《操作系统综合实验》

实验报告

05 文件系统

姓 名　王 栗 政
班 级　2018039
学 号　20179100018
教 师　冯鹏斌

实验日期　2022.5.21

一、实验题目

The following exercise examines the relationship between files and inodes on a UNIX or Linux system. On these systems, files are represented with inodes. That is, an inode is a file (and vice versa). You can complete this exercise on the Linux virtual machine that is provided with this text. You can also complete the exercise on any Linux, UNIX, or Mac OS X system, but it will require creating two simple text files named file1.txt and file3.txt whose contents are unique sentences.

二、相关原理与知识

1. 硬链接和软链接

硬链接

- 具有相同inode节点号的多个文件互为硬链接文件；
- 删除硬链接文件或者删除源文件任意之一，文件实体并未被删除；
- 只有删除了源文件和所有对应的硬链接文件，文件实体才会被删除；
- 硬链接文件是文件的另一个入口；
- 可以通过给文件设置硬链接文件来防止重要文件被误删；
- 创建硬链接命令 ln 源文件 硬链接文件；
- 硬链接文件是普通文件，可以用rm删除；
- 对于静态文件（没有进程正在调用），当硬链接数为0时文件就被删除。注意：如果有进程正在调用，则无法删除或者即使文件名被删除但空间不会释放。

软链接

- 软链接类似windows系统的快捷方式；
- 软链接里面存放的是源文件的路径，指向源文件；
- 删除源文件，软链接依然存在，但无法访问源文件内容；
- 软链接失效时一般是白字红底闪烁；
- 创建软链接命令 ln -s 源文件 软链接文件；
- 软链接和源文件是不同的文件，文件类型也不同，inode号也不同；
- 软链接的文件类型是“l”，可以用rm删除。

区别

原理上，硬链接和源文件的inode节点号相同，两者互为硬链接。软连接和源文件的inode节点号不同，进而指向的block也不同，软连接block中存放了源文件的路径名。

实际上，硬链接和源文件是同一份文件，而软连接是独立的文件，类似于快捷方式，存储着源文件的位置信息便于指向。

使用限制上，不能对目录创建硬链接，不能对不同文件系统创建硬链接，不能对不存在的文件创建硬链接；可以对目录创建软连接，可以跨文件系统创建软连接，可以对不存在的文件创建软连接。

2. ln命令相关知识

ln命令 用来为文件创建链接，链接类型分为硬链接和符号链接两种，默认的链接类型是硬链接。如果要创建符号链接必须使用"-s"选项。

```
    ln [选项]... [-T] 目标 链接名  (第一种格式)
     或: ln [选项]... 目标     (第二种格式)
     或: ln [选项]... 目标... 目录  (第三种格式)
     或: ln [选项]... -t 目录 目标... (第四种格式)
选项:
     --backup[=CONTROL]       # 为每个已存在的目标文件创建备份文件
     -b                       # 类似--backup，但不接受任何参数
     -d, -F, --directory          # 创建指向目录的硬链接(只适用于超级用户)
     -f, --force                  # 强行删除任何已存在的目标文件
     -i, --interactive            # 覆盖既有文件之前先询问用户
     -L, --logical                # 取消引用作为符号链接的目标
     -n, --no-dereference         # 把符号链接的目的目录视为一般文件
     -P, --physical               # 直接将硬链接到符号链接
     -r, --relative               # 创建相对于链接位置的符号链接
     -s, --symbolic               # 对源文件建立符号链接，而非硬链接
     -S, --suffix=SUFFIX          # 用"-b"参数备份目标文件后，备份文件的字尾会被加上一个备份字符
串，预设的备份字符串是符号“~”，用户可通过“-S”参数来改变它
     -t, --target-directory=DIRECTORY # 指定要在其中创建链接的DIRECTORY
     -T, --no-target-directory    # 将“LINK_NAME”视为常规文件
     -v, --verbose                # 打印每个链接文件的名称
     --help      # 显示此帮助信息并退出
     --version   # 显示版本信息并退出
```

3. inode相关知识

文件储存在硬盘上，硬盘的最小存储单位叫做"扇区"（Sector）。每个扇区储存512字节（相当于0.5KB）。操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个"块"（block）。这种由多个扇区组成的"块"，是文件存取的最小单位。"块"的大小，最常见的是4KB，即连续八个sector组成一个 block。

