武漢大學
WUHAN UNIVERSITY

第八节
系统调用

System Call

《操作系统实践A》

08 实验目的

- 什么是系统调用?
  - 用户程序请求操作系统内核服务的接口
  - 是用户态和内核态之间的桥梁
  - 提供了安全、可控的内核功能访问机制
- 实验目的:
  - 机制分析：深入理解 xv6 的系统调用分发机制。
  - 交互理解：掌握用户态与内核态的数据传输与特权级切换。
  - 框架实现：设计并实现一套完整的系统调用（Process, File, Memory）。

08 核心学习资料

武漢大學
WUHAN UNIVERSITY

- 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

文件	路径	说明
系统调用入口	kernel/syscall.c	系统调用分发表
陷阱处理	kernel/trap.c	中断和异常处理
用户态封装	user/usys.S	系统调用汇编存根
系统调用定义	kernel/syscall.h	系统调用号定义

08 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

结构体	文件位置	作用
trapframe	kernel/proc.h	保存用户态寄存器
syscalls[]	kernel/syscall.c	系统调用函数指针数组
proc	kernel/proc.h	进程控制块

08 RISC-V系统调用约定

- 调用约定：
  - ecall指令
    - ✓ 功能：从用户态陷入到监督模式（S-mode）
    - ✓ 硬件行为：
    - ✓ pc → sepc（保存返回地址）
    - ✓ 异常原因 → scause（值为8：来自U-mode的ecall）
    - ✓ stvec → pc（跳转到陷阱处理程序）
    - ✓ 特权级：U-mode → S-mode
  - 寄存器使用
  - 参数传递

寄存器	用途	说明
a7	系统调用号	标识要调用的系统调用
a0-a5	参数1-6	传递最多6个参数
a0	返回值	系统调用的返回结果

08 RISC-V系统调用约定

- 调用约定：
  - 特权级切换：用户模式到监督模式的转换过程

用户态 (U-mode)

↓ ecall指令

【硬件自动操作】

- sepc ← pc
- scause ← 8
- sstatus.SPP ← U-mode
- sstatus.SPIE ← sstatus.SIE
- sstatus.SIE ← 0（关中断）
- pc ← stvec

↓

监督态 (S-mode)

- 执行 trampoline.S (uservec)
- 保存用户寄存器到 trapframe
- 切换到内核页表
- 调用 usertrap() → syscall()
- 执行具体系统调用
- 准备返回（userret）

↓ sret指令

【硬件自动操作】

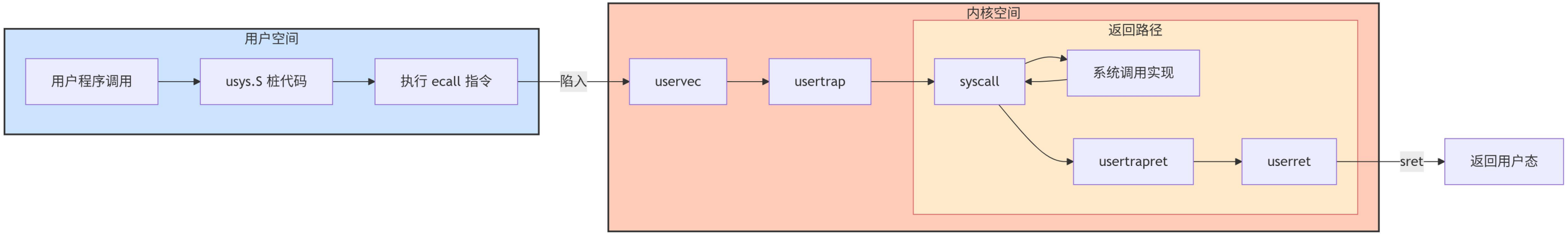
- pc ← sepc
- 特权级 ← sstatus.SPP
- sstatus.SIE ← sstatus.SPIE

↓

用户态 (U-mode)

