The Guide to ChronOS

Betty Liu u2061245

March 25, 2024

Contents

Chapter 1

Introduction to ChronOS

Please ensure to stay updated with the GitHub repo here to obtain the latest version of the lab sheet.

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

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.

After completing the code, you can compare your own code or start development directly based on a specific branch. Skip the tedious preparation work and dive straight into your favorite area.

You can get the codes by

```
git clone git@github.com:acyanbird/chronos_labs.git
cd chronos_labs/
```

```
git branch -r | grep -v '\->' | sed "s,\x1B\[[0-9;]*[a-zA-Z],,g" | while read remote; do git branch --track "${remote#origin/}" "$remote"; done
git fetch --all
git pull --all
```

You can get the code from each task by

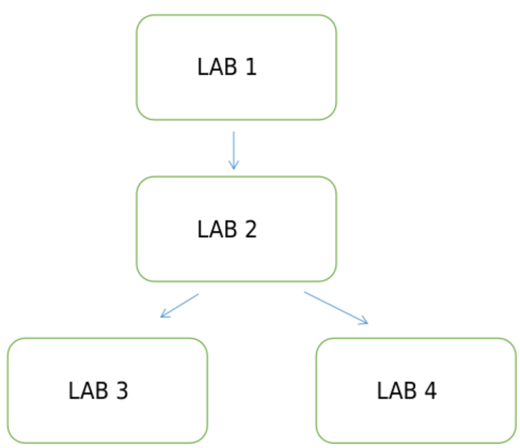
```
git checkout <branch-name>
# for example
git checkout lab2-3
```

If you want to work from existing branch

```
# like you currently on lab2-3 and want to work on it
git checkout -b <your-branch-name>
```

Learning Order:

Lab 0 is a simple tutorial on the Rust language. You can skip directly to Lab 1 for learning. I will provide a link to return to Lab 0 later. Please complete Lab 1 and Lab 2 in sequence, and then you can choose to proceed to either Lab 3 or Lab 4. These two labs are independent of each other.



Chapter 2

Lab 0 - Introduction to Rust Programming language

You can start from Lab 1 directly, you will jump back to here if you encounter new component from Rust.

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. You don't need to read through this Lab completely right now. Throughout the subsequent implementation of the operating system, if you encounter new Rust content, we will reference you back to the relevant sections of Lab 0. Combining examples will help you better understand their application.

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:

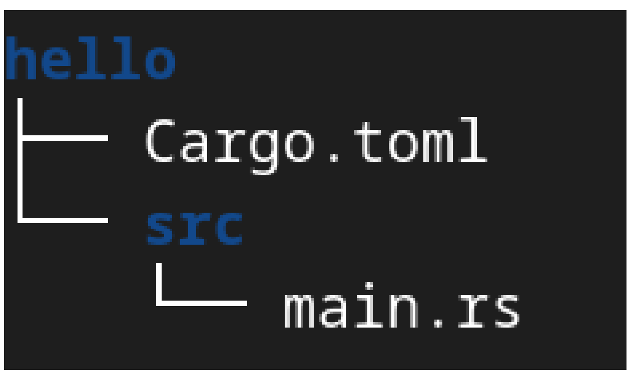


Figure 2.1: project structure

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.

```rust
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.

```rust
// 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.

```rust
// 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.

2.7 Module

In a Rust project, you can import this module using the following syntax:

```rust
mod vga;
```

It tells the compiler to load a module named vga. This means the compiler will search for a file named vga.rs in the current directory. And allowing you to use the functionality and variables provided by that module in the current scope.

For example

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

It can use test_rolldown() function that declare in vga.rs file.

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

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

The most important function in Rust is main, which is the entry point of many programs. But in our case the thing is a bit different, we don't have a standard main function since we are doing the system development. In lab1-1 it is using _start(). We use the panic as example to explain every part of function

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

- **fn keyword**; In Rust, a function is defined using the fn keyword, followed by the function name (panic in this case)
- **function parameters;** These are specified within parentheses if there is any (could be none) _info is a parameter of type &core::panic::PanicInfo
- **->**; indicates the return type of the function, ! means the function doesn't return a value

2.11 Match

It allows you to compare a value against a series of patterns and execute code based on which pattern matches. It's similar to a switch statement in other languages but more powerful. For our example:

```
match byte {
    // if not acceptable ASCII, print a space with error color
    0x20..=0x7e | b'\n' => self.write_byte(byte, COLOR),
    _ => self.write_byte(b' ', ERROR_COLOR),
    }
```

- 0x20..=0x7e | b'\n': This is a pattern that matches any byte within the range 0x20 to 0x7e or the newline character b'\n'. If the byte matches this pattern, self.write_byte(byte, COLOR) is executed.
- _: This is the wildcard pattern that matches any value that hasn't been matched by the previous patterns. Then self.write_byte(b' ', ERROR_COLOR) is executed, printing a space with error color.

2.12 Trait

In Rust, a trait defines a set of methods that types can implement. It allows for defining shared behavior across different types. It usually require programmer to provide n

2.13 Declarative Macro

2.14 Reference and Dereference

2.15 Self-study materials

However, to master this language, you'll need to continue learning as you go. Here are some recommended books and platforms for learning Rust:

- Official Rust Website: The official Rust website provides comprehensive documentation, tutorials, and resources for learning Rust.
- The Rust Programming Language: Affectionately nicknamed “the book,” The Rust Programming Language will give you an overview of the language from first principles[6]
- Rust by Example: Rust by Example (RBE) is a collection of runnable examples that illustrate various Rust concepts and standard libraries.[3] If you prefer learning a new language by reading example code, then this book might be more suitable for you.
- A half-hour to learn Rust: Providing lots of Rust snippets. It has brief explanation of each code snippet, suitable for quick browsing.
- rustlings: This project contains small exercises to get you used to reading and writing Rust code.[4]

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.

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

If you are not install Rust already, please follow instructions here 2.2

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
```

- Suitable IDE

After installing Rust, choose a suitable IDE as the development environment, as introduced in Lab 0. Please follow instruction here **2.2.1**

- 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.

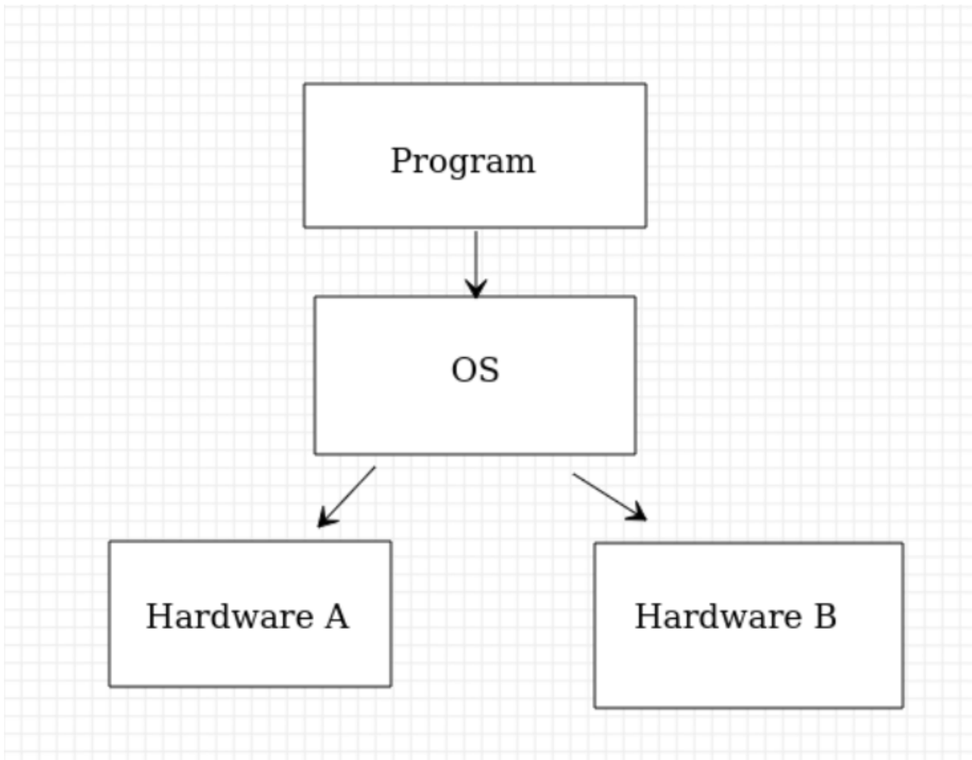


Figure 3.1: Common Rust Program

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.

