|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| 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

构造包含操作系统内核和多个应用程序的单一执行程序

通过批处理支持多个程序的自动加载和运行

操作系统利用硬件特权级机制，实现对操作系统自身的保护

实现特权级的穿越

支持跨特权级的系统调用功能

# 2 实验过程设计

1. 了解特权级机制的软硬件设计思路，以及 RISC-V 的特权级架构，包括特权指令的描述
2. 设计实现被批处理系统逐个加载并运行的应用程序
3. 实现批处理操作系统，应用放置采用 “静态绑定” 的方式，而操作系统加载应用则采用 “动态加载” 的方式
4. 了解 在 RISC-V 64 处理器提供的 U/S 特权级下，批处理操作系统和应用程序如何相互配合，完成特权级切换的

# 3 源代码及注释

表 3-1 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | 00hello\_world.rs |
| **主要功能:** | 在屏幕上打印一行 Hello world from user mode program! |
| **源代码:**  **#![no\_std]**  **#![no\_main]**  ***//引入了外部库：user/src 目录下的 lib.rs***  **#[macro\_use]**  **extern crate user\_lib;**  **#[no\_mangle]**  **fn main() -> i32 {**  **println!("Hello world from user mode program!");**  **0**  **}** | |
| **文件名:** | 01store\_fault.rs |
| **主要功能:** | 访问一个非法的物理地址，测试批处理系统是否会被该错误影响 |
| **源代码:**  **#![no\_std]**  **#![no\_main]**  **//引入了外部库：user/src 目录下的 lib.rs**  **#[macro\_use]**  **extern crate user\_lib;**  **#[no\_mangle]**  **fn main() -> i32 {**  **println!("Into Test store\_fault, we will insert an invalid store operation...");**  **println!("Kernel should kill this application!");**  **unsafe {**  **core::ptr::null\_mut::<u8>().write\_volatile(0);**  **}**  **0**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | Power.rs |
| **主要功能:** | 不断在计算操作和打印字符串操作之间进行特权级切换 |
| **源代码:**  **#![no\_std]**  **#![no\_main]**  **//引入了外部库：user/src 目录下的 lib.rs**  **#[macro\_use]**  **extern crate user\_lib;**  **const SIZE: usize = 10;**  **const P: u32 = 3;**  **const STEP: usize = 100000;**  **const MOD: u32 = 10007;**  **#[no\_mangle]**  **fn main() -> i32 {**  **let mut pow = [0u32; SIZE];**  **let mut index: usize = 0;**  **pow[index] = 1;**  **for i in 1..=STEP {**  **let last = pow[index];**  **index = (index + 1) % SIZE;**  **pow[index] = last \* P % MOD;**  **if i % 10000 == 0 {**  **println!("{}^{}={}(MOD {})", P, i, pow[index], MOD);**  **}**  **}**  **println!("Test power OK!");**  **0**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | lib.rs |
| **主要功能:** | 定义了用户库的入口点 \_start |
| **源代码:**  **#![no\_std]**  **#![feature(linkage)]**  **#![feature(panic\_info\_message)]**  **#[macro\_use]**  **pub mod console;**  **mod lang\_items;**  **mod syscall;**  **//定义了用户库的入口点 \_start ：**  **#[no\_mangle]**  **//使用 Rust 的宏将 \_start 这段代码编译后的汇编代码中放在一个名为 .text.entry 的代码段中，方便我们在后续链接的时候调整它的位置使得它能够作为用户库的入口。**  **#[link\_section = ".text.entry"]**  **pub extern "C" fn \_start() -> ! {**  **clear\_bss();**  **//调用用户库提供的 exit 接口退出应用程序**  **exit(main());**  **//将 main 函数的返回值告知批处理系统。**  **panic!("unreachable after sys\_exit!");**  **}**  **#[linkage = "weak"]**  **#[no\_mangle]**  **//调用 main 函数得到一个类型为 i32 的返回值**  **fn main() -> i32 {**  **panic!("Cannot find main!");**  **}**  **//手动清空需要零初始化的 .bss 段**  **fn clear\_bss() {**  **extern "C" {**  **fn start\_bss();**  **fn end\_bss();**  **}**  **(start\_bss as usize..end\_bss as usize).for\_each(|addr| unsafe {**  **(addr as \*mut u8).write\_volatile(0);**  **});**  **}**  **use syscall::\*;**  **//将sys\_write 和 sys\_exit在用户库 user\_lib 中进一步封装，从而更加接近在 Linux 等平台的实际系统调用接口：**  **pub fn write(fd: usize, buf: &[u8]) -> isize {**  **sys\_write(fd, buf)**  **}**  **pub fn exit(exit\_code: i32) -> isize {**  **sys\_exit(exit\_code)**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | syscall.rs |
| **主要功能:** | 应用程序通过 ecall 调用批处理系统提供的接口, 触发名为 Environment call from U-mode 的异常 |
| **源代码:**  **use core::arch::asm;**  **const SYSCALL\_WRITE: usize = 64;**  **const SYSCALL\_EXIT: usize = 93;**  **fn syscall(id: usize, args: [usize; 3]) -> isize {**  **let mut ret: isize;**  **unsafe {**  **asm!(**  **"ecall",**  **inlateout("x10") args[0] => ret,**  **in("x11") args[1],**  **in("x12") args[2],**  **in("x17") id**  **);**  **}**  **ret**  **}**  **/// 功能：将内存中缓冲区中的数据写入文件。**  **/// 参数：`fd` 表示待写入文件的文件描述符；**  **/// `buf` 表示内存中缓冲区的起始地址；**  **/// `len` 表示内存中缓冲区的长度。**  **/// 返回值：返回成功写入的长度。**  **//将 syscall 进行包装：**  **/// syscall ID：64**  **pub fn sys\_write(fd: usize, buffer: &[u8]) -> isize {**  **syscall(SYSCALL\_WRITE, [fd, buffer.as\_ptr() as usize, buffer.len()])**  **}**  **/// 功能：退出应用程序并将返回值告知批处理系统。**  **/// 参数：`xstate` 表示应用程序的返回值。**  **/// 返回值：该系统调用不应该返回。**  **/// syscall ID：93**  **//将 syscall 进行包装：**  **pub fn sys\_exit(exit\_code: i32) -> isize {**  **syscall(SYSCALL\_EXIT, [exit\_code as usize, 0, 0])**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | console.rs |
| **主要功能:** | 实现格式化输出 |
| **源代码:**  **use super::write;**  **use core::fmt::{self, Write};**  **struct Stdout;**  **const STDOUT: usize = 1;**  **impl Write for Stdout {**  **fn write\_str(&mut self, s: &str) -> fmt::Result {**  **write(STDOUT, s.as\_bytes());**  **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)+)?));**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | 03priv\_inst.rs |
| **主要功能:** | 会尝试在用户态执行内核态的特权指令 sret |
| **源代码:**  **#![no\_std]**  **#![no\_main]**  **#[macro\_use]**  **extern crate user\_lib;**  **use core::arch::asm;**  **#[no\_mangle]**  **fn main() -> i32 {**  **println!("Try to execute privileged instruction in U Mode");**  **println!("Kernel should kill this application!");**  **unsafe {**  **asm!("sret");**  **}**  **0**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | 04priv\_csr.rs |
| **主要功能:** | 试图在用户态修改内核态 CSR sstatus |
| **源代码:**  **#![no\_std]**  **#![no\_main]**  **#[macro\_use]**  **extern crate user\_lib;**  **use riscv::register::sstatus::{self, SPP};**  **#[no\_mangle]**  **fn main() -> i32 {**  **println!("Try to access privileged CSR in U Mode");**  **println!("Kernel should kill this application!");**  **unsafe {**  **sstatus::set\_spp(SPP::User);**  **}**  **0**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | link\_app.S |
| **主要功能:** | 插入应用程序的二进制镜像 |
| **源代码:**  **.align 3**  **.section .data**  **.global \_num\_app**  **\_num\_app:**  **.quad 5**  **.quad app\_0\_start**  **.quad app\_1\_start**  **.quad app\_2\_start**  **.quad app\_3\_start**  **.quad app\_4\_start**  **.quad app\_4\_end**  **.section .data**  **.global app\_0\_start**  **.global app\_0\_end**  **app\_0\_start:**  **.incbin "../user/target/riscv64gc-unknown-none-elf/release/00hello\_world.bin"**  **app\_0\_end:**  **//下面的五个数据段分别插入了五个应用程序的二进制镜像**  **.section .data**  **.global app\_1\_start**  **.global app\_1\_end**  **//指示开始位置。**  **app\_1\_start:**  **.incbin "../user/target/riscv64gc-unknown-none-elf/release/01store\_fault.bin"**  **//指示结束位置。**  **app\_1\_end:**  **.section .data**  **.global app\_2\_start**  **.global app\_2\_end**  **app\_2\_start:**  **.incbin "../user/target/riscv64gc-unknown-none-elf/release/02power.bin"**  **app\_2\_end:**  **.section .data**  **.global app\_3\_start**  **.global app\_3\_end**  **app\_3\_start:**  **.incbin "../user/target/riscv64gc-unknown-none-elf/release/03priv\_inst.bin"**  **app\_3\_end:**  **.section .data**  **.global app\_4\_start**  **.global app\_4\_end**  **app\_4\_start:**  **.incbin "../user/target/riscv64gc-unknown-none-elf/release/04priv\_csr.bin"**  **app\_4\_end:** | |

