

lzuos_lab2

小组成员：

姓名：张金鹏 学号：320210931971 班级：2021级计算机基地班

姓名：王易博 学号：320210900721 班级：2021级计算机基地班

- lzuos_lab2
 - 前置知识
 - 名词扫盲
 - qemu处理中断
 - plic相关
 - UART & interrup
 - riscv 架构相关
 - 题目
 - 题目一
 - 题目二
 - 题目三
 - 题目四

前置知识

名词扫盲

uart : **Universal Asynchronous Receiver-Transmitter,通用异步收发传输器** (Universal Asynchronous Receiver/Transmitter, 通常称为**UART**) 是一种**异步收发传输器**, 是**电脑硬件**的一部分, 将数据通过**串行通信**进行传输。UART通常用在与其他通信接口 (如**EIA RS-232**) 的连接上。

hart: **通讯协定,Highway Addressable Remote Transducer**,高速可寻址远程传感器

plic: **RISC-V PLIC总结 - 知乎 (zhihu.com)** PLIC(platform-level interrupt controller), **平台级中断控制器**。用来将外部的全局中断请求处理后转至中断目标。**PLIC理论上支持1023个外部中断源和15872个上下文**, 但真实设计实现时一般不需要这么多。

4 mode: user,supervisor,hypervisor,machine

sbi_probe_extension() : 检查是否有某些拓展

__builtin_constant_p(n) : 内建函数, 判断是否为常量, 是返回1

qemu处理中断

- 1.首先QEMU是一个仿真器，是对硬件进行仿真。中断在硬件上的实现其实主要就是一个信号线，一端连接中断控制器（主要是对中断信号进行仲裁，然后把仲裁后的结果汇报给cpu，然后由cpu在执行指令的间隔检测这些信号，当检测到中断时根据中断号和中断向量表跳转到相应的中断处理程序中），一端连接需要触发中断的部件（例如timer、串口、网卡等）。
- 2.QEMU主要是通过qemu_irq这个结构体实现上述一套机制。qemu_irq这个结构体中最重要的一个成员是一个叫做handler的函数指针，用于触发中断。更为详细的说来就是首先要实例化一个qemu_irq中断，实现一个handler函数并赋值给qemu_irq的handler成员。在handler函数中需要根据硬件规格和输入参数设置中断控制器，再通过中断控制器将信号传递给cpu。
- 3.实例化了qemu_irq后还要将给中断挂载到总线上：在qemu中每个总线都有一个irqp的中断指针数组，irqp中的每项指向一个设备的中断，赋值我们初始化好的qemu_irq实例给irqp所指向的中断即完成了中断互联。
- 4.之后就可以使用qemu的中断了。例如对于串口来说，向串口的某段地址中写内容就会触发irq的handler，执行该handler就会把信号传递给中断控制器，中断控制器再把信号传递给cpu，cpu在执行指令间隙check到中断信号就会处理串口所触发的中断。

