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

|  |
| --- |
| **姓名:韩熔** |
| **学号:20301036** |

目录

[1 实验目的 3](#_Toc118133029)

[2 实验过程设计 3](#_Toc118133030)

[3 源代码及注释 3](#_Toc118133031)

[4 运行结果与分析 4](#_Toc118133032)

[5 实验总结 4](#_Toc118133033)

# 1 实验目的

将我们的内核正确对接到 Qemu 模拟器上，使得 Qemu 模拟器可以成功执行内核的第一条指令，并可以在屏幕上打印 Hello world! 和关机操作。

# 2 实验过程设计

1. 将我们的内核正确对接到Qemu 模拟器上，使得 Qemu 模拟器可以成功执行内核的第一条指令
2. 通过链接脚本调整链接器的行为，调整内核的内存布局
3. 手动加载内核可执行文件，丢弃内核可执行文件中的元数据得到内核镜像，手动帮助 Qemu 完成了可执行文件的加载
4. 为操作系统内核支持函数调用，分配并使用启动栈。通过分配栈空间并正确设置栈指针在内核中能使函数调用并成功将控制权转交给 Rust代码，从此可以利用Rust 语言来编写内核的各项功能
5. 通过使用 RustSBI 提供的服务，实现格式化输出，并实现错误处理
6. 编译构建代码并去除内核可执行文件中的元数据，再使用 Qemu 运行我们的内核

# 3 源代码及注释

表 3-1 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | main.rs |
| **主要功能:** | 入口 |
| **源代码:**  **#![no\_main]**  **#![no\_std]**  **#![feature(panic\_info\_message)]**  **#[macro\_use]**  **mod console;**  **mod lang\_items;**  **mod sbi;**  **//fn main() {**  **// println!("Hello, world!");**  **//}**  **use core::arch::global\_asm;**  **global\_asm!(include\_str!("entry.asm"));**  **#[no\_mangle]**  **pub fn rust\_main() -> ! {**  **clear\_bss();**  **//loop {}**  **println!("Hello, world!");**  **panic!("Shutdown machine!");**  **}**  **fn clear\_bss() {**  **extern "C" {**  **fn sbss();**  **fn ebss();**  **}**  **(sbss as usize..ebss as usize).for\_each(|a| {**  **unsafe { (a as \*mut u8).write\_volatile(0) }**  **});**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | lang\_items.rs |
| **主要功能:** | 在 panic 函数中打印错误信息并关机 |
| **源代码:**  **// os/src/lang\_items.rs**  **use core::panic::PanicInfo;**  **use crate::sbi::shutdown;**  **#[panic\_handler]**  **fn panic(info: &PanicInfo) -> ! {**  **if let Some(location) = info.location() {**  **println!(**  **"Panicked at {}:{} {}",**  **location.file(),**  **location.line(),**  **info.message().unwrap()**  **);**  **} else {**  **println!("Panicked: {}", info.message().unwrap());**  **}**  **shutdown()**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | entry.asm |
| **主要功能:** | 在 Qemu 上执行了内核的第一条指令 |
| **源代码:**  **# os/src/entry.asm**  **.section .text.entry**  **.globl \_start**  **\_start:**  **la sp, boot\_stack\_top**  **call rust\_main**  **.section .bss.stack**  **.globl boot\_stack**  **boot\_stack:**  **.space 4096 \* 16**  **.globl boot\_stack\_top**  **boot\_stack\_top:** | |

|  |  |
| --- | --- |
| **文件名:** | Linker.ld |
| **主要功能:** | 实现与 Qemu 的正确对接 |
| **源代码:**  **OUTPUT\_ARCH(riscv)**  **ENTRY(\_start)**  **BASE\_ADDRESS = 0x80200000;**  **SECTIONS**  **{**  **. = BASE\_ADDRESS;**  **skernel = .;**  **stext = .;**  **.text : {**  **\*(.text.entry)**  **\*(.text .text.\*)**  **}**  **. = ALIGN(4K);**  **etext = .;**  **srodata = .;**  **.rodata : {**  **\*(.rodata .rodata.\*)**  **\*(.srodata .srodata.\*)**  **}**  **. = ALIGN(4K);**  **erodata = .;**  **sdata = .;**  **.data : {**  **\*(.data .data.\*)**  **\*(.sdata .sdata.\*)**  **}**  **. = ALIGN(4K);**  **edata = .;**  **.bss : {**  **\*(.bss.stack)**  **sbss = .;**  **\*(.bss .bss.\*)**  **\*(.sbss .sbss.\*)**  **}**  **. = ALIGN(4K);**  **ebss = .;**  **ekernel = .;**  **/DISCARD/ : {**  **\*(.eh\_frame)**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | sbi.rs |
| **主要功能:** | 调用” RustSBI 的服务，封装服务 SBI\_CONSOLE\_PUTCHAR以及关机服务 SBI\_SHUTDOWN |
| **源代码:**  **#![allow(unused)]**  **const SBI\_SET\_TIMER: usize = 0;**  **const SBI\_CONSOLE\_PUTCHAR: usize = 1;**  **const SBI\_CONSOLE\_GETCHAR: usize = 2;**  **const SBI\_CLEAR\_IPI: usize = 3;**  **const SBI\_SEND\_IPI: usize = 4;**  **const SBI\_REMOTE\_FENCE\_I: usize = 5;**  **const SBI\_REMOTE\_SFENCE\_VMA: usize = 6;**  **const SBI\_REMOTE\_SFENCE\_VMA\_ASID: usize = 7;**  **const SBI\_SHUTDOWN: usize = 8;**  **use core::arch::asm;**  **#[inline(always)]**  **fn sbi\_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {**  **let mut ret;**  **unsafe {**  **asm!(**  **"ecall",**  **inlateout("x10") arg0 => ret,**  **in("x11") arg1,**  **in("x12") arg2,**  **in("x17") which,**  **);**  **}**  **ret**  **}**  **// os/src/sbi.rs**  **pub fn console\_putchar(c: usize) {**  **sbi\_call(SBI\_CONSOLE\_PUTCHAR, c, 0, 0);**  **}**  **// os/src/sbi.rs**  **pub fn shutdown() -> ! {**  **sbi\_call(SBI\_SHUTDOWN, 0, 0, 0);**  **panic!("It should shutdown!");**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | console.rs |
| **主要功能:** | 实现格式化输出 |
| **源代码:**  **use crate::sbi::console\_putchar;**  **use core::fmt::{self, Write};**  **struct Stdout;**  **impl Write for Stdout {**  **fn write\_str(&mut self, s: &str) -> fmt::Result {**  **for c in s.chars() {**  **console\_putchar(c as usize);**  **}**  **Ok(())**  **}**  **}**  **pub fn print(args: fmt::Arguments) {**  **Stdout.write\_fmt(args).unwrap();**  **}**  **#[macro\_export]**  **macro\_rules! print {**  **($fmt: literal $(, $($arg: tt)+)?) => {**  **$crate::console::print(format\_args!($fmt $(, $($arg)+)?));**  **}**  **}**  **#[macro\_export]**  **macro\_rules! println {**  **($fmt: literal $(, $($arg: tt)+)?) => {**  **$crate::console::print(format\_args!(concat!($fmt, "\n") $(, $($arg)+)?));**  **}**  **}** | |

表 3-2 关键命令解释

|  |  |  |
| --- | --- | --- |
| **序号** | **命令名** | **主要作用** |
| **1** | **qemu-system-riscv64 \**  **-machine virt \**  **-nographic \**  **-bios bootloader/rustsbi-qemu.bin \**  **-device loader,file=os.bin,addr=0x80200000** | 启动 Qemu 并运行我们的内核 |
| **2** | $ cargo build --release | 生成内核可执行的文件 |
| **3** | **$ file target/riscv64gc-unknown-none-elf/release/os** | 通过 file 工具查看属性 |
| **4** | **$ rust-objcopy --strip-all target/riscv64gc-unknown-none-elf/release/os -O binary target/riscv64gc-unknown-none-elf/release/os.bin** | 丢弃内核可执行文件中的元数据得到内核镜像 |

