PRÁCTICA FINAL

Creación del sistema de ficheros ASSOOFS

Ampliación de Sistemas Operativos (ASSOO)
Grado en Ingeniería Informática

Universidad de León

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 las rutinas de *libfs*. Para hacerlo, es necesario implementar un nuevo módulo que permita al kernel gestionar sistemas de ficheros de tipo *assoofs*.

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/`

Índice

1. Creación de un módulo básico

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/fs.h>           /* libfs stuff           */

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
```

La siguiente secuencia de comandos detalla los pasos que hay que seguir para primero compilar el módulo con la herramienta *make*. Después insertarlo en el kernel con el comando *insmod*. A continuación probar que funciona con el comando *dmesg* que mostrará todos los mensajes del kernel. Por último, el comando *rmmod* permite borrar el módulo 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. En cada una de estas tareas crearemos un fichero de código fuente diferente.

1. Definir y declarar las estructuras de datos y constantes necesarias. Las estructuras y constantes las definiremos en el fichero `assoofs.h`.
2. Implementar un programa que permita formatear dispositivos de bloques como ASSOOFS. El programa lo implementaremos en el fichero `mkassoofs.c`.
3. Implementar un módulo para que el kernel del SO pueda interactuar con un dispositivo de bloques con formato ASSOOFS. El módulo lo implementaremos en el fichero `assoofs.c`.

En los apartados 2.1, 2.2 y 2.3 se detalla cada una de estas tareas.

2.1. Estructuras de datos necesarias

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

```
#define ASSOOFS_MAGIC 0x20200406
#define ASSOOFS_DEFAULT_BLOCK_SIZE 4096
#define ASSOOFS_FILENAME_MAXLEN 255
#define ASSOOFS_LAST_RESERVED_BLOCK ASSOOFS_ROOTDIR_BLOCK_NUMBER
#define ASSOOFS_LAST_RESERVED_INODE ASSOOFS_ROOTDIR_INODE_NUMBER
const int ASSOOFS_SUPERBLOCK_BLOCK_NUMBER = 0;
const int ASSOOFS_INODESTORE_BLOCK_NUMBER = 1;
const int ASSOOFS_ROOTDIR_BLOCK_NUMBER = 2;
const int ASSOOFS_ROOTDIR_INODE_NUMBER = 1;
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_BLOCK_NUMBER;
    root_inode.dir_children_count = 1;

    ret = write(fd, &root_inode, sizeof(root_inode));

    if (ret != sizeof(root_inode)) {
        printf("The inode store was not written properly.\n");
        return -1;
    }

    printf("root directory inode written succesfully.\n");
    return 0;
}

static int write_welcome_inode(int fd, const struct assoofs_inode_info *i) {
    off_t nbytes;
    ssize_t ret;

    ret = write(fd, i, sizeof(*i));
    if (ret != sizeof(*i)) {
        printf("The welcomefile inode was not written properly.\n");
        return -1;
    }
    printf("welcomefile inode written succesfully.\n");

    nbytes = ASSOOFS_DEFAULT_BLOCK_SIZE - (sizeof(*i) * 2);
    ret = lseek(fd, nbytes, SEEK_CUR);
    if (ret == (off_t)-1) {
        printf("The padding bytes are not written properly.\n");
        return -1;
    }

    printf("inode store padding bytes (after two inodes) written sucessfully.\n");
    return 0;
}

int write_dirent(int fd, const struct assoofs_dir_record_entry *record) {
    ssize_t nbytes = sizeof(*record), ret;

    ret = write(fd, record, nbytes);
```

```
    if (ret != nbytes) {
        printf("Writing the rootdirectory datablock (name+inode_no pair for welcomefile) has failed.\n");
        return -1;
    }
    printf("root directory datablocks (name+inode_no pair for welcomefile) written succesfully.\n");

    nbytes = ASSOOFS_DEFAULT_BLOCK_SIZE - sizeof(*record);
    ret = lseek(fd, nbytes, SEEK_CUR);
    if (ret == (off_t)-1) {
        printf("Writing the padding for rootdirectory children datablock has failed.\n");
        return -1;
    }
    printf("Padding after the rootdirectory children written succesfully.\n");
    return 0;
}

