操作系统lab4：RV64 用户态程序

教师：李环、柳晴
学号：3220104519
姓名：蔡佳伟

一、实验目的

- 创建用户态进程，并完成内核态与用户态的转换
- 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
- 实现用户态 ELF 程序的解析和加载

二、实验过程

2.1 准备工程

修改`vmlinux.lds`，将用户态程序uapp加载至`.data`段：

```
53
54      .data : ALIGN(0x1000) {
55          _sdata = .;
56
57          *(.sdata .sdata*)
58          *(.data .data.*)
59
60          _edata = .;
61
62          . = ALIGN(0x1000);
63          _sramdisk = .;
64          *(.uapp .uapp*)
65          _eramdisk = .;
66          . = ALIGN(0x1000);
67      } >ramv AT>ram
68
```

修改`defs.h`，在`defs.h`添加如下内容：

```
#define USER_START (0x0000000000000000) // user space star
#define USER_END (0x0000004000000000) // user space end vi
```

修改最外层的`MAKEFILE`并同步文件夹

```
.PHONY:all run debug clean
all: clean
    $(MAKE) -C lib all
    $(MAKE) -C init all
    $(MAKE) -C user all
    $(MAKE) -C arch/riscv all
    @echo -e '\n'Build Finished OK

run: all
    @echo Launch qemu...
    @qemu-system-riscv64 -nographic -machine virt -kernel v

debug: all
    @echo Launch qemu for debug...
    @qemu-system-riscv64 -nographic -machine virt -kernel v

clean:
    $(MAKE) -C lib clean
    $(MAKE) -C init clean
    $(MAKE) -C user clean
    $(MAKE) -C arch/riscv clean
    $(shell test -f vmlinux && rm vmlinux)
```

2.2 创建用户态进程

修改`proc.h`中的结构体

```
/* 线程状态段数据结构 */
struct thread_struct
{
    uint64_t ra;    // 32
    uint64_t sp;    // 40
    uint64_t s[12]; // 48
    uint64_t sepc, sstatus, sscratch;// 144 152 160
};

/* 线程数据结构 */
struct task_struct
{
    uint64_t state;    // 线程状态 0
    uint64_t counter;  // 运行剩余时间 8
    uint64_t priority; // 运行优先级 1最低 10最高 16
    uint64_t pid;      // 线程id 24

    struct thread_struct thread;
    uint64_t *pgd; // 168
};
```

修改`proc.c`中`task_init()`，

对于每个进程，初始化我们刚刚在 thread_struct 中添加的三个变量，具体而言：

- 将 sepc 设置为 USER_START
- 配置 sstatus 中的 SPP（使得 sret 返回至 U-Mode）、SUM（S-Mode 可以访问 User 页面）

- 将 sscratch 设置为 U-Mode 的 sp，其值为 USER_END （将用户态栈放置在 user space 的最后一个页面）

```
97  void task_init()
99      srand(2024);
100
101     // 1. 调用 kalloc() 为 idle 分配一个物理页
102     // 2. 设置 state 为 TASK_RUNNING;
103     // 3. 由于 idle 不参与调度，可以将其 counter / priority 设置为 0
104     // 4. 设置 idle 的 pid 为 0
105     // 5. 将 current 和 task[0] 指向 idle
106     idle = (struct task_struct *)kalloc();
107     if (idle == NULL)
108         printk(RED "Error : kalloc() failed in task_init()\n" CLEAR);
109
110     idle->state = TASK_RUNNING;
111     idle->counter = 0;
112     idle->priority = 0;
113     idle->pid = 0;
114     uint64_t sstatus = 0;
115     sstatus |= (1UL << 5);                          // SPIE: 开启中断
116     sstatus |= (1UL << 18);                         // SUM: 允许访问用户内存
117     idle->thread.sstatus = sstatus;
118     current = idle;
119     task[0] = idle;
120
```

```
    for (int i = 1; i < NR_TASKS; i++)
    {
        task[i] = (struct task_struct *)kalloc();
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = PRIORITY_MIN + rand() % (PRIORITY_MAX - PRIORITY_MIN +
        task[i]->pid = i;
        task[i]->thread.ra = (uint64_t)__dummy;
        task[i]->thread.sp = (uint64_t)task[i] + PGSIZE;
        task[i]->thread.sepc = USER_START;
        uint64_t sstatus = 0b0;
        sstatus |= (1UL << 5);                          // SPIE: 开启中断
        sstatus |= (1UL << 18);                         // SUM: 允许访问用户内存
        task[i]->thread.sstatus = sstatus;
        task[i]->thread.sscratch = USER_END;
        task[i]->pgd = (int64_t*)alloc_page();
        // //pgd_set_to_zero(task[i]->pgd);

        map_kernel_pgd(task[i]->pgd);
        map_user_pages(task[i]->pgd);
        map_stack_of(task[i]->pgd);

        //load_program(task[i]);
        printk("task[%d] pid = %d priority = %d\n", i, task[i]->pid, task[i]->prio
    }

    printk("...task_init done!\n");
```

