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 实验步骤

3.1 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`, `Makefile`。并按照以下步骤将这些文件正确放置。

```
.
└── arch
    └── riscv
        └── kernel
            ├── Makefile
            └── vmlinux.lds.S
```

  这里我们通过 `vmlinux.lds.S` 模版生成 `vmlinux.lds` 文件。链接脚本中的 `ramv` 代表 `VMA ( Virtual Memory Address )` 即虚拟地址， `ram` 则代表 `LMA ( Load Memory Address )`, 即我们 OS image 被 load 的地址，可以理解为物理地址。使用以上的 vmlinux.lds 进行编译之后，得到的 `System.map` 以及 `vmlinux` 采用的都是虚拟地址，方便之后 Debug。

- 从本实验开始我们需要使用刷新缓存的指令扩展，并自动在编译项目前执行 `clean` 任务来防止对头文件的修改无法触发编译任务。在项目顶层目录的 `Makefile` 中需要做如下更改：

```
# Makefile
...
ISA=rv64imafd_zifencei
...
all: clean
    ${MAKE} -C lib all
    ${MAKE} -C test all
    ${MAKE} -C init all
    ${MAKE} -C arch/riscv all
    @echo -e '\n'Build Finished OK
...
```

3.2 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。

3.2.1 `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
```

- `RISC-V Vitrual Memory System(Sy39)` 中的相关存储结构定义如下所示
  - 物理地址`pa`：
    - `PPN[2]-PPN[0]`代表每级页表的物理页号，`page offset`代表在页内的偏移

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

- 虚拟地址`va`：
  - `VPN[2]-VPN[0]`代表每级页表的物理页号，`page offset`代表在页内的偏移

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

```

- 页表项`Page Table Entry(PTE)`：
  - 其中 9-0 位为`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。

```
 63      54 53        28 27        19 18        10 9   8 7 6 5 4 3 2 1 0
 ------------------------------------------------------------------------
| Reserved |   PPN[2]   |   PPN[1]   |   PPN[0]   | RSW |D|A|G|U|X|W|R|V|
 ------------------------------------------------------------------------
                                                    |   | | | | | | | |
                                                    |   | | | | | | `---- V - Valid
                                                    |   | | | | | `------ R - Readable
                                                    |   | | | | `-------- W - Writable
                                                    |   | | | `---------- X - Executable
                                                    |   | | `------------ U - User
```

```
11                                                  |   | | `-----
    --------- G - Global
12                                                  |   | `-------
    --------- A - Accessed
13                                                  |   `---------
    --------- D - Dirty (0 in page directory)
14                                                  `-------------
    --------- Reserved for supervisor software
```

基于上述存储结构，我们可以通过以下方式实现等值映射和高位映射

1. 从虚拟地址 `va` 中获取 `vpn = (va >> 30) & 0x1ff;`
2. 从物理地址 `pa` 中获取 `ppn = (pa >> 12)`，因为物理页大小为4KB，所以 `ppn = pa >> 12`
3. 为页表项赋值 `early_pgtbl[vpn] = (ppn<<10) | 0xf`, 将页表项 低4位（`V | R | W | X`位）置 `1`

补全后的 `setup_vm()` 函数逻辑如下：

```c
// arch/riscv/kernel/vm.c

/* early_pgtbl: 用于 setup_vm 进行 1GB 的 映射。 */
unsigned long  early_pgtbl[512] __attribute__((__aligned__(0x1000)));

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);
    uint64 va, pa; //1 GB = 2^30 B
    va = PHY_START;
    pa = PHY_START; //0x80000000
    uint64 vpn = (va >> 30) & 0x1ff; //38-30
    uint64 ppn = (pa >> 12);
    // printk("vpn :%ld\n", vpn);
    early_pgtbl[vpn] = (ppn << 10 | 0xf);
    // early_pgtbl[vpn] = 536870927; // V R W X set to 1
    // printk("tbl_vpn: %lx\n",  early_pgtbl[vpn]);

    va = VM_START;
    vpn = (va >> 30) & 0x1ff;
    early_pgtbl[vpn] = (ppn << 10 | 0xf);
    // printk("vpn :%ld\n", vpn);
    // printk("vpn: %lx\n", vpn);

    // printk("end_list\n");
    // Empty();
}
```

