



# 课程实验报告

## 0x01 操作系统的启动

课程名称	操作系统原理实验
专业名称	计算机科学与技术
学生姓名	陈政宇
学生学号	23336003
实验地点	东校园-实验中心大楼 B201
实验成绩	
实验日期	2025 年 3 月 6 日

# 目录

<b>1 编译内核 ELF .....</b>	<b>3</b>
1.1 编译产物信息 .....	3
1.2 内核入口 .....	4
1.3 Segments 信息 .....	5
<b>2 在 UEFI 中加载内核 .....</b>	<b>7</b>
2.1 代码解释 .....	7
2.1.1 代码补全 .....	7
2.1.2 set_entry 函数 .....	9
2.1.3 jump_to_entry 函数 .....	9
2.1.4 entry_point! 宏 .....	10
2.1.5 物理内存访问方式 .....	10
2.1.6 栈 .....	11
2.2 调试过程 .....	11
2.2.1 layout asm 指令 .....	11
2.2.2 DBG_INFO 编译选项 .....	12
2.2.3 调试环境 .....	12
<b>3 UART 与日志输出 .....</b>	<b>13</b>
3.1 串口驱动的实现 .....	13
3.1.1 初始化 .....	13
3.1.2 发送数据 .....	14
3.1.3 接收数据 .....	14
3.2 日志输出 .....	14
3.2.1 Initialize .....	15
3.2.2 Log .....	15
<b>4 思考题 .....</b>	<b>17</b>
4.1 Cargo.toml .....	17
4.2 max_phys_addr .....	17
4.3 内容输出 .....	18
4.4 移除 -nographic 选项 .....	19
4.5 如何将串口重定向到宿主机的标准 I/O .....	20
<b>5 附加题 .....</b>	<b>21</b>
5.1 bitflags .....	21
5.2 清屏并输出学号 .....	21
5.3 添加 log_level 并根据启动参数进行日志输出 .....	22
5.4 调试器观测 .....	25

# 1 编译内核 ELF

## ? Question

在 `pkg/kernel` 目录下运行 `cargo build --release`，之后找到编译产物，并使用 `readelf` 命令查看其基本信息，回答以下问题：

- 请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？
- 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。
- 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。

## 1.1 编译产物信息

使用 `readelf -h` 命令查看编译产物的基本信息：

1	ELF Header:	elf
2	Magic:	7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3	Class:	ELF64
4	Data:	2's complement, little endian
5	Version:	1 (current)
6	OS/ABI:	UNIX - System V
7	ABI Version:	0
8	Type:	EXEC (Executable file)
9	Machine:	Advanced Micro Devices X86-64
10	Version:	0x1
11	Entry point address:	0xffffffff0000002360
12	Start of program headers:	64 (bytes into file)
13	Start of section headers:	56064 (bytes into file)
14	Flags:	0x0
15	Size of this header:	64 (bytes)
16	Size of program headers:	56 (bytes)
17	Number of program headers:	7
18	Size of section headers:	64 (bytes)
19	Number of section headers:	12
20	Section header string table index:	10

从配置文件中可以观察到：

- "arch" : "x86\_64" 表示目标平台是 64 位 x86 架构。
- "target-pointer-width" : 64 也说明这是一个 64 位目标。
- 其他设置如 "os" : "none", "disable-redzone": true 则表明这是一个无操作系统（裸机）环境，并非运行在具体的操作系统上。

从 ELF 头部信息可以看到：

- Class: ELF64 表示这是一个 64 位的 ELF 文件。
- Machine: Advanced Micro Devices X86-64 表示目标平台是 64 位 x86 架构。
- Entry point address: 0xffffffff0000002360 这个地址也是链接脚本中预期的高端地址（通常内核映像会映射到高端地址，如 0xffffffff0000000000 开始）。

综上，从 ELF 文件头信息来看，其架构（64 位、x86\_64）以及入口地址布局均与配置文件中描述的目标（x86\_64-unknown-none）相符，说明编译产物和配置文件描述是一致的。

## 1.2 内核入口

内核的入口点不是由单一部分“控制”的，而是经过源码、链接脚本和加载过程多层协同确定。下面结合源码、链接脚本和加载步骤进行解析：

### 1. 源码层面

在内核源码（例如 pkg/kernel/src/main.rs）中，我们使用宏声明了入口函数。例如：

```
1 boot::entry_point!(kernel_main);  
2  
3 pub fn kernel_main(boot_info: &'static boot::BootInfo) -> ! {  
4     ysos::init(boot_info);  
5     // 内核主循环和其它初始化...  
6     loop { /* ... */ }  
7 }
```

这里的 `boot::entry_point!` 宏会生成一个包装函数并将其标记为程序的起始点。在编译时，它会让链接器认识到入口符号（通常最终转换为 `_start`）应指向我们在源码中指定的 `kernel_main`。

### 2. 链接脚本层面

在内核的链接脚本（如 pkg/kernel/config/kernel.ld）中，有类似下面的语句：

```
1 ENTRY(_start)
```

```
2  KERNEL_BEGIN = 0xffffffff0000000000;  
3  SECTIONS {  
4      . = KERNEL_BEGIN;  
5      .text ALIGN(4K): { *(.text .text.*) }  
6      ...  
7  }
```

这里的 ENTRY(\_start) 告诉链接器：最终的 ELF 文件入口点为符号 \_start。通常在编译过程中，由宏生成的包装代码会在某处定义 \_start，它负责初始化 CPU 状态、堆栈和其它关键资源，然后跳转调用 kernel\_main。同时，链接脚本将内核各个段按照预设的高端虚拟地址（比如 KERNEL\_BEGIN 指定的地址）加载到内存中。

3. 加载过程

在 bootloader 阶段（在 pkg/boot 中构建），bootloader 会读取内核的 ELF 文件（比如 KERNEL.ELF），解析 ELF 头部，从中获得入口点地址（例如从 ELF Header 的 “Entry point address: 0xffffffff0000002360” 可以看出）。Bootloader 将内核映像加载到内存中，并构造 BootInfo（包括内存布局、系统表、启动参数等），最终调用类似 jump\_to\_entry(&bootinfo, stacktop) 的函数。该函数取得内核 ELF 文件的入口点地址，并跳转到该地址，使得 CPU 开始执行内核代码。

总结来说：

- 源码定义：在源码中通过宏（如 boot::entry\_point!）明确指定了内核入口函数（如 kernel\_main）。
- 链接控制：通过链接脚本指明入口符号 ENTRY(\_start)，并将所有段映射到预定的虚拟地址范围中。
- 加载执行：Bootloader 解析 ELF 文件并获知入口地址，加载内核映像后跳转执行，从而最终调用源码中定义的 kernel\_main。

1.3 Segments 信息

Seg	Type	Virt Addr	Size (F/M)	Flags	Align	Remarks
1	LOAD	0xffff...0000	0x1374/0x1374	R	0x1000	ELF hdr/rodata
2	LOAD	0xffff...2000	0x41bf/0x41bf	R, E	0x1000	.text
3	LOAD	0xffff...7000	0x1018/0x1018	R, W	0x1000	.data
4	LOAD	0xffff...9000	0x0/0x20	R, W	0x1000	.bss
5	GNU_RELRO	0xffff...8000	0x18/0x18	R	0x1	RELRO
6	GNU_EH_FRAME	0xffff...1348	0x0c/0x0c	R	0x4	EH frame
7	GNU_STACK	0x0	0x0/0x0	R, W	0x0	堆栈属性标记

**【说明】****• Segment 1 – LOAD (ELF header/rodata)**

此段存储了 ELF 文件头和程序头信息，同时还包含部分只读数据，如常量和调试信息。它以只读权限加载，对齐要求为 0x1000，文件大小与内存大小一致，从而保证启动时能够正确读取头部信息。

