



浙江工业大学

# UNIX 模拟文件管理系统

姓名：华俊豪

班级：计算机+自动化 0901

学号：200926630107

## 目录

一、	磁盘分区 .....	3
二、	UNIX 文件的物理结构 .....	3
三、	UNIX 文件目录结构 .....	6
四、	UNIX 存储空间的管理 .....	7
五、	UNIX 打开文件管理 .....	9
六、	其它数据结构和预定义 .....	11
七、	功能模块 .....	13
7.1	磁盘管理模块 .....	13
7.2	目录管理模块 .....	14
7.3	文件管理模块 .....	20
7.4	用户管理模块 .....	25
八、	实验总结 .....	26
九、	参考文献 .....	26

## 一、磁盘分区

开辟一块 32MB 的文件 VirtualDisk 用做虚拟磁盘（文件卷）。磁盘块大小为 512B

磁盘引导块#0	超级块#1 SuperBlock	磁盘 i 结点#2-#513 Dinode	MFD主目录#514	用户表#515-#516	数据块#517-#65535
---------	---------------------	--------------------------	------------	--------------	----------------

图 1-1 文件卷结构图

- ✧ #0 块为磁盘引导块，不属于文件系统管辖；保留。
- ✧ #1 块为文件卷的管理块（超级块 SuperBlock），它记录该文件卷上磁盘块的使用情况；
- ✧ #2~#514 为磁盘 inode 结点去，其中#514 为 MFD 主目录，不可改变，因此可分配的 inode 结点为#2~#513，共 512 块。
- ✧ #515-516 为用户表，存放用户信息；
- ✧ #517~65535 为文件数据区，共 65019 块；

## 二、UNIX 文件的物理结构

UNIX 文件的物理结构采用多级索引链接结构。在文件磁盘 i 结点数据结构（详见数据结构设计-dinode）中有一个文件物理地址索引表 `i_addr[13]`。利用这 13 个地址指针实现文件物理结构的索引，如图 1-2 所示。

