Writing an Operating System in Rust (rCore)

From a new rust project

- We need to
  - Create a rust project
  - Remove the dependency on the operating system
  - Use the qemu emulator to run the kernel image

cargo new os

```
os
├── Cargo.toml
└── src
    └── main.rs
```

```rust
//main.rs
fn main() {
    println!("Hello, world!");
}
```

Rust std vs core libraty

- Rust standard library(std)
  - By default, all Rust crates link the standard library
  - The standard library depends on the operating system for features such as threads, files

- Rust core library(core)
  - The Rust Core Library is the dependency-free foundation of The Rust Standard Library.
  - The core library is minimal and platform-agnostic

Remove the dependency on the operating system

SUSTech Southern University of Science and Technology

std
↓
#![no_std]
↓
core

Handle these errors:

```
error: cannot find macro `println` in this scope
 --> src/main.rs:3:5
  |
3 |     println!("Hello, world!");
  |     ^^^^^^^

error: language item required, but not found: `eh_personality`

error: `#[panic_handler]` function required, but not found
```

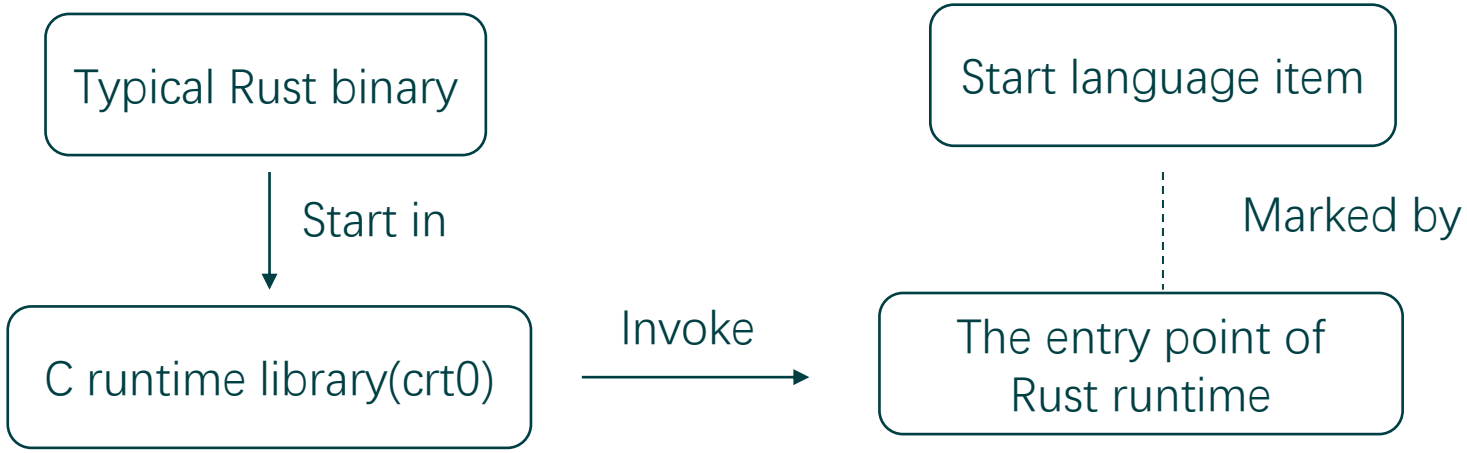
- Remove the println macro
- When panic happens, abort!
- Define our panic handle function

```rust
use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

Remove the dependency on the operating system

SUSTech Southern University of Science and Technology

```
error: requires `start` lang_item
```



- Overwrite the crt0 entry point:
  - Remove the main function
  - #![no_miain] : don't use the normal entry point
  - Add _start() function

```rust
#![no_main]

#[no_mangle]
pub extern "C" fn _start() -> ! {
    loop {}
}
```

Build for a Bare Metal Target

linker error:

```
error: linking with `cc` failed: exit status: 1
```

Cause: the default configuration of the linker assumes that our program depends on the C runtime, which it does not.

Solution: building for a bare metal target

Rust uses a string called target triple to describe different environments.

- x86_64-unknown-linux-gnu
- riscv64imac-unknown-none-elf (bare metal)

cargo build --target riscv64imac-unknown-none-elf

Build our kernel

cargo build --target riscv64imac-unknown-none-elf

→ target/riscv64imac-unknown-none-elf/debug/os

File type: ELF 64-bit LSB executable, UCB RISC-V

Use command objdump to view the imformation of the file

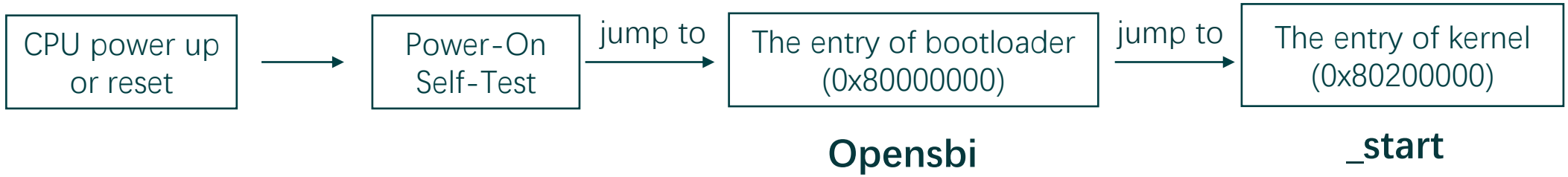
```
$ rust-objdump target/riscv64imac-unknown-none-elf/debug/os -x --arch-name=riscv64

target/riscv64imac-unknown-none-elf/debug/os:    file format ELF64-riscv

architecture: riscv64
start address: 0x0000000000011000

Sections:
Idx Name          Size     VMA          Type
  0               00000000 0000000000000000
  1 .text         0000000c 0000000000011000 TEXT
```

Build our kernel

Use command:
rust-objcopy target/riscv64imac-unknown-none-elf/debug/os --strip-all -O binary target/riscv64imac-unknown-none-elf/debug/kernel.bin

elf executable => binary file (kernel image)

Booting:

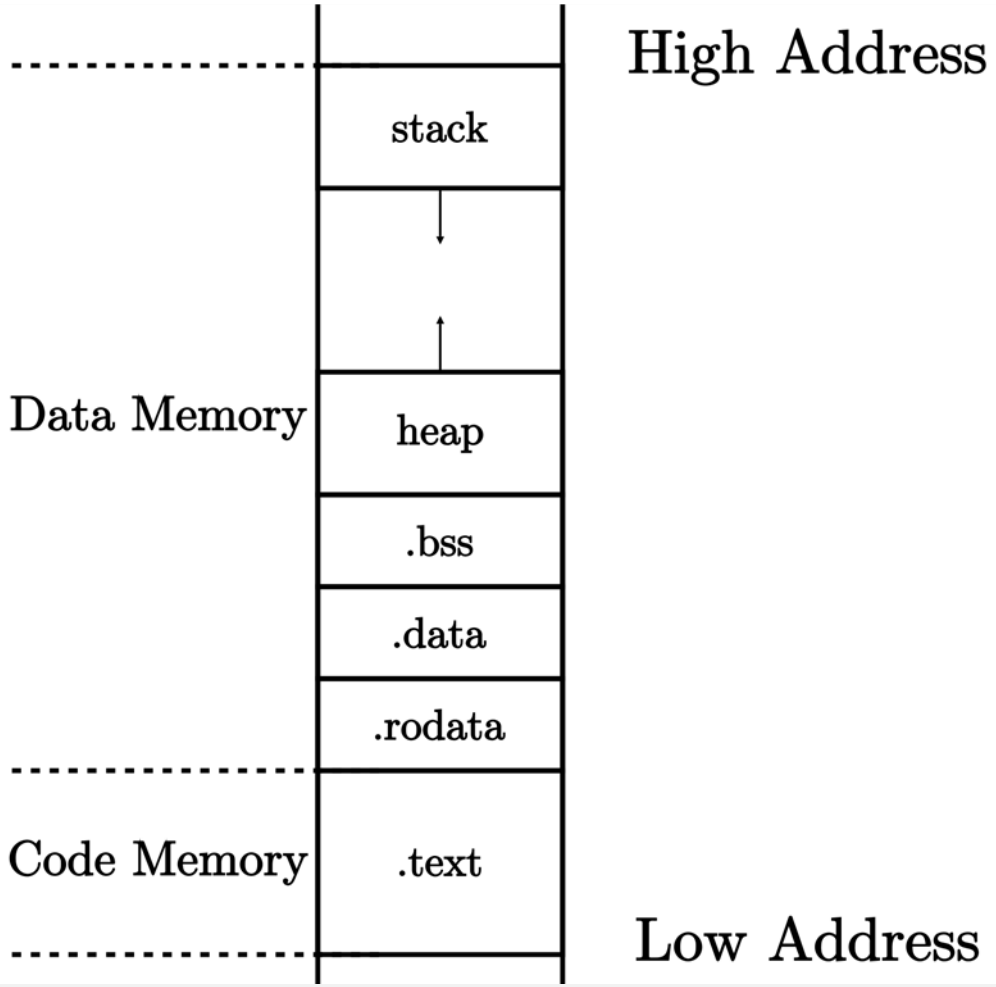


If we want to run kernel on qemu, we need to:

- adjust the memory layout
- rewrite the entry point

Run kernel on qemu

Adjust the memory layout: linker.ld




```
OUTPUT_ARCH(riscv)

ENTRY(_start)

BASE_ADDRESS = 0x80200000;

SECTIONS
{
    . = BASE_ADDRESS;

    kernel_start = .;

    text_start = .;

    .text : {
        *(.text.entry)
        *(.text .text.*)
    }
```

```
rodata_start = .;

    .rodata : {
        *(.rodata .rodata.*)
    }

    data_start = .;

    .data : {
        *(.data .data.*)
    }

    bss_start = .;

    .bss : {
        *(.sbss .bss .bss.*)
    }

    kernel_end = .;
}
```

Run kernel on qemu

SUSTech Southern University of Science and Technology

We want to rewrite the entry point to set the operating environment of the kernel

```
# os/src/entry.asm
    .section .text.entry
    .globl _start
_start:
    la sp, boot_stack_top
    call rust_main

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096 * 16
    .globl boot_stack_top
boot_stack_top:
```

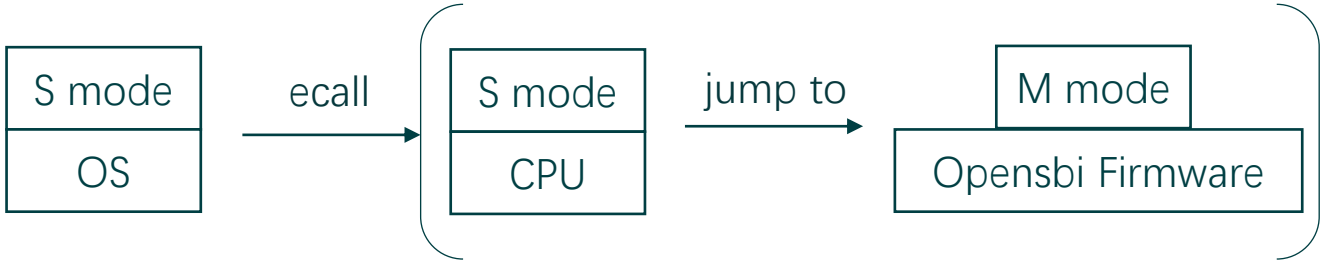
```
//main.rs
#![feature(global_asm)]

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

#[no_mangle]
pub extern "C" fn rust_main() -> ! {
    loop {}
}
```

```
$ qemu-system-riscv64 --machine virt \
  --nographic --bios default
```

Print something in our screen

SUSTech Southern University of Science and Technology

Opensbi provides some interfaces in C function format.




