

**本科实验报告**

课 程 名 称： 操作系统原理实验

课 程 编 号： 08060029

学 生 姓 名： 江泽凯

学 号： 2019050024

学 院： 信息科学技术学院

系： 计算机科学系

专 业： 计算机科学与技术

指 导 教 师： 刘波

教 师 单 位： 信息学院计算机系

开 课 时 间：2020~ 2021 学年度第 二学期

**暨南大学教务处**

**20 年 月 日**

操作系统原理 **课程实验项目目录**

学生姓名： 学号：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 实验项目编号 | 实验项目名称 | \*实验项目类型 | 成绩 | 指导教师 |
| 1 | 0806002901 | 操作系统用户接口实验 | 验证性 |  | 刘波 龚家政 |
| 2 | 0806002902 | 进程的控制 | 设计性 |  | 刘波 龚家政 |
| 3 | 0806002903 | 进程间的通讯 | 设计性 |  | 刘波 龚家政 |
| 4 | 0806002904 | 生产者消费者问题 | 综合性 |  | 刘波 龚家政 |
| 5 | 0806002905 | 简单文件系统实现 | 综合性 |  | 刘波 龚家政 |
| 6 |  |  |  |  |  |
| 7 |  |  |  |  |  |
| 8 |  |  |  |  |  |
| 9 |  |  |  |  |  |
| 10 |  |  |  |  |  |
| 11 |  |  |  |  |  |
| 12 |  |  |  |  |  |
| 13 |  |  |  |  |  |
| 14 |  |  |  |  |  |
| 15 |  |  |  |  |  |
| 16 |  |  |  |  |  |
| 17 |  |  |  |  |  |
| 18 |  |  |  |  |  |

\*实验项目类型：演示性、验证性、综合性、设计性实验。

\*此表由学生按顺序填写。

**暨南大学本科实验报告专用纸**

课程名称 操作系统原理实验 成绩评定

实验项目名称 二级文件管理系统 指导教师 刘波，龚家政

实验项目编号 0806002901 实验项目类型 综合性 实验地点 N116

学生姓名 江泽凯 学号 2019050024

学院 信息科学技术学院 系 计算机科学系 专业 计算机科学与技术

实验时间2021年6月11日 上午～ 6 月11日 上午 温度 ℃湿度

1. 实验目的和要求

文件管理是操作系统的一个非常重要的组成部分。 学生应独立用高级语言编写和调试一个简单的文件系统，模拟文件管理的工作过程。从而对各种文件操作命令的实质内容和执行过程有比较深入的了解，掌握它们的实施方法，加深理解课堂上讲授过的知识。

1）掌握文件系统的工作原理。

2）理解文件系统的主要数据结构。

3）加深理解文件系统的内部功能和实现方法。

1. 实验原理和主要内容
2. 设计并实现一个多用户的二级文件系统程序，要求提供以下操作。

①文件创建/删除命令： create/ delete  
②目录创建/删除命令： mkdir/rmdir  
③显示目录内容命令：ls

④文件打开/关闭open/ close

⑤文件读/写 read/write

⑥切换当前目录 cd  
拓展：①显示目录内容时要列出文件名和文件长度等信息。

②创建文件，查找空闲块时，不一定使用first\_fit算法，可以使用其他的磁盘调度算法。（如：SCAN、最短寻找时间优先算法等）

1. **实验指导**

可以在内存中开辟一个虚拟磁盘空间作为文件存储器，并将该虚拟文件系统以文件的形式保存到磁盘上，以便下次可以将其恢复到内存的虚拟磁盘空间中。对模拟文件系统的操作是对该文件的读写，创建磁盘即为创建该文件。  
 文件系统以块为基本分配单位，首先需要设置每个块( BLOCK\_SIZE)的大小（如512字  
节）。最开始的第0块为引导块；然后是FAT表，可以分配多个块存储FAT表；接下来就是根目录了。格式化时，根目录中需要增加两个目录项“.”和“..”,分别表示当前目录和上层目录。

需要使用的数据结构包括FAT表、文件控制块、文件打开表等，可参考以下内容进行设计。

①FAT表项中的内容表示文件的下一个盘块号，如果当前块为文件的最后一个盘块（即没有下一个盘块），则相应的表项内容可设置为0xFF( END\_OF\_FILE),表示文件结束。如果FAT表项的内容为0( NOT\_USED),则表示该表项对应的块未被使用。  
 ②每个文件都有一个文件控制块，需要包含文件名、文件属性、文件所在的第一个块的块号、文件大小等属性。  
 ③文件打开表用于记录文件访问的动态变化过程，应包含打开的文件指针、打开文件的权限、文件指针所在的块、文件指针所在块内偏移等信息。

系统中最主要的操作的实现方法可参考以下各项。

(1)文件系统格式化（format）

以读/写方式建立一个新的文件，返回一个文件指针。后续程序可以通过该文件指针对文进行相关的读操作。格式化过程中，文件系统读入内存，只需将位示图中的所有内容置为0,并在根目录中填入“.”和“..”项的内容即可。

(2)创建文件( create).

