2023秋《操作系统》课程实验报告

实验三

21301161 齐家昕

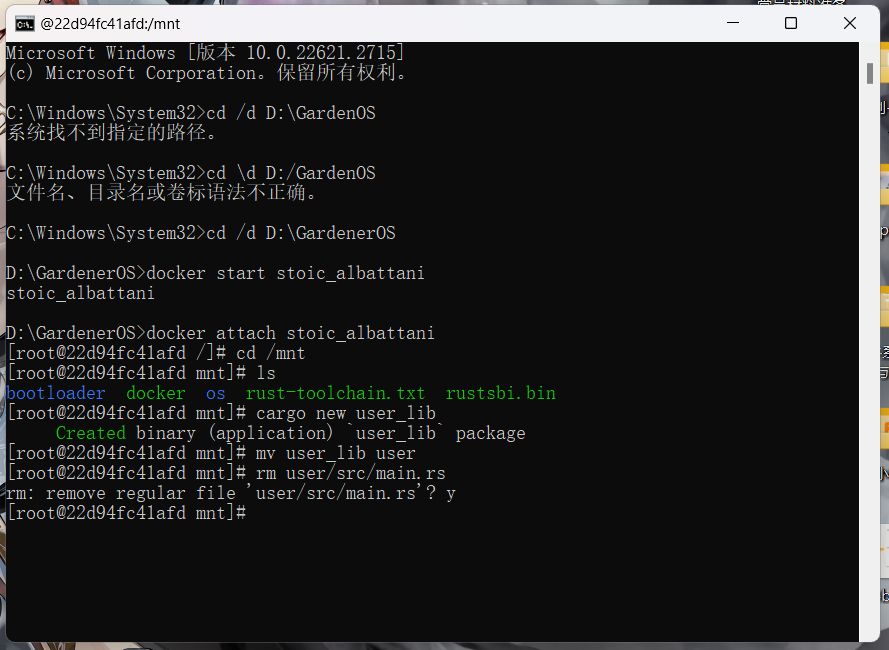
1. 实验步骤

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。

1. 设计和实现应用程序

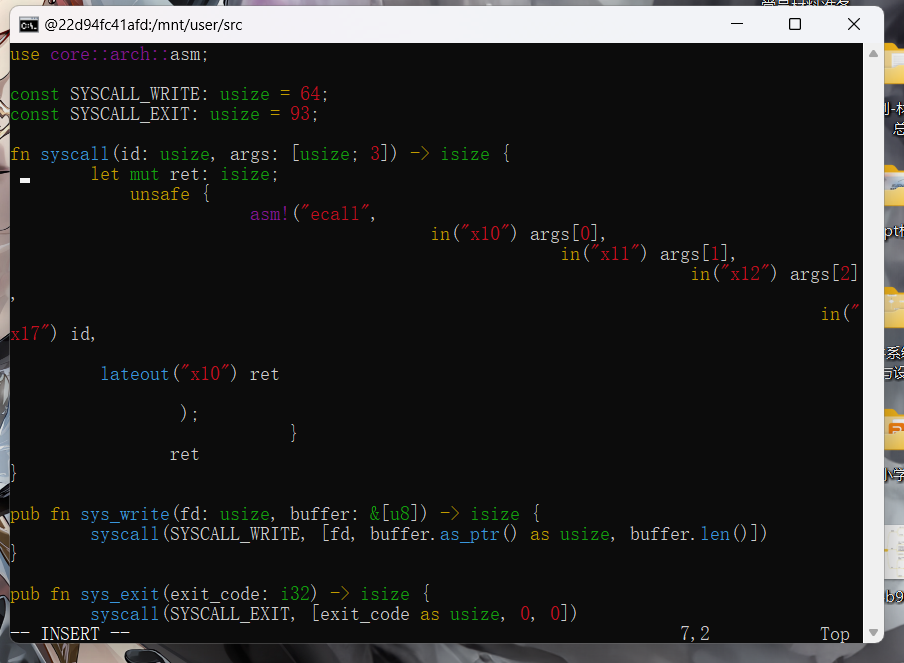
实现一个简单的批处理操作系统，需要先实现应用程序，要求应用程序在用户态下运行。应用程序及其对应的库文件放置在工作目录下的user目录下（和os目录是同一级的）。这部分的实现与上一节裸机环境和最小化内核部分有很多相同的实现。

创建user目录：



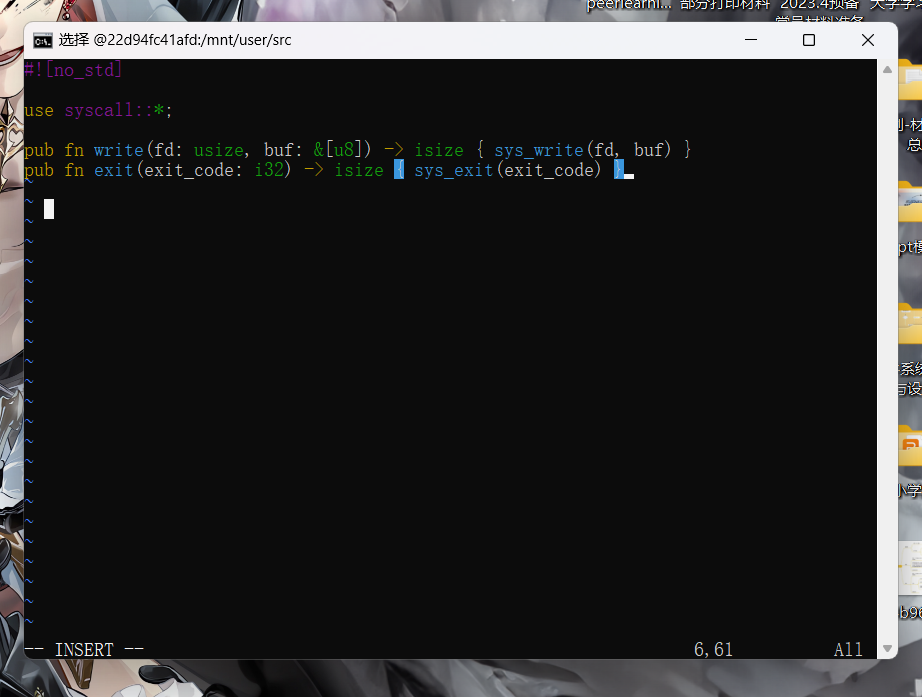
（1）首先实现应用程序与系统约定的两个系统调用sys\_write和sys\_exit

