Lab 4: RISC-V 虚拟内存管理

1. 实验目的

结合课堂学习的页式内存管理以及虚拟内存的相关知识，尝试在已有的程序上开启 MMU 并实现页映射，保证之前的进程调度能在虚拟内存下正常运行。

2. 实验内容及要求

- 实现Sv39分配方案下的三级页表映射
- 了解页表映射的权限机制，设置不同的页表映射权限，完成对section的保护及相应的异常处理。

本次实验为单人完成。

3. 实验步骤

3.1 实验背景

3.1.1 建立映射

3.1.2 组织文件结构

```
lab4
├── arch
│   └── riscv
│       ├── boot
│       ├── include
│       │   ├── put.h
│       │   ├── sched.h
│       │   └── vm.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   ├── vm.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── put.h
│   └── rand.h
├── init
│   ├── main.c
│   └── Makefile
├── lib
│   ├── Makefile
│   ├── put.c
```

```
│   └── rand.c
└── Makefile
```

本次实验中将设置`satp`寄存器开启Sv39内存分配方案，随后设置物理地址与虚拟地址的映射，最后为不同section设置不同的映射权限，并实现相应的异常处理。为了实现映射，需要分配内存空间给页表，并对页表项进行相应的设置。由于开启了MMU，还需要对进程调度的相关代码进行一定的修改。

3.1.3 Sv39 分页方案

a. 虚拟内存

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存，而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

通过内存地址虚拟化，可以使得只有软件在实际访问某虚拟内存地址时，操作系统才动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为**按需分页**（demand paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为**页换入换出**（page swap in/out）。

b. MMU

在使用虚拟内存的情况下，虚拟地址不是被直接送到内存总线上，而是被送到**内存管理单元**（Memory Management Unit，MMU），MMU把虚拟地址映射为物理内存地址。

c. RISC-V Sv39 分页

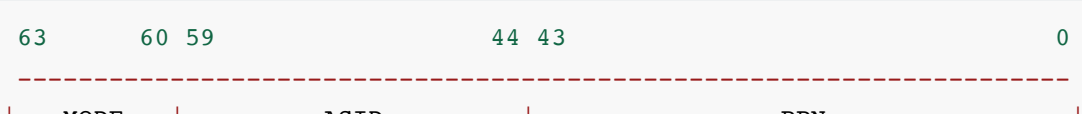
S模式的虚拟内存系统将内存划分为固定大小的页来进行地址转换和对内存内容的保护。启用分页的时候，大多数地址（包括 load和 store的有效地址和PC中的地址）都是虚拟地址 。要访问物理内存，它们必须被转换为真正的物理地址 ，这通过遍历一种称为页表的高基数树实现。

本次实验使用了Sv39分页方案。 Sv39使用4KB大小的基页，页表项的大小是8个字节，为了保证页表大小和页面大小一致，树的基数相应地降到2^9，树也变为三层。 Sv39的 512 GB地址空间（虚拟地址）划分为2^9个 1GB大小的吉页 。每个吉页被进一步划分为2^9个2MB大小的巨页。每个巨页再进一步分为2^9个4KB大小的基页。

在未分页时访问的地址都是物理地址。分页后，监管者模式和用户模式下访问的地址是虚拟地址。

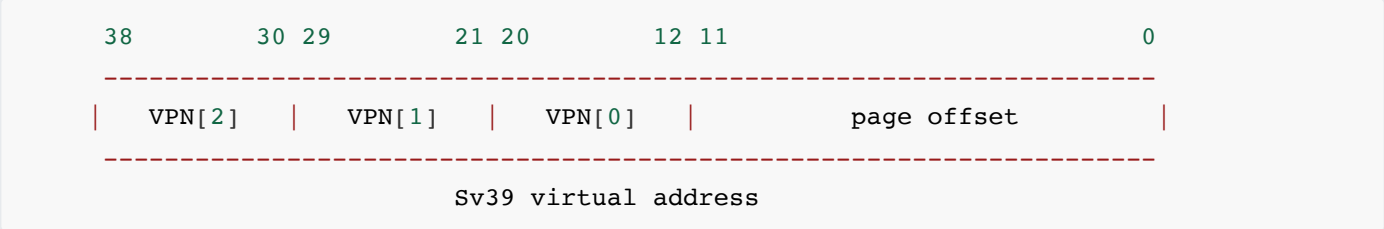
d. satp (Supervisor Address Translation and Protection Register)

