Lab 3: RV64 虚拟内存管理

1 实验目的

- 学习虚拟内存的相关知识，实现物理地址到虚拟地址的切换。
- 了解 RISC-V 架构中 SV39 分页模式，实现虚拟地址到物理地址的映射，并对不同的段进行相应的权限设置。

2 实验环境

- Environment in previous labs

3 背景知识

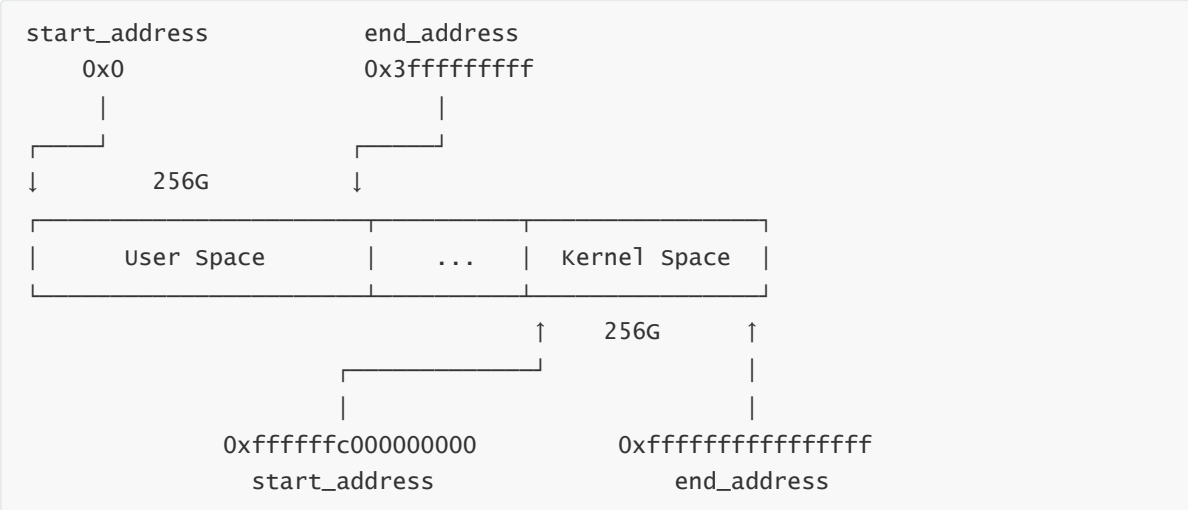
3.0 前言

在 lab2 中我们赋予了 OS 对多个线程调度以及并发执行的能力，由于目前这些线程都是内核线程，因此他们可以共享运行空间，即运行不同线程对空间的修改是相互可见的。但是如果我们需要线程相互**隔离**，以及在多线程的情况下更加**高效**的使用内存，就必须引入`虚拟内存`这个概念。

虚拟内存可以为正在运行的进程提供独立的内存空间，制造一种每个进程的内存都是独立的假象。同时虚拟内存到物理内存的映射也包含了对内存的访问权限，方便 Kernel 完成权限检查。

在本次实验中，我们需要关注 OS 如何**开启虚拟地址**以及通过设置页表来实现**地址映射**和**权限控制**。

3.1 Kernel 的虚拟内存布局



通过上图我们可以看到 RV64 将 `0x0000004000000000` 以下的虚拟空间作为 `user space`。将 `0xffffffc000000000` 及以上的虚拟空间作为 `kernel space`。由于我们还未引入用户态程序，目前我们只需要关注 `kernel space`。

具体的虚拟内存布局可以参考这里。

> 在 `RISC-V Linux Kernel Space` 中有一段区域被称为 `direct mapping area`，为了方便 kernel 可以高效率的访问 RAM，kernel 会预先把所有物理内存都映射至这一块区域 (PA + OFFSET == VA)， 这种映射也被称为 `linear mapping`。在 RISC-V Linux Kernel 中这一段区域为 `0xffffffe000000000 ~ 0xffffffff00000000`, 共 124 GB 。

3.2 RISC-V Virtual-Memory System (Sv39)

3.2.1 `satp` Register（Supervisor Address Translation and Protection Register)

```
 63      60 59                 44 43                                 0
 ------------------------------------------------------------------------
|   MODE    |         ASID         |                PPN                 |
 ------------------------------------------------------------------------
```