具体代码在user/src/syscall.rs中，具体内容如下：



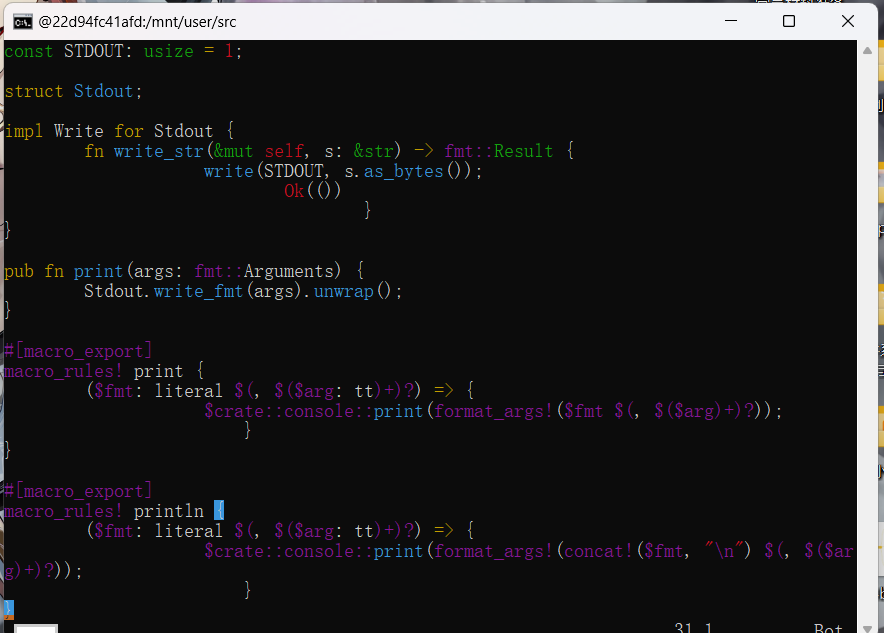
同时，还需要在lib.rs实现进一步的封装。

在user/src/lib.rs中增加如下内容：



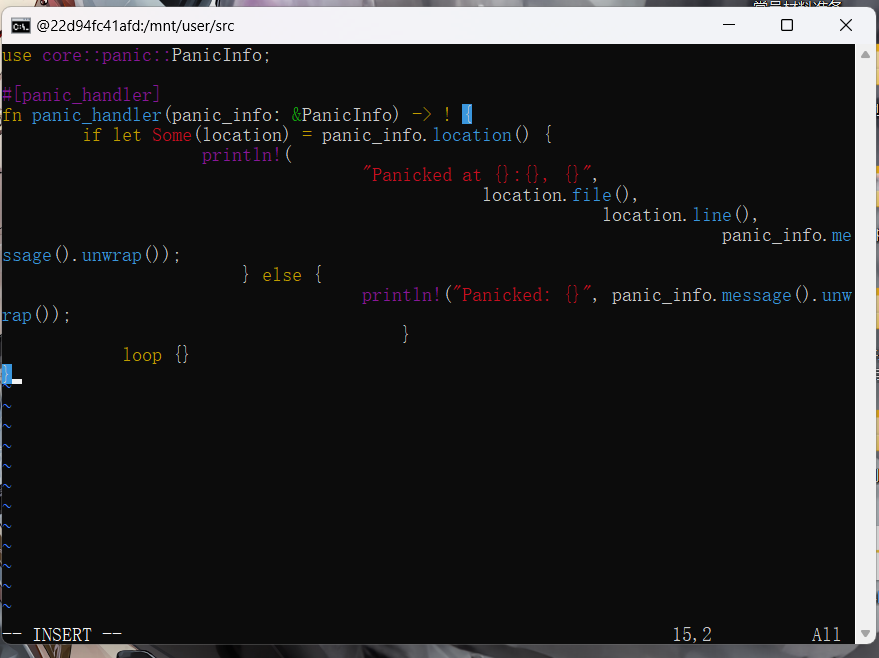
（2）实现格式化输出

为了实现格式化输出，我们还需要把 Stdout::write\_str 改成基于 write 的实现，且传入的 fd 参数设置为 1，它代表标准输出， 也就是输出到屏幕。具体代码在user/src/console.rs中，具体内容如下：



（3）实现语义支持

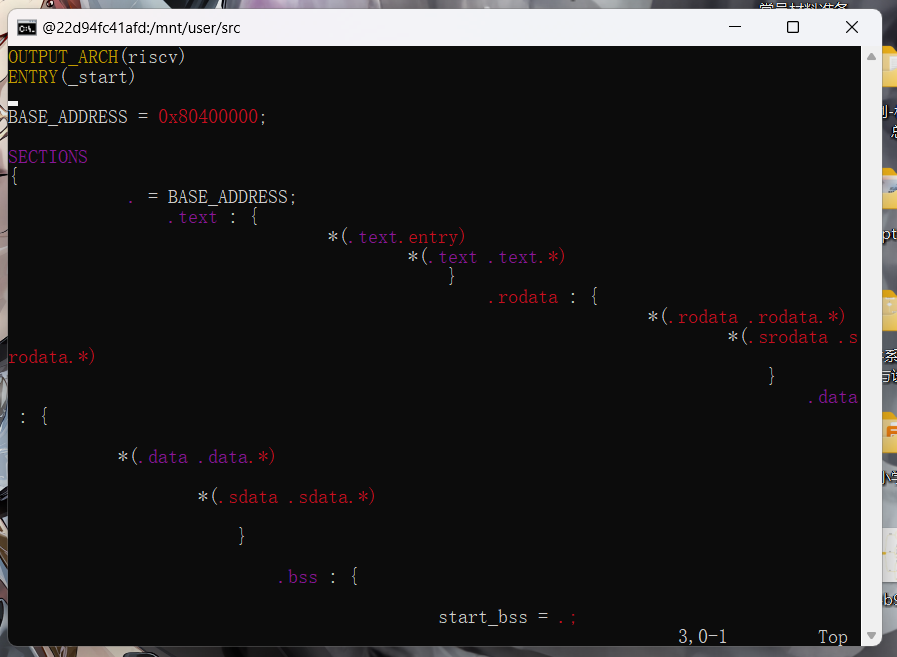
此外，还需要实现对panic的处理。具体代码在user/src/lang\_items.rs。



（4）应用程序内存布局

我们还需要将应用程序的起始物理地址调整为 0x80400000，这样应用程序都会被加载到这个物理地址上运行，从而进入用户库的入口点，并会在初始化之后跳转到应用程序主逻辑。实现方式类似前一节的linker.ld。

具体代码在user/src/linker.ld，具体内容如下：

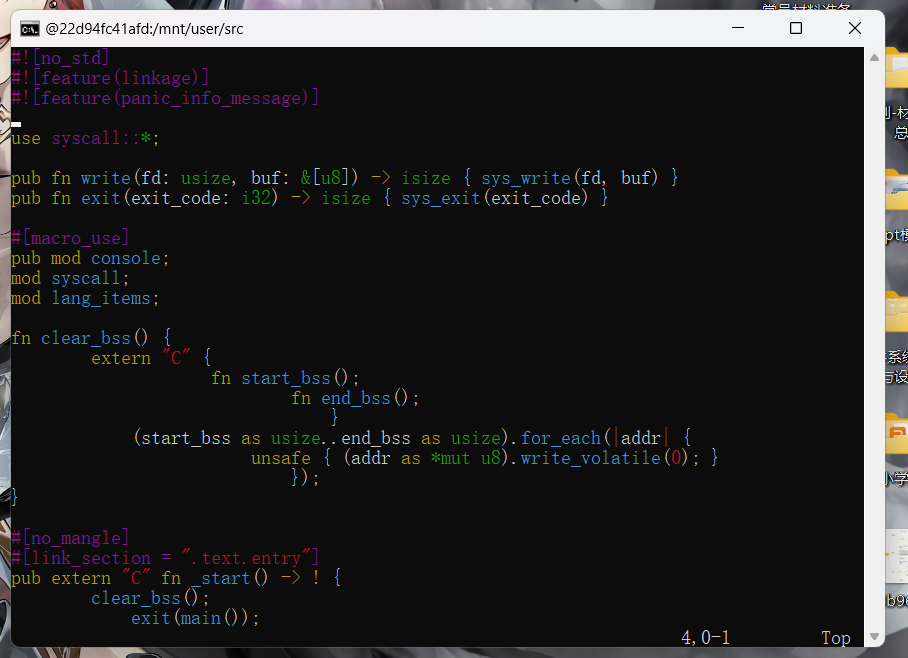


同时，注意增加配置文件使用linker.ld文件，user/.cargo/config配置文件的内容如下：

（5）最终形成运行时库lib.rs

定义用户库的入口点 \_start，\_start这段代码编译后会存放在.text.entry代码段中，这在前面内存布局中已经定义了。此外，通过#[linkage = "weak"]确保lib.rs和bin下同时存在main的编译能够通过。在lib.rs增加的代码如下：

