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

- 计算机（Intel Core i5以上，4GB内存以上）系统
- Ubuntu 22.04.2 LTS

3. 背景知识

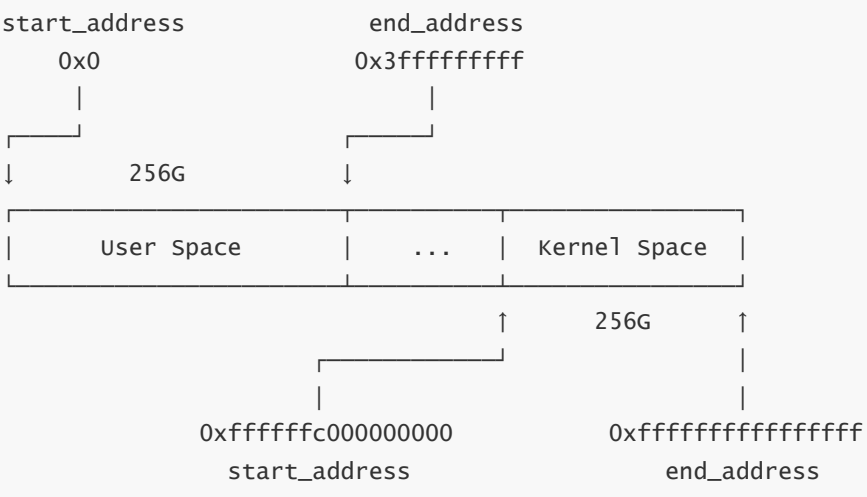
3.1 前言

在 Lab2 中我们赋予了操作系统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，方便内核完成权限检查。

