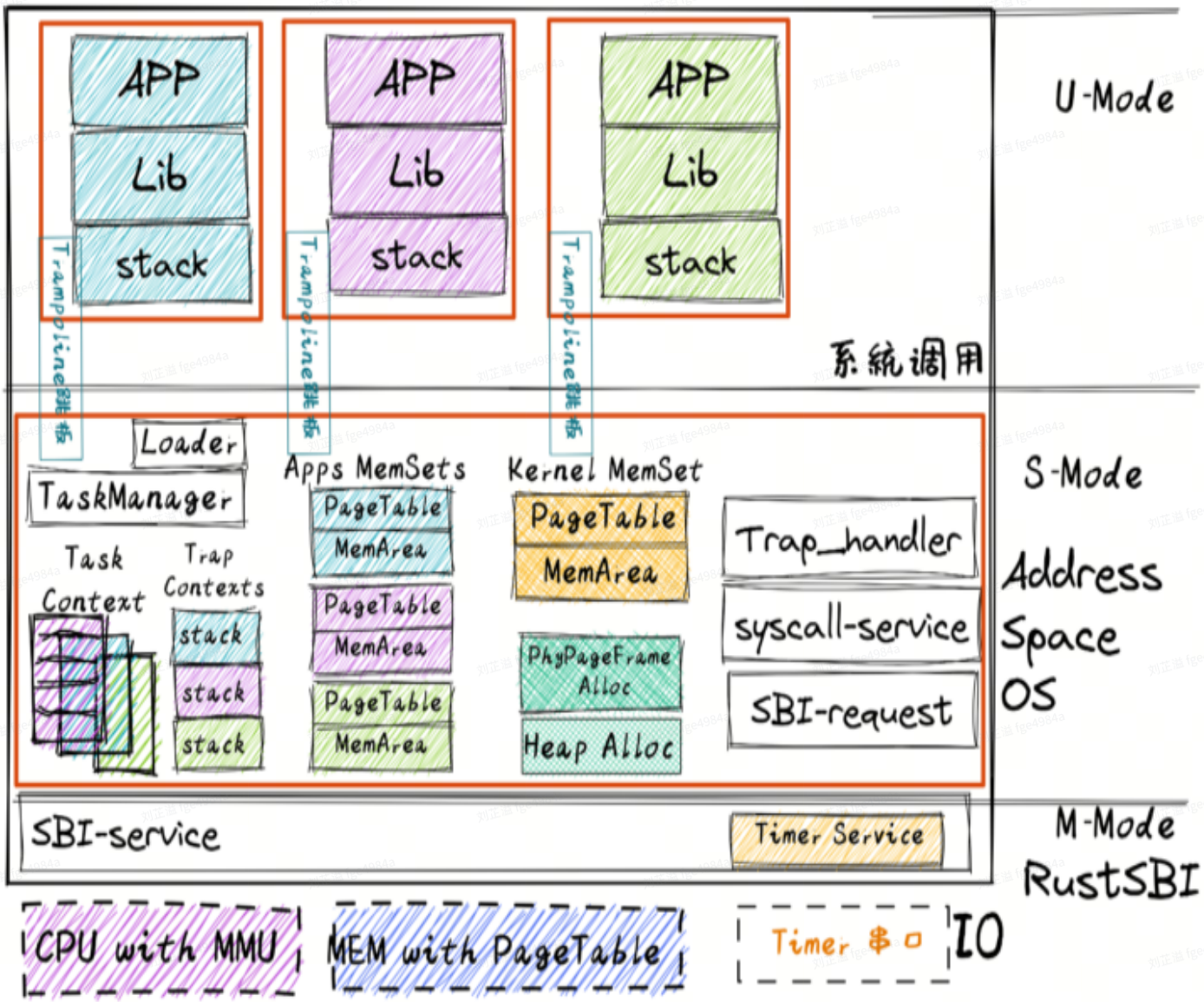
Address Space OS

Address Space OS



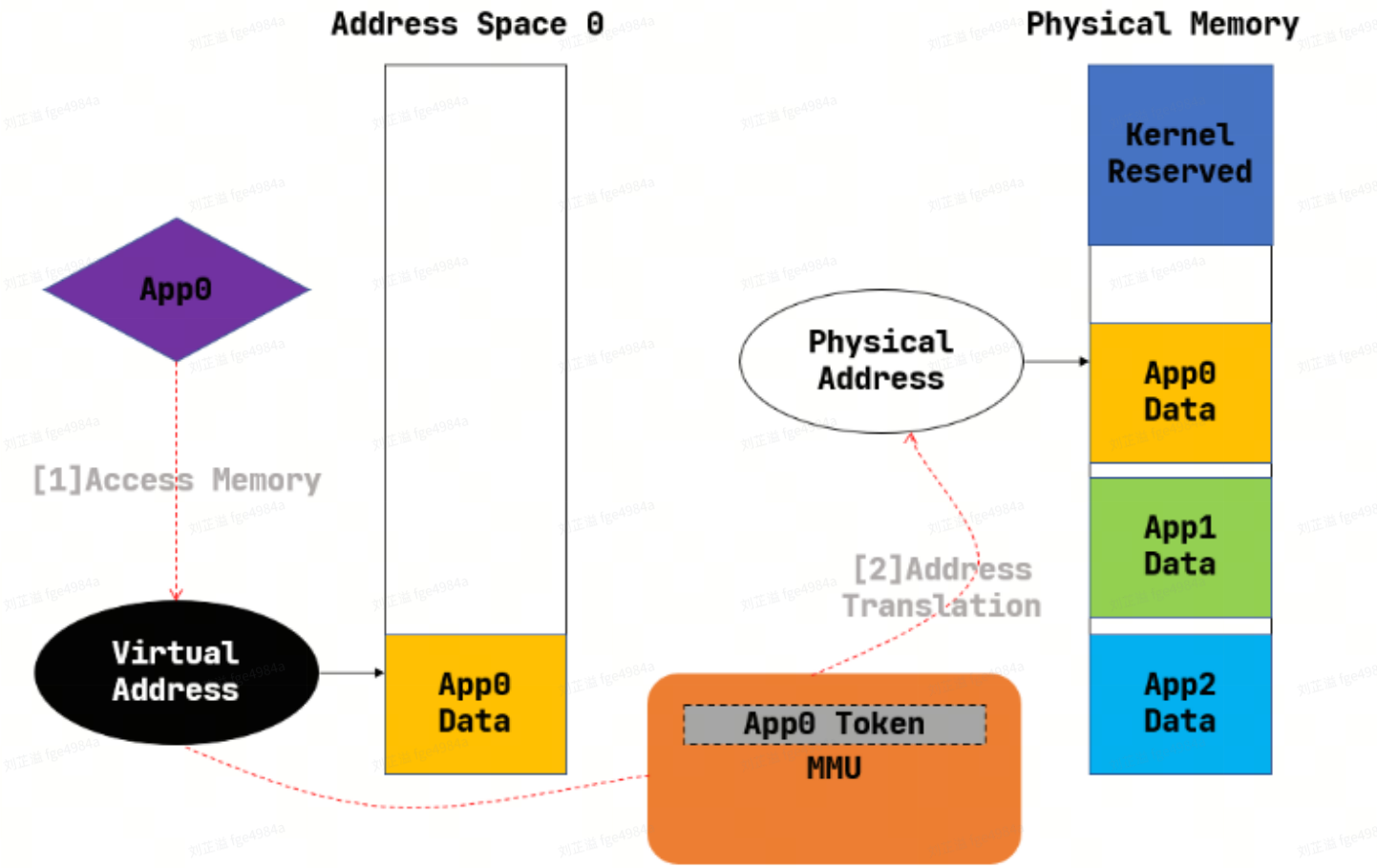
动态内存分配

- `alloc` 库需要我们提供给它一个 `全局的动态内存分配器` ，它会利用该分配器来管理堆空间，从而使得与堆相关的智能指针或容器数据结构可以正常工作。具体而言，我们的动态内存分配器需要实现它提供的 `GlobalAlloc` Trait，这个 Trait 有两个必须实现的抽象接口：

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

- 两个接口中都有一个 `alloc::alloc::Layout` 类型的参数， 它指出了分配的需求，分为两部分，分别是所需空间的大小 `size` ，以及返回地址的对齐要求 `align` 。

地址空间



SV39多级页表的硬件机制

虚拟地址和物理地址

- 默认情况下 MMU 未被使能（启用），此时无论 CPU 位于哪个特权级，访存的地址都会作为一个物理地址交给对应的内存控制单元来直接访问物理内存。我们可以通过修改 S 特权级的一个名为 `satp` 的 CSR 来启用分页模式，在这之后 S 和 U 特权级的访存地址会被视为一个虚拟地址，它需要经过 MMU 的地址转换变为一个物理地址，再通过它来访问物理内存；而 M 特权级的访存地址，我们可设定是内存的物理地址。