注意：最开始的两行代码需要放在 #![no\_std]的下面，放在后面会出现错误。



（6）应用程序模板

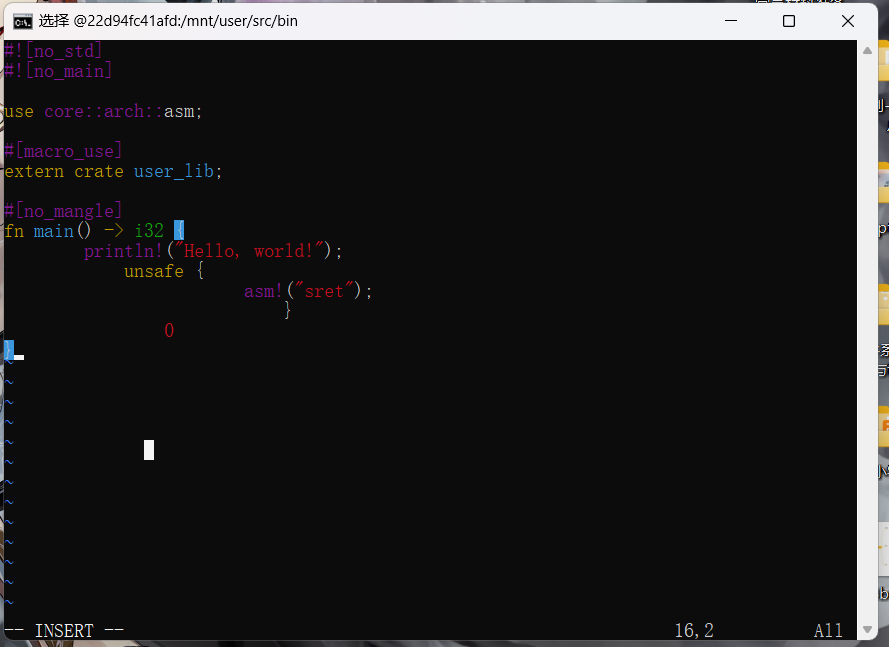
应用程序都存放在usr/src/bin下，模板如下。这段模板代码引入了外部库，就是lib.rs定义以及它所引用的子模块。

