PRÁCTICA FINAL

Creación de un sistema de ficheros propio

Ángel Manuel Guerrero Higueras
Vicente Matellán Olivera

Mayo 2016

1 Objetivo

El kernel de Linux incluye un conjunto de rutinas conocido como *libfs* diseñada para simplificar la tarea de escribir sistemas de ficheros. *libfs* se encarga de las tareas más habituales de un sistema de ficheros permitiendo al desarrollador centrarse en la funcionalidad más específica.

El objetivo de esta práctica es en construir un sistema de ficheros basado en inodos con una funcionalidad muy básica utilizando *libfs*. Para ello, es preciso implementar un nuevo módulo que permita al kernel gestionar sistemas de ficheros de tipo *assoofs*. En el siguiente aparatado se detallan los pasos para crear un nuevo módulo para el kernel.

La práctica consta de una parte básica que supone el 80% de la nota y de una parte opcional que supone el 20%. Para aprobar la práctica es obligatorio tener, al menos, la parte básica implementada.

1.1 Parte básica (80%)

El sistema de ficheros contendrá un dos contadores y una carpeta. Leer uno de estos contadores mostrará su valor actual al usuario, incrementándolo a continuación. Un ejemplo de interacción con el sistema de ficheros podría ser el siguiente:

```
# mount -t assoofs /dev/loop0 /mnt/assoofs
# cat /mnt/assoofs/contador1
0
# cat /mnt/assoofs/contador1
1
# cat /mnt/assoofs/contador1
2
# ...

# cd /mnt/assoofs/carpeta1
# cat /mnt/assoofs/carpeta1/contador2
0
# cat /mnt/assoofs/carpeta1/contador2
```

```
1
# cat /mnt/assoofs/carpeta1/contador2
2
# ...
```

También se podrá asignar un valor a un contador escribiendo sobre el fichero:

```
# echo 666 > /mnt/assoofs/contador1
# cat /mnt/assoofs/contador1
666
#
```

1.2 Parte opcional (20%)

A continuación se detallan algunas mejoras que se pueden añadir al sistema de ficheros *assoofs*.

1.2.1 Creación de nuevos contadores

El módulo debe permitir la creación de nuevos ficheros (contadores) en el sistema de ficheros.

1.2.2 Creación de nuevas carpetas

El módulo debe permitir la creación de nuevas carpetas en el sistema de ficheros.

1.2.3 Escritura del contenido de los ficheros

Las operaciones de escritura deben permitir escribir contenido (texto) en los ficheros.

1.2.4 Lectura del contenido de los ficheros

Las operaciones de lectura deben mostrar el contenido de los ficheros.

2 Creación de un módulo

El siguiente fragmento de código muestra la implementación de un módulo sencillo cargable en el kernel de Linux.

```
#include <linux/module.h>     /* Needed by all modules */
#include <linux/kernel.h>     /* Needed for KERN_INFO */
#include <linux/init.h>       /* Needed for the macros */
#include <linux/pagemap.h>    /* PAGE_CACHE_SIZE */
#include <linux/fs.h>         /* libfs stuff */
#include <asm/atomic.h>       /* atomic_t stuff */
#include <asm/uaccess.h>      /* copy_to_user */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Angel Manuel Guerrero Higueras");

static int __init init_hello(void)
```

```
{
    printk(KERN_INFO "Hello world\n");
    return 0;
}

static void __exit cleanup_hello(void)
{
    printk(KERN_INFO "Goodbye world\n");
}

module_init(init_hello);
module_exit(cleanup_hello);
```

Para compilar el programa anterior se utilizará la herramienta *make*. Para ello, se necesita un fichero de configuración *Makefile* similar al siguiente:

```
obj-m += helloWorldModule.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

La siguiente secuencia de comandos detalla los pasos que hay que seguir para compilar el módulo con la herramienta *make* y, posteriormente, insertarlo (comando *insmod*) y borrarlo (comando *rmmod*) en el kernel. El comando *dmesg* muestra todos los mensajes del kernel:

