to mutable static variables. It's a way to tell the Rust compiler that you, the programmer, will ensure the safety of the code within the unsafe block.

2.6.1 External Code

2.7 Module

2.8 Structs

A struct, short for structure, is a custom data type that lets you package together related data under a single name. We use it to making code more organized, readable, and maintainable. It's similar to structs in C. In our case

```
struct VGAChar {
    ascii: u8,
    color: u8,
}
```

VGAChar is a struct with 2 fields, ascii and color both with data type u8. We can create instance by

```
let character = VGAChar {
        ascii: b'A', // ASCII code for 'A'
        color: 0x04; // black background, red foreground
    };
```

Using character.ascii and character.color access the ascii and color fields of the character instance, respectively. You can use these to read (and, if mutable, modify) the data stored in an instance of a struct.

2.9 Impl

TODO

It is a keyword used to implement functionality for a particular type, such as a struct or enum. It allows you to define methods, associated functions, and trait implementations for the specified type. The impl keyword can be used indefinitely, but typically, organizing related functionality into one or a few impl blocks is a clearer and more maintainable practice.

2.10 Function

2.11 Match

Chapter 3

Lab 1 - Getting started

These lab is running and tested on Linux (Debian) only for now.

3.1 Expected Outcome

In this lab, we'll create a Rust program entirely from scratch, free from any reliance on a host operating system. Our goal is to develop a minimal 64-bit kernel that can display text using VGA through QEMU. Below is the expected output.

3.2 Preparation

- QEMU

To run the experiment's output, we need to use QEMU. Here, we won't list the installation steps for QEMU on various operating systems. Please visit the official website at `https://www.qemu.org/download/` to download the appropriate installer and follow the on-screen instructions for installation.

- Nightly Rust

Rust offers various versions, but for operating system development, we require certain experimental features not available in the stable version. Thus, we can't use the stable version. To install Nightly Rust, simply enter the following command in your terminal.

```
rustup update nightly
```

- bootimage

This tool assists us in generating files for the virtual machine (QEMU). To install it, use the following command in your terminal using Cargo.

```
cargo install bootimage
```

- llvm-tools-preview

The llvm-tools-preview is a dependency for the bootimage tool. You can install it using command

```
rustup component add llvm-tools-preview
```

3.3 Task 1 - Standalone Rust Binary

3.3.1 Introduction

When we create a Rust program, similar to Lab 0, it usually relies on an existing operating system. Rust comes with a standard library that depends on the features of that operating system.

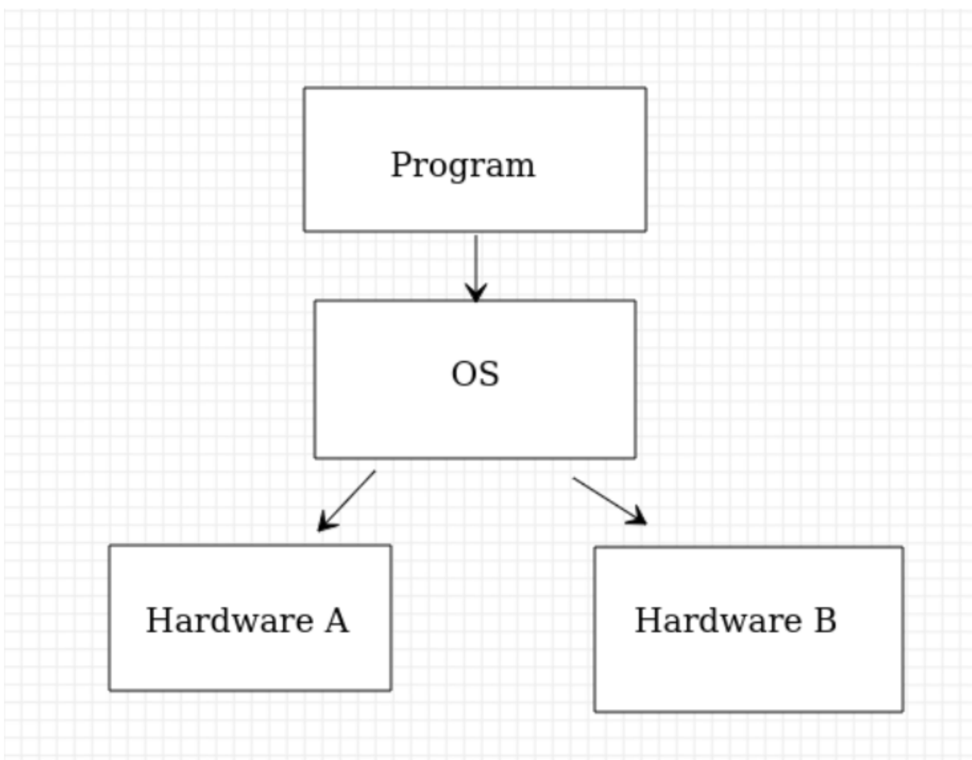


Figure 3.1: Common Rust Program

But since our goal is to build an operating system from the ground up, we can't rely on the existing one. So, we disable the standard library using no_std, and this lets us work directly with hardware.