- 虚拟地址和物理地址都被分成两部分：它们的低 12 位，即 [11:0] 被称为 **页内偏移** (Page Offset)，它描述一个地址指向的字节在它所在页面中的相对位置。而虚拟地址的高 27 位，即 [38:12] 为它的虚拟页号 VPN，同理物理地址的高 44 位，即 [55:12] 为它的物理页号 PPN，页号可以用来定位一个虚拟/物理地址属于哪一个虚拟页面/物理页帧。

地址与页号相互转换

- 其中 `PAGE_SIZE` 为 4096 ， `PAGE_SIZE_BITS` 为 12 ，它们均定义在 `config` 子模块中，分别表示每个页面的大小和页内偏移的位宽。从物理页号到物理地址的转换只需左移 12 位即可，但是物理地址需要保证它与页面大小对齐才能通过右移转换为物理页号。
- 对于不对齐的情况，物理地址不能通过 `From/Into` 转换为物理页号，而是需要通过它自己的 `floor` 或 `ceil` 方法来进行下取整或上取整的转换。

```
// os/src/mm/address.rs
impl PhysAddr {
    pub fn floor(&self) -> PhysPageNum { PhysPageNum(self.0 / PAGE_SIZE) }
    pub fn ceil(&self) -> PhysPageNum { PhysPageNum((self.0 + PAGE_SIZE - 1) / PAGE_SIZE) }
}
```