```rust
/// SBI 调用
#[inline(always)]
fn sbi_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {
    let ret;
    unsafe {
        llvm_asm!("ecall"
            : "={x10}" (ret)
            : "{x10}" (arg0), "{x11}" (arg1), "{x12}" (arg2), "{x17}" (which)
            : "memory"      // 如果汇编可能改变内存，则需要加入 memory 选项
            : "volatile");  // 防止编译器做激进的优化
    }
    ret
}
```

```rust
const SBI_CONSOLE_PUTCHAR: usize = 1;
const SBI_CONSOLE_GETCHAR: usize = 2;
const SBI_SHUTDOWN: usize = 8;

pub fn console_putchar(c: usize) {
    sbi_call(SBI_CONSOLE_PUTCHAR, c, 0, 0);
}

pub fn console_getchar() -> usize {
    sbi_call(SBI_CONSOLE_GETCHAR, 0, 0, 0)
}

pub fn shutdown() -> ! {
    sbi_call(SBI_SHUTDOWN, 0, 0, 0);
    unreachable!()
}
```

Implemment formatted output

SUSTech Southern University of Science and Technology

```rust
use crate::sbi::*;
use core::fmt::{self, Write};

struct Stdout;

impl Write for Stdout {

    fn write_str(&mut self, s: &str) -> fmt::Result {
        let mut buffer = [0u8; 4];
        for c in s.chars() {
            for code_point in c.encode_utf8(&mut buffer).as_bytes().iter() {
                console_putchar(*code_point as usize);
            }
  Ok(())
    }
}

pub fn print(args: fmt::Arguments) {
    Stdout.write_fmt(args).unwrap();
}
```

```rust
#[macro_export]
macro_rules! print {
    ($fmt: literal $(, $($arg: tt)+)?) => {
        $crate::console::print(format_args!($fmt $(, $($arg)+)?));
    }
}

#[macro_export]
macro_rules! println {
    ($fmt: literal $(, $($arg: tt)+)?) => {
        $crate::console::print(format_args!(concat!($fmt, "\n") $(, $($arg)+)?));
    }
}
```

```
let a = “world”.to_string();
println!(“hello”);
println!(“hello {}”, a);
```

RISC-V Interrupt

SUSTech Southern University of Science and Technology

Control and Status Registers(CSR)

- Registers automatically filled by hardware
  - sepc
  - scause
  - stval
- Registers that guide the hardware to handle interrupts
  - stvec {base,mode}
  - sstatus
  - sie
  - sip
  - sscratch

Interrupt instruction

- ecall
- sret
- ebreak
- mret
- csrrw dst, csr, src
- csrr dst, csr
- csrw csr, src
- csrc(i) csr, rs1
- csrs(i) csr, rs1

Context:

```
use riscv::register::{sstatus::Sstatus, scause::Scause};

#[repr(C)]
#[derive(Debug)]
pub struct Context {
    pub x: [usize; 32],
    pub sstatus: Sstatus,
    pub sepc: usize
}
```

Interrupt processing

SUSTech Southern University of Science and Technology

When an interrupt happens,we need to:

- Save context in the stack(interrupt.asm)
- Jump to the interrupt_handle function(handle.rs)
- Restore the context(interrupt.asm)

Process the break_point interrupt:

```rust
use super::context::Context;
use riscv::register::stvec;

global_asm!(include_str!("./interrupt.asm"));

pub fn init() {
    unsafe {
        extern "C" {
            fn __interrupt();
      }
        stvec::write(__interrupt as usize, stvec::TrapMode::Direct);
    }
}
```

```rust
#[no_mangle]
pub fn handle_interrupt(context: &mut Context, scause: Scause, stval: usize) {
    match scause.cause() {
        Trap::Exception(Exception::Breakpoint) => breakpoint(context),
        _ => fault(context, scause, stval),
    }
}
```

```rust
fn breakpoint(context: &mut Context) {
    println!("Breakpoint at 0x{:x}", context.sepc);
    context.sepc += 2;
}
```

interrupt.asm

```
.altmacro
.set    REG_SIZE, 8
.set    CONTEXT_SIZE, 34


.macro SAVE reg, offset
    sd  \reg, \offset*8(sp)
.endm

.macro SAVE_N n
    SAVE  x\n, \n
.endm


.macro LOAD reg, offset
    ld  \reg, \offset*8(sp)
.endm

.macro LOAD_N n
    LOAD  x\n, \n
.endm
```

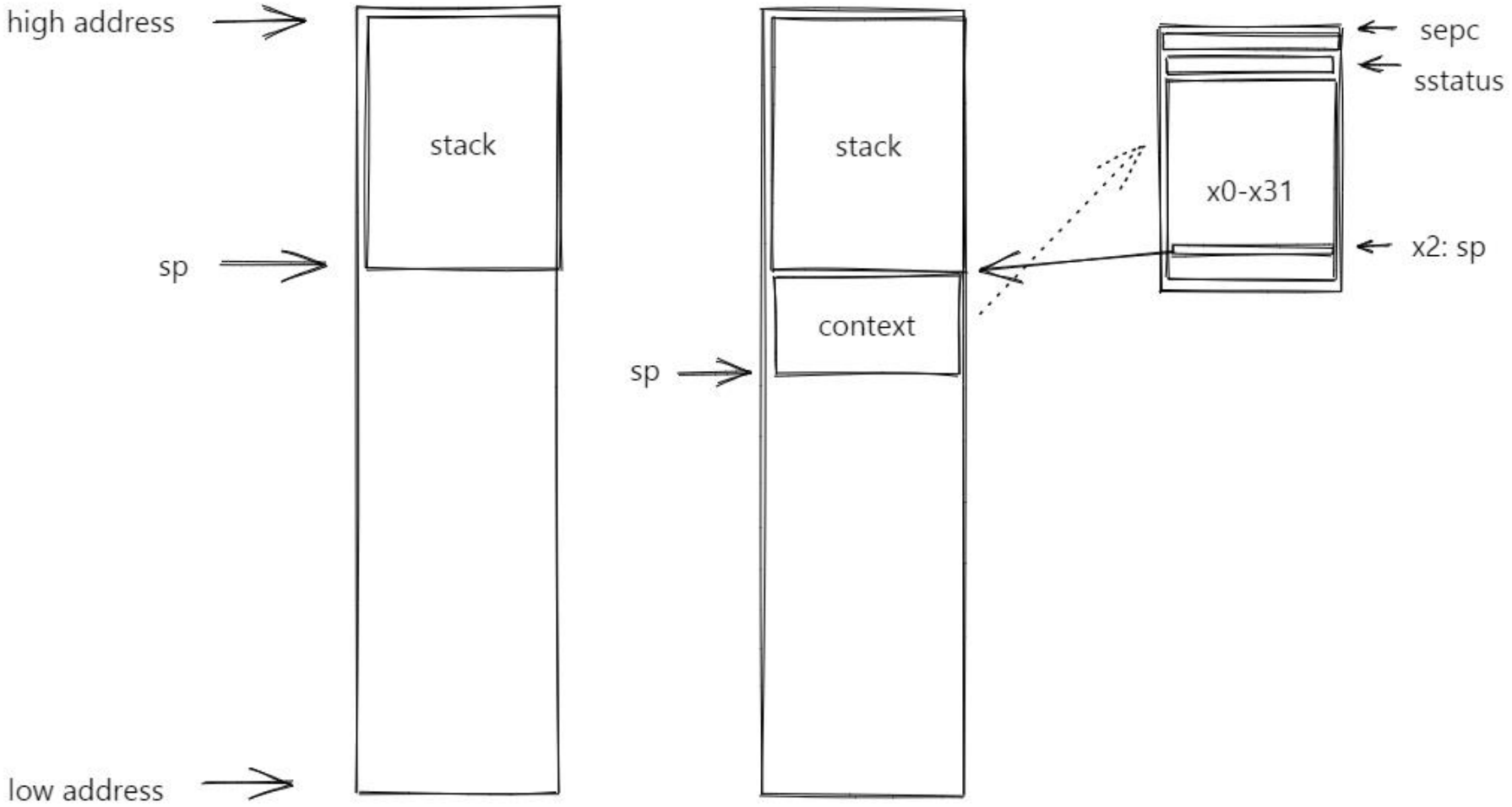
```
.section .text
    .globl __interrupt

__interrupt:
    addi    sp, sp, -34*8

    SAVE    x1, 1
    addi    x1, sp, 34*8
    SAVE    x1, 2
    .set    n, 3
    .rept   29
        SAVE_N  %n
        .set    n, n + 1
    .endr

    csrr    s1, sstatus
    csrr    s2, sepc
    SAVE    s1, 32
    SAVE    s2, 33

    # context: &mut Context
    mv      a0, sp
    # scause: Scause
    csrr    a1, scause
    # stval: usize
    csrr    a2, stval
    jal  handle_interrupt
```

```
.globl __restore
__restore:
    LOAD    s1, 32
    LOAD    s2, 33
    csrw    sstatus, s1
    csrw    sepc, s2
    LOAD    x1, 1
    .set    n, 3
    .rept   29
        LOAD_N  %n
        .set    n, n + 1
    .endr

    LOAD    x2, 2
    sret
```

interrupt.asm

```asm
.section .text
    .globl __interrupt

__interrupt:
    addi    sp, sp, -34*8

    SAVE    x1, 1
    addi    x1, sp, 34*8
    SAVE    x1, 2
    .set    n, 3
    .rept   29
        SAVE_N  %n
        .set    n, n + 1
    .endr

    csrr    s1, sstatus
    csrr    s2, sepc
    SAVE    s1, 32
    SAVE    s2, 33

    # context: &mut Context
    mv      a0, sp
    # scause: Scause
    csrr    a1, scause
    # stval: usize
    csrr    a2, stval
    jal  handle_interrupt
```

Save context:



Clock interrupt

- Initial
  - sie::set_timer
  - sstatus::set_sie
  - set next timeout

```
static INTERVAL: usize = 100000;
fn set_next_timeout() {
    set_timer(time::read() + INTERVAL);
}
```

- Handle clock interrupt
  - match cause
  - Trap::Interrupt(Interrupt::SupervisorTimer) => supervisor_timer(context)
  - set next timeout

Dynamic memory allocation

The core library is minimal: it isn't even aware of heap allocation.

In order to implement dynamic memory allocation, we need to implement the Trait GlobalAlloc, and marked by #[global_allocator] lang item

```
unsafe fn alloc(&self, layout: Layout) -> *mut u8;
unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
```

Memory allocation algorithm: Buddy system (https://github.com/rcore-os/buddy_system_allocator)

Heap allocation

```rust
use buddy_system_allocator::LockedHeap;

/// pub const KERNEL_HEAP_SIZE: usize = 0x80_0000;
/// in segment bss
static mut HEAP_SPACE: [u8; KERNEL_HEAP_SIZE] = [0; KERNEL_HEAP_SIZE];


/// [`LockedHeap`] implements [`alloc::alloc::GlobalAlloc`] trait
#[global_allocator]
static HEAP: LockedHeap = LockedHeap::empty();

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

#[alloc_error_handler]
fn alloc_error_handler(_: alloc::alloc::Layout) -> ! {
    panic!("alloc error")
}
```

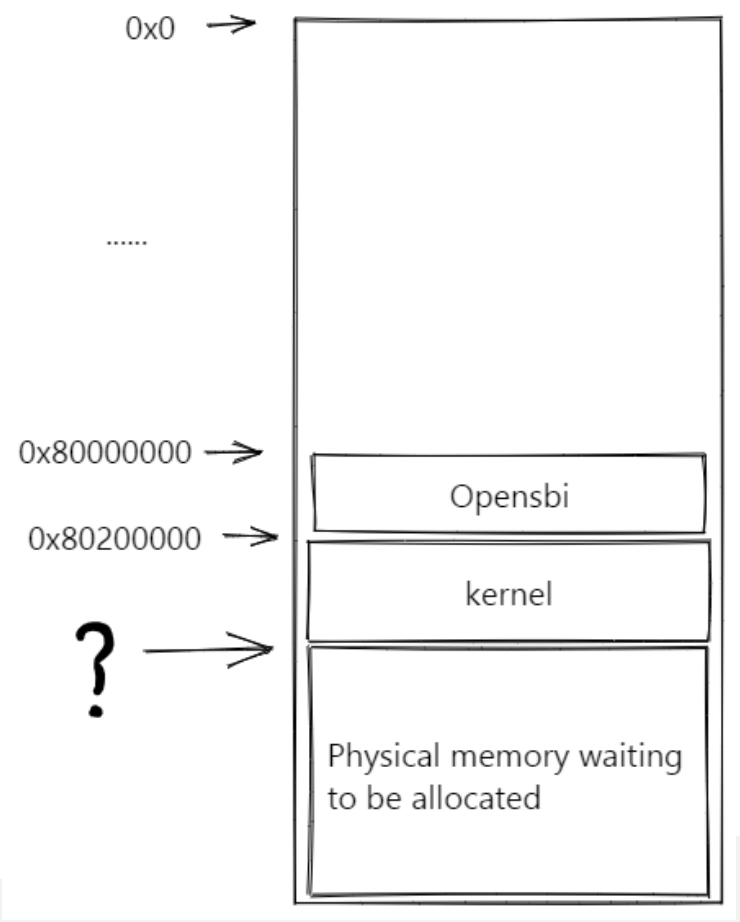
Some tests in rust_main:

```rust
#[no_mangle]
pub extern "C" fn rust_main() -> ! {
    // [......]

    use alloc::boxed::Box;
    use alloc::vec::Vec;
    let v = Box::new(5);
    assert_eq!(*v, 5);
    core::mem::drop(v);

    let mut vec = Vec::new();
    for i in 0..10000 {
        vec.push(i);
    }
    assert_eq!(vec.len(), 10000);
    for (i, value) in vec.into_iter().enumerate() {
        assert_eq!(value, i);
    }
    println!("heap test passed");

    //[......]

}
```

Physical memory

Physical memory in a RISC-V Virt computer simulated by QEMU:

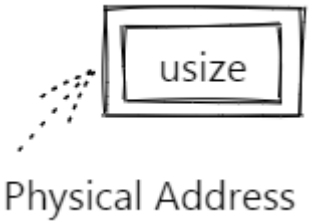
```
static const struct MemmapEntry {
    hwaddr base;
    hwaddr size;
} virt_memmap[] = {
    [VIRT_DEBUG] =        {        0x0,         0x100 },
    [VIRT_MROM] =         {     0x1000,        0xf000 },
    [VIRT_TEST] =         {   0x100000,        0x1000 },
    [VIRT_RTC] =          {   0x101000,        0x1000 },
    [VIRT_CLINT] =        {  0x2000000,       0x10000 },
    [VIRT_PCIE_PIO] =     {  0x3000000,       0x10000 },
    [VIRT_PLIC] =         {  0xc000000,
VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_UART0] =        { 0x10000000,         0x100 },
    [VIRT_VIRTIO] =       { 0x10001000,        0x1000 },
    [VIRT_FLASH] =        { 0x20000000,     0x4000000 },
    [VIRT_PCIE_ECAM] =    { 0x30000000,    0x10000000 },
    [VIRT_PCIE_MMIO] =    { 0x40000000,    0x40000000 },
    [VIRT_DRAM] =         { 0x80000000,           0x0 },
};
```



Physical memory

Encapsulate an usize integer as the physical address:

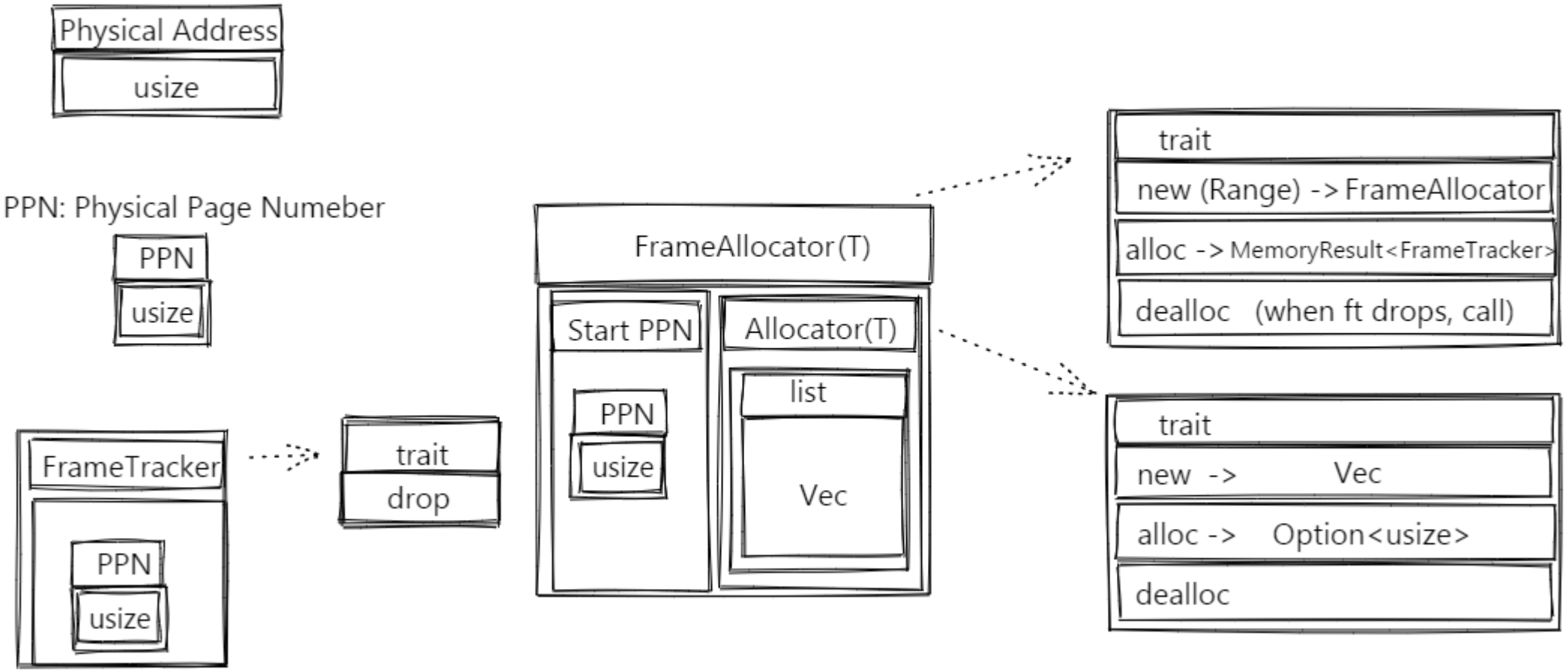
```rust
#[repr(C)]
#[derive(Copy, Clone, Debug, Default, Eq, PartialEq, Ord, PartialOrd, Hash)]
pub struct PhysicalAddress(pub usize); //Physical Address
```



Kernel_end is defined in linker.ld

```rust
lazy_static! {
    pub static ref KERNEL_END_ADDRESS: PhysicalAddress = PhysicalAddress(kernel_end as usize);
}

extern "C" {fn kernel_end();}
```

Physical memory management



Physical memory management

SUSTech Southern University of Science and Technology

Initialization: we need lazy_static and Mutex crate

