Lab 4: 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

- Docker in Lab0

3 背景知识

3.0 前言

在 lab3 中我们赋予了 OS 对多个线程调度及并发执行的能力，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，我们必须引入`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**。

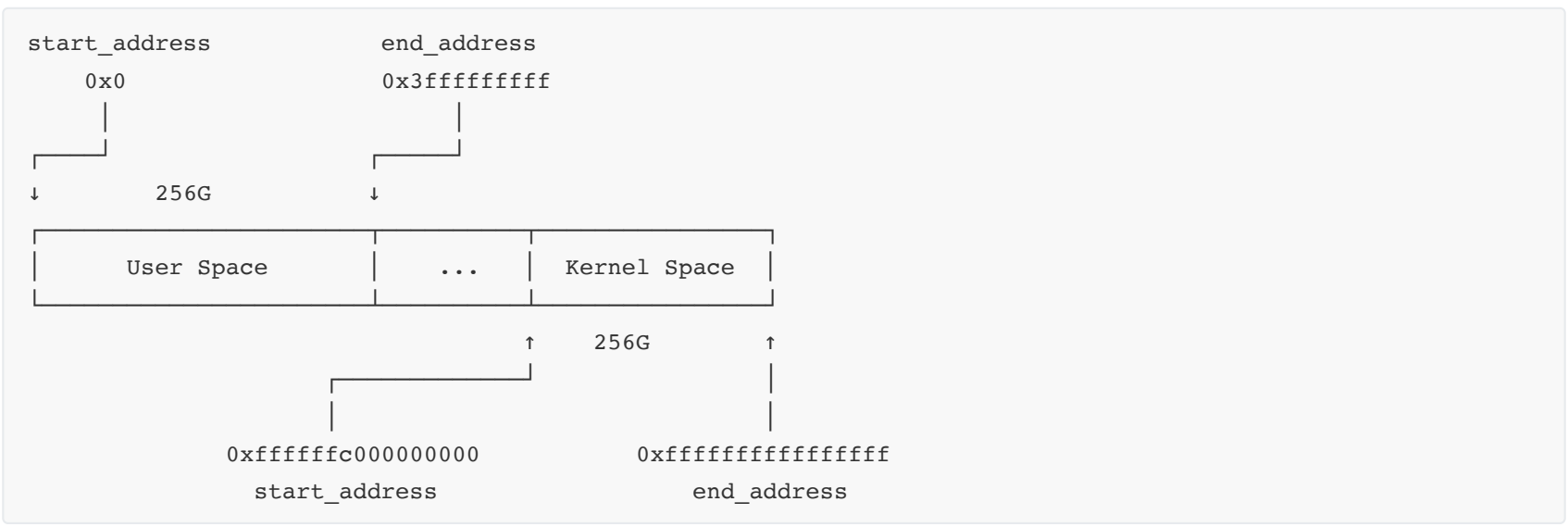
3.1 虚拟内存

MMU（Memory Management Unit），负责`虚拟地址`到`物理地址`的转换。程序在cpu上运行时，他使用的虚拟地址会由MMU进行翻译。为了加速地址翻译的过程，现代CPU都引入了TLB（Translation Lookaside Buffer）。

分页机制的基本思想是将程序的虚拟地址空间划分为连续的，等长的虚拟页。`虚拟页和物理页的页长固定且相等`（一般情况下为4KB），从而操作系统可以方便的为每个程序构造页表，即虚拟页到物理页的映射关系。

逻辑上，该机制下的虚拟地址有两个部分组成：`1.虚拟页号`；`2.页内偏移`；在具体的翻译过程中，MMU首先解析得到虚拟地址中的虚拟页号，并通过虚拟页号查找到对应的物理页，最终用该物理页的起始地址加上页内偏移得到最终的物理地址。

3.2 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 `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 。

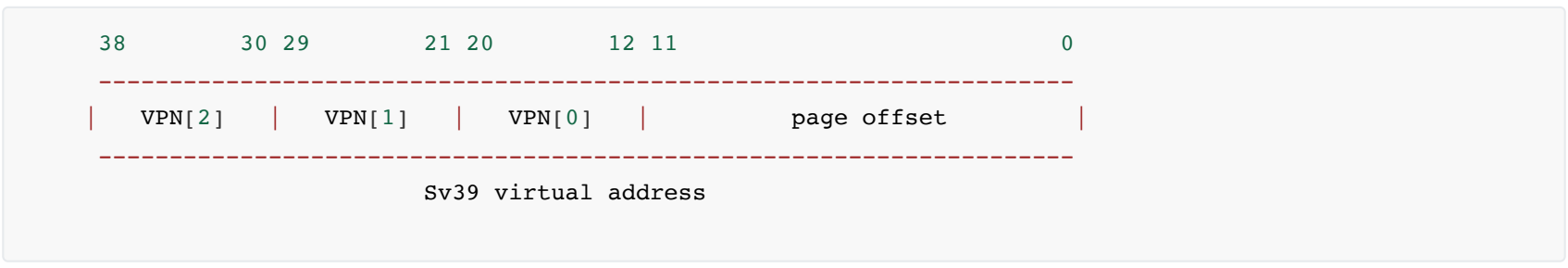
3.3 RISC-V Virtual-Memory System (Sv39)