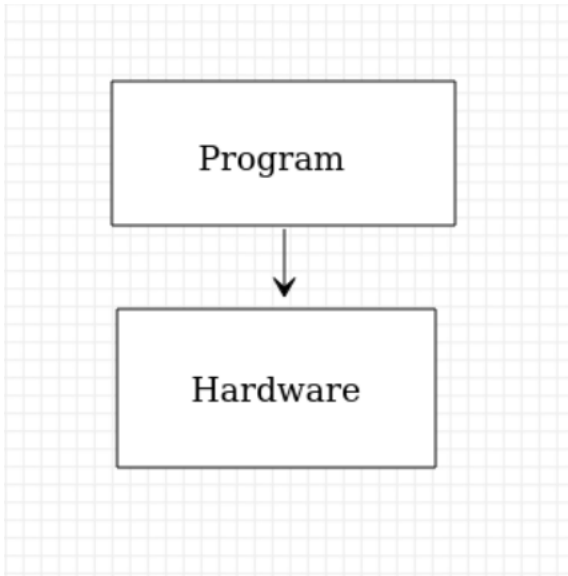


Figure 3.2: Standalone Rust

See branch lab-1-1 for the code.

3.3.2 Implementation

Step 1: Setup a New Rust Project

Open a terminal and create a new Rust project by running

```
cargo new chronos_lab --bin --edition 2018
```

This creates a new binary project named chronos_lab. You could use your own name. Change into the project directory with

```
cd <your project name>
```

Step 2: Editing Cargo.toml

Open the Cargo.toml file in your project's root directory.

In the initial Cargo.toml file, the [package] section includes predefined name, version, and edition information. You can now leave them unchanged.

Add configurations for development and release profiles to change the panic strategy to abort, which disables stack unwinding during a panic. Add these lines at the end of the Cargo.toml file:

```
[profile.dev]
panic = "abort" # Configures the compiler to abort the program on panic
    during development builds.
[profile.release]
panic = "abort" # Configures the compiler to abort the program on panic
    during release builds.
```

Here is the detail explaination of this part:

- [profile.dev] and [profile.release]: These sections allow you to specify settings for development (cargo build) and release (cargo build --release) profiles, respectively.
- panic = "abort": By default, Rust tries to recover from errors (panics) by unwinding the program's stack, which can't be done without additional support. In this case, we want the program to just stop immediately when an error happens. Setting panic = "abort" makes the program do that.

Step 3: Writing the Freestanding Rust Code

Open the src/main.rs file. Replace its contents with the following code:

```
#![no_std] // disable the Rust standard library
#![no_main] // disable all Rust-level entry points

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() {
    loop {}
}

#[panic_handler] // this function is called on panic
fn panic(_info: &core::panic::PanicInfo) -> ! {
    // the ‘!‘ type means "this function never returns"
    // place holder for now, we'll write this function later
    loop {}
}
```

Here is the detail explaination of this part:

- #![no_std]: This attribute disables the standard library. It is used for low-level programming, where direct control over the system is required.
- #![no_main]: Rust programs typically start execution from the main function. This attribute disables it, which is necessary for creating a freestanding binary.
- #[no_mangle]: This attribute prevents Rust from changing the name of the _start function, ensuring the linker can find it.

- pub extern "C" fn _start() -> !: Defines the entry point for our program. The function will use the C ABI for compatibility with C code 2.6.1. The ! return type indicates that this function will never return.
- #[panic_handler]: Specifies the function to call when a panic occurs. Panics can occur for various reasons, such as out-of-bounds array access.

For more information see 2.5

Step 5: Building the Project

By default, the linker includes the C runtime, which can lead to errors. To avoid this problem, we have two options. One way is to pass different parameters based on the operating system we're using. However, a more direct approach is to specify that we're compiling for an embedded system. This way, the linker won't attempt to link the C runtime environment, ensuring a successful build without linker errors.

First add the target architecture. Open your terminal run the command

```
rustup target add thumbv6m-none-eabi
```

This command uses rustup, the Rust toolchain installer, to add support for compiling Rust code for the thumbv6m-none-eabi target, which is a common architecture for ARM Cortex-M microcontrollers. You can also choose alternative targets as long as the underlying environment doesn't include an operating system.

Execute

```
cargo build --target=thumbv6m-none-eabi
```

to compile your project for the thumbv6m-none-eabi target. This tells Cargo, Rust's package manager and build system, to compile the project for the specified architecture rather than the default target platform that is your host machine.

3.3.3 Output

At this point, the program won't produce any output. If all the steps proceed smoothly, it should compile successfully without reporting any errors. For example:

```
cargo build --target=thumbv6m-none-eabi
   Compiling chronos_labs v0.1.0 (/home/lucia/2023cse/project/
       chronos_labs)
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
```

3.4 Task 2 - Build Minimal Kernel

3.4.1 Introduction

We'll use Rust to create a small 64-bit kernel for the x86 architecture base on program we made for previous task. We will use the bootloader tool to create a bootable disk image, allowing us to launch it using QEMU.

