**操作系统课程设计-UNIX文件系统**

****

**专业：计算机科学与技术**

**姓名： 杨晨**

**学号：1650917**

**授课教师： 方钰**

一、需求分析 3

1.1 程序任务 3

1.2 输入、输出形式 3

1.3 程序功能 3

二、概要设计 4

2.1 任务的分解 4

2.2数据类型的定义 5

2.3 主程序流程 16

三、详细设计 16

3.1 重点函数说明 16

3.2 类及函数调用关系 26

四、调试分析 26

4.1 测试过程及测试数据 26

4.2 遇到的问题及解决方案 27

五、用户使用说明 28

5.1 开发平台和编译说明 28

5.2用户登录说明 28

5.3 用户操作命令说明 28

六、运行结果分析 29

七、实验总结 32

7.1 综合实验过程的收获 32

7.2 遇到问题及解决问题过程的思考 32

7.3 综合实验中对课程的认识 33

7.4 不足及改进 33

八、参考文献 33

# 一、需求分析

## 1.1 程序任务

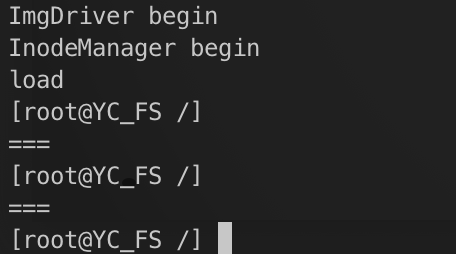
使用一个普通的大文件模拟UNIX V6++的一个文件卷，即把一个大文件当成一张磁盘来使用。这张磁盘中，以块为单位来储存信息，每个块大小为 512 字节。



定义了磁盘的文件结构、SuperBlock 结构、INode 节点结构、INode 节点分配和回收算法、物理块的分配和回收、目录和文件索引等等，并实现了缓冲区及多用户登录。

## 1.2 输入、输出形式

本次课程设计的输入输出形式使用命令行方式，基本形式如下：



根据输入指令（cd，mkdir，fcreat，format等）输出不同结果（ls，print，mkdir等），具体形式见后文。

## 1.3 程序功能

本程序实现的主要功能如下：

（1）实现磁盘文件结构

* 定义自己的磁盘文件结构
* SuperBlock结构
* 磁盘Inode节点结构，包括：索引结构，及：逻辑块号到物理块号的映射
* 磁盘Inode节点的分配与回收算法设计与实现
* 文件数据区的分配与回收算法设计与实现

（2）实现文件目录结构

* 目录文件结构
* 目录检索算法的设计与实现

（3）文件打开结构

（4）磁盘高速缓存

（5）文件操作接口

* 已实现的文件操作接口包括：format(格式化磁盘)，ls(列出当前目录下文件)，cd(进入文件夹)，mkdir(创建文件夹)，fcreat(创建文件)，fopen(打开文件)，fclose(关闭文件)，fread(从文件中读)，fwrite(向文件中写)，flseek(设置文件读写指针)，fdelete(删除文件)，movein(从外部导入文件)，print(打印Inode、Block等内存及磁盘信息)

# 二、概要设计

## 2.1 任务的分解

首先磁盘设定为65536个物理块，整个磁盘可以分成三个区域，第一部分为SuperBlock区域，为前两个物理块(0-1024字节)，记录Inode节点及物理块Block的分配和使用情况，第二个区域为Inode节点区，为第3块至第18块，每个Inode对应一个文件(夹)，记录文件的大小、索引情况，剩余区域盘块用于记录文件内容。

磁盘读写由ImgDriver类负责，ImgDriver类负责向物理空间中进行读写。

SuperBlock的管理由SuperBlockManager这个类进行管理。

Inode区域读写由InodeManager类进行管理。

从磁盘读写文件均通过缓冲区Buffer进行，由BufferManager进行管理，提供高速缓存功能，使用该类的接口进行文件操作。

用户各项操作及文件操作通过Operator类进行处理。

## 2.2数据类型的定义

2.2.1 SuperBlock:

文件控制超级块，SuperBlock的定义如下

1. **class** SuperBlock
2. {
3. **public**:
4. **int** s\_isize;            // inode总个数
5. **char** s\_ibitmap[IBITMAP\_SIZE];// 记录 inode 分配情况 128字节 128\*8位
6. **int** s\_ilock;
8. **int** s\_fsize;            // 盘块总数
9. **int** s\_nfree;            // 直接管理的空闲盘块数
10. **int** s\_free[100];        // 直接管理的空闲盘块
11. **int** s\_flock;
13. **int** s\_dirty;
14. **char** padding[472];      //填充至1024字节
15. **public**:
16. SuperBlock();
17. ~SuperBlock();
19. **void** printSuperBlock();
20. **void** printFree();
21. };

2.2.2 SuperBlockManager:

这个类提供了对SuperBlock的管理，包括内存Inode和硬盘Block的分配回收和管理。

1. **class** SuperBlockManager
2. {
3. **private**:
4. ImgDriver\* ID;
5. SuperBlock\* SB;
6. **public**:
7. SuperBlockManager();
8. ~SuperBlockManager();
9. **void** load();
10. **void** save();
11. /\*
12. \* 分配一个Inode节点
13. \*/
14. **int** allocInode();
15. /\*
16. \* 释放一个Inode节点
17. \*/
18. **void** freeInode(**int** no);
19. /\*
20. \* 分配一个Block节点
21. \*/
22. **int** allocBlock();
23. /\*
24. \* 分配一个Block节点
25. \*/
26. **void** freeBlock(**int** no);
27. };

2.2.3 Inode:

包括两个类DiskInode和Inode，DiskInode是在硬盘中保存的类，Inode是读入内存后保存节点的类，二者有一定区别。

1. **class** DiskInode{
2. **public**:
3. unsigned **int** d\_mode;    //文件工作方式
5. **int** d\_count;            //引用计数
6. **int** d\_nlink;            //文件联结计数，文件在目录树中不同路径的数量
8. **short** d\_uid;            //用户编号
9. **short** d\_gid;            //用户组编号
11. **int** d\_size;             // 文件大小
12. **int** d\_addr[10];         //逻辑块号和物理块号转换
14. **int** d\_atime;
15. **public**:
16. DiskInode();
17. ~DiskInode();
18. };
20. **class** Inode{
21. **public**:
22. **enum** INodeFlags{
23. ILOCK = 0x1,        /\* 索引节点上锁 \*/
24. IUPD  = 0x2,        /\* 内存inode被修改过，需要更新相应外存inode \*/
25. IACC  = 0x4,        /\* 内存inode被访问过，需要修改最近一次访问时间 \*/
26. IFDIR = 0x4000,     /\* 文件类型：目录文件 \*/
27. ILARG = 0x1000      /\* 文件长度类型：大型或巨型文件 \*/
28. };
30. unsigned **int** i\_mode;    //文件工作方式
32. **int** i\_count;            //引用计数
33. **int** i\_nlink;            //文件联结计数，文件在目录树中不同路径的数量
35. **short** i\_uid;            //用户编号
36. **short** i\_gid;            //用户组编号
38. **int** i\_size;             // 文件大小
39. **int** i\_addr[10];         //逻辑块号和物理块号转换
41. **int** i\_atime;
43. **int** i\_number;           //块编号
45. **public**:
46. Inode();
47. ~Inode();
48. /\*
49. \* 根据盘块 i\_addr 读取文件数据
50. \*/
51. **void** ReadI();
52. /\*
53. \* 根据 i\_addr 写入文件数据
54. \*/
55. **void** WriteI();
56. **void** printInode();
57. };

