> OS-Lab3 批处理与特权级 21301117 周书扬
>
> 本实验的主要目的是实现一个简单的批处理操作系统并理解特权级的概念。

一. 实验步骤

1. 设计和实现应用程序

1.1 创建user目录

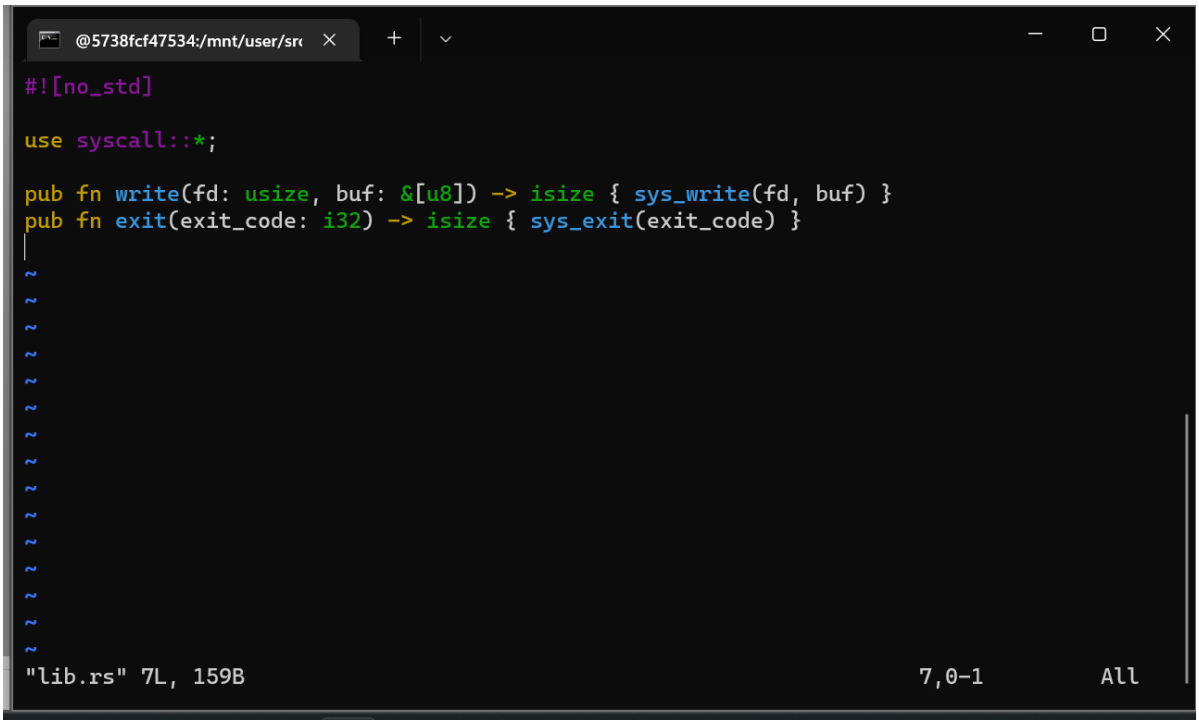
```
@5738fcf47534:/mnt
C:\Users\Zoransy>docker attach 5738
[root@5738fcf47534 /]# cd os
bash: cd: os: No such file or directory
[root@5738fcf47534 /]# cd mnt
[root@5738fcf47534 mnt]# ll
total 0
drwxrwxrwx 1 root root 4096 Nov 14 15:52 Documents
drwxrwxrwx 1 root root 4096 Nov 14 16:36 bootloader
drwxr-xr-x 1 root root 4096 Nov 14 15:45 os
[root@5738fcf47534 mnt]# cargo new user_lib
     Created binary (application) `user_lib` package
[root@5738fcf47534 mnt]# mv userlib user
mv: cannot stat 'userlib': No such file or directory
[root@5738fcf47534 mnt]# mv user_lib user
[root@5738fcf47534 mnt]# cd user
[root@5738fcf47534 user]# ll
total 0
-rw-r--r-- 1 root root  177 Nov 29 15:27 Cargo.toml
drwxr-xr-x 1 root root 4096 Nov 29 15:27 src
[root@5738fcf47534 user]# cd ..
[root@5738fcf47534 mnt]# rm user/src/main.rs
rm: remove regular file 'user/src/main.rs'?
[root@5738fcf47534 mnt]#
```

1.2 实现应用程序与系统约定的两个系统调用sys_write和sys_exit

- 实现syscall.rs

```
@5738fcf47534:/mnt/user/src
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",
             in("x10") args[0],
             in("x11") args[1],
             in("x12") args[2],
             in("x17") id,
             lateout("x10") ret
        );
    }
    ret
}

pub fn sys_write(fd: usize, buffer: &[u8]) -> isize {
    syscall(SYSCALL_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
}
```

- 进一步封装lib.rs

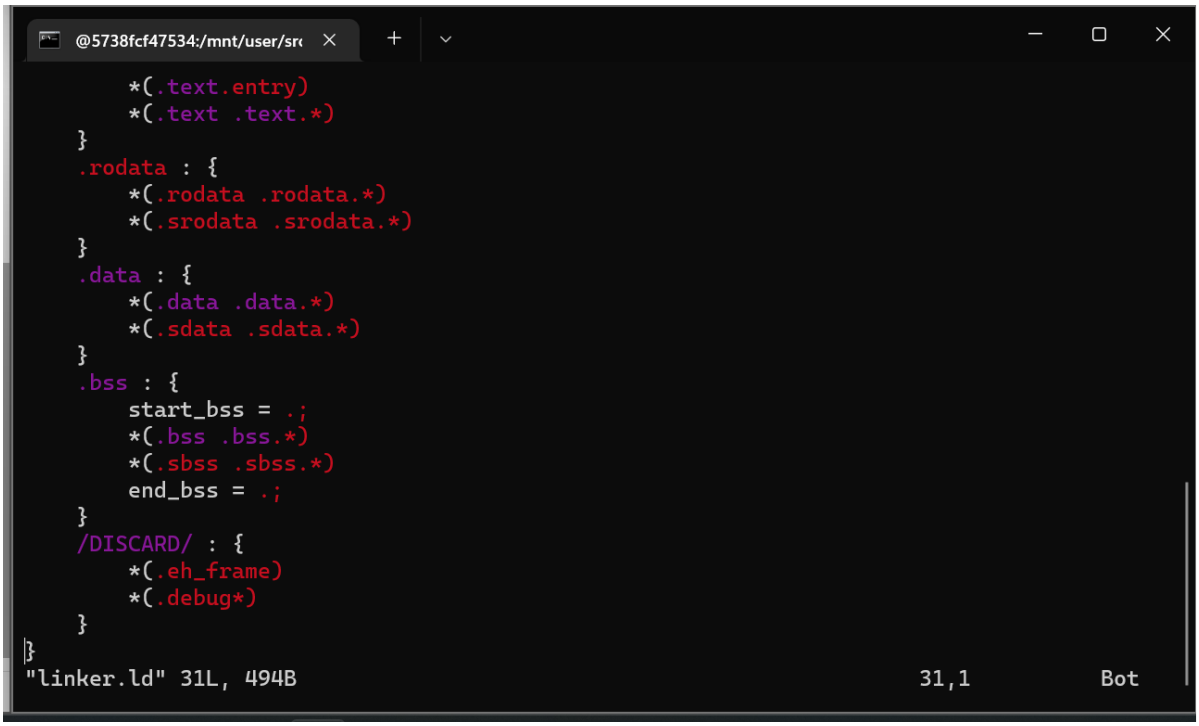

@5738fcf47534:/mnt/user/sr
#![no_std]
use syscall::*;
pub fn write(fd: usize, buf: &[u8]) -> isize { sys_write(fd, buf) }
pub fn exit(exit_code: i32) -> isize { sys_exit(exit_code) }
"lib.rs" 7L, 159B
7,0-1
All

