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

- Environment in previous labs

3 实验步骤

3.1 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
  ```

satp寄存器

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

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

实验中MODE设为8

虚拟地址和物理地址

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
```

页表项

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 - Vali
                                                     |   | | | | | | `------ R - Read
                                                     |   | | | | | `-------- W - Writ
                                                     |   | | | | `---------- X - Exec
                                                     |   | | | `------------ U - User
                                                     |   | | `-------------- G - Glob
                                                     |   | `---------------- A - Acce
                                                     |   `------------------ D - Dirt
                                                     `---------------------- Reserved
```

- 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 开启虚拟内存映射

3.2.1 `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)。

完成 `setup_vm` 函数

```
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);
    uint64 index, entry;
    // PA == VA
    // index = PHY_START[38:30]
    index = ((uint64)PHY_START & 0x0000007FC0000000) >> 30;
    // entry[53:28] = PHY_START[55:30], entry[3:0] = 0xF
    entry = ((PHY_START & 0x00FFFFFFC0000000) >> 2) | 0xF;
    early_pgtbl[index] = entry;

    // PA + PV2VA_OFFSET == VA
    // index = VM_START[38:30]
    index = ((uint64)VM_START & 0x0000007FC0000000) >> 30;
    // entry[53:28] = PHY_START[55:30], entry[3:0] = 0xF
    entry = ((PHY_START & 0x00FFFFFFC0000000) >> 2) | 0xF;
    early_pgtbl[index] = entry;
    printk("...setup_vm done!\n");
}
```

early_pgtbl[512]为根页表

首先调用memset将early_pgtbl这一页表初始化为0

第一步，进行等值映射

- 索引值为与物理地址等值的虚拟地址的VPN2，即物理地址的38：30位

  将起始物理地址与38：30位为1的立即数进行与运算，然后右移30位，得到索引值VPN2

- 页表项53：28位是物理地址的PPN2，3：0位置1
  将起始物理地址与55：30位为1的立即数进行与运算，然后右移2位，与0xF进行或运算，将后四位置1，得到页表项

由此完成了等值映射`early_pgtbl[index] = entry;`

第二步，映射至高地址

- 索引值为虚拟地址的VPN2，即虚拟地址的38：30位

  将起始虚拟地址与38：30位为1的立即数进行与运算，然后右移30位，得到索引值VPN2

- 页表项53：28位是物理地址的PPN2，3：0位置1
  将起始物理地址与55：30位为1的立即数进行与运算，然后右移2位，与0xF进行或运算，将后四位置1，得到页表项

由此完成了等值映射`early_pgtbl[index] = entry;`

完成`relocate`函数，设置satp

```
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 # set ra = ra + PA2VA_OFFSET
    add sp, sp, t0 # set sp = sp + PA2VA_OFFSET

    # set satp with early_pgtbl
    la t1, early_pgtbl
    sub t1, t1, t0 # PA of early_pgtbl
    srli t1, t1, 12 # PA >> 12 = PPN
    li t2, 8 # MODE = 8
    slli t2, t2, 60 # MODE: satp[63:60]
    or t1, t1, t2
    csrw satp, t1

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret
```

ra和sp寄存器先偏移PA2VA_OFFSET，进入虚拟地址

再将early_pgtbl装载到satp寄存器

satp的43：0位为PPN，early_pgtbl的物理地址右移12位即可得到PPN

satp的MODE位设为8，通过或运算给satp前4位赋值

最后csrw写入satp寄存器

3.2.2 `setup_vm_final` 的实现

通过三级页表完成128M物理内存的映射，不再需要映射OpenSBI

完成 `setup_vm_final` 函数

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

    // No OpenSBI mapping required
    // mapping kernel text X|-|R|V
    create_mapping((uint64 *)swapper_pg_dir, (uint64)&_stext, (uint64)(&_stext)
- PA2VA_OFFSET, 0x2000, 11);

    // mapping kernel rodata -|-|R|V
    create_mapping((uint64 *)swapper_pg_dir, (uint64)&_srodata, (uint64)
(&_srodata) - PA2VA_OFFSET, 0x1000, 3);

    // mapping other memory -|W|R|V
    create_mapping((uint64 *)swapper_pg_dir, (uint64)&_sdata, (uint64)(&_sdata)
- PA2VA_OFFSET, PHY_SIZE - 0x3000, 7);
```

```
    uint64 satp = (((uint64)swapper_pg_dir - PA2VA_OFFSET) >> 12) | (uint64)0x8
<< 60;
    // set satp with swapper_pg_dir
    __asm__ volatile("csrw satp, %[base]" ::[base] "r"(satp)
                     :);

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

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

    printk("...setup_vm_final done!\n");
    return;
}
```

