从零构建操作系统-学生指导手册

总说明

开发环境要求

- **推荐开发平台**：Ubuntu 22.04 LTS（或 WSL2）
- **基础工具**：git / make / Python3 / qemu-system-riscv64 / riscv64-unknown-elf-gcc
- **学习方法**：理解原理 → 参考xv6 → 独立实现 → 测试调试
- **进度要求**：每完成一个阶段运行测试并提交代码

核心学习资料

- **xv6-riscv源码**：https://github.com/mit-pdos/xv6-riscv
- **RISC-V规范**：https://riscv.org/technical/specifications/
  - 重点：Volume II: Privileged Specification（特权级规范）https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view
  - 在线版本：https://riscv.github.io/riscv-isa-manual/snapshot/privileged/#_preface
- **xv6手册**：https://pdos.csail.mit.edu/6.828/2025/xv6/book-riscv-rev5.pdf

统一提交要求

每个实验需提交以下内容：

```
1. 源代码仓库（通过git管理）
   ├── 完整实现代码（带注释）
   ├── Makefile
   ├── README.md（编译运行说明）
   └── 规范的目录结构

2. 综合实验报告（report.md）
   ├── 系统设计部分
   │   ├── 架构设计说明
   │   ├── 关键数据结构
   │   ├── 与xv6对比分析
   │   └── 设计决策理由
   ├── 实验过程部分
   │   ├── 实现步骤记录
   │   ├── 问题与解决方案
   │   └── 源码理解总结
```

```
    └── 测试验证部分
        ├── 功能测试结果
        ├── 性能数据
        ├── 异常测试
        └── 运行截图/录屏
```

实验0：开发环境搭建

环境搭建步骤

1. 安装基础依赖

```
# Ubuntu/Debian系统
sudo apt-get update
sudo apt-get install -y build-essential git python3 qemu-system-misc expect gdb-multiarch

# 验证QEMU版本（建议5.0+）
qemu-system-riscv64 --version
```

2. 安装RISC-V工具链

```
# 方案A：使用预编译包（推荐）
wget https://github.com/riscv-collab/riscv-gnu-toolchain/releases/download/2023.07.07/riscv64-elf-ubuntu-20.04-gcc-nightly-2023.07.07-nightly.tar.gz
sudo tar -xzf riscv64-elf-ubuntu-20.04-gcc-nightly-2023.07.07-nightly.tar.gz -C /opt/
echo 'export PATH="/opt/riscv/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc

# 方案B：包管理器安装（可能版本较旧）
sudo apt-get install gcc-riscv64-unknown-elf

# 验证安装
riscv64-unknown-elf-gcc --version
```

3. 获取参考资料

```
# 克隆xv6源码作为参考
git clone https://github.com/mit-pdos/xv6-riscv.git
cd xv6-riscv && make qemu  # 验证能否正常运行
```

4. 创建项目结构

```
mkdir riscv-os && cd riscv-os
git init
mkdir -p kernel/{boot,mm,trap,proc,fs,net} include scripts
```

5. 验证环境

创建测试文件验证交叉编译：

```
echo 'int main(){ return 0; }' > test.c
riscv64-unknown-elf-gcc -c test.c -o test.o
file test.o  # 应显示RISC-V 64-bit
```

实验1：RISC-V引导与裸机启动

实验目标

通过参考xv6的启动机制，理解并实现最小操作系统的引导过程，最终在QEMU中输出"Hello OS"。

核心学习资料

xv6关键文件分析

- `kernel/entry.S` - 启动汇编代码
  - 重点理解：栈设置、BSS清零、跳转到C代码
  - 思考：为什么需要设置栈？栈应该多大？
- `kernel/kernel.ld` - 链接脚本
  - 重点理解：入口点设置、段的组织、符号定义
  - 思考：各个段的作用和排列顺序
- `kernel/uart.c` - 串口驱动
  - 重点理解：UART寄存器操作、字符输出实现
  - 思考：如何简化为最小实现？

RISC-V启动机制

- **特权级规范第3.1节**：机器模式启动状态
- **QEMU virt平台**：内存布局和设备地址

```
# 查看QEMU设备树
qemu-system-riscv64 -machine virt,dumpdtb=virt.dtb -nographic
dtc -I dtb -O dts virt.dtb | grep -A5 -B5 "uart\|memory"
```

任务列表

任务1：理解xv6启动流程

学习方法：

1. 阅读 `kernel/entry.S`，回答：
   - 为什么第一条指令是设置栈指针？
   - `la sp, stack0` 中的stack0在哪里定义？
   - 为什么要清零BSS段？
   - 如何从汇编跳转到C函数？
2. 分析 `kernel/kernel.ld`，思考：
   - `ENTRY(_entry)` 的作用是什么？
   - 为什么代码段要放在0x80000000？
   - `etext`、`edata`、`end` 符号有什么用途？

深入思考：

- xv6支持多核，你的单核系统可以如何简化？
- xv6的内存管理很复杂，最小系统需要哪些部分？

任务2：设计最小启动流程

设计要求：

1. 绘制你的启动流程图
2. 确定内存布局方案
3. 列出必需的硬件初始化步骤

关键问题：

- 栈应该放在内存的哪个位置？需要多大？
- 是否需要清零BSS段？为什么？
- 最简串口输出需要配置哪些寄存器？

任务3：实现启动汇编代码

参考xv6实现思路，但要大幅简化：

实现步骤：

1. 创建 `kernel/entry.S`
2. 设置入口点和栈指针
3. 清零BSS段（如果需要）
4. 跳转到C主函数

调试检查点：

```
# 在关键位置插入调试代码
_start:
    li t0, 0x10000000    # UART基地址
    li t1, 'S'           # 启动标记
```

```
    sb t1, 0(t0)          # 输出字符S表示启动

    # 设置栈后再输出一个字符验证
    la sp, stack_top
    li t1, 'P'            # 栈设置完成标记
    sb t1, 0(t0)
```