1.1.2 实现格式化输出

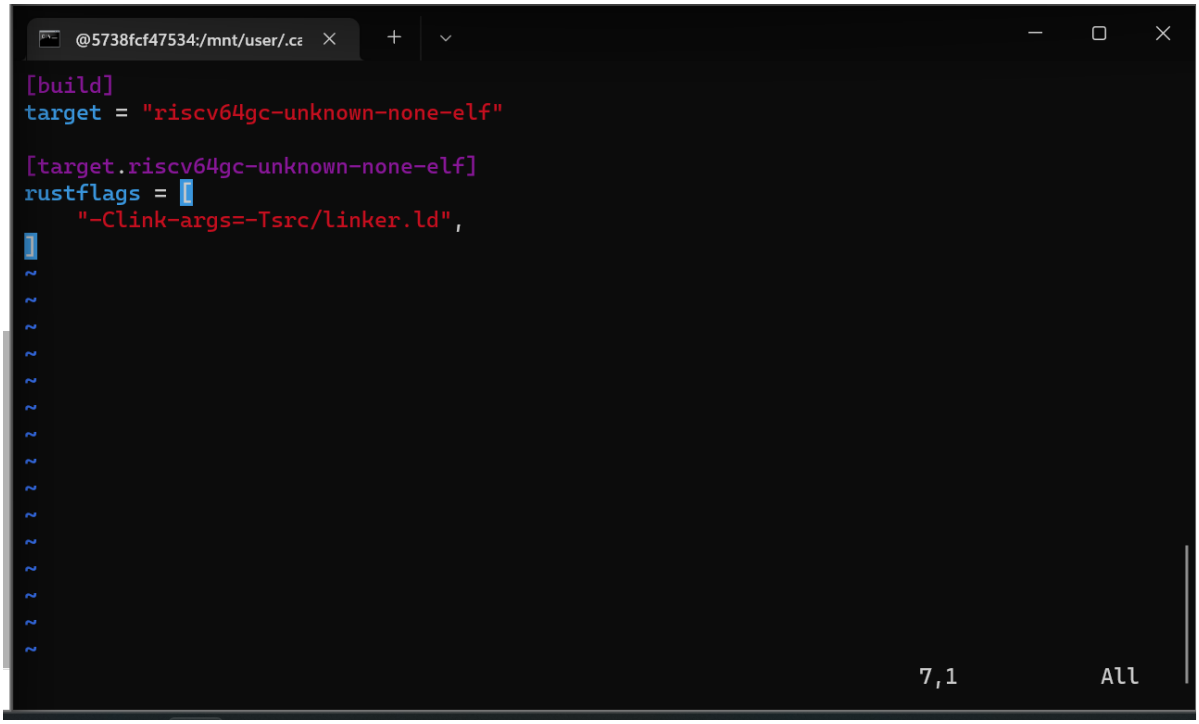

@5738fcf47534:/mnt/user/sr
use core::fmt::{self, Write};
use super::write;
const STDOUT: usize = 1;
struct Stdout;
impl Write for Stdout {
fn write_str(&mut self, s: &str) -> fmt::Result {
write(STDOUT, s.as_bytes());
Ok(())
}
}
pub fn print(args: fmt::Arguments) {
Stdout.write_fmt(args).unwrap();
}
#[macro_export]
macro_rules! print {
($fmt: literal $(, $($arg: tt)+)?) => {
$crate::console::print(format_args!($fmt $(, $($arg)+)?));
"console.rs" 25L, 464B
1,1
Top