int write_block(int fd, char *block, size_t len) {
    ssize_t ret;

    ret = write(fd, block, len);
    if (ret != len) {
        printf("Writing file body has failed.\n");
        return -1;
    }
    printf("block has been written succesfully.\n");
    return 0;
}

int main(int argc, char *argv[])
{
    int fd;
    ssize_t ret;
    char welcomefile_body[] = "Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.\n";

    struct assoofs_inode_info welcome = {
        .mode = S_IFREG,
        .inode_no = WELCOMEFILE_INODE_NUMBER,
        .data_block_number = WELCOMEFILE_DATABLOCK_NUMBER,
        .file_size = sizeof(welcomefile_body),
    };

    struct assoofs_dir_record_entry record = {
        .filename = "README.txt",
        .inode_no = WELCOMEFILE_INODE_NUMBER,
    };

    if (argc != 2) {
        printf("Usage: mkassoofs <device>\n");
        return -1;
    }

    fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("Error opening the device");
        return -1;
    }

    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`.

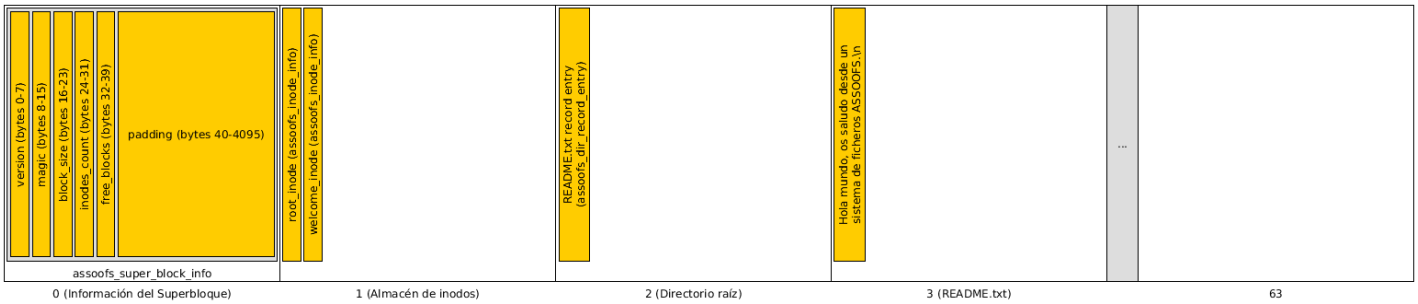

Figura 2: Contenido de un dispositivo de bloques con formato ASSOOFS después de ejecutar `mkassoofs`.

2.3. Implementación de un módulo para ASSOOFS

Para implementar un sistema de ficheros básico 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. Estas funcionalidades suponen los **requisitos básicos** que debe cumplir un sistema de ficheros ASSOOFS.

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. En concreto necesitaremos poder crear nuevos inodos y acceder a la información de los ya existentes.
5. Declarar una estructura e implementar funciones mínimas para manejar archivos y directorios. En concreto se pide poder leer y escribir archivos existentes y crear archivos y carpetas nuevos.

Además de las funcionalidades básicas enumeradas, es recomendable la implementación de las siguientes funcionalidades **opcionales**:

- Mantener una cache de inodos (fácil).
- Utilizar semáforos para acceder a las estructuras principales (fácil).
- Borrado de ficheros con el comando `rm` (difícil).
- Mover ficheros con el comando `mv` (difícil).

El detalle de cada paso se describe en los siguientes sub-apartados. El apéndice A muestra la estructura básica del módulo con las funciones principales.

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`. Tenemos que declarar 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 persistente del superbloque del dispositivo de bloques (ver anexo C). 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 persistente 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 C) en el campo `s_fs_info` del superbloque `sb`.
4. Crear el inodo raíz (ver anexo E).
   - Para crear el inodo sigue los pasos del anexo E.
   - Guardaremos la información persistente del inodo raíz en el campo `i_private`. Esta información está guardada en el disco, es el primer registro del almacén de inodos del bloque 1. Esta operación se realiza en más situaciones, por tanto, es interesante definir una función auxiliar para ello: `assoofs_get_inode_info`.
   - 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 E).

