

Lab2: 时钟中断处理

1 实验目的

以时钟中断为例学习在RISC-V上的异常处理相关机制。

2 实验内容及要求

- 理解RISC-V异常委托与异常处理机制
- 利用OpenSBI平台的SBI调用触发时钟中断，并通过代码设计实现定时触发时钟中断的效果

3 实验步骤

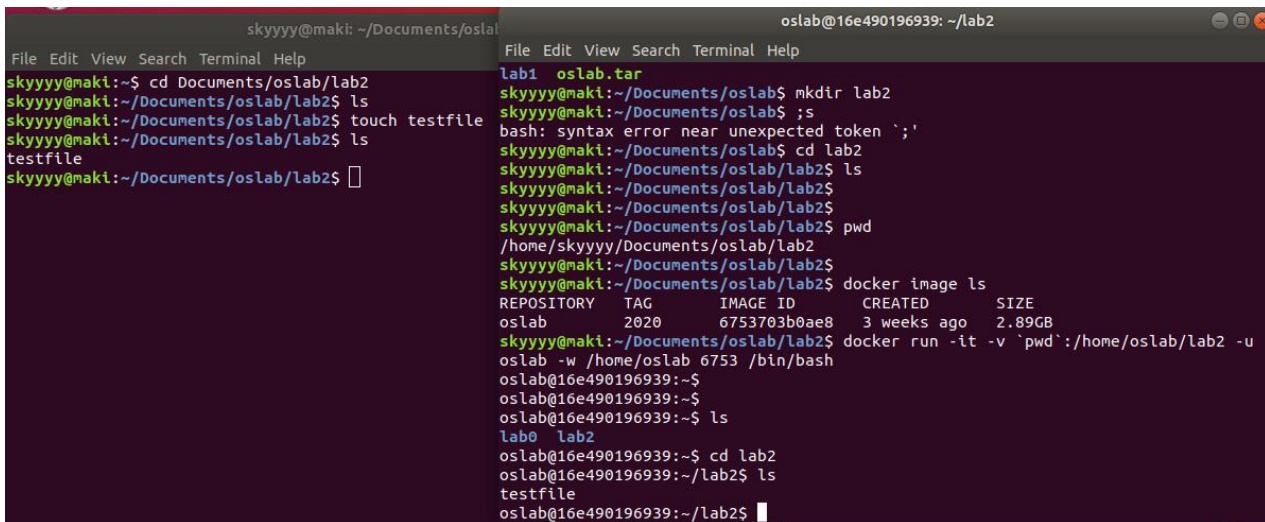
3.1 搭建实验环境，理解实验执行流程

本次实验目标是定时触发时钟中断并在相应的中断处理函数中输出相关信息，代码实现逻辑如下：

- 在初始化阶段，设置CSR寄存器以允许以S模式的时钟中断发生，利用SBI调用触发第一次时钟中断。
- SBI调用触发时钟中断后，OpenSBI平台自动完成了M模式的中断处理。由于设置了S模式的中断使能，随后触发了S模式下的时钟中断，相关异常处理CSR进行自动转换，需要保存寄存器现场后调用相应的时钟中断处理函数。
- 在时钟中断处理函数中打印相关信息，并设置下一次时钟中断，从而实现定时触发时钟中断并打印相关信息的效果。函数返回后恢复寄存器现场，调用监管者异常返回指令sret回到发生中断的指令。

在 `init.c` 中设置CSR寄存器允许中断发生，在 `clock.c` 中触发时钟中断事件，在 `entry.S` 及 `trap.c` 中编写中断处理函数。

