**同济大学计算机系**

**操作系统课程设计说明书**

****

**学 号 1552160**

**姓 名 贺 聪**

**专 业 计算机科学与技术**

**授课老师 方 钰**

目录

[1.需求分析 2](#_Toc516832789)

[1.1程序任务 2](#_Toc516832790)

[1.2程序功能 2](#_Toc516832791)

[2.概要设计 4](#_Toc516832792)

[2.1系统层次图： 4](#_Toc516832793)

[2.2数据结构 4](#_Toc516832794)

[2.2.1 Superblock 5](#_Toc516832795)

[2.2.2 inode 5](#_Toc516832796)

[2.2.3 FAT 6](#_Toc516832797)

[2.2.4 Directory目录项 7](#_Toc516832798)

[2.2.5 文件打开表 8](#_Toc516832799)

[2.3任务分解 8](#_Toc516832800)

[2.3.1 创建文件 8](#_Toc516832801)

[2.3.2 删除文件 9](#_Toc516832802)

[2.4主程序流程 9](#_Toc516832803)

[3.详细设计 9](#_Toc516832804)

[3.1格式化文件卷 9](#_Toc516832805)

[3.2列目录 10](#_Toc516832806)

[3.3创建文件 12](#_Toc516832807)

[3.4打开文件 13](#_Toc516832808)

[3.5关闭文件 15](#_Toc516832809)

[3.6读文件 15](#_Toc516832810)

[3.7写文件 17](#_Toc516832811)

[3.8定位文件读写指针 17](#_Toc516832812)

[3.9删除文件 18](#_Toc516832813)

[4.调试分析 19](#_Toc516832814)

[4.1测试案例 19](#_Toc516832815)

[4.2输出结果： 19](#_Toc516832816)

[4.3问题 24](#_Toc516832817)

[4.4复杂度分析 25](#_Toc516832818)

[5.用户使用说明 25](#_Toc516832819)

[6.实验总结 26](#_Toc516832820)

[7.参考文献 26](#_Toc516832821)

# 1.需求分析

## 1.1程序任务

使用一个普通的大文件（myDisk.txt，称之为一级文件） 来模拟UNIX V6++的一个文件卷。用户在CMD界面下输入相应功能号来完成相应功能。功能号如下：

* 功能号：1 ——格式化文件卷
* 功能号：2 ——列目录
* 功能号：3 ——新建文件
* 功能号：4 ——打开文件
* 功能号：5 ——关闭文件
* 功能号：6 ——读文件
* 功能号：7 ——写文件
* 功能号：8 ——定位文件读写指针
* 功能号：9 ——删除文件
* 功能号：0 ——退出

## 1.2程序功能

1. 格式化文件卷

删除文件数据取中的所有数据，格式化superblock，所有的inode和FAT。

1. 列目录

分级列出文件目录。一级目录为root；然后列出二级目录下的所有目录文件和普通文件，各级目录之间用‘/’分开（如root/usr）；依此类推，直至列出文件卷中所有目录以及每个目录下的文件。

1. 新建文件

用户输入文件名，文件名以目录路径的形式给出（如root/usr/jerry），系统根据文件名创建文件。若文件名中有未创建的目录文件（如usr），则先创建目录文件，再再创建后面的目录文件或普通文件。为新创建的文件分配inode和一块文件数据块。

1. 打开文件

用户输入文件名（路径名），以及打开模式（1-只读；2-只写；3-可读可写），系统根据路径名找到文件，并给该文件分配文件号，并返回该文件号。如果用户输入的文件名有误，则提示用户文件未找到，并返回-1。

1. 关闭文件

用户给出文件号（该文号必须是代开文件函数返回的文件号），系统回收该文件号，表示该文件关闭。若用户输入的文件号有误，则提示用户“错误的文件号！”。

1. 读文件

用户给出文件号（该文号必须是代开文件函数返回的文件号）和读取长度，系统根据文件号，定位到文件的inode和数据区，并从数据取中读取指定长度的内容，并打印出来。若用户输入的文件号有误，则提示用户“错误的文件号！”。若用户输入的长度加上该文件的文件去写指针超过该文件的大小，则提示用户“读取错误！超出文件范围，请检查读取长度和文件读写指针”

1. 写文件

用户给出文件号（该文号必须是代开文件函数返回的文件号），写入内容和写入长度，系统根据文件号定位到文件的inode和数据区，并从读写指针处开始，依次写入指定长度的内容。若用户输入的文件号有误，则提示用户“错误的文件号！”。

1. 定位文件读写指针

用户给出文件号（该文号必须是代开文件函数返回的文件号）和读写指针位置，系统根据文件号和读写指针的位置重置相应文件的读写指针。若用户输入的文件号有误，则提示用户“错误的文件号！”。

1. 删除文件

用户输入文件名（路径名），系统根据文件名删除相应的文件，后手该文件的inode和磁盘数据块，在该文件的上一级目录文件中删除相应的目录项。若用户输入的文件名有误，则提示用户“文件未找到”。

# 2.概要设计

## 2.1系统层次图：

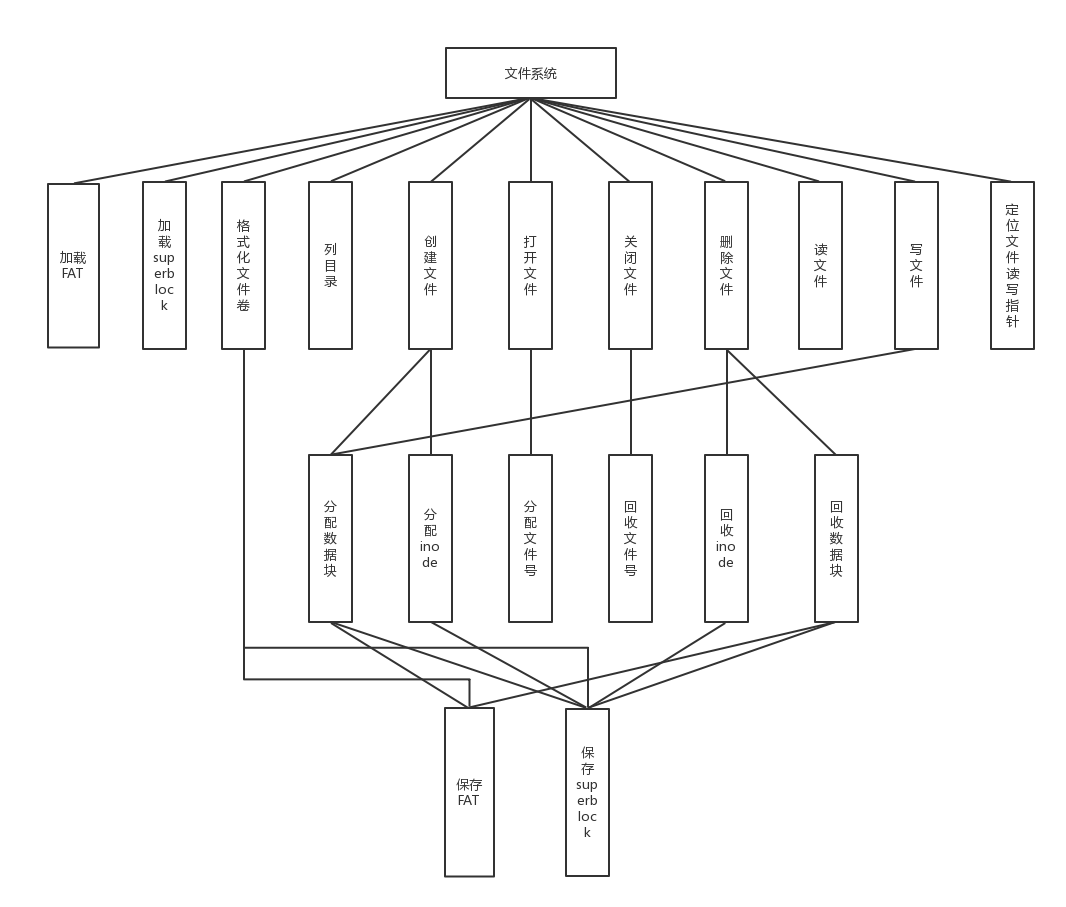


图2.1.1 系统层次图

## 2.2数据结构



图2.2.1 文件卷数据结构图

在磁盘上创建myDisk.txt来模拟UNIX V6++的一个逻辑磁盘。逻辑磁盘为1MB，磁盘中的信息以块为单位，每块大小为512B。逻辑磁盘分为4个区，分别为superblock区，inode区，FAT区和文件数据区。Superblock区占用1个盘块，0#盘块；inode区占用15个盘块，1#-15#盘块；FAT区占用16个盘块，16#-37#盘块；文件数据区占用2000个盘块，48#-2047#盘块。

### 2.2.1 Superblock

Superblock存储逻辑磁盘的使用情况，其中包含磁盘各个区占用的盘块数和起始地址。Superblock还管理空闲inode和空闲数据盘块。其中空闲的inode以栈的形式管理。当要分配inode时，分配的inode起始地址=f\_inode[--f\_inode\_count]；当要回收inode时，f\_inode[f\_inode\_count++]=回收的inode起始地址。

表2.2.1 Superblock数据结构

|  |  |
| --- | --- |
| 数据名 | 说明 |
| int c\_inode | inode占用的盘块数 |
| int c\_fat | fat区占用盘块数 |
| int c\_data | 文件数据区占用的盘块数 |
| int c\_total | 盘块总数 |
| int start\_inode; | inode区的起始地址 |
| int start\_fat; | FAT区的起始地址 |
| int inode\_root\_dir; | 根目录的inode地址 |
| int start\_data; | 文件数据区的起始地址 |
| int f\_data\_count; | 文件数据区空闲的盘块数 |
| int f\_data\_entry | 空闲盘块入口地址 |
| int f\_data\_exit | 空闲盘块出口地址 |
| int f\_inode\_count | 空闲的inode数 |
| short int f\_inode[120] | 直接管理的最多120个空闲的inode，数组中每个元素为空闲的inode起始地址。 |

### 2.2.2 inode

Inode用来管理文件信息，一个文件对应一个inode（包括目录文件和普通文件）。每个inode 64字节，系统共定义120个inode(系统最多创建120个文件)。

表2.2.2 inode数据结构

|  |  |
| --- | --- |
| 数据名 | 说明 |
| int occupied\_block[10] | 文件逻辑块号和数据块号索引表 |
| int filesize; | 文件大小（单位：字节） |
| int type; | 文件类型：0-普通数据文件；1目录文件 |
| int filenum; | 文件标识号，每个文件分配一个唯一的标识号 |
| int mode; | 文件状态：0表示关闭，1表示只读方式打开，2表示只写方式打开，3表示可读可写方式打开 |
| int padding[2]; | 填充字段，使每个inode大小为64字节 |

其中，逻辑块号和数据块号索引表采用一级索引，文件大小范围为0B-5120B；occupied\_block数组中存储的是数据盘块的地址，若occupied\_block[i]还未分配数据块，则occupied\_block[i]=-1;

