

Lab4 中断

一、实验概述

理解中断机制，在我们最小化内核的基础上，增加对中断的支持。完成断点中断的处理。

二、实验目的

1. 了解CPU的中断机制
2. 了解RISC-V架构是如何支持CPU中断的
3. 掌握与软件相关的中断处理

三、实验项目整体框架概述

```
. //lab 目录
├── kern
│   ├── debug
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   └── intr.c //这里提供了设置中断使能位的接口（其实只封装了一句riscv指令）
│   ├── init
│   │   ├── entry.S
│   │   └── init.c : //需要调用中断机制的初始化函数。
│   ├── libs
│   ├── mm
│   └── trap
│       ├── trap.c //根据中断类型进行不同中断的处理，并且进行中断初始化
│       ├── trapentry.S //中断入口点
│       └── trap.h
├── libs
├── Makefile
├── tools
│   ├── function.mk
│   └── kernel.ld
```

重要文件介绍：

`kern/driver/intr.c(h)`：中断也需要CPU的硬件支持，这里提供了设置中断使能位的接口（其实只封装了一句riscv指令）。

`kern/init/init.c`：需要调用中断机制的初始化函数。

`kern/trap/trapentry.S`：我们把中断入口点设置为这段汇编代码。这段汇编代码把寄存器的数据挪来挪去，进行上下文切换。

`kern/trap/trap.c(h)`：分发不同类型的中断给不同的handler, 完成上下文切换之后对中断的具体处理，例如外设中断要处理外设发来的信息，时钟中断要触发特定的事件。中断处理初始化的函数也在这里，主要是把中断向量表(stvec)设置成所有中断都要跳到 `trapentry.S` 进行处理。

四、实验内容

1. 根据实验指导书理解lab代码
2. 阅读RISC-V手册中有关中断的部分
3. 完成实验练习

五、实验流程及相关知识点

第一步. 添加代码

在/kern/init/init.c的cputs();下面添加代码

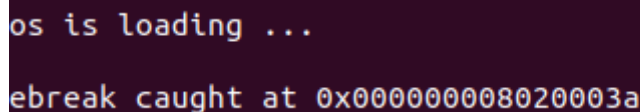
```
idt_init();

intr_enable();

asm volatile("ebreak");
```

第二步. 观察执行结果

执行 `make qemu`



```
os is loading ...
ebreak caught at 0x000000008020003a
```

ebreak指令会产生一个断点中断信号。在这次实验中我们会对断点中断信号进行响应，抓到ebreak中断后打印出一行信息。

第三步. 中断实现

系统首先通过idt_init()函数初始化中断响应相关的寄存器，之后在intr_enable()函数中使能中断响应（中断可以被禁止响应），经过以上步骤，一旦发生中断，系统会在抓到中断后先进行现场保护，然后自动跳转至中断响应函数所在的代码入口地址（存在相关寄存器中）继续执行对应的代码，执行完对应代码后再回到中断响应时的现场。实现的具体过程如下：

idt_init(); 函数实现如下

```
void idt_init(void) {
    extern void __alltraps(void);
    /* Set sscratch register to 0, indicating to exception vector that we are
     * presently executing in the kernel */
    write_csr(sscratch, 0);
    /* Set the exception vector address */
    write_csr(stvec, &__alltraps);
}
```

这点代码是对中断功能进行初始化。首先会使用extern关键字引用外部函数，这个函数在trapentry.s文件中实现，这个函数的地址就是我们的中断响应的入口地址，在第三步中我们会把中断入口地址写入stvec寄存器。第二句话是写sscratch寄存器，值为0，标志着目前我们的操作系统全部处于内核态。如果sscratch在用户态，则sscratch保存内核栈的地址；在内核态，sscratch的值为0。

intr_enable(); 函数，主要是设置标志位，开启中断响应。

中断处理需要初始化，所以我们在 `init.c` 里调用一些初始化的函数

```
//kern/driver/intr.c
#include <intr.h>
#include <riscv.h>
/* intr_enable - enable irq interrupt, 设置sstatus的Supervisor中断使能位 */
void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
/* intr_disable - disable irq interrupt */
void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }
```

`asm volatile("ebreak");` 这是一句内联汇编，在c文件中插入一句汇编代码 `ebreak` 触发一个断点中断。当中断触发后，cpu会寻找 `stvec` 寄存器中的值，然后跳转到这个位置进行中断处理。依次是保存上下文，中断处理，恢复上下文。接下来我们会讲解这些过程。首先我们需要了解一下RISC-V中断的相关知识。

