**课 程 设 计 报 告**

设计题目：模拟实现UNIX的文件系统

班 级：物联网1702班、物联网1701班

组长学号：20174507

组长姓名：王德君

指导教师：吕鸣松

设计时间：2020年6月

设计分工

组长学号及姓名：20174507 王德君

分工：

1. 整体结构设计
2. 磁盘文件结构实现（superblock、inode table、datablock）
3. 文件目录结构实现（多级目录）
4. 文件打开结构实现
5. 磁盘高缓实现
6. 文件操作接口实现（open、ls/dir/tree、cd、stat）
7. 用户权限控制实现（login、logout、chmod、chown）

组员1学号及姓名：20174541 刘宇辰

分工：

1. 主程序流程实现
2. 文件操作接口实现（read、seek、write）

组员2学号及姓名：20174699 贾逸斐

分工：

1. 文件操作接口实现（mkdir、rmdir、create、delete、close）

组员3学号及姓名：20174662 孙毅

分工：

1. 文件操作接口实现（format）
2. 命令行功能实现（history、autoTest、exit、快捷键支持）

**目录**

[摘要 1](#_Toc12221)

[1. 概述 2](#_Toc5006)

[2. 课程设计任务及要求 2](#_Toc31102)

[2.1. 设计任务 2](#_Toc18423)

[2.2. 设计要求 3](#_Toc32352)

[3. 算法及数据结构 4](#_Toc19863)

[3.1. 算法的总体思想（流程） 4](#_Toc16593)

[3.2. 磁盘驱动模块DeviceDriver 6](#_Toc23542)

[3.2.1. 功能 6](#_Toc9662)

[3.2.2. 数据结构 6](#_Toc18860)

[3.2.3. 算法 7](#_Toc8507)

[3.3. 磁盘高速缓冲模块BufferManager 7](#_Toc15096)

[3.3.1. 功能 7](#_Toc3382)

[3.3.2. 数据结构 7](#_Toc30216)

[3.3.3. 算法 8](#_Toc8991)

[3.4. 文件系统模块FileSystem 9](#_Toc4772)

[3.4.1. 功能 9](#_Toc5990)

[3.4.2. 数据结构 9](#_Toc10269)

[3.4.3. 算法 12](#_Toc3638)

[3.5. 目录管理模块FileManager 17](#_Toc29603)

[3.5.1. 功能 17](#_Toc6883)

[3.5.2. 数据结构 17](#_Toc634)

[3.5.3. 算法 20](#_Toc18398)

[3.6. 系统打开文件管理模块OpenFileManager 20](#_Toc1700)

[3.6.1. 功能 20](#_Toc21293)

[3.6.2. 数据结构 20](#_Toc29170)

[3.6.3. 算法 22](#_Toc27573)

[3.7. 用户接口模块Users 24](#_Toc24034)

[3.7.1. 功能 24](#_Toc1531)

[3.7.2. 数据结构 24](#_Toc21095)

[3.7.3. 算法 27](#_Toc14444)

[4. 程序设计与实现 28](#_Toc6193)

[4.1. 重要函数程序流程图 28](#_Toc10324)

[4.1.1. 逻辑块号到物理块号的转换—Bmap() 28](#_Toc31428)

[4.1.2. 索引节点的获取—IGet() 30](#_Toc1883)

[4.1.3. 索引节点的释放—IPut() 32](#_Toc28818)

[4.1.4. 目录的搜索—NameI() 33](#_Toc30699)

[4.1.5. 文件读、写公共过程—Rdwr() 36](#_Toc2326)

[4.1.6. 文件读、写函数—ReadI() & WriteI() 37](#_Toc20325)

[4.2. 程序说明及实验结果 39](#_Toc180)

[4.2.1. 登录功能 39](#_Toc19197)

[4.2.2. 列出目录及文件信息 40](#_Toc20657)

[4.2.3. 列出历史命令 41](#_Toc21596)

[4.2.4. 切换当前目录 42](#_Toc951)

[4.2.5. 文件操作 43](#_Toc7295)

[4.2.6. 文件系统格式化 47](#_Toc7115)

[5. 参考文献 47](#_Toc18030)

[6. 结论、收获、体会和建议 48](#_Toc31900)

# **摘要**

文件系统是操作系统用于明确存储设备或分区上的文件的方法和数据结构；即在存储设备上组织文件的方法。从系统角度来看，文件系统是对文件存储设备的空间进行组织和分配，负责文件存储并对存入的文件进行保护和检索的系统。具体地说，它负责为用户建立文件，存入、读出、修改、转储文件，控制文件的存取，当用户不再使用时撤销文件等。

在本次课程设计中，我们选择的是模拟实现Unix文件系统这一课题，目标是实现一个支持多用户登录、拥有多级子目录的文件系统，结合老师上课所讲知识，在小组成员的通力合作之下，我们如期完成了这一项目的设计。不仅成功实现了多用户登录/登出、多级子目录实现、读/写文件等全部基础功能，并且在此基础上开发了若干附加功能。

该文件系统采用成组链接法进行空闲盘块的分配，在iNode分配方面，我们选用了“位视图法+缓冲队列”这一方法。同时，文件系统支持磁盘高速缓存、系统打开文件表、用户权限控制、以及查看历史命令。尽管文件系统是以命令行形式运行，但是本系统自带用户帮助手册，只需键入一个命令，就可以查看每个命令的全部功能与详细介绍。

**关键词：**Unix、文件系统、iNode、磁盘高速缓存、系统打开文件表

# **概述**

本次课程设计，小组编程设计了一个模拟Unix文件系统。设计过程中，我们使用了一个普通的大文件（如 D:\myDisk.img，称之为一级文件）模拟UNIX FileSystem的一个文件卷，即作为一张逻辑磁盘。其中磁盘中存储的信息以块为单位，每块512B。

该模拟文件系统可以实现文件系统的大部分管理功能，如多用户的登录、登出；多级子目录；目录的创建和修改；文件的读、写、删除等基本功能，采用成组链接法管理空闲磁盘块，通过位视图法+缓冲队列进行iNode分配。此外，文件系统还提供搜索、存取权限控制、文件拷入拷出系统、查看历史命令记录等功能。而且对于每一个功能都具有报错机制，当用户输入错误时，可以通过交互性语句，对用户进行引导。与此同时，本文件系统自带用户帮助手册，对于每一个命令，从功能到用法，都有详尽的解释说明，对于初用本文件系统的用户有极大的帮助作用。

本文件系统采用六个模块分别承担不同的功能：设备驱动模块(DeviceDriver) 、高速缓存管理模块 (BufferManager)、系统盘块管理模块 (FileSystem)、系统文件操作功能实现模块(FileManager)、打开文件管理模块(OpenFileManager)、用户操作接口模块(User)。六个模块基本上涵盖了文件系统全部的功能。

在为期两周多的课程设计过程中，小组人员齐心协力，经过不懈努力，先后完成了主程序流程的设计、各部分数据结构以及算法的分析和实现，在完成全部基础功能的情况下，又为文件系统增加了若干额外功能，向本次课程设计提交了一份令人满意的答卷。

# **课程设计任务及要求**

## **设计任务**

多用户、多级目录结构文件系统的设计与实现，模拟UNIX（或LINUX，或FAT）系统的文件管理功能。包括:

多用户：usr1, …, usr8 (1-8个用户)

多级目录：可有多级子目录

具有login (用户登录)

系统初始化（建文件卷、提供登录模块）

文件的创建：create

文件的打开：open

文件的读：read

文件的写：write

文件关闭：close

删除文件：delete

创建目录：mkdir

改变目录：chdir

列出文件目录：dir

退出：logout

格式化：format

以上是基本内容，可以根据实际文件系统提供的命令和系统调用，自己增加和实现附加的功能。

## **设计要求**

1. 在深入理解操作系统基本原理的基础上，对于选定的题目，以小组为单位，先确定设计方案；
2. 设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；
3. 编程序实现系统，要求实现可视化的运行界面，界面应清楚地反映出系统的运行结果；
4. 确定测试方案，选择测试用例，对系统进行测试；
5. 运行系统并要通过验收，讲解运行结果，说明系统的特色和创新之处，并回答指导教师的提问；
6. 提交课程设计报告。

# **算法及数据结构**

## **算法的总体思想（流程）**

* **模块构成**

经过分析，整个文件系统可以由以下几个部分组成：

1. DeviceDriver：设备驱动模块
2. BufferManager：高速缓存管理模块
3. FileSystem：系统盘块管理模块
4. FileManager：系统文件操作功能实现模块
5. OpenFileManager：打开文件管理模块
6. User：用户操作接口模块

* **模块调用关系（类图）**



图3.1-1 文件系统各模块主要类结构

其中，FileSystem类用于管理文件系统的存储资源，包括空闲磁盘块，空闲外存索引节点(DiskInode)这类外存资源，对这类资源的分配和释放算法，以及文件系统挂载点的信息，其中m\_Mount[0]项记录根文件系统的超级块(SuperBlock)信息。

InodeTable和OpenFileTable类用于管理文件系统模块中内存相关数据结构，主要是内存索引节点(Inode)表和系统打开文件控制块表这两个用于管理当前打开文件在内存中的管理机构。这些类的详细结构将在后面具体介绍。

* **系统运行流程**

首先，用户登陆系统时，会对用户键入的登录信息进行核实，进行用户账号认证，登陆成功后，方可开始使用其余功能。与此同时，文件系统开始调用系统的初始化，并进入等待状态，等待用户输入下一步的指令。

当用户输入指令后，首先对指令进行判断，若指令错误（如输入不存在的指令），则优先返回报错信息。若输入的指令正确，则进入下一步判断状态。

对不同的指令，调用执行对应接口所包含的函数.在调用函数成功之后，首先判断用户输入的参数是否正确，如果输入参数错误，则提示输入错误，并重新进入等待状态，等待用户输入正确的参数。若参数正确，则开始执行相应函数，并返回用户所需要的值。