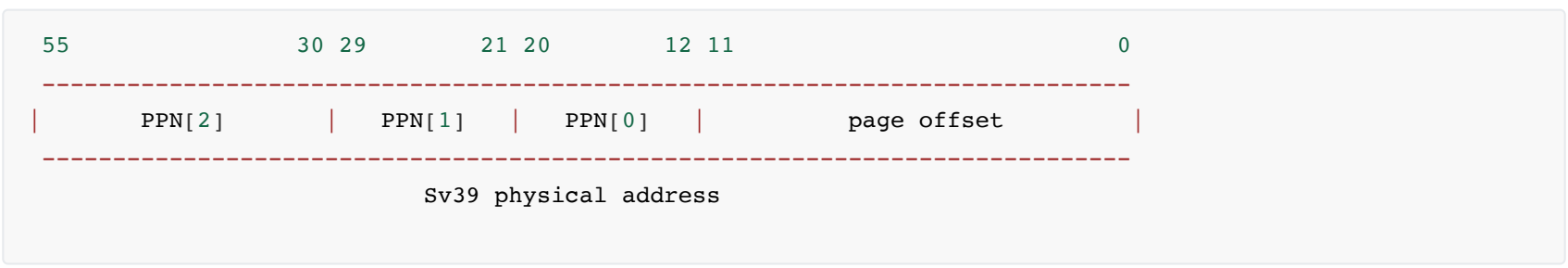
3.3.1 `satp` Register（Supervisor Address Translation and Protection Register）

