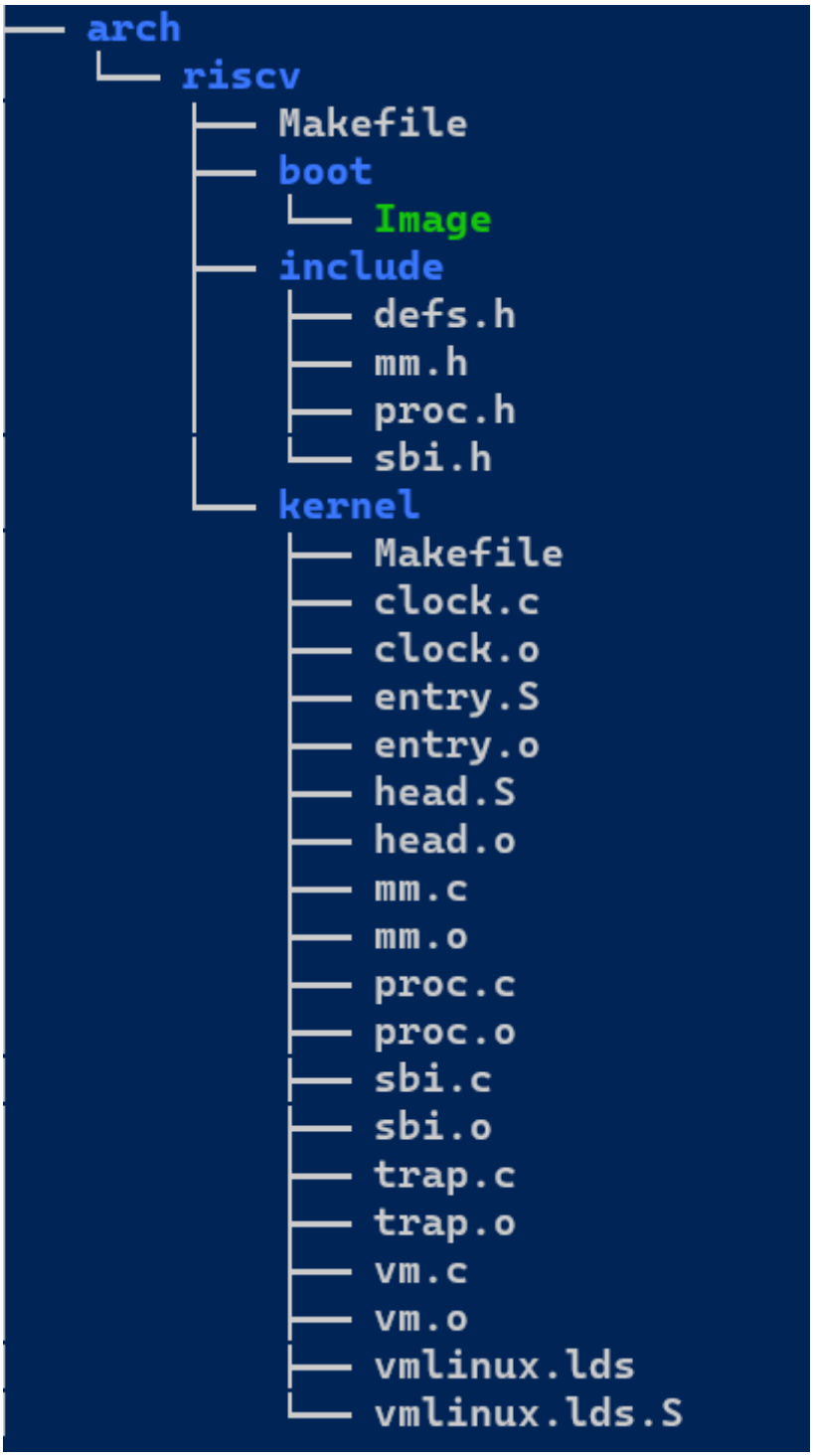
# OS lab4

## 1. 准备工程

按照要求在`defs.h`中添加以下宏定义：

```
7
8   #define PHY_START 0x0000000080000000
9   #define PHY_SIZE  128 * 1024 * 1024 // 128MB,  QEMU 默认内存大小
0   #define PHY_END   (PHY_START + PHY_SIZE)
1
2   #define PGSIZE 0x1000 // 4KB
3   #define PGROUNDUP(addr) ((addr + PGSIZE - 1) & (~(PGSIZE - 1)))
4   #define PGROUNDDOWN(addr) (addr & (~(PGSIZE - 1)))
5
6   #define OPENSBI_SIZE (0x200000)
7
8   #define VM_START (0xffffffe000000000)
9   #define VM_END   (0xffffffff00000000)
0   #define VM_SIZE  (VM_END - VM_START)
1
2   #define PA2VA_OFFSET (VM_START - PHY_START)
3
```

将 `vmlinux.lds.S`, `Makefile` 放在正确的地方




# 2. 开启虚拟内存映射

## 2.1 setup_vm

首先，我们需要将 0x80000000 开始的 1GB 区域进行一次是等值映射 ( PA == VA )，和一次将其映射至高地址 ( PA + PV2VA_OFFSET == VA )

```
void setup_vm(void)
{
    unsigned long true_early_pgtbl = (unsigned long)early_pgtbl - PA2VA_OFFSET;
    memset((void *)true_early_pgtbl, 0, PGSIZE);
    ((unsigned long *)true_early_pgtbl)[getVPN(PHY_START, 2)] = (0xf) |
((getPPN(PHY_START, 2)) << 28);
    ((unsigned long *)true_early_pgtbl)[getVPN(VM_START, 2)] = (0xf) |
((getPPN(PHY_START, 2)) << 28);
    printk("setup_vm done!\n");
}
```

由于reallocate后均使用虚拟地址，所以我们在`setup_vm`中必须要对原本的给定的页表地址减去地址偏移，使其变为`early_pgtbl`真正的物理地址，然后我们再为真正的页表分配1GB空。

