> OS-Lab4 多道程序与协作式调度 21301117 周书扬
>
> 本实验的主要目的是实现一个支持多道程序和协作式调度的操作系统。

一. 实验步骤

1. 实现应用程序

1.1 应用程序的放置

- 编写`build.py`脚本

```
@5738fcf47534:/mnt/user
import os

base_address = 0x80400000
step = 0x20000
linker = 'src/linker.ld'

app_id = 0
apps = os.listdir('src/bin')
apps.sort()
for app in apps:
    app = app[:app.find('.')]
    lines = []
    lines_before = []
    with open(linker, 'r') as f:
        for line in f.readlines():
            lines_before.append(line)
            line = line.replace(hex(base_address), hex(base_address+step*app_id))
            lines.append(line)
    with open(linker, 'w+') as f:
        f.writelines(lines)
    os.system('cargo build --bin %s --release' % app)
@@
                                                                1,9           Top
```

- 修改`user/Makefile`

```
@5738fcf47534:/mnt/user
TARGET := riscv64gc-unknown-none-elf
MODE := release
APP_DIR := src/bin
TARGET_DIR := target/$(TARGET)/$(MODE)
APPS := $(wildcard $(APP_DIR)/*.rs)
ELFS := $(patsubst $(APP_DIR)/%.rs, $(TARGET_DIR)/%, $(APPS))
BINS := $(patsubst $(APP_DIR)/%.rs, $(TARGET_DIR)/%.bin, $(APPS))

OBJDUMP := rust-objdump --arch-name=riscv64
OBJCOPY := rust-objcopy --binary-architecture=riscv64

elf: $(APPS)
    @python3 build.py

binary: elf
    $(foreach elf, $(ELFS), $(OBJCOPY) $(elf) --strip-all -O binary $(patsubst $(TARGET
_DIR)/%, $(TARGET_DIR)/%.bin, $(elf));)

build: binary

clean:
        @cargo clean
                                                                2,8           Top
```

