目录

1 实验内容

1.1 实验背景

本次实验需要实现一个二级文件系统，在宿主机的一级操作系统中生成一个大文件（以下称为磁盘文件），通过读写磁盘文件的方式模拟文件系统读写硬盘的过程，实现在磁盘文件中读写小文件、在两级文件系统之间迁移文件等功能。

在开始进行本次实验前，需要阅读 UnixV6++ 文件系统部分的源码，在理解其设计的基础上设计本次实验所要实现的二级文件系统。

1.2 实验目标

本次实验的总体目标是实现一个客户端-服务器结构的的二级文件系统，支持多客户端同时访问服务器进行文件操作的功能，以下是需要实现的具体内容：

- 完成磁盘文件的格式化，将其划分为类 UnixV6++ 的磁盘文件格式
- 实现以下的文件操作，并提供客户端命令行交互界面：
  - 打开/关闭文件
  - 创建/删除文件或文件夹
  - 读取/写入文件内容
  - 移动文件指针
  - 列出当前目录下的文件与文件夹
  - 更改当前工作目录（可返回父目录）
  - 查看文件状态与磁盘状态
- 实现文件扇区权限管理，避免多线程读写冲突
- 通过在两级文件系统之间转移文件，测试文件系统读写正确性
- 撰写实验报告与用户手册

1.3 实现方式

- 实验/运行平台：Windows 10
- 编码语言：使用 $C\#$ 编码，目标框架：.Net 7.0
- 进程间通信方式：使用 TCP 协议在本地进行通讯

2 总体设计

本项目客户端仅负责向服务器发送控制台输入的指令内容，并输出服务器返回的文本信息，文件系统的核心功能均由服务器实现。文件服务器以自底向上的方式构建，共包含磁盘层、扇区层、文件层、用户层的四层结构。

2.1 磁盘结构

本项目实现的磁盘结构与 UnixV6++ 系统类似，分为超级块、Inode 区和数据区三部分，整个磁盘被划分为多个大小为 512 字节的扇区。

超级块占据 0-1 号扇区，其中的内容主要包括当前磁盘剩余空闲扇区和空闲 Inode 数量，以及用户信息，初始时仅包含一个用户的信息；Inode 区占据 2-1023 号扇区，每个扇区中存储 8 个外存 Inode，初始是仅包含根目录 Inode；数据区占据后续的 16384 个扇区，初始时完全为空。

2.2 磁盘层

磁盘层由 DiskManager 类实现，提供直接读写 disk.img（以下称为磁盘文件）的接口。系统初始化时，DiskManager 以内存映射的方式打开磁盘文件，若没有找到磁盘文件则创建该文件。该类提供的接口将直接对磁盘文件进行读写，不进行访问权限检查，不使用线程锁。

2.3 扇区层

扇区层主要由 FileSystem 类和 BlockManager 类实现，提供扇区级别的操作接口。

FileSystem 类提供超级块操作接口与用户信息操作接口，在系统初始化时检查超级块签名是否正确，不正确则初始化超级块与用户信息。超级块信息主要包括磁盘中的剩余扇区数与 Inode 数。

BlockManager 类提供扇区与 Inode 的操作接口。该类允许外部申请/释放扇区与 Inode 的权限，并允许外部对已申请权限的扇区或 Inode 进行读写操作，拒绝未授权区域的操

作，保证磁盘文件不被意外破坏。该类的接口使用了线程锁，保证多个客户端同时操作扇区/Inode 时不会发生冲突。

2.4 文件层

文件层主要由 FileManager 类实现，提供文件操作接口。FileManager 类提供了实验目标中说明的文件操作接口，在内部使用 OpenFile 结构表示已打开文件，每个 OpenFile 与 Inode 一一对应。FileManager 进行文件操作时，主要调用 BlockManager 提供的扇区/Inode 接口函数，并保证在申请到文件扇区/Inode 权限的情况下才进行操作。

2.5 用户层

用户层主要由 View 类实现，负责解析客户端发来的指令，并调用文件层或扇区层提供的接口进行文件操作，将操作结果或出错信息发送回客户端。

2.6 进程通信

进程间传输的网络消息以 SerializeMsg 类的派生类表示，并由 MsgParser 进行解析，完成对 TCP 协议数据包分包/黏包的处理。

解决方案启动时，首先开启一个服务器进程，该进程不断监听是否有客户端请求连接服务器。接下来开启一个客户端进程，该进程创建后会立即尝试连接服务器。

当服务器成功连接到一个客户端后，为其创建一个新线程，并在新线程中创建一个 FileManager 对象和一个 View 对象，然后创建一个用户信息副本，保证不同的客户端能够独立在文件层和用户层进行操作。服务器的扇区层和磁盘层对象是唯一的，以此保证磁盘内容的唯一性。

3 详细设计与实现

3.1 DiskManager 实现

DiskManager 在服务器存在唯一实例，该实例在服务器启动的第一时间被创建，并始终存在直到服务器被关闭。

创建 DiskManager 时将检查磁盘文件是否存在，若不存在则创建磁盘文件，并根据总扇区数设置其大小。

DiskManager 通过内存映射的方式操作磁盘文件，与在 Linux 中使用系统调用 mmap 类似。经过对比测试，使用内存映射的方式随机读写文件效率，比直接使用文件流读写文件快了 2 倍以上（顺序读写时文件流更快，但实际操作磁盘文件时以随机读写为主）。