- MODE 字段的取值如下图：

- 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

3.3.2 RISC-V Sv39 Virtual Address and 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

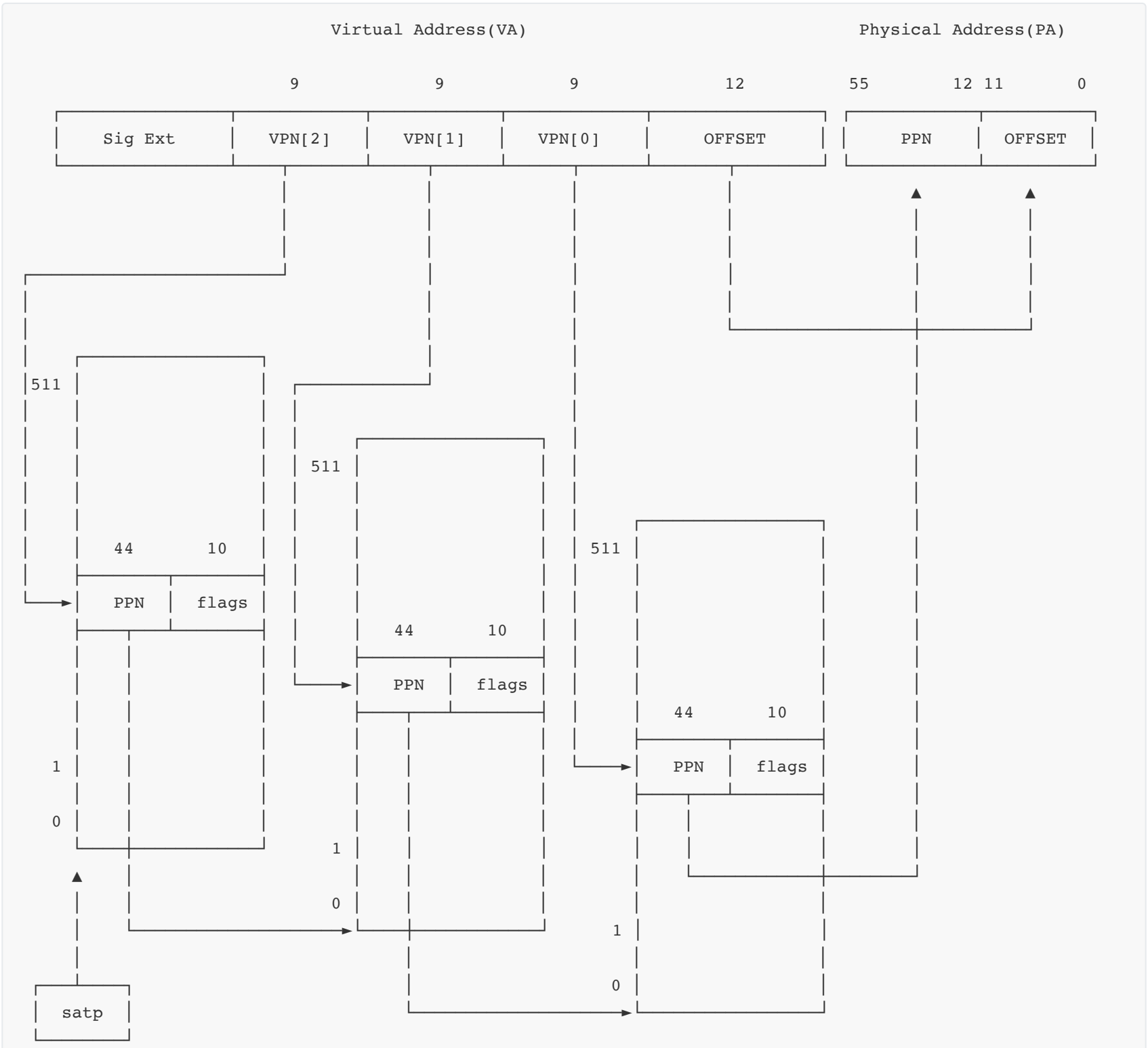
3.3.3 RISC-V Sv39 Page Table Entry

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

- 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

3.3.4 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 :



> 注:
>
> 1. 通过从satp的`PPN`中获取根页表的物理地址。
> 2. 通过Virtual Address(VA) 的VPN段，获取PTE。可以把pagetable看成一个数组，VPN看成下标。由于PAGE_SIZE为4KB，PTE为64bit(8B)，所以一页中有4KB/8B=512个PTE，而每级VPN刚好有9位，与512个PTE一一对应。

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`, `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 `LMA ( 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。

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：

- 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。

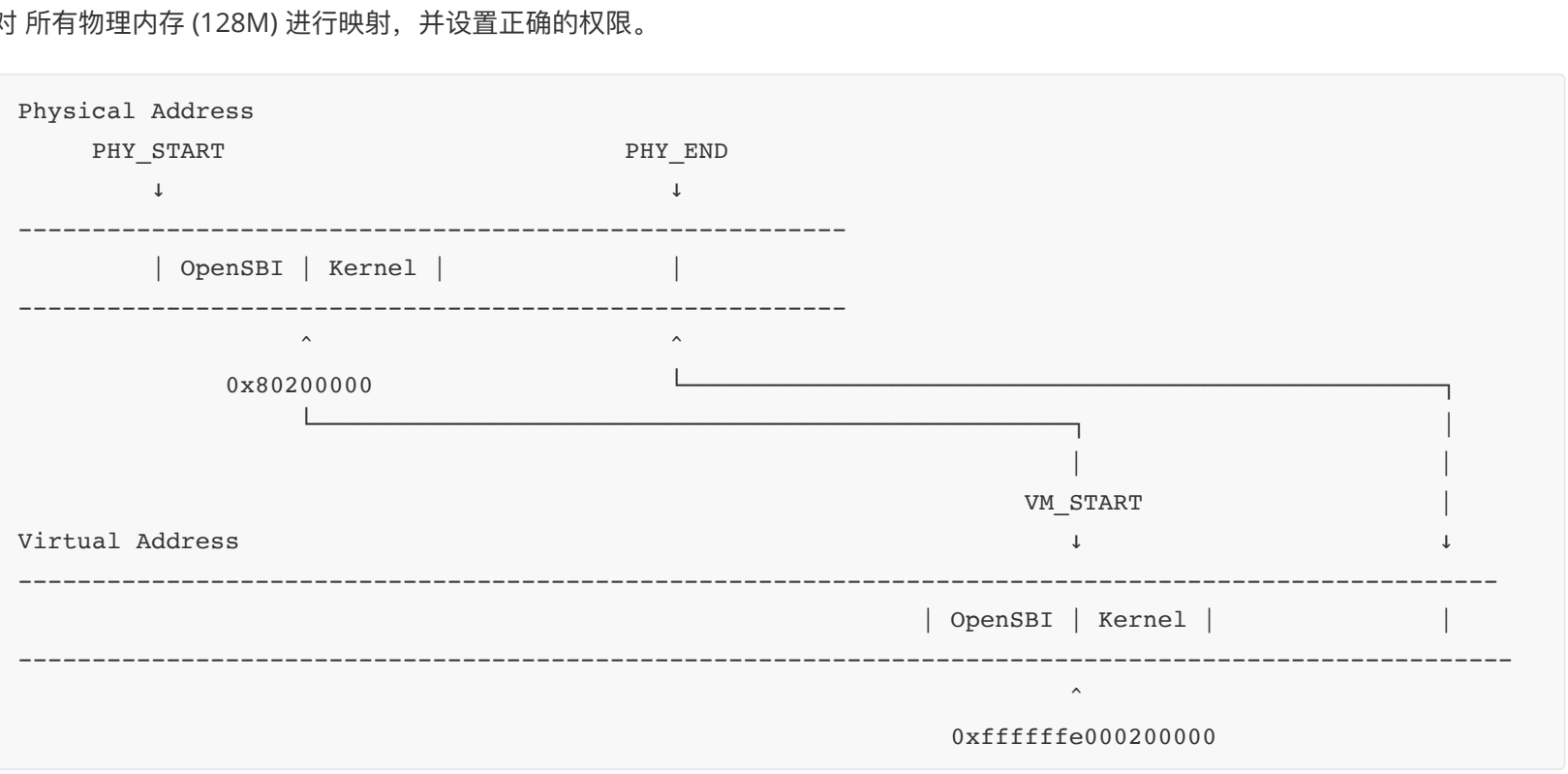
```
// arch/riscv/kernel/vm.c

/* 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
    */
}
```

```
# head.S

_start:

    call setup_vm
    call relocate

    ...

    j start_kernel

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

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

    # set satp with early_pgtbl

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

    # flush tlb
    sfence.vma zero, zero

    ret

    .section .bss.stack
    .globl boot_stack
boot_stack:
    ...
```

