浙江大学

本科实验报告

课程名称：	操作系统
实验名称：	RV64 虚拟内存管理
姓　　名：	蔡雨谦
学　　院：	计算机学院
系：	计算机系
专　　业：	计算机科学与技术
学　　号：	3210102466
指导教师：	寿黎但

2023 年　　11 月　　23 日

浙江大学实验报告

一、 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。

二、 实验环境

- Ubuntu 22.04.2 LTS Windows Subsystem for Linux 2

三、 操作方法与实验步骤

1. 准备工程

从课程提供的 github 仓库获取实验代码框架，准备进行后续实验。本次实验需要修改或完成的文件有：

- Vm.c
- Head.s
- Def.h

需要修改 defs.h，在 defs.h 添加如下内容：

```
#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)
```

从本实验开始我们需要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。在项目顶层目录的 Makefile 中需要做如下更改：

```
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
```

```
ISA=rv64imafd_zifencei
ABI=lp64
```

2. 开启虚拟内存映射。

2.1 setup_vm 的实现

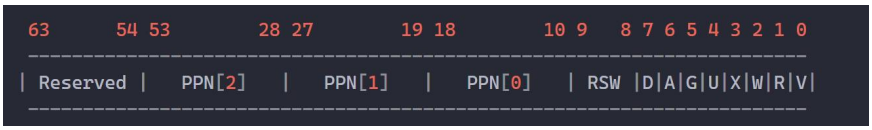
将 0x80000000 开始的 1GB 区域进行两次映射，其中一次是等值映射（PA == VA），另一次是将其映射到 direct mapping area（使

得 PA + PV2VA_OFFSET == VA)。Va 的 vpn2 作为页表的 index，pa 的 ppn2 左移 28 位再加上权限设置构成 PTE。

```
    memset(early_pgtbl, 0x0, PGSIZE);
    uint64 pa = PHY_START;
    //等值映射
    uint64 va = pa;
    //vpn2作为index
    int index = (va >> 30) & 0x1ff;
    //ppn2转化为pte
    early_pgtbl[index] = ((pa >> 30) << 28) | 0b1111;

    //映射到direct mapping area
    va = pa + PA2VA_OFFSET;
    index = (va >> 30) & 0x1ff;
    early_pgtbl[index] = ((pa >> 30) << 28) | 0b1111;
    printk("setup_vm done.\n");
```

虚拟地址、物理地址和 PTE 的结构如下：



2.2 setup_vm_final 的实现

设置 text 段。权限为可执行、可读。

```
  // mapping kernel text X|-|R|V
  va = (uint64)_stext;
  pa = va - PA2VA_OFFSET;
  sz = _etext - _stext;
  perm = 0b1011;
  create_mapping(swapper_pg_dir, va, pa, sz, perm);
```

设置 rodata 段，权限为可读。

```
  // mapping kernel rodata -|-|R|V
  va = (uint64)_srodata;
  pa = va - PA2VA_OFFSET;
  sz = _erodata - _srodata;
  perm = 0b11;
  create_mapping(swapper_pg_dir, va, pa, sz, perm);
```

设置 data（memory）段，权限为可读写。

```
// mapping other memory -|W|R|V
va = (uint64)_sdata;
pa = va - PA2VA_OFFSET;
sz = _edata - _sdata;
perm = 0b111;
create_mapping(swapper_pg_dir, va, pa, sz, perm);
```

设置 satp 寄存器，刷新 TLB 和缓存。

```
// set satp with swapper_pg_dir

// YOUR CODE HERE
csr_write(satp, (uint64)swapper_pg_dir);

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

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

2.3 create_mapping 的实现

每次 while 循环进行一个页的映射设置，即将物理地址保存在三级页表中，每次循环后将物理地址和虚拟地址。

Vpn2 作为一级页表的 index，进入二级页表。如果该 index 表示的页无效就重新分配。

```
// vpn2
uint64 vpn2 = ((va)>>30) & 0x1ff;
//if not valid, alloc a new page
if ((pgtbl[vpn2] & 1) == 0)
{
    page = (uint64*)kalloc();
    pgtbl[vpn2] = (1 | ((uint64)page >> 12) << 10);
}
table = &(pgtbl[vpn2]);
```

VPN1 作为二级页表的 index，进入三级页表。同样页表无效时就分配页表。

```
// vpn1
uint64 vpn1 = ((va)>>21) & 0x1ff;
if ((table[vpn1] & 1) == 0)
{
    page = (uint64*)kalloc();
    table[vpn1] = (1 | ((uint64)page >> 12) << 10);
}
table = &(table[vpn1]);
```

三级页表中在 index 为 vpn0 的位置存放物理地址。