1.1.3 实现语义支持

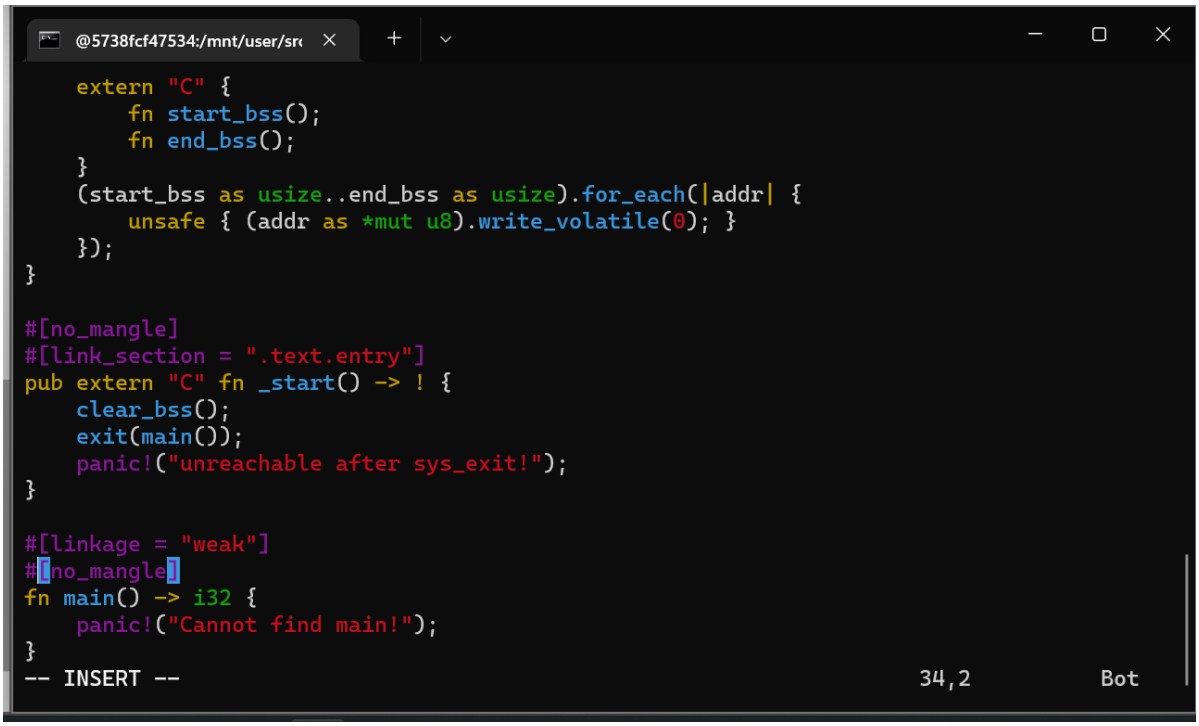
1.1.4 将应用程序的起始物理地址调整为 0x80400000



1.1.5 增加配置文件使用linker.ld文件



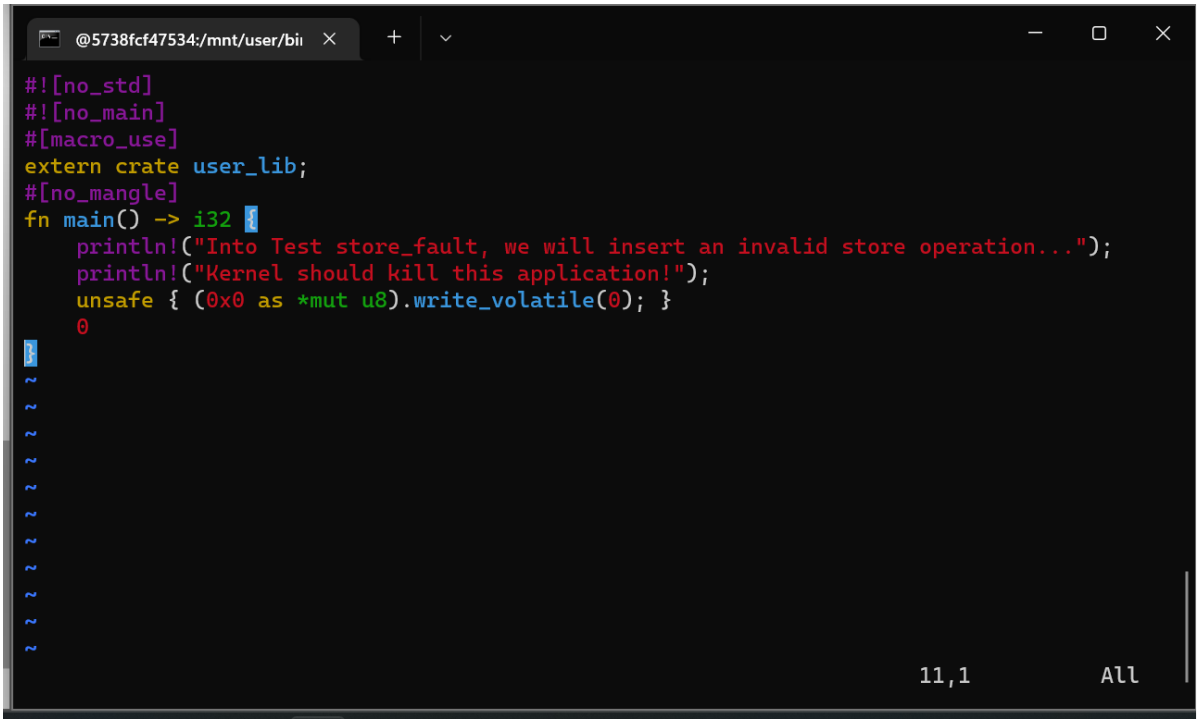
1.1.6 定义用户库入口点，形成用户运行时库lib.rs



1.1.7 基于模板实现应用程序

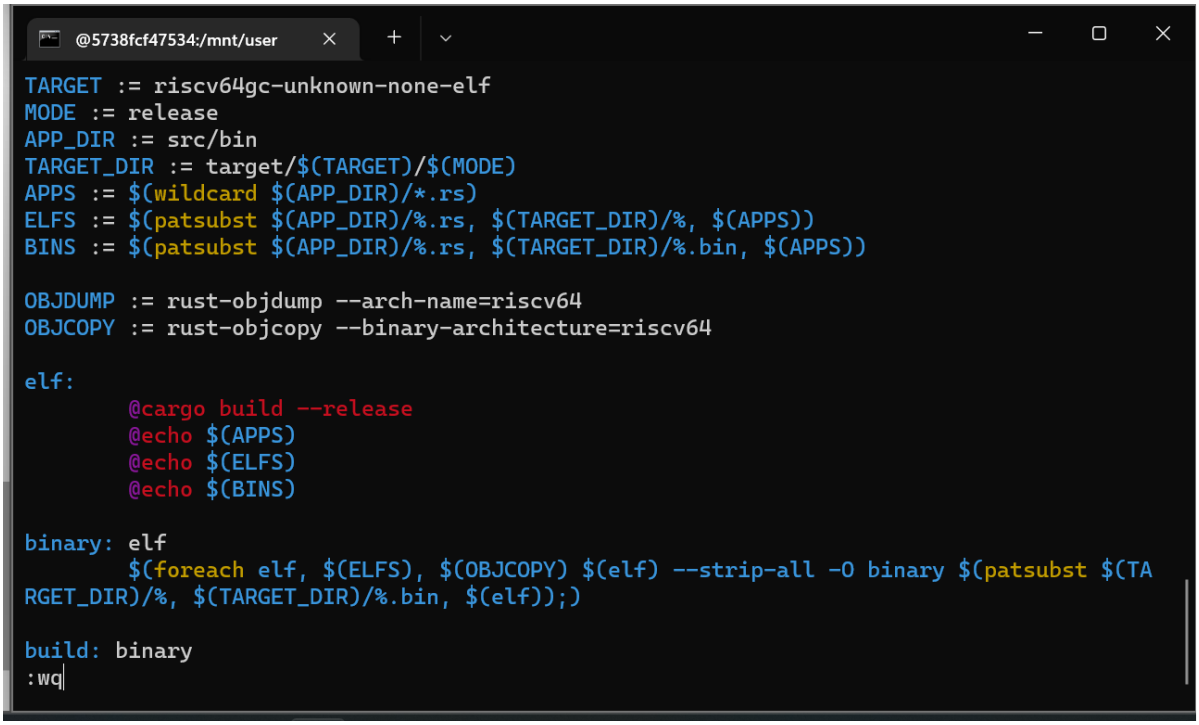
- 00hello_world.rs

- 01store_fauld.rs



- 02power.rs

1.1.8 实现Makefile文件进行编译



- *然后编译失败了，所以重新看了一遍之前的步骤，发现删除main.rs的时候要y加回车所以没删掉，删除之后编译通过了

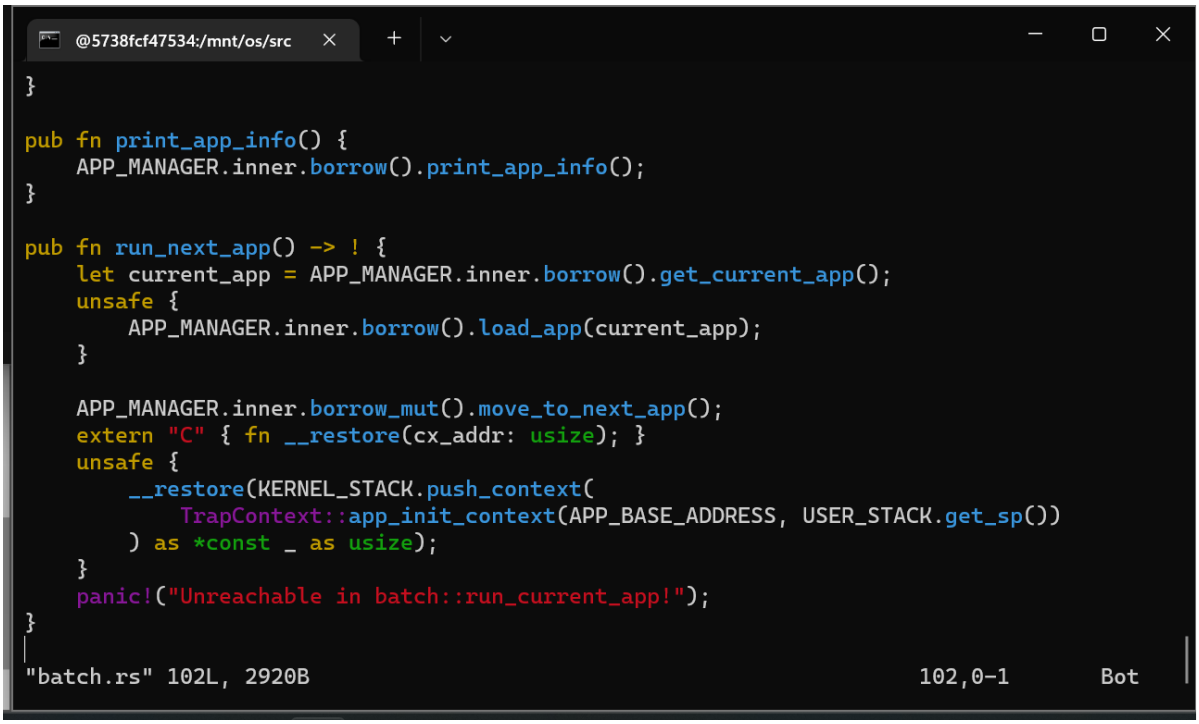
```
@5738fcf47534:/mnt/user
[root@5738fcf47534 user]# ls
Cargo.lock  Cargo.toml  Makefile  src  target
[root@5738fcf47534 user]# ls src
bin  console.rs  lang_items.rs  lib.rs  linker.ld  main.rs  syscall.rs
[root@5738fcf47534 user]# rm src/main.rs
rm: remove regular file 'src/main.rs'? y
[root@5738fcf47534 user]# ls src
bin  console.rs  lang_items.rs  lib.rs  linker.ld  syscall.rs
[root@5738fcf47534 user]# make build
    Finished release [optimized] target(s) in 0.12s
src/bin/00hello_world.rs src/bin/01store_fault.rs src/bin/02power.rs
target/riscv64gc-unknown-none-elf/release/00hello_world target/riscv64gc-unknown-none-e
lf/release/01store_fault target/riscv64gc-unknown-none-elf/release/02power
target/riscv64gc-unknown-none-elf/release/00hello_world.bin target/riscv64gc-unknown-no
ne-elf/release/01store_fault.bin target/riscv64gc-unknown-none-elf/release/02power.bin
rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/00
hello_world --strip-all -O binary  target/riscv64gc-unknown-none-elf/release/00hello_wo
rld.bin;  rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/
release/01store_fault --strip-all -O binary  target/riscv64gc-unknown-none-elf/release/
01store_fault.bin;  rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown
-none-elf/release/02power --strip-all -O binary  target/riscv64gc-unknown-none-elf/rele
ase/02power.bin;
[root@5738fcf47534 user]#
```

