Linux Kernel Project 4: File System

薛春宇 518021910698

---

1 实验要求

以 *Linux* 内核中的 `/fs/romfs` 作为文件系统源码修改的基础，实现以下功能，其中 `romfs.ko` 接受三种参数：`hided_file_name`, `encrypted_file_name` 和 `exec_file_name`：

- `hided_file_name=xxx`：隐藏名字为 `xxx` 的文件或路径；
- `encrypted_file_name=xxx`：加密名字为 `xxx` 的文件中的内容；
- `exec_file_name=xxx`：修改名字为 `xxx` 的文件的权限为可执行。

上述功能通过生成并挂载一个格式为 `romfs` 的镜像文件 `test.img` 来检查，镜像文件可通过 `genromfs` 来生成。

---

2 实验环境

- *Linux OS* 版本：*Ubuntu 18.04*
- *Kernel* 版本：*Linux-5.5.11*
- *Gcc* 版本：*7.4.0*

---

3 实验内容

本节中，将分别介绍**文件隐藏**、**文件加密**和**文件权限修改**三种功能的具体实现，即涉及到的相关理论知识。本实验需要修改 *Linux* 内核中的 `/fs/romfs/super.c` 源码，重新编译生成 `romfs.ko` 模块并加载。

3.1 前期准备

在开始实验之前，我们需要对 *romfs* 的相关概念进行学习和了解。***romfs***（`ROM filesystem`）是一个缺乏许多功能的极为简单的文件系统，用途是将重要的文件存入 *EEPROM*。在 *Linux* 和其他一些类 *Unix* 系统上可以使用该文件系统。该文件系统非常适合用作初始化 `ROM` 内核驻留模块，具有体积小、可靠性好、读取速度快等优点，可以在需要时才加载。使用 *romfs* 文件系统可以构造出一个最小的内核，节省内存。

此外，我们对 `romfs.ko` 模块的实现进行一些基本的配置。由于 `romfs.ko` 模块共接收三个字符串类型的参数，我们需要 `super.c` 中进行相应的参数声明。注意，使用 `module_param()` 函数需要引入 `<linux/moduleparam.h>` 头文件。

```
static char *hided_file_name;
static char *encrypted_file_name;
static char *exec_file_name;

module_param(hided_file_name, charp, 0644);
module_param(encrypted_file_name, charp, 0644);
module_param(exec_file_name, charp, 0644);
```

由于 `romfs` 镜像文件的生成需要使用 `genromfs` 指令，我们还需要在 `root` 权限下使用 `apt-get install genromfs` 指令安装相关工具。

3.2 功能一：文件隐藏

3.2.1 目录项文件名的读取

在 `/fs/romfs/super.c` 文件中，**`romfs` 通过 `romfs_readdir()` 函数 (*Line 199 ~ 262*) 将目录文件读入到内核空间，并进行相应目录项的填充**。该函数中进行如下用于文件名读取的迭代过程：

- 利用 `romfs_dev_read()` 函数读取当前目录项的 `inode` 信息并存入 `struct romfs_inode ri`，后者是 `rom` 文件系统下的索引节点结构体，在 `linux/romfs_fs.h` 头文件中定义：

```
struct romfs_inode {
    _be32 next;             // 指向下一个inode的指针
    _be32 spec;             // 存储与目录、硬链接、设备文件相关的信息
    _be32 size;             // 文件大小
    _be32 checksum;         // 校验和
    char name[0];           // 文件名，仅1个字节
}
```

- 利用 `romfs_dev_strnlen()` 函数从 `inode` 信息中计算文件名长度，并修改 `fsname` 的 `size`；
- 再次利用 `romfs_dev_read()` 函数，根据文件名长度获取文件名 `fsname` 等信息；
- 利用 `ri` 结构体中的 `next` 指针继续访问下一目录项，直到完成当前目录下全部目录项的访问。

注意到，在上述迭代过程中，`romfs_dev_read()` 函数被用于从 `romfs image` 中读取数据，在 `/fs/romfs/storage.c` 中定义；`romfs_dev_strlen()` 函数同样定义在 `storage.c` 中，作用是根据超级块的 `inode` 信息修改缓冲区的大小。

3.2.2 目录项的读取和填充

`romfs_readdir()` 函数调用 `dir_emit()` 函数来进行目录项的读取以及填充，该函数定义在 `fs.h` 头文件中：

```
static inline bool dir_emit(struct dir_context *ctx,
          const char *name, int namelen,
          u64 ino, unsigned type)
{
  return ctx->actor(ctx, name, namelen, ctx->pos, ino, type) == 0;
}
```

