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

（说明：写出本次实验的主要实验目的与原因。）

保障系统安全和多应用支持是操作系统的两个核心目标，本章从这两个目标出发，思考如何设计应用程序，并进一步展现了操作系统的一系列新功能:

1. 构造包含操作系统内核和多个应用程序的单一执行程序
2. 通过批处理支持多个程序的自动加载和运行
3. 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
4. 实现特权级的穿越
5. 支持跨特权级的系统调用功能

# 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。）

设计实现被批处理系统逐个加载并运行的应用程序。

实际上，如果应用程序的代码都符合用户态特权级的约束，那它完全可以正常在用户态中运行；但如果应用程序执行特权指令或非法操作（如执行非法指令，访问一个非法的地址等），那会产生异常，并导致程序退出。

# 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:** | lib.rs |
| **主要功能:** | 链接操作 |
| **源代码:**   1. #![feature(linkage)] 2. #[no\_mangle] 3. #[link\_section = ".text.entry"]//第 2 行使用 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 4. pub extern "C" fn \_start() -> ! { 5. clear\_bss();//手动清空需要零初始化的 .bss 段 6. exit(main());//调用 main 函数得到一个类型为 i32 的返回值 7. panic!("unreachable after sys\_exit!");}//调用 main 函数得到一个类型为 i32 的返回值 | |

表 3-2 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/syscall.rs |
| **主要功能:** | 参数/返回值绑定和 ecall 指令的插入 |
| **源代码:**   1. // user/src/syscall.rsuse core::arch::asm;fn syscall(id: usize, args: [usize; 3]) -> isize { 2. let mut ret: isize; 3. unsafe { 4. asm!( 5. "ecall", 6. inlateout("x10") args[0] => ret, 7. in("x11") args[1], 8. in("x12") args[2], 9. in("x17") id 10. ); 11. } 12. ret} | |

表 3-3 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/sbi.rs |
| **主要功能:** |  |
| **源代码:**   1. // os/src/sbi.rs 2. use core::arch::asm; 3. #[inline(always)] 4. fn sbi\_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize { 5. let mut ret; 6. unsafe { 7. asm!( 8. "ecall", 9. inlateout("x10") arg0 => ret, 10. in("x11") arg1, 11. in("x12") arg2, 12. in("x17") which, 13. ); 14. } 15. ret} | |

表 3-4 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/syscall.rs |
| **主要功能:** | 系统调用 |
| **源代码:**   1. // user/src/syscall.rs 2. const SYSCALL\_WRITE: usize = 64; 3. const SYSCALL\_EXIT: usize = 93; 4. pub fn sys\_write(fd: usize, buffer: &[u8]) -> isize { 5. syscall(SYSCALL\_WRITE, [fd, buffer.as\_ptr() as usize, buffer.len()]) 6. } 7. pub fn sys\_exit(xstate: i32) -> isize { 8. syscall(SYSCALL\_EXIT, [xstate as usize, 0, 0]) 9. } | |

表 3-5 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/console.rs |
| **主要功能:** | 标准输出 |
| **源代码:**   1. // user/src/console.rs 2. const STDOUT: usize = 1; 3. impl Write for Stdout { 4. fn write\_str(&mut self, s: &str) -> fmt::Result { 5. write(STDOUT, s.as\_bytes()); 6. Ok(()) 7. }} | |

表 3-6 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/bin/03priv\_inst.rs |
| **主要功能:** | 尝试在用户态执行内核态的特权指令sret |
| **源代码:**   1. // user/src/bin/03priv\_inst.rs 2. use core::arch::asm; 3. #[no\_mangle] 4. fn main() -> i32 { 5. println!("Try to execute privileged instruction in U Mode"); 6. println!("Kernel should kill this application!"); 7. unsafe { 8. asm!("sret"); 9. } 10. 0} | |

表 3-7 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/bin/04priv\_csr.rs |
| **主要功能:** | 试图在用户态修改内核态 CSR sstatus |
| **源代码:**   1. // user/src/bin/04priv\_csr.rs 2. use riscv::register::sstatus::{self, SPP}; 3. #[no\_mangle] 4. fn main() -> i32 { 5. println!("Try to access privileged CSR in U Mode"); 6. println!("Kernel should kill this application!"); 7. unsafe { 8. sstatus::set\_spp(SPP::User); 9. } 10. 0} | |