任务4：编写链接脚本

参考xv6的基本结构，简化复杂部分：

设计考虑：

1. 确定起始地址（通常是0x80000000）
2. 组织代码段、数据段、BSS段
3. 定义必要的符号供C代码使用

验证方法：

```
# 编译后检查内存布局
riscv64-unknown-elf-objdump -h kernel.elf
riscv64-unknown-elf-nm kernel.elf | grep -E "(start|end|text)"
```

任务5：实现串口驱动

参考xv6的uart.c，实现最小功能：

学习要点：

1. UART 16550的基本寄存器：
   - THR (Transmit Holding Register): 0x10000000
   - LSR (Line Status Register): 0x10000005
2. 输出一个字符的完整流程
3. 为什么需要检查LSR的THRE位？

实现策略：

```
// 先实现最基本的字符输出
void uart_putc(char c);

// 成功后实现字符串输出
void uart_puts(char *s);
```

调试建议：

- 先在汇编中直接写UART验证硬件工作
- 再在C函数中实现相同功能
- 最后实现完整的字符串输出

任务6：完成C主函数

设计考虑：

- 函数名可以不是main，与链接脚本保持一致
- 程序结束后应该做什么？死循环还是关机？
- 如何防止程序意外退出导致系统重启？

调试策略

分阶段调试法

1. **硬件验证阶段**：在汇编中直接写UART
2. **启动验证阶段**：验证能跳转到C函数
3. **功能验证阶段**：实现完整的Hello输出

常见问题诊断

问题：QEMU启动后无任何输出

- 检查链接脚本的起始地址
- 验证UART基地址是否正确
- 确认程序是否被正确加载

问题：输出乱码或不完整

- 检查UART初始化是否充分
- 验证字符发送间隔是否太快
- 确认字符串是否正确终止

GDB调试技巧

```
# 启动调试环境
make qemu-gdb  # 在一个终端
gdb-multiarch kernel/kernel.elf  # 在另一个终端
(gdb) target remote :1234
(gdb) b _start
(gdb) c
(gdb) layout asm
(gdb) si  # 单步执行汇编
```

思考题

1. **启动栈的设计**：
   - 你如何确定栈的大小？考虑哪些因素？
   - 如果栈太小会发生什么？如何检测栈溢出？
2. **BSS段清零**：
   - 写一个全局变量，不清零BSS会有什么现象？
   - 哪些情况下可以省略BSS清零？
3. **与xv6的对比**：

- 你的实现比xv6简化了哪些部分？
- 这些简化在什么情况下会成为问题？

4. **错误处理**：
   - 如果UART初始化失败，系统应该如何处理？
   - 如何设计一个最小的错误显示机制？

实验2：内核printf与清屏功能实现

实验目标

通过深入分析xv6的输出系统，理解格式化字符串处理原理，独立实现功能完整的内核printf和清屏功能。

核心学习资料

xv6输出系统架构分析

- `kernel/printf.c` - 格式化输出实现
  - 重点函数：`printf()`，`printint()`，`printptr()`
  - 学习要点：可变参数处理、数字转字符串算法
- `kernel/uart.c` - 硬件抽象层
  - 重点函数：`uartputc()`，`uartinit()`
  - 理解：设备驱动的抽象设计
- `kernel/console.c` - 控制台抽象层
  - 重点函数：`consputc()`，`consolewrite()`
  - 思考：为什么需要这个中间层？

相关技术规范

- **ANSI转义序列**：https://en.wikipedia.org/wiki/ANSI_escape_code
  - 重点：清屏、光标控制、颜色设置
- **C语言可变参数**：stdarg.h的使用方法
  - 关键宏：va_start, va_arg, va_end

任务列表

任务1：深入理解xv6输出架构

分析重点：

1. 研读 `printf.c` 中的核心函数：

- `printf()` 如何解析格式字符串？
- `printint()` 如何处理不同进制转换？
- 负数处理有什么特殊考虑？

2. 理解分层设计：

```
printf() -> consputc() -> uartputc() -> 硬件寄存器
```

- 每一层的职责是什么？
- 这种设计有什么优势？

深入思考：

- xv6为什么不使用递归进行数字转换？
- `printint()` 中处理 `INT_MIN` 的技巧是什么？
- 如何实现线程安全的printf？

任务2：设计你的输出系统架构

设计要求：

1. 画出你的系统架构图
2. 定义各层的接口
3. 说明与xv6设计的异同

关键设计决策：

- 是否需要缓冲区？为什么？
- 如何处理格式错误？
- 是否支持可变宽度格式？

架构建议：

```
// 硬件层
void uart_putc(char c);

// 控制台层
void console_putc(char c);
void console_puts(const char *s);

// 格式化层
int printf(const char *fmt, ...);
int sprintf(char *buf, const char *fmt, ...);
```

任务3：实现数字转换核心算法

学习xv6的printint实现，理解以下问题：

1. 为什么要将负数转为正数处理？
2. 如何避免递归导致的栈溢出？
3. 字符数组的组织方式

实现挑战：

```
// 你需要考虑的边界情况
static void print_number(int num, int base, int sign) {
    // 如何处理 INT_MIN？
    // 如何处理 base=16 的字母输出？
    // 如何实现逆序输出？
}
```

调试策略：

- 先实现十进制正数
- 再处理负数边界情况
- 最后支持十六进制

任务4：实现格式字符串解析

参考xv6的状态机思路：

1. 普通字符直接输出
2. 遇到%进入格式处理状态
3. 解析格式符并调用相应处理函数

实现要点：

```
int printf(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);

    // 你的解析逻辑：
    // 如何区分 %d, %x, %s, %c, %%？
    // 如何提取对应的参数？
    // 如何处理未知格式符？

    va_end(ap);
}
```

测试用例设计：

```
void test_printf_basic() {
    printf("Testing integer: %d\n", 42);
    printf("Testing negative: %d\n", -123);
    printf("Testing zero: %d\n", 0);
    printf("Testing hex: 0x%x\n", 0xABC);
    printf("Testing string: %s\n", "Hello");
    printf("Testing char: %c\n", 'X');
    printf("Testing percent: %%\n");
}

void test_printf_edge_cases() {
    printf("INT_MAX: %d\n", 2147483647);
```

```
    printf("INT_MIN: %d\n", -2147483648);
    printf("NULL string: %s\n", (char*)0);
    printf("Empty string: %s\n", "");
}
```