其中涉及到目录结构体指针 `dir_context *ctx` 及其 `actor` 字段的使用，该结构体同样定义在 `fs.h` 文件中，目录填充实际上由 `filldir_t` 类型的字段 `actor` 完成。`filldir_t` 用于内核指定目录项布局，并将目录读入内核空间。

```
struct dir_context;
typedef int (*filldir_t)(struct dir_context *, const char *, int, loff_t, u64,
       unsigned);
struct dir_context {
  filldir_t actor;
  loff_t pos;
};
```

3.2.3 文件隐藏功能的实现

如 *3.2.1* 节所示，为了隐藏某一文件，我们需要在循环中遍历所有文件的文件名，通过字符串匹配函数 `strcmp()` 来判断是否为目标文件。该字符串匹配函数需要在 *3.2.2* 节中的 `dir_emit()` 函数之前被调用，以实现特定文件的隐藏。

完整代码解析见 *Appendix A* 中的 `romfs_readdir()` 函数。

```
    // If fsname == hided_file_name, jump to file_hided point
    // Implemented by Chunyu Xue
    if (hided_file_name && !strcmp(hided_file_name, fsname))
        goto file_hided;

    if ((nextfh & ROMFH_TYPE) == ROMFH_HRD)
        ino = be32_to_cpu(ri.spec);
        // dir_emit(): 目录填充函数，将当前文件的相关信息填充在目录中
        if (!dir_emit(ctx, fsname, j, ino, romfs_dtype_table[nextfh & ROMFH_TYPE]))
            goto out;
// Jump here
file_hided:
        offset = nextfh & ROMFH_MASK;
}
// ...
```

3.3 功能二：文件加密

3.3.1 文件内容读取机制

在 `/fs/romfs/super.c` 文件中，**`romfs` 通过 `romfs_readpage()` 函数读取文件的内容**。`romfs_readpage()` 函数首先获得指向当前页对应索引节点的指针 `*inode`，然后将页通过 `kmap()` 映射到 `buf` 中，通过 `page_offset()` 和 `i_size_read()` 函数分别获得页偏移量和 `inode` 的大小。

根据 `offset` 和 `size` 的关系，我们需要调整写入页的数据大小。若 `offset < size`，即 `inode` 并未被完全写入页面，则计算未被写入的页数据大小。需要注意的是，由于一次只能对一个页进行写入，因此 `fillsize` 最多为 `PAGE_SIZE`。随后，我们获得 `inode` 在 `super block` 上的偏移量 `pos`，并通过 `romfs_dev_read()` 从 `inode` 中读取数据写入 `buf`。

```
1  static int romfs_readpage(struct file *file, struct page *page)
2  {
3      struct inode *inode = page->mapping->host;                  // 获取inode指针
4      // ...
5      buf = kmap(page);                                           // 将页映射到buf中
6      if (!buf)
7          return -ENOMEM;
8      /* 32 bit warning -- but not for us :) */
9      offset = page_offset(page);                                 // 获取页偏移量
10     size = i_size_read(inode);                                  // 获取inode大小
11     // ...
12     if (offset < size) {
13         size -= offset;                                         // 计算未被写入的页
   数据大小
14         fillsize = size > PAGE_SIZE ? PAGE_SIZE : size;         // 确定填充大小
15         pos = ROMFS_I(inode)->i_dataoffset + offset;            // 获取文件内容的偏
   移位置
16         ret = romfs_dev_read(inode->i_sb, pos, buf, fillsize);  // 将数据读入buf
17         // ...
18     }
19     // ...
20 }
```

3.3.2 目录项文件名的获取

实现文件加密，我们首先需要获取当前目录项对应的文件名，进而通过字符串匹配针对指定文件存放在缓冲区中的内容进行加密，获取方式可以参考 `romfs_readdir()` 函数的实现。唯一的问题是，在 `romfs_readdir()` 中我们直接从文件结构体指针 `struct file *file` 中利用 `file_inode()` 获得了当前目录项的 `inode`，直接读出文件名的 `offset`，进而利用 `romfs_dev_read()` 函数读取文件名；而**在 `romfs_readpage()` 中，却很难获得该文件名的偏移量**。

