实验六：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：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：

c

```
// 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 switch(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 2：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：

c

```
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;
                switch(&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 3：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：

c

```
// 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：
    - 生产者-消费者问题
    - 读者-写者问题
    - 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

调度器测试

c

```
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

c

```
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

```
c
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. 调度策略:
    - 轮转调度的公平性如何?
    - 如何实现实时调度?
2. 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?
3. 资源管理:
    - 如何实现进程资源限制?
    - 如何处理进程资源泄漏?
4. 扩展性:
    - 如何支持多核调度?
    - 如何实现负载均衡?