浙江大学

本科实验报告

课程名称：操作系统

姓　　名：夏尤楷

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3210104331

指导教师：夏莹杰

2023 年 12 月 10 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：459510812@qq.com 手机：15058004449

实验地点：玉泉曹光彪西-503 实验日期：2023 年 12 月 9 日

一、实验目的和要求

1. 创建用户态进程，并设置 sstatus 来完成内核态转换至用户态。
2. 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。
3. 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

（一） 准备工程

需要修改 vmlinux.lds，将用户态程序 uapp 加载至.data 段。修改后的.data 的如下：

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

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

        _edata = .;

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

```
14.
15.     } >ramv AT>ram
```

在 defs.h 添加如下内容：

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

从 repo 同步以下文件和文件夹，并按照下面的位置来放置这些新文件：

```
.
├── arch
│   └── riscv
│       ├── Makefile
│       ├── include
│       │   ├── mm.h
│       │   └── stdint.h
│       └── kernel
│           └── mm.c
├── include
│   └── elf.h (this is copied from newlib)
└── user
    ├── Makefile
    ├── getpid.c
    ├── link.lds
    ├── printf.c
    ├── start.S
    ├── stddef.h
    ├── stdio.h
    ├── syscall.h
    └── uapp.S
```

修改根目录下的 Makefile，将 user 纳入工程管理。修改后的 Makefile 的 all 段、clean 段如下：

```
1.  all: clean
2.   ${MAKE} -C lib all
3.   ${MAKE} -C init all
4.   ${MAKE} -C user all
5.   ${MAKE} -C arch/riscv all
6.   @echo -e '\n'Build Finished OK
7.  ...
8.  clean:
9.   ${MAKE} -C lib clean
10.  ${MAKE} -C init clean
11.  ${MAKE} -C user clean
12.  ${MAKE} -C arch/riscv clean
13.  $(shell test -f vmlinux && rm vmlinux)
14.  $(shell test -f System.map && rm System.map)
15.  @echo -e '\n'Clean Finished
```

（二） 创建用户态进程

本次实验只创建 4 个用户态进程，修改 proc.h 中的 NR_TASKS。

修改后如下：

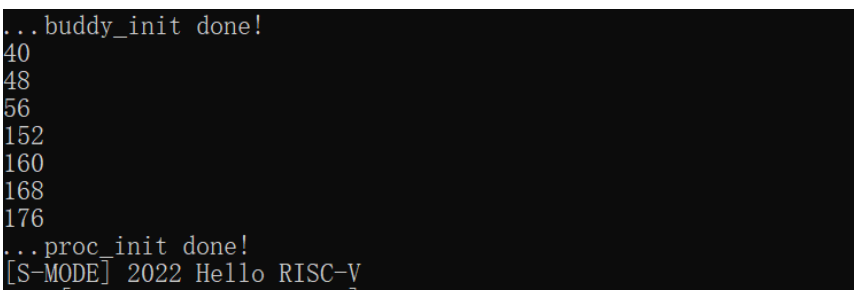
```
#define NR_TASKS  (1 + 4)
```

由于创建用户态进程要对 sepc、sstatus、sscratch 做设置，我们将其加入 thread_struct 中。由于多个用户态进程需要保证相对隔离，因此不可以共用页表。我们为每个用户态进程都创建一个页表，所以要将相应成员加入 task_struct 中。两数据结构修改后如下。

```
// proc.h

typedef unsigned long* pagetable_t;

struct thread_struct {
    uint64_t ra;
    uint64_t sp;
    uint64_t s[12];

    uint64_t sepc, sstatus, sscratch;
};

struct task_struct {
    struct thread_info* thread_info;
    uint64_t state;
    uint64_t counter;
    uint64_t priority;
    uint64_t pid;

    struct thread_struct thread;

    pagetable_t pgd;
};
```

其中 pgd 为页表的首地址。

其中，对于结构体 thread_struct thread 中的各成员和 pgd 相对结构体 task_struct 的偏移量如下图所示：（第二行开始的数字从上到下依次为 ra、sp、s、sepc、sstatus、sscratch、pgd 的偏移量）



