Lab4 RV64 虚拟内存管理

赵伊蕾 学号：3200104866

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. 实验过程

2.1 `setup_vm`分配1GB区域

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`。

本实验中，我们需要为每个进程分配地址空间，打开虚拟地址并将物理地址映射到虚拟地址。映射过程一共有两部分，等值映射和线性映射。如下图所示：

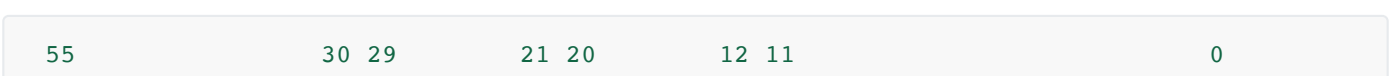
1. **等值映射**（PA == VA）物理地址`0x80000000`之后的内存片段映射到虚拟地址对应的`0x80000000`之后的内存

   等值映射是因为有写代码片段要求物理地址与虚拟地址一样，所以为了减少区分、方便起见，我们将所有的代码都先等值映射一遍。

2. **线性映射**（PA + PV2VA_OFFSET == VA）物理地址`0x80000000`之后的内存片段映射到虚拟地址对应的`0xffffffe000000000`之后的内存段。

   物理地址添上某一位移之后映射到虚拟地址上。

上述内容在setup_vm函数中实现。



```
 ------------------------------------------------------------------------------
|       PPN[2]       |   PPN[1]   |   PPN[0]   |          page offset          |
 ------------------------------------------------------------------------------
                            `Sv39 physical address`

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

我们要设置页表内存的话，首先需要知道PTE的index，然后把虚拟地址映射到对应的PTE中。由上图所见，物理页的`PPN[2]`代表的就是index，所以我们要先取出index值(9bit)。另外PTE的组成如上所示，所以我们取出physical page的前30位，然后放到PTE的0-27位置。特别还需要注意，PTE最后几位是标志位置，所以需要设置读写权限。由于这里`X` `W` `R` `V`都需要设置成1，则我们应该对addr做`addr |= 0xf`的计算，使得最后4位被设置成1。线性映射的思路一致，只不过此时index的计算要用VA来计算，也就是从(VM_START)开始，这样才会把内存映射到`0xffffffe000000000`上。

```c
#define VPN2(va) ((va >> 30) & 0x1ff)   // VPN[2] 9 bit
void setup_vm(void) {
    /* 1. 为early_pgtbl 分配一页内存*/
    memset(early_pgtbl, 0x0, PGSIZE);

    uint64 pa = PHY_START, va = PHY_START, addr;
    int index = VPN2(va);

    /* 2. 直接映射到0x80000000
       V R W X 都设置成1的话，还需要在末尾4位添上0x1111  */
    addr = ((pa >> 30) & 0x3ffff) << 28 | 0xf;
    early_pgtbl[index] = addr;
```

```
    /* 3. 线性映射到0xffffffe000000000*/
    va = VM_START;
    index = VPN2(va);
    early_pgtbl[index] = addr;

}
```

2.2 设置`satp`寄存器

本实验使用的是SV39模式来定义物理地址和虚拟地址。由于我们为系统分配页表，还需要通过一个寄存器来存入页表的头地址，所以我们引入`satp`寄存器，对这一值进行保存。`satp`寄存器结构如下：

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

由于本实验使用的是SV39模式，所以我们需要将`satp`的MODE变量设置成`8`。然后将分配出来的顶级页表的物理页号存入寄存器。我们的物理页的大小为 4KB， 因此PPN可以通过`PA >> 12`得到。

在程序中我们在`relocate`函数中实现对`satp`寄存器实现赋值，完成对页表的映射。

