FSlab 实验报告

By 徐晨 2017202113

目录

一.块组织及节点结构

（一）文件信息存储结构

总的块组织结构采用 VFS 结构，即采用 SuperBlock Inodebitamp Databitmap InodeTable + Datablock 的形式。

Superblock 的信息用一个块的信息存下，具体的结构为：

```
struct statvfs {
    unsigned long  f_bsize; //块大小
    fsblkcnt_t     f_blocks;//块数量
    fsblkcnt_t     f_bfree; //空闲块数量
    fsblkcnt_t     f_bavail;//可用块数量
    fsfilcnt_t     f_files; //文件节点数
    fsfilcnt_t     f_ffree; //空闲节点数
```

fsfilcnt_t f_favail;//可用节点数

unsigned long f_namemax;

//文件名长度上限

};

直接使用 statvfs 的结构体作为 SuperBlock 的内容，这里可以用一块 4kb 的块来存储信息，所以固定 SuperBlock 的块号为 0.

然后是 bitmap 的结构：

Inodebitmap:计算由于可以支持 32768 个文件，所以 Inodebitmap 的大小为 2^15 个 bit，即 2^12 = 4kb，只用一个块即可以存下。而 Databitmap 大小要参照整个块的大小 2^16 来计算，每个块可以支持 2^15 个大小，所以可以放 2 个 DataBitmap，所以前 4 个块的结构如图所示：

SuperBlock	Inodebitmap	Databitmap	Databitmap

图 前 4 个块的分布

Inode 的信息：

```
typedef struct{
    mode_t    mode;
    nlink_t nlink;
    uid_t     uid;
    gid_t     gid;
    off_t     size;
    time_t atime;
    time_t mtime;
    time_t ctime;
    size_t inode_number;
    int direct_pointer[DP_NUMBER];
    int indirect_pointer[IP_NUMBER];
  }my_stat;
```

比较重要的结构是：使用了 14 个直接指针，和两个间接指针。每一个间接指针指向的块可以存下 4096/4 = 1024 个块即支持的大小为 2^(10+4) = 4M

即每个 inode 支持的文件大小为 14*4kb+4M*2 = 56kb+8M.

这样的结构：Inode 大小有 128B，总共有 2^15 个 inode，所以总共有 2^22B = 4M

需要的块大小为 2^22/2^12 = 1024 个块。

Inode	Inode	…	…	…	Inode	Inode

文件一个有 256M = 2^28/2^12 = 2^16 = 65536 个块

所以除去上述的块，data 占有的块有 65536-1024-4 = 64508 块，但是为了计算方便，前 1028 个块置为 1，表示占用，后面的块作为 datablock 来使用。

真实的利用空间为 64508/65536 = 98.43%,真实的空间为 251.98M。

（二）datablock 存储结构

Datablock 分为三种情况：

（1） 普通文件拥有的块：4096 个 char 型字符来存储

（2） 目录文件拥有的块：

每一个目录文件下的存储的格式为：

```
typedef struct{
char filename[24];
```

```
        int filetype;
        int inode_number;
            }dir_files;
```

前 24 个字符为文件名，后面有两个属性：文件类型和 inode 编号，这样的结构一共有 32 个字节大小，每一个块里可以存 4096/32 = 128 个文件，这里只用到了 14 个直接指针，所以**每个目录只能支持 14*128 = 1792 个文件及目录。**平均深度最深控制为 512/12 = 43 的目录深度，最低的目录深度为 512/24 = 21 的目录深度。

（3） 指针块：间接指针用一个 int 型的数字来代表指针指向下一个快，结构为：int pointer_nextblock[MAX_DATA_IN_BLOCK/sizeof(int)];

把上述三种结构整合在一起的方式是使用 union，结构如下：

```
typedef union
{
    int pointer_nextblock[MAX_DATA_IN_BLOCK/sizeof(int)];
    char file_data[MAX_DATA_IN_BLOCK];        // And all the rest of the space in the block can be used for actual data storage
    dir_data dir;
  }disk_block;
```

