**实验报告**

## 一、结构设置

1. **总体结构**

总体结构采用VSFS结构，即SuperBlock + Inode Bitmap + Data Bitmap + InodeBlock + DataBlock的结构。

其中，SuperBlock占据第一个块。一个block共有4\*1024\*8=32768 bit，可以表示32768个文件或数据块的占用情况，由于一共65536个块且需要支持32768个文件，所以inode bitmap需要占一个数据块，data bitmap需要占两个数据块。

自定义的inode的结构大小为92B，所以一个block能放4\*1024/92 = 44个inode。一共需要32768个inode，所以需要32768/44 = 745个inode block。

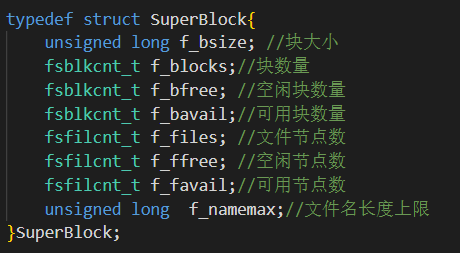
剩余的空间为datablock，去除前面的部分还剩余64787个block，大小为64787\*4KB=259148KB，大于250MB=256000KB，故满足“至少 250MiB 的真实可用空间”的要求。

总体结构大致如下表所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SuperBlock | Inode Bitmap | Data Bitmap1 | Data Bitmap2 | Inode Block | Data Block |
| 0 | 1 | 2 | 3 | 4-748 | 749-65535 |

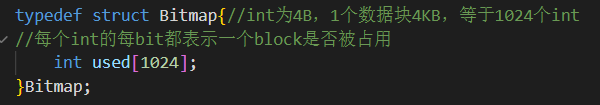
1. **SuperBlock结构**

Superblock结构未修改，表明整个文件系统的一些基本信息。



1. **Bitmap结构**

Bitmap的结构用int来实现，每个int的32位能代表32个文件或子目录的使用情况（1为被占用，0为空闲可用）。具体操作时需要使用位操作提取对应位置的使用情况。



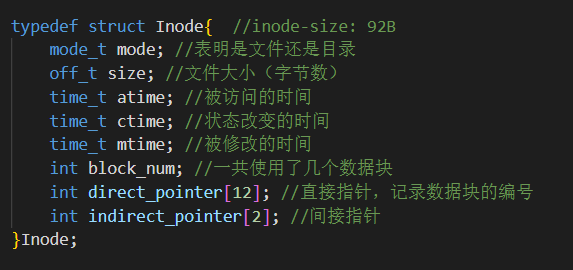
1. **Inode结构**

Inode中，有一些存储基本信息的变量，还有block\_num记录使用的数据块的数量。为了节省空间，没有使用8B的指针结构，而是用4B的int直接存储指向的数据块的实际编号，因为总共65536个块，只需要16bit便足够完全表示，所以可以用int来存储。

在indirect间接指针块中，为了防止将无意义内容识别为指针（数据块编号），将该指针块的第一个int用来记录该指针块的指针数。因此一个indirect block能存储4KB/4B – 1 = 1023个指向数据块的指针。间接指针块结构如下：

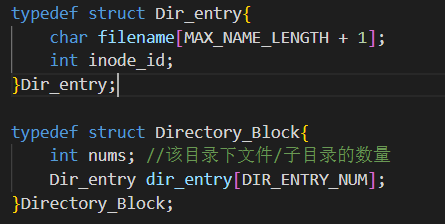
|  |  |  |  |
| --- | --- | --- | --- |
| Indirect Block | | | |
| num | Pointer1 | Pointer2 | …… |

由此可以计算一个文件最大可能的大小。12个直接指针共12\*4KB = 48KB。两个间接指针共2\*1023\*4KB = 8184KB，所以单个文件可能的最大大小=48+8184 = 8232KB > 8MB，符合要求。



1. **目录数据块结构**

目录数据块的目录项为文件或子目录，需要存储对应的名字和inode编号以供查找。所以定义Dir\_entry结构表示目录项，由存储名字的char数组和inode编号组成。Directory\_Block结构表示目录数据块，由其目录项项数和目录项数组组成。



## 二、辅助函数

使用的辅助函数数量较多，在此不一一讲解，以免篇幅过长，仅介绍它们的作用，具体的实现请见 ”fs.c”

**Read\_SuperBlock()** 读取super block块，返回一个SuperBlock结构。

**Write\_SuperBlock(SuperBlock superblock)** 将传入的参数写进super block块。

有关Bitmap的操作主要通过位运算来实现。

**Read\_InodeBitmap()** 读取inode bitmap，并查找空闲的inode，返回空闲可用的inode编号。

**Read\_DataBitmap(int blockid)** 读取参数块号对应的data bitmap，并查找空闲的datablock，返回空闲可用的data block编号

**Write\_Bitmap(int blockid, int changeid, int isdelete)** 修改bitmap块中参数对应的位置的空闲状态。如果isdelete为0，表示将空闲块修改为占用状态，即0变1；否则表示释放一个占用块，状态由占用变空闲，即1变0。