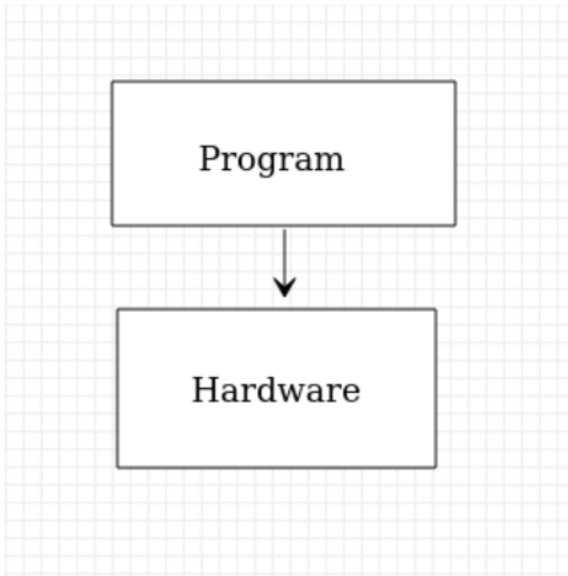


Figure 3.2: Standalone Rust

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 calling convention.8.1 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 and 2.10

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.

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"
```

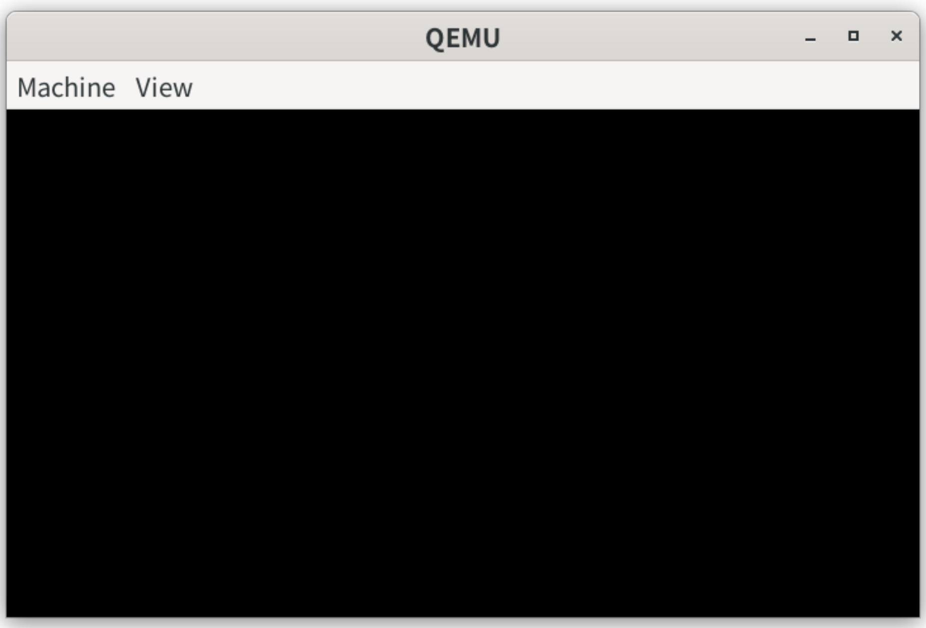
[target.'cfg(target_os = "none")'] include "chronos_labs.json" file, and runner key defines command gets executed bootimage runner after the project has been successfully compiled.

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, see8.2

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 8.3

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

```rust
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 8.3, 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

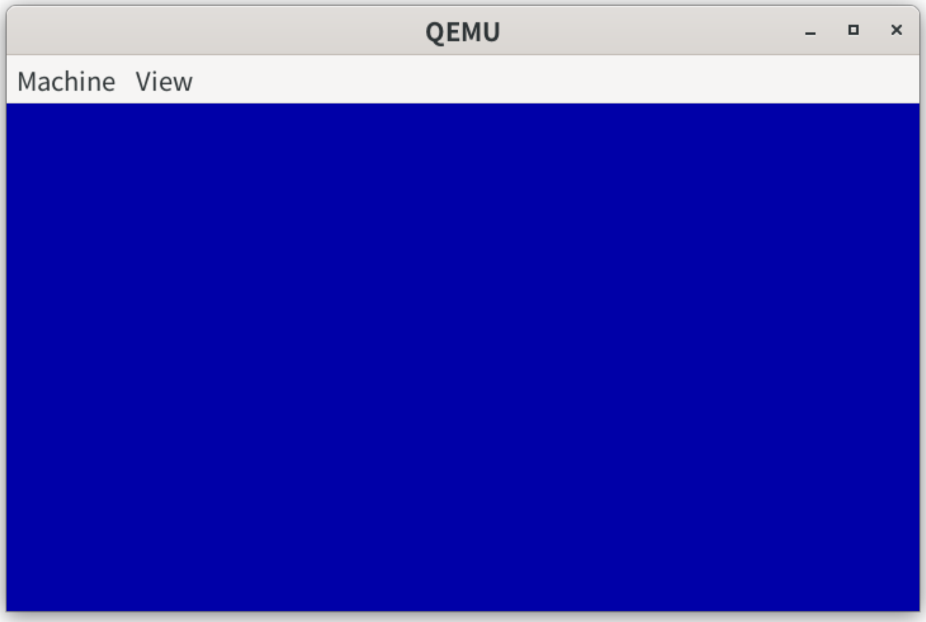
Step 3: Assign values

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 support of safety output string, number and support Rust's formatting macros instead of write on buffer directly. In this process, we will use traits to implement more functionality with less, cleaner, and more concise code. We will establish an interface that ensures safety and simplicity by isolating all unsafe operations within a dedicated module.
You will implement the section of the operating system that manages output.

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.8.3

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! See the reference on `https://wiki.osdev.org/Printing_To_Screen`

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;
```

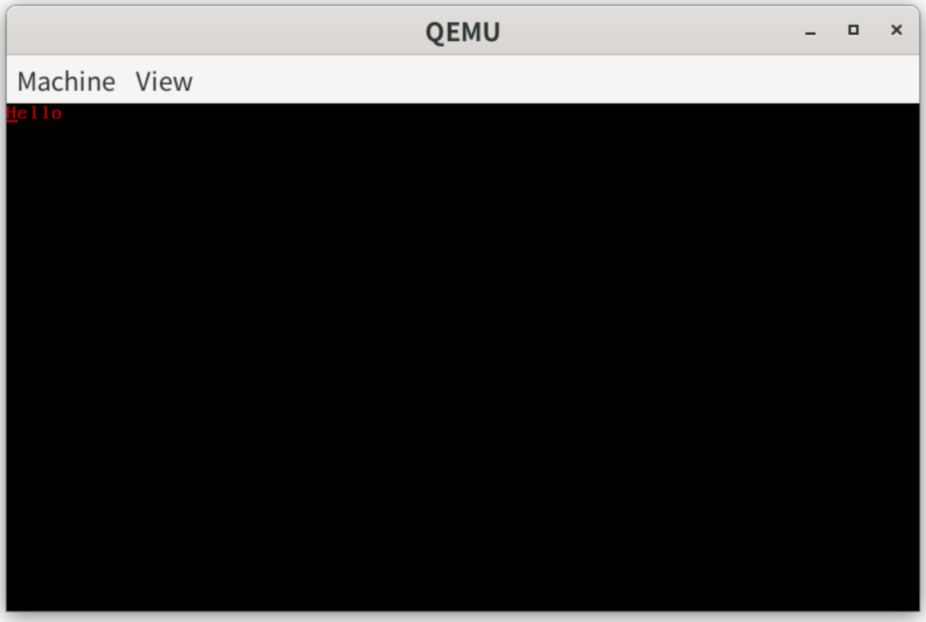
- b'H': It represents a byte literal(a single byte of data). In this case, corresponds to the ASCII encoding of the uppercase letter 'H'.
- COLOR: Color part of the character, represent red foreground and black background.

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;
```

