实验 4：首个用户态进程的创建

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和管理进程。

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：

```
c
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：

c

```
// 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

测试与调试策略
进程创建测试

```
c
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);
    }
}
```

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?