Lab2: 时钟中断处理

1.实验目的

- 学习 RISC-V 的异常处理相关寄存器与指令，完成对异常处理的初始化。
- 理解 CPU 上下文切换机制，并正确实现上下文切换功能。
- 编写异常处理函数，完成对特定异常的处理。
- 调用OpenSBI提供的接口，完成对时钟中断事件的设置。

2.实验环境

Docker in Lab0

3.实验步骤

3.1 环境搭配

在本次实验中，我实现了之前没有实现的将docker内的容器向外对应的操作。在这个操作中，需要重新对编译器'gcc-riscv64-linux-gnu'进行安装。这里的安装语句也附在这里的命令之中。

```
docker start oslab22
docker exec -it oslab22 /bin/bash
export RISCV=/opt/riscv
export PATH=$PATH:$RISCV/bin
cd /home/oslab/os22fall-stu/src
apt-get install gcc-riscv64-linux-gnu
```

3.2 修改vmlinux.lds

这一块主要是要加入.text.init和.text.entry的模块

```
/* 目标架构 */
OUTPUT_ARCH( "riscv" )

/* 程序入口 */
ENTRY( _start )

/* kernel代码起始位置 */
BASE_ADDR = 0x80200000;

SECTIONS
{
    /* . 代表当前地址 */
    . = BASE_ADDR;

    /* 记录kernel代码的起始地址 */
    _skernel = .;

    /* ALIGN(0x1000) 表示4KB对齐 */
```

```
    /* _stext, _etext 分别记录了text段的起始与结束地址 */
    .text : ALIGN(0x1000){
        _stext = .;

        *(.text.init)
        /* 加入.text.init*/
        *(.text.entry)
        /* 实现中断处理逻辑会放置在.text.entry*/
        *(.text .text.*)

        _etext = .;
    }

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

        *(.rodata .rodata.*)

        _erodata = .;
    }

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

        *(.data .data.*)

        _edata = .;
    }

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

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

        _ebss = .;
    }

    /* 记录kernel代码的结束地址 */
    _ekernel = .;
}
```

3.3 修改head.S

这里要实现目标功能四条。

1.设置 stvec， 将 _traps (_trap 在 4.3 中实现) 所表示的地址写入 stvec， 这里我们采用 Direct 模式，而 _traps 则是中断处 理入口函数的基地址。（请仔细阅读RISC-V Privileged Spec中对 stvec 格式的介绍）

2.开启时钟中断，将 sie[STIE] 置 1，**对此中断进行处理。**具体寄存器图如下：

SXLEN-1	10	9	8	7	6	5	4	3	2	1	0
WPRI		SEIE	UEIE	**WPRI**		STIE	UTIE	**WPRI**		SSIE	USIE
SXLEN-10		1	1	2		1	1	2		1	1

Figure 4.5: Supervisor interrupt-enable register (`sie`).

3.设置第一次时钟中断，参考后文 4.5小节中介绍 clock_set_next_event() 的汇编逻辑实现。

4.开启 S 态下的中断响应， 将 sstatus[SIE] 置 1，**响应异常**。具体寄存器图如下：

MXLEN-1	MXLEN-2	36	35	34	33	32	31	23	22	21	20	19	18	17
SD	**WPRI**		SXL[1:0]		UXL[1:0]		**WPRI**		TSR	TW	TVM	MXR	SUM	MPRV
1	MXLEN-37		2		2		9		1	1	1	1	1	1

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XS[1:0]		FS[1:0]		MPP[1:0]		**WPRI**		SPP	MPIE	**WPRI**	SPIE	UPIE	MIE	**WPRI**	SIE	UIE
2		2		2		2		1	1	1	1	1	1	1	1	1

```
.extern start_kernel
    .section .text.init#将_start放到.text.init里面
    .globl _start
    .align 2
_start:
    la sp, boot_stack_top
    # set stvec = _trap
    la t0, _trap
    #把_trap放到t0中
    csrw stvec, t0
    #用指令把其写进去
    # enable timer interrupt sie.STIE = 1
    csrr t0, sie
    ori t0, t0, 0x20#和立即数做or，这里是or二进制的100000
    csrw sie, t0 #再写回sie
    # set first time interrupt
    rdtime t0#得到mtime的值
    li t1, 10000000#单纯赋值
    add a0, t0, t1
    add a6, zero, zero
    add a7, zero, zero
    ecall
    # enable interrupt sstatus.SIE = 1
    csrr t0, sstatus#单纯借用一下寄存器t0用来改变值
    ori t0, t0, 0x2#和立即数做or，这里是or二进制的0x10
    csrw sstatus, t0

    j start_kernel

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096
    .globl boot_stack_top
boot_stack_top
```

