



中山大學  
SUN YAT-SEN UNIVERSITY

# YSOS Lab1 实验报告

## 实验准备与操作系统启动

姓 名： 牛渲溟  
学 号： 23336187  
教学班号： 吴岸聪老师班级  
专 业： 计算机科学与技术（人工智能方向）  
院 系： 实验报告学院

2024-2025 学年第二学期

# 一. 实验零：实验准备

## 1. 实验目的

1.1. Rust 学习和巩固，了解标准库提供的基本数据结构和功能。

1.2. QEMU 与 Rust 环境搭建，尝试使用 QEMU 启动 UEFI Shell。

1.3. 了解 x86 汇编、计算机的启动过程，UEFI 的启动过程，实现 UEFI 下的 Hello, world!。

## 2. 安装项目开发环境：

本文采用 Linux24.04 搭配 VSCode (Remote) + Python / make + GDB 结合 gef 进行开发、调试

## 3. 尝试使用 **Rust** 进行编程

### 3.1. 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!

```
1 use std::thread;
2 use std::time::Duration;
3
4 /// 倒计时函数，每秒输出剩余秒数
5 pub fn count_down(seconds: u64) {
6     for i in (1..=seconds).rev() {
7         println!("{}", i);
8         thread::sleep(Duration::from_secs(1));
9     }
10    println!("Countdown finished!");
11 }
12
13 pub fn main() {
14     // 调用倒计时函数，以5秒为例
15     count_down(5);
16 }
```

### 3.2. 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!。

```
1 use std::fs;
2
3 /// 读取并打印文件内容的函数
4 pub fn read_and_print(file_path: &str) {
5     // 尝试读取文件内容，如果失败则 panic
6     let contents = fs::read_to_string(file_path)
7         .expect("File not found!");
8
9     // 打印文件内容
10    println!("{}", contents);
11 }
```

 Rust

### 3.3. 创建一个函数 file\_size(file\_path: &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。

```
1 use std::fs;
2
3 /// 获取文件大小的函数，如果文件不存在返回错误
4 pub fn file_size(file_path: &str) -> Result<u64, String> {
5     match fs::metadata(file_path) {
6         Ok(metadata) => Ok(metadata.len()),
7         Err(_) => Err("File not found!".to_string())
8     }
9 }
```

 Rust

### 3.4. 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。

```
1 fn main() {
2     // 1. 首先调用 count_down(5) 函数进行倒计时
3     println!("开始倒计时...");
4     count_down(5);
5     // 2. 然后调用 read_and_print("/etc/hosts") 函数尝试读取并输出文件内容
```

 Rust

```

6     println!("\n读取 /etc/hosts 文件内容:");
7     read_and_print("/etc/hosts");
8
9     // 3. 最后使用 std::io 获取用户输入的路径, 并调用 file_size 函数
10    println!("\n请输入文件路径来检查文件大小 (输入 'q' 退出):");
11
12    loop {
13        print!("> ");
14        io::stdout().flush().unwrap(); // 确保提示符立即显示
15
16        let mut input = String::new();
17        io::stdin().read_line(&mut input).expect("读取输入失败");
18
19        let input = input.trim();
20
21        if input == "q" {
22            println!("程序结束");
23            break;
24        }
25
26        match file_size(input) {
27            Ok(size) => println!("文件 '{}' 大小: {} 字节", input, size),
28            Err(e) => println!("错误: {}", e)
29        }
30    }
31 }

```

### 3.5. 实现一个进行字节数转换的函数, 并格式化输出

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

```

1  /// 将字节数转换为人类可读的格式, 返回值和单位 Rust
2  /// 使用1024进制和二进制前缀(B, KiB, MiB, GiB等)
3  pub fn humanized_size(size: u64) -> (f64, &'static str) {
4      const UNITS: [&str; 6] = ["B", "KiB", "MiB", "GiB", "TiB", "PiB"];
5
6      if size == 0 {
7          return (0.0, UNITS[0]);
8      }
9
10     // 计算适合的单位
11     let index = (size as f64).log(1024.0).floor() as usize;
12     let index = index.min(UNITS.len() - 1); // 防止溢出
13

```

```
14 // 计算转换后的值
15 let value = size as f64 / (1024.0_f64.powi(index as i32));
16
17 (value, UNITS[index])
18 }
```