|  |  |
| --- | --- |
| **文件名:** | batch.rs |
| **主要功能:** | 应用管理器 AppManager |
| **源代码:**  **//! batch subsystem**  **use crate::sync::UPSafeCell;**  **use crate::trap::TrapContext;**  **use core::arch::asm;**  **use lazy\_static::\*;**  **//** **KernelStack 和 UserStack 分别表示用户栈和内核栈，它们都只是字节数组的简单包装**  **//内核栈和用户栈的大小分别为 8KiB**  **const USER\_STACK\_SIZE: usize = 4096 \* 2;**  **const KERNEL\_STACK\_SIZE: usize = 4096 \* 2;**  **const MAX\_APP\_NUM: usize = 16;**  **const APP\_BASE\_ADDRESS: usize = 0x80400000;**  **const APP\_SIZE\_LIMIT: usize = 0x20000;**  **#[repr(align(4096))]**  **struct KernelStack {**  **data: [u8; KERNEL\_STACK\_SIZE],**  **}**  **#[repr(align(4096))]**  **struct UserStack {**  **data: [u8; USER\_STACK\_SIZE],**  **}**  **static KERNEL\_STACK: KernelStack = KernelStack {**  **data: [0; KERNEL\_STACK\_SIZE],**  **};**  **static USER\_STACK: UserStack = UserStack {**  **data: [0; USER\_STACK\_SIZE],**  **};**  **impl KernelStack {**  **fn get\_sp(&self) -> usize {**  **self.data.as\_ptr() as usize + KERNEL\_STACK\_SIZE**  **}**  **pub fn push\_context(&self, cx: TrapContext) -> &'static mut TrapContext {**  **let cx\_ptr = (self.get\_sp() - core::mem::size\_of::<TrapContext>()) as \*mut TrapContext;**  **unsafe {**  **\*cx\_ptr = cx;**  **}**  **unsafe { cx\_ptr.as\_mut().unwrap() }**  **}**  **}**  **//实现 get\_sp 方法来获取栈顶地址**  **//将 sp 寄存器的值修改为 get\_sp 的返回值**  **impl UserStack {**  **fn get\_sp(&self) -> usize {**  **self.data.as\_ptr() as usize + USER\_STACK\_SIZE**  **}**  **}**  **//应用管理器 AppManager 结构体定义如下：**  **//需要保存和维护的信息都在 AppManager 里面**  **struct AppManager {**  **num\_app: usize,**  **current\_app: usize,**  **app\_start: [usize; MAX\_APP\_NUM + 1],**  **}**  **impl AppManager {**  **pub fn print\_app\_info(&self) {**  **println!("[kernel] num\_app = {}", self.num\_app);**  **for i in 0..self.num\_app {**  **println!(**  **"[kernel] app\_{} [{:#x}, {:#x})",**  **i,**  **self.app\_start[i],**  **self.app\_start[i + 1]**  **);**  **}**  **}**  **//将参数 app\_id 对应的应用程序的二进制镜像加载到物理内存以 0x80400000 起始的位置**  **unsafe fn load\_app(&self, app\_id: usize) {**  **if app\_id >= self.num\_app {**  **println!("All applications completed!");**  **use crate::board::QEMUExit;**  **crate::board::QEMU\_EXIT\_HANDLE.exit\_success();**  **}**  **println!("[kernel] Loading app\_{}", app\_id);**  **// clear icache 用来清理 i-cache 的**  **asm!("fence.i");**  **// clear app area**  **core::slice::from\_raw\_parts\_mut(APP\_BASE\_ADDRESS as \*mut u8, APP\_SIZE\_LIMIT).fill(0);**  **let app\_src = core::slice::from\_raw\_parts(**  **self.app\_start[app\_id] as \*const u8,**  **self.app\_start[app\_id + 1] - self.app\_start[app\_id],**  **);**  **let app\_dst = core::slice::from\_raw\_parts\_mut(APP\_BASE\_ADDRESS as \*mut u8, app\_src.len());**  **app\_dst.copy\_from\_slice(app\_src);**  **}**  **pub fn get\_current\_app(&self) -> usize {**  **self.current\_app**  **}**  **pub fn move\_to\_next\_app(&mut self) {**  **self.current\_app += 1;**  **}**  **}**  **//以尽量少的 unsafe code 来初始化 AppManager 的全局实例 APP\_MANAGER**  **//找到 link\_app.S 中提供的符号 \_num\_app ，并从这里开始解析出应用数量以及各个应用的起始地址**  **//使用了外部库 lazy\_static 提供的 lazy\_static! 宏**  **lazy\_static! {**  **static ref APP\_MANAGER: UPSafeCell<AppManager> = unsafe {**  **UPSafeCell::new({**  **extern "C" {**  **fn \_num\_app();**  **}**  **let num\_app\_ptr = \_num\_app as usize as \*const usize;**  **let num\_app = num\_app\_ptr.read\_volatile();**  **let mut app\_start: [usize; MAX\_APP\_NUM + 1] = [0; MAX\_APP\_NUM + 1];**  **let app\_start\_raw: &[usize] =**  **core::slice::from\_raw\_parts(num\_app\_ptr.add(1), num\_app + 1);**  **app\_start[..=num\_app].copy\_from\_slice(app\_start\_raw);**  **AppManager {**  **num\_app,**  **current\_app: 0,**  **app\_start,**  **}**  **})**  **};**  **}**  **/// init batch subsystem**  **pub fn init() {**  **print\_app\_info();**  **}**  **/// print apps info**  **pub fn print\_app\_info() {**  **APP\_MANAGER.exclusive\_access().print\_app\_info();**  **}**  **/// run next app**  **pub fn run\_next\_app() -> ! {**  **let mut app\_manager = APP\_MANAGER.exclusive\_access();**  **let current\_app = app\_manager.get\_current\_app();**  **unsafe {**  **app\_manager.load\_app(current\_app);**  **}**  **app\_manager.move\_to\_next\_app();**  **drop(app\_manager);**  **// before this we have to drop local variables related to resources manually**  **// and release the resources**  **extern "C" {**  **fn \_\_restore(cx\_addr: usize);**  **}**  **unsafe {**  **//在内核栈上压入一个 Trap 上下文**  **\_\_restore(KERNEL\_STACK.push\_context(TrapContext::app\_init\_context(**  **APP\_BASE\_ADDRESS,**  **USER\_STACK.get\_sp(),**  **)) as \*const \_ as usize);**  **}**  **panic!("Unreachable in batch::run\_current\_app!");**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | Up.rs |
