浙江大学

本科实验报告

课程名称：操作系统

姓　　名：黄文杰

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：软件工程

学　　号：3210103379

指导教师：夏莹杰

2023 年　　10 月　　29 日

浙江大学操作系统实验报告

实验名称：Lab 3: RV64 虚拟内存管理

电子邮件地址：3210103379@zju.edu.cn 手机：15167970568

实验地点：曹西-503 实验日期：2023 年 10 月 29 日

一、实验目的和要求

• 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。

二、实验过程

1. 准备工程

• 此实验基于 lab2 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)
```

添加之后的 defs.h 如下：

vm.c defs.h

```
#ifndef _DEFS_H
#define _DEFS_H

// #include "types.h"
#define PHY_START 0x0000000080000000
#define PHY_SIZE  128 * 1024 * 1024 // 128MB,  QEMU 默认内存大小
#define PHY_END   (PHY_START + PHY_SIZE)

#define PGSIZE 0x1000 // 4KB
#define PGROUNDUP(addr) ((addr + PGSIZE - 1) & (~(PGSIZE - 1)))
#define PGROUNDDOWN(addr) (addr & (~(PGSIZE - 1)))

#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)

#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    asm volatile ("csrr %0,"  #csr          \
                  :"=r" (__v));             \
    __v;                                    \
})

#define csr_write(csr, val)                         \
({                                                  \
    uint64 __v = (uint64)(val);                     \
    asm volatile ("csrw " #csr ", %0"               \
                  : : "r" (__v)                     \
                  : "memory");                      \
})
```

• 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。

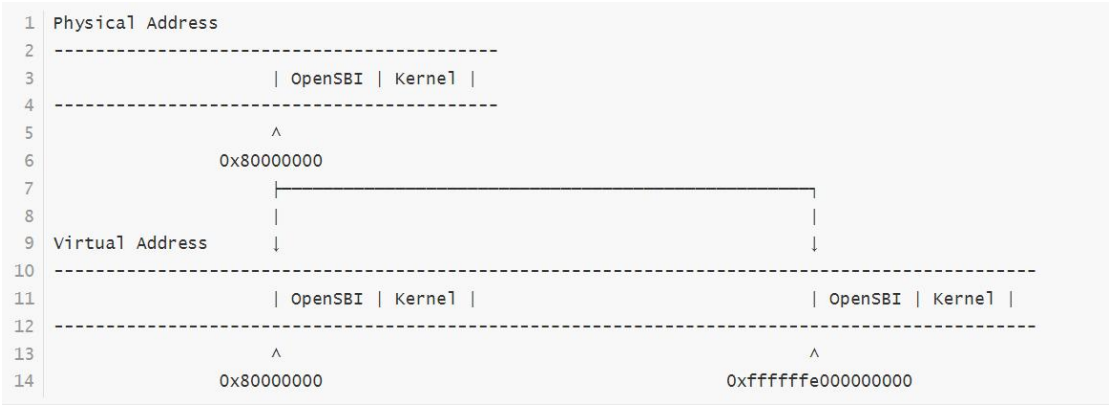
```
.
└── arch
    └── riscv
        └── kernel
            ├── Makefile
            └── vmlinux.lds.S
```

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。

2. 开启虚拟内存映射。

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：



实现代码如下：

*vm.c

```c
#include "defs.h"
#include "types.h"
#include "vm.h"
#include "mm.h"
#include "string.h"
#include "printk.h"

/* 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, 0x0, PGSIZE);
    unsigned long pa = PHY_START;
    unsigned long va = PHY_START;
    int index = (va >> 30) & 0x1ff;
    early_pgtbl[index] = (((pa >> 30) & 0x3ffffff) << 28) | 0xf;
    index = VM_START;
    early_pgtbl[index] = (((pa >> 30) & 0x3ffffff) << 28) | 0xf;
    printk("setup_vm is done !")
}
```

其中，第一次映射实现了 va=PHY_START 和 pa=PHY_START 的一一对应。第二次映射实现了 va=VM_START 和 pa=PHY_START 的一一对应，其中 PHY_START 和 VM_START 分别为物理地址和虚拟地址的起始地址。

·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。

实现的代码如下：