如此循环往复，直至用户完成所有需求，停止输入，并选择关闭系统，此时文件系统会刷新缓存，再退出。

总体流程图如下：

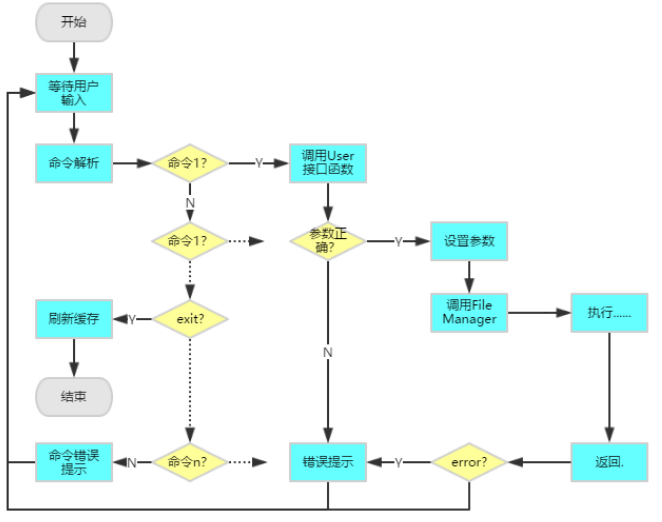


图3.1-2 总体流程图

## **磁盘驱动模块DeviceDriver**

### **功能**

DeviceDriver模块，是设备驱动模块，直接负责磁盘文件直接读写。

### **数据结构**

class DeviceDriver {

public:

/\* 磁盘镜像文件名 \*/

static const char\* DISK\_FILE\_NAME;

private:

/\* 磁盘文件指针 \*/

FILE\* fp;

public:

DeviceDriver();

~DeviceDriver();

/\* 检查镜像文件是否存在 \*/

bool Exists();

/\* 打开镜像文件 \*/

void Construct();

/\* 实际写磁盘函数 \*/

void write(const void\* buffer, unsigned int size,

int offset = -1, unsigned int origin = SEEK\_SET);

/\* 实际写磁盘函数 \*/

void read(void\* buffer, unsigned int size,

int offset = -1, unsigned int origin = SEEK\_SET);

};

### **算法**

DeviceDriver类较为简单，中提供了最底层的物理文件读写函数，其中使用了C++所支持的fseek等函数，可以为上层FileManger类中的读写函数等API提供服务。

## **磁盘高速缓冲模块BufferManager**

### **功能**

BufferManager模块，是高速缓存管理模块，主要负责管理系统中所有的缓存块，包括申请、释放、读写、清空一块缓存的功能函数接口，以及系统退出时刷新所有缓存块。

### **数据结构**

class BufferManager {

public:

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 \*/

private:

Buffer\* bufferList; /\* 缓存队列控制块 \*/

Buffer nBuffer[NBUF]; /\* 缓存控制块数组 \*/

unsigned char buffer[NBUF][BUFFER\_SIZE]; /\* 缓冲区数组 \*/

unordered\_map<int, Buffer\*> map;

DeviceDriver\* deviceDriver;

public:

BufferManager();

~BufferManager();

/\* 申请一块缓存，用于读写设备上的块blkno。\*/

Buffer\* GetBlk(int blkno);

/\* 释放缓存控制块buf \*/

void Brelse(Buffer\* bp);

/\* 读一个磁盘块，blkno为目标磁盘块逻辑块号。 \*/

Buffer\* Bread(int blkno);

/\* 写一个磁盘块 \*/

void Bwrite(Buffer\* bp);

/\* 延迟写磁盘块 \*/

void Bdwrite(Buffer\* bp);

/\* 清空缓冲区内容 \*/

void Bclear(Buffer\* bp);

/\* 将队列中延迟写的缓存全部输出到磁盘 \*/

void Bflush();

/\* 获取空闲控制块Buf对象引用 \*/

//Buffer& GetFreeBuffer();

/\* 格式化所有Buffer \*/

void FormatBuffer();

private:

void InitList();

void DetachNode(Buffer\* pb);

void InsertTail(Buffer\* pb);

void debug();

};

### **算法**

使用缓冲池实现磁盘高缓，从而为文件系统添加延迟写等支持。

## **文件系统模块FileSystem**

### **功能**

FileSystem模块，是系统盘块管理模块，主要负责对镜像文件的存储空间管理，包括 SuperBlock 空间占用、DiskINode 空间分布、数据块区空间分布的管理。需要提供分配、回收 DiskINode 节点、数据块节点以及格式化磁盘文件的接口。

### **数据结构**

class SuperBlock {

public:

const static int MAX\_NFREE = 100;

const static int MAX\_NINODE = 100;

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[MAX\_NFREE]; // 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[MAX\_NINODE]; // 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

};

class DirectoryEntry {

public:

static const int DIRSIZ = 28; /\* 目录项中路径部分的最大字符串长度 \*/

public:

int m\_ino; /\* 目录项中INode编号部分 \*/

char name[DIRSIZ]; /\* 目录项中路径名部分 \*/

};

class FileSystem {

public:

// Block块大小

static const int BLOCK\_SIZE = 512;

// 磁盘所有扇区数量

static const int DISK\_SIZE = 16384;

// 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;

// 外存INode区位于磁盘上的起始扇区号

static const int INODE\_ZONE\_START\_SECTOR = 2;

// 磁盘上外存INode区占据的扇区数

static const int INODE\_ZONE\_SIZE = 1022;

// 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\_NUMBER\_PER\_SECTOR = BLOCK\_SIZE / sizeof(DiskINode);

// 文件系统根目录外存INode编号

static const int ROOT\_INODE\_NO = 0;

// 外存INode的总个数

static const int INode\_NUMBERS = INODE\_ZONE\_SIZE\* INODE\_NUMBER\_PER\_SECTOR;

// 数据区的起始扇区号

static const int DATA\_ZONE\_START\_SECTOR = INODE\_ZONE\_START\_SECTOR + INODE\_ZONE\_SIZE;

// 数据区的最后扇区号

static const int DATA\_ZONE\_END\_SECTOR = DISK\_SIZE - 1;

// 数据区占据的扇区数量

static const int DATA\_ZONE\_SIZE = DISK\_SIZE - DATA\_ZONE\_START\_SECTOR;

public:

DeviceDriver\* deviceDriver;

SuperBlock\* superBlock;

BufferManager\* bufferManager;

public:

FileSystem();

~FileSystem();

/\* 格式化SuperBlock \*/

void FormatSuperBlock();

/\* 格式化整个文件系统 \*/

void FormatDevice();

/\* 系统初始化时读入SuperBlock \*/

void LoadSuperBlock();

/\* 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dev上分配一个空闲外存INode，一般用于创建新的文件。\*/

INode\* IAlloc();

/\* 释放编号为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 \*/

Buffer\* Alloc();

/\* 释放存储设备dev上编号为blkno的磁盘块 \*/

void Free(int blkno);

};

### **算法**

* **FileSystem::IAlloc()**

该函数的作用是实现空闲外存索引节点的分配。空闲外存索引节点表使用栈方式管理DiskInode。当需要分配DiskInode时，如果s\_ninode不为0，则将spb->s\_inode[--spb->s\_ninode]所指示的DiskInode分配出去。如果s\_ninode已为0，说明空闲外存索引节点表已不包含任何空闲节点。于是，就要搜索DiskInode区，将找到的空闲好顺次等入s\_inode表中，直至该表已满或已搜索完整个DiskInode区。内核中用于分配空闲外存索引节点的函数是FileSystem::IAlloc()，其流程如下图：



图3.4.3-1 FileSystem::IAlloc()流程图

首先检查超级块索引节点表是否被上锁，如果被上锁表示当前正有其它进程也在访问索引节点表，则需要分配DiskInode的进程进入睡眠状态等待其解锁；如果索引节点表没有被上锁，则当前进程可以进行DiskInode的分配。

进而检查超级块索引节点表是否为空，如果为空的话，则先要通过搜索DiskInode区寻找空闲索引节点，填入超级块的索引节点表中，直至填满超级块索引节点表或找不到空闲索引节点，然后才可进行分配DiskInode的操作。由于此搜索过程需要等待磁盘I/O操作完成，会让当前进程睡眠并调度其它进程上台，因而其前后需要对超级块索引节点表进行上锁和解锁，构成临界区，防止其它进程在此期间操作索引节点表，譬如，其它进程也发现索引节点表为空，需要搜索DiskInode区寻找空闲DiskInode，两个进程同时扫描DiskInode区将会导致同一空闲DiskInode被重复记入超级块索引节点表中。

在从索引节点表“栈顶”获取空闲DiskInode编号之后，需要将其读入内存并验证该索引节点是否真的空闲，如果是则初始化并返回该索引节点；如果不是真的空闲，则需要放弃这个已分配的索引节点，重新从索引节点表“栈顶”获取新的索引节点编号。

* **FileSystem::IFree()**

该函数的作用是实现空闲外存索引节点的释放。相对于DiskInode的分配过程，其释放的流程比较简单。内核中是否一个索引节点的函数是FileSystem::IFree()，其流程如下图：



图3.4.3-2 FileSystem::IFree()流程图

当释放一个DiskInode时，如果超级块索引节点表中空闲DiskInode的个数小于100，则将该索引节点编号记入“栈顶”；只有在以下两种情况下不将释放的索引节点编号记入表中，而是任其散落在磁盘DiskInode区中：

1. 超级块索引节点表被上锁；修改超级块索引节点表需要首先获取该锁，而释放一个DiskInode时让进程为获取该锁而睡眠则代价过高。
2. 当超级块索引节点表中记录的DiskInode已满情况下，让释放的DiskInode散落在DiskInode区中。

* **对空闲盘块的管理**

超级块中的空闲盘块索引表s\_free[100]用栈方式管理空闲盘块，但是最多只能直接管理100个空闲块，它实际管理的块数由s\_nfree表示。若空闲块数超过100，采用分组链式索引法对它们进行管理。

每100个空闲块构成一组(第一组99块)，余下部分也构成一组。最后一组直接由超级块中的空闲块索引表s\_free[100]管理，其余各组的索引表则分别存放在它们下一组第一个盘块的开头404个字节中。例如：若空闲块数349，则第一组包含99块，第二、三组皆为100块，第四组50块。所以s\_nfree为50，索引表s\_free也只用0~49各项。第一组的索引表存放在第二组第一块的404个字节中，其位置和格式与s\_nfree、s\_free相同； 其中，第二个字的值为0，它是空闲盘块链的结束标志，表示下面不再有空闲块可供使用，其余各组的情况与此类似。

**s\_nfree:50**

**s\_free[0]**

.