表 3-8 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/link\_app.S |
| **主要功能:** | 汇编代码 |
| **源代码:**   1. # os/src/link\_app.S 2. .align 3 3. .section .data 4. .global \_num\_app 5. \_num\_app: 6. .quad 5 7. .quad app\_0\_start 8. .quad app\_1\_start 9. .quad app\_2\_start 10. .quad app\_3\_start 11. .quad app\_4\_start 12. .quad app\_4\_end 13. .section .data 14. .global app\_0\_start 15. .global app\_0\_end 16. app\_0\_start: 17. .incbin "../user/target/riscv64gc-unknown-none-elf/release/00hello\_world.bin" 18. app\_0\_end: 19. .section .data 20. .global app\_1\_start 21. .global app\_1\_end 22. app\_1\_start: 23. .incbin "../user/target/riscv64gc-unknown-none-elf/release/01store\_fault.bin" 24. app\_1\_end: 25. .section .data 26. .global app\_2\_start 27. .global app\_2\_end 28. app\_2\_start: 29. .incbin "../user/target/riscv64gc-unknown-none-elf/release/02power.bin" 30. app\_2\_end: 31. .section .data 32. .global app\_3\_start 33. .global app\_3\_end 34. app\_3\_start: 35. .incbin "../user/target/riscv64gc-unknown-none-elf/release/03priv\_inst.bin" 36. app\_3\_end: 37. .section .data 38. .global app\_4\_start 39. .global app\_4\_end 40. app\_4\_start: 41. .incbin "../user/target/riscv64gc-unknown-none-elf/release/04priv\_csr.bin" 42. app\_4\_end: | |

表 3-9 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/batch.rs |
| **主要功能:** | 结构体定义 |
| **源代码:**   1. // os/src/batch.rs 2. struct AppManager { 3. num\_app: usize, 4. current\_app: usize, 5. app\_start: [usize; MAX\_APP\_NUM + 1], 6. } | |

表 3-10 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/sync/up.rs |
| **主要功能:** | 允许我们在 单核 上安全使用可变全局变量。  UPSafeCell 对于 RefCell 简单进行封装，它和 RefCell 一样提供内部可变性和运行时借用检查 |
| **源代码:**   1. // os/src/sync/up.rs 2. pub struct UPSafeCell<T> { 3. /// inner data 4. inner: RefCell<T>, 5. } 6. unsafe impl<T> Sync for UPSafeCell<T> {} 7. impl<T> UPSafeCell<T> { 8. /// User is responsible to guarantee that inner struct is only used in 9. /// uniprocessor. 10. pub unsafe fn new(value: T) -> Self { 11. Self { inner: RefCell::new(value) } 12. } 13. /// Exclusive access inner data in UPSafeCell. Panic if the data has been borrowed. 14. pub fn exclusive\_access(&self) -> RefMut<'\_, T> { 15. self.inner.borrow\_mut() 16. }   **}** | |

表 3-11 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | // os/src/batch.rs |
| **主要功能:** | 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。 |
| **源代码:**   1. // os/src/batch.rs 2. lazy\_static! { 3. static ref APP\_MANAGER: UPSafeCell<AppManager> = unsafe { UPSafeCell::new({ 4. extern "C" { fn \_num\_app(); } 5. let num\_app\_ptr = \_num\_app as usize as \*const usize; 6. let num\_app = num\_app\_ptr.read\_volatile(); 7. let mut app\_start: [usize; MAX\_APP\_NUM + 1] = [0; MAX\_APP\_NUM + 1]; 8. let app\_start\_raw: &[usize] = core::slice::from\_raw\_parts( 9. num\_app\_ptr.add(1), num\_app + 1 10. ); 11. app\_start[..=num\_app].copy\_from\_slice(app\_start\_raw); 12. AppManager { 13. num\_app, 14. current\_app: 0, 15. app\_start, 16. } 17. })};} | |

表 3-12 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | AppManager |
| **主要功能:** | 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 |
| **源代码:**   1. unsafe fn load\_app(&self, app\_id: usize) { 2. if app\_id >= self.num\_app { 3. panic!("All applications completed!"); 4. } 5. println!("[kernel] Loading app\_{}", app\_id); 6. // clear icache 7. asm!("fence.i"); 8. // clear app area 9. core::slice::from\_raw\_parts\_mut( 10. APP\_BASE\_ADDRESS as \*mut u8, 11. APP\_SIZE\_LIMIT 12. ).fill(0); 13. let app\_src = core::slice::from\_raw\_parts( 14. self.app\_start[app\_id] as \*const u8, 15. self.app\_start[app\_id + 1] - self.app\_start[app\_id] 16. ); 17. let app\_dst = core::slice::from\_raw\_parts\_mut( 18. APP\_BASE\_ADDRESS as \*mut u8, 19. app\_src.len() 20. ); 21. app\_dst.copy\_from\_slice(app\_src);} | |