任务5：实现清屏功能

ANSI转义序列学习：

- `\033[2J` - 清除整个屏幕
- `\033[H` - 光标回到左上角
- `\033[K` - 清除当前行

实现思路：

```
void clear_screen(void) {
    // 方案1：发送ANSI转义序列
    // 方案2：输出足够多的换行符
    // 方案3：直接控制显示硬件（复杂）
}
```

扩展功能：

- 光标定位：`goto_xy(int x, int y)`
- 颜色输出：`printf_color(color, fmt, ...)`
- 清除行：`clear_line()`

任务6：综合测试与优化

功能测试：

1. 基本格式化功能
2. 边界条件处理
3. 性能测试（大量输出）
4. 错误恢复测试

性能优化考虑：

- 字符串输出是否可以批量发送？
- 数字转换是否可以查表优化？
- 格式解析是否可以预编译？

调试建议

分模块调试

1. **底层验证**：先确保单字符输出正常
2. **数字转换**：单独测试各种数字格式
3. **字符串处理**：测试各种字符串边界情况
4. **综合测试**：复杂格式字符串测试

常见问题诊断

输出不完整：

- 检查UART发送是否等待完成
- 验证字符串是否正确终止
- 确认缓冲区大小是否足够

数字输出错误：

- 验证进制转换算法
- 检查负数处理逻辑
- 测试INT_MIN等边界值

格式解析错误：

- 打印解析过程的中间状态
- 验证va_arg的参数类型匹配
- 检查未知格式符的处理

思考题

1. **架构设计：**
   - 为什么需要分层？每层的职责如何划分？
   - 如果要支持多个输出设备（串口+显示器），架构如何调整？
2. **算法选择：**
   - 数字转字符串为什么不用递归？
   - 如何在不使用除法的情况下实现进制转换？
3. **性能优化：**
   - 当前实现的性能瓶颈在哪里？
   - 如何设计一个高效的缓冲机制？
4. **错误处理：**
   - printf遇到NULL指针应该如何处理？
   - 格式字符串错误时的恢复策略是什么？

实验3：页表与内存管理

实验目标

通过深入分析xv6的内存管理系统，理解虚拟内存的工作原理，独立实现物理内存分配器和页表管理系统。

核心学习资料

RISC-V内存管理机制

- **RISC-V特权级规范 第12章： Supervisor-Level ISA**
  - 12.4 Sv39: Page-Based 39-bit Virtual-Memory System
- **在线文档：**https://github.com/riscv/riscv-isa-manual

xv6内存管理源码分析

- `kernel/kalloc.c` - 物理内存分配器
  - 重点函数：`kinit()`，`kalloc()`，`kfree()`
  - 学习要点：空闲页链表管理、简单分配算法
- `kernel/vm.c` - 虚拟内存管理
  - 重点函数：`walk()`，`mappages()`，`uvmcreate()`
  - 学习要点：页表遍历、映射建立、地址转换
- `kernel/riscv.h` - RISC-V相关定义
  - 重点内容：页表项格式、权限位定义、地址操作宏

内存管理理论基础

- **操作系统概念 第9-10章：**内存管理和虚拟内存
  - 在线图书：https://www.os-book.com/OS10/index.html
- **深入理解计算机系统 第9章：**虚拟内存

任务列表

任务1：深入理解Sv39页表机制

学习重点：

1. 分析39位虚拟地址的分解：

```
 38        30 29        21 20        12 11         0
 |  VPN[2]  |   VPN[1]   |   VPN[0]   |   offset   |
```

- 每个VPN段的作用是什么？
- 为什么是9位而不是其他位数？

2. 理解页表项（PTE）格式：
   - V位：有效性标志
   - R/W/X位：读/写/执行权限
   - U位：用户态访问权限
   - 物理页号（PPN）的提取方式

深入思考：

- 为什么选择三级页表而不是二级或四级？

- 中间级页表项的R/W/X位应该如何设置？
- 如何理解"页表也存储在物理内存中"？

任务2：分析xv6的物理内存分配器

代码阅读指导：

1. 研读 `kalloc.c` 的核心数据结构：

```
struct run {
    struct run *next;
};
```

- 这个设计有什么巧妙之处？
- 为什么不需要额外的元数据存储？

2. 分析 `kinit()` 的初始化过程：
    - 如何确定可分配的内存范围？
    - 空闲页链表是如何构建的？
    - 为什么要按页对齐？
3. 理解 `kalloc()` 和 `kfree()` 的实现：
    - 分配算法的时间复杂度是多少？
    - 如何防止double-free？
    - 这种设计的优缺点是什么？

设计思考：

- 如果要实现内存统计功能，应该如何扩展？
- 如何检测内存泄漏？
- 更高效的分配算法有哪些？

任务3：设计你的物理内存管理器

设计要求：

1. 确定内存布局方案
2. 选择合适的数据结构
3. 实现分配和释放接口

关键设计决策：

```
// 你需要决定的接口设计
void pmm_init(void);              // 初始化内存管理器
void* alloc_page(void);           // 分配一个物理页
void free_page(void* page);       // 释放一个物理页
void* alloc_pages(int n);         // 分配连续的n个页面（可选）

// 你需要考虑的问题：
// 1. 如何确定可用内存范围？
// 2. 如何处理内存碎片？
```

```
// 3. 是否需要支持不同大小的分配？
```

实现策略：

1. 首先实现最简单的链表方案
2. 添加基本的错误检查
3. 考虑性能优化（如适用）

任务4：理解xv6的页表管理

代码阅读重点：

1. 分析 `walk()` 函数的递归遍历：
   - 如何从虚拟地址提取各级索引？
   - 遇到无效页表项时如何处理？
   - 为什么需要 `alloc` 参数？
2. 研究 `mappages()` 的映射建立：
   - 如何处理地址对齐？
   - 权限位是如何设置的？
   - 映射失败时的清理工作
3. 理解地址转换宏定义：

```
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
#define PTE_PA(pte)    (((pte) >> 10) << 12)
```