以不同的方式去引用这些块，每一个 disk_block 的大小为 4096 是一个块的大小，用以代表每一个块存储的信息。

二．实现函数的重要细节

（一）辅助函数的实现

1.bitmap 的相关操作

int find_empty_Ibm()　找到一个空闲的 inodenuber.返回的是 inode 编号

void set_Ibm (int inode_number)　　把相应编号的 inode 置为 1，表示被使用

void empty_Ibm (int inode_number) 把相应编号的 inode 置为 0，表示未使用

int find_empty_Dbm()找到一个空闲的 datablock 编号.返回的是 inode 编号

void set_Dbm (int inode_number) 把相应编号的 datablock 置为 1，表示被使用

void empty_Dbm (int inode_number) 把相应编号的 datablock 置为 0，表示未使用

实现的细节是用位操作：

```
for (int byte_position = 0; byte_position < disk_size; byte_position++)
{
    for (int bit_position = 0; bit_position < _bitsize; bit_position++)
    {
        if (!(buff[byte_position] & (0x1 << bit_position)))
        {
            return byte_position * _bitsize + bit_position;
        }
    }
}
```

以 find_empty_Ibm()为例，利用对应的 inodenumber 找到对应的位置，具体的转化公式为 **inodenumber/8 & (0x1<<(inodenumber%8))**

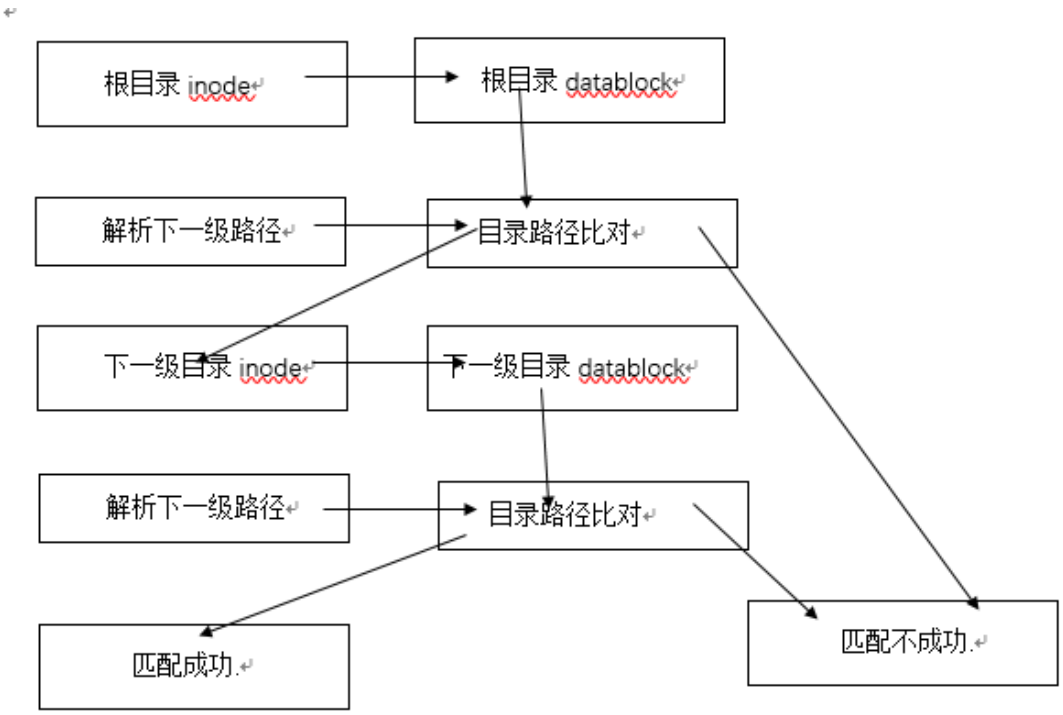
然后对相应的位进行操作。

2.路径解析的函数

A．inode_t match_path(const char*path)