为了解决上述问题，我们注意到在 `/fs/romfs/internal.h` 头文件中定义了 `romfs_node_info` 数据结构，其字段包括表示非数据区域大小的 `i_metasize`，以及表示从 `fs` 开头到该 `inode` 偏移量的 `i_dataoffset`。分析字段含义可知，**文件名的 `offset` 可以通过 `inode` 偏移量减去非数据区域大小获得**。

```c
struct romfs_inode_info {
    struct inode  vfs_inode;
    unsigned long i_metasize; /* size of non-data area */
    unsigned long i_dataoffset; /* from the start of fs */
};
```

那么，我们应该如何得到一个文件对应的 `romfs_inode_info` 结构体呢？**通过 `internal.h` 中定义的 `struct romfs_inode_info *ROMFS_I()` 函数，我们得以将 `inode` 类型的指针转换为 `romfs_inode_info` 类型的指针**，进而通过 `inode->i_dataoffset - inode->i_metasize` 获得 `offset`。

```c
static inline struct romfs_inode_info *ROMFS_I(struct inode *inode) {
    return container_of(inode, struct romfs_inode_info, vfs_inode);
}
```

在获得文件名的 `offset` 之后，即可按照上述方案获得当前目录项的文件名。为此，**我们添加了 `get_file_name()` 函数来实现获取文件名的功能**。需要特别注意的是，C 语言中需要将待返回的字符串设置为静态变量，否则在将该字符串的地址返回后，该函数会自动调用内存释放函数清除这一部分的内存，因此无法在函数外访问该局部字符串变量。

```c
// Get file name, implemented by Chunyu Xue
static char* get_file_name(struct inode *i) {
    unsigned long offset;
    int j, ret;
    static char fsname[ROMFS_MAXFN];          // 注意，fsname要设置为静态变量，否则在返回地址后会被释放
    struct romfs_inode_info *inode;

    // 将inode结构体指针转换为romfs_inode_info类型的指针
    inode = ROMFS_I(i);
    // 成员变量相减获得offset
    offset = (inode->i_dataoffset) - (inode->i_metasize);
    // 获取文件名的长度
    j = romfs_dev_strnlen(i->i_sb, offset + ROMFH_SIZE, sizeof(fsname) - 1);
    if (j < 0)
        return false;
    // 从设备中读取文件名
    ret = romfs_dev_read(i->i_sb, offset + ROMFH_SIZE, fsname, j);
    if (ret < 0)
        return false;
    // 添加文件名结束符
    fsname[j] = '\0';
    return fsname;
}
```

3.3.3 文件加密的实现

如 *3.2.1* 节所示，在 `romfs_readpage()` 函数将数据从 `inode` 读入 `buf` 之后，我们利用 `strcmp()` 函数来对当前目录项的文件名 `get_file_name(inode)` 和目标文件名 `encrypted_file_name` 进行比对，若相同，则对该文件在 `buf` 中的数据进行加密。加密操作需要使用 `char*` 类型的缓冲区指针，用来修改缓冲区的内容。综合上述过程的完整代码解析见 *Appendix B* 中的 `get_file_name()` 函数和 `romfs_readpage()` 函数。

```
static int romfs_readpage(struct file *file, struct page *page)
{
    // 缓冲区指针，用于修改缓冲区内容
    char* bufp;
    // ...
    if (offset < size) {
        // 将数据从inode读入buf

        // Whether fsname == encrypted_file_name, implemented by Chunyu Xue
        if ((strcmp(get_file_name(inode), encrypted_file_name) == 0) && fillsize > 0) {
            // (If equals) Encrypt the content
            bufp = (char*)buf;
            int i;
            for (i = 0; i < fillsize; i ++) {
                *bufp = *bufp + 1;
                bufp++;
            }
        }
   }
   // ...
}
```