表 3-13 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/batch.rs |
| **主要功能:** | 声明两个类型 KernelStack 和 UserStack 分别表示用户栈和内核栈 |
| **源代码:**  // os/src/batch.rs  const USER\_STACK\_SIZE: usize = 4096 \* 2;  const KERNEL\_STACK\_SIZE: usize = 4096 \* 2;  #[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] }; | |

表 3-14 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/trap/context.rs |
| **主要功能:** | TrapContext类型定义 |
| **源代码:**   1. // os/src/trap/context.rs 2. #[repr(C)]pub struct TrapContext { 3. pub x: [usize; 32], 4. pub sstatus: Sstatus, 5. pub sepc: usize,} | |

表 3-15 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/trap/mod.rs |
| **主要功能:** | 需要修改 stvec 寄存器来指向正确的 Trap 处理入口点。 |
| **源代码:**   1. // os/src/trap/mod.rs 2. global\_asm!(include\_str!("trap.S")); 3. pub fn init() { 4. extern "C" { fn \_\_alltraps(); } 5. unsafe { 6. stvec::write(\_\_alltraps as usize, TrapMode::Direct); 7. } 8. } | |

表 3-16 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/trap/trap.S |
| **主要功能:** | 保存 Trap 上下文的\_\_alltraps的实现  第 7 行我们使用.align将\_\_alltraps的地址 4 字节对齐  第 9 行的csrrw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。  第 12 行，我们准备在内核栈上保存 Trap 上下文，于是预先分配 34×8 字节的栈帧  第 25~28 行，我们将 CSR sstatus 和 sepc 的值分别读到寄存器 t0 和 t1 中然后保存到内核栈对应的位置上。  第 30~31 行专门处理 sp 的问题。  第 33 行令 a0←sp，让寄存器 a0 指向内核栈的栈指针也就是我们刚刚保存的 Trap 上下文的地址 |
| **源代码:**   1. # os/src/trap/trap.S 2. .macro SAVE\_GP n 3. sd x\n, \n\*8(sp) 4. .endm 5. .align 2 6. \_\_alltraps: 7. csrrw sp, sscratch, sp 8. # now sp->kernel stack, sscratch->user stack 9. # allocate a TrapContext on kernel stack 10. addi sp, sp, -34\*8 11. # save general-purpose registers 12. sd x1, 1\*8(sp) 13. # skip sp(x2), we will save it later 14. sd x3, 3\*8(sp) 15. # skip tp(x4), application does not use it 16. # save x5~x31 17. .set n, 5 18. .rept 27 19. SAVE\_GP %n 20. .set n, n+1 21. .endr 22. # we can use t0/t1/t2 freely, because they were saved on kernel stack 23. csrr t0, sstatus 24. csrr t1, sepc 25. sd t0, 32\*8(sp) 26. sd t1, 33\*8(sp) 27. # read user stack from sscratch and save it on the kernel stack 28. csrr t2, sscratch 29. sd t2, 2\*8(sp) 30. # set input argument of trap\_handler(cx: &mut TrapContext) 31. mv a0, sp 32. call trap\_handler | |

表 3-17 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/trap/mod.rs |
| **主要功能:** | 在trap\_handler 函数中完成分发和处理  第 4 行声明返回值为 &mut TrapContext 并在第 25 行实际将传入的Trap 上下文 cx 原样返回  第 7 行根据 scause 寄存器所保存的 Trap 的原因进行分发处理。  第 8~11 行，发现触发 Trap 的原因是来自 U 特权级的 Environment Call，也就是系统调用  第 12~20 行，分别处理应用程序出现访存错误和非法指令错误的情形。  第 21 行开始，当遇到目前还不支持的 Trap 类型的时候，我们的批处理操作系统整个 panic 报错退出。 |
| **源代码:**   1. // os/src/trap/mod.rs 2. #[no\_mangle] 3. pub fn trap\_handler(cx: &mut TrapContext) -> &mut TrapContext { 4. let scause = scause::read(); 5. let stval = stval::read(); 6. match scause.cause() { 7. Trap::Exception(Exception::UserEnvCall) => { 8. cx.sepc += 4; 9. cx.x[10] = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]) as usize; 10. } 11. Trap::Exception(Exception::StoreFault) | 12. Trap::Exception(Exception::StorePageFault) => { 13. println!("[kernel] PageFault in application, kernel killed it."); 14. run\_next\_app(); 15. } 16. Trap::Exception(Exception::IllegalInstruction) => { 17. println!("[kernel] IllegalInstruction in application, kernel killed it."); 18. run\_next\_app(); 19. } 20. \_ => { 21. panic!("Unsupported trap {:?}, stval = {:#x}!", scause.cause(), stval); 22. } 23. } 24. cx 25. } | |