`assoofs_get_inode_info` Esta función auxiliar nos permitirá obtener la información persistente del inodo número `inode_no` del superbloque `sb`. Su prototipo es el siguiente:

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

Esta función realiza las siguientes tareas:

1. Acceder a disco para leer el bloque que contiene el almacén de inodos:

   ```
   struct assoofs_inode_info *inode_info = NULL;
   struct buffer_head *bh;

   bh = sb_bread(sb, ASSOOFS_INODESTORE_BLOCK_NUMBER);
   inode_info = (struct assoofs_inode_info *)bh->b_data;
   ```

2. Recorrer el almacén de inodos en busca del inodo `inode_no`:

```
struct assoofs_super_block_info *afs_sb = sb->s_fs_info;
struct assoofs_inode_info *buffer = NULL;
int i;
for (i = 0; i < afs_sb->inodes_count; i++) {
    if (inode_info->inode_no == inode_no) {
        buffer = kmalloc(sizeof(struct assoofs_inode_info), GFP_KERNEL);
        memcpy(buffer, inode_info, sizeof(*buffer));
        break;
    }
    inode_info++;
}
```

3. Liberal recursos y devolver a información del inodo `inode_no` si estaba en el almacén:

```
brelse(bh);
return buffer;
```

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 = {
    .lookup = assoofs_lookup,
    .create = assoofs_create,
    .mkdir = assoofs_mkdir,
};
```

Es necesario implementar las funciones para cada operación. Los siguientes sub-apartados explican los pasos a seguir con cada una. Cuando alguna operación se repite en otras funciones se recomienda el uso de funciones auxiliares.

assoofs_lookup Esta función busca la entrada (`struct dentry`) con el nombre correcto (`child_dentry->d_name.name`) en el directorio padre (`parent_inode`). Se utiliza para recorrer y mantener el árbol de inodos. Su prototipo es el siguiente:

```
struct dentry *assoofs_lookup(struct inode *parent_inode, struct dentry *child_dentry, unsigned int flags);
```

El primer parámetro es el inodo del directorio padre. El segundo es la entrada que se busca en el directorio padre. El último parámetro no lo utilizaremos.

`assoofs_lookup` tiene que realizar las siguientes tareas:

1. Acceder al bloque de disco con el contenido del directorio apuntado por `parent_inode`.

```
struct assoofs_inode_info *parent_info = parent_inode->i_private;
struct super_block *sb = parent_inode->i_sb;
struct buffer_head *bh;
bh = sb_bread(sb, parent_info->data_block_number);
```

2. Recorrer el contenido del directorio buscando la entrada cuyo nombre se corresponda con el que buscamos. Si se localiza la entrada, entonces tenemos construir el inodo correspondiente.

```
struct assoofs_dir_record_entry *record;
record = (struct assoofs_dir_record_entry *)bh->b_data;
for (i=0; i < parent_info->dir_children_count; i++) {
    if (!strcmp(record->filename, child_dentry->d_name.name)) {
        struct inode *inode = assoofs_get_inode(sb, record->inode_no); // Función auxiliar que obtine la información de
     un inodo a partir de su número de inodo.
        inode_init_owner(inode, parent_inode, ((struct assoofs_inode_info *)inode->i_private)->mode);
        d_add(child_dentry, inode);
        return NULL;
    }
    record++;
}
```

La operación de obtener la información de un inodo a partir de su número se realiza más veces, por tanto, es interesante definir una función auxiliar para ello: `assoofs_get_inode`.

3. En nuestro caso, la función debe devolver `NULL`, incluso cuando no se encuentre la entrada.

assoofs_get_inode Esta función auxiliar nos permitirá obtener un puntero al inodo número `ino` del superbloque `sb`. Su prototipo es el siguiente:

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

La función debe realizar las siguientes tareas:

1. Obtener la información persistente del inodo `ino`. Ver la función auxiliar `assoofs_get_inode_info` descrita anteriormente.
2. Crear una nueva variable de tipo `struct inode` e inicializarla con la función `new_inode` (ver anexo E). Asignar valores a los campos `i_ino`, `i_sb`, `i_op`, `i_fop`, `i_atime`, `i_mtime`, `i_ctime` e `i_private` del nuevo inodo.
   - Antes de asignar valor al campo `i_fop` debemos saber si el inodo que buscamos es un fichero o un directorio. Lo sabremos consultando el valor del campo `mode` de la información persistente del inodo obtenida en el paso 1. Para comprobarlo disponemos de las macros `S_IFDIR` y `S_IFREG`:

     ```
     struct inode *inode;
     inode_info = assoofs_get_inode_info(sb, ino);

     if (S_ISDIR(inode_info->mode))
         inode->i_fop = &assoofs_dir_operations;
     else if (S_ISREG(inode_info->mode))
         inode->i_fop = &assoofs_file_operations;
     else
         printk(KERN_ERR "Unknown inode type. Neither a directory nor a file.");
     ```

   - Usaremos la función `current_time()` para asignar la fecha del sistema a los campos `i_atime`, `i_mtime` y `i_ctime` del nuevo inodo. El prototipo de la función es el siguiente:

     ```
     struct timespec64 current_time(struct inode *inode);
     ```

   - En el campo `i_private` del nuevo inodo guardaremos la información persistente del inodo obtenida en el paso 1.
3. Por último, devolvemos el inodo `inode` recién creado.

assoofs_create Esta función nos permitirá crear nuevos inodos para archivos. Su prototipo es el siguiente:

