浙江大學

课程名称：	操作系统原理与实践
题　　目：	Lab 4: RV64 用户态程序
授课教师：	申文博
助　　教：	王鹤翔、陈淦豪、许昊瑞
姓　　名：	潘潇然
学　　号：	3220106049
地　　点：	32舍367

一、实验过程与步骤

1. 准备工程，在 `defs.h` 添加以下内容，并按文档要求放置好各文件位置

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

修改根目录下的Makefile，加入 `user`

```
all: clean
    ...
    $(MAKE) -C user all
    ...
clean:
    ...
    $(MAKE) -C user clean
    ...
```

2. 创建用户态进程
   - 由于在本次实验中创建用户态进程我们需要对 `sepc` ， `sstauts` ， `sscratch` 进行设置，同时还需要加入用户态页表，因此需要在线程数据结构中加入对应内容

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

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

  struct thread_struct thread;
  uint64_t *pgd;  // 用户态页表
};
```

- 接下来，我们需要对这个新的线程数据结构进行初始化，同时对每个进程创建页表，设置用户栈等信息。

```
extern char _sramdisk[];
extern char _eramdisk[];
extern uint64_t swapper_pg_dir[];

void *memcpy(void *dst, const void *src, size_t n) {
  char *str1 = (char *)dst;
  char *str2 = (char *)src;
  for (uint64_t i = 0; i < n; ++i) *(str1++) = *(str2++);
  return dst;
}
void task_init() {
  ...

  for (int i = 1; i < NR_TASKS; ++i) {
    void *task_page = kalloc();
    if (!task_page) {
      printk("kalloc failed\n");
      return;
    }
    task[i] = (struct task_struct *)task_page;
    task[i]->pid = i;
    task[i]->state = TASK_RUNNING;
    task[i]->counter = 0;
    task[i]->priority =
        PRIORITY_MIN + rand() % (PRIORITY_MAX - PRIORITY_MIN + 1);
    task[i]->thread.ra = (uint64_t)__dummy;
    task[i]->thread.sp = (uint64_t)((unsigned long)task_page + PGSIZE);
    uint64_t sstatus = 0;
    sstatus &= ~(1 << 8);
    task[i]->thread.sstatus = sstatus | (1 << 5) | (1 << 18);
    task[i]->thread.sscratch = USER_END;
    task[i]->pgd = (uint64_t *)kalloc();
    if (!task[i]->pgd) {
      printk("kalloc failed\n");
      return;
    }
    memcpy((void *)task[i]->pgd, (void *)swapper_pg_dir, PGSIZE);
    // 将uapp所在的页面映射到每个进程的页表中
    task[i]->thread.sepc = (uint64_t)USER_START;
    void *uapp = alloc_pages(((uint64_t)_eramdisk - (uint64_t)_sramdisk +
PGSIZE - 1) / PGSIZE);
    memcpy(uapp, _sramdisk, (uint64_t)_eramdisk - (uint64_t)_sramdisk);
```

```
      create_mapping(task[i]->pgd, USER_START, (uint64_t)uapp - PA2VA_OFFSET,
         (uint64_t)(_eramdisk - _sramdisk), 0x1f);
      // 用户态栈
      void *user_stack_page = kalloc();
      if (!user_stack_page) {
        printk("kalloc failed\n");
        return;
      }
      create_mapping(task[i]->pgd, USER_END - PGSIZE,
                     (uint64_t)user_stack_page - PA2VA_OFFSET, PGSIZE, 0x17);
    }

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

- 我们首先将 `sepc` 设置为 `USER_START`，`sscratch` 设置为 `USER_END`，即用户态的栈指针地址。接下来，我们要设置 `sstatus`。其中 `SPP` 位于 `sstatus[8]`，为0表示进入S-Mode之前是用户态，因此我们需要设置为0；`SPIE` 位于 `sstatus[5]`，将其置为1，表示进入S-Mode之前终端使能开启，这样我们返回用户态之后就能正常开启中断使能了；`SUM` 位于 `sstatus[18]`，将其设为1使得在S-Mode之下可以访问用户态的页面。我们通过位运算设置上述各位。