创建文件时，首先要判断该文件是否已存在，如不存在，再创建文件。创建文件时，首先査找该文件，如果找到表示文件存在，否则为该文件分配空闲的文件控制块，并填写相应的属性等信息。如果创建的是目录文件，则需要为其申请新的一块，并在块中填入“.”和“..”项的内容。由此可见，创建一个普通文件和创建一个目录文件是不同的，普通文件的初始大小为0,初始块号为 END\_OF\_FILE,而目录文件初始化时是有大小的。  
(3)删除文件(rm).  
 与创建文件 create操作类似。  
(4)获取文件控制块( get\_fcb).

这个操作主要根据路径找到它对应的文件的文件控制块。根据路径参数首先找到目录名，从当前查找目录找到该目录，如果找到，则把该目录设为当前查找目录，继续递归查找，直到找到路径的最后一项；如果没找到，则返回错误。

(5)打开文件(open)

首先检查文件是否已经打开。如果没有打开，则由文件对应的路径，利用 get\_fcb操作找到需打开的文件的文件控制块，并加到文件打开表中。文件打开表中的文件指针所在的块及块内偏移，表示当前对打开的文件的操作已经处于该文件所占磁盘空间的某个块的某个字节位置。文件刚打开时初值为0。

(6)关闭文件( close).  
与打开文件open操作类似

(7)文件读操作(read).  
 首先判断文件是否已经打开。若文件已打开，则由 get\_fcb操作找到相应的文件控制块，开始递归读取文件中的内容。读完之后，需要修改文件打开表中相应指针的值。如果读取的字节数超过文件最大长度，则只能读到文件结尾为止。

(8)文件写操作( write)

与文件读操作read类似。需注意：新创建的普通文件的大小为0,此时对该文件进行写操作时，必须先申请一个空闲块，再进行递归的写操作。

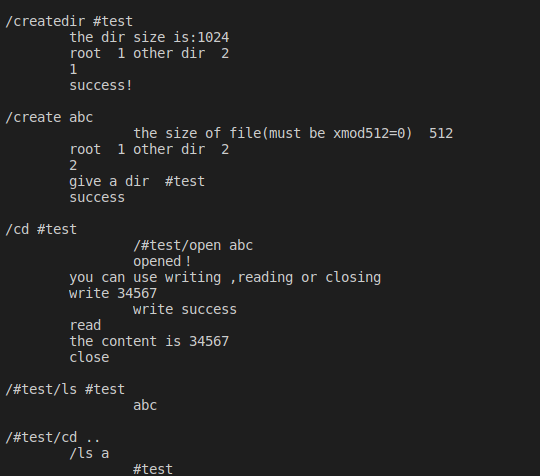
(9)显示当前目录下的文件和目录(1s)

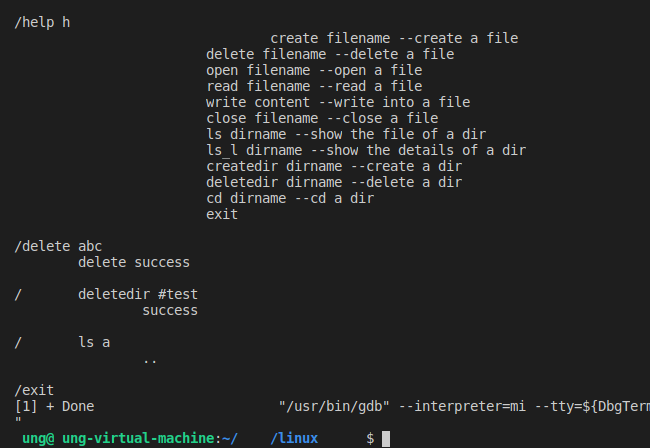
遍历当前目录下所有的目录项，操作方法与 get\_fcb类似也可参考 Unix/Linux系统中的多级索引方式来设计文件的物理结构，请自行完成相关设计  
内容。

实验提示

这是本学期所有实验中最有难度的一个实验，涵盖了文件系统层次结构、磁盘管理、文件系统接口等内容，既可以模仿 Linux的EXT2文件系统，也可以参照 Windows的FAT方式，或者两者综合取长补短。补充资料简要介绍了 Linux文件系统，本实验的实验指导目的是希望读者能综合运用操作系统原理中所学的知识和 Linux系统实验中锻炼的动手能力，独立思考，通过自己的分析和比较，选择适合自己的方案，完成该实验。

**示例结果测试图如下：（不一定要一模一样，鼓励同学们自行设计Linux文件系统）**





补充资料——Linux文件系统

一、文件系统

文件系统是对常规文件和目录的组织集合。用于创建文件系统的命令是 mkfs。

Linux 的强项之一便是支持种类繁多的文件系统，如下所示。

* + 传统的 ext2 文件系统。
  + 各种原生（native）UNIX 文件系统，比如，Minix、System V 以及 BSD 文件系统。
  + 微软的 FAT、FAT32 以及 NTFS 文件系统。
  + ISO 9660 CD-ROM 文件系统。
  + Apple Macintosh 的 HFS。
  + 一系列网络文件系统，包括广为使用的 SUN NFS（Linux 对 NFS 的实现信息请参见http://nfs.sourceforge.net/） 、IBM 和微软的 SMB、Novell NCP 以及 Carnegie Mellon 大学开发的 Coda 文件系统。
  + 一系列日志文件系统，包括 ext3、ext4、Reiserfs、JFS、XFS 以及 Btrfs。