```
static int assoofs_create(struct inode *dir, struct dentry *dentry, umode_t mode, bool excl);
```

El primer parámetro es el inodo del directorio dónde se pretende crear el archivo al que apunta el nuevo inodo. El segundo parámetro representa la entrada en el directorio padre del nuevo archivo (de aquí sacaremos el nombre). El tercer parámetro nos dice el modo del nuevo archivo (permisos). El último parámetro no lo utilizaremos.

`assoofs_create` tiene que realizar las siguientes tareas:

1. Crear el nuevo inodo, para ello sigue los pasos del apéndice E, teniendo en cuenta las siguientes consideraciones adicionales:
   - El número del nuevo inodo lo asignaré a partir de la información persistente del superbloque, que entre otras cosas me dice cuantos inodos tengo:

     ```
     struct inode *inode;
     uint64_t count;
     sb = dir->i_sb; // obtengo un puntero al superbloque desde dir
     count = ((struct assoofs_super_block_info *)sb->s_fs_info)->inodes_count; // obtengo el número de inodos de la
         información persistente del superbloque

     inode = new_inode(sb);
     inode->i_ino = (count + ASSOOFS_START_INO - ASSOOFS_RESERVED_INODES + 1); // Asigno número al nuevo inodo a partir
         de count
     ```

     En este punto debería comprobar que el valor de `count` no es superior al número máximo de objetos soportados en assoofs (`ASSOOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED`)
   - Hay que guardar en el campo `i_private` la información persistente del mismo (`struct assoofs_inode_info`). En este caso, no llamo a `assoofs_get_inode_info`, se trata de un nuevo inodo y tengo que crearlo desde cero:

     ```
     struct assoofs_inode_info *inode_info;
     inode_info = kmalloc(sizeof(struct assoofs_inode_info), GFP_KERNEL);
     inode_info->mode = mode; // El segundo mode me llega como argumento
     inode_info->file_size = 0;
     inode->i_private = inode_info;
     ```

   - Para las operaciones sobre ficheros utilizaremos `assoofs_file_operations`.

```
inode->i_fop=&assoofs_file_operations;
```

- Hay que asignarle un bloque al nuevo inodo, por lo que habrá que consultar el mapa de bits del superbloque. Esta operación se realiza más veces y es útil definir una función auxiliar para ello: `assoofs_sb_get_a_freeblock`. `assoofs_sb_get_a_freeblock` a su vez, tendrá que actualizar la información persistente del superbloque, en concreto el valor del campo `free_blocks`. Esta operación también se repite en más lugares por lo que se recimienda definir una función auxiliar: `assoofs_save_sb_info`.

