



中山大學  
SUN YAT-SEN UNIVERSITY

# 操作系统实验报告

## 实验零：环境搭建与实验准备

姓 名： 刘家祥  
学 号： 23336152  
教学班号： 计科二班  
专 业： 计算机科学与技术  
院 系： 计算机学院

2024~2025 学年第二学期

# 环境搭建与实验准备

## 一. 配置实验环境

### 1. 安装项目开发环境

选择在 Windows + WSL 2 平台上，使用 VSCode(Remote WSL)连接到 WSL2 进行开发、调试。首先在 Windows 下安装 WSL2: <https://github.com/microsoft/WSL/releases/download/2.4.11/wsl.2.4.11.0.x64.msi>，然后在 WSL2 中安装 Ubuntu-24.04 LTS: `wsl --install -d Ubuntu`。

#### 1.1. 使用以下命令更新 apt 源并进行软件升级：

```
sudo apt update && sudo apt upgrade
```

#### 1.2. 安装 qemu 等工具：

```
sudo apt install qemu-system-x86 build-essential gdb
```

#### 1.3. 安装 rustup：

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh  
source "$HOME/.cargo/env"
```

**注意：**本实验在 VSCode 下进行，所以也需要安装 VSCode，使用 `code` 命令打开项目文件会自动安装 Linux 版 VSCode，接下来在 Windows 本地下载安装 Remote - WSL 插件，即可畅通无阻地在 VSCode 中进行实验。

```
camellia@LAPTOP-Camellia: ~  
wsl: 检测到 localhost 代理配置, 但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。  
To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo_root" for details.  
  
camellia@LAPTOP-Camellia:~$ sudo apt update  
[sudo] password for camellia:  
Hit:1 http://security.ubuntu.com/ubuntu noble-security InRelease  
Hit:2 http://archive.ubuntu.com/ubuntu noble InRelease  
Get:3 http://archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]  
Hit:4 http://archive.ubuntu.com/ubuntu noble-backports InRelease  
Get:5 http://archive.ubuntu.com/ubuntu noble-updates/main amd64 Packages [916 kB]  
Get:6 http://archive.ubuntu.com/ubuntu noble-updates/main Translation-en [206 kB]  
Get:7 http://archive.ubuntu.com/ubuntu noble-updates/restricted amd64 Packages [729 kB]  
Get:8 http://archive.ubuntu.com/ubuntu noble-updates/restricted Translation-en [145 kB]  
Fetched 2121 kB in 5s (388 kB/s)  
Reading package lists... Done  
Building dependency tree... Done  
Reading state information... Done  
12 packages can be upgraded. Run 'apt list --upgradable' to see them.  
camellia@LAPTOP-Camellia:~$ sudo apt install qemu-x86 build-essential gdb  
Reading package lists... Done  
Building dependency tree... Done  
Reading state information... Done  
E: Unable to locate package qemu-x86  
camellia@LAPTOP-Camellia:~$ sudo apt install qemu-system-x86 build-essential gdb  
Reading package lists... Done  
Building dependency tree... Done  
Reading state information... Done  
The following additional packages will be installed:  
  acl alsa-topology-conf alsa-ucm-conf bzip2 cpp cpp-13 cpp-13-x86-64-linux-gnu cpp-x86-64-linux-gnu cpu-checker  
Setting up librados2 (19.2.0-0ubuntu0.24.04.2) ...  
Setting up g++-13-x86-64-linux-gnu (13.3.0-6ubuntu2~24.04) ...  
Setting up gcc-x86-64-linux-gnu (4:13.2.0-7ubuntu1) ...  
Setting up gstreamer1.0-plugins-good:amd64 (1.24.2-1ubuntu1.1) ...  
Setting up qemu-system-x86 (1:8.2.2+ds-0ubuntu1.6) ...  
Setting up gcc (4:13.2.0-7ubuntu1) ...  
Setting up librbdl1 (19.2.0-0ubuntu0.24.04.2) ...  
Setting up g++-x86-64-linux-gnu (4:13.2.0-7ubuntu1) ...  
Setting up qemu-block-extra (1:8.2.2+ds-0ubuntu1.6) ...  
Created symlink /etc/systemd/system/multi-user.target.wants/run-qemu.mount → /usr/lib/systemd/system/run-qemu.mount.  
Setting up g++-13 (13.3.0-6ubuntu2~24.04) ...  
Setting up qemu-system-modules-opengl (1:8.2.2+ds-0ubuntu1.6) ...  
Setting up qemu-system-gui (1:8.2.2+ds-0ubuntu1.6) ...  
Setting up g++ (4:13.2.0-7ubuntu1) ...  
update-alternatives: using /usr/bin/g++ to provide /usr/bin/c++ (c++) in auto mode  
Setting up qemu-system-modules-spice (1:8.2.2+ds-0ubuntu1.6) ...  
Setting up build-essential (12.10ubuntu1) ...  
Setting up libheif1:amd64 (1.17.6-1ubuntu4.1) ...  
Setting up libgd3:amd64 (2.3.3-9ubuntu5) ...  
Setting up libc-devtools (2.39-0ubuntu8.4) ...  
Setting up libheif-plugin-aomdec:amd64 (1.17.6-1ubuntu4.1) ...  
Setting up libheif-plugin-libde265:amd64 (1.17.6-1ubuntu4.1) ...  
Setting up libheif-plugin-aomenc:amd64 (1.17.6-1ubuntu4.1) ...  
Processing triggers for libc-bin (2.39-0ubuntu8.4) ...  
Processing triggers for man-db (2.12.0-4build2) ...  
Processing triggers for libglib2.0-0t64:amd64 (2.80.0-6ubuntu3.2) ...  
Processing triggers for hicolor-icon-theme (0.17-2) ...  
camellia@LAPTOP-Camellia:~$
```

成功更新 apt 源并安装软件包、安装了 qemu 等工具。

