浙江大学

本科实验报告

课程名称：操作系统

姓　　名：徐文皓

学　　院：计算机科学与技术学院

系：软件工程系

专　　业：软件工程

学　　号：3210102377

指导教师：夏莹杰

2023 年　　12 月　　03 日

浙江大学操作系统实验报告

实验名称：RV64 用户态程序

电子邮件地址：____手机：____

实验地点：线上 实验日期：2023 年 12 月 03 日

一、实验目的和要求

本实验的要求是：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、实验过程

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

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

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

     _edata = .;

      . = ALIGN(0x1000);
     uapp_start = .;
     *(.uapp .uapp*)
     uapp_end = .;
     . = ALIGN(0x1000);

 } >ramv AT>ram
...
```

在 defs.h 中新增以下内容。

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

将本实验新增的文件添加到相应位置。在顶层的 Makefile 中做出如下更改，将 user 纳入工程管理。

```
# Makefile
...
all: clean
    ${MAKE} -C lib all
    ${MAKE} -C test all
    ${MAKE} -C init all
    ${MAKE} -C user all
    ${MAKE} -C arch/riscv all
    @echo -e '\n'Build Finished OK
...
clean:
    ${MAKE} -C lib clean
    ${MAKE} -C test clean
    ${MAKE} -C init clean
    ${MAKE} -C user clean
    ${MAKE} -C arch/riscv clean
    $(shell test -f vmlinux && rm vmlinux)
    $(shell test -f System.map && rm System.map)
    @echo -e '\n'Clean Finished
```

在 proc.h 中删除 thread_info 相关部分，并修改__switch_to 的偏移。

```
# arch/riscv/kernel/entry.S
    .globl __switch_to
__switch_to:
    # save state to prev process
    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)
    # 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)

    ret
```

修改 proc.h 的其他部分，向 thread_struct 中添加 sepc、sstatus 和 sscratch 变量用于稍后创建用户态进程，向 task_struct 中添加 pgd 变量用于维护进程所使用的页表，以保证多个用户态进程的相对隔离。

```
// proc.h

#define NR_TASKS  (1 + 4)

typedef unsigned long* pagetable_t;

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

    uint64 sepc, sstatus, sscratch;
};

struct task_struct {
    struct thread_info* thread_info;
    uint64 state;
    uint64 counter;
    uint64 priority;
    uint64 pid;

    struct thread_struct thread;

    pagetable_t pgd;
};
```

修改 proc.c。

```
1  //arch/riscv/kernel/proc.c
2  void task_init() {
3    ...
4    for (int i = 1; i < NR_TASKS; i++) {
5      ...
6      task[i]->thread.ra = (uint64)__dummy;
7      task[i]->thread.sp = (uint64)task[i] + PGSIZE;
8
9      task[i]->thread.sepc = USER_START;
10     task[i]->thread.sscratch = USER_END;
11     task[i]->thread.sstatus = (task[i]->thread.sstatus | (uint64)0x40020) & (~((uint64)0x100)); // set 18(SUM), 5(SPIE), clear 8(SPP)
12
13     task[i]->pgd = (pagetable_t)kalloc();
14     memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);
15
16     uint64* uapp = (uint64*)alloc_pages(((uint64)uapp_end - (uint64)uapp_start + PGSIZE - 1) / PGSIZE);
17     memcpy(uapp, uapp_start, ((uint64)uapp_end - (uint64)uapp_start));
18     create_mapping(task[i]->pgd,USER_START,(uint64)uapp-PA2VA_OFFSET,((uint64)uapp_end - (uint64)uapp_start),0x1f);
19
20     uint64 uModeStack = kalloc() + PGSIZE;
21     create_mapping(task[i]->pgd, USER_END - PGSIZE, uModeStack - PGSIZE - PA2VA_OFFSET, PGSIZE, 0x17);
22
23     printk("\nSET [pid=%d counter=%d priority=%d]\n", task[i]->pid, task[i]->counter, task[i]->priority);
24   }
25   printk("...proc_init done!\n");
26 }
```

如上图，在第 9 行至第 10 行，我们设置 sepc 为用户态程序开始的虚拟地址以便稍后从此处开始执行，设置 sscratch 为用户态程序栈顶地址，即 USER_END。

在第 11 行，我们对 sstatus 进行设置。参考手册中给出的相关内容：

The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

The SIE bit enables or disables all interrupts in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable individual interrupt sources using the sie CSR.

The SPIE bit indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1.

我们将 SUM 位(sstatus[18])置 1，使得内核在 supervisor 模式下可以访问 user 模式的页面。将 SPIE 位(sstatus[5])置 1，使得从中断中返回后 SIE 为被置为 1，从而使得中断处于开启状态。将 SPP(sstatus[8])为置 0，使得从中断中返回后回到用户态。

在第 13 行至第 14 行，我们为这一进程分配页表空间，并初始化为内核页表。过程中，我们使用了 memcpy()来拷贝内存内容，函数实现如下：

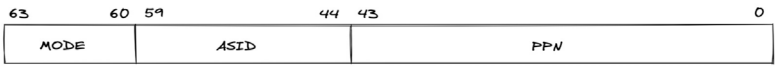
```c
//lib/string.c
void *memcpy(void * dst, void * src, uint64 size){
    char *cdst = (char *)dst;
    char *csrc = (char *)src;
    for (uint64 i = 0; i < size; i++)
        *(cdst + i) = *(csrc + i);
    return dst;
}
```

在第 16 行至第 18 行，我们将 uapp 的内容读到新分配的内存空间，建立映射并设置页表权限为 UXWRV，使得可以在用户态对这段内容进行执行。在第 20 行至第 21 行，为用户栈分配内存空间，建立映射并设置页表权限为 UWRV。至此，我们完成了对用户态进程的初始化。

接下来，我们对__switch_to 进行适当修改。考虑到前文对 task_struct 设置的新结构，在 s11 后新增的成员变量依次为 sepc、sstatus、sscratch 和 pgd，在切换线程时，与其他成员变量类似地，将前三个成员变量依序保存、切换即可。

