Lab 5: RV64 用户态程序

课程名称 ： 操作系统 **指导老师 ：** 寿黎但

姓名：

庄毅非 3200105872 胡競文 3200105990

分工：

庄毅非：修改 head.S，start_kernel，完成 elf 文件加载

胡競文：实现进程初始化，实现 ecall 中断处理和系统调用

1、实验目的

- 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。

2、实验环境

- Environment in previous labs

3、实验步骤

3.1 准备工作

- 这一步主要就是将仓库中的 elf.h 等文件添加到 lab5，并修改对应的 makefile 来确保项目可以编译。
- 修改根目录下的 `Makefile`，在 all 和 clean 操作中添加对 user 文件夹中的 `Makefile` 文件的操作

```
all:
    ${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 the qemu ......
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios
default

debug: all
    @echo Launch the 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 System.map && rm System.map)
    @echo -e '\n'Clean Finished

```

3.2 创建用户态进程

- 修改 `proc.h` 文件夹中的 `NR_TASK` 为 1 + 4，表示实验中只有一个 idle 内核线程和 4 个用户线程

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

- 修改 `thread_struct` 如下，添加对 sepc，sstatus 和 sscratch 寄存器的存储，在 `__switch_ro` 中使用

```
struct thread_struct
{
  unsigned long ra; // return address
  unsigned long sp;
  unsigned long s[12];
  unsigned long sepc;
  unsigned long sstatus;
  unsigned long sscratch;
};

```

- 修改 task_init 如下

```
// kernel/vm.c
// 将 src 开始的一个 page 拷贝到 dest 为起始的一个 page 上
void copy(void *src, void *dest)
{
    unsigned long *beh = (unsigned long *)(dest);
    unsigned long *pre = (unsigned long *)(src);
    for (int i = 0; i < PGSIZE / 8; i++)
    {
        ((unsigned long *)(beh))[i] = ((unsigned long *)(pre))[i];
    }
}
// kernel/proc.c
void task_init()
{
```

```
    // 初始化 idle 线程
    idle = (struct task_struct *)kalloc();
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;
    current = idle;
    task[0] = idle;

    for (int i = 1; i <= NR_TASKS - 1; ++i)
    {
        task[i] = (struct task_struct *)kalloc();
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand();
        task[i]->pid = i;
        task[i]->thread.ra = (unsigned long)__dummy;
        task[i]->thread.sp = (unsigned long)task[i] + PGSIZE;
        task[i]->thread_info = (struct thread_info *)(alloc_page());
        task[i]->thread_info->kernel_sp = task[i]->thread.sp;
        // 分配用户栈
        task[i]->thread_info->user_sp = (alloc_page());
        // 用户页表初始化
        pagetable_t pgtbl = (unsigned long *)(alloc_page());
        // 将内核页表拷贝到用户页表中
        copy(swapper_pg_dir, pgtbl);
        // 完成用户栈的映射
        create_mapping(pgtbl, USER_END - PGSIZE, (unsigned long)(task[i]->thread_info->user_sp - PA2VA_OFFSET), PGSIZE, READABLE | WRITABLE | USER | VALID);
        // 用户程序占据的页面数
        unsigned long user_page_count = PGROUNDUP(uapp_end - uapp_start) / 4096;
        // 为用户程序分配所需的 pages
        unsigned long *user_app_pages = (unsigned long *)alloc_pages(user_page_count);
        for (int i = 0; i < user_page_count; i++)
        {
            // 将用户程序拷贝到各自维护的 pages 中，保障多个进程之间的地址空间隔离
            copy((unsigned long *)((unsigned long)(uapp_start) + PGSIZE * i), (unsigned long *)((unsigned long)(user_app_pages) + PGSIZE * i));
        }
        // 完成用户代码段的映射
```

```
        create_mapping(pgtbl, USER_START, (unsigned long)((unsigned long)(user_app_pages)-PA2VA_OFFSET), (unsigned long)uapp_end - (unsigned long)uapp_start, USER | VALID | READABLE | WRITABLE | EXECUTABLE);
        // set sepc
        task[i]->thread.sepc = (unsigned long)(USER_START);
        // set sstatus
        unsigned long sstatus = csr_read(sstatus);
        // set spp bit to 0
        sstatus &= (~(1 << 8));
        // set SPIE bit to 1
        sstatus |= (1 << 5);
        // set SUM bit to 1
        sstatus |= (1 << 18);
        task[i]->thread.sstatus = sstatus;
        // user mode stack
        task[i]->thread.sscratch = USER_END;
        // set thread satp
        unsigned long satp = csr_read(satp);
        task[i]->pgd = (pagetable_t)(((satp >> 44) << 44) | (((unsigned long)(pgtbl)-PA2VA_OFFSET) >> 12));
    }
    /* YOUR CODE HERE */
    printk("...proc_init done!\n");
}

