2023秋《操作系统》课程实验报告

实验六

21301161 齐家昕

1. 实验步骤

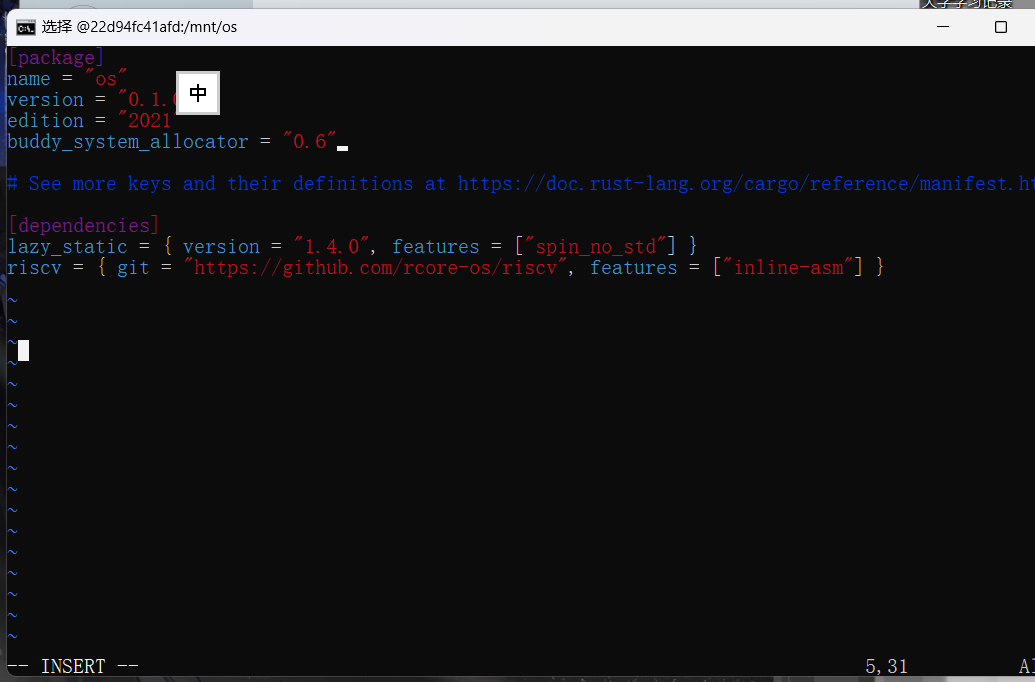
本实验的主要目的实现内存的动态申请和释放管理。

1. 在内核中支持动态内存分配

因为在操作系统内核中没有标准库的支持，我们需要利用alloc库定义的接口实现基本的动态内存分配器。由于内存分配器的实现比较复杂，直接使用已有的的伙伴分配器。

首先，在Cargo.toml配置文件中增加依赖。

buddy\_system\_allocator = "0.6"



同时，还需要在main.rs中引入alloc库的依赖，增加如下代码：

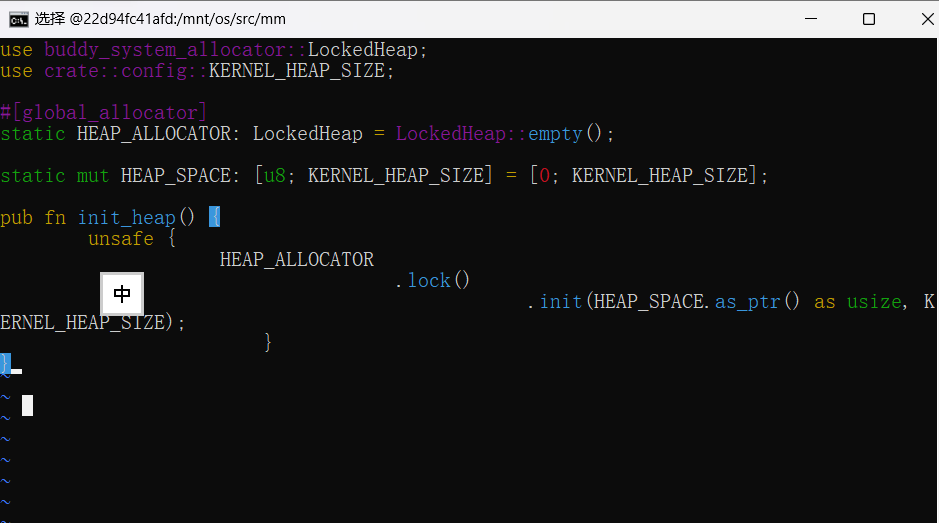
// os/src/main.rs

extern crate alloc;



然后，根据alloc留好的结构提供全局动态内存分配器：

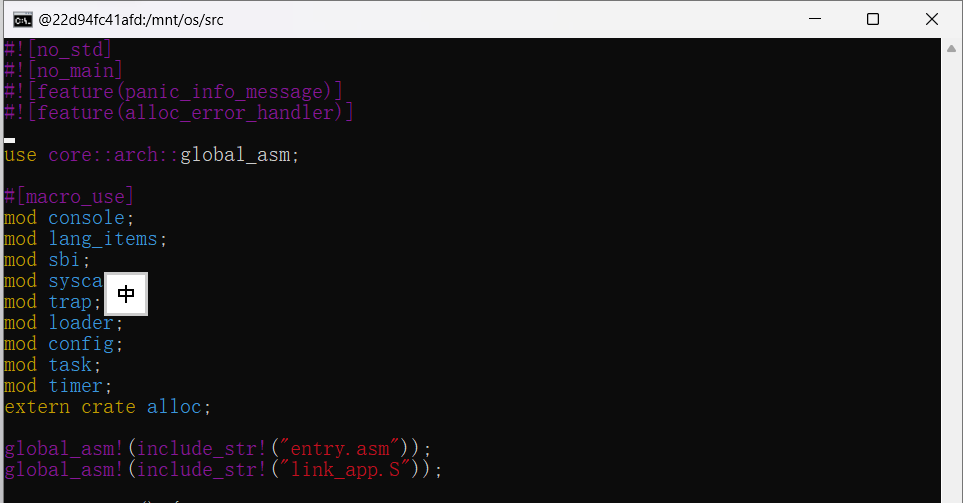
// os/src/mm/heap\_allocator.rs



同时，还需要处理动态内存分配失败的情况，具体增加如下代码：

// os/src/main.rs

#![feature(alloc\_error\_handler)]



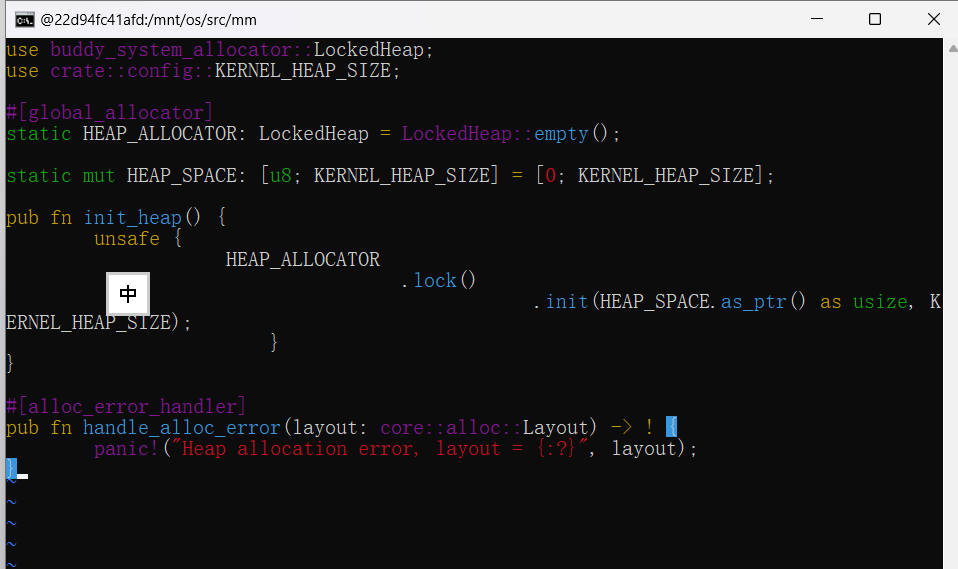
// os/src/mm/heap\_allocator.rs

#[alloc\_error\_handler]

pub fn handle\_alloc\_error(layout: core::alloc::Layout) -> ! {

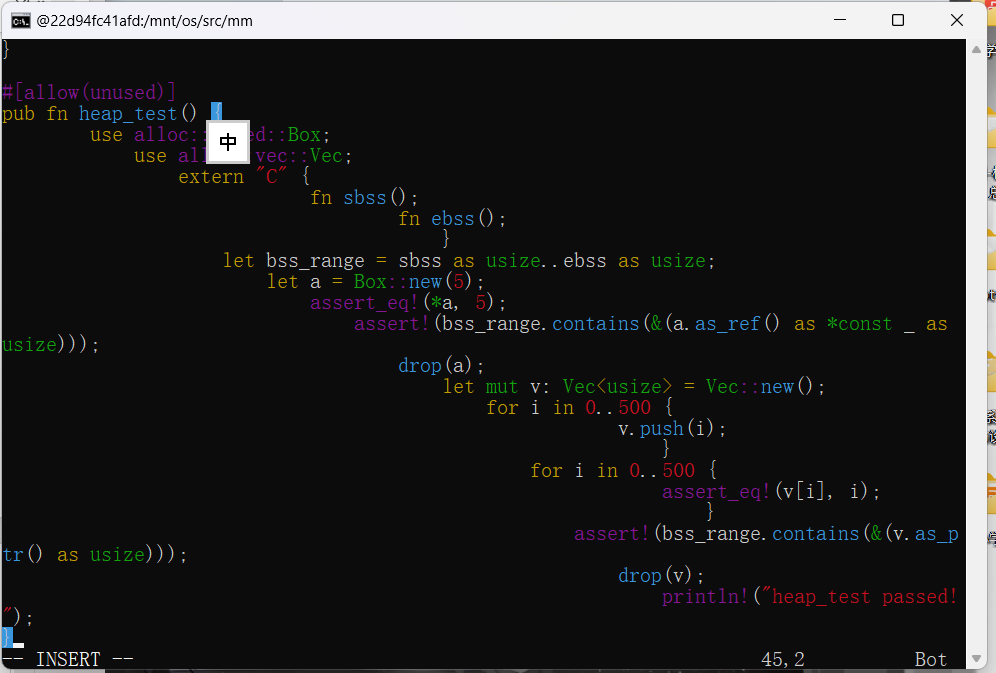
panic!("Heap allocation error, layout = {:?}", layout);

