浙江大学

《操作系统原理与实践》
实验报告

实验名称	：	Lab 4: RV64 用户态程序
姓　　名	：	王晓宇
学　　号	：	3220104364
电子邮箱	：	3220104364@zju.edu.cn
联系电话	：	19550222634
授课教师	：	申文博
助　　教	：	陈淦豪&王鹤翔&许昊瑞

2024 年 11 月 26 日

Lab 4: RV64 用户态程序

Lab 4: RV64 用户态程序

1 实验内容及简要原理介绍

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 实验简要原理介绍

处理器存在两种不同的模式：**用户模式（U-Mode）**和**内核模式（S-Mode）**。

- 在用户模式下，执行代码无法直接访问硬件，必须委托给系统提供的接口才能访问硬件或内存；
- 在内核模式下，执行代码对底层硬件具有完整且不受限制的访问权限，它可以执行任何 CPU 指令并引用任何内存地址。

处理器根据处理器上运行的代码类型在这两种模式之间切换。应用程序以用户模式运行，而核心操作系统组件以内核模式运行。

2 实验具体过程与代码实现

2.1 准备工程

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

```
    .rodata : ALIGN(0x1000) {
        _srodata = .;

        *(.srodata .srodata.*)
        *(.rodata .rodata.*)

        _erodata = .;
    } >ramv AT>ram

    /* lab4新增 */
    .data : ALIGN(0x1000) {
        _sdata = .;

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

        _edata = .;

        . = ALIGN(0x1000);
        _sramdisk = .;
        *(.uapp .uapp*)
        _eramdisk = .;
        . = ALIGN(0x1000);
    } >ramv AT>ram
    /* end lab4新增 */

    .bss : ALIGN(0x1000) {
        _sbss = .;

        *(.bss.stack)
        *(.sbss .sbss.*)
        *(.bss .bss.*)
```

需要修改 `defs.h`，在 `defs.h` 添加如下内容：

```
//lab4 new
#define USER_START (0x0000000000000000) // user space start virtual address
#define USER_END (0x0000004000000000) // user space end virtual address
//end____lab4 new

#define csr_read(csr)                   \
```

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

```
.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 vmlinux -bios default

debug: all
    @echo Launch qemu for debug...
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default -S -s

clean:
    $(MAKE) -C lib clean
    $(MAKE) -C init clean
    $(MAKE) -C user clean
    $(MAKE) -C arch/riscv clean
    $(shell test -f vmlinux && rm vmlinux)
    $(shell test -f vmlinux.asm && rm vmlinux.asm)
    $(shell test -f System.map && rm System.map)
    @echo -e '\n'Clean Finished

```

2.2 创建用户态进程

2.2.1 结构体更新

- 次实验只需要创建 4 个用户态进程，修改 `proc.h` 中的 `NR_TASKS` 即可
- 由于创建用户态进程要对`sepc` ，`sstatus` ，`sscratch`做设置，我们需要将其加入 `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，我们需要为每个用户态进程都创建一个页表并记录在 `task_struct` 中，可以参考的修改如下：

```c
/* 线程状态段数据结构 */
struct thread_struct {
    uint64_t ra;
    uint64_t sp;
    uint64_t s[12];
    //added in Lab4
    //必须加在s[12]后面，因为在__switch_to函数中会计算ra等变量的位置
    uint64_t sepc, sstatus, sscratch;
    //end added in Lab4
};

/* 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;
    //added in Lab4
    //必须加在thread_struct后面，因为在__switch_to函数中会计算thread变量的位置
    uint64_t *pgd;  // 用户态页表
    //end added in Lab4
};
```

这里我的结构体定义放在`proc.h`中了(其实无伤大雅)，这里要注意新增的sepc等变量的存放位置，因为我们在`__switch_to`函数中对结构体的上下文作了入栈出栈的操作实现了上下文存储和恢复，对这些变量的地址是固定的，如果你没有精力去修改你的`__switch_to`汇编，只需要加到后面即可，稍后我们还会讨论一个结构体存放位置的细节。

以下展示`__switch_to`的入栈操作，可以看到我的偏移量计算已经完成了，`ra`的偏移量在`a0`也就是`task_struct`的地址的32位处，即`thread`的地址。这部分保持不变导致了我们的新增变量只能添加到后面。

```
__switch_to:
    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)
```