对于每个进程，创建属于它自己的页表：

- 为了避免 U-Mode 和 S-Mode 切换的时候切换页表，我们将内核页表 swapper_pg_dir 复制到每个进程的页表中
- 对于每个进程，分配一块新的内存地址，将 uapp 二进制文件内容拷贝过去，之后再将其所在的页面映射到对应进程的页表中

设置用户态栈，对每个用户态进程，其拥有两个栈

- 用户态栈：我们可以申请一个空的页面来作为用户态栈，并映射到进程的页表中
- 内核态栈；在 lab3 中已经设置好了，就是 thread.sp

```
void map_kernel_pgd(uint64_t *pgd) {
    // 假设 pgd 是指向进程的页全局目录的指针
    // 将内核空间的虚拟地址映射到该进程的页表中
    for (int i = VA_VPN2(VM_START); i < VA_VPN2(VM_END); ++i) {
        // 设置内核空间的页目录项
        printk("%d ",i);
        pgd[i] = swapper_pg_dir[i]; // 使用共享的内核页表条目
    }

}
```

```
void map_user_pages(uint64_t *pgd) {
    // 假设用户程序的虚拟地址是从 USER_START 开始的
    // printk("%lx",pgd);
    // int64_t* temp = alloc_pages(((int64_t)_eramdisk - (int64_t)_sramdisk)/PGSIZE
    // for(int i=0;i<((int64_t)_eramdisk - (int64_t)_sramdisk)/PGSIZE + 1; ++i){
    //     temp[i] = _sramdisk[i];
    // }
    // create_mapping(pgd, USER_START, (int64_t)temp - PA2VA_OFFSET, _eramdisk - _s
    uint64_t uapp_size = _eramdisk - _sramdisk;
    char *uapp = (char *)alloc_pages((uapp_size + PGSIZE - 1) / PGSIZE);
    // memcpy(uapp, _sramdisk, uapp_size);
    for (int j = 0; j < uapp_size; ++j) {
        uapp[j] = _sramdisk[j];
    }
    create_mapping(pgd, USER_START, (uint64_t)uapp - PA2VA_OFFSET, uapp_size, 0x1F)
}
```

```
void map_stack_of(uint64_t *pgd) {

    // check_page_table(0xffffffe000201154, (uint64_t)pgd);+
    int64_t user_stack = (int64_t)alloc_page();
    create_mapping(pgd, USER_END - PGSIZE, user_stack - PA2VA_OFFSET, PGSIZE, 0x17)
}
```

