# 19271169-张东植-实验报告6

- **Gitlab repo: http://202.205.102.126:88/ZhangDongZhi/os-lab.git**

The main purpose of this experiment is to achieve dynamic memory application and release management. Through this experiment, dynamic memory application and release management can be realized. It includes dynamic memory allocation, basic definition of virtual address and physical address, data structure of page table, management and allocation of physical frame, multilevel page table management, address space of kernel and application, etc.

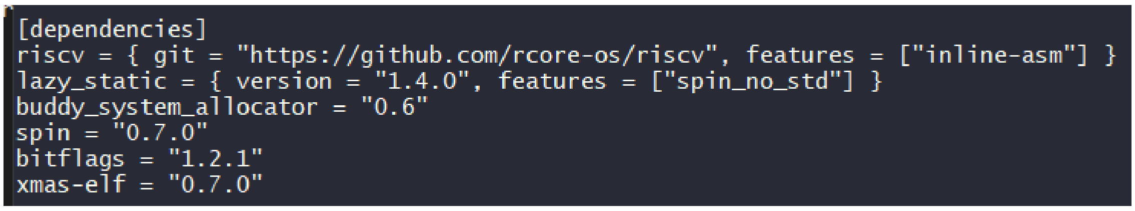
## 1. 实现内核支持动态内存分配

Since there is no standard library support in the operating system kernel, we need to implement the basic dynamic memory allocator using the interface defined by the Alloc library. Since the implementation of the memory allocator is complicated, the existing partner allocator is used directly.

1. First, add dependencies to the **"cargo.toml"** file. At the same time, we also need to introduce the dependency of **alloc** library in "**main.rs**", and add the following code.

```
buddy_system_allocator = "0.6"

// os/src/main.rs
extern crate alloc;
```

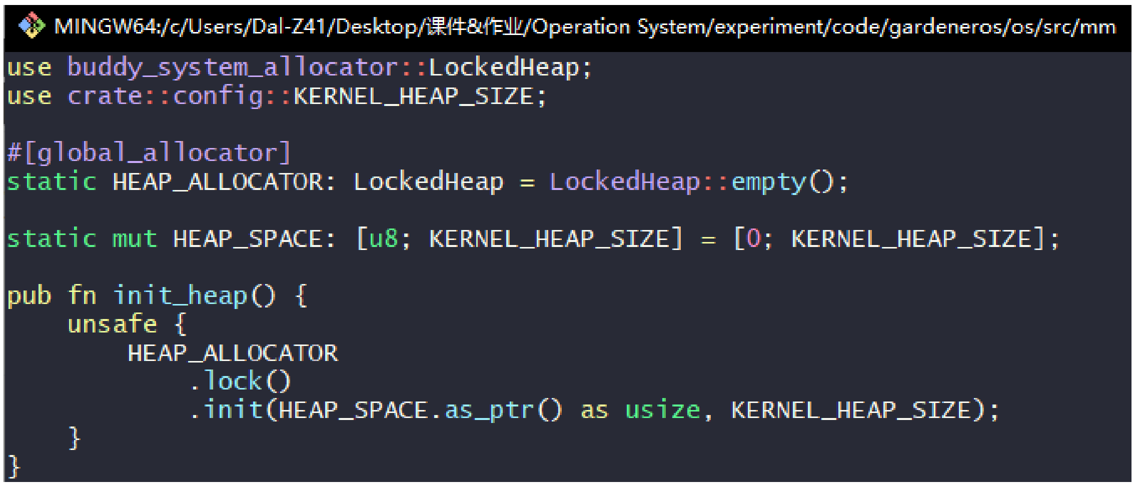
```
extern crate alloc;
```



```
[dependencies]
riscv = { git = "https://github.com/rcore-os/riscv", features = ["inline-asm"] }
lazy_static = { version = "1.4.0", features = ["spin_no_std"] }
buddy_system_allocator = "0.6"
spin = "0.7.0"
bitflags = "1.2.1"
xmas-elf = "0.7.0"
```

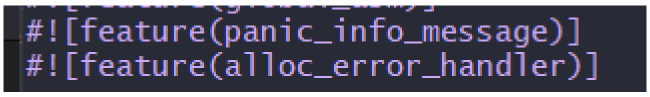
2. Provides a global dynamic memory allocator based on the structure left by alloc. Also, we need to deal with dynamic memory allocation failures.

```
// os/src/mm/heap_allocator.rs

use buddy_system_allocator::LockedHeap;
use crate::config::KERNEL_HEAP_SIZE;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap = LockedHeap::empty();

static mut HEAP_SPACE: [u8; KERNEL_HEAP_SIZE] = [0; KERNEL_HEAP_SIZE];

pub fn init_heap() {
    unsafe {
        HEAP_ALLOCATOR
            .lock()
            .init(HEAP_SPACE.as_ptr() as usize, KERNEL_HEAP_SIZE);
    }
}

// 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);
}
```

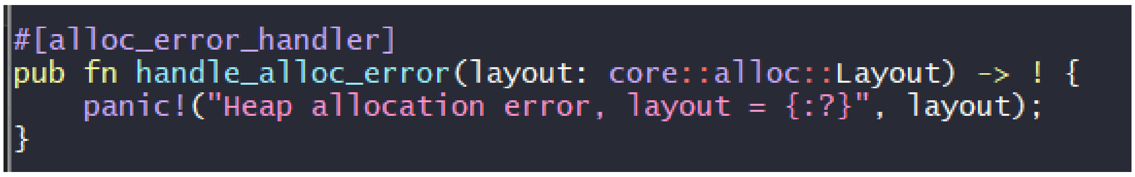


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/mm

```rust
use buddy_system_allocator::LockedHeap;
use crate::config::KERNEL_HEAP_SIZE;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap = LockedHeap::empty();

static mut HEAP_SPACE: [u8; KERNEL_HEAP_SIZE] = [0; KERNEL_HEAP_SIZE];

pub fn init_heap() {
    unsafe {
        HEAP_ALLOCATOR
            .lock()
            .init(HEAP_SPACE.as_ptr() as usize, KERNEL_HEAP_SIZE);
    }
}
```



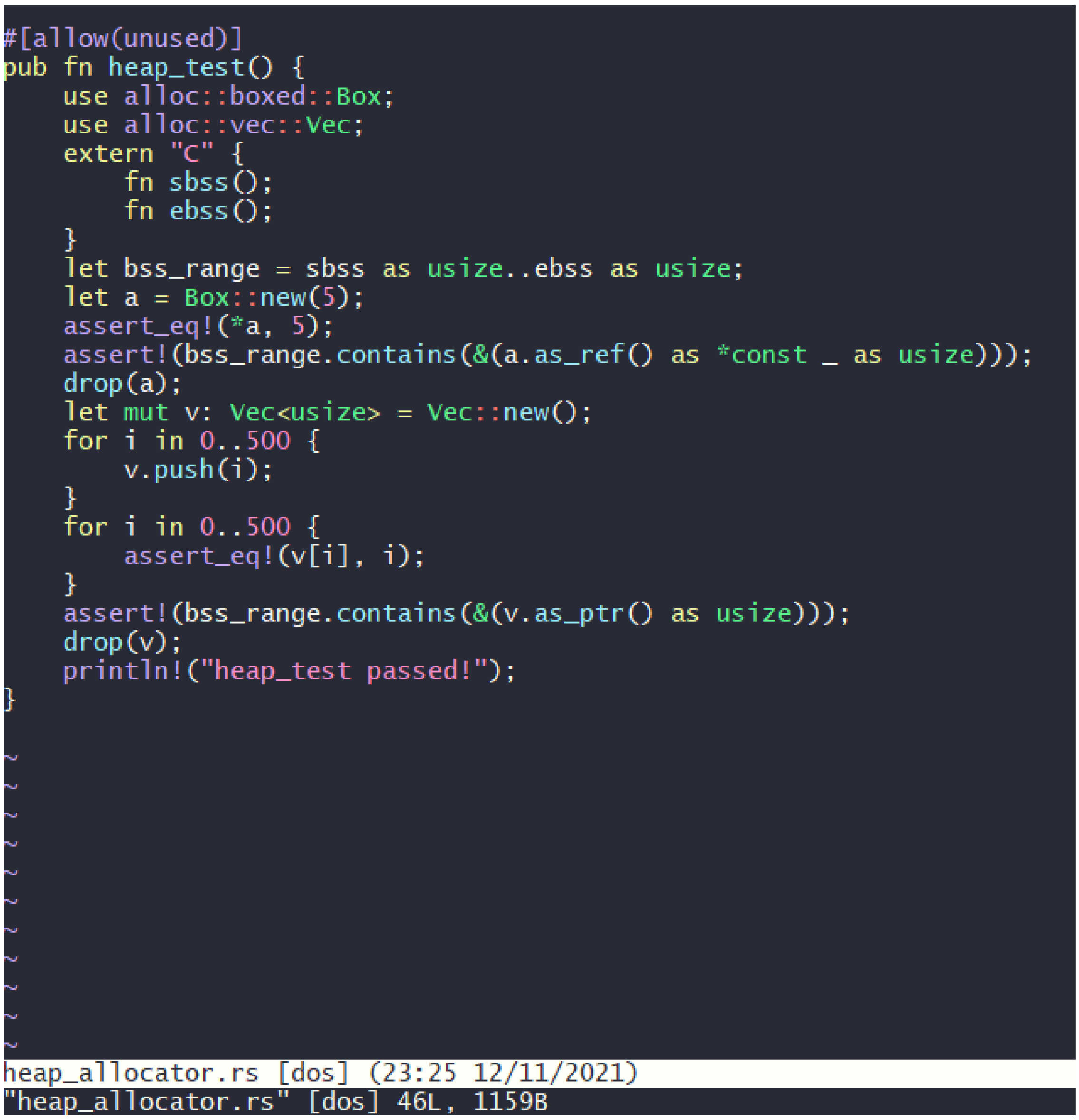
```rust
#![feature(panic_info_message)]
#![feature(alloc_error_handler)]
```



```rust
#[alloc_error_handler]
pub fn handle_alloc_error(layout: core::alloc::Layout) -> ! {
    panic!("Heap allocation error, layout = {:?}", layout);
}
```

3. Then, the implementation tests dynamic memory allocation.

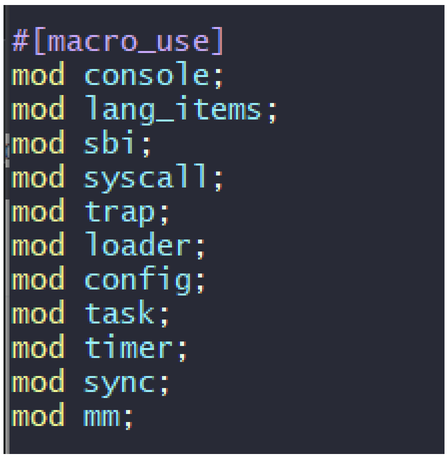
```rust
// os/src/mm/heap_allocator.rs

#[allow(unused)]
pub fn heap_test() {
    use alloc::boxed::Box;
    use alloc::vec::Vec;
    extern "C" {
        fn sbss();
        fn ebss();
    }
    let bss_range = sbss as usize..ebss as usize;
    let a = Box::new(5);
    assert_eq!(*a, 5);
    assert!(bss_range.contains(&(a.as_ref() as *const _ as usize)));
    drop(a);
    let mut v: Vec<usize> = Vec::new();
    for i in 0..500 {
        v.push(i);
    }
    for i in 0..500 {
        assert_eq!(v[i], i);
    }
    assert!(bss_range.contains(&(v.as_ptr() as usize)));
    drop(v);
    println!("heap_test passed!");
}
```



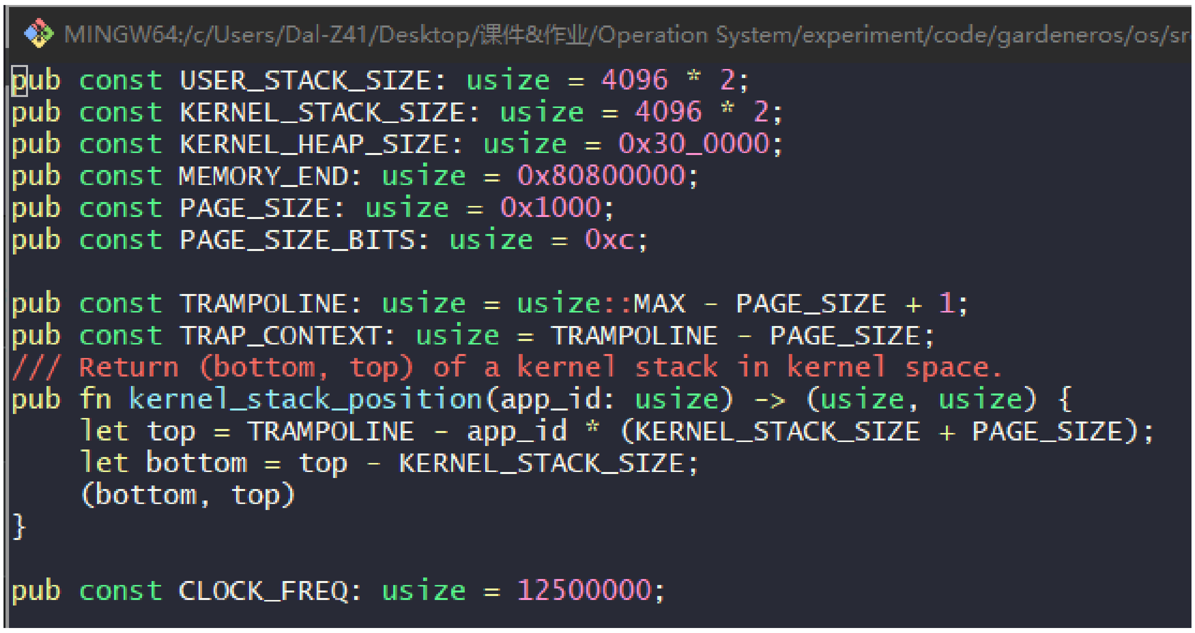
```
#[allow(unused)]
pub fn heap_test() {
    use alloc::boxed::Box;
    use alloc::vec::Vec;
    extern "C" {
        fn sbss();
        fn ebss();
    }
    let bss_range = sbss as usize..ebss as usize;
    let a = Box::new(5);
    assert_eq!(*a, 5);
    assert!(bss_range.contains(&(a.as_ref() as *const _ as usize)));
    drop(a);
    let mut v: Vec<usize> = Vec::new();
    for i in 0..500 {
        v.push(i);
    }
    for i in 0..500 {
        assert_eq!(v[i], i);
    }
    assert!(bss_range.contains(&(v.as_ptr() as usize)));
    drop(v);
    println!("heap_test passed!");
}

~
~
~
~
~
~
~
~
~
~
~
heap_allocator.rs [dos] (23:25 12/11/2021)
"heap_allocator.rs" [dos] 46L, 1159B
```

4. Finally, modify other parts of the code.

```
// 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

mod heap_allocator;

pub fn init() {
    heap_allocator::init_heap();
    heap_allocator::heap_test();
}
```

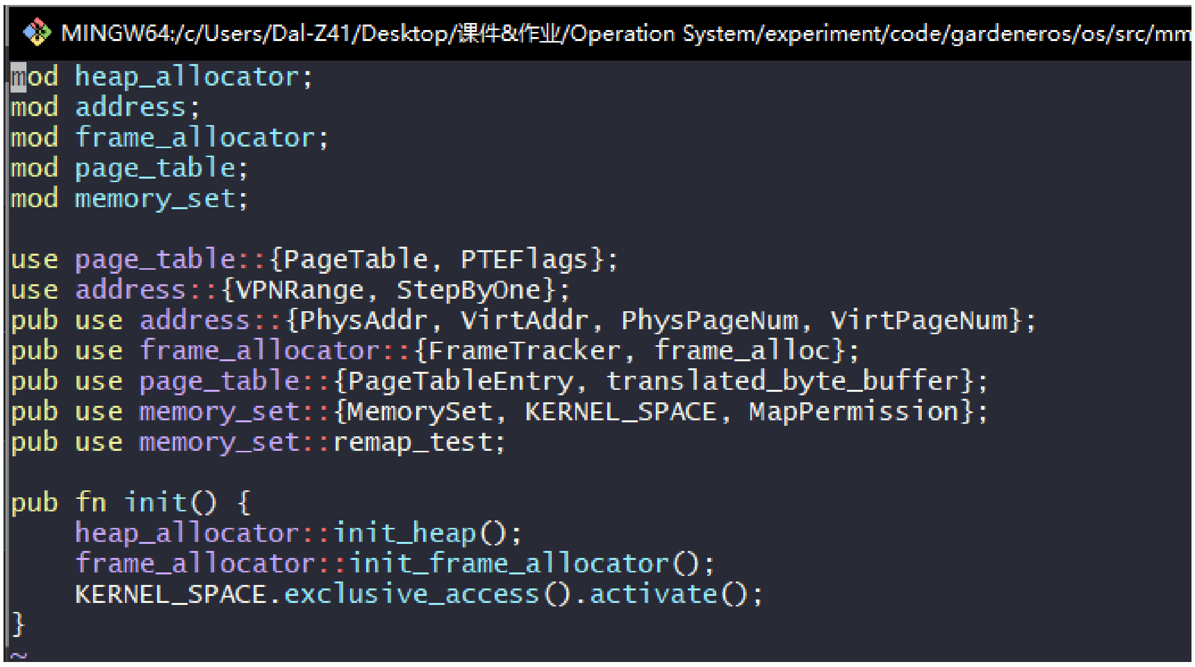


```
#[macro_use]
mod console;
mod lang_items;
mod sbi;
mod syscall;
mod trap;
mod loader;
mod config;
mod task;
mod timer;
mod sync;
mod mm;
```



MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/sr

```
pub const USER_STACK_SIZE: usize = 4096 * 2;
pub const KERNEL_STACK_SIZE: usize = 4096 * 2;
pub const KERNEL_HEAP_SIZE: usize = 0x30_0000;
pub const MEMORY_END: usize = 0x80800000;
pub const PAGE_SIZE: usize = 0x1000;
pub const PAGE_SIZE_BITS: usize = 0xc;

pub const TRAMPOLINE: usize = usize::MAX - PAGE_SIZE + 1;
pub const TRAP_CONTEXT: usize = TRAMPOLINE - PAGE_SIZE;
/// Return (bottom, top) of a kernel stack in kernel space.
pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
    let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
    let bottom = top - KERNEL_STACK_SIZE;
    (bottom, top)
}

pub const CLOCK_FREQ: usize = 12500000;
```



MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/mm

```
mod heap_allocator;
mod address;
mod frame_allocator;
mod page_table;
mod memory_set;

use page_table::{PageTable, PTEFlags};
use address::{VPNRange, StepByOne};
pub use address::{PhysAddr, VirtAddr, PhysPageNum, VirtPageNum};
pub use frame_allocator::{FrameTracker, frame_alloc};
pub use page_table::{PageTableEntry, translated_byte_buffer};
pub use memory_set::{MemorySet, KERNEL_SPACE, MapPermission};
pub use memory_set::remap_test;

pub fn init() {
    heap_allocator::init_heap();
    frame_allocator::init_frame_allocator();
    KERNEL_SPACE.exclusive_access().activate();
}
~
```

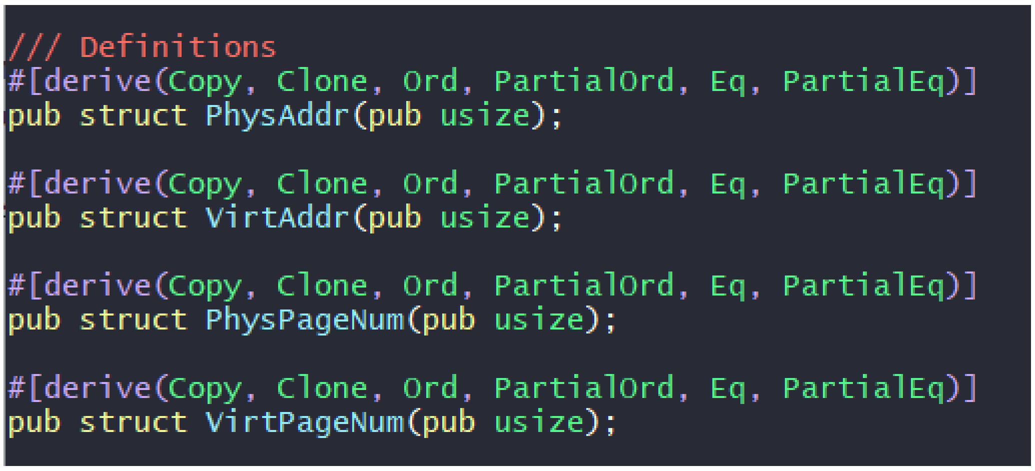
## 2. 实现虚拟地址与物理地址

This section implements the virtual address and physical address data structure.