.

s\_free[49]

**s\_nfree:100**

**s\_free[0]**

**.**

**..**

**s\_free[99]**

**s\_nfree:100**

**s\_free[0]**

**..**

**.**

**s\_free[99]**

**s\_nfree:100**

**s\_free[0]**

**..**

**.**

**s\_free[99]**

SuperBlock

第一组空闲块

(共99块)

**。。。**

**。。。**

**。。。**

**。。。**

**。。。**

**。。。**

第四组空闲块

(共50块)

第三组空闲块

(共100块)

第二组空闲块

(共100块)

**。。。**

图3.4.3-3 空闲盘块分组链式索引

* **FileSystem::Alloc()**

该函数的作用是实现空闲盘块的分配。分配空闲盘块总是从索引表中取其最后一项的值，即spb->s\_free[--spb->s\_nfree];，相当于出栈。当发现这是直接管理的最后一个空闲块被分配后(s\_nfree减1后为0)，就将该盘块的头404字读入超级块的s\_nfree和索引表s\_free[100]中，使得用间接方式管理的下一组变为直接管理。如此类推直至最后一组。当系统中最后一个空闲块被分配使用后，s\_nfree的值为1。当再次企图分配盘块时，发现spb->s\_free[--spb->s\_nfree]的值为0，说明已到空闲盘块链尾，没有盘块可供分配。由此可见，虽然通常s\_nfree解释为该时刻直接管理的空闲盘块数，但其更一般的意义是s\_free表中有意义的剩余项数。例如：s\_nfree为50时，说明s\_free[0]~ s\_free[49]的内容是有意义的。所谓有意义既包括指示一个空闲块，也可以是空闲盘块链结尾标志。

内核中用于分配空闲盘块的函数是FileSystem::Alloc()，其流程如下图：



图3.4.3-4 FileSystem::Alloc()流程图

在开始正式分配盘块编号之前，先要获取空闲块索引表的锁，如果空闲磁盘块索引表正在被上锁，表明有其它进程正在操作空闲块索引表，当前进程需要进入睡眠等待其解锁。

检查通过之后，从索引表“栈顶”得到空闲盘块编号，并且检查取出编号之后索引表是否变为空，如果为空的话表明当前空闲块是超级块目前直接管理的最后一个空闲块，那也就意味着该盘块上记录着下一组空闲盘块编号，因此在使用该盘块之前，先要将它上面记录的下一组空闲盘块编号读入到超级块直接管理的空闲块索引表中，然后才可将盘块分配作它用；如果该盘块不是栈中的最后一块，也就是说盘块上没有包含任何重要信息，因此可以直接清空后移作它用，设置超级修改标志标明SuperBlock内存副本已改动，需要在以后更新到磁盘上，随后返回新分配空闲盘块对应的缓存。

* **FileSystem::Free()：**

该函数的作用是实现空闲盘块的释放。释放空闲盘块时，将盘块号记入空闲盘块索引表中第一个未被占用的项spb->s\_free[spb->s\_nfree++]，相当于压栈。例如若s\_nfree原先只为50，则将释放块号登入s\_free[50]，然后s\_nfree加1为51。如果在记入索引表之前发现索引表已经占满，则应将当前超级块直接管理的100个空闲盘块编号写到当前正在释放的盘块上。这样，原先由超级块直接管理的100个空闲块就变成为由释放块间接管理。然后将此释放块块号压入栈底，s\_nfree置为1。内核中用于分配空闲盘块的函数是FileSystem::Free()，其流程如下图：



图3.4.3-5 FileSystem::Free()流程图

由此可见，对空闲盘块的分配和释放类似于栈，是使用后进先出算法。但其管理机构分成两级，一级常驻内存，另一级则驻在各组的第一个盘块上，充分利用空闲盘块来存储空闲盘块编号，节约了文件系统管理数据结构占据的磁盘开销。

## **目录管理模块FileManager**

### **功能**

Filemanager模块，是系统文件操作功能实现模块，主要封装文件系统中对文件处理的操作过程，负责对文件系统访问的具体细节。包括打开文件、创建文件、关闭文件、Seek文件指针、读取文件、写入文件、删除文件等系统功能的实现。

### **数据结构**

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:

/\* 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\* fileSystem;

/\* 对全局对象g\_INodeTable的引用，该对象负责内存INode表的管理 \*/

INodeTable\* 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\* openFileTable;

public:

FileManager();

~FileManager();

/\* 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(enum DirectorySearchMode mode);

/\* 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);

/\* 取消文件 \*/

void UnLink();

/\* 删除文件夹 \*/

void Rmdir();

/\* 获取文件信息 \*/

void Status();

/\* 向父目录的目录文件写入一个目录项 \*/

void WriteDir(INode\* pINode);

/\* 改变文件访问模式 \*/

//void ChMod();

/\* 改变当前工作目录 \*/

void ChDir();

/\* 列出当前INode节点的文件项 \*/

void Ls();

/\* 列出指定目录Inode节点的文件项 \*/

void Ls2();

/\* 显示历史命令 \*/

void History();

};

### **算法**

* **FileManager::Rdwr()：**

Rdwr()函数首先根据 read、write 系统调用的打开文件描述符 fd 参数获取打开文件控制块结构，然后检查本次读、写请求类型是否符合对该文件的打开方式，如果不符合则拒绝执行读、写操作，譬如要对一个以只读方式打开的文件进行写操作，显然不会通过此处的检查；检查通过则继续。

## **系统打开文件管理模块OpenFileManager**

### **功能**

打开文件管理模块，负责对打开文件的管理，为用户打开文件建立数据结构之间的勾连关系，为用户提供直接操作文件的文件描述符接口。每个进程都有一张进程打开文件描述符表，每一个被进程打开的文件都在描述符表中占据一项。不同于内存索引节点表以及系统打开文件这类属于内核全局的数据结构，进程打开文件描述符表是属于每个进程各自所有的，记录该进程的打开文件情况。

### **数据结构**

/\*

\* 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 MAX\_FILES = 100; /\* 打开文件控制块File结构的数量 \*/

/\* 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。

\*/

File m\_File[MAX\_FILES];

OpenFileTable();

~OpenFileTable();

/\* 在系统打开文件表中分配一个空闲的File结构 \*/

File\* FAlloc();

/\* 对打开文件控制块File结构的引用计数count减1，若引用计数count为0，则释放File结构。\*/

void CloseF(File\* pFile);

void Format();

};

/\*

\* 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 \*/

private:

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\* fileSystem; /\* 对全局对象g\_FileSystem的引用 \*/

public:

INodeTable();

~INodeTable();

/\*

\* 根据外存INode编号获取对应INode。如果该INode已经在内存中，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();

void Format();

};

### **算法**

* **OpenFIles::AllocFreeSlot()：**

OpenFiles 类中提供 AllocFreeSlot()成员函数用于在打开文件描述符表中分配一个空闲表项。该函数线性扫描进程打开文件描述符表，寻找 File 指针为 NULL 的空闲项分配，并将该空闲项在 OpenFiles.ProcessOpenFileTable[]数组中的索引作为打开文件描述符 fd，返回给执行Open()系统调用的进程。

* **OpenFileTable::FAlloc()：**

FAlloc()函数进行 File 控制块的分配，分配算法比较简单，线性搜索 OpenFileTable 中的File[NFILE]数组，寻找其中引用计数 f\_count 为 0 的空闲 File 控制块对象，如果分配成功则与进程打开文件描述符表中的对应 File 指针建立勾连关系；否则的话设置出错标志指示 Open系统调用执行失败。

* **OpenFileTable::CloseF()：**

CloseF()函数用于进程关闭文件时释放对 File 控制块的引用。由于 File 控制块可能被多个进程共享，所以 CloseF()函数一般只是递减该 File 控制块的引用计数 f\_count 值，并不释放该 File 控制块 f\_inode 指向的内存索引节点。只有在当前进程为引用该 File 控制块的唯一进程情况下，当前进程释放才将 File 控制块及其指向的内存 Inode 一起释放，即将 f\_count 值设置为 0，表示该 File 控制块为空闲，并调用 IPut()函数将对应内存 Inode 也释放成为空闲内存 Inode。

* **InodeTable::IGet()：**

函数首先检查要读入DiskInode 是否已有内存拷贝，如果有但是该内存Inode 被上锁，表示该 Inode 正在被另一进程使用，于是当前进程对该 Inode 增设 IWANT 标志指明有进程正在等待个 Inode 被解锁，然后当前进程睡眠，直至另一进程使用完这个 Inode将其解锁时会唤醒等待使用该 Inode 的睡眠进程；如果没有上锁或 Inode 已经解锁，则进一步检查该 Inode 是否用于连接子文件系统，如果是则查找子文件系统根目录 Inode 并将其返回；经过以上判断后当前进程已经获得了对该 Inode 的使用权，于是增加其引用计数，并增设 ILOCK 标志对其上锁，以避免其它进程同时访问该 Inode 导致的数据不一致性。

如果内存中还没有相应 DiskInode 的内存拷贝，则尝试在 InodeTable 中分配一空闲 Inode项，分配成功则对其进行适当初始化，使其能够反映该 Inode 来自于哪个外存索引节点，增加内存 Inode 的引用计数并上锁，然后驱动磁盘读入相应 DiskInode 的内容，并返回该内存Inode 指针。如果InodeTable 已满，分配空闲 Inode 失败的话，则直接返回NULL。

* **InodeTable::IPut()：**

函数首先检查当前进程是否为引用该内存 Inode 的唯一进程，如果是则准备释放该内存 Inode，将其上锁，因为在整个释放过程中可能因为磁盘操作而使得该进程睡眠，此时有可能另一个进程会对该内存 Inode 进行操作，这将有可能导致错误。