**• Segment 2 – LOAD (.text)**

该段承载了内核的可执行代码，即编译后的 .text 段。它具有只读和可执行的权限，禁止写操作以确保代码安全，并按照 0x1000 的对齐要求加载，其大小取决于编译生成的代码量。

**• Segment 3 – LOAD (.data)** v 本段用于存放已初始化的全局和静态变量，构成了 .data 段。数据在运行时允许读写，且按照 0x1000 的对齐要求加载，其大小准确反映了初始化数据的实际尺寸，保证程序启动时能够获得正确的初始数据状态。**• Segment 4 – LOAD (.bss)**

此段存储了未初始化的全局和静态变量，也就是 .bss 段。虽然文件中并不包含实际数据，但加载时操作系统会为其分配内存并将其清零，该段同样采用 0x1000 的对齐要求，以保证足够空间和页对齐。

**• Segment 5 – GNU\_RELRO**

该段代表 RELRO 区域，在重定位完成后会将部分数据区域设置为只读，从而提高安全性。这部分区域通常包含全局偏移表的一部分，具有只读属性，并且对齐要求较低，仅为 0x1，用于防止运行时意外修改。

**• Segment 6 – GNU\_EH\_FRAME**

此段存放异常处理帧数据，用于支持堆栈展开以及相关的异常处理机制（例如 C++ 异常和调试信息），数据非常小，权限为只读，对齐要求为 0x4，确保异常信息在内存中以正确格式排列。

**• Segment 7 – GNU\_STACK**

该段不实际分配内存，仅作为标记，指示加载器如何设置程序堆栈的属性。虽然显示为可读写，但它主要用来设定堆栈的保护属性，并没有实际的文件或内存大小，对齐要求为 0x0。

## 2 在 UEFI 中加载内核

### ? Question

完成代码任务，回答下列问题：

- `set_entry` 函数做了什么？为什么它是 `unsafe` 的？
- `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。
- `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？
- 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？可以参考这篇文章进行学习。
- 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？

根据上述调试过程，回答以下问题，并给出你的回答与必要的截图：

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

具体代码任务实现见附件。

### 2.1 代码解释

#### 2.1.1 代码补全

##### 1. 加载 ELF 文件

```
1 // 2. Load ELF files
2 let mut kernel_file = open_file(KERNEL_PATH);
3 let kernel_data = load_file(&mut kernel_file);
4 let elf_file = ElfFile::new(kernel_data)
5     .expect("Failed to parse ELF file");
6 unsafe {
7     set_entry(elf_file.header.pt2.entry_point() as usize);
```

```
8 }
```

## 2. 修改控制寄存器

```
1 // FIXME: root page table is readonly, disable write protect (Cr0) Rust
2 unsafe {
3     Cr0::update(|flags| {
4         flags.remove(Cr0Flags::WRITE_PROTECT);
5     });
6 }
```

## 3. 映射内核文件

```
1 // FIXME: map physical memory to specific virtual address offset Rust
2 let mut frame_allocator = UEFIFrameAllocator;
3 map_physical_memory(
4     config.physical_memory_offset,
5     max_phys_addr,
6     &mut page_table,
7     &mut frame_allocator,
8 );
9 // FIXME: load and map the kernel elf file
10 let _ = load_elf(&elf_file, config.physical_memory_offset, &mut page_table, &mut frame_allocator)
11     .expect("Failed to load ELF file");
12 info!("Kernel ELF loaded");
13 // FIXME: map kernel stack
14 map_range(
15     config.kernel_stack_address,
16     config.kernel_stack_size,
17     &mut page_table,
18     &mut frame_allocator,
19 )
20 .expect("Failed to map kernel stack");
21 // FIXME: recover write protect (Cr0)
22 unsafe {
23     Cr0::update(|flags| {
24         flags.insert(Cr0Flags::WRITE_PROTECT);
25     });
26 }
27 free_elf(elf_file);
```



### 2.1.2 set\_entry 函数

set\_entry 函数的主要作用是将传入的内核入口地址写入到全局静态变量 ENTRY 中。这个变量稍后会被用来跳转到内核的实际入口点，从而开始内核的执行流程。之所以将 set\_entry 标记为 unsafe，有以下原因：

- 在 Rust 中，修改全局可变静态变量（static mut）本身就是不安全操作，因为可能存在数据竞争或其他未定义行为，因此必须在 unsafe 块中进行操作。
- 调用者必须确保传入的 entry 地址是一个有效且正确的内核入口点。如果传入了错误或无效的地址，后续跳转 (jump\_to\_entry) 时会引发未定义行为甚至系统崩溃。

因此，set\_entry 本身虽然仅仅做了一次赋值，但其 unsafe 限定符提醒程序员需要格外小心，确保提供一个正确的入口地址，维护系统安全。

### 2.1.3 jump\_to\_entry 函数

jump\_to\_entry 函数的作用是切换到内核入口点，也就是跳转到全局变量 ENTRY 所保存的地址处开始执行内核代码。具体过程如下：

1. 设置栈指针 内联汇编中的 `mov rsp, {stacktop}` 将当前 CPU 的栈指针 `rsp` 设置为传入的 `stacktop`，为内核提供全新的栈空间。
2. 传递参数 通过 `in("rdi") bootinfo`，将传递给内核的参数（即 BootInfo 结构体指针）放入 `rdi` 寄存器。根据 x86\_64 的 System V 调用约定，第一个函数参数必须放入 `rdi` 中。
3. 调用内核入口 接下来的 `call {ENTRY}` 指令会将 `ENTRY` 寄存器中的地址作为目标，调用该地址指向的函数。`call` 指令会先将 return address 压入新设定的栈中，然后跳转。对于内核入口来说，该函数并不返回，因此 return address 实际上不会被使用。

利用调试器的说明：

- 观察栈指针 `rsp`：当执行 `jump_to_entry` 后，在调试器中（LLDB）设置断点于内核入口处，可以看到 `rsp` 的值已经更新为传递的 `stacktop`。
- 检查 `rdi`：根据调用约定，内核入口函数的第一个参数应位于 `rdi` 在调试器中打印 `rdi`，确认它确实持有传入的 `bootinfo` 地址。
- 查看 `call` 指令行为：在 x86\_64 架构中，`call` 指令会将当前指向调用指令之后的地址压栈，然后跳转到目标地址。结合调试器反汇编输出，可以看到跳转前后的流程，确认内核入口函数的地址与 `ENTRY` 一致。

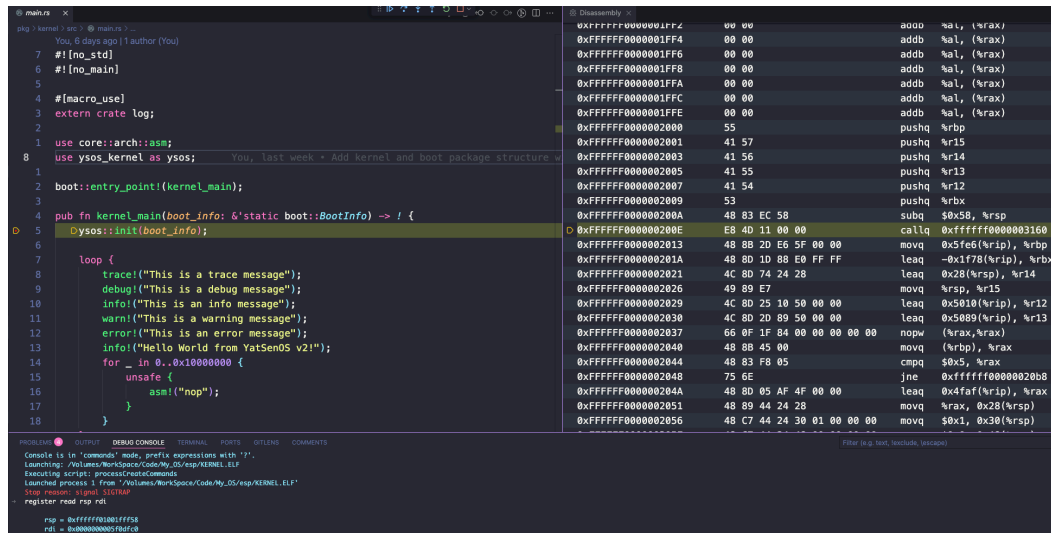


图 2.1.1: 调试器中内核入口点的栈和寄存器状态

### 2.1.4 entry\_point! 宏

entry\_point! 宏的作用是生成一个名为 `_start` 的外部函数，它符合内核启动所要求的入口点形式。这个宏确保传入的函数路径（也就是内核真实的入口函数）具有正确的签名（即 `fn(&'static BootInfo) -> !`），然后在生成的 `_start` 函数中调用该入口函数，把 bootloader 传递给内核的 `BootInfo` 参数传递过去，并从而开始执行内核代码。