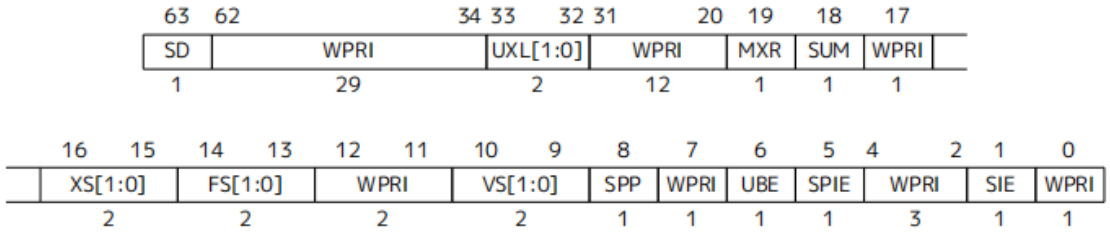

*Figure 43. Supervisor-mode status register (**sstatus**) when SXLEN=64.*

- 接下来，我们首先为每个进程创建自己的页表，我们首先将内核态页表 `swapper_pg_dir` 复制到 `task->pgd`，这里的 `memcpy` 是我们自己实现的。
- 复制之后，我们首先根据 `uapp` 的始末地址，即 `_sramdisk` 和 `_eramdisk` 计算需要的页数量。之后，我们将 `uapp` 映射到新申请的页上，并存入页表，设置正确的权限 `0x1f` (U|X|W|R|V)
- 之后我们申请用户态栈，我们新申请一页并从用户态地址末尾开始映射，并设置权限 `0x17` (U|-|W|R|V)

3. 修改 `__switch_to`。在 `__switch_to` 中，我们首先需要存储 `prev` 的各项寄存器值，包括新增加的 `sepc`，`sstatus`，`sscratch`，之后再取出来 `next` 的寄存器值。在此之后，我们通过 `next` 的栈指针将我们在 `task_init` 中设置的 `pgd` 取出，并通过和lab3中相同的操作转换为 `satp`，这里记得要减去 `PA2VA_OFFSET`。最后我们通过 `sfence.vma` 刷新TLB和ICache

```
__switch_to:
    # save state to prev process
    addi t0,a0,32
    sd ra, 0(t0)
    sd sp, 8(t0)
    sd s0, 16(t0)
    sd s1, 24(t0)
    sd s2, 32(t0)
    sd s3, 40(t0)
    sd s4, 48(t0)
    sd s5, 56(t0)
    sd s6, 64(t0)
    sd s7, 72(t0)
    sd s8, 80(t0)
    sd s9, 88(t0)
    sd s10, 96(t0)
    sd s11, 104(t0)
    csrr t1, sepc
    sd t1, 112(t0)
    csrr t1, sstatus
    sd t1, 120(t0)
    csrr t1, sscratch
    sd t1, 128(t0)

    # restore state from next process
    addi t0,a1,32
    ld ra, 0(t0)
    ld sp, 8(t0)
    ld s0, 16(t0)
    ld s1, 24(t0)
    ld s2, 32(t0)
    ld s3, 40(t0)
    ld s4, 48(t0)
    ld s5, 56(t0)
    ld s6, 64(t0)
    ld s7, 72(t0)
    ld s8, 80(t0)
    ld s9, 88(t0)
    ld s10, 96(t0)
    ld s11, 104(t0)
    ld t1, 112(t0)
    csrw sepc, t1
    ld t1, 120(t0)
    csrw sstatus, t1
    ld t1, 128(t0)
    csrw sscratch, t1
```

```
    # satp
    ld t1, 136(t0)
    li t0, 0xffffffdf80000000
    sub t1, t1, t0
    srli t1, t1, 12
    li t2, 8
    slli t2, t2, 60
    or t1, t1, t2
    csrw satp, t1

    sfence.vma zero, zero

    ret
```

4. 更新中断处理逻辑

   - 修改 `__dummy` 。我们需要通过 `__dummy` 进入用户态模式，因此我们需要在此进行用户栈和内核栈的切换。在前面设置中，我们将 `sscratch` 设置为用户栈顶，因此我们只需要交换 `sp` 和 `sscratch` 即可。之后调用 `sret` ，这时候我们就会跳转到 `sepc` 对应的地址，即用户程序开端。

     ```
         .extern dummy
         .globl __dummy
     __dummy:
         csrr t0, sscratch
         csrw sscratch, sp
         mv sp, t0
         sret
     ```

   - 修改 `_traps` 。首先注意到 `pt_regs` 中由低地址往高地址依次存储，因此我们的 `sd` 和 `ld` 也要按照此顺序，同时需要加入 `sepc` 和 `sstatus` 。此外，我们在进入 `_traps` 时，首先需要判断此trap是否是由用户态触发，如果是我们则需要切换到内核线程，并在结束后切换回用户线程。这里我们通过判断 `sscratch` 实现，因为如果来自用户线程，此寄存器值就不为0。