```asm
# arch/riscv/kernel/entry.S
__switch_to:
    # save state to prev process
    ...
    sd s11, 136(a0)

    csrr t0, sepc
    sd t0, 144(a0)
    csrr t0, sstatus
    sd t0, 152(a0)
    csrr t0, sscratch
    sd t0, 160(a0)

    csrr t0, satp
    slli t0, t0, 12
    li t1, 0xffffffdf80000000 # PA2VA_OFFSET
    add t0, t0, t1
    sd t0, 168(a0)

    # restore state from next process
    ld ra, 32(a1)
    ...
    ld s11, 136(a1)

    ld t0, 144(a1)
    csrw sepc, t0
    ld t0, 152(a1)
    csrw sstatus, t0
    ld t0, 160(a1)
    csrw sscratch, t0
    ld t0, 168(a1)

    li t1, 0xffffffdf80000000 # PA2VA_OFFSET
    sub t0, t0, t1
    srli t0, t0, 12
    li t2, 0x8000000000000000 # set stap[63]
    or t2, t2, t0
    csrw satp, t2

    # flush tlb
    sfence.vma zero, zero
    # flush icache
    fence.i

    ret
```

对于 pgd，我们考虑 satp 的结构，即[43:0]为顶级页表的页号，加之 ASID 段在实验中被我们置 0，因此 satp<<12(即 PPN<<12)即为顶级页表的物理地址。先将 satp 左移 12 位取得顶级页表的物理地址，再转换为虚拟地址存入 pgd 即可对其进行保存。切换时采用相反逻辑即可。在切换后，刷新 TLB 和 ICache。



satp Register

接下来，修改 trap 入口/返回逻辑以及 trap 处理函数。

在线程初始化时，thread_struct.sp 保存了 S-Mode sp，而 thread_struct.sscratch 保存了 U-Mode sp，因此将这两个成员变量进行切换即可实现用户栈和内核栈之间的切换。

首先，修改__dummy，使得线程在第一次被调度时能够顺利完成从内核栈到用户栈的切换。

```
# arch/riscv/kernel/entry.S
__dummy:
    csrr t0, sscratch
    csrw sscratch, sp
    add sp, t0, zero
    sret
```

修改_traps，在进入 trap 时将 sp 从用户栈切换至内核栈，在退出 trap 时将 sp 从内核栈切换回用户栈。需要注意的是，对于内核线程，其 sp 永远指向 S-Mode Stack，而 sscratch 为 0，因此在内核线程中发生中断时无需切换 sp——为了实现这一逻辑，我们首先判断 sscratch 是否为 0，为 0 则跳过栈的切换。

