File System Lab

周炎亮 2018202196

一、实验目标

完成一个运行在用户态的文件系统(FUSE)，对（模拟的）块设备进行读写，从而实现一些基本的文件读写以及管理功能。

二、实验思路

（一）、文件系统基本组成

本次实验的思路参照课件中的**VSFS (Very Simple File System)**，将block分为superblock、i-bmap、d-bmap、inode table以及data block。并结合本次实验的要求进行分区。

superblock——superblock原本的用途是记录文件系统的一些参数，例如有多少inode, data block, inode table的位置等，但是这些信息都已经在之后的宏定义中有所体现，故是“无用的”。不过在之后的fs_statfs 函数中需要返回剩余可用节点个数以及剩余可用块个数的信息，因此可以使用superblock来记录。

```
typedef struct {
    inode_id_t free_inode_num;
    block_id_t free_data_block_num;
    char padding[BLOCK_SIZE-sizeof(inode_id_t)-sizeof(block_id_t)];
} superblock_t;
```

其中`inode_id_t`以及`block_id_t`均为自定义的`unsigned short`类型。

方便起见，令superblock占用一个块。

i-bmap——因为i-bmap只需存储inode是否已被占用的信息，加上每个block的大小为4096字节，为了最大化利用空间，用一位来存储对应inode是否可用的信息，因此单个block可存放的inode信息的总数为4096*8=32768，刚好为inode的总数，因此仅需使用一个block作为i-bmap即可。

d-bmap——d-bmap同i-bmap，也是用一位存储一个block是否可用的信息。因为block的总数为65536，因此使用两个block来存放。

inode table——要确定inode table占用的block个数，首先要确定inode的总数以及单个inode占用的空间大小。根据实验要求，inode的总数INODE_NUM为32768。至于单个inode占用的空间大小，参考课件中VSFS的inode定义：

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

提取其中有用的部分为：mode、size、time(atime)、mtime、ctime以及指针。通过在服务器上查看前五项对应类型的大小可知，前五项的总大小为36字节。因为`BLOCK SIZE` $= 4096 = 2^{12}$，为了存取的方便，最好将单个inode大小设置为2的幂，即64字节或128字节。对于pointer，因为其存储的为block的编号，根据之前的定义，block编号的类型`block_id_t(unsigned short)`只占用2个字节，又在分析中发现，indirect pointer指向的block刚刚好可以存放$4096/2 = 2048$个pointer，这些指针所指向的block的大小之和刚好为$2048 * 4096 = 8M$，即实验要求的最大文件的大小，因此理论上仅使用一个indirect pointer即可满足条件。

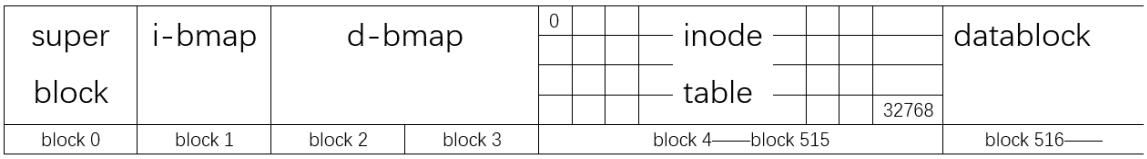
如果在之前五项的基础上加上该indirect pointer，并再增加一个同类型的indirectPtr_num_in_block用以记录其指向的block中存放的有效pointer个数，inode的大小也仅为40字节，最小仍有$64 - 40 = 24$字节的空间可供利用，因此还可以再加上11个direct pointer以及记录有效direct pointer个数信息的directPtr_num，刚好使单个inode大小达到64字节。定义如下：

```
#define MAX_DIRECT_PTR_NUM 11
//size:64
typedef struct {
    //size:24
    time_t  atime;
    time_t  mtime;
    time_t  ctime;
    //size:12
    off_t   size;
    mode_t  mode;
    //size:28
    unsigned short directPtr_num;
    unsigned short indirectPtr_num_in_block;
    block_id_t directPtr[MAX_DIRECT_PTR_NUM];
    block_id_t indirectPtr;
} inode_t;
```

