

华南农业大学信息（软件）学院

《操作系统分析与设计实习》成绩单

开设时间：2024 学年第一学期

小组成员、组内分工、工作量比例、各成员个人成绩									
学号	114514 1919810	姓名	我自己	分工	全部	工作量比例	100%	成绩	
实验题目	模拟磁盘文件系统实现								
各人分工与体会	在此次课程设计中，我结合操作系统课程学习到的知识，模拟实现了一个 FAT 磁盘文件管理系统。通过实现这个磁盘文件管理系统，我深入地了解了文件管理系统的工作原理，提高了程序设计的水平。此外，我还精心设计调试了用户界面，确保程序演示的时候有一个舒适的视觉效果和流畅的操作体验。								
教师评语	<p>评价指标：</p> <div><div>● 题目内容和要求完成情况</div><div>优 <input type="checkbox"/> 良 <input type="checkbox"/> 中 <input type="checkbox"/> 差 <input type="checkbox"/></div><div>● 对算法原理的理解程度</div><div>优 <input type="checkbox"/> 良 <input type="checkbox"/> 中 <input type="checkbox"/> 差 <input type="checkbox"/></div><div>● 程序设计水平</div><div>优 <input type="checkbox"/> 良 <input type="checkbox"/> 中 <input type="checkbox"/> 差 <input type="checkbox"/></div><div>● 程序运行效果及正确性</div><div>优 <input type="checkbox"/> 良 <input type="checkbox"/> 中 <input type="checkbox"/> 差 <input type="checkbox"/></div><div>● 课程设计报告结构清晰</div><div>优 <input type="checkbox"/> 良 <input type="checkbox"/> 中 <input type="checkbox"/> 差 <input type="checkbox"/></div><div>● 报告中总结和分析详尽</div><div>优 <input type="checkbox"/> 良 <input type="checkbox"/> 中 <input type="checkbox"/> 差 <input type="checkbox"/></div></div>								
					教师签名				

## 一、需求分析

用文件模拟磁盘，设计一个文件系统。

### 1. 核心需求

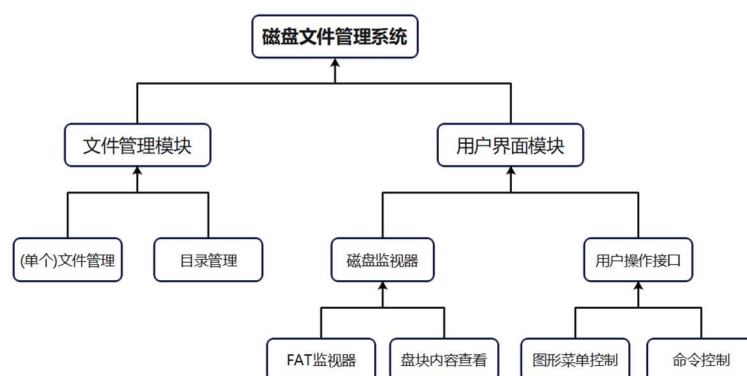
- (1) 使用文件分配表 FAT。
- (2) 用数组模拟缓冲区。
- (3) 定长子目录，支持多级目录结构。
- (4) 支持绝对路径和相对路径。
- (5) 实现对目录的建立、查看、修改、删除操作。
- (6) 实现对文件的建立、打开、关闭、读取、写入（随机写、截短）、权限管理、属性修改、删除操作。

### 2. 其他需求

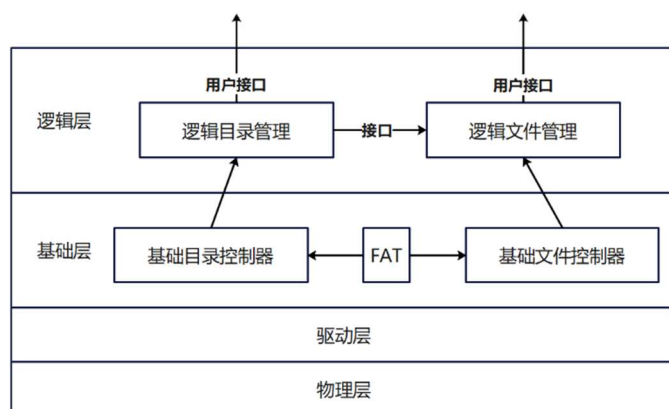
- (1) 图形化用户界面。
- (2) 实现对 FAT 和盘块的实时监视。
- (3) 运行日志记录。
- (4) 独立的命令控制模块，可执行自定义的脚本

## 二、概要设计

### 1. 模块设计



## 2. 文件管理模块层次设计



### (1) 物理层

模拟物理磁盘的真实文件 (disk.data), 文件信息在磁盘上的真实存储结构。

### (2) 驱动层

把对物理盘块的读写请求转换为编程语言支持的真实文件读写方式。

### (3) 基础层

对逻辑层提供服务, 管理 FAT, 依照 FAT 把对逻辑盘块的读写请求转换为对物理盘块的读写请求。

**基础目录控制器:** 目录或文件的建立和初始化、删除、目录项的序列化存储、目录项的读取解析、目录项修改。

**基础文件控制器:** 以盘块为单位读写文件

**FAT (模块):** 管理 FAT, 提供盘块的申请、修改和释放服务。

### (4) 逻辑层

对外提供用户接口, 内部处理用户需求, 管理读写缓冲区。操作的是逻辑盘块, 在逻辑层面对磁盘进行读写。

**逻辑目录管理:** 目录或文件的建立、查看、修改属性、删除、路径解析。

**逻辑文件管理:** 文件的打开、关闭、读取、写入 (随机写、截短)、权限管理。

## 三、详细设计

### 1. 存储结构

磁盘：一个文件（disk.data），划分为 128 块，每块 64 字节。初始化时，第 0 块和第 1 块划分为 FAT，第 2 块划分为根目录，其余块写 0 并置为空闲。