```
assoofs_sb_get_a_freeblock(sb, &inode_info->data_block_number);
```

- Guardar la información persistente del nuevo inodo en disco (en el almacén de inodos). Esta operación se realiza más veces, por lo que puede ser útil definir una función auxiliar para ello: `assoofs_add_inode_info`.

```
assoofs_add_inode_info(sb, inode_info);
```

2. Modificar el contenido del directorio padre, añadiendo una nueva entrada para el nuevo archivo o directorio. El nombre lo sacaremos del segundo parámetro.

```
struct assoofs_inode_info *parent_inode_info;
struct assoofs_dir_record_entry *dir_contents;

parent_inode_info = dir->i_private;
bh = sb_bread(sb, parent_inode_info->data_block_number);

dir_contents = (struct assoofs_dir_record_entry *)bh->b_data;
dir_contents += parent_inode_info->dir_children_count;
dir_contents->inode_no = inode_info->inode_no; // inode_info es la información persistente del inodo creado en el paso 2.

strcpy(dir_contents->filename, dentry->d_name.name);
mark_buffer_dirty(bh);
sync_dirty_buffer(bh);
brelse(bh);
```

3. Actualizar la información persistente del inodo padre indicando que ahora tiene un archivo más. Se recomienda definir una función auxiliar para esta operación: `assoofs_save_inode_info`. Para actualizar la información persistente de un inodo es necesario recorrer el almacen y localizar dicho inodo, para ello se recomienda definir otra función auxiliar: `assoofs_search_inode_info`.

```
parent_inode_info->dir_children_count++;
assoofs_save_inode_info(sb, parent_inode_info);
```

assoofs_sb_get_a_freeblock Esta función auxiliar nos permitirá obtener un bloque libre:

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

Para ello debe seguir los siguientes pasos:

- Obtenemos la información persistente del superbloque que previamente habíamos guardado en el campo `s_fs_info`:

```
struct assoofs_super_block_info *assoofs_sb = sb->s_fs_info;
```

- Recorremos el mapa de bits en busca de un bloque libre (bit = 1):

```
int i;
for (i = 2; i < ASSOOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED; i++)
    if (assoofs_sb->free_blocks & (1 << i))
        break; // cuando aparece el primer bit 1 en free_block dejamos de recorrer el mapa de bits, i tiene la posición
     del primer bloque libre

*block = i; // Escribimos el valor de i en la dirección de memoria indicada como segundo argumento en la función
```

Antes de asignar el valor, conviene comprobar no hemos alcanzado el número máximo de objetos en un sistema assoofs (`ASSOOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED`).

- Por último, hay que actualizar el valor de `free_blocks` y guardar los cambios en el superbloque. Devolveremos 0 su todo ha ido bien.

```
assoofs_sb->free_blocks &= ~(1 << i);
assoofs_save_sb_info(sb);
return 0;
```

assoofs_save_sb_info Esta función auxiliar nos permitirá actualizar la información persistente del superbloque cuando hay un cambio:

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

Para hacerlo, basta con leer de disco la información persistente del superbloque con `sb_bread` y sobreescribir el campo `b_data` con la información en memoria:

```
struct buffer_head *bh;
struct assoofs_super_block *sb = vsb->s_fs_info; // Información persistente del superbloque en memoria
bh = sb_bread(vsb, ASSOOFS_SUPERBLOCK_BLOCK_NUMBER);
bh->b_data = (char *)sb; // Sobreescribo los datos de disco con la información en memoria
```

Para que el cambio pase a disco, basta con marcar el buffer como sucio y sincronizar:

```
mark_buffer_dirty(bh);
sync_dirty_buffer(bh);
brelse(bh);
```

assoofs_add_inode_info Esta función auxiliar 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);
```

- Acceder a la información persistente del superbloque (`sb->s_fs_info`) para obtener el contador de inodos (`inodes_count`).
- Leer de disco el bloque que contiene el almacén de inodos.

```
bh = sb_bread(sb, ASSOOFS_INODESTORE_BLOCK_NUMBER);
```

- Obtener un puntero al final del almacén y escribir un nuevo valor al final.

```
inode_info = (struct assoofs_inode_info *)bh->b_data;
inode_info += assoofs_sb->inodes_count;
memcpy(inode_info, inode, sizeof(struct assoofs_inode_info));
```

- Marcar el bloque como sucio y sincronizar.

```
mark_buffer_dirty(bh);
sync_dirty_buffer(bh);
```

- Actualizar el contador de inodos de la información persistente del superbloque y guardar los cambios.

```
assoofs_sb->inodes_count++;
assoofs_save_sb_info(sb);
```

assoofs_save_inode_info Esta función auxiliar 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);
```

La función tiene que realizar lo siguiente:

- Obtener de disco el almacén de inodos.
- Buscar los datos de `inode_info` en el almacén. Para ello se recomienda utilizar una función auxiliar.

