PRÁCTICA FINAL

Creación de un sistema de ficheros (ASSOOFS)

Miguel Ángel Conde González
Antonio Gómez García
Ángel Manuel Guerrero Higueras
Juan Delfín Pélaez Álvarez

Mayo 2018

Resumen

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.

Índice

1. Creación de un módulo

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

```
1 #include <linux/module.h>       /* Needed by all modules */
2 #include <linux/kernel.h>       /* Needed for KERN_INFO  */
3 #include <linux/init.h>         /* Needed for the macros */
4 #include <linux/fs.h>           /* libfs stuff           */
5 #include <asm/uaccess.h>        /* copy_to_user          */
6 #include <linux/buffer_head.h>  /* buffer_head           */
```

```c
#include <linux/slab.h>         /* kmem_cache            */

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);
```

Guardaremos la rutina de código anterior en un fichero llamado `helloWorldModule.c`. Para compilar `helloWorldModule.c` utilizaremos la herramienta *make*. Para ello, se necesita un fichero de configuración `Makefile` similar al siguiente:

```
obj-m := helloWorldModule.o

all: ko

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

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

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

```
# ls
helloWorldModule.c  Makefile
# 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
# dmesg
...
[ 2424.977652] Hello world
# rmmod helloWorldModule
# dmesg
...
[ 2424.977652] Hello world
[ 2488.350933] Goodbye world
```

`helloWorldModule.c` y `Makefile` tienen que estar en la misma carpeta desde la cual ejecutemos el comando *make*.

2. Implementación de ASSOOFS

Para implementar el sistema de ficheros ASSOOFS hay que realizar las siguientes de tareas:

1. Definir y declarar las estructuras de datos y constantes necesarias.
2. Implementar un programa que permita formatear dispositivos de bloques como ASSOOFS.
3. Implementar un módulo para que el kernel del SO pueda interactuar con un dispositivo de bloques con formato ASSOOFS.

2.1. Estructuras de datos necesarias

El fichero `assoofs.h`, cuyo contenido muestra el siguiente listado, contiene las estructuras de datos y constantes necesarias:

```c
#define ASSOOFS_MAGIC 0x20170509
#define ASSOOFS_DEFAULT_BLOCK_SIZE 4096
#define ASSOOFS_FILENAME_MAXLEN 255
#define ASSOOFS_START_INO 10
#define ASSOOFS_RESERVED_INODES 3
#define ASSOOFS_LAST_RESERVED_BLOCK ASSOOFS_ROOTDIR_DATABLOCK_NUMBER
#define ASSOOFS_LAST_RESERVED_INODE ASSOOFS_INODESTORE_BLOCK_NUMBER
const int ASSOOFS_SUPERBLOCK_BLOCK_NUMBER = 0;
const int ASSOOFS_INODESTORE_BLOCK_NUMBER = 1;
const int ASSOOFS_ROOTDIR_DATABLOCK_NUMBER = 2;
const int ASSOOFS_ROOTDIR_INODE_NUMBER = 1;
```

```c
const int ASSOOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED = 64;

struct assoofs_super_block_info {
    uint64_t version;
    uint64_t magic;
    uint64_t block_size;
    uint64_t inodes_count;
    uint64_t free_blocks;
    char padding[4056];
};

struct assoofs_dir_record_entry {
    char filename[ASSOOFS_FILENAME_MAXLEN];
    uint64_t inode_no;
};