- MODE 字段的取值如下图：

```
                          RV 64
     ------------------------------------------------------------
    |  Value   |  Name   |  Description                          |
    |----------------------------------------------------------|
    |    0     | Bare    | No translation or protection          |
    |  1 - 7   | ---     | Reserved for standard use             |
    |    8     | Sv39    | Page-based 39 bit virtual addressing  | <-- 我们使用的mode
    |    9     | Sv48    | Page-based 48 bit virtual addressing  |
    |    10    | Sv57    | Page-based 57 bit virtual addressing  |
    |    11    | Sv64    | Page-based 64 bit virtual addressing  |
    | 12 - 13  | ---     | Reserved for standard use             |
    | 14 - 15  | ---     | Reserved for standard use             |
     ------------------------------------------------------------
```

- ASID (Address Space Identifier)： 此次实验中直接置 0 即可。
- PPN (Physical Page Number)：顶级页表的物理页号。我们的物理页的大小为 4KB， PA >> 12 == PPN。
- 具体介绍请阅读 RISC-V Privileged Spec 4.1.10 。

3.2.2 RISC-V Sv39 Virtual Address and Physical Address

```
     38        30 29        21 20        12 11                          0
     ------------------------------------------------------------------------
    |   VPN[2]    |   VPN[1]    |   VPN[0]    |          page offset          |
     ------------------------------------------------------------------------
                           Sv39 virtual address
```

```
 55                30 29        21 20        12 11                          0
 ------------------------------------------------------------------------------
|       PPN[2]        |   PPN[1]    |   PPN[0]    |          page offset          |
 ------------------------------------------------------------------------------
                           Sv39 physical address
```

- Sv39 模式定义物理地址有 56 位，虚拟地址有 64 位。但是，虚拟地址的 64 位只有低 39 位有效。通过虚拟内存布局图我们可以发现，其 63-39 位为 0 时代表 user space address， 为 1 时 代表 kernel space address。

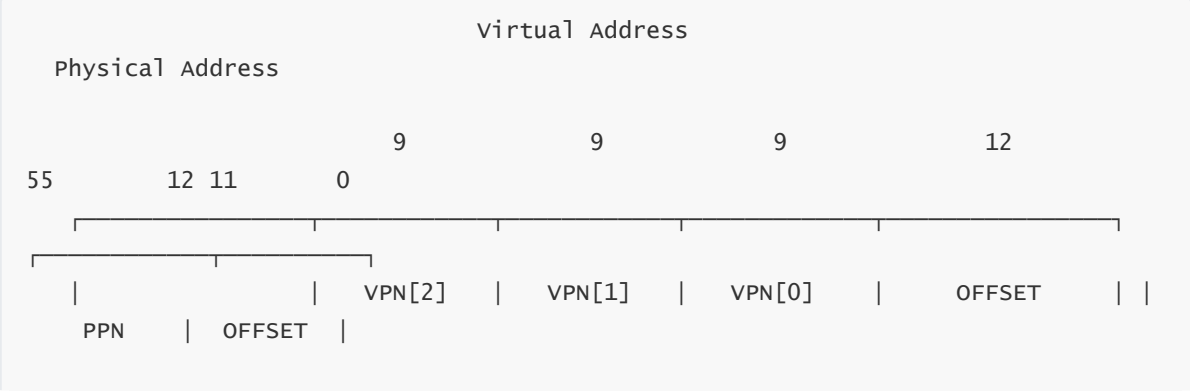
- Sv39 支持三级页表结构，VPN2-0分别代表每级页表的`虚拟页号`，PPN2-0分别代表每级页表的`物理页号`。物理地址和虚拟地址的低12位表示页内偏移（page offset）。
- 具体介绍请阅读 RISC-V Privileged Spec 4.4.1 。

