同济大学

实验报告

装

订

线

同济大学计算机系
操作系统课程设计
用户二级文件系统设计报告

班级：计算机一班

姓名：王哲源

学号：1652228

指导老师：邓蓉

完成日期：2019.5.3

目录

装 订 线

一. 实验目的

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

二. 实验内容

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。

裁剪 Unix V6++内核，用以管理二级文件系统。

三. 实验基础任务要求及任务描述

3.1 基础目标

设计并实现一个类 Unix V6++的二级文件系统。该系统基于某大文件（本次使用 myDisk.disk）作为文件系统的磁盘阵列，仿照 Unix V6++的二级文件系统，将文件进行分块划分，构成若干个磁盘块，每块大小定为 512 字节，在其中存储文件系统内的所有东西。

3.2 该实验要求能够支持以下几种操作：

void ls(); 列目录

Int fopen(char *name, int mode);

Void fclose(int fd);

Int fread(int fd, char *buffer, int length);

Int fwrite(int fd, char *buffer, int length);

Int flseek(int fd, int position);

Int fcreat(char *name, int mode);

Int fdelete(char *name)；

3.3 同时实现以下功能：

将宿主机中的文件存入虚拟磁盘

将虚拟磁盘中存放的文件取出，保存在宿主机文件系统中。

要求.txt 能够被记事本打开，可执行程序能够跑起来。

创建新目录，删除已经存在的目录。

3.4 附加要求：

实现文件系统缓存，减少文件系统对磁盘的访问次数

四. 设计概述

4.1 总体设计

设计总共分为以下三块：

文件系统的基础设计

高速缓存的实现

API 功能的实现已经测试 UI 设计

API 功能由文件系统实现的各类操作进行拼接组合，高速缓存则在文件系统可以运行的基础上再作追加，因此本次设计的主要工作来源于文件系统的设计

文件系统采用 Unix V6++源码进行裁切，但不同于完整的 Unix V6++操作系统，本次的课程设计存在以下几种情况：

不存在多用户系统。Linux 操作系统支持多用户操作，但由于本次文件系统课设为单一的二级文件系统，因此不存在复杂的多用户操作，并且对于单一用户必然为超级用户，因此省去了对用户权限判断的工作量

不存在多线程操作。该情况同 1，一并省去了对于文件锁以及进程休眠情况的判断，相应的内容可以被略去

单一设备。单一的文件系统不存在外设等各类情况，且由于本次课设内仅存在块文件，因此对于各类设备号、挂载等操作可以一并忽视，同时对于高速缓存，即 BufferManager 可直接交由文件系统一并管理，单一的设备链只需要利用一条双向循环链表进行实现，并且每次被访问到的缓存块直接移动至链尾即可

实验验证时的图片大小均为 1~3M 左右，因此本次课程设计所需要的磁盘空间需求量不大，可对于磁盘块数目进行适当削减，提高程序的可调性

文件的读写由于单一文件系统对于效率的要求并不高，并且不存在多线程，文件的读写相对简便，因此这里舍弃了异步写的实现，并且并未实现预读操作

4.2 开发环境

运行平台：CentOS 7 64 位

编译环境：4.8.5 20150623 (Red Hat 4.8.5-36) (GCC)

注：由于源码使用的是 Unix V6++源码裁切而成的，而 Unix V6++本身运行于 32 位操作系统，而本次使用的虚拟机为 64 位操作系统，会导致指针字节数不同等一系列错误，因此本次的编译环境需要额外安装 32 位的编译库（具体参见 readme.md 文件），且在编译时加上-m32 参数采用 32 位编译器编译（具体参见 Makefile 文件）

五. 详细设计

5.1 部分类及功能函数定义

5.1.1 超级块 SuperBlock

```
/*
 * 文件系统存储资源管理块(Super Block)的定义。
 */
class SuperBlock
{
    /* Functions */
public:
    /* Constructors */
    SuperBlock();
    /* Destructors */
    ~SuperBlock();

    /* 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_fmod;         /* 内存中 super block 副本被修改标志，意味着需要更新外存对应的 Super Block */
    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[49];    /* 填充使 SuperBlock 块大小等于 1024 字节，占据 2 个扇区 */
};
```

单用户系统使得本次课程设计删去了用于上锁的 s_flock 及 s_ilock 参数，因此注意对 padding 补全两个字节保证 SuperBlock 类依旧占用 1024 字节