2.2.2 修改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）、`SPIE`（sret 之后开启中断）、`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）
- 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
  - 将 `uapp` 所在的页面映射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` 二进制文件拷贝过去
- 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`

```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 = 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)task[i] + PGSIZE;
        //added in Lab4--------------------------------------------
        task[i]->thread.sepc = USER_START;
        task[i]->thread.sstatus |=  (uint64_t)0x40020;  //SPIE(5): 1, SUM(18): 1
        task[i]->thread.sstatus &= ~((uint64_t)0x100);  //SPP: 0
        task[i]->thread.sscratch = USER_END;

        //reuse swapper_pg_dir in user space
        task[i]->pgd = (uint64_t*)alloc_page();
        memcpy((void *)task[i]->pgd, (void *)&swapper_pg_dir, PGSIZE);

            //load elf program++++++++++++++++++++++++++++++++…

            //Or Or Or Or Or Or

            //load simple program++++++++++++++++++++++++++
            //
                //Attention: `alloc` create virtual address(just constant offset) space for user space
                //copy uapp to user space
                uint64_t* uapp_copyed = (uint64_t*)alloc_pages(((uint64_t)&_eramdisk - (uint64_t)&_sramdisk) / PGSIZE) + 1);
                memcpy((void*)uapp_copyed, (void*)&_sramdisk, (uint64_t)&_eramdisk - (uint64_t)&_sramdisk);
                create_mapping(task[i]->pgd, USER_START, (uint64_t)uapp_copyed - PA2VA_OFFSET, (uint64_t)&_eramdisk - (uint64_t)&_sramdisk, 0x1f);//U|X|W|R|V
            //
            //end load simple program++++++++++++++++++++++++++
        //new stack for user space
        //Size of stack is PGSIZE
        uint64_t* new_stack = (uint64_t*)alloc_page();
        create_mapping(task[i]->pgd, USER_END - PGSIZE, (uint64_t)new_stack - PA2VA_OFFSET, PGSIZE, 0x17);//U|W|R|V
        //end added in Lab4----------------------------------------
    }
```

这里对 `sstatus` 中的 `SPP`（使得 sret 返回至 U-Mode）、`SUM`（S-Mode 可以访问 User 页面）位的位置我们查阅手册可以得知，对相应位置置位即可

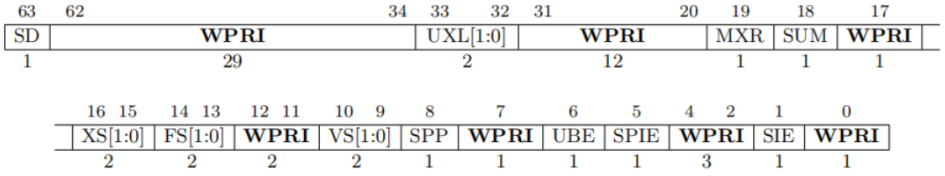


Figure 4.2: Supervisor-mode status register (**sstatus**) when SXLEN=64.

这里的变量赋值较为常规，我们首先看到拷贝实现：

这里借助`string.h`的`memcpy`实现思路，我们也在本地的`string.h`也实现了一份`memcpy`：

```
//added in Lab4
void *memcpy(void *dest, void *src, uint64_t n) {
    char *d = dest;
    char *s = src;
    while (n--) {
        *(d++) = *(s++);
    }
    return dest;
}
//end added in Lab4
```

我们再将注意力放在新建页表上，本次实验新引进了`alloc_pages`函数，找到函数定义即可发现，新建页表函数返回是页的虚拟地址：

mm.c /mnt/d/cs/os24fall-stu/src/lab4/arch/riscv/kernel - 定义 (2)

```
123
124
125 void *alloc_pages(uint64_t nrpages) {
126     uint64_t pfn = buddy_alloc(nrpages);
127     if (pfn == 0)
128         return 0;
129     return (void *)(PA2VA(PFN2PHYS(pfn)));
130 }
131
132 void *alloc_page() {
```

所以我们这里运用`create_mapping`实现页表映射的时候，传入物理地址的时候，要将`alloc_pages`返回值减去`PA2VA_OFFSET`来获得物理地址。

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**。

```
1 __switch_to:
```

```
    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)

    # added in lab4
    csrr t0,sepc
    sd t0,144(a0)
    csrr t0,sstatus
    sd t0,152(a0)
    csrr t0,sscratch
    sd t0,160(a0)
    # end of added in lab4

    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)

    # added in lab4
    ld t0,144(a1)
    csrw sepc,t0
    ld t0,152(a1)
    csrw sstatus,t0
    ld t0,160(a1)
    csrw sscratch,t0

    # change the PGD
    ld t1,168(a1)
    li t0,0xffffffdf80000000
    sub t1,t1,t0
    # set `satp` with `swapper_pg_dir`
    srli t1, t1, 12
    li t0, 0x8
    slli t0, t0, 30
    slli t0, t0, 30
    or t0, t0, t1
    csrw satp, t0

    # need a fence to ensure the new translations are in use
    sfence.vma zero, zero
    # end of added in lab4

    ret
```

