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：

text

```
| 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：

c

```
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 bmpstart;   // 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：

```c
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：

```c
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：

```c
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：

```
c
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：

```
c
// 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：

```
c
struct buf {
    int valid;              // 缓存是否有效
    int dirty;              // 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：

```
c
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：

```
c
// 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:

c
```
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:

c
```
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:

c
```
// 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. 恢复过程如何确保原子性？
// 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：

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

2. 路径解析算法：

c
```
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：

c
```
// 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 *path, 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

c
```
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

```
c
void test_concurrent_access(void) {
    printf("Testing concurrent file access\n");

    // 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\n");
}
```

崩溃恢复测试

```
c
void test_crash_recovery(void) {
    printf("Testing crash recovery...\n");

    // 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

```
c
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

```
c
void debug_filesystem_state(void) {
    printf("== Filesystem Debug Info ==\n");

    // 显示超级块信息
    struct superblock sb;
    read_superblock(&sb);
    printf("Total blocks: %d\n", sb.size);
    printf("Free blocks: %d\n", count_free_blocks());
    printf("Free inodes: %d\n", count_free_inodes());

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

    // 显示日志状态
    printf("Log outstanding: %d\n", log.outstanding);
    printf("Log committing: %d\n", log.committing);
}
```

inode 追踪

```
c
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

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