```
inode_pos = assoofs_search_inode_info(sb, (struct assoofs_inode_info *)bh->b_data, inode_info);
```

- Actualizar el inodo, marcar el bloque como sucio y sincronizar.

```
memcpy(inode_pos, inode_info, sizeof(*inode_pos));
mark_buffer_dirty(bh);
sync_dirty_buffer(bh);
```

- Si todo va bien devolvemos el valor cero.

assoofs_search_inode_info Esta función auxiliar 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);
```

Para ello tenemos que recorrer el almacén de inodos, desde `start`, que marca el principio dle almacen, hasta encontrar los datos del inodo `search` o hasta el final del almacén.

```
uint64_t count = 0;
while (start->inode_no != search->inode_no && count < ((struct assoofs_super_block_info *)sb->s_fs_info)->inodes_count) {
    count++;
    start++;
}

if (start->inode_no == search->inode_no)
    return start;
else
    return NULL;
```

assoofs_mkdir Esta función nos permitirá crear nuevos inodos para directorios. Su prototipo es el siguiente:

```
static int assoofs_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode);
```

El primer parámetro es el inodo del directorio dónde se pretende crear el directorio al que apunta el nuevo inodo. El segundo parámetro representa la entrada en el directorio padre del nuevo directorio (de aquí sacaremos el nombre). El tercer parámetro nos dice el modo del nuevo directorio (permisos).

Las tareas que tiene que realizar `assoofs_mkdir` son las mismas que `assoofs_create`, solamente cambian los siguientes valores:

```
inode->i_fop=&assoofs_dir_operations;
inode_info->dir_children_count = 0;
inode_info->mode = S_IFDIR | mode;
```

El modo del nuevo inodo, que en el caso de los directorios se calcula a partir del argumento `mode` mediante la expresión `S_IFDIR | mode`.

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

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,
};
```

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,
};
```

Es necesario implementar las funciones para cada operación. Los siguientes sub-apartados explican los pasos a seguir con cada una. En algunos casos se recomienda el uso de funciones auxiliares.

assoofs_iterate Esta función permite mostrar el contenido de un directorio. Para representar el contenido de un directorio se utiliza un `struct dir_context`) que es necesario inicializar. Su prototipo es el siguiente:

```
static int assoofs_iterate(struct file *filp, struct dir_context *ctx);
```

`assoofs_iterate` debe realizar las siguientes tareas:

1. Acceder al inodo, a la información persistente del inodo, y al superbloque correspondientes al argumento `filp`:

   ```
   struct inode *inode;
   struct super_block *sb;
   assoofs_inode_info *inode_info;

   inode = filp->f_path.dentry->d_inode;
   sb = inode->i_sb;
   inode_info = inode->i_private;
   ```

2. Comprobar si el contexto del directorio ya está creado. Si no lo hacemos provocaremos un bucle infinito. Basta comprobar que el campo `pos` del contexto `ctx` es distinto de cero:

```
if (ctx->pos) return 0;
```

3. Hay que comprobar que el inodo obtenido en el paso 1 se corresponde con un directorio:

```
if ((!S_ISDIR(inode_info->mode))) return -1;
```

4. Accedemos al bloque donde se almacena el contenido del directorio y con la información que contiene inicializamos el contexto `ctx`:

```
struct buffer_head *bh;
bh = sb_bread(sb, inode_info->data_block_number);
record = (struct assoofs_dir_record_entry *)bh->b_data;
for (i = 0; i < inode_info->dir_children_count; i++) {
    dir_emit(ctx, record->filename, ASSOOFS_FILENAME_MAXLEN, record->inode_no, DT_UNKNOWN);
    ctx->pos += sizeof(struct assoofs_dir_record_entry);
    record++;
}
brelse(bh);
return 0;
```

