Report of Lab5

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lab5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：

```
lab5
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   ├── sched.h
│       │   ├── syscall.h
│       │   └── vm.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   ├── vm.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   └── ...
├── init
│   └── ...
├── lib
│   └── ...
└── Makefile
```

2. 实验流程说明

总的来说，本次实验就是将`task[0-4]`看成不同的用户，每个用户都认为自己拥有全部的虚拟空间，都维护一张自己的用户顶级根页表，里面记录着自己的虚拟空间与物理空间的映射关系。因此每当发生时钟中断导致task切换时，就从lab3的普通的进程切换变成了用户的切换。同时，发生时钟中断用户会进入s mode，因此每个用户就需要一个内核栈来记录自己与内核交互的信息。这个实验的关键是用户页表的维护与切换以及内核栈、用户栈等的维护与切换。

3. 添加系统调用处理函数

本实验中将`handler_s`统一为`handler_s(size_t scause, size_t sepc, uintptr_t *regs);`其中regs概念对应linux中的pt_regs，将上下文内容所在的地址传入处理函数，也就是`trap_s`内的栈帧sp

3.1 handler_s()的实现

`handler_s()`是s mode下处理中断与异常的统一接口，因此首先我们通过最高位是否为1区分中断与异常。到本次实验为止，我们需要考虑的中断仅有时钟中断，而异常包含page fault异常以及ecall from U mode。我们可以通过Exception Code来区分不同的异常类型。因此`handler_s()`实现如下：

```
void handler_s(size_t scause, size_t sepc, uintptr_t *regs)
{
    if (scause >> 63) {     // interrupt
        if ( ( (scause << 1) >> 1 ) == 5 ) {    // supervisor timer interrupt
            asm volatile("ecall");
            do_timer();
        }
    }
    else {              // exception
        temp_regs = regs;

        if(scause == 15){   // store pagefault exception
            puts("ERROR: Store Page Fault\n");
        }
        else if(scause == 13){  // load pagefault exception
            puts("ERROR: Load Page Fault\n");
        }
        else if(scause == 12){  // instruction pagefault exception
            puts("ERROR: Instruction Page Fault\n");
        }
        else if(scause == 8){   // ecall from U mode
            uint64 a0, a1, a2, a7;
            asm volatile("ld %0, 17*8(%1)":"+r"(a7):"r"(regs)); // read a7

            if(a7 == SYS_WRITE){
                asm volatile("ld %0, 10*8(%3)\n"
                             "ld %1, 11*8(%3)\n"
                             "ld %2, 12*8(%3)\n"
                             : "+r"(a0), "+r"(a1), "+r"(a2) :"r"(regs));
                sys_write(a0, a1, a2);
            }
            else if(a7 == SYS_GETPID){
                sys_getpid();
                asm volatile("sd a0, 10*8(%0)"::"r"(regs));
            }
        }


        asm volatile("sd %0, 32*8(%1)"::"r"(sepc + 4), "r"(regs));  // modify the sepc->sepc+4
        }
    return;
}
```

值得注意的是，在`handler_s()`内若判断为ecall from U mode类型的异常，我们需要根据寄存器`a7`来进一步区分所调用的系统调用处理函数类型。但此时我们不能直接读寄存器`a0-a7`，因为之前执行的程序可能会修改寄存器，所以我们只能从`trap_s`的栈内读取参数`a0-a5`以及系统调用类型`a7`。这也意味着在SYS_GETPID中将得到的pid存入寄存器`a0`后，我们需要将这个值存入`regs`的上下文中，否则pid会是乱码。

3.2 sys_write()的实现

- 64 号系统调用 `sys_write(unsigned int fd, const char* buf, size_t count)`
  该调用将用户态传递的字符串打印到屏幕上，此处`fd`为标准输出（1），`buf`为用户需要打印的起始地址，`count`为字符串长度，返回打印的字符数。

  具体实现方法如下:

  我们首先要进入用户页表，这样才能访问buf的内容，这样就需要暂存一下所需要的buf以及count值，并在返回前回到内核页表下

```
void  sys_write(unsigned int fd, const char* buf, size_t count)
{
    temp_buf = buf;
    temp_count = count;

    write_csr(satp,MAKE_SATP(current->mm->upgtbl));
    asm volatile("sfence.vma");

    for(int i = 0; i < temp_count; i++)
        putchar(temp_buf[i]);

    write_csr(satp,0x8000000000080ffe);
    asm volatile("sfence.vma");
    asm volatile("mv a0, %0"::"r"(count));
}
```

3.3 sys_getpid()的实现

- 172 号系统调用 `sys_getpid()`
  该调用从`current`中获取当前的`pid`放入`a0`中返回，无参数。

```
extern struct task_struct* current;
void sys_getpid(void)
{
    long curPid = current->pid;

    asm volatile("mv a0, %0"::"r"(curPid));
}
```

4. 修改进程初始化以及进程调度相关逻辑

4.1 task struct的修改

为了支持用户态我们需要对task struct进行修改。

- 加入`struct mm_struct`

  `mm_struct`记录了用户与虚拟内存映射相关的内容，包括进程的顶级根页表地址、用户的用户栈以及用户的内核栈

  ```
  typedef struct MM_STRUCT{
      unsigned long long *upgtbl;      // user top page table
      unsigned long long ustack;      // page table of user stack (physical)
      unsigned long long kstack;      // page table of kernel stack (virtual)
      int size;
  }mm_struct;
  ```

- `task[i].thread`结构体中加入一些重要CSR寄存器

  为了记录当前用户的一些状态，在原来的上下文切换保存的寄存器的基础上，我们还需要添加`sepc`，`sscratch`，`sstatus`以及`satp`

```c
/* 进程状态段数据结构 */
struct thread_struct {
    unsigned long long ra;
    unsigned long long sp;
    unsigned long long s0;
    unsigned long long s1;
    unsigned long long s2;
    unsigned long long s3;
    unsigned long long s4;
    unsigned long long s5;
    unsigned long long s6;
    unsigned long long s7;
    unsigned long long s8;
    unsigned long long s9;
    unsigned long long s10;
    unsigned long long s11;
    unsigned long long sepc;       // 保存的sepc
    unsigned long long sscratch;   // 保存的sscratch
    unsigned long long sstatus;    // 保存的sstatus
    unsigned long long satp;       // 保存的satp
};
```

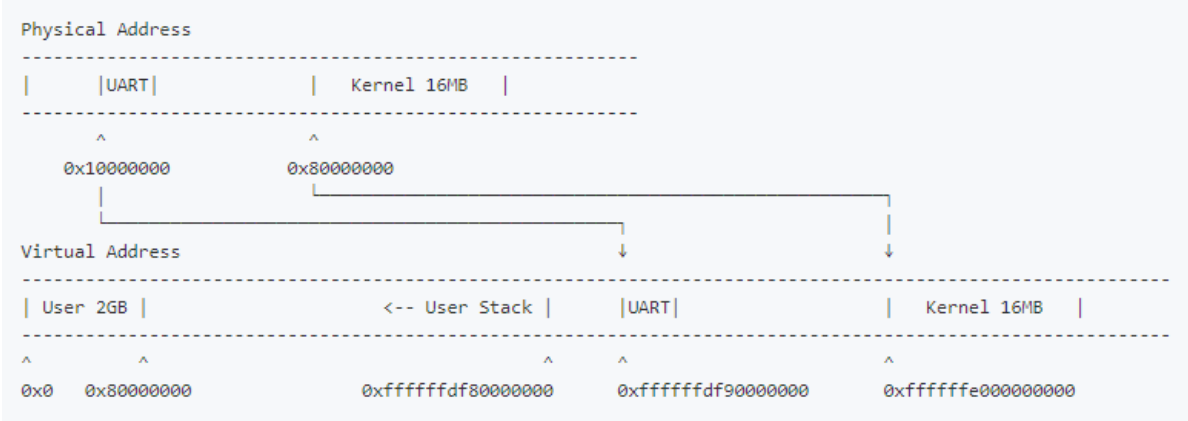
总的`task_struct`结构体如下

```c
/* 进程数据结构 */
struct task_struct {
    long state;    // 进程状态 Lab3中进程初始化时置为TASK_RUNNING
    long counter;  // 运行剩余时间
    long priority; // 运行优先级 1最高 5最低
    long blocked;
    long pid;      // 进程标识符
   // Above Size Cost: 40 bytes

    struct thread_struct thread; // 该进程状态段
    mm_struct *mm;     //虚拟内存映射相关
};
```

