Lab 5: RV64 用户态程序

1. 实验目的

- 创建用户态进程，并设置 sstatus 来完成内核态转换至用户态。
- 正确设置用户进程的用户态栈和内核态栈， 并在异常处理时正确切换。
- 补充异常处理逻辑，完成指定的系统调用（ SYS_WRITE, SYS_GETPID ）功能。

2. 实验环境

Environment in previous labs

3. 实验步骤和原理分析

3.1 修改 task_init

- 对每个用户态进程，其拥有两个 stack： U-Mode Stack 以及 S-Mode Stack， 其中 S-Mode Stack 在 lab3 中我们已经设置好了。我们可以通过 alloc_page 接口申请一个空的页面来作为 U-Mode Stack 。其原因为：**当用户态程序在用户态运行时，其使用的栈为用户态栈，当调用SYSCALL时候，陷入内核处理时使用的栈为内核态栈，因此需要区分用户态栈和内核态栈，并在异常处理的过程中需要对栈进行切换**
- 为每个用户态进程创建自己的页表（**要求**）并将 uapp 所在页面，以及 U-Mode Stack 做相应的映射；同时为了避免 U-Mode 和 SMode切换的时候切换页表，我们也将内核页表 （ swapper_pg_dir ） 复制到每个进程的页表中。注意程序运行过程中，有部分数据不在栈上，而在初始化的过程中就已经被分配了空间（比如我们的 uapp 中的 counter 变量），所以二进制文件需要先被拷贝到一块某个进程专用的内存之后再进行映射，防止所有的进程共享数据，造成期望外的进程间相互影响。
- 对每个用户态进程我们需要将 sepc 修改为 USER_START，配置修改好 sstatus 中的 **SPP** （使得 sret 返回至 U-Mode ），**SPIE** （ sret 之后开启中断 ）， **SUM** （ S-Mode 可以访问 User 页面 ）， sscratch 设置为 U-Mode 的 sp，其值为**USER_END** （即 U-Mode Stack 被放置在 user space 的最后一个页面）。
- 修改 __switch_to， 需要加入保存/恢复 sepc sstatus sscratch 以及切换页表的逻辑。
- 在切换了页表之后，需要通过 fence.i 和 vma.fence 来刷新 TLB 和 ICache。

```
    // 将 sepc 修改为 USER_START
     task[i]->thread.sepc = USER_START;
    // 配置 sstatus 中的 :
    // 1. SPP （ 使得 sret 返回至 U-Mode ） spp = 0
    // 2. SPIE （ sret 之后开启中断 ） spie = 1
    // 3. SUM （ S-Mode 可以访问 User 页面 ） sum = 1
    uint64 sstatus = csr_read(sstatus);
    sstatus &= ~(1<<8);     // set sstatus[SPP] = 0
    sstatus |= 1<<5;        // set sstatus[SPIE] = 1
    sstatus |= 1<<18;       // set sstatus[SUM] = 1
    task[i]->thread.sstatus = sstatus;
    // sscratch 设置为 U-Mode 的 sp，其值为 USER_END
    task[i]->thread.sscratch = USER_END;

    // 创建
    // 将 uapp 所在页面，以及 U-Mode Stack 做相应的映射，
```

```
    uint64 user_stack = alloc_page();
    create_mapping((uint64*)task[i]->pgd, USER_END - PGSIZE,
                   user_stack - PA2VA_OFFSET, PGSIZE, 0x17);

    // 创建页表,复制内核页表
    task[i]->pgd = (pagetable_t)alloc_page();
    memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);

    //以下是修改的_swtich_to
    ld s0,152(t0)
    csrw sepc,s0
    ld s0,160(t0)
    csrw sstatus,s0
    ld s0,168(t0)
    csrw sscratch,s0
```

3.2 修改中断入口/返回逻辑 (_trap) 以及中断处理函数 (trap_handler)

RISC-V 中只有一个栈指针寄存器(sp)，因此需要我们来完成用户栈与内核栈的切换。

__dummy

我们初始化时， thread_struct.sp 保存了 S-Mode sp ， thread_struct.sscratch 保存了 U-Mode sp ， 因此在 S-Mode -> U->Mode 的时候，我们只需要交换对应的寄存器的值即可

```
    csrrw sp, sscratch, sp
    sret
```