```
camellia@LAPTOP-Camellia: ~ × + ▾
18.2 MiB / 18.2 MiB (100 %) 3.6 MiB/s in 4s
info: installing component 'rust-std'
26.7 MiB / 26.7 MiB (100 %) 8.4 MiB/s in 3s
info: installing component 'rustc'
69.5 MiB / 69.5 MiB (100 %) 7.8 MiB/s in 8s
info: installing component 'rustfmt'
info: default toolchain set to 'stable-x86_64-unknown-linux-gnu'

stable-x86_64-unknown-linux-gnu installed - rustc 1.85.0 (4d91de4e4 2025-02-17)

Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload your PATH environment variable to include
Cargo's bin directory ($HOME/.cargo/bin).

To configure your current shell, you need to source
the corresponding env file under $HOME/.cargo.

This is usually done by running one of the following (note the leading DOT):
. "$HOME/.cargo/env" # For sh/bash/zsh/ash/dash/pdksh
source "$HOME/.cargo/env.fish" # For fish
source "$HOME/.cargo/env.nu" # For nushell
camellia@LAPTOP-Camellia:~$ source "$HOME/.cargo/env"
camellia@LAPTOP-Camellia:~$ rustc --version
rustc 1.85.0 (4d91de4e4 2025-02-17)
camellia@LAPTOP-Camellia:~$ cargo --version
cargo 1.85.0 (d73d2caf9 2024-12-31)
camellia@LAPTOP-Camellia:~$
```

成功安装 rustup。

```
camellia@LAPTOP-Camellia:~$ rustc --version
rustc 1.85.0 (4d91de4e4 2025-02-17)
camellia@LAPTOP-Camellia:~$ rustc +nightly --version
rustc 1.87.0-nightly (f5a1ef712 2025-03-07)
camellia@LAPTOP-Camellia:~$ qemu-system-x86_64 --version
QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.6)
Copyright (c) 2003-2023 Fabrice Bellard and the QEMU Project developers
camellia@LAPTOP-Camellia:~$ gcc --version
gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
camellia@LAPTOP-Camellia:~$ gdb --version
GNU gdb (GDB) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
camellia@LAPTOP-Camellia:~$
```

使用命令行检查软件包版本得到上述结果：所需工具已经基本安装并且版本达到实验需求。

注意：由于在 0x01 中使用 gdb15.0 会出现一些非必要的错误信息，又因 Ubuntu24.04 无 gdb12.1 安软件包，遂使用下载源码的方式更换为 gdb12.1

## 二. 尝试使用 Rust 进行编程

### 1. 使用 Rust 编写一个程序，完成以下任务：

#### 1.1. 创建一个函数 `count_down(seconds: u64)`

- 该函数接收一个 `u64` 类型的参数，表示倒计时的秒数。
- 函数应该每秒输出剩余的秒数，直到倒计时结束，然后输出 `Countdown finished!`。

```
fn count_down(seconds: u64) {  
    for i in (1..=seconds).rev() {  
        println!("{}", i);  
        sleep(Duration::from_secs(1));  
    }  
    println!("Countdown finished!");  
}
```

#### 1.2. 创建一个函数 `read_and_print(file_path: &str)`

- 该函数接收一个字符串参数，表示文件的路径。
- 函数应该尝试读取并输出文件的内容。如果文件不存在，函数应该使用 `expect` 方法主动 panic，并输出 `File not found!`。

```
fn read_and_print(file_path: &str) → io::Result<()> {  
    let mut file = File::open(file_path)?;  
    let mut contents = String::new();  
    file.read_to_string(&mut contents)?;  
    println!("{}", contents);  
    Ok(())  
}
```

#### 1.3. 创建一个函数 `file_size(file_path: &str) → Result<u64, &str>`

- 该函数接收一个字符串参数，表示文件的路径，并返回一个 `Result`。
- 函数应该尝试打开文件，并在 `Result` 中返回文件大小。如果文件不存在，函数应该返回一个包含 `File not found!` 字符串的 `Err`。

```
fn file_size(file_path: &str) → Result<u64, &str> {
    let file = File::open(file_path).map_err(|_| "File not found!")?;
    let metadata = file.metadata().map_err(|_| "Could not read metadata!")?;
    Ok(metadata.len())
}
```

#### 1.4. 在 main 函数中，按照如下顺序调用上述函数：

- 首先调用 `count_down(5)` 函数进行倒计时
- 然后调用 `read_and_print("/etc/hosts")` 函数尝试读取并输出文件内容
- 最后使用 `std::io` 获取几个用户输入的路径，并调用 `file_size` 函数尝试获取文件大小，并处理可能的错误。

**注意：** 此处给出的 main 函数为完整的 main 函数，包含了对后续函数的调用。

```
fn main() → io::Result<()> {
    // 调用 terminal_colors 模块中的函数
    terminal_colors::print_colored_messages();

    // 调用 shape 模块中的函数
    let rectangle = shape::Shape::Rectangle {
        width: 10.0,
        height: 20.0,
    };
    let circle = shape::Shape::Circle { radius: 10.0 };

    println!("Rectangle area: {}", rectangle.area());
    println!("Circle area: {}", circle.area());