08 任务1：理解系统调用的实现原理

- 分析系统调用的完整流程：



- 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？

08 任务1：理解系统调用的实现原理

- **每个环节的作用是什么?**
- 1. 用户程序调用：应用程序调用系统调用封装函数（如 write()），传入参数。
- 2. usys.S 桩代码：将系统调用号加载到 a7 寄存器，执行 ecall 指令触发陷阱。
- 3. 执行 ecall 指令：CPU 保存当前 PC 到 sepc，设置异常原因到 scause，切换到内核态并跳转到 stvec 指向的 trampoline。
- 4. uservec (trampoline.S)：保存所有用户寄存器到 trapframe，切换到内核页表和内核栈，跳转到 usertrap()。
- 5. usertrap (trap.c)：检查 scause 确认是系统调用，调用 syscall() 进行分发处理。
- 6. syscall (syscall.c)：从 trapframe->a7 获取系统调用号，通过函数指针数组调用对应的系统调用实现函数。
- 7. 系统调用实现：执行具体的系统调用功能（如 sys_write() 执行写操作），返回结果。
- 8. usertrapret (trap.c)：准备返回用户态，设置相关寄存器状态，调用 userret()。

08 任务1：理解系统调用的实现原理

- **每个环节的作用是什么?**
- 9. userret (trampoline.S)：切换回用户页表，从 trapframe 恢复所有用户寄存器，执行 sret 返回用户态。
- 10. 返回用户态：sret 指令恢复 PC 和特权级，用户程序从系统调用处继续执行。
- **参数是如何传递的?**
- 用户程序将参数放入寄存器 a0-a5（最多6个参数），uservec 将这些寄存器保存到 trapframe，内核通过 argint()、argaddr() 等函数从 trapframe 中读取参数。对于指针参数，内核使用 copyin()/copyout() 安全地访问用户内存空间。
- **返回值如何返回?**
- 系统调用实现函数返回结果后，syscall() 将返回值写入 trapframe->a0，userret 从 trapframe 恢复 a0 寄存器到用户态，用户程序通过 a0 获得返回值。成功时返回 ≥ 0，失败时返回 -1。

08 任务1：理解系统调用的实现原理

- 研究RISC-V的ecall机制：
  - **ecall指令的作用**：RISC-V 的环境调用指令，用于触发从用户态到内核态的切换。执行时硬件自动完成：保存当前 PC 到 sepc，设置异常原因码到 scause（系统调用为8），关闭中断，切换特权级到 S-mode，并跳转到 stvec 指向的陷阱处理入口。
  - **scause寄存器中系统调用的编码**：scause 寄存器记录陷阱发生的原因。当用户态执行 ecall 触发系统调用时，scause 被硬件设置为 8（Environment call from U-mode）。内核通过读取 scause 的值来区分是系统调用、中断还是其他异常，从而进行相应处理。
  - **sepc寄存器的作用和更新**：sepc (Supervisor Exception Program Counter) 保存触发陷阱时的指令地址。执行 ecall 时，硬件将当前 PC 值保存到 sepc。系统调用处理完毕后，内核将 sepc + 4（跳过 ecall 指令），然后执行 sret 指令，硬件自动将 sepc 的值恢复到 PC，使程序从系统调用的下一条指令继续执行。

08 任务1：理解系统调用的实现原理

- 理解特权级切换：
  - **用户栈到内核栈的转换：**系统调用触发时，CPU 从用户栈切换到内核栈。sscratch 寄存器保存 trapframe 地址，uservec 通过它获取内核栈指针 sp，完成栈的切换，确保内核操作不污染用户栈空间。
  - **寄存器状态的保存和恢复**
    - 保存阶段：uservec 将所有用户寄存器（32个通用寄存器 + PC）保存到 trapframe 结构中。
    - 恢复阶段：userret 在返回前从 trapframe 恢复所有寄存器，确保用户程序状态完整恢复，就像系统调用从未发生过一样。
  - **页表的切换时机**
    - 进入内核：uservec 中从用户页表切换到内核页表（写 satp 寄存器），使内核能访问内核地址空间。
    - 返回用户：userret 中切换回用户页表，执行 sret 后程序运行在用户地址空间。