这里的保存上下文和之前的一致，调整栈指针即可，另外通过设置**satp**来改变页表和通过 `sfence.vma` 来刷新 TLB 和 ICache

2.4 更新中断处理逻辑

与 ARM 架构不同的是，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，使用的运行栈也是用户栈，因此当触发异常时，我们首先要对栈进行切换（从用户栈切换到内核栈）。同理，当我们完成了异常处理，从 S-Mode 返回至 U-Mode 时，也需要进行栈切换（从内核栈切换到用户栈）。

2.4.1 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:
    # be deleted in lab4
    # la t0, dummy
    # csrw sepc,t0
    # end of be deleted in lab4

    # added in lab4
    # swap the kernel stack and user stack
    csrr t0,sscratch
    csrw sscratch,sp
    mv sp,t0
    # end of added in lab4

    sret
```

我们在之前结构体修改中完成了对`sepc`的修改，使其指向了`USER_START`,所以这里将之前默认进程的执行空间`dummy`改为了`USER_START`，这里只需要将设置sepc的部分注释掉即可，之后程序便会跳转到我们指定程序的第一条指令：

src > lab4 > user > ASM start.S

```
1       .section .text.init
2       .global _start
3   _start:
4       j   main
5
```

> _dummy

```
int counter = 0;

static inline long getpid() {
    long ret;
    asm volatile ("li a7, %1\n"
                  "ecall\n"
                  "mv %0, a0\n"
                : "+r" (ret)
                : "i" (SYS_GETPID));
    return ret;
}

int main() {
    register void *current_sp __asm__("sp");
    while (1) {
        printf("[U-MODE] pid: %ld, sp is %p, this is print No.%d\n", getpid(), current_sp, ++counter);
        for (unsigned int i = 0; i < 0x4FFFFFFF; i++);
    }
    return 0;
}
```

可以看到`main`函数的作用是输出此时的`PID`、`sp`、执行计数。

2.4.2 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:

    # added in lab4
    # swap the kernel stack and user stack
    csrr t0,sscratch
    beq t0, x0, label1
    csrw sscratch,sp
    mv sp,t0
    # end of added in lab4
label1:

  \\save the context as in Lab3

    # added in lab4
    # swap the kernel stack and user stack
    csrr t0,sscratch
    beq t0, x0, label2
    csrw sscratch,sp
    mv sp,t0
    # end of added in lab4
label2:
    sret
```

这里和之前的汇编代码做出跳转的修改，即判断`sscratch`是否为**0**，判断是否是内核线程来决定是否要对栈做出修改互换。这里如果为**0**，我们直接跳过互换栈的修改，直接进行`trap_handler`的操作；在执行结束之后我们再利用跳转的操作，如果是内核线程则不进行栈的互换以实现状态不变；如果是用户态线程触发的异常，我们要将用户栈和内核栈互换回来。

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还需要寄存器的值，因此我们需要在 `trap_handler()` 里面进行捕获。修改 `trap_handler()` 如下：

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    uint64_t scause_code = scause;
    uint64_t highest_bit = (scause_code >> 63) & 1;
    switch (highest_bit)
    {
    case 1:
        if(scause == 0x8000000000000005){
            // printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }else{
            printk("Non-Timer Interrupt\n");
        }
        break;
    case 0:
        if(scause == 0x8){
            // printk("Environment call from U-mode\n");
            switch (regs->reg17)
            {
            case 64://sys_write
                sys_write(regs->reg10, (const char*)regs->reg11, (uint64_t)regs->reg12, regs);
                break;

            case 172://sys_getpid
                sys_getpid(regs);
                break;
            }
        }
        regs->sepc += 0x4;
    }