该函数是把该路径的 inode 编号返回，如果不成功就返回-1.实现的思路是：

一层一层地寻找对应地 inode->datablock



解析路径：

```
while (*p != '\0' && *p != '/')
{
    *index = *p;
    index++;
    p++;
}
*index = '\0';
```

找到结束或者找到’ /’ 代表找到一层路径。

目录路径匹配：

首先判断需要几个块，然后把每一个块所在的 block 里的数据整块取出，然后进行遍历，注意剩余块的遍历，核心代码：

```
for (int i = 0; i < block_numer; i++)
{
    disk_block dir_search;
    disk_read(inode_data[inode_blockoffset].direct_pointer[i], &dir_search);
    for (int j = 0; j < 128; j++)
    {
        if (strcmp(dir_search.dir.files[j].filename, dir_name) == 0)
        {
            return 0;
        }
    }

}
if (block_left != 0)
{
    printf("get left!!!\n");
    fflush(stdout);
    disk_block dir_search;
    disk_read(inode_data[inode_blockoffset].direct_pointer[block_numer], &dir_search);
    for (int j = 0; j < block_left; j++)
    {
        printf("dir_file:name:%s\n", dir_search.dir.files[j].filename);
        if (strcmp(dir_search.dir.files[j].filename, dir_name) == 0)
        {
            printf("match inode:%d name:%s!!\n", dir_search.dir.files[j].inode_number, dir_search.dir.files[j].filename);



            return dir_search.dir.files[j].inode_number;
        }
    }
}
```

B．int collect_parent_now_path(const char *path, char *parent_path, char *file_name)

该函数传入的参数有完成的路径，待解析的父级目录的指针，待解析的文件名。

实现思路是从后往前扫描，扫的第一个/的前面都是父级目录，再把倒序扫描的字符串倒序，从而得到文件名。核心代码：

```
while (*path_pointer != '/')
{
    temp_filename[index++] = *path_pointer;
    path_pointer--;
}
//temp_filename[index] = '\0';
strcpy(file_name, strrev(temp_filename));
strncpy(parent_path, path, pathlen - index);

parent_path[pathlen - index] = '\0';
```

（二）mkfs（）初始化的实现

这个函数的主要功能有：

A. 初始化 superblock，每个块大小为 4096，文件名最长大小为 24，文件最大数为 32768，可使用的块为 64508 个块。

B. 初始化 bitmap,前 1028 个块被使用，置为 1

C. 写根目录的相关信息，

```
root_stat[0].mode = DIRMODE;
root_stat[0].nlink = 1;
root_stat[0].uid = getuid();
root_stat[0].gid = getgid();
root_stat[0].size = 0;
root_stat[0].atime = time(NULL);
root_stat[0].mtime = time(NULL);
root_stat[0].ctime = time(NULL);
root_stat[0].inode_number = root_inode_number;
memset(root_stat->direct_pointer, 0, sizeof(root_stat->direct_pointer));
memset(root_stat->indirect_pointer, 0, sizeof(root_stat->indirect_pointer));
```

（三）fs_getarr 函数的实现

思路：利用 match_path 函数找到对应的 inode，然后把相应的信息取出，并填入相应的结构体即可。这里的实现比较简单，需要注意的一点是找到的 inode 编号需要找到对应的块号，**块号公式为 blockid = inodeId/32 + 4。**这里的+4 是因为这里设置的 inode 编号从 0 开始，但是前面有 4 个块被占用了，所以需要加 4。在块里对应的偏移量为 inodeid%32.

