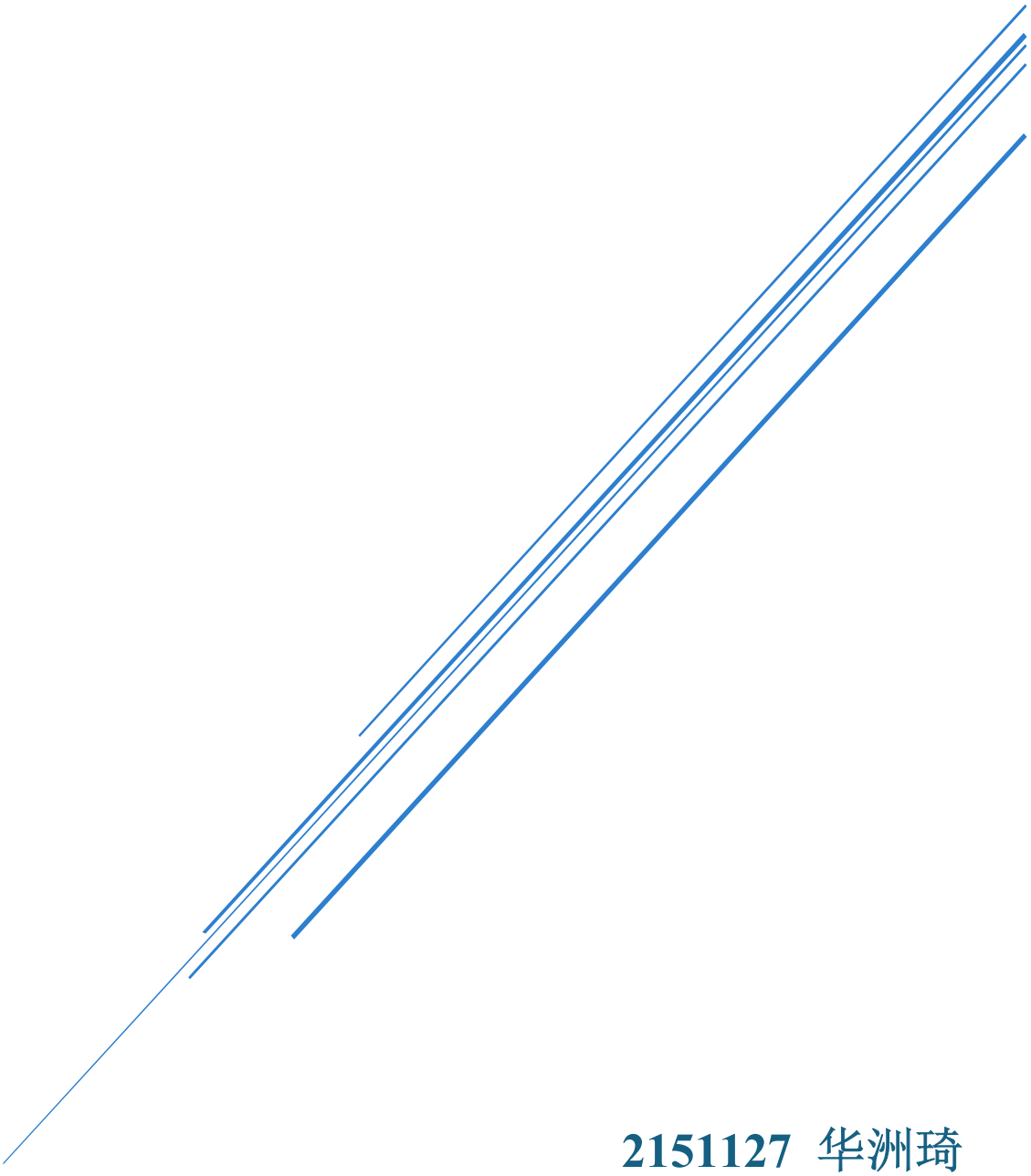
类 UNIX 文件系统

2023-2024 操作系统课程设计



2151127 华洲琦

henryhua0721@foxmail.com

目录

一、设计说明

1.1 实验目的

构造一个类 UNIX 二级文件系统：使用一个普通的大文件（如 c:\myDisk.img ，称之为一级文件） 来模拟 UNIX V6++的一张磁盘。磁盘中存储的信息以块为单位。每块 512 字节。

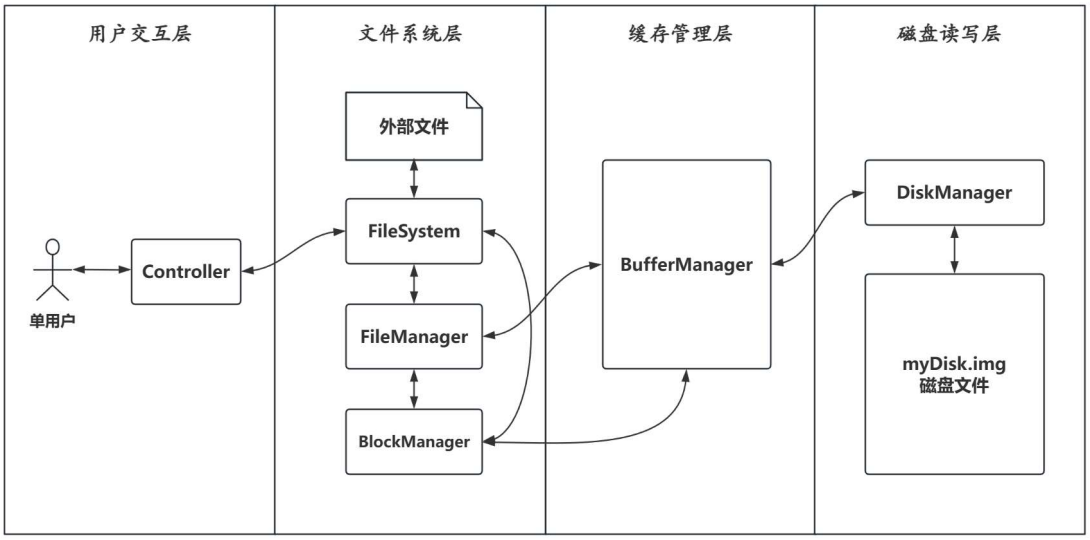
1) 实现对该逻辑磁盘的基本读写操作
   - 地址转换方式设计
   - 高效的缓存队列实现
2) 在该逻辑磁盘上定义二级文件系统结构
   - SuperBlock 及 Inode 区所在位置及大小设计
   - Inode 节点实现
   - SuperBlock 实现
3) 文件系统的目录结构
   - 目录文件结构
   - 目录检索算法的设计与实现
   - 目录结构增、删、改的设计与实现
4) 文件打开结构
   - 文件打开结构设计
   - 内存 Inode 节点的分配与回收
   - 文件打开过程
   - 文件关闭过程
5) 文件操作接口