```
# make
make -C /lib/modules/3.13.0-86-generic/build M=/root modules
make[1]: se ingresa al directorio "/usr/src/linux-headers-3.13.0-86-generic"
  CC [M]  /root/helloWorldModule.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/helloWorldModule.mod.o
  LD [M]  /root/helloWorldModule.ko
make[1]: se sale del directorio "/usr/src/linux-headers-3.13.0-86-generic"
# insmod helloWorldModule.ko
[ 2424.977652] Hello world
# rmmod helloWorldModule
# dmesg
...
[ 2424.977652] Hello world
[ 2488.350933] Goodbye world
```

3 Implementación de un sistema de ficheros

Para implementar un sistema de ficheros es necesario realizar una serie de tareas que se describen a continuación.

3.1 Inicialización y configuración del superbloque

Un modulo que implemente un sistema de ficheros debe registrarlo en la capa VFS en el momento de cargarlo:

```
static int __init assoofs_init(void)
{
    return register_filesystem(&assoofs_type);
}
```

El argumento *assoofs_type* es una estructura que se debe inicializar como sigue:

```
static struct file_system_type assoofs_type = {
    .owner    = THIS_MODULE,
    .name     = "assoofs",
    .mount    = assoofs_get_super,
    .kill_sb = kill_litter_super,
};
```

Es necesario implementar la función *assoofs_get_super* que se ejecuta cuando se registra el sistema de ficheros.

```
static struct dentry *assoofs_get_super(
    struct file_system_type *fst,
    int flags, const char *devname, void *data)
{
    return mount_bdev(fst, flags, devname, data,
        assoofs_fill_super);
}
```

La función anterior llama a su vez a la función *assoofs_fill_super* que se encargará de configurar el sistema de ficheros. Su prototipo es similar al siguiente:

```
static int assoofs_fill_super (struct super_block *sb,
    void *data, int silent);
```

La función *assoofs_fill_super* debe incluir algunas configuraciones básicas:

```
sb->s_blocksize = PAGE_CACHE_SIZE;
sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
sb->s_magic = LFS_MAGIC;
sb->s_op = &assoofs_s_ops;
```

assoofs_s_ops es una estructura que define las operaciones que se puede realizar con el superbloque.

```
static struct super_operations assoofs_s_ops = {
    .statfs         = simple_statfs,
    .drop_inode     = generic_delete_inode,
};
```

3.2 Creación del nodo raíz

Además de crear y configurar el superbloque la función *assoofs_fill_super* debe incluir el código necesario para crear el inodo raíz del sistema de ficheros.

Para crear el inodo para el directorio raíz debemos invocar a la función *assoofs_make_inode*:

```
root = assoofs_make_inode (sb, S_IFDIR | 0755);
if (! root)
    goto out;
```

assoofs_make_inode devuelve un puntero al nuevo inodo que representará el directorio raíz.

También hay que asignar al nuevo inodo dos estructuras con las operaciones que soporta (incluídas en las rutinas de *libfs*):

```
root->i_op = &simple_dir_inode_operations;
root->i_fop = &simple_dir_operations;
```

Más adelante se detalla la implementación de la función *assoofs_make_inode*.

El inodo raíz es un directorio y en estos casos es necesario utilizar una estructura especial (*struct dentry*) que añadiremos a la cache de directorios para que la capa VFS pueda localizarlo:

```
root_dentry = d_make_root(root);
if (! root_dentry)
    goto out_iput;
sb->s_root = root_dentry;
```

3.3 Creación de ficheros

En este punto el superbloque tiene un directorio raíz inicializado. Todas las operaciones sobre directorios serán manejadas por las rutinas de *libfs* y por la capa VFS.

Lo que *libfs* no puede hacer es poner en el directorio raíz los archivos y carpetas que piden en el enunciado. Lo último que debe hacer la función *assoofs_fill_super* es crear en el directorio raíz el fichero *contador1* y la carpeta *carpeta1.* y el fichero *contador2* dentro de esta última.

Para ello lo más adecuado es escribir una nueva función que haga el trabajo y que tenga un prototipo similar al siguiente:

```
static void assoofs_create_files (struct super_block *sb,
    struct dentry *root);
```

Después de crear el inodo raíz, la función *assoofs_fill_super* debe llamar a *assoofs_create_files*, y esta función a su vez, debe crear los archivos *contador1* y *contador2* y la carpeta *carpeta1*:

```
static atomic_t counter;

static void assoofs_create_files (struct super_block *sb,
    struct dentry *root)
{
    /* ... */
    atomic_set(&counter, 0);
    assoofs_create_file(sb, root, "counter", &counter);
    /* ... */
}
```

Para crear los contadores utilizaremos el tipo **atomic_t**.

assoofs_create_file es la función que crea realmente un fichero sobre un directorio.