表 3-18 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | # os/Cargo.toml |
| **主要功能:** | 引入 riscv 库 |
| **源代码:**   1. # os/Cargo.toml 2. [dependencies] 3. riscv = { git = "https://github.com/bjtu-os/riscv", features = ["inline-asm"] } | |

表 3-19 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/syscall/mod.rs |
| **主要功能:** | 根据 syscall ID 分发到具体的处理函数 |
| **源代码:**   1. // os/src/syscall/mod.rs 2. pub fn syscall(syscall\_id: usize, args: [usize; 3]) -> isize { 3. match syscall\_id { 4. SYSCALL\_WRITE => sys\_write(args[0], args[1] as \*const u8, args[2]), 5. SYSCALL\_EXIT => sys\_exit(args[0] as i32), 6. \_ => panic!("Unsupported syscall\_id: {}", syscall\_id), 7. } 8. } | |

表 3-20 源代码及注释

|  |  |
| --- | --- |
| **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:**   1. // os/src/syscall/fs.rs 2. const FD\_STDOUT: usize = 1; 3. pub fn sys\_write(fd: usize, buf: \*const u8, len: usize) -> isize { 4. match fd { 5. FD\_STDOUT => { 6. let slice = unsafe { core::slice::from\_raw\_parts(buf, len) }; 7. let str = core::str::from\_utf8(slice).unwrap(); 8. print!("{}", str); 9. len as isize 10. }, 11. \_ => { 12. panic!("Unsupported fd in sys\_write!"); 13. } 14. } 15. } 16. // os/src/syscall/process.rs 17. pub fn sys\_exit(xstate: i32) -> ! { 18. println!("[kernel] Application exited with code {}", xstate); 19. run\_next\_app() 20. } | |

表 3-21 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/trap/context.rs |
| **主要功能:** | 内核栈上压入一个为启动应用程序而特殊构造的 Trap 上下文，再通过 \_\_restore 函数 |
| **源代码:**   1. // os/src/trap/context.rs 2. impl TrapContext { 3. pub fn set\_sp(&mut self, sp: usize) { self.x[2] = sp; } 4. pub fn app\_init\_context(entry: usize, sp: usize) -> Self { 5. let mut sstatus = sstatus::read(); 6. sstatus.set\_spp(SPP::User); 7. let mut cx = Self { 8. x: [0; 32], 9. sstatus, 10. sepc: entry, 11. }; 12. cx.set\_sp(sp); 13. cx 14. } 15. } | |

表 3-22 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/batch.rs |
| **主要功能:** | 内核栈上压入一个 Trap 上下文，其 sepc 是应用程序入口地址 0x80400000 ，其 sp 寄存器指向用户栈，其 sstatus 的 SPP 字段被设置为 User |
| **源代码:**   1. // os/src/batch.rs 2. pub fn run\_next\_app() -> ! { 3. let mut app\_manager = APP\_MANAGER.exclusive\_access(); 4. let current\_app = app\_manager.get\_current\_app(); 5. unsafe { 6. app\_manager.load\_app(current\_app); 7. } 8. app\_manager.move\_to\_next\_app(); 9. drop(app\_manager); 10. // before this we have to drop local variables related to resources manually 11. // and release the resources 12. extern "C" { fn \_\_restore(cx\_addr: usize); } 13. unsafe { 14. \_\_restore(KERNEL\_STACK.push\_context( 15. TrapContext::app\_init\_context(APP\_BASE\_ADDRESS, USER\_STACK.get\_sp()) 16. ) as \*const \_ as usize); 17. } 18. panic!("Unreachable in batch::run\_current\_app!"); 19. } | |