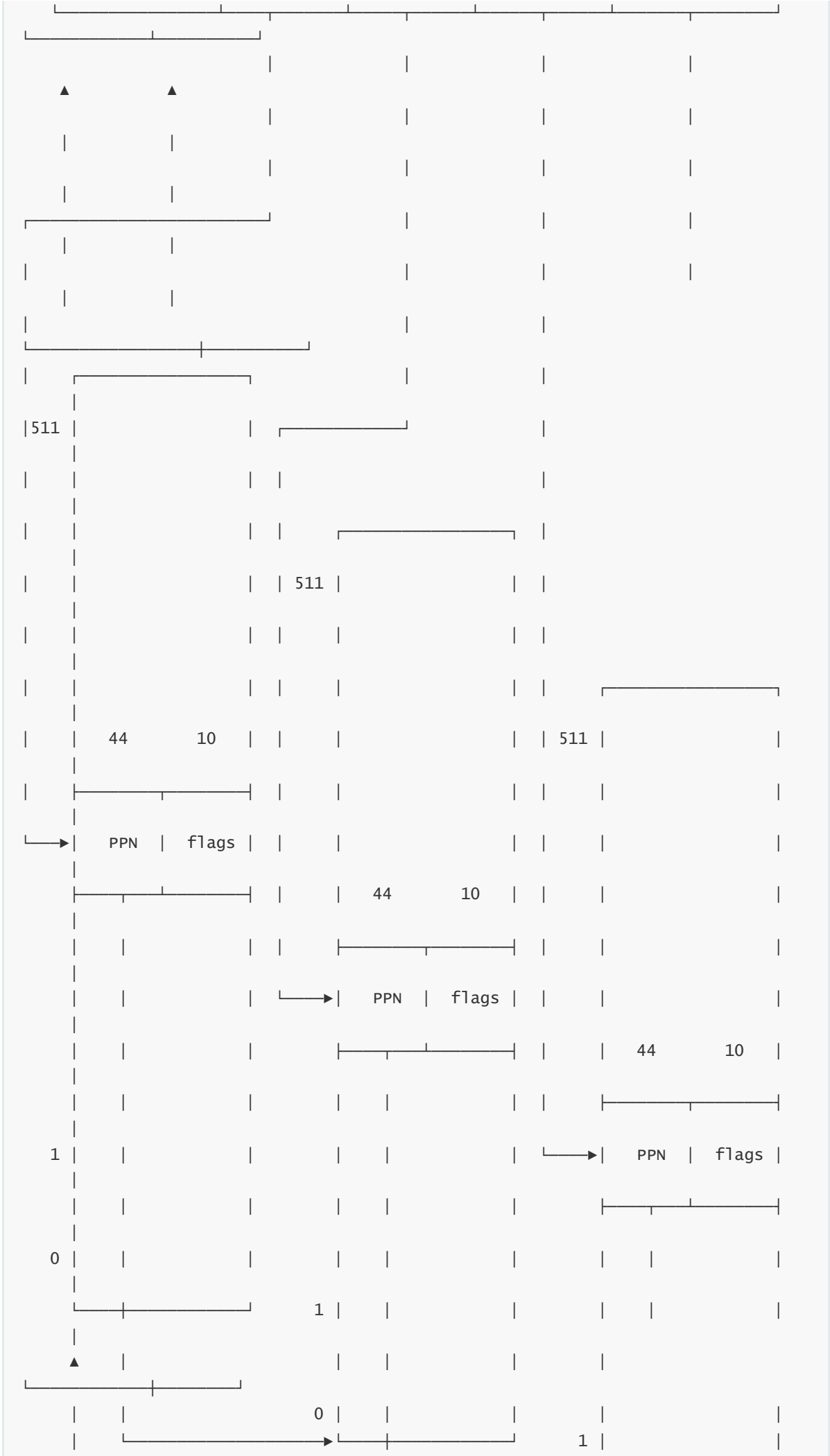
3.2.3 RISC-V Sv39 Page Table Entry

- 0 ~ 9 bit: protection bits
  - V : 有效位，当 V = 0, 访问该 PTE 会产生 Pagefault。
  - R : R = 1 该页可读。
  - W : W = 1 该页可写。
  - X : X = 1 该页可执行。
  - U , G , A , D , RSW 本次实验中设置为 0 即可。
- 具体介绍请阅读 RISC-V Privileged Spec 4.4.1

3.2.4 RISC-V Address Translation

虚拟地址转化为物理地址流程图如下，具体描述见 RISC-V Privileged Spec 4.3.2 :




511
511
511
44
10
PPN
flags
44
10
PPN
flags
44
10
PPN
flags
1
0
1
0
1

4 实验步骤

4.1 准备工程

- 此次实验基于 lab3 同学所实现的代码进行。
- 需要修改 `defs.h`, 在 `defs.h` **添加**如下内容：

```
#define OPENSBI_SIZE (0x200000)

#define VM_START (0xffffffe000000000)
#define VM_END   (0xffffffff00000000)
#define VM_SIZE  (VM_END - VM_START)

#define PA2VA_OFFSET (VM_START - PHY_START)
```