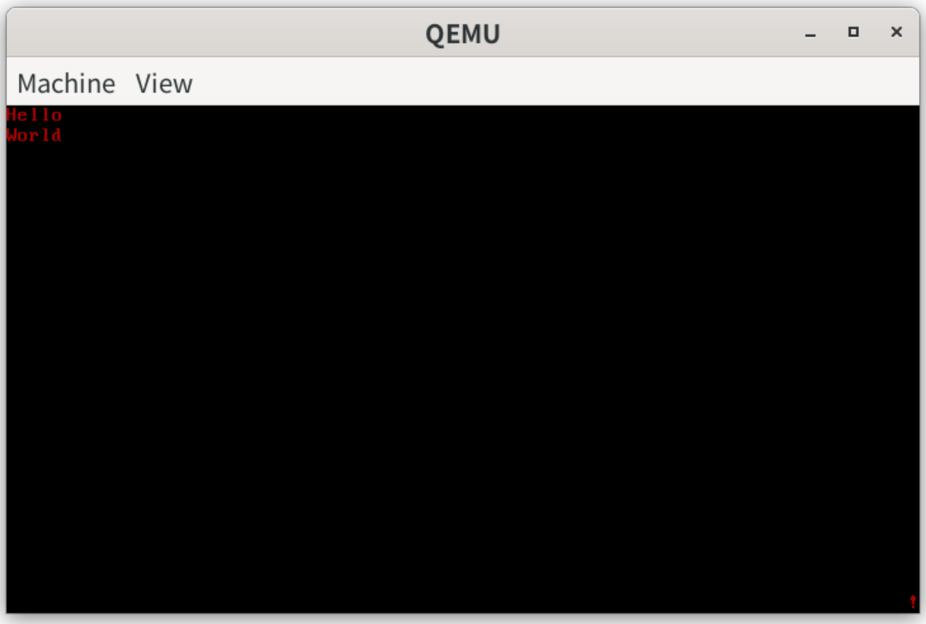
Or at the end

```
    // 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.

```rust
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.

```rust
#[repr(C)]
#[derive(Clone, Copy)]
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 8.4

- #[derive(Clone, Copy)]: Generates code to implement the Clone and Copy for that type, needed by volatile.

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

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

Add these lines under VGAChar

```rust
#[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

```rust
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

```rust
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

```rust
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();
```

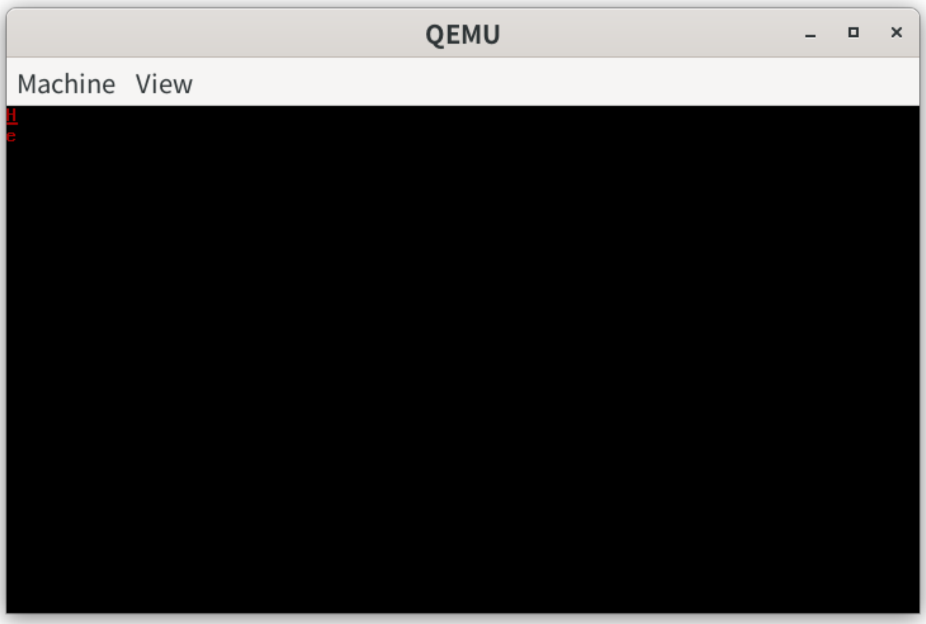
```
    loop {}
}
```

More about module?2.7

4.3.3 Output

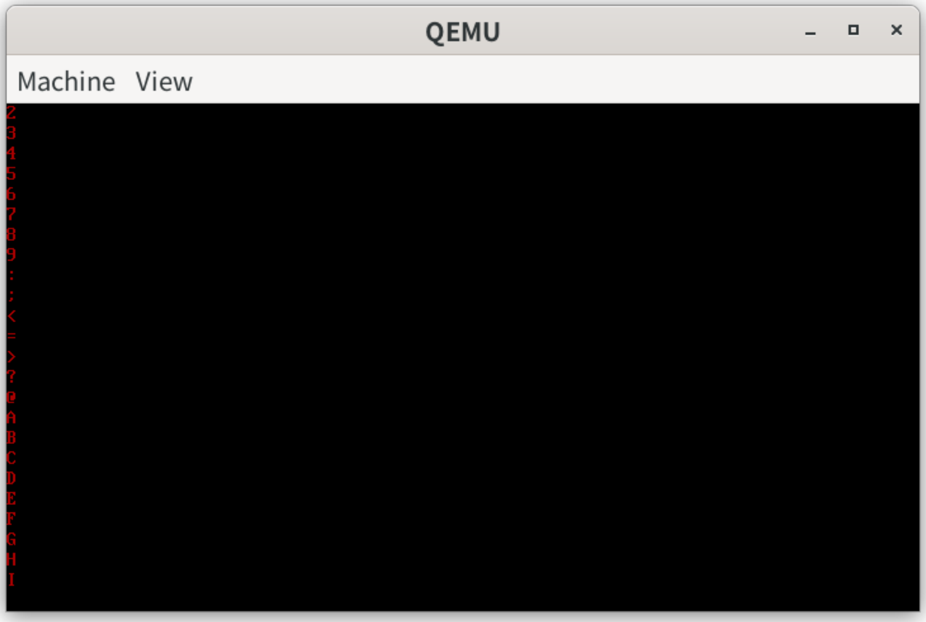
We have 2 test functions, rember to run them one by one!

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 write! and writeln!

4.4.1 Introduction

In this section, we will extend the write_byte function to write_string, using it to output strings. Furthermore, we will leverage Rust's trait functionality to accomplish formatted output.

4.4.2 Implementation

Step 1: write_string function

```
impl Writer {
    pub fn write_string(&mut self, s: &str) {
        for byte in s.bytes() {
            match byte {
                // if not acceptable ASCII, print a space with
                error color
```

```
                0x20..=0x7e | b'\n' => self.write_byte(byte,
                COLOR),
                _ => self.write_byte(b' ', ERROR_COLOR),
            }
        }
    }
}
```

There's a loop that iterates over each byte in the string s. For each byte it calls write_byte to writes the byte to VGA buffer after validation. That is because not all UTF-8 encodings are supported by VGA; it only supports ASCII and the encodings defined in Code page 437[5]. Therefore, if a character (such as a Chinese character) cannot be displayed, we will output a blue block as a substitute. You can also choose your own error output.

Figure 4.1: Code page 437

You can learn more about match here2.11

Step 2: Implement trait

Trait has many use cases, but we won't delve into all of them now. According to the documentation[6], after implementing fn write_str(&mut self, s: &str) -> Result, the trait can provide us with various functionalities like formatting output.

We just need to substitute the previously implemented write_string function, ensuring that the name write_str remains unchanged.

```
impl core::fmt::Write for Writer {
    fn write_str(&mut self, s: &str) -> core::fmt::Result {
        self.write_string(s);
        Ok(())
    }
```

```
}
```

- core::fmt::Result: It is an enum with two value, Ok and Err represents the result of a formatting operation.
- Ok(()): It means that the operation was successful, but there is no specific value to return, so it returns the unit value ().

