浙江大学

本科实验报告

课程名称：	操作系统
实验名称：	RV64 用户态程序
姓　　名：	蔡雨谦
学　　院：	计算机学院
系：	计算机系
专　　业：	计算机科学与技术
学　　号：	3210102466
指导教师：	寿黎但

2023 年　　12 月　　7 日

浙江大学实验报告

一、 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。

二、 实验环境

- Ubuntu 22.04.2 LTS Windows Subsystem for Linux 2

三、 操作方法与实验步骤

1. 准备工程

从课程提供的 github 仓库获取实验代码框架，在 lab3 的基础上准备进行后续实验。本次实验需要修改或完成的文件有：

- Vmlinux.lds
- Head.s
- Def.h

修改 vmlinux.lds，将用户态程序 uapp 加载至 .data 段。按如下修改：

```
.data : ALIGN(0x1000){
    _sdata = .;

    *(.sdata .sdata*)
    *(.data .data.*)

    _edata = .;

    . = ALIGN(0x1000);
    _sramdisk = .;
    *(.uapp .uapp*)
    _eramdisk = .;
    . = ALIGN(0x1000);

} >ramv AT>ram
```

需要修改 defs.h，在 defs.h 添加 如下内容：

```
#define USER_START (0x0000000000000000) // user space start virtual address
#define USER_END   (0x0000004000000000) // user space end virtual address
```

2. 创建用户态进程

修改后的 thread_struct：

```
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];

    uint64 sepc, sstatus, sscratch;
};
```

修改后的 task_struct：

```
struct task_struct {
    struct thread_info* thread_info;//可以尝试删除并修改__switch_to
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
    uint64 pid;      // 线程id
    /*size = 6 * 8 = 48*/

    struct thread_struct thread;

    pagetable_t pgd;
};
```

修改 task_init 函数：

将 sepc 设置为 USER_START。配置 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）。将 sscratch 设置为 U-Mode 的 sp，其值为 USER_END。

为每个用户态进程分配单独的页表，并把内核页表复制进这些页表中。将 uapp 所在的页面映射到每个进行的页表中。

```
//初始化我们刚刚在 thread_struct 中添加的三个变量
task[i]->thread.sepc = USER_START;
uint64 sstatus = csr_read(sstatus);
task[i]->thread.sstatus = sstatus & ~(0x100) | 0x40020;//spp[8]=0   spie[5]=1  sum[18]=1
task[i]->thread.sscratch = USER_END;
//每个用户态进程都分配单独的页表
task[i]->pgd=(uint64*)alloc_page();
//将内核页表 （ swapper_pg_dir ） 复制到每个进程的页表中。
for(int j = 0; j<512; j++)
{
   task[i]->pgd[j] = swapper_pg_dir[j];
}
//将 uapp 所在的页面映射到每个进行的页表中
create_mapping(task[i]->pgd, USER_START, (uint64)_sramdisk-PA2VA_OFFSET, (uint64)_eramdisk-(uint64)_sramdisk, 0b11111);
//申请用户态栈,放在user space 的最后一个页面
uint64 u_mode_stack = alloc_page();
create_mapping(task[i]->pgd, USER_END-PGSIZE, u_mode_stack - PA2VA_OFFSET, PGSIZE, 0b11111);
task[i]->pgd=task[i]->pgd-PA2VA_OFFSET;
```

修改_switch_to:

保存/恢复 sepc sstatus sscratch：

```
#sepc
csrr t1, sepc
sd t1, 160(a0)
#sstatus
csrr t1, sstatus
sd t1, 168(a0)
#sscratch
csrr t1, sscratch
sd t1, 176(a0)
```

```
#sepc
ld t1, 160(a1)
csrw sepc, t1
#sstatus
ld t1, 168(a1)
csrw sstatus,t1
#sscratch
ld t1, 176(a1)
csrw sscratch,t1
```

切换页表：

```
//切换页表,修改SATP的PPN为用户态页面的PPN
ld t1, 184(a1)
srli t1, t1, 12 // PPN
li t2, 8 // MODE = 8
slli t2, t2, 60
or t1, t1, t2
csrw satp, t1
```

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

修改__dummy：

切换模式时交换 sp 的值。

```
__dummy:
    # YOUR CODE HERE
    # la t0, dummy
    # csrw sepc, t0
    # sret
    #修改 __dummy。在 4.2 中 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。
    la t0,0
    csrw sepc,t0
    csrr t0, sscratch
    csrw sscratch, sp
    add sp, t0, zero
    sret
```