2.3 修改__switch_to

在前面新增了`sepc`、`sstatus`、`sscratch`之后，需要将这些变量在切换进程时保存在栈上，因此需要更新`__switch_to`中的逻辑，同时需要增加切换页表的逻辑。在切换了页表之后，需要通过`sfence.vma`来刷新`TLB`和`ICache`。

```
__switch_to:
    # save state to prev process a0 保存当前线程的 ra，sp，s0~s11
    # task_struct 存放在该页的低地址部分，将线程的栈指针sp指向该页的高地址
    sd ra, 32(a0)
    sd sp, 40(a0)
    sd s0, 48(a0)
    sd s1, 56(a0)
    sd s2, 64(a0)
    sd s3, 72(a0)
    sd s4, 80(a0)
    sd s5, 88(a0)
    sd s6, 96(a0)
```

```
    sd s7, 104(a0)
    sd s8, 112(a0)
    sd s9, 120(a0)
    sd s10, 128(a0)
    sd s11, 136(a0)

    # 读取sepc并存储到a0+144
    csrr t1, sepc
    sd t1, 144(a0)

    # 读取sstatus并存储到a0+152
    csrr t1, sstatus
    sd t1, 152(a0)

    # 读取sscratch并存储到a0+160
    csrr t1, sscratch
    sd t1, 160(a0)

    # 对satp进行按位与操作后存储到a0+168
    csrr t1, satp           # 读取satp
    li t2, 0x0fffffffffff   # 加载48位掩码
    and t0, t1, t2          # 按位与
    slli t0, t0, 12
    li t1, 0xffffffdf80000000 # 进行偏移
    add t0, t0, t1
    sd t0, 168(a0)          # 存储t0到内存地址a0+168


    # restore state from next process
    ld ra, 32(a1)
    ld sp, 40(a1)
    ld s0, 48(a1)
    ld s1, 56(a1)
    ld s2, 64(a1)
    ld s3, 72(a1)
    ld s4, 80(a1)
    ld s5, 88(a1)
    ld s6, 96(a1)
    ld s7, 104(a1)
    ld s8, 112(a1)
    ld s9, 120(a1)
    ld s10, 128(a1)
    ld s11, 136(a1)

    # 从内存中加载sepc的值并写入到 CSR
    ld t1, 144(a1)          # 从a1+144加载数据到t1
    csrw sepc, t1           # 将t1的值写入sepc

    # 从内存中加载 sstatus 的值并写入到 CSR
    ld t1, 152(a1)          # 从a1+152加载数据到t1
    csrw sstatus, t1        # 将t1的值写入sstatus

    # 从内存中加载 sscratch 的值并写入到 CSR
    ld t1, 160(a1)          # 从a1+160加载数据到t1
    csrw sscratch, t1       # 将t1的值写入sscratch
```

```
    # 从内存中加载satp的值，进行掩码操作后写入到CSR
    ld t0, 168(a1)           # 从a1+168加载数据到t0
    li t1, 0x0fffffdf80000000
    sub t0, t0, t1
    srli t0, t0, 12
    li t1, 0x08000000000000000 # 将掩码加载到t1
    or t0, t0, t1           # 对t0和t1按位与

    csrw satp, t0            # 将t0的值写入satp

    sfence.vma               # 刷新虚拟内存的地址转换缓存
    ret
```

2.4 更新中断处理逻辑

修改`__dummy`

在我们初始化线程时，`thread_struct.sp`保存了内核态栈`sp`，`thread_struct.sscratch`保存了用户态栈`sp`，因此在`__dummy`进入用户态模式的时候，我们需要切换这两个栈，只需要交换对应的寄存器的值即可。

```
__dummy:
    la t0, dummy
    #csrw sepc, t0
    #mv t1, sp
    #csrr sp, sscratch
    #csrw sscratch, t1
    csrrw sp, sscratch, sp
    sret
```