# 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。）

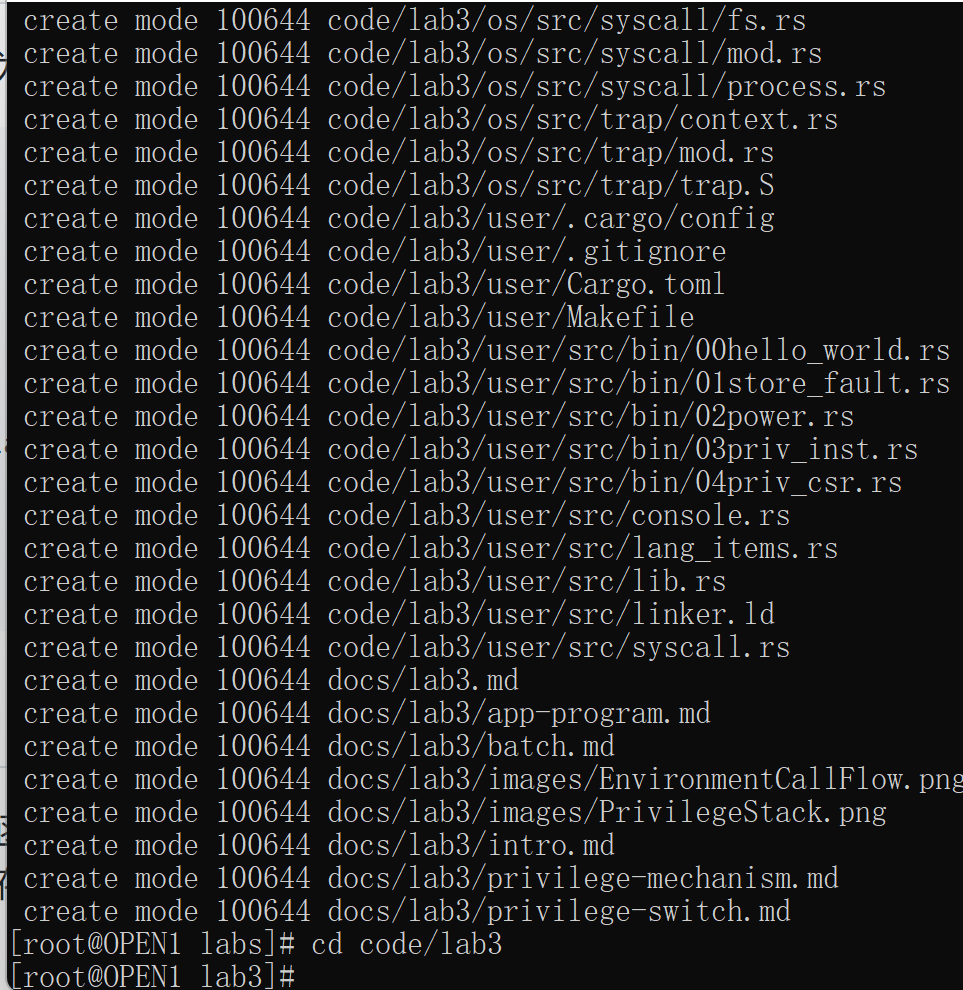


图4-1 使用git pull项目

使用 qemu-system-riscv64 –version 查找确认本机qemu环境是否安装成功。

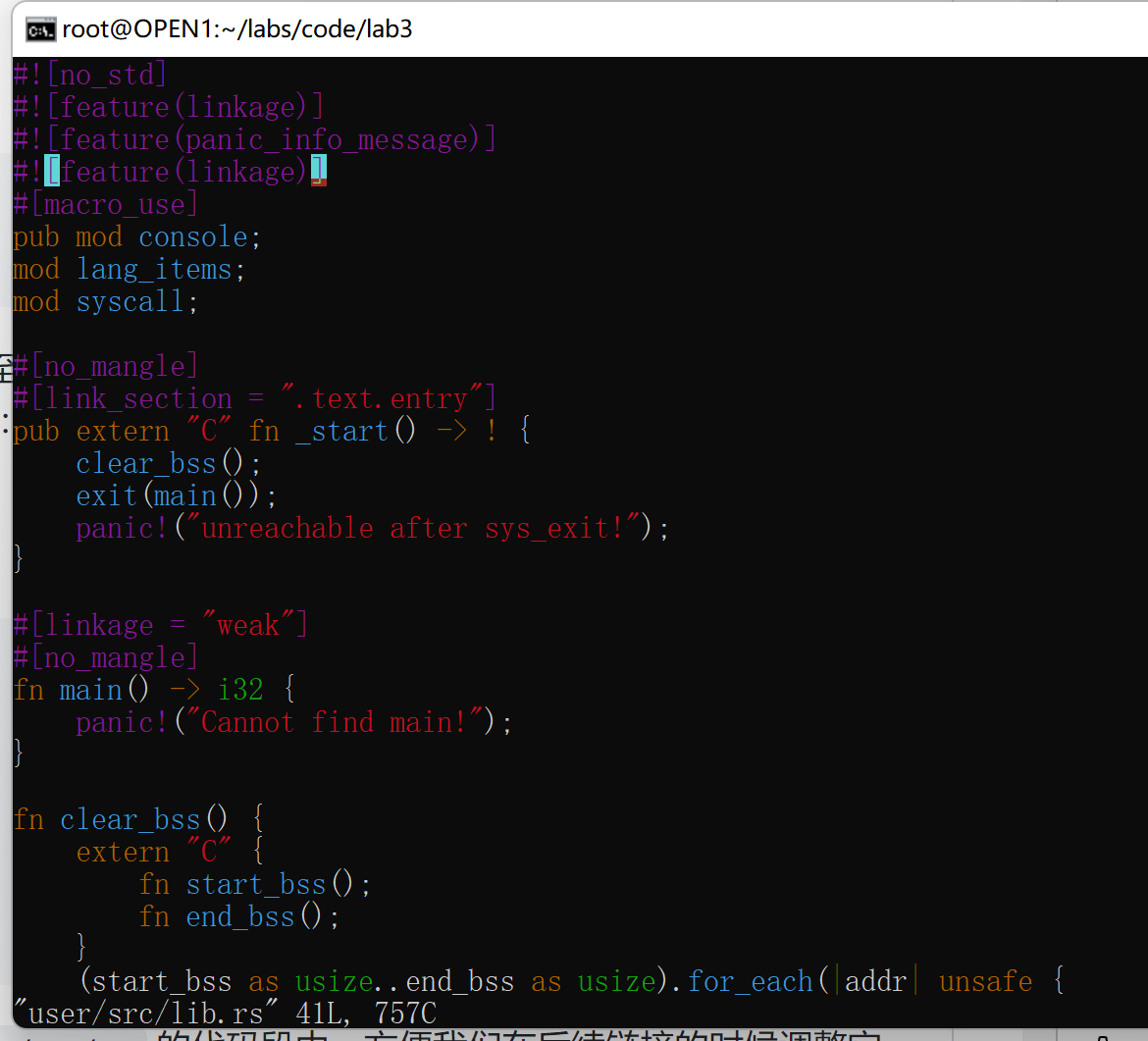


图4-2 查看lib.rs文件的内容

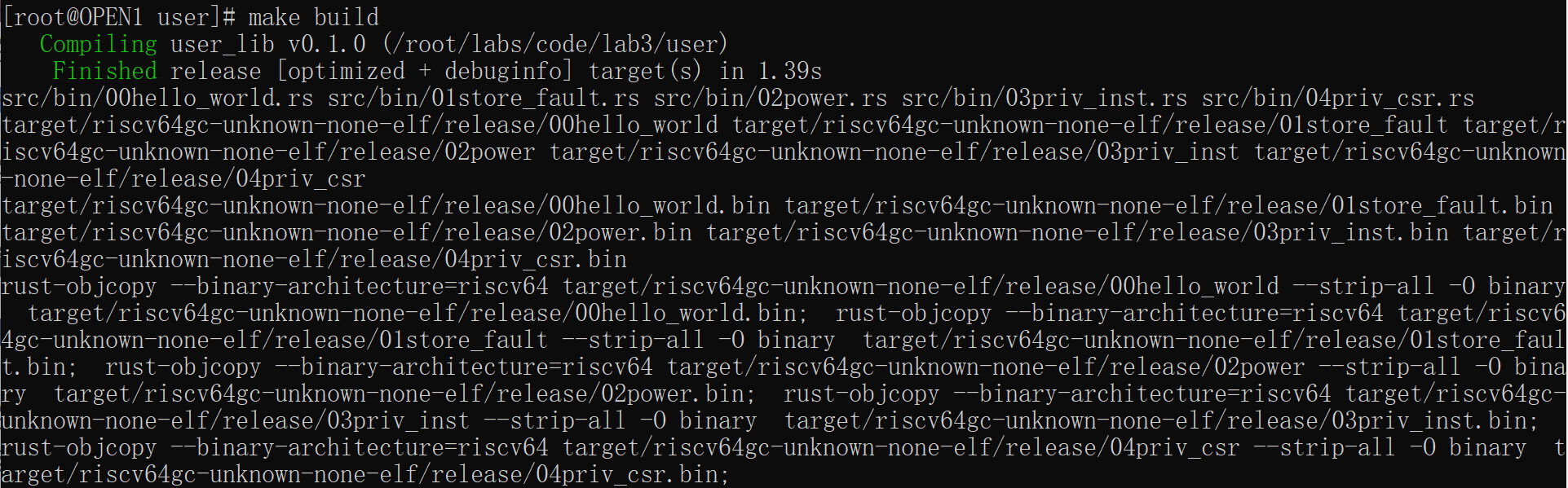


图4-3成功执行