}



然后，实现测试动态内存分配。

// os/src/mm/heap\_allocator.rs

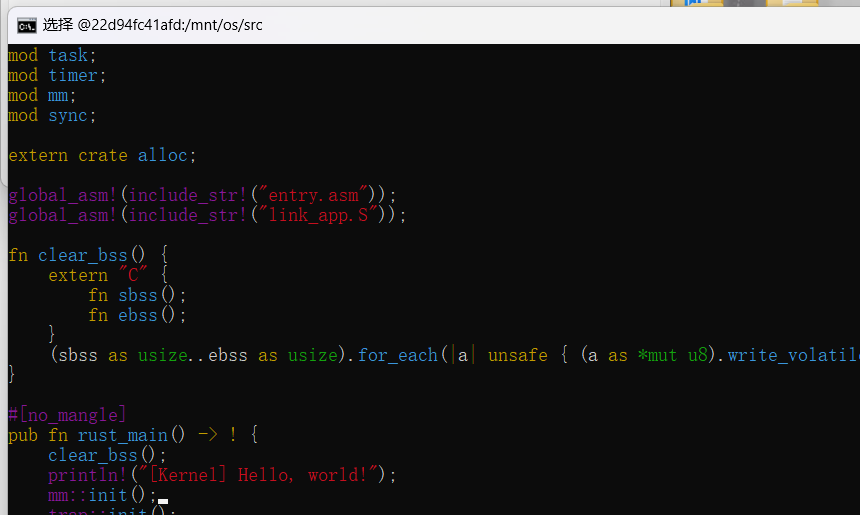


最后，修改其他部分代码，具体包括：

// os/src/main.rs

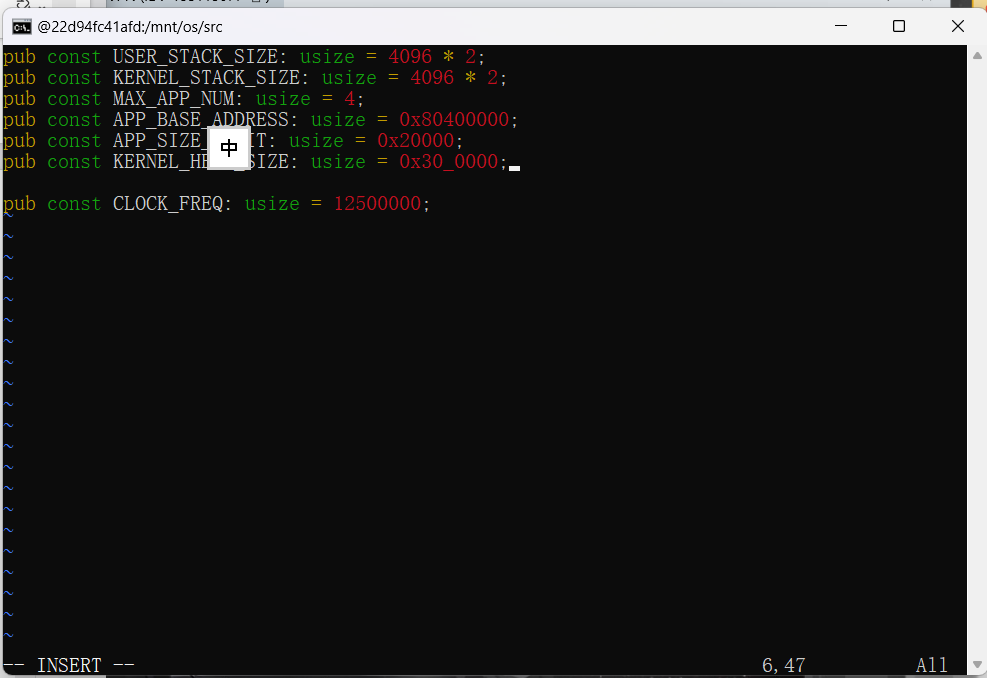
mod mm;

mm::init();

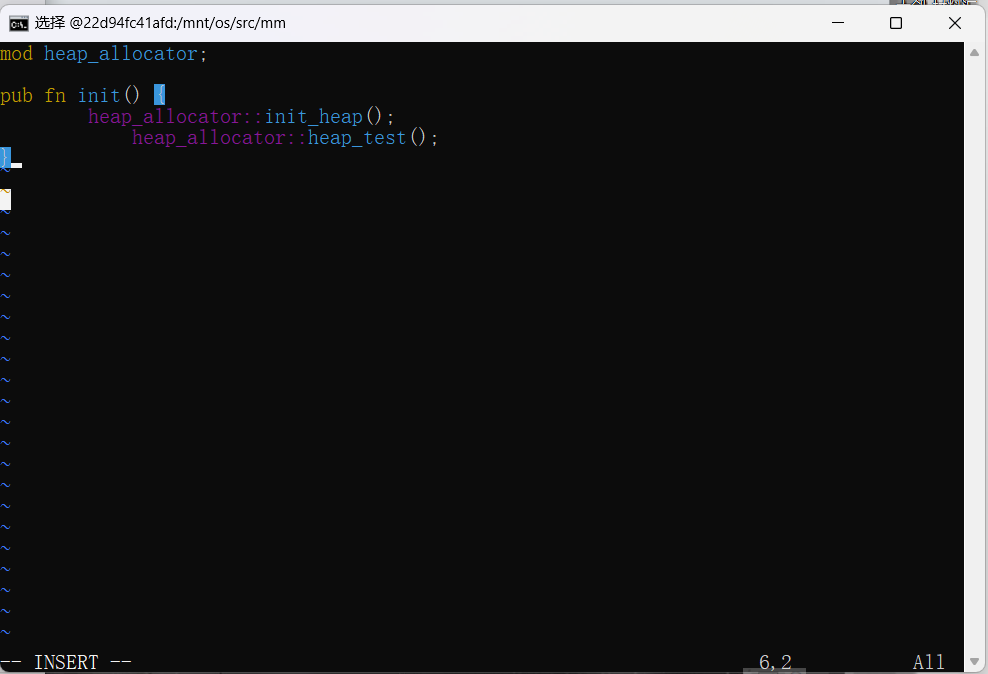


// os/src/config.rs

pub const KERNEL\_HEAP\_SIZE: usize = 0x30\_0000;



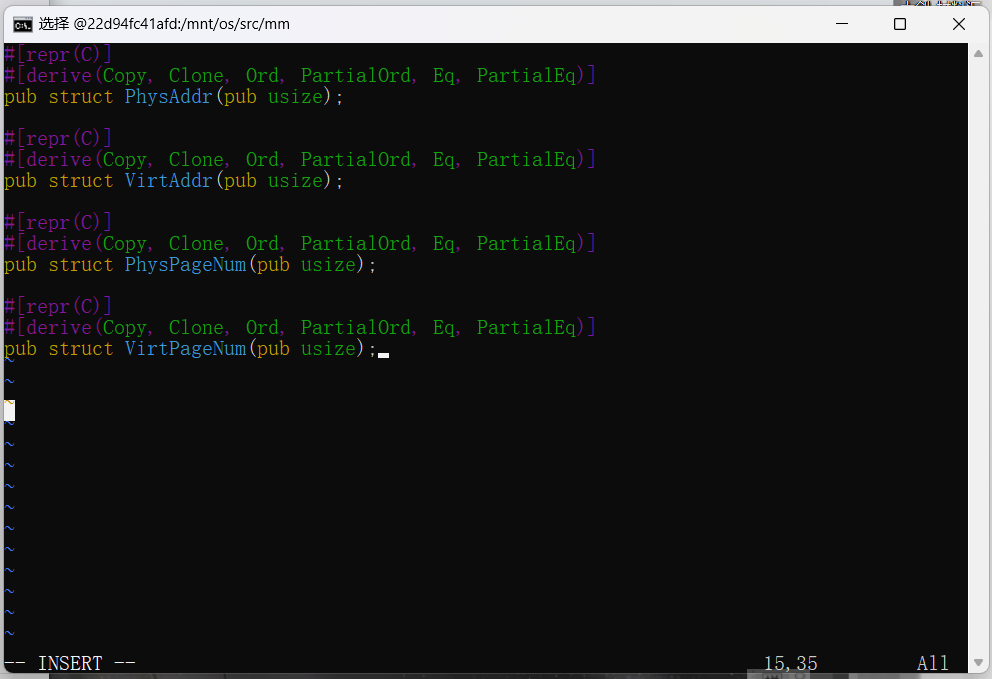
// os/src/mm/mod.rs



2. 实现虚拟地址与物理地址的基本定义

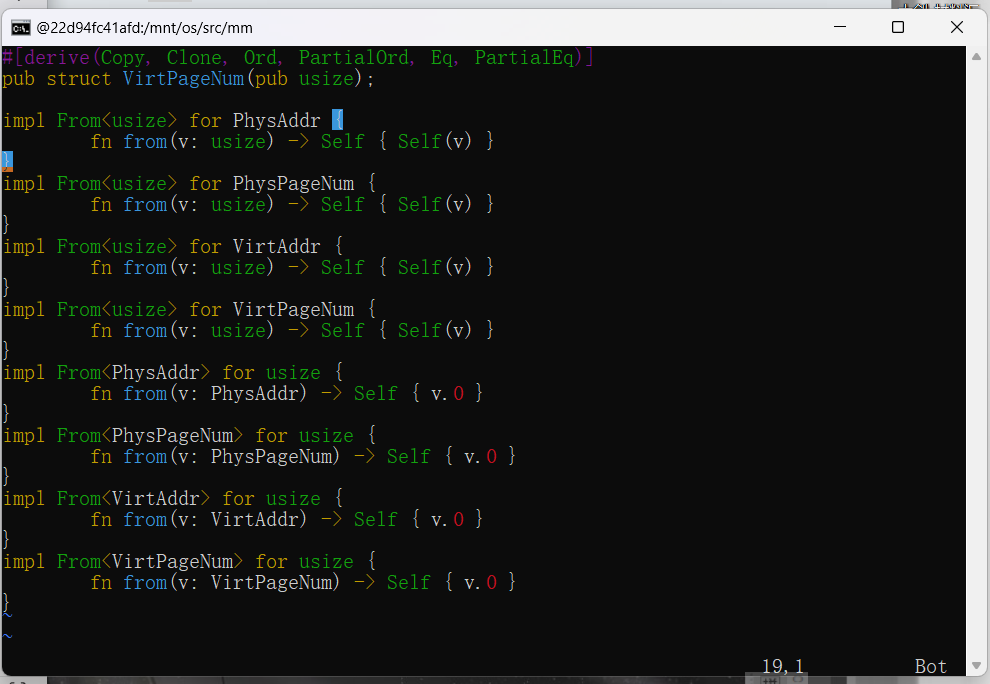
首先，定义所需要的基本数据结构，包括物理地址、虚拟地址、物理页号、虚拟页号。定义在os/src/mm/address.rs中，具体如下：