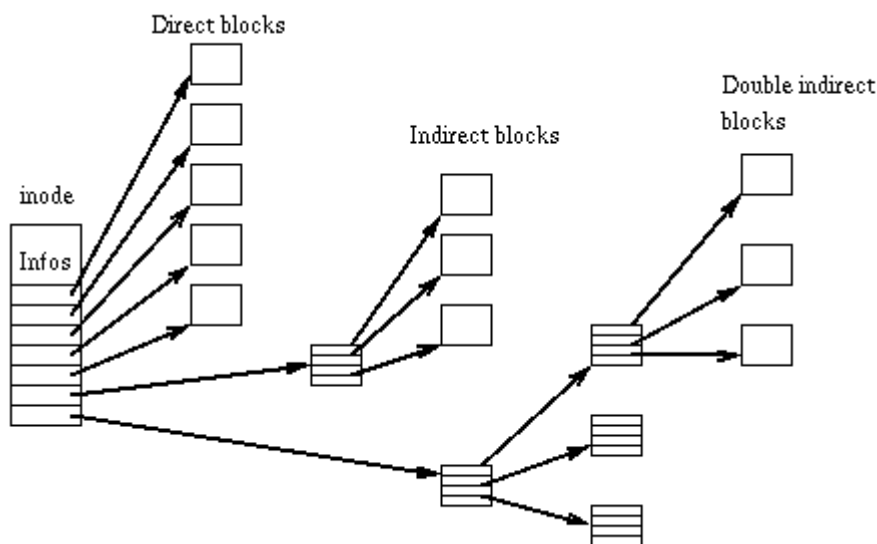


图 1-2 UNIX 文件的物理结构（from google image）

文件物理地址索引表 `I_addr[13]`说明：

- `I_addr[0]~i_addr[9]` 为直接索引，这 10 个指针指向文件的第 0-9 个逻辑块对于的物理块地址。只需读盘一次。
- `I_addr[10]`为一次间接索引指针，他指向一个一次间接索引表（详见数据结构-Indir），该表占用一个物理块（512B），地址指针（unsigned short）占用 2B,则一个索引表可以存放 256 个索引指针，对应与 256 个物理块，则一次间接索引对应的逻辑块号为 10~（255+10）块。需要两次读盘。

```

/*磁盘索引结点
文件控制块：FCB = BFD + SFD
基本文件目录表：BFD
每个分配内存 B
62B
64 * 8 = 512
#2 - #513
共 512* 8 = 4096 个 i 结点
*/
struct dinode
{
    unsigned short i_uid; //属主的 UID                2B
    unsigned short i_gid; // 属主的组 ID (GID) 2B    文件主：x01H,同组用户：
x02h,其它用户：x04h
    unsigned short i_type; //文件类型                2B
    unsigned short i_mode; // 文件存取权限    2B
    unsigned short i_nlink; // 文件共享连接数    2B

    time_t CreateTime; // 文件创建时间 di_ctime    8B
    // time_t LastModified; // 最近一次修改时间 di_mtime    8B
    time_t LastAccessed; // 最近一次访问时间 di_atime    8B

    unsigned short i_size; // 文件长度 2B
    unsigned short i_addr[13]; // 26B, 文件最大为 G , 若为目录, 则存放目录的
SFD 物理块号
//0-9 : 0-9
// 10 : 512/2 = 256 : 10-265
// 11 : 256*256= 65536 : 266 - 65801
// 12 : 256*256*256 = 16777216 : 65802 - 16843017
// total 10 + 256 + 256*256 + 256*256*256 = 16843018
};

```

## 2. 内存 i 结点

```
/*
82B
sizeof : 88B ?
*/
struct inode
{
    struct dinode _base;
    // add
    unsigned short i_flag; //内存结点状态 2B
    unsigned short i_dev;  // 设备号 2B
    unsigned short i_Index; // 索引 i 结点号 2B
    unsigned short i_count; // 引用计数。当文件打开时，其引用计数加，关闭时
引用计数减 2B
};

// 文件目录项:  directory ,Catalog
// 512B = 16B * 32
// SFD signal file directory
// 16B
```

## 3. 间接索引表

```
//
// 一块
// 256*2 = 512
struct Indir // 间接索引表
{
    unsigned short index[256];
};
```

### 三、UNIX 文件目录结构

Unix 采用树状目录结构，每个目录表示为一个目录表结构，表中的每一个目录项（详见数据结构—SFD）占用 16B, 2 个字节为该文件的结点号（内部标识），14B 为该文件的文件名（外部标识），所以 UNIX 的目录其实是文件内部标识和外部标识的一个映射表。

在 Unix 中不区分文件和目录，认为目录也是一种文件，因此其目录的存放结构和文件一样，都用一个 inode 唯一表示一个文件，用 inode 的 i\_type 属性表征类型，不同的是目录的文件内容为目录表（详见数据结构—Dir）。

#### ● 相关的数据结构：

##### 1. 文件目录项

```
// directory ,Catalog
// 512B = 16B * 32
// SFD signal file directory
// 16B

struct SFD{
    unsigned short d_ino; // 文件内部标识 2B :  $2^{16} = 64K$  个 i 结点
    char d_name[14]; // 文件符号名
};
```

##### 2. 目录

```
// 16*31
// 500 分配一块 B，可有条目录项
struct dir
{
    struct SFD SFD[DIRSIZE];
    unsigned int size; // 目录项个数
};
```

## 四、UNIX 存储空间的管理

### ➤ 超级块:

- 磁盘分区中已经提到#1 为**超级块**（**数据结构- SuperBlock**），是操作系统用来描述磁盘使用状态的数据结构，内存中也有一个与磁盘管理块相对应的**数据结构 filsys**。在磁盘被挂接到文件系统上时，UNIX 将超级块的信息复制到内存管理块 filsys 中，此后对于磁盘空闲块的所有操作都在内存中进行。

### ➤ UNIX 空闲磁盘块的成组链接法

- 将所有空闲块按照每 50 块（实际情况为 100 块）为单位进行分组，然后将每组中 100 块的地址指针以此送到其下一组的第 0 块中保存。这样，每一组的第 0 块（**数据结构- free\_block**）中存放了其前一组的 50 块的地址，因此第 0 块一方面充当了数据结构，另一方面该块依然是可以被分配的空闲块。这样，整个磁盘物理有多少空闲块都不需要占用额外的磁盘空间用来存放数据结构。
- 另一方面，由于每组只有 50 块，所有 UNIX 只需在内存 filsys 中开辟 50 个地址指向当前组的 50 块，就可以实现在内存中分配或回收空闲块，因此有很好的效率。