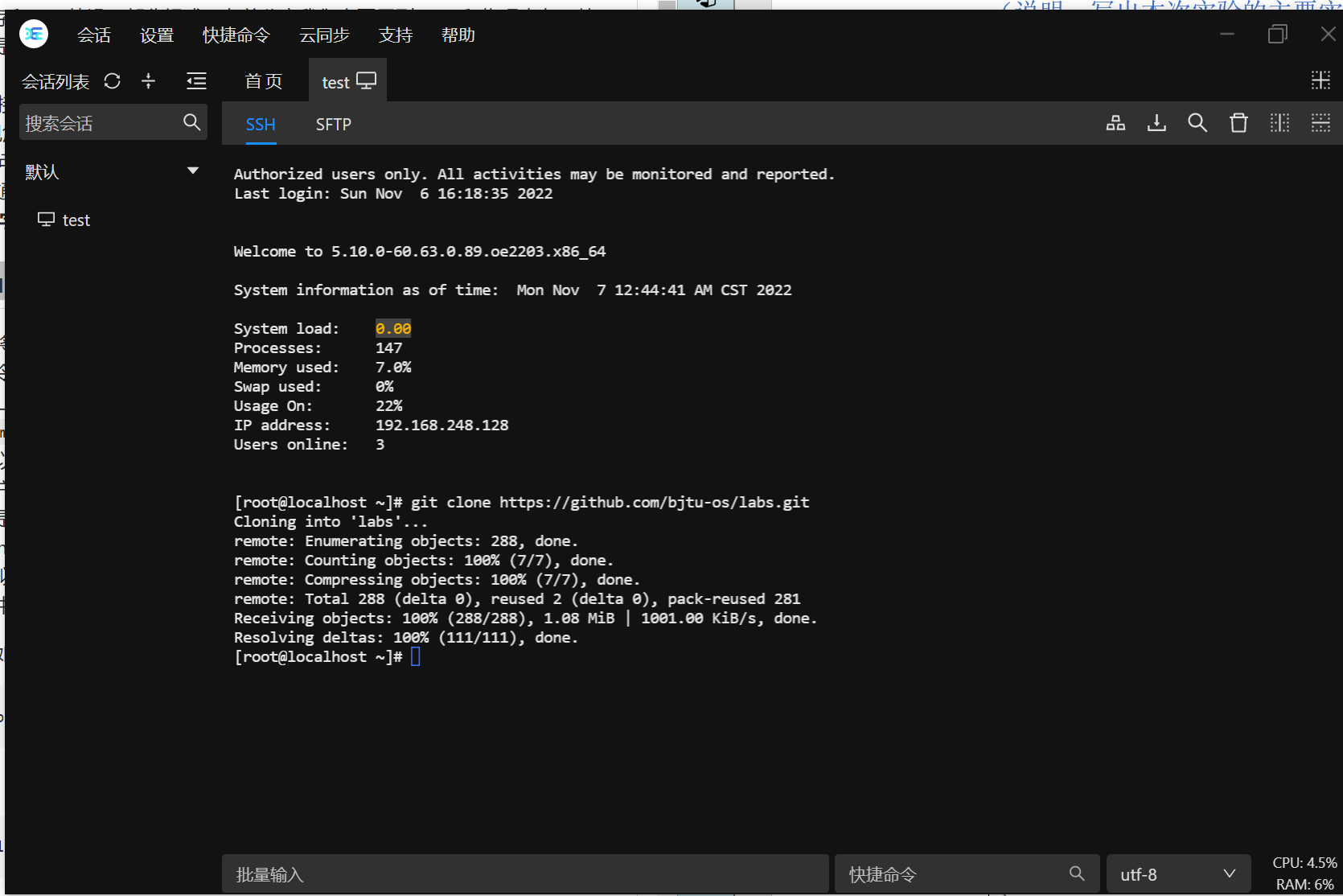
# 4 运行结果与分析

1. 编写操作系统内核前置知识

Qemu 模拟器

（1）从 Github 下载课程代码

git clone https://github.com/bjtu-os/labs.git



（2）进入项目

cd labs/code/lab2

（3）使用如下命令来启动 Qemu 并运行我们的内核

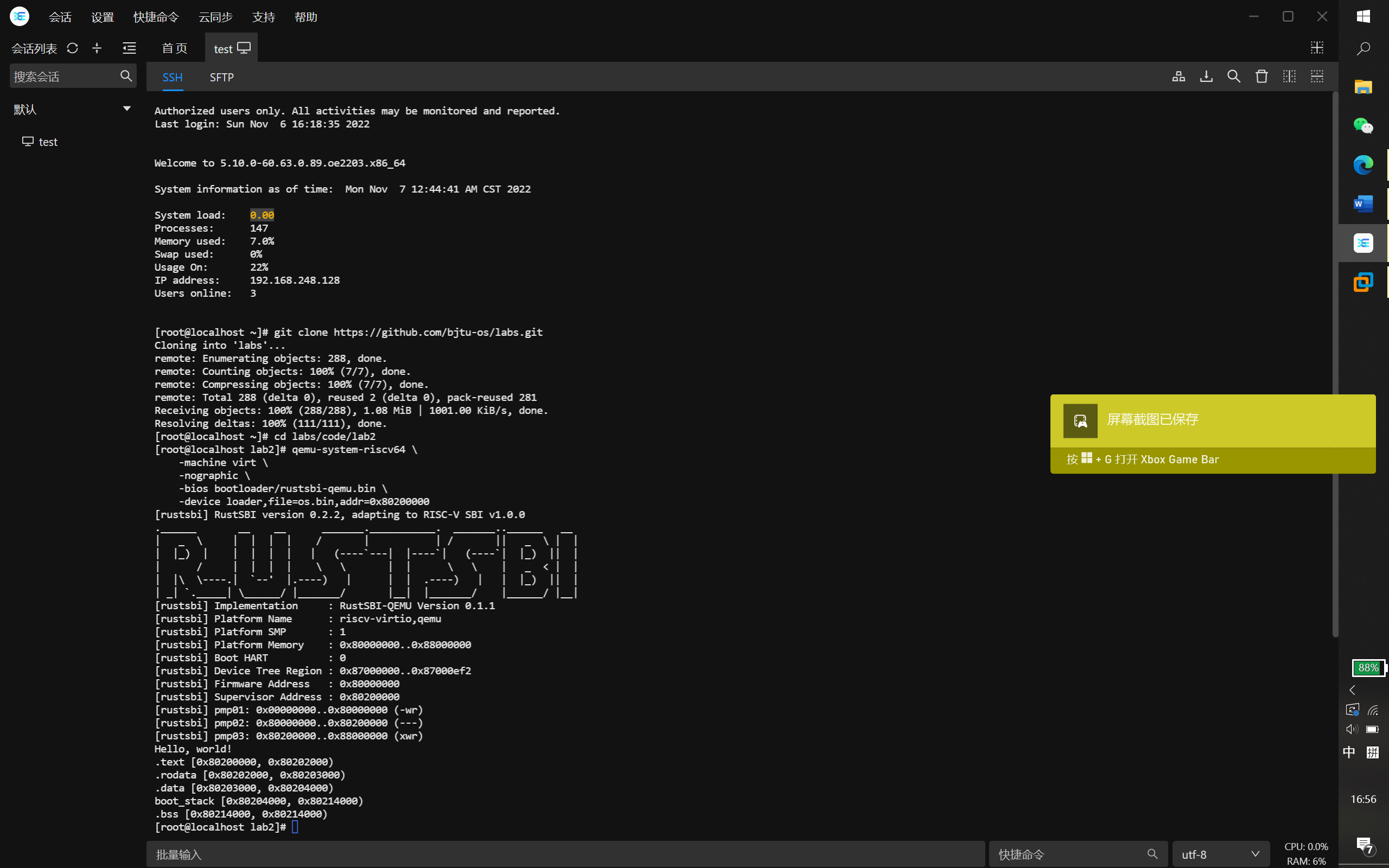
qemu-system-riscv64 \

-machine virt \

-nographic \

-bios bootloader/rustsbi-qemu.bin \

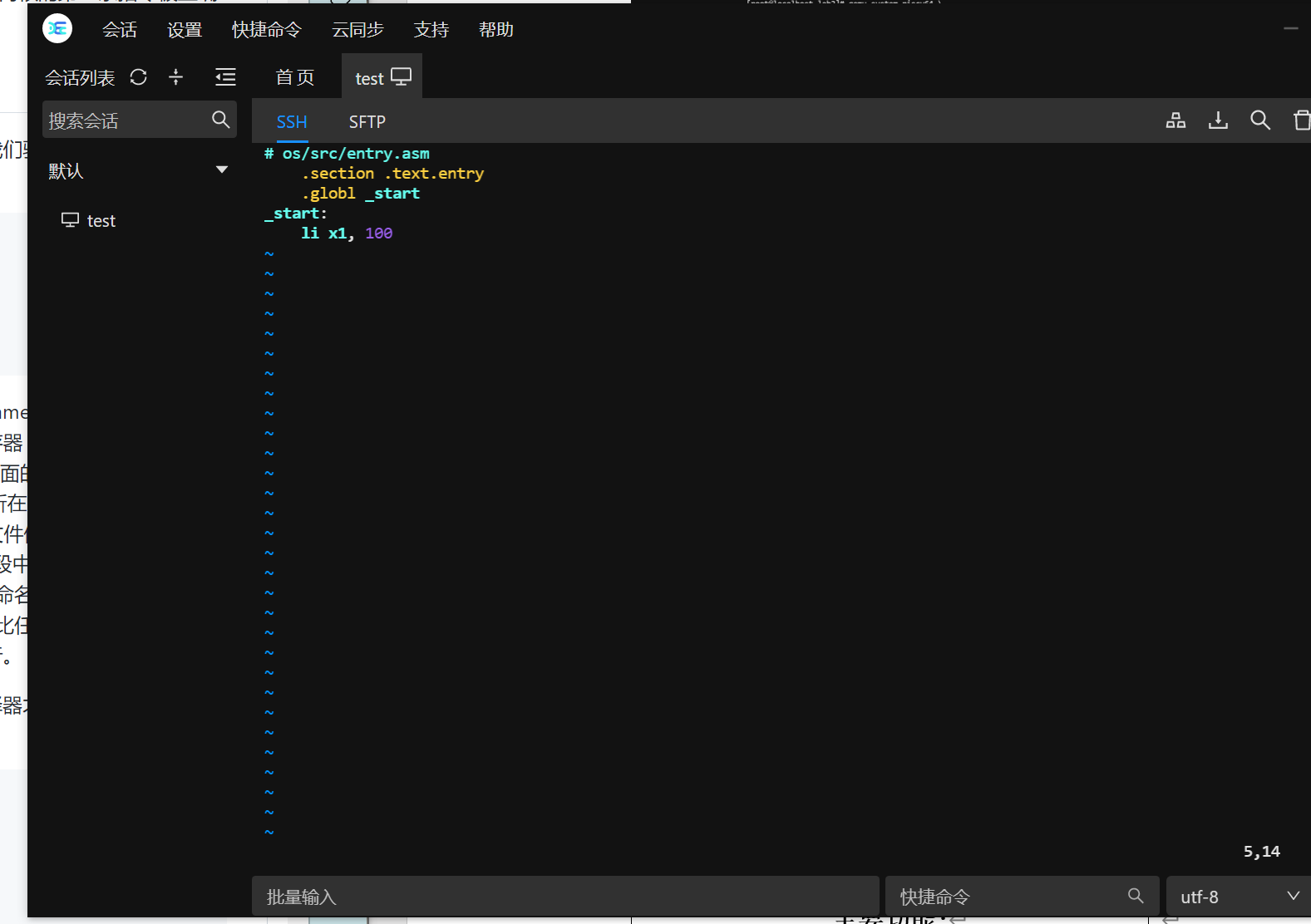
-device loader,file=os.bin,addr=0x80200000