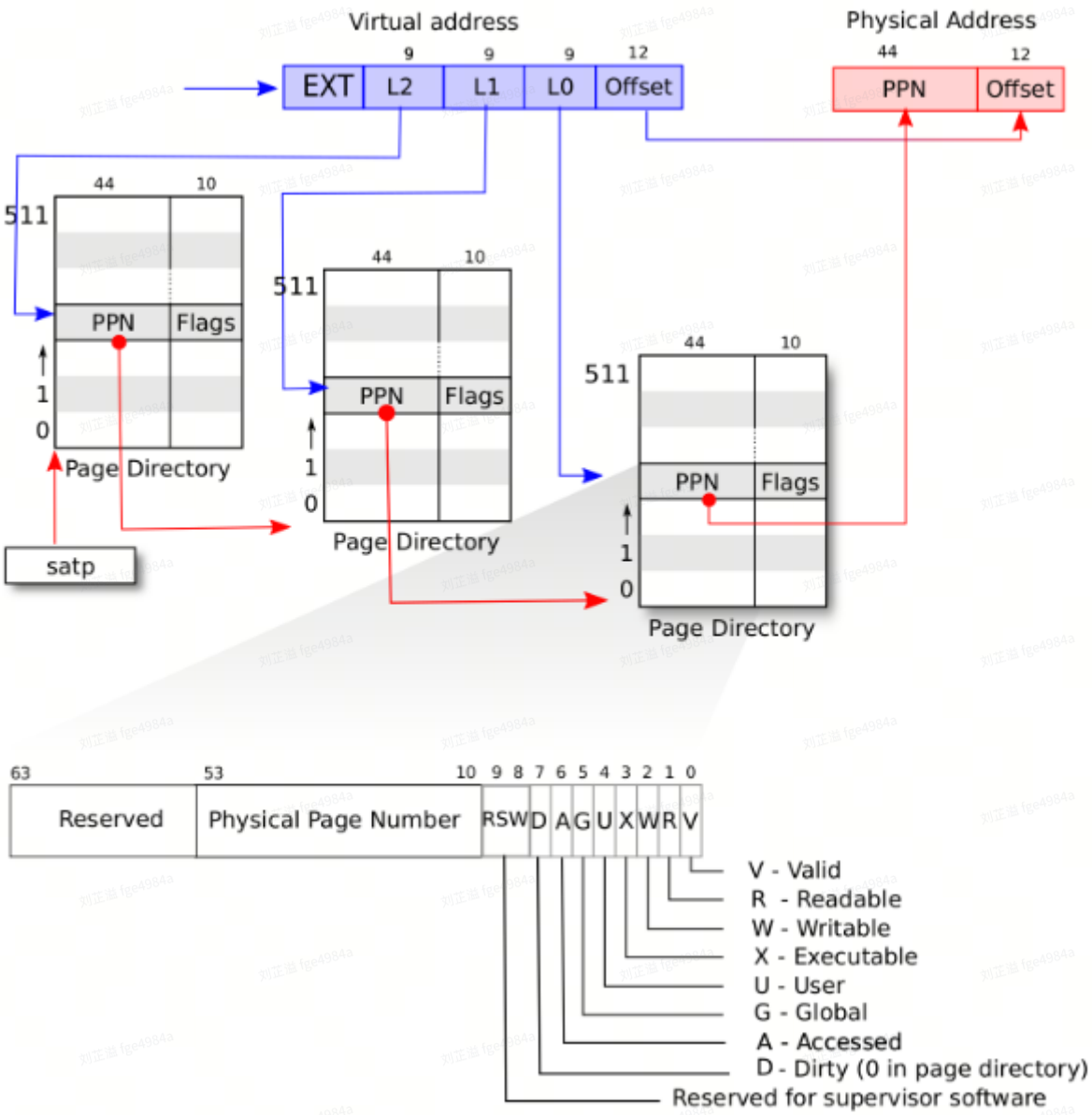
多级页表

具体来说，假设我们有虚拟地址 (VPN0,VPN1,VPN2,offset)：

- 我们首先会记录装载「当前所用的一级页表的物理页」的页号到 satp 寄存器中；
- 把 VPN0 作为偏移在一级页表的物理页中找到二级页表的物理页号；
- 把 VPN1 作为偏移在二级页表的物理页中找到三级页表的物理页号；
- 把 VPN2 作为偏移在三级页表的物理页中找到要**访问位置的物理页号**；
- 物理页号**对应的物理页基址**（即物理页号左移12位）加上 offset 就是虚拟地址对应的物理地址。

这样处理器通过这种多次转换，终于从虚拟页号找到了一级页表项，从而得出了物理页号和虚拟地址所对应的物理地址。刚才我们提到若页表项满足 R,W,X 都为 0，表明这个页表项指向下一级页表。在这里三级和二级页表项的 R,W,X 为 0 应该成立，因为它们指向了下一级页表：

- 修改stap寄存器，说**明内核切换到了一个与先前映射方式完全不同的页表**。此时快表里面存储的映射已经失效了，这种情况下内核要在修改 satp 的指令后面马上使用 sfence.vma 指令刷新清空整个 TLB。



管理SV39多级页表

可用物理页的分配与回收

我们用一个左闭右开的物理页号区间来表示可用的物理内存，则：

- 区间的左端点应该是 `ekernel` 的物理地址以上取整方式转化成的物理页号；
- 区间的右端点应该是 `MEMORY_END` 以下取整方式转化成的物理页号。

这个区间将被传给我们后面实现的物理页帧管理器用于初始化。

```
// os/src/mm/frame_allocator.rs
trait FrameAllocator {
    fn new() -> Self;
    fn alloc(&mut self) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum);
}
```