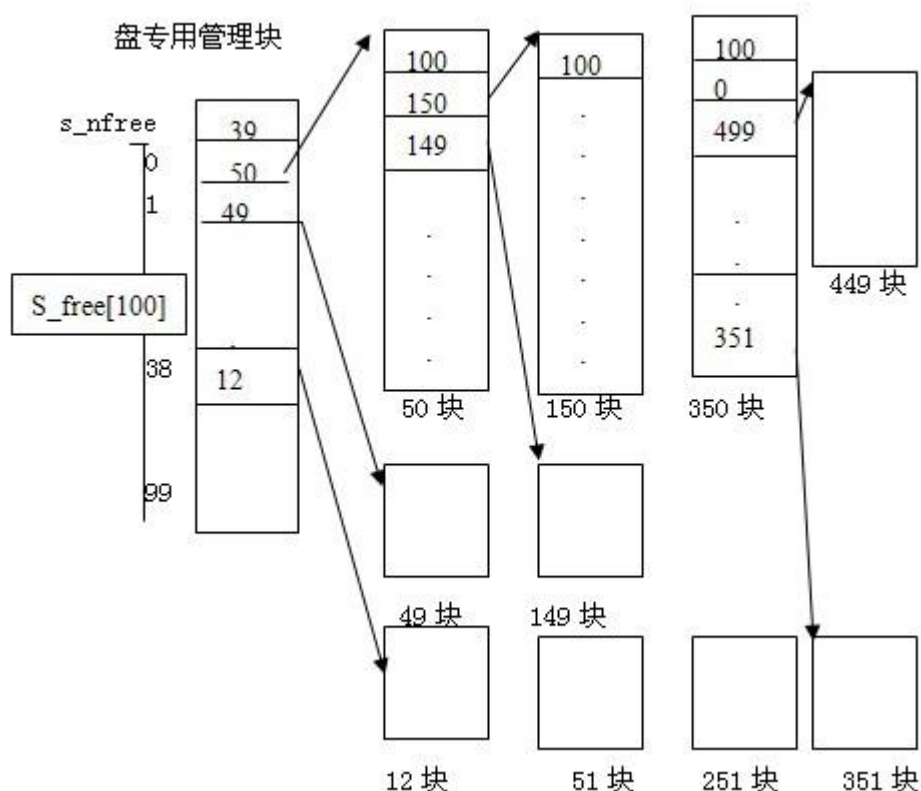


图 1-3 UNIX 成组链式法 (from google Images)

### ● 相关的数据结构:

#### 1. 超级块

```

/*
unix : 成组链接法
每块为单位进行分组
filsys : 安装后常驻内存
456B

分配 512B
#1
*/
struct SuperBlock
{
    unsigned int s_isize; // i 结点区总块数 4B
    unsigned int s_fsize; // 文件数据区总块数 4B

    unsigned int s_tfree; // 空闲数据区总块数 4B
    unsigned int s_tinode; // 空闲 i 结点总数 4B

    unsigned int s_nfree; // 直接管理的空闲块数,栈顶 4B
    unsigned int s_free[GROUFSIZE]; // 空闲块号栈 unsigned int 2^32 = 4G
    200B

    unsigned int s_ninode; // 直接管理的空闲 i 结点数,栈顶 4B
    unsigned int s_inode[GROUFSIZE]; // 空闲 i 结点栈 200B

    char s_flock; // 处理空闲块表时加锁标志 1B
    char s_iloc; // 处理空闲索引结点表时加锁标志 1B
    char s_fmod; // flag: 超级块被修改标志 1B
    char s_ronly; // flag: file system is read only 1B
    time_t s_time; // time of last superblock update 8B
    char s_fname[6]; // 文件系统名称 6B
    long s_type; // 文件系统类型 4B
};

```

## 2. 空闲盘块

```

// 51*4 = 204B < 512B
struct free_stack
{
    unsigned int s_nfree;
    unsigned int s_free[50];
};

```

## 3. filsys

```

extern struct SuperBlock filsys; // 常驻内存超级块

```



## 五、UNIX 打开文件管理

- 每个进程在其进行控制块的 **User 结构** (*数据结构- User*), 中有一个**进程打开文件表** `u_ofile[NFILE]`, `NFILE` 为该进程最多能同时打开文件的个数 (默认为 10), 其值在 UNIX 中可以修改, 为简单起见, 在本系统中设为一个常量。`u_ofile[]`数组含有该进程所有已经打开文件的返回的文件描述符(`fd`), 这里为一个指向**系统打开文件表** (*数据结构- file*), 的指针, 即每个文件的内部标识与其在系统打开文件表中对应的一个表项之间的索引, 它是面向进程的。
- 用户进程对于文件的使用存在着多对多的联系, 父子多个进程可以使用同一方式, 同一读写指针来共享同一个文件, 多个独立进程也可以使用不同的方式以不同的读写指针共享一个文件。  
在系统打开文件表中描述了进程对文件在动态使用中产生的诸如打开方式是读还是写, 读写的位置, 是否是父子进程的继承等既不属于进程又不属于文件属性, 而两者关联并使用的过程中才具有的属性, 详见 *数据结构- file*。
- **进程打开文件表-系统打开文件表-内存 i 结点表的关系**