    // 调用 unique_id 模块中的函数
    let id1 = unique_id::UniqueId::new();
    let id2 = unique_id::UniqueId::new();
    println!("ID1: {}, ID2: {}", id1.value(), id2.value());
    println!("ID1 == ID2: {}", id1 == id2);

    // 调用 count_down 函数进行倒计时
    count_down(5);

    // 调用 read_and_print 函数读取并输出文件内容
    match read_and_print("/etc/hosts") {
        Ok(_) ⇒ println!("File content displayed successfully."),
        Err(e) ⇒ println!("Error reading file: {}", e),
    }
}
```

```

// 获取用户输入的文件路径，并调用 file_size 函数
println!("\n请输入要检查的文件路径（每行一个，输入空行结束）");
let mut buffer = String::new();

loop {
    buffer.clear();
    io::stdin().read_line(&mut buffer)?;

    let path = buffer.trim();
    if path.is_empty() {
        break;
    }

    match file_size(path) {
        Ok(size) => {
            let (human_size, unit) = humanized_size(size);
            println!("文件 {} 的大小为: {} 字节 {:.4} {}",
                    path, size, human_size, unit);
        },
        Err(e) => println!("错误: {}", e),
    }
}

Ok(())
}

```

## 2. 实现一个进行字节数转换的函数，并格式化输出：

### 2.1. 实现函数 `humanized_size(size: u64) → (f64, &'static str)` 将字节数转换为人类可读的大小和单位

- 使用 1024 进制，并使用二进制前缀（B, KiB, MiB, GiB）作为单位

### 2.2. 补全格式化代码，使得你的实现能够通过如下测试：

```

#[test]
fn test_humanized_size() {
    let byte_size = 1554056;
    let (size, unit) = humanized_size(byte_size);
    assert_eq!("Size : 1.4821 MiB", format!("{:.*} {}", size, unit));
}

```

- 字节转换函数：

```
// 添加新函数：将字节数转换为人类可读的格式
fn humanized_size(size: u64) → (f64, &'static str) {
    const UNITS: [&str; 4] = ["B", "KiB", "MiB", "GiB"];

    if size == 0 {
        return (0.0, UNITS[0]);
    }

    let base = 1024.0;
    // 计算适合的单位序号（对数以1024为底）
    let exponent = (size as f64).log(base).floor() as usize;

    // 限制最大单位为 GiB
    let exponent = exponent.min(UNITS.len() - 1);

    // 计算转换后的值
    let converted_size = size as f64 / base.powi(exponent as i32);

    (converted_size, UNITS[exponent])
}
```

- 补全后的格式化代码：

```
#[test]
fn test_humanized_size() {
    let byte_size = 1554056;
    let (size, unit) = humanized_size(byte_size);
    assert_eq!("Size : 1.4821 MiB", format!("Size : {: >7.4} {}", size, unit));
}
```

- 测试代码

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_humanized_size() {
        let byte_size = 1554056;
        let (size, unit) = humanized_size(byte_size);
        assert_eq!("Size : 1.4821 MiB", format!("Size : {: >7.4} {}", size,
unit));
    }
}
```



```

#[test]
fn test_various_sizes() {
    // 测试字节
    let (size, unit) = humanized_size(500);
    assert_eq!(500.0, size);
    assert_eq!("B", unit);

    // 测试KB
    let (size, unit) = humanized_size(2048);
    assert_eq!(2.0, size);
    assert_eq!("KiB", unit);

    // 测试MB
    let (size, unit) = humanized_size(3 * 1024 * 1024);
    assert_eq!(3.0, size);
    assert_eq!("MiB", unit);

    // 测试GB
    let (size, unit) = humanized_size(5 * 1024 * 1024 * 1024);
    assert_eq!(5.0, size);
    assert_eq!("GiB", unit);
}
}

```

### 3. 利用现有的 crate 在终端中输出彩色的文字

输出一些带有颜色的字符串，并尝试直接使用 `print!` 宏输出一到两个相同的效果。  
尝试输出如下格式和内容：

- `INFO: Hello, world!`，其中 `INFO` 为绿色，后续内容为白色
- `WARNING: I'm a teapot!`，颜色为黄色，加粗，并为 `WARNING` 添加下划线
- `ERROR: KERNEL PANIC!!!`，颜色为红色，加粗，并尝试让这一行在控制行窗口居中
- 一些你想尝试的其他效果和内容……(暂无)

**注意：** 考虑到报告篇幅以及实验重点，此处不再给出 `terminal_colors.rs`、`unique_id.rs`、`shape.rs` 三个文件的代码，仅展示测试结果。

- 在项目文件目录下执行 `cargo run` 得到如下结果：

```
问题 输出 调试控制台 终端 窗口 评论
+ target/debug/os_lab0 - os_lab0 ... X
camellia@LAPTOP-Camellia:~/os_lab0$ cargo run
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.14s
Running `target/debug/os_lab0`
INFO: Hello, world!
WARNING: I'm a teapot!
ERROR: KERNEL PANIC!!!
直接打印: 这是青色文字
多彩文字:
背景高亮文字
闪烁效果
反相效果
多种样式组合
Rectangle area: 200
Circle area: 314.1592653589793
ID1: 0, ID2: 1
ID1 == ID2: false
5
4
3
2
1
Countdown finished!
# This file was automatically generated by WSL. To stop automatic generation of this file, add the following entry to /etc/wsl.conf:
# [network]
# generateHosts = false
127.0.0.1      localhost
127.0.1.1     LAPTOP-Camellia.    LAPTOP-Camellia

# The following lines are desirable for IPv6 capable hosts
::1          ip6-localhost ip6-loopback
fe80::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters

File content displayed successfully.

请输入要检查的文件路径（每行一个，输入空行结束）：

