Report of Lab4

王睿 3180103650

1. 环境搭建

1.1 建立映射

利用前面实验的映射方法建立本地目录lab4与docker image内实验目录的映射

1.2 组织文件结构

组织文件结构如下：

```
lab4
├── arch
│   └── riscv
│       ├── 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
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── put.c
│   └── rand.c
└── Makefile
```

2. 创建映射

2.1 paging_init()的实现

我们通过`paging_init()`函数实现页表的初始化、建立物理地址与虚拟地址的映射等一系列操作。在`paging_init()`内，通过调用`create_mapping()`实现具体的映射建立。根据映射图以及后续对各section保护的要求，调用过程如下：

```
/****** 1. equal value mapping  ******/
create_mapping(kernel_pgtbl, UARTBASE, UARTBASE, PGSIZE, PTE_R | PTE_W);                          // UART direct mapping
create_mapping(kernel_pgtbl, KERNBASE, KERNBASE, (uint64)text_end - KERNBASE, PTE_R | PTE_X);             // text direct mapping
create_mapping(kernel_pgtbl, (uint64)rodata_start, (uint64)rodata_start, (uint64)rodata_end - (uint64)rodata_start, PTE_R);       // rodata direct mapping
create_mapping(kernel_pgtbl, (uint64)data_start, (uint64)data_start, (uint64)data_end - (uint64)data_start, PTE_R | PTE_W); //data direct mapping
create_mapping(kernel_pgtbl, (uint64)bss_start, (uint64)bss_start, KERNBASE + KERNSIZE - (uint64)bss_start, PTE_R | PTE_W); // bss to end of 16MB direct mapping

/****** 2. kernel mapping to high mem: 0xffffffe000000000  ******/
uint64 offset = (uint64)HIGHBASE - (uint64)KERNBASE;
create_mapping(kernel_pgtbl, KERNBASE + offset, KERNBASE, (uint64)text_end - KERNBASE, PTE_R | PTE_X);             // text direct mapping
create_mapping(kernel_pgtbl, (uint64)rodata_start + offset, (uint64)rodata_start, (uint64)rodata_end - (uint64)rodata_start, PTE_R);       // rodata direct mapping
create_mapping(kernel_pgtbl, (uint64)data_start + offset, (uint64)data_start, (uint64)data_end - (uint64)data_start, PTE_R | PTE_W); //data direct mapping
create_mapping(kernel_pgtbl, (uint64)bss_start + offset, (uint64)bss_start, KERNBASE + KERNSIZE - (uint64)bss_start, PTE_R | PTE_W); // bss to end of 16MB direct map
```

2.2 create_mapping()的实现

`create_mapping((uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm)`是页表映射的统一接口，负责建立从虚拟地址`va`与物理地址`pa`间大小为`sz`的映射，其中`pgtbl`是根页表的基地址。

因为实验采用Sv39的分配方案，支持3级页表映射，每次从`va`找到`pa`需要经过三次PTE的转换，因此我将具体从每一个`va`到`pa`的3次转换过程封装到了一个函数`uint64* setup_3mapping(uint64 *pgtbl, uint64 va)`中。