在本次实验中，我们需要关注内核如何**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**。

3.2 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 `RISC-V Linux Kernel Space` 中有一段虚拟地址空间中的区域被称为 `direct mapping area`，为了方便访问内存，内核会预先把所有物理内存都映射至这一块区域，这种映射也被称为 `linear mapping`，因为该映射方式就是在物理地址上添加一个偏移，使得 `VA = PA + PA2VA_OFFSET`。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)

```
 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 。

3.3.2 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，`VPN[2] VPN[1] VPN[0]` (Virtual Page Number) 分别代表每级页表的`虚拟页号`，`PPN[2] PPN[1] PPN[0]` (Physical Page Number) 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

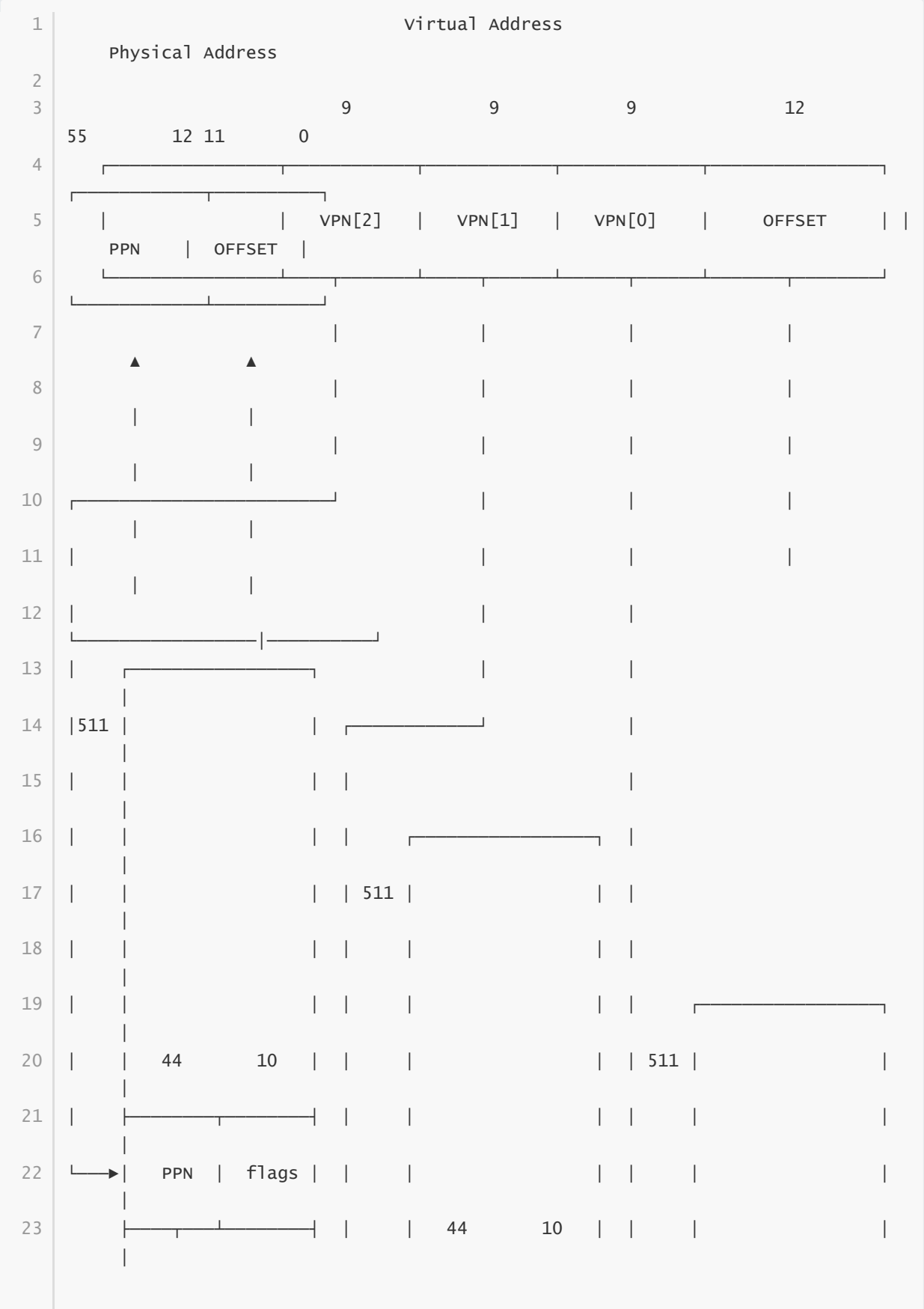
```
 63      54 53        28 27        19 18        10 9   8 7 6 5 4 3 2 1 0
 ------------------------------------------------------------------------
| Reserved |   PPN[2]   |   PPN[1]   |   PPN[0]   | RSW |D|A|G|U|X|W|R|V|
 ------------------------------------------------------------------------
                                                    ↑   ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
                                                    |   | | | | | | | `---- V - Valid