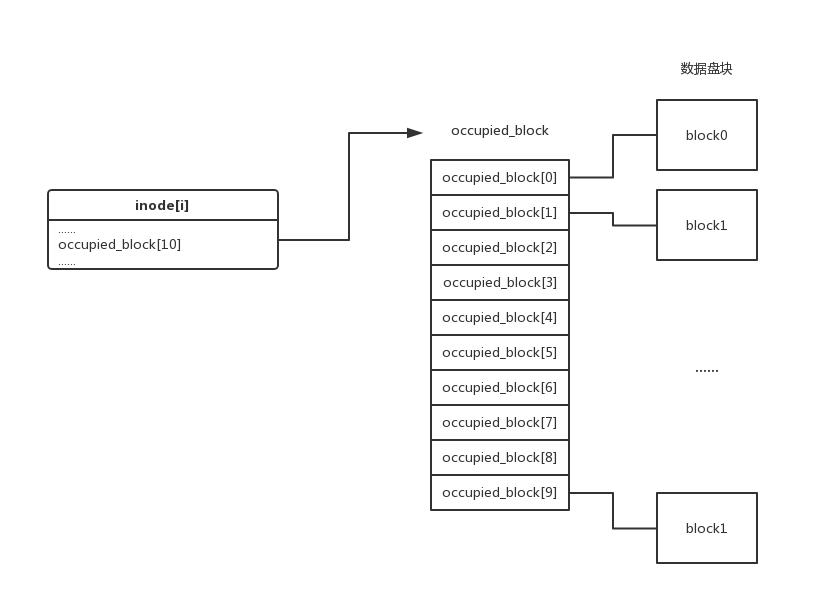


图2.2.2 逻辑块号和数据块号索引表

### 2.2.3 FAT

FAT是用来管理磁盘盘块的表格。定义磁盘盘块数据结构：

表2.2.3.1 盘块数据结构表

|  |  |
| --- | --- |
| 数据名 | 说明 |
| int start\_pos | 盘块起始地址（总为512的倍数） |
| int next\_num | 下一盘块编号 |

struct FAT //2048\*8/512=占用32块block

{

Block blocktable[Block\_Count];

};

其中，文件数据区的空闲盘块以显式链接的方式管理。文件启动时，先把FAT加载到内存，系统关闭时将内存的FAT存储到磁盘的FAT区。

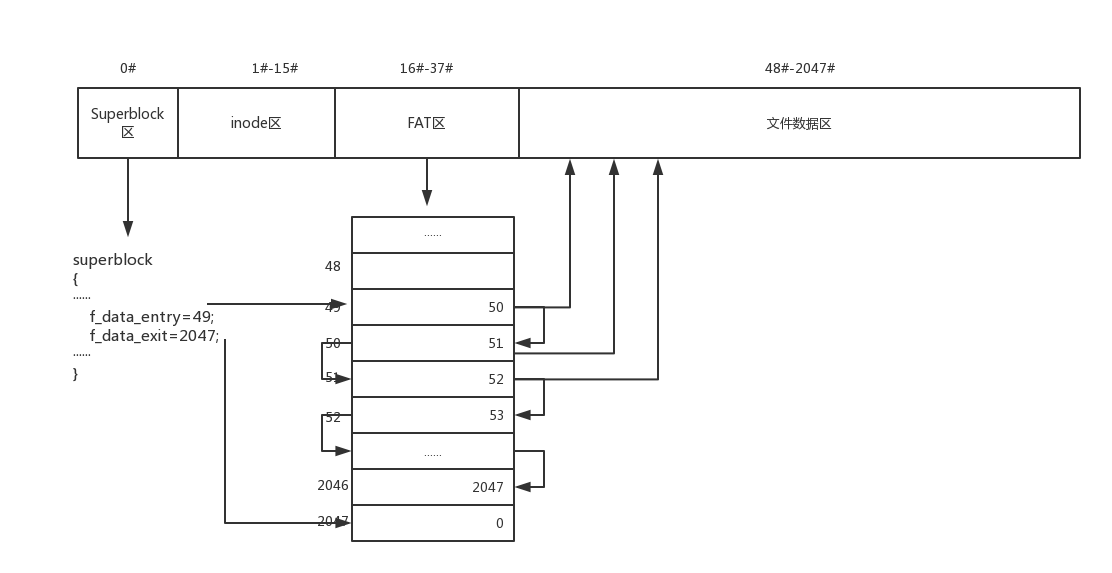


图2.2.3.1 FAT数据结构图

* 当要分配数据块时:

superblock.f\_data\_count--；

分配的数据块起始地址=FAT.blocktable[f\_data\_entry].start\_pos；

F\_data\_entry= FAT.blocktable[f\_data\_entry].next\_num; FAT.blocktable[f\_data\_entry].next\_num=0;

* 当要回收数据块时：

Superblock.f\_data\_count++;

FAT.blocktable[F\_data\_exit].next\_num=回收的数据块号；

FAT.blocktable[回收的数据块号].next\_num=0;

Superblock.f\_data\_exit=回收的数据块号;

### 2.2.4 Directory目录项

每个目录文件的组成单位为目录项，记录该目录文件下的文件。定义目录项数据结构：

表2.2.4.1 目录项数据结构

|  |  |
| --- | --- |
| 数据名 | 说明 |
| char filename[28]; | 文件名 |
| int inode; | 文件的inode地址 |

每个目录项的大小为32字节，一个数据盘块可以存储16个目录项。

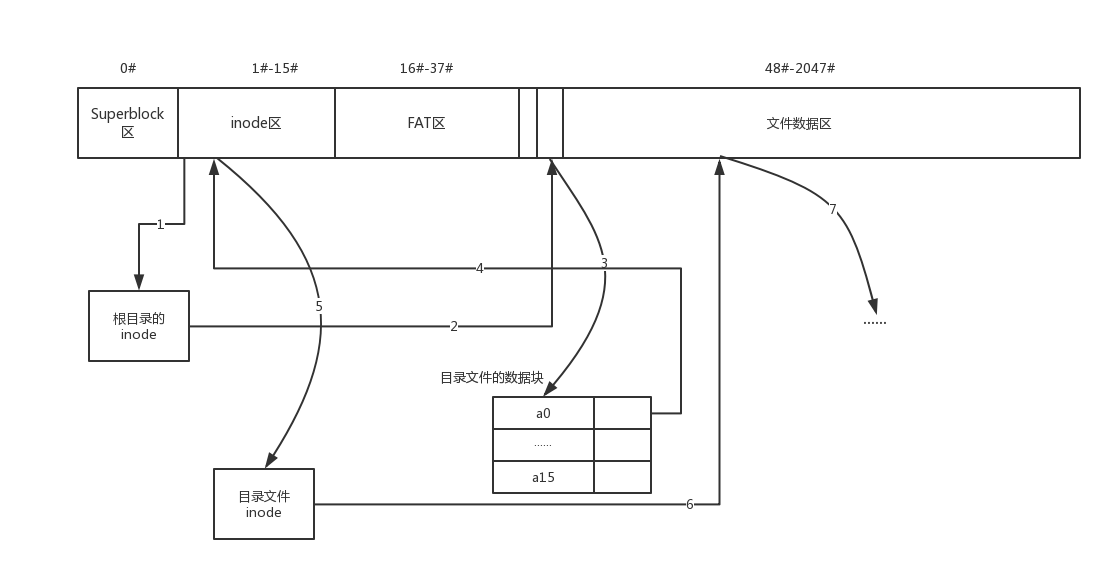


图2.2.4.1 文件目录结构（不带勾连）

### 2.2.5 文件打开表

文件打开表常驻内存，记录所有打开的文件的相关信息。定义文件表中每一项的数据结构File：

表2.2.5.1 文件打开表中的项File数据结构

|  |  |
| --- | --- |
| 数据名 | 说明 |
| int mode; | 打开方式：1-只读；2-只写；3-可读可写 |
| int inode\_addr; | 文件的inode在磁盘中的地址 |
| int flseek; | 文件读写指针位置（单位：字节） |

File \*file=new File[10]；系统定义最多同时打开10个文件。文件打开时，从文件打开表中分配一个空闲的File，并将该File在文件打开表中的项数作为文件号返回。

## 2.3任务分解

### 2.3.1 创建文件

创建文件功能模块首相根据用户输入的文件名按文件目录结构依次查找，若中间有未查找到的目录文件，则调用“分配inode”和“分配数据块”模块来完成对目录文件的创建。到最后再调用“分配inode”和“分配数据块”模块来完成对文件的创建。因此该功能模块分解为：定位-分配inode-分配数据块；

### 2.3.2 删除文件

删除文件功能模块首相根据用户输入的文件名按文件目录结构依次查找，若中间有未查找到的目录文件，则直接报错。最后调用“回收inode”和“回收数据块”模块来完成对文件的删除。因此该功能模块分解为：定位-回收inode-回收数据块；

## 2.4主程序流程

1. 打开逻辑磁盘myDisk.txt；
2. 加载Superblock；
3. 加载FAT；
4. 打印功能表；
5. 用户选择相应功能号完成相应功能

# 3.详细设计

## 3.1格式化文件卷

算法步骤为：

1. 格式化superblock：

设置参数：

c\_inode=15; //inode占用的盘块数15（120\*64/512）

c\_fat=32; //FAT占用盘块数32

c\_data=2000; //文件数据区占用的盘块数2000

c\_total=2048; //盘块总数2048

start\_inode=512; //inode区的起始地址512

start\_fat=8192; //FAT区的起始地址8192=16\*512

inode\_root\_dir = 512;

start\_data=24576; //文件数据区的起始地址24576=48\*512

f\_data\_count=1999; //文件数据区空闲的盘块数,第一块已分配给根目录

f\_data\_entry=49; //空闲盘块入口地址, 48#已分配给根目录

f\_data\_exit = 2047; //空闲盘块出口地址

f\_inode\_count=119; //空闲的inode数119；第一块inode已分配给根目录

设置空闲inode栈：

f\_inode[i] = 512 + 64 \* (119 - i);

1. 格式化inode:

设置参数：

Inode[i].occupied\_block[0-9]=-1; //还未分配数据盘块

Inode[i].filesize=0； //文件大小为0

Inode[i].filenum=i;

1. 格式化FAT:

设置起始地址：

fat->blocktable[i].start\_pos = 512 \* i; //第i个盘块的起始地址为512\*i；

设置空闲盘块链表：

for (int j = 49; j <2046; j++)

{

fat->blocktable[j].next\_num = j + 1;

}

fat->blocktable[2047].next\_num =0; //最后一个盘块为链尾

## 3.2列目录

目录按层次分别列出。首先列出一级目录下的所有文件，然后列出二级目录下的所有文件，然后是三级目录，依次类推直至所有文件都已列出。

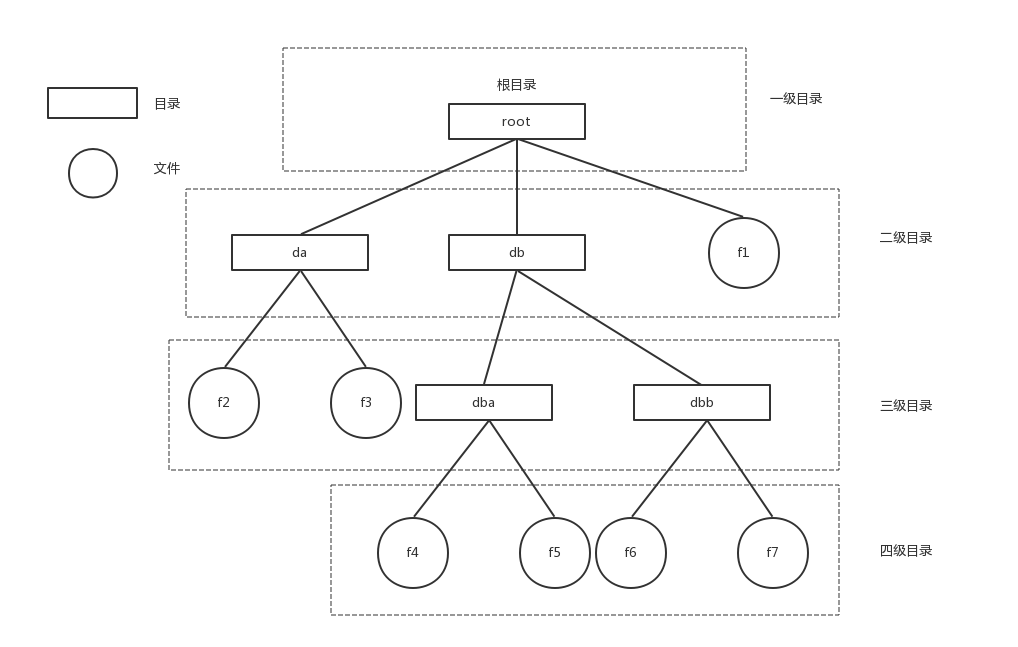


图3.2.1 目录层次图

算法实现：

算法用队列实现。开始时将根目录进队列，然后每次从队列头取出一个元素，判断该文件是否为目录文件，若为目录文件，则打印出该目录文件名称，然后将该目录文件下的所有文件依次进队列；若为普通文件，则只打印出该文件名称。依次下去直至队列空。