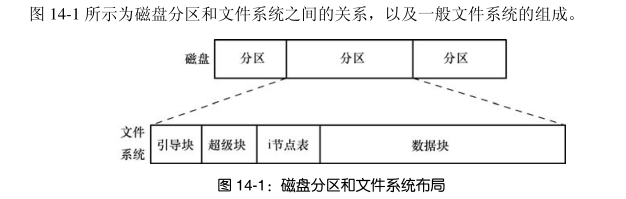
从 Linux 的专有文件/proc/filesystems 中可以查看当前为内核所知的文件系统类型。

ext2 文件系统

多年来， ext2 （扩展文件系统二世） 是 Linux 上使用最为广泛的文件系统， 也是原始 Linux文件系统——ext 的继任者。 近来， 随着各种日志文件系统的兴起， 对 ext2 的使用也日趋减少。有时，在介绍通用文件系统概念时，以一款特定的文件系统实现为例会容易一些，出于这一目的，本章将以 ext2 为例来介绍文件系统。

文件系统结构

在文件系统中，用来分配空间的基本单位是逻辑块，亦即文件系统所在磁盘设备上若干连续的物理块。例如，在 ext2 文件系统上，逻辑块的大小为 1024、2048 或 4096 字节。 （使用mkfs(8)命令创建文件系统时，可指定逻辑块的大小作为命令行参数。 ）



文件系统由以下几部分组成。

①引导块：总是作为文件系统的首块。引导块不为文件系统所用，只是包含用来引导操作系统的信息。操作系统虽然只需一个引导块，但所有文件系统都设有引导块（其中的绝大多数都未使用） 。

②超级块： 紧随引导块之后的一个独立块， 包含与文件系统有关的参数信息， 其中包括：

- i 节点表容量；

- 文件系统中逻辑块的大小；

- 以逻辑块计，文件系统的大小；

驻留于同一物理设备上的不同文件系统，其类型、大小以及参数设置（比如，块大小）都可以有所不同。这也是将一块磁盘划分为多个分区的原因之一。

③i节点表： 文件系统中的每个文件或目录在 i 节点表中都对应着唯一一条记录。 这条记录登记了关乎文件的各种信息。 下一节会深入讨论 i 节点。 有时也将i 节点表称为 i-list。

④数据块：文件系统的大部分空间都用于存放数据，以构成驻留于文件系统之上的文件和目录。

二、i 节点

针对驻留于文件系统上的每个文件，文件系统的 i 节点表会包含一个 i 节点（索引节点的简称） 。对 i 节点的标识，采用的是 i 节点表中的顺续位置，以数字表示。文件的 i 节点号（或简称为 i 号）是 ls –li 命令所显示的第一列。i 节点所维护的信息如下所示。

- 文件类型（比如，常规文件、目录、符号链接，以及字符设备等） 。

- 文件属主（亦称用户 ID 或 UID） 。

- 文件属组（亦称为组 ID 或 GID） 。

- 3 类用户的访问权限：属主（有时也称为用户） 、属组以及其他用户（属主和属组用户之外的用户） 。详情请见 15.4 节。

- 3 个时间戳：对文件的最后访问时间（ls –lu 所显示的时间） 、对文件的最后修改时间（也是 ls –l 所默认显示的时间） ，以及文件状态的最后改变时间（ls –lc 所显示的最后改变 i 节点信息的时间） 。值得注意的是，与其他 UNIX 实现一样，大多数 Linux 文件。系统不会记录文件的创建时间。

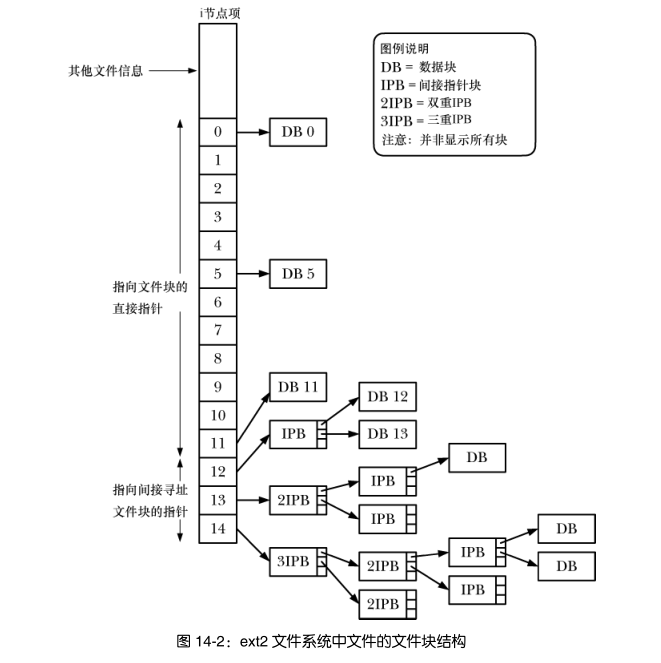
- 指向文件的硬链接数量。

- 文件的大小，以字节为单位。

- 实际分配给文件的块数量，以 512 字节块为单位。这一数字可能不会简单等同于文件的字节大小，分配给文件的块数可能会低于根据文件正常大小（以字节为单位）所计算出的块数。

