综合实验报告 Lab7：文件系统

零、实验概述

1. 实验目标

- 搭建基于 virtio block 的块设备层与 32 块缓存，确保所有磁盘访问都走 `bread/bwrite`；
- 实现写前日志(`begin_op/log_write/end_op`) 保证崩溃一致性；
- 完整落地 inode/目录/路径解析/位图分配，支持直接+间接块；
- 通过课件提出的完整性、并发、性能、自检等测试，验证实现符合 Lab7 要求。

2. 完成情况

- irtio block 驱动 + 块缓存 + I/O 统计；
- 日志层 commit/recover 流程，支持并发 begin/end；
- inode/目录/路径/位图函数与 mkfs 工具；
- `lab7_*` 测试：integrity/concurrent/performance/debug；
- 未完成项：双重间接块、在线 fsck 仍在 TODO 列表。

3. 开发环境

- 硬件：x86_64 主机，QEMU virt (2 vCPU, 256 MB)；
- 操作系统：Ubuntu 24.04 LTS；
- 工具链：`riscv64-unknown-elf-gcc 12.2.0`, `gdb-multiarch 15.0.50`；
- 模拟器：`qemu-system-riscv64 8.2.2 -machine virt -nographic -global virtio-mmio.force-legacy=false`；
- 构建命令：`make && make qemu`，调试：`make qemu-gdb`。

一、系统设计

1. 架构设计

- **块设备与缓存层（kernel/dev/virtio_disk.c, kernel/fs/bio.c）**
  virtio 驱动初始化 modern MMIO 队列，通过 `virtio_disk_rw()` 把块请求交给硬件；`bio.c` 在此之上构建带 LRU 的 32 块缓存，所有磁盘访问都先经过 `bread()/bwrite()`，并维护 `disk_read_count/disk_write_count` 供调试。

- **日志层（kernel/fs/log.c）**
  写前日志实现 `begin_op()/log_write()/end_op()` 接口，`log_state` 在 `log_init()` 中根据超级块决定日志区域起止；`commit()` 负责按"复制日志 → 写日志头 → 安装数据 → 清空日志头"的顺序保证崩溃恢复的原子性。

- **文件系统内核（kernel/fs/fs.c、dir.c、file.c）**
  `fs.c` 管理超级块、inode 缓存、位图分配与 `readi/writei`；`dir.c` 负责目录项解析与路径遍历；`file.c` 提供文件描述符层。调用路径为：系统调用或内核测试 → `file.c`/`dir.c` → inode 层 → `bread/bwrite` → virtio。

- **用户接口与测试（kernel/boot/main.c、tools/mkfs.py）**
  `mkfs.py` 构建 8 MB 镜像（8192 块），填充超级块、inode 区、位图与根目录；`main.c` 提供 `lab7_open_file()`、`lab7_unlink()` 等辅助函数，并自动串行执行完整性/并发/性能测试和调试输出。

2. 关键数据结构

数据结构	位置	说明
`struct superblock`	include/fs/fs.h	描述磁盘布局（总块数/数据块数/inode/log 起始块）。`fs_init()` 会校验 magic 并缓存副本。
`struct inode` / `struct dinode`	kernel/fs/fs.c	`inode` 为内存态条目（含锁和引用计数），`dinode` 为磁盘版，持久化类型、大小、直接/间接块地址（11+1）。
`struct buf`	kernel/fs/bio.c	块缓存节点，带 `valid/disk/refcnt` 与双向链表指针，实现最近最少使用替换及 `bpin/bunpin`。
`struct log_state` / `struct logheader`	kernel/fs/log.c	记录日志范围、挂起操作数、提交状态及已记录块号，`lh.block[]` 保存待提交的真实块。
`struct dirent`	include/fs/dir.h	目录项，保存文件名与 inode 号，路径解析依赖它进行线性扫描。

