Lab 2: Rinux 时钟中断处理

1 实验目的

- 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。
- 理解 CPU 上下文切换机制，并正确实现上下文切换功能。
- 编写异常处理函数，完成对特定异常的处理。
- 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

- Docker in Lab0

3 背景知识

3.0 前言

在 `lab1` 中我们成功的将一个最简单的 OS 启动起来， 但还没有办法与之交互。我们在课程中讲过操作系统启动之后由**事件**（`event`）驱动，在本次实验中我们将引入一种重要的事件 **异常**， 异常给了 OS 与硬件、软件交互的能力。在 `lab1` 中我们介绍了在 RISC-V 中有三种特权级 (M 态、 S 态、 U 态)， 在Boot阶段，OpenSBI 已经帮我们将 M 态的异常处理进行了初始化，这一部分不需要我们再去实现，因此本次试验我们重点关注 S 态的异常处理。

3.1 RISC-V 中的 Interrupt 和 Exception

3.1.1 什么是 Interrupt 和 Exception

> We use the term **exception** to refer to an unusual condition occurring at run time **associated with an instruction** in the current RISC-V hart. We use the term **interrupt** to refer to an **external asynchronous event** that may cause a RISC-V hart to experience an unexpected transfer of control. We use the term **trap** to refer to **the transfer of control to a trap handler** caused by either an exception or an interrupt.

上述是 RISC-V Unprivileged Spec 1.6 节中对于 `Trap`、 `Interrupt` 与 `Exception` 的描述。总结起来 `Interrupt` 与 `Exception` 的主要区别如下表：

Interrupt	Exception
Hardware generate	Software generate
These are **asynchronous external requests** for service (like keyboard or printer needs service).	These are **synchronous internal requests** for service based upon abnormal events (think of illegal instructions, illegal address, overflow etc).
These are **normal events** and shouldn't interfere with the normal running of a computer.	These are **abnormal events** and often result in the termination of a program

上文中的 `Trap` 描述的是一种控制转移的过程, 这个过程是由 `Interrupt` 或者 `Exception` 引起的。这里为了方便起见，我们在这里约定 `Trap` 为 `Interrput` 与 `Exception` 的总称（注意，在这些名词的翻译甚至名词本身，不同的地方都可能代表不同的意思，但是在RISCV中，就是上述的意思）

> 在下文中 我们用 **异常** 代指 **Trap**

3.1.2 相关寄存器

除了32个通用寄存器之外，RISC-V 架构还有大量的 **控制状态寄存器** `Control and Status Registers(CSRs)`，下面将介绍几个和异常机制相关的重要寄存器。

Supervisor Mode 异常相关寄寄存器:

- `sstatus` (Supervisor Status Register)中存在一个 SIE (Supervisor Interrupt Enable) 比特位，当该比特位设置为 1 时，会对所有的 S 态异常**响应**， 否则将会禁用所有 S 态异常。
- `sie` (Supervisor Interrupt Eable Register)。在 RISC-V 中， `Interrupt` 被划分为三类 `Software Interrupt` ， `Timer Interrupt` ， `External Interrupt` 。在开启了 `sstatus[SIE]` 之后，系统会根据 `sie` 中的相关比特位来决定是否对该 `Interrupt` 进行**处理**。
- `stvec` (Supervisor Trap Vector Base Address Register) 即所谓的"中断向量表基址"。 `stvec` 有两种模式： `Direct 模式` ，适用于系统中只有一个中断处理程序, 其指向中断处理入口函数 （ 本次实验中我们所用的模式 ） 。 `Vectored 模式` ，指向中断向量表， 适用于系统中有多个中断处理程序 （ 该模式可以参考 RISC-V 内核源码 ） 。
- `scause` (Supervisor Cause Register), 会记录异常发生的原因，还会记录该异常是 `Interrupt` 还是 `Exception` 。
- `sepc` (Supervisor Exception Program Counter), 会记录触发异常的那条指令的地址。

Machine Mode 异常相关寄寄存器:

- 类似于 Supervisor Mode， Machine Mode 也有相对应的寄存器，但由于本实验同学不需要操作这些寄存器，故不在此作介绍。

以上寄存器的详细介绍请同学们参考 RISC-V Privileged Spec

3.1.3 相关特权指令

- `ecall`（Environment Call），当我们在 S 态执行这条指令时，会触发一个 `ecall-from-s-mode-exception`，从而进入 M 模式中的中断处理流程(如设置定时器等)；当我们在 U 态执行这条指令时，会触发一个 `ecall-from-u-mode-exception`，从而进入 S 模式中的中断处理流程(常用来进行系统调用)。
- `sret` 用于 S 态异常返回, 通过 `sepc` 来设置 `pc` 的值， 返回到之前程序继续运行。

以上指令的详细介绍请同学们参考 RISC-V Privileged Spec

3.2 上下文处理

由于在处理异常时，有可能会改变系统的状态。所以在真正处理异常之前，我们有必要对系统的当前状态进行保存，在异常处理完成之后，我们再将系统恢复至原先的状态，就可以确保之前的程序继续正常运行。
这里的系统状态通常是指寄存器，这些寄存器也叫做CPU的上下文（`Context`）.