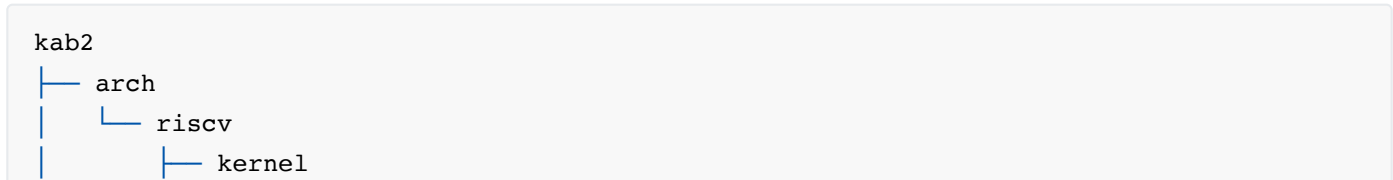
1. 创建容器，映射文件夹



```
skyyyy@maki: ~/Documents/oslab
File Edit View Search Terminal Help
skyyyy@maki:~$ cd Documents/oslab/lab2
skyyyy@maki:~/Documents/oslab/lab2$ ls
skyyyy@maki:~/Documents/oslab/lab2$ touch testfile
skyyyy@maki:~/Documents/oslab/lab2$ ls
testfile
skyyyy@maki:~/Documents/oslab/lab2$

oslab@16e490196939: ~/lab2
File Edit View Search Terminal Help
lab1 oslab.tar
skyyyy@maki:~/Documents/oslab$ mkdir lab2
skyyyy@maki:~/Documents/oslab$ ;s
bash: syntax error near unexpected token `;'
skyyyy@maki:~/Documents/oslab$ cd lab2
skyyyy@maki:~/Documents/oslab/lab2$ ls
skyyyy@maki:~/Documents/oslab/lab2$
skyyyy@maki:~/Documents/oslab/lab2$
skyyyy@maki:~/Documents/oslab/lab2$ pwd
/home/skyyyy/Documents/oslab/lab2
skyyyy@maki:~/Documents/oslab/lab2$
skyyyy@maki:~/Documents/oslab/lab2$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
oslab 2020 6753703b0ae8 3 weeks ago 2.89GB
skyyyy@maki:~/Documents/oslab/lab2$ docker run -it -v `pwd`:/home/oslab/lab2 -u
oslab -w /home/oslab 6753 /bin/bash
oslab@16e490196939:~$
oslab@16e490196939:~$ ls
lab0 lab2
oslab@16e490196939:~$ cd lab2
oslab@16e490196939:~/lab2$ ls
testfile
oslab@16e490196939:~/lab2$
```

2. 组织文件结构



```
├── clock.c
├── head.S
├── entry.S
├── init.c
├── main.c
├── Makefile
├── print.c
├── sbi.c
├── trap.c
└── vmlinux.lds
└── Makefile
├── include
│   ├── defs.h
│   ├── riscv.h
│   └── test.h
├── Makefile
└── README.md
```

3. 修改必要文件

由于裸机程序需要在 `.text` 段起始位置执行，所以需要利用 `vmlinux.lds` 中 `.text` 段的定义来确保 `head.S` 中的 `.text` 段被放置在其他 `.text` 段之前，首先修改 `head.S` 中的 `.text` 命名为 `.text.init`：

```
<<<<< before
.section .text
=====
.section .text.init
>>>>> after
```

修改 `entry.S` 中的 `.text` 命名为 `.text.entry`：

```
<<<<< before
.section .text
=====
.section .text.entry
>>>>> after
```

然后修改 `vmlinux.lds` 文件中的 `.text` 展开方式：

```

<<<<< before
.text : {
    *(.text)
    *(.text.*)
}
=====
.text : {
    *(.text.init)
    *(.text.entry)
    *(.text)
    *(.text.*)
}
>>>>> after

```

3.2 编写init.c中的相关函数

M模式事务由OpenSBI管理，因此只要进行S模式下的处理。为了满足时钟中断在S模式下的使能条件，需要对寄存器进行如下设置：

1. 设置 stvec 寄存器。stvec 寄存器中存储着S模式下发生中断时跳转的地址，编写相关的中断处理函数并将地址存入 stvec 中。
2. 将 sie 寄存器中的stie位 置位。sie[stie] 为1时，s模式下的时钟中断开启。
3. 将 sstatus 寄存器中的sie位置位。sstatus[sie] 位为s模式下的中断总开关，这一位为1时，才能响应中断。

1. 编写 intr_enable() / intr_disable()

sstatus寄存器

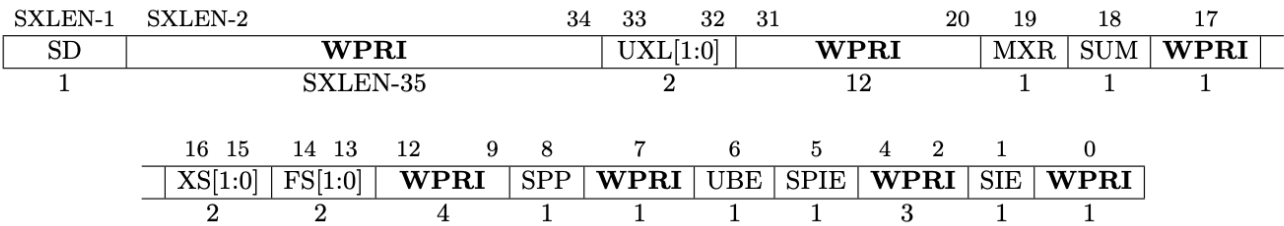


Figure 4.2: Supervisor-mode status register (sstatus) for RV64.