C# 中提供了 MemoryMappedFile 这一结构，用于表示文件映射到的内存，并且该内存可以被多个进程共享，可作为另一种进程间通信的方式。考虑到此种方式还需要大量进程间共享锁来控制扇区的访问权限，因此最终没有采用这种进程间通信方式，但保留了内存映射操作文件的方式。

通过 MemoryMappedFile 对象可以生成 MemoryMappedViewAccessor 这一对象，其中包含对映射内存的读写操作 API，DiskManager 可直接使用这些 API 对映射内存进行读写，并在服务器关闭时将内存中的内容写回磁盘文件。

以下是 DiskManager 所实现的接口类定义：

```
internal interface IDiskManager
{
    // 读取字节
    public void ReadBytes(byte[] buffer, int offset, int count);
    // 写入字节
    public void WriteBytes(byte[] buffer, int offset);
    public void WriteBytes(byte[] buffer, int offset, int count);

    // 在指定位置读取结构体
    public void Read<T>(int position, out T value) where T : unmanaged;
    // 在指定位置写入结构体
    public void Write<T>(int position, ref T value) where T : unmanaged;

```

```
    /// <summary>
    /// 读取数组
    /// </summary>
    /// <param name="position">读取起始位置</param>
    /// <param name="array">数据存储数组</param>
    /// <param name="offset">数组序号偏移量</param>
    /// <param name="count">读取元素数量</param>
    public void ReadArray<T>(int position, T[] array, int offset, int count)
        where T : unmanaged;
    /// <summary>
    /// 写入数组
    /// </summary>
    /// <param name="position">读取起始位置</param>
    /// <param name="array">数据存储数组</param>
    /// <param name="offset">数组序号偏移量</param>
    /// <param name="count">写入元素数量</param>
    public void WriteArray<T>(int position, T[] array, int offset, int count)
        where T : unmanaged;
}
```

3.2 FileSystem 实现

FileSystem 同样在服务器存在唯一实例，该对象依赖 IDiskManager 运行，因此它在 DiskManager 被创建后创建，并始终存在直到服务器被关闭。

FileSystem 负责管理超级块与用户信息，因为用户信息是存储在超级块中的，为避免歧义，如没有特别说明，以下提到的的“超级块”均指超级块中非用户信息的部分。以下是超级块结构体（含用户信息）和用户结构体中，需要存储到磁盘的数据与常量的定义：

```
/// <summary>
/// 超级块结构体
/// </summary>
internal struct SuperBlock
{
    public const int SIZE = DiskManager.SUPER_BLOCK_SIZE * DiskManager.
        SECTOR_SIZE; // 超级块大小
    public const int MAX_USER_COUNT = SIZE / DiskUser.SIZE - 1; // 最大用户数
    public const int NO_USER_START = MAX_USER_COUNT * DiskUser.SIZE; // 非用
```

```
        户信息区起始位置
    public const int NO_USER_SIZE = SIZE - NO_USER_START; // 非用户信息区大小

    public const int SIGNATURE_SIZE = 12;   // 签名区大小
    public const int PADDING_SIZE = SIZE - MAX_USER_COUNT * DiskUser.SIZE -
        sizeof(int) * 7 - SIGNATURE_SIZE;   // 填充区大小

    public unsafe fixed byte users[MAX_USER_COUNT * DiskUser.SIZE]; // 用户信
        息表
    private int m_UserCount;    // 当前用户数（实际未实现多用户，因此此值总为
        1）

    private int m_Modified; // 被更改标识
    private int m_ModifyTime;   // 最近一次更新时间

    private int m_FreeSector;    // 数据区空闲盘块数
    private int m_DataSector;    // 数据区总盘快数

    private int m_FreeInode; // 空闲外存Inode数
    private int m_InodeCount;   // 外存Inode总数

    public unsafe fixed byte signature[SIGNATURE_SIZE]; // 签名区
    private unsafe fixed byte m_Padding[PADDING_SIZE];  // 填充区
}

/// <summary>
/// 磁盘中存储的用户信息类
/// </summary>
internal struct DiskUser
{

    public const int SUPER_USER_ID = 1; // 超级用户ID
    public const int DEFAULT_GROUP_ID = 1; // 默认分组ID
    public const int NAME_MAX_COUNT = DirectoryEntry.NAME_MAX_COUNT;   // 用
        户名最大长度
    public const int PASSWORD_MAX_COUNT = DirectoryEntry.NAME_MAX_COUNT;   //
         密码最大长度

    public const int SIZE = 4 * DirectoryEntry.NAME_MAX_COUNT + 4 * sizeof(
```

```
        int); // 用户信息大小

    public unsafe fixed byte name[NAME_MAX_COUNT];     // 用户名
    public unsafe fixed byte password[PASSWORD_MAX_COUNT]; // 密码

    public int uid;         // 用户标识号
    public int gid;         // 用户组标识号
    //  (此处的DirectoryEntry定义与UnixV6++相同)
    public DirectoryEntry home;     // 用户主目录
    public DirectoryEntry current;  // 用户当前目录
}
```

创建 FileSystem 时将检查超级块签名是否正确，若不正确则视为磁盘已损坏或未初始化，此时将格式化超级块内容（同时创建唯一的超级用户）并写入磁盘（未更正签名，因为后续 BlockManager 还要检查签名来判断是否需要格式化）。读取或格式化超级块后，FileSystem 将读取用户信息并保存于内存 User 结构中。

FileSystem 实现了如下的超级块管理接口，外部可从中获取/更改超级块信息，并将内存中的超级块信息同步到磁盘中。

```
internal interface ISuperBlockManager
{
    // 内存中的超级块副本，可直接进行读写
    public ref SuperBlock Sb { get; }
    // 同步超级块信息到磁盘
    public void UpdateSuperBlock();
}
```