ext2 中的 i 节点和数据块指针

类似于大多数 UNIX 文件系统，ext2 文件系统在存储文件时，数据块不一定连续，甚至不一定按顺序存放（尽管 ext2 会尝试将数据块彼此靠近存储） 。为了定位文件数据块，内核在 i 节点内维护有一组指针。图 14-2 所示为在 ext2 文件系统上完成上述任务的情况。



在 ext2 中，每个 i 节点包含 15 个指针。其中的前 12 个指针（图 14-2 中编号为 0～11 的指针）指向文件前 12 个块在文件系统中的位置。接下来，是一个指向指针块的指针，提供了文件的第 13 个以及后续数据块的位置。指针块中指针的数量取决于文件系统中块的大小。每个指针需占用 4 字节，因此指针的数量可能在 256（块容量为 1024 字节）～1024（块容量为4096 字节）之间。这样就考虑了大型文件的情况。即便是对于巨型文件，第 14 个指针（图中编号为 13）是一个双重间接指针—指向指针块，其块中指针进而指向指针块，此块中指针最终才指向文件的数据块。只要有体量巨大的文件，就会随之产生更深一层的递进：图中 i节点的最后一个指针属于三重间接指针。

这一貌似复杂的系统，其设计意图是为了满足多重需求。首先，该系统在维持 i 节点结构大小固定的同时，支持任意大小的文件。其次，文件系统既可以以不连续方式来存储文件块，又可通过 lseek()随机访问文件，而内核只需计算所要遵循的指针。最后，对于在大多数系统中占绝对多数的小文件而言，这种设计满足了对文件数据块的快速访问：通过 i 节点的直接指针访问，一击必中。

三、磁盘和分区

常规文件和目录通常都存放在硬盘设备里。 （其他设备也能存放文件和目录，比如，CD-ROM、flash 内存卡以及虚拟磁盘等，但这里主要关注的是硬盘设备。 ）

磁盘驱动器

硬盘驱动器是一种机械装置，由一个或多个高速旋转（每分钟旋转数以千计）的盘片组成。通过在磁盘上快速移动的读/写磁头，便可获取/修改磁盘表面的磁性编码信息。磁盘表面信息物理上存储于称为磁道（track）的一组同心圆上。磁道自身又被划分为若干扇区，每个扇区则包含一系列物理块。物理块的容量一般为 512 字节（或 512 的倍数） ，代表了驱动器可读/写的最小信息单元。

尽管现代磁盘速度很快，但读写磁盘信息耗时依然不菲。首先，磁头要移动到相应磁道（寻道时间） ；然后，在相应扇区旋转到磁头下之前，驱动器必须一直等待（旋转延迟） ；最后，还要从所请求的块上传输数据（传输时间） 。执行上述操作所耗费的时间总量通常以毫秒为单位。相形之下，同样的时间可供现代 CPU 执行数百万条指令。