这之后，我们根据提示，将va，pa的30~38位取出分别作为`early_pgtbl`的index。

而对于页表项，我们要首先将0~3位置为1，代表将Page Table Entry 的权限 V | R | W | X 开启。然后将va,pa的的28~53位取出，左移28位作为PTE的PPN[2]，其余的位均置为0.

```
 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|
 -------------------------------------------------------------------------
```

完成了`setup_vm`函数的实现后，我们需要通过`relocate`函数，完成对`satp`的设置，以及跳转到对应的虚拟地址。

在relocate中，我们首先需要将`ra`，`sp`寄存器加上一个偏移量，使其变为虚拟地址。

```
li t0, 0xffffffe000000000
li t1, 0x0000000080000000
sub t0, t0, t1
add ra, ra, t0
add sp, sp, t0
```

然后我们需要将`satp`设置为`early_pgtbl`,注意t1一开始用la赋值时`early_pgtbl`是虚拟地址，需要减去一个偏移量来得到对应的物理地址再右移12位得到PPN。然后将8左移60位与之前所得相加，写入`satp`。最后刷新 TLB，刷新 icache。

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

```
    # PPN
    la t1, early_pgtbl
    sub t1,t1,t0
    srli t1, t1, 12
    # MODE
    li t2,8
    slli t2, t2, 60
    # set satp
    add t1,t1,t2
    csrw satp,t1
```

```
    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i
```

## 2.2 `setup_vm_final` 的实现

首先修改`mm_init`，由于reallocate后我们全部使用逻辑地址，所以要为PHY_END`加上一个偏移量使其变为逻辑地址。


我们再来来讲一下这部分的核心函数`create_mapping`，我们需要根据传入的根页表地址，虚拟地址，物理地址，size，和perm(末尾几位标志位)来为对应的虚拟地址和物理地址间创建映射关系。

```
void create_mapping(uint64 *pgtbl, uint64 virtualAddress, uint64
physicalAddress, uint64 size, int perm)
```