struct assoofs_inode_info {
    mode_t mode;
    uint64_t inode_no;
    uint64_t data_block_number;
    union {
        uint64_t file_size;
        uint64_t dir_children_count;
    };
};
```

El sistema de ficheros ASSOOFS soporta un máximo de 64 bloques como muestra la figura 1.

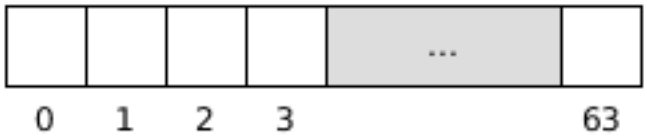

Figura 1: Dispositivo de bloques con formato ASSOOFS.

2.2. Implementación un programa que permita formatear dispositivos de bloques como ASSOOFS

Para formatear dispositivos de bloques como ASSOOFS necesitaremos un programa parecido a `mkassoofs.c`, cuyo código se muestra a continuación:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "assoofs.h"
#define WELCOMEFILE_DATABLOCK_NUMBER (ASSOOFS_LAST_RESERVED_BLOCK + 1)
#define WELCOMEFILE_INODE_NUMBER (ASSOOFS_LAST_RESERVED_INODE + 1)

static int write_superblock(int fd) {
    struct assoofs_super_block_info sb = {
        .version = 1,
        .magic = ASSOOFS_MAGIC,
        .block_size = ASSOOFS_DEFAULT_BLOCK_SIZE,
        .inodes_count = WELCOMEFILE_INODE_NUMBER,
        .free_blocks = (~0) & ~(15),
    };
    ssize_t ret;

    ret = write(fd, &sb, sizeof(sb));
    if (ret != ASSOOFS_DEFAULT_BLOCK_SIZE) {
        printf("Bytes written [%d] are not equal to the default block size.\n", (int)ret);
        return -1;
    }

    printf("Super block written succesfully.\n");
    return 0;
}

static int write_root_inode(int fd) {
    ssize_t ret;
    struct assoofs_inode_info root_inode;

    root_inode.mode = S_IFDIR;
    root_inode.inode_no = ASSOOFS_ROOTDIR_INODE_NUMBER;
    root_inode.data_block_number = ASSOOFS_ROOTDIR_DATABLOCK_NUMBER;
    root_inode.dir_children_count = 1;
```

```
41
42      ret = write(fd, &root_inode, sizeof(root_inode));
43
44      if (ret != sizeof(root_inode)) {
45          printf("The inode store was not written properly.\n");
46          return -1;
47      }
48
49      printf("root directory inode written succesfully.\n");
50      return 0;
51  }
52
53  static int write_welcome_inode(int fd, const struct assoofs_inode_info *i) {
54      off_t nbytes;
55      ssize_t ret;
56
57      ret = write(fd, i, sizeof(*i));
58      if (ret != sizeof(*i)) {
59          printf("The welcomefile inode was not written properly.\n");
60          return -1;
61      }
62      printf("welcomefile inode written succesfully.\n");
63
64      nbytes = ASSOOFS_DEFAULT_BLOCK_SIZE - (sizeof(*i) * 2);
65      ret = lseek(fd, nbytes, SEEK_CUR);
66      if (ret == (off_t)-1) {
67          printf("The padding bytes are not written properly.\n");
68          return -1;
69      }
70
71      printf("inode store padding bytes (after two inodes) written sucessfully.\n");
72      return 0;
73  }
74
75  int write_dirent(int fd, const struct assoofs_dir_record_entry *record) {
76      ssize_t nbytes = sizeof(*record), ret;
77
78      ret = write(fd, record, nbytes);
79      if (ret != nbytes) {
80          printf("Writing the rootdirectory datablock (name+inode_no pair for welcomefile) has failed.\n");
81          return -1;
82      }
83      printf("root directory datablocks (name+inode_no pair for welcomefile) written succesfully.\n");
84
85      nbytes = ASSOOFS_DEFAULT_BLOCK_SIZE - sizeof(*record);
86      ret = lseek(fd, nbytes, SEEK_CUR);
87      if (ret == (off_t)-1) {
88          printf("Writing the padding for rootdirectory children datablock has failed.\n");
89          return -1;
90      }
91      printf("Padding after the rootdirectory children written succesfully.\n");
92      return 0;
93  }
94
95  int write_block(int fd, char *block, size_t len) {
96      ssize_t ret;
97
98      ret = write(fd, block, len);
99      if (ret != len) {
100         printf("Writing file body has failed.\n");
101         return -1;
102     }
103     printf("block has been written succesfully.\n");
104     return 0;
105 }
106
107 int main(int argc, char *argv[]) {
108     int fd;
109     ssize_t ret;
110     char welcomefile_body[] = "Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.\n";
111
112     struct assoofs_inode_info welcome = {
113         .mode = S_IFREG,
114         .inode_no = WELCOMEFILE_INODE_NUMBER,
115         .data_block_number = WELCOMEFILE_DATABLOCK_NUMBER,
116         .file_size = sizeof(welcomefile_body),
117     };
118
119     struct assoofs_dir_record_entry record = {
120         .filename = "README.txt",
121         .inode_no = WELCOMEFILE_INODE_NUMBER,
122     };
123
124     if (argc != 2) {
125         printf("Usage: mkassoofs <device>\n");
126         return -1;
127     }
128
129     fd = open(argv[1], O_RDWR);
130     if (fd == -1) {
131         perror("Error opening the device");
132         return -1;
133     }
134
```

```
    ret = 1;
    do {
        if (write_superblock(fd))
            break;

        if (write_root_inode(fd))
            break;

        if (write_welcome_inode(fd, &welcome))
            break;

        if (write_dirent(fd, &record))
            break;

        if (write_block(fd, welcomefile_body, welcome.file_size))
            break;

        ret = 0;
    } while (0);

    close(fd);
    return ret;
}
```