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做inode，中文译名为"索引节点"。

每一个文件都有对应的inode，里面包含了与该文件有关的一些信息。

inode包含文件的元信息，具体来说有以下内容：

* 文件的字节数
* 文件拥有者的User ID
* 文件的Group ID
* 文件的读、写、执行权限
* 文件的时间戳，共有三个：ctime指inode上一次变动的时间，mtime指文件内容上一次变动的时间，atime指文件上一次打开的时间。
* 链接数，即有多少文件名指向这个inode
* 文件数据block的位置

可以用stat命令，查看某个文件的inode信息：

```
$ stat file_name
```

总之，除了文件名以外的所有文件信息，都存在inode之中。

inode也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。一个是数据区，存放文件数据；另一个是inode区（inode table），存放inode所包含的信息。

每个inode节点的大小，一般是128字节或256字节。inode节点的总数，在格式化时就给定，一般是每1KB或每2KB就设置一个inode。假定在一块1GB的硬盘中，每个inode节点的大小为128字节，每1KB就设置一个inode，那么inode table的大小就会达到128MB，占整块硬盘的12.8%。

查看每个硬盘分区的inode总数和已经使用的数量，可以使用df命令。
查看每个inode节点的大小，可以用如下命令：
$ sudo dumpe2fs -h /dev/hda | grep "Inode size"
由于每个文件都必须有一个inode，因此有可能发生inode已经用光，但是硬盘还未存满的情况。这时，就无法在硬盘上创建新文件。

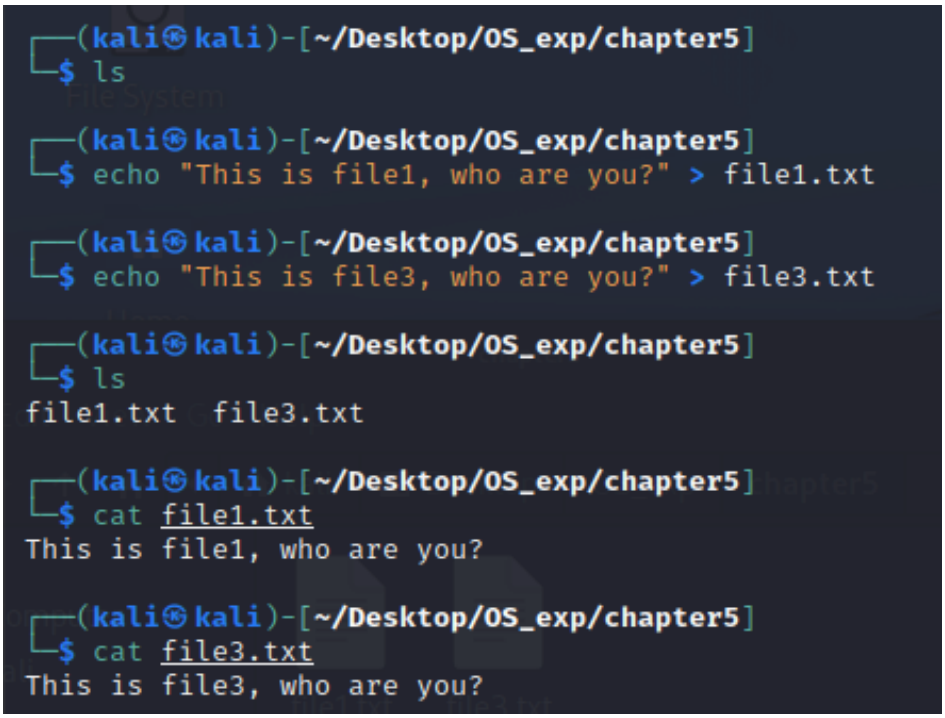
用户通过文件名，打开文件，实际上，系统内部这个过程分成三步：首先，系统找到这个文件名对应的inode号码；其次，通过inode号码，获取inode信息；最后，根据inode信息，找到文件数据所在的block，读出数据。

Unix/Linux系统中，目录（directory）也是一种文件。打开目录，实际上就是打开目录文件。
目录文件的结构非常简单，就是一系列目录项（dirent）的列表。每个目录项，由两部分组成：所包含文件的文件名，以及该文件名对应的inode号码。

