同济大学

TONGJI UNIVERSITY

操作系统课程设计

课题名称	操作系统课程设计
副 标 题	简单二级文件系统
学　　院	电子与信息工程学院
专　　业	计算机科学与技术
学生姓名	郑星云
学　　号	2151610
指导老师	方钰
日　　期	2024 年 03 月 24 日

目录

装 订 线

装 订 线

1 需求分析

1.1 程序功能概述

本项目实现了一个类 UNIX 的二级文件系统，通过一个模拟磁盘的大型文件（例如 disk.img，亦称作一级文件）来模拟 UNIX 系统的磁盘。此磁盘按照 512 字节的块存储信息。用户能够通过命令行与文件系统进行交互，完成如创建、删除和读写文件等操作。

1.2 支持的功能及特性

1. **文件系统的初始化**：加载文件系统的元数据，包括超级块、inode 位图、块位图、inode 表、数据块区等。若文件系统不存在，则初始化一个新的文件系统。

2. **基本读写操作**：通过逻辑地址对磁盘执行基础的读写操作。

3. **高速缓存**：实现了内存中的高速缓存，具备以下特点：

3.1 以 LRU（最近最少使用）策略管理缓存：在需要淘汰缓存时，选择最长时间未被使用的缓存。

3.2 尽量复用缓存：如果数据块已在缓存中，则无需重新进行磁盘读写。

3.3 当复用不可行时，通过一级文件系统接口完成对应 512 字节数据块的读写操作。

3.4 实现延迟写机制：数据不直接写入磁盘，而是先写入缓存。

4. **目录管理**：支持创建和删除目录。

5. **文件操作**：支持文件的创建、删除、读写。

6. **命令行接口**：提供了一套命令行工具，允许用户通过命令行与文件系统交互。可用命令如下：

命令	简介	用法
cat	读取文件的内容	cat <file_name>
cd	更改当前目录	cd <dir>
download	从文件系统下载真实文件	download <path_in_system> <real_file_path>
echo	将消息打印到控制台	echo <message> [count]
exit	退出 shell	exit
fclose	关闭文件	fclose <file_id>
flist	列出所有打开的文件	flist
fopen	打开一个文件	fopen <file_name>
format	格式化磁盘	format
fseek	移动文件指针	fseek <file_id> <offset>
fwrite	多次将内容写入文件	fwrite <file_id> <data> [times]
help	显示可用命令	help

装 订 线

init	初始化文件系统	init
ls	列出目录内容	ls
mkdir	创建一个新目录	mkdir <dir_name>
pwd	打印工作目录	pwd
rm	删除文件或目录	rm <file_name>
save	将文件系统立刻保存到磁盘	save
touch	创建一个新文件	touch <file_name>
upload	上传一个真实文件到文件系统	upload <path_in_system> <real_file_path>

1.3 输入输出形式

用户通过命令行输入指令，程序解析后执行对应操作，并将结果输出至命令行。

装 订 线

2 概要设计

2.1 任务分解

该项目主要分解为三个任务：

1. 命令行界面(Shell)：提供用户接口，用户可以通过命令行输入命令，实现对文件的操作。

2. 磁盘管理器(DiskManger)：提供一个一级文件系统的访问接口，能够按块从磁盘中读写数据。

3. 文件系统(FileSystem)：提供一个二级文件系统的访问接口，负责对镜像文件的存储空间进行管理，包括分配 Inode 空间、分配数据块空间、文件的读写等操作。

文件系统中可以再细分为：

1. 高速缓存管理：使用 LRU 算法维护缓存队列。

2. 位示图：使用位示图维护磁盘上数据块的使用情况。

3. 文件混合索引树：实现文件数据块的分配和索引。

4. 打开文件表：维护打开文件的信息，包括文件的读写指针、文件的 Inode 等。

2.2 整体结构

文件系统总大小为约为 1G,（总共 2098161 个扇区，每个扇区 512 字节），整体分为三个部分：超级块(SuperBlock)、Inode 区、数据区。

A. 超级块(SuperBlock)

起始地址位于第 0 号扇区，总计 513 个扇区。其中包含 DiskInode 的数量、数据块的数量、脏标记、以及 Inode 位示图和数据块位示图。其中 Inode 位示图(IB)共 496 字节，数据块位示图(DB)共 262144 字节。

B. Inode 区

起始地址位于第 513 号扇区，总计 496 个扇区，每个扇区可以存储 8 个 Inode，共有 3968 个磁盘 Inode。

C. 数据区

起始地址位于第 1009 号扇区，总计 2097152 个扇区。

<table>
<tr><td colspan="3">SuperBlock</td><td rowspan="2">Inode</td><td rowspan="2">Data</td></tr>
<tr><td>基本信息</td><td>IB</td><td>DB</td></tr>
<tr><td colspan="2">0#</td><td>1#~512#</td><td>513#~1008#</td><td>1009#~2098160#</td></tr>
</table>

图 2.1 整体结构示意图

2.3 数据结构定义

2.3.1 高速缓存块(BufferCache)

内存中的高速缓存块，用于加速对磁盘的读写操作。当用户需要读取磁盘上的数据时，首先会在高速缓存中查找，如果找到了，则直接返回；如果没有找到，则需要从磁盘中读取数据，并将数据写入高速缓存中。

当高速缓存块需要被换出或系统关闭时，根据脏标志判断是否需要将将缓存块中的数据写回到磁盘中。

属性名	类型	描述
block_no	uint32_t	缓存块的对应的物理块号
dirty	bool	缓存块是否被修改
data	char[BLOCK_SIZE]	缓存块的数据