**Read\_Inode(int inode\_id)** 读取编号对应的inode并将其以Inode结构返回

**Write\_Inode(int inode\_id, Inode inode)** 将参数传入的inode结构写入编号inode\_id对应的inode。

**Read\_Dir\_DB(int blockid, void\* buffer, fuse\_fill\_dir\_t filler)** 读取一个数据块中所有文件或子目录

有关目录的操作：

**New\_Indirect\_Block(int indirect\_block)** 在参数对应的block处创建并初始化间接指针块

**Open\_New\_DirBlock(int blockid)** 将参数对应的block设置为新的目录数据块并初始化

**Insert\_In\_Dir(int dir\_inode, Dir\_entry file\_or\_dir)** 在编号为dir\_inode的目录中插入目录项file\_or\_dir。其实现情况比较复杂，因为需要找到该目录的最后一个目录块，并根据其是否已满、以及属于直接指针还是间接指针等不同情况做不同处理。

**Find\_From\_DirDB(int blockid, char\* name)** 从编号为blockid的目录块中按名字name找文件或子目录，找到则返回对应的inode编号，否则返回-1

**Find\_InWhich\_Block(int dir\_inode, char\* name)** 从编号为dir\_inode的目录中，查找名为name的文件或子目录在哪个数据块，若找到返回对应的数据块编号，否则返回-1

有关文件路径的操作：

**Get\_Inodeid\_From\_Path(char\* path)** 从路径path得到对应文件的inode编号,找不到返回-1

**Get\_FileName(char\* path)** 从路径path中提取文件名

**Get\_Father\_Path(char\* path)** 从路径path中提取父目录路径

**Find\_DB\_From\_Order(Inode inode, int order)** 找到文件的第order个数据块的块号

## 三、具体实现

需要完成的函数数量较多，所以不在此展示具体实现代码，仅仅简单讲解实现思路，具体代码请见 “fs.c”

1. **mkfs**初始化文件系统

需要初始化SuperBlock块、Bitmap块，并为根目录录入信息。

1. **fs\_getattr**查询一个目录文件或常规文件的信息

用所给路径读出对应的inode结构，并存入所给的结构体stat中即可

1. **fs\_readdir** 查询一个目录文件下的所有文件

首先用所给路径读出该目录的inode结构，然后根据block\_num依次遍历其直接指针、间接指针1和间接指针2，利用Read\_Dir\_DB函数打印每个目录数据块的信息，并最终在修改atime后重新写回该目录的inode即可

1. **fs\_read** 对一个常规文件进行读操作

因为要从offset位置开始读size大小，但是由于先定位offset再向后找size大小的情况比较复杂，所以直接读取到offset+size的位置再删除前offset个字节，操作更简单。读取时依次遍历其直接指针、间接指针1和间接指针2，过程中只要达到了offset+size处就可以停止。需要注意的是，可能文件总大小小于offset+size，那便只需要读到文件结束处即可。最终返回实际读取的大小。

1. **fs\_mkdir** 创建一个目录文件

这里操作比较简单，主要调用辅助函数。对待创建的子目录，首先通过bitmap找到空闲的inode并修改为占用状态，然后初始化子目录的inode并写入对应位置。对其父目录，则要将该子目录对应的目录项利用Insert\_In\_Dir函数，插入到父目录的目录块中即可。

1. **fs\_rmdir** 删除一个目录文件

删除子目录，首先要删除该目录本身的数据，即修改其所有指针指向的数据块在bitmap中的状态为空闲可用，以及修改其inode在bitmap中的状态为空闲可用。然后针对父目录，需要将该子目录所在的目录块中，位于其后的所有目录项依次前移一位，以达到在父目录中删除的效果。

1. **fs\_unlink** 删除一个常规文件

删除文件的操作与fs\_rmdir中删除子目录的操作相同。

1. **fs\_rename** 更改一个目录文件或常规文件的名称（或路径）

实际可以拆分为两个操作，即在旧路径中删除，在新路径中插入。不过删除操作有所不同的是，不需要修改该文件自身的inode和datablock，只需要将父目录中其后的目录项依次前移一位即可。插入操作则直接调用辅助函数Insert\_In\_Dir。

1. **fs\_truncate** 修改一个常规文件的大小信息

将更改大小分为增大和缩减两种情况。增大时需要分配新的数据块，要依次尝试直接指针、间接指针1和间接指针2，同时还要注意是否需要分配新的间接指针块等细节处理。缩减时，从末尾开始向前删除，但不需要真的删除数据块中的内容，只需要修改对应的bitmap即可，其中的数据便可自然作废。

1. **fs\_utime** 修改一个目录文件或常规文件的时间信息

只需要读出对应inode并修改时间即可

1. **fs\_mknod** 创建一个常规文件

与fs\_mkdir类似，主要调用辅助函数。对待创建的文件，首先通过bitmap找到空闲的inode并修改为占用状态，然后初始化文件的inode并写入对应位置。对其父目录，则要将该文件对应的目录项利用Insert\_In\_Dir函数，插入到父目录的目录块中即可。

1. **fs\_write** 对一个常规文件进行写操作

根据提示，write前先检查大小是否足够，所以先调用fs\_truncate分配足够大小。然后确定需要写入的数据块范围，遍历它们，依次写入buffer中的内容即可。

1. **fs\_statfs** 查询文件系统整体的统计信息

读出superblock块的内容存入所给结构体即可。

根据PPT提示，fs\_open、fs\_release、fs\_opendir、fs\_releasedir无需修改

## 四、效果展示

用traces中的样例测试，除了测试点13拷贝test.in有问题外，均能正常运行且输出与相应ans一致。

例如测试10的效果：