- fformat: 格式化文件卷
- ls: 列目录
- mkdir: 创建目录
- fcreat: 新建文件
- fopen: 打开文件
- fclose: 关闭文件
- fread: 读文件
- fwrite: 写文件
- flseek: 定位文件读写指针
- fdelete: 删除文件
- 。。。

1.2 设计说明

本人设计的项目为单用户单进程单设备的情况，按照磁盘读写、缓存管理和顶层控制进行模块划分，参考了 UNIX V6++源代码相关数据结构以及 BufferManager 等单元的设计方法。系统分为用户交互、文件系统、缓存管理和磁盘读写四个系统分层，数

据流动和层级关系如下：


用户交互层
文件系统层
缓存管理层
磁盘读写层
外部文件
FileSystem
FileManager
BlockManager
Controller
单用户
BufferManager
DiskManager
myDisk.img
磁盘文件

二、文件系统结构说明

2.1 磁盘划分

磁盘使用 32MB 大小的物理空间存储，题目要求盘块大小为 512 字节，共划分为 65536 个盘块。盘块号和功能如下：

- **[0-199]** 前 200 个盘块保留（reserved）用于特定目的，存储操作系统的引导程序、文件系统元数据。
- **[200-201]** 200-201 盘块被用作超级块（SuperBlock），它存储了文件系统的关键信息（文件系统的类型、大小、盘块大小、空闲盘块的数量等）。
- **[202-1023]** 202-1023 盘块被用作 Inode 区域。Inode（Index Node）是文件系统中的数据结构，用于存储文件的元数据，如文件的权限、所有者、大小、创建时间、修改时间等。每个文件和目录都对应一个 Inode，Inode 区域存储了所有文件和目录的 Inode 信息。
- **[1024-65535]** 1024-65535 盘块是文件数据区域，用于存储实际的文件数据。

磁盘可视化如下：

2.2 SuperBlock 设计

在磁盘中 SuperBlock 占用 200/201 两个盘块，故存储的信息也分为以下两块：

```
class SuperBlock{
public:
    //-------------------- Block-1 -------------------- //
    int     s_isize;        // 外存Inode盘块数
    int     s_fsize;        // 盘块总数
    int     s_nfree;        // 直接管理的空闲盘块数
    int     s_free[100];    // 直接管理的空闲盘块索引表
    int     pad_blk1[25];   // 字节填充

    //-------------------- Block-2 -------------------- //
    int     s_ninode;       // 直接管理的空闲外存Inode数
    int     s_inode[100];   // 直接管理的空闲外存Inode索引表
    int     pad_blk2[27];   // 字节填充
    int     s_fmod;         // 内存中SPB副本被修改标志，即需要更新外存
};
```

其中第一个盘块使用 412 字节，填充 100 字节；第二个盘块使用 404 字节，填充 108 字节，避免了跨两个盘块读取同一段数据的问题。

2.3 Inode 设计

2.3.1 内存 Inode

```
// 内存中的索引节点，记录文件的相关信息
class Inode{
public:
    enum INodeFlag      // 状态标志位，记录内存inode状态
    {
        IUPD = 0x2,     // 内存Inode已经修改标志
        IDIR = 0x8,     // 当前Inode为目录标志
        ITEXT = 0x20    // 当前Inode为正文标志
    };
    static const int DISKINODE_SIZE = 64;                                   // Inode大小
    static const int ADDRESS_PER_INDEX_BLOCK = BLOCK_SIZE / sizeof(int);    // 每个索引表项的大小
    // 对于不同大小文件使用不同级别的索引表（设置访问的逻辑块号的最大值）
    static const int SMALL_FILE_BLOCK = 6;                                  // 小型文件：直接索引表最多可寻址的逻辑块号
    static const int LARGE_FILE_BLOCK = 128 * 2 + 6;                        // 大型文件：经一次间接索引表最多可寻址的逻辑块号
    static const int HUGE_FILE_BLOCK = 128 * 128 * 2 + 128 * 2 + 6;         // 巨型文件：经二次间接索引最大可寻址文件逻辑块号
public:
    unsigned int    i_mode;         // 内存inode是否已经被修改
    int             i_number;       // 外存inode区中的编号
    int             i_size;         // 文件大小，字节为单位
    int             i_addr[10];     // 用于文件逻辑块号和物理块号转换的基本索引表
};
```

- 使用了枚举类型 INodeFlag 来表示内存 inode 的状态标志位
- 区分小文件、大文件、巨型文件最多可寻址的逻辑块号
- 出于设计方便，简化部分功能（i_mode 状态位、删除部分数据结构）

2.3.2 磁盘 Inode

```
// 外存Inode，和内存Inode
class DiskInode{
public:
    unsigned int d_mode;            // 状态标志位
    int     d_size;                 // 文件大小（单位为字节）
    int     d_addr[10];             // 用于文件逻辑块号和物理块号转换的基本索引表
    int     padding[4] = {};        // 填充 -> 64字节
};
```

