# 浙江大学实验报告

## 一、实验内容

### 4.1 准备工程

根据实验指导复制文件，确保目录结构与需要修改的代码一致即可。

### 4.2 开启异常处理

这一步骤中，我们需要对`stvec`，`sie`，`sstatus`寄存器进行初始化，同时设置第一次时钟中断，即调用`opensbi`中的接口设置`mtimecmp`寄存器的值，使得当`mtimecmp`寄存器的值大于`mtime`时，会触发中断，进入`_trap`函数。

```
# set stvec
la a0, _traps
csrw stvec, a0

# set sstatus[sie]
csrsi sstatus,2

# set the first time interrupt
lui a1,0x0400
rdtime a0
add a0,a0,a1
addi a7,zero,0
addi a6,zero,0
addi a1,zero,0
addi a2,zero,0
addi a3,zero,0
addi a4,zero,0
addi a5,zero,0
ecall

# set sie[STIE]
addi a0,zero,32
csrs sie,a0
```

### 4.3 实现上下文切换

这一部分我们需要使用汇编实现上下文切换机制，包括保存CPU的寄存器上下文以及`mepc`到栈上。处理中断时，程序会优先处理高优先级的中断，`_trap`首先需要在栈上开一块足够大的空间，并将需要保存的信息压入栈上。

```
_traps:
    # YOUR CODE HERE
    # -----------
    # 1. save 32 registers and sepc to stack
    addi sp, sp, -256
```

```
    sd ra,0(sp)
    sd gp,8(sp)
    sd tp,16(sp)
    sd t0,24(sp)
    sd t1,32(sp)
    sd t2,40(sp)
    sd s0,48(sp)
    sd s1,56(sp)
    sd a0,64(sp)
    sd a1,72(sp)
    sd a2,80(sp)
    sd a3,88(sp)
    sd a4,96(sp)
    sd a5,104(sp)
    sd a6,112(sp)
    sd a7,120(sp)
    sd s2,128(sp)
    sd s3,136(sp)
    sd s4,144(sp)
    sd s5,152(sp)
    sd s6,160(sp)
    sd s7,168(sp)
    sd s8,176(sp)
    sd s9,184(sp)
    sd s10,192(sp)
    sd s11,200(sp)
    sd t3, 208(sp)
    sd t4, 216(sp)
    sd t5, 224(sp)
    sd t6, 232(sp)
    csrr t6, sepc
    sd t6, 240(sp)
    sd sp, 248(sp)
```

随后, 读取`mepc`和`mcause`寄存器并作为参数(即保存到`a0`, `a1`)中, 并调用`trap_handler`函数。

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

最后等待`trap_handler`函数结束, 恢复寄存器上下文以及`mepc`的值, 并返回到`mepc`处继续执行。

```
    # -----------
    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from
stack
    ld ra,0(sp)
    ld gp,8(sp)
    ld tp,16(sp)
    ld t0,24(sp)
    ld t1,32(sp)
    ld t2,40(sp)
    ld s0,48(sp)
    ld s1,56(sp)
```

```
    ld a0,64(sp)
    ld a1,72(sp)
    ld a2,80(sp)
    ld a3,88(sp)
    ld a4,96(sp)
    ld a5,104(sp)
    ld a6,112(sp)
    ld a7,120(sp)
    ld s2,128(sp)
    ld s3,136(sp)
    ld s4,144(sp)
    ld s5,152(sp)
    ld s6,160(sp)
    ld s7,168(sp)
    ld s8,176(sp)
    ld s9,184(sp)
    ld s10,192(sp)
    ld s11,200(sp)
    ld t3, 208(sp)
    ld t4, 216(sp)
    ld t5, 224(sp)
    ld t6, 240(sp)
    csrw sepc, t6
    ld t6, 232(sp)
    ld sp, 248(sp)
    addi sp,sp,256
    # -----------
    # 4. return from trap
    # -----------
    sret
```

### 4.4 实现异常处理函数

在这一部分，实验指导的注释已经给出了函数需要做的事情。

根据注释的指导，首先需要判断是否产生了一个时钟中断。查询相关的riscv手册，可以得知，产生时钟中断时，`scause`的最高位置1，exception code为5，由此得知`mcause`值为0x8000000000000005。（这里要注意位数问题，详见讨论心得部分）判断为时钟中断后，根据实验指导输出一定的信息即可。这里为了方便，我给每次输出赋予一个序号，并且会输出下一次时钟中断的时间（具体值由时钟中断相关函数打印）。输出信息之后，调用设置下一次中断时间的函数，设置下一次时钟中断发生的时间即可。代码如下：

```
#include "clock.h"
#include "printk.h"
static int i=0;

void trap_handler(unsigned long long scause, unsigned long long sepc) {
    if (scause == 0x8000000000000005){
        printk("[%d] Supervisor Mode Timer Interrupt ", i=(i+1)%3600);
        printk("Next time: ");
        clock_set_next_event();
    }
    else {
        ;
    }
```

```
    return ;
}
```