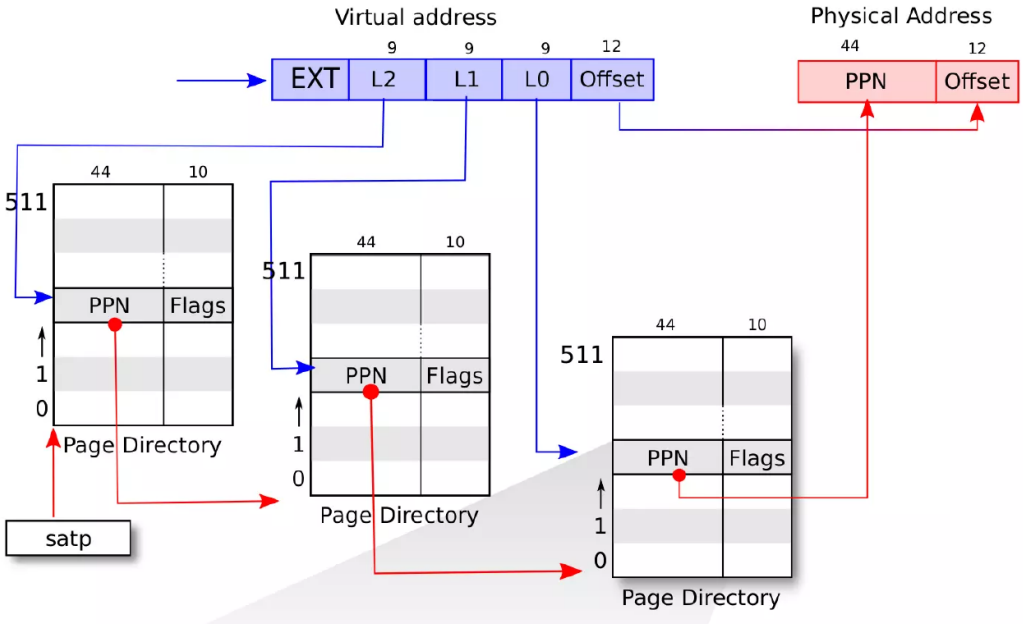
- `create_mapping()`的具体实现如下

  先将`va`与`va+sz-1`向下取整到4KB的整数倍，这样方便进行整数的空页表分配。遍历每4KB的`va`，通过`setup_3mapping()`建立与对应的`pa`的映射。并对`setup_3mapping()`返回的物理页的PTE的PPN与读写权限进行赋值。不断重复上述过程直到完成全部页表映射建立

```c
int create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm)
{
    uint64 va_begin, va_end, a;
    uint64 *pte;

    va_begin = PGROUNDDOWN(va);
    va_end = PGROUNDDOWN(va + sz - 1);

    a = va_begin;
    while(1){
        pte = setup_3mapping(pgtbl, a);
        if(pte == 0)
            return -1;
        *pte = PA2PTE(pa) | PTE_V | perm;

        if(a == va_end)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

  - `setup_3mapping()`的具体实现如下

    因为页表是Sv39机制，具体地址转换过程如下图：



因此为了获取每一级的PTE，我们先通过将`va`右移一定的位数获得对应的VPN[i]，然后通过页表目录在对应的页表中找到entry，如果该页表不存在则调用`alloc_free_page()`函数分配一个空页表，并对PTE的PPN与读写权限进行赋值；这样逐级寻址，获得最终物理页的地址，并返回物理页的PTE。

```
uint64* setup_3mapping(uint64 *pgtbl, uint64 va)
{
    if(va > MAXVA)
        puts("ERROR: va > MAXVA!\n");

    for(int level = 2; level > 0; level--){
        uint64 *pte = &pgtbl[PX(level, va)];
        if(*pte & PTE_V)
            pgtbl = (uint64*)PTE2PA(*pte);
        else{
            pgtbl = (uint64*)kalloc();
            if(pgtbl == 0)
                return 0;
            memset(pgtbl, 0, PGSIZE);
            *pte = PA2PTE(pgtbl) | PTE_V;
        }
    }
    return &pgtbl[PX(0, va)];
}
```

- `alloc_free_page()`具体实现如下

```
void *alloc_free_page(void)
{
    uint32 max_bit_index = kernel_bitmap.btmp_byte_len * 8;
    uint32 i = 0;
    for(i = 0; i < max_bit_index; i++){
        if(bitmap_test(&kernel_bitmap, i))
            break;
    }
    if(i == max_bit_index)
        return (void*)0;

    bitmap_set(&kernel_bitmap, i, 1);
    char *kaddr = (char*)(PGROUNDUP((uint64)_end) + i * PGSIZE);
    memset(kaddr, 0, PGSIZE);

    return (void*)kaddr;
}
```

从bitmap中找到为空的index所对应的页表，然后分配，分配完成后将该index设置为1，表示已被占用

我利用bitmap来实现内存空页表的管理，bitmap的数据结构定义如下：