```
48 relocate:
49     # set ra = ra + PA2VA_OFFSET
50     # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
51
52     ######################
53     #   YOUR CODE HERE   #
54     ######################
55
56     # 把 t0 的值设为 0xffffffdf80000000 (PA2VA_OFFSET)
57     # 先让 t0 -> 0x80000000
58     addi t0, x0, 1
59     slli t0, t0, 31
60     # 再让 t1 -> 0xffffffdf00000000
61     lui t1, 0xfffff
62     li t2, 0xfdf
63     add t1, t1, t2
64     slli t1, t1, 32
65     # 两者相加 t0 = 0xffffffdf00000000 + 0x80000000
66     add t0, t0, t1
67     add ra, ra, t0   # Add PA2VA_OFFSET to ra
68     add sp, sp, t0   # Add PA2VA_OFFSET to sp
69
70     # set satp with early_pgtbl
71
72     ######################
73     #   YOUR CODE HERE   #
74     ######################
75     la t1, early_pgtbl
76     sub t1,t1,t0
77     li t2,8
78     slli t2, t2, 60
79     srli t1,t1,12
80     or t1,t1,t2
81     csrw satp,t1
82
83
84     # flush tlb
85     sfence.vma zero, zero
```

2.2 setup_vm_final 的实现

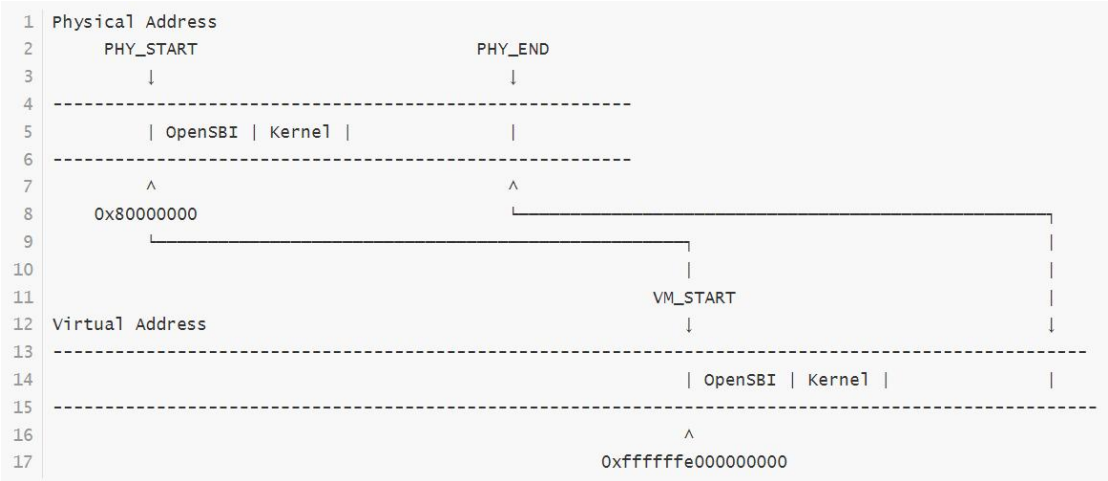
·由于 setup_vm_final 中需要申请页面的接口， 应该在其之前完成内存管理初始化， 可能需要修改 mm.c 中的代码，mm.c 中初始化的函数接收的起始结束

地址需要调整为虚拟地址。

修改如下：

```
void mm_init(void) {
    kfreerange(_ekernel, (char *)(PHY_END+PA2VA_OFFSET));
    printk("...mm_init done!\n");
    return;
}
```

• 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。

实现的代码如下：

create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm):

```c
/* 创建多级页表映射关系 */
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
    */