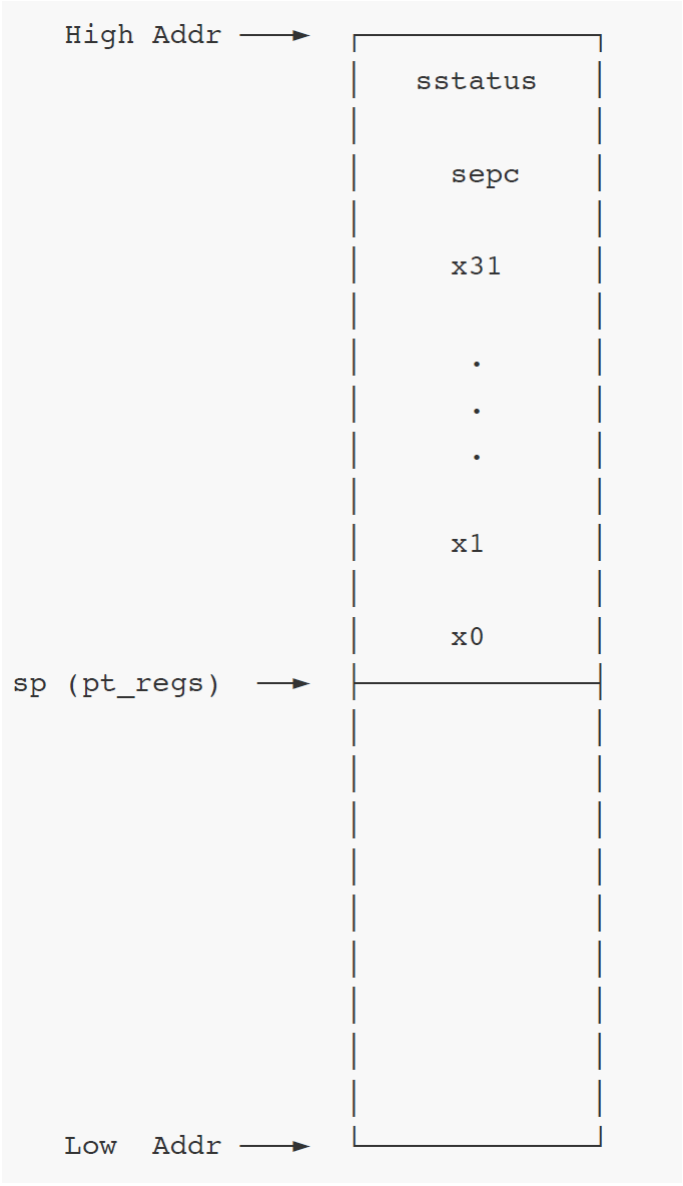
_trap

同理，在 _trap 的首尾我们都需要做类似的操作。注意如果是内核线程(没有 U-Mode Stack) 触发了异常，则不需要进行切换。

```
_traps:
    csrr t0,sscratch
    beq t0,x0,switch_entry
    csrrw sp,sscratch,sp
```

trap_handler

在 _trap 中我们将寄存器的内容连续的保存在 S-Mode Stack上， 因此我们可以将这一段看做一个叫做 pt_regs 的结构体。我们可以从这个结构体中取到相应的寄存器的值（比如 syscall 中我们需要从 a0 ~ a7 寄存器中取到参数 ）。这个结构体中的值也可以按需添加，同时需要在 _trap 中存入对应的寄存器值以供使用。



首先，完善pt_regs的定义，包括寄存器和sepc、sstatus。

```
struct pt_regs{
    uint64 regs[31]; // regs[0] -> x1
    uint64 sepc;
    uint64 sstatus;
};
```

其次，在trap_handler中，完善此处的对ecall的处理，判断类型，并且跳转处理，以下列出了跳转执行的接口，将会在下一个环节列出实现的细节。

```
    if (scause == ecall_U){ //ecall from User-mode
            // printk("ecall trap,");
            // uint64 sys_write(unsigned int fd, const char* buf, size_t count)
            // 系统调用号放在a7,返回值需要放在a0,三个参数分别a0,a1,a2
            // uint64 sys_getpid()
            // 系统调用号放在a7,返回值需要放在a0
            switch (regs->regs[16]) {
            case SYS_WRITE:
                // printk("it's sys_write\n");
                ret = sys_write(regs->regs[9],regs->regs[10],regs->regs[11]);
                break;
            case SYS_GETPID:
```

```
                // printk("it's sys_getpid\n");
                ret = sys_getpid();
                break;
            default:
                printk("other trap,scause = 0x%lx",scause);
                break;
            }
            // 让sepc+4，跳过ecall,执行下一条
            // 返回值放入结构体中
            regs->sepc = regs->sepc + 0x4;
            regs->regs[9] = ret;
        }
```