注意：下面的模板代码不需创建，只是作为后续创建应用程序的参考。

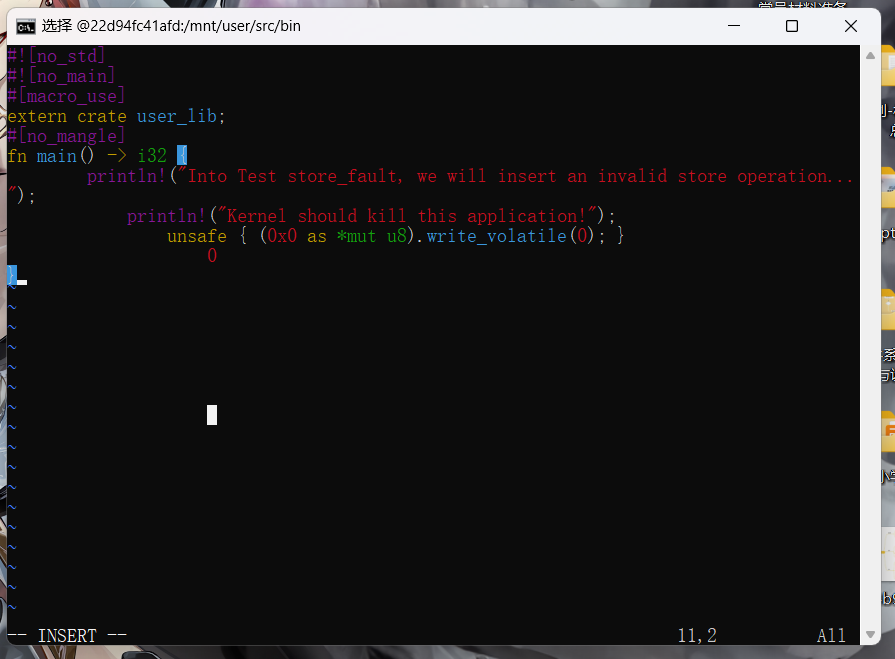
（7）实现多个不同的应用程序

基于上述模板，我们可以在bin下实现多个不同的应用程序。

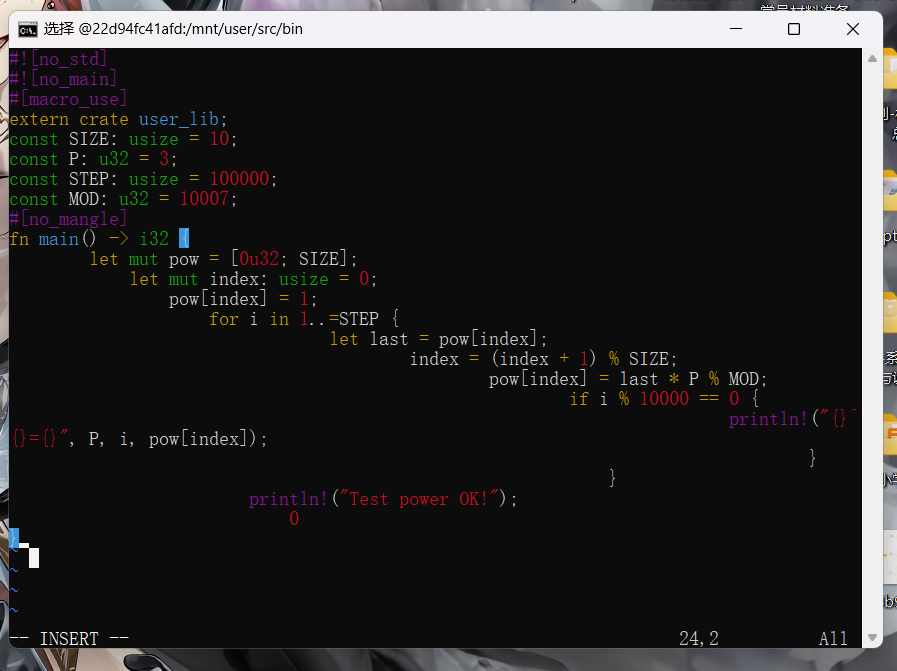
其中user/src/bin/00hello\_world.rs内容具体如下：



/user/src/bin/01store\_fault.rs内容具体如下：

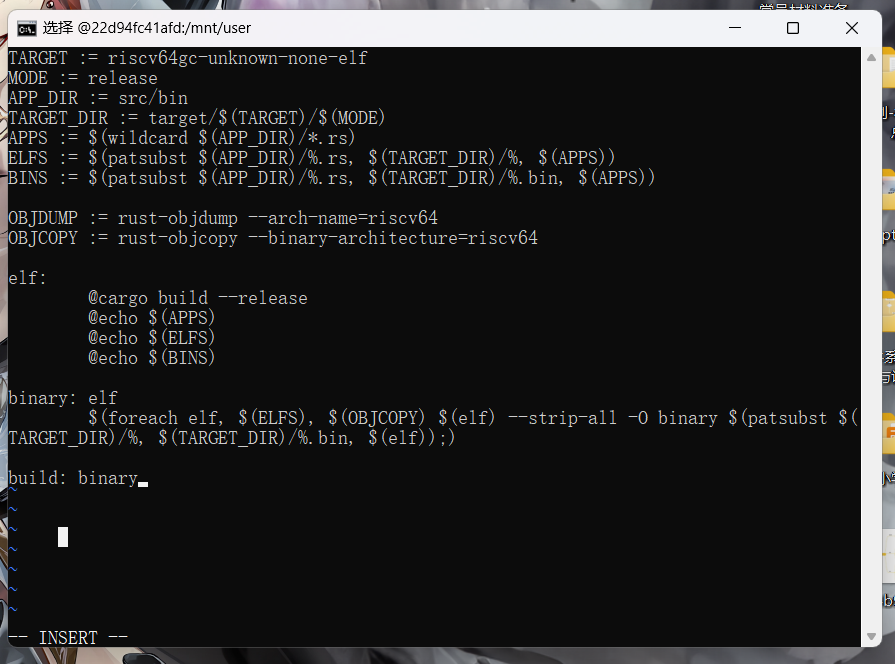


/user/src/bin/02power.rs内容具体如下：

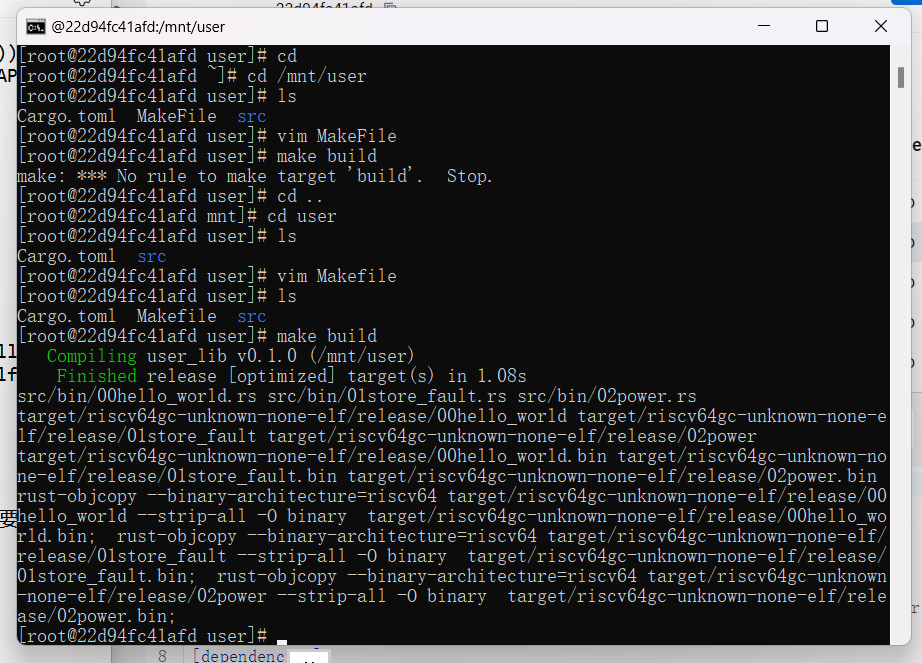


1. 编译生成应用程序二进制码

编写Makefile文件，user/Makefile内容如下：



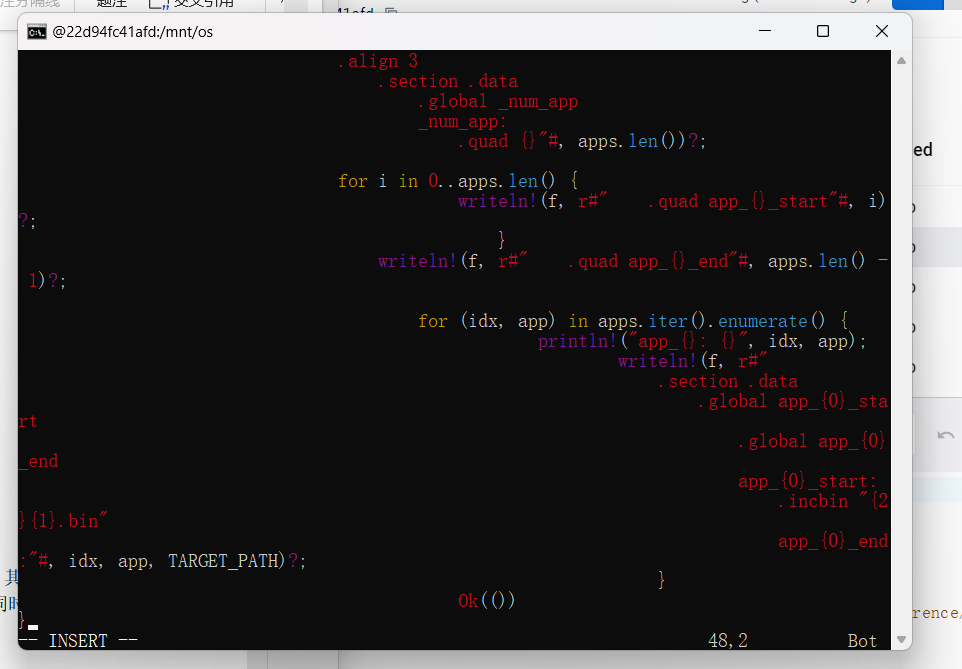
执行make build进行编译。



编译完成后，我们可以利用qemu-riscv64模拟器执行编译生成的程序。需要注意的是，特权指令在这里是无法直接执行的。至此，应用程序设计实现完成。

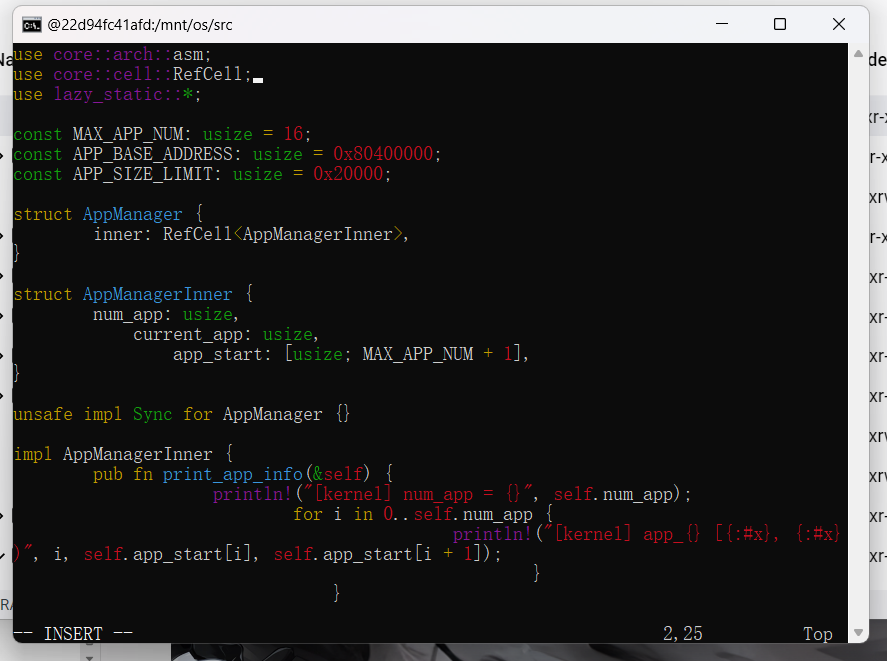
2. 链接应用程序到内核

为了将编译生成的应用程序二进制码动态链接到内容中，我们需要编写一个编译脚本os/build.rs以生成专门用于链接的脚本文件link\_app.S。注意是该汇编脚本直接放在项目文件夹 os 中，而不是源码文件夹 src。当执行cargo build命令时就会执行build.rs脚本。



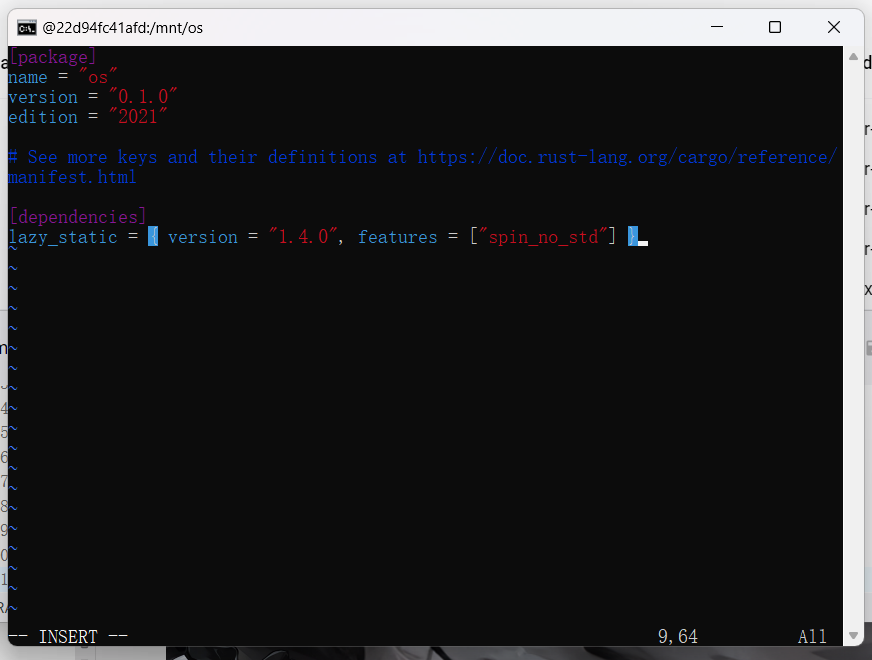
3. 找到并加载应用程序二进制码

为了实现批处理操作系统，我们在os目录下实现一个batch子模块。其主要功能是保存应用程序的数据及对应的位置信息，以及当前执行到第几个应用程序。同时，也会初始化应用程序所需的内存并加载执行应用程序。