5.1.2 缓存块 Buf

```
class Buf
{
public:
    enum BufFlag    /* b_flags 中标志位 */
    {
        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，清 B_BUSY 标志
时，要唤醒这种进程 */
        B_ASYNC = 0x40,     /* 异步 I/O，不需要等待其结束 */
        B_DELWRI = 0x80     /* 延迟写，在相应缓存要移做他用时，再将其内容写到相应
块设备上 */
    };

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

    int     padding;        /* 4 字节填充，使得 b_forw 和 b_back 在 Buf 类中与 Devtab
类
                            * 中的字段顺序能够一致，否则强制转换会出错。 */
                           /* 缓存控制块队列勾连指针 */
    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 = -1;       /* 磁盘逻辑块号 */
};
```

由于不存在各类设备，因此这里删去 av_forw 及 av_back，将缓存交由缓存控制器单独管理，并且删去了与 IO 错误相关的变量，将其最简化为双向链表上的一块分区

5.1.3 高速缓存管理器 BufferManager

```
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();

    void Initialize(char *start);                    /* 缓存控制块队列的初始化。
将缓存控制块中 b_addr 指向相应缓冲区首地址。*/

    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。dev 为主、次设备号，blkno 为目标磁盘
块逻辑块号。 */
    // Buf* Breada(short adev, int blkno, int rablkno);  先不预读了。。

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

    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
*/

    Buf& GetBFreeList();                /* 获取自由缓存队列控制块 Buf 对象引用 */

private:
    Buf* InCore(int blkno); /* 检查指定字符块是否已在缓存中 */

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 */

    char* p;                            /* 指向整个文件系统用 mmap 映射到内存后的
起始地址 */
};
```

这里的 BufferManager 基本重写了，删去了与设备有关的操作与变量，并且将整个缓存块组交由自己控制而非设备控制器控制

缓存块组由 15 块缓存块互相前后链接形成的双向链表组成，每当

GetBlk 找到对应缓存块或分配新的缓存块时，便会将其挂至双向链表逆向一侧的第一个点保证其不会被优先摘取，即 LRU 的缓存块替换思想。同时此处的异步写 Bawrite 虽然被保留，但并未被使用

5.1.4 文件系统类 FileSystem

```
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1, 2 两个扇区。 */

    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 - 202 */ 1000 - 2;       /* 磁盘上外存Inode 区占据的扇区数 */

    static const int DATA_ZONE_START_SECTOR = /* 1024 */ 1000;      /* 数据区的起始扇区号 */
    static const int DATA_ZONE_END_SECTOR = /* 18000 - 1 */ 10000 - 1;  /* 数据区的结束扇区号 */
    static const int DATA_ZONE_SIZE = 10000 - DATA_ZONE_START_SECTOR;   /* 数据区占据的扇区数量 */

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

    /*
     * @comment 初始化成员变量
     */
    void Initialize();

    /*
    * @comment 系统初始化时读入SuperBlock
```

```
    */
    void LoadSuperBlock();

    /*
     * @comment 根据文件存储设备的设备号dev 获取
     * 该文件系统的SuperBlock
     */
    SuperBlock* GetFS();
    /*
     * @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);

private:
    /*
     * @comment 检查设备dev 上编号blkno 的磁盘块是否属于
     * 数据盘块区
     */
    bool BadBlock(SuperBlock* spb, int blkno);

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

相较于原版的 Unix V6++操作系统，本次大幅减少了磁盘块数目，并且删去了对于设备文件的操作函数，其他相较于原 Unix V6++代码并没有大幅的改动

5.1.5 用户结构 User

```
class User
{
public:
    /* u_error's Error Code */
    /* 1~32 来自linux 的内核代码中的/usr/include/asm/errno.h, 其余for V6++ */
    enum ErrorCode
    {
        my_NOERROR = 0, /* No error */
        my_EPERM = 1,   /* Operation not permitted */
        my_ENOENT = 2,  /* No such file or directory */
        my_ESRCH = 3,   /* No such process */
        my_EINTR = 4,   /* Interrupted system call */
        my_EIO = 5, /* I/O error */
        my_ENXIO = 6,   /* No such device or address */
        my_E2BIG = 7,   /* Arg list too long */
        my_ENOEXEC = 8, /* Exec format error */
        my_EBADF = 9,   /* Bad file number */
        my_ECHILD = 10, /* No child processes */
        my_EAGAIN = 11, /* Try again */
        my_ENOMEM = 12, /* Out of memory */
        my_EACCES = 13, /* Permission denied */
        my_EFAULT = 14, /* Bad address */
        my_ENOTBLK = 15,    /* Block device required */
        my_EBUSY = 16,  /* Device or resource busy */
        my_EEXIST = 17, /* File exists */
        my_EXDEV = 18,  /* Cross-device link */
        my_ENODEV = 19, /* No such device */
        my_ENOTDIR = 20,    /* Not a directory */
        my_EISDIR = 21, /* Is a directory */
        my_EINVAL = 22, /* Invalid argument */
        my_ENFILE = 23, /* File table overflow */
        my_EMFILE = 24, /* Too many open files */
        my_ENOTTY = 25, /* Not a typewriter(terminal) */
        my_ETXTBSY = 26,    /* Text file busy */
        my_EFBIG = 27,  /* File too large */
        my_ENOSPC = 28, /* No space left on device */
```