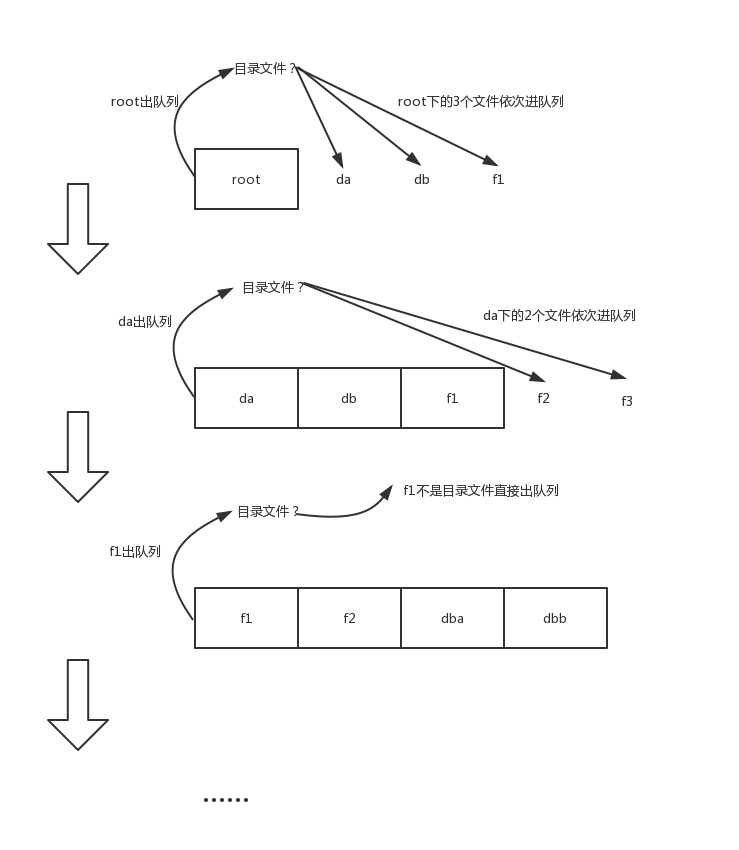


图3.2.2 列目录算法示意图

算法伪代码：

inode\_queue.insert(root); //inode\_queue.insert() 进队列函数（插入队尾）

while(inode\_queue.fetch(p)==OK)

{

if(p.type==目录文件)

{

cout<<p.filename<<endl;

dict\_num=p.filesize/32; //目录项个数

根据filesize和occupied\_block数组取出所有的目录项存数数组

Directory dict[dict\_num];

for(i=0;i<dict\_num;i++)

{

inode q=dict[i].inode对应的inode；

inode\_queue.inert(q);

}

}

else

{

cout<<p.filename<<endl;

}

}

根据该算法，图3.2.1所示的目录结构打印出来的目录形式为：

一级目录（&文件）：

Root

二级目录（&文件）：

Root/da

Root/db

Root/f1

三级目录（&文件）：

Root/da/f2

Root/da/f3

Root/db/dba

Root/db/dbb

四级目录（&文件）：

Root/db/dba/f4

Root/db/dba/f5

Root/db/dbb/f6

Root/db/dbb/f7

## 3.3创建文件

算法步骤：

1. 定义inode in,初始时in=根目录的inode;
2. 定义char filename[]，初始时，filename=二级目录名称；
3. Int dict\_num=in.filesize/32；定义dict\_num表示当前目录文件的目录项个数；
4. 从in.occupied\_block中加载所有的目录项到数组Directory dict[dict\_num];
5. 搜索dict数组，找到dict[i].filename=filename；若未找到，则跳转到8
6. In=dict[i].inode对应的inode；
7. 判断文件名是否读取完。若否，则置filename=下一集目录名称，跳转到3；若是则置in.type=0，表示最后一个为普通文件，然后结束。
8. 为新文件分配inode，并设置inode信息：filesize=0；type=1；
9. 为新文件分配一块数据盘块，occupied\_block[0]=分配的盘快地址；
10. In=新分配的inode，跳转到7；

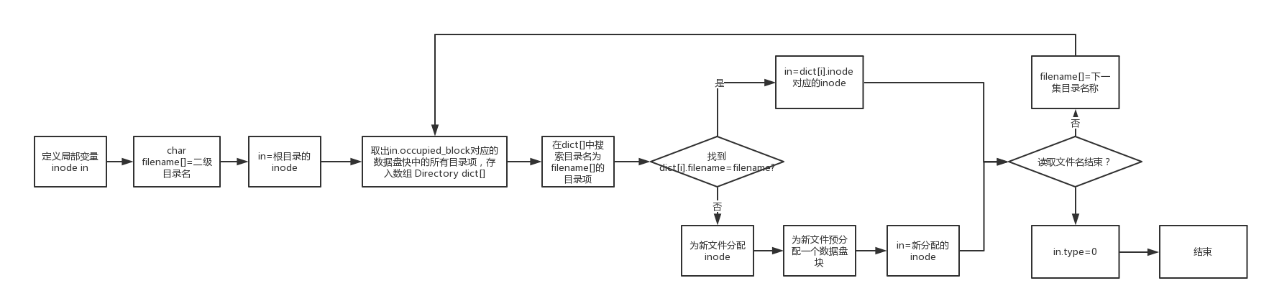


图3.3.1 创建文件流程图

该模块调用了AllocInode()函数和AllocBlock()函数为新文件分配inode和文件数据块。

其中AllocInode()函数如下：

int FileSystem::AllocInode()

{

if (SB.f\_inode\_count <= 0)

{

cout << "超过文件最大数，不可再创建文件！" << endl;

return ERROR;

}

int addr=SB.f\_inode[--SB.f\_inode\_count];

savesuperblock();

return addr;

}

AllocBlock()函数如下：

int FileSystem::AllocBlock()

{

if (SB.f\_data\_count <= 0)

{

cout << "警告：磁盘空间不足！" << endl;

return ERROR;

}

int entry = SB.f\_data\_entry;

int addr = fat.blocktable[entry].start\_pos;

SB.f\_data\_entry = fat.blocktable[entry].next\_num;

fat.blocktable[entry].next\_num = -1;

SB.f\_data\_count--;

savesuperblock();

saveFAT();

return addr;

}

## 3.4打开文件

算法步骤：

1. 定义inode in,初始时in=根目录的inode;
2. 定义char filename[]，初始时，filename=二级目录名称；
3. Int dict\_num=in.filesize/32；定义dict\_num表示当前目录文件的目录项个数；
4. 从in.occupied\_block中加载所有的目录项到数组Directory dict[dict\_num];
5. 搜索dict数组，找到dict[i].filename=filename；若未找到，则提示用户文件未找到，然后返回-1。
6. In=dict[i].inode对应的inode；
7. 判断文件名是否读取完。若否，则置filename=下一集目录名称，跳转到3；若是则为该文件分配文件号i，file[i].inode\_addr=in的地址；file[i].mode=mode；file[i].flseek=0;返回文件号i；

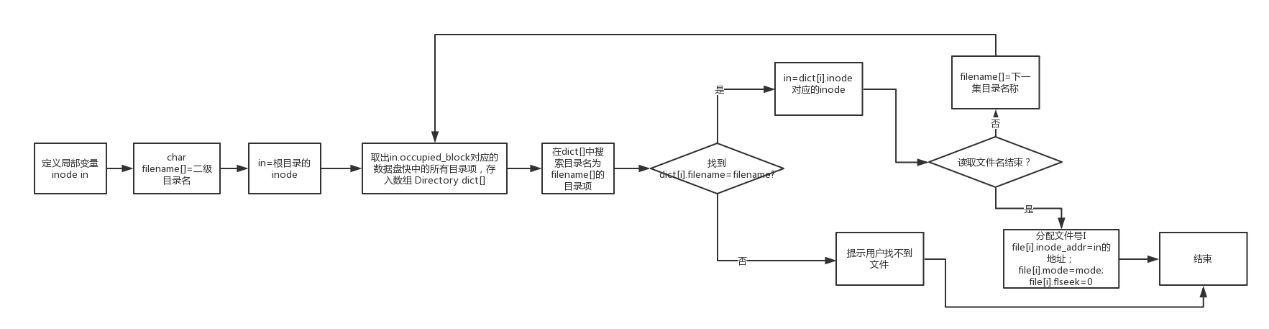


图3.4.1 打开文件流程图

其中，文件名的查找方式如下图所示（箭头上标有序号）：

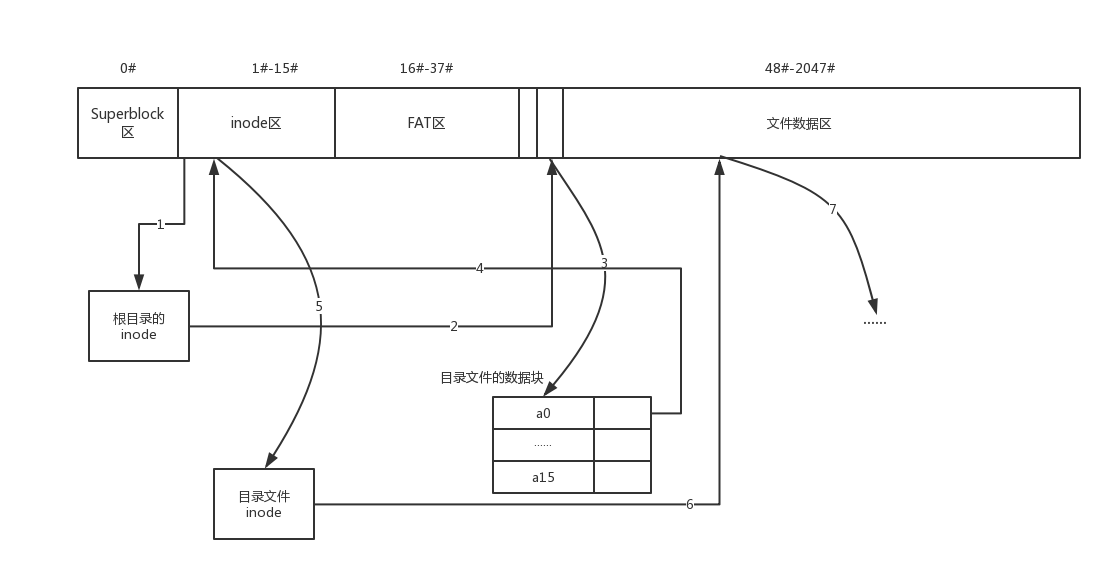


图3.4.2 文件名查找示意图

该模块调用AllocatedFile()函数为新打开的文件分配文件号。AllocatedFile()函数如下：

int FileSystem::AllocFile()