了解RISC-V中断相关

在RISC-V里，中断(interrupt)和异常(exception)统称为"trap"。

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)**。其中有几个重要的寄存器和中断机制有关。

有些时候，禁止CPU产生中断很有用。（就像你在做重要的事情，如操作系统lab的时候，并不想被打断）。所以，`sstatus` 寄存器(Supervisor Status Register)里面有一个二进制位 `SIE` (supervisor interrupt enable，在RISC-V标准里是 2^1 对应的二进制位)，数值为0的时候，如果当程序在S态运行，将禁用全部中断。（对于在U态运行的程序，`SIE`这个二进制位的数值没有任何意义），`sstatus` 还有一个二进制位 `UIE` (user interrupt enable)可以在置零的时候禁止用户态程序产生中断。

在中断产生后，应该有个**中断处理程序**来处理中断。CPU怎么知道中断处理程序在哪？实际上，RISC-V架构有个CSR叫做 `stvec` (Supervisor Trap Vector Base Address Register)，即所谓的“中断向量表基址”。中断向量表的作用就是把不同种类的中断映射到对应的中断处理程序。如果只有一个中断处理程序，那么可以让 `stvec` 直接指向那个中断处理程序的地址。

对于RISC-V架构，`stvec` 会把最低位的两个二进制位用来编码一个“模式”，如果是“00”就说明更高的 `SXLEN-2` 个二进制位存储的是唯一的**中断处理程序的地址**(`SXLEN`是 `stval` 寄存器的位数)，如果是“01”说明更高的 `SXLEN-2` 个二进制位存储的是**中断向量表基址**，通过不同的异常原因来索引中断向量表。但是怎样用62个二进制位编码一个64位的地址？RISC-V架构要求这个地址是四字节的，总是在较高的62位后补两个0。

手册P110

机器和监管者自陷向量 (trap-vector) 基地址寄存器 (`mtvec`和 `stvec`) CSR。他们是位宽为 `XLEN` 的读 / 写寄存器，用于保存自陷向量的配置，包括向量基址 (`BASE`) 和向量模式 (`MODE`) 。 `BASE` 域中的值必须按 4 字节对齐。 `MODE = 0` 表示所有异常都把 `PC` 设置为 `BASE`。 `MODE = 1` 会在一部中断时将 `PC` 设置为 `(BASE+(4×cause))`。

当我们触发中断进入 S 态进行处理时，**以下寄存器会被硬件自动设置**，将一些信息提供给中断处理程序：

sepc(supervisor exception program counter)，它会记录触发中断的那条指令的地址；

scause，它会记录中断发生的原因，还会记录该中断是不是一个外部中断；

stval，它会记录一些中断处理所需要的辅助信息，比如指令获取(instruction fetch)、访存、缺页异常，它会把发生问题的目标地址或者出错的指令记录下来，这样我们在中断处理程序中就知道处理目标了。

特权指令

RISCV支持以下和中断相关的特权指令：

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 态中断返回到 U 态，实际作用为 $pc \leftarrow sepc$ ，回顾**sepc**定义，返回到通过中断进入 S 态之前的地址。

ebreak(environment break)，执行这条指令会触发一个断点中断从而进入中断处理流程。

mret，用于 M 态中断返回到 S 态或 U 态，实际作用为 $pc \leftarrow mepc$ ，回顾**mepc**定义，返回到通过中断进入 M 态之前的地址。（一般不用涉及）

接下来我们进入__alltraps函数：

中断入口点

trapentry.S 这个中断入口点的作用是保存和恢复上下文，并把上下文包装成结构体送到trap函数那里去。

```
.globl __alltraps

.align(2) #中断入口点 __alltraps必须四字节对齐
__alltraps:
    SAVE_ALL #保存上下文

    move a0, sp #传递参数。
    #按照RISCV calling convention, a0寄存器传递参数给接下来调用的函数trap。
    #trap是trap.c里面的一个C语言函数，也就是我们的中断处理程序
    jal trap
    #trap函数指向完之后，会回到这里向下继续执行__trapret里面的内容，RESTORE_ALL,sret

.globl __trapret
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret
```

SAVE_ALL 和 RESTORE_ALL 是在trapentry.S定义的两个宏，是完成保存上下文的功能。保存完之后，我们会进入到具体的中断处理，这里会调用trap函数，执行完之后会进行恢复现场，中断处理结束。

