|  |
| --- |
| Nullptr小组 |
| Minifs系统设计文档 |
|  |

|  |
| --- |
| Nullptr小组成员：刘硕，何祎君， 张嘉熙，张歆，彭青峰  2018-9-15 |

目录

[一、整体架构设计](#_Toc1816_WPSOffice_Level1) [2](#_Toc1816_WPSOffice_Level1)

[（一）索引](#_Toc5327_WPSOffice_Level2) [2](#_Toc5327_WPSOffice_Level2)

[（二）目录](#_Toc22389_WPSOffice_Level2) [3](#_Toc22389_WPSOffice_Level2)

[（三）数据内容](#_Toc14182_WPSOffice_Level2) [4](#_Toc14182_WPSOffice_Level2)

[二、数据结构设计](#_Toc5327_WPSOffice_Level1) [4](#_Toc5327_WPSOffice_Level1)

[（一） 索引](#_Toc2204_WPSOffice_Level2) [4](#_Toc2204_WPSOffice_Level2)

[（二） 目录](#_Toc18383_WPSOffice_Level2) [5](#_Toc18383_WPSOffice_Level2)

[三、算法设计](#_Toc22389_WPSOffice_Level1) [6](#_Toc22389_WPSOffice_Level1)

[（一）外存操作](#_Toc8164_WPSOffice_Level2) [6](#_Toc8164_WPSOffice_Level2)

# 一、整体架构设计

整个系统将由索引、目录、数据空间三部分组成。索引就是一个FAT表，用于存储地址信息；目录用来存储文件、文件夹信息以及文件之间的层级关系；数据空间则用来存放具体的数据。

### （一）索引

索引是文件的索引编号与文件在外存中存储位置（即分块编号）的一张对应表。

索引的本身结构是一个静态链表，存储上是使用的int型的数组，大小为262144\*4B，数组下标与数据内容分块中块的编号一一对应。由于文件的大小往往各不相同，所以有的文件需要多个块才能存完。所以我们选择通过静态链表的方式，链式存储文件数据块对应的所有地址。每个文件的第一个索引编号通过该文件对应目录信息获得，而后索引数组中每个数组元素的内容都对应存储当前文件的下一个分块的索引编号（即分块编号），若文件在当前分块已存储完毕，那么该在该处存入-1代表文件结束。例如一个文件在外存中存放在第1,4,7分块中，其中1,4分块都存满4KB，第10分块存了100B，那么索引数组index[1]=4,index[4]=7,index[7]=-1。

因为在索引在初始化时被赋为0，所以0即代表了索引所对应的数据块未被使用；因此0号索引不应该被开放以供使用（会与为使用的数据块产生冲突）。所以0号索引位置始终储存一个1，用来代表系统已被初始化。

由于索引和目录一共占用了9MB的空间，所以实际分块会少于原有的计算262144块，但为了方便，这部分空间不再进行拆分。

为了使查找过程方便，我们会在mount一个空间的时候就将索引全部读入内存中。所以使用数组存储索引相比于另一种使用多级索引的方式优点在于外存开销小，查询时间快很多，查询算法简单，缺点在于内存开销会相对较大。

### （二）目录

目录是存放文件的属性信息以及记录文件之间的层级结构关系的文件。目录在外存中集中存放，与文件内容分开。

目录本身在外存的结构也是一个静态链表，实现也是使用的自定义类型的数组，每个数组元素大小大约为32B左右，该数组存放两种类型的节点：一种是文件节点，其中要存储文件的首个索引编号，文件属性信息（如文件名，扩展名，创建时间等），还要保存每个文件或文件夹的parent信息；另一种是文件夹节点，其中保存文件夹的创建时间信息以及它的parent信息。parent信息就是一个int类型的数，表示该文件的父亲信息在目录数组中存放的位置编号，只有根目录上没有parent。

由于目录要经常进行修改，所以我们将目录固定存储在索引后的8MB的空间中，以方便于查找和修改，增加读取速度。和索引一样，在mount一个空间的时候，会将所有的目录信息加载到内存当中。

而且由于我们的结构时多层的，目录应以一颗多路树的结构加载到内存中，建树的根据是目录在外存中的数组中存放的parent信息。建好树后，大部分节点应该都有指向父亲和儿子的指针（根目录没有父亲指针，叶子节点没有儿子指针），树的所有叶子结点即为文件节点，非叶节点都是文件夹节点。输入指令时输入的都是文件名，通过文件名在目录里进行遍历查找得到文件的首个索引编号，再从外存中真正读取文件到内存。