plic相关

流程：

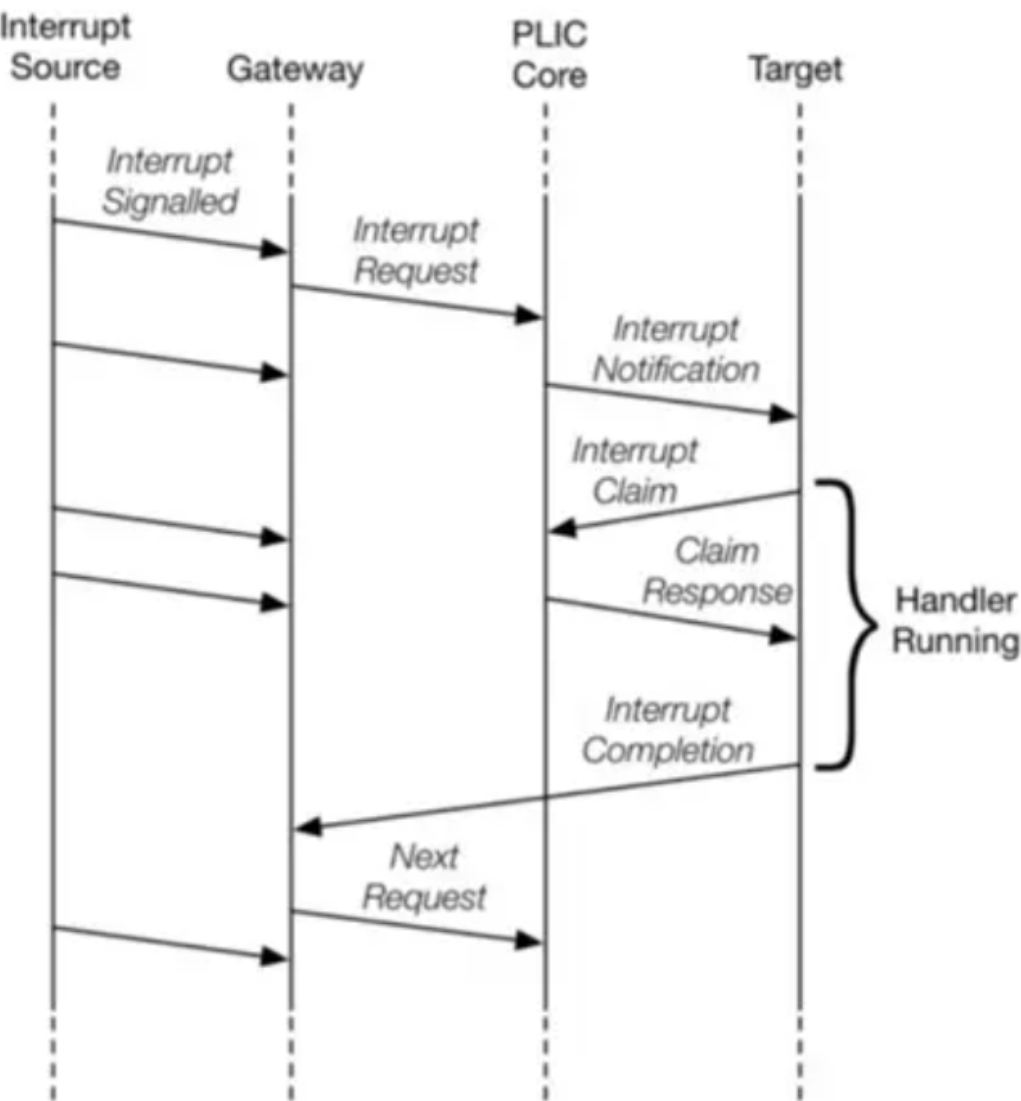


Figure 2. PLIC Interrupt Flow

知乎 @啥是知识

好的，我可以为你解释一下你的代码中的枚举类型，并给出一些例子。你的代码中有一个枚举类型 `qemu_irq`，它表示了 QEMU 虚拟机中的不同中断源。每个中断源都有一个对应的整数值，用来标识它在平台级中断控制器 (PLIC) 中的编号。这些中断源包括：

- `UART_QEMU_IRQ`：UART 串口设备的中断，编号为 10。这个中断源可以用来在虚拟机中实现串口通信，例如通过 `qemu -serial stdio` 命令。
- `RTC_GOLDFISH_IRQ`：Google Goldfish 实时时钟设备的中断，编号为 11。这个中断源可以用来在虚拟机中获取和设置系统时间，例如通过 `hwclock` 命令。
- `VIRTIO_IRQ`：VirtIO 设备的中断，编号为 1 到 8。这些中断源可以用来在虚拟机中支持各种虚拟化设备，例如网络、磁盘、控制台等。
- `VIRTIO_COUNT`：VirtIO 设备的数量，为 8。这个常量可以用来在代码中限制 VirtIO 设备的个数，避免超出 PLIC 的容量。
- `PCIE_IRQ`：PCIe 设备的中断，编号为 32 到 35。这些中断源可以用来在虚拟机中支持 PCIe 总线和设备，例如显卡、声卡等。
- `VIRTIO_NDEV`：虚拟机中最大的中断数量，为 53。这个常量可以用来在代码中分配和管理 PLIC 的内存空间。

VirtIO 设备是一种虚拟化设备，它可以在 QEMU 虚拟机中提供网络、存储、控制台等类型的 IO¹。VirtIO 设备的编号从 1 到 8，分别对应以下设备：

- `VIRTIO_IRQ + 0`：VirtIO 控制台设备，可以用来在虚拟机中实现终端输入输出²。
- `VIRTIO_IRQ + 1`：VirtIO 网络设备，可以用来在虚拟机中实现网络连接³。
- `VIRTIO_IRQ + 2`：VirtIO 块设备，可以用来在虚拟机中实现磁盘访问⁴。
- `VIRTIO_IRQ + 3`：VirtIO 随机数生成器设备，可以用来在虚拟机中实现随机数获取⁵。
- `VIRTIO_IRQ + 4`：VirtIO 内存气球设备，可以用来在虚拟机中实现内存管理⁶。
- `VIRTIO_IRQ + 5`：VirtIO GPU 设备，可以用来在虚拟机中实现图形显示⁷。
- `VIRTIO_IRQ + 6`：VirtIO 输入设备，可以用来在虚拟机中实现键盘和鼠标输入⁸。
- `VIRTIO_IRQ + 7`：VirtIO SCSI 设备，可以用来在虚拟机中实现 SCSI 总线和设备⁹。

UART & interrupt

`UART_QEMU_IRQ` 是一个 QEMU 模拟器中的中断请求号，用于与 UART 设备通信。UART 是一种通用异步收发器 (Universal Asynchronous Receiver/Transmitter)，它可以实现串行数据的发送和接收。UART 的中断请求号是 UART 设备的一个属性，它可以用于在发生某些事件时通知 CPU，例如数据到达、发送完成、错误发生等。UART 的中断请求号需要与中断控制器 (Interrupt Controller) 相连，以便中断控制器根据中断优先级和屏蔽来决定是否响应中断。QEMU 模拟器中的 UART 设备有多种类型，例如 NS16550A、PL011、OMAP 等，它们的中断请求号可能不同，需要根据具体的设备类型和平台来确定。你可以在[这里](#)看到一个 OMAP UART 设备的代码，其中的 `irq` 参数就是 UART 的中断请求号。如果你了解更多关于 QEMU

看相关代码可以知道，这个项目使用的 UART 设备为 NS16550A

riscv 架构相关

SSTATUS_SIE 全局中断使能位 supervisor模式下

<code>csrr rd, csr</code>	<code>csrrs rd, csr, x0</code>	Read CSR
<code>csrw csr, rs</code>	<code>csrrw x0, csr, rs</code>	Write CSR
<code>csrs csr, rs</code>	<code>csrrs x0, csr, rs</code>	Set bits in CSR
<code>csrc csr, rs</code>	<code>csrrc x0, csr, rs</code>	Clear bits in CSR

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR. If *rd*=*x0*, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