```
struct bitmap
{
    uint32 btmp_byte_len;       // the size of bitmap in bytes
    uint8 *bits;                // start address of the bitmap
};
```

之所以将bitmap的起始地址定义为`uint8`是为了方便利用字节来更好的找到bit的index

- bitmap判断某一个index对应页表是否为空

```
int bitmap_test(struct bitmap *btmp, uint32 bit_idx) // return whether the bitmap[bit_idx] is free or not
{
    uint32 byte_index = bit_idx / 8;    // find the byte index in the bitmap
    uint32 bit_offset = bit_idx % 8;    // find the bit offset within that byte

    return btmp->bits[byte_index] & (BITMAP_MASK << bit_offset);
}
```

先找到对应index所在的的byte，然后精确定位index在该byte内的offset

- bitmap设置某一位index对应的页表为满或为空

```
void bitmap_set(struct bitmap *btmp, uint32 bit_idx, int value)
{
    if(value != 0 || value != 1)
        puts("set bitmap error!\n");

    uint32 byte_index = bit_idx / 8;    // find the byte index in the bitmap
    uint32 bit_offset = bit_idx % 8;    // find the bit offset within that byte

    if(value)
        btmp->bits[byte_index] |= (BITMAP_MASK << bit_offset);
    else
        btmp->bits[byte_index] &= ~(BITMAP_MASK << bit_offset);
}
```

3. 修改head.S

3.1 修改系统启动部分代码

- 在`_start`开头先设置`satp`寄存器为0，暂时关闭MMU

```
_start:
    # clear satp register -> disable MMU for now
    li t0, 0
    csrw satp, t0
```

- 进入S模式后，在适当位置调用`paging_init`函数进行映射
- 设置`satp`寄存器的值以打开MMU
  - 注意`satp`中的PPN字段以4KB为单位

    在vm.c中声明了`kernel_pgtbl`全局变量存储了顶级根目录的地址，并通过`paging_init()`对其进行了页分配与映射建立等操作。因此在此处设置`stap`的PPN字段时，我们读取`kernel_pgtbl`的值，右移12位即是最后的PPN字段。因为实验采用Sv39分配机制，因此需要对`satp`的mode字段赋值为8。最终实现代码如下：

```
    li t0, 8            # set satp mode = Sv39
    slli t0, t0, 60     # t0 = 8L << 60
    la t1, kernel_pgtbl # t1 = &kernel_pgtbl
    ld t2, (t1)         # t2 = value of kernel_pgtbl
    srli t1, t2, 12     # move to the lower 12 bits of satp
    or t0, t0, t1       # t0 = SATP_SV39 | (kernel_pgtbl >> 12)
    csrw satp, t0       # set satp, enable MMU
```

- 设置`stvec`为异常处理函数`trap_s`在**虚拟地址空间下**的地址

获取`trap_s`在虚拟地址空间的方法也是通过偏移。我们先读取`trap_s`的物理地址，将其减去0x80000000，然后加上虚拟高地址空间的基址0xffffffe000000000，就得到了`trap_s`的虚拟地址

```
    # write stvec = &trap_s (va)
    la t0, trap_s       # t0 = &trap_s(pa)
    li t1, 0x080000000  # t1 = KERNBASE
    sub t0, t0, t1      # t0 = &trap_s - 0x80000000 (offset)
    li t1, 0xffffffe000000000   # t1 = HIGHBASE
    add t0, t0, t1      # t0 = &trap_s(pa) - KERNBASE + HIGHBASE (t0 = &trap_s(va))
    csrw stvec, t0
```

- 设置 `sp` 的值为虚拟地址空间下的 `init_stack_top`

  方法同上