## 4. 运行 UEFI Shell

### 4.1. 初始化仓库

#### 4.1.1. 克隆到本地

```
1 (base) niuxh@niuxh-virtual-machine:~$ git clone http://github.com/
YatSen0S/YatSen0S-Tutorial-Volume-2.git
2 Cloning into 'YatSen0S-Tutorial-Volume-2'...
3 warning: redirecting to https://github.com/YatSen0S/YatSen0S-Tutorial-Volume-2.
git/
4 remote: Enumerating objects: 1949, done.
5 remote: Counting objects: 100% (840/840), done.
6 remote: Compressing objects: 100% (455/455), done.
7 remote: Total 1949 (delta 570), reused 443 (delta 370), pack-reused 1109 (from
1)
8 Receiving objects: 100% (1949/1949), 3.97 MiB | 1.08 MiB/s, done.
9 Resolving deltas: 100% (872/872), done.
10 (base) niuxh@niuxh-virtual-machine:~$ ls
11 YatSen0S-Tutorial-Volume-2
```

#### 4.1.2. 初始化你的仓库

```
1 (base) niuxh@niuxh-virtual-machine:~/YatSen0S-Tutorial-Volume-2$ git
init
2 Reinitialized existing Git repository in /home/niuxh/YatSen0S-Tutorial-
Volume-2/.git/
3 (base) niuxh@niuxh-virtual-machine:~/YatSen0S-Tutorial-Volume-2$ git add .
4 (base) niuxh@niuxh-virtual-machine:~/YatSen0S-Tutorial-Volume-2$ git commit -m
5
6 (base) niuxh@niuxh-virtual-machine:~/YatSen0S-Tutorial-Volume-2$ git commit -m
"init"
7 On branch main
8 Your branch is up to date with 'origin/main'.
9
10 nothing to commit, working tree clean
```

#### 4.1.3. 校验文件完整性(展示部分)

```

1 (base) niuxh@niuxh-virtual-machine:~/YatSenOS-Tutorial-Volume-2$ git ls-
tree --full-tree -r --name-only HEAD
2 .github/workflows/deploy.yml
3 .gitignore
4 .prettierrc.json
5 LICENSE
6 README.md

```

Shell

## 4.2. 使用 QEMU 启动 UEFI Shell

```

1 (base) niuxh@niuxh-virtual-machine:~/YatSenOS-Tutorial-Volume-2/
src/0x00$ qemu-system-x86_64 -bios ./assets/OVMF.fd -net none -
nographic
2 BdsDxe: failed to load Boot0001 "UEFI QEMU DVD-ROM QM00003 " from PciRoot(0x0)/
Pci(0x1,0x1)/Ata(Secondary,Master,0x0): Not Found
3 BdsDxe: loading Boot0002 "EFI Internal Shell" from Fv(7CB8BDC9-F8EB-4F34-
AAEA-3EE4AF6516A1)/FvFile(7C04A583-9E3E-4F1C-AD65-E05268D0B4D1)
4 BdsDxe: starting Boot0002 "EFI Internal Shell" from Fv(7CB8BDC9-F8EB-4F34-
AAEA-3EE4AF6516A1)/FvFile(7C04A583-9E3E-4F1C-AD65-E05268D0B4D1)
5 UEFI Interactive Shell v2.2
6 EDK II
7 UEFI v2.70 (EDK II, 0x00010000)
8 Mapping table
9 BLK0: Alias(s):
10 PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
11 Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
12 Shell>

```

Shell

## 5. 启动 YSOS

### 5.1. 配置 Rust Toolchain

```

1 {
2     "rust-analyzer.cargo.target": "x86_64-unknown-uefi"
3 }

```

JSON

### 5.2. 运行第一个 UEFI 程序

```

1 #![no_std]
2 #![no_main]
3
4 #[macro_use]
5 extern crate log;

```

Rust

```
6  extern crate alloc;
7
8  use core::arch::asm;
9  use uefi::{Status, entry};
10
11 #[entry]
12 fn efi_main() -> Status {
13     uefi::helpers::init().expect("Failed to initialize utilities");
14     log::set_max_level(log::LevelFilter::Info);
15
16     let std_num = "23336187";
17
18     loop {
19         info!("Hello World from UEFI bootloader! @ {}", std_num);
20
21         for _ in 0..0x10000000 {
22             unsafe {
23                 asm!("nop");
24             }
25         }
26     }
27 }
```

```
1 [ INFO]: pkg/boot/src/main.rs@019: Hello World from UEFI bootloader! @
  23336187
```

