Lab 4: RV64 用户态程序

姓名：高铭健

学号：3210102322

日期：2023.12.10

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

- 计算机（Intel Core i5以上，4GB内存以上）系统
- Ubuntu 22.04.2 LTS

3. 背景知识

3.1 用户模式和内核模式

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。

3.2 目标

- 当启动用户态应用程序时，内核将为该应用程序创建一个进程，并提供了专用虚拟地址空间等资源。
    - 每个应用程序的虚拟地址空间是私有的，一个应用程序无法更改属于另一个应用程序的数据。
    - 每个应用程序都是独立运行的，如果一个应用程序崩溃，其他应用程序和操作系统将不会受到影响。
- 用户态应用程序可访问的虚拟地址空间是受限的。
    - 在用户态下，应用程序无法访问内核的虚拟地址，防止其修改关键操作系统数据。
    - 当用户态程序需要访问关键资源的时候，可以通过**系统调用**来完成用户态程序与操作系统之间的互动。

3.3 系统调用约定

系统调用是用户态应用程序请求内核服务的一种方式。在 RISC-V 中，我们使用 `ecall` 指令进行系统调用。当执行这条指令时，处理器会提升特权模式，跳转到异常处理函数，处理这条系统调用。

3.4 sstatus[SUM] PTE[U]

当页表项 PTE[U] 置 0 时，该页表项对应的内存页为内核页，运行在 U-Mode 下的代码*无法*访问。当页表项 PTE[U] 置 1 时，该页表项对应的内存页为用户页，运行在 S-Mode 下的代码*无法*访问。如果想让 S 特权级下的程序能够访问用户页，需要对 sstatus[SUM] 位置 1 。但是无论什么样的情况下，用户页中的指令对于 S-Mode 而言都是**无法执行**的。

3.5 用户态栈与内核态栈

当用户态程序在用户态运行时，其使用的栈为**用户态栈**。当进行系统调用时，陷入内核处理时使用的栈为**内核态栈**。因此，需要区分用户态栈和内核态栈，且需要在异常处理的过程中对栈进行切换。

3.6 ELF 程序

ELF, short for Executable and Linkable Format. 是当今被广泛使用的应用程序格式。例如当我们运行 `gcc <some-name>.c` 后产生的 `a.out` 输出文件的格式就是 ELF。

```
$ cat hello.c
#include <stdio.h>

int main() {
    printf("hello, world\n");
    return 0;
}
$ gcc hello.c
$ file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=dd33139196142abd22542134c20d85c571a78b0c,
for GNU/Linux 3.2.0, not stripped
```

将程序封装成 ELF 格式的意义包括以下几点：

- ELF 文件可以包含将程序正确加载入内存的元数据（metadata）。
- ELF 文件在运行时可以由加载器（loader）将动态链接在程序上的动态链接库（shared library）正确地从硬盘或内存中加载。
- ELF 文件包含的重定位信息可以让该程序继续和其他可重定位文件和库再次链接，构成新的可执行文件。

4. 实验步骤

4.1 准备工程

- 修改 `vmlinux.lds`，将用户态程序 `uapp` 加载至 `.data` 段。添加如下片段：

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

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

        _edata = .;

        . = ALIGN(0x1000);
        _sramdisk = .;
        *(.uapp .uapp*)
        _eramdisk = .;
        . = ALIGN(0x1000);

    } >ramv AT>ram
```

- 修改 `defs.h`，在 `defs.h` **添加** USER space的起始虚拟地址和结尾虚拟地址：

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

- 从 `repo` 同步以下文件和文件夹：

