Lab2 RV64 时钟中断处理

赵伊蕾 学号：3200104866

王柯棣 学号：3200105119

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. 实验任务

2.1 完成 `head.s` 使得开启trap处理

运行start_kernel之前，需要先对控制状态寄存器(Control and Status Registers)进行初始化，主要步骤如下：

- 设置 `stvec`， 将 `_traps` 所表示的地址写入 `stvec`，便于trap发生之后地址指针可以直接切换到 `s-mode` 的地址。这里我们采用 `Direct 模式`，而 `_traps` 则是 trap 处理入口函数的基地址。
- 开启时钟中断，将 `sie[STIE]` 置 1。
  - sie寄存器内部组成

XLEN-1 … 10	9	8	7 … 6	5	4	3 … 2	1	0
0	SEIE	UEIE	0	STIE	UTIE	0	SSIE	USIE
XLEN-10	1	1	2	1	1	2	1	1

  - sie[STIE]是第6位，所以可以用 `ori` 语句将 `sie` 的值和 `0x20` 进行计算。
- 设置第一次时钟中断，参考 `clock_set_next_event()` 中的逻辑用汇编实现。
  - 使用 `rdtime` 汇编指令获得当前 `time` 寄存器中的值
  - 设置间断的时间间隔，修改下一时刻的值
  - ecall
- 开启 S 态下的中断响应， 将 `sstatus[SIE]` 置 1。
  - 原理和修改sie一样

31	30 … 20	19	18	17
SD	WPRI	MXR	SUM	WPRI
1	11	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4 2	1	0
XS[1:0]	FS[1:0]	WPRI	VS[1:0]	SPP	WPRI	UBE	SPIE	WPRI	SIE	WPRI
2	2	2	2	1	1	1	1	3	1	1

Figure 4.1: Supervisor-mode status register (**sstatus**) when SXLEN=32.

  - sstatus[SIE]是第二位，所以可以用 `ori` 语句将 `sstatus` 的值和 `0x2` 进行计算。

```
_start:
    # YOUR CODE HERE
    # -----------------
    # set stvec = _traps
    la t0, _traps
```

```
    csrw stvec, t0
    # ------------------
    # set sie[STIE] = 1
    csrr t0, sie ## read in sie
    ori t0, t0, 0x20
    csrw sie, t0 ## write to sie
    # ------------------
    # set first time interrupt
    rdtime t0
    li t1, 10000000 ## time interval
    add a0, t0, t1
    mv a6, zero     ## set_timer ecall
    mv a7, zero
    ecall
    # ------------------
    # set sstatus[SIE] = 1
    csrr t0, sstatus ##read in ssatus
    ori t0, t0, 0x2  ##sie is the second bit
    csrw sstatus, t0 ##write to sstatus
    # ------------------
    la sp, boot_stack_top
    call start_kernel
    # ------------------
    # - your lab1 code -
    # ------------------

    .section .bss.stack
    .globl boot_stack
```

2.2 补全`arch/riscv/kernel/entry.S`中的trap逻辑

- 保存 CPU 的寄存器（上下文）到内存中（栈上）。
    - 这部分将通过不断移动栈指针来完成存储
    - 把sepc的值也传入栈中保存，
- 将 `scause` 和 `sepc` 中的值传入 trap 处理函数 `trap_handler`，我们将会在 `trap_handler` 中实现对 trap 的处理。
    - 把`scause`和`sepc`的值分别读入`a0`和`a1`寄存器里，用作函数调用
    - `call trap_handler`函数
- 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）
    - 从栈中把原来存入的寄存器的值全部load出来，其中还包括了`sepc`的值
    - 特别注意sp的值是要最后读入，不然可能会影响到当前这段内存其他数据的读出
- 从 trap 中返回。
    - `sret`

```
_traps:
    # YOUR CODE HERE
    # -----------
    # 1. save 32 registers and sepc to stack
    sd ra, -1*8(sp)
    sd sp, -2*8(sp)
```

```
    sd gp, -3*8(sp)
    ...
    csrr t0, sepc
    sd t0, -32*8(sp)
    addi sp, sp, -256
    ##save sepc
    csrr t0, sepc
    sd t0, -32*8(sp)
    addi sp, sp, -256
    # -----------
    # 2. call trap_handler
    csrr a0, scause
    csrr a1, sepc
    call trap_handler
    # -----------
    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from
stack
    ld t0, 0(sp)
    csrw sepc, t0
    addi sp, sp, 256
    ld ra, -1*8(sp)
    ld gp, -3*8(sp)
    ...
    ld t6, -31*8(sp)
    ld sp, -2*8(sp)     #finally load the value of sp
    # -----------
    # 4. return from trap
    sret
    # -----------
```

