|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| 北京交通大学软件学院  **《操作系统》课程**  **实验报告** | | |

|  |
| --- |
| **姓名: 张鑫成** |
| **学号: 20271055** |

目录

[1 实验目的 2](#_Toc2613)

[2 实验过程设计 3](#_Toc10991)

[3 源代码及注释 3](#_Toc31697)

[4 运行结果与分析 7](#_Toc28975)

[5 实验总结 10](#_Toc5496)

# 1 实验目的

（说明：写出本次实验的主要实验目的与原因。）

实现裸机上的执行环境以及一个最小化的操作系统内核。

# 2 实验过程设计

（说明：写出本次实验的主要实验流程或个人实验的实施过程。）

1. 学习了编写操作系统内核前置知识，更好地理解实践操作背后的动机和目的。
2. 编写进入内核后的第一条指令，这样更方便我们验证我们的内核镜像是否正确对接到 Qemu 上。
3. 通过 链接脚本 (Linker Script) 调整链接器的行为，使得最终生成的可执行文件的内存布局符合我们的预期。我们修改 Cargo 的配置文件来使用我们自己的链接脚本 os/src/linker.ld 而非使用默认的内存布局。
4. 分配并使用启动栈。
5. 使用 RustSBI 提供的服务。
6. 实现格式化输出。
7. 处理致命错误。

# 3 源代码及注释

（说明：写出本次实验使用的命令或编写的源代码。请在下列表格中说明源代码的文件名和代码功能概述或者命令名和该命令的主要作用。要求针对核心功能代码进行注释。）

表 3-1 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/.cargo/config |
| **主要功能:** | 配置文件,使用我们自己的链接脚本 |
| **源代码:**   1. # os/.cargo/config 2. [build]target = "riscv64gc-unknown-none-elf" 3. [target.riscv64gc-unknown-none-elf]rustflags = [ 4. "-Clink-arg=-Tsrc/linker.ld", "-Cforce-frame-pointers=yes" 5. ] | |

表 3-2 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/linker.ld |
| **主要功能:** | 调整链接器的行为，使得最终生成的可执行文件的内存布局符合我们的预期。 |
| **源代码:**   1. OUTPUT\_ARCH(riscv)ENTRY(\_start)BASE\_ADDRESS = 0x80200000; 2. SECTIONS{ 3. . = BASE\_ADDRESS; 4. skernel = .; 5. stext = .; 6. .text : { 7. \*(.text.entry) 8. \*(.text .text.\*) 9. } 10. . = ALIGN(4K); 11. etext = .; 12. srodata = .; 13. .rodata : { 14. \*(.rodata .rodata.\*) 15. \*(.srodata .srodata.\*) 16. } 17. . = ALIGN(4K); 18. erodata = .; 19. sdata = .; 20. .data : { 21. \*(.data .data.\*) 22. \*(.sdata .sdata.\*) 23. } 24. . = ALIGN(4K); 25. edata = .; 26. .bss : { 27. \*(.bss.stack) 28. sbss = .; 29. \*(.bss .bss.\*) 30. \*(.sbss .sbss.\*) 31. } 32. . = ALIGN(4K); 33. ebss = .; 34. ekernel = .; 35. /DISCARD/ : { 36. \*(.eh\_frame) 37. }} | |

表 3-3 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/entry.asm |
| **主要功能:** | 分配启动栈空间 |
| **源代码:**   1. # os/src/entry.asm 2. .section .text.entry 3. .globl \_start\_start: 4. la sp, boot\_stack\_top 5. call rust\_main 6. .section .bss.stack 7. .globl boot\_stackboot\_stack: 8. .space 4096 \* 16 9. .globl boot\_stack\_topboot\_stack\_top: | |