由于inode号码与文件名分离，这种机制导致了一些Unix/Linux系统特有的现象。
1．有时，文件名包含特殊字符，无法正常删除。这时，直接删除inode节点，就能起到删除文件的作用。
2．移动文件或重命名文件，只是改变文件名，不影响inode号码。
3．打开一个文件以后，系统就以inode号码来识别这个文件，不再考虑文件名。因此，通常来说，系统无法从inode号码得知文件名。
第3点使得软件更新变得简单，可以在不关闭软件的情况下进行更新，不需要重启。因为系统通过inode号码，识别运行中的文件，不通过文件名。更新的时候，新版文件以同样的文件名，生成一个新的inode，不会影响到运行中的文件。等到下一次运行这个软件的时候，文件名就自动指向新版文件，旧版文件的inode则被回收。

三、实验过程

首先，按照题目要求创建`file1.txt`和`file3.txt`两个文件，并写入不同的内容。



查看`file1.txt`与`file3.txt`的inode号分别为2097228和2097229

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls -li
total 8
2097228 -rw-r--r-- 1 kali kali 28 May  9 02:35 file1.txt
2097229 -rw-r--r-- 1 kali kali 28 May  9 02:35 file3.txt
```

之后通过`ln`创建`file1.txt`的硬链接`file2.txt`，查看inode号可以发现file1.txt与file2.txt相同，均为2097228

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ln file1.txt file2.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls -li
total 12
2097228 -rw-r--r-- 2 kali kali 28 May  9 02:35 file1.txt
2097228 -rw-r--r-- 2 kali kali 28 May  9 02:35 file2.txt
2097229 -rw-r--r-- 1 kali kali 28 May  9 02:35 file3.txt
```

查看`file2.txt`的内容发现和`file1.txt`内容相同，之后改变`file2.txt`的内容，发现`file1.txt`的内容也被改变，这是由于硬链接的各个文件可以看作同一文件的不同入口。

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file2.txt
This is file1, who are you?

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ echo "This is file2, who are you?" > file2.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file2.txt
This is file2, who are you?

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file1.txt
This is file2, who are you?
```

删除`file1.txt`，发现`file2.txt`的inode和内容都没有发生改变

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ rm file1.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls -li
total 8
2097228 -rw-r--r-- 1 kali kali 28 May  9 02:42 file2.txt
2097229 -rw-r--r-- 1 kali kali 28 May  9 02:35 file3.txt
```

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file2.txt
This is file2, who are you?
```

之后删除`file2.txt`并用`strace`查看相关系统调用，可以看到`newfstatat`获取文件状态，之后使用`faccessat2`获取对该文件的权限，最后使用`unlinkat`将文件删除。（最后貌似使用`lseek`将指针指向标准输入的当前位置，但是返回值为-1，报错非法seek）

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ strace rm file2.txt
execve("/usr/bin/rm", ["rm", "file2.txt"], 0x7ffe81fc08d8 /* 54 vars */) = 0
brk(NULL)                               = 0x55f679676000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=70851, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 70851, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7eff4195b000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000y\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\200\0\300\4\0\0\0\1\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\320\276\243\212\v\307^\t\263h8\371\266h\r\350"..., 68, 880) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1835120, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7eff41959000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 1868664, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7eff41790000
mprotect(0x7eff417b6000, 1654784, PROT_NONE) = 0
mmap(0x7eff417b6000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7eff417b6000
mmap(0x7eff418fe000, 307200, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16e000) = 0x7eff418fe000
mmap(0x7eff4194a000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b9000) = 0x7eff4194a000
mmap(0x7eff41950000, 33656, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7eff41950000
close(3)                                = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7eff4178e000
arch_prctl(ARCH_SET_FS, 0x7eff4195a5c0) = 0
mprotect(0x7eff4194a000, 12288, PROT_READ) = 0
mprotect(0x55f67819f000, 4096, PROT_READ) = 0
mprotect(0x7eff4199c000, 8192, PROT_READ) = 0
munmap(0x7eff4195b000, 70851)           = 0
brk(NULL)                               = 0x55f679676000
brk(0x55f679697000)                     = 0x55f679697000
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=3041456, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 3041456, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7eff414a7000
close(3)                                = 0
ioctl(0, TCGETS, {B38400 opost isig icanon echo ...}) = 0
newfstatat(AT_FDCWD, "file2.txt", {st_mode=S_IFREG|0644, st_size=28, ...}, AT_SYMLINK_NOFOLLOW) = 0
geteuid()                               = 1000
newfstatat(AT_FDCWD, "file2.txt", {st_mode=S_IFREG|0644, st_size=28, ...}, AT_SYMLINK_NOFOLLOW) = 0
faccessat2(AT_FDCWD, "file2.txt", W_OK, AT_EACCESS) = 0
unlinkat(AT_FDCWD, "file2.txt", 0)      = 0
lseek(0, 0, SEEK_CUR)                   = -1 ESPIPE (Illegal seek)
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

使用`ln -s`命令创建`file3.txt`的软链接`file4.txt`，查看inode发现`file4.txt`的inode与`file3.txt`并不相同

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ln -s file3.txt file4.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls
file3.txt  file4.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls -li
total 4
2097229 -rw-r--r-- 1 kali kali 28 May  9 02:35 file3.txt
2097228 lrwxrwxrwx 1 kali kali  9 May  9 02:46 file4.txt → file3.txt
```