2. 链接应用程序到内核

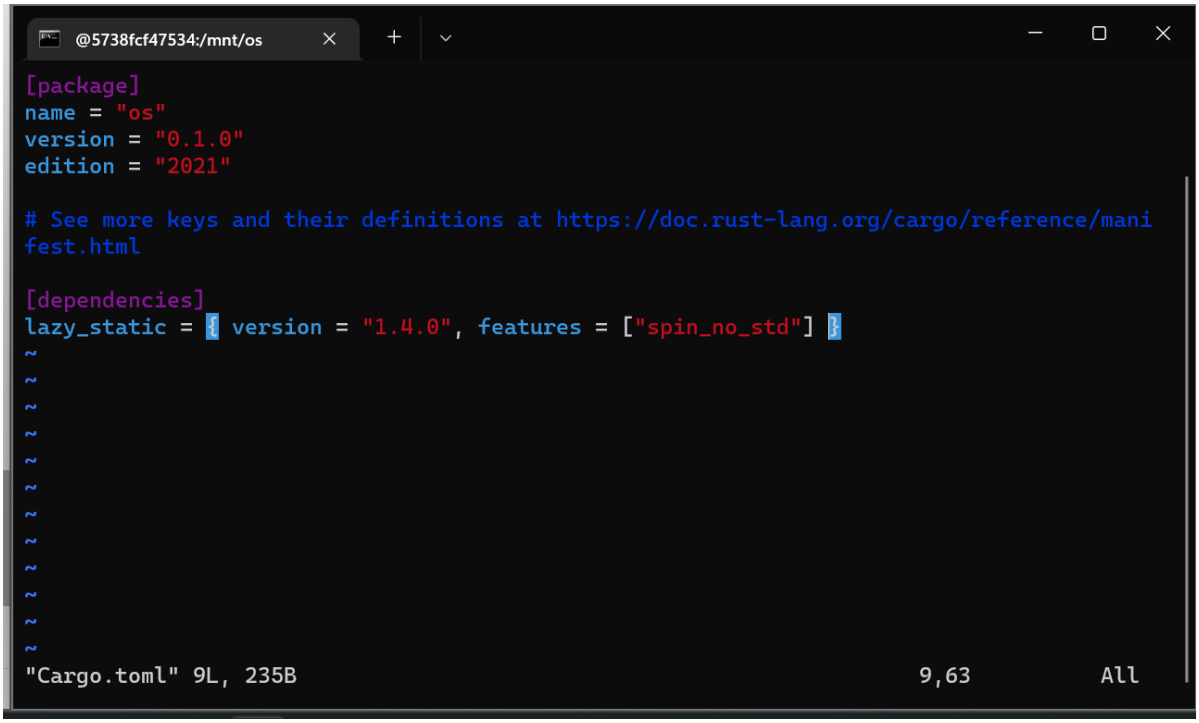
```
@5738fcf47534:/mnt/os
    .section .data
    .global _num_app
_num_app:
    .quad {}"#, apps.len())?;

    for i in 0..apps.len() {
        writeln!(f, r#"    .quad app_{}_start"#, i)?;
    }
    writeln!(f, r#"    .quad app_{}_end"#, apps.len() - 1)?;

    for (idx, app) in apps.iter().enumerate() {
        println!("app_{}: {}", idx, app);
        writeln!(f, r#"
    .section .data
    .global app_{0}_start
    .global app_{0}_end
app_{0}_start:
    .incbin "{2}{1}.bin"
app_{0}_end:"#, idx, app, TARGET_PATH)?;
    }
    Ok(())
}
:wq
```