表 3-4 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/sbi.rs |
| **主要功能:** | 提供输出、关键服务 |
| **源代码:**   1. // os/src/sbi.rs 2. #![allow(unused)] // 此行请放在该文件最开头 3. const SBI\_SET\_TIMER: usize = 0; 4. const SBI\_CONSOLE\_PUTCHAR: usize = 1; 5. const SBI\_CONSOLE\_GETCHAR: usize = 2; 6. const SBI\_CLEAR\_IPI: usize = 3; 7. const SBI\_SEND\_IPI: usize = 4; 8. const SBI\_REMOTE\_FENCE\_I: usize = 5; 9. const SBI\_REMOTE\_SFENCE\_VMA: usize = 6; 10. const SBI\_REMOTE\_SFENCE\_VMA\_ASID: usize = 7; 11. const SBI\_SHUTDOWN: usize = 8; 12. use core::arch::asm; 13. #[inline(always)] 14. fn sbi\_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize { 15. let mut ret; 16. unsafe { 17. asm!( 18. "ecall", 19. inlateout("x10") arg0 => ret, 20. in("x11") arg1, 21. in("x12") arg2, 22. in("x17") which, 23. ); 24. } 25. ret 26. } 27. pub fn console\_putchar(c: usize) { 28. sbi\_call(SBI\_CONSOLE\_PUTCHAR, c, 0, 0); 29. } 30. pub fn shutdown() -> ! { 31. sbi\_call(SBI\_SHUTDOWN, 0, 0, 0); 32. panic!("It should shutdown!"); 33. } | |

表 3-5 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/console.rs |
| **主要功能:** | 编写 println! 宏 |
| **源代码:**   1. // os/src/console.rsuse crate::sbi::console\_putchar;use core::fmt::{self, Write}; 2. struct Stdout; 3. impl Write for Stdout { 4. fn write\_str(&mut self, s: &str) -> fmt::Result { 5. for c in s.chars() { 6. console\_putchar(c as usize); 7. } 8. Ok(()) 9. }} 10. pub fn print(args: fmt::Arguments) { 11. Stdout.write\_fmt(args).unwrap();} 12. #[macro\_export]macro\_rules! print { 13. ($fmt: literal $(, $($arg: tt)+)?) => { 14. $crate::console::print(format\_args!($fmt $(, $($arg)+)?)); 15. }} 16. #[macro\_export]macro\_rules! println { 17. ($fmt: literal $(, $($arg: tt)+)?) => { 18. $crate::console::print(format\_args!(concat!($fmt, "\n") $(, $($arg)+)?)); 19. }} | |

表 3-6 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/lang\_items.rs |
| **主要功能:** | 处理致命错误 |
| **源代码:**   1. use crate::sbi::shutdown;use core::panic::PanicInfo; 2. #[panic\_handler]fn panic(info: &PanicInfo) -> ! { 3. if let Some(location) = info.location() { 4. println!( 5. "Panicked at {}:{} {}", 6. location.file(), 7. location.line(), 8. info.message().unwrap() 9. ); 10. } else { 11. println!("Panicked: {}", info.message().unwrap()); 12. } 13. shutdown()} | |

表 3-6 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | oos/src/main.rs |
| **主要功能:** | 主函数 |
| **源代码:**   1. // os/src/main.rs 2. #![feature(panic\_info\_message)] 3. #![no\_main] 4. #![no\_std] 5. #[macro\_use] 6. mod console; 7. mod lang\_items; 8. mod sbi; 9. use core::arch::global\_asm; 10. global\_asm!(include\_str!("entry.asm")); 11. #[no\_mangle] 12. pub fn rust\_main() -> ! { 13. clear\_bss(); 14. println!("Hello, world!"); 15. panic!("Shutdown machine!"); 16. } 17. fn clear\_bss() { 18. extern "C" { 19. fn sbss(); 20. fn ebss(); 21. } 22. (sbss as usize..ebss as usize).for\_each(|a| { 23. unsafe { (a as \*mut u8).write\_volatile(0) }   });  } | |

# 4 运行结果与分析

（说明：此部分请写出代码或命令运行结果及其分析。运行结果可以通过截图的方式提交,此外需要对截图进行分析与说明。）

启动 Qemu 并运行内核:

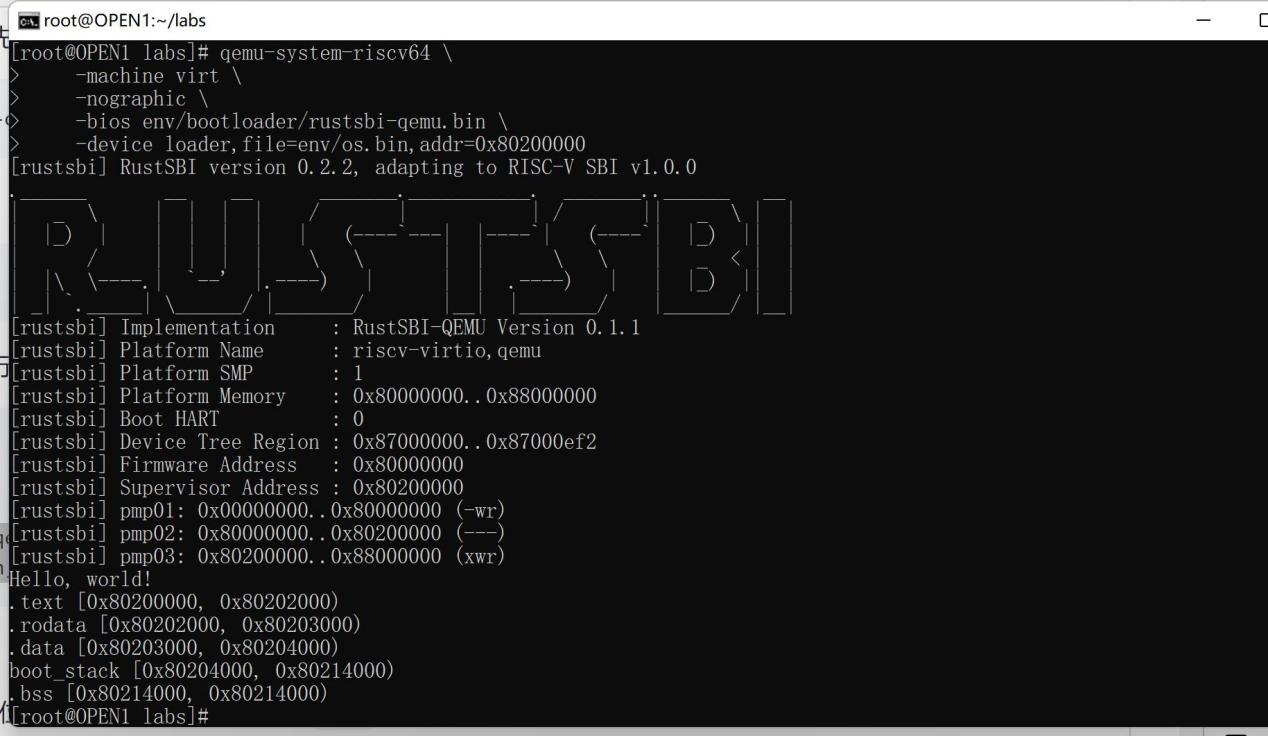


图4.1 Qumu运行图

编写内核第一条指令并嵌入到我们的内核项目中，同时在main.rs中嵌入这段汇编代码。

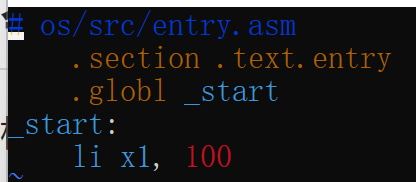