打印这些偏移量的代码为：

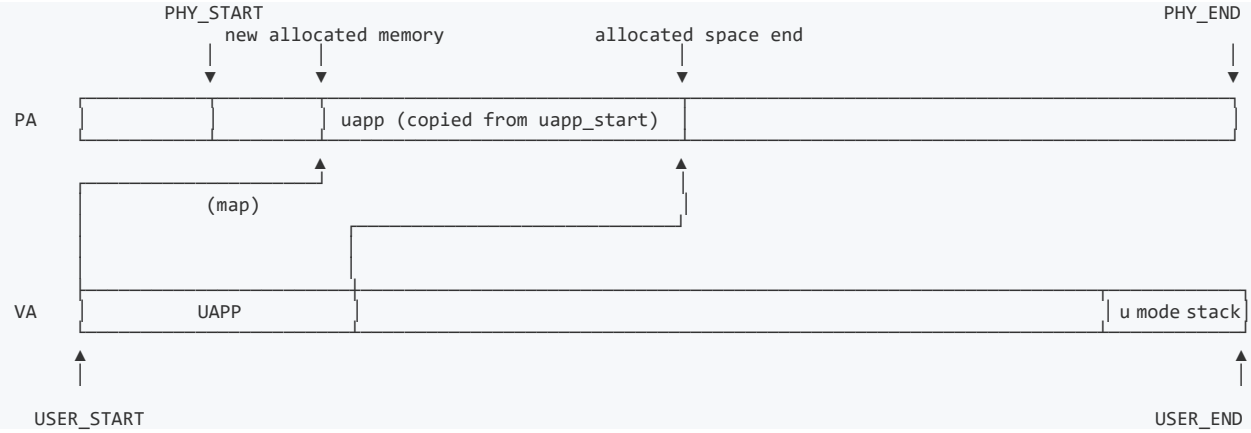
```
#define OFFSET(TYPE , MEMBER) ((unsigned long)(&(((TYPE *)0)->MEMBER)))

const uint64 OffsetOfThreadInTask = (uint64)OFFSET(struct task_struct, thread);
const uint64 OffsetOfRaInTask =
    OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, ra);
const uint64 OffsetOfSpInTask =
    OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sp);
const uint64 OffsetOfSInTask =
    OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, s);
const uint64 OffsetOfSepcInTask =
    OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sepc);
const uint64 OffsetOfSstatusInTask =
    OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sstatus);
const uint64 OffsetOfSscratchInTask =
    OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sscratch);
const uint64 OffsetOfPgdInTask = (uint64)OFFSET(struct task_struct, pgd);
printk("%d\n",OffsetOfRaInTask);
printk("%d\n",OffsetOfSpInTask);
printk("%d\n",OffsetOfSInTask);
printk("%d\n",OffsetOfSepcInTask);
printk("%d\n",OffsetOfSstatusInTask);
printk("%d\n",OffsetOfSscratchInTask);
printk("%d\n",OffsetOfPgdInTask);
```

这些代码在 task_init( )中。根据得到的偏移量，修改__switch_to 的代码。

接下去我们要修改 task_init( )函数。

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 lab2 中我们已经设置好了。我们通过 alloc_page 接口申请一个空的页面来作为 U-Mode Stack。

为每个用户态进程创建自己的页表。注意用户态程序 uapp 运行过程中，有部分数据不在栈 U-Mode Stack 上，而在初始化的过程中就已经被分配了空间（比如 counter 变量），所以用户态程序需要先被拷贝到一块某个进程专用的内存，防止所有的进程共享数据，造成期望外的进程间相互影响。如下图。



拷贝完成后，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 S-Mode 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。

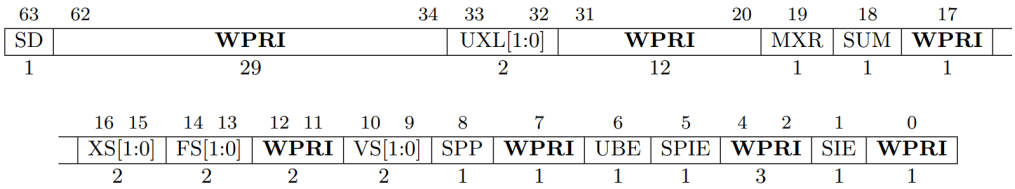


Figure 4.2: Supervisor-mode status register (`sstatus`) when SXLEN=64.

