OS_Lab3_report

Lab3：中断

练习一：时钟中断

首先我们编写如下代码完成这个练习：

代码块

```
        case IRQ_S_TIMER:
            /* LAB3 EXERCISE1 - 时钟中断处理任务 */
            /* (1) 设置下次时钟中断 */
            clock_set_next_event();

            /* (2) 计数器（ticks）加一
             * ticks是全局变量，记录总的时钟中断次数
             * 这个变量定义在clock.c中
             */
            ticks++;

            /* (3) 当计数器达到100时，输出提示信息并增加打印计数 */
            if (ticks % TICK_NUM == 0) {
                print_ticks();
            }

            /* (4) 当ticks达到1000时（即打印了10次），关机
             * 这是为了防止测试程序无限运行
             */
            if (ticks == 10 * TICK_NUM) {
                sbi_shutdown();
            }
            break;
```

接下来，我们梳理时钟中断的相关代码和流程：

在初始化时，clock_init() 初始化时钟中断系统，设置定时器中断，为系统提供时间基准，启动周期性的时钟中断。

代码块

```
void clock_init(void) {
    set_csr(sie, MIP_STIP);    // 启用监管者态时钟中断
```

```
    clock_set_next_event();     // 设置第一次时钟中断
    ticks = 0;                  // 初始化计数器
}
```

其中set_csr(sie, MIP_STIP)通过设置sie允许时钟中断，clock_set_next_event设置下次时钟中断的时间点，其具体通告sbi_set_timer函数实现，最后将初始化计数器赋值为0.

idt_init（）初始化中断描述符表。其首先声明外部符号__alltraps（负责保存所有通用寄存器和CSR），然后设置新的栈*指针*将sscratch寄存器设置为0告诉异常向量程序当前正在内核态执行，然后设置stvec寄存器指向__alltraps入口，当异常发生时，硬件会跳转到stvec指向的地址开始执行。

代码块

```
void idt_init(void) {
    extern void __alltraps(void);
    write_csr(sscratch, 0);
    write_csr(stvec, &__alltraps);
}
```

当定时器到期时，硬件会：

1. 自动设置scause为IRQ_S_TIMER
2. 保存当前PC到sepc
3. 跳转到stvec指向的__alltraps函数：保存当前的寄存器组，将当前栈指针（指向trap frame）作为参数传递给trap函数，jal指令保存返回地址到ra，并跳转到trap函数

代码块

```

__alltraps:
    SAVE_ALL

    move  a0, sp
    jal trap
    # sp should be the same as before "jal trap"

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

```

4. trap函数根据trap类型分发处理，trap_dispatch会根据scause区分中断和异常并调用相应的处理函数，处理完成后返回控制权会回到trapentry.S的__trapret，那里会恢复所有保存的寄存器并执行sret返回。

代码块

```
void trap(struct trapframe *tf) {
    trap_dispatch(tf);
}
```

trap_dispatch检查scause的最高位来区分中断和异常

如果cause < 0，说明最高位为1，是中断

如果cause >= 0，说明最高位为0，是异常

代码块

```
static inline void trap_dispatch(struct trapframe *tf) {
    if ((intptr_t)tf->cause < 0) {
        // 最高位为1，表示中断，调用中断处理函数
        interrupt_handler(tf);
    } else {
        // 最高位为0，表示异常，调用异常处理函数
        exception_handler(tf);
    }
}
```

interrupt_handler函数（即我们编写的code）中根据scause寄存器的低63位确认具体的中断或异常编号，若是IRQ_S_TIMER时钟中断，便 设置下次时钟中断，将计数器（ticks）加一，当计数器达到100时，输出提示信息并增加打印计数 ，当ticks达到1000时（即打印了10次），调用sbi_shutdown()关机。

扩展练习一

Q1:

mov a0, sp指令的目的是将当前栈指针作为参数传递给C语言的trap()函数。在RISC-V ABI中，a0是第一个函数参数寄存器。这里sp指向刚刚保存的trap frame结构体，该结构体包含了被中断程序的所有上下文信息，trap()函数需要访问这些信息来决定如何处理异常。

Q2：

寄存器在栈中的位置由trapframe结构体定义确定：

代码块

```
struct trapframe {
    struct pushregs gpr;  // 32个通用寄存器（0-31索引位置）
    uintptr_t status;     // sstatus（32索引位置）
    uintptr_t epc;        // sepc（33索引位置）
    uintptr_t badvaddr;   // sbadaddr（34索引位置）
    uintptr_t cause;      // scause（35索引位置）
};
```

每个位置占用8字节（REGBYTES），所以：

- 通用寄存器保存在0*8(sp)到31*8(sp)
- CSR保存在32*8(sp)到35*8(sp)

这种固定布局确保了C代码能够通过结构体成员访问对应的寄存器值。

Q3：

需要保存所有寄存器，理由如下：

1. 完整性保证：RISC-V规范要求异常处理程序必须保存所有可能被修改的寄存器，以确保被中断程序的执行环境完全恢复
2. ABI要求：RISC-V应用程序二进制接口(ABI)规定异常处理程序需要保存完整的上下文
3. 安全性考虑：保存所有寄存器可以防止任何意外的数据丢失，即使某些中断处理不需要用到某些寄存器
4. 简化设计：统一保存所有寄存器简化了代码逻辑，避免了根据不同中断类型有选择地保存寄存器带来的复杂性和潜在bug
5. 性能考虑：虽然保存所有寄存器有一定开销，但在现代处理器上这点开销是可以接受的，而且比维护复杂的条件保存逻辑更可靠因此，即使某些中断处理可能只需要用到少数寄存器，__alltraps仍然保存完整的执行上下文以确保系统可靠性