为了方便，这里定义了一组宏：

```
#define _blockid(inode_number) (inode_number/32)
#define _BID(inode_number) (inode_number/32 +4)
#define _blockoffset(inode_number) (inode_number%32)
```

（四）fs_readdir 函数的实现

这个函数实现的思路是首先找到该目录的 inode 编号，这里通过上述的函数比较容易实现。然后就是需要计算出这个目录所包含几个文件，这里**文件数目的计算公式为 file_counter = size/32 + !!（size%32)**，因为每个文件占用 32 个字节，还要考虑剩余块的情况。

然后就是把每一个块取出，按顺序遍历，把文件名用 filler 函数填入 buffer 即可。核心代码如图：

```
for (int i = 0; i < pointer_count; i++)
{
    //char blockdata[4048];
    disk_block block_dir_data;
    disk_read(dir_data[_blockoffset(dir_inode_numer)].direct_pointer[i], &block_dir_data);
    for (int j = 0; j < 128; j++)
    {

        filler(buffer, block_dir_data.dir.files[j].filename, NULL, 0);
    }

}
if (pointer_left != 0)
{
    disk_block block_dir_data;
    disk_read(dir_data[_blockoffset(dir_inode_numer)].direct_pointer[pointer_count], &block_dir_data);
    for (int j = 0; j < pointer_left; j++)
    {
        //disk_read(dir_data[_blockoffset(dir_inode_numer)].direct_pointer[j], &dir_data);
        filler(buffer, block_dir_data.dir.files[j].filename, NULL, 0);
    }
}
```

结果测试：

```
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ ls
test  test2  test4
```

(五）fs_read 函数的实现

总的核心思路是首先把能读出来的 size 计算好。具体的计算公式为：

If（偏移量+要读的大小 > 文件的大小）读出的 size 就只能是文件的大小减去偏移量。

Else 读出的 size 就是传入的 size 大小

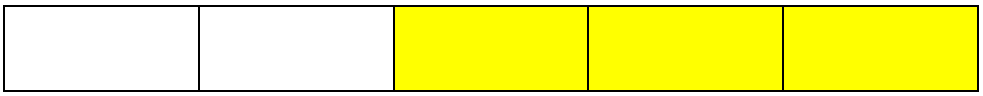
然后得到从块 a->块 b 的读写。

其中 a 的编号的计算公式为：offset/4096 + !!(offset%4096)

B 编号的计算公式为：（offset+size（读出的））/4096 + !!(offset+size（读出的））%4096

然后把 a->b 块的字符依次取出，放入 buffer 中，这里块的读写分为三种情况：

（1） 开头的块 a：



开头的块可能会出现偏移的情况，所以只用把偏移量后面的块读出就好

（2）中间的块：

直接整块读写

（3）最后的块 b：

最后的块可能会出现不够读的情况，所以只用把剩余的数据读出就好。

然后读的块又分为两种块：

（1） 直接指针指向的块

这类块直接从磁盘中读出直接指针所指的块即可。

```
    char data[disk_size];
    disk_read(inode_info[blockoffset].direct_pointer[begin_block_number], data);
    memcpy(buffer, data + (offset % disk_size), read_size);
    return read_size;
```

（2） 间接指针指向的块

这里首先要把间接指针所指的块读出，然后再把对应偏移量的指针读出，再从磁盘中读出对应的数据。

```
    char data[disk_size];
    disk_block pointer_data[0];
    disk_read(inode_info[blockoffset].indirect_pointer[0], pointer_data);
    disk_read(pointer_data[0].pointer_nextblock[(begin_block_number - 14)], data);
    memcpy(buffer, data + (offset % disk_size), read_size);
```

总的函数的核心代码如图（以中间块的读为例）：

```
for (int i = begin_block_number+1; i <= end_block_number-1; i++)
{

    if (i < 14) {
        char data[disk_size];

        disk_read(inode_info[blockoffset].direct_pointer[i], data);
        memcpy(Buff, data , disk_size );
        //return size;
    }
    else if ((i - 14) < 1024)
    {
        char data[disk_size];
        disk_block pointer_data[0];
        disk_read(inode_info[blockoffset].indirect_pointer[0], pointer_data);
        disk_read(pointer_data[0].pointer_nextblock[(i - 14)], data);
        memcpy(Buff, data, disk_size);
        //return size;

    }
    else if ((i - 14) < 2 * 1024)
    {

        char data[disk_size];
        disk_block pointer_data[1];
        disk_read(inode_info[blockoffset].indirect_pointer[1], pointer_data);
        disk_read(pointer_data[0].pointer_nextblock[(i - 14 -1024)], data);
        memcpy(Buff, data, disk_size);
        //return size;
    }
    Buff += disk_size;
    //offset_i += 1;
}
```