2.2.4 InodeManager:

这个类提供了对Inode的操作，主要包括节点的分配回收及写回，从硬盘中读取对应节点等操作。

1. **class** InodeManager{
2. **private**:
3. set<Inode\*> Iset;
4. ImgDriver\* ID;
6. **public**:
7. InodeManager();
8. ~InodeManager();
9. **void** saveAll();
10. /\*
11. \* 从磁盘中读一个 Inode
12. \*/
13. Inode\* readFromDisk(**int** no);
14. /\*
15. \* 添加一个节点
16. \*/
17. **void** addInode(Inode\* inode);
18. /\*
19. 获取一个空闲Inode
20. Inode getNewInode();
21. \*/
22. /\*
23. 向磁盘写一个 Inode
24. \*/
25. **void** writeInode(Inode\* inode);
27. **void** printInode(**int** no);
28. };

2.2.5 ImgDriver:

ImgDriver直接对物理磁盘进行读写，向主程序提供物理读写接口。

1. **class** ImgDriver
2. {
3. **private**:
4. fstream fs;
5. **static** **const** **char**\* DISK\_FILE\_NAME;
6. **public**:
7. ImgDriver();
8. ~ImgDriver();
9. //写操作
10. **bool** write(**const** **char**\* buf, **const** **int** buf\_size, **const** **int** offset);
11. //读操作
12. **bool** read(**char**\* buf, **const** **int** buf\_size, **const** **int** offset);
13. };

2.2.6 Buffer:

Block为物理块，buffer为内存高速缓存，由100块block组成，与文件内容进行交互时通过高速缓存buffer进行。

1. **struct** folder\_node
2. {
3. **char** file\_name[28];
4. **int** inode\_number;
5. };
7. **class** Block{
8. **public**:
9. /\* flags中标志位 \*/
10. **enum** BlockFlag {
11. B\_WRITE = 0x1,      /\* 写操作。将缓存中的信息写到硬盘上去 \*/
12. B\_READ = 0x2,       /\* 读操作。从盘读取信息到缓存中 \*/
13. B\_DONE = 0x4,       /\* I/O操作结束 \*/
14. B\_ERROR = 0x8,      /\* I/O因出错而终止 \*/
15. B\_BUSY = 0x10,      /\* 相应缓存正在使用中 \*/
16. B\_WANTED = 0x20,    /\* 有进程正在等待使用该buf管理的资源，清B\_BUSY标志时，要唤醒这种进程 \*/
17. B\_ASYNC = 0x40,     /\* 异步I/O，不需要等待其结束 \*/
18. B\_DELWRI = 0x80     /\* 延迟写，在相应缓存要移做他用时，再将其内容写到相应块设备上 \*/
19. };
20. **char** content[512];
21. **bool** is\_folder;
22. **int** b\_flag;
23. **int** b\_blkno;
24. **bool** b\_use;
25. **bool** is\_index;
26. **public**:
27. Block();
28. ~Block();
29. **void** clear();
30. /\*
31. \* 由content转换为 folder list
32. \*/
33. folder\_node\* getFolderList();
34. /\*
35. \*由content转换为 index list
36. \*/
37. **int**\* getIndexList();
38. **void** printBlock();
39. };
40. **class** Buffer{
41. **public**:
42. Block\* b\_blocks;
43. **public**:
44. Buffer();
45. ~Buffer();
46. **void** clear();
47. };

2.2.7 BufferManager:

这个类提供对于buffer的管理，包括分配回收，读写文件块等操作。

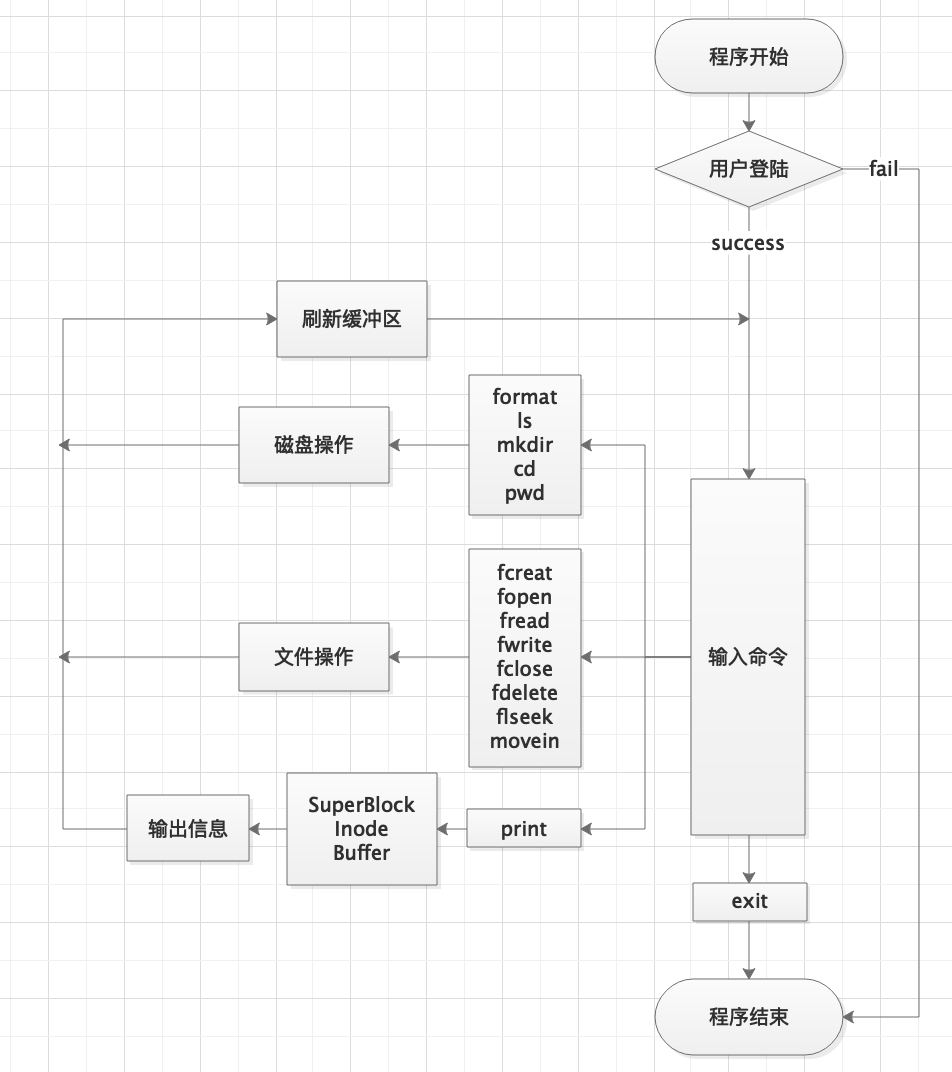
1. **class** BufferManager
2. {
3. **private**:
4. ImgDriver\* ID;
5. SuperBlock\* SB;
6. SuperBlockManager\* SBM;
7. Buffer\* BF;
8. **public**:
9. BufferManager();
10. ~BufferManager();
11. **void** clear();
12. /\*
13. \* 找到buffer中一个空闲的Block
14. \*/
15. **int** allocBlock();
16. /\*
17. \* 释放一个Block
18. \*/
19. **void** freeBlock(**int** no);
20. /\*
21. \* 从buffer中读一个block，如果读不到，则从硬盘读入
22. \*/
23. **int** readBlock(**int** d\_blkno);
24. /\*
25. \* 写入一个block
26. \*/
27. **void** writeBlock(**int** no, Block\* block);
28. /\*
29. \* 读取文件中一个 Block 输入块号 t 和 Inode
30. \*/
31. Block\* readFileBlock(Inode \*f\_inode, **int** t);
32. **void** setNewFileBlock(Inode \*f\_inode, **int** blkno);
33. **void** writeFileBlock(Inode \*f\_inode, Block\*new\_block, **int** t);
34. /\*
35. \* 刷新buffer，将延迟写的缓存Block写入磁盘
36. \*/
37. **void** flushBuffer();
38. **void** printBuffer(**int** no);
39. **void** printBlock(**int** no);
40. };