| **主要功能:** | 允许我们在单核上安全使用可变全局变量 |
| **源代码:**  **//! Uniprocessor interior mutability primitives**  **use core::cell::{RefCell, RefMut};**  **/// Wrap a static data structure inside it so that we are**  **/// able to access it without any `unsafe`.**  **///**  **/// We should only use it in uniprocessor.**  **///**  **/// In order to get mutable reference of inner data, call**  **/// `exclusive\_access`.**  **pub struct UPSafeCell<T> {**  **/// inner data**  **inner: RefCell<T>,**  **}**  **unsafe impl<T> Sync for UPSafeCell<T> {}**  **impl<T> UPSafeCell<T> {**  **/// User is responsible to guarantee that inner struct is only used in**  **/// uniprocessor.**  **pub unsafe fn new(value: T) -> Self {**  **Self {**  **inner: RefCell::new(value),**  **}**  **}**  **/// Exclusive access inner data in UPSafeCell. Panic if the data has been borrowed.**  **pub fn exclusive\_access(&self) -> RefMut<'\_, T> {**  **self.inner.borrow\_mut()**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | context.rs |
| **主要功能:** | 在 Trap 发生时需要保存的物理资源内容 |
| **源代码:**  **use riscv::register::sstatus::{self, Sstatus, SPP};**  **/// Trap Context**  **//包含所有的通用寄存器 x0~x31 ，还有 sstatus 和 sep**  **#[repr(C)]**  **pub struct TrapContext {**  **/// general regs[0..31]**  **pub x: [usize; 32],**  **/// CSR sstatus**  **pub sstatus: Sstatus,**  **/// CSR sepc**  **pub sepc: usize,**  **}**  **impl TrapContext {**  **/// set stack pointer to x\_2 reg (sp)**  **pub fn set\_sp(&mut self, sp: usize) {**  **self.x[2] = sp;**  **}**  **//app\_init\_context 方法，修改其中的 sepc 寄存器为应用程序入口点 entry， sp 寄存器为我们设定的一个栈指针，并将 sstatus 寄存器的 SPP 字段设置为 User 。**  **/// init app context**  **pub fn app\_init\_context(entry: usize, sp: usize) -> Self {**  **let mut sstatus = sstatus::read(); // CSR sstatus**  **sstatus.set\_spp(SPP::User); //previous privilege mode: user mode**  **let mut cx = Self {**  **x: [0; 32],**  **sstatus,**  **sepc: entry, // entry point of app**  **};**  **cx.set\_sp(sp); // app's user stack pointer**  **cx // return initial Trap Context of app**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | // os/src/trap/mod.rs |
| **主要功能:** | Trap 上下文保存和恢复 |
| **源代码:**  **//! Trap handling functionality**  **//!**  **//! For rCore, we have a single trap entry point, namely `\_\_alltraps`. At**  **//! initialization in [`init()`], we set the `stvec` CSR to point to it.**  **//!**  **//! All traps go through `\_\_alltraps`, which is defined in `trap.S`. The**  **//! assembly language code does just enough work restore the kernel space**  **//! context, ensuring that Rust code safely runs, and transfers control to**  **//! [`trap\_handler()`].**  **//!**  **//! It then calls different functionality based on what exactly the exception**  **//! was. For example, timer interrupts trigger task preemption, and syscalls go**  **//! to [`syscall()`].**  **mod context;**  **use crate::batch::run\_next\_app;**  **use crate::syscall::syscall;**  **use core::arch::global\_asm;**  **use riscv::register::{**  **mtvec::TrapMode,**  **scause::{self, Exception, Trap},**  **stval, stvec,**  **};**  **global\_asm!(include\_str!("trap.S"));**  **//引入了一个外部符号 \_\_alltraps ，并将 stvec 设置为 Direct 模式指向它的地址**  **/// initialize CSR `stvec` as the entry of `\_\_alltraps`**  **pub fn init() {**  **extern "C" {**  **fn \_\_alltraps();**  **}**  **unsafe {**  **stvec::write(\_\_alltraps as usize, TrapMode::Direct);**  **}**  **}**  **#[no\_mangle]**  **//完成分发和处理**  **/// handle an interrupt, exception, or system call from user space**  **pub fn trap\_handler(cx: &mut TrapContext) -> &mut TrapContext {**  **let scause = scause::read(); // get trap cause**  **let stval = stval::read(); // get extra value**  **//根据 scause 寄存器所保存的 Trap 的原因进行分发处理**  **match scause.cause() {**  **Trap::Exception(Exception::UserEnvCall) => {**  **cx.sepc += 4;**  **cx.x[10] = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]) as usize;**  **}**  **Trap::Exception(Exception::StoreFault) | Trap::Exception(Exception::StorePageFault) => {**  **println!("[kernel] PageFault in application, kernel killed it.");**  **run\_next\_app();**  **}**  **Trap::Exception(Exception::IllegalInstruction) => {**  **println!("[kernel] IllegalInstruction in application, kernel killed it.");**  **run\_next\_app();**  **}**  **\_ => {**  **panic!(**  **"Unsupported trap {:?}, stval = {:#x}!",**  **scause.cause(),**  **stval**  **);**  **}**  **}**  **cx**  **}**  **pub use context::TrapContext;** | |

|  |  |
| --- | --- |
| **文件名:** | trap.S |
| **主要功能:** | 保存 Trap 上下文 |
| **源代码:**  **.altmacro**  **.macro SAVE\_GP n**  **sd x\n, \n\*8(sp)**  **.endm**  **.macro LOAD\_GP n**  **ld x\n, \n\*8(sp)**  **.endm**  **.section .text**  **.globl \_\_alltraps**  **.globl \_\_restore**  **//使用 .align 将 \_\_alltraps 的地址 4 字节对齐，这是 RISC-V 特权级规范的要求**  **.align 2**  **\_\_alltraps:**  **//csrrw 原型是 可以将 CSR 当前的值读到通用寄存器 rd 中，然后将通用寄存器 rs 的值写入该 CSR 。因此这里起到的是交换 sscratch 和 sp 的效果。在这一行之前 sp 指向用户栈， sscratch 指向内核栈（原因稍后说明），现在 sp 指向内核栈， sscratch 指向用户栈。**  **csrrw sp, sscratch, sp**  **# now sp->kernel stack, sscratch->user stack**  **# allocate a TrapContext on kernel stack**  **addi sp, sp, -34\*8**  **# save general-purpose registers**  **sd x1, 1\*8(sp)**  **# skip sp(x2), we will save it later**  **sd x3, 3\*8(sp)**  **# skip tp(x4), application does not use it**  **# save x5~x31**  **.set n, 5**  **.rept 27**  **SAVE\_GP %n**  **.set n, n+1**  **.endr**  **# we can use t0/t1/t2 freely, because they were saved on kernel stack**  **csrr t0, sstatus**  **csrr t1, sepc**  **sd t0, 32\*8(sp)**  **sd t1, 33\*8(sp)**  **# read user stack from sscratch and save it on the kernel stack**  **csrr t2, sscratch**  **sd t2, 2\*8(sp)**  **# set input argument of trap\_handler(cx: &mut TrapContext)**  **mv a0, sp**  **call trap\_handler**  **//从栈上的 Trap 上下文恢复的 \_\_restore ：**  **\_\_restore:**  **# case1: start running app by \_\_restore**  **# case2: back to U after handling trap**  **mv sp, a0**  **# now sp->kernel stack(after allocated), sscratch->user stack**  **# restore sstatus/sepc**  **ld t0, 32\*8(sp)**  **ld t1, 33\*8(sp)**  **ld t2, 2\*8(sp)**  **csrw sstatus, t0**  **csrw sepc, t1**  **csrw sscratch, t2**  **# restore general-purpuse registers except sp/tp**  **ld x1, 1\*8(sp)**  **ld x3, 3\*8(sp)**  **.set n, 5**  **.rept 27**  **LOAD\_GP %n**  **.set n, n+1**  **.endr**  **# release TrapContext on kernel stack**  **addi sp, sp, 34\*8**  **# now sp->kernel stack, sscratch->user stack**  **csrrw sp, sscratch, sp**  **sret** | |

