Lab 2: RV64 时钟中断处理

学号：3200105088　姓名 巩德志

1 实验目的

- 学习 RISC-V 的 trap 处理相关寄存器与指令，完成对 trap 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 trap 处理函数，完成对特定 trap 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

- Environment in previous labs

3 实验步骤

3.1 准备工程

从 `repo` 同步以下代码: `stddef.h` `printk.h` `printk.c`，并按如下目录结构放置。**还需要将之前所有 `print.h puti puts` 的引用修改为 `printk.h printk`**。

```
.
├── Makefile
├── arch
├── include
│   ├── printk.h
│   ├── stddef.h
│   └── types.h
├── init
└── lib
    ├── Makefile
    └── printk.c
```

修改 `vmlinux.lds` 以及 `head.S`

3.2 开启 trap 处理

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 trap 处理入口函数的基地址。

   将_traps的地址加载到t0，调用csrw向csr寄存器stvec中写入t0

   ```
   # set stvec = _traps
   la t0, _traps
   csrw stvec, t0
   ```

2. 开启时钟中断，将 `sie[STIE]` 置 1。

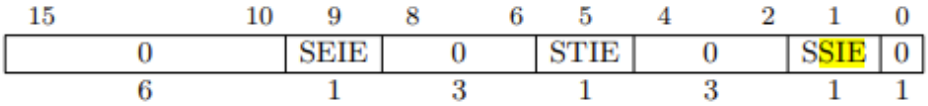

Figure 4.7: Standard portion (bits 15:0) of `sie`.

由上图知，SIE的STIE是第五位，所以SIE第五位设1，即0x20

```
    # set sie[STIE] = 1
    li t0, 0x20
    csrw sie, t0
```

3. 设置第一次时钟中断，参考 `clock_set_next_event()`（`clock_set_next_event()` 在 4.5 中介绍）中的逻辑用汇编实现。

调用rdtime获得当前的时间，time+offset存到a0，其他寄存器置0。

Function Name	Function ID	Extension ID
sbi_set_timer （设置时钟相关寄存器）	0	0x00
sbi_console_putchar （打印字符）	0	0x01
sbi_console_getchar （接收字符）	0	0x02
sbi_shutdown （关机）	0	0x08

a6=0， a7=0，对应sbi_set_timer （设置时钟相关寄存器）
然后ecall调用sbi进行中断

```
    # set first time interrupt
    rdtime t0
    li t1, 10000000
    add t2, t0, t1
    mv a0, t2
    li a1, 0
    li a2, 0
    li a3, 0
    li a4, 0
    li a5, 0
    li a6, 0 # fid
    li a7, 0 # ext
    ecall
```

4. 开启 S 态下的中断响应， 将 `sstatus[SIE]` 置 1。

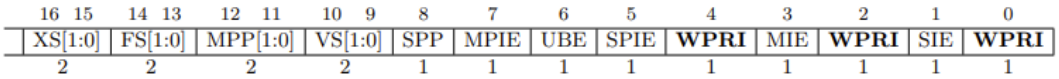

Figure 3.6: Machine-mode status register (**mstatus**) for RV32.

由上图知，sstatus的SIE是第一位，所以SIE位设1，即0x2

```
    # set sstatus[SIE] = 1
    li t0,0x2
    csrw sstatus,t0
```

3.3 实现上下文切换

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）。

```
# 1. save 32 registers and sepc to stack
    addi sp, sp, -256
    sd x1, 0(sp)
    sd x2, 8(sp)
    sd x3, 16(sp)
    sd x4, 24(sp)
    sd x5, 32(sp)
    sd x6, 40(sp)
    sd x7, 48(sp)
    sd x8, 56(sp)
    sd x9, 64(sp)
    sd x10, 72(sp)
    sd x11, 80(sp)
    sd x12, 88(sp)
    sd x13, 96(sp)
    sd x14, 104(sp)
    sd x15, 112(sp)
    sd x16, 120(sp)
    sd x17, 128(sp)
    sd x18, 136(sp)
    sd x19, 144(sp)
    sd x20, 152(sp)
    sd x21, 160(sp)
    sd x22, 168(sp)
    sd x23, 176(sp)
    sd x24, 184(sp)
    sd x25, 192(sp)
    sd x26, 200(sp)
    sd x27, 208(sp)
    sd x28, 216(sp)
    sd x29, 224(sp)
    sd x30, 232(sp)
    sd x31, 240(sp)
    csrr t0, sepc
    sd t0, 248(sp)
```