                                                    |   | | | | | | `------ R - Readable
                                                    |   | | | | | `-------- W - Writable
                                                    |   | | | | `---------- X - Executable
                                                    |   | | | `------------ U - User
                                                    |   | | `-------------- G - Global
                                                    |   | `---------------- A - Accessed
                                                    |   `------------------ D - Dirty (0 in page directory)
                                                    `----------------------
Reserved for supervisor software
```

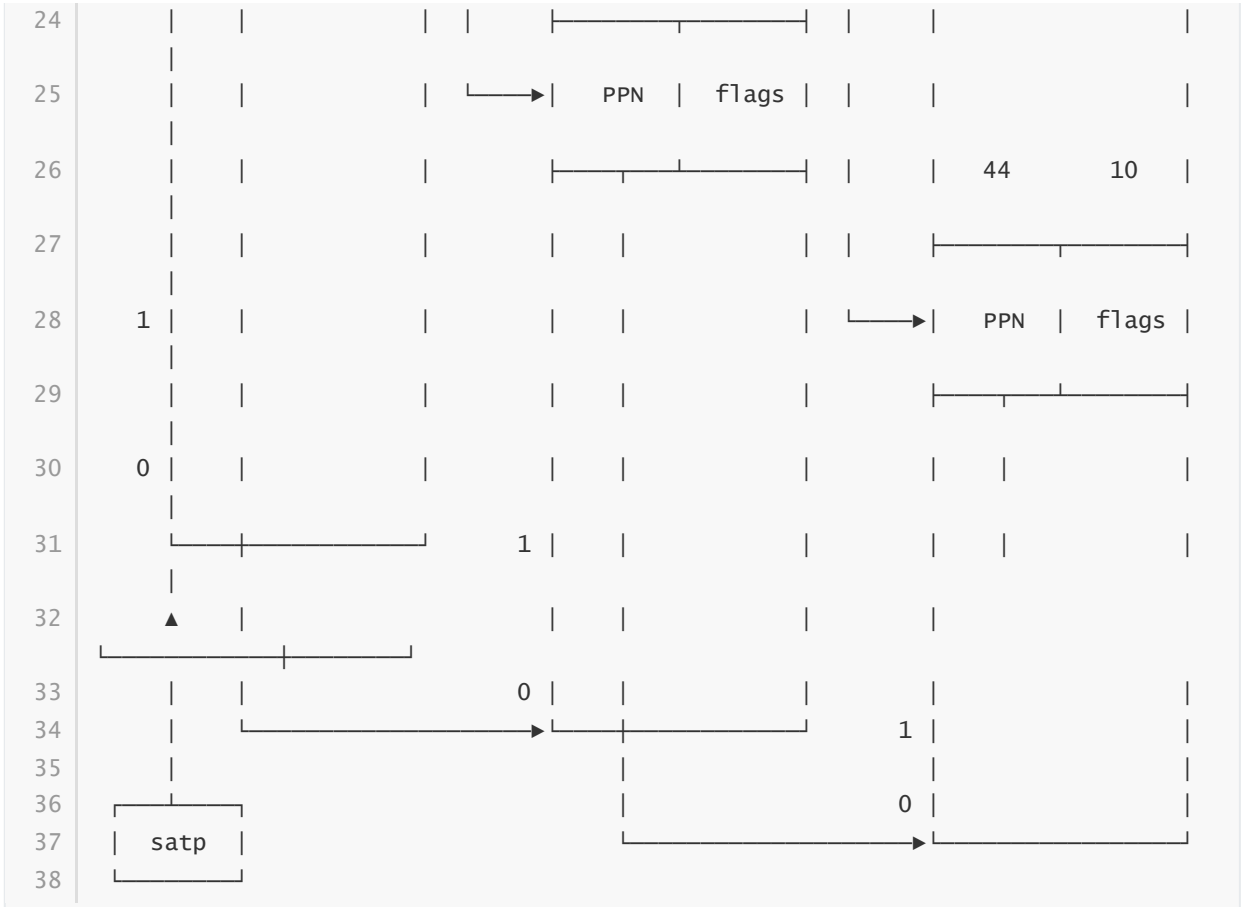
- 0 ~ 9 bit: protection bits
  - V : 有效位，当 V = 0，访问该 PTE 会产生 Pagefault。
  - R : R = 1 该页可读。
  - W : W = 1 该页可写。
  - X : X = 1 该页可执行。
  - U，G，A，D，RSW 本次实验中设置为 0 即可。

- 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 :





4. 实验步骤

4.1 准备工程

- 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`。并按照以下步骤将这些文件正确放置。

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

- 更改项目顶层目录的 `Makefile`，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，修改后的顶层目录的 `Makefile` 如下：