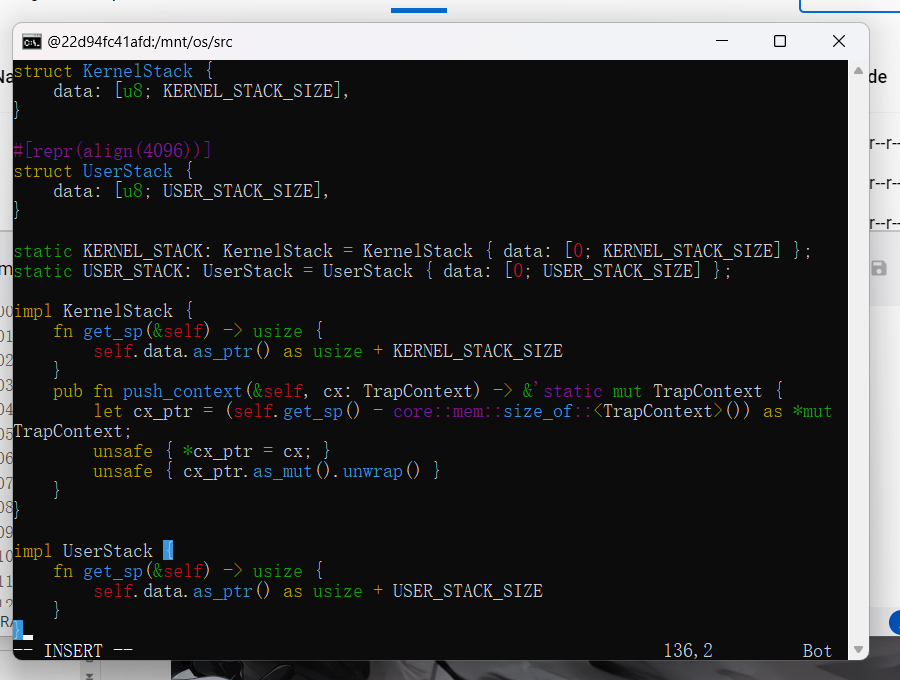
因为使用了外部库 lazy\_static 提供的 lazy\_static! 宏，因此需要在Cargo.toml中加入依赖。lazy\_static!宏提供了全局变量的运行时初始化功能，我们借助lazy\_static!声明了一个 AppManager结构的全局实例APP\_MANAGER，使得只有在第一次使用它时才会进行实际的初始化工作。

修改os/Cargo.toml配置文件。

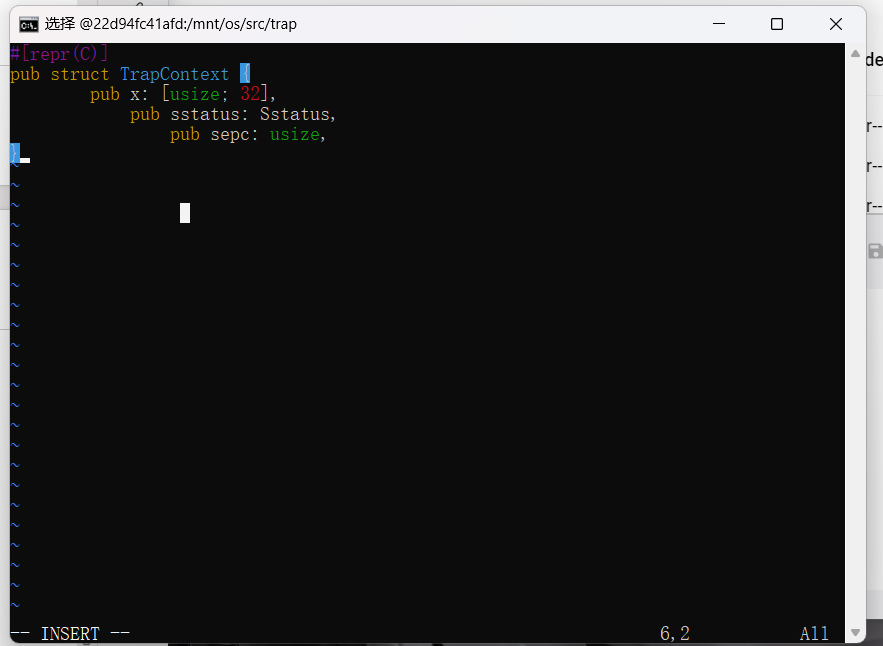
l

4. 实现用户栈和内核栈

为了实现特权级的切换，还需要实现用户栈和内核栈。在batch.rs中增加如下实现。需要注意在RISC-V中栈是向下增长的。



其中，TrapContext是在trap中定义的。



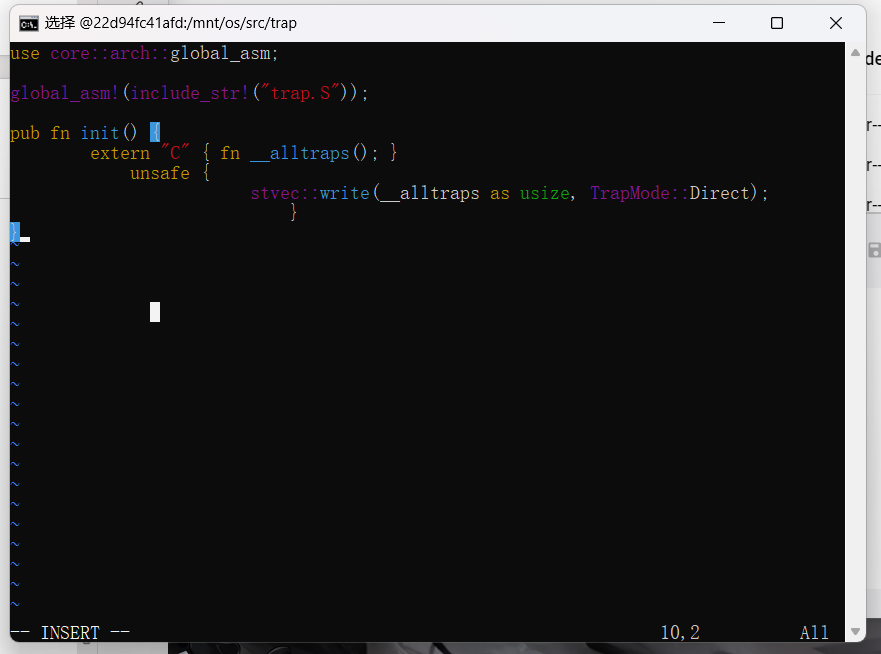
5. 实现trap管理

特权级切换的主要内容就是实现对trap的管理。其主要内容就是当应用程序通过ecall进入到内核状态时，要保存被中断的应用程序的上下文。同时，还要根据CSR寄存器内容完成系统调用的分发与处理。在完成系统调用后，还需要恢复被中断的应用程序的上下文，并通 sret 让应用程序继续执行。