```
    # initialize stack pointer (va)
    la t0, init_stack_top   # t0 = &init_stack_top(pa)
    li t1, 0x080000000      # t1 = KERNBASE
    sub t0, t0, t1
    li t1, 0xffffffe000000000
    add t0, t0, t1          # t0 = &init_stack_top(va)
    mv sp, t0               # sp = &init_stack_top(va)
```

- 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
  - 可以先将 `start_kernel` 的虚拟地址装载在寄存器中，并使用 `jr` 指令进行跳转

```
    la t0, start_kernel # t0 = &start_kernel(pa)
    li t1, 0x080000000
    sub t0, t0, t1
    li t1, 0xffffffe000000000
    add t0, t0, t1      # t1 = &start_kernel(va)
    jr t0               # jump from S mode to start_kernel(va)
```

3.2 修改M模式下异常处理代码

- 通过 `mscratch` 寄存器保存 `stack_top` 的物理地址，并在 `trap_m` 内通过 `csrrw` 指令与 `sp` 的值进行交换，使得在 `trap_m` 内用以 `stack_top` 为顶的4KB空间作为栈空间保存寄存器的值
  - 设置 `mscratch` 寄存器的值为 `stack_top` 的物理地址

```
    la t0, stack_top        # t0 = &stack_top(pa)
    csrw mscratch, t0       # set mscratch register = &stack_top(pa)
```

  - 在 `trap_m` 入口处与返回处交换 `sp` 与 `mscratch`

```
    csrrw sp, mscratch, sp
```

4. 修改sched.c

4.1 修改task_init()调整为虚拟地址

将 `task_struct` 对应的task space由高地址调整到虚地址内，即对每一个地址空间基址加一个 `offset`

以 `current` 为例

```
    current = (uint64)0x80010000 + space_offset;
```

4.2 在进程调度时打印task_struct地址

- 为了输出16进制的值，我写了一个 `putx()` 函数，用于输出一个64位变量的十六进制值，如下

```
void putx(uint64 x)
{
    int i;
    puts("0x");
    for(i = 15; i >= 0; i--)
        *UART16550A_DR = (unsigned char)("0123456789ABCDEF"[(x >> 4 * i) & 15]);
}
```

它能输出以`0x`开头的16位的16进制数

5. 完成对不同section的保护

- 通过修改调用`create_mapping`时的`perm`参数，修改对不同section所在页属性的设置，完成对不同section的保护

```
/****** 1. equal value mapping  ******/
create_mapping(kernel_pgtbl, UARTBASE, UARTBASE, PGSIZE, PTE_R | PTE_W);                                        // UART direct mapping
create_mapping(kernel_pgtbl, KERNBASE, KERNBASE, (uint64)text_end - KERNBASE, PTE_R | PTE_X);                   // text direct mapping
create_mapping(kernel_pgtbl, (uint64)text_end, (uint64)text_end, (uint64)rodata_end - (uint64)text_end, PTE_R); // rodata direct mapping
create_mapping(kernel_pgtbl, (uint64)rodata_end, (uint64)rodata_end, (uint64)data_end - (uint64)rodata_end, PTE_R | PTE_W); //data direct mapping
create_mapping(kernel_pgtbl, (uint64)data_end, (uint64)data_end, KERNBASE + KERNSIZE - (uint64)data_end, PTE_R | PTE_W);    // bss to end of 16MB direct mapping

/****** 2. kernel mapping to high mem: 0xffffffe000000000  ******/
uint64 offset = HIGHBASE - KERNBASE;
create_mapping(kernel_pgtbl, KERNBASE + offset, KERNBASE, (uint64)text_end - KERNBASE, PTE_R | PTE_X);                   // text high mapping
create_mapping(kernel_pgtbl, (uint64)text_end + offset, (uint64)text_end, (uint64)rodata_end - (uint64)text_end, PTE_R);           // rodata high mapping
create_mapping(kernel_pgtbl, (uint64)rodata_end + offset, (uint64)rodata_end, (uint64)data_end - (uint64)rodata_end, PTE_R | PTE_W); //data high mapping
create_mapping(kernel_pgtbl, (uint64)data_end + offset, (uint64)data_end, KERNBASE + KERNSIZE - (uint64)data_end, PTE_R | PTE_W);    // bss to end of 16MB high mapping
```

- 在`head.S`中，通过修改`medeleg`寄存器，将instruction/load/store page fault托管到S模式下

  设置`medeleg`寄存器的12，13，15位