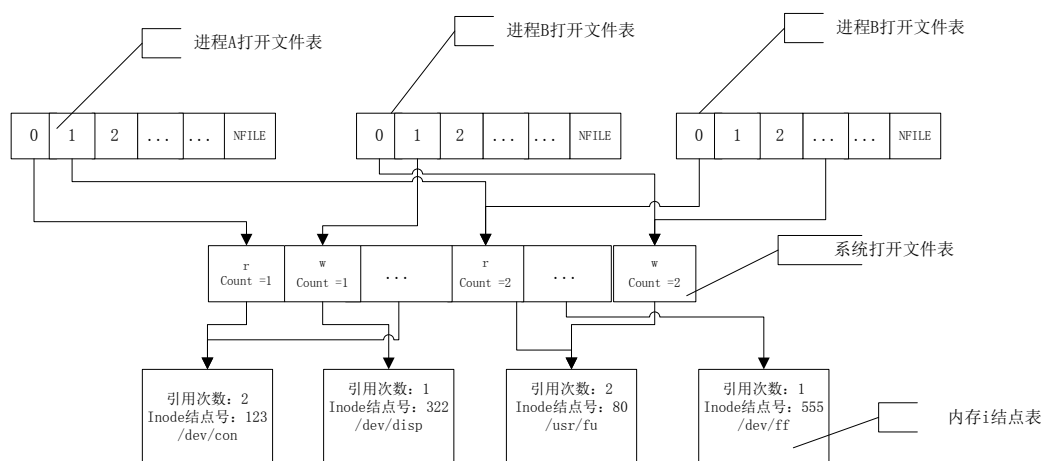


图 1-4 进程打开文件表-系统打开文件表-内存 i 结点表的关系

### ● 相关的数据结构:

#### 1. 系统打开表

// 13B

`struct file`

{

`char f_mode;` // 打开方式 , 读 r 还是写 w 还 wr 1B

`unsigned int f_count;` // 描述父子进程的共享计数 4B

`unsigned int f_offset;` // 读写位置指针 4B

`struct inode * f_inode;` // 指向内存 i 结点指针 4B

};

`extern struct file sys_file[OPENFILEMAX];` // 系统文件表

`extern unsigned short p_file;` // 系统打开文件个数

#### 2. 用户表

```
// 92B //90B
// unsigned int UserCount ; 4B
// 92 * 10 + 4 = 924 < 512 * 2 = 1024
// USERMAX = 10
// #522-#523
struct user
{
    char uid[20];
    char pwd[20];
    unsigned short u_defalut_mode; //
    unsigned short u_uid;
    unsigned short u_gid;
    unsigned int u_ofile[NOFILE]; // 用户打开文件表, default 10
    unsigned int p_file; // 打开文件个数
};
```

### 3. 内存 i 结点表

```
extern unsigned short p_inode; // 活动 inode 个数
extern struct inode inodeList[INODEMAX]; // 内存 i 结点表活动 inode . 打开的文件表
```

## 六、其它数据结构和预定义

- 为提高维护便利性而设计的数据结构

```
extern unsigned short inodeIndex[INODEMAX]; //活动 inode index
extern inode cur_inode; //当前活动 i 结点
extern struct user curUser; // 当前用户
extern char** curDir; // 当前目录
extern unsigned short dirLen; // 目录长度
extern unsigned short d_inoList[50]; // 记录当前路径下的各目录的物理块号
extern unsigned short p_ino; //头指针
```

- 预定义：最常用的常量

```
// 32M
#define BLOCKSIZE 512 // 块大小 512bytes
#define BLOCKNUM 64*1024 // 65536 块个数
#define INODEBLOCKNUM 512 // i 结点块个数 1%
#define INODEMAX 32 // 最大内存 i 结点个数
#define USERMAX 10 // 最大用户数目
#define OPENFILEMAX 50
#define DATABLOCKNUM BLOCKNUM - DATA_OFFSET //数据块大小
#define DIFILE 0x0001 // 文件
#define DIDIR 0x0002 // 目录
#define GROUPSIZE 50 //组大小
#define INODE_OFFSET 2 // i 结点块号偏移量
#define MFD_OFFSET 514
#define USER_OFFSET 515 //用户表偏移量
#define DATA_OFFSET 517 //数据块块号偏移量
#define NOFILE 10 // 用户打开文件表, default 10
#define DIRSIZE 31 // 一块目录的长度

#define DEFAULTMODE 0X0000 //默认模式
```