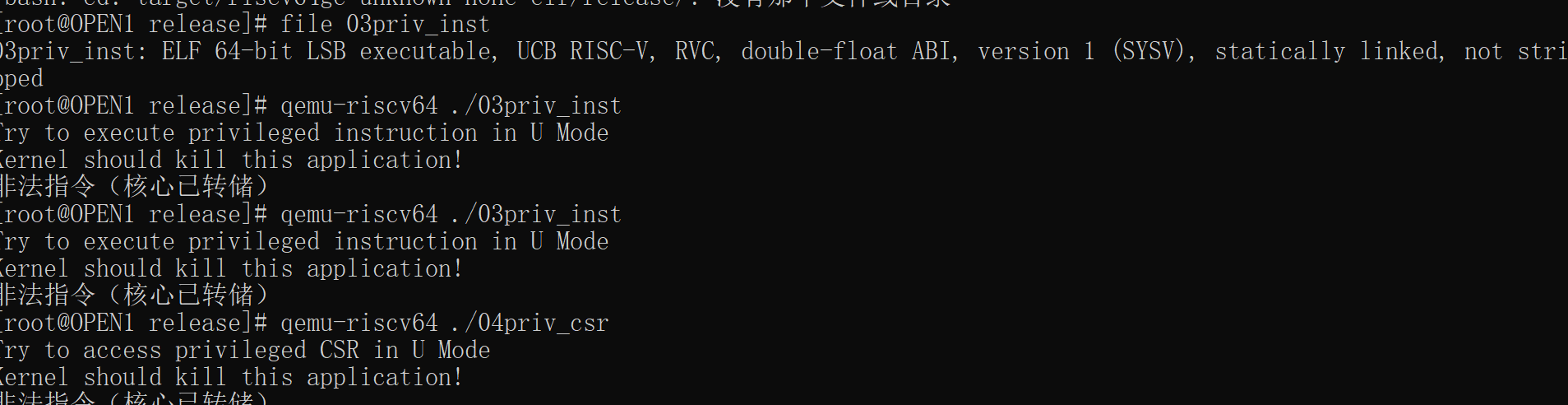


图4-4尝试在用户态模拟器执行

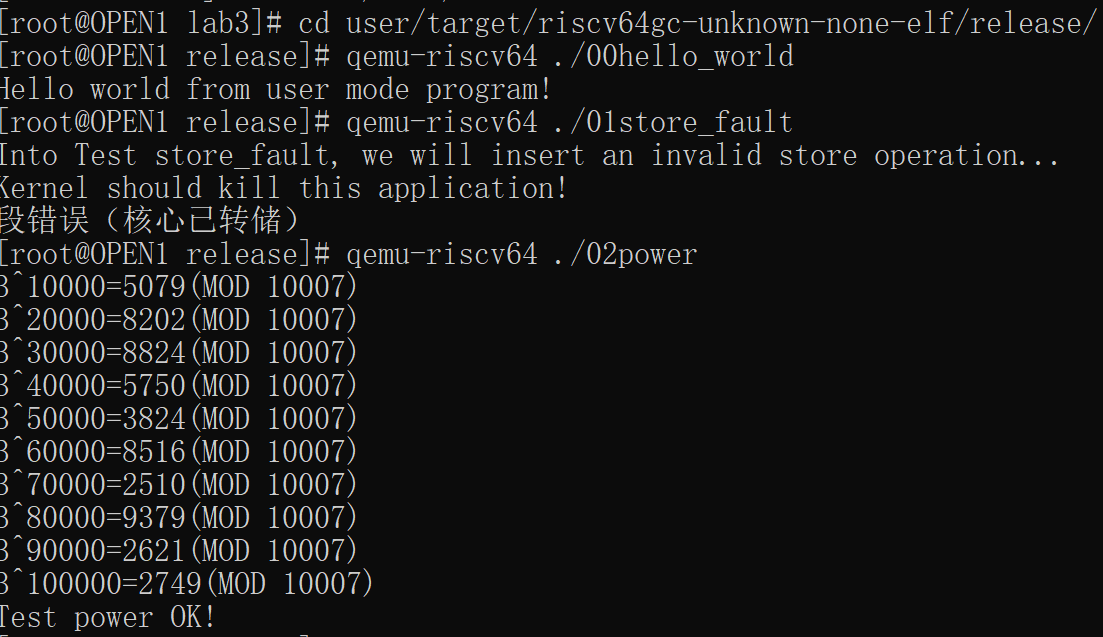


图4-5 对于一般的用户态应用程序，在 qemu-riscv64 模拟器

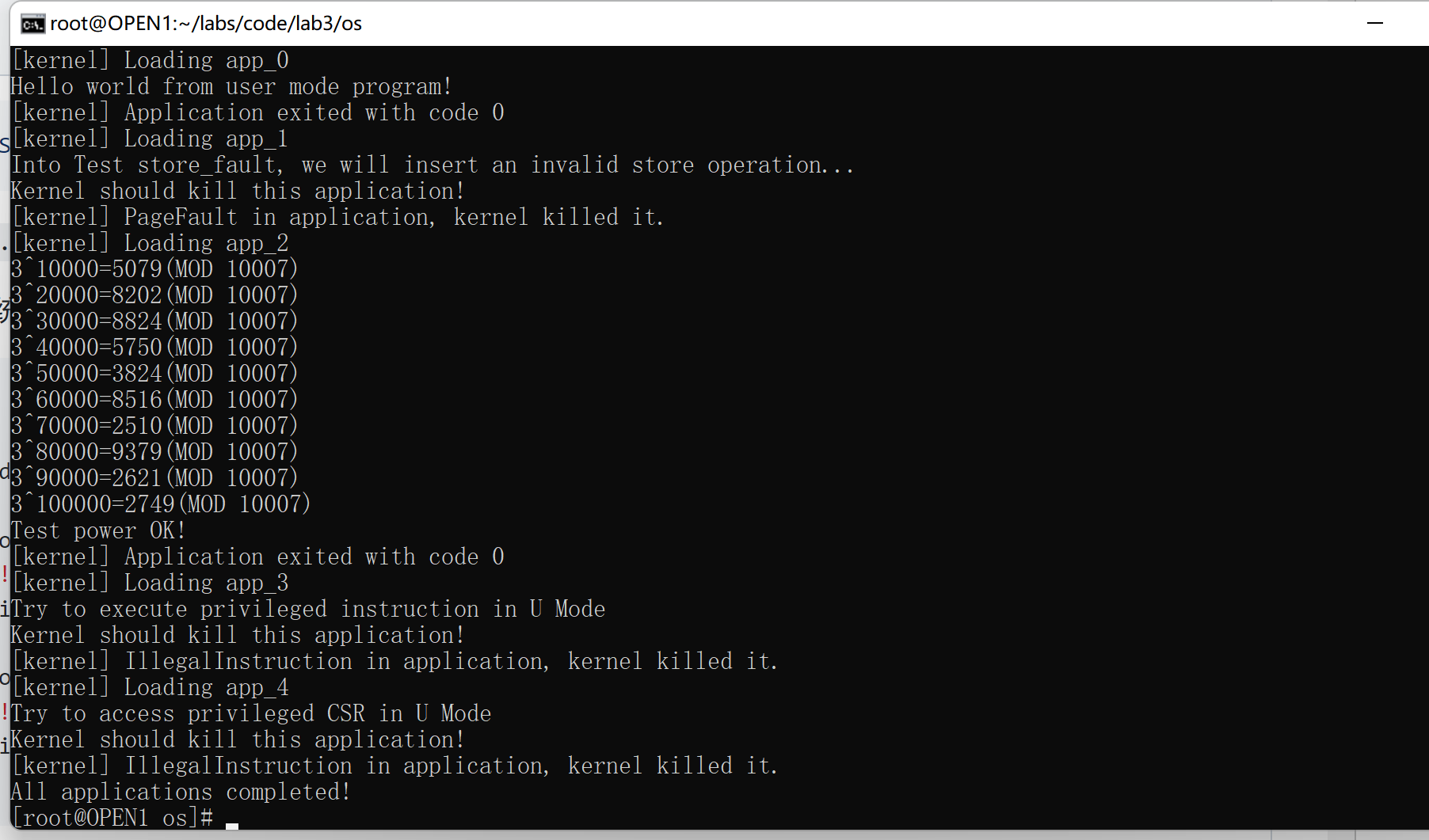


图4-6 在 os 目录下，构建并运行整个系统：

我们使用make run让系统运行的过程中，这个汇编代码link\_app.S就生成了。

# 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。）