// os/src/mm/address.rs



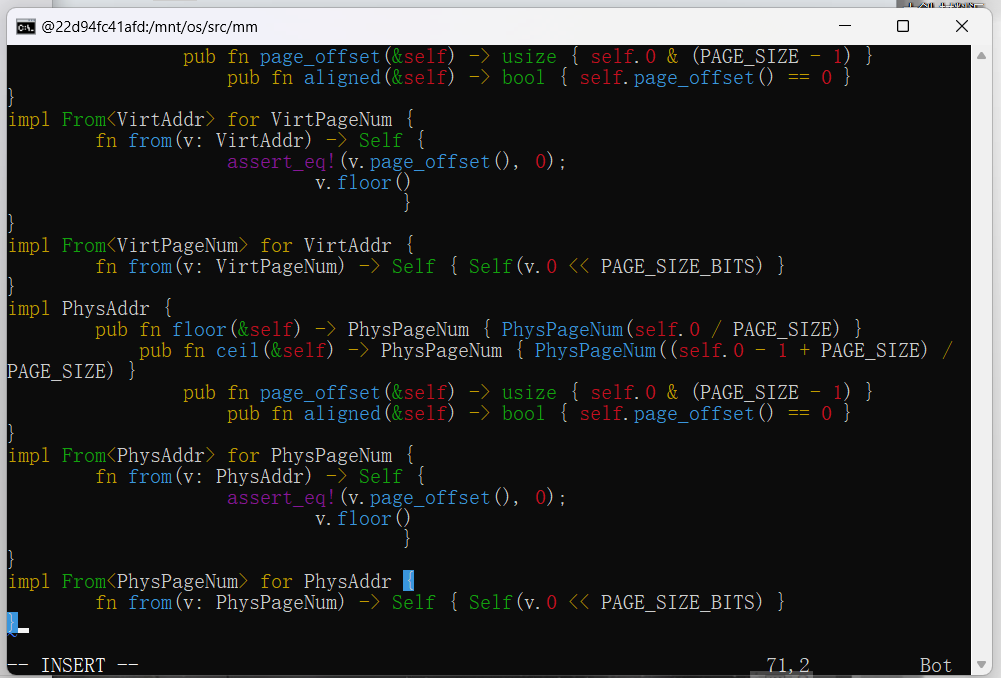
同时，还需要实现上述这些类型和usize之间的相互转换。

//os/src/mm/address.rs

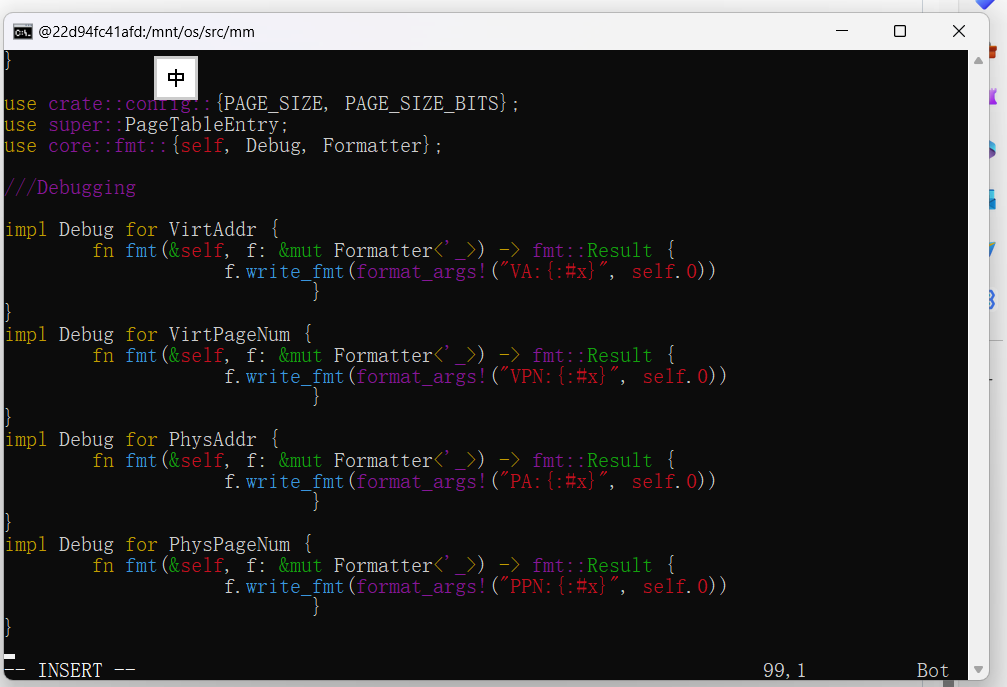


其次，实现地址和页号之间的相互转换。具体增加如下代码：

//os/src/mm/address.rs



最后，实现查询索引等其他内容。



3. 定义页表项数据结构

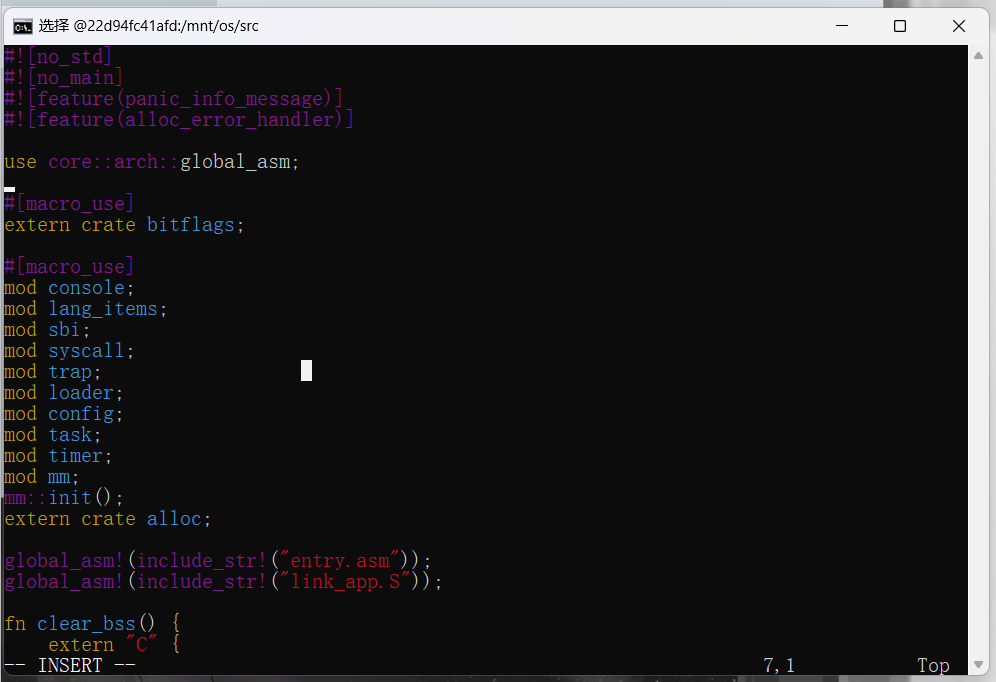
定义页表项之前需要了解SV39分页模式下页表项每一位的含义。为了实现页表项的定义，我们需要使用Rust中常用的比特标记位的crate bitflags。

首先，实现页表项中的标志位PTEFlags。

// os/src/main.rs

#[macro\_use]

extern crate bitflags;



// os/src/mm/page\_table.rs

use bitflags::\*;

bitflags! {

pub struct PTEFlags: u8 {

const V = 1 << 0;

const R = 1 << 1;

const W = 1 << 2;

const X = 1 << 3;

const U = 1 << 4;

const G = 1 << 5;

const A = 1 << 6;

const D = 1 << 7;

}

}

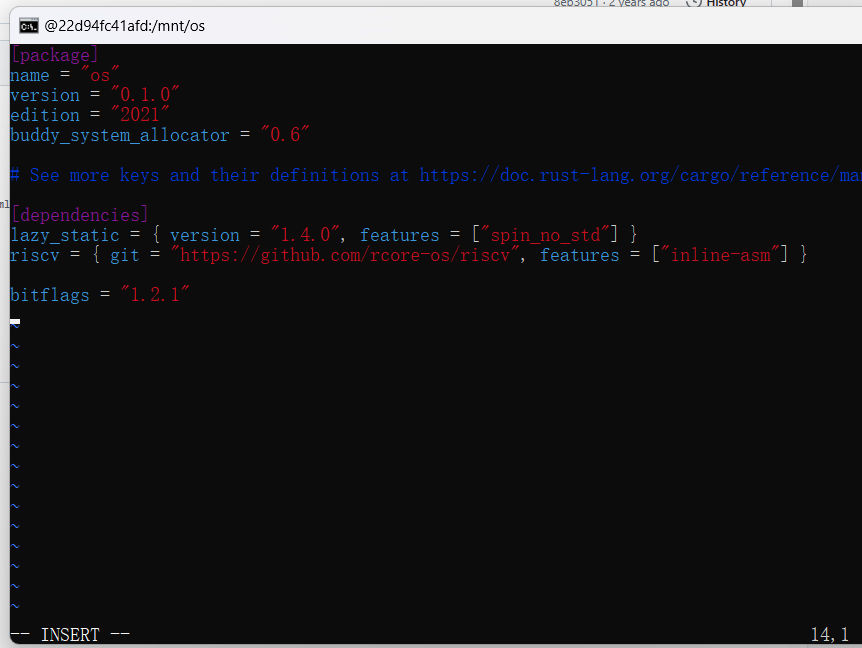


同时，需要在配置文件中增加bitflgs的依赖。

# os/Cargo.toml

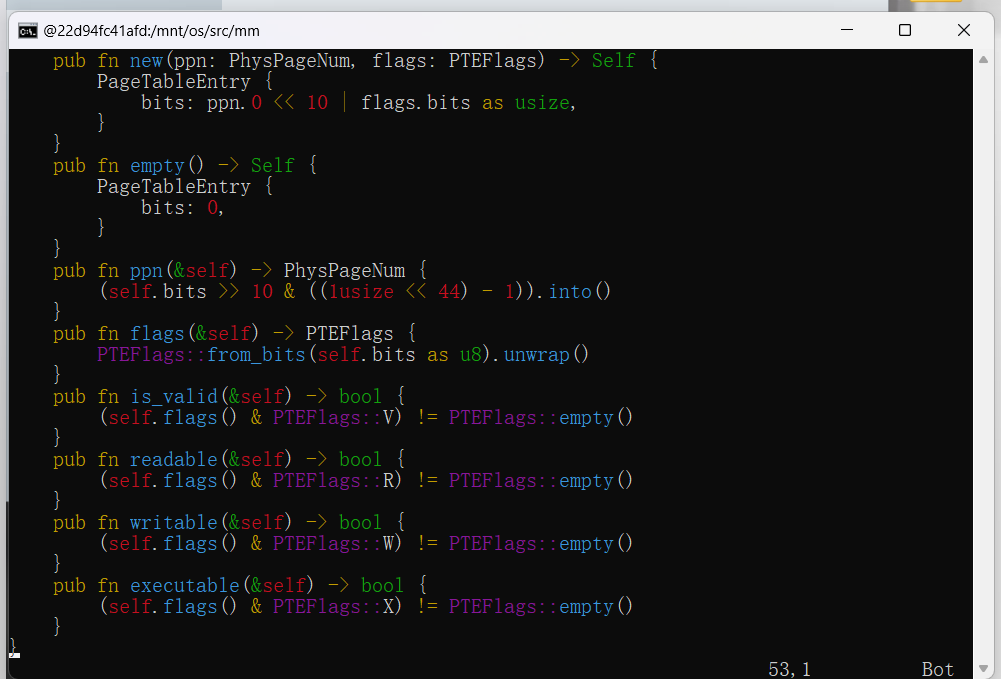
[dependencies]

bitflags = "1.2.1"



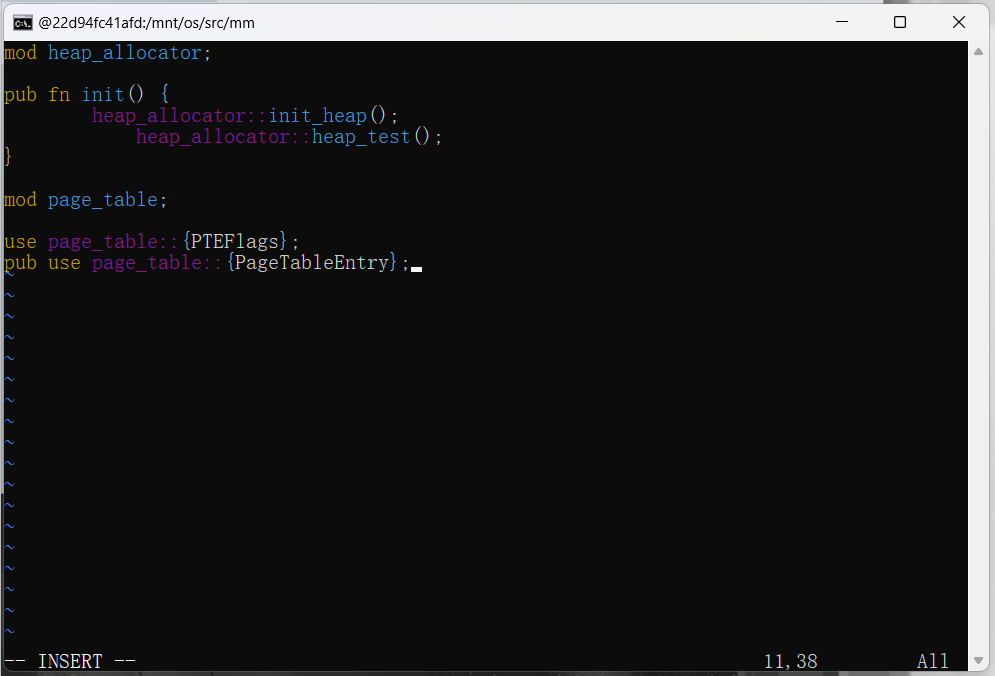
接着实现PageTableEntry。

//os/src/mm/page\_table.rs



修改/src/mm/mod.rs增加如下内容：

mod page\_table;