See branch lab-1-2 for the code.

3.4.2 Implementation

Step 1: Create a custom target specification file

In the previous task, we referenced an embedded environment as our compilation target. However, to build our custom operating system, we need to write a custom target specification file. Create a chronos_labs.json file in the root directory, although you can choose any name for this file. Create this file:

```
touch chronos_labs.json
```

Here is the content of the 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"
}
```

You don't necessarily need to understand what each fields represents, but here are a few of the parameters that are more unique compared to other operating systems and might be worth understanding:

- "llvm-target": "x86_64-unknown-none": Specifies the target architecture for the compiler. Here, it's for 64-bit x86 architecture without a specific vendor or operating system.
- "arch": "x86_64": The architecture of the target system, indicating a 64-bit processor.

- "linker-flavor": "ld.lld" and "linker": "rust-lld": Specify which linker to use, here it's cross-platform LLD linker included with Rust.
- "panic-strategy": "abort": Determines how to handle panic situations. "abort" means the program will immediately stop, without trying to unwind the stack.
- "disable-redzone": true: Disables the red zone, or sometimes it could lead to stack corruption.o
- "features": "-mmx,-sse,+soft-float": Specifies CPU features to enable or disable. Here, MMX and SSE are disabled, while software-based floating-point calculations are enabled. Disable of mmx and sse features means we disable the Single Instruction Multiple Data (SIMD) instructions because it will cause interruption too frequently. And enable soft-float that simulates all floating-point operations using software functions that rely on regular integers will solve the error by disable SIMD.

Step 2: Create .cargo/config.toml

In your project's root directory, create a folder named .cargo

```
mkdir .cargo
```

Inside the .cargo folder, create a file named config.toml

```
cd .cargo && touch config.toml
```

Open config.toml and paste the following contents:

```
[unstable]
build-std = ["core", "compiler_builtins"]
build-std-features = ["compiler-builtins-mem"]

[build]
target = "chronos_labs.json" #replace with your file name
```

Here are explanations for each lines:

[unstable]

build-std = ["core", "compiler_builtins"]: Tells Cargo to compile essential Rust libraries core and compiler_builtins from scratch

build-std-features = ["compiler-builtins-mem"]: Activates memory functions in compiler_builtins

[build]

target = "chronos_labs.json": Points to a custom target file to specify how to compile for a particular setup.

Step 3: Use bootimage

Add bootimage to dependency, open Cargo.toml and add under [dependencies]

```
[dependencies]
bootloader = "0.9.23"
```

Also add these in .cargo/config.toml

```
[target.'cfg(target_os = "none")']
runner = "bootimage runner"
```

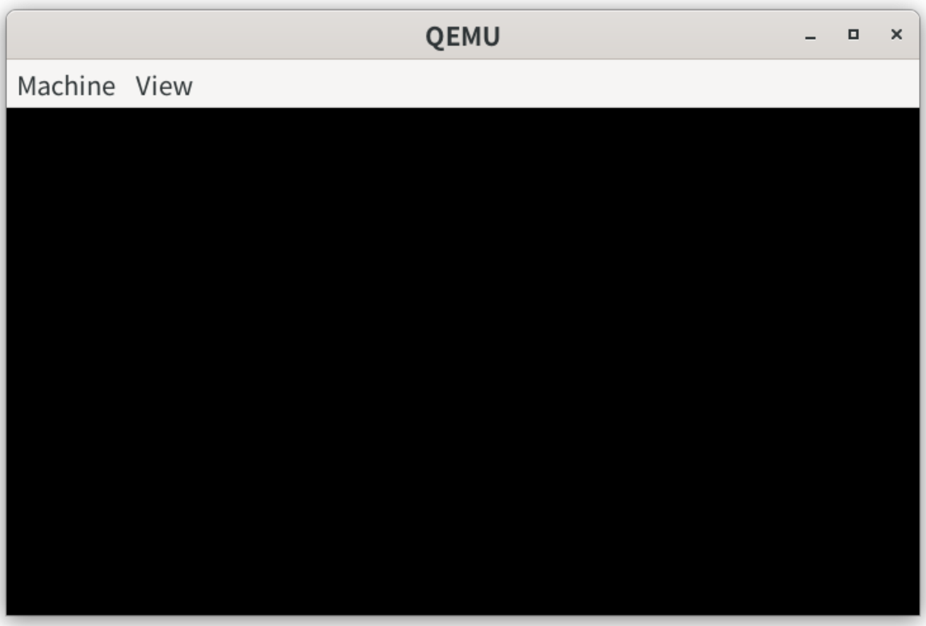
[target.'cfg(target_os = "none")'] include "chronos_labs.json" file, and runner key defines command gets executed bootimage runner after the project has been successfully compiled.

Now we can use cargo run to execute this project. The cargo run command is a convenient tool used in Rust projects to compile and run the application code in one step.

If you interested in what did bootimage tool do, see5.1

3.4.3 Output

The output should be a blank QEMU window:



3.5 Task 3 - Show something!

3.5.1 Introduction

We use VGA text buffer to make output for this operating system. It is because it's simple and straightforward to write to. VGA text mode provides a direct way to display text on the screen by writing characters and their attributes (like color) to a specific area of memory. More inforamtion about it 5.2

In the Windows operating system, encountering an error often results in the appearance of a blue screen. Therefore, we will also attempt to display a blue screen in our system.

3.5.2 Implementation

Open src/main.rs

Step 1: Define Constants

Add these constant