FileSystem 还实现了如下的用户管理接口，外部可从中获取一个指定的用户信息，或更新用户信息。

```
internal interface IUserManager
{
    // 获取一个用户信息
    public User GetUser(int uid);
    // 设置一个用户信息
    public void SetUser(User user, int uid);
}
```

3.3 BlockManager 实现

BlockManager 同样在服务器存在唯一实例，该对象依赖 IDiskManager 和 ISuperBlockManager 运行，因此它在 DiskManager 和 FileSystem 被创建后创建，并始终存在直到服务器被关闭。

BlockManager 负责管理扇区和 Inode 权限的申请与释放，以及提供读写扇区数据的接口。扇区 Sector 的定义非常简单，仅记录了该扇区的编号。为保持 Inode 大小不变，磁盘中 Inode 数据（DiskInode 结构体）的定义与 UnixV6++ 是相同的，内存 Inode 则与磁盘 Inode 类似，仅多记录了 Inode 编号方便使用。

BlockManager 创建时，将检查超级块签名是否正确，若不正确则格式化硬盘 Inode 区和数据区，并将签名更正，但未创建根目录。之后 BlockManager 将遍历所有已经使用的 Inode 并记录其使用情况，然后根据已使用 Inode 的地址记录，查找所有已使用扇区并记录。之后需要获取空闲 Inode/扇区时，BlockManager 直接根据当前记录的 Inode/扇区使用情况查找空闲内容并返回。

BlockManager 实现了如下的扇区操作接口，外部可通过此接口申请/释放扇区权限，并对扇区数进行读写操作。读写时必须提供已经申请到权限的 Sector 结构，否则无法读写。为保证多个线程访问 BlockManager 时不会冲突，以下接口的实现均使用了线程锁，控制同时仅有一个线程访问扇区权限操作。需要特别说明的是，扇区权限控制已经避免了多个线程同时读写同一扇区，因此读写扇区数据并非临界区，真正的临界区仅为对扇区权限状态和磁盘扇区使用状态进行访问操作的部分。

```
internal interface ISectorManager
{
    // 获取一个空闲盘块的序号
    public int GetEmptySector();
    // 获取一个指定盘块的控制权
    public Sector GetSector(int sectorNo);
    // 归还一个指定盘块的控制权
    public void PutSector(Sector sector);

    // 读取一个盘块中的内容
    public void ReadBytes(Sector sector, byte[] buffer, int size =
```

```
        DiskManager.SECTOR_SIZE, int position = 0);
    // 向一个盘块中写入内容
    public void WriteBytes(Sector sector, byte[] buffer, int size =
        DiskManager.SECTOR_SIZE, int position = 0);

    // 在指定位置读取结构体 其大小不得超过盘块大小
    public void ReadStruct<T>(Sector sector, out T value, int position = 0)
        where T : unmanaged;
    // 在指定位置写入结构体 其大小不得超过盘块大小
    public void WriteStruct<T>(Sector sector, ref T value, int position = 0)
        where T : unmanaged;

    /// <summary>
    /// 读取数组 总大小不得超过盘块大小
    /// </summary>
    /// <param name="sector">读取起始位置</param>
    /// <param name="array">数据存储数组</param>
    /// <param name="offset">数组序号偏移量</param>
    /// <param name="count">读取元素数量</param>
    /// <param name="position"></param>
    public void ReadArray<T>(Sector sector, T[] array, int offset, int count,
         int position = 0) where T : unmanaged;

    /// <summary>
    /// 写入数组 总大小不得超过盘块大小
    /// </summary>
    /// <param name="sector">读取起始位置</param>
    /// <param name="array">数据存储数组</param>
    /// <param name="offset">数组序号偏移量</param>
    /// <param name="count">写入元素数量</param>
    /// <param name="position"></param>
    public void WriteArray<T>(Sector sector, T[] array, int offset, int count
        , int position = 0) where T : unmanaged;

    // 清除扇区中的数据 使其变为空闲盘块
    public void ClearSector(params Sector[] sectors);
}
```

BlockManager 还实现了如下的 Inode 操作接口，外部可通过此接口申请/释放 Inode 权

限，并能够更新已申请到权限的 Inode 数据。此接口的实现同样使用了（另一把）线程锁来保护临界区，同理，对 Inode 的读写操作也不在临界区范围内。

```
internal interface IInodeManager
{
    // 通知FileManager是否需要创建根目录
    public bool Formatting { get; set; }
    // 获取一个空闲Inode的序号
    public Inode GetEmptyInode();
    // 获取一个Inode的控制权
    public Inode GetInode(int inodeNo);
    // 释放一个Inode
    public void PutInode(int inodeNo);
    // 更新一个Inode的数据到磁盘中
    public void UpdateInode(int inodeNo, Inode inode);
}
```

3.4 FileManager 实现

FileManager 在每个面向客户端的线程中都有一个实例，该对象依赖 ISectorManager 和 IInodeManager 运行，并且需要一个用户的内存 User 对象的副本，从中读写用户当前工作目录。FileManager 在服务器与客户端成功连接后创建，在此之前需要先创建 User 副本。

内存 User 对象与 FileSystem 实现小节中的 DiskUser 类似，仅新增了一个字典，用于记录使用打开文件命令打开的文件。注意，FileManager（或 View）打开的文件并不会被记录到 User 中，而是由 FileManager（或 View）自行管理。