    /*
        1.从satp的PPN中获取根页表的物理地址
        2通过pagetable中的VPN段获取PTE。(可以把pagetable看成一个数组，VPN看成下标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)。
        3.检查PTE的 v bit，如果不合法，应该产生page fault异常。
        4检查PTE的 Rwx bits,如果全部为0，则从PTE中的PPN[2-0]得到的是下一级页表的物理地址则回到
        第二步。否则当前为最后一级页表，PPNI2-01得到的是最终物理页的地址。
        5.将得到最终的物理页地址，与偏移地址相加，得到最终的物理地址。
        6.对齐注意
    */

    unsigned long offset = 0;

    while (offset < sz) {
        unsigned long vpn2 = (va >> 30) & 0x1ff;
        unsigned long vpn1 = (va >> 21) & 0x1ff;
        unsigned long vpn0 = (va >> 12) & 0x1ff;
        unsigned long *pgtbl1;
        unsigned long *pgtbl0;
        // 处理第二层页表
        ..
        if (!(pgtbl[vpn2] & 0x1)) {
            // 给Invalid页新分配空间
            pgtbl1 = (unsigned long*)(kalloc()-PA2VA_OFFSET);
            pgtbl[vpn2] = (((unsigned long)pgtbl1 >> 2) | 0x1);
        }
        else {
            pgtbl1 = (unsigned long*)((pgtbl[vpn2] >>10) << 12);
        }
        // 处理第三层页表
        if (!(pgtbl1[vpn1] & 0x1)) {
            // 给Invalid页新分配空间
            pgtbl0 = (unsigned long*)(kalloc()-PA2VA_OFFSET);
            pgtbl1[vpn1] = (((unsigned long)pgtbl0 >> 2) | 0x1);
        }
        else {
            pgtbl0 = (unsigned long*)((pgtbl1[vpn1] >>10) << 12);
        }
        // 处理需要映射的物理地址
        if (!(pgtbl0[vpn0] & 0x1)) {
            pgtbl0[vpn0] = ((pa>>12)<<10) | (perm & 0xf);
        }

        va+=PGSIZE;
        pa+=PGSIZE;
        offset += PGSIZE;
    }

    return;
}
```

其实现思路如下：逐级遍历 vpn2-vpn0，通过 vpn 和对应级别的根页表地址找到对应的页表项，并判断 v bit 是否为 0，若为 0 则需要为其新开辟一块空间并赋值，然后将其 v bit 设为 1；若不为 0 则将其作为下一级根页表的地址，重复如

此操作直至执行至最低层级，此时需要将物理地址以及权限写入对应的页表项。

void setup_vm_final(void):

```
38 void setup_vm_final(void) {
39     memset(swapper_pg_dir, 0x0, PGSIZE);
40     // No OpenSBI mapping required
41
42     // mapping kernel text X|-|R|V
43     unsigned long pa = PHY_START + OPENSBI_SIZE;
44     unsigned long va = VM_START  + OPENSBI_SIZE;
45     unsigned long size = (unsigned long)_srodata-(unsigned long)_stext;
46     create_mapping(swapper_pg_dir,va,pa,size,11);
47     printk("mapping kernel text !\n");
48
49     // mapping kernel rodata -|-|R|V
50     pa += size;
51     va += size;
52     size = (unsigned long)_sdata-(unsigned long)_srodata;
53     create_mapping(swapper_pg_dir,va,pa,size,3);
54     printk("mapping kernel rodata !\n");
55
56     // mapping other memory -|W|R|V
57     pa += size;
58     va += size;
59     size = PHY_SIZE - ((unsigned long)_sdata-(unsigned long)_stext);
60     create_mapping(swapper_pg_dir,va,pa,size,7);
61     printk("mapping other memory !\n");
62
63     // set satp with swapper_pg_dir
64
65     // YOUR CODE HERE
66     asm volatile (
67         "addi t0, x0, 8\n"
68         "slli t0, t0, 60\n"
69         "mv t1, %[addr]\n"
70         "srli t1, t1, 12\n"
71         "or t0, t0, t1\n"
72         "csrw satp, t0"
73         :
74         :[addr] "r" ((unsigned long)swapper_pg_dir - PA2VA_OFFSET)
75         :"memory"
76     );
77
78     // flush TLB
79     asm volatile("sfence.vma zero, zero");
80
81     // flush icache
82     asm volatile("fence.i");
83
84     return;
85 }
```

• 在 head.S 中适当的位置调用 setup_vm_final 。

修改后的 head.S 代码如下：