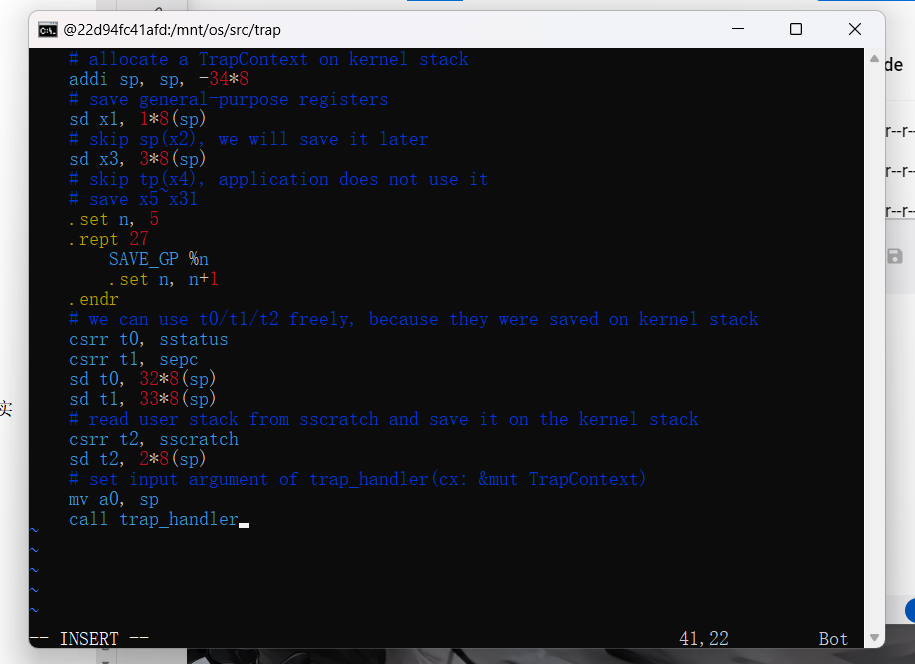
Trap处理的流程大致如下：首先通过将 Trap上下文保存在内核栈上，然后跳转到trap处理函数完成 Trap 分发及处理。当处理函数返回之后，再从保存在内核栈上的Trap上下文恢复寄存器。最后通过一条 sret 指令回到应用程序继续执行。

（1）Trap 上下文的保存与恢复

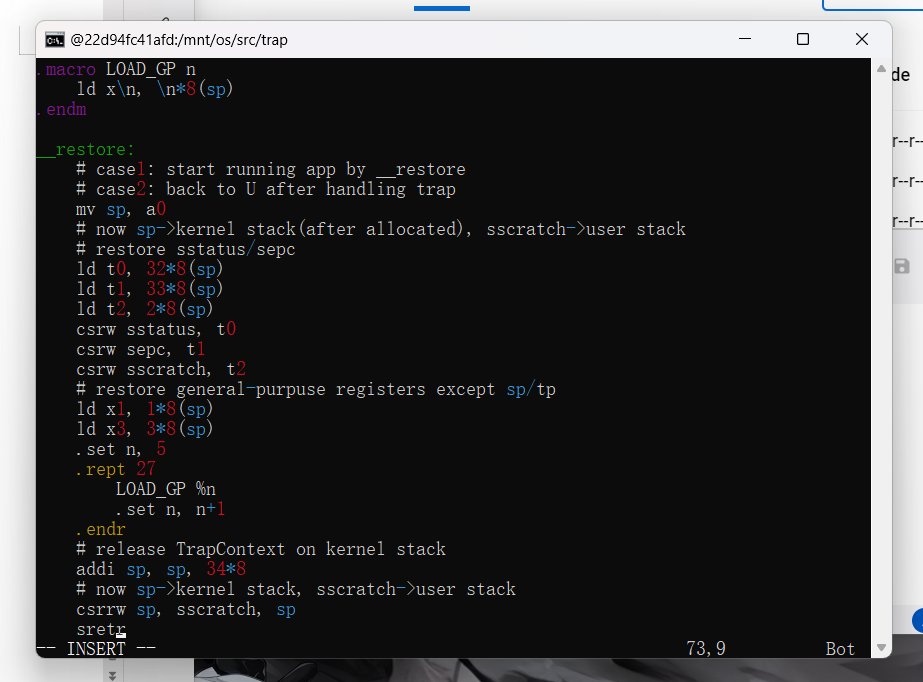
首先，修改stvec寄存器来指向正确的 Trap 处理入口点。



这里我们引入了外部符号\_\_alltraps来将Trap上线文保存在内核栈上。从上面的代码可以看出 \_\_alltraps 的实现在os/src/trap/trap.S中。