对每个用户态进程，将 sepc 修改为 USER_START。参照 riscv 手册相关内容（如上图），将 sstatus[8]（SPP）置为 0，使得 `sret` 返回至 U-Mode；将 sstatus[5]（SPIE）置为 1，使得 `sret` 之后中断开启；将 sstatus[18]（SUM）置为 1，使得 S-Mode 可以访问用户态进程页面。将 sscratch 设置为 U-Mode 的 `sp`，其值为 USER_END（即 U-Mode Stack 被放置在用户态进程空间的最后一个页面）。

修改后的 task_init( )代码如下：

```
1.  void task_init() {
2.      // 1. 调用 kalloc() 为 idle 分配一个物理页
3.      // 2. 设置 state 为 TASK_RUNNING;
4.      // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
5.      // 4. 设置 idle 的 pid 为 0
6.      // 5. 将 current 和 task[0] 指向 idle
7.      idle = (struct task_struct*)kalloc();
8.      idle->state = TASK_RUNNING;
9.      idle->counter = 0;
10.     idle->priority = 0;
11.     idle->pid = 0;
12.     current = idle;
13.     task[0] = idle;
14.
15.     for (int i = 1; i < NR_TASKS; ++i) { // 初始化其他进程
16.         task[i] = (struct task_struct*)kalloc();
17.         task[i]->pid = i;
18.         task[i]->state = TASK_RUNNING;
19.         task[i]->counter = 0;
20.         task[i]->priority = rand();
21.         task[i]->thread.ra = (uint64)&__dummy;
22.         task[i]->thread.sp = (uint64)task[i] + PGSIZE;
23.         //创建进程自己的页表并拷贝
24.         task[i]->pgd = (pagetable_t)alloc_page();
25.         for (int j = 0; j < 512; ++j)
26.             task[i]->pgd[j] = swapper_pg_dir[j];
27.
28.         task[i]->thread.sepc = USER_START;
29.         // SPP=0, SPIE=1, SUM=1
30.         task[i]->thread.sstatus = csr_read(sstatus);
31.         task[i]->thread.sstatus &= ~(1 << 8);
32.         task[i]->thread.sstatus |= (1 << 5);
33.         task[i]->thread.sstatus |= (1 << 18);
34.         task[i]->thread.sscratch = USER_END;
35.
36.         //为用户栈分配空间
37.         uint64_t U_stack_top = kalloc();
38.
39.         //拷贝二进制文件置进程专用的物理内存
```

```
        uint64_t size = PGROUNDUP((uint64_t)uapp_end - (uint64_t)uapp_start) / PGSIZE;
        uint64_t copy_addr = alloc_pages(size);
        for (int j = 0; j < uapp_end - uapp_start; ++j)
            ((char *)copy_addr)[j] = uapp_start[j];

        create_mapping(task[i]->pgd, USER_START, (uint64)copy_addr - PA2VA_OFFSET,
            size * PGSIZE, 31); // 映射用户段  U|X|W|R|V
        create_mapping(task[i]->pgd, USER_END - PGSIZE,
            U_stack_top - PA2VA_OFFSET, PGSIZE, 23); // 映射用户栈 U|-|W|R|V
    }
    // to get the offset of the members in the struct
    #define OFFSET(TYPE , MEMBER) ((unsigned long)(&(((TYPE *)0)->MEMBER)))

    const uint64 OffsetOfThreadInTask = (uint64)OFFSET(struct task_struct, thread);
            const uint64 OffsetOfRaInTask =
        OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, ra);
            const uint64 OffsetOfSpInTask =
        OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sp);
            const uint64 OffsetOfSInTask =
        OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, s);
            const uint64 OffsetOfSepcInTask =
        OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sepc);
            const uint64 OffsetOfSstatusInTask =
        OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sstatus);
            const uint64 OffsetOfSscratchInTask =
        OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sscratch);
    const uint64 OffsetOfPgdInTask = (uint64)OFFSET(struct task_struct, pgd);
    printk("%d\n",OffsetOfRaInTask);
    printk("%d\n",OffsetOfSpInTask);
    printk("%d\n",OffsetOfSInTask);
    printk("%d\n",OffsetOfSepcInTask);
    printk("%d\n",OffsetOfSstatusInTask);
    printk("%d\n",OffsetOfSscratchInTask);
    printk("%d\n",OffsetOfPgdInTask);

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

其中 create_mapping 函数即为在 lab3 的 vm.c 中定义的函数。

接下去我们修改__switch_to，需要加入保存和恢复 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。修改后代码如下：

```
    .globl __switch_to
__switch_to:
    # save state to prev process
    addi t0, a0, 40
    sd ra, 0*8(t0)
    sd sp, 1*8(t0)
    sd s0, 2*8(t0)
    sd s1, 3*8(t0)
    sd s2, 4*8(t0)
    sd s3, 5*8(t0)
    sd s4, 6*8(t0)
    sd s5, 7*8(t0)
    sd s6, 8*8(t0)
    sd s7, 9*8(t0)
    sd s8, 10*8(t0)
```