扩展练习二

1.csrw sscratch, sp；csrrw s0, sscratch, x0 的操作和目的

代码实现了栈指针的保存和恢复操作，首先将当前的栈指针sp保存到sscratchCSR中，然后通过原子操作将sscratch中的原值（即原来的栈指针）读取到s0寄存器中，同时将x0(值为0)写入sscratch。

目的：

1. 保存原始栈指针：在异常发生时，可能需要知道原始的栈指针位置，用于调试或栈回溯
2. 设置sscratch为已知值：写入0作为标记，表示当前正在内核态处理异常（在用户态时sscratch保存内核栈指针）
3. 为后续保存栈指针到trap frame做准备：s0中的值稍后会被保存到trap frame的sp字段

2.SAVE_ALL中保存stval/scause等CSR但RESTORE_ALL中不恢复的原因

CSR的分类和特性：

需要恢复的CSR（在RESTORE_ALL中恢复）：

- sstatus：状态寄存器，包含特权级、中断使能等信息，必须恢复以确保正确的执行环境
- sepc：异常程序计数器，保存异常发生时的PC，必须恢复以便从正确位置继续执行

不需要恢复的CSR（只保存不恢复）：

- scause：异常原因寄存器，由硬件自动设置，只读，用于指示异常类型
- sbadaddr/stval：坏地址寄存器，由硬件自动设置，只读，用于存储导致异常的地址

保存但不恢复的意义：

1. 调试和分析：保存这些CSR值到trap frame，供C语言处理函数访问和分析异常原因
2. 信息传递：trap frame中的这些值可以被print_trapframe()等调试函数使用，帮助开发者理解异常情况
3. 规范要求：RISC-V规范规定这些CSR是只读的，由硬件管理，不应该被软件修改
4. 避免错误：如果错误地修改了这些CSR，可能导致系统状态不一致或异常处理逻辑混乱

扩展练习三

代码块

```
        case CAUSE_ILLEGAL_INSTRUCTION:

            /* (1) 输出异常类型 */
            cprintf("Exception type:Illegal instruction\n");

            /* (2) 输出异常指令地址
             * sepc指向导致异常的指令地址
             */
            cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);

            /* (3) 更新tf->epc以跳过非法指令
             * RISC-V指令通常是4字节长（压缩指令除外）
             * 将epc增加4，使CPU跳过这条非法指令继续执行
             */
            tf->epc += 4;
            break;

        case CAUSE_BREAKPOINT:
            /* (1) 输出异常类型 */
            cprintf("Exception type: breakpoint\n");

```

```
22              /* (2) 输出断点指令地址 */
23              cprintf("ebreak caught at 0x%08x\n", tf->epc);
24
25              /* (3) 更新tf->epc以跳过断点指令
26               * ebreak指令也是4字节长
27               */
28              tf->epc += 4;
29              break;
```

知识点

在实验中，通过trapentry.S中的汇编代码保存和恢复完整的CPU上下文（即trapframe），并通过trap.c中的trap_dispatch函数根据scause寄存器的值分发处理不同的中断和异常。这直接对应了OS原理中的中断处理核心概念。

其次是定时器中断的实现。实验中通过clock.c里的sbi_set_timer调用，利用SBI（Supervisor Binary Interface）来设置硬件定时器，并在中断处理函数中维护一个全局的ticks计数器。这对应了OS原理中的系统时钟与定时器管理。原理上，系统时钟为进程调度的时间片轮转、任务的延时与超时等机制提供了基础节拍。实验中的实现虽然简单（仅用于计数和演示），但它构建了实现这些高级调度功能所必需的底层硬件驱动和中断服务。

中断的嵌套（Nested Interrupts）与优先级管理是一个关键但缺失的环节。在当前的简单实现中，进入中断处理后，通常会禁用新的中断，直到当前中断处理完成，这是一种扁平化的处理模型。但在复杂的实时或通用操作系统中，中断具有不同的优先级。例如，一个高优先级的时钟中断可能需要能够抢占（preempt）一个正在处理的低优先级的磁盘I/O中断。实现这一点需要更复杂的硬件中断控制器支持和软件设计，包括管理多个中断栈、在中断处理期间有选择地开放更高优先级的中断，以保证系统的实时性和响应性。

中断处理的延迟执行机制（Deferred Work / Bottom Halves） 也是现代内核设计的重要一环。原理上，为了尽快恢复被中断的程序并保持系统响应，中断服务例程（ISR，或称为“顶半部”）应尽可能快地执行，只完成对硬件的紧急操作（如从网卡缓冲区拷贝数据）。而耗时较长的处理（如处理整个网络协议栈）则应被推迟到稍后，在一个可以被中断的、更安全的环境中执行（称为“底半部”）。本实验的中断处理是一体化的，所有工作都在中断上下文中同步完成，这在中断任务复杂时会严重影响系统性能。