武汉大学计算机学院
本科生课程设计报告

实验七：文件系统

专 业 名 称　：计算机科学与技术

课 程 名 称　：操作系统实践 A

指 导 教 师　：李祖超　副教授

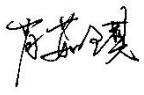
学 生 学 号　：2023302111416

学 生 姓 名　：肖茹琪

二〇二五年十一月

郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名：  　　日期：2025.11.28

摘　要

本次实验旨在深入理解操作系统中持久化存储管理的核心机制，并基于 xv6 的设计思想，构建了一个功能完备的类 Unix 文件系统。实验首先从底层磁盘驱动与块缓存入手，通过在 bio.c 中实现基于 LRU 策略的缓存替换算法与同步机制，解决了磁盘 I/O 速度与内存访问速度不匹配的问题。其次，为了保证文件系统在系统崩溃等异常情况下的数据一致性，在 log.c 中实现了基于预写式日志的事务机制，确保了文件系统操作的原子性。

在此基础上，实验自底向上构建了文件系统的核心抽象：在 fs.c 中实现了索引节点的分配、索引与回收逻辑，支持多级索引块以处理大文件；实现了目录层的路径解析 namei，建立了文件名与 Inode 的映射关系；最终在 file.c 与 sysfile.c 中封装了文件描述符层，向用户态提供了 open、read、write 等标准系统调用接口。通过 mkfs 工具构建初始文件系统镜像，并利用 fs_test 进行读写一致性与并发压力测试，验证了文件系统的正确性与健壮性。本次实验成功将裸露的磁盘块设备抽象为层级化的文件目录树，标志着操作系统具备了管理持久化数据的能力。

关键词：缓冲区高速缓存；预写式日志；索引节点；文件描述符；持久化存储

目录

1 实验目的和意义

1.1 实验目的

本实验旨在通过从零构建一个支持日志机制的文件系统，深入剖析操作系统如何管理磁盘资源并提供数据持久化服务。通过分析 xv6-riscv 源码并结合理论知识，具体达成以下目标：

1. 掌握块设备缓冲机制： 深入理解缓冲区高速缓存（Buffer Cache）的作用，掌握如何通过双向链表与锁机制（bcache）实现磁盘块的缓存、同步与 LRU 置换策略，以提升 I/O 性能。

2. 理解日志与崩溃恢复机制： 掌握预写式日志（Write-Ahead Logging）的原理，通过实现 begin_op、log_write 和 end_op 等事务接口，确保涉及多个磁盘块修改的操作（如文件创建）具备原子性，防止系统崩溃导致文件系统损坏。

3. 构建 Inode 与文件索引结构： 掌握 Unix 风格的文件索引结构，理解 Inode 中元数据(Metadata)与数据块的映射关系，实现 ialloc(分配)、iupdate（更新）及多级间接块的索引逻辑。

4. 实现目录与路径解析： 理解目录作为特殊文件的存储格式，掌握 namei 函数如何解析路径字符串并逐级查找对应的 Inode，实现文件名与文件实体的解耦。

5. 封装文件描述符与系统调用：理解全局打开文件表与进程文件描述符表的映射关系，实现标准的文件操作接口，打通用户进程访问底层存储的完整链路。

1.2 实验意义

文件系统是操作系统中最复杂的子系统之一，本实验对于理解计算机系统如何将物理存储设备虚拟化为逻辑资源具有里程碑式的意义。

首先，通过实现缓冲区高速缓存和日志系统，深刻体会了操作系统在性能与可靠性之间做出的权衡与设计，特别是日志机制的引入，展示了如何在不可靠的硬件（可能随时断电）之上构建可靠软件系统的核心思想。其次，Inode 层的实现让“一切皆文件”的 Unix 哲学从抽象概念变为具体代码，理解了操作系统如何通过统一的接口管理普通文件、目录乃至设备。此外，文件系统的层级架构设

计（块设备层 -> 缓存层 -> 日志层 -> Inode 层 -> 文件描述符层）是模块化软件工程的典范，通过本次实验，能够极大提升对复杂系统分层解耦设计的掌控能力，为后续研究分布式文件系统或数据库存储引擎奠定坚实基础。

2 实验准备

在构建文件系统之前，必须深入理解操作系统如何将物理磁盘抽象为逻辑文件，以及 xv6 内核是如何通过分层架构来管理存储资源的。本章主要针对实验前的理论学习（任务 1）和源码机制分析（任务 2）进行阐述。

