Lab 4: RV64 用户态程序

姓名：姜雨童
学号：3220103450

1 实验内容及原理

1.1 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

1.2 实验环境

- Environment in previous labs

2 实验过程与代码实现

2.1 准备工作

- 递归且不覆盖地将 `lab3` 中内容拷贝到 `lab4`：

```
jyt@fine:~/os24fall-stu/src$ cp -Rn lab3/* lab4
```

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

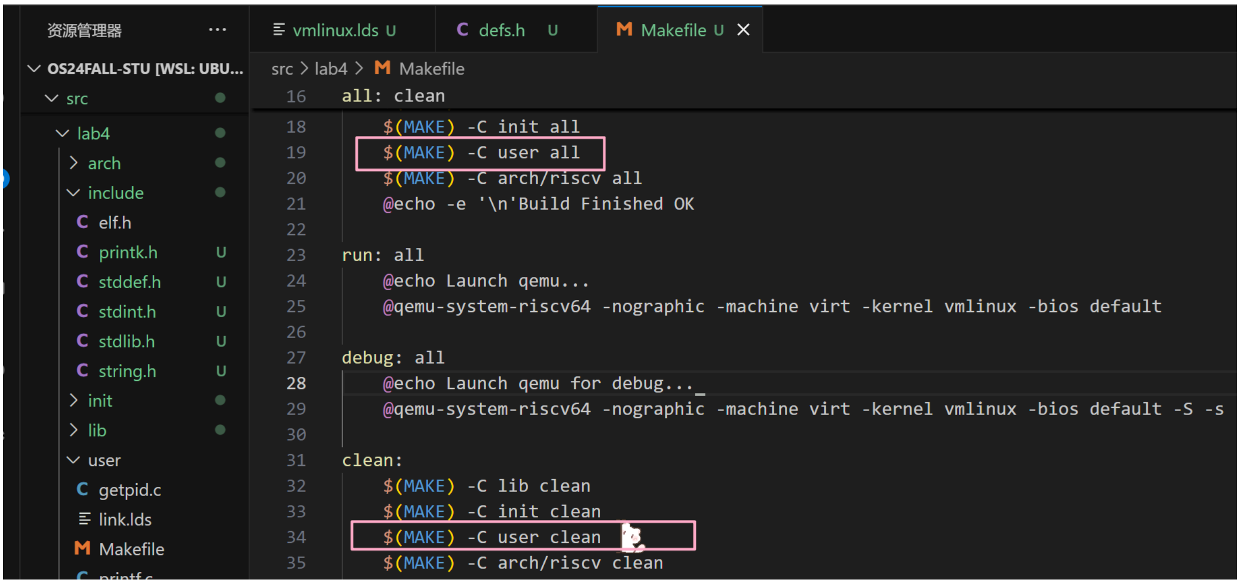
vmlinux.lds U

src > lab4 > arch > riscv > kernel > vmlinux.lds

```
{
    } >ramv AT>ram

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

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

        _edata = .;

        . = ALIGN(0x1000);
        _sramdisk = .;
        *(.uapp .uapp*)
        _eramdisk = .;
        . = ALIGN(0x1000);
    } >ramv AT>ram
```

- 修改 `defs.h`：

- 修改**根目录**下的 Makefile, 将 `user` 文件夹下的内容纳入工程管理:



2.2 创建用户态进程

2.2.1 结构体更新

修改 `proc.h` 中的 `NR_TASKS` （本实验只需要实现4个用户态进程）：

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

将 `sepc` , `sstatus` , `sscratch` 加入 `thread_struct` 中，以便创建用户态进程时对其做设置。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，需要为每个用户态进程创建一个页表并记录在 `task_struct` 中：

src > lab4 > arch > riscv > include > C proc.h > task_struct

```c
#ifndef __PROC_H__
/* 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; // lab4_add
};

/* 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 lab4_add
};
```

2.2.2 修改 `task_init()`

- 对于每个进程，初始化在 `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）

    第18位 `SUM` 置一使得 S 特权级下的程序能够访问用户页；第8位 `SPP` 置零来返回U-Mode（When an SRET instruction (see Section 3.3.2) 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.）。

63	62–34	33–32	31–20	19	18	17	
SD	WPRI	UXL[1:0]	WPRI	MXR	SUM	WPRI	
1	29	2	12	1	1	1	

16–15	14–13	12–11	10–9	8	7	6	5	4–2	1	0
XS[1:0]	FS[1:0]	WPRI	VS[1:0]	SPP	WPRI	UBE	SPIE	WPRI	SIE	WPRI
2	2	2	2	1	1	1	1	3	1	1

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

  - 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）
- 对于每个进程，创建属于它自己的页表：
  - 为避免 U-Mode 和 S-Mode 切换的时候切换页表，将内核页表 `swapper_pg_dir` 复制到每个进程的页表中
  - 对每个进程，分配新的内存地址，**拷贝** `uapp` 二进制文件内容，再将其所在的页面映射到对应进程页表中
- 设置用户态栈，对每个用户态进程，其拥有两个栈：
  - 用户态栈：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`

根据 `PTE` 的格式，设置用户态栈页面时，需要把第5位（User Mode）和第0位（有效位）置一；而拷贝 `uapp` 内容并映射页表时，需要把低4位都置一（可执行/写/读/有效位）。

63	62 61	60 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
N	PBMT	Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	7	26	9	9	2	1	1	1	1	1	1	1	1

Figure 62. Sv39 page table entry.