3.4 实现entry.S

这里主要有五个任务要实现：

1.在 arch/riscv/kernel/ 目录下添加 entry.S 文件。

2. 保存CPU的寄存器（上下文）到内存中（栈上）。

寄存器	ABI名称	说明
x0	zero	0值寄存器，硬编码为0，写入数据忽略，读取永远为0
x1	ra	用于返回地址
x2	sp	用于栈指针
x3	gp	用于通用指针
x4	tp	用于线程指针
x5	t0	用于存放临时数据或者备用链接寄存器
x6~x7	t1~t2	用于存放临时数据寄存器
x8	s0/fp	需要保存的寄存器或者帧指针寄存器
x9	s1	需要保存的寄存器
x10~x11	a0~a1	函数传递参数寄存器或者函数返回值寄存器
x12~x17	a2~a7	函数传递参数寄存器
x18~x27	s2~s11	需要保存的寄存器
x28~x31	t3~t6	用于存放临时数据寄存器

2. 将 scause 和 sepc 中的值传入异常处理函数 trap_handler (trap_handler 在 4.4 中介绍)，我们将会在 trap_handler 中实 现对异常的处理。
3. 在完成对异常的处理之后， 我们从内存中（栈上）恢复CPU的寄存器（上下文）。
4. 从 trap 中返回。

```
.equ reg_size, 0x8

    .section .text.entry
    .align 2
    .globl _trap
_trap:
    # save 31 register (x0 hardwired zero) and sepc
    csrw sscratch, sp #把sp的值放到sscratch里

    addi sp, sp, -32*reg_size

    sd x1, 0*reg_size(sp)
    #这里是空出了一个x2，用于存取栈指针
    sd x3, 2*reg_size(sp)
    sd x4, 3*reg_size(sp)
    sd x5, 4*reg_size(sp)
    sd x6, 5*reg_size(sp)
    sd x7, 6*reg_size(sp)
    sd x8, 7*reg_size(sp)
    sd x9, 8*reg_size(sp)
    sd x10, 9*reg_size(sp)
    sd x11, 10*reg_size(sp)
    sd x12, 11*reg_size(sp)
    sd x13, 12*reg_size(sp)
    sd x14, 13*reg_size(sp)
    sd x15, 14*reg_size(sp)
    sd x16, 15*reg_size(sp)
    sd x17, 16*reg_size(sp)
    sd x18, 17*reg_size(sp)
```

```
    sd x19, 18*reg_size(sp)
    sd x20, 19*reg_size(sp)
    sd x21, 20*reg_size(sp)
    sd x22, 21*reg_size(sp)
    sd x23, 22*reg_size(sp)
    sd x24, 23*reg_size(sp)
    sd x25, 24*reg_size(sp)
    sd x26, 25*reg_size(sp)
    sd x27, 26*reg_size(sp)
    sd x28, 27*reg_size(sp)
    sd x29, 28*reg_size(sp)
    sd x30, 29*reg_size(sp)
    sd x31, 30*reg_size(sp)

    csrr t0, sepc #把sepc的值放到t0之中
    #记录触发异常的那条指令的地址
    sd t0, 31*reg_size(sp)#t0也放到栈里

# CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits,
then writes it to integer register rd.
# The initial value in rs1 is written to the CSR. If rd=x0, then the instruction
shall not read the CSR and
# shall not cause any of the side effects that might occur on a CSR read.
    csrrw t0, sscratch, x0#sscratch就设0了
    sd t0, 1*reg_size(sp)#把原sp的值放到了栈里

    csrr a0, scause#发生原因
    csrr a1, sepc#地址
    mv a2, sp#栈

    call trap_handler

    # load sepc and 31 register (x2(sp) should be loaded last)
    ld t0, 31*reg_size(sp)#sepc的值
    csrw sepc, t0#再把sepc放回去

    ld x1, 0*reg_size(sp)
    ld x2, 1*reg_size(sp)
    ld x3, 2*reg_size(sp)
    ld x4, 3*reg_size(sp)
    ld x5, 4*reg_size(sp)
    ld x6, 5*reg_size(sp)
    ld x7, 6*reg_size(sp)
    ld x8, 7*reg_size(sp)
    ld x9, 8*reg_size(sp)
    ld x10, 9*reg_size(sp)
    ld x11, 10*reg_size(sp)
    ld x12, 11*reg_size(sp)
    ld x13, 12*reg_size(sp)
    ld x14, 13*reg_size(sp)
    ld x15, 14*reg_size(sp)
    ld x16, 15*reg_size(sp)
    ld x17, 16*reg_size(sp)
    ld x18, 17*reg_size(sp)
    ld x19, 18*reg_size(sp)
    ld x20, 19*reg_size(sp)
    ld x21, 20*reg_size(sp)
    ld x22, 21*reg_size(sp)
```

```
    ld x23, 22*reg_size(sp)
    ld x24, 23*reg_size(sp)
    ld x25, 24*reg_size(sp)
    ld x26, 25*reg_size(sp)
    ld x27, 26*reg_size(sp)
    ld x28, 27*reg_size(sp)
    ld x29, 28*reg_size(sp)
    ld x30, 29*reg_size(sp)
    ld x31, 30*reg_size(sp)
    #完成了全体的返回原设置
    sret
```

