操作系统课程设计报告

——基于 Unix v6++ 的文件系统设计与实现

学　　院：	电子信息与工程学院
专　　业：	计算机科学与技术
学　　号：	1952339
姓　　名：	张馨月
指导教师：	邓蓉
完成日期：	2022 年 07 月 01 日

目录

操作系统课程设计报告

——基于 Unix v6++ 的文件系统设计与实现

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）。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。

1.2 内容

识别、研读 UNIX V6++ 系统中文件系统和缓存管理模块。按需摘取其中的代码和数据结构，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 secondFileSystem，宿主操作系统可以是 windows 也可以是 Linux。

1. 本实验用某个大文件，如 myDisk.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，磁盘中存储的信息以块为单位。每块 512 字节.

2. myDisk.img 文件拥有标准的 UNIX 卷格式。

引导块	超级块	inode区	数据区

3. 使用 UNIX V6 文件管理系统的内核设计思想。

4. 基于上述思想设计文件管理模块，实现以下 API

```
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);
```

并实现功能：

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。

把随意的一个纯文本文件,你的课设报告和你的头像存进这个文件系统,分别放在 /home-/texts，/home/reports 和/home/photos 文件夹。

5. 实现高速缓存。

6. 提高文件的并发访问速度。

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，根据老师给出的报告范本设计参考思路以及 Unix V6++ 的自身结构，本系统分为以下几个模块：

- 磁盘驱动模块 DiskDriver：负责初始化磁盘镜像文件，调用系统调用对磁盘镜像文件进行读写；
- 高速缓存管理模块 BufferManager：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；
- 文件系统资源管理模块 FileSystem：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；
- 打开文件管理模块 OpenfileManager：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；
- 文件管理接口模块 Filemanager：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；
- 用户操作接口模块 User：主要将用户的界面执行命令转化为对相应函数的调用，同时对输出进行处理，也包含检查用户输入的正确性与合法性。
- 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。

3 详细设计

3.1 数据结构

3.1.1 DiskDriver 类

```
#ifndef DISKDRIVER_H
#define DISKDRIVER_H

#include <iostream>
using namespace std;

class DiskDriver {
public:
    static const int BLOCK_SIZE = 512;  //  数据块大小为512字节

    DiskDriver();
    ~DiskDriver();
    void Initialize();  //  初始化磁盘镜像
    //void IO(Buf* bp);  //  根据缓存控制块读写
    bool Exists();
    void read(void* buffer, unsigned int size, int offset = -1, unsigned int origin = SEEK_SET);   //  读文件
    void write(const void* buffer, unsigned int size, int offset = -1, unsigned int origin = SEEK_SET);
private:
    static const char *DISK_FILE_NAME;  //  磁盘镜像文件名
    FILE *fp;   //  磁盘镜像文件指针
};

#endif
```

DiskDriver 负责对镜像文件的初始化及的读写。

Initialize() 函数在磁盘镜像文件不存在时会创建磁盘镜像文件，并且进行格式化，填入超级块、外存 Inode、数据区等，已存在则直接打开。

分别对三个部分数据进行初始化，并写入磁盘镜像文件。关于初始化的规则参见后面的 FileSystem 中的 SuperBlock 具体定义及外存 Inode 区和空闲盘块的管理。IO() 函数根据传入的缓存块 bp 的相关参数对文件进行读写操作。

3.1.2 Buf 类

```
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;        /* 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 */ //#
    int     b_no;
};
```

buf 定义记录了相应缓存的使用情况等信息；同时兼任 I/O 请求块，记录该缓存相关的 I/O 请求和执行结果。

3.1.3 BufferManager 类

```
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 FormatBuffer();                //  格式化缓存控制块

    void Initialize();                  /* 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 */
    //void IOWait(Buf* bp);             /* 同步方式I/O，等待I/O操作结束 */
    //void IODone(Buf* bp);             /* I/O操作结束善后处理 */

    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。 */

    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 */

    //Buf& GetBFreeList();              /* 获取自由缓存队列控制块Buf对象引用 */
    void DetachNode(Buf *bp);   //  LRU Cache算法，每次从头部取出，使用后放到尾部

private:
    //void GetError(Buf* bp);               /* 获取I/O操作中发生的错误信息 */ //#
    void NotAvail(Buf* bp);             /* 从自由队列中摘下指定的缓存控制块buf */ //#
    //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 SwBuf;                          /* 进程图像传送请求块 */
    Buf m_Buf[NBUF];                    /* 缓存控制块数组 */
    unsigned char Buffer[NBUF][BUFFER_SIZE];    /* 缓冲区数组 */
    map<int, Buf*> mp;
    DiskDriver* m_DiskDriver;       /* 指向设备管理模块全局对象 */
};

#endif
```