批处理系统的核心思想是：将多个程序打包到一起输入计算机。而当一个程序运行结束后，计算机会自动加载下一个程序到内存并开始执行。当软件有了代替操作员的管理和操作能力后，便开始形成真正意义上的操作系统了。

人们希望一个应用程序的错误不要影响到其它应用程序、操作系统和整个计算机系统。这就需要操作系统能够终止出错的应用程序，转而运行下一个应用程序。这种保护计算机系统不受有意或无意出错的程序破坏的机制被称为特权级(Privilege) 机制。

为确保操作系统的安全，对应用程序而言，需要限制的主要有两个方面：

1. 应用程序不能访问任意的地址空间（这个在后面章节会进一步讲解，本章不会涉及）
2. 应用程序不能执行某些可能破坏计算机系统的指令（本章的重点）

确保应用程序能够得到操作系统的服务，即应用程序和操作系统还需要有交互的手段。

为了实现这样的特权级机制，需要进行软硬件协同设计。一个比较简洁的方法就是，处理器设置两个不同安全等级的执行环境：用户态特权级的执行环境和内核态特权级的执行环境。且明确指出可能破坏计算机系统的内核态特权级指令子集，规定内核态特权级指令子集中的指令只能在内核态特权级的执行环境中执行。处理器在执行指令前会进行特权级安全检查，如果在用户态执行环境中执行这些内核态特权级指令，会产生异常。

RISC-V 架构中，只有 M 模式是必须实现的，剩下的特权级则可以根据跑在 CPU 上应用的实际需求进行调整。