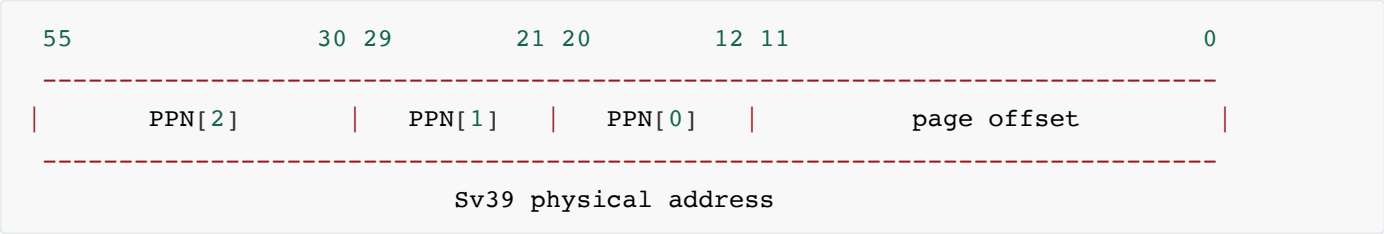
satp（Supervisor Address Translation and Protection，监管者地址转换和保护）为S模式的控制状态寄存器，用来控制了分页系统，



- MODE：可以开启分页并选择页表级数，8表示Sv39分配方案，0表示禁用虚拟地址映射。
- ASID (Address Space Identifier)： 用来区分不同的地址空间，**此次实验中直接置0即可**。
- PPN (Physical Page Number)： 保存了根页表的物理地址，通常`PPN = physical address >> 12`。M模式的程序在第一次进入 S模式之前会把零写入 satp以禁用分页，然后 S模式的程序在初始化页表以后会再次进行satp寄存器的写操作。

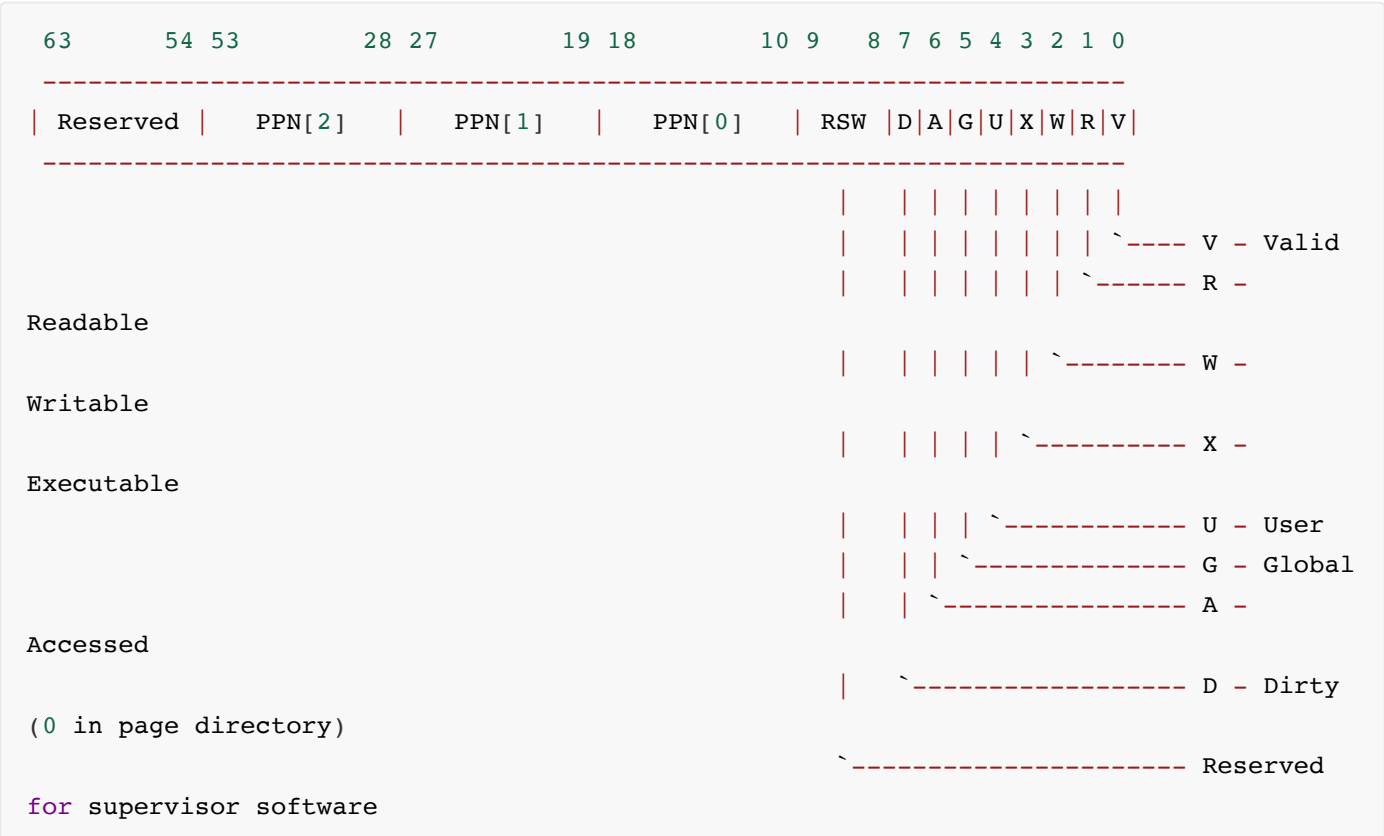
e. RISC-V Sv39 Virtual Address and Physical Address





