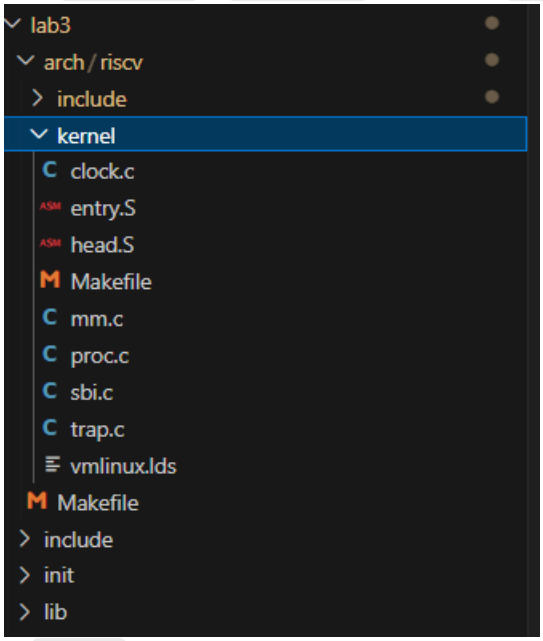
浙江大学实验报告

课程名称：操作系统
实验项目名称：RV64 虚拟内存管理
学生姓名：展翼飞 学号：3190102196
电子邮件地址：1007921963@qq.com
实验日期： 2024年 10 月 30 日

一、实验内容

1. 准备工程

- 使用`git fetch`与`git merge`从仓库同步`lab3 vmlinux.lds`代码，并从`lab2`拷贝之前完成的代码



- 在`defs.h`中添加本实验虚拟内存相应的宏定义

```
11
12  #define OPENSBI_SIZE (0x200000)
13
14  #define VM_START (0xffffffe000000000)
15  #define VM_END (0xffffffff00000000)
16  #define VM_SIZE (VM_END - VM_START)
17
18  #define PA2VA_OFFSET (VM_START - PHY_START)
```

- 修改根目录`Makefile`的`ISA`，支持指令屏障，使用刷新缓存的指令扩展

```
8   #rv64imafd 是 RISC-V 64 位基础指令集，而 rv64imafd_zifencei 是在其基础上增加了对指令屏障的支持
9   #该指令用于确保在执行特定指令之前，前面的所有指令都已经完成。它通常用于多核系统中的内存一致性
10  ISA     :=  rv64imafd_zifencei
```

2.PIE

- 因为 GOT 表里的地址都是最终的虚拟地址，所以在 kernel 启用虚拟地址之前，一切从 GOT 表取出的地址都是错的，这种情况下需要手动给 `la` 得到的地址减去 `PA2VA_OFFSET` 才能得到正确的物理地址。

为了避免这种情况，需要在 Makefile 的 `CF` 中加一个 `-fno-pie` 强制不编译出 PIE 的代码

```
TEST_SCHED  :=  0
INCLUDE :=  -I $(shell pwd)/include -I $(shell pwd)/arch/riscv/include
CF      :=  -fno-pie -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-builti
CFLAG   :=  $(CF) $(INCLUDE) -DTEST_SCHED=$(TEST_SCHED)
```