BufferManager 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。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。

本系统未实现预读，故去除了 Breada() 函数，另外由于是单进程，故写操作也是同步的，去除了 Bawrite() 函数。自由缓存队列管理为 LRU 算法，将最近使用的缓存块从原队列取出，并立即放入队列尾部。

3.1.4 Inode 类

```
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，清ILOCK标志时，要唤醒这种进程 */
    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 IFBLK = 0x6000;       /* 块设备特殊类型文件，为0表示常规数据文件 */
static const unsigned int ILARG = 0x1000;       /* 文件长度类型：大型或巨型文件 */
static const unsigned int ISUID = 0x800;        /* 执行时文件时将用户的有效用户ID修改为文件所有者的User ID */
static const unsigned int ISGID = 0x400;        /* 执行时文件时将用户的有效组ID修改为文件所有者的Group ID */
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);     /* 文件主对文件的读、写、执行权限 */
static const unsigned int IRWXG = ((IRWXU) >> 3);           /* 文件主同组用户对文件的读、写、执行权限 */
static const unsigned int IRWXO = ((IRWXU) >> 6);           /* 其他用户对文件的读、写、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;

/* static member */
static int rablock;     /* 顺序读时，使用预读技术读入文件的下一字符块，rablock记录了下一逻辑块号
                           经过bmap转换得到的物理盘块号。将rablock作为静态变量的原因：调用一次bmap的开销
                           对当前块和预读块的逻辑块号进行转换，bmap返回当前块的物理盘块号，并且将预读块
                           的物理盘块号保存在rablock中。 */
```

```
public:
    Inode();
    ~Inode();
    /*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();
    /* 根据Inode对象中的物理磁盘块索引表，将数据写入文件*/
    void WriteI();
    /* 将文件的逻辑块号转换成对应的物理盘块号*/
    int Bmap(int lbn);
    /* 更新外存Inode的最后的访问时间、修改时间*/
    void IUpdate(int time);
    /*释放Inode对应文件占用的磁盘块*/
    void ITrunc();
    /* 清空Inode对象中的数据*/
    void Clean();
    /* 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);
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 */

    short   i_dev;          /* 外存inode所在存储设备的设备号 */
    int     i_number;       /* 外存inode区中的编号 */

    short   i_uid;          /* 文件所有者的用户标识数 */
    short   i_gid;          /* 文件所有者的组标识数 */

    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 */
};
```

Inode 对应系统中每一个打开的文件、当前访问目录，内存 Inode 通过 i_number 来确定其对应的外存 Inode。

DiskInode 位于文件存储设备的外存 Inode 区中，每个文件唯一对应的外存 Inode，其作用是记录了该文件对应的控制信息。

其中 Bmap() 函数用于逻辑盘块号到物理盘块号的映射，具体映射机制参见后面 Unix v6++ 的混合索引机制。

Inode 类的函数基本同 Unix v6++。

3.1.5 SuperBlock 类

```
/*
 * 文件系统存储资源管理块(Super Block)的定义。
 */
class SuperBlock
{
public:
    const static int MAX_NFREE = 100;  //  最大空闲存储资源管理块数量
    const static int MAX_NINODE = 100; //  最大Inode数量
                                       /* Functions */
public:
    /* Constructors */
    SuperBlock();
    /* Destructors */
    ~SuperBlock();
    //  格式化文件系统存储资源管理块
    void Format();

    /* 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_flock; /* 封锁空闲盘块索引表标志 */
    int s_ilock; /* 封锁空闲Inode表标志 */

    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[47]; /* 填充使SuperBlock块大小等于1024字节，占据2个扇区 */
};
```

SuperBlock 类定义了文件系统存储资源管理块。由于本系统为单用户单线程，去除了锁的使用，但由于 SuperBlock 大小要求为 1024 字节，占满两个扇区，故此处不删除锁相关变量。