首先调用memset初始化根页表

调用create_mapping对三段空间分别进行映射，分别是_stext, _srodata, 和其他内存空间 _sdata

create_mapping需要传入根页表的基地址，开始映射处的物理地址和开始映射处的虚拟地址，映射的大小，映射的读写权限

根页表基地址即swapper_pg_dir

开始映射的虚拟地址分别为 _stext, _srodata, _ sdata

对应的物理地址为虚拟地址减去映射偏移量PA2VA_OFFSET

_stext, _srodata,映射大小分别为0x2000 和 0x1000， 其他空间的映射大小为需要映射的总内存减去前两者的内存

读写权限按照注释提示设置

最后设置satp寄存器，同relocate

完成 `create_mapping` 函数

```
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
    */
    unsigned long end_addr = va + sz;
    while (va < end_addr)
    {
        uint64 vpn2 = ((va & 0x7fc0000000) >> 30);
        uint64 vpn1 = ((va & 0x3fe00000) >> 21);
        uint64 vpn0 = ((va & 0x1ff000) >> 12);

        uint64 *pgtbl2 = pgtbl; // PA of first level page

        uint64 *pgtbl1; // PA of second level page
```

```
        // V bit = 0
        if (!(pgtbl2[vpn2] & 1))
        {
            pgtbl1 = (uint64 *)(kalloc() - PA2VA_OFFSET);
            // set V bit = 1
            pgtbl2[vpn2] |= (((uint64)pgtbl1 >> 2) | 1);
        }
        // V bit = 1
        else
        {
            pgtbl1 = (uint64 *)((pgtbl[vpn2] & 0x00ffffffffffffc00) << 2);
        }

        uint64 *pgtbl0; // PA of third level page
        // V bit = 0
        if (!(pgtbl1[vpn1] & 1))
        {
            pgtbl0 = (uint64 *)(kalloc() - PA2VA_OFFSET);
            // set V bit = 1
            pgtbl1[vpn1] |= (((uint64)pgtbl0 >> 2) | 1);
        }
        // V bit = 1
        else
        {
            pgtbl0 = (uint64 *)((pgtbl1[vpn1] & 0x00ffffffffffffc00) << 2);
        }

        // physical page
        // V bit = 0
        if (!(pgtbl0[vpn0] & 1))
        {
            pgtbl0[vpn0] |= (((pa >> 2) & 0x00ffffffffffffc00) | perm);
        }

        va += 0x1000;
        pa += 0x1000;
    }
}
```

通过while循环，按页映射，

虚拟地址的VPN2, VPN1, VPN0分别对应38：30， 29：21， 20：12位，

三个VPN分别是三级页表的索引

对于一二级页表，

如果页表项的V位为0，则需要调用 `kalloc()` 申请一个新页，该函数返回的是新页的虚拟地址，减去PA2VA_OFFSET得到页的物理地址，物理地址右移两位，格式转为页表项，并将页表项的V位赋值为1