**（三）数据内容**

数据内容块即为存放文件的部分。

因为文件可能会被删除的原因，所以无法进行连续的存储；所以我们决定以离散的方式存放文件，其中每个分块大小为4KB。然后对于一个文件，它的首地址将被存储于其目录信息中；然后它的剩余块编号都会由索引负责链式存储。

由于索引占据1MB，目录占据8MB的空间，所以最终可用的数据空间即为1015MB，也就是259840块。

**二、数据结构设计**

1. **索引**

索引是由一个int型的数据构成的，其中数组中的每一位代表它自己的编号，而里面则存储的是下一块数据的地址；

由于最多有(1<<30)/(1<<12)=(1<<18)个分块(实际小于这个)，所以地址信息最多只用到18位byte，用int存储是绰绰有余了。

在物理上，我们选择将索引放在1G空间的最开始的位置，占据1MB的空间。

1. **目录**

由与目录负责储存的是文件和文件夹的信息，所以根据系统的实际要求，需要文件名、扩展名、文件大小、修改日期和修改时间的信息；除此之外，为了确定文件所对应的存储位置，需要一定空间来存储文件的首地址；再由于我们有文件夹的设计，所以还需要存储文件或文件夹之间的父子信息，为了固定目录在外存中的存储大小，我们选择只存储其父亲节点编号信息，因为任何节点都会有且只有1个父亲（除去根目录），而可能有非常多的儿子信息。如果存储儿子信息则需要花费不定大小的储存空间，这对于目录在外存中的存储是有非常大的影响的。最后为了确定文件的种类以及处理已删除的文件信息，我们添加了一个标记为用于记录当前的文件状态。所以最终目录的数据结构设计如下：

char state;//文件状态

char name[8];//文件名

char extension[3];//扩展名

int location;//索引编号

int size;//文件大小

int parent;//父亲节点编号

int date;//修改日期

int time;//修改时间

共计1+8+3+4+4+4+4+4=32B的空间，刚好能整齐的存在外存中方便读取。

以上是目录信息在外存中的存储结构，当目录被读入内存后则需要建立一棵多叉树，以方便文件的操作。鉴于时间有限，所以我们没有选取更加优秀的B树来实现这种树形的数据结构，我们选择使用vector容器来实现一个变长数组来存放每个节点的状态，在每个节点中则选择使用list建立一个链表来实现儿子节点的信息的存储。

**三、算法设计**

**（一）外存操作**

首先，整个系统的所有操作都是建立在与外存的交互之上的，所以关于外存的操作则在整个系统中显得尤为重要。在设计上，我们选择先由底层FileOperator类完成全部的外存操作，然后再由上层的FileSystem整合调用，完成一系列的写入写出操作。

由于我们的索引和目录都是连续存储的，而且数据在储存过程中不需要被压缩，所以可以快速地通过一次操作就进行读入。而在处理数据块上，则通过每次寻找到空索引的方式确定存储位置，然后移动文件指针，写入4KB大小的数据（不足4KB则写对应大小的数据）。

文件在进行读写操作之前都必须被读入进内存中，这时候内存中就需要一个缓冲区来实现数据的存储。我们内存中的数据缓冲区是通过动态开辟来实现的，这样可以节约内存，上限为64MB；当需要读入的文件小于64MB时，则只开辟对应大小的内存，一次性全部读入；而当文件大于64MB时，则一次只读64MB，分多次进行读写操作。

除此之外，我们还对数据的写入进行了优化。在之前的算法中，系统需要对每一条索引、目录和数据块的修改进行一次外存的写入操作，如果当这些修改之间是彼此相距很远的话，这些修改还是有必要的，但是在实际使用过程中，这些修改往往都是连续的。这样的做法会大大增加写入次数，造成不必要的开销。所以我们决定对于每两条相邻的距离很近的修改做出合并：对于索引，合并距离不超过1024条的；对于目录，合并距离不超过128条的；对于数据块，合并彼此之间相邻的。以一个4MB的文件的读入为例，在优化前，需要进行1024次索引、1次目录、1024次数据块的修改；在优化后，则只需进行1次索引、1次目录、1次数据块的修改。这样的策略在系统空间比较充足的时候是可以大大提升写入效率的，即使在空间不太充足需要利用碎片空间的时候也能对局部的写入提升效率。