操作系统lab3：RV64 虚拟内存管理

教师：李环、柳晴
学号：3220104519
姓名：蔡佳伟

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、实验过程

2.1 准备工程

修改了`defs.h`,添加了以下内容

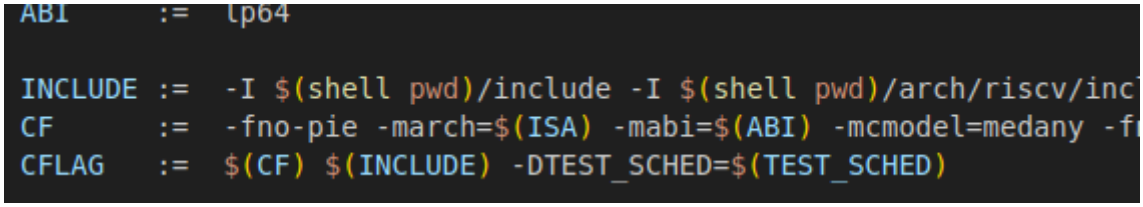
```
#define OPENSBI_SIZE (0x200000)

#define VM_START (0xffffffe000000000)
#define VM_END   (0xffffffff00000000)
#define VM_SIZE (VM_END - VM_START)

#define PA2VA_OFFSET (VM_START - PHY_START)
```

并且新建了之后要用到的`vm.h`和`vm.c`两个文件分别在`include`和`kernel`目录下。文件结构如下：

```
cjw@cjw-virtual-machine:~/OSlab3/os24fall-stu/src/lab3$ tree
.
├── arch
│   └── riscv
│       ├── include
│       │   ├── clock.h
│       │   ├── defs.h
│       │   ├── mm.h
│       │   ├── proc.h
│       │   ├── sbi.h
│       │   └── vm.h
│       ├── kernel
│       │   ├── clock.c
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── mm.c
│       │   ├── proc.c
│       │   ├── sbi.c
│       │   ├── trap.c
│       │   ├── vm.c
```

```
|       |   └── vmlinux.lds
|       └── Makefile
├── include
|   ├── printk.h
|   ├── stddef.h
|   ├── stdint.h
|   ├── stdlib.h
|   └── string.h
├── init
|   ├── main.c
|   ├── Makefile
|   └── test.c
├── lib
|   ├── Makefile
|   ├── printk.c
|   ├── rand.c
|   └── string.c
└── Makefile

8 directories, 30 files
```

同步了`vmlinux.lds`代码，并且修改了最外层的Makefile，在CF这一行加了`-fno-pie`,如图所示。



2.2 开启虚拟内存映射

2.2.0 理解变量设置

`early_pgtbl[512]`

- 表示定义了一个包含 512 个`uint64_t`类型元素的数组。
- 由于每个 `uint64_t` 是 8 字节，所以整个数组 `early_pgtbl` 的总大小为：
  512×8=4096 字节512×8=4096 字节 即 4KB。

`__attribute__((__aligned__(0x1000)))`

- 这部分是一个 **GCC扩展属性**，它用于指定变量的内存对齐要求。
- `__aligned__(0x1000)` 让编译器将 `early_pgtbl` 的地址按照 4KB（即 `0x1000` 字节）对齐。4KB 是我们本次实验的内存页大小。这样做可以确保 `early_pgtbl` 在内存中与页表的结构一致，并且能够与系统的虚拟内存管理机制（比如分页）正常配合。