这样，单个block可以存放的inode个数`INODE_NUM_PER_BLOCK`为$4096/64 = 64$。存放所有inode的block个数`INODE_BLOCK_NUM` $= 32768/64 = 512$。

data block——在总共的65536个block的基础上，去掉前面的1+1+2+512=516个block，即为可用的data block。

因此，block的分配如下：



（二）、其他相关定义

本实验中自定义的宏定义及相关数据类型如下：

```
typedef unsigned short inode_id_t;
typedef unsigned short block_id_t;

#define ROOT_INODE (inode_id_t)0    //根目录对应的inode编号，设其为0

#define BIT_PER_BLOCK (8*4096)
#define SUPERBLOCK_ID 0 //superblock块编号
#define IBMAP_ID 1  //i-bmap块编号
#define DBMAP_ID 2  //d-bmap块编号
#define INODE_NUM 32768
#define INODE_NUM_PER_BLOCK (BLOCK_SIZE/sizeof(inode_t))    //单个块中inode个数
#define INODE_BLOCK_NUM (INODE_NUM/INODE_NUM_PER_BLOCK)     //inode table占用的块总数
#define INODE_BLOCK_START_ID 4  //inode table起始块编号
#define DATA_BLOCK_START_ID (INODE_BLOCK_START_ID+INODE_BLOCK_NUM) //datablock起始块编号
```

此外，因为目录inode对应的block存放的是该目录下的文件（子目录或普通文件）信息，在本实验中要用到的为文件的inode编号以及文件名，因为要求中的文件名长度为24，故上述两项的总大小为26字节。也为了存取方便，将其padding至32字节，故对应的定义如下：

```
#define FILE_NAME_LENGTH 24
typedef struct {
    char name[FILE_NAME_LENGTH];
    inode_id_t inode_id;
    char padding[6];
} entry_t;
```

（三）、代码思路

一些自定义函数及相关作用如下（解释放在最后）：

`void init_fs()`——被`mkfs()`调用，初始化整个文件系统：初始化superblock中的两个参数，初始化i-bmap和d-bmap所有位为0，并将i-bmap前516个位置为1，代表不可用。

`inode_t read_inode(inode_id_t inode_id)`——根据inode的编号从inode table中读取相应inode信息并返回。

`void write_inode(inode_id_t inode_id,inode_t inode)`——将传入的inode写入inode_id对应的inode table位置中。

`void inode_init(inode_id_t inode_id,mode_t mode)`——初始化编号为inode_id的inode，将其三个时间设为time(NULL)，mode设为传入的参数mode（REGMODE或DIRMODE），其余参数均设为0，并写入对应的inode table中。

`bool find_free_inode(inode_id_t *inode_id)`——遵循first fit原则，从头开始在i-bmap中寻找可用inode结点，若存在则将其赋值给inode_id，返回true，否则返回false。

`bool read_inode_map(inode_id_t inode_id)`——判断inode_id对应的inode是否已被占用，若是返回true，否则返回false。

`void write_inode_map(inode_id_t inode_id,bool flag)`——将inode_id位置的i-bmap根据flag的真假设为1/0。

`bool find_free_block(block_id_t *block_id)`、`bool read_block_map(block_id_t block_id)`以及`void write_block_map(block_id_t block_id,bool flag)`类似于上述三个inode的函数。

`void *get_buffer_from_inode(inode_t inode)`——根据inode中的pointer信息从block中读取inode.size大小的数据并返回。

`int get_entry(entry_t *entry,inode_t inode,const char *path,size_t name_length)`——根据子路径path以及对应的长度和父目录的inode信息寻找文件，若找到则将其信息赋值给entry并返回1，否则返回0。

`int find_inode_by_path(const char *path,inode_t *inode,inode_id_t *inode_id)`——根据路径寻找inode以及inode_id，若成功则返回0，若中途存在目录不存在或找不到文件等信息返回相应错误值。

`int find_parent_inode_by_path(const char *path,inode_t *parent_inode,inode_id_t *parent_inode_id,char child_path[FILE_NAME_LENGTH])`——根据路径找到父目录的inode和inode_id信息，并将该path对应的文件名赋值给child_path，返回0。若出错则返回对应错误值。

`int inode_block_realloc(inode_t inode,inode_id_t inode_id,void* buffer,off_t new_size)`——本质是将文件覆盖重写：删除inode对应文件存储的所有内容，将其内容重新填充为new_size大小的buffer，并更新其ctime和mtime。若成功则返回0，否则返回对应的错误值。

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

该函数只需调用find_inode_by_path函数根据path找到相应inode，若函数返回值为0则将inode中的信息赋值给attr，否则直接返回函数的错误值。代码如下：

```
int fs_getattr (const char *path, struct stat *attr)
{
    //printf("Getattr is called:%s\n",path);
    inode_t inode;
    int error=find_inode_by_path(path,&inode,NULL);
    if(error<0)
        return error;
    else
    {
        attr->st_mode=inode.mode;
        attr->st_nlink=1;
        attr->st_uid=getuid();
        attr->st_gid=getgid();
        attr->st_size=inode.size;
        attr->st_atime=inode.atime;
        attr->st_mtime=inode.mtime;
        attr->st_ctime=inode.ctime;
    }
    return 0;
}
```

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

本函数只需先调用find_inode_by_path函数根据path找到相应inode并判断其是否为目录文件，之后再调用get_buffer_from_inode函数从目录inode中获得其存储的文件信息entry数组，一一将每个文件名赋值给buffer。最后更新该目录的atime并利用write_inode函数写回至inode_table。代码如下：

```
int fs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t
offset, struct fuse_file_info *fi)
{
    //printf("Readdir is called:%s\n", path);
    inode_t inode;
    inode_id_t inode_id;
    int error=find_inode_by_path(path,&inode,&inode_id);
    if(error<0)
        return error;
    if (inode.mode!=DIRMODE)
        return -ENOTDIR;
    int file_number=inode.size/sizeof(entry_t);
    entry_t *file=get_buffer_from_inode(inode);
    for(int i=0;i<file_number;i++)
    {
        static char name_buffer[FILE_NAME_LENGTH + 1];
        memcpy(name_buffer, file[i].name, FILE_NAME_LENGTH);
        filler(buffer, name_buffer, NULL, 0);
    }
    free(file);
    inode.atime=time(NULL);
    write_inode(inode_id,inode);
    return 0;
}
```

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

该函数类似fs_readdir，只需先调用find_inode_by_path函数根据path找到相应inode并判断其是否为常规文件，然后根据该inode利用get_buffer_from_inode获得其存储的内容，加上偏移量后赋值给buffer。最后更新inode的atime并返回读取到的ret_size。

```
int fs_read(const char *path, char *buffer, size_t size, off_t offset, struct
fuse_file_info *fi)
{
    //printf("Read is called:%s\n",path);
    inode_t inode;
    inode_id_t inode_id;
    int error=find_inode_by_path(path,&inode,&inode_id);
    if(error<0)
        return error;
    if (inode.mode!=REGMODE)
        return -EISDIR;
    off_t ret_size=(inode.size-offset)<size?(inode.size-offset):size;
    void *p=get_buffer_from_inode(inode);
    //buffer+=offset;
    memcpy(buffer,p+offset,ret_size);
    free(p);
    inode.atime=time(NULL);
    write_inode(inode_id,inode);
    return ret_size;
```

```
}
```

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

这两个函数一个是新建文件夹，一个是新建普通文件，其本质是一样的（都是创建文件），唯一的不同即为文件的mode不一样。因此可以使用同一个函数`int make_new_file(const char *path, mode_t mode)`来创建新文件。

要创建文件，首先需要判断是否有空余的inode和block，因此需要调用find_free_inode和find_free_block函数。之后，利用inode_init函数初始化新得到inode（因为文件被创建时为空，因此暂不需要为其分配block）。因为可用inode个数减一了，故也要对superblock作相应的修改。之后，找到其父目录的inode和inode_id信息，为该新文件新建一个entry，调用add_entry函数将该entry放入至父目录对应的block中。代码如下：

```
int make_new_file(const char *path, mode_t mode)
{
    inode_id_t new_inode_id;
    block_id_t new_block_id;
    if(!find_free_inode(&new_inode_id)||!find_free_block(&new_block_id))
        return -ENOSPC;
    //初始化节点，修改superblock
    inode_init(new_inode_id,mode);
    write_inode_map(new_inode_id,1);
    superblock_t superblock;
    disk_read(SUPERBLOCK_ID,&superblock);
    superblock.free_inode_num--;
    disk_write(SUPERBLOCK_ID,&superblock);
    //寻找父目录的inode
    inode_id_t parent_inode_id;
    inode_t parent_inode;
    char child_path[FILE_NAME_LENGTH];
    int
error=find_parent_inode_by_path(path,&parent_inode,&parent_inode_id,child_path);
    if(error<0)
        return error;
    //将新文件加入到父目录下
    entry_t entry;
    entry.inode_id=new_inode_id;
    memcpy(entry.name,child_path,FILE_NAME_LENGTH);
    return add_entry(parent_inode,parent_inode_id,entry);
}
```

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

类似上面两个，本质都为删除文件，且删除时还无需考虑文件mode。因此也使用相同的`int delete_file(const char *path)`函数。

要删除一个文件，首先要判断该文件的存在性。之后，调用find_parent_inode_by_path函数获得其父目录inode信息，利用get_buffer_from_inode获得其父目录存储的文件信息entry[]，一一比对，将该文件对应的entry从中删除，并将删除该entry后的内容利用inode_block_realloc重新填充至父目录中。

最后，也调用inode_block_realloc函数，将new_size参数赋值为0。因为该函数是先删除文件原本信息再将新内容buffer根据new_size填充至文件中，因此该操作相当于将该文件从data block层面删除。之后调用write_inode_map函数将该文件从i-bmap中删除，从而达到完全删除的作用。因为这样可用inode数量加一，因此要更新superblock。代码如下：

```
int delete_file(const char *path)
{
    inode_t inode;
    inode_id_t inode_id;
    int error=find_inode_by_path(path,&inode,&inode_id);
    if(error<0)
        return error;
    //寻找父目录信息
    inode_id_t parent_inode_id;
    inode_t parent_inode;
    char child_path[FILE_NAME_LENGTH];

error=find_parent_inode_by_path(path,&parent_inode,&parent_inode_id,child_path);
    if(error<0)
        return error;
    //将其从父目录中删除
    entry_t *entry=get_buffer_from_inode(parent_inode);
    int file_number=parent_inode.size/sizeof(entry_t);
    int i;
    for(i=0;i<file_number;i++)
    {
        if(memcmp(child_path,entry[i].name,FILE_NAME_LENGTH)==0)
            break;
    }
    memmove(entry+i,entry+i+1,(file_number-i-1)*sizeof(entry_t));
    inode_block_realloc(parent_inode,parent_inode_id,entry,parent_inode.size-
sizeof(entry_t));
    free(entry);
    //将其从datablock中删除
    inode_block_realloc(inode,inode_id,NULL,0);
    //将其从inode bit map中删除
    write_inode_map(inode_id,0);
    //更新superblock
    superblock_t superblock;
    disk_read(SUPERBLOCK_ID,&superblock);
    superblock.free_inode_num++;
    disk_write(SUPERBLOCK_ID,&superblock);
    return 0;
}
```

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

要修改文件名，其本质则是更改该文件对应的entry所在的位置即可。

要将oldpath重命名为newname，首先可能会出现newname已有对应的文件，因此首先要调用delete_file函数将其删除。之后，分别找到oldpath和newname对应的父目录的inode及inode id信息。因为文件被访问和修改了，所以要先修改该文件的atime和mtime。

之后，根据old_parent_inode找到该文件对应的entry。这时会出现两种情况：两个父目录相同或不同，若两个父目录相同，则只需要修改entry的name信息并重新写回即可。若不同，则从oldpath对应的父目录中删除该entry，再将其用add_entry函数写入到newname对应的父目录下。代码如下：

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

    if(strcmp(oldpath, newname)==0) return 0;
    delete_file(newname);
```

```
    inode_t old_parent_inode,new_parent_inode;
    inode_id_t old_parent_inode_id,new_parent_inode_id;
    char old_child_name[FILE_NAME_LENGTH],new_child_name[FILE_NAME_LENGTH];
    int
error=find_parent_inode_by_path(oldpath,&old_parent_inode,&old_parent_inode_id,o
ld_child_name);
    if(error<0) return error;

error=find_parent_inode_by_path(newname,&new_parent_inode,&new_parent_inode_id,n
ew_child_name);
    if(error<0) return error;

    inode_t inode;
    inode_id_t inode_id;
    error=find_inode_by_path(oldpath,&inode,&inode_id);
    if(error<0) return error;
    inode.atime=time(NULL);
    inode.mtime=time(NULL);
    write_inode(inode_id,inode);

    entry_t *entry=get_buffer_from_inode(old_parent_inode);
    int file_number=old_parent_inode.size/sizeof(entry_t);
    int i;
    for(i=0;i<file_number;i++)
    {
        if(memcmp(old_child_name,entry[i].name,FILE_NAME_LENGTH)==0)
            break;
    }
    if(new_parent_inode_id==old_parent_inode_id)
    {

        memcpy(entry[i].name,new_child_name,FILE_NAME_LENGTH);

inode_block_realloc(old_parent_inode,old_parent_inode_id,entry,old_parent_inode.
size);
        free(entry);
        return 0;
    }
    else
    {
        entry_t new_entry=entry[i];
        memmove(entry+i,entry+i+1,(file_number-i-1)*sizeof(entry_t));

inode_block_realloc(old_parent_inode,old_parent_inode_id,entry,old_parent_inode.
size-sizeof(entry_t));
        free(entry);
        memcpy(new_entry.name,new_child_name,FILE_NAME_LENGTH);
        return add_entry(new_parent_inode,new_parent_inode_id,new_entry);
    }

    //return 0;
}
```

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

该函数只需先根据path找到文件是否存在，若文件不存在，则新建该path的文件。之后，调用inode_block_realloc函数将偏移量为offset，大小为size的内容buffer写入该文件即可。代码如下：

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

    inode_id_t inode_id;
    inode_t inode;
    int error=find_inode_by_path(path,&inode,&inode_id);
    //文件不存在
    if(error==-ENOENT)
    {
        int error=make_new_file(path,REGMODE);
        if(error<0) return 0;
        error=find_inode_by_path(path,&inode,&inode_id);
        if(error<0) return 0;
    }
    else if(error<0)    return 0;
    off_t new_size=(inode.size>(size+offset))?inode.size:(size+offset);
    char *p=get_buffer_from_inode(inode);
    p=(char*)realloc(p,new_size);
    char *p_=p+offset;
    memcpy(p_,buffer,size);
    error=inode_block_realloc(inode,inode_id,p,new_size);
    free(p);
    if(error<0)
        return 0;
    else
        return size;
}
```

int fs_truncate (const char *path, off_t size)

类似fs_write函数，首先根据path找到文件是否存在，若文件不存在，则新建该path的文件。之后调用get_buffer_from_inode函数获得文件中原本就存放的数据，再利用realloc将其大小修改为size，最后调用inode_block_realloc函数将内容写回至文件中。代码如下：

```
int fs_truncate (const char *path, off_t size)
{
    //printf("Truncate is called:%s\n",path);
    inode_id_t inode_id;
    inode_t inode;
    int error=find_inode_by_path(path,&inode,&inode_id);
    if(error==-ENOENT)
    {
        int error=make_new_file(path,REGMODE);
        if(error<0) return 0;
        error=find_inode_by_path(path,&inode,&inode_id);
        if(error<0) return 0;
    }
    if(error<0) return error;
    void *buffer=get_buffer_from_inode(inode);
    if(inode.size<size)
        buffer=(void*)realloc(buffer,size);
    error=inode_block_realloc(inode,inode_id,buffer,size);
```

```
    free(buffer);
    return error;
}
```