FAT：一共有 128 个字节，每一个字节存储对应块的使用状况（0 表示空闲，3-127 表示占用且记录了后继的盘块，255 表示占用且为最后的盘块，其余值保留不使用）。

目录：一个目录占用一个盘块，划分为 8 条目录项，每条目录项占 8 字节；前两条固定记录当前目录和父目录。

目录项（FCB）：目录项大小为 8 个字节，第 1-3 字节记录目录名或文件名，4-5 字节记录扩展名，第 6 字节记录类型/权限，第 7 字节记录起始盘块号，第 8 字节记录文件长度。

读写缓冲区（r\_buffer、w\_buffer）：大小为 64 字节的 byte 数组，向磁盘读取数据时，先将数据按盘块读入 r\_buffer；向磁盘写入数据时，先将数据写入 w\_buffer，待 w\_buffer 满后，按盘块将 w\_buffer 中的内容写入磁盘。

### 2. 驱动层

#### （1）DiskDriver

功能：调用编程语言提供的文件读写接口。模拟磁盘驱动，操作磁头在磁盘上以盘块为单位读写。

主要方法：

```
/// 读盘块  
public void Write(int blockNum, byte[] buffer);  
/// 写盘块  
public void Read(int blockNum, byte[] buffer);
```

### 3. 基础层的主要类

#### （1）BasicDirController（基础目录控制器）

功能：对应基础层的基础目录控制，向逻辑层提供接口，负责目录或文件的建立和初始化、删除、目录项的序列化存储、目录项的读取解析、目录项修改。

### 主要属性:

```
FAT8 fat8;//文件分配表
DiskDriver diskDriver;//调用驱动层服务
```

### 主要逻辑和方法（伪代码）:

```
/// 格式化磁盘：创建空 FAT，根目录
public void FormatDisk()
{
    调用 DiskDriver，向磁盘写入 8192 字节的 0;
    FAT 的前三字节设为已占用
    调用 DiskDriver 写入 FAT;
    初始化根目录;
}
```

```
/// 初始化一个盘块为目录
private void InitFolderBlock(byte blockNum, byte superioBlockNum)
{
    第一条目录项指向当前目录;
    第二条目录项指向父目录;
    其余目录项初始化为空目录项;
    调用 DiskDriver，写入新目录;
}
```

```
/// 创建文件夹
public void CreateFolder(string name, string typeName, byte superioFolderBlockNum, byte
dirEntrieNum)
{
    向 FAT 申请空盘块;
    if (FAT 已满)
    {
        抛异常;
    }
    新建目录项并序列化;
    初始化新目录;
    调用 DiskDriver，写入新目录;
    向 FAT 报告占用申请到的盘块;
}
```

```

/// 创建文件
public void CreateFile(string name, string extendName, byte superioFolderBlockNum, byte
fileType, byte dirEntrieNum)
{
    向 FAT 申请空盘块;
    if (FAT 已满)
    {
        抛异常;
    }
    新建目录项并序列化;
    调用 DiskDriver, 向文件第一个盘块写 0;
    调用 DiskDriver, 写入文件所属目录项;
    向 FAT 报告占用申请到的盘块;
}

```

```

/// 删除目录项
public void DeleteDir(byte beginBlockNum, byte superioFolderBlockNum)
{
    for(在父目录中查找匹配的项)
    {
        if (匹配)
        {
            在父目录中将属于此目录的目录项设为空;
            向 FAT 释放所属盘块;
            return;
        }
    }
    未找到, 抛异常;
}

```

```

/// 删除指定起始盘块的文件
public void DeleteFile(byte beginBlockNum, byte superioFolderBlockNum)
{
    获得目标文件起始盘块的迭代器 FAT_Iterator 实例;
    while(FAT_Iterator 迭代返回值 != 255)
    {
        向 FAT 释放迭代所得盘块;
    }
    删除文件所属目录项;
}

```

## (2) BasicFileController (基础文件控制器)

功能：对应基础层的基础文件控制，向逻辑层提供接口，负责以盘块为单位读写文件。

主要属性：

```
FAT_Iterator fat_Iter;//目标文件起始盘块的迭代器
byte beginBlockNum;//所属文件起始盘块号
DiskDriver diskDriver;//调用驱动层服务
```

主要逻辑和方法（伪代码）：

```
/// 读取，把逻辑盘块号转换为物理盘块号
public void Read(byte logicBlockNum, byte[] buffer)
{
    盘块迭代器重置；
    迭代器迭代到目标逻辑盘块的物理盘块号；
    diskDriver.Read(targetBlockNum, buffer);
    调用 DiskDriver，读取目标物理盘块；
    return 目标物理盘块内容；
}
```

```
/// 写入，把逻辑盘块号转换为物理盘块号
public void Write(byte logicBlockNum, byte[] buffer)
{
    盘块迭代器重置；
    迭代器迭代到目标逻辑盘块的物理盘块号；
    if(物理盘块号存在)
    {
        调用 DiskDriver，向目标物理盘块写入；
    }
    else
    {
        向 FAT 申请空盘块；
        if (FAT 已满)
        {
            抛异常；
        }
        向 FAT 报告占用申请到的盘块；
        FAT 把旧的末尾盘块接上新申请的盘块；
        调用 DiskDriver，向目标物理盘块写入；
    }
}
```

```

/// 截取文件长度到指定值
public void CutLenth(byte lenth)
{
    盘块迭代器重置;
    迭代器迭代到目标长度处的物理盘块号;
    FAT 中目标盘块记为结尾;
    FAT 释放往后的盘块;
}

```

### (3) FAT8 (文件分配表)

功能：管理盘块分配，提供修改接口

主要属性：

```
byte[] fat8Array = new byte[128]; // 盘块分配记录值
```

主要方法：

```

/// 寻找空盘块
public byte FindEmptyBlock();

/// 占用此盘块并赋值
public void UseBlock(byte offset, byte value);

/// 释放该盘块
public void FreeBlock(byte offset);

/// 统计剩余空盘块数量
public byte GetEmptyBlockCount();

```

### (4) FAT\_Iterator (物理盘块迭代器)

功能：根据 FAT 获得连续的物理盘块编号。

主要属性：

```

public static FAT8 fat; // 文件分配表

byte index; // 起始盘块号
byte iterator; // 迭代时的盘块号

```



#### 主要方法:

```
///构造器
public FAT_Iterator(byte index);

/// 获得下一个盘块编号
public byte Next();

/// 从 iterator 指向的盘块开始, 连续释放盘块直到文件末尾
public void FreeBehind();

/// 重置盘块号流到初始盘块号
public void Reset();
```

## 4. 逻辑层的主要类

### (1) LogicDirManager (逻辑目录管理)

功能: 对应逻辑层的逻辑目录管理, 提供用户接口, 负责目录或文件的建立、查看、修改属性、删除、路径解析; 用户当前路径的载体。

#### 主要属性:

```
static public byte currentFolderBlockNum ;//记录当前用户所在的目录所属的逻辑盘块号
static FCB[] currentDirTable; //记录当前目录的所有目录项
static public string currentAbsPath; //记录当前路径
```

#### 主要逻辑和方法 (伪代码):

```
/// 在当前目录创建文件夹
static public void CreateFolder(string name)
{
    name 格式检查;
    if(文件已存在 or 文件夹满)
    {
        抛异常;
    }
    调用 BasicDirController, 创建目录;
}
```

```

/// 创建文件
static public void CreateFile(string fullName, byte fileType)
{
    name 格式检查;
    if(文件已存在 or 文件夹满)
    {
        抛异常;
    }
    调用 BasicDirController, 创建文件;
}

```

```

/// 删除当前目录的目标文件夹，递归删除目标文件夹下面的文件夹和文件
static public void DeleteFolder(string name)
{
    try
    {
        DeleteFolder_Recursion(name);
    }
    catch (Exception e)
    {
        if (没有回到操作发起的目录)
        {
            回溯到操作发起的目录;
        }
        抛异常;
    }
}

static private void DeleteFolder_Recursion(string name)
{
    尝试进入目标目录 name, 失败返回;
    调用 BasicDirController, 读入当前目录的所有目录项;
    foreach (遍历当前目录的所有目录项)
    {
        if(目录项指向目录)
        {
            DeleteFolder_Recursion(目录项指向的目录); //递归
            删除目录项;
        }
        else
        {
            if(文件已经打开)
            {
                抛异常;
            }
        }
    }
}

```

```

        }
        调用 BasicDirController，删除目录项指向的文件；
    }
}
返回上级目录;
}

```

```

/// 判断路径是否存在（路径解析）
static public byte QueryPath(string path, bool isNeedStay = false)
{
    解析 path，得到目录名表；
    for(folderName: 遍历目录名表)
    {
        在当前目录中查找 folderName；
        if (folderName 不存在)
        {
            return 255;
        }
        进入 folderName 指向的目录；
    }

    if(不需要留在目标目录)
    {
        回溯到操作发起时的目录；
    }
    return 查找到的目标路径目录的所在逻辑盘块号；
}

```

```

/// 在当前目录寻找文件（夹）并创建 OFTLE
static public OFTLE Get_OFTLE_ByName(string fullName)
{
    检查并解析 fullName；
    for (遍历 currentDirTable)
    {
        if (匹配)
        {
            if (文件已打开)
            {
                抛异常；
            }
            OFTLE oftle = new OFTLE(匹配的目录项);
            return oftle;
        }
    }
}

```

```

    }
    抛异常;
}

```

## (2) LogicFileStream (逻辑文件管理)

功能：对应逻辑层的逻辑文件管理，提供用户接口，负责文件的打开、关闭、读取、写入（随机写、截短）、权限管理。

每个打开的文件，对应一个 LogicFileStream 的实例。

**主要属性：**

```

public OFTLE oftle;//打开文件信息表
public bool isOpen = false;//文件是否打开
private byte[] r_buffer;//读缓冲区
private byte[] w_buffer;//写缓冲区

```

**主要逻辑和方法（伪代码）：**

```

/// 构造器，获得 LogicFileStream 的实例，即打开文件
public LogicFileStream(OFTLE oftle)
{
    设置状态为打开;
    读取 oftle;
    new 基础层的 BasicFileController 实例;
    初始化读写缓冲区;
    读写指针 = 0;
    读写盘块号 = 0;
    调用 BasicFileController，读写缓冲区读入文件的第一个盘块;
}

```

```

/// 写文件（一个字节）
public void Write(byte data)
{
    if(文件为只读)
    {
        抛异常;
    }

    写缓冲区记录 data;
    写指针++;
    if(写指针超出缓冲区)
    {

```

```

        调用 BasicFileController，当前缓冲区内容写回磁盘；
        清空写缓冲区；
        写指针 = 0；
        写盘块号++；
        if(写盘块号>文件当前大小)
        {
            文件当前大小++；
        }
        else
        {
            调用 BasicFileController 读取下一个盘块内容到写缓冲区；
        }
    }
}