La figura 2 muestra el contenido de un dispositivo de bloques con formato ASSOOFS después de ejecutar `mkassoofs`.

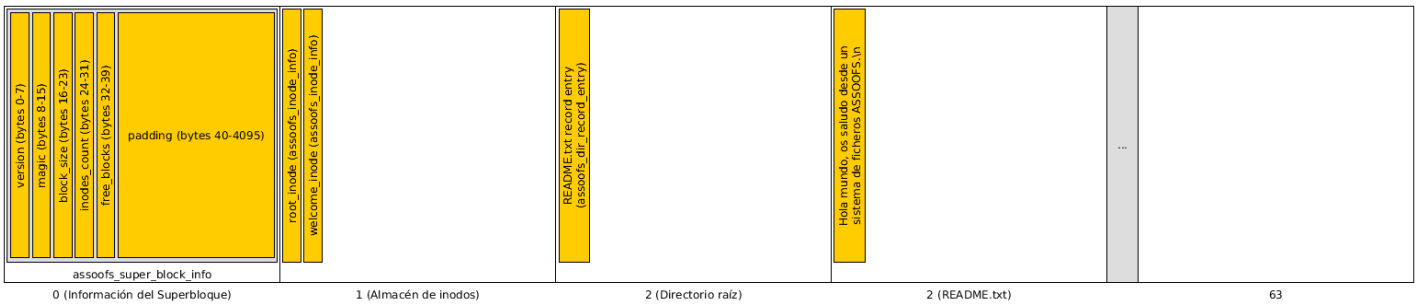

Figura 2: Contenido de un dispositivo de bloques con formato ASSOOFS después de ejecutar `mkassoofs`.

2.3. Implementación de un módulo para ASSOOFS

Para implementar un sistema de ficheros es necesario seguir los pasos que se enumeran a continuación. Se recomienda empezar a partir de un módulo básico como el que se muestra en el apartado 1.

1. Inicializar y registrar el nuevo sistema de ficheros en el kernel.
2. Implementar una función que permita montar dispositivos con el nuevo sistema de ficheros.
3. Implementar una función para inicializar el superbloque.
4. Declarar una estructura e implementar funciones para manejar inodos.
5. Declarar una estructura e implementar funciones para manejar archivos y directorios.

Algunas recomendaciones:

- Mantener una cache de inodos.
- Utilizar semáforos para acceder a las estructuras principales.

El detalle de cada paso se describe en los siguientes sub-apartados.

2.3.1. Inicializar y registrar el nuevo sistema de ficheros en el kernel

Lo primero es definir dos funciones, `assoofs_init` y `assoofs_exit`, que se ejecutaran cuando se cargue y se borre respectivamente el módulo en el kernel. `assoofs_init` tiene que registrar el nuevo sistema de ficheros en el kernel. `assoofs_exit` tiene que eliminar la información del nuevo sistema de ficheros del kernel. Para ello, tendrán que hacer uso de las funciones `register_filesystem` y `unregister_filesystem` respectivamente. Los prototipos de ambas funciones son los siguientes:

```
extern int register_filesystem(struct file_system_type *);
extern int unregister_filesystem(struct file_system_type *);
```

Ambas funciones requieren un argumento de tipo `struct file_system_type`, que se define como sigue:

```
struct file_system_type {
  const char *name;
  int fs_flags;
#define FS_REQUIRES_DEV   1
#define FS_BINARY_MOUNTDATA 2
#define FS_HAS_SUBTYPE    4
#define FS_USERNS_MOUNT   8 /* Can be mounted by userns root */
#define FS_RENAME_DOES_D_MOVE 32768 /* FS will handle d_move() during rename() internally. */
  struct dentry *(*mount) (struct file_system_type *, int,
          const char *, void *);
  void (*kill_sb) (struct super_block *);
  struct module *owner;
  struct file_system_type * next;
  struct hlist_head fs_supers;

  struct lock_class_key s_lock_key;
  struct lock_class_key s_umount_key;
  struct lock_class_key s_vfs_rename_key;
  struct lock_class_key s_writers_key[SB_FREEZE_LEVELS];

  struct lock_class_key i_lock_key;
  struct lock_class_key i_mutex_key;
  struct lock_class_key i_mutex_dir_key;
};
```

Nosotros tenemos que definir nuestra propia variable de tipo `struct file_system_type`. cuya dirección pasaremos a `register_filesystem` y `unregister_filesystem`. Lo haremos como sigue:

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

2.3.2. Implementar una función que permita montar dispositivos con el nuevo sistema de ficheros: `assoofs_mount`

La función `assoofs_mount` permitirá montar un dispositivo de bloques con formato ASSOOFS. Se invocará cuando una vez registrado el nuevo sistema de ficheros un usuario utilice el comando mount con los argumentos `-t assoofs` entre otros. Su prototipo es el siguiente:

```
static struct dentry *assoofs_mount(struct file_system_type *fs_type,
    int flags, const char *dev_name, void *data)
```

Para montar el dispositivo se utilizará la función `mount_bdev`, cuyo protopipo es el siguiente:

```
extern struct dentry *mount_bdev(struct file_system_type *fs_type,
  int flags, const char *dev_name, void *data,
  int (*fill_super)(struct super_block *, void *, int));
```

Sus argumentos son los mismos que `assoofs_mount`, con la excepción del último, que es un puntero a la función que queremos ejecutar para llenar nuestro superbloque. Nosotros llamaremos a esta función `assoofs_fill_super`.

2.3.3. Implementar una función para inicializar el superbloque: `assoofs_fill_super`

El prototipo de `assoofs_fill_super` es el siguiente:

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

`assoofs_fill_super` tiene que realizar las siguientes tareas y **devolver 0** si todo va bien:

1. Leer la información del superbloque del dispositivo de bloques (ver anexo A). En nuestro caso la información del superbloque está en el bloque 0.
2. Comprobar los parámetros del superbloque, al menos: número mágico y tamaño de bloque.
3. Escribir la información leída del dispositivo de bloques en el superbloque, representado por el parámetro `sb` de `assoofs_fill_super`, que no es otra cosa más que un puntero a una variable de tipo `struct super_block`:
   - Asignaremos el número mágico `ASSOOFS_MAGIC` definido en `assoofs.h` al campo `s_magic` del superbloque `sb`.
   - Asignaremos el tamaño de bloque `ASSOOFS_DEFAULT_BLOCK_SIZE` definido en `assoofs.h` al campo `s_maxbytes` del superbloque `sb`.
   - Asignaremos operaciones (campo `s_op` al superbloque `sb`. Las operaciones del superbloque se definen como una variable de tipo `struct super_operations` como sigue:

     ```
     static const struct super_operations assoofs_sops = {
         .drop_inode    = generic_delete_inode,
     };
     ```

   - Para no tener que acceder al bloque 0 del disco constantemente guardaremos la información leída del bloque 0 del disco (en una variable de tipo `struct assoofs_super_block_info`, ver anexo A) en el campo `s_fs_info` del superbloque `sb`.

