

浙江大学实验报告

Lab 2: Rlinux 时钟中断处理

课程名称: 操作系统 实验类型: 综合

实验项目名称: Rlinux 时钟中断处理

学生姓名: 汪辉 专业: 计算机科学与技术 学号: 3190105609

同组学生姓名: 个人实验 指导老师: 季江民

电子邮件: 3190105609@zju.edu.cn

实验地点: 曹西503 实验日期: 2021 年 11 月 4 日

分工说明: lab 2内容组员两人都各自完成了所有内容。

1 实验目的

1. 学习 RISC-V 的异常处理相关寄存器与指令, 完成对异常处理的初始化。
2. 理解 CPU 上下文切换机制, 并正确实现上下文切换功能。
3. 编写异常处理函数, 完成对特定异常的处理。
4. 调用 OpenSBI 提供的接口, 完成对时钟中断事件的设置。

2 实验内容

完善代码

本实验需要添加、修改以下内容:

1. arch/riscv/kernel/head.S
2. arch/riscv/kernel/vmlinux.lds
3. arch/riscv/kernel/entry.S
4. arch/riscv/kernel/trap.c
5. arch/riscv/kernel/clock.c

修改 vmlinux.lds

按照实验指导, 加入 .text.init:

```
.text : ALIGN(0x1000){
    _stext = .;
    *(.text.init)    /* lab2 */
    *(.text.entry)
    *(.text .text.*)
    _etext = .;
}
```

lab 2中之后 start 会被放在 init 部分, 而中断处理的部分则被放在 .text.entry。

修改 head.S

lab 1中的 head.S 只需将 kernel 启动起来，在lab 2中， kernel 启动之前，设置用到的 CSR 寄存器，并且开始第一次中断。

```
# set stvec = _traps
    la t0, _traps # t0 = _traps1
    csrw stvec, t0

# set sie[STIE] = 1
    addi t1, x0, 32
    csrs sie, t1

# set first time interrupt
    rdttime a0
    li t0, 100000000
    add a0, a0, t0
    addi a1, x0, 0
    addi a2, x0, 0
    addi a3, x0, 0
    addi a4, x0, 0
    addi a5, x0, 0
    addi a6, x0, 0
    addi a7, x0, 0
    ecall

# set sstatus[SIE] = 1
    csrsi sstatus, 2
```

添加 entry.S

entry.S 实现启用中断并完成保护上下文的任务：

1. 保存寄存器内容，将上下文保护在内存（栈）中

```
# -----
# 1. save 32 registers and sepc to stack
csrw sscratch, sp # save the initial sp in sscratch
addi sp, sp, -33 * 8 # 一个寄存器占据8个字节

sd x0, 0*8(sp)
sd x1, 1*8(sp)
sd x3, 3*8(sp)
sd x4, 4*8(sp)
sd x5, 5*8(sp)
sd x6, 6*8(sp)
sd x7, 7*8(sp)
sd x8, 8*8(sp)
sd x9, 9*8(sp)
sd x10, 10*8(sp)
sd x11, 11*8(sp)
sd x12, 12*8(sp)
sd x13, 13*8(sp)
sd x14, 14*8(sp)
sd x15, 15*8(sp)
sd x16, 16*8(sp)
sd x17, 17*8(sp)
```

```

sd x18, 18*8(sp)
sd x19, 19*8(sp)
sd x20, 20*8(sp)
sd x21, 21*8(sp)
sd x22, 22*8(sp)
sd x23, 23*8(sp)
sd x24, 24*8(sp)
sd x25, 25*8(sp)
sd x26, 26*8(sp)
sd x27, 27*8(sp)
sd x28, 28*8(sp)
sd x29, 29*8(sp)
sd x30, 30*8(sp)
sd x31, 31*8(sp)
# RISC-V不能直接从CSR写到内存，需要csrr把CSR读取到通用寄存器，再从通用寄存器sd到内存
csrrw s0, sscratch, x0 # load sscratch which is the initial sp in s0 and
sd it
csrr s1, sepc
sd s0, 2*8(sp)
sd s1, 32*8(sp)