### 4.5 实现时钟中断相关函数

该文件内需要完成两个函数，第一个函数获取当前时间，第二个函数设置下一次时钟中断发生的时间。

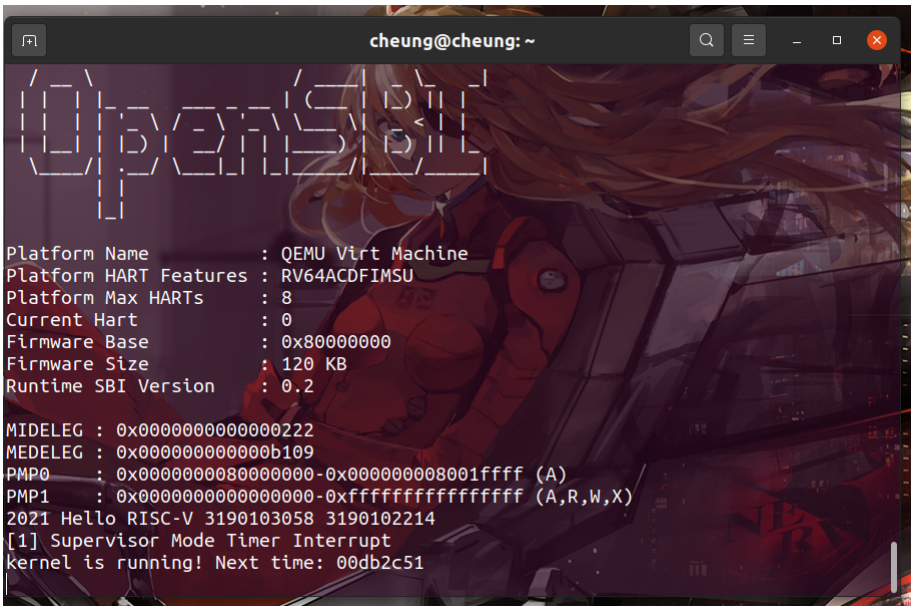
第一个函数非常简单，利用内联汇编，调用`rdtime`即可。

第二个函数首先计算出下一次时钟中断发生的时间并打印，然后编写内联汇编，正确设置参数后使用`sbi_ecall`来设置下一次的时间。这部分的内联汇编与`head.S`中设置第一次触发时钟中断的汇编代码基本一致。

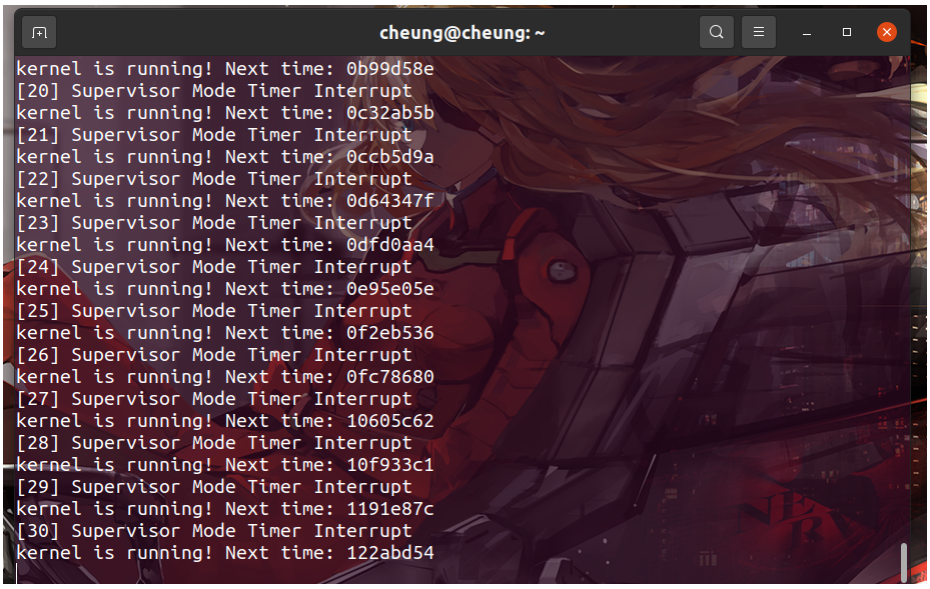
```
#include"printk.h"
// QEMU中时钟的频率是10MHz, 也就是1秒钟相当于10000000个时钟周期。
unsigned long TIMECLOCK = 10000000;

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 res;
    __asm__ volatile(
        "rdtime %[res]\n"
        : [res] "=r"(res)               // output
        :                               // input
        :                               // modification
    );
    return res;
}

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
    printk("%x\n", next);

    __asm__ volatile(
        "addi a0, %[next], 0\n"
        "addi a1, zero, 0\n"
        "addi a2, zero, 0\n"
        "addi a3, zero, 0\n"
        "addi a4, zero, 0\n"
        "addi a5, zero, 0\n"
        "addi a6, zero, 0\n"
        "addi a7, zero, 0\n"
        "ecall\n"
        :           // output
        : [next] "r" (next) // input
        : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
    );
}
```