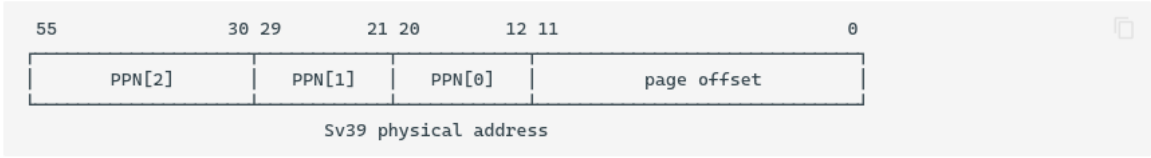
2.2.1 `setup_vm`实现

首先实现等值映射，其次进行直接映射，代码如下：

```c
#define PTE_R (1UL << 1) // read
#define PTE_W (1UL << 2) // write
#define PTE_X (1UL << 3) // execute

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
     * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    **/
    printk("start setup_vm!\n");
    unsigned long PA = PHY_START; // 初始化物理地址
    unsigned long VA_EQ = PA; // 等值映射

    int index = (VA_EQ >> 30) & 0x1ff; // 得到页表的索引
    unsigned long PPN = (PA >> 30) & 0x3ffffff; // PPN = PPN[2] 55-30
    unsigned long PTE = (PPN << 28) | 0xf; // PTE = PPN[2] and V R W X = 1
    early_pgtbl[index] = PTE;

    unsigned long VA_DIRECT = PA + PA2VA_OFFSET;
    index = (VA_DIRECT >> 30) & 0x1ff;
    PPN = (PA >> 30) & 0x3ffffff;
    PTE = (PPN << 28) | 0xf;
    early_pgtbl[index] = PTE;

    printk("setup_vm done!\n");
}
```

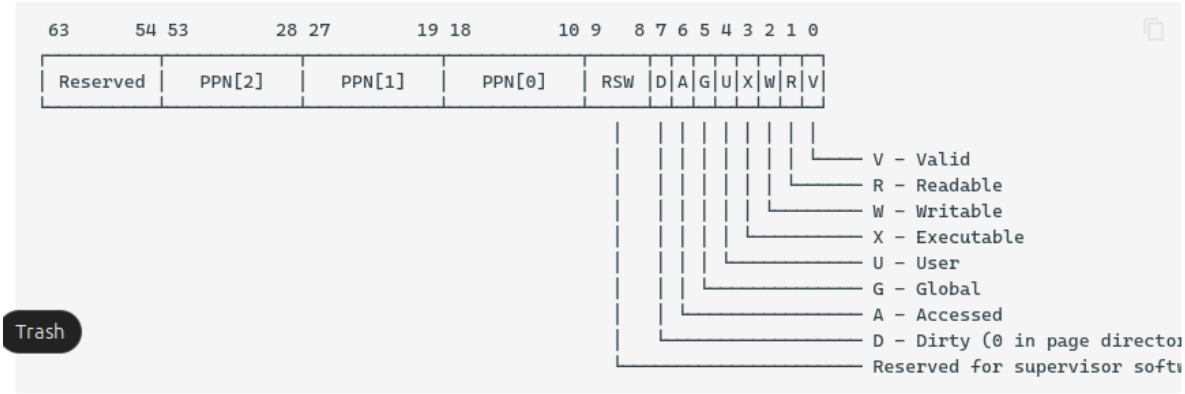
因为Sv39的物理地址是56位，我们先得到了va的低30位作为页内偏移，因为只使用根页表，取出最低9bit，得到根页表的索引。然后得到PPN的值，即为PPN[2]。

3.3.2 Sv39 虚拟地址和物理地址



由于PTE格式如下，将PPN左移28位，再将需要的权限位设置为1。

3.3.3 Sv39 页表项



Trash

然后修改 `head.S` 文件，首先调用 `setup_vm` 函数，然后调用 `relocate` 函数，完成对 `satp` 的设置，以及跳转到对应的虚拟地址。

```
1   .extern start_kernel
2   .extern mm_init
3   .extern task_init
4   .extern setup_vm
5   .extern setup_vm_final
6       .section .text.init
7       .globl _start
8
9   _start:
10      # ------------------
11      # - your code here -
12
13      # set sp move to top of boot_stack
14      la sp, boot_stack_top
15
16      call setup_vm
17      call relocate
18
19      call mm_init
```

```
relocate:

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

    # flush tlb
    sfence.vma zero, zero

    # set satp with early_pgtbl
    la t1, early_pgtbl
    srl t1, t1, 12
    li t2, 0x8000000000000000
    or t1, t1, t2
    li t3, 0xf0000fffffffffff
    and t1, t1, t3
    csrw satp, t1

    ret
```