<error-type> pointer_data

结果测试：

```
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ echo 2017202113 >
 test3
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ ls
test  test2  test3
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ cat test3
2017202113
```

（六）fs_mknod 和 mkdir 的实现

这两个函数分为三步实现：

（1） 解析路径，得到文件名和父路径。该操作的实现上述提过，就不再赘述了。

（2） 找到一个 inode，把对应文件的信息写进去，并且把对应对应的 inodebitmap 置空。

（3） 把更新父级目录的 inode 信息，并且把创建的文件信息写入父目录的 datablock。具体写入 block 的思路可见下面的写操作。该步操作的核心代码如图：

```
disk_block dirfile[1];
disk_read(parent_inode_info[_blockoffset(parent_inode_number)].direct_pointer[dp_number], dirfile);
strcpy(dirfile[0].dir.files[dp_left].filename, filename);
dirfile[0].dir.files[dp_left].filetype = REGMODE;
dirfile[0].dir.files[dp_left].inode_number = inode_numer;
parent_inode_info[_blockoffset(parent_inode_number)].size += 32;
disk_write(parent_inode_info[_blockoffset(parent_inode_number)].direct_pointer[dp_number], dirfile);
disk_write(_blockid(parent_inode_number)+4, parent_inode_info);
```

（七）fs_rmdir 和 fs_unlink 的实现

这两个函数的实现思路分为四步来实现：

（1） 解析路径，得到文件名和父路径。该操作的实现上述提过，就不再赘述了。

（2） 更改该父目录 inode 的相关信息

（3） 释放该文件的 inode，以及释放已经有的 datablock

要删除的块的数量就是为 size/4096+!!(size%4096)

由于目录文件是空，所以不需要考虑 datablock 的情况，若是普通文件，则释放块分为两种情况

A. 直接指针所指的块，就直接释放直接指针所指的块。

B. 间接指针所指的块，如果是间接指针所指的最后一个块，则需要释放考虑释放间接指针所在的块。

该部分的核心代码(以释放二级指针为例)为：

```
for (int i = 0; i < 14; i++)
{
    empty_Dbm(inode_stat[now_inode_offset].direct_pointer[i]);
}
disk_block indirect_pointer_data[1];
disk_read(inode_stat[now_inode_offset].indirect_pointer[0], indirect_pointer_data);
for (int j = 0; j < 1024; j++)
{
    empty_Dbm(indirect_pointer_data[0].pointer_nextblock[j]);
    empty_Dbm(indirect_pointer_data[0]);
}
disk_read(inode_stat[now_inode_offset].indirect_pointer[1], indirect_pointer_data);
for (int j = 0; j < (inode_datablock_count - 14- 1024); j++)
{
    empty_Dbm(indirect_pointer_data[1].pointer_nextblock[j]);
    empty_Dbm(indirect_pointer_data[1]);
}
```

（4） 把父目录的 datablock 的该文件删除，这里分为两种情况处理。

A. 如果该文件是该目录的最后一个文件，则直接删除即可，若该文件是该块的最后一个文件，那么也应该释放该块。

B. 如果该文件是目录的中间的文件，为了避免大规模的数据搬移，则考虑把最后一个文件搬移到空缺的位置，若要搬移的文件是该块的最后一个文件，那么也应该释放该块。

文件	文件
待删除文件	最后一个文件
文件	

该部分的核心代码为：

```
if (dir_data2[0].dir.files[j].inode_number == inode_number)
{//找到对应的inode_number 需要把它删除
    if (inode_number != last_file_inode)
    {//最后一个文件
        memcpy(&(dir_data2[0].dir.files[j]), &last_files, 32);
        disk_write(parent_stat[parent_blockoffset].direct_pointer[datablock_number], dir_data2);
        return 0;
    }
}
```