Shell

## 6. 思考题

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

6.1.1. 现代操作系统（Windows）的启动过程

6.1.1.1. 固件初始化

进行硬件自检，初始化硬件，转交控制权给引导加载程序。

6.1.1.2. 启动加载

6.1.1.3. 引导加载

6.1.1.4. 内核初始化，系统服务启动

## 6.1.2. UEFI 和 Legacy (BIOS) 的区别

### 6.1.2.1. 架构设计

UEFI 提供更加现代化的设计以及更加丰富的扩展, 甚至图形化界面。BIOS 是传统的启动方式, 只支持文本界面以及有限的扩展。

### 6.1.3. 磁盘支持

UEFI 支持大磁盘多分区, 而 BIOS 只支持小磁盘少分区。

### 6.1.4. 启动速度

UEFI 启动速度更快, 支持并行加载。BIOS 启动速度较慢。

### 6.1.5. 安全性

UEFI 相对安全而 BIOS 没有基本的安全措施。

### 6.1.6. 总结

现代操作系统基本采用 UEFI 启动, BIOS 逐渐只用于兼容老旧设备

## 6.2. 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?

### 6.2.1. 源代码来源

源代码来源于网络上的仓库, 如 GitHub、GitLab 等, 也可以是本地的仓库。从网络下载下来之后储存在 `./cargo/registry/src`

### 6.2.2. 编译时机

第三方库在第一次被使用时会被编译, 编译后的结果会被缓存到 `./cargo/registry/cache` 中, 下次使用时会直接使用缓存。

### 6.2.3. 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?

`main` 函数在操作系统中可以作为程序入口, 但是在 UEFI 的 `no_std` 环境下, 不可以使用 `main` 进入程序, 所以需要使用 `entry` 宏来指定程序入口。


## 7. 加分题



## 7.1. 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）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）实现 `cat` 命令，输出某个文件的内容

```
1 use std::env;
2 use std::fs;
3 use std::io::{self, Write};
4 use std::path::Path;
5 use std::time::{UNIX_EPOCH, SystemTime};
6
7 /// 将字节数转换为人类可读的格式
8 fn humanized_size(size: u64) -> (f64, &'static str) {
9     const UNITS: [&str; 6] = ["B", "KiB", "MiB", "GiB", "TiB", "PiB"];
10
11     if size == 0 {
12         return (0.0, UNITS[0]);
13     }
14
15     let index = (size as f64).log(1024.0).floor() as usize;
16     let index = index.min(UNITS.len() - 1);
17
18     let value = size as f64 / (1024.0_f64.powi(index as i32));
19
20     (value, UNITS[index])
21 }
22
23 /// 实现 cd 命令：切换当前工作目录
24 fn cmd_cd(args: &[String]) -> io::Result<()> {
25     let target_dir = if args.is_empty() {
26         // 如果没有参数，切换到用户主目录
27         env::var("HOME").unwrap_or_else(|_| String::from("."))
28     } else {
29         args[0].clone()
30     };
31
32     env::set_current_dir(Path::new(&target_dir))?;
33     println!("当前目录: {}", env::current_dir()?.display());
34     Ok(())
35 }
36
37 /// 实现 ls 命令：列出当前目录内容
38 fn cmd_ls(args: &[String]) -> io::Result<()> {
39     let target_dir = if args.is_empty() {
40         // 如果没有参数，列出当前目录
```

 Rust

```

41     String::from(".")
42 } else {
43     args[0].clone()
44 };
45
46 let entries = fs::read_dir(target_dir)?;
47
48 println!("{:<30} {:<10} {:<20}", "文件名", "大小", "修改时间");
49 println!("{:-<70}", "");
50
51 for entry in entries {
52     let entry = entry?;
53     let metadata = entry.metadata()?;
54
55     // 获取文件大小
56     let size = metadata.len();
57     let (size_value, size_unit) = humanized_size(size);
58     let size_display = format!("{:.2} {}", size_value, size_unit);
59
60     // 获取修改时间
61     let modified = metadata
62         .modified()?
63         .duration_since(UNIX_EPOCH)
64         .unwrap_or_default()
65         .as_secs();
66
67     // 格式化时间 (简化处理)
68     let time_display = format!("{}", 秒, modified);
69
70     // 显示文件类型 (目录/文件)
71     let file_type = if metadata.is_dir() { "目录/" } else { "文件  " };
72
73     println!("{ }{:<30} {:<10} {}",
74         file_type,
75         entry.file_name().to_string_lossy(),
76         size_display,
77         time_display
78     );
79 }
80
81 Ok(())
82 }
83
84 /// 实现 cat 命令: 显示文件内容
85 fn cmd_cat(args: &[String]) -> io::Result<()> {