1.2 增加`yield`系统调用

- 增加sys_yield系统调用

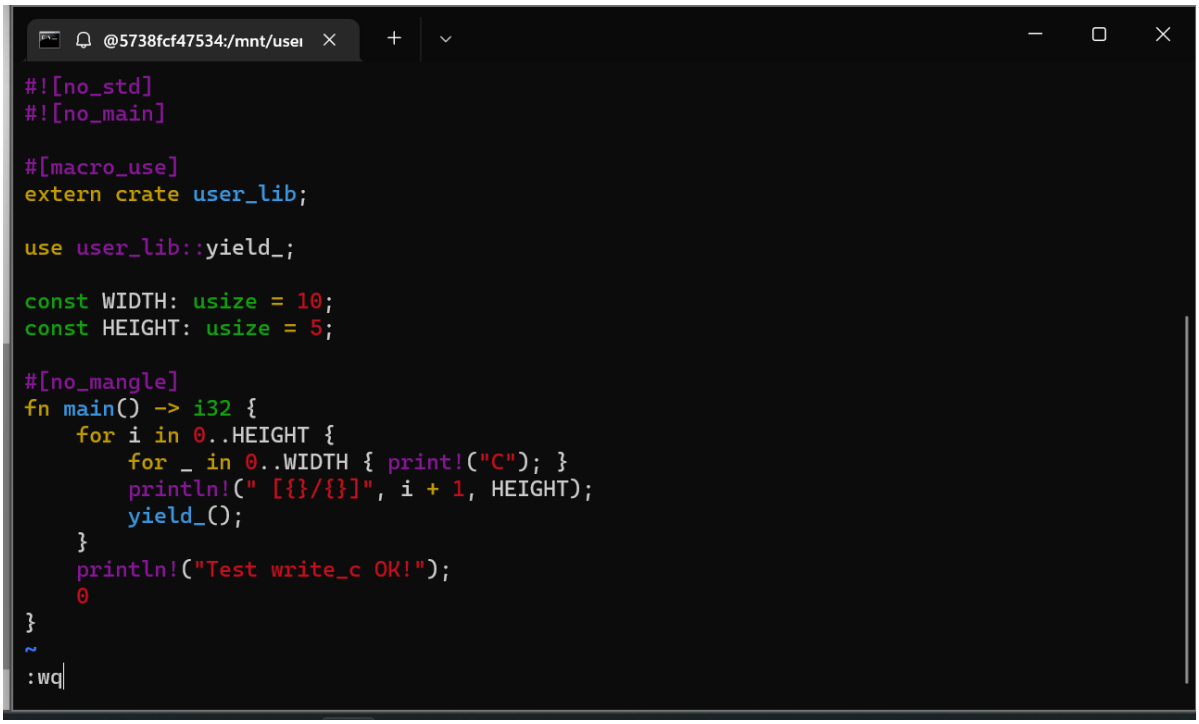
- 增加yield_用户库的封装

1.3 实现测试应用程序

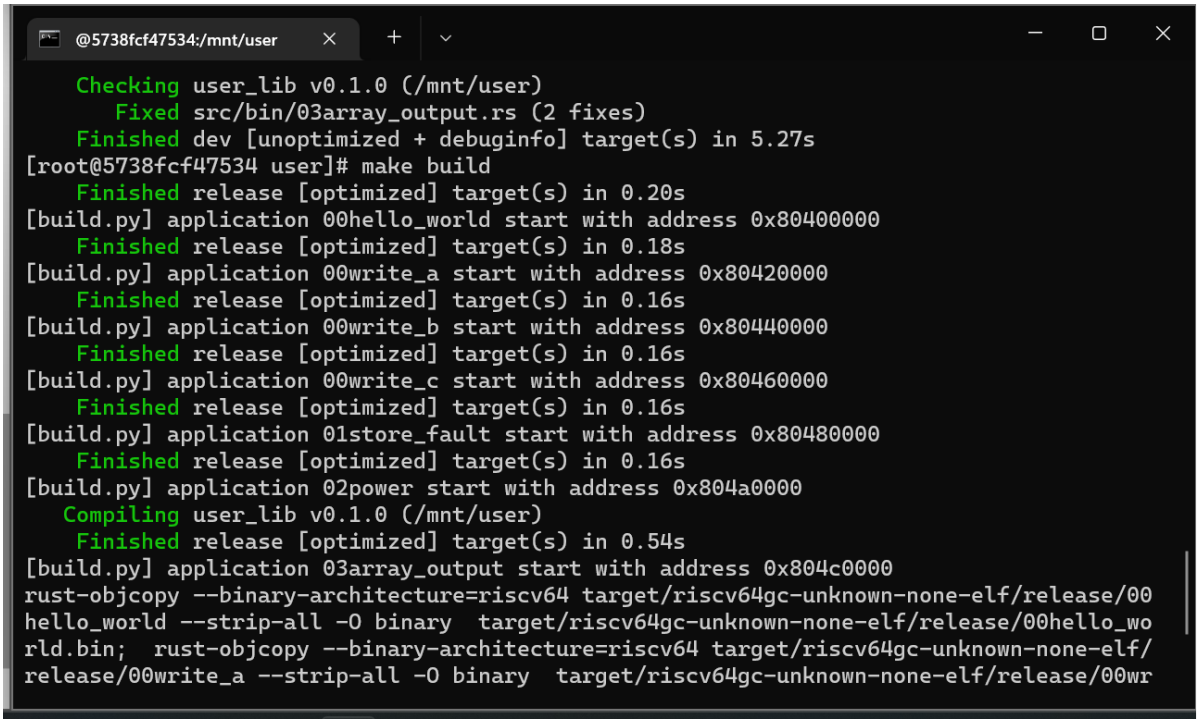
- `write_a.rs`

- `write_b.rs`

- `write_c.rs`

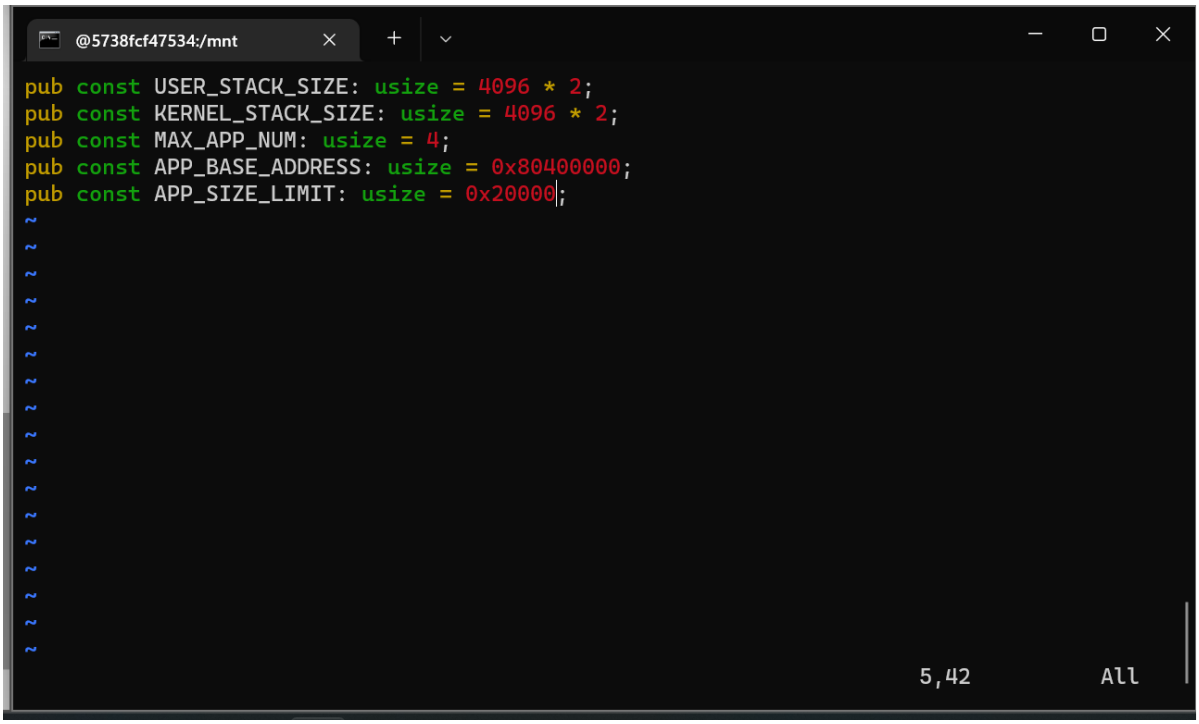


- 编译应用程序

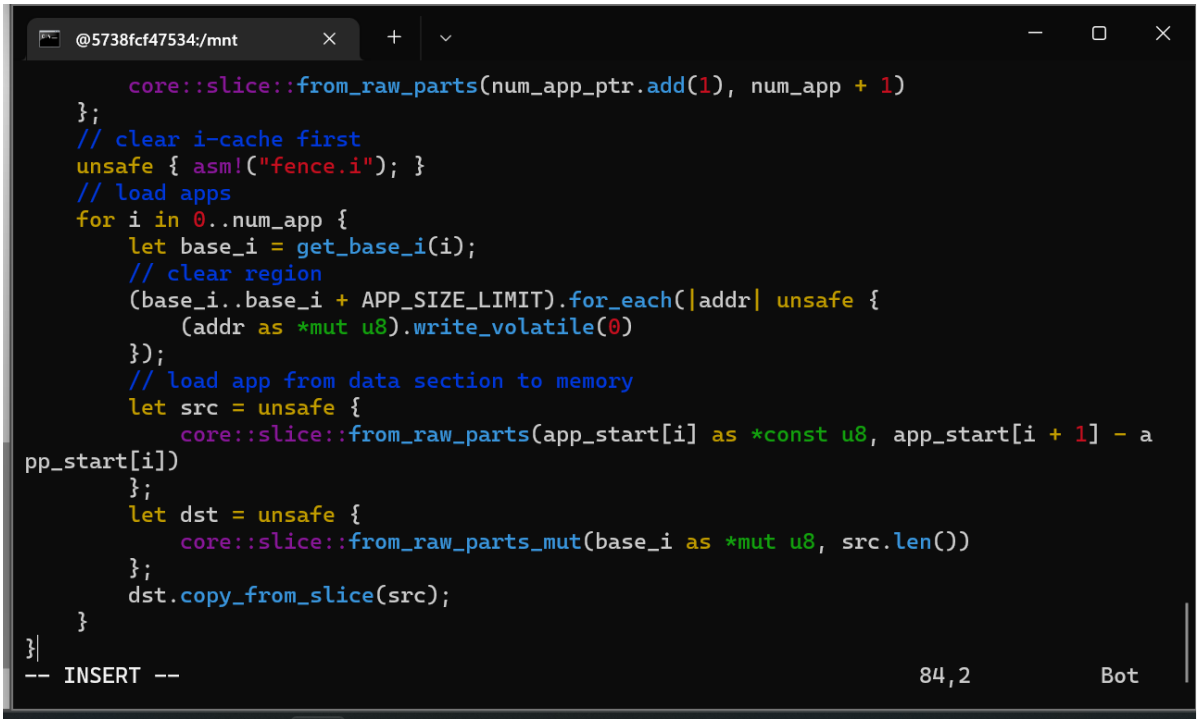


2. 多道程序的加载

- 把常量定义到`os/src/config.rs`中



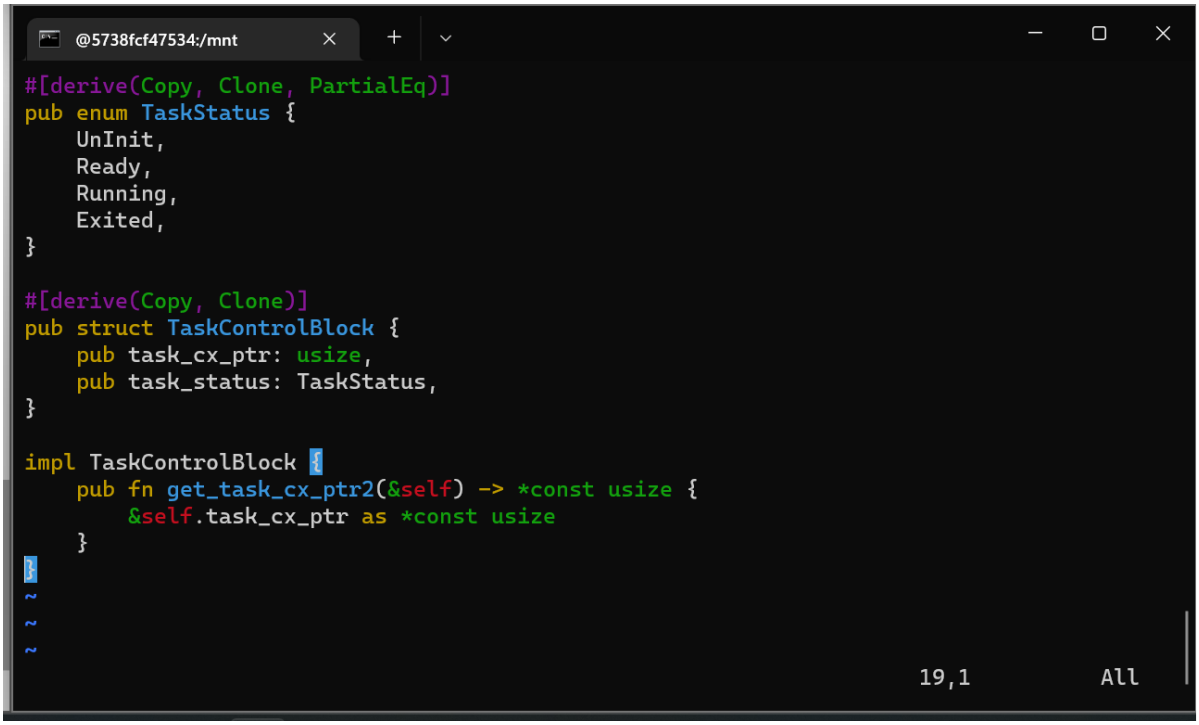
- 在`os/src/loader`中增加加载应用程序的代码



3. 任务的设计与实现

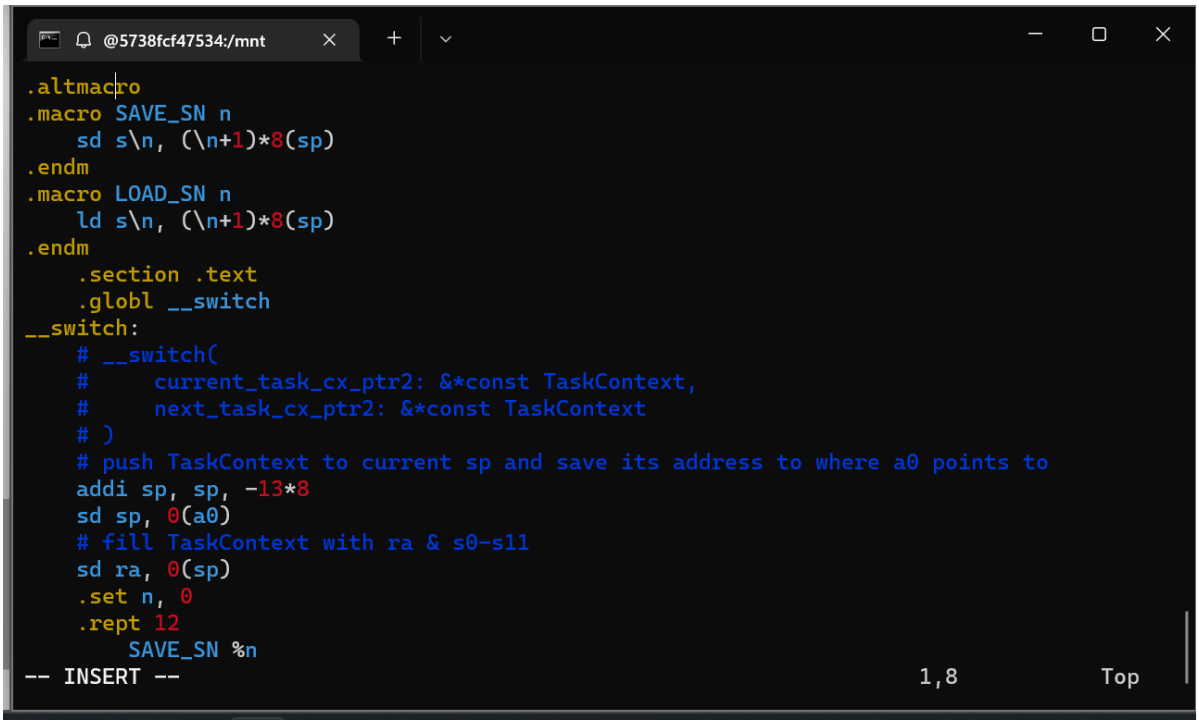
3.1 任务的上下文

3.2 任务的运行状态及任务控制块



3.3 任务切换

- 汇编代码


@5738fcf47534:/mnt
.altmacro
.macro SAVE_SN n
sd s\n, (\n+1)*8(sp)
.endm
.macro LOAD_SN n
ld s\n, (\n+1)*8(sp)
.endm
.section .text
.globl __switch