实现挑战：

- 如何避免页表遍历中的无限递归？
- 映射过程中的内存分配失败应该如何恢复？
- 如何确保页表的一致性？

任务5：实现你的页表管理系统

核心接口设计：

```
// 页表类型定义
typedef uint64* pagetable_t;

// 基本操作接口
pagetable_t create_pagetable(void);
int map_page(pagetable_t pt, uint64 va, uint64 pa, int perm);
void destroy_pagetable(pagetable_t pt);

// 辅助函数（内部使用）
pte_t* walk_create(pagetable_t pt, uint64 va);
pte_t* walk_lookup(pagetable_t pt, uint64 va);
```

实现步骤指导：

1. **地址解析实现：**

```
// 从虚拟地址提取页表索引
#define VPN_SHIFT(level) (12 + 9 * (level))
#define VPN_MASK(va, level) (((va) >> VPN_SHIFT(level)) & 0x1FF)
```

2. **页表遍历实现：**
   - 从根页表开始逐级查找
   - 检查每一级页表项的有效性
   - 必要时创建中间级页表
3. **映射建立实现：**
   - 确保地址按页对齐
   - 正确设置权限位
   - 处理映射冲突

调试检查点：

```
// 实现页表打印功能用于调试
void dump_pagetable(pagetable_t pt, int level) {
    // 递归打印页表内容
    // 显示虚拟地址到物理地址的映射关系
    // 标明权限位设置
}
```

任务6：启用虚拟内存

参考xv6的内核初始化：

1. 研读 `kvminit()` 的内核页表创建：
   - 哪些内存区域需要映射？
   - 为什么采用恒等映射？
   - 设备内存的权限设置
2. 分析 `kvminithart()` 的页表激活：
   - satp寄存器的格式和设置
   - `sfence.vma` 指令的作用
   - 激活前后的注意事项

实现策略：

```
void kvminit(void) {
    // 1. 创建内核页表
    kernel_pagetable = create_pagetable();

    // 2. 映射内核代码段（R+X权限）
    map_region(kernel_pagetable, KERNBASE, KERNBASE,
               (uint64)etext - KERNBASE, PTE_R | PTE_X);
```

```
    // 3. 映射内核数据段（R+W权限）
    map_region(kernel_pagetable, (uint64)etext, (uint64)etext,
               PHYSTOP - (uint64)etext, PTE_R | PTE_W);

    // 4. 映射设备（UART等）
    map_region(kernel_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
}

void kvminithart(void) {
    // 激活内核页表
    w_satp(MAKE_SATP(kernel_pagetable));
    sfence_vma();
}
```

关键技术细节：

- SATP寄存器格式：`MODE[63:60] | ASID[59:44] | PPN[43:0]`
- MODE=8表示Sv39模式
- `sfence.vma` 用于刷新TLB

测试与调试策略

分层测试方法

1. **物理内存分配器测试：**

```
void test_physical_memory(void) {
    // 测试基本分配和释放
    void *page1 = alloc_page();
    void *page2 = alloc_page();
    assert(page1 != page2);
    assert(((uint64)page1 & 0xFFF) == 0);  // 页对齐检查

    // 测试数据写入
    *(int*)page1 = 0x12345678;
    assert(*(int*)page1 == 0x12345678);

    // 测试释放和重新分配
    free_page(page1);
    void *page3 = alloc_page();
    // page3可能等于page1（取决于分配策略）

    free_page(page2);
    free_page(page3);
}
```

2. **页表功能测试：**

```
void test_pagetable(void) {
```

```
    pagetable_t pt = create_pagetable();

    // 测试基本映射
    uint64 va = 0x1000000;
    uint64 pa = (uint64)alloc_page();
    assert(map_page(pt, va, pa, PTE_R | PTE_W) == 0);

    // 测试地址转换
    pte_t *pte = walk_lookup(pt, va);
    assert(pte != 0 && (*pte & PTE_V));
    assert(PTE_PA(*pte) == pa);

    // 测试权限位
    assert(*pte & PTE_R);
    assert(*pte & PTE_W);
    assert(!(*pte & PTE_X));
}
```

3. **虚拟内存激活测试：**

```
void test_virtual_memory(void) {
    printf("Before enabling paging...\n");

    // 启用分页
    kvminit();
    kvminithart();

    printf("After enabling paging...\n");

    // 测试内核代码仍然可执行
    // 测试内核数据仍然可访问
    // 测试设备访问仍然正常
}
```

常见问题诊断

问题：启用分页后系统崩溃

- 检查点1：内核代码是否正确映射？
- 检查点2：栈空间是否映射？
- 检查点3：设备地址是否映射？
- 调试方法：在启用前后打印关键地址的映射状态

问题：页表映射失败

- 检查地址对齐：虚拟地址和物理地址都必须页对齐
- 检查内存不足：中间页表创建可能失败
- 检查权限冲突：重复映射可能导致权限不一致

问题：地址转换错误

- 验证VPN提取算法是否正确
- 检查PTE格式是否符合RISC-V规范
- 确认物理地址计算是否正确

GDB调试技巧

```
# 查看页表内容
(gdb) x/64gx $satp_register_content
# 查看特定虚拟地址的映射
(gdb) monitor info mem
# 检查页表遍历过程
(gdb) b walk_create
(gdb) watch $a0  # 监视页表指针变化
```

性能优化考虑

内存分配优化

1. **批量分配**：一次性分配多个连续页面
2. **分级分配**：针对不同大小需求使用不同分配器
3. **缓存优化**：保持少量预分配页面池

页表优化

1. **TLB友好**：合理安排虚拟地址布局
2. **大页支持**：对于大块内存使用大页映射
3. **延迟映射**：按需创建页表项

思考题

1. **设计对比**：
   - 你的物理内存分配器与xv6有什么不同？
   - 为什么选择这种设计？有什么权衡？
2. **内存安全**：
   - 如何防止内存分配器被恶意利用？
   - 页表权限设置的安全考虑有哪些？
3. **性能分析**：
   - 当前实现的性能瓶颈在哪里？
   - 如何测量和优化内存访问性能？
4. **扩展性**：
   - 如果要支持用户进程，需要什么修改？
   - 如何实现内存共享和写时复制？
5. **错误恢复**：