内核需要使用这个宏来声明自己的入口点，原因在于：

- 确保入口签名正确：使用宏可以在编译期验证传入的函数签名，避免因签名错误导致启动失败。
- 统一导出符号：通过 `#[unsafe(export_name = "_start")]`，宏生成的函数会被导出为 `_start` 符号，这正是链接器寻找的默认入口点。
- 简化流程：宏隐藏了入口函数包装的细节，开发者只需提供实际的内核入口函数，宏负责生成符合要求的包装代码，从而确保内核启动时能正确接收 bootloader 提供的信息并开始执行。

entry\_point! 宏为内核入口点提供了一层安全验证和抽象，使得内核入口声明既符合编译器和链接器的要求，又降低了出错的可能。

### 2.1.5 物理内存访问方式

内核要能够直接访问物理内存主要是通过通过在页表中建立合适的映射实现的。常见的方法有：

1. 身份映射 (Identity Mapping)：将物理地址直接映射到同样的虚拟地址中，但这种方法受限于虚拟地址空间大小。

2. 固定偏移映射 (**Fixed Offset Mapping**): 为整个物理内存预留一块连续的虚拟地址区域, 虚拟地址 = 物理地址 + 固定偏移值。这样可以通过已知偏移直接计算物理地址对应的虚拟地址。
3. 递归页表映射 (**Recursive Paging**): 利用页表自身的映射关系来访问任意页表项, 从而间接访问物理内存。

在本实验中, 采用的是固定偏移映射的方法: 函数 `map_physical_memory` 遍历了物理内存帧, 并把每一帧映射到了虚拟地址空间中, 映射后的虚拟地址等于物理帧地址加上一个传入的 `offset`。同时, `BootInfo` 中配置的 `physical_memory_offset` 就是这个固定偏移值, 这样内核就可以基于虚拟地址直接访问物理内存。

### 2.1.6 栈

ELF 文件仅描述运行时需要加载到内存中的代码和数据段, 而栈并不是二进制中固定的一部分, 它在程序加载后由操作系统 (或在内核情形下, 由 Bootloader 或内核初始化代码) 动态分配和初始化。

当内核启动时, 启动代码会为内核提供预先分配好的内核栈, 这个地址通常在链接脚本和配置文件 (如 `boot.conf` 中的 `kernel_stack_address` 和 `kernel_stack_size`) 中固定下来。这样可以保证内核在跳转前已拥有合适的栈空间。由于栈是一个运行时动态分配的区域, 不需要保存在 ELF 文件中, 所以 ELF 文件不会描述栈的相关属性。栈的位置不能随意放置, 而是需要遵循系统设计中的地址布局安排, 否则可能破坏内存隔离和安全保护。

## 2.2 调试过程

### 2.2.1 layout asm 指令

`layout asm` 是 GDB 的 TUI (文本用户界面) 模式下用于显示汇编代码的指令。它会将 GDB 的窗口分割为几个区域, 其中一个区域显示当前执行地址附近的汇编指令, 这有助于观察低级代码及调试汇编级别的问题。

若要找到当前运行内核对应的 Rust 源码, 可以使用:

```
1 info line *$pc
```

该指令将根据当前程序计数器（PC）的值显示对应的源文件名和代码行信息，从而帮助定位源代码位置。如果内核编译时包含调试信息，你也可以切换到源码布局（例如使用 `layout src`）来直接查看源代码。

### 2.2.2 DBG\_INFO 编译选项

如果在编译时没有启用 `DBG_INFO=true`，生成的 ELF 文件中就不会包含调试符号和源代码映射信息。这会导致调试过程发生以下变化：

- 无法显示源代码：调试器（如 GDB）无法将当前执行地址映射回对应的 Rust 源码，因此你只能看到反汇编输出，而看不到源码行信息。
- 断点设置受限：无法基于函数名或源码行设置断点，只能使用内存地址来设置断点，这会降低调试的便捷性。
- 调试信息不足：常用的调试命令（如 `info line *$pc`、`layout src`）无法提供详细信息，定位问题时只能依赖反汇编和符号推断。
- 调试效率降低：没有调试信息，理解程序逻辑和内核行为会变得更困难，调试整个内核时需要更多手动分析汇编代码和地址映射。

因此，为了便于调试，通常会在开发和调试阶段启用调试信息（`DBG_INFO=true`），而在发布版本中可能关闭它以减小二进制文件大小和提高安全性。

### 2.2.3 调试环境

#### 1. 选择调试环境

我通常会选择 VS Code 作为开发环境，然后安装 LLDB 插件（例如 CodeLLDB）进行调试。这种环境集成了源代码、变量、寄存器、调用栈、断点设置等功能，使得调试内核或者低级代码时可以直观地跟踪程序执行。在调试配置（`launch.json`）中，将 `target` 指定为生成的 ELF 文件，同时配置调试参数，比如调试符号路径、架构等。

#### 2. 调试界面中的信息

在 VS Code LLDB 调试期间，GUI 界面通常会展示以下内容：

- 源代码窗口：显示当前执行位置对应的源代码或者反汇编（当没启用调试信息时）。
- 调用栈：列出函数调用链，可以点击进入某个调用层级查看源码或反汇编。
- 变量视图：显示局部变量、全局变量（例如 `BootInfo` 数据结构）和寄存器变量的值。
- 断点窗口：列出当前设置的所有断点，并允许快速添加、删除或编辑断点。

- 调试控制台：实时显示 LLDB 的输出信息，同时可以输入调试命令，比如 `frame info` 或 `register read` 等命令。
- 寄存器窗口（有时以扩展窗格呈现）：展示 CPU 寄存器内容，如 RIP、RSP、RDI 等，可帮助理解调用约定和参数传递。