2.2.8 Operator:

这个类提供一切对于用户使用的接口，包括用户登录，读取用户输入命令，输出对应信息。

1. **class** Operator{
2. **public**:
3. ImgDriver\* ID;
4. SuperBlock\* SB;
5. SuperBlockManager\* SBM;
6. InodeManager\* IM;
7. BufferManager\* BM;
8. **int** user\_id;
9. string users\_name;
10. **int** current\_inode\_id;
11. stack<**int**> CL;                  //目录编号
12. stack<string> CLN;              //目录名称
14. //以下是文件处理的内容
15. **bool** f\_isopen;              //文件是否打开
16. string f\_name;        //打开的文件名称
17. Inode\* f\_inode;             //打开的文件的Inode
18. **int** f\_seek;                  //偏移量指针
19. **public**:
20. Operator();
21. ~Operator();
22. /\*
23. \* 用户登陆
24. \*/
25. **void** login();
26. /\*
27. \* 初始化
28. \*/
29. **void** init();
30. /\*
31. \* 接受用户指令
32. \*/
33. **void** receiveOrder();
34. /\*
35. \* 格式化
36. \*/
37. **void** format();
38. /\*
39. \* 列出文件列表
40. \*/
41. **void** ls();
42. /\*
43. \* 创建文件夹
44. \*/
45. **void** mkdir(string folder\_name);
46. /\*
47. \* 进入文件夹
48. \*/
49. **void** cd(string folder\_name);
50. /\*
51. \* 将外部文件拷贝进内部
52. \*/
53. **void** movein(string out\_file\_name, string in\_file\_name);
54. /\*
55. \* 创建文件
56. \*/
57. **void** fcreate(string folder\_name);
58. /\*
59. \* 打开文件
60. \*/
61. **void** fopen(string folder\_name);
62. /\*
63. \* 关闭文件
64. \*/
65. **void** fclose();
66. /\*
67. \* 写入文件内容
68. \*/
69. **void** fwrite(**const** **char**\* buffer, **const** **int** size);
70. /\*
71. \* 读文件内容
72. \*/
73. **void** fread(**char**\* buffer, **const** **int** size);
74. /\*
75. \* 设置文件读写指针位置
76. \*/
77. **bool** flseek(**int** pos);
78. /\*
79. \* 删除文件
80. \*/
81. **void** fdelete(string file\_name);
82. **void** pwd();
83. **void** flushAll();
84. **void** printFileInformation();
85. };

## 2.3 主程序流程



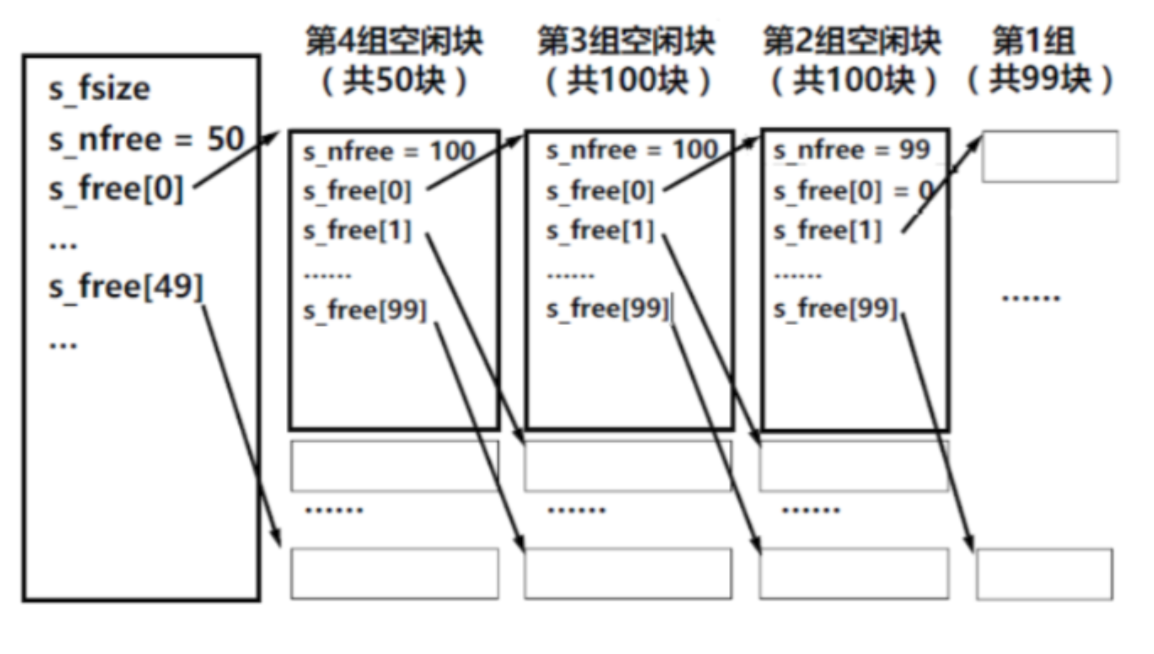
# 三、详细设计

## 3.1 重点函数说明

**3.1.1物理盘块的分配与回收（SuperBlock）：**

物理盘块在SuperBlock内的组织方式为成组链接法

1. **int** SuperBlockManager::allocBlock()
2. {
3. **int** content[101];
4. SB->s\_dirty = 1;
5. **int** blk\_no = SB->s\_free[SB->s\_nfree - 1];
6. SB->s\_nfree -= 1;
7. **if**(SB->s\_nfree == 0)
8. {
9. ID->read((**char**\*)content, **sizeof**(content), blk\_no \* BLOCK\_SIZE);
10. SB->s\_nfree = content[0];
11. **for**(**int** i = 0; i < 100; i++)
12. {
13. SB->s\_free[i] = content[i + 1];
14. }
15. }
16. **return** blk\_no;
17. }
18. **void** SuperBlockManager::freeBlock(**int** no)
19. {
20. **int** content[101];
21. SB->s\_dirty = 1;
22. **if**(SB->s\_nfree == 100)
23. {
24. content[0] = SB->s\_nfree;
25. **for**(**int** i = 0; i < 100; i++)
26. {
27. content[i + 1] = SB->s\_free[i];
28. }
29. ID->write((**char**\*)content, **sizeof**(content), no \* BLOCK\_SIZE);
30. SB->s\_nfree = 0;
31. }
32. SB->s\_free[SB->s\_nfree] = no;
33. SB->s\_nfree += 1;
34. **return** ;
35. }



分配和回收物理块本质就是对由一个个小的栈组成的链表进行维护。超级块记录这个链表的第 一项，每次增减新的空闲块都是对超级块内记录的 100 个数据进行修改。当超级块记录的数据超过