4.2进程初始化

由于增加了用户态支持，用户的虚拟地址映射方式如下，虚拟地址0x0-0x80000000供用户态使用，0xffffffdf90000000-0xffffffe000000000作为设备地址（UART等）、以及内核用于进行虚拟地址映射的地址，0xffffffe000000000开始的地址直接与物理内存进行映射。



4.2.1 用户栈的映射建立

如上图所示，用户栈顶的虚拟地址都统一为 0xffffffdf80000000，因此在建立映射时我们先通过`kalloc()`在物理空间中寻找一个空闲的page，然后分配给该用户作为用户栈，并通过`kvmmap()`在用户的顶级根页表内建立从用户的虚拟地址到物理地址的映射。需要注意的是，建立映射时的其实虚拟地址应该为用户栈顶的虚拟地址减去一个PAGE_SIZE。需要注意的是，这里的权限需要有`PTE_U`，这样用户态才能使用。具体实现如下:

```
//分配用户栈的物理页 user stack
task[i]->mm->ustack = kalloc();
if(task[i]->mm->ustack == 0)
    panic("kalloc");
kvmmap(task[i]->mm->upgtbl, (uint64)USTACK_TOP - PAGE_SIZE, (uint64)(task[0]->mm->ustack), PAGE_SIZE, PTE_R | PTE_W | PTE_U);
```

4.2.2 内核栈的映射建立

用户的内核栈的建立与用户栈相似，我们通过`kalloc()`获得一个page的物理空间作为内核栈，并指定某一个虚拟空间地址作为内核栈顶，然后通过调用`kvmmap()`来建立映射。同时，在`mm_struct`内记录内核栈顶的指针。具体实现如下:

```
// 用户态内核栈映射
kstack_pa = kalloc();
kstack_va = PA2VA(PHY_END)- 5 * PAGE_SIZE;
kvmmap(task[i]->mm->upgtbl, kstack_va, kstack_pa, PAGE_SIZE, PTE_R | PTE_W | PTE_U);
task[i]->mm->kstack = kstack_va + PAGE_SIZE;
```

4.2.3 用户态程序的映射建立

我通过一个函数`uvminit()`来完成用户态可执行程序的映射。函数定义如下:

```
void uvminit(pagetable_t pagetable)
{
  char* mem = 0x84000000;
  mappages(pagetable, 0, PAGE_SIZE, (uint64)mem, PTE_W|PTE_R|PTE_X|PTE_U);

}
```