```

```

/// 读文件（一个字节）
public byte Read()
{
    byte data = 读指针指向的读缓冲区；
    读指针++；
    if(读指针超出缓冲区)
    {
        读指针 = 0；
        读盘块号++；
        if(读盘块号>文件当前大小)
        {
            读盘块号--；
            读指针 = 63；
            return 0；
        }
        调用 BasicFileController 读取下一个盘块内容到读缓冲区；
    }
    return data；
}

```

```

/// 读指针跳到指定位置（从文件头开始，按字节）
public void R_Seek(int offset)
{
    if(offset 在文件大小范围内)
    {
        解析 offset 到读指针和读盘块号；
        调用 BasicFileController 读取对应盘块内容到读缓冲区；
    }
}

```

```
    else
    {
        抛异常;
    }
}
```

```
/// 写指针跳到指定位置（从文件头开始，按字节）
public void W_Seek(int offset, bool isNeedWriteBack = true)
{
    if (offset 在文件大小范围内)
    {
        调用 BasicFileController，当前缓冲区内容写回磁盘；
        解析 offset 到写指针和写盘块号；
        调用 BasicFileController 读取对应盘块内容到写缓冲区；
    }
    else
    {
        抛异常;
    }
}
```

```
/// 写指针跳到文件尾部（最后一个盘块的最后一个字节）
public void W_SeekEnd()
{
    写指针 = 63;
    写盘块号 = 文件大小;
    调用 BasicFileController 读取对应盘块内容到写缓冲区;
}
```

```
/// 截短文件，截短后读指针和写指针跳到文件开头
public void CutLenth(byte lenth)
{
    if(lenth 在文件大小范围内)
    {
        文件大小 = lenth;
        调用 BasicFileController 进行截短;
        W_Seek(0, false);
        R_Seek(0);
    }
    else
    {
        抛异常;
    }
}
```

```

    }
}

```

```

/// 关闭文件
public void Close()
{
    if(isOpen)
    {
        try
        {
            调用 BasicFileController, 当前缓冲区内容写回磁盘;
        }
        catch (Exception e) {}
        basicFileController.Close();
        oftle.Close();
        isOpen = false;
    }
}

```

### (3) OFTLE (已打开文件表)

功能：记录打开文件的相关内容。(目录页也视为文件，但不提供编辑接口)

主要属性：

```

static int openingFileCount = 0; //已打开文件数
static List<OFTLE> openedOFTLEs = new List<OFTLE>(); //已打开文件列表

bool isThisOpen; //是否已打开
public string fullName; //文件全名
public string absPath; //文件的绝对路径
public byte beginBlockNum; //文件起始物理盘块号
public byte size; //文件大小
public FCB fcb; //文件所属目录项
public FileAccess access; //文件访问权限
public FilePointer r_pointer; //读指针
public FilePointer w_pointer; //写指针

```

主要逻辑和方法 (伪代码)：

```

/// 构造器
public OFTLE(FCB fcb)
{
    if (已打开文件数 > 4)

```

```

    {
        抛异常;
    }

    状态设为打开;
    解析 fcb 到各个属性;

    已打开文件数++;
    已打开文件列表.Add(this);
}

```

```

/// 关闭文件，释放许可
public void Close()
{
    调用 BasicDirController，把 fcb 写入磁盘;
    if(打开状态)
    {
        已打开文件数--;
        打开状态 = false;
        已打开文件列表.Remove(this);
    }
}

```

#### (4) FCB（文件控制块/目录项）

功能：解析、记录目录项，提供序列化方法。

**主要属性：**

```

public string name;//名称
public string extendName;//拓展名
public byte dirType;//类型/权限
public byte beginBlockNum;//指向的文件（夹）起始物理盘块号
public byte fileSize;//文件大小
public byte FCB_blockNum; // 该 FCB 存放在的盘块号
public byte FCB_lineNum; // 该 FCB 在所在的盘块中的第几行

```

**主要逻辑和方法（伪代码）：**

```

/// 解析字节数据为目录项内容
public static FCB Parse(byte[] data, byte FCB_blockNum, byte FCB_lineNum)
{
    FCB ret = new FCB();
    把数组 data 中的目录项数据解析到 ret 中;
}

```



```

return ret;
}

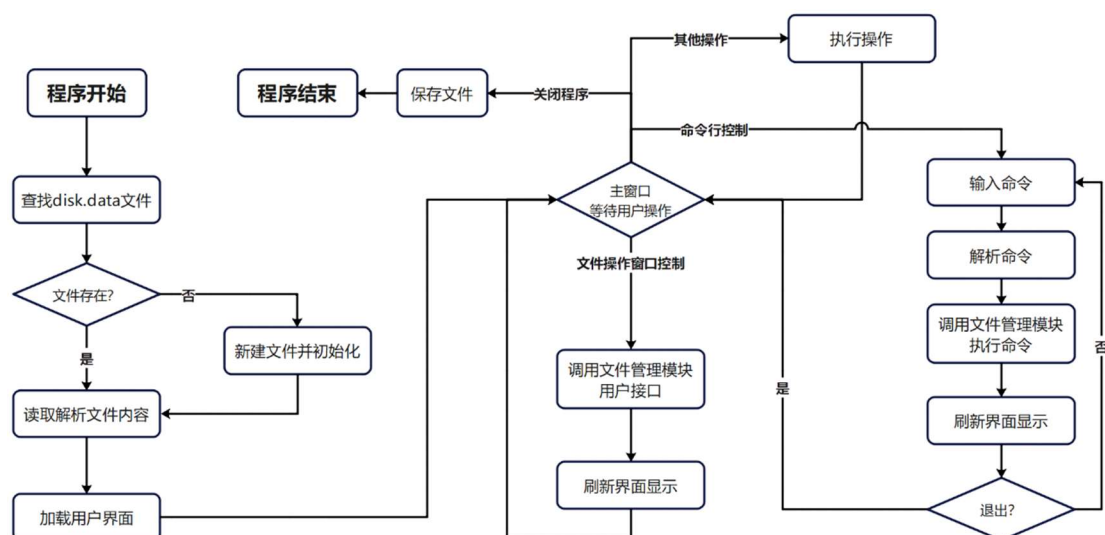
```

```

/// 把 FCB 实例转换为可直接写入磁盘的字节数组
public byte[] ToByteArray()
{
    检查名称合法性;
    if (名称非法)
    {
        抛异常;
    }
    执行编码;
    return 编码结果;
}