SuperBlock 对于空闲 Inode 的管理，任何时候只管理 s_ninode 个空闲 Inode，按栈的方式使用，当管理的 Inode 分配完后，再重新在 Inode 区搜索 100 个空闲的 Inode 加入管理。

而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。

3.1.6 FileSystem 类

```
/*
 * 文件系统类(FileSystem)管理文件存储设备中
 * 的各类存储资源，磁盘块、外存INode的分配、
 * 释放。
 */
class FileSystem
{
public:
    /* static consts */
    static const int SUPER_BLOCK_SECTOR_NUMBER = 200; /* 定义SuperBlock位于磁盘上的扇区号，占据200，201两个扇区。 */
    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 - 2;  /* 磁盘上外存Inode区占据的扇区数 */
    static const int INODE_NUM = INODE_NUMBER_PER_SECTOR * INODE_ZONE_SIZE;
    static const int DISK_SIZE = 16384; //  磁盘所有扇区数量
    static const int DATA_ZONE_START_SECTOR = 1024;                     /* 数据区的起始扇区号 */
    static const int DATA_ZONE_END_SECTOR = DISK_SIZE - 1;              // 18000 - 1; /* 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 */
    // Block块大小
    static const int BLOCK_SIZE = 512;
    // 定义SuperBlock位于磁盘上的扇区号，占据两个扇区
    static const int SUPERBLOCK_START_SECTOR = 0;
public:
    FileSystem();
    ~FileSystem();
    //  格式化文件系统
    void FormatFS();
    /*初始化成员变量*/
    void Initialize();
    /*系统初始化时读入SuperBlock*/
    void LoadSuperBlock();
    /*根据文件存储设备获取该文件系统的SuperBlock*/
    SuperBlock *GetFS();
    /*将SuperBlock对象的内存副本更新到存储设备的SuperBlock中去*/
    void Update();
    /*在存储设备上分配一个空闲外存INode，一般用于创建新的文件。*/
    Inode *IAlloc();
    /*释放存储设备上编号为number的外存INode，一般用于删除文件。*/
    void IFree(int number);
    /*在存储设备上分配空闲磁盘块*/
    Buf *Alloc();
    /*释放存储设备编号为blkno的磁盘块*/
    void Free(int blkno);
public:
    DiskDriver *diskDriver;
    SuperBlock *superBlock;
    BufferManager *bufferManager;
```

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，故超级块区从 0# 盘块开始，其他区也做相应修改。

外存索引节点分配 IAlloc 函数：外存 DiskINode 索引节点采用栈式管理 DiskINode，当需要分配 DiskINode 时，如果 s_ninode 不为 0，则将栈顶节点分配，栈指针减一；否则 s_ninode 为 0，说明外存索引节点表中已不包含任何空闲节点，就重新搜索整个 DiskINode 区，将找到的 DiskINode 号顺次登入 s_inode 表中，直到该表已满或者已搜索完整个 DiskINode 区。

外存索引节点分释放 IFree 函数：当释放 DiskINode 节点时，如果超级块索引节点表中空闲 DiskINode 数小于 SuperBlock::MAX_NINODE，则将该索引节点编号记入“栈顶”，若其中记录的空闲 DiskINode 已满，则将其散落在磁盘 DiskINode 区中。

外存空闲盘块分配 Alloc 函数：外存中的空闲盘块采用分组链式索引法进行管理。超级块中的空闲块索引表用栈式管理空闲数据块，但是它最多只能直接管理 MAX_NFREE（100）个空闲块。所有空闲块按照每 MAX_NFREE（100）个进行构成一组，最后一组直接由超级块中的空闲索引表进行管理，其余各组的索引表分别存放在它们下一组第一个盘块的开头中。

分配空闲盘块时，总是从索引表中取其最后一项的值，即 s_free[–s_nfree], 相当于出栈，当及时直接管理的最后一个空闲盘块时，就将该盘块的前 404 字节读入超级块的 s_nfree 和索引表 s_free 数组中，使得用间接方式管理的下一组变为直接管理。

3.1.7 File 类

```
class File
{
public:
    /* Enumerate */
    enum FileFlags
    {
        FREAD = 0x1,            /* 读请求类型 */
        FWRITE = 0x2,           /* 写请求类型 */
        FPIPE = 0x4             /* 管道类型 */
    };

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


    /* Member */
    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 */
};
```