要注意的是这里的权限为`PTE_W|PTE_R|PTE_X|PTE_U`

这样我们就可以通过调用`uvminit(task[i]->mm->upgtbl)`来将用户态程序映射到用户虚拟空间0x0起始处。

4.2.4 用户态中内核页表的建立

我们还需要建立内核页表，建立方式很简单，只需要复制内核页表映射的方式即可，只不过此时的pagetable从`kpgtbl`变成了`task[i]->mm->upgtbl`。同时我们只需映射到bss_end，后面的虚拟空间留给存储用户的变量使用。

```
// 内核页表映射
// map devices
uart = PA2VA(get_device_addr(UART_MMIO));
kvmmap(task[i]->mm->upgtbl, uart, VA2PA(uart), get_device_size(UART_MMIO), (PTE_R | PTE_W));

poweroff = PA2VA(get_device_addr(POWEROFF_MMIO));
kvmmap(task[i]->mm->upgtbl, poweroff, VA2PA(poweroff), get_device_size(POWEROFF_MMIO), (PTE_R | PTE_W));

// map kernel text executable and read-only.
kvmmap(task[i]->mm->upgtbl, (uint64)&text_start, VA2PA((uint64)&text_start), (uint64)&text_end - (uint64)&text_start, (PTE_R | PTE_X));
// map kernel data and the physical RAM we'll make use of.
kvmmap(task[i]->mm->upgtbl, (uint64)&rodata_start, VA2PA((uint64)&rodata_start), (uint64)&rodata_end - (uint64)&rodata_start, (PTE_R));
kvmmap(task[i]->mm->upgtbl, (uint64)&data_start, VA2PA((uint64)&data_start), (uint64)&data_end - (uint64)&data_start, (PTE_R | PTE_W));
kvmmap(task[i]->mm->upgtbl, (uint64)&bss_start, VA2PA((uint64)&bss_start), (uint64)&bss_end - (uint64)&bss_start, (PTE_R | PTE_W));
```

4.2.5 进程thread结构体内重要寄存器赋值

在task初始化的末尾，我们需要设置一些寄存器，使得程序能够正确执行

- task[i]->thread.sp

  我们设置用户态的sp为用户栈的栈顶（虚拟地址），表示后续执行用户态程序时将用户栈作为自己的栈

- task[i]->threa.ra

  我们设置ra为`__sret`，这样当`switch_to`的最后执行`ret`就能进入`__sret`，在`__sret`内设置了`sepc=0x0`，并执行`sret`，从而进入用户态程序

- task[i]->thread.satp

  我们通过用户的`upgtbl`来设置当前用户的satp，并记录。

- task[i]->thread.sscratch

  我们将用户的内核栈的sp存在sscratch中，当进入trap_s时要通过交换sscratch与sp来将sp由用户栈指针换成内核栈指针

```
// 进程thread内重要寄存器赋值
task[i]->thread.sp = (uint64)USTACK_TOP;
task[i]->thread.ra = (unsigned long long) & __sret;
task[i]->thread.satp = MAKE_SATP(task[i]->mm->upgtbl);
task[i]->thread.sscratch = kstack_va + PAGE_SIZE;
```

4.3 内核态与用户态的切换

内核态与用户态的切换是本实验最复杂也是最容易出错的地方。我们需要明确在刚进入trap_s处理异常时，我们处于用户页表下，而进入handler_s时，我们需要切换为内核页表，因为只有内核页表才有所有task[i]的pcb信息。在返回时trap_s恢复上下文时，就又需要恢复为用户页表。同时我们还需要注意切换内核栈与用户栈，这也意味着sp的切换与保存，这一部分的内容与lab3联系紧密，如果当时进程切换有些概念不清楚，就会产生大麻烦。