100 项时，则将前 100 项形成一个新的栈插入链表中;如果超级块记录的数据不足 0 项，则取出链 表中的头结点，填入超级块中。

**3.1.2 Inode节点的分配与回收（SuperBlock）**

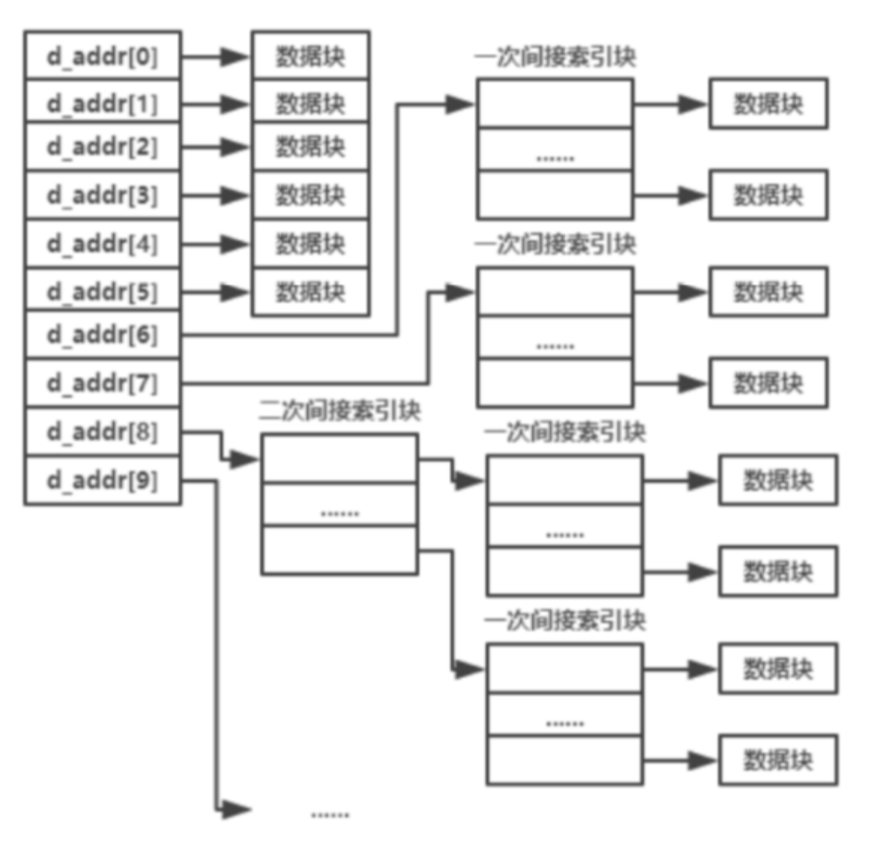
Inode在SuperBlock中采用位示图法

1. **int** SuperBlockManager::allocInode()
2. {
3. **char** c;
4. **int** one = 1;
5. **for**(**int** i = 0; i < IBITMAP\_SIZE; i++)
6. {
7. c = SB->s\_ibitmap[i];
8. cout << "c: " << (**int**)c << endl;
9. **for**(**int** j = 0; j < 8; j++)
10. {
11. **if**((c & (one << (7 - j))) == 0)
12. {
13. cout << "分配的节点为 " << 8 \* i + j << endl;
14. //分配成功
15. (SB -> s\_ibitmap)[i] |= (one << (7 - j));
16. //补充操作
17. **return** (8 \* i + j);
18. }
19. }
20. }
21. cout << "Inode 节点用尽" << endl;
22. **return** -1;
23. }
24. **void** SuperBlockManager::freeInode(**int** no)
25. {
26. **char** c;
27. **int** one = 1;
28. **for**(**int** i = 0; i < IBITMAP\_SIZE; i++)
29. {
30. c = SB->s\_ibitmap[i];
31. cout << "c: " << (**int**)c << endl;
32. **for**(**int** j = 0; j < 8; j++)
33. {
34. **if**(no == 0)
35. {
36. //将s\_ibitmap的第 8 \* i + j 位置零
37. cout << "free inode: " << 8 \* i + j << endl;
38. SB->s\_ibitmap[i] = (c & !(one << (7 - j)));
39. **return**;
40. }
41. no--;
42. }
43. }
44. cout << "ERROR: free inode failed." << endl;
45. **return** ;
46. }

申请和回收外存 INode 节点时，都是通过检查、搜索保存在超级块中的􏰁示图来进行的。相对比较简单。

**3.1.3逻辑盘块到物理盘块的映射（BufferManager）：**

物理盘块地址记录在Inode节点i\_addr数组中，前六个地址为直接映射，第7、8个地址为一级间接索引，第9、10个地址为二级间接索引。原理如下图：



1. Block\* BufferManager::readFileBlock(Inode\* f\_inode, **int** t)
2. {
3. **int** no;
4. **int** i\_blkno;
5. **int**\* in;
6. **if**(t < 6)
7. {
8. no = f\_inode->i\_addr[t];
9. }
10. **else** **if**(t >= 6 && t <= 6 + 128 \* 2)
11. {
12. i\_blkno = readBlock(f\_inode->i\_addr[(t - 6) / 128 + 6]);
13. BF->b\_blocks[i\_blkno].b\_blkno = f\_inode->i\_addr[(t - 6) / 128 + 6];
14. in = g\_Buffer.b\_blocks[i\_blkno].getIndexList();
15. **if**((t - 6) / 128 + 6 == 6)
16. no = in[t - 6];
17. **else**
18. no = in[t - 6 - 128];
19. **if**(no == 0)cout << "ERROR: read file error." << endl;
20. }
21. **else**
22. {
23. i\_blkno = readBlock(f\_inode->i\_addr[(t - 6 - 128 \* 2) / 128 / 128 + 8]);
24. BF->b\_blocks[i\_blkno].b\_blkno = f\_inode->i\_addr[(t - 6 - 128 \* 2) / 128 / 128 + 8];
25. in = BF->b\_blocks[i\_blkno].getIndexList();
27. **int** t2;
28. **if**((t - 6 - 128 \* 2) / 128 / 128 + 8 == 8)
29. t2 = (t - 6 - 128 \* 2) / 128;
30. **else**
31. t2 = (t - 6 - 128 \* 2 - 128 \* 128) / 128;
32. **int** i\_blkno2 = readBlock(in[t2]);
33. g\_Buffer.b\_blocks[i\_blkno2].b\_blkno = in[t2];
34. **int**\* in2 = g\_Buffer.b\_blocks[i\_blkno2].getIndexList();
35. **if**((t - 6 - 128 \* 2) / 128 / 128 + 8 == 8)
36. no = in2[(t - 6 - 128 \* 2) % 128];
37. **else**
38. no = in2[(t - 6 - 128 \* 2 - 128 \* 128) % 128];
39. **if**(no == 0)cout << "ERROR: read file error." << endl;
40. }
41. **int** i\_blkno2 = readBlock(no);
42. **return** &BF->b\_blocks[i\_blkno2];
43. }