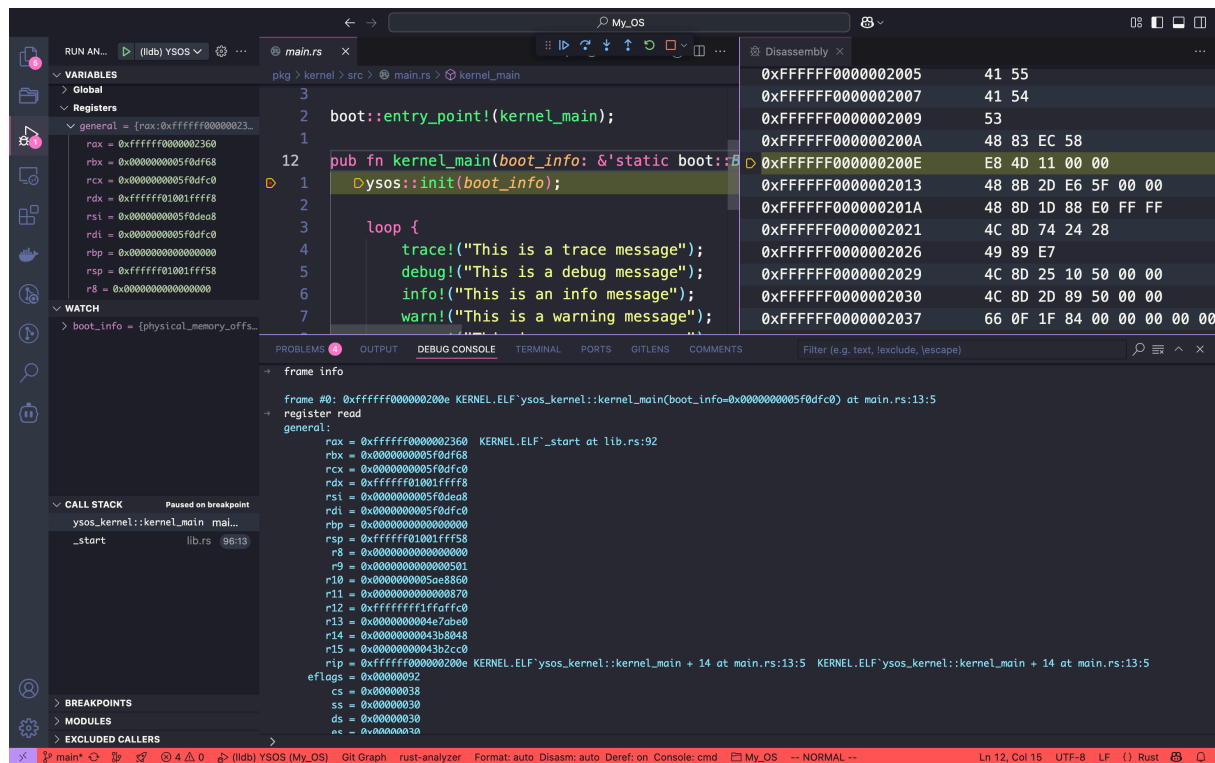


图 2.2.2: VS Code LLDB 调试界面

使用 VS Code LLDB 调试器时，这些窗口协同工作，可以清楚看到当前代码执行、变量状态、调用流程以及汇编代码，当你在断点处暂停时，还可在调试控制台输入 LLDB 命令来检查寄存器、内存映射等信息。

## 3 UART 与日志输出

### 3.1 串口驱动的实现

#### 3.1.1 初始化

```
1 pub fn init(&self) {
2     // FIXME: Initialize the serial port
3     unsafe {
```

Rust

```
4      // Disable interrupts
5      outb(self.port + 1, 0x00);
6      // Enable DLAB (set baud rate divisor)
7      outb(self.port + 3, 0x80);
8      // Set divisor to 1 (lo byte) 115200 baud
9      outb(self.port + 0, 0x01);
10     // Set divisor hi byte to 0
11     outb(self.port + 1, 0x00);
12     // 8 bits, no parity, one stop bit
13     outb(self.port + 3, 0x03);
14     // Enable FIFO, clear them, with 14-byte threshold
15     outb(self.port + 2, 0xC7);
16     // IRQs enabled, RTS/DSR set
17     outb(self.port + 4, 0x0B);
18 }
19 }
```

### 3.1.2 发送数据

```
1 /// Sends a byte on the serial port. Rust
2 pub fn send(&mut self, data: u8) {
3     // FIXME: Send a byte on the serial port
4     while unsafe { inb(self.port + 5) } & 0x20 == 0 {}
5     unsafe {
6         outb(self.port, data);
7     }
8 }
```

### 3.1.3 接收数据

```
1 /// Receives a byte on the serial port no wait. Rust
2 pub fn receive(&mut self) -> Option<u8> {
3     // FIXME: Receive a byte on the serial port no wait
4     if unsafe { inb(self.port + 5) } & 0x01 != 0 {
5         return Some(unsafe { inb(self.port) });
6     }
7     None
8 }
```

## 3.2 日志输出




### 3.2.1 Initialize

```

1  /// 初始化日志系统, 并根据传入的字符串设置日志级别
2  pub fn init(log_level: &str) {
3      static LOGGER: Logger = Logger;
4      log::set_logger(&LOGGER).unwrap();
5
6      // 根据启动参数设置日志级别
7      let level = parse_log_level(log_level);
8      log::set_max_level(level);
9
10     info!("Logger Initialized. (log_level = {})", log_level);
11 }

```

 Rust

### 3.2.2 Log

```

1  impl log::Log for Logger {
2      fn enabled(&self, _metadata: &Metadata) -> bool {
3          true
4      }
5
6      fn log(&self, record: &Record) {
7          // FIXME: Implement the logger with serial output
8          interrupts::without_interrupts(|| {
9              if let Some(mut serial) = get_serial() {
10                 // 直接使用固定前缀避免使用 format! 宏
11                 let prefix = match record.level() {
12                     Level::Info    => " INFO",
13                     Level::Warn    => " WARN",
14                     Level::Error   => "ERROR",
15                     Level::Debug   => "DEBUG",
16                     Level::Trace   => "TRACE",
17                 };
18                 let color = level_color(record.level());
19
20                 // 固定前缀宽度采用简单的右侧填空 (这里假设所有前缀长度已经固定)
21                 if record.level() == Level::Info {
22                     let _ = writeln!(
23                         serial,
24                         "{}[{:<5}]{} {}",
25                         color,

```

 Rust

```
26         prefix,  
27         RESET_COLOR,  
28         record.args()  
29     );  
30     } else if record.level() == Level::Warn {  
31         // WARNING : 采用黄色、加粗和下划线, 同时前缀宽度固定  
32         let _ = writeln!(  
33             serial,  
34             "{}[\x1b[1m\x1b[4m{:<5}]{}{}]",  
35             color,  
36             prefix,  
37             RESET_COLOR,  
38             color,  
39             record.args(),  
40             RESET_COLOR  
41         );  
42     } else if record.level() == Level::Error {  
43         // ERROR : 采用红色和加粗, 前缀固定宽度  
44         let _ = writeln!(  
45             serial,  
46             "{}\x1b[1m[{:<5}]{}{}]",  
47             "\x1b[31m",  
48             prefix,  
49             record.args(),  
50             RESET_COLOR  
51         );  
52     } else {  
53         // 对于其他日志级别直接以同样的格式输出  
54         let _ = writeln!(  
55             serial,  
56             "{}[{:<5}]{} {}",  
57             color,  
58             prefix,  
59             RESET_COLOR,  
60             record.args()  
61         );  
62     }  
63 }  
64 });  
65 }  
66 fn flush(&self) {}  
67 }
```



## 4 思考题

### 4.1 Cargo.toml

在根目录 Cargo.toml 中将 boot 包的默认 feature 禁用，主要目的是防止 boot 包默认开启的一些功能会引入不适合内核构建环境的依赖或行为，比如依赖标准库支持、默认的 panic 处理程序或全局分配器实现。内核通常运行在 no\_std 环境下，需要严格掌控内存分配、异常处理和其他底层行为，而 boot 包默认的 feature 可能会开启那些与内核设计冲突的代码。

结合 Cargo.toml 中的设置，boot 包在 [features] 下定义了“boot” feature，该 feature 显式地启用了“uefi/alloc”、“uefi/logger”、“uefi/panic\_handler”和“uefi/global\_allocator”。这样，内核开发者可以在需要时精确控制引入哪些功能，而不是使用 boot 包默认开启的所有功能。禁用默认 feature 有助于：

- 避免不必要的依赖：防止默认 feature 依赖于标准库或其他与 no\_std 环境不兼容的库；
- 防止功能冲突：避免 boot 包默认配置中的 panic\_handler 或全局分配器与内核自己实现的部分产生冲突；
- 精确控制功能：确保内核只使用显式启用的那些、经过验证且适合内核环境的功能和实现。

因此，通过在根目录 Cargo.toml 中对 boot 包使用 default-features = false，内核可以更好地控制引导加载过程中的行为，保证整体系统的一致性和安全性。

### 4.2 max\_phys\_addr

下面解释一下 max\_phys\_addr 的计算过程及其目的：

#### 1. 计算过程：

```
1 let max_phys_addr = mmap
2   .entries()
3   .map(|m| m.phys_start + m.page_count * 0x1000)
4   .max()
5   .unwrap()
6   .max(0x1_0000_0000); // include IOAPIC MMIO area
```