内联汇编使用CSR命令设置 sstatus[sie] 的值

```

void intr_enable(void) {
    //设置sstatus[sie]=1,打开s模式的中断开关
    //your code
    __asm__ volatile (
        "li t0, 0x0000000000000002\n"
        "csrs sstatus, t0\n"
        :
        :
    );
}

```

```

}

void intr_disable(void) {
    //设置sstatus[sie]=0,关闭s模式的中断开关
    //your code
    __asm__ volatile (
        "li t0, 0xfffffffffffffd\n"
        "csrc sstatus, t0\n"
        :
        :
    );
}

```

代码解释

```
li t0, 0x0000000000000002
```

以 `t0` 作为掩码，由于 `sstatus[sie]` 在 `sstatus` 寄存器的第1位，将掩码设置为 `0x0000000000000002`。

```
csrs sstatus, t0
```

`csrs csr, rs` 表示根据 `rs` 中的值按位对 `csr` 的值置1，这里利用掩码将 `sstatus[sie]` 置1。

`void intr_disable(void)` 中的代码解释同上。

2. 编写 `idt_init()`

使用 `riscv.h` 文件中的相应函数向 `stvec` 寄存器中写入中断处理后跳转函数 `trap_s` (entry.S)的地址。

```

void idt_init(void) {
    extern void trap_s(void);
    //向stvec寄存器中写入中断处理后跳转函数的地址
    //your code
    unsigned long long int addr = &trap_s;
    __asm__ volatile (
        "mv t0, %[addr]\n"
        "csrw stvec, t0\n"
        :
        : [addr] "r" (addr)
    );
}

```

3.3 编写 `clock.c` 中的相关函数

时钟中断 实际上就是"每隔若干个时钟周期执行一次的程序", 可以利用OpenSBI提供的 `sbi_set_timer()` 接口实现, 向该函数传入一个时刻, 函数将在那个时刻触发一次时钟中断。使用SBI调用完成了M模式下的中断触发与处理, 不需要考虑 `mtime`、`mtimecmp` 等相关寄存器。

OpenSBI提供的接口一次只能设置一个时钟中断事件。因此需要一开始只设置一个时钟中断, 之后每次发生时钟中断的时候, 在相应的中断处理函数中设置下一次的时钟中断。

在 `clock.c` 中定时触发时钟中断事件, `clock_init()` 用于启用时钟中断并设置第一个时钟中断; `clock_set_next_event()` 用于调用SBI函数 `set_sbi_timer()` 触发时钟中断; `get_cycle()` 函数通过 `rdtime` 伪指令读取一个叫做 `time` 的CSR的数值, 表示CPU启动之后经过的真实时间。

qemu中外设晶振的频率为10mhz, 即每秒钟 `time` 的值将会增大 10^7 。

Q2: 为了使得每次时钟中断的间隔为1s, `timebase` (即 `time` 的增加量) 需要设置为 10^7 ; 若需要时钟中断间隔为0.1s, `timebase` 需要设置为 10^6 。

```
static uint64_t timebase = 10000000; //自行修改timebase为合适的值, 使得时钟中断间隔为1s
```

1. 编写 `clock_init()`

```
void clock_init(void) {
    puts("ZJU OS LAB 2          Student_ID:3190102362\n");
    //对sie寄存器中的时钟中断位设置 (sie[stie]=1) 以启用时钟中断。
    //设置第一个时钟中断。
    //your code
    __asm__ volatile (
        "li t0, 0x0000000000000020\n"
        "csrs sie, t0\n"
        :
        :
    );
    ticks ++;
    trigger_time_interrupt(get_cycles());
}
```

2. 编写 `clock_set_next_event()`

```
void clock_set_next_event(void) {
    //获取当前cpu cycles数并计算下一个时钟中断的发生时刻,通过sbi_set_timer()触发时钟中断。
    //your code
    ticks ++;
    trigger_time_interrupt(get_cycles() + timebase);
}
```