4. Crear el inodo raíz (ver anexo B).
    - Para crear el inodo sigue los pasos del anexo B.
    - Para las operaciones sobre inodos utilizar la estructura definida en el apartado 2.3.4. Para las operaciones sobre archivos y directorios utilizar las estructuras definidas en el apartado 2.3.5.
    - Por último, marcaremos el nuevo inodo como raíz y lo guardaremos en el superbloque. Para ello, asignaremos el resultado de la función `d_make_root` al campo `s_root` del superbloque (ver anexo B).

2.3.4. Declarar una estructura e implementar funciones para manejar inodos

Para manejar inodos tenemos que declarar una estructura de tipo `struct inode_operations` como sigue:

```
static struct inode_operations assoofs_inode_ops = {
    .create = assoofs_create,
    .lookup = assoofs_lookup,
    .mkdir = assoofs_mkdir,
};
```

Después hay que implementar las funciones para cada operación:

```
struct dentry *assoofs_lookup(struct inode *parent_inode, struct dentry *child_dentry, unsigned int flags);
static int assoofs_create(struct inode *dir, struct dentry *dentry, umode_t mode, bool excl);
static int assoofs_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode);
```

2.3.5. Declarar una estructura e implementar funciones para manejar archivos y directorios

Para manejar ficheros tenemos que declarar una estructura de tipo `struct file_operations` como sigue:

```
const struct file_operations assoofs_file_operations = {
    .read = assoofs_read,
    .write = assoofs_write,
};
```

Para manejar directorios tenemos que declarar una estructura de tipo `struct file_operations` como sigue:

```
const struct file_operations assoofs_dir_operations = {
    .owner = THIS_MODULE,
    .iterate = assoofs_iterate,
};
```

Después hay que implementar las funciones para cada operación:

```
ssize_t assoofs_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos);
ssize_t assoofs_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos);
static int assoofs_iterate(struct file *filp, struct dir_context *ctx);
```

Funciones auxiliares Además de las operaciones sobre inodos y archivos/carpetas definidas anteriormente será necesario contar con algunas funciones auxiliares. Para la gestión de inodos, necesitaremos, por lo menos, las siguientes:

- `assoofs_get_inode`: nos permitirá obtener un puntero al inodo número `ino` del superbloque `sb`:

```
static struct inode *assoofs_get_inode(struct super_block *sb, int ino);
```

- `assoofs_get_inode_info`: nos permitirá obtener la información persistente del inodo número `inode_no` del superbloque `sb`:

```
struct assoofs_inode_info *assoofs_get_inode_info(struct super_block *sb, uint64_t inode_no);
```

- `assoofs_save_inode_info`: nos permitirá actualizar en disco la información persistente de un inodo:

```
int assoofs_save_inode_info(struct super_block *sb, struct assoofs_inode_info *inode_info);
```

- `assoofs_search_inode_info`: nos permitirá obtener un puntero a la información persistente de un inodo concreto:

```
struct assoofs_inode_info *assoofs_search_inode_info(struct super_block *sb, struct assoofs_inode_info *start,
    struct assoofs_inode_info *search);
```

- `assoofs_add_inode_info`: nos permitirá guardar en disco la información persistente de un inodo nuevo:

```
void assoofs_add_inode_info(struct super_block *sb, struct assoofs_inode_info *inode);
```

Para gestionar el superbloque necesitaremos, al menos, las siguientes funciones auxiliares:

- `assoofs_save_sb_info`: nos permitirá atualizar la información persistente del superbloque cuando hay un cambio:

```
void assoofs_save_sb_info(struct super_block *vsb);
```

- `assoofs_sb_get_a_freeblock`: nos permitirá obtener un bloque libre:

```
int assoofs_sb_get_a_freeblock(struct super_block *sb, uint64_t *block);
```

  El bloque libre se devolverá en el cotenido de la variable apuntada por el segundo parámetro `block`.