栈式物理页帧管理策略StackFrameAllocator

- 物理页号区间 [`current` , `end`) 此前均 *从未* 被分配出去过，而向量 `recycled` 以后入先出的方式保存了被回收的物理页号

```rust
// os/src/mm/frame_allocator.rs
pub struct StackFrameAllocator {
    current: usize,  //空闲内存的起始物理页号
    end: usize,      //空闲内存的结束物理页号
    recycled: Vec<usize>,
}
```

- 初始化

```rust
// os/src/mm/frame_allocator.rs
impl FrameAllocator for StackFrameAllocator {
    fn new() -> Self {
        Self {
        current: 0,
        end: 0,
        recycled: Vec::new(),
        }
    }
}

impl StackFrameAllocator {
    pub fn init(&mut self, l: PhysPageNum, r: PhysPageNum) {
        self.current = l.0;
        self.end = r.0;
    }
}
```

- 分配和回收
  - 在分配 `alloc` 的时候，首先会检查栈 `recycled` 内有没有之前回收的物理页号，如果有的话直接弹出栈顶并返回；否则的话我们只能从之前从未分配过的物理页号区间 [`current` , `end`) 上进行分配，我们分配它的左端点 `current` ，同时将管理器内部维护的 `current` 加 `1` 代表 `current` 已被分配了
  - 在回收 `dealloc` 的时候，我们需要检查回收页面的合法性，然后将其压入 `recycled` 栈中。回收页面合法有两个条件：
    - 该页面之前一定被分配出去过，因此它的物理页号一定 < `current` ；

        - 该页面没有正处在回收状态，即它的物理页号不能在栈 `recycled` 中找到。
    - 我们通过 `recycled.iter()` 获取栈上内容的迭代器，然后通过迭代器的 `find` 方法试图寻找一个与输入物理页号相同的元素。其返回值是一个 `Option` ，如果找到了就会是一个 `Option::Some` ，这种情况说明我们内核其他部分实现有误，直接报错退出。

分配/回收物理页帧的接口

- `frame_alloc` 的返回值类型并不是 `FrameAllocator` 要求的物理页号 `PhysPageNum` ，而是将其进一步包装为一个 `FrameTracker` 。这里借用了 RAII 的思想，将一个物理页帧的生命周期绑定到一个 `FrameTracker` 变量上

```
// os/src/mm/frame_allocator.rs
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);
}
```

- 当一个 `FrameTracker` 生命周期结束被编译器回收的时候，我们需要将它控制的物理页帧回收到 `FRAME_ALLOCATOR` 中：

```
// os/src/mm/frame_allocator.rs
impl Drop for FrameTracker {
    fn drop(&mut self) {
        frame_dealloc(self.ppn);
    }
}
```

页表基本数据结构

- 每个应用的地址空间都对应一个不同的多级页表，这也就意味这不同页表的起始地址（即页表根节点的地址）是不一样的。因此 `PageTable` 要保存它根节点的物理页号 `root_ppn` 作为页表唯一的区分标志。此外，向量 `frames` 以 `FrameTracker` 的形式保存了页表所有的节点（包括根节点）所在的物理页帧

```rust
// os/src/mm/page_table.rs
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],
        }
    }
}
```

建立和解除虚拟地址和物理地址的映射关系

- 在多级页表中找到一个虚拟地址对应的页表项。找到之后，只要修改页表项的内容即可完成键值对的插入和删除。在寻找页表项的时候，可能出现页表的中间级节点还未被创建的情况，这个时候我们需要手动分配一个物理页帧来存放这个节点，并将这个节点接入到当前的多级页表的某级中。

```rust
impl VirtPageNum {
    /// Get the indexes of the page table entry
    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
    }
}

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

```rust

    /// Find PageTableEntry by VirtPageNum, create a frame for a 4KB page table if not exist
    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, idx) in idxs.iter().enumerate() {
            let pte = &mut ppn.get_pte_array()[*idx];
            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
    }
    /// Find PageTableEntry by VirtPageNum
    fn find_pte(&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, idx) in idxs.iter().enumerate() {
            let pte = &mut ppn.get_pte_array()[*idx];
            if i == 2 {
                result = Some(pte);
                break;
            }
            if !pte.is_valid() {
                return None;
            }
            ppn = pte.ppn();
        }
        result
    }
    /// set the map between virtual page number and physical page number
    #[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);
```

```
68          *pte = PageTableEntry::new(ppn, flags | PTEFlags::V);
69      }
70      /// remove the map between virtual page number and physical page number
71      #[allow(unused)]
72      pub fn unmap(&mut self, vpn: VirtPageNum) {
73          let pte = self.find_pte(vpn).unwrap();
74          assert!(pte.is_valid(), "vpn {:?} is invalid before unmapping", vpn);
75          *pte = PageTableEntry::empty();
76      }
77  }
```

- 手动查页表；后面对系统调用对应用地址空间修改使用

```
1   /// Temporarily used to get arguments from user space.
2   pub fn from_token(satp: usize) -> Self {
3       Self {
4           root_ppn: PhysPageNum::from(satp & ((1usize << 44) - 1)),
5           frames: Vec::new(),
6       }
7   }
8   /// get the page table entry from the virtual page number
9   pub fn translate(&self, vpn: VirtPageNum) -> Option<PageTableEntry> {
10      self.find_pte(vpn).map(|pte| *pte)
11  }
```

地址空间抽象

逻辑段：一段连续地址的虚拟内存

- 其中 `VPNRange` 描述一段虚拟页号的连续区间，表示该逻辑段在地址区间中的位置和长度。它是一个迭代器
- 一个逻辑段可能包含多个物理页帧