```

- 修改 `__switch_to`，加入保存 / 恢复 `sepc` `sstatus` `sscratch` 以及 切换页表的逻辑。

```
 .globl __switch_to
__switch_to:
    # save state to prev process
    # YOUR CODE HERE
    # skip the first 5 virables
    addi t0, a0, 8 * 5
    sd ra, 0(t0)
    sd sp, 8(t0)

    addi t0, t0, 8 * 2
    sd s0, 0*8(t0)
    sd s1, 1*8(t0)
    sd s2, 2*8(t0)
    sd s3, 3*8(t0)
    sd s4, 4*8(t0)
    sd s5, 5*8(t0)
    sd s6, 6*8(t0)
```

```
    sd s7, 7*8(t0)
    sd s8, 8*8(t0)
    sd s9, 9*8(t0)
    sd s10, 10*8(t0)
    sd s11, 11*8(t0)
    # save sepc, sstatus and sscratch
    addi t0,t0,12 * 8
    csrr s0,sepc
    sd s0, 0 * 8(t0)
    csrr s0,sstatus
    sd s0,1 * 8(t0)
    csrr s0,sscratch
    sd s0, 2 * 8(t0)

    addi t0,t0,3 * 8
    # save satp
    csrr s0,satp
    sd s0,0(t0)
    # restore state from next process
    # YOUR CODE HERE
    addi t1, a1, 8 * 5
    ld ra, 0(t1)
    ld sp, 8(t1)
    addi t1, t1, 8 * 2
    ld s0, 0*8(t1)
    ld s1, 1*8(t1)
    ld s2, 2*8(t1)
    ld s3, 3*8(t1)
    ld s4, 4*8(t1)
    ld s5, 5*8(t1)
    ld s6, 6*8(t1)
    ld s7, 7*8(t1)
    ld s8, 8*8(t1)
    ld s9, 9*8(t1)
    ld s10, 10*8(t1)
    ld s11, 11*8(t1)
    # restore sepc, sstatus and sscratch
    addi t1,t1,12 * 8
    ld t2, 0(t1)
    csrw sepc, t2
    ld t2, 8(t1)
    csrw sstatus, t2
    ld t2, 16(t1)
    csrw sscratch, t2
    addi t1,t1,3 * 8;
```

```
    ld t2,0(t1)
    # switch satp
    csrw satp,t2
    # flush tlb and Icache
    sfence.vma
    fence.i
    ret

```

3.3 修改中断入口 / 返回逻辑（ _trap ）以及中断处理函数 （ trap_handler ）

- 修改 `__dummy`

```
__dummy:
    # 在返回用户段的时候，交换 sp 寄存器和 sscratch 寄存器，完成用户栈和内核栈之间的切换
    csrrw sp, sscratch, sp
    sret
```

- 修改 `_traps`，如果触发中断的是用户线程（根据 sscratch 寄存器是否为 0 判断），那么执行栈的切换

```
#在本实验中，触发中断的线程应该只有用户线程，所以这里只留下 csrrw sp,sscratch,sp 指令，程序也可以正常运行
    csrrw sp,sscratch,sp
    addi sp,sp,-34 * 8

    sd x1, 1 * 8(sp)
    addi x1,sp,34 * 8
    sd x1, 2 * 8(sp)
    sd x3, 3 * 8(sp)
    ...
    ld sp,2 * 8(sp)
#交换内核栈和用户栈
    csrrw sp,sscratch,sp
    sret
```