3. 与 xv6 的对比

1. **设备与构建链**：保留 xv6 的 virtio block 思路，但驱动初始化 modern 接口并明确禁止 legacy（`-global virtio-mmio.force-legacy=false`），避免 PFN 相关歧义。
2. **工具链**：`tools/mkfs.py` 通过 Python `ctypes` 和结构体计算布局，并一次性写入 `./..` 目录与位图；相比 xv6 的 C 版 mkfs 更易拓展与调试。
3. **调试可观测性**：新增 `debug_filesystem_state()`、`debug_inode_usage()` 与磁盘 I/O 计数器，可在 panic 前打印超级块、空闲块/inode、读写次数，有助于定位 Lab7 问题。
4. **测试驱动**：xv6 靠用户态程序测试，本实验将测试流程直接集成在 `main.c`，确保 `make qemu` 后自动验证文件系统。

4. 设计决策

1. **事务边界**：`lab7_open_file()` 在进入文件系统前即 `begin_op()`，确保任意测试都在事务中执行，避免日志溢出。
2. **缓存容量**：短期保持 `NBUF=32`，并用 `LAB7_CONCURRENT_*` 宏限制自测写入规模；后续若扩容镜像可以把缓存调大。
3. **路径解析策略**：沿用线性 `namex()`，并允许父目录引用计数与锁配合，保证 `lab7_unlink()` 的并发安全。
4. **根目录兜底**：`fs_init()` 检测到根 inode 类型为空时会自动创建 `./..` 与首个数据块，确保镜像损坏时仍能自愈启动。

二、实验过程

1. 实现步骤

1. **virtio block 驱动落地**：初始化描述符表与 `virtq`，通过 `kvaddr_to_pa()` 提供 DMA 可见地址，并在每次提交后写 `VIRTIO_MMIO_QUEUE_NOTIFY`。

2. **块缓存与统计**：`binit()` 把 32 个 `struct buf` 串成环形链表，`bget()` 先查命中再从尾部回收空闲块，所有 I/O 都通过 `virtio_disk_rw()` 完成并计入 `disk_*_count`。
3. **超级块与 inode 缓存**：`fs_init()` 调 `read_superblock()` 读 block1，随后 `iinit()` 初始化 50 个 inode 缓存条目，每个 inode 有独立 sleeplock。
4. **位图分配逻辑**：`balloc()` 逐块遍历位图块（`BBLOCK` 宏），用 bit 操作找到未使用块并立即清零；`bfree()` 则复位相应 bit，所有修改都通过 `log_write()` 落日志。
5. **inode_bmap 与读写路径**：支持 11 个直接块 + 1 个一级间接块，`readi/writei` 在循环中根据偏移取块、读写数据并在写时更新 inode size。
6. **目录 & 路径接口**：`dirlookup/dirlink` 封装目录扫描和插入，`namex()` 负责路径拆分、处理相对路径（引用进程 `cwd`）；`inode_create()` 统一创建文件/目录。
7. **测试与工具**：`lab7_*` 系列函数包装 `filealloc()` 等接口，`test_filesystem_*` 在启动阶段运行；`tools/mkfs.py` 计算布局并写入根目录和位图，确保 `fs_init()` 可顺利 mount。

2. 遇到的问题与解决方案

问题	定位与修复
VirtIO `used_idx` 不前进	发现 QEMU 默认是 legacy 模式，而驱动按 modern MMIO 写寄存器；在 `Makefile` 启动参数中加入 `-global virtio-mmio.force-legacy=false`，并仅保留 modern 路径。
`panic: ilock: no type`	旧版 mkfs 只写超级块导致根 inode `type=0`，在 `ilock()` 读取后 panic；重写 `tools/mkfs.py`，确保根目录 inode 与数据块、位图都初始化。
`bget: no buffers`	并发测试期间，`NBUF=32` 无法容纳所有 pin 住的块；通过减少 `LAB7_CONCURRENT_WORKERS/ITERS` 与 `LAB7_PERF_*` 的默认值，并在 README 中提示需要时增大 `NBUF`。
目录删除失败	初版 `lab7_unlink()` 未处理目录非空情况，删除目录后导致 `nlink` 计数错误；改为在写空目录项前调用 `lab7_is_dir_empty()` 并同步更新父目录 `nlink`。
日志耗尽	多个测试嵌套时会出现 “log full” panic；增加事务边界检查，让 `begin_op()` 在日志空间不足时睡眠等待 `end_op()`。

