

CS353: Linux kernel Project 4 实验报告

预备知识:

ROMFS相关知识

关于ROMFS最为权威的资料是内核源代码树下的“[Documentation/filesystems/romfs.txt](#)”，以下分析多数来自于该文件。

ROMFS是一种简单的只读文件系统，主要是用来当做初始文件系统来使用的。romfs的操作是基于块设备的，它的底层结构非常简单。为了快速访问，每个单元被设计为起始于16字节边界。一个最小的文件为32字节（文件内容为空，并且文件名长度小于16字节）。其具体的文件结构如下：

The layout of the filesystem is the following:

offset	content
--------	---------

0	- r o m \	
4	1 f s - /	The ASCII representation of those bytes (i.e. "-rom1fs-")
8	full size	The number of accessible bytes in this fs.
12	checksum	The checksum of the FIRST 512 BYTES.
16	volume name	The zero terminated name of the volume, padded to 16 byte boundary.
xx	file	
:	headers	:

File headers之前的字节由如下的数据结构来控制：include/linux/romfs_fs.h

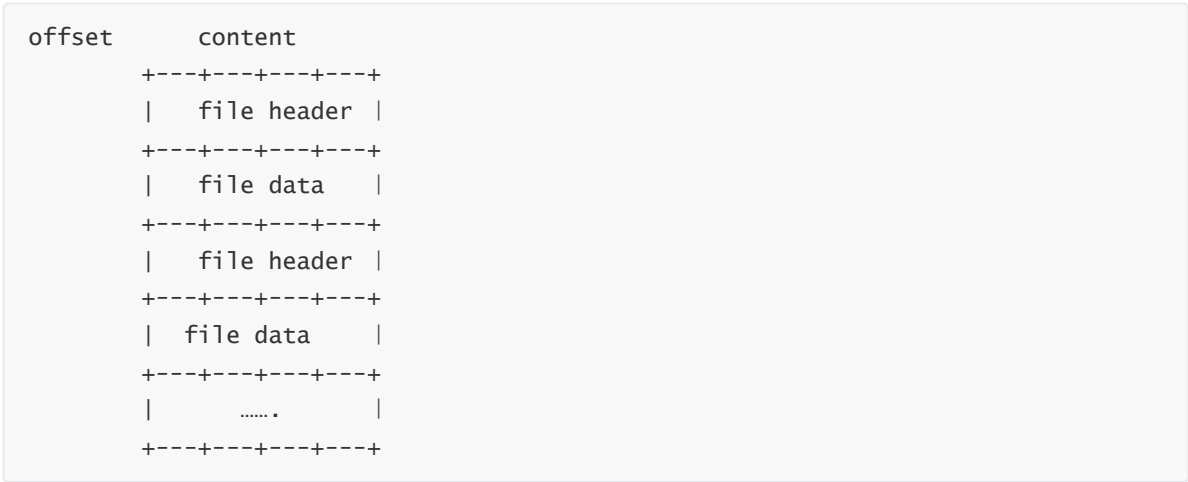
```
/* On-disk "super block" */
struct romfs_super_block {
    __be32 word0;
    __be32 word1;
    __be32 size;
    __be32 checksum;
    char name[0];          /* volume name */
};
```

(1) 这个数据结构中的word0和word1的是固定的值：“-rom1fs-”，由如下的宏定义说明：
include/linux/romfs_fs.h

```
#define ROMSB_WORD0 __mk4('-', 'r', 'o', 'm')
#define ROMSB_WORD1 __mk4('1', 'f', 's', '-')
```

- (2) 而size是对整个文件系统的大小的说明。
- (3) checksum是对前512个字节的校验和（如果小于512，就以实际大小计算）。
- (4) name是当前这个文件系统的名称。

在super_block之后，开始存放文件头（file header）和文件数据（file data）



File header的格式如下

offset	content	
	+---+---+---+---+	
0	next filehdr x	The offset of the next file header (zero if no more files)
	+---+---+---+---+	
4	spec.info	Info for directories/hard links/devices
	+---+---+---+---+	
8	size	The size of this file in bytes
	+---+---+---+---+	
12	checksum	Covering the meta data, including the file name, and padding
	+---+---+---+---+	
16	file name	The zero terminated name of the file,
:	:	padded to 16 byte boundary

在内核源代码中如下：include/linux/romfs_fs.h

```

/* On disk inode */
struct romfs_inode {
    __be32 next;          /* low 4 bits see ROMFH_ */
    __be32 spec;
    __be32 size;
    __be32 checksum;
    char name[0];
};