中断时上下文的处理

我们已经知道,在发生中断的时候, CPU会跳到 stvec.我们准备采用 Direct 模式,也就是只有一个中断处理程序, stvec 直接跳到中断处理程序的入口点,那么需要我们对 stvec 寄存器做初始化。

上下文

中断的处理需要“放下当前的事情但之后还能回来接着之前往下做”，对于CPU来说，实际上只需要把原先的寄存器保存下来，做完其他事情把寄存器恢复回来就可以了。这些寄存器也被叫做CPU的**context(上下文, 情境)**。

我们要用汇编实现上下文切换(context switch)机制，这包含两步：

- 保存CPU的寄存器（上下文）到内存中（栈上）
- 从内存中（栈上）恢复CPU的寄存器

为了方便我们组织上下文的数据（几十个寄存器），我们定义一个结构体。

```
// kern/trap/trap.h
#ifndef __KERN_TRAP_TRAP_H__
#define __KERN_TRAP_TRAP_H__

#include <defs.h>

struct pushregs {
    uintptr_t zero; // Hard-wired zero
    uintptr_t ra;   // Return address
    uintptr_t sp;   // Stack pointer
    uintptr_t gp;   // Global pointer
    uintptr_t tp;   // Thread pointer
    uintptr_t t0;   // Temporary
    uintptr_t t1;   // Temporary
    uintptr_t t2;   // Temporary
    uintptr_t s0;   // Saved register/frame pointer
    uintptr_t s1;   // Saved register
    uintptr_t a0;   // Function argument/return value
    uintptr_t a1;   // Function argument/return value
    uintptr_t a2;   // Function argument
    uintptr_t a3;   // Function argument
    uintptr_t a4;   // Function argument
    uintptr_t a5;   // Function argument
    uintptr_t a6;   // Function argument
    uintptr_t a7;   // Function argument
    uintptr_t s2;   // Saved register
    uintptr_t s3;   // Saved register
    uintptr_t s4;   // Saved register
    uintptr_t s5;   // Saved register
    uintptr_t s6;   // Saved register
    uintptr_t s7;   // Saved register
    uintptr_t s8;   // Saved register
    uintptr_t s9;   // Saved register
    uintptr_t s10;  // Saved register
    uintptr_t s11;  // Saved register
    uintptr_t t3;   // Temporary
    uintptr_t t4;   // Temporary
    uintptr_t t5;   // Temporary
    uintptr_t t6;   // Temporary
};

struct trapframe {
    struct pushregs gpr;
    uintptr_t status; //sstatus
    uintptr_t epc; //sepc
    uintptr_t badvaddr; //sbadvaddr
    uintptr_t cause; //scause
};
```

```
};

void trap(struct trapframe *tf);
```

C语言里面的结构体，是若干个变量在内存里直线排列。也就是说，一个 `trapFrame` 结构体占据 36 个 `uintptr_t` 的空间（在 64 位 RISC-V 架构里我们定义 `uintptr_t` 为 64 位无符号整数），里面依次排列通用寄存器 `x0` 到 `x31`，然后依次排列 4 个和中断相关的 CSR，我们希望中断处理程序能够利用这几个 CSR 的数值。