- 代码首先根据 UEFI 的内存映射获取所有内存区域（entries）。

- 对于每一个区域，计算该区域的上界，即起始物理地址 `m.phys_start` 加上该区域所占的页数 (`m.page_count`) 乘以页大小 (0x1000，即 4096 字节)。这表示该区域在物理内存中的结束地址。
- 使用 `.max()` 取所有计算结果中的最大值，得到整个系统的物理内存上界。
- 最后，再用 `.max(0x1_0000_0000)` 确保结果至少为 0x1\_0000\_0000，这主要是确保包括 IOAPIC 及其它内存映射 I/O 设备所在的区域（通常在 4GB 附近）。

## 2. 为什么这么做?

- **确定映射范围：** 在后续将物理内存映射到虚拟地址时，需要知道实际物理内存的最大范围，才能准确地建立页表映射。这样内核便可以直接访问所有实际的内存区域。
- **考虑 MMIO 区域：** 有时 UEFI 内存映射中可能不包含某些内存映射 I/O (MMIO) 的区域（例如用于 IOAPIC 的地址）。通过用 `.max(0x1_0000_0000)` 强制取最大值至少为 4GB，可以确保这些地址区域也在映射范围内，防止遗漏对硬件设备寄存器的访问。
- **稳定性和兼容性：** 保证无论内存映射如何（如实际 RAM 小于 4GB），内核的虚拟内存映射总能把 IOAPIC 等设备所需的地址空间包含在内，从而避免运行时因访问不到这些关键外设而导致问题。

总之，`max_phys_addr` 的计算既确保了能覆盖全部实际的 RAM 区域，也强制包含一些必要的内存映射 I/O 区域，使得内核可以直接通过固定偏移映射来访问物理内存中的所有区域。

## 4.3 内容输出

在我们的设计中，内核本身只有在进入内核后才启用串口驱动进行日志输出，而在此之前的所有输出都是由 bootloader 提供的。实际上，bootloader 运行在 UEFI 环境下，它利用 UEFI 自身提供的文本输出服务来显示信息。也就是说：

- 在启动过程中，使用 UEFI 文本控制台 (Console Out) 输出启动信息和调试日志，因此你能在 UEFI 界面上看到启动流程的信息；
- 当 bootloader 退出 UEFI Boot Services 并跳转到内核前，所有关键的启动信息都已经通过 UEFI 控制台输出展示；
- 内核进入后会启用自己的串口驱动，此时日志输出切换到串口设备，而之前的输出则由 bootloader 完成。

调试时可以看到，在 bootloader 阶段通过调用诸如 `uefi::helpers::init()` 和 `log::info!(...)` 等接口打印的启动信息都显示在 UEFI 提供的标准输出上，这与后期内核使用串口驱动输出形成了一个衔接。

## 4.4 移除 -nographic 选项

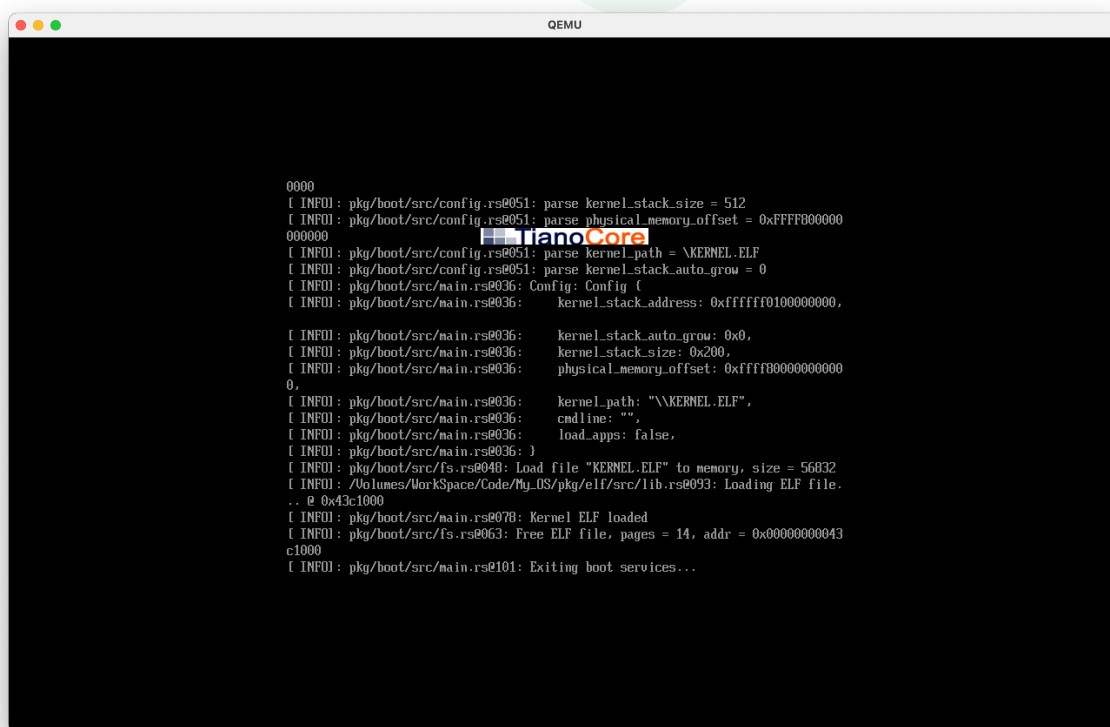
在当前项目中，通过在 QEMU 命令行中指定 **-nographic** 参数，可以禁用 QEMU 的图形输出，进而将串口输出重定向到主机的标准输入/输出（终端窗口）。这使得所有调试信息及内核输出都直接显示在命令行中，以便开发者观察。

如果在 Makefile 中取消了 **-nographic** 选项，则 QEMU 不再以纯文本模式启动，而是会打开一个默认的图形窗口（通常是 SDL 或 GTK 窗口）。在这种情况下，串口的输出不会自动显示在主机终端上，而是出现在这个图形窗口中，这使得你在命令行中无法直接看到内核启动后的串口消息。

观察执行：

```
1 make run QEMU_OUTPUT=""
```

Shell



```
0000
[ INFO ] pkg/boot/src/config.rs@051: parse kernel_stack_size = 512
[ INFO ] pkg/boot/src/config.rs@051: parse physical_memory_offset = 0xFFFF000000
000000
[ INFO ] pkg/boot/src/config.rs@051: parse kernel_path = \KERNEL.ELF
[ INFO ] pkg/boot/src/config.rs@051: parse kernel_stack_auto_grow = 0
[ INFO ] pkg/boot/src/main.rs@036: Config: Config {
[ INFO ] pkg/boot/src/main.rs@036:   kernel_stack_address: 0xffffffff00000000,

[ INFO ] pkg/boot/src/main.rs@036:   kernel_stack_auto_grow: 0x0,
[ INFO ] pkg/boot/src/main.rs@036:   kernel_stack_size: 0x200,
[ INFO ] pkg/boot/src/main.rs@036:   physical_memory_offset: 0xffff000000000000
0,
[ INFO ] pkg/boot/src/main.rs@036:   kernel_path: "\\KERNEL.ELF",
[ INFO ] pkg/boot/src/main.rs@036:   cmdline: "",
[ INFO ] pkg/boot/src/main.rs@036:   load_apps: false,
[ INFO ] pkg/boot/src/main.rs@036: }
[ INFO ] pkg/boot/src/fs.rs@040: Load file "KERNEL.ELF" to memory, size = 56832
[ INFO ] /Volumes/Workspace/Code/My_OS/pkg/elf/src/lib.rs@093: Loading ELF file.
.. @ 0x43c1000
[ INFO ] pkg/boot/src/main.rs@078: Kernel ELF loaded
[ INFO ] pkg/boot/src/fs.rs@063: Free ELF file, pages = 14, addr = 0x000000000043
c1000
[ INFO ] pkg/boot/src/main.rs@101: Exiting boot services...
```