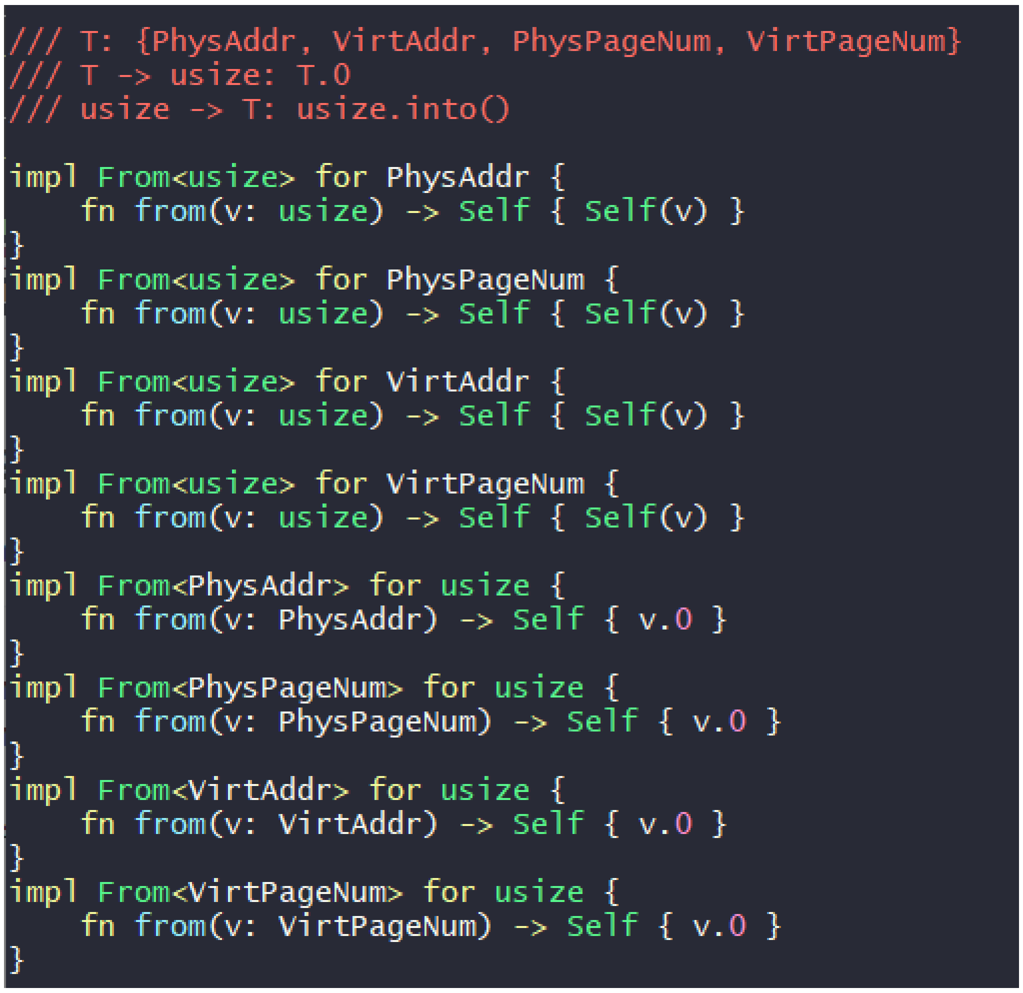
### 2.1 Data Structure Definition

First, define the basic data structures required, including physical address, virtual address, physical page number, and virtual page number. Besides, implement conversions between these types and usize.

```rust
#[repr(C)]
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct PhysAddr(pub usize);

#[repr(C)]
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtAddr(pub usize);

#[repr(C)]
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct PhysPageNum(pub usize);

#[repr(C)]
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtPageNum(pub usize);

impl From<usize> for PhysAddr {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<usize> for PhysPageNum {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<usize> for VirtAddr {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<usize> for VirtPageNum {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<PhysAddr> for usize {
    fn from(v: PhysAddr) -> Self { v.0 }
}
impl From<PhysPageNum> for usize {
    fn from(v: PhysPageNum) -> Self { v.0 }
}
impl From<VirtAddr> for usize {
    fn from(v: VirtAddr) -> Self { v.0 }
}
impl From<VirtPageNum> for usize {
    fn from(v: VirtPageNum) -> Self { v.0 }
}
```



```rust
/// Definitions
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct PhysAddr(pub usize);

#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtAddr(pub usize);

#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct PhysPageNum(pub usize);

#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtPageNum(pub usize);
```

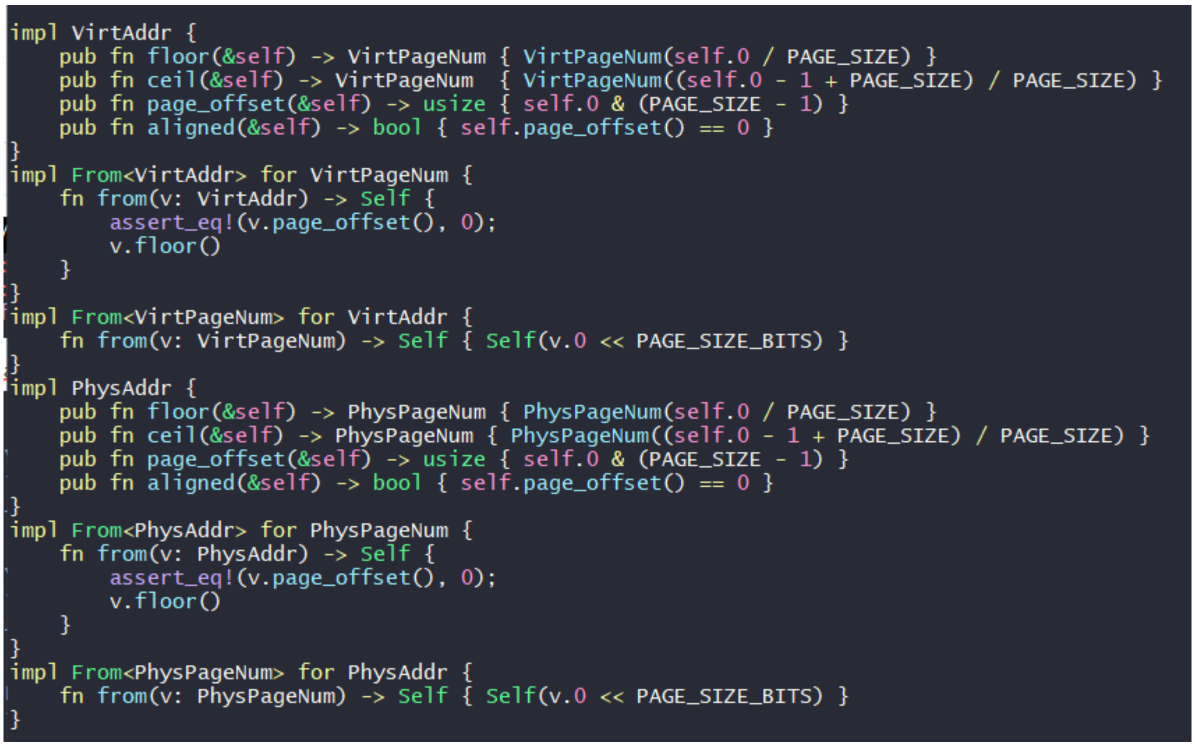


```rust
/// T: {PhysAddr, VirtAddr, PhysPageNum, VirtPageNum}
/// T -> usize: T.0
/// usize -> T: usize.into()

impl From<usize> for PhysAddr {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<usize> for PhysPageNum {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<usize> for VirtAddr {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<usize> for VirtPageNum {
    fn from(v: usize) -> Self { Self(v) }
}
impl From<PhysAddr> for usize {
    fn from(v: PhysAddr) -> Self { v.0 }
}
impl From<PhysPageNum> for usize {
    fn from(v: PhysPageNum) -> Self { v.0 }
}
impl From<VirtAddr> for usize {
    fn from(v: VirtAddr) -> Self { v.0 }
}
impl From<VirtPageNum> for usize {
    fn from(v: VirtPageNum) -> Self { v.0 }
}
```

## 2.2 Convert between Page and Address

Then, implement the convert between physical address and page number.

```rust
impl VirtAddr {
    pub fn floor(&self) -> VirtPageNum { VirtPageNum(self.0 / PAGE_SIZE) }
    pub fn ceil(&self) -> VirtPageNum  { VirtPageNum((self.0 - 1 + PAGE_SIZE) /
PAGE_SIZE) }
    pub fn page_offset(&self) -> usize { self.0 & (PAGE_SIZE - 1) }
    pub fn aligned(&self) -> bool { self.page_offset() == 0 }
}

impl From<VirtAddr> for VirtPageNum {
    fn from(v: VirtAddr) -> Self {
        assert_eq!(v.page_offset(), 0);
        v.floor()
    }
}

impl From<VirtPageNum> for VirtAddr {
    fn from(v: VirtPageNum) -> Self { Self(v.0 << PAGE_SIZE_BITS) }
}

impl PhysAddr {
    pub fn floor(&self) -> PhysPageNum { PhysPageNum(self.0 / PAGE_SIZE) }
    pub fn ceil(&self) -> PhysPageNum { PhysPageNum((self.0 - 1 + PAGE_SIZE) /
PAGE_SIZE) }
    pub fn page_offset(&self) -> usize { self.0 & (PAGE_SIZE - 1) }
    pub fn aligned(&self) -> bool { self.page_offset() == 0 }
}

impl From<PhysAddr> for PhysPageNum {
    fn from(v: PhysAddr) -> Self {
        assert_eq!(v.page_offset(), 0);
        v.floor()
    }
}

impl From<PhysPageNum> for PhysAddr {
    fn from(v: PhysPageNum) -> Self { Self(v.0 << PAGE_SIZE_BITS) }
}
```



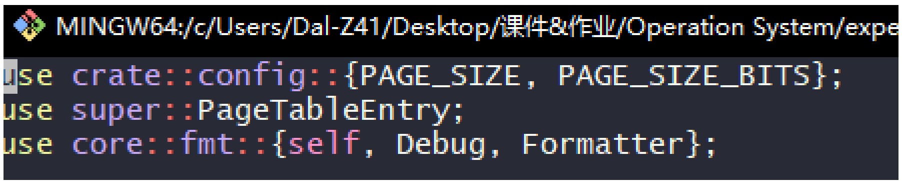
```
impl VirtAddr {
    pub fn floor(&self) -> VirtPageNum { VirtPageNum(self.0 / PAGE_SIZE) }
    pub fn ceil(&self) -> VirtPageNum  { VirtPageNum((self.0 - 1 + PAGE_SIZE) / PAGE_SIZE) }
    pub fn page_offset(&self) -> usize { self.0 & (PAGE_SIZE - 1) }
    pub fn aligned(&self) -> bool { self.page_offset() == 0 }
}
impl From<VirtAddr> for VirtPageNum {
    fn from(v: VirtAddr) -> Self {
        assert_eq!(v.page_offset(), 0);
        v.floor()
    }
}
impl From<VirtPageNum> for VirtAddr {
    fn from(v: VirtPageNum) -> Self { Self(v.0 << PAGE_SIZE_BITS) }
}
impl PhysAddr {
    pub fn floor(&self) -> PhysPageNum { PhysPageNum(self.0 / PAGE_SIZE) }
    pub fn ceil(&self) -> PhysPageNum { PhysPageNum((self.0 - 1 + PAGE_SIZE) / PAGE_SIZE) }
    pub fn page_offset(&self) -> usize { self.0 & (PAGE_SIZE - 1) }
    pub fn aligned(&self) -> bool { self.page_offset() == 0 }
}
impl From<PhysAddr> for PhysPageNum {
    fn from(v: PhysAddr) -> Self {
        assert_eq!(v.page_offset(), 0);
        v.floor()
    }
}
impl From<PhysPageNum> for PhysAddr {
    fn from(v: PhysPageNum) -> Self { Self(v.0 << PAGE_SIZE_BITS) }
}
```

## 2.3 Indexed Search

Finally, achieve query index and other content.

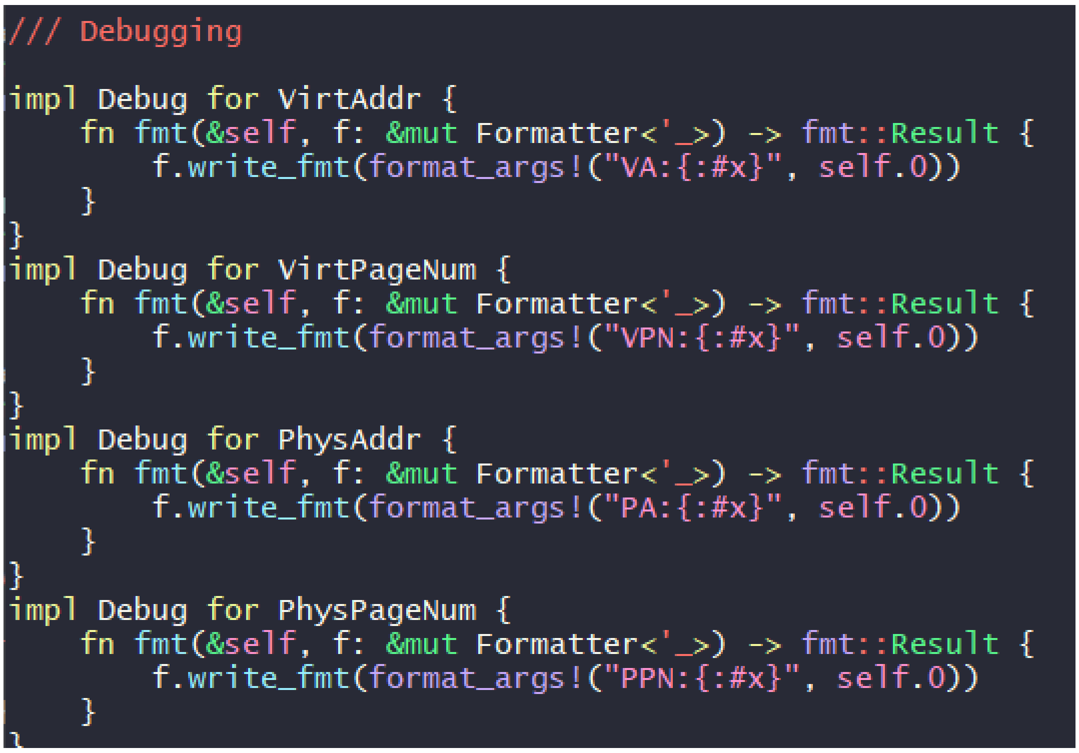
```
use crate::config::{PAGE_SIZE, PAGE_SIZE_BITS};
use super::PageTableEntry;
use core::fmt::{self, Debug, Formatter};

/// Debugging
```

```rust
impl Debug for VirtAddr {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("VA:{:#x}", self.0))
    }
}
impl Debug for VirtPageNum {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("VPN:{:#x}", self.0))
    }
}
impl Debug for PhysAddr {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("PA:{:#x}", self.0))
    }
}
impl Debug for PhysPageNum {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("PPN:{:#x}", self.0))
    }
}

impl VirtPageNum {
    pub fn indexes(&self) -> [usize; 3] {
        let mut vpn = self.0;
        let mut idx = [0usize; 3];
        for i in (0..3).rev() {
            idx[i] = vpn & 511;
            vpn >>= 9;
        }
        idx
    }
}

impl PhysPageNum {
    pub fn get_pte_array(&self) -> &'static mut [PageTableEntry] {
        let pa: PhysAddr = self.clone().into();
        unsafe {
            core::slice::from_raw_parts_mut(pa.0 as *mut PageTableEntry, 512)
        }
    }
    pub fn get_bytes_array(&self) -> &'static mut [u8] {
        let pa: PhysAddr = self.clone().into();
        unsafe {
            core::slice::from_raw_parts_mut(pa.0 as *mut u8, 4096)
        }
    }
    pub fn get_mut<T>(&self) -> &'static mut T {
        let pa: PhysAddr = self.clone().into();
        unsafe {
            (pa.0 as *mut T).as_mut().unwrap()
        }
    }
}

pub trait StepByOne {
    fn step(&mut self);
}
impl StepByOne for VirtPageNum {
    fn step(&mut self) {
        self.0 += 1;
    }
}

#[derive(Copy, Clone)]
pub struct SimpleRange<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    l: T,
    r: T,
}
impl<T> SimpleRange<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    pub fn new(start: T, end: T) -> Self {
        assert!(start <= end, "start {:?} > end {:?}!", start, end);
        Self { l: start, r: end }
    }
    pub fn get_start(&self) -> T { self.l }
    pub fn get_end(&self) -> T { self.r }
}
impl<T> IntoIterator for SimpleRange<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    type Item = T;
    type IntoIter = SimpleRangeIterator<T>;
    fn into_iter(self) -> Self::IntoIter {
        SimpleRangeIterator::new(self.l, self.r)
    }
}
pub struct SimpleRangeIterator<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    current: T,
    end: T,
}
impl<T> SimpleRangeIterator<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    pub fn new(l: T, r: T) -> Self {
        Self { current: l, end: r, }
    }
}
impl<T> Iterator for SimpleRangeIterator<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        if self.current == self.end {
            None
        } else {
            let t = self.current;
            self.current.step();
            Some(t)
        }
    }
}
pub type VPNRange = SimpleRange<VirtPageNum>;
```



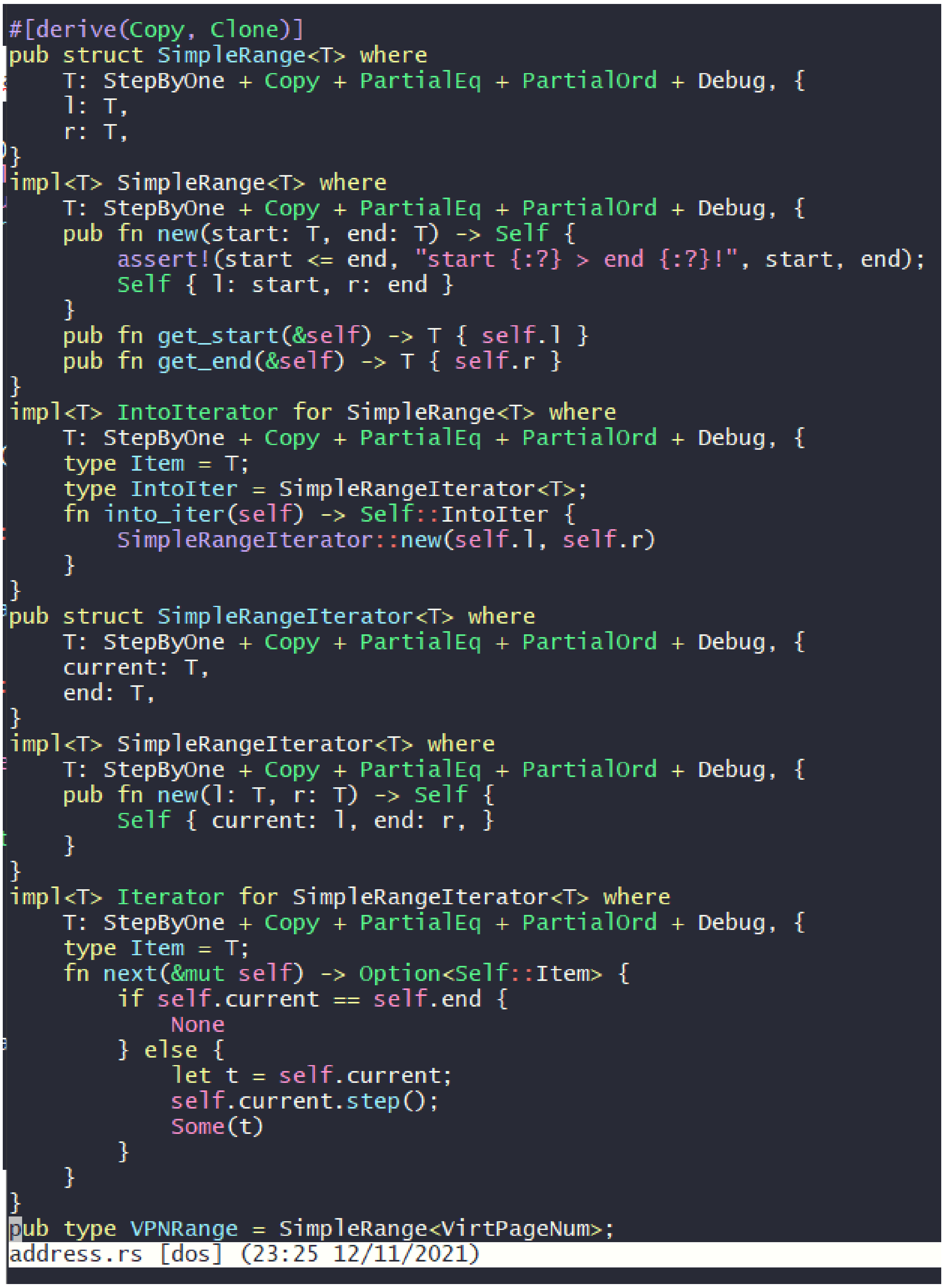
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/expe

```
use crate::config::{PAGE_SIZE, PAGE_SIZE_BITS};
use super::PageTableEntry;
use core::fmt::{self, Debug, Formatter};
```



```
/// Debugging

impl Debug for VirtAddr {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("VA:{:#x}", self.0))
    }
}
impl Debug for VirtPageNum {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("VPN:{:#x}", self.0))
    }
}
impl Debug for PhysAddr {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("PA:{:#x}", self.0))
    }
}
impl Debug for PhysPageNum {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("PPN:{:#x}", self.0))
    }
}
```

```rust
impl VirtPageNum {
    pub fn indexes(&self) -> [usize; 3] {
        let mut vpn = self.0;
        let mut idx = [0usize; 3];
        for i in (0..3).rev() {
            idx[i] = vpn & 511;
            vpn >>= 9;
        }
        idx
    }
}

impl PhysPageNum {
    pub fn get_pte_array(&self) -> &'static mut [PageTableEntry] {
        let pa: PhysAddr = self.clone().into();
        unsafe {
            core::slice::from_raw_parts_mut(pa.0 as *mut PageTableEntry, 512)
        }
    }
    pub fn get_bytes_array(&self) -> &'static mut [u8] {
        let pa: PhysAddr = self.clone().into();
        unsafe {
            core::slice::from_raw_parts_mut(pa.0 as *mut u8, 4096)
        }
    }
    pub fn get_mut<T>(&self) -> &'static mut T {
        let pa: PhysAddr = self.clone().into();
        unsafe {
            (pa.0 as *mut T).as_mut().unwrap()
        }
    }
}

pub trait StepByOne {
    fn step(&mut self);
}
impl StepByOne for VirtPageNum {
    fn step(&mut self) {
        self.0 += 1;
    }
}
```



```
#[derive(Copy, Clone)]
pub struct SimpleRange<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    l: T,
    r: T,
}
impl<T> SimpleRange<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    pub fn new(start: T, end: T) -> Self {
        assert!(start <= end, "start {:?} > end {:?}!", start, end);
        Self { l: start, r: end }
    }
    pub fn get_start(&self) -> T { self.l }
    pub fn get_end(&self) -> T { self.r }
}
impl<T> IntoIterator for SimpleRange<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    type Item = T;
    type IntoIter = SimpleRangeIterator<T>;
    fn into_iter(self) -> Self::IntoIter {
        SimpleRangeIterator::new(self.l, self.r)
    }
}
pub struct SimpleRangeIterator<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    current: T,
    end: T,
}
impl<T> SimpleRangeIterator<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    pub fn new(l: T, r: T) -> Self {
        Self { current: l, end: r, }
    }
}
impl<T> Iterator for SimpleRangeIterator<T> where
    T: StepByOne + Copy + PartialEq + PartialOrd + Debug, {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        if self.current == self.end {
            None
        } else {
            let t = self.current;
            self.current.step();
            Some(t)
        }
    }
}
pub type VPNRange = SimpleRange<VirtPageNum>;
address.rs [dos] (23:25 12/11/2021)
```

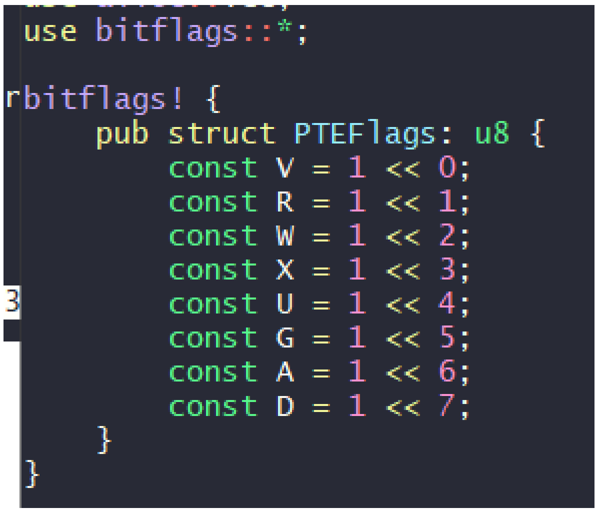
## 3. 页表项数据结构定义

To implement the page table entry definition, we need to use **crate bitflags** of the bits commonly used in Rust.

