实验 3：中断处理与时钟管理

实验目标

通过分析 xv6 的中断处理机制，理解操作系统如何响应硬件事件，实现完整的中断处理框架和时钟中断驱动的任务调度。

核心学习资料

RISC-V 中断机制

- **RISC-V 特权级规范 第 3 章：Machine-Level ISA**
  - 3.1.9 节：Machine Interrupt Registers
  - 3.2.1 节：Machine Timer Registers
- **RISC-V 特权级规范 第 12 章：Supervisor-Level ISA**
  - 12.1.3 节：Supervisor Interrupt Registers
  - 重点理解：mie、mip、sie、sip 寄存器的作用

xv6 中断处理源码分析

- **kernel/trap.c** - 中断和异常处理
  - 重点函数：usertrap()、kerneltrap()、devintr()
  - 学习要点：中断分发、异常处理、系统调用入口
- **kernel/kernelvec.S** - 内核态中断向量
  - 重点：上下文保存和恢复机制
- **kernel/start.c** - 机器模式初始化
  - 重点函数：timer 中断的设置和代理

时钟管理理论

- **SBI 规范**：https://github.com/riscv-non-isa/riscv-sbi-doc
  - 第 4.6 节：Timer Extension
- **操作系统概念 第 5 章**：CPU 调度

任务列表

任务 1：理解 RISC-V 中断架构

学习重点：

1. 分析中断特权级委托：
   - Machine Mode → Supervisor Mode 委托
   - medeleg：异常委托寄存器
   - mideleg：中断委托寄存器
   - 为什么需要中断委托？
   - 哪些中断应该委托给 S 模式？
2. 理解中断寄存器组合：
   - mie/sie：中断使能寄存器
   - mip/sip：中断挂起寄存器
   - mtvec/stvec：中断向量基址
   - mcause/scause：中断原因寄存器

深入思考：

- 时钟中断为什么在 M 模式产生，却在 S 模式处理？
- 如何理解“中断是异步的，异常是同步的”？

任务 2：分析 xv6 的中断处理流程

代码阅读指导：

1. 研读 start.c 中的机器模式设置：

c
```
// 时钟中断委托给 S 模式
w_mideleg(r_mideleg() | (1 << 5));
// 设置机器模式陷阱向量
w_mtvec((uint64)timervec);
```

- 为什么时钟中断需要特殊处理？
- timervec 的作用是什么？

2. 分析 kernelvec.S 的上下文切换：
   - 哪些寄存器需要保存？
   - 为什么不保存所有寄存器？
   - 栈的使用策略是什么？
3. 理解 trap.c 的中断分发：

c
```
void kerneltrap(void) {
    // 中断还是异常？
    // 如何确定中断源？
    // 如何调用相应处理函数？
}
```

关键问题：

- 中断处理中的重入问题如何解决？
- 中断处理时间过长会有什么后果？

任务 3：设计你的中断处理框架

架构设计要求：

1. 设计中断向量表结构
2. 定义中断处理函数接口
3. 实现中断的注册和注销机制

设计考虑：

c
```
// 中断处理函数类型
typedef void (*interrupt_handler_t)(void);

// 中断控制接口
void trap_init(void);                                   // 初始化中断系统
void register_interrupt(int irq, interrupt_handler_t h); // 注册中断处理函数
void enable_interrupt(int irq);                         // 开启特定中断
void disable_interrupt(int irq);                        // 关闭特定中断

// 你需要考虑的问题：
// 1. 如何设计中断优先级？
// 2. 是否支持中断嵌套？
// 3. 如何处理共享中断？
```

实现策略：

1. 先实现最基本的时钟中断处理
2. 逐步添加其他中断源支持

3. 考虑性能和可扩展性

任务 4：实现上下文保存与恢复

参考 xv6 的 kernelvec.S，理解：

1. 哪些寄存器必须保存？
    - 调用者保存寄存器 vs 被调用者保存寄存器
    - 临时寄存器的处理策略
    - CSR 寄存器的保存需求
2. 栈的管理：
    - 中断栈的分配
    - 栈溢出检测
    - 多级中断的栈管理

实现挑战：

asm

```
# 你的中断入口实现框架
kernelvec:
    # 保存上下文
    # 你需要决定：
    # 1. 保存到哪里？内核栈？专用区域？
    # 2. 保存哪些寄存器？
    # 3. 如何快速保存和恢复？

    # 调用 C 处理函数
    call kerneltrap

    # 恢复上下文并返回
```