2.3 添加 `trap.c` 文件实现终端处理函数

trap.c文件主要处理trap过程当中需要调用到的`trap_handler`函数。

- 通过 `scause` 判断trap类型
  - 如果是trap，则`scause`的最开头值为1，可以通过下方式子来判断。

```
scause >> 63 == 1
```

- 如果是interrupt 判断是否是timer interrupt
  -

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

  - 如图所示，如果`exception code == 5`表示现在是`timer interrupt`，可以通过下方式子进行判断。

```
(scause & 0b11111) == 5
```

- 如果是timer interrupt 则打印输出相关信息, 并通过 `clock_set_next_event()` 设置下一次时钟中断
- 这个函数里不需要考虑interrupt / exception其他情况

```
void trap_handler(unsigned long scause, unsigned long sepc) {
    // 其他interrupt / exception 可以直接忽略

    // YOUR CODE HERE
    printk("kernel is running!\n");
    // Exception code == 5 <- supervisor time interrupt
    if(scause >> 63 == 1 && (scause & 0b11111) == 5){
        //trap && time interrupt
        printk("[S] Supervisor Mode Timer Interrupt\n");
        clock_set_next_event();
    }
}
```

2.4 完成 `clock.c` 实现时钟中断

`trap.c` 文件中主要处理两个函数：`get_cycles()` 和 `clock_set_next_event()`。

`get_cycles()` 的功能是通过编写内联函数使用 `rdtime` 获取 `time` 寄存器中的值并返回。具体写法与Lab1中一致。

```
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 time = 0;
    asm volatile(
        "rdtime %[time]\n"
        : [time] "=r" (time)
        :
        : "memory"
    );
}
```

`clock_set_next_event()` 函数是通过sbi_ecall来完成对下一次时钟中断的设置。

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	0x80

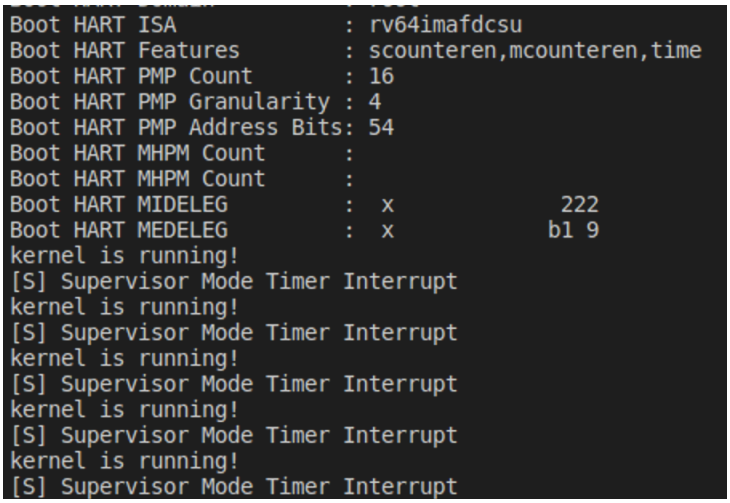
之前我们在Lab1已经使用过了 `sbi_ecall` 的打印字符串，所以在这里是同样的道理，只不过使用 `extension` 为 `0x00` 来对是时钟相关寄存器进行设置。

```
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
    //FunctionID = 0, ExtentionID = 0
    sbi_ecall(0x0, 0x0, next, 0,0,0,0,0);
}
```

