实验四-RV64 用户模式

姓名:王晶晶 学号:3200104880 学院:计算机科学与技术学院 课程名称：计算机系统Ⅲ

实验时间：2022.5.13 实验地点：紫金港东4-509 指导老师：周亚金

一、实验目的和要求

- 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。

二、实验原理

2.1 用户模式

RISC-V特权模式共有三种：U (user) 模式、S (supervisor) 模式和 M (machine) 模式。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

M模式是对硬件操作的抽象，有最高级别的权限。S模式介于M模式和U模式之间，在操作系统中对应于内核态。当用户需要内核资源时，向内核申请，并切换到内核态进行处理。U模式用于执行用户程序，在操作系统中对应于用户态，有最低级别的权限。

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。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。处理器根据处理器上运行的代码类型在两种模式之间切换。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。

当启动用户模式应用程序时，内核将为该应用程序创建一个进程，为应用程序提供了专用虚拟地址空间等资源。因为应用程序的虚拟地址空间是私有的，所以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，同时，用户模式应用程序可访问的虚拟地址空间也受到限制，在用户模式下无法访问操作系统的虚拟地址，可防止应用程序修改关键操作系统数据。

2.2 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。

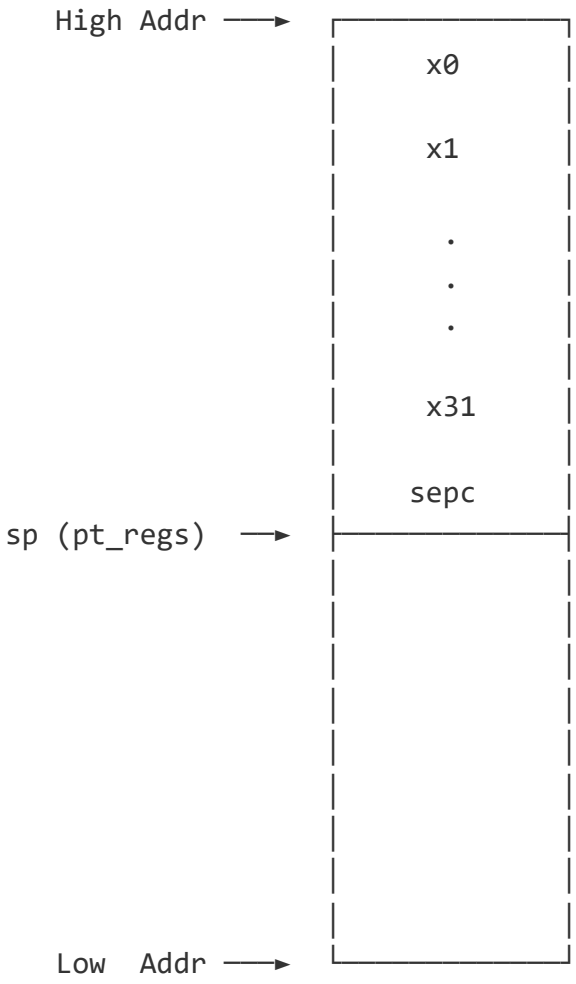
系统调用参数使用a0 - a5，系统调用号使用a7，系统调用的返回值会被保存到a0, a1中。

本次实验在ecall中需要用到的参数是a0,a1,a2。

在用户态进程调用syscall，调用 `sys_write` 需要用到的参数是a0,a1,a2,调用 `sys_getpid` ，返回值存入a0。

func	a0	a1	a2
trap_handler	scause	sepc	*regs
sys_write	fd	*buf	count
sys_getpid	return pid		

其中*regs的布局如下。



2.3 用户栈和内核栈

用户态程序与内核并不共用栈，因此内核的实现需要区分用户态栈和内核态栈，在异常处理的过程中需要对栈进行切换。
从用户态进入内核态时，即发生中断或异常时，需要切换用户栈为内核栈，在用户态下，使用sscratch来直接保存指向内核栈的指针，并在切换到内核态栈时，将当前用户栈的栈指针存入sscratch中，以便后续切换时再次进行寄存器值的交换。

```
csrrw sp, sscratch, sp
```