**3.1.4文件读写算法（以fread为例）**

首先是readFileBlock函数，该函数作用为读取文件第t块内容，返回值为Block，其中包括了对于大文件的处理。

1. Block\* BufferManager::readFileBlock(Inode\* f\_inode, **int** t)
2. {
3. **int** no;
4. **int** i\_blkno;
5. **int**\* in;
6. **if**(t < 6)
7. {
8. no = f\_inode->i\_addr[t];
9. }
10. **else** **if**(t >= 6 && t <= 6 + 128 \* 2)
11. {
12. i\_blkno = readBlock(f\_inode->i\_addr[(t - 6) / 128 + 6]);
13. BF->b\_blocks[i\_blkno].b\_blkno = f\_inode->i\_addr[(t - 6) / 128 + 6];
14. in = g\_Buffer.b\_blocks[i\_blkno].getIndexList();
15. **if**((t - 6) / 128 + 6 == 6)
16. no = in[t - 6];
17. **else**
18. no = in[t - 6 - 128];
19. **if**(no == 0)cout << "ERROR: read file error." << endl;
20. }
21. **else**
22. {
23. i\_blkno = readBlock(f\_inode->i\_addr[(t - 6 - 128 \* 2) / 128 / 128 + 8]);
24. BF->b\_blocks[i\_blkno].b\_blkno = f\_inode->i\_addr[(t - 6 - 128 \* 2) / 128 / 128 + 8];
25. in = BF->b\_blocks[i\_blkno].getIndexList();
27. **int** t2;
28. **if**((t - 6 - 128 \* 2) / 128 / 128 + 8 == 8)
29. t2 = (t - 6 - 128 \* 2) / 128;
30. **else**
31. t2 = (t - 6 - 128 \* 2 - 128 \* 128) / 128;
32. **int** i\_blkno2 = readBlock(in[t2]);
33. g\_Buffer.b\_blocks[i\_blkno2].b\_blkno = in[t2];
34. **int**\* in2 = g\_Buffer.b\_blocks[i\_blkno2].getIndexList();
35. **if**((t - 6 - 128 \* 2) / 128 / 128 + 8 == 8)
36. no = in2[(t - 6 - 128 \* 2) % 128];
37. **else**
38. no = in2[(t - 6 - 128 \* 2 - 128 \* 128) % 128];
39. **if**(no == 0)cout << "ERROR: read file error." << endl;
40. }
41. **int** i\_blkno2 = readBlock(no);
42. **return** &BF->b\_blocks[i\_blkno2];
43. }

然后是fread函数，根据所需要读的大小，先计算需要读多少个逻辑块以及当前逻辑块是否读完，读完当前逻辑块后计算需要读多少个完整的逻辑块，最后计算剩余还需读多少字节，将其拼接在一个buffer中，具体实现如下：

1. **void** Operator::fread(**char**\* buffer, **const** **int** size)
2. {
3. **if**(size > 10000)
4. {
5. cout << "ERROR: read too long, cannot print!" << endl;
6. **return** ;
7. }
8. **int** no = f\_seek / BLOCK\_SIZE;
9. Block\* read\_block = **new** Block;
10. read\_block = BM->readFileBlock(f\_inode, no);
11. **int** size\_r = size;
12. **int** buffer\_offset = 0;
13. **int** sur0 = f\_seek % BLOCK\_SIZE;
14. //先读第一块（可能是不全的）
15. **if**(size\_r + sur0 > BLOCK\_SIZE)
16. {
17. memcpy(buffer + buffer\_offset, read\_block->content + sur0, BLOCK\_SIZE - sur0);
18. size\_r = size - BLOCK\_SIZE + sur0;
19. buffer\_offset += BLOCK\_SIZE - sur0;
20. no++;
21. }
22. **else**{
23. memcpy(buffer + buffer\_offset, read\_block->content + sur0, size\_r);
24. buffer\_offset += size\_r;
25. }
26. **while**(size\_r > BLOCK\_SIZE)
27. {
28. read\_block = BM->readFileBlock(f\_inode, no);
29. memcpy(buffer + buffer\_offset, read\_block->content, BLOCK\_SIZE);
30. buffer\_offset += BLOCK\_SIZE;
31. size\_r -= BLOCK\_SIZE;
32. no++;
33. }
34. **if**(size\_r > 0 && no != 0)
35. {
36. read\_block = BM->readFileBlock(f\_inode, no);
37. memcpy(buffer + buffer\_offset, read\_block->content, size\_r);
38. buffer\_offset += size\_r;
39. }
40. **return**;
41. }

写操作与读操作类似，唯一不同的是，如果写入的数据超出文件当前数据，需要进行文件Block的申请扩充，修改SuperBlock和相应Inode节点信息。

**3.1.5文件目录项操作**

使用目录栈保存当前目录：

1. stack<**int**> CL;                  //目录编号
2. stack<string> CLN;              //目录名称