修改`_traps`

同理，在`_traps`的首尾我们都需要做类似的操作，进入trap的时候需要切换到内核栈，处理完成后需要再切换回来。

注意如果是内核线程（没有用户栈）触发了异常，则不需要进行切换。（内核线程的`sp`永远指向的内核栈，且`sscratch`为 0）

```
_traps:
    la t0, current
    ld t1, 0(t0)
    ld t0, 24(t1)
    #beqz t0, noswap
    #mv t1, sp
    #csrr sp, sscratch
    #csrw sscratch, t1
    csrrw sp, sscratch, sp
noswap:
    # 1. save 32 registers and sepc to stack
    addi sp, sp, -32*8    # 64位寄存器，每个寄存器占8字节
    sd ra, 0(sp)
    sd gp, 8(sp)
```

```
    sd tp, 16(sp)
    sd t0, 24(sp)
    sd t1, 32(sp)
    sd t2, 40(sp)
    sd s0, 48(sp)
    sd s1, 56(sp)
    sd a0, 64(sp)
    sd a1, 72(sp)
    sd a2, 80(sp)
    sd a3, 88(sp)
    sd a4, 96(sp)
    sd a5, 104(sp)
    sd a6, 112(sp)
    sd a7, 120(sp)
    sd s2, 128(sp)
    sd s3, 136(sp)
    sd s4, 144(sp)
    sd s5, 152(sp)
    sd s6, 160(sp)
    sd s7, 168(sp)
    sd s8, 176(sp)
    sd s9, 184(sp)
    sd s10, 192(sp)
    sd s11, 200(sp)
    sd t3, 208(sp)
    sd t4, 216(sp)
    sd t5, 224(sp)
    sd t6, 232(sp)
    csrr t0, sepc
    sd t0, 240(sp)

    # 2. call trap_handler
    csrr a0, scause
    csrr a1, sepc
    mv a2, sp
    call trap_handler

    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from
stack
    ld t0, 240(sp)
    csrw sepc, t0
    ld ra, 0(sp)
    ld gp, 8(sp)
    ld tp, 16(sp)
    ld t0, 24(sp)
    ld t1, 32(sp)
    ld t2, 40(sp)
    ld s0, 48(sp)
    ld s1, 56(sp)
    ld a0, 64(sp)
    ld a1, 72(sp)
    ld a2, 80(sp)
    ld a3, 88(sp)
    ld a4, 96(sp)
    ld a5, 104(sp)
```

```
    ld a6, 112(sp)
    ld a7, 120(sp)
    ld s2, 128(sp)
    ld s3, 136(sp)
    ld s4, 144(sp)
    ld s5, 152(sp)
    ld s6, 160(sp)
    ld s7, 168(sp)
    ld s8, 176(sp)
    ld s9, 184(sp)
    ld s10, 192(sp)
    ld s11, 200(sp)
    ld t3, 208(sp)
    ld t4, 216(sp)
    ld t5, 224(sp)
    ld t6, 232(sp)
    addi sp, sp, 256

    # 4. return from trap
    # jalr zero, 0(ra)
    # mv t1, sp
    # csrr sp, sscratch
    # csrw sscratch, t1

    csrrw sp, sscratch, sp

    sret
```

进入`trap`的时候会切换到内核态，处理完成后需要切换回来。需要判断是否处于内核态，如果是，则需要切换用户态和内核态的两个栈，否则就跳过。

在调用`trap_handler`之前，需要先将栈指针压入参数中，方便访问储存在栈中的用户态程序的寄存器。

返回时，如果是从用户态切换的，也需要交换两个栈指针。

修改`trap_handler`

需要添加一个参数regs来表示用户态寄存器的值的指针。 需要添加一个接口来提供`syscall`，此时`scause`=8。

```
uint64_t syscall_id = regs->a7;
                switch (syscall_id)
                {
                    case 64: // sys_write
                    regs->a0 = sys_write(regs->a0, regs->a1, regs->a2);
                    break;
                    case 172: // sys_getpid
                    regs->a0 = sys_getpid();
                    break;
                }
            // 针对系统调用这一类异常， 我们需要手动将 sepc + 4
                regs->sepc += 4;
                break;
```

2.5 添加系统调用

本次实验要求的系统调用函数原型以及具体功能如下：

- 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 为标准输出即 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；
- 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

同学们需要：

- 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，我们需要手动完成 sepc + 4

```c
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs)
{
    // 通过 `scause` 判断 trap 类型
    uint64_t interruptCode = scause >> 63;
    uint64_t exceptionCode = scause & 0x7FFFFFFFFFFFFFFF;

    if (interruptCode == 0) // exception
    {
        printk("Exception occurred at: 0x%lx, Type: ", sepc);
        switch (exceptionCode)
        {
            case 8:
                // 系统调用参数使用a0-a5，系统调用号使用a7，系统调用的返回值会被保存到a0,a1中
                uint64_t syscall_id = regs->a7;
                switch (syscall_id)
                {
                    case 64: // sys_write
                    regs->a0 = sys_write(regs->a0, regs->a1, regs->a2);
                    break;
                    case 172: // sys_getpid
                    regs->a0 = sys_getpid();
                    break;
                }
            // 针对系统调用这一类异常，我们需要手动将sepc + 4
                regs->sepc += 4;
                break;
        }
    }
    else if (interruptCode==1) // 判断是 interrupt
    {
        // 如果是 interrupt 判断是否是 timer interrupt
        // exception 5
        if (exceptionCode==5)
        {
            // printk("Supervisor timer interrupt at: 0x%lx\n", sepc);
            // 如果是 timer interrupt 则打印输出相关信息，并通过 `clock_set_next_event()` 设置下一次时钟中断
            // printk("Interrupt occurred at: 0x%lx\n", sepc);
            clock_set_next_event();
```

```c
                do_timer();
            }
            else
            {
            }
        }
}
```

```c
#include "syscall.h"
#include "printk.h"  // 假设打印函数是 printk
#include "proc.h"

extern struct task_struct *current;

// 实现 sys_write 系统调用
long sys_write(unsigned int fd, const char* buf, size_t count) {
    if (fd != 1) {  // 判断a0需要等于1
        return -1;
    }
    for (size_t i = 0; i < count; i++) {
        printk("%c", buf[i]);  // printk打印单个字符
    }
    return count;  // 返回打印的字符数
}

// 实现 sys_getpid 系统调用
long sys_getpid() {
    return current->pid;  // 从current获取PID
}
```