```
12 _start:
13     # la sp, boot_stack_top
14     # sp:0x80200000
15     #li t0, 0x80200   # 将0x80208加载到t0寄存器
16     #slli t0, t0, 12  # 左移12位，将高位部分移到正确位置
17     #mv sp, t0        # 将t0的值复制到sp寄存器
18
19     # 把 t0 的值设为 0xffffffdf80000000 (PA2VA_OFFSET)
20     # 先让 t0 -> 0x80000000
21     addi t0, x0, 1
22     slli t0, t0, 31
23     # 再让 t1 -> 0xffffffdf00000000
24     lui t1, 0xfffff
25     li t2, 0xfdf
26     add t1, t1, t2
27     slli t1, t1, 32
28     # 两者相加 t0 = 0xffffffdf00000000 + 0x80000000
29     add t0, t0, t1
30
31     la t1, boot_stack_top
32     # let boot_stack_top-PA2VA_OFFSET
33     sub t1,t1,t0
34     mv sp,t1
35
36     call setup_vm
37     call relocate
38     call mm_init
39     call setup_vm_final
40     call task_init
```

同时，由于该版本的编译器在开启 satp 之后所有的符号表都已经是虚拟地址，所以不能直接将 boot_stack_top 赋值给 sp 栈指针，要先去减去 PA2VA_OFFSET

3. 编译计测试

编译运行后的结果如下：

```
setup_vm is done !
...mm_init done!
mapping kernel text !
mapping kernel rodata !
mapping other memory !
...proc_init done!
Hello RISC-V
idle process is running!
_stext(address) = -137436856320
_srodata(address) = -137436848128
_stext(value) = 147
_srodata(value) = 46

switch to [PID = 8 COUNTER = 1]
[PID = 8] is running. thread space begin at 0xffffffe007fb6000

switch to [PID = 17 COUNTER = 1]
[PID = 17] is running. thread space begin at 0xffffffe007fad000

switch to [PID = 18 COUNTER = 1]
[PID = 18] is running. thread space begin at 0xffffffe007fac000
```

三、讨论和心得

由于对虚拟地址的相关知识不太熟悉，在该实验上我花了很多时间，并且在参考实验指导的时候，我发现其中的内容不是很详细，很多地方我还是没有搞清楚。我把我在实验过程中遇到的问题进行了汇总：

1. Kernel 根目录下的 MakeFile 文件貌似有问题，需要自己修改才能正常编译。下面是我和组员修改后的代码:

打开(O) Makefile ~/os_Lab/lab3/arch/riscv/kernel 保存(S)

```
1 ASM_SRC         = $(filter-out vmlinux.lds.S,$(sort $(wildcard *.S)))
2 C_SRC       = $(sort $(wildcard *.c))
3 OBJ               = $(patsubst %.S,%.o,$(ASM_SRC)) $(patsubst %.c,%.o,$(C_SRC))
4
5 all:$(OBJ) vmlinux.lds
6
7 vmlinux.lds: vmlinux.lds.S
8         $(GCC) -E -P -I../include -I../../../include -o $@ $^
9
10 %.o:%.S
11         ${GCC}  ${CFLAG} -c $<
12
13 %.o:%.c
14         ${GCC}  ${CFLAG} -c $<
15
16 clean:
17         $(shell rm *.o vmlinux.lds 2>/dev/null)
```

2. Head.s 中_start 段的第一句本来应该是把栈指针（sp）设为 boot_stack_top，但在该实验中，貌似这样设程序运行不了，我的做法是把 OpenSBI 之后的地址赋值给了 sp（0x80200000）,但我想不明白为什么设为 boot_stack_top 不行。（后来在助教的解释下，我知道了由于编译器的版本不同，有的编译器在开启 satp 之后所有的符号表都已经处于虚拟状态了，所以不能直接将 boot_stack_top 赋值给 sp，要先减去 PA2VA_OFFSET,修改后的代码见第二张图）

