Lab 2: RV64 时钟中断处理

课程名称：操作系统

指导老师：寿黎但

姓名：

庄毅非 3200105872

胡競文 3200105990

分工：

庄毅非：负责汇编部分 `head.S` 和 `entry.S` ,和一部分报告撰写

胡競文：负责 `trap.c` 和 `clock.c` ，修改 `Makefile` 文件和一部分报告撰写

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. 实验过程

3.1 准备工程

- 将 `print.h` 、 `puti` 、 `puts` 修改为 `printk.h` 、 `printk`
- 依照实验手册修改 `vmlinux.lds` 以及 `head.S`

3.2 开启trap处理

修改 `head.S` 如下：

```
.extern start_kernel

    .section .text.init
```

```
    .globl _start
_start:
    la sp,boot_stack_top
    # ------------------
    # set stvec = _traps
    la t0, _traps
    csrw stvec,t0
    # ------------------
    # enable time interrupt
    # set sie[STIE] = 1
    li t0,0x20
    csrw sie,t0
    # ------------------
    # set mtimecmp register
    # load mtime value
    rdtime a0
    li t1, 10000000
    add a0,t1,a0
    call sbi_set_timer
    # ------------------
    # enable time interrupt
    # set sstatus[SIE] = 1
    csrr t0,sstatus
    ori t0,t0,0x2
    csrw sstatus,t0
    # ------------------
    # call start_kernel
    call start_kernel
    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096 # ⟵ change to your stack size
    .globl boot_stack_top
boot_stack_top:
```

3.3 实现上下文切换

添加并写入 `entry.S` 如下：

```
.section .text.entry
.align 2
.globl _traps
_traps:
    # YOUR CODE HERE
    # -- -- -- -- -- -

    # 1. save 32 registers and sepc to stack
  addi sp,sp,-33*8
  sd x1 ,32*8(sp)
  addi x1,sp,33 * 8
    sd x1 ,31*8(sp)
  sd x3 ,30*8(sp)
  sd x4 ,29*8(sp)
  sd x5 ,28*8(sp)
  sd x6 ,27*8(sp)
  sd x7 ,26*8(sp)
  sd x8 ,25*8(sp)
  sd x9 ,24*8(sp)
  sd x10 ,23*8(sp)
  sd x11 ,22*8(sp)
  sd x12 ,21*8(sp)
  sd x13 ,20*8(sp)
  sd x14 ,19*8(sp)
  sd x15 ,18*8(sp)
  sd x16 ,17*8(sp)
  sd x17 ,16*8(sp)
  sd x18 ,15*8(sp)
  sd x19 ,14*8(sp)
  sd x20 ,13*8(sp)
  sd x21 ,12*8(sp)
```

```
  sd x22 ,11*8(sp)
  sd x23 ,10*8(sp)
  sd x24 ,9*8(sp)
  sd x25 ,8*8(sp)
  sd x26 ,7*8(sp)
  sd x27 ,6*8(sp)
  sd x28 ,5*8(sp)
  sd x29 ,4*8(sp)
  sd x30 ,3*8(sp)
  sd x31 ,2*8(sp)
  # save spec
  csrr t0,sepc
  sd t0,8(sp)



  #-- -- -- -- -- -

  # 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, 1 * 8(sp)
  csrw sepc,t0
  ld x1 ,32*8(sp)
  ld x3 ,30*8(sp)
  ld x4 ,29*8(sp)
  ld x5 ,28*8(sp)
  ld x6 ,27*8(sp)
```

```
66    ld x7 ,26*8(sp)
67    ld x8 ,25*8(sp)
68    ld x9 ,24*8(sp)
69    ld x10 ,23*8(sp)
70    ld x11 ,22*8(sp)
71    ld x12 ,21*8(sp)
72    ld x13 ,20*8(sp)
73    ld x14 ,19*8(sp)
74    ld x15 ,18*8(sp)
75    ld x16 ,17*8(sp)
76    ld x17 ,16*8(sp)
77    ld x18 ,15*8(sp)
78    ld x19 ,14*8(sp)
79    ld x20 ,13*8(sp)
80    ld x21 ,12*8(sp)
81    ld x22 ,11*8(sp)
82    ld x23 ,10*8(sp)
83    ld x24 ,9*8(sp)
84    ld x25 ,8*8(sp)
85    ld x26 ,7*8(sp)
86    ld x27 ,6*8(sp)
87    ld x28 ,5*8(sp)
88    ld x29 ,4*8(sp)
89    ld x30 ,3*8(sp)
90    ld x31 ,2*8(sp)
91    ld sp , 31 * 8(sp)
92    #-- -- -- -- -- -
93
94    # 4. return from trap
95    sret
96    #-- -- -- -- -- -
```