```

```

86     if args.is_empty() {
87         println!("错误: 缺少文件路径参数");
88         return Ok(());
89     }
90
91     let file_path = &args[0];
92     let contents = fs::read_to_string(file_path)?;
93     println!("{}", contents);
94
95     Ok(())
96 }

```

## 7.2. 你对 Rust 的 unsafe 有什么看法？

unsafe 关键字用于标记不安全的代码块，Rust 语言的设计目标之一是安全性，所以尽量避免使用 unsafe 关键字。unsafe 代码块可以绕过 Rust 的安全检查，可能导致内存泄漏、空指针、数据竞争等问题。unsafe 代码块应该尽量避免使用，只有在必要的情况下才使用，同时应该尽量减少 unsafe 代码块的范围，以减少潜在的安全问题。当用 unsafe 时，rust 几乎退化成了 C 语言。

# 二. 实验一：操作系统的启动

## 1. 编译内核 ELF

运行 cargo build --release 之后，在 esp/KERNEL.ELF 找到编译产物

### 1.1. 查看编译产物的架构相关信息，与配置文件中的描述是否一致

跟据配置文件描述，编译产物具有以下特性：

64 位指针宽度，Little-Endian 字节序；没有操作系统支持（裸机）；使用软浮点 ABI（x86-softfloat）；编译过程中使用 rust-ld 作为链接器，并附带特定的链接参数。

编译产物架构信息如下：

```

1  (base) niuxh@niuxh-virtual-machine:~/YatSenOS-Tutorial-Volume-2/
src/0x01/esp$ readelf KERNEL.ELF -h
2  ELF Header:
3  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
4  Class:                               ELF64
5  Data:                                   2's complement, little endian
6  Version:                               1 (current)
7  OS/ABI:                                UNIX - System V
8  ABI Version:                           0

```

9	Type:	EXEC (Executable file)
10	Machine:	Advanced Micro Devices X86-64
11	Version:	0x1
12	Entry point address:	0xffffffff0000004450
13	Start of program headers:	64 (bytes into file)
14	Start of section headers:	143656 (bytes into file)
15	Flags:	0x0
16	Size of this header:	64 (bytes)
17	Size of program headers:	56 (bytes)
18	Number of program headers:	7
19	Size of section headers:	64 (bytes)
20	Number of section headers:	12
21	Section header string table index:	10

编译产物的架构信息与配置文件描述一致。

## 1.2. 找出内核的入口点，它是被如何控制的？结合源码、链接、加载的过程，谈谈你的理解

跟据 pkg/kernel/config/boot.conf，内核入口点为 `_start`。

控制过程如下：

在 `kernel_main` 函数里调用 `boot::entry_point!(kernel_main)`，将 `kernel_main` 与 `_start` 关联起来跟据 `pkg/kernel/config/kernel.ld` 的 `KERNEL_BEGIN = 0xffffffff0000000000`，可以知道传递进入的参数为 `0xffffffff0000000000`，即内核的起始地址。

## 1.3. 请找出编译产物的 segments 的数量，并且用表格的形式说明每一个 segments 的权限、是否对齐等信息。

共有 7 个 segments，如下表所示：

Type	VirtAddr	PhysAddr	FileSiz	MemSiz	Flags	Align
LOAD	0xffffffff0000000000	0xffffffff0000000000	0x3284	0x3284	R	0x1000
LOAD	0xffffffff0000004000	0xffffffff0000004000	0x104a4	0x104a4	R E	0x1000
LOAD	0xffffffff0000015000	0xffffffff0000015000	0x2088	0x2088	RW	0x1000
LOAD	0xffffffff0000018000	0xffffffff0000018000	0x0	0x801310	RW	0x1000
GNU_RELRO	0xffffffff0000017000	0xffffffff0000017000	0x88	0x88	R	0x1
GNU_EH_FRAME	0xffffffff0000003238	0xffffffff0000003238	0x14	0x14	R	0x4
GNU_STACK	0x0	0x0	0x0	0x0	RW	0x0