08 任务1：理解系统调用的实现原理

- 深入思考：
  - **为什么需要陷阱帧(trapframe)?**
    - 保存用户态上下文 - 用户的32个寄存器、PC等状态必须完整保存，否则系统调用返回后程序无法继续执行
    - 实现状态隔离 - 内核和用户使用不同的栈和寄存器，trapframe 是两者之间的"中转站"，避免相互污染
    - 支持参数传递和返回 - 系统调用参数（a0-a7）和返回值都通过 trapframe 在用户态和内核态之间传递
  - **系统调用和中断处理有什么相同和不同?**
    - 系统调用是程序主动请求服务，中断是硬件强制打断程序执行。

08 任务2：分析xv6的系统调用分发机制

- 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?

08 任务2：分析xv6的系统调用分发机制

- 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？

08 任务2：分析xv6的系统调用分发机制

- 理解用户内存访问：
  - copyout() 和 copyin() 的作用
    - copyout() - 将数据从内核空间复制到用户空间，copyin() - 将数据从用户空间复制到内核空间，这两个函数在复制过程中会进行安全检查：验证用户地址是否合法、是否越界、页表映射是否存在，防止内核访问非法内存。
  - 为什么不能直接访问用户内存?
    - 地址空间隔离 - 内核和用户使用不同页表，用户虚拟地址在内核页表中无效或映射到错误位置，安全风险 - 用户可能传递恶意指针（如内核地址、空指针），直接访问会导致内核崩溃或安全漏洞，权限检查 - 需要验证用户是否有权限访问该内存区域（读/写权限、地址范围）
  - 如何防止用户传递恶意指针?
    - 地址范围检查，页表验证，权限检查，边界检查

08 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：
  - 如何验证用户提供的指针?
  - 如何处理系统调用失败?
  - 如何支持可变参数的系统调用?
  - 如何实现系统调用的权限检查?

```
// 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);
```

08 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
  - 2. 文件操作类
  - 3. 内存管理类

08 任务4：实现基础系统调用

- 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
```

08 任务4：实现基础系统调用

武漢大學
WUHAN UNIVERSITY

- 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
```

08 任务4：实现基础系统调用

- 3. 内存管理类：

```
void* sys_sbrk(void); // 调整堆大小
```

- 可选：mmap, munmap等

08 任务4：实现基础系统调用

- 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);
}
```

08 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
```

08 任务5：实现用户态系统调用接口

- 参考xv6的usys.pl，理解：
- 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?

08 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?
}
```

任务6：系统调用安全性

- 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

08 任务6：系统调用安全性

- 3. 权限检查：
  - 文件访问权限
  - 进程操作权限
  - 资源使用限制

任务6：系统调用安全性

- 4. 竞态条件防护：
  - TOCTTOU攻击防护
  - 原子操作保证
  - 锁的正确使用

- 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);
}
```

08 调试建议

- 系统调用跟踪
- 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

```
// 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...
}
```

08 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？

武漢大學
WUHAN UNIVERSITY

第九节
文件系统

File System

《操作系统实践A》

09 实验目的

- 通过深入分析 xv6 操作系统文件系统（FS, File System）的实现细节，掌握现代操作系统文件系统的基本结构、核心机制与一致性保障方法，最终能够 **独立设计并实现一个功能完善的带日志的文件系统（Journaling File System）。

- 主要学习目标
  - 1. 理解文件系统的基本原理
  - 2. 掌握 xv6 文件系统的架构与实现
  - 3. 实现写前日志系统（Write-Ahead Logging, WAL）
  - 4. 设计与实现自己的日志文件系统

09 核心学习资料

武漢大學
WUHAN UNIVERSITY

- 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

09 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

09 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
  - virtio-blk的核心优势
  - 高性能：半虚拟化： Guest OS（虚拟机内的系统）知道自己在虚拟化环境中运行，并安装了专门的virtio-blk驱动。它与QEMU的Hypervisor通过一个高效的、基于共享内存和环形缓冲区的通信机制（virtqueue）交换数据。
  - 对比： 相比于模拟真实硬件（如SATA控制器），它避免了大量不必要的硬件仿真和中断开销，I/O路径更短，性能显著更高。
  - 低开销： 减少了VM Exits（虚拟机退出），降低了CPU占用率。