File 类记录了进程打开文件的读、写请求类型、文件读写位置等。

3.1.8 OpenFiles 类

```
class OpenFiles
{
    /* static members */
public:
    static const int NOFILES = 15;  /* 进程允许打开的最大文件数 */

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

    /* Members */
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 */
};
```

OpenFiles 类是进程的 u 结构中包含的一个对象，维护了当前进程的所有打开文件。

3.1.9 IOParameter 类

```
/*
 * 文件I/O的参数类
 * 对文件读、写时需用到的读、写偏移量、
 * 字节数以及目标区域首地址参数。
 */
class IOParameter
{
    /* Functions */
public:
    /* Constructors */
    IOParameter();
    /* Destructors */
    ~IOParameter();

    /* Members */
public:
    unsigned char* m_Base;  /* 当前读、写用户目标区域的首地址 */
    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 */
};

#endif
```

IOParameter 记录了对文件读、写时需用到的读、写偏移量、字节数以及目标区域首地址参数。

3.1.10 OpenfileTable 类

```
/*
 * 打开文件管理类(OpenFileManager)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 OpenFileTable
{
public:
    static const int NFILE  = 100;  /* 打开文件控制块File结构的数量 */

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

    // 格式化当前打开文件表
    void Format();
    /*
     * @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);

    /* Members */
public:
    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。*/
};
```

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。

3.1.11 InodeTable 类

```
/*
 * 内存Inode表(class InodeTable)
 * 负责内存Inode的分配和释放。
 */
class InodeTable
{
public:
    static const int NINODE = 100;  /* 内存Inode的数量 */
public:
    InodeTable();
    ~InodeTable();

    //  格式化Inode表
    void Format();
    /* 初始化对g_FileSystem对象的引用*/
    void Initialize();
    /* 根据指定外存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);
    /* 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);
    /* 将所有被修改过的内存Inode更新到对应外存Inode中*/
    void UpdateInodeTable();
    /* 检查编号为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);
    /*在内存Inode表中寻找一个空闲的内存Inode*/
    Inode* GetFreeInode();
public:
    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 */
};
```

InodeTable 类负责内存 Inode 的分配和释放。

3.1.12 FileManager 类

```
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 */
    };
public:
    FileManager();
    ~FileManager();
    /* 初始化对全局对象的引用*/
    void Initialize();
    /*Open()系统调用处理过程*/
    void Open();
    /*Creat()系统调用处理过程*/
    void Creat();
    /*Open()、Creat()系统调用的公共部分*/
    void Open1(Inode* pInode, int mode, int trf);
    /*Close()系统调用处理过程*/
    void Close();
    /*Seek()系统调用处理过程*/
    void Seek();
    /*Read()系统调用处理过程*/
    void Read();
    /*Write()系统调用处理过程*/
    void Write();
    /*读写系统调用公共部分代码*/
    void Rdwr(enum File::FileFlags mode);
    /* 目录搜索，将路径转化为相应的Inode，返回上锁后的Inode*/
    Inode* NameI(/*char (*func)(), */enum DirectorySearchMode mode);
    /* 获取路径中的下一个字符*/
    static char NextChar();
    /*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);
    /*向父目录的目录文件写入一个目录项*/
    void WriteDir(Inode* pInode);
    /*设置当前工作路径*/
    void SetCurDir(char* pathname);
    /* 改变当前工作目录 */
    void ChDir();
    /* 取消文件 */
    void UnLink();
    /* 用于建立特殊设备文件的系统调用 */
    void MkNod();
    //  列出当前Inode节点的文件项
    void Ls();
public:
    /* 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;
};
```