```
const BUFFER_HEIGHT: usize = 25;
const BUFFER_WIDTH: usize = 80;
const BACKGROUND_COLOR: u16 = 0x1000; // blue background, black
    foreground
```

- const BUFFER_HEIGHT: usize = 25; Defines a constant named BUFFER_HEIGHT with a type of usize (an unsigned size type, which means it's a number that can't be negative and its size varies based on the computer architecture). The value 25 represents the number of text lines that the VGA text buffer can display at one time.
- const BUFFER_WIDTH: usize = 80; Similar to the first line, this defines a constant named BUFFER_WIDTH, also of type usize. The value 80 represents the number of characters that can fit on a single line of the VGA text buffer.
- const BACKGROUND_COLOR: u16 = 0x1000; This line defines a constant named BACKGROUND_COLOR with a type of u16 (a 16-bit unsigned integer). The value 0x1000 is a hexadecimal number that specifies the color attributes for the text and background. It will be 000100000000 in binary. Recover from 5.2, we know that it sets the background color to blue and the foreground (text) color to black.

See more about variables in Rust on 2.4

Step 2: Initializes Buffer

Add this line inside _start()

```
let vga_buffer = unsafe { core::slice::from_raw_parts_mut(0xb8000 as *
    mut u16, 2000) };
```

- unsafe { ... }: In Rust, unsafe blocks are used for performing unsafe operations, such as direct hardware access or low-level memory operations. Here, the unsafe block is used for operations related to hardware interaction.

- core::slice::from_raw_parts_mut(0xb8000 as *mut u16, 2000): This is a function call that creates a mutable slice

- 0xb8000 as *mut u16: This is a memory address conversion, casting the hexadecimal address 0xb8000 as a mutable pointer to a u16 type. 0xb8000 is starting address of VGA buffer, and each element is a 16-bit character/color combination.
- 2000: This is the length of the slice, indicating that the slice contains 2000 u16 elements. That's the size of VGA buffer by 80*25 = 2000

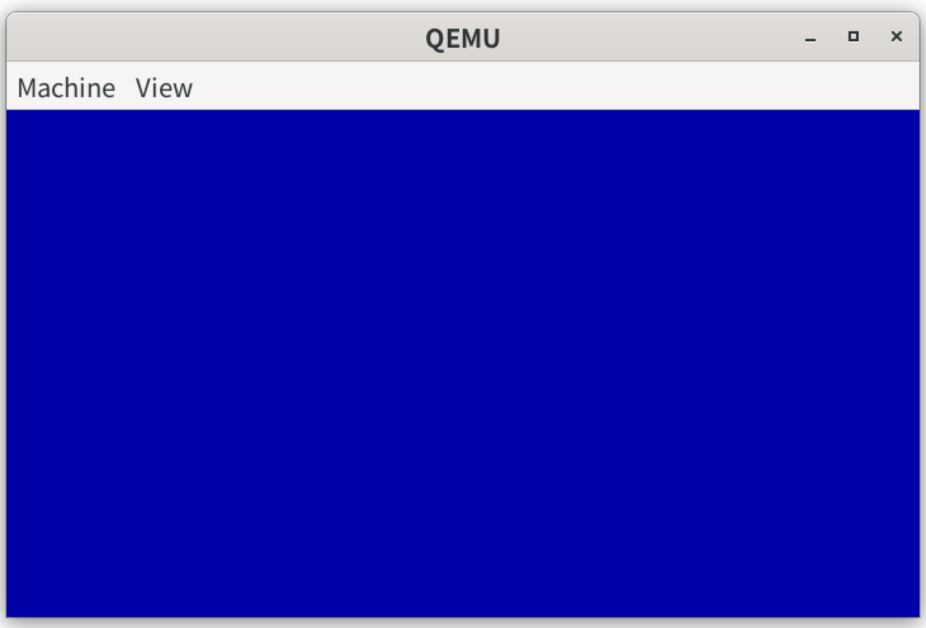
Step 3: Assign values

Add a for loop below