```
# arch/riscv/kernel/entry.S
_traps:

    csrr t0, sscratch
    beq t0, zero, _traps_start
    csrw sscratch, sp
    add sp, zero, t0
_traps_start:
    # 1. save 32 registers and sepc to stack
    addi sp, sp, -8*33
    sd x0, 0(sp)
    ...
    sd x31, 248(sp)
    csrr a0, sepc
    sd a0, 256(sp)

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

    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
    ld a0, 256(sp)
    csrw sepc, a0

    ld x31, 248(sp)
    ...
    ld x2, 16(sp)
    addi sp, sp, 8*33

    # 4. return from trap
    csrr t0, sscratch
    beq t0, zero, _traps_end
    csrw sscratch, sp
    add sp, zero, t0

_traps_end:
    sret
```

在继续叙述前，我们先引入pt_regs结构体，其成员变量为寄存器状态和sepc。

```
// arch/riscv/include/trap.h
struct pt_regs {
    uint64 x[32];
    uint64 sepc;
};
```

在 trap_handler()的参数表中，我们新增 struct pt_regs *regs 这一参数，用于向稍后的系统调用传递参数。在进入 trap_handler 后，我们首先通过 scause[63]判断 trap 类型：如果是中断，判断是否为时钟中断，如果是则设置新时钟并触发时钟中断处理，否则报告错误信息及 scause 值；如果是异常，判断是否为系统调用异常，如果是则进行系统调用处理，否则报告错误信息及 scause 值——对于异常，我们将 sepc 移动到异常发生的下一条地址后再返回，避免程序进入反复触发异常的死循环状态。在后续实验中如果涉及到需要重新执行触发异常的指令，我们再对此处进行修正。

```
// arch/riscv/kernel/trap.c
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *regs)
{
    unsigned long interrupt = scause & 0x8000000000000000; // 1 - Interrupt | 0 - Exception
    unsigned long cause = scause & 0x7fffffffffffffff;
    if (interrupt){ // interrupt
        if(cause == 5){ // timer interrupt
            // printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }
        else{
            printk("[Error] Unexpected Interrupt with scause = 0x%16x\n", scause);
        }

    }
    else{ // exception
        if (cause == 8){ // syscall
            syscall(regs);
        }
        else{
            printk("[Error] Unexpected Exception with scause = 0x%16x\n", scause);
        }
        regs->sepc = regs->sepc + 4;
    }
    return;
}
```

相应地，在进入 trap 后、进入 trap 处理函数前，我们将 sp 的值置入 a2，由于在进入 trap 后将寄存器和 sepc 的值保存到了栈上，稍后也将从栈上恢复这些值，因此对栈进行操作即可实现在程序进出 trap 前后修改相关寄存器的值。

```
# arch/riscv/include/trap.h
_traps:
    ...
    csrr a0, sepc
    sd a0, 256(sp)

    csrr a0,scause
    csrr a1,sepc
    add a2, zero, sp
    call trap_handler
    ...
```

实现 syscall()，完成 64 号系统调用 sys_write(unsigned int fd, const char* buf, size_t count)和 172 号系统调用 sys_getpid()的相关逻辑。从 a7 中获取系统调用种类：当为 64 时，为 sys_write()，从 a0、a1、a2 中依次获取 fd、buf、count 的值，

当 fd 为 1 时，将 buf 中取出 count 个字符并调用 printk()进行输出，否则输出错误信息，将成功输出的字符个数置于 a0 中返回；当为 172 时，将当前进程号置于 a0 中返回。

```c
// arch/riscv/kernel/trap.c
void syscall(struct pt_regs *regs){
    uint64 a7 = regs->x[17];
    if(a7 == SYS_WRITE){
        int fd = regs->x[10];
        int i = 0;
        if(fd==1){
            char *buf = (char *)regs->x[11];
            int count = regs->x[12];
            for (i = 0; i < count; i++)
                printk("%c", buf[i]);
        }
        else{
            printk("Error SYS_WRITE call with fd = 0x%x\n", fd);
        }
        regs->x[10] = i; // return chars printed
    }
    else if(a7 == SYS_GETPID){
        regs->x[10] = current->pid;
    }
    return;
}
```

在 start_kernel()中，OS Boot 完成后立即调用 schedule()，即可不再等待 1 个时间片，而是直接调度 uapp 运行。

