Lab 3: RV64 虚拟内存管理

张志心 3210106357

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 相关宏

```
// arch/riscv/include/defs.h
#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 VPN0(addr) (((uint64)(addr) >> 12) & 0x1ff)
#define VPN1(addr) (((uint64)(addr) >> 21) & 0x1ff)
#define VPN2(addr) (((uint64)(addr) >> 30) & 0x1ff)
```

3.2 开启虚拟内存映射

3.2.1 setup_vm 的实现

将 `0x80000000` 开始的 **1 GB** 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` （`PA + PA2VA_OFFSET == VA`）。

```c
// arch/riscv/kernel/vm.c

/* early_pgtbl: 用于 setup_vm 进行 1GB 的 映射。 */
unsigned long early_pgtbl[512] __attribute__((__aligned__(0x1000)));

void setup_vm(void) {
    memset(early_pgtbl, 0x0, PGSIZE);
    uint64 PA = PHY_START;
    uint64 VA = VM_START;
    early_pgtbl[VPN2(PA)] = ((PA >> 12) << 10) | 0xf;
    early_pgtbl[VPN2(VA)] = ((PA >> 12) << 10) | 0xf;
    printk("...setup_vm done!\n");
}
```

通过 `relocate` 函数，完成对 `satp` 的设置，以及跳转到对应的虚拟地址上。

```asm
# arch/riscv/kernel/head.S
.extern start_kernel
    # ...
_start:
    li t0, 0xffffffdf80000000 # PA2VA_OFFSET
    la sp, boot_stack_top
    sub sp, sp, t0
    call setup_vm
    call relocate
    call mm_init
    call task_init
    # ...
    jal start_kernel # jump to start_kernel

relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET
    li t0, 0xffffffdf80000000 # PA2VA_OFFSET
    add ra, ra, t0
    add sp, sp, t0

    # set satp with early_pgtbl
    la t0, early_pgtbl
    li t1, 0xffffffdf80000000 # PA2VA_OFFSET
    sub t0, t0, t1
    li t1, 0x8000000000000000
    srli t0, t0, 12
    or t0, t0, t1
    csrw satp, t0

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret

    #...
```

3.2.2 setup_vm_final 的实现

修改 `mm.c` 中初始化函数接收的起始结束地址为虚拟地址。

```
// arch/riscv/kernel/mm.c

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

对所有物理内存(128M)进行映射，并设置正确的权限。

```
// arch/riscv/kernel/vm.c

/* swapper_pg_dir: kernel pagetable 根目录， 在 setup_vm_final 进行映射。 */
unsigned long  swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));

extern char _stext[];
extern char _srodata[];
extern char _sdata[];

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,
(_srodata - _stext), 0xb);

    // mapping kernel rodata -|-|R|V
    create_mapping(swapper_pg_dir, (uint64)_srodata, (uint64)_srodata -
PA2VA_OFFSET, (_sdata - _srodata), 0x3);

    // mapping other memory -|W|R|V
    create_mapping(swapper_pg_dir, (uint64)_sdata, (uint64)_sdata - PA2VA_OFFSET,
PHY_SIZE - (_sdata - _stext), 0x7);

    // set satp with swapper_pg_dir
    uint64 _satp = (((uint64)swapper_pg_dir - PA2VA_OFFSET) >> 12) |
(0x8000000000000000);
    csr_write(satp, _satp);

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

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

创建三级页表映射关系。

```c
// arch/riscv/kernel/vm.c

/**** 创建多级页表映射关系 ****/
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm) {
    uint64 *now_pgtbl, *nex_pgtbl;
    for(int i = 0, num = sz / PGSIZE; i < num; ++i, va += PGSIZE, pa += PGSIZE) {
        uint64 vpn2 = VPN2(va), vpn1 = VPN1(va), vpn0 = VPN0(va);
        now_pgtbl = pgtbl;
        // first level
        if(now_pgtbl[vpn2] & 0x1) {
            now_pgtbl = (uint64 *)((((uint64)now_pgtbl[vpn2] >> 10) << 12)
                                   + PA2VA_OFFSET);
        } else {
            nex_pgtbl = (uint64 *)kalloc();
            now_pgtbl[vpn2] = ((((uint64)nex_pgtbl - PA2VA_OFFSET) >> 12) << 10) |
0x1;
            now_pgtbl = nex_pgtbl;
        }
        // second level
        if(now_pgtbl[vpn1] & 0x1) {
            now_pgtbl = (uint64 *)((((uint64)now_pgtbl[vpn1] >> 10) << 12)
                                   + PA2VA_OFFSET);
        } else {
            nex_pgtbl = (uint64 *)kalloc();
            now_pgtbl[vpn1] = ((((uint64)nex_pgtbl - PA2VA_OFFSET) >> 12) << 10) |
0x1;
            now_pgtbl = nex_pgtbl;
        }
        // third level
        now_pgtbl[vpn0] = ((pa >> 12) << 10) | perm | 0x1;
    }
}
```