You can learn more about trait here

Step 3: Public Buffer and writer

Because we need to define Writer in main.rs, we have to make Buffer and Writer visible. This is a somewhat unsafe operation, but for now, let's proceed with this to complete the testing.

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

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

- pub: It allows the item to be visible and it can be accessed and used outside of its module.

Step 4: Create lib.rs

Using lib.rs to import all modules is standard practice within Rust community. We organizing all module imports in one place for clarity and consistency in the project structure.

```
#![no_std]
pub mod vga;
```

Test

In main file, using

```
use chronos_labs::vga::Writer;
use chronos_labs::vga::Buffer;
```

Instead of

```
mod vga;
```

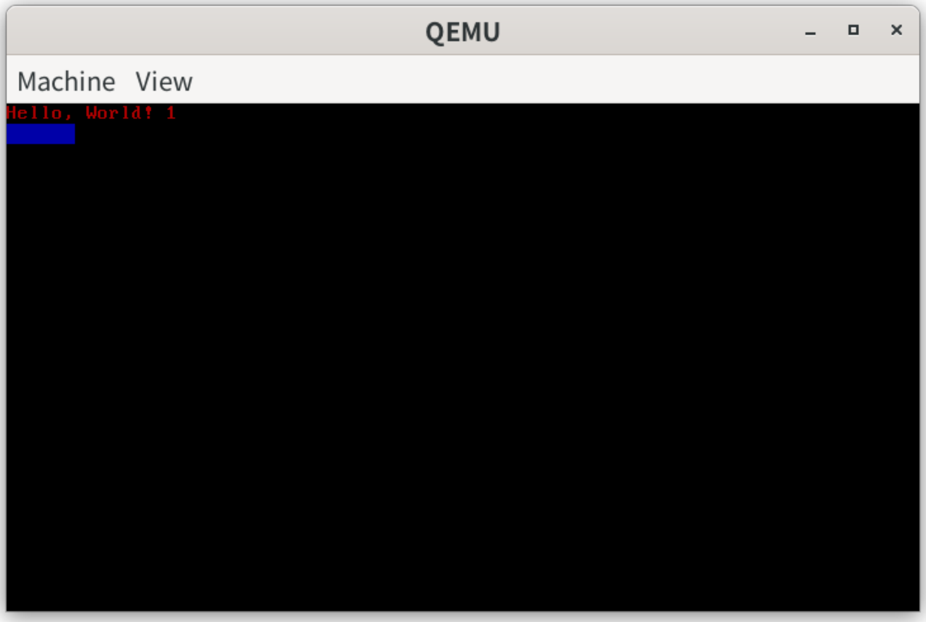
To directly access to these two structure. Remember replace the chronos_labs if you using other name.

```
#[no_mangle]    // don't mangle the name of this function
pub extern "C" fn _start() {
    let mut writer = Writer {
        column_position: 0,
        row_position: 0,
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };

    let num : i32 = 1;

    use core::fmt::Write;
    writeln!(writer, "Hello, World! {}", num).unwrap();
    write!(writer, "汉字").unwrap();

    loop {}
}
```

- unwrap(): This is used to simplify code when handling Result types. We could use this becuase we are confident that the result will always be Ok().

4.4.3 Output

We can see the format is working so we could insert variable into string. Each chinese character using 3 bytes, so 6 blue boxes is printed in total.



4.5 Task 4 - Writer in vga.rs

4.5.1 Introduction

We can directly create a Writer instance inside the vga module and access it by use. By instantiating Writer within vga instead of _start(), this encapsulation ensures cleaner and safer code. It also facilitates future calls to the Writer.

4.5.2 Implementation

Step 1: Add dependency

```
[dependencies]
...
spin = "0.5.2"
lazy_static = { version = "1.0", features = ["spin_no_std"] }
```

- lazy_static: This crate in Rust provides a convenient way to define lazily initialized static variables. In our case is a writer instance that would be available throughout the entire program execution.
- spin: Add support of a mutex lock.

Step 2: Initialize static WRITER

Add these below struct writer

```
lazy_static!{
    pub static ref WRITER: Mutex<Writer> = Mutex::new(Writer {
        column_position: 0,
        row_position: 0,
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    });
}
```

- Mutex<Writer>: It represents a mutex lock where the data type protected inside the lock is Writer.

Why we are using mutex lock? Because we want to make WRITER changable during program (write into buffer). So we only allow one process modify the writer at each time.

Step 3: Change visibility

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

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

We don't need to define the Buffer, row_position, column_position, and other fields in main, so they can be made private again.

Step 4: Change lib.rs

```
#![no_std]
mod vga;
pub use vga::WRITER;
```

We export WRITER only.

Step 5: Clear screen

Add a function clear screen to reset the output window

```
impl Writer {
    pub fn clear_screen(&mut self) {
        for row in 0..BUFFER_HEIGHT {
            self.clear_row(row);
        }
        self.column_position = 0;
        self.row_position = 0;
    }
}
```

For now, you can call function by using

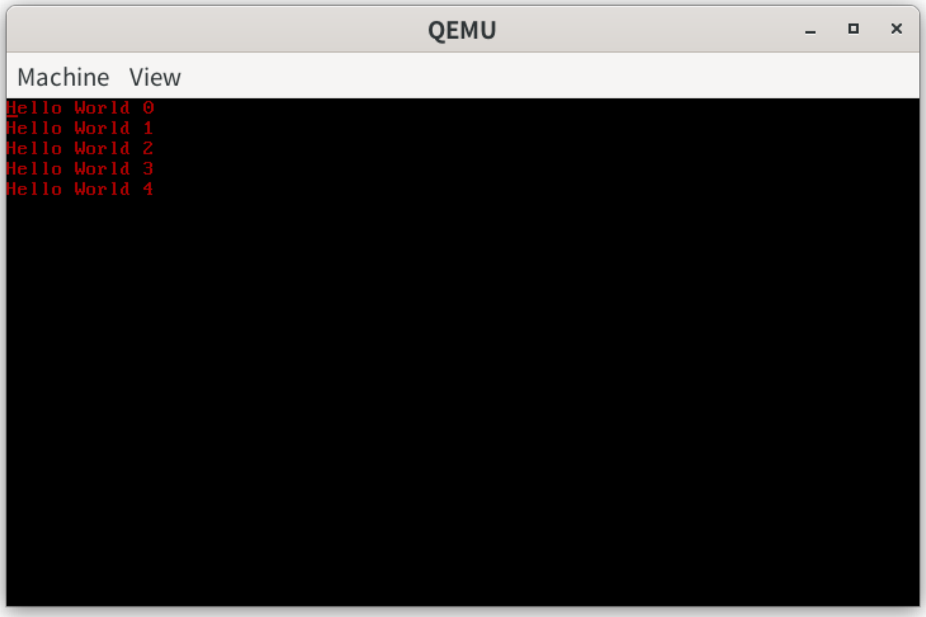
```
WRITER.lock().clear_screen();
```

Test
In main.rs

```
...
use chronos_labs::WRITER;
use core::fmt::Write;

#[no_mangle]    // don't mangle the name of this function
pub extern "C" fn _start() {
    for i in 0..5 {
        writeln!(WRITER.lock(), "Hello World {}", i).unwrap();
    }

    // WRITER.lock().clear_screen();   // uncomment this line to
    clear the screen

    loop {}
}
...
```

Imports the WRITER static variable from the vga module and import core::fmt::Write to use writeln! write! and writeln! is called

4.5.3 Output


QEMU
Machine View
Hello World 0
Hello World 1
Hello World 2
Hello World 3
Hello World 4

Chapter 5

Lab 3 - Keybroad Input

5.1 Expected Outcome

In this lab, you will implement the aspect of the operating system that manages interrupts and hardware input.