3. Compilar la solución completa

El siguiente listado muestra el contenido del fichero Makefile para compilar la solución completa. Para que funcione debe cumplirse lo siguiente:

- La implementación del módulo está un fichero llamado `assoofs.c`.
- El fichero `assoofs.h` contiene estructuras y constantes necesarias para compilar la solución.
- El fichero `mkassoofs.c` contiene un programa para formatear dispositivos de bloques como ASSOOFS.

```
obj-m := assoofs.o

all: ko mkassoofs

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

mkassoofs_SOURCES:
  mkassoofs.c assoofs.h

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

4. Formatear, montar y probar un dispositivo ASSOOFS

Para probar nuestro sistema de ficheros tenemos que seguir los siguientes pasos:

1. Compilar:

```
$ make
make -C /lib/modules/3.19.0-15-generic/build M=/home/ubuntu modules
make[1]: Entering directory '/usr/src/linux-headers-3.19.0-15-generic'
  CC [M]  /home/ubuntu/assoofs.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/ubuntu/assoofs.mod.o
  LD [M]  /home/ubuntu/assoofs.ko
make[1]: Leaving directory '/usr/src/linux-headers-3.19.0-15-generic'
cc     mkassoofs.c   -o mkassoofs
```

2. Crear una imagen para contener el sistema de ficheros:

```
$ dd bs=4096 count=100 if=/dev/zero of=image
100+0 records in
100+0 records out
409600 bytes (410 kB) copied, 0.000294943 s, 1.4 GB/s
```

3. Crear el sistema de ficheros:

```
$ ./mkassoofs image
Super block written succesfully
root directory inode written succesfully
welcomefile inode written succesfully
inode store padding bytes (after the two inodes) written sucessfully
root directory datablocks (name+inode_no pair for welcomefile) written succesfully
padding after the rootdirectory children written succesfully
block has been written succesfully
```

Una vez realizado lo anterior, los siguientes pasos hay que ejecutarlos con el usuario root (`sudo su`). **Ojo a las rutas**, no tienen porque ser iguales a las del ejemplo:

4. Insertar el módulo en el kernel:

```
# insmod assoofs.ko
```

5. Crear un punto de montaje:

```
# mkdir mnt
```

6. Montamos la imagen creada en el punto de montaje:

```
# mount -o loop -t assoofs image mnt
```

7. Comprobamos los mensajes del kernel:

```
# dmesg
...
[20999.690170] Sucessfully registered assoofs
[21131.422986] The magic number obtained in disk is: [268640275]
[21131.422988] assoofs filesystem of version [1] formatted with a block size of [4096] detected in the device.
[21131.423014] assoofs is succesfully mounted on [/dev/loop2]
```

8. Comprobamos que el sistema de ficheros se comporta como esperamos:

```
# cd mnt/
# ls
README.txt
# cat README.txt
Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.
# cp README.txt README.txt.bak
# ls
README.txt  README.txt.bak
# cat README.txt.bak
Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.
# mkdir tmp
# ls
README.txt  README.txt.bak  tmp
# cp README.txt tmp/HOLA
# cat tmp/HOLA
Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.
# cd ..
# umount mnt/
# rmmod assoofs
# insmod assoofs.ko; mount -o loop -t assoofs image ~/mnt
# ls -l mnt/
total 0
---------- 1 root root 0 May  8 13:14 README.txt
---------- 1 root root 0 May  8 13:14 README.txt.bak
drwxr-xr-x 1 root root 0 May  8 13:14 tmp
```

Referencias

- Linux Device Drivers, Third Edition By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman.
- The Linux Kernel Module Programming Guide. `http://www.tldp.org/LDP/lkmpg/2.6/html/index.html`
- SIMPLEFS: A simple, kernel-space, on-disk filesystem from the scratch. `https://github.com/psankar/simplefs`
- Creating Linux virtual filesystems. `https://lwn.net/Articles/57369/`

A. Leer bloques de disco

Para manejar bloques utilizaremos variables de tipo `struct buffer_head`. Para leer bloques de disco se utiliza la función `sb_bread` que devuelve un `struct buffer_head`:

```
static inline struct buffer_head *
sb_bread(struct super_block *sb, sector_t block)
{
  return __bread(sb->s_bdev, block, sb->s_blocksize);
}
```

El primer argumento es un puntero al superbloque de nuestro sistema de ficheros, el segundo es el identificador de bloque (0, 1, ..., 63). Devuelve una variable de tipo `struct buffer_head`.

El contenido del bloque se almacena en el campo `b_data` del `struct buffer_head` devuelto por la función. Para acceder al contenido es preciso hacer un cast al tipo de datos que corresponda. Por ejemplo, para leer la información del superbloque en un sistema de ficheros ASSOOFS haremos lo siguiente:

```
struct buffer_head *bh;
struct assoofs_super_block_info *assoofs_sb;
bh = sb_bread(sb, ASSOOFS_SUPERBLOCK_BLOCK_NUMBER); // sb lo recibe assoofs_fill_super como argumento
assoofs_sb = (struct assoofs_super_block_info *)bh->b_data;
```

Después de utilizar el bloque podemos liberar la memoria asignada con la función *brelse*:

```
brelse(bh);
```

B. Crear inodos

Para crear inodos utilizaremos la función `new_inode`. Devuelve un puntero a una variable de tipo `struct inode` y recibe como argumento el superbloque del sistema de ficheros donde queremos crear el nuevo inodo.

```
extern struct inode *new_inode(struct super_block *sb);
```

`new_inode` permite inicializar una variable de tipo `struct inode`:

```
struct inode *root_inode;
root_inode = new_inode(sb);
```

Después de inicializar el inodo, asignaremos propietario y permisos con la función `inode_init_owner`, cuyo prototipo se muestra a continuación:

```
extern void inode_init_owner(struct inode *inode, const struct inode *dir, mode_t mode);
```

Y que se invoca como sigue:

```
inode_init_owner(root_inode, NULL, S_IFDIR); // S_IFDIR para directorios, S_IFREG para ficheros.
```

El segundo argumento se corresponde con el inodo del directorio que contiene el fichero o el directorio, que se corresponderá con el inodo padre del nuevo inodo. Indicando `NULL` en este argumento, estamos diciendo que el nuevo inodo no tiene padre, lo que sólo ocurre con el directorio raíz. En otro caso tendremos que indicar un inodo padre.

Después, asignaremos información al inodo. En concreto: el número de inodo; el superbloque del sistema de ficheros al que pertenece; fechas de creación, modificación y acceso; y operaciones que soporta el inodo.

```
root_inode->i_ino = ASSOOFS_ROOTDIR_INODE_NUMBER; // número de inodo
root_inode->i_sb = sb; // puntero al superbloque
root_inode->i_op = &assoofs_inode_ops; // dirección de una variable de tipo struct inode_operations previamente declarada
root_inode->i_fop = &assoofs_file_operations; // dirección de una variable de tipo struct file_operations previamente
    declarada
root_inode->i_atime = root_inode->i_mtime = root_inode->i_ctime = CURRENT_TIME; // fechas.
root_inode->i_private = assoofs_get_inode_info(sb, ASSOOFS_ROOTDIR_INODE_NUMBER); // Información persistente del inodo
```

Por último tenemos que introducir el nuevo inodo en el árbol de inodos. Hay dos formas de hacer esto:

1. Cuando el nuevo inido se trate del inodo raíz lo marcaremos como tal y lo guardaremos en el superbloque. Para ello, asignaremos el resultado de la función `d_make_root` al campo `s_root` del superbloque `sb`. El prototipo de `d_make_root` es el siguiente:

   ```
   extern struct dentry * d_make_root(struct inode *);
   ```

   Y se invoca como sigue:

   ```
   sb->s_root = d_make_root(root_inode);
   ```

2. Cuando se trate de un inodo normal (no raíz). Utilizaremos la función `d_add` para introducir el nuevo inodo en el árbol de inodos. Su prototipo es el siguiente:

   ```
   static inline void d_add(struct dentry *entry, struct inode *inode);
   ```

   Y se invoca como sigue:

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

   El primer argumento es un puntero a una variable de tipo `struct dentry` que representa al directorio padre. Su valor nos vendrá dado como argumento en la función desde la que queramos crear un nuevo inodo. En nuestro caso: `assoofs_lookup`, `assoofs_create` y `assoofs_mkdir` (ver apartado 2.3.4).

   El segundo argumento, es el `struct inode` que representa al nuevo nodo.

C. Guardar bloques en disco

El siguiente ejemplo permite actualizar el bloque 0 de un sistema ASSOOFS.