`arch/riscv/kernel/proc.c`:

```c
extern void __dummy();
extern char _sramdisk[], _eramdisk[];
extern uint64_t swapper_pg_dir[];
extern void create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, uint64_t perm);
...
void task_init() {
    ... // omitted
    /* YOUR CODE HERE */
    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 + 1);
        task[i]->pid = i;
        task[i]->thread.ra = (uint64_t)__dummy;
        task[i]->thread.sp = (uint64_t)task[i] + PGSIZE;
        // printk("Set [PID = %d PRIORITY = %d COUNTER = %d]\n", i, task[i]->priority, task[i]->counter); //* test

        // lab4_add begin
        task[i]->thread.sepc = USER_START;
        task[i]->thread.sstatus = 1 << 18; // sstatus[18]:SUM=1; sstatus[8]SPP=0
        task[i]->thread.sscratch = USER_END;

        task[i]->pgd = (uint64_t *)alloc_page();
        // memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);
        for (int j = 0; j < 512; j++)
            task[i]->pgd[j] = swapper_pg_dir[j];

        uint64_t ramdisk = _eramdisk - _sramdisk;
        uint64_t uapp_pg_num = (ramdisk + PGSIZE - 1) / PGSIZE;
        char *uapp_pg = (char *)alloc_pages(uapp_pg_num);
        for (int j = 0; j < ramdisk; j++)
            uapp_pg[j] = _sramdisk[j];
        create_mapping(task[i]->pgd, USER_START, (uint64_t)uapp_pg - PA2VA_OFFSET, PGSIZE * uapp_pg_num, 0x1F);

        uint64_t u_stack = (uint64_t)alloc_page(); // 用户栈在user space的最后一个页面
        create_mapping(task[i]->pgd, USER_END - PGSIZE, u_stack - PA2VA_OFFSET, PGSIZE, 0x17);
        // lab4_add end
    }
```

```c

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

2.3 修改 `__switch_to`

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（写 `satp` 的值并通过 `sfence.vma` 刷新TLB，其中 `task_struct` 上的 `pgd` 保存了下一个用户进程的页表的虚拟地址）。

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

```
__switch_to:
    # save state to prev process
    ...
    # lab4: sepc, sstatus, sscratch
    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
    ...
    # lab4: sepc, sstatus, sscratch
    ld t1, 112(t0)
    csrw sepc, t1
    ld t1, 120(t0)
    csrw sstatus, t1
    ld t1, 128(t0)
    csrw sscratch, t1

    # lab4切换页表: 写satp, 刷新TLB
    li t0, 0xffffffdf80000000 # PA2VA_OFFSET
    li t1, 0x8000000000000000 # mode sv39
    ld t2, 168(a1) # a1:struct task_struct *next, 168(a1):pgd
    sub t2, t2, t0 # pa
    srli t2, t2, 12 # PPN
    or t2, t2, t1
    csrw satp, t2

    sfence.vma zero, zero

    ret
```

2.4 更新中断处理逻辑

RISC-V 中只有一个栈指针寄存器 `sp`，在用户进程触发异常时，要用户栈切换到内核栈；完成异常处理返回用户进程时，要从内核栈切换到用户栈。

2.4.1 修改 `__dummy`

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`（Typically, sscratch is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, sscratch is **swapped with** a user register to provide an initial working register.）。因此在 `__dummy` 进入用户态模式的时候，切换内核态与用户态只需要交换对应的寄存器的值即可。

`arch/riscv/kernel/entry.S`：

```
__dummy:
    csrrw sp, sscratch, sp
    sret
```

2.4.2 修改 `_traps`

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用户栈。

但内核线程（没有用户栈）触发异常时，不需要进行切换。（其 `sp` 永远指向的内核栈，且 `sscratch` 为 0）

`arch/riscv/kernel/entry.S`：

```
_traps:
    csrr t0, sscratch
    bnez t0, 1f # 为0是内核进程，不需要切换
    csrrw sp, sscratch, sp
1:
    # 1. save 32 registers and sepc to stack
    addi sp, sp, -256
    sd x1, 0(sp)
    ... // omitted
    sd x31, 240(sp)
    csrr t0, sepc
    sd t0, 248(sp)
    # 2. call trap_handler
    csrr a0, scause
    csrr a1, sepc
    mv a2, sp // 修改了trap_handler接口，传regs
    jal trap_handler
    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
    ld t0, 248(sp)
    csrw sepc, t0
    ld x1, 0(sp)
    ... // omitted
    ld sp, 8(sp)
    addi sp, sp, 256

    csrr t0, sscratch
    beqz t0, 2f
    csrrw sp, sscratch, sp
2:
    # 4. return from trap
    sret
```

2.4.3 修改 `trap_handler`

`uapp` 使用 `ecall` 会产生 `Environment Call from U-mode`，而且处理系统调用的时候需要用到寄存器的值，因此要在 `trap_handler()` 里进行捕获（此处需要修改函数接口）。

在 `_traps` 中将寄存器的内容**连续**地保存在内核栈上，可以将这一段看做一个叫做 `pt_regs` 的结构体。在 `proc.h` 补充 `struct pt_regs` 的定义，如下：

```
struct pt_regs{
    uint64_t ra, sp, gp, tp;
    uint64_t t0, t1, t2;
    uint64_t s0, s1;
    uint64_t a0, a1, a2, a3, a4, a5, a6, a7;
    uint64_t s2, s3, s4, s5, s6, s7, s8, s9, s10, s11;
    uint64_t t3, t4, t5, t6;
    uint64_t sepc;
};
```

在 `trap.c/trap_handler` 中补充处理 `syscall` 的逻辑（同时要修改函数接口）：

查看RISC-V特权指令手册得知 `Environment Call from U-mode` 的异常码为8；查看[syscall(2)](syscall(2))手册页了解该架构下的调用说明——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。

Interrupt	Exception Code	Description
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

本次实验需要实现以下两个系统调用功能（函数在2.5节中实现）：

- 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

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    printk(PURPLE "scause = %lx\n" CLEAR, csr_read(scause)); // test*
    if (scause & 0x8000000000000000) // interrupt
    {
       ... // omitted
    } else { // exception
        switch (scause & 0x7fffffffffffffff)
        {
```

```c
            case 8: // Environment Call from U-mode
                // printk(YELLOW "[Environment Call from U-mode]\n" CLEAR);
                uint64_t syscall_ID = regs->a7;
                // printk(PURPLE "syscall_ID = %d\n"CLEAR, syscall_ID); // test*
                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;
                    default:
                        break;
                }
                regs->sepc += 4;
                break;
            default:
                // printk(RED "[Exception]\n" CLEAR); // [TODO]
                break;
        }
    }
}
```