Chapter 6

Lab 4 - Paging

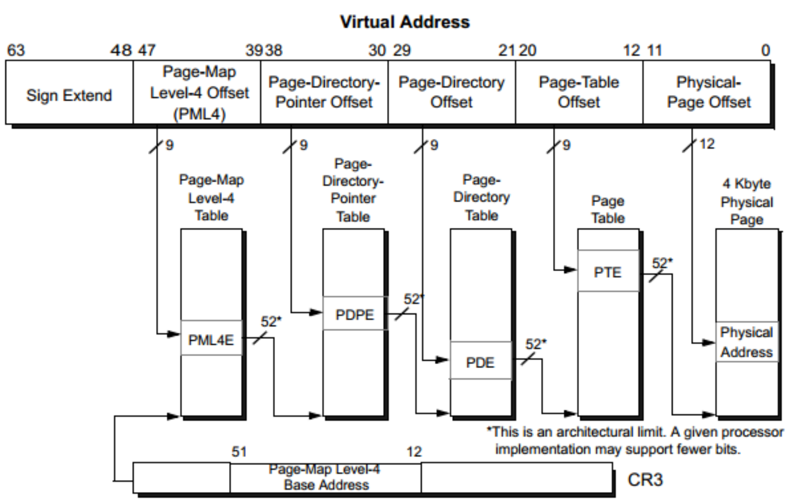
6.1 Expected Outcome

This part is associated with CS257:Advanced Computer Architecture's virtual memory section. It is strongly recommended that you complete this lab after studying/reviewing that course.

In this lab, we will be able to understand paging, translate virtual address to physical address and create mapping. Because this kernel is built on x86_64 architecture and using bootloader, it is already support multilevel page tables.

6.2 Quick Overview

If you're not very clear on the concepts of virtual addresses and physical addresses, please check here.8.5



[8]

Paging is a memory management technique in computers. It maps a process's virtual address space to physical memory. Physical memory is divided into fixed-size blocks called "pages," and the process's virtual address space is divided into equally sized blocks called "page frames." The operating system uses a page table to track the mapping between virtual and physical addresses. When a process accesses a virtual address, the operating system translates it into the corresponding physical address using the page table, enabling virtual memory.
In our operating system, virtual addresses map to the entire physical address space, allowing them to access any physical address. To avoid conflicts with actual addresses, we use a larger offset, such as 1T. This ensures that virtual addresses do not overlap with physical addresses.
The process of translating virtual addresses into physical addresses will be gradually introduced and implemented step by step in the upcoming labs.

6.3 Task 1 - Get in touch with L4 table

6.3.1 Introduction

In this task, we will access the CR3 register to obtain the physical address of the L4 page table. Then we need to virtual page that is mapped to the physical frame at address 0x1000 because we can't directly access physical address due to security reason. After that, we will traverse the L4 page table, print non-empty entries, and obtain the information stored in the L3 page table.

6.3.2 Implementation

Step 1: Add dependency
Edit Cargo.toml

```
[dependencies]
...
bootloader = { version = "0.9", features =
["map_physical_memory"]}
x86_64 = "0.14.2"
```

We need one more feature support of bootloader, and the support of x86_64 crate for page table support.

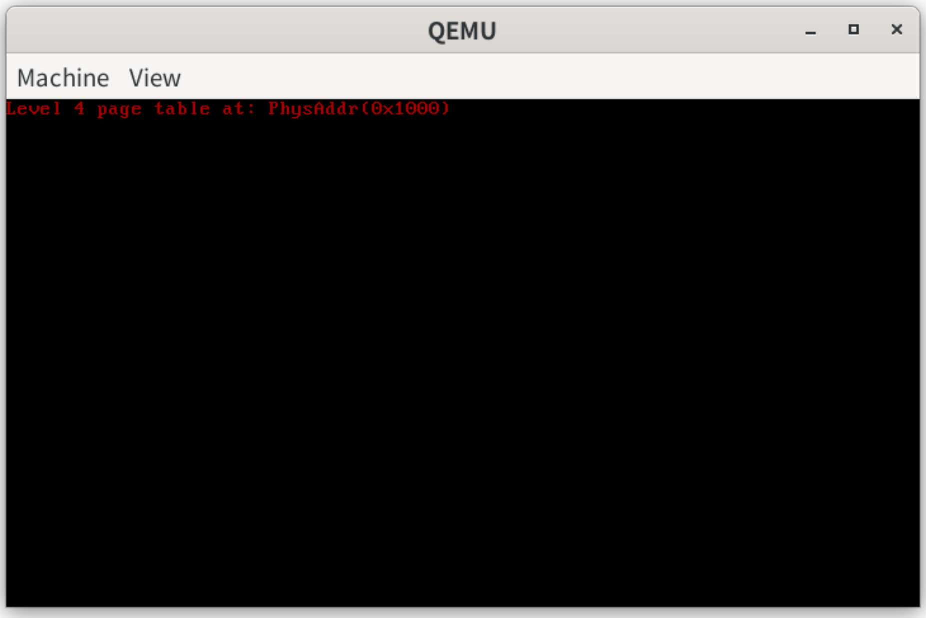
Step 2: Read CR3 register

The CR3 register stores the physical address of the L4 page table we want to access. Let's access it and print out the address of the L4 page table. Edit src/main.rs

```
...
use x86_64::registers::control::Cr3;
```

```
#[no_mangle]     // don't mangle the name of this function
pub extern "C" fn _start(boot_info: &'static BootInfo) -> ! {
    let (l4_entry, _) = Cr3::read();
    writeln!(WRITER.lock(),
             "Level 4 page table at: {:?}",
             l4_entry.start_address()
    ).unwrap();
```

_ menas we will not using this variable, so just ignore it.
We could get the output of address



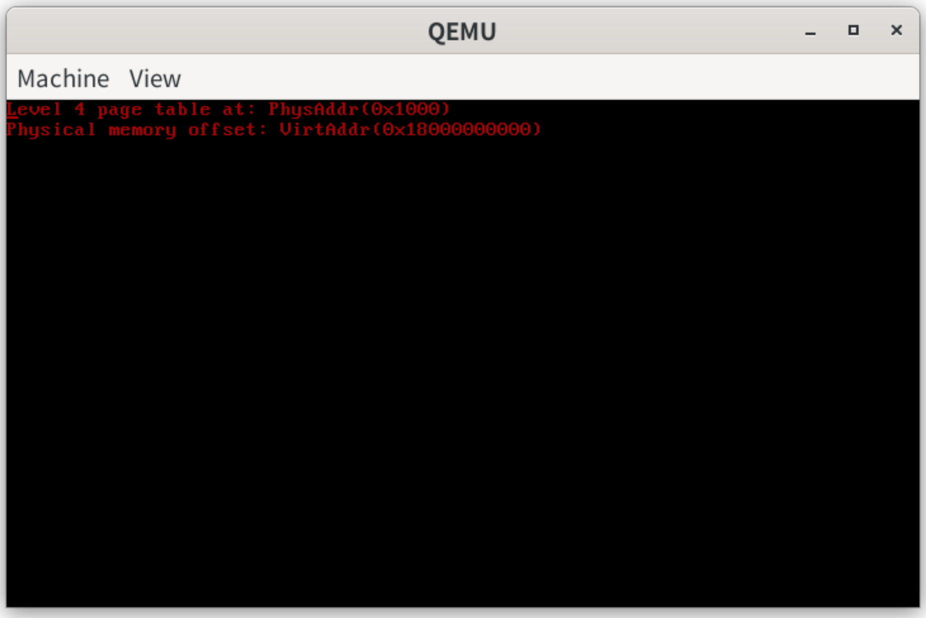
Step 3: Get physical memory offset

We need to read the physical memory offset in the boot_info, so our function should accept boot_info as input parameter.

```
...
use bootloader::BootInfo;

#[no_mangle]     // don't mangle the name of this function
pub extern "C" fn _start(boot_info: &'static BootInfo) -> ! {
    let (l4_entry, _) = Cr3::read();
    writeln!(WRITER.lock(),
             "Level 4 page table at: {:?}",
             l4_entry.start_address()
    ).unwrap();
```

```


    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    writeln!(WRITER.lock(),
             "Physical memory offset: {:?}",
             phys_mem_offset
    ).unwrap();
```

Then we could get the physical memory offset to create the virtual page.