```

(1) 其中next的前面28位是指向下一个文件的地址，应为整个文件系统以16字节对齐，所以任何一个文件的起始地址的最后4位始终为“0”。而这最后的4位并没就此浪费，而是进行了新的利用，具体定义在include/linux/romfs_fs.h如下：

```

#define ROMFH_TYPE 7
#define ROMFH_HRD 0
#define ROMFH_DIR 1
#define ROMFH_REG 2
#define ROMFH_SYM 3
#define ROMFH_BLK 4
#define ROMFH_CHR 5
#define ROMFH_SCK 6
#define ROMFH_FIF 7
#define ROMFH_EXEC 8

```

(2) spec这个字段存放的是目录 / 硬链接 / 设备文件的相关信息：
这个域是文件类型相关的，也就是说对于不同的文件类型，这个域表示的含义是不一样的。

	mapping	spec.info means
0	hard link	link destination [file header]
1	directory	first file's header
2	regular file	unused, must be zero [MBZ]
3	symbolic link	unused, MBZ (file data is the link content)
4	block device	16/16 bits major/minor number
5	char device	- " -
6	socket	unused, MBZ
7	fifo	unused, MBZ

在本次实验中，主要用到的ri.spec域都是 directory类型的信息，用于指定第一个文件header的offset

- (3) size是这个文件的大小。
- (4) checksum这个域只是文件头和文件名的校验和。
- (5) name是文件的名称

具体实现：

首先，在super.c内核模块中，声明本次实验所需要的模块参数。

```

#include <linux/moduleparam.h>

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);

```

Task1 隐藏文件

- 实现思路

根据实验指导书的提示，我们通过修改 romfs_readdir() 函数来完成实验要求。阅读源码，我们发现该函数会循环更新offset，以此来遍历file文件下的所有子文件。在每一次循环中，通过 romfs_dev_read(i->i_sb, offset + ROMFH_SIZE, fsname, j) 获得子文件的文件名称。为隐藏特定文件，我们在获得文件名——fsname[] 后，我们会判断 fsname[] 是否等于模块参数

hided_file_name。当两者相同时，将会跳过 `dir_emit(ctx, fsname, j, ino, romfs_dtype_table[nextfh & ROMFH_TYPE])` 这一函数，避免向用户展示file目录下的特定文件。以此来实现特定文件的隐藏。

- 具体实现

注：具体的源码逻辑在下面添加了详细的代码注释，就不再赘述。

```
// 读取file文件下的所有文件名
static int romfs_readdir(struct file *file, struct dir_context *ctx)
{
    struct inode *i = file_inode(file);           // 获得file所对应的inode
    struct romfs_inode ri;                         // 存在磁盘上的inode信息
    unsigned long offset, maxoff;
    int j, ino, nextfh;
    char fsname[ROMFS_MAXFN];
    int ret;
    maxoff = romfs_maxsize(i->i_sb);               // 记录romfs所支持的最大偏移量
    offset = ctx->pos;                              // 记录当前的目录上下文的偏移量
    if (!offset) {                                  // 若当前目录下的偏移量为0
        offset = i->i_ino & ROMFH_MASK;             // inode在romfs image 中的offset,
        因为romfs是16byte对其的，所以低4
        位都是0，可以用于存储其他信息
        ret = romfs_dev_read(i->i_sb, offset, &ri, ROMFH_SIZE); // 从superblock的
        offset处读取ROMFH_SIZE大小（16
        byte）的内容存入romfs_inode中
        if (ret < 0)
            goto out;
        offset = be32_to_cpu(ri.spec) & ROMFH_MASK; // 对于目录文件(next & 0x1 =
        1)，这个域指向其内第一个文件的
        文件头在romfs映像中的偏移。
    }
    /* Not really failsafe, but we are read-only... */
    for (;;) {
        if (!offset || offset >= maxoff) {
            offset = maxoff;
            ctx->pos = offset;
            goto out;
        }
        ctx->pos = offset;                          // 保存现在romfs中的偏移量（现在
        的偏移量应该正指向文件的开头）

        /* Fetch inode info */
        ret = romfs_dev_read(i->i_sb, offset, &ri, ROMFH_SIZE); // 从offset处读取
        16byte存入romfs_inode中
        （romfs_inode的大小也是16byte）
        if (ret < 0)
            goto out;

        j = romfs_dev_strnlen(i->i_sb, offset + ROMFH_SIZE, // 文件名的起始地址=文
        件头的偏移量（offset）+
        romfs_inode的大小（16byte）。获得文件名长度
        sizeof(fsname) - 1);
        if (j < 0)
            goto out;
```

```

        ret = romfs_dev_read(i->i_sb, offset + ROMFH_SIZE, fsname, j); //获得文件名
        if (ret < 0)
            goto out;
        fsname[j] = '\0';
        ino = offset;
        nextfh = be32_to_cpu(ri.next); //ri.next字段指向下一个文件的起始地址

        if (strcmp(fsname, hided_file_name) != 0){ //若当前文件名称与hided_file_name相同，则跳过
            if ((nextfh & ROMFH_TYPE) == ROMFH_HRD)
                ino = be32_to_cpu(ri.spec);
            if (!dir_emit(ctx, fsname, j, ino, //将目录内容映射到用户空间
                romfs_dtype_table[nextfh & ROMFH_TYPE]))
                goto out;
        }
        offset = nextfh & ROMFH_MASK; //更新偏移量
    }
out:
    return 0;
}