2.5 添加系统调用

增加 `syscall.c`，`syscall.h` 文件，并在其中实现 `getpid()` 以及 `write()` 逻辑：

`arch/riscv/include/syscall.h`：

```c
#ifndef _SYSCALL_H_
#define _SYSCALL_H_

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

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

#endif  // end _SYSCALL_H_
```

`arch/riscv/kernel/syscall.c`：

```c
#include "../include/syscall.h"

extern struct task_struct *current;
uint64_t sys_write(unsigned int fd, const char* buf, size_t count)
{
    if (fd != 1 | count <= 0)
    {
        printk(RED "[Error] fd != 1 OR sys_write_count <= 0\n" CLEAR);
        return -1;
    }
    int i;
    for (i = 0; i < count; i++)
```

```c
        printk("%c", buf[i]);
    return i;
}
uint64_t sys_getpid()
{
    return current->pid;
}
```

2.6 调整时钟中断

将程序由等待一个时间片后才进行调度，更改为`OS boot`完成之后立即调度`uapp`运行：在`start_kernel()`中，`test()`之前调用`schedule()`

src > lab4 > init > main.c > start_kernel()

```c
int start_kernel() {
    // _srodata = 0xFF;

    schedule(); // lab4
    test();
    return 0;
}
```

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：

```asm
    call sbi_set_timer
    # set sstatus[SIE] = 1
    # li t1, 0x2
    # csrs sstatus, t1
    # (previous) jump to start_kernel
    jal start_kernel
```

2.7 测试纯二进制文件

项目能够正确编译运行：

```
2024 ZJU Operating System
Set [PID = 4 PRIORITY = 1 COUNTER = 1]
Set [PID = 3 PRIORITY = 4 COUNTER = 4]
Set [PID = 2 PRIORITY = 10 COUNTER = 10]
Set [PID = 1 PRIORITY = 7 COUNTER = 7]
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
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 0x3fffffffe0, this is print No.3
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[U-MODE] pid: 3, sp is 0x3fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0x3fffffffe0, this is print No.2
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
[U-MODE] pid: 4, sp is 0x3fffffffe0, this is print No.1
```

```
[U-MODE] pid: 3, sp is 0x3fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0x3fffffffe0, this is print No.2
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
[U-MODE] pid: 4, sp is 0x3fffffffe0, this is print No.1
Set [PID = 4 PRIORITY = 1 COUNTER = 1]
Set [PID = 3 PRIORITY = 4 COUNTER = 4]
Set [PID = 2 PRIORITY = 10 COUNTER = 10]
Set [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.5
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.6
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.7
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-MODE] pid: 1, sp is 0x3fffffffe0, this is print No.4
[U-MODE] pid: 1, sp is 0x3fffffffe0, this is print No.5
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
QEMU: Terminated
jyt@fine:~/os24fall-stu/src/lab4$
```

2.8 添加 ELF 解析与加载

修改 `uapp.S` 中的 payload：

```
vmlinux.lds U    proc.h U    proc.c U
user > uapp.S
.section .uapp

.incbin "uapp"
```

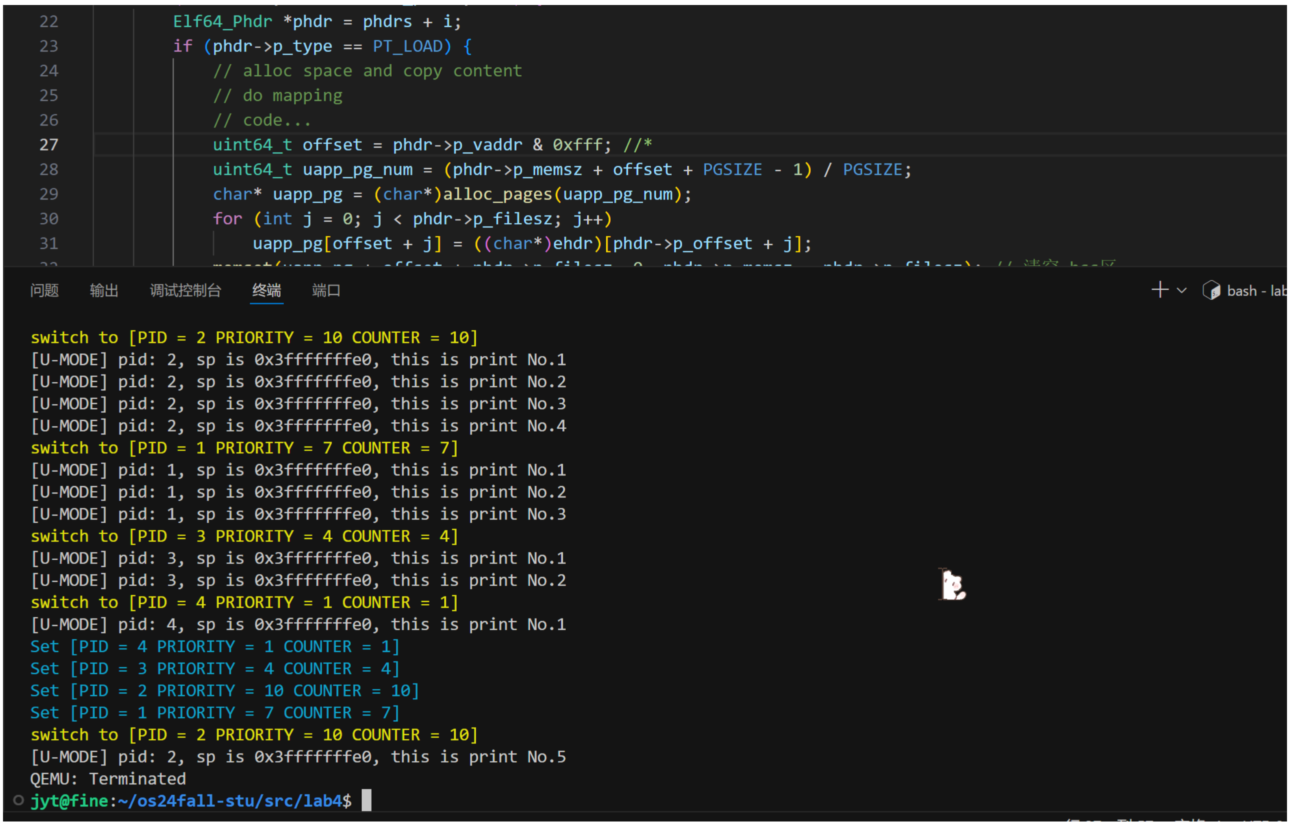
修改 `task_init` 中的初始化步骤，按照 ELF 相关的结构体定义（见 `elf.h`），从 `_sramdisk` 开始的 ELF 文件中**拷贝**内容到开辟的内存中。