- 预定义：尚未使用的常量（便于扩展）：

```
// u 文件主
// g 同组用户
// o 其他用户
// a 所有用户
#define RUNTIMEMODFUID 0X4000 // 运行时可改变 UID
#define RUNTIMEMODFGID 0X2000 // 运行时可改变 GID
```

```
// 文件主权限
#define UEXEC 0X0100 // execute 执行文件
#define UREAD 0X0200 // read 读文件
#define UWRITE 0X0400 // write 写文件
#define UDELETE 0X0800 // delete 删除文件
// 同组权限
#define GEXEC 0X0010 // execute 执行文件
#define GREAD 0X0020 // read 读文件
#define GWRITE 0X0040 // write 写文件
#define GDELETE 0X0080 // delete 删除文件
// 其它权限
#define OEXEC 0X0001 // execute 执行文件
#define OREAD 0X0002 // read 读文件
#define OWRITE 0X0004 // write 写文件
#define ODELETE 0X0008 // delete 删除文件

#define LOCK 0X01// 加锁标志
#define UNLOCK 0X00 // 未加锁

#define MODIFIED 0x01//被修改标志
#define UNMODIFIED 0x00// 未被修改

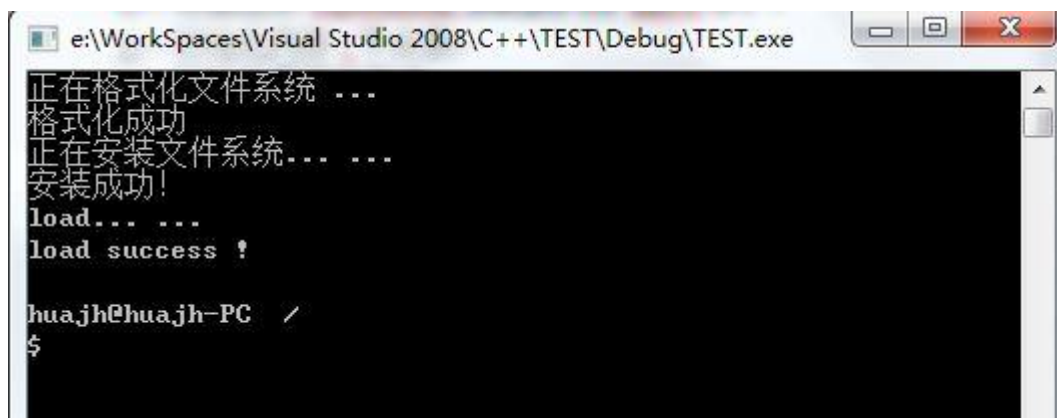
#define READONLY 0X01 // 只读
#define CANWRITE 0X02// 可写

// 文件系统类型
#define BIN 0X0001
#define DEV 0X0002
#define ETC 0X0004
#define LIB 0X0008
#define MNT 0X0010
#define SBIN 0X0020
#define TMP 0X0040
#define USR 0X0080
#define VAR 0X0100
#define HOME 0X0200
```

## 七、功能模块

### 7.1 磁盘管理模块

- 格式化文件系统 `format()`
  - 为 VirtualDisk 申请 32MB 空间。
- 安装文件系统 `install()`
  - 初始化超级块
  - 分配#2-#513 给 inode 块, #514 给 MFD 主目录,#515-#516 给用户表, #517-#65535 给数据块;
  - 采用成组链式法初始化磁盘空闲块, 链接空闲块;
  - 创建一个超级用户 user。
- 加载文件系统 `load()`
  - 将超级块 `superBlock` 加载到内存 `filsys` 中;
  - 初始化内存 i 节点表;
  - 初始化系统打开表;
  - 设定当前主目录。
- 保存文件系统使用信息 `save()` (程序运行结束时执行)
  - 将 `filsys` 的文件管理系统存回磁盘中的 `superBlock` 中。



```
e:\WorkSpaces\Visual Studio 2008\C++\TEST\Debug\TEST.exe
正在格式化文件系统 ...
格式化成功
正在安装文件系统... ..
安装成功!
load... ..
load success !
hua.jh@hua.jh-PC /
$
```

- 分配 inode 空闲块 `_alloc_inode()`
  - 申请一块空闲 inode 节点块，返回 inode 区的逻辑物理块号
- 分配数据块空闲块 `_alloc_data()`
  - 申请一块空闲数据块，返回数据块区的逻辑物理块号
- 释放一块 inode 块 `_release_inode(index)`
  - 释放给定的 inode 逻辑物理块号所指向的物理块，加入 inode 空闲栈中。
- 释放一块数据块 `_release_data(index)`
  - 释放给定的数据块逻辑物理块号所指向的物理块，加入 data 空闲栈中。