### 3.1 Flag bit

First, implement the flag bit **PTEFlags** in the page table entry.

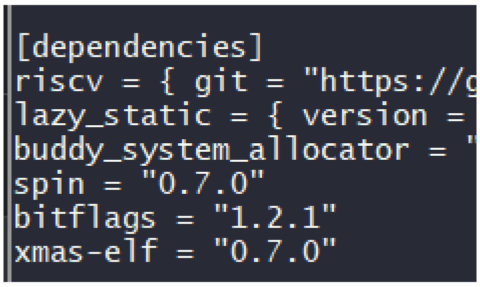
```
// os/src/main.rs
#[macro_use]
extern crate bitflags;

// 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;
    }
}
```



```
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;
    }
}
```

## 3.2 Dependencies

Meanwhile, we need to add **bitflgs** dependencies to the config file.



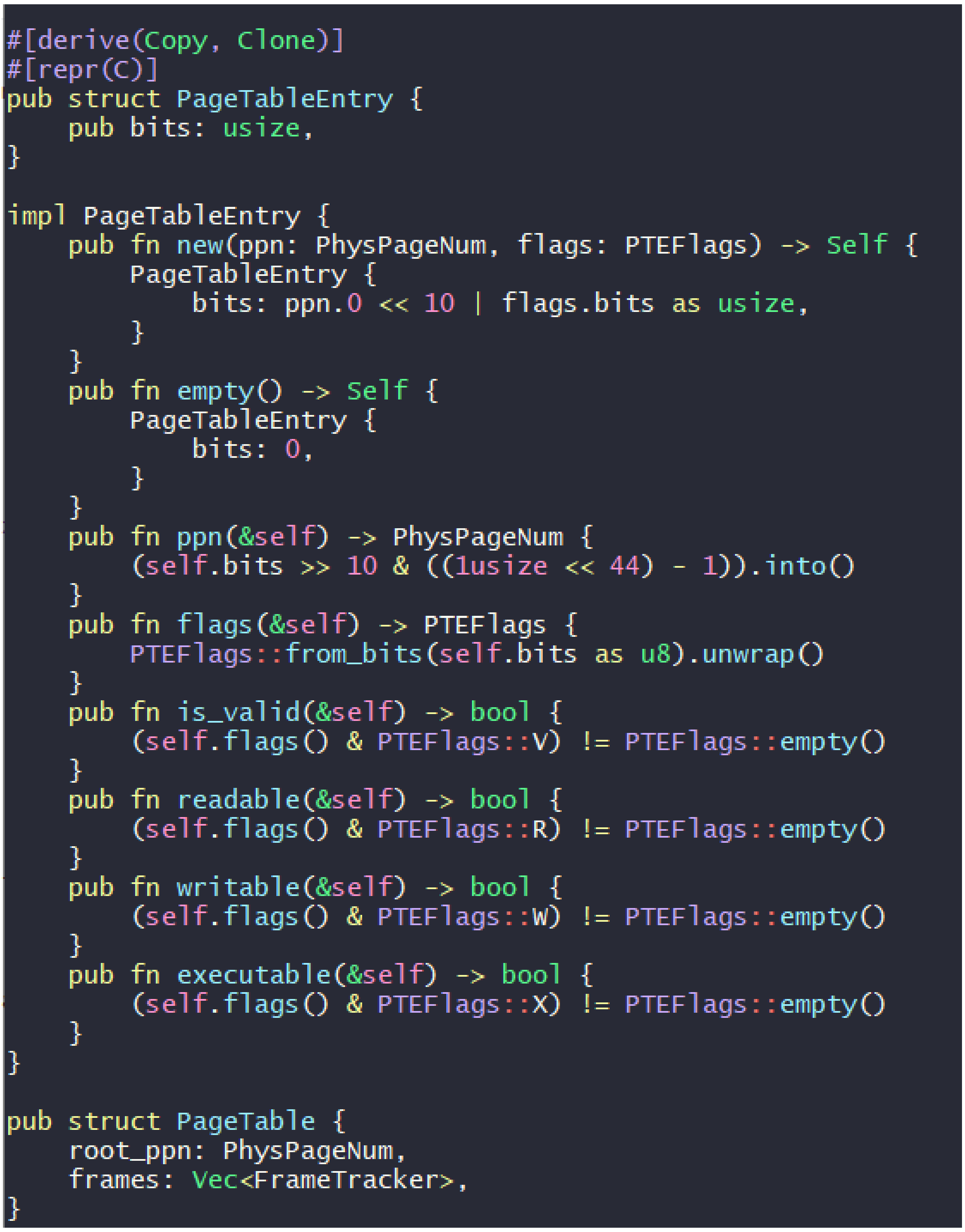
```
[dependencies]
riscv = { git = "https://g
lazy_static = { version =
buddy_system_allocator = "
spin = "0.7.0"
bitflags = "1.2.1"
xmas-elf = "0.7.0"
```

## 3.3 Table Entry

Implement **Page Table Entry** in file **"page_table.rs"**, and add usages in file **"mod.rs"**.

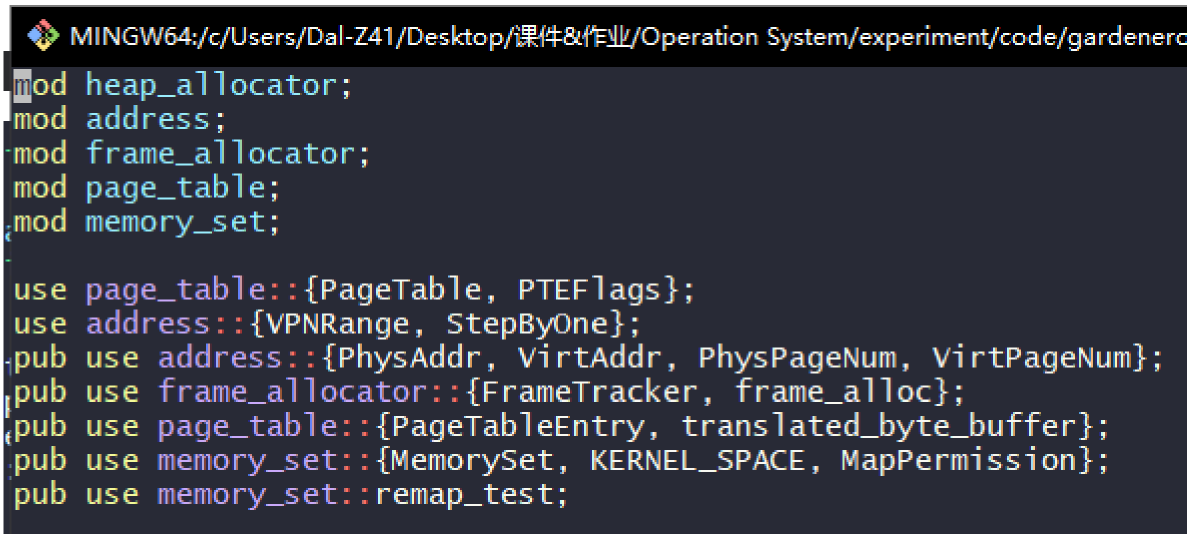
```
//os/src/mm/page_table.rs
use super::{frame_alloc, PhysPageNum, FrameTracker, VirtPageNum, VirtAddr,
StepByOne};

#[derive(Copy, Clone)]
#[repr(C)]
```

```rust
pub struct PageTableEntry {
    pub bits: usize,
}

impl PageTableEntry {
    pub fn new(ppn: PhysPageNum, flags: PTEFlags) -> Self {
        PageTableEntry {
            bits: ppn.0 << 10 | flags.bits as usize,
        }
    }
    pub fn empty() -> Self {
        PageTableEntry {
            bits: 0,
        }
    }
    pub fn ppn(&self) -> PhysPageNum {
        (self.bits >> 10 & ((1usize << 44) - 1)).into()
    }
    pub fn flags(&self) -> PTEFlags {
        PTEFlags::from_bits(self.bits as u8).unwrap()
    }
    pub fn is_valid(&self) -> bool {
        (self.flags() & PTEFlags::V) != PTEFlags::empty()
    }
    pub fn readable(&self) -> bool {
        (self.flags() & PTEFlags::R) != PTEFlags::empty()
    }
    pub fn writable(&self) -> bool {
        (self.flags() & PTEFlags::W) != PTEFlags::empty()
    }
    pub fn executable(&self) -> bool {
        (self.flags() & PTEFlags::X) != PTEFlags::empty()
    }
}
```



```rust
#[derive(Copy, Clone)]
#[repr(C)]
pub struct PageTableEntry {
    pub bits: usize,
}

impl PageTableEntry {
    pub fn new(ppn: PhysPageNum, flags: PTEFlags) -> Self {
        PageTableEntry {
            bits: ppn.0 << 10 | flags.bits as usize,
        }
    }
    pub fn empty() -> Self {
        PageTableEntry {
            bits: 0,
        }
    }
    pub fn ppn(&self) -> PhysPageNum {
        (self.bits >> 10 & ((1usize << 44) - 1)).into()
    }
    pub fn flags(&self) -> PTEFlags {
        PTEFlags::from_bits(self.bits as u8).unwrap()
    }
    pub fn is_valid(&self) -> bool {
        (self.flags() & PTEFlags::V) != PTEFlags::empty()
    }
    pub fn readable(&self) -> bool {
        (self.flags() & PTEFlags::R) != PTEFlags::empty()
    }
    pub fn writable(&self) -> bool {
        (self.flags() & PTEFlags::W) != PTEFlags::empty()
    }
    pub fn executable(&self) -> bool {
        (self.flags() & PTEFlags::X) != PTEFlags::empty()
    }
}

pub struct PageTable {
    root_ppn: PhysPageNum,
    frames: Vec<FrameTracker>,
}
```

```rust
// mod.rs
mod page_table;

use page_table::{PTEFlags};
pub use page_table::{PageTableEntry};
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardenerc

```
mod heap_allocator;
mod address;
mod frame_allocator;
mod page_table;
mod memory_set;

use page_table::{PageTable, PTEFlags};
use address::{VPNRange, StepByOne};
pub use address::{PhysAddr, VirtAddr, PhysPageNum, VirtPageNum};
pub use frame_allocator::{FrameTracker, frame_alloc};
pub use page_table::{PageTableEntry, translated_byte_buffer};
pub use memory_set::{MemorySet, KERNEL_SPACE, MapPermission};
pub use memory_set::remap_test;
```

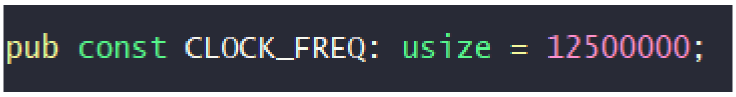
## 4. 物理帧的管理与分配

This section includes the implementations of managing and distributing physical frames.

### 4.1 End-address Settings

In file **"linker.ld"**, **ekernel** determines the end address of kernel data, after which all physical memory is available. We limit the size of physical memory by setting parameters in the **"config"** submodule.

```
// os/src/config.rs
pub const MEMORY_END: usize = 0x80800000;
```



```
pub const CLOCK_FREQ: usize = 12500000;
```

### 4.2 Frame Management

Implement this part in file **"frame_allocator.rs"**.

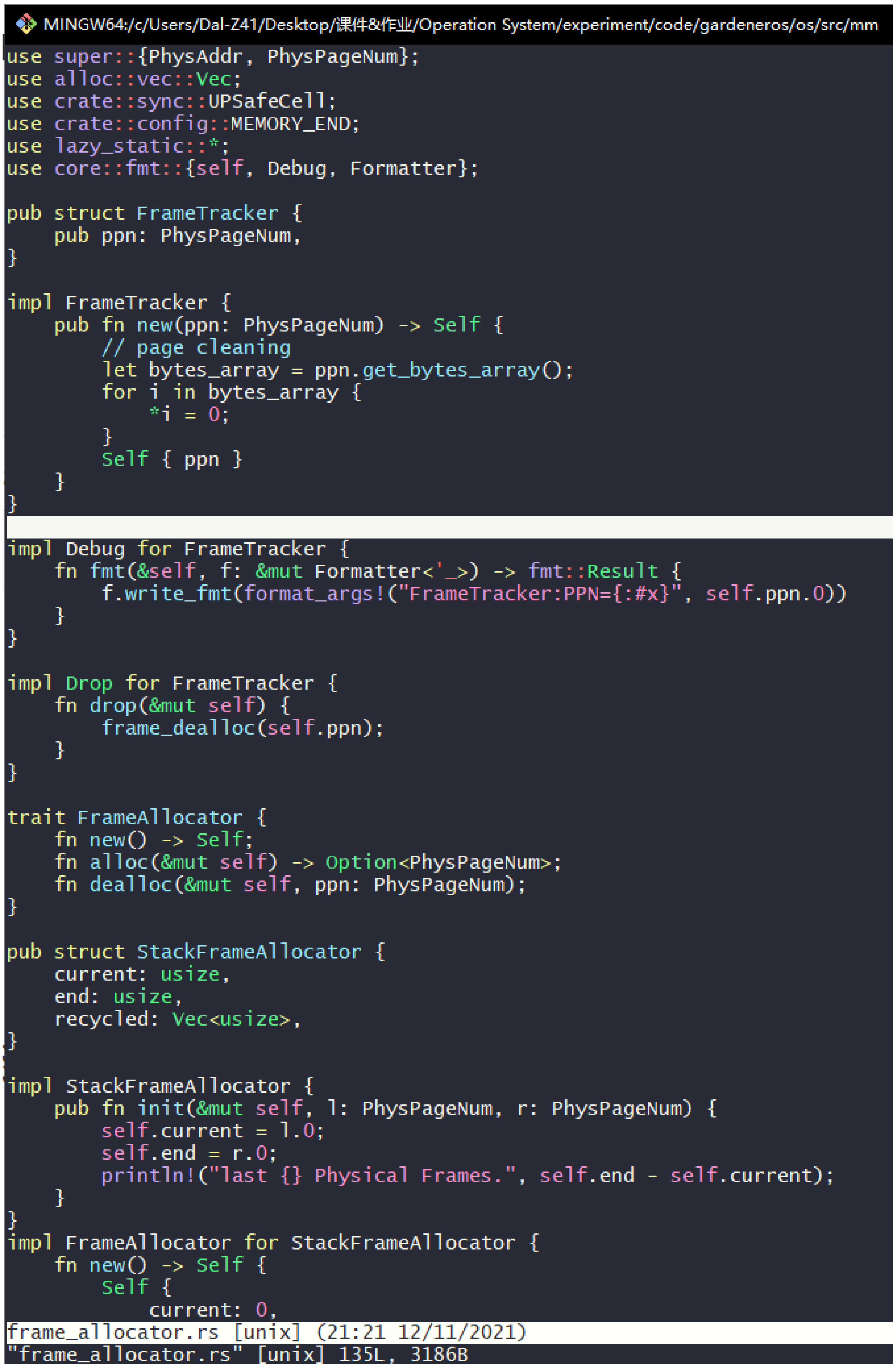
```
use super::{PhysAddr, PhysPageNum};
use alloc::vec::Vec;
use spin::Mutex;
use crate::config::MEMORY_END;
use lazy_static::*;
use core::fmt::{self, Debug, Formatter};

pub struct FrameTracker {
    pub ppn: PhysPageNum,
}

impl FrameTracker {
    pub fn new(ppn: PhysPageNum) -> Self {
        // page cleaning
        let bytes_array = ppn.get_bytes_array();
        for i in bytes_array {
            *i = 0;
        }
```

```rust
        Self { ppn }
    }
}

impl Debug for FrameTracker {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("FrameTracker:PPN={:#x}", self.ppn.0))
    }
}

impl Drop for FrameTracker {
    fn drop(&mut self) {
        frame_dealloc(self.ppn);
    }
}

trait FrameAllocator {
    fn new() -> Self;
    fn alloc(&mut self) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum);
}

pub struct StackFrameAllocator {
    current: usize,
    end: usize,
    recycled: Vec<usize>,
}

impl StackFrameAllocator {
    pub fn init(&mut self, l: PhysPageNum, r: PhysPageNum) {
        self.current = l.0;
        self.end = r.0;
        println!("last {} Physical Frames.", self.end - self.current);
    }
}
impl FrameAllocator for StackFrameAllocator {
    fn new() -> Self {
        Self {
            current: 0,
            end: 0,
            recycled: Vec::new(),
        }
    }
    fn alloc(&mut self) -> Option<PhysPageNum> {
        if let Some(ppn) = self.recycled.pop() {
            Some(ppn.into())
        } else {
            if self.current == self.end {
                None
            } else {
                self.current += 1;
                Some((self.current - 1).into())
            }
        }
    }
    fn dealloc(&mut self, ppn: PhysPageNum) {
        let ppn = ppn.0;
        // validity check
        if ppn >= self.current || self.recycled
            .iter()
            .find(|&v| {*v == ppn})
            .is_some() {
            panic!("Frame ppn={:#x} has not been allocated!", ppn);
        }
        // recycle
        self.recycled.push(ppn);
    }
}

type FrameAllocatorImpl = StackFrameAllocator;

lazy_static! {
    pub static ref FRAME_ALLOCATOR: UPSafeCell<FrameAllocatorImpl> = unsafe {
        UPSafeCell::new(FrameAllocatorImpl::new())
    };
}

pub fn init_frame_allocator() {
    extern "C" {
        fn ekernel();
    }
    FRAME_ALLOCATOR
        .exclusive_access()
        .init(PhysAddr::from(ekernel as usize).ceil(),
PhysAddr::from(MEMORY_END).floor());
}

pub fn frame_alloc() -> Option<FrameTracker> {
    FRAME_ALLOCATOR
        .exclusive_access()
        .alloc()
        .map(|ppn| FrameTracker::new(ppn))
}

fn frame_dealloc(ppn: PhysPageNum) {
    FRAME_ALLOCATOR
        .exclusive_access()
        .dealloc(ppn);
}

#[allow(unused)]
pub fn frame_allocator_test() {
    let mut v: Vec<FrameTracker> = Vec::new();
    for i in 0..5 {
        let frame = frame_alloc().unwrap();
        println!("{:?}", frame);
        v.push(frame);
    }
    v.clear();
    for i in 0..5 {
        let frame = frame_alloc().unwrap();
        println!("{:?}", frame);
        v.push(frame);
    }
    drop(v);
    println!("frame_allocator_test passed!");
```

```
}
```

Because the **mutex** is used to wrap the stack physical page-frame allocator, "**spin crate**" also needs to be introduced in the config file.

```
//os/Cargo.toml
spin = "0.7.0"
```



```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/src/mm
use super::{PhysAddr, PhysPageNum};
use alloc::vec::Vec;
use crate::sync::UPSafeCell;
use crate::config::MEMORY_END;
use lazy_static::*;
use core::fmt::{self, Debug, Formatter};

pub struct FrameTracker {
    pub ppn: PhysPageNum,
}

impl FrameTracker {
    pub fn new(ppn: PhysPageNum) -> Self {
        // page cleaning
        let bytes_array = ppn.get_bytes_array();
        for i in bytes_array {
            *i = 0;
        }
        Self { ppn }
    }
}

impl Debug for FrameTracker {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        f.write_fmt(format_args!("FrameTracker:PPN={:#x}", self.ppn.0))
    }
}

impl Drop for FrameTracker {
    fn drop(&mut self) {
        frame_dealloc(self.ppn);
    }
}

trait FrameAllocator {
    fn new() -> Self;
    fn alloc(&mut self) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum);
}

pub struct StackFrameAllocator {
    current: usize,
    end: usize,
    recycled: Vec<usize>,
}

impl StackFrameAllocator {
    pub fn init(&mut self, l: PhysPageNum, r: PhysPageNum) {
        self.current = l.0;
        self.end = r.0;
        println!("last {} Physical Frames.", self.end - self.current);
    }
}
impl FrameAllocator for StackFrameAllocator {
    fn new() -> Self {
        Self {
            current: 0,
frame_allocator.rs [unix] (21:21 12/11/2021)
"frame_allocator.rs" [unix] 135L, 3186B
```

## 4.3 Test

In order to implement the physical frame management test, we also need to modify some parts of the codes.