Sv39模式定义物理地址有56位，虚拟地址有64位。但是，虚拟地址的64位只有39位有效，**63-39位在本次实验中（高地址映射）需要为1保证地址有效**。Sv39支持三级页表结构，VPN[2-0] (Virtual Page Number) 分别代表每级页表的虚拟页号，PPN[2-0] (Physical Page Number) 分别代表每级页表的物理页号。物理地址和虚拟地址的低12位表示页内偏移（page offset）。

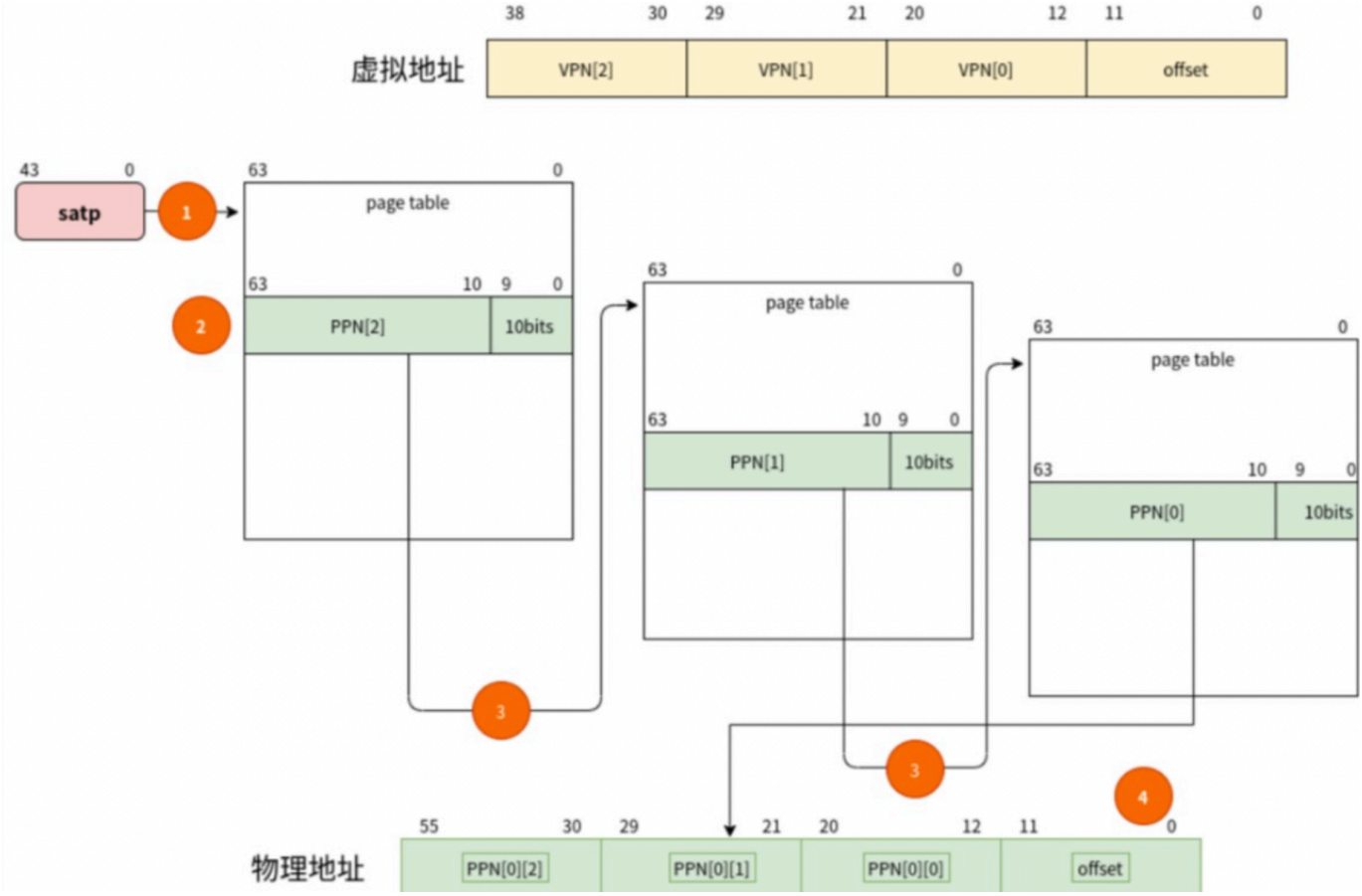
f. RISC-V Sv39 Page Table Entry



- 0 ~ 9 bit: protection bits
  - V: 有效位，当 V = 0, 访问该PTE会产生Pagefault。
  - R: R = 1 该页可读。
  - W: W = 1 该页可写。
  - X: X = 1 该页可执行。

- **U,G,A,D,RSW本次实验中设置为0即可。**

g. RISC-V Address Translation Details



虚拟地址翻译为物理地址的过程:

- 1.从satp的`PPN`中获取根页表的物理地址。
- 2.通过pagetable中的VPN段,获取PTE。(可以把pagetable看成一个数组，VPN看成下标。PAGE_SIZE为4KB，PTE为64bit(8B), 所以一页中有4KB/8B=512个PTE，而每级VPN刚好有9位，与512个PTE一一对应)。
- 3.检查PTE的`V bit`，如果不合法，应该产生page fault异常。
- 4.检查PTE的`RWX` bits,如果全部为0，则从PTE中的PPN[2-0]得到的是下一级页表的物理地址，则回到第二步。否则当前为最后一级页表，PPN[2-0]得到的是最终物理页的地址。
- 5.将得到最终的物理页地址，与偏移地址相加，得到最终的物理地址。
- 6.对齐注意

> Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB megapages and 1 GiB gigapages, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

> If $i > 0$ and $pte.ppn[i-1:0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.

3.2 修改head.S

3.2.1 修改系统启动部分代码

`vmlinux.lds`中规定将内核放在物理内存`0x80000000`、虚拟内存`0xffffffe000000000`的位置，因此物理地址空间下的地址x在虚拟地址空间下的地址为`x-0x80000000+0xffffffe000000000`。