3.3 添加系统调用

- 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。(具体见user/printf.c)
- 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。（ 具体见 user/getpid.c)
- 增加 syscall.c syscall.h 文件， 并在其中实现 getpid 以及 write 逻辑。
- 系统调用的返回参数放置在 a0 中 (不可以直接修改寄存器， 应该修改 regs 中保存的内容)。
- 针对系统调用这一类异常， 我们需要手动将 sepc + 4 （ sepc 记录的是触发异常的指令地址， 由于系统调用这类异常处理完成之
  后， 我们应该继续执行后续的指令，因此需要我们手动修改 sepc 的地址，使得 sret 之后 程序继续执行），关于这一点的实现在上面一个环节的代码最后完成。

  具体实现如下：

```
unsigned long sys_write(unsigned int fd, const char* buf, size_t count) {
    if (fd == 1) {
        puts(buf);
        return count;
    } else {
        return 0;
    }
}

unsigned long sys_getpid() {
    return current->pid;
}
```

3.4 修改 head.S 以及 start_kernel

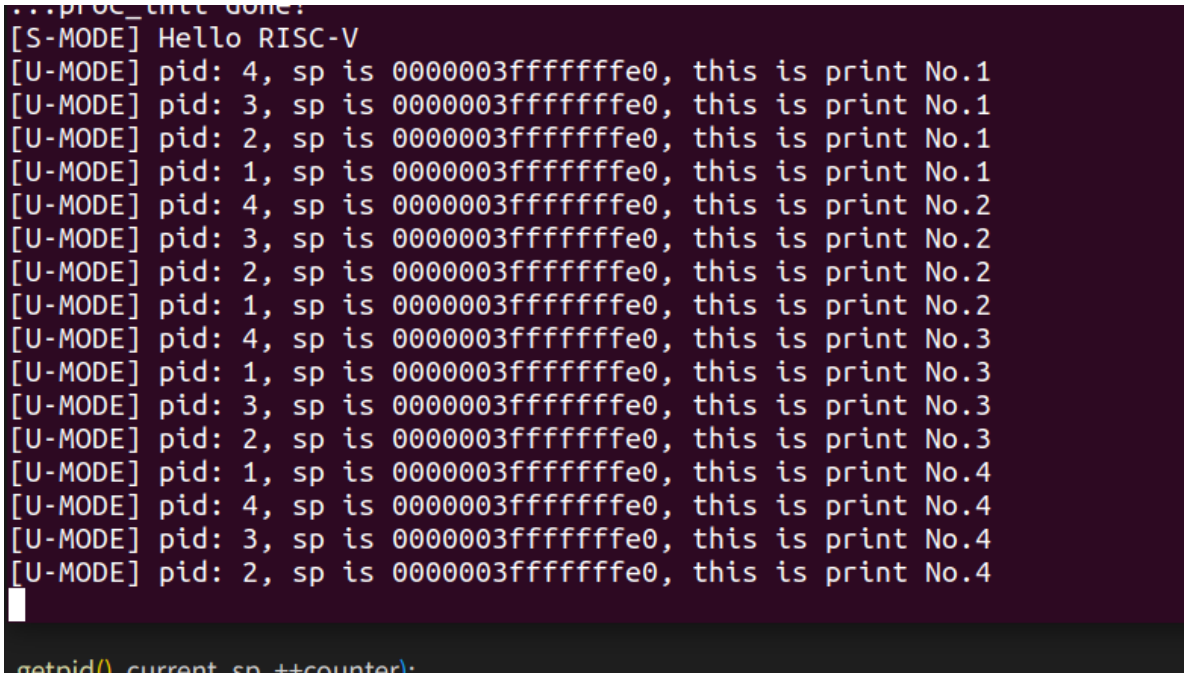
- 在之前的 lab 中， 在 OS boot 之后，我们需要等待一个时间片，才会进行调度。我们现在更改为 OS boot 完成之后立即调度 uapp 运行。
- 在 start_kernel 中调用 schedule() 注意放置在 test() 之前。
- 将 head.S 中 enable interrupt sstatus.SIE 逻辑注释，确保 schedule 过程不受中断影响