## 2. 在 UEFI 中加载内核

代码将在附加文件中给出

## 2.1. gdb 调试

1	[ Legend: Code   Stack   Heap ]				Shell
2	Start	End	Offset	Perm Path	
3	0x0000000000000000	0x0000000004c00000	0x0000000004c00000	rw-	
4	0x0000000004c00000	0x0000000004e00000	0x000000000200000	r--	
5	0x0000000004e00000	0x0000000005c00000	0x000000000e00000	rw-	
6	0x0000000005c00000	0x0000000005e00000	0x000000000200000	r--	
7	0x0000000005e00000	0x0000001000000000	0x0000000ffa200000	rw-	
8	0xffff800000000000	0xffff800100200000	0x0000000100200000	rw-	
9	0xfffff00000000000	0xfffff00000015000	0x0000000000015000	r--	
10	0xfffff00000015000	0xfffff00000081a000	0x00000000000805000	rw-	
11	0xfffff01000000000	0xfffff01002000000	0x0000000000200000	rw-	

## 2.2. set\_entry 函数做了什么？为什么它是 unsafe 的？

set\_entry 函数将传入的地址设置为新的入口地址，这样在调用 bootloader 的时候，就会跳转到新的地址执行。

set\_entry 是 unsafe 的，因为它直接操作了内存，可能会导致内存泄漏、空指针、数据竞争等问题。

## 2.3. jump\_to\_entry 函数做了什么？要传递给内核的参数位于哪里？查询 call 指令的行为和 x86\_64 架构的调用约定，借助调试器进行说明。

### 2.3.1. jump\_to\_entry 函数作用

jump\_to\_entry 函数的作用是设置内核启动时的栈，并跳转到内核入口地址，将栈指针(rsp)设置为传入的 stacktop，确保内核使用预分配的栈

### 2.3.2. 传递给内核的参数位置

跟据 x86\_64 架构的调用约定，函数参数第一个为 rdi，所以传递给内核的第一个参数 bootinfo 位于 rdi 寄存器中，第二个参数 stacktop 位于 rsp

### 2.3.3. 寄存器行为

1	gef> x/4gx \$rdi				Shell
2	0x5f0dfc0:	0x00000000059ee018	0x0000000000000087		
3	0x5f0dfd0:	0x0000000000000003	0x0000000000000000		

可以看到，rdi 寄存器中存放了 bootinfo 的内容，内存映射切片的数据指针 (0x59ee018) 内存映射切片的 length (0x87) physical\_memory\_offset (这里显示 0x3) system\_table 指针 (0x0)

1	gef> x/4gx \$rsp				Shell
2	0xfffff01001fff40:	0xfffff0000004053	0x0000000000000000		
3	0xfffff01001fff50:	0x0000000000000000	0x0000000000000000		

可以看到，rsp 寄存器中存放了 stacktop 的内容，内核入口地址 (0xfffff0000004053)

### 2.3.4. call 指令行为

1	0xffffffff0000007e5e:	int3	
2	0xffffffff0000007e5f:	int3	
3	=> 0xffffffff0000007e60	<_ZN11sysos_kernel4init17h128b3dcb3d2f586dE+0>:	push rbx
4	0xffffffff0000007e61	<_ZN11sysos_kernel4init17h128b3dcb3d2f586dE+1>:	sub rsp,0x60
5	0xffffffff0000007e65	<_ZN11sysos_kernel4init17h128b3dcb3d2f586dE+5>:	mov rbx,rdi
6	0xffffffff0000007e68	<_ZN11sysos_kernel4init17h128b3dcb3d2f586dE+8>:	mov rdi,QWORD PTR [rdi]
7	0xffffffff0000007e6b	<_ZN11sysos_kernel4init17h128b3dcb3d2f586dE+11>:	call 0xffffffff0000009000 <_ZN4uefi5table16set_system_table17h3da559f5a2f670ffE>

jump\_to\_entry 函数通过修改 rsp 和利用 call 指令跳转到内核入口，从而把 bootinfo 参数（放在 rdi 中）以及新的栈环境传递给内核，将 stacktop 传递到 rsp。这正符合 x86\_64 调用约定中第一个参数通过寄存器传递的规则，同时 call 指令会自动把返回地址压栈，但由于内核入口函数永不返回，这个返回地址不会被使用。