2.6 调整时钟中断

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
- 即在 `start_kernel()` 中，`test()` 之前调用 `schedule()`
- 将 `head.S` 中设置 `sstatus.SIE` 的逻辑注释掉，确保 `schedule` 过程不受中断影响

init > C main.c > ...

```c
#include "printk.h"
extern void test();
// extern void _traps();
int start_kernel()
{
    printk("2024");
    printk(" ZJU Operating System\n");
    // unsigned long long val;
    // printk("sscratch before: %lx\n", val);
    // csr_write(sscratch, 0x12345678);
    // csr_read(sscratch, val);
    // printk("sscratch after: %lx\n", val);
    schedule();
    test();
    return 0;
}
```

```
    # set first time interrupt
    li a0,  0x0
    call sbi_set_timer
    # set sstatus[SIE] = 1
    # csrr t0, sstatus
    # ori t0,t0,0x2;  // 1 SIE：S 模式全局中断使能位
    # csrw sstatus, t0
```

2.7 测试纯二进制文件

```
...task_init done!
2024 ZJU Operating System
SCHEDULE(): INFO OF ALL PROCESSES
pid = 0, counter = 0, priority = 0
pid = 1, counter = 0, priority = 7
pid = 2, counter = 0, priority = 10
pid = 3, counter = 0, priority = 4
pid = 4, counter = 0, priority = 1
switch from 0 to [PID = 2 PRIORITY = 10 COUNTER = 10]
Exception occurred at: 0x18, Type: Exception occurred at: 0x103c, Ty
pe: [U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.1
Exception occurred at: 0x18, Type: Exception occurred at: 0x103c, Ty
pe: [U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.2
Exception occurred at: 0x18, Type: Exception occurred at: 0x103c, Ty
pe: [U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.3
```

2.8 添加ELF解析与加载

首先将`uapp.S`中的payload换成ELF文件

```
user > ASM uapp.S
1   .section .uapp
2
3   .incbin "uapp.bin"
```

然后使用`load_program`来导入elf文件，再设置`sepc`为程序入口

```c
static uint64_t load_program(struct task_struct* task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)_sramdisk;

    Elf64_Phdr* phdrs = (Elf64_Phdr *)(_sramdisk + ehdr->e_phoff);

    uint64_t 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 = phdrs + i;
        if (phdr->p_type == PT_LOAD) {
            // alloc space and copy content
            // do mapping
            uint64_t offset = phdr->p_vaddr & (PGSIZE-1);
            uint64_t size = phdr->p_memsz + offset;
            char* va = (char*)alloc_pages((size + PGSIZE - 1) / PGSIZE);
            // memcpy(va + offset, (char*)ehdr + phdr->p_offset, phdr->p_filesz
            for (int j = 0; j < phdr->p_filesz; ++j) {
                va[offset + j] = ((char*)ehdr)[phdr->p_offset + j];
            }
            memset(va + offset + phdr->p_filesz, 0, phdr->p_memsz - phdr->p_fil
            // create_mapping(task->pgd, phdr->p_vaddr, (uint64_t)va - PA2VA_OF
            // 1 << 4 | 1 << 0 | (phdr->p_flags & 0x4) >> 1 | (phdr->p_flags &
            // | (phdr->p_flags & 0x1) << 3);
            create_mapping(task->pgd, phdr->p_vaddr, (uint64_t)va - PA2VA_OFFSE
        }
    }

    // allocate user stack and do mapping

    uint64_t user_stack = (uint64_t)alloc_page();
    create_mapping(task->pgd, USER_END - PGSIZE, user_stack - PA2VA_OFFSET, PGS
    // pc for the user program
    task->thread.sepc = ehdr->e_entry;
    return 0;
```