```
//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;

pub use address::{PhysAddr, VirtAddr, PhysPageNum, VirtPageNum, StepByOne};
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();
}
```

In the end, verify that the physical frame allocation test is successful by executing "**make run**".

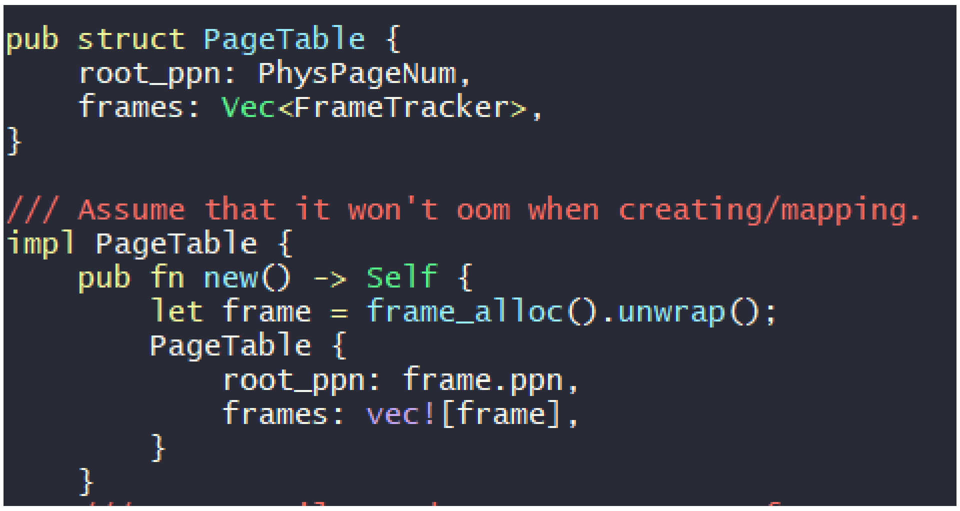
# 5. 多级页表管理

This section includes the implementations of multi-level page table management.

## 5.1 Page Table Structure

Implement the basic data structure of the page table.

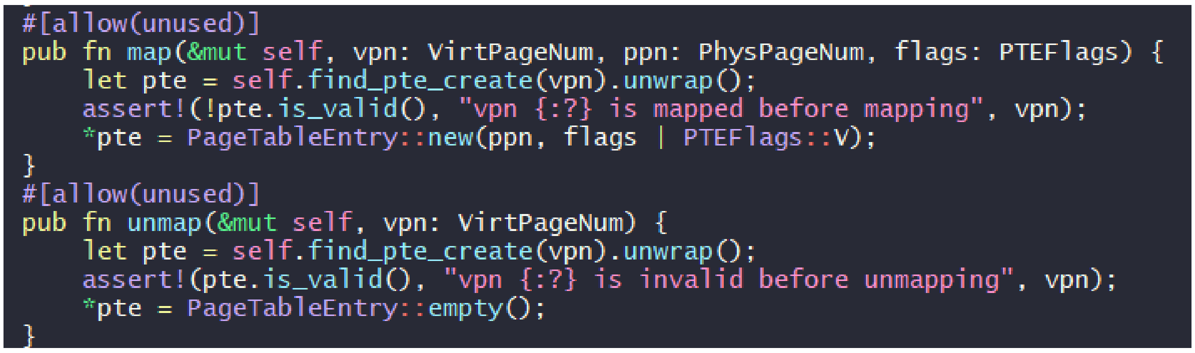
```
// os/src/mm/page_table.rs
use alloc::vec::Vec;
use alloc::vec;

pub struct PageTable {
    root_ppn: PhysPageNum,
    frames: Vec<FrameTracker>,
}

impl PageTable {
    pub fn new() -> Self {
        let frame = frame_alloc().unwrap();
        PageTable {
            root_ppn: frame.ppn,
            frames: vec![frame],
        }
    }
}
```


```
pub struct PageTable {
    root_ppn: PhysPageNum,
    frames: Vec<FrameTracker>,
}

/// Assume that it won't oom when creating/mapping.
impl PageTable {
    pub fn new() -> Self {
        let frame = frame_alloc().unwrap();
        PageTable {
            root_ppn: frame.ppn,
            frames: vec![frame],
        }
    }
```

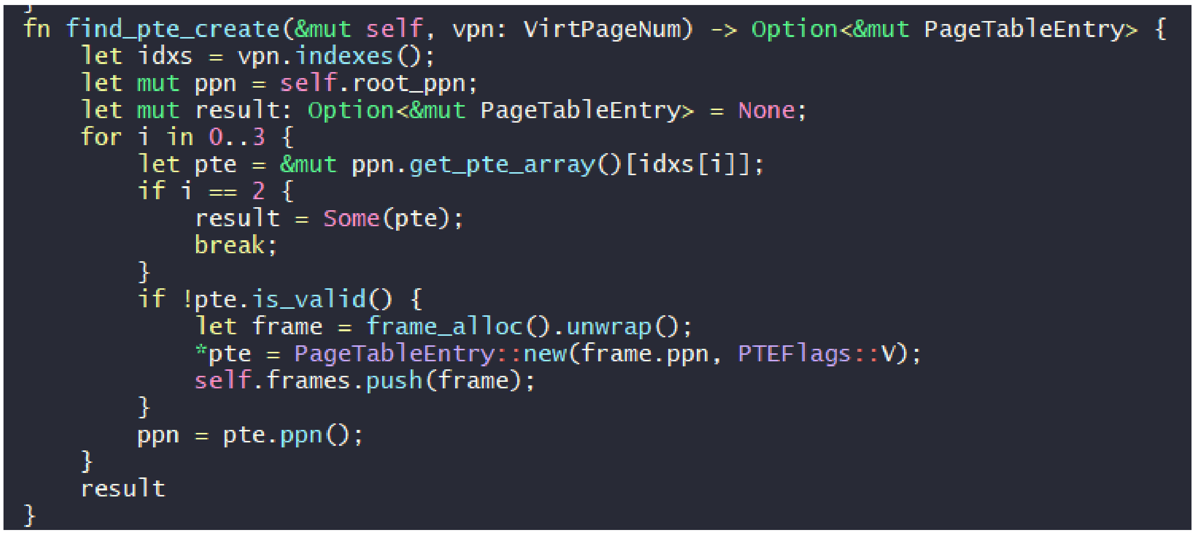
## 5.2 Mapping between Virtual & Physical

Besides, we need to be able to establish and remove the mapping between virtual and physical addresses.

```
// os/src/mm/page_table.rs

impl PageTable {
    fn find_pte_create(&mut self, vpn: VirtPageNum) -> Option<&mut
PageTableEntry> {
        let idxs = vpn.indexes();
        let mut ppn = self.root_ppn;
        let mut result: Option<&mut PageTableEntry> = None;
        for i in 0..3 {
            let pte = &mut ppn.get_pte_array()[idxs[i]];
            if i == 2 {
                result = Some(pte);
                break;
            }
            if !pte.is_valid() {
                let frame = frame_alloc().unwrap();
                *pte = PageTableEntry::new(frame.ppn, PTEFlags::V);
                self.frames.push(frame);
            }
            ppn = pte.ppn();
        }
        result
    }

    #[allow(unused)]
    pub fn map(&mut self, vpn: VirtPageNum, ppn: PhysPageNum, flags: PTEFlags) {
        let pte = self.find_pte_create(vpn).unwrap();
        assert!(!pte.is_valid(), "vpn {:?} is mapped before mapping", vpn);
        *pte = PageTableEntry::new(ppn, flags | PTEFlags::V);
    }

    #[allow(unused)]
    pub fn unmap(&mut self, vpn: VirtPageNum) {
        let pte = self.find_pte_create(vpn).unwrap();
        assert!(pte.is_valid(), "vpn {:?} is invalid before unmapping", vpn);
        *pte = PageTableEntry::empty();
    }
}
```



```
#[allow(unused)]
pub fn map(&mut self, vpn: VirtPageNum, ppn: PhysPageNum, flags: PTEFlags) {
    let pte = self.find_pte_create(vpn).unwrap();
    assert!(!pte.is_valid(), "vpn {:?} is mapped before mapping", vpn);
    *pte = PageTableEntry::new(ppn, flags | PTEFlags::V);
}
#[allow(unused)]
pub fn unmap(&mut self, vpn: VirtPageNum) {
    let pte = self.find_pte_create(vpn).unwrap();
    assert!(pte.is_valid(), "vpn {:?} is invalid before unmapping", vpn);
    *pte = PageTableEntry::empty();
}
```

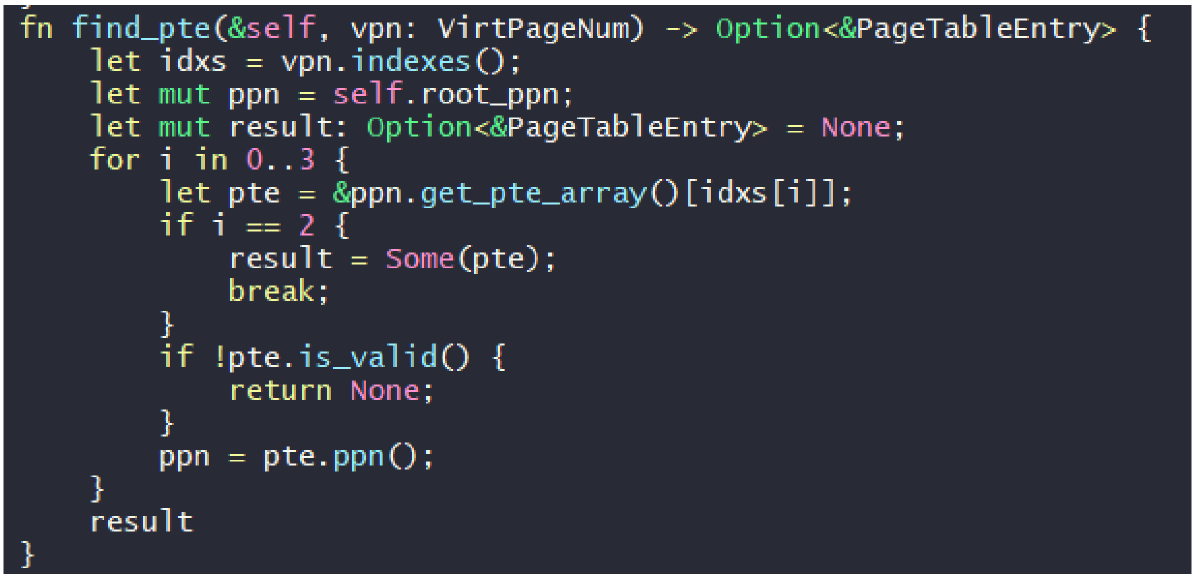


```
fn find_pte_create(&mut self, vpn: VirtPageNum) -> Option<&mut PageTableEntry> {
    let idxs = vpn.indexes();
    let mut ppn = self.root_ppn;
    let mut result: Option<&mut PageTableEntry> = None;
    for i in 0..3 {
        let pte = &mut ppn.get_pte_array()[idxs[i]];
        if i == 2 {
            result = Some(pte);
            break;
        }
        if !pte.is_valid() {
            let frame = frame_alloc().unwrap();
            *pte = PageTableEntry::new(frame.ppn, PTEFlags::V);
            self.frames.push(frame);
        }
        ppn = pte.ppn();
    }
    result
}
```

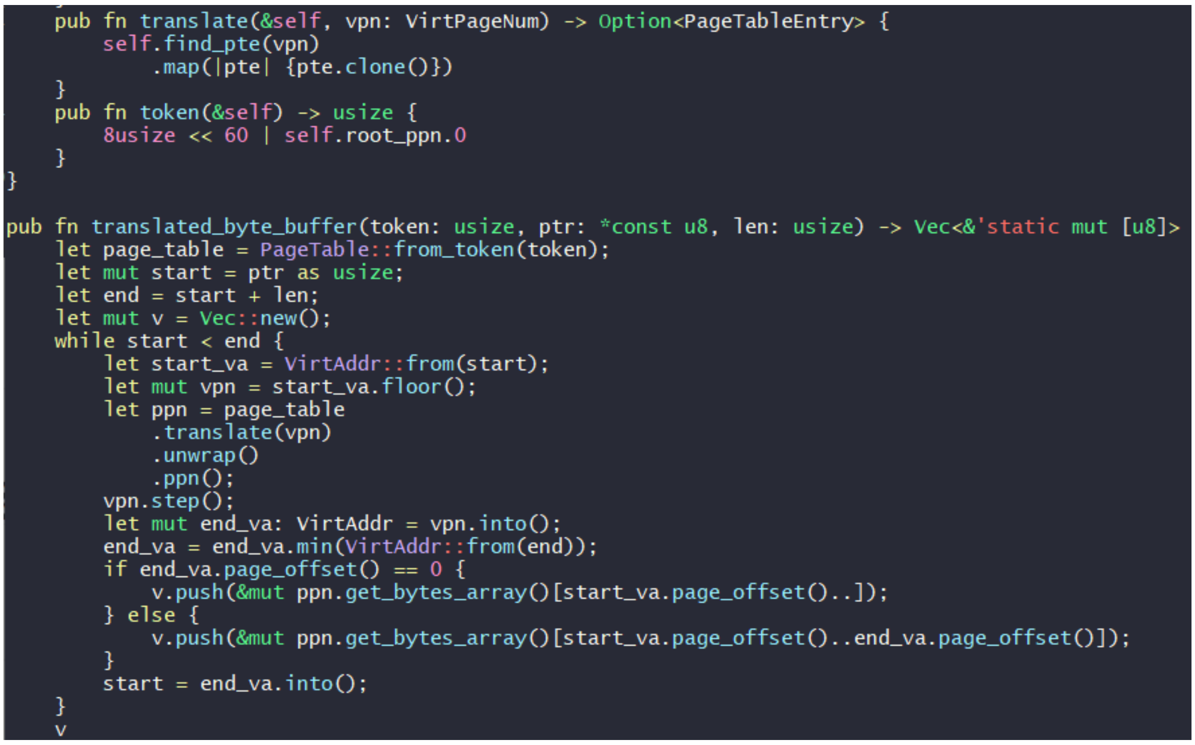
## 5.3 Manual Query

Meanwhile, in order to facilitate the subsequent implementation, also provides a manual query page table method

```
// os/src/mm/page_table.rs

impl PageTable {
    /// Temporarily used to get arguments from user space.
    pub fn from_token(satp: usize) -> Self {
        Self {
            root_ppn: PhysPageNum::from(satp & ((1usize << 44) - 1)),
            frames: Vec::new(),
        }
    }

    fn find_pte(&self, vpn: VirtPageNum) -> Option<&PageTableEntry> {
        let idxs = vpn.indexes();
        let mut ppn = self.root_ppn;
        let mut result: Option<&PageTableEntry> = None;
        for i in 0..3 {
            let pte = &ppn.get_pte_array()[idxs[i]];
            if i == 2 {
                result = Some(pte);
                break;
            }
            if !pte.is_valid() {
                return None;
            }
            ppn = pte.ppn();
        }
        result
    }

    pub fn translate(&self, vpn: VirtPageNum) -> Option<PageTableEntry> {
        self.find_pte(vpn)
            .map(|pte| {pte.clone()})
    }
    pub fn token(&self) -> usize {
        8usize << 60 | self.root_ppn.0
    }
}
```



```
fn find_pte(&self, vpn: VirtPageNum) -> Option<&PageTableEntry> {
    let idxs = vpn.indexes();
    let mut ppn = self.root_ppn;
    let mut result: Option<&PageTableEntry> = None;
    for i in 0..3 {
        let pte = &ppn.get_pte_array()[idxs[i]];
        if i == 2 {
            result = Some(pte);
            break;
        }
        if !pte.is_valid() {
            return None;
        }
        ppn = pte.ppn();
    }
    result
}
```



```
    pub fn translate(&self, vpn: VirtPageNum) -> Option<PageTableEntry> {
        self.find_pte(vpn)
            .map(|pte| {pte.clone()})
    }
    pub fn token(&self) -> usize {
        8usize << 60 | self.root_ppn.0
    }
}

pub fn translated_byte_buffer(token: usize, ptr: *const u8, len: usize) -> Vec<&'static mut [u8]>
    let page_table = PageTable::from_token(token);
    let mut start = ptr as usize;
    let end = start + len;
    let mut v = Vec::new();
    while start < end {
        let start_va = VirtAddr::from(start);
        let mut vpn = start_va.floor();
        let ppn = page_table
            .translate(vpn)
            .unwrap()
            .ppn();
        vpn.step();
        let mut end_va: VirtAddr = vpn.into();
        end_va = end_va.min(VirtAddr::from(end));
        if end_va.page_offset() == 0 {
            v.push(&mut ppn.get_bytes_array()[start_va.page_offset()..]);
        } else {
            v.push(&mut ppn.get_bytes_array()[start_va.page_offset()..end_va.page_offset()]);
        }
        start = end_va.into();
    }
    v
```

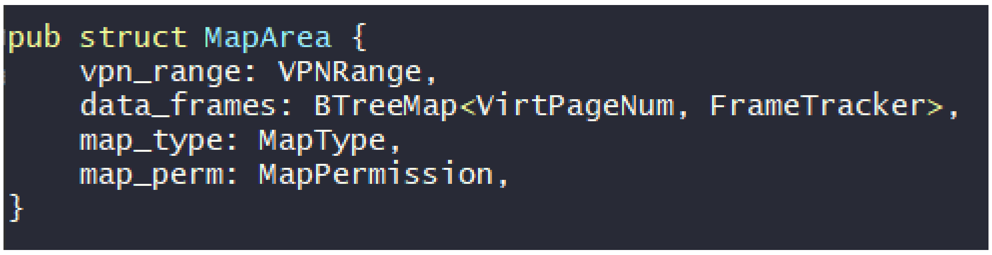
## 6. 内核与应用的地址空间

This section includes the implementations of the address space of the kernel and the application.

## 6.1 Address space Abstraction

1. Firstly, a logical segment **MapArea** is used to describe the virtual memory of a contiguous address. VPNRange is a continuous space of virtual page numbers. Implemented in the Address submodule.

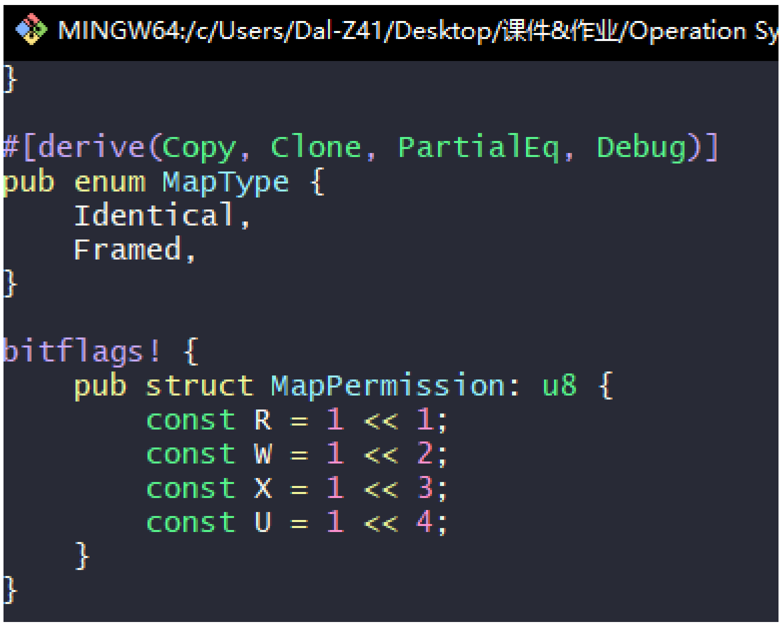
```rust
// os/src/mm/memory_set.rs

pub struct MapArea {
    vpn_range: VPNRange,
    data_frames: BTreeMap<VirtPageNum, FrameTracker>,
    map_type: MapType,
    map_perm: MapPermission,
}
```



```rust
pub struct MapArea {
    vpn_range: VPNRange,
    data_frames: BTreeMap<VirtPageNum, FrameTracker>,
    map_type: MapType,
    map_perm: MapPermission,
}
```

2. Next, MapType is used to describe how all virtual page numbers in the logical segment are mapped to physical page frames. At the same time, MapPermisssion is used to control the access mode of the logical segment, which is a subset of the PTEFlags of the page table entry.

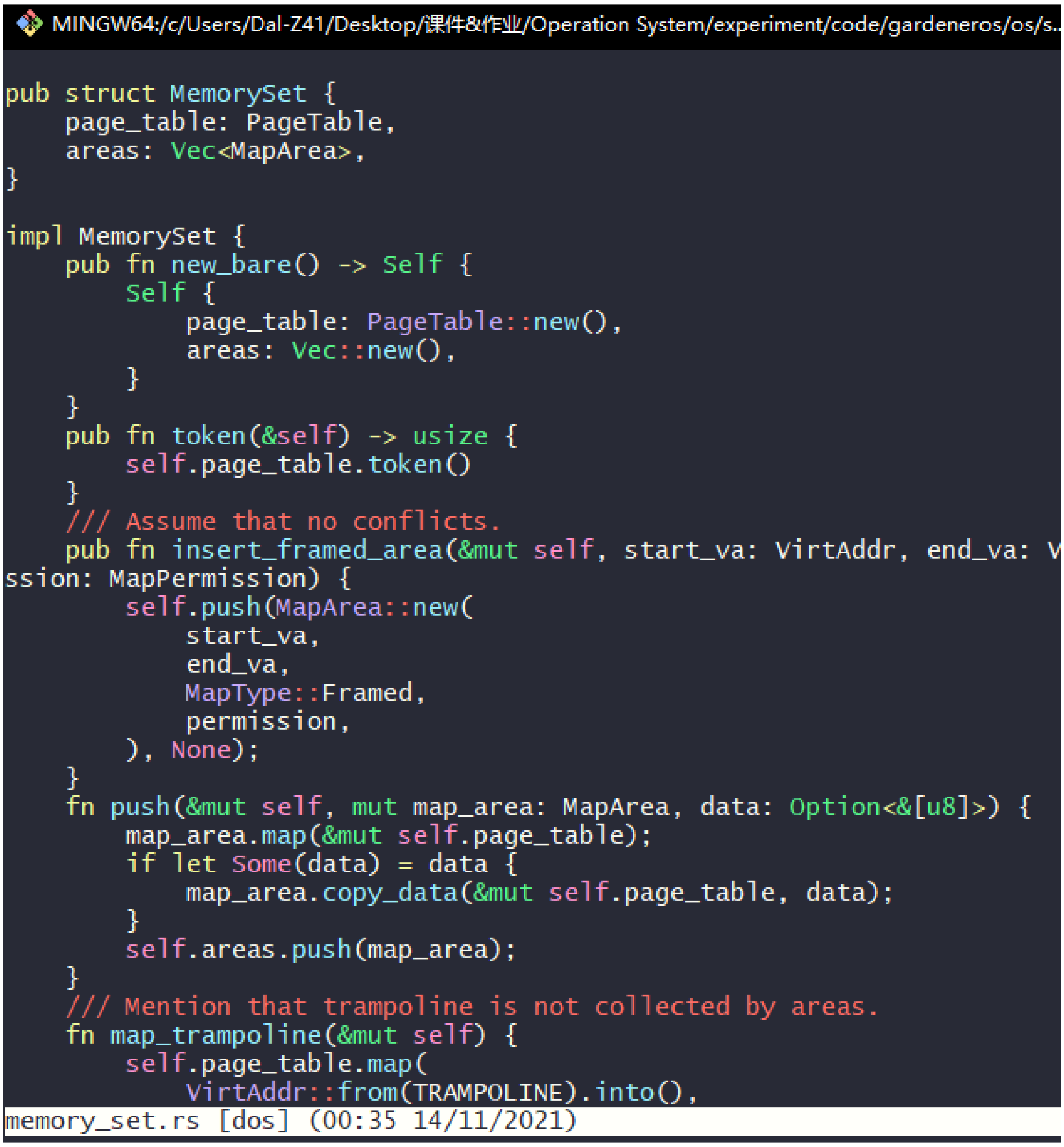
```rust
// os/src/mm/memory_set.rs