FileManager 负责提供文件操作，与一级文件系统内置的 File 类和 Directory 类表现类似，实现了如下的 IFileManager 文件操作接口：

```
internal interface IFileManager
{
    // 当前用户
    public User CurrentUser { get; set; }

    // 打开文件
    public OpenFile Open(string path, bool access = true);
```

```
    // 关闭文件
    public void Close(OpenFile file);

    // 创建文件
    public void CreateFile(string path);
    // 删除文件
    public void DeleteFile(string path);

    // 创建目录
    public void CreateDirectory(string path);
    // 删除目录
    public void DeleteDirectory(string path, bool deleteSub);

    // 从文件读取二进制数据
    public void ReadBytes(OpenFile file, byte[] data, int size = -1);
    // 写入二进制数据到文件
    public void WriteBytes(OpenFile file, byte[] data);
    // 从文件读取结构体
    public void ReadStruct<T>(OpenFile file, out T value) where T : unmanaged
        ;
    // 写入结构体到文件
    public void WriteStruct<T>(OpenFile file, ref T value)where T : unmanaged
        ;

    // 移动文件指针
    public void Seek(OpenFile file, int pos, SeekType type = SeekType.Begin);

    // 获取当前用户目录下的文件和目录
    //  (此处的Entry是外存目录项的内存表示，数据结构基本相同)
    public IEnumerable<Entry> GetEntries();

    // 判断一个路径是否为一个已存在的文件
    public bool FileExists(string path);
    // 判断一个路径是否为一个已存在的目录
    public bool DirectoryExists(string path);

    // 根据路径查找文件Inode序号
    public int GetFileInode(string path);

```

```
45      // 切换当前工作目录
46      public void ChangeDirectory(string path);
47      // 获取当前用户的工作目录
48      public string GetCurrentPath();
49  }
50
51  // 文件指针移动类型
52  internal enum SeekType
53  {
54      Begin,
55      Current,
56      End
57  }
```

FileManager 在访问文件时，需要先从 IInodeManager 中获取文件的 Inode，并根据读写需要从 ISectorManager 中申请扇区的使用权，然后才能使用已申请到权限的 Inode 和 Sector 对文件或目录进行读写操作。这一机制的实现难点是考虑如何保证申请权限的 Inode/Sector 在使用完毕后被正确释放，为此，我为 Inode、Sector 和记录 Inode 的 OpenFile 类均实现了 IDisposable 接口，并尽量使用 using 语句创建它们的对象，保证在访问出错的情况下也能正确释放权限。

OpenFile 的结构非常简单，除了记录该文件对应的 Inode 外，仅记录了该文件的文件指针位置和文件访问权限被释放情况。

FileManager 的 Open 函数除了打开文件外，还会更新文件的访问时间。写入文件的接口则会同时更新文件的访问与修改时间。

FileManager 在创建目录时，实现了默认的隐藏目录创建，使用户能够通过访问 . 和 .. 的方式访问当前目录/父目录。特别地，根目录的父目录被设置为其自身。

3.5 View 实现

View 同样在每个面向客户端的线程中都有一个实例，该对象依赖 ISuperBlockManager、IUserManager、IFileManager 以及先前创建的 User 副本运行，因此将在 FileManager 创建之后创建，并在客户端断开连接后销毁。

View 主要负责将客户端发来的字符串消息解析为命令和参数，并根据命令调用相应的

IFileManager 接口函数实现功能，并将结果返回给客户端。

另外，View 还需要使用 IUserManager 对用户信息进行读写（主要是当前工作目录），以及通过 ISuperBlockManager 获取磁盘信息。

3.6 进程间通信实现

进程间通信使用了 TCP 协议的套接字实现。实现通信的难点在于，尽管 TCP 协议保证了数据传输的稳定性（丢包后会重发）与有序性（数据包按序到达），但不能保证每次发送的消息都在一个独立的数据包中，即会发生分包和黏包的情况。

为了保证通信时收到的消息与发出的消息相同，需要将被分包的数据合并，以及将黏包拆分为多个消息。为了正确解析消息，需要在消息原始内容前添加头信息，表示本消息的总长度。当数据包长度大于消息长度时，解析器认为发生了黏包，并将消息从数据包中拆分出来；反之，解析器认为发生了分包，将当前数据包内容存储在缓存中，等待后续数据包，直到缓存长度达到消息长度时再合并数据包。

在本项目中，数据包解析器 MsgParser 被包含在 View 中，用于解析客户端发来的消息（客户端直接创建一个全局的 MsgParser 解析消息）。消息以 SerializeMsg 类表示，并可以从中派生出多种消息，每种消息具有一个唯一的消息类型码，同样作为头信息添加到了消息内容前。SerializeMsg 及其派生类能够方便地转换为带头信息的字节数组，或从字节数组转为需要使用的数据。以下是 SerializeMsg 类的定义：