更仔细地说，我们在这个函数中干了以下几件事情：

1. 根据要分配的`size`与固定的`PGSIZE`（4KB）相除得出要分配的**页数**
2. 以`va`的VPN[2]作为根页表的初始index，确定`pgtbl`中的储存着我们要的二级页表的Entry的地址
3. 再根据`va`的VPN[1]作为二级页表的初始index，确定`二级页表`中的储存着我们要的三级页表的Entry的地址
4. 将我们要建立映射的物理地址`pa`的PPN与参数中给定的末位标志位`perm`组合后写入3中的Entry
5. 每完成一页的映射，需要对根页表、二级页表、三级页表的index进行更新。如三级页表index每满512（表示当前三级页表已满），需要二级页表index指向一个新的页。
6. 同时，我们在读取页表项前，需要读取该页表项的末尾位来确定该页是否`valid`，如果当前页表项无效，需要使用`kalloc`来新分配一页，并将新分配的页的地址的PPN左移10位组成有效的页表项。
7. 完成以上工作后，一轮循环结束，4kb的虚拟地址通过三级页表映射到了对应的物理地址上。我们回到2，直到所有虚拟页都完成映射。

```
void create_mapping(uint64 *pgtbl, uint64 virtualAddress, uint64
physicalAddress, uint64 size, int perm) {
    int page_num = size / (uint64)PGSIZE; //一共需要映射的页数
    int pgtblIndex = getVPN(virtualAddress,2);//目前已分配到的根页表Entry的Index
    int secondLevelPageTableEntryIndex = getVPN(virtualAddress,1);
    int thirdLevelPageTableEntryIndex = getVPN(virtualAddress,0);
    uint64* secondEntryAddress;//根页表的Entry中储存的PPN，也就是我们要的二级页表地址
    uint64* thirdEntryAddress;//二级页表的Entry中储存的PPN，也就是我们要的三级页表地址

    printk("create_mapping begin! page_num:%d\n",page_num);

    for(int i = 0; i < page_num; i++){

        //step1: 获取根页表存的二级页表的Entry的地址
```

```
        uint64* nowPgtblEntryAddress = pgtbl + pgtblIndex;//pgtbl的某个Entry(储存我们要的二级页表)的地址

        //step2: 在根页表上找到对应的二级页表的地址，对应的Entry没有内容的话就new一个二级页表存到Entry

        //检查是否在一个空Entry上
        if(isPageNotValid(nowPgtblEntryAddress)){
            //new一个二级页表来填充这块区域
            secondEntryAddress = (uint64*)kalloc();
            *nowPgtblEntryAddress = (getPPNFromPA((uint64)secondEntryAddress - PA2VA_OFFSET) << 10) + 0x1;
        }
        else{
            secondEntryAddress = (getPPNFromEntry(*nowPgtblEntryAddress) << 12) + PA2VA_OFFSET;
        }

        //step3: 获取二级页表的存三级页表的Entry的地址

        //二级页表的某个Entry(储存我们要的三级页表)的地址
        uint64* nowSecondEntryAddress = secondEntryAddress + secondLevelPageTableEntryIndex;

        //step4: 在二级页表上找到对应的三级页表的地址，对应的Entry没有内容的话就new一个三级页表存到Entry

        //检查是否在一个空Entry上
        if(isPageNotValid(nowSecondEntryAddress)){
            //new一个三级页表来填充这块区域
            thirdEntryAddress = (uint64*)kalloc();
            *nowSecondEntryAddress = (getPPNFromPA((uint64)thirdEntryAddress - PA2VA_OFFSET) << 10) + 0x1;
        }
        else{
            thirdEntryAddress = ((getPPNFromEntry(*nowSecondEntryAddress)) << 12) + PA2VA_OFFSET;
        }

        //step5: 获取三级页表的存真实物理地址的Entry的地址
        uint64* nowPhysicalEntryAddress = thirdEntryAddress + thirdLevelPageTableEntryIndex;

        //step6: 写入三级页表(这个Entry一定是一个空内容)
        uint64 nowPhysicalAddress = i * PGSIZE + physicalAddress;
        *nowPhysicalEntryAddress = (getPPNFromPA(nowPhysicalAddress) << 10) + perm;

        //step7: 增加并更新各个页表的index
        thirdLevelPageTableEntryIndex++;

        //如果装满了二级页表和三级页表，则需要更换根页表的Entry(即+1)
        if(thirdLevelPageTableEntryIndex == 512){
            thirdLevelPageTableEntryIndex = 0;
            secondLevelPageTableEntryIndex++;
            printk("Second Level Page +1\n");
        }
        if(secondLevelPageTableEntryIndex == 512){
```

```
            secondLevelPageTableEntryIndex = 0;
            pgtblIndex++;
            printk("PGT Level Page +1\n");
        }
        // if(pgtbl == 512){/*error*/}

    }
    printk("create_mapping finish!\n");
}
```