解释 `sbi.c` 中 `trigger_time_interrupt()` 函数。

```
void trigger_time_interrupt(unsigned long long stime_value) {
    sbi_call(0, stime_value, 0, 0);
}
```

Type	Function	Function ID
Timer	sbi_set_timer	0
Console	sbi_console_putchar	1
	sbi_console_getchar	2
IPI	sbi_clear_ipi	3
	sbi_send_ipi	4
Memory Model	sbi_remote_fence_i	5
	sbi_remote_sfence_vma	6
	sbi_remote_sfence_vma_as	7
	id	
Shutdown	sbi_shutdown	8

由表可知，SBI函数 `set_sbi_timer()` 的调用编号为0，因此函数 `sbi_call` 的第一个参数 `sbi_type` 为0；第二个参数 `stime_value` 为调用 `set_sbi_timer()` 时的时间；最后两个参数可设置为0。

3.4 编写并调用中断处理函数

1. 在 `entry.S` 中调用中断处理函数

在 `init.c` 中，函数 `idt_init()` 向 `stvec` 寄存器存入了中断处理函数的地址，中断发生后将自动进行硬件状态转换，程序将读取 `stvec` 的地址并进行跳转，运行 `trap_s` 函数。该函数需要在栈中保存 `调用者保存寄存器` 及 `sepc` 寄存器，读取 `scause` CSR寄存器设置参数 `a0` 调用 `handle_s` 函数进行中断处理，调用返回后需要恢复寄存器并使用 `sret` 命令回到发生中断的指令。

寄存器大小为8字节，因此需要使用 `ld` 和 `sd` 进行双字的加载/存储。

调用者保存寄存器

寄存器	接口名称	描述	在调用中是否保留?
Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero 硬编码 0	—
x1	ra	Return address 返回地址	No
x2	sp	Stack pointer 栈指针	Yes
x3	gp	Global pointer 全局指针	—
x4	tp	Thread pointer 线程指针	—
x5	t0	Temporary/alternate link register 临时寄存器	No /备用链接寄存器
x6-7	t1-2	Temporaries 临时寄存器	No
x8	s0/fp	Saved register/frame pointer 保存寄存器	Yes /帧指针
x9	s1	Saved register 保存寄存器	Yes
x10-11	a0-1	Function arguments/return values 函数参数	No /返回值
x12-17	a2-7	Function arguments 函数参数	No
x18-27	s2-11	Saved registers 保存寄存器	Yes
x28-31	t3-6	Temporaries 临时寄存器	No
f0-7	ft0-7	FP temporaries 浮点临时寄存器	No
f8-9	fs0-1	FP saved registers 浮点保存寄存器	Yes
f10-11	fa0-1	FP arguments/return values 浮点参数/返回值	No
f12-17	fa2-7	FP arguments 浮点参数	No
f18-27	fs2-11	FP saved registers 浮点保存寄存器	Yes
f28-31	ft8-11	FP temporaries 浮点临时寄存器	No

由上图可知，调用前需要保存 ra, t0 - t6, a0 - a7

保存 sepc 寄存器的原因：

S模式下中断发生时 PC 会被存入 sepc 寄存器，之后 PC 被设置为 stvec，即将中断处理函数的地址装入 PC。为了使中断结束后程序能从中断发生的位置继续运行，需要保存 sepc 寄存器来恢复 PC。

entry.s 代码如下

```
.section .text.entry

.global trap_s
.extern handler_s
.equ reg_size, 0x8
.align 2

trap_s:
    #save callee-saved registers and spec
    addi sp, sp, -16
    sd ra, 0(sp)
    sd t0, 1(sp)
    sd t1, 2(sp)
    sd t2, 3(sp)
    sd a0, 4(sp)
    sd a1, 5(sp)
    sd a2, 6(sp)
    sd a3, 7(sp)
    sd a4, 8(sp)
    sd a5, 9(sp)
```

```

sd a6, 10(sp)
sd a7, 11(sp)
sd t3, 12(sp)
sd t4, 13(sp)
sd t5, 14(sp)
sd t6, 15(sp)
csrr t0, sepc
sd t0, 16(sp)

# call handler_s(scause)
csrr a0, scause
call handler_s

# load sepc and callee-saved registers
ld t0, 16(sp)
csrw sepc, t0
ld ra, 0(sp)
ld t0, 1(sp)
ld t1, 2(sp)
ld t2, 3(sp)
ld a0, 4(sp)
ld a1, 5(sp)
ld a2, 6(sp)
ld a3, 7(sp)
ld a4, 8(sp)
ld a5, 9(sp)
ld a6, 10(sp)
ld a7, 11(sp)
ld t3, 12(sp)
ld t4, 13(sp)
ld t5, 14(sp)
ld t6, 15(sp)
addi sp, sp, 16

sret