磁盘分区

可将每块磁盘划分为一个或多个（不重叠的）分区。内核则将每个分区视为位于/dev 路

径下的单独设备。

磁盘分区可容纳任何类型的信息，但通常只会包含以下之一。

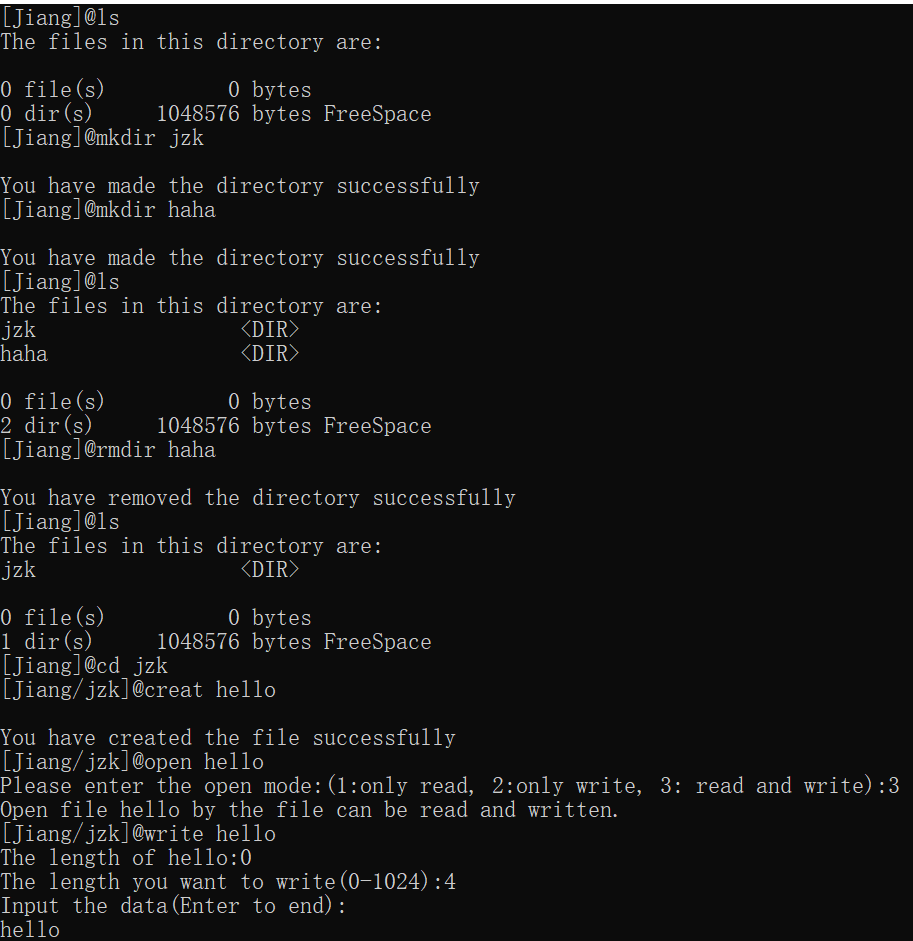
- 文件系统：用来存放常规文件。

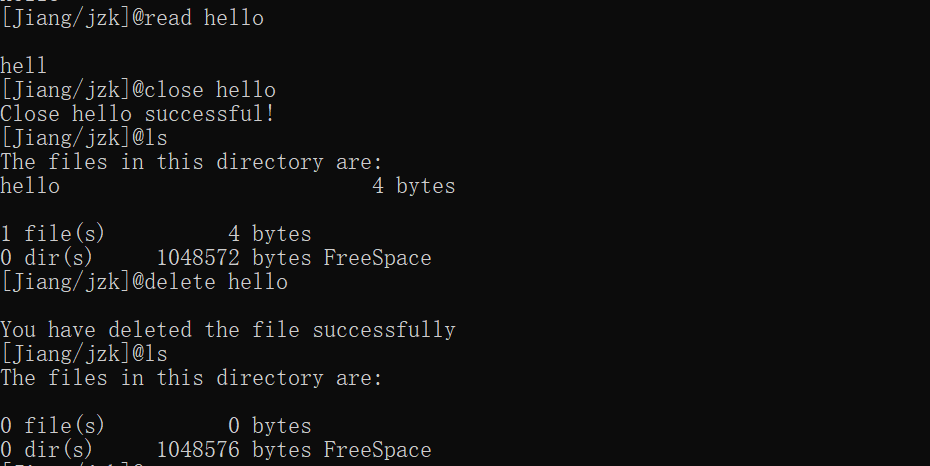
- 数据区域：可做为裸设备对其进行访问。

- 交换区域：供内核的内存管理之用。

可通过 mkswap(8)命令来创建交换区域。特权级进程(CAP\_SYS\_ADMIN)可利用 swapon()系统调用向内核报告将磁盘分区用作交换区域。 swapoff()系统调用则会执行反向功能—告之内核，停止将磁盘分区用作交换区域。尽管 SUSv3 并未对上述系统调用进行规范，但它们却获得了许多 UNIX 实现的支持。其他信息请参考 swapon(2)、swapon(8)手册页。

1. 调试分析与测试结果





四、附录 （程序清单）

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define BLOCK\_SIZE 512 // 数据块的大小

#define BLOCK\_NUM 512 // 数据块的块数

#define INODE\_NUM 32 // i节点的数目

#define FILENUM 8 // 打开文件表的数目

typedef struct

{

char user\_name[10]="Jiang"; // 用户名

} User;

// i节点(32B)

typedef struct

{

short inum; // 文件i节点号

char file\_name[10]; // 文件名

char type; // 文件类型

char user\_name[10]; // 文件所有者

short iparent; // 父目录的i节点号

short length; // 文件长度

short address[2]; // 存放文件的地址

} Inode;

// 打开文件表(16B)

typedef struct

{

short inum; // i节点号

char file\_name[10]; // 文件名

short mode; // 读写模式(1:read, 2:write,

// 3:read and write)

short offset; // 偏移量

} File\_table;

int argc; // 命令的参数的个数

char \*argv[5]; // 命令的参数

int cur\_dir; // 当前目录

char the\_buffer[2\*BLOCK\_SIZE]; // 缓冲区

User user; // 当前用户

char bitmap[BLOCK\_NUM]; // 位示图数组

Inode inode\_array[INODE\_NUM]; // i节点数组

File\_table file\_array[FILENUM]; // 打开文件表数组

char image\_name[10] = "hd.dat"; // 文件系统名称

FILE \*fp; // 打开文件指针

//函数声明

void init(void);

int analyse(char \*);

void save\_inode(int);

int get\_BLOCK\_NUM(void);

void read\_blk(int);

void write\_blk(int);

void release\_blk(int);

void pathset();

void delet(int innum);

int check(int i);

void help(void);

void cd(void);

void ls(void);

void mkdir(void);

void creat(void);

void open(void);

void read(void);

void write(void);

void close(void);

void del(void);

void command(void);

void rmdir();