三、实验代码实现

3.1 创建用户态进程，并对其进行初始化

先为每个用户态进程分配一个用户栈的页。

```
//allocate to store user_stack
unsigned long * user_stack=kalloc();
```

为了避免在用户模式和特权模式切换时切换页表，为每一个进程复制内核页表到进程页表。

```
//allocate to store kernel page
unsigned long * kernel_pg=kalloc();
task[i]->pgd=(unsigned long)kernel_pg-PA2VA_OFFSET;
for(int i=0;i<512;i++)
kernel_pg[i]=swapper_pg_dir[i];
```

建立用户栈和用户页表的虚拟地址映射，同时建立用户进程和虚拟地址的地址映射。

```
create_mapping(kernel_pg,USER_END-PGSIZE,(unsigned long)user_stack-PA2VA_OFFSET,PGSIZ
create_mapping(kernel_pg,USER_START,(unsigned long)uapp_start-PA2VA_OFFSET,(unsigned
```

初始化sepc, sstatus, sscratch寄存器。

```
task[i]->thread.sstatus=csr_read(sstatus);
task[i]->thread.sstatus|=0x00040020;
csr_write(sstatus,task[i]->thread.sstatus);
task[i]->thread.sepc=USER_START;
task[i]->thread.sscratch=USER_END;
```

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

区分当前进程为用户态还是特权态，特权态无需进行栈切换。

```
_traps:
    # YOUR CODE HERE
    # -----------
    # judge user or kernel
    csrr t0,sscratch
    beqz t0,is_kernel
    csrrw sp,sscratch,sp
    .
    .
    .
    csrr t0,sscratch
    beqz t0,final #kernel
    csrrw sp,sscratch,sp

    # -----------
final:
        # 4. return from trap
    sret
    # -----------
```

修改trap_handler传入的参数。

```
csrr a0,scause
csrr a1,sepc
addi a2,sp,8 # place to store sepc and then the reversed order of 32 registers
call trap_handler
```

__switch_to时进行页表的切换

```
# change page table
addi t0,a1,40
ld t1,136(t0)  #PA
srli t1,t1,12
li t2,(8<<60) #set the mode SV39
or t1,t1,t2
csrw satp,t1
sfence.vma zero,zero #open the vitual mapping
```

3.3 添加系统调用

```
extern struct task_struct* current;
int sys_getpid(){
    return current->pid;
}
unsigned int sys_write(unsigned int fd, const char* buf, unsigned int count){
    if(fd==1){
        printk("%s",buf);
        return count;
    }
}
```

四、实验结果分析

在启动OpenSBI后，kernel会先初始化内存，输出"mm_init done!"表示对于从kernel结束地址开始的虚拟地址都已经被初始化。

完成后再开启虚拟映射并初始化页表，最后初始化四个用户态进程，并建立用户进程的页表映射。输出"proc_init done!"。

```
...mm_init done!
third level pa: 000000002008040b
text section: ffffffe000200000~ffffffe000202000
third level pa: 0000000020080803
rodata section: ffffffe000202000~ffffffe000203000
third level pa: 0000000021fffc07
third level pa: 0000000021fef417
third level pa: 000000002008101f
third level pa: 0000000021fed817
third level pa: 000000002008101f
third level pa: 0000000021febc17
third level pa: 000000002008101f
third level pa: 0000000021fea017
third level pa: 000000002008101f
...proc_init done!
```

在执行完head.S部分的启动代码后，转入main()执行。首先输出"Hello RISC-V"，然后直接进行线程的调度。对于线程，在main函数中调用getpid和syswrite系统调用，向屏幕打印线程的pid和sp信息。

