

- 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：

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

2. 理解超级块（superblock）的作用：

```
1 struct superblock {
2     uint magic;           // 文件系统魔数
3     uint size;            // 文件系统大小（块数）
4     uint nblocks;         // 数据块数量
5     uint ninodes;         // inode数量
6     uint nlog;            // 日志块数量
7     uint logstart;        // 日志起始块号
8     uint inodestart;      // inode区起始块号
9     uint bmapstart;       // 位图起始块号
10 };
```

- 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：

```

1 struct dinode {
2     short type;           // 文件类型
3     short major;         // 主设备号
4     short minor;         // 次设备号
5     short nlink;         // 硬链接计数
6     uint size;           // 文件大小
7     uint addrs[NDIRECT+1]; // 数据块地址
8 };

```

- 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：

```

1 struct inode {
2     uint dev;           // 设备号
3     uint inum;          // inode号
4     int ref;            // 引用计数
5     struct sleeplock lock; // 保护inode内容
6     int valid;          // inode已从磁盘读取？
7     // 从磁盘拷贝的内容
8     short type;
9     short major;
10    short minor;
11    short nlink;
12    uint size;
13    uint addrs[NDIRECT+1];
14 };

```

- 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：

```

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

```

```

2    // 1. 在inode位图中找空闲inode
3    // 2. 初始化inode内容
4    // 3. 写入磁盘
5    // 4. 返回内存中的inode
6    }

```

- 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:

```

1    static uint bmap(struct inode *ip, uint bn) {
2        // bn是文件内的逻辑块号
3        // 返回对应的物理块号
4        // 处理直接块和间接块的映射
5    }

```

- 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:

```

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

```

```

13     uint32_t direct[12];           // 直接块指针
14     uint32_t indirect;            // 一级间接块
15     uint32_t double_indirect;     // 二级间接块（可选）
16 };
17 // 你需要考虑的问题：
18 // 1. 如何平衡小文件和大文件的效率？
19 // 2. 是否需要扩展属性支持？
20 // 3. 如何优化目录性能？
21 // 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：

```

1     struct buf {
2         int valid;                // 缓存是否有效
3         int disk;                 // 是否需要写回磁盘
4         uint dev;                 // 设备号
5         uint blockno;             // 块号
6         struct sleeplock lock;    // 保护缓存内容
7         uint refcnt;              // 引用计数
8         struct buf *prev, *next; // LRU链表
9         uchar data[BSIZE];       // 实际数据
10    };

```

2. 缓存管理策略：

```

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

```

实现挑战：

```

1     // 你的块缓存实现
2     struct buffer_head {
3         uint32_t block_num;        // 块号
4         char *data;                // 数据指针
5         int dirty;                 // 脏位
6         int ref_count;             // 引用计数
7         struct buffer_head *next;  // 哈希链表
8         struct buffer_head *lru_next, *lru_prev; // LRU链表
9     };
10    // 关键函数设计
11    struct buffer_head* get_block(uint dev, uint block);
12    void put_block(struct buffer_head *bh);
13    void sync_block(struct buffer_head *bh);
14    void flush_all_blocks(uint dev);
15    // 考虑的问题：
16    // 1. 缓存大小如何确定？

```

```
17 // 2. 什么时候触发写回？
18 // 3. 如何处理I/O错误？
19 // 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：

```
1 struct logheader {
2     int n;                // 日志中的块数
3     int block[LOGSIZE]; // 每个块在文件系统中的位置
4 };
```

3. 事务处理流程：

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

实现要点：

```
1 // 日志系统状态
2 struct log_state {
3     struct spinlock lock;
4     int start;          // 日志区起始块号
5     int size;           // 日志区大小
6     int outstanding;    // 未完成的系统调用数
7     int committing;    // 是否正在提交
8     int dev;           // 设备号
9 };
10 // 关键实现函数
11 void log_init(int dev, struct superblock *sb);
12 void begin_transaction(void);
13 void end_transaction(void);
14 void log_block_write(struct buffer_head *bh);
15 void recover_log(void);
16 // 设计考虑：
17 // 1. 日志大小如何确定？
18 // 2. 如何处理日志满的情况？
19 // 3. 恢复过程如何确保幂等性？
20 // 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:

```
1 struct dirent {
2     ushort inum;        // inode号, 0表示空闲
3     char name[DIRSIZ];  // 文件名
4 };
```

2. 路径解析算法:

```
1 static struct inode* namex(char *path, int nameiparent, char *name) {
2     // 解析路径, 返回对应的inode
3     // nameiparent=1时返回父目录inode
4 }
```

实现挑战:

```
1 // 目录操作接口
2 struct inode* dir_lookup(struct inode *dp, char *name, uint *poff);
3 int dir_link(struct inode *dp, char *name, uint inum);
4 int dir_unlink(struct inode *dp, char *name);
5 // 路径解析
6 struct inode* path_walk(char path); struct inode path_parent(char *path, char *name);
7 // 需要考虑的问题:
8 // 1. 目录的最大大小限制
9 // 2. 长文件名的支持
10 // 3. 目录遍历的效率
11 // 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

```
1 void test_filesystem_integrity(void) {
2     printf("Testing filesystem integrity...\n");
3     // 创建测试文件
4     int fd = open("testfile", O_CREATE | O_RDWR);
5     assert(fd >= 0);
6
7     // 写入数据
8     char buffer[] = "Hello, filesystem!";
9     int bytes = write(fd, buffer, strlen(buffer));
10    assert(bytes == strlen(buffer));
11    close(fd);
12
13    // 重新打开并验证
14    fd = open("testfile", O_RDONLY);
15    assert(fd >= 0);
```

```

16
17     char read_buffer[64];
18     bytes = read(fd, read_buffer, sizeof(read_buffer));
19     read_buffer[bytes] = '\0';
20
21     assert(strcmp(buffer, read_buffer) == 0);
22     close(fd);
23
24     // 删除文件
25     assert(unlink("testfile") == 0);
26
27     printf("Filesystem integrity test passed\n");
28 }

```

并发访问测试

```

1 void test_concurrent_access(void) {
2     printf("Testing concurrent file access...
3 ");
4     // 创建多个进程同时访问文件系统
5     for (int i = 0; i < 4; i++) {
6         if (fork() == 0) {
7             // 子进程：创建和删除文件
8             char filename[32];
9             snprintf(filename, sizeof(filename), "test_%d", i);
10
11             for (int j = 0; j < 100; j++) {
12                 int fd = open(filename, O_CREATE | O_RDWR);
13                 if (fd >= 0) {
14                     write(fd, &j, sizeof(j));
15                     close(fd);
16                     unlink(filename);
17                 }
18             }
19             exit(0);
20         }
21     }
22
23     // 等待所有子进程完成
24     for (int i = 0; i < 4; i++) {
25         wait(NULL);
26     }
27
28     printf("Concurrent access test completed

```

崩溃恢复测试

```

1 void test_crash_recovery(void) {

```



```

2     printf("Testing crash recovery...
3 ");
4 // 模拟崩溃场景：
5 // 1. 开始大量文件操作
6 // 2. 在中途"崩溃"（重启系统）
7 // 3. 检查文件系统一致性
8
9 // 注意：这个测试需要特殊的测试框架
10 // 可以通过修改内核代码来模拟崩溃

```

性能测试

```

1 void test_filesystem_performance(void) {
2     printf("Testing filesystem performance...\n");
3     uint64 start_time = get_time();
4
5     // 大量小文件测试
6     for (int i = 0; i < 1000; i++) {
7         char filename[32];
8         snprintf(filename, sizeof(filename), "small_%d", i);
9
10        int fd = open(filename, O_CREATE | O_RDWR);
11        write(fd, "test", 4);
12        close(fd);
13    }
14
15    uint64 small_files_time = get_time() - start_time;
16
17    // 大文件测试
18    start_time = get_time();
19    int fd = open("large_file", O_CREATE | O_RDWR);
20    char large_buffer[4096];
21    for (int i = 0; i < 1024; i++) { // 4MB文件
22        write(fd, large_buffer, sizeof(large_buffer));
23    }
24    close(fd);
25
26    uint64 large_file_time = get_time() - start_time;
27
28    printf("Small files (1000x4B): %lu cycles\n", small_files_time);
29    printf("Large file (1x4MB): %lu cycles\n", large_file_time);
30
31    // 清理测试文件
32    for (int i = 0; i < 1000; i++) {
33        char filename[32];
34        snprintf(filename, sizeof(filename), "small_%d", i);
35        unlink(filename);
36    }

```

```
37     unlink("large_file");
38 }
```

调试建议

文件系统状态检查

```
1 void debug_filesystem_state(void) {
2     printf("=== Filesystem Debug Info ===
3 ");
4     // 显示超级块信息
5     struct superblock sb;
6     read_superblock(&sb);
7     printf("Total blocks: %d
8 ", sb.size);
9     printf("Free blocks: %d
10 ", count_free_blocks());
11     printf("Free inodes: %d
12 ", count_free_inodes());
13
14     // 显示块缓存状态
15     printf("Buffer cache hits: %d
16 ", buffer_cache_hits);
17     printf("Buffer cache misses: %d
```

inode追踪

```
1 void debug_inode_usage(void) {
2     printf("=== Inode Usage ===\n");
3     for (int i = 0; i < NINODE; i++) {
4         struct inode *ip = &icache.inode[i];
5         if (ip->ref > 0) {
6             printf("Inode %d: ref=%d, type=%d, size=%d\n",
7                 ip->inum, ip->ref, ip->type, ip->size);
8         }
9     }
10 }
```

磁盘I/O统计

```
1 void debug_disk_io(void) {
2     printf("=== Disk I/O Statistics ===
3 ");
4     printf("Disk reads: %d
5 ", disk_read_count);
6     printf("Disk writes: %d
7 ", 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