2.3.2 高速缓存队列与映射表

高速缓存队列用于维护一系列指向高速缓存块的指针，使用 LRU 算法进行管理。每当有数据被访问，就将对应的缓存块移动到队列的尾部，当需要换出缓存块时，将队列头部的缓存块换出。

为了快速检索缓存块，使用一个映射表，将缓存块的物理块号映射到高速缓存队列的迭代器。

综合使用高速缓存队列和映射表，可以实现高速缓存的快速查找、插入、删除等操作。

2.3.3 位示图(Bitmap)

位示图用于记录磁盘上数据块的使用情况，每个位表示一个数据块的使用情况，0 表示空闲，1 表示已被使用。

相比于 UNIX V6++ 中使用的栈式管理方法，位示图具有占用磁盘空间小，访问速度快，实现方法简单的优点，并且可以更方便的分配连续内存空间。

2.3.4 DiskInode

磁盘 Inode，用于记录文件的元数据信息，包括文件的大小、文件的数据块指针、文件的类型等。每一个磁盘 Inode 共计 64 字节。

属性名	类型	描述
file_size	uint32_t	文件大小
file_type	uint32_t	文件类型
block_pointers	uint32_t[10]	文件数据块指针
padding	uint32_t[4]	填充字节

2.3.5 Inode

内存 Inode。和磁盘 Inode 相比，内存 Inode 多了引用计数和 Inode 编号两个字段。引用计数用于记录打开文件的数量，当引用计数为 0 时，可以将 Inode 写回磁盘。Inode 编号用于标识 Inode 在 Inode 区中的位置。

属性名	类型	描述
file_size	uint32_t	文件大小
file_type	uint32_t	文件类型
block_pointers	uint32_t[4]	文件数据块指针
reference_count	uint32_t	引用计数，记录使用该 Inode 的文件数量
inode_no	uint32_t	Inode 编号
padding	uint32_t[2]	填充字节

2.3.6 数据块索引结构

采用三级索引的方式来管理文件的数据块，包括：5 个直接索引指针、2 个一次间接索引指针、2 个二次间接索引指针、1 个三次间接索引指针。如图 2.2 所示。三级间接索引最大可以支持存储 $(5 + 2 \times 128 + 2 \times 128 \times 128 + 128 \times 128 \times 128) \times 512B \approx 1.0157GB$ 的数据。详细结构如 图 2 所示。

装 订 线

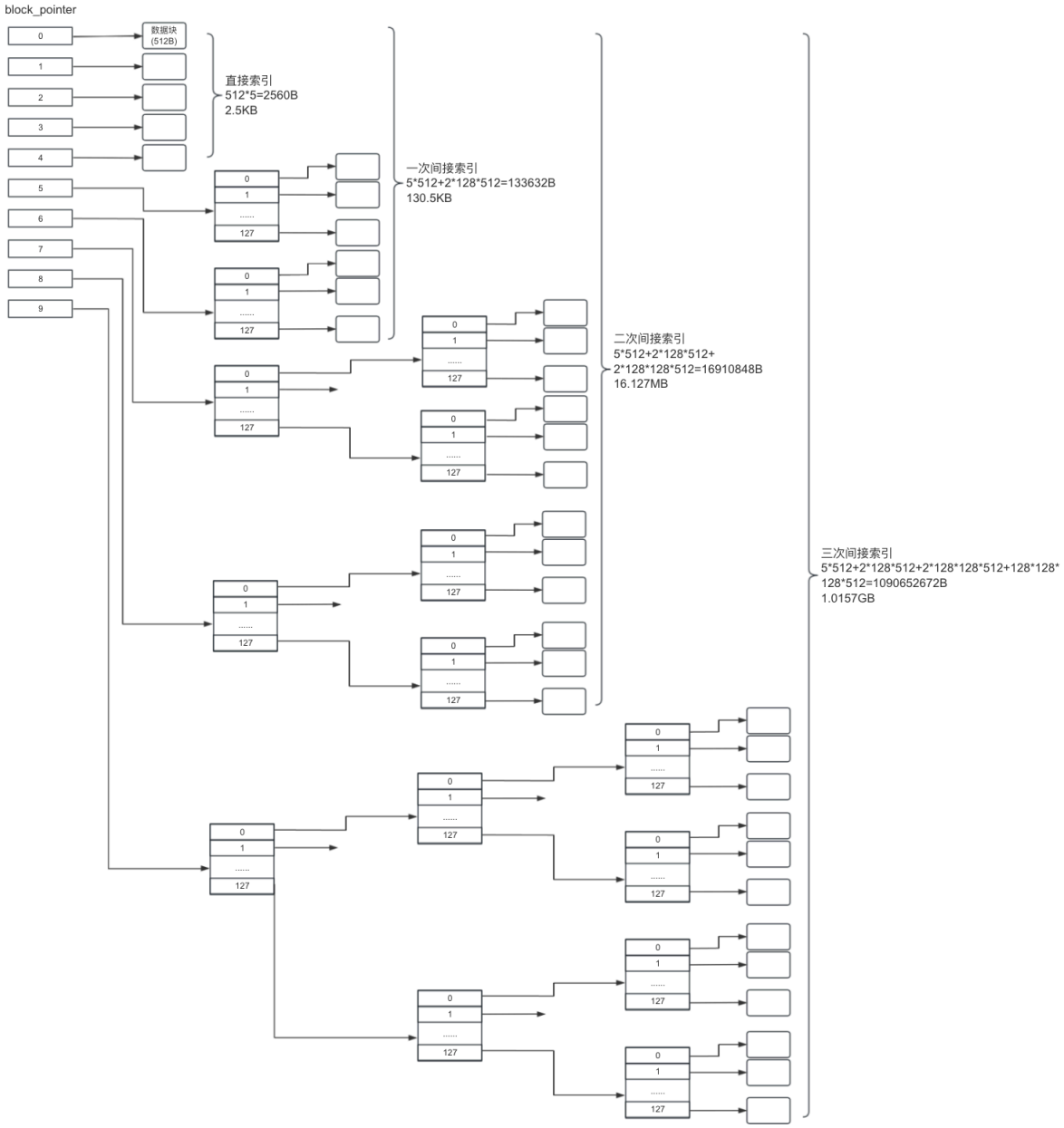


图 2.2　数据块索引结构示意图

2.3.7　文件(File)