`dir_emit` nos permite añadir nuevas entradas al contexto. Cada vez que añadamos una entrada al contexto, debemos incrementar el valor del campo `pos` con el tamaño de la nueva entrada.

assoofs_read Esta función permite leer de un archivo. Su prototipo es el siguiete:

```
ssize_t assoofs_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos);
```

El primer argumento representa el fichero que quiero leer. El segundo y el tercer argumento representan la dirección del buffer (en el espacio del usuario) y la longitud donde se copiaran los datos leídos del fichero `filp`. El último argumento es el desplazamiento respecto al principio del fichero desde donde empezará la lectura.

`assoofs_read` tiene que realizar las siguientes tareas:

- Obtener la información persistente del inodo a partir de `filp`:

```
struct assoofs_inode_info *inode_info = filp->f_path.dentry->d_inode->i_private;
```

- Comprobar el valor de `ppos` por si hemos alcanzado el final del fichero:

```
if (*ppos >= inode_info->file_size) return 0;
```

- Acceder al contenido del fichero.

```
struct buffer_head *bh;
char *buffer;

bh = sb_bread(filp->f_path.dentry->d_inode->i_sb, inode_info->data_block_number);
buffer = (char *)bh->b_data;
```

- Copiar en el buffer `buf` el contenido del fichero leído en el paso anterior con la función `copy_to_user`:

```
int nbytes;
nbytes = min((size_t) inode_info->file_size, len); // Hay que comparar len con el tamaño del fichero por si llegamos al
    final del fichero
copy_to_user(buf, buffer, nbytes);
```

- Incrementar el valor de `ppos` y devolver el número de bytes leídos.

```
*ppos += nbytes;
return nbytes;
```

assoofs_write Esta función permite escribir en un archivo. Su prototipo es el siguiente:

```
ssize_t assoofs_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos);
```

El primer argumento representa el fichero que quiero escribir. El segundo y el tercer argumento representan la dirección del buffer (en el espacio del usuario) y la longitud desde dónde se escribirán los datos en el fichero `filp`. El último argumento es el desplazamiento respecto al principio del fichero desde donde empezará la escritura.

Los pasos a seguir son similares `assoofs_read`, pero en este caso tenemos que escribir en el fichero los datos obtenidos de `buf` mediante `copy_from_user`:

```
buffer = (char *)bh->b_data;
buffer += *ppos;
copy_from_user(buffer, buf, len)
```

También hay que incrementar el valor de ppos (`*ppos+=len`) y marcar el bloque como sucio y sincronizarlo.

Por último, hay que actualizar el campo `file_size` de la información persistente del inodo y devilcer el número de bytes escritos

```
inode_info->file_size = *ppos;
assoofs_save_inode_info(sb, inode_info);
return len;
```

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/
# 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
```

A. Diseño básico de `assoofs.c`

El siguiente pseudocódigo muestra la estructura básica del dichero `assoofs.c`:

```
#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/fs.h>           /* libfs stuff           */
#include <linux/buffer_head.h>  /* buffer_head           */
#include <linux/slab.h>         /* kmem_cache            */
#include "assoofs.h"

/*
 *  Operaciones sobre ficheros
 */
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);
const struct file_operations assoofs_file_operations = {
    .read = assoofs_read,
    .write = assoofs_write,
};

ssize_t assoofs_read(struct file * filp, char __user * buf, size_t len, loff_t * ppos) {
    printk(KERN_INFO "Read request\n");
    return 0;
}

ssize_t assoofs_write(struct file * filp, const char __user * buf, size_t len, loff_t * ppos) {
    printk(KERN_INFO "Write request\n");
    return 0;
}

/*
 *  Operaciones sobre directorios
 */
static int assoofs_iterate(struct file *filp, struct dir_context *ctx);
const struct file_operations assoofs_dir_operations = {
    .owner = THIS_MODULE,
    .iterate = assoofs_iterate,
};

static int assoofs_iterate(struct file *filp, struct dir_context *ctx) {
    printk(KERN_INFO "Iterate request\n");
    return 0;
}