- 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。

- 在sv39中，`satp`寄存器的结构如下图所示

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

  - 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。
  - MODE：value为8时，代表采用Page-based 39 bit virtual addressing，即我们所需要的模式

所以我们仅仅需要的`satp`寄存器的`MODE`字段设置为8，并将之前设置好的顶级页表`early_pgtbl`的物理页号填入`PPN`字段即可。

但是在实践过程中，发现了问题，在于vmlinux.lds中已经进行了虚拟内存的映射，所以生成的符号表中的地址都是虚拟地址，但是在relocate之前，我们在运行时需要的是物理地址，因此虚拟地址需要减掉PA2VA_OFFSET来得到实际的物理地址

> 参考：较新的工具链编译出来的会使用 GOT 表来进行寻址，而根据链接脚本，GOT 表内的项肯定都是虚拟地址，这样在启用虚拟地址之前直接寻址就会出现问题

补全后的head.S如下图所示：

```
#include "defs.h"

.extern early_pgtbl
.extern start_kernel

    .section .text.init
    .globl _start
_start:
    li t0, PA2VA_OFFSET
    la sp, boot_stack_top
    sub sp, sp, t0

    call setup_vm
    call relocate

    #---

    call mm_init  #初始化内存管理
    call setup_vm_final

    #---

    call task_init
    # set stvec
    la t2, _traps
    csrw stvec, t2

    #---

    #set sie
    li t5, 0x20
```

```
    csrr t3, sie
    or t3, t3, t5
    csrw sie, t3

    #---

    # set interrupt
    call clock_set_next_event

    #----

    # set sstatus
    li t6, 0x2
    csrr t3, sstatus
    or t3, t3, t6
    csrw sstatus, t3

    #---

    jal start_kernel

relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    li t5, PA2VA_OFFSET
    add ra, ra, t5
    add sp, sp, t5

    # set satp with early_pgtbl

    li t0, 8
    slli t0, t0, 60 # Mode = 8
    la t1, early_pgtbl
    sub t1, t1, t5
    srli t1, t1, 12
    or t0, t0, t1

    csrw satp, t0

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret

    #---
    .section .bss.stack
    .globl boot_stack


boot_stack:
    .space 4096 # <-- change to your stack size

```

```
    .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

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。

3.2.2 `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。

  具体而言，需要修改 `mm_init` 函数的结束地址为虚拟地址