在释放内存 Inode 过程中考虑是否要将对应的 DiskInode 一并释放掉，通过判断该文件有没有目录路径指向它，如果发现该文件已被用户删除，仅仅因为当前进程正在使用而没有释放其占据的 DiskInode 和数据盘块，那么此时当前进程释放该文件占据的数据盘块，释放其 DiskInode 收归 SuperBlock 直接管理，通常情况下文件没有被用户删除则只需要将内存拷贝更新至对应 DiskInode 即可。

## **用户接口模块Users**

### **功能**

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。

### **数据结构**

class User {

public:

static const int EAX = 0; /\* u.ar0[EAX]；访问现场保护区中EAX寄存器的偏移量 \*/

enum ErrorCode {

U\_NOERROR = 0, /\* No u\_error \*/

//U\_EPERM = 1, /\* Operation not permitted \*/

U\_ENOENT = 2, /\* No such file or directory \*/

//U\_ESRCH = 3, /\* No such process \*/

//U\_EINTR = 4, /\* Interrupted system call \*/

//U\_EIO = 5, /\* I/O u\_error \*/

//U\_ENXIO = 6, /\* No such device or address \*/

//U\_E2BIG = 7, /\* Arg list too long \*/

//U\_ENOEXEC = 8, /\* Exec format u\_error \*/

U\_EBADF = 9, /\* Bad file number \*/

//U\_ECHILD = 10, /\* No child processes \*/

//U\_EAGAIN = 11, /\* Try again \*/

//U\_ENOMEM = 12, /\* Out of memory \*/

U\_EACCES = 13, /\* Permission denied \*/

//U\_EFAULT = 14, /\* Bad address \*/

//U\_ENOTBLK = 15, /\* Block device required \*/

//U\_EBUSY = 16, /\* Device or resource busy \*/

//U\_EEXIST = 17, /\* File exists \*/

//U\_EXDEV = 18, /\* Cross-device link \*/

//U\_ENODEV = 19, /\* No such device \*/

U\_ENOTDIR = 20, /\* Not a directory \*/

//U\_EISDIR = 21, /\* Is a directory \*/

//U\_EINVAL = 22, /\* Invalid argument \*/

U\_ENFILE = 23, /\* File table overflow \*/

U\_EMFILE = 24, /\* Too many open files \*/

//U\_ENOTTY = 25, /\* Not a typewriter(terminal) \*/

//U\_ETXTBSY = 26, /\* Text file busy \*/

U\_EFBIG = 27, /\* File too large \*/

U\_ENOSPC = 28, /\* No space left on device \*/

//U\_ESPIPE = 29, /\* Illegal seek \*/

//U\_EROFS = 30, /\* Read-only file system \*/

//U\_EMLINK = 31, /\* Too many links \*/

//U\_EPIPE = 32, /\* Broken pipe \*/

//U\_ENOSYS = 100,

//EFAULT = 106

};

public:

string username; /\* 使用username作为当前用户标识符，即User\_ID\*/

INode\* cdir; /\* 指向当前目录的Inode指针 \*/

INode\* pdir; /\* 指向父目录的Inode指针 \*/

DirectoryEntry dent; /\* 当前目录的目录项 \*/

char dbuf[DirectoryEntry::DIRSIZ]; /\* 当前路径分量 \*/

string curDirPath; /\* 当前工作目录完整路径 \*/

string dirp; /\* 系统调用参数(一般用于Pathname)的指针 \*/

long arg[5]; /\* 存放当前系统调用参数 \*/

/\* 系统调用相关成员 \*/

unsigned int 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.ar0[R0] \*/

ErrorCode u\_error; /\* 存放错误码 \*/

OpenFiles ofiles; /\* 进程打开文件描述符表对象 \*/

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数参数 \*/

FileManager\* fileManager;

string ls;

string history;

public:

User();

~User();

void Ls();

void Ls(string path);

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);

void History();

void Rmdir(string dirName);

void Stat(string fileName);

//void Pwd();

private:

bool IsError();

void EchoError(enum ErrorCode err);

unsigned int INodeMode(string mode);

int FileMode(string mode);

bool checkPathName(string path);

};

### **算法**

* **User::Read()：**

Read()函数的作用是根据 User中的文件 I/O 参数，包括用户目标区首地址、要读取的字节数以及当前文件读写字节偏移量，通过访问该文件的 Inode 找到对应的数据块，经由高速缓存将进程需要的数据送入用户目标区。其中采取了预读策略，如果发现进程正在顺序读文件，则在读取文件当前的逻辑块时将下一块一并读入，如果预读命中则可以有效减少下次 I/O 操作的开销，提高进程的前进速度。

* **User::Write()：**

对于 Write()函数，其功能是将用户指定内存区中的数据经由高速缓存写入文件，数据写入文件的位置同样取决于User.u\_IOParam中的文件 I/O 参数，整个工作流程与ReadI()函数相似，但仍有以下几方面需注意：(1)文件中的某些字符块可能只需要部分重写，为了保护不需要重写的部分，应该先将该字符块读至缓存，部分改写后再写回；(2) 全部已经改写或新扩充已写满的字符块立即写回相应盘块，仅部分改写的字符块则暂缓，用延迟写方式处理，这样做是考虑到进程可能在稍后会再次读写该字符块，使用延迟写方式可以让该字符块在高速缓存队列中停留较长时间不被分配做它用，从而增加了在高速缓存中直接找到该字符块的概率。

# **程序设计与实现**

## **重要函数程序流程图**

程序的总体流程为：main 函数等待用户输入，用户输入后，对其命令进行简单解析，然后由命令查找 User 提供的操作接口，将参数交由 User 函数进行处理和判断合法性，若基本符合命令的参数约定，则由User调用FileManager中功能函数实现文件系统的具体功能。而 FileManger 中函数的具体功能则依赖于 FileSystem、BufferManager、OpenFileManager 的相关函数接口调用。

### **逻辑块号到物理块号的转换—Bmap()**

进程存取文件中的数据时使用的是字节偏移量，将整个文件看作是从字节0开始直至文件大小的字节流。内核将用户的字节流转换成逻辑块的看法，把文件分割成大小相等的逻辑块，文件从第0个逻辑块开始直至文件大小对应的逻辑块号结束。

对文件进行读、写时，总需要先将文件逻辑号变换成文件在存储设备上的物理块号，从上面关于Unix V6++中文件索引结构的内容可以看出，文件逻辑块与物理块之间的对应关系包含在索引节点中的i\_addr[10]数组中，于是从逻辑块号到物理块号的转换就是对索引节点中基本索引表的操作了。

内核中执行这项转换工作的是内存索引节点Inode类的成员Bmap()函数，其流程大致如下：



图4.1.1-1 函数Inode::Bmap()流程图

Inode::Bmap()函数中进行逻辑块号转换的执行路径大致分为两条，分别是用于对小型文件以及大型或巨型文件。

首先判断文件长度属于小型文件还是大型或巨型文件，如果为小型文件，从基本索引表i\_addr[0 ~ 5]中获得物理盘块号即可。如果该逻辑块号还没有相应的物理盘块号与之对应，即相应索引表项为0，则分配一个物理块。这通常发生在对文件的写入，若写入位置超没有对应的数据内容，即对当前文件进行扩充写入，就需要分配额外的物理盘块存放文件数据，并为之建立逻辑块号与物理盘块号之间的映射。

如果是大型或巨型文件，则通过计算出逻辑块号在基本索引表的对应项，该项记录着间接索引表块的物理块号，对于大型文件为一次间接索引表块，而对于巨型文件则为二次间接索引表块，根据物理块号读入此一次(或二次)间接索引表块，并根据要转换的逻辑块号计算出位于一次间接索引表中的表项下标，从而得到实际文件数据的物理块号。须注意，巨型文件要比大型文件增加一环节，即依据逻辑块号计算得到二次间接索引表中表项的下标，找到并读入一次间接索引表块，然后按照大型文件操作一次间接索引表块方法找到数据盘块。

具体来说，如果需要转换的逻辑块号是0 <= lbn < 6，那么直接返回基本索引表i\_addr[lbn]中相应值；如果逻辑块号在6 <= lbn < 128 \* 2 + 6之间，那么为一次间接索引，通过i\_addr[6]或i\_addr[7]找到一次间接索引表块，然后在这一索引表块中找到对应的物理块号；如果逻辑块号在(128 \* 2 + 6 ) <= lbn < (128 \* 128 \* 2 + 128 \* 2 + 6)之间，那么通过i\_addr[8]或i\_addr[9]找到二次间接索引表块，根据二次间接索引表块找到一次间接索引表块，进而最后找到对应的物理块号。

### **索引节点的获取—IGet()**

每当需要访问一个文件时，都要找到该文件对应的DiskInode，并预先在内存索引节点表中分配一个空闲内存Inode项，然后将DiskInode的信息拷贝至内存Inode中。然而也有可能要访问文件的索引节点已经在内存中，之前强调过无论有多少进程需要访问一个DiskInode，它都不应该被读入内存两次，也就是说最多只能有一个内存Inode副本，因此在索引节点的获取过程中需要将这种情况考虑进去。

Unix V6++中使用InodeTable::IGet(dev, inumber)函数来获得指定DiskInode的内存拷贝，其接受设备号dev和外存索引节点编号inumber来定位DiskInode。其大致流程如下图：



图4.1.2-1 函数InodeTable::IGet(dev, inumber)流程图

InodeTable::IGet()函数首先检查要读入DiskInode是否已有内存拷贝，如果有但是该内存Inode被上锁，表示该Inode正在被另一进程使用，于是当前进程对该Inode增设IWANT标志指明有进程正在等待个Inode被解锁，然后当前进程睡眠，直至另一进程使用完这个Inode将其解锁时会唤醒等待使用该Inode的睡眠进程；如果没有上锁或Inode已经解锁，则进一步检查该Inode是否用于连接子文件系统，如果是则查找子文件系统根目录Inode并将其返回；经过以上判断后当前进程已经获得了对该Inode的使用权，于是增加其引用计数，并增设ILOCK标志对其上锁，以避免其它进程同时访问该Inode导致的数据不一致性。