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

首先根据path找到文件是否存在，若文件不存在，则新建该path的文件。之后修改文件对应的inode中的三个time信息并重新写回至inode table即可。代码如下：

```
int fs_utime (const char *path, struct utimbuf *buffer)
{
    //printf("Utime is called:%s\n",path);
    inode_t inode;
    inode_id_t inode_id;
    int error=find_inode_by_path(path,&inode,&inode_id);
    if(error==-ENOENT)
    {
        int error=make_new_file(path,REGMODE);
        if(error<0) return 0;
        error=find_inode_by_path(path,&inode,&inode_id);
        if(error<0) return 0;
    }
    if(error<0) return error;
    inode.atime = buffer->actime;
    inode.mtime = buffer->modtime;
    inode.ctime = time(NULL);
    write_inode(inode_id,inode);
    return 0;
}
```

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

该函数需要获取的一些内容只需调用宏定义或者读取superblock即可。代码如下：

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

    superblock_t superblock;
    disk_read(SUPERBLOCK_ID,&superblock);
    stat->f_bsize=BLOCK_SIZE;
    stat->f_blocks=BLOCK_NUM-DATA_BLOCK_START_ID;
    stat->f_bavail=superblock.free_data_block_num;
    stat->f_bfree=superblock.free_data_block_num;
    stat->f_files=INODE_NUM;
    stat->f_ffree=superblock.free_inode_num;
    stat->f_favail=superblock.free_inode_num;
    stat->f_namemax=FILE_NAME_LENGTH;
    return 0;
}
```

部分自定义函数分析：

考虑到一些自定义函数比较简单，光从作用的描述大概就能推断出其原理，而部分自定义函数较为复杂，故在此做展开。

int inode_block_realloc(inode_t inode,inode_id_t inode_id,void* buffer,off_t new_size)

该函数的本质是将文件覆盖重写，首先删除文件中原有的内容：只需根据inode的pointer以及pointer_num的信息将对应的data block bit map置为0即可，若有indirect pointer，则也只需读取对应的block，将indirect pointer指向的block id以及该block中存储的所有指针的bit map都置为0即可，在这过程中因为剩余可用的block数量增加，因此需要修改superblock。

之后，每次寻找一个空闲的block，写入BLOCK SIZE大小的数据，更新pointer及pointer_num。若用完了所以direct pointer后还未写完，则再使用indirect pointer。期间也要更新superblock。最后，修改inode的ctime和mtime，写回至inode table中。代码如下：

```
int inode_block_realloc(inode_t inode,inode_id_t inode_id,void* buffer,off_t
new_size)
{
    if(new_size>2048*BLOCK_SIZE)
        return -ENOSPC;
    superblock_t superblock;
    disk_read(SUPERBLOCK_ID,&superblock);
    for(unsigned short i=0;i<inode.directPtr_num;i++)
    {
        write_block_map(inode.directPtr[i],0);
        superblock.free_data_block_num++;
    }
    block_id_t indirect_block_id[BLOCK_SIZE/sizeof(block_id_t)];
    disk_read(inode.indirectPtr,indirect_block_id);
    for(unsigned short i=0;i<inode.indirectPtr_num_in_block;i++)
    {
        write_block_map(indirect_block_id[i],0);
        superblock.free_data_block_num++;
    }

    inode.directPtr_num=0;
    inode.indirectPtr_num_in_block=0;
    inode.size=new_size;
    off_t tmp_size=0;
    void *tmp_block=(void*)malloc((new_size+BLOCK_SIZE-
1)/BLOCK_SIZE*BLOCK_SIZE);
    memcpy(tmp_block,buffer,new_size);
    int block_count=0;
    while(tmp_size<new_size&&inode.directPtr_num<MAX_DIRECT_PTR_NUM)
    {
        if(!find_free_block(&inode.directPtr[inode.directPtr_num]))
            return -ENOSPC;

disk_write(inode.directPtr[inode.directPtr_num],tmp_block[block_count++]);
        write_block_map(inode.directPtr[inode.directPtr_num],1);
        superblock.free_data_block_num--;
        inode.directPtr_num++;
        tmp_size+=BLOCK_SIZE;
    }
    if(tmp_size<new_size)
    {
        if(!find_free_block(&inode.indirectPtr))
            return -ENOSPC;
        block_id_t tmp_indirect_block_id[BLOCK_SIZE/sizeof(block_id_t)];
        memset(tmp_indirect_block_id,0,BLOCK_SIZE);
        while(tmp_size<new_size)
        {
```