- QEMU磁盘模拟：virtio-blk设备的使用
  - 终端 1：启动QEMU并等待调试器连接。

```
qemu-system-riscv64 -machine virt -nographic -bios none -kernel my_os.elf -s -S
```

  - 终端 2：启动GDB并连接。

```
riscv64-unknown-elf-gdb my_os.elf
(gdb) target remote localhost:1234
(gdb) b _start  # 在起始处设置断点
(gdb) continue   # 开始执行
```

  - 使用VirtIO块设备：为QEMU虚拟机添加一个VirtIO块设备，并关联到虚拟硬盘镜像 fs.img

```
qemu-system-riscv64 -machine virt -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0
```

09 任务1：理解xv6文件系统布局

武漢大學
WUHAN UNIVERSITY

- 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?

09 任务1：理解xv6文件系统布局

- 学习重点：

区域名称	块范围	作用	内容	特点/工作机制
引导块 (Boot Block)	块0	存储系统启动代码	引导加载程序 (bootloader)	• 仅第0块用于引导 • 非启动系统时此块可为空 • 确保系统正确加载内核
超级块 (Super Block)	块1	存储文件系统元数据	• 文件系统大小(总块数) • 数据块数量 • inode数量 • 日志区域大小 • 魔数(识别文件系统类型)	• 操作系统首先读取此块 • 包含文件系统关键结构信息
日志区域 (Log Blocks)	块2到?	实现事务性写入，保证文件系统一致性	事务操作记录	工作方式: 1. 写入操作先记录到日志 2. 操作完成后提交日志 3. 实际写入目标位置 4. 崩溃恢复时重放或撤销 大小: 通常固定数量块(如30块)

09 任务1：理解xv6文件系统布局

区域名称	块范围	作用	内容	特点/工作机制
inode块 (Inode Blocks)	?到?	存储文件元数据	每个inode包含: • 文件类型 • 文件大小 • 链接计数 • 数据块指针 • 权限信息	组织方式: • 每个inode固定大小(256字节) • 每个块包含多个inode • 通过索引快速访问
位图块 (Bitmap Block)	块?	跟踪数据块的使用状态	位图数据	工作机制: • 每个位对应一个数据块 • 0 = 空闲, 1 = 已使用 • 快速查找空闲块 数量: 通常1个块足够管理大量数据块
数据块 (Data Blocks)	?到end	存储实际文件内容	• 文件数据 • 目录条目(文件名+inode编号) • 间接块指针	管理方式: • 通过位图分配和释放 • 支持直接和间接指针 • 存储用户文件和目录数据

09 任务1：理解xv6文件系统布局

- 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?

09 任务1：理解xv6文件系统布局

➢ 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

09 任务1：理解xv6文件系统布局

- 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？

09 任务2：分析xv6的inode管理机制

- 代码阅读：
- 1. 研读inode缓存管理：
  - 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

```
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];
};
```

09 任务2：分析xv6的inode管理机制

- 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

09 任务2：分析xv6的inode管理机制

- 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

09 任务2：分析xv6的inode管理机制

- 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？

09 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
- 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?

```
// 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）
};
```

09 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
};
```

09 任务4：实现块缓存系统

- 参考xv6的bio.c，理解：
  - 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
```

09 任务4：实现块缓存系统

- 实现考虑的问题：
  - 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?

```
// 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);
```

09 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
};
```

任务5：实现日志系统

- 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
```

09 任务5：实现日志系统

- 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?

```
// 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);
```

09 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
}
```

09 任务6：实现目录和路径解析

- 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

```
// 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);
```

- 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”);
}
```

测试与调试策略

- 崩溃恢复测试，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

```
void test_crash_recovery(void) {
    printf("Testing crash recovery…");
}
```

- 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;
```

- 性能测试

```
// 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);
```

- 性能测试

```
    // 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");
}
```

09 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", buffer_cache_misses);
}
```

调试建议

- 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);
        }
    }
}
```

09 调试建议

- 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);
}
```

09 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？