fslab实验报告

郭丹琪 2018202067

1. 组织结构

一共65536个块，每个块的大小为4K。要求至少支持32768个文件或目录，要求有250MB真实可用空间，即存文件内容的空间至少250MB。

组织结构如下图所示。其中super block占一个块，块号为0。inode bitmap占一个块，块号为1。data bitmap占两个块，分别为data bitmap1和data bitmap2，块号分别为2和3。inode block从4开始，每个inode的大小为104字节，一个block里可以放39个inode，因为需要至少支持32768个文件，所以一共需要841个block，inode块从4到844。data block从845开始到65535，一共有64691个data block，真实可用空间为64691 * 4KB = 258764KB = 252.69MB，满足要求。

Super Block	Inode Bitmap	Data Bitmap1	Data Bitmap2	Inode Block	Data Block

Bitmap的结构体：

```
typedef struct Bitmap{
    int if_use[1024];
}Bitmap;
```

每个inode或data block只需要用一位来记录是否被占用，1表示被占用，0表示空的可用。int有32位，可以用来记录32个位置的block或者inode是否被占用，在使用的时候用位运算得到对应的记录位。

每个Inode Block的结构如下：

<table>
<tr><td colspan="4">Inode Block:</td></tr>
<tr><td colspan="3">Inode Block1</td><td>Inode Block2</td><td rowspan="2">……</td></tr>
<tr><td>Inode0</td><td>Inode1</td><td>……</td><td>……</td></tr>
</table>

Inode的结构体为：

```
typedef struct Inode{
    mode_t mode; //文件 or 目录
    off_t size; //文件字节数
    time_t atime; //被访问的时间
    time_t ctime; //状态改变的时间
    time_t mtime; //被修改的时间
    int block_num; //一共有几个块
    int direct_pointer[15]; //记录数据块的编号
    int indirect_pointer[2]; //indirect block的编号
}Inode;
```

pointer里记录的是数据块的实际编号，可以直接用disk_read读到对应数据块的内容。要求单个文件最大8MB。Inode中有15个direct pointer，记录的数据块的空间为15 * 4KB = 60KB。有2个indirect pointer，记录indirect block的编号。indirect block的结构如下图，块中第一个int存的是该块里当前存了几个pointer，所以每个indirect block里最多可以存1023个pointer。由2个indirect pointer可以记录的空间为2 *（1024 - 1） * 4KB = 8184KB。所以可以支持单个文件最大8244KB，即8MB + 52KB。

Indirect block			
num	Pointer1	Pointer2	……

目录的Inode和文件的Inode形式是相同的，但是目录的数据块形式不同。目录的数据块结构：

```
typedef struct Dirent{
    char file_name[MAX_FILE_NAME + 1];
    int inode_num; //inode number
}Dirent;

//目录的数据块结构结构体
typedef struct Dic_Datablock{
    int use_num; //该块里已用的数目
    Dirent dirent[DIRENT_NUM_PRE_BLOCK];
}Dic_Datablock;
```

目录的 Data Block:			
use_num	Dirent1	Dirent2	……

目录的数据块里存着一个Dic_Datablock结构体，其中的use_num记录当前块中记录多少个文件或子目录。每个文件或子目录的记录在Dirent结构体里，记录了其名字和inode编号。

2. 实现函数的重要细节

2.1 辅助函数

把一些常用的、繁琐的操作封装成辅助的函数，在完成文件系统的函数时逻辑会更清晰，代码更简洁。

写、读相关的辅助函数：

```
void Write_Superblock(Superblock superblock)//写super block
Superblock Get_Superblock()//获得super block
int Find_Free(int block_num)//找可用的inode或data block，返回对应编号
void Write_Bitmap(int block_num, int write_num)//写入的时候修改bitmap
void Rm_Bitmap(int block_num, int write_num)//删除的时候修改bitmap
void Write_Inode(int inode_num, Inode inode)//写inode
Inode Get_Inode(int inode_num)//用inode_num读取inode
void Init_Indirect_Block(int indirect_block)//初始化indirect block
```

其中Find_Free返回的和Write_Bitmap、Rm_Bitmap用到的inode编号或者data block编号是对应在bitmap中的编号，在其他地方使用的时候需要把data block编号加上DATA_START得到实际的块号，inode编号需要拆解出所在的inode块和在其中的偏移。Find_Free、Write_Bitmap和Rm_Bitmap中对Bitmap的操作用到的主要是位运算，先根据编号算出是在Bitmap中的哪个int中，偏移是多少，然后对

对应int进行相应移位，找到对应的bit。

与文件路径相关的函数：

```
char* Get_Father_Path(char* path)//得到父目录的路径
char* Get_File_Name(char* path)//得到文件名
int Path_To_Inode_Num(char* path)//由路径得到文件或目录对应的inode num
```