```

2. 将sepc和scause作为参数传入并调用trap_handler处理中断

```

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

```

3. handler 处理完成后，恢复寄存器内容（上下文）

```

# -----
# 3. restore sepc and 32 registers (x2(sp) should be restore last) from
stack
ld s1, 32*8(sp)
csrw sepc, s1 # restore sepc

ld x0, 0*8(sp)
ld x1, 1*8(sp)
ld x3, 3*8(sp)
ld x4, 4*8(sp)
ld x5, 5*8(sp)
ld x6, 6*8(sp)
ld x7, 7*8(sp)
ld x8, 8*8(sp)
ld x9, 9*8(sp)
ld x10, 10*8(sp)
ld x11, 11*8(sp)
ld x12, 12*8(sp)
ld x13, 13*8(sp)
ld x14, 14*8(sp)
ld x15, 15*8(sp)
ld x16, 16*8(sp)
ld x17, 17*8(sp)
ld x18, 18*8(sp)

```

```

ld x19, 19*8(sp)
ld x20, 20*8(sp)
ld x21, 21*8(sp)
ld x22, 22*8(sp)
ld x23, 23*8(sp)
ld x24, 24*8(sp)
ld x25, 25*8(sp)
ld x26, 26*8(sp)
ld x27, 27*8(sp)
ld x28, 28*8(sp)
ld x29, 29*8(sp)
ld x30, 30*8(sp)
ld x31, 31*8(sp)

ld x2, 2*8(sp) # restore sp(x2) last

```

4. 从 trap 中返回

```

# -----
# 4. return from trap
sret
# -----

```

添加 trap.c

```

#include "printk.h"
// trap.c
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
    if ((scause & 0x8000000000000000) == 0x8000000000000000) // interrupt
    {
        /* code */
        if ( (scause<<1) >>1 == 5 ) // user timer interrupt
        {
            printk("[S] Supervisor Mode Timer Interrupt\n") ;
            clock_set_next_event() ;
        }
        // else , ignored
    }
    // else // exception
    // ignored

    // # YOUR CODE HERE
}

```

添加 clock.c

```
// clock.c
#include "sbi.h"
#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 rdttime 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 ret ;
    __asm__ volatile ("rdtime %[ret]" : [ret]="r"(ret)) ;
    return ret ;
    // # YOUR CODE HERE
}
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(0,0,next,0,0,0,0,0) ;
    // # YOUR CODE HERE
}
```

使用 printk.c

lab 2 提供 printk 函数来打印内容，其格式与 printf 一致。将 lab 1 中自己实现的 puti、puts 都用 printk 取而代之，除了在 trap.c 以为，main.c 和 test.c 也要用到 printk 打印内容。

```
//main.c
#include "printk.h"
#include "sbi.h"
extern void test();
int start_kernel() {
    printk("%d",2021);
    printk(" Hello RISC-V + 3190105609\n");
    test(); // DO NOT DELETE !!!
    return 0;
}
```

```
//test.c
#include "printk.h"
#include "defs.h"
// Please do not modify
void test() {
    while (1) {
        for ( int i = 0 ; i < 1e8 ; i++ ) ;
        printk("kernel is running!\n");
    }
}
```

实验结果

`make run` 时钟中断并打印显示

运行中, `test` 内每10的8次方次循环后打印一次内容显示当前 "kernel is running", 相当于是定时显示 `kernel` 的运行状态, 这样可以更方便地看出具体的中断效果。

```
Launch the qemu .....
```

```
OpenSBI v0.6
```

```

      _ _ _ _ _
     /   \       /   _   _   \   _   _
    |   |   _ _ _ _ _ | ( _ | | ) | | | | | |
    |   |   ' _ \ / _ \ ' _ \ \ _ \ | |
    |   |   |_) | _/ | | |_) | |_) | |
    \___/| . _/ \___|_| | |___/|___/___|
           | |
           |_|
```

```
Platform Name           : QEMU Virt Machine
```

```
Platform HART Features  : RV64ACDFIMSU
```

```
Platform Max HARTs     : 8
```

```
Current Hart           : 0
```

```
Firmware Base          : 0x80000000
```

```
Firmware Size          : 120 KB
```

```
Runtime SBI Version    : 0.2
```

```
MIDELEG : 0x0000000000000222
```

```
MEDELEG : 0x000000000000b109
```

```
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
```

```
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
```

```
2021 Hello RISC-V + 3190105609
```

```
kernel is running!
```

```
kernel is running!
```

```
kernel is running!
```

```
kernel is running!
```

```
kernel is running!
```

```
[S] Supervisor Mode Timer Interrupt
```

```
kernel is running!
```

```
kernel is running!
```

```
kernel is running!
```

```
kernel is running!
```

```
QEMU: Terminated
```