这里由于每个进程在初始化时给的时间片是随机的，故在进程的main函数中while循环输出进程信息的时候，执行的次数不相同。

```
[S-MODE] Hello RISC-V
SET [PID = 1 PRIORITY = 1 COUNTER = 10]
SET [PID = 2 PRIORITY = 4 COUNTER = 10]
SET [PID = 3 PRIORITY = 10 COUNTER = 5]
SET [PID = 4 PRIORITY = 4 COUNTER = 2]
switch to [PID = 4 PRIORITY = 4 COUNTER = 2]
[U-MODE] pid: 4, sp is 0000003fffffffe0
switch to [PID = 3 PRIORITY = 10 COUNTER = 5]
[U-MODE] pid: 3, sp is 0000003fffffffe0
[U-MODE] pid: 3, sp is 0000003fffffffe0
switch to [PID = 1 PRIORITY = 1 COUNTER = 10]
[U-MODE] pid: 1, sp is 0000003fffffffe0
[U-MODE] pid: 1, sp is 0000003fffffffe0
[U-MODE] pid: 1, sp is 0000003fffffffe0
switch to [PID = 2 PRIORITY = 4 COUNTER = 10]
[U-MODE] pid: 2, sp is 0000003fffffffe0
[U-MODE] pid: 2, sp is 0000003fffffffe0
[U-MODE] pid: 2, sp is 0000003fffffffe0
```

五、实验中遇到的问题及解决方法

- 线程初始化不对

```
task[i]->thread.sstatus=csr_read(sstatus);
task[i]->thread.sstatus|=0x00040020;
//task[i]->thread.sstatus=(csr_read(sstatus))|0x00040020;
csr_write(sstatus,task[i]->thread.sstatus);
task[i]->thread.sepc=USER_START;
task[i]->thread.sscratch=USER_END;
```

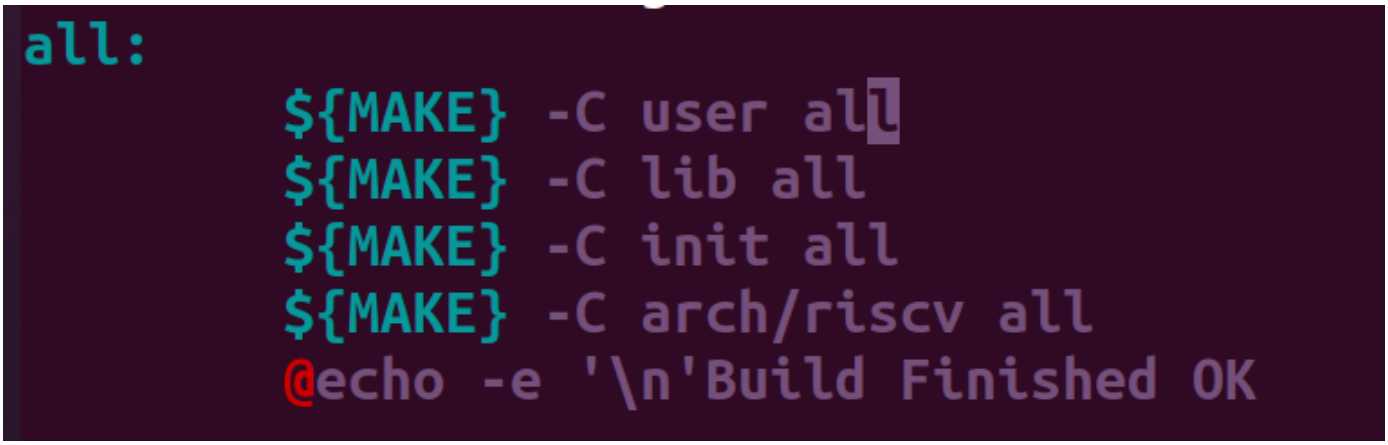
  被注释掉的写法不正确，不能把csr_read和别的操作混合在一起。

  猜测是由于内联汇编宏定义展开的问题。

  一定要先进行sstatus的读，存入一个地址后再取出进行位运算，最后写入sstatus寄存器。

  这个bug找了好久。

- Makefile



一开始把 `${MAKE} -C user all` 放在最后，编译后发现报错无法找到什么东西，后来把user all放在前面，make就过了。但现在尝试复现这个bug，发现放在哪里都不会报错了，不知道怎么回事。

六、思考题与心得体会

1. 系统调用的返回参数放置在a0中，为什么不可以直接修改寄存器， 而应该修改参数 regs 中保存的内容?