```
relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    li t0, PA2VA_OFFSET
    add ra, ra, t0
    add sp, sp, t0
    # set satp with early_pgtbl
    ## 1. 设置mode为8
    li t0, 8
    slli t0, t0, 60
    ## 2. load PPN 页表起始物理地址
    la t1, early_pgtbl
    ### 32bit -> 44bit
    srli t1, t1, 12
    ## 3. 高位+低位部分相加 -> 组成satp寄存器
    add t0, t0, t1
    csrw satp, t0、
    # flush tlb
    sfence.vma zero, zero
    # flush icache
    fence.i
    ret
```

2.3 `setup_vm_final`实现三层页表分配

首先，需要修改`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。也就是原来的`PHY_END`还需要加上`PA2VA_OFFSET`。有个错误是直接写成`VM_END`，这是因为虚拟地址这一片的长度和物理地址这一片长度并不相等，由于映射关系，所以我们必须利用偏移量来确定虚拟内存中`end`的位置，而不能直接用`VM_END`。

```
void mm_init(void) {
    // 原本只要清理physical page就可以，现在还要把virtual address那部分对应的page也free了
    kfreerange(_ekernel, (char *)(PHY_END + PA2VA_OFFSET));
    printk("...mm_init done!\n");
}
```

在完成`setup_vm_final`函数之前，我们需要先完成`create_mapping`函数，完成地址。函数思路就是处理三层页表，一层层建立PTE并存入页表中，以此将物理地址和虚拟地址映射起来。

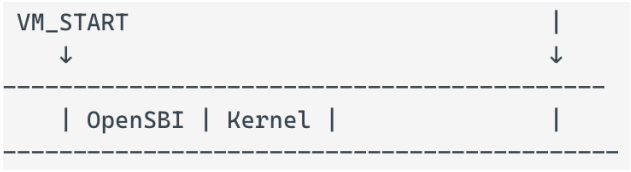
```
void create_mapping(uint64 * root_pgtbl, uint64 va, uint64 pa, uint64 sz, int perm) {
  while(va < end){
    VPN[0] = VPN0(va);     VPN[1] = VPN1(va);    VPN[2] = VPN2(va);
    /* 处理最外层页表 */
    pgtbl[2] = root_pgtbl;       // 第二层页表
    pte[2] = pgtbl[2][VPN[2]];  // 拿出PTE
    if(pte[2] invalid){
      new_page = kalloc();  // 分配
      pte[2] = ((((uint64)new_pg - PA2VA_OFFSET) >> 12) << 10); // set pte
      pte[2] |= 0x1;    // set valid flag
      pgtbl[2][VPN[2]] = pte[2];
    }

    /* 处理第二层页表 */
    /*....... 和上面一致*/
    /* 处理最内层页表 */
    pgtbl[0] = (uint64 *)((pte[1] >> 10) << 12);  // 计算pte
    pte[0] = ((pa >> 12) << 10) | (perm & 15);    // 设置权限 0x1111 = 15
    pgtbl[0][vpn[0]] = pte[0];
    va += PGSIZE;
    pa += PGSIZE;
  }
}
```

完成好物理地址和虚拟地址的三层映射之后，我们需要对完成更细粒度的内存映射，也就是完成对不同section的地址分配保护。这里我们对kernel的`.text`、`.rodata`、other memory三段进行分配，其实就是把code段和变量段单独分配地址进行保护。在程序给出的`vmlinux.lds.S`文件中给出了kernel段内存分布的情况与section分配大小。

```
.text : ALIGN(0x1000){
        _stext = .;
        *(.text.init)
        *(.text.entry)
        *(.text .text.*)  #.text这个section共分配了2*0x1000长度的内存空间
        _etext = .;
    } >ramv AT>ram
```

所以，在setup_vm_final函数中，我们需要对三段进行分配。由于OpenSBI运行在M态，直接使用物理地址，所以我们不需要对OpenSBI再进行映射，只需要从kernel段开始就行。



函数的第二部分主要是将`swapper_page_dir`的物理地址存入`satp`寄存器中，思路和relocate函数一致，只不过要用内敛汇编。