}
```

这里我们查询了`sstatus`异常code的介绍：

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*

Table 4.2: Supervisor cause register (scause) values after trap. Synchronous exception priorities are given by Table 3.7.

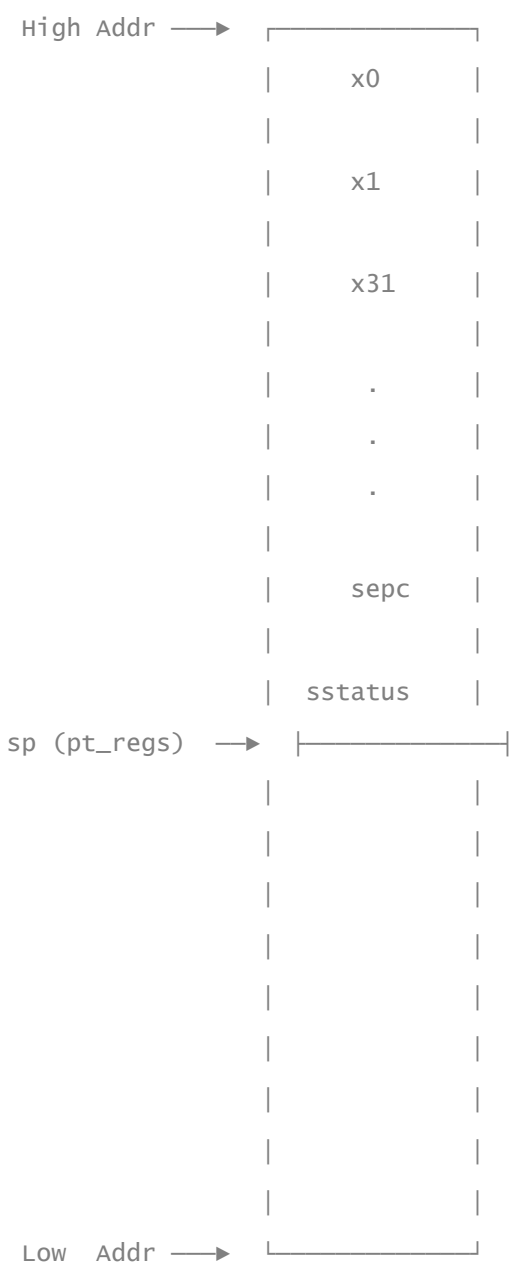
我们很容易对中断异常的判断进行分类讨论，即当`Interrupt`位为0时是异常，此时`Exception code`为8时说明是`Environment Call from U-mode`，我们要对其进行系统调用表查询以实现打印或者获取PID的功能。

> 需要注意的是：我们在这里增加了`trap_handler`的参数个数，我们将栈指针当作结构体的头部传入，这样使得我们可以对栈上内容直接进行修改，因此要在`_traps`中调用`trap_handler`之前，要对a2寄存器赋值sp来实现变量传入

```
_traps:
    ---
    csrr a0,scause
 csrr a1,sepc
 # added in lab4
 mv a2,sp
 # end of added in lab4
 call trap_handler
 ---
```

在 `_traps` 中我们将寄存器的内容**连续**的保存在内核栈上，因此我们可以将这一段看做一个叫做 `pt_regs` 的结构体。我们可以从这个结构体中取到相应的寄存器的值（比如 `syscall` 中我们需要从 **a0 ~ a7** 寄存器中取到参数）。这个结构体中的值也可以按需添加，同时需要在 `_traps` 中存入对应的寄存器值以供使用，示例如下图：

> 这里我和实验指导的保存栈顺序并不一致，因此要做出一定修改，我的栈保存顺序如下：



```
37  struct pt_regs
38  {
39      //因为我的保存上下文顺序不一样，从高地址开始存储x0-x31
40      //所以这里的顺序也是从x31-x0开始捕获
41      uint64_t sstatus;
42      uint64_t sepc;
43      uint64_t reg31;
44      uint64_t reg30;
45      uint64_t reg29;
46      uint64_t reg28;
47      uint64_t reg27;
48      uint64_t reg26;
49      uint64_t reg25;
50      uint64_t reg24;
51      uint64_t reg23;
52      uint64_t reg22;
53      uint64_t reg21;
54      uint64_t reg20;
55      uint64_t reg19;
56      uint64_t reg18;
57      uint64_t reg17;
58      uint64_t reg16;
59      uint64_t reg15;
60      uint64_t reg14;
61      uint64_t reg13;
62      uint64_t reg12;
63      uint64_t reg11;
64      uint64_t reg10;
65      uint64_t reg9;
66      uint64_t reg8;
67      uint64_t reg7;
68      uint64_t reg6;
69      uint64_t reg5;
70      uint64_t reg4;
71      uint64_t reg3;
72      uint64_t reg2;
73      uint64_t reg1;
74      uint64_t reg0;
75  };
```

这里巧妙地利用了内存分配连续的性质，我们可以将栈上的变量以增加结构体大小的方法来进行参数的寻找和修改，我们对变量的直接修改直接影响到栈的保存内容，这对于我们之后的变量修改是十分便利的。

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

src > lab4 > arch > riscv > include > C syscall.h > ...