```

- 可以看出顺利输出了倒计时、文件内容、文件大小等信息，同时也展示了彩色文字的效果。

## 三. 运行 UEFI Shell

### 1. 初始化仓库

#### 1.1. 克隆实验仓库到本地

```
$ git clone https://github.com/YatSen0S/YatSen0S-Tutorial-Volume-2
```

#### 1.2. 参考参考实验 0x00 代码的文件结构，初始化你的仓库。

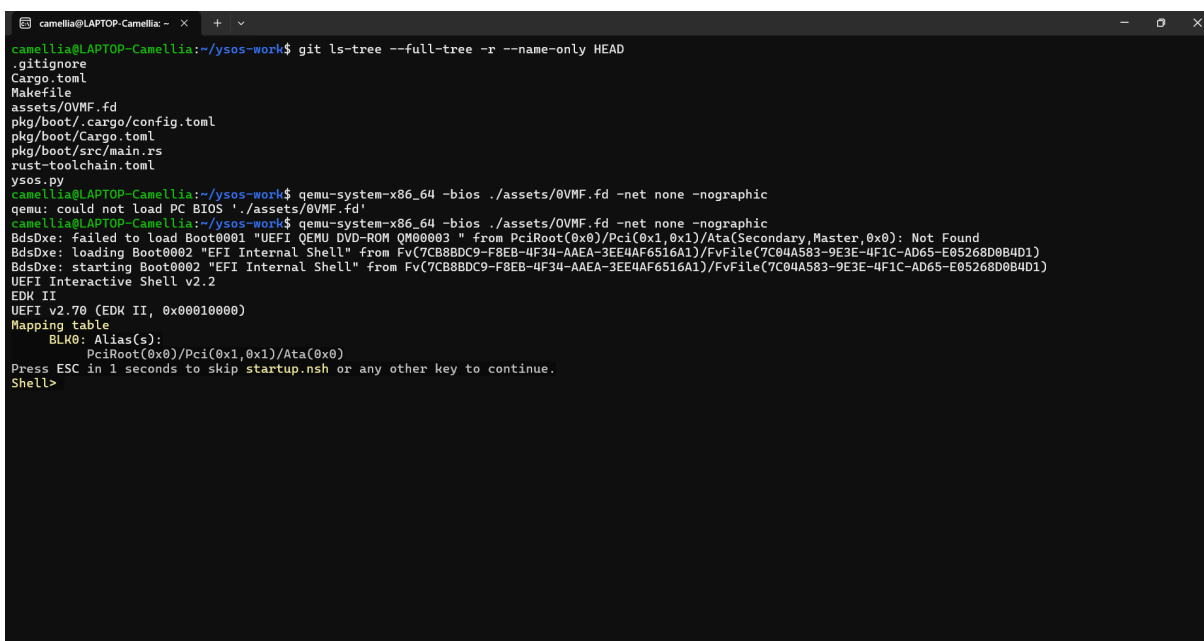
在用户目录下创建一个文件夹 `ysos-0x00`，并将 `YatSen0S-Tutorial-Volume-2/src/0x00` 中的内容复制到该目录中。

#### 1.3. 初始化仓库

```
$ cd ~/ysos-0x00
$ git init
$ git add .
$ git commit -m "init"
```

## 1.4. 检验文件是否完整

```
$ git ls-tree --full-tree -r --name-only HEAD
.gitignore
Cargo.toml
Makefile
assets/OVMF.fd
pkg/boot/.cargo/config
pkg/boot/Cargo.toml
pkg/boot/src/main.rs
rust-toolchain.toml
ysos.py
```



```
camellia@LAPTOP-Camellia: ~  
camellia@LAPTOP-Camellia:~/ysos-work$ git ls-tree --full-tree -r --name-only HEAD  
.gitignore  
Cargo.toml  
Makefile  
assets/OVMF.fd  
pkg/boot/.cargo/config.toml  
pkg/boot/Cargo.toml  
pkg/boot/src/main.rs  
rust-toolchain.toml  
ysos.py  
camellia@LAPTOP-Camellia:~/ysos-work$ qemu-system-x86_64 -bios ./assets/OVMF.fd -net none -nographic  
qemu: could not load PC BIOS './assets/OVMF.fd'  
camellia@LAPTOP-Camellia:~/ysos-work$ qemu-system-x86_64 -bios ./assets/OVMF.fd -net none -nographic  
BdsDxe: failed to load Boot0001 "UEFI QEMU DVD-ROM QM00003 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Secondary,Master,0x0): Not Found  
BdsDxe: loading Boot0002 "EFI Internal Shell" from Fv(7CB8BDC9-F8EB-4F34-AAEA-3EE4AF6516A1)/FvFile(7C04A583-9E3E-4F1C-AD65-E05268D0B4D1)  
BdsDxe: starting Boot0002 "EFI Internal Shell" from Fv(7CB8BDC9-F8EB-4F34-AAEA-3EE4AF6516A1)/FvFile(7C04A583-9E3E-4F1C-AD65-E05268D0B4D1)  
UEFI Interactive Shell v2.2  
EDK II  
UEFI v2.70 (EDK II, 0x00010000)  
Mapping table  
BLK0: Alias(s):  
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)  
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.  
Shell>
```