3.3 异常处理程序

异常处理程序根据 `scause` 的值， 进入不同的处理逻辑，在本次试验中我们需要关心的只有 `Superviosr Timer Interrupt`。

3.4 时钟中断

时钟中断需要 CPU 硬件的支持。CPU 以"时钟周期"为工作的基本时间单位，对逻辑门的时序电路进行同步。而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”。下面介绍与时钟中断相关的寄存器以及如何产生时钟中断。

- `mtime` 与 `mtimecmp`（Machine Timer Register）。`mtime` 是一个实时计时器， 由硬件以恒定的频率自增。`mtimecmp` 中保存着下一次时钟中断发生的时间点，当 `mtime` 的值大于或等于 `mtimecmp` 的值，系统就会触发一次时钟中断。因此我们只需要更新 `mtimecmp` 中的值，就可以设置下一次时钟中断的触发点。`OpenSBI` 已经为我们提供了更新 `mtimecmp` 的接口 `sbi_set_timer`（见 `lab1` 4.4节）。
- `mcounteren`（Counter-Enable Registers）。由于 `mtime` 是属于 M 态的寄存器，我们在 S 态无法直接对其读写， 幸运的是 OpenSBI 在 M 态已经通过设置 `mcounteren` 寄存器的 `TM` 比特位， 让我们可以在 S 态中可以通过 `time` 这个**只读**寄存器读取到 `mtime` 的当前值，相关汇编指令是 `rdtime`。

以上寄存器的详细介绍请同学们参考 RISC-V Privileged Spec

4 实验步骤

4.1 准备工程

- 此次实验基于 lab1 同学所实现的代码进行。
- 在 `lab1` 中我们实现的 `puti` `puts` 使用起来较为繁琐，因此在这里我们提供了简化版的 `printk`。从 `repo` 同步以下代码: `stddef.h` `printk.h` `printk.c`，并按如下目录结构放置。**还需要将之前所有 `print.h puti puts` 的引用修改为 `printk.h printk`**。

  `printk` 函数，你可以简单的认为他和我们在C语言中常用的 `printf` 函数提供一样的功能，但是略微有不同。在我们的实现中，你只需要为你可能打印的格式（如 `%s,%d,%x,%c`）提供支持即可。

```
.
├── Makefile
├── arch
├── include
│   ├── printk.h
│   ├── stddef.h
│   └── types.h
├── init
└── lib
    ├── Makefile
    └── printk.c
```

- 修改 `vmlinux.lds` 以及 `head.S`

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< 原先的 vmlinux.lds
...

.text : ALIGN(0x1000){
    _stext = .;

    *(.text.entry)
    *(.text .text.*)

    _etext = .;
}

...

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> 修改之后的 vmlinux.lds
...

.text : ALIGN(0x1000){
    _stext = .;

    *(.text.init)       <- 加入了 .text.init
    *(.text.entry)      <- 之后我们实现 中断处理逻辑 会放置在 .text.entry
    *(.text .text.*)

    _etext = .;
}

...
```

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< 原先的 head.S
extern start_kernel

    .section .text.entry        <- 之前的 _start 放置在 .text.entry section
```

```
    .globl _start
_start:
    ...

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096

    .globl boot_stack_top
boot_stack_top:

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> 修改之后的 head.S
extern start_kernel

    .section .text.init         <- 将 _start 放入.text.init section
    .globl _start
_start:
    ...

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096

    .globl boot_stack_top
boot_stack_top:
```

4.2 开启异常处理

在运行 `start_kernel` 之前，我们要对上面提到的 CSR 进行初始化，初始化包括以下几个步骤：

1. 设置 `stvec` ， 将 `_traps` (`_trap` 在 4.3 中实现) 所表示的地址写入 `stvec` ，这里我们采用 `Direct 模式` , 而 `_traps` 则是中断处理入口函数的基地址。（请仔细阅读RISC-V Privileged Spec中对 `stvec` 格式的介绍）
2. 开启时钟中断，将 `sie[STIE]` 置 1。
3. 设置第一次时钟中断，参考 `clock_set_next_event()` (`clock_set_next_event()` 在 4.5 中介绍) 中的逻辑用汇编实现。
4. 开启 S 态下的中断响应， 将 `sstatus[SIE]` 置 1。