```
1   // os/src/mm/memory_set.rs
2   pub struct MapArea {
3       vpn_range: VPNRange,
4       data_frames: BTreeMap<VirtPageNum, FrameTracker>,
5       map_type: MapType,
6       map_perm: MapPermission,
7   }
```

地址空间：一系列有关联的逻辑段

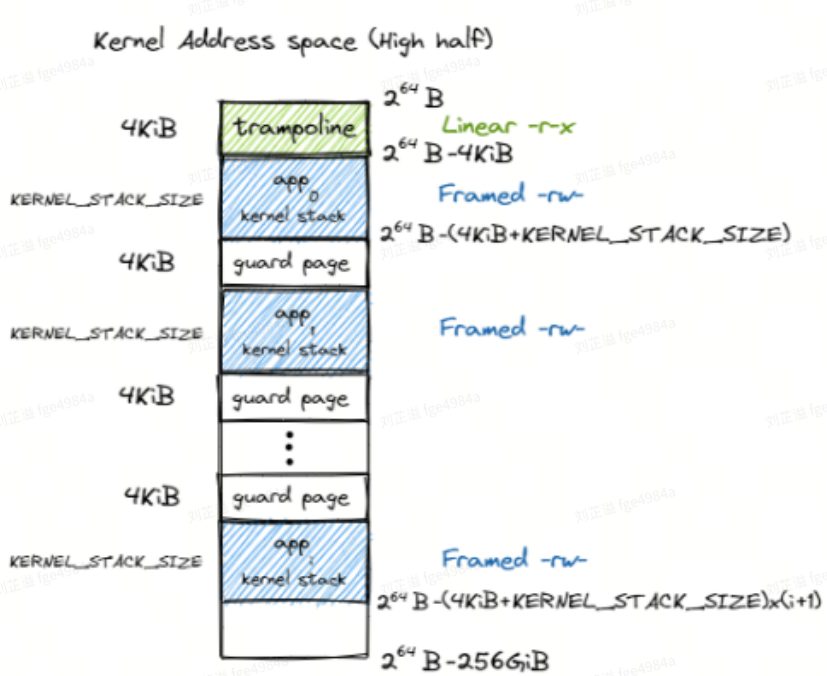
```rust
// os/src/mm/memory_set.rs
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
}

impl MemorySet {
    /// Create a new empty `MemorySet`.
    pub fn new_bare() -> Self {
        Self {
            page_table: PageTable::new(),
            areas: Vec::new(),
        }
    }
    /// Get the page table token
    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);
    }
}

impl MapArea {
    pub fn new(
        start_va: VirtAddr,
        end_va: VirtAddr,
```

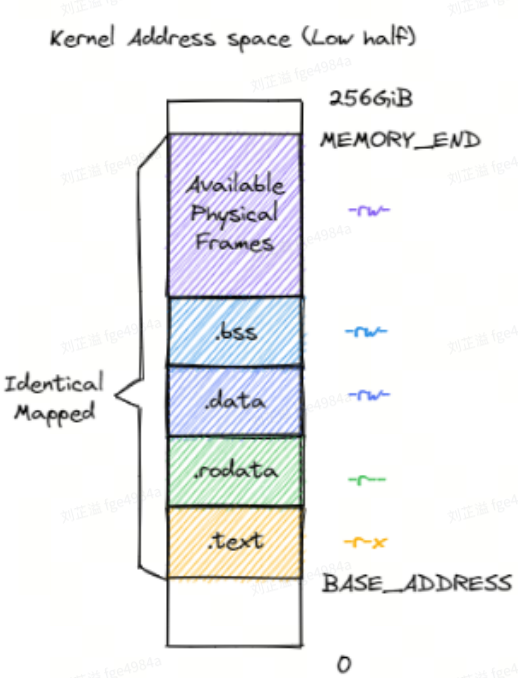
```
44          map_type: MapType,
45          map_perm: MapPermission,
46      ) -> Self {
47          let start_vpn: VirtPageNum = start_va.floor();
48          let end_vpn: VirtPageNum = end_va.ceil();
49          Self {
50              vpn_range: VPNRange::new(start_vpn, end_vpn),
51              data_frames: BTreeMap::new(),
52              map_type,
53              map_perm,
54          }
55      }
56      /// 遍历逻辑段中的所有虚拟页面，
57      /// 并以每个虚拟页面为单位依次在多级页表中进行键值对的插入或删除
58      pub fn map_one(&mut self, page_table: &mut PageTable, vpn: VirtPageNum) {
59          let ppn: PhysPageNum;
60          match self.map_type {
61              MapType::Identical => {
62                  ppn = PhysPageNum(vpn.0);
63              }
64              MapType::Framed => {
65                  let frame = frame_alloc().unwrap();
66                  ppn = frame.ppn;
67                  self.data_frames.insert(vpn, frame);
68              }
69          }
70          let pte_flags = PTEFlags::from_bits(self.map_perm.bits).unwrap();
71          page_table.map(vpn, ppn, pte_flags);
72      }
73      #[allow(unused)]
74      pub fn unmap_one(&mut self, page_table: &mut PageTable, vpn: VirtPageNum)
    {
75          if self.map_type == MapType::Framed {
76              self.data_frames.remove(&vpn);
77          }
78          page_table.unmap(vpn);
79      }
80      pub fn map(&mut self, page_table: &mut PageTable) {
81          for vpn in self.vpn_range {
82              self.map_one(page_table, vpn);
83          }
84      }
85      #[allow(unused)]
86      pub fn unmap(&mut self, page_table: &mut PageTable) {
87          for vpn in self.vpn_range {
88              self.unmap_one(page_table, vpn);
89          }
```