#[derive(Copy, Clone, PartialEq, Debug)]
pub enum MapType {
    Identical,
    Framed,
}

bitflags! {
    pub struct MapPermission: u8 {
        const R = 1 << 1;
        const W = 1 << 2;
        const X = 1 << 3;
        const U = 1 << 4;
    }
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation Sy

```
}

#[derive(Copy, Clone, PartialEq, Debug)]
pub enum MapType {
    Identical,
    Framed,
}

bitflags! {
    pub struct MapPermission: u8 {
        const R = 1 << 1;
        const W = 1 << 2;
        const X = 1 << 3;
        const U = 1 << 4;
    }
}
```

3. We can then implement an address space, which is a series of related logical segments, represented by a **MemorySet**. At the same time, the MemorySet method is implemented.

```
// os/src/mm/memory_set.rs

pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
}

impl MemorySet {
    pub fn new_bare() -> Self {
        Self {
            page_table: PageTable::new(),
            areas: Vec::new(),
        }
    }
    pub fn token(&self) -> usize {
        self.page_table.token()
    }
    /// Assume that no conflicts.
    pub fn insert_framed_area(&mut self, start_va: VirtAddr, end_va:
VirtAddr, permission: MapPermission) {
        self.push(MapArea::new(
            start_va,
            end_va,
            MapType::Framed,
            permission,
        ), None);
    }
    fn push(&mut self, mut map_area: MapArea, data: Option<&[u8]>) {
        map_area.map(&mut self.page_table);
        if let Some(data) = data {
            map_area.copy_data(&mut self.page_table, data);
        }
        self.areas.push(map_area);
    }
    /// Mention that trampoline is not collected by areas.
    fn map_trampoline(&mut self) {
        self.page_table.map(
            VirtAddr::from(TRAMPOLINE).into(),
            PhysAddr::from(strampoline as usize).into(),
            PTEFlags::R | PTEFlags::X,
        );
    }
    pub fn activate(&self) {
        let satp = self.page_table.token();
        unsafe {
            satp::write(satp);
            asm!("sfence.vma");
        }
    }
    pub fn translate(&self, vpn: VirtPageNum) -> Option<PageTableEntry> {
        self.page_table.translate(vpn)
    }
}
```

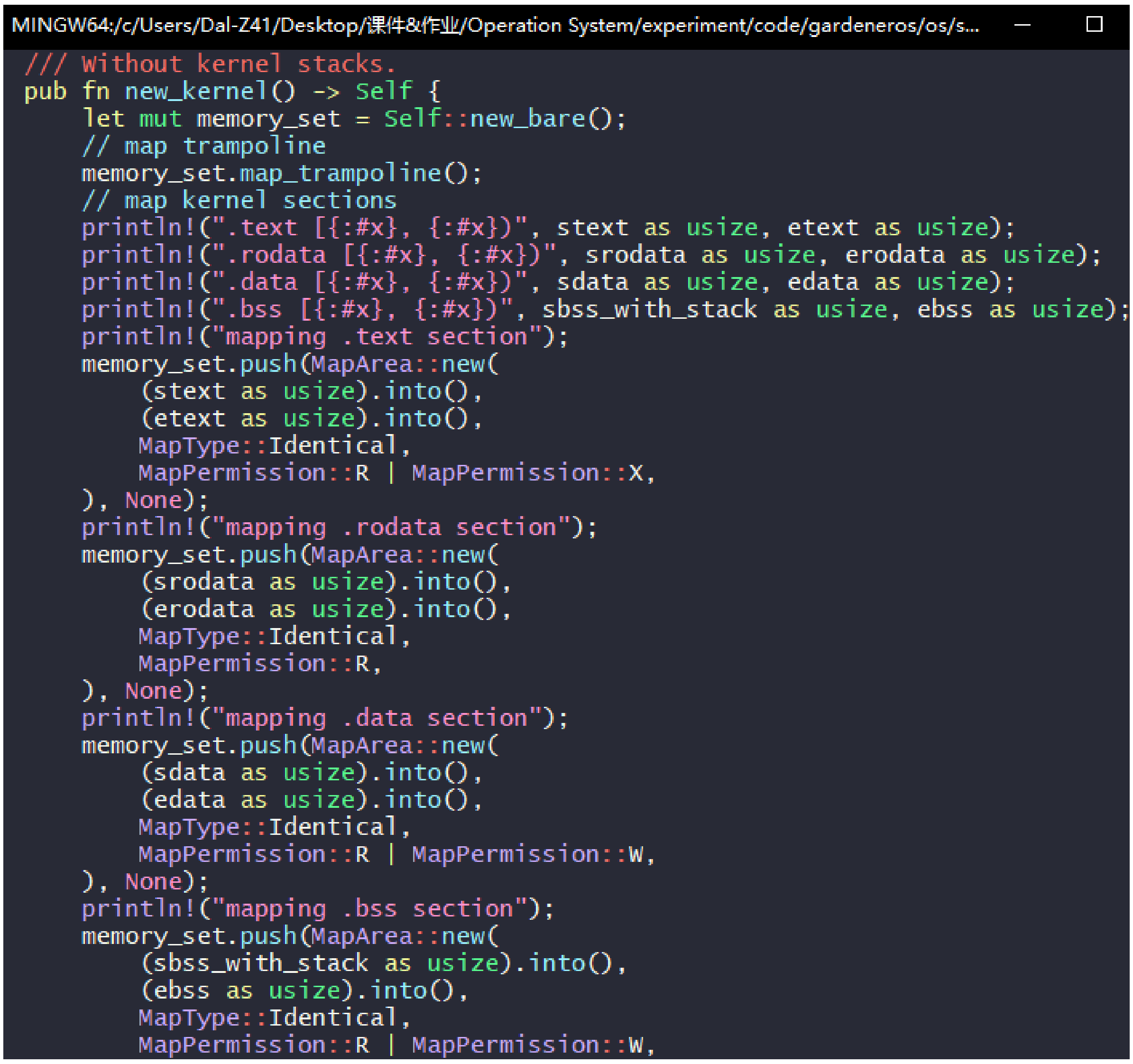


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/s..

```
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
}

impl MemorySet {
    pub fn new_bare() -> Self {
        Self {
            page_table: PageTable::new(),
            areas: Vec::new(),
        }
    }
    pub fn token(&self) -> usize {
        self.page_table.token()
    }
    /// Assume that no conflicts.
    pub fn insert_framed_area(&mut self, start_va: VirtAddr, end_va: V
ssion: MapPermission) {
        self.push(MapArea::new(
            start_va,
            end_va,
            MapType::Framed,
            permission,
        ), None);
    }
    fn push(&mut self, mut map_area: MapArea, data: Option<&[u8]>) {
        map_area.map(&mut self.page_table);
        if let Some(data) = data {
            map_area.copy_data(&mut self.page_table, data);
        }
        self.areas.push(map_area);
    }
    /// Mention that trampoline is not collected by areas.
    fn map_trampoline(&mut self) {
        self.page_table.map(
            VirtAddr::from(TRAMPOLINE).into(),
memory_set.rs [dos] (00:35 14/11/2021)
```

## 6.2 Kernel address space

Implement the method of creating the kernel address space **"new_kernel"**.

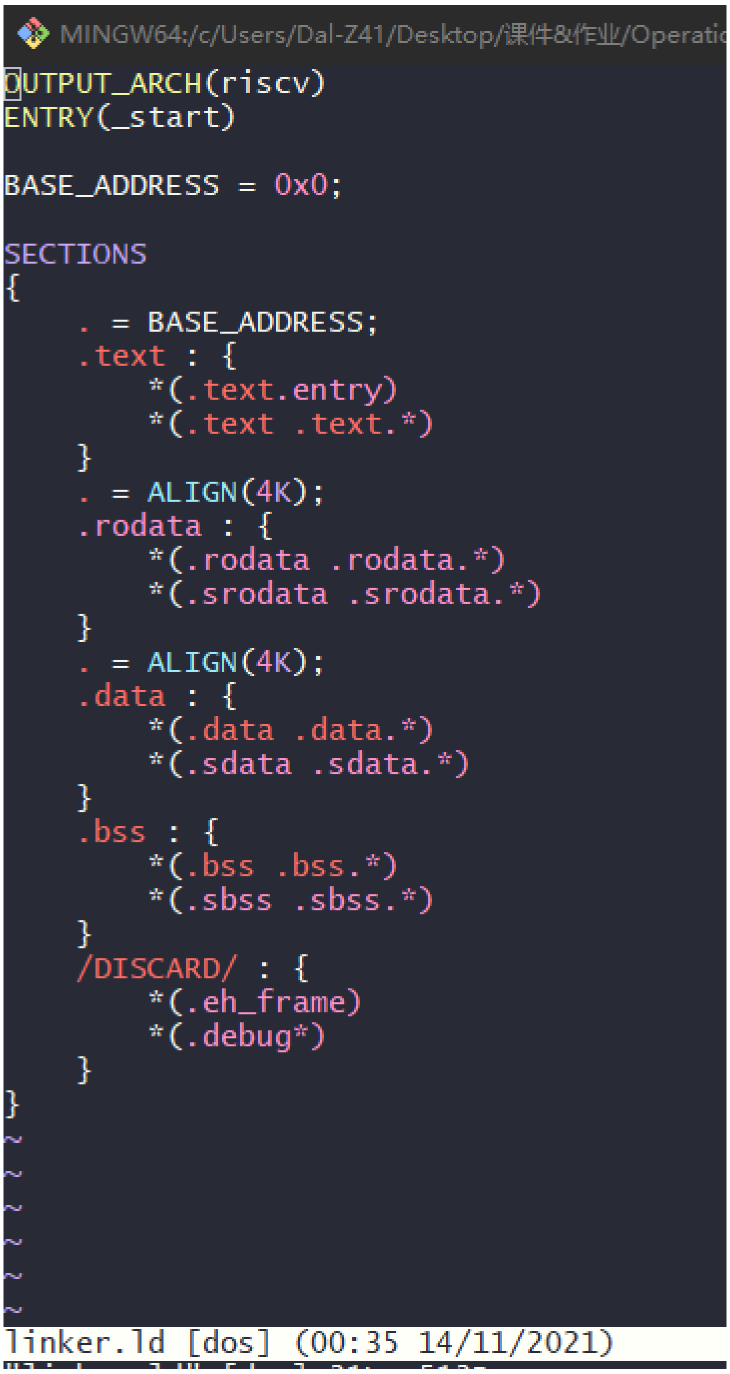

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/s...

```
/// Without kernel stacks.
pub fn new_kernel() -> Self {
    let mut memory_set = Self::new_bare();
    // map trampoline
    memory_set.map_trampoline();
    // map kernel sections
    println!(".text [{:#x}, {:#x})", stext as usize, etext as usize);
    println!(".rodata [{:#x}, {:#x})", srodata as usize, erodata as usize);
    println!(".data [{:#x}, {:#x})", sdata as usize, edata as usize);
    println!(".bss [{:#x}, {:#x})", sbss_with_stack as usize, ebss as usize);
    println!("mapping .text section");
    memory_set.push(MapArea::new(
        (stext as usize).into(),
        (etext as usize).into(),
        MapType::Identical,
        MapPermission::R | MapPermission::X,
    ), None);
    println!("mapping .rodata section");
    memory_set.push(MapArea::new(
        (srodata as usize).into(),
        (erodata as usize).into(),
        MapType::Identical,
        MapPermission::R,
    ), None);
    println!("mapping .data section");
    memory_set.push(MapArea::new(
        (sdata as usize).into(),
        (edata as usize).into(),
        MapType::Identical,
        MapPermission::R | MapPermission::W,
    ), None);
    println!("mapping .bss section");
    memory_set.push(MapArea::new(
        (sbss_with_stack as usize).into(),
        (ebss as usize).into(),
        MapType::Identical,
        MapPermission::R | MapPermission::W,
```

## 6.3 Application address space

Since the application is dynamically loaded, all applications use the same link script. Pay particular attention to changing BASE_ADDRESS to 0x0.

```
// user/src/linker.ld
OUTPUT_ARCH(riscv)
ENTRY(_start)

BASE_ADDRESS = 0x0;

SECTIONS
{
    . = BASE_ADDRESS;
    .text : {
        *(.text.entry)
        *(.text .text.*)
    }
    . = ALIGN(4K);
    .rodata : {
        *(.rodata .rodata.*)
        *(.srodata .srodata.*)
    }
    . = ALIGN(4K);
    .data : {
        *(.data .data.*)
        *(.sdata .sdata.*)
    }
    .bss : {
        start_bss = .;
        *(.bss .bss.*)
        *(.sbss .sbss.*)
        end_bss = .;
    }
    /DISCARD/ : {
        *(.eh_frame)
        *(.debug*)
    }
}
```

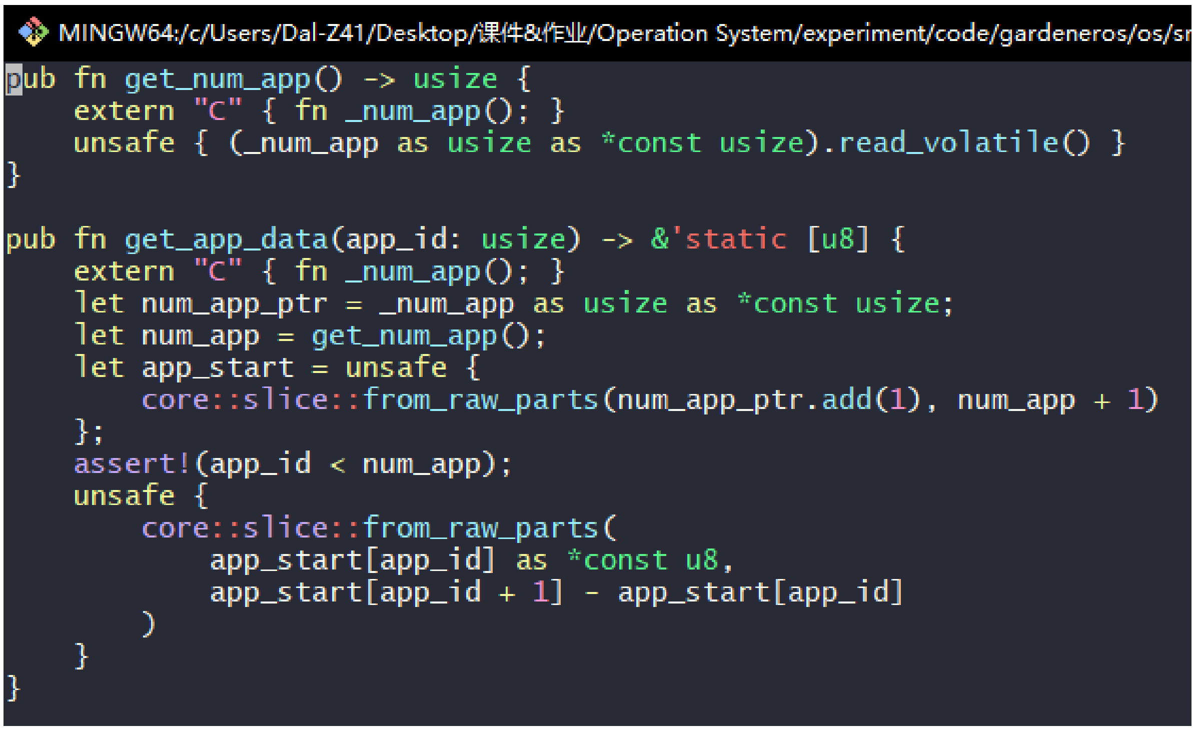


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operati

```
OUTPUT_ARCH(riscv)
ENTRY(_start)

BASE_ADDRESS = 0x0;

SECTIONS
{
    . = BASE_ADDRESS;
    .text : {
        *(.text.entry)
        *(.text .text.*)
    }
    . = ALIGN(4K);
    .rodata : {
        *(.rodata .rodata.*)
        *(.srodata .srodata.*)
    }
    . = ALIGN(4K);
    .data : {
        *(.data .data.*)
        *(.sdata .sdata.*)
    }
    .bss : {
        *(.bss .bss.*)
        *(.sbss .sbss.*)
    }
    /DISCARD/ : {
        *(.eh_frame)
        *(.debug*)
    }
}
~
~
~
~
~
~
linker.ld [dos] (00:35 14/11/2021)
```

And we can modify the thin Loader submodule now that we can use ELF executable files directly. At the same time, the ELF format data needs to be parsed to get a complete application address space. Because xMAS-ELF is used, you need to add dependencies in the configuration file Cargo. Toml.

```
// os/src/loader.rs
pub fn get_num_app() -> usize {
    extern "C" { fn _num_app(); }
    unsafe { (_num_app as usize as *const usize).read_volatile() }
}

pub fn get_app_data(app_id: usize) -> &'static [u8] {
    extern "C" { fn _num_app(); }
    let num_app_ptr = _num_app as usize as *const usize;
    let num_app = get_num_app();
    let app_start = unsafe {
        core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1)
    };
    assert!(app_id < num_app);
    unsafe {
        core::slice::from_raw_parts(
            app_start[app_id] as *const u8,
            app_start[app_id + 1] - app_start[app_id]
        )
    }
}

// os/Cargo.toml
xmas-elf = "0.7.0"
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/sr

```
pub fn get_num_app() -> usize {
    extern "C" { fn _num_app(); }
    unsafe { (_num_app as usize as *const usize).read_volatile() }
}

pub fn get_app_data(app_id: usize) -> &'static [u8] {
    extern "C" { fn _num_app(); }
    let num_app_ptr = _num_app as usize as *const usize;
    let num_app = get_num_app();
    let app_start = unsafe {
        core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1)
    };
    assert!(app_id < num_app);
    unsafe {
        core::slice::from_raw_parts(
            app_start[app_id] as *const u8,
            app_start[app_id + 1] - app_start[app_id]
        )
    }
}
```

Finally, note that implementing the memory_set submodule also adds the following code:

```
//os/src/mm/memory_set.rs
use super::{PageTable, PageTableEntry, PTEFlags};
use super::{VirtPageNum, VirtAddr, PhysPageNum, PhysAddr};
use super::{FrameTracker, frame_alloc};
```

```rust
use super::{VPNRange, StepByOne};
use alloc::collections::BTreeMap;
use alloc::vec::Vec;
use riscv::register::satp;
use alloc::sync::Arc;
use lazy_static::*;
use crate::sync::UPSafeCell;
use crate::config::{
    MEMORY_END,
    PAGE_SIZE,
    TRAMPOLINE,
    TRAP_CONTEXT,
    USER_STACK_SIZE
};

//os/src/config.rs
pub const TRAMPOLINE: usize = usize::MAX - PAGE_SIZE + 1;
pub const TRAP_CONTEXT: usize = TRAMPOLINE - PAGE_SIZE;
```

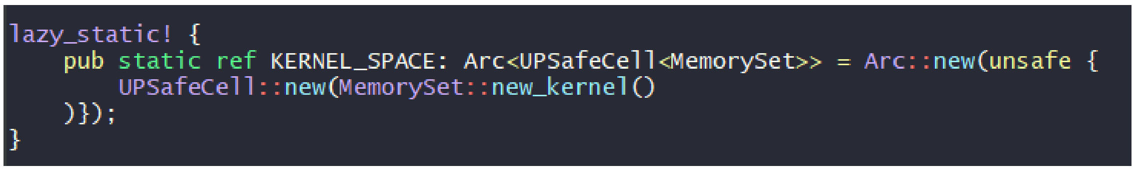
# 7. 实现基于地址空间的分时多任务

This section includes the implementations of the time-sharing multi-task based on address space.

## 7.1 Virtual address space on paging mode

1. First, create the kernel address space.

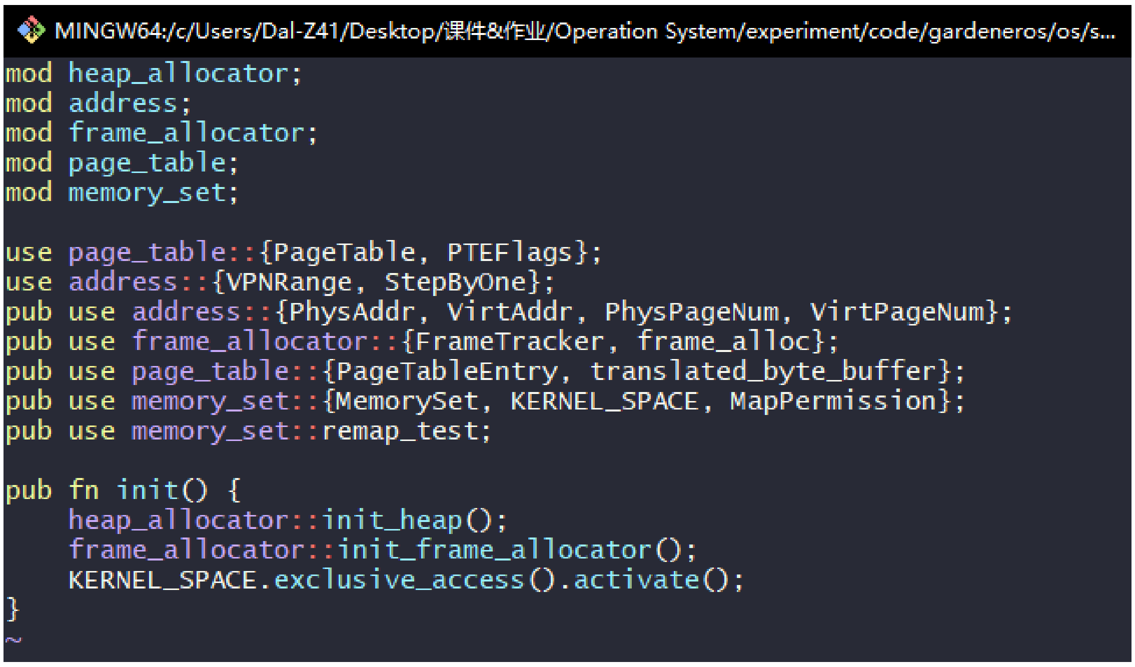
```rust
// os/src/mm/memory_set.rs
lazy_static! {
    pub static ref KERNEL_SPACE: Arc<UPSafeCell<MemorySet>> =
Arc::new(unsafe {
        UPSafeCell::new(MemorySet::new_kernel()
    )});
}
```