     ```
     _traps:
         csrr t0, sscratch
         beq t0, x0, _skip_init_traps
         csrw sscratch, sp
         mv sp, t0
     _skip_init_traps:
         # 1. save 32 registers and sepc to stack
         addi sp, sp, -272
         sd x0, 0(sp)
     ```

```
    sd x1, 8(sp)
    sd x2, 16(sp)
    sd x3, 24(sp)
    sd x4, 32(sp)
    sd x5, 40(sp)
    sd x6, 48(sp)
    sd x7, 56(sp)
    sd x8, 64(sp)
    sd x9, 72(sp)
    sd x10, 80(sp)
    sd x11, 88(sp)
    sd x12, 96(sp)
    sd x13, 104(sp)
    sd x14, 112(sp)
    sd x15, 120(sp)
    sd x16, 128(sp)
    sd x17, 136(sp)
    sd x18, 144(sp)
    sd x19, 152(sp)
    sd x20, 160(sp)
    sd x21, 168(sp)
    sd x22, 176(sp)
    sd x23, 184(sp)
    sd x24, 192(sp)
    sd x25, 200(sp)
    sd x26, 208(sp)
    sd x27, 216(sp)
    sd x28, 224(sp)
    sd x29, 232(sp)
    sd x30, 240(sp)
    sd x31, 248(sp)
    csrr t0, sepc
    sd t0, 256(sp)
    csrr t0, sstatus
    sd t0, 264(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 x0, 0(sp)
    ld x1, 8(sp)
    ld x3, 24(sp)
```

```
        ld x4, 32(sp)
        ld x5, 40(sp)
        ld x6, 48(sp)
        ld x7, 56(sp)
        ld x8, 64(sp)
        ld x9, 72(sp)
        ld x10, 80(sp)
        ld x11, 88(sp)
        ld x12, 96(sp)
        ld x13, 104(sp)
        ld x14, 112(sp)
        ld x15, 120(sp)
        ld x16, 128(sp)
        ld x17, 136(sp)
        ld x18, 144(sp)
        ld x19, 152(sp)
        ld x20, 160(sp)
        ld x21, 168(sp)
        ld x22, 176(sp)
        ld x23, 184(sp)
        ld x24, 192(sp)
        ld x25, 200(sp)
        ld x26, 208(sp)
        ld x27, 216(sp)
        ld x28, 224(sp)
        ld x29, 232(sp)
        ld x30, 240(sp)
        ld x31, 248(sp)
        ld t0, 256(sp)
        csrw sepc, t0
        ld t0, 264(sp)
        csrw sstatus, t0
        ld x2, 16(sp)
        addi sp, sp, 272

        csrr t0, sscratch
        beq t0, x0, _skip_sret_traps
        csrw sscratch, sp
        mv sp, t0
    _skip_sret_traps:
        # 4. return from trap
        sret
```

- 修改 `trap_handler` 。在 `trap_handler` 中我们需要添加一个参数 `struct pt_regs *regs` 以进行系统调用。如果 `scause` 为 `0x8` ，表示这里为 `ecall` ，那么我们就要调用我们在后面即将讲述的 `sys_call` 函数并手动将 `sepc` 加4跳出trap

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
  //   printk("[S] Supervisor Mode Trap: scause: %lx, sepc: %lx\n", scause,
  //   sepc);
  if (scause & 0x8000000000000000)       // if interrupt
    if (scause == 0x8000000000000005) {  // if timer interrupt
      //   printk("[S] Supervisor Mode Timer Interrupt\n");
      clock_set_next_event();
      do_timer();
    } else {
      printk("[S] Supervisor Mode Other Interrupt (scause: %lx, sepc: %lx).\n",
             scause, sepc);
      while (1);
    }
  else {
    if (scause == 0x8) {
      //   printk("[S] Supervisor Mode Environment Call from U-mode.\n");
      sys_call(regs);
      regs->sepc += 4;
    } else {
      printk("[S] Supervisor Mode Exception (scause: %lx, sepc: %lx).\n",
             scause, sepc);
      while (1);
    }
  }
  return;
}
```

5. 添加系统调用。我们首先捕获 `a7`，即 `eid`，根据编号跳转到对应的函数。
   - 如果是 `sys_write`，我们首先判断 `fd` 是否为1，是则继续输出，否则输出报错。我们将要输出的内容和其长度从 `a1` 和 `a2` 中读出并利用 `printk` 进行输出
   - 如果是 `sys_getpid`，我们直接将返回 `a0` 设为当前进程 `current` 的pid即可