之后我们修改`file4.txt`的内容，发现`file3.txt`的内容也被修改

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ echo "This is file4, who are you?" > file4.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file4.txt
This is file4, who are you?

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file3.txt
This is file4, who are you?
```

删除`file3.txt`，查看`file4.txt`发现其变为红色

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ rm file3.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls -li
total 0
2097228 lrwxrwxrwx 1 kali kali 9 May  9 02:46 file4.txt → file3.txt
```

查看`file4.txt`的内容，发现没有报错没有该文件，说明软链接所创建的是源文件的一个快捷方式，软链接中存放的是源文件的路径，修改软链接可以对源文件造成修改。

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file4.txt
cat: file4.txt: No such file or directory
```

之后向`file4.txt`中写入内容，查看当前文件夹下的文件，发现`file3.txt`被重新创建，inode仍然与`file4.txt`不同，且内容与`file4.txt`相同，符合了我们上面的判断，`file4.txt`中存放的是`file3.txt`的路径。

```
┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ echo "This is file4, who are you?" > file4.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls
file3.txt  file4.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ ls -li
total 4
2097229 -rw-r--r-- 1 kali kali 28 May  9 02:53 file3.txt
2097228 lrwxrwxrwx 1 kali kali  9 May  9 02:46 file4.txt → file3.txt

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file4.txt
This is file4, who are you?

┌──(kali㉿kali)-[~/Desktop/OS_exp/chapter5]
└─$ cat file3.txt
This is file4, who are you?
```

四、实验结果与分析

在本次实验中，我通过实践体会到了硬链接和软链接之间的区别，学习到了Linux下硬链接和软链接背后的原理，体会到了Linux文件系统的优雅。

硬链接相当于一个文件的不同入口，每次创建硬链接会使inode节点中的“链接数”加1，而每删除一个该inode对应的文件名就会使该inode节点中的“链接数”减1，当这个值减到0，就表明没有文件名指向这个inode，系统就会回收这个inode号码，以及所对应的block区域。在创建目录时，默认会生成两个目录项："."和".."。前者的inode号码就是当前目录的inode号码，等同于当前目录的"硬链接"；后者的inode号码就是当前目录的父目录的inode号码，等同于父目录的"硬链接"。所以，任何一个目录的"硬链接"总数，总是等于2加上它的子目录总数（含隐藏目录）。并且修改硬链接的权限会修改该inode对应的所有文件名的权限，对应了文件系统内部使用的是inode号而不是文件名。

软链接中存放的是源文件的路径，读取软链接时，系统会自动将访问者导向源文件，因此无论打开的是哪一个软链接，最终读取的都是源文件，并且软链接依赖于源文件而存在。在修改软链接权限时，软链接的权限并不会发生更改，而是会对源文件的权限进行更改。

五、问题总结

本实验相对比较基础，未曾遇到什么问题

六、参考资料

[1] 理解inode https://www.ruanyifeng.com/blog/2011/12/inode.html

[2] linux文件管理（inode、文件描述符表、文件表）https://blog.csdn.net/wwwlyj123321/article/details/100298377

[3] Linux硬链接和软连接的区别与总结 https://xzchsia.github.io/2020/03/05/linux-hard-soft-link/

[4] Linux软连接和硬链接 https://www.cnblogs.com/itech/archive/2009/04/10/1433052.html