这一部分和载入二进制文件时的实现方法是类似的，在 `arch/riscv/kernel/proc.c` 中编写 `load_program` 函数并在 `task_init` 中调用：

```c
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) {
            // alloc space and copy content
            // do mapping
            // code...
            uint64_t offset = phdr->p_vaddr & 0xfff; //*
            uint64_t uapp_pg_num = (phdr->p_memsz + offset + PGSIZE - 1) / PGSIZE;
            char* uapp_pg = (char*)alloc_pages(uapp_pg_num);
            for (int j = 0; j < phdr->p_filesz; j++)
                uapp_pg[offset + j] = ((char*)ehdr)[phdr->p_offset + j];
            memset(uapp_pg + offset + phdr->p_filesz, 0, phdr->p_memsz - phdr->p_filesz); // 清空.bss区

            uint64_t flag = phdr->p_flags;
            uint64_t perm = 0x17 | (flag & 0x4) >> 1 | (flag & 0x2) << 1 | (flag & 0x1) << 3; // p_flags只包含RWX
            create_mapping(task->pgd, phdr->p_paddr, (uint64_t)uapp_pg - PA2VA_OFFSET, PGSIZE * uapp_pg_num, perm);
```

```c
        }
    }

void task_init() {
    ... // omiited
    for (int i = 1; i < NR_TASKS; i++)
    {
        ... // omitted
        task[i]->pgd = (uint64_t *)alloc_page();
        // memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);
        for (int j = 0; j < 512; j++)
            task[i]->pgd[j] = swapper_pg_dir[j];

        //* load program using ELF
        load_program(task[i]);

        //* load program binary
        // uint64_t ramdisk = _eramdisk - _sramdisk;
    }
}
```

