Lab 4: Rinux 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 实验内容及要求

- 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
- 实现 SV39 分配方案下的三级页表映射
- 了解页表映射的权限机制，设置不同的页表映射权限
- 阅读背景知识介绍，跟随实验步骤完成实验，以截图的方式记录命令行的输入与输出，
- 如有需要，对每一步的命令以及结果进行必要的解释

3 实验步骤

3.1 准备工程

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
```

3.2 开启虚拟内存映射

3.2.1 setup_vm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
```

此处的实现步骤如下：

- 分别进行等值映射和高值映射
- 取出VA中9位，作为初始页表的index，方法是右移30位后，只对最后的9位做一个and操作
- 赋值，对初始页表中对应的index项，赋的值为物理页号（通过物理地址移动12位，因为4KB=2^12，偏移位即为12）和最后设定好的权限

```
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
```

实现的代码如下

```c
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
    */

    unsigned long PA = 0x80000000;
    //等值映射
    unsigned long VA1 = PA;
    //映射至高位
    unsigned long VA2 = PA + PA2VA_OFFSET;
    int index;
    // 页表的虚拟页号9位,| high bit | 9 bit | 30 bit |,取出这个9bit
    index = (VA1 >> 30) & 0x1ff;
```

```
    //移动30,然后&一下保留有1的，不需要的位数全设为0

    //4kB one table PPN = pa>>12
    // 求出物理页号并设置10位属性（V|R|W|X 为1）
    early_pgtbl[index] = ((PA >> 12) << 10) | 0xf;//先求页号，再把页号后面移动10位，把最后4位设置为1

    //同理
    index = (VA2 >> 30) & 0x1ff;
    early_pgtbl[index] = ((PA >> 12) << 10) | 0xf;

    printk("...setup_vm\n");
}
```

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。以下是寄存器值对应的含义：

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

- MODE：设置为8
- ASID：此次实验置为0
- PPN：顶级页表的物理页号，求法是把初始的PA>>12

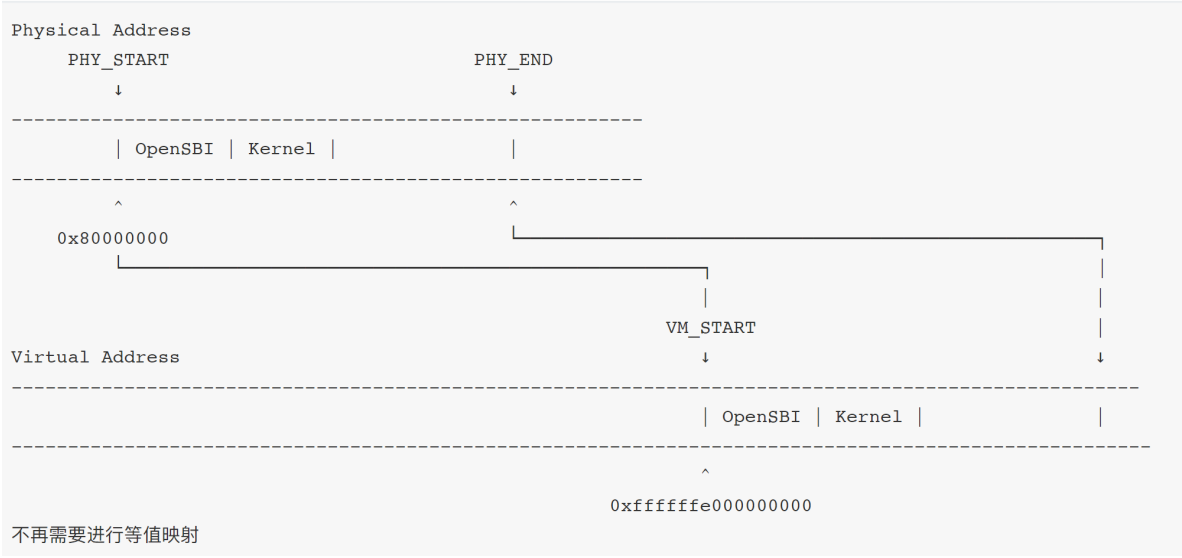
```
relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    # PA2VA_OFFSET: FFFFF_FDF80_00000_0
    li t0, 0xffffffdf80000000
    add ra, ra, t0
    add sp, sp, t0
    #物理变虚拟

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

    # set satp with early_pgtbl
    la t0, early_pgtbl#初始地址
    # >>12
    srli t0, t0, 12#ppn是去除偏移后的
    li t1, 0x8000000000000000#设置对应的mode和中间为0的ASID
    or t0, t0, t1#这里保留有1的部分，同样的是前面8，后面PPN
```