- 在`_supervisor`开头先设置`satp`寄存器为0，暂时关闭MMU
- 设置`stvec`为异常处理函数`trap_s`在**虚拟地址空间下**的地址
- 记录`start_kernel`在**虚拟地址空间下**的地址，加载到寄存器中
- 设置`sp`的值为`stack_top`的物理地址（设置低地址空间栈环境）
- 调用`paging_init`函数进行映射
- 设置`satp`的值以打开MMU，注意修改`satp`后需要执行`sfence.vma`指令同步虚拟内存相关映射
- 设置`sp`的值为虚拟地址空间下的`init_stack_top`
- 使用`jr`指令跳转到虚拟地址下的`start_kernel`（读取之前记录在寄存器中的值），并在虚拟地址空间中执行后续语句与进程调度

```
_supervisor:
  #your code
  #将satp设为0，关闭MMU
  csrw satp, zero
  #stvec设置为trap_s在虚拟地址空间下的地址
  la t1, trap_s
  li t2,  0xffffffdf80000000  #0xffffffdf80000000 = -0x80000000 + 0xffffffe000000000
  add t1, t1, t2
  csrw stvec, t1
  #记录start_kernel在虚拟地址空间下的地址，加载到寄存器中
  la t1, start_kernel
  add t1, t1, t2
  #设置sp的值为stack_top的物理地址（设置低地址空间栈环境）
  la sp, stack_top
  #调用paging_init函数进行映射
  call paging_init
  #设置satp的值以打开MMU，修改satp后需要执行sfence.vma指令同步虚拟内存相关映射
  add t0, zero, 8 # satp[63:60] = 8
  sll t0, t0, 16 # satp[59:44] = 0
  sll t0, t0, 44
  la t3, _end
  srl t3, t3, 12 # PPN = physical address >> 12
  or t0, t0, t3
  csrw satp, t0
  sfence.vma
  #设置sp的值为虚拟地址空间下的init_stack_top
  la sp, init_stack_top
  add sp, sp, t2
  #使用jr指令跳转到虚拟地址下的start_kernel，并在虚拟地址空间中执行后续语句与进程调度
  jr t1
```

Q1：satp的值应当设置为?(映射时将从`_end`开始为页表分配物理空间)