```c
// init/main.c
int start_kernel() {
    printk("2023");
    printk("[S-Mode] Hello RISC-V\n");

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

    return 0;
}

```

在_start 中，我们删除 csrs sstatus, 2 指令，禁止在内核态触发中断，从而确保 schedule()过程不受中断影响。

```c
# arch/riscv/kernel/head.S
_start:
    ...
    call sbi_ecall

    # csrs sstatus, 2 # set sstatus[SIE] = 1

    call start_kernel
    ...
```

至此，内核可以按预期结果运行。

```
...buddy_init done!
NR_TASKS =
SET [pid=1 counter=4 priority=37]

SET [pid=2 counter=9 priority=88]

SET [pid=3 counter=8 priority=52]

SET [pid=4 counter=5 priority=66]
...proc_init done!
2023[S-Mode] Hello RISC-V

SWITCH TO [pid=1 counter=4 priority=37]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2

SWITCH TO [pid=4 counter=5 priority=66]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2

SWITCH TO [pid=3 counter=8 priority=52]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3
```

接下来，为内核添加 ELF 支持。

首先，将 uapp.S 中的 payload 更换成我们的 ELF 文件。

```
// user/uapp.S
.section .uapp

.incbin "uapp"
```

接下来，参考 Elf64_Ehdr 和 Elf64_Phdr 中的一些成员变量。

```
Elf64_Ehdr   // 你可以将 uapp_start 强制转化为改类型的指针，
               然后把那一块内存当成此类结构体来读其中的数据，其中包括：
    e_ident  // Magic Number，你可以通过这个域来检测自己是不是真的正在读一个 Ehdr，
               值一定是 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
    e_entry  // 程序的第一条指令被存储的用户态虚拟地址
    e_phnum  // ELF 文件包含的 Segment 的数量
    e_phoff  // ELF 文件包含的 Segment 数组相对于 Ehdr 的偏移量

Elf64_Phdr   // 存储了程序各个 Segment 相关的 metadata
             // 你可以将 uapp_start + e_phoff 强制转化为此类型，就会指向第一个 Phdr，
             // uapp_start + e_phoff + 1 * sizeof(Elf64_Phdr)，则指向第二个‘
    p_filesz // Segment 在文件中占的大小
    p_memsz  // Segment 在内存中占的大小
    p_vaddr  // Segment 起始的用户态虚拟地址
    p_offset // Segment 在文件中相对于 Ehdr 的偏移量
    p_type   // Segment 的类型
    p_flags  // Segment 的权限（包括了读、写和执行）
```

实现 load_program。首先得到第一个 Segment 的起始地址以及 Segment 的数量，对于每个类型为 PT_LOAD 的 Segment，我们进行如下操作：获取虚拟地址，分配内存空间，拷贝 Segment 内容(并将[p_vaddr + p_filesz, p_vaddr + p_memsz)对应的区间置 0)，获取 Segment 的参数，建立映射。

随后，为用户栈分配空间并建立映射，设置 sepc、sstatus 和 sscratch 即可。

```
// arch/riscv/kernel/proc.c
static uint64_t load_program(struct task_struct* task) {
  Elf64_Ehdr* ehdr = (Elf64_Ehdr*)uapp_start;

  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 = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
    if (phdr->p_type == PT_LOAD) {
      uint64_t segment_va = phdr->p_vaddr;
      uint64_t page_offset = segment_va & 0xfff;
      uint64_t segment_size = phdr->p_memsz + (phdr->p_vaddr & 0xfff);
      uint64_t* segment_pa = (uint64_t*)alloc_pages((segment_size + PGSIZE - 1) / PGSIZE);

      memcpy((char*)segment_pa + page_offset, (char*)(ehdr) + phdr->p_offset, phdr->p_filesz);
      memset((char*)(segment_pa + page_offset + phdr->p_filesz), 0, phdr->p_memsz - phdr->p_filesz);

      int perm = 0x11 | ((phdr->p_flags & PF_X) << 3) | ((phdr->p_flags & PF_W) << 1) | ((phdr->p_flags & PF_R) >> 1);
      create_mapping(task->pgd, segment_va, (uint64_t)segment_pa - PA2VA_OFFSET, segment_size, perm);