```
        my_ESPIPE = 29, /* Illegal seek */
        my_EROFS = 30,  /* Read-only file system */
        my_EMLINK = 31, /* Too many links */
        my_EPIPE = 32,  /* Broken pipe */
        my_ENOSYS = 100,
        //my_EFAULT = 106
    };

public:
    /* 系统调用相关成员 */
    int u_ar0;                  /* 原*u_ar0，指向核心栈现场保护区中EAX寄存器
                                存放的栈单元，本字段存放该栈单元的地址。
                                在V6中r0存放系统调用的返回值给用户程序，
                                x86平台上使用EAX存放返回值，替代u.u_ar0[R0]
                                这里简化，用int代替*/

    int u_arg[5];               /* 存放当前系统调用参数 */
    char* 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 */
    char u_curdir[128];                     /* 当前工作目录完整路径 */

    ErrorCode u_error;          /* 存放错误码 */
    int u_segflg;               /* 表明I/O针对用户或系统空间 */

    /* 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 */
    IOParameter u_IOParam;  /* 记录当前读、写文件的偏移量，用户目标区域和剩余字节
数参数 */
};
```

单一的文件系统不存在信号的处理，因此本次的课程设计删去了 User 结构中对于信号的宏定义，同时单一的用户系统不需要对用户组及用户权限等进行判别，只需要保留支持基本的文件操作成员即可。

另一方面，由于不存在各类中断返回操作，因此用于保存寄存器返回

参数的*u_ar0 这里直接使用 int 类型进行替代即可

5.1.6 文件系统内核 Kernel

```
class Kernel
{
public:
    Kernel();
    ~Kernel();
    static Kernel& Instance();
    void Initialize(char* p);       /* 该函数完成初始化内核大部分数据结构的初始化 */

    BufferManager& GetBufferManager();
    FileSystem& GetFileSystem();
    FileManager& GetFileManager();
    User& GetUser();        /* 获取当前进程的User 结构 */

private:
    void InitBuffer(char* p);
    void InitFileSystem();

private:
    static Kernel instance;     /* Kernel 单体类实例 */

    BufferManager* m_BufferManager;
    FileSystem* m_FileSystem;
    FileManager* m_FileManager;
    User* m_User;
};
```

通过将本次课程设计所需要的各类内核类型封装至一个静态的Kernel类型中，保证每个内核内封装的内核模块仅有一个副本，保证了单体内核的模式，也方便各类间的数据传递。

5.2 API 实现类——SecondFileSystem

```
class SecondFileSystem
{
private:
    const int DEFAULT_MODE = 040755;

public:
```

```
    SecondFileSystem();
    ~SecondFileSystem();

public:
    /* 文件开关 */
    int fopen(char* path, int mode);
    int fclose(int fd);
    /* 文件读写 */
    int fread(int fd, char* buffer, int length);
    int fwrite(int fd, char* buffer, int length);
    /* 文件操作 */
    int fseek(int fd, int pos, int ptrname);
    int fcreate(char* filename, int mode);
    int fdelete(char* filename);
    /* 目录操作 */
    int ls();
    int mkdir(char* dirname);
    int cd(char* dirname);
    int bkdir();
};
```


该类中封装了所有文件系统操作的 API，利用剪切好的 Unix V6++内核函数进行组装，最终实现各类操作，具体实现如下：

1. Fopen

通过将文件名传递至 u.u_dirp，及打开方式传递至 u.u_arg[0]，调用 filemanager 的 Open()函数即可

2. Fclose

类似 Fopen，但此时 u.u_arg[0]内保存的是句柄号，调用 filemanager 的 Close()函数实现

3. Fread

Arg[0]传入文件句柄，arg[1]传入 buffer 地址，arg[2]传入长度，再调用 Read()即可

4. Fwrite

类似 fread，仅需要调用 Write()函数可实现写的操作。这里不得不佩服 Unix V6++内核对于读写代码复用的设计

5. Fseek

类似 fread 与 fwrite，但 Arg[1]传入的是 offest 偏移量，而 arg[2]的 ptrname 则控制的是移动方式，调用函数为 Seek()

6. Fcreate

U_drip 为文件名，而 u_arg[1]中保存的为文件权限，调用函数 Creat()

7. Fdelete

类似 fcreate，但不需要对 u_arg[1]传参，调用 Unlink()即可

8. Ls

该功能为本次唯一一个不能直接调用函数的功能，但实现的本质是利用 fopen 打开当前目录对应的目录文件，再利用 fread 从中读出所有当前目录的文件名，直到至文件尾即可

9. Mkdir

由于 linux 系统中一切皆文件的特性，该功能类似 fcreate，但需要注意 arg[2]需要置为 0，调用 MkNod()函数建立新的 inode 节点

10.Cd

分别向 u_dirp 与 u_arg[0]传入路径名的地址，再调用 ChDir()即可

11.Bkdir

该功能本质为 cd 的变种，通过自 u.u_curdir 提取出当前所处目录的完整路径，再通过由 root 一层一层 cd 至当前目录的上一层即可

六. 效果展示

运行程序，输入-help 可以查看支持的语句