可以看到，首先我们调整栈指针位置，刷新TLB，保证刷新虚拟地址到物理地址的映射缓存。然后加载`early_pgtbl`的地址到内存中，右移12位得到顶级页表的物理页号。然后将mode字段修改为8，我们所需要的Sv39的mode。

satp（Supervisor Address Translation and Protection Register）是RISC-V中控制虚拟内存分页模式的寄存器，其结构如下：

```
 63      60 59                  44 43                                  0
┌──────────┬──────────────────────┬──────────────────────────────────────┐
│   MODE   │         ASID         │                PPN                   │
└──────────┴──────────────────────┴──────────────────────────────────────┘
```

- MODE字段的取值如下图：

```
                      RV 64
┌──────────┬──────────┬──────────────────────────────────────┐
│  Value   │   Name   │  Description                         │
├──────────┼──────────┼──────────────────────────────────────┤
│    0     │ Bare     │ No translation or protection         │
│  1 - 7   │ ---      │ Reserved for standard use            │
│    8     │ Sv39     │ Page-based 39 bit virtual addressing │ <-- 我们使用的 mode
│    9     │ Sv48     │ Page-based 48 bit virtual addressing │
│    10    │ Sv57     │ Page-based 57 bit virtual addressing │
│    11    │ Sv64     │ Page-based 64 bit virtual addressing │
│ 12 - 13  │ ---      │ Reserved for standard use            │
│ 14 - 15  │ ---      │ Reserved for standard use            │
└──────────┴──────────┴──────────────────────────────────────┘
```

