

同济大学计算机系

操作系统课程设计报告



学 号 2152809

姓 名 曾崇然

专 业 计算机科学与技术

授课老师 邓蓉老师

目录

一. 任务描述.....	4
1.1 目的.....	4
1.2 内容.....	4
1.3 要求,	4
二. 概要设计.....	4
三. 详细设计.....	5
3.1 磁盘驱动模块.....	5
3.1.1 DiskManager 类设计:	5
3.2 高速缓存管理模块.....	6
3.2.1 Buf 类设计.....	6
3.2.2 BufferManager 类设计	7
3.3 文件系统模块.....	8
3.3.1 SuperBlock 类设计	8
3.3.2 FileSystem 类设计	9
3.4 打开文件管理模块.....	10
3.4.1 File 类设计	10
3.4.2 OpenFiles 类设计	10
3.4.3 IOParameter 类设计	11
3.4.4 OpenFileTable 类设计	11
3.4.5 Inode 类设计.....	12
3.4.6 DiskInode 类设计	13
3.4.6 InodeTable 类设计	14
3.5 文件管理模块.....	15
3.5.1 DirectoryEntry 类设计.....	15
3.5.2 FileManager 类设计	16
3.6 核心模块.....	17
3.6.1 Kernel 类设计	17
3.6.2 State 类设计.....	18
3.6.3 Space 类设计	19
3.7 main 函数.....	19
3.7.1 获取各个模块.....	19
3.7.2 指令处理函数.....	20
3.7.3 系统调用函数.....	20
3.7.4 主模块.....	21
四. 代码测试.....	21
4.1 实验环境.....	21
4.2 使用说明.....	21
4.3 指令说明和测试.....	22
4.3.1 format 指令	22
4.3.2 ls 指令	22
4.3.3 cd 指令.....	22
4.3.4 mkdir 指令	23

4.3.5 create 指令	23
4.3.6 rm 指令	23
4.3.7 open 指令	23
4.3.8 close 指令	24
4.3.9 read 指令	24
4.3.10 write 指令	24
4.3.11 lseek 指令	24
4.3.12 link 指令	24
4.3.13 unlink 指令	24
4.3.14 fin 指令	25
4.3.15 cat	25
4.3.16 fout	26
4.3.17 exit 指令	27
4.3.18 shutdown 指令	28

一. 任务描述

1.1 目的

阅读、裁剪操作系统源代码（文件相关部分），深入理解操作系统文件概念和文件系统实现细节，培养剖析大型软件、设计系统程序的能力。

1.2 内容

1. 剖析 Unix V6++源代码，深入理解其文件管理模块、高速缓存管理模块和硬盘驱动模块的设计思路和实现技术。
2. 裁剪 Unix V6++内核，用以管理二级文件系统。

1.3 要求,

1. 设计满足以下指标的简单二级文件系统 SecondaryFS，宿主操作系统可以是 windows 也可以是 Linux。
2. 编写 2 个应用程序：
 - (1) Initialize 程序，格式化二级文件系统 c:\SecondaryFS.img。

超级块	inode 区	数据区
-----	---------	-----

- (2) SecondaryFS 程序，接收客户端输入的文件操作命令，访问二级文件系统。
3. 为二级文件系统 SecondaryFS 设计一个简单的测试用用户接口：
 - (1) 提供文件系统访问命令：每个命令行对应一个系统调用。
 - (2) 命令 cd，改变文件系统会话的当前工作目录。文件系统会话指用户使用文件系统的整个过程。
 - (3) 命令 fin [extername] [intername]，将外部名为 extername 的文件存入二级文件系统，内部文件名为 intername。外部指 windows 或 Linux 系统。
 - (4) 命令 fout [intername] [extername]，将内部文件名为 intername 的文件写入外部名为 extername 的文件。
 - (5) 命令 shutdown，安全关闭二级文件系统 SecondaryFS。将脏缓存写回镜像文件。
 - (6) 命令 exit，关闭程序 SecondaryFS。相当于断电，文件系统会丢数据。

二. 概要设计

本次课程设计的目标是实现一个单用户进程的二级文件管理系统。本系统中，使用一个二进制的大文件模拟磁盘，每 512 个字节分为一个数据段，用以模拟磁盘的各个扇区。再 UnixV6++中，很多用以管理以及文件系统的数据结构均可以用来管理二级文件管理系统，

区别主要再与磁盘驱动接口，在一级文件管理系统中，是通过向硬盘发送 DMA 命令，来在物理磁盘上读写数据，而在二级文件系统中，是通过宿主机操作系统的 read 和 write 系统调用来对虚拟磁盘镜像进行读写操作。本次设计使用 Windows 的 fread, fwrite, fseek 系统调用来进行磁盘相关的操作，基于此结合 UnixV6++的文件管理和高速缓存系统实现二级文件管理系统。主要分为以下几个模块：

- 磁盘驱动模块：负责创建、打开、关闭、读写和对磁盘进行读写的相关操作
- 高速缓存管理模块：管理系统中的缓存块，包括分配缓存，将磁盘读入缓存，将缓存写入磁盘等相关操作
- 文件系统模块：管理文件系统相关的资源，如 SuperBlock 的管理，Inode 和磁盘块的分配和回收等
- 打开文件管理模块：负责对打开文件机构的管理，建立用户与打开文件内核数据的勾连关系
- 文件管理模块：负责提供对文件系统进行操作的接口，如格式化系统，打开文件，关闭文件，定位读写位置，获取文件信息，目录搜索，写入目录项，设置当前工作路径，创建异名引用，取消引用，删除文件或者文件夹等等
- 核心模块：用于对各个模块进行统一的管理，并且保存当前的系统的状态，提供一个内存空间用于进行读写操作
- main 函数：主要用于提供用户操作界面，解析用户的命令，调用系统中相应的系统调用进行操作

三. 详细设计

3.1 磁盘驱动模块

3.1.1 DiskManager 类设计：

```
class DiskManager {
private:
    const char* diskPath = "SecondaryFS.img"; //磁盘镜像路径
    FILE* diskFd;

public:
    DiskManager();
    ~DiskManager();

    void createDisk(); //创建磁盘
    void openDisk(); //打开硬盘
    void seekOneBlock(int blkno); //选择一个块的一个位置
    void readOneBlock(unsigned char* buffer); //读出一个块
    void writeOneBlock(unsigned char* buffer); //写入一个块
    void closeDisk(); //关闭硬盘
};
```