```
export
CROSS_=riscv64-linux-gnu-
GCC=${CROSS_}gcc
LD=${CROSS_}ld
OBJCOPY=${CROSS_}objcopy

ISA=rv64imafd_zifencei
ABI=lp64

INCLUDE = -I $(shell pwd)/include -I $(shell pwd)/arch/riscv/include
CF = -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g
CFLAG = ${CF} ${INCLUDE}

.PHONY:all run debug clean
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

TEST:
    ${MAKE} -C lib all
    ${MAKE} -C test test
    ${MAKE} -C init all
    ${MAKE} -C arch/riscv all
    @echo -e '\n'Build Finished OK

run: all
    @echo Launch the qemu ......
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default

test-run: TEST
    @echo Launch the qemu ......
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default

debug: all
    @echo Launch the qemu for debug ......
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default -S -s

test-debug: TEST
    @echo Launch the qemu for debug ......
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default -S -s

clean:
    ${MAKE} -C lib clean
```

```
47      ${MAKE} -C test clean
48      ${MAKE} -C init clean
49      ${MAKE} -C arch/riscv clean
50      $(shell test -f vmlinux && rm vmlinux)
51      $(shell test -f System.map && rm System.map)
52      @echo -e '\n'Clean Finished
```

4.2 开启虚拟内存映射。

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)，另一次是将其映射到 `direct mapping area` (使得 `PA + PV2VA_OFFSET == VA`)。
    - 将`PHY_START`右移30位，结果和0x1ff相与得到中间9位，作为`early_pgtbl`数组的索引
    - 将`early_pgtbl[index]`等值映射，对齐`PAGESIZE`后再和15按位或将权限 V | R | W | X 位设置为 1
    - 同样将`PA + PV2VA_OFFSET`做映射

  代码如下图所示：

```
1  /* early_pgtbl: 用于 setup_vm 进行 1GB 的 映射。 */
2  unsigned long  early_pgtbl[512] __attribute__((__aligned__(0x1000)));
3  void setup_vm(void) {
4      /*
5      1. 由于是进行 1GB 的映射 这里不需要使用多级页表
6      2. 将 va 的 64bit 作为如下划分： | high bit | 9 bit | 30 bit |
7          high bit 可以忽略dfd
8          中间9 bit 作为 early_pgtbl 的 index
9          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。
10     3. Page Table Entry 的权限 V | R | W | X 位设置为 1
11     */
12     uint64 index = PHY_START >> 30;
13     index = index & 0x1ff;
14     early_pgtbl[index] = PHY_START >> 2 | 15;
15
16     index = (PHY_START + PA2VA_OFFSET) >> 30;
17     index = index & 0x1ff;
18     early_pgtbl[index] = PHY_START >> 2 | 15;
19 }
```

- 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。
    - 首先将`ra`和`sp`的值都增加了偏移量`PA2VA_OFFSET`，进入虚拟地址运行
    - 根据pagetable设置`satp`寄存器，减去偏移量使用物理地址，`mode` 位设为 8；`PPN`位设为PA >> 12
    - flush `tlb`和`icache`

```
1  relocate:
```

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

    ######################
    #   YOUR CODE HERE   #
    li t0, 0xffffffdf80000000
    add ra, ra, t0
    add sp, sp, t0
    ######################

    # set satp with early_pgtbl

    ######################
    #   YOUR CODE HERE   #
    la t0, early_pgtbl
    li t1, 0xffffffdf80000000
    sub t0, t0, t1
    #  PA >> 12 == PPN
    addi t1, x0, 12
    srl t0, t0, t1
    # mode = 8
    addi t2, x0, 1
    slli t2, t2, 63
    or t0, t0, t2
    csrw satp,t0
    ######################

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret
```

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，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，在原来的 `PHY_END` 加上偏移量 `PA2VA_OFFSET`

```
void mm_init(void) {
    kfreerange(_ekernel, (char *)(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，采用三级页表映射。

- 首先获取`text`段的起始物理地址、虚拟地址以及地址空间大小，使用下一段的地址`_srodata - _stext`计算地址空间大小，将`perm`设为1011，即`X|-|R|V`
- 调用`create_mapping`进行地址映射
- 与`text`段类似，依次计算`rodata`段和其他部分的起始地址和空间大小，`rodata`的`perm`设为0011，即`-|-|R|V`，其他段的`perm`设为0111，即`-|W|R|V`，调用`create_mapping`进行地址映射