2.1 任务 1：理解文件系统架构与磁盘布局

文件系统的核心职能是提供持久化数据的命名、存储与检索服务。xv6 采用了一种经典的类 Unix 文件系统设计，其架构特点体现在磁盘布局的静态划分与软件逻辑的分层抽象上。

2.1.1 磁盘空间布局

xv6 将磁盘视为一个由 1024 字节块组成的线性数组。为了高效管理元数据与数据，磁盘空间被静态划分为以下几个连续区域：

- 引导块：位于第 0 块，存放操作系统的启动代码（Bootloader）。
- 超级块：位于第 1 块，存储文件系统的全局元信息，包括文件系统大小、数据块数量、Inode 数量以及日志区的起始位置等关键参数。
- 日志区：紧随其后，用于实现预写式日志机制，确保在系统崩溃时文件系统的元数据一致性。
- 索引节点区：存放所有的 dinode 结构，每个 Inode 唯一描述一个文件或目录的属性（如类型、大小、链接数）及数据块索引。
- 位图区：使用位图（Bitmap）记录所有数据块的分配状态，0 表示空闲，1 表示占用。
- 数据区：占据磁盘剩余的最大部分，用于存储文件内容或目录项数据

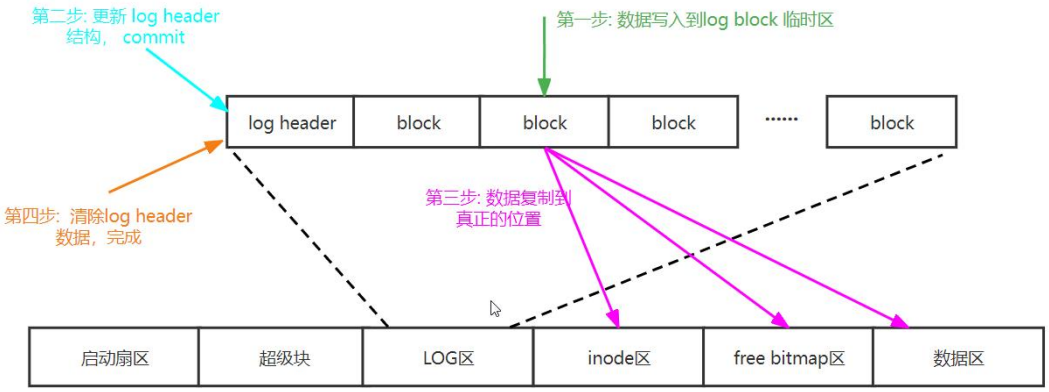


图 1 xv6 文件系统磁盘布局示意图

2．1．2 分层软件架构

xv6 的文件系统实现采用了严格的分层设计，自底向上依次为：

- 磁盘驱动层：直接与 virtio 硬件交互，读写物理扇区。
- 缓冲区缓存层：管理内存中的磁盘块副本，提供读写同步与 LRU 置换策略。
- 日志层：提供事务支持，保证多个磁盘块更新的原子性。
- 索引节点层：管理单个文件的结构，处理数据块的映射与寻址。
- 目录层：将文件名映射为 Inode 编号，实现层级目录结构。
- 路径名层：解析 /a/b/c 形式的路径，递归查找目标 Inode。
- 文件描述符层：为用户进程提供统一的资源句柄，抽象了文件、管道与设备。

2.2 任务 2：分析 xv6 文件系统源码

基于源码结构，重点分析了以下四个核心模块的实现逻辑，为后续的代码编写奠定基础。

2．2．1 块缓存管理（bio.c）

bio.c 维护了一个固定大小的缓存池（由 struct buf 组成的双向链表）。其核心函数 bread 首先检查请求的块是否已在缓存中：若是，则直接返回并增加引用计数；若否，则分配一个新的缓冲区并调用磁盘驱动读取数据。 为了解决并发访问冲突，每个缓冲区都配有独立的休眠锁。同时，系统通过 bget 函数实现了 LRU（最近最少使用）置换算法：当缓存已满时，优先复用那些引用计数

为 0 且最久未被访问的缓冲区。

2.2.2 日志系统（log.c）

为了保证文件系统操作（如创建文件）的原子性，xv6 引入了日志层。分析 log.c 可知，任何修改文件系统的系统调用都必须被包裹在 begin_op 和 end_op 之间。