进入某文件夹时，进行push操作，退出某文件夹时进行pop操作，暂不支持绝对路径

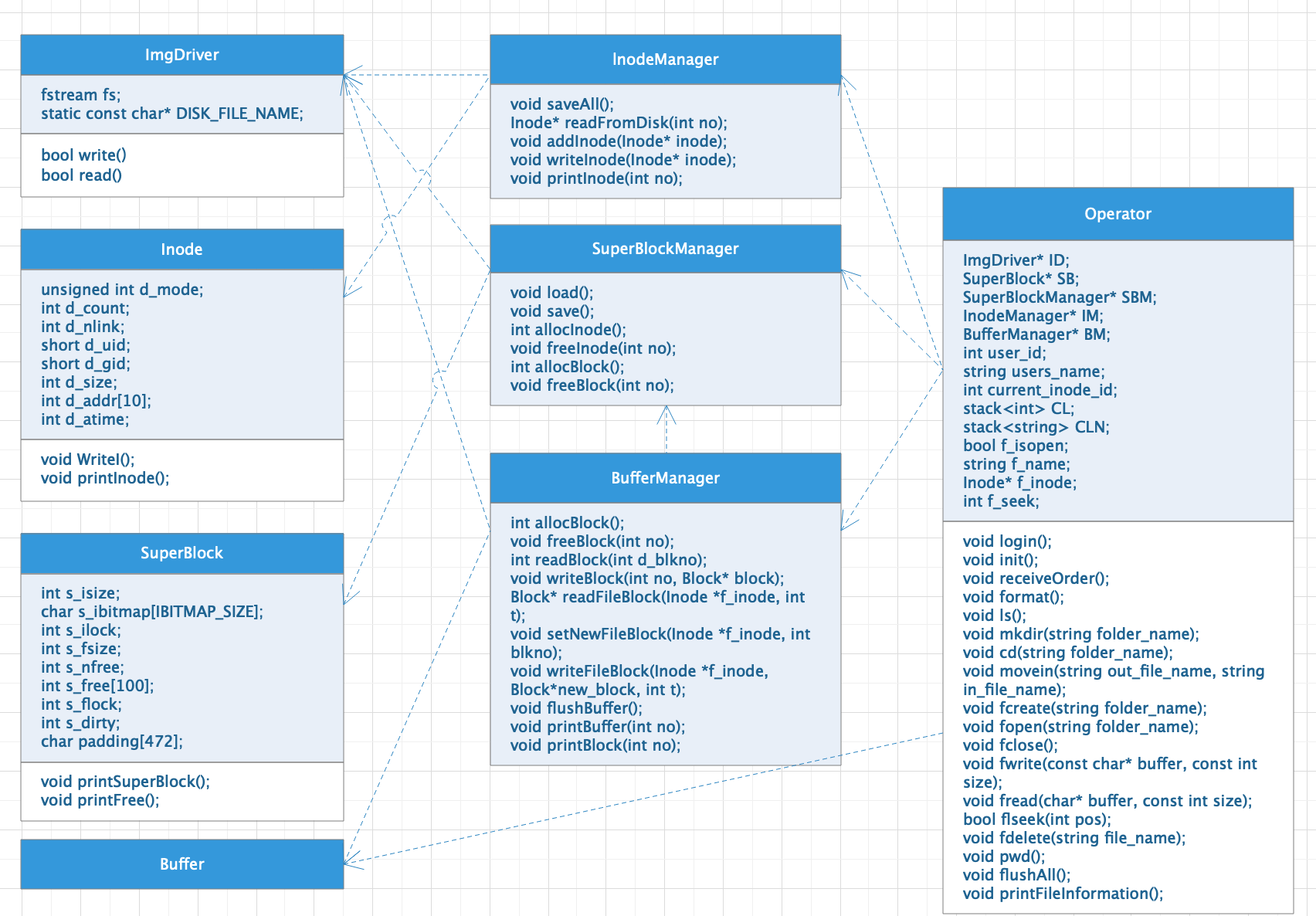
**3.1.6用户输入处理**

首先是对于输入字符串的处理，输入字符串以空格为间隔分割为多个字符串。

1. vector<string> split(string str, **const** string pattern)
2. {
3. string::size\_type pos;
4. vector<string> result;
5. str += pattern;//扩展字符串以方便操作
6. **int** size = str.size();
7. **for** (**int** i = 0; i<size; i++) {
8. pos = str.find(pattern, i);
9. **if** (pos<size) {
10. std::string s = str.substr(i, pos - i);
11. result.push\_back(s);
12. i = pos + pattern.size() - 1;
13. }
14. }
15. **return** result;
16. }

接下来进一步对于vector类型的输入结果进行处理，处理逻辑为简单的if-elseif结构。

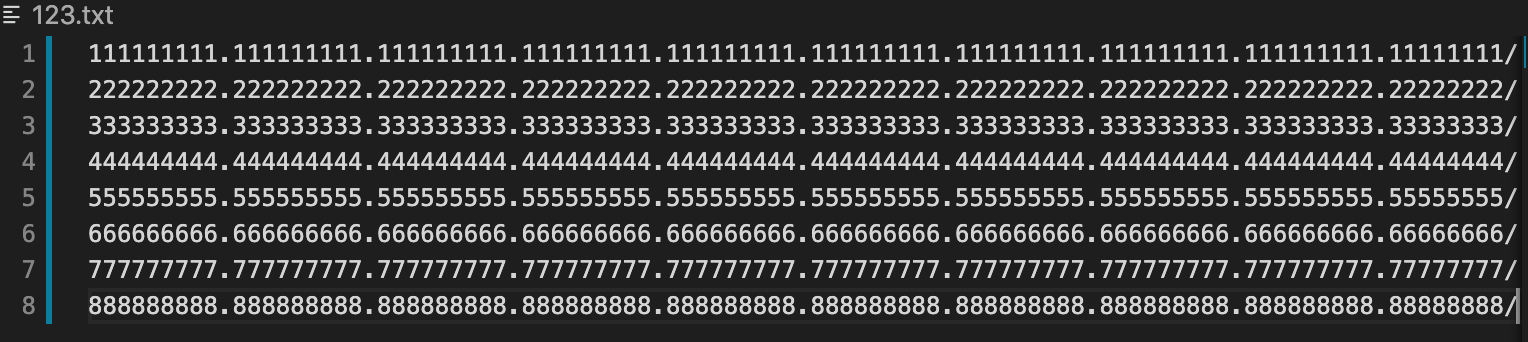
## 3.2 类及函数调用关系



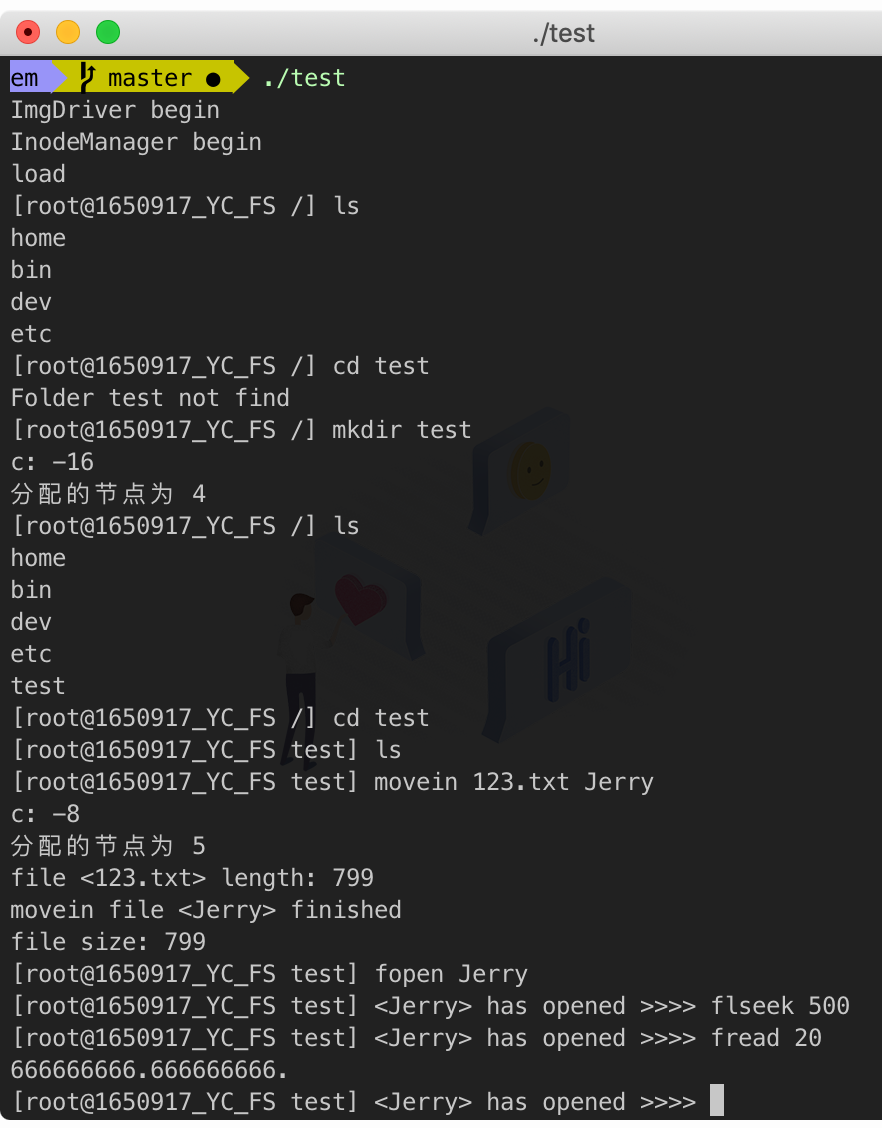
# 四、调试分析

## 4.1 测试过程及测试数据

首先逐条编写测试每条指令，根据print指令输出值进行分析和调试，综合测试如下：登录后新建文件夹，新建文件，拷贝文件进文件系统，设置读写指针，从指定位置读出数据。写入的文件内容如下：