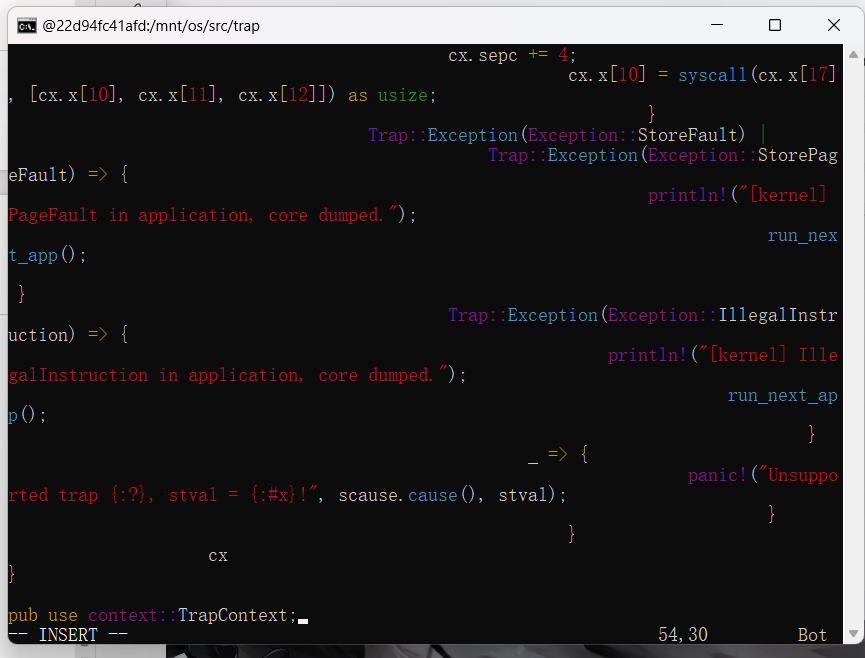


当Trap处理函数返回之后，还需要从栈上的Trap 上下文恢复的。通过 \_\_restore来实现。

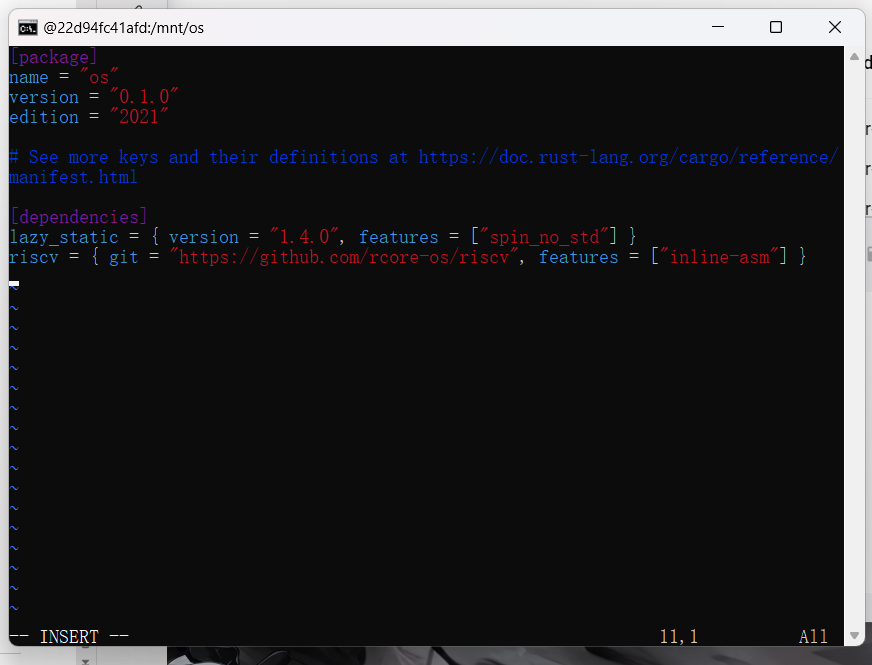


（2）Trap 分发与处理

我们通过实现trap\_handler 函数完成Trap的分发和处理。

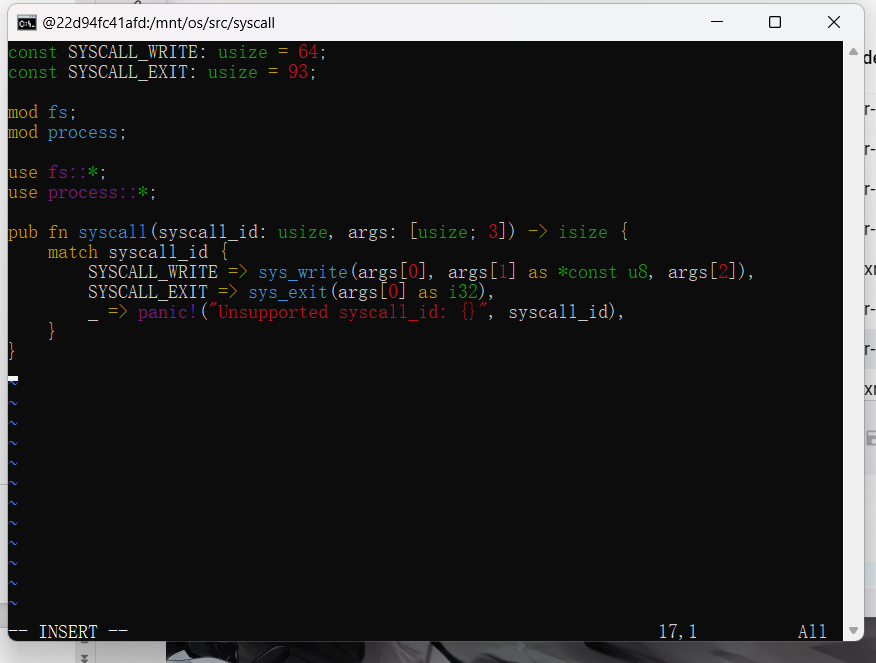


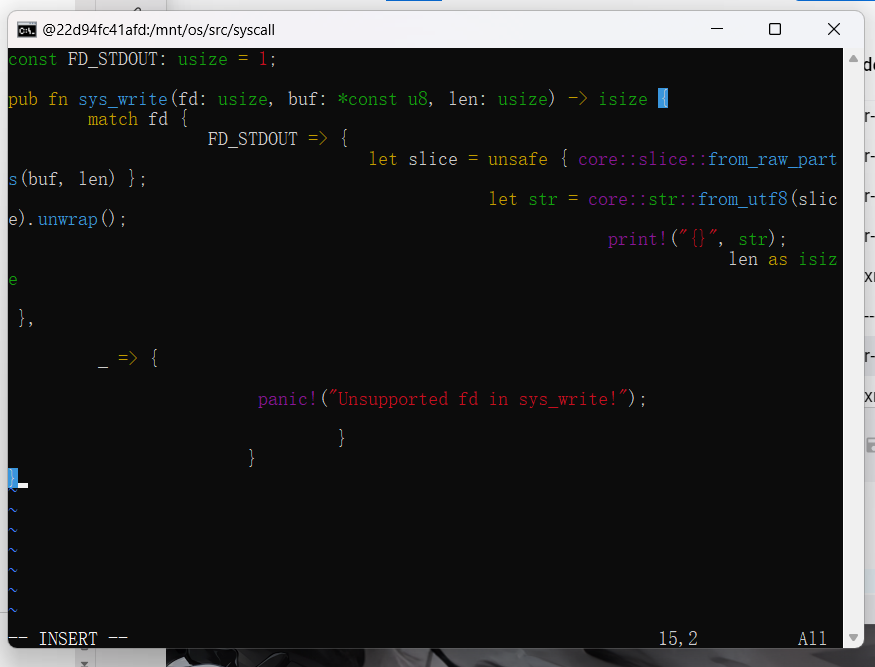
因为引入了riscv库，所以需要修改配置文件Cargo.toml.