```
# set medeleg[12 & 13 & 15]
li t0, 0x0B000
csrs medeleg, t0    # enable page fault delegation
```

- 修改`strap.c`中的handler，添加对page fault的打印
  - 在`entry.S`中添加对page fault同步异常的分类

```
.align 3
s_exception:
    csrr t0, scause

    li t1, 0x08000      # t1 = bit 15
    and t2, t0, t1
    beq t1, t2, store_pgfault_exception

    li t1, 0x2000   # t1 = bit 13
    and t2, t0, t1
    beq t1, t2, load_pgfault_exception

    li t1, 0x4000   # t1 = bit 12
    and t2, t0, t1
    beq t1, t2, inst_pgfault_exception

    # modify the sepc in the stack (spec+=4)
    ld t0, 33 * WORD_SIZE(sp)
    addi t0, t0, 4
    sd t0, 33 * WORD_SIZE(sp)

    j trap_s_ret
```

  - 在`strap.c`中添加对page fault发生时的输出

```
void print_inst_pgfault(void){
    puts("ERROR: Instruction Page Fault\n");
}

void print_ld_pgfault(void){
    puts("ERROR: Load Page Fault\n");
}

void print_st_pgfault(void){
    puts("ERROR: Store Page Fault\n");
}
```

6. 实验结果

```
oslab@6646621cb857:~/lab4$ make run
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use the -bios option to avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
ZJU OS LAB 4             GROUP-32
task init...
[PID = 1] Process Create Successfully! counter = 1
[PID = 2] Process Create Successfully! counter = 4
[PID = 3] Process Create Successfully! counter = 5
[PID = 4] Process Create Successfully! counter = 4
[PID = 0] Context Calculation: counter = 0
[!] Switch from task 0 [task struct: 0xFFFFFFE000010000, sp: 0xFFFFFFE0000A8000] to task 1 [task struct: 0xFFFFFFE000011000, sp: 0xFFFFFFE0000A9000], prio: 5, counter: 1
[PID = 1] Context Calculation: counter = 1
[!] Switch from task 1 [task struct: 0xFFFFFFE000011000, sp: 0xFFFFFFE0000A9000] to task 4 [task struct: 0xFFFFFFE000014000, sp: 0xFFFFFFE0000AC000], prio: 5, 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: 0xFFFFFFE000014000, sp: 0xFFFFFFE0000AC000] to task 2 [task struct: 0xFFFFFFE000012000, sp: 0xFFFFFFE0000AA000], prio: 5, 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: 0xFFFFFFE000012000, sp: 0xFFFFFFE0000AA000] to task 3 [task struct: 0xFFFFFFE000013000, sp: 0xFFFFFFE0000AB000], prio: 5, counter: 5
[PID = 3] Context Calculation: counter = 5
```

7. 思考题

- 思考题：如何验证这些属性是否成功被保护?

  编写汇编语言分别对响应的段内的内容进行读、写与执行，可以看是否触发trap_m以及在trap_m内观察对应的mcause寄存器的值来判断是否正确保护了属性。

8. 实验心得

本次实验难度很大，我在ddl前没有赶完，期间也遇到了很多很多bug。经过本次实验我对riscv的内存映射建立有了深刻的理解，也意识到了后面的实验难度很大，应该提前安排时间完成...

实验过程中发现其实很多问题并不在于映射的建立，而在于前面lab2对中断处理的一些疏忽的操作，在当时只使用物理地址的情况下没有问题，到了现在虚地址与实地址建立映射就出现了问题。