```
for i in 0..(BUFFER_HEIGHT * BUFFER_WIDTH) {
        vga_buffer[i] = BACKGROUND_COLOR;
    }
```

Sets each element of the vga_buffer array to BACKGROUND_COLOR. In this case, leave character section empty.

3.5.3 Output



You may try to display other color than blue by yourself? You could find the color code from `https://wiki.osdev.org/Printing_To_Screen`

Chapter 4

Lab 2 - VGA output

4.1 Expected Outcome

In this lab, we'll implement **print!** and **println!** in our operating system, facilitating future feature implementation and output. We establish an interface that ensures safety and simplicity by isolating all unsafe operations within a dedicated module.

4.2 Task 1 - Print text at a specified position using ASCII encoding

4.2.1 Introduction

At the end of lab 1, we traversed the entire VGA buffer to output a blue screen. Now we will continue to use slices to output specific text at specified positions.

4.2.2 Implementation

Step 1: Modify the BACKGROUND_COLOR constant

In the previous implementation, we used the u16 data type for assignment because we didn't need to output specific characters. However, this time we will only define the color part. If you forget how to define it, you can refer to the documentation.5.2

Change constant name into COLOR

```
const COLOR: u8 = 0x04; // black background, red foreground
```

You could change into different value to represent different color as you wish!

Step 2: Modify vga_buffer

We change pointer datatype from u16 to u8 because we want to assign character and color seperately. Length change to 4000 so the end of buffer remain unchanged.

```
let vga_buffer = unsafe { core::slice::from_raw_parts_mut(0xb8000 as *mut
    u8, 4000) };
```

Step 3: Print character
Delete the for loop under buffer definition and change into

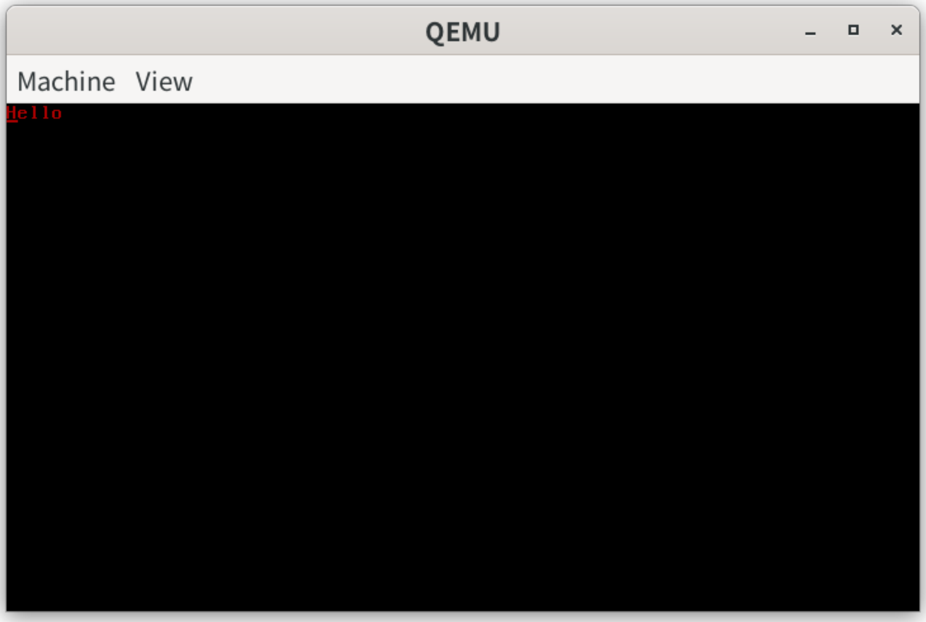
```
vga_buffer[0] = b'H';
vga_buffer[1] = COLOR;
vga_buffer[2] = b'e';
vga_buffer[3] = COLOR;
vga_buffer[4] = b'l';
vga_buffer[5] = COLOR;
vga_buffer[6] = b'l';
vga_buffer[7] = COLOR;
vga_buffer[8] = b'o';
vga_buffer[9] = COLOR;
```

- b'H': It represents a byte literal(a single byte of data). In this case, corresponds to the ASCII encoding of the uppercase letter 'H'.
- COLOR: Color part of the character, represent red foreground and black background.

Remember to keep

```
loop {}
```

Don't delete it!
You would see the output like this if everything going well



Step 4: Print somewhere else
Remember that

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

You can print words at the new line