图 4.4.3: QEMU 图形窗口中的串口输出


## 4.5 如何将串口重定向到宿主机的标准 I/O

可以利用 QEMU 的 `-serial stdio` 选项，该选项会将虚拟机的串口（通常是设备 COM1）重定向至标准输入输出。可以尝试以下两种方式：

### 1. 直接构造命令行

在命令行启动，可以输入：

```
1 make run QEMU_OUTPUT="-serial stdio"
```


 Shell

这样，即使没有 `-nographic`，所有串口输出都会输出到命令行里。

### 2. 修改 `ysos.py` 中的参数

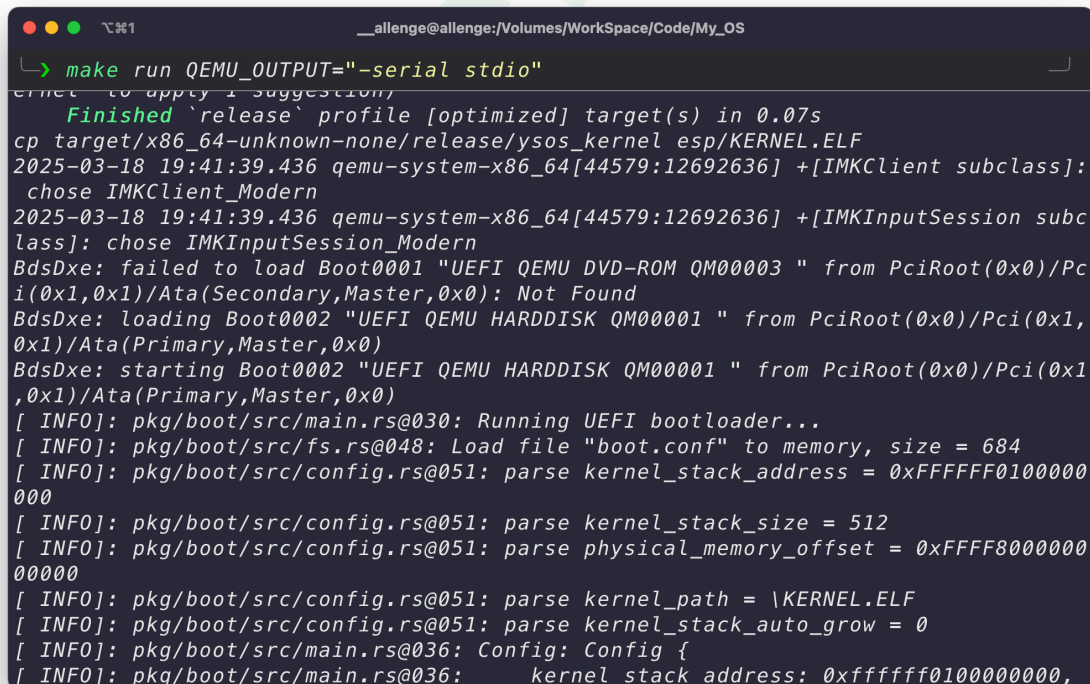
在 `ysos.py` 脚本中，目前 `qemu_args` 包含了 `-nographic` 参数默认值（通过 `-o` 参数传递）。可以将默认输出值改为 `-serial stdio`，例如：

```
1 parser.add_argument('-o', '--output', default='-serial
  stdio', help='Set output for qemu, default is -serial stdio')
```

 Python

这样，当执行 `make run` 或者用 `ysos.py` 启动 QEMU 时，会将串口输出重定向到宿主机的终端，而不是开启单独的图形窗口。

效果如下，可以看到终端显示了输出。



```
__allenge@allenge:/Volumes/WorkSpace/Code/My_OS
> make run QEMU_OUTPUT="-serial stdio"
Finished `release` profile [optimized] target(s) in 0.07s
cp target/x86_64-unknown-none/release/ysos_kernel esp/KERNEL.ELF
2025-03-18 19:41:39.436 qemu-system-x86_64[44579:12692636] +[IMKClient subclass]:
  chose IMKClient_Modern
2025-03-18 19:41:39.436 qemu-system-x86_64[44579:12692636] +[IMKInputSession subclass]: chose IMKInputSession_Modern
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 "UEFI QEMU HARDDISK QM00001 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)
BdsDxe: starting Boot0002 "UEFI QEMU HARDDISK QM00001 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)
[ INFO]: pkg/boot/src/main.rs@030: Running UEFI bootloader...
[ INFO]: pkg/boot/src/fs.rs@048: Load file "boot.conf" to memory, size = 684
[ INFO]: pkg/boot/src/config.rs@051: parse kernel_stack_address = 0xFFFFF0100000000
[ INFO]: pkg/boot/src/config.rs@051: parse kernel_stack_size = 512
[ INFO]: pkg/boot/src/config.rs@051: parse physical_memory_offset = 0xFFFF800000000000
[ INFO]: pkg/boot/src/config.rs@051: parse kernel_path = \KERNEL.ELF
[ INFO]: pkg/boot/src/config.rs@051: parse kernel_stack_auto_grow = 0
[ INFO]: pkg/boot/src/main.rs@036: Config: Config {
[ INFO]: pkg/boot/src/main.rs@036:   kernel_stack_address: 0xfffff0100000000,
```

图 4.5.4: QEMU 串口输出重定向至终端

## 5 附加题

### 5.1 bitflags

在 Rust 中，bitflags 宏库提供了一种方便的方式来定义和操作位标志（bit flags）。通过 bitflags! 宏，可以定义一个包含多个位标志的枚举类型，然后使用位运算符和方法来设置、清除和检查这些标志。

```

1  bitflags! {
2      /// UART 16550 的 LCR (Line Control Register) 寄存器控制标志
3      pub struct LcrFlags: u8 {
4          /// DLAB: Divisor Latch Access Bit, 用于访问波特率分频寄存器
5          const DLAB          = 0x80;
6          /// 设置字长 5 位 (0b00)
7          const WORD_LENGTH_5 = 0x00;
8          /// 设置字长 6 位 (0b01)
9          const WORD_LENGTH_6 = 0x01;
10         /// 设置字长 7 位 (0b10)
11         const WORD_LENGTH_7 = 0x02;
12         /// 设置字长 8 位 (0b11)
13         const WORD_LENGTH_8 = 0x03;
14         /// STOP: 设置停止位, 若设置则使用 2 个停止位 (5位字长时为 1.5 个停止位)
15         const STOP_BITS     = 0x04;
16         /// PARITY_ENABLE: 启用奇偶校验
17         const PARITY_ENABLE = 0x08;
18         /// EVEN_PARITY: 偶校验 (与 PARITY_ENABLE 配合使用)
19         const EVEN_PARITY   = 0x10;
20         /// STICK_PARITY: 固定奇偶模式 (与 PARITY_ENABLE 配合使用)
21         const STICK_PARITY  = 0x20;
22     }
23 }
```

### 5.2 清屏并输出学号

在字符串最前面添加了清屏的 escape sequence (\x1B[2J\x1B[H)，并在末尾添加了学号信息。

```

1  #[macro_use]
2  mod macros;
3
```



```
11
12 struct Logger;
13
14 impl log::Log for Logger {
15     fn enabled(&self, _metadata: &Metadata) -> bool {
16         true
17     }
18
19     fn log(&self, record: &Record) {
20         // FIXME: Implement the logger with serial output
21         interrupts::without_interrupts(|| {
22             if let Some(mut serial) = get_serial() {
23                 let prefix = match record.level() {
24                     Level::Info => " INFO",
25                     Level::Warn => " WARN",
26                     Level::Error => "ERROR",
27                     Level::Debug => "DEBUG",
28                     Level::Trace => "TRACE",
29                 };
30                 let color = level_color(record.level());
31
32                 // 固定前缀宽度采用简单的右侧填空（这里假设所有前缀长度已经固定）
33                 if record.level() == Level::Info {
34                     let _ = writeln!(
35                         serial,
36                         "{}[{:<5}]{} {}",
37                         color,
38                         prefix,
39                         RESET_COLOR,
40                         record.args()
41                     );
42                 } else if record.level() == Level::Warn {
43                     // WARNING : 采用黄色、加粗和下划线，同时前缀宽度固定
44                     let _ = writeln!(
45                         serial,
46                         "{}[\x1b[1m\x1b[4m{:<5}]{}{}] {}",
47                         color,
48                         prefix,
49                         RESET_COLOR,
50                         color,
51                         record.args(),
52                         RESET_COLOR
```