修改_trap:

U-mode 和 S-mode 中断时行为不同。前者只需交换 thread_struct.sp 和 thread_struct.sscratch，进入 S-mode；后者进入 S-trap 进行后续处理。

```
_traps:
    csrr t0,sstatus
    andi t0, t0, 0x100
    // sstatus的spp不为0时为S 模式，进入S_trap
    bne t0, zero,_S_traps
    //用户模式trap只需交换thread_struct.sp和thread_struct.sscratch
    csrr t1, sscratch
    csrw sscratch, sp
    addi sp, t1, 0
```

原来的_trap 改为_S_trap,加入对模式的判断：

```
    # -----------
    // 如果是U-mode，就交换thread_struct.sp和thread_struct.sscratch。
    csrr t0, sstatus
    andi t0, t0, 0x100
    bne t0, zero, _return
    csrr t1, sscratch
    csrw sscratch, sp
    addi sp, t1, 0
    # -----------
_return:
    # 4. return from trap
    sret
    # -----------
```

修改 trap_handler 捕获 ECALL_FROM_U_MODE exception。

```
    //ECALL_FROM_U_MODE exception
    else
    {
        if(scause==0x8)
        {
            syscall(regs);
        }
    }
}
```

4. **添加系统调用**

判断是调用 sys_write 还是 sys_getpid。还要手动修改 sepc 的地址，使得 sret 之后程序继续执行。

```
void syscall(struct pt_regs* regs)
{
    //64 号系统调用sys_write
    if(regs->a7==SYS_WRITE)
    {
        sys_write(regs->a0, (const char*)regs->a1, regs->a2);
        regs->sepc+=4;
    }
    //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
    else if(regs->a7==SYS_GETPID)
    {
        regs->a0=sys_getpid();
        regs->sepc+=4;
    }
}
```

fd 为 1 时输出字符。

```
void sys_write(unsigned int fd, const char* buf, size_t count)
{
    for (int i = 0; i < count; i++)
    {
        if(fd==1)
        {
            printk("%c", buf[i]);
        }
    }
}

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

5. **修改 head.S 以及 start_kernel**

修改 start_kernel：

将 schedule()在 test 之前调用。

```
schedule();

test(); // DO NOT DELETE !!!
```

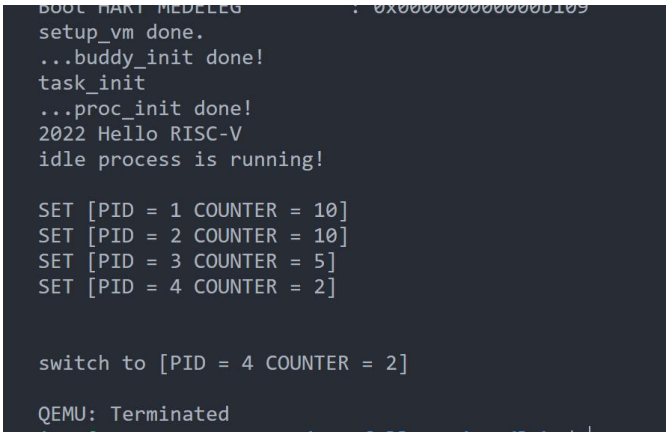
修改 head.S：

注释掉 head.S 中的 enable interrupt sstatus.SIE。

```
# set sstatus[SIE] = 1
# csrr t0, sstatus
# #sie是第2位
# ori t0, t0, 0x2
# csrw sstatus, t0
```

6. **测试**

没能成功进入用户模式，应该是 task_init 或__switch_to 的实现出了问题，但是没能找到。



四、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）

   应该是一对一。

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

   p_filesz 相较于 p_memsz 少了.bss 部分。

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？

   虚拟地址虽然相同，但是最终映射的物理地址不一样，所以并不会产生冲突。

   我原本觉得只有内核才能获取物理地址，但在网上查询后得知，linux 在/proc/pid 目录下有一个名为 pagemap 的文件，记录着进程的物理地址，而且是允许用户态进程查看的。用户可以通过这一文件查看自己栈的物理地址。

五、讨论心得

这次实验我没有完全实现成功，无法正常进入用户态，分析下来应该是 task_init 函数中没能正确分配页面空间或是__switch__to 中没能正确实现进程模式切换逻辑的缘故，但最终还是没能在验收之前找到问题所在，后面的 ELF 支持部分也就没能完成。整个实验前后也是花了数个星期，已经尽力去完成了，希望助教能够手下留情。。