- 修改 `trap_handler`，完成对 `ecall` 的捕获（如果 scause 的第八位为 1，首位为 0，表示用户调用 ecall）

```
// syscall.h
struct pt_regs
{
  // unsigned long x[32];
  unsigned long zero;
```

```
  unsigned long ra;
  unsigned long sp;
  unsigned long gp;
  unsigned long tp;
  unsigned long t0, t1, t2;
  unsigned long fp;
  unsigned long s1;
  unsigned long a0, a1, a2, a3, a4, a5, a6, a7;
  unsigned long s2, s3, s4, s5, s6, s7, s8, s9, s10, s11;
  unsigned long t3, t4, t5, t6;
  unsigned long sepc;
  unsigned long sstatus;
};
// trap.c
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs
*regs)
{
    if ((scause >> 63) & 1)
    {
        // interrupt
        if ((scause & 5) == 5)
        {
            clock_set_next_event();
            do_timer();
        }
    }
    else
    {
        // exception
        if (scause & 8)
        {
            // ecall from u_mode
            syscall(regs);
        }
    }
}
```

3.4 添加系统调用

- 添加 `syscall.h` 文件

```
#ifndef __SYSCALL_H__
#define __SYSCALL_H__

```

```c
#define SYS_WRITE 64
#define SYS_GETPID 172

struct pt_regs
{
  // unsigned long x[32];
  unsigned long zero;
  unsigned long ra;
  unsigned long sp;
  unsigned long gp;
  unsigned long tp;
  unsigned long t0, t1, t2;
  unsigned long fp;
  unsigned long s1;
  unsigned long a0, a1, a2, a3, a4, a5, a6, a7;
  unsigned long s2, s3, s4, s5, s6, s7, s8, s9, s10, s11;
  unsigned long t3, t4, t5, t6;
  unsigned long sepc;
  unsigned long sstatus;
};
int sys_write(unsigned int fd, const char *buf, unsigned long count);
int sys_getpid();
void syscall(struct pt_regs *);
#endif
```

- `syscall.c` 文件内容如下

```c
#include "syscall.h"
#include "printk.h"
#include "mm.h"
#include "defs.h"
#include "vm.h"
#include "proc.h"
extern struct task_struct *current;
int sys_write(unsigned int fd, const char *buf, unsigned long count)
{
    // 分配新的连续 page 存储用户要输出的字符串
    char *buffer = (char *)(alloc_page(PGROUNDUP(count) / PGSIZE));
    // 拷贝用户字符串
    for (int i = 0; i < count; i++)
    {
        buffer[i] = buf[i];
    }
    // 设置结束字符
    buffer[count] = '\0';
```

```
    // 调用 printk 函数完成输出，记录输出的字符数
    int result = printk(buffer);
    // 释放申请的内存页面
    free_pages((unsigned long)(buffer));
    return result;
}
int sys_getpid()
{
    // 返回当前线程的线程号
    return current->pid;
}
void syscall(struct pt_regs *regs)
{
    unsigned long sys_code = regs->a7;
    int return_value = -1;
    // 判断服务类型
    switch (sys_code)
    {
    case SYS_GETPID:
    {
        return_value = sys_getpid();
        break;
    }
    case SYS_WRITE:
    {
        unsigned long fd = regs->a0;
        unsigned long buf = regs->a1;
        unsigned long count = regs->a2;
        return_value = sys_write(fd, (const char *)buf, count);
        break;
    }
    }
    // write result to a0
    regs->a0 = return_value;
    regs->sepc += 4;
}
```

