- 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？

实验5：进程管理与调度

实验目标

通过深入分析xv6的进程管理机制，理解操作系统如何创建、管理和调度进程，实现完整的进程生命周期管理和简单的调度算法。

核心学习资料

进程管理理论基础

- **操作系统概念 第3-5章**：进程、线程、CPU调度
- **xv6手册 第2-4章**：操作系统组织、页表、陷阱和系统调用
- **RISC-V调用约定**：https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

xv6进程管理源码分析

- `kernel/proc.h` - 进程结构体定义
  - 重点：`struct proc` 的字段含义和生命周期
- `kernel/proc.c` - 进程管理核心函数
  - 重点函数：`allocproc()`，`fork()`，`exit()`，`wait()`，`scheduler()`
  - 学习要点：进程状态转换、内存管理、调度策略
- `kernel/swtch.S` - 上下文切换汇编代码
  - 理解：寄存器保存策略、栈切换机制
- `kernel/sysproc.c` - 进程相关系统调用
  - 重点：`sys_fork()`，`sys_exit()`，`sys_wait()`，`sys_kill()`

任务列表

任务1：深入理解进程抽象

学习重点：

1. 分析xv6的进程结构体：

```
struct proc {
  struct spinlock lock;
  enum procstate state;    // 进程状态
  void *chan;              // 等待通道
  int killed;              // 是否被杀死
  int xstate;              // 退出状态
  int pid;                 // 进程ID
  pagetable_t pagetable;   // 用户页表
  struct trapframe *trapframe;  // 陷阱帧
  struct context context;  // 调度上下文
  // ...更多字段
};
```

   - 每个字段的作用是什么？
   - 进程状态转换图是怎样的？
   - 为什么需要锁保护？

2. 理解进程生命周期：

- UNUSED → USED → RUNNABLE → RUNNING → SLEEPING → ZOMBIE
- 每个状态转换的触发条件是什么？
- 哪些操作需要原子性保护？

深入思考：

- 为什么需要ZOMBIE状态？
- 进程表的大小限制有什么影响？
- 如何防止进程ID重复？

任务2：分析xv6的进程创建机制

代码阅读指导：

1. 研读 `allocproc()` 函数：
   - 如何在进程表中找到空闲槽位？
   - 进程ID是如何分配的？
   - 用户栈是如何设置的？
   - 陷阱帧的初始化过程
2. 深入理解 `fork()` 实现：

```
int fork(void) {
  // 1. 分配新进程结构
  // 2. 复制用户内存
  // 3. 复制陷阱帧
  // 4. 设置返回值
  // 5. 标记为RUNNABLE
}
```

   - 为什么父子进程有不同的返回值？
   - 内存复制是如何实现的？
   - 失败时的资源清理策略
3. 分析进程退出机制：
   - `exit()` 与 `wait()` 的协作关系
   - 资源回收的时机和方式
   - 孤儿进程的处理

关键问题：

- fork()的性能瓶颈在哪里？
- 如何实现写时复制优化？

任务3：设计你的进程管理系统

设计要求：

1. 确定进程结构体设计

2. 选择合适的进程表组织方式
3. 设计进程ID分配策略

核心接口设计：

```
// 进程管理基本接口
struct proc* alloc_process(void);         // 分配进程结构
void free_process(struct proc *p);        // 释放进程资源
int create_process(void (*entry)(void)); // 创建新进程
void exit_process(int status);            // 终止当前进程
int wait_process(int *status);            // 等待子进程

// 你需要考虑的设计问题：
// 1. 进程表用数组还是链表？
// 2. 如何高效查找特定PID的进程？
// 3. 是否需要进程组和会话的概念？
// 4. 如何处理进程资源限制？
```

实现策略：

1. 先实现基本的进程创建和销毁
2. 再添加父子关系管理
3. 最后考虑性能优化

任务4：实现上下文切换机制

参考xv6的swtch.S，理解：

1. 上下文切换的本质：
   - 哪些寄存器需要保存？
   - 为什么不保存所有寄存器？
   - 调用者保存 vs 被调用者保存的区别
2. 栈的切换：
   - 内核栈 vs 用户栈的管理
   - 栈指针的保存和恢复
   - 栈溢出的检测和预防

实现挑战：

```
// 上下文结构体设计
struct context {
    uint64 ra;  // 返回地址
    uint64 sp;  // 栈指针
    // 需要保存哪些其他寄存器？
    // 为什么这样选择？
};

// 上下文切换函数
void swtch(struct context *old, struct context *new);
```