```
void setup_vm_final(void) {

    memset(swapper_pg_dir, 0x0, PGSIZE);
    uint64 va = VM_START + OPENSBI_SIZE, pa = PHY_START + OPENSBI_SIZE;
    // 1. mapping kernel text X|-|R|V     0x1011 = 11
    // text段长度0x2000
    uint64 size = 0x2000;
    create_mapping(swapper_pg_dir, va, pa, size, 11);
    // 2. mapping kernel rodata -|-|R|V   0x0011 = 3
    // rodata段长度0x1000

    // 3. mapping other memory -|W|R|V    0x0111 = 7
    // 其他地址长度 0x80000000 - 0x3000

    // set satp with swapper_pg_dir
    uint64 PG_DIR = (uint64)swapper_pg_dir - PA2VA_OFFSET;
    // 这部分和relocate中的思路一样，只不过这次是从PG_DIR开始写入
    asm volatile(
        "li t0, 8\n"
        "slli t0, t0, 60\n"
        "mv t1, %[PG_DIR]\n"
        "srli t1, t1, 12\n"
        "add t0, t0, t1\n"
        "csrw satp, t0"
        :
        :[PG_DIR]"r"(PG_DIR)
        :"memory"
    );
    // flush TLB
    asm volatile("sfence.vma zero, zero");
```

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

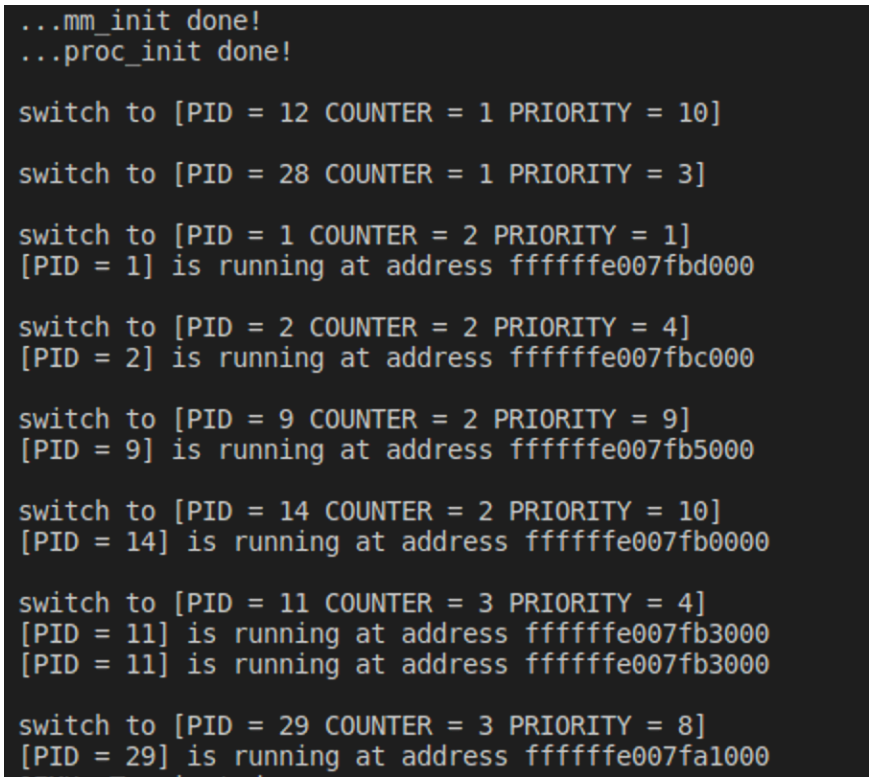
2.4 函数调用

最后，在操作系统启动的时候初始化调用页表映射初始化函数，创建页表。