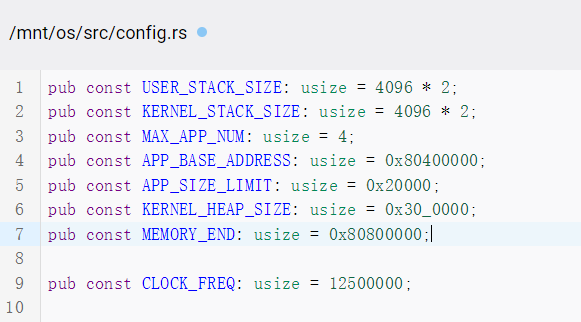
4. 实现物理帧的管理与分配

（1）设置物理内存的终止地址

在os/src/linker.ld中，ekernel确定了内核数据的终止地址，该地址之后的物理内存都是可用的。我们通过在config子模块设置参数来限制物理内存的大小。

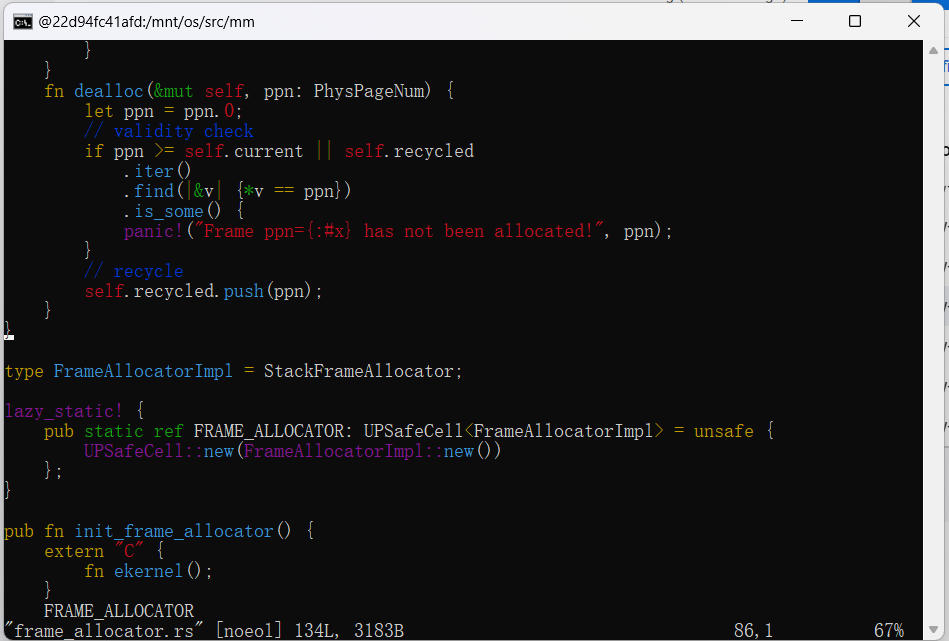
// os/src/config.rs

pub const MEMORY\_END: usize = 0x80800000;



（2）实现物理帧管理

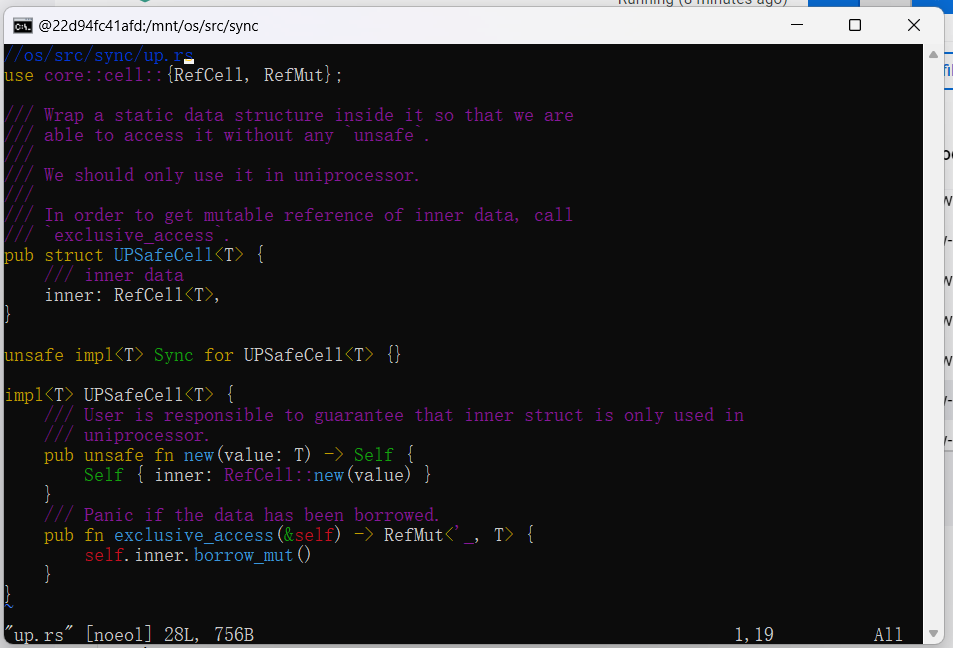
具体实现在os/src/mm/frame\_allocator.rs中，具体代码如下：



（3）增加sync模块

此外，还需要增加sync模块，实现UPSafeCell。

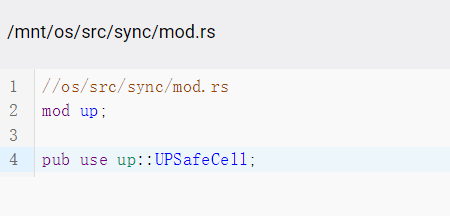
//os/src/sync/up.rs



//os/src/sync/mod.rs

mod up;

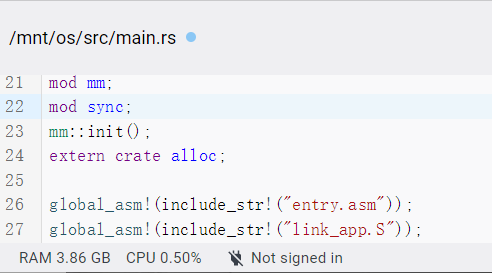
pub use up::UPSafeCell;



修改main.rs，增加对sync模块：

// os/src/main.rs

mod sync;



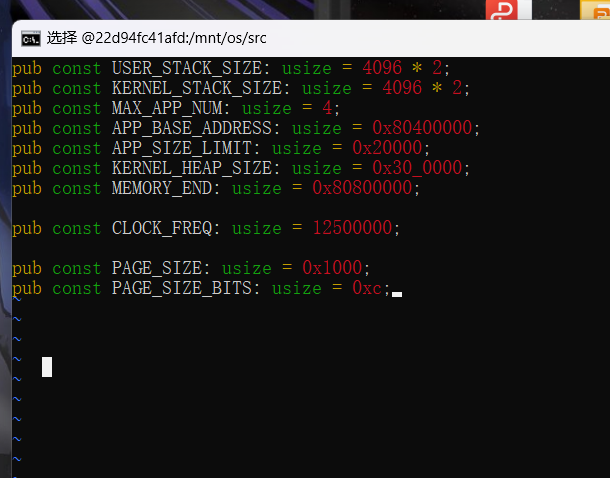
（4）物理帧管理测试

最后，为了实现物理帧管理的测试，我们还需要修改如下部分的代码：

//os/src/config.rs

pub const PAGE\_SIZE: usize = 0x1000;

pub const PAGE\_SIZE\_BITS: usize = 0xc;



//os/src/mm/mod.rs

mod address;

mod frame\_allocator;

use address::{VPNRange, StepByOne};

pub use address::{PhysAddr, VirtAddr, PhysPageNum, VirtPageNum};

pub use frame\_allocator::{FrameTracker, frame\_alloc};

pub fn init() {

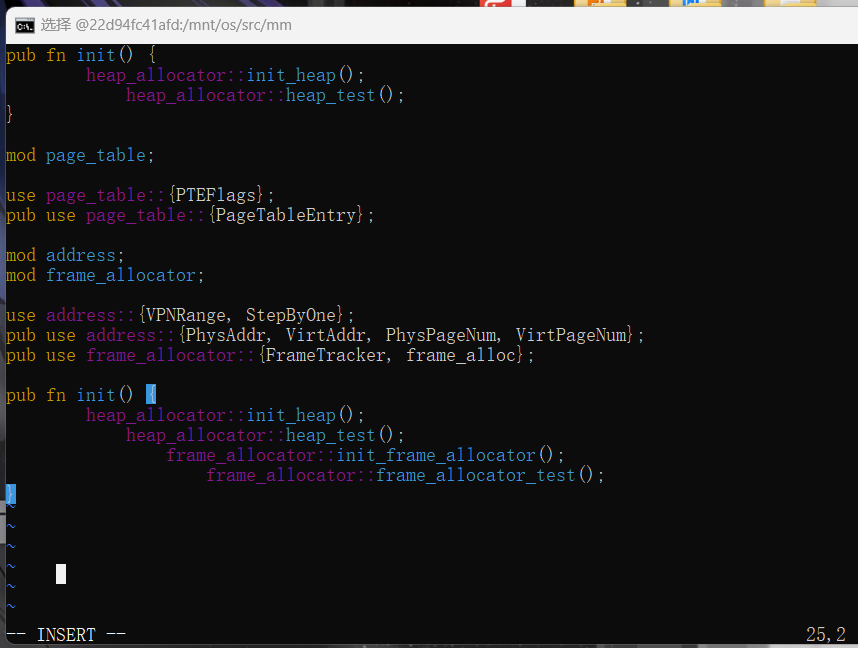
heap\_allocator::init\_heap();

heap\_allocator::heap\_test();

frame\_allocator::init\_frame\_allocator();

frame\_allocator::frame\_allocator\_test();