- 从 `repo` 同步以下代码: `vmlinux.lds.S` 并按照以下步骤将这些文件正确放置。

```
.
└── arch
    └── riscv
        └── kernel
            └── vmlinux.lds
```

4.2 开启虚拟内存映射。

在 RISC-V 中开启虚拟地址被分为了两步：`setup_vm` 以及 `setup_vm_final`，下面将介绍相关的具体实现。

4.2.1 `setup_vm` 的实现

- 将 0x80000000 开始的 1GB 区域进行两次映射，其中一次是等值映射 (PA == VA) ，另一次是将其映射至高地址 (PA + PV2VA_OFFSET == VA)。如下图所示：

```
Physical Address
----------------------------------------
                    | OpenSBI | Kernel |
----------------------------------------
                    ^
                0x80000000
                    ├─────────────────────────────────────────────────────┐
                    |                                                     |
Virtual Address     ↓                                                     ↓
--------------------------------------------------------------------------------
------------------
                    | OpenSBI | Kernel |                                  |
OpenSBI | Kernel |
--------------------------------------------------------------------------------
------------------
```

```
                    ^                                                   ^
                0x80000000
0xffffffe000000000
```

- 完成上述映射之后，通过 `relocate` 函数，完成对 `satp` 的设置，以及跳转到对应的虚拟地址。
- 至此我们已经完成了虚拟地址的开启，之后我们运行的代码也都将在虚拟地址上运行。

```
void setup_vm(void) {
    /*
    1. 由于是进行 1GB 的映射 这里不需要使用多级页表
    2. 将 va 的 64bit 作为如下划分： | high bit | 9 bit | 30 bit |
        high bit 可以忽略
        中间9 bit 作为 early_pgtbl 的 index
        低 30 bit 作为 页内偏移 这里注意到 30 = 9 + 9 + 12， 即我们只使用根页表， 根页表的
每个 entry 都对应 1GB 的区域。
    3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    */

    memset(early_pgtbl, 0, PGSIZE);
    memset(swapper_pg_dir, 0, PGSIZE);

    // 第一次映射，等值映射
    int vpn, ppn;
    // 这里是根页表，所以右移 30 位，即 1G 空间
    vpn = (PHY_START >> 30);
    ppn = (PHY_START >> 30) & 0x3ffffff;
    // 将顶级页表的权限位低 4 位设置为 1111
    early_pgtbl[vpn] = (ppn<<28) + 15;

    // 第二次映射，映射到direct mapping area
    vpn = (VM_START >> 30) & 0x1ff;
    early_pgtbl[vpn] = (ppn<<28) + 15;

}
```

```
    .extern start_kernel
    .section .text.init
    .globl _start
_start:
    # la sp, boot_stack_top # set the stack pointer
    lui sp, 0x40104
    slli sp, sp, 1

    jal ra, setup_vm # set up vitual memory (just a transition)
    jal ra, relocate

    jal ra, mm_init # initialize physical memory

    jal ra, setup_vm_final # set up real virtual memory

    jal ra, task_init # initialize task threads

    # set stvec
    la t0, _traps
    csrw stvec, t0
```