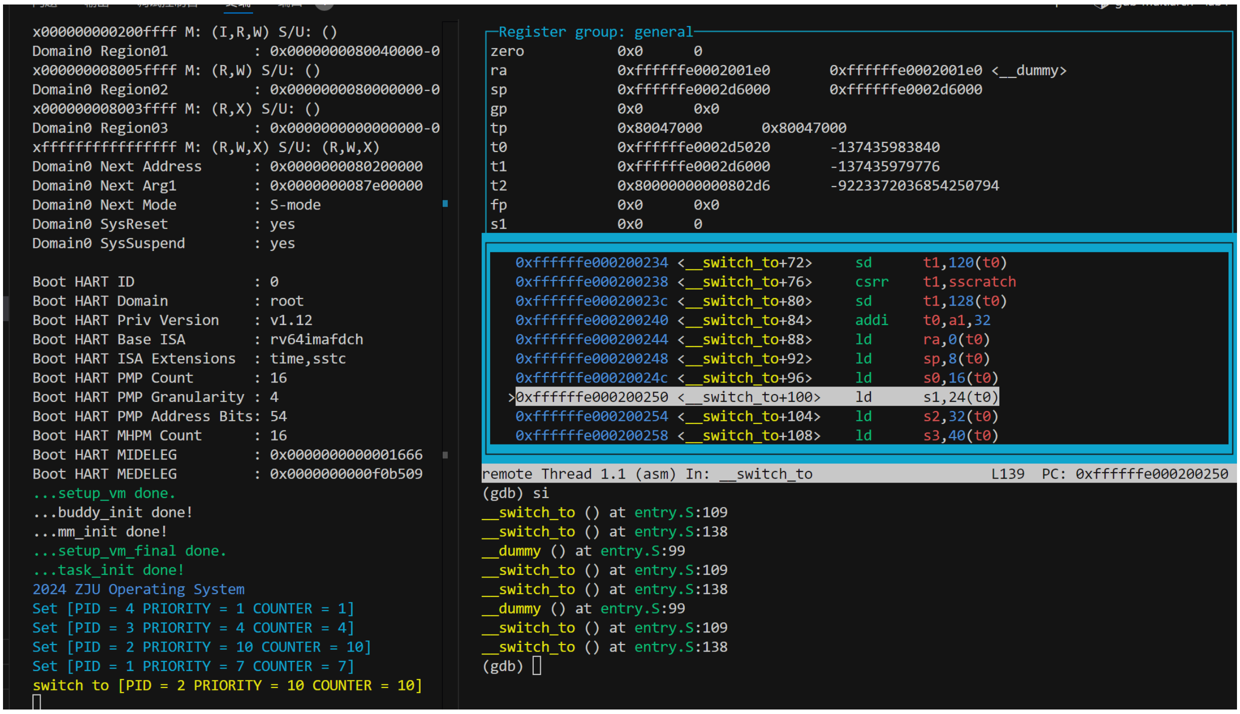
运行测试发现结果符合预期：



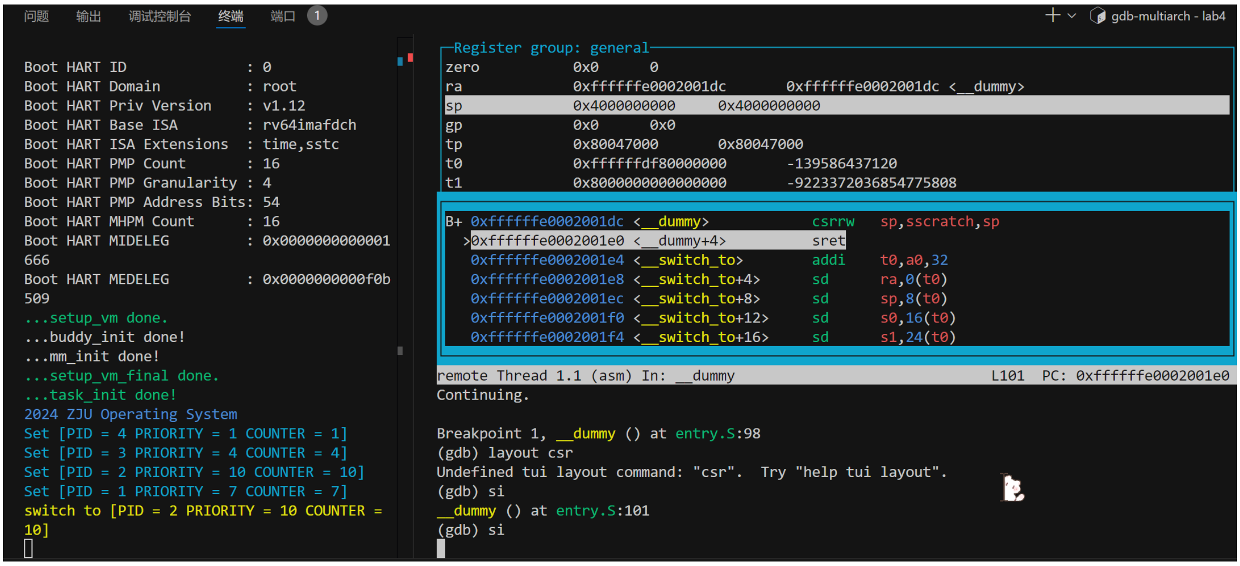
- 需要注意的是，`p_flags` 只包含了“读写执行”的权限位，而且顺序和 `PTE` 中不一样（图见2.2.2节），要进行位移操作（也不能忘记设置第5位UserMode和第0位有效位），并注意运算符的优先级。
- `PGSIZE` 为 4 KB ，所以页内偏移 `offset` 使用与每页大小减一的掩码（`0xfff`）进行与运算。

3 实验中遇到的问题及解决方法

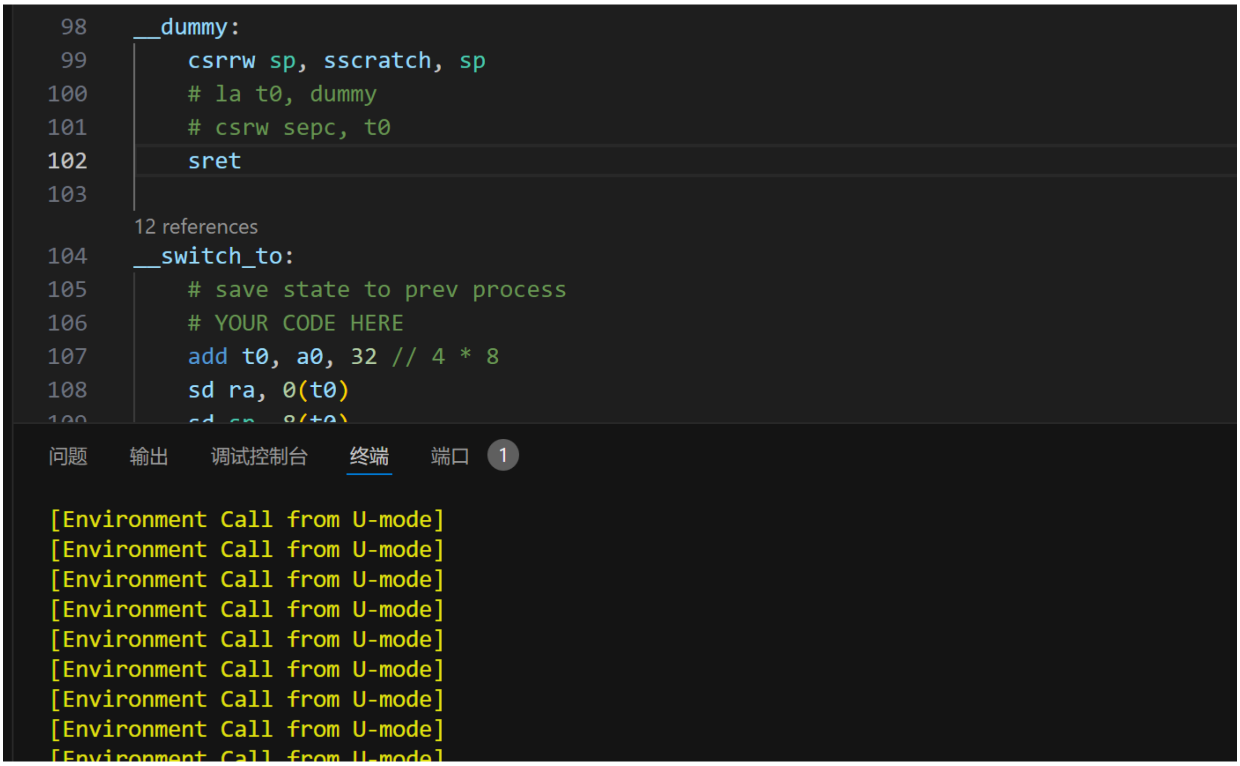
1. 测试时发现代码在`switch`后会卡住，一直无法执行下去（图3-1-1），使用gdb调试发现无法退出`__dummy`进入用户态（图3-1-2）。打印了提示信息后发现重复输出`Environment Call from U-mode`的提示并陷入死循环（图3-1-3）。