这一部分涉及到比较多的代码，就不一一展示，具体可以看工程文件。

- trap_s

```
trap_s:
    csrrw tp , sscratch , tp
    addi tp , tp, -36*reg_size
    sd t0,5*reg_size(tp)
    mv t0,sp
    sd t0,2*reg_size(tp)
    mv sp,tp
    addi tp , tp ,36*reg_size
    csrrw tp,sscratch,tp

    sd x1, 1*reg_size(sp)
    #sd x2, 2*reg_size(sp)
    sd x3, 3*reg_size(sp)
    sd x4, 4*reg_size(sp)
    #sd x5, 5*reg_size(sp)
    sd x6, 6*reg_size(sp)
    sd x7, 7*reg_size(sp)
    sd x8, 8*reg_size(sp)
    sd x9, 9*reg_size(sp)
    sd x10, 10*reg_size(sp)
    sd x11, 11*reg_size(sp)
    sd x12, 12*reg_size(sp)
    sd x13, 13*reg_size(sp)
    sd x14, 14*reg_size(sp)
    sd x15, 15*reg_size(sp)
    sd x16, 16*reg_size(sp)
    sd x17, 17*reg_size(sp)
    sd x18, 18*reg_size(sp)
    sd x19, 19*reg_size(sp)
    sd x20, 20*reg_size(sp)
    sd x21, 21*reg_size(sp)
    sd x22, 22*reg_size(sp)
    sd x23, 23*reg_size(sp)
    sd x24, 24*reg_size(sp)
    sd x25, 25*reg_size(sp)
```

```
    sd x26, 26*reg_size(sp)
    sd x27, 27*reg_size(sp)
    sd x28, 28*reg_size(sp)
    sd x29, 29*reg_size(sp)
    sd x30, 30*reg_size(sp)
    sd x31, 31*reg_size(sp)

    csrr t0, sepc
    sd t0, 32*reg_size(sp)

    csrr t0, sstatus
    sd t0, 33 * reg_size(sp)

    csrr t0,sscratch
    sd t0,34*reg_size(sp)

    csrr t0, satp
    sd t0, 35*reg_size(sp)

    li t0,KSATP
    csrw satp,t0
    sfence.vma

    csrr a0, scause
    csrr a1, sepc
    mv a2 , sp

    li t0,0x40000
    csrs sstatus,t0


    call handler_s

    ld t0, 35*reg_size(sp)
    csrw satp,t0
    sfence.vma

    ld t0,34*reg_size(sp)
    csrw sscratch,t0

    ld t0, 33*reg_size(sp)
    csrw sstatus,t0
    li t0,0x100
    csrc sstatus,t0

    ld t0, 32*reg_size(sp)
    csrw sepc, t0

    ld x1, 1*reg_size(sp)
    #ld x2, 2*reg_size(sp)
    ld x3, 3*reg_size(sp)
    ld x4, 4*reg_size(sp)
    #ld x5, 5*reg_size(sp)
    ld x6, 6*reg_size(sp)
    ld x7, 7*reg_size(sp)
    ld x8, 8*reg_size(sp)
    ld x9, 9*reg_size(sp)
    ld x10, 10*reg_size(sp)
```

```
    ld x11, 11*reg_size(sp)
    ld x12, 12*reg_size(sp)
    ld x13, 13*reg_size(sp)
    ld x14, 14*reg_size(sp)
    ld x15, 15*reg_size(sp)
    ld x16, 16*reg_size(sp)
    ld x17, 17*reg_size(sp)
    ld x18, 18*reg_size(sp)
    ld x19, 19*reg_size(sp)
    ld x20, 20*reg_size(sp)
    ld x21, 21*reg_size(sp)
    ld x22, 22*reg_size(sp)
    ld x23, 23*reg_size(sp)
    ld x24, 24*reg_size(sp)
    ld x25, 25*reg_size(sp)
    ld x26, 26*reg_size(sp)
    ld x27, 27*reg_size(sp)
    ld x28, 28*reg_size(sp)
    ld x29, 29*reg_size(sp)
    ld x30, 30*reg_size(sp)
    ld x31, 31*reg_size(sp)

    ld t0, 5*reg_size(sp)
    ld sp, 2*reg_size(sp)

    sret
```

5. 用户态程序测试

我们通过`-initrd`选项将用户态程序的二进制镜像加载到物理内存的0x84000000处，并在task初始化时映射到用户页表的虚拟空间内，这样用户就能够执行自己的用户态代码。