答：

```
63      60 59                  44 43                                     0
 ------------------------------------------------------------------------
|    8    |         0            |   (physical address) &_end >> 12    |
 ------------------------------------------------------------------------
```

3.2.2 修改M模式下的异常处理代码

- 由于M模式下依然使用物理地址，使用虚拟地址将导致内存访问错误。因此，需要保留一片物理地址区域用于异常处理前保存所有寄存器的值。
- `mscratch`寄存器是M mode下专用的临时寄存器。通常，它就用于保存M mode下上下文物理空间的地址。lds文件中分配出了1个page的空间用于储存进程上下文，其顶部标记为`stack_top`，在head.S进入S mode之前的适当位置，将`mscratch`寄存器设置为`stack_top`的物理地址。
- 在M mode异常处理函数`trap_m`的开头，将`mscratch`与`sp`寄存器的值交换（hint: 使用`csrrw`指令），使用上下文空间作为`trap_m`的栈并保存`x1-x31`寄存器。
- 在`trap_m`返回前将`mscratch`与`sp`寄存器的值重新交换回来。

3.3 实现映射机制

3.3.1 创建映射

为了增加一个虚拟地址到物理地址的映射，根据Rv39分配方案中的地址翻译规则，首先需要**根据根页表基地址以及虚拟地址找到相应页表项的地址，并在该过程中为页表分配物理页面，随后根据物理地址及映射的权限设置页表项具体的值**，使得后续MMU工作时能够根据根页表基地址和虚拟地址得到正确的物理地址。

在`vm.c`中编写函数`create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm)`，用作页表映射的统一接口，其中参数作用如下：

- `pgtbl`为根页表的基地址
- `va`,`pa`分别为需要映射的虚拟、物理地址的基地址
- `sz`为映射的大小
- `perm`为映射的读写权限

对于多级页表，父节点页表项存储的值是第一个子节点的位置,而同一个节点的子节点是连续存储的，理论上只需要根节点我们就能得到所有页表项的地址。因此可以进行**按需分配**，当需要使用该虚拟地址对应的物理地址时，再分配内存给相应的页表项。若需要为页表空间分配物理页，可以自由管理分配`_end`地址之后的物理内存（提示：一个页面占4KB，如`&_end+0x1000*page_count`）。

代码

```
int page_count = 0;

uint64_t* change_pagelevel(uint64_t* pgtbl, uint64_t va){
    /*表示页表级数*/
    for(int i = 2; i >= 1; i --){
```

```
        uint64_t* pte_address;
        if(i == 2){
            /*把pagetable看成一个数组，VPN看成下标*/
            uint64_t vpn2 = ((uint64_t)va >> 30)&(0x1ff);
            pte_address = &pgtbl[vpn2];
        }
        else if(i == 1){
            uint64_t vpn1 = ((uint64_t)va >> 21)&(0x1ff);
            pte_address = &pgtbl[vpn1];
        }
        /*检查pte的v位是否为1*/
        if((_Bool)((*pte_address) & 0x1)) {
            uint64_t ppn = (uint64_t)( (*pte_address) >>10)&(0xfffffffffff);
            pgtbl = (uint64_t*)(ppn << 12);
        }
        else{
            page_count ++;
            if(PAGE_SIZE * page_count <= MAP_SIZE){
                pgtbl = (uint64_t)(&_end + PAGE_SIZE * page_count);
                uint64_t ppn = (uint64_t)((uint64_t)pgtbl >> 12)&(0xfffffffffff);
                *pte_address = ((uint64_t)(*pte_address) & 0xffc0000000000000) |
((uint64_t)(ppn << 10)) | ((uint64_t)0) | ((uint64_t)V_ENABLE);
             }
            else{
                puts("[!] Insufficient Space to be allocated.\n");
                return PAGE_FAULT;
            }
        }
    }
    uint64_t vpn0 = ((uint64_t)va >> 12)&(0x1ff);
    return &pgtbl[vpn0];
}

void create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, int perm)
{
  /*your code*/
    uint64_t alimit = va + sz - 1;
    for(; va <= alimit; va = va + PAGE_SIZE, pa = pa + PAGE_SIZE){
        uint64_t* pte_address = change_pagelevel(pgtbl, va);
        if(pte_address != PAGE_FAULT){
            uint64_t ppn = (uint64_t)(pa >> 12)&(0xfffffffffff);
            *pte_address = ((uint64_t)(*pte_address) & 0xffc0000000000000) |
((uint64_t)(ppn << 10)) | ((uint64_t)perm) | ((uint64_t)V_ENABLE);
        }
    }
}
```