```

## 5. 用户主程序流程



## 四、调试分析

### 1. 设计与实现的讨论和分析

#### (1) 文件管理模块的功能划分

在一开始编写文件管理模块的时候，并没有仔细划分目录管理模块和单文件管理模块负责的任务。在编写打开文件的功能时，遇到了一个问题：打开文件的操作应该分到哪里？如果分到单文件管理模块，则无法依据目录信息获得文件；如果分到目录管理模块，则难以清晰区别各个文件个体。在经过思考后，我将打开文件的操作拆分为两个步骤。第一步先从目录管理模块获得需要打开的文件信息，第二步把文件信息传给单文件管理模块，由单文件管理模块管理文件的读写。

后面我又仔细参考了现有成熟编程语言的文件操作接口，明确了目录管理模块和单文件管理模块负责的任务。目录管理模块设置为一个静态类，因为大多数情况下这些操作是独立的。单文件管理模块负责文件打开后的操作，打开文件会新建一个属于该文件的管理实例（文件流），在该文件流上实现对文件的读写。

#### (2) 文件管理模块的层次设计

在完成了功能划分后，我继续编写各个功能的用户接口和内部逻辑时，发现有很多功能都混杂到了一个函数中；例如新建文件的操作，包含目录容量检查、文件名称合法性检查、FAT 盘块的申请占用、数据写入磁盘等操作。如果将各个步骤写在一起，模块就会变得十分臃肿，逻辑难以区分清晰。

经过思考，我发现系统对逻辑结构的管理和对物理结构的管理是相对独立的，于是我将各个功能划分到了“逻辑层”和“基础层”中。逻辑层只能看见文件系统中的逻辑结构（目录项含义、目录树结构、文件处于连续的逻辑盘块号），不会参与物理结构的管理。基础层只能看见文件系统中单个盘块，而不知晓其中的含义，不会参与逻辑结构的建设。逻辑层负责操作逻辑盘块，以逻辑盘块号向基础层申请服务。基础层接收逻辑层的逻辑盘块操作申请，依据 FAT 转换为对物理盘块的操作。

这样的层次结构设计不仅使得每个模块的职责得以划分明确，还使得系统获得了一定的可扩展性。例如，借助驱动层，我可以在不改变文件系统逻辑的情况下，轻松更换底层的存储介质，如果将来想要将文件系统迁移到真正的物理磁盘上，只需要修改驱动层的实现即可，而不需要改动上层的代码。

### （3）磁盘状态监视

在设计初期，我就确定了实现对磁盘三种层次的监视：磁盘占用量监视、FAT 监视、单盘块内容查看。

FAT 监视模块：

我一开始的想法是给 FAT 每个单元都绘制一个方格，方格上写上所属盘块的编号和记录值，占用的盘块设为红色，空闲盘块设为绿色。在后来的开发中，我发现简单的两种颜色并不能直观地显示文件在磁盘中的分配情况，于是改为使用灰色表示空闲盘块，随机的彩色表示盘块占用，且属于同一个文件的盘块分配相同的颜色；修改后通过分辨不同的颜色就能清楚地看见文件在磁盘的存储结构。（在开发者选项中，有一个功能是当鼠标悬停在 FAT 监视器上时，属于同一个文件的方块会同时高亮显示，因为效果不是很好所以没有放出来）。

单盘块内容查看模块：

鼠标点击 FAT 监视模块上的单元格，就能查看单元格对应盘块的具体内容，分别以十六进制和 ASCII 格式显示在模块上，能够清楚地了解磁盘上每一个 bit 是情况。设计这个模块不仅是为了增加本系统的可玩性，更重要的是我能够在调试文件管理功能时随时查看写入磁盘的内容，省去了很多麻烦。

### （4）命令控制模块

一开始的设计并没有这个模块，后来在调试时发现有时候要创建复杂的目录结构或者填满磁盘，而这些操作每次都要手动创建文件和文件夹，十分麻烦。于是我设计了一套命令和命令解析执行器，目标是让系统能够自动执行外部文件上预设的命令脚本。后来我把命令内容和解析执行的部分完善好，加入执行结果反馈，设置了一个用户入口，让用户能够在系统内使用命令控制系统。

## 2. 调试过程中遇到的问题及解决方法

### （1）文件部分内容被 0 覆盖问题

在调试写指针的 Seek 功能时，发现指针在文件中间写入后，文件编辑器中无法显示后续的文本内容。检查盘块内容发现盘块中写指针未达到的地方全部被 0 覆盖。检查代码发现每次 Seek 操作后，写缓冲区会清空，当写指针在缓冲区未全部写入就执行盘块写回时，会把缓冲区空白的部分直接覆盖上去，导致原内容被 0 覆盖。

解决方法：每次 Seek 操作后，缓冲区不清空，而是读取 Seek 位置的盘块到缓冲区，这样当缓冲区写回时就不会改变原内容。

## （2） 目录名无法匹配问题

在进行目录项名称匹配时，出现明明目标存在，但是匹配不上的问题。通过断点调试发现读取解析的目录项的名称 `string` 结尾存在未预期的 `'\0'` 字符，继续调试发现，`'\0'` 字符的出现是由于用于存储名称数据的 `byte` 数组存在 0 值，在调用解码模块时，解码模块没有忽略 0 值，而是解码为了 `'\0'`。目录名存储长度是固定的 5 字节（3 字节名称，2 字节拓展名），当设定的名称不足完整长度时，序列化后的 `byte` 数组后面会填充 0。读取名称数据时，后面的 0 也会被读入存储名称数据的 `byte` 数组中。

解决方法：每次解码目录名后，强制清除 `string` 中的 `'\0'`。

## （3） UI 刷新无响应问题

当用户短时间执行很多操作（如执行命令脚本）时，UI 刷新会进入未响应状态，等待一段时间操作执行结束后，UI 才会一次性刷新，这个现象不利于用户实时监控状态。经搜索学习后了解到，WinForm 的 UI 线程被长时间运行的操作阻塞时，会忽略窗口的重绘请求，导致 UI 无响应。

解决方法：在需要刷新 UI 的位置，调用 `Application.DoEvents()`，这个方法会强制 UI 线程响应重绘请求。

# 五、实验总结与心得体会

在本次课程设计中，我独立完成了整个系统的设计和实现，从设计构思开始，整个项目耗时约 3-4 周。从上学期课程上了解到需要完成课程设计的消息后，我就萌生出了很多有趣的想法，于是当老师正式布置本次课程设计的任务后，我就马上就开始了设计编写工作。整个项目分为了三个时间段完成，6 月时完成了核心逻辑的规划和程序的基本框架；8 月暑假时完成了核心逻辑的编写和调试，9 月开学后完善了用户界面和程序的整体调试工作。

```
*****
*****
*****
*****
*****
*****
*****
*****
```

\*\*\*\*\*

\*\*\*\*\*(github 版已删除部分)。

总而言之，在完成课程设计的过程中，我巩固了上学期所学的操作系统知识，对文件系统有了更为深入的认识，提高了程序设计水平，这是一段难忘的经历。

## 六、使用说明

### 主界面





## 1. 文件操作区

双击文件图标可打开文件编辑窗口，双击文件夹图标可进入文件夹。

在空白区域右键可选操作



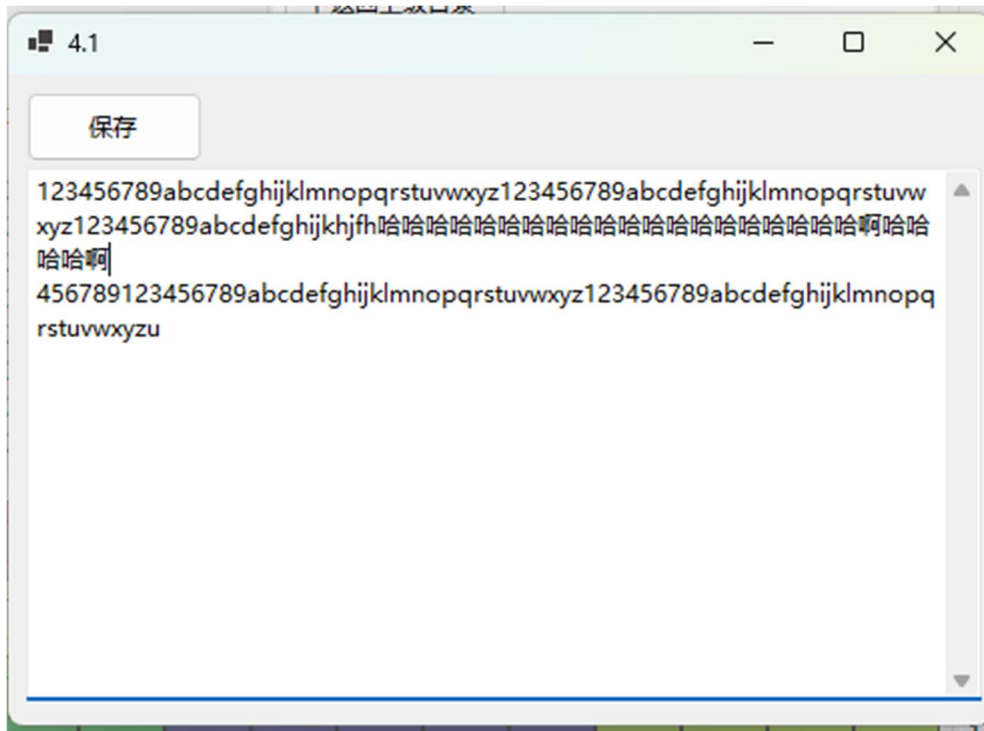
选中文件右键可选操作



选中文件夹右键可选操作



文件编辑窗口（文件只读时不可修改）



查看属性，可在属性页修改属性





## 2. FAT 监视器

通过 FAT 监视器可以监视 FAT 的记录值和分配状态。灰色部分表示 FAT 的对应盘块空闲，彩色表示已分配，且属于同一文件的盘块使用相同的颜色表示，不同文件使用不同颜色表示。

# 0	# 1	# 2	# 3	# 4	# 5	# 6	# 7	# 8	# 9	# 10	# 11	# 12	# 13	# 14	# 15
255	255	255	0	0	0	0	0	0	0	0	0	0	0	0	0
# 16	# 17	# 18	# 19	# 20	# 21	# 22	# 23	# 24	# 25	# 26	# 27	# 28	# 29	# 30	# 31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
# 32	# 33	# 34	# 35	# 36	# 37	# 38	# 39	# 40	# 41	# 42	# 43	# 44	# 45	# 46	# 47
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
# 48	# 49	# 50	# 51	# 52	# 53	# 54	# 55	# 56	# 57	# 58	# 59	# 60	# 61	# 62	# 63
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
# 64	# 65	# 66	# 67	# 68	# 69	# 70	# 71	# 72	# 73	# 74	# 75	# 76	# 77	# 78	# 79
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
# 80	# 81	# 82	# 83	# 84	# 85	# 86	# 87	# 88	# 89	# 90	# 91	# 92	# 93	# 94	# 95
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
# 96	# 97	# 98	# 99	# 100	# 101	# 102	# 103	# 104	# 105	# 106	# 107	# 108	# 109	# 110	# 111
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
# 112	# 113	# 114	# 115	# 116	# 117	# 118	# 119	# 120	# 121	# 122	# 123	# 124	# 125	# 126	# 127
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

根目录存储区域

FAT存储区域

FAT对应盘块编号

FAT记录值

# 0	# 1	# 2	# 3	# 4	# 5	# 6	# 7	# 8	# 9	# 10	# 11	# 12	# 13	# 14	# 15
255	255	255	255	255	255	7	8	255	17	255	255	255	91	94	16
# 16	# 17	# 18	# 19	# 20	# 21	# 22	# 23	# 24	# 25	# 26	# 27	# 28	# 29	# 30	# 31
9	255	23	24	28	25	29	255	255	26	27	255	255	30	255	255
# 32	# 33	# 34	# 35	# 36	# 37	# 38	# 39	# 40	# 41	# 42	# 43	# 44	# 45	# 46	# 47
33	34	255	36	37	38	255	40	41	42	43	97	45	46	47	48
# 48	# 49	# 50	# 51	# 52	# 53	# 54	# 55	# 56	# 57	# 58	# 59	# 60	# 61	# 62	# 63
49	255	51	52	53	54	80	56	57	58	59	60	61	75	255	64
# 64	# 65	# 66	# 67	# 68	# 69	# 70	# 71	# 72	# 73	# 74	# 75	# 76	# 77	# 78	# 79
65	66	67	68	69	255	71	72	73	74	255	76	77	78	79	255
# 80	# 81	# 82	# 83	# 84	# 85	# 86	# 87	# 88	# 89	# 90	# 91	# 92	# 93	# 94	# 95
81	82	83	84	255	86	87	88	255	255	255	92	93	255	95	96
# 96	# 97	# 98	# 99	# 100	# 101	# 102	# 103	# 104	# 105	# 106	# 107	# 108	# 109	# 110	# 111
255	98	99	100	101	106	103	104	105	255	107	255	0	0	0	0
# 112	# 113	# 114	# 115	# 116	# 117	# 118	# 119	# 120	# 121	# 122	# 123	# 124	# 125	# 126	# 127
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

鼠标在上面划过时，划过的块会显示为白色高亮，此时可点击该块查看盘块的具体内容  
(见下文-第3项)

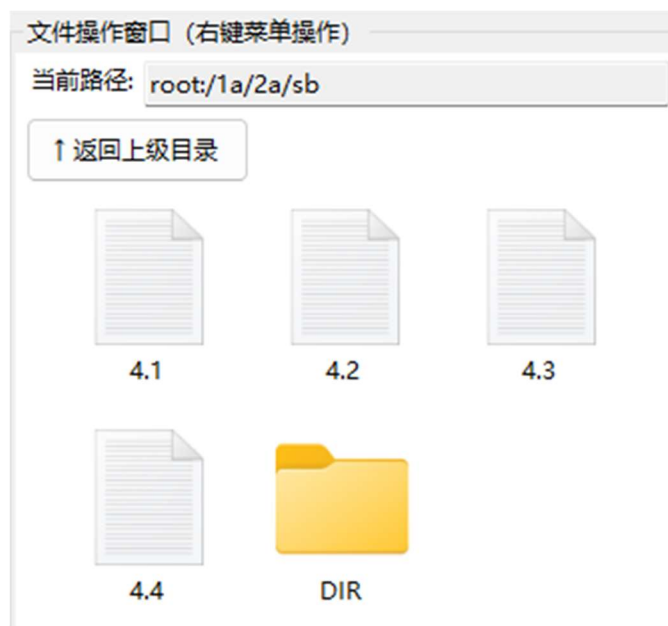
28	25	29	255	255	26
# 36 37	# 37 38	# 38 255	# 39 40	# 40 41	# 41 42
# 52 53	# 53 54	# 54 80	# 55 56	# 56 57	# 57 58
# 68 69	# 69 255	# 70 71	# 71 72	# 72 73	# 73 74
# 84	# 85	# 86	# 87	# 88	# 89

### 3. 盘块监视器

鼠标点击 FAT 监视模块上的单元格，就能查看单元格对应盘块的具体内容，分别以十六进制和 ASCII 格式显示在本模块上，每行 8 字节，刚好对应一行目录项，例如下图的 ASCII 部分可以看见一个目录的存储结构

盘块内容查看 (单击下方FAT示意图的任意块)															
十六进制								ascii							
2E	00	00	00	00	08	1F	00	.							
2E	2E	00	00	00	08	0A	00	..							
34	00	00	31	00	04	20	03	4	1						
34	00	00	32	00	04	23	04	4	2		#				
34	00	00	33	00	04	2C	06	4	3		,				
34	00	00	34	00	04	32	0A	4	4		2				
46	75	63	00	00	08	3E	01	Fuc			>				
24	00	00	00	00	00	00	00	\$							

上图盘块对应的目录：



#### 4. 监控管理选项区



磁盘占用进度条：

显示磁盘占用量，颜色会随着磁盘占用量而变化，由绿-黄-橙-红变化。

“换个颜色”按钮：

由于 FAT 监视器的单元格颜色是随机生成的，难免会有颜色难看或者看不清的问题，于是提供本按钮，用户可以自行刷新单元格的颜色。

“命令行控制”按钮：

用户可选择使用命令进行操作（详见下文-第 6 项）。

“开发者选项”按钮：

可使用一些附加功能（详见下文-第 7 项）。

“重置系统”按钮：

删除磁盘文件（disk.data），重启应用。

“关于”按钮：

显示一些关于本系统的信息。

## 5. 运行日志

记录用户操作，右键菜单可清空

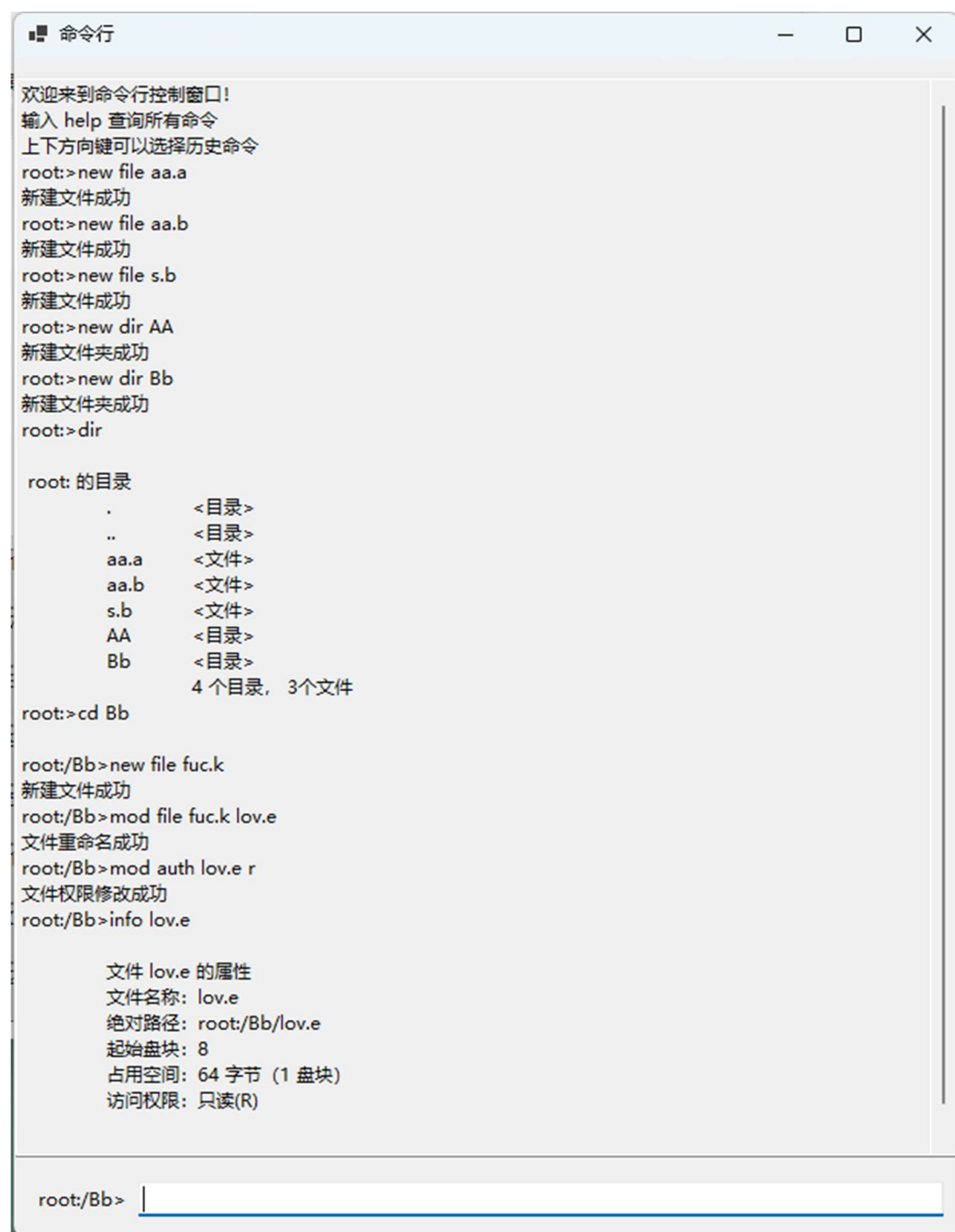


## 6. 命令行控制

点击“命令行控制”按钮可打开命令行控制窗口，输入“help”后可查看所有支持的命令及其格式。开发者选项要打开“开发者选项”后才能使用。



部分命令执行展示。



```
命令
欢迎来到命令行控制窗口!
输入 help 查询所有命令
上下方向键可以选择历史命令
root:>new file aa.a
新建文件成功
root:>new file aa.b
新建文件成功
root:>new file s.b
新建文件成功
root:>new dir AA
新建文件夹成功
root:>new dir Bb
新建文件夹成功
root:>dir

root: 的目录
.          <目录>
..         <目录>
aa.a       <文件>
aa.b       <文件>
s.b        <文件>
AA         <目录>
Bb         <目录>
           4 个目录, 3 个文件

root:>cd Bb

root/Bb>new file fuc.k
新建文件成功
root/Bb>mod file fuc.k lov.e
文件重命名成功
root/Bb>mod auth lov.e r
文件权限修改成功
root/Bb>info lov.e

文件 lov.e 的属性
文件名称: lov.e
绝对路径: root/Bb/lov.e
起始盘块: 8
占用空间: 64 字节 (1 盘块)
访问权限: 只读(R)

root/Bb> |
```

## 7. 开发者选项

可以打开一些拓展的功能