```

2. 在 `trap.c` 中编写中断处理函数

正常情况下，异常处理函数需要根据 `[m|s]cause` 寄存器的值判断异常的种类后分别处理不同类型的异常，本次实验中简化为只判断并处理时钟中断。

为了实现"定时触发中断"，需要在中断处理函数中设置下一时钟中断；此外，在 `clock.c` 中利用 `tick` 变量进行时钟中断发生次数的统计。

```

void handler_s(uint64_t cause){
    //interrupt
    if ( cause & 0x10000000000000000 == 0x10000000000000000 ) {
        // supervisor timer interrupt
        uint64_t cause1 = cause & 0x7fffffffffffffff;
    }
}

```



```
if ( cause1 == 5 ) {
    //设置下一个时钟中断，打印当前的中断数目。
    //your code
    puts("[S] Supervisor Mode Timer Interrupt ");
    put_num(ticks);
    puts("\n");
    clock_set_next_event();
}
}
```

触发时钟中断时， [m|s]cause 寄存器的值为：

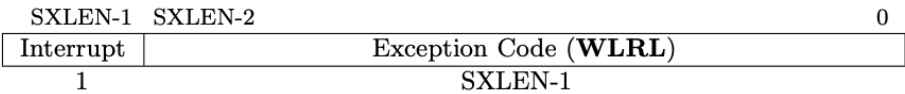


Figure 4.11: Supervisor Cause register **scause**.

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	Available 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	Available for custom use
0	32-47	Reserved
0	48-63	Available for custom use
0	≥64	Reserved

[m|s]cause 寄存器的值为 0x8000 0000 0000 0005

寄存器最高位为1，表示中断；ExceptionCode为5，表示时钟中断。

【同步异常与中断的区别】当处理同步异常时应该在退出前给 `epc` 寄存器+4（一条指令的长度），当处理中断时则不需要

同步异常中 `epc` 指向导致异常的指令，为了程序继续运行，需要将 `epc` 指向导致异常的指令的下一条指令，否则会一直触发同步异常；

中断发生后 `epc` 指向中断处理后应该恢复执行的位置，因此不需要改变。

3.5 编译与测试

实验结果如下，每隔一秒触发一次时钟中断，打印一次信息。

```
MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffff (A,R,W,X)
ZJU OS LAB 2          Student_ID:3190102362
[S] Supervisor Mode Timer Interrupt 1
[S] Supervisor Mode Timer Interrupt 2
[S] Supervisor Mode Timer Interrupt 3
[S] Supervisor Mode Timer Interrupt 4
[S] Supervisor Mode Timer Interrupt 5
[S] Supervisor Mode Timer Interrupt 6
[S] Supervisor Mode Timer Interrupt 7
[S] Supervisor Mode Timer Interrupt 8
[S] Supervisor Mode Timer Interrupt 9
[S] Supervisor Mode Timer Interrupt 10
[S] Supervisor Mode Timer Interrupt 11
[S] Supervisor Mode Timer Interrupt 12
[S] Supervisor Mode Timer Interrupt 13
[S] Supervisor Mode Timer Interrupt 14
[S] Supervisor Mode Timer Interrupt 15
[S] Supervisor Mode Timer Interrupt 16
[S] Supervisor Mode Timer Interrupt 17
[S] Supervisor Mode Timer Interrupt 18
[S] Supervisor Mode Timer Interrupt 19
[S] Supervisor Mode Timer Interrupt 20
[S] Supervisor Mode Timer Interrupt 21
[S] Supervisor Mode Timer Interrupt 22
[S] Supervisor Mode Timer Interrupt 23
[S] Supervisor Mode Timer Interrupt 24
[S] Supervisor Mode Timer Interrupt 25
[S] Supervisor Mode Timer Interrupt 26
[S] Supervisor Mode Timer Interrupt 27
QEMU: Terminated
oslab@16e490196939:~/lab2$
```

4 讨论和心得

由于详尽的实验指导，本次实验的代码实现并不困难，利用OpenSBI省去了很多汇编过程。但和lab1一样，实验所涉及的相关知识依然非常丰富，需要更多时间进行学习理解。