```

Task2 文件加密

- **实现思路：**根据实验指导书的提示，我们可以修改 `romfs_readpage()` 函数来实现对特定文件的古典加密。`romfs_readpage()` 用于将romfs映像的内容读入内存页中。在判断是否要进行加密操作时，我们可以比较当前读取的文件的文件名是否与参数：`encrypted_file_name`一致。若两者相同，则可以改写buf中的内容，实现加密。

难点在于获得当前文件的文件名称。这里提供两种实验思路：

1. 直接使用函数参数的file指针获得文件名称。具体的调用路径如下：

```
file->f_path.dentry->d_name.name
```

参考链接：[In Linux, how can I get the filename from the "struct file" structure, while stepping thru the kernel with kgdb? - Stack Overflow](#)

2. 根据上文中的ROMFS的相关知识，我们了解了在romfs映像中文件的结构。每一个 `romfs_inode` 对应的文件名起始偏移量都在文件起始偏移量的16byte后。为获得文件的起始偏移量，我们首先使用 `ROMFS_I(inode)` 获得inode在内存中的数据结构 `romfs_inode_info`。用其 `dataoffset` 字段（起始数据的偏移量）减去 `metadatasize` 字段（元数据的大小），即可获得该文件在映像中的起始偏移量，之后仿照 `romfs_readdir()` 中的方法来获得文件名称。

从简便性的角度考虑，我选择了第一种实验思路。

- **具体实现**

注：具体的源码逻辑在下面添加了详细的代码注释，就不再赘述。

```

//将file文件从磁盘读入page中
static int romfs_readpage(struct file *file, struct page *page)
{
    struct inode *inode = page->mapping->host;
    loff_t offset, size;
    unsigned long fillsize, pos;
    void *buf;

```

```

int ret;

buf = kmap(page);          //将page映射到buffer上
if (!buf)
    return -ENOMEM;

offset = page_offset(page); //Return byte-offset into filesystem object for
page
size = i_size_read(inode);  //获得文件大小
fillsize = 0;
ret = 0;
if (offset < size) {        //offset < size 则说明该页应该读取文件
    size -= offset;         //还剩多少字节的文件需要读取
    fillsize = size > PAGE_SIZE ? PAGE_SIZE : size; //若剩余文件的大小大于一页，
    将fillsize设置为一页，否则fillsize=文件大小

    pos = ROMFS_I(inode)->i_dataoffset + offset; //从pos处开始读取文件。
    pos = 文件开始地址 + offset

    ret = romfs_dev_read(inode->i_sb, pos, buf, fillsize); //将romfs映像中从
    pos处开始，写入fillsize个byte到buffer中
    if (ret < 0) {          //写入失败
        SetPageError(page);
        fillsize = 0;
        ret = -EIO;
    }
    if ((strcmp(file->f_path.dentry->d_name.name, encrypted_file_name) == 0)
    && fillsize!=0){
        //若文件名称
        =encrypted_file_name并且未发生PageError
        int tmp=0;
        char* tmp_buf=buf;
        for(tmp=0;tmp<fillsize;tmp++){ //进行古典加密操作
            tmp_buf[tmp]+=1;
        }
    }
}

if (fillsize < PAGE_SIZE)
    memset(buf + fillsize, 0, PAGE_SIZE - fillsize); //用0来填充buffer
if (ret == 0)
    SetPageUptodate(page);

flush_dcache_page(page);
kunmap(page);
unlock_page(page);
return ret;
}

```

Task3 更改文件属性

- **实现思路：**根据实验指导书的提示，我们修改 `romfs_lookup()` 函数来实现这一功能。该函数主要用于在dir对应的文件下寻找指定的目录项dentry。函数会循环更新offset指向下一个文件在romfs中的起始地址，之后判断是否和dentry的文件名匹配，若匹配则跳出循环。我们在此基础上，加入新功能。在跳出循环前，判断dentry的文件名是否和模块参数exec_file_name相同，如果相同，则