在讲解了最重要的`create_mapping`函数后，我们只需要在`setup_vm_final`中根据不同虚拟地址段的要求完成对应的映射即可。

首先我们定义变量`swapper_pg_dir`作为kernel pagetable 根页表

```
unsigned long  swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
```

再通过`vmlinux.lds.s`中的`_stext`，`_srodata`，`_sdata_`来计算各个段的大小。

```
uint64 virtualAddress = VM_START + OPENSBI_SIZE;
uint64 physicalAddress = PHY_START + OPENSBI_SIZE;
uint64 kernalTextSize = (uint64)_srodata - (uint64)_stext;
uint64 kernalRodataSize = (uint64)_sdata - (uint64)_srodata;
uint64 otherMemorySize = (uint64)(PHY_SIZE) - ((uint64)_sdata - (uint64)_stext);
```

对于kernel text段, 其protection bits中的V, R , X位应当被置为1， 对应十进制数字为11，所以我们进行如下映射：

```
// mapping kernel text X|-|R|V
create_mapping(swapper_pg_dir, virtualAddress, physicalAddress, kernalTextSize,
11);
```

接下来我们要为kernel rodata段进行映射，其V, R位应当被置为1， 对应十进制数字为3，由于kernel rodata段在kernel text段之后，此时的虚拟地址与物理地址都需要增加一个偏移量：

```
// mapping kernel rodata -|-|R|V
virtualAddress += kernalTextSize;
physicalAddress += kernalTextSize;
create_mapping(swapper_pg_dir, virtualAddress, physicalAddress,
kernalRodataSize, 3);
```

再接下来我们要为 other memory段进行映射，其W, V, R位应当被置为1， 对应十进制数字为7，同样虚拟地址与物理地址需要加偏移量：

```
// mapping other memory -|W|R|V
virtualAddress += kernalRodataSize;
physicalAddress += kernalRodataSize;
create_mapping(swapper_pg_dir, virtualAddress, physicalAddress,otherMemorySize,
7);
```

在完成地址映射后，我们需要将现在的根页表`swapper_pg_dir`的未relocate的地址（意味着需要减去`PA2VA_OFFSET`）写入`satp`寄存器。

```
uint64 phyPGDir = (uint64)swapper_pg_dir - PA2VA_OFFSET;
asm volatile (
        "mv t1, %[phyPGDir]\n"
        "srli t1, t1, 12\n"
        "li t0, 8\n"
        "slli t0, t0, 60\n"
        "add t0, t0, t1\n"
        "csrw satp, t0"
        :
        :[phyPGDir]"r"(phyPGDir)
        :"memory"
);
```

最后刷新TLB和icache，完成映射：

```
printk("satp set done!\n");

// flush TLB
asm volatile("sfence.vma zero, zero");
// flush icache
asm volatile("fence.i");
```

最后在`head.S`调用`setup_vm_final`：

```
call setup_vm
call relocate
call mm_init
call setup_vm_final
```

# 3. 思考题

## 3.1 验证 `.text`, `.rodata` 段的属性是否成功设置

make run后查看System.map, 发现`.text`, `.rodata`的地址已经是虚拟地址，设置成功。

System.map

Ubuntu > home > suonan > lab4 > System.map

```
ffffffe000200000 A BASE_ADDR
ffffffe000203000 D TIMECLOCK
ffffffe000203008 d _GLOBAL_OFFSET_TABLE_
ffffffe000200194 T __dummy
ffffffe000200120 T __switch_to
ffffffe000205008 B _ebss
ffffffe000203008 D _edata
ffffffe000208fa0 B _ekernel
ffffffe00020224c R _erodata
ffffffe000201d14 T _etext
ffffffe000204000 B _sbss
ffffffe000203000 D _sdata
ffffffe000200000 T _skernel
ffffffe000202000 R _srodata
ffffffe000200000 T _start
ffffffe000200000 T _stext
ffffffe0002001a4 T _traps
ffffffe000204000 B boot_stack
ffffffe000205000 B boot_stack_top
```