保存上下文

保存上下文是用汇编语言实现的。首先我们定义一个汇编宏 `SAVE_ALL`，用来保存所有寄存器到栈顶（实际上把一个 `trapFrame` 结构体放到了栈顶）。

```
# kern/trap/trapentry.S
#include <riscv.h>

.macro SAVE_ALL #定义汇编宏

    csrw sscratch, sp #保存原先的栈顶指针到sscratch

    addi sp, sp, -36 * REGBYTES #REGBYTES是riscv.h定义的常量，表示一个寄存器占据几个
字节
    #让栈顶指针向低地址空间延伸 36个寄存器的空间，可以放下一个trapFrame结构体。
    #除了32个通用寄存器，我们还要保存4个和中断有关的CSR

    #依次保存32个通用寄存器。但栈顶指针需要特殊处理。
    #因为我们想在trapFrame里保存分配36个REGBYTES之前的sp
    #也就是保存之前写到sscratch里的sp的值
    STORE x0, 0*REGBYTES(sp)
    STORE x1, 1*REGBYTES(sp)
    STORE x3, 3*REGBYTES(sp)
    STORE x4, 4*REGBYTES(sp)
    STORE x5, 5*REGBYTES(sp)
    STORE x6, 6*REGBYTES(sp)
    STORE x7, 7*REGBYTES(sp)
    STORE x8, 8*REGBYTES(sp)
    STORE x9, 9*REGBYTES(sp)
    STORE x10, 10*REGBYTES(sp)
    STORE x11, 11*REGBYTES(sp)
    STORE x12, 12*REGBYTES(sp)
    STORE x13, 13*REGBYTES(sp)
    STORE x14, 14*REGBYTES(sp)
    STORE x15, 15*REGBYTES(sp)
    STORE x16, 16*REGBYTES(sp)
    STORE x17, 17*REGBYTES(sp)
    STORE x18, 18*REGBYTES(sp)
    STORE x19, 19*REGBYTES(sp)
    STORE x20, 20*REGBYTES(sp)
    STORE x21, 21*REGBYTES(sp)
    STORE x22, 22*REGBYTES(sp)
    STORE x23, 23*REGBYTES(sp)
    STORE x24, 24*REGBYTES(sp)
    STORE x25, 25*REGBYTES(sp)
    STORE x26, 26*REGBYTES(sp)
    STORE x27, 27*REGBYTES(sp)
    STORE x28, 28*REGBYTES(sp)
```

```

STORE x29, 29*REGBYTES(sp)
STORE x30, 30*REGBYTES(sp)
STORE x31, 31*REGBYTES(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STORE到内存
csrrw s0, sscratch, x0
csrr s1, sstatus
csrr s2, sepc
csrr s3, sbadaddr
csrr s4, scause

STORE s0, 2*REGBYTES(sp)
STORE s1, 32*REGBYTES(sp)
STORE s2, 33*REGBYTES(sp)
STORE s3, 34*REGBYTES(sp)
STORE s4, 35*REGBYTES(sp)
.endm #汇编宏定义结束

```

恢复上下文

然后是恢复上下文的汇编宏, 恢复的顺序和当时保存的顺序反过来, 先加载两个CSR, 再加载通用寄存器。

```

# kern/trap/trapentry.S
.macro RESTORE_ALL

LOAD s1, 32*REGBYTES(sp)
LOAD s2, 33*REGBYTES(sp)

# 注意之前保存的几个CSR并不都需要恢复
csrw sstatus, s1
csrw sepc, s2

# 恢复sp之外的通用寄存器, 这时候还需要根据sp来确定其他寄存器数值保存的位置
LOAD x1, 1*REGBYTES(sp)
LOAD x3, 3*REGBYTES(sp)
LOAD x4, 4*REGBYTES(sp)
LOAD x5, 5*REGBYTES(sp)
LOAD x6, 6*REGBYTES(sp)
LOAD x7, 7*REGBYTES(sp)
LOAD x8, 8*REGBYTES(sp)
LOAD x9, 9*REGBYTES(sp)
LOAD x10, 10*REGBYTES(sp)
LOAD x11, 11*REGBYTES(sp)
LOAD x12, 12*REGBYTES(sp)
LOAD x13, 13*REGBYTES(sp)
LOAD x14, 14*REGBYTES(sp)
LOAD x15, 15*REGBYTES(sp)
LOAD x16, 16*REGBYTES(sp)
LOAD x17, 17*REGBYTES(sp)
LOAD x18, 18*REGBYTES(sp)
LOAD x19, 19*REGBYTES(sp)
LOAD x20, 20*REGBYTES(sp)
LOAD x21, 21*REGBYTES(sp)
LOAD x22, 22*REGBYTES(sp)
LOAD x23, 23*REGBYTES(sp)

```

```

LOAD x24, 24*REGBYTES(sp)
LOAD x25, 25*REGBYTES(sp)
LOAD x26, 26*REGBYTES(sp)
LOAD x27, 27*REGBYTES(sp)
LOAD x28, 28*REGBYTES(sp)
LOAD x29, 29*REGBYTES(sp)
LOAD x30, 30*REGBYTES(sp)
LOAD x31, 31*REGBYTES(sp)
# 最后恢复sp
LOAD x2, 2*REGBYTES(sp)
.endm

```

接下来我们到具体的中断处理程序trap中去看一看。

中断处理程序

scause

当处理自陷时，寄存器 `scause` 中被写入一个指示导致自陷的事件的代码。如果自陷由中断引起，则置上中断位。“异常代码”字段包含指示最后一个异常的代码。具体的中断/异常映射关系，见中文手册100页。