## 7.2 目录管理模块

- 目录的添加 `mkdir()`
  - 创建单级目录

A terminal window showing the execution of the mkdir command. The prompt is hua.jh@hua.jh-PC /. The first command is \$ ls, which outputs abc. The second command is \$ mkdir hua. The third command is \$ ls, which outputs abc and hua.

```
hua.jh@hua.jh-PC /  
$ ls  
abc  
  
hua.jh@hua.jh-PC /  
$ mkdir hua  
  
hua.jh@hua.jh-PC /  
$ ls  
abc      hua
```

- 无法创建已经存在的目录或文件

```
hualjh@hualjh-PC /  
$ ls  
hao          hua  
  
hualjh@hualjh-PC /  
$ mkdir hua  
mkdir: cannot create directory 'hua' : Such file or directory has existed  
  
hualjh@hualjh-PC /
```

### ■ 创建多级目录

```
hualjh@hualjh-PC /  
$ mkdir a1/a2/a3  
mkdir: cannot create directory 'a1/a2/a3' : No such file or directory  
  
hualjh@hualjh-PC /  
$ mkdir -p a1/a2/a3  
  
hualjh@hualjh-PC /  
$ cd a1  
  
hualjh@hualjh-PC /a1  
$ cd a2  
  
hualjh@hualjh-PC /a1/a2  
$ cd a3  
  
hualjh@hualjh-PC /a1/a2/a3
```

## ● 目录的删除 rmdir()

### ■ 删除一个空目录

```
hualjh@hualjh-PC /  
$ ls  
hua          jun          hao  
  
hualjh@hualjh-PC /  
$ rmdir jun  
  
hualjh@hualjh-PC /  
$ ls  
hua          hao
```

### ■ 删除多级目录下的目录

```
hualjh@hualjh-PC /hua
$ ls
bbb          aaa

hualjh@hualjh-PC /hua
$ cd ..

hualjh@hualjh-PC /
$ rmdir hua/bbb

hualjh@hualjh-PC /
$ cd hua

hualjh@hualjh-PC /hua
$ ls
aaa
```

- -p 选项：删除所有已经为空的父目录

```
hualjh@hualjh-PC /hua
$ ls
aaa

hualjh@hualjh-PC /hua
$ cd ..

hualjh@hualjh-PC /
$ ls
hao          hua

hualjh@hualjh-PC /
$ rmdir -p hua/aaa

hualjh@hualjh-PC /
$ ls
hao
```

- -删除项目类型不能为文件

```
hualjh@hualjh-PC /hua
$ create gg.x

hualjh@hualjh-PC /hua
$ ls
gg.x

hualjh@hualjh-PC /hua
$ cd ..

hualjh@hualjh-PC /
$ rmdir hua/gg.x
bash: rmdir:is a file , not directory
```



- 目录或文件的重命名 `_rename()`

- 不能重命名为已经存在于目录下的文件名

```
hualjh@hualjh-PC /  
$ ls  
hao          hua  
  
hualjh@hualjh-PC /  
$ rename hua hao  
rename: cannot rename 'hua' : 'hao' file or directory has existed .  
hualjh@hualjh-PC /
```

- 正确的重命名

```
hualjh@hualjh-PC /  
$ ls  
abc  
  
hualjh@hualjh-PC /  
$ rename abc hualjh  
  
hualjh@hualjh-PC /  
$ ls  
hualjh
```

- 目录的拷贝 `copy()`

- 单目录的拷贝

```
hualjh@hualjh-PC /hao  
$ ls  
  
hualjh@hualjh-PC /hao  
$ cd ..  
  
hualjh@hualjh-PC /  
$ copy hua hao  
  
hualjh@hualjh-PC /  
$ cd hao  
  
hualjh@hualjh-PC /hao  
$ ls  
hua  
  
hualjh@hualjh-PC /hao  
$ cd hua  
  
hualjh@hualjh-PC /hao/hua  
$ ls  
aaa          gg.x
```

## ■ 子目录的拷贝

```
hualjh@hualjh-PC /hua
$ ls
aaa          gg.x

hualjh@hualjh-PC /hua
$ cd ..

hualjh@hualjh-PC /
$ copy ggg/kkk hua

hualjh@hualjh-PC /
$ cd hua

hualjh@hualjh-PC /hua
$ ls
aaa          gg.x          kkk
```