Q2：解释`create_mapping`的代码设计思路。

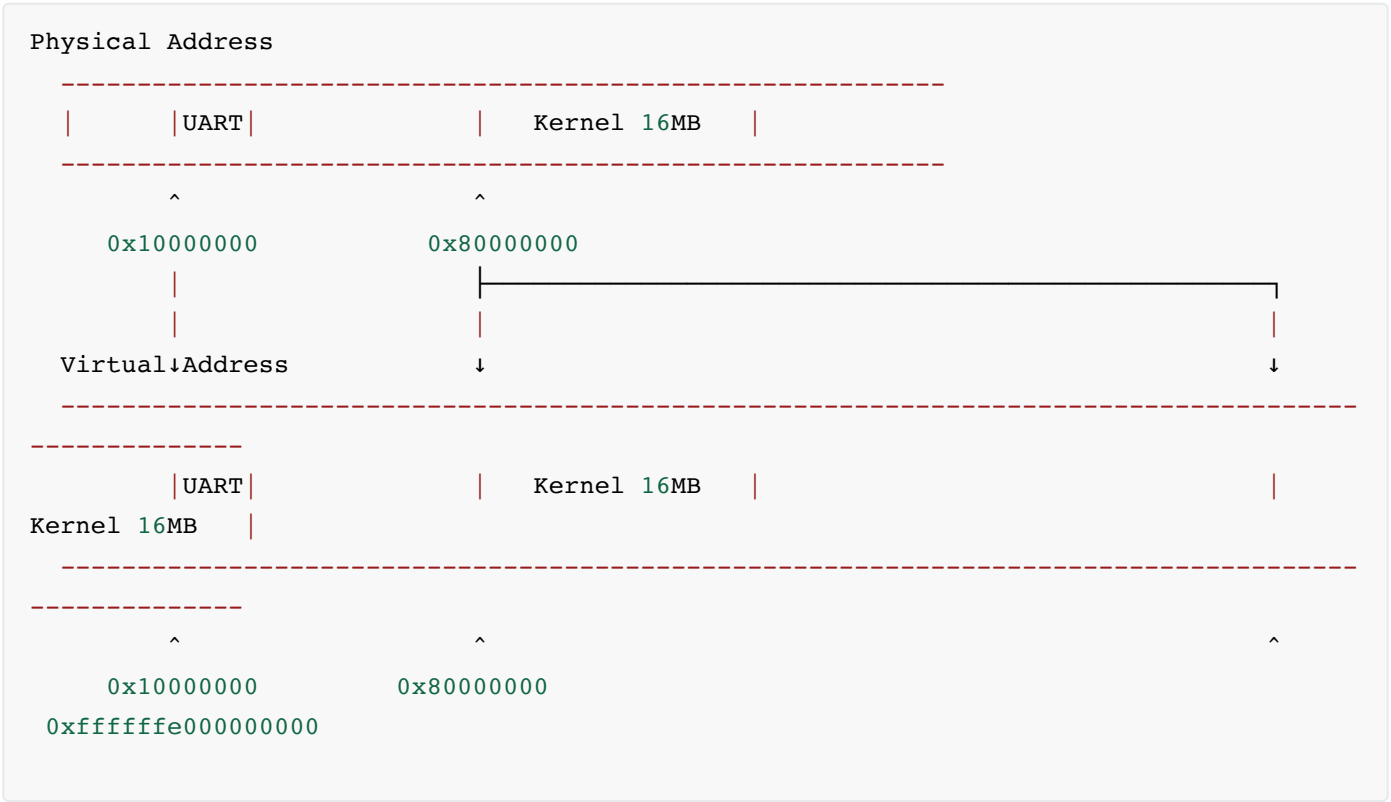
答：虚拟地址最大值为`va + sz - 1`，以这个值为界限，4kb为物理页和虚拟页的页长循环创建页表的映射，每次循环后va和pa需要加一页长。由于Sv39采用三层页表结构，为了得到最后映射的pte，另外创建了`uint64_t* change_pagelevel(uint64_t* pgtbl, uint64_t va)`函数，传入页表根地址和需要映射的虚拟地址，返回最终pte的地址。在该函数中根据Sv39的虚拟地址结构依次得到vpn[2]和vpn[1]，进而得到下一层页表的pte地址。每得到一层pte的地址后需要检查pte的v位，若合法则进入下一层；若不合法则需要分配`&_end + PAGE_SIZE`之后的地址创建页表。最后返回pgtbl[vpn0]的地址，即需要的最后一层pte地址。在`create_mapping`函数中对最后pte的pte的值进行修改，与物理地址形成映射。