```
.
├── 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，添加 `${MAKE} -C user all`；代码如下所示：

```
export
CROSS_=riscv64-linux-gnu-
GCC=${CROSS_}gcc
LD=${CROSS_}ld
OBJCOPY=${CROSS_}objcopy

ISA=rv64imafd_zifencei
ABI=lp64

INCLUDE = -I $(shell pwd)/include -I $(shell pwd)/arch/riscv/include
CF = -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g
CFLAG = ${CF} ${INCLUDE}

.PHONY:all run debug clean
all: clean
    ${MAKE} -C lib all
    ${MAKE} -C test all
    ${MAKE} -C init all
    ${MAKE} -C arch/riscv all
    ${MAKE} -C user all
    @echo -e '\n'Build Finished OK

TEST:
    ${MAKE} -C lib all
    ${MAKE} -C test test
    ${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

test-run: TEST
    @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

test-debug: TEST
    @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 test clean
    ${MAKE} -C init 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
```

4.2 创建用户态进程

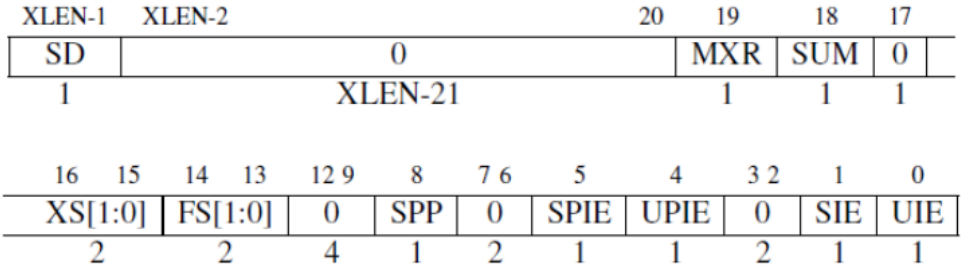
- 创建 4 个用户态进程，修改 `proc.h` 中的 `NR_TASKS`： ：

```
#define NR_TASKS  (1 + 3)
```

- 用户态进程要对 `sepc` `sstatus` `sscratch` 做设置，将其加入 `thread_struct` 中，同时将内核和用户栈指针以及 `satp` 都直接添加到 `task_struct` 方便处理。因为每一个进程都创建一个页表，所以添加 `pagetable_t pgd`（没有使用 `struct thread_info *thread_info`，使之前的 `switch_to` 中变量的偏移量不用修改）：

```

struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];

    uint64 sepc, sstatus, sscratch;
};

struct task_struct {
    struct thread_info *thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
    uint64 pid;      // 线程id
    struct thread_struct thread;
    uint64 satp;
    uint64 kernel_sp;
    uint64 user_sp;
    pagetable_t pgd;
};
```

- 修改 task_init
  - 对于每个进程，初始化在 `thread_struct` 中添加的三个变量：
    - 将 `sepc` 设置为 `USER_START`。
    - 将 `sstatus` 中的 `SPP` 置0（使得 sret 返回至 U-Mode），`SPIE` 置1（sret 之后开启中断），`SUM` 置1（S-Mode 可以访问 User 页面）。



- 将 `sscratch` 设置为 `U-Mode` 的 sp，其值为 `USER_END` （即，用户态栈被放置在 `user space` 的最后一个页面）。
- 直接设置`satp`，将`PPN`清空，并设置为当前进程申请的页表的物理页号

```
//将 sepc 设置为 USER_START
task[i]->thread.sepc = 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）
uint64 sstatus = csr_read(sstatus);
sstatus |= 0x0000000000040020;
sstatus &= 0xffffffffffffeff;
task[i]->thread.sstatus = sstatus;
//将 sscratch 设置为 U-Mode 的 sp，其值为 USER_END （即，用户态栈被放置在 user space 的最后一个页面）
task[i]->thread.sscratch = USER_END;
task[i]->satp = (csr_read(satp) & 0xfffff00000000000) | (((uint64)new_pgtbl - PA2VA_OFFSET) >> 12);
```

- 对于每个进程，创建属于它自己的页表，使用`alloc_page()`函数进行内存申请：

```
//对于每个进程，创建属于它自己的页表
uint64* new_pgtbl = (uint64*)alloc_page();
```