//命令解析：

int command\_analyse(char \*str)

{

int i;

char the\_buffer[20];

char \*ptr\_char;

char \*syscmd[]={"", "cd", "ls", "mkdir", "creat", "open", "read", "write", "close", "delete","rmdir"}; //, "logout", "clear","format","quit","rmdir"};

argc = 0;

for(i = 0, ptr\_char = str; \*ptr\_char != '\0'; ptr\_char++)

{

if(\*ptr\_char != ' ')

{

while(\*ptr\_char != ' ' && (\*ptr\_char != '\0'))

the\_buffer[i++] = \*ptr\_char++;

argv[argc] = (char \*)malloc(i+1);

strncpy(argv[argc], the\_buffer, i);

argv[argc][i] = '\0';

argc++;

i = 0;

if(\*ptr\_char == '\0') break;

}

}

if(argc != 0)

{

for(i = 0; (i < 15) && strcmp(argv[0], syscmd[i]); i++);

return i;

}

else return 15;

}

//文件创建/删除命令

void creat(void)

{

int i;

if(argc != 2)

{

printf("command creat must have one args. \n");

return ;

}

for(i = 0; i < INODE\_NUM; i++)

{

if((inode\_array[i].inum > 0) && (inode\_array[i].type == '-') && !strcmp(inode\_array[i].file\_name, argv[1]))

{

printf("The file has been exsited.\n");

return ;

}

}

for(i = 0; i < INODE\_NUM; i++)

if(inode\_array[i].inum < 0) break;

if(i == INODE\_NUM)

{

printf("There is no else INode.\n");

exit(-1);

}

inode\_array[i].inum = i;

strcpy(inode\_array[i].file\_name, argv[1]);

inode\_array[i].type = '-';

strcpy(inode\_array[i].user\_name, user.user\_name);

inode\_array[i].iparent = cur\_dir;

inode\_array[i].length = 0;

save\_inode(i);

}

void delet(int innum)

{

inode\_array[innum].inum = -1;

if(inode\_array[innum].length >= 0)

{

release\_blk(inode\_array[innum].address[0]);

if(inode\_array[innum].length >= 512)

release\_blk(inode\_array[innum].address[1]);

}

save\_inode(innum);

}

void del(void)

{

int i,chk;

if(argc != 2)

{

printf("Command delete must have one args. \n");

return ;

}

for(i = 0; i < INODE\_NUM; i++)

if((inode\_array[i].inum > 0) && (inode\_array[i].type == '-') && !strcmp(inode\_array[i].file\_name, argv[1]))

break;

if(i == INODE\_NUM)

{

printf("This file doesn't exist.\n");

return ;

}

chk=check(i);

if(chk!=1)

{

printf("This file is not yours !\n");

return ;

}

delet(i);

}

//目录创建/删除命令

void mkdir(void)

{

int i;

if(argc != 2)

{

printf("command mkdir must have two args. \n");

return ;

}

for(i = 0; i < INODE\_NUM; i++)

if(inode\_array[i].inum < 0) break;

if(i == INODE\_NUM)

{

printf("Inode is full.\n");

exit(-1);

}

inode\_array[i].inum = i;

strcpy(inode\_array[i].file\_name, argv[1]);

inode\_array[i].type = 'd';

strcpy(inode\_array[i].user\_name,user.user\_name);

inode\_array[i].iparent = cur\_dir;

inode\_array[i].length = 0;

save\_inode(i);

}

void rmdir()

{

int i,j,t,flag=0;

if(argc != 2)

{

printf("Command delete must have one args. \n");

return ;

}

for(i = 0; i < INODE\_NUM; i++)

if((inode\_array[i].inum > 0) && (inode\_array[i].iparent == cur\_dir) && (inode\_array[i].type == 'd')&&(!strcmp(inode\_array[i].file\_name,argv[1])))

{

int chk=check(i);

if(chk!=1)

{

printf("This directory is not your !\n");

return ;

}

else j=inode\_array[i].inum;

for(t=0;t<INODE\_NUM;t++)

{

if((inode\_array[t].inum>0)&&(inode\_array[t].iparent==j)&&(inode\_array[i].type == '-'))

delet(t);

else if((inode\_array[t].inum>0)&&(inode\_array[t].iparent==j)&&(inode\_array[i].type == 'd'))

delet(t);

}

if(t == INODE\_NUM)

delet(j);

}

if(i == INODE\_NUM)

delet(i);

return;

}

//显示目录内容命令

void ls(void)

{

int i;

int dcount=0,fcount=0;

short bcount=0;

if(argc != 1)

{

printf("Command ls must have one args. \n");

return ;

}

for(i = 0; i < INODE\_NUM; i++)

if((inode\_array[i].inum> 0) && (inode\_array[i].iparent == cur\_dir))

{

if(inode\_array[i].type == 'd' && check(i))