3.3.2 设置映射

在`vm.c`中编写`paging_init`函数，`perm`映射权限可设为7。

- 调用`create_mapping`函数将内核起始（`0x80000000`）的16MB空间映射到高地址（以`0xffffffe000000000`为起始地址）。
- 对内核起始地址（`0x80000000`）的16MB空间做等值映射。
- 将必要的硬件地址（即`0x10000000`为起始地址的UART）进行等值映射，无偏移。

映射图示



代码

```
void paging_init()
{
    uint64_t *pgtbl = &_end;
  /*3.3.2 设置映射*/
    int perm = 7;
    /*kernel起始的16mb映射到高地址*/
    create_mapping(pgtbl, (uint64_t)V_KERNEL, (uint64_t)P_KERNEL, (uint64_t)MAP_SIZE,
perm);
    /*kernel起始地址的16mb做等值映射*/
    create_mapping(pgtbl,(uint64_t)P_KERNEL, (uint64_t)P_KERNEL, (uint64_t)MAP_SIZE,
perm);
    /*UART等值映射*/
    create_mapping(pgtbl, (uint64_t)UART_ADDR, (uint64_t)UART_ADDR, (uint64_t)MAP_SIZE,
perm);
}
```

Q3：为什么要进行等值映射？（不同模式下使用虚拟地址/物理地址）

答：M模式下使用物理地址，托管给S模式后S模式开启虚拟内存使用虚拟地址。

3.4 修改进程调度相关代码`sched.c`

3.4.1 必要修改

- 将各进程初始化时的`counter`和`priority`设置为相应数组中的值。

```
task[i]->counter = COUNTER_INIT_COUNTER[i];
task[i]->priority = PRIORITY_INIT_COUNTER[i];
```

- 修改所有进程均处理完毕情况下的操作：在lab3中我们为task0分配一个时间片以重设各task，在本实验中，当所有进程均运行完毕的情况下，利用`rand()`函数随机为**task[1-4]**的`counter`分配一个数值，并再次调用`schedule()`根据调度算法得到下一个需要运行的task。
- 无法使用lab3中利用openSBI完成的打印函数，需要用现有的打印函数对此做更新。
- 本次实验中仅需要编写SJF下的进程调度。其他未提及的修改自行根据代码进行调整。

3.4.2 修改task_init()调整为虚拟地址

由于开启了MMU，因此需要修改进程初始化中涉及地址的内容，将进程的地址以及进程结构体成员变量`sp`栈指针的值划分到虚拟地址空间（提示：未分配的空闲物理地址转换为虚拟地址）。将相关代码写入下方代码框。

```
task[0] =(struct task_struct*)(Kernel_Page + offset);
task[0]->thread.sp=(unsigned long long) task[0] + TASK_SIZE;
task[i]=(struct task_struct*)(Kernel_Page + offset + i*TASK_SIZE);
task[i]->thread.sp=(unsigned long long) task[i] + TASK_SIZE;
```

3.5 完成对不同section的保护

3.5.1 权限保护

修改内核起始地址到高地址的映射以及等值映射，对其中不同的段执行不同的权限保护。通过修改调用`create_mapping`时的`perm`参数，修改对内核空间不同section所在页属性的设置，完成对不同section的保护，其中text段的权限为 `r-x`, rodata段为`r--`, 其他段为 `rw-`。

参考`vmliunx.lds`中对地址的分配，使用相应的标签（如`(uint64_t)&text_start`）计算不同段的起始物理地址、起始虚拟地址及映射大小，给出修改后的`paging_init()`。

```
void paging_init()
{
    uint64_t *pgtbl = &_end;
    /*your code*/
  /*3.5 对不同section的保护*/
    /*kernel高地址映射*/
    /*text section*/
    create_mapping(pgtbl, (uint64_t)(&text_start + offset), (uint64_t)(&text_start),
(uint64_t)(TEXT_SIZE), R_ENABLE | X_ENABLE);
    /*rodata section*/
    create_mapping(pgtbl, (uint64_t)(&rodata_start + offset), (uint64_t)
(&rodata_start), (uint64_t)(RODATA_SIZE), R_ENABLE);
    /*other sections*/
    create_mapping(pgtbl, (uint64_t)(&data_start + offset), (uint64_t)(&data_start),
(uint64_t)(MAP_SIZE-TEXT_SIZE-RODATA_SIZE), R_ENABLE | W_ENABLE);

    /*kernel等值映射*/
    /*text section*/
    create_mapping(pgtbl, (uint64_t)(&text_start), (uint64_t)(&text_start), (uint64_t)
(TEXT_SIZE), R_ENABLE | X_ENABLE);
    /*rodata section*/
    create_mapping(pgtbl, (uint64_t)(&rodata_start), (uint64_t)(&rodata_start),
(uint64_t)(RODATA_SIZE), R_ENABLE);
    /*other sections*/
    create_mapping(pgtbl, (uint64_t)(&data_start), (uint64_t)(&data_start), (uint64_t)
(MAP_SIZE-TEXT_SIZE-RODATA_SIZE), R_ENABLE | W_ENABLE);

    /*UART等值映射*/
    create_mapping(pgtbl, (uint64_t)UART_ADDR, (uint64_t)UART_ADDR,
(uint64_t)PAGE_SIZE, R_ENABLE | W_ENABLE);
}
```