__switch:
__switch(
current_task_cx_ptr2: &*const TaskContext,
next_task_cx_ptr2: &*const TaskContext
#)
# push TaskContext to current sp and save its address to where a0 points to
addi sp, sp, -13*8
sd sp, 0(a0)
fill TaskContext with ra & s0-s11
sd ra, 0(sp)
.set n, 0
.rept 12
SAVE_SN %n
-- INSERT --
1,8
Top

- rust函数


@5738fcf47534:/mnt
use core::arch::global_asm;
global_asm!(include_str!("switch.S"));
extern "C" {
pub fn __switch(
current_task_cx_ptr2: *const usize,
next_task_cx_ptr2: *const usize
);
}
10,1
All

3.4 任务管理器

```
@5738fcf47534:/mnt
mod context;
mod switch;
mod task;

use crate::config::MAX_APP_NUM;
use crate::loader::{get_num_app, init_app_cx};
use core::cell::RefCell;
use lazy_static::*;
use switch::__switch;
use task::{TaskControlBlock, TaskStatus};

pub use context::TaskContext;

pub struct TaskManager {
    num_app: usize,
    inner: RefCell<TaskManagerInner>,
}

struct TaskManagerInner {
    tasks: [TaskControlBlock; MAX_APP_NUM],
    current_task: usize,
}
"os/src/task/mod.rs" 127L, 3360B                              1,1           Top
```

上述代码还调用了`loader`子模块的`init_app_cx`。因此，还需要在`os/src/loader.rs`增加如下代码：

```
@5738fcf47534:/mnt
        // clear region
        (base_i..base_i + APP_SIZE_LIMIT).for_each(|addr| unsafe {
            (addr as *mut u8).write_volatile(0)
        });
        // load app from data section to memory
        let src = unsafe {
            core::slice::from_raw_parts(app_start[i] as *const u8, app_start[i + 1] - a
pp_start[i])
        };
        let dst = unsafe {
            core::slice::from_raw_parts_mut(base_i as *mut u8, src.len())
        };
        dst.copy_from_slice(src);
    }
}

pub fn init_app_cx(app_id: usize) -> &'static TaskContext {
    KERNEL_STACK[app_id].push_context(
        TrapContext::app_init_context(get_base_i(app_id), USER_STACK[app_id].get_sp()),
        TaskContext::goto_restore(),
    )
}
-- INSERT --                                                  91,2          Bot
```