```c
#ifndef __SYSCALL_H__


void sys_write(unsigned int fd, const char* buf, uint64_t count, struct pt_regs *regs);
void sys_getpid(struct pt_regs *regs);

#endif
```

C syscall.c U

src > lab4 > arch > riscv > kernel > C syscall.c > ...

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

extern struct task_struct *current;

void sys_write(unsigned int fd, const char* buf, uint64_t count, struct pt_regs *regs){

    if(fd == 1){
        uint64_t i = 0;
        for( ; i < count; i++){
            printk("%c", buf[i]);
        }
        regs->reg10 = i;
    }
}


void sys_getpid(struct pt_regs *regs){
    regs->reg10 = current->pid;
}
```

这里参照给定传参格式，我们可以调用`printk`实现`sys_write`的输出并返回打印字符数；利用`pt_regs`的结构体内容，我们可以很快提取到他对应进程的PID

2.6 调整时钟中断

接下来我们需要修改 `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()`

```c
int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");

    // printk("stext: %llx\n", &_stext);
    // printk("srodata: %llx\n", &_srodata);

    // if(*_stext = 0x1) {
    //     printk("stext write: pass!\n");
    // }
    // if(*_srodata = 0x1) {
    //     printk("srodata write: pass!\n");
    // }

    // asm volatile("call _srodata");

    //added by lab4
    schedule();
    //end added by 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

```
_start:
    la sp, boot_stack_top

    call setup_vm
    call relocate

    call mm_init
    call setup_vm_final
    call task_init

    la t0, _traps
    csrw stvec, t0
    li t0, 0x20
    csrs sie, t0
    rdtime t0
    li t1, 10000000
    add a0, t0, t1
    call sbi_set_timer

    # canceled by lab4
    # li t0, 0x2
    # csrs sstatus, t0
    # end canceled by lab4

    call start_kernel
```

2.7 测试纯二进制文件

> 输出结果：

```
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
...buddy_init done!
...mm_init done!
...task_init done!
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

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
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.4
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.5
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.6

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-MODE] pid: 1, sp is 0x3fffffffe0, this is print No.4
```

2.8 添加 ELF 解析与加载

2.8.1 ELF 格式

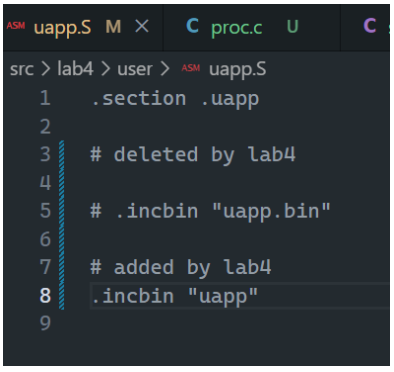
通过 `readelf` 来查看 ELF 可执行文件

```
axin0401@LAPTOP-0P208VUK:/mnt/d/cs/os24fall-stu/src/lab4/user$ readelf -a -W uapp
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x100e8
  Start of program headers:          64 (bytes into file)
  Start of section headers:          5960 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         9
  Section header string table index: 8

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000000100e8 0000e8 00106c 00  AX  0   0  4
  [ 2] .rodata           PROGBITS        0000000000011158 001158 000081 00   A  0   0  8
  [ 3] .bss              NOBITS          00000000000121e0 0011d9 0003f0 00  WA  0   0  8
  [ 4] .comment          PROGBITS        0000000000000000 0011d9 00002b 01  MS  0   0  1
  [ 5] .riscv.attributes RISCV_ATTRIBUTES 0000000000000000 001204 00002e 00      0   0  1
  [ 6] .symtab           SYMTAB          0000000000000000 001238 0003d8 18      7  24  8
  [ 7] .strtab           STRTAB          0000000000000000 001610 0000ef 00      0   0  1
  [ 8] .shstrtab         STRTAB          0000000000000000 0016ff 000049 00      0   0  1
Key to Flags:
```

2.8.2 ELF文件解析

首先我们需要将 `uapp.S` 中的 **payload** 给换成我们的 **ELF** 文件：



这时候从 `_sramdisk` 开始的数据就变成了名为 `uapp` 的 **ELF** 文件，也就是说 `_sramdisk` 处 **32-bit** 的数据不再是第一条指令，而是 **ELF Header** 的开始。

> 这里单纯的拷贝除去Header的ELF时，会在第一条指令没对齐的情况下导致系统找不到执行指令，这也是我们稍后考虑的Offset问题。

我们在**proc.c**中添加**load_program**函数，以获取**ELF**文件并实现载入：