      load_phdr_cnt++;
    }
  }
  // allocate user stack and do mapping
  // code...
  uint64 uModeStack = kalloc() + PGSIZE;
  create_mapping(task->pgd, USER_END - PGSIZE,
                 uModeStack - PGSIZE - PA2VA_OFFSET, PGSIZE, 0x17);
  // following code has been written for you
  // set user stack
  // pc for the user program
  task->thread.sepc = ehdr->e_entry;
  // sstatus bits set
  task->thread.sstatus =
      (task->thread.sstatus | (uint64)0x40020) &
      (~((uint64)0x100));  // set 18(SUM), 5(SPIE), clear 8(SPP)
  // user stack for user program
  task->thread.sscratch = USER_END;
}
```

相应地，修改 task_init()即可。

```c
// arch/riscv/kernel/proc.c
void task_init() {
  ...
  for (int i = 1; i < NR_TASKS; i++) {
    task[i] = (struct task_struct*)kalloc();
    task[i]->state = TASK_RUNNING;
    task[i]->pid = i;
    task[i]->counter = task_test_counter[i];
    task[i]->priority = task_test_priority[i];
    task[i]->thread.ra = (uint64)__dummy;
    task[i]->thread.sp = (uint64)task[i] + PGSIZE;

    task[i]->pgd = (pagetable_t)kalloc();
    memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);

    load_program(task[i]);

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

  printk("...proc_init done!\n");
}
```

至此，内核可以按照我们的预期运行。

```
NR_TASKS =
SET [pid=1 counter=4 priority=37]

SET [pid=2 counter=9 priority=88]

SET [pid=3 counter=8 priority=52]

SET [pid=4 counter=5 priority=66]
...proc_init done!
2023[S-Mode] Hello RISC-V

TO [pid=1 counter=4 priority=37]
] pid: 1, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2

SWITCH TO [pid=4 counter=5 priority=66]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2

SWITCH TO [pid=3 counter=8 priority=52]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3

SWITCH TO [pid=2 counter=9 priority=88]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.2
```

三、讨论和心得

本次实验的理解难度其实并不高，做完后回看感觉逻辑还是清晰的，只是将实验指导转换为代码的过程会有一些“难产”。得幸于同学的大力帮助，众多困难都被我一一克服。

只是有一点困难值得记录，就是在进行指针加减的时候务必要注意数据类型，或者直接将指针强制转换为 uint64 或 char*类型再进行转换。否则，实际地址的加减量与对指针加减的量会呈现倍数关系——这为本实验的进行添置了一些障碍。

其次，设置 ELF 的部分同样提醒我们，页偏移不可置之不顾，在进行映射操作时必须考虑是否要对页偏移进行处理。

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？

是一对一的关系。

在实验中，我们为 4 个内核态线程 task[i]各自分配了一个用户态线程 uapp。每个用户态线程在发生 trap 时自然进入到它所对应的内核态线程 task[i]中进行相应的处理，用户态线程之间、内核态线程之间互相独立。每个用户态线程都独立地拥有内核页表，可以独立地调用内核线程，并不会因为其他用户态线程的阻塞而阻塞。因此，本实验中所使用的用户态线程和内核态线程是一对一的关系。

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

BSS(Block Started by Symbol)段通常是指用来存放程序中未初始化的或者初始值为 0 的全局变量的一块内存区域。BSS 段属于静态内存分配。

可加载段可能包含未初始化数据的.bss 部分。将这些数据存储在磁盘上是浪费的，它只在 ELF 文件加载到内存后才占据空间。因此，只需在 p_memsz 中包含.bss 部分所需的空间，而在 p_filesz 中只考虑需要我们存储到磁盘中的部分，在 ELF 文件加载到内存后才为.bss 段提供 p_memsz-p_filesz 大小的空间即可。

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？

不同进程的页表是独立的。这些相同的栈虚拟地址经过进程自己的页表会被映射到不同的物理地址，因而是彼此独立、互不干扰的。

在实际的 Linux 系统中，内核为用户态程序提供了/proc/pid/pagemap 文件接口，这个文件允许用户查看自己的虚拟页面被映射到哪些物理帧。而截止到 lab4，我们实验中的内核显然没有提供这样的接口，页表由内核管理而对用户隐藏，因此，在不进入内核的情况下，用户无法得知自己栈所在的物理地址。