图4.2 编写内核第一条指令图

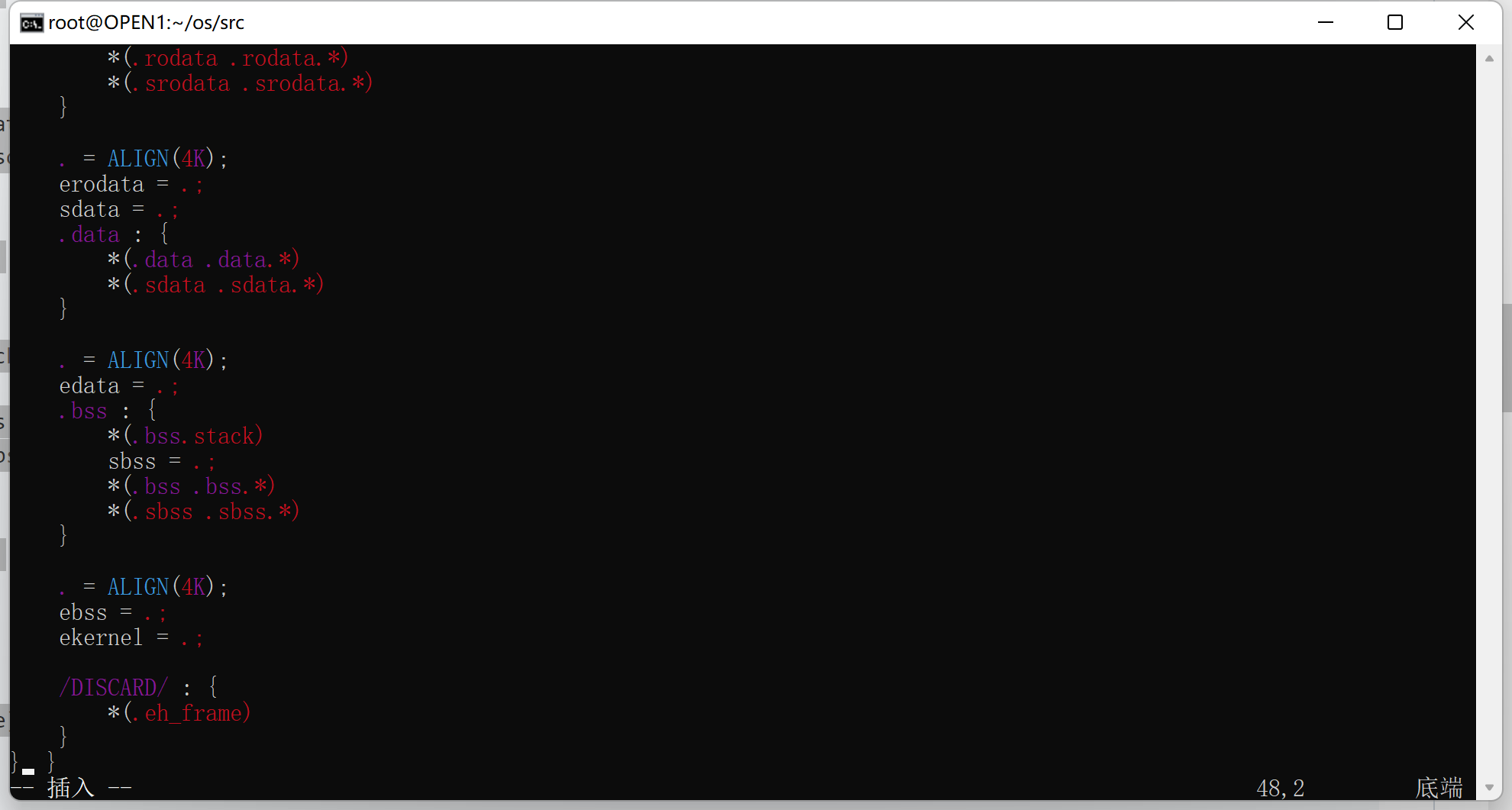
调整内核内存布局，链接脚本调整链接器的行为，修改Cargo的配置文件来使用我们自己的链接脚本os/src/linker.ld而非使用默认的内存布局。