- 页表创建失败时如何清理已分配的资源？
- 如何检测和处理内存泄漏？

实验4：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：

```
// 时钟中断委托给S模式
w_mideleg(r_mideleg() | (1L << 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：

```
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：

```
// 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
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：

```
# 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：

```
// 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：

```
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：

```
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

```
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

```
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

```
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？

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？

实验6：系统调用

实验目标

通过分析xv6的系统调用机制，深入理解用户态与内核态的交互方式，实现完整的系统调用框架和常用系统调用功能。

核心学习资料

系统调用理论基础

- **操作系统概念 第2章**：操作系统结构
- **RISC-V特权级规范 第12.1节**：Supervisor Trap Handling
- **xv6手册 第2.5节和第4章**：系统调用和陷阱

xv6系统调用源码分析

- `kernel/syscall.c` - 系统调用分发机制
  - 重点函数：`syscall()`，`argint()`，`argstr()`，`argaddr()`
  - 学习要点：参数传递、返回值处理、错误检查
- `kernel/sysproc.c` - 进程相关系统调用实现
- `kernel/sysfile.c` - 文件相关系统调用实现
- `user/usys.pl` - 用户态系统调用桩代码生成
- `kernel/trampoline.S` - 用户态/内核态切换

RISC-V系统调用约定

- **调用约定**：ecall指令、寄存器使用、参数传递
- **特权级切换**：用户模式到监督模式的转换过程

任务列表

任务1：理解系统调用的实现原理

学习重点：

1. 分析系统调用的完整流程：

```
用户程序调用 → usys.S桩代码 → ecall指令 →
uservec → usertrap → syscall → 系统调用实现 →
返回用户态
```

- 每个环节的作用是什么？
- 参数是如何传递的？

- 返回值如何返回？

2. 研究RISC-V的ecall机制：
   - ecall指令的作用
   - scause寄存器中系统调用的编码
   - sepc寄存器的作用和更新
3. 理解特权级切换：
   - 用户栈到内核栈的转换
   - 寄存器状态的保存和恢复
   - 页表的切换时机

深入思考：

- 为什么需要陷阱帧(trapframe)？
- 系统调用和中断处理有什么相同和不同？

任务2：分析xv6的系统调用分发机制

代码阅读指导：

1. 研读 `syscall.c` 中的核心分发逻辑：

```
void syscall(void) {
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;  // 系统调用号
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();  // 调用并保存返回值
  } else {
    // 处理无效系统调用
  }
}
```

- 系统调用号是如何传递的？
- 返回值存储在哪里？
- 错误处理机制是什么？

2. 分析参数提取函数：

```
int argint(int n, int *ip);      // 获取整数参数
int argaddr(int n, uint64 *ip);  // 获取地址参数
int argstr(int n, char *buf, int max);  // 获取字符串参数
```

- 参数是从哪里提取的？
- 如何处理不同类型的参数？
- 边界检查是如何实现的？

3. 理解用户内存访问：

- `copyout()` 和 `copyin()` 的作用
- 为什么不能直接访问用户内存？
- 如何防止用户传递恶意指针？

任务3：设计你的系统调用框架

设计要求：

1. 定义系统调用表结构
2. 设计参数传递机制
3. 实现错误处理策略

核心组件设计：

```
// 系统调用描述符
struct syscall_desc {
    int (*func)(void);          // 实现函数
    char *name;                 // 系统调用名称
    int arg_count;              // 参数个数
    // 可选：参数类型描述
};

// 系统调用表
extern struct syscall_desc syscall_table[];

// 系统调用分发器
void syscall_dispatch(void);

// 参数提取辅助函数
int get_syscall_arg(int n, long *arg);
int get_user_string(const char __user *str, char *buf, int max);
int get_user_buffer(const void __user *ptr, void *buf, int size);

// 你需要考虑的问题：
// 1. 如何验证用户提供的指针？
// 2. 如何处理系统调用失败？
// 3. 如何支持可变参数的系统调用？
// 4. 如何实现系统调用的权限检查？
```

任务4：实现基础系统调用

必需实现的系统调用：

1. **进程控制类：**

```
   int sys_fork(void);        // 创建子进程
   int sys_exit(void);        // 终止进程
   int sys_wait(void);        // 等待子进程
   int sys_kill(void);        // 发送信号
   int sys_getpid(void);      // 获取进程ID
```

2. 文件操作类：

```
    int sys_open(void);        // 打开文件
    int sys_close(void);       // 关闭文件
    int sys_read(void);        // 读文件
    int sys_write(void);       // 写文件
```

3. 内存管理类：

```
    void* sys_sbrk(void);      // 调整堆大小
    // 可选：mmap, munmap等
```

实现策略：

```
// 以sys_write为例
int sys_write(void) {
    int fd;
    char *buf;
    int count;

    // 1. 提取参数
    if (argint(0, &fd) < 0 ||
        argaddr(1, (uint64*)&buf) < 0 ||
        argint(2, &count) < 0) {
        return -1;
    }

    // 2. 参数有效性检查
    if (fd < 0 || fd >= NOFILE || count < 0) {
        return -1;
    }

    // 3. 调用内核函数实现
    return filewrite(myproc()->ofile[fd], buf, count);
}
```

任务5：实现用户态系统调用接口

参考xv6的usys.pl，理解：

1. 桩代码生成机制：

```
    # 每个系统调用的桩代码格式
    .global write
    write:
     li a7, SYS_write     # 系统调用号加载到a7
     ecall                # 陷入内核
     ret                  # 返回
```

2. 用户库函数设计：

```
    // 用户库中的系统调用声明
```

```
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
// ...
```

实现考虑：

- 如何处理系统调用的错误返回？
- 是否需要errno机制？
- 如何提供用户友好的接口？

任务6：系统调用安全性

安全检查要点：

1. **指针验证：**

```
// 检查用户指针是否有效
int check_user_ptr(const void *ptr, int size) {
    // 1. 指针是否在用户地址空间？
    // 2. 内存区域是否有相应权限？
    // 3. 是否会越界访问？
}
```