```
lazy_static! {
    pub static ref FRAME_ALLOCATOR: Mutex<FrameAllocator<AllocatorImpl>> = Mutex::new(FrameAllocator::new(Range::from(
            PhysicalPageNumber::ceil(PhysicalAddress::from(*KERNEL_END_ADDRESS))..PhysicalPageNumber::floor(MEMORY_END_ADDRESS),
        )
    ));
}
```

Allocation:

FrameAllocator alloc → Allocator alloc → FrameTracker

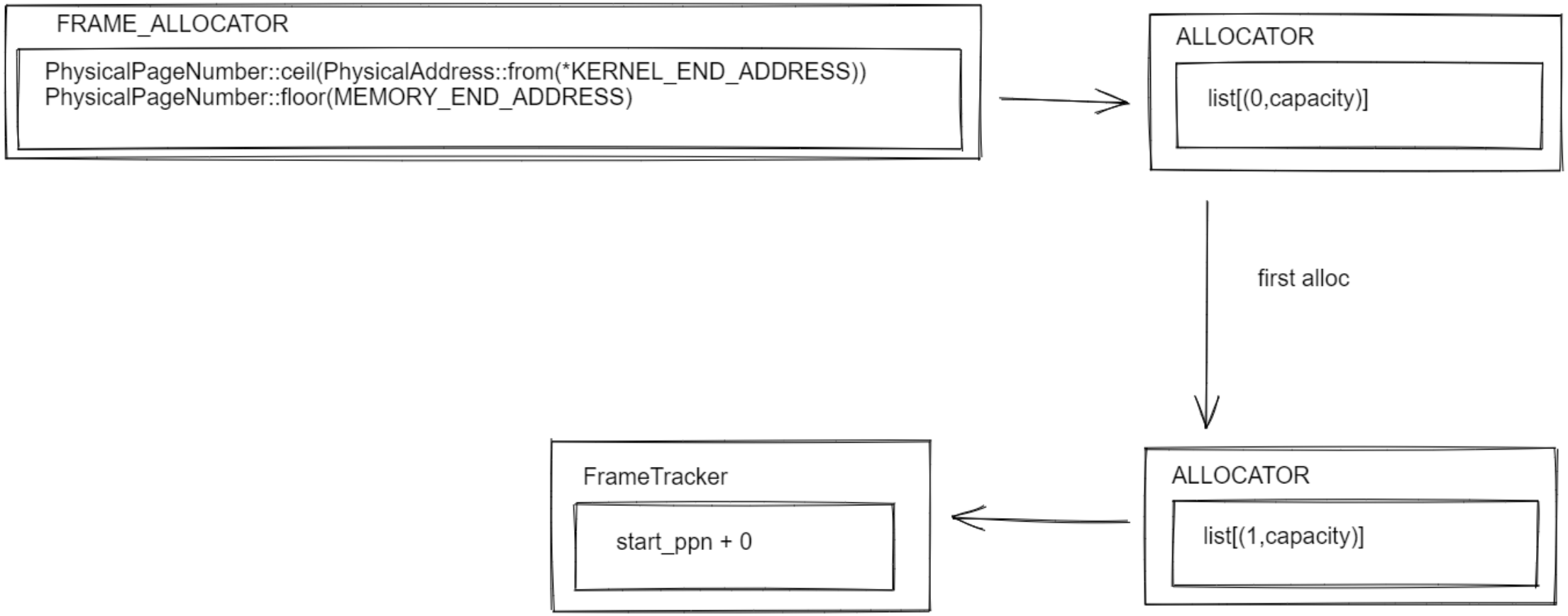
Deallocation：

FrameTracker drop → FrameAllocator dealloc → Allocator dealloc

Allocation

Allocation:

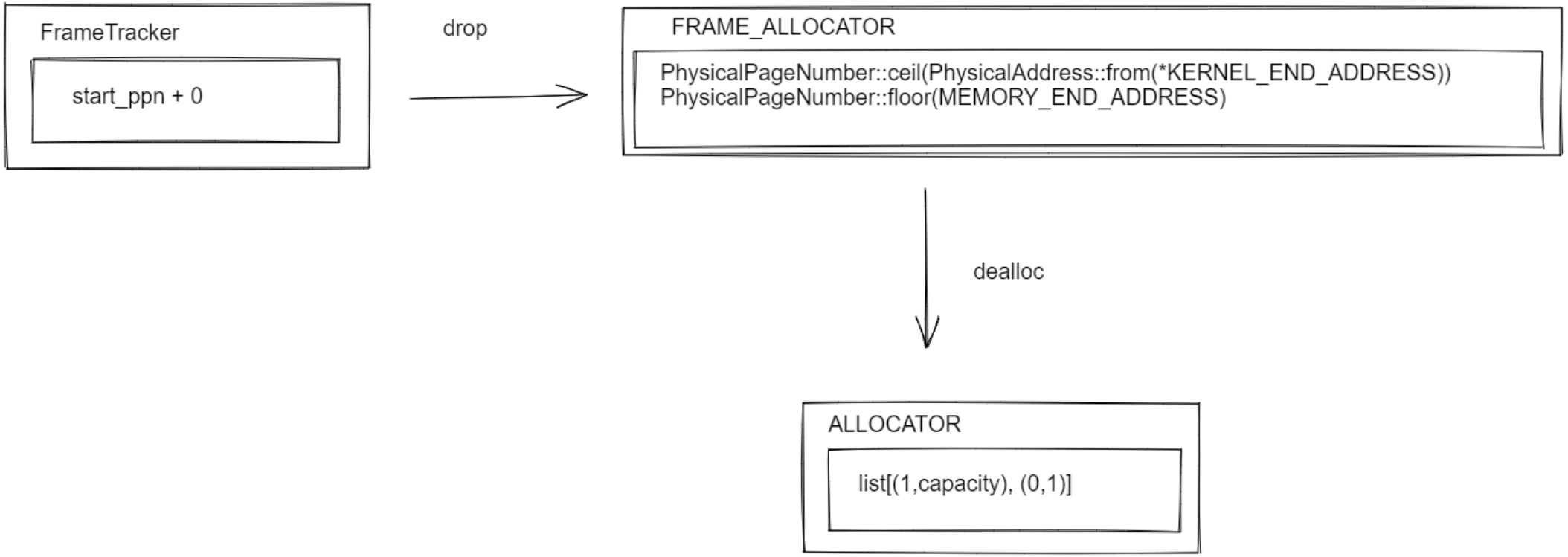
FrameAllocator alloc → Allocator alloc → FrameTracker



Deallocation

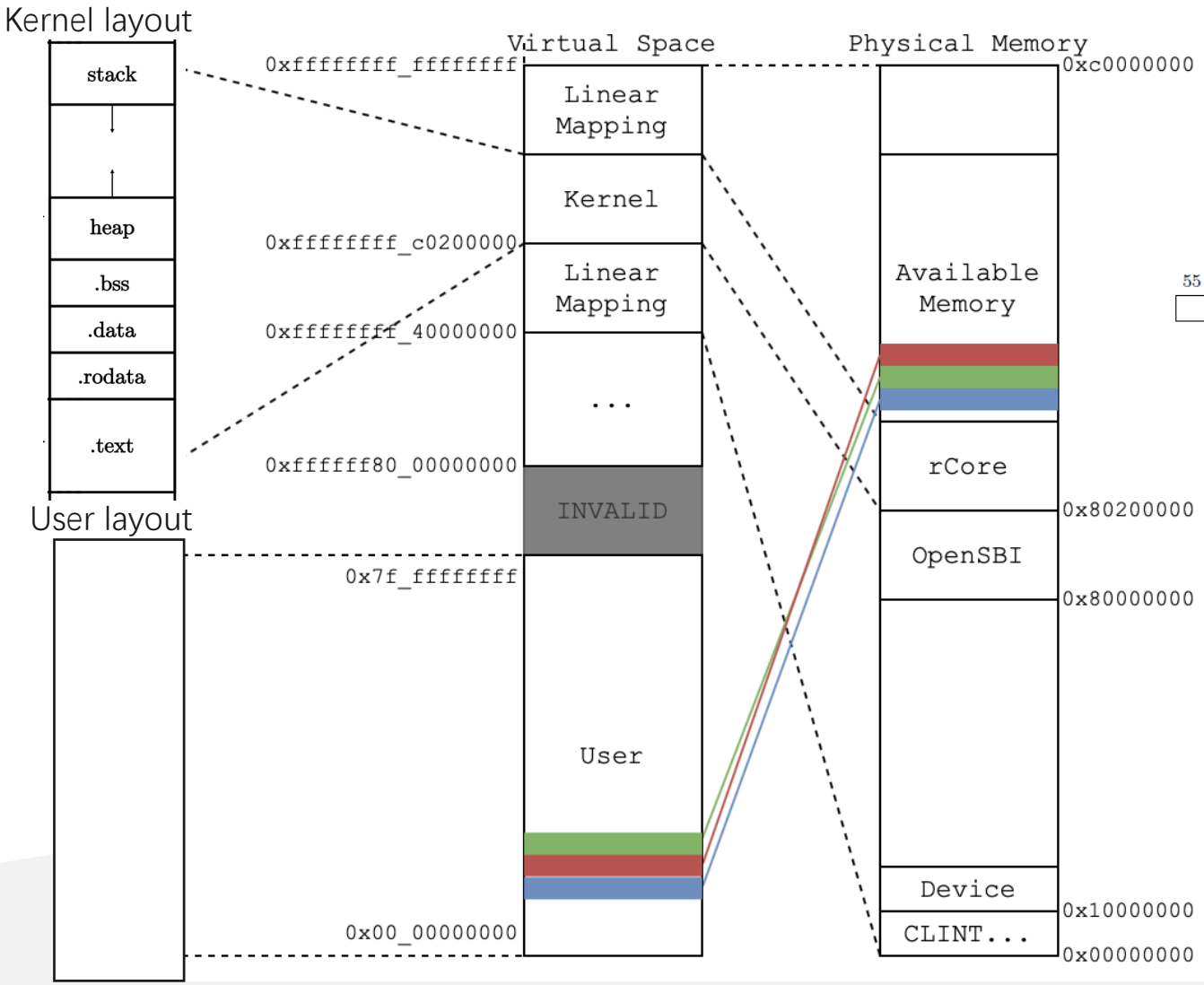
Deallocation：

FrameTracker drop → FrameAllocator dealloc → Allocator dealloc



Physical memory and virtual memory

Sv39 mode (supported by risc-v hardware)



38 … 30	29 … 21	20 … 12	11 … 0
VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	12

Figure 4.16: Sv39 virtual address.

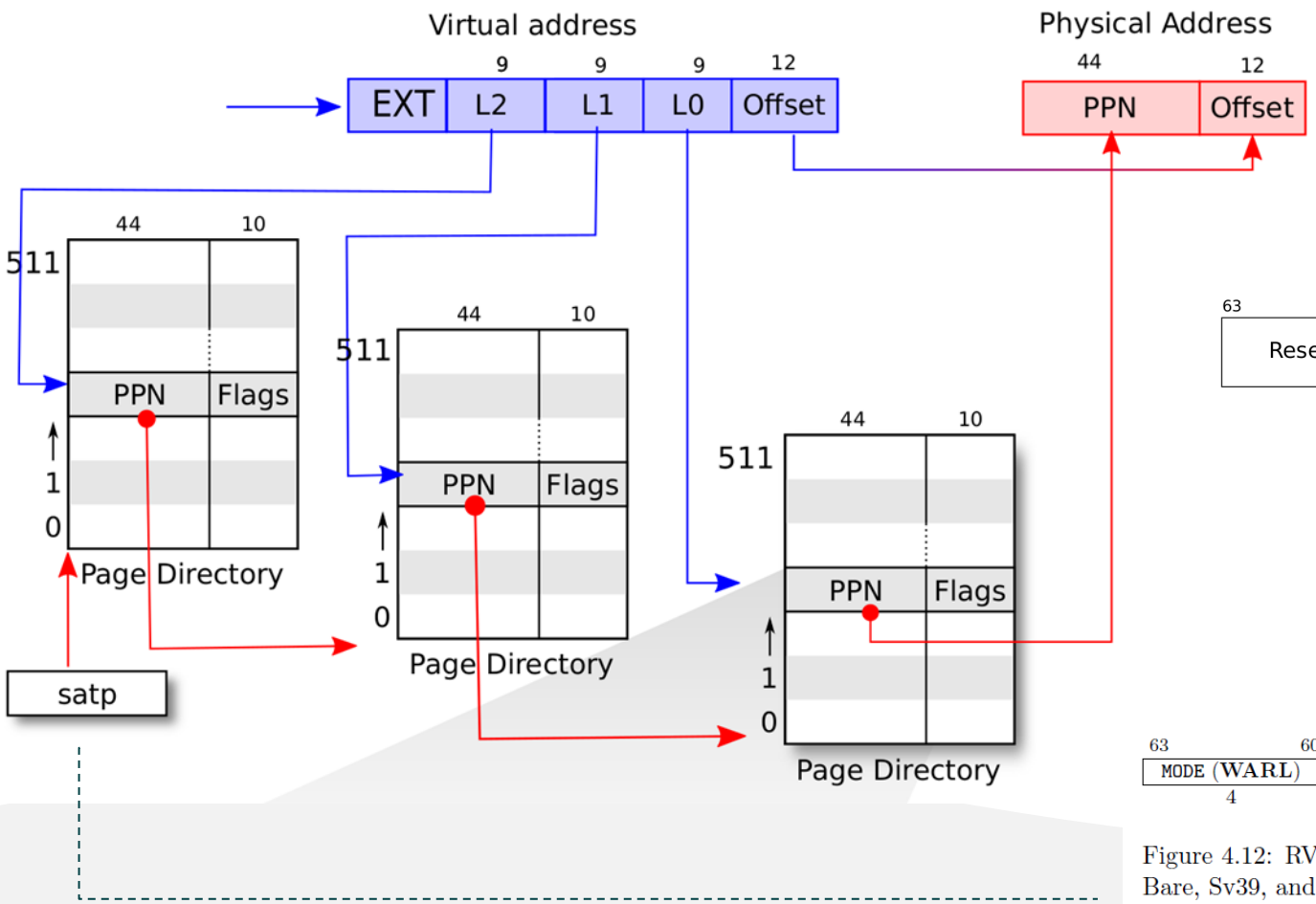
55 … 30	29 … 21	20 … 12	11 … 0
PPN[2]	PPN[1]	PPN[0]	page offset
26	9	9	12

Figure 4.17: Sv39 physical address.

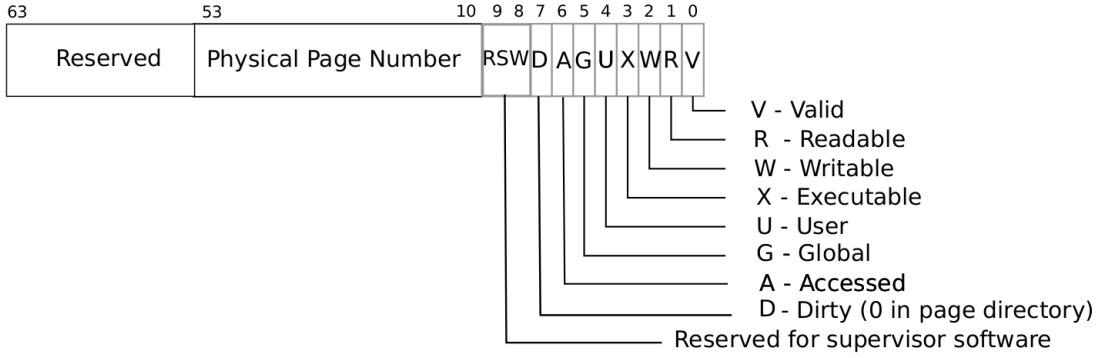
The virtual address has 64 bits, but only 0-38 bits are valid. The value of bits 39-63 must be equal to the value of 38 bits, otherwise it is an invalid address.

Page Table

Virtual address ⟶ Physical address



X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.



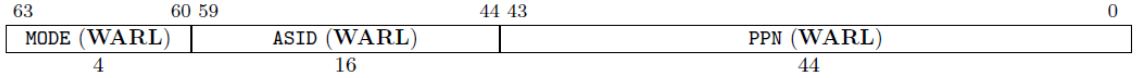


Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.

Modify kernel

Transfer the running environment of the kernel from the physical address space to the virtual address space 0x80200000 -> 0xffffffff80200000

- Modify the base address and align each with page size
- Modify the KERNEL_END_ADDRESS
- Modify the enrty point of kernel