- 为了避免 `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，注意在前面声明的`_sramdisk[]`类型应该是`uint64`，然后使用for循环复制一个页的大小即可。

```
extern uint64 _sramdisk[];
...
for(unsigned long i = 0;i < PGSIZE;i++){
    new_pgtbl[i] = swapper_pg_dir[i];
}
```

- 将 `uapp` 所在的页面映射到每个进行的页表中，在程序运行过程中，有部分数据不在栈上，而在初始化的过程中就已经被分配了空间。所以，二进制文件需要先被 拷贝 到一块某个进程专用的内存之后再进行映射，防止所有的进程共享数据，造成预期外的进程间相互影响。

  使用`alloc_page()`函数申请一片内存，将`_sramdisk`之后的文件复制到这片内存中

然后调用`create_mapping()`函数进行映射，将`USER_START`开始的user指令映射到申请的内存中，大小为1个页，权限是V =1（有效） | R = 1（可读） | W = 1（可写） | X = 1（可执行） | U = 1（说明是用户页）

```
//二进制文件需要先被 拷贝 到一块某个进程专用的内存之后再进行映射，防止所有的进程共享数据
uint64* new_addrress = (uint64*)alloc_page();
for(unsigned long i = 0;i < PGSIZE;i++){
    new_addrress[i]  = _sramdisk[i];
}
//将 uapp 所在的页面映射到每个进行的页表中
uint64 va = USER_START;
uint64 pa = (uint64)new_addrress - PA2VA_OFFSET;
uint64 sz = PGSIZE;
int perm = 0b11111;
printk("task[i]->uapp %|x %|x %|x %d\n",va,pa,sz,perm);
create_mapping(new_pgtbl, va, pa, sz, perm);
```

- 设置用户态栈。对每个用户态进程，其拥有两个栈： 用户态栈和内核态栈；通过`alloc_page`接口申请一个空的页面来作为用户态栈，然后调用`create_mapping()`函数进行映射，将`USER_END - PGSIZE`开始的use页映射到申请的内存中，权限是V =1（有效） | R = 1（可读） | W = 1（可写） | X = 0（用户页中的指令对于 S-Mode 而言是**无法执行**的） | U = 1（说明是用户页）

```
//设置用户态栈,通过 alloc_page 接口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
task[i]->user_sp = alloc_page();
va = USER_END - PGSIZE;
pa = task[i]->user_sp - PA2VA_OFFSET;
sz = PGSIZE;
perm = 0b10111;
printk("task[i]->user %|x %|x %|x %d\n",va,pa,sz,perm);
create_mapping(new_pgtbl, va, pa, sz, perm);
```

- 修改 __switch_to，在切换进程时添加保存进程的`sepc sstatus sscratch satp`，同样LD时获得新进程的这些值

```
__switch_to:
    #加入保存/恢复 sepc sstatus sscratch 以及 切换页表的逻辑
    csrr t1, sepc
    csrr t2, sstatus
    csrr t3, sscratch
    csrr t4, satp
    # save state to prev process
    # YOUR CODE HERE
    addi t0, a0, 40
    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)
    sd t1, 112(t0)
    sd t2, 120(t0)
    sd t3, 128(t0)
    sd t4, 136(t0)
    # restore state from next process
    # YOUR CODE HERE
    addi t0, a1, 40
    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)
    ld t2, 120(t0)
    ld t3, 128(t0)
    ld t4, 136(t0)
    csrw sepc, t1
    csrw sstatus, t2
    csrw sscratch, t3
    csrw satp, t4
    # flush
    sfence.vma zero, zero
    ret
```

- 在切换页表之后，通过 `fence.i` 和 `vma.fence` 来刷新 TLB 和 ICache：

```
...
# flush tlb
sfence.vma zero, zero
# flush icache
fence.i
```