文件是对打开文件的抽象，包含了文件的读写指针、文件的 Inode 指针、文件名等信息。当对文件进行读写操作时，文件的读写指针会随之移动。每一个打开的文件都会对应唯一一个文件结构，当文件被关闭时，文件结构会被释放。

属性名	类型	描述
inode	Inode*	指向分配给这个内存的 Inode 指针

offset	uint32_t	文件读写指针
reference_count	uint32_t	引用计数，记录使用该文件的进程数量
file_name	char[28]	文件名

2.3.8 打开文件表(OpenFileTable)

打开文件表用于维护打开文件的信息。当用户打开一个文件时，会在打开文件表中分配一个空间给这个文件。打开文件表的大小是固定的，最多同时能打开 16 个文件，当打开文件表已满时，无法再打开新的文件。已经存在于打开文件表中的文件无法被重复打开。

2.3.9 目录项(DirectoryEntry)

目录项用于抽象目录中的文件信息，包括文件名和文件的 Inode 编号。目录项的大小为 32 字节。一个目录文件的数据块最多可以存放 16 个目录项。

属性名	类型	描述
inode_id	uint32_t	文件的 Inode 编号
file_name	char[28]	文件名

装 订 线

3 详细设计

3.1 位示图空闲空间管理

该文件系统中，Inode 区和数据区的空间管理均通过位示图来实现。位示图是一个二进制数组，每一个位对应一个数据块或 Inode 节点。位示图的每一个位表示对应的数据块或 Inode 节点是否被分配。Inode 区和数据区的处理方式几乎相同，这里以数据区为例进行说明。

使用 c++的 bitset 容器来实现位示图。bitset 类是一个固定长度的二进制数组，可以通过下标迅速访问每一个位。

分配空闲空间时，遍历位示图，找到第一个为 0 的位，将其置为 1，返回该位的下标。释放空间时，将对应的位置为 0。

如果每次分配空间都需要从头开始遍历位示图，效率较低。采取循环遍历的方式，每次从上次分配的位置开始遍历，直到找到一个空闲位。

3.2 多级索引结构

每一个 Inode 节点都有 10 个数据块指针。在未分配任何数据块时，这 10 个指针都指向 0。将文件的数据块进行编号，第 1~5 号数据块存在前 5 个指针中，第 6~133 号数据块存在第 6 个指针指向的一级索引块中，以此类推。保证文件的数据块号是连续的，不存在分配的数据块之间有空隙，即分配第 i 个数据块时，前 i-1 个数据块都已经分配。

3.2.1 创建索引节点

根据文件大小，可以计算出当前已经分配的数据块数目(file_size /512)。因为保证分配连续，所以可以得到即将分配的数据块编号。完整流程如下：

1. **计算新块编号**：基于文件当前大小，计算出新数据块的编号（ new_block_num)。
2. **直接指针情况**：
   - 如果新块编号小于 5，使用直接指针。将新块分配给对应的直接指针位置。
3. **一级间接指针情况**：
   - 如果新块编号在 5 到 5+PTRS_PER_BLOCK*2 的范围内，使用一级间接指针。
   - 计算一级和二级索引的位置。
   - 如果是一级索引的第一个块，分配并初始化一级索引块。
   - 在一级索引块中为新数据块分配位置。
4. **二级间接指针情况**：
   - 如果新块编号在一定范围内，使用二级间接指针。
   - 计算一级、二级和三级索引的位置。

- 如果是二级索引的第一个块，分配并初始化二级索引块。
- 如果是三级索引的第一个块，分配并初始化三级索引块。
- 在三级索引块中为新数据块分配位置。

5. **三级间接指针情况：**
   - 如果新块编号还更大，尝试使用三级间接指针。
   - 计算一级、二级、三级和四级索引的位置。
   - 如果是一级索引的第一个块，分配并初始化一级索引块。
   - 如果是二级索引的第一个块，分配并初始化二级索引块。
   - 如果是三级索引的第一个块，分配并初始化三级索引块。
   - 在三级索引块中为新数据块分配位置。
6. **文件过大错误：**如果新块编号超出了所有指针情况的处理范围，抛出“文件过大”的错误。

每次分配数据块时，会将数据块中原有的数据清空，防止磁盘中的垃圾数据影响文件内容。

3.2.2 查询索引节点

查询索引节点的过程与创建索引节点的过程类似，只是不需要分配新的数据块。输入数据块编号，根据数据块编号找到对应的数据块。

查询间接索引时，会将索引块移入高速缓存队列末尾。如果重复查询相同的间接索引块，可以直接从高速缓存队列中取出，减少磁盘 I/O 次数。

3.2.3 删除索引节点

删除索引节点只在删除文件时使用。删除文件时，需要释放所有的数据块。删除索引节点时从头遍历多级索引树中的所有指针，并不修改数据块的真是内容，只是将对应的位示图位置为 0。