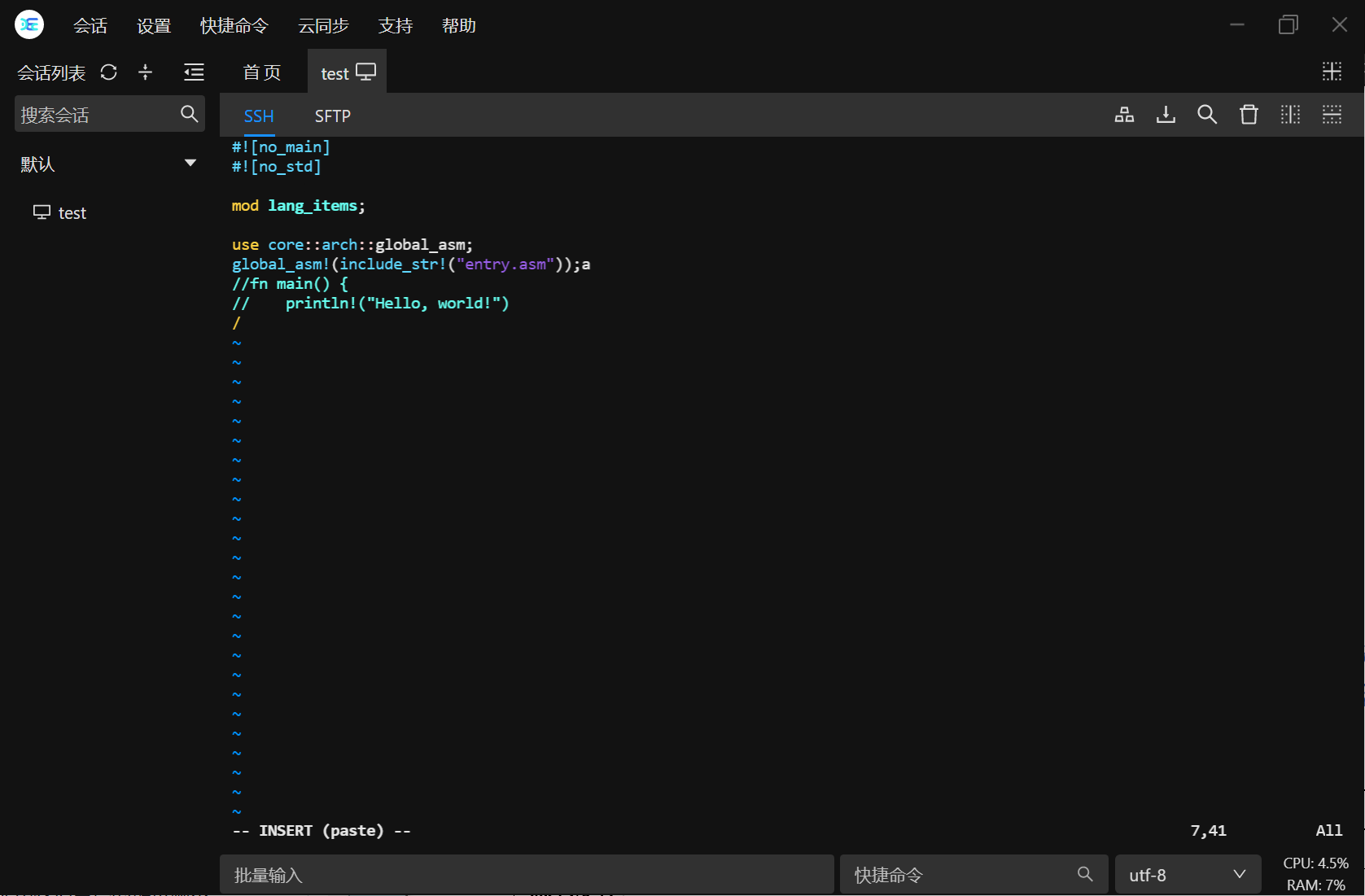
二．编写操作系统内核

编写内核第一条指令

1. 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。



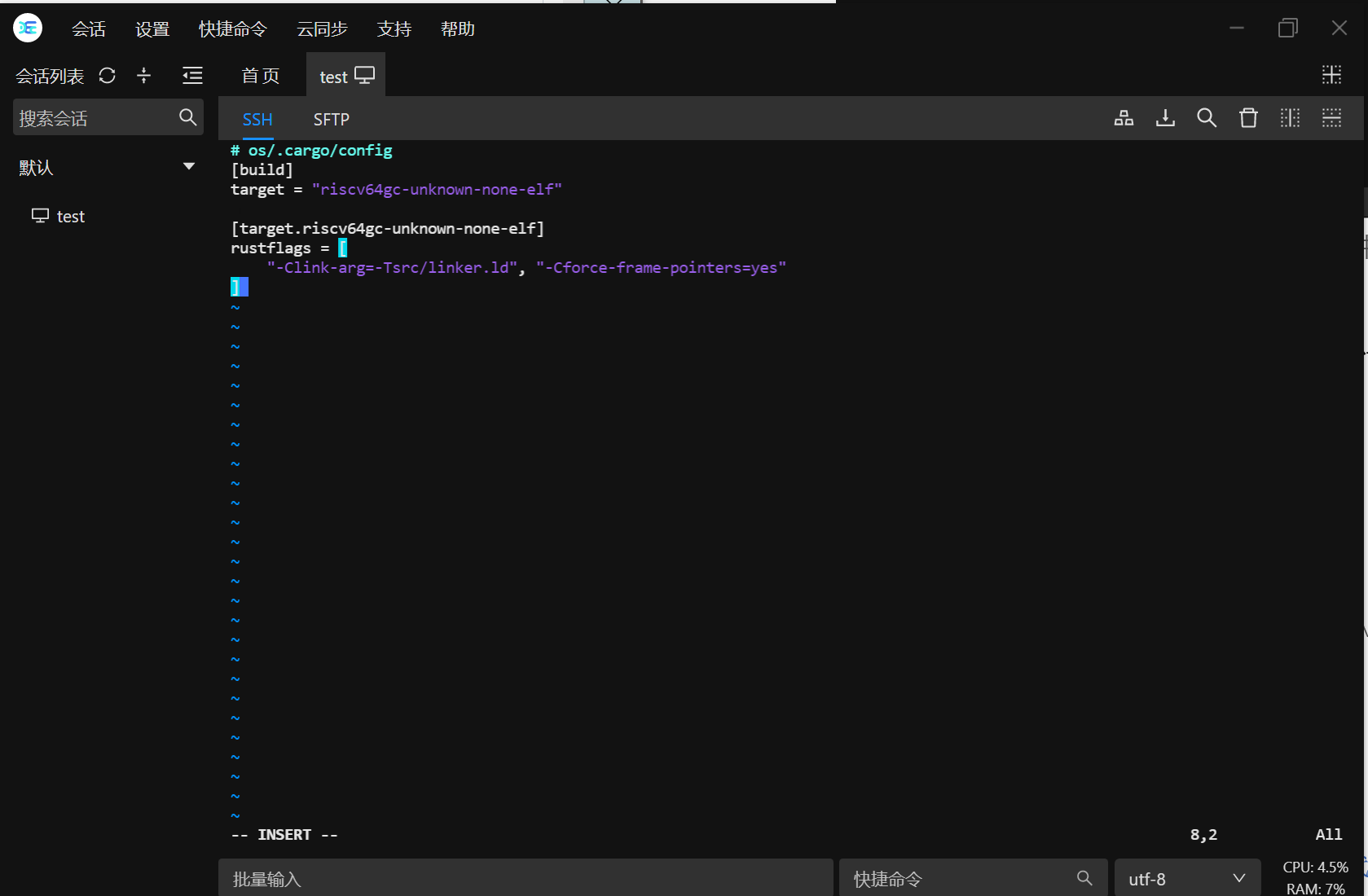
1. 在 main.rs 中嵌入这段汇编代码



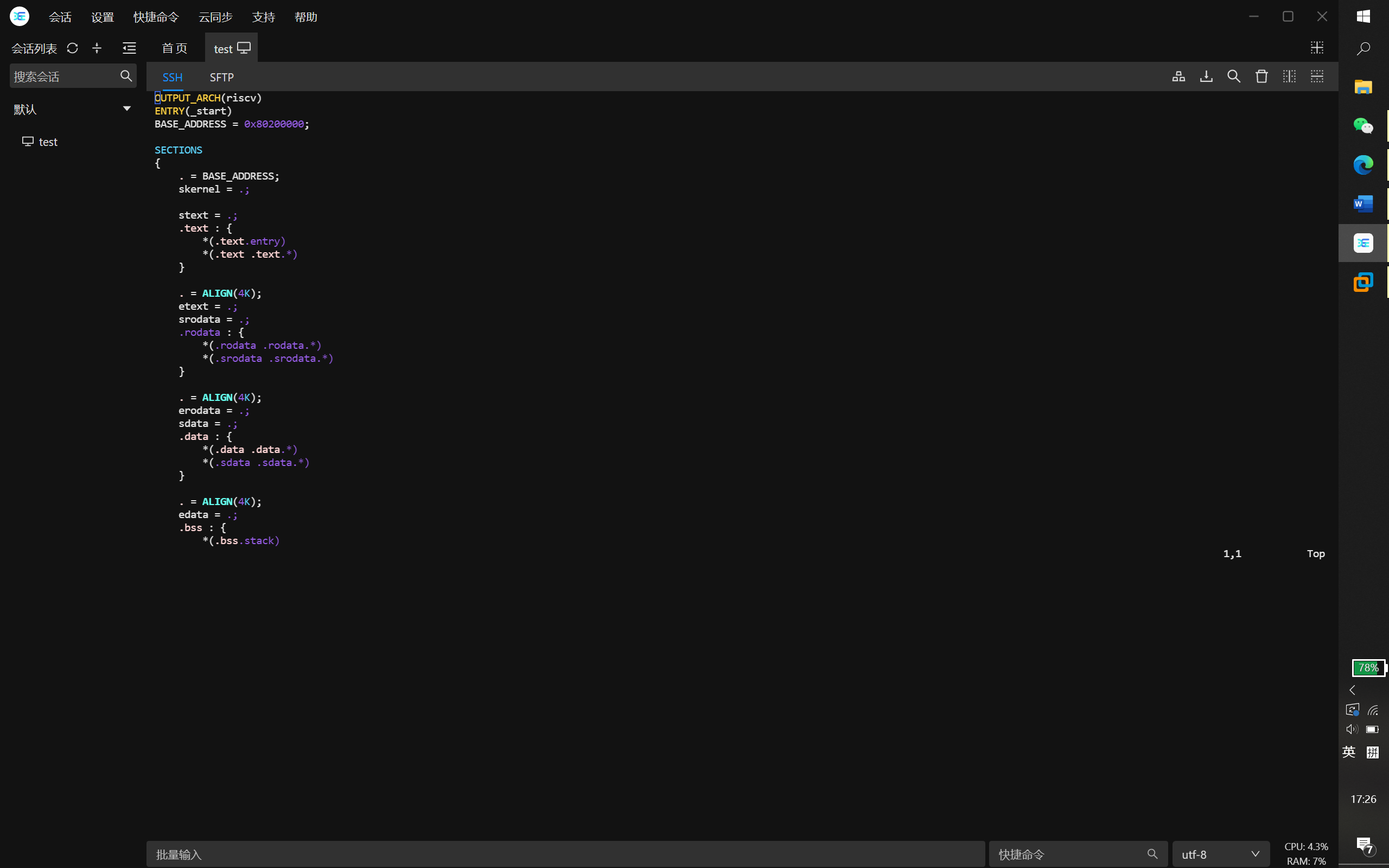
global\_asm! 宏已被加入到 Rust 核心库 core 中。

调整内核的内存布局

1. 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

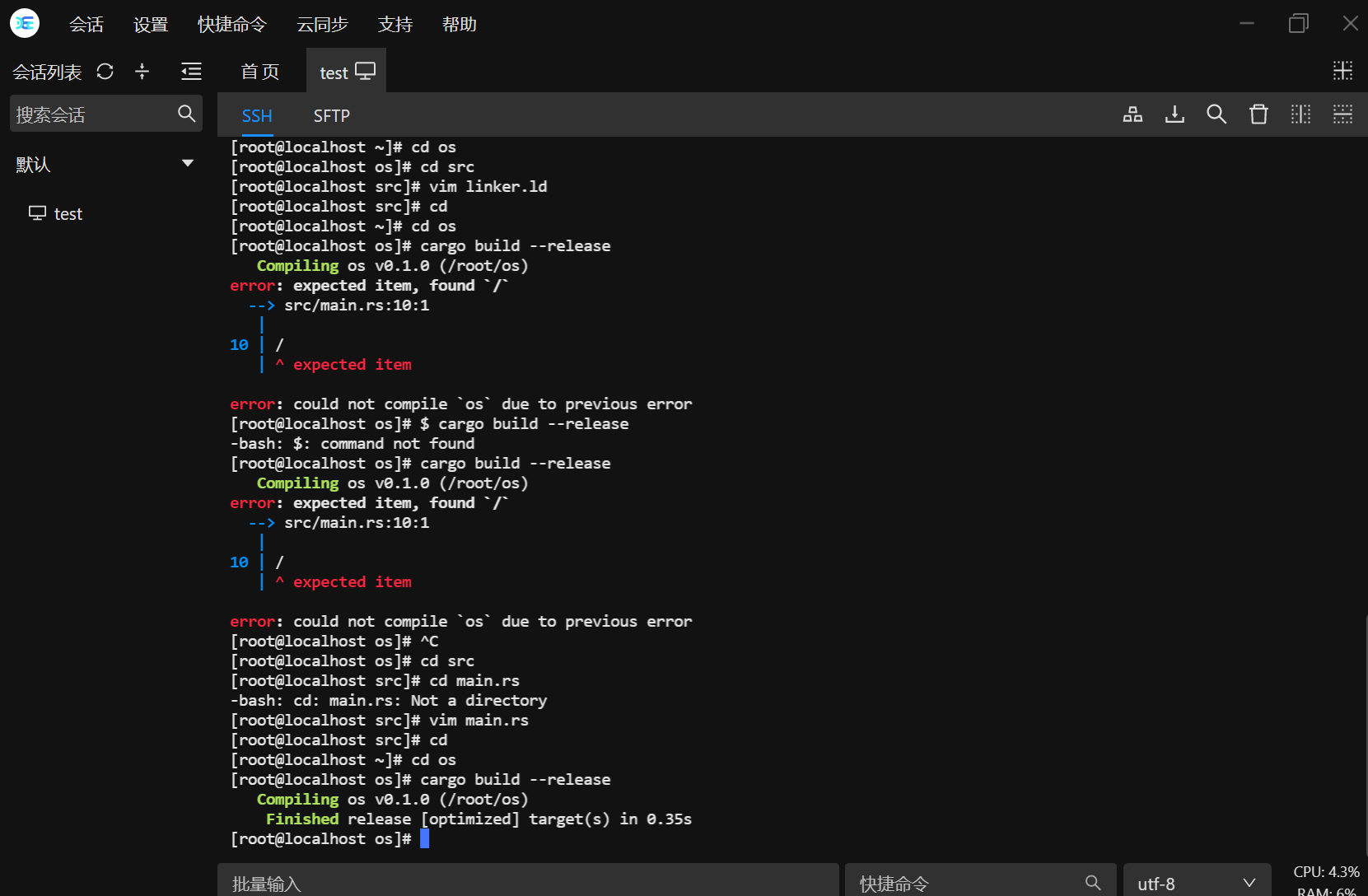


链接脚本 os/src/linker.ld 如下