3. 源码理解概要

- **块分配 (`balloc/bfree`)**：通过 `BPB=BSIZE*8` 的位图块管理磁盘空间，查找空闲块后立刻写零并记录日志，保证事务中即看到干净块。
- **地址映射 (`inode_bmap`)**：直接块不足时分配一级间接块，间接表存放 1024 个物理块号，兼顾简单性与 4 MB 左右的大文件需求。
- **路径解析 (`namex`)**：逐个路径组件 `skip elem`，对父目录请求 (`nameiparent`) 在倒数第二层返回，避免额外扫描。
- **日志提交 (`commit`)**：复制数据到日志区 → 写日志头 → 安装事务 → 清空日志头；恢复时 `recover_from_log()` 只需读取日志头、安装并清零即可，具备幂等性。

三、测试与验证

1. 测试

测试	描述	样例结果（`make qemu`，LAB7 宏为 2/2/1）
`test_filesystem_integrity`	打开 `testfile` 写入 `"Hello, filesystem!"`，再读回并比对，然后删除文件。	顺利通过，日志显示 `[LAB7] test_filesystem_integrity passed`。
`test_concurrent_access`	2 个 worker、每个循环 2 次；创建 `test_<worker>_<iter>`，写入整型并立即删除，用来验证 inode/块回收与 `dirlink/dirlookup`。	完成后输出 `[LAB7] test_concurrent_access completed`。
`test_filesystem_performance`	写 2 个 4B 小文件 + 1 个 4MB 大文件，统计 `timer_get_ticks()`。	`[LAB7] Small files (2x4B): 1 ticks`；`[LAB7] Large file (4MB): 1 ticks`。
`debug_*` 调试函数	`debug_filesystem_state()` 打印超级块/空闲块/空闲 inode，`debug_inode_usage()` 遍历 `dinode`，`debug_disk_io()` 告知读写次数。	例：`Free blocks: 8145`、`Free inodes: 198`、`Disk reads: 28`, `Disk writes: 189`。

2. 崩溃与恢复策略

- `test_crash_recovery()` 目前仅给出流程提示：在事务执行期间强制杀掉 QEMU，再次启动时 `recover_from_log()` 会自动把已提交事务重放，未提交事务被忽略。
- 在调试中曾通过 `pkill -f qemu-system-riscv64` 模拟断电，验证日志能够保证 `testfile` 的一致性（重新启动后仍可创建并读取文件）。

3. 局限与后续验证计划

- 当前仅实现一级间接块，单文件最大约 4MB；若需要压力测试，可增大 `BSIZE` 或拓展到二级间接块。
- `LAB7_CONCURRENT_*` 为了避免缓存耗尽而调小，后续打算在引入更大缓存后恢复到 4 worker × 100 iter，以覆盖更多并发场景。
- `debug_inode_usage()` 通过直接读取磁盘 inode，而非加锁 `icache`，避免测试阶段阻塞；未来可扩展为仅打印非空目录，降低输出噪音。

四、结论与展望

- 文件系统从块设备、日志、inode/目录到自测流程均可一次性启动并通过测试， Lab7 的功能闭环已经形成。
- 现有实现强调清晰度和调试性：日志与调试命令可复用到后续实验，`mkfs.py`/`lab7_*.c` 也便于扩展。
- 下一步计划：增加更大的块缓存与 `sleep/wakeup`，在文件系统操作中减少忙等；扩展 `inode_addrs` 支持双重间接块；为日志加入大小自适应策略，支持更长事务。