```rust
    }
    /// data: start-aligned but maybe with shorter length
    /// assume that all frames were cleared before
    pub fn copy_data(&mut self, page_table: &mut PageTable, data: &[u8]) {
        assert_eq!(self.map_type, MapType::Framed);
        let mut start: usize = 0;
        let mut current_vpn = self.vpn_range.get_start();
        let len = data.len();
        loop {
            let src = &data[start..len.min(start + PAGE_SIZE)];
            let dst = &mut page_table
                .translate(current_vpn)
                .unwrap()
                .ppn()
                .get_bytes_array()[..src.len()];
            dst.copy_from_slice(src);
            start += PAGE_SIZE;
            if start >= len {
                break;
            }
            current_vpn.step();
        }
    }
}
```

内核地址空间

- 左侧是高256GB地址分布，右侧是低256GB地址





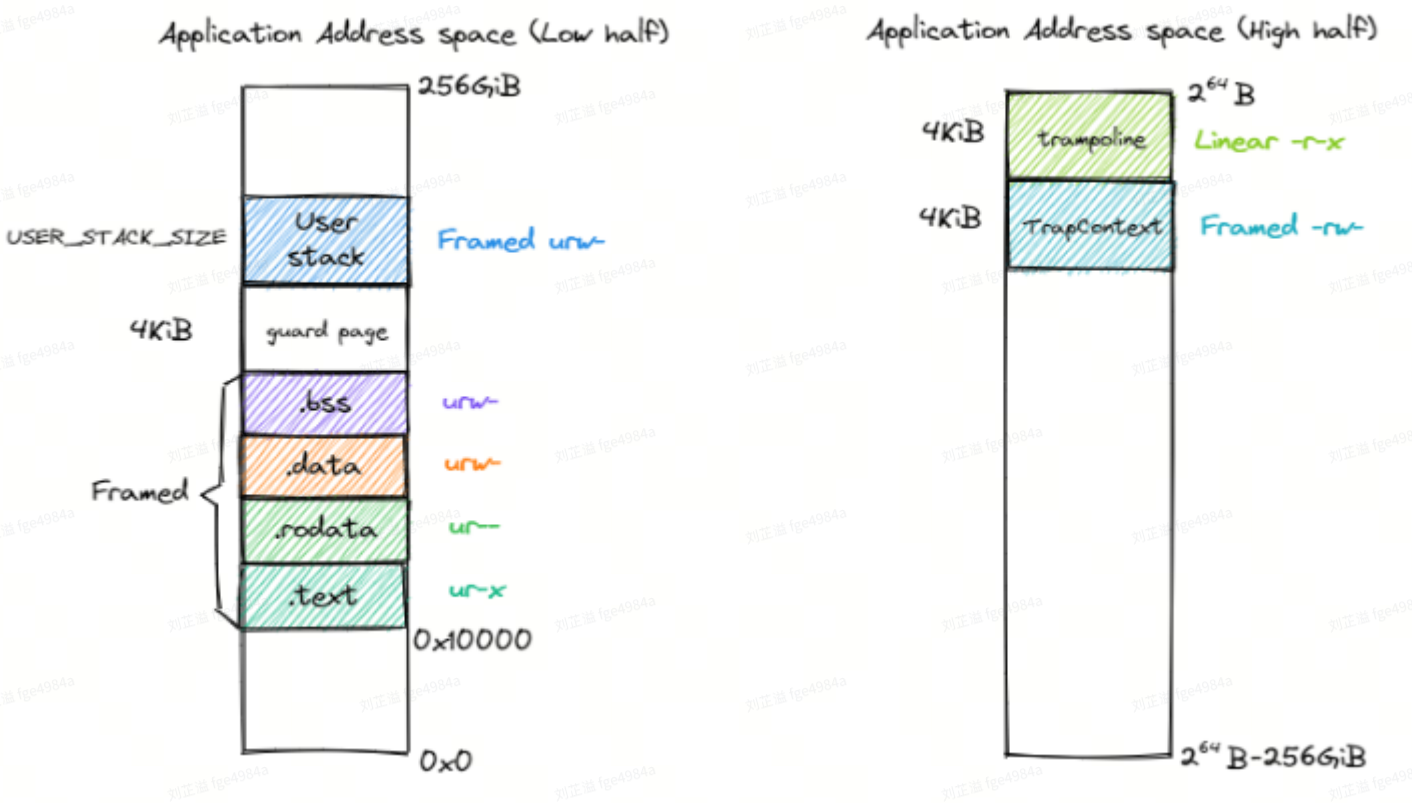
- 注意相邻两个内核栈之间会预留一个 **保护页面** (Guard Page)，它是内核地址空间中的空洞，多级页表中并不存在与它相关的映射。它的意义在于当内核栈空间不足（如调用层数过多或死递归）的时候，代码会尝试访问空洞区域内的虚拟地址，然而它无法在多级页表中找到映射，便会触发异常，此时控制权会交给内核 trap handler 函数进行异常处理。

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