```
char _stext[];
char _srodata[];
char _sdata[];
char _sbss[];
char _ekernel[];
void setup_vm_final(void) {
    // mapping kernel text X|-|R|V
    uint64 va_text = (uint64)_stext;
    uint64 pa_text = (uint64)_stext - PA2VA_OFFSET;
    uint64 sz_text = (uint64)_srodata - (uint64)_stext;
    int perm_text = 0b1011;
    create_mapping(swapper_pg_dir, va_text,pa_text,sz_text, perm_text);

    // mapping kernel rodata -|-|R|V
    uint64 va_rodata = (uint64)_srodata;
    uint64 pa_rodata = (uint64)_srodata - PA2VA_OFFSET;
    uint64 sz_rodata = (uint64)_sdata - (uint64)_srodata;
    int perm_rodata = 0b0011;
    create_mapping(swapper_pg_dir, va_rodata, pa_rodata, sz_rodata,
perm_rodata);

    // mapping other memory -|W|R|V
    uint64 va_orther = (uint64)_sdata;
    uint64 pa_orther = (uint64)_sdata - PA2VA_OFFSET;
    uint64 sz_orther = PHY_SIZE - OPENSBI_SIZE - ((uint64)_sdata -
(uint64)_stext);
    int perm_orther = 0b0111;
    create_mapping(swapper_pg_dir, va_orther, pa_orther, sz_orther,
perm_orther);
    return;
}
```

- 对每段地址进行映射：
  - 通过`for`循环对该段虚拟地址进行映射，每次增加一页 (一个`PGSIZE`)。
  - 计算多级页表中的各级偏移量
  - 首先检查最高级的页表项是否存在，如果不存在（V bit为0），则使用`kalloc()`分配一个新的页面作为下一级页表，并设置V bit为1。
  - 获取下一级页表的地址，将地址对齐，根据下一级的偏移量查找对应的地址，重复直到找到最终真正的地址
  - 进行下一页的映射

```c
/* 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

    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
    */

   for(uint64 i = va ; i < va + sz ; i += PGSIZE){
        uint64 offset[3];
        uint64 page[3];
        offset[0] = va >> 12 & 0x1FF;
        offset[1] = va >> 21 & 0x1FF;
        offset[2] = va >> 30 & 0x1FF;
        if(!pgtbl[offset[2]] & 1){
            uint64 new_page_addr = kalloc();
            new_page_addr -= 0xffffffdf80000000;
            printk(new_page_addr);
            pgtbl[offset[2]] = (new_page_addr & 1111111111111000) >> 2;
            pgtbl[offset[2]] |= 0x0000000000000001;
        }
        page[2] = pgtbl[offset[2]];

        uint64* pgtbl_next = (uint64*)((page[2] & 1111111111111000) << 2 );
        if(!pgtbl[offset[1]] & 1){
            uint64 new_page_addr = kalloc();
            new_page_addr -= 0xffffffdf80000000;
            pgtbl[offset[1]] = (new_page_addr & 1111111111111000) >> 2;
            pgtbl[offset[1]] |= 0x0000000000000001;
        }
        page[1] = pgtbl[offset[1]];