For both CSRRS and CSRRC, if $rs1=x0$, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if $rs1$ specifies a register holding a zero value other than $x0$, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate ($uimm[4:0]$) field encoded in the $rs1$ field instead of a value from an integer

register. For CSRRSI and CSRRCI, if the $uimm[4:0]$ field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if $rd=x0$, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

<code>mv rd, rs</code>	<code>addi rd, rs, 0</code>	Copy register
------------------------	-----------------------------	---------------

题目

题目一

1时钟中断的过程分析：从初始化到中断响应过程解释。

clock_init()定义如下：

```
void clock_init()
{
    /* QEMU 的时钟频率为 10MHz, 设置timebase = 100000表示时钟中断频率为100Hz */
    timebase = 100000;
    ticks = 0;
    /* 开启时钟中断 (设置CSR_MIE) */
    set_csr(sie, 1 << IRQ_S_TIMER);
    clock_set_next_event();
    kputs("Setup Timer!");
}
```

第一行的解释已经给出，QEMU 的时钟频率为 10MHz，设置 `timebase = 100000` 表示时钟中断频率为 100Hz，然后是置位寄存器的函数 `set_csr()`，该函数定义如下：


```
#define set_csr(reg,bit) ({ unsigned long __tmp; if (__builtin_constant_p(bit) &&
(unsigned long)(bit) < 32) asm volatile("csrrs %0, " #reg " , %1" : "=r"(__tmp) :
"i"(bit)); else asm volatile("csrrs %0, " #reg " , %1" : "=r"(__tmp) : "r"(bit));
__tmp; })
```

该函数接受两个参数`reg`和`bit`，首先通过gcc内建函数`__builtin_constant_p()`来检查`bit`是否为常量，是常量返回1，否则返回0。因此if语句需要判断`bit`为常量并且其值小于32（通过unsigned long限制了`bit`不能为负数），如果满足该条件则执行第一条汇编指令，将寄存器`reg`的值复制到寄存器`tmp`中，并将立即数`bit`与`reg`的值进行按位或操作，然后将结果写回寄存器`reg`。否则将通用寄存器`bit`的值与寄存器`reg`的值进行按位或操作，并将结果写回寄存器`reg`。最后，返回通用寄存器`__tmp`的值，也就是寄存器`reg`的原始值。

参数`sie`：其作用是提供supervisor mode的全局使能中断。其在`mstatus`寄存器中位置如下图：

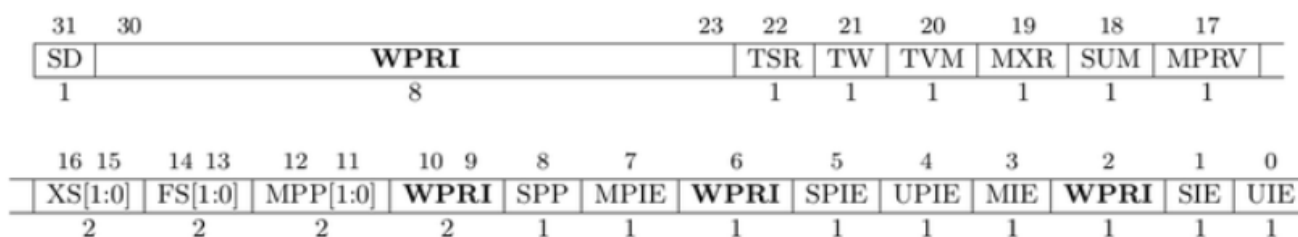


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

`IRQ_S_TIMER`在代码中被定义为5，因此传入的立即数是32。但是`sie`的定义我并没有找到，不知道是怎么回事。(后来根据前人提示，这就是默认的关联变量，用来指代寄存器的名称)

因此`set_csr(sie, 1 << IRQ_S_TIMER)`的功能是：将`sie`寄存器中的第6bit置为1。设置`CSR_MIE`位有效，开启supervisor mode的全局使能中断。

然后执行`clock_set_next_event()`函数，设置下一次时钟中断。相关代码如下：

```
void clock_set_next_event()
{
    sbi_set_timer(get_cycles() + timebase);
}

void sbi_set_timer(uint64_t stime_value)
{
    register uint64_t a0 asm("a0") = stime_value;
    register uint64_t a6 asm("a6") = 0;
    register uint64_t a7 asm("a7") = TIMER_EXTENTION;
    __asm__ __volatile__(
        "ecall\n\t"
        : /* empty output list */
        : "r"(a0), "r"(a6), "r"(a7)
        : "memory");
}

static inline uint64_t get_cycles()
{
    uint64_t n;
```

```

__asm__ __volatile__("rdtime %0" : "=r"(n));
return n;
}

```

首先get_cycles()函数获取开机后经过的时钟数，这是通过rdtime指令得到的,读取64位系统时间寄存器到r中，返回r的值。

sbi_set_timer()函数接收一个64位无符号整数stime_value，在本次调用中传入的是开机后系统经过的时钟周期数与timebase的和，然后将其赋值给a0寄存器，然后将a6寄存器赋值为0，将a7寄存器赋值为TIMER_EXTENTION，已经给出了宏定义。

```

#define TIMER_EXTENTION 0x54494D45

```

然后执行ecall指令，该指令将处理器从用户模式改为特权模式,相当于执行了系统调用。a7寄存器保存的是系统调用号，表示请求的服务类型。a0寄存器保存的是系统调用参数，表示请求的具体内容。

刚才的C语言代码中，ecall指令用来设置定时器，定时器的扩展功能号是0，所以a6寄存器的值被设置为0。a7寄存器的值被设置为TIMER_EXTENTION，这是一个常量，表示定时器的系统调用号。a0寄存器的值被设置为stime_value，这是一个64位无符号整数，表示定时器的超时值。