测试过程如下图：



## 4.2 遇到的问题及解决方案

此处只列出几个重要错误。

（1）大文件盘块映射错误

解决方案：检查文件映射算法，修改盘块映射算法，根据print命令输出相应Inode节点及盘块内容进行分析调试。

（2）存储大文件时SuperBlock记录错误

解决方案：在对大文件进行盘块分配时，SuperBlock记录逻辑出现错误，导致其并不能正确记录文件存储盘块，二级映射无法正常记录，最终通过修改算法得以解决。

（3）当没有使用exit命令退出时，之前所修改的内容无法正确记录，同时，新创建文件夹及文件节点时会再次分配已分配节点，造成记录错误。

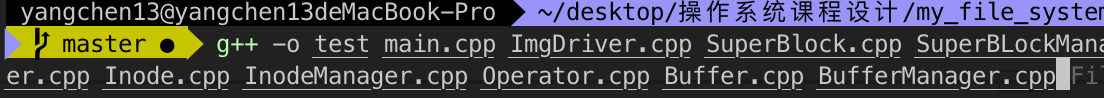
解决方案：修改缓冲区刷新逻辑，将缓冲区刷新时间设置为每次指令运行结束后，即每条指令结束后刷新全部SuperBlock、Inode和缓冲区内容，保证每次指令运行后磁盘中保存内容为最新内容。

# 五、用户使用说明

## 5.1 开发平台和编译说明

开发平台为 Mac OS操作系统，使用的编译器为g++

编译命令如下图：



## 5.2用户登录说明

程序需先输入用户名密码进行登录，目前内部提供可以登录的账号密码如下：

账号： root 密码： 123

账号： user1 密码： 123

账号： user2 密码： 123

## 5.3 用户操作命令说明

cd [folder]:进入文件夹folder

mkdir [folder]:创建文件夹folder

ls:列出当前目录下所有文件及文件夹

pwd:列出当前路径

format:格式化整个磁盘

print Inode [inode\_no]:输出第inode\_no个inode的信息

print Block [block\_no]:输出第block\_no个block的信息

print SuperBlock:输出SuperBlock的信息

freat [file]:创建文件file

fopen [file]:打开文件file

fread [length]:从当前位置读取length长度的内容，需先打开文件

fwrite [string]:从当前位置写入string，需先打开文件

fclose:关闭当前打开的文件

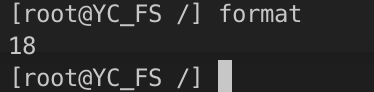
flseek [pos]:设置文件读写指针位置为pos，需先打开文件

fdelete [file]:删除文件file

movein [outside\_file] [inside\_file]:将外部带路径文件outside\_file拷贝进当前目录，并命名为inside\_file

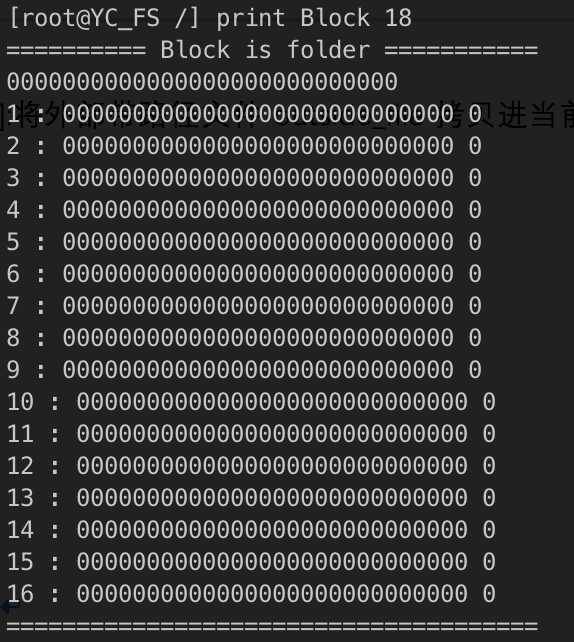
# 六、运行结果分析

1.format格式化



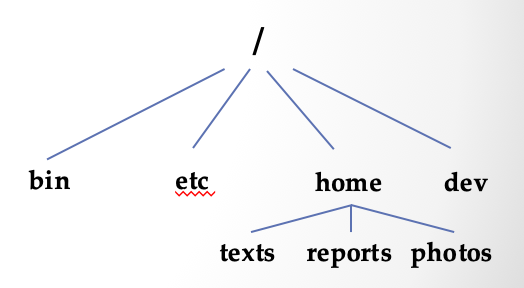
输出信息表示第18块Block为根目录的文件目录保存位置

此时调用输出函数 print Block 18可查看根目录文件内容



上图输出信息表示当前目录没有任何文件及文件夹。

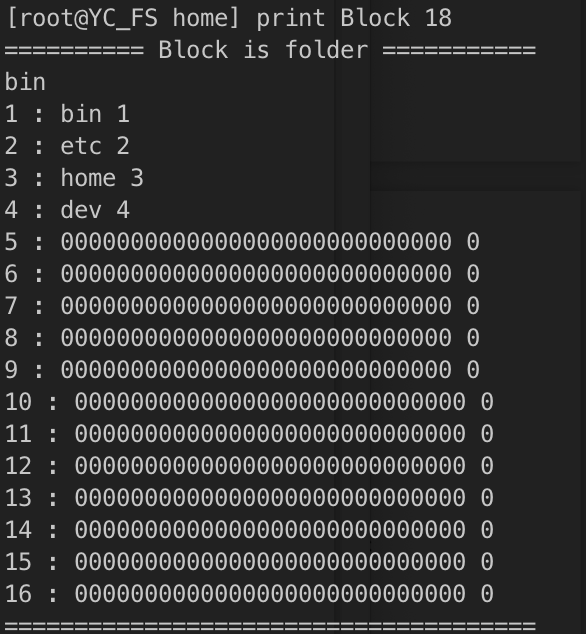
2.创建如图文件目录





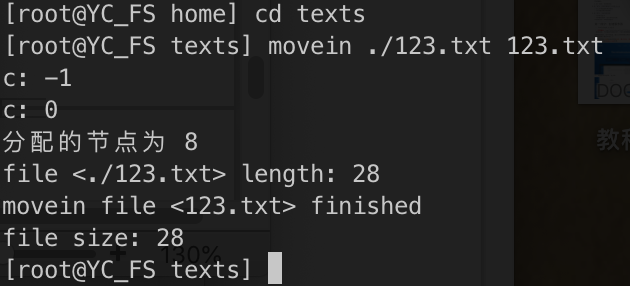


此时调用print查看18号block信息，输出结果如下图：

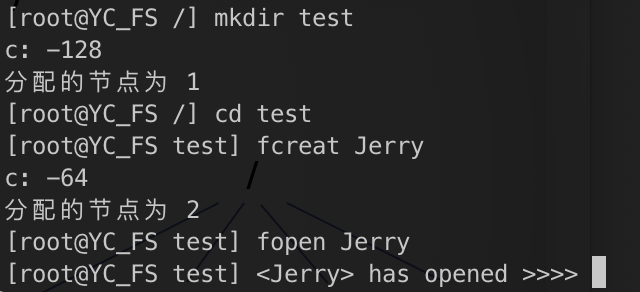


3.将一个纯文本文件，课设报告和一张图片存入该文件系统

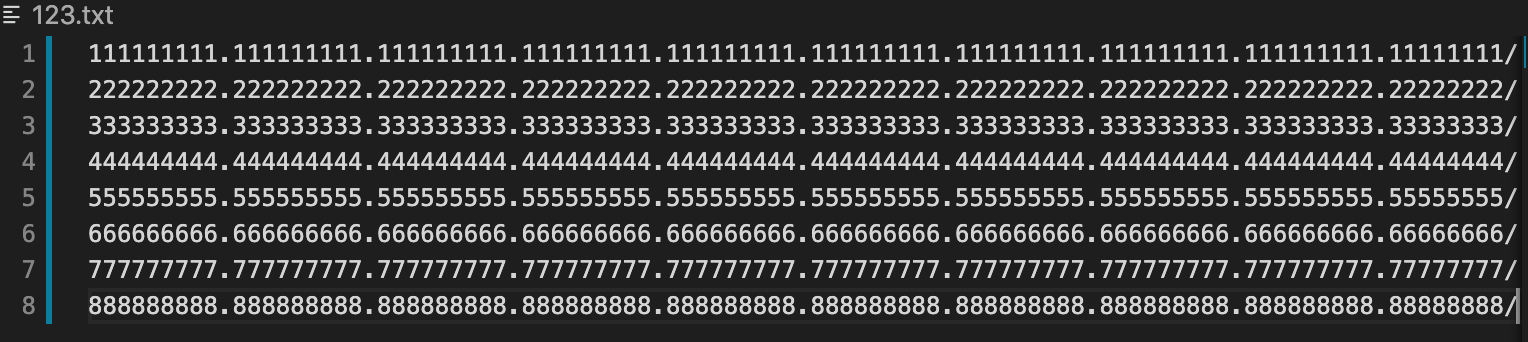
在/home/texts目录中执行movein ./123.txt 123.txt，另外两文件同理