栈指针先后退256个字节，后通过sd指令将x1-x31压栈，通过csrr读出sepc的值，然后压栈

3. 将 `scause` 和 `sepc` 中的值传入 trap 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 trap 的处理。

```
    # 2. call trap_handler
    csrr a0, scause
    csrr a1, sepc
    call trap_handler
```

将scause读到a0，将sepc读到a1，通过a0 a1传给trap_handler

4. 在完成对 trap 的处理之后， 我们从内存中（栈上）恢复CPU的寄存器（上下文）。

```
    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from
stack
    ld t0, 248(sp)
    csrw sepc, t0
    ld x31, 240(sp)
    ld x30, 232(sp)
    ld x29, 224(sp)
    ld x28, 216(sp)
    ld x27, 208(sp)
    ld x26, 200(sp)
    ld x25, 192(sp)
    ld x24, 184(sp)
    ld x23, 176(sp)
    ld x22, 168(sp)
    ld x21, 160(sp)
    ld x20, 152(sp)
    ld x19, 144(sp)
    ld x18, 136(sp)
    ld x17, 128(sp)
    ld x16, 120(sp)
    ld x15, 112(sp)
    ld x14, 104(sp)
    ld x13, 96(sp)
    ld x12, 88(sp)
    ld x11, 80(sp)
    ld x10, 72(sp)
    ld x9, 64(sp)
    ld x8, 56(sp)
    ld x7, 48(sp)
    ld x6, 40(sp)
    ld x5, 32(sp)
    ld x4, 24(sp)
    ld x3, 16(sp)
    ld x1, 0(sp)
    ld x2, 8(sp)
    addi sp, sp, 256
```

读出sepc的值，通过csrw写入sepc
通过ld将寄存器赋原值，注意x2保存的是栈指针，需要最后一个load

最后栈指针前进256字节

5. 从 trap 中返回。

```
    # 4. return from trap
    sret
```

3.4 实现 trap 处理函数

```
if (scause == 0x8000000000000005)
    {
        printk("kernel is running!\n[S] Supervisor Mode Timer Interrupt\n");
        clock_set_next_event();
    }
```

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.

由上图知，trap类型是interrupt对应末位是1，timer interrupt对应exception code为5，所以判断scause == 0x8000000000000005

3.5 实现时钟中断相关函数

```
unsigned long get_cycles()
{
    // 编写内联汇编，使用 rdtime 获取 time 寄存器中 (也就是mtime 寄存器 )的值并返回
    // YOUR CODE HERE
```

```
    unsigned long ret;
    __asm__ volatile(
        "rdtime %[ret]"
        : [ret] "=r"(ret)
        :
        : "memory");
    return ret;
}
```

```
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
    // YOUR CODE HERE
    sbi_ecall(0x0, 0x0, next, 0, 0, 0, 0, 0);
}
```

Function ID=0， Extension ID=0， 实现设置时钟相关寄存器， next传入设置的时间

3.6 编译及测试

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

运行结果如上图

4 思考题

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。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*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	*Reserved*
0	11	Environment call from M-mode
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 3.6: Machine cause register (`mcause`) values after trap.

`medeleg`中的每一位置位对应`mcause`寄存器中的返回值，`0x000000000000b109`中将第0、3、8、12、13、15位置为1，表示exception code为0、3、8、12、13、15的exception委托给低特权的trap处理程序。

`mideleg`中的每一位与`mip`寄存器对应，`0x0000000000000222`中将第1、5、9位置为1，即SSIP,STIP,SEIP置为1，表示中断被委派给S mode处理。