{

dcount++;

printf("%-20s<DIR>\n", inode\_array[i].file\_name);

}

if(inode\_array[i].type == '-' && check(i))

{

fcount++;

bcount+=inode\_array[i].length;

printf("%-20s%12d bytes\n", inode\_array[i].file\_name,inode\_array[i].length);

}

}

printf("\n%d file(s)%11d bytes\n",fcount,bcount);

printf("%d dir(s) %11d bytes FreeSpace\n",dcount,1024\*1024-bcount);

}

//文件打开/关闭open/ close

void open()

{

int i, inum, mode, filenum,chk;

if(argc != 2)

{

printf("command open must have one args. \n");

return ;

}

for(i = 0; i < INODE\_NUM; i++)

if((inode\_array[i].inum > 0) && (inode\_array[i].type == '-') && !strcmp(inode\_array[i].file\_name,argv[1]))

break;

if(i == INODE\_NUM)

{

printf("The file you want to open doesn't exsited.\n");

return ;

}

inum = i;

chk=check(i);

if(chk!=1)

{

printf("This file is not your !\n");

return ;

}

printf("Please enter the open mode:(1:only read, 2:only write, 3: read and write):");

scanf("%d", &mode);

gets(the\_buffer);

if((mode < 1) || (mode > 3))

{

printf("the number you entered is wrong.\n");

return;

}

for(i = 0; i < FILENUM; i++)

if(file\_array[i].inum < 0) break;

if(i == FILENUM)

{

printf("The file table is full, please close some file.\n");

return ;

}

filenum = i;

file\_array[filenum].inum = inum;

strcpy(file\_array[filenum].file\_name, inode\_array[inum].file\_name);

file\_array[filenum].mode = mode;

file\_array[filenum].offset = 0;

printf("Open file %s by ", file\_array[filenum].file\_name);

if(mode == 1) printf("the file can be only read.\n");

else if(mode == 2) printf("the file can be only written.\n");

else printf("the file can be read and written.\n");

}

void close(void)

{

int i;

if(argc != 2)

{

printf("Command close must have one args. \n");

return ;

}

for(i = 0; i < FILENUM; i++)

if((file\_array[i].inum > 0) && !strcmp(file\_array[i].file\_name, argv[1]))

break;

if(i == FILENUM)

{

printf("This file doesn't be opened.\n");

return ;

}

else

{

file\_array[i].inum = -1;

printf("Close %s successful!\n", argv[1]);

}

}

//文件读/写 read/write

void read()

{

int i, start=0, num, inum;

if(argc != 2)

{

printf("command read must have one args. \n");

return;

}

for(i = 0; i < FILENUM; i++)

if((file\_array[i].inum > 0) && !strcmp(file\_array[i].file\_name,argv[1]))

break;

if(i == FILENUM)

{

printf("Open %s first.\n", argv[1]);

return ;

}

else if(file\_array[i].mode == 2)

{

printf("Can't read %s.\n", argv[1]);

return ;

}

inum = file\_array[i].inum;

if(inode\_array[inum].length > 0)

{

gets(the\_buffer);

read\_blk(inum);

for(i = 0; the\_buffer[i] != '\0'; i++)

printf("%c", the\_buffer[i]);

printf("\n");

}

}

void write()

{

int i, inum, length=0;

if(argc != 2)

{

printf("Command write must have one args. \n");

return ;

}

for(i = 0; i < FILENUM; i++)

if((file\_array[i].inum>0)&&!strcmp(file\_array[i].file\_name,argv[1]))

break;

if(i == FILENUM)

{

printf("Open %s first.\n", argv[1]);

return ;

}

else if(file\_array[i].mode == 1)

{

printf("Can't write %s.\n", argv[1]);

return ;

}

inum = file\_array[i].inum;

printf("The length of %s:%d\n", inode\_array[inum].file\_name, inode\_array[inum].length);

if(inode\_array[inum].length == 0)

{

printf("The length you want to write(0-1024):");

scanf("%d", &length);

gets(the\_buffer);

if((length < 0) && (length >1024))

{

printf("Input wrong.\n");

return ;

}

inode\_array[inum].length = length;

inode\_array[inum].address[0] = get\_BLOCK\_NUM();

if(length > 512)

inode\_array[inum].address[1] = get\_BLOCK\_NUM();

save\_inode(inum);

printf("Input the data(Enter to end):\n");

gets(the\_buffer);

write\_blk(inum);

}

else

printf("This file can't be written.\n");

}

//切换当前目录

void cd(void)

{

int i;

if(argc != 2)

{

printf("Command cd must have two args. \n");

return ;

}

if(!strcmp(argv[1], ".."))

cur\_dir = inode\_array[cur\_dir].iparent;

else

{

// 遍历i节点数组

for(i = 0; i < INODE\_NUM; i++)

if((inode\_array[i].inum>0)&&(inode\_array[i].type=='d')&&(inode\_array[i].iparent==cur\_dir)&&!strcmp(inode\_array[i].file\_name,argv[1])&&check(i))

break;

if(i == INODE\_NUM)

printf("This directory isn't exsited.\n");

else

cur\_dir = i;