磁盘 Inode 和内存中的数据结构大体对应，同样使用状态标志位、文件大小、索引表和填充。

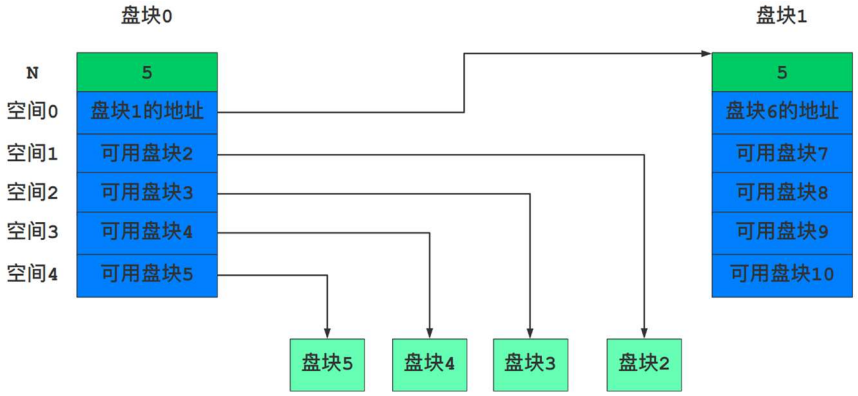
2.4 磁盘管理算法

2.4.1 文件数据区

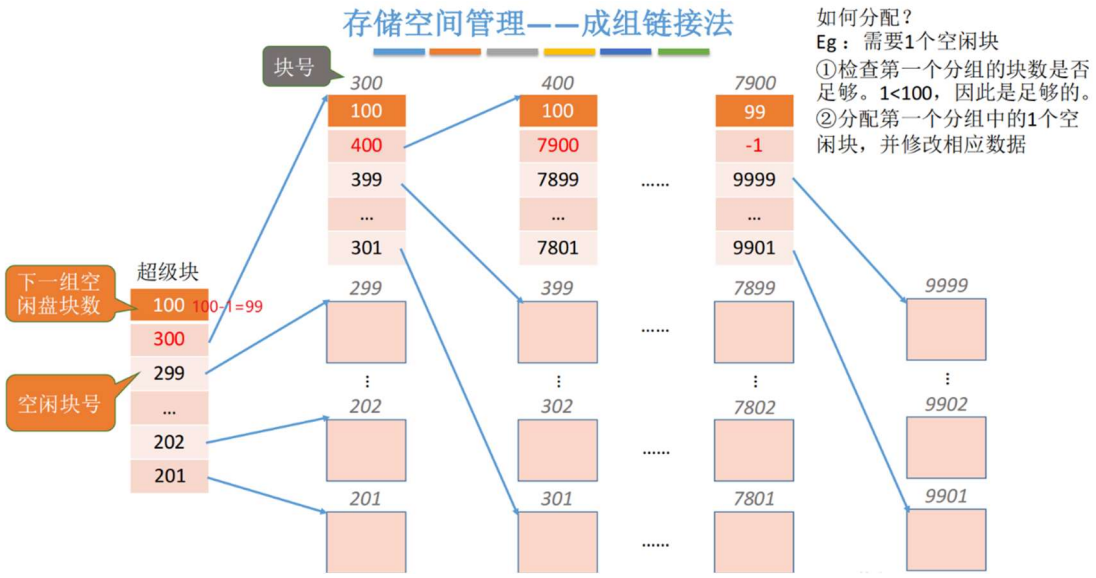
本系统文件数据区使用成组链接法实现。

多个盘块结合为盘块组（Block Group），每个盘块组包含一定数量的盘块，每组第一个盘块作为盘块组的头部（Group Header）用于存储盘块组的元数据信息（空闲盘块的数量、下一个空闲盘块组内盘块号列表）。

两组之间的链接原理如下：



多组的示意图如下：



本系统预设每组 100 个盘块，需要注意的是：

- 最后一个链接块（上图块 7900）需要预留一位用于存放结束标志
- 最后一个盘块写不满，需要调整写入大小

实现代码如下：

```
    // 非最后一个块的写入
    int blkno = FILE_DATA_START;
    memset(buffer, 0, BLOCK_SIZE);
    for (; blkno < BLOCK_SUM; blkno += 100) {
        // 成组链接，第i（1024，1124，1224，…）个盘块存储i+100到i+199共100个盘块
        *((int*)buffer) = 100;
        for (int i = 0; i < 100; ++i) {
            *((int*)buffer + i + 1) = blkno + 100 + i;
        }
        m_BufferManager->write(buffer, blkno);

        // 第一个盘块用于链接，剩余99个盘块全部刷0
        memset(buffer, 0, BLOCK_SIZE);
        for (int i = 1; i < 100; ++i) {
            if (blkno + i >= BLOCK_SUM) // 防止写出边界
                break;
            m_BufferManager->write(buffer, blkno + i);
        }
    }
    blkno -= 100;
```

```
    // 最后一个盘块实际写入的条目数
    int actualBlockNumber = 100 - (blkno - BLOCK_SUM);
    *((int*)buffer) = actualBlockNumber;
    *((int*)buffer + actualBlockNumber + 1) = 0;
    m_BufferManager->write(buffer, blkno);
```

2.4.2 Inode 区

直接使用 SuperBlock 中保留的 100 个空闲 Inode。分配完后从 Inode 区重新寻找 100 个空闲的空闲盘块。

Inode 分配代码实现如下：

```
int BlockManager::allocateInode()
{
    m_superblock.s_fmod = 1;
    int ret = m_superblock.s_inode[--m_superblock.s_ninode];
    if (m_superblock.s_ninode == 0) {    // 当前100个空闲inode均已经分配完毕，遍历inode区获取下100个空闲的inode
        int inodeNo = 0;
        for (int blkno = INODE_START; blkno < FILE_DATA_START; ++blkno) {
            m_BufferManager->read(buffer, blkno);
            for (int i = 0; i < BLOCK_SIZE / sizeof(DiskInode); ++i) {
                // 如果d_mode为零，说明inode块未被占用，收取之
                if (*((int*)(buffer + sizeof(DiskInode) * i)) == 0) {
                    m_superblock.s_inode[m_superblock.s_ninode++] = inodeNo;
                    if (m_superblock.s_ninode == 100) {
                        return ret;
                    }
                }
                ++inodeNo;
            }
        }
    }
    return ret;
}
```

三、目录结构说明

3.1 目录结构设计

本系统目录结构和 Unix v6++相似，使用 32 字节记录一个目录项，其中 28 字节用于记录目录名（同一目录下不允许重名），4 字节用于存放目录对应的 Inode 号。

目录结构示例如下：

[0-27字节]：目录名	[28-31字节]：Inode号
.	0
..	0
dev	78
home	77

一个目录表的第一项和第二项用于存放当前目录和父目录。当表示文件系统 root 根目录时当前目录和父目录相同，如上图。

磁盘格式化时，对于根目录的处理代码实现如下：

```
    // .当前目录为"~"
    root[0] = '.';
    m_BufferManager->write(root, blknoTransform(0, inode), 0, FILE_ITEM_BYTES);
    // ..上一级目录也为自身
    root[1] = '.';
    m_BufferManager->write(root, blknoTransform(0, inode), FILE_ITEM_BYTES, FILE_ITEM_BYTES);
```

3.2 目录搜索

由于上述目录表项的实现是 [28 Bytes]目录名 + [4 Bytes]目录 Inode，故对于指令 ls / cd 等对于目录检索的需求，只需要使用前 28 字节目录名和输入内容匹配即可。

下面的 FileManager::findDirItem 函数是搜索函数的实现：

```
int FileManager::findDirItem(string name)
{   // 将原本的文件偏移保存，等到使用完毕后放回
    const int lastOffset = fileOffset;
    fileOffset = 0;
    // 遍历整个文件，查找路径或文件名
    char buf[FILE_ITEM_BYTES + 1] = {};
    read(FILE_ITEM_BYTES, (uint8_t *)buf, pathInode);  // 前两项跳过
    read(FILE_ITEM_BYTES, (uint8_t*)buf, pathInode);
    int itemOffset = 2;
    while (read(FILE_ITEM_BYTES, (uint8_t*)buf, pathInode) == FILE_ITEM_BYTES) {
        if (buf == name) {
            fileOffset = lastOffset;
            return itemOffset;
        }
        ++itemOffset;
        memset(buf, 0, FILE_ITEM_BYTES + 1);
    }
    fileOffset = lastOffset;
    return -1;
}
```

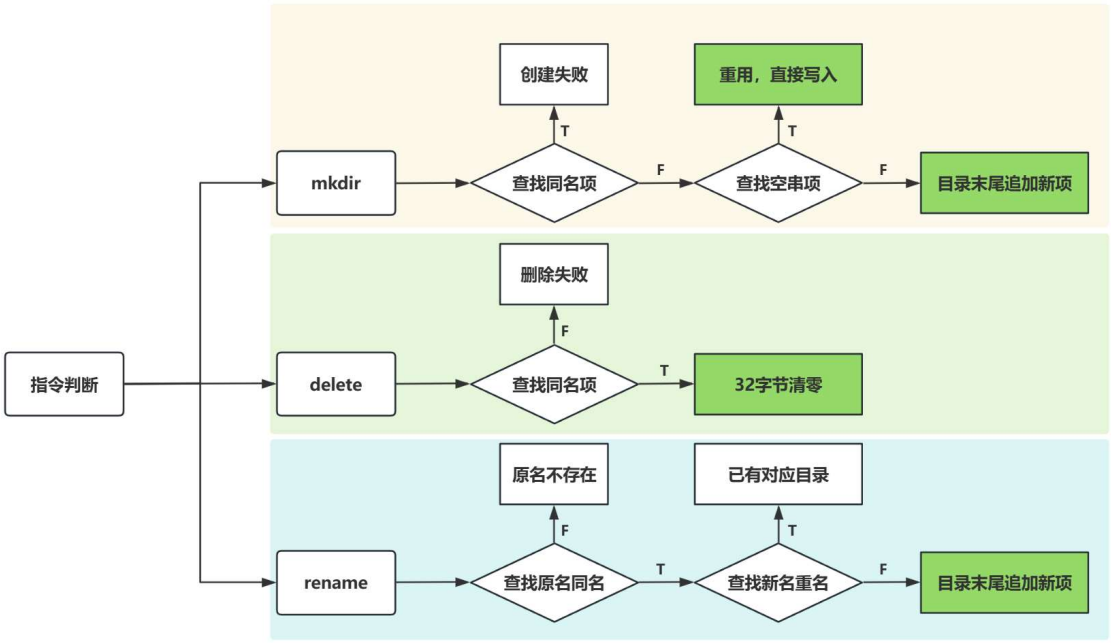
上述函数使用 `itemOffset` 偏移量衡量查找到的目录表项的位置。

3.3 增删改

增删改分别对应指令：

- `mkdir [folder name]`
- `delete [folder name]`
- `rename [old name] [new name]`

都在上述目录检索函数 `findDirItem` 基础上进一步判断并实现，实现逻辑如下：

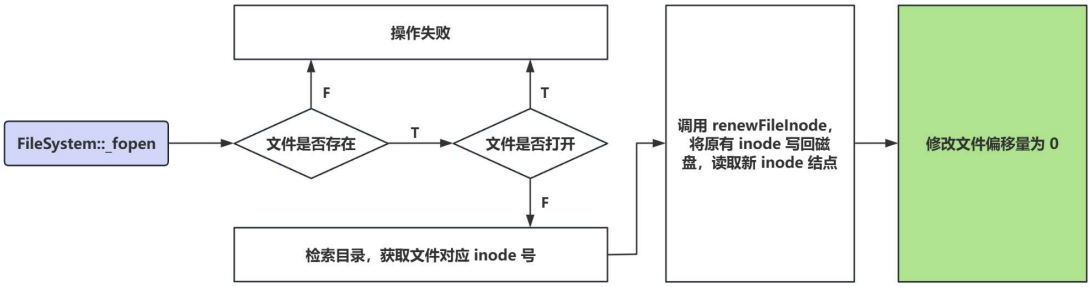


四、文件打开结构

本系统在单用户单线程设计下，文件打开功能较为简单。设计思路为使用全局的 `currentFileName` 记录当前打开的文件名，只需要判断要打开的文件是否存在、是否已经打开即可。

文件打开结构的定义只需要内存中维护文件 `inode` 与路径 `pathInode`，内存 `Inode` 节点数据结构的定义及分配与回收主要依靠 `SuperBlock` 实现

此处重点介绍文件打开函数 `FileSystem::_fopen`，实现逻辑如下：



函数具体实现如下：

```
bool FileSystem::_fopen(string _fileName)
{
    if (_fileName == currentFileName) {
        cout << "[ERROR]: 当前文件已打开！" << endl;
        return false;
    }

    int inodeNo = m_FileManager.searchItemInDirectory(_fileName);
    if (inodeNo == -1) {
        cout << "[ERROR]: 文件不存在" << endl;
        return false;
    }

    bool res = m_FileManager.renewFileInode(inodeNo);
    if (res) {
        m_FileManager.setFileOffset(0);
        currentFileName = _fileName;
    }
    return res;
}
```

五、文件系统实现

5.1 文件读写操作实现

5.1.1 文件写

写文件的逻辑如下，需要注意的是逻辑盘块是否连续：

1) 计算第一次写入的偏移量和长度
2) 检查文件指针指向的逻辑盘块是否与之前连续，若不连续，则需要先分配盘块
3) 循环写入数据，直到所有数据都写入完毕
4) 更新文件大小和 Inode 信息