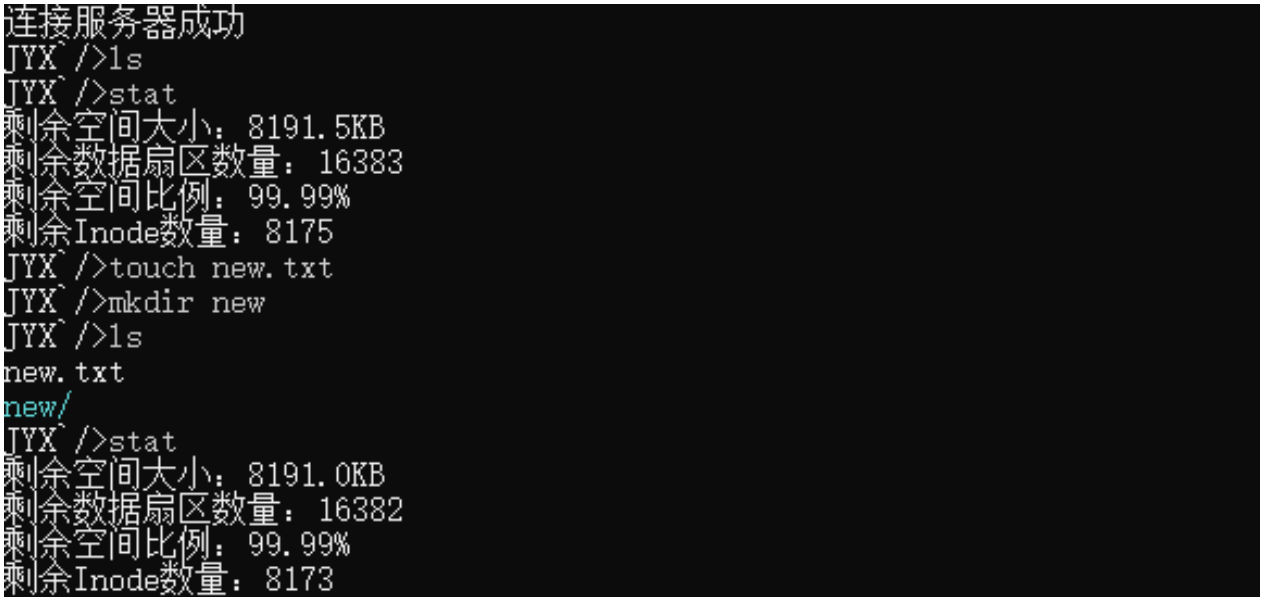
```
// SeializeData类中实现了将常用的数据类型与字节数组相互转换的方法，子类能够方便地使用
public abstract class SerializeMsg : SerializeData
{
    // 子类需要定义一个唯一的消息ID
    public abstract int MsgID { get; }
    // 子类实际要实现的获取消息长度的方法
    public abstract int GetByteCount_Msg();
    // 子类实际要实现的序列化方法
    protected abstract void ToBytesDetail_Msg();
    // 子类实际要实现的反序列化方法
    protected abstract void ReadBytesDetail_Msg();
}
```

4 测试结果

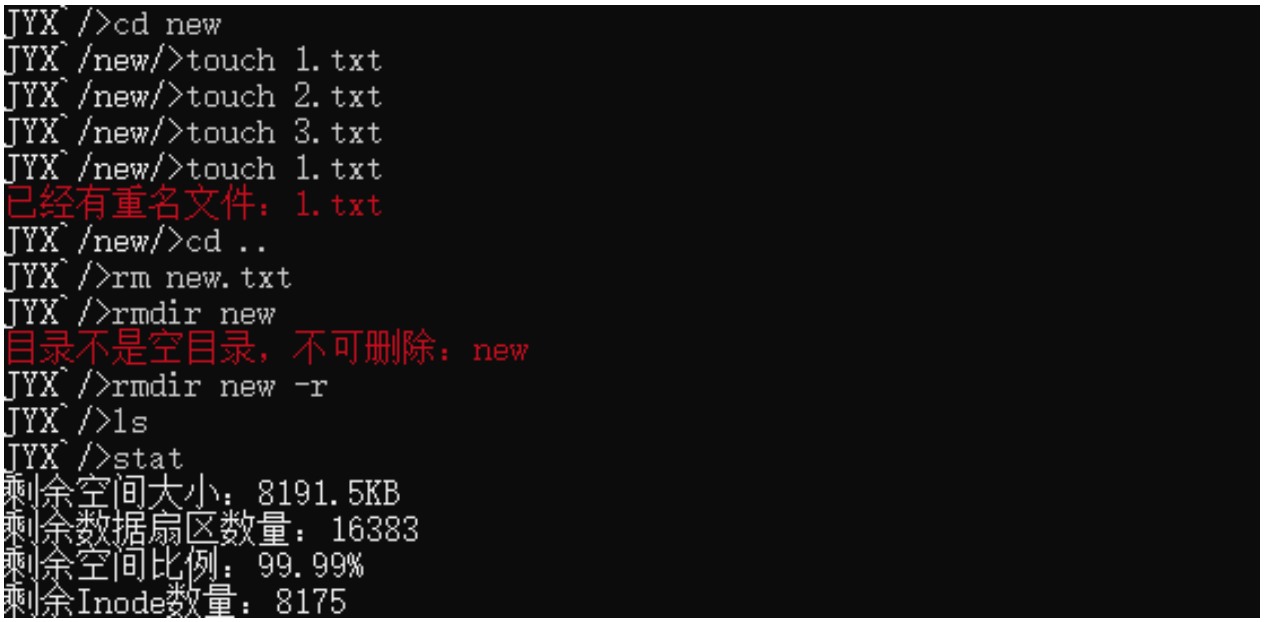
本项目的测试主要包括文件读写测试与共享文件测试，即需要测试文件系统的基本功能，以及多客户端访问时的系统稳定性。

4.1 文件增删测试

首先需要进行文件与目录的增删测试，毕竟如果不能创建文件，也不可能进行读写测试。在文件夹中使用 touch 和 mkdir 命令分别创建一个文件和目录后，可通过 ls 命令查看现有的的文件/目录（蓝色字体的是目录），并可以通过 stat 命令查看当前磁盘剩余资源状态，其中 Inode 减少了 2 个，扇区减少了 1 个（因为新创建的空文件不占用扇区，空目录有两个默认目录项，需要至少 1 个扇区）。