|  |  |
| --- | --- |
| **文件名:** | // os/src/syscall/mod.rs |
| **主要功能:** | 处理系统调用 |
| **源代码:**  **//将传进来的参数 args 转化成能够被具体的系统调用处理函数接受的类型**  **pub fn syscall(syscall\_id: usize, args: [usize; 3]) -> isize {**  **match syscall\_id {**  **SYSCALL\_WRITE => sys\_write(args[0], args[1] as \*const u8, args[2]),**  **SYSCALL\_EXIT => sys\_exit(args[0] as i32),**  **\_ => panic!("Unsupported syscall\_id: {}", syscall\_id),**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | // os/src/syscall/fs.rs |
| **主要功能:** | 将传进来的参数 args 转化成能够被具体的系统调用处理函数接受的类型 |
| **源代码:**  **// os/src/syscall/fs.rs**  **const FD\_STDOUT: usize = 1;**  **pub fn sys\_write(fd: usize, buf: \*const u8, len: usize) -> isize {**  **match fd {**  **FD\_STDOUT => {**  **let slice = unsafe { core::slice::from\_raw\_parts(buf, len) };**  **let str = core::str::from\_utf8(slice).unwrap();**  **print!("{}", str);**  **len as isize**  **},**  **\_ => {**  **panic!("Unsupported fd in sys\_write!");**  **}**  **}**  **}**  **// os/src/syscall/process.rs**  **pub fn sys\_exit(xstate: i32) -> ! {**  **println!("[kernel] Application exited with code {}", xstate);**  **run\_next\_app()**  **}** | |