```
BASE_ADDRESS = 0xffffffff80200000;

SECTIONS
{
    . = BASE_ADDRESS;

    kernel_start = .;

    . = ALIGN(4K);
    text_start = .;

    .text : {
        *(.text.entry)
        *(.text .text.*)
    }
    ...
}
```

```
lazy_static! {
    pub static ref KERNEL_END_ADDRESS: VirtualAddress =
VirtualAddress(kernel_end as usize);
}
pub const KERNEL_MAP_OFFSET: usize = 0xffff_ffff_0000_0000;
```

Modify kernel

```
# %hi gets [12,32) bits
# %lo gets [0,12) bits
_start:
    lui t0, %hi(boot_page_table)
    li t1, 0xffffffff00000000
    sub t0, t0, t1
    srli t0, t0, 12
    #the mark of Sv39 in satp
    li t1, (8 << 60)
    or t0, t0, t1
    # write in satp
    csrw satp, t0
    #refresh TLB
    sfence.vma

    lui sp, %hi(boot_stack_top)
    addi sp, sp,
    %lo(boot_stack_top)

    lui t0, %hi(rust_main)
    addi t0, t0, %lo(rust_main)
    jr t0
```

```
    .section .data
    .align 12
boot_page_table:
    .quad 0
    .quad 0
    # 2th items : 0x8000_0000 -> 0x8000_0000, 0xcf:  VRWXAD = 1
    .quad (0x80000 << 10) | 0xcf
    .zero 507 * 8
    # 510: 0xffff_ffff_8000_0000 -> 0x8000_0000, 0xcf  VRWXAD = 1
    .quad (0x80000 << 10) | 0xcf
    .quad 0
```

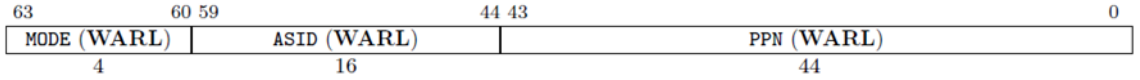


Figure 4.12: RV64 Supervisor address translation and protection register **satp**, for MODE values Bare, Sv39, and Sv48.