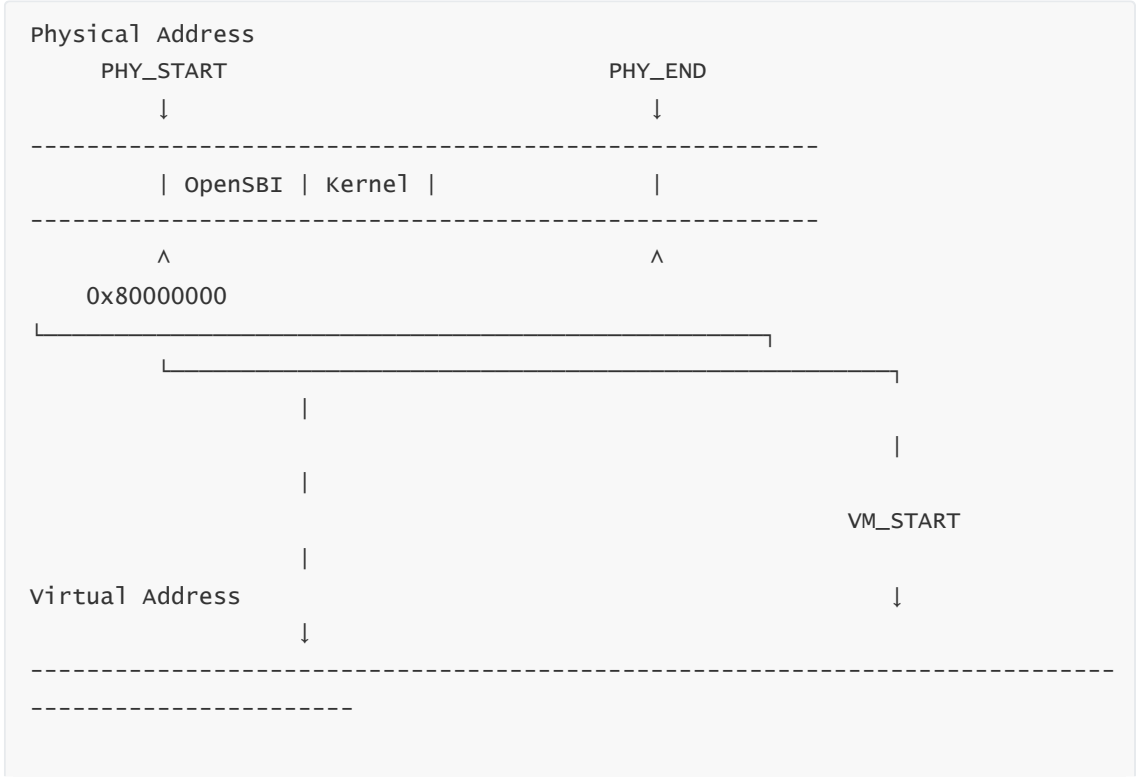
```
    # set on STIE bit of sie
    csrr t0, sie
    ori t0, t0, 0x20
    csrw sie, t0

    # set on SIE bit and SPIE of sstatus
    csrr t0, sstatus
    ori t0, t0, 0x22
    csrw sstatus, t0

    # set first time interrupt
    jal ra, clock_set_next_event

    jal x0, start_kernel  # jump to start_kernel

relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)

    ######################
    #   YOUR CODE HERE   #
    ######################

    # calculate 0xffffffdf80000000 (PA2VA_OFFSET) in t0
    # first turn t0 -> 0x80000000
    addi t0, x0, 1
    slli t0, t0, 31
    # turn t1 -> 0xffffffdf00000000
    lui t1, 0xfffff
    # addi t1, t1, 0xfdf
    addi t1, t1, 0x7ef
    addi t1, t1, 0x7f0
    slli t1, t1, 31
    slli t1, t1, 1
    # t0 = 0xffffffdf00000000 + 0x80000000
    add t0, t0, t1

    add ra, ra, t0 # set ra = ra + PA2VA_OFFSET
    add sp, sp, t0 # set sp = sp + PA2VA_OFFSET

    # set satp with early_pgtbl
    la t2, early_pgtbl
    sub t2, t2, t0
    add t1, x0, t2
    srli t1, t1, 12
    # MODE field = 8
    addi t0, x0, 8
    slli t0, t0, 20
    slli t0, t0, 20
    slli t0, t0, 20
    or t1, t1, t0
    csrw satp, t1

    ######################
    #   YOUR CODE HERE   #
```

```
    ######################

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096 * 4

    .globl boot_stack_top
boot_stack_top:
```

> Hint 1: `sfence.vma` 指令用于刷新 TLB
>
> Hint 2: `fence.i` 指令用于刷新 icache
>
> Hint 3: 在 set satp 前，我们只可以使用**物理地址**来打断点。设置 satp 之后，才可以使用虚拟地址打断点，同时之前设置的物理地址断点也会失效，需要删除

4.2.2 `setup_vm_final` 的实现

- 由于 setup_vm_final 中需要申请页面的接口， 应该在其之前完成内存管理初始化， 可能需要修改 mm.c 中的代码，mm.c 中初始化的函数接收的起始结束地址需要调整为虚拟地址。
- 对所有物理内存 (128M) 进行映射，并设置正确的权限。



```
                                                                    | OpenSBI |
Kernel |               |
------------------------------------------------------------------------------
------------------------
                                                                        ^
                                                                0xffffffe000000000
```