处理

trap.c的中断处理函数trap, 实际上把中断处理,异常处理的工作分发给了 `interrupt_handler()`, `exception_handler()`, 这些函数再根据中断或异常的不同类型来处理。

```

// kern/trap/trap.c
/* trap_dispatch - dispatch based on what type of trap occurred */
static inline void trap_dispatch(struct trapframe *tf) {
    //scause的最高位是1, 说明trap是由中断引起的
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
        exception_handler(tf);
    }
}

/* *
 * trap - handles or dispatches an exception/interrupt. if and when trap()
 * returns,
 * the code in kern/trap/trapentry.S restores the old CPU state saved in the
 * trapframe and then uses the iret instruction to return from the exception.
 * */
void trap(struct trapframe *tf) { trap_dispatch(tf); }

```

`interrupt_handler()` 和 `exception_handler()` 的实现还比较简单，只是简单地根据 `scause` 的数值更仔细地分了下类，做了一些输出就直接返回了。switch里的各种case, 如 `IRQ_U_SOFT`, `CAUSE_USER_ECALL`, 是riscv ISA 标准里规定的。我们在 `riscv.h` 里定义了这些常量。

第四步.时钟中断

时钟中断可以理解为每隔一段时间执行一次的程序。即每隔一段时间，会固定触发一次的中断。在时钟中断的处理时，我们可以完成进程调度，维护相关等操作。

时钟中断需要硬件支持。在ucore中实现时钟中断，我们需要使用：OpenSBI提供的 `sbi_set_timer()` 接口，我们可以通过这个接口传入一个时刻，在这个时刻会触发一次时钟中断。`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。

需要注意的一点是，我们需要每隔一定时间就触发一次时钟中断，但是通过 `sbi_set_timer()` 接口每次只能设置一次时钟中断。所以我们在初始化的时候设置第一次时钟中断，在每一次时钟中断的处理中，设置下一次的时钟中断。

时钟中断的准备工作：

```
//libs/sbi.c

//当time寄存器的值为stime_value的时候触发一个时钟中断
void sbi_set_timer(unsigned long long stime_value) {
    sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
}

// kern/driver/clock.c

//volatile告诉编译器这个变量可能在其他地方被瞎改一通，所以编译器不要对这个变量瞎优化
volatile size_t ticks;

//对64位和32位架构，读取time的方法是不同的
//32位架构下，需要把64位的time寄存器读到两个32位整数里，然后拼起来形成一个64位整数
//64位架构简单的一句rdtime就可以了
//__riscv_xlen是gcc定义的一个宏，可以用来区分是32位还是64位。
static inline uint64_t get_time(void) { //返回当前时间
#ifdef __riscv_xlen == 64
    uint64_t n;
    __asm__ __volatile__ ("rdtime %0" : "=r"(n));
    return n;
#else
    uint32_t lo, hi, tmp;
    __asm__ __volatile__ (
        "1:\n"
        "rdtimeh %0\n"
        "rdtime %1\n"
        "rdtimeh %2\n"
        "bne %0, %2, 1b"
        : "=&r"(hi), "=&r"(lo), "=&r"(tmp));
    return ((uint64_t)hi << 32) | lo;
#endif
}
```

时钟中断的初始化：

```
// Hardcode timebase
static uint64_t timebase = 100000;

void clock_init(void) {
    // sie这个CSR可以单独使能/禁用某个来源的中断。默认时钟中断是关闭的
    // 所以我们要在初始化的时候，使能时钟中断
    set_csr(sie, MIP_STIP); // enable timer interrupt in sie
    //设置第一个时钟中断事件
    clock_set_next_event();
    // 初始化一个计数器，每次产生时钟中断tick+1
```

```

    ticks = 0;

    cprintf("setup timer interrupts\n");
}
//设置时钟中断: timer的数值变为当前时间 + timebase 后, 触发一次时钟中断
void clock_set_next_event(void) { sbi_set_timer(get_time() + timebase); }

```

时钟中断的处理:

```

void interrupt_handler(struct trapframe *tf) {
    //...
    switch (cause) {
        //...
        case IRQ_S_TIMER:
            clock_set_next_event(); //发生这次时钟中断的时候, 我们要设置下一次时钟中断
            //完成进程调度等工作
            break;
        //...
    }
}

```

最后在init_main()中打开注释 //clock_init();

六、本节知识点回顾

在本次实验中, 你需要了解以下知识点:

1. RISC-V 的中断机制
2. 如何使用汇编语言保存上下文
3. 中断处理的基本流程
4. 时钟中断

七、下一实验简单介绍

在下一实验中, 我们将进行进程相关实验。