进入新创建的文件夹并创建一些文件，当尝试创建的文件已存在时会收到错误提示（创建目录重名时同理）。返回上级目录并尝试使用 rmdir 命令删除目录，由于 rmdir 命令默认仅删除空目录，因此此时收到错误提示。添加-r 参数后，可以删除目录及其中的文件/子目录。删除文件和目录后，先前被占用的 Inode/扇区资源已被释放。



4.2 文件读写测试

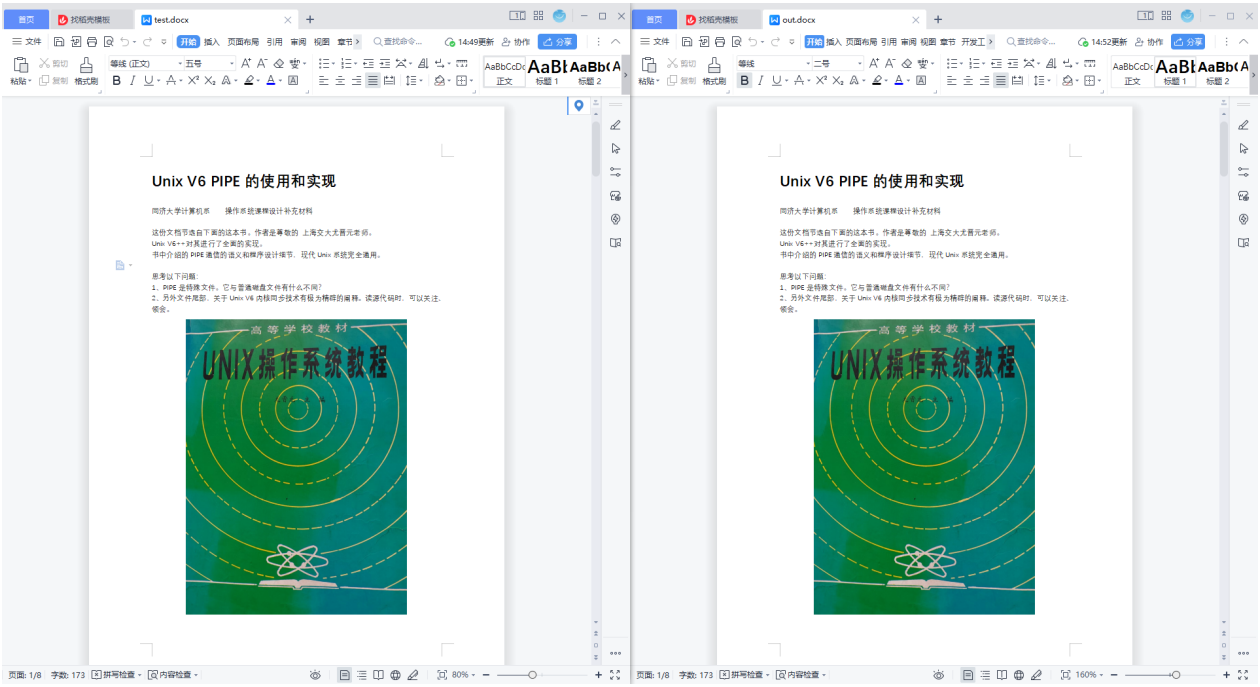
尽管文件系统实现了文件打开/关闭等功能，但这些功能不适合在命令行中使用，因此并没有提供相关的命令。对文件读写的正确性测试主要通过讲外部文件读入二级文件系统，再写回外部并比较其内容完成。另外，本系统也可以查看 UTF-8 编码的纯文本文件内容（尝试查看其它内容会乱码）。

使用 input 命令将一个纯文本文件读入本系统，并使用 cat 命令，可查看到其中的内容。为防止 TCP 消息解析缓冲区溢出，cat 命令限制了可查看文件的长度（当然这是可以改进的，只是因为本系统中没有完整的文本编辑器，个人认为查看大文本文件也没有太大意义）。

```
JYX`/test/>input D:\C#\Homework\OperatingSystemHW\README.md md
JYX`/test/>cat md
# OperatingSystemHW
## 项目介绍
本项目为同济大学操作系统课程设计作品，使用C#实现了一个二级文件系统，即在一个大文件中模仿文件系统的方式存储小文件。

## 使用方式
本项目的解决方案中包含两个项目，分别为OperatingSystemHW（服务器）与Client（客户端）。可以同时开启多个客户端程序，开启的客户端会自动连接到服务器。服务器使用本机端口8109，客户端端口由系统自动分配。

克隆仓库到本地后，启动解决方案，会分别生成一个服务器实例和一个客户端实例，并自动在生成的服务器可执行程序目录中生成名为disk.img的磁盘文件。若要开启多个客户端，需要找到生成的客户端可执行程序并手动执行。

## 客户端命令
客户端命令模仿Linux命令编写。文件路径默认为从当前工作目录起始的相对路径，使用绝对路径需要在原路径前增加斜线“/”，例如：
```
cd /   // 将当前工作目录更改为根目录
touch a.txt   // 在当前工作目录中创建名为a.txt的文件
touch /a.txt   // 在根目录中创建名为a.txt的文件
```
- ls：显示当前目录下的文件和文件夹列表。
- clear：清除控制台的显示内容。
- cat：显示指定文件的内容。参数为文件路径。
- stat：显示指定文件的状态信息，如文件大小、修改时间等。参数为文件路径，若不提供参数，则此命令将显示磁盘状态信息，如剩余空间大小等。
- touch：创建一个新的空文件。参数为文件路径。
```