```
// write "World" at the next line
    vga_buffer[160] = b'W';
    vga_buffer[161] = COLOR;
    vga_buffer[162] = b'o';
    vga_buffer[163] = COLOR;
    vga_buffer[164] = b'r';
    vga_buffer[165] = COLOR;
    vga_buffer[166] = b'l';
    vga_buffer[167] = COLOR;
    vga_buffer[168] = b'd';
    vga_buffer[169] = COLOR;
```

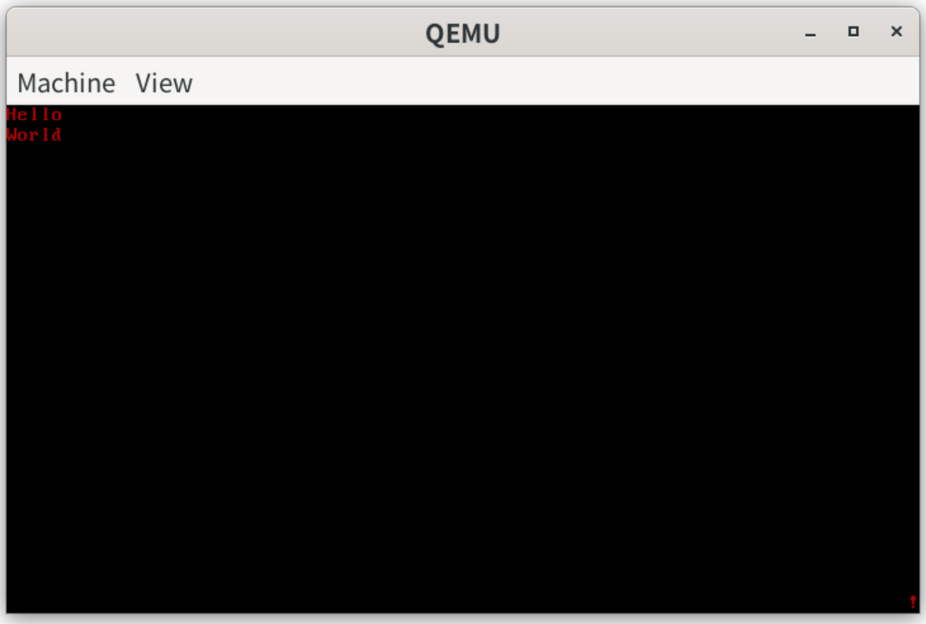
Or at the end

```
    // End
    vga_buffer[3998] = b'!';
    vga_buffer[3999] = COLOR;
```

Now you can visually see the correspondence between memory addresses and screen positions.

4.2.3 Output

The code may look cumbersome, but it's okay because it's just to demonstrate the correspondence between the VGA buffer and the screen. We'll implement the print! and println! functions in a more elegant way.



4.3 Task 2 - Write byte

4.3.1 Introduction

In this task, we will create a vga.rs file to handle VGA output specifically, aiming to improve code readability. At the same time, we'll use a more elegant approach to output single characters, recognize newline characters, and handle situations where characters exceed the screen.

4.3.2 Implementation

Step 1: Create src/vga.rs file

Create a vga.rs file in the src folder, and copy the necessary declarations. We will import this file as module to main. More about module?2.7

```
const BUFFER_HEIGHT: usize = 25;
```

```
const BUFFER_WIDTH: usize = 80;
const COLOR: u8 = 0x04; // black background, red foreground
```

Step 2: Create a new struct represent single VGA character

We create VGAChar that contain both ASCII and color. And in Rust, the ordering of fields in default structs is not defined, so we use the repr(C) attribute to ensure that the struct's fields are laid out in memory exactly in order.

```
#[repr(C)]
struct VGAChar {
    ascii: u8,
    color: u8,
}
```

More about struct see 2.8

It also relate to the big endian and little endian that define in specification file 5.3

Step 3: Struct for buffer

We use the volatile library, which helps us prevent Rust's compiler from optimizing out our write operations to the buffer.

Add support in Cargo.toml

```
[dependencies]
bootloader = "0.9.23"
volatile = "0.2.6"
```

Add these lines under VGAChar

```
#[repr(transparent)]
struct Buffer {
    chars: [[Volatile<VGAChar>; BUFFER_WIDTH]; BUFFER_HEIGHT],
        // 2D array
}
```

We use a 2D array to represent rows and columns.

- #[repr(transparent)]: Buffer has the same memory layout as its inner 2D array of Volatile<VGAChar> elements

Step 4: Implement Writer

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

For line buffer: &'static mut Buffer

It is a field named buffer, is a mutable reference to a Buffer instance.

- & denotes a reference
- 'static is a lifetime specifier. When used in a context like this, it indicates that the reference can live for the entire duration of the program.
- mut means it allows modification of the Buffer instance it refers to

We use writer to implement output function.The first character will appear in the top-left corner. When a row is filled or a newline character is entered, input will move to the next row. If the entire screen is filled, all input will shift up one row, clearing the initial first row of input. To achieve this, we need to keep track of the current input position, namely the row and column numbers.

This is the write_byte function that can output single character

```
impl Writer {
    pub fn write_byte(&mut self, byte: u8, color: u8) {
        match byte {
            b'\n' => self.new_line(),
            byte => {
                if self.column_position >= BUFFER_WIDTH {
                    self.new_line();
                }

                let row = self.row_position;
                let col = self.column_position;

                self.buffer.chars[row][col].write(VGAChar {
                    ascii: byte,
                    color,
                });
                self.column_position += 1;
            }
        }
    }
}
```

You can find more information about 'impl' here.2.9

In the next step, we will implement the new_line function.

We use the write method from the volatile library to write code at the corresponding position (therefore Buffer is with the mut parameter). The color is then a shorthand for color: color, which can be omitted when the variable name matches the field name.

After written into each character, we plus the column_position.

Step 5: new_line() function