`make debug` 调试分析

进入 `_traps` 段, 单步运行至 `call trap_handler` 之前, 查看 `sepc` 内容:

```
oslab@05e5bdae83bc: ~/lab2

0x802000cc <_traps+120> sd      t4,232(sp)
0x802000d0 <_traps+124> sd      t5,240(sp)
0x802000d4 <_traps+128> sd      t6,248(sp)
0x802000d8 <_traps+132> csrrw   s0,sscratch,zero
0x802000dc <_traps+136> csrr    s1,sepc
0x802000e0 <_traps+140> sd      s0,16(sp)
0x802000e4 <_traps+144> sd      s1,256(sp)
0x802000e8 <_traps+148> csrr    a0,scause
0x802000ec <_traps+152> csrr    a1,sepc
>0x802000f0 <_traps+156> jal     ra,0x802001f8 <trap_handler>
0x802000f4 <_traps+160> ld      s1,256(sp)
0x802000f8 <_traps+164> csrw    sepc,s1
0x802000fc <_traps+168> ld      zero,0(sp)

remote Thread 1.1 In: _traps L55 PC: 0x802000f0
(gdb) ni
_traps () at entry.S:14
(gdb) info reg sepc
sepc          0x8020028c      2149581452
(gdb) S
```

进入 handler 运行完 trap 重新返回到 _traps 段的 sret 之前，再次查看 sepc 内容：

```
oslab@05e5bdae83bc: ~/lab2

0x80200168 <_traps+276>      ld      t3,224(sp)
0x8020016c <_traps+280>      ld      t4,232(sp)
0x80200170 <_traps+284>      ld      t5,240(sp)
0x80200174 <_traps+288>      ld      t6,248(sp)
0x80200178 <_traps+292>      ld      sp,16(sp)
>0x8020017c <_traps+296>      sret
0x80200180 <get_cycles>      rdtim   a0
0x80200184 <get_cycles+4>      ret
0x80200188 <clock_set_next_event> rdtim   a1
0x8020018c <clock_set_next_event+4> auipc   a2,0x2
0x80200190 <clock_set_next_event+8> ld      a2,-396(a2)
0x80200194 <clock_set_next_event+12> add     a2,a1,a2
0x80200198 <clock_set_next_event+16> li      a7,0

remote Thread 1.1 In: _traps L96 PC: 0x8020017c
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) ni
_traps () at entry.S:96
(gdb) info reg sepc
sepc          0x8020028c      2149581452
(gdb)
```

执行 trap 当中，查看 sstatus 和 scause：

```
oslab@05e5bdae83bc: ~/lab2

0x80200060 <_traps+12> sd      ra,8(sp)
0x80200064 <_traps+16> sd      gp,24(sp)
0x80200068 <_traps+20> sd      tp,32(sp)
0x8020006c <_traps+24> sd      t0,40(sp)
0x80200070 <_traps+28> sd      t1,48(sp)
0x80200074 <_traps+32> sd      t2,56(sp)
0x80200078 <_traps+36> sd      s0,64(sp)
0x8020007c <_traps+40> sd      s1,72(sp)
0x80200080 <_traps+44> sd      a0,80(sp)
0x80200084 <_traps+48> sd      a1,88(sp)
0x80200088 <_traps+52> sd      a2,96(sp)
0x8020008c <_traps+56> sd      a3,104(sp)
0x80200090 <_traps+60> sd      a4,112(sp)

remote Thread 1.1 In: _traps L11 PC: 0x80200054
Run till exit from #0 _traps () at entry.S:14

Breakpoint 1, _traps () at entry.S:11
(gdb) info reg sstatus
sstatus      0x80000000000006120      -9223372036854750944
(gdb) info reg scause
scause       0x80000000000000005      -9223372036854775803
(gdb)
```