```rust
lazy_static! {
    pub static ref KERNEL_SPACE: Arc<UPSafeCell<MemorySet>> = Arc::new(unsafe {
        UPSafeCell::new(MemorySet::new_kernel()
    )});
}
```

2. The memory management subsystem is then initialized in rust_main.

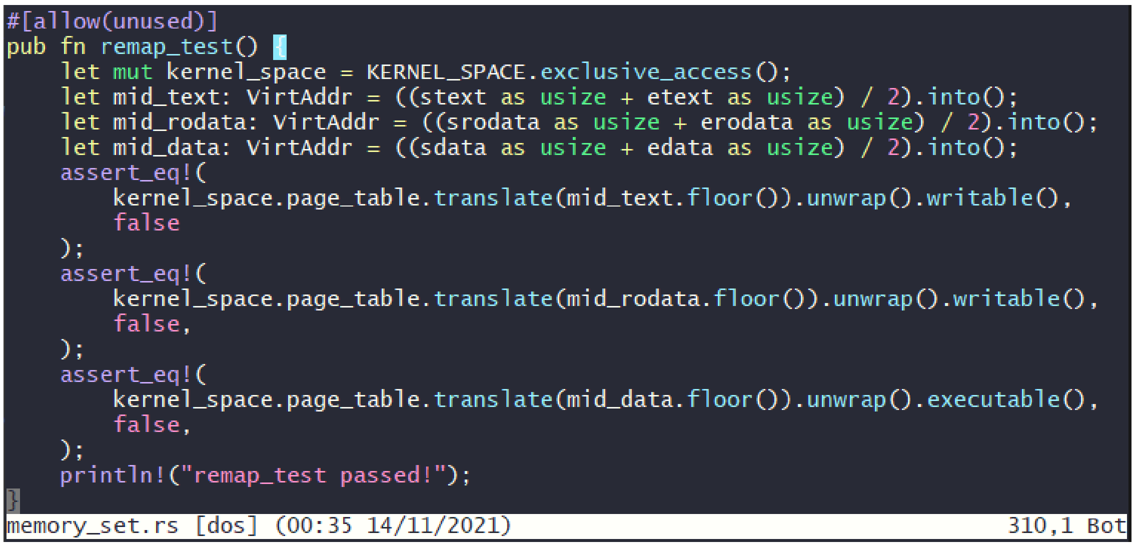
```rust
// os/src/mm/mod.rs
mod memory_set;

use page_table::{PageTable, PTEFlags};
pub use frame_allocator::{FrameTracker, frame_alloc, frame_dealloc};
pub use page_table::{PageTableEntry, translated_byte_buffer};
pub use memory_set::{MemorySet, KERNEL_SPACE, MapPermission};

pub fn init() {
    heap_allocator::init_heap();
    frame_allocator::init_frame_allocator();
    KERNEL_SPACE.exclusive_access().activate();
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/s...

```
mod heap_allocator;
mod address;
mod frame_allocator;
mod page_table;
mod memory_set;

use page_table::{PageTable, PTEFlags};
use address::{VPNRange, StepByOne};
pub use address::{PhysAddr, VirtAddr, PhysPageNum, VirtPageNum};
pub use frame_allocator::{FrameTracker, frame_alloc};
pub use page_table::{PageTableEntry, translated_byte_buffer};
pub use memory_set::{MemorySet, KERNEL_SPACE, MapPermission};
pub use memory_set::remap_test;

pub fn init() {
    heap_allocator::init_heap();
    frame_allocator::init_frame_allocator();
    KERNEL_SPACE.exclusive_access().activate();
}
~
```

3. Next, check the multi-level page table Settings for the kernel address space.

```rust
// os/src/mm/memory_set.rs
#[allow(unused)]
pub fn remap_test() {
    let mut kernel_space = KERNEL_SPACE.exclusive_access();
    let mid_text: VirtAddr = ((stext as usize + etext as usize) / 2).into();
    let mid_rodata: VirtAddr = ((srodata as usize + erodata as usize) /
2).into();
    let mid_data: VirtAddr = ((sdata as usize + edata as usize) / 2).into();
    assert_eq!(

 kernel_space.page_table.translate(mid_text.floor()).unwrap().writable(),
        false
    );
    assert_eq!(

 kernel_space.page_table.translate(mid_rodata.floor()).unwrap().writable(),
        false,
    );
    assert_eq!(

 kernel_space.page_table.translate(mid_data.floor()).unwrap().executable(),
        false,
    );
    println!("remap_test passed!");
}
```



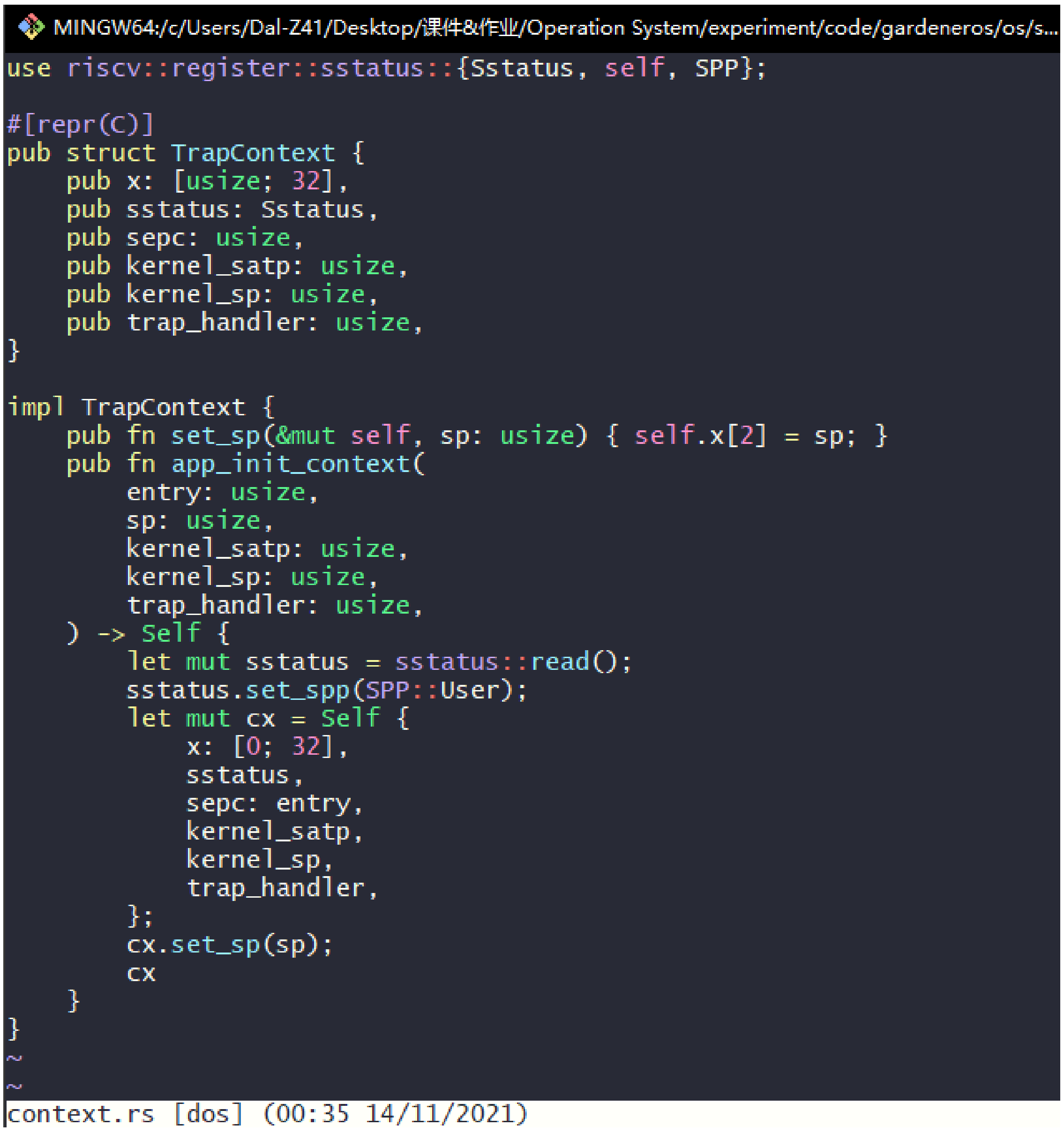
```
#[allow(unused)]
pub fn remap_test() {
    let mut kernel_space = KERNEL_SPACE.exclusive_access();
    let mid_text: VirtAddr = ((stext as usize + etext as usize) / 2).into();
    let mid_rodata: VirtAddr = ((srodata as usize + erodata as usize) / 2).into();
    let mid_data: VirtAddr = ((sdata as usize + edata as usize) / 2).into();
    assert_eq!(
        kernel_space.page_table.translate(mid_text.floor()).unwrap().writable(),
        false
    );
    assert_eq!(
        kernel_space.page_table.translate(mid_rodata.floor()).unwrap().writable(),
        false,
    );
    assert_eq!(
        kernel_space.page_table.translate(mid_data.floor()).unwrap().executable(),
        false,
    );
    println!("remap_test passed!");
}
memory_set.rs [dos] (00:35 14/11/2021)                                310,1 Bot
```

## 7.2 Springboard Mechanism

This page is called a springboard page because the assembly code for this page switches address Spaces during execution.

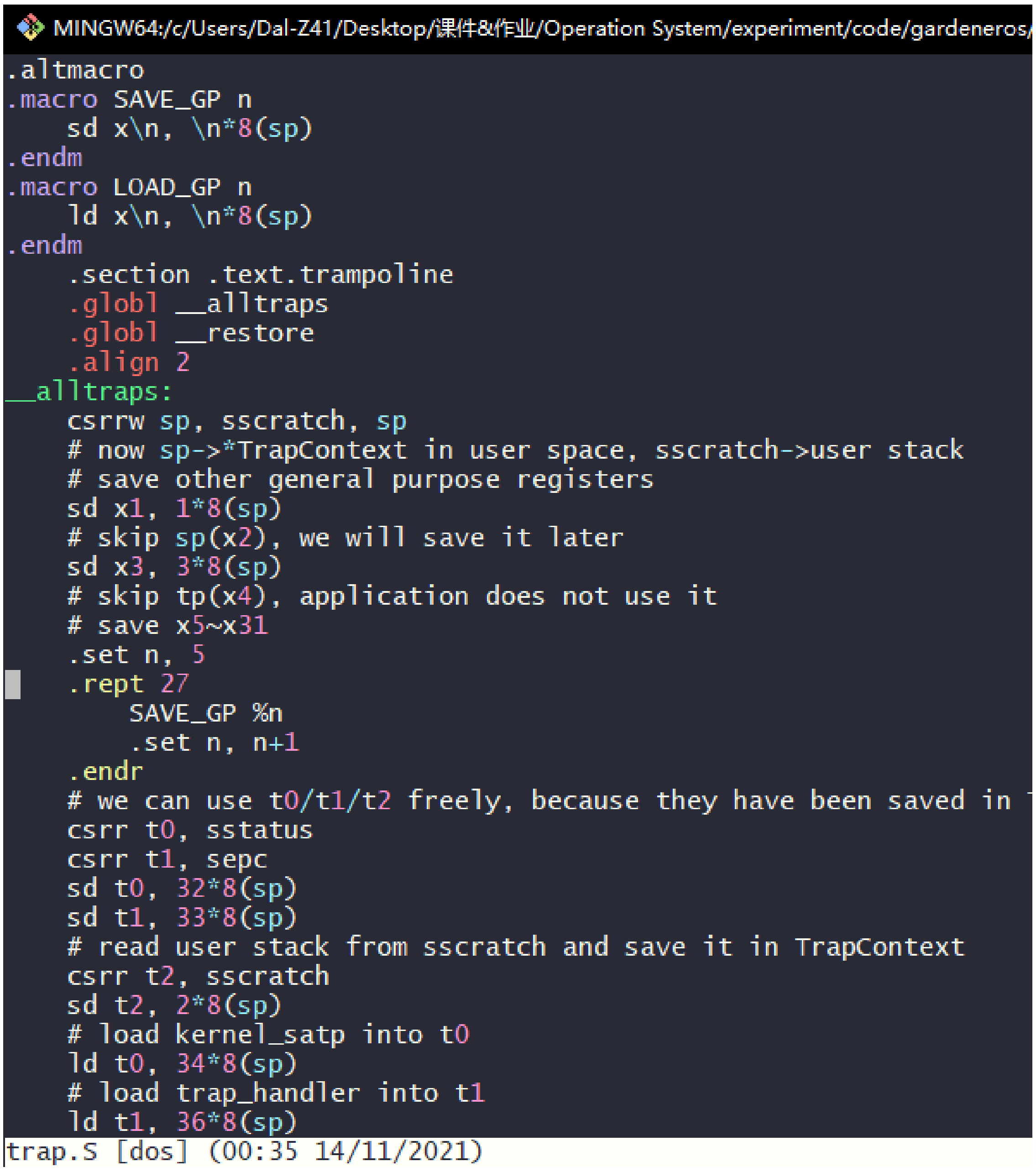
1. First, extend the Trap protocol.

```
//os/src/trap/context.rs
#[repr(C)]
pub struct TrapContext {
    pub x: [usize; 32],
    pub sstatus: Sstatus,
    pub sepc: usize,
    pub kernel_satp: usize,
    pub kernel_sp: usize,
    pub trap_handler: usize,
}

impl TrapContext {
    pub fn set_sp(&mut self, sp: usize) { self.x[2] = sp; }
    pub fn app_init_context(
        entry: usize,
        sp: usize,
        kernel_satp: usize,
        kernel_sp: usize,
        trap_handler: usize,
    ) -> Self {
        let mut sstatus = sstatus::read();
        sstatus.set_spp(SPP::User);
        let mut cx = Self {
            x: [0; 32],
            sstatus,
            sepc: entry,
            kernel_satp,
            kernel_sp,
            trap_handler,
        };
        cx.set_sp(sp);
        cx
    }
}
```



```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/s...
use riscv::register::sstatus::{Sstatus, self, SPP};

#[repr(C)]
pub struct TrapContext {
    pub x: [usize; 32],
    pub sstatus: Sstatus,
    pub sepc: usize,
    pub kernel_satp: usize,
    pub kernel_sp: usize,
    pub trap_handler: usize,
}

impl TrapContext {
    pub fn set_sp(&mut self, sp: usize) { self.x[2] = sp; }
    pub fn app_init_context(
        entry: usize,
        sp: usize,
        kernel_satp: usize,
        kernel_sp: usize,
        trap_handler: usize,
    ) -> Self {
        let mut sstatus = sstatus::read();
        sstatus.set_spp(SPP::User);
        let mut cx = Self {
            x: [0; 32],
            sstatus,
            sepc: entry,
            kernel_satp,
            kernel_sp,
            trap_handler,
        };
        cx.set_sp(sp);
        cx
    }
}
~
~
context.rs [dos] (00:35 14/11/2021)
```

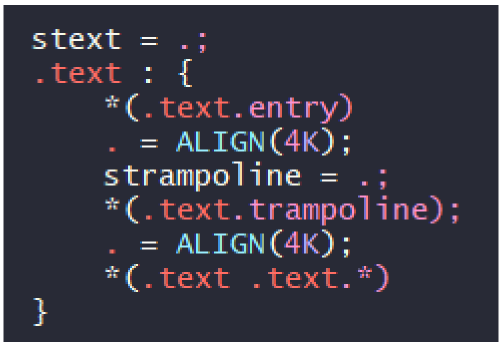
2. Then, the address space is switched.


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/

```
.altmacro
.macro SAVE_GP n
    sd x\n, \n*8(sp)
.endm
.macro LOAD_GP n
    ld x\n, \n*8(sp)
.endm
    .section .text.trampoline
    .globl __alltraps
    .globl __restore
    .align 2
__alltraps:
    csrrw sp, sscratch, sp
    # now sp->*TrapContext in user space, sscratch->user stack
    # save other general purpose registers
    sd x1, 1*8(sp)
    # skip sp(x2), we will save it later
    sd x3, 3*8(sp)
    # skip tp(x4), application does not use it
    # save x5~x31
    .set n, 5
    .rept 27
        SAVE_GP %n
        .set n, n+1
    .endr
    # we can use t0/t1/t2 freely, because they have been saved in
    csrr t0, sstatus
    csrr t1, sepc
    sd t0, 32*8(sp)
    sd t1, 33*8(sp)
    # read user stack from sscratch and save it in TrapContext
    csrr t2, sscratch
    sd t2, 2*8(sp)
    # load kernel_satp into t0
    ld t0, 34*8(sp)
    # load trap_handler into t1
    ld t1, 36*8(sp)
trap.S [dos] (00:35 14/11/2021)
```

3. Then, create the springboard page. Place the entire assembly code in "**trap.s**" in ".text.trampoline" segment and align it to a page of the segment when adjusting the memory layout.

```
# os/src/linker.ld

stext = .;
.text : {
    *(.text.entry)
    . = ALIGN(4K);
    strampoline = .;
    *(.text.trampoline);
    . = ALIGN(4K);
    *(.text .text.*)
}
```

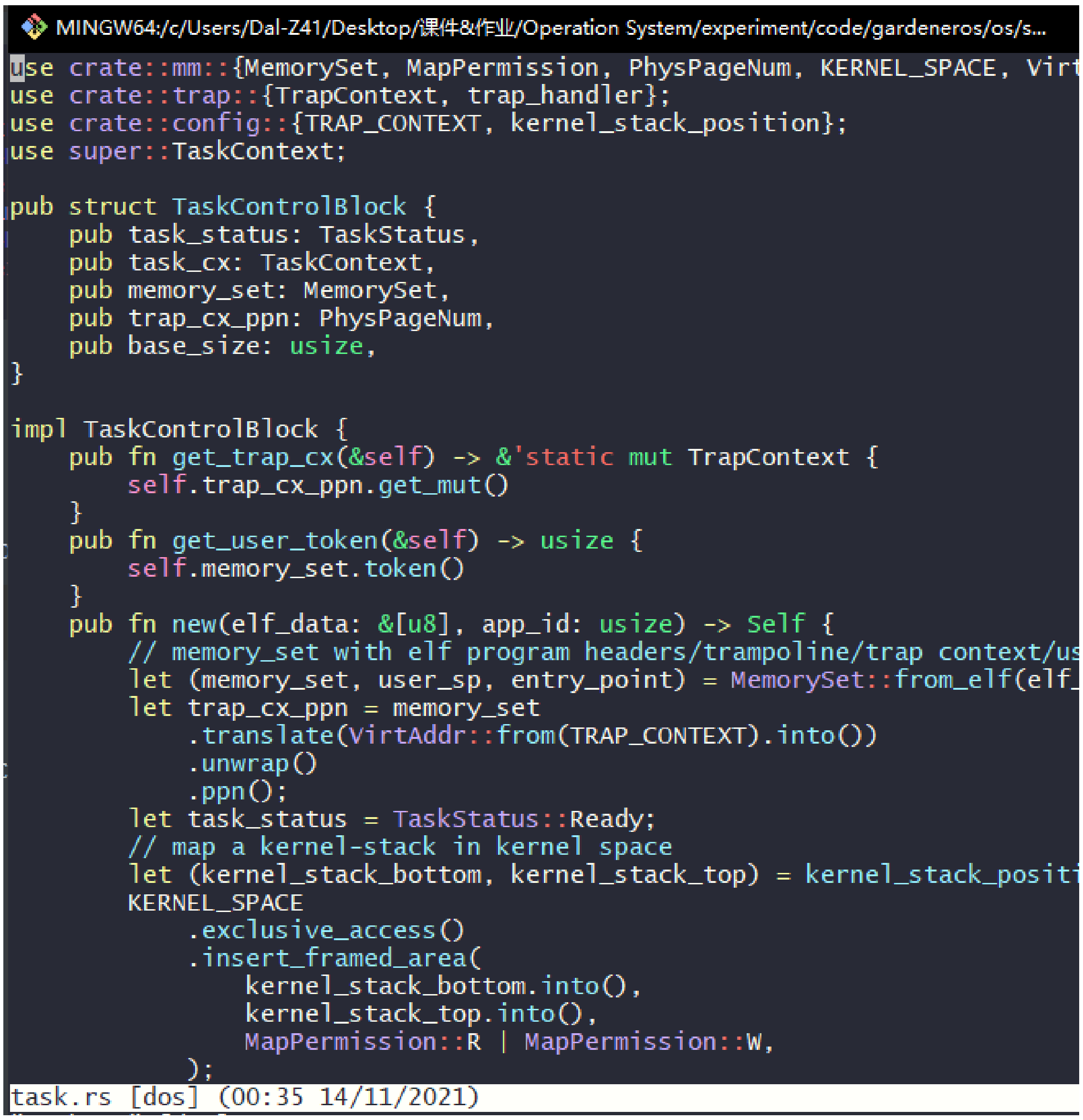


```
stext = .;
.text : {
    *(.text.entry)
    . = ALIGN(4K);
    strampoline = .;
    *(.text.trampoline);
    . = ALIGN(4K);
    *(.text .text.*)
}
```