然后保留`t1`寄存器的mode位和后44位PPN，（按照要求把ASID位置0）。最后写入`satp`完成初始化过程。

此外，还要调整`mm_init`的`kfreerange`的结束地址，把`PHY_END`调整为`(VM_START+PHY_SIZE)`，即虚拟地址。

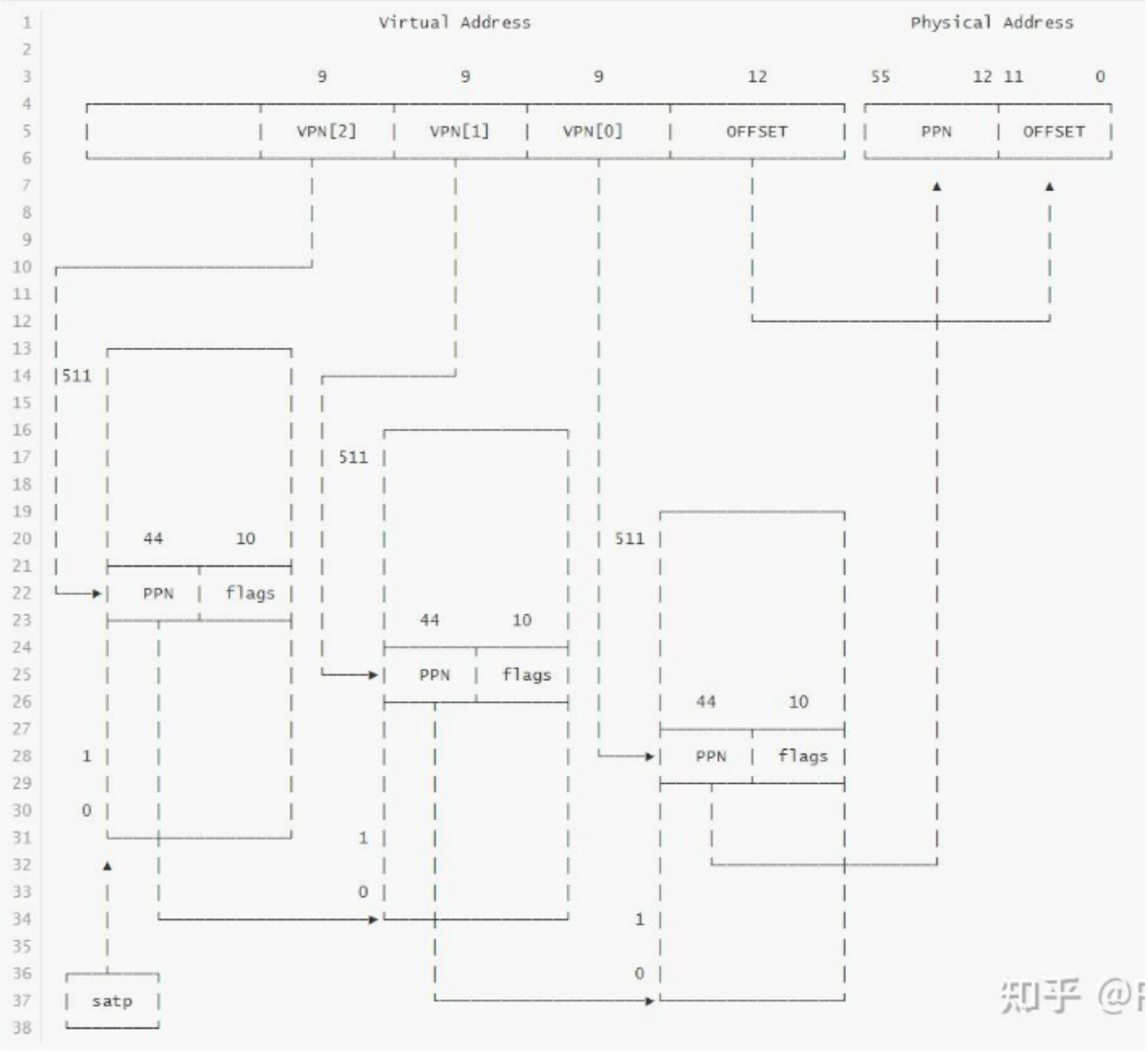
```
void mm_init(void) {
    kfreerange(_ekernel, (char *)(VM_START+PHY_SIZE));
    printk("...mm_init done!\n");
}
```

2.2.2 `setup_vm_final`的实现

`head.S`的调用顺序如下，因为`setup_vm_final`中需要申请页面来建立多级页表,先要调用mm_init```来完成内存管理初始化。

```
_start:
    # ------------------
    # - your code here -

    # set sp move to top of boot_stack
    la sp, boot_stack_top

    call setup_vm
    call relocate

    call mm_init

    call setup_vm_final
    call task_init
```

在`create_mapping`中设置三级页表映射关系，先计算出映射的页数（向上取整），然后每一页会被分配一个虚拟地址，计算这个地址三个部分的虚拟页号（分别是右移12,21,30位）。



对于每一层的页表（从最高层开始），先检查`pte[VPN[j]]`来判断页表项是否已分配。如果该页表项的有效位为0，即未分配，就分配一个新的页表项，并更新该项为一个指向下一级页表的地址。右移12位得到物理页号，保留48位有效地址，左移10为用于标记权限，适配Sv39页表项的格式。如果该页表项已经有效（即页表已经分配），则将`pte`指向该项对应的页表地址，继续向下一层页表递归查找。最后将`va`和`pa`都增加一个页的大小（`PGSIZE`），以便映射下一个页面。

```
unsigned long swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
extern char _stext[],_etext[];
extern char _srodata[],_erodata[];
extern char _sdata[],_edata[];
void create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, uint64_t perm){
    int page_num = (sz + PGSIZE - 1) / PGSIZE;
    for(int i = 0; i < page_num; i++){
        uint64_t VPN[3];
        VPN[0] = (va >> 12) & 0x1ff;
        VPN[1] = (va >> 21) & 0x1ff;
        VPN[2] = (va >> 30) & 0x1ff;
        uint64_t *pte = pgtbl;
        for(int j = 2; j > 0; j--){
            if((pte[VPN[j]] & 0x1) == 0){
                uint64_t new_pte = kalloc();
                pte[VPN[j]] = (((new_pte - PA2VA_OFFSET) >> 12) & 0xfffffffffff) << 10 | 0x1;
                pte = (uint64_t *)new_pte;
            }
            else{
                pte = (uint64_t *)((pte[VPN[j]] >> 10 << 12) + PA2VA_OFFSET);
            }
        }
        pte[VPN[0]] = (((pa >> 12) & 0xfffffffffff) << 10) | perm;
        va += PGSIZE;
        pa += PGSIZE;
    }
}
```