trap以外，查看sstatus内容：

```
oslab@05e5bdae83bc: ~/lab2

0x80200288 <test+32> mv      a5,s0
0x8020028c <test+36> addiw   a5,a5,-1
0x80200290 <test+40> bnez    a5,0x8020028c <test+36>
0x80200294 <test+44> mv      a0,s1
0x80200298 <test+48> jal     ra,0x802004ec <printk>
>0x8020029c <test+52> j       0x80200288 <test+32>
0x802002a0 <puts>      lbu     a2,0(a0)
0x802002a4 <puts+4>   beqz    a2,0x802002f4 <puts+84>
0x802002a8 <puts+8>   addi    sp,sp,-16
0x802002ac <puts+12> sd      s0,0(sp)
0x802002b0 <puts+16> sd      ra,8(sp)
0x802002b4 <puts+20> addi    s0,a0,1
0x802002b8 <puts+24> li      a7,0

remote Thread 1.1 In: test L7 PC: 0x8020029c
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) ni
(gdb) finish
Run till exit from #0 _traps () at entry.S:12
test () at test.c:7
(gdb) info reg sstatus
sstatus      0x80000000000006022      -9223372036854751198
(gdb)
```

此时，sstatus[1] 的 SIE 被置位，即关闭了S态下的中断响应。

3 思考题

1. 通过查看 RISC-V Privileged Spec 中的 medeleg 和 mideleg 解释 make run 时输出中 MIDELEG 值的含义。

`mideleg` 是 machine exception delegation register, `mideleg` 是 machine interrupt delegation register。make run 输出中 MIDELEG 值为 0x0000000000000222, 低12位具体为 0x222 即: 0b001000100010。查阅文档: `mideleg[1]` 控制是否将核间中断交给s模式处理; `mideleg[5]` 控制是否将定时中断交给s模式处理、`mideleg[9]` 控制是否将中断控制器控制的中断交给s模式处理。

输出中这三位均为1说明将这三种权限都赋予了s模式。

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。

实验中依次保护了全部32个寄存器。实际上, 根据具体寄存器功能和内容, 可以分析: `x0` 肯定不需保护, 因为 `x0` 为硬件0寄存器, 不被任何指令改变; `x2-4` 需要保护, 涉及到对内存(栈)和线程的管理; `x5-7`、`x28-31` (`t0-6`) 为7个临时寄存器, 考虑到可能在运行中用于临时存放变量, 而 `trap` 程序也可能使用, 需要保护; `x8-9`、`x18-27` (`s0-11`) 为指定受保护的寄存器; `x10-17` (`a0-7`) 为函数入口参数寄存器, 尽管中断程序 `trap` 只传递两个参数 `sepc`、`scause`, 应该保护前两者 `a0`、`a1` 就足够, 但是考虑到在 `trap` 中使用到 `ecall` 系统调用时仍旧更改了全部8个参数寄存器, 除 `ext_id`、`fun_id` 和时钟参数以外的5个参数都置为0, 故最好还是全部保护。

综上, 认为除了 `x0` 以外应该保护其余所有的31个寄存器。

4 心得体会

- 按照实验指导, 明确每一步的目的后, 代码实现并不复杂。
- 操作系统实验与计算机体系结构内容联系紧密, 最近两门课的实验都几种解决了异常和中断的实现, 关于RISC V处理两者的实现有了更深的理解。
- `printk` 函数要求参数实际上为字符串, 是根据C标准库常用的 `printf` 格式完成的简易版。由于最近Java、C++代码写太多, 误以为 `printk` 重载参数为字符串和整型两种具体实现, 导致卡壳很久, 经过助教的层层帮助才明白真相, 感谢助教的耐心解答。