{

int i = 0;

for (i; i < 10; i++)

{

if (file[i]->mode == 0)

return i;

}

if (i == 10)

{

cout << "最多打开10个文件！" << endl;

return ERROR;

}

}

## 3.5关闭文件

该模块功能十分简单，仅回收文件号fd：

1. 首先检查fd的合法性，若小于0，表示之前文件打开错误，则提示用户错误的文件号
2. 然后检查file[fd]的状态，若为mode==0，则表示用户输入了一个错误的文件号，提示用户错误的文件号；
3. 然后回收file[fd]；

伪代码如下：

if (fd < 0)

{

cout << "error！错误的文件号！" << endl;

return;

}

if (file[fd]->mode == 0)

{

cout << "error！错误的文件号：" << fd << endl;

return;

}

delete file[fd];

file[fd] = new File;

## 3.6读文件

算法流程：

1. 首先检查文件的mode；若mode=0,提示用户错误的文件号；若mode=1，合法；若mode=2，表示文件只可读，提示用户文件无读取权限；mode=3，合法；
2. 加载file[fd]的inode到inode in;
3. 判断读取范围，若file[fd].flseek+length>filesize，提示用户超出文件范围；
4. 从file[fd].flseek开始，读取length长度内容到buffer。

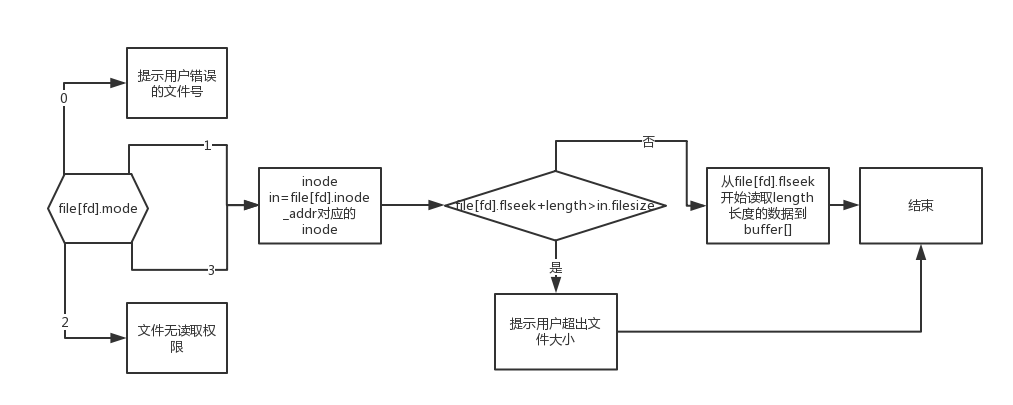


图3.6.1 读文件流程图

其中步骤4要考虑要读取的内容可能不在同一个数据块中，可能要跨块取数据，具体解决办法如下：

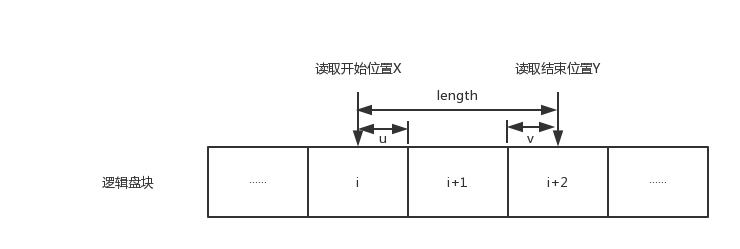


图3.6.1 跨块取数据模型

首先计算各参数：

int start\_block = file[fd]->flseek / 512; //开始的逻辑块号i

left\_block\_count = (length > (512 - file[fd]->flseek % 512)) ? ((length - (512 - file[fd]->flseek % 512)) / 512) : 0; //整块的个数（像i+1这种整块都要读取的数据块数）

u = (length > (512 - file[fd]->flseek % 512)) ? (512 - file[fd]->flseek % 512) : (length);

v = (length > (512 - file[fd]->flseek % 512)) ? (length - u) % 512 : 0;

然后分别处理第一块，中间整块和最后一块：

if (length <= (512 - file[fd]->flseek % 512)) //不跨块

{

/\*处理第一块i\*/

iofile.read(buff, length);

strcpy(buffer, buff);

return OK;

}

Else //跨块

{

/\*处理第一块i\*/

iofile.read((char\*)&buff[0], u);

buffptr += u;

/\*处理中间块\*/

for (int i = 1; i <= left\_block\_count; i++)

{

iofile.seekg(block\_num[start\_block + i]);

iofile.read((char\*)&buff[buffptr], 512);

buffptr += 512;

}

/\*处理最后一块\*/

iofile.seekg(block\_num[start\_block + left\_block\_count + 1]);

iofile.read((char\*)&buff[buffptr], v);

strcpy(buffer, buff);

return OK;

}

## 3.7写文件

算法流程：

1. 首先检查文件的mode；若mode=0,提示用户错误的文件号；若mode=1，合法；若mode=2，表示文件只可读，提示用户文件无读取权限；mode=3，合法；
2. 加载file[fd]的inode到inode in;
3. 判断读取范围，若file[fd].flseek+length>filesize，分配足够的block；
4. 从file[fd].flseek开始，将buffer[]中的数据写入length长度到数据盘块。

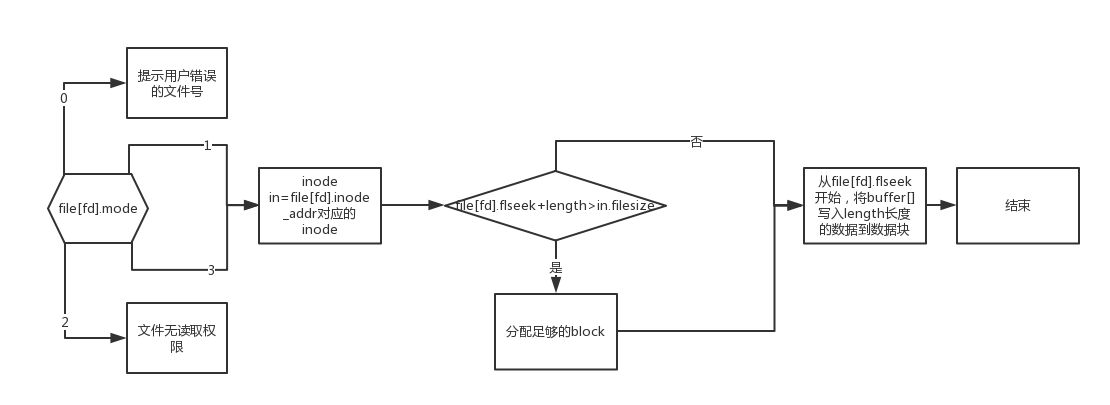


图3.7.1 写文件流程图

其中步骤3中分配足够的盘块的具体处理流程如下：

首先计算已分配的数据盘块：

Int existing\_block = (in.filesize==0)?1:(in.filesize / 512 + (in.filesize % 512) ? 1 : 0);

然后计算所需的盘块数：

need\_length = file[fd]->flseek + length; //所需长度

int i = (need\_length % 512) ? 1 : 0;

need\_block = need\_length / 512 + i; //所需盘块

alloc\_count = need\_block - existing\_block; //还需申请的盘块

然后申请盘块：

for (int i = existing\_block; i < need\_block; i++)

{

int blockaddr = AllocBlock();

in.occupied\_block[i] = blockaddr;

} //空间分配结束

## 3.8定位文件读写指针

该模块功能也十分简单，仅设置文件号为fd的文件的读写指针：

if (fd < 0)

{

cout << "error！错误的文件号！" << endl;

return ERROR;

}

if (file[fd]->mode == 0)

{

cout << "error！错误的文件号：" << fd << endl;

return ERROR;

}

file[fd]->flseek = position;

return OK;

## 3.9删除文件

算法流程：

1. 定义inode in,初始时in=根目录的inode;
2. 定义char filename[]，初始时，filename=二级目录名称；
3. Int dict\_num=in.filesize/32；定义dict\_num表示当前目录文件的目录项个数；
4. 从in.occupied\_block中加载所有的目录项到数组Directory dict[dict\_num];
5. 搜索dict数组，找到dict[i].filename=filename；若未找到，则跳转到8
6. In=dict[i].inode对应的inode；
7. 判断当前目录是否为最后一级目录文件。若否，则置filename=下一集目录名称，跳转到3；若是，则删除该目录文件下相应的目录项，然后回收要删除文件的inode和数据盘块，然后结束。
8. 提示用户未找到文件

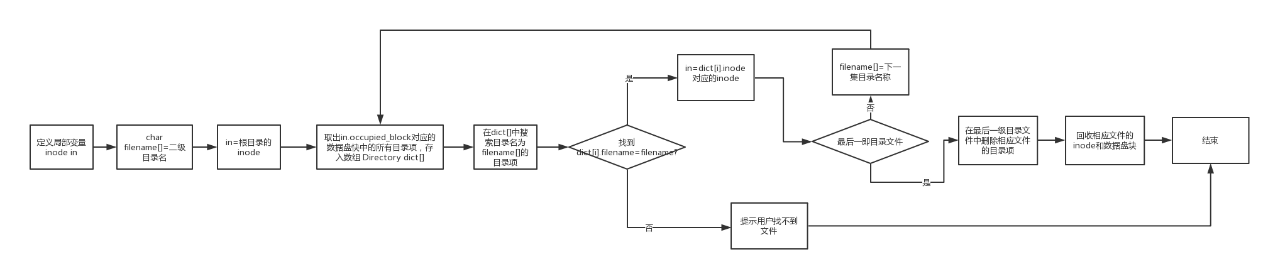


图3.9.1 删除文件流程图

其中删除最后一级目录文件中的相应目录项要考虑的因素很多，主要分为如下几个因素：

1. 删除的目录项若不是该目录文件的最后一个目录项，则会造成中间空间留白浪费。解决办法是将最后一个目录项挪动到删除的目录项位置。
2. 若最后一个目录项挪到删除的目录项位置后，造成一个数据盘块空闲（最后一个目录项独占一个数据盘块的情况），则要回收该数据盘块。
3. 若要回收的数据盘块是第一块occupied\_block[0]，则不回收（因为刚创建好的文件也会预先分配一个数据盘块，系统默认所有的文件现在或将来是肯定会有数据的）。

该模块调用了RecycleInode()函数和RecycleBlock()函数来回收inode和数据盘块。

RecycleInode()函数如下：

void FileSystem::RecycleInode(int inode\_addr)

{

SB.f\_inode[SB.f\_inode\_count++] = inode\_addr;

savesuperblock();

}

RecycleBlock()函数如下：

void FileSystem::RecycleBlock(int block\_addr)

{

int block\_num = block\_addr / 512;

int exit = SB.f\_data\_exit;

fat.blocktable[exit].next\_num = block\_num;

fat.blocktable[block\_num].next\_num = -1;

SB.f\_data\_count++;

SB.f\_data\_exit = block\_num;

savesuperblock();

saveFAT();