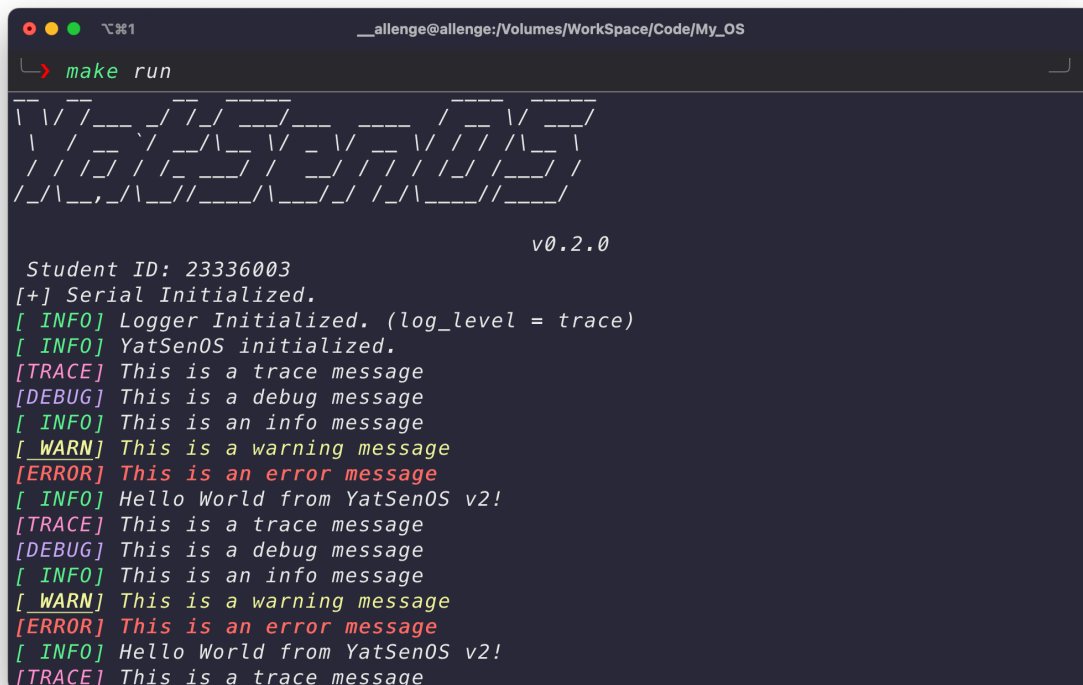


```
53         );
54     } else if record.level() == Level::Error {
55         // ERROR : 采用红色和加粗, 前缀固定宽度
56         let _ = writeln!(
57             serial,
58             "{}\x1b[1m[{:<5}] {}{}",
59             "\x1b[31m",
60             prefix,
61             record.args(),
62             RESET_COLOR
63         );
64     } else {
65         // 对于其他日志级别直接以同样的格式输出
66         let _ = writeln!(
67             serial,
68             "{}[{:<5}]{} {}",
69             color,
70             prefix,
71             RESET_COLOR,
72             record.args()
73         );
74     }
75 }
76 });
77 }
78
79 fn flush(&self) {}
80 }
```



## 最终启动效果展示

在启动时，清屏后显示了学号信息，并根据启动参数设置了日志级别，输出了初始化日志。



```
__allenge@allenge:/Volumes/WorkSpace/Code/My_OS
> make run

v0.2.0

Student ID: 23336003
[+] Serial Initialized.
[ INFO] Logger Initialized. (log_level = trace)
[ INFO] YatSen0S initialized.
[TRACE] This is a trace message
[DEBUG] This is a debug message
[ INFO] This is an info message
[ WARN] This is a warning message
[ERROR] This is an error message
[ INFO] Hello World from YatSen0S v2!
[TRACE] This is a trace message
[DEBUG] This is a debug message
[ INFO] This is an info message
[ WARN] This is a warning message
[ERROR] This is an error message
[ INFO] Hello World from YatSen0S v2!
[TRACE] This is a trace message
```

图 5.3.5: 最终效果

## 5.4 调试器观测

下面使用 LLDB 调试器检查内核初始化后寄存器和内存段信息：

1. 启动调试会话后，内核运行到断点时会自动暂停，此时 可以使用以下指令查看关键寄存器的状态：

```
1 (lldb) register read
```

其中：

- **rsp**：显示当前栈指针的值，它应指向内核堆栈的顶部。
- **rbp**：显示当前栈帧指针的值，可用于分析函数调用栈。
- **rip**：显示指令指针，指向当前执行的指令地址；你可以用它判断当前处于 kernel\_main 内哪一行。

结果如图所示。

```

→ register read
general:
  rax = 0x0000000000000005
  rbx = 0xffffffff0000000a9
  rcx = 0x0000000000000000
  rdx = 0x000000000000003f8
  rsi = 0xffffffff000000325  KERNEL.ELF`ysos_kernel::utils::logger::init::LOGGER::he142b46769a02ee9 (.llvm.6012415823576687107) + 54
  rdi = 0x000000000000003f8
  rbp = 0xffffffff0000009018  KERNEL.ELF`log::MAX_LOG_LEVEL_FILTER  KERNEL.ELF`log::MAX_LOG_LEVEL_FILTER
  rsp = 0xffffffff01001fff58
  r8 = 0x000000000000003fd
  r9 = 0x000000000000000a
  r10 = 0xffffffff00000071b0
  r11 = 0x0000000000000001
  r12 = 0xffffffff0000007040
  r13 = 0xffffffff00000070c0
  r14 = 0xffffffff01001fff80
  r15 = 0xffffffff01001fff58
  rip = 0xffffffff0000002048  KERNEL.ELF`ysos_kernel::kernel_main + 72 at main.rs:16:9  KERNEL.ELF`ysos_kernel::kernel_main + 72 at main.rs:16:9
  eflags = 0x00000046
  cs = 0x00000038
  ss = 0x00000030
  ds = 0x00000030
  es = 0x00000030
  fs = 0x00000030
  gs = 0x00000030
  fs_base = 0x0000000000000000
  gs_base = 0x0000000000000000
  k_gs_base = 0x0000000000000000
  cr0 = 0x00000000000010033
  cr2 = 0x0000000000000000
  cr3 = 0x0000000000005c01000
  cr4 = 0x00000000000000668
  cr8 = 0x0000000000000000

```

图 5.4.6: LLDB 寄存器状态

可以看见，

```

1  rip = 0xffffffff0000002048  KERNEL.ELF`ysos_kernel::kernel_main + 72 at
   main.rs:16:9  KERNEL.ELF`ysos_kernel::kernel_main + 72 at main.rs:16:9
2  rbp = 0xffffffff0000009018  KERNEL.ELF`log::MAX_LOG_LEVEL_FILTER
   KERNEL.ELF`log::MAX_LOG_LEVEL_FILTER
3  rsp = 0xffffffff01001fff58

```

### 1. 指令指针 (RIP)

- 当前的 RIP 为 0xffffffff0000002048，对应符号 `ysos_kernel::kernel_main + 72`，这表示程序已经进入内核入口函数 `kernel_main`，并且执行到了函数起始地址偏移 72 字节处。
- 根据调试信息，文件 `main.rs` 的第 16 行第 9 列处正是当前位置。这说明内核初始化 (`ysos::init(boot_info)` 调用) 已经执行完毕，进入了循环体（或其它后续代码）的执行阶段。