- 可以看出文件完整，已经成功初始化仓库。

## 2. 使用 QEMU 启动 UEFI Shell

```
qemu-system-x86_64 -bios ./assets/OVMF.fd -net none -nographic
```



# 1. 了解现代操作系统(Windows)的启动过程，UEFI 和 Legacy(BIOS)的区别是什么？

Windows启动过程：上电自检、固件初始化、引导加载程序、内核加载、驱动程序和系统服务启动和用户环境初始化。

## 1.1. 架构和功能

– Legacy BIOS:

- 诞生较早，基于 16 位实模式运行。
- 功能简单，主要任务是初始化硬件并加载操作系统。
- 使用 MBR 分区表，最多支持 4 个主分区，硬盘容量限制为 2TB。
- 不支持图形界面，启动时通常是纯文本模式。

– UEFI:

- 现代标准，基于 32 位或 64 位保护模式运行。
- 功能强大，支持复杂的预启动环境(如网络启动、安全检查、图形界面)。
- 使用 GPT 分区表，支持 128 个分区，硬盘容量无限制。
- 支持图形界面，启动时可以显示图形界面，甚至可以在启动前运行小型应用程序。

## 1.2. 启动方式

– Legacy BIOS:

- 通过 MBR 引导，加载引导程序。
- 启动过程较慢，因为需要从磁盘的固定位置读取数据。

– UEFI:

- 直接加载 EFI 文件(存储在 FAT32 格式的 EFI 系统分区中)。
- 启动速度更快，(因为)支持并行加载和优化。

## 1.3. 安全性

– Legacy BIOS:

- 缺乏内置安全机制，容易受到低级恶意软件如(Bootkit)的攻击。

– UEFI:

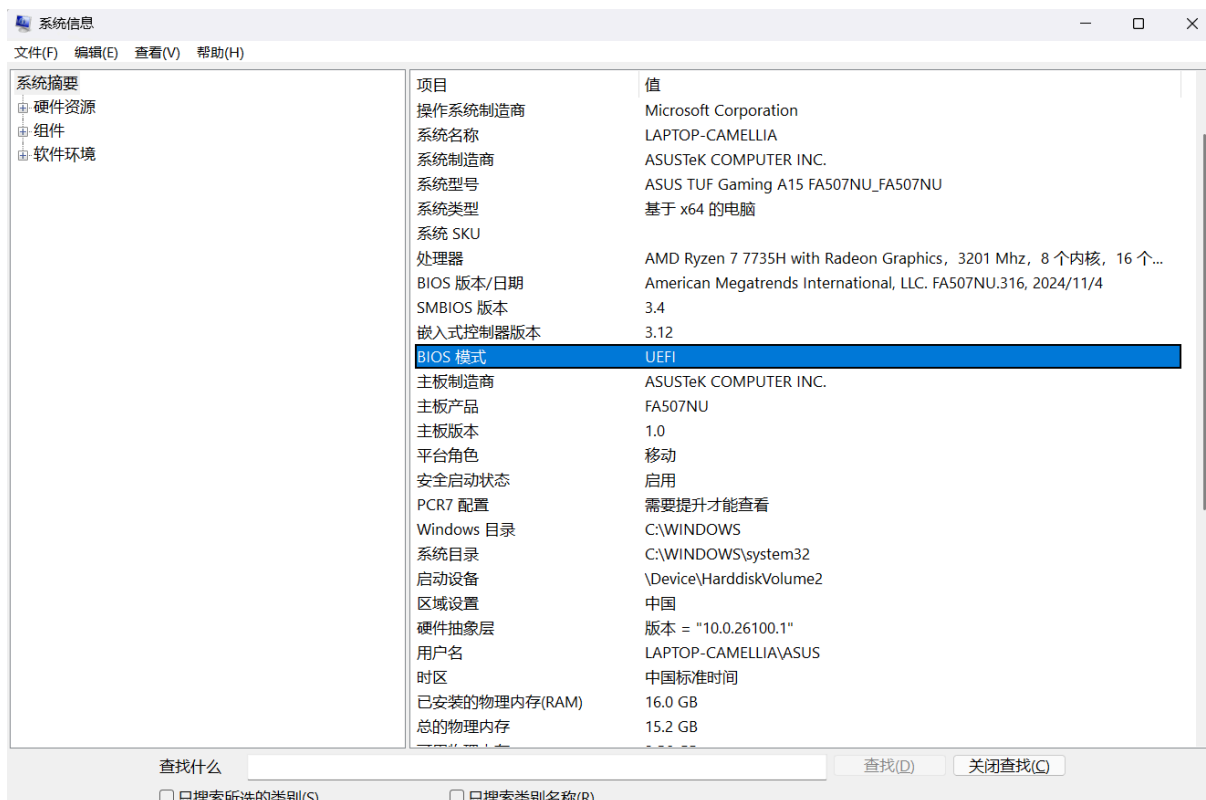
- 支持 Secure Boot(安全启动)，通过数字签名验证引导加载程序和操作系统的完整性，防止未经授权的代码运行。

## 1.4. 兼容性

– Legacy BIOS:

- 只支持传统的操作系统(如早期的 Windows XP), 无法直接启动依赖 UEFI 的操作系统。
- UEFI:
  - 向下兼容 Legacy BIOS(通过 CSM), 但现代操作系统(Windows 11)要求禁用 CSM 并使用纯 UEFI 模式。