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

- 修改 `__dummy`，交换`thread_struct.sp`和保存在`thread_struct.sscratch`的 `U-Mode sp`来完成进程栈的切换，同时将`sepc`设为0，也就是要返回`user space`的开头执行指令：

```
__dummy:
    # YOUR CODE HERE
    #交换对应的寄存器的值
    csrr t1, sscratch
    mv t2, sp
    mv sp, t1
    csrw sscratch, t2
    # sepc置0
    addi t0, x0, 0
    csrw sepc, t0
    sret #从中断中返回
```

- 修改 `_trap` ,在 `_trap` 的首尾进行内核和用户的切换，但是如果已经在内核态那么显然不用切换，判断当前的`sscratch`是否 为 0，是的话就说明已经在内核态，不用切换

```
_traps:
    # YOUR CODE HERE
    # -----------
        #在 _trap 的首尾我们都需要做类似的操作(判断是否在内核态)
        csrr t0, sscratch
        addi t1, x0, 0
        beq t0, t1, Return
        #交换对应的寄存器的值
        csrr t1, sscratch
        mv t2, sp
        mv sp, t1
        csrw sscratch, t2
        # 1. save 32 registers and sepc to stack
    Next:
        addi sp, sp, -280
        #存入32个寄存器
        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) #寄存器a0
        sd x11, 88(sp) #寄存器a1
        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)
        #存sepc
        csrr t0, sepc
        sd t0, 256(sp)
        csrr t0, sstatus
        sd t0, 264(sp)
        csrr t0, sscratch
        sd t0, 272(sp)


    # -----------

        # 2. call trap_handler
        csrr a0, scause
        csrr a1, sepc
        mv a2, sp #寄存器a2
        jal trap_handler

    # -----------

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

        ld t0, 272(sp)
```

```
        csrw sscratch, t0
        ld t0, 264(sp)
        csrw sstatus, t0
        ld t0, 256(sp)
        csrw sepc, t0
        ld x0, 0(sp)
        ld x1, 8(sp)
        ld x2, 16(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) #寄存器a0
        ld x11, 88(sp) #寄存器a1
        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)

        addi sp, sp, 280
    # -----------
        #在 _trap 的首尾我们都需要做类似的操作(判断是否在内核态)
        csrr t0, sscratch
        addi t1, x0, 0
        beq t0, t1, Return
        #交换对应的寄存器的值
        csrr t1, sscratch
        mv t2, sp
        mv sp, t1
        csrw sscratch, t2
    Return:
        # 4. return from trap
        sret
```

- 在 `trap_handler` 里面进行捕获用户态的 `ecall`。修改 `trap_handler`，传入 `pt_regs`，其中包括了全部的寄存器，同时定义需要SYS_WRITE和SYS_PID的对应寄存器的值。

  首先根据scause的值是否为8判断是否进行 `ecall` 处理，然后根据 `a7` 寄存器的值判断需要SYS_WRITE还是SYS_PID，如果是64则调用 `sys_write()` 函数，传入需要输出的字符串a1和大小a2；如果是172则调用 `sys_getpid()` 函数，获得当前pid并赋给a0（修改 `regs→reg[10]`）

  最后将 `sepc + 4` 返回异常指令的下一条指令

```
#define SYS_WRITE 64
#define SYS_PID 172
struct pt_regs
{
    uint64 reg[32];
    uint64 sepc;
    uint64 sstatus;
    uint64 sscatch;
};
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs
*regs) {
    // 通过 `scause` 判断trap类型
    // 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
    if(scause & 0x8000000000000000){//trap 类型为interrupt
        if(scause == 0x8000000000000005){//timer interrupt
//            printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }
    }
    else if(scause == 8){
        if(regs->reg[17] == SYS_WRITE){
            char *buffer = (char *)(regs->reg[11]);
            int cnt = sys_write(1, buffer, regs->reg[12]);
            regs->sepc += 4;
        }
        else if(regs->reg[17] == SYS_PID){
            uint64 pid = sys_getpid();
            regs->sepc += 4;
            regs->reg[10] = pid;
//            printk("pid: %d", pid);
        }
        else{
//            printk("error!\n");
        }
    }
}
```

4.4 添加系统调用

- 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（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。(具体见 user/printf.c)