```
if(!find_free_block(&tmp_indirect_block_id[inode.indirectPtr_num_in_block]))
                return -ENOSPC;

disk_write(tmp_indirect_block_id[inode.indirectPtr_num_in_block],tmp_block[block
_count++]);

write_block_map(tmp_indirect_block_id[inode.indirectPtr_num_in_block],1);
            superblock.free_data_block_num--;
            inode.indirectPtr_num_in_block++;
            tmp_size+=BLOCK_SIZE;
        }
        disk_write(inode.indirectPtr,tmp_indirect_block_id);
        write_block_map(inode.indirectPtr_num_in_block,1);
        superblock.free_data_block_num--;
    }
    inode.mtime=time(NULL);
    inode.ctime=time(NULL);
    write_inode(inode_id,inode);
    disk_write(SUPERBLOCK_ID,&superblock);
    free(tmp_block);
    return 0;
}
```

void *get_buffer_from_inode(inode_t inode)

该函数是根据inode中存储的pointer信息，从所指向的block中读取文件内容。只需利用malloc创建大小为inode.size的buffer，根据pointer_num和pointer依次访问每个block并将其存入buffer中即可，最后返回buffer。代码如下：

```
void *get_buffer_from_inode(inode_t inode)
{
    if(inode.size==0)
        return NULL;
    int tmp_size=0;
    void *buffer=(void*)malloc((inode.size+BLOCK_SIZE-1)/BLOCK_SIZE*BLOCK_SIZE);
    void *p=buffer;
    for(int i=0;i<inode.directPtr_num&&tmp_size<inode.size;i++)
    {
        disk_read(inode.directPtr[i],p);
        p+=BLOCK_SIZE;
        tmp_size+=BLOCK_SIZE;
    }
    block_id_t indirect_pointers[BLOCK_SIZE/sizeof(block_id_t)];
    disk_read(inode.indirectPtr,indirect_pointers);
    for(int j=0;j<inode.indirectPtr_num_in_block&&tmp_size<inode.size;j++)
    {
        disk_read(indirect_pointers[j],p);
        p+=BLOCK_SIZE;
        tmp_size+=BLOCK_SIZE;
    }
    void *ret_buffer=(void*)malloc(inode.size);
    memcpy(ret_buffer,buffer,inode.size);
    free(buffer);
    return ret_buffer;
}
```

int add_entry(inode_t parent_inode,inode_id_t parent_inode_id,entry_t entry)

该文件是在新建文件或移动文件时调用的，作用是将新的entry添加至父目录中。只需先根据父目录parent_inode获得原本entry数组，为其大小重新分配为parent_inode.size+sizeof(entry_t)后在数组末尾添加该新的entry，然后利用inode_block_realloc函数重新写回至parent_inode中即可。

```
int add_entry(inode_t parent_inode,inode_id_t parent_inode_id,entry_t entry)
{
    void *buffer=get_buffer_from_inode(parent_inode);
    off_t new_size=parent_inode.size+sizeof(entry);
    buffer=(void*)realloc(buffer,new_size);
    void *p=buffer+parent_inode.size;
    memcpy(p,(void*)&entry,sizeof(entry));
    int error=inode_block_realloc(parent_inode,parent_inode_id,buffer,new_size);
    free(buffer);
    return error;
}
```

三、实验总结

本次实验，不光让我对文件系统，特别是VSFS有了一个更深入的了解，函数中多处对于指针的操作，还让我对指针有了一个更深的掌握。并且也让我的debug能力上升了。