在 `head.S` 中调用 `setup_vm_final`

```
# arch/riscv/kernel/head.S
# ...
_start:
    # ...
    call setup_vm
    call relocate
    call mm_init
    call setup_vm_final # <---- addition
    call task_init
    # ...
```

3.3 编译及测试

`make run` 后输出截图如下（使用 **SJF** 进程调度算法）：

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
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!
...proc_init done!
2022 Hello RISC-V
switch to [PID = 8 COUNTER = 1]
[PID = 8] is running. auto_inc_local_var = 1
switch to [PID = 7 COUNTER = 2]
[PID = 7] is running. auto_inc_local_var = 1
[PID = 7] is running. auto_inc_local_var = 2
switch to [PID = 14 COUNTER = 2]
[PID = 14] is running. auto_inc_local_var = 1
[PID = 14] is running. auto_inc_local_var = 2
switch to [PID = 11 COUNTER = 3]
[PID = 11] is running. auto_inc_local_var = 1
[PID = 11] is running. auto_inc_local_var = 2
[PID = 11] is running. auto_inc_local_var = 3
switch to [PID = 13 COUNTER = 3]
[PID = 13] is running. auto_inc_local_var = 1
[PID = 13] is running. auto_inc_local_var = 2
[PID = 13] is running. auto_inc_local_var = 3