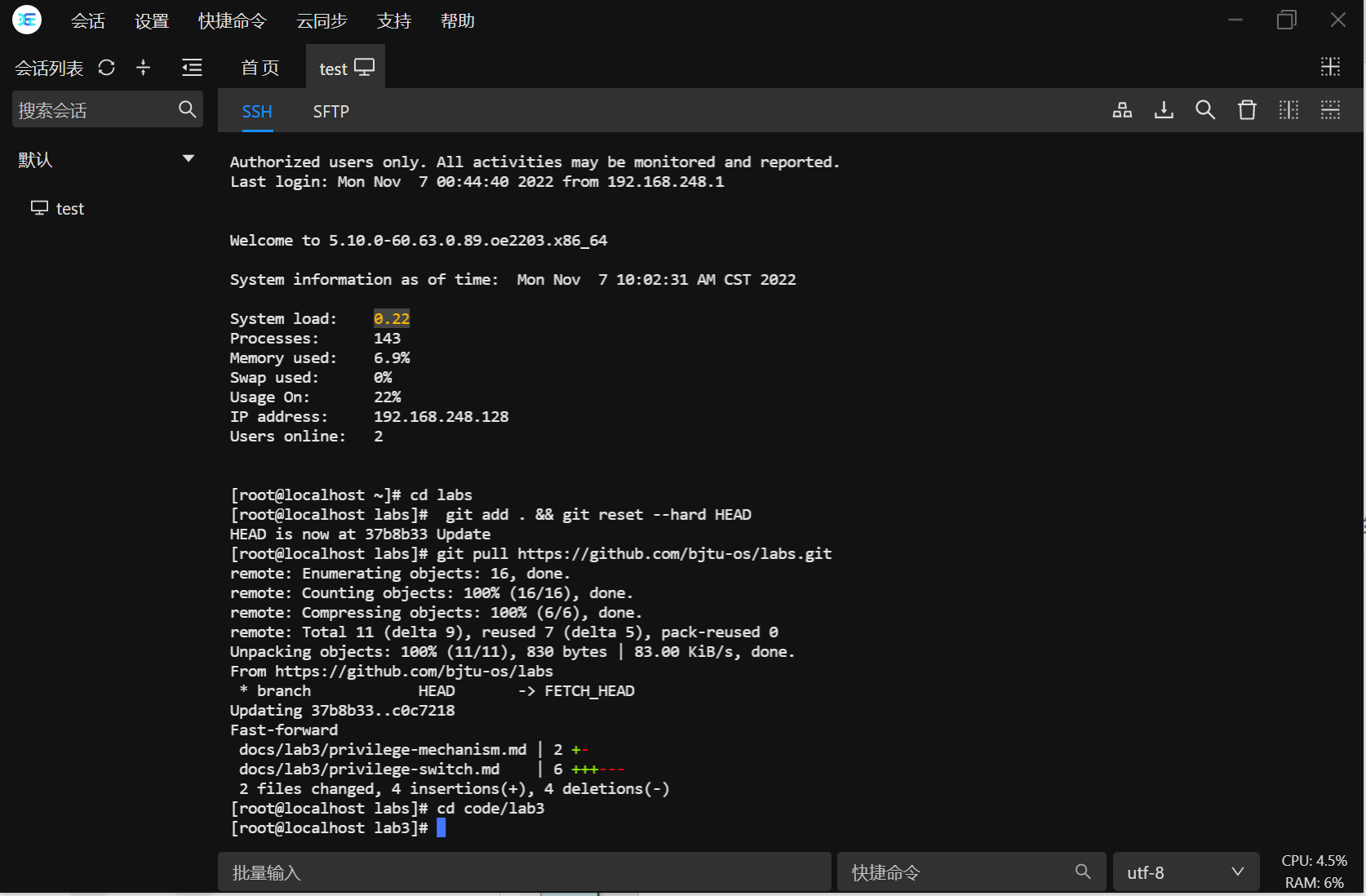
|  |  |
| --- | --- |
| **文件名:** | linker.ld |
| **主要功能:** | 链接脚本 |
| **源代码:**  **OUTPUT\_ARCH(riscv)**  **//将 \_start 所在的 .text.entry 放在整个程序的开头，也就是说批处理系统只要在加载之后跳转到 0x80400000 就已经进入了 用户库的入口点，并会在初始化之后跳转到应用程序主逻辑；**  **ENTRY(\_start)**  **//将程序的起始物理地址调整为 0x80400000 ，三个应用程序都会被加载到这个物理地址上运行**  **BASE\_ADDRESS = 0x80400000;**  **SECTIONS**  **{**  **. = BASE\_ADDRESS;**  **.text : {**  **\*(.text.entry)**  **\*(.text .text.\*)**  **}**  **.rodata : {**  **\*(.rodata .rodata.\*)**  **\*(.srodata .srodata.\*)**  **}**  **.data : {**  **\*(.data .data.\*)**  **\*(.sdata .sdata.\*)**  **}**  **//提供了最终生成可执行文件的 .bss 段的起始和终止地址，方便 clear\_bss 函数使用。**  **.bss : {**  **start\_bss = .;**  **\*(.bss .bss.\*)**  **\*(.sbss .sbss.\*)**  **end\_bss = .;**  **}**  **/DISCARD/ : {**  **\*(.eh\_frame)**  **\*(.debug\*)**  **}**  **}** | |