```
_start:
    # la sp, boot_stack_top
    # sp:0x80200000
    li t0, 0x80200   # 将0x80208加载到t0寄存器
    slli t0, t0, 12  # 左移12位，将高位部分移到正确位置
    mv sp, t0        # 将t0的值复制到sp寄存器
```

（修改后的代码：）

```
_start:
    # la sp, boot_stack_top
    # sp:0x80200000
    #li t0, 0x80200   # 将0x80208加载到t0寄存器
    #slli t0, t0, 12  # 左移12位，将高位部分移到正确位置
    #mv sp, t0        # 将t0的值复制到sp寄存器

    # 把 t0 的值设为 0xffffffdf80000000 (PA2VA_OFFSET)
    # 先让 t0 -> 0x80000000
    addi t0, x0, 1
    slli t0, t0, 31
    # 再让 t1 -> 0xffffffdf00000000
    lui t1, 0xfffff
    li t2, 0xfdf
    add t1, t1, t2
    slli t1, t1, 32
    # 两者相加 t0 = 0xffffffdf00000000 + 0x80000000
    add t0, t0, t1

    la t1, boot_stack_top
    # let boot_stack_top-PA2VA_OFFSET
    sub t1,t1,t0
    mv sp,t1

    call setup_vm
    call relocate
    call mm_init
    call setup_vm_final
    call task_init
```

3. 我曾碰到过循环运行初始化代码的情况，这似乎是由于我 vm.c 文件没有写正确的原因。但我还是对循环运行初始化代码的情况感到困惑，因为根据我的输出信息，程序还并未运行时钟中断的部分，所以循环输出初始化信息的情况不应该由时钟终端引发，我猜测是由于某种错误引发了系统异常，在这种异常下 kernel 的代码段被重新执行。

四、思考题

1. 验证 .text, .rodata 段的属性是否成功设置，给出截图。

.text 段属性是可执行和可读，.rodata 段的属性是可读。

程序能够编译和正常运行，可以说明.text 段的可执行属性设置成功。

对于.text 和 .rodata 的可读属性，可以通过在 main.c 中添加如下打印语句来验证。

```
5 extern void test();
6 extern char _srodata[];
7 extern char _stext[];
8
9 int start_kernel() {
10     printk("Hello RISC-V\n");
11     printk("idle process is running!\n");
12     printk("_stext(address) = %ld\n", _stext);
13     printk("_srodata(address) = %ld\n", _srodata);
14     printk("_stext(value) = %ld\n", *_stext);
15     printk("_srodata(value) = %ld\n", *_srodata);
```

输出结果如下：

```
setup_vm is done !
...mm_init done!
mapping kernel text !
mapping kernel rodata !
mapping other memory !
...proc_init done!
Hello RISC-V
idle process is running!
_stext(address) = -137436856320
_srodata(address) = -137436848128
_stext(value) = 147
_srodata(value) = 46
```

由此可知这两个段的可读属性设置成功！（要注意一定要用 char 类型的数组或指针来接受_srodata 和_stext，否则就不能输出这两个段首部的值）

下面校验两个段的写属性，在 main.c 增加如下几行代码：

```
 9 int start_kernel() {
10     printk("Hello RISC-V\n");
11     printk("idle process is running!\n");
12     printk("_stext(address) = %ld\n", _stext);
13     printk("_srodata(address) = %ld\n", _srodata);
14     printk("_stext(value) = %ld\n", *_stext);
15     printk("_srodata(value) = %ld\n", *_srodata);
16     *_stext = 1;
17     *_srodata = 2;
18     printk("_stext(value) = %ld\n", *_stext);
19     printk("_srodata(value) = %ld\n", *_srodata);
```

编译后的输出结果如下图：

```
setup_vm is done !
...mm_init done!
mapping kernel text !
mapping kernel rodata !
mapping other memory !
...proc_init done!
Hello RISC-V
idle process is running!
_stext(address) = -137436856320
_srodata(address) = -137436848128
_stext(value) = 147
_srodata(value) = 46

switch to [PID = 8 COUNTER = 1]
```

可见两个段都没有设置写属性，这符合我们的预期。

2. 为什么我们在 setup_vm 中需要做等值映射?

在 setup_vm 函数中进行等值映射（identity mapping）的目的是为了确保虚拟地址空间中的一部分能够直接映射到相同的物理地址，从而在启动操作系统的早期阶段建立起一个合适的内存布局，也能保证在建立三级页表需要访问物理地址的时候不会出现内存访问的错误。因为 CPU 发出的所有内存访问指令都是虚拟地址，都需要通过 satp 中存储的顶级页表信息转换成对应的物理地址，所以对三级页表（页表项存储的是低位的虚拟地址）的访问也需要有一个低位虚拟地址到物理地址的映射，否则 CPU 会找不到对应的空间。

等值映射的主要用途包括：

1. 早期初始化：在启动操作系统的早期阶段，内核可能还没有建立完整的虚拟

内存管理系统。等值映射允许内核使用物理地址来访问内核代码和数据，而无需考虑虚拟地址空间。这有助于引导和初始化系统。

2. 页面表初始化：在启动时，操作系统需要初始化虚拟内存管理系统，包括建立多级页表（page table）等数据结构。通过等值映射，可以简化这一过程，因为一部分虚拟地址空间与物理地址是一一对应的。

3. 异常处理：等值映射可以简化异常处理过程。例如，如果发生页错误（page fault）等异常，内核可以使用物理地址来访问页表，而无需转换虚拟地址。

4. 内核调试：在内核调试期间，等值映射允许开发人员更轻松地访问内核代码和数据，而无需进行虚拟地址到物理地址的转换。

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。

（1）首次设置 satp 寄存器后，PC 指向一个低地址，此时的 CPU 将认为这是一个“虚拟地址”，但无法找到其对应的有效物理地址，导致系统抛出中断，此时，只需要将中断处理地址设为下一步要执行的指令的虚拟 PC 地址，程序即可正常运行。因此需要在 Head.s 的 relocate 部分新增如下几行代码，其目的是计算出 “csrw satp,t1” 这条指令的下一条指令的虚拟地址（要加上偏移 PA2VA_OFFSET），并赋值给 stevc，这样当程序中断时就会执行下一条指令，程序仍能正常运行。

```
# set satp with early_pgtbl

######################
#   YOUR CODE HERE   #
######################
la t1, early_pgtbl
sub t1,t1,t0
li t2,8
slli t2, t2, 60
srli t1,t1,12
or t1,t1,t2
auipc t2,0
addi t2,t2,20
add t2,t2,t0
csrw stvec, t2
csrw satp,t1
```

（2）将通过 PTE 获取的 PPN 得到的地址转换为“高位”虚拟地址，使其能够被正常映射，因此需要修改 create_mapping 如下：

```
    // 处理第二层页表
    if (!(pgtbl[vpn2] & 0x1)) {
        // 给Invalid页新分配空间
        //pgtbl1 = (unsigned long*)(kalloc()-PA2VA_OFFSET);
        //pgtbl[vpn2] = (((unsigned long)pgtbl1 >> 2) | 0x1);
        pgtbl1 = (unsigned long*)(kalloc());
        pgtbl[vpn2] = (((((unsigned long)pgtbl1 - PA2VA_OFFSET) >> 12) << 10) | 0x1);
    }
    else {
        //pgtbl1 = (unsigned long*)((pgtbl[vpn2] >>10) << 12);
        pgtbl1 = (unsigned long*)(((pgtbl[vpn2] >>10) << 12) + PA2VA_OFFSET);
    }
    // 处理第三层页表
    if (!(pgtbl1[vpn1] & 0x1)) {
        // 给Invalid页新分配空间
        //pgtbl0 = (unsigned long*)(kalloc()-PA2VA_OFFSET);
        //pgtbl1[vpn1] = (((unsigned long)pgtbl0 >> 2) | 0x1);
        pgtbl0 = (unsigned long*)(kalloc());
        pgtbl1[vpn1] = (((((unsigned long)pgtbl0 - PA2VA_OFFSET) >> 12) << 10) | 0x1);
    }
    else {
        //pgtbl0 = (unsigned long*)((pgtbl1[vpn1] >>10) << 12);
        pgtbl0 = (unsigned long*)(((pgtbl1[vpn1] >>10) << 12)+ PA2VA_OFFSET);
    }
```

（需要注意的是，此时 kalloc 函数返回值已经是虚拟地址了）

此时去除 setup_vm 中的等值映射部分：