3.5 修改 head.S 以及 start_kernel

- 修改 `head.S`

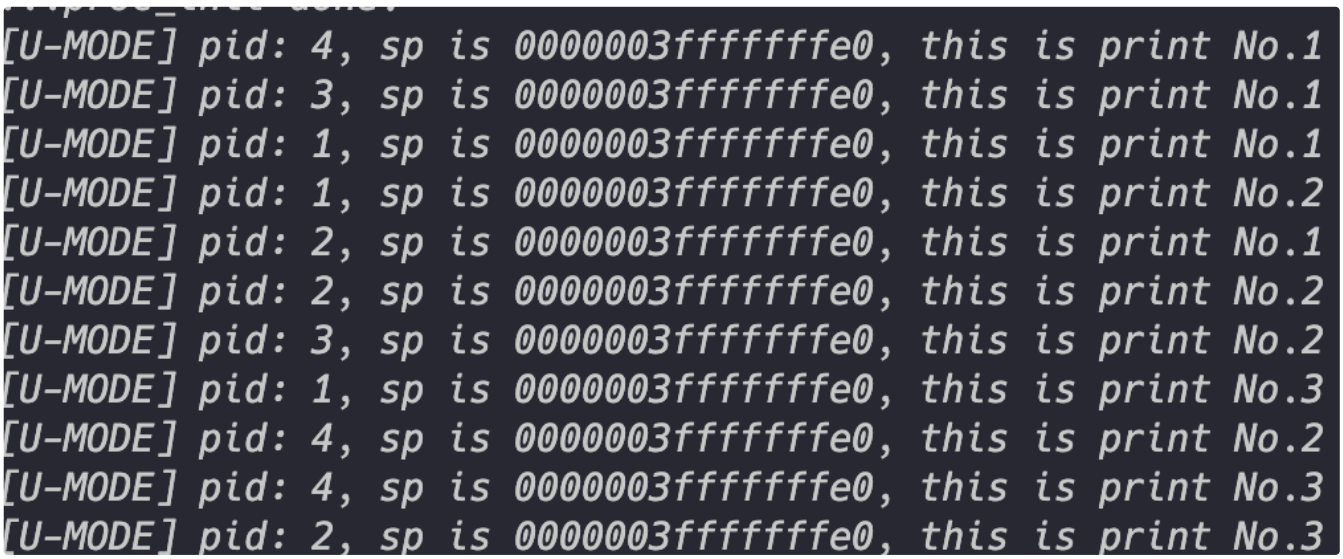
```
# 将原来对 sstatus 设置的 sie bit 代码注释，保证线程调度的时候不会被中断
# set sstatus[SIE] = 1
# csrr t0,sstatus
# ori t0,t0,0x2
# csrw sstatus,t0
```

- 修改 start_kernel 函数

```
#include "printk.h"
#include "sbi.h"
#include "proc.h"
extern void test();

int start_kernel()
{
  // 调用 schedule 函数，直接触发调度
  schedule();
  test();

  return 0;
}