```
    sd s9, 11*8(t0)
    sd s10, 12*8(t0)
    sd s11, 13*8(t0)
    csrr t1, sepc
    sd t1, 14*8(t0)
    csrr t1, sstatus
    sd t1, 15*8(t0)
    csrr t1, sscratch
    sd t1, 16*8(t0)

    # restore state from next process
    addi t0, a1, 40
    ld ra, 0(t0)
    ld sp, 8(t0)
    ld s0, 2*8(t0)
    ld s1, 3*8(t0)
    ld s2, 4*8(t0)
    ld s3, 5*8(t0)
    ld s4, 6*8(t0)
    ld s5, 7*8(t0)
    ld s6, 8*8(t0)
    ld s7, 9*8(t0)
    ld s8, 10*8(t0)
    ld s9, 11*8(t0)
    ld s10, 12*8(t0)
    ld s11, 13*8(t0)
    ld t1, 14*8(t0)
    csrw sepc, t1
    ld t1, 15*8(t0)
    csrw sstatus, t1
    ld t1, 16*8(t0)
    csrw sscratch, t1

    # switch page table
    addi t0, zero, 1
    slli t0, t0, 63
    ld t1, 176(a1)
    li t2, PA2VA_OFFSET
    sub t1, t1, t2  #VA->PA
    srli t1, t1, 12 #get PPN
    or t0, t0, t1
    csrw satp, t0

    # flush tlb
```

```
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret
```

（三） 修改中断入口/返回逻辑（_traps）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。由于我们的用户态进程运行在 U-Mode 下，使用的运行栈也是 U-Mode Stack，因此当触发异常时，我们首先要对栈进行切换（U-Mode Stack -> S-Mode Stack）。同理，让我们完成了异常处理，从 S-Mode 返回至 U-Mode，也需要进行栈切换（S-Mode Stack -> U-Mode Stack）。

某个进程第一次被运行时，会先在 S-Mode 下进入__dummy 函数，再从这里返回 U-Mode。因此，我们要对__dummy 作一些修改。在初始化进程时，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。修改后代码如下：

```
__dummy:
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0
    sret
```

同理，在_traps 的首尾我们都需要做类似的交换寄存器的操作。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 S-Mode Stack 和 U-Mode Stack 之间的切换。由于内核线程的 sp 永远指向 S-Mode Stack，sscratch 为 0，所以我们可以根据 sscratch 的值来判断该线程是否是内核线程。同时，我们在_traps 还要保存和恢复进程的 sepc、sstatus 等寄存器的值，以供后续使用。在调用 trap_handler 之前，先将栈指针的值存入寄存器 a2，供 trap_handler 使用。

修改后的_traps 代码如下：

```
_traps:
    csrr t0, sscratch
```

```
    beqz t0, _S_mode
    csrw sscratch, sp
    mv sp, t0   # switch stack from U-mode to S-mode

_S_mode:
    addi sp,sp,-34*8
    sd x0,0*8(sp)
    sd x1,1*8(sp)
    sd x2,2*8(sp)
    sd x3,3*8(sp)
    sd x4,4*8(sp)
    sd x5,5*8(sp)
    sd x6,6*8(sp)
    sd x7,7*8(sp)
    sd x8,8*8(sp)
    sd x9,9*8(sp)
    sd x10,10*8(sp)
    sd x11,11*8(sp)
    sd x12,12*8(sp)
    sd x13,13*8(sp)
    sd x14,14*8(sp)
    sd x15,15*8(sp)
    sd x16,16*8(sp)
    sd x17,17*8(sp)
    sd x18,18*8(sp)
    sd x19,19*8(sp)
    sd x20,20*8(sp)
    sd x21,21*8(sp)
    sd x22,22*8(sp)
    sd x23,23*8(sp)
    sd x24,24*8(sp)
    sd x25,25*8(sp)
    sd x26,26*8(sp)
    sd x27,27*8(sp)
    sd x28,28*8(sp)
    sd x29,29*8(sp)
    sd x30,30*8(sp)
    sd x31,31*8(sp)
    csrr t0, sepc
    sd t0, 32*8(sp)
    csrr t0, sstatus
    sd t0, 33*8(sp)
    # save 32 registers and sepc to stack

```

