FSlab report

罗思佳2021201679

一、组织结构

（一）块设备结构

汇总一下说明文档中的信息：

虚拟块设备大小：256M

至少250MB的真实可用空间

访问粒度（块大小）：4096字节 （4KB)

块编号：0到65535

至少支持文件及目录数：32768

文件名最大长度：24字节

单个文件最大大小：8MB

首先对于**superblock**，用一个块来存，块号为0。直接使用fuse.h中提供的statvfs结构体来存储，不用自定义结构体了。

对于**inodebitmap**，因为可以支持32768个文件或目录，一个bit对应一个文件，所以大小是2^15bit，即2^12byte = 4kb,用一个块就可以放下。

对于**databitmap**，因为总的块数是65535，为2^16，一个块能放2^15，所以需要2个块。

对于一个文件，最大为8MB，一个块是4KB，所以需要inode至少指向2^11个块，一个间接指针指向的块能够存下4096/4=2^10个块，所以需要两个间接指针，直接指针选择12个。

inode的大小设为**128B**，共有2^15个inode，所以总大小为2^22=4MB，需要1024个块。

故前4个块:

0	1	2	3
superblock	inodebitmap	databitmap1	databitmap2

第5到第1028个块：

4	5	.....	1027
inode	inode	.....	inode

因为总共有65536个块，除去前1028个块，剩下的64508个块都作为datablock。

真实可用空间为 256M * （64508/65536） ≈ 252M ，满足要求。

（二）相关结构体

1.inode结构体

inode的存储信息见下面的注释

```
typedef struct
{
    mode_t mode;//文件或目录
    nlink_t nlink;//文件的链接数
    uid_t uid;//文件所有者
    gid_t gid;//文件所有者的组
    off_t size;//文件字节数
    time_t atime;//被访问时间
    time_t mtime;//被修改时间
    time_t ctime;//状态改变时间
    int block_num;//一共有几个块
    int block[P_NUMBER];//包括直接指针和间接指针指向的block的块号
} inode;
```

2.directory

代表一个文件或目录，包含文件名和inode编号

```
typedef struct {
    char file_name[FILENAME_SIZE];//文件名
    int inode_num;//inode编号
}directory;
```

二、函数的实现

（一）辅助函数

1.读写inode、datablock相关：

```
void change_inodebitmap_state(int inode_num);//改变inodebitmap状态
void change_databitmap_state(int data_num); //改变databitmap状态
```

用于改变对应编号位置的bitmap状态，即将0改为1，将1改为0。

```
struct inode initialize_inode(mode_t f_mode);  //初始化inode
```

初始化inode的成员

```
void init_datablock(int data_num);  //初始化datablock
```

将datablock初始化为0

```
int get_inode_num_by_path(char * path);//根据路径获取inode number
struct inode get_inode_by_num(int inode_num);//根据编号读取inode
```

分别是根据文件路径获取inode number， 根据inode number获取inode

```
int get_indirect_num(int block_pos);//获取间接指针的块号
void get_indirect_block(int block_pos, int * inode_pointer);//读取间接指针指向的
datablock
```

分别是获取间接指针指向的数据块的块号，读取间接指针指向的数据块到inode_pointer中。

```
int get_data_num(struct inode inode_content, int block_num);//根据块号找到datablock
```

函数逻辑：如果block_num小于DIRECT_NUM，说明是直接指针否则在间接指针指向的数据块中。

```
int find_empty_inode();          //找到空闲的inode块
int find_empty_datablock();    //找到空闲的datablock
```

对bitmap进行遍历，找到第一个空闲的inode/datablock。

```
void write_inode(int inode_num, struct inode inode_content);//根据块号写入inode
```

inode_num是在inode bitmap中的编号，需要根据inode_num计算出所在的inode块号和在块中的偏移量，然后进行`disk_read`和`disk_write`。

2.路径、目录相关：

```
char* get_upper_path(char * path); //获取上一级目录的路径
int find_directory(int inode_num, char * file_name, int * inum_pointer); //根据
inode number和name找到目录
void read_data_in_dir(int inum,void * buffer, fuse_fill_dir_t filler); //目录的
datablock
void get_dir_content(int dir_pos, struct directory * dir_pointer); //读取目录内容
```

`find_directory`函数用于根据inode number和filename找到目录，对inode指向的数据以及所含的文件进行遍历，找到了则返回目录的inode number，找不到返回-1。

```
int insert_file_to_dir(const char * path, int inode_num);//往目录插入新的文件/目录
int insert_dir_info(int dir_pos, struct directory dir);//插入目录内容
int rm_directory(int inode_pos, int inode_num, int * dir_inum_list); //删除目录
```

`insert_file_to_dir`：先找到该路径对应的父目录，尝试将新目录插入到直接指针指向的数据块，一种情况是插入失败且block_num小于直接指针个数，新开辟一个块再插入；另一种情况是插入失败且直接指针满了，则尝试插入到间接指针指向的数据块中，也要判断间接指针是第一个还是第二个。

（二）核心函数

```
int mkfs()
```

包括对superblock、两个bitmap和根目录的inode进行初始化。对根目录的root_inode的初始化分为几步，首先是对inode里的几个成员赋初值，然后找空闲的inode和datablock位置，写入bitmap，然后写对应的inode和datablock。

```
//查询一个目录文件或常规文件的信息
int fs_getattr(const char *path, struct stat *attr)
```