这里按照文字提示完成即可，后面的几位按照提示来设置，然后设置`satp`寄存器，`0x8000000000000000`是设置`satp`寄存器的标志，表示启用虚拟内存，并且以`Sv39`模式工作（39位虚拟地址空间）。`((unsigned long)swapper_pg_dir - PA2VA_OFFSET) >> 12`将根页表的物理地址右移 12 位，得到页表基地址（RISC-V 页表的格式要求基地址的页对齐）。减去`PA2VA_OFFSET`是为了确保物理地址与虚拟地址偏移正确。`csr_write(satp, satp)`将计算得到的`satp`值写入到`satp`寄存器中，从而启用虚拟内存。

最后刷新TLB，然后就完成了虚拟内存的最终配置。分别映射了代码段、只读数据段和数据段。

```
void setup_vm_final(void){
    memset(swapper_pg_dir, 0x0, PGSIZE);

    create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_stext, (uint64_t)_stext-PA2VA_OFFSET, (uint64_t)(_etext - _stext), 0xb);
    create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_srodata, (uint64_t)_srodata-PA2VA_OFFSET, (uint64_t)(_erodata - _srodata), 0x3);
    create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_sdata, (uint64_t)_sdata-PA2VA_OFFSET, PHY_END + PA2VA_OFFSET - (uint64_t)_sdata, 0x7);
    unsigned long satp = 0x8000000000000000 | ((unsigned long)swapper_pg_dir - PA2VA_OFFSET) >> 12;
    csr_write(satp, satp);

    asm volatile("sfence.vma zero, zero");

    return;
}
```

2.2.3 测试结果

运行`make run`得到如下结果，说明正确实现上述功能。

```
start setup_vm!
setup_vm done!
...mm_init done!
...task_init done!
2024 ZJU Operating System
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
first schedule
Schedule
i = 0 task[i]->counter = 0
i = 1 task[i]->counter = 0
i = 2 task[i]->counter = 0
i = 3 task[i]->counter = 0
i = 4 task[i]->counter = 0
i = 5 task[i]->counter = 0
i = 6 task[i]->counter = 0
i = 7 task[i]->counter = 0
i = 8 task[i]->counter = 0
i = 9 task[i]->counter = 0
```

三、讨论和心得

本次实验我遇到的困难主要在于由于理论内容掌握得不扎实，对为什么进行某些操作不太理解，所以在写代码的过程中也遇到了一些困难。但我经过在网上查询一些资料，仔细阅读实验指导中参考资料的重要部分，对虚拟内存的相关知识更加清晰了。因为实验指导中的文字提示，告诉我们代码要大致怎么实现还是比较清晰的，主要问题是在不知道为什么要这样操作上。

四、思考题

1. 验证 .text，.rodata 段的属性是否成功设置，给出截图。

在成功设置后，`.text`，`.rodata`段是不允许写入的，所以在设置之后对其进行写入，如果不能写入则证明设置成功。

在`setup_vm_final`函数的末尾添加

```
printk(GREEN"setup_vm_final done!\n"CLEAR);
printk("_stext = %x\n", *_stext);
*_stext = 0x31;
printk("_stext = %x\n", *_stext);
printk("_srodata = %x\n", *_srodata);
*_srodata = 0x31;
printk("_srodata = %x\n", *_srodata);
printk(RED"done!\n"CLEAR);
```