本系统沿用了 UnixV6++ 对文件占用扇区的索引方式，即三级索引，需要使用二级索引盘块的文件可称为巨大文件（实际大小约为 131KB-16MB）。使用 input 读入一个巨大文件后，再使用 output 命令将其写回外部。查看此文件内容，可以发现该 Word 文档没有被损坏，内容也与移入本系统前一致，说明本系统能够正确地对巨大文件索引与读写。

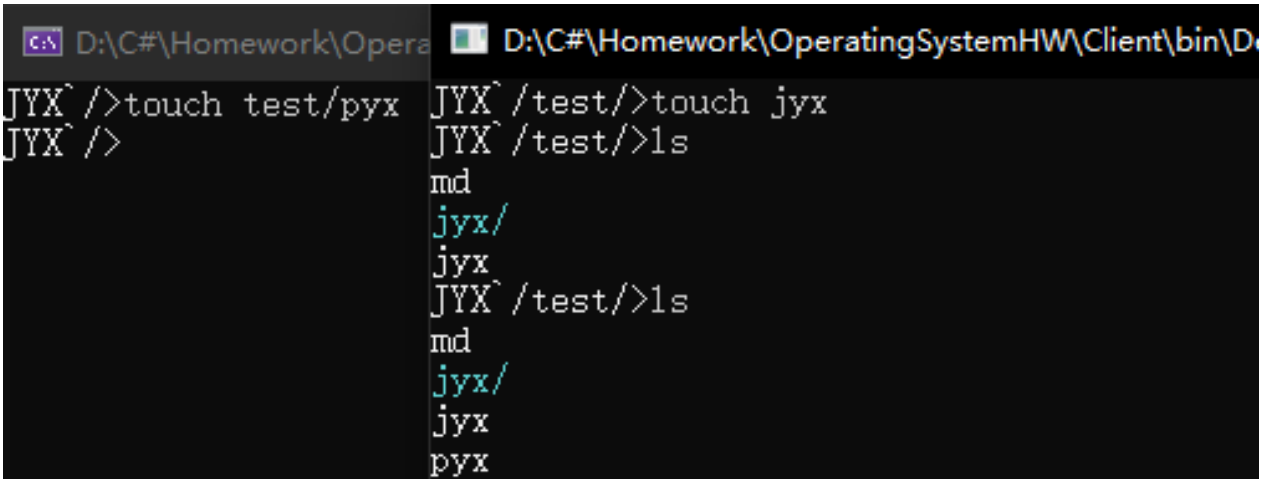
```
JYX`/>input test.docx test
JYX`/>stat
剩余空间大小：1585.5KB
剩余数据扇区数量：3171
剩余空间比例：19.35%
剩余Inode数量：8172
JYX`/>stat test
文件名：test
文件大小：6546.5KB 文件Inode编号：5
最后访问时间：2023-05-26 21:27:54
最后修改时间：2023-05-26 21:27:54
JYX`/>output test out.docx
```



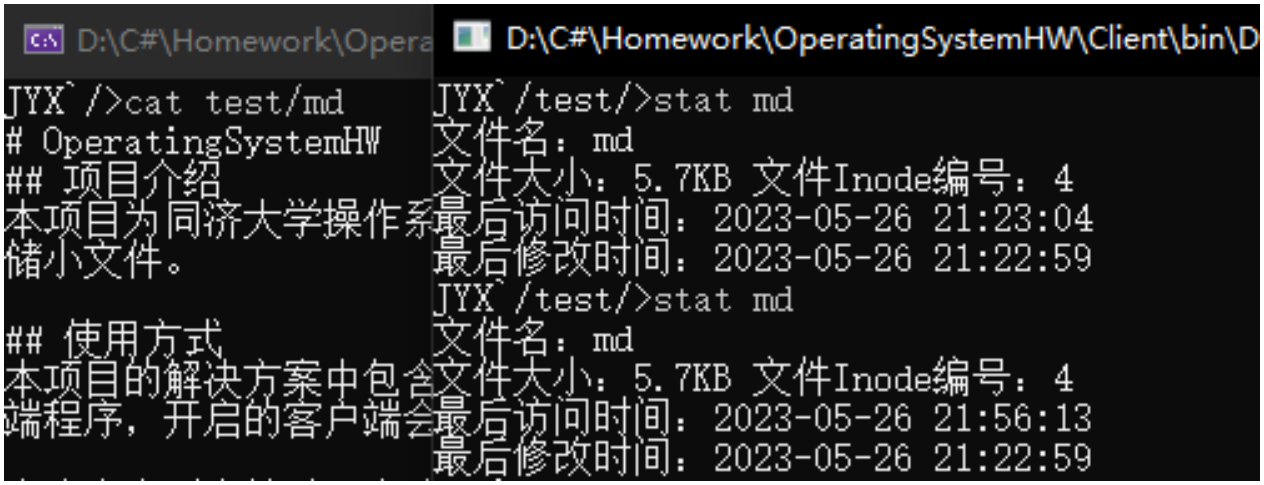
4.3 共享文件测试

本系统实现了多客户端同时访问文件系统，并通过扇区/Inode 权限控制，限制了同一时间仅能有一个客户端访问同一扇区/Inode。

以下测试结果的左右两侧为不同的客户端控制台信息，一个客户端对文件进行操作后，另一客户端能够实时读取文件系统的更新，且两个客户端能够工作在不同目录下。



创建文件测试，文件目录实时更新



访问文件测试，访问时间实时更新

5 总结

5.1 遭遇的困难

在实现本项目的过程中，因为基本舍弃了先前参考的 UnixV6++ 源码结构（主要原因是对 C++ 不熟悉，从源码中裁剪出文件系统也很困难），因此首先面临的困难便是重新设计文件系统结构。