```
    csrr a0, scause
    csrr a1, sepc
    mv a2, sp
    jal trap_handler
    # call trap_handler

    ld x0,0*8(sp)
    ld x1,1*8(sp)
    ld x2,2*8(sp)
    ld x3,3*8(sp)
    ld x4,4*8(sp)
    ld x5,5*8(sp)
    ld x6,6*8(sp)
    ld x7,7*8(sp)
    ld x8,8*8(sp)
    ld x9,9*8(sp)
    ld x10,10*8(sp)
    ld x11,11*8(sp)
    ld x12,12*8(sp)
    ld x13,13*8(sp)
    ld x14,14*8(sp)
    ld x15,15*8(sp)
    ld x16,16*8(sp)
    ld x17,17*8(sp)
    ld x18,18*8(sp)
    ld x19,19*8(sp)
    ld x20,20*8(sp)
    ld x21,21*8(sp)
    ld x22,22*8(sp)
    ld x23,23*8(sp)
    ld x24,24*8(sp)
    ld x25,25*8(sp)
    ld x26,26*8(sp)
    ld x27,27*8(sp)
    ld x28,28*8(sp)
    ld x29,29*8(sp)
    ld x30,30*8(sp)
    ld x31,31*8(sp)
    ld t0, 32*8(sp)
    csrw sepc, t0
    ld t0, 33*8(sp)
    csrw sstatus, t0
    addi x2,x2,34*8
```

```
    # restore sepc and 32 registers (x2(sp) should be restore last) from stack

    csrr t0, sscratch
    beqz t0, __return
    csrw sscratch, sp
    mv sp, t0   # switch stack from S-mode to U-mode

__return:
    sret
    # return from trap
```

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

uapp 使用 `ecall` 会产生 ECALL_FROM_U_MODE exception。我们在 trap_handler 里面进行捕获。查询 riscv 手册，发现该异常对应的 `scause` 寄存器内的值为 0x8（即中断位为 0，异常码为 8），如左表。

本实验中，uapp 在进行系统调用的时候使用 ecall。本次实验要求的系统调用有 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。系统调用发生时，调用号位于进程的 `a7`（`x17`）寄存器中，在 trap_handler 中读取该寄存器值，根据具体值决定进行那种系统调用。将系统调用的返回值存于进程的 `a0`（`x10`）寄存器中。

系统调用 sys_write 的三个参数分别依次从进程的 `a0`（`x10`）、`a1`（`x11`）、`a2`（`x12`）这三个寄存器中读取。

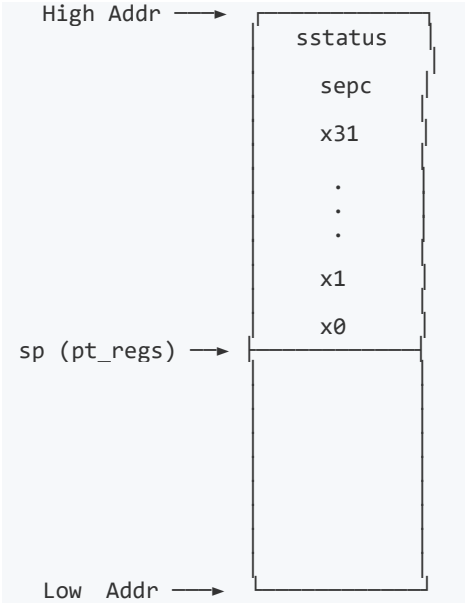
针对系统调用这一类异常，在其处理完成之后，我们应该继续执行后续的指令，而 `sepc` 记录的是触发异常的指令地址，因此我们需要手动将 `sepc` + 4。

对寄存器的读取和写入都是对栈中的存储的寄存器值的操作，而非对寄存器本身的操作。为此，我们修改 trap_handler 的声明如下：

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs);
```

其中，在`_traps` 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，而该结构体的地址就是 `sp` 寄存器内的值，我们已经在`_traps` 中通过

a2 寄存器将这个值传入 trap_handler。我们可以从这个结构体中取到相应的寄存器的值（如 syscall 中我们需要从 a0~a7 寄存器中取到参数，并将返回值写入 a0）。在本实验中，由于在_traps 中我们从低地址到高地址连续保存的寄存器依次为 x0~x31 和 sepc、sscratch，如下图：



所以我们在 trap.c 中，将结构体 pt_regs 定义如下：

```
struct pt_regs{
    uint64_t x[32];
    uint64_t sepc;
    uint64_t sstatus;
};
```