- 因为执行系统调用后，回到trap的入口处，a0是从栈中恢复出来的，如果将返回值直接存入a0，从栈上ld出来的数据并没有改变，a0还是进入中断前的值，而不是执行系统调用后重新填入的返回值，故根本上是要将存在栈上的a0变成系统调用的返回值，才能保证ld出的a0是正确的值。

```
    csrr a0,scause
    csrr a1,sepc
    addi a2,sp,8 # place to store
    call trap_handler
       # 3. restore sepc and 32 re
    ld t0,0(sp)
    csrw sstatus,t0
    ld t0,8(sp)
    csrw sepc,t0
    ld t6,16(sp)
    ld t5,24(sp)
    ld t4,32(sp)
    ld t3,40(sp)
    ld s11,48(sp)
    ld s10,56(sp)
    ld s9,64(sp)
    ld s8,72(sp)
    ld s7,80(sp)
    ld s6,88(sp)
    ld s5,96(sp)
    ld s4,104(sp)
    ld s3,112(sp)
    ld s2,120(sp)
    ld a7,128(sp)
    ld a6,136(sp)
    ld a5,144(sp)
    ld a4,152(sp)
```

```
ld a3,160(sp)
ld a2,168(sp)
ld a1,176(sp)
ld a0,184(sp)
```

2. 针对系统调用这一类异常，为什么需要手动将sepc+4?

- 因为在异常处理时，sepc存入的是发生异常的当前指令，而不是异常之后的第一条指令。这点不同于RISC-V的控制转移指令，一般会将下一条指令存入ra。故而发生异常时，需要将sepc+4才能在sret时返回执行发生异常后的下一条指令，转入正确的控制流。

3. 为什么要将 head.S 中 enable interrupt sstatus.SIE 逻辑注释，以确保schedule过程不受中断影响?

- 若将sie位置1，则打开了时钟中断，需要等待一个时间片后进行进程的调度。

```
...proc_init done!
[S-MODE] Hello RISC-V
[S] Supervisor Mode Timer Interrupt
SET [PID = 1 PRIORITY = 1 COUNTER = 10]
SET [PID = 2 PRIORITY = 4 COUNTER = 10]
SET [PID = 3 PRIORITY = 10 COUNTER = 5]
SET [PID = 4 PRIORITY = 4 COUNTER = 2]
switch to [PID = 4 PRIORITY = 4 COUNTER = 2]
[U-MODE] pid: 4, sp is 0000003fffffffe0
[S] Supervisor Mode Timer Interrupt
[S] Supervisor Mode Timer Interrupt
switch to [PID = 3 PRIORITY = 10 COUNTER = 5]
```

若改成在main中手动开启第一次schedule()调度，同时在start中就开启时钟中断，则考虑代码执行时间较长或时钟中断时间果断，则可能在schedule的过程中发生时钟中断执行schedule则两者发生冲突，第一次schedule会被影响。

```c
void trap_handler(unsigned long scause, unsigned long sepc,struct pt_regs *regs) {
    unsigned long x=scause,y=sepc,xi,yi;
    xi=0x8000000000000000;
    if(x>=xi){
        xi=0x8000000000000005;
        if(x==xi){
            //a supervisor time interrupt
            //printk("kernel is running!\n");
            //printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }
        else{
            //other interrupts
            ;
        }
    }
```