3.4 功能三：文件权限修改

3.4.1 *inode* 对象的读取

在 `/fs/romfs/super.c` 文件中，**`romfs` 通过 `romfs_lookup()` 函数在目录中查找 `entry`，该函数通过调用 `romfs_iget()` 来根据 `inode` 的偏移量从镜像文件中读取 `inode` 对象**。注意到，在读取 `inode` 之前，`romfs_lookup()` 会从目录项中获取文件名 `name`，因此本模块中不需要自己实现获取文件名的函数。

```
static struct dentry *romfs_lookup(struct inode *dir, struct dentry *dentry,
            unsigned int flags) {
    // ...
    maxoff = romfs_maxsize(dir->i_sb);
    offset = be32_to_cpu(ri.spec) & ROMFH_MASK;
    name = dentry->d_name.name;                                    // 获取文件名
    len = dentry->d_name.len;                                      // 获取文件名长度
    for (;;) {
```

```
        // ...
        /* try to match the first 16 bytes of name */
        ret = romfs_dev_strcmp(dir->i_sb, offset + ROMFH_SIZE, name, len);
        if (ret < 0) goto error;
        if (ret == 1) {
            /* Hard link handling */
            if ((be32_to_cpu(ri.next) & ROMFH_TYPE) == ROMFH_HRD)
                offset = be32_to_cpu(ri.spec) & ROMFH_MASK;             // 获取inode的offset
            inode = romfs_iget(dir->i_sb, offset);                      // 根据offset从image中读取inode
        }
        /* next entry */
        offset = be32_to_cpu(ri.next) & ROMFH_MASK;
    }
    // ...
}
```

3.4.2 文件权限的设置

观察 `fs.h` 函数中 `inode` 结构体的成员定义可知，`inode` 通过 `i_mode` 字段来定义文件权限。因此我们需要在 `romfs_lookup()` 函数读取 `inode` 之后对该字段进行修改。阅读 `stat.h` 中文件权限的宏定义后可知，`S_IXUGO` 宏将文件的用户、组用户和其他用户权限均设置为 *1*，即可执行权限。因此，我们只需要**将 `inode` 中的 `i_mode` 字段与代表可执行权限的宏 `S_IXUGO` 进行按位或操作，即可完成文件可执行权限的修改**。

```
struct inode {
    umode_t     i_mode;
    unsigned short    i_opflags;
    kuid_t      i_uid;
    kgid_t      i_gid;
    unsigned int    i_flags;
    // ...
}
```

3.4.3 文件权限修改的实现

如 *3.4.1* 和 *3.4.2* 节所示，每一次 `for` 循环的迭代过程中，我们需要在 `inode` 被读入之后利用 `strcmp()` 函数对当前目录项的文件名 `name` 和目标文件名 `exec_file_name` 进行比对，若相同，则将该文件 `inode` 的 `i_mode` 字段与 `S_IXUGO` 宏按位相或，即可将 `user`、`group` 和 `other` 均设置为具有可执行权限。综合上述过程的完整代码解析见 *Appendix C* 中的 `romfs_lookup()` 函数。

```
static struct dentry *romfs_lookup(struct inode *dir, struct dentry *dentry,
           unsigned int flags)
{
    const char *name;   /* got from dentry */
    // 获取文件名
    name = dentry->d_name.name;
    // ...
```

```
    for (;;) {
        // ...
        if (ret == 1) {
          // 根据offset从image中读取inode

          // If name == exec_file_name, modify the permission of this file into
only executable
          // Implemented by Chunyu Xue
          if (exec_file_name && !strcmp(exec_file_name, name))
            inode->i_mode |= S_IXUGO;
          break;
        }
        // ...
    }
    // ...
```

4 实验结果及分析

4.1 `Makefile` 编译及测试文件的准备

编写如下 `Makefile`，并使用 `make` 指令进行编译：

```
obj-$(CONFIG_ROMFS_FS) += romfs.o
romfs-y := storage.o super.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

ifneq ($(CONFIG_MMU),y)
romfs-$(CONFIG_ROMFS_ON_MTD) += mmap-nommu.o
endif

all:
  make -C $(KDIR) M=$(PWD) modules
clean:
  make -C $(KDIR) M=$(PWD) clean
```

```
root@linux-kernel-server:~/LinuxKernel/Project4# ls
internal.h  Kconfig  Makefile  mmap-nommu.c  storage.c  super.c  test
root@linux-kernel-server:~/LinuxKernel/Project4# make
make -C /lib/modules/5.5.11/build M=/root/LinuxKernel/Project4 modules
make[1]: Entering directory '/usr/src/linux-5.5.11'
  CC [M]  /root/LinuxKernel/Project4/storage.o
  CC [M]  /root/LinuxKernel/Project4/super.o
  LD [M]  /root/LinuxKernel/Project4/romfs.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /root/LinuxKernel/Project4/romfs.mod.o
  LD [M]  /root/LinuxKernel/Project4/romfs.ko
make[1]: Leaving directory '/usr/src/linux-5.5.11'
root@linux-kernel-server:~/LinuxKernel/Project4#
```