至此时钟周期的初始化已经完成，时钟周期每达到一个timebase，便会产生一次中断。接下来开始启用中断的过程：

启用中断是由enable_interrupt()完成的，定义如下：

```

#define enable_interrupt() set_csr(sstatus, SSTATUS_SIE)

#define set_csr(reg, bit) \
({ \
    unsigned long __tmp; \
    if (__builtin_constant_p(bit) && (unsigned long)(bit) < 32) \
        asm volatile("csrrs %0, " #reg " , %1" \
            : "=r"(__tmp) \
            : "i"(bit)); \
    else \
        asm volatile("csrrs %0, " #reg " , %1" \
            : "=r"(__tmp) \
            : "r"(bit)); \
    __tmp; \
})

```

实质上就是通过寄存器置位函数对sstatus的SSTATUS_SIE位置1，表示super mode下的时钟中断开启。

处理中断的函数如下：


```

# 代码段开始
.section .text
.globl __alltraps
.align 4
__alltraps:
    # 保存上下文
    SAVE_ALL
    # 将sp存入a0作为trap函数的第一个参数
    mv a0, sp
    # 跳转到trap函数执行（PC+4的值存入了x1寄存器），执行结束后会返回到这里继续向下执行恢复上下文
    call trap
.globl __trapret
__trapret:
    # 恢复上下文
    move sp, a0
    RESTORE_ALL
    # 从内核态中断中返回
    sret

```

```

/**
 * @brief 中断处理函数，从 trapentry.s 跳转而来，做中断类型的检查与分派
 * @param tf 中断保存栈
 */
struct trapframe *trap(struct trapframe *tf)
{
    return trap_dispatch(tf);
}

/**
 * @brief 中断类型的检查与分派
 * @param tf 中断保存栈
 * @todo 清理注释
 */
static inline struct trapframe *trap_dispatch(struct trapframe *tf)
{
    return ((int64_t)tf->cause < 0) ? interrupt_handler(tf) :
exception_handler(tf);
}

/**
 * @brief 中断处理函数，将不同种类中断分配给不同分支
 * @param tf 中断保存栈
 */
struct trapframe *interrupt_handler(struct trapframe *tf)
{
    /** 置cause的最高位为0 */

```

```
int64_t cause = (tf->cause << 1) >> 1;
switch (cause) {
case IRQ_U_SOFT:
    kputs("User software interrupt\n");
    break;
case IRQ_S_SOFT:
    kputs("Supervisor software interrupt\n");
    break;
case IRQ_H_SOFT:
    kputs("Hypervisor software interrupt\n");
    break;
case IRQ_M_SOFT:
    kputs("Machine software interrupt\n");
    break;
case IRQ_U_TIMER:
    kputs("User timer interrupt\n");
    break;
case IRQ_S_TIMER:
    clock_set_next_event();
    if (++ticks % PLANED_TICK_NUM == 0) {
        kprintf("%u ticks\n", ticks);
        if (++ticks / PLANED_TICK_NUM == 100) {
            sbi_shutdown();
        }
    }
    enable_interrupt(); /* 允许嵌套中断 */
    break;
case IRQ_H_TIMER:
    kputs("Hypervisor timer interrupt\n");
    break;
case IRQ_M_TIMER:
    kputs("Machine timer interrupt\n");
    break;
case IRQ_U_EXT:
    kputs("User external interrupt\n");
    break;
case IRQ_S_EXT:
    external_handler(tf);
    break;
case IRQ_H_EXT:
    kputs("Hypervisor external interrupt\n");
    break;
case IRQ_M_EXT:
    kputs("Machine external interrupt\n");
    break;
default:
    print_trapframe(tf);
    break;
}
return tf;
}

void sbi_shutdown()
{
```

```

register uint64_t a0 asm("a0") = 0;
register uint64_t a1 asm("a1") = 0;
register uint64_t a7 asm("a7") = RESET_EXTENTION;
register uint64_t a6 asm("a6") = 0;
__asm__ __volatile__("ecall \n\t"
                    : /* empty output list */
                    : "r"(a0), "r"(a1), "r"(a6), "r"(a7)
                    : "memory");
}

```

每次到达设定时钟周期后，都会跳转到case IRQ_S_TIMER执行分支，在终端上每隔10次ticks打印当前ticks值，如果时钟中断次数（ticks）到达10000次，就通过sbi_shutdown()关机。同时通过结尾enable_interrupt()函数实现了嵌套中断的功能。

题目二

2 键盘中断，当按下Ctrl+r 打印栈帧并重启；当按下CTRL+s 打印栈帧并关机。

当操作系统检测到中断时，进入trapentry.s汇编去保存状态，执行中断操作和恢复状态。

```

.section .text
.globl __alltraps
.align 4
__alltraps:
    # 保存上下文
    SAVE_ALL
    # 将sp存入a0作为trap函数的第一个参数
    mv a0, sp
    # 跳转到trap函数执行（PC+4的值存入了x1寄存器），执行结束后会返回到这里继续向下执行恢复上下文
    call trap
.globl __trapret
__trapret:
    # 恢复上下文
    move sp, a0
    RESTORE_ALL
    # 从内核态中断中返回
    sret

```

保存上下文后跳转到trap.c里的trap函数

```

/**
 * @brief 中断处理函数，从 trapentry.s 跳转而来，做中断类型的检查与分派
 * @param tf 中断保存栈
 */
struct trapframe *trap(struct trapframe *tf)
{
    return trap_dispatch(tf);
}