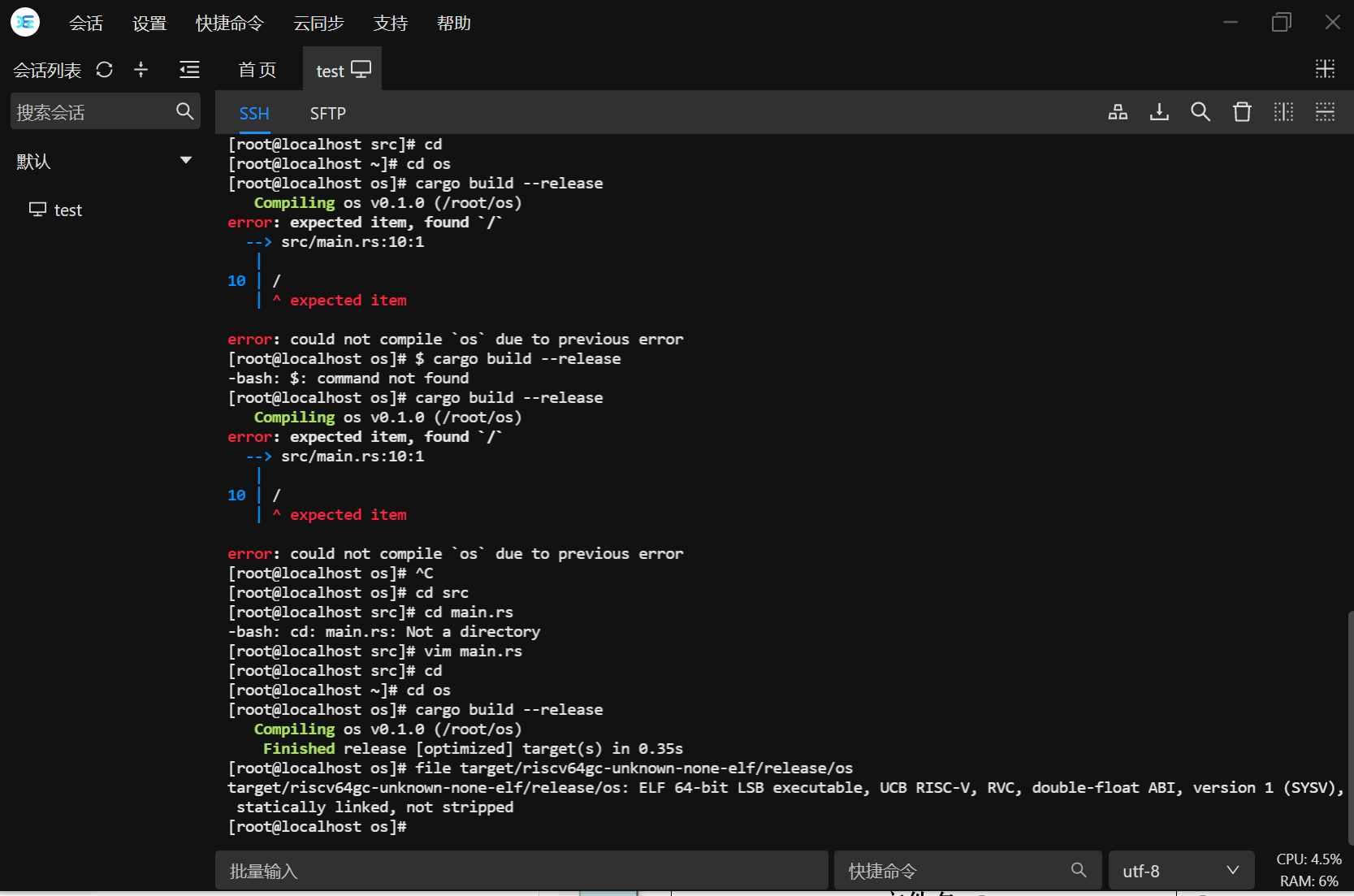
1. 生成内核可执行文件，切换到 os 目录下并进行以下操作：

$ cargo build --release



（3）通过 file 工具查看它的属性

$ file target/riscv64gc-unknown-none-elf/release/os

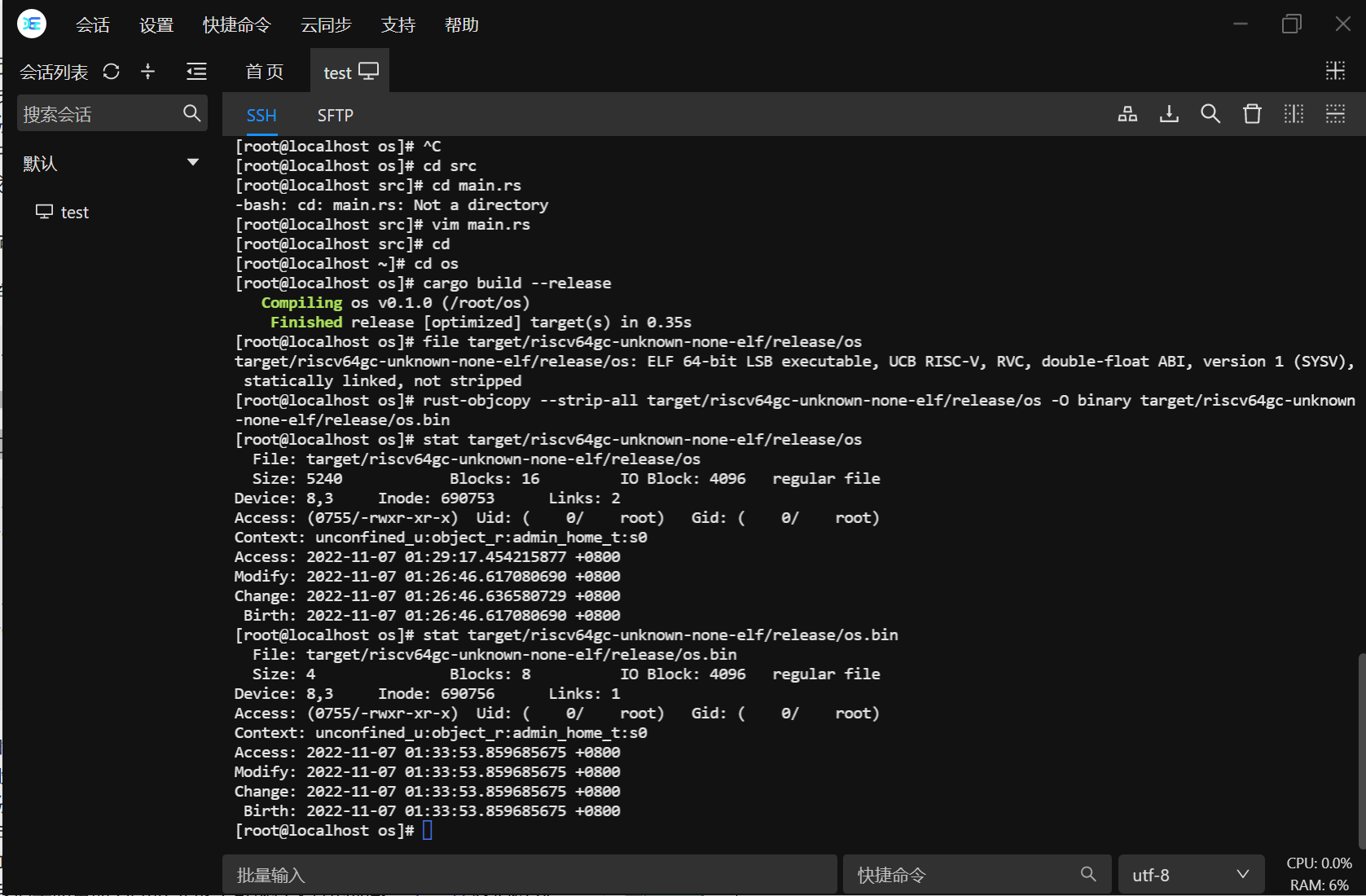


手动加载内核可执行文件

1. 丢弃内核可执行文件中的元数据得到内核镜像

$ rust-objcopy --strip-all target/riscv64gc-unknown-none-elf/release/os -O binary target/riscv64gc-unknown-none-elf/release/os.bin