```
_start:
    ...
    call setup_vm
    call relocate
    call mm_init
    call setup_vm_final
    ...
```

3. 实验结果

在原先Lab3进程调度基础之上，我们输出每个进程运行的虚拟地址。



我们可以观察得到，每个进程将会被分配到0x1000的地址空间，也就是1个page。并且，由于我们在进程初始化的时候对32个进程是从0-31的顺序分配内存的，所以刚好在虚拟内存段中，PID小的进程地址空间排列在下方(地址大)。

```
/* YOUR CODE HERE */
for(int i = 1; i < NR_TASKS; i++){
    task[i] = (struct task_struct *)kalloc();
    task[i]-> state = TASK_RUNNING;
```

4. 思考题

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

我通过`objdump`指令查看我们操作系统对内存分配的情况。我们可以看到几个section的size、起始的虚拟内存地址等。比如`.text`的size是0x1964，从`0xffffffe000200000`内存地址开始；紧接着的section是`.rodata`，被分配到`0xffffffe000202000`。当然，也通过查看`System.map`来进行查看。

```
root@c30e77226e10:/os22fall-stu/src/lab4# objdump -x vmlinux

vmlinux:     file format elf64-little
vmlinux
architecture: UNKNOWN!, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xffffffe000200000

Program Header:
    LOAD off    0x0000000000001000 vaddr 0xffffffe000200000 paddr 0x0000000080200000 align 2**12
         filesz 0x00000000000020e4 memsz 0x00000000000020e4 flags r-x
    LOAD off    0x0000000000004000 vaddr 0xffffffe000203000 paddr 0x0000000080203000 align 2**12
         filesz 0x0000000000000008 memsz 0x0000000000005fa0 flags rw-

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00001964  ffffffe000200000  0000000080200000  00001000  2**12
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000000e4  ffffffe000202000  0000000080202000  00003000  2**12
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000008  ffffffe000203000  0000000080203000  00004000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00004fa0  ffffffe000204000  0000000080204000  00004008  2**12
                  ALLOC
```

2. **为什么我们在 `setup_vm` 中需要做等值映射?**

对某些特殊的物理地址，我们要求其虚拟地址的值与物理地址相同，所以要进行等值映射。比如说在进行页表切换的时候，2个页表的物理地址和虚拟地址要求相同。

在未分页时，我们访问的地址都是物理地址，然而分页后，监管者模式和用户模式下访问的地址就是虚拟地址了。如果没有等值映射，会导致我们在程序启动的时候最终访问的物理地址不再是之前的`0x80000000`，而是某个虚拟地址，那么程序会无法运行下去了。如果不做等值映射，程序一开始运行在低地址0x80000000，但是在做下一条指令PC+4之后，将无法找到高地址；所以需要先做等值映射，保证所有的程序段都能映射过去。

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。**

我这边是以risc-v架构举例的，不同架构的MMU开启策略可能略微有所不同。

通过查看Linux下arch/riscv/kernel/head.S中的relocate_enable_vm函数

```
#ifdef CONFIG_MMU
    .global relocate_enable_mmu
relocate_enable_mmu:
    /* Relocate return address */
```

这个函数将重定位返回的地址，做法如下：

> 1）Compute satp for kernel page tables, but don't load it yet.
>
> 2）Load trampoline page directory, which will cause us to trap stvec if VA != PA, or simply fall through if VA == PA. We need a full fence here because setup_vm() just wrote these PTEs and we need to ensure the new trnaslations are in us.

> 3）Load trampoline page directory, which will cause us to trap stvec if VA != PA, or simply fall through if VA == PA. We need a full fence here because setup_vm() just wrote these PTEs and we need to ensure the new trnaslations are in us.

也就是说，Linux还是由两次开启MMU的操作，**第一次开启**MMU使用的是setup_vm()建立的`trampoline_gd_dir`页表，**第二次开启**MMU使用的是`early_pg_dir`页表。由于Linux中没有做等值映射，所以`trampoline_page_dir`在这其中的作用就很大了，是正常打开MMU的关键。如果VA!=PA，则利用异常处理来切换VA。