```

```

}

/**
 * @brief 中断类型的检查与分派
 * @param tf 中断保存栈
 * @todo 清理注释
 */
static inline struct trapframe *trap_dispatch(struct trapframe *tf)
{
    return ((int64_t)tf->cause < 0) ? interrupt_handler(tf) :
exception_handler(tf);
}

```

再跳转到trap_dispatch去判断是异常处理还是中断处理。在这里我们键盘中断跳转到中断处理程序的IRQ_S_EXT中

```

case IRQ_S_EXT:
    external_handler(tf);
    if (flag == 2)
    {
        flag = 0;
        sbi_shutdown();
    }
    else if (flag == 1)
    {
        flag = 0;
        sbi_reboot();
    }
    break;

```

在这里的flag是我们设定判定系统进行重启还是关机的标志符。在external_handler(tf)函数中，进入s态外部中断的具体类型判断和中断处理。

```

static struct trapframe *external_handler(struct trapframe *tf)
{
    uint32_t int_id;
    switch (int_id = plic_claim()) {
    case 1 ... 8:
        kprintf("virtio\n");
        break;
    case UART_SUNXI_IRQ:
    case UART_QEMU_IRQ:
        gobal_tf = tf;
        uart_interrupt_handler();
        break;
    case RTC_GOLDFISH_IRQ:
    case RTC_SUNXI_IRQ:
        rtc_interrupt_handler();

```

```

        kprintf("!!RTC ALARM!!\n");
        set_alarm(read_time() + 1000000000);
        kprintf("timestamp now: %u\n", read_time());
        kprintf("next alarm time: %u\n", read_alarm());
        break;
/* Unsupported interrupt */
default:
    kprintf("Unknown external interrupt: %d\n", int_id);
}
plic_complete(int_id);
return tf;
}

```

在case UART_QEMU_IRQ中，即是我们的键盘中断处理的过程，由于是运行在qemu虚拟机中，我们在uart_qemu.c中找到具体的处理函数。

```

static void uart_16550a_interrupt_handler()
{
    volatile struct uart_qemu_regs *regs = (struct uart_qemu_regs
*)uart_device.uart_start_addr;
    while (regs->LSR & (1 << LSR_DR)) {
        int8_t c = uart_read();
        if (c > -1){
            uart_16550a_putc(c);
            keyboard_interrupt(c);
        }
    }
}

```

```

void keyboard_interrupt(int8_t c){
    if (c == 0x12)
    {
        /*ctrl r*/
        print_trapframe(gobal_tf);
        flag = 1;
    }
    else if (c == 0x13)
    {
        /*ctrl s*/
        print_trapframe(gobal_tf);
        flag = 2;
    }
}

```

在这里判断键盘输入的是ctrl r还是ctrl s来进行相关操作。在处理这里时，我们犯了一个错误，尝试在keyboard_interrupt函数中直接调用sbi_reboot()和sbi_shutdown()进行系统的关机和重启。但此时中断处理函数并未进行完毕，重启会重置中断向量表以及相关内容。plic_complete(int_id)函数也并未执行。我们在处理中发

现，重启并不涉及trap.c里的变量值，例如我不在IRQ_S_EXT中重启前重置flag的值为0，重启后他的值依然是2，导致系统不断的调用sbi_reboot()进行重启。至于为何不会重置trap.c的变量值，查阅相关资料以及询问人工智能均得不到合理的结果。故若未执行plic_complete(int_id)函数，此id并未置为已经处理完成，无法再处理相应中断，也就不再能处理键盘中断。因此我们在keyboard_interrupt函数时，仅仅将flag值置为我们需要操作的值，完成整个external_handler函数后，将对应中断id置为已经处理完成，可以再次处理，再由interrupt_handler(tf)函数进行系统的重启操作，此时中断操作的内容已经完成，故再重启之后也可以触发键盘中断。

题目三

3 仿照断点异常的测试，测试一个其他类型的异常。

由第二题的处理过程可以知道，异常和中断的前期检测过程是一样的，不同的是在trap_dispatch()里会根据当前tf->cause的值进行分发，在RISC v中，trapframe-cause用来保存发生中断或异常的原因，如果最高位为0，则表示发生异常，否则即为中断。因此异常和中断是在这里分开的，由此进入异常处理函数：

```
struct trapframe *exception_handler(struct trapframe *tf)
{
    switch (tf->cause) {
        case CAUSE_MISALIGNED_FETCH:
            kputs("misaligned fetch");
            print_trapframe(tf);
            sbi_shutdown();
            break;
        case CAUSE_FAULT_FETCH:
            kputs("fault fetch");
            print_trapframe(tf);
            sbi_shutdown();
            break;
        case CAUSE_ILLEGAL_INSTRUCTION:
            kprintf("illegal instruction: %p", tf->epc);
            break;
        case CAUSE_BREAKPOINT:
            kputs("breakpoint");
            print_trapframe(tf);
            tf->epc += 2;
            break;
        case CAUSE_MISALIGNED_LOAD:
            kputs("misaligned load");
            print_trapframe(tf);
            sbi_shutdown();
            break;
        case CAUSE_FAULT_LOAD:
            kputs("fault load");
            print_trapframe(tf);
            sbi_shutdown();
            break;
        case CAUSE_MISALIGNED_STORE:
            kputs("misaligned store");
            print_trapframe(tf);
            sbi_shutdown();
```



```

        break;
    case CAUSE_FAULT_STORE:
        kputs("fault store");
        print_trapframe(tf);
        sbi_shutdown();
        break;
    case CAUSE_USER_ECALL:
        kputs("user_ecall");
        break;
    case CAUSE_SUPERVISOR_ECALL:
        kputs("supervisor_ecall");
        print_trapframe(tf);
        sbi_shutdown();
        break;
    case CAUSE_HYPERVISOR_ECALL:
        kputs("hypervisor_ecall");
        print_trapframe(tf);
        sbi_shutdown();
        break;
    case CAUSE_MACHINE_ECALL:
        kputs("machine_ecall");
        print_trapframe(tf);
        sbi_shutdown();
        break;
    case CAUSE_INSTRUCTION_PAGE_FAULT:
        /*
         * kputs("instruction page fault");
         * print_trapframe(tf);
         * sbi_shutdown();
         * break;
         */
    case CAUSE_LOAD_PAGE_FAULT:
        /*
         * kputs("load page fault");
         * print_trapframe(tf);
         * sbi_shutdown();
         * break;
         */
    case CAUSE_STORE_PAGE_FAULT:
        //wp_page_handler(tf);
        break;
    default:
        kputs("unknown exception");
        print_trapframe(tf);
        sbi_shutdown();
        break;
    }
    return tf;
}