如果内存中还没有相应DiskInode的内存拷贝，则尝试在InodeTable中分配一空闲Inode项，分配成功则对其进行适当初始化，使其能够反映该Inode来自于哪个外存索引节点，增加内存Inode的引用计数并上锁，然后驱动磁盘读入相应DiskInode的内容，并返回该内存Inode指针。如果InodeTable已满，分配空闲Inode失败的话，则直接返回NULL。

需要注意，进程并没有在这里睡眠等待直至内核中有空闲的Inode项可供分配，因为系统中内存Inode的分配和释放是由用户程序调用Open和Close系统调用间接控制的，内核无法保证在较短的时间内系统中会有空闲Inode可用，一个因为等待空闲Inode项而睡眠的进程可能一直无法醒来，相当于进入“假死”状态，所以直接返回NULL表示系统调用在申请空闲Inode项环节失败，由用户程序来决定下一步该怎么走。

### **索引节点的释放—IPut()**

当进程关闭一个文件时，需要释放该文件对应的索引节点。由于同一文件可能同时被多个进程打开，这些进程共享同一个内存Inode，即这个内存Inode可能被多个File控制块指向，因此IPut()函数通常只是递减该Inode的引用计数值，并不真正释放内存索引节点。

Unix V6++中使用InodeTable::IPut()函数来释放当前进程对该内存Inode的引用，其大致流程如下图：



图4.1.3-1 函数InodeTable::IPut()流程图

InodeTable::IPut()函数首先检查当前进程是否为引用该内存Inode的唯一进程，如果是则准备释放该内存Inode，将其上锁，因为在整个释放过程中可能因为磁盘操作而使得该进程睡眠，此时有可能另一个进程会对该内存Inode进行操作，这将有可能导致错误。

在释放内存Inode过程中考虑是否要将对应的DiskInode一并释放掉，通过判断该文件有没有目录路径指向它，如果发现该文件已被用户删除，仅仅因为当前进程正在使用而没有释放其占据的DiskInode和数据盘块，那么此时当前进程释放该文件占据的数据盘块，释放其DiskInode收归SuperBlock直接管理，通常情况下文件没有被用户删除则只需要将内存拷贝更新至对应DiskInode即可。

然后解锁内存Inode，唤醒因等待该Inode解锁而进入睡眠的进程，之所以前面已经判断过当前进程是引用该内存Inode的唯一进程，而这里又要唤醒潜在可能等待该Inode的睡眠进程，这是因为在将内存Inode更新到对应DiskInode过程中，当前进程可能因为磁盘操作而睡眠，在此期间系统中对此Inode的需求可能发生变化，譬如有新的进程打算使用该Inode发现已被上锁，因此需要在此处唤醒潜在可能的睡眠等待进程。随后则可以清空该Inode的标志位将其转为空闲Inode项。

而如果当前释放进程不是引用该Inode的唯一进程，释放过程就相当简单了，只需要递减该Inode的引用计数，然后解锁Inode，唤醒等待使用该Inode的睡眠进程。

### **目录的搜索—NameI()**

目录的搜索过程实际上是将路径名到索引节点转换的过程，用户程序对文件的存取是通过路径名的，而内核中使用的是索引节点号而不是路径名字符串，所以就需要路径名转换为索引节点号。

Unix V6++中执行目录搜索的是FileManager::NameI()函数，它负责将用户给出的路径名转换为对应文件的索引节点，其大致流程如下图：



图4.1.4-1 函数FileManager::NameI()流程图

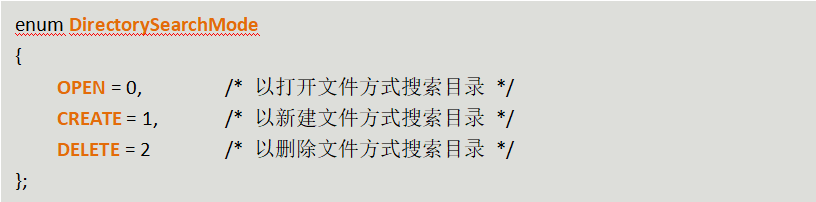
每个进程都与一个当前工作目录相联系，进程的User中包含一个指向当前目录内存Inode 的指针。在NameI()函数进行目录搜索的过程一般都是从该进程的当前工作目录开始的，除非路径名是以'/'开始，则表示要从根目录开始搜索。无论是从当前工作目录或是从根目录开始搜索，NameI()函数都可以直接从User中或是FileManager.rootDirInode中找到相应的内存Inode。

从当前工作目录或根目录的内存Inode起始，逐级搜索路径分量对应的下一级目录(或文件)索引节点号。在NameI()函数每次循环期间，如果当前路径分量不是路径中的最后一个分量，都要验证该路径分量确实是目录，否则就违反了非目录文件只能作为目录树中的叶子节点的规定。

在每一级搜索当前工作目录过程中，根据用户给定的路径名中每一个路径分量，在相应的目录文件中目录项逐个匹配。由于目录文件的目录项数据在磁盘上，所以首先需要通过目录文件的索引节点，从目录文件的第0块逻辑块开始文件末尾，依次将每一块的逻辑块号转换为对应的物理块号并读入缓存，然后将当前路径分量与缓存中的目录项数据比较，如果匹配成功，则意味着完成了当前一级上目录的搜索，记录下一级路径分量的DiskInode号，然后释放读入目录项数据的缓存以及当前工作目录的索引节点，获取下一级目录(或文件)DiskInode的内存副本，为后面下一级目录搜索做好准备。

如果已经比较完该目录文件的所有目录项仍然没有找到匹配项，则记录下目录文件中的首个空闲目录项的偏移量位置，用于为新创建文件分配目录项做准备。因为匹配失败的情况对于打开、删除文件时的路径搜索意味着查找失败，而对于创建文件则恰恰需要这种匹配失败的发生。

NameI()函数接受一个enum DirectorySearchMode枚举类型的传入参数，该参数用于指定目录搜索模式，其定义如下图：



代码4.1.4-1 目录搜索模式DirectorySearchMode

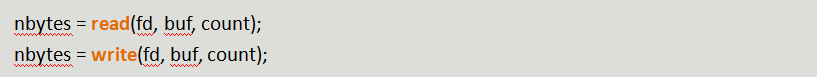
DirectorySearchMode.OPEN表示要根据给出的文件路径名，在目录结构中找到相应的目录项。打开文件时进行的目录搜索属于这一类。DirectorySearchMode.CREATE表示在目录结构中要形成一新目录项。创建文件时的目录搜索属于这一类。DirectorySearchMode.DELETE表示应删除给定路径名的相应目录项。用于删除文件时进行的目录搜索。

不论对于哪种请求，NameI()函数都有执行成功或失败两种可能。如果失败，则将出错码送入User.u\_error中，函数返回NULL。如果执行成功，则对于不同的目录搜索模式具有以下不同含义：DirectorySearchMode.OPEN返回搜索到的目录或文件的Inode指针；DirectorySearchMode.CREATE返回NULL的同时将当前工作目录Inode指针保存在User.u\_pdir中，并记下空闲目录项的偏移量，用于随后写入目录项内容。DirectorySearchMode.DELETE返回要删除目录项所在目录文件的内存Inode指针，同时记下要删除目录项的偏移量，为随后清空该目录项做好准备。

### **文件读、写公共过程—Rdwr()**

在Unix V6++中，文件系统的系统调用在内核中的处理函数都包含在了FileManager类中，由该类向上对用户程序提供文件的打开、关闭以及文件读写的功能，同时向下管理着内存中打开文件管理机构和文件系统存储资源等，可以说FileManager类起着衔接用户对文件的操作以及对内核数据结构相应处理的作用，这里要解释的关于文件的读、写操作过程相关函数基本都是来自于FileManager类中。

系统调用read和write的格式一般如下图：



代码4.1.5-1 read、write系统调用格式

其中，fd是打开文件描述符编号，buf是用户进程中目标数据区内存首地址，count是要读、写的字节数，返回值nbytes表示实际读、写的字节数。

read和write系统调用都会执行内核中的文件读、写公共函数FileManager::Rdwr()，只是使用不同参数表明要进行的读、写操作类型，read系统调用传递给Rdwr()的参数为File::FREAD，而write系统调用传递的参数为File::FWRITE。Rdwr()的大致流程如下图：



图4.1.5-1 函数FileManager::Rdwr()流程图

Rdwr()函数首先根据read、write系统调用的打开文件描述符fd参数获取打开文件控制块结构，然后检查本次读、写请求类型是否符合对该文件的打开方式，如果不符合则拒绝执行读、写操作，譬如要对一个以只读方式打开的文件进行写操作，显然不会通过此处的检查；检查通过则继续。

随后将read、write系统调用的参数如目标数据区首地址、要求读/写的字节数送入进程User中的文件I/O参数成员User.u\_IOParam。判断本次读、写是针对普通文件还是管道文件，然后分别调用相应的读、写函数，就普通文件而言，最后调用到的实际文件读、写函数分别是Inode::ReadI()和Inode::WriteI()，由它们具体负责经由高速缓存读取或写入文件数据。最后Rdwr()函数根据读、写字节数，修改File控制块中文件读写字节偏移量，并设置系统调用返回值为实际读、写的字节数。

### **文件读、写函数—ReadI() & WriteI()**

内核中实际进行文件读、写操作的函数分别是Inode::ReadI()和Inode::WriteI()，由它们具体负责经由高速缓存读取或写入文件数据。它们的流程图如下两图：

图4.1.6-1 函数ReadI() 流程图 图4.1.6-2 函数WriteI()流程图

ReadI()函数的作用是根据User.u\_IOParam中的文件I/O参数，包括用户目标区首地址、要读取的字节数以及当前文件读写字节偏移量，通过访问该文件的Inode找到对应的数据块，经由高速缓存将进程需要的数据送入用户目标区。其中采取了预读策略，如果发现进程正在顺序读文件，则在读取文件当前的逻辑块时将下一块一并读入，如果预读命中则可以有效减少下次I/O操作的开销，提高进程的前进速度。