3.3 文件高速缓存

文件系统中的高速缓存采用 LRU（Least Recently Used）算法。LRU 算法是一种基于时间局部性的缓存替换算法，将最近最少使用的数据块替换出缓存。

Inode 和数据块采用同样的方法管理缓存，只是缓存的数据和大小不同。接下来以数据块为例进行说明。

3.3.1 高速缓存块

高速缓存块是一个结构体，包含数据块的内容、数据块编号、是否被修改等信息。高速缓存块的大小与数据块的大小相同，均为 512B。

在文件系统创建时，就在内存空间中分配一定数量的高速缓存块。这些缓存快以数组的形式组织，通常情况下不会直接访问这个数组，而是通过缓存队列中的指针来访问。

3.3.2 缓存队列

缓存队列是一个双向链表，每一个节点对应一个高速缓存指针(BufferCache*)。缓存队列的队首是最近最少使用的数据块，队尾是最近使用的数据块。每次访问一个数据块时，将该数据块移动到队尾。

缓存队列的大小是固定的，当缓存队列已满时，需要替换一个数据块。替换时，将队首的数据块移出缓存队列，将新的数据块插入队尾。

双向链表的插入和删除操作都是 O(1)的，所以 LRU 算法的时间复杂度是 O(1)。

3.3.3 映射表

每次分配数据块时，会先在缓存队列中检查是否已经存在该数据块。如果存在，直接返回该数据块的指针。如果不存在，需要从磁盘中读取数据块。

为了快速查找数据块是否在缓存队列中，使用一个映射表。映射表是一个哈希表，将数据块编号映射到缓存队列中的迭代器。这样可以在 O(1)的时间内查找数据块是否在缓存队列中。

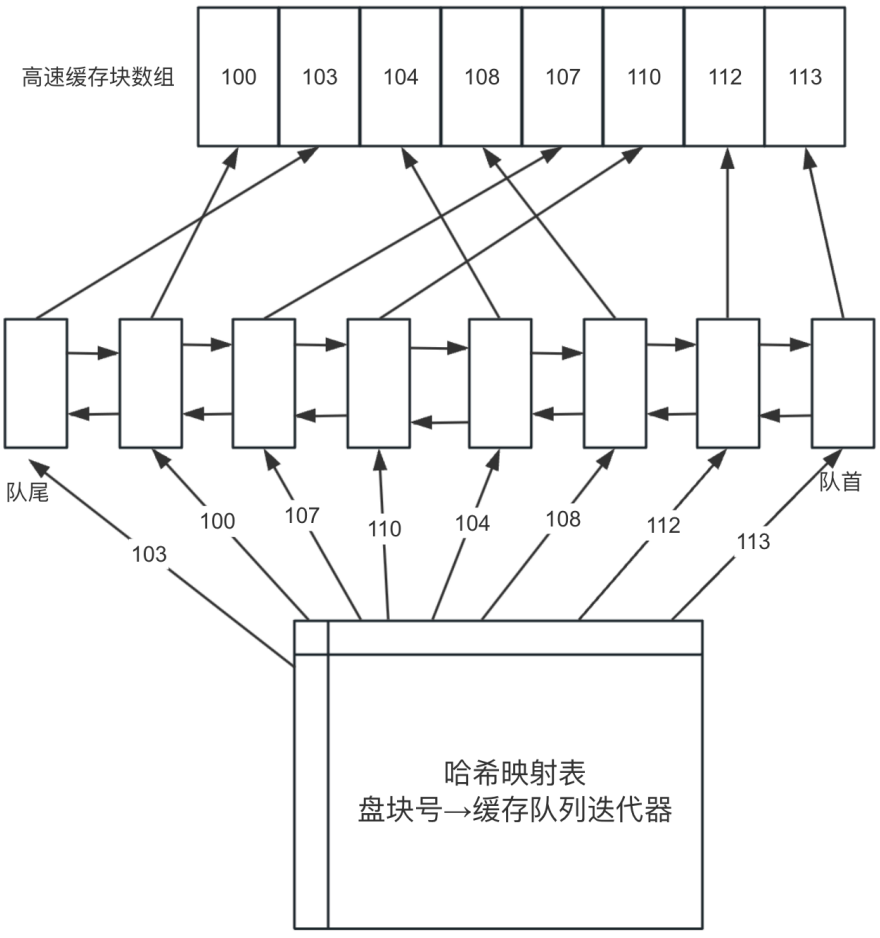


图 3.1 高速缓存结构

装 订 线

3.3.4 分配高速缓存

在哈希映射表中查找是否已经分配过该数据块。

- 如果已经分配过，直接返回该数据块的指针，将该数据块移动到缓存队列队尾。
- 如果没有分配过，检查是否还有空闲的高速缓存块。
  - 如果有，分配一个高速缓存块，将数据块读入缓存块中。
  - 如果没有，换出缓存队列队首的数据块，将新的数据块读入缓存块中。

3.3.5 延迟写

当数据写入高速缓存后，并不会立刻写回内存。每次写入高速缓存时，会将该缓存块的脏位置1，表示该数据块已经被修改过。当该数据块被替换时时，如果为脏，才将该数据块写回磁盘。另外，如果一次写入写到了缓存块底部，会立刻写回磁盘。