**注意:** 在自己电脑上使用 Win + R, 输入 msinfo32, 查看“系统信息”, 在“BIOS”模式一项中会显示”UEFI “或” Legacy “。



## 2. 尝试解释 Makefile 中的命令做了哪些事情？或许你可以参考下列命令来得到更易读的解释：

```
python ysos.py run --dry-run
```

- 条件逻辑, 如果模式是 `release` , 添加 `--release` 参数

```
ifeq (${MODE}, release)
    BUILD_ARGS := --release # 如果模式是 release, 添加 --release 参数
endif
```

- 依赖关系链

```
// run依赖build
run: build launch

// build依赖$(ESP)$
build: $(ESP)

// $(ESP)依赖于$(ESP)/EFI/BOOT/BOOTX64.EFI
$(ESP): $(ESP)/EFI/BOOT/BOOTX64.EFI

// $(ESP)/EFI/BOOT/BOOTX64.EFI依赖于target/x86_64-unknown-uefi/$(MODE)/
ysos_boot.efi
$(ESP)/EFI/BOOT/BOOTX64.EFI: target/x86_64-unknown-uefi/$(MODE)/ysos_boot.efi

// target/x86_64-unknown-uefi/$(MODE)/ysos_boot.efi依赖于boot目录
target/x86_64-unknown-uefi/$(MODE)/ysos_boot.efi: pkg/boot
```

## 2.1. make build

根据上述依赖链，首先执行 `cd pkg/boot && cargo build $(BUILD_ARGS)`，编译 UEFI 应用。然后创建必要的 EFI 引导目录结构，将编译好的 EFI 文件复制到 UEFI 引导加载程序的标准位置。

## 2.2. make launch

启动 QEMU 虚拟机：

- 使用 OVMF 作为 UEFI 固件
- 禁用网络
- 加载配置的内存参数
- 使用 `nographic` 模式，不显示图形界面
- 将 ESP 目录映射为 FAT 格式的启动磁盘
- 启用快照模式(不保存对磁盘的更改)

## 2.3. make intdbg 和 make debug

提供两种不同的调试模式：

- `intdbg`：启用中断和 CPU 重置的调试输出( `-no-reboot -d int,cpu_reset` )
- `debug`：启用 GDB 远程调试功能( `-s -S` )，可以使用 GDB 连接到端口 1234

## 2.4. make clean

```
clean:
    @cargo clean
```

- 清理所有 Cargo 生成的构建文件。

## 2.5. 构建流程总结

- 当执行 `make run` 时：首先出发 `build`，编译 UEFI 应用，然后创建 EFI 引导目录结构，将编译好的 EFI 文件复制到 UEFI 引导加载程序的标准位置，最后执行 `launch` 目标启动 QEMU。

## 3. 利用 cargo 的包管理和 docs.rs 的文档，我们可以很方便的使用第三方库。这些库的源代码在哪里？它们是什么时候被编译的？

### 3.1. 第三方库的来源

#### 3.1.1. 默认来源: crates.io

当你在项目的 `Cargo.toml` 文件中添加一个依赖，例如 `serde = "1.0"`，Cargo 会从 crates.io 下载对应版本的 `serde` crate 的源代码。crates.io 托管了这些 crate 的源代码及其元数据。

#### 3.1.2. 自定义来源

crate 的作者也可以选择将源代码托管在其他地方，比如 GitHub、GitLab 或其他代码仓库。在这种情况下，他们会在 crates.io 上发布 crate 的元数据，并在 `Cargo.toml` 中指定源代码的实际位置（例如一个 Git 仓库的 URL）。Cargo 会根据这些信息从指定的位置下载源代码。

**注意：**因此，第三方库的源代码通常要么直接托管在 crates.io 上，要么由 crates.io 指向的外部位置（比如 GitHub）提供。

### 3.2. 第三方库什么时候被编译

第三方库的编译是由 Cargo 在构建项目时完成的，具体时间取决于一下几个因素：

#### 3.2.1. 运行 `cargo build` 时

当你运行 `cargo build` 时，Cargo 会检查项目的 `Cargo.toml` 文件，下载并编译所有依赖的 crate。这些 crate 会被编译成 `.rlib` 文件，然后链接到你的项目中。

#### 3.2.2. 本地缓存的影响

Cargo 会将下载的源代码和编译后的结果缓存到本地（默认位于 `~/.cargo` 目录下）。如果某个 crate 的版本已经下载并编译过，且没有发生变化，Cargo 会直接使用缓存中的结果，而不是重新编译。这样可以显著加快构建速度。



### 3.2.3. 版本更新触发重新编译

如果你更新了依赖的版本（例如将 `serde = "1.0"` 改为 `serde = "1.1"` ），或者某个 crate 的源代码有新版本发布且你的 `Cargo.toml` 允许使用最新版本（例如 `serde = "^1.0"` ），Cargo 会在下次构建时重新下载源代码并编译。

**注意：**因此，第三方库的编译时间取决于你运行 `cargo build` 的具体时刻，以及依赖的版本是否需要更新或重新编译。

## 4. 为什么我们需要使用 `#[entry]` 而不是直接使用 `main` 函数作为程序的入口？

### 4.1. `no_std` 和 `no_main` 环境

代码使用了 `#![no_std]` 和 `#![no_main]` 属性，表明：

- 不使用 Rust 标准库（无法访问普通的运行时环境）
- 不使用常规的 `main` 函数入口点机制

### 4.2. UEFI 特定的接口要求

UEFI 规范要求 UEFI 应用程序必须实现特定的入口点函数 `efi_main` ，而不是常规的 `main` 函数。这个函数的签名和行为与 `main` 函数不同，需要使用 `#[entry]` 宏来标记。