        uint64* pgtbl_nnext = (uint64*)((page[1] & 1111111111111000) << 2);
        page[0] = ((pa & 1111111111111000) << 2) | perm;
        pgtbl_nnext[offset[0]] = page[0];
        va += PGSIZE;
        pa += PGSIZE;
   }
   return;
}
```

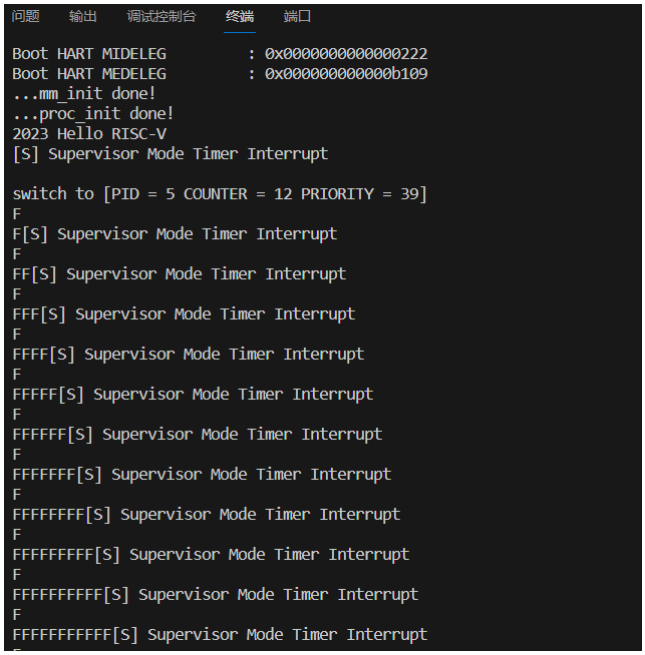
- 在 head.S 中 适当的位置调用 `setup_vm_final`、`setup_vm_final`、`setup_vm`、`relocate`

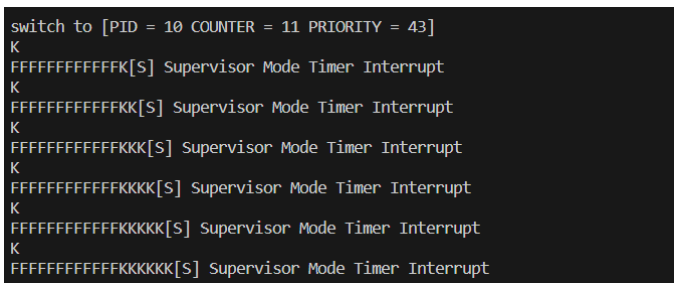
  将栈顶指针减去偏移量获得物理地址，后续调用 `relocate` 再转化为虚拟地址

```
_start:
    #将栈顶指针放入sp
    la sp, boot_stack_top
    li t0, 0xffffffdf80000000
    sub sp, sp, t0
    call setup_vm
    call relocate
    call mm_init
    jal setup_vm_final
    jal task_init
    jal test_init
```

4.3 编译及测试

- 运行后的结果如下图所示，与lab2相同





思考题

1. 验证 `.text`，`.rodata` 段的属性是否成功设置，给出截图。
    - 在GDB中调试程序，先在`start_kernel`处打上断点，再运行到`start_kernel`

```
(gdb) b start_kernel
Breakpoint 1 at 0xffffffe0002011c8: file main.c, line 5.
(gdb) continue
Continuing.

Breakpoint 1, start_kernel () at main.c:5
5           printk("2023");
(gdb)
```

- 使用 `info files` 命令查看文件信息：

```
(gdb) info files
Symbols from "/mnt/d/study/OS/os23fall-stu/src/lab3/vmlinux".
Remote target using gdb-specific protocol:
        `/mnt/d/study/OS/os23fall-stu/src/lab3/vmlinux', file type elf64-littleriscv.
        Entry point: 0xffffffe000200000
        0xffffffe000200000 - 0xffffffe000202598 is .text
        0xffffffe000203000 - 0xffffffe00020334e is .rodata
        0xffffffe000204000 - 0xffffffe000204008 is .data
        0xffffffe000204008 - 0xffffffe000204058 is .got
        0xffffffe000204058 - 0xffffffe000204068 is .got.plt
        0xffffffe000205000 - 0xffffffe00020a1c5 is .bss
        While running this, GDB does not access memory from...
Local exec file:
        `/mnt/d/study/OS/os23fall-stu/src/lab3/vmlinux', file type elf64-littleriscv.
        Entry point: 0xffffffe000200000
        0xffffffe000200000 - 0xffffffe000202598 is .text
        0xffffffe000203000 - 0xffffffe00020334e is .rodata
        0xffffffe000204000 - 0xffffffe000204008 is .data
        0xffffffe000204008 - 0xffffffe000204058 is .got
        0xffffffe000204058 - 0xffffffe000204068 is .got.plt
        0xffffffe000205000 - 0xffffffe00020a1c5 is .bss
```

可以看到`.text`，`.rodata`的地址为虚拟地址

- 使用 `maintenance info sections` 命令来查看详细的段信息