延迟写优点是，即减少了磁盘 I/O 次数，又保证了数据的一致性。

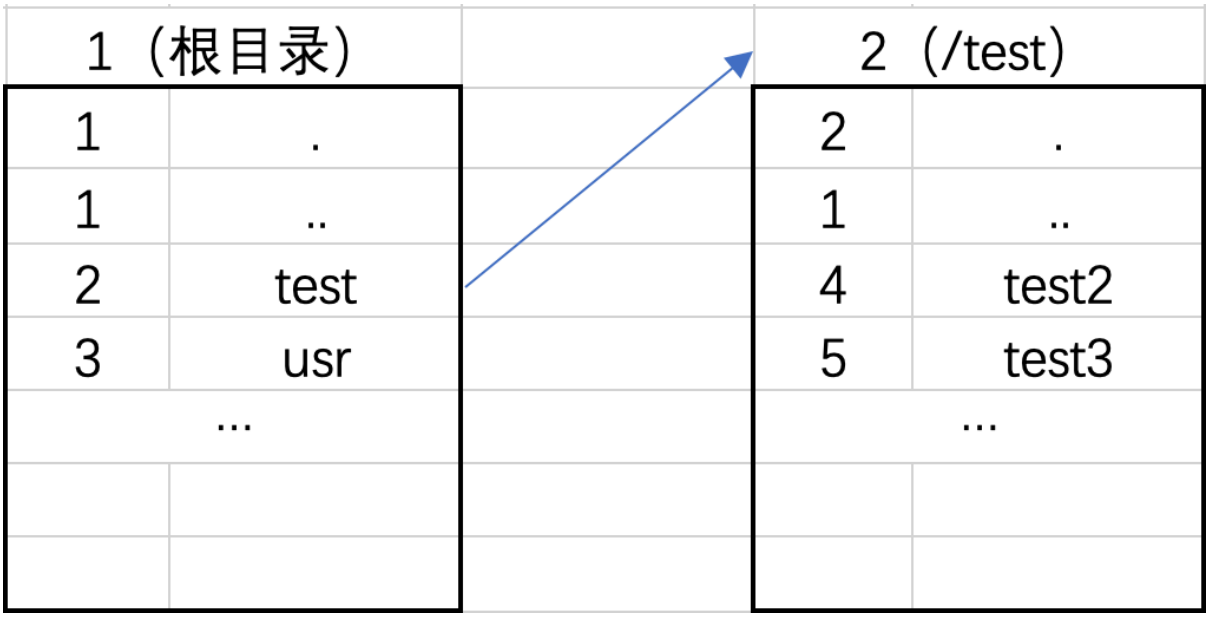
3.4 目录结构

文件分为普通文件和目录文件。目录文件是一种特殊的文件，其数据块中存储文件名和文件的 Inode 号。

一个数据块可以分配 16 个目录项。每个目录项包含文件名和文件的 Inode 号。文件名最长为 28 个字符，Inode 号占 4 个字节。

每一个目录的前两个目录项分别是"."和".."。其中"."表示当前目录，".."表示父目录。根目录的父目录是根目录本身。

当文件系统初始化时，会创建根目录。根目录的 Inode 号为 1。第 0 号 Inode 节点不使用，作为保留节点。



装 订 线

图 3.2　目录结构

3.4.1　获取 Inode 结构

根据对应的 Inode 号，可以快速从磁盘中找到 Inode 结构。Inode 区中，每一个扇区存储 8 个 Inode 结构。所以第 i 个 Inode 结构的位置为 $i/8$ 号扇区的第 $i \bmod 8$ 个 Inode 结构。

3.4.2　目录检索

内存中始终存在一个当前 Inode 号。初始为根目录的 Inode 号(即 1 号)。

路径分为绝对路径和相对路径。绝对路径以根目录为起点，相对路径以当前目录为起点。

目录检索的过程是一个递归的过程。从当前当前目录或根目录开始，根据文件名查找对应的 Inode 号。如果是目录文件，递归查找，直到找到对应文件。

3.4.3　创建目录(mkdir)

1. **检查目录名长度**：如果提供的目录名长度超过 28 字节，抛出异常。
2. **检查目录是否已存在**：通过列出当前目录下的所有项 ls()，检查目标目录名是否已存在，如果存在，抛出异常。
3. **创建新的目录节点**(inode)：分配一个新的 inode 给新目录，并设置其类型为目录(DIRECTORY)，大小为两个目录项(DirectoryEntry)的大小。
4. **初始化新目录的内容**：
   - 在新目录的第一个数据块中，创建两个特殊的目录项：. (代表当前目录自身) 和 .. (代表父目录)。
5. **更新父目录**：
   - 检查父目录是否需要扩展 (即，当前的数据块是否已满)，如果是，分配一个新的数据块给父目录。
   - 在合适的位置添加新目录的目录项，并更新父目录的大小。

3.4.4　删除目录

删除目录的实现和 UNIX V6++的实现有所不同，并不是将删除的目录项 Inode 编号标记为 0，而是将该目录文件中的最后一个目录项移动到删除的目录项位置，然后将目录文件的大小减 1。

这样做的缺点是增加了一次无用的内存拷贝，但是在删除目录项时不需要修改目录文件的内容，只需要修改目录文件的大小，简化了实现。

3.5　文件读写实现

3.5.1　打开文件表

内存中存在一个打开文件表。每一个被打开的文件都会在打开文件表中有一个对应的表项。表项包含文件的 Inode 号、文件的读写位置、文件引用次数和文件名。

打开文件表是一个定长数组，数组的大小是 16。当打开文件表已满时，无法再打开新的文件。

当文件关闭时，会将对应的文件引用次数减 1。当文件引用次数为 0 时，释放对应的打开文件表项。

需要对文件进行读写操作时，需要传递文件打开编号。根据文件打开编号，获取当前文件的读写指针。

3.5.2 文件读取(fread)