}

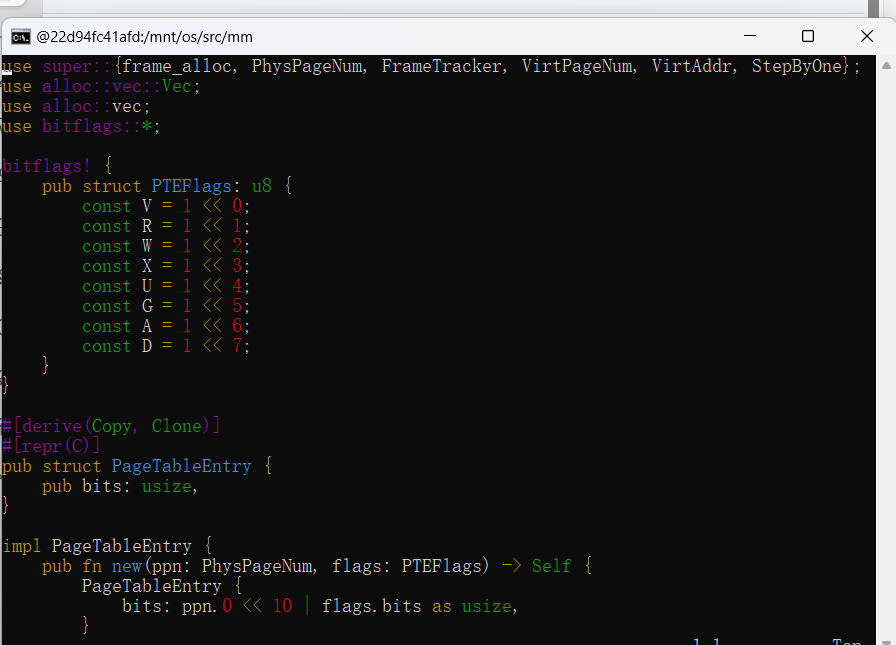


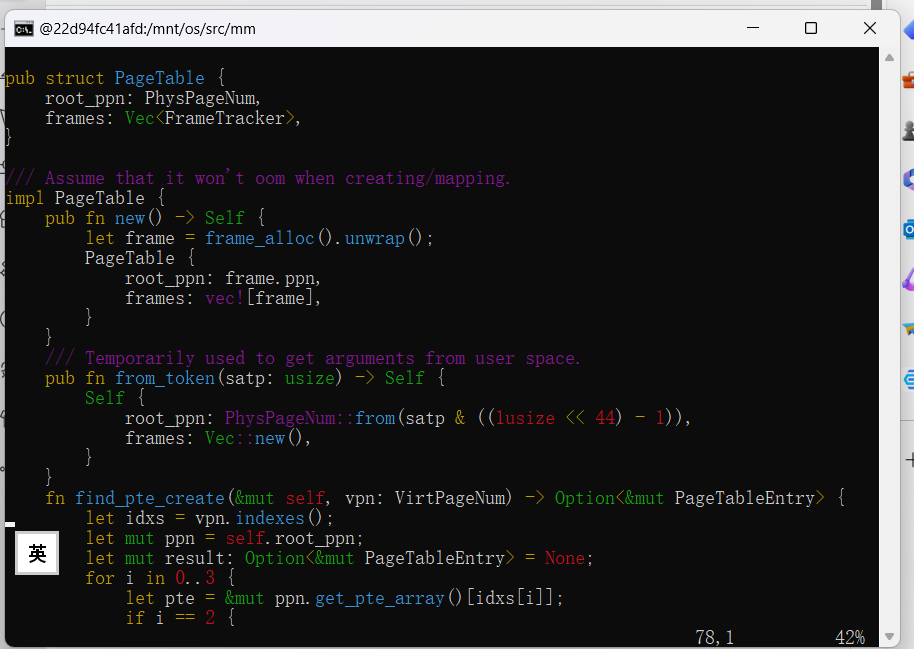
执行make run确认物理帧分配测试是否成功。

5. 多级页表管理

实现页表的基本数据结构。还需要能够建立和拆除虚实地址之间的映射关系。为了方便后续的实现，还提供了手动查询页表的方法。

// os/src/mm/page\_table.rs





6. 内核与应用的地址空间

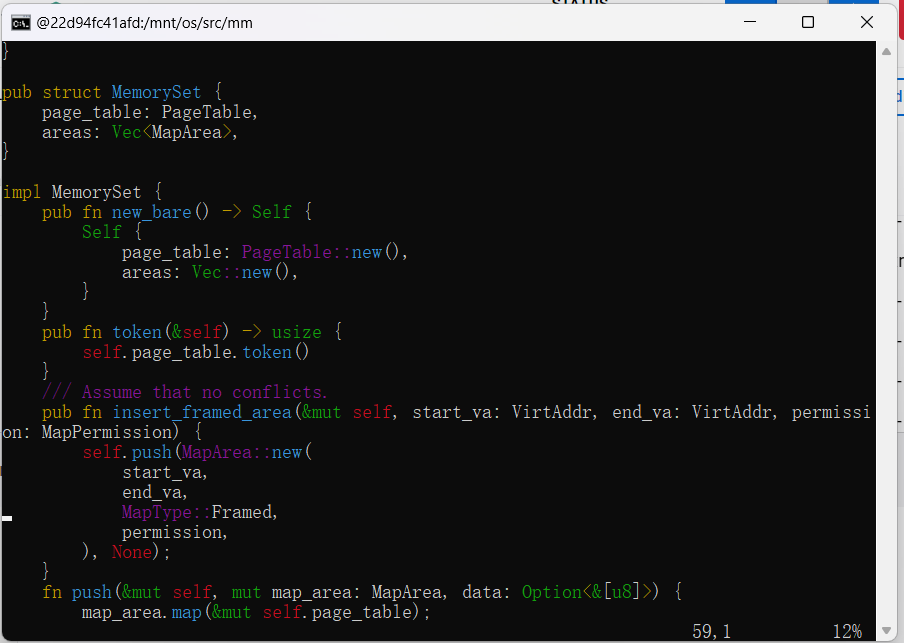
内核与应用地址空间的实现在os/src/mm/memory\_set子模块。

（1）实现地址空间抽象

首先，以逻辑段MapArea描述一段连续地址的虚拟内存。其中VPNRange是一段虚拟页号的连续空间。在address子模块中实现。接着，用MapType描述逻辑段内所有虚拟页号映射到物理页帧的方式。同时，利用MapPermisssion控制逻辑段的访问方式，其是页表项标志位PTEFlags的子集。然后，我们就可以实现地址空间了，也就是一系列有关联的逻辑段，用MemorySet来表示。

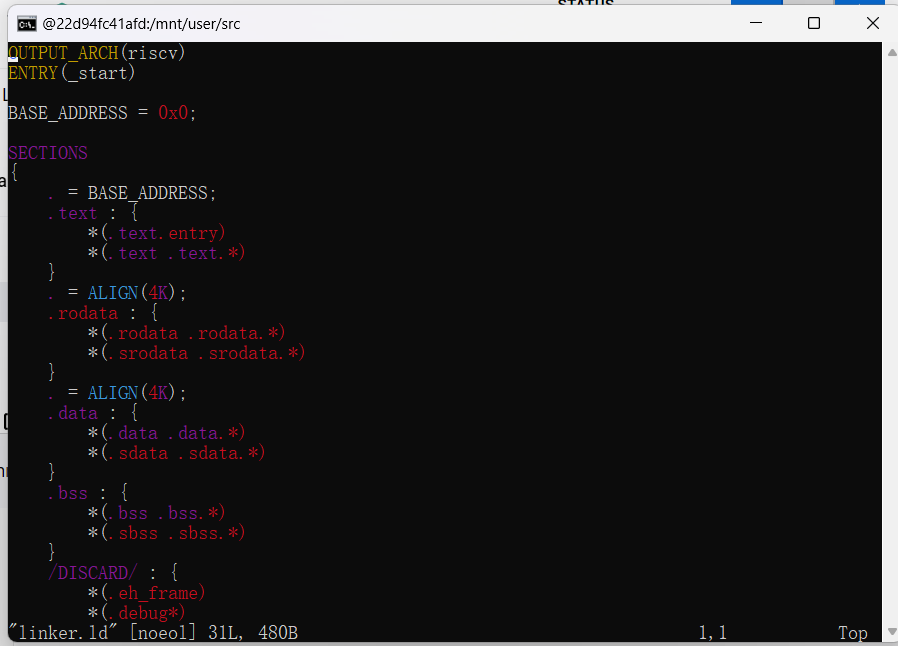
（2）实现内核地址空间

实现创建内核地址空间的方法new\_kernel。



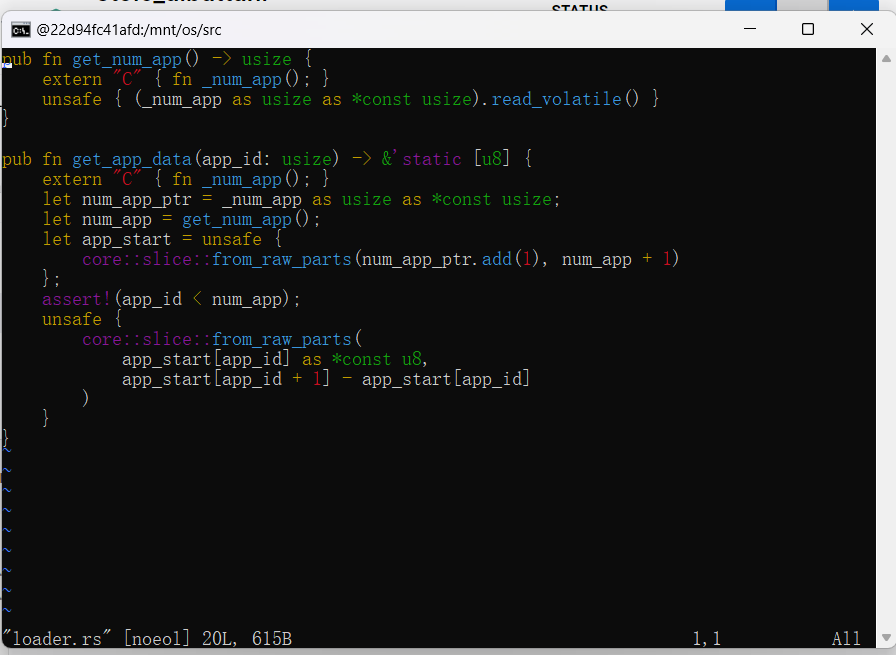
（3）实现应用地址空间

因为应用程序是动态加载了，所以所有应用使用同一个链接脚本。特别注意将BASE\_ADDRESS修改为0x0。



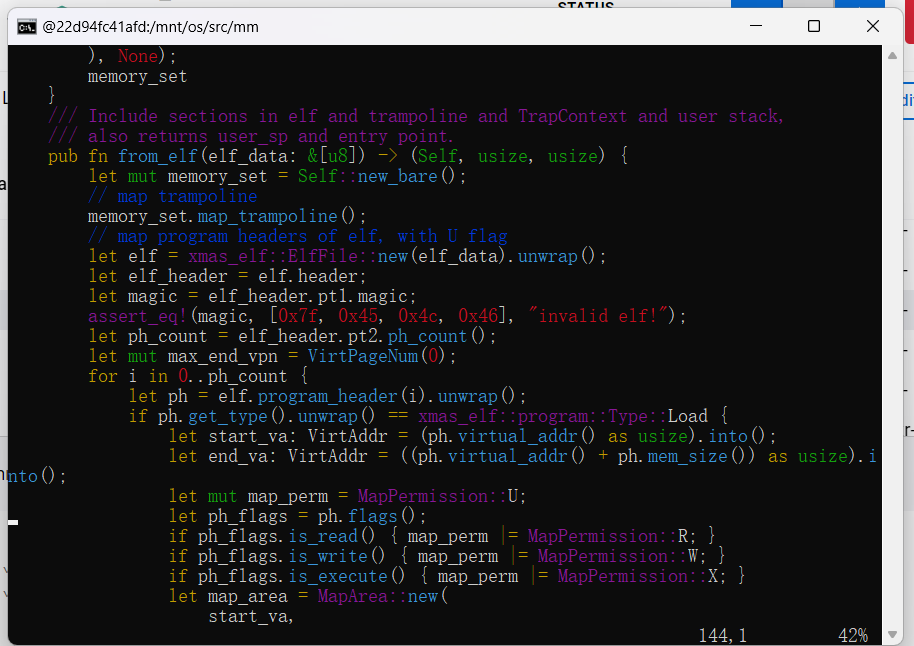
因为可以直接使用ELF格式的可执行文件了，因此我们修改精简loader子模块。

// os/src/loader.rs



同时，还需要解析ELF格式的数据，从而得到一个完整的应用地址空间。

// os/src/mm/memory\_set.rs



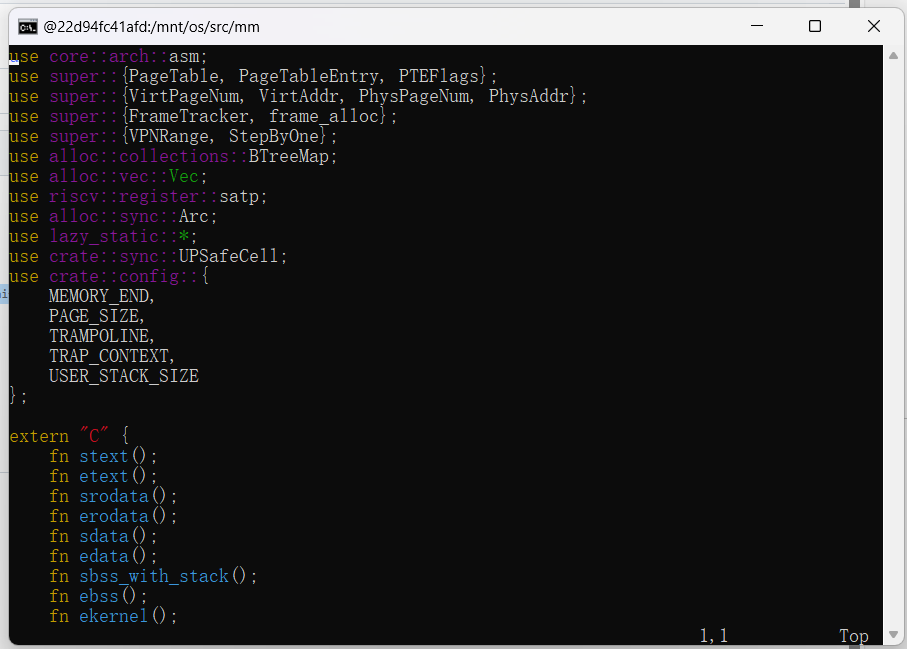
因为使用xmas-elf，所以需要在配置文件Cargo.toml增加依赖。

// os/Cargo.toml

xmas-elf = "0.7.0"