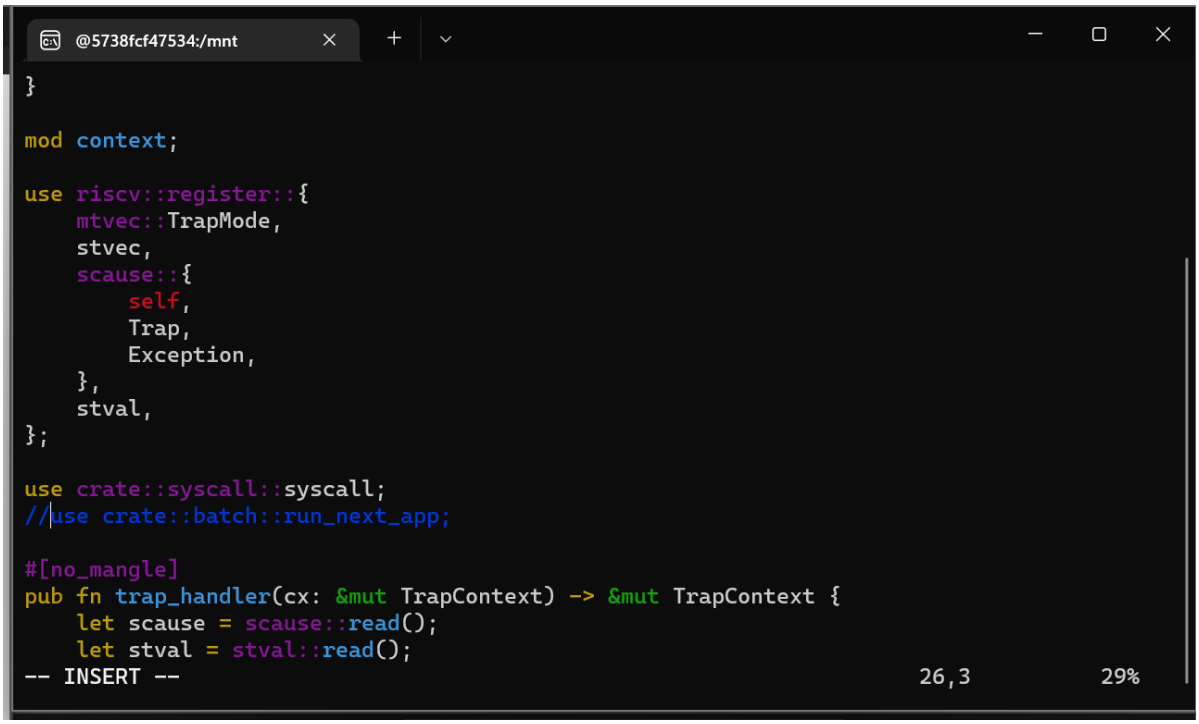
4. 实现sys_yield和sys_exit系统调用

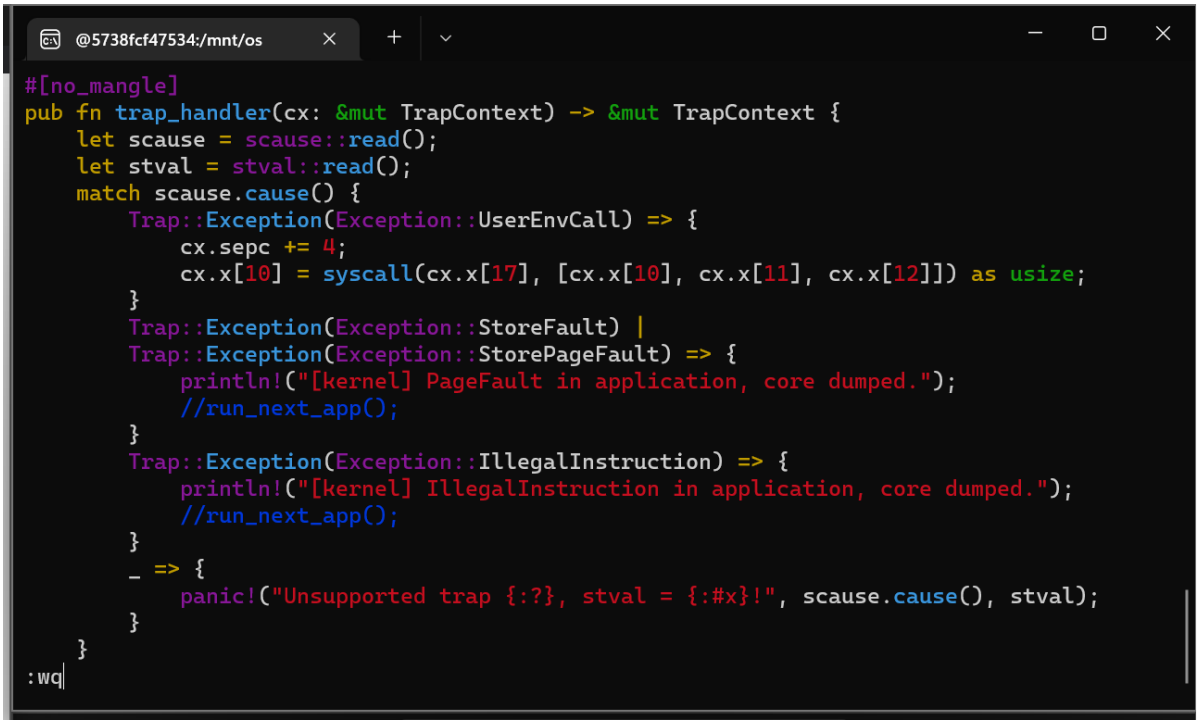
- 修改`/os/src/syscall/process.rs`

- 修改`os/src/syscall/mod.rs`

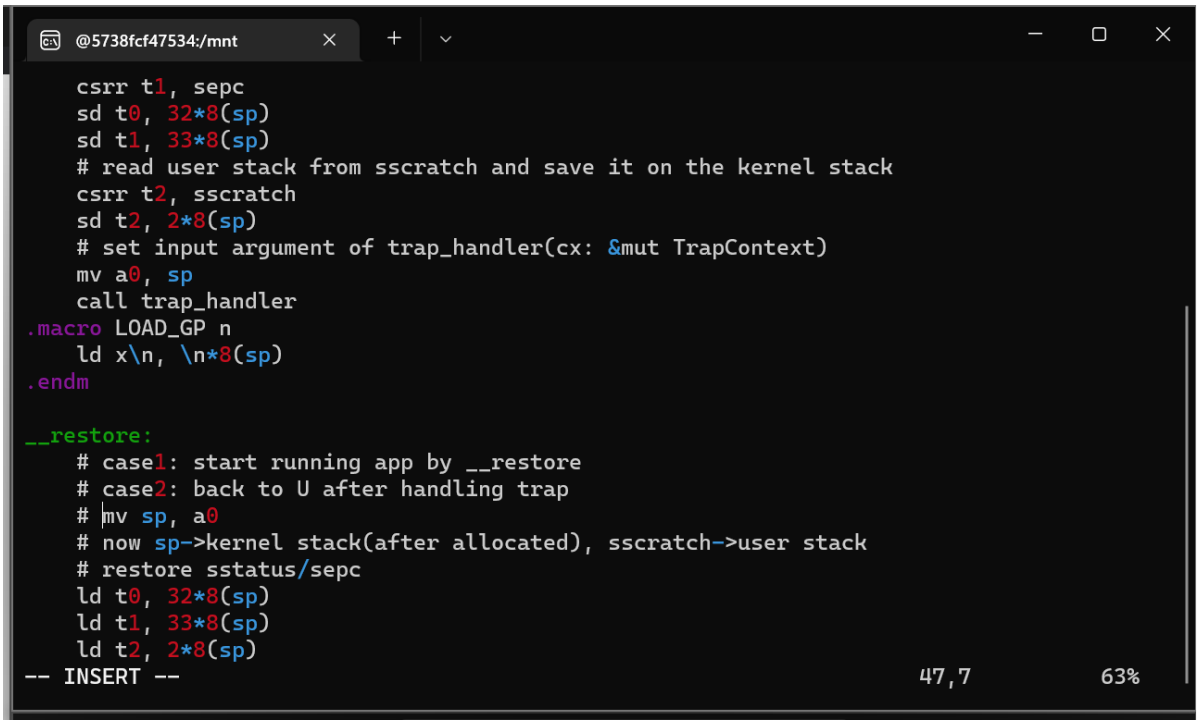
5. 修改其他部分代码

- 注释trap子模块中`run_next_app()`部分的代码





- 注释掉`trap.S`中`__restore`中的`mv sp, a0`。

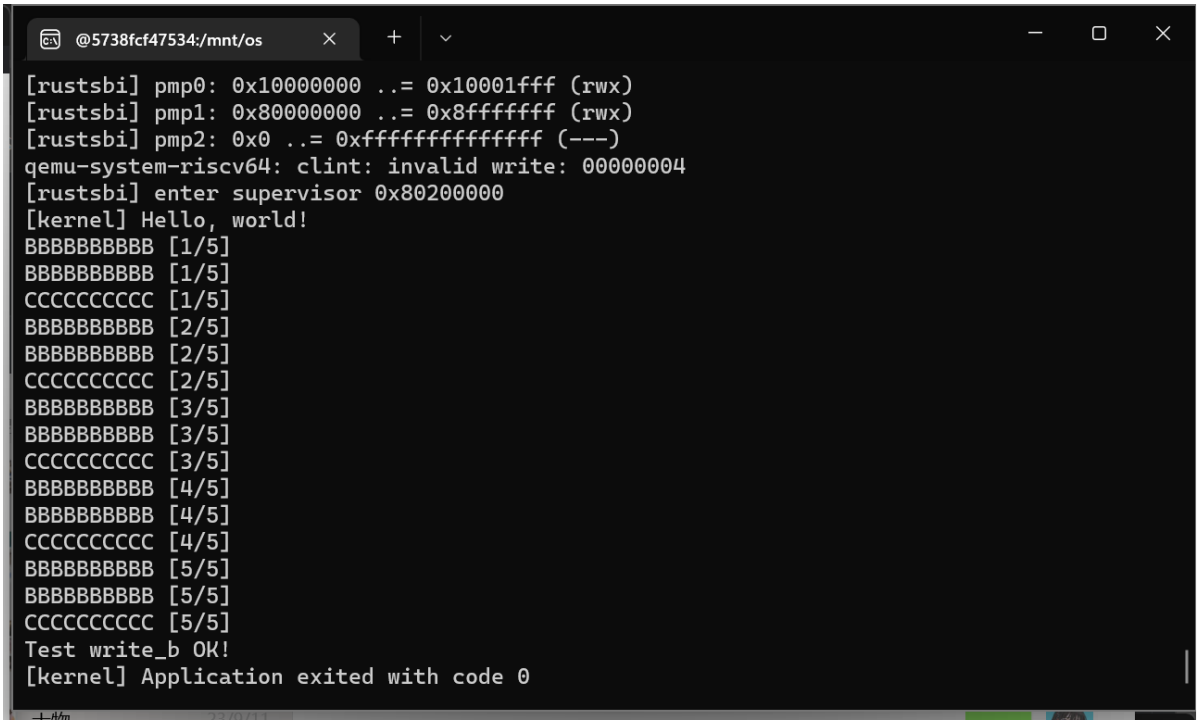


- 修改`main.rs`

6. 成功运行!