```
void mm_init(void) {
    //结束地址更改为虚拟地址
    kfreerange(_ekernel, (uint64)(PHY_END + PA2VA_OFFSET));
    printk("...mm_init done!\n");
}
```

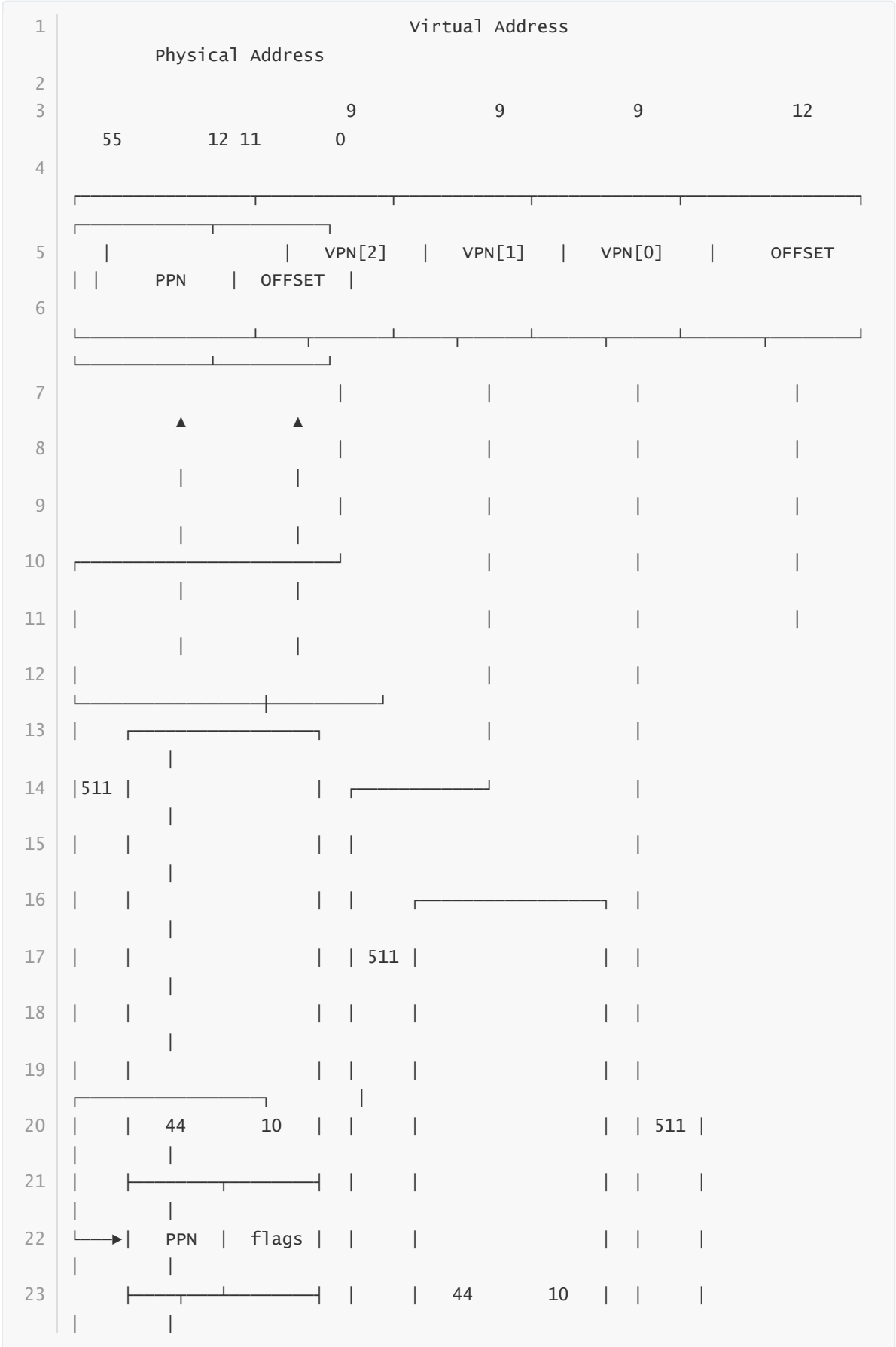
- 对 所有物理内存 (128M) 进行映射，并设置正确的权限。

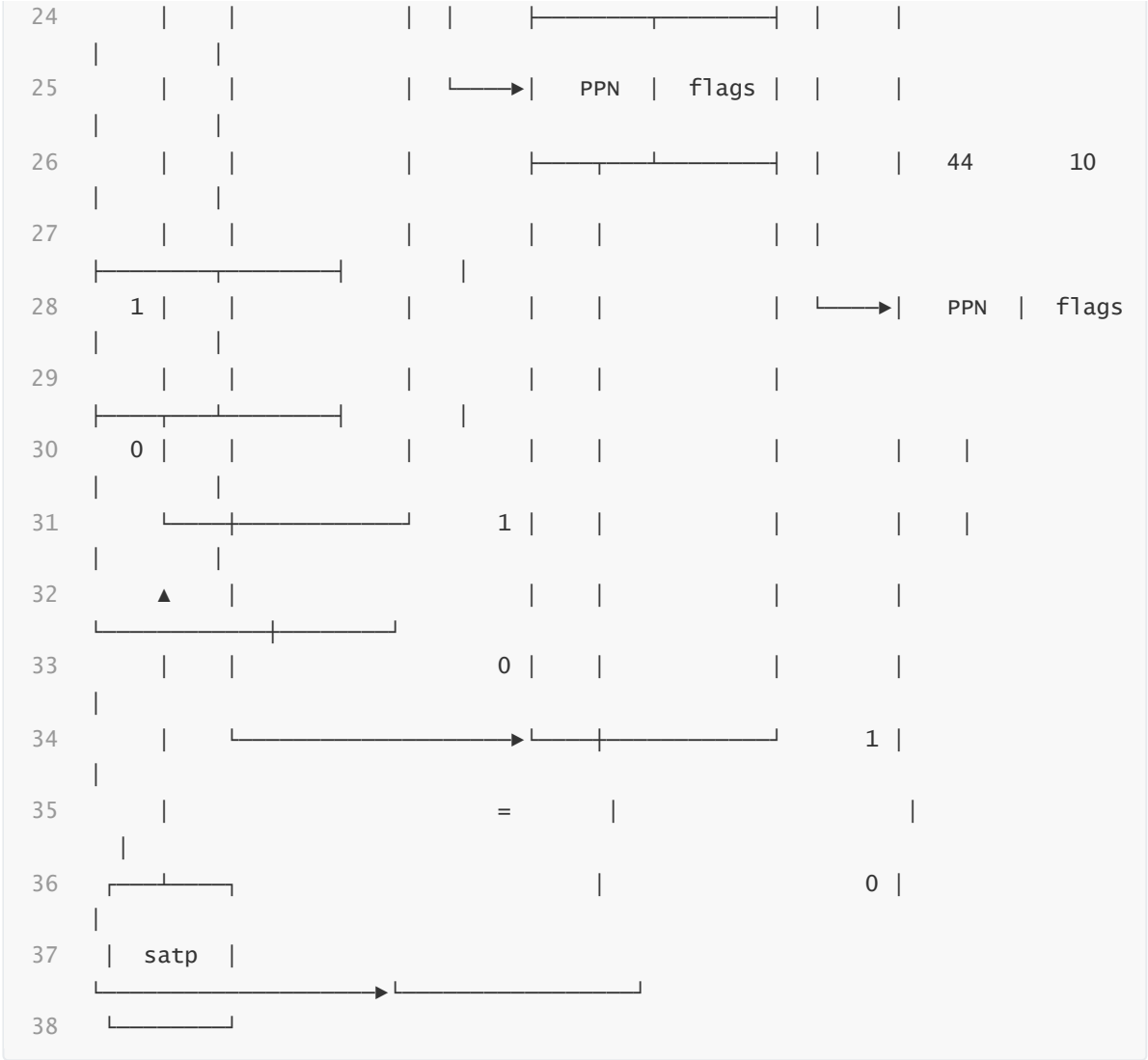
```
Physical Address
     PHY_START                              PHY_END
         ↓                                     ↓
-----------------------------------------------------------
         | OpenSBI | Kernel |                  |
-----------------------------------------------------------
         ^                                     ^
    0x80000000
└──────────────────────────────────────────────────────┐
         └──────────────────────────────────────────────────────┐
                        |                                       |
                        |                                   VM_START
                        |                                       ↓
Virtual Address         ↓
-------------------------------------------------------------------------------
                                                        | 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 。
- 请不要修改 create_mapping 的函数声明，并注意阅读下方对参数的描述。该函数会被用于测试实验的正确性。