```
(gdb) maintenance info sections
Exec file: `/mnt/d/study/OS/os23fall-stu/src/lab3/vmlinux', file type elf64-littleriscv.
 [0]      0xffffffe000200000->0xffffffe000202598 at 0x00001000: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
 [1]      0xffffffe000203000->0xffffffe00020334e at 0x00004000: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
 [2]      0xffffffe000204000->0xffffffe000204008 at 0x00005000: .data ALLOC LOAD DATA HAS_CONTENTS
 [3]      0xffffffe000204008->0xffffffe000204058 at 0x00005008: .got ALLOC LOAD DATA HAS_CONTENTS
 [4]      0xffffffe000204058->0xffffffe000204068 at 0x00005058: .got.plt ALLOC LOAD DATA HAS_CONTENTS
 [5]      0xffffffe000205000->0xffffffe00020a1c5 at 0x00005068: .bss ALLOC
 [6]      0x00000000->0x000018fb at 0x00005068: .debug_info READONLY HAS_CONTENTS
 [7]      0x00000000->0x00000d1a at 0x00006963: .debug_abbrev READONLY HAS_CONTENTS
 [8]      0x00000000->0x000003c0 at 0x00007680: .debug_aranges READONLY HAS_CONTENTS
 [9]      0x00000000->0x00000285 at 0x00007a40: .debug_rnglists READONLY HAS_CONTENTS
 [10]     0x00000000->0x0000209f at 0x00007cc5: .debug_line READONLY HAS_CONTENTS
 [11]     0x00000000->0x00000549 at 0x00009d64: .debug_str READONLY HAS_CONTENTS
 [12]     0x00000000->0x000001b5 at 0x0000a2ad: .debug_line_str READONLY HAS_CONTENTS
 [13]     0x00000000->0x0000002b at 0x0000a462: .comment READONLY HAS_CONTENTS
 [14]     0x00000000->0x00000032 at 0x0000a48d: .riscv.attributes READONLY HAS_CONTENTS
 [15]     0x00000000->0x00000660 at 0x0000a4c0: .debug_frame READONLY HAS_CONTENTS
```

可以看到：

`.text`的属性为`ALLOC`（段在进程的虚拟地址空间中有分配的空间）；`LOAD`（段从文件加载到内存中）；`READONLY`（段是只读的)；`CODE`（段包含可执行代码）

`.rodata`的属性为`ALLOC`（段在进程的虚拟地址空间中有分配的空间）；`LOAD`（段从文件加载到内存中）；`READONLY`（段是只读的)；`DATA`（段包含只读数据）

属性设置正确

2. 为什么我们在 `setup_vm` 中需要做等值映射?

   答：在建立三级页表的时候需要通过物理页号在页表中找到物理地址，此时的运行在虚拟地址上，直接使用物理地址会产生访问错误，所以需要进行等值映射

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。

   答：在通过物理页号得到下一级页表中的物理地址时直接加上偏移量`PA2VA_OFFSET`，得到对应的虚拟地址，就不用在 `setup_vm` 中做等值映射了

```
uint64* pgtbl_next = (uint64*)((page[2] & 1111111111111000) << 2 +
PA2VA_OFFSET);
uint64* pgtbl_nnext = (uint64*)((page[1] & 1111111111111000) << 2 +
PA2VA_OFFSET);
```