```
192.168.137.64 ×
[root@vm-linux filesystem]# ./SecondFileSystem
Initialize Buffer...OK.
Initialize File System...OK.
Initialize File Manager...OK.
==========系统初始化完毕==========

------------------------------------------------------------------
|            The file system has already loaded                  |
|       Use -help to check the language it supported             |
------------------------------------------------------------------
SecondFileSystem@TaihouDaisuki> -help
*****************************************
* The language supported are as followed *
*=======================================*
*              Operation Part           *
*---------------------------------------*
* -fopen [filename] [mode]              *
* -fclose [fd]                          *
* -fread [fd] [length]                  *
* -fwrite [fd] [source] [length]        *
* -fseek [fd] [position] [ptrname]      *
* -fcreate [filename] [mode]            *
* -fdelete [filename]                   *
* -ls                                   *
* -mkdir [dirname]                      *
* -cd [dirname]                         *
*       (P.S. -cd ../ means <backdir>)  *
* -quit                                 *
*=======================================*
*                 Test Part             *
*---------------------------------------*
* -fread -2 [fd] [length]               *
*       (Read to buffer)                *
* -fwrite -2 [fd]                       *
*        (Write from buffer)            *
* -load [filename]                      *
*      (Load file to buffer from outside)*
* -save [filename]                      *
*      (Save file from buffer to outside)*
* -check                                *
*      (check the buffer state)         *
*****************************************
SecondFileSystem@TaihouDaisuki>
```

装载当前目录下的图片文件，此处使用 img_in.png，使用-check 可以查看装载情况

```
SecondFileSystem@TaihouDaisuki> -load img_in.png
< Attention: File loaded successfully >
SecondFileSystem@TaihouDaisuki> -check
< Attention: Buffer has been loaded, size = 3813542 >
SecondFileSystem@TaihouDaisuki>
```

在当前目录下建立文件夹 test1，进入后创建存储图片文件的临时文件

```
SecondFileSystem@TaihouDaisuki> -mkdir test1
SecondFileSystem@TaihouDaisuki> -ls
< The ls report get the returns as followed >
--------------------------------------------
<< test1 >>
--------------------------------------------
SecondFileSystem@TaihouDaisuki> -cd test1
[/test1]
SecondFileSystem@TaihouDaisuki> -fcreate img 3
< Success: The fd of created file = 1 >
SecondFileSystem@TaihouDaisuki> -ls
< The ls report get the returns as followed >
--------------------------------------------
<< img >>
--------------------------------------------
SecondFileSystem@TaihouDaisuki>
```

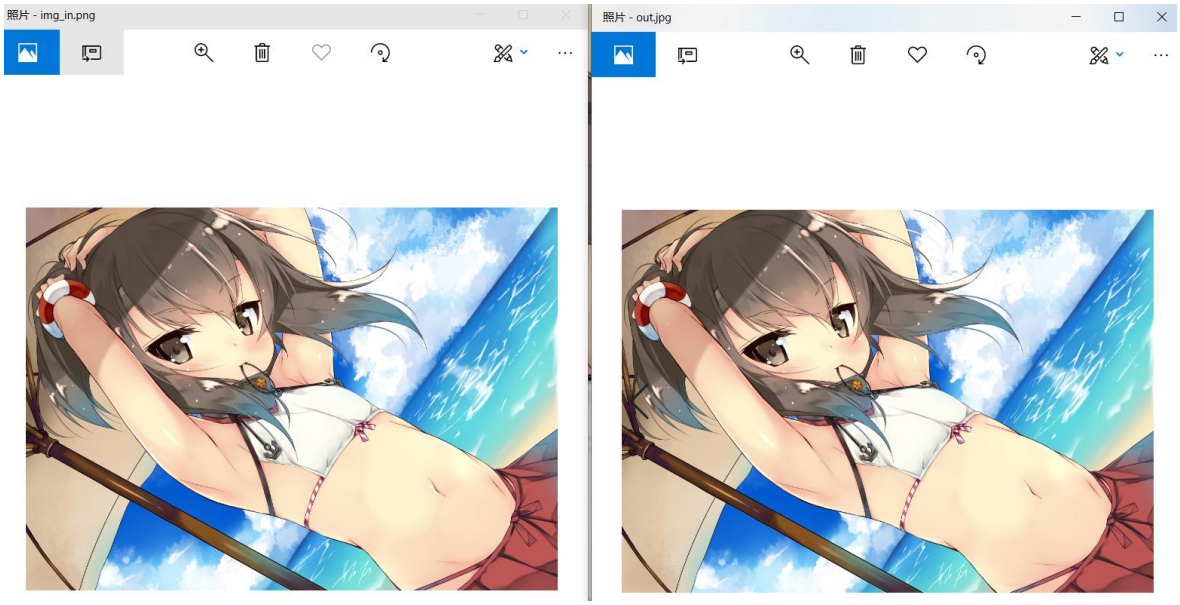
打开 img，向内导入图片数据