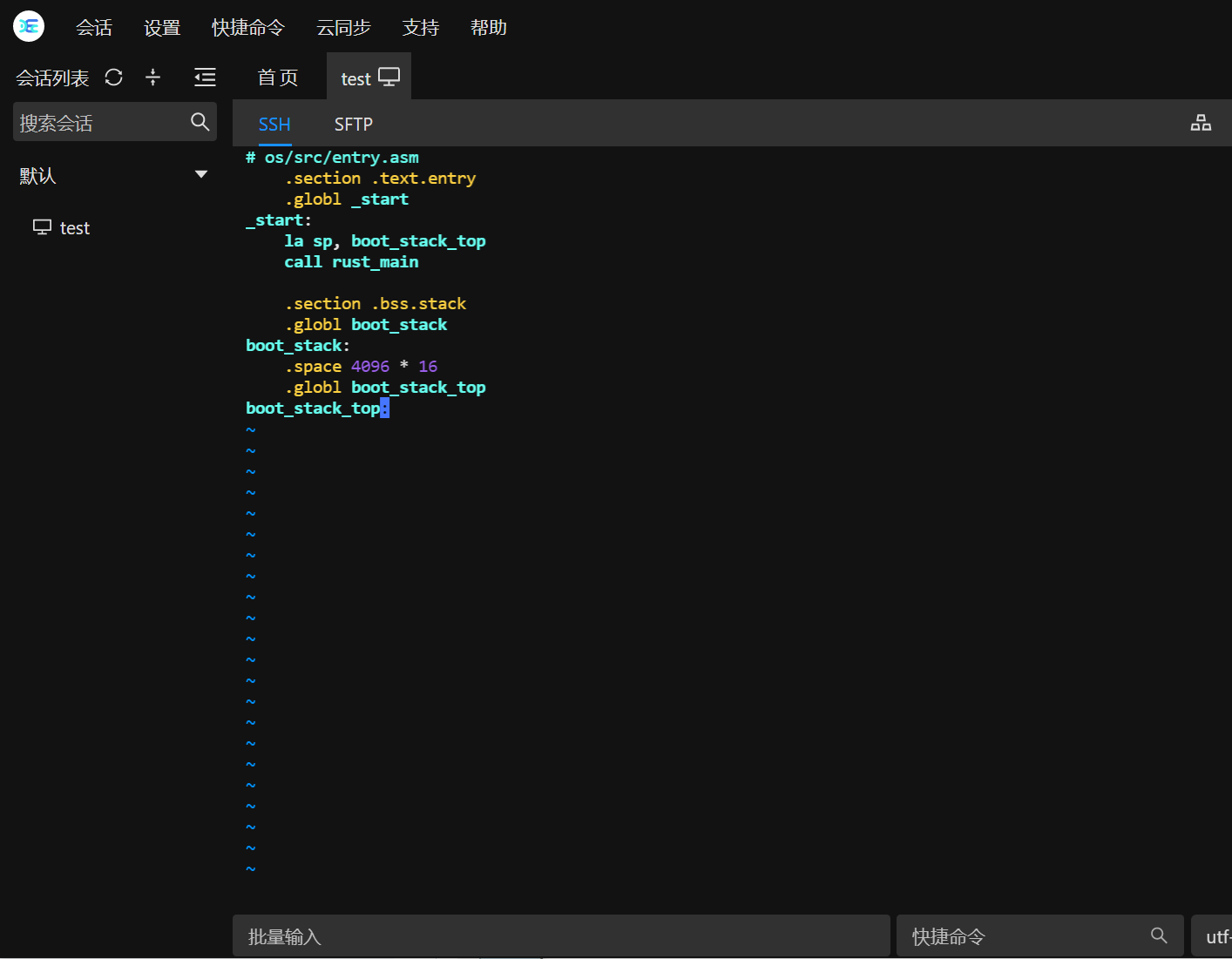
1. 使用 stat 工具来比较内核可执行文件和内核镜像的大小



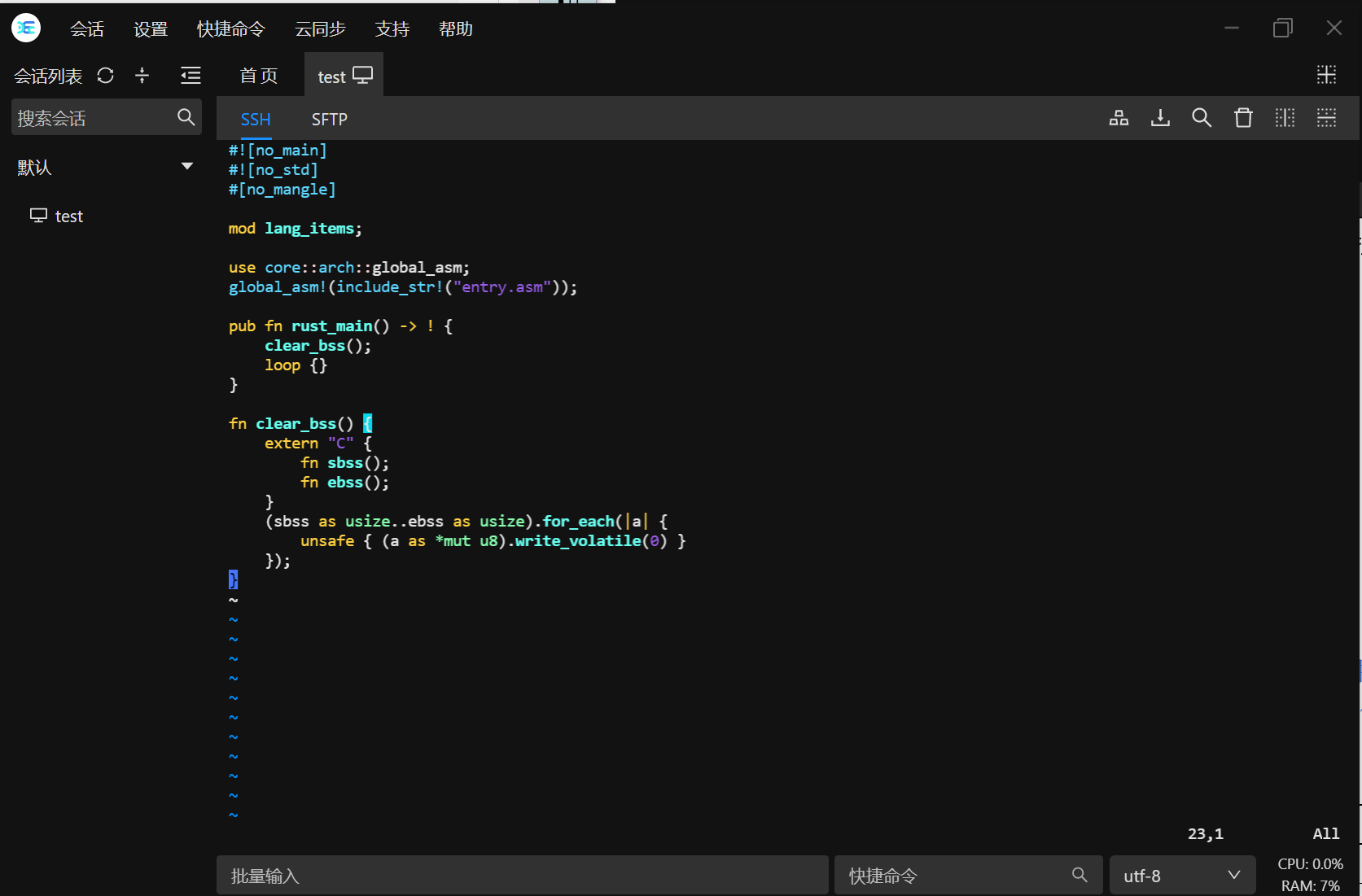
三．为操作系统内核支持函数调用

分配并使用启动栈

（1）在 entry.asm 中分配启动栈空间，并在控制权被转交给 Rust 入口之前将栈指针 sp 设置为栈顶的位置。



（2）通过伪指令 call 调用 Rust 编写的内核入口点 rust\_main 将控制权转交给 Rust 代码，该入口点在 main.rs 中实现，顺便完成对 .bss 段的清零