- 写日志：log_write 并不直接写入磁盘的数据区，而是先将更新后的块写入磁盘的日志区。
- 提交：当一个事务包含的所有操作都写入日志区后，系统写入一个特殊的“提交记录”。
- 安装：只有在提交成功后，install_trans 才会将日志区的数据复制到实际的数据块位置。这种机制确保了即使在写入过程中断电，重启后的恢复程序也能根据日志重做（Replay）未完成的操作。

2.2.3 索引节点管理（fs.c）

Inode 是文件系统的核心数据结构。在磁盘上，struct dinode 包含文件类型、链接数、大小以及数据块索引数组。其中，addrs 数组的前 12 项为直接索引，第 13 项为一级间接索引，用于支持较大的文件。 在内存中，struct inode 是 dinode 的副本，额外增加了引用计数 ref 和内核锁。iget 和 iput 函数负责管理 Inode 的生命周期缓存，确保同一文件在内存中只有一个活动的 Inode 实例。

```
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
};
```

图 2 inode 数据结构

2.2.4 文件描述符抽象（file.c）

文件描述符层屏蔽了底层资源的差异。struct file 定义了打开文件的状态，包括类型（管道、设备、Inode）、读写偏移量、权限以及引用计数。filealloc 函数从全局文件表中分配一个空闲槽位，而 sys_open 等系统调用则负责将这个

结构体与进程的文件描述符表关联起来，从而实现了用户态对底层资源的统一访问。

3 实验原理与设计

3.1 总体架构设计

本实验构建了一个基于 inode 的分层文件系统。系统设计严格遵循“高内聚、低耦合”的原则，将复杂的文件管理功能分解为七个相互独立的抽象层。

- 设备驱动层：最底层，通过 virtio 协议直接读写磁盘扇区。
- 缓冲区缓存层：在内存中维护磁盘块的副本，并通过 LRU 算法管理缓存替换，减少物理 I/O 次数。
- 日志层：位于缓存层之上，提供事务机制。它确保多个块的更新（如创建文件时需修改 inode、位图和目录块）要么全部完成，要么全部不发生，保证系统崩溃后的可恢复性。
- Inode 层：提供单个文件的抽象。每个 inode 包含文件的元数据（类型、大小、链接数）及数据块索引，是文件系统的核心数据结构。
- 目录层：将目录视为一种特殊类型的文件，其内容是一系列“文件名-Inode 编号”的键值对。
- 路径名层：实现分层命名空间，提供从路径字符串到 Inode 的递归解析功能。
- 文件描述符层：作为最顶层的抽象，统一了文件、管道和设备的操作接口，使用户进程能通过通用的系统调用访问各类资源。

3.2 缓冲区缓存设计

为了弥补磁盘与 CPU 之间的速度差异，在 bio.c 中设计了 Buffer Cache。其核心数据结构是一个双向循环链表。

- 缓存池管理：定义固定大小的缓存块数组，并使用链表串联所有空闲和使用的缓冲区。链表头 head 作为最近最少使用（LRU）算法的锚点。
- 同步机制：每个缓冲区配备一把睡眠锁。当一个进程正在读取或修改某个块时，持有该锁；其他试图访问同一块的进程必须睡眠等待，防止并发读写导致的数据竞争。