根据路径path找到inode编号，进而找到inode，然后对attr的成员逐一赋值。

```
//查询一个目录文件下的所有文件
int fs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t
offset, struct fuse_file_info *fi)
```

先调用`get_inode_num_by_path`函数获取path对应的inode编号，然后调用`get_inode_by_num`函数获取目录的dir_inode。先读取直接指针指向的数据块，调用`read_data_in_dir`函数，然后再读取间接指针指向的数据块（如果有的话），通过间接指针读取indirect block，然后遍历该indirect block存有的indirect pointer，用`read_data_in_dir`函数读取指向的数据块存有的文件和目录。最后需要更新访问时间`atime`。

```
//对一个常规文件进行读操作
int fs_read(const char *path, char *buffer, size_t size, off_t offset, struct
fuse_file_info *fi)
```

先根据path获取文件的inode number，然后获取inode，计算出要读的字节`read_size`，然后读直接指针指向的内容，如果读出的内容大小满足offset+size，则退出遍历，否则继续读indirect block中的pointer指向的数据块的内容，需要调用`read_indirect_num`函数和`get_indirect_block`函数获取indirect block存有的pointer，进一步获取指向的数据块。最后返回读取到的字节数。

```
//创建一个目录文件
int fs_mkdir (const char *path, mode_t mode)
```

先创建并初始化dir_inode，调用`find_empty_inode`函数找未使用的inode编号和未使用的datablock编号，如果没找到就返回-ENOSPC；然后更新两个bitmap，调用`insert_file_to_dir`函数插入新文件，插入失败则返回-ENOSPC，成功则写入inode和datablock。

```
//删除一个目录文件
int fs_rmdir (const char *path)
  //删除一个常规文件
int fs_unlink (const char *path)
```

两个函数的思路基本一样。先判断根据路径能否找到inode编号，如果不能就返回0；然后根据编号获取inode，先释放直接指针指向的块，需要调用`change_databitmap_state`函数；再释放indirect block的pointer指向的数据块。之后再调用`change_inodebitmap_state(inum)`函数将对应的inode number位置改为未使用，最后对父目录进行更新。

```
//更改一个目录文件或常规文件的名称（或路径）
int fs_rename (const char *oldpath, const char *newname)
```

实际上就是把对应的目录从旧的的父目录的数据块中删除，放到新的父目录的数据块中。先获取上一级目录的inode编号和inode，调用`rm_directory`函数删除，如果未成功删除，则从indirect block中找并删除。之后更新父目录的mtime和ctime，调用`insert_file_to_dir`函数插入到新目录中。

```
//修改一个常规文件的大小信息
int fs_truncate (const char *path, off_t size)
```

修改文件大小分为两种情况，变大和变小。首先还是先获取文件的inode，如果文件要变大，就分配新的块给文件，注意分配新的块是在直接指针还是间接指针；变小就删掉文件末尾的数据块。

```
//修改一个目录文件或常规文件的时间信息
int fs_utime (const char *path, struct utimbuf *buffer)
```

根据path读出对应的inode，然后修改buffer的时间，更新inode的ctime然后写回去即可。

```
//创建一个常规文件
int fs_mknod (const char *path, mode_t mode, dev_t dev)
```

与`f_mkdir`类似，使用`find_empty_inode`找到未使用的inode编号，改变inodebitmap的状态，然后创建并初始化常规文件inode，插入到目录中，并写到inode里。

```
//对文件进行写操作
int fs_write (const char *path, const char *buffer, size_t size, off_t offset,
struct fuse_file_info *fi)
```

写的时候需要先用`fs_truncate`分配足够的大小，定义开始写的块号`begin_block`和开始写的偏移量`begin_off`，结束写的块号`end_block`和偏移量`end_off`，定义一个变量`buffer_off`记录写到哪里了，然后遍历这些数据块，将buffer中的内容写入。起始数据块和结尾数据块要注意写入的大小。

```
//查询文件系统整体的统计信息
int fs_statfs (const char *path, struct statvfs *stat)
```

把superblock的内容读出来，然后存到stat里。

三、遇到的问题

1.一开始没有搞清楚fs_rename函数的作用，以为重命名只需要将文件的父目录的datablock进行修改就可以了，但这样没有考虑移动文件的情况，上网查资料得知，对应的mv命令的作用是`为文件或目录改名、或将文件或目录移入其它位置`，所以rename需要先将对应的directory从父目录下删除，再插入到新目录下，就考虑了全部的情况。

2.还遇到了一些指针的问题，可能还是因为对指针的使用不够熟练，所以在定义和指针操作上还是会犹豫。例如`get_upper_path`函数，返回值类型是char*，最初定义了局部变量指针f_path就直接返回了，后来调试的时候意识到了这个严重而又容易被忽略的错误，需要malloc动态分配内存。

四、实验总结

这次的实验的代码量还是挺大的，而且需要自己从0开始写一个文件系统，还是挺有难度的，所以一步一步来，先设计好组织结构，定义好相关数据结构，然后根据可能的需求添加一些辅助函数，例如对bitmap的操作、对inode、datablock的操作，将一些复杂的步骤封装成函数，这样在实现主要函数时就不容易在细节上出错，思路也会更加清晰。同时用好宏定义也是很重要的，因为实验涉及到的数字很多，容易混淆，出错了也不好改，使用宏定义可以使代码看起来更加明白易懂。除此之外，写代码的时候还是会因为粗心而出现各种各样的小问题，调试起来很费劲，以后来要进一步加强练习和总结踩过的坑，避免犯同样的错误。