表 3-2 关键命令解释

|  |  |  |
| --- | --- | --- |
| **序号** | **命令名** | **主要作用** |
| **1** | **qemu-riscv64 ./03priv\_inst** | 在qemu-riscv64模拟器下执行应用程序03priv\_inst，但在执行特权指令会出错 |
| **2** | **qemu-riscv64 ./04priv\_csr** | 在qemu-riscv64模拟器下执行应用程序04priv\_csr，执行访问特权级 CSR 的指令出错 |
| **3** | **qemu-riscv64 ./00hello\_world** | 在qemu-riscv64模拟器下执行应用程序00hello\_world，正确显示了字符串 |
| **4** | **qemu-riscv64 ./01store\_fault** | 在qemu-riscv64模拟器下执行应用程序01store\_fault，导致应用和 qemu-riscv64 被 Linux 内核杀死 |
| **5** | **qemu-riscv64 ./02power** | 在qemu-riscv64模拟器下执行应用程序02power，测试通过 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

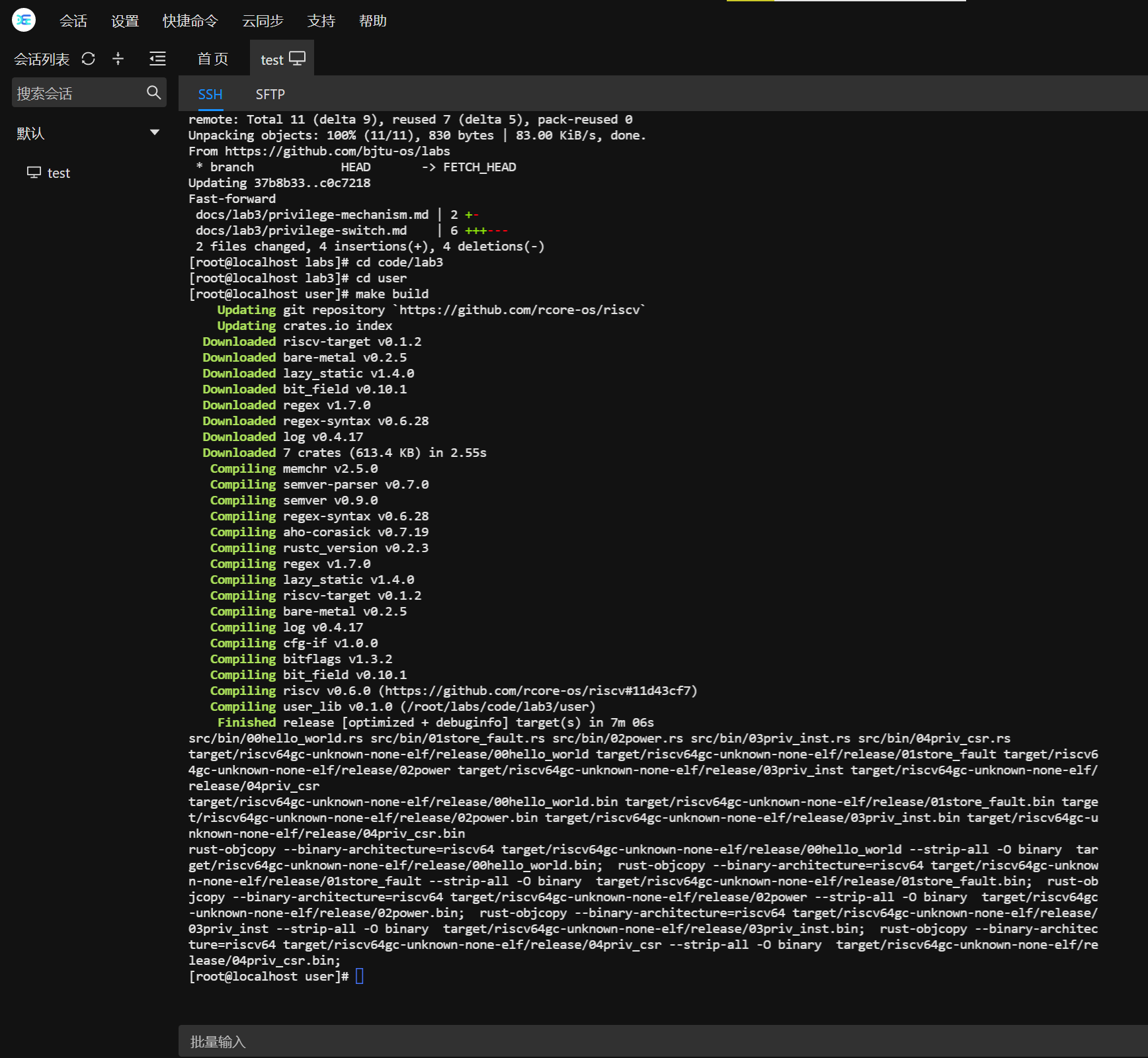
# 4 运行结果与分析

1. 更新lab3的代码

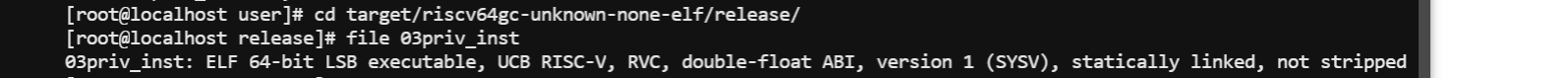


1. 在用户态模拟器 qemu-riscv64 执行应用 03priv\_inst和应用 04priv\_csr，由于这两个应用都会修改内核态，所以不能正确执行

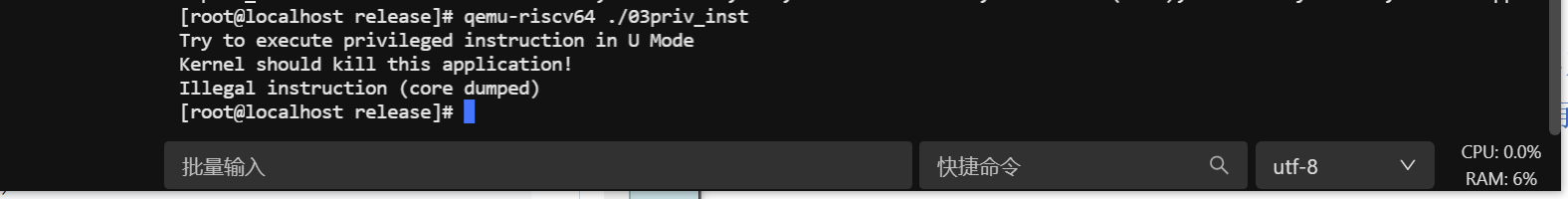
在 user 目录下 make build



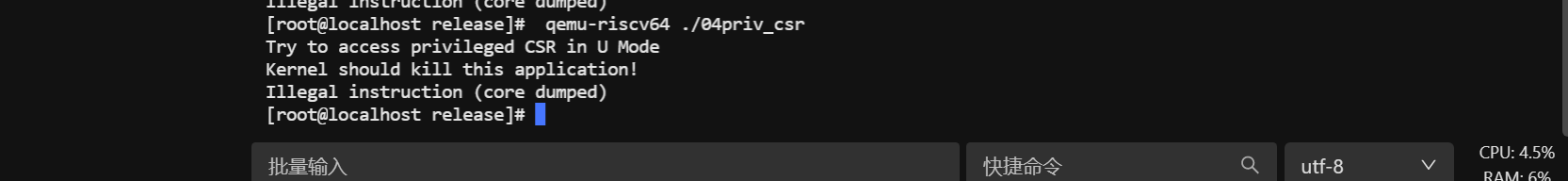
在 target/riscv64gc-unknown-none-elf/release 目录下生成一个同名的 ELF 可执行文件



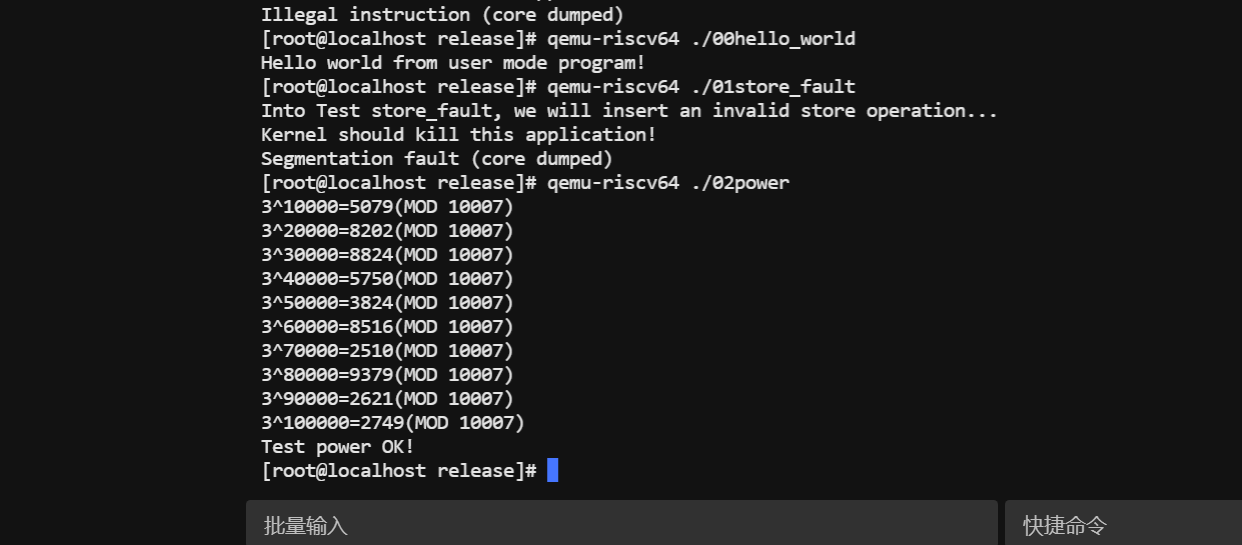
在qemu-riscv64模拟器下执行应用程序03priv\_inst，执行特权指令出错