- 不再需要进行等值映射
- 不再需要将 OpenSBI 的映射至高地址，因为 OpenSBI 运行在 M 态， 直接使用的物理地址。
- 采用三级页表映射。
- 在 head.S 中 适当的位置调用 setup_vm_final 。

```
/* swapper_pg_dir: kernel pagetable 根目录， 在 setup_vm_final 进行映射。 */
unsigned long swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));

extern uint64 _stext;
extern uint64 _srodata;
extern uint64 _sdata;

void setup_vm_final(void) {
    memset(swapper_pg_dir, 0x0, PGSIZE);

    // mapping kernel text X|-|R|V
    create_mapping((uint64*)swapper_pg_dir, (uint64)&_stext, (uint64)(&_stext) -
PA2VA_OFFSET, 2U, 5);

    // mapping kernel rodata -|-|R|V
    create_mapping((uint64*)swapper_pg_dir, (uint64)&_srodata, (uint64)
(&_srodata) - PA2VA_OFFSET, 1U, 1);

    // mapping other memory -|W|R|V
    create_mapping((uint64*)swapper_pg_dir, (uint64)&_sdata, (uint64)(&_sdata) -
PA2VA_OFFSET, 32766U, 3);

    // set satp with swapper_pg_dir
    asm volatile("csrw satp, %[base]":: [base] "r" ((uint64)swapper_pg_dir):);

    // flush TLB
    asm volatile("sfence.vma zero, zero");

    // flush icache
    asm volatile("fence.i");

    return;
}

/* 创建多级页表映射关系 */
/*
    pgtbl 为根页表的基地址
    va, pa 为需要映射的虚拟地址、物理地址
    sz 为映射的大小 (单位为4KB)
    perm 为映射的读写权限
*/
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm) {
```

```
    /*
    pgtbl 为根页表的基地址
    va, pa 为需要映射的虚拟地址、物理地址
    sz 为映射的大小
    perm 为映射的读写权限

    创建多级页表的时候可以使用 kalloc() 来获取一页作为页表目录
    可以使用 V bit 来判断页表项是否存在
    */

    uint64 *tbl[2]; // 存储一、二级页表
    uint64 PTE[2]; // 一、二级的页表项
    uint64 *page_ = NULL; // 存储新分配页的地址
    int v1, v2, v3; // 存储虚拟地址的高，中，低 9 位
    uint64 offset = 0;


    for(uint64 va_ = va; va_ < sz + va; va_ += PGSIZE) {

        // 根据 Sv39 的虚拟地址映射规则，虚拟地址去掉 12 位偏移量后，高，中，低 9 位为三级页表的索引
        v1 = (va_ >> 30) & 0x1ff;
        v2 = (va_ >> 21) & 0x1ff;
        v3 = (va_ >> 12) & 0x1ff;

        // 填充第一级页表
        PTE[1] = pgtbl[v1];

        if((PTE[1] & 1) == 0) {
            page_ = (uint64*)kalloc();

            PTE[1] = ((((uint64)page_ - PA2VA_OFFSET) >> 12) << 10) + 1;
            pgtbl[v1] = PTE[1];
        }

        // 创建并填充第二级页表
        tbl[1] = (uint64*)((PTE[1] >> 10) << 12);

        PTE[0] = tbl[1][v2];
        if((PTE[0] & 1) == 0) {
            page_ = (uint64*)kalloc();
            PTE[0] = ((((uint64)page_ - PA2VA_OFFSET) >> 12) << 10) + 1;
            tbl[1][v2] = PTE[0];
        }

        // 填充第三级页表
        tbl[0] = (uint64*)((PTE[0] >> 10) << 12);

        // 添加页表的权限，直接放置在低 8 位
        tbl[0][v3] = (((pa + offset) >> 12) << 10) + perm;
        offset += PGSIZE;

    }

    return;
}
```