```

以下是常用的异常类型编码:

- 0: 指令地址不对齐
- 1: 指令访问错误
- 2: 非法指令
- 3: 断点
- 4: 加载地址不对齐
- 5: 加载访问错误
- 6: 存储地址不对齐
- 7: 存储访问错误
- 8: 环境调用 (从U模式)
- 9: 环境调用 (从S模式)
- 10: 保留
- 11: 环境调用 (从M模式)
- 12: 指令页错误
- 13: 加载页错误
- 14: 保留
- 15: 存储页错误
- 16: 用户软件中断 (N模式)
- 17: 超级用户软件中断 (H模式)
- 18: 虚拟指令错误 (虚拟化扩展)
- 19: 保留
- 20: 虚拟加载错误 (虚拟化扩展)
- 21: 保留
- 22: 虚拟存储错误 (虚拟化扩展)

其中断电异常已经被测试，考虑到由于加载的指令是已经给定的，不太好修改但是可以添加，而且又因为lab2是运行在S态下面的，因此就选择测试`case CAUSE_SUPERVISOR_ECALL`，和`CAUSE_ILLEGAL_INSTRUCTION`。然后为了方便，就直接在主函数里面测试，这两个测试函数分别为`my_illegal_ins()`和`my_ecall()`，这两个函数的位置如下：

```
kprintf("timestamp now: %u\n", read_time());
set_alarm(read_time() + 1000000000);
kprintf("alarm time: %u\n", read_alarm());

enable_interrupt(); // 启用interrupt, sstatus的SSTATUS_SIE位置1

__asm__ __volatile__("ebreak \n\t");
my_illegal_ins();
my_ecall();
kputs("SYSTEM END");

while (1)
    ; /* infinite loop */
return 0;
```

两个用来触发异常的函数如下：

```
void my_illegal_ins()
{
```

```

    __asm__ __volatile__(
        "li a0, 0x12345678\n" // 设置a0寄存器为一个非法的指令地址
        "jalr x0, a0, 0\n" // 跳转到a0寄存器指向的地址, 并将返回地址保存到x0寄存器 (忽略)
    );
    kprintf("This line will not be executed.\n"); // 这一行不会被执行, 因为前面的指令会
    触发异常
    return ;
}

```

运行结果如下:

```

Invalid read at addr 0x12345678, size 2, region '(null)', reason: rejected
fault fetch
trapframe at 0x80214ea0

registers:
zero      0x80214fd0
ra        0x80200144
sp        0x80214fc0
gp        0x0
tp        0x80018000
t0        0x0
t1        0x0
t2        0x1000
s0        0x80214fd0
s1        0x1
a0        0x12345678
a1        0x0
a2        0x80202318
a3        0xa
a4        0xa
a5        0x800000000000006000
a6        0x0
a7        0x1
s2        0x80000000a00006800
s3        0x80200000
s4        0x82200000

```

```

trap information:
status     0x800000000000006120
epc        0x12345678
badvaddr   0x12345678
cause      0x1

```

可以看出成功触发了非法指令异常, 然后是S态ecall异常, 以下是测试代码:

```

void my_ecall()
{
    kprintf("This is you ecall 1 , print a number into stdout:\n");
}