  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。

- 增加 `syscall.c` `syscall.h` 文件， `sys_write()` 简单输出传入的字符串， `sys_getpid()` 需要先使用当前进程 `task_struct *current` 直接返回其pid

```c
//
// syscall.h
//
#include "proc.h"
#include "sbi.h"
#include "printk.h"


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

unsigned long sys_getpid();
```

```c
//
// syscall.c
//
#include "syscall.h"
extern struct task_struct *current;
int sys_write(unsigned int fd, const char *buf, size_t count){
    int write_size;
    for(int i = 0;i < count ;i++){
        printk("%c", buf[i]);
        write_size++;
    }
    return write_size;
}

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

4.5 修改 head.S 以及 start_kernel

- 在 start_kernel 中调用 schedule() ，放在 test() 之前：

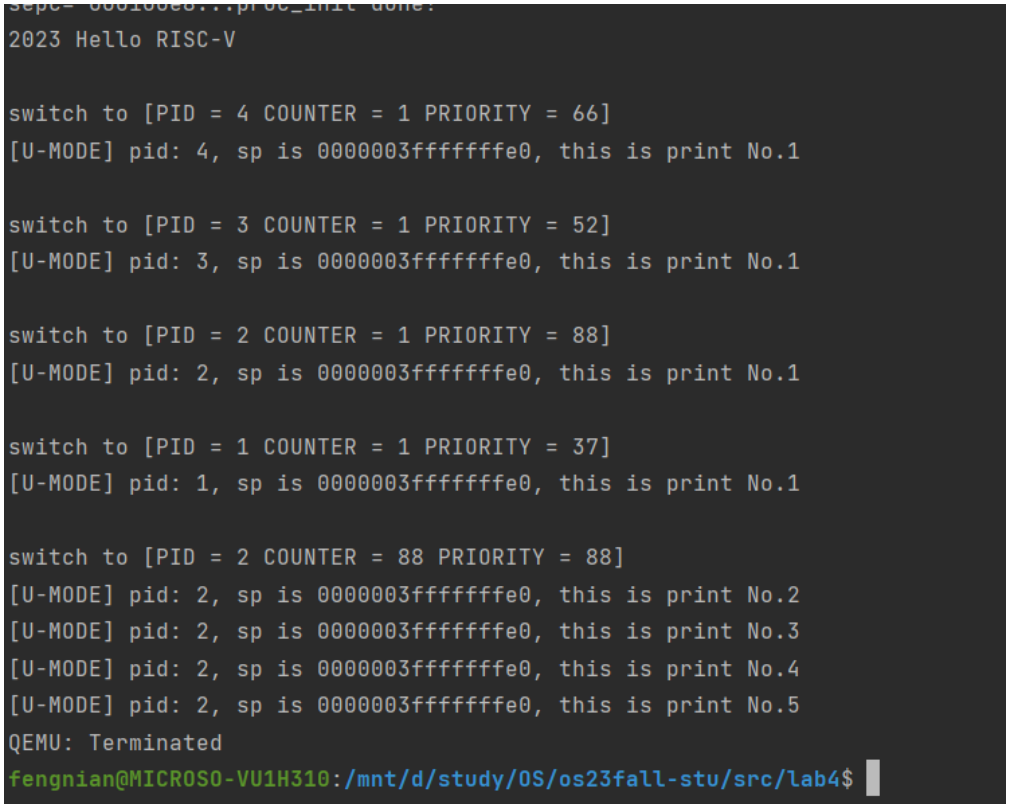
```c
#include "printk.h"
#include "sbi.h"
extern void test();
extern void schedule();
int start_kernel() {
    printk("2023");
    printk(" Hello RISC-V\n");
    schedule();
    test(); // DO NOT DELETE !!!
    return 0;
}

```

- 将 head.S 中 enable interrupt sstatus.SIE 逻辑删去，确保 schedule 过程不受中断影响：