在Path_To_Inode_Num中，用path每拆解出一个目录名，就用后续Find_In_Dic和Find_In_Dic_Data函数找到该目录对应的inode_num。然后用该inode_num作为下一层的father_inode_num，拆解出的下一个目录名就用该father_inode_num找其对应的inode_num。

在目录里找数据：

```
int Find_In_Dic_Data(int data_block, char* name) //从data block里找文件或子目录的inode_num
int Find_In_Dic(int dic_inode_num, char* name) //找到文件或子目录是在存目录的哪个data block里，返回的是data block的块号
void Print_Dic_Data(int data_block_num, void *buffer, fuse_fill_dir_t filler)//fs_readdir中用到的，读出一个data block里的所有文件
```

用Find_In_Dic可以由父目录的inode和文件的名字得到父目录中存文件的Dirent的数据块的块号，方便后续从父目录中删除文件或子目录。而用Find_In_Dic_Data是由存文件的Dirent的数据块的块号和文件的名字得到文件的inode num。

把新文件或子目录插入目录中：

```
void Insert_File_Into_Dic(int dic_inode_num, Dirent file_dirent)//往父目录里插文件或子目录
```

在创建完新文件或子目录后，需要用该函数把对应的file_dirent存入父目录的数据块，dic_inode_num是对应父目录的inode num。在该函数中，先得到父目录的最后一个数据块，然后判断该数据块是否已满，未满的话将新的文件或子目录的dirent放入，已满的话分配新块，再调用该函数。这里比较繁琐的是父目录里的数据块可能有多种情况，可能最后一个数据块是direct的，也可能是indirect block中的，而indirect block又要判断是在第一个indirect block还是第二个中。在分配新块的时候也得判断是可以直接放入direct pointer还是要在indirect block中。

2.2 创建

```
int mkfs()
```

对super block、几个bitmap、根目录的inode进行了初始化设置。对根目录的inode的设置分三步，首先用Find_Free找到空的inode位置inode_num，用Write_Bitmap占下这个位置，然后对其inode进行初始化，最后用Write_Inode把初始化后的inode写到inode block里对应inode_num的位置上。

```
int fs_mkdir (const char *path, mode_t mode)
int fs_mknod (const char *path, mode_t mode, dev_t dev)
```

创建新文件和新目录的步骤是一样的，总的来说就是先创建文件或目录本身，将其写入块中，然后再将其插入目录树中。因为将繁琐的步骤都放在了辅助函数中，所以在实现上较容易，只要调用几个辅助函数即可，具体步骤为：

```
1.用Find_Free找到空的inode位置，用Write_Bitmap占下这个位置
2.创建对应文件或目录的inode
3.用Write_Inode把初始化后的inode写到inode block里的对应位置上
4.创建文件或目录对应的dirent
5.用Get_Father_Path和Path_To_Inode_Num找到对应父目录
6.用Insert_File_Into_Dic将新文件或目录的dirent写到父目录中
```

2.3 删除

```
int fs_rmdir (const char *path)
int fs_unlink (const char *path)
```

删除文件或目录的步骤也是一样的，总的来说就是先将其从父目录的数据中去掉，然后在data bitmap里删除对应的所占的数据块，最后在inode bitmap里删除所占的inode。具体步骤为：

```
1.用Find_In_Dic找到该文件或子目录在父目录中的数据块
2.删除数据块中对应的dirent，也就是把对应dirent后的dirent往前移
3.然后用Find_In_Dic_Data找到该文件或子目录的inode，找到其中记录的数据块并用Rm_Bitmap在
data bitmap里删除
4.用Rm_Bitmap在inode bitmap里删除所占的inode
```

2.4 读、打开

```
int fs_getattr (const char *path, struct stat *attr)
int fs_statfs (const char *path, struct statvfs *stat)
```

把相应内容读出来，然后存到所给的结构体里即可。

```
int fs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t
offset, struct fuse_file_info *fi)
```

先用Path_To_Inode_Num找到该目录对应的inode_num，然后找到目录中每一个数据块的编号，用辅助函数中的Print_Dic_Data读出数据块中的每一个文件和子目录的dirent。

这个函数主要麻烦在找目录中每一个数据块的编号，因为数据块编号除了存在direct pointer中的，还有存在indirect block中的。在找目录中每一个数据块的编号时，先根据inode里存的block_num，即数据块的数目，遍历direct pointer，然后再判断是否还有indirect block。如果有的话，通过indirect pointer读取indirect block，再根据indirect block里第一个int记录的该block里存的pointer的数目，遍历indirect block中所有pointer。用遍历到的这些pointer到Print_Dic_Data函数里就可以读出对应数据块中每一个文件和子目录。

```
int fs_read(const char *path, char *buffer, size_t size, off_t offset, struct
fuse_file_info *fi)
```