使用 `mkdir test` 创建测试文件夹，并在其中使用 `touch` 指令创建 `aa`、`bb` 和 `ft` 三个文件，分别对应文件隐藏、文件加密和文件权限修改三种操作。查看文件初始权限如下，注意到均无可执行权限。

```
root@linux-kernel-server:~/LinuxKernel/Project4# ls -l test
total 12
-rw-r--r-- 1 root root 7 May 31 21:36 aa
-rw-r--r-- 1 root root 8 May 31 21:36 bb
-rw-r--r-- 1 root root 7 May 31 21:36 ft
```

4.2 镜像的生成、挂载及模块载入

分别使用如下指令进行 `romfs` 镜像的生成、`romfs.ko` 模块的载入以及镜像文件的挂载（需要首先创建 `mnt` 文件夹用于挂载）。

```
genromfs –V "vromfs" –f test.img –d test
insmod romfs.ko hided_file_name=aa encrypted_file_name=bb exec_file_name=ft
mount -o loop test.img mnt
```

4.3 结果分析

使用 `ls -l mnt` 指令查看挂载文件夹下的内容，观察到 `aa` 文件被隐藏，`ft` 文件增加了可执行权限。

```
root@linux-kernel-server:~/LinuxKernel/Project4# genromfs -V "vromfs" -f test.img -d test
root@linux-kernel-server:~/LinuxKernel/Project4# insmod romfs.ko hided_file_name=aa encrypted_file_name=bb exec_file_name=ft
root@linux-kernel-server:~/LinuxKernel/Project4# mount -o loop test.img mnt
root@linux-kernel-server:~/LinuxKernel/Project4# ls -l mnt
total 0
-rw-r--r-- 1 root root 8 Jan  1  1970 bb
-rwxr-xr-x 1 root root 7 Jan  1  1970 ft
root@linux-kernel-server:~/LinuxKernel/Project4#
```

分别使用 `cat /test/bb` 和 `cat /mnt/bb` 指令查看原始 `bb` 文件和镜像 `bb` 文件的内容，发现所有字符全都进行了 *+1* 操作。

```
root@linux-kernel-server:~/LinuxKernel/Project4# cat test/bb
123abc

root@linux-kernel-server:~/LinuxKernel/Project4# cat mnt/bb
234bcd
```

至此，实验内容全部完成。

---

5 实验心得

本次实验是 *Linux* 内核课程的第四次正式 *project*，旨在掌握面向 *Linux* 简单文件系统 `romfs` 的简单操作，包括文件隐藏、文件加密和文件权限修改。在文件隐藏功能的实现过程中，我通过阅读源码的方式深入分析了 `romfs` 对文件的管理方式及描述符 `romfs_inode`，并在初步掌握 `romfs_dev_read()`、`dir_emit` 和 `dir_context` 等函数/结构体的作用的基础上，厘清了文件名和目录项读取的逻辑关系。同时，在文件加密和权限修改功能的实现过程中，我分析了 `romfs` 的文件内容读取机制，掌握了利用 `romfs_inode_info` 描述符获得文件名 `offset` 的方法，并在掌握 `romfs` 读取 `inode` 的方法和相关文件权限定义的基础上，实现了文件权限的修改。

在实验过程中，我首先根据实验指导书上的提示，通过关键词匹配的方式分别找到了几个关键函数和结构体的源码实现，并依次进行调用函数的深入搜索、研究，最终细化到底层实现，力求完全掌握该结构体的功能及所调用函数的意义。由于目前网上针对 `romfs` 的教程大多质量参差不齐，且内核版本也浮动较大，因此我必须不断进行试错，不断修改报错的 *API* 或结构体类名。例如，在编写 `get_file_name(inode)` 函数的过程中，由于忽略了 *C* 语言针对局部变量自动回收空间的设置，在未将 `fsname` 设置为静态变量之前，我一直得不到正确的结果。这个问题较为隐蔽，经过一番搜索之后才得以发现。

本次实验中，我不仅更深一步地掌握了 *Linux* 源码阅读的方法，增强了自身阅读源码的能力，还提高了发现问题、解决问题的能力。希望能在下面的实验中也能收获满满！