3.5 实现trap.c

这里的目的是实现异常处理函数 trap_handler() , 其接收的两个参数分别是 scause 和 sepc 两个寄存器中的值

```
#include "trap.h"
#include "types.h"
#include "clock.h"
#include "printk.h"

void handler_interrupt(uint64 scause, uint64 sepc, uint64 regs) {
    //通过scause判断trap类型
    switch (scause & ~TRAP_MASK) {
        //判断是否是timer interrupt
        case STI:
            printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();//设置下一次时钟中断
            break;
        default:
            break;
    }
}

void handler_exception(uint64 scause, uint64 sepc, uint64 regs) {
}

void trap_handler(uint64 scause, uint64 sepc, uint64 regs) {
    //判断这里是interrupt还是exception
    if (scause & TRAP_MASK) // #define TRAP_MASK (1UL << 63)
        handler_interrupt(scause, sepc, regs);
    else
        handler_exception(scause, sepc, regs);
}
```

3.6 实现clock.c

1. 在arch/riscv/kernel/目录下添加clock.c文件。
2. 在clock.c中实现get_cycles(): 使用rdtime汇编指令获得当前time寄存器中的值。

   在clock.c中实现clock_set_next_event(): 调用sbi_ecall，设置下一个时钟中断事件

```
#include "clock.h"
#include "sbi.h"
```

```
// QEMU中时钟的频率是10MHz, 也就是1秒钟相当于10000000个时钟周期。
unsigned long TIMECLOCK = 10000000;

unsigned long get_cycles() {
    // 使用 rdtime 编写内联汇编，获取 time 寄存器中 (也就是mtime 寄存器 )的值并返回
    unsigned long n;
    asm volatile("rdtime %0" : "=r"(n));
    return n;
}

void clock_set_next_event() {
    // 下一次 时钟中断 的时间点
    unsigned long next = get_cycles() + TIMECLOCK;

    // 使用 sbi_ecall 来完成对下一次时钟中断的设置
    sbi_ecall(SBI_SETTIMER, 0, next, 0, 0, 0, 0, 0);
}
```

4. 实验结果

使用make run进行验证，成功通过验证。

```
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
QEMU: Terminated
```

5.心得与思考

1. 在我们使用make run时，OpenSBI会产生如下输出:

```
OpenSBI v0.9
____                   _____ ____ _____
/ __ \                 / ____|  _ \_   _|
| |  | |_ __   ___ _ __ | (___ | |_) || |
| |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
| |__| | |_) |  __/ | | |____) | |_) || |_
\____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

......

Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

......
```

通过查看RISC-V Privileged Spec中的medeleg和mideleg解释上面MIDELEG值的含义。

答：medlege和midlege是S模式下的代理相关的寄存器。其中medlege是有关异常的代理寄存器，midleg是有关中断的代理寄存器。其目的是提高性能，使得S模式下某些异常可以不在M模式下处理，类似将处理部分异常的权限下放给S模式。而midlege为interrupt设置了代理位。第五位就是S模式clock中断的代理，1000100010的第五位为1也符合情况。

心得：这次实验通过构建head.S，trap.c，clock.c的关系来设置S mode的时钟中断。通过本实验的操作，我理解了interrupt，exception的关系，同时学会了设置clock，同时也能够理解了lab1没有全部理解的顶层结构，了解了运作方式。

本实验的难点是对riscv的异常的学习，通过阅读riscv的说明书，我了解了有关异常寄存器的功能与实现细节，知道了对应的位数与对应异常的关系。其实这一部分和体系结构的lab2是刚好连在一起的内容，放在一起学确实可以互相借鉴参考，裨益很大。