## 2.4. entry\_point! 宏做了什么？内核为什么需要使用它声明自己的入口点？

### 2.4.1. 功能

创建标准入口点函数，并将 bootinfo 作为参数传递给内核入口函数

### 2.4.2. 为什么需要使用

链接器兼容性：链接器需要一个名为 \_start 的符号作为程序入口点，这个宏确保生成了这个标准命名的入口

## 2.5. 如何为内核提供直接访问物理内存的能力？你知道几种方式？代码中所采用的是哪一种？

### 2.5.1. 方式

恒等映射 偏移映射 按需映射 临时映射

### 2.5.2. 代码中采用的方式

偏移映射，通过设置 physical\_memory\_offset，将物理地址映射到虚拟地址

## 2.6. 为什么 ELF 文件中不描述栈的相关内容？栈是如何被初始化的？它可以被任意放置吗？

### 2.6.1. 原因

ELF 文件中不描述栈的相关内容, 是因为栈是在运行时动态分配的, 不需要在编译时确定栈的大小和位置。而 ELF 在编译时需要确定。

### 2.6.2. 初始化

在配置文件 `booy.conf` 中确定栈大小与位置 `kernel_stack_address=0xFFFFFFFF0100000000` `kernel_stack_size=512`, 然后在 `main.rs` 里通过 `map_range` 函数分配并映射栈空间 在跳转到内核入口点前, 计算栈顶地址并通过 `jump_to_entry` 设置 RSP

### 2.6.3. 放置

不可以随意放置, 要满足条件, 避免冲突, 考虑生长方向, 安全边界

## 2.7. 请解释指令 `layout asm` 的功能。倘若想找到当前运行内核所对应的 Rust 源码, 应该使用什么 GDB 指令?

### 2.7.1. `layout asm`

`layout asm` 命令在 GDB 中启用一个汇编指令显示窗口, 显示当前执行位置附近的汇编代码, 高亮显示当前指令

### 2.7.2. 找到当前运行内核所对应的 Rust 源码

调用 `gdb` 指令的 `backtrace`, 可以找到当前运行内核所对应的 Rust 源码



```
1 gef> backtrace
2 #0  0xffffffff0000007e60 in ysos_kernel::init ()
3 #1  0xffffffff0000004053 in ysos_kernel::kernel_main ()
4 #2  0x0000000000000000 in ?? ()
```

### 2.7.3. 假如在编译时没有启用 `DBG_INFO=true`, 调试过程会有什么不同?

缺少调试符号信息

源代码位置信息将不可用或极度有限 无法将汇编指令映射回 Rust 源代码行 `.gdbinit` 中的 `break ysos_kernel::init` 断点设置失败, 因为符号信息缺失

## 2.8. 你如何选择了你的调试环境? 截图说明你在调试界面 (TUI 或 GUI) 上可以获取到哪些信息?

选择了 `gdb` 调试环境, 截图如下:

```

gef> gef-remote localhost 1234
0x000000000000ffff in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

registers
$rax : 0xffffffff000004450 → <_start+0000> push rax
$rbx : 0x0000000100000000 → 0x0000000100000000
$rcx : 0x0000000005f0dfc0 → 0x00000000059ee018 → "IBI SYSTF"
$rdx : 0xffffffff01001ffff8 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$rsp : 0xffffffff01001fff40 → 0xffffffff000004053 → <_sysos_kernel::kernel_main+0013> call 0xffffffff000007b30 <_ZN11sysos_kernel6memory9allocat
r4init17h310408bb809de12eE>
$rbp : 0xffffffff1ffaffc0
$rsi : 0x0000000005f0de88 → 0x00000000043a1018 → 0x0000000000000003 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$rdi : 0x0000000005f0dfc0 → 0x00000000059ee018 → "IBI SYSTF"
$rip : 0xffffffff000008000 → <_sysos_kernel::init+0000> push rbx
$r8 : 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r9 : 0x0000000000000000 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r10 : 0x0000000005ae8860 → 0x7707309600000000
$r11 : 0x0000000000000000 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r12 : 0x00000000043b8048 → 0x0000000000000003 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r13 : 0x00000000043b3178 → 0x00000000043a6760 → 0xc0000000029abe9
$r14 : 0xffffffff0000000000 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r15 : 0x0000000005f0df40 → 0x0000000005c01000 → 0x0000000005c02023 → 0xc080030000000005
$eflags: [zero carry PARITY ADJUST SIGN trap interrupt direction overflow resume virtualx86 identification]
$cs: 0x38 $ss: 0x30 $ds: 0x30 $es: 0x30 $fs: 0x30 $gs: 0x30

stack
0xffffffff01001fff40|+0x0000: 0xffffffff000004053 → <_sysos_kernel::kernel_main+0013> call 0xffffffff000007b30 <_ZN11sysos_kernel6memory9allocat
r4init17h310408bb809de12eE> ← $rsp
0xffffffff01001fff48|+0x0008: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff50|+0x0010: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff58|+0x0018: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff60|+0x0020: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff68|+0x0028: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff70|+0x0030: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff78|+0x0038: 0x0000000000000000 → 0x0000000000000000 → [loop detected]

code:x86:64
0xffffffff000007ff9 <_sysos_kernel::memory::allocator::init+04c9> call 0xffffffff00000114a0 <_ZN4core9panicking5panic17h6eald095a27a335bE>
0xffffffff000007ffe int3
0xffffffff000007fff int3
→ 0xffffffff000008000 <_sysos_kernel::init+0000> push rbx
0xffffffff000008001 <_sysos_kernel::init+0001> sub rsp, 0x60
0xffffffff000008005 <_sysos_kernel::init+0005> mov rbx, rdi
0xffffffff000008008 <_sysos_kernel::init+0008> mov rdi, QWORD PTR [rdi]
0xffffffff00000800b <_sysos_kernel::init+000b> call 0xffffffff000008e90 <_ZN4uefi5table16set_system_table17h3da559f5a2f670ffe>
0xffffffff000008010 <_sysos_kernel::init+0010> call 0xffffffff000007740 <_ZN11sysos_kernel7drivers6serial4init17hdd9f189d3ff14497E>

threads
[#0] Id 1, stopped 0xffffffff000008000 in _sysos_kernel::init (), reason: BREAKPOINT

trace
[#0] 0xffffffff000008000 → _sysos_kernel::init()
[#1] 0xffffffff000004053 → _sysos_kernel::kernel_main()
[#2] 0x0 → add BYTE PTR [rax], al