最终修改 trap_handler 如下：

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    if (scause >> 63){ // 通过 `scause` 判断trap 类型
        if (scause % 8 == 5) { // 如果是interrupt 判断是否是timer interrupt
            // 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("[S] Supervisor mode time interrupt!\n");
            clock_set_next_event();
            do_timer();
        }
    }
     // `clock_set_next_event()` 见 4.3.4 节
     else if (scause == 8) {
         uint64_t ret;
         if (regs->x[17] == SYS_WRITE) {
```

```
            ret = sys_write((unsigned int)(regs->x[10]),
                  (const char*)(regs->x[11]), (size_t)(regs->x[12]));
            regs->sepc += 4;
        }
        else if (regs->x[17] == SYS_GETPID) {
            ret = sys_getpid();
            regs->sepc += 4;
        }
        regs->x[10] = ret;
    }
}
```

（四） 添加系统调用

本次实验实现以下系统调用：

1. 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。

2. sys_getpid()：该调用从 current 中获取当前的 pid 放入 a0 中返回，无参数。

在目录 arch/riscv/include 下添加 syscall.h 文件，在目录 arch/riscv/kernel 下添加 syscall.c 文件，在其中实现 sys_write 和 sys_getpid。syscall.h 代码如下：

```
#define SYS_WRITE   64
#define SYS_GETPID  172

#include "proc.h"
#include "stddef.h"
#include "stdint.h"

extern struct task_struct* current;

uint64_t sys_write(unsigned int fd, const char* buf, size_t count);
uint64_t sys_getpid();
```

syscall.c 代码如下：

```
#include "syscall.h"
#include "printk.h"

uint64_t sys_write(unsigned int fd, const char* buf, size_t count)
```

```
{
    uint64_t length = 0;
    if (fd == 1) {  //stardard output
        for (size_t i = 0; i < count; ++i) {
            length += (uint64_t)printk("%c", buf[i]);
        }
    }

    return length;
}

uint64_t sys_getpid()
{
    return current->pid;
}
```

（五） 修改 head.S 以及 start_kernel

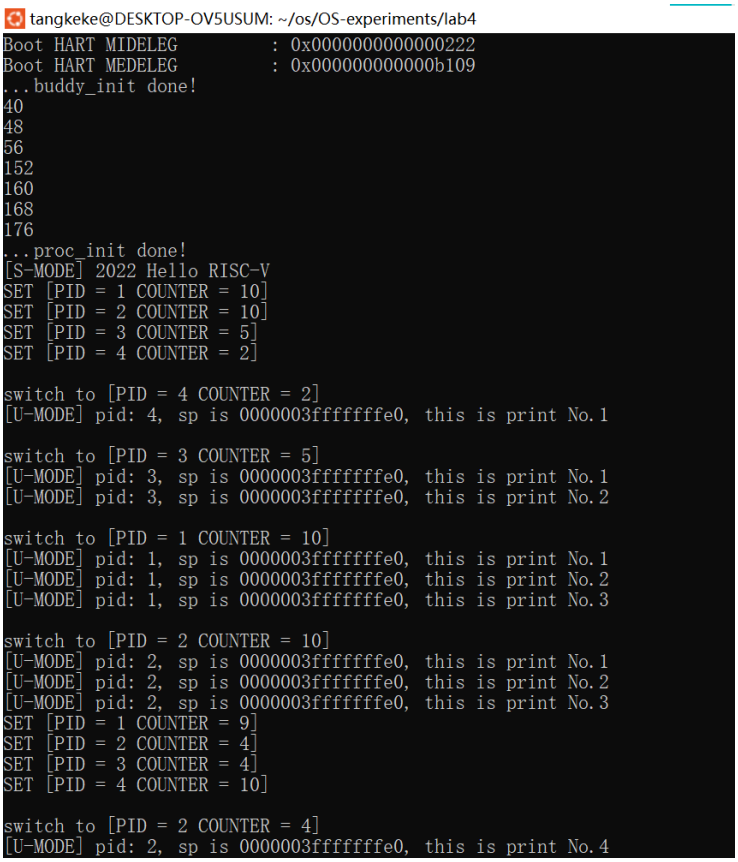
在之前的 lab 中，在 OS boot 之后，我们需要等待一个时间片，才会进行调度。我们现在更改为 OS boot 完成之后立即调度 uapp 运行。

在 init/main.c 的 start_kernel 中，在 test()之前调用 schedule()。修改后 main.c 的代码如下：

```
#include "printk.h"
#include "sbi.h"
#include "defs.h"
#include "proc.h"

extern void test();

int start_kernel() {
    printk("[S-MODE] 2022");
    printk(" Hello RISC-V\n");