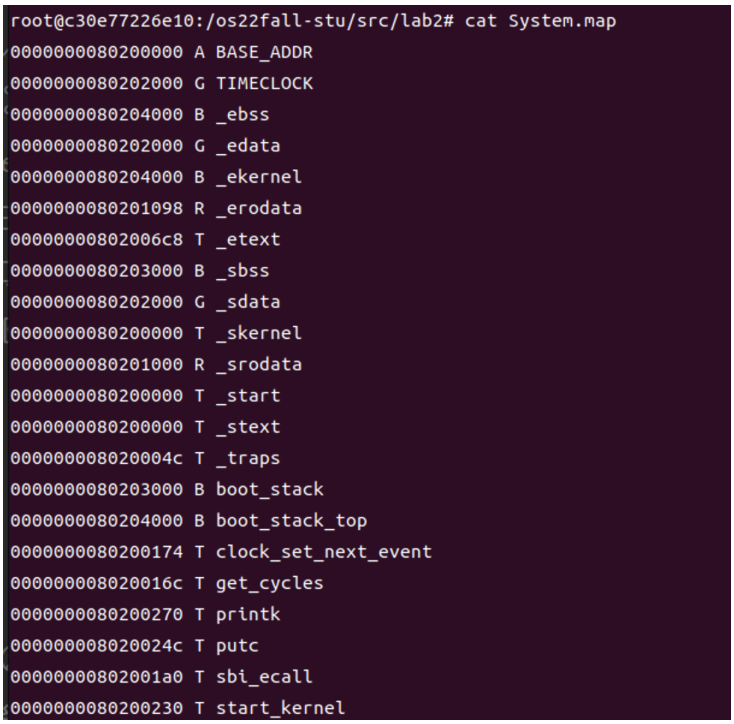
3. 实验结果

3.1 运行方式

在`lab2`文件夹下输入指令`make run`，终端将输出结果如下：



使用指令`nm vmlinux`或`cat System.map`查看自定义符号的值



思考题

首先通过搜索初步了解到这两个寄存器的名称：

0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.

之后进一步搜索，了解到两个寄存器的作用分别是将异常/终端委派给更低的特权模式进行处理，被置为1的位说明该种中断/异常被委派给了低特权模式而不是由M-mode全部处理：

To increase performance, implementations can provide individual read/write bits within `medeleg` and `mideleg` to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (`medeleg`) and machine interrupt

In systems with all three privilege modes (M/S/U), setting a bit in `medeleg` or `mideleg` will delegate the corresponding trap in S-mode or U-mode to the S-mode trap handler. If U-mode traps

由此可以提高操作系统处理中断的效率，减少模式的切换。然而题目要求我们解析OpenSBI相应寄存器的值的含义，查遍了`RISC-V Privileged Spec`后也没有找到相应的表项，甚至更新到了21版本也没啥用，最后竟然在中文手册里找到了：

M 模式还可以通过 medeleg CSR 将同步异常委托给 S 模式。该机制类似于刚才提到的中断委托，但 medeleg 中的位对应的不再是中断，而是图 10.3 中的同步异常编码。例如，置上 medeleg[15]便会把 store page fault（store 过程中出现的缺页）委托给 S 模式。

mideleg（Machine Interrupt Delegation，机器中断委托）CSR 控制将哪些中断委托给 S 模式。与 mip 和 mie 一样，mideleg 中的每个位对应于图 10.3 中相同的异常。例如，mideleg[5]对应于 S 模式的时钟中断，如果把它置位，S 模式的时钟中断将会移交 S 模式的异常处理程序，而不是 M 模式的异常处理程序。

查阅相应表格：

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
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 address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

图 10.3：RISC-V 异常和中断的原因。中断时 mcause 的最高有效位置 1，同步异常时置 0，且低有效位标识了中断或异常的具体原因。只有在实现了监管者模式时才能处理监管者模式中断和页面错误异常（参见第 10.5 节）。（来自[Waterman and Asanovic 2017]中的表 3.6。）

`mideleg`值为`0x0000000000000222`，对应的是`mcause`最高位为1时的选项。其第1/5/9位为1，别的位为0，表明`Supervisor software interrupt`的三种类型中断全部被委派，也就是说在S态下发生这三种中断时，不需要跳转到M态对异常进行处理。这也和lab2对于时钟中断异常在S态下运行我们的处理代码一致（`medeleg[5]`被设为1表明其被委派）。

`medeleg`值为`0x000000000000b109`，其第0/4/8/12/13/15位被置为1，表明`Instrucion address misaligned`/`Load address misaligned`/`Environment call from U-mode`/`Instruction page fault`/`Load page fault`/`Store page fault`这些异常都被委派办理了，不需要切换到M-mode来执行。（似乎与本次实验并没有什么关系，或许会和内存管理有关？）