```c
//lab4 add
extern uint64_t* swapper_pg_dir;
extern char* _sramdisk;
extern char* _eramdisk;

void load_program(struct task_struct *task) {
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)&_sramdisk;
    Elf64_Phdr *phdrs = (Elf64_Phdr *)((char*)(&_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
            uint64_t offset = phdr->p_vaddr & (PGSIZE-0x1);
            uint64_t* elf_copyed = (uint64_t*)alloc_pages(((uint64_t)phdr->p_memsz+(uint64_t)offset) / PGSIZE) + 1);
            memcpy((void*)(elf_copyed+offset), (void*)((char*)&_sramdisk+phdr->p_offset), (uint64_t)phdr->p_filesz);
            memset((void*)(elf_copyed+offset+phdr->p_filesz), 0, (uint64_t)(phdr->p_memsz - phdr->p_filesz));

            // do mapping
            uint64_t perm = ((phdr->p_flags & PF_X) ? 0x8 :0x0) | ((phdr->p_flags & PF_W) ? 0x4 :0x0) |((phdr->p_flags & PF_R) ? 0x2 :0x0);//|XWR
            perm |= 0x11; //U|V
            create_mapping(task->pgd, phdr->p_vaddr, (uint64_t)elf_copyed - PA2VA_OFFSET, (uint64_t)phdr->p_memsz+(uint64_t)offset, perm);//U|X|W|R|V
        }
    }
    task->thread.sepc = ehdr->e_entry;
}
//end lab4 add
```

其中相对文件偏移 `p_offset` 指出相应 **segment** 的内容从 **ELF** 文件的第 `p_offset` 字节开始，在文件中的大小为 `p_filesz`，它需要被分配到以 `p_vaddr` 为首地址的虚拟内存位置，在内存中它占用大小为 `p_memsz`。也就是说，这个 **segment** 使用的内存就是 `[p_vaddr, p_vaddr + p_memsz)` 这一连续区间，然后将 **segment** 的内容从 **ELF** 文件中读入到这一内存区间，并将 `[p_vaddr + p_filesz, p_vaddr + p_memsz)` 对应的物理区间**清零**。（本段内容引用自南京大学 PA）

需要注意的是，因为虚拟内存和物理内存的映射是按页为单位的，如果 `e_entry` 并没有按页对齐的话，需要在拷贝的时候考虑 **offset** 的问题。

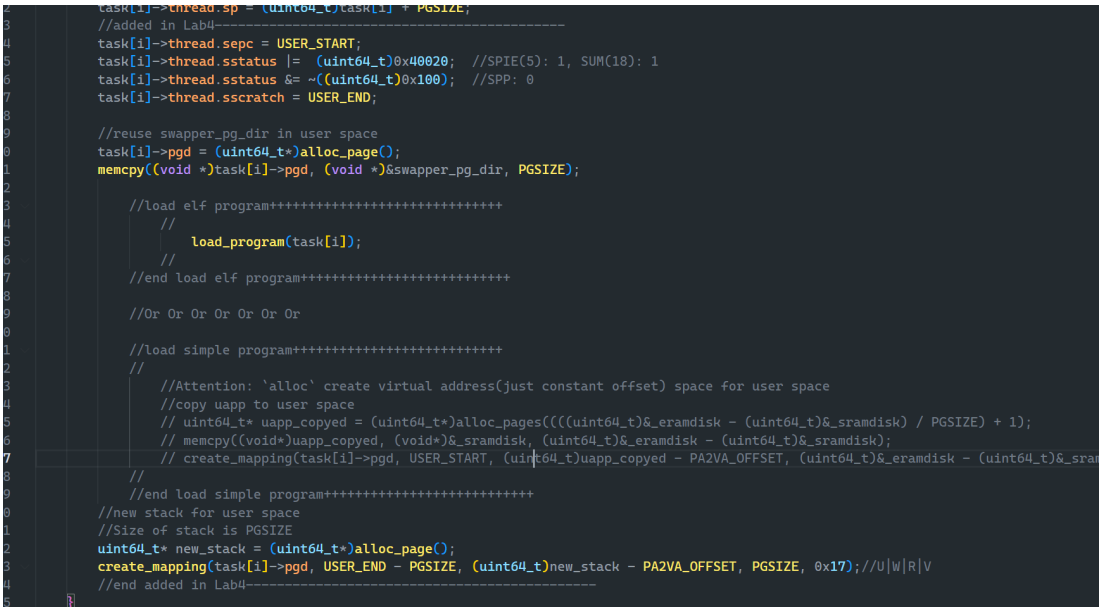
所以我们将除去Header部分按照其处于页中的原位置完全拷贝下即可。

> 这里注意到我的extern变量是char*类型，例如extern char* _sramdisk;，但是在获取其地址时，我仍然需要利用&操作获取变量地址；但是当我extern变量时采取extern char _sramdisk[];时便可以使用变量名直接调用。
>
> 在本次实验中这段话或许是我理解有些问题，我感觉还是要做出一定修改，也有可能是我的C编译器版本有问题？或是其他原因等等，还望助教指正，这里建议修改为：`_sramdisk` 类型使用的是 `char _sramdisk[]`。

这里有不少例子可以举，为了避免同学们在实验中花太多时间，我们告诉大家可以怎么找到实验中这些相关变量：(注意以下的 `_sramdisk` 类型使用的是 `char*`，如果你在使用其他类型，需要根据你使用的类型去调整针对指针的算数运算)

- `Elf64_Ehdr *ehdr = (Elf64_Ehdr *)_sramdisk`，从地址_sramdisk开始，便是我们要找的Ehdr
- `Elf64_Phdr *phdrs = (Elf64_Phdr *)(_sramdisk + ehdr->phoff)`，是一个Phdr数组，其中的每个元素都是一个 `Elf64_Phdr`
- `phdrs + 1` 是指向第二个Phdr的指针
- `phdrs->p_type == PT_LOAD`，说明这个Segment的类型是LOAD，需要在初始化时被加载进内存

现在我们在**task_init**中对载入做出以下修改补充：