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

    return 0;
}
```

将 arch/riscv/kernel/head.S 中设置 sstatus.SIE 为 1 的代码逻辑注释，确保 schedule 过程不受中断影响。

完成后编译运行程序，运行结果如下图：



程序正确运行。

（六） 添加 ELF 支持

首先我们需要将 user/uapp.S 中的 payload 给换成我们的 ELF 文件。

```
.section .uapp

.incbin "uapp"
```

然后修改 arch/riscv/kernel/proc.c，添加包含：

```
#include "elf.h"
```

然后修改 task_init()函数。定义函数 load_program 如下：

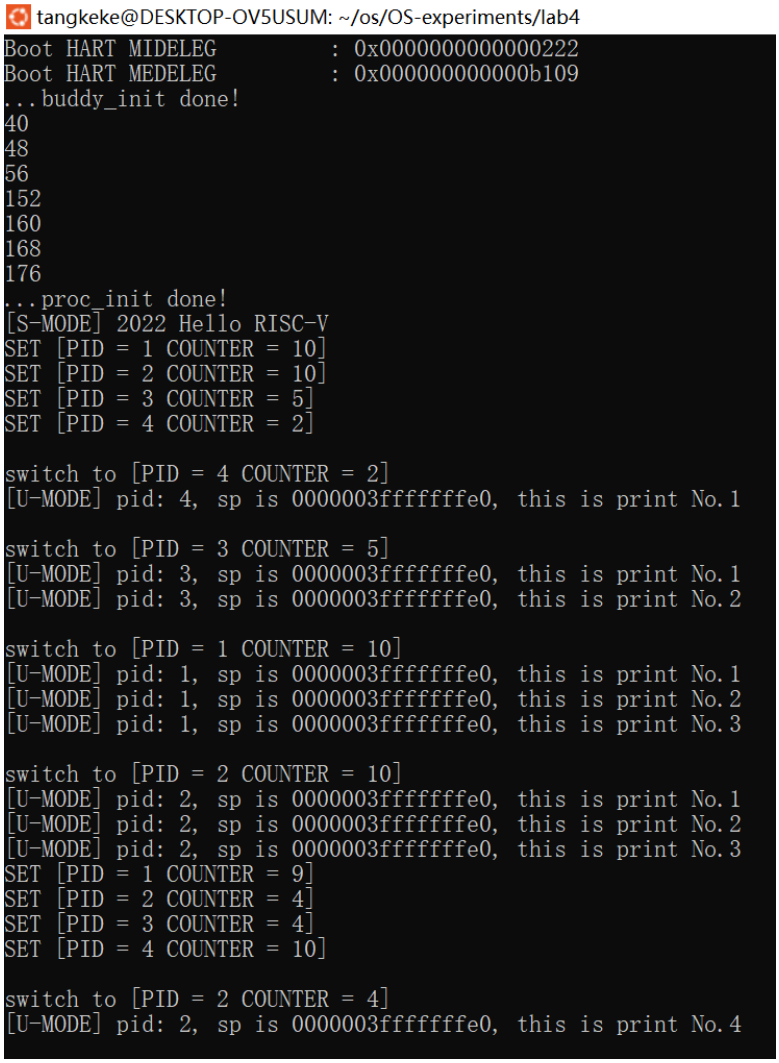
```
static uint64_t load_program(struct task_struct* task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)uapp_start;

    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) {
            // alloc space and copy content
            uint64_t offset = (uint64_t)(phdr->p_vaddr) - PGROUNDDOWN(phdr->p_vaddr);
            //align to the page
            uint64_t size = PGROUNDUP(phdr->p_memsz + offset) / PGSIZE;
            uint64_t target_addr = alloc_pages(size);
```

```
            uint64_t src_start = (uint64_t)uapp_start + phdr->p_offset;
            for (int j = 0; j < phdr->p_memsz; ++j)
                ((char*)(target_addr + offset))[j] = ((char *)src_start)[j];
            memset((void *)(target_addr + offset + phdr->p_filesz),
                  0, phdr->p_memsz - phdr->p_filesz);

            // do mapping
            uint64_t perm = 0x11;
                perm |= (1 & phdr->p_flags) << 3;         //X
                perm |= (2 & phdr->p_flags) << 1;         //W
                perm |= (4 & phdr->p_flags) >> 1;         //R
            create_mapping(task->pgd, PGROUNDDOWN(phdr->p_vaddr), target_addr - PA2VA_OFFSET,
                phdr->p_memsz + offset, perm);
         }
     }