在qemu-riscv64模拟器下执行应用程序04priv\_csr，执行访问特权级 CSR 的指令出错



1. 在用户态模拟器 qemu-riscv64执行一般的用户态应用程序，是可以正确执行的



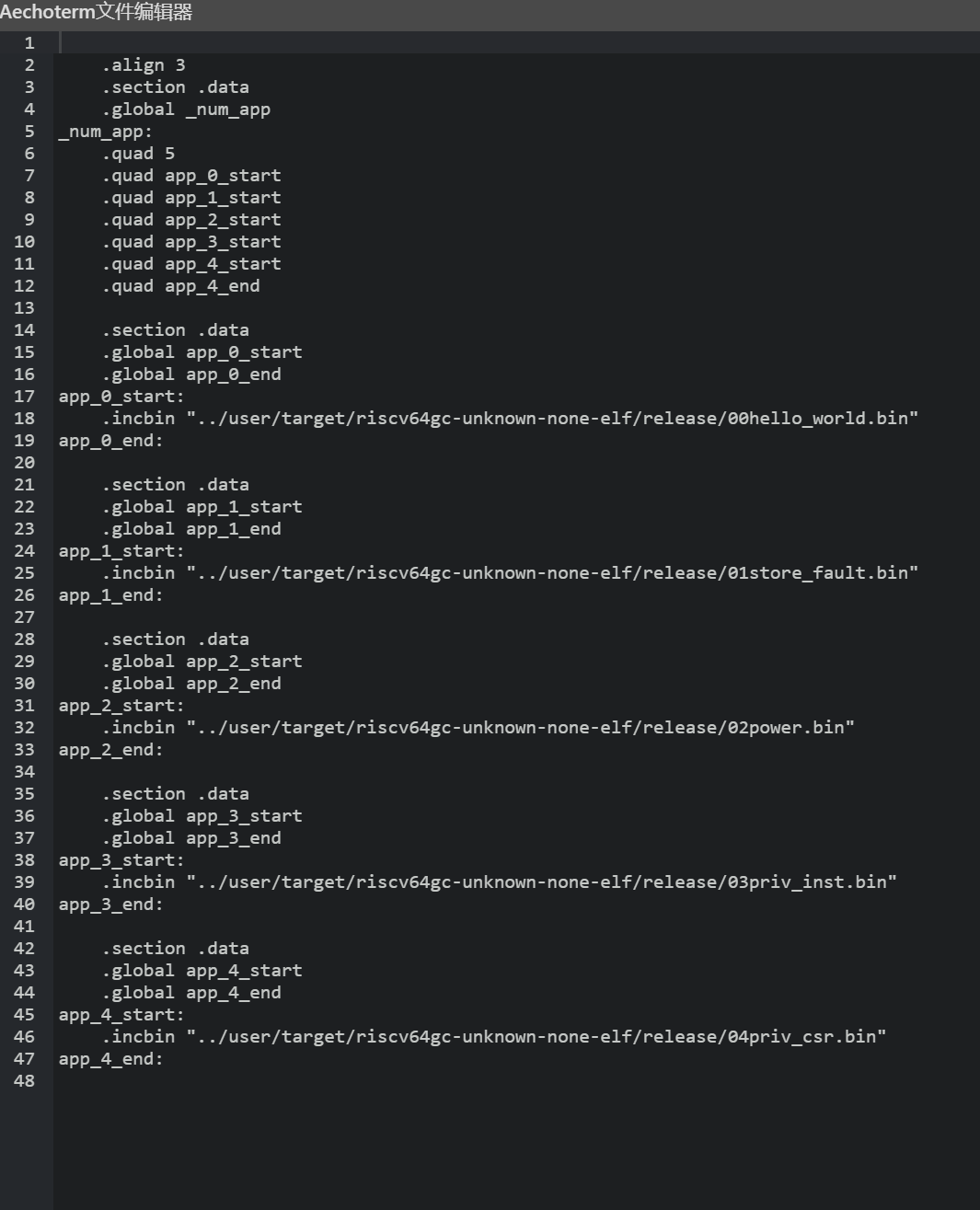
执行00hello\_world，正确显示了字符串；执行01store\_fault，导致应用和 qemu-riscv64 被 Linux 内核杀死；执行02power，测试通过。说明三个用户态应用程序都能顺利在用户态模拟器 qemu-riscv64执行。

1. 将应用程序链接到内核

在 os 目录下，构建并运行整个系统：



生成了link\_app.S



# 5 实验总结

（1）本次实验的主要目的是设计与实现批处理系统，用来自动安排程序的执行

实验的执行过程总体很顺畅，没遇到问题，但突然接触了大量代码，有些知识难以理解

比如load\_app为什么要插入汇编指令 fence.i？经过查询了解到：

*在 Qemu 模拟器的默认配置下，各类缓存如 i-cache/d-cache/TLB 都处于机制不完全甚至完全不存在的状态。目前在 Qemu 平台上，即使我们不加上刷新 i-cache 的指令，大概率也是能够正常运行的。但在 K210 物理计算机上，如果没有执行汇编指令 fence.i ，就会产生由于指令缓存的内容与对应内存中指令不一致导致的错误异常。*

解答了我的困惑

（2）在用户态模拟器 qemu-riscv64执行了五个应用程序，其中03priv\_inst会尝试在用户态执行内核态的特权指令 sret ，而应用 04priv\_csr 则会试图在用户态修改内核态 CSR sstatus，他们都越权了，应用程序应当执行在另外一个无法破坏操作系统的受限执行环境中，所以它们都执行失败。而其余三个用户态应用程序都可以正常执行。