```

3.6 测试二进制文件能否运行

- 程序运行截图如下



3.7 添加 ELF 支持

- 将 `user/uapp.S` 修改如下，使程序在运行的时候载入文件从 `uapp.bin` 变为 `uapp.elf`

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

.incbin "uapp"
```

- 修改线程初始化代码如下

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

    unsigned long phdr_start = (unsigned long)ehdr + ehdr->e_phoff;
    int phdr_cnt = ehdr->e_phnum;
    pagetable_t pgtbl = (unsigned long *)(((((unsigned long)(task->pgd)) & 0xfffffffffff) << 12) + PA2VA_OFFSET);
    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)
        {
            // do mapping
            // code...
            // 读取载入段虚拟地址
            unsigned long va = phdr->p_vaddr;
            // 计算映射段在文件对应的大小需要多少个 page
            unsigned long filePages = PGROUNDUP(phdr->p_filesz) / PGSIZE;
            // 计算映射段在内存中对应的大小需要多少个 page
            unsigned long memPages = (PGROUNDUP(phdr->p_memsz)) / PGSIZE;
            // 复制代码段到本进程维护的页表中，保证不同进程之间使用的地址空间是隔离的
            unsigned long mappedPage = alloc_pages(memPages);
            for (int i = 0; i < PGROUNDUP(phdr->p_filesz) / 8; i++)
            {
                ((unsigned long *)(mappedPage))[i] = ((unsigned long *)(PGROUNDDOWN(((unsigned long)uapp_start + phdr->p_offset))))[i];
            }
            // 将上述映射的文件长度对应的 page 映射到页表中
            create_mapping(pgtbl, va, (unsigned long)(mappedPage)-PA2VA_OFFSET, phdr->p_filesz, (phdr->p_flags << 1) | USER | VALID);
            // 将多余的内存长度对应的 page 映射到页表中
```

```
            create_mapping(pgtbl, va + PGSIZE * filePages, (unsigned long)(mappedPage) + PGSIZE * filePages - PA2VA_OFFSET, (memPages - filePages) * PGSIZE, (phdr->p_flags << 1) | USER | VALID);
            load_phdr_cnt++;
        }
    }
    // 设置 sepc 为 elf 文件的起始代码
    task->thread.sepc = ehdr->e_entry;
}
void task_init()
{
    idle = (struct task_struct *)kalloc();
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;
    current = idle;
    task[0] = idle;

    for (int i = 1; i <= NR_TASKS - 1; ++i)
    {
        task[i] = (struct task_struct *)kalloc();
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand();
        task[i]->pid = i;
        task[i]->thread.ra = (unsigned long)__dummy;
        task[i]->thread.sp = (unsigned long)task[i] + PGSIZE;
        task[i]->thread_info = (struct thread_info *)(alloc_page());
        task[i]->thread_info->kernel_sp = task[i]->thread.sp;
        task[i]->thread_info->user_sp = (alloc_page());
        // 用户页表初始化
        pagetable_t pgtbl = (unsigned long *)(alloc_page());
        // 拷贝内核页表
        copy(swapper_pg_dir, pgtbl);
        // 映射用户栈
        create_mapping(pgtbl, USER_END - PGSIZE, (unsigned long)(task[i]->thread_info->user_sp - PA2VA_OFFSET), PGSIZE, READABLE | WRITABLE | USER | VALID);
        // set sstatus
        unsigned long sstatus = csr_read(sstatus);
        // set spp bit to 0
        sstatus &= (~(1 << 8));
        // set SPIE bit to 1
        sstatus |= (1 << 5);
```

```
        // set SUM bit to 1
        sstatus |= (1 << 18);
        task[i]->thread.sstatus = sstatus;
        // user mode stack
        task[i]->thread.sscratch = USER_END;

        // printk("SET [PID = %d] COUNTER = %d\n", task[i] -> pid, task[i] -> counter);
        unsigned long satp = csr_read(satp);
        task[i]->pgd = (pagetable_t)(((satp >> 44) << 44) | (((unsigned long)(pgtbl)-PA2VA_OFFSET) >> 12));
        // load uapp.elf
        load_program(task[i]);
    }
    /* YOUR CODE HERE */
    printk("...proc_init done!\n");
}

```

- 完成上述修改之后，每一个用户线程在初始化的时候都会将 `elf` 文件对应的 `load` 段装载到自己维护的页表中，这样实现了不同进程之间的地址空间的隔离，程序也能够运行 `elf` 文件中对应的代码。
- 程序运行输出截图如下。

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsuh
Boot HART Features        : scounteren,mcounteren,mcountinhibit,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
...buddy_init done!
setup_vm_final finish
...proc_init done!
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.4
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.4
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.5
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3
```

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）

程序中使用的用户态线程和内核态线程是 1 对 1 的关系。

4.2 为什么 Phdr 中，`p_filesz` 和 `p_memsz` 是不一样大的?

因为在 elf 文件中的一个 load 段可能有为无需初始化的变量或者数组等数据，这些数据都会在程序装载的时候初始化为 0，如果在 elf 文件中为这些数据预留空间的话，就会造成磁盘空间的浪费。但是这些数据在内存中是需要对应的空间的，所以在许多情况下 `p_memsz` 会稍大于 `p_filesz` 。

在 linuxbase.org 的文档中也有对 load 段 `p_memsz` 稍大于 `p_filesz` 段的说明。

PT_LOAD
The array element specifies a loadable segment, described by `p_filesz` and `p_memsz`. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (`p_memsz`) is larger than the file size (`p_filesz`), the ``extra'' bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the `p_vaddr` member.