```
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, 0x0, PGSIZE);
    unsigned long pa = PHY_START;
    unsigned long va = PHY_START;
    int index = (va >> 30) & 0x1ff;
    //early_pgtbl[index] = (((pa >> 30) & 0x3ffffff) << 28) | 0xf;
    va = VM_START;
    index = (va >> 30) & 0x1ff;
    early_pgtbl[index] = (((pa >> 30) & 0x3ffffff) << 28) | 0xf;
    printk("setup_vm is done !\n");
}
```

编译运行，发现程序可以正常执行：

```
setup_vm is done !
...mm_init done!
mapping kernel text !
mapping kernel rodata !
mapping other memory !
...proc_init done!
Hello RISC-V
idle process is running!
_stext(address) = -137436856320
_srodata(address) = -137436848128
_stext(value) = 147
_srodata(value) = 46

switch to [PID = 8 COUNTER = 1]
[PID = 8] is running. thread space begin at 0xffffffe007fb6000

switch to [PID = 17 COUNTER = 1]
[PID = 17] is running. thread space begin at 0xffffffe007fad000
```

五、附录

S模式提供了一种传统的虚拟内存系统，它将内存划分为固定大小的页来进行地址转换和对内存内容的保护。启用分页的时候，大多数地址（包括 load和 store的有效地址和PC中的地址）都是虚拟地址 。要访问物理内存，它们必须被转换为真正的物理地址 ，这通过遍历一种称为页表的高基数树实现。页表中的叶节点指示虚地址是否已经被映射到了真正的物理页面，如果是，则指示了哪些权限模式和通过哪种类型的访问可以操作这个页。访问未被映射的页或访问权限不足会导致页错误例外 page fault exception）。

RV64支持多种分页方案，本次实验使用了Sv39。 Sv39使用4KB大的基页，页表项的大小是8个字节，为了保证页表大小和页面大小一致，树的基数相应地降到2^9，树也变为三层。 Sv39的512 GB地址空间（虚拟地址）划分为2^9个 1GB大小的吉页 。每个吉页被进一步划分为2^9个2MB大小的巨页。每个巨页再进一步分为2^9个4KB大小的基页。

B. `satp` (Supervisor Address Translation and Protection Register)

一个叫satp（Supervisor Address Translation and Protection，监管者地址转换和保护）的 S模式控制状态寄存器控制了分页系统，其内容如下所示：

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

- MODE：可以开启分页并选择页表级数，8表示Sv39分配方案，0表示禁用虚拟地址映射。
- ASID (Address Space Identifier)： 用来区分不同的地址空间，此次实验中直接置0即可。