- 虚拟地址转化为物理地址的流程图如下





根据RISC-V的address translation过程，我们可以得到如下映射过程

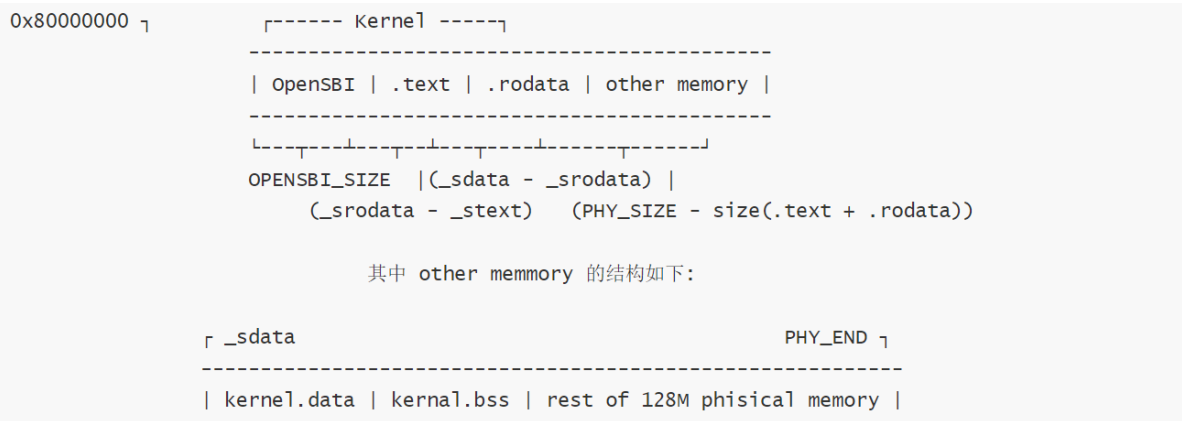
- 顶级页表`tbl[2]`就是之前的`early_pgtbl`，从`va`中获取二级页表的虚拟页号`VPN[2] = (va >> 30) & 0x1ff`，再从`tbl[2]`中获取页表项`PTE = tbl[2][VPN[2]]`
- 根据PTE的有效位`V`判断PTE是否有效，若无效则进行新的页分配，并重新赋值`PTE = (uint64)(((((uint64)page - (uint64)PA2VA_OFFSET) >> 12) << 10 ) | 1)`
- 二级页表`tbl[1] = (uint64 *)(tbl[2][VPN[2]] >> 10 << 12)`，然后从`va`中获取`VPN[1] = (va >> 21) & 0x1ff`，从`tbl[1]`中获取PTE`tbl[1][VPN[1]]`
- 同上进行判断，并获取一级页表`tbl[0] = (uint64 *)((tbl[1][VPN[1]] >> 10) << 12)`，获取`VPN[0]`后直接根据传入的perm和pa进行赋值`tbl[0][VPN[0]] = ((pa >> 12) << 10) | (perm & 0xf);`

所以实现的`create_mapping()`函数如下图所示：

```
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, uint64 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[3] = {NULL, NULL, NULL};
    uint64 VPN[3] = {0,0,0};
    uint64 end = va + sz;

    while(va < end)
    {
        //Process the tbl[2]
        tbl[2] = pgtbl;
        VPN[2] = (va >> 30) & 0x1ff; //38-30
        if(!(tbl[2][VPN[2]]& 0x1)){
            uint64 * page = (uint64 *)kalloc();
            tbl[2][VPN[2]] = (uint64)(((((uint64)page -
(uint64)PA2VA_OFFSET) >> 12) << 10 ) | 1); //PA >> 12 = PPN
        }
        //Process the tbl[1]
        tbl[1] = (uint64 *)(tbl[2][VPN[2]] >> 10 << 12);
        VPN[1] =    ; //29-21
        if(!(tbl[1][VPN[1]] & 0x1)){
            uint64 * page = (uint64 *)kalloc();
            tbl[1][VPN[1]] = (uint64)(((((uint64)page -
(uint64)PA2VA_OFFSET) >> 12) << 10 ) | 1); //PA >> 12 = PPN
        }
        //Process the tbl[0]
        tbl[0] = (uint64 *)((tbl[1][VPN[1]] >> 10) << 12);
        VPN[0] = (va >> 12) & 0x1ff;//20-12
        tbl[0][VPN[0]] = ((pa >> 12) << 10) | (perm & 0xf);

        va += PGSIZE;
        pa += PGSIZE;
    }
}
```