（八）fs_rename 操作

该函数的核心思想是解析路径，然后找到该父目录的 inode，以及对应的 datalobck,然后进行匹配，匹配完成后，就把原来的父目录 datablock 里的文件名修改再写回即可。核心代码：

```
disk_block blockdata2[1];
disk_read(inode_stat[offset].direct_pointer[block_numer], blockdata2);
for (int j = 0; j < block_left; j++)
{
    if (strcmp(blockdata2[0].dir.files[j].filename, old_filename) == 0)
    {
        printf("\n\n\n rename succeed:old%s new%s!\n", old_filename,new_filename);
        //memcpy(blockdata[0].dir.files[j].filename, new_filename, 24);
        memset(blockdata2[0].dir.files[j].filename, 0, 24);
        strcpy(blockdata2[0].dir.files[j].filename, new_filename);
        disk_write(inode_stat[offset].direct_pointer[block_numer], blockdata2);
        return 0;
    }
}
```

（九）fs_wrte 操作

该函数的核心思想可以分为三步：

（1） 解析路径，得到父目录和该文件信息

（2） 修改父目录的 inode 以及该文件的 inode 信息

（3） 写入文件信息

首先数据的写入都是整块的写入，并且存在部分并发的情况，所以这里的每一次写都要新找一块 datablock、并且把相应的 databitmap 修改，找对应的 block 的话公式为 **blockid = offset/4096,**这里的 datablock 可以分为两种情况：

A．如果是直接指针指向的快

就直接分配一个空闲的数据块就可以，这种情况的 blockid<14

B．如果是第一次分配间接指针所指向的块的话，需要分配一块间接指针指向的块。这种情况的 blockid>14 && blockid<(14+2048)

该部分的核心代码如图所示：

```
if (nowsize % disk_size != 0)
{//同时这里也是处理追加写的方式
    //now_block_number += 1;

    int data_left_number = nowsize % disk_size;

    if (now_block_number < 14)
    {//这里只需要用到直接指针就好
        char wait2write_data[disk_size];
        disk_read(inode_info[blockoffset].direct_pointer[now_block_number], wait2write_data);
        memcpy(wait2write_data + data_left_number, buffer, size);
        disk_write(inode_info[blockoffset].direct_pointer[now_block_number], wait2write_data);
        disk_write(_blockid(inode_number) + 4, inode_info);
        return size;
    }
    else if (now_block_number < 14 + 1024)
    {//这里用到了第一层间接指针
        char wait2write_data[disk_size];
        int now_indirect_block_number = now_block_number - 14;
        disk_block indirect_pointer_data[1];
        disk_read(inode_info[blockoffset].indirect_pointer[0], indirect_pointer_data);
        disk_read(indirect_pointer_data[0].pointer_nextblock[now_indirect_block_number], wait2write_data);
        memcpy(wait2write_data + data_left_number, buffer, size);
        disk_write(indirect_pointer_data[0].pointer_nextblock[now_indirect_block_number], wait2write_data);
        disk_write(_blockid(inode_number) + 4, inode_info);
        return size;

    }
    else if (now_block_number < 14 + 2*1024)
    {
        char wait2write_data[disk_size];
        int now_indirect_block_number = now_block_number - 14- 1024;
        disk_block indirect_pointer_data[1];
        disk_read(inode_info[blockoffset].indirect_pointer[1], indirect_pointer_data);
        disk_read(indirect_pointer_data[0].pointer_nextblock[now_indirect_block_number], wait2write_data);
        memcpy(wait2write_data + data_left_number, buffer, size);
        disk_write(indirect_pointer_data[1].pointer_nextblock[now_indirect_block_number], wait2write_data);
        disk_write(_blockid(inode_number) + 4, inode_info);
        return size;
    }
```