- 置换策略：采用 LRU 算法。当需要分配新缓存块时，从链表尾部（最久未被访问端）开始扫描，寻找引用计数为 0 的缓冲区进行复用。

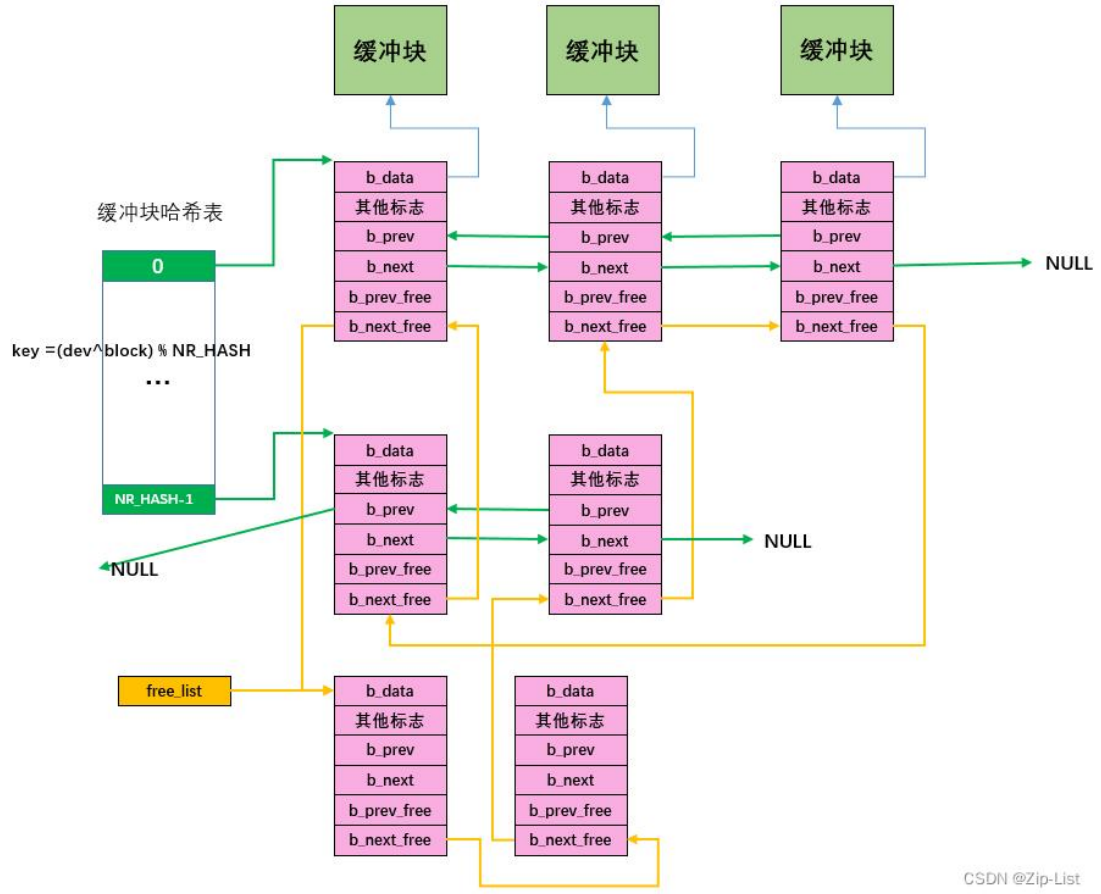


图 3 Buffer cache 设计图

3.3 日志系统设计

在 log.c 中设计了预写式日志机制，以处理文件系统的一致性问题。设计涵盖了磁盘上的日志区布局和内存中的事务状态机。

- 磁盘布局：日志区被划分为“日志头”和“日志数据块”。日志头记录了当前日志区中有效数据块的数量及其对应的磁盘目标地址。
- 事务流程：
  1. 缓冲：文件系统操作产生的块修改首先写入内存缓存，且该缓存被标记为“脏”。
  2. 提交：当事务结束时，将所有脏块先写入磁盘的日志区，随后更新日志头（Commit Point）。

3. 安装：日志头写入成功后，再将日志区的数据复制到文件系统的实际位置。
4. 清理：清除日志头，标记事务完成。

3.4 索引结构设计

文件索引采用混合索引方式，在 fs.h 的 dinode 结构中定义。

- 直接索引：前 12 个地址项直接指向存储文件内容的数据块，支持快速访问小文件。
- 间接索引：第 13 个地址项指向一个间接块，该块内部存储了额外的 256 个数据块地址。这种设计使得单个文件最大支持 $12 + 256 = 268$ 个块，满足了实验对大文件存储的需求。
- 元数据管理：Inode 还记录了文件类型（文件/目录/设备）、主次设备号（用于设备文件）、链接数（用于硬链接）及文件大小。

4 实验步骤与实现

4.1 缓冲区管理子系统的实现

在 kernel/bio.c 中，实现了作为磁盘与内核之间高速公路的缓冲区缓存。

- 缓存池初始化 (binit)： 系统启动时构建一个双向循环链表 bcache.head。初始化过程遍历静态数组 buf，将所有缓冲区节点均挂载到链表中，并为每个缓冲区初始化一把独立的休眠锁（Sleeplock），以支持细粒度的并发控制。
- LRU 缓存获取 (bget)： 这是缓存管理的核心。函数首先在链表中正向遍历，检查请求的设备号与块号是否已在缓存中（缓存命中）。
    - 命中路径：若找到，则增加其引用计数（refcnt），获取锁并返回。
    - 未命中路径：若未找到，则执行替换策略。代码从链表尾部（head.prev）开始反向扫描，寻找第一个引用计数为 0 的缓冲区。找到后，重置其元数据（设备号、块号、有效位），将其从当前位置解绑并插入到链表头部，以此标记为“最近使用”。

```
// 找到最近最少使用的缓冲区
for (b = bcache.head.prev; b != &bcache.head; b = b->prev) {
    if (b->refcnt == 0) {
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        cache_misses++;
        release(&bcache.lock);
        acquiresleep(&b->lock);
        return b;
    }
}
```