而在`setup_vm_final`函数中，我们需要进行三块程序段的映射，因为不同程序段之间的访问权限是不同的，所以我们需要分别调用`create_mapping`进行重映射，`kernel`的内存结构如下图所示



在进行重映射之后，我们需要用`swapper_pg_dir`再次设置`satp`，由于已经进行了`relocate`，此时代码段执行在虚拟内存上面，因此我们需要减去`PA2VA_OFFSET`得到顶级页表的物理地址，再去赋值`PPN`

- 完整的`setup_vm_final`的实现如下所示：

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

```
    // No OpenSBI mapping required
    uint64 pa, va;
    // mapping kernel text X|-|R|V
    //X|-|R|V: 1011 -> 11
    pa = PHY_START + OPENSBI_SIZE;
    va = VM_START + OPENSBI_SIZE;
    uint64 text_size = (uint64)_srodata - (uint64)_stext;
    // printk("Hello1\n");
    create_mapping(swapper_pg_dir, va, pa, text_size, 11);

    // mapping kernel rodata -|-|R|V
    // -|-|R|V: 0011 -> 3
    uint64 rodata_size = (uint64) _sdata - (uint64)_srodata;
    pa += text_size;
    va += text_size;
    create_mapping(swapper_pg_dir, va, pa, rodata_size, 3);

    // mapping other memory -|W|R|V
    //-|W|R|V: 0111 -> 7
    uint64 memory_size = PHY_SIZE - ((uint64)_sdata - (uint64)_stext);
    pa += rodata_size;
    va += rodata_size;
    create_mapping(swapper_pg_dir, va, pa, memory_size, 7);

    // set satp with swapper_pg_dir

    //pa = va - PA2VA_OFFSET
    uint64 swap_dir_pa = (uint64)swapper_pg_dir - (uint64)PA2VA_OFFSET;
    __asm__ volatile(
        "li t0, 8\n"
        "slli t0, t0, 60\n"
        "mv t1, %[addr]\n"
        "srli t1, t1, 12\n"
        "or t0, t0, t1\n"
        "csrw satp, t0"
        :
        : [addr] "r" (swap_dir_pa)
        : "memory"
    );

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

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

3.3 编译及测试

- 由于加入了一些新的 .c 文件，可能需要修改一些Makefile文件，请同学自己尝试修改，使项目可以编译并运行。
- 结果输出如下：

```
OpenSBI v0.9
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : timer,mfdeleg
Platform HART Count       : 1
Firmware Base             : 0x80000000
Firmware Size             : 100 KB
Runtime SBI Version       : 0.2

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x0000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff
(R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...mm_init done!
...proc_init done!
2022 Hello RISC-V
[S] Supervisor Mode Timer Interrupt
[PID = 8] is running. thread space begin at 0xffffffe007fb6000
[S] Supervisor Mode Timer Interrupt
[PID = 7] is running. thread space begin at 0xffffffe007fb7000
[S] Supervisor Mode Timer Interrupt
[PID = 7] is running. thread space begin at 0xffffffe007fb7000
```