如果页表项的V位为1，则分别把页表项左移两位，作为下一级页表的地址

对于三级页表

如果如果页表项的V位为0，直接将物理地址右移两位转为页表项，并通过或运算设置权限位，再与原来的页表项进行或运算，设置新的页表项

三级映射结束后，虚拟地址、物理地址分别加一页的内存0x1000，继续下一页的映射

3.3 编译及测试

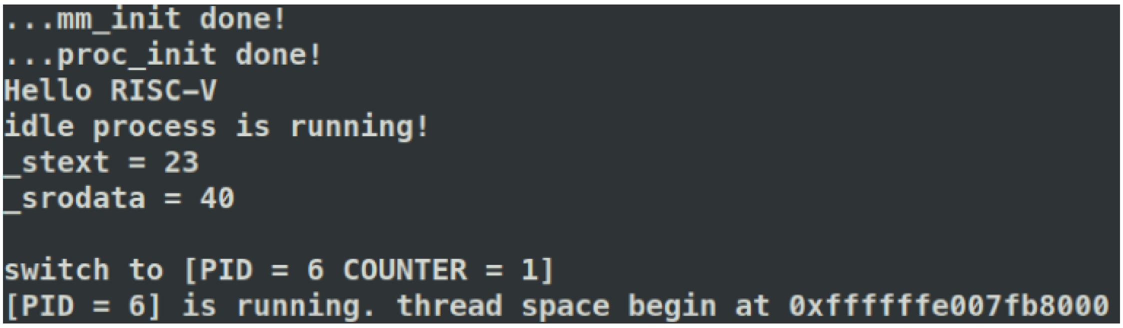
```
SET [PID = 28 COUNTER = 3]
SET [PID = 29 COUNTER = 5]
SET [PID = 30 COUNTER = 10]
SET [PID = 31 COUNTER = 10]

switch to [PID = 6 COUNTER = 2]
[PID = 6] is running. thread space begin at 0xffffffe007fb8000
[PID = 6] is running. thread space begin at 0xffffffe007fb8000

switch to [PID = 8 COUNTER = 2]
[PID = 8] is running. thread space begin at 0xffffffe007fb6000
[PID = 8] is running. thread space begin at 0xffffffe007fb6000

switch to [PID = 13 COUNTER = 2]
[PID = 13] is running. thread space begin at 0xffffffe007fb1000
[PID = 13] is running. thread space begin at 0xffffffe007fb1000

switch to [PID = 16 COUNTER = 2]
[PID = 16] is running. thread space begin at 0xffffffe007fae000
[PID = 16] is running. thread space begin at 0xffffffe007fae000

switch to [PID = 17 COUNTER = 2]
[PID = 17] is running. thread space begin at 0xffffffe007fad000
[PID = 17] is running. thread space begin at 0xffffffe007fad000

switch to [PID = 27 COUNTER = 2]
[PID = 27] is running. thread space begin at 0xffffffe007fa3000
[PID = 27] is running. thread space begin at 0xffffffe007fa3000

switch to [PID = 12 COUNTER = 3]
[PID = 12] is running. thread space begin at 0xffffffe007fb2000
[PID = 12] is running. thread space begin at 0xffffffe007fb2000
[PID = 12] is running. thread space begin at 0xffffffe007fb2000

switch to [PID = 28 COUNTER = 3]
[PID = 28] is running. thread space begin at 0xffffffe007fa2000
[PID = 28] is running. thread space begin at 0xffffffe007fa2000
[PID = 28] is running. thread space begin at 0xffffffe007fa2000

switch to [PID = 1 COUNTER = 4]
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
[PID = 1] is running. thread space begin at 0xffffffe007fbd000
```

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 中添加以下代码

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

   执行结果

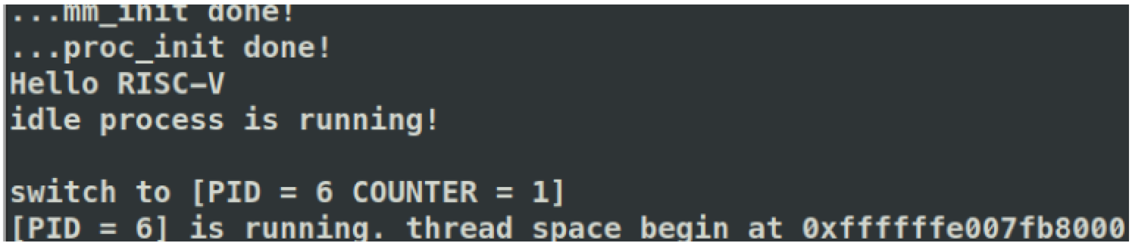


   说明读权限设置成功

   添加如下代码尝试写

```
*_stext = 0;

*_srodata = 0;
```

   执行结果



   不能正常执行，所以没有写权限

2. 为什么我们在 `setup_vm` 中需要做等值映射?
   开启虚拟地址后，不能直接访问物理地址。在三级映射中，页表地址事实上都是物理地址，如果不做等值映射，那么虚拟内存中不存在这一段地址，会出错
   但该实验中由于QEMU的特殊机制，不进行等值映射的情况下程序仍然可以正常运行

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。

   将页表项中读取到的页号计算得到的物理地址转换为虚拟地址，供下一次访问使用。

```
pgtbl[1] = (unsigned long*)(((pte[2] >> 10) << 12) + PA2VA_OFFSET);

pgtbl[0] = (unsigned long*)(((pte[1] >> 10) << 12) + PA2VA_OFFSET);
```