- 移动目录 move()

```
hualjh@hualjh-PC /  
$ ls  
hao          jun  
  
hualjh@hualjh-PC /  
$ mv jun hao  
  
hualjh@hualjh-PC /  
$ ls  
hao  
  
hualjh@hualjh-PC /  
$ cd hao  
  
hualjh@hualjh-PC /hao  
$ ls  
jun  
  
hualjh@hualjh-PC /hao  
$ cd jun  
  
hualjh@hualjh-PC /hao/jun  
$ ls  
hua  
  
hualjh@hualjh-PC /hao/jun  
$ cd hua  
  
hualjh@hualjh-PC /hao/jun/hua  
$ ls  
aaa          gg.x  
  
hualjh@hualjh-PC /hao/jun/hua
```

- 目录中文件的显示 ls()

```
hualjh@hualjh-PC /  
$ mkdir hua  
  
hualjh@hualjh-PC /  
$ mkdir jun  
  
hualjh@hualjh-PC /  
$ mkdir hao  
  
hualjh@hualjh-PC /  
$ ls  
hualjh      hua          jun          hao
```

- 进入和退出目录 `cd()`

```
hualjh@hualjh-PC /  
$ cd hualjh  
  
hualjh@hualjh-PC /hualjh  
$ mkdir gg1  
  
hualjh@hualjh-PC /hualjh  
$ cd gg1  
  
hualjh@hualjh-PC /hualjh/gg1  
$ cd ..  
  
hualjh@hualjh-PC /hualjh  
$
```

## 7.3 文件管理模块

- 文件的创建 `_create()`

```
hualjh@hualjh-PC /hualjh  
$ create gg.x  
  
hualjh@hualjh-PC /hualjh  
$ create hh.doc  
  
hualjh@hualjh-PC /hualjh  
$ ls  
gg1          gg.x          hh.doc
```

- 文件的删除 `_delete()`

```
hualjh@hualjh-PC /  
$ ls  
hual  jun          hao          gg.x          ff.h  
dd.c  
  
hualjh@hualjh-PC /  
$ del dd.c  
  
hualjh@hualjh-PC /  
$ ls  
hual  jun          hao          gg.x          ff.h
```

- 打开文件\_open()

- 一个文件必须先打开才能读写，并且必须在打开的时候说明打开方式，  
-w 表示可写方式打开，-r 表示只读方式打开。
- 如果文件处于-w 写方式下，则其它进程不能打开该文件
- 如果文件处于-r 只读方式下，则其它进程也可以只读的方式打开该文件

- 写文件\_write()

- 一个文件以-w 写方式打开后才能写文件

- 读文件\_read()

- 以-w 写方式或者-r 只读方式都可以读文件

- 关闭文件\_close()

- 一个进程不能多次打开同一个文件，必须写关闭(close)才能重新打开

```
hualjh@hualjh-PC /hua
$ create gg.x

hualjh@hualjh-PC /hua
$ open -r gg.x

hualjh@hualjh-PC /hua
$ write gg.x
bash: write : 'gg.x' : cannot write file : the current mode is read only .

hualjh@hualjh-PC /hua
$ open -w gg.x
open : cannot open file again 'gg.x' : You have open the file

hualjh@hualjh-PC /hua
$ close gg.x

hualjh@hualjh-PC /hua
$ open -w gg.x

hualjh@hualjh-PC /hua
$ write gg.x
input whatever you want :
hua jun hao

just for show : hua jun hao

hualjh@hualjh-PC /hua
$ read gg.x

content : hua jun hao

hualjh@hualjh-PC /hua
```

- 文件的拷贝 `copy()`

```
hualjh@hualjh-PC /jun
$ ls
gg.x

hualjh@hualjh-PC /jun
$ cd ..

hualjh@hualjh-PC /
$ copy hua/fff.x jun

hualjh@hualjh-PC /
$ cd jun

hualjh@hualjh-PC /jun
$ ls
gg.x          fff.x
```

- 移动文件 `_move()`

```
hualjh@hualjh-PC /hao
$ ls
jun

hualjh@hualjh-PC /hao
$ cd ..

hualjh@hualjh-PC /
$ mv xyz/dd.cpp hao

hualjh@hualjh-PC /
$ cd hao

hualjh@hualjh-PC /hao
$ ls
jun          dd.cpp

hualjh@hualjh-PC /hao
```

- 文件的重命名 `rename()`

```
hualjh@hualjh-PC /hao
$ ls
jun          dd.cpp

hualjh@hualjh-PC /hao
$ rename dd.cpp gg.xxx

hualjh@hualjh-PC /hao
$ ls
jun          gg.xxx

hualjh@hualjh-PC /hao
```