4. 测试与结果

4.1 实验测试结果

```
docker start oslab22
docker exec -it oslab22 /bin/bash
cd /home/oslab/os22fall-stu/src/lab5/lab5
export RISCV=/opt/riscv
export PATH=$PATH:$RISCV/bin
make run //运行
make debug //调试
riscv64-unknown-linux-gnu-gdb vmlinux //调试使用语句
layout src//查看源码
cd ./arch/riscv/kernel//完成最后一个要求，查看异常
i r ra//查看单个寄存器的值
target remote localhost:1234//开启调试
readelf -a uapp//查看
```

测试结果符合要求，成功切换U-MODE，执行程序



4.2 添加ELF支持

```
static uint64 load_program(uint64 cnt) {
    // printk("task id:%d\n",task[cnt]->pid);
    Elf64_Ehdr *ehdr = (Elf64_Ehdr*)uapp_start;
    uint64 phdr_start = (uint64_t)ehdr+ehdr->e_phoff;
    int phdr_cnt = ehdr->e_phnum;
    Elf64_Phdr *phdr;
    int load_phdr_cnt = 0;
    for (int i = 0; i < phdr_cnt; i++) {
        phdr= (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) *i);
        if (phdr->p_type == PT_LOAD) {
            // copy the program section to another space
            uint64 mem_sz = phdr->p_memsz;
            uint64 file_sz = phdr->p_filesz;
            uint64 pg_cnt = get_page_cnt(mem_sz);
            uint64 user_space = alloc_page(pg_cnt);

            uint64 ph_start = (uint64)uapp_start + phdr->p_offset;
            uint64 offset = phdr->p_vaddr & 0xfff;
            memset((char*)user_space, 0, pg_cnt*PGSIZE);
```

```
            memcpy((char*)user_space+offset,(char*)ph_start,file_sz);
            // memset((char*)(user_space + file_sz), 0, mem_sz - file_sz);

            // mapping the program section with corresponding size and flag
            uint64 va = phdr->p_vaddr;

            uint64 pa = user_space - PA2VA_OFFSET;
            uint64 flag = (phdr->p_flags <<1)| 0x11;
            // printk("flag = 0x%lx\n",flag);
            create_mapping((uint64*)(task[cnt]->pgd),va,pa,mem_sz,flag);
            // printk("load_program part %d finish\n",i);
        }
    }
    // pc for the user program
    task[cnt]->thread.sepc = ehdr->e_entry;
}
```

5. 实验心得与思考题

1. 我们在实验中使用的用户态线程和内核态线程的对应关系是怎样的？（一对一，一对多，多对一还是多对多）

   多对一

2. 为什么 Phdr 中， p_filesz 和 p_memsz 是不一样大的?

   p_filesz 和 p_memsz 之间的大小差异可能是由于可执行文件中的数据在加载到内存中之前需要被解压缩或者初始化的原因造成的。

3. 为什么多个进程的栈虚拟地址可以是相同的？用户有没有常规的方法知道自己栈所在的物理地址?

   在虚拟内存系统中，操作系统通过地址转换机制将虚拟地址转换为物理地址，以便硬件能够访问进程的内存。 因此，每个进程都有自己的虚拟地址空间，所以虚拟地址可以相同，但是实际上所有进程共享一个物理内存空间。用户进程通常没有办法知道自己栈所在的物理地址。这是因为虚拟地址转换的过程是由操作系统完成的，而用户进程无法直接访问操作系统的地址转换机制。

实验心得：

在本次实验中，要实现的内容较为复杂。虽然我能根据实验指导书上的要求逐个模块的去完成，但是在整体理解上还是碰到了不少麻烦。我觉得在修改中断和前半部分的task_init中我们要处理的都不算特别复杂，但是这个elf却是实实在在的要花很多心思去理解，去弄懂其中的一些映射关系。多亏了指导书里已经较为全面的提纲（和google的指点，百度确实不行），我较好的完成了本次实验。