```

## 3. UART 与日志输出

### 3.1. 在 pkg/boot/lib.rs 中的 ENTRY 是如何被处理的？

所有对 ENTRY 的操作都被标记为 unsafe，表明这些操作可能导致未定义行为或者线程不安全 jump\_to\_entry 函数中的 assert!(ENTRY != 0) 确保不会跳转到无效地址 这些函数只在 bootloader 自身使用，不会被内核直接访问

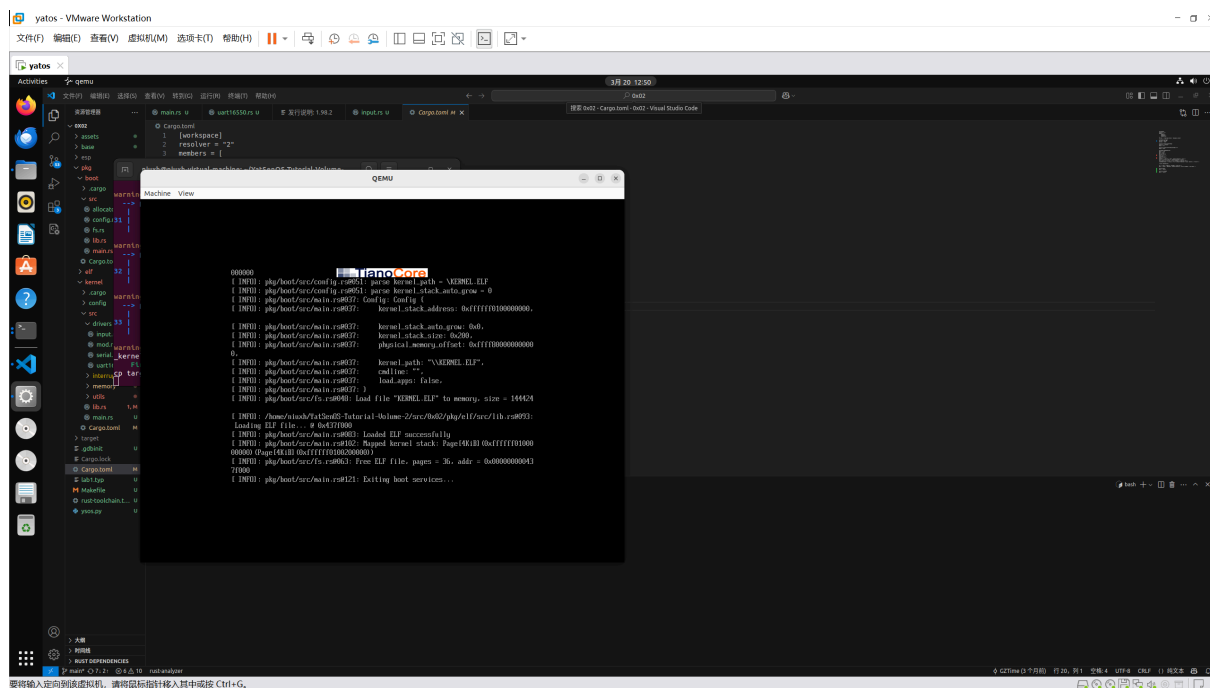
### 3.2. 日志输出





4.4. 在 QEMU 中，我们通过指定 `-nographic` 参数来禁用图形界面，这样 QEMU 会默认将串口输出重定向到主机的标准输出。

4.4.1. 假如我们将 Makefile 中取消该选项，QEMU 的输出窗口会发生什么变化？请观察指令 `make run QEMU_OUTPUT=` 的输出，结合截图分析对应现象



会产生图形化界面，QEMU 的输出窗口会显示图形化界面

4.4.2. 在移除 `-nographic` 的情况下，如何依然将串口重定向到主机的标准输入输出？请尝试自行构造命令行参数，并查阅 QEMU 的文档，进行实验。

使用 `qemu-system-x86_64 -bios assets/OVMF.fd -net none -m 96M -serial mon:stdio -drive format=raw,file=fat:esp -snapshot`

## 5. 加分题

5.1. 线控寄存器的每一比特都有特定的含义，尝试使用 `bitflags` 宏来定义这些标志位，并在 `uart16550` 驱动中使用它们：已实现

5.2. 尝试在进入内核并初始化串口驱动后，使用 `escape sequence` 来清屏，并编辑 `get_ascii_header()` 中的字符串常量，输出你的学号信息：已实现

5.3. 尝试添加字符串型启动配置变量 `log_level`，并修改 `logger` 的初始化函数，使得内核能够根据启动参数进行日志输出：已实现

5.4. 尝试使用调试器，在内核初始化之后（`ysos::init` 调用结束后）下断点，查看、记录并解释如下的信息：

```
1  gef> info registers Shell
2  rax                0xffffffff0000004450  0xffffffff0000004450
3  rbx                0x100000000          0x100000000
4  rcx                0x5f0dfc0          0x5f0dfc0
5  rdx                0xffffffff01001ffff8  0xffffffff01001ffff8
6  rsi                0x5f0de88          0x5f0de88
7  rdi                0x5f0dfc0          0x5f0dfc0