```
struct buffer_head *bh;
struct assoofs_super_block *sb = vsb->s_fs_info;
bh = sb_bread(vsb, ASSOOFS_SUPERBLOCK_BLOCK_NUMBER);
bh->b_data = (char *)sb;
mark_buffer_dirty(bh);
sync_dirty_buffer(bh);
brelse(bh);
```

D. Caché de inodos

Mantener una caché con la información persistente de nuestros inodos mejorará el rendimiento de ASSOOFS. Para hacerlo lo primero que tenemos que hacer es declarar una variable global en nuestro módulo de tipo `kmem_cache`.

```
static struct kmem_cache *assoofs_inode_cache;
```

Para inicializar la caché de inodos podemos utilizar la función `kmem_cache_create` como sigue:

```
assoofs_inode_cache = kmem_cache_create("assoofs_inode_cache", sizeof(struct assoofs_inode_info), 0, (
    SLAB_RECLAIM_ACCOUNT| SLAB_MEM_SPREAD), NULL);
```

Esto lo haremos en la función `assoofs_init`. También tenemos que liberar la caché cuando descarguemos el móvulo del kernel. Para ello invocaremos a `kmem_cache_destroy` en `assoofs_exit`:

```
kmem_cache_destroy(assoofs_inode_cache);
```

Cuando queramos reservar memoria para la información persistente de un inodo lo haremos como sigue:

```
struct assoofs_inode_info *inode_info;
inode_info = kmem_cache_alloc(assoofs_inode_cache, GFP_KERNEL);
```

Las operaciones del superbloque del apartado 2.3.3 se definen como sigue:

```
static const struct super_operations assoofs_sops = {
    .drop_inode    = generic_delete_inode,
};
```

Si usamos una caché de inodos, tendremos que crear nuestra propia función para eliminar inodos en lugar de utilizar `generic_delete_inode`. La función para borrar inodos tiene que parecerse a la siguiente:

```
void assoofs_destroy_inode(struct inode *inode) {
    struct assoofs_inode *inode_info = inode->i_private;
    printk(KERN_INFO "Freeing private data of inode %p (%lu)\n", inode_info, inode->i_ino);
    kmem_cache_free(assoofs_inode_cache, inode_info);
}
```

E. Operaciones binarias sobre `free_blocks`

El siguiente programa ilustra las operaciones binarias necesarias sobre `free_blocks`:

```
#include <stdio.h>
#include <math.h>
#include <limits.h>
#include <stdint.h>
#define pbit(v, ds)  !!((v) & 1 << (ds))

void binary(int v) {
    int i = 32;

    while(i--) putchar(pbit(v, i) + '0');
}

int main() {
    int i;

    uint64_t value0 = (~0); // Complemento a 1 del valor 0
    printf("Value-0 = Complemento a 1 del valor 0: ~0 --->\n\tBinario = ");
    binary(value0);
    printf(", decimal = %lu\n", value0);

    uint64_t value1 = ~(15); // Complemento a 1 del valor 15 (1111)
    printf("Value-1 = Complemento a 1 del valor 15 (1111): ~15 --->\n\tBinario = ");
    binary(value1);
    printf(", decimal = %lu\n", value1);

    uint64_t value2 = (~0 & ~15); // Complemento a 1 del valor 0 AND complemento a 1 del valor 15 (1111)
    printf("Value-2 = Value-0 AND Value-1: ~0 & ~15 --->\n\tBinario = ");
    binary(value2);
    printf(", decimal = %lu\n", value2);
    printf("\n");

    for (i=2; i<32; i++) {
        printf("i = %d, Value-2 &= ~(1 << i) ---> Binario = ", i);
        value2 &= ~(1 << i);
        binary(value2);
        printf(", decimal = %lu\n", value2);
    }

    return 0;
}
```