Step 4: Create virtual page mapped to 0x1000

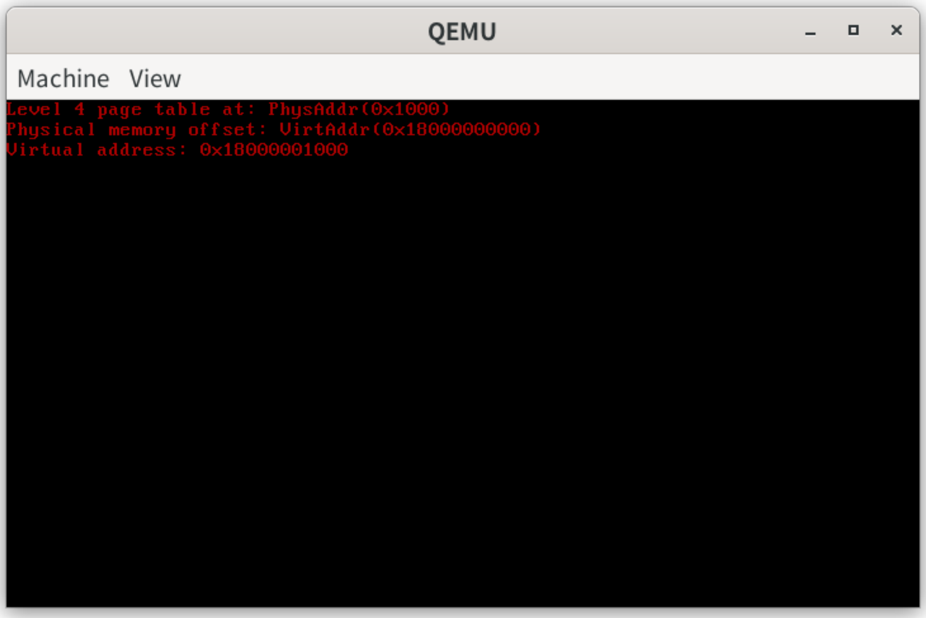
We could get the virtual address by physical address + physical page offset. Add under phys_mem_offset

```
...
use x86_64::structures::paging::PageTable;
use x86_64::VirtAddr;

...

    let virt = phys_mem_offset + l4_entry.start_address().as_u64();
    let l4_ptr: *mut PageTable = virt.as_mut_ptr();
    writeln!(WRITER.lock(), "Virtual address: {:?}", l4_ptr).unwrap();
```

```
    let l4_table = unsafe { &*l4_ptr };
```

- let l4_ptr: *mut PageTable : It declares variable l4_ptr in type *mut PageTable. *mut PageTable is the type of the variable. It's a mutable pointer (*mut) to a PageTable.

Then we could see the virtual address of page.



Step 5: Traverse the L4 page table
Under l4_table

```
for (i, entry) in l4_table.iter().enumerate() {
        // try to print all entries?
        // writeln!(WRITER.lock(), "L4 Entry {}: {:?}", i, entry).unwrap();

        if !entry.is_unused() {
            writeln!(WRITER.lock(), "L4 Entry {}: {:?}", i, entry).unwrap();
        }
    }
```

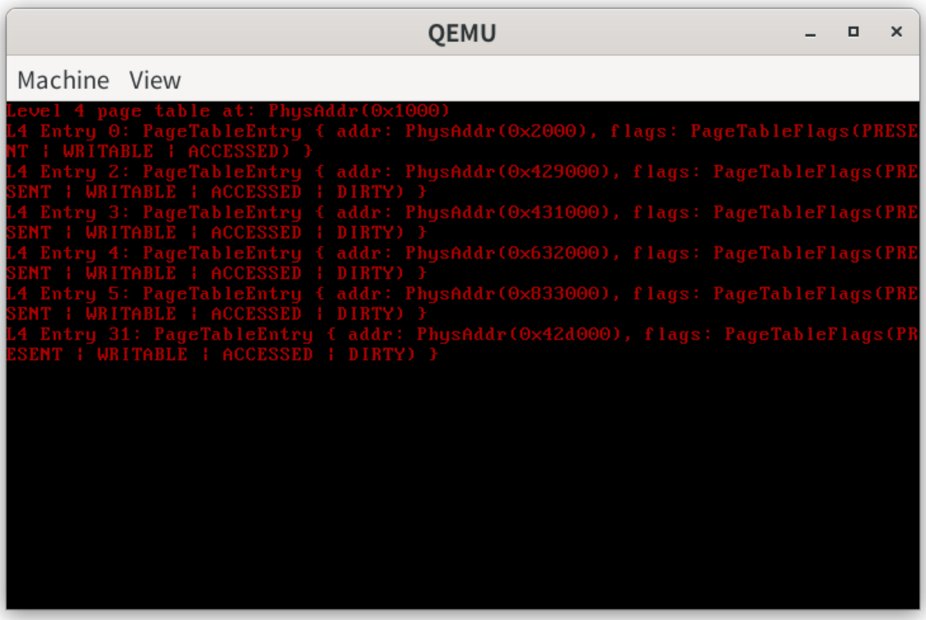
- .iter(): It is a method is used to create an iterator over a collection. It returns an iterator that allows you to iterate over the elements of the collection.

- .enumerate(): It yields tuples containing the index and the corresponding value from the original iterator.

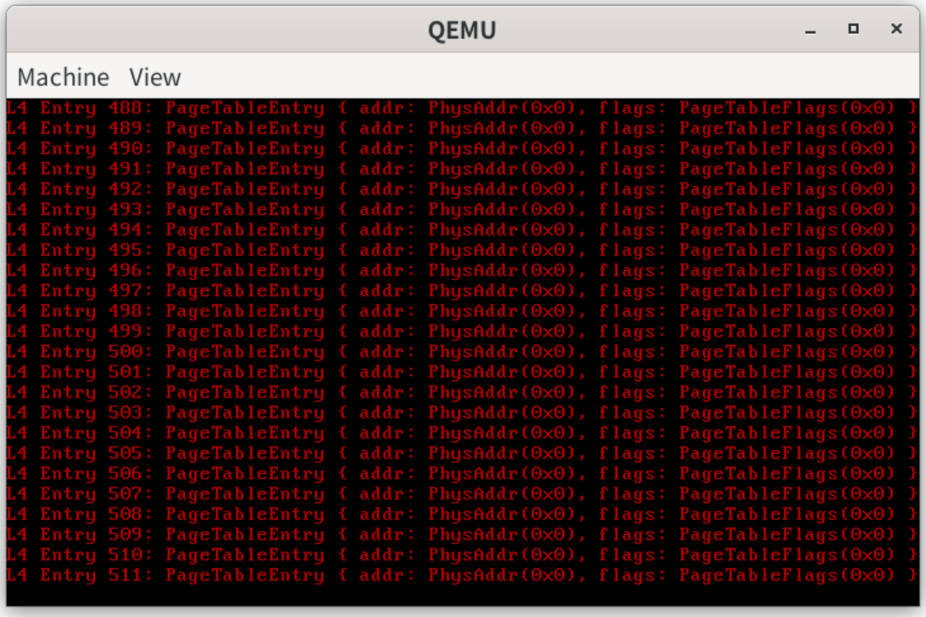
We print only non-empty entries in the page table because it helps us focus on relevant information. Printing all entries might overwhelm us with unnecessary data, making it harder to understand the page table's structure and content. However, you can print all entries if you want to experiment and inspect thoroughly.

6.3.3 Output

The output will display the physical address of the L4 page table stored in the CR3 register, along with the output of each non-empty entry in the L4 page table.



We also can have a look of all entries in L4 table



6.4 Task 2 - Traverse four-level page table

6.4.1 Introduction

In this task, we will learn how to read entries in the page table and use this information to access the next level of the page table. This task is similar to reading CR3 and generating the mapped-to physical address at a specified location. However, to maintain readability, we limit the output to only the first few non-empty entries. In the labsheet, I will demonstrate how to print the L4 and L3 page tables. You can implement the methods for printing the L2 and L1 page tables by yourself. Of course, the solution would be available at branch lab4-2.