FileManager 类封装了对文件的系统调用的核心态操作，供顶层 API 模块进行调用。

3.1.13 DirectoriEntry 类

```
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 */
    };
public:
    FileManager();
    ~FileManager();
    /* 初始化对全局对象的引用*/
    void Initialize();
    /*Open()系统调用处理过程*/
    void Open();
    /*Creat()系统调用处理过程*/
    void Creat();
    /*Open()、Creat()系统调用的公共部分*/
    void Open1(Inode* pInode, int mode, int trf);
    /*Close()系统调用处理过程*/
    void Close();
    /*Seek()系统调用处理过程*/
    void Seek();
    /*Read()系统调用处理过程*/
    void Read();
    /*Write()系统调用处理过程*/
    void Write();
    /*读写系统调用公共部分代码*/
    void Rdwr(enum File::FileFlags mode);
    /* 目录搜索，将路径转化为相应的Inode，返回上锁后的Inode*/
    Inode* NameI(/*char (*func)(), */enum DirectorySearchMode mode);
    /* 获取路径中的下一个字符*/
    static char NextChar();
    /*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);
    /*向父目录的目录文件写入一个目录项*/
    void WriteDir(Inode* pInode);
    /*设置当前工作路径*/
    void SetCurDir(char* pathname);
    /* 改变当前工作目录 */
    void ChDir();
    /* 取消文件 */
    void UnLink();
    /* 用于建立特殊设备文件的系统调用 */
    void MkNod();
    //  列出当前Inode节点的文件项
    void Ls();
public:
    /* 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;
};
```

DirectoryEntry 类定义了目录项的结构，由 Inode 编号加路径名组成。目录文件保存在磁盘上，一个盘块可保存 16 个目录项，根据目录项的 Inode 编号找到目录中的文件。

3.1.14 User 类

```
public:
    User();
    ~User();

    void Ls();
    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 Write(string fd, string inFile, string size);
    void Read(string fd, string outFile, string size);
    /* 系统调用相关成员 */
    unsigned int    u_ar0[5];       /* 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] */
    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 */
    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 */
    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 */
    string u_ls;
private:
    bool IsError();
    void EchoError(enum ErrorCode err);
    int INodeMode(string mode);
    int FileMode(string mode);
    bool checkPathName(string path);

    FileManager* fileManager;
};
```

User 结构主要记录一些参数，方便其他函数使用时不用大量传参。

3.1.15 Utility 类

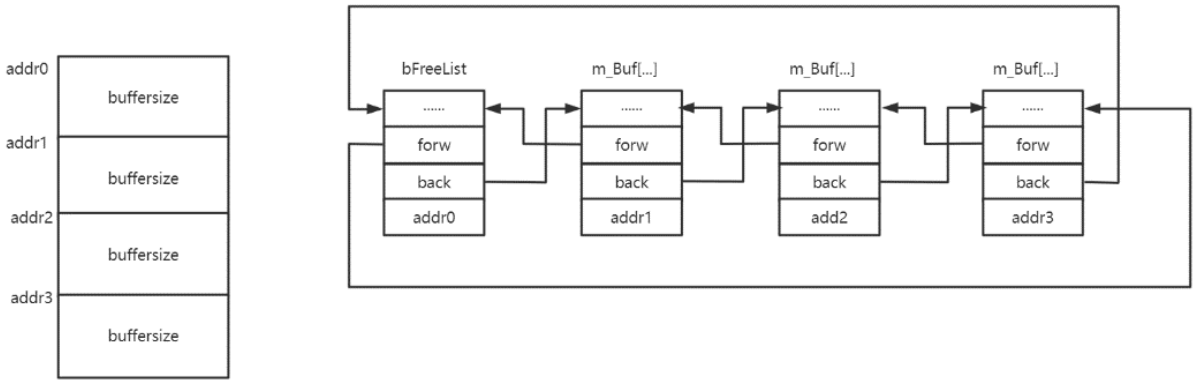
```
*@comment 定义一些工具常量
* 由于使用了编译选项-fno-builtin，
* 编译器不提供这些常量的定义。
*/

class Utility
{
public:

    static void MemSet(void *s, int ch, size_t n);

    static void MemCopy(void *des, const void* src, unsigned int count);

    int MemCmp(const void *buf1, const void *buf2, unsigned int count);

    static void StringCopy(char* src, char* dst);

    static int StringLength(char* pString);

    /* 以src为源地址，dst为目的地址，复制count个双字 */
    static void DWordCopy(int* src, int* dst, int count);

    static int Min(int a, int b);

    /* 用于在读、写文件时，高速缓存与用户指定目标内存区域之间数据传送 */
    static void IOMove(unsigned char* from, unsigned char* to, int count);

    static time_t time(time_t* t);
};
```