三、讨论和心得

在交换寄存器的值的时候，可以使用`csrrw`指令来交换，也可以使用中间寄存器`#mv t1, sp`
`#csrr sp, sscratch`
`#csrw sscratch, t1`来交换。在这次实验中，我学到了用户态和内核态切换的时候的原理和需要做的设置有哪些，实际操作了内核态和用户态的交换。还认识到了ELF文件和其作用。

`head.S`还需要注释掉设置SIE为1的一段。这里保持 sstatus.SIE 为 0 可以在 S 态禁用中断来防止被时钟中断打断。曾经我们让大家在用户态进程初始化的时候设置 sstatus.SPIE 使得进入用户态进程后 sstatus.SIE 自动被置为 sstatus.SPIE 的值（即 1），这样在用户态进程中可以接收到时钟中断。在本次实验中，我还发现因为发生中断时，都是从用户态切换到内核态，所以可以不用判断，在`_trap`直接交换`sp`和`sscratch`两个栈的值，结束后再直接换回来。不过不知道之后的实验是否还可以这样操作。

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）

一对一。实验中，我们创建用户态线程的时候，每个线程都有单独的内核栈和用户栈。这种安排会使用户态线程执行系统调用时，不会因为其他线程的调用而被阻塞。这样的设计提高了线程的运行效率，因为每个线程都可以独立地进行调用而不会相互干扰。

2.系统调用返回为什么不能直接修改寄存器?

系统调用返回时，我们要将返回值返回用户态程序，而在进行系统调用的时候，寄存器里的是内核态的寄存器状态，并且在`trap_handler`结束后，用户态的寄存器状态会覆盖寄存器原有的内核态寄存器的值。所以直接修改寄存器，无法返回信息到用户态中，修改栈的值，才是将返回值写入到即将被覆盖到寄存器上的值。

3.针对系统调用，为什么要手动将 sepc + 4？

如果不执行，`sepc`将始终指向系统调用的那一条命令，从而陷入死循环。

```
ype: Exception occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Excep
tion occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Exception occur
red at: 0x18, Type: Exception occurred at: 0x18, Type: Exception occurred at: 0x
18, Type: Exception occurred at: 0x18, Type: Exception occurred at: 0x18, Type:
Exception occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Exception
occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Exception occurred a
t: 0x18, Type: Exception occurred at: 0x18, Type: Exception occurred at: 0x18, T
ype: Exception occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Excep
tion occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Exception occur
red at: 0x18, Type: Exception occurred at: 0x18, Type: Exception occurred at: 0x
18, Type: Exception occurred at: 0x18, Type: Exception occurred at: 0x18, Type:
Exception occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Exception
occurred at: 0x18, Type: Exception occurred at: 0x18, Type: Exception occurred a
```

4.为什么 Phdr 中，p_filesz 和 p_memsz 是不一样大的，它们分别表示什么?

`p_filesz`指定了segment在文件中的大小，即在磁盘实际上占用的空间大小。`p_memsz`表示程序头描述的内存区域总大小，这个字段不仅包括文件中数据，还包括了如`.bss`段（未初始数据段）需要的空间大小等。

5.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?

每个进程的虚拟地址空间是独立的，每一个进程都有自己的页表，不同进程可能会使用相同的虚拟地址，但是各自的页表会将其映射到不同的物理地址上。在实验中，用户进程的栈所在的虚拟地址都是相同的，因为我们使用了相同的方式来分配它们。 栈所在的物理地址通常来说用户是没办法知道的，页表是将虚拟地址映射到物理地址的桥梁， 但是页表是由操作系统本身管理的，用户无法访问，所以只能使用虚拟地址。