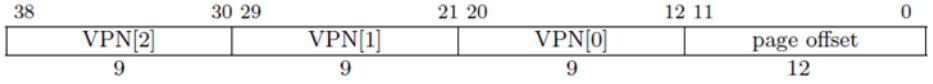


Figure 4.16: Sv39 virtual address.

Implement the Page Table

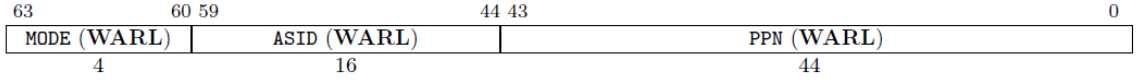


Figure 4.12: RV64 Supervisor address translation and protection register satp, for MODE values Bare, Sv39, and Sv48.

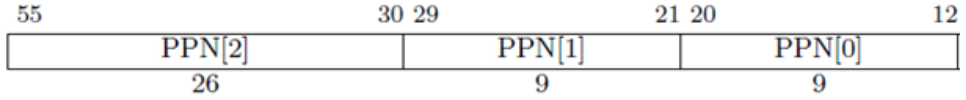
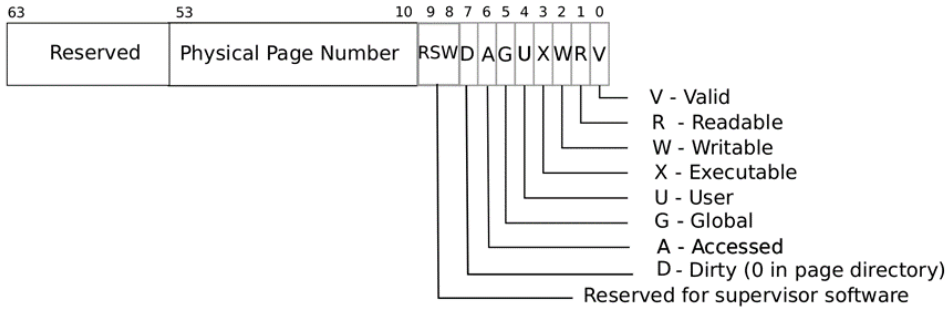
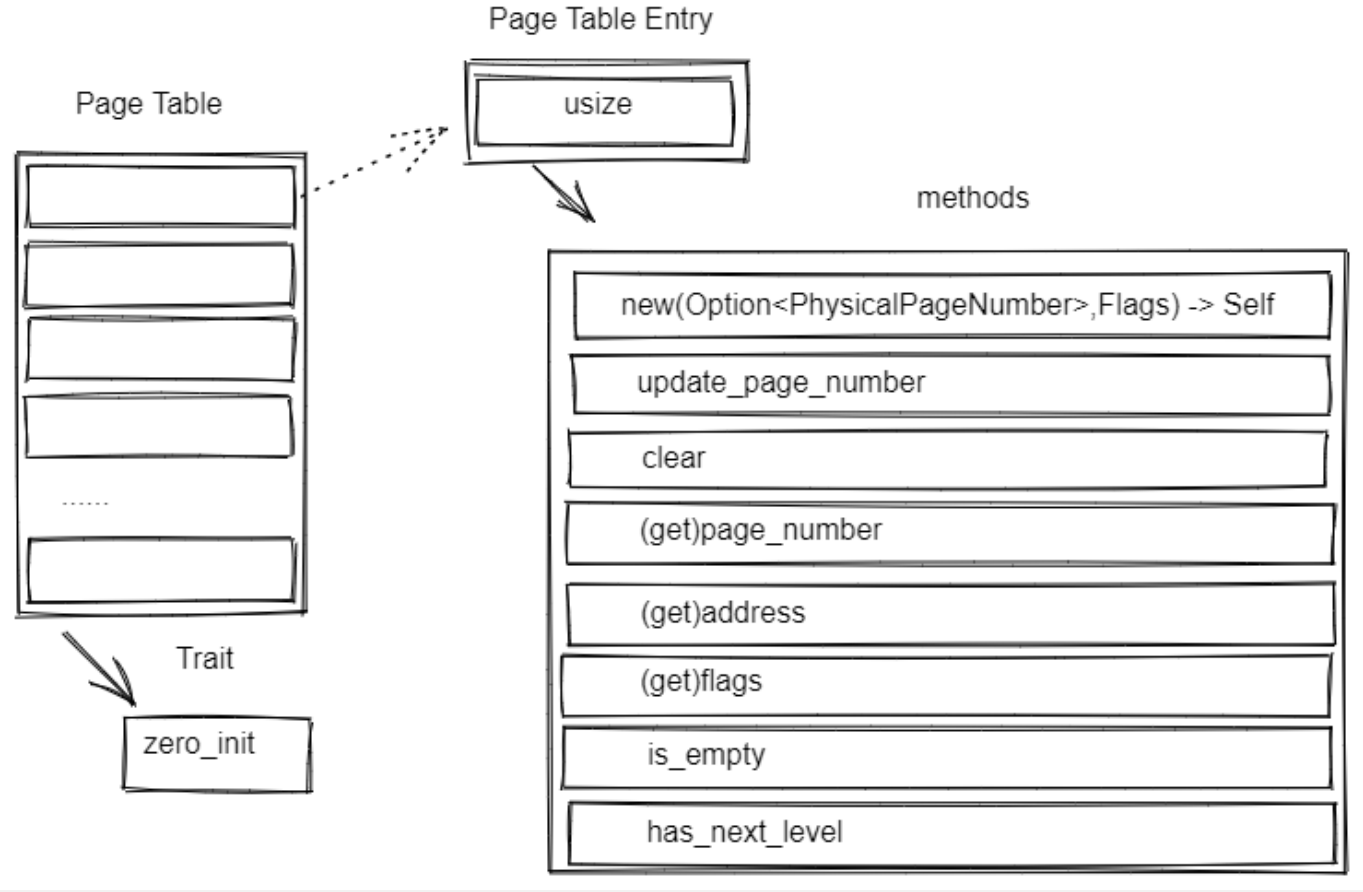


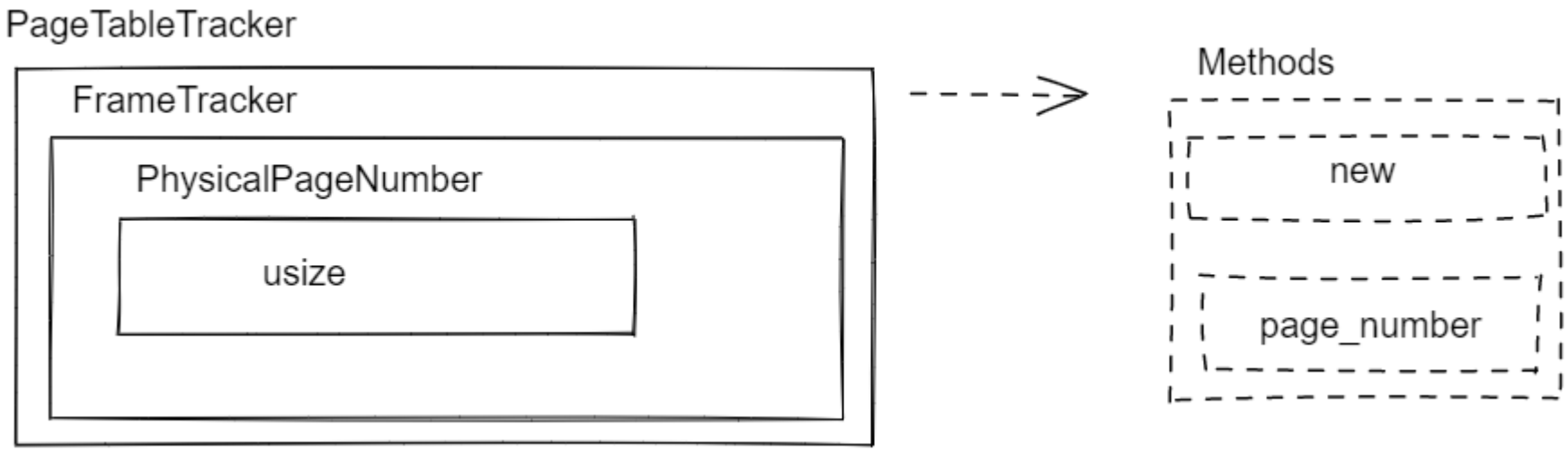
Figure 4.17: Sv39 physical address.





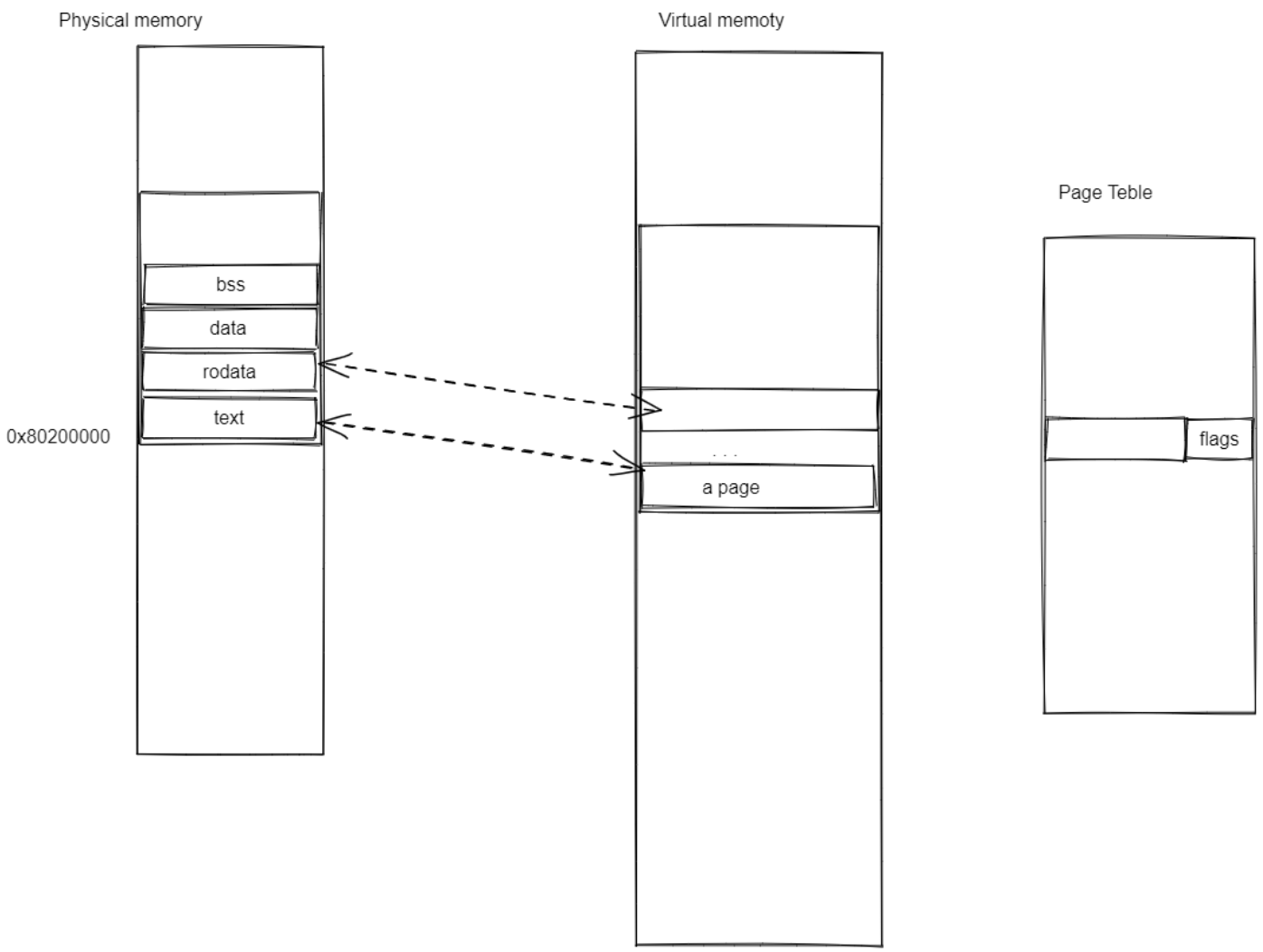
Implement the Page Table

The page table will be placed in the physical page we allocated.
We use a struct PageTableTracker to record our page table.



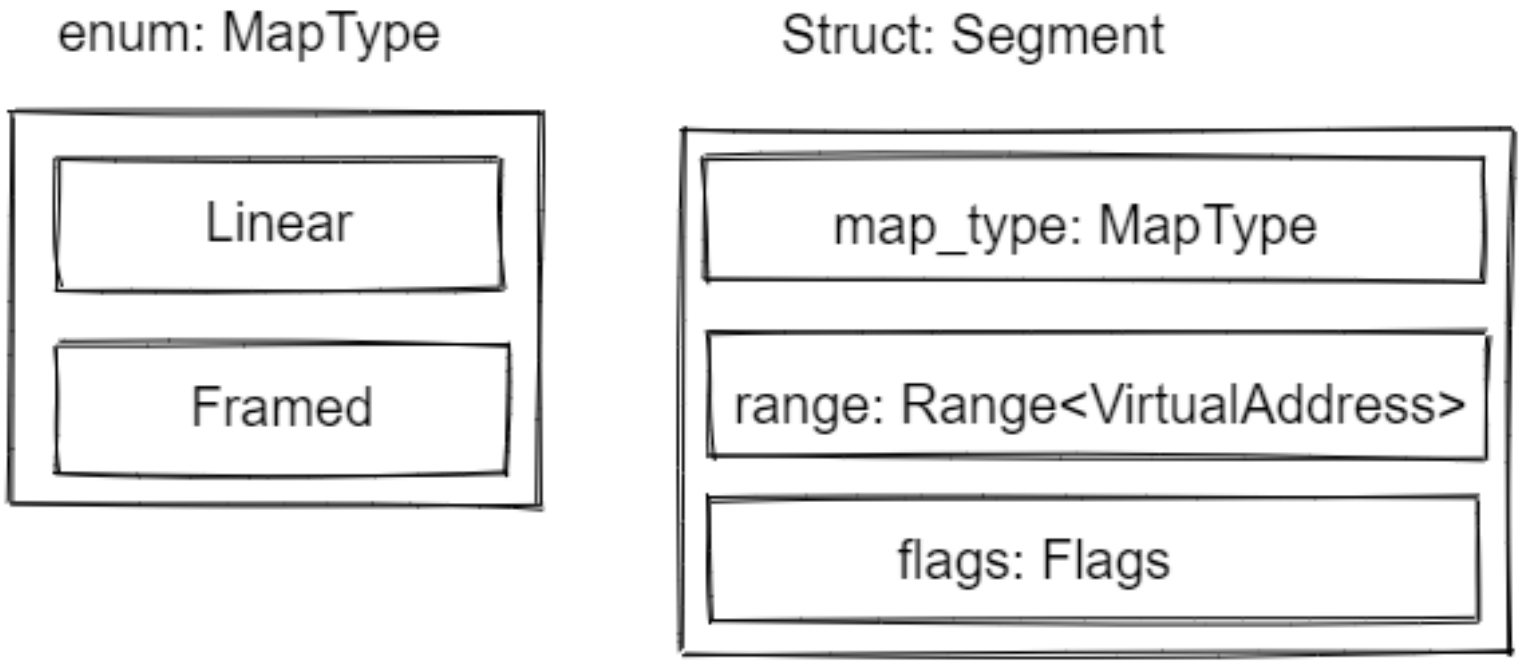
Kernel remapping

- Remap these segments separately so
- Set correct access permissions

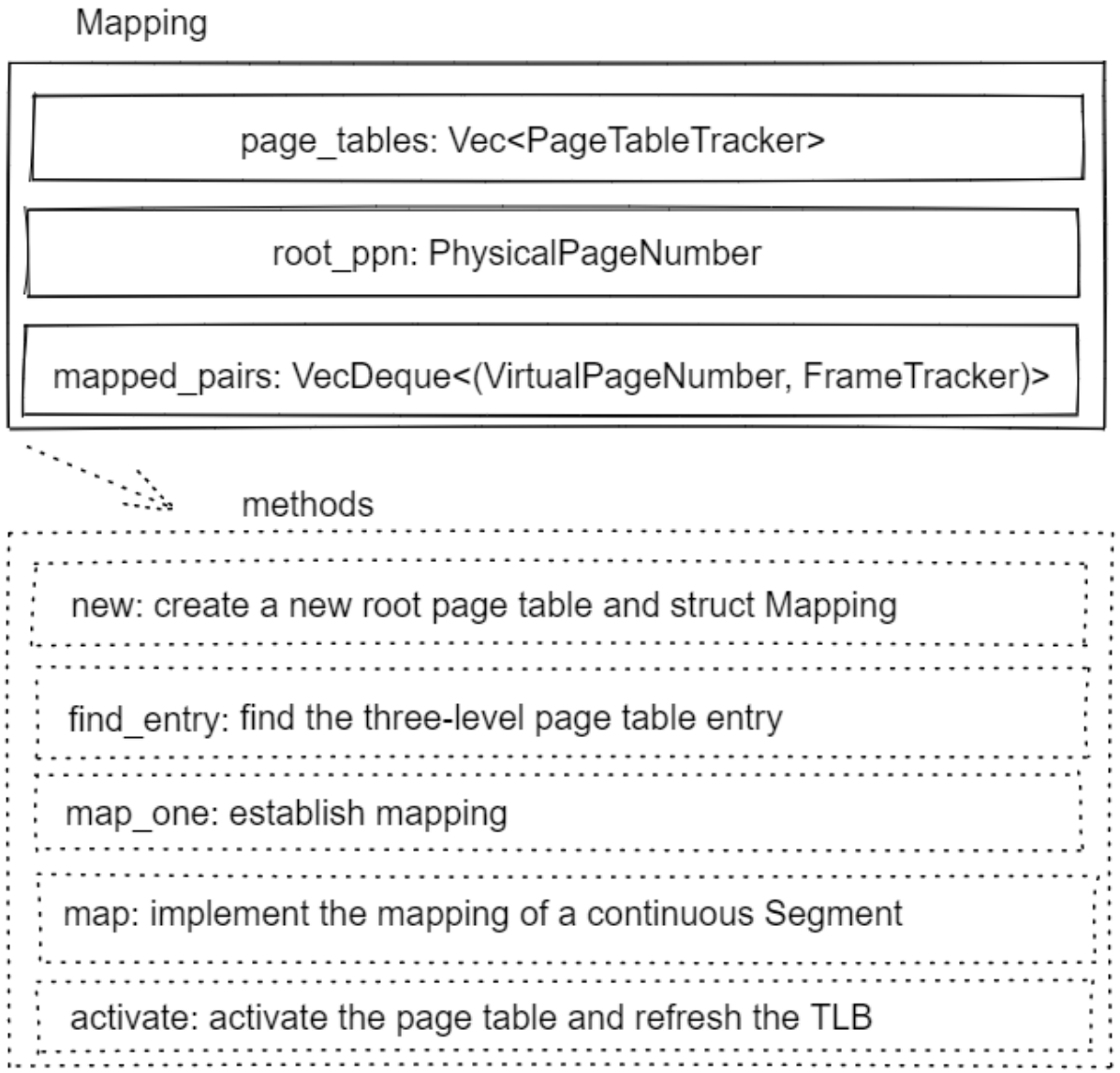


Kernel segment

We use enum and struct to encapsulate the type of memory segment mapping and the memory segment itself:



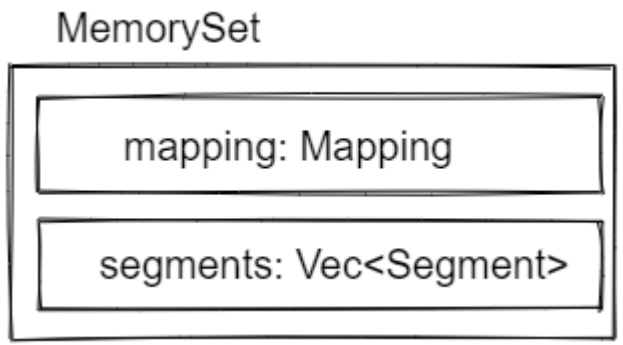
Implement mapping



Implement Memoryset

SUSTech Southern University of Science and Technology

We write each segment of the kernel into the encapsulated Mapping structure according to different attributes, and use it as a new structure MemorySet for threads.