关键技术点：

- 上下文切换必须是原子操作
- 中断状态的管理
- 多级栈的处理

任务5：实现调度器

参考xv6的调度策略：

1. 分析 `scheduler()` 函数：
   - 轮转调度的实现方式
   - 如何避免忙等待？
   - 为什么需要开启中断？
2. 理解调度时机：
   - 主动调度 vs 抢占调度
   - `yield()` 函数的作用
   - 时钟中断如何触发调度

调度器设计考虑：

```
void scheduler(void) {
    struct proc *p;
    struct cpu *c = mycpu();

    c->proc = 0;
    for(;;) {
        // 开启中断，允许设备中断
        intr_on();

        // 你的调度算法：
        // 1. 如何选择下一个运行的进程？
        // 2. 如何处理优先级？
        // 3. 如何避免饥饿？
        // 4. 如何平衡公平性和效率？

        for(p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if(p->state == RUNNABLE) {
                // 找到可运行进程，切换过去
                p->state = RUNNING;
                c->proc = p;
                swtch(&c->context, &p->context);
                c->proc = 0;
            }
            release(&p->lock);
        }
    }
```

```
}
```

扩展调度算法：

- 优先级调度
- 多级反馈队列
- 完全公平调度器(CFS)

任务6：实现进程同步原语

基于xv6的sleep/wakeup机制：

1. 理解条件变量的概念：

```
// 等待条件满足
void sleep(void *chan, struct spinlock *lk);
// 唤醒等待特定条件的进程
void wakeup(void *chan);
```

2. 分析典型使用模式：
   a. 生产者-消费者问题
   b. 读者-写者问题
   c. 信号量的实现

实现要点：

- 避免lost wakeup问题
- 锁的正确使用
- 中断状态的管理

测试与调试策略

进程创建测试

```
void test_process_creation(void) {
    printf("Testing process creation...\n");

    // 测试基本的进程创建
    int pid = create_process(simple_task);
    assert(pid > 0);

    // 测试进程表限制
    int pids[NPROC];
    int count = 0;
    for (int i = 0; i < NPROC + 5; i++) {
        int pid = create_process(simple_task);
        if (pid > 0) {
            pids[count++] = pid;
        } else {
            break;
```

```
        }
    }
    printf("Created %d processes\n", count);

    // 清理测试进程
    for (int i = 0; i < count; i++) {
        wait_process(NULL);
    }
}
```

调度器测试

```
void test_scheduler(void) {
    printf("Testing scheduler...\n");

    // 创建多个计算密集型进程
    for (int i = 0; i < 3; i++) {
        create_process(cpu_intensive_task);
    }

    // 观察调度行为
    uint64 start_time = get_time();
    sleep(1000);  // 等待1秒
    uint64 end_time = get_time();

    printf("Scheduler test completed in %lu cycles\n",
           end_time - start_time);
}
```

同步机制测试

```
void test_synchronization(void) {
    // 测试生产者-消费者场景
    shared_buffer_init();

    create_process(producer_task);
    create_process(consumer_task);

    // 等待完成
    wait_process(NULL);
    wait_process(NULL);

    printf("Synchronization test completed\n");
}
```

调试建议

进程状态调试

```
void debug_proc_table(void) {
    printf("=== Process Table ===\n");
    for (int i = 0; i < NPROC; i++) {
        struct proc *p = &proc[i];
        if (p->state != UNUSED) {
            printf("PID:%d State:%d Name:%s\n",
                   p->pid, p->state, p->name);
        }
    }
}
```

调度器调试

- 在调度器中添加统计信息
- 跟踪进程运行时间
- 分析调度延迟

内存泄漏检测

- 跟踪进程创建和销毁
- 检查页表释放
- 监控进程表使用情况

思考题

1. **进程模型**：
   - 为什么选择这种进程结构设计？
   - 如何支持轻量级线程？
2. **调度策略**：
   - 轮转调度的公平性如何？
   - 如何实现实时调度？
3. **性能优化**：
   - fork()的性能瓶颈如何解决？
   - 上下文切换开销如何降低？
4. **资源管理**：
   - 如何实现进程资源限制？
   - 如何处理进程资源泄漏？
5. **扩展性**：
   - 如何支持多核调度？
   - 如何实现负载均衡？