}

}

void init(void)

{

int i;

if((fp = fopen(image\_name, "r+b")) == NULL)

{

printf("Can't open file %s.\n", image\_name);

exit(-1);

}

for(i = 0; i < BLOCK\_NUM; i++)

bitmap[i] = fgetc(fp);

for(i = 0; i < INODE\_NUM; i++)

fread(&inode\_array[i], sizeof(Inode), 1, fp);

cur\_dir = 0;

for(i = 0; i < FILENUM; i++)

file\_array[i].inum = -1;

}

void save\_inode(int num)

{

if((fp=fopen(image\_name, "r+b")) == NULL)

{

printf("Can't open file %s\n", image\_name);

exit(-1);

}

fseek(fp, 512+num\*sizeof(Inode), SEEK\_SET);

fwrite(&inode\_array[num], sizeof(Inode), 1, fp);

fclose(fp);

}

int get\_BLOCK\_NUM(void)

{

int i;

for(i = 0; i < BLOCK\_NUM; i++)

if(bitmap[i] == '0') break;

// 未找到空闲数据块

if(i == BLOCK\_NUM)

{

printf("Data area is full.\n");

exit(-1);

}

bitmap[i] = '1';

if((fp=fopen(image\_name, "r+b")) == NULL)

{

printf("Can't open file %s\n", image\_name);

exit(-1);

}

fseek(fp, i, SEEK\_SET);

fputc('1', fp);

fclose(fp);

return i;

}

void read\_blk(int num)

{

int i, len;

char ch;

int add0, add1;

len = inode\_array[num].length;

add0 = inode\_array[num].address[0];

if(len > 512)

add1 = inode\_array[num].address[1];

if((fp = fopen(image\_name, "r+b")) == NULL)

{

printf("Can't open file %s.\n", image\_name);

exit(-1);

}

fseek(fp, 1536+add0\*BLOCK\_SIZE, SEEK\_SET);

ch = fgetc(fp);

for(i=0; (i < len) && (ch != '\0') && (i < 512); i++)

{

the\_buffer[i] = ch;

ch = fgetc(fp);

}

if(i >= 512)

{

fseek(fp, 1536+add1\*BLOCK\_SIZE, SEEK\_SET);

ch = fgetc(fp);

for(; (i < len) && (ch != '\0'); i++)

{

the\_buffer[i] = ch;

ch = fgetc(fp);

}

}

the\_buffer[i] = '\0';

fclose(fp);

}

void write\_blk(int num)

{

int i, len;

int add0, add1;

add0 = inode\_array[num].address[0];

len = inode\_array[num].length;

if((fp = fopen(image\_name, "r+b")) == NULL)

{

printf("Can't open file %s.\n", image\_name);

exit(-1);

}

fseek(fp, 1536+add0\*BLOCK\_SIZE, SEEK\_SET);

for(i=0; (i<len)&&(the\_buffer[i]!='\0')&&(i < 512); i++)

fputc(the\_buffer[i], fp);

if(i == 512)

{

add1 = inode\_array[num].address[1];

fseek(fp, 1536+add1\*BLOCK\_SIZE, SEEK\_SET);

for(; (i < len) && (the\_buffer[i] != '\0'); i++)

fputc(the\_buffer[i], fp);

}

fputc('\0', fp);

fclose(fp);

}

void release\_blk(int num)

{

FILE \*fp;

if((fp=fopen(image\_name, "r+b")) == NULL)

{

printf("Can't open file %s\n", image\_name);

exit(-1);

}

bitmap[num] = '0';

fseek(fp, num, SEEK\_SET);

fputc('0', fp);

fclose(fp);

}

void pathset()

{

char path[50];

int m,n;

if(inode\_array[cur\_dir].inum == 0)

strcpy(path,user.user\_name);

else

{

strcpy(path,user.user\_name);

m=0;

n=cur\_dir;

while(m != cur\_dir)

{

while(inode\_array[n].iparent != m)

{

n = inode\_array[n].iparent;

}

strcat(path,"/");

strcat(path,inode\_array[n].file\_name);

m = n;

n = cur\_dir;

}

}

printf("[%s]@",path);

}

int check(int i)

{

int j;

char \*uuser,\*fuser;

uuser=user.user\_name;

fuser=inode\_array[i].user\_name;

j=strcmp(fuser,uuser);

if(j==0) return 1;

else return 0;

}

void errcmd()

{

printf("This command is not allowed\n");

}

void command(void)

{

char cmd[100];

system("cls");

do

{

pathset();

gets(cmd);

switch(command\_analyse(cmd))

{

case 1:

cd();

break;

case 2:

printf("The files in this directory are:\n");

ls();

break;

case 3:

mkdir();

break;

case 4:

creat();

break;

case 5:

open();

break;

case 6:

read();

break;

case 7:

write();

break;

case 8:

close();

break;

case 9:

del();

break;

case 10:

rmdir();

break;

case 11:

errcmd();

break;

default:

break;

}

}while(1);

}

int main(void)

{

init();

command();

return 0;

}

**暨南大学本科实验报告专用纸(附页)**