```

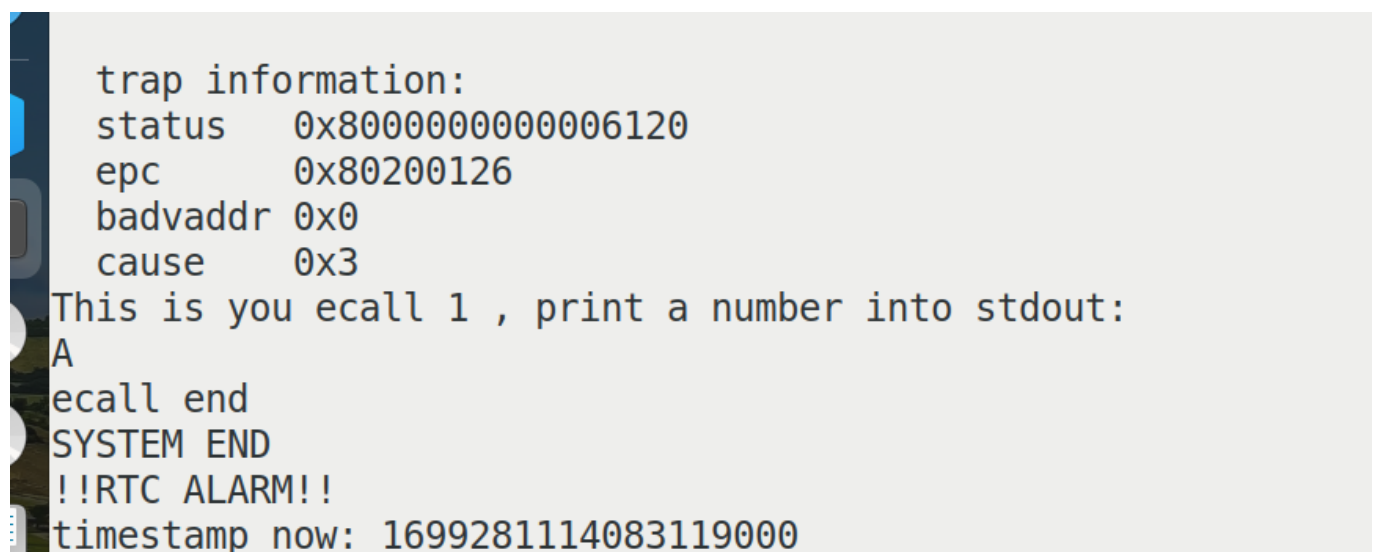
```

    int num = 65; // 定义一个整数变量
__asm__ __volatile__(
    "mv a0, %0\n" // 将整数变量的值移动到a0寄存器中
    "li a7, 1\n" // 设置系统调用编号为1, 表示打印整数
    "ecall\n" // 执行ecall指令, 请求服务
    : // 输出操作数, 无
    : "r"(num) // 输入操作数, 指定寄存器
    : "a0", "a7" // 破坏列表, 指定使用的寄存器
);

kprintf("\necall end\n");
return ;
}

```

运行结果如图:



```

trap information:
status      0x800000000000006120
epc         0x80200126
badvaddr    0x0
cause       0x3
This is you ecall 1 , print a number into stdout:
A
ecall end
SYSTEM END
!!RTC ALARM!!
timestamp now: 1699281114083119000

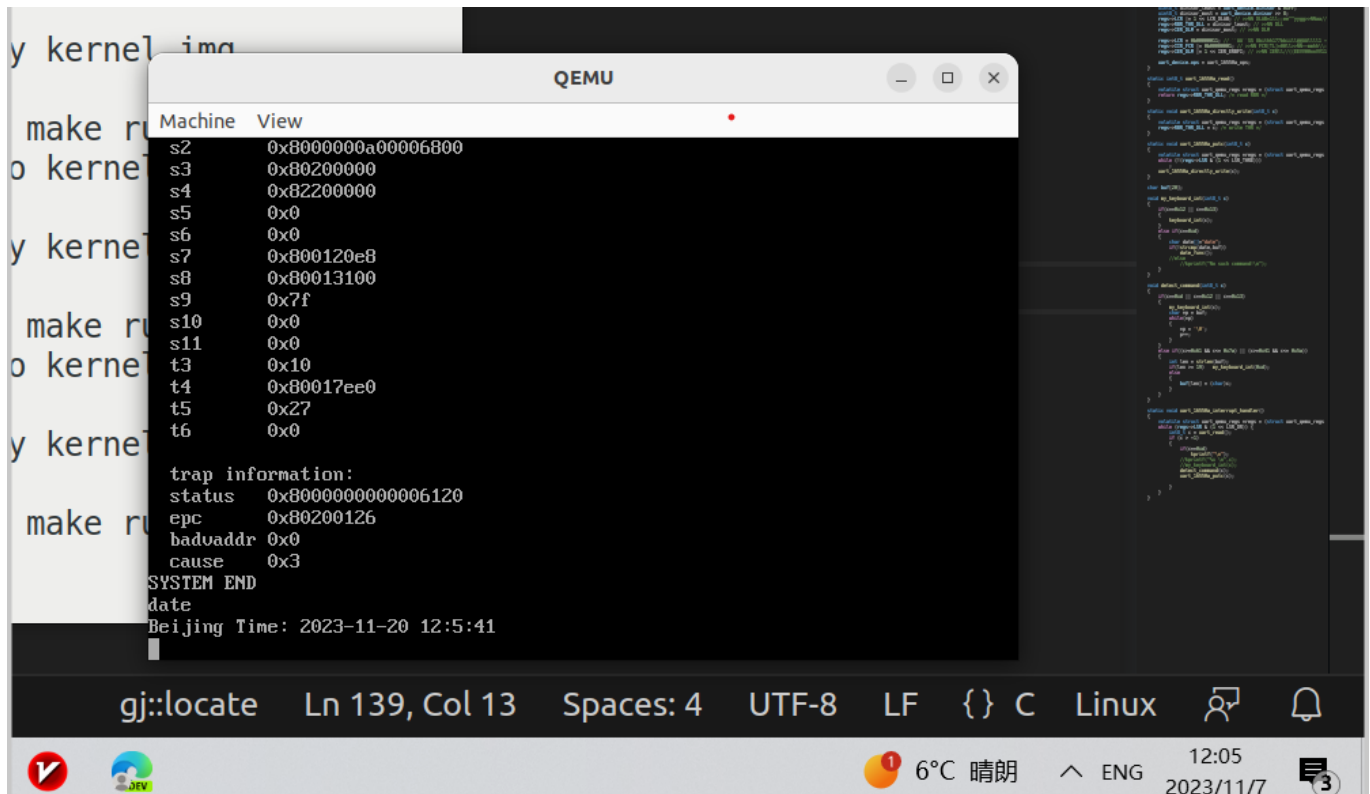
```

