03 综合实践：构建极简 Ext2 架构模拟

1. 本节导读

在之前的实验中，我们学习了如何观察文件系统以及如何调用 API 来克隆目录。但你是否思考过：当你执行 mkdir 或 write 时，磁盘上的二进制数据到底发生了什么变化？

本实验将带你进入“内核开发者”的视角。我们将在内存中开辟一块 4MB 的空间作为“虚拟磁盘”，从零开始构建一个符合 Ext2 架构 的极简文件系统。你将亲自设计超级块、位图和 Inode，并手动管理数据块的分配。通过模拟 `mkdir`, `touch`, `cd`, `rm` 等核心指令，你将理解文件系统是如何在杂乱的二进制数据中建立起秩序的。

---

2. 课前基础知识储备：虚拟磁盘布局

在真实世界中，文件系统管理的是扇区。我们将 4MB 内存数组模拟为一块磁盘。为了管理它，我们需要将其划分为不同的功能区。

2.1 磁盘布局

区域名称	长度/结构	功能描述
超级块 (Superblock)	结构体	记录块大小、总块数、空闲块/Inode 总数。
组描述符 (GDT)	结构体	记录位图区、Inode 区、数据区的起始地址。
块位图 (Block Bitmap)	一维数组	标记数据块占用（0-空闲，1-占用）。
Inode 位图 (Inode Bitmap)	一维数组	标记 Inode Table 条目占用。
Inode 表 (Inode Table)	二维数组	存放所有文件的元数据（Inode 数组）。
数据区 (Data Blocks)	块集合	存放文件内容或目录条目。

【关键信息解读】

- 管理开销：文件系统需要“为了管理数据而牺牲空间”。例如，即使磁盘是空的，超级块和位图也会预先占用空间。
- 位图的效率：位图通过 1 位（bit）映射 1 个块（1KB），极大地节省了寻找空闲空间时的搜索开销。

【观察与思考】

- 如果我们将块大小从 1KB 增加到 4KB，位图的大小会发生什么变化？对管理超大文件和大量小文件分别有什么优劣？

2.2 核心数据结构

A. Inode（索引节点）

每个文件/目录由一个 Inode 唯一标识，包含以下属性：

- 文件名：固定长度字符串。

- 文件类型：目录 或 数据文件。
- 文件权限：w/r/x。
- 文件大小：字节为单位。
- 修改时间：程序运行起的时钟滴答数（`float`）。
- 首块指针：指向该文件在数据区中的第一个数据块编号。

B. 链式数据块结构

文件内容采用**隐式链接**存储。

- 指针机制：每个数据块（如 512 字节）的前 **2 字节**作为指针，存储下一块的编号。
- 结束标记：若指针值为 `-1`，则表示该块为文件的末尾块。
- 有效载荷：每块实际可用的存储空间为 `BlockSize - 2` 字节。

【关键信息解读】

- 链式存储的代价：由于每个块的前 2 字节被挪作他用，文件系统产生了“内部碎片”。一个 1024 字节的文件在我们的系统中必须占用 2 个块（第一个块只能存 1022 字节）。
- 随机访问瓶颈：这种设计下，如果你想读取文件的第 100 块，必须从第 1 块开始顺着链条读取前 99 块。这与 Ext4 使用的 Extents 索引有本质区别。

C. 目录文件结构

目录本质上也是一种文件。

- 存储内容：目录的数据块中存储其包含的子文件的 Inode 编号。
- 条目格式：每 2 字节记录一个子 Inode。

---

3. 实验任务

步骤1：运行流控制与持久化

- 初始化 (`enter`)：系统启动。若存在上次保存的镜像文件，则加载镜像并恢复到上次退出的目录状态；否则初始化一个空的根目录 `/`。
- 持久化 (`exit`)：将整段内存空间及当前工作目录状态写入宿主机文件。
- 退出 (`q`)：结束模拟程序。

【观察与思考】

- 如果程序没有执行 exit 就意外崩溃，你之前创建的文件会消失吗？这在现实中对应什么现象？

步骤2：模拟指令集

系统需提供命令行交互界面，支持以下操作：

- `ls`：列出当前目录下所有条目的名称、类型、权限、大小及修改时间。
- `cd <dir>`：切换当前工作目录（需支持 `.` 和 `..`）。
- `mkdir <name>`：创建新目录，分配 Inode 及一个初始数据块。
- `rmdir <name>`：删除空目录，释放对应的 Inode 和数据块位图。
- `touch <name>`：

    - 若文件不存在：创建新数据文件，分配 Inode，不分配数据块。
    - 若文件存在：仅更新其最后修改时间。
- `touch <size> <name>`：
    - 若文件不存在：创建新文件，并根据 `size` 大小通过分配算法申请连续/不连续的数据块。
    - 若文件存在：修改文件大小，并重新调整其占用的数据块链。
- `rm <name>`：删除数据文件，顺着块链表回收所有数据块位图及 Inode 位图。

【关键信息解读】

- 元数据先行：在创建文件时，通常先写数据块，最后修改位图和 Inode；而在删除时，先改位图。这种顺序是为了在系统崩溃时尽量保持一致性。
- `rm` 与 `rmdir` 的差异：`rmdir` 必须检查目录块中除了 `.` 和 `..` 之外是否还有其他 Inode 编号。

【观察与思考】

- 在执行 touch 5000 test.txt 分配了 5 个块后，如果分配到第 3 个块时发现磁盘满了，你的程序应该如何处理？是回滚之前分配的 2 个块，还是保留残缺的文件？

步骤3：核心算法设计

- 空间分配算法：遍历块位图，寻找值为 0 的索引。在链式分配中，每申请一个新块，需修改前一个块头部的 2 字节指针。
- 回收算法：删除文件时，必须深度遍历其数据块链表，将位图中对应的位逐一清零。
- 路径与目录管理
    - 查找逻辑：在当前工作目录对应的数据块中，遍历读取 Inode 编号，通过 Inode Table 检索对应的文件名。
    - 层级维护：在创建目录时，自动在数据块内写入 `.`（当前目录 Inode）和 `..`（父目录 Inode）。

【关键信息解读】

- 当你执行 touch 创建新文件时，文件系统需要完成两件事：
    - 分配 Inode：在 Inode Bitmap 中找到第一个为 0 的位。
    - 分配数据块：在 Data Bitmap 中找到第一个为 0 的位。
- 目录的本质是“特殊的文件”。在 Ext2 中，根目录 / 默认占用第 0 号 Inode。目录的内容是一个 struct entry 数组。

【观察与思考】

- 为什么在我们的设计中，文件名是存在 Inode 结构体里的，而 Linux 却是存在目录项（Dentry）里的？如果文件名存放在 Inode 里，实现“硬链接”会遇到什么困难？

---

4. 实验总结与后续预告

通过本次综合实践，你完成了一次从“0”到“1”的系统构建：

- 理解物理布局：明白了位图、超级块、数据区是如何在空间上共存的。
- 掌握链式存储：亲自处理了块指针的串联与回收。
- 文件操作闭环：通过模拟 Shell 实现了从创建到销毁的完整生命周期。

实验报告要求提交 fs_sim.c 源码，报告中须描述代码设计思路，并且具有完整的测试流程，同时诚实的记录你在实验中遇到的问题和解决方案。