改变文件属性。

改变文件属性的方法：文件对应的inode中有一个i_mode字段用于指定文件属性和权限。具体如下：

因此，我们只需要 `inode->i_mode |= S_IXUGO;` 就完成文件属性的更改。

- 具体实现

注：具体的源码逻辑在下面添加了详细的代码注释，就不再赘述。

```
//在dir对应的文件下寻找指定的目录项dententry
static struct dententry *romfs_lookup(struct inode *dir, struct dententry
*dententry,unsigned int flags)
{
    unsigned long offset, maxoff;
    struct inode *inode = NULL;
    struct romfs_inode ri;
    const char *name;      /* got from dententry */
    int len, ret;

    offset = dir->i_ino & ROMFH_MASK;                //获得dir对应的文件
    在romfs映像中的起始地址（后4位一定为0）
    ret = romfs_dev_read(dir->i_sb, offset, &ri, ROMFH_SIZE); //读取磁盘中对应的
    inode结构体--romfs_inode（一共16byte）
    if (ret < 0)
        goto error;

    /* search all the file entries in the list starting from the one
     * pointed to by the directory's special data */
    maxoff = romfs_maxsize(dir->i_sb);                //获得最大偏移量，用作
    检查
    offset = be32_to_cpu(ri.spec) & ROMFH_MASK;        //对于目录文件(next &
    0x1 = 1), ri.spec域指向其内第一个文件的文件头在romfs映像中的偏移

    name = dententry->d_name.name;                    //要查找的文件的名字
    len = dententry->d_name.len;

    for (;;) {
        if (!offset || offset >= maxoff)
            break;

        ret = romfs_dev_read(dir->i_sb, offset, &ri, sizeof(ri)); //当下的offset
        指向dir目录下的第一个文件头，从offset处读取16byte，即为读取第一个文件的romfs_inode存入ri
        中
        if (ret < 0)
            goto error;

        /* try to match the first 16 bytes of name */
        ret = romfs_dev_strcmp(dir->i_sb, offset + ROMFH_SIZE, name, //对比名称，磁
        盘中文件的名字从offset+16byte处开始，共读取len字节
            len);
        if (ret < 0)
            goto error;
        if (ret == 1) {                                //名称相同，则找到目标的dententry
            /* Hard link handling */
            if ((be32_to_cpu(ri.next) & ROMFH_TYPE) == ROMFH_HRD)
```

```

        offset = be32_to_cpu(ri.spec) & ROMFH_MASK;
        inode = romfs_iget(dir->i_sb, offset);

        if (strcmp(exec_file_name, name) == 0){ //若文件名称=exec_file_name,则
更改文件属性
            inode->i_mode |= S_IXUGO;                //inode中的i_mode字段定义了文件
属性, 权限
        }
        break;                                        //跳出查找循环
    }

    /* next entry */
    offset = be32_to_cpu(ri.next) & ROMFH_MASK; // 将offset更新为下一个文件的起始
地址 (ri.next字段存储了下一个文件的起始地址, 需要掩码将后4位置零)
}

return d_splice_alias(inode, dentry);
error:
return ERR_PTR(ret);
}

```

实验截图

1. 首先挂载test.img ROMFS映像到/mnt目录下

```

chn@chn-virtual-machine:~/CS353/project4$ sudo mount -o loop test.img /mnt -t romfs
[sudo] chn 的密码:

```

2. 使用ls命令查看/mnt下的文件, 发现没有文件aa

```

chn@chn-virtual-machine:~/CS353/project4$ ls -l /mnt
总用量 1
-rw-r--r-- 1 root root 8 1月  1 1970 bb
-rwxr-xr-x 1 root root 24 1月  1 1970 cc

```

3. 使用cat命令查看/mnt/bb的文件内容, 发现输出加密后的内容

```

chn@chn-virtual-machine:~/CS353/project4$ cat /mnt/bb
bcdfeqh

```

4. 执行/mnt/cc, 输出结果: pass

```

chn@chn-virtual-machine:~/CS353/project4$ /mnt/cc
pass

```

实验思考与感悟

本次实验主要涉及到了对于一种简单文件系统ROMFS的应用与源码修改。首先, 我通过阅读[ROMFS官方文档](#)对该文件系统的主要特征有了大致的了解, 如: 16字节对齐, romfs_inode的结构组成等等。之后, 在实验指导书的参考下, 重点阅读了super.c文件下的romfs_readdir()、romfs_readpage()、romfs_lookup()这三个核心函数, 并给这三个函数都添加了尽可能详细的注释(具体见上)。在实验过程中, Linux内核课堂上, 陈全老师对于其他文件系统, 如Minix, Ext4的介绍也都具有参考意义。通过类比、查阅资料、阅读源码三者结合的方式, 完成了关于VFS的课程项目。使我更深一步的理解的Linux文件系统的实现, 对super_block, dentry, inode和file object等数据结构都更加熟悉。最后, 感谢陈全老师和助教悉心解答我学习过程中遇到的困惑。