     // allocate user stack and do mapping
     uint64 U_stack_top = alloc_page();
     create_mapping(task->pgd, USER_END - PGSIZE, U_stack_top - PA2VA_OFFSET, PGSIZE, 23);

     // following code has been written for you
     // set user stack
     // pc for the user program
     task->thread.sepc = ehdr->e_entry;
     // sstatus bits set
     // SPP=0, SPIE=1, SUM=1
     task->thread.sstatus = csr_read(sstatus);
     task->thread.sstatus &= ~(1 << 8);
     task->thread.sstatus |= 1 << 18;
     task->thread.sstatus |= 1 << 5;
     // user stack for user program
     task->thread.sscratch = USER_END;
 }
```

然后将 task_init()中初始化进程 task[i]的循环的代码修改如下：

```
    for (int i = 1; i < NR_TASKS; ++i) { // 初始化其他进程
        task[i] = (struct task_struct*)kalloc();
        task[i]->pid = i;
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand();
        task[i]->thread.ra = (uint64)&__dummy;
        task[i]->thread.sp = (uint64)task[i] + PGSIZE;
        //创建进程自己的页表并拷贝
        task[i]->pgd = (pagetable_t)alloc_page();
        for (int j = 0; j < 512; ++j)
            task[i]->pgd[j] = swapper_pg_dir[j];

        load_program(task[i]);
    }
```

修改完毕后，编译运行程序，结果如下图：



程序正确运行。

三、讨论和心得

这次实验加深了对用户态进程的理解。不过最大的感想却是在 lab3 中写的 create_mapping 函数上。在一开始我们对每个用户态进程的页表沿用 lab3 的 create_mapping 时，程序出现了匪夷所思的错误，这大大拖慢了我们的实验进度。

后来我们在不断调试中，发现了问题出在 create_mapping 函数中访问低级页表时所使用的地址上。在 lab3 的这个函数中，我们根据上一级级页表中存储的 PPN 去访问下一级页表时，是直接拿下一级页表所在的物理地址去访问下一级页表的。这种实现思路其实是错误的，因为在通过设置 satp 寄存器来开启虚拟地址后，程序会认为所有地址都是虚拟地址，因而在使用任何一个地址时，程序都会首先通过 satp 所指向的页表，把这个地址转换成物理地址，来访问相应的内存空间。只不过在 lab3 中，在使用 create_mapping 做 swapper_pg_dir 的映射时，虽然已经开启虚拟地址，但是所使用的虚拟地址页表 early_pgtbl 刚好包含虚拟地址到物理地址的等值映射，因此直接使用物理地址值访问下一级页表时，这个地址值刚好能够转换成值相等的物理地址。这就造成了直接使用物理地址也可行的假象。

但是 swapper_pg_dir 及其低级页表内并没有等值映射，值为物理地址的值的虚拟地址在页表中并没有被映射到物理地址。所以在后来给每个用户态进程的页表使用 create_mapping 时，在访问低级页表时，由于此时已开启虚拟地址，并以 swapper_pg_dir 及其低级页表为虚拟地址页表，而这些页表并没有等值映射，所以直接拿物理地址值访问低级页表时，这些地址值是无法（或者可以但错误）映射到对应的物理地址的。在意识到这一点后，错误被顺利地解决，程序也可以正常运行了。

通过这些波折，我们更深刻地理解了虚拟地址的运行原理。

四、思考题

1. 应该是一对一的。因为每个用户态线程在转到内核态时，所使用的栈都是互相独立的，这说明不同用户态线程所转到的内核态线程是不同的。
2. 在可加载段中，可能包含.bss 节，该节包含的是未初始化或是要初始化为 0 的数据，这些数据没有必要存在磁盘中，因此磁盘中就不存储这些数据，以节省空间。但显然，在内存中，需要给这些数据分配空间，这就造成了 p_memsz（段在内存中占用的空间大小）大于 p_filesz（段在磁盘中占用的空间大小）。

   至于在其他段中，有些段在磁盘中存储了数据，但是这些数据并不需要加载到内存中，这就造成了 p_filesz 非零但是 p_memsz 为零的现象。
3. 因为只要不同进程使用的用来作虚拟地址映射的页表内容不同，相同的虚拟地址就会被映射到不同的物理地址。于是，即使多个进程的栈虚拟地址相同，每个进程对栈操作时，实际读写的内存空间也是不同的，这就保证了不同进程使用的栈的互相独立。

   没有常规的方法。用户栈所在的物理地址只有在内核态才能知道。

五、附录

无。