```rust
impl MemorySet {
    pub fn new_kernel() -> MemoryResult<MemorySet> {
        extern "C" {
            fn text_start();
            fn rodata_start();
            ...
        }

        let segments = vec![
            // .text , r-x
            Segment {
                map_type: MapType::Linear,
                range: Range::from((text_start as usize)..(rodata_start as usize)),
                flags: Flags::READABLE | Flags::EXECUTABLE,
            },
                ...
        ];
        let mut mapping = Mapping::new()?;
        for segment in segments.iter() {
            mapping.map(segment, None)?;
        }
        Ok(MemorySet { mapping, segments })
    }

    pub fn activate(&self) {
        self.mapping.activate()
    }
}
```

Thread and Process

```rust
pub struct Thread {
    /// ID
    pub id: ThreadID,
    /// the stack of thread
    pub stack: Range<VirtualAddress>,
    /// belongs to the process
    pub process: Arc<Process>,
    /// Use `Mutex` to wrap some variable variables
    pub inner: Mutex<ThreadInner>,
}
```

```rust
/// variable part
pub struct ThreadInner {
    /// context
    pub context: Option<Context>,
    /// is sleeping
    pub sleeping: bool,
    /// is dead
    pub dead: bool,
}
```

```rust
/// Process
pub struct Process {
    /// Whether it belongs to user mode
    pub is_user: bool,
    /// use 'Mutex' to wrap some variable parts.
    pub inner: Mutex<ProcessInner>,
}

pub struct ProcessInner {
    /// Common memory map of threads in the process
    pub memory_set: MemorySet,
}
```

Thread Processor

SUSTech Southern University of Science and Technology

An abstract thread processor:

```rust
/// Thread scheduling and management
pub struct Processor {
    current_thread: Option<Arc<Thread>>,
    scheduler: SchedulerImpl<Arc<Thread>>,
    sleeping_threads: HashSet<Arc<Thread>>,
}
```

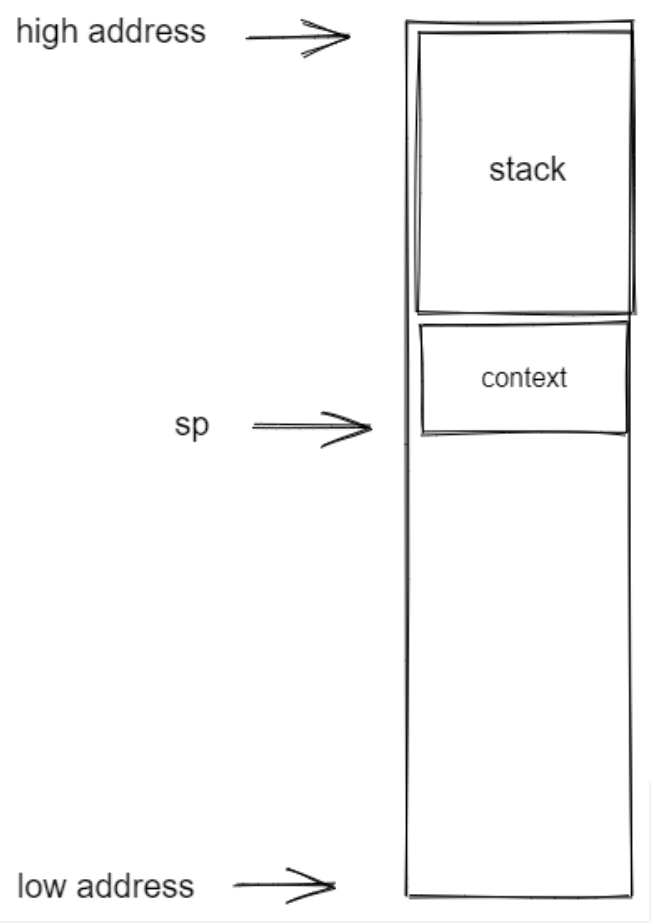
Scheduler(FIFO)

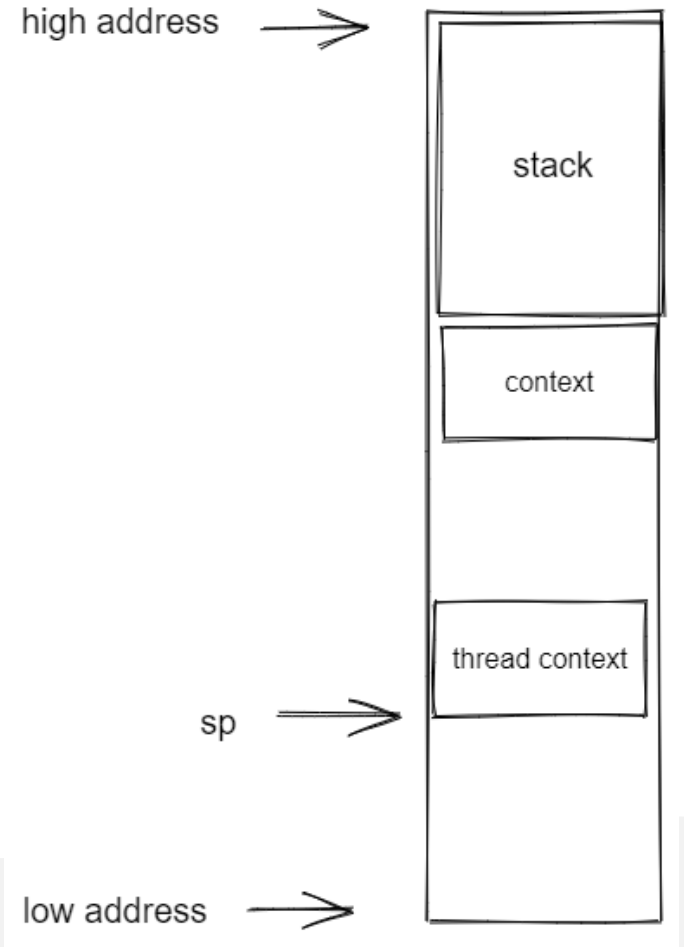
```rust
pub struct FifoScheduler<ThreadType: Clone + Eq>
{   pool: LinkedList<ThreadType>,
}
```

Thread

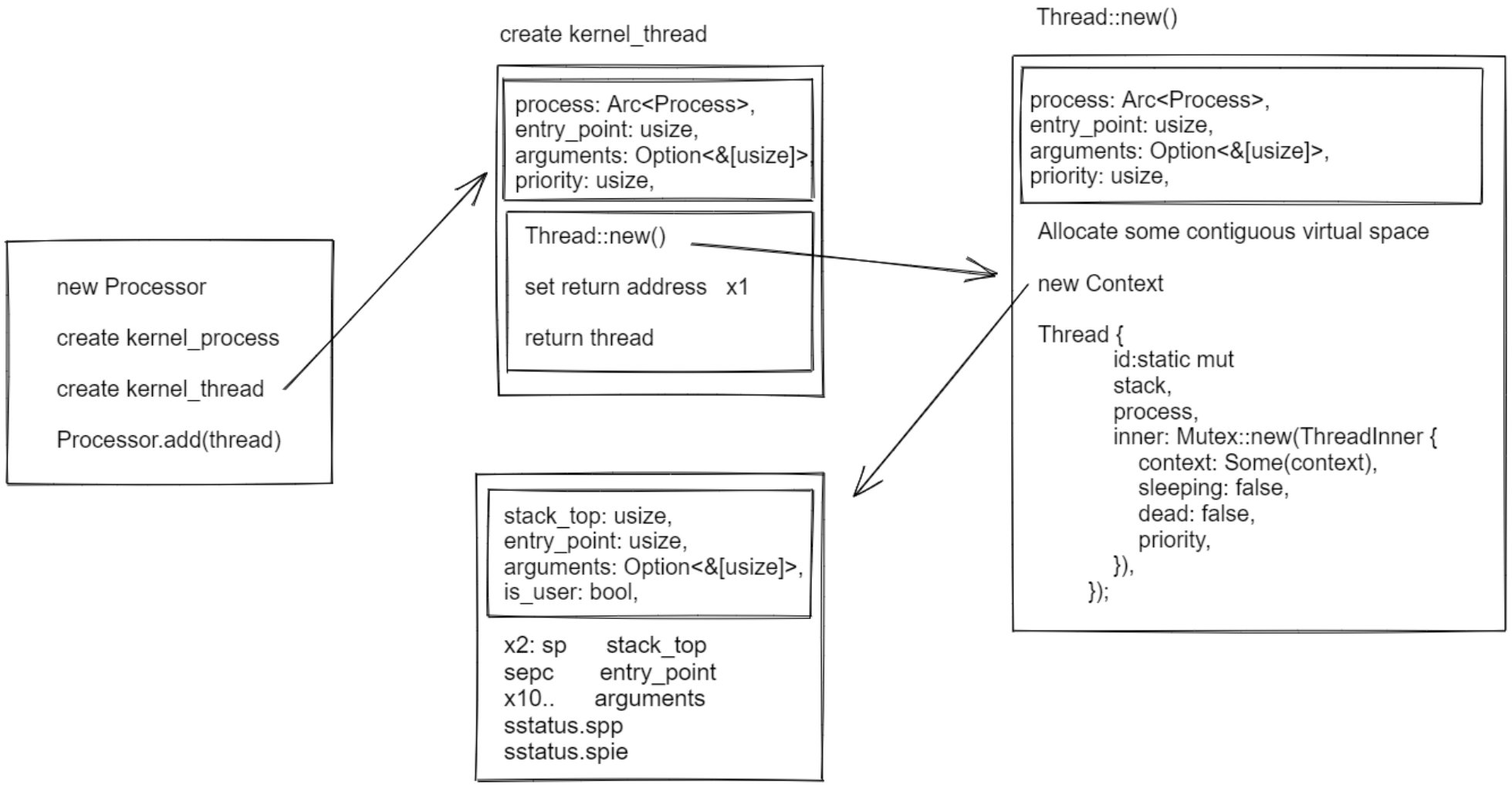
When we have finished handling the interrupt:

```
.globl __restore
__restore:
    mv      sp, a0
```





Create a thread



Create a thread

```rust
{
    let mut processor = PROCESSOR.lock();
    // create a kernel process
    let kernel_process = Process::new_kernel().unwrap();

    for i in 1..9usize {
        processor.add_thread(create_kernel_thread(
            kernel_process.clone(),
            sample_process as usize,
            Some(&[i]),
        ));
    }
}
```

```rust
fn kernel_thread_exit() {
    PROCESSOR.lock().current_thread().as_ref().inner().dead = true;
    unsafe { llvm_asm!("ebreak" :::: "volatile") };
}
```

```rust
/// create a kernelthread
pub fn create_kernel_thread(
    process: Arc<Process>,
    entry_point: usize,
    arguments: Option<&[usize]>,
    priority: usize,
) -> Arc<Thread> {
    // create thread
    let thread: Arc<Thread> = Thread::new(process, entry_point, arguments, priority).unwrap();
    // set the return address: kernel_thread_exit
    thread: Arc<Thread>
        .as_ref(): &Thread
        .inner(): MutexGuard<ThreadInner>
        .context: Option<Context>
        .as_mut(): Option<&mut Context>
        .unwrap(): &mut Context
        .set_ra(kernel_thread_exit as usize);

    thread
}
```

Create a thread

```rust
extern "C" {
    fn __restore(context: usize);
}
// get the Context of the first thread
let context = PROCESSOR.lock().prepare_next_thread();
// Call the first thread
unsafe { __restore(context as usize) };
unreachable!()
```

```asm
.globl __restore
__restore:
    mv      sp, a0

    LOAD    t0, 32
    LOAD    t1, 33
    csrw    sstatus, t0
    csrw    sepc, t1

    addi    t0, sp, CONTEXT_SIZE * REG_SIZE
    csrw    sscratch, t0

    LOAD    x1, 1
    .set    n, 3
    .rept   29
        LOAD_N  %n
        .set    n, n + 1
    .endr

    LOAD    x2, 2
    sret
```

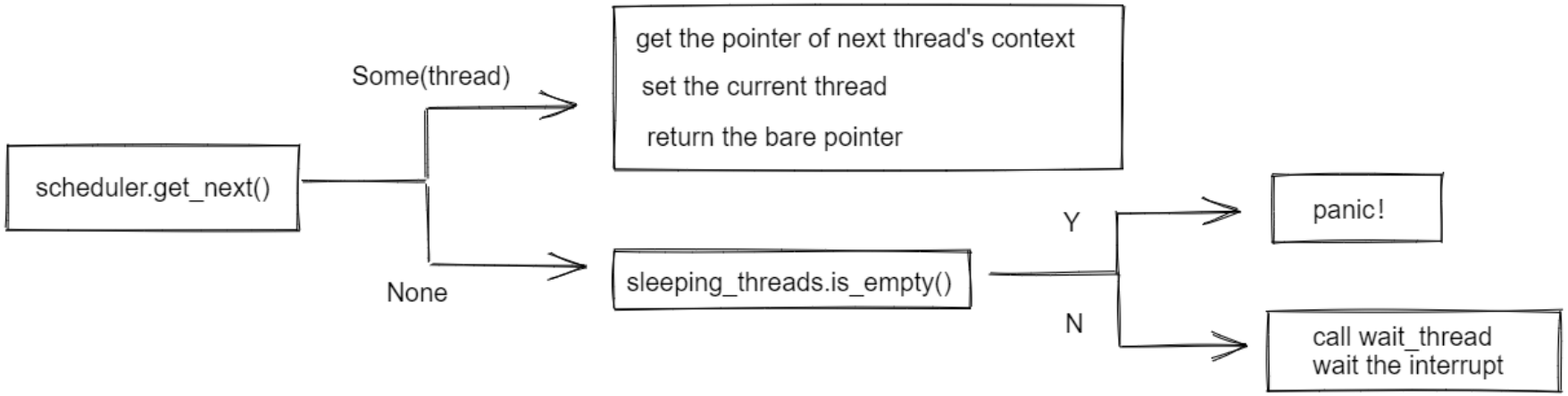
Thread switching

Add a return value to the interrupt handler to handle process switching

```
#[no_mangle]
pub fn handle_interrupt(context: &mut Context, scause: Scause, stval:usize) -> *mut
Context{
        {
            let mut processor = PROCESSOR.lock();
            let current_thread = processor.current_thread();
            if current_thread.as_ref().inner().dead {
                println!("thread {} exit", current_thread.id);
                processor.kill_current_thread();
                return processor.prepare_next_thread();
            }
        }
    match scause.cause() {
        Trap::Exception(Exception::Breakpoint) => breakpoint(context),
        Trap::Interrupt(Interrupt::SupervisorTimer) => supervisor_timer(context),
        _ => fault("unimplemented interrupt type", scause, stval),
    }
}
```

Thread switching

```rust
fn breakpoint(context: &mut Context) -> *mut Context {
    println!("Breakpoint at 0x{:x}", context.sepc);
    context.sepc += 2;
    context
}

fn supervisor_timer(context: &mut Context) -> *mut Context
{   timer::tick();
    PROCESSOR.lock().park_current_thread(context);
    PROCESSOR.lock().prepare_next_thread()
}
```



Thanks!