（图3-1-1）

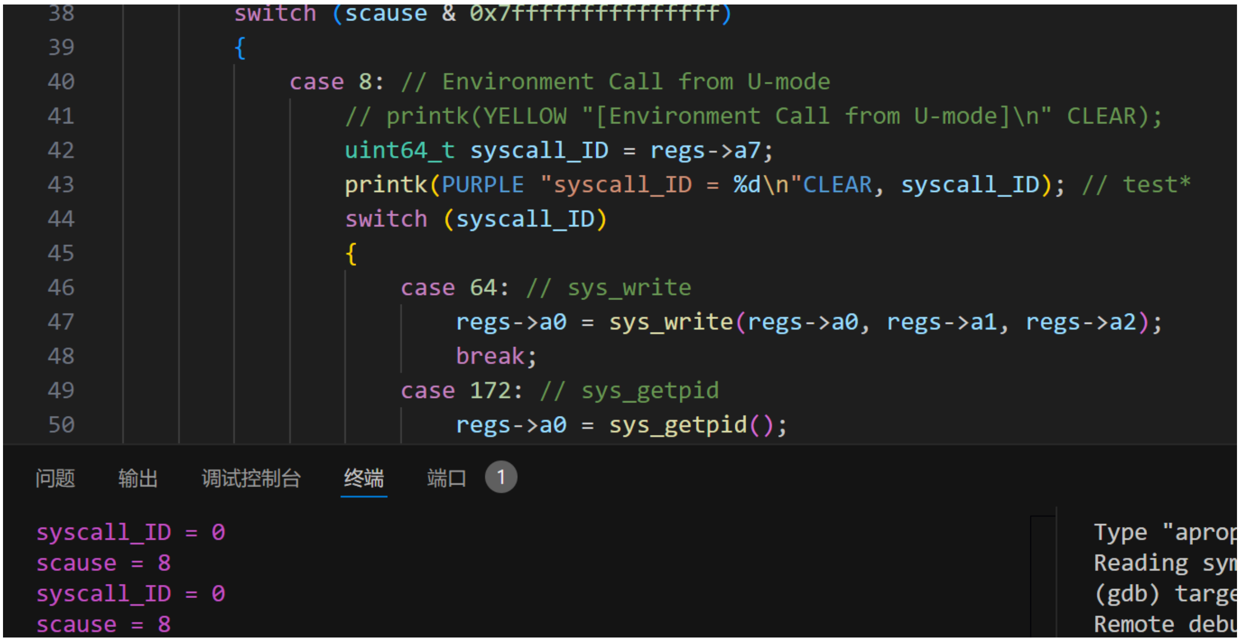


（图3-1-2）



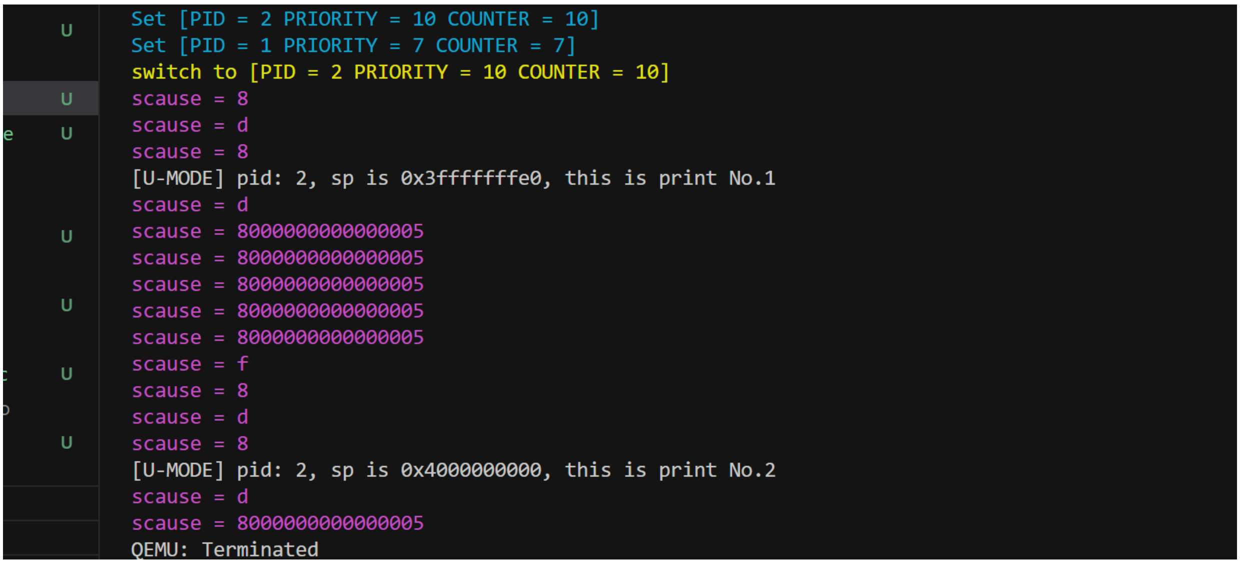
（图3-1-3）

由于程序能够从用户态进行调用，考虑可能是传递参数的环节出现问题，打印出`syscall_ID`后可以看到传递值出错（图3-1-4，值为0，而非64/172）。检查后发现是`entry.S/_traps`中存储寄存器的顺序与`struct pt_regs`对不上（没存`x0`，但是在结构体中定义了`uint64_t zero`），修改后解决问题。