```
[S] Supervisor Mode Timer Interrupt
[PID = 14] is running. thread space begin at 0xffffffe007fb0000
[S] Supervisor Mode Timer Interrupt
[PID = 14] is running. thread space begin at 0xffffffe007fb0000
[S] Supervisor Mode Timer Interrupt
[PID = 11] is running. thread space begin at 0xffffffe007fb3000
[S] Supervisor Mode Timer Interrupt
[PID = 11] is running. thread space begin at 0xffffffe007fb3000
[S] Supervisor Mode Timer Interrupt
[PID = 11] is running. thread space begin at 0xffffffe007fb3000
[S] Supervisor Mode Timer Interrupt
[PID = 13] is running. thread space begin at 0xffffffe007fb1000
[S] Supervisor Mode Timer Interrupt
[PID = 13] is running. thread space begin at 0xffffffe007fb1000
[S] Supervisor Mode Timer Interrupt
[PID = 13] is running. thread space begin at 0xffffffe007fb1000
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
[S] Supervisor Mode Timer Interrupt
[PID = 12] is running. thread space begin at 0xffffffe007fb2000
[S] Supervisor Mode Timer Interrupt
[PID = 12] is running. thread space begin at 0xffffffe007fb2000
[S] Supervisor Mode Timer Interrupt
[PID = 12] is running. thread space begin at 0xffffffe007fb2000
[S] Supervisor Mode Timer Interrupt
[PID = 12] is running. thread space begin at 0xffffffe007fb2000
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running. thread space begin at 0xffffffe007fba000
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running. thread space begin at 0xffffffe007fba000
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running. thread space begin at 0xffffffe007fba000
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running. thread space begin at 0xffffffe007fba000
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running. thread space begin at 0xffffffe007fba000
[S] Supervisor Mode Timer Interrupt
[PID = 15] is running. thread space begin at 0xffffffe007faf000
[S] Supervisor Mode Timer Interrupt
[PID = 15] is running. thread space begin at 0xffffffe007faf000
[S] Supervisor Mode Timer Interrupt
[PID = 15] is running. thread space begin at 0xffffffe007faf000
[S] Supervisor Mode Timer Interrupt
[PID = 15] is running. thread space begin at 0xffffffe007faf000
[S] Supervisor Mode Timer Interrupt
[PID = 15] is running. thread space begin at 0xffffffe007faf000
[S] Supervisor Mode Timer Interrupt
[PID = 15] is running. thread space begin at 0xffffffe007faf000
[S] Supervisor Mode Timer Interrupt
```

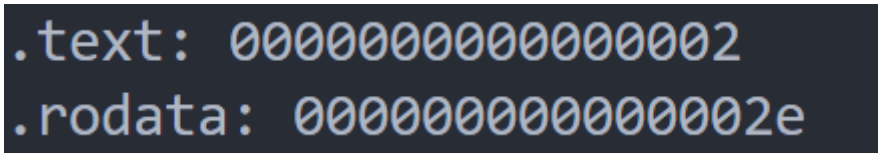
```
[PID = 9] is running. thread space begin at 0xffffffe007fb5000
[S] Supervisor Mode Timer Interrupt
[PID = 9] is running. thread space begin at 0xffffffe007fb5000
[S] Supervisor Mode Timer Interrupt
[PID = 9] is running. thread space begin at 0xffffffe007fb5000
[S] Supervisor Mode Timer Interrupt
[PID = 9] is running. thread space begin at 0xffffffe007fb5000
[S] Supervisor Mode Timer Interrupt
[PID = 9] is running. thread space begin at 0xffffffe007fb5000
[S] Supervisor Mode Timer Interrupt
[PID = 9] is running. thread space begin at 0xffffffe007fb5000
```

4 思考题

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

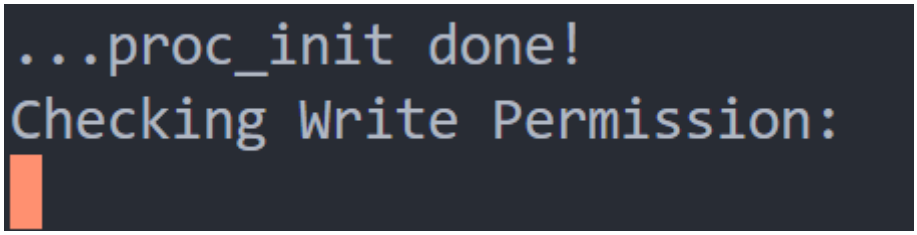
- 根据上述设置，我们知道`.text`段应该具有可执行，可读权限，`.rodata`段具有可读权限、
  - 由于编译后程序刚开始在`.text`段上正常运行，显然`.text`段具有可执行权限
  - 验证两段代码的可写权限，我们尝试打印其中的数据内容