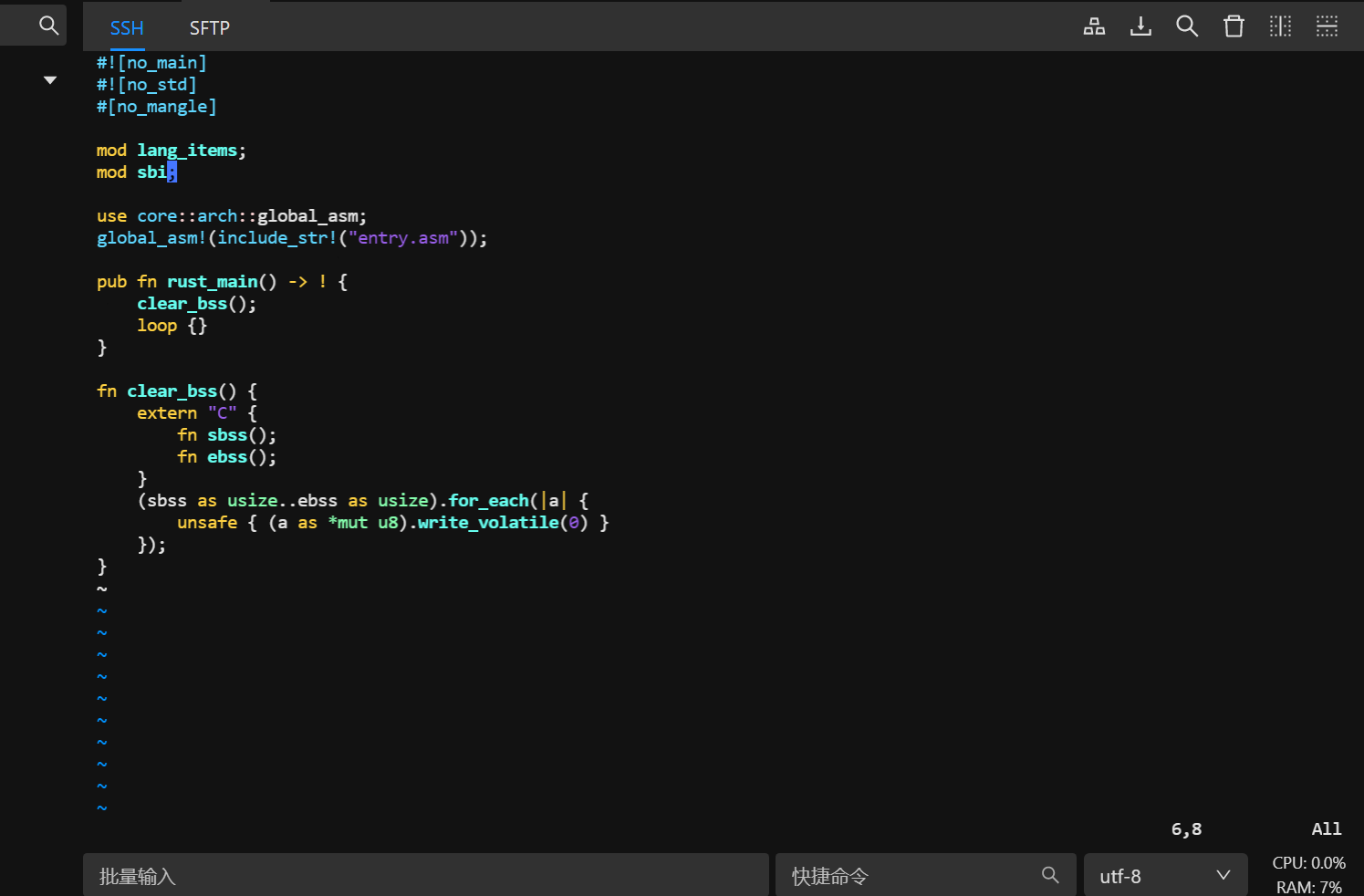
四．基于 SBI 服务完成输出和关机

使用 RustSBI 提供的服务

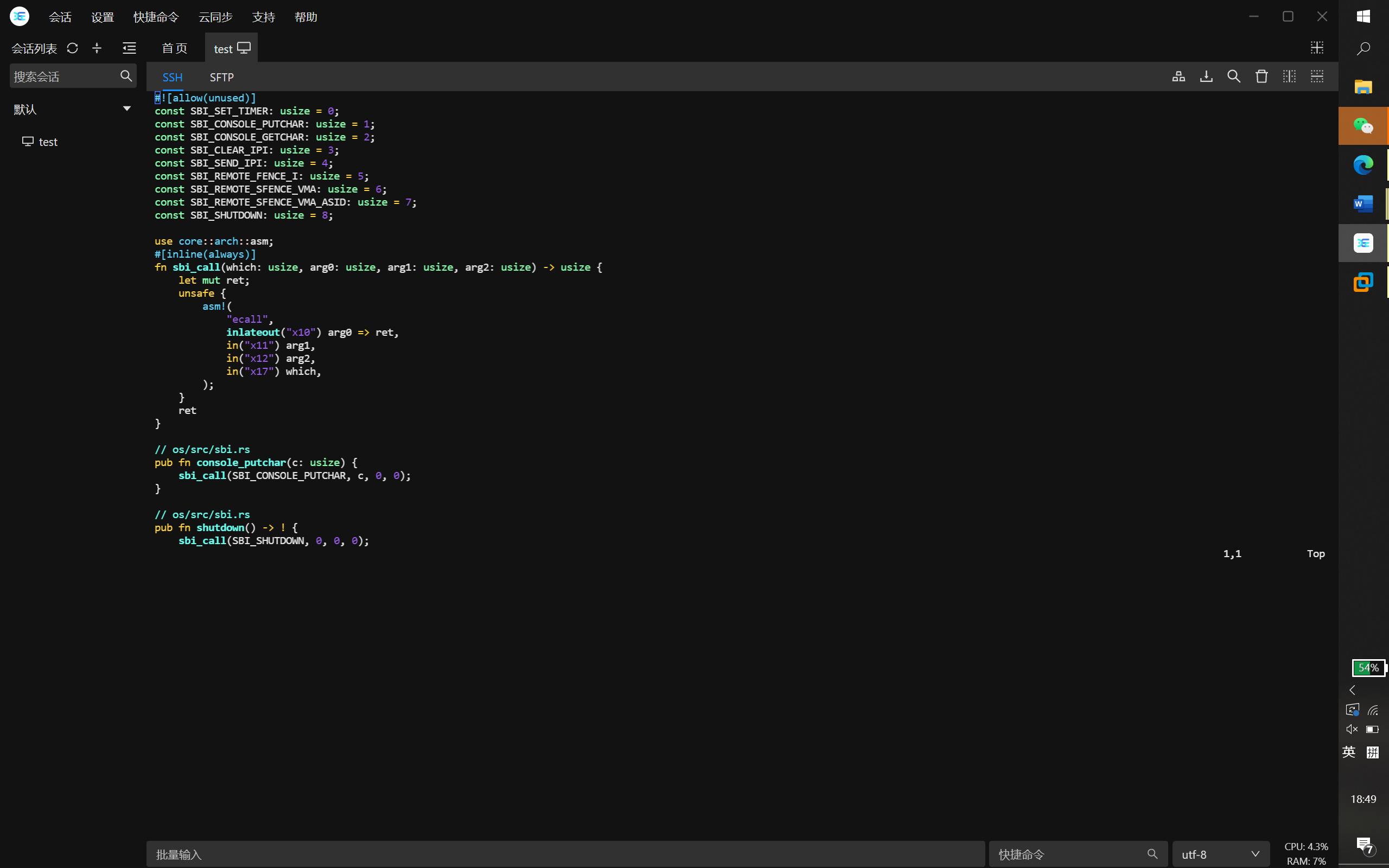
（1）在 main.rs 中加入 mod sbi 将该子模块加入我们的项目

// os/src/main.rs

mod sbi;