DiskManager 负责对用于模拟磁盘的镜像文件进行操作。

其属性包含磁盘的路径和磁盘文件的 FILE 结构指针。

方法包括创建一个空的 img 文件 (createDisk()), 其大小为 0, 打开硬盘 (openDisk()), 选择一个块的位置 (seekOneBlock(int blkno)), 读入一个块到缓存 (readOneBlock()), 写缓存到一个块 (writeOneBlock()), 关闭磁盘 (closeDisk())。

该类的设计是基于二级文件管理系统的概念, 使用了 Windows 的系统调用来进行文件操作, 来模拟磁盘的读写操作。

3.2 高速缓存管理模块

3.2.1 Buf 类设计

```
class Buf {  
public:  
    bool is_dirty; //是否为脏缓冲  
    bool is_new; //是否为新分配  
  
    unsigned char* b_addr; /* 指向该缓冲区管理的缓冲区的首地址 */  
    int b_blkno; /* 磁盘逻辑块号 */  
};
```

buf 类定义了相应的缓存块的使用情况。

属性包含该缓存是否为脏, 是否为新分配的, 其管理的缓存块的首地址以及其对应的物理盘块号。

由于本系统初步设计时只考虑一个设备和一个用户, 同时任何时候在系统中只有一个进程, 因此对原 Unix 的 buf 类进行了裁剪。去掉了一些进程管理相关的状态和属性; 由于只有一个进程, 所以可以将所有的缓存都视作自由缓存, 在缓存用完时直接从中重新分配; 由于只有一个设备, 所以去掉一些设备号相关的属性, 最后简化了一些错误信息相关的属性, 简化结果如上图。

3.2.2 BufferManager 类设计

```
class BufferManager {
public:
    /* static const member */
    static const int NBUF = 15;          /* 缓存控制块、缓冲区的数量 */
    static const int BUFFER_SIZE = 512; /* 缓冲区大小。以字节为单位 */

public:
    BufferManager();
    ~BufferManager();

    Buf* GetBlk(int blkno); //申请一块缓存，用于读写字符块blkno

    Buf* Bread(int blkno); //读一个磁盘块

    void Bwrite(Buf* bp); //写一个磁盘块

    void Bflush(); //将缓存全部输出到磁盘

    void ResetBufferManager(); //重置

private:
    int usedBufNum; // 被使用的缓存块的数量
    Buf m_Buf[NBUF]; //缓存管理块
    unsigned char Buffer[NBUF][BUFFER_SIZE] = {}; //缓存区
    queue<int> bufQueue; //被分配给块的缓存队列
};
```

BufferManager 定义了系统中的缓存块的管理和相关操作，用于减少文件管理系统的 IO 次数。

其属性包含缓存块的数量和大小，已经被使用的缓存块的数量，用于管理缓存的缓存管理块数组，缓存区，被使用的缓存的缓存队列。

方法包括分配缓存，将磁盘块读入缓存，将缓存写入磁盘，将缓存全部更新到磁盘等。

在 UnixV6++ 的缓存控制模块的基础上进行了裁剪和修改，初始化缓存队列的内容放到了构造函数里，在系统初始化时即进行缓存队列的初始化；根据本二级文件管理系统的硬盘系统提供的接口和简化的缓存队列的结构重写了缓存管理系统的 GetBlk 函数，Bread 函数，Bwrite 函数，Bflush 函数；省略了释放控制块的函数（因为只有一个进程，所以可以在要分配新的缓存块时直接进行淘汰和重新分配）；同时简化了等待，异步读、异步写等相关的函数，用 buf 中的 is_dirty 属性来表示缓存块是否为脏，以判断是否需要特定的时候进行写回；去除了进程图像相关的函数；简化了错误信息处理的相关函数；只保留了剩余的函数所需的属性，最终的缓存管理模块的设计如上。

3.3 文件系统模块

3.3.1 SuperBlock 类设计

```
class SuperBlock {
    /* Functions */
public:
    SuperBlock();
    ~SuperBlock();

    void ResetSuperBlock();

    /* Members */
public:
    int s_isize; //外存Inode区占用的盘块数
    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

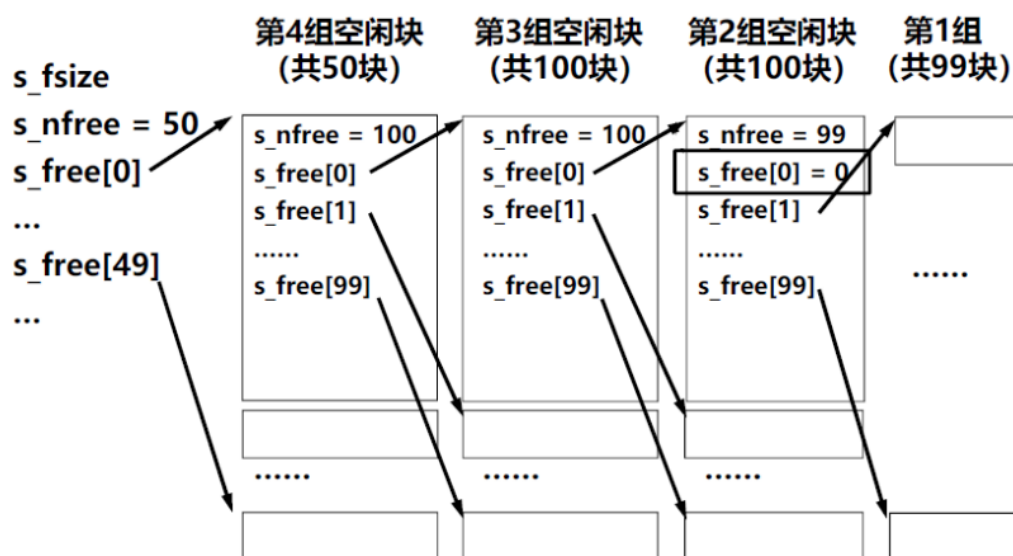
    int s_ninode; //直接管理的空闲外存Inode数量
    int s_inode[100]; //直接管理的空闲外存Inode索引表

    int s_fmod; //内存中super block副本被修改标识，意味着需要更新外存对应的Super Block

    int padding[51]; //填充，使之占据两个扇区
};
```

