浙江大学实验报告

一、实验内容

准备工程

根据实验指导，从git仓库中将需要的代码拷贝至本地，并在需要改动的文件内增加相应的代码。

开启虚拟内存映射

`setup_vm`的实现

根据实验指导，在这一步我们需要完成`setup_vm`函数并修改`head.S`。

首先完成`setup_vm`函数。根据实验指导要求，这个函数完成的是建立等值映射与映射至高地址的线性映射。注意到用于映射的页表在定义时声明了对齐`__attribute__((__aligned__(0x1000)))`，因此其首地址必定是`0x1000`的整数倍，可以直接用于建立页表。注意到这里只进行一次映射，不需要用到多级页表，且一个entry对应1GB的区域。用一个右移运算与一个与运算提取虚拟地址的中间9个bit作为index。用类似的方法，提取物理地址的高位并与四个权限位组合成一个entry。因为只需要建立两个映射，简单填入两项即可。

```
unsigned long  early_pgtbl[512] __attribute__((__aligned__(0x1000)));
void setup_vm(void)
{
    early_pgtbl[VM_START >> 30 & 0x1ff] = ((PHY_START >> 30 & 0x3ffffff) << 28)
+ (V_MASK | R_MASK | W_MASK | X_MASK);
    early_pgtbl[PHY_START >> 30 & 0x1ff] = ((PHY_START >> 30 & 0x3ffffff) << 28)
+ (V_MASK | R_MASK | W_MASK | X_MASK);
    return;
}
```

完成`setup_vm`后不难知道，需要先把映射的页表建立好，才能把页表的地址赋值给`satp`寄存器。因此在`_start`中，先调用`setup_vm`，再调用`relocate`进行赋值，然后对内存进行初始化，可以保证整个系统运行时都采用虚拟地址。在`relocate`函数中，根据注释的指示，首先对返回地址和栈指针加上线性映射的偏移。此处我们尝试了几种方法想直接将C语言中的宏定义赋值给寄存器，但是都失败了，最后只好直接手动键入偏移的数值。

其次完成对`satp`寄存器的赋值。首先把代表着mode的8赋值给`t0`并左移60位，完成对mode的设置；然后获取页表的物理页号，设置在`t0`的最低位作为PPN；最后，清除TLB，把`t0`赋值给`satp`并返回即可。

```
_start:
    ...
    call setup_vm
    call relocate
    call mm_init
    ...
relocate:
    # set ra = ra + PA2VA_OFFSET
```

```
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    # ld t0, PA2VA_OFFSET
    li t0, 0xffffffdf80000000
    add ra,ra,t0
    add sp,sp,t0

    # set satp
    # mode
    li t0, 0x8
    slli t0,t0,60
    # PPN
    la t1, early_pgtbl
    srli t1,t1,12
    add t0,t0,t1
    sfence.vma zero,zero
    csrw satp,t0
    ret
```

`setup_vm_final` 的实现

首先明确在这一步要完成的任务：在上一步已经建立起了简单的线性映射。在这里，还需要对除OpenSBI外的所有物理内存空间建立三级页表映射。通过上一步，整个系统都开始采用虚拟地址工作，因此不再需要建立等值映射。

由于 `head.S` 中先调用了内存初始化 `mm_init` 再调用 `setup_vm_final`，我们需要先检查内存初始化的函数。查看 `mm_init` 函数，可以得知这个函数释放了一定的空间，便于未来使用。分析得知，需要释放的是物理内存中不属于OpenSBI也不属于kernel的部分，否则系统就无法运行了。需要注意的是，这里传给 `kfreerange()` 的两个参数都应是虚拟地址。将该函数修改如下：

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

`create_mapping()` 建立三级映射。参数中的 `va` 和 `pa` 实际上是需要建立映射的空间的虚拟首地址和物理首地址，空间的大小由 `sz` 指定。这里我们先假设 `va`、`pa` 和 `sz` 都是对齐的，即都是0x1000的倍数。

空间的大小可能不止一个页，因此需要按页循环建立三级映射。在前两级页表，我们要注意的是，要建立的页表项可能已经存在了；检查有效位，如果页表项已经存在，我们直接利用已有的页表项即可。对于第三级页表，只要输入的参数不同，就不会出现重复的页表项，组合物理地址页号和权限组成entry即可。在这里，前两级页表的R、X和W权限位都为0，系统才会认为它们不是最后一级页表。传入的权限参数只出现在最后一级页表的entry。

```
unsigned long swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
void create_mapping(unsigned long* pgtbl, unsigned long va, unsigned long pa,
unsigned long sz, int perm)
{
    // TODO: setup the three level page table map
    // !: sz may be over a pagesize
    for(int i=0; sz>0; i++){
        int VPN[3];
        unsigned long *addr2, *addr3;
        unsigned long cur_va = va + i*PGSIZE;
        // !: one va however multiple pa?
```

```
        VPN[0] = (cur_va) >> 12 & 0x1ff;
        VPN[1] = (cur_va) >> 21 & 0x1ff;
        VPN[2] = (cur_va) >> 30 & 0x1ff;

        // first level
        if(pgtbl[VPN[2]] & V_MASK)
        {
            //!: physical?
            addr2 = (unsigned long)((pgtbl[VPN[2]] >> 10) << 12) + PA2VA_OFFSET;
        }
        else
        {
            addr2 = kalloc(); // !: ?
            pgtbl[VPN[2]] = ((((unsigned long)addr2 - PA2VA_OFFSET) >> 12) <<
10) | V_MASK;
        }
        // second level
        if(addr2[VPN[1]] & V_MASK)
        {
            //!: physical?
            addr3 = (unsigned long)((addr2[VPN[1]] >> 10) << 12) + PA2VA_OFFSET;
        }
        else
        {
            addr3 = kalloc(); // !: ?
            addr2[VPN[1]] = ((((unsigned long)addr3 - PA2VA_OFFSET) >> 12) <<
10) | V_MASK;
        }
        // third level
        addr3[VPN[0]] = (((pa >> 12) + i) << 10) | perm;
        // map next page
        sz -= PGSIZE;
    }
}
```

在`setup_vm_final`里，对kernel中的四个section和内存中不属于OpenSBI也不属于kernel的空间建立三级映射，并再一次设置`satp`寄存器。调用`create_mapping`时，确保传入的参数都是对齐的，我们调用了`PGROUNDUP`这一宏定义函数来确保对齐。这非常重要，因为如果不确保对齐，可能出现两种情况：第一种是同一个页被重复建立映射；第二种是有空白区域没有建立映射（这一种情况在这里基本不会出现）。最后通过内联汇编给`satp`赋值，这一部分汇编指令与`relocate`中的基本一样。

```
void setup_vm_final(void){
    memset(swapper_pg_dir,0x0,PGSIZE);
    create_mapping(swapper_pg_dir,(unsigned long)_stext,
                   (unsigned long)(_stext - PA2VA_OFFSET),
                   (unsigned long)PGROUNDUP(_etext - _stext),
                   X_MASK | R_MASK | V_MASK);
    create_mapping(swapper_pg_dir,(unsigned long)_srodata,
                   (unsigned long)(_srodata - PA2VA_OFFSET),
                   (unsigned long)PGROUNDUP(_erodata - _srodata),
                   R_MASK | V_MASK);
    create_mapping(swapper_pg_dir,(unsigned long)_sdata,
                   (unsigned long)(_sdata - PA2VA_OFFSET),
                   (unsigned long)PGROUNDUP(_edata - _sdata),
                   W_MASK | R_MASK | V_MASK);
    create_mapping(swapper_pg_dir,(unsigned long)_sbss,
```

```
                    (unsigned long)(_sbss - PA2VA_OFFSET),
                    (unsigned long)PGROUNDUP(_ebss - _sbss),
                    W_MASK | R_MASK | V_MASK);

    create_mapping(swapper_pg_dir,(unsigned long)PGROUNDUP(_ekernel),
                    (unsigned long)PGROUNDUP(_ekernel - PA2VA_OFFSET),
                    (unsigned long)( PHY_END -  PGROUNDUP(_ekernel -
PA2VA_OFFSET)),
                    W_MASK | R_MASK | V_MASK);

    // set the stap register and clear tlb
    // !: swapper_pgtbl is virtual?
    __asm__ volatile(
        // mode
        "li a0, 0x8\n"
        "slli a0, a0, 60\n"
        // PPN
        "addi a1, %[PPN], 0\n"
        "srli a1, a1, 12\n"
        "add a0, a0, a1\n"
        "csrw satp, a0\n"
        "sfence.vma zero, zero\n"
        :   // output
        : [PPN] "r" ((unsigned long)swapper_pg_dir-PA2VA_OFFSET)   // input
        : "a0", "a1"   // modification
        );

    return;
}
```

编译及测试

运行程序，可以看到正常输出。首先测试SJF调度。可以看到第一次调度时，剩余时间最短的第一个进程被调度，然后是第三个进程，第三个进程结束后是第五个进程……由以下两个图可以看到，线程执行的顺序与剩余执行时间的升序一致。需要注意的是，由于一开始的进程是`idle`，根据实验指导要求，此时直接进行调度，`schedule()`会被调用并且第一个线程会有输出。

直接运行程序，可以看到线程可以正常切换，程序正常运行。每个线程的首地址也是虚拟地址。分析各个线程的虚拟首地址，两个线程的线程号的差乘上0x1000，正好是这两个线程的首地址的差。这说明每个线程占用了不超过一个页的大小，也说明页的分配是一种单向的简单分配。事实上，查看`mm.c`中的函数，可以看到空闲页通过链表链接，每次申请一个页时，将链表中的最后一个空闲页分配出去。

```
cheung@cheung: ~
**********
...proc_init done!32
2021 Hello RISC-V 3190103058 3190102214
kernel is running!

switch to [PID = 6 COUNTER = 1]
[PID = 6] is running. thread space begin at = ffffffe007fb9000

switch to [PID = 8 COUNTER = 1]
[PID = 8] is running. thread space begin at = ffffffe007fb7000

switch to [PID = 13 COUNTER = 1]
[PID = 13] is running. thread space begin at = ffffffe007fb2000

switch to [PID = 16 COUNTER = 1]
[PID = 16] is running. thread space begin at = ffffffe007faf000

switch to [PID = 17 COUNTER = 1]
[PID = 17] is running. thread space begin at = ffffffe007fae000

switch to [PID = 27 COUNTER = 1]
[PID = 27] is running. thread space begin at = ffffffe007fa4000

switch to [PID = 12 COUNTER = 2]
[PID = 12] is running. thread space begin at = ffffffe007fb3000
[PID = 12] is running. thread space begin at = ffffffe007fb3000

switch to [PID = 28 COUNTER = 2]
[PID = 28] is running. thread space begin at = ffffffe007fa3000
[PID = 28] is running. thread space begin at = ffffffe007fa3000

switch to [PID = 1 COUNTER = 3]
[PID = 1] is running. thread space begin at = ffffffe007fbe000
```

```
cheung@cheung: ~
switch to [PID = 2 COUNTER = 3]
[PID = 2] is running. thread space begin at = ffffffe007fbd000
[PID = 2] is running. thread space begin at = ffffffe007fbd000
[PID = 2] is running. thread space begin at = ffffffe007fbd000

switch to [PID = 9 COUNTER = 3]
[PID = 9] is running. thread space begin at = ffffffe007fb6000
[PID = 9] is running. thread space begin at = ffffffe007fb6000
[PID = 9] is running. thread space begin at = ffffffe007fb6000

switch to [PID = 14 COUNTER = 3]
[PID = 14] is running. thread space begin at = ffffffe007fb1000
[PID = 14] is running. thread space begin at = ffffffe007fb1000
[PID = 14] is running. thread space begin at = ffffffe007fb1000

switch to [PID = 11 COUNTER = 4]
[PID = 11] is running. thread space begin at = ffffffe007fb4000
[PID = 11] is running. thread space begin at = ffffffe007fb4000
[PID = 11] is running. thread space begin at = ffffffe007fb4000
[PID = 11] is running. thread space begin at = ffffffe007fb4000

switch to [PID = 29 COUNTER = 4]
[PID = 29] is running. thread space begin at = ffffffe007fa2000
[PID = 29] is running. thread space begin at = ffffffe007fa2000
[PID = 29] is running. thread space begin at = ffffffe007fa2000
[PID = 29] is running. thread space begin at = ffffffe007fa2000

switch to [PID = 15 COUNTER = 5]
[PID = 15] is running. thread space begin at = ffffffe007fb0000
[PID = 15] is running. thread space begin at = ffffffe007fb0000
[PID = 15] is running. thread space begin at = ffffffe007fb0000
[PID = 15] is running. thread space begin at = ffffffe007fb0000
[PID = 15] is running. thread space begin at = ffffffe007fb0000
```