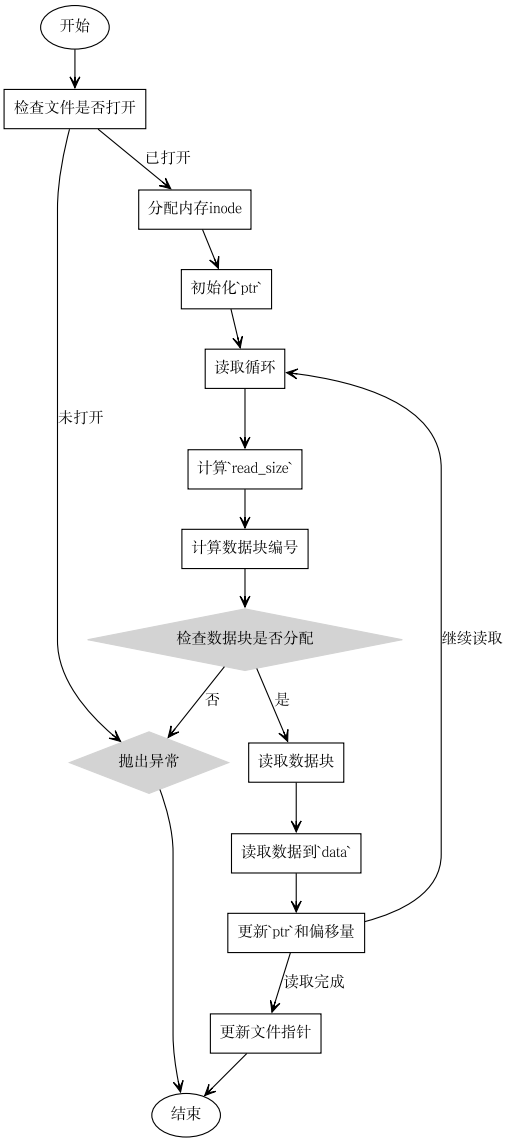


图 3.3 文件读取流程图

1. **检查文件是否打开**：通过 file_id 从 open_files 映射中获取文件。如果文件没有处于打开状态，抛出异常。
2. **分配内存 inode**：为打开的文件分配一个内存中的 inode 结构。
3. **初始化**：初始化一个 ptr 指针，为 offset 为文件的当前偏移量。
4. **读取循环**：
   - 使用 ptr 变量遍历文件，ptr 从 offset 开始，直到读取的数据大小达到请求的 size 或文件末尾。
   - 在循环中，计算需要从当前数据块读取的数据量 read_size，这取决于几个因素：块的剩余空间、请求的大小、以及文件的剩余大小。
   - 根据 ptr 计算当前的数据块编号 block_no，然后检查该数据块是否已分配。如果没有分配，抛出异常。
   - 读取数据块到缓冲区 buffer，然后从 buffer 中读取数据到目标 data 数组中。读取的数据量为 read_size，并根据 ptr 在数据块中的位置调整。
   - 更新 ptr 和 open_file 的 offset 以反映读取的数据量。
5. **更新文件指针**：循环结束后，open_file 的 offset 更新为最新的 ptr 值。

3.5.3 文件写入(fwrite)

1. **校验文件是否已打开**：
   - 通过 file_id 从 open_files 集合中获取文件的引用。
   - 如果文件未处于打开（繁忙）状态，则抛出异常。
2. **分配内存中的 inode**：
   - 为打开的文件分配一个内存中的 inode 节点。
3. **初始化偏移量**：
   - offset 变量设置为文件当前的偏移量。
   - 使用 ptr 变量遍历数据，从 offset 开始。
4. **写入循环**：
   - 循环条件是已写入的数据量（ptr - offset）小于需要写入的总数据量（size）。
   - 如果文件大小正好是数据块大小的倍数，意味着需要一个新的数据块，调用 alloc_new_block(inode)分配。
   - 计算当前偏移量所在的数据块编号 block_no。
   - 分配或获取对应的数据块缓冲区 buffer。
   - 计算此次循环可以写入的数据量 write_size，取决于当前数据块剩余空间和剩余需要写入的数据量。

- 使用 write_buffer 函数将数据从 data 数组写入到缓冲区 buffer 中，写入位置和大小由 ptr 和 write_size 决定。
- 更新 ptr 以反映已写入的数据量。

5. **更新 inode 和文件状态：**
   - 文件的大小更新为当前偏移量 ptr 和原文件大小的较大值，以反映新增加的数据。
   - 更新打开文件的偏移量 open_file.offset 为 ptr，以便下次写操作能从正确的位置继续。