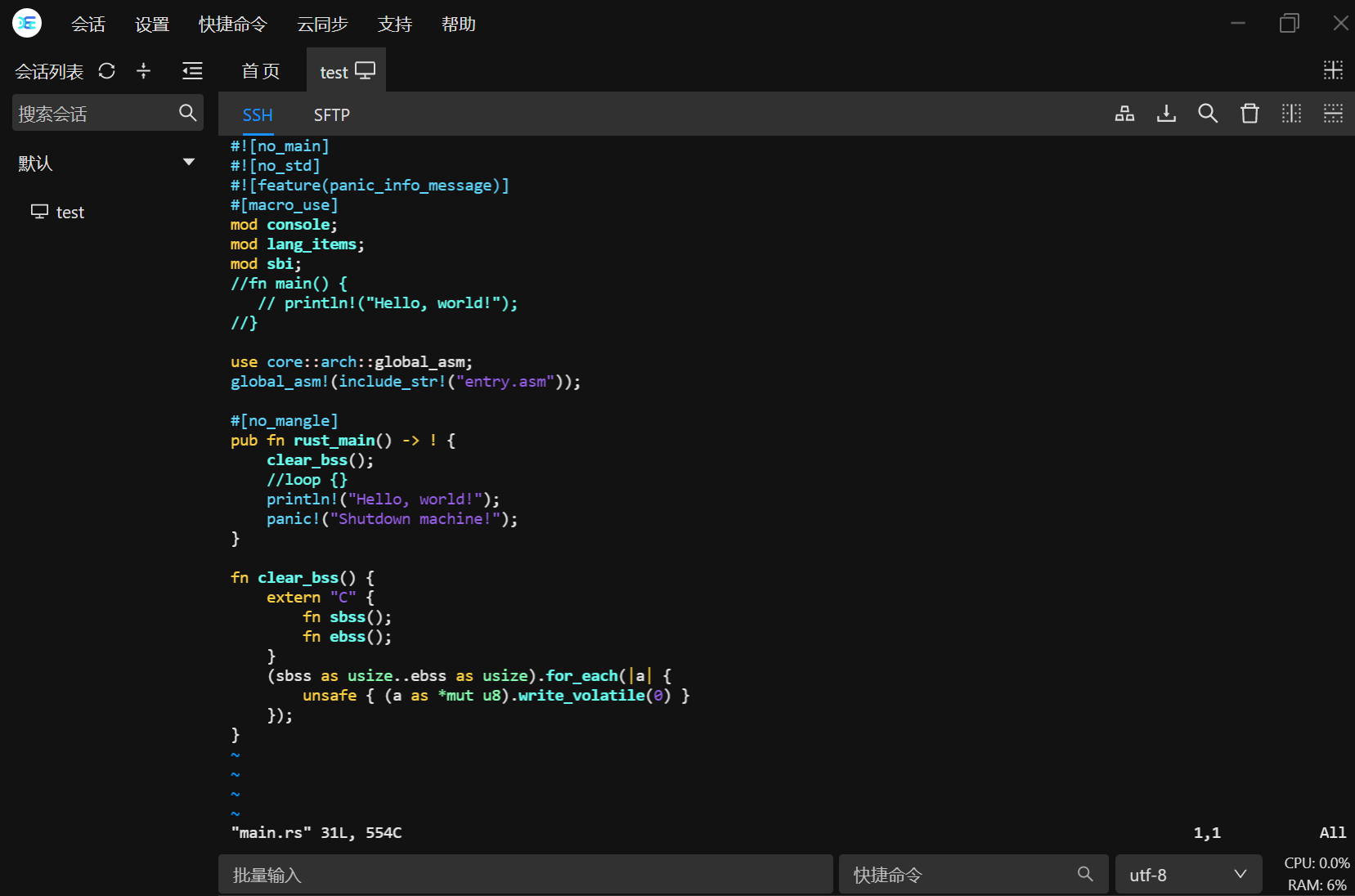
****

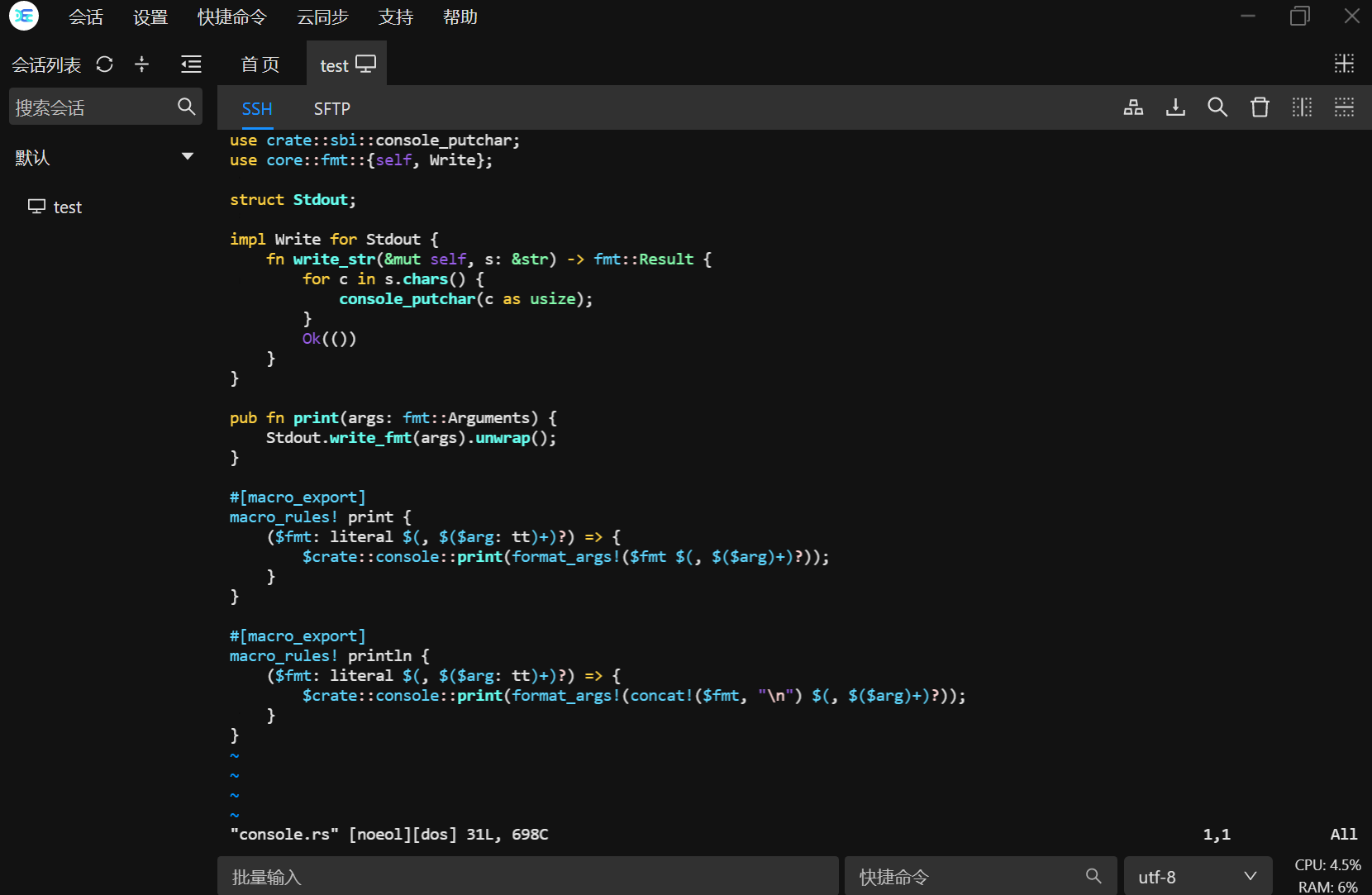
（2）将内核与 RustSBI 通信的相关功能实现在子模块 sbi 中



实现格式化输出

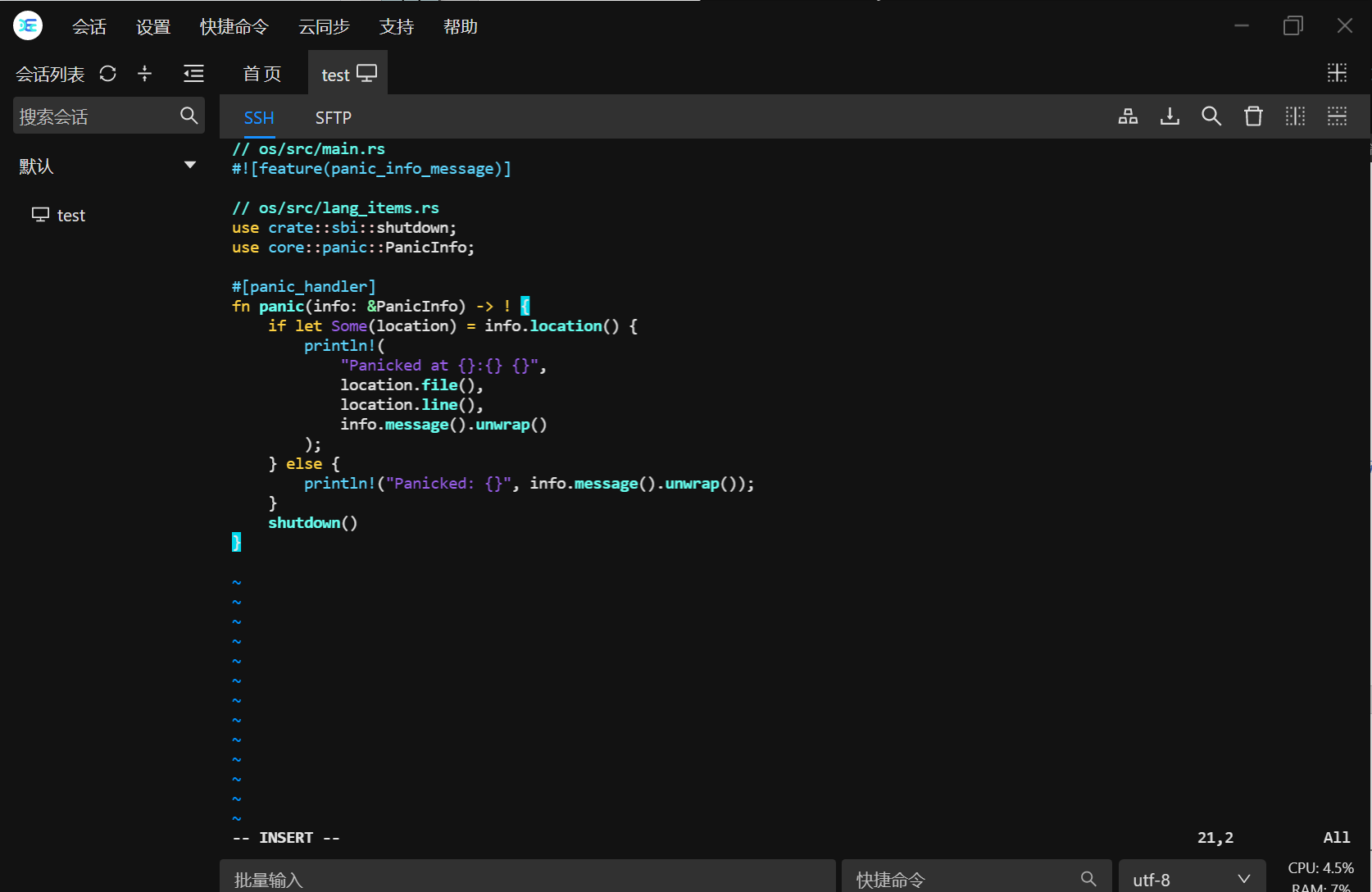
1. 尝试自己编写基于 console\_putchar 的 println! 宏。