最后，需要注意实现memory\_set子模块，还需要增加如下代码：

//os/src/mm/memory\_set.rs

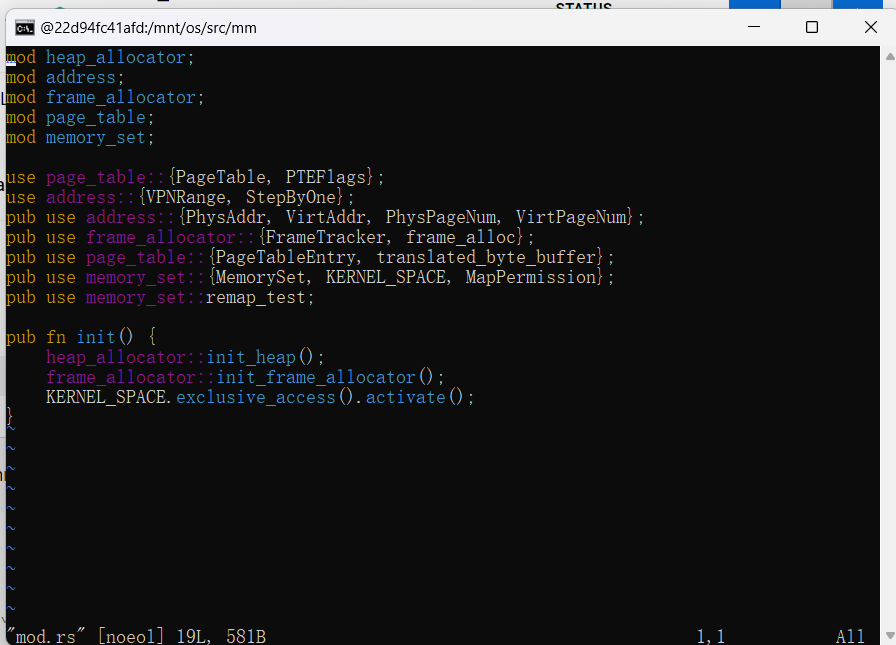


7. 实现基于地址空间的分时多任务

（1）建立基于分页模式的虚拟地址空间

首先，创建内核地址空间。然后，在rust\_main中进行内存管理子系统的初始化。

// os/src/mm/mod.rs



接着，检查内核地址空间的多级页表设置。

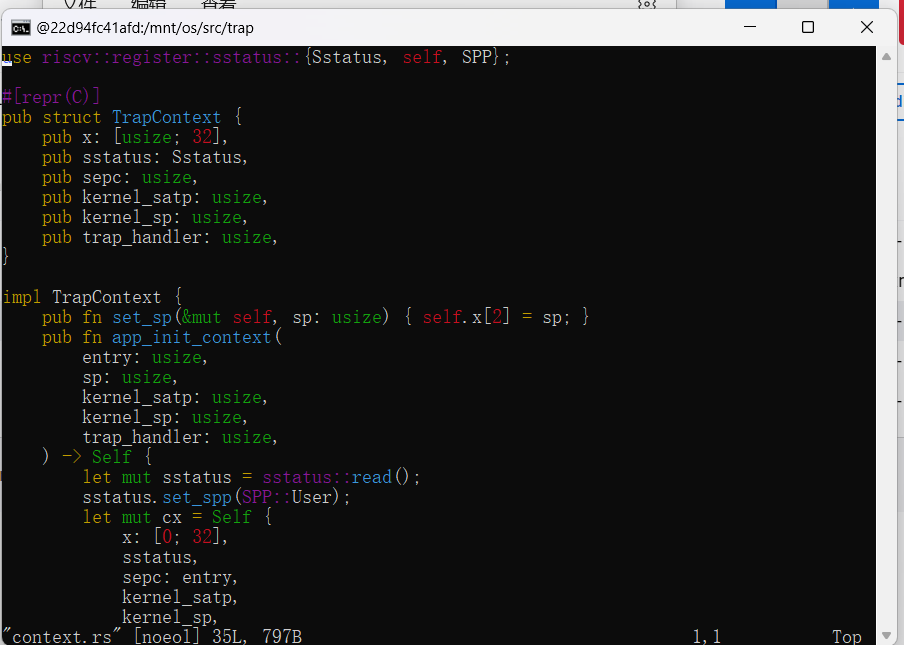
// os/src/mm/memory\_set.rs



1. 实现跳板机制

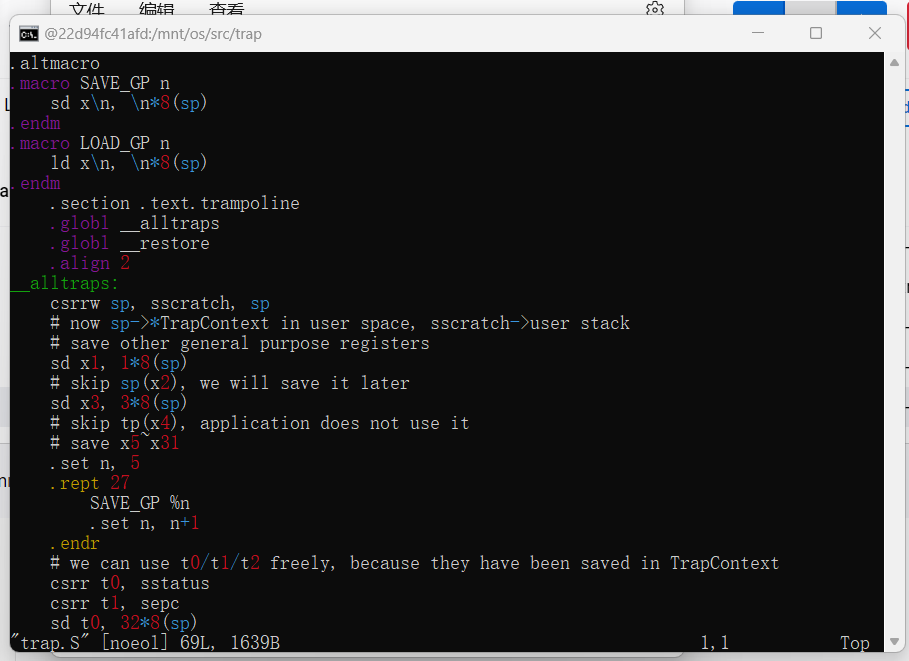
首先，扩展Trap上线文。

//os/src/trap/context.rs



接着，实现地址空间的切换。

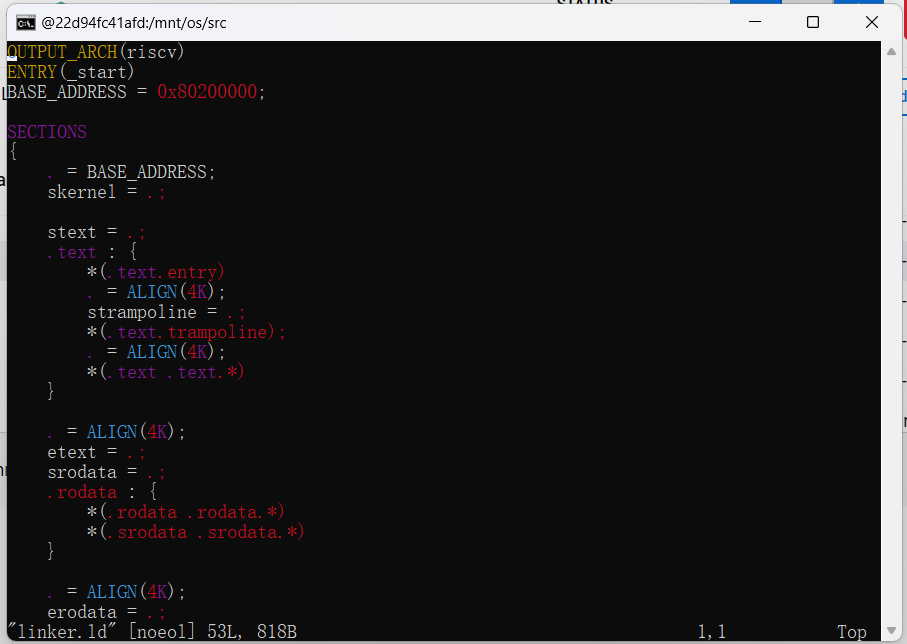
//os/src/trap/trap.S



然后，建立跳板页面。

将 trap.S 中的整段汇编代码放置在 .text.trampoline 段，并在调整内存布局的时候将它对齐到代码段的一个页面中。

# os/src/linker.ld

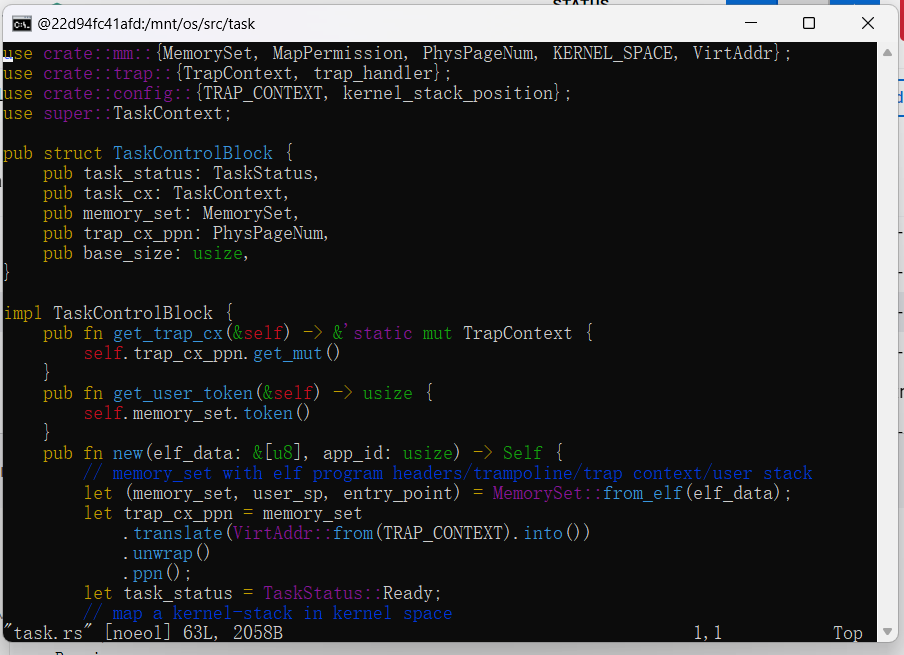