```
// vpn0
uint64 vpn0 = ((va)>>12) & 0x1ff;
if ((table[vpn0] & 1) == 0)
{
    page = (uint64*)kalloc();
    table[vpn0] = (1 | ((uint64)page >> 12) << 10);
}
table[vpn0] = ((pa >> 12) << 10) | perm;
```

2.4 relocate 的实现

Relocate 函数用于设置 satp 以及跳转到虚拟地址。

将 PA2VA_OFFSET 的值保存到 t0 寄存器，将 ra 和 sp 都对应到虚拟地址。

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

    #   YOUR CODE HERE   #
    #addi立即数最大0x7ff
    addi t0, zero, 0xff
    slli t0, t0, 8
    addi t0, t0, 0xff
    slli t0, t0, 8
    addi t0, t0, 0xff
    slli t0, t0, 8
    addi t0, t0, 0xdf
    slli t0, t0, 8
    addi t0, t0, 0x80
    slli t0, t0, 24
    #t0 == 0xffffffdf80000000 (PA2VA_OFFSET)

    add ra, ra, t0
    add sp, sp, t0
```

Satp 结构如下。

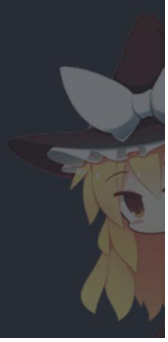
early_pgtbl 是虚拟地址，减去 PA2VA_OFFSET 得到物理地址，由物理地址获取 PPN。

```
    # set satp with early_pgtbl
#  63      60 59                 44 43                                 0
#  ------------------------------------------------------------------------
# |   MODE    |         ASID         |                PPN                |
#  ------------------------------------------------------------------------
    #   YOUR CODE HERE   #
    #MODE = 8
    addi t2, zero, 8
    #ASID ( Address Space Identifier ) :  此次实验中直接置 0 即可。
    slli t2, t2, 60
    #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
    la t1, early_pgtbl
    sub t1, t1, t0
    srli t1, t1, 12
    add t1, t1, t2
    csrw satp, t1

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret
```



3. 测试

```
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
setup_vm done.

...mm_init done!
task_init
...proc_init done!
2022 Hello RISC-V
idle process is running!


switch to [PID = 1 COUNTER = 4]

dummy
[PID = 1] is running. auto_inc_local_var = 1
[PID = 1] is running. auto_inc_local_var = 2
[PID = 1] is running. auto_inc_local_var = 3
[PID = 1] is running. auto_inc_local_var = 4


switch to [PID = 3 COUNTER = 8]

dummy
[PID = 3] is running. auto_inc_local_var = 1
[PID = 3] is running. auto_inc_local_var = 2
[PID = 3] is running. auto_inc_local_var = 3
[PID = 3] is running. auto_inc_local_var = 4
[PID = 3] is running. auto_inc_local_var = 5
[PID = 3] is running. auto_inc_local_var = 6
[PID = 3] is running. auto_inc_local_var = 7
[PID = 3] is running. auto_inc_local_var = 8


switch to [PID = 2 COUNTER = 9]
```

四、思考题

1. 验证 .text，.rodata 段的属性是否成功设置，给出截图。

   程序能够顺利运行说明.text 可执行。

   尝试对.text 和.rodata 进行读操作：

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

   结果：

```
2022 Hello RISC-V
idle process is running!
_stext = 27
_srodata = 10
```

   说明可读。

   尝试进行写操作：

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

   结果：

```
2022 Hello RISC-V
idle process is running!
_stext = 0
_srodata = 0
```

应该不可写才对，但是这里却写成功了，说明写权限没有设置正确，但是我没有弄清楚为什么。

2. 为什么我们在 setup_vm 中需要做等值映射?

   在没有做等值映射时，程序无法运行，但是我发现删去 head.S 里的 fence.i 后就能正常运行了。我在互联网上找到的这道思考题的相关解答是建立三级页表时，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。但是这一解释并没有解释清楚为什么删去 fence.i 就没有问题了。

   在实验验收时，经过助教的解释，我大致明白了。在 csrw satp, t1 之后就已经开启了虚拟地址，而此时的 PC 值是之前 PC 值加 4，仍是物理地址，如果不进行等值映射，物理地址的 PC 会导致内存访问错误。

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。

   不在 setup_vm 中做等值映射的话，就要在 create_mapping 中创建页表时直接使用虚拟地址。即：pgtbl[vpn1] = (1 | ((uint64)page >> 12) << 10)修改为 pgtbl[vpn1] = (1 | ((uint64)page >> 12) << 10) + PA2VA_OFFSET，table[vpn0] = (1 | ((uint64)page >> 12) << 10)修改为 table[vpn0] = (1 | ((uint64)page >> 12) << 10) + PA2VA_OFFSET。

五、讨论心得

1、head.S 文件里，la sp, boot_stack_top 执行后 sp 的值变成虚拟地址的值，导致无法正常运行，于是我将 sp 设置成 lab2 里 la sp, boot_stack_top 执行后 sp 的值，即 0x80205000，结果就可以执行了。

2、在 vm_setup_final 函数里，我应该正确设置了权限，但是思考题中的结果表明写权限并未正确设置，没有想明白原因。

```
// mapping kernel text X|-|R|V
va = (uint64)_stext;
pa = va - PA2VA_OFFSET;
sz = _etext - _stext;
perm = 0b1011;
create_mapping(swapper_pg_dir, va, pa, sz, perm);

// mapping kernel rodata -|-|R|V
va = (uint64)_srodata;
pa = va - PA2VA_OFFSET;
sz = _erodata - _srodata;
perm = 0b11;
create_mapping(swapper_pg_dir, va, pa, sz, perm);
```