3.4 实现trap处理函数

`scause` 格式如下：

其Interrupt位为**第64位**。

timer interrupt的信息如下：

由图可知，判断timer interrupt需要判断 `scause` **的第64位与[63:0]位**的值。

由此添加并写入 `trap.c` 如下：

```c
#include "clock.h"
#include "printk.h"
void trap_handler(unsigned long scause, unsigned long sepc) {
    // 通过 `scause` 判断trap类型
    // 如果是interrupt 判断是否是timer interrupt
    // 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
    // `clock_set_next_event()` 见 4.5 节
    // 其他interrupt / exception 可以直接忽略
    //printk("scaue:%x\n",scause);
     if((scause >> 63) & 1){// interrupt == 1
        if((scause & 5) == 5){
            //timer interrupt
            printk("kernel is running!\n");
            printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
        }
    }
}
```

3.5 实现时钟中断相关函数

创建并写入 `clock.c` 如下：

```c
// clock.c
```

```c
#include "clock.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 result;
  __asm__ volatile("rdtime %[value]\n" : [value] "=r"(result) : :
"memory");
  return result;
}

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(0x00, 0, next, 0, 0, 0, 0, 0);
}
```

3.6 编译及测试

由于 `printk.h` 等改动，修改 `lib/Makefile` 如下：

```
C_SRC       = $(sort $(wildcard *.c))
OBJ         = $(patsubst %.c,%.o,$(C_SRC))

file = printk.o # change "print.o" to "printk.o"
all:$(OBJ)

%.o:%.c
    ${GCC} ${CFLAG} -c $<
clean:
    $(shell rm *.o 2>/dev/null)
```

然后进行 `make run` ，得到结果如图：

4. 思考题

`medeleg` 和 `mideleg` 是机器的中断和异常委托寄存器，其中记录对某些异常和中断是否进行重定向的信息。

例如若在 `medeleg` 或 `mideleg` 中设置某个位，系统会将 `S-mode` 或 `U-mode` 中的相应陷阱委托给 `S-mode` 的陷阱处理程序，而不需要由 `M-mode` 来处理。

- `medeleg` 为异常保存陷阱委托，是按照trap的发生原因对应到 `msatus` 的相应位置，即目前 `medeleg = 0x000000000000b109` ，是二进制的 `1011000100001001` ，则 `Interrupt = 0, Exception ⩾ 64` 。对照表格如下：

  

  由此可知目前状态为 `Reserved for future standard use` 。
- `mideleg` 为中断保存陷阱委托，寄存器中布局与 `mip` 寄存器中的布局匹配。

  `mip` 布局如下：

  

  其中 `USIP,SSIP` 为低权限软件中断， `UTIP,STIP` 为低权限定时器中断， `UEIP,SEIP` 为低权限外部中断，其余位是只读的，不可写入。

  当前 `mideleg = 0x0000000000000222` ，二进制为 `1000100010` ，其中第 `1、5、9` 位被置为 `1` ，即 `SSIP, STIP, SEIP` 位。

  `SEIP` 位为1表示 `m-mode` 向 `s-mode` 指示外部中断正在挂起； `STIP` 位为1表示 `m-mode` 将定时器中断 (supervidor timer interrupt)委托给 `s-mode` ； `SSIP` 位为

1表示可以通过 `CSR` 来访问更高权限级别的代码读取和写入。