## 7.3 Application loading and execution

1. First, modify the task submodule and update the management of the task control block.

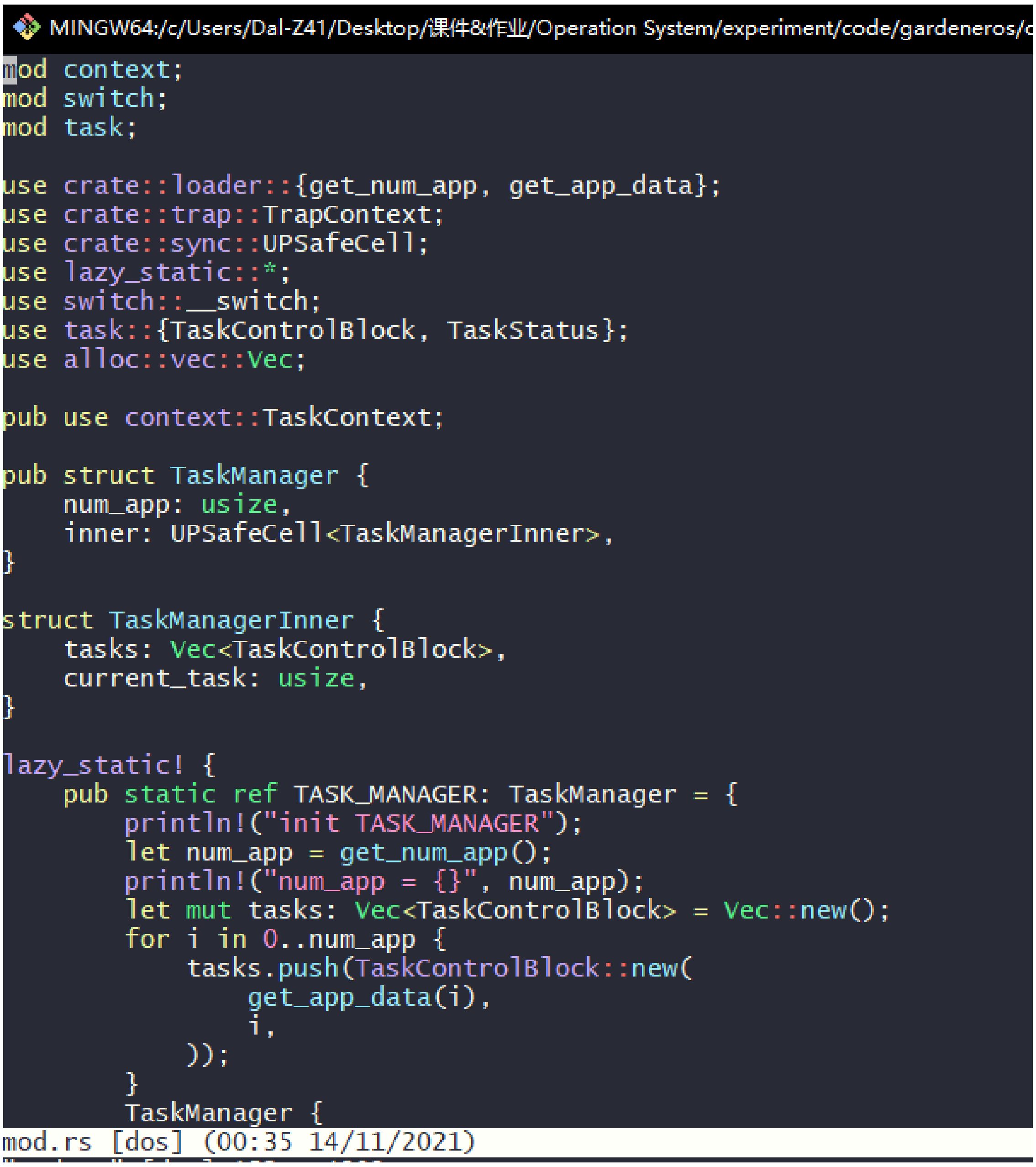
```
// os/src/config.rs
/// Return (bottom, top) of a kernel stack in kernel space.
pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
    let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
    let bottom = top - KERNEL_STACK_SIZE;
    (bottom, top)
}
```



MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/s...

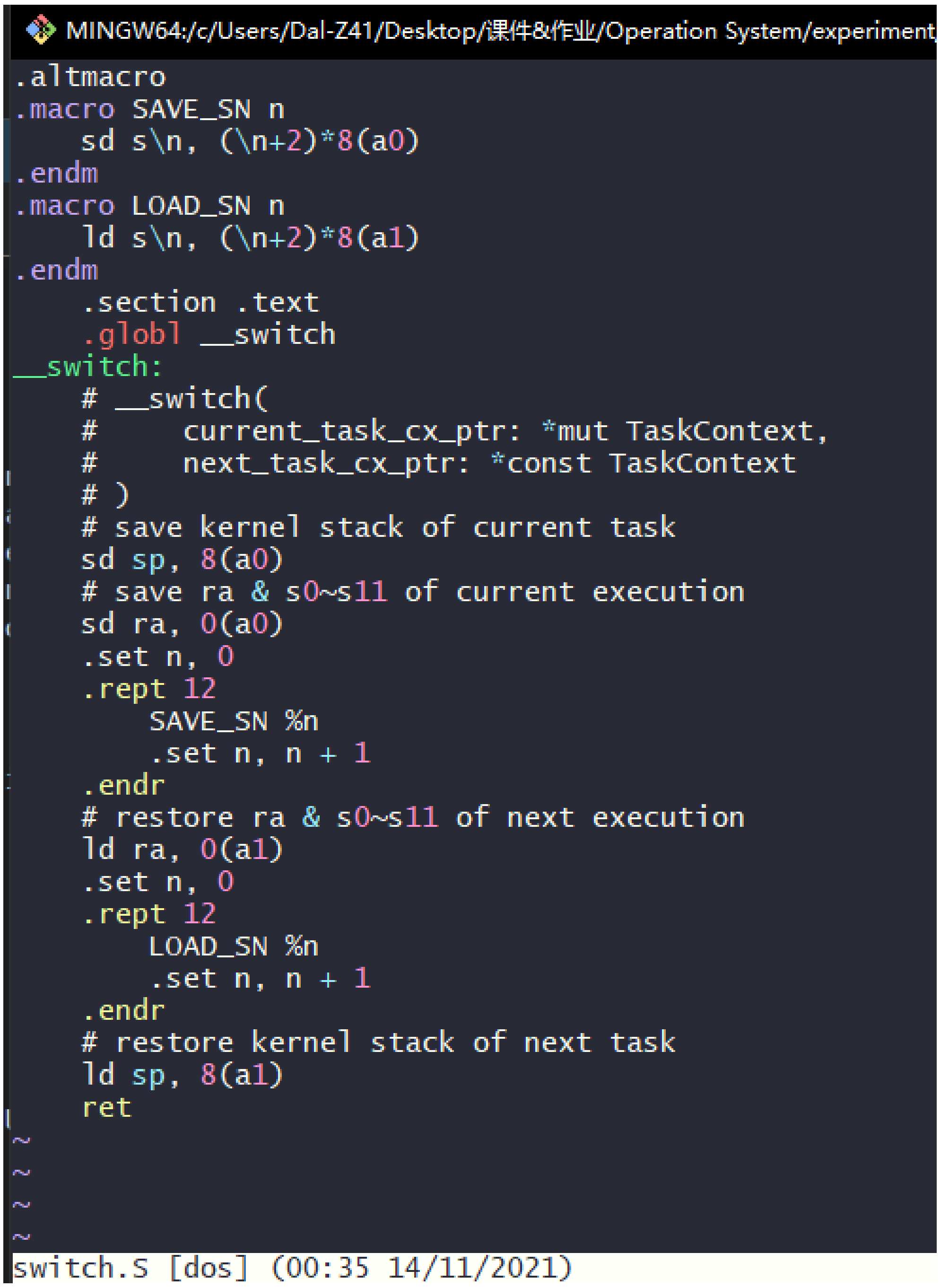
```
use crate::mm::{MemorySet, MapPermission, PhysPageNum, KERNEL_SPACE, Virt
use crate::trap::{TrapContext, trap_handler};
use crate::config::{TRAP_CONTEXT, kernel_stack_position};
use super::TaskContext;

pub struct TaskControlBlock {
    pub task_status: TaskStatus,
    pub task_cx: TaskContext,
    pub memory_set: MemorySet,
    pub trap_cx_ppn: PhysPageNum,
    pub base_size: usize,
}

impl TaskControlBlock {
    pub fn get_trap_cx(&self) -> &'static mut TrapContext {
        self.trap_cx_ppn.get_mut()
    }
    pub fn get_user_token(&self) -> usize {
        self.memory_set.token()
    }
    pub fn new(elf_data: &[u8], app_id: usize) -> Self {
        // memory_set with elf program headers/trampoline/trap context/us
        let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_
        let trap_cx_ppn = memory_set
            .translate(VirtAddr::from(TRAP_CONTEXT).into())
            .unwrap()
            .ppn();
        let task_status = TaskStatus::Ready;
        // map a kernel-stack in kernel space
        let (kernel_stack_bottom, kernel_stack_top) = kernel_stack_positi
        KERNEL_SPACE
            .exclusive_access()
            .insert_framed_area(
                kernel_stack_bottom.into(),
                kernel_stack_top.into(),
                MapPermission::R | MapPermission::W,
            );
task.rs [dos] (00:35 14/11/2021)
```

2. At kernel initialization time, all applications need to be loaded into the global application manager. Also, modify the **TaskManager** implementation. Modify "**switch.S**", and at the same time, modify "**switch.rs**".



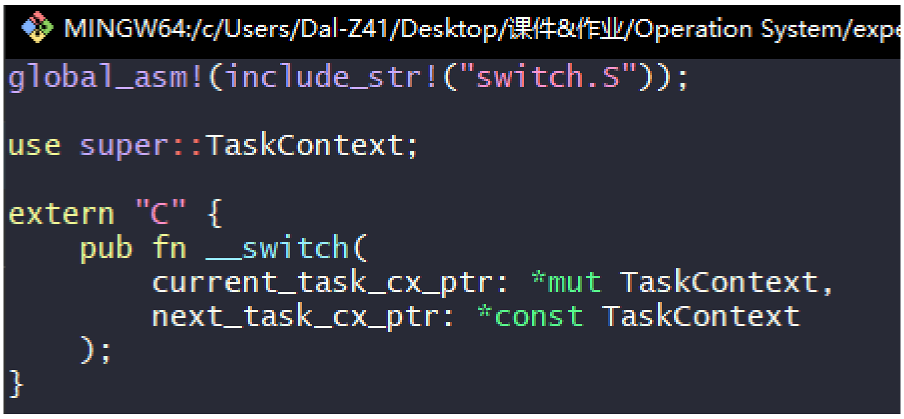
```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/
```

```rust
mod context;
mod switch;
mod task;

use crate::loader::{get_num_app, get_app_data};
use crate::trap::TrapContext;
use crate::sync::UPSafeCell;
use lazy_static::*;
use switch::__switch;
use task::{TaskControlBlock, TaskStatus};
use alloc::vec::Vec;

pub use context::TaskContext;

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

struct TaskManagerInner {
    tasks: Vec<TaskControlBlock>,
    current_task: usize,
}

lazy_static! {
    pub static ref TASK_MANAGER: TaskManager = {
        println!("init TASK_MANAGER");
        let num_app = get_num_app();
        println!("num_app = {}", num_app);
        let mut tasks: Vec<TaskControlBlock> = Vec::new();
        for i in 0..num_app {
            tasks.push(TaskControlBlock::new(
                get_app_data(i),
                i,
            ));
        }
        TaskManager {
mod.rs [dos] (00:35 14/11/2021)
```



MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment

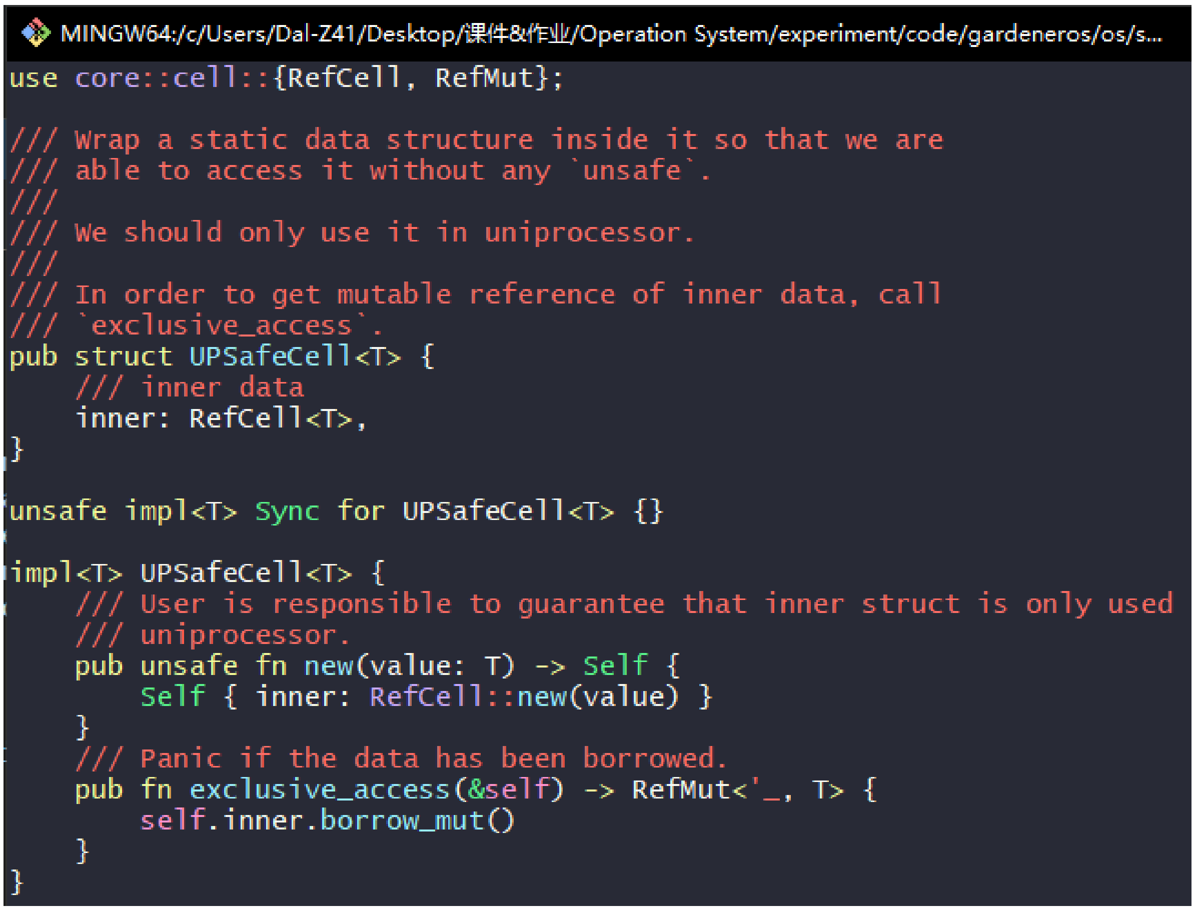
```
.altmacro
.macro SAVE_SN n
    sd s\n, (\n+2)*8(a0)
.endm
.macro LOAD_SN n
    ld s\n, (\n+2)*8(a1)
.endm
    .section .text
    .globl __switch
__switch:
    # __switch(
    #     current_task_cx_ptr: *mut TaskContext,
    #     next_task_cx_ptr: *const TaskContext
    # )
    # save kernel stack of current task
    sd sp, 8(a0)
    # save ra & s0~s11 of current execution
    sd ra, 0(a0)
    .set n, 0
    .rept 12
        SAVE_SN %n
        .set n, n + 1
    .endr
    # restore ra & s0~s11 of next execution
    ld ra, 0(a1)
    .set n, 0
    .rept 12
        LOAD_SN %n
        .set n, n + 1
    .endr
    # restore kernel stack of next task
    ld sp, 8(a1)
    ret
~
~
~
~
switch.S [dos] (00:35 14/11/2021)
```



MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/expe

```
global_asm!(include_str!("switch.S"));

use super::TaskContext;

extern "C" {
    pub fn __switch(
        current_task_cx_ptr: *mut TaskContext,
        next_task_cx_ptr: *const TaskContext
    );
}
```

3. In addition, you need to add the sync module to implement **UPSafeCell**.


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/s...

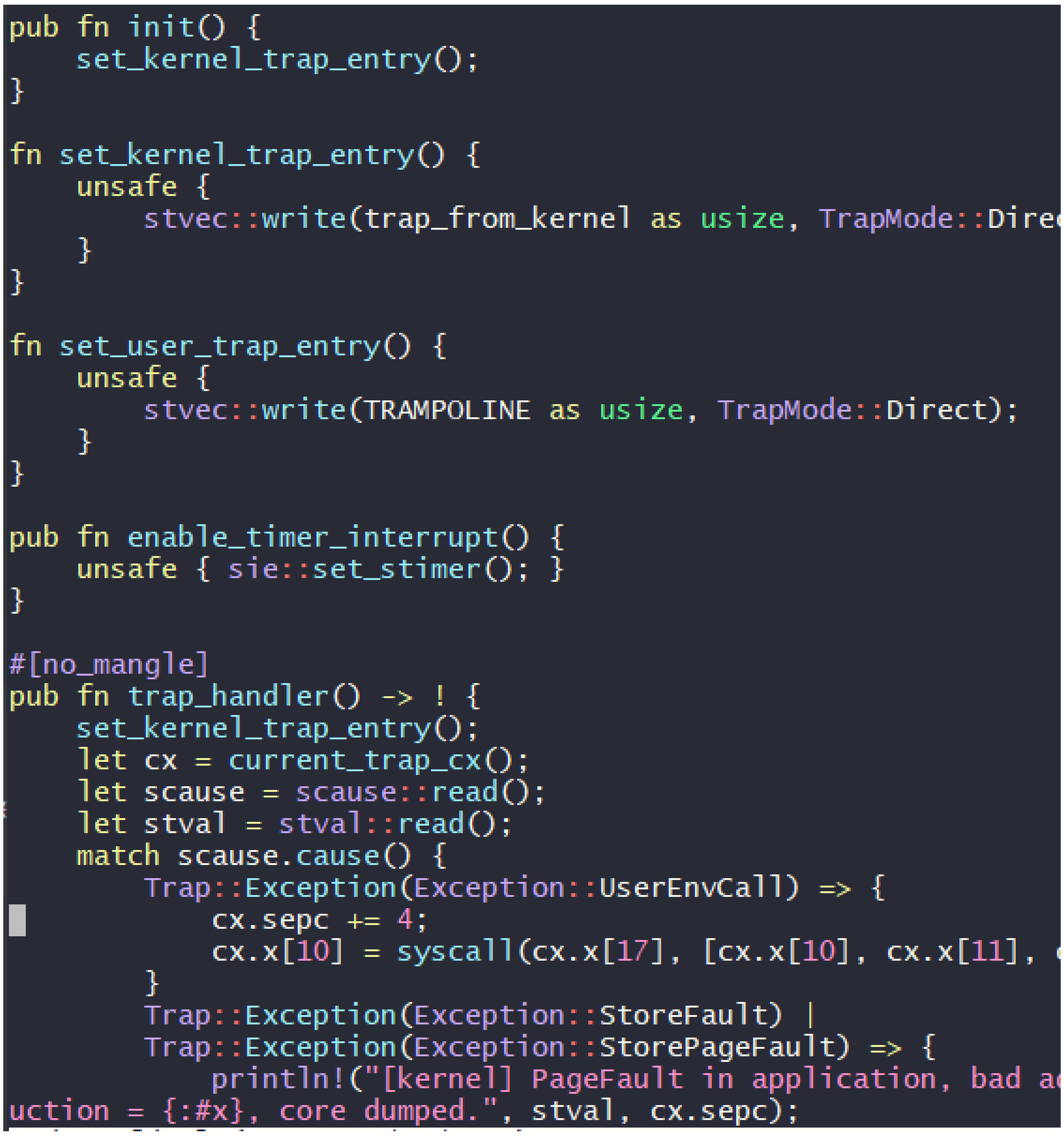
```rust
use core::cell::{RefCell, RefMut};

/// Wrap a static data structure inside it so that we are
/// able to access it without any `unsafe`.
///
/// We should only use it in uniprocessor.
///
/// In order to get mutable reference of inner data, call
/// `exclusive_access`.
pub struct UPSafeCell<T> {
    /// inner data
    inner: RefCell<T>,
}

unsafe impl<T> Sync for UPSafeCell<T> {}

impl<T> UPSafeCell<T> {
    /// User is responsible to guarantee that inner struct is only used
    /// uniprocessor.
    pub unsafe fn new(value: T) -> Self {
        Self { inner: RefCell::new(value) }
    }
    /// Panic if the data has been borrowed.
    pub fn exclusive_access(&self) -> RefMut<'_, T> {
        self.inner.borrow_mut()
    }
}
```

## 7.4 Trap handling

1. First modify the "**init**" function. Then add "**set_kernel_trap_entry**" to the beginning of trap_handler. At the same time, after the trap is processed, call "**trap_return**" to return the user state.

```rust
//os/src/trap/mod.rs
use crate::task::{
    current_user_token,
    current_trap_cx,
};

use crate::config::{TRAP_CONTEXT, TRAMPOLINE};

pub fn init() {
    set_kernel_trap_entry();
}

fn set_kernel_trap_entry() {
    unsafe {
        stvec::write(trap_from_kernel as usize, TrapMode::Direct);
    }
}

fn set_user_trap_entry() {
    unsafe {
        stvec::write(TRAMPOLINE as usize, TrapMode::Direct);
    }
}
```

```
#[no_mangle]
pub fn trap_handler() -> ! {
    set_kernel_trap_entry();

    ...
}

#[no_mangle]
pub fn trap_return() -> ! {
    set_user_trap_entry();

    ...
}

#[no_mangle]
pub fn trap_from_kernel() -> ! {
    panic!("a trap from kernel!");
```