```
        info!("mapping .data section");
        memory_set.push(
            MapArea::new(
                (sdata as usize).into(),
                (edata as usize).into(),
                MapType::Identical,
                MapPermission::R | MapPermission::W,
            ),
            None,
        );
        info!("mapping .bss section");
        memory_set.push(
            MapArea::new(
                (sbss_with_stack as usize).into(),
                (ebss as usize).into(),
                MapType::Identical,
                MapPermission::R | MapPermission::W,
            ),
            None,
        );
        info!("mapping physical memory");
        memory_set.push(
            MapArea::new(
                (ekernel as usize).into(),
                MEMORY_END.into(),
                MapType::Identical,
                MapPermission::R | MapPermission::W,
            ),
            None,
        );
        memory_set
    }
```

应用地址空间

- tranpoline和TrapContext两个虚拟页面虽然位于应用地址空间，但是它们并不包含 U 标志位，事实上它们在地址空间切换的时候才会发挥作用



- 在 `.text` 和 `.rodata` 中间以及 `.rodata` 和 `.data` 中间我们进行了页面对齐，因为前后两个逻辑段的访问方式限制是不同的，由于我们只能以页为单位对这个限制进行设置，因此就只能将下一个逻辑段对齐到下一个页面开始放置。而 `.data` 和 `.bss` 两个逻辑段由于访问限制相同（可读写），它们中间则无需进行页面对齐。

```
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*)
    }
}
```

ELF文件解析

```
/// Include sections in elf and trampoline and TrapContext and user stack,
/// also returns user_sp_base and entry point.
pub fn from_elf(elf_data: &[u8]) -> (Self, usize, usize) {
    let mut memory_set = Self::new_bare();
    // map trampoline
    memory_set.map_trampoline();
    // map program headers of elf, with U flag
    let elf = xmas_elf::ElfFile::new(elf_data).unwrap();
    let elf_header = elf.header;
    let magic = elf_header.pt1.magic;
    assert_eq!(magic, [0x7f, 0x45, 0x4c, 0x46], "invalid elf!");
    let ph_count = elf_header.pt2.ph_count();
    let mut max_end_vpn = VirtPageNum(0);
    for i in 0..ph_count {
        let ph = elf.program_header(i).unwrap();
        if ph.get_type().unwrap() == xmas_elf::program::Type::Load {
            let start_va: VirtAddr = (ph.virtual_addr() as usize).into();
            let end_va: VirtAddr = ((ph.virtual_addr() + ph.mem_size()) as
usize).into();
            let mut map_perm = MapPermission::U;
            let ph_flags = ph.flags();
            if ph_flags.is_read() {
                map_perm |= MapPermission::R;
            }
            if ph_flags.is_write() {
                map_perm |= MapPermission::W;
            }
            if ph_flags.is_execute() {
                map_perm |= MapPermission::X;
            }
```

```
            let map_area = MapArea::new(start_va, end_va, MapType::Framed,
map_perm);
            max_end_vpn = map_area.vpn_range.get_end();
            memory_set.push(
                map_area,
                Some(&elf.input[ph.offset() as usize..(ph.offset() +
ph.file_size()) as usize]),
            );
        }
    }
    // map user stack with U flags
    let max_end_va: VirtAddr = max_end_vpn.into();
    let mut user_stack_bottom: usize = max_end_va.into();
    // guard page
    user_stack_bottom += PAGE_SIZE;
    let user_stack_top = user_stack_bottom + USER_STACK_SIZE;
    memory_set.push(
        MapArea::new(
            user_stack_bottom.into(),
            user_stack_top.into(),
            MapType::Framed,
            MapPermission::R | MapPermission::W | MapPermission::U,
        ),
        None,
    );
    // used in sbrk
    memory_set.push(
        MapArea::new(
            user_stack_top.into(),
            user_stack_top.into(),
            MapType::Framed,
            MapPermission::R | MapPermission::W | MapPermission::U,
        ),
        None,
    );
    // map TrapContext
    memory_set.push(
        MapArea::new(
            TRAP_CONTEXT_BASE.into(),
            TRAMPOLINE.into(),
            MapType::Framed,
            MapPermission::R | MapPermission::W,
        ),
        None,
    );
    (
        memory_set,
```

```
75          user_stack_top,
76          elf.header.pt2.entry_point() as usize,
77      )
78  }
```

基于地址空间的分时多任务

- 对内核建立唯一的内核地址空间存放内核的代码、数据，同时对于每个应用维护一个它们自己的用户地址空间，因此在 Trap 的时候就需要进行地址空间切换，而在任务切换的时候无需进行（因为这个过程全程在内核内完成）。

内存管理子系统初始化

```
1  // os/src/mm/mod.rs
2  pub use memory_set::KERNEL_SPACE;
3
4  pub fn init() {
5      heap_allocator::init_heap();
6      frame_allocator::init_frame_allocator();
7      KERNEL_SPACE.exclusive_access().activate();
8  }
```

跳板机制的实现

- 扩展TrapContext：
  - 必须先切换到内核地址空间，这就需要将**内核地址空间的 token 写入 satp 寄存器**；
  - 之后还需要**保存应用的内核栈栈顶的位置，这样才能以它为基址保存 Trap 上下文**
- 切换地址空间

  切换到内核地址空间并跳转到 trap handler 了。
  - 第 30 行将内核地址空间的 token 载入到 t0 寄存器中；
  - 第 32 行将 trap handler 入口点的虚拟地址载入到 t1 寄存器中；
  - 第 34 行直接将 sp 修改为应用内核栈顶的地址；
  - 第 36~37 行将 satp 修改为内核地址空间的 token 并使用 `sfence.vma` 刷新快表，这就切换到了内核地址空间；
  - 第 39 行 最后通过 `jr` 指令跳转到 t1 寄存器所保存的trap handler 入口点的地址。