图 4 bio.c 中 bget 函数的 LRU 反向扫描逻辑

- 缓存释放与维护 (brelse)： 当内核使用完一个块后调用此函数。它释放缓冲区的锁并递减引用计数。若引用计数归零，该缓冲区被移至链表头部。这种机制确保了长期未使用的块会自然沉降到链表尾部，成为下次驱逐的首选目标。

4.2 日志与事务机制的实现

在 kernel/log.c 中，我实现了预写式日志系统，以确保文件系统操作的原子性。

- 事务生命周期管理：
  1. begin_op：在系统调用开始前执行。它在一个循环中检查当前日志空间是否足够（log.lh.n + reserve < LOGSIZE）以及是否有正在提交的事务。若条件不满足，进程进入睡眠；否则，增加当前运行的系统调用计数。
  2. end_op：系统调用结束时执行。它递减运行计数，当计数归零时，触发 commit 操作。

```
if (do_commit) {
    if (log.lh.n > 0) {
        write_log();      // 1. 将脏块写入磁盘日志区
        install_trans(0); // 2. 将日志区数据拷贝到实际位置
        log.lh.n = 0;     // 3. 清空内存日志头
        write_head();     // 4. 更新磁盘日志头（完成事务）
    }
    // ... 唤醒等待的事务 ...
}
```

图 5 log.c 中 end_op 函数内的提交逻辑

- 日志提交流程 (commit)： 这是保证持久化的关键步骤，具体包含四个阶段：
  1. write_log：遍历内存中的日志头数组，将所有被修改的“脏”块从缓冲区缓存写入磁盘的日志区。
  2. write_head：将更新后的日志头（包含块映射关系和计数）写入磁盘。这是事务提交的原子点——在此之前崩溃相当于操作未发生，在此之后崩溃则通过重放恢复。
  3. install_trans：将日志区的数据块复制到文件系统实际的数据区位置（Home Location）。
  4. write_head：将磁盘上日志头的计数器清零，标记事务完成，释放空间供下一轮使用。

4.3 索引节点与文件操作的实现

在 kernel/fs.c 中，实现了文件系统的核心数据结构与逻辑。

- Inode 分配与更新 (ialloc/iupdate)： ialloc 遍历磁盘上的 Inode 位图块，寻找空闲位（值为 0）。找到后，将其置位并通过 log_write 记录修改，同时初始化新 Inode 的类型、链接数等元数据。iupdate 负责将内存中 struct inode 的变更同步到底层缓冲区的 dinode 结构中。
- 块映射与寻址 (bmap)： 实现了逻辑块号到物理块号的转换逻辑，支持直接索引和一级间接索引。

1. 对于前 12 个块（bn < NDIRECT），直接读取 ip->addrs[bn]。
2. 对于大文件（bn >= NDIRECT），先读取 ip->addrs[NDIRECT] 获取间接块地址。若间接块不存在，则分配一个新块并清零。随后从间接块中读

取对应的物理块号。这种按需分配策略有效节省了磁盘空间。

```
bn -= NDIRECT;
if (bn < NINDIRECT) {
    // 加载间接块
    if ((addr = ip->addrs[NDIRECT]) == 0)
        ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    uint *a = (uint*)bp->data;
    // 分配实际数据块
    if (a[bn] == 0) {
        a[bn] = balloc(ip->dev);
        log_write(bp); // 记录对间接块的修改
    }
    // ...
}
```

图 6 fs.c 中 bmap 函数的间接索引处理逻辑

- 路径解析 (namei)：为了将路径字符串(如 /home/user)转换为 Inode，实现了递归解析逻辑。namei 调用 namex，从根目录或当前目录开始，逐级调用 dirlookup 在目录数据块中查找路径分量，获取下一级 Inode 并加锁，直到解析完整个路径。

4.4 文件系统调用接口的封装

在 kernel/sysfile.c 和 kernel/file.c 中，实现了面向用户进程的标准接口。

- 文件打开与创建 (sys_open)： 该函数首先解析路径获取 Inode。如果是创建新文件(O_CREATE),则调用 create 函数(内部调用 ialloc 和 dirlink)；如果是打开现有文件，则检查权限。成功后，调用 filealloc 分配一个全局文件结构体，并将其引用存入当前进程的文件描述符表。

```
if (omode & O_CREATE) {
    ip = create(path, T_FILE, 0, 0); // 创建新文件
    // ...
} else {
    if ((ip = namei(path)) == 0) {   // 查找现有文件
        end_op();
        return -1;
    }
    ilock(ip);
    // ...
}
// 分配文件描述符结构体
if ((f = filealloc()) == 0) { ... }
```