可以看出这个异常也能够被正常捕获且执行。

题目四

4利用qemu 模拟器中的 GOLDFISH_RTC, 实现Linux中date命令的功能（输出当前北京时间的年月日时分秒）。

首先看结果演示:



对date命令的处理和在Linux下bash的操作一样，输入date按回车，然后打印当前时区的时间。

思路如下：

首先date和回车的输入仍然属于键盘中断管理的范畴，不过不同于第二题是对单个字符进行处理，这一次需要识别的是“date”字符串，因此需要进行对键盘输入的缓存。即定义一个缓存数组，保存每次从键盘输入的合法字符（在本题中仅认为大小写字母为合法字母），当输入回车时，结束对键盘输入字符的识别，开始进入对读入字符串的处理，由于本题中仅需要实现date功能，因此只需要处理这一个函数即可，如下：

```
char buf[20];

void detect_command(int8_t c)
{
    if(c==0xd || c==0x12 || c==0x13)
    {
        my_keyboard_int(c);
        char *p = buf;
        while(*p)
        {
            *p = '\0';
            p++;
        }
    }
    else if((c>=0x61 && c<= 0x7a) || (c>=0x41 && c<= 0x5a))
    {
        int len = strlen(buf);
        if(len >= 19)    my_keyboard_int(0xd);
        else
        {
            buf[len] = (char)c;
        }
    }
}
```

```

    }
}
}

```

这里复用了第二题的处理函数，即将对 ctrl s 和 ctrl r 的识别集成到 detect_command() 中，将这两个单字符也作为指令识别。如果输入的是回车，则先识别指令，再清空缓存区，如果识别的是合法字符，则将其添加到缓存区。

接下来开始识别并执行输入串：

```

void my_keyboard_int(int8_t c)
{
    if(c==0x12 || c==0x13)
    {
        keyboard_int(c);
    }
    else if(c==0xd)
    {
        char date[]="date";
        if(!strcmp(date,buf))
            date_func();
        //else
        //kprintf("No such command!\n");
    }
}

```

这里便是对第二题的 my_keyboard_int() 进行了改造，打印栈帧并重启关机的功能仍然得到保留，并且在输入字符为回车时识别缓存区的命令，这里则直接用 date[] 来进行字符串比较来识别，如果识别成功则直接跳转到对应的处理函数 date_func()，由于本题中只需要识别一个指令，因此可以这样简单的处理，如果有多个命令的话，则最好使用 function_match_table() 来实现，用函数指针来执行，然后用函数名做匹配，这样可以提高代码的简洁度和可读性。处理函数如下：

```

void date_func()
{
    uint64_t ns = goldfish_rtc_read_time();
    uint64_t sec = ns_to_sec(ns); // 将纳秒转换为秒
    date_t date = sec_to_date(sec); // 将秒转换为日期结构体
    date_to_string(date); // 将日期结构体转换为字符串并打印
}

```

这里使用的都是 <time.h> 下的同名库函数，但因为 lab 中不能使用标准库，于是选择手动实现 <time.h> 的一些功能，并整合至 include 目录下的 time.h 文件中，相关实现如下：

```

typedef struct {
    int year;
    int month;
}

```



```

    int day;
    int hour;
    int minute;
    int second;
} date_t;

#include <kdebug.h>
#define SECONDS_PER_YEAR 31536000
#define SECONDS_PER_DAY 86400
#define SECONDS_PER_HOUR 3600
#define SECONDS_PER_MINUTE 60
#define BEIJING_TIME_OFFSET 8

int days_per_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int is_leap_year(int year) {
    if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) {
        return 1;
    } else {
        return 0;
    }
}

uint64_t ns_to_sec(uint64_t ns) {
    return ns / 1000000000;
}

date_t sec_to_date(uint64_t sec) {
    date_t date;
    sec += BEIJING_TIME_OFFSET * SECONDS_PER_HOUR; // 转换为北京时间
    date.year = 1970 + sec / SECONDS_PER_YEAR; // 计算年份
    sec %= SECONDS_PER_YEAR; // 计算剩余的秒数
    if (is_leap_year(date.year)) { // 判断是否是闰年
        days_per_month[1] = 29; // 闰年二月有29天
    } else {
        days_per_month[1] = 28; // 平年二月有28天
    }
    date.day = 1 + sec / SECONDS_PER_DAY; // 计算天数
    sec %= SECONDS_PER_DAY; // 计算剩余的秒数
    date.month = 0; // 从一月开始
    while (date.day > days_per_month[date.month]) { // 计算月份
        date.day -= days_per_month[date.month]; // 减去当前月的天数
        date.month++; // 进入下一个月
    }
    date.month++; // 月份从1开始
    date.hour = sec / SECONDS_PER_HOUR; // 计算小时
    sec %= SECONDS_PER_HOUR; // 计算剩余的秒数
    date.minute = sec / SECONDS_PER_MINUTE; // 计算分钟
    date.second = sec % SECONDS_PER_MINUTE; // 计算秒钟
    return date;
}

void date_to_string(date_t date) {
    kprintf("Beijing Time: %u-%u-%u %u:%u:%u\n", date.year, date.month, date.day,

```

```
    date.hour, date.minute, date.second);  
}
```

由于时间的获取计算属于算法实现部分，这里不再赘述，其中核心的就是利用了qemu 模拟器中的GOLDFISH_RTC机制，使用`goldfish_rtc_read_time()` 来获取从1970年以来的纳秒时间戳，通过计算得到当前的北京时间。

到此date命令即为实现完毕。