五、思考题与解答

1. **xv6 的简单文件系统有哪些优缺点？**
   优点是实现精炼、可教学、易调试；缺点是目录线性扫描、仅一级间接块导致扩展性差，缺乏权限/ACL

等特性。本实验通过添加调试输出和 Python mkfs 在不牺牲简单性的前提下提升可维护性。

2. **写前日志如何确保一致性？恢复时再次崩溃会怎样？**
   `log_write()` 只把块号加入头部并 pin 缓存，`commit()` 先把真实块复制到日志，再写日志头，最后把日志块拷回主区域并清空头部；恢复时读取日志头、重放并清空。因为每次重放都是幂等复制，即使恢复过程中再次崩溃，只要日志头未清零，就会在下一次启动时继续复制，保持一致性。

3. **目前性能瓶颈在哪里？**
   主要在块缓存容量有限和目录线性扫描。并发测试容易把 32 块缓存占满导致阻塞，目录操作仍要全量扫描 14 字节定长项。可通过扩大缓存、加入哈希式目录缓存或 extent/范围锁来提升性能。

4. **如何支持更大的文件与更大的文件系统？**
   可以拓展到多级间接或 extent-based 分配，并把 `BSIZE` 从 1 KB 调大到 4 KB 以上，同时让 `superblock` 记录 64 位块号；`mkfs.py` 也要同步调整布局计算，避免位图过大。必要时可引入双写或 copy-on-write 机制来保证日志空间充足。

5. **如何检测并修复文件系统损坏，并在在线状态下执行检查？**
   离线可通过 `fsck` 风格的扫描：校验超级块、inode 引用计数与位图一致性，并尝试回收孤儿 inode；在线可以周期性执行 scrub 线程，利用日志记录和块校验和比对来检测 silent data corruption，配合冗余或快照实现不停机修复。

六、问题与总结

1. 典型问题

问题	现象	原因	解决	预防
`virtio_disk_rw` 不返回	QEMU 卡住无日志	默认 legacy 模式，驱动写 modern 寄存器	在命令行加 `force-legacy=false` 并仅保留 modern 流程	每次升级 QEMU 前验证设备模式
`panic: ilock: no type`	mount 时崩溃	mkfs 镜像未写根 inode	重写 `tools/mkfs.py`，创建 `./..` 和位图	生成镜像后用 `hexdump` 检查超级块/根目录
`bget: no buffers`	并发测试 early panic	`NBUF=32` 不足，所有 buf 被 pin	降低 `LAB7_CONCURRENT_*` 默认值，提示需要时增大 `NBUF`	在 README 写明如何调参，后续扩容缓存
目录删除后 `nlink` 错乱	再次访问目录报错	删除目录时未更新父目录链接数	`lab7_unlink` 调整：dir 空检查 + 更新父 `nlink`	写目录操作统一走 `dirlink/dirunlink`，保持对称
日志溢出	`panic: log_write: too big`	多事务嵌套，`outstanding` 超限	`begin_op()` 在日志空间不足时睡眠，`LAB7` 测试串行执行	增加日志容量或约束应用行为

2. 实验收获

- 从块层到日志层再到 inode/目录，完整体验了一个简化文件系统的构建路径；
- 通过 `debug_filesystem_state/usage` 等辅助函数，学会如何快速定位磁盘布局或 inode 泄露问题；
- 理解写前日志在崩溃恢复中的作用，并通过 `recover_from_log()` 验证幂等性；
- 学会使用 Python `mkfs.py` 快速生成镜像并在调试时重置环境。

3. 改进方向

- 扩展到双重间接块或 extent 分配，以支持更大的文件；
- 为目录扫描加入缓存/哈希，降低 `namex()` 的线性成本；
- 增大块缓存并在 `bget()` 中加入更细粒度 pin/锁，适配更多并发测试；
- 规划在线 fsck/校验和机制，增强运行时一致性保障。