### 2. 栈帧指针 (RBP)

- RBP 为 0xffffffff0000009018。当前值指向一个与符号 `log::MAX_LOG_LEVEL_FILTER` 关联的位置。这可能说明在内核初始化过程中，调用了日志相关的代码，当前 RBP 记录了上层调用者的信息或内核堆栈的某个边界。

### 3. 栈指针 (RSP)

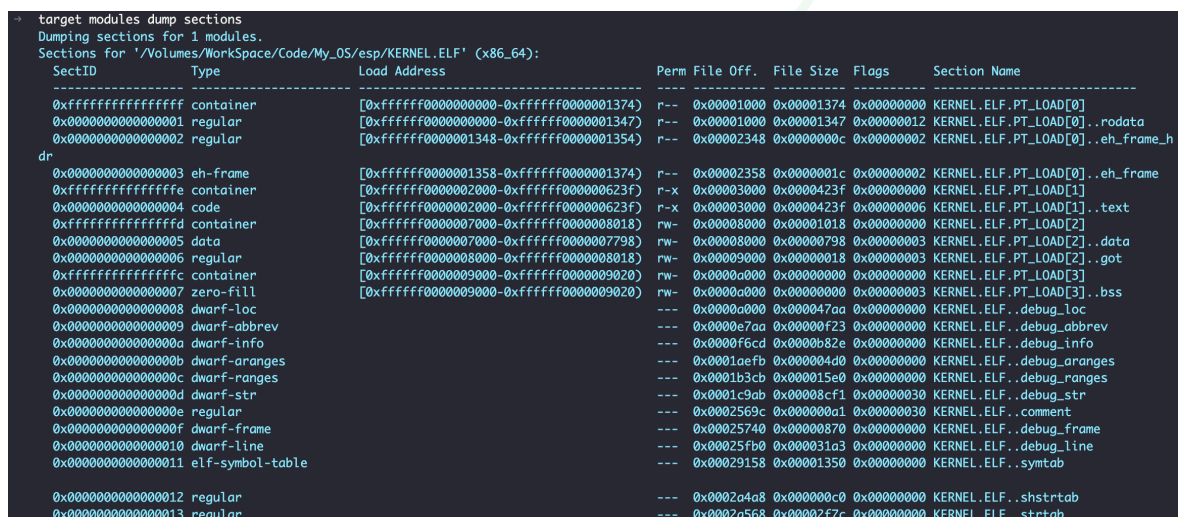
- RSP 为 0xfffff01001fff58，表明当前内核使用的栈位于一块高地址区域。这个地址由内核链接脚本和堆栈初始化时确定，确保内核具有足够的栈空间并且与其他内存区域隔离。

2. 查看内存段信息 要检查内核代码、数据、BSS 等段在内存中的分布，可以使用 LLDB 的模块和段信息命令。

```
1 (lldb) target modules dump sections
```

这些命令会列出加载的模块（例如内核镜像）以及各个段的虚拟地址范围、大小和属性。从输出中可以查找：

- **.text** 段：通常标记为 R-X（只读、可执行）
- **.data** 段：标记为 RW（可读写）
- **.bss** 段：同样为 RW，但在 ELF 文件中没有实际存储数据，仅在内存中分配区域如下图所示。



```
target modules dump sections
Dumping sections for 1 modules.
Sections for '/Volumes/Workspace/Code/My_OS/esp/KERNEL.ELF' (x86_64):
```

SectID	Type	Load Address	Perm	File Off.	File Size	Flags	Section Name
0xffffffffffffffff	container	[0xffffffff000000000-0xffffffff0000001374)	r--	0x00001000	0x00001374	0x00000000	KERNEL.ELF.PT_LOAD[0]
0x0000000000000001	regular	[0xffffffff000000000-0xffffffff0000001347)	r--	0x00001000	0x00001347	0x00000012	KERNEL.ELF.PT_LOAD[0]..rodata
0x0000000000000002	regular	[0xffffffff0000001348-0xffffffff0000001354)	r--	0x00002348	0x0000000c	0x00000002	KERNEL.ELF.PT_LOAD[0]..eh_frame_h
dr							
0x0000000000000003	eh-frame	[0xffffffff0000001358-0xffffffff0000001374)	r--	0x00002358	0x0000001c	0x00000002	KERNEL.ELF.PT_LOAD[0]..eh_frame
0xfffffffffffffffe	container	[0xffffffff0000002000-0xffffffff000000623f)	r-x	0x00003000	0x0000423f	0x00000000	KERNEL.ELF.PT_LOAD[1]
0x0000000000000004	code	[0xffffffff0000002000-0xffffffff000000623f)	r-x	0x00003000	0x0000423f	0x00000006	KERNEL.ELF.PT_LOAD[1]..text
0xfffffffffffffffd	container	[0xffffffff0000007000-0xffffffff0000008018)	rw-	0x00008000	0x00001018	0x00000000	KERNEL.ELF.PT_LOAD[2]
0x0000000000000005	data	[0xffffffff0000007000-0xffffffff0000007798)	rw-	0x00008000	0x00000798	0x00000003	KERNEL.ELF.PT_LOAD[2]..data
0x0000000000000006	regular	[0xffffffff0000008000-0xffffffff0000008018)	rw-	0x00009000	0x00000018	0x00000003	KERNEL.ELF.PT_LOAD[2]..got
0xfffffffffffffffc	container	[0xffffffff0000009000-0xffffffff0000009020)	rw-	0x0000a000	0x00000020	0x00000000	KERNEL.ELF.PT_LOAD[3]
0x0000000000000007	zero-fill	[0xffffffff0000009000-0xffffffff0000009020)	rw-	0x0000a000	0x00000020	0x00000003	KERNEL.ELF.PT_LOAD[3]..bss
0x0000000000000008	dwarf-loc	[0xffffffff0000009000-0xffffffff0000009020)	---	0x0000a000	0x000047aa	0x00000000	KERNEL.ELF..debug_loc
0x0000000000000009	dwarf-abbrev		---	0x0000e7aa	0x00000f23	0x00000000	KERNEL.ELF..debug_abbrev
0x000000000000000a	dwarf-info		---	0x0000f6cd	0x0000b82e	0x00000000	KERNEL.ELF..debug_info
0x000000000000000b	dwarf-aranges		---	0x0001aefb	0x000004d0	0x00000000	KERNEL.ELF..debug_aranges
0x000000000000000c	dwarf-ranges		---	0x0001b3cb	0x000015e0	0x00000000	KERNEL.ELF..debug_ranges
0x000000000000000d	dwarf-str		---	0x0001c9ab	0x00008cf1	0x00000030	KERNEL.ELF..debug_str
0x000000000000000e	regular		---	0x0002569c	0x000000a1	0x00000030	KERNEL.ELF..comment
0x000000000000000f	dwarf-frame		---	0x00025740	0x00000870	0x00000000	KERNEL.ELF..debug_frame
0x0000000000000010	dwarf-line		---	0x00025fb0	0x000031a3	0x00000000	KERNEL.ELF..debug_line
0x0000000000000011	elf-symbol-table		---	0x00029158	0x00001350	0x00000000	KERNEL.ELF..symtab
0x0000000000000012	regular		---	0x0002a4a8	0x000000c0	0x00000000	KERNEL.ELF..shstrtab
0x0000000000000013	regular		---	0x0002a568	0x00002f7c	0x00000000	KERNEL.ELF..strtab

图 5.4.7: LLDB 段信息

可以看到：

- 内核的代码段（.text）从 0xffffffff0000002000 开始，是只读可执行的；
- 数据段（.data）从 0xffffffff0000007000 开始，是可读写的；
- BSS 段从 0xffffffff0000009000 开始，也是可读写的。