3. 找到并加载应用程序二进制码

- 实现batch.rs

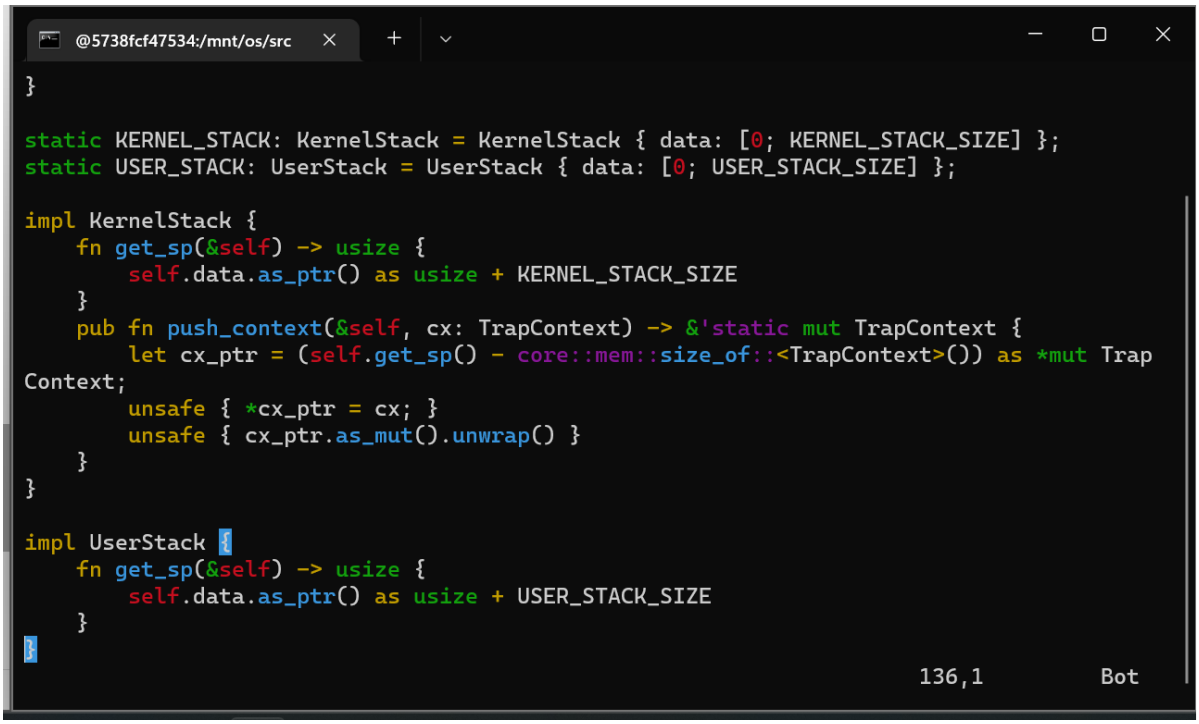


- 增加依赖

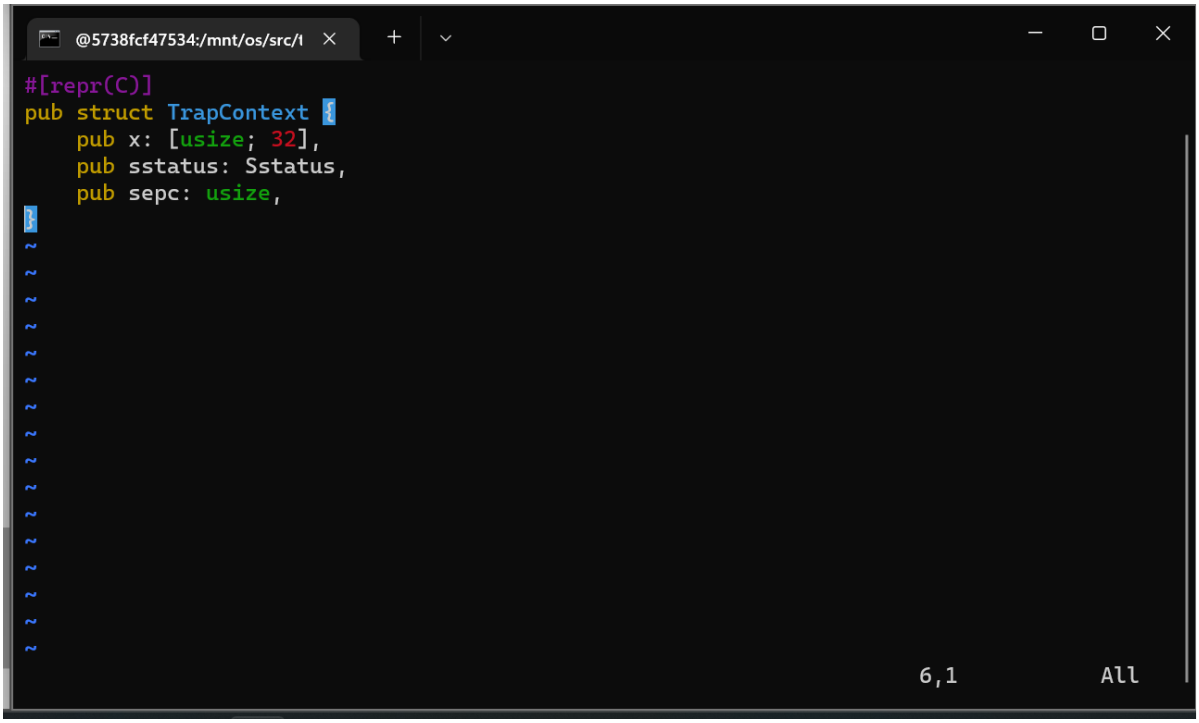


4. 实现用户栈和内核栈

- 增加batch.rs中的实现



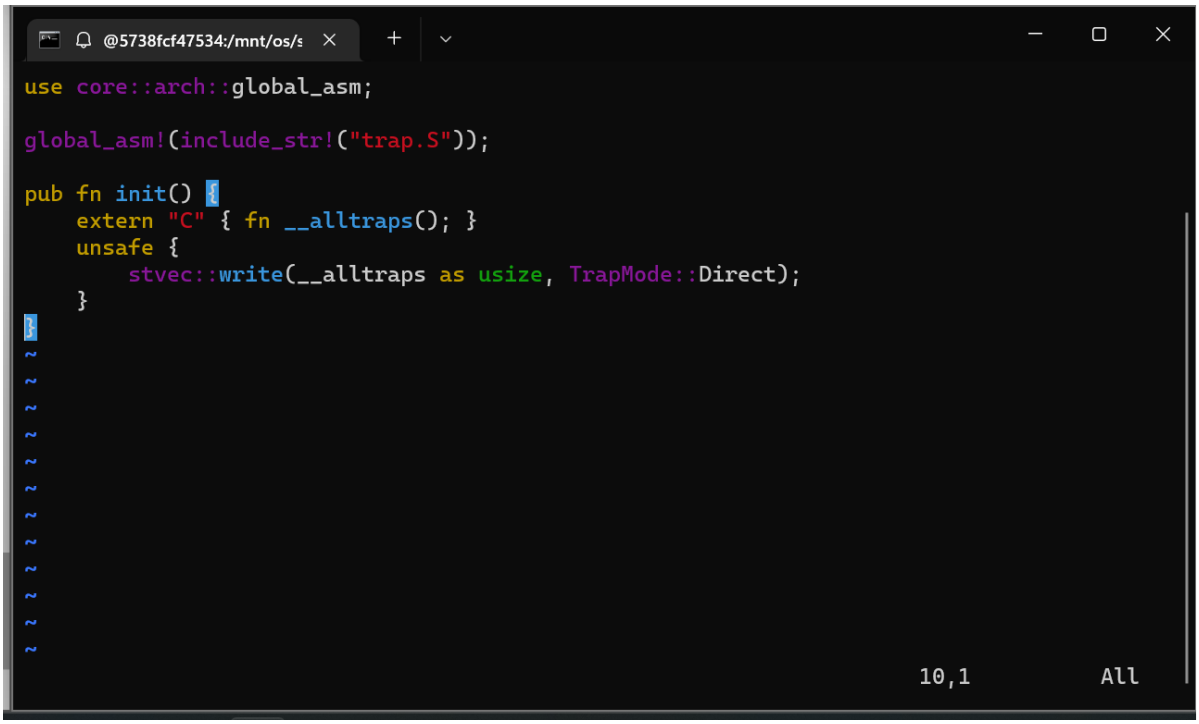
- 定义TrapContext



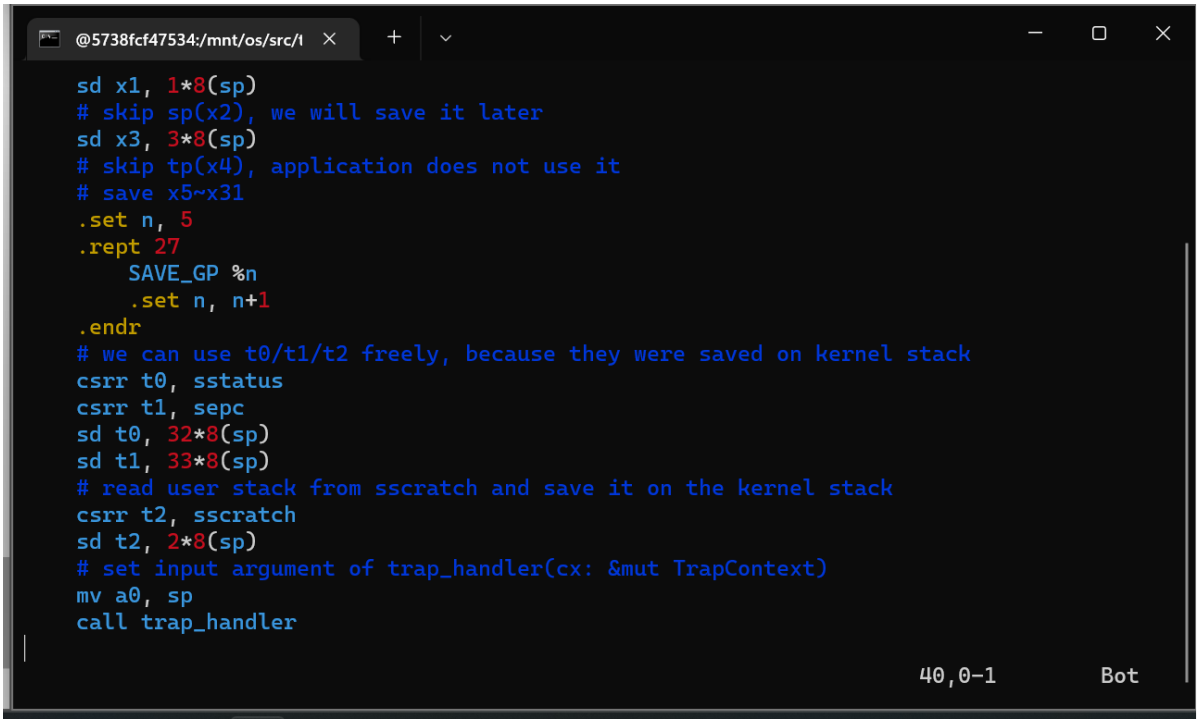
5. 实现trap管理

5.1 Trap上下文的保存与恢复

- 实现mod.rs

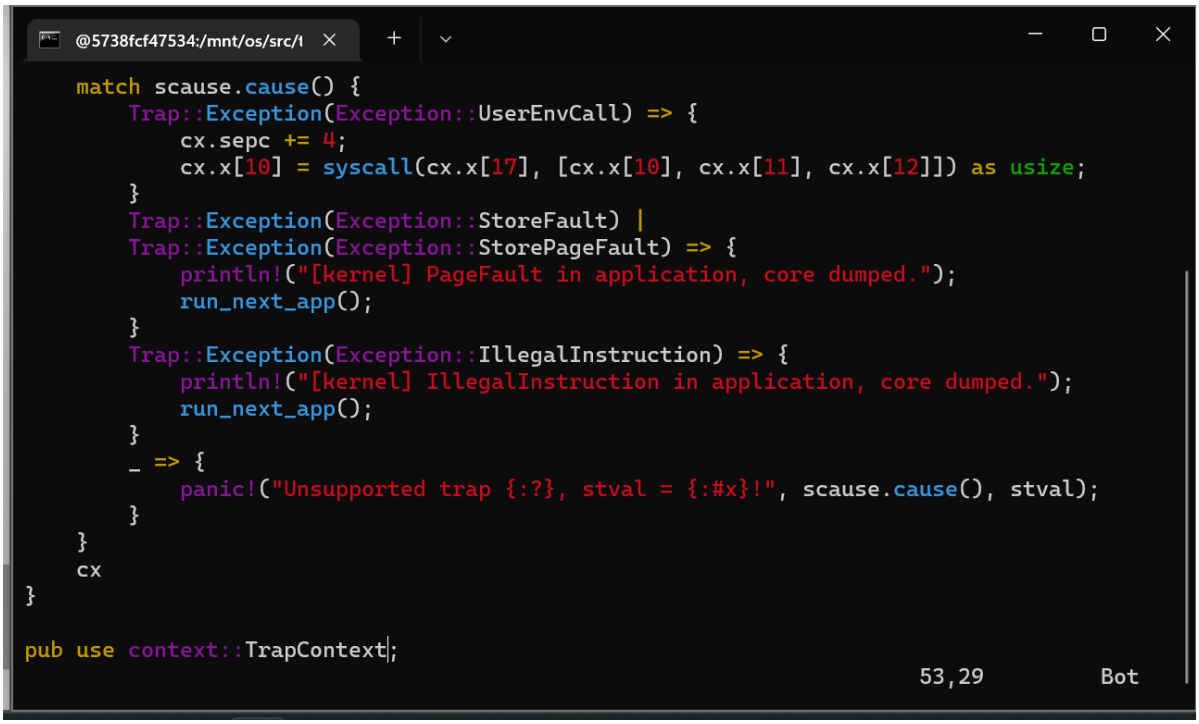


- 实现trap.S



5.2 Trap分发与管理

- 在mod.rs中实现trap_handler

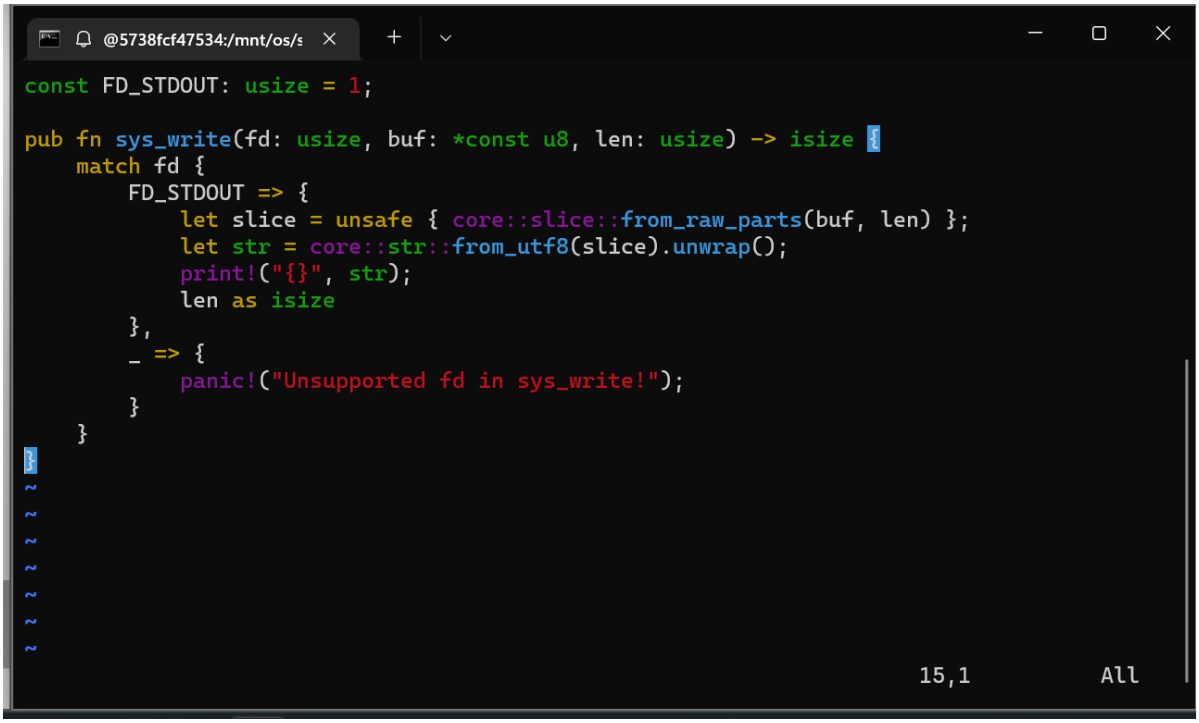