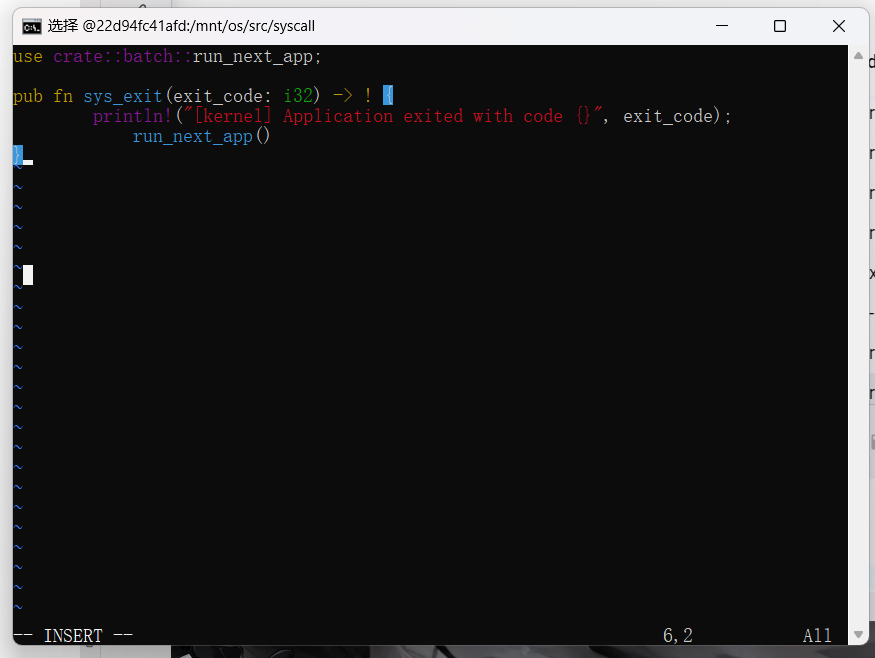


（3）系统调用处理

为了实现对系统调用的处理，我们还需要实现syscall模块。syscall函数并不真正的处理系统调用，而是根据syscall ID分发到具体的处理函数进行处理。

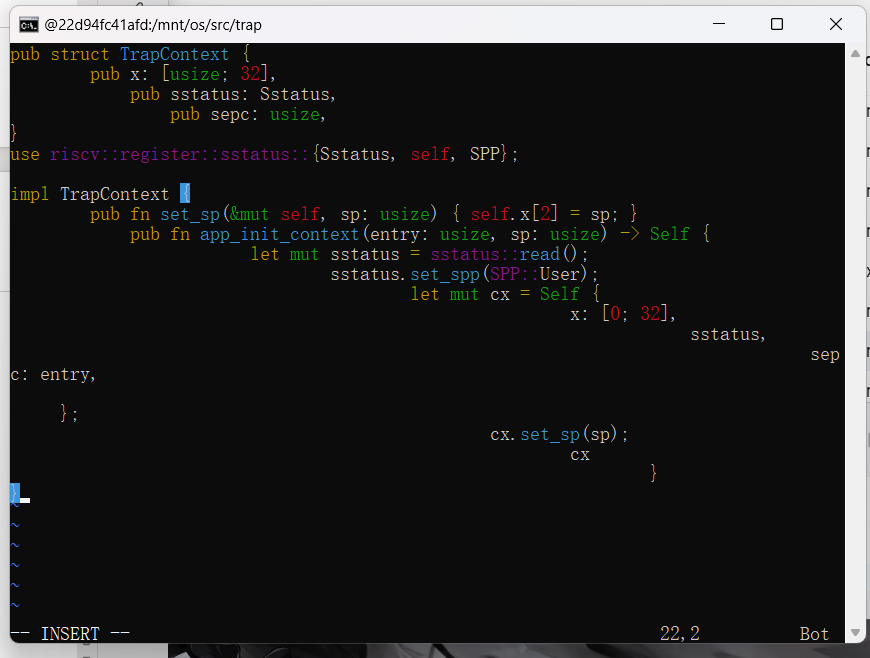






6. 执行应用程序

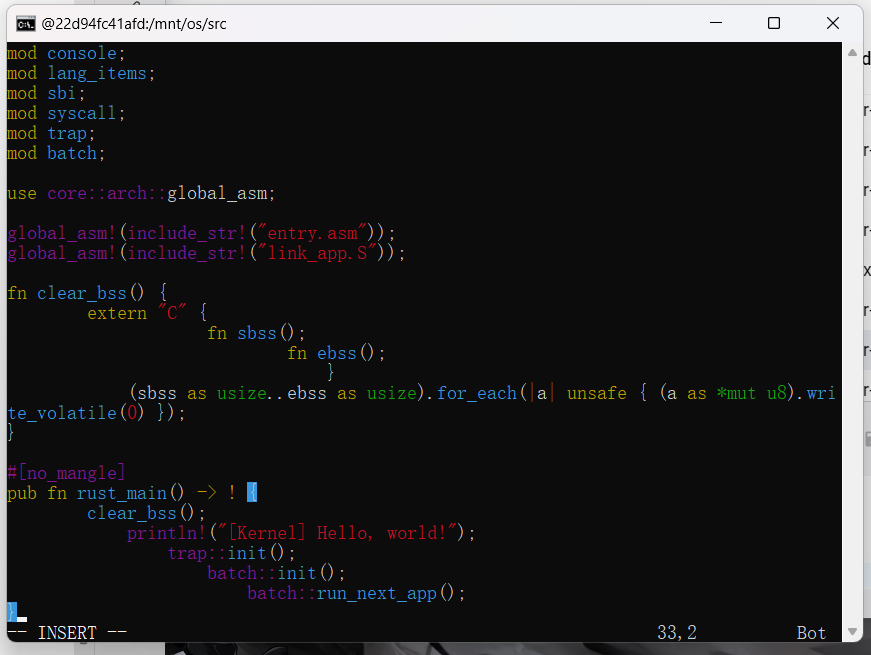
在执行应用程序之前，需要跳转到应用程序入口0x80400000，切换到用户栈，设置sscratch指向内核栈，并且用S特权级切换到U特权级。我们可以通过复用\_\_restore的代码来实现这些操作。这样的话，只需要在内核栈上压入一个启动应用程序而特殊构造的Trap上线文，再通过\_\_restore函数就可以实现寄存在为启动应用程序所需的上下文状态。



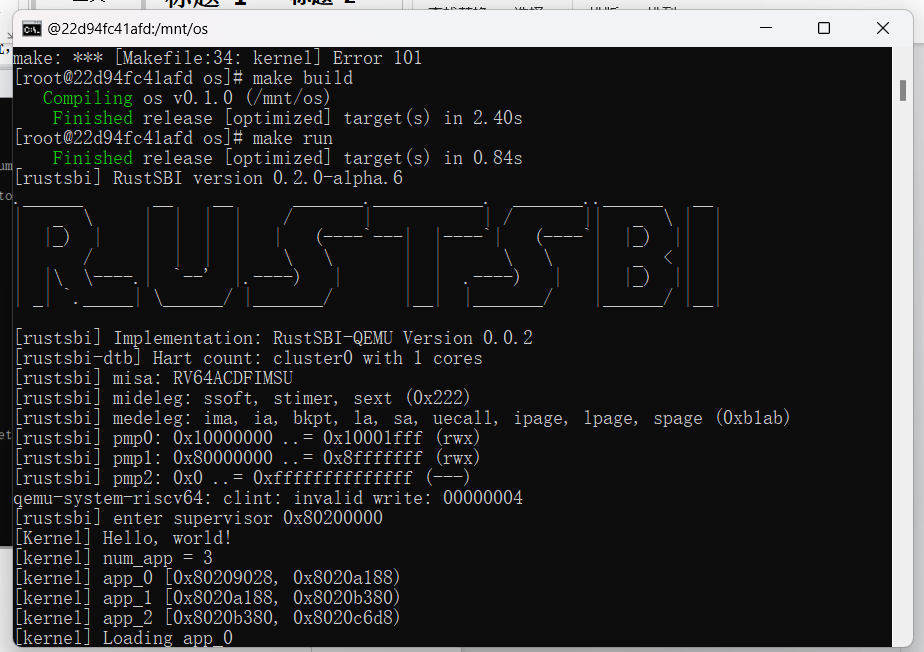
同时，在batch.rs的run\_next\_app中我们可以看到调用了\_\_restore在内核栈上压入了一个Trap上下文。

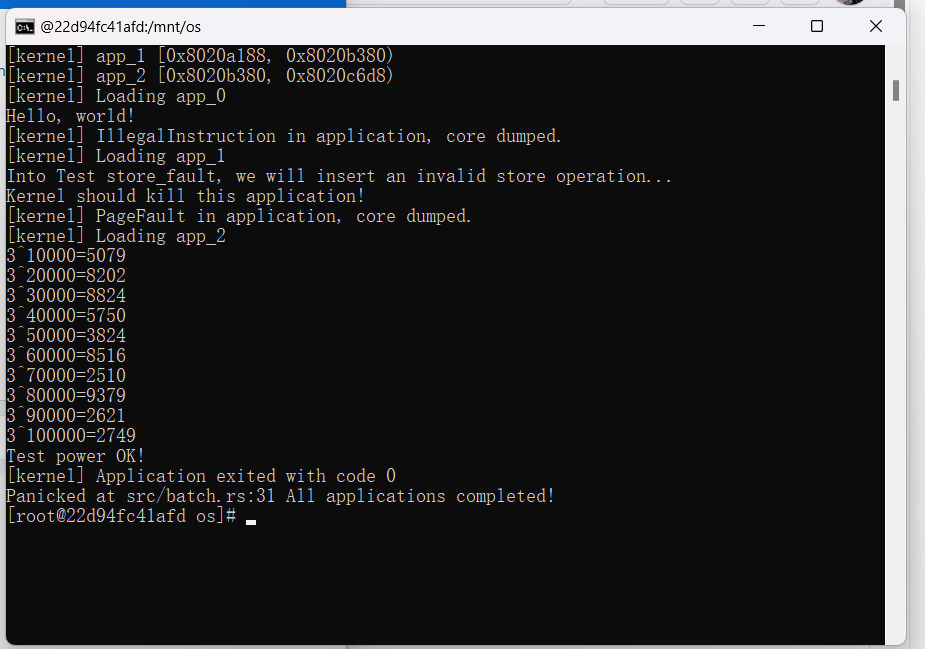
1. 修改main.rs

最后，修改main.rs增加新实现的模块，并且调用batch子模块进行初始化并批量执行应用程序。



至此，批处理操作系统的实现完成。运行操作系统，查看系统运行结果是否正确。





1. 思考问题

（1）分析应用程序的实现过程，并实现一个自己的应用程序；

（2）分析应用程序的链接、加载和执行过程；

在链接过程中，将应用程序的目标文件与操作系统的内核链接在一起，程序的起始地址被设置为 0x80400000，为可执行文件在内存中的加载地址。通过link\_app.S将获得的所有二进制文件写入操作系统中。加载过程发生在操作系统启动时，负责将可执行文件加载到内存中的指定位置。操作系统的加载器负责这个任务，它会将程序复制到内存中，并调整程序计数器等寄存器的值，以确保执行从正确的内存地址开始。执行过程将加载后的应用程序从其起始地址开始执行。

（3）分析Trap是如何实现的？

Trap处理的流程大致如下：首先通过将 Trap上下文保存在内核栈上，然后跳转到trap处理函数完成 Trap 分发及处理。当处理函数返回之后，再从保存在内核栈上的Trap上下文恢复寄存器。最后通过一条 sret 指令回到应用程序继续执行。

1. Git提交截图