> 这里如果直接运行会报一个数组越界的错

把`user/src/bin`里面之前的程序除了本次实验的打印ABC全部删除，就好了



二. 思考问题

1. 分析应用程序是如何加载的

通过`builder.py`脚本实现将每个应用程序分配不同的内存地址用于加载，应用程序应该加载的内存地址由`os/src/loader.rs`中的`get_base_i`计算。另外，不同于批处理操作系统，多道程序操作系统所用的应用程序在内核初始化的时候就一起加载到内存中。

2. 分析多道程序如何设计和实现的

多道程序支持应用程序主动交出CPU的使用权。我们把一个计算执行过程称之为任务。一个应用程序的任务切换到另外一个应用程序的任务称为任务切换。任务切换过程中需要保存的恢复任务重新执行所需的寄存器、栈等内容称为任务的上下文。任务切换需要有任务上下文的支持。

我们通过实现在`os/src/task/context.rs`中的`TaskContext`数据结构来记录任务的上下文信息；在`os/src/task/task.rs`中实现在内核中维护任务的运行状态；在`os/src/task/switch.S`中实现任务切换的汇编代码，然后在`switch.rs`中将其封装为rust的函数；最后在`os/src/task/mod.rs`中实现一个全局的任务管理器来管理任务控制描述的应用程序。

3. 分析所实现的多道程序操作系统中的任务是如何实现的，以及它和理论课程里的进程和线程有什么区别和联系

任务实现过程和问题2中描述的方式一致

与进程线程的区别：

本实验中的多道程序仅支持应用程序主动交出CPU的使用权，而课程中的进程线程支持操作系统分配CPU资源

联系：

都支持上下文切换

三. Git提交截图