switch to [PID = 1 COUNTER = 4]
```

4 思考题

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

   在 `trap_handler` 中增加调试输出（在非时钟中断的时候输出）：

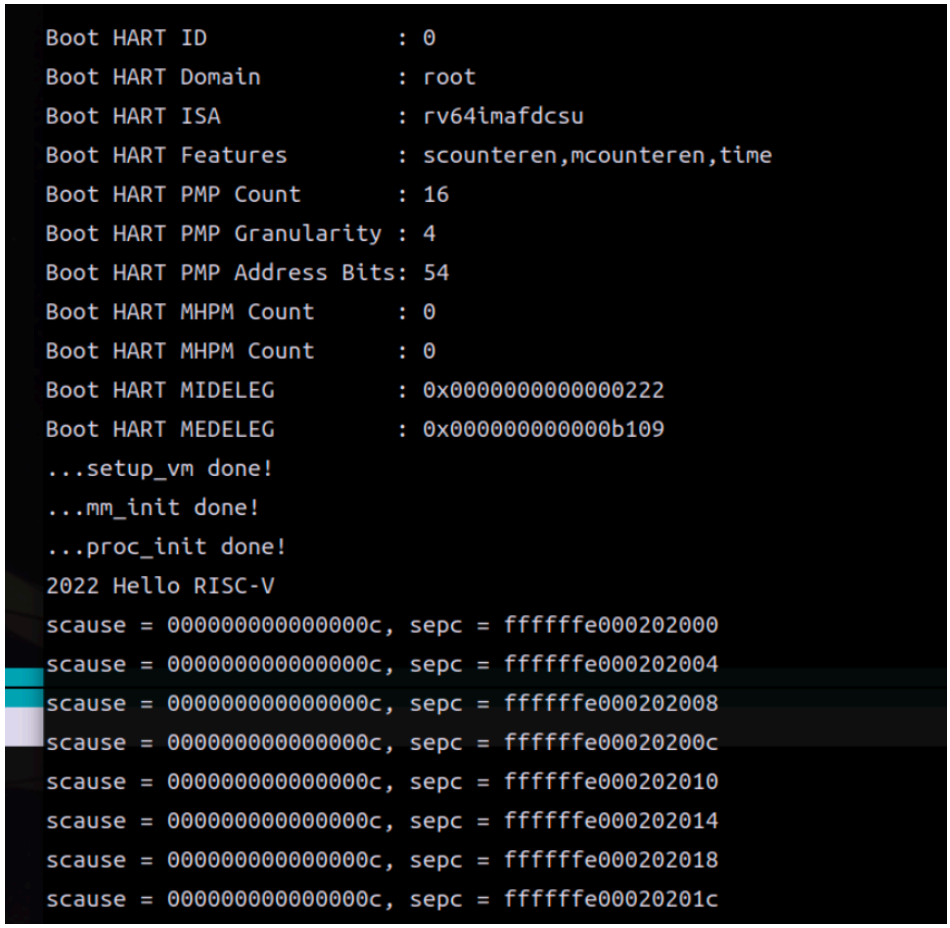
   ```
   printk("scause = %lx, sepc = %llx\n", scause, sepc);
   ```

   - `.text` 段的执行权限正常，因为程序可以正常执行。
   - 测试 **`.rodata`** 段的执行权限：

     在 `start_kernel` 中加入：

     ```
     asm volatile("jal _rodata");
     ```

     测试结果：其中 `scause = 0xc` 说明为 **Instruction Page Fault**。`.rodata` 段没有执行权限。

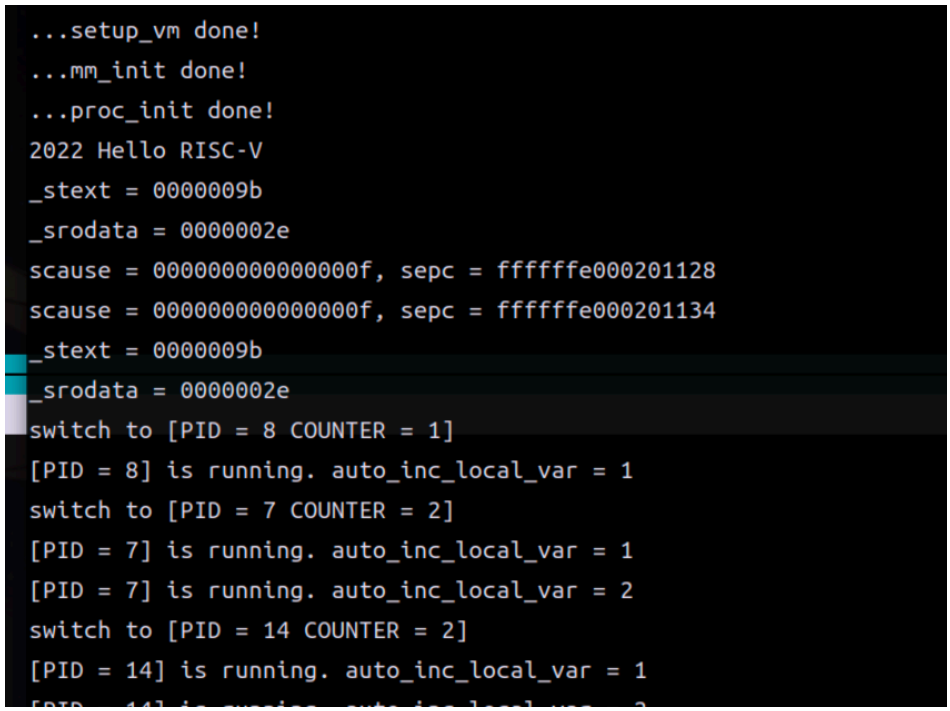
     

   - 测试 **`.text`** 段和 **`.raodata`** 段的读写权限：

     在 `start_kernel` 中加入：分别对 `_stext` 和 `_srodata` 段进行一次读写。

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

测试结果：读操作正常，写操作异常（`scause = 0xf` 说明为 **Store/AMO Page Fault**）。`.text` 段和 `.rodata` 段有读权限但没有写权限。



2. 为什么我们在 `setup_vm` 中需要做等值映射?

   因为写入 **satp** 后，**PC** 仍然在物理地址上，继续执行必须要访问物理地址，如果没有等值映射就会导致访存失败。

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。

   使用异常处理机制来令**satp**修改后能够正确执行下一条指令，我们在修改**satp**之前，先将**stvec**指向**satp**下一条指令的虚拟地址，这样**stap**修改之后，下一条指令会触发访存异常，从而跳到**stvec**指示的值，这样就达成了继续执行的目的，此后将**stvec**改回即可。