SuperBlock 类定义了文件系统超级块的相关属性和内容，用于进行文件系统的整体配置和管理。

属性包含 Inode 文件系统的盘块分配信息，用于空闲盘块管理的相关信息，用于空闲外存管理的相关信息，是否被修改，以及填充信息。SuperBlock 采用如下方式对空闲盘块进行管理：



相较于 Unix V6++ 原有的 SuperBlock 结构裁剪去锁相关属性，只读属性，更新时间属性，

补充对应字节的 padding。

3.3.2 FileSystem 类设计

```
class FileSystem {
public:
    /* static const */
    static const int SUPER_BLOCK_SECTOR_NUMBER = 0; //定义SuperBlock在磁盘上的扇区号

    static const int ROOTINO = 1; //文件系统根目录外存Inode编号

    static const int INODE_NUMBER_PER_SECTOR = 8; //外存Inode对象长度为64字节，每个磁盘块可以存放512/64=8个外存Inode
    static const int INODE_ZONE_START_SECTOR = 2; //外存Inode位于磁盘上的起始扇区号
    static const int INODE_SIZE = 1024 - 2; //磁盘上外存Inode区占据的扇区数

    static const int DATA_ZONE_START_SECTOR = 1024; //数据区起始扇区号
    static const int DATA_ZONE_END_SECTOR = 18000 - 1; //数据区的结束扇区号
    static const int DATA_ZONE_SIZE = 18000 - DATA_ZONE_START_SECTOR; //数据区占据的扇区数量

    /* Functions */
public:
    FileSystem();
    ~FileSystem();

    void LoadSuperBlock(); //系统初始化时读入SuperBlock

    void Update(); //将SuperBlock对象的内存副本更新到存储设备的SuperBlock中去

    Inode* IAlloc(); //分配一个空闲外存Inode

    void IFree(int number); //释放编号为number的外存Inode

    Buf* Alloc(); //分配空闲磁盘块

    void Free(int blkno); //释放编号为blkno的磁盘块
};
```

定义了文件系统全局的一些配置参数，提供对 Inode 和磁盘相关操作的接口。

属性包含了文件系统的一些静态的参数，包含 SuperBlock 在磁盘的扇区号，根目录的外存 Inode 号，外存 Inode 长度，Inode 区的起始扇区号，数据区起始扇区号，数据区的结束扇区号，数据区占据的扇区数量等。

方法包含读入 SuperBlock，将内存的 SuperBlock 更新到外存，Inode 的分配和回收，磁盘块的分配和释放。

该类基于 UnixV6++设计思想，裁剪去根据设备号获取 SuperBlock 的函数（因为本系统中只有一个设备），略去装配函数和检查 Block 是否坏的函数（UnixV6++里面好像是个空的函数）。略去 m_Mount 属性和 updlock 属性。

3.4 打开文件管理模块

3.4.1 File 类设计

```
class File {
public:
    File();
    ~File();

    bool is_av; //是否空闲
    Inode* f_inode; //指向打开文件的内存Inode指针
    int f_offset; //文件读写位置指针
};
```

打开文件控制块类，记录了打开的文件的相关信息。

属性包含了该控制块是否空闲，指向打开文件的内存 Inode 指针以及文件的读写位置。

在本系统中对 UnixV6++中的结构进行了裁剪和修改，去掉了读写要求（打开的文件均可读可写），去掉了进程相关的 f_count 属性，修改后的类如上图。

3.4.2 OpenFiles 类设计

```
class OpenFiles {
    /* static members */
public:
    static const int NOFILES = 100; //允许打开的最大文件数

    /* Functions */
public:
    OpenFiles();
    ~OpenFiles();

    int AllocFreeSlot(); //打开文件时，在打开文件描述符表中分配一个空闲表项
    File* GetF(int fd); //根据文件描述符参数找到对应的打开文件控制块File结构
    void SetF(int fd, File* pFile); //为已分配到的空闲描述符fd和空闲File对象建立勾连关系

public:
    File* OpenFileTable[NOFILES]; //File对象指针数组，指向系统打开文件表中的File对象
};
```

用户（当前系统只有单用户，后续添加多用户）打开文件描述符表，维护某用户打开的所有文件。

属性包括允许某用户打开的最大的文件数量（当前当用户系统为 100，之后在多用户系统中会根据实际情况进行缩减）。

方法包含分配空闲表项，获取文件描述符对应的 File 结构，为已经分配的空闲描述符和 File 对象建立勾连关系。

略去了 UnixV6++中的 Clone 函数，因为这个函数之后没用到。

3.4.3 IOParameter 类设计

```
class IOParameter {
    /* Function */
public:
    IOParameter();
    ~IOParameter();

    /* Members */
public:
    unsigned char* m_Base; //当前读写目标区域首地址
    int m_Offset; //当前读、写文件的字节偏移量
    int m_Count; //当前还剩余的读、写字节数量
};
```

文件 IO 参数类，记录了对文件读写需要用到的读写偏移量、字节数以及目标区域的首地址参数。

属性包括读写目标区域的首地址，当前读写文件的字节偏移量，还剩余的读写字节数量。

3.4.4 OpenFileTable 类设计

```
class OpenFileTable {
public:
    /* static consts */
    static const int NFILE = 100; //打开文件控制块File结构的数量

    /* Functions */
public:
    OpenFileTable();
    ~OpenFileTable();

    File* FAlloc(); //在系统打开文件表中分配一个空闲的File结构

    void CloseF(File* pFile); //释放File结构

    void ResetOpenFileTable(); //重置

    /* Members */
public:
    File m_File[NFILE];
};
```

负责对打开文件机构的管理，为打开文件建立内核数据结构之间的勾连关系。

属性包含系统打开文件表，是所有用户共享的（目前还只有一个）。

方法包含分配空闲 File 结构，释放 File 结构等。

3.4.5 Inode 类设计

```
class Inode {
public:
    /* static const member */
    static const int BLOCK_SIZE = 512; //文件逻辑块大小
    static const int ADDRESS_PER_INDEX_BLOCK = BLOCK_SIZE / sizeof(int); //每个索引块包含的物理盘块号

    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：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

    /* Functions */
public:
    Inode();
    ~Inode();

    void ReadI(); //根据Inode对象中的物理磁盘块索引表，读取相应文件数据

    void WriteI(); //根据Inode对象中的物理磁盘块索引表，将数据写入文件

    int Bmap(int lbn); //将文件的逻辑块号转换成对应的物理盘块号

    void IUpdate(); //更新外存Inode

    void ITrunc(); //释放Inode对应文件占用的磁盘块

    void Clean(); //清空Inode对象的数据

    void ICopy(Buf* bp, int number); //将包含外存Inode字符块中信息拷贝到内存Inode中

    /* Members */
public:
    bool is_changed; //该Inode是否被修改过，需要更新到外存

    unsigned int i_mode; //该Inode是否被使用以及类型

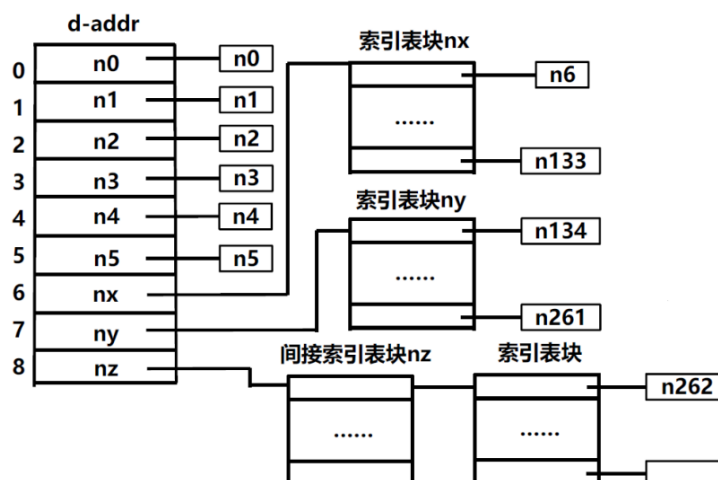
    int i_nlink; //文件连结计数

    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
};
```

系统中每一个打开的文件、当前访问目录都对应唯一的内存 Inode，其记录文件或者目录的相关信息。

其属性包含更新标志位，是否被使用以及其类型（文件或者目录），文件连结计数（该 Inode 连结了多少了路径），外存 Inode 区中的编号，文件的大小，用于文件逻辑块和物理块号进行转换的基本索引表，索引采用 UnixV6++的混合索引机制。



其方法包含：读取对应的文件的数据，将数据写入文件，根据文件的逻辑块号将其转化为物理块号，更新外存 Inode，释放 Inode 对应文件的磁盘块，清空 Inode 数据，将外存 Inode 数据复制到内存 Inode 中。

本类在 UnixV6++ 的 Inode 结构的基础上进行裁剪和修改：只保留了二级文件管理系统所需的属性，即是否被修改过和是否被使用；

3.4.6 DiskInode 类设计

```
class DiskInode {
    /* Functions */
public:
    DiskInode();
    ~DiskInode();

    /* Members */
public:
    unsigned int d_mode; //该inode是否被使用
    int d_nlink; //文件连结计数

    int d_size; //文件大小，字节为单位
    int d_addr[10]; //用于文件逻辑块号和物理块号的基本索引表

    int padding[3]; //16字节填充
};
```

定义外存索引节点，存储于文件存储设备上的外存 Inode 区中，记录对应文件的控制信息，长度为 64 字节。

其属性包括 d_mode，表明其是否被使用以及对应的文件类型的属性，文件的连结计数，文件的大小，用于文件逻辑块和物理块转换的基本索引表，以及填充。

本类基于 UnixV6++ 的思想，对 UnixV6++ 的 DiskInode 进行了裁剪，在当前单用户的条件下，去除了用户相关，并裁剪掉了修改时间相关属性。

3.4.6 InodeTable 类设计

```
class InodeTable {
    /* static consts */
public:
    static const int NINODE = 100; //内存Inode数量

    /* Functions */
public:
    InodeTable();
    ~InodeTable();

    Inode* IGet(int inumber); //根据Inode编号将其读入内存中

    void IPut(Inode* pNode); //若已经没有目录项指向它，则释放此文件占用的磁盘块

    void UpdateInodeTable(); //将所有被修改过的内存Inode更新到对应外存Inode中

    int IsLoaded(int inumber); //检查编号为inumber的外存Inode是否有内存拷贝

    Inode* GetFreeInode(); //在内存Inode表中寻找一个空闲的内存Inode

    void ResetInodeTable(); //重置

    /* Members */
public:
    Inode m_Inode[NINODE]; //内存Inode数组
};
```

内存 Inode 表类，负责内存 Inode 的分配和释放等操作。

属性包含内存 Inode 数组。

方法包含读入 Inode 到内存；释放文件占用的磁盘块，更新 Inode 到外存，检查 Inode 是否已经在内存，寻找空闲的内存 Inode。

该类基于 UnixV6++的设计思想，在其基础上进行了裁剪和修改，由于我每次使用 FileSystem 类都是通过 Kernel 获取，因此去掉了其 m_FileSystem 属性，因此也去掉了为其进行初始化的 Initialize 函数；将 IGet, IPut, IsLoaded 函数的参数修改为不需要设备号的形式（只有一个设备）；

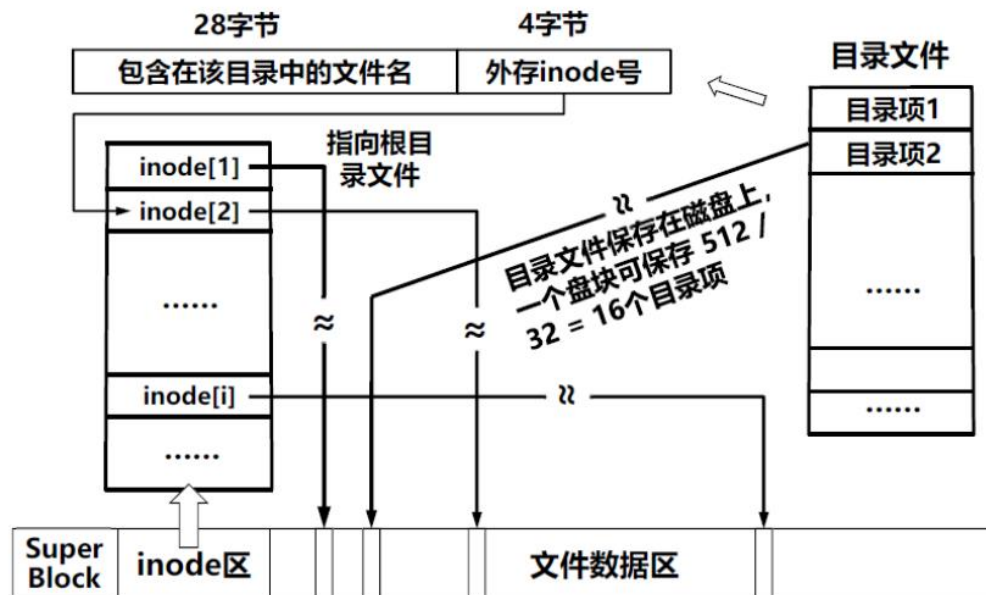
3.5 文件管理模块

3.5.1 DirectoryEntry 类设计

```
class DirectoryEntry {  
    /* static members */  
public:  
    static const int DIRSIZE = 28; //目录项中路径部分的最大字符串长度  
  
    /* Functions */  
public:  
    DirectoryEntry();  
    ~DirectoryEntry();  
  
public:  
    int m_ino; //目录项中Inode编号部分  
    char m_name[DIRSIZE]; //目录项中路径名部分  
};
```

目录项类，存储目录项对应文件的 Inode 编号和路径名称，大小为 32 字节。

属性包含目录项的 Inode 编号和目录项中路径名部分，用于找到目录项中的文件，示意图如下：



3.5.2 FileManager 类设计

```
class FileManager {
public:
    //目录搜索模式
    enum DirectorySearchMode {
        OPEN = 0, //以打开文件方式搜索目录
        CREATE = 1, //新建方式
        DELETE = 2 //删除方式
    };

    /* Functions */
public:
    FileManager();
    ~FileManager();

    void FormatSystem(); //格式化磁盘

    void Open(); //打开文件的系统调用

    void Creat(); //创建文件的系统调用

    void OpenI(Inode* pNode, int trf); //Open()和Create()系统调用的公共部分

    void Close(); //关闭文件系统调用

    void Seek(); //Seek系统调用

    void FStat(); //获取文件信息

    void Stat(); //Fstat获取文件信息

    void StatI(Inode* pNode, unsigned long long statBuf); //FStat()和Stat()的共享例程

    void Read(); //Read()系统调用

    void Write(); //Write()系统调用

    void Rdwr(int mode); //读写系统公用系统调用

    Inode* NameI(char (*func)(), enum DirectorySearchMode mode); //目录搜索，将路径转化为相应的Inode

    static char NextChar(); //获取路径中的下一个字符

    Inode* MakNode(unsigned int mode); //用于为创建新文件分配内核资源

    void WriteDir(Inode* pNode); //向父目录的目录文件写入一个目录项

    void SetCurDir(char* pathname); //设置当前工作路径

    void ChDir(); //改变当前工作目录

    void Link(); //创建文件的异名引用

    void UnLink(); //取消文件

    void Remove(); //删除文件或者文件夹

    void ResetFileManager(); //重置
```

封装了文件系统的各种系统调用在核心态下的处理过程。
属性包含指向根目录的内存 Inode 的指针。

方法包含了各种系统调用在核心态下的处理过程（如 Open, Creat, Seek 等）以及辅助方法（如 NameI, NextChar 等）

本类基于 UnixV6++ 的 FileManager 类思想，保留了单用户二级文件系统必要的方法，并对函数进行了修改，去除了进程等其他本次实验中不用的部分，并针对一些特定的系统调用添加了一定函数，方便调用，如 Remove。

3.6 核心模块

3.6.1 Kernel 类设计

```
class Kernel {
public:
    Kernel();
    ~Kernel();
    static Kernel& Instance();

    BufferManager& GetBufferManager();
    DiskManager& GetDiskManager();
    FileManager& GetFileManager();
    FileSystem& GetFileSystem();
    State& GetState();
    OpenFileTable& GetOpenFileTable();
    InodeTable& GetInodeTable();
    SuperBlock& GetSuperBlock();
    Space& GetSpace();

private:
    void InitBuffer();
    void InitFileSystem();
    void InitState();
    void InitSpace();

private:
    static Kernel instance; //Kernel单元类实例

    BufferManager* m_BufferManager = NULL;
    DiskManager* m_DiskManager = NULL;
    FileManager* m_FileManager = NULL;
    FileSystem* m_FileSystem = NULL;
    State* m_State = NULL;
    OpenFileTable* m_OpenFileTable = NULL;
    InodeTable* m_InodeTable = NULL;
    SuperBlock* m_SuperBlock = NULL;
    Space* m_Space = NULL;
};
```

基于 UnixV6++ 的设计思想，采用单例模式，将二级文件系统中要求只有一个的类放于 Kernel 类中，通过其属性中的指针指向对应的全局变量，并设置对应的方法来获取对应的模

块。裁剪去了完整操作系统的其他模块，如进程。

由于当前设计只有一个块设备用于存储，所以只有一个 SuperBlock，将其放于 Kernel 中；当前设计只有一个用户，将 User 修改为 State，放于 Kernel 中；由于没有用户空间，所以单独设计了一个 Space 类来替代，也将其放于 Kernel 中；本系统中 OpenFileTable 和 InodeTable 也分别只有一个，为了方便管理，也将其放于 Kernel 中，最终的结果如上图。

3.6.2 State 类设计

```
class State {
public:
    int arg[10] = {}; //系统调用的参数
    char* dirp = NULL; //系统调用参数，一般用于Pathname指针

    Inode* cdir = NULL; //指向当前目录的Inode指针
    Inode* pdir = NULL; //指向父目录的Inode指针

    DirectoryEntry dent; //当前目录的目录项
    char dbuf[DirectoryEntry::DIRSIZE] = {}; //当前路径分量
    char curdir[128] = {}; //当前工作目录的完整路径

    OpenFiles ofiles; //打开文件描述符对象

    IOParameter IOParam; //记录当前读、写文件的偏移量，用户目标区域和剩余字节参数

public:
    State();
    ~State();

    void ResetState(); //重置状态
};
```

本系统当前的设计没有考虑多用户，会在之后的设计中进行完善。当前的 State 类对应原 UnixV6++中的 User 类，存储用户相关信息。

属性包含系统调用的参数，当前目录的 Inode 指针和指向父目录的 Inode 指针，当前目录的目录项，当前的路径分量，当前工作目录的完整路径，打开文件描述符对象和当前的 IO 参数。

本类相较于 UnixV6++的 User 类只保留了二级文件管理系统所必须的文件管理相关的属性。

3.6.3 Space 类设计

```
class Space {  
    /* Members */  
public:  
    char pathParam[128]; //路径参数  
    unsigned char* buffer; //用于读写的缓冲区  
  
    /* Functions */  
public:  
    Space();  
    ~Space();  
  
    void ResetSpace(); //重置  
};
```

该类用于模拟用户空间，进行 IO 操作。
属性包含路径参数和用于读写的缓存（内存空间）。

3.7 main 函数

3.7.1 获取各个模块

```
//全局变量  
FileManager& fileManager = Kernel::Instance().GetFileManager();  
DiskManager& diskManager = Kernel::Instance().GetDiskManager();  
State& state = Kernel::Instance().GetState();  
FileSystem& fileSystem = Kernel::Instance().GetFileSystem();  
InodeTable& inodeTable = Kernel::Instance().GetInodeTable();  
BufferManager& bufferManager = Kernel::Instance().GetBufferManager();  
SuperBlock& superBlock = Kernel::Instance().GetSuperBlock();  
OpenFileTable& openFileTable = Kernel::Instance().GetOpenFileTable();  
Space& space = Kernel::Instance().GetSpace();
```

获取文件管理系统的各个模块，方便后续使用。

3.7.2 指令处理函数

```
//用来将输入的指令转化为参数的函数
void transInstruction(string instruction) {
    string op;
    string tmp;
    State& state = Kernel::Instance().GetState();
    Space& space = Kernel::Instance().GetSpace();

    int segCount = 0; //计数指令以空格分开的段数
    char* p = (char*)state.arg;
```

获取用户的输入，校验指令的正确性，将对应的参数放入到指定的位置，供系统调用函数使用。

3.7.3 系统调用函数

<pre>//format函数 >void format() { ... }</pre>	<pre>//unlink函数 >void unlink() { ... }</pre>
<pre>//open函数 >void open() { ... }</pre>	<pre>>void rm() { ... }</pre>
<pre>//close函数 >void close() { ... }</pre>	<pre>//ls函数 >void ls() { ... }</pre>
<pre>//lseek函数 >void lseek() { ... }</pre>	<pre>//cd函数 >void cd() { ... }</pre>
<pre>//read函数 >void read() { ... }</pre>	<pre>//mkdir函数 >void mkdir() { ... }</pre>
<pre>//write函数 >void write() { ... }</pre>	<pre>//fin函数 >void fin() { ... }</pre>
<pre>//create函数 >void create() { ... }</pre>	<pre>>void fout() { ... }</pre>
<pre>//cat函数 >void cat() { ... }</pre>	<pre>//shutdown函数 >void shutdown() { ... }</pre>
<pre>//link函数 >void link() { ... }</pre>	<pre>//exit函数 >void exit() { ... }</pre>
	<pre>>void test() { ... }</pre>

使用 FileManager 类中提供的内核功能实现的文件管理系统的各个系统调用。

3.7.4 主模块

```
int main() {  
    ifstream f("SecondaryFS.img");  
    if (!f.good()) {  
        cout << "no disk, want format?(y/n)? " << endl;  
        char choice;  
        while (true) {  
            choice = _getch();  
            if (choice == 'y') {  
                diskManager.createDisk();  
                format();  
                break;  
            }  
            else if (choice == 'n') {  
                return 0;  
            }  
        }  
    }  
}
```

程序的入口,负责二级文件管理系统的初始化,提供交互和执行对应的指令,打印结果。

四. 代码测试

4.1 实验环境

宿主机: Windows11, 64 位

编译环境: gcc version 13.2.0

4.2 使用说明

在 Secondary_Cmake 目录下打开 Window 命令行:

```
(c) Microsoft Corporation. 保留所有权利。  
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake>
```

创建构建目录并进入目录生成构建系统:

```
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake>mkdir build  
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake>cd build  
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake\build>cmake -G "MinGW Makefiles" ..  
-- The C compiler identification is GNU 13.2.0  
-- The CXX compiler identification is GNU 13.2.0  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Check for working C compiler: C:/Users/20930/mingw64/bin/gcc.exe - skipped  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Check for working CXX compiler: C:/Users/20930/mingw64/bin/c++.exe - skipped  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done (5.1s)  
-- Generating done (0.1s)  
-- Build files have been written to: D:/courses/OSCourseDesign/SecondaryFS/SecondaryFS_CMake/build
```

进行编译和链接：

```
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake\build>cmake --build .  
[ 8%] Building CXX object CMakeFiles/SecondaryFS.dir/main.cpp.obj  
[ 16%] Building CXX object CMakeFiles/SecondaryFS.dir/BufferSubSystem/BufferManager.cpp.obj  
[ 25%] Building CXX object CMakeFiles/SecondaryFS.dir/DiskSubSystem/DiskManager.cpp.obj  
[ 33%] Building CXX object CMakeFiles/SecondaryFS.dir/FileSubSystem/File.cpp.obj  
[ 41%] Building CXX object CMakeFiles/SecondaryFS.dir/FileSubSystem/FileManager.cpp.obj  
[ 50%] Building CXX object CMakeFiles/SecondaryFS.dir/FileSubSystem/FileSystem.cpp.obj  
[ 58%] Building CXX object CMakeFiles/SecondaryFS.dir/FileSubSystem/INode.cpp.obj  
[ 66%] Building CXX object CMakeFiles/SecondaryFS.dir/FileSubSystem/OpenFileManager.cpp.obj  
[ 75%] Building CXX object CMakeFiles/SecondaryFS.dir/Kenel/Kernel.cpp.obj  
[ 83%] Building CXX object CMakeFiles/SecondaryFS.dir/Kenel/Space.cpp.obj  
[ 91%] Building CXX object CMakeFiles/SecondaryFS.dir/Kenel/State.cpp.obj  
[100%] Linking CXX executable SecondaryFS.exe  
[100%] Built target SecondaryFS
```

执行程序：

```
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake\build>SecondaryFS.exe  
no disk, want format?(y/n)?
```

4.3 指令说明和测试

4.3.1 format 指令

对文件系统进行格式化，清空磁盘，重建结构，使其回到初始状态（在没有磁盘镜像文件时初始文件并格式化）

指令格式：format

```
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake\build>SecondaryFS.exe  
no disk, want format?(y/n)?  
/:>format  
/:>|
```

执行指令后磁盘已经被初始化。

4.3.2 ls 指令

显示目录下的文件或者文件夹

指令格式：ls：显示当前目录下的文件或目录；ls [path]：显示指定目录下的文件或者目录。

```
/:>ls  
bin etc dev home shell  
/:>ls dev  
tty1  
/:>|
```

如图，ls 指令显示了当前路径和指定路径的文件和目录

4.3.3 cd 指令

进入到指定的目录下

指令格式：cd [path]

```
/:>cd bin
/bin:>cd /dev
/dev:>cd ..
/:>|
```

如图，根据 cd 指令指定的路径进入了不同的目录

4.3.4 mkdir 指令

在指定路径创建目录。

指令格式: mkdir [path]

```
/:>mkdir home/test
/:>ls home
test
/:>|
```

如图，mkdir 指令成功在 home 目录下创建 test 目录

4.3.5 create 指令

在指定路径创建文件

指令格式: mkdir [path]

```
/:>create home/test/test0.txt
/:>ls home/test
test0.txt
/:>|
```

如图，在 home/test 目录下创建了 test0.txt 文件

4.3.6 rm 指令

删除文件或者文件夹（递归删除其下所有文件和文件夹）

指令格式: rm [path]

```
/:>rm home/test/test0.txt
/:>ls home/test
no file in this directory
/:>|
```

如图，删除了 home/test 目录下的 test0.txt 文件

4.3.7 open 指令

打开文件（分配 File 结构进行管理，用于读写）

指令格式: open [path]

4.3.8 close 指令

关闭文件

指令格式: `close [path]`

说明: 因为是在命令行输入指令, 所以不用文件管理符 `fd` 作为参数, 使用了文件路径作为参数

4.3.9 read 指令

将文件内容读入用户空间

指令格式: `read [path] [num]`, `num` 表示要读写的字节数量

4.3.10 write 指令

将用户空间中的内容写入文件

指令格式: `wirte [path] [num]`

4.3.11 lseek 指令

移动文件的读写指针

指令格式: `lseek [path] [num] [mode]`, `mode` 表示移动的起点, `SEEK_SET` 为从头开始, `SEEK_CUR` 为从当前开始, `SEEK_END` 为从文件末尾开始

以上几条指令在命令行界面难以看到结果, 故没有截图演示, `fin` 和 `fout` 指令基于上述几条指令完成, 可以通过该两条指令看到现象。

4.3.12 link 指令

为在指定目录下为文件创建链接

指令格式: `link [path1] [path2]` `path1` 为被链接文件或者目录, `path2` 为链接到的路径

```
/:>link home/test home/test1
/:>ls home
test test1
/:>
```

如图, 成功建立了链接

4.3.13 unlink 指令

取消指定目录的链接

指令格式: `unlink [path]`












```
/:>unlink home/test1
/:>ls home
test
/:>
```

如图，成功取消了链接

4.3.14 fin 指令

将外部文件输入二级文件管理系统

指令格式：fin [path1] [path2] path1 为外部文件的路径，path2 为在二级文件管理系统内的路径，将准备好的不同大小和类型的测试文件粘贴到文件夹内（test5 为第一次作业的报告）：

	cmake_install.cmake	2024/4/8 19:59	CMake 源文件	2 KB
	CMakeCache.txt	2024/4/8 19:59	文本文档	17 KB
	Makefile	2024/4/8 19:59	文件	16 KB
	SecondaryFS.exe	2024/4/8 20:01	应用程序	134 KB
	SecondaryFS.img	2024/4/8 20:21	光盘映像文件	9,000 KB
	test1.txt	2024/3/26 10:14	文本文档	3 KB
	test2.txt	2024/3/27 13:57	文本文档	113 KB
	test3.txt	2024/3/26 21:19	文本文档	1,819 KB
	test4.jpg	2024/3/27 13:41	JPG 文件	293 KB
	test5.pdf	2024/4/1 10:19	Microsoft Edge PD...	1,139 KB

```
/:>ls home/test
no file in this directory
/:>fin test1.txt home/test/test1_in.txt
/:>fin test2.txt home/test/test2_in.txt
/:>fin test3.txt home/test/test3_in.txt
/:>fin test4.jpg home/test/test4_in.jpg
/:>fin test5.pdf home/test/test5_in.pdf
/:>ls home/test
test1_in.txt test2_in.txt test3_in.txt test4_in.jpg test5_in.pdf
/:>
```

依次 fin 5 个测试文件，5 个文件成功输入二级文件管理系统

4.3.15 cat

查看指定路径的文件内容

指令格式：cat [path]

```

/:>cat home/test/test1_in.txt
jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test fi
jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test fil
ng's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file
g's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file
's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file j
s test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file ji
test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jin
test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing
est file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing'
st file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's
t file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's
file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's t
file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's te
ile jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's tes
le jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test
e jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test
jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test f
jing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test fi
ing's test file jing's test file jing's test file jing's test file jing's test file jing's test file jing's test fil
ng's test file jing's test file

```

如图，成功显示文件内容

4.3.16 fout

将二级文件管理系统中的文件输出到宿主机











指令格式：fout[path1][path2] path2 为外部文件的路径，path1 为在二级文件管理系统内的路径

```

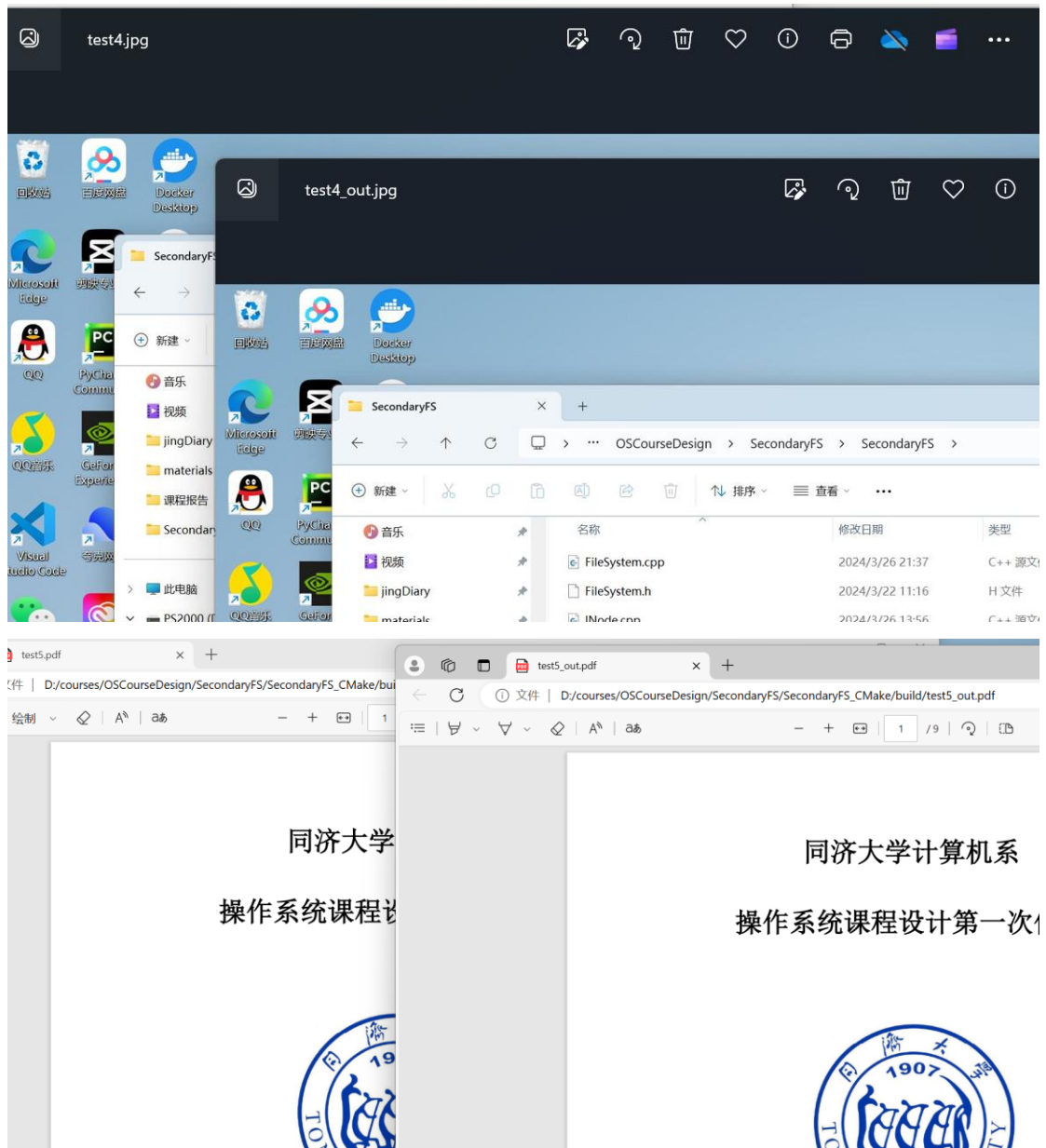
/:>fin test1.txt test1.txt
/:>fout test1.txt test1_out.txt
/:>fin test2.txt test2.txt
/:>fout test2.txt test2_out.txt
/:>fin test3.txt test3.txt
/:>fout test3.txt test3_out.txt
/:>fin test4.jpg test4.jpg
/:>fout test4.jpg test4_out.jpg
/:>fin test5.pdf test5.pdf
/:>fout test5.pdf test5_out.pdf
/:>|

```

依次输入 5 个测试文件并输出，结果如下：

	test1.txt	2024/3/26 10:14	文本文档	3 KB
	test1_out.txt	2024/4/8 21:06	文本文档	3 KB
	test2.txt	2024/3/27 13:57	文本文档	113 KB
	test2_out.txt	2024/4/8 21:07	文本文档	113 KB
	test3.txt	2024/3/26 21:19	文本文档	1,819 KB
	test3_out.txt	2024/4/8 21:07	文本文档	1,819 KB
	test4.jpg	2024/3/27 13:41	JPG 文件	293 KB
	test4_out.jpg	2024/4/8 21:08	JPG 文件	293 KB
	test5.pdf	2024/4/1 10:19	Microsoft Edge PD...	1,139 KB
	test5_out.pdf	2024/4/8 21:08	Microsoft Edge PD...	1,139 KB

文件大小完全一致，查看文件内容：



也完全一致，可见该文件系统对小、大、巨型文件都能够正常的进行输入输出

4.3.17 exit 指令

直接退出系统，缓存不落盘，相当于掉电

指令格式：exit

```

/ :>format
/ :>ls
bin etc dev home shell
/ :>create test.txt
/ :>ls
bin etc dev home shell test.txt
/ :>exit
already exit

```

```
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake\build>SecondaryFS.exe
/:>ls
bin etc dev home shell
```

图中可看出所作操作没有更新到磁盘

4.3.18 shutdown 指令

将缓存更新到磁盘后退出

指令格式: shutdown

```
D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake\build>SecondaryFS.exe
/:>ls
bin etc dev home shell
/:>create test.txt
/:>ls
bin etc dev home shell test.txt
/:>shutdown
already exit

D:\courses\OSCourseDesign\SecondaryFS\SecondaryFS_CMake\build>SecondaryFS.exe
/:>ls
bin etc dev home shell test.txt
/:>
```

由图可知 shutdown 指令在退出前将修改更新到了磁盘