因为这个页面的汇编代码在执行的时候会进行地址空间的切换，所以这个页面被称为跳板页面。

（3）加载和执行应用程序

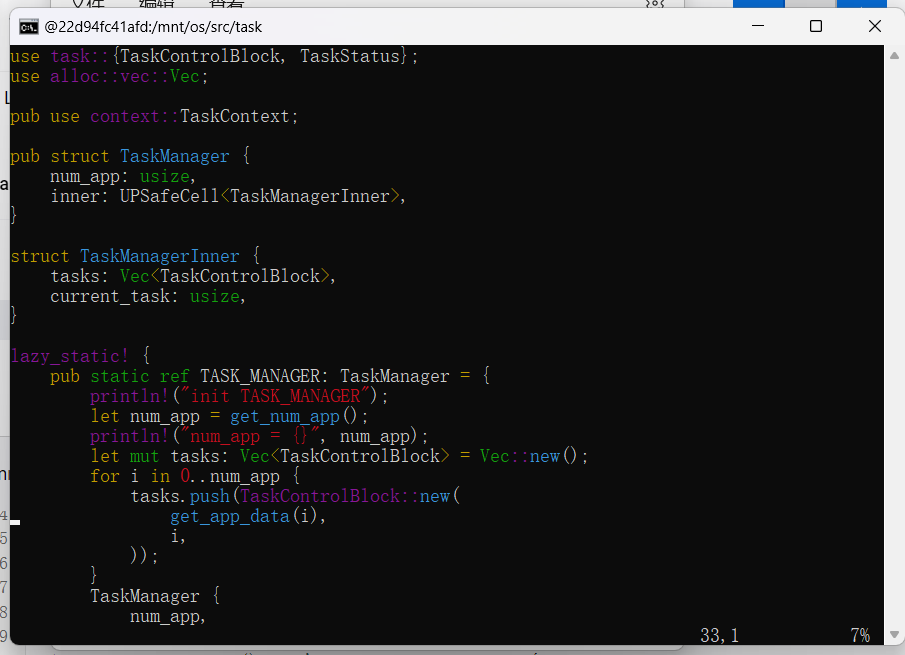
首先，修改任务子模块，并更新任务控制块的管理。

// os/src/task/task.rs

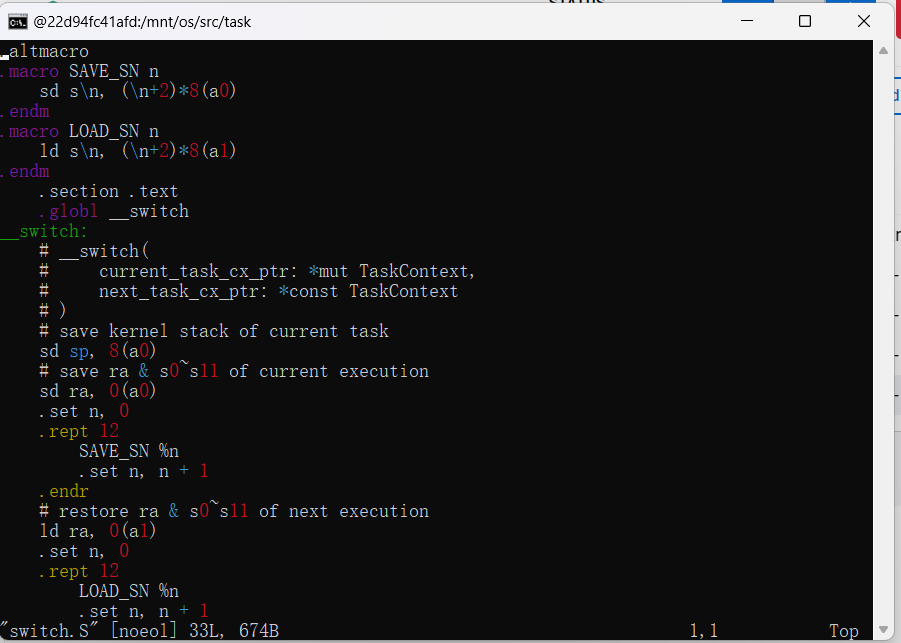


在内核初始化的时候，需要将所有的应用程序加载到全局应用管理器中。同时，也要修改TaskManager的实现。

//os/src/task/mod.rs

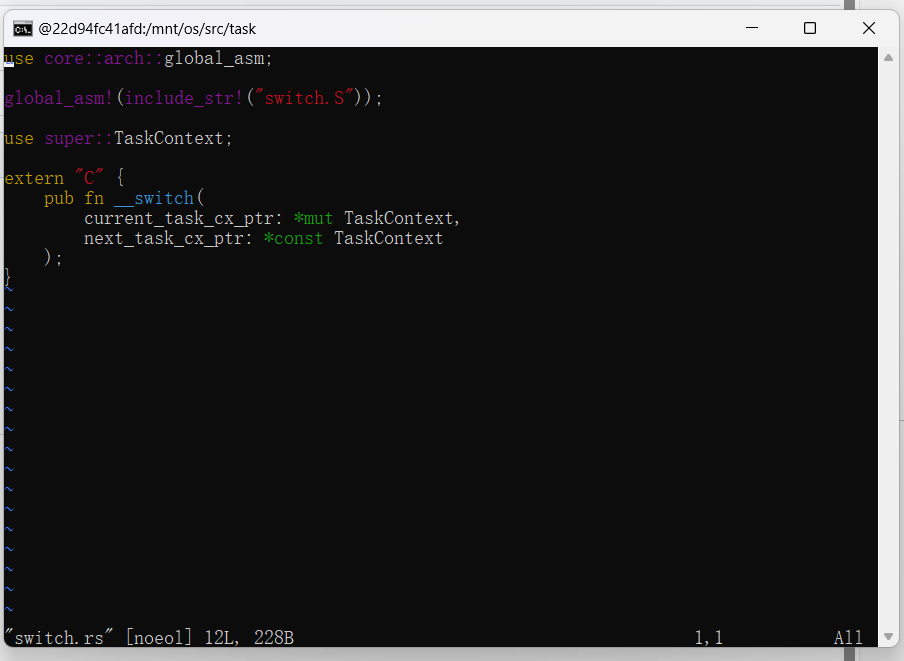


修改/os/src/task/switch.S为：



同时，修改switch.rs。

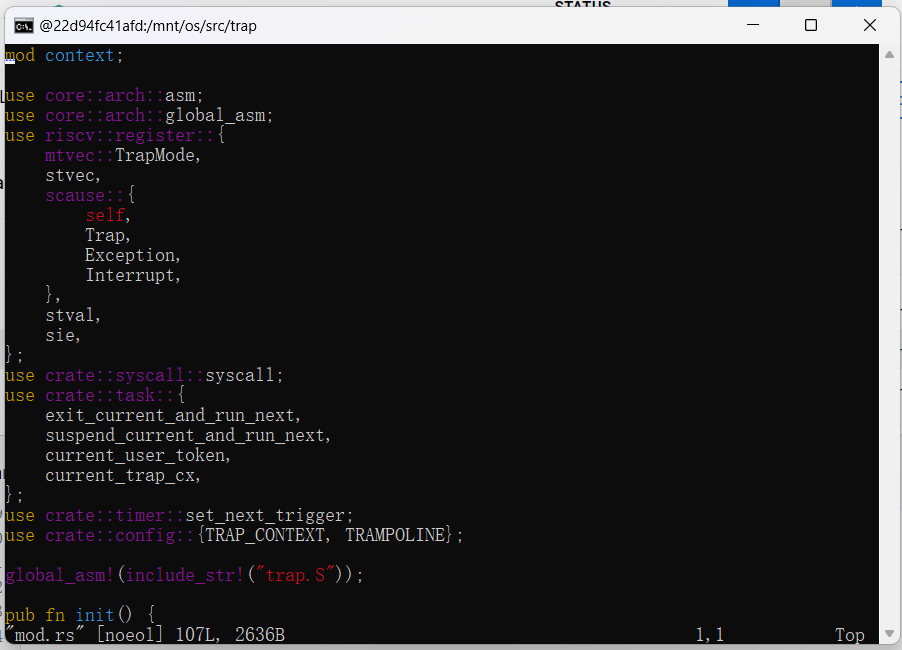
//os/src/task/switch.rs



（4）改进Trap的处理

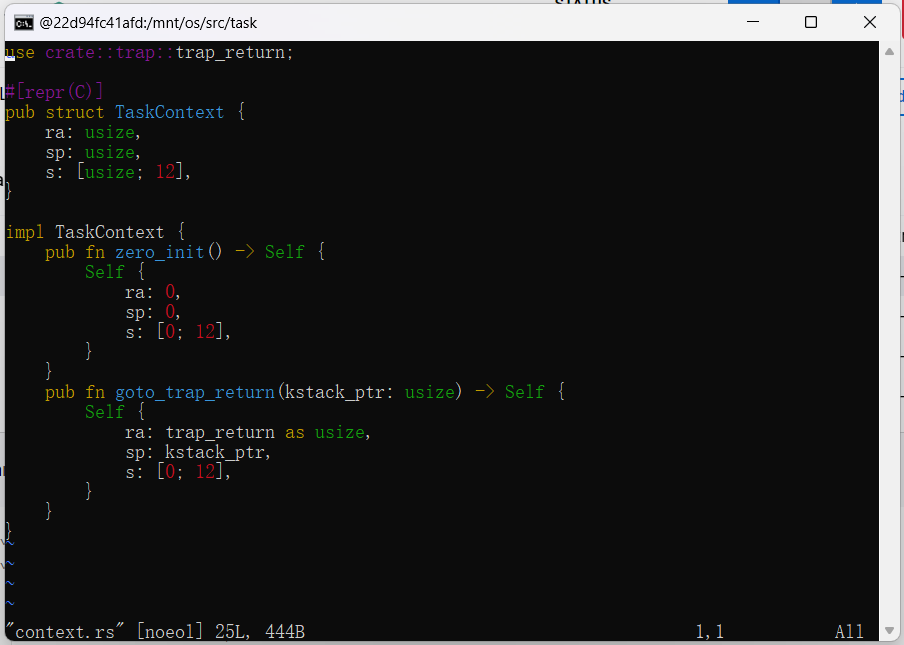
首先修改init函数。然后再trap\_handler的开头增加set\_kernel\_trap\_entry。同时，在处理完trap后还要调用trap\_return 返回用户态。