3.6 编译及测试

```
ZJU OS LAB 4     Student:3190102362 王奕天
task init...
[PID = 1] Process Create Successfully! counter = 1
[PID = 2] Process Create Successfully! counter = 2
[PID = 3] Process Create Successfully! counter = 3
[PID = 4] Process Create Successfully! counter = 4
[*PID = 0] Context Calculation: counter = 0
[!] Switch from task 0 [task struct: 0xffffffe000210000, sp: 0xffffffe000211000] to task 1 [task struct: 0xffffffe000211000, sp: 0xffffffe000212000], prio: 1, counter: 1
[*PID = 1] Context Calculation: counter = 1
[!] Switch from task 1 [task struct: 0xffffffe000211000, sp: 0xffffffe000212000] to task 2 [task struct: 0xffffffe000212000, sp: 0xffffffe000213000], prio: 2, counter: 2
[*PID = 2] Context Calculation: counter = 2
[*PID = 2] Context Calculation: counter = 1
[!] Switch from task 2 [task struct: 0xffffffe000212000, sp: 0xffffffe000213000] to task 3 [task struct: 0xffffffe000213000, sp: 0xffffffe000214000], prio: 3, counter: 3
[*PID = 3] Context Calculation: counter = 3
[*PID = 3] Context Calculation: counter = 2
[*PID = 3] Context Calculation: counter = 1
[!] Switch from task 3 [task struct: 0xffffffe000213000, sp: 0xffffffe000214000] to task 4 [task struct: 0xffffffe000214000, sp: 0xffffffe000215000], prio: 4, counter: 4
[*PID = 4] Context Calculation: counter = 4
[*PID = 4] Context Calculation: counter = 3
[*PID = 4] Context Calculation: counter = 2
[*PID = 4] Context Calculation: counter = 1
[PID = 1] Reset counter = 1
[PID = 2] Reset counter = 4
[PID = 3] Reset counter = 5
[PID = 4] Reset counter = 4
[*PID = 4] Context Calculation: counter = 4
[*PID = 4] Context Calculation: counter = 3
[*PID = 4] Context Calculation: counter = 2
[*PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 [task struct: 0xffffffe000214000, sp: 0xffffffe000215000] to task 1 [task struct: 0xffffffe000211000, sp: 0xffffffe000211ef0], prio: 1, counter: 1
[*PID = 1] Context Calculation: counter = 1
[!] Switch from task 1 [task struct: 0xffffffe000211000, sp: 0xffffffe000211ef0] to task 2 [task struct: 0xffffffe000212000, sp: 0xffffffe000212ef0], prio: 2, counter: 4
[*PID = 2] Context Calculation: counter = 4
[*PID = 2] Context Calculation: counter = 3
[*PID = 2] Context Calculation: counter = 2
[*PID = 2] Context Calculation: counter = 1
[!] Switch from task 2 [task struct: 0xffffffe000212000, sp: 0xffffffe000212ef0] to task 3 [task struct: 0xffffffe000213000, sp: 0xffffffe000213ef0], prio: 3, counter: 5
[*PID = 3] Context Calculation: counter = 5
[*PID = 3] Context Calculation: counter = 4
[*PID = 3] Context Calculation: counter = 3
[*PID = 3] Context Calculation: counter = 2
[*PID = 3] Context Calculation: counter = 1
```

4 讨论和心得

本次实验代码部分在地址转换的过程中需要有比较清晰的思路，在vpn获取过程中卡了一段时间，最后想起来需要使用掩码确保只得到9位vpn；在`paging_init`部分，debug时发现映射大小不能在程序中用直接段地址相减得到，相减得到的值与实际大小并不一致，因此在映射前输出段地址，相减后得到大小。

另外最让我痛苦的是写完代码后没有任何输出，但ddl后一天代码原封不动却能输出预计的结果，我不明白发生了什么，可能是刚开始内存不够？也许这就是qemu的威力吧 :)。