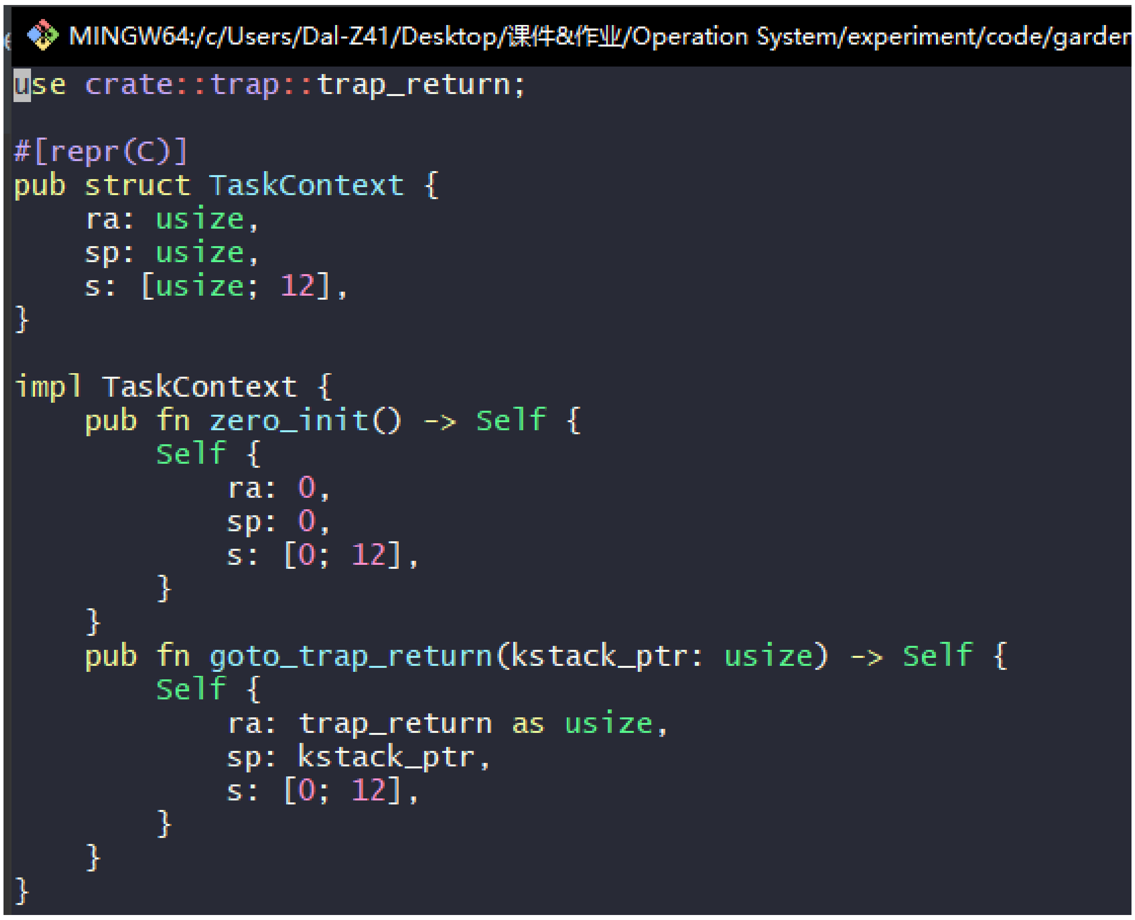


```
pub fn init() {
    set_kernel_trap_entry();
}

fn set_kernel_trap_entry() {
    unsafe {
        stvec::write(trap_from_kernel as usize, TrapMode::Dire
    }
}

fn set_user_trap_entry() {
    unsafe {
        stvec::write(TRAMPOLINE as usize, TrapMode::Direct);
    }
}

pub fn enable_timer_interrupt() {
    unsafe { sie::set_stimer(); }
}

#[no_mangle]
pub fn trap_handler() -> ! {
    set_kernel_trap_entry();
    let cx = current_trap_cx();
    let scause = scause::read();
    let stval = stval::read();
    match scause.cause() {
        Trap::Exception(Exception::UserEnvCall) => {
            cx.sepc += 4;
            cx.x[10] = syscall(cx.x[17], [cx.x[10], cx.x[11],
        }
        Trap::Exception(Exception::StoreFault) |
        Trap::Exception(Exception::StorePageFault) => {
            println!("[kernel] PageFault in application, bad a
uction = {:#x}, core dumped.", stval, cx.sepc);
```

2. At the same time, when each application first obtains CPU privileges, the top of the kernel stack is placed in a task context constructed when the kernel loads the application.

```
//os/src/task/context.rs
```

```
use crate::trap::trap_return;

#[repr(C)]
pub struct TaskContext {
    ra: usize,
    sp: usize,
    s: [usize; 12],
}

impl TaskContext {
    pub fn zero_init() -> Self {
        Self {
            ra: 0,
            sp: 0,
            s: [0; 12],
        }
    }
    pub fn goto_trap_return(kstack_ptr: usize) -> Self {
        Self {
            ra: trap_return as usize,
            sp: kstack_ptr,
            s: [0; 12],
        }
    }
}
```



```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/garden
use crate::trap::trap_return;

#[repr(C)]
pub struct TaskContext {
    ra: usize,
    sp: usize,
    s: [usize; 12],
}

impl TaskContext {
    pub fn zero_init() -> Self {
        Self {
            ra: 0,
            sp: 0,
            s: [0; 12],
        }
    }
    pub fn goto_trap_return(kstack_ptr: usize) -> Self {
        Self {
            ra: trap_return as usize,
            sp: kstack_ptr,
            s: [0; 12],
        }
    }
}
```

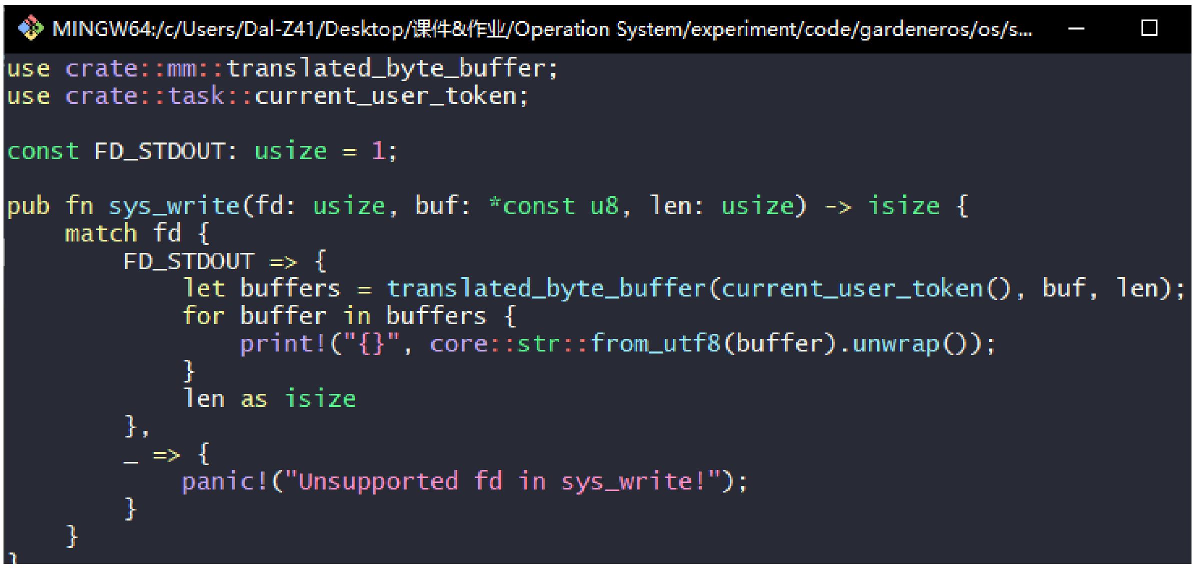
## 7.5 sys_write

"**sys_write**" cannot directly apply space data due to address space isolation. To this end, the page table **"page_table"** provides an auxiliary function that converts the buffer of the application address space into something directly accessible from the kernel address space. Thus, we can modify the **"sys_write"** system call.

```
//os/src/mm/page_table.rs
pub fn translated_byte_buffer(token: usize, ptr: *const u8, len: usize) ->
Vec<&'static mut [u8]> {
    let page_table = PageTable::from_token(token);
    let mut start = ptr as usize;
    let end = start + len;
    let mut v = Vec::new();
    while start < end {
        let start_va = VirtAddr::from(start);
        let mut vpn = start_va.floor();
        let ppn = page_table
            .translate(vpn)
            .unwrap()
            .ppn();
        vpn.step();
        let mut end_va: VirtAddr = vpn.into();
        end_va = end_va.min(VirtAddr::from(end));
        if end_va.page_offset() == 0 {
            v.push(&mut ppn.get_bytes_array()[start_va.page_offset()..]);
        } else {
            v.push(&mut ppn.get_bytes_array()
[start_va.page_offset()..end_va.page_offset()]);
        }
        start = end_va.into();
    }
    v
}
```

```
pub fn translated_byte_buffer(token: usize, ptr: *const u8, len: usize) -> Vec<&'sta
tic mut [u8]> {
    let page_table = PageTable::from_token(token);
    let mut start = ptr as usize;
    let end = start + len;
    let mut v = Vec::new();
    while start < end {
        let start_va = VirtAddr::from(start);
        let mut vpn = start_va.floor();
        let ppn = page_table
            .translate(vpn)
            .unwrap()
            .ppn();
        vpn.step();
        let mut end_va: VirtAddr = vpn.into();
        end_va = end_va.min(VirtAddr::from(end));
        if end_va.page_offset() == 0 {
            v.push(&mut ppn.get_bytes_array()[start_va.page_offset()..]);
        } else {
            v.push(&mut ppn.get_bytes_array()[start_va.page_offset()..end_va.page_of
fset()]);
        }
        start = end_va.into();
    }
    v
}
page_table.rs [dos] (00:35 14/11/2021)                                     157,1 Bo
```

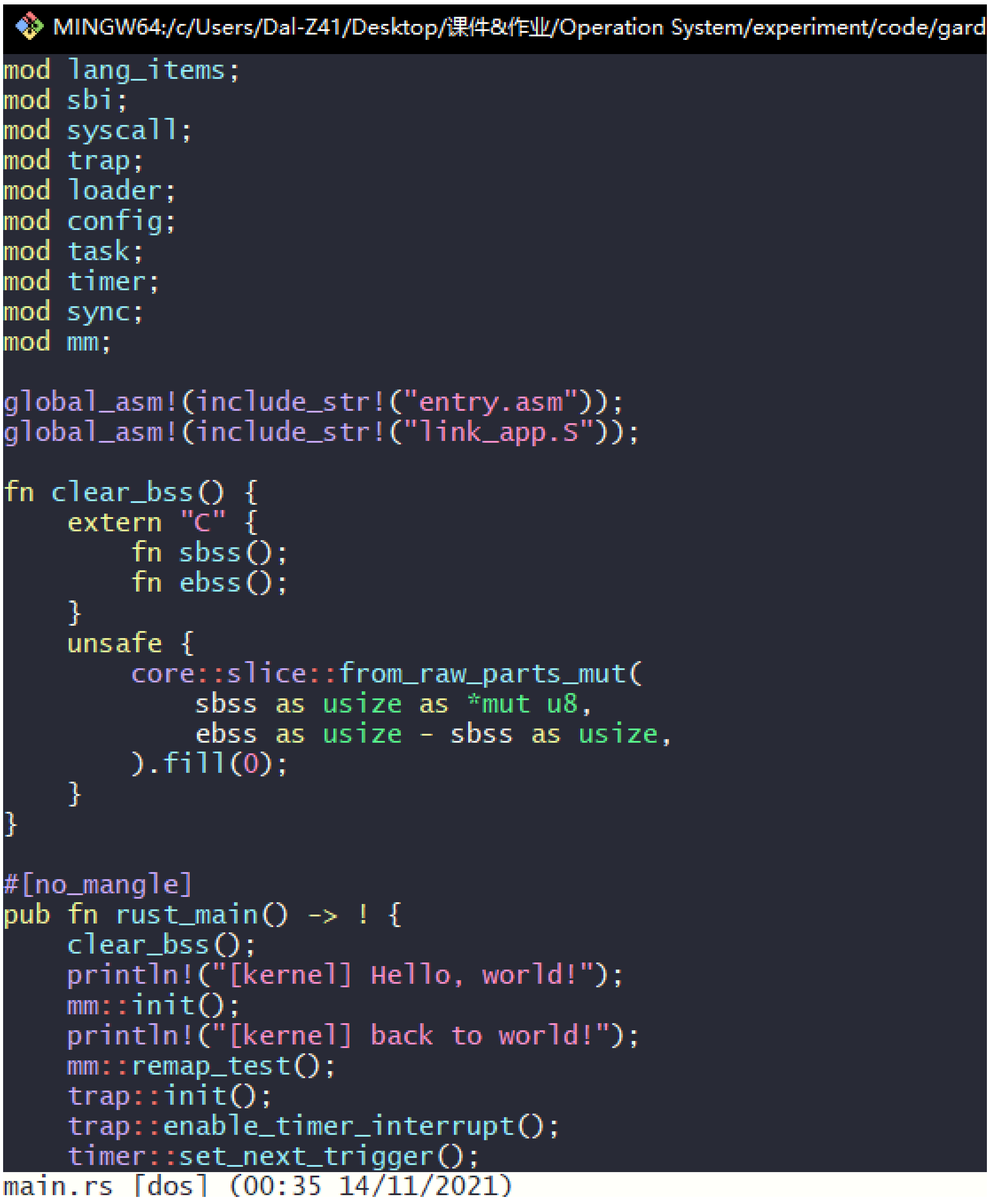
```
//os/src/syscall/fs.rs
use crate::mm::translated_byte_buffer;
use crate::task::current_user_token;

const FD_STDOUT: usize = 1;

pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
    match fd {
        FD_STDOUT => {
            let buffers = translated_byte_buffer(current_user_token(), buf,
len);
            for buffer in buffers {
                print!("{}", core::str::from_utf8(buffer).unwrap());
            }
            len as isize
        },
        _ => {
            panic!("Unsupported fd in sys_write!");
        }
    }
}
```


MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gardeneros/os/s...

```
use crate::mm::translated_byte_buffer;
use crate::task::current_user_token;

const FD_STDOUT: usize = 1;

pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
    match fd {
        FD_STDOUT => {
            let buffers = translated_byte_buffer(current_user_token(), buf, len);
            for buffer in buffers {
                print!("{}", core::str::from_utf8(buffer).unwrap());
            }
            len as isize
        },
        _ => {
            panic!("Unsupported fd in sys_write!");
        }
    }
}
```

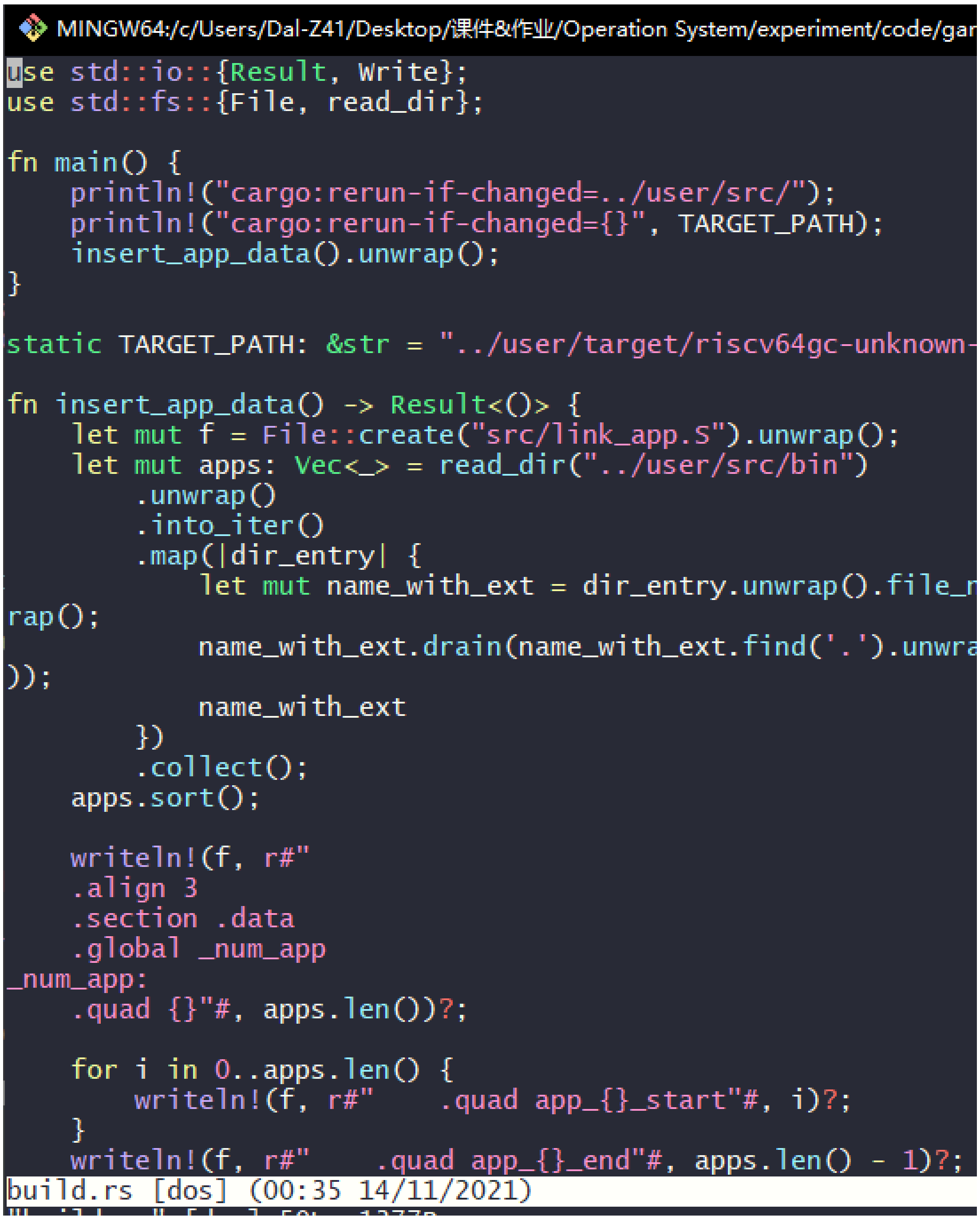
## 8. 完善程序

1. Delete **"build.py"**. Since the starting address of the application is the same, you don't need build.py anymore, just delete it. Also, modify the Makefile file.
2. Modify **"main.rs"**.

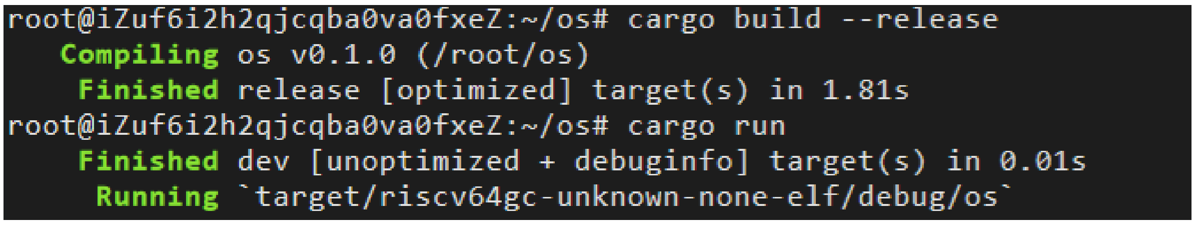

MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gard

```rust
mod lang_items;
mod sbi;
mod syscall;
mod trap;
mod loader;
mod config;
mod task;
mod timer;
mod sync;
mod mm;

global_asm!(include_str!("entry.asm"));
global_asm!(include_str!("link_app.S"));

fn clear_bss() {
    extern "C" {
        fn sbss();
        fn ebss();
    }
    unsafe {
        core::slice::from_raw_parts_mut(
            sbss as usize as *mut u8,
            ebss as usize - sbss as usize,
        ).fill(0);
    }
}

#[no_mangle]
pub fn rust_main() -> ! {
    clear_bss();
    println!("[kernel] Hello, world!");
    mm::init();
    println!("[kernel] back to world!");
    mm::remap_test();
    trap::init();
    trap::enable_timer_interrupt();
    timer::set_next_trigger();
main.rs [dos] (00:35 14/11/2021)
```

3. Modify **"os/build.rs"**.



```
MINGW64:/c/Users/Dal-Z41/Desktop/课件&作业/Operation System/experiment/code/gar
use std::io::{Result, Write};
use std::fs::{File, read_dir};

fn main() {
    println!("cargo:rerun-if-changed=../user/src/");
    println!("cargo:rerun-if-changed={}", TARGET_PATH);
    insert_app_data().unwrap();
}

static TARGET_PATH: &str = "../user/target/riscv64gc-unknown-

fn insert_app_data() -> Result<()> {
    let mut f = File::create("src/link_app.S").unwrap();
    let mut apps: Vec<_> = read_dir("../user/src/bin")
        .unwrap()
        .into_iter()
        .map(|dir_entry| {
            let mut name_with_ext = dir_entry.unwrap().file_n
rap();
            name_with_ext.drain(name_with_ext.find('.').unwra
));
            name_with_ext
        })
        .collect();
    apps.sort();

    writeln!(f, r#"
    .align 3
    .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)?;
build.rs [dos] (00:35 14/11/2021)
```



```
root@iZuf6i2h2qjcqba0va0fxeZ:~/os# cargo build --release
   Compiling os v0.1.0 (/root/os)
    Finished release [optimized] target(s) in 1.81s
root@iZuf6i2h2qjcqba0va0fxeZ:~/os# cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/riscv64gc-unknown-none-elf/debug/os`
```