任务 5：实现时钟中断与调度

时钟中断处理：

1. 理解 SBI 时钟接口：

c

```
// 设置下次时钟中断时间
void sbi_set_timer(uint64 time);
// 获取当前时间
uint64 get_time(void);
```

2. 实现时钟中断处理函数：

c

```
void timer_interrupt(void) {
    // 1. 更新系统时间
    // 2. 处理定时器事件
    // 3. 触发任务调度
    // 4. 设置下次中断时间
}
```

调度器集成：

- 如何在时钟中断中触发调度？
- 调度的时机选择有什么考虑？

- 如何确保调度的原子性？

任务 6：异常处理机制

异常类型理解：

1. 指令地址未对齐
2. 指令访问故障
3. 非法指令
4. 断点
5. 加载地址未对齐
6. 加载访问故障
7. 存储地址未对齐
8. 存储访问故障
9. 用户模式环境调用
10. 监督模式环境调用

实现要求：

c

```
void handle_exception(struct trapframe *tf) {
    uint64 cause = r_scause();
    switch (cause) {
        case 8: // 系统调用
            handle_syscall(tf);
            break;
        case 12: // 指令页故障
            handle_instruction_page_fault(tf);
            break;
        case 13: // 加载页故障
            handle_load_page_fault(tf);
            break;
        case 15: // 存储页故障
            handle_store_page_fault(tf);
            break;
        default:
            panic("Unknown exception");
    }
}
```

测试与调试策略

中断功能测试

c

```
void test_timer_interrupt(void) {
    printf("Testing timer interrupt...\n");

    // 记录中断前的时间
    uint64 start_time = get_time();
    int interrupt_count = 0;
```

```
    // 设置测试标志
    volatile int *test_flag = &interrupt_count;

    // 在时钟中断处理函数中增加计数
    // 等待几次中断
    while (interrupt_count < 5) {
        // 可以在这里执行其他任务
        printf("Waiting for interrupt %d...\n", interrupt_count + 1);
        // 简单延时
        for (volatile int i = 0; i < 1000000; i++);
    }

    uint64 end_time = get_time();
    printf("Timer test completed: %d interrupts in %lu cycles\n",
           interrupt_count, end_time - start_time);
}
```

异常处理测试

```
c
void test_exception_handling(void) {
    printf("Testing exception handling...\n");
    // 测试除零异常（如果支持）
    // 测试非法指令异常
    // 测试内存访问异常
    printf("Exception tests completed\n");
}
```

性能测试

```
c
void test_interrupt_overhead(void) {
    // 测量中断处理的时间开销
    // 测量上下文切换的成本
    // 分析中断频率对系统性能的影响
}
```

调试建议

分阶段调试

1. 基础设置验证：
    - 验证中断寄存器设置是否正确
    - 检查中断向量地址是否对齐
    - 确认中断使能位设置
2. 中断触发测试：
    - 使用简单的时钟中断测试
    - 在中断处理函数中添加输出确认被调用
    - 验证中断频率是否符合预期
3. 上下文完整性：
    - 在中断前后检查寄存器值

    - 验证栈指针的正确性
    - 确认中断返回后程序继续正常执行

常见问题诊断

- **问题：中断无响应**
    - 检查中断使能位设置
    - 验证中断向量地址
    - 确认中断源是否正确配置
- **问题：系统在中断处理后崩溃**
    - 检查栈指针保存和恢复
    - 验证上下文保存的完整性
    - 确认中断处理函数没有破坏调用约定
- **问题：中断频率异常**
    - 检查时钟设置参数
    - 验证 SBI 调用是否正确
    - 确认时间计算没有溢出

思考题

1. 中断设计：
    - 为什么时钟中断需要在 M 模式处理后再委托给 S 模式？
    - 如何设计一个支持中断优先级的系统？
2. 性能考虑：
    - 中断处理的时间开销主要在哪里？如何优化？
    - 高频率中断对系统性能有什么影响？
3. 可靠性：
    - 如何确保中断处理函数的安全性？
    - 中断处理中的错误应该如何处理？
4. 扩展性：
    - 如何支持更多类型的中断源？
    - 如何实现中断的动态路由？
5. 实时性：
    - 当前实现的中断延迟特征如何？
    - 如何设计一个满足实时要求的中断系统？