### 4.3. 底层环境初始化

`#[entry]` 属性宏（来自 UEFI-rs 库）会：

- 确保函数符合 UEFI 规范要求的格式
- 处理 UEFI 环境的初始化工作
- 管理与 UEFI 固件的通信
- 提供正确的调用约定和参数传递

### 4.4. 直接与固件交互

作为引导加载程序，这段代码需要直接与 UEFI 固件交互，而不是通过操作系统提供的抽象。`#[entry]` 宏帮助建立这种直接的交互机制。

**注意：**总的来说，`#[entry]` 是一种特殊的入口点声明方式，专为像 UEFI 引导加载程序这样的裸机环境设计，确保程序能够在没有操作系统支持的情况下正确初始化和运行。

## 六. 加分项

### 1. 😊 基于控制行颜色的 Rust 编程题目，参考 log crate 的文档，为不同的日志级别输出不同的颜色效果，并进行测试输出。

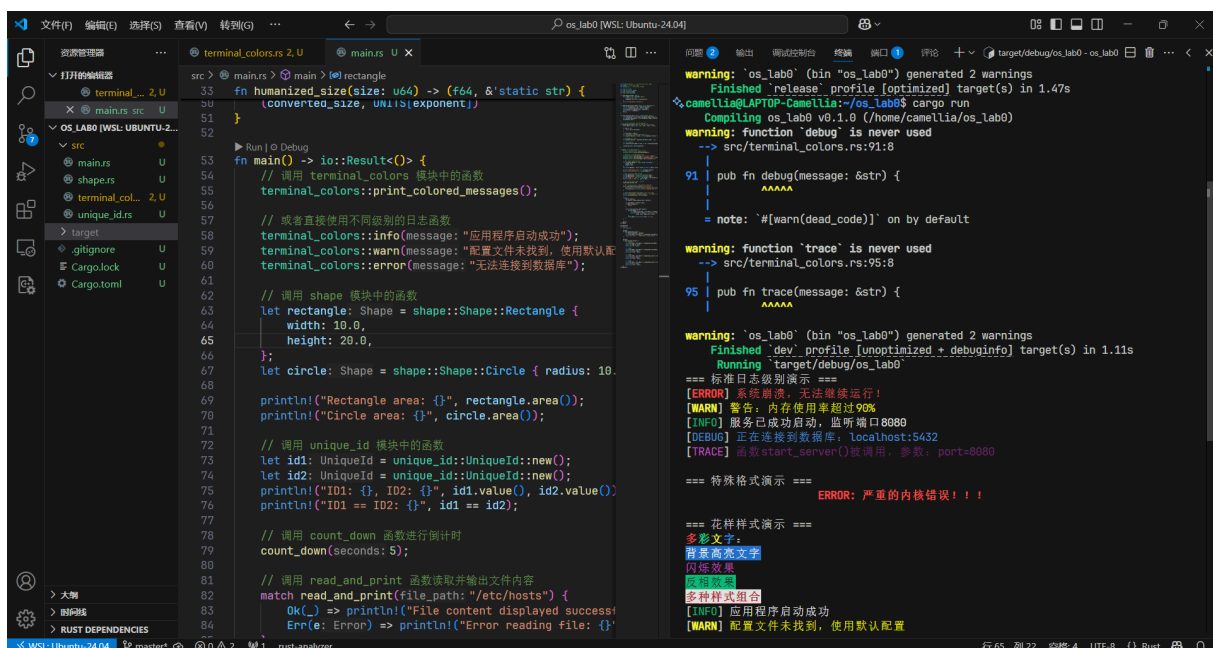
- 增加了日志级别枚举 - 使用标准的 ERROR、WARN、INFO、DEBUG、TRACE 日志级别
- 创建了统一的日志打印函数 - 根据日志级别应用不同的颜色和样式
- 添加了便利函数 - 可以直接用 error()、warn() 等函数快速打印日志
- 保留了原有的特殊样式
- 保留了居中显示错误和其他花样效果的演示

增加了测试

- 添加了测试代码验证所有功能

```
/// 日志级别枚举
#[derive(Debug, PartialEq, Eq, Clone, Copy)]
pub enum LogLevel {
    ERROR,
    WARN,
    INFO,
    DEBUG,
    TRACE,
}
```

其他代码略



可以看出成功实现了分级日志打印，不同级别的日志有不同的颜色和样式。

## 2. 🤔 基于第一个 Rust 编程题目，实现一个简单的 shell 程序：

- 实现 cd 命令，可以切换当前工作目录（可以不用检查路径是否存在）

```
// 执行cd命令
pub fn cmd_cd(args: &[&str]) → io::Result<()> {
    if args.is_empty() {
        // 如果没有参数，则切换到用户主目录
        let home = env::var("HOME").unwrap_or_else(|_| String::from("/"));
        env::set_current_dir(home)?;
    } else {
        // 切换到指定目录
        env::set_current_dir(args[0])?;
    }

    // 打印当前目录
    let current_dir = env::current_dir()?;
    terminal_colors::info(&format!("当前目录: {}", current_dir.display()));

    Ok(())
}
```