对于WriteI()函数，其功能是将用户指定内存区中的数据经由高速缓存写入文件，数据写入文件的位置同样取决于User.u\_IOParam中的文件I/O参数，整个工作流程与ReadI()函数相似，但仍有以下几方面需注意：(1)文件中的某些字符块可能只需要部分重写，为了保护不需要重写的部分，应该先将该字符块读至缓存，部分改写后再写回；(2) 全部已经改写或新扩充已写满的字符块立即写回相应盘块，仅部分改写的字符块则暂缓，用延迟写方式处理，这样做是考虑到进程可能在稍后会再次读写该字符块，使用延迟写方式可以让该字符块在高速缓存队列中停留较长时间不被分配做它用，从而增加了在高速缓存中直接找到该字符块的概率。

## **程序说明及实验结果**

### **登录功能**

1. 用户通过用户名和密码登录(login)的实验结果如下图所示：

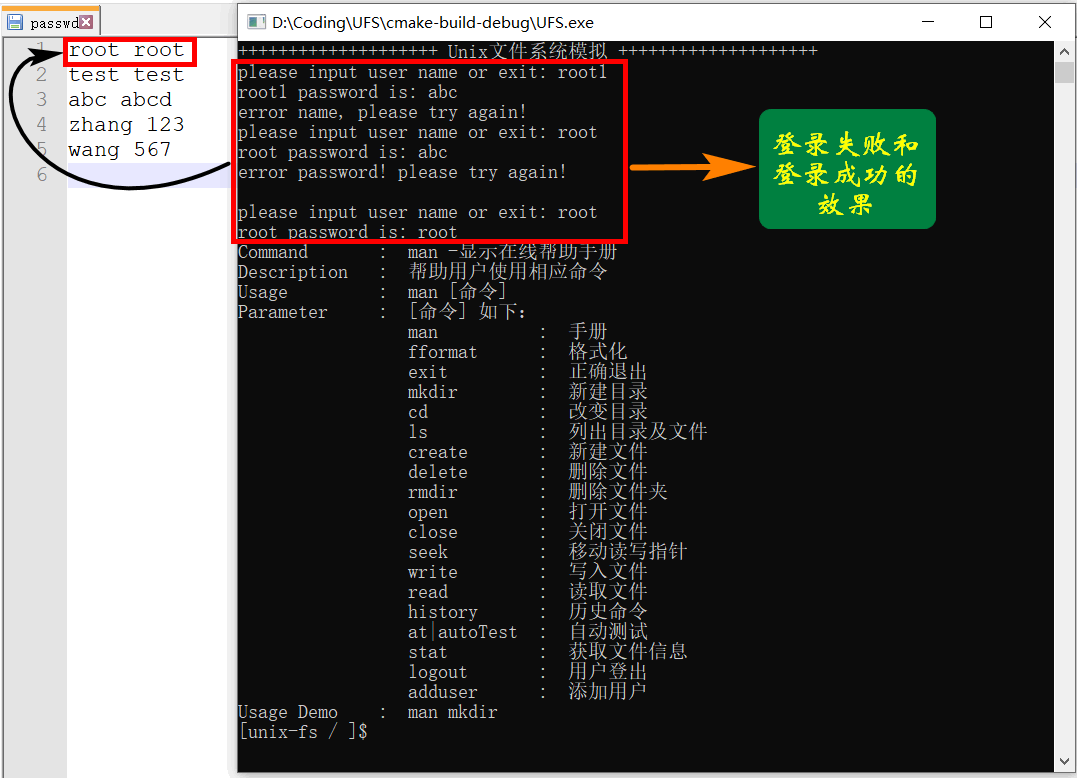
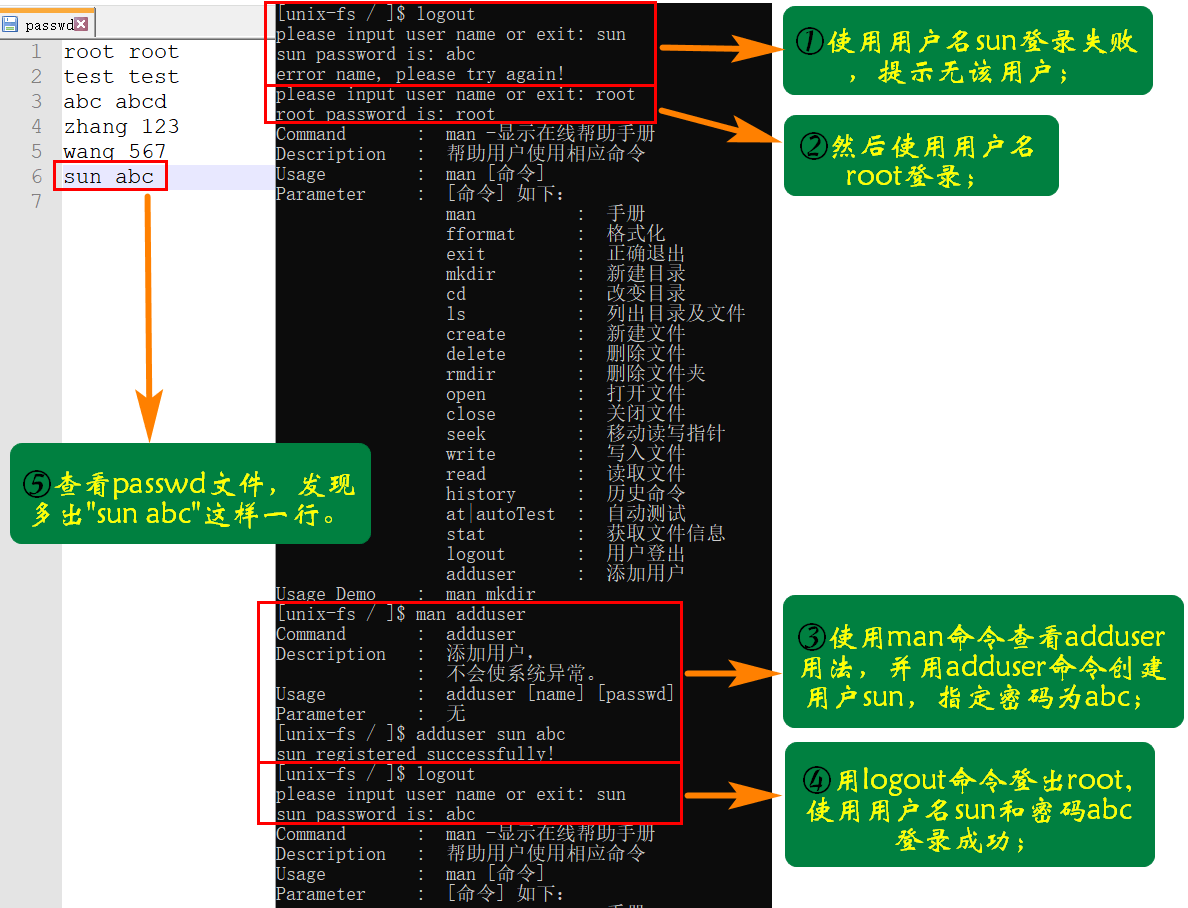