## 编译及测试

运行`make run`，查看输出，可以看到正常产生了输出。




结合光标闪烁的频率，不难发现系统确实是每隔大约一秒输出一次信息，由此可以验证时钟中断的正确性。




运行`make debug`，通过gdb验证trap前后各寄存器的值相同。如下两张图是刚进入trap时各寄存器的值：

```
$zero: 0x0000000000000000  →  0x0000000000000000
$ra  : 0x00000000802002e4  →  0x00054603fedff06f  →  0x00054603fedff06f
$sp  : 0x0000000080203fd0  →  0x0000000000000000  →  0x0000000000000000
$gp  : 0x0000000000000000  →  0x0000000000000000
$tp  : 0x0000000080016000  →  0x0000000080000000  →  0x000584b300050433  →  0
$t0  : 0x0000000080200000  →  0x0505051300000517  →  0x0505051300000517
$t1  : 0x0000000000000000  →  0x0000000000000000
$t2  : 0x0000000000000001  →  0x0000000000000001
$fp  : 0x0000000014dc9380  →  0x0000000014dc9380
$s1  : 0x0000000080201070  →  0x69206c656e72656b  →  0x69206c656e72656b
$a0  : 0x0000000000000013  →  0x0000000000000013
$a1  : 0x0000000000000000  →  0x0000000000000000
$a2  : 0x0000000000000000  →  0x0000000000000000
$a3  : 0x0000000000000000  →  0x0000000000000000
$a4  : 0x0000000000000000  →  0x0000000000000000
$a5  : 0x000000000ed62715  →  0x000000000ed62715
$a6  : 0x0000000000000000  →  0x0000000000000000
$a7  : 0x0000000000000001  →  0x0000000000000001
$s2  : 0x8000000000006800  →  0x8000000000006800
$s3  : 0x0000000080200000  →  0x0505051300000517  →  0x0505051300000517
$s4  : 0x0000000087e00000  →  0x860f0000edfe0dd0  →  0x860f0000edfe0dd0
$s5  : 0x0000000000000000  →  0x0000000000000000
$s6  : 0x0000000000000000  →  0x0000000000000000
$s7  : 0x00000000800120f0  →  0x0000000000000000  →  0x0000000000000000
$s8  : 0x000000000000007f  →  0x000000000000007f
$s9  : 0x0000000080011010  →  0x0000000200000000  →  0x0000000200000000
$s10 : 0x0000000000000000  →  0x0000000000000000
$s11 : 0x0000000000000000  →  0x0000000000000000
$t3  : 0x0000000000000000  →  0x0000000000000000
$t4  : 0x0000000000000000  →  0x0000000000000000
$t5  : 0x0000000000000001  →  0x0000000000000001
$t6  : 0x0000000087e00000  →  0x860f0000edfe0dd0  →  0x860f0000edfe0dd0
```

```
gef➤  i registers sepc
sepc           0x802002d4       0x802002d4
gef➤
```