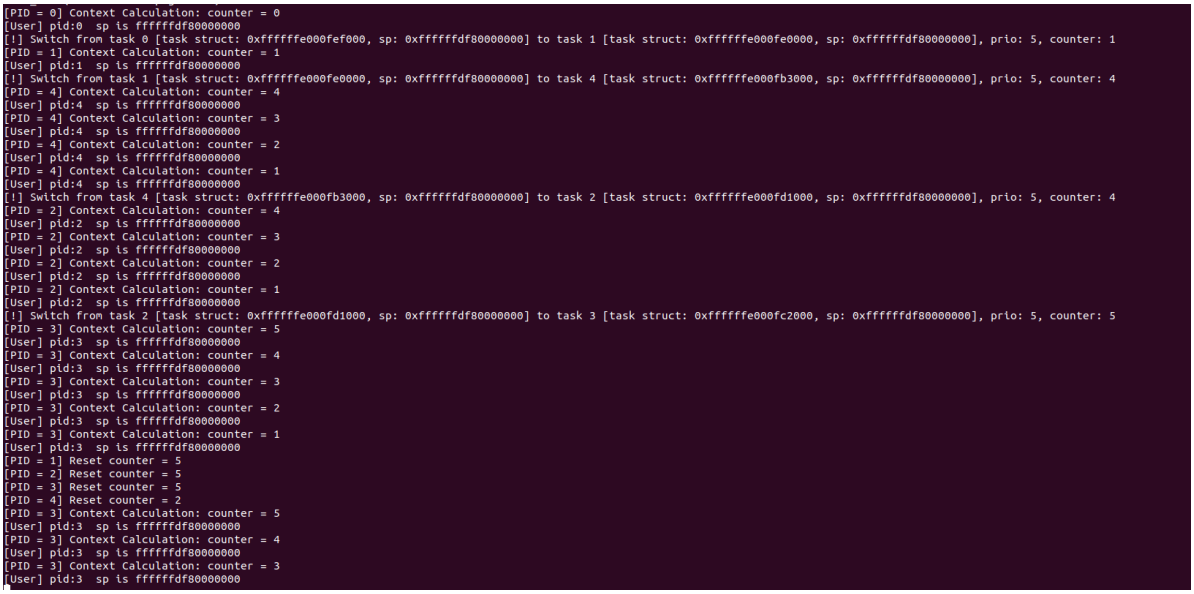
`hello.bin`中关键部分的代码如下，可以看到程序在main中执行了一次172号系统调用（SYS_GETPID）来获得进程的pid，并在printf中执行了64号系统调用（SYS_WRITE）来打印字符串

```
0000000000000004 <main>:
   4:   fd010113                addi    sp,sp,-48
   8:   02113423                sd      ra,40(sp)
   c:   02813023                sd      s0,32(sp)
  10:   00913c23                sd      s1,24(sp)
  14:   01213823                sd      s2,16(sp)
  18:   01313423                sd      s3,8(sp)
  1c:   00010993                mv      s3,sp
  20:   00000917                auipc   s2,0x0
  24:   3b890913                addi    s2,s2,952 # 3d8 <printf+0x358>
  28:   fff00493                li      s1,-1
  2c:   0ac00893                li      a7,172
  30:   00000073                ecall
  34:   00050413                mv      s0,a0
  38:   00098613                mv      a2,s3
  3c:   00040593                mv      a1,s0
  40:   00090513                mv      a0,s2
  44:   03c000ef                jal     ra,80 <printf>
  48:   00048793                mv      a5,s1
  4c:   fff7879b                addiw   a5,a5,-1
```

```
  50:   fe079ee3                bnez    a5,4c <main+0x48>
  54:   fd9ff06f                j       2c <main+0x28>

000000000000004c <printf>:
  ...
 380:   04000893                li      a7,64
 384:   00078513                li      a0,1
 388:   00070593                mv      a1,a4 # buf
 38c:   00068613                mv      a2,a3 # count
 390:   00000073                ecall
```

6. 实验结果



7. 实验心得

个人感觉本次实验难度相当大，可能实验手册相对比较简单也是原因之一，不过这也让我在完成实验后感到收获也非常大。通过本次实验我对用户态有了很深刻的理解，对用户的用户栈和内核栈的切换、用户间的切换、虚拟地址与物理地址映射、内核页表与用户页表等有了更加全面的认识