图4.2.1-1 login

1. 登出(logout)和用户注册(adduser)的实验结果如图所示：

图4.2.1-2 logout & adduser

### **列出目录及文件信息**

1. 列出指定目录下的文件列表(ls)的实验结果如图：

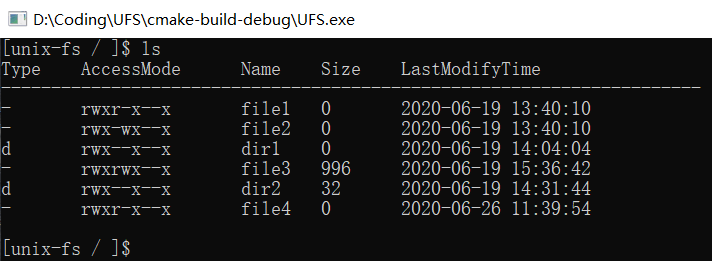


图4.2.2-1 ls

1. 列出指定文件(夹)信息(stat)的实验结果如下图：

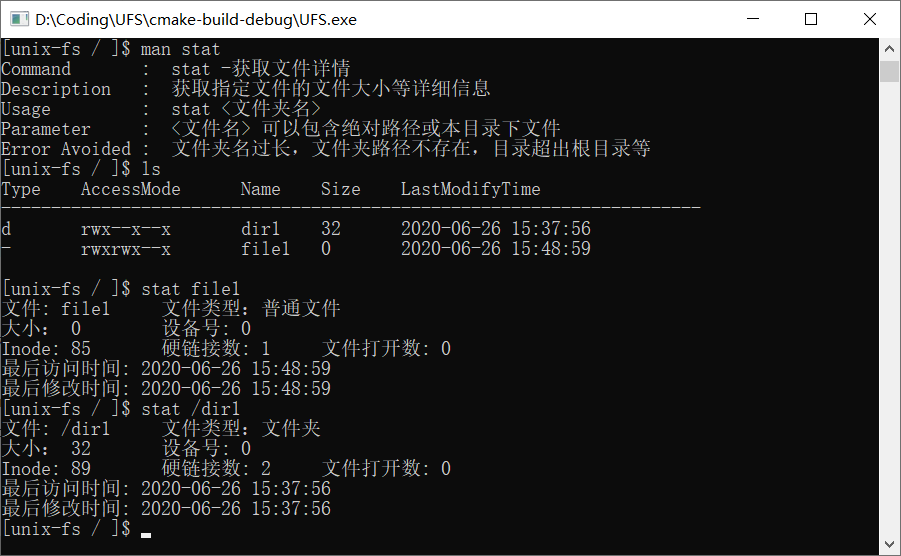


图4.2.2-2 stat

### **列出历史命令**

1. 列出历史命令(history)的实验结果如下图4.2.3-1所示：

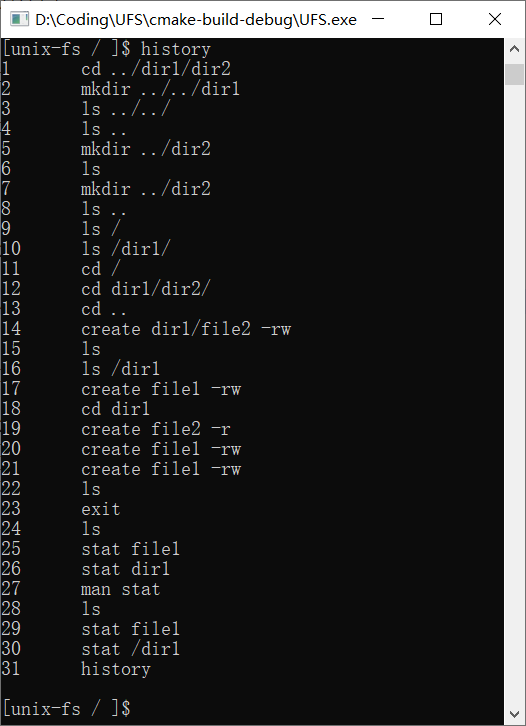


图4.2.3-1 history

### **切换当前目录**

1. 切换当前目录(cd)的实验结果如图所示：

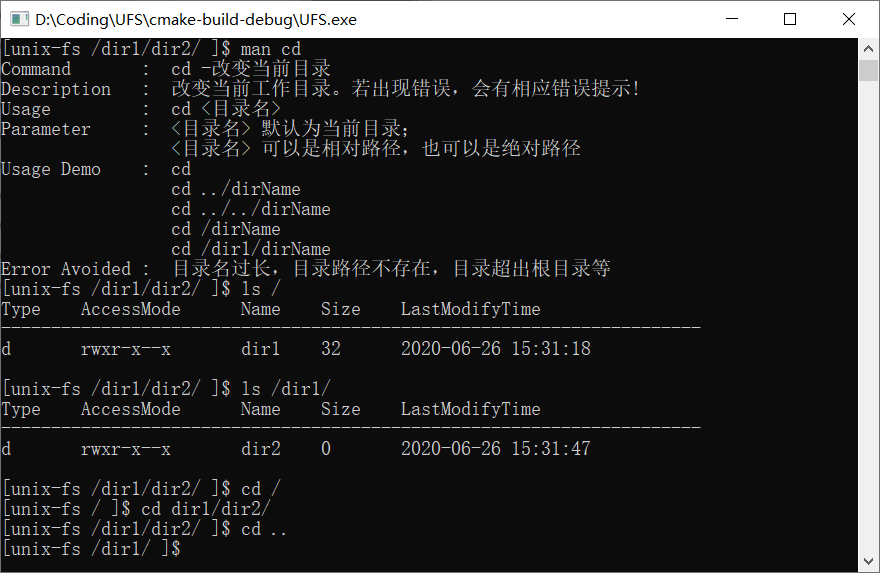


图4.2.4-1 cd

### **文件操作**

1. 文件创建(create)和删除(delete)的实验结果如下图所示：

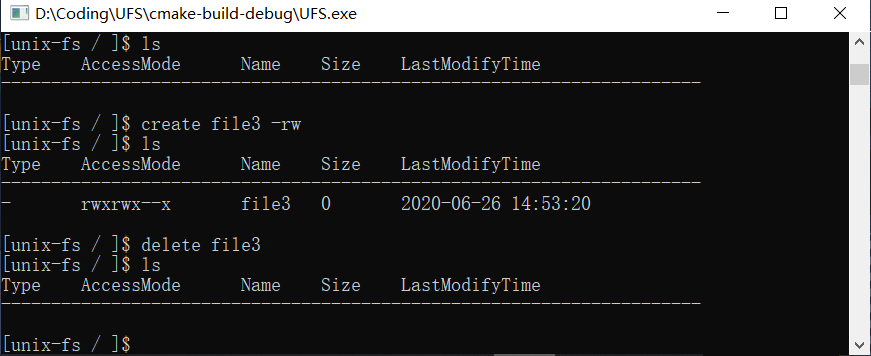


图4.2.5-1 create & delete

1. 文件目录创建(mkdir)和删除(rmdir)的实验结果如图：

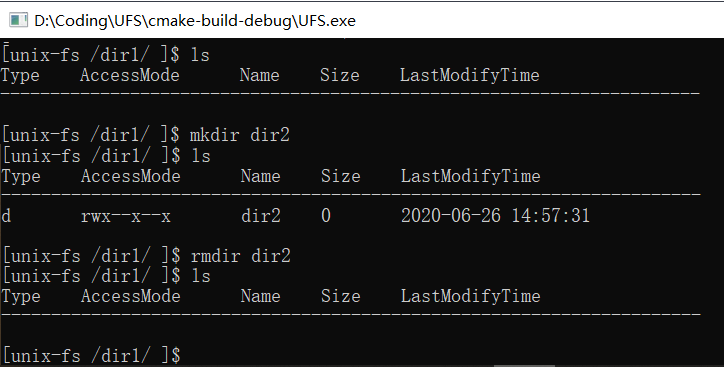


图4.2.5-2 mkdir & rmdir

1. 文件打开(open)和关闭(close)的实验结果如图4.2.5-3所示：

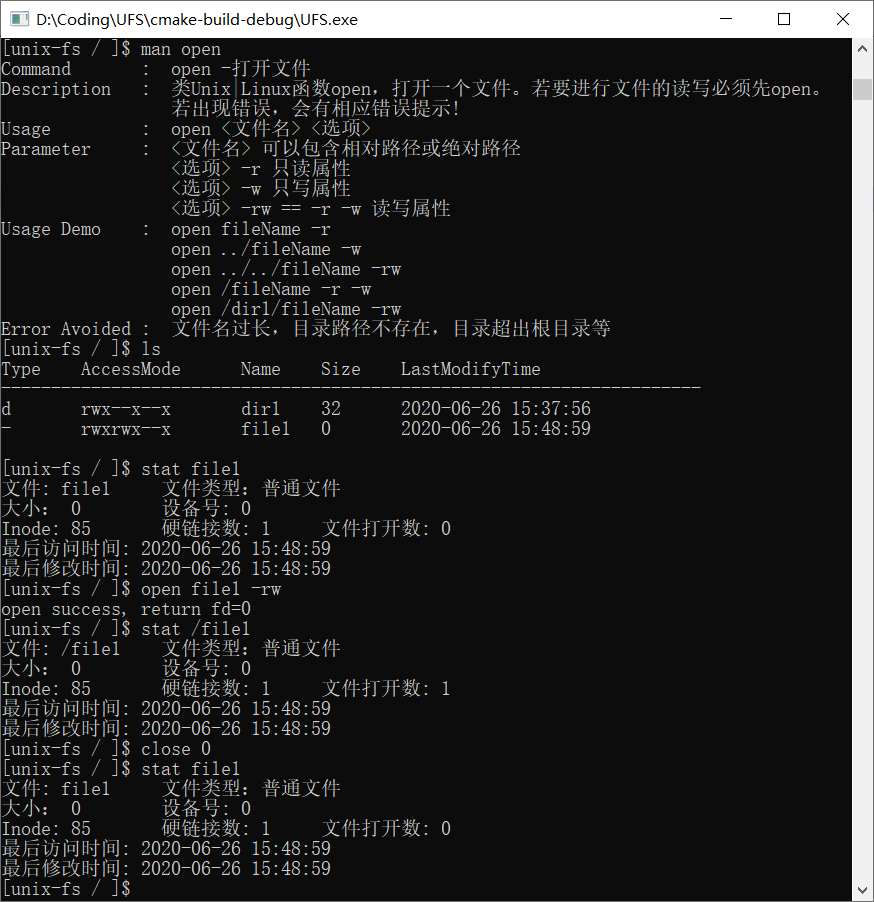


图4.2.5-3 open & close

1. 文件读(read)、写(write)和读写指针移动(seek)的实验结果如图4.2.5-4、4.2.5-5和4.2.5-6所示：

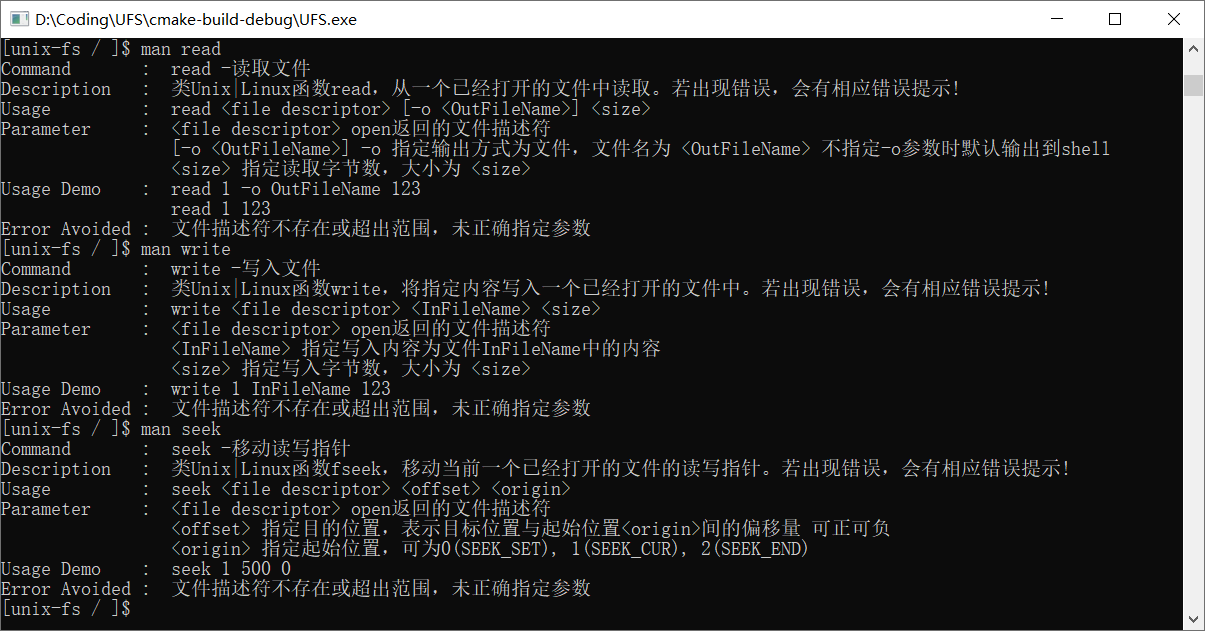


图4.2.5-4 read、write、close命令说明

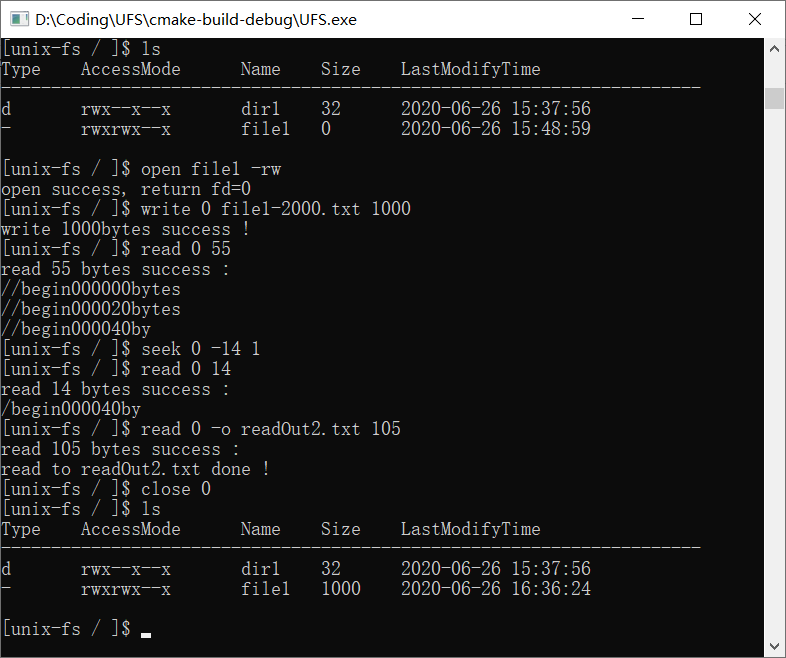


图4.2.5-5 mkdir & rmdir

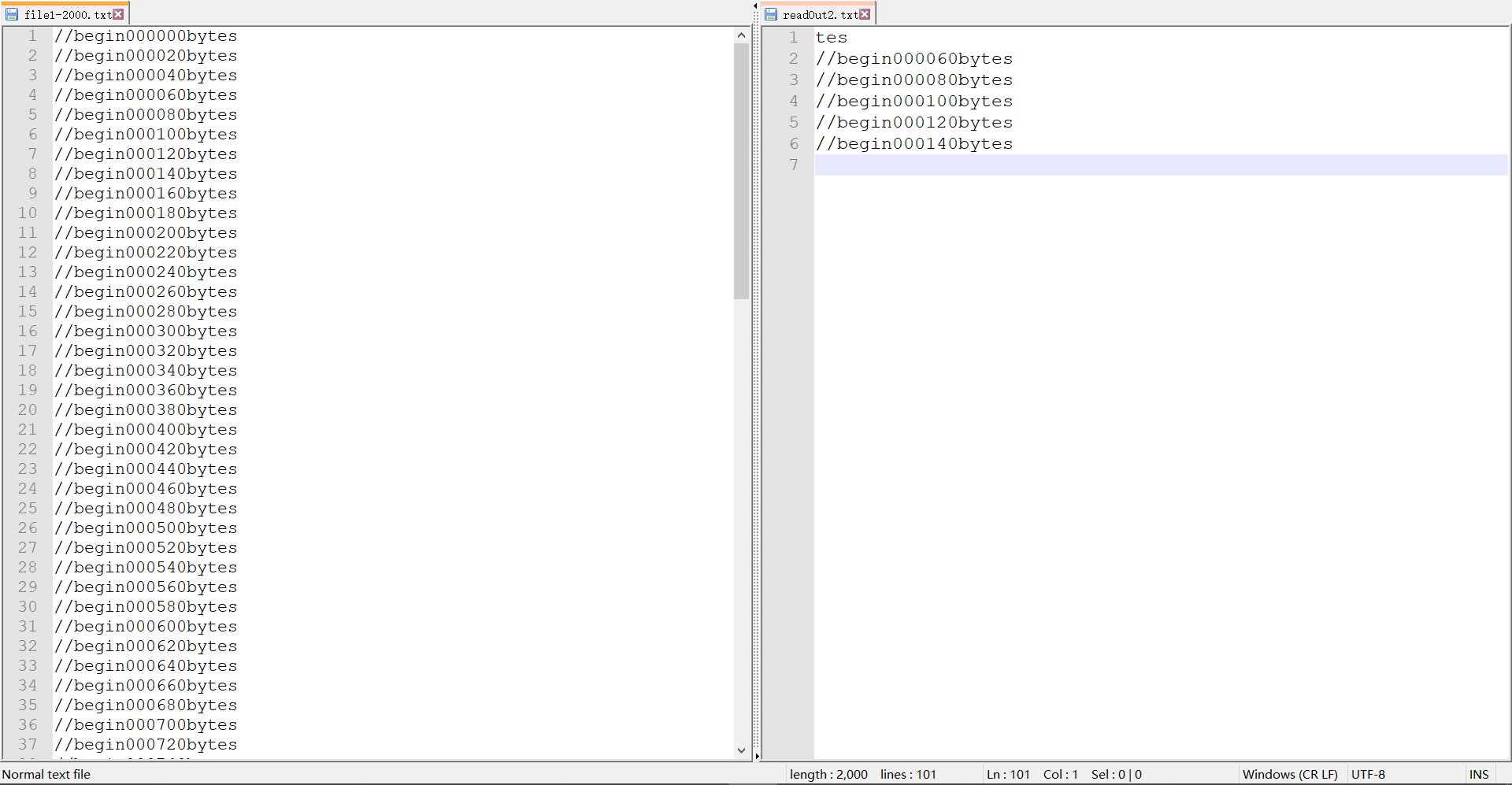


图4.2.5-6 文件读写演示

### **文件系统格式化**

1. 文件系统格式化(fformat)的实验结果如图4.2.6-1所示：

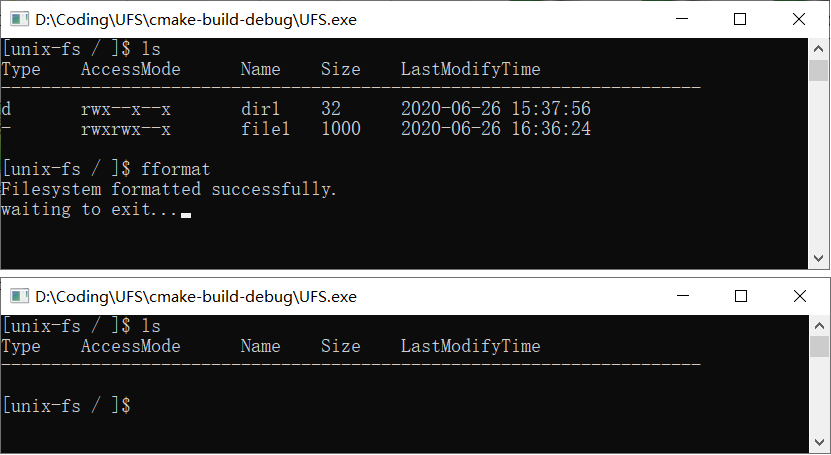