按照下方模版修改 `arch/riscv/kernel/head.S` ， 并补全 `_start` 中的逻辑。

```
.extern start_kernel

    .section .text.init
    .globl _start
_start:
    # YOUR CODE HERE

    # -----------------
```

```
9
10          # set stvec = _traps
11
12      # ------------------
13
14          # set sie[STIE] = 1
15
16      # ------------------
17
18          # set first time interrupt
19
20      # ------------------
21
22          # set sstatus[SIE] = 1
23
24      # ------------------
25
26      # ------------------
27      # - your lab1 code -
28      # ------------------
29
30      ...
```

> Debug 提示： 可以先不实现 stvec 和 first time interrupt， 先关注 sie 和 sstatus 是否设置正确。

4.3 实现上下文切换

我们要使用汇编实现上下文切换机制， 包含以下几个步骤：

1. 在 `arch/riscv/kernel/` 目录下添加 `entry.S` 文件。
2. 保存CPU的寄存器（上下文）到内存中（栈上）。
3. 将 `scause` 和 `sepc` 中的值传入异常处理函数 `trap_handler` (`trap_handler` 在 4.4 中介绍) ，我们将会在 `trap_handler` 中实现对异常的处理。
4. 在完成对异常的处理之后， 我们从内存中（栈上）恢复CPU的寄存器（上下文）。
5. 从 trap 中返回。

按照下方模版修改 `arch/riscv/kernel/entry.S` ， 并补全 `_traps` 中的逻辑。

```
1       .section .text.entry
2       .align 2
3       .globl _traps
4   _traps:
5       # YOUR CODE HERE
6       # -----------
7
8           # 1. save 32 registers and sepc to stack
9
10      # -----------
11
```

```
        # 2. call trap_handler

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

        # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack

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

        # 4. return from trap

    # -----------
```

> Debug 提示： 可以先不实现 call trap_handler， 先实现上写文切换逻辑。通过 gdb 跟踪各个寄存器，确保上下文的 save 与 restore 正确实现并正确返回。

4.4 实现异常处理函数

1. 在 `arch/riscv/kernel/` 目录下添加 `trap.c` 文件。
2. 在 `trap.c` 中实现异常处理函数 `trap_handler()`, 其接收的两个参数分别是 `scause` 和 `sepc` 两个寄存器中的值。

```
// 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

    # YOUR CODE HERE
}
```

4.5 实现时钟中断相关函数

1. 在 `arch/riscv/kernel/` 目录下添加 `clock.c` 文件。
2. 在 `clock.c` 中实现 `get_cycles()` : 使用 `rdtime` 汇编指令获得当前 `time` 寄存器中的值。
3. 在 `clock.c` 中实现 `clock_set_next_event()` : 调用 `sbi_ecall`，设置下一个时钟中断事件。

```
// clock.c

// 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
    # 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
    # YOUR CODE HERE
}

```

4.6 编译及测试

由于加入了一些新的 .c 文件，可能需要修改一些Makefile文件，请同学自己尝试修改，使项目可以编译并运行。

下面是一个正确实现的输出样例。（仅供参考）

```
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
```

思考题

1. 在我们使用make run时， OpenSBI 会产生如下输出:

```
 OpenSBI v0.9
 ____                   _____ ____ _____
/ __ \                 / ____|  _ \_   _|
| |  | |_ __   ___ _ __ | (___ | |_) || |
```

```
| |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
| |__| | |_) |  __/ | | |____) | |_) || |_
\____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

......

Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

......
```

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

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。

实验任务与要求

- 请各位同学独立完成作业，任何抄袭行为都将使本次实验判为0分。
- 请学习基础知识，并按照实验步骤指导完成实验，撰写实验报告。实验报告的要求：
  - 各实验步骤的截图以及结果分析
  - 实验结束后的心得体会
  - 对实验指导的建议（可选）

浙江大学实验报告

课程名称：　操作系统

实验项目名称：

学生姓名：　　学号：

电子邮件地址：

实验日期：　年　月　日

一、实验内容

\# 记录实验过程并截图，对每一步的命令以及结果进行必要的解释

二、思考题

\# 这里回答实验思考题有关的内容

三、讨论、心得（20分）

\# 在这里写：实验过程中遇到的问题及解决的方法，你做本实