> Hint 1: `sfence.vma` 指令用于刷新 TLB
>
> Hint 2: 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。



- 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 。

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

```c
/* 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)));

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

    // No OpenSBI mapping required

    // mapping kernel text X|-|R|V
    create_mapping(...);

    // mapping kernel rodata -|-|R|V
    create_mapping(...);

    // mapping other memory -|W|R|V
    create_mapping(...);

    // set satp with swapper_pg_dir

    YOUR CODE HERE

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


/* 创建多级页表映射关系 */
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，可设置不同section所在页的属性，完成对不同section的保护

    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
    */
}
```

4.3 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。
- 输出示例

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

...
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

...mm_init done!
...proc_init done!
Hello RISC-V
idle process is running!

switch to [PID = 28 COUNTER = 1]
[PID = 28] is running! thread space begin at 0xffffffe007fa2000

switch to [PID = 12 COUNTER = 1]
[PID = 12] is running! thread space begin at 0xffffffe007fb2000

switch to [PID = 14 COUNTER = 2]
[PID = 14] is running! thread space begin at 0xffffffe007fb0000
[PID = 14] is running! thread space begin at 0xffffffe007fb0000

switch to [PID = 9 COUNTER = 2]
[PID = 9] is running! thread space begin at 0xffffffe007fb5000
[PID = 9] is running! thread space begin at 0xffffffe007fb5000

switch to [PID = 2 COUNTER = 2]
[PID = 2] is running! thread space begin at 0xffffffe007fbc000
[PID = 2] is running! thread space begin at 0xffffffe007fbc000

switch to [PID = 1 COUNTER = 2]
[PID = 1] is running! thread space begin at 0xffffffe007fbd000
[PID = 1] is running! thread space begin at 0xffffffe007fbd000

switch to [PID = 29 COUNTER = 3]
[PID = 29] is running! thread space begin at 0xffffffe007fa1000
[PID = 29] is running! thread space begin at 0xffffffe007fa1000
[PID = 29] is running! thread space begin at 0xffffffe007fa1000

switch to [PID = 11 COUNTER = 3]
[PID = 11] is running! thread space begin at 0xffffffe007fb3000
...
```

思考题

1. 验证 `.text`, `.rodata` 段的属性是否成功设置，给出截图。
2. 为什么我们在 `setup_vm` 中需要做等值映射?
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。

作业提交

- 请各位同学独立完成作业，任何抄袭行为都将使本次实验判为0分。
- 请学习基础知识，并按照实验步骤指导完成实验，撰写实验报告。实验报告的要求：
  - 各实验步骤的截图以及结果分析
  - 实验结束后的心得体会
  - 对实验指导的建议（可选）

同学需要提交实验报告以及整个工程代码。在提交前请使用 `make clean` 清除所有构建产物。