（十）fs_truncate 操作

该函数的实现，可以分为三种情况：

（1） 如果 truncate 函数两次的大小在同一个块上，则只需修改 inode 的 size 即可。

（2） 如果新的大小小于原来的大小，则需要释放 block，具体实现思路和 unlink 比较像，需要注意的一点是不是释放到 0 大小，而是释放到指定的大小。

（3） 如果新的大小大于原来的大小，则需要添加 block，具体实现思路和 mknod 比较像，需要注意的一点是不是从 0 大小开始分配块，而是从原来的大小开始分配块。

（十一）fs_utime 和 fs_statfs 操作

Fs_utime 函数的实现比较简单，就是解析路径得到 inodenumber，修改时间再写回即可。

Fs_statfs 函数的思路也比较简单，从 superblock 中读出以下信息，扫描 bitmap，然后把可用节点，和可用文件数修改再写回即可。

三．遇到的问题及解决思路

（一）出现的一些高并发情况

在对大文件进行大量读写时，fuse 系统常常会把大文件的读写拆分为几个多次的读写来提高并发程度，但是如果在这个过程中进行大量的读写则会造成并发错误。

一个解决的思路是：在 fs_read 或者其他操作最后一个一次操作时才把 inode 休息修改了写回，这样可以避免一些高并发情况。

（二）块数据的清理过程

在块数据的清理过程中，如果要把数据块的信息都清理了，其实是一个比较费时间的过程。

而解决的办法就是直接修改 size 而不用去清理原来 datablock 里的数据，只用把 bitmap 值修改了置为空，那么该数据块就不会被使用，这样可以去节省大量清理的时间。

（三）块粒度的问题

由于每次修改块里的信息都要把整块的信息读出再写回，造成大量的 IO，这里可采用一种解决办法就是设一个 buffer，读出的信息不着急写回，等所有操作结束以后再写回，能提高效率。

四．反思与总结

（一）文件系统效率的一点思考

由于文件的磁头在连续访问比较省时间，所以可以考虑在组织 bitmap 的时候采用树或者其他组织结构，尽量保证数据是连续存放的，可以提高存储效率。

每次找到一个文件的信息，需要从根节点开始遍历，如果文件数深度比较深，则会造成大量的读写放大，可以考虑的一种的方式为每次进一层目录就把该目录信息的 inode 缓存到一个 cache 里，提高文件系统的利用率。

（二）对设计系统的总结

设计一个系统需要考虑的东西比较多，包括系统实现的鲁棒性，以及系统实现出现的一些高并发等情况的处理。同时也要兼顾系统的效率与质量，所以要要在保证系统正确运行的情况下，去尽量兼顾性能，同时也需要做一些 trade-off.

最后放一个所有操作的截图：

```
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout$ cd mnt
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ mkdir test
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ mkdir test2
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ echo 2017202113 >
 test3
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ ls
test  test2  test3
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ cat test3
2017202113
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ echo xc >> test3
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ cat test3
2017202113
xc
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ mv test3 test4
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ ls
test  test2  test4
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ cat test4
2017202113
xc
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ rm test2
rm: cannot remove 'test2': Is a directory
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ rm -r test2
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ ls
test  test4
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt$ cd test
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt/test$ echo 2017202113 > test5
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt/test$ cat test5
2017202113
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt/test$ cp test5 test6
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt/test$ cat test6
2017202113
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt/test$ rm test6
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt/test$ ls
test5
2017202113@VM-0-17-ubuntu:~/fslab-handout15/fslab-handout/mnt/test$
```

大文件：

```
-rw-r--r-- 1 2017202113 2017202113 4810611 Jun  1 16:37 4698kb_test.txt
drwxr-xr-x 1 2017202113 2017202113      32 Jun  1 16:32 test
-rw-r--r-- 1 2017202113 2017202113      14 Jun  1 16:31 test4
```