8  rbp                0xffffffff1ffaaffc0  0xffffffff1ffaaffc0
9  rsp                0xffffffff01001fff40  0xffffffff01001fff40
10 rip                0xffffffff0000008000  0xffffffff0000008000 <ysos_kernel::init>

1  gef> info files Shell
2  Symbols from "/home/niuxh/YatSenOS-Tutorial-Volume-2/src/0x02/esp/KERNEL.ELF".
3  Remote target using gdb-specific protocol:
4      `/home/niuxh/YatSenOS-Tutorial-Volume-2/src/0x02/esp/KERNEL.ELF', file
   type elf64-x86-64.
5  Entry point: 0xffffffff0000004450
6  0xffffffff0000000000 - 0xffffffff00000031ef is .rodata
7  0xffffffff00000031f0 - 0xffffffff0000003204 is .eh_frame_hdr
8  0xffffffff0000003208 - 0xffffffff000000323c is .eh_frame
9  0xffffffff0000004000 - 0xffffffff0000014334 is .text
10 0xffffffff0000015000 - 0xffffffff0000016958 is .data
11 0xffffffff0000017000 - 0xffffffff0000017088 is .got
12 0xffffffff0000018000 - 0xffffffff00000819310 is .bss
```

5.4.1. 内核的栈指针、栈帧指针、指令指针等寄存器的值。

指令指针（RIP）`0xffffffff0000008000`  
栈指针（RSP）`= 0xffffffff01001fff40`  
栈顶指针（RBP）`= 0xffffffff1ffaaffc0`

5.4.2. 内核的代码段、数据段、BSS 段等在内存中的位置。

.rodata 段：从地址 `0xffffffff0000000000` 到 `0xffffffff00000031ef`  
.text 段：从 `0xffffffff0000004000` 到 `0xffffffff0000014334`  
.data 段：从 `0xffffffff0000015000` 到 `0xffffffff0000016958`  
.bss 段：从 `0xffffffff0000018000` 到 `0xffffffff00000819310`