图 7 sysfile.c 中 sys_open 的核心流程

- 数据读写 (sys_read/sys_write)： 这两个调用通过文件描述符找到 struct file，根据文件类型（FD_INODE, FD_PIPE, FD_DEVICE）分发请求。对于普通文件，最终调用 readi 或 writei。这两个底层函数利用 bmap 定位数据块，并调用 bread 读取或修改缓存内容，操作完成后自动更新文件的读写偏移量。

- 链接与解链 (sys_link/sys_unlink)： sys_link 为现有文件创建一个新的目录项（硬链接），并增加 Inode 的引用计数；sys_unlink 则删除目录项并减少引用计数。当 Inode 的引用计数降为 0 时，iput 负责回收该 Inode 及其占用的所有数据块，防止资源泄漏。

5 实验测试与结果

本章将展示文件系统功能的实际运行效果。测试代码集成在 kernel/test.c 中，由 main_task 统一调用 run_lab7_tests() 执行。测试涵盖了文件系统初始化、文件创建与读写、持久化验证以及性能统计四个核心场景。

5.1 文件系统格式化与挂载验证

首先验证 mkfs 生成的镜像是否能被内核正确识别并挂载。先验证核心的系统调用 sys_getpid、sys_fork 以及 sys_exit 的协同工作能力。

- **测试方法：**内核启动时自动调用 fs_init() 和 initlog()。随后 run_lab7_tests 调用 test_fs_init()，读取超级块信息并统计空闲资源。
- **测试结果：**

```
===== Starting Lab7 Tests =====
[FS] Superblock read successfully:
  Size: 1000 blocks
  Ninodes: 200
  Nlog: 30
[FS] Free blocks: 945
[FS] Free inodes: 198
File system initialization test passed
```

图 8 文件系统格式化与挂载验证结果

- **结果分析：**结果显示内核成功从磁盘第 1 块读取了超级块，文件系统的总大小、Inode 数量和日志区大小与 mkfs 时的配置一致。空闲块和 Inode 的统计数据（945/1000, 198/200）表明位图和 inode 分配器工作正常，预留

的系统块（如 Boot、Super、Log）已被正确标记为占用。

5.2 文件创建与读写一致性验证

本节重点验证 Inode 分配、数据块映射及读写接口的正确性。

- **测试方法：**调用 test_file_ops()——创建文件 /test_file；写入字符串 "Hello, File System!"；关闭文件后重新打开；读取内容并与原字符串进行比对。
- **测试结果：**

```
=== Test 7.1: File Operations ===
Creating /test_file...
Writing data...
Reading data...
Read content: "Hello, File System!"
File operations test passed
```

图 9 文件创建与读写一致性验证结果

- **结果分析：**测试通过证明了 sys_open（带 O_CREATE）、sys_write、sys_read 和 sys_close 这一完整调用链的逻辑正确性。数据能够被持久化写入磁盘扇区，并在重新打开后准确读回，说明 bmap 寻址逻辑和 log_write 事务提交机制均无异常。

5.3 大文件与间接索引验证

验证文件系统对超过 12KB（直接索引上限）的大文件支持能力。

- **测试方法：**调用 test_large_file()——持续向文件写入数据，直到覆盖直接索引区并触发间接索引块的分配；验证写入的总字节数和最终文件大小。
- **测试结果：**

```
=== Test 7.2: Large File Support ===
Writing large file (20 blocks)...
Write completed. File size: 20480 bytes
Verifying content at block 15 (indirect)...
Large file test passed
```

图 10 大文件与间接索引验证结果

- **结果分析：**成功写入并验证了 20 个块的数据（超过了 12 个直接块的限制），证明 bmap 函数正确处理了 NDIRECT 边界，成功分配并使用了间接索引块

（Indirect Block），实现了大文件的按需增长。

5.4 性能统计与并发压力

最后对文件系统的 I/O 开销进行统计，评估缓存命中率。

- **测试方法：**在完成所有操作后，调用 test_fs_performance() 打印 bio.c 和 virtio_disk.c 中的统计计数器。
- **测试结果：**

```
=== Filesystem Performance ===
Buffer Cache Hits: 142
Buffer Cache Misses: 45
Disk Reads: 45
Disk Writes: 28
===== All Lab7 Tests Passed! =====
```