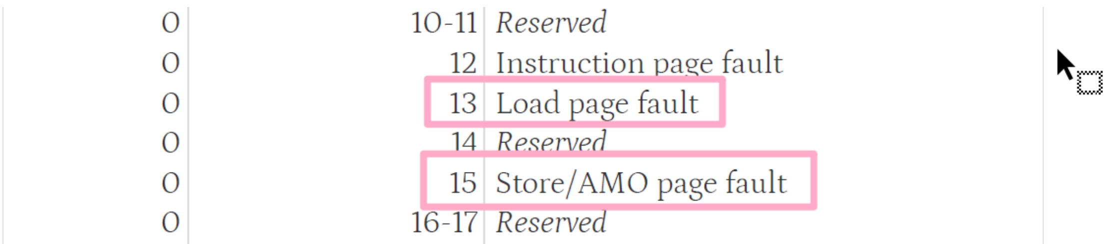


（图3-1-4）

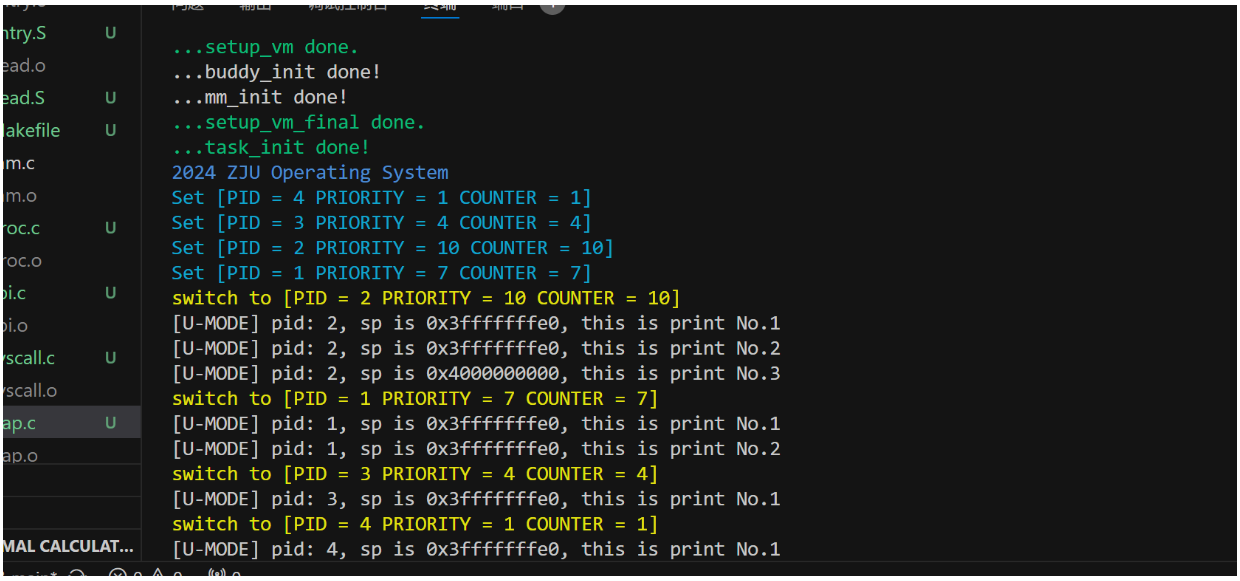
2. 测试时发现除了目标值外，`scause`还有两种需求外的值：d和f（图3-2-1，`scause`为8时代表`Environment Call from U-mode`，为8000000000000005时代表时钟中断）。查表后得知二者分别代表Load page fault和Store page fault（图3-2-2）。此时若将提示信息注释化，得到的结果和预期略有出入（图3-2-3，进程1和2多花一个时间片才进行调度；图3-2-4，进程直接调度）。