4.3 编译及测试

编译结果

```
...setup_vm done!
...mm_init done!
...proc_init done!

switch to [PID = 13 COUNTER = 1]
[PID = 13] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007ff0000

switch to [PID = 16 COUNTER = 1]
[PID = 16] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007fed000

switch to [PID = 17 COUNTER = 1]
[PID = 17] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007fec000

switch to [PID = 27 COUNTER = 1]
[PID = 27] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007fe2000

switch to [PID = 12 COUNTER = 2]
[PID = 12] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007ff1000
[PID = 12] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007ff1000

switch to [PID = 28 COUNTER = 2]
[PID = 28] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007fe1000
[PID = 28] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007fe1000

switch to [PID = 1 COUNTER = 3]
[PID = 1] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007ffc000
[PID = 1] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007ffc000
[PID = 1] is running. auto_inc_local_var = 3. Physical address at 0xffffffe007ffc000

switch to [PID = 2 COUNTER = 3]
[PID = 2] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007ffb000
[PID = 2] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007ffb000
[PID = 2] is running. auto_inc_local_var = 3. Physical address at 0xffffffe007ffb000

[PID = 9] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007ff4000
[PID = 9] is running. auto_inc_local_var = 3. Physical address at 0xffffffe007ff4000

switch to [PID = 14 COUNTER = 3]
[PID = 14] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007fef000
[PID = 14] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007fef000
[PID = 14] is running. auto_inc_local_var = 3. Physical address at 0xffffffe007fef000

switch to [PID = 11 COUNTER = 4]
[PID = 11] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007ff2000
[PID = 11] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007ff2000
[PID = 11] is running. auto_inc_local_var = 3. Physical address at 0xffffffe007ff2000
[PID = 11] is running. auto_inc_local_var = 4. Physical address at 0xffffffe007ff2000

switch to [PID = 29 COUNTER = 4]
[PID = 29] is running. auto_inc_local_var = 1. Physical address at 0xffffffe007fe0000
[PID = 29] is running. auto_inc_local_var = 2. Physical address at 0xffffffe007fe0000
[PID = 29] is running. auto_inc_local_var = 3. Physical address at 0xffffffe007fe0000
[PID = 29] is running. auto_inc_local_var = 4. Physical address at 0xffffffe007fe0000
```

思考题

1. 验证 `.text`, `.rodata` 段的属性是否成功设置，给出截图。

   编译后内核正确启动并执行，则权限设置成功，`text` 段具有执行属性。

   在main.c中添加如下代码：

```c
extern char _stext[];
extern char _sdata[];
extern char _srodata[];

int start_kernel() {
    printk("2023 Hello RISC-V\n");

    printk("_stext = %ld\n", *_stext);
    printk("_sdata = %ld\n", *_sdata);
    printk("_srodata = %ld\n", *_srodata);

    *_stext = 0;
    *_sdata = 0;

    test(); // DO NOT DELETE !!!

    return 0;
}
```

输出结果如下，则说明设置成功。

```
...setup_vm done!
...mm_init done!
...proc_init done!
2023 Hello RISC-V
_stext = 55
_sdata = 128
_srodata = 46
idle process is running!
```

2. 为什么我们在 `setup_vm` 中需要做等值映射?

   因为在函数 `setup_vm_final` 中，程序使用了页表项中的物理页号，将其转换为物理地址，用该物理地址来访问下一级页表。如果没有等值映射，将会导致直接的物理地址内存访问错误。

3. 在 Linux 中，是不需要做等值映射的。请探索一下不在 `setup_vm` 中做等值映射的方法。

   可以将 `setup_vm_final` 函数中读取得到的页号的物理地址转换为虚拟地址，供下次访问使用。

```c
void setup_vm(void) {
/*
1. 由于是进行 1GB 的映射 这里不需要使用多级页表
2. 将 va 的 64bit 作为如下划分： | high bit | 9 bit | 30 bit |
high bit 可以忽略
中间9 bit 作为 early_pgtbl 的 index
低 30 bit 作为 页内偏移 这里注意到 30 = 9 + 9 + 12， 即我们只使用根页表， 根
页表的每个 entry 都对应 1GB 的区域。
3. Page Table Entry 的权限 V | R | W | X 位设置为 1
*/
```

```
    memset(early_pgtbl, 0, PGSIZE);
    memset(swapper_pg_dir, 0, PGSIZE);
    int vpn, ppn;
    // 直接映射到direct mapping area
    vpn = (VM_START >> 30) & 0x1ff;
    ppn = (PHY_START >> 30) & 0x3ffffff;
    early_pgtbl[vpn] = (ppn<<28) + 15;
}
```