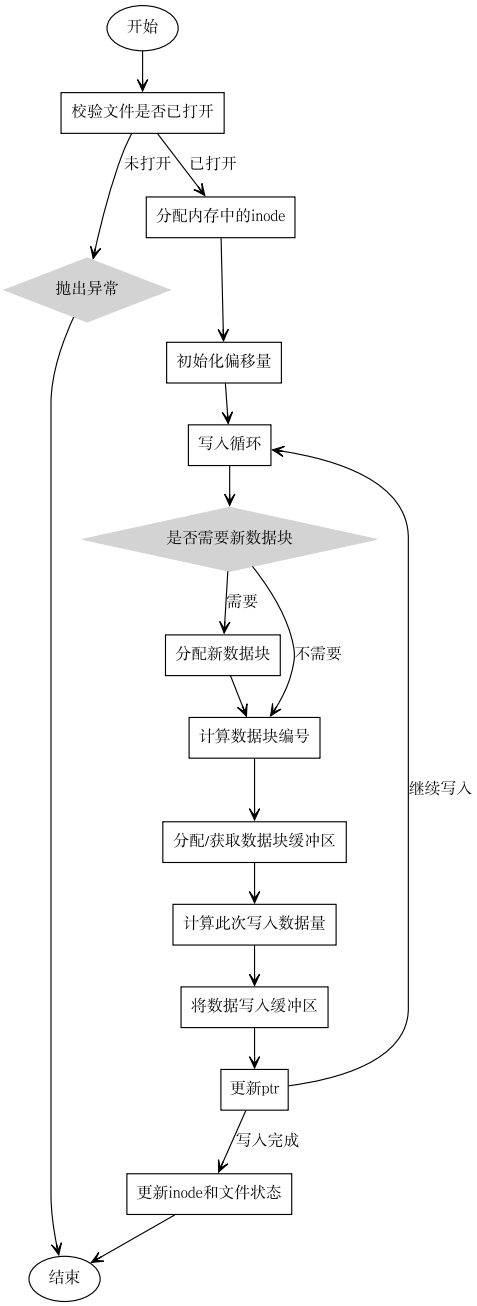


图 3.4　文件写入流程图

4 运行结果分析

4.1 测试任务 1

要求：

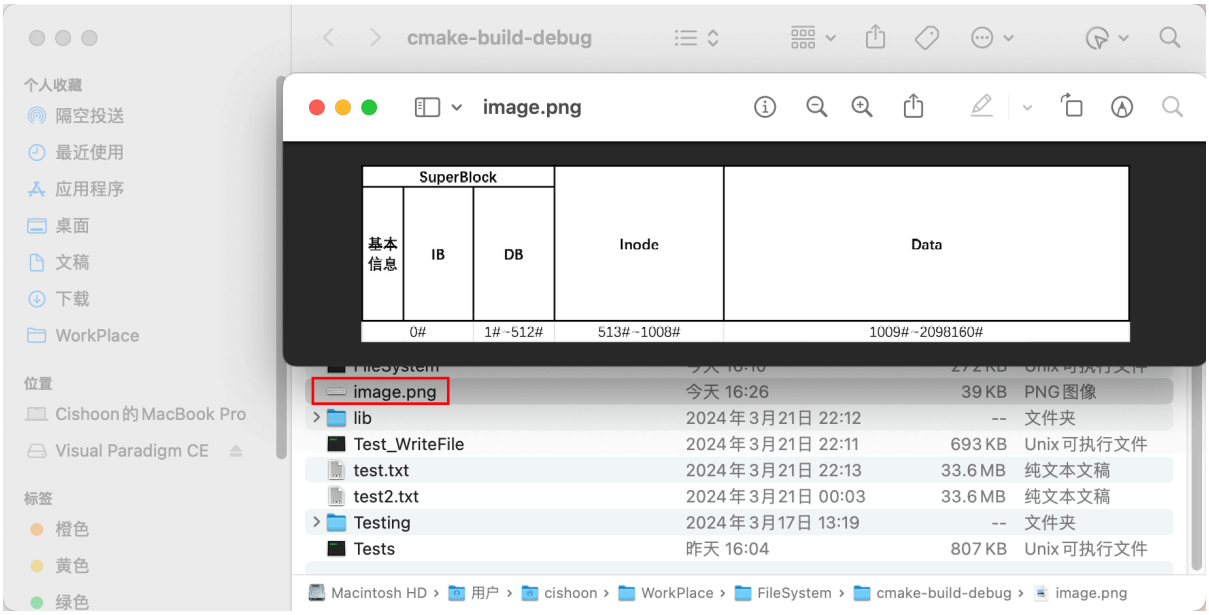
- 格式化文件卷
- 用 mkdir 命令创建子目录，建立如图所示的目录结构
- 把课设报告，关于课程设计报告的 README.md 和一张图片存放仅这个文件系统，分别放在/home/texts，/home/reports，/home/photos 文件夹下

```
[~]# format
Formatting disk...
Disk formatted.
[/]# mkdir bin
[/]# mkdir etc
[/]# mkdir home
[/]# mkdir dev
[/]# ls
.     ..    bin   etc   home  dev
[/]# cd home
[home]# mkdir texts
[home]# mkdir reports
[home]# mkdir photos
[home]# ls
.        ..       texts    reports  photos
```

```
[home]# cd texts
[texts]# upload README.md /Users/cishoon/WorkPlace/FileSystem/操作系统课程设计-简单文件系统.md
[================================================] 100% 6.21 KB / 6.21 KB
[texts]# cat README.md
## 需求分析

1. 实现对该逻辑磁盘的基本读写操作

    逻辑地址能够转换到物理地址

    每一个逻辑块512字节



2. 内存中有高速缓存，可以自己简化设计
```

```
[texts]# cd ..
[home]# cd reports
[reports]# ls
.   ..
[reports]# upload report.pdf /Users/cishoon/Documents/操作系统/课程设计/tongji-undergrad-thesis-typst-main/main.pdf
[================================================] 100% 959.59 KB / 959.59 KB
[reports]# ls
.           ..          report.pdf
[reports]# cd ../photos
[photos]# upload image.png /Users/cishoon/Documents/操作系统/课程设计/tongji-undergrad-thesis-typst-main/figures/整体设计.png
[================================================] 100% 37.70 KB / 37.70 KB
[photos]# ls
.           ..          image.png
```

可以把图片和报告下载到本地查看。



4.2 测试任务 2

要求：

- 新建文件 /test/Jerry，打开该文件，任意写入 800 个字节
- 将文件读写指针定位到 500 字节，读出 500 个字节到字符串 abc。
- 将 abc 写回文件。

```
[photos]# cd /
[/]# ls
.      ..     bin    etc    home   dev
[/]# mkdir test
[/]# cd test
[test]# touch Jerry
[test]# fopen Jerry
[0]
[test]# fwrite
Usage: fwrite <file_id> <data> [times]
[test]# fwrite 0 a 800
```