6.4.2 Implementation

Step 1: Add limitation

Add constant to limit how many non-empty entry you want to show. We set 2 for now.

```
...
use x86_64::VirtAddr;

const DISPLAY_ENTRY: i32 = 2;
```

Then define the counter:

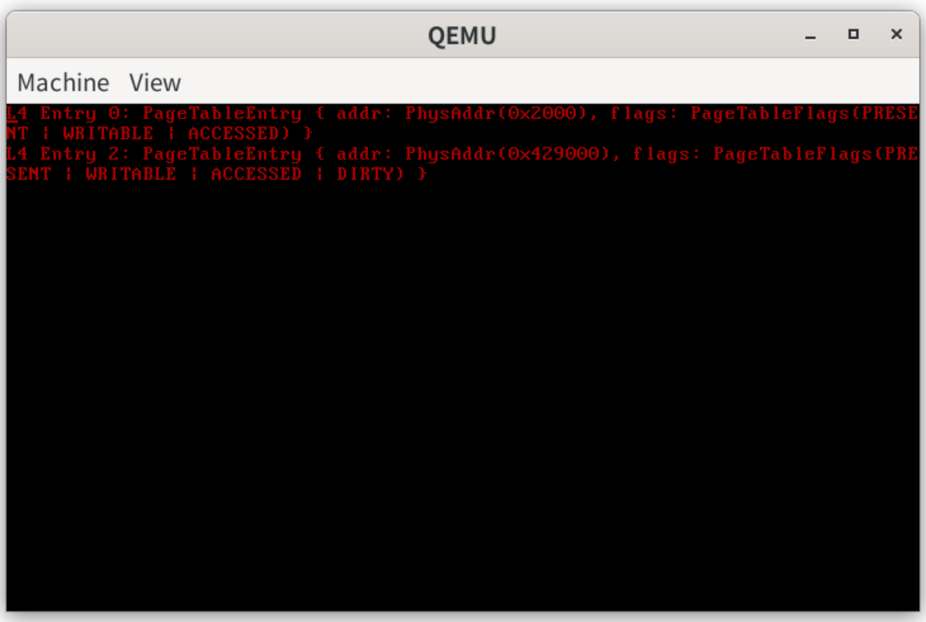
```
...
let l4_table = unsafe { &*l4_ptr };
    // counter
    let mut l4_counter = 0;
    let mut l3_counter = 0;
    for (i, entry) in l4_table.iter().enumerate() {
...
```

Step 2: Show first 2 entries of L4

In each loop, increment the counter by 1. Then check weather l4_counter is less then DISPLAY_ENTRY

```
    for (i, entry) in l4_table.iter().enumerate() {
        if !entry.is_unused() && l4_counter < DISPLAY_ENTRY {
            writeln!(WRITER.lock(), "L4 Entry {}: {:?}", i,
            entry).unwrap();
            l4_counter += 1;
        }
```

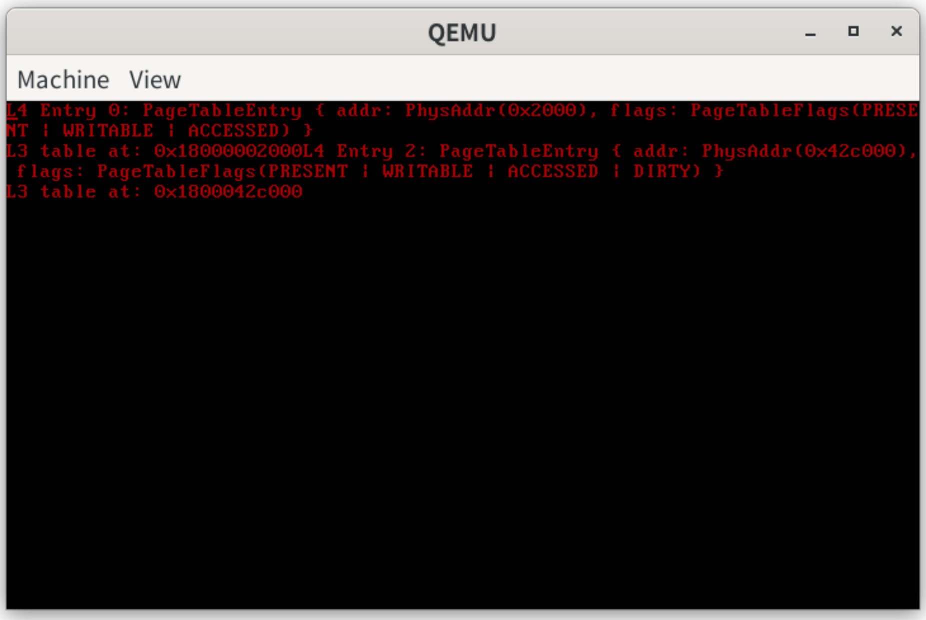
We could see the output



Step 3: Access L3 page table

To read the entries in the L4 page table, you can follow a similar approach to reading the address stored in CR3. Create a new page table that points to this physical address.

```
            let phys = entry.frame().unwrap().start_address();
            let virt = phys.as_u64() +
            boot_info.physical_memory_offset;
            let l3_ptr = VirtAddr::new(virt).as_mut_ptr();
            let l3_table: &PageTable = unsafe { &*l3_ptr };
            write!(WRITER.lock(), "L3 table at: {:p}",
            l3_ptr).unwrap();
```



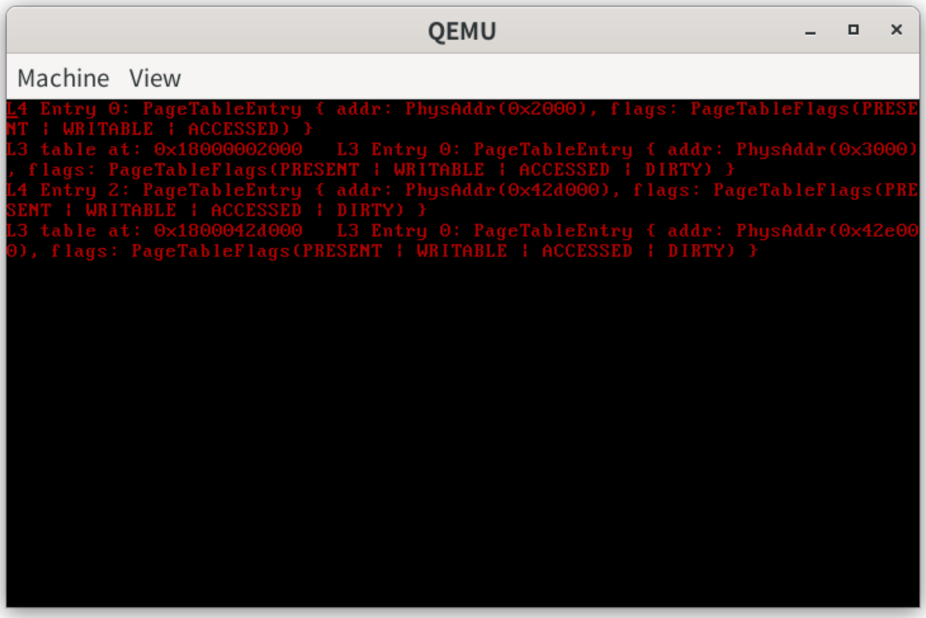
We can observe that this virtual address consists of a entry plus a physical memory offset.