3.2.2 set_vm_final的实现

- 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 。



不再需要进行等值映射

- 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，实现如下：

```
void mm_init(void) {
    kfreerange(_ekernel,  (char *)VM_START + PHY_SIZE);
    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。

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

    // No OpenSBI mapping required

    // mapping kernel text X|-|R|V
    create_mapping(swapper_pg_dir, (uint64)_stext, (uint64)_stext -
PA2VA_OFFSET, (uint64)(_etext - _stext), 11);

    // mapping kernel rodata -|-|R|V
    create_mapping(swapper_pg_dir, (uint64)_srodata, (uint64)_srodata -
PA2VA_OFFSET, (uint64)(_erodata - _srodata), 3);
    //MMU((uint64)_srodata);
    // mapping other memory -|W|R|V
    create_mapping(swapper_pg_dir, (uint64)_sdata, (uint64)_sdata- PA2VA_OFFSET,
PHY_END + PA2VA_OFFSET - (uint64)_sdata, 7);

    // set satp with swapper_pg_dir
    uint64 _satp = 0x8000000000000000 | (((uint64)swapper_pg_dir - PA2VA_OFFSET)
>> 12);
    csr_write(satp, _satp);//模式是8，中间是0，最后是PPN物理页号

    // flush TLB
    asm volatile("sfence.vma zero, zero");
    printk("...setup_vm_final done!\n");
    return;
}
```

3.2.3 create_mapping的实现

虚拟地址与物理地址：

```
 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
```

create_mapping()函数中的每次循环映射一页 4KB 的物理地址到虚拟地址，这里分析二级页表对应的具体流程，其余的和此非常类似：

1. 根据根页表的基地址和VPN2得到 PTE 的内容
2. 若 PTE 的有效位 V 为 0，则申请一块内存空间作为新的二级页表，并将新建的二级页表的地址存放到 PTE 中，并将有效位 V 置 1
3. 根据 PTE 的内容求出二级页表的物理地址，在二级页表中用同样的方法新建一个一级页表或求得一级页表的地址
4. 在一级页表中求得 PTE 的地址，将物理地址存入 PTE，将有效位 V 置 1，根据`perm`改写 RWX 权限位

需要注意的地方是`kalloc()`函数获取的地址为虚拟地址，需要减去`PA2VA_OFFSET`才能作为页表的物理地址。

```c
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm) {
    /*
    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
    */
     // va: 9 | 9 | 9 | 12
    uint64 *second_pgtbl;
    uint64 *third_pgtbl;
    uint64 *p_second_pgtbl;
    uint64 *p_third_pgtbl;
    int i = sz / PGSIZE + 1;//一共多少页

    while (i--)
    {
        int vpn2 = (va >> 30) & 0x1ff;
        int vpn1 = (va >> 21) & 0x1ff;
        int vpn0 = (va >> 12) & 0x1ff;//取出对应的，由上图虚拟地址分布可得
        // top page table
        if (pgtbl[vpn2] & 0x1)//valid为1
        //>>10 propretities <<12 4kB  +PA2VA :va->pa
        //先除掉最后位，再把4KB的偏移加上
        //Page Table Entry一项左移10位，再右移12位，就得到了physical address
            second_pgtbl = (uint64 *)((((uint64)pgtbl[vpn2] >> 10) << 12));
        //若 PTE 的有效位 V 为 0,则申请一块内存空间作为新的二级页表
        //kalloc()函数获取的地址为虚拟地址，需要减去PA2VA_OFFSET才能作为页表的物理地址。
        else
        {
```

```
            second_pgtbl = (uint64 *)kalloc();//虚拟地址
            p_second_pgtbl = (uint64)second_pgtbl - PA2VA_OFFSET;//虚拟到物理的转换
            pgtbl[vpn2] = ((((uint64)second_pgtbl - PA2VA_OFFSET) >> 12) << 10)
| 0x1;
            //页表里把这一项的value设置为1
        }
        // second page table
        if(p_second_pgtbl[vpn1] & 0x1)
            third_pgtbl = (uint64 *)((((uint64)p_second_pgtbl[vpn1] >> 10) <<
12) );
        else
        {
            third_pgtbl = (uint64 *)kalloc();
            p_third_pgtbl = (uint64)third_pgtbl - PA2VA_OFFSET;
            p_second_pgtbl[vpn1] = ((((uint64)third_pgtbl - PA2VA_OFFSET) >> 12)
<< 10) | 0x1;
        }
        // third page table
        if (!(p_third_pgtbl[vpn0] & 0x1) )
        //store pa: >>12:offset and set properties perm
            p_third_pgtbl[vpn0] = ((pa >> 12) << 10) | perm; //设置权限
        //next
        va += PGSIZE;//一页过后加下
        pa += PGSIZE;
    }
}
```