2. **缓冲区保护：**
   - 防止缓冲区溢出
   - 检查字符串是否正确终止
   - 限制数据传输大小
3. **权限检查：**
   - 文件访问权限
   - 进程操作权限
   - 资源使用限制
4. **竞态条件防护：**
   - TOCTTOU攻击防护
   - 原子操作保证
   - 锁的正确使用

测试与调试策略

基础功能测试

```
void test_basic_syscalls(void) {
    printf("Testing basic system calls...\n");
```

```
    // 测试getpid
    int pid = getpid();
    printf("Current PID: %d\n", pid);

    // 测试fork
    int child_pid = fork();
    if (child_pid == 0) {
        // 子进程
        printf("Child process: PID=%d\n", getpid());
        exit(42);
    }
    else if (child_pid > 0) {
        // 父进程
        int status;
        wait(&status);
        printf("Child exited with status: %d\n", status);
    }
    else {
        printf("Fork failed!\n");
    }
}
```

参数传递测试

```
void test_parameter_passing(void) {
    // 测试不同类型参数的传递
    char buffer[] = "Hello, World!";
    int fd = open("/dev/console", O_RDWR);

    if (fd >= 0) {
        int bytes_written = write(fd, buffer, strlen(buffer));
        printf("Wrote %d bytes\n", bytes_written);
        close(fd);
    }

    // 测试边界情况
    write(-1, buffer, 10);      // 无效文件描述符
    write(fd, NULL, 10);        // 空指针
    write(fd, buffer, -1);      // 负数长度
}
```

安全性测试

```
void test_security(void) {
    // 测试无效指针访问
    char *invalid_ptr = (char*)0x1000000;  // 可能无效的地址
    int result = write(1, invalid_ptr, 10);
```

```
    printf("Invalid pointer write result: %d\n", result);

    // 测试缓冲区边界
    char small_buf[4];
    result = read(0, small_buf, 1000);  // 尝试读取超过缓冲区大小

    // 测试权限检查
    // ...
}
```

性能测试

```
void test_syscall_performance(void) {
    uint64 start_time = get_time();

    // 大量系统调用测试
    for (int i = 0; i < 10000; i++) {
        getpid();  // 简单的系统调用
    }

    uint64 end_time = get_time();
    printf("10000 getpid() calls took %lu cycles\n",
           end_time - start_time);
}
```

调试建议

系统调用跟踪

```
// 在syscall.c中添加调试信息
void syscall(void) {
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7;

    // 调试输出
    if (debug_syscalls) {
        printf("PID %d: syscall %d (%s)\n",
               p->pid, num, syscall_names[num]);
    }

    // 原有逻辑...
}
```

参数检查调试

- 在参数提取函数中添加验证日志

- 跟踪用户内存访问
- 记录异常的参数值

性能分析

- 测量系统调用延迟
- 分析频繁调用的系统调用
- 识别性能瓶颈

思考题

1. **设计权衡**：
   - 系统调用的数量应该如何确定？
   - 如何平衡功能性和安全性？
2. **性能优化**：
   - 系统调用的主要开销在哪里？
   - 如何减少用户态/内核态切换开销？
3. **安全考虑**：
   - 如何防止系统调用被滥用？
   - 如何设计安全的参数传递机制？
4. **扩展性**：
   - 如何添加新的系统调用？
   - 如何保持向后兼容性？
5. **错误处理**：
   - 系统调用失败时应该如何处理？
   - 如何向用户程序报告详细的错误信息？

实验7：文件系统

实验目标

通过深入分析xv6的简化文件系统，理解现代文件系统的核心概念和实现原理，独立实现一个功能完整的日志文件系统。

核心学习资料

文件系统理论基础

- **操作系统概念 第13-14章**：文件系统接口和实现
- **xv6手册 第10章**：文件系统

xv6文件系统源码分析

- `kernel/fs.h` - 文件系统结构定义
  - 重点：超级块、inode、目录项的格式
- `kernel/fs.c` - 文件系统核心实现
  - 重点函数：`ialloc()`, `iget()`, `iput()`, `namei()`
- `kernel/file.c` - 文件描述符管理
  - 重点：打开文件表、文件描述符分配
- `kernel/log.c` - 日志系统实现
  - 重点：事务处理、崩溃恢复、写前日志
- `kernel/bio.c` - 块缓存管理
  - 重点：缓存策略、磁盘I/O调度

磁盘和存储

- **理解磁盘结构**：扇区、柱面、磁头
- **QEMU磁盘模拟**：virtio-blk设备的使用

任务列表

任务1：理解xv6文件系统布局

学习重点：

1. 分析磁盘布局结构：

```
| boot | super | log | inode blocks | bitmap | data blocks |
|  0   |   1   | 2-? |    ?-?       |   ?    |    ?-end    |
   - 每个区域的作用是什么？
   - 为什么要这样组织？
   - 各区域的大小如何确定？
```

2. 理解超级块（superblock）的作用：

```
   struct superblock {
      uint magic;        // 文件系统魔数
      uint size;         // 文件系统大小（块数）
      uint nblocks;      // 数据块数量
      uint ninodes;      // inode数量
      uint nlog;         // 日志块数量
      uint logstart;     // 日志起始块号
      uint inodestart;   // inode区起始块号
      uint bmapstart;    // 位图起始块号
    };
```

- 为什么需要这些元数据？

- 如何确保超级块的一致性？

3. 深入理解inode结构：

```
    struct dinode {
      short type;           // 文件类型
      short major;          // 主设备号
      short minor;          // 次设备号
      short nlink;          // 硬链接计数
      uint size;            // 文件大小
      uint addrs[NDIRECT+1]; // 数据块地址
    };
```

- 直接块和间接块的设计思路
- 如何支持大文件？
- 硬链接机制的实现

深入思考：

- 为什么选择这种简单的布局？
- 如何提高空间利用率？
- 现代文件系统有什么改进？

任务2：分析xv6的inode管理机制

代码阅读指导：

1. 研读inode缓存管理：

```
    struct inode {
      uint dev;           // 设备号
      uint inum;          // inode号
      int ref;            // 引用计数
      struct sleeplock lock; // 保护inode内容
      int valid;          // inode已从磁盘读取？
  // 从磁盘拷贝的内容
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
 };
```