- 增加依赖

5.3 系统调用处理

- 实现syscall模块 mod.rs

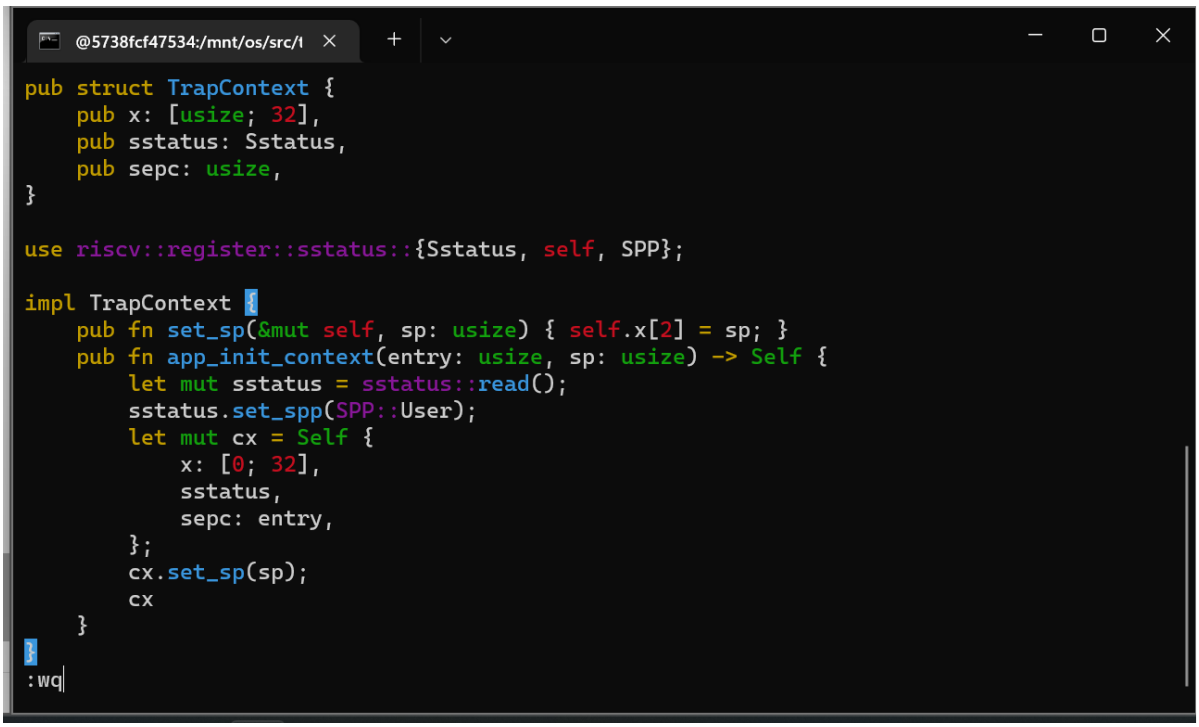
- 实现fs.rs



- 实现process.rs

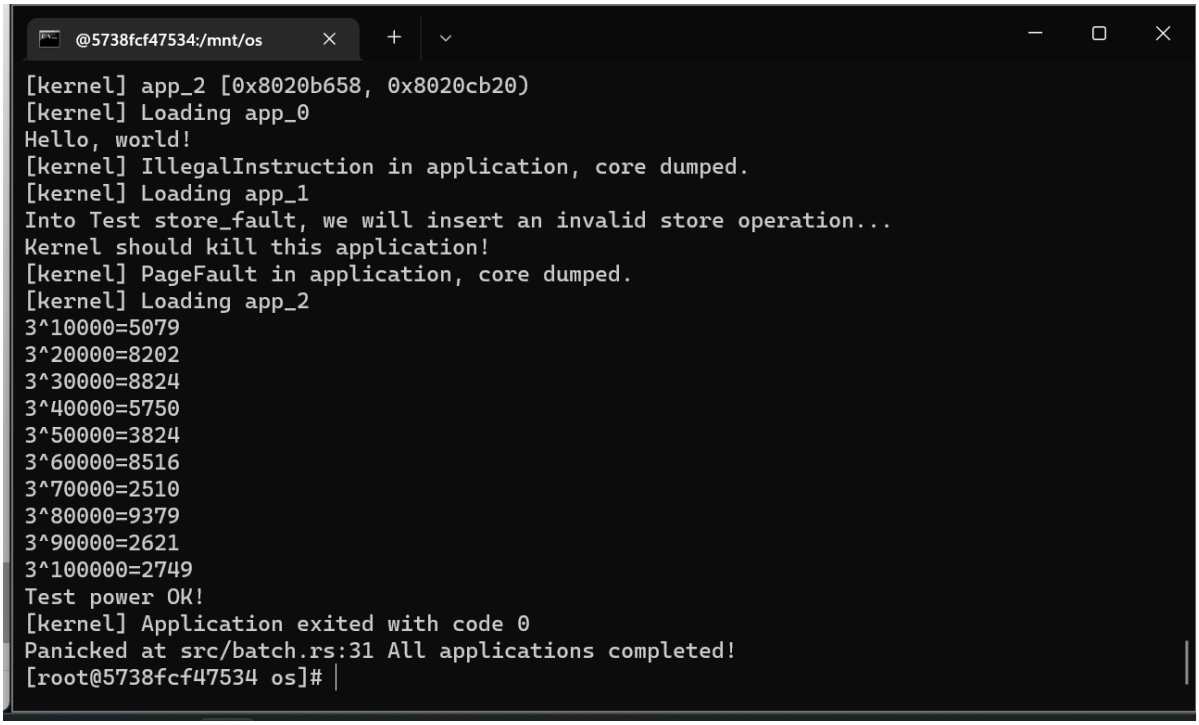

@5738fcf47534:/mnt/os/src/s
use crate::batch::run_next_app;
pub fn sys_exit(exit_code: i32) -> ! {
println!("[kernel] Application exited with code {}", exit_code);
run_next_app()
}
"process.rs" 7L, 163B
7,0-1
All

6. 执行应用程序

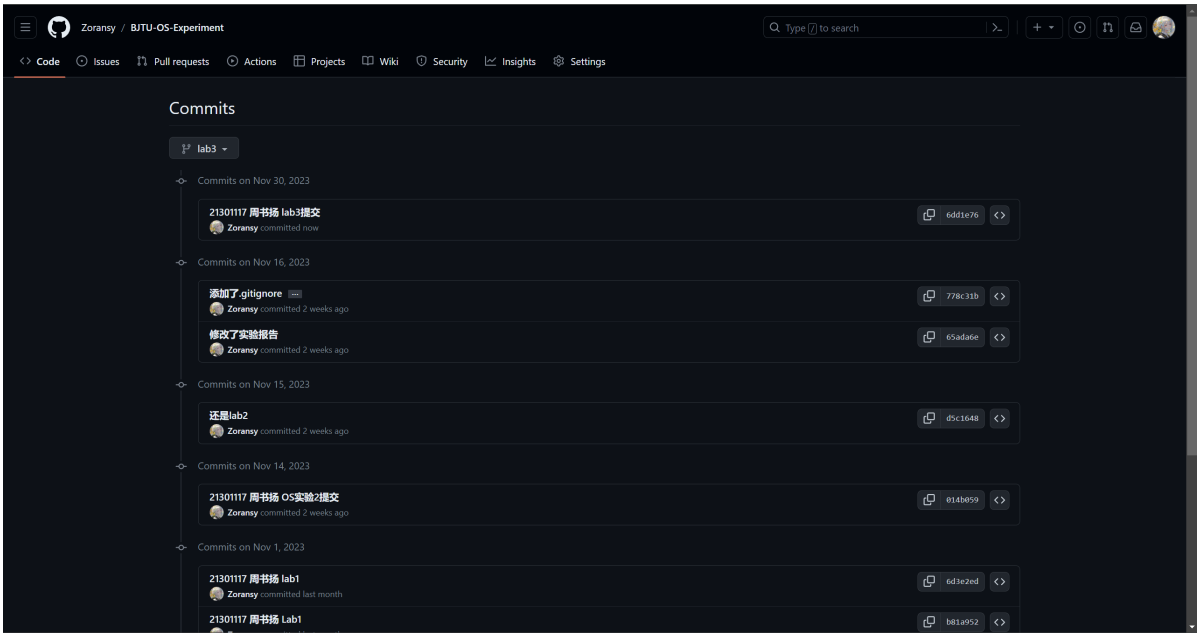

@5738fcf47534:/mnt/os/src/t
pub struct TrapContext {
pub x: [usize; 32],
pub sstatus: Sstatus,
pub sepc: usize,
}
use riscv::register::sstatus::{Sstatus, self, SPP};
impl TrapContext {
pub fn set_sp(&mut self, sp: usize) { self.x[2] = sp; }
pub fn app_init_context(entry: usize, sp: usize) -> Self {
let mut sstatus = sstatus::read();
sstatus.set_spp(SPP::User);
let mut cx = Self {
x: [0; 32],
sstatus,
sepc: entry,
};
cx.set_sp(sp);
cx
}
}
:wq