}

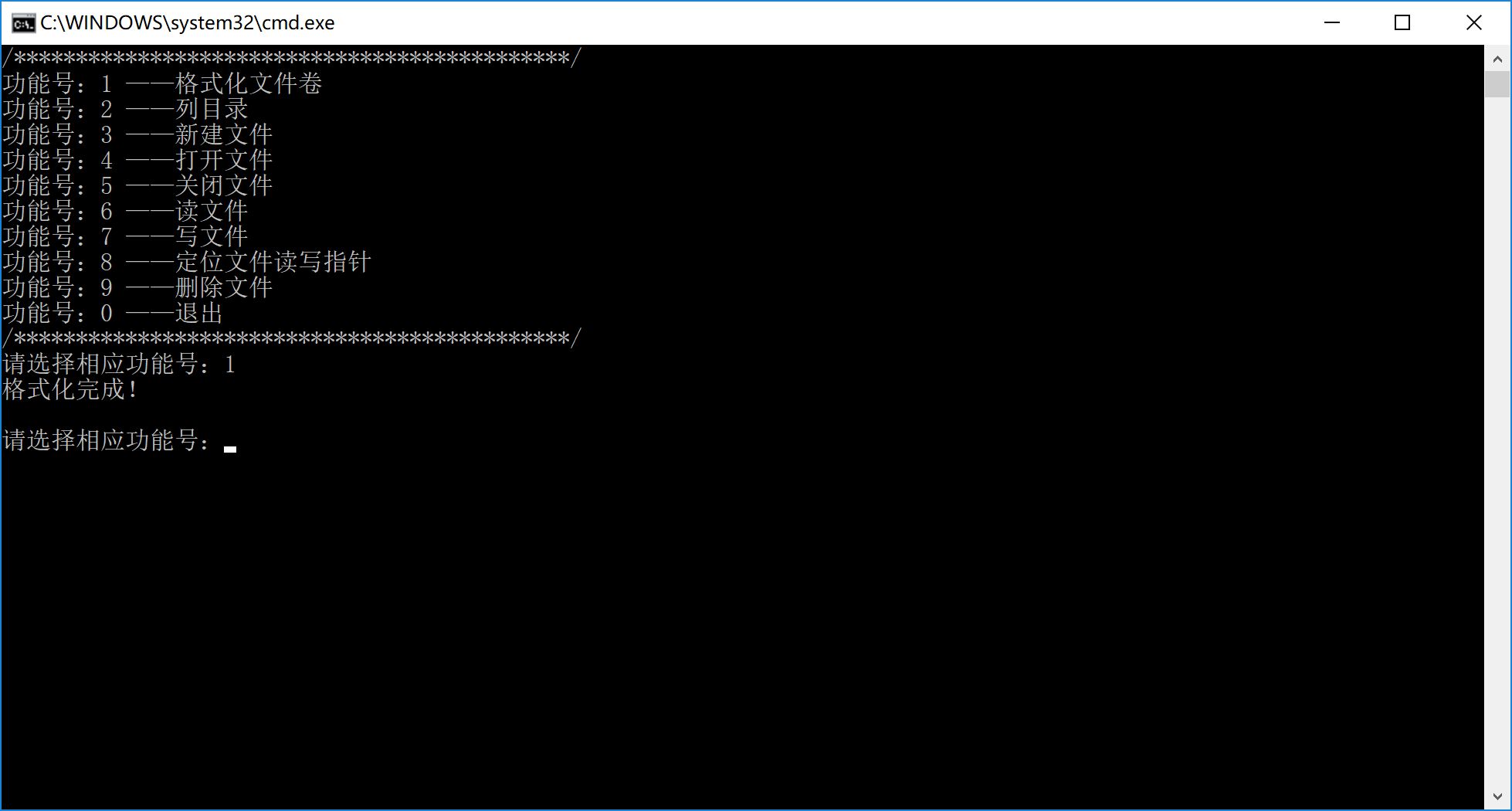
# 4.调试分析

## 4.1测试案例

1. 格式化文件卷；
2. 列目录；
3. 新建文件 root/test/jerry
4. 列目录
5. 打开文件 root/test/jerry
6. 写入800个字节，其中第500-519字节全为‘2’，其余字节全为‘1’；
7. 将文件读写指针定位到第500字节；
8. 读出20字节内容
9. 删除文件 root/test/jerry
10. 列目录