代码实现如下：

```
214 void FileManager::write(int size, uint8_t* content, Inode &inode)
215 {
216     int offset = fileOffset % BLOCK_SIZE;       // 计算第一次的偏移
217     int blkno = fileOffset / BLOCK_SIZE;        // 计算第一次对应的盘块号（逻辑盘块号）
218     int length = min(size, BLOCK_SIZE - offset);// 计算第一次的长度
219     // 如果文件指针指向的逻辑盘块与之前不连续，需要先给之前分配盘块
220     int borderBlock = inode.i_size / BLOCK_SIZE * BLOCK_SIZE;   // 最后一个盘块对应的序号
221     for (int newBlkno = borderBlock + 1; newBlkno < blkno; ++newBlkno) {
222         blknoTransform(newBlkno, inode);    // 转换时，会针对没有分配的序号进行盘块分配
223     }
224     fileOffset += size;
225     while (size > 0) {
226         // 调用FileManager的地址转换，进行逻辑块号和物理块号的转换
227         if (!m_BufferManager->write(content, blknoTransform(blkno, inode), offset, length)) {
228             cout << "ERROR: Write buffer failed!" << endl;
229             exit(EXIT_FAILURE);
230         };
231         // 更新参数
232         size -= length;
233         content += length;
234         ++blkno;
235         length = min(BLOCK_SIZE, size);
236         offset = 0;         // 由于一次写操作在逻辑上是连续的，因此后续偏移都是0
237     }
238
239     inode.i_size = max(inode.i_size, fileOffset);   // 更新inode中的文件大小
240     inode.i_mode |= inode.IUPD;
241 }
```

5.1.2 文件读

在文件读中，需要注意的是相比于写多了读取数据量和文件中拥有数据量的比较。实现逻辑如下：

1) 检查要读取的数据是否超过文件大小，如果超过则只读取文件剩余部分
2) 计算第一次读取的偏移量和长度
3) 循环读取数据，直到所有数据都读取完毕
4) 更新文件偏移量

文件读写都借助 `blknoTransform` 函数实现物理地址与逻辑地址之间的相互转换。

代码实现如下：

```cpp
int FileManager::read(int size, uint8_t* content, Inode& inode)
{
    if (fileOffset + size > inode.i_size) {
        size = inode.i_size - fileOffset;
    }
    int totalSiz  int FileManager::fileOffset
    int offset =  当前打开文件的偏移量                    // 计算第一次的偏移
    int blkno = fileOffset / BLOCK_SIZE;          // 计算第一次对应的盘块号（逻辑盘块号）
    int length = min(size, BLOCK_SIZE - offset);// 计算第一次的长度
    if (length <= 0) {
        if (length < 0)
            cout << "ERROR: Reading exceeds the file limit!" << endl;
        return 0;
    }
    int actualSize = 0;
    while (size > 0) {
        // 调用FileManager的地址转换，进行逻辑块号和物理块号的转换
        actualSize = m_BufferManager->read(content, blknoTransform(blkno, inode), offset, length);
        totalSize += actualSize;
        fileOffset += actualSize;
        // 更新参数
        size -= actualSize;
        content += actualSize;
        ++blkno;
        // 考虑可能读取超出文件大小的内容
        length = min(BLOCK_SIZE, min(size, inode.i_size - fileOffset));
        offset = 0;          // 由于一次读操作在逻辑上是连续的，因此后续偏移都是0
    }
    return totalSize;
}
```

5.2 文件其他操作实现

5.2.1 文件删除

文件删除的操作本质上并没有“清零”文件内容，而是将物理内存盘块标记为可用即可，后续其他程序可以覆盖原有的数据。（和 Steam 一秒“删除”游戏原理一样）。

实现逻辑上，先获取文件的所有物理盘块号，接着将所有物理盘块号标记为可用，最后将文件的大小设置为 0 并更新 inode 信息即可。

需要注意的是，对于一些从未写入内容的文件，没有分配盘块，因此使用三目运算符进行特判。

```cpp
bool FileManager::deleteFile(Inode inode)
{
    vector<int>blocks;
    int borderBlock = inode.i_size ? inode.i_size / BLOCK_SIZE * BLOCK_SIZE + 1 : 0; // 总盘块数
    for (int i = 0; i < borderBlock; ++i)
    { // 获取所有的物理盘块号
        blocks.push_back(blknoTransform(i, inode));
    }
    memset(buffer, 0, BLOCK_SIZE);
    for (int i = 0; i < blocks.size(); ++i) {
        m_BufferManager->write(buffer, blocks[i]);
        m_BlockManager->recycleBlock(blocks[i]);
    }
    return true;
}
```

5.2.2 文件创建

本系统将文件创建和目录创建使用一个成员函数实现，主要实现逻辑如下：

1) **文件名检查**：检查是否超过 28 字节且不是“.”和“..”两个保留目录项。
2) **重名检查**：查看目录中是否有重名文件/目录

3) **Inode 分配和修改**

- 使用 `m_BlockManager->allocateInode` 分配一个新的 `inode`
- 读取新的 `inode` 信息，将新的 `inode` 的 `i_mode` 设置为 `0`，`i_number` 设置为分配的 `inode` 号
- 将新的 `inode` 的 `i_size` 设置为 `0`，`i_addr` 数组所有元素设置为 `0`

4) **将新的 inode 写入当前目录文件**

5) **写回新的 inode 信息至磁盘**

各部分代码实现如下：

- **文件名检查**

```
298     // 截断，最长28字节
299     if (fileName.size() > 28) {
300         fileName = fileName.substr(0, 28);
301     }
302     if (fileName == ".") {
303         cout << "ERROR: Cannot use . as file or path name" << endl;
304         return false;
305     }
306     else if (fileName == "..") {
307         cout << "ERROR: Cannot use .. as file or path name" << endl;
308         return false;
309     }
310
```

- **重名检查**

```
311     // 首先查找是否有重名，若有则返回false
312     if (searchItemInDirectory(fileName) != -1) {
313         cout << "Creation failed! Name has already existed." << endl;
314         return false;
315     }
316
```

- **Inode 分配与修改**

```
317     // 申请inode结点，清空原有内容，修改i_mode，写回
318     int newInodeNo = m_BlockManager->allocateInode();
319     Inode newInode = readInode(newInodeNo);
320     newInode.i_mode = 0;
321     newInode.i_number = newInodeNo;
322     if (isDir)
323         newInode.i_mode |= newInode.IDIR;
324     else
325         newInode.i_mode |= newInode.ITEXT;  // 模式为文件正文段
326     newInode.i_size = 0;
327     for (int i = 0; i < 10; ++i) {
328         newInode.i_addr[i] = 0;
329     }
330
```

- **Inode 写回（至目录文件和磁盘）**

```
331     // 将这个inode结点写入当前路径文件内
332     int lastOffset = fileOffset;
333     int itemOffset = findDirItem("");
334     // 等于-1，说明没有可以使用的，从尾部开始继续写，否则说明有被删除的部分，则覆盖之
335     fileOffset = (itemOffset == -1) ? pathInode.i_size : itemOffset * FILE_ITEM_BYTES;
336
337     {
338         uint8_t item[FILE_ITEM_BYTES] = {}; // 可以不考虑尾零
339
340         const char* ch = fileName.c_str();
341         for (int i = 0; i < 28 && ch[i] != '\0'; ++i) {
342             item[i] = ch[i];
343         }
344         *((int*)(item + 28)) = newInodeNo;
345         write(FILE_ITEM_BYTES, item, pathInode);
346     }
347
```

六、高速缓存结构设计

6.1 缓存控制块设计

缓存控制块（Cache Control Block，CCB）的作用主要是存储缓存块的占用信息、数据信息和队列信息。

由于本系统设计的是单用户单进程单设备，因此简化了缓存队列，只保留**空闲队列**。

缓存控制块结构如下：

```
// 高速缓存控制块
class Buffer
{
public:
    enum BufFlag{
        B_BUSY = 0x2,   // 占用标志，当缓存块的内容从磁盘中读入后，B_BUSY置1
        B_DELWRI = 0x4, // 延迟写标志
    };
    // 单进程单设备：简化了缓存队列，只保留了空闲队列和I/O请求队列
    uint8_t b_flags = 0;
    Buffer* av_forw = nullptr;
    Buffer* av_back = nullptr;
    int     b_wcount = 0;           /* 需传送的字节数 */
    uint8_t* b_addr = nullptr;      /* 指向该缓存控制块所管理的缓冲区的首地址 */
    int     b_blkno = 0;            /* 磁盘逻辑块号 */
};
```

参数作用如下表：

参数名称	参数作用
b_flags	无符号 8 位整数，用于存储缓存块的标志位： ● B_BUSY：当缓存块的内容从磁盘中读入后，该标志位置 1，表示该缓存块正在被使用 ● B_DELWRI：当缓存块需要延迟写入磁盘时，该标志位置 1，表示该缓存块的内容已经修改，但尚未写入磁盘
av_forw	指向 Buffer 类型的指针，指向当前缓存块的前一个缓存块
av_back	指向当前缓存块的后一个缓存块，和 av_forw 构成空闲缓存队列（双向链表）
b_wcount	整型变量，用于存储缓存块中需要传送的字节数
b_addr	指向 uint8_t 类型的指针，指向缓存块所管理的缓冲区的首地址
b_blkno	整型变量，用于存储缓存块对应的磁盘逻辑块号

6.2 缓存管理类结构

本程序使用一个 BufferManager 类统一管理缓存的信息、分配和使用，主要存储着以下几类信息：

1） 配置参数

- NBUF：定义了缓存控制块和缓冲区的数量，为 15。

2) **数据结构**
   - bufferList：缓存控制块队列的队首结点，用于记录缓存块的访问次序，体现 LRU 算法。
   - m_Buf：缓存控制块数组，大小为 NBUF
   - Buffers：缓冲区数组，大小为 NBUF，每个缓冲区的大小为 BLOCK_SIZE
3) **操作方法**
   - write：将指定内容写入到对应缓存中
   - read：从对应缓存中读取指定内容
   - destroy：析构函数，负责释放资源
4) **辅助方法**
   - moveBlk2Rear：将某一块缓存挪到队尾，实现 LRU 算法
   - searchFreeBlk：寻找一个未被占用的缓存块
   - GetBlk：申请一块缓存，用于读写指定磁盘块
   - Bread：从磁盘读取一个磁盘块到缓存
   - Bwrite：将一个缓存块写入磁盘

代码实现如下：

```
// 缓存管理类
class BufferManager
{
public:
    static const int NBUF = 15;         // 缓存控制块、缓冲区的数量
public:
    BufferManager();
    // 对于盘块blkno，将长度为length的内容content拷贝到对应缓存中（起始偏移为offset），返回是否成功
    bool write(uint8_t* content, int blkno, int offset = 0, int length = BLOCK_SIZE);
    // 对于盘块blkno，从offset偏移读取长度为length的内容到holder中，返回成功读取的字节数
    int read(uint8_t* holder, int blkno, int offset = 0, int length = BLOCK_SIZE);
    void destroy();                     // 读写操作默认读写整个盘块
private:
    void moveBlk2Rear(Buffer* bp);      // 将某一块缓存挪到队尾（LRU算法）
    Buffer* searchFreeBlk();            // 寻找一个未被占用的缓存块，若没有找到，返回nullptr
    Buffer* GetBlk(int blkno);          // 申请一块缓存，用于读写字符块blkno
    Buffer* Bread(int blkno);           // 读一个磁盘块到缓存，blkno为目标磁盘块逻辑块号
    void Bwrite(Buffer* bp);            // 将一个缓存块写入磁盘

private:
    // bufferList 中av_forw指示队尾块，av_back指示队首块
    Buffer bufferList;                  // 缓存控制块队列（队首结点），用来记录各个缓存块的访问次序，体现LRU算法
    Buffer m_Buf[NBUF];                 // 缓存控制块数组
    uint8_t Buffers[NBUF][BLOCK_SIZE];  // 缓冲区数组

    DiskManager myDisk;                 // 磁盘管理类，负责“磁盘”读写操作
};
```

6.3 缓存队列的设计及分配和回收算法

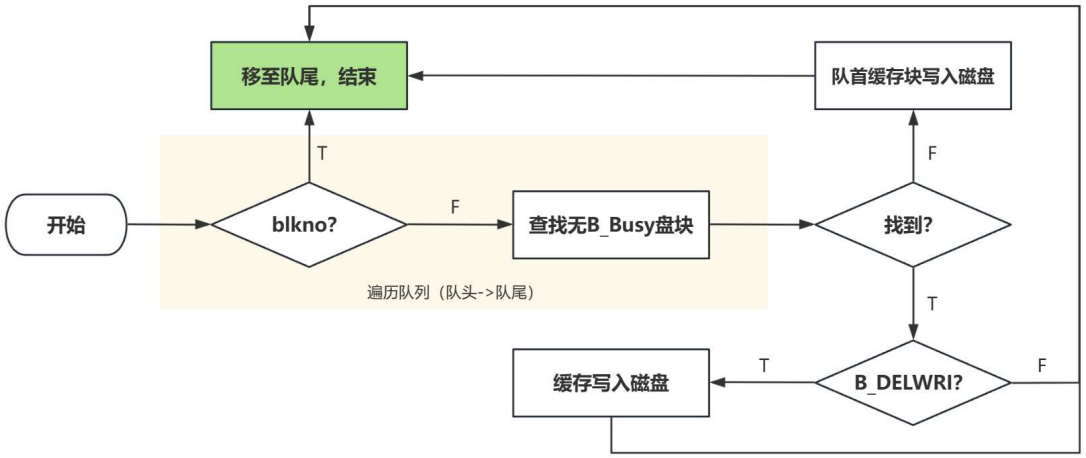
参考 Unix V6++源码，由于多用户多设备的存在，缓存队列使用两组四个队列实现。本系统由于设计为单用户单设备，故使用单独的 LRU 缓存队列实现。

LRU (Least Recently Used) 算法是一种常用的缓存淘汰算法，它根据数据的访问时间进行淘汰，即最近最少使用的数据会被优先淘汰。

实现方式为使用 Buffer 类中的 av_forw 和 av_back 两个指针，分别指向空闲队列的

前一个和后一个缓存块，构成双向链表。一旦某个缓存块被使用，则移动到队列尾部（即队列头部装的是不那么经常被使用的元素）。当缓存队列满需要替换，替换出队列头部元素即可。

实现流程图如下：



代码实现如下，逻辑和上图相同：

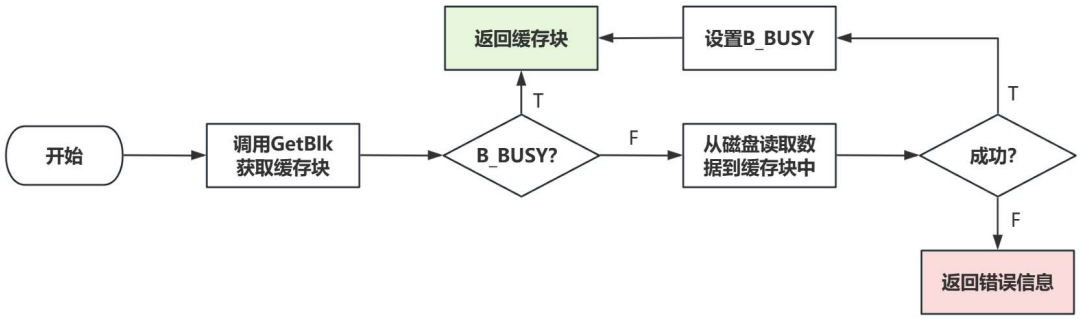
```
Buffer* BufferManager::GetBlk(int blkno)
{
    Buffer* buf = nullptr;
    // 判断这一块字符块是否已经在缓存中（是否可以重用）
    for (int i = 0; i < NBUF; ++i){
        buf = &m_Buf[i];
        if (buf->b_blkno == blkno) {
            // 说明这一块字符块已经在缓存中
            // 因为只有一个进程，所以不需要考虑B_BUSY标志位
            moveBlk2Rear(buf);              // 将这块缓存块放到队尾
            return buf;
        }
    }
    // 这一块字符块不在缓存中，需要申请一块缓存块，将对应字符块从磁盘中换入
    // 首先找一块空闲的缓存块
    buf = searchFreeBlk();
    if (buf && (buf->b_flags & buf->B_DELWRI)) { // 说明找到一块空闲，但带有DELWRI标志，此时将其写入磁盘
        Bwrite(buf);
    }
    else if (buf == nullptr) {
        // 说明没有找到空闲，由于是单进程，即使是sleep也不能等到空闲
        // 因此此处直接抢占一块磁盘（也使用LRU算法，即从队首取下一块）
        buf = bufferList.av_back;
        Bwrite(buf);
    }
    // 说明找到一块空闲且没有延时写标志，可以直接使用
    buf->b_flags = 0;    // 缓存块刷新
    buf->b_blkno = blkno;
    moveBlk2Rear(buf);  // 将这块缓存块放到队尾
    return buf;
}
```

6.4 借助缓存实现对一级文件的读写操作

6.4.1 读文件

本程序借助函数 Bread 实现借助缓存对一级文件读出操作（读单个盘块）。函数接收参数 blkno（要读取的磁盘块的逻辑块号），返回指向缓存块的指针（失败返回 null）。

实现流程如下：



代码实现如下：

```
Buffer* BufferManager::Bread(int blkno)
{
    Buffer* bp = GetBlk(blkno);
    if (bp->b_flags & bp->B_BUSY) {
        // 含BUSY位，说明内容已经换入，无需操作，直接返回
        return bp;
    }
    else { // 不含BUSY位，说明是刚分配的缓存块，需要先读取一下磁盘信息
        if (!myDisk.readDisk(bp->b_blkno, bp->b_addr)) {
            cout << "ERROR: Read disk failed! Block number: " << bp->b_blkno << endl;
            exit(EXIT_FAILURE);
        }
        bp->b_flags |= bp->B_BUSY;  // 读取完毕，置BUSY位
        return bp;
    }
    // 不论是Bread还是Bwrite，都需要先从磁盘中读取信息，因此返回时均有BUSY位
}
```

6.4.2 写文件

写文件和读文件类似，使用函数 Bwrite 实现。实现流程如下：

1) 将缓存块的 b_wcount 变量设置为 BLOCK_SIZE，表示要写入整个缓存块
2) 调用 myDisk.writeDisk 函数将缓存块的内容写入磁盘
3) 如果写入成功，则将缓存块的 b_flags 变量设置为 0，表示缓存块已刷新
4) 如果写入失败，则输出错误信息并退出程序

代码实现如下：

```
void BufferManager::Bwrite(Buffer* bp)
{
    bp->b_wcount = BLOCK_SIZE;
    if (!myDisk.writeDisk(bp->b_blkno, bp->b_addr)) {
        cout << "ERROR: Write disk failed! Block number: " << bp->b_blkno << endl;
        exit(EXIT_FAILURE);
    }
    bp->b_flags = 0;    // 写操作完成后，缓存块刷新
}
```

七、结果展示

7.1 格式化文件卷

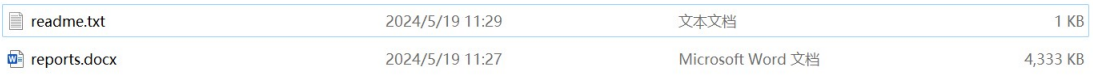
7.2 创建子目录

一级子目录：

二级子目录（home 下）：

7.3 文件存储

将报告 reports.docx 和 ReadMe.txt 放在解决方案目录下：



使用 import 指令存储文件：

```
FS /home/reports> import reports.docx

FS /home/reports> ls
1 items under this directory:
reports.docx

FS /home/reports> size reports.docx
Size of reports.docx: 4436136 bytes.

FS /home/reports>
```

```
FS /home/texts> import readme.txt

FS /home/texts> ls
1 items under this directory:
readme.txt

FS /home/texts> size readme.txt
Size of readme.txt: 147 bytes.
```

从文件大小判断导入成功：

在 photos 文件夹下存储图片：

```
FS /home> cd photos

FS /home/photos> import img.jpg

FS /home/photos> ls
1 items under this directory:
img.jpg

FS /home/photos> size img.jpg
Size of img.jpg: 18513 bytes.

FS /home/photos>
```

将图片重命名后重新导出：

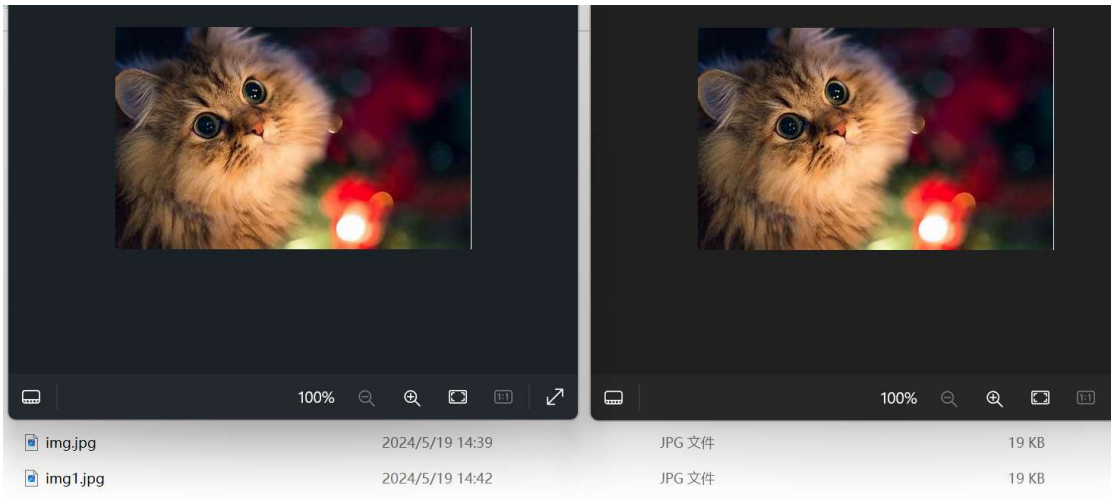
```
FS /home/photos> rename img.jpg img1.jpg

FS /home/photos> ls
1 items under this directory:
img1.jpg

FS /home/photos> export img1.jpg

FS /home/photos>
```

和原图比较，比对通过：



7.4 文件新建和写入

新建 Jerry 文件：

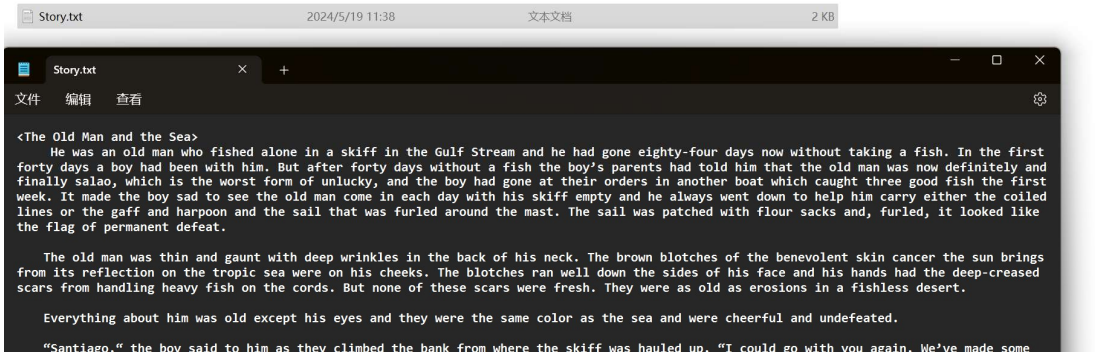
```
FS /> cd test

FS /test> create Jerry

FS /test> ls
1 items under this directory:
jerry

FS /test>
```

采用文件读入，使用外部的 Story.txt（存储老人与海英文原文节选），内容如下：



接着通过从文件写方式，将 800 字节写入 Jerry：

```
FS /test> open Jerry

FS /test(jerry)> fwrite 800 story.txt

FS /test(jerry)> close

FS /test> size Jerry
Size of jerry: 800 bytes.

FS /test>
```

写入成功，可以看到大小为 800 字节，可以选择在控制台输出：

```
FS /test(jerry)> fread 800 *
Read 800 bytes successfully(64 bytes per line):
<The Old Man and the Sea>
    He was an old man who fished alone in a skiff in the Gulf St
ream and he had gone eighty-four days now without taking a fish.
 In the first forty days a boy had been with him. But after fort
y days without a fish the boy's parents had told him that the ol
d man was now definitely and finally salao, which is the worst f
orm of unlucky, and the boy had gone at their orders in another
boat which caught three good fish the first week. It made the bo
y sad to see the old man come in each day with his skiff empty a
nd he always went down to help him carry either the coiled lines
 or the gaff and harpoon and the sail that was furled around the
 mast. The sail was patched with flour sacks and, furled, it loo
ked like the flag of permanent defeat.

    The old man was thin an
```

7.4 文件指针修改和读出

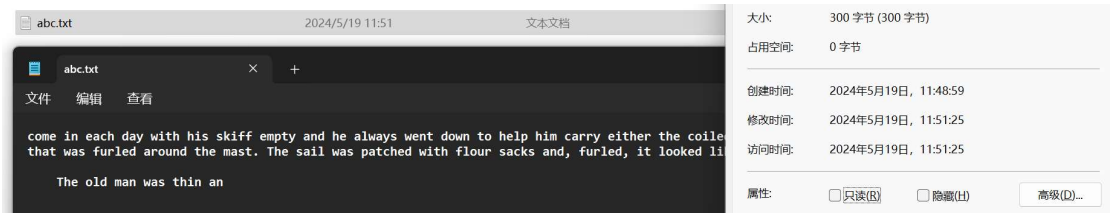
定位到 500 字节，尝试读出 500 字节（但是只能读出 300 字节）：

```
FS /test(jerry)> seek 500

FS /test(jerry)> fread 500 abc.txt
Read 300 bytes successfully.

FS /test(jerry)>
```

abc 被默认导出至文件系统外部，查看 abc 内容和大小：



7.5 容错测试

```
FS /test> open jerry

FS /test(jerry)> open jerry
[ERROR]: 当前文件已打开!

FS /test(jerry)>
```

```
FS /> help help help
help: 命令语法不正确，正确语法为：[help (指令名)]

FS /> cd 1
ERROR: Target path doesn't exist.

FS /> lss home
[Error]: 无法识别的命令 lss
请使用 'help' 命令查看使用手册
```