```
impl Writer {
    pub fn new_line(&mut self) {
        if self.row_position < BUFFER_HEIGHT - 1 {
```

```
                self.column_position = 0;
                self.row_position += 1; // change to new line
            } else { // if the row is full, scroll up
                for row in 1..BUFFER_HEIGHT {
                    for col in 0..BUFFER_WIDTH {
                        let character = self.buffer.chars[row][col].read();
                        self.buffer.chars[row - 1][col].write(character);
                    }
                }
                self.clear_row(BUFFER_HEIGHT - 1);
                self.column_position = 0;
            }
        }

    fn clear_row(&mut self, row: usize) { // new function to clear a row
        for col in 0..BUFFER_WIDTH {
            self.buffer.chars[row][col].write(VGAChar {
                ascii: b' ',
                color: COLOR,
            });
        }
    }
}
```

A new impl blocks is used to keep clearer and more maintainable codes, you can merge these two functions into previous block if you wish.

The new_line function is a used to move the next character to the start of the next line. If the character is already on the last line of the screen, it scrolls the screen up by one line by reading and write each character one line above. And clearing the last line by writing space to each character. The clear_row function is private function meaning it can only be accessed within its own module.

Testing

Here is 2 testing function. Write in src/vga.rs

```
pub fn test_print() {
    let mut writer = Writer {
        column_position: 0,
        row_position: 0,
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };

    writer.write_byte(b'H', COLOR);
    writer.write_byte(b'\n', COLOR);
    writer.write_byte(b'e', COLOR);
}
```

```
pub fn test_rolldown() {
    let mut writer = Writer {
        column_position: 0,
        row_position: 0,
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };

    for i in 1..= 25 {
        let line: u8 = i + b'0';
        writer.write_byte(line, COLOR);
        writer.write_byte(b'\n', COLOR);
    }
}
```

We creates an instance of the Writer struct, and test the basic function. To use these, goto main.rs

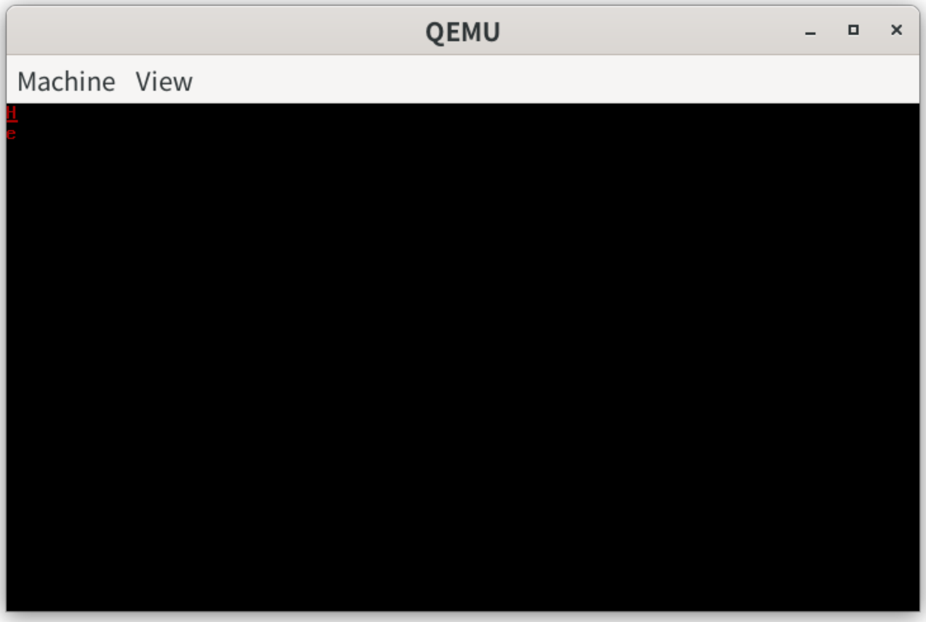
```
mod vga; // import the `vga` module