要从文件的offset位置读出size大小的内容到buffer中，即读文件offset ~ offset+size区域的内容。为了方便起见，从文件的第一个数据块开始遍历，将其中的内容都记录content中，即将文件从开始到offset + size部分的内容复制到content中，最后再将content里从offset开始的部分复制到buffer中。在遍历数据块的时候，先遍历direct pointer指向的数据块，若读出内容的大小满足offset + size，则退出遍历，否则读取indirect block，遍历indirect block中的pointer指向的数据块。若数据块都遍历完后读出内容的大小仍不够offset + size，则返回实际读出的大小。

2.5 写

```
int fs_write (const char *path, const char *buffer, size_t size, off_t offset,
struct fuse_file_info *fi)
int File_i_Data_Block(Inode inode, int i)//找inode文件里第i个数据块
```

写的部分跟读的有些类似，只是在写的时候，需要先用fs_truncate给文件分得足够的大小，然后确定从offset到offset + size的数据块，遍历这些数据块，将buffer里的内容写在对应数据块里。同时为了简化函数，将从文件中找第i个数据块的操作封装成File_i_Data_Block函数。fs_write的具体步骤：

```
1.用fs_truncate给文件分配offset + size的大小
2.计算offset、offset + size对应的数据块分别是文件中的第几个数据块、在数据块中对应的偏移
3.用File_i_Data_Block遍历这些数据块，将buffer中对应的内容写入。如果是起始的数据块，要从偏移
处开始写。起始数据块和结尾数据块要注意写入的大小。
```

```
int fs_truncate (const char *path, off_t size)
```

更改文件大小要分两种情况，一种是变大，一种是变小。先计算之间相差的块数，变大的就分配新的数据块给文件，变小的就去掉文件末尾的数据块。分配新的数据块时要注意新分配的块的pointer可以放在哪里，是否需要分配新的indirect block。

```
int fs_utime (const char *path, struct utimbuf *buffer)
```

把对应inode读出来修改时间即可。

```
int fs_rename (const char *oldpath, const char *newname)
```

实际上rename就是把对应的dirent从旧的父目录的数据中删除，然后放到新的父目录的数据中。从旧的父目录的数据中删除的操作和删除目录或文件中的一样，放到新的父目录中的操作和创建目录或文件中的一样，直接调用封装好的辅助函数Insert_File_Into_Dic。

3. 遇到的问题及解决方案

1. 在测试mkdir的时候发现有错误，报错mkdir: cannot create directory ‘dir1’: Input/output error，用ls的时候也出现报错ls: cannot access dir1: Input/output error，而且虽然都有报错，但是创建的dir1、dir2目录的名字还是有打印出来。初步判断是在fs_mkdir中出现bug。首先发现在用find_free函数找空数据块的地方有错，因为在find_free函数中是在找bitmap中空闲的编号，对应的是data block在数据块区域的编号，并不是全局所有块中的编号，实际在用的时候还需要加上数据块的起始编号DATA_START。把这个地方的bug解决掉后发现还有错误。之后试着去掉可能出现问题的各个地方，发现是在Insert_File_Into_Dic函数里把目录的数据块写回去的地方出错。然后尝试了很久后发现错误是在用memcpy时，把目录的数据块的结构体复制到空间的时候，没有把结构体指针转为char*，所以出错。把程序中所有出现memcpy的地方都修改后，测试结果就正确了。但是后续得到memcpy中的指针类型实际不影响复制，在复制的时候都会被当成void *，所以不确定错误判断是否正确。
2. 在创建文件的时候测试touch abcdefghijklmnopqrstuvwx，但是ls的时候出现报错ls: cannot access abcdefghijklmnopqrstuvwx: No such file or directory，与上一个问题相同的是abcdefghijklmnopqrstuvwx文件名还是有打印出来。考虑到创建名字较短的文件不会出现这个问题，所以判断可能是因为名字的char数组刚好是最大文件名的话可能不够，所以把文件名的char数组多开了一位，问题解决。

4. 反思总结

1. 发现每次做lab的时候都会有一些奇奇怪怪的地方出错，主要的大的逻辑部分、代码部分都是对的，但是结果要么就没法运行，要么会有一些小问题。这种时候debug会花很长时间，就一直找不到问题在哪，就会在这种地方卡非常久。但最后就会发现bug出现在一些跟结构设计没什么关系的小的地方。反思就是今后写代码的时候，在写的时候就要注意小的、可能会出错的地方，不能直接按思路逻辑一路写下去，写一部分要停下来看看有没有可能出错的地方。
2. 比较复杂的步骤可以封装成函数，这样在实现操作的主要的函数的部分就不容易出错，逻辑思路也会比较清晰。
3. 一些确定的数字或者简单运算可以用宏定义来写，这样代码会更清晰、明确。
4. 一开始就要分清楚一些相似的量或函数，这样才能减少后续不必要的麻烦。