- 内存inode和磁盘inode的关系
- 引用计数的作用和管理
- 缓存一致性如何保证

2. 分析inode分配算法：

```
    struct inode* ialloc(uint dev, short type) {
```

```
    // 1. 在inode位图中找空闲inode
    // 2. 初始化inode内容
    // 3. 写入磁盘
    // 4. 返回内存中的inode
  }
```

- 如何快速找到空闲inode？
- 分配失败时的处理策略
- 并发分配的同步机制

3. 理解文件数据块管理：

```
  static uint bmap(struct inode *ip, uint bn) {
    // bn是文件内的逻辑块号
    // 返回对应的物理块号
    // 处理直接块和间接块的映射
  }
```

- 逻辑块号到物理块号的转换
- 间接块的实现机制
- 如何扩展文件大小

关键问题：

- inode缓存的替换策略是什么？
- 如何防止inode泄漏？
- 大文件的性能问题如何解决？

任务3：设计你的文件系统布局

设计要求：

1. 确定磁盘分区方案
2. 设计inode和数据块组织
3. 选择合适的块大小

设计考虑：

```
// 你的文件系统布局设计
#define BLOCK_SIZE      4096    // 块大小选择的考虑
#define SUPERBLOCK_NUM  1       // 超级块位置
#define LOG_START       2       // 日志区起始
#define LOG_SIZE        30      // 日志区大小
// inode设计
struct my_inode {
    uint16_t mode;              // 文件模式和类型
    uint16_t uid;               // 所有者ID
    uint32_t size;              // 文件大小
    uint32_t blocks;            // 分配的块数
    uint32_t atime, mtime, ctime; // 时间戳
```

```
    uint32_t direct[12];        // 直接块指针
    uint32_t indirect;          // 一级间接块
    uint32_t double_indirect;   // 二级间接块（可选）
};
// 你需要考虑的问题：
// 1. 如何平衡小文件和大文件的效率？
// 2. 是否需要扩展属性支持？
// 3. 如何优化目录性能？
// 4. 是否支持符号链接？
```

任务4：实现块缓存系统

参考xv6的bio.c，理解：

1. 缓存结构设计：

```
struct buf {
  int valid;           // 缓存是否有效
  int disk;            // 是否需要写回磁盘
  uint dev;            // 设备号
  uint blockno;        // 块号
  struct sleeplock lock; // 保护缓存内容
  uint refcnt;         // 引用计数
  struct buf *prev, *next; // LRU链表
  uchar data[BSIZE];  // 实际数据
};
```

2. 缓存管理策略：

```
struct buf* bread(uint dev, uint blockno);  // 读取块
void bwrite(struct buf *b);                 // 写入块
void brelse(struct buf *b);                 // 释放块
```

实现挑战：

```
// 你的块缓存实现
struct buffer_head {
    uint32_t block_num;         // 块号
    char *data;                 // 数据指针
    int dirty;                  // 脏位
    int ref_count;              // 引用计数
    struct buffer_head *next;   // 哈希链表
    struct buffer_head *lru_next, *lru_prev; // LRU链表
};
// 关键函数设计
struct buffer_head* get_block(uint dev, uint block);
void put_block(struct buffer_head *bh);
void sync_block(struct buffer_head *bh);
void flush_all_blocks(uint dev);
// 考虑的问题：
// 1. 缓存大小如何确定？
```

```
// 2. 什么时候触发写回？
// 3. 如何处理I/O错误？
// 4. 预读策略是否需要？
```

任务5：实现日志系统

参考xv6的log.c，深入理解：

1. 日志的作用和原理：
   - 写前日志(Write-Ahead Logging)
   - 事务的原子性保证
   - 崩溃恢复机制
2. 日志结构设计：

```
    struct logheader {
      int n;               // 日志中的块数
      int block[LOGSIZE]; // 每个块在文件系统中的位置
    };
```

3. 事务处理流程：

```
    void begin_op(void);    // 开始事务
    void log_write(struct buf *b); // 记录写操作
    void end_op(void);      // 提交事务
```

实现要点：

```
// 日志系统状态
struct log_state {
    struct spinlock lock;
    int start;              // 日志区起始块号
    int size;               // 日志区大小
    int outstanding;        // 未完成的系统调用数
    int committing;         // 是否正在提交
    int dev;                // 设备号
};
// 关键实现函数
void log_init(int dev, struct superblock *sb);
void begin_transaction(void);
void end_transaction(void);
void log_block_write(struct buffer_head *bh);
void recover_log(void);
// 设计考虑：
// 1. 日志大小如何确定？
// 2. 如何处理日志满的情况？
// 3. 恢复过程如何确保幂等性？
// 4. 如何优化日志性能？
```

任务6：实现目录和路径解析

理解xv6的目录机制：

1. 目录项格式：

```
    struct dirent {
      ushort inum;        // inode号，0表示空闲
      char name[DIRSIZ];  // 文件名
    };
```

2. 路径解析算法：

```
static struct inode* namex(char *path, int nameiparent, char *name) {
      // 解析路径，返回对应的inode
      // nameiparent=1时返回父目录inode
}
```

实现挑战：

```
 // 目录操作接口
 struct inode* dir_lookup(struct inode *dp, char *name, uint *poff);
 int dir_link(struct inode *dp, char *name, uint inum);
 int dir_unlink(struct inode *dp, char *name);
// 路径解析
 struct inode* path_walk(char path); struct inode path_parent(char *pa
th, char *name);
// 需要考虑的问题：
 // 1. 目录的最大大小限制
 // 2. 长文件名的支持
 // 3. 目录遍历的效率
 // 4. 硬链接和符号链接的处理
```