```
    task[i]->thread.sp = (uint64_t)task[i] + PGSIZE;
    //added in Lab4------------------------------------------
    task[i]->thread.sepc = USER_START;
    task[i]->thread.sstatus |=  (uint64_t)0x40020;  //SPIE(5): 1, SUM(18): 1
    task[i]->thread.sstatus &= ~((uint64_t)0x100);  //SPP: 0
    task[i]->thread.sscratch = USER_END;

    //reuse swapper_pg_dir in user space
    task[i]->pgd = (uint64_t*)alloc_page();
    memcpy((void *)task[i]->pgd, (void *)&swapper_pg_dir, PGSIZE);

        //load elf program++++++++++++++++++++++++++++++
            //
                load_program(task[i]);
            //
        //end load elf program++++++++++++++++++++++++++

        //Or Or Or Or Or Or

        //load simple program++++++++++++++++++++++++++
        //
            //Attention: `alloc` create virtual address(just constant offset) space for user space
            //copy uapp to user space
            // uint64_t* uapp_copyed = (uint64_t*)alloc_pages((((uint64_t)&_eramdisk - (uint64_t)&_sramdisk) / PGSIZE) + 1);
            // memcpy((void*)uapp_copyed, (void*)&_sramdisk, (uint64_t)&_eramdisk - (uint64_t)&_sramdisk);
            // create_mapping(task[i]->pgd, USER_START, (uint64_t)uapp_copyed - PA2VA_OFFSET, (uint64_t)&_eramdisk - (uint64_t)&_sram
        //
        //end load simple program++++++++++++++++++++++++++
    //new stack for user space
    //Size of stack is PGSIZE
    uint64_t* new_stack = (uint64_t*)alloc_page();
    create_mapping(task[i]->pgd, USER_END - PGSIZE, (uint64_t)new_stack - PA2VA_OFFSET, PGSIZE, 0x17);//U|W|R|V
    //end added in Lab4------------------------------------------
}
```

这里将之前**uapp**的拷贝部分删掉，直接进行**ELF**文件的拷贝，如果想要再进行**.o**文件的载入，只需要将原部分注释回来即可。

> *参考：*
>
> [ELF 文件解析 3-段 - 知乎 (zhihu.com)]
>
> `p_flags` （**Program Header-Flags**）：此字段（4 字节）给出本段内容的属性，指明了段的权限。虽然 ELF 文件格式中没有规定，但是一个可执行程序至少会有一个可加载的段。当为可加载段创建内存镜像时，系统会按照 p_flags 的指示给段赋予一定的权限。下方为源码中定义、对应取值及其含义。
>
> ```
> /* Legal values for p_flags (segment flags).  */
>
> #define PF_X            (1 << 0)        /* Segment is executable */
> #define PF_W            (1 << 1)        /* Segment is writable */
> #define PF_R            (1 << 2)        /* Segment is readable */
> #define PF_MASKOS       0x0ff00000      /* OS-specific */
> #define PF_MASKPROC     0xf0000000      /* Processor-specific */
> ```
>
> 对应字段的值与属性的关系，值为 0x1：可执行；值为 0x2：可写；值为 0x4：可读。特殊地，被 PF_MASKOS 所覆盖的权限值是为特殊操作系统保留的，被 PF_MASKPROC 所覆盖的权限值是为特殊处理器保留的。

3 实验结果与分析

仅载入.o:

```
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
...buddy_init done!
...mm_init done!
...task_init done!
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

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

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

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.4
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.5
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.6

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

仅载入ELF:

```
Boot HART MEDELEG         : 0x0000000000f0b509
...buddy_init done!
...mm_init done!
...task_init done!
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

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

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

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.4
[U-MODE] pid: 2, sp is 0x3ffffffe0, this is print No.5
QEMU: Terminated
axin0401@LAPTOP-0P208VUK:/mnt/d/cs/os24fall-stu/src/lab4$
```

4 思考题与心得体会

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。

   在本次实验中每个用户线程`task`（用户应用程序创建）都直接映射到一个内核线程，拥有自己的独立页表，每个用户线程都在属于自己的内存空间上进行程序执行；在输出打印我们也可以看到，每个用户态线程的计数器相互独立，互不影响。

2. 系统调用返回为什么不能直接修改寄存器？

   我们在`_trap`中调用`trap_handler`时，系统会将此时的寄存器自动地保存，类似于我们实现的上下文保存机制，如果直接进行寄存器的修改，我们在从`trap_handler`返回时，系统又会将之前保存的寄存器内容重新覆盖，造成实际修改并不成功。

   我们将寄存器结构体直接传入函数中（实际上就是栈指针）进行修改时，实际上已经对内存地址上的值进行了修改，达到了我们想要的效果。

3. 针对系统调用，为什么要手动将 `sepc + 4`？

当异常发生时，会自动将当前的PC值保存到`sepc`寄存器中，这个值表示异常发生时指令的地址，因为系统调用也是Exception的一种，我们处理完成异常之后，该异常指令就应该跳过不执行了，否则会一直跳入异常处理程序造成系统死循环。修改 `spec` 的地址之后，这样 `sret` 之后程序就可以继续正常运行了。

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？

> 参考：p_filesz与p_memsz的Elf32_Phdr差异-腾讯云开发者社区-腾讯云 (tencent.com)

`p_memsz`标识的大小通常要比`p_filesz` 大。

`p_memsz` 标识的大小包含未初始化数据的`.bss`部分。在ELF文件中，`.bss`段是一个特殊的段，用于存储未初始化的全局变量和静态变量，`.bss`段在可执行文件中不占用实际的磁盘空间，它只在ELF文件加载到内存后运行时才占据空间，方便了我们压缩可执行文件大小，节省了磁盘空间又不影响程序执行。

- `p_filesz` **(File Size)**：

  `p_filesz`表示该段在文件中的大小（以字节为单位），它指示了在文件中实际存储的数据量（不包含.bss段）。

- `p_memsz` **(Memory Size)**：

  `p_memsz`表示该段在内存中的大小（以字节为单位），它指示了在内存中分配给该段的空间大小，包含代码或数据的段（以及.bss段）。

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？

我们为每个线程分配了属于他的页表，在栈虚拟地址相同的情况下，在切换完成线程之后更换页表，不同进程的栈通过自己的页表映射得到的物理地址不同，所以其访问的内存实际上不是同一块，这样就显得合理多了。

> 参考:揭示/proc/pid/pagemap的力量：在Linux中将虚拟地址映射到物理地址_linux pagemap-CSDN博客

没有常规的方法知道自己的物理地址，`/proc/pid/pagemap`是Linux操作系统中的一个特殊文件，它提供了一种机制将虚拟内存地址映射到物理内存地址。在Linux中，每个进程都有一个唯一的进程ID(PID)，`/proc/pid/pagemap`文件存储了与该进程相关联的页面映射信息。访问`/proc/pid/pagemap`文件

需要**root**权限或具有相应权限的用户身份，用户态程序权限一般是不足以访问该文件来获取物理地址的。