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() {
    // vga::test_print();
    vga::test_rolldown();
    loop {}
}
```

4.3.3 Output

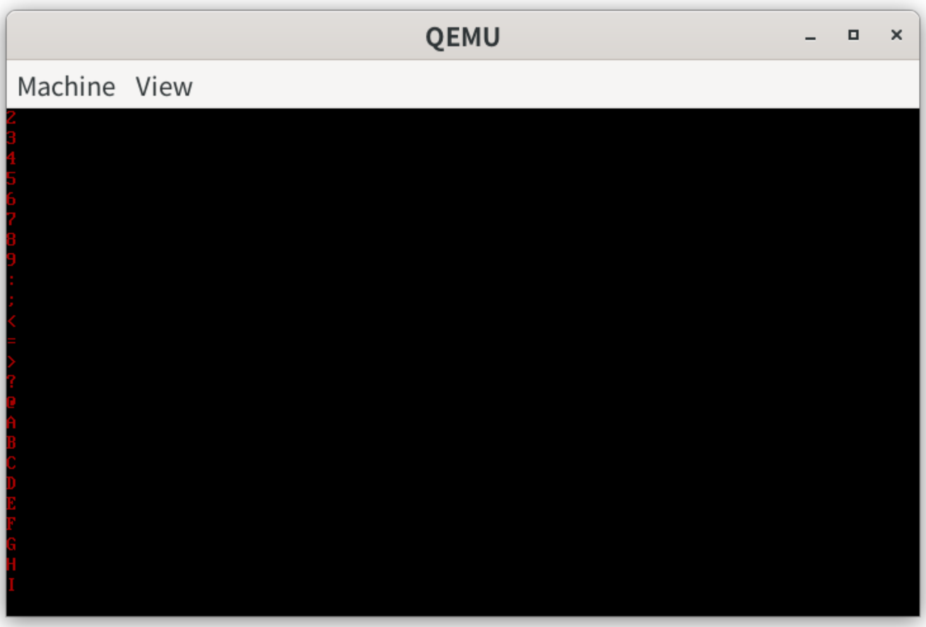
We have 2 test functions, rember to run them one by one!

For the vga::test_print(); we have



This indicates that both the input bytes and the newline character functionality are working correctly.

And for vga::test_rolldown(); we have



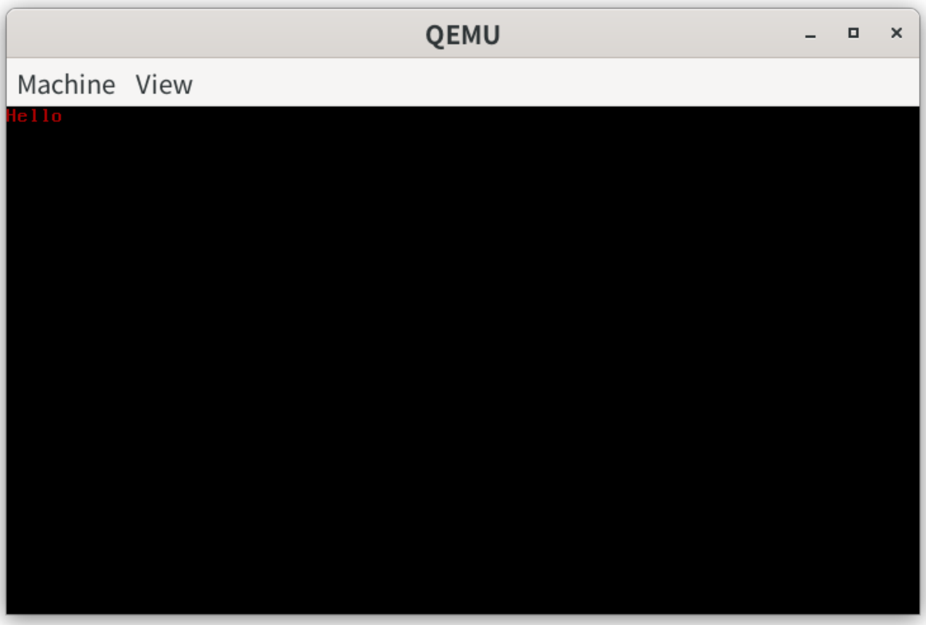
The result is normal because we sequentially output the subsequent content of the ASCII table. We can see that the '1' that should have been outputted in the first row disappears and is replaced by '2'. This indicates that the functionality of shifting the other content upwards when the row limit is exceeded is working correctly.

4.4 Task 3 - Enable print! and println! macro

4.4.1 Introduction

4.4.2 Implementation

4.4.3 Output



Chapter 5

Extra knowledge

5.1 What did bootimage tool do?

5.2 VGA text buffer

The VGA (Video Graphics Array) text buffer is a specific area of memory used to display text on the screen in a text-mode environment. It ist located at the memory address 0xB8000. It extends to 0xB8FA0, covering a space that allows for 25 lines of 80 characters each, making up the standard 80x25 text mode.

Each character in the VGA text buffer is represented by two bytes:

The first byte represents the ASCII code of the character, determining which character to display. The second byte defines the color of the character, with the lower 4 bits specifying the foreground color and the upper 4 bits the background color.

Attribute								Character							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Blink[n 1]	Background color			Foreground color[n. 2][n. 3]				Code point							

[1]

5.3 Big endian and little endian

Big endian and little endian are two ways of ordering bytes in multi-byte data types. The difference lies in the order in which the bytes are stored in memory, we are using little endian according to the target specification file. In a little endian system, the least significant byte (LSB) is stored at the lowest memory address. Conversely, in a big endian system, it's the opposite. Here is the example

We have structure VGAChar

```
struct VGAChar {
    ascii: u8,
    color: u8,
}
```

If you create a VGAChar with an ascii of 0x41 ('A') and a color of 0x04, it would be stored in memory as 41 04 in a little endian system. This is because ascii_character is the first field in the struct, so it gets the lower memory address.

Bibliography

[1] VGA text mode (2024) Wikipedia. Available at: https://en.wikipedia.org/wiki/VGA_text_mode (Accessed: 29 January 2024).