7. 修改main.rs

8. 运行系统内核



二. git提交截图



三. 思考题

1. 分析应用程序的实现过程，并实现一个自己的应用程序

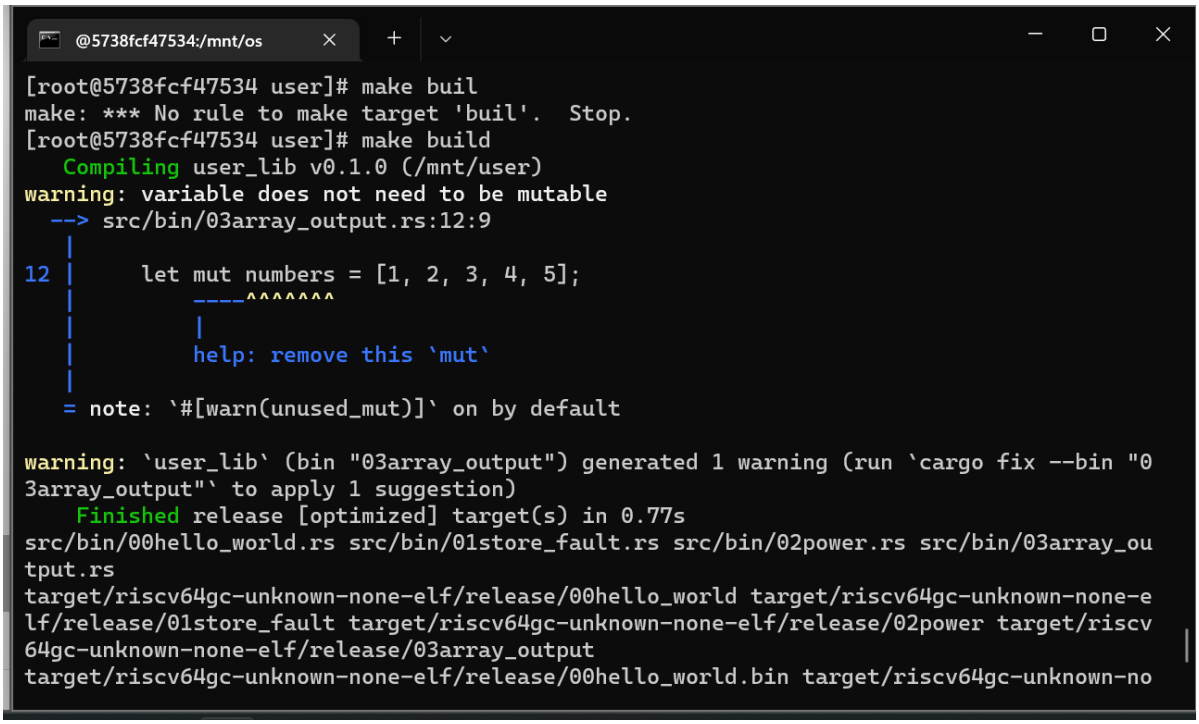
应用程序执行过程：

- 首先编写正确的程序源代码
- 编译源代码到二进制机器码
- 使用编译脚本动态链接编译后的机器码
- 根据batch模块加载应用程序

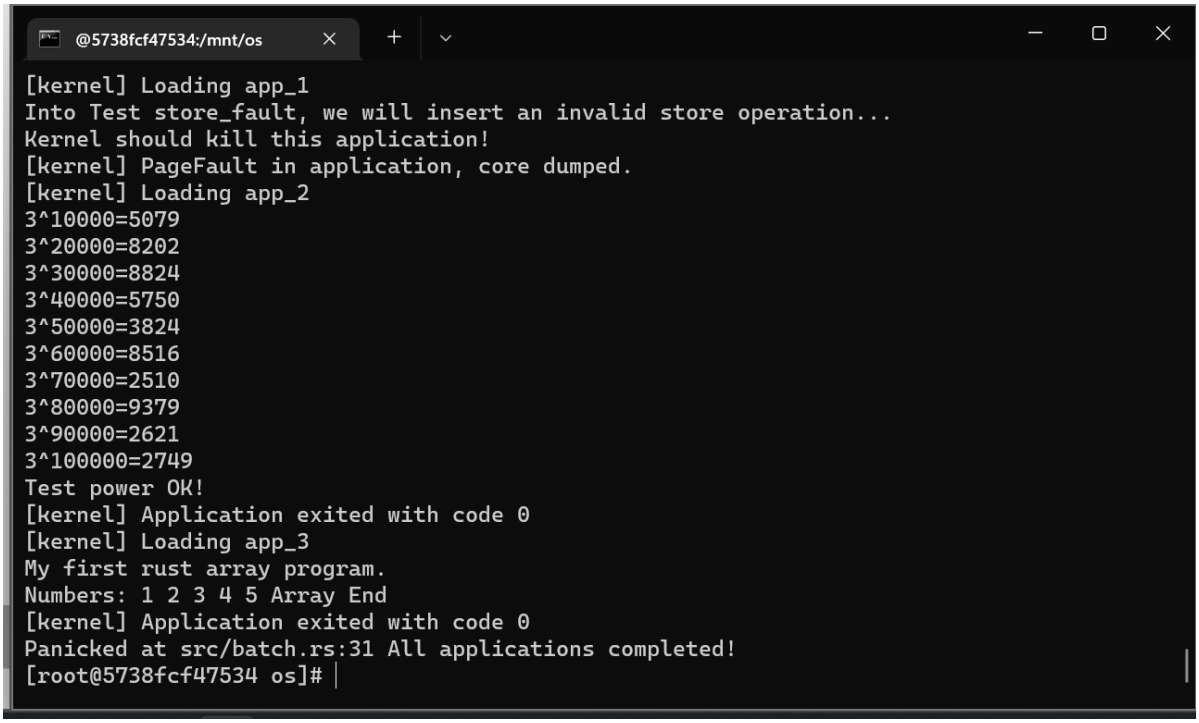
实现一个自己的应用程序

实现了一个打印数组的功能

- 在user目录编译



- 运行系统内核查看输出



程序代码：

```
#![no_std]
#![no_main]

use core::arch::asm;

#[macro_use]
extern crate user_lib;

#[no_mangle]
fn main() -> i32 {
    println!("My first rust array program.");
    let mut numbers = [1, 2, 3, 4, 5];
```

```
    print!("Numbers: ");
    for num in numbers.iter() {
        print!("{} ", num);
    }
    println!("Array End"); // 换一下行，因为打印数组用的是print不是println
    0
}
```

2. 应用程序的链接、加载和执行过程

2.1 链接

应用程序编译为机器码后通过`build.rs`脚本生成一个汇编文件`src/link_app.S`，该汇编文件包括关于应用程序的元数据，在链接时将应用程序的数据正确地嵌入到可执行文件中。该脚本由主要由以下几部分组成：

- **`main`函数：**
  - `println!("cargo:rerun-if-changed=../user/src/");`：通知 Cargo，如果`../user/src/`目录下的文件发生变化，就重新运行该脚本。
  - `println!("cargo:rerun-if-changed={}", TARGET_PATH);`：通知 Cargo，如果`TARGET_PATH`目录下的文件发生变化，就重新运行该脚本。
  - `insert_app_data().unwrap();`：调用`insert_app_data`函数，该函数将应用程序的元数据插入到`src/link_app.S`文件中。
- `static TARGET_PATH: &str = "../user/target/riscv64gc-unknown-none-elf/release/";`：
  - 定义了目标路径，指定了用户应用程序编译输出的位置。
- **`insert_app_data`函数：**
  - 打开`src/link_app.S`文件以供写入。
  - 通过`read_dir("../user/src/bin")`读取用户应用程序目录下的所有二进制文件。
  - 构建一个排序后的应用程序名称的列表。
  - 在`src/link_app.S`文件中写入关于应用程序数量和地址的元数据。
  - 遍历应用程序列表，为每个应用程序生成对应的汇编代码，将应用程序二进制数据嵌入到可执行文件中。

2.2 加载

通过`batch.rs`模块实现保存程序内存位置以及运行索引等信息，主要函数功能如下：

- **`AppManager`结构体和`AppManagerInner`结构体：**
  - `AppManager`是一个包装了`AppManagerInner`的结构体，使用`RefCell`来提供内部可变性。
  - `AppManagerInner`存储了应用程序的相关信息，包括应用程序数量、当前应用程序索引以及应用程序在内存中的起始地址。
- **`lazy_static!`宏：**
  - 使用`lazy_static!`宏声明了一个全局的`APP_MANAGER`变量，该变量在第一次使用时进行实际初始化。这个全局变量用于管理应用程序的加载和运行。

- **`init` 函数和 `print_app_info` 函数：**
  - `init` 函数调用了 `print_app_info` 函数，用于打印应用程序的信息。在实际应用中，可能还会进行一些初始化操作。
  - `print_app_info` 函数输出应用程序的数量以及它们在内存中的位置信息。
- **run_next_app` 函数：**
  - 获取当前应用程序的索引。
  - 调用 `load_app` 函数加载当前应用程序的二进制数据到内存中。
  - 将应用程序索引递增，以便下次运行下一个应用程序。
  - 调用外部函数 `__restore` 来切换到应用程序的上下文，实现从内核态到用户态的切换。这里使用了一个名为 `KERNEL_STACK` 的堆栈，以及 `TrapContext` 结构体来保存和恢复上下文信息。
  - 最后，通过 `panic!` 宏表明在 `run_next_app` 函数中的代码应该是不可达的，因为切换到应用程序上下文后，控制流不会回到这里。

2.3 执行过程

> 因为感觉实验手册写得足够详细了所以直接复制了

在执行应用程序之前，需要跳转到应用程序入口`0x80400000`，切换到用户栈，设置`sscratch`指向内核栈，并且用S特权级切换到U特权级。我们可以通过复用`restore`的代码来实现这些操作。这样的话，只需要在内核栈上压入一个启动应用程序而特殊构造的Trap上下文，再通过`restore`函数就可以实现寄存在为启动应用程序所需的上下文状态。

3. Trap的实现方式

- **Trap 上下文的保存与恢复：**
  - 在 `os/src/trap/trap.S` 中，`__alltraps` 宏保存了 Trap 上下文。
  - 在 `os/src/trap/trap.S` 中，`__restore` 宏从保存的内核栈上的 Trap 上下文中恢复寄存器状态。
- **Trap 分发与处理：**
  - `os/src/trap/mod.rs` 中的 `trap_handler` 函数用于根据 `scause` 寄存器的值进行 Trap 的分发与处理。
  - 当 Trap 是用户环境调用（`UserEnvCall`）时，调用 `syscall` 处理系统调用，并更新上下文中的寄存器。
  - 当 Trap 是存储故障（`StoreFault` 或 `StorePageFault`）时，输出错误信息并调用 `run_next_app` 启动下一个应用程序。
  - 当 Trap 是非法指令（`IllegalInstruction`）时，同样输出错误信息并调用 `run_next_app` 启动下一个应用程序。
  - 对于其他类型的 Trap，抛出 panic。
- **Trap 处理入口点的设置：**
  - 在 `os/src/trap/mod.rs` 中的 `init` 函数中，通过 `stvec::write` 设置 `stvec` 寄存器，将 Trap 处理入口点指向 `__alltraps`。