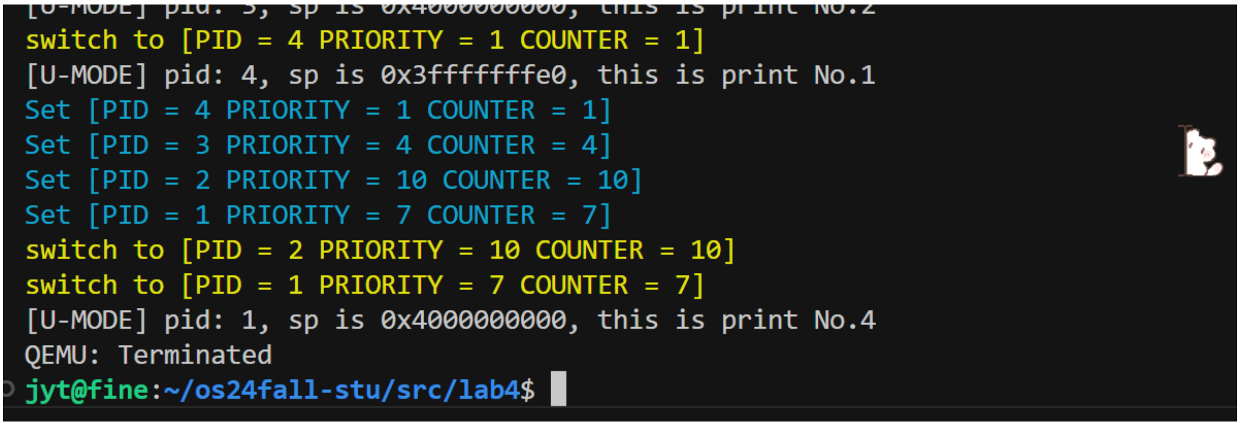
（图3-2-1）



（图3-2-2）

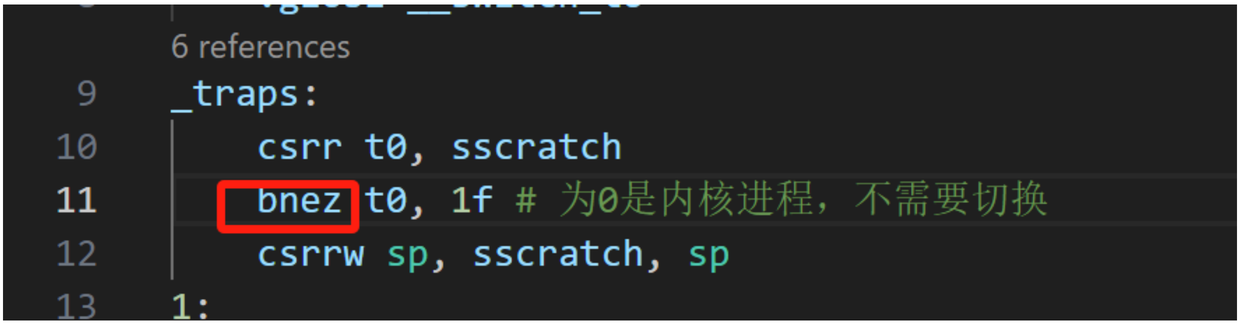


（图3-2-3）

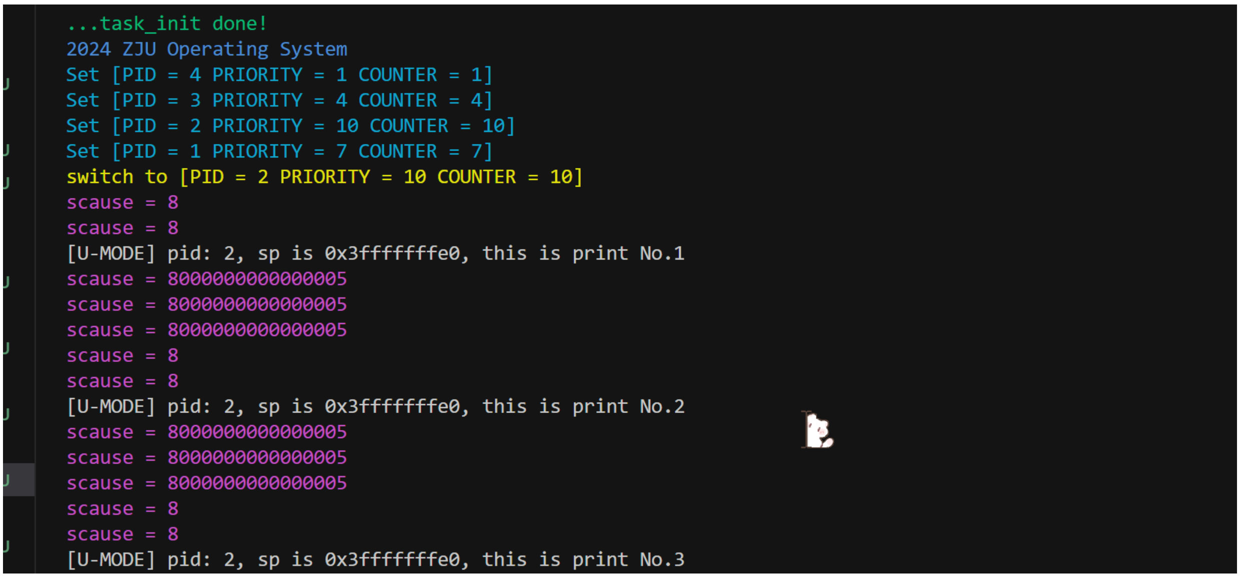


（图3-2-4）

仔细检查后发现是`traps`跳转逻辑写反了，将`beqz`（为零时跳转）写成了`bnez`（非零时跳转），修改后程序正常，不再出现错误（图3-2-6）。



（图3-2-5）



（图3-2-6）

4 思考题

4.1 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）

用户态线程和内核态线程是一对一的，因为在创建线程的时候，给每个用户态线程都设置了内核栈和用户栈（在2.2.2节中实现），使用户线程能够独立进行系统调用，而不受其他线程的干扰。

4.2 系统调用返回为什么不能直接修改寄存器？

在系统调用返回后会恢复寄存器的值（`entry.S`中实现），直接修改寄存器的话，新写入的数据会被旧数据覆盖，导致写入的数据无效。

4.3 针对系统调用，为什么要手动将 sepc + 4？

使用`ecall`指令实现系统调用，而`sepc`记录发生中断异常时的`pc`值，如果不加4（`trap.c`）就会回到调用`ecall`的地方，进入死循环。

4.4 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`是需要从磁盘中读取的数据占用的空间大小，也就是segment的大小；而`p_memsz`是在内存中占用的空间大小，它包含了`p_filesz`和`.bss`（全局变量区）。可以通过下图（来源：[用户程序和系统调用 · GitBook]）对其有一个更清晰的认识。

```
|   Offset    VirtAddr    PhysAddr   |FileSiz  MemSiz   Flg  Align
+-- 0x001000  0x03000000  0x03000000 +0x1d600  0x27240  RWE  0x1000
                 |                       |       |
                 |   +-------------------+       |
                 |   |                           |
                 |   |     |           |         |
                 |   |     |           |         |
                 |   |     +-----------+ ---     |
                 |   |     |00000000000|  ^      |
                 |   | --- |00000000000|  |      |
                 |   |  ^  |...........|  |      |
                 |   |  |  |...........|  +------+
                 |   +--+  |...........|  |
                 |      |  |...........|  |
                 |      v  |...........|  v
                 +-------> +-----------+ ---
                           |           |
                           |           |
                              Memory
```

4.5 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？

进程的虚拟地址是独立的，但是页表会被映射到不同的物理地址上，也就是不同进程的相同虚拟地址对应的物理地址是不同的，因此多个进程的栈虚拟地址可以是相同的。

操作系统负责管理页表，包括将虚拟地址映射到物理地址（页块），但是物理地址对用户是透明不可见的，因此用户只能知道虚拟地址，没有常规方法指导自己栈所在的物理地址。

5 心得体会

本实验要实现多个用户进程，以及用户态栈和内核态栈的切换，对我而言比较有难度，ELF的部分也是一知半解。

代码编写环节个人认为最难的是创建用户态进程时页表映射等部分（2.2.2节），各个参数（页表对应的虚拟地址/物理地址等）都得传对。另外因为是一个小型的项目，每个环节都是耦合的，如果只关注当前编写的代码，很容易和其他部分脱节导致出现bug（比如3.1中寄存器顺序不对导致的错误）。