```
extern struct task_struct *current;

void sys_call(struct pt_regs *regs) {
  //   printk("syscall %d\n", regs->s[17]);
  if (regs->s[17] == SYS_WRITE)
    sys_write(regs);
  else if (regs->s[17] == SYS_GETPID)
    sys_getpid(regs);
}

void sys_write(struct pt_regs *regs) {
  if (regs->s[10] == (uint64_t)1) {
```

```
    char *buf = (char *)regs->s[11];
    uint64_t count = regs->s[12];
    for (uint64_t i = 0; i < count; i++) printk("%c", buf[i]);
    regs->s[10] = count;
  } else
    printk("fd value error,fd = %d\n", regs->s[10]);
}

void sys_getpid(struct pt_regs *regs) {
  //   printk("pid = %d\n", current->pid);
  regs->s[10] = current->pid;
}
```

6. 调整时钟中断
   - 在 `start_kernel()` 中，在 `test()` 之前调用 `schedule()`

```
int start_kernel() {
  printk("2024");
  printk(" ZJU Operating System\n");
  schedule();
  test();
  return 0;
}
```

   - 将 `head.S` 中设置 sstatus.SIE 的逻辑注释掉
7. 测试纯二进制文件。运行 `make TEST_SCHED=1 run`，得到如下输出，成功！

```
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]

switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.2
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.3
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.4
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.5

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-MODE] pid: 1, sp is 0x3fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0x3fffffffe0, this is print No.2
```

```
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.3
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.4

switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.2
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.3

switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
[U-MODE] pid: 4, sp is 0x3ffffffe0, this is print No.1
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]

switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.6
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.7
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.8
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.9

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.5
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.6
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.7

switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.4
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.5

switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
[U-MODE] pid: 4, sp is 0x3ffffffe0, this is print No.2
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
...
```

8. 接下来，我们加入加载elf文件的部分
   - 首先修改 `uapp.S`，将payload修改为ELF文件

     ```
     .section .uapp

     .incbin "uapp"
     ```

     - 接下来在 `proc.c` 中加入 `load_program` 函数。对每个类型为 `PT_LOAD` 的Segment，我们首先获取其 `p_vaddr`，即Segment起始的用户态虚拟地址。由于此

地址不一定按页对齐，因此我们需要取出其低12位，即offset。这样就可以得到我们最终要映射的内存大小是 `p_memsz+offset`，并据此决定需要分配的页数并申请新页。之后我们将文件中从 `p_offset` 字节开始的 `p_filesz` 大小的内容复制到新页中。之后我们清零 `[p_vaddr+p_filesz,p_vaddr+p_memsz]`。之后，我们根据 `p_flags` 为线程页表设置权限，这里要注意到 `p_flags` 每一位的含义并不与SV39页表项的一致，因此需要进行一定的移位处理。之后我们建立映射。最后我们将 `sepc` 设置为 `e_entry`，即用户程序开始。

```
void load_program(struct task_struct *task) {
  Elf64_Ehdr *ehdr = (Elf64_Ehdr *)_sramdisk;
  Elf64_Phdr *phdrs = (Elf64_Phdr *)(_sramdisk + ehdr->e_phoff);
  for (int i = 0; i < ehdr->e_phnum; ++i) {
    Elf64_Phdr *phdr = phdrs + i;
    if (phdr->p_type == PT_LOAD) {
      uint64_t start_pg = phdr->p_vaddr;
      uint64_t pg_offset = phdr->p_vaddr & 0xfff;
      uint64_t size = phdr->p_memsz + pg_offset;
      void *page = alloc_pages((size + PGSIZE - 1) / PGSIZE);
      if (!page) {
        printk("alloc_pages failed\n");
        return;
      }
      memcpy((void *)(page + pg_offset), (void *)(_sramdisk + phdr->p_offset),
             phdr->p_filesz);
      memset(page + pg_offset + phdr->p_filesz, 0,
             phdr->p_memsz - phdr->p_filesz);
      uint64_t perm = 0x11 | ((phdr->p_flags & PF_X) << 3) |
                      ((phdr->p_flags & PF_W) << 1) |
                      ((phdr->p_flags & PF_R) >> 1);
      create_mapping(task->pgd, start_pg, (uint64_t)page - PA2VA_OFFSET, size,
                     perm);
    }
  }
  task->thread.sepc = (uint64_t)ehdr->e_entry;
}
```

- 此外，我们需要修改 `task_init` 函数，这里通过判断文件头是否是magic number，决定是否进入elf文件加载，否则按加载二进制文件处理。