```
        # set sstatus[SIE] = 1
        #addi t0, x0, 1
        #slli t0, t0, 1
        #csrr t1, sstatus
        #or t2, t0, t1
        #csrw sstatus, t2
```

4.6 测试

- 输出结果如下



这里将第一轮调度的时间片都设为1，打印结束后进入第二轮，根据打印出的No.*可以看出各进程是分离的

4.7 添加 ELF 支持

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

```
.section .uapp

.incbin "uapp"

```

- 将程序 load 进入内存。

  先将内核页表复制到每个进程的页表中（与之前一样）

  将 segment 的内容从ELF文件中读入 `[p_vaddr, p_vaddr + p_memsz)` 内存区间，先调用 `alloc_pages(page_cnt)` 申请足够页数的内存空间，然后将 `_sramdisk` - `_eramdisk` 这一段文件复制到刚才申请的内存（注意这里使用的是 `char _sramdisk[]`），之后调用 `create_mapping()` 进行映射，权限与之前相同

  用户态栈的设置和之前相同

```
extern char _sramdisk[];
static uint64_t load_program(struct task_struct* task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)_sramdisk;

    uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;
    int phdr_cnt = ehdr->e_phnum;
    uint64* new_pgtbl = (uint64*)alloc_page();
    //将内核页表 （ swapper_pg_dir ） 复制到每个进程的页表中
    for(unsigned long i = 0;i < PGSIZE;i++){
        new_pgtbl[i] = swapper_pg_dir[i];
    }

    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
            int page_cnt;
            if(phdr->p_memsz % PGSIZE == 0){
                page_cnt = phdr->p_memsz / PGSIZE;
            }
            else{
                page_cnt = phdr->p_memsz / PGSIZE + 1;
            }
            uint64* new_space = (uint64 *)alloc_pages(page_cnt);
            uint64* src = (uint64 *)(_sramdisk);
            memcopy(new_space , src, phdr->p_memsz);
            // do mapping
```

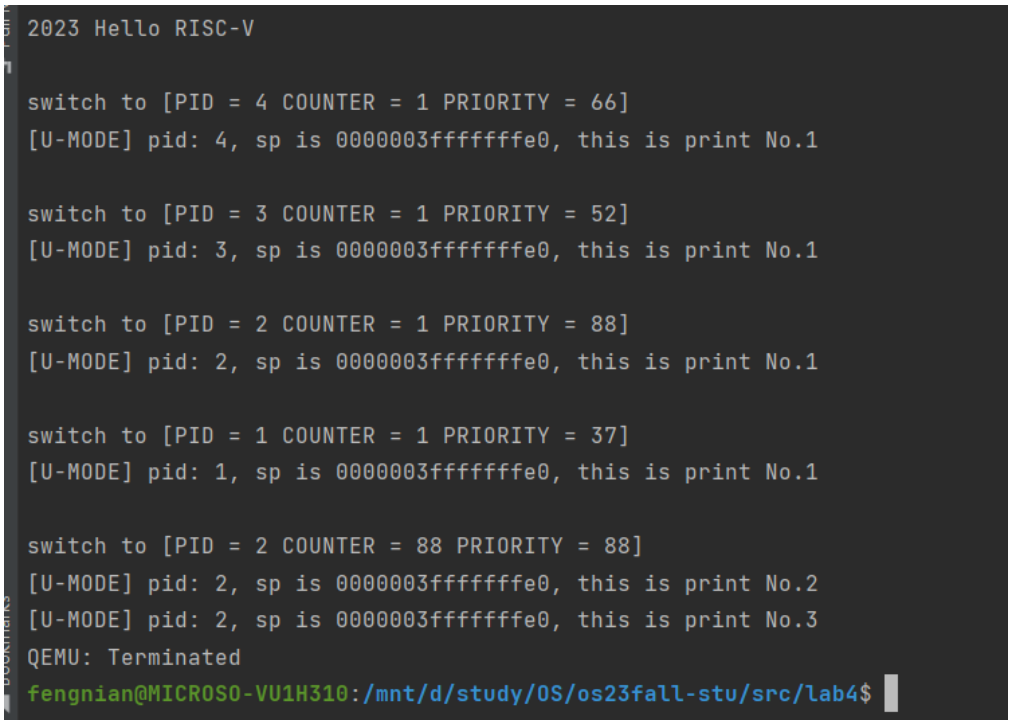
```
            uint64 va = phdr->p_vaddr;
            uint64 pa = (uint64)new_space - PA2VA_OFFSET;
            create_mapping(new_pgtbl, va, pa, phdr->p_memsz,31);
        }
    }
    //设置用户态栈,通过 alloc_page 接口申请一个空的页面来作为用户态栈，并映射到进程的页表中
    task->user_sp = alloc_page();
    uint64 va = USER_END - PGSIZE;
    uint64 pa = task->user_sp - PA2VA_OFFSET;
    uint64 sz = PGSIZE;
    uint64 perm = 0b10111;
    create_mapping(new_pgtbl, va, pa, sz, perm);
    task->satp = (csr_read(satp) & 0xfffff00000000000) | (((uint64)new_pgtbl - PA2VA_OFFSET) >> 12);
    //将 sepc 设置为 ehdr->e_entry
    task->thread.sepc = ehdr->e_entry;
    printk("sepc= %|x",ehdr->e_entry);
    //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）
    uint64 sstatus = csr_read(sstatus);
    sstatus |= 0x0000000000040020;
    sstatus &= 0xffffffffffffeff;
    task->thread.sstatus = sstatus;
    //将 sscratch 设置为 U-Mode 的 sp，其值为 USER_END （即，用户态栈被放置在 user space 的最后一个页面）
    task->thread.sscratch = USER_END;
}
```

- 将之前的`task_init`替换，直接调用`load_program(task[i])`：