Utility 为实用工具类，封装了一些实用公共函数。本系统保留了上述五个在系统中使用到的函数。

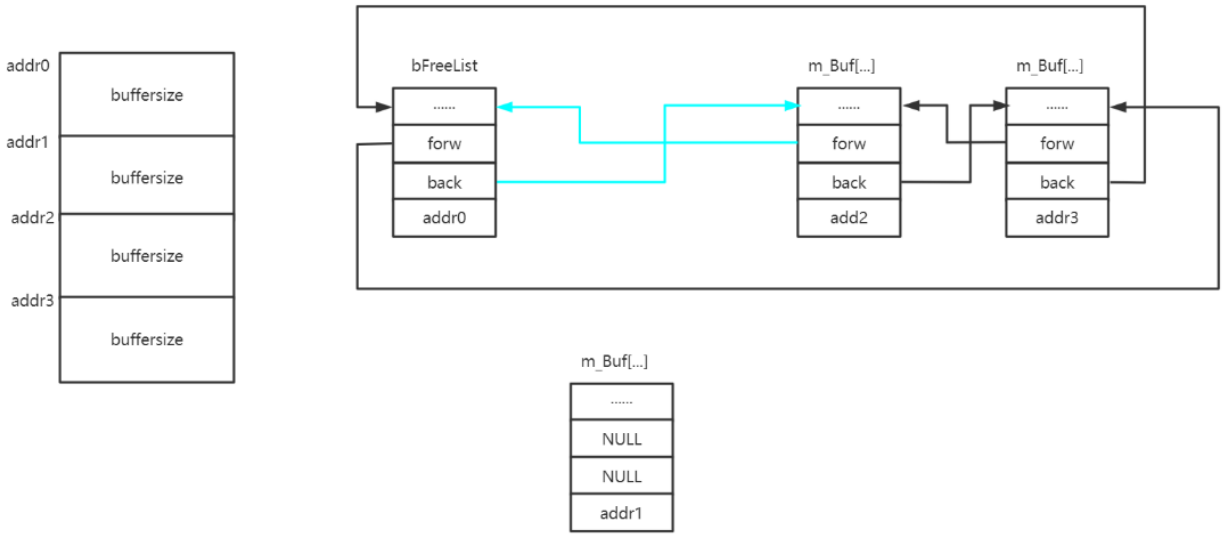
3.2 高速缓存设计

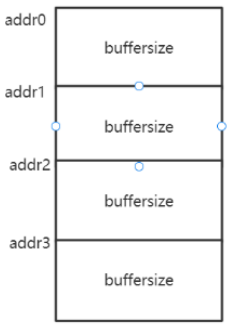
Unix 系统使用自由队列和设备队列来对缓存块进行管理，本次课程项目是实现 Unix V6 的文件系统，不包含进程等一系列的设计，并且使用一个大文件当一张磁盘用，使用系统调用来实现任务。因为是单用户系统，且只有一个设备块（即用来模拟的大文件），所以拟将自由队列和设备队列整合为一个缓存队列。、

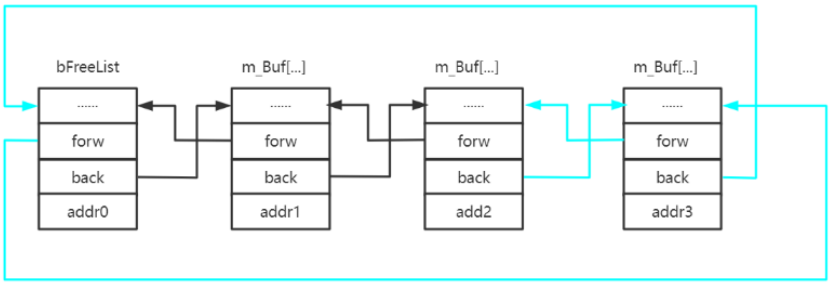
高速缓存结构：使用双向循环链表存储缓存队列，每个节点存储前后节点地址以及指向的缓存地址



队列维护算法：使用 LRU 算法维护队列，每次使用时分离队列中第一个节点（过去一段时间最久未使用的节点），放弃这一块缓存。每次访问缓存队列时，都将找到的缓存块移动到队列尾部，对于新建的缓存块，也放在队列尾部，以此来维护 LRU 算法。







4 顶层设计

4.1 系统初始化

本系统的初始化全部放在各个类型的构造函数中，在 main 中依照全局变量的定义次序进行初始化，即可达到系统初始化的目的。还给出了 Fformat 指令用于格式化文件系统。