```
SecondFileSystem@TaihouDaisuki> -fopen img 3
< Success: The return file fd = 4 >
SecondFileSystem@TaihouDaisuki> -fwrite -2 4
< Success: Fwrite returns = 3813542 >
SecondFileSystem@TaihouDaisuki> -check
< Attention: Buffer is empty >
SecondFileSystem@TaihouDaisuki>
```

再从中取出数据，导出至某 jpg 文件中

```
SecondFileSystem@TaihouDaisuki> -fopen img 3
< Success: The return file fd = 8 >
SecondFileSystem@TaihouDaisuki> -fread -2 8 3813542
< Success: Fread returns = 3813542 >
SecondFileSystem@TaihouDaisuki> -check
 < Attention: Buffer wait to be saved, size = -3813542 >
SecondFileSystem@TaihouDaisuki> -save out.jpg
< Attention: Buffer saved successfully >
SecondFileSystem@TaihouDaisuki> -check
< Attention: Buffer is empty >
SecondFileSystem@TaihouDaisuki>
```

检查图片文件，完全一致



装

订

线

七. 感想与体会

一直到这次实验之前，个人都感觉操作系统离自己非常遥远，是一个非常高深莫测的存在。但没想到这次居然也要自己接手来编写文件系统，所以新情非常复杂。

不得不说，这次的课程设计前期准备时间花费了我将近半个月的时间研读 Unix V6++的代码，操作系统无疑是一个十分大的工程项目，其内部的类型定义与函数环环相扣，想要一时间滤清所有的思路是十分困难的，最后我只能从最顶部开始下手，由 Kernel 一步一步向下挖掘，找到最底部的函数调用与定义，开始细细的去研究这份“精致”的代码。

但这份阅读却使我收获了许多，特别是对于大型工程中内部参数的传递有了新的理解。对于大型项目，其内部的数据和参数多到让人数不清，单单依靠形参和实参的传递无论从时间上还是从空间上来说都是十分浪费的，而 Unix V6++采用了利用引用类型直接向上层类中的公有部分写入参数，由上层类函数调用时直接访问的思想使我大开眼界，明白了当前

我们所编写的各类小程序还仅停留于参数的传递过程之中。

同时，本次的课程设计也暴露了我的许多不足，一方面在失去 IDE 的支持下，大型工程的调试对于我而言可以说非常棘手了，最后不得不采用 cout 的方式不停的输出中间变量来进行调试。当然这也与我模块编写完成后没有进行细致的测试的坏习惯有关；另一方面，自己对于双向链表的应用仍然存在缺陷。在缓存替换算法中由于双向链表的编写失误，导致我在缓存块的查错上耗费了将近 12 个小时的时间，期中将近 9 小时都处于对着打印数据束手无策的情况下，而个人的过于自信又让我忽视了双向链表可能带来的错误，由此造成的大量时间浪费也算是给自己好好的上了一课。

虽然本次课程设计完成了，但其中仍有不足之处，比如对于预读的操作个人偷懒并未实现，希望在今后的学习中能够更加深入的研读源码，加深自己对于操作系统的理解