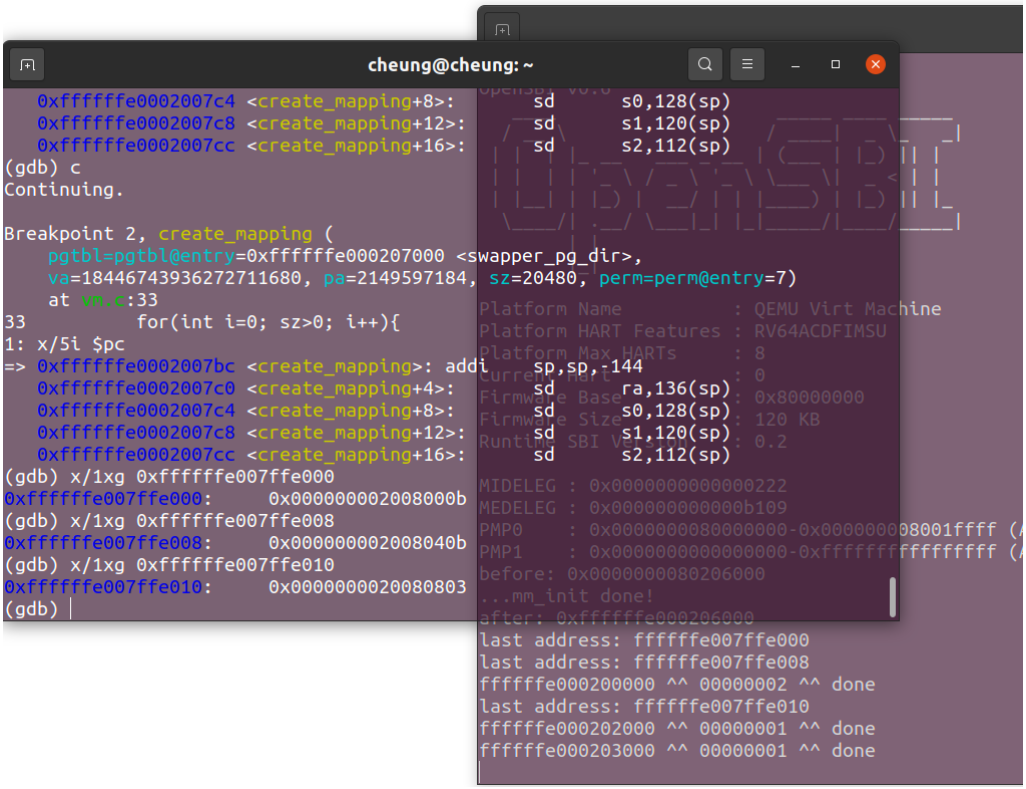
二、思考题

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

首先在`create_mapping`中输出这两个段的最后一级页表中各页表项的地址，如下：

```
after: 0xffffffe000206000
last address: ffffffe007ffe000
last address: ffffffe007ffe008
ffffffe000200000 ^^ 00000002 ^^ done
last address: ffffffe007ffe010
ffffffe000202000 ^^ 00000001 ^^ done
```

根据地址，在gdb中调试时查看改地址的内容如下：



可以看到，输出的三个entry分别是`.text`的两个entry和`.rodata`的一个entry。手动进行地址映射，并检查最低的四位权限位，可以知道设置正确。

2. 为什么我们在`setup_vm`中需要做等值映射?

因为在设置`satp`之后，`rip`以及部分上下文还在物理地址空间中，如果不建立等值映射，首先会出现的就是`rip`找不到`csrw`的下一条指令。如果函数还需要用到其他上下文，也会出现问题。

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`relocate`的代码修改如下。主要思想是利用缺页异常，在给`satp`赋值之前先设置异常处理函数的入口为`aaa`。在执行`csrw satp,t0`后，TLB已被清空且没有进行等值映射，缺页异常触发，程序进入`aaa`。在`aaa`中，直接返回到调用`relocate`的地方，这样则可以不必设置等值映射。

```
relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    # ld t0, PA2VA_OFFSET
    li t0, 0xffffffdf80000000
    add ra,ra,t0
    add sp,sp,t0

    # set stvec to return to the caller of relocate
    la t1, aaa
    add t1,t1,t0
    csrw stvec,t1
    # set satp
    # mode
    li t0, 0x8
    slli t0,t0,60
    # PPN
    la t1, early_pgtbl
    srli t1,t1,12
    add t0,t0,t1
    sfence.vma zero,zero
```

```
    csrw satp,t0
    # flush tlb
aaa:
    sfence.vma zero,zero
    ret
```

三、讨论心得

本次实验的最主要问题即是要注意地址的对齐，建立映射等步骤根据实验指导中的图例与背景知识进行即可。但是在一开始，我们没有考虑到地址对齐的问题，程序无法正常跑起来。经过讨论，我们修正这个问题后，程序即可正常运行。思考题第三题是一个很有难度的问题，我们思考无果，最后查到了Linux的源代码了解到了现在这种方法。但是在使用这种方法的时候，我们有一个疑问，就是系统的权限模式似乎变得没有那么明确了。我们不是很确定使用这种方法后，系统返回到我们创建的进程中时是处于S-mode还是U-mode。