```
static struct dentry *assoofs_create_file (
    struct super_block *sb, struct dentry *dir,
    const char *name, atomic_t *counter)
{
    struct dentry *dentry;
    struct inode *inode;
    struct qstr qname;
```

Lo primero que se debe hacer es inicializar una entrada de directorio (*struct dentry*) e insertarla en la cache de directorios (función *d_alloc*) de la capa VFS:

```
qname.name = name;
qname.len = strlen (name);
qname.hash = full_name_hash(name, qname.len);
dentry = d_alloc(dir, &qname);
if (! dentry)
    goto out;
```

Después hay que crear un inodo para el nuevo fichero.

```
inode = assoofs_make_inode(sb, S_IFREG | 0644);
if (! inode)
    goto out_dput;
inode->i_fop = &assoofs_file_ops;
inode->i_private = counter;
```

Al nuevo inodo hay que asignarle una estructura con las operaciones que soporta (*assoofs_file_ops*).

Utilizaremos el puntero *i_private* para almacenar el contador *atomic_t* asignado al fichero.

El último paso es añadir la estructura *dentry* a la cache de directorios.

```
d_add(dentry, inode);
return dentry;
```

3.4 Creación de inodos

Utilizaremos la función *assoofs_make_inode* para crear inodos:

```
static struct inode *assoofs_make_inode(struct super_block *sb,
    int mode)
{
    struct inode *ret = new_inode(sb);

    if (ret) {
        ret->i_mode = mode;
        ret->i_uid.val = ret->i_gid.val = 0;
        ret->i_blocks = 0;
        ret->i_atime = ret->i_mtime = ret->i_ctime =
            CURRENT_TIME;
    }
    return ret;
}
```

3.5 Implementación de las operaciones sobre ficheros

Las operaciones que soportan los ficheros se definen utilizando una estructura *file_operations*:

```
static struct file_operations assoofs_file_ops = {
    .open  = assoofs_open,
    .read  = assoofs_read_file,
    .write = assoofs_write_file,
};
```

La operación más sencilla es *open()*. Solamente tenemos que copiar el puntero al contador *atomic_t* en la estructura *file*.

```
static int assoofs_open(struct inode *inode,
    struct file *filp)
{
    filp->private_data = inode->i_private;
    return 0;
}
```

La operación *read()* debe incrementar el contador y devolver su valor al espacio del usuario. El prototipo debe ser similar a este:

```
static ssize_t assoofs_read_file(struct file *filp, char *buf,
    size_t count, loff_t *offset);
```

Lo primero que debe hacer es leer e incrementar el contador:

```
v = atomic_read(counter);
if (*offset > 0)
    v -= 1;  /* the value returned when offset was zero */
else
    atomic_inc(counter);
```

Después hay que devolver el valor al espacio de usuario:

```
len = snprintf(tmp, TMPSIZE, "%d\n", v);
if (*offset > len)
    return 0;
if (count > len - *offset)
    count = len - *offset;

if (copy_to_user(buf, tmp + *offset, count))
        return -EFAULT;
    *offset += count;
    return count;
```

La operación write() debe hacer algo parecido a lo siguiente:

```
static ssize_t assoofs_write_file(
    struct file *filp, const char *buf,
    size_t count, loff_t *offset)
{
    atomic_t *counter = (atomic_t *) filp->private_data;

    /* ... */

    memset(tmp, 0, TMPSIZE);
    if (copy_from_user(tmp, buf, count))
        return -EFAULT;
```

```
    atomic_set(counter, simple_strtol(tmp, NULL, 10));
    return count;
}
```

En este punto solamente queda definir una función parecida a *assoofs_create_file*, que permita crear directorios (p.e. *assoofs_create_dir*).

4 Referencias

- Linux Device Drivers, Third Edition By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman.