

同濟大學

操作系统课程设计 实验报告



学 号: 21****9
姓 名: *****
专 业: 计算机科学与技术
指导老师: 邓蓉
日 期: 2024. 4. 19

目录

1 任务描述.....	3
1.1 目的	3
1.2 内容	3
1.3 要求	3
2 概要设计.....	4
3 详细设计.....	5
3.1 磁盘模块设计	5
3.2 高速缓存模块设计	6
3.2.1 Buf 类定义.....	6
3.2.2 BufferManager 类定义	6
3.3 文件系统资源管理模块设计.....	8
3.3.1 Inode 类定义.....	8
3.3.2 DiskInode 类定义	10
3.3.3 SuperBlock 类定义.....	11
3.4 文件管理接口设计	12
3.4.1 InodeTable 类定义	12
3.4.2 OpenFileTable 类定义	13
3.4.3 文件系统资源模块定义.....	15
3.5 文件管理总接口设计	16
3.6 用户模块设计	19
3.7 顶层设计	21
3.7.1 API 封装	21
3.7.2 Common 类设计	23
3.7.3 系统初始化流程.....	24
4 代码测试.....	25
4.1 实验环境	25
4.2 功能测试	26
5 调试时遇到的问题.....	31
5.1 初始化的先后顺序.....	31
5.2 文本入出正常但图片读入出时被提前截断	31
5.3 文件指针及 offset 位置的理解	31
6 实验总结.....	32
6.1 实验流程	32
6.2 实验感受	32

1 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

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：

- 1、使用磁盘镜像文件 c:\SecondaryFS.img，存储整个文件卷中的所有信息。一个文件卷实际上就是一张逻辑磁盘，磁盘存储的信息以块为单位。每块 512 字节。
- 2、c:\SecondaryFS.img 文件是一个标准 UNIX V6++文件卷，格式如下：

超级块 SuperBlock	Inode区	数据区 DataBlock
-------------------	--------	------------------

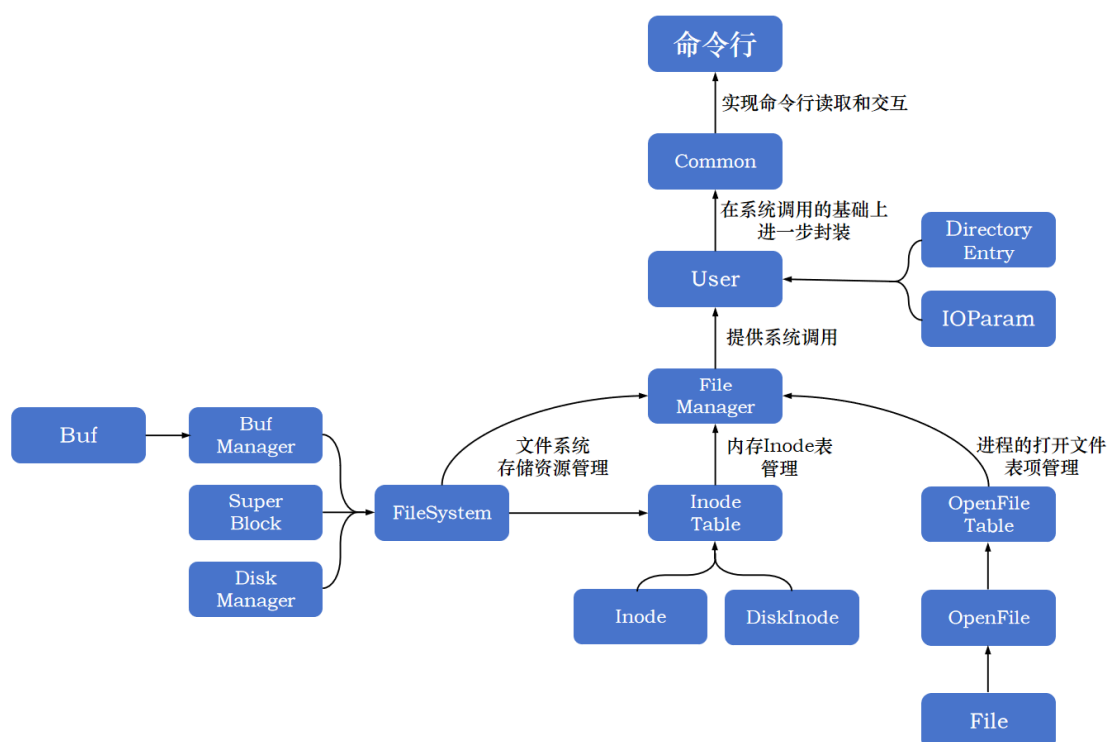
- 3、基于上述思想设计实现文件系统内核，实现以下系统调用：
 - 针对普通文件：open, close, lseek, read, write, create
 - 针对目录树：link, unlink, ls, mkdir
- 4、编写 2 个应用程序：
 - Initialize 程序，格式化二级文件系统 c:\SecondaryFS.img
 - SecondaryFS 程序，接收客户端输入的文件操作命令，访问二级文件系统。
- 5、为二级文件系统 SecondaryFS 设计一个简单的测试用户接口：
 - 文件系统访问命令，每个命令行对应一个系统调用。命令行的第一个字符串是命令 f****，对应 1.4 节介绍的一个系统调用 ****。
 - 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。
 - 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。
 - 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。
 - 命令 shutdown，安全关闭二级文件系统 SecondaryFS。将脏缓存写回镜像文件。
 - 命令 exit，关闭程序 SecondaryFS。相当于断电，文件系统会丢数据。

2 概要设计

本次课程设计的目标是实现一个单用户单进程的二级文件管理系统。在本系统中，使用一个二进制大文件模拟磁盘，每 512 个字节分为一个数据段，用以模拟磁盘的各个扇区。事实上，Unix V6++中很多用以管理一级文件系统均可用来管理二级文件系统，区别主要在于磁盘驱动接口，在一级文件系统中，是通过文件系统向硬盘发 DMA 命令，来在物理磁盘上读写数据，而在二级文件系统中，依托于宿主机的一级文件系统，我们可以通过操作系统提供的 read 和 write 系统调用来对虚拟磁盘镜像进行读写操作。基于此，再结合 Unix v6++的文件管理和高速缓冲管理，本系统分为以下几个模块：

- 磁盘驱动 DiskManager 模块：初始化磁盘镜像文件，调用 C++库对其进行读写；
- 高速缓存管理 BufManager 模块：负责管理系统中的所有缓存块，包括缓存的分配、回收、调用磁盘驱动读写缓存块等；
- 文件系统资源管理 FileSystem 模块：负责管理文件存储设备中的各类存储资源，包括、外存 Inode 的分配、释放等；
- 打开文件管理 OpenFileTable 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 的勾连关系等；
- 文件管理接口 FileManager 模块：负责提供对文件系统的操作接口，包括打开、删除、读写、 创建文件等操作；
- 用户 User 模块：包括封装 main 中各指令的函数，调用 FileManager 模块的接口；
- 顶层模块 main：包括系统初始化模块、API 模块，命令解析模块等，通过调用内核函数实现 API，负责直接与用户交互。

各模块调用关系如下图所示。



3 详细设计

本实验在 Windows 系统、VS2022 集成环境中开发。附件中提交源码，双击.sln 文件即可打开项目编译。

3.1 磁盘模块设计

DiskManager 类定义负责对镜像文件的初始化及的读写，定义如下。

```
class DiskManager;
extern DiskManager globalDiskManager;

class DiskManager {
public:
    /* 磁盘镜像文件名 */
    static const char* DISK_FILE_NAME;

private:
    /* 磁盘文件指针 */
    FILE* img_fp;

public:
    DiskManager();
    ~DiskManager();

    /* 检查镜像文件是否存在 */
    bool Exists();

    /* 打开镜像文件 */
    void OpenDisk();

    /* 实际写磁盘函数 */
    void write(const void* buffer, unsigned int size,
              int offset = -1, unsigned int origin = SEEK_SET);

    /* 实际读磁盘函数 */
    void read(void* buffer, unsigned int size,
             int offset = -1, unsigned int origin = SEEK_SET);
};
```

Common.h 中将声明全局磁盘管理对象 globalDiskManager 并将在 main.cpp 中与其他对象统一实例化，以保证唯一性。类中 Exist() 在系统运行之初调用，若不存在则自动在当前文件夹下创建 FS.img 镜像文件；OpenDisk() 调用 C++ 库函数 fopen 打开 FS.img，并将获得的句柄存入 img_fp 中，便于后续读写。Read()、Write() 函数用于 FS.img 文件的读写。

3.2 高速缓存模块设计

3.1.1 Buf 类定义

缓存控制块类，记录给定缓存块的使用情况等信息，兼任I/O请求块，记录该缓存相关的I/O请求和执行结果。构造函数将其中b_flags、b_resid、b_wcount置0； b_forw、b_back、b_addr 置NULL；将b_blkno、b_error置-1。

```
class Buf{
public:
    /* b_flags中标志位 */
    enum BufFlag{
        B_WRITE = 0x1,          /* 写操作。将缓存中的信息写到硬盘上去 */
        B_READ = 0x2,           /* 读操作。从盘读取信息到缓存中 */
        B_DONE = 0x4,           /* I/O操作结束 */
        B_ERROR = 0x8,          /* I/O因出错而终止 */
        B_BUSY = 0x10,          /* 相应缓存正在使用中 */
        B_WANTED = 0x20,        /* 有进程正在等待使用该buf管理的资源*/
        B_ASYNC = 0x40,         /* 异步I/O，不需要等待其结束 */
        B_DELWRI = 0x80         /* 延迟写，缓存要移做他用时，再将其内容写回磁盘 */
    };

public:
    unsigned int b_flags; /* 缓存控制块标志位 */

    Buf* b_forw;
    Buf* b_back;

    int b_wcount;          /* 需传送的字节数 */
    unsigned char* b_addr; /* 指向该缓存控制块所管理的缓冲区的首地址 */
    int b_blkno;           /* 磁盘逻辑块号 */
    int b_error;           /* I/O出错时信息 */
    int b_resid;           /* I/O出错时尚未传送的剩余字节数 */

    Buf();
    ~Buf();
};
```

3.1.2 BufferManager 类定义

缓存控制块类，负责缓存块和自由缓存队列的管理，包括分配、读写、回收等操作。本系统未实现预读，故去除了原有的 Breada()函数；由于是单进程，因此所有写操作也都是同步的，故去除了 Bawrite()函数；由于只有一个 FS.img 磁盘文件，因此本次实验也去除了设备号、挂载类 m_Mount 等部分。具体实现如下：

```
class BufferManager{
public:
```

```

/* static const member */
static const int NBUF = 100;          /* 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();

    void Initialize();

    Buf* GetBlk(int blkno); /* 申请一块缓存，用于读写设备dev上的字符块blkno。*/
    void Brelse(Buf* bp); /* 释放缓存控制块buf */

    Buf* Bread(int blkno); /* 读一个磁盘块。blkno为目标磁盘块逻辑块号。 */

    void Bwrite(Buf* bp); /* 写一个磁盘块 */
    void Bdwrite(Buf* bp); /* 延迟写磁盘块 */

    void ClrBuf(Buf* bp); /* 清空缓冲区内容 */
    void Bflush(); /* 将dev指定设备队列中延迟写的缓存全部输出到磁盘 */

    void Bformat();

    void Bdetach(Buf* bp);
    void Binsert(Buf* bp);

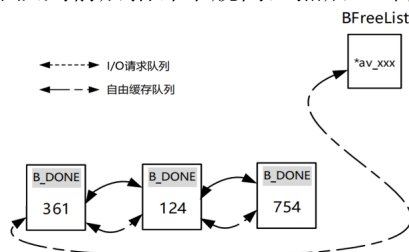
private:
    Buf* bFreeList; /* 自由缓存队列控制块 */
    Buf m_Buf[NBUF]; /* 缓存控制块数组 */
    unsigned char Buffer[NBUF][BUFFER_SIZE]; /* 缓冲区数组 */
    DiskManager* m_DeviceManager; /* 指向设备管理模块全局对象 */
    unordered_map<int, Buf*> m_map;

};

```

bFreeList 作为自由缓存队列的头指针，**m_DiskDriver** 指向磁盘驱动模块全局对象，通过调用其接口在缓存块和虚拟磁盘文件之间进行读写。

自由缓存队列管理为 LRU 算法，设置自由缓存队列为 NBUF=100 个。其链接示意图如下，将最近使用的缓存块从原队列前后指针中脱离，然后立即放入队列尾部。



3.3 文件系统资源管理模块设计

3.3.1 Inode 类定义

该类表示内存中的 Inode。系统中每一个打开的文件、当前访问目录都对应一个唯一的内存 Inode。通过 `i_number` 将其与对应外存 Inode 关联起来。由于不涉及异步写，所有使用、写回都是同步执行，故本系统未设置锁开关标志；用户权限检测函数、用户表示、组标识、磁盘号等成员也做了删除。

```
class Inode{
public:
    /* i_flag中标志位 */
    enum INodeFlag
    {
        ILOCK = 0x1,      /* 索引节点上锁 */
        IUPD = 0x2,      /* 内存inode被修改过，需要更新相应外存inode */
        IACC = 0x4,      /* 内存inode被访问过，需要修改最近一次访问时间 */
        IMOUNT = 0x8,    /* 内存inode用于挂载子文件系统 */
        IWANT = 0x10,    /* 有进程正在等待该内存inode被解锁 */
        ITEXT = 0x20     /* 内存inode对应进程图像的正文段 */
    };

    /* static const member */
    static const unsigned int IALLOC = 0x8000; /* 文件被使用 */
    static const unsigned int IFMT = 0x6000; /* 文件类型掩码 */
    static const unsigned int IFDIR = 0x4000; /* 文件类型：目录文件 */
    static const unsigned int IFCHR = 0x2000; /* 字符设备特殊类型文件 */
    static const unsigned int ILARG = 0x1000; /* 文件长度类型：大型或巨型文件 */
    static const unsigned int ISVTX = 0x200; /* 使用后仍然位于交换区上的正文段 */
    static const unsigned int IREAD = 0x100; /* 对文件的读权限 */
    static const unsigned int IWRITE = 0x80; /* 对文件的写权限 */
    static const unsigned int IEXEC = 0x40; /* 对文件的执行权限 */
    static const unsigned int IRWXU = (IREAD | IWRITE | IEXEC); /* 文件主执行权限 */
    static const int BLOCK_SIZE = 512; /* 文件逻辑块大小：512字节 */
    static const int ADDRESS_PER_INDEX_BLOCK = BLOCK_SIZE / sizeof(int);
    /* 每个间接索引表(或索引块)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：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 */
    static const int PIPSIZ = SMALL_FILE_BLOCK * BLOCK_SIZE;

public:
    unsigned int i_flag; /* 状态的标志位，定义见enum INodeFlag */
};
```



```

unsigned int i_mode; /* 文件工作方式信息 */

int i_count; /* 引用计数 */
int i_nlink; /* 文件联结计数，即该文件在目录树中不同路径名的数量 */
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]; /* 用于文件逻辑块好和物理块好转换的基本索引表 */

int i_lastr; /* 存放最近一次读取文件的逻辑块号，用于判断是否需要预读 */

public:
    /* Constructors */
    Inode();
    /* Destructors */
    ~Inode();

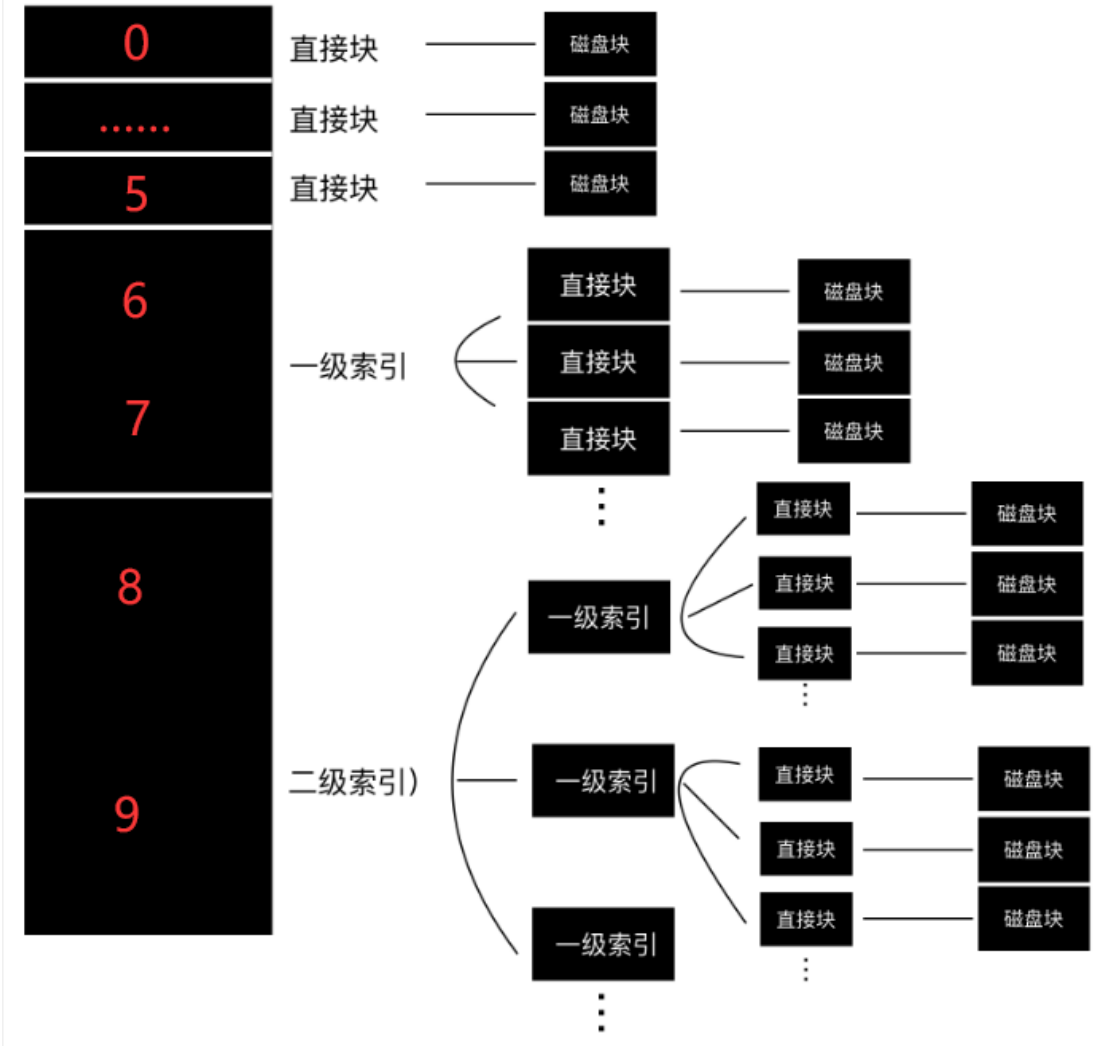
    /*
     * @comment 根据Inode对象中的物理磁盘块索引表，读取相应
     * 的文件数据
     */
    void ReadI();
    /*
     * @comment 根据Inode对象中的物理磁盘块索引表，将数据写入文件
     */
    void WriteI();
    /*
     * @comment 将文件的逻辑块号转换成对应的物理盘块号
     */
    int Bmap(int lbn);
    /*
     * @comment 更新外存Inode的最后的访问时间、修改时间
     */
    void IUpdate(int time);
    /*
     * @comment 释放Inode对应文件占用的磁盘块
     */
    void ITrunc();
    /*
     * @comment 清空Inode对象中的数据
     */
    void Clean();
    /*
     * @comment 将包含外存Inode字符块中信息拷贝到内存Inode中
     */

```

```
void ICopy(Buf* bp, int inumber);
};
```

值得注意的是：在Inode构造函数中不应当使用Clean清空Inode对象中的数据（旧文件信息）。因为Clean()函数主要是想用在IAAlloc()中清空新分配DiskInode的原有数据。它不能清除i_dev, i_number, i_flag, i_count, 这些是内存Inode而非DiskInode的旧文件信息，而Inode类构造函数需要将其初始化为无效值。Inode析构函数无需做处理，其内部数据将在FileSystem中的Update函数操作下统一写回磁盘。

这里 Unix v6++采用三级混合索引机制结构，在 Inode 及 DiskInode 类中有一个 d_addr[]数组，负责映射逻辑盘块号到物理盘块号。将逻辑盘块 n 对应的物理块号存放在 d_addr[n]中。将文件按大小分为三级，分别为小型文件（0~6 盘块，最大 3K）、大型文件（7~6+128*2 盘块）、巨型文件（128*2+7~6+128*2+128*128*2 盘块）。d_addr[]数组中 0#~5#为直接索引，直接记录对应数据物理盘块号；6#~7#为一次间接索引，它指向一个盘块号，再从该盘块中获取索引，一个盘块大小为 512 字节，int 型数据大小 4 字节，故可存储 128 个索引；8#~9#为二次间接索引，它指向的盘块为间接索引块，从该盘块获取对应索引表块，再从索引表块找到对应物理盘块。结构示意图如下：



3.3.2 DiskInode 类定义

以下是外存索引节点(DiskInode)的定义为方便管理，与Inode类在同一个文件中定义。外存Inode位于文件存储设备上的外存Inode区中。每个文件有唯一对应的外存Inode，

其作用是记录了该文件大小、存储位置、修改时间等控制信息。

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

    /* Members */
public:
    unsigned int d_mode; /* 状态的标志位，定义见enum INodeFlag */
    int d_nlink; /* 文件联结计数，即该文件在目录树中不同路径名的数量 */
    int d_size; /* 文件大小，字节为单位 */
    int d_addr[10]; /* 用于文件逻辑块好和物理块好转换的基本索引表 */
    int d_atime; /* 最后访问时间 */
    int d_mtime; /* 最后修改时间 */
};
```

3.3.3 SuperBlock 类定义

是本文件系统的资源管理模块，由于本系统为单用户单线程，去除了锁的使用，但由于 SuperBlock 大小要求是 1024 个字节占满两个扇区，故此处不删除锁相关变量。SuperBlock 控制块在程序运行时从文件中读入副本进入缓存，退出时若 s_fmod 存在改动则会在检查 Inode 改动时一起被写回磁盘。

```
class SuperBlock{
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量 */
    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_flock; /* 封锁空闲盘块索引表标志 */
    int s_iloc; /* 封锁空闲Inode表标志 */
    int s_fmod; /* 内存中super block副本被修改标志，需要更新到磁盘 */
    int s_ronly; /* 本文件系统只能读出 */
    int s_time; /* 最近一次更新时间 */
    int padding[47]; /* 填充使SuperBlock块大小等于1024字节，占据2个扇区 */

public:
    SuperBlock() {};
```

```
~SuperBlock() {};  
};
```

对于空闲盘块的管理采用成组链接法，按“栈的栈”进行管理，释放一盘块时，进栈，分配一盘块时，退栈。对所有空闲盘块分组，最后一组由 SuperBlock 直接管理，每个分组首盘块的开始 101 个字存放前一个分组的索引，第一组 24 为 99 块，故第二组首盘块的 0# 索引为 0 即为空闲盘块结束标志，基本结构示意图如下：



3.4 文件管理接口设计

3.4.1 InodeTable 类定义

InodeTable 类负责内存中 Inode 的分配和释放，此部分老师已给出 PDF 源码注释，除删去磁盘设备号 dev 形参外，没有太多改动。

```
class InodeTable{  
public:  
    static const int NINODE = 100; /* 内存Inode的数量 */  
    Inode m_Inode[NINODE]; /* 内存Inode数组，每个打开文件都会占用一个内存Inode */  
    FileSystem* m_FileSystem; /* 对全局对象g_FileSystem的引用 */  
  
public:  
    InodeTable();  
    ~InodeTable();  
    /*  
    * @comment 根据指定设备号dev，外存Inode编号获取对应  
    * Inode。如果该Inode已经在内存中，对其上锁并返回该内存Inode，  
    * 如果不在内存中，则将其读入内存后上锁并返回该内存Inode  
    */  
    Inode* IGet(int inumber);  
    /*  
    * @comment 减少该内存Inode的引用计数，如果此Inode已经没有目录项指向它，
```

```

    * 且无进程引用该Inode，则释放此文件占用的磁盘块。
    */
void IPut(Inode* pNode);
/*
    * @comment 将所有被修改过的内存Inode更新到对应外存Inode中
    */
void UpdateInodeTable();
/*
    * @comment 检查设备dev上编号为inumber的外存inode是否有内存拷贝，
    * 如果有则返回该内存Inode在内存Inode表中的索引
    */
int IsLoaded(int inumber);
/*
    * @comment 在内存Inode表中寻找一个空闲的内存Inode
    */
Inode* GetFreeInode();
/*
    * @comment 将所有内存Inode初始化一遍
    */
void Format();
};

```

3.4.2 OpenFileTable 类定义

包括File、OpenFiles、OpenFileTable三部分，控制进程的打开文件结构表。其中File 类记录进程打开文件的读、写请求类型，文件读写位置等信息；OpenFile 类定义了进程打开文件描述符表，记录了当前进程的所有打开文件；OpenFileTable 类负责内核中对打开文件机构的管理，为进程打开文件建立内核数据结构之间的勾连关系。勾连关系指进程 u 区中打开文件描述符指向打开文件表中的 File 打开文件控制结构，以及从 File 结构指向文件对应的内存 Inode。

```

class File{
public:
    /* Enumerate */
    enum FileFlags{
        FREAD = 0x1,          /* 读请求类型 */
        FWRITE = 0x2          /* 写请求类型 */
    };

    unsigned int f_flag;      /* 对打开文件的读、写操作要求 */
    int f_count;              /* 当前引用该文件控制块的进程数量 */
    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 */

public:
    File();

```

```
    ~File();  
};
```

```
class OpenFiles{  
public:  
    static const int NOFILES = 100;    /* 进程允许打开的最大文件数 */  
  
private:  
    File* ProcessOpenFileTable[NOFILES];    /* File对象的指针数组，指向系统打开文件  
表中的File对象 */  
  
public:  
    OpenFiles();  
    ~OpenFiles();  
    /*  
    * @comment 进程请求打开文件时，在打开文件描述符表中分配一个空闲表项  
    */  
    int AllocFreeSlot();  
    /*  
    * @comment 根据用户系统调用提供的文件描述符参数fd，  
    * 找到对应的打开文件控制块File结构  
    */  
    File* GetF(int fd);  
    /*  
    * @comment 为已分配到的空闲描述符fd和已分配的打开文件表中  
    * 空闲File对象建立勾连关系  
    */  
    void SetF(int fd, File* pFile);  
};
```

```
class OpenFileTable{  
public:  
    /* static consts */  
    static const int NFILE = 100; /* 打开文件控制块File结构的数量 */  
  
    File m_File[NFILE];    /* 系统打开文件表，为所有进程共享，进程打开文件描  
述符表  
                                中包含指向打开文件表中对应File结构的指针。*/  
  
public:  
    OpenFileTable();  
    ~OpenFileTable();  
  
    /*
```

```

    * @comment 在系统打开文件表中分配一个空闲的File结构
    */
File* FAlloc();
/*
    * @comment 对打开文件控制块File结构的引用计数f_count减1,
    * 若引用计数f_count为0, 则释放File结构。
    */
void CloseF(File* pFile);
/*
    * @comment 初始化打开文件表
    */
void Format();
};

```

3.4.3 文件系统资源模块定义

FileSystem 类负责管理文件存储设备中的各类存储资源, 以及磁盘块、外存 Inode 的分配、释放。该部分一些常量数据有所更改, 因为本系统的磁盘结构不存在引导区, 故超级块区直接设置从#0 盘块开始, SuperBlock 占磁盘上的#0、#1 扇区, Inode 区占#2~#1023 号扇区, 数据区占#1024~#16383。另外本系统去除了文件系统装配块类 Mount。因为只有磁盘一个设备, 故不需要挂载子文件系统。

```

class FileSystem{
public:
    /* static consts */
    static const int SUPER_BLOCK_SECTOR_NUMBER = 0; /* 定义SuperBlock位于磁盘上的扇区号, 占据200, 201两个扇区。 */
    static const int DISK_SIZE = 16384;
    static const int ROOTINO = 0; /* 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区位于磁盘上的起始扇区号 */
    static const int INODE_ZONE_SIZE = 1024 - 2; /* 磁盘上外存Inode区占据的扇区数量 */

    static const int DATA_ZONE_START_SECTOR = 1024; /* 数据区的起始扇区号 */
    static const int DATA_ZONE_END_SECTOR = 16383; /* 数据区的结束扇区号 */
    static const int DATA_ZONE_SIZE = DISK_SIZE - DATA_ZONE_START_SECTOR; /* 数据区占据的扇区数量 */

private:
    BufferManager* m_BufferManager; /* FileSystem类需要缓存管理模块(BufferManager)提供的接口 */
    SuperBlock* m_SuperBlock;

```

```

    DiskManager* m_DeviceManager;

public:
    FileSystem();
    ~FileSystem();

    /*
     * @comment 系统初始化时读入SuperBlock
     */
    void LoadSuperBlock();
    /*
     * @comment 将SuperBlock对象的内存副本更新到
     * 存储设备的SuperBlock中去
     */
    void Update();
    /*
     * @comment 在存储设备dev上分配一个空闲
     * 外存INode，一般用于创建新的文件。
     */
    Inode* IAlloc();
    /*
     * @comment 释放存储设备dev上编号为number
     * 的外存INode，一般用于删除文件。
     */
    void IFree(int number);
    /*
     * @comment 在存储设备dev上分配空闲磁盘块
     */
    Buf* Alloc();
    /*
     * @comment 释放存储设备dev上编号为blkno的磁盘块
     */
    void Free(int blkno);
    /*
     * @comment 重新构建SuperBlock
     */
    void FormatSuperBlock();
    /*
     * @comment 重新构建整个文件系统
     */
    void FormatFileSystem();
};

```

3.5 文件管理总接口设计

文件管理类 (FileManager) 封装了文件系统的各种系统调用在核心态下处理过程，如对文件的 Open()、Close()、Read()、Write() 等对文件系统访问的具体细节，这样外层可以更轻松便捷地调用。此处函数大多没有形参是因为这里采用的是通过用户类中的 u_arg[5] 数组传入、u_arg0 沟通数据。

不过由于 u_arg 是 int 型数组，传递指针类型时经过提升截断的过程可能导致地址值发生变化，因此这里 Read()、Write()、RdWr() 函数传入了 unsigned char* buffer 形参作为读入读出的内容的首地址。

```
class FileManager{
public:
    /* 目录搜索模式，用于NameI()函数 */
    enum DirectorySearchMode{
        OPEN = 0,          /* 以打开文件方式搜索目录 */
        CREATE = 1,        /* 以新建文件方式搜索目录 */
        DELETE = 2         /* 以删除文件方式搜索目录 */
    };

    /* 根目录内存Inode */
    Inode* rootDirInode;

    /* 对全局对象g_FileSystem的引用，该对象负责管理文件系统存储资源 */
    FileSystem* m_FileSystem;

    /* 对全局对象g_InodeTable的引用，该对象负责内存Inode表的管理 */
    InodeTable* m_InodeTable;

    /* 对全局对象g_OpenFileTable的引用，该对象负责打开文件表项的管理 */
    OpenFileTable* m_OpenFileTable;

public:
    FileManager();
    ~FileManager();

    /*
     * @comment Open() 系统调用处理过程
     */
    void Open();

    /*
     * @comment Creat() 系统调用处理过程
     */
    void Creat();

    /*
     * @comment Open()、Creat() 系统调用的公共部分
     */
    void Open1(Inode* pInode, int mode, int trf);
```

```
/*
 * @comment Close() 系统调用处理过程
 */
void Close();

/*
 * @comment Seek() 系统调用处理过程
 */
void Seek();

/*
 * @comment Read() 系统调用处理过程
 */
void Read(unsigned char* buffer);

/*
 * @comment Write() 系统调用处理过程
 */
void Write(unsigned char* buffer);

/*
 * @comment 读写系统调用公共部分代码
 */
void Rdwr(enum File::FileFlags mode, unsigned char* buffer = nullptr); /*
 * @comment 目录搜索，将路径转化为相应的Inode，
 * 返回上锁后的Inode
 */
Inode* NameI(enum DirectorySearchMode mode);

/*
 * @comment 被Creat() 系统调用使用，用于为创建新文件分配内核资源
 */
Inode* MakNode(unsigned int mode);

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

/*
 * @comment 改变当前工作目录
 */
void ChDir();

/*
 * @comment 删除文件
 */
void UnLink();

/*
 * @comment 列出当前所有文件
 */
void Ls();
```

```
};
```

3.6 用户模块设计

由于只有一个用户、一个进程，本次用户类中删除了很多进程管理、用户组或权限、用户栈核心栈指针、时间等等的相关成员和函数，此外该类中还定义了 `u_arg` 数组暂存参数，方便其他函数使用时不用大量传递形参；系统进行读写时都是读取用户类中的 `u_IOParm` 进行操作；遇到错误也是存放在 `User` 的标志位中，是一个很妙的设计。同时也因为本次实验没有区分进程和运行状态，故没有额外定义 `Kernel` 函数，而是继续发挥 `User` 类的“桥梁作用”，将 `FileManager` 中的接口直接在 `User` 类中进行进一步封装，这样顶层 `Common` 模块可以更加简洁地传递参数，模块与功能之间划分清晰。

```
class User{
public:
    static const int EAX = 0; /* u.u_ar0[EAX]; 访问现场保护区中EAX寄存器的偏移量 */

    /* u_error's Error Code */
    enum ErrorCode{
        myNOERROR = 0, /* No error */
        myEPERM = 1, /* Operation not permitted */
        myENOENT = 2, /* No such file or directory */
        ... ..
        myEPIPE = 32, /* Broken pipe */
        myENOSYS = 100,
    };

    /* 系统调用相关成员 */
    unsigned int u_ar0[1024];

    int u_arg[5]; /* 存放当前系统调用参数 */
    string u_dirp; /* 系统调用参数(一般用于Pathname)的指针 */

    /* 文件系统相关成员 */
    Inode* u_cdir; /* 指向当前目录的Inode指针 */
    Inode* u_pdir; /* 指向父目录的Inode指针 */
    DirectoryEntry u_dent; /* 当前目录的目录项 */
    char u_dbuf[DirectoryEntry::DIRSIZ]; /* 当前路径分量，长度应当为
DirectoryEntry::DIRSIZ */
    string u_curdir; /* 当前工作目录完整路径 */
    ErrorCode u_error; /* 存放错误码 */
    /* 文件系统相关成员 */
    OpenFiles u_ofiles; /* 进程打开文件描述符表对象 */
    /* 文件I/O操作 */
    IOParmeter u_IOParm; /* 记录读写文件的偏移量，用户目标区域和剩余字节数参数 */
    FileManager* u_FileManager;
    string ls;
```

```

public:
    User();
    ~User();

    void Cd(string dirName);
    void Mkdir(string dirName);
    void Create(string fileName, string mode);
    void Delete(string fileName);
    void Open(string fileName, string mode);
    void Close(string fd);
    void Seek(string fd, string offset, string origin);
    void Ls();
    void Update();
    void Read(string sfd, string size);
    void Write(string sfd, string inputStream, string size);
    void FileIn(string exName, string inName, string size);
    void FileOut(string inName, string exName, string size);
    bool IsError();
    void EchoError(enum ErrorCode err);
    int INodeMode(string mode);
    int FileMode(string mode);
    bool checkPathName(string path);
};

```

除了对 fileManager 中的系统调用进行封装提供给 CmdLine(), 本类中还对用户操作中的错误进行读取并分别展示, 设计函数如下。

```

bool User::IsError() {
    if (u_error != myNOERROR) {
        //cout << "errno = " << u_error;
        EchoError(u_error);
        u_error = myNOERROR;
        return true;
    }
    return false;
}

void User::EchoError(enum ErrorCode err) {
    string estr;
    switch (err) {
        case User::myNOERROR:
            estr = " No u_error ";
            break;
        case User::myENOENT:

```

```

        estr = " No such file or directory ";
        break;
    case User::myEBADF:
        estr = " Bad file number ";
        break;
    case User::myEACCES:
        estr = " Permission denied ";
        break;
    case User::myENOTDIR:
        estr = " Not a directory ";
        break;
    case User::myENFILE:
        estr = " File table overflow ";
        break;
    case User::myEMFILE:
        estr = " Too many open files ";
        break;
    case User::myEISDIR:
        estr = " Try to open a directory ";
        break;
    case User::myEFBIG:
        estr = " File too large ";
        break;
    case User::myENOSPC:
        estr = " No space left on device ";
        break;
    default:
        break;
}
cout << estr << endl;
}

```

3.7 顶层设计

3.7.1 API 封装

FileManagerer 封装好文件系统接口后，User 类中编写的 API 的实现就简单许多，只需要将对应参数传入并调用相应接口即可，这些函数实现在 User.h 中，介绍如下：

(1) Fcreat

函数原型为 `int Fcreat(char *name, int mode)`；作用为创建文件。将 `name` 传给 User 结构的 `u_dirp`，`mode` (-rw 等分别对应)传给 `u_arg[1]`，接着调用 `FileManagerer::Creat()`，返回 `u_ar0[EAX]`，即创建文件的 fd。

(2) Fopen

函数原型为 `int Fopen(char *name, int mode);` 作用为打开文件。将 `name` 传给 `User` 结构的 `u_dirp`, `mode` 传给 `u_arg[1]`, 接着调用 `FileManager::Open()`, 返回 `u_ar0`, 即创建文件的 `fd`。

(3) Fread

函数原型为 `int Fread(int fd, char *buffer, int length);` 作用为从文件中读取数据。将 `fd` 传给 `u_arg[0]`, `buffer` 转为 `int` 型传给 `u_arg[1]`, `length` 传给 `u_arg[2]`, 接着调用 `FileManager::Read()`, 返回 `u_ar0`, 即读取到的字节数。

(4) Fwrite

函数原型为 `int Fwrite(int fd, char *buffer, int length);` 作用为向文件中写入数据。将 `fd` 传给 `u_arg[0]`, `buffer` 转为 `int` 型传给 `u_arg[1]`, `length` 传给 `u_arg[2]`, 接着调用 `FileManager::Write()`, 返回 `u_ar0`, 即写入的字节数。

(5) Fdelete

函数原型为 `void Fdelete(char *name);` 作用为删除文件。将 `name` 传给 `u_dirp`, 调用 `FileManager::Unlink()`。

(6) Flseek

函数原型为 `int Flseek(int fd, int position, int whence);` 作用为重定位文件当前读写指针。将 `fd` 传给 `u_arg[0]`, `position` 传给 `u_arg[1]`, `whence` 传给 `u_arg[2]`, 接着调用 `FileManager::Seek()`, 返回 `u_ar0`。

(7) Fclose

函数原型为 `void Fclose(int fd);` 作用为关闭已打开的文件。将 `fd` 传给 `u_arg[0]`, 接着调用 `FileManager::Close()`。

(8) Ls

函数原型为 `void Ls();` 作用为显示当前目录。该 API 在 `FileManager` 中没有对应接口, 但可以用其他 API 的组合实现。首先使用 `Fopen` 打开当前目录文件, 接着使用 `Fread` 每次 32 个字节的读取文件, 输出后 28 个字节即为该目录项的文件名, 最后调用 `Fclose` 关闭打开的目录文件。

(9) Mkdir

函数原型为 `void Mkdir(char *name);` 作用为创建目录文件。将 `name` 传给 `u_dirp`, 040755(默认模式)传给 `u_arg[1]`, 调用 `FileManager::MkNod()`。

(10) Cd

函数原型为 `void Cd(char *name);` 作用为改变工作目录。将 `name` 传给 `u_dirp`, 并转为 `int` 型传给 `u_arg[0]`, 调用 `FileManager::ChDir()`。

(11) Fin

函数原型为 `void Fin(string exName, string inName);` 作用为将外部文件传入内部文件(附带创建)。基于几个 API 的组合实现: 首先调用 `User::creat` 创建名 `inName` 的文件, 接着从 `exName` 读取文件内容到 `string` 字符串中, 同时也可以获取文件总字节数, 然后调用 `Fwrite` 向内部文件写入读取到的内容, 最后要记得 `Fclose` 关闭文件, 因为这里的 `Fopen` 是内部调用而不是用户执行的, 因此也要在内部“析构”掉。

(12) Fout

函数原型为 `void Fout(string intername, string extername);`作用为将内部文件传到外部文件。首先调用 `Fopen` 打开名为 `inName` 的文件，接着从打开文件表中获取表项，从而寻找到文件 `Inode`，获取文件字节数，然后再调用 `Read` 从 `inName` 文件中读入这些字节长度，最后读取到地址写入外部文件 `exName` 中，最后调用 `Fclose` 关闭文件。

(13) Help

函数原型为 `void Help();`作用为输出帮助信息。

3.7.2 Common 类设计

通过命令行与用户交互，此处只展示头文件。为了保证磁盘、用户、文件系统等等对象的唯一性，这里将 `DiskManager`、`User`、`FileSystem` 等类都在 `Common.h` 中声明全局变量，并在 `main.cpp` 中进行统一实例化。

```
#include "User.h"
#include "Buf.h"
#include "BufferManager.h"
#include "FileSystem.h"
#include "InodeTable.h"
#include "FileManager.h"
#include "OpenFileTable.h"
#include "DiskManager.h"
#include <vector>
using namespace std;

extern bool DEBUG;

extern User globalUser;
extern DiskManager globalDiskManager;
extern BufferManager globalBufferManager;
extern OpenFileTable globalOpenFileTable;
extern SuperBlock globalSuperBlock;
extern FileSystem globalFileSystem;
extern InodeTable globalInodeTable;
extern FileManager globalFileManager;

// 处理输入的命令行
vector<string> getCmd(string str);

// 进行对应的运算
void exeCmd(vector<string> cmdList);

//展示用法帮助
void Usage();
```

3.7.3 系统初始化流程

完成 Buffer、Disk、FileSystem 和 User 的初始化，加载超级块，并创建根目录结构。由于未创建 Kernel 类，该部分也在 User 类内实现，并在 User 的构造函数中调用。Buffer、Disk、FileSystem、User 源代码中的四大件均在相关类的构造函数中实现。

在 3.7.1 中 Common.h 定义全局外部变量，在 main.cpp 中被统一初始化，各类的构造函数展示如下，也都是系统初始化流程的一部分。

```
BufferManager::BufferManager() {
    bFreeList = new Buf;
    for (int i = 0; i < NBUF; ++i) {
        if (i) {
            m_Buf[i].b_forw = m_Buf + i - 1;
        }
        else {
            m_Buf[i].b_forw = bFreeList;
            bFreeList->b_back = m_Buf + i;
        }

        if (i + 1 < NBUF) {
            m_Buf[i].b_back = m_Buf + i + 1;
        }
        else {
            m_Buf[i].b_back = bFreeList;
            bFreeList->b_forw = m_Buf + i;
        }
        m_Buf[i].b_addr = Buffer[i];
    }
    m_DeviceManager = &globalDiskManager;
}
```

```
DiskManager::DiskManager() {
    fp = fopen(DISK_FILE_NAME, "rb+");
}

DiskManager::~DiskManager() {
    if (fp) {
        fclose(fp);
    }
}
```

```
FileManager::FileManager() {
    m_FileSystem = &globalFileSystem;
    m_OpenFileTable = &globalOpenFileTable;
    m_InodeTable = &globalInodeTable;
    rootDirInode = m_InodeTable->IGet(0);
}
```



```
    rootDirInode->i_count += 0xff;
}
```

```
InodeTable::InodeTable() {
    m_FileSystem = &globalFileSystem;
}
```

```
FileSystem::FileSystem() {
    m_DeviceManager = &globalDiskManager;
    m_SuperBlock = &globalSuperBlock;
    m_BufferManager = &globalBufferManager;

    if (!m_DeviceManager->Exists())
        FormatFileSystem();
    else
        LoadSuperBlock();
}
FileSystem::~FileSystem() {
    Update();
    m_DeviceManager = NULL;
    m_SuperBlock = NULL;
}
```

4 代码测试

4.1 实验环境

宿主机：Windows 10, 64bit

编译器：VS2022, Debug x64 模式

4.2 功能测试

实验要求中 link, unlink, 则使用 fcreat、rm 指令的形式展示。在测试环境中编译打开命令行窗口，依次测试如下（当前工作目录下未创建 FS.img 时会自动创建并初始化）。

help 指令获取使用帮助。

❏ D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe

```
[u2152189 /]$ help

ls                list directorys and files in current path
mkdir <path>      create a directory
cd <path>          change current path
fcreat <path>     create a file or directory
mode: -rw         read and write
      -r          only read
      -w          only write
rm <path>         delete a file or directory
fopen <path> [mode] open a file
mode: -rw         read and write
      -r          only read
      -w          only write
fclose <fd>       close a file
fseek <fd> <pos> [mode] set the read/write pointer of file(fd={fd}) at {pos}
mode: 0           [default] pos is the offset from beginning
      1           pos is the offset from current position
      2           pos is the offset from end
fread <fd> <count> read file(fd={fd}) for {count} byte
fwrite <fd> <content> <count> write file(fd={fd}) for {count} byte from {content}
fin <ex_path> <in_path> <count> import file from {ex_path} for {count} byte to {in_path}
fout <in_path> <ex_path> <count> export file(fd={fd}) for {count} byte as {path}
help             more detailed command info
shutdown         save your changes and shutdown the shell
exit            format the disk and exit the system

[u2152189 /]$ _
```

ls 查看当前目录下所有内容；mkdir 创建文件目录。Shutdown 表示仅关闭窗口，关闭前将所有内存 Inode 改动写回磁盘，第二次打开时发现创建的目录仍然存在，随后使用 exit 指令退出，第三次打开时发现第一次创建的目录已经消失。

❏ Microsoft Visual Studio 调试控制台

```
[u2152189 /]$ ls
[u2152189 /]$ mkdir bin
[u2152189 /]$ mkdir etc
[u2152189 /]$ mkdir home
[u2152189 /]$ mkdir dev
[u2152189 /]$ ls
bin etc home dev
[u2152189 /]$ shutdown
All Changes Saved, Exiting

D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe (进程 16856)已退出，代码为 0。
按任意键关闭此窗口. . .
```

```
Microsoft Visual Studio 调试控制台
[u2152189 /]$ ls
bin etc home dev
[u2152189 /]$ exit
Exiting... This will Lose All Your Data.

D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe (进程 17172)已退出, 代码为 0。
按任意键关闭此窗口. . .
```

```
D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe
[u2152189 /]$ ls
[u2152189 /]$
```

使用 `fcreat` 命令可以创建新文件并赋予读写权限。`cd` 可以进入已有目录，若不是目录文件会弹出错误并给出提示。下图中在 `/` 目录下 `test` 不是目录、`/bin` 目录下没有文件夹，分别给出不同的错误提示信息。

```
D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe
[u2152189 /]$ fcreat test -rw
[u2152189 /]$ mkdir bin
[u2152189 /]$ ls
test bin
[u2152189 /]$ cd test
errno = 20 Not a directory
[u2152189 /]$ cd bin
[u2152189 /bin/]$ ls

[u2152189 /bin/]$ cd bin
errno = 2 No such file or directory
[u2152189 /bin/]$
```

创建一个文件后无法 `cd`、创建一个目录后无法 `fopen`、参数数量错误均会做出提示，抛出错误后的具体展示逻辑编写在 `User.cpp` 中，相关代码已在 3.6 中进行展示。

```
C:\Users\86187\Documents\GitHub\Secondary-File-System\x64\Debug\SecondaryFS.exe
[u2152189 /]$ mkdir dir
[u2152189 /]$ fopen dir -rw
Try to open a directory
打开文件dir失败
[u2152189 /]$ fcreat file -rw
[u2152189 /]$ cd file
Not a directory
[u2152189 /]$ ls
test dir file
[u2152189 /]$ cd dir
[u2152189 /dir/]$ cd ..
[u2152189 /]$ fopen file -rw
打开文件file成功! 句柄fd=2
[u2152189 /]$
```

```
C:\Users\86187\Documents\GitHub\Secondary-File-System\x64\Debug\SecondaryFS.exe
[u2152189 /]$ fopen file
Command - fopen requires 2 args!
Usage:
ls                list directorys and files in current path
mkdir <path>      create a directory
cd <path>         change current path
fcreat <path> [mode] create a file or directory
mode: -rw        read and write
      -r         only read
      -w         only write
rm <path>         delete a file or directory
fopen <path> [mode] open a file
mode: -rw        read and write
      -r         only read
      -w         only write
fclose <fd>       close a file
fseek <fd> <pos> [mode] set the read/write pointer of file(fd={fd}) at {pos}
mode: 0          pos is the offset from beginning
      1          pos is the offset from current position
      2          pos is the offset from end
fread <fd> <count> read file (fd={fd}) for {count} byte
fwrite <fd> <content> <count> write file (fd={fd}) for {count} byte from {content}
fin <ex_path> <in_path> import external file from {ex_path} into secondary filesystem {in_path}
fout <in_path> <ex_path> export secondary filesystem file (fd={fd}) to external file {path}
help             detailed command info
shutdown        save your changes and shutdown the cmd shell
exit            format the disk and exit
[u2152189 /]$
```

使用 fwrite 命令可以通过特定句柄向文件中写入特定长度的特定字符串，使用 fread 命令可通过特定句柄从文件中读出特定长度的字符串。在 fwrite 命令写入完成后文件读写指针位于文件末尾，此时立即 fread 会读出乱码。重新打开之后打开文件表更新，文件指针指向文件开头，这时 fread 读出正确。

```
D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe
[u2152189 /]$ ls
[u2152189 /]$ fcreat test -rw
[u2152189 /]$ fopen test -rw
打开文件test成功! 句柄fd=1
[u2152189 /]$ fwrite 1 hello,world! 12
向文件fd= 1写入12字节成功!
[u2152189 /]$ fread 1 12
从文件fd= 1成功读取12字节:
屯屯屯屯屯屯
[u2152189 /]$ fclose 1
成功关闭文件fd=1
[u2152189 /]$ fopen test -rw
打开文件test成功! 句柄fd=1
[u2152189 /]$ fread 1 12
从文件fd= 1成功读取12字节:
hello,world!
[u2152189 /]$
```

此时执行 shutdown 命令后窗口会关闭，重新进入并 fread 读入发现文件内容还在。

```
D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe
[u2152189 /]$ ls
test
[u2152189 /]$ fopen test -rw
打开文件test成功! 句柄fd=0
[u2152189 /]$ fread 0 12
从文件fd= 0成功读取12字节:
hello,world!
[u2152189 /]$
```

在上一次结果上继续检查 `lseek` 效果。第一个参数是句柄 `fd`，第二个参数是偏移数据，第三个参数是 `mode`，0 表示从文件开头开始、1 表示从当前指针位置开始、2 表示从文件末尾开始。`fread 0 12` 后指针在 12 位置、`lseek 0 0 0` 将句柄 0 的文件指针移动到文件开始、`fread 0 8` 后指针在 8 位置、`lseek 0 2 1` 将指针移动到 10 位置，最后 `fread 0 2` 将读取写入的最后两个字节，符合预期。

```
D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe

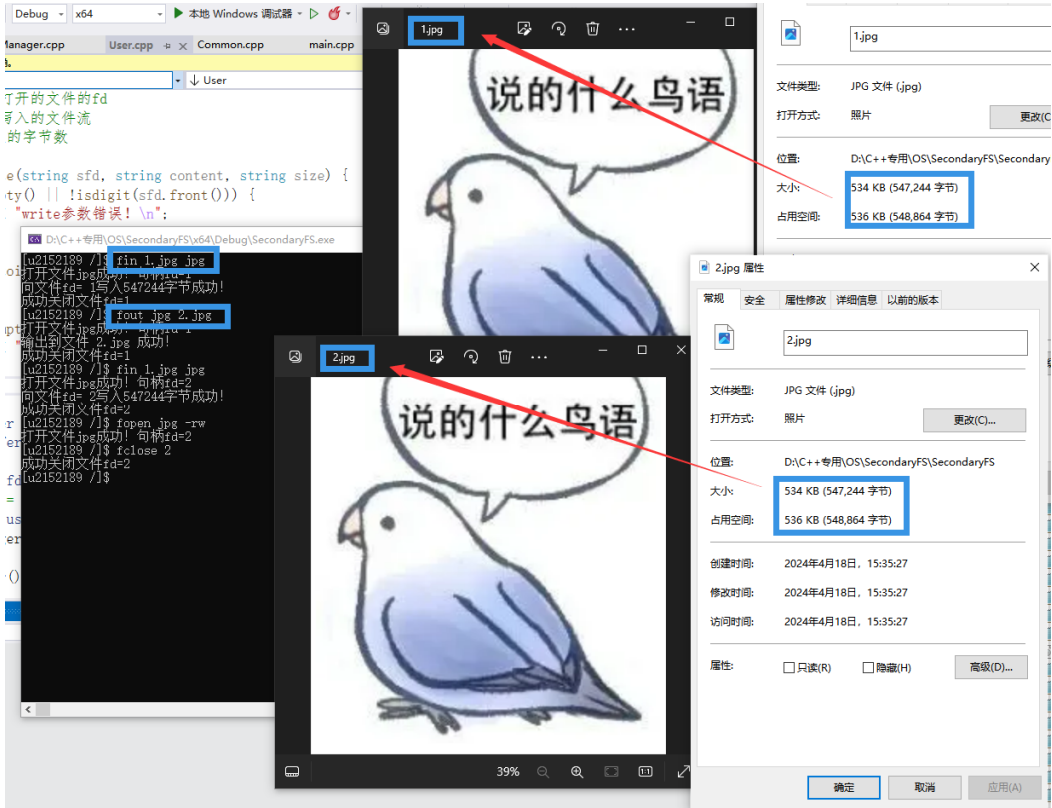
[u2152189 /]$ ls
test
[u2152189 /]$ fopen test -rw
打开文件test成功! 句柄id=0
[u2152189 /]$ fread 0 12
从文件fd= 0成功读取12字节:
hello,world!
[u2152189 /]$ lseek 0 0 0
[u2152189 /]$ fread 0 8
从文件fd= 0成功读取8字节:
hello,wo
[u2152189 /]$ lseek 0 2 1
[u2152189 /]$ fread 0 2
从文件fd= 0成功读取2字节:
d!
```

在以上的基础上继续测试 `fclose` 命令和 `rm` 命令。

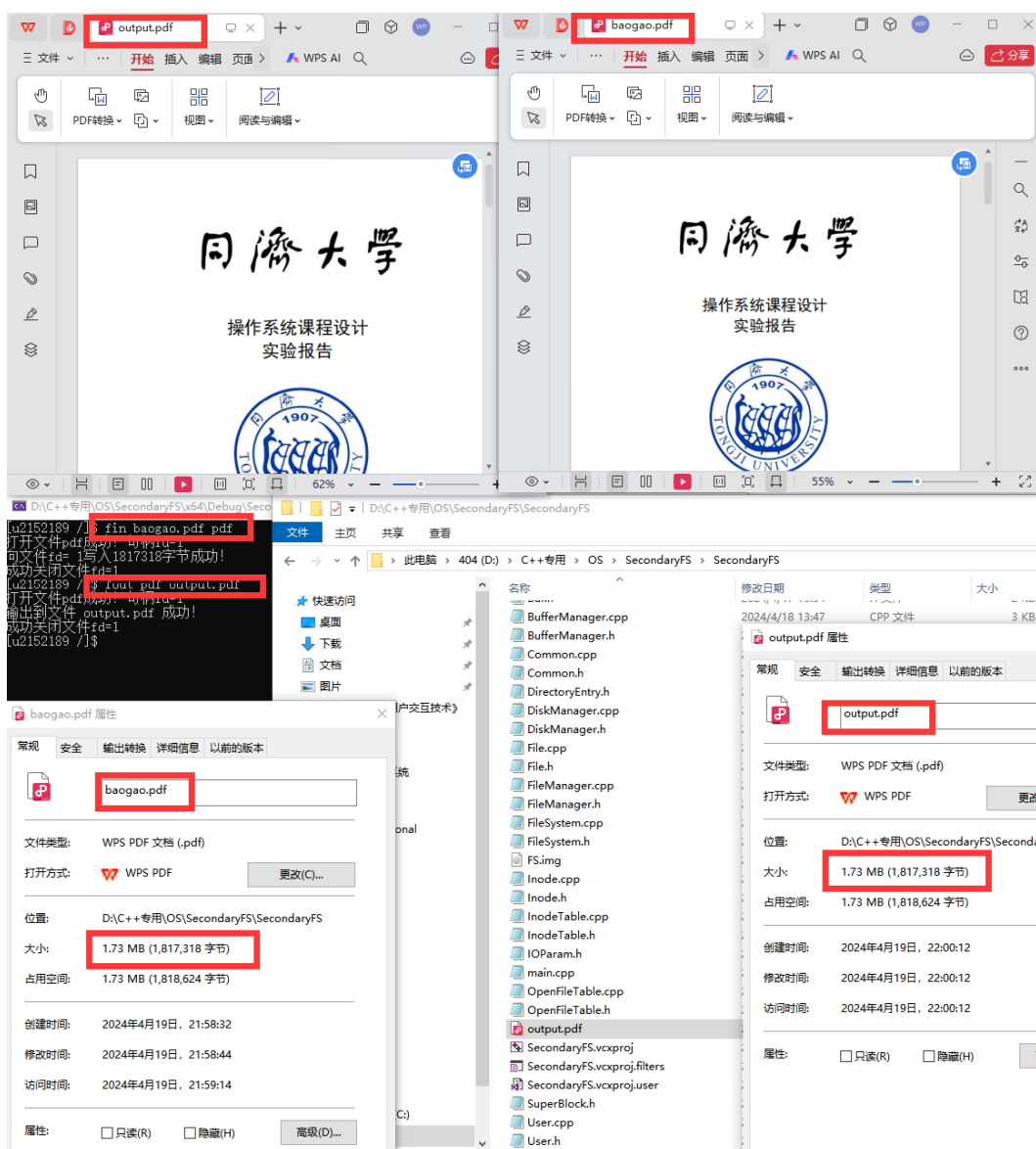
```
D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe

[u2152189 /]$ fread 0 2
从文件fd= 0成功读取2字节:
d!
[u2152189 /]$ fclose 0
成功关闭文件fd=0
[u2152189 /]$ ls
test
[u2152189 /]$ rm test
[u2152189 /]$ ls
[u2152189 /]$
```

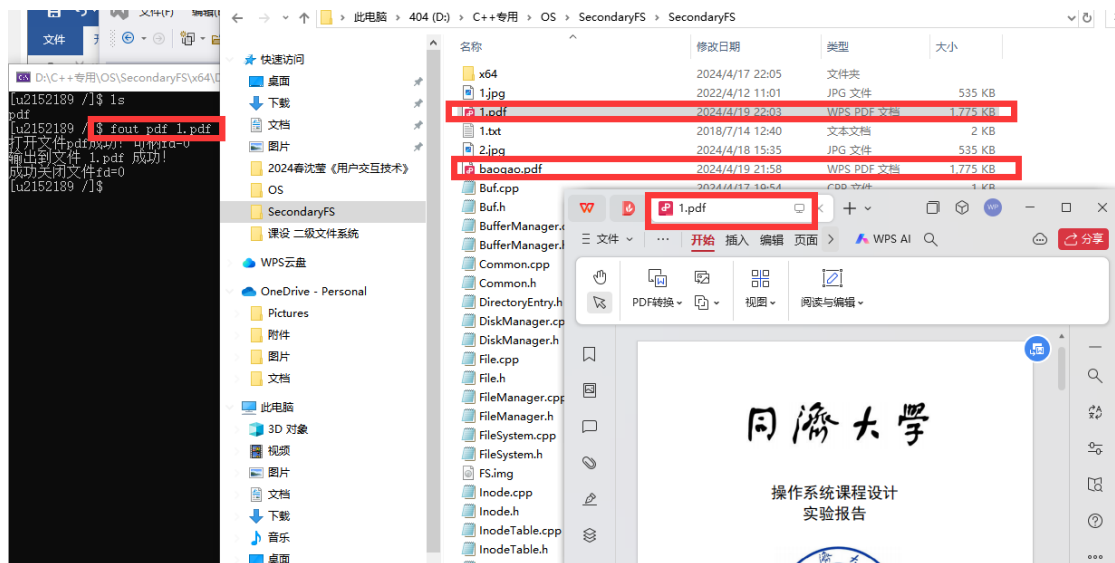
继续测试 `jpg` 格式图片文件读入 1. `jpg` 格式，输出 2. `jpg`，字节数与文件打开均正常



将本 PDF 导入后再导出也能够正常打开，字节数不变。



并且 shutdown 关闭磁盘后重新编译打开命令行，仍存在且仍然可以正常导出。



5 调试时遇到的问题

5.1 初始化的先后顺序

初始化也是遇到的最大问题，虽然 Initialize 和 format 函数在一周前就已经写好，但当时只是想着找到特定扇区和 DiskInode，赋个值就行，但整个模块运行时才发现有很多因素没有考虑到，比如：

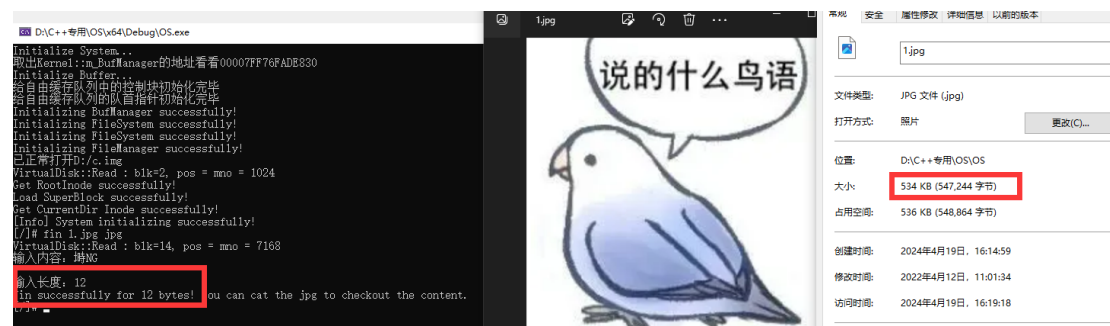
- 未初始化 SuperBlock 导致默认 Inode 为 0，一直提示磁盘已满没搜索到可用外存
- 未初始化 Buf 中的标志位，导致第一次读取 0 时，存在缓存队列（初始化就是 0），

但是 flag&DONE 不是 1，于是返回值为 nullptr 报错等等

最终的决定是：各类不单独构建 Initialize 函数，初始化过程写在构造函数中，而在进程关闭时的善后过程则在析构函数中实现，如执行“shutdown”命令时应当把缓存模块中有修改的 Inode 和 SuperBlock 写回磁盘，将其封装为 Update() 函数写在 FileSystem 的析构函数中。

其他还存在比如延迟写回标记等细节，好在本次实验是在 Unix V6++ 可运行源码上裁剪，很多细节和错误码设计已经处理好了。

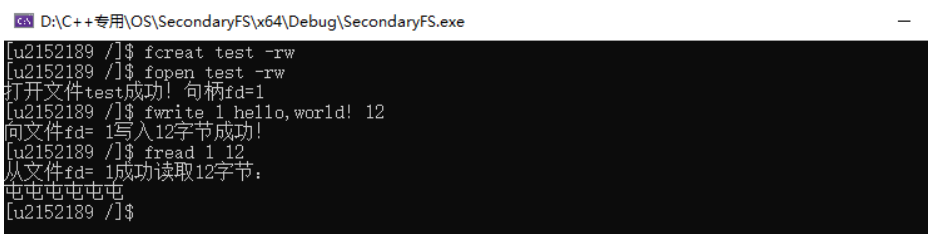
5.2 文本入出正常但图片读入出时被提前截断



如上图所示，图片大小为 547244 字节，但最终只读入了 12 字节，最后通过逐步调试发现问题出在 C++ 库读写文件上，默认打开方式下读入遇到 0 号字符自动终止（通过 Ultra Edit 文件查看 1.jpg 文件发现第 13 字节为 00，确认猜想）。最终发现使用 fopen 时以二进制格式打开即可，这样的话读至文档末尾 EOF 才退出。

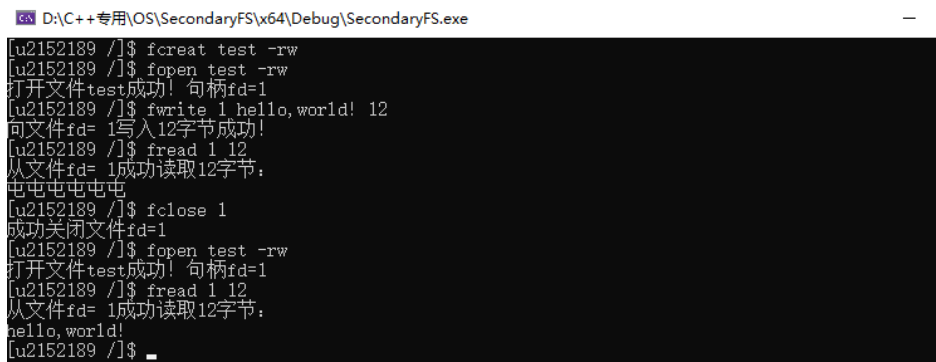
调试时还发现二进制模式打开时，Windows 系统中换行符将占用 \n\r2 个字节。

5.3 文件指针及 offset 位置的理解



如上图所示，写入文件后读取文件，发现输出为空。调试发现是因为写入后的文件指针指向了文件末尾，这时若继续读取自然无法读入内容，检查 Unix V6++ 源码发现也是这样，而且文件指针是写在进程的打开文件表表项中的，于是我就手动修改调用 ReadI 前赋值文件指针在 0 处。

这样“看似”读取问题解决了，但后来加入 seek 指针移动函数后我才意识到真正的问题：这样的话读取文件只能从头读取，这才意识到其实最开始遇到的并不是问题，关闭文件重新打开、或者设置文件指针从文件起始地址偏移 0 即可，如下图。



```
D:\C++专用\OS\SecondaryFS\x64\Debug\SecondaryFS.exe
[u2152189 /]$ fcreat test -rw
[u2152189 /]$ fopen test -rw
打开文件test成功! 句柄fd=1
[u2152189 /]$ fwrite 1 hello,world! 12
向文件fd= 1写入12字节成功!
[u2152189 /]$ fread 1 12
从文件fd= 1成功读取12字节:
hello,world!
[u2152189 /]$ fclose 1
成功关闭文件fd=1
[u2152189 /]$ fopen test -rw
打开文件test成功! 句柄fd=1
[u2152189 /]$ fread 1 12
从文件fd= 1成功读取12字节:
hello,world!
[u2152189 /]$
```

除了 fread 和 fwrite，最后测试 fin 和 fout 时也遇到了正常写入字节，但写出文件输出为 0 的问题，内耗了很长时间才意识到：这难道不是和上面的原因一样吗？于是恍然大悟，同时也意识到自己没有调用 fclose（fin 和 fout 中都同时内置 fopen 和 fclose 的话便会自动刷新文件指针位置了）。这段调试中自己给自己造成的乌龙也加深了我对文件指针的存储和赋值过程的认知。

6 实验总结

6.1 实验流程

本次课程设计我从读写模块开始，首先根据老师提供的参考报告建立 File、Disk Manager、Buf、Inode 等基础数据结构类，之后开始从 ReadI/WriteI->Bread->GetBlk->DiskManager::Read/Write，然后加入缓存和 Inode 管理模块，并编写 Bmap 实现多级混合索引目录的形式存储文件。之后开始 DiskInode/SuperBlock 类的实现，于是建立起自由缓存队列的实现、内存 Inode 表 InodeTable 和进程的打开文件表 OpenFileTable，最后封装成 FileSystem 文件系统模块，然后关联之前所有的函数封装给出 FileManager 接口，对应的是 Unix V6++代码中的各类系统调用，至此内部系统搭建完毕。再外部新建一个 Common.cpp，读取用户的输入做命令行交互。

6.2 实验感受

首先感受是邓蓉老师理论课上，听懂是一个阶段，之后手写各种流程的复盘是一个阶段，这次课设中代码实现又是一个阶段。写代码要考量的东西太多了，比如 Inode 是否已经在内存中，不在内存中搬运过来要分配，何种情况要分配内存块，内存块与自由缓存队列的 IO，编写函数实现写回等等细枝末节的东西。同时也发现文件自身有物理结构，也就是一个文件的内容是怎样通过混合目录索引被划分为 512B 的块的，还有最后想要读文件时磁盘是如何通过 Bmap() 函数查到全部的这些块并正确地“组装”回一个新的文件，这些在做题和看知识点时都知道如此，但具体想要上手实现就犯了难

这次实验使我对文件系统的生命周期也有了更深的理解。文件中有各类标志位，如标识该 Inode 指向一个目录文件，应当在哪里定义，应当如何从磁盘读取进来。实例化的初始化时也有着先后关系，还要注意进入环境时哪些数据不能被初始化、退出环境时哪些数据需要保留下来、下次再进入时正确如何读取恢复，都是非常令人头大的问题。

写代码时也加深了对细节的理解。如编写 `cd` 命令时突然想到 Unix 如何区分目录和文件的 `Inode`, 使得目录可以进入而文件不可进入。最后发现 `i_mode` 参数可以通过 `0x1`、`2`、`4` 可以字节按位与检测是否存在。比如系统调用时统一传入 `user` 结构也是一种很新颖的方式, 但有一个问题是 `u_arg` 是 `int` 型数组, 如果将 `buffer` 地址赋值给 `int` 变量, 再转回来的提升和截断过程中很可能经历一些隐藏的 bug, 因此我修改了一些函数使其直接以形参的形式传送。

在阅读裁剪代码的过程中, 我也学习到了很多 C++ 知识, 如通过 `static`、`extern` 定义全局唯一变量, 或者通过 `static` 成员函数和引用获取; 还有对于多个标志位的情况, 赋值 `0x1`、`0x2`、`0x4`, 可以对字节按位与 `&` 来检测某个或某些标志位是否存在。