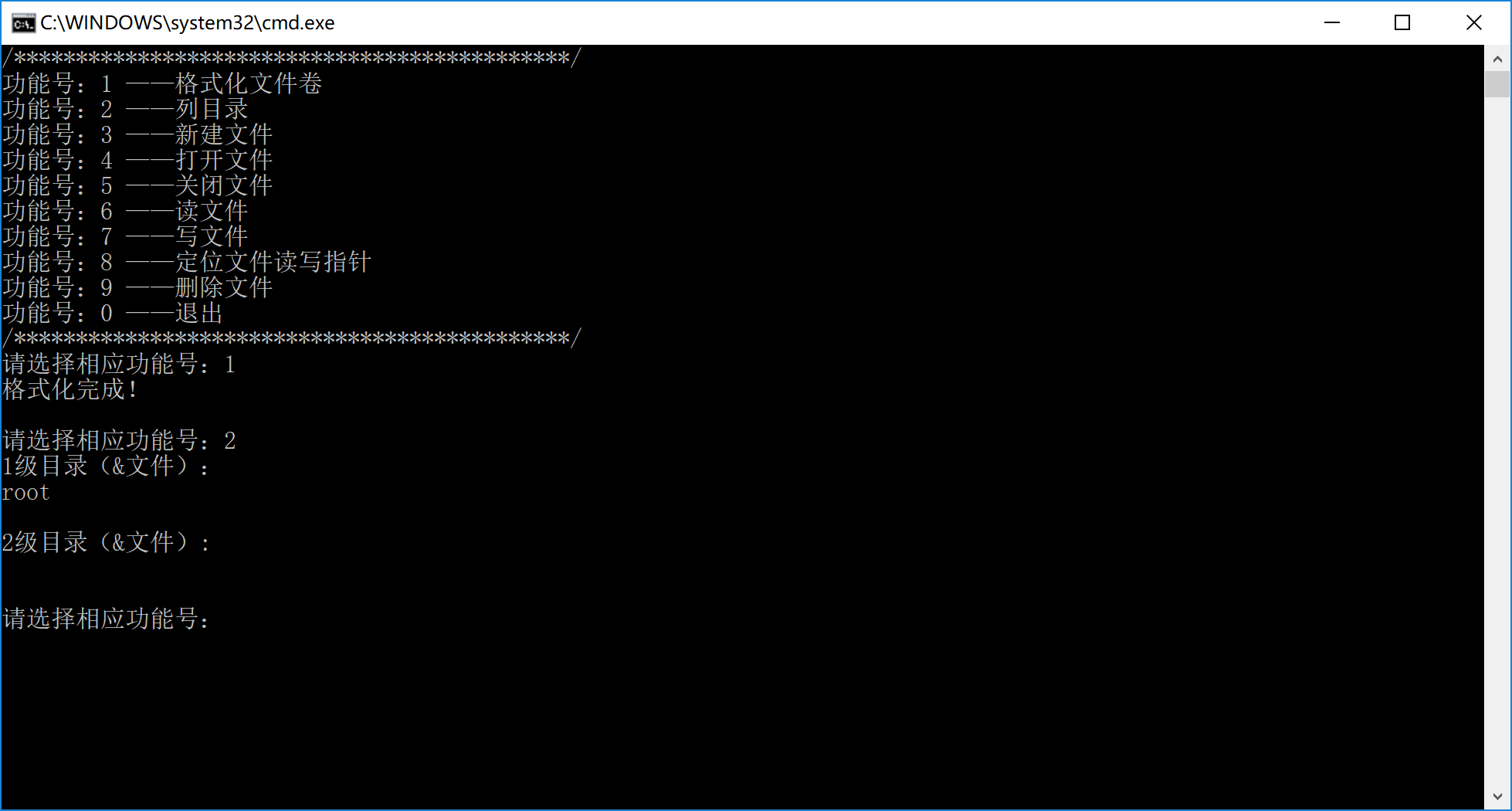
## 4.2输出结果：

1. 格式化文件卷



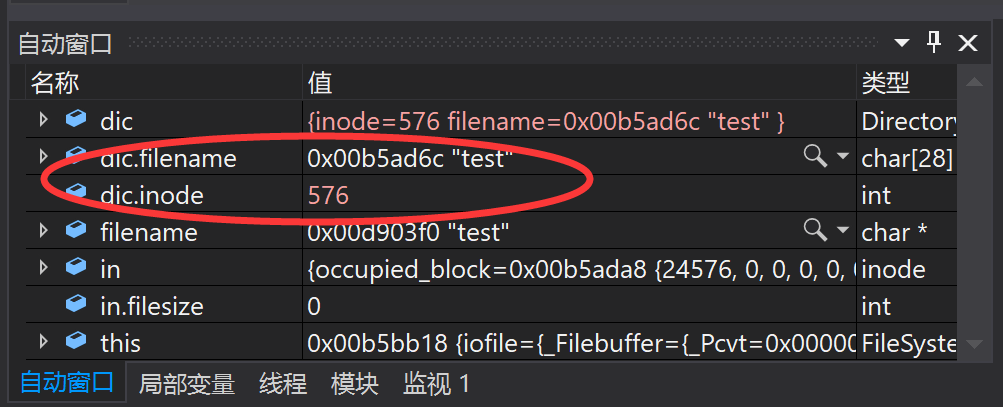
1. 列目录

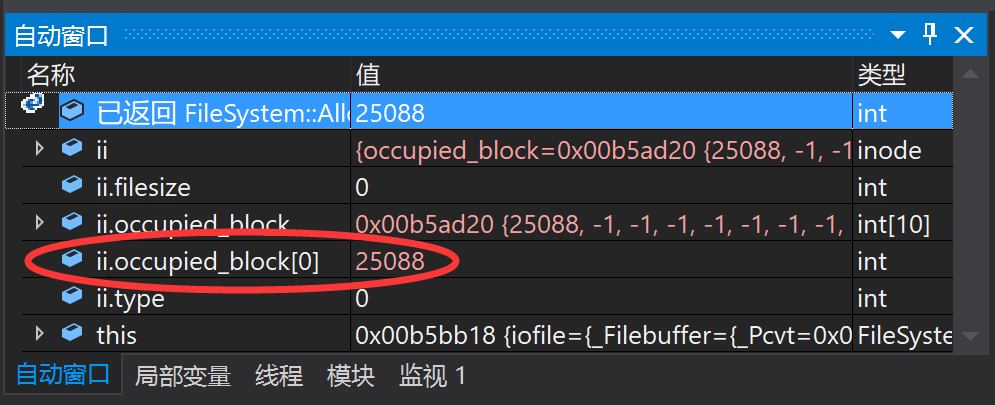
此时文件卷中只用根目录

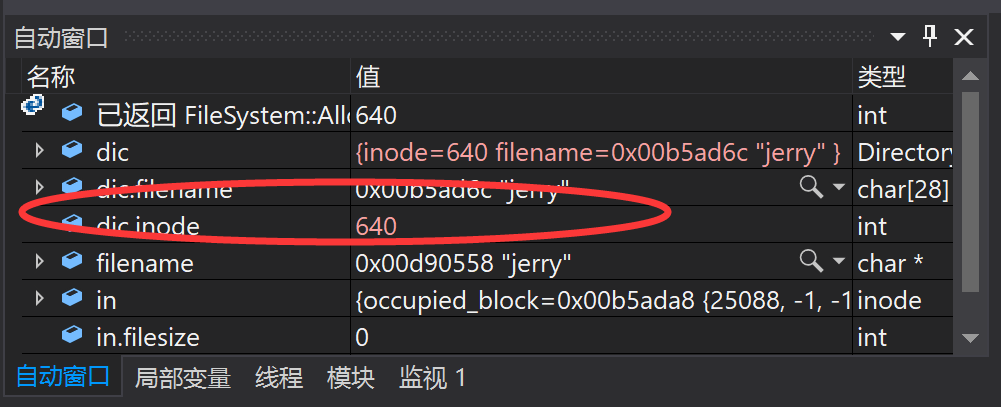


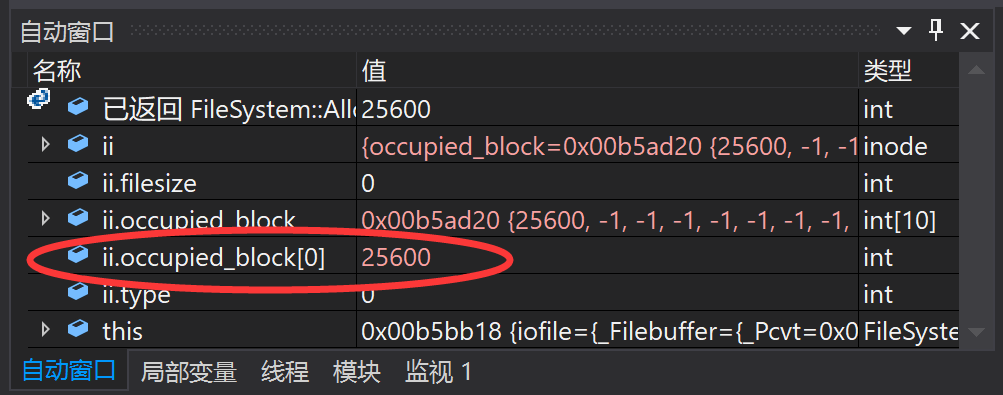
1. 新建文件 root/test/jerry

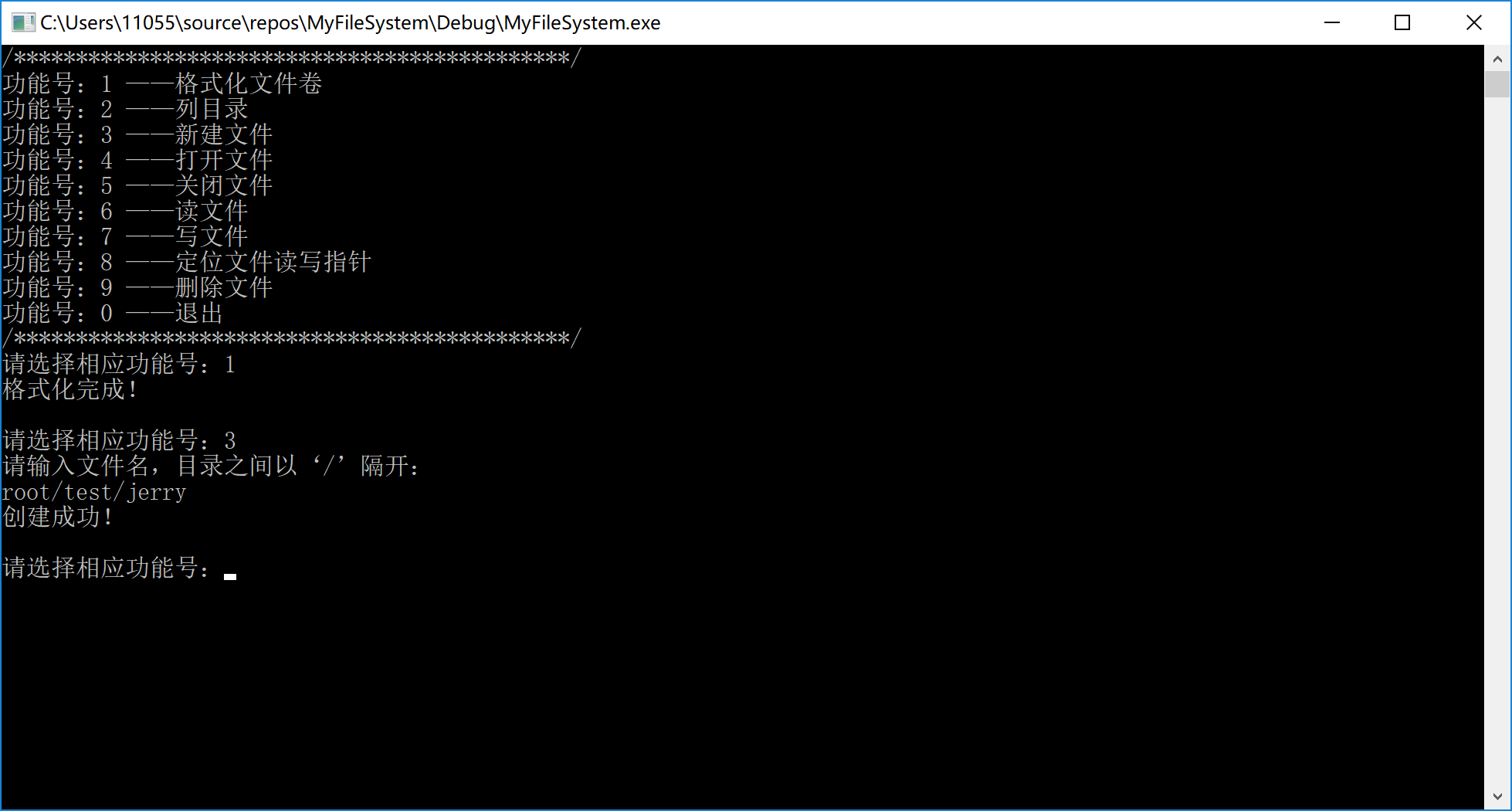
系统调用fcreate(“root/test/jerry”)函数创建文件。首先定位到二级目录test，发现root目录文件中无此目录项，则为test申请一个inode，申请到的inode地址为576B（第二块inode），为test申请一个数据盘块，occupied\_block[0]=25088B(第49块数据盘块)；然后定位到jerry，发现root目录下没有此目录项，则为jerry申请inode，申请到的inode地址为640B（第三块inode），为jerry申请数据盘块，occupied\_block[0]=25600B(第50块数据盘块)。





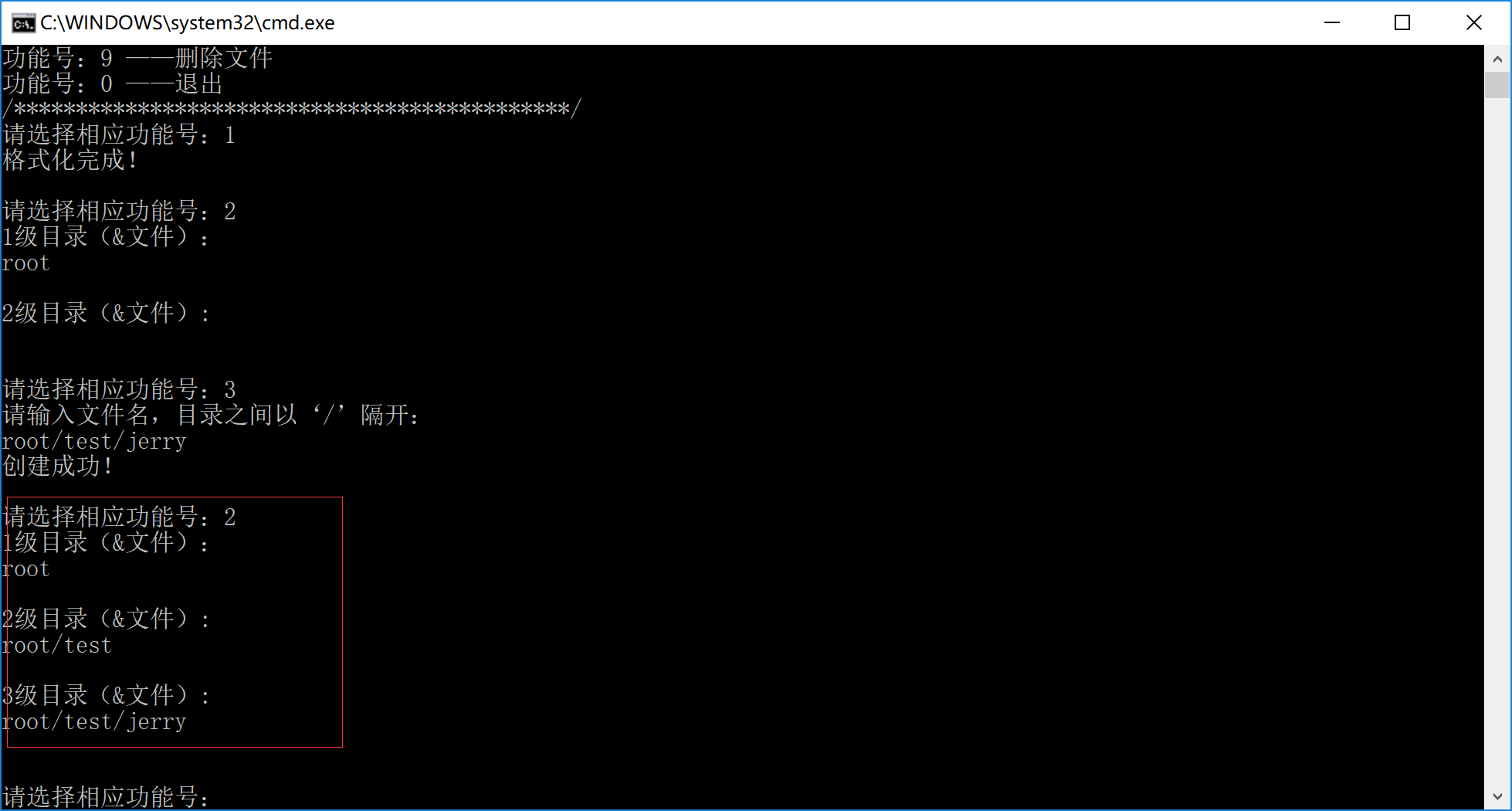






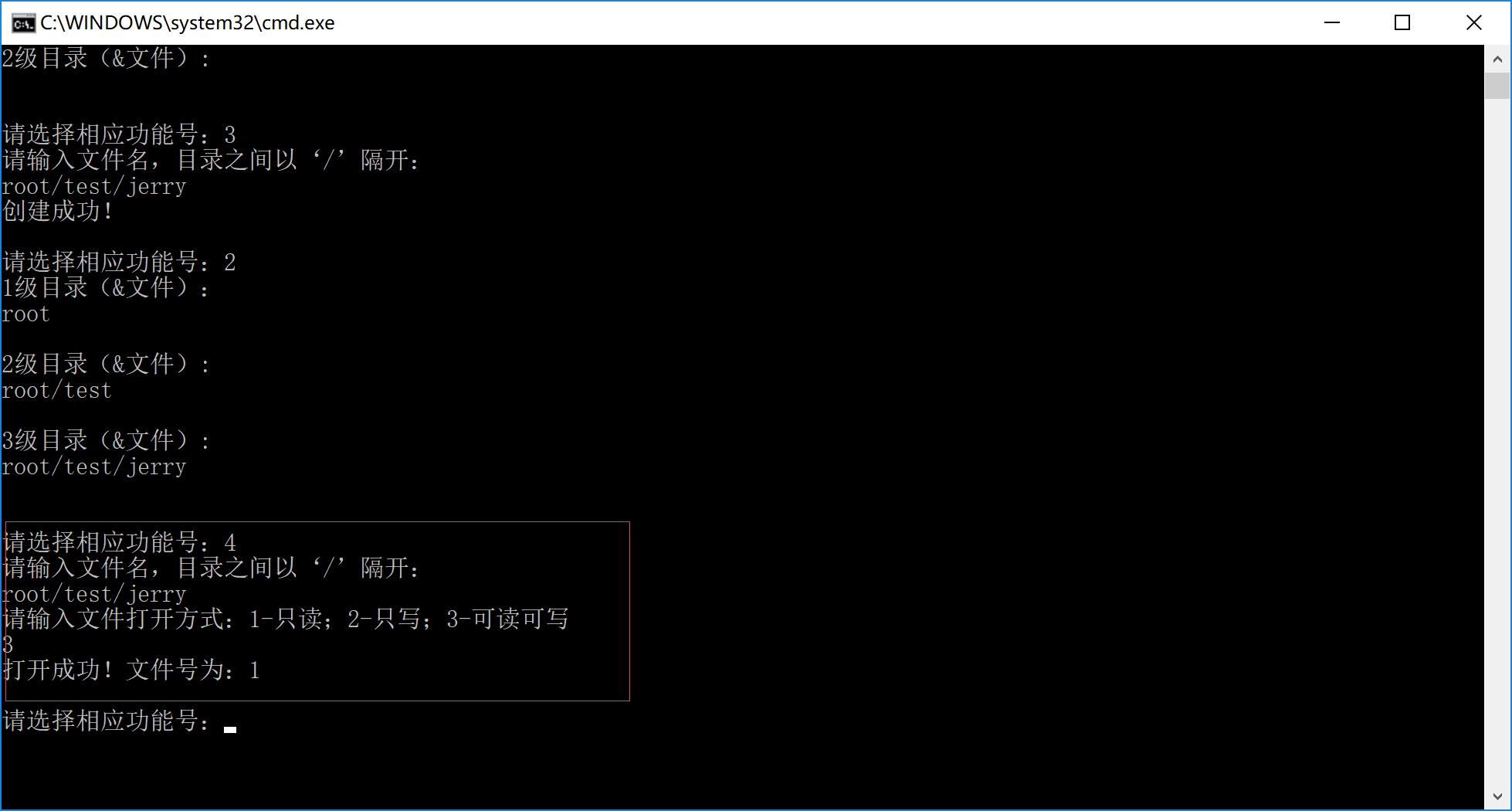
1. 列目录

此时系统中有文件root/test/jerry



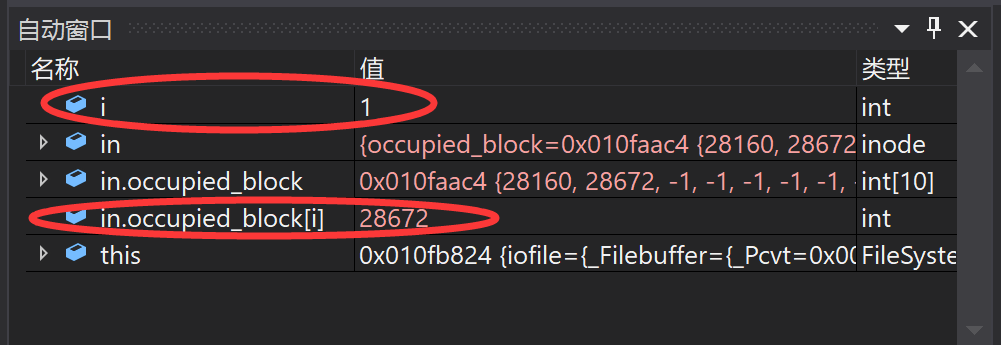
1. 打开文件 root/test/jerry

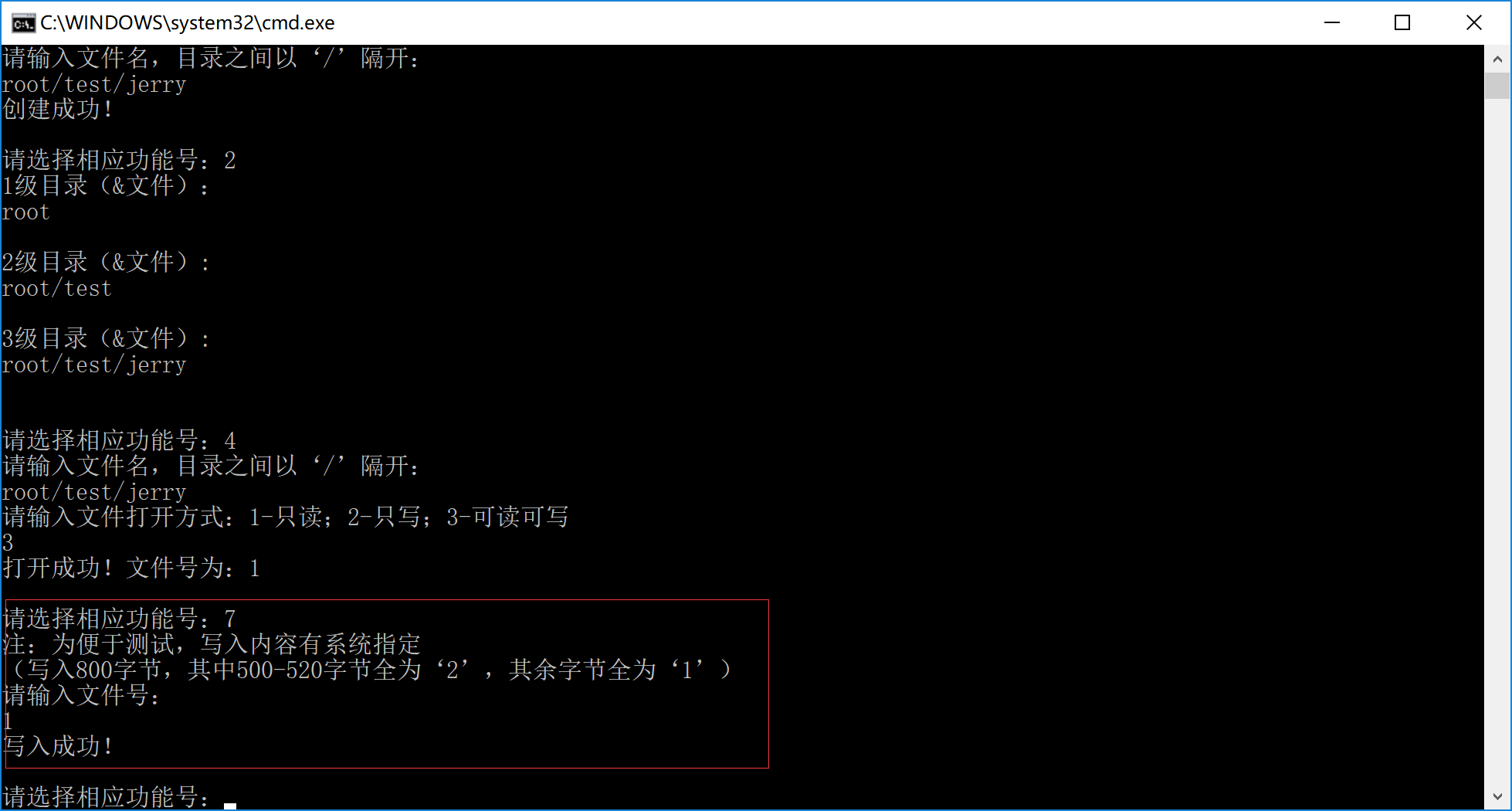
系统调用fopen(“root/test/jerry”,3)打开文件。系统定位到test，找到它的inode，从而得到occupied\_block[0]所指的数据盘块，从数据盘块中得到一个目录项，找到jerry的inode，然后为其分配file，返回文件号。



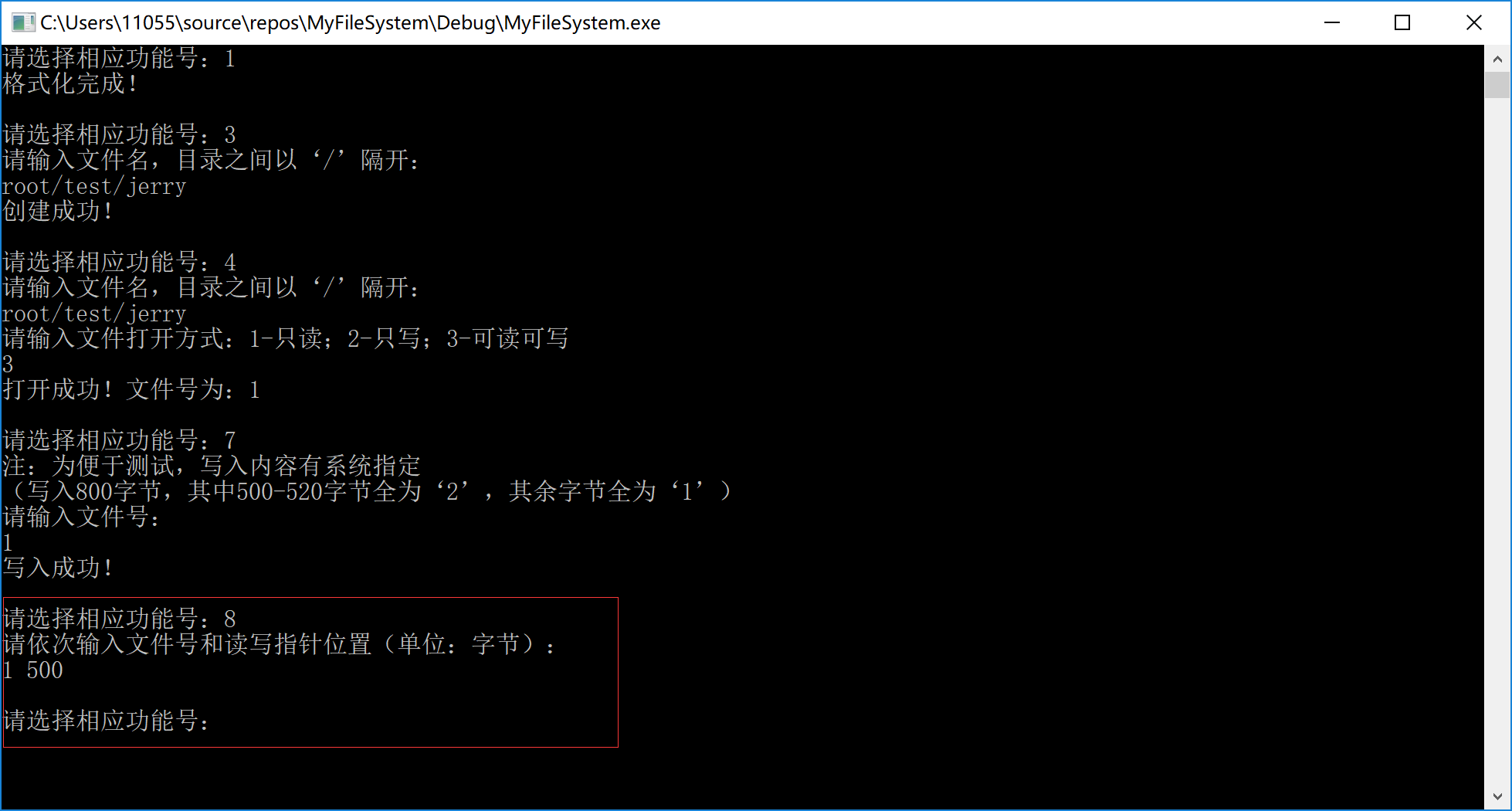
1. 写入800个字节，其中第500-519字节全为‘2’，其余字节全为‘1’；

首先定位到文件的inode，发现filesize为0，要为其申请1个数据盘块（原本已经深申请了一个数据盘块）：

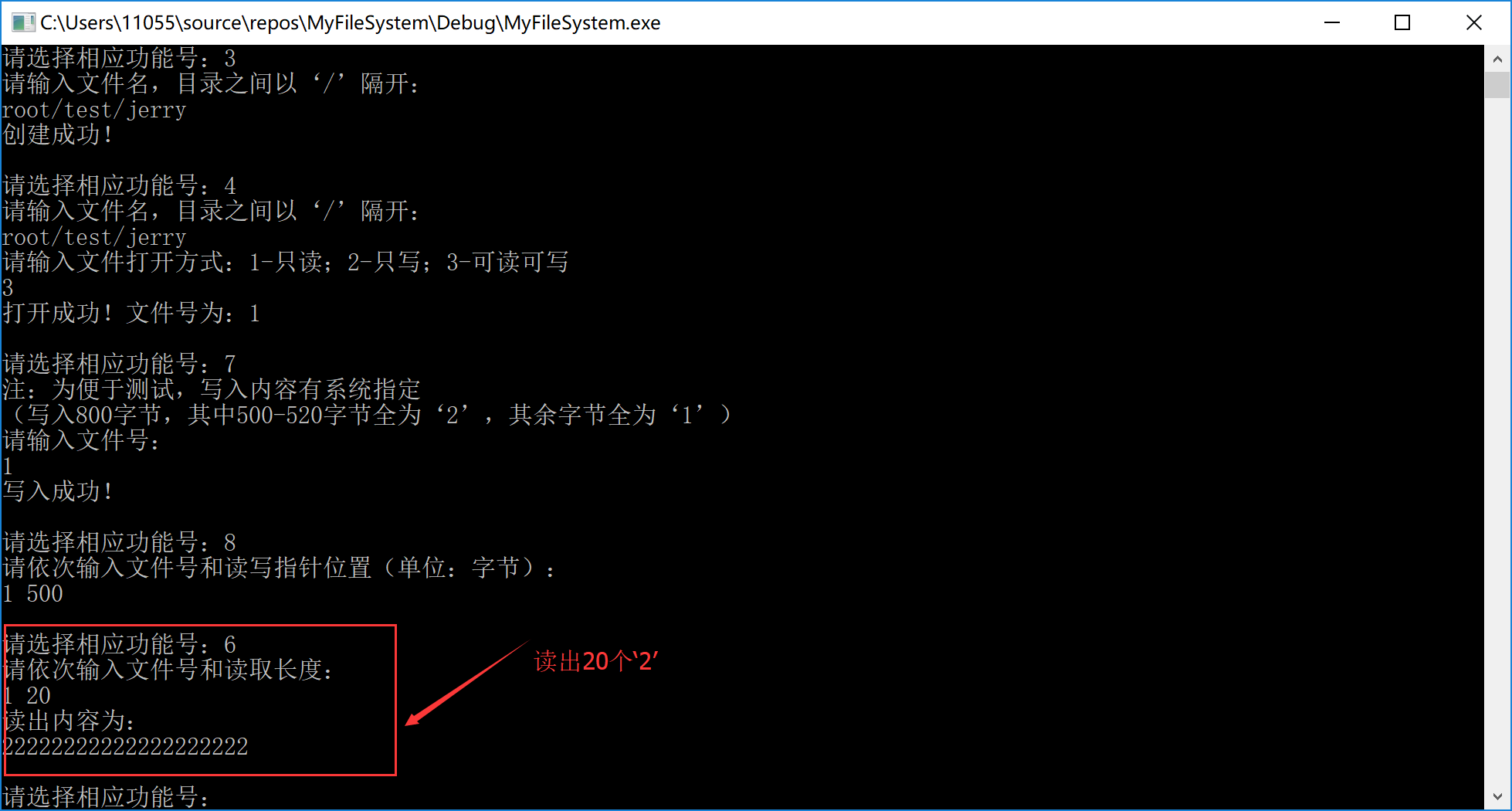




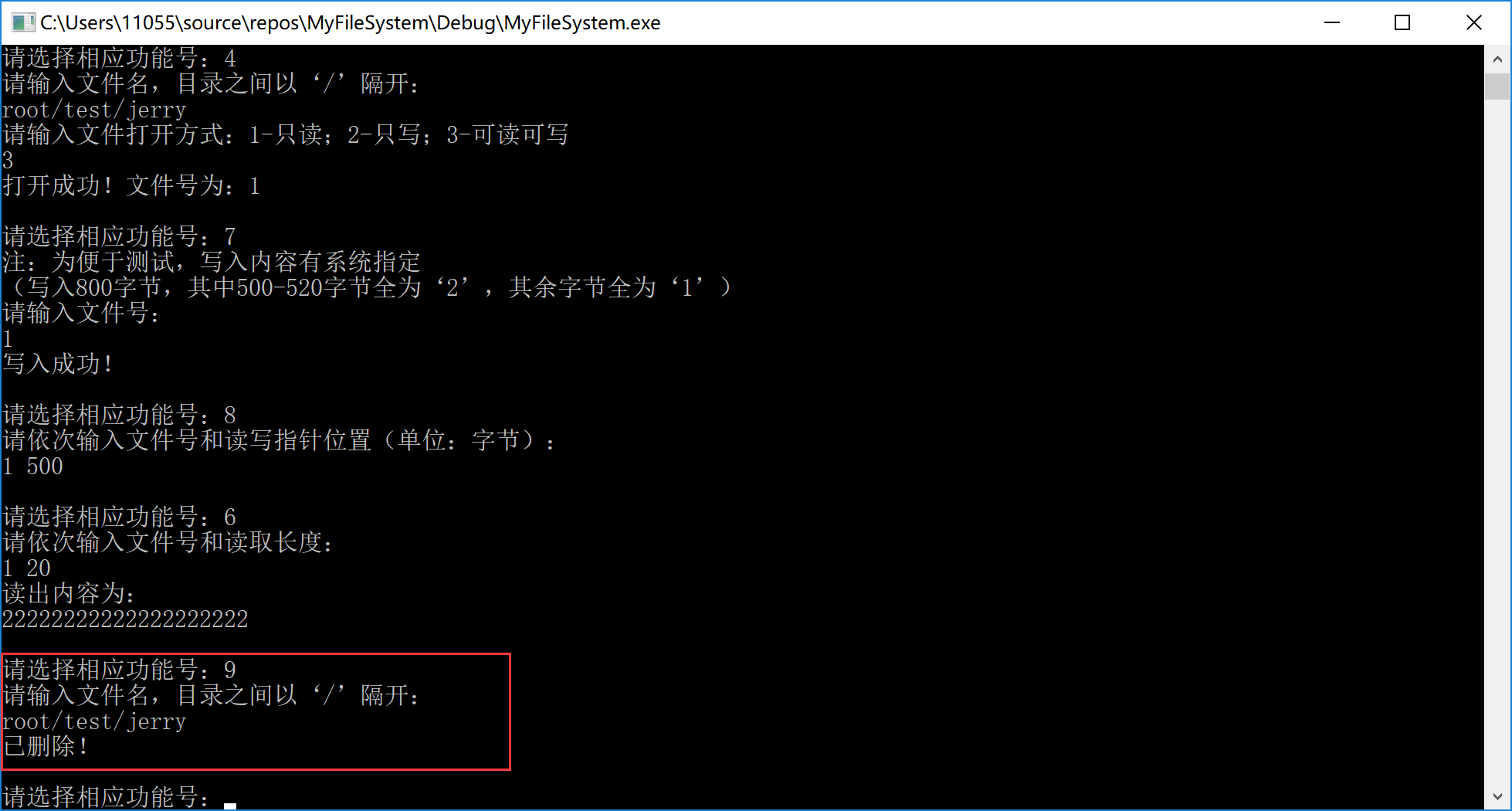
1. 将文件读写指针定位到第500字节；



1. 读出20字节内容



1. 删除文件 root/test/jerry



1. 列目录

此时文件卷中只有两个目录文件：root/test



## 4.3问题

当打开一个文件后，该文件在内存的文件打开表中有相应记录，并给它分配了一个文件号。但是当用户在没有关闭该文件的情况下删除了该文件，在内存中还是有该文件的记录，但是在磁盘中已无该文件，当用户再用内存中的文件号对此文件操作时会出现致命错误。

解决办法是在删除文件的同时，检查该文件中的inode.mode，若mode!=0，表示该文件还处于打开状态，在内存中还有记录，应同时删除内存中的记录，并回收文件号。若mode=0，则表示该文件未打开，直接删除即可。

## 4.4复杂度分析

1. 格式化文件卷：O(n)，n为磁盘盘块规模数
2. 列目录：O(n\*m)，n为文件个数，m为平均每个目录文件的目录个数
3. 创建文件：O(n\*m)，n为目录级数，m为平均每个目录文件的目录个数
4. 打开文件：O(n\*m)，n为目录级数，m为平均每个目录文件的目录个数
5. 关闭文件：O(1)
6. 读文件：O(n)，n为读取的磁盘块数
7. 写文件：O(n)，n为写入的磁盘块数
8. 定位文件读写指针：O(1)
9. 删除文件：O(n\*m)，n为目录级数，m为平均每个目录文件的目录个数

# 5.用户使用说明

用户选择功能号完成相应功能，功能号如下：

* 功能号：1 ——格式化文件卷
* 功能号：2 ——列目录
* 功能号：3 ——新建文件
* 功能号：4 ——打开文件
* 功能号：5 ——关闭文件
* 功能号：6 ——读文件
* 功能号：7 ——写文件
* 功能号：8 ——定位文件读写指针
* 功能号：9 ——删除文件
* 功能号：0 ——退出
* 若选择1，格式化完成后系统会提醒“格式化完成”。
* 若选择2，系统按目录级数分别列出每一级目录下的目录文件和普通文件。
* 若选择3，用户输入文件名（路径名），一级目录必须为root，如root/test/jerry；创建好之后系统会有提醒“创建完成”。
* 若选择4，用户输入文件名（路径名），一级目录必须为root，如root/test/jerry；然后输入打开方式，1-只读；2-只写；3-可读可写；系统给出文件号。
* 若选择5，用户输入文件号，文件号必须是打开文件函数给出的文件号。
* 若选择6，用户依次输入文件号和读取长度，系统打印出读取的内容。
* 若选择7，用户输入文件号（写入内容由系统指定）。写完后系统会提醒“写入成功”
* 若选择8，用户输入文件号和文件读写指针位置。
* 若选择9，用户输入文件名（路径名），一级目录必须为root，如root/test/jerry；删除后系统会提醒“已删除”

# 6.实验总结

本次实验只是设计了一个小型文件系统。所有文件的数据盘块都只采用了一级索引，实现起来比较简单。比较起UNIX V6++的混合索引方式要容易实现地多。设计虽然简单，但是在真正写代码时要考虑的细节还是很多，比如说删除文件中删除最后一级目录文件中相应目录项时，不是简单地将目录项删除即可，要将最后一个目录项填补到删除位置；若最后一级目录项是独占一个数据盘块，填补后还要回收数据盘块；若将要回收地数据盘是第一个数据盘块，则不回收。每一个函数在写代码时都要充分考虑到各种细节。在这里不得不佩服Thompson，在短短时间内写出一个操作系统，还如此完善，考虑到各种细节。

本次实验，让我收获颇丰，让我对UNIX的文件系统有了更加深刻的了解。同时也锻炼了自己的代码能力，算法分析能力和细节考虑能力。

# 7.参考文献

[1]操作系统原理（讲义），同济大学，2017-09

[2] unix/linux文件系统基础知识<https://blog.csdn.net/hzrandd/article/details/50298959>

[3]John Lion，莱昂氏UNIX源代码分析，机械工业出版社

[4]尤晋元，UNIX操作系统教程，西安电子科技大学出版社，1985