3.开启虚拟内存映射

3.1 `setup_vm` 的实现

1. **setup_vm:** 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`）。

```
void setup_vm() {
    /*
    * 1. 由于是进行 1GiB 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 1GiB 的区域
    * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    **/

    // 将虚拟地址 0x80000000 等值映射到物理地址 0x80000000
    // virtual address = 0x80000000;  >> 30(1gib 映射) => VPN = 2
    //PPN[2] = 0x80000000 >> 12 = 0x80000
    uint64_t PPN = (uint64_t)0x80000 << 10;  //0x80000为opensbi起始地址的物理块号 PPN字段从PTE的第10位开始，因此左移10位
    early_pgtbl[2] = (uint64_t)(PPN | 15U);

    /*
    PTE 解释:
    - 0x20000000U: 表示实际物理地址 0x80000000 >> 2，用于指向目标物理页。
    - 15U: 设置 PTE 权限位 V | R | W | X
    */

    // 将虚拟地址 0xffffffe000000000 等值映射到物理地址 0x80000000
    // virtual address = 0xffffffe000000000;  >> 30 :  VPN = 384
    // 即虚拟地址 0xffffffe000000000 对应 early_pgtbl 的第 384 个条目
    // PTE 值 (0 | 0x20000000U | 15U) 保持不变
    early_pgtbl[384] = (uint64_t)(PPN | 15U);

    /*
    此条目使得虚拟地址 0xffffffe000000000 等值映射到物理地址 0x80000000。
    这是一个备用的高位内核地址，用于某些 RISC-V 系统的高地址访问。
    */

    // 输出调试信息，表示 setup_vm 函数执行完成
    printk("...setup_vm done!\n");
}
```

因为每个页表项进行1GiB的映射，所以对需要映射的虚拟地址右移30位可以得到该1GIB映射的页表项编号Index，等值映射和 `direct mapping area` 分别对应根页表2和384项。
而它们映射到的物理页都是OpenSBI的起始地址页，也即物理地址0x8000000，将它右移12位得到SV39模式的4KiB物理页号0x8000，由于PTE中物理页号存储从第十位开始，放入PTE中仍需左移10位，得到需要放入PTE中的PPN。
完成这些后，我们可以对根页表中的第2和384项进行设置，完成虚拟地址的映射，只需要将PPN与0xf即15进行按位或，将页表项VRWX位置1，完成页表项设置。

2. **relocate:** 完成`relocate`编写实现设置satp寄存器与跳转到虚拟地址，在`head.s`中`_start`起始处调用`setup_vm`与`relocate`

```
relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    li t0, 0xffffffe000000000  # 加载 VM_START 到寄存器 t0
    li t1, 0x0000000080000000  # 加载 PHY_START 到寄存器 t1

    sub t2, t0, t1             # 计算偏移量 (VM_START - PHY_START)，将结果存储到 t2
    add ra, ra, t2         # 将 ra 加上 PA2VA_OFFSET
    add sp, sp, t2         # 将 sp 加上 PA2VA_OFFSET


    # need a fence to ensure the new translations are in use
    sfence.vma zero, zero

    # set satp with early_pgtbl

    addi t0,x0,1
    slli t0,t0,63   #0x8000000000000000 采用sv39分页模式
    la t1,early_pgtbl
    #sub t1,t1,t2
    srli t1,t1,12   #  Get PPN 根页表物理页号
    add t0,t0,t1
    csrw satp,t0

    ret
```

首先需要将ra与sp，也即pc返回值与堆栈指针加上`PA2VA_OFFSET`，使其从`relocate`返回后运行在虚拟地址之上，并使用`sfence.vma`指令确保虚拟地址转换启用。
随后将根页表物理地址右移12位得到根页表物理页号，而0x8000000000000000代表64位寄存器最高四位为8，在`satp`中代表使用SV39分页模式，二者相加并设置`satp`寄存器。

3. **修改mm_init:**
   修改 `arch/riscv/kernel/mm.c` 的 `mm_init` 函数，将结束地址调整为虚拟地址，保证`kalloc`正常运行

```
void mm_init(void) {
    // 将从 _ekernel 到 PHY_END 之间的内存全部标记为空闲
    kfreerange(_ekernel, (char *)VM_START+PHY_SIZE);
    printk("...mm_init done!\n");  // 打印内存管理初始化完成的日志
}
```

3.2 `setup_vm_final` 的实现

1. **`setup_vm_final` 的实现**
   在映射部分中，我们需要通过`system.map`寻找对应段的大小，可知`kernel`的`text` 和`rodata`分别需要映射2页和1页，而剩余的内存经计算共32768-（2 + 1）=32765页，调用`create_mapping`函数创建对应页表项的虚拟内存映射，随后以SV39模式通过内联汇编将根页表物理页号写入`satp`：

```c
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_t)&_stext,(uint64_t)(&_stext)-PA2VA_OFFSET,2,11);

    // mapping kernel rodata -|-|R|V
    create_mapping(swapper_pg_dir,(uint64_t)&_srodata,(uint64_t)(&_srodata)-PA2VA_OFFSET,1,3);

    // mapping other memory -|W|R|V
    // 128MB = 128 * 1024 = 131072 kB  131072 / 4 = 32768 sz
    create_mapping(swapper_pg_dir,(uint64_t)&_sdata,(uint64_t)(&_sdata)-PA2VA_OFFSET,32765,7);

    // set satp with swapper_pg_dir

    uint64_t tmp_satp = 0x8000000000000000;
    //printk("%lx\n",swapper_pg_dir);
    uint64_t swapper_p = (uint64_t)swapper_pg_dir - PA2VA_OFFSET;
    //printk("%lx\n",swapper_p);
    uint64_t swapper_ppn = swapper_p >> 12;
    //printk("%lx\n",swapper_ppn);
    tmp_satp += swapper_ppn;
    //printk("%lx\n",tmp_satp);

    asm volatile("csrw satp,%[src]"::[src]"r"(tmp_satp):);
    //printk("SUCCESS\n");


    // flush TLB
    asm volatile("sfence.vma zero, zero");
    //printk("SUCCESSFUL1\n");

    // flush icache
    asm volatile("fence.i");
    //printk("SUCCESSFUL2\n");
    return;
}
```

在`head.s`中调用`setup_vm_final`：

```
_start:
    # Initial stack 设置堆栈指针
    la sp, boot_stack_top               # 加载 boot_stack_top 地址到 sp 中，设置堆栈顶

    call setup_vm                       # 进行等值映射和direct mapping area 映射

    call relocate                       # 完成对 satp 的设置，以及跳转到对应的虚拟地址

    #进行内存初始化
    call mm_init

    call setup_vm_final                 # 进行三级页表虚拟内存映射
```

2. `create_mapping`的实现

```c
/* 创建多级页表映射关系 */
create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, uint64_t perm) {
    /*
     * pgtbl 为根页表的基地址
     * va, pa 为需要映射的虚拟地址、物理地址
     * sz 为映射的大小，单位为4kb
     * perm 为映射的权限（即页表项的低 8 位）
     *
     * 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
    **/
    printk("create mapping for va = %llx, pa = %llx\n", va, pa);

    //sz的单位是4kb,即sz代表要映射的页面个数
    //判断是leaf page table: pte.v = 1 ,并且pte.r或者pte.x其中一个为1
    for(int i=0;i<sz;i++){

        //设置第二级页表
        uint64_t vpn_2 = (va >> 30) & 0x1ff;     //va中的31-39位为vpn_2 0x1ff = b0001 1111 1111
        uint64_t *level2;

        //printk("%d\n",vpn_2);
        if(pgtbl[vpn_2] & 0x1 ){  //存在二级页表索引
            uint64_t ppn = (pgtbl[vpn_2] >> 10) & 0xfffffffffff;   //右移12位 取PTE中44位PPN
            level2 = (uint64_t *)(ppn << 12);     //得到对应二级页表地址 offset 为0
            //printk("a:%lx\n",level2);
        }else{
            level2 = (uint64_t *)kalloc() - PA2VA_OFFSET;   //新申请一个页用于存储页表

            //printk("b:%lx\n",level2);
            pgtbl[vpn_2] = (uint64_t)level2 >> 2; //地址中的ppn位于12-55位，pte中的ppn位于10-53位
            pgtbl[vpn_2] |= 0x1;                  //设置valid
        }
        //设置第三级页表
        uint64_t vpn_1 = (va >> 21) & 0x1ff;
        uint64_t *level3;

        if(level2[vpn_1] & 0x1 ){
            uint64_t ppn = (level2[vpn_1] >> 10) & 0xfffffffffff;   //右移12位之后取44位 ppn2: 26位 ppn1: 9位 ppn0: 9位
            level3 = (uint64_t *)(ppn << 12);
            //printk("c:%lx\n",level3);
        }else{
            level3 = kalloc() - PA2VA_OFFSET;
            //printk("d:%lx\n",level3);
            level2[vpn_1] = (uint64_t)level3 >> 2;
            level2[vpn_1] |= 0x1;
        }

        //第三级页表映射到物理页
        uint64_t vpn_0 = (va >> 12) & 0x1ff;
        if( !(level3[vpn_0] & 0x1) ){
            level3[vpn_0] = (uint64_t)pa >> 2;
            level3[vpn_0] |= perm;
        }
        //0x1000 B = 16^3 B = 2^12 B = 4kB,即一个页面的大小
        va = va + 0x1000;
        pa = pa + 0x1000;
    }
    //printk("end\n");
}
```

4.编译与测试

尚未debug完成

二、思考题

1. **验证 `.text`，`.rodata` 段的属性是否成功设置，给出截图。**
   完成`setup_vm`后，编译并查看`system.map`，可知段成功设置在虚拟地址上

```
ffffffe0002001b4 T __dummy
ffffffe0002001c4 T __switch_to
ffffffe000207000 B _ebss
ffffffe000203008 D _edata
ffffffe000207000 B _ekernel
ffffffe000202290 R _erodata
ffffffe000201fc0 T _etext
ffffffe000204000 B _sbss
ffffffe000203000 D _sdata
ffffffe000200000 T _skernel
ffffffe000202000 R _srodata
ffffffe000200000 T _start
ffffffe000200000 T _stext
ffffffe000200094 T _traps
ffffffe000204000 B boot_stack
ffffffe000205000 B boot_stack_top
```

2. **为什么我们在 `setup_vm` 中需要做等值映射？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。你需要回答以下问题：**
   - **本次实验中如果不做等值映射，会出现什么问题，原因是什么；**
     如果不进行等值映射，那么在 `setup_vm` 函数完成设置页表并切换页表后，将开启虚拟地址，但程序运行的代码仍是物理地址，被视为虚拟地址后此段可能在页表中不存在有意义的物理地址映射。
   - **简要分析 Linux v5.2.21 或之后的版本中的内核启动部分（直至 `init/main.c` 中 `start_kernel` 开始之前），特别是设置 satp 切换页表附近的逻辑；**
   - **回答 Linux 为什么可以不进行等值映射，它是如何在无等值映射的情况下让 pc 从物理地址跳到虚拟地址；**
   - **Linux v5.2.21 中的 `trampoline_pg_dir` 和 `swapper_pg_dir` 有什么区别，它们分别是在哪里通过 satp 设为所使用的页表的；**
   - **尝试修改你的 kernel，使得其可以像 Linux 一样不需要等值映射。**

三、问题与心得

1. 调试时在运行到`relocate`设置好 `satp` 寄存器之前，只可以使用**物理地址**来打断点。因为符号表、`vmlinux.lds` 里面记录的函数名的地址都是虚拟地址。如果使用符号的虚拟地址设置断点会导致程序不会遇到设置到的断点(虚拟地址相比物理地址高出offset)。可以在目录下编译生成的 `vmlinux.asm` 中找到所有代码的虚拟地址，然后将其转换成物理地址，再使用 `b *<addr>` 命令设置断点，设置 satp 之后，才可以使用虚拟地址打断点，同时之前设置的物理地址断点也会失效，需要删除。

2. 为什么setupvm_final对于kernel 的text rodata段与剩余内存要分开调用create mapping？
   因为他们映射的物理内存页需要设置的访问权限不同。