- 查看文件信息 `lsattr()`

```
hualjh@hualjh-PC /
$ create dd.c

hualjh@hualjh-PC /
$ lsattr
base:  lsattr:command not found

hualjh@hualjh-PC /
$ lsattr dd.c
filename : dd.c
uid : 1
gid : 1
CreateTime : 1324792379
LastAccessed : 1324792379
Type : File
Size : 0
```

- 帮助 `help()`

- 显示某个命令的帮助文档

```

hua.jh@hua.jh-PC /
$ mkdir --help
mkdir:
    [语法]: mkdir [-p] 目录名
    [说明]: 本命令用于建立目录, 要求对其父目录具有写权限
    -p 建立目录时建立其所有不存在的父目录

hua.jh@hua.jh-PC /
$ copy --help
copy:
    拷贝: copy 目录名/文件名 新目录名/文件名

hua.jh@hua.jh-PC /
$ rmdir --help
rmdir:
    [语法]: rmdir [-p] 目录名
    [说明]: 本命令用于删除目录
    -p 删除所有已经为空的父目录
    -s 当使用-p 选项时, 出现错误不提示

hua.jh@hua.jh-PC /

```

## ■ 显示所有命令的帮助文档

```

hua.jh@hua.jh-PC /
$ help
mkdir:
    [语法]: mkdir [-p] 目录名
    [说明]: 本命令用于建立目录, 要求对其父目录具有写权限
    -p 建立目录时建立其所有不存在的父目录

rmdir:
    [语法]: rmdir [-p] 目录名
    [说明]: 本命令用于删除目录
    -p 删除所有已经为空的父目录
    -s 当使用-p 选项时, 出现错误不提示

cd:
    cd 回到注册进入时的目录
    cd /tmp 进入 /tmp 目录
    cd ../ 进入上级目录

rename:
    重命名: rename 目录名/文件名 新目录名/文件名

copy:
    拷贝: copy 目录名/文件名 新目录名/文件名

mv:
    移动: mv 目录名/文件名 新目录名/文件名

create:
    创建文件: create 文件名

del:
    删除文件: del 文件名

open:
    打开文件: open [-w|-r] 文件名
    [说明]: -w 写方式打开, -r 只读方式打开
    写文件内容: write: 文件名

read:
    读文件内容: read: 文件名

close:
    关闭文件: close: 文件名

lsattr:
    查看文件信息: lsattr: 目录名/文件名

```



## 7.4 用户管理模块

- 用户登录 login()

```
load... ...
load success !
input uid: huajh
input password: 1
用户名或密码错误!
input uid: huajh
input password: 123

huajh@huajh-PC /
$
```

- 用户注销 logout()

```
input uid: huajh
input password: 123

huajh@huajh-PC /
$ ls
hua          gg.x          jun          ff.x          dd.c

huajh@huajh-PC /
$ logout
注销成功!
input uid:
```

## 八、实验总结

通过这次操作系统大型实验总得来说有两种感受。

第一，深入理解操作系统的内存分配和资源管理机制，从算法效率角度进一步窥测操作系统对内存资源的调度算法。本次实验采用的是经典的 UNIX 文件系统，从 inode, superBlock 等数据结构、文件物理结构、文件目录结构、成组链式法下的内存分配管理、到多进程下文件的共享和读写保护机制、打开文件管理方法，均与真正的 UNIX 系统在一定程度上保持一致。这无疑使我能最大程度得深入理解 UNIX 文件系统机制。当然随着计算机软硬件的进步，这个经典的文件系统早就被现代更加高效优秀的 UNIX 文件系统所替代，但如 inode 数据结构，树状目录结构等这些东西在现代系统中依然保留，这足以说明该结构的简单高效性。

第二的感受是，C/C++下的程序设计比 JAVA,C#高级语言的程序开发要更基本。程序员必须充分了解程序从什么时候申请了空间，并明白应什么时候回收。虽然 C/C++指针的操作，给程序设计带来了一定难度。但熟练之后，这个难度不会妨碍编程效率，反而会因为程序员深入理解了内存的分配，而使得程序的可靠性和可维护性更强。

一点遗憾，在数据结构中已经设计了用户对文件的访问权限管理。但由于时间问题没有及时实现。

## 九、参考文献

- [1] William Stallings 编，《操作系统的设计与精髓（6th）》，2011 年 4 月，电子工业出版社
- [2]胡明庆，高巍，钟梅编著，《操作系统教程与实验》，2007 年 1 月，清华大学出版社