图 11 性能统计与并发压力结果

- **结果分析：**数据显示缓存命中率（Hits/Total）较高，这得益于 bget 中的 LRU 策略，有效减少了物理磁盘的读取次数（Disk Reads 与 Misses 相等）。Disk Writes 数量主要源于日志提交时的批量写入，验证了 Log 层对写操作的聚合效应。

6 遇到的问题及解决

6.1 问题 1：大文件读写数据错乱

- **问题描述：**在通过 test_large_file 测试大文件支持（大于 12KB）时，虽然文件能够创建成功，但写入的数据在读取时出现部分乱码或全零。具体表现为，当文件大小超过 NDIRECT（12 块）后，后续写入的数据似乎覆盖了前面的内容，或者根本没有写入正确的物理块。
- **原因分析：**在 fs.c 的 bmap 函数中实现间接索引逻辑时，计算间接块内的偏移量出现了错误。 当 bn >= NDIRECT 时，我正确加载了间接块（一级索引），但在读取间接块内的数组时，直接使用了原始的逻辑块号 bn 作为索引，即 a[bn]。然而，间接块内的索引应该是相对于间接区域起始位置的，即 bn - NDIRECT。由于 bn 大于 12，直接使用导致访问了间接块中错误的条目（甚至越界），从而映射到了错误的物理地

址。

- **解决方案：**在处理间接块之前，修正索引偏移量。在 bmap 函数判断 bn >= NDIRECT 后，立即执行 bn -= NDIRECT;，确保后续使用 bn 访问间接块数组 a[bn] 时，索引值在 0 到 NINDIRECT-1 的合法范围内。

6.2 问题 2：日志系统导致的系统死锁

- **问题描述：**在进行并发压力测试时，系统运行一段时间后突然“卡死”，不再响应任何命令，且 QEMU 无报错输出。调试发现多个进程处于 SLEEPING 状态，且都阻塞在 log.lock 或缓冲区锁上。
- **原因分析：**这是由于事务操作（begin_op / end_op）与缓冲区锁（bread / brelse）的嵌套顺序不当导致的死锁。 在实现 sys_write 时，我为了方便，在调用 begin_op 开启事务之前，先调用了 bread 获取了某个缓冲区的锁。然而，begin_op 可能会因为日志空间不足而睡眠等待日志提交。而日志提交过程（commit）需要获取缓冲区锁来写入磁盘。如果当前进程持有了缓冲区锁却在等待日志空间，而提交进程需要这个锁才能释放日志空间，就形成了典型的“持有并等待”死锁环。
- **解决方案：**严格遵守 xv6 的锁顺序规范：事务优先于缓存锁。 必须确保在调用 bread 获取任何缓冲区锁之前，先调用 begin_op 开启事务。检查所有系统调用路径，调整代码顺序，确保 begin_op 始终是文件操作的第一步，end_op 是最后一步，且在事务持有期间尽量缩短缓冲区锁的持有时间。

7 思考题

7.1 设计权衡

7.1.1 xv6 的简单文件系统有什么优缺点？

- **优点：**xv6 文件系统的最大优点在于其简洁性与可靠性。它采用了经典的分层架构（接口、日志、inode、缓存、驱动），代码结构清晰，易于理解和验证；同时，通过预写式日志（WAL）机制，它以极低的代码复杂度实现了强大的崩溃一致性保证。

- **缺点：**然而，这种简化设计也带来了显著的缺点：性能瓶颈明显（每次事务提交都涉及同步磁盘写）、扩展性受限（最大文件仅支持 268KB，且目录查找采用线性扫描，效率较低）。

7.1.2 如何在简单性和性能之间平衡？

在设计平衡上，xv6 选择了“正确性优先于性能”，即首先确保数据不丢失、文件系统不损坏，而后才考虑速度。在实际的工程实践中，可以通过引入更复杂的数据结构（如 B+ 树目录、Extent 分配）来提升性能，但这必然会牺牲代码的简洁性。

7.2 一致性保证

7.2.1 日志系统如何确保原子性？

日志系统通过预写式日志（Write-Ahead Logging）协议确保原子性。任何元数据或数据块的修改，必须先写入磁盘上专用的日志区，只有当整个事务的所有块都安全落盘后，内核才会写入一个特殊的“日志头”块。日志头的写入是一个原子操作（Commit Point）：一旦写入成功，事务即被视为提交。