//os/src/trap/mod.rs



同时，在每一个应用程序第一次获得CPU权限时，内核栈顶放置在内核加载应用的时候构造的一个任务上下文。

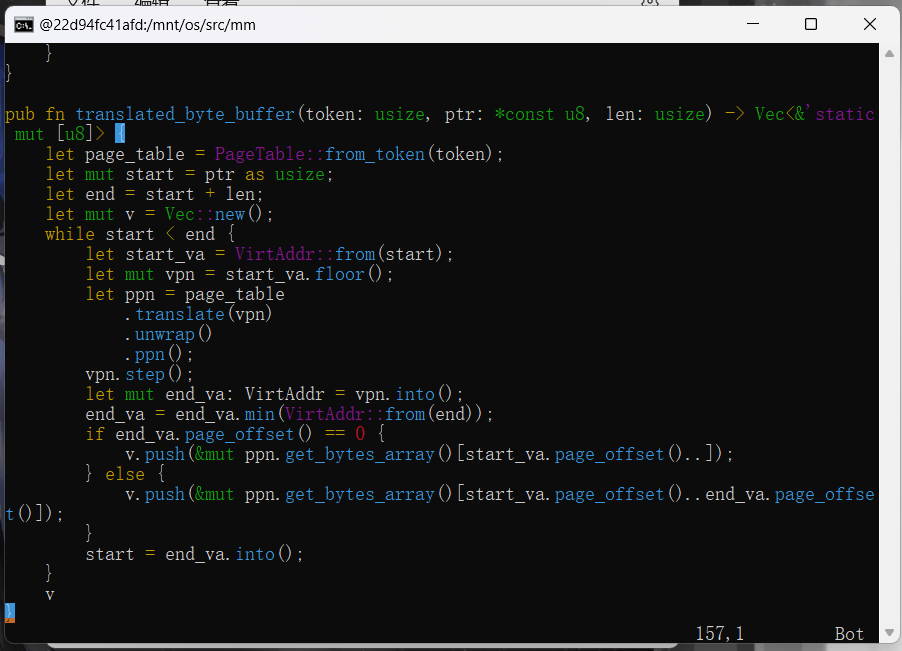
//os/src/task/context/rs



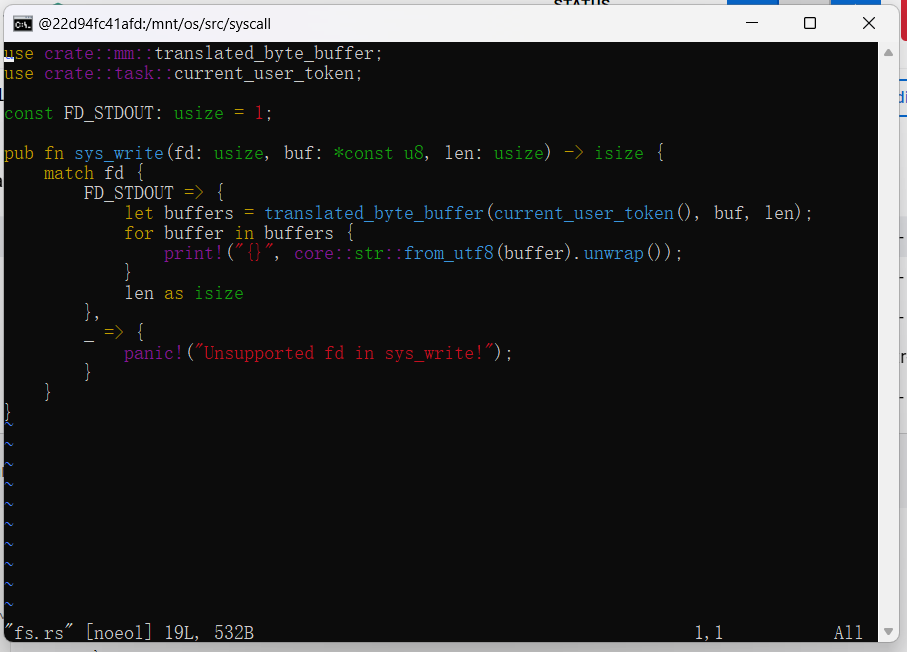
1. 改进sys\_write的实现

由于地址空间的隔离，sys\_write无法直接方法应用空间的数据。为此，页表page\_table提供一个将应用地址空间的缓冲区转化为内核地址空间可以直接访问的辅助函数。

//os/src/mm/page\_table.rs



从而，可以修改sys\_write系统调用。

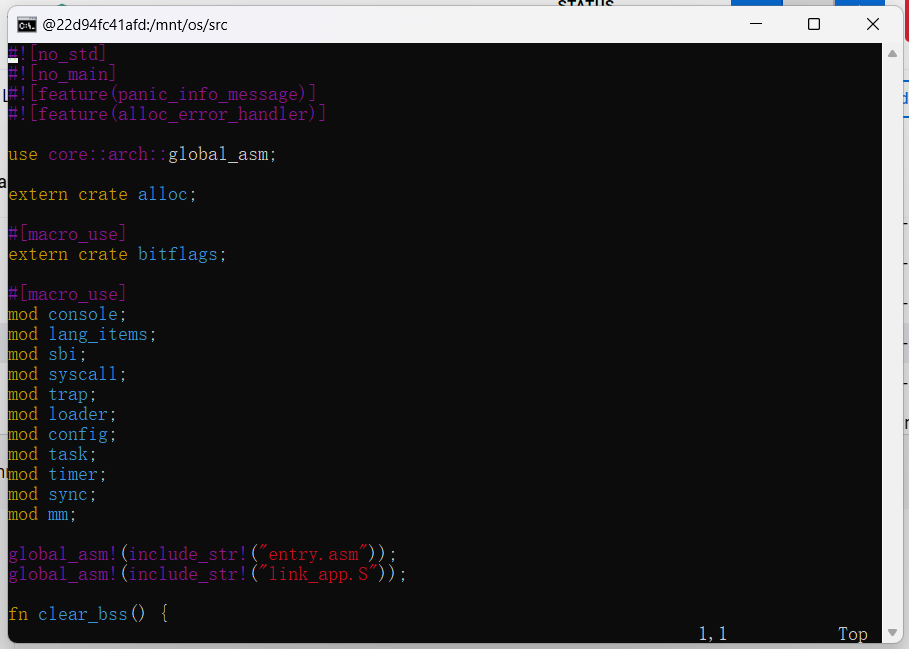


8. 修改应用程序

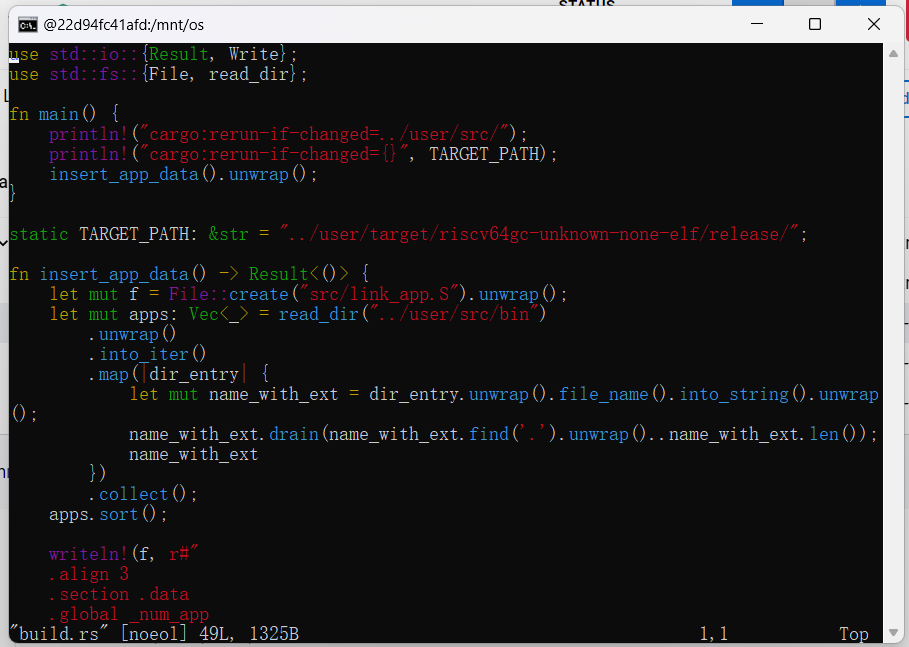
删除user/src/lib.rs中的clear\_bss()，除了删除clear\_bss()的实现外，注意删除\_start()中调用的clear\_bss()。删除build.py。因为应用程序的起始地址是相同的了，所以不在需要build.py，直接删除即可。同时，修改Makefile文件。

9. 修改main.rs

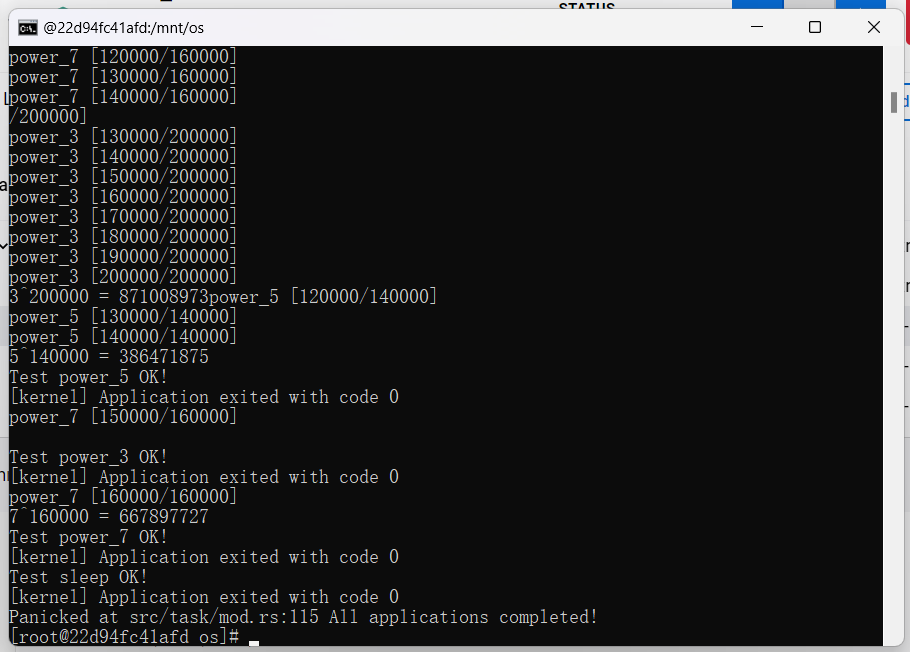
//os/src/main.rs



修改os/build.rs



至此，内存管理部分完成。



1. 思考问题

（1）分析虚拟地址和物理地址的设计与实现；

虚拟地址：使用 buddy\_system\_allocator 库实现了内核中的动态内存分配器。在 os/src/mm/address.rs 中定义了物理地址、虚拟地址、物理页号和虚拟页号等基本数据结构。使用了 bitflags crate 实现了页表项中的标志位，并定义了页表项数据结构 PageTableEntry。定义了页表项的各个标志，控制着虚拟地址与物理地址的映射和访问权限。通过 frame\_allocator.rs 文件实现了物理帧的管理与分配，确保内核可以有效地利用物理内存。这包括设置物理内存的终止地址、物理帧的分配、sync 模块的增加等。实现了页表的基本数据结构，并提供了建立和拆除虚实地址之间映射关系的方法。多级页表管理使得操作系统能够支持大型的虚拟地址空间。用 MemorySet 来描述内核与应用的地址空间，其中逻辑段 MapArea 描述了一段连续地址的虚拟内存，提供了将虚拟地址映射到物理页帧的方式，并通过权限控制实现对逻辑段的访问方式。建立了基于分页模式的虚拟地址空间，实现了跳板机制用于地址空间的切换。

物理地址： 在 os/src/linker.ld 中，通过 ekernel 确定了内核数据的终止地址，该地址之后的物理内存可用。在 os/src/mm/frame\_allocator.rs 中实现了物理帧的管理与分配。通过 FrameAllocator trait 提供了获取物理帧和释放物理帧的方法，确保内核可以有效地利用物理内存。增加了 sync 模块，其中实现了 UPSafeCell。这是为了确保对物理帧的访问是线程安全的。

（2）分析物理帧是如何管理与分配的；

物理帧的管理与分配通过设置物理内存终止地址、实现物理帧管理器，以及引入同步机制，确保了内核对物理内存的有效管理。这为操作系统提供了可靠的物理内存分配机制，支持多任务环境下的内存操作。

（3）分析内核的地址空间以及应用程序的地址空间是如何实现的；

操作系统的内核和应用程序地址空间是通过多级页表管理实现的。内核地址空间由MemorySet抽象表示，包括逻辑段的映射关系、权限控制等，通过new\_kernel初始化。应用程序地址空间在动态加载时，使用同一链接脚本，解析ELF格式数据，通过 MemorySet构建。分时多任务通过建立基于分页模式的虚拟地址空间和跳板机制实现。内核初始化时加载应用程序，TaskManager管理任务，Trap处理异常。通过页表提供的辅助函数，实现了系统调用sys\_write在不同地址空间的数据访问。

（4）分析基于地址空间的分时多任务是如何实现的；

基于地址空间的分时多任务实现依赖多级页表机制，通过内核和应用地址空间的切换，以及跳板机制，实现任务上下文的切换。内核地址空间由 MemorySet 抽象管理，应用地址空间通过 ELF 解析得到。任务管理器负责加载应用程序，同时 trap 处理进行切换。每个应用程序获得 CPU 权限时，内核栈顶置于构造的任务上下文。通过页表的辅助函数，系统调用 sys\_write 跨地址空间进行数据访问。这种机制为多任务执行提供了基础，通过分页和地址空间隔离实现了安全、高效的任务切换。

1. Git提交截图