- PPN (Physical Page Number)：保存了根页表的物理地址，通常 `PPN = physical address >> 12` 。M模式的程序在第一次进入 S模式之前会把零写入 satp以禁用分页，然后 S模式的程序在初始化页表以后会再次进行satp寄存器的写操作。

- MODE 字段的取值如下图：

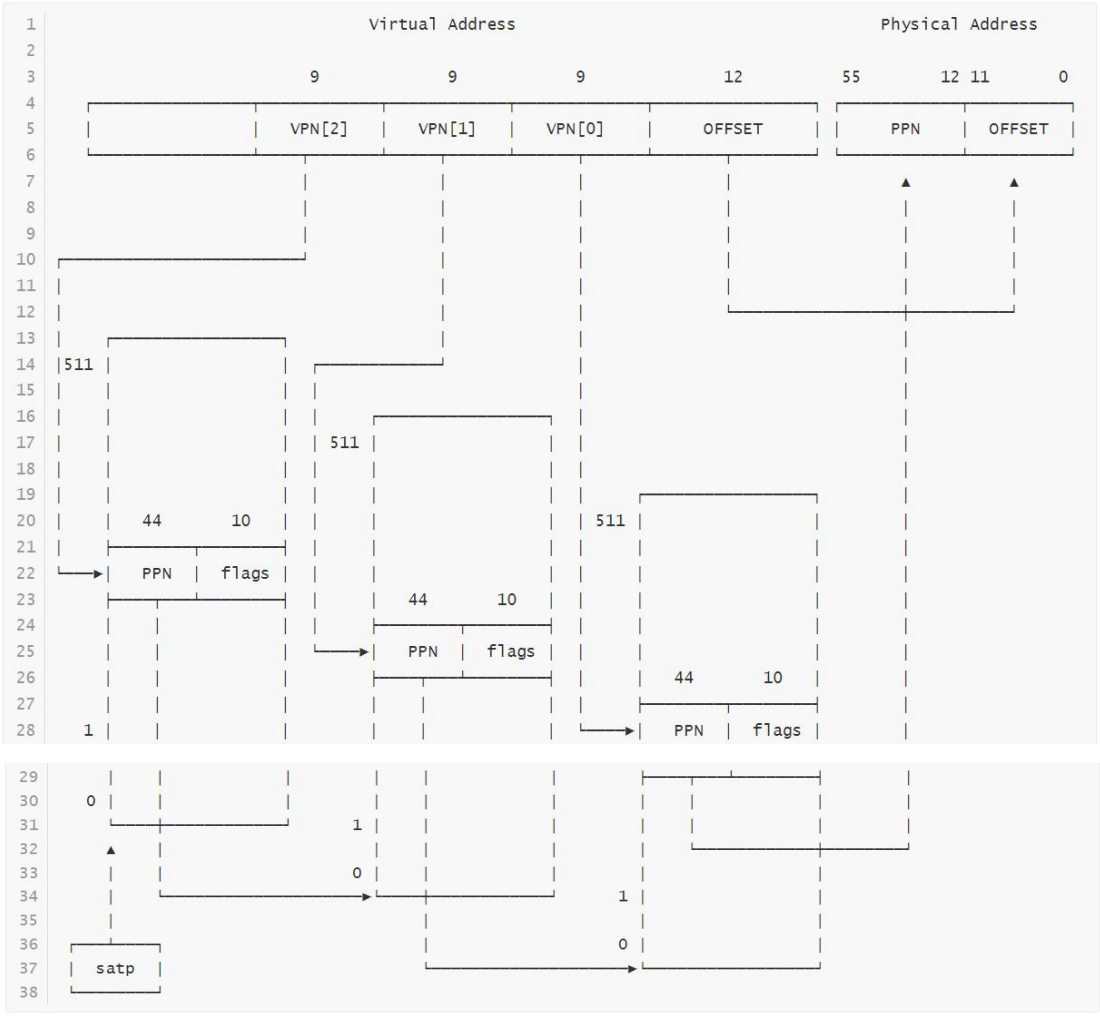
```
                        RV 64
     ------------------------------------------------------------
    |  Value  |  Name  |  Description                            |
    |---------------------------------------------------------------|
    |    0    | Bare   | No translation or protection            |
    |  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               |
     ------------------------------------------------------------
```

2. RISC-V Address Translation Details

虚拟地址翻译为物理地址的完整过程请参考Virtual Address Translation Process，建议仔细阅读，简化版内容如下：

- 1.从satp的 `PPN` 中获取根页表的物理地址。
- 2.通过pagetable中的VPN段,获取PTE。(可以把pagetable看成一个数组，VPN看成下标。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)。
- 3.检查PTE的 `V bit` ，如果不合法，应该产生page fault异常。
- 4.检查PTE的 `RWX` bits,如果全部为0，则从PTE中的PPN[2-0]得到的是下一级页表的物理地址，则回到第二步。否则当前为最后一级页表，PPN[2-0]得到的是最终物理页的地址。
- 5.将得到最终的物理页地址，与偏移地址相加，得到最终的物理地址。
- 6.对齐注意

> Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB megapages and 1 GiB gigapages, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.
> If i > 0 and pte.ppn[i − 1 : 0] ≠ 0, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.



3.2.3 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。

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               |
     ------------------------------------------------------------
```

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）。