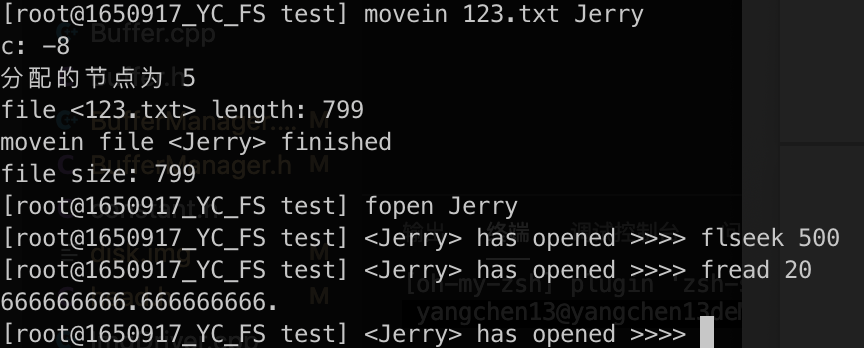
4.新建文件/test/Jerry，打开该文件，写入800字节，将指针指向500字节，读出20字节



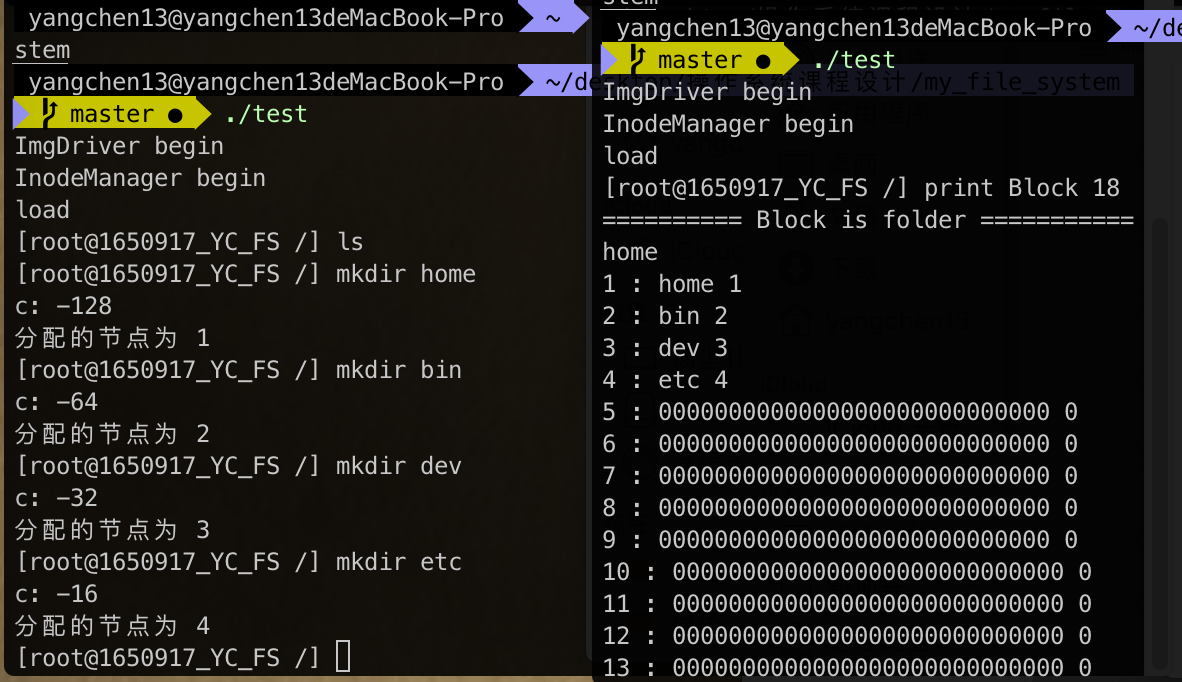
800字节内容如下图：



写入并从第500字节开始读出20字节。



5.多用户登录测试：同时开两个终端，一个终端读写文件，从另一个终端查看Block内容，再进行功能交换，查看结果。



# 七、实验总结

## 7.1 综合实验过程的收获

此次课程设计极大的锻炼了我对于面向对象的C++编程的能力，主要在于类的设计，如何做到有效的划分，使得各类达到高内聚、低耦合，同时使得接口尽可能简洁高效，在完成此次课程设计中也出现过很多的问题，通过编写测试文件不断进行测试更正，对于我的编程能力也有一定的提高。

## 7.2 遇到问题及解决问题过程的思考

在课程设计过程中，遇到过许多的问题，其中有一些问题是自己对于c++文件操作的理解不够，通过查资料等方式加深了自己对于c++文件操作的理解。

还有一些问题出现在类的设计上，由于设计问题，导致接口使用起来并不简洁高效，而是出现复杂繁琐等问题，在高层调用时需要反复查看底层代码，该类问题仍需要自己加强面向对象的设计能力，在类的设计上多学习参考其他人的成功经验。

另外，在具体的文件操作问题上也出现了一些问题，原因是自己对于UNIX V++文件系统的理解不够深刻，对于一些细节的把握不够清晰，导致在进行设计时走了很多弯路，在进行设计之前首先应该对于自己所要设计的东西有充分的理解，这样才能最大程度上减少整个系统的设计中出现的问题。

## 7.3 综合实验中对课程的认识

在此次课程设计中，通过自己亲身实践，将UNIX V6++的文件系统进行实现，增强了自己对于操作系统中的文件系统的理解，包括硬盘的分区情况，超级块SuperBlock的结构，内存Inode和磁盘Inode之间的关系，物理盘块的分配和回收，文件的存储方式，在自己实践完成的过程中，对于文件的形式和结构以及文件系统有了更加深刻的理解，不在只停留于书本知识，也让我对于操作系统这门课有了更深刻的理解。

## 7.4 不足及改进

基于本次课程设计，由于时间关系等因素，仍可以改进的方面如下：

1. 添加文件移出功能，目前仅能通过读取字节判断是否存取正确。
2. 类间调用关系可以进一步优化，降低类与类之间耦合的程度。
3. 当前程序不支持使用绝对路径，可进一步增加对于绝对路径的支持。

# 八、参考文献

[1] Linux 文件系统(一)—虚拟文件系统VFS----超级块、inode、dentry、file https://blog.csdn.net/shanshanpt/article/details/38943731

[2] 浅析unix文件系统 <https://yisaer.github.io/2017/03/23/pwd/>

[3] https://github.com/Sen666666/SecondFileSystem