测试与调试策略

文件系统完整性测试

```
 void test_filesystem_integrity(void) {
     printf("Testing filesystem integrity…\n");
    // 创建测试文件
    int fd = open("testfile", O_CREATE | O_RDWR);
    assert(fd >= 0);

    // 写入数据
    char buffer[] = "Hello, filesystem!";
    int bytes = write(fd, buffer, strlen(buffer));
    assert(bytes == strlen(buffer));
    close(fd);

    // 重新打开并验证
    fd = open("testfile", O_RDONLY);
    assert(fd >= 0);
```

```
    char read_buffer[64];
    bytes = read(fd, read_buffer, sizeof(read_buffer));
    read_buffer[bytes] = '\0';

    assert(strcmp(buffer, read_buffer) == 0);
    close(fd);

    // 删除文件
    assert(unlink("testfile") == 0);

    printf("Filesystem integrity test passed\n");
}
```

并发访问测试

```
void test_concurrent_access(void) {
  printf("Testing concurrent file access…
");
  // 创建多个进程同时访问文件系统
  for (int i = 0; i < 4; i++) {
    if (fork() == 0) {
        // 子进程：创建和删除文件
        char filename[32];
        snprintf(filename, sizeof(filename), "test_%d", i);

        for (int j = 0; j < 100; j++) {
            int fd = open(filename, O_CREATE | O_RDWR);
            if (fd >= 0) {
                write(fd, &j, sizeof(j));
                close(fd);
                unlink(filename);
            }
        }
        exit(0);
     }
  }

  // 等待所有子进程完成
  for (int i = 0; i < 4; i++) {
    wait(NULL);
  }

  printf("Concurrent access test completed
```

崩溃恢复测试

```
void test_crash_recovery(void) {
```

```
    printf("Testing crash recovery…
");
  // 模拟崩溃场景：
  // 1. 开始大量文件操作
  // 2. 在中途"崩溃"（重启系统）
  // 3. 检查文件系统一致性

  // 注意：这个测试需要特殊的测试框架
  // 可以通过修改内核代码来模拟崩溃
```

性能测试

```
void test_filesystem_performance(void) {
    printf("Testing filesystem performance…\n");
    uint64 start_time = get_time();

    // 大量小文件测试
    for (int i = 0; i < 1000; i++) {
        char filename[32];
        snprintf(filename, sizeof(filename), "small_%d", i);

        int fd = open(filename, O_CREATE | O_RDWR);
        write(fd, "test", 4);
        close(fd);
    }

    uint64 small_files_time = get_time() - start_time;

    // 大文件测试
    start_time = get_time();
    int fd = open("large_file", O_CREATE | O_RDWR);
    char large_buffer[4096];
    for (int i = 0; i < 1024; i++) {  // 4MB文件
        write(fd, large_buffer, sizeof(large_buffer));
    }
    close(fd);

    uint64 large_file_time = get_time() - start_time;

    printf("Small files (1000x4B): %lu cycles\n", small_files_time);
    printf("Large file (1x4MB): %lu cycles\n", large_file_time);

    // 清理测试文件
    for (int i = 0; i < 1000; i++) {
        char filename[32];
        snprintf(filename, sizeof(filename), "small_%d", i);
        unlink(filename);
    }
```

```
    unlink("large_file");
}
```

调试建议

文件系统状态检查

```
void debug_filesystem_state(void) {
    printf("=== Filesystem Debug Info ===
");
    // 显示超级块信息
    struct superblock sb;
    read_superblock(&sb);
    printf("Total blocks: %d
", sb.size);
    printf("Free blocks: %d
", count_free_blocks());
    printf("Free inodes: %d
", count_free_inodes());

    // 显示块缓存状态
    printf("Buffer cache hits: %d
", buffer_cache_hits);
    printf("Buffer cache misses: %d
```

inode追踪

```
void debug_inode_usage(void) {
     printf("=== Inode Usage ===\n");
     for (int i = 0; i < NINODE; i++) {
         struct inode *ip = &icache.inode[i];
         if (ip->ref > 0) {
             printf("Inode %d: ref=%d, type=%d, size=%d\n",
                    ip->inum, ip->ref, ip->type, ip->size);
         }
     }
 }
```

磁盘I/O统计

```
 void debug_disk_io(void) {
     printf("=== Disk I/O Statistics ===
");
     printf("Disk reads: %d
", disk_read_count);
     printf("Disk writes: %d
", disk_write_count);
```

思考题

1. **设计权衡：**
   - xv6的简单文件系统有什么优缺点？
   - 如何在简单性和性能之间平衡？
2. **一致性保证：**
   - 日志系统如何确保原子性？
   - 如果在恢复过程中再次崩溃会怎样？
3. **性能优化：**
   - 文件系统的主要性能瓶颈在哪里？
   - 如何改进目录查找的效率？
4. **可扩展性：**
   - 如何支持更大的文件和文件系统？
   - 现代文件系统有哪些先进特性？
5. **可靠性：**
   - 如何检测和修复文件系统损坏？
   - 如何实现文件系统的在线检查？

实验8：系统扩展项目

实验概述

经过前面的七个实验，你已经构建了一个基本的操作系统，接下来需要进一步完善这个系统。你可以从用户角度思考，选择一个扩展方向进行独立设计和实现，使得系统变得更好用或更有效率。

扩展项目列表

以下是各个可选项目的任务概述：

项目1：优先级调度系统

分析当前简单调度器（如轮转调度）的性能瓶颈，设计并实现支持进程优先级的调度算法，平衡实时任务的响应性需求与普通任务的公平性保证，提升系统整体调度效率。

项目2：ELF加载器与用户空间

实现标准ELF可执行文件格式的解析和加载机制，建立完整的用户态程序执行环境，包括虚拟内存映射、程序段加载、动态链接支持等，使系统能够运行标准编译的用户程序。

项目3：进程间通信系统

设计并实现多种IPC机制（如管道、消息队列、共享内存、信号量等），为用户进程提供高效的数据交换和同步原语，支持复杂的多进程协作应用场景。

项目4：内核日志系统

构建结构化的内核日志框架，支持不同级别的日志记录、缓冲管理、格式化输出，为系统调试、性能分析和故障诊断提供完善的信息收集和查看机制。

项目5：Copy-on-Write Fork

优化进程创建机制，实现写时复制技术，使fork操作只复制页表而延迟实际内存复制，显著减少内存使用和进程创建开销，提升系统性能和资源利用率。