我们可以通过直接打印页表项地址并去内存中寻找该地址来判断是否设置成功，于是我们在`setup_vm_final`中加入打印语句打印最后一级PTE：

```
//step6: 写入三级页表(这个Entry一定是一个空内容)
uint64 nowPhysicalAddress = i * PGSIZE + physicalAddress;
if(flag == 0){
    printk("test: %lx\n",nowPhysicalEntryAddress);
    printk("%lx\n",(getPPNFromPA(nowPhysicalAddress) << 10) + perm);
flag++;
}

*nowPhysicalEntryAddress = (getPPNFromPA(nowPhysicalAddress) << 10) + perm;
```

并用gdb查看确实成功修改了：

```
Boot HART MHPM Count       : 0
Boot HART MIDELEG          : 0x0000000000000222
Boot HART MEDELEG          : 0x000000000000b109
setup_vm done!
...mm_init done!
setup_vm_final begin!
pgdir memset done!
mapping begin!
create_mapping begin! page_num:2
test: ffffffe007ffe000
000000002008000b
create_mapping finish!
create_mapping begin! page_num:1
test: ffffffe007ffe010
0000000020080803
create_mapping finish!
create_mapping begin! page_num:32765
test: ffffffe007ffe018
0000000020080c07
create_mapping finish!
mapping done!
satp set done!
```

```
remote Thread 1.1 In: task_init
Continuing.

Breakpoint 3, task_init () at proc.c:18
(gdb) x/4x ffffffe007ffe000
No symbol "ffffffe007ffe000" in current context.
(gdb) x/4x 0xffffffe007ffe000
0xffffffe007ffe000:     0x2008004b      0x00000000      0x2008044b      0x00000000
(gdb) x/4x 0xffffffe007ffe010
0xffffffe007ffe010:     0x20080803      0x00000000      0x20080c07      0x00000000
(gdb) x/4x 0xffffffe007ffe018
0xffffffe007ffe018:     0x20080c07      0x00000000      0x200810c7      0x00000000
(gdb)
```

## 3.2 为什么我们在 `setup_vm` 中需要做等值映射?

因为在程序刚开始运行时，程序运行在物理地址上，如果不做等值映射，在relocate后，pc值未改变，但其余代码都已经被映射到了虚拟地址上。此时pc就无法正确找到下一条指令的位置。

## 3.3 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

通过阅读源码，我了解到了linux的确没有进行等值映射，而是通过设置 `satp` 后通过设置 `stvec` 指向的地址，使得触发缺页异常机制时能够让地址顺利切换到虚拟地址。

我们同样可以借鉴这样的思路，在 `relocate` 函数中将将 `stvec` 寄存器指向我们新创建的标签地址：

```
relocate:
    #如果不进行等值映射
    la t0, __virtual_switch
    csrw stvec, t0
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    li t0, 0xffffffe000000000
    li t1, 0x0000000080000000
    sub t0, t0, t1
    add ra, ra, t0
    # addi ra, ra, 4
    add sp, sp, t0
    ...
```

然后在entry.S的`_traps`标签前创建`__virtual_switch`：

```
__virtual_switch:
        la t0, mm_init
        csrw sepc, t0
_traps:
        ...
```

在上面的代码中，我将`sepc`的值设置为了mm_init函数的地址（此时已经是虚拟地址），这样在下方的`_traps`中的`sret`时，pc中的地址就会变为虚拟地址，也就完成了从物理地址到虚拟地址的地址转换。

那么我们在什么时候会遇到第一次中断呢?

```
# set satp
    add t1,t1,t2
    csrw satp,t1
# flush tlb
    sfence.vma zero, zero
# flush icache
    fence.i
```

实际上我们在设置`satp`后刷新TLB和cache后，pc仍然指向物理地址，而我们的指令由于没做等值映射都在虚拟地址了，此时就会因为缺页异常而跳转到我们的`stvec`中的位置完成我们上述的地址转换了。