Appendix A：文件隐藏函数实现

```
/*
 * read the entries from a directory
 */
static int romfs_readdir(struct file *file, struct dir_context *ctx)
{
    // ...
    for (;;) {
        // ...
        // 读取当前目录项的inode信息
        ret = romfs_dev_read(i->i_sb, offset, &ri, ROMFH_SIZE);
        if (ret < 0)
            goto out;
        // 从inode信息中获取文件名长度，即修改fsname的size
        j = romfs_dev_strnlen(i->i_sb, offset + ROMFH_SIZE, sizeof(fsname) - 1);
        if (j < 0)
            goto out;
        // 从romfs image中读取当前目录项的文件名等信息
        ret = romfs_dev_read(i->i_sb, offset + ROMFH_SIZE, fsname, j);
        if (ret < 0)
```

```
            goto out;
        // 为文件名添加结束符
        fsname[j] = '\0';
        ino = offset;
        nextfh = be32_to_cpu(ri.next);
        // If fsname == hided_file_name, jump to file_hided point
        // Implemented by Chunyu Xue
        if (hided_file_name && !strcmp(hided_file_name, fsname))
          goto file_hided;
        if ((nextfh & ROMFH_TYPE) == ROMFH_HRD)
          ino = be32_to_cpu(ri.spec);
        // dir_emit(): 目录填充函数，将当前文件的相关信息填充在目录中
        if (!dir_emit(ctx, fsname, j, ino,
              romfs_dtype_table[nextfh & ROMFH_TYPE]))
          goto out;
// Jump here
file_hided:
        offset = nextfh & ROMFH_MASK;
    }
out:
  return 0;
}
```

Appendix B：文件加密的实现

```
// Get file name, implemented by Chunyu Xue
static char* get_file_name(struct inode *i) {
    unsigned long offset;
    int j, ret;
    static char fsname[ROMFS_MAXFN];          // 注意，fsname要设置为静态变量，否则在返回地址后会被释放
    struct romfs_inode_info *inode;

    // 将inode结构体指针转换为romfs_inode_info类型的指针
    inode = ROMFS_I(i);
    // 成员变量相减获得offset
    offset = (inode->i_dataoffset) - (inode->i_metasize);
    // 获取文件名的长度
    j = romfs_dev_strnlen(i->i_sb, offset + ROMFH_SIZE, sizeof(fsname) - 1);
    if (j < 0)
        return false;
    // 从设备中读取文件名
    ret = romfs_dev_read(i->i_sb, offset + ROMFH_SIZE, fsname, j);
    if (ret < 0)
        return false;
    // 添加文件名结束符
    fsname[j] = '\0';
```

```
    return fsname;
}

// ...

/*
 * read a page worth of data from the image
 */
static int romfs_readpage(struct file *file, struct page *page)
{
    // 缓冲区指针，用于修改缓冲区内容
    char* bufp;
    // ...
    if (offset < size) {
        size -= offset;
        fillsize = size > PAGE_SIZE ? PAGE_SIZE : size;
        // 获取文件内容的偏移位置
        pos = ROMFS_I(inode)->i_dataoffset + offset;
        // 将数据读入buf
        ret = romfs_dev_read(inode->i_sb, pos, buf, fillsize);
        if (ret < 0) {
            SetPageError(page);
            fillsize = 0;
            ret = -EIO;
        }

        // Whether fsname == encrypted_file_name, implemented by Chunyu Xue
        if ((strcmp(get_file_name(inode), encrypted_file_name) == 0) && fillsize >
0) {
            // (If equals) Encrypt the content
            bufp = (char*)buf;
            int i;
            for (i = 0; i < fillsize; i ++) {
                *bufp = *bufp + 1;
                bufp++;
            }
        }
   }
   // ...
}
```

Appendix C：文件权限修改的实现

```
/*
 * look up an entry in a directory
 */
static struct dentry *romfs_lookup(struct inode *dir, struct dentry *dentry,
```

```
            unsigned int flags)
{
    const char *name;   /* got from dentry */
    // 获取文件名
    name = dentry->d_name.name;
    // ...
    for (;;) {
        // ...
        /* try to match the first 16 bytes of name */
        ret = romfs_dev_strcmp(dir->i_sb, offset + ROMFH_SIZE, name,
                   len);
        if (ret < 0)
          goto error;
        if (ret == 1) {
          /* Hard link handling */
          if ((be32_to_cpu(ri.next) & ROMFH_TYPE) == ROMFH_HRD)
            offset = be32_to_cpu(ri.spec) & ROMFH_MASK;
          // 读取inode
          inode = romfs_iget(dir->i_sb, offset);

          // If name == exec_file_name, modify the permission of this file into
only executable
          // Implemented by Chunyu Xue
          if (exec_file_name && !strcmp(exec_file_name, name))
            inode->i_mode |= S_IXUGO;
          break;
        }
        /* next entry */
        offset = be32_to_cpu(ri.next) & ROMFH_MASK;
    }
    // ...
}
```