之后再return，运行后卡在这里

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : time,sstc
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
start setup_vm!
setup_vm done!
...mm_init done!
setup_vm_final done!
_stext = 17
```

说明无法成功写入`_stext`

注释掉修改`_stext`的语句后，卡在这里

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : time,sstc
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
start setup_vm!
setup_vm done!
...mm_init done!
setup_vm_final done!
_stext = 17
_stext = 17
_srodata = 2e
```

说明均设置成功，`_srodata`也不允许写入。

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；

在`relocate`执行过程中，虽然会设置`satp`寄存器的值，但是在设置完之后，`pc`仍然是物理地址，所以需要在原本的位置做一次等值映射，保证这个函数仍然可以运行这个位置的代码，使之顺利运行完。由于我们设置过`ra`，所以这个函数运行完之后会返回到虚拟地址继续运行。

如果不做等值映射，`relocate`无法正常运行。

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；

设置`satp`切换页表附近的代码如下：

```
 relocate:
 /* Relocate return address */
 li a1, PAGE_OFFSET
 la a0, _start
 sub a1, a1, a0
 add ra, ra, a1

 /* Point stvec to virtual address of intruction after satp write
*/

 la a0, 1f
 add a0, a0, a1
 csrw CSR_STVEC, a0

 /* Compute satp for kernel page tables, but don't load it yet */
 la a2, swapper_pg_dir
 srl a2, a2, PAGE_SHIFT
 li a1, SATP_MODE
 or a2, a2, a1

 /*
 * Load trampoline page directory, which will cause us to trap
* stvec if VA != PA, or simply fall through if VA == PA.  We
need a
 * full fence here because setup_vm() just wrote these PTEs and
we need
 * to ensure the new translations are in use.
 */

 la a0, trampoline_pg_dir
 srl a0, a0, PAGE_SHIFT
 or a0, a0, a1
 sfence.vma
 csrw CSR_SATP, a0
.align 2
1:
 /* Set trap vector to spin forever to help debug */
 la a0, .Lsecondary_park
 csrw CSR_STVEC, a0
```

```
 /* Reload the global pointer */
 .option push
 .option norelax
 la gp, __global_pointer$
 .option pop
 /*
 * Switch to kernel page tables.  A full fence is necessary in
order to
 * avoid using the trampoline translations, which are only
correct for
 * the first superpage.  Fetching the fence is guarnteed to work
 * because that first superpage is translated the same way.
 */
 csrw CSR_SATP, a2
 sfence.vma

 ret
```

1. 在`relocate`函数中，首先获取物理地址到虚拟地址的地址偏移量，然后获取`_start`的地址，也就是程序的入口。接下来计算`PAGE_OFFSET`与`_start`之间的差值，也就是从物理地址到虚拟地址的偏移量，并将这个偏移量添加到`ra`寄存器中，使得函数在返回的时候，可以直接返回到虚拟地址的对应位置。
2. 然后对`stvec`寄存器进行设置，使得里面的地址是正确的陷阱处理程序的地址（冒号1这一行在虚拟地址中的地址），这样可以使得在出现中断时可以跳转到第31行在虚拟地址中的位置。
3. 之后，设置`satp`寄存器。首先获取`swapper_pg_dir`的地址，然后右移`PAGE_SHIFT`位，将页表基地址转换为页表索引。然后将`SATP_MODE`加载到寄存器中，并且进行按位或操作，以设置`satp`寄存器的模式位和页表索引。
4. 将`trampoline_pg_dir`符号的地址加载到寄存器中，并且右移`PAGE_SHIFT`位，以将`trampoline`页表基地址转换为页表索引。接着进行寄存器之间的按位或操作，以设置`satp`寄存器的模式位和页表索引。然后执行`sfence.vm`，最后将寄存器`a0`中的值写入`satp`寄存器。
5. 重新对`stvec`寄存器进行设置，这是在第一次中断之后的跳转的地址，此时后续的中断不在需要跳转到这个地址，这个地址只是为了过渡到虚拟地址空 间，所以此时需要重新设计中断的内容为后续正确的中断内容。
6. 重新加载全局指针操作，首先保存当前的汇编选项并禁用全局指针的自动优化，然后将`_global_pointer$`符号的地址加载到全局指针寄存器`gp`中，最后恢复之前的汇编选项。
7. 最后切换到内核页表，将`a2`的值写入`satp`控制状态寄存器并且执行`sfence.vma`，最后函数返回。