图4.3 调整内核布局图

以release模式生成了内核可执行文件。

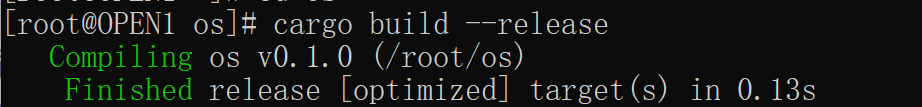


图4.4 内核可执行文件生成图

丢弃内核可执行文件中的元数据得到内核镜像,然后使用stat工具来比较内核可执行文件和内核镜像的大小：

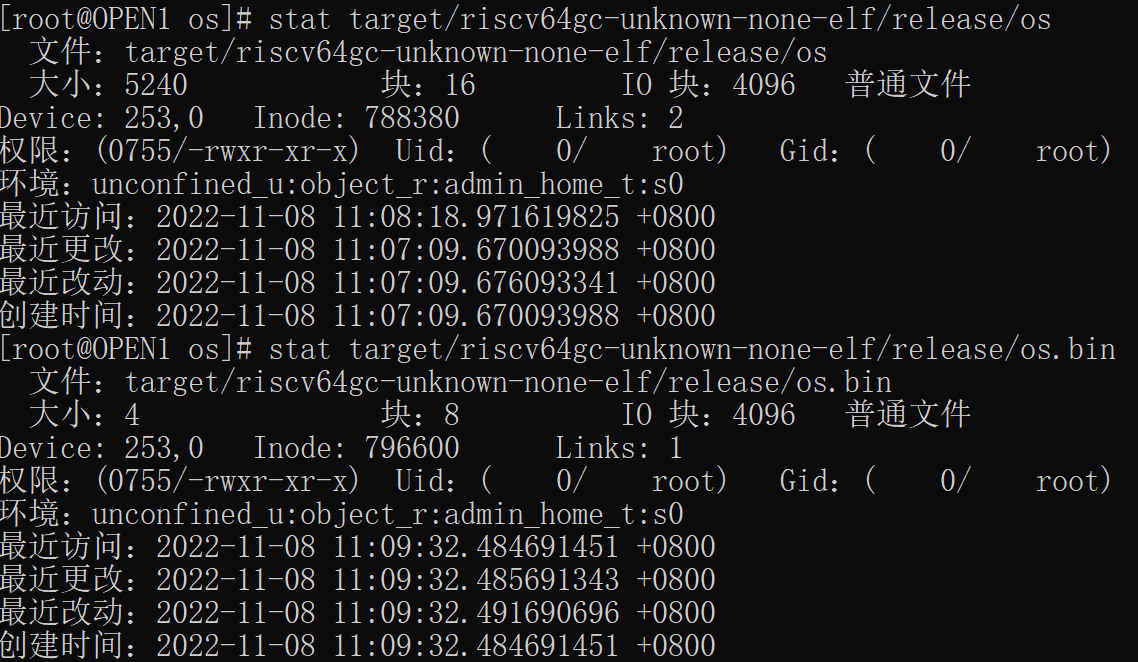


图4.5 可执行文件和内核镜像大小比对图

分配并使用启动栈，同时在main.rs中完成对.bss段的清零。

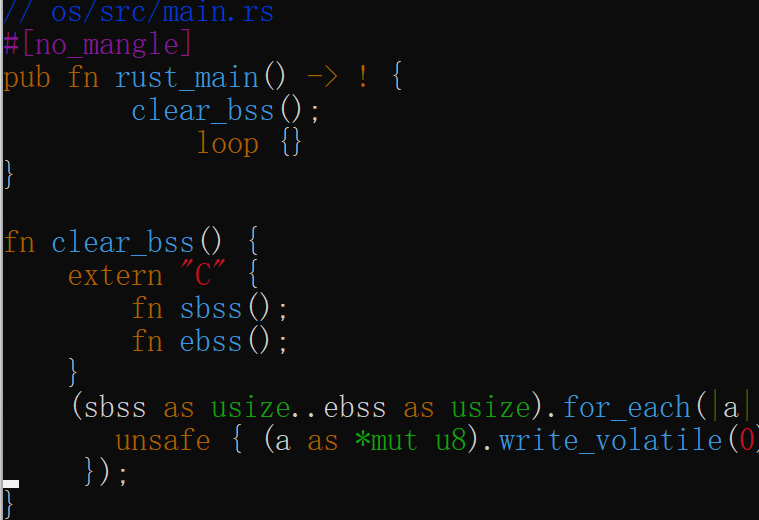


图4.6 分配启动栈、清零.bss段图

当内核发出请求时，计算机会转由 RustSBI 控制来响应内核的请求，待请求处理完毕后，计算机控制权会被交还给内核。我们将内核与 RustSBI 通信的相关功能实现在子模块。

在 sbi.rs 中定义 RustSBI 支持的服务类型常量。SBI\_CONSOLE\_PUTCHAR 可以用来在屏幕上输出一个字符，将这个功能封装成 console\_putchar 函数。将关服务SBI\_SHUTDOWN 封装成 shutdown 函数。尝试编写基于 console\_putchar的 println!宏。

处理致命错误：尝试打印更加详细的信息，包括panic所在的源文件和代码行数。尝试从传入的PanicInfo中解析这些信息，和panic的报错信息一起打印出来。在 main.rs 开头加上 #![feature(panic\_info\_message)] 当打印完毕之后，直接调用 shutdown 函数关机。

在rust\_main里完成clear\_bss();println!("Hello, world!");panic!("Shutdown machine!");

编译构建我们现在的代码，运行成果截图如下：

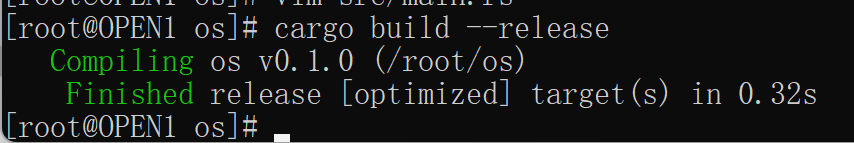


图4.7 成功运行截图

使用 Qemu 运行我们的内核，使用../labs/env/bootloader/rustsbi-qemu.bin代替默认的rustsbi-qemu.bin的存储路径。运行内核结果如图：