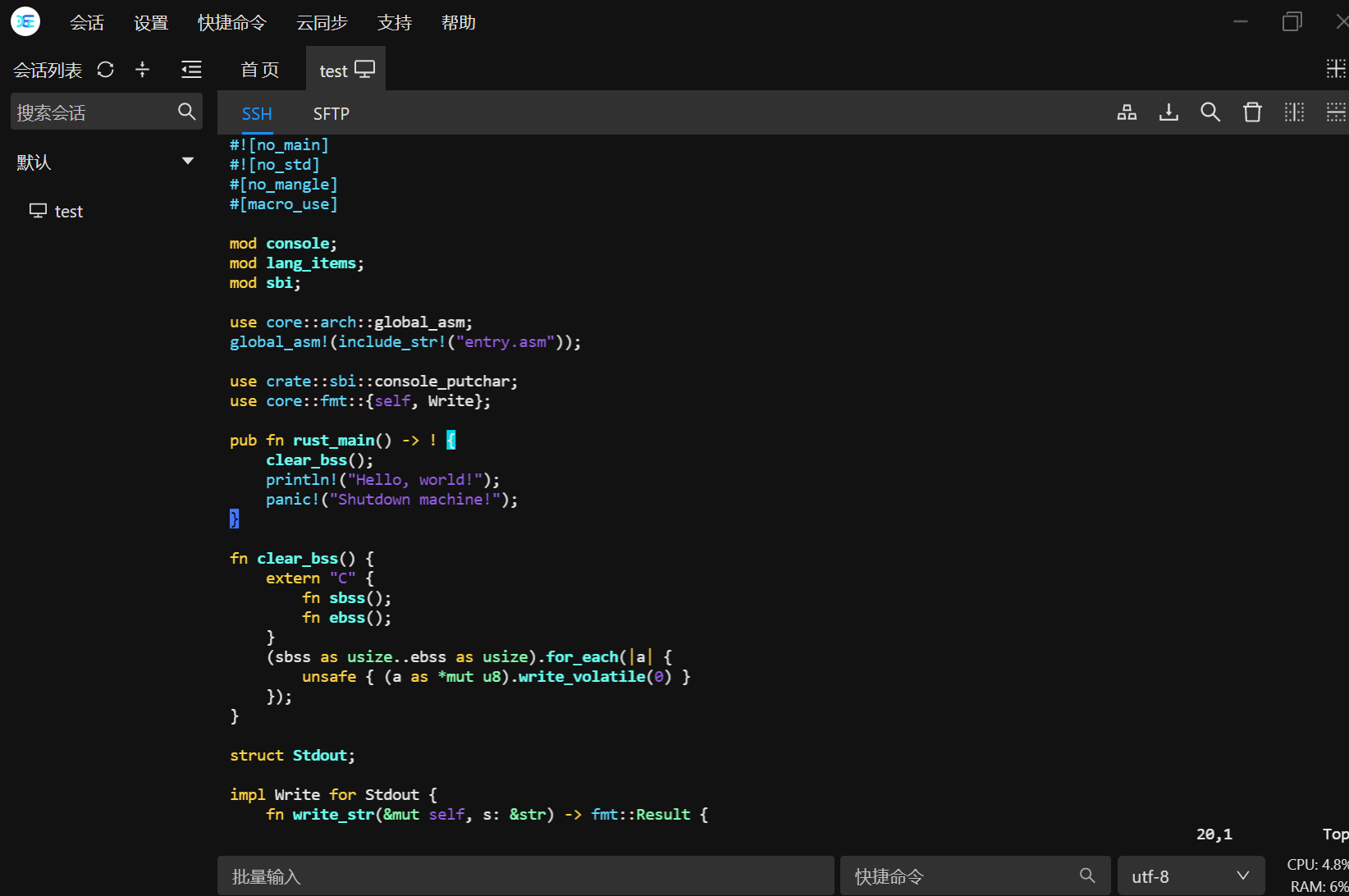


处理致命错误

1. 在 panic 函数中打印错误信息并关机：

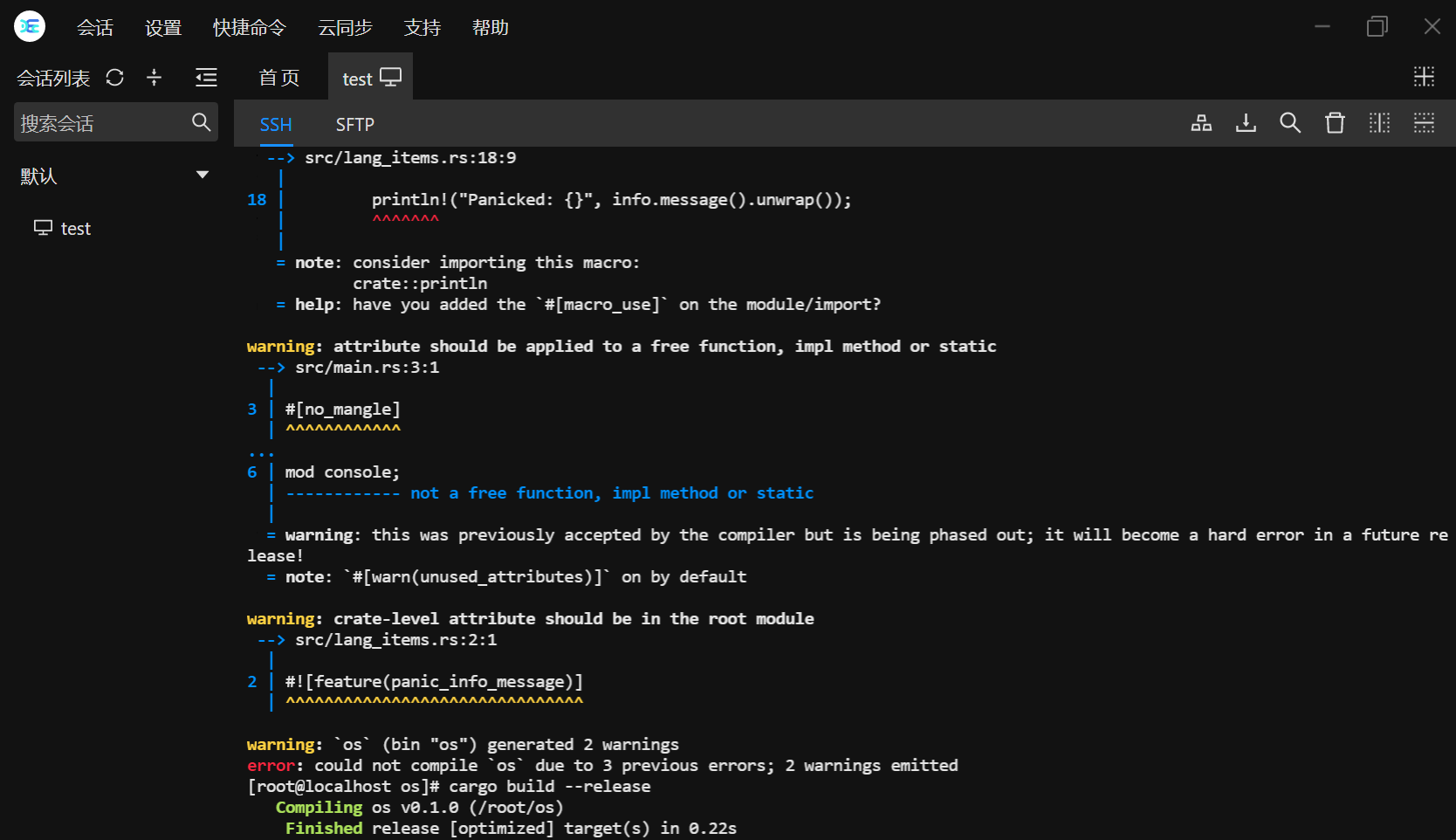


1. 为了测试我们的实现是否正确，我们将 rust\_main 改为



1. 在 os 主目录下，编译构建我们现在的代码

cargo build –release



1. 使用该命令去除内核可执行文件中的元数据
2. 再使用 Qemu 运行我们的内核

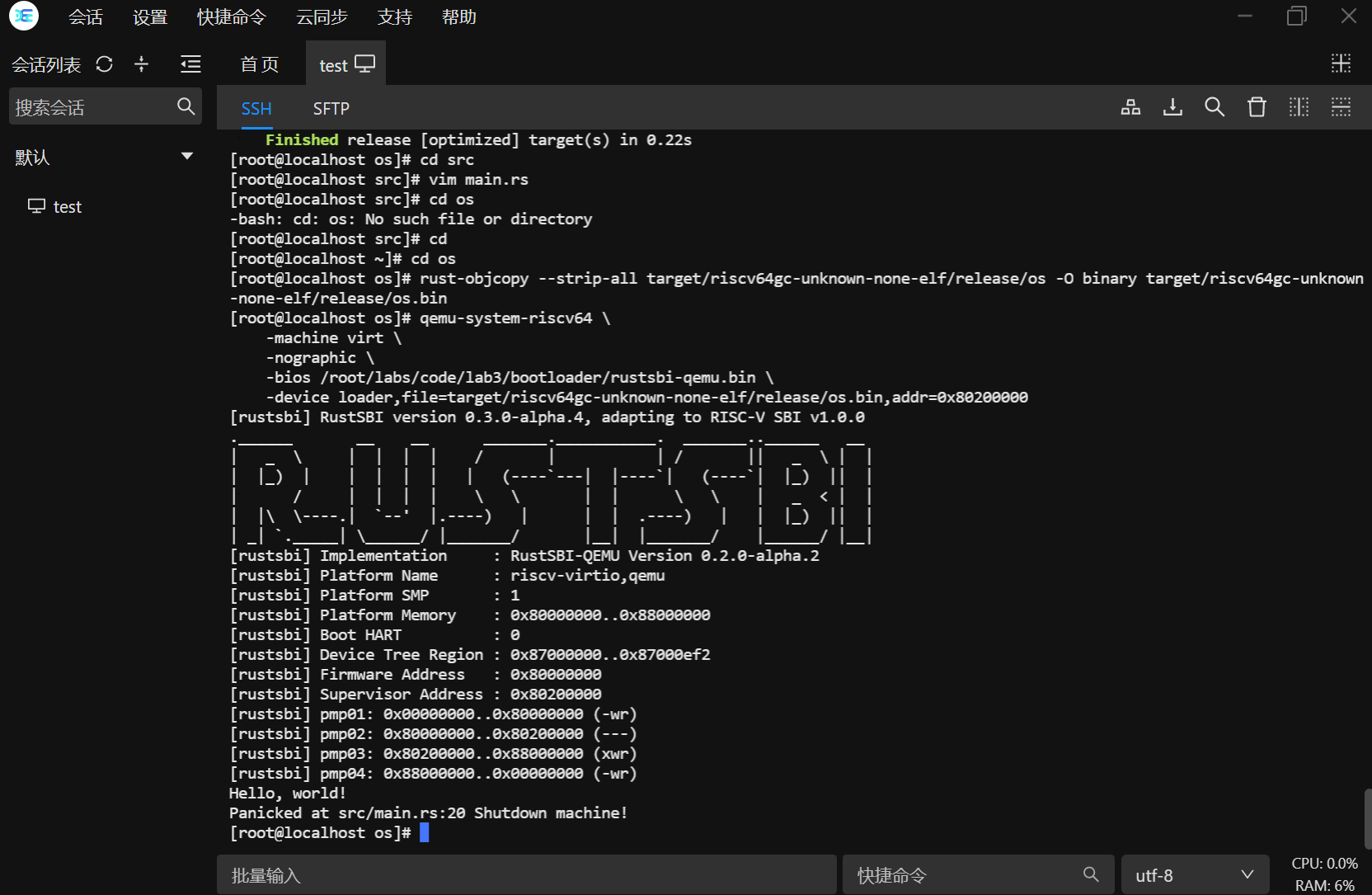
# qemu-system-riscv64 \

-machine virt \

-nographic \

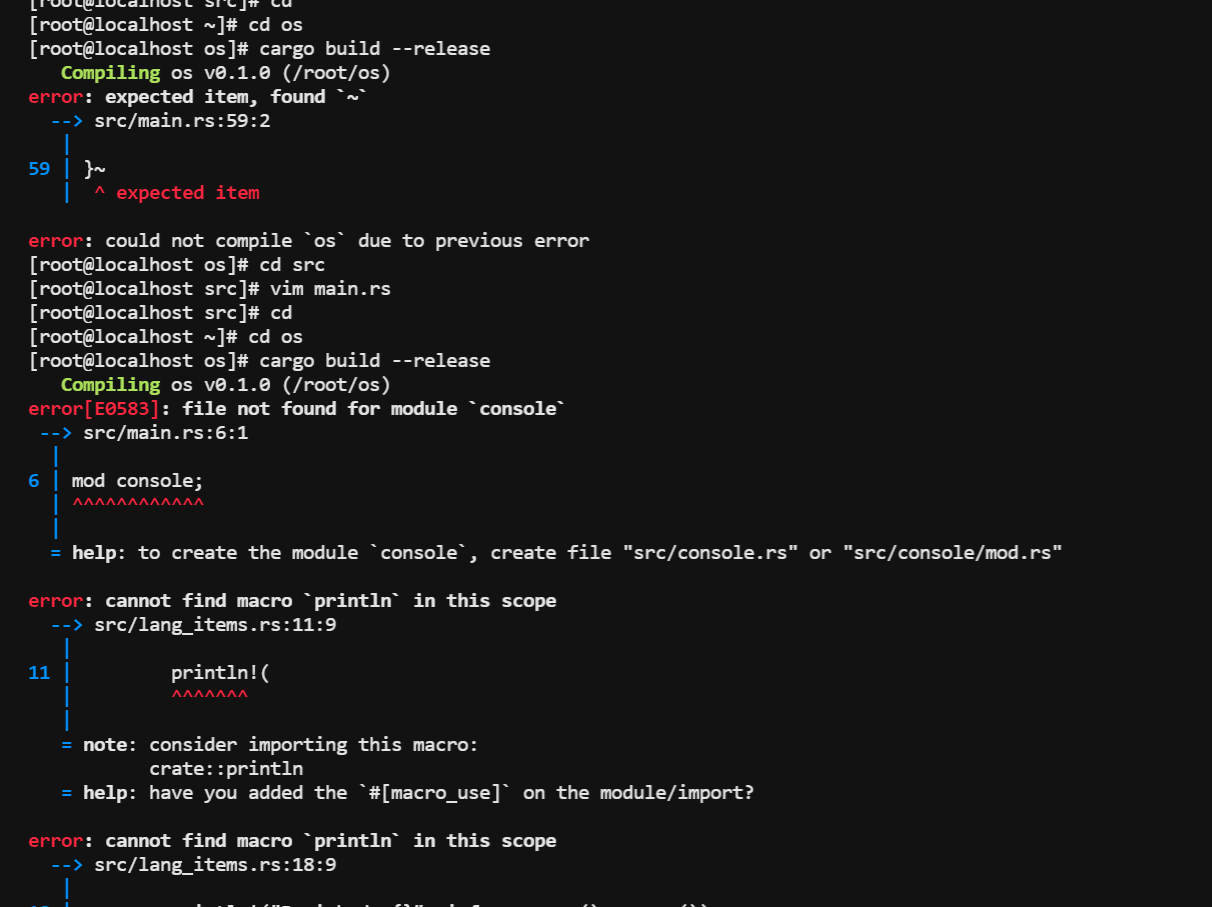
-bios /root/labs/code/lab3/bootloader/rustsbi-qemu.bin \

-device loader,file=target/riscv64gc-unknown-none-elf/release/os.bin,addr=0x80200000

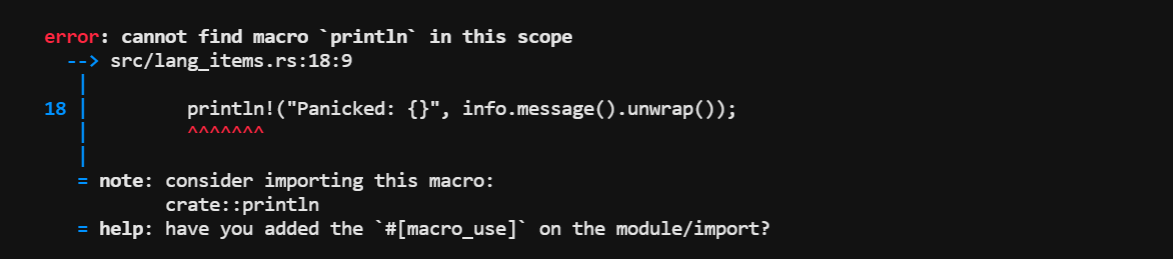


# 5 实验总结

1. 编译过程中出现了许多问题



这里是在main.rs文件里误加入了一些多余的符号



这里是忘记在main.rs中加入#[macro\_use]

都是粗心造成的一些低级错误

（2）

通过本次实验我学习到了很多知识

首先是计算机的各个硬件组成部分，如 CPU 和物理内存；

还学习到了Qemu 模拟器的抽象模型、使用方法以及启动流程

了解了程序内存布局和编译流程，特别是链接的相关知识。

最终实现了一个可以在屏幕上打印 Hello world! 和关机操作的内核。