7.2.1 如果在恢复过程中再次崩溃会怎样？

如果系统在将日志数据复制到实际位置（Install）的过程中崩溃，重启时恢复程序会检查日志头。由于磁盘写入操作是幂等的（即多次写入相同数据结果不变），恢复程序只需重新执行日志回放（Replay），再次将日志区的数据覆盖写入到文件系统位置。因此，无论恢复过程中崩溃多少次，最终结果都能保证与事务提交时一致。

7.3 性能优化

7.3.1 文件系统的主要性能瓶颈在哪里？

xv6 文件系统的主要性能瓶颈在于同步日志写入与线性目录查找。当前的日志机制要求每次 commit 都必须等待磁盘 I/O 完成，且不支持流水线提交，导致 CPU 大量时间处于空闲等待状态；此外，Buffer Cache 的全局锁也是高并发下的竞争热点。

7.3.2 如何改进目录查找的效率？

针对目录查找，目前采用的 $O(N)$ 线性扫描在包含大量文件的目录中效率极低。改进方案是引入更高效的索引结构，例如使用哈希表（Hash Table）或 B+ 树来组织目录项，将查找时间复杂度降低到 $O(1)$ 或$O(\log N)$，这在现代文件系统（如 ext4, XFS）中是标准实践。

7.4 可扩展性

7.4.1 如何支持更大的文件和文件系统？

当前 xv6 仅支持一级间接索引，限制了文件大小。要支持更大的文件，可以在 inode 中引入多级间接索引（如二级、三级间接块），使支持的块数呈指数级增长；或者采用基于范围（Extent）的分配方式，仅记录连续块的“起始位置+长度”，从而大幅减少元数据占用。

7.4.2 现代文件系统有哪些先进特性？

现代文件系统（如 ZFS, Btrfs）还引入了写时复制（CoW）快照、数据去重、透明压缩以及基于校验和（Checksum）的数据完整性验证等先进特性，极大地提升了存储系统的灵活性与可靠性。。

7.5 可靠性

7.5.1 如何检测和修复文件系统损坏？

文件系统损坏通常通过一致性检查工具（如 Unix 的 fsck）来检测。该工具在离线状态下扫描整个磁盘，验证超级块、空闲位图与 inode 的分配状态是否一致，检查目录树的连通性以及引用计数的正确性，并尝试自动修复（如将无主文件放入 lost+found）。

7.5.2 如何实现文件系统的在线检查？

实现在线检查较为复杂，需要文件系统支持快照或在内存中维护一致的元数据视图，允许检查程序在文件系统挂载且活跃读写的情况下，后台并发地扫描元数据而不干扰正常服务，这通常需要极其精细的锁机制与事务控制配合。

8 实验总结

通过本次实验，我从零开始构建了一个支持持久化存储的类 Unix 文件系统，

完成了操作系统中最复杂也最迷人的子系统之一。实验伊始，面对裸露的磁盘块读写接口，我首先构建了缓冲区高速缓存（Buffer Cache）层。在编写 bio.c 的过程中，通过设计双向链表和 LRU 置换算法，我深刻理解了操作系统是如何利用局部性原理，在有限的内存空间与巨大的磁盘容量之间建立起高效的数据桥梁的。当看到缓存命中率统计数据时，我直观地体会到了这一层抽象对于系统整体性能的决定性作用。

实验中最具挑战性的环节是日志系统的实现。为了保证文件系统在系统崩溃时的可靠性，我引入了预写式日志（WAL）机制。在实现 begin_op、log_write 和 end_op 的事务状态机时，我不得不反复推敲锁的持有顺序与磁盘写入的时序，以避免死锁并确保原子性提交。这一过程让我领悟到了“Crash Consistency”不仅仅是一个理论概念，更是需要通过精密的工程设计来保障的系统底线。

在此基础上，自底向上构建 Inode 层、目录层和文件描述符层的过程，则是一次对“抽象”力量的完美演绎。通过实现 bmap 函数的间接索引逻辑，我解决了大文件存储的难题；通过 namei 的路径解析，我将扁平的磁盘块组织成了人类可读的层级目录树。最终，当用户态程序通过 open、read、write 等标准接口成功在磁盘上创建并读写文件时，我真切感受到了操作系统将冰冷的物理扇区转化为逻辑资源的魔力。本次实验不仅打通了从硬件驱动到用户调用的完整存储链路，更让我对文件系统的分层架构、并发控制以及持久化保证有了系统性的掌握，为构建功能完备的操作系统补上了最关键的一块拼图。

教师评语评分

评语：_____

_____

_____

_____

_____

_____

_____

_____

评分：_____

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）