```
void task_init() {
    test_init(NR_TASKS);
    // 1. 调用 kalloc() 为 idle 分配一个物理页
    idle = kalloc();
    // 2. 设置 state 为 TASK_RUNNING;
    idle->state = TASK_RUNNING;
    // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
    idle->counter = 0;
    idle->priority = 0;
    // 4. 设置 idle 的 pid 为 0
    idle->pid = 0;
    // 5. 将 current 和 task[0] 指向 idle
    current = idle;
    task[0] = idle;
    for(int i = 1 ; i < NR_TASKS ; i++){
        task[i]  = kalloc();
        task[i]->pid = i;
        task[i]->counter = task_test_counter[i];
        task[i]->priority = task_test_priority[i];
        task[i]->state = TASK_RUNNING;
```

```
        uint64 ra_address = &__dummy;
        uint64 sp_address = (uint64)task[i] + PGSIZE;
        task[i]->thread.ra = ra_address;
        task[i]->thread.sp = sp_address;
        task[i]->kernel_sp = sp_address;
        load_program(task[i]);
    }
}

```

- 添加 ELF 支持后的测试结果：



  与之前的结果相同

5. 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）

   答：使用的是一对一的用户态线程和内核态线程

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

   答：`p_filesz` 是段在可文件中空间大小，只包含了文件所占的实际字节数，`p_memsz` 是指段在内存中占的大小，它可能包含了未初始化的数据或者初始化为0的数据（比如.bss段，如果一个数据未被初始化，就不需要为其分配空间，所以 .bss 并不占用可执行文件的大小，仅仅记录需要用多少空间来存储这些未初始化的数据，而不分配实际空间），这些数据存储在磁盘上会浪费内存，只有在 ELF 文件加载到内存中后才会占用空间，因此占用了额外的内存，这就导致`p_filesz` 的大小一般小于等于 `p_memsz`

3. 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?

   答：

1. 因为每个进程初始化的时候都申请了一块物理地址做栈，然后把虚拟地址映射过去，虽然多个进程的栈虚拟地址是相同的，但实际映射到物理地址上是不同的
2. 应该没有办法，这个物理地址只有内核态能看到