图4.3.6-1 format

# **参考文献**

1. 尤晋元,UNIX 操作系统教程[M],西安:电子科技大学出版社,1985 年 6 月:
2. John Lions,莱昂氏 UNIX 源代码分析[M],机械工业出版社,2006 年 8 月:
3. 汤小丹、梁红兵等,计算机操作系统[M], 西安:电子科技大学出版社,2013 年 11 月:
4. Unix v6++源码

# **结论、收获、体会和建议**

在整个大学期间我们做过很多的课程设计，但操作系统可能是我们花费时间精力最多的课程设计了。付出就有回报，通过本次操作系统课程设计，我觉得无论是系统分析和设计架构的能力，编程语言C++的运用和理解，还是寻找项目代码 bug 的能力，分析错误调试程序的能力都有较大的提升。

要想使项目的整体规划和模块划分具有较好的合理性，做好需求分析是非常重要的。如果对系统功能要求不够明确，那么便不能很好的开展后续的工作。需求分析脚踏实地了，才能整个项目有一个较好的全局认识，然后在此基础上进行系统架构的设计和模块的划分。具体就是确定好模块的大致功能，模块之间的相互调用关系，每个功能模块类的定义，函数接口的定义，做好这些准备工作，才能一步一步稳扎稳打的实际编程。

C++语言是建立在 C 语言之上的一门面向对象的语言，虽然从计算机入门到现在个人技术栈一直都是以 C/C++语言为主，语言的语法知识应该比较熟悉了，但是仍从本次课程设计学到不少知识。例如类的静态成员数据是不占用类的大小的，每一个类的占用空间大小是有规则定义，可能同样的成员，定义顺序不一样占用空间的大小是不一样的；不同编译器对 C++语法编译规则是有细微差别的，不同操作系统运行一样的标准 C++语法代码也是有着一定区别。

本次课程设计时间跨度比较大，本来准备先写好操作系统课程设计，写到一半又被其它课程设计和作业无情中断，导致前前后后费时较多。某些模块的小问题在这种来回切换的模式下被遗留忘记，导致后面整体测试时大费周章的寻找错误。不过这样也有好处，不知不觉中提高了自己调试代码的能力，进一步的提升了如何分析错误的原因，快速准确定位 bug所在。

Unix 系统是现代各类 Unix 操作系统的源头，Unix v6 是非常优秀的系统程序，结构清晰，短小精悍，Unix v6++又用面向对象的思想对其重新进行了设计，并用 C++加以实现。

本次课程设计就是仿照 Unix v6++中的文件系统进行改造而来，在这过程中，不免需要通透了解 Unix v6++的设计思想，详细阅读 Unix v6++中的源码，到现在可以说对其文件系统这一块的代码较为熟悉。但是 Unix v6++中的文件系统仅仅是其中的一个子部分，还有进程管理、内存管理、设备管理等模块，对整个 Unix 操作系统的进一步认识还有很长的路程要走。