```
printk(".text: %lx\n", *(_stext + 1));
printk(".rodata: %lx\n", *(_srodata + 1));
```



  - 验证两个程序段的可写权限，我们尝试修改数据后再次打印，可以发现程序在此卡住，说明我们不具有写权限

```
 printk("Checking Write Permission:\n");
    *(_stext) = 0x35;
    printk(".text: %lx\n", *(_stext));
    *(_stext) = 0x9b;

printk("Checking Write Permission:\n");
    *(_srodata) = 0x36;
    printk(".rodata: %lx\n", *(_srodata));
    *(_srodata) = 0x2e;
```



2. 为什么我们在 `setup_vm` 中需要做等值映射?

对于CPU来说，CPU发出的指令都是虚拟地址，都需要基于`satp`寄存器中的顶级页表转换为实际的物理地址。 在使用`create_mapping`创建三级页表的时候，我们需要读取`PTE`存储的实际物理页号`PPN`，但此时CPU会认为它是一个虚拟地址，所以需要额外的等值映射来进行物理地址的转换。

- 例如，我们有一个地址`(va)0x0000000080200dfc`，在`setup_vm`两次映射后，我们得到`(va)0xffffffe000200dfc = (va)0x0000000080200dfc = (pa)0x0000000080200dfc`，当我们拿到`tbl[2][VPN[2]]`的时候，会得到`0x0000000080200dfc`，此时如果没有等值映射，那么是找不到对应虚拟地址的值的。因此需要进行一次等值映射

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。

从思考题2中，我们可以发现，需要等值映射的关键之处在于需要为低位的虚拟地址做一个等值的物理地址的转换，那么很简单的方法就是在拿到这个和物理地址值相等的虚拟地址的时候，加上`PA2VA_OFFSET`，把它变成应为的虚拟地址即可。

具体实现如下：

- 在`setup_vm`中，去除等值映射的部分

```
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);
    uint64 va, pa; //1 GB = 2^30 B
    va = PHY_START;
    pa = PHY_START; //0x80000000
    uint64 vpn = (va >> 30) & 0x1ff; //38-30
    uint64 ppn = (pa >> 12);
    //去除等值映射
    // early_pgtbl[vpn] = (ppn << 10 | 0xf);

    va = VM_START;
    vpn = (va >> 30) & 0x1ff;
    early_pgtbl[vpn] = (ppn << 10 | 0xf);
    // printk("vpn :%ld\n", vpn);
    // printk("vpn: %lx\n", vpn);

    // printk("end_list\n");
    // Empty();
}

```

- 在`creating_mapping`中，对两个页表项拿到的虚拟地址做一个偏移

```
tbl[1] = (uint64 *)(((tbl[2][VPN[2]] >> 10) << 12) + (uint64)PA2VA_OFFSET);
...
tbl[0] = (uint64 *)(((tbl[1][VPN[1]] >> 10) << 12) + (uint64)PA2VA_OFFSET);
```

经测试，结果和去掉等值映射前相同

5 心得体会

在本次实验中，我学习到了虚拟内存的相关知识，并学会了虚拟内存到物理内存之间的转换，实现了RiscV架构下sv39的三级页表虚拟寻址，并成功去除了等值映射。

在实现的过程中，遇到了C语言运算符优先级的坑，导致检查了很长时间逻辑，最后发现逻辑并没有什么问题。不好评价(x)，以及遇到了一些类型强制转换的问题，在实现三级页表的过程中并无遇到太大问题。当然跑通了还是挺有成就感的。