Step 4: Traverse L3 page table

We access each L3 page table pointed to by the printed L4 entries. After iterating through each L3 page table, we reset the counter to zero.

```
            // print non-empty entries of the level 3 table
            for (i, entry) in l3_table.iter().enumerate() {
                if !entry.is_unused()  && l3_counter <
                DISPLAY_ENTRY{
                    writeln!(WRITER.lock(), "   L3 Entry {}:
                    {:?}", i, entry).unwrap();
```

```
                    l3_counter += 1;
                }
            }
            l3_counter = 0;
```

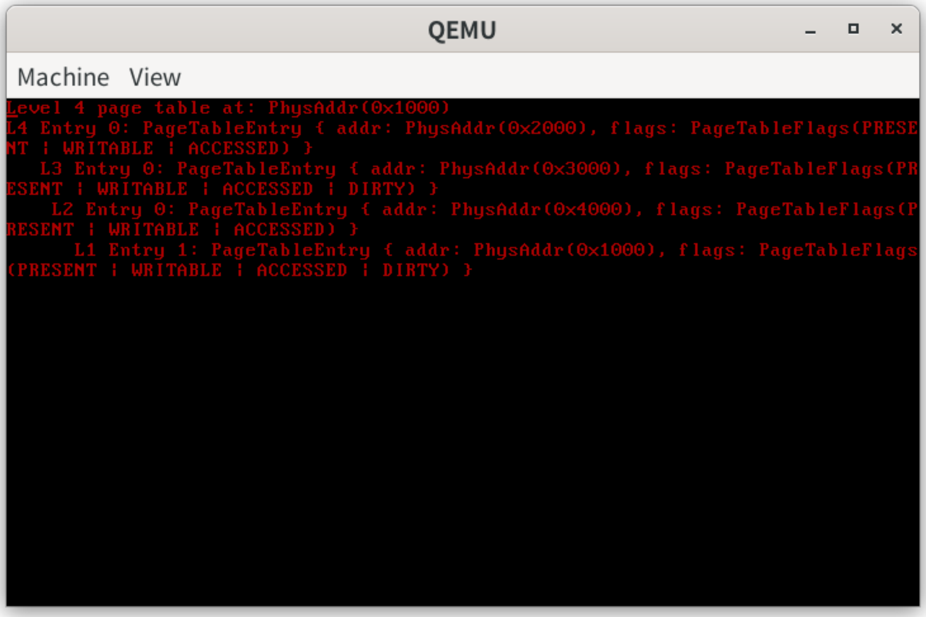


Step 5: Extend to L2 and L1

Due to space constraints and the similarity of the implementation steps to L3, we won't expand on it here.

6.4.3 Output

This is the output when we only display first entry of every level of table. For here we limit to just print the first non-empty entry.



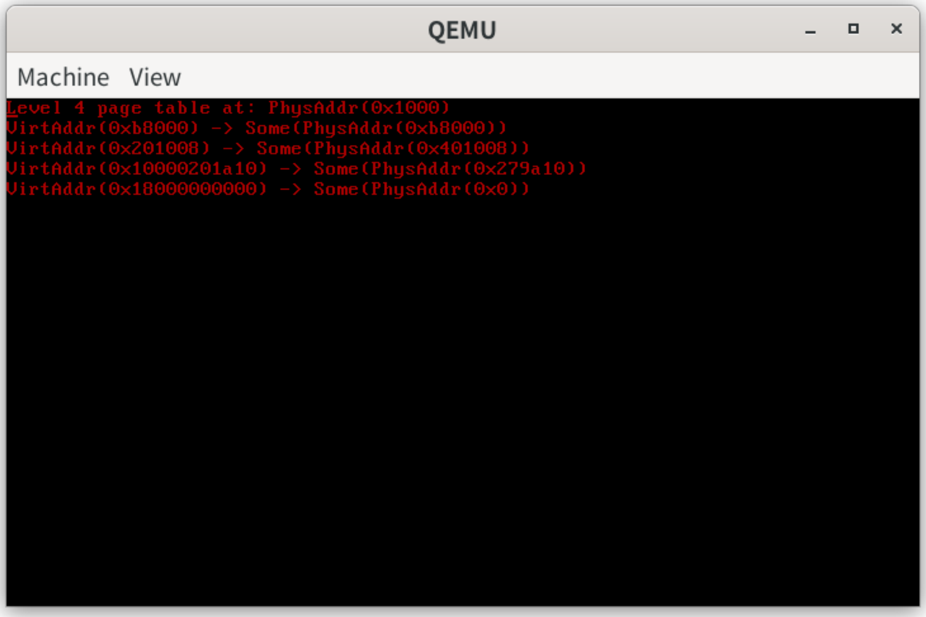
6.5 Task 3 - Address translation

6.5.1 Introduction

6.5.2 Implementation

6.5.3 Output

You would get the output of the translation from virtual address to physical address.


QEMU
Machine View
Level 4 page table at: PhysAddr(0x1000)
VirtAddr(0xb8000) -> Some(PhysAddr(0xb8000))
VirtAddr(0x201008) -> Some(PhysAddr(0x401008))
VirtAddr(0x10000201a10) -> Some(PhysAddr(0x279a10))
VirtAddr(0x18000000000) -> Some(PhysAddr(0x0))

Chapter 7

Utility - Test tool

Before using it, remember to comment

```
#[profile.dev]
#panic = "abort"
```

7.1 Lab 2

The test is available at branch test-lab2

If you want to add them to your code add this function to vga.rs. And copy and paste tests folder.

```
impl Writer {
    pub fn get_ascii(&mut self, row: usize, col: usize) -> u8 {
        self.buffer.chars[row][col].read().ascii
    }
}
```

Using command

```
cargo test --test lab2
```

And for the exception handling, make sure you provide blue color for the exception, or the test would fail.

Chapter 8

Extra knowledge

8.1 Calling convention

Calling conventions are a standardized method for functions to be implemented and called by the machine. A calling convention specifies the method that a compiler sets up to access a subroutine.[7] Here's a simple C code example demonstrating a function call using the cdecl convention:

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 5);  // Function call using cdecl
    printf("The result is: %d\n", result);
    return 0;
}
```

- Add function takes two integers as arguments and returns their sum.
- When add(3, 5) is called, 5 is pushed onto the stack first, followed by 3.
- After add returns, the main function cleans up the stack by removing the two arguments.
- The sum, which is the return value of add, is placed in the EAX register (on x86) and then stored in the result variable.

We use the C calling convention because it is widely recognized, while Rust does not yet have a stable calling convention released, as it is still under development. See here

8.2 What did bootimage tool do?

The process of booting a system is a complex topic. You can find more information about it here.

The BIOS will wake up the bootloader, which first compiles all dependencies into a standalone executable. Then, it compiles our kernel into an ELF file and jumps to the _start function to begin execution.

8.3 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 is 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							

[3]

8.4 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.

8.5 Virtual Address & Physical Address

Virtual addresses and physical addresses both serve as pointers to data storage locations, but they operate differently.
Virtual address is like a label created by the CPU for programs to use when they want to access memory. It helps programs talk to memory without worrying about where exactly things are stored physically. It can be seen as an abstraction of actual physical storage devices. Different processes can have the same virtual address, but physical addresses are unique.
Physical address corresponds to a specific location in the physical memory (RAM) of a computer, representing an actual spot on the computer's memory hardware. It serves as a direct reference to where data is stored within the computer's memory system. The conversion between virtual addresses and physical addresses is the responsibility of the operating system.

8.6 CR3 register

CR3, or Control Register 3, is a special-purpose register in the x86 architecture used for memory management. It stores the base address of the page table of the currently running process. For our case, it is the physical address for level 4 page table. When the process changed, address in CR3 also changed. It ensures isolation between processes and efficient use of system resources.

Bibliography

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

[2] Learn rust (no date) Rust Programming Language. Available at: https://www.rust-lang.org/learn (Accessed: 13 February 2024).

[3] Klabnik, S. (2014) Rust by example, Introduction - Rust By Example. Available at: https://doc.rust-lang.org/rust-by-example/ (Accessed: 13 February 2024).

[4] Rustlings (2015) rustlings. Available at: https://rustlings.cool/ (Accessed: 13 February 2024).

[5] Code page 437 (2024) Wikipedia. Available at: https://en.wikipedia.org/wiki/Code_page_437 (Accessed: 16 February 2024).

[6] Trait core::fmt::write (no date) Rust. Available at: https://doc.rust-lang.org/nightly/core/fmt/trait.Write.html (Accessed: 17 February 2024).

[7] Cary, D. (2007) X86 disassembly/calling conventions, Wikibooks, open books for an open world. Available at: https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions (Accessed: 27 February 2024).

[8] Frank, M. (2013) On x86 architecture, why are there fewer bits for virtual address space than physical?, Super User. Available at: https://superuser.com/questions/655121/on-x86-architecture-why-are-there-fewer-bits-for-virtual-address-space-than-phy (Accessed: 24 March 2024).