3.3 编译及测试

```
switch to [PID = 27 COUNTER = 1]
[PID = 27] is running. thread space begin at 0xffffffe007fa3000

switch to [PID = 22 COUNTER = 1]
[PID = 22] is running. thread space begin at 0xffffffe007fa8000

switch to [PID = 20 COUNTER = 1]
[PID = 20] is running. thread space begin at 0xffffffe007faa000

switch to [PID = 15 COUNTER = 1]
[PID = 15] is running. thread space begin at 0xffffffe007faf000

switch to [PID = 14 COUNTER = 1]
[PID = 14] is running. thread space begin at 0xffffffe007fb0000

switch to [PID = 10 COUNTER = 1]
[PID = 10] is running. thread space begin at 0xffffffe007fb4000

switch to [PID = 7 COUNTER = 1]
[PID = 7] is running. thread space begin at 0xffffffe007fb7000
```

4. 思考题

1. 验证.text, .rodata段的属性是否成功设置，给出截图。

   在start_kernel函数程序中添加以下代码读取`.text`，`.rodata`段的起始地址中的内容：

```
MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
...setup_vm
...mm_init done!
...setup_vm_final done!
...proc_init done!
_stext = 23
_srodata = 46
```

我们对.text并未设置写功能，尝试写操作后：

```
    printk("_stext = %ld\n", *_stext);
    *_stext = 0;
    printk("_srodata = %ld\n", *_srodata);
```

出现问题：

```
...setup_vm
...mm_init done!
...setup_vm_final done!
...proc_init done!
_stext = 23
SET [PID = 1 COUNTER = 3]
```

2. 为什么我们在setup_vm中需要做等值映射?

   因为在setup_vm_final中建立三级页表时，需要读取页表项中物理页号，转换成物理地址，再去访问下一级页表，如果不做等值映射，直接使用该物理地址将导致内存访问错误。

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。

首先，在set_vm中取消等值映射：

```
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
    */

    unsigned long PA = 0x80000000;
    //映射至高位
    unsigned long VA2 = PA + PA2VA_OFFSET;

    int index;
    //同理
    index = (VA2 >> 30) & 0x1ff;
    early_pgtbl[index] = ((PA >> 12) << 10) | 0xf;

    printk("...setup_vm\n");
}
```

然后 create_map中页表使用虚拟地址：

```
        if(pgtbl[vpn2] & 0x1)
        //>>10 propretities <<12 4kB  +PA2VA :va->pa
            second_pgtbl = (uint64 *)((((uint64)pgtbl[vpn2] >> 10) << 12) );
        //若 PTE 的有效位 V 为 0,则申请一块内存空间作为新的二级页表
        //kalloc()函数获取的地址为虚拟地址，需要减去PA2VA_OFFSET才能作为页表的物理地址。
        else
        {
            second_pgtbl = (uint64 *)kalloc();
            p_second_pgtbl = (uint64)second_pgtbl - PA2VA_OFFSET;
            pgtbl[vpn2] = ((((uint64)second_pgtbl - PA2VA_OFFSET) >> 12) << 10) | 0x1;
        }
        // second page table
        if(p_second_pgtbl[vpn1] & 0x1)
            third_pgtbl = (uint64 *)((((uint64)p_second_pgtbl[vpn1] >> 10) << 12) );
        else
        {
            third_pgtbl = (uint64 *)kalloc();
            p_third_pgtbl = (uint64)third_pgtbl - PA2VA_OFFSET;
            p_second_pgtbl[vpn1] = ((((uint64)third_pgtbl - PA2VA_OFFSET) >> 12) << 10) | 0x1;
        }
        // third page table
        if( !(p_third_pgtbl[vpn0] & 0x1) )
        //store pa: >>12:offset and set properties perm
            p_third_pgtbl[vpn0] = ((pa >> 12) << 10) | perm;

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

使用p_second_pgtbl,p_third_pgtbl的地方改为second ， third。同时取出的时候，加上PA2VA_OFFSET.

结果是一样的：

```
switch to [PID = 27 COUNTER = 1]
[PID = 27] is running. thread space begin at 0xffffffe007fa3000

switch to [PID = 22 COUNTER = 1]
[PID = 22] is running. thread space begin at 0xffffffe007fa8000

switch to [PID = 20 COUNTER = 1]
[PID = 20] is running. thread space begin at 0xffffffe007faa000

switch to [PID = 15 COUNTER = 1]
[PID = 15] is running. thread space begin at 0xffffffe007faf000

switch to [PID = 14 COUNTER = 1]
[PID = 14] is running. thread space begin at 0xffffffe007fb0000
```

5.实验心得

此次实验的实现感觉比较困难，主要是因为理论和实践之中总感觉有一些区别，所以实现的时候特别难以理解，比如虽然上课已经学了多级页表的概念，但是到底如何层层设置就比较难以理解；再比如对于这些位的偏移，一开始就很糊涂不知道到底怎么设置，后来对应着每一个具体地址，一点点的想怎么把地址去转换出需要的值，菜一点点好了起来。这个过程中十分感谢我的舍友和队友对我的帮助。总体而言，在实验的最后，对于需要实现的内容我已经都理解了，但是整体架构却还是有一些模糊，希望这之后能够更充分的体会。