在完成文件系统的主要功能前，考虑到可能面临较大的实现困难，我仅设计了一个单线程的文件系统（没有客户端和服务器的概念），此时的文件层和用户层均为单例对象。在完成了单进程测试后，我才尝试添加了网络通信模块，并实现多线程服务器，这需要对原本的系统结构进行调整。得益于先前对系统层级划分得较为清晰，我在调整系统结构时没有遇到太大的困难，但设计系统结构确实耗费了不少时间精力。

在进行具体的编码时，我发现尽管使用了依赖注入的方式构造对象，但进行单元测试仍然比较困难，因为无论如何文件系统都需要依赖磁盘文件的内容，而读取/写入磁盘文件的操作在进行单元测试前本身就是不可靠的，我只能通过查看二进制文件的方式测试底层接口，再逐级向上构造上层抽象。所幸磁盘层的代码比较简单，我几乎没有遇到什么错误。

然而，由于使用底层接口构造一个合适的测试用例，同样也有很大的工作量，因此上层接口的测试也不太顺利。尤其是通过 Inode 索引文件盘块的函数，需要依赖分散的索引盘块内容，逻辑也是最为复杂的，我只能使用不同大小的文件进行读入读出对比测试，在发生错误时调试修改，目前仍然没有十足的把握认为这个函数能够应对所有情况。

另一个较为困难的部分是网络通信的实现。由于不太了解 TCP 协议的性质，我开始时对数据包内容和发送内容不符的现象百思不得其解（所幸没有使用更不可靠的 UDP 协议）。查阅资料后，我使用添加头信息的方式解决了分包和黏包的处理，但实现消息的解析还是耗费了不少时间。

5.2 改进的方向

本系统还有不少可改进的内容，部分内容我已经作出了一定尝试，但限于时间原因没有完全实现。以下是我认为比较有意义也具有可行性的改进方向，其中也说明了我认为改

进过程中可能存在的困难：

- 文件并发访问：我已经花费了不少精力设计扇区级的访问权限控制，理论上可以支持并发访问文件的不同扇区（未验证并发访问磁盘层是否会出错）。然而目前的读写操作会先申请文件的 Inode 权限，再申请扇区权限，然后对扇区进行读写，整个过程中 Inode 权限没有被释放，因此其它线程也无法访问这个文件。如果需要即时释放 Inode，可能会导致现有代码中依赖原先申请的权限的部分无法运行，这需要大量的测试来保证对权限作用范围的“缩减”是正确的。不过总而言之，实现文件并发访问是目前性价比最高的改进方向。

- 访问权限区分：目前的文件访问权限是没有区分读者和写者的，打开文件的方式也没有进行区分，无法实现多个进程同时读取相同文件。若要实现访问权限类型的区分，需要更改权限分配与检查的机制。这并不是最大的难点，难点在于文件层几乎所有接口实现都需要更改调用扇区层接口的方式，因为访问权限的划分必然导致至少申请访问权限的接口定义发生变化。而如此大量的改动，很可能出现权限申请类型错误的问题，或者不得不拆分原本完全可复用的函数，因为其中需要申请的访问权限类型可能在不同的调用时是不同的。

- 用户信息管理：在现在的系统中，用户信息被存放于前两个扇区中，至多可存储 7 个用户的信息。然而，本系统实际上并没有增删用户与修改用户名/密码的方法，甚至没有登录界面。不同的用户可以对文件具有不同的访问权限，这在文件系统中是有意义的，但实现完整的用户管理需要实现较多的新用户管理方法和控制台命令，虽然原理简单，但开发工程量也不小。

5.3 收获的能力

在本次课程设计的实现过程中，我首先学习了 UnixV6++ 的源码，对一个文件系统的运行方式有了更深入的理解。在此基础上，我完成了对一个新的文件系统的设计，尽管在设计时没有优先考虑系统的性能，但我也学习到了如何设计更为清晰的系统结构，这也在我后期调整系统结构时发挥了重要的积极作用。

对二级文件系统的测试是十分困难的，但在完成内存映射和文件流两种方式读写文件效率的对比测试时，我仍然体会到了面向接口编程对系统灵活性的提升，这使得我可以轻松地使用两个不同的继承 IDiskManager 的类来对文件读写进行测试。

$C\#$ 是一门包含垃圾回收机制（GC）的语言，在其中使用的大多数对象都被称为托管类型对象，由 GC 负责控制它们的生命周期。但是为了使用内存映射操作文件，我需要严格控制对象读写的内存地址，这需要使用 $C\#$ 中的非托管对象来表示待读写数据，以及使用不安全代码块来操作指针。在编写 $C\#$ 代码使用系统调用，或与 C++ 程序进行交互时，使用非托管结构体和不安全代码是十分常见的，我也在这次课程设计中收获了不少这方面的经验。

最后，在本次课程设计中我还掌握了一些网络编程的基础技能，对常用的 TCP 套接字通信有了一定的应用经验。这也让我认识到计算机网络知识的重要性，因为在不了解原理的情况下，我对网络传输中发生的问题是束手无策的。