- 实现 ls 命令，尝试列出当前工作目录下的文件和文件夹，以及有关的信息（如文件大小、创建时间等）

```
// 执行ls命令
pub fn cmd_ls(args: &[&str]) → io::Result<()> {
    let path = if args.is_empty() {
        PathBuf::from(".")
    } else {
        PathBuf::from(args[0])
    };

    let entries = fs::read_dir(path)?;

    println!("{:<30} {:<10} {:<20} {}", "名称", "大小", "修改时间", "类型");
    println!("{:<30} {:<10} {:<20} {}", "----", "----", "-----", "----");
}
```

```

for entry_result in entries {
    let entry = entry_result?;
    let metadata = entry.metadata()?;

    let file_name = entry.file_name();
    let file_name_str = file_name.to_string_lossy();

    let file_size = format_file_size(&metadata);

    let modified_time = metadata
        .modified()?
        .duration_since(UNIX_EPOCH)
        .map(|d| format_time(UNIX_EPOCH + d))
        .unwrap_or_else(|_| String::from("未知"));

    let file_type = if metadata.is_dir() {
        "目录"
    } else if metadata.is_file() {
        "文件"
    } else if metadata.is_symlink() {
        "符号链接"
    } else {
        "其他"
    };

    println!("{:<30} {:<10} {:<20} {}", file_name_str, file_size,
modified_time, file_type);
}

Ok(())
}

```

- 实现 cat 命令，输出某个文件的内容

```

pub fn cmd_cat(args: &[&str]) → io::Result<()> {
    // 检查是否提供了文件路径参数
    if args.is_empty() {
        terminal_colors::error("请指定要查看的文件");
        return Ok(());
    }

    // 遍历并显示每个文件的内容
    for file_path in args {
        terminal_colors::info(&format!("≡≡≡ 文件内容: {} ≡≡≡", file_path));
        match read_and_print(file_path) {
            Ok(_) => {},
            Err(e) => terminal_colors::error(&format!("无法读取文件 {}: {}",
file_path, e)),
        }
        println!();
    }

    Ok(())
}

```

#### · Shell 主循环

```

pub fn run_shell() → io::Result<()> {
    terminal_colors::info("欢迎使用简易Shell! 输入'exit'或'quit'退出");

    let mut input = String::new();

    loop {
        // 显示提示符
        let current_dir = env::current_dir()?.to_string_lossy().to_string();
        print!("\n{}> ", current_dir);
        io::stdout().flush()?;

        // 读取用户输入
        input.clear();
        io::stdin().read_line(&mut input)?;
    }
}

```

```

// 解析命令和参数
let input = input.trim();
if input.is_empty() { continue; }

let parts: Vec<&str> = input.split_whitespace().collect();
let command = parts[0].to_lowercase();
let args = &parts[1..];

// 根据命令执行不同的操作
match command.as_str() {
    "cd" => { /* 执行cd命令 */ },
    "ls" => { /* 执行ls命令 */ },
    "cat" => { /* 执行cat命令 */ },
    "exit" | "quit" => {
        terminal_colors::info("退出shell...");
        break;
    },
    _ => {
        terminal_colors::error(&format!("未知命令: {}", command));
        terminal_colors::info("支持的命令: cd, ls, cat, exit");
    }
}

Ok(())
}
}

```

运行结果如下:

The screenshot shows a Rust IDE with the following components:

- Source Code (Left Panel):** Displays the Rust code for a simple shell. It includes functions for parsing commands, executing 'cd', 'ls', and 'cat', and a main loop that runs the shell.
- Terminal (Right Panel):** Shows the output of the program. It starts with a welcome message, followed by the execution of 'ls' (listing files in the current directory), 'cd src' (changing to the src directory), and 'cat shell.rs' (displaying the contents of the shell.rs file).

The terminal output is as follows:

```

=== 启动简易Shell ===
[INFO] 欢迎使用简易Shell! 输入'exit'或'quit'退出

/home/camellia/os_lab0> ls
名称      类型      大小      修改时间
----
Cargo.toml  96B      2025-03-21 02:08:38 文件
Cargo.lock 2.63KiB  2025-03-21 02:08:55 文件
target     4.00KiB  2025-04-01 09:24:00 目录
.git       4.00KiB  2025-03-23 09:02:41 目录
.gitignore 8B       2025-03-19 03:48:43 文件
src        4.00KiB  2025-04-01 11:37:26 目录

/home/camellia/os_lab0> cd src
[INFO] 当前目录: /home/camellia/os_lab0/src

/home/camellia/os_lab0/src> ls
名称      类型      大小      修改时间
----
shape.rs  869B     2025-03-21 02:20:47 文件
terminal_colors.rs 3.53KiB  2025-04-01 09:21:19 文件
shell.rs  5.46KiB  2025-04-01 11:37:47 文件
unique_id.rs 1.30KiB  2025-03-21 02:30:42 文件
main.rs   4.15KiB  2025-04-01 11:42:05 文件

/home/camellia/os_lab0/src> cat shell.rs
[INFO] === 文件内容: shell.rs ===
use std::env;
use std::fs::{self, DirEntry, Metadata};
use std::io::{self, Read, Write};
use std::path::{Path, PathBuf};
use std::time::SystemTime, UNIX_EPOCH;

use crate::terminal_colors;
use crate::read_and_print;

```

可以看出，成功实现了简易 Shell 程序，支持 cd、ls、cat 等基本命令，可以方便地查看文件内容、切换目录等操作。