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；

从上一问中可以知道，内核在运行到`csrw CSR_SATP, a0`时会因为地址的异常而陷入中断，但是因 为此时之前已经设置过`stvec`寄存器中的内容，使得此时的中断地址为冒号1那一行虚拟地址中的位置，也就是说，通过第一次中断的`pc`跳转，使得`pc`从物理地址跳转到了虚拟地址。

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；

`trampoline_pg_dir` 是用于在内核从物理地址到虚拟地址的过渡期间提供临时映射的页表，确保分页机制启用时系统的稳定性。 `swapper_pg_dir` 是内核在启动完成后使用的主要页表，提供完整的虚拟地址空间映射，保证内核的正常操作。 在 `relocate` 函数中， `trampoline_pg_dir` 被设置为 `satp` 的内容，以启用分页机制。通过这一临时页表，内核可以继续执行并进入后续的页表切换流程。 在 `relocate` 函数的最后阶段， `swapper_pg_dir` 被加载到 `satp` 寄存器中。这是最终的页表切换，使得内核从过渡页表 `trampoline_pg_dir` 转换到 `swapper_pg_dir` ，实现完整的虚拟内存地址空间。

尝试修改你的 kernel，使得其可以像 Linux 一样不需要等值映射。

对 `setup_vm` 修改，删除等值映射

```
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
     * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    **/
    printk("start setup_vm!\n");
    unsigned long PA = PHY_START; // 初始化物理地址
    unsigned long VA_EQ = PA; // 等值映射

    int index = (VA_EQ >> 30) & 0x1ff; // 得到页表的索引
    unsigned long PPN = (PA >> 30) & 0x3ffffff; // PPN = PPN[2] 55-30
    unsigned long PTE = (PPN << 28) | 0xf; // PTE = PPN[2] and V R W X =
    // early_pgtbl[index] = PTE;

    unsigned long VA_DIRECT = PA + PA2VA_OFFSET;
    index = (VA_DIRECT >> 30) & 0x1ff;
    PPN = (PA >> 30) & 0x3ffffff;
    PTE = (PPN << 28) | 0xf;
    early_pgtbl[index] = PTE;

    printk("setup_vm done!\n");
}
```