4.2 API

在 User 类中实现了接口的封装后，直接在 main 函数中调用对应的接口即可。

4.2.1 fcreate

实现为 User::Create(string fileName, string mode)。新建文件。首先检查 dirname 是否存在，然后解析文件模式，并存入 u_arg[1]，然后调用 fileManager 的 Creat 方法。

4.2.2 fopen

实现为 User::Open(string fileName, string mode)。打开文件。首先检查 dirname 是否存在，然后解析文件模式，并存入 u_arg[1]，然后调用 fileManager 的 Open 方法。

4.2.3 fread

实现为 User::Read(string sfd, string outFile, string size)。将 file1 内容输出到 file2，默认输出到 shell。将三个参数分别填入 u_arg[0 3] 中，然后调用 fileManager 的 Read 方法。使用 fstream 读入文件内容并输出到屏幕。

4.2.4 fwrite

实现为 User::Write(string sfd, string inFile, string size)。将 file2 中内容写入 file1, 指定写入 size 字节。打开文件并写入，然后调用 fileManager 的 Write 方法。

4.2.5 fdelete

实现为 User::Delete(string fileName)。删除文件。首先检查 dirname 是否存在，然后调用 fileManager 的 UnLink 方法。

4.2.6 flseek

实现为 User::Seek(string sfd, string offset, string origin)。移动读写指针。将三个参数分别填入 u_arg[0 3] 中，然后调用 fileManager 的 Seek 方法。

4.2.7 fclose

实现为 User::Close(string sfd)。关闭文件。首先检查文件描述符正确性，然后填入 u_arg[0]，调用 fileManager 的 Close 方法

4.2.8 ls

实现为 User::Ls()。列出目录及文件。调用 fileManager 的 Ls 方法

4.2.9 mkdir

实现为 User::Mkdir(string dirName)。创建目录文件。首先检查 dirname 是否存在，然后填入 u_arg[1] 参数，调用 fileManager 的 Creat 方法。

4.2.10 cd

实现为 User::Cd(string dirName)。改变目录。首先检查 dirname 是否存在，然后调用 fileManager 的 ChDir 方法

4.2.11 help

实现为 void Help()。输出使用手册。

5 代码测试

5.1 运行环境

宿主机：Linux Rocky8，64 位

编译器：gcc version 8.5.0 20210514 (Red Hat 8.5.0-10) (GCC)

5.2 使用说明

```
[root@VM-0-17-rockylinux ~]# cd SecondFS/
[root@VM-0-17-rockylinux SecondFS]# ./secondFS
Welcome to Elaine's Second File System! Input "help" to get the usage.
[root@SecondFS / ]#help
Fformat                              : 格式化文件系统
exit                                 : 退出文件系统
mkdir <dir>                          : 新建目录
cd <dir>                             : 改变目录
ls                                   : 列出目录及文件
create <file> [-r -w]                : 新建文件
delete <file>                        : 删除文件
open <file> [-r -w]                  : 打开文件
close <file>                         : 关闭文件
seek <file> <offset> <origin>        : 移动读写指针
write <file1> <file2> <size>         : 将file2中内容写入file1,指定写入size字节
read <file1> [-o <file2>] <size>     : 将file1内容输出到file2,默认输出到shell
autoTest                             : 使用自动测试
help done
[root@SecondFS / ]#
```

5.3 功能测试

```
[root@VM-0-17-rockylinux SecondFS]# ./secondFS
Welcome to Elaine's Second File System! Input "help" to get the usage.
[root@SecondFS / ]#mkdir haha
[root@SecondFS / ]#create aa -rw
[root@SecondFS / ]#open aa -w
open success, return fd=2
[root@SecondFS / ]#write 2 ./src/main.cpp 20
write 20bytes success !
[root@SecondFS / ]#close 2
[root@SecondFS / ]#open aa -r
open success, return fd=2
[root@SecondFS / ]#read 2 10
read 10 bytes success :
#include <
[root@SecondFS / ]#ls
haha
aa

[root@SecondFS / ]#delete haha
[root@SecondFS / ]#ls
aa

[root@SecondFS / ]#exit
[root@VM-0-17-rockylinux SecondFS]# ./secondFS
Welcome to Elaine's Second File System! Input "help" to get the usage.
[root@SecondFS / ]#ls
aa

[root@SecondFS / ]#
```