在执行sret之前，再次查看各个寄存器的值，可以看到trap前后通用寄存器的值不变，调用`trap_handler`前后`sepc`值不变。

```
$zero: 0x0000000000000000  →  0x0000000000000000
$ra  : 0x00000000802002e4  →  0x00054603fedff06f  →  0x00054603fedff06f
$sp  : 0x0000000080203fd0  →  0x0000000000000000  →  0x0000000000000000
$gp  : 0x0000000000000000  →  0x0000000000000000
$tp  : 0x0000000080016000  →  0x0000000080000000  →  0x000584b300050433  →  0x000584b300050433
$t0  : 0x0000000080200000  →  0x0505051300000517  →  0x0505051300000517
$t1  : 0x0000000000000000  →  0x0000000000000000
$t2  : 0x0000000000000001  →  0x0000000000000001
$fp  : 0x0000000014dc9380  →  0x0000000014dc9380
$s1  : 0x0000000080201070  →  0x69206c656e72656b  →  0x69206c656e72656b
$a0  : 0x0000000000000013  →  0x0000000000000013
$a1  : 0x0000000000000000  →  0x0000000000000000
$a2  : 0x0000000000000000  →  0x0000000000000000
$a3  : 0x0000000000000000  →  0x0000000000000000
$a4  : 0x0000000000000000  →  0x0000000000000000
$a5  : 0x000000000ed62715  →  0x000000000ed62715
$a6  : 0x0000000000000000  →  0x0000000000000000
$a7  : 0x0000000000000001  →  0x0000000000000001
$s2  : 0x8000000000006800  →  0x8000000000006800
$s3  : 0x0000000080200000  →  0x0505051300000517  →  0x0505051300000517
$s4  : 0x0000000087e00000  →  0x860f0000edfe0dd0  →  0x860f0000edfe0dd0
$s5  : 0x0000000000000000  →  0x0000000000000000
$s6  : 0x0000000000000000  →  0x0000000000000000
$s7  : 0x00000000800120f0  →  0x0000000000000000  →  0x0000000000000000
$s8  : 0x000000000000007f  →  0x000000000000007f
$s9  : 0x0000000080011010  →  0x0000000200000000  →  0x0000000200000000
$s10 : 0x0000000000000000  →  0x0000000000000000
$s11 : 0x0000000000000000  →  0x0000000000000000
$t3  : 0x0000000000000000  →  0x0000000000000000
$t4  : 0x0000000000000000  →  0x0000000000000000
$t5  : 0x0000000000000001  →  0x0000000000000001
$t6  : 0x0000000087e00000  →  0x860f0000edfe0dd0  →  0x860f0000edfe0dd0
```

```
gef➤  i registers sepc
sepc           0x802002d4       0x802002d4
gef➤
```

## 二、思考题

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

- `medeleg` 代表machine exception delegation register，`mideleg` 代表machine interrupt delegation register，二者可以指定某一些异常或者中断直接由更低等级的模式来处理，提高性能，而不是所有的异常或中断都需要经过机器模式。
- 类似地，在有M/S/U三种模式的系统中，可能还有 `sedeleg` 和 `sideleg` 寄存器。
- 在有M/S/U三种模式的系统中，`medeleg` 和 `mideleg` 指定了哪些发生在S模式或U模式中的异常或中断交由S模式中的trap handler来处理。
- 在有M/U两种模式的系统中，`medeleg` 和 `mideleg` 指定了哪些发生在U模式中的异常或中断交由U模式中的trap handler来处理。
- 程序中 `mideleg` 为0x0x0000000000000222，代表有三种中断交由S模式处理。其中第二个2代表 `mideleg[5]`，代表S模式的时钟中断
- 程序中 `medeleg` 为0x0x000000000000b109，代表有六种异常交由S模式处理。如 `medeleg[15]`，代表store时的缺页异常交由S模式处理

2. 思考，在“上下文切换的过程”中，我们需要保护哪些寄存器。为什么。

- 所有通用寄存器，要确保trap执行后，原来的程序还能正常继续执行，其寄存器值不能被改变
- `sepc` 或者 `mepc`，有可能中断发生在异常处理的过程中，此时要确保异常处理结束后可以正常返回
- `pc`，确保程序切换回来时能继续执行

## 三、讨论心得

本次实验中，我在完成我负责的部分时主要遇到两个问题：

1. `entry.S` 是trap的入口，在其中调用 `trap_handler` 时一定要用 `call` 而不能用 `j`。此处是调用一个函数，而不是简单地进入某一段代码。否则，函数结束了不能正确返回，最后堆栈溢出会导致程序异常。
2. 第二个问题是 `scause` 的判断问题。寄存器为64位，而unsigned long型变量是32位的，如果不注意到这里会导致程序无法正确判断出时钟中断，进而影响后续的执行。

本次实验是第一次小组实验，队员之间保持沟通，合作交流，可以提高效率，有效减轻彼此的工作量。