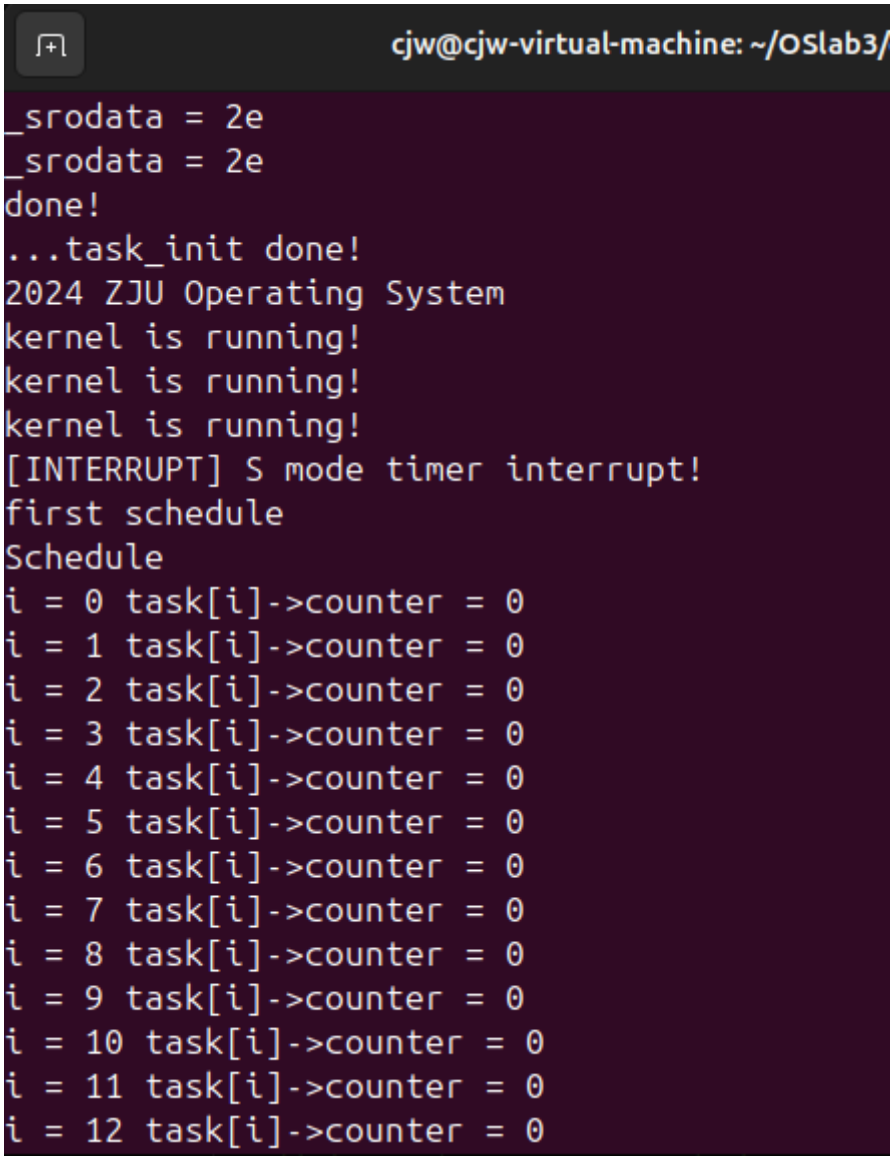
此时运行会卡住

```
()
Domain0 Region03          : 0x0000000000000000-0xff
: (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087e00000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes
Domain0 SysSuspend        : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : time,sstc
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
start setup_vm!
setup_vm done!
```

修改`head.S`中`relocate`函数

```
relocate:

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

    # point stvec to virtual address of instruction after satp write
    la a0, temp
    add a0, a0, t0
    csrw stvec, a0

    # flush tlb
    sfence.vma zero, zero

    # set satp with early_pgtbl
    la t1, early_pgtbl
    srl t1, t1, 12
    li t2, 0x8000000000000000
    or t1, t1, t2
    li t3, 0xf0000fffffffffff
    and t1, t1, t3
    csrw satp, t1

    ret

temp:
    la t0, _traps
    csrw stvec, t0
    sfence.vma zero, zero
    ret
```

此时运行，发现运行成功了，可见修改正确。


cjw@cjw-virtual-machine: ~/OSlab3/
_srodata = 2e
_srodata = 2e
done!
...task_init done!
2024 ZJU Operating System
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
first schedule
Schedule
i = 0 task[i]->counter = 0
i = 1 task[i]->counter = 0
i = 2 task[i]->counter = 0
i = 3 task[i]->counter = 0
i = 4 task[i]->counter = 0
i = 5 task[i]->counter = 0
i = 6 task[i]->counter = 0
i = 7 task[i]->counter = 0
i = 8 task[i]->counter = 0
i = 9 task[i]->counter = 0
i = 10 task[i]->counter = 0
i = 11 task[i]->counter = 0
i = 12 task[i]->counter = 0