由于命令行中不能创建字符串变量，这个要求使用 gtest 测试代码来实现。

```
TEST(FileSystemTest, Test_test_Jerry) {
  FileSystem fs;
  fs.format();
  fs.touch("test");
  auto fd = fs.fopen("test");
  fs.fseek(fd, 0);
  std::string long_text(800, 'a');
  fs.fwrite(fd, long_text.c_str(), 800);
  fs.fseek(fd, 500);
  char buffer[500] {0};
  fs.fread(fd, buffer, 500);
  // buffer 此时的长度应该是 300
  EXPECT_EQ(strlen(buffer), 300);

  fs.fwrite(fd, buffer, strlen(buffer));
  // 总文件大小是 800 + 300
  fs.fseek(fd, 0);
  char buffer2[1101] {0};
  fs.fread(fd, buffer2, 1100);
  EXPECT_EQ(strlen(buffer2), 1100);
  fs.fclose(fd);
  SUCCEED();
}
```

运行结果如下，能够通过测试。

```
/Users/cishoon/WorkPlace/FileSystem/cmake-build-debug/Tests --gtest_filter=FileSystemTest.Test_test_Jerry:FileSystemTest/*.Test_test_Jerry:FileSystemTest
*.Test_test_Jerry/*:*/FileSystemTest.Test_test_Jerry/*:*/FileSystemTest/*.Test_test_Jerry --gtest_color=no
Testing started at 16:37 ...
Running main() from /Users/cishoon/WorkPlace/FileSystem/external/googletest/googletest/src/gtest_main.cc
Process finished with exit code 0
```

4.3 上传/下载大文件

该文件系统的数据区大小为 1GB，可以上传和下载大文件。这里上传了一个大小为 773.11MB 的 Clion 安装包，耗时约为 30 秒。下载和上传过程中，显示直观的进度条。

```
[~]# upload Clion.dmg /Users/cishoon/Downloads/CLion-2023.3.3-aarch64.dmg
[==================================================] 100% 773.11 MB / 773.11 MB
[~]# download Clion.dmg Clion.dmg
[==================================>                ] 70% 544.54 MB / 773.11 MB
```

4.4 复杂路径解析

文件系统可以通过 Inode 节点解析复杂的路径。

```
[/]# ls
.      ..     root   home   etc    bin    usr    dev
[/]# cd root/../home/.././etc/../bin/./.
[bin]#
```

4.5 测试文件夹创建与删除

连续创建 500 个文件夹，然后删除，再创建，最后检查是否都存在。

```
TEST(FileSystemTest, Test_rmdir_many) {
const int NUM = 500;
FileSystem fs;
fs.format();
for (int i = 1; i <= NUM; i++) {
    fs.mkdir("test" + std::to_string(i));
}
for (int i = 1; i <= NUM; i++) {
    fs.rm("test" + std::to_string(i));
}
for (int i = 1; i <= NUM; i++) {
    fs.mkdir("test" + std::to_string(i));
}
auto entries = fs.ls();
```

```
    // 检查是否有 test 目录
    for (int i = 1; i <= NUM; i++) {
        bool found = false;
        for (auto &entry : entries) {
            if (entry == "test" + std::to_string(i)) {
                found = true;
                break;
            }
        }
        if (!found) {
            FAIL();
        }
    }
    SUCCEED();
}
```

运行结果如下，能够通过测试。

```
/Users/cishoon/WorkPlace/FileSystem/cmake-build-debug/Tests --gtest_filter=FileSystemTest.Test_rmdir_many:FileSystemTest/*.Test_rmdir_many:FileSystemTest
 .Test_rmdir_many/*:*/FileSystemTest.Test_rmdir_many/*:*/FileSystemTest/*.Test_rmdir_many --gtest_color=no
Testing started at 16:49 ...
Running main() from /Users/cishoon/WorkPlace/FileSystem/external/googletest/googletest/src/gtest_main.cc
Process finished with exit code 0
```

装 订 线

5 用户使用说明

本项目使用 CMake 来构建和测试 FileSystem，用户需要按照以下说明操作：

5.1 前置要求

- 确保已安装 CMake（至少版本 3.20）和合适的 C++编译器。本项目使用 C++17 标准。
- 该项目针对 GNU C++和 Clang 编译器进行了特定的优化设置，也支持 MSVC 编译器。对于其他编译器，会显示一条消息提示可能未能正确设置优化标志。
- Google Test 框架已作为子目录集成，用于执行单元测试。
- 磁盘空间应不少于 1G。
- 本项目测试环境为 Linux/macOS，未在 Windows 环境下测试。

5.2 构建项目

1. 打开命令行或终端。
2. 导航到项目根目录。
3. 创建一个构建目录并进入：

```
mkdir build
cd build
```

4. 运行 CMake 来生成构建系统：

```
cmake ..
```

5. 构建项目：

```
make
```

这将会构建主项目 FileSystem 以及测试项目 Tests。

5.3 运行应用

- 在构建目录中，运行生成的 FileSystem 可执行文件来启动程序，命令为

```
./FileSystem
```

- 第一次成功启动后，输入 format ，会自动在当前文件夹创建一个 image.img 文件。
- 进入命令行界面，输入 help 可以查看所有支持命令的使用方法。

5.4 运行测试

- 为了运行所有测试，可以使用 CTest，一个 CMake 的测试驱动程序。在构建目录中，运行命令 ctest 来执行所有配置的测试。

- 也可以直接运行 Tests 可执行文件来进行单元测试。这些测试验证了文件系统的不同组件是否按预期工作。

装 订 线