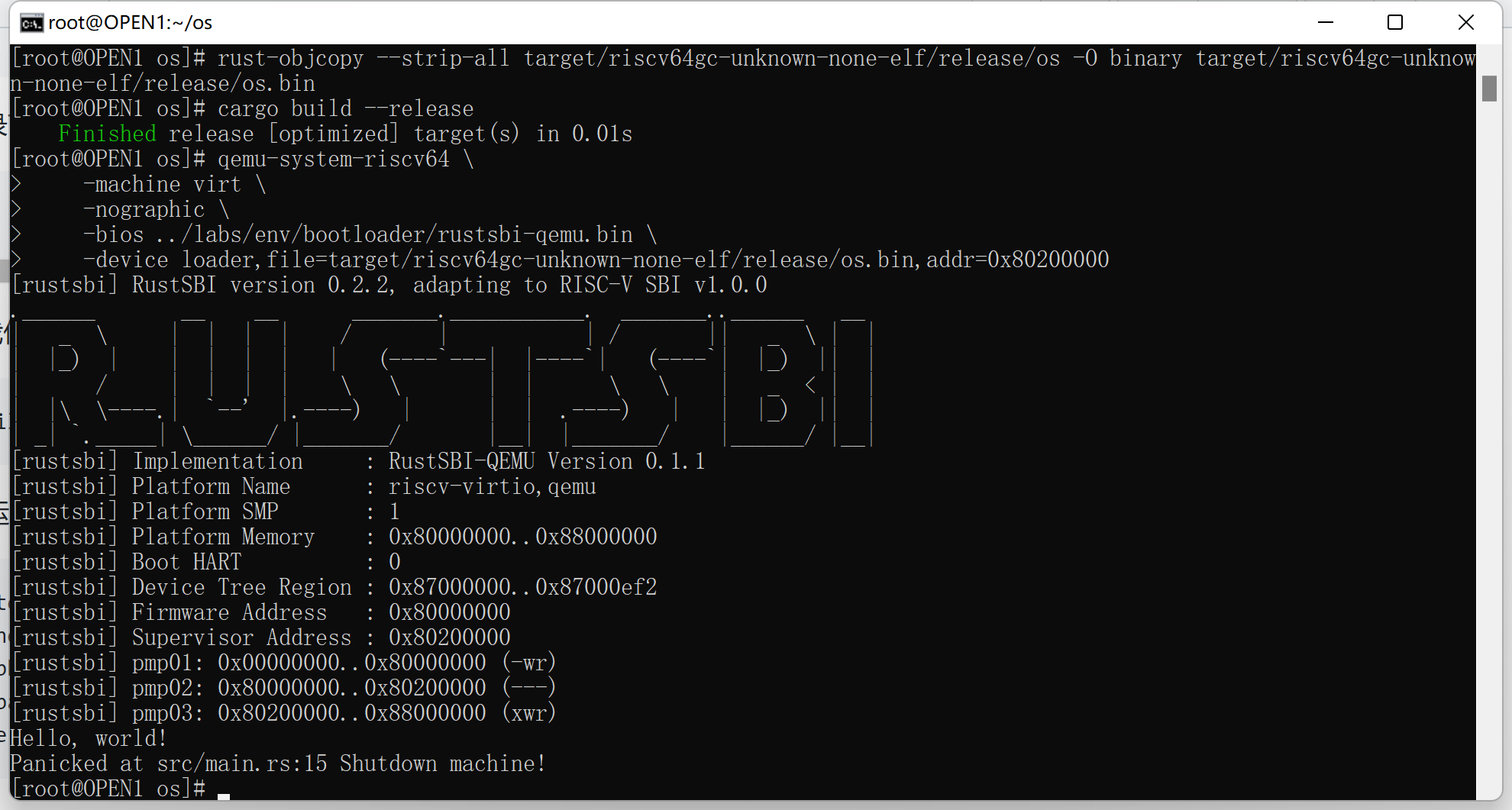


图4.8 Qemu 运行内核结果截图

# 5 实验总结

（说明：此部分请写出针对本次实验的总结。也可以写出对于相同功能的不同实现方案，或是书写针对现有实验的改进方案等独创的且与本次实验有关的内容。）

学习了计算机组成原理部分内容，计算机主要由处理器，物理内存和 I/O 外设三部分组成，CPU 可以通过物理地址来寻址，并逐字节地访问物理内存中保存的数据。

对 Qemu模拟器,进行了初步了解，通过 链接脚本 (Linker Script) 调整链接器的行为，修改 Cargo 的配置文件来使用我们自己的链接脚本，从而保证了链接器的内存布局符合我们的要求。

进一步如了解，内核、栈之间的关系，基于sbi服务完成输出和关机，使用自己编写的RustSBI通信功能，简便console\_putchar，自己编写基于console\_putchar的println宏！。处理致命错误也是编程的重要一环。进一步再编译构建之后通过Qemu运行内核。