```
Elf64_Ehdr *ehdr = (Elf64_Ehdr *)_sramdisk;
    if (*(uint64_t *)ehdr->e_ident == 0x10102464c457f) {
      load_program(task[i]);
    } else {
      task[i]->thread.sepc = (uint64_t)USER_START;
      void *uapp = alloc_pages(
          ((uint64_t)_eramdisk - (uint64_t)_sramdisk + PGSIZE - 1) / PGSIZE);
      memcpy(uapp, _sramdisk, (uint64_t)_eramdisk - (uint64_t)_sramdisk);
      create_mapping(task[i]->pgd, USER_START, (uint64_t)uapp - PA2VA_OFFSET,
                     (uint64_t)(_eramdisk - _sramdisk), 0x1f);
    }
```

- 之后运行，我们同样得到正确的结果

```
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]

switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.2
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.3
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.4
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.5

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.2
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.3
[U-MODE] pid: 1, sp is 0x3ffffffe0, this is print No.4

switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.2
[U-MODE] pid: 3, sp is 0x3ffffffe0, this is print No.3
...
```

二、实验心得

这次实验我认为整体的逻辑还算比较清晰，但在实验过程中还是遇到了一些困难。首先是我一开始的 `_traps` 存储顺序和文档中的示意图并不相同，因此我进行了修改，但在修改过程中，不小心在sd过程中过早读取了 `x2`，即 `sp` 的值，导致程序出现错误，陷入了死循环。另外，我一开始并没有存储 `x0`，因此实际上在 `syscall` 中捕获对应寄存器值时，我应该调整参数，但我一开始忘记了，就出现了错误。

此外在实验过程中，由于我需要知道在某些特定时刻的CSR寄存器，我就会在对应位置加入一行类似 `csrr t0, sepc` 的汇编指令，这样通过一些插件我就可以获取此时CSR寄存器的值，帮助我进行debug。

三、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）

- 一对一的关系。因为我们在 `task_init` 中为每一个内核态线程 `task[i]` 都载入了一个elf文件，其中包含对应的一个用户态线程。因此每个用户态线程在进入 `_traps` 的时候也都是切换到了对应的内核线程中。这些内核线程是相互独立的。因此是一对一的关系。

2. 系统调用返回为什么不能直接修改寄存器？

- 因为在进入系统调用前后，我们在 `_traps` 中通过栈在存储和恢复寄存器的值。因此这时候，我们如果在系统调用中，直接对寄存器进行修改，在退出系统调用，返回 `_traps` 的时候，对应寄存器的值就会被恢复为栈所存储的值。因此我们需要通过 `regs` 来修改栈上的内容，从而达到修改对应寄存器值的目的。

3. 针对系统调用，为什么要手动将 sepc + 4？

- 因为在系统调用后，我们希望执行系统调用的下一行指令。如果我们不手动将 `sepc` 加4，`sepc` 对应地址仍然是系统调用，会陷入死循环。因此我们加4之后才可以正常运行程序。

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` (file size)表示此Segment在ELF文件的大小，`p_memsz` (memory size)表示此Segment加载到内存后实际占用的大小
- 如果 `p_memsz` 大于 `p_filesz`，则表示该段在内存中比在文件中占用更多空间。通常，这部分额外的空间需要被初始化为0。`p_filesz` 不能大于 `p_memsz`。
- 这是因为，`.bss` 段存储未初始化的静态变量，把这些变量存在磁盘上是浪费的，因此它并不会储存在ELF文件中，只在ELF文件加载到内存后才占据空间，而同前面提到的，这部分通常设为0，仅作占位。

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？

- 因为每个进程的页表是相互独立的，相同的虚拟地址会经过页表映射到不同的物理地址，互不冲突。
- 常规方法下用户无法知道自己栈所在的物理地址，因为虚拟内存将物理内存与用户进程隔离开，用户在用户态无法知道页表的具体映射机制，没有对应的权限去walk对应的页表。这样也是处于安全考虑，避免用户访问其他进程或内核。
- 但如果具备某些权限，在Linux下可以通过访问 `/proc/pid/pagemap` 文件，在该文件中每一页生成了一个64bit的描述符，来描述虚拟地址这一页对应的物理页帧号信息，内容如下。其中我们观察到0-54位即为物理帧号，再通过其他syscall获取页大小，就可以计算得到物理地址。

  > - Bits 0-54 page frame number (PFN) if present
  > - Bits 0-4 swap type if swapped
  > - Bits 5-54 swap offset if swapped
  > - Bit 55 pte is soft-dirty (see :ref:`Documentation/admin-guide/mm/soft-dirty.rst <soft_dirty>`)
  > - Bit 56 page exclusively mapped (since 4.2)
  > - Bits 57-60 zero
  > - Bit 61 page is file-page or shared-anon (since 3.5)
  > - Bit 62 page swapped
  > - Bit 63 page present