```
1  # read user stack from sscratch and save it in TrapContext
```

```
    csrr t2, sscratch
    sd t2, 2*8(sp)
    # load kernel_satp into t030
    ld t0, 34*8(sp)
    # load trap_handler into t132
    ld t1, 36*8(sp)
    # move to kernel_sp34   ld sp, 35*8(sp)
    # switch to kernel space36
    csrw satp, t0
    sfence.vma
    # jump to trap_handler39
    jr t1
```

当内核将 Trap 处理完毕准备返回用户态的时候会 *调用* `__restore` （符合RISC-V函数调用规范），它有两个参数：第一个是 Trap 上下文在应用地址空间中的位置，这个对于所有的应用来说都是相同的，在 a0 寄存器中传递；第二个则是即将回到的应用的地址空间的 token ，在 a1 寄存器中传递。

- 第 44~45 行先切换回应用地址空间（注：Trap 上下文是保存在应用地址空间中）；
- 第 46 行将传入的 Trap 上下文位置保存在 `sscratch` 寄存器中，这样 `__alltraps` 中才能基于它将 Trap 上下文保存到正确的位置；
- 第 47 行将 sp 修改为 Trap 上下文的位置，后面基于它恢复各通用寄存器和 CSR；
- 第 64 行最后通过 `sret` 指令返回用户态。

```
    # switch to user space
    csrw satp, a1
    sfence.vma
    csrw sscratch, a0
    mv sp, a0
```

- 内核还是应用的地址空间，**跳板的虚拟页均位于同样位置**，且它们也将会映射到同一个实际存放这段汇编代码的物理页帧。也就是说，在执行 `__alltraps` 或 `__restore` 函数进行地址空间切换的时候，应用的用户态虚拟地址空间和操作系统内核的内核态虚拟地址空间对切换地址空间的指令所在页的映射方式均是相同的，这就说明了这段切换地址空间的指令控制流仍是可以连续执行的。
- 不使用call trap_handler而使用jr：**跳转指令实际被执行时的虚拟地址和在编译器/汇编器/链接器进行后端代码生成和链接形成最终机器码时设置此指令的地址是不同的。(跳转指令和 `trap_handler` 都在代码段之内，但是跳转指令实际执行在最高页)**

Trap处理实现

- 准备好 `__restore` 需要两个参数：分别是 Trap 上下文在应用地址空间中的虚拟地址和要继续执行的应用地址空间的 token 。
- 我们把 `stvec` 设置为内核和应用地址空间共享的跳板页面的起始地址 `TRAMPOLINE` 而不是编译器在链接时看到的 `__alltraps` 的地址。这是因为启用分页模式之后，内核只能通过跳板页面上的虚拟地址来实际取得 `__alltraps` 和 `__restore` 的汇编代码。

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

#[no_mangle]
/// return to user space
/// set the new addr of __restore asm function in TRAMPOLINE page,
/// set the reg a0 = trap_cx_ptr, reg a1 = phy addr of usr page table,
/// finally, jump to new addr of __restore asm function
pub fn trap_return() -> ! {
    set_user_trap_entry();
    let trap_cx_ptr = TRAP_CONTEXT_BASE;
    let user_satp = current_user_token();
    extern "C" {
        fn __alltraps();
        fn __restore();
    }
    let restore_va = __restore as usize - __alltraps as usize + TRAMPOLINE;
    // trace!("[kernel] trap_return: ..before return");
    unsafe {
        asm!(
            "fence.i",
            "jr {restore_va}",         // jump to new addr of __restore asm function
            restore_va = in(reg) restore_va,
            in("a0") trap_cx_ptr,      // a0 = virt addr of Trap Context
            in("a1") user_satp,        // a1 = phy addr of usr page table
            options(noreturn)
        );
    }
}
```

- 当每个应用第一次获得 CPU 使用权即将进入用户态执行的时候，它的内核栈顶放置着我们在 内核加载应用的时候 构造的一个任务上下文；在 `__switch` 切换到该应用的任务上下文的时候，内核将会跳转到 `trap_return` 并返回用户态开始该应用的启动执行。

```rust
// os/src/task/context.rs
impl TaskContext {
pub fn goto_trap_return() -> Self {
    Self {
        ra: trap_return as usize,
        s: [0; 12],
        }
    }
}
```

改进的sys_write实现

- 参数中的 `token` 是某个应用地址空间的 token， `ptr` 和 `len` 则分别表示该地址空间中的一段缓冲区的起始地址和长度(注：这个缓冲区的应用虚拟地址范围是连续的)。
- 由于内核和应用地址空间的隔离， `sys_write` 不再能够直接访问位于应用空间中的数据，而需要手动查页表才能知道那些数据被放置在哪些物理页帧上并进行访问。
- 为此，页表模块 `page_table` 提供了将应用地址空间中一个缓冲区转化为在内核空间中能够直接访问的形式的辅助函数：

```rust
/// Translate&Copy a ptr[u8] array with LENGTH len to a mutable u8 Vec through page table
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
}
```