/*
 *  Operaciones sobre inodos
 */
static int assoofs_create(struct inode *dir, struct dentry *dentry, umode_t mode, bool excl);
struct dentry *assoofs_lookup(struct inode *parent_inode, struct dentry *child_dentry, unsigned int flags);
static int assoofs_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode);
static struct inode_operations assoofs_inode_ops = {
    .create = assoofs_create,
    .lookup = assoofs_lookup,
    .mkdir = assoofs_mkdir,
};

struct dentry *assoofs_lookup(struct inode *parent_inode, struct dentry *child_dentry, unsigned int flags) {
    printk(KERN_INFO "Lookup request\n");
    return NULL;
}


static int assoofs_create(struct inode *dir, struct dentry *dentry, umode_t mode, bool excl) {
    printk(KERN_INFO "New file request\n");
    return 0;
}

static int assoofs_mkdir(struct inode *dir , struct dentry *dentry, umode_t mode) {
    printk(KERN_INFO "New directory request\n");
    return 0;
}

/*
 *  Operaciones sobre el superbloque
 */
static const struct super_operations assoofs_sops = {
    .drop_inode = generic_delete_inode,
};

```

```c
/*
 *  Inicialización del superbloque
 */
int assoofs_fill_super(struct super_block *sb, void *data, int silent) {
    printk(KERN_INFO "assoofs_fill_super request\n");
    // 1.- Leer la información persistente del superbloque del dispositivo de bloques
    // 2.- Comprobar los parámetros del superbloque
    // 3.- Escribir la información persistente leída del dispositivo de bloques en el superbloque sb, incluído el campo s_op con las operaciones que soporta.
    // 4.- Crear el inodo raíz y asignarle operaciones sobre inodos (i_op) y sobre directorios (i_fop)
    return 0;
}

/*
 *  Montaje de dispositivos assoofs
 */
static struct dentry *assoofs_mount(struct file_system_type *fs_type, int flags, const char *dev_name, void *data) {
    printk(KERN_INFO "assoofs_mount request\n");
    struct dentry *ret = mount_bdev(fs_type, flags, dev_name, data, assoofs_fill_super);
    // Control de errores a partir del valor de ret. En este caso se puede utilizar la macro IS_ERR: if (IS_ERR(ret)) ...
}

/*
 *  assoofs file system type
 */
static struct file_system_type assoofs_type = {
    .owner   = THIS_MODULE,
    .name    = "assoofs",
    .mount   = assoofs_mount,
    .kill_sb = kill_litter_super,
};

static int __init assoofs_init(void) {
    printk(KERN_INFO "assoofs_init request\n");
    int ret = register_filesystem(&assoofs_type);
    // Control de errores a partir del valor de ret
}

static void __exit assoofs_exit(void) {
    printk(KERN_INFO "assoofs_exit request\n");
    int ret = unregister_filesystem(&assoofs_type);
    // Control de errores a partir del valor de ret
}

module_init(assoofs_init);
module_exit(assoofs_exit);
```

B. Operaciones binarias sobre `free_blocks`

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

```c
#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 = %llu\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 = %llu\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 = %llu\n", value2);
    printf("\n");

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

    return 0;
}
```

C. Cómo 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);
```

D. Cómo 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);
```

E. Cómo 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` reserva la memoria necesaria e inicializa 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. Además, en el campo `i_private` guardaremos un `struct assoofs_inode_info` con la información persistente del 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_dir_operations; // dirección de una variable de tipo struct file_operations previamente declarada.
     En la práctica tenemos 2: assoofs_dir_operations y assoofs_file_operations. La primera la utilizaremos cuando creemos
    inodos para directorios (como el directorio raíz) y la segunda cuando creemos inodos para ficheros.
root_inode->i_atime = root_inode->i_mtime = root_inode->i_ctime = current_time(root_inode); // fechas.
root_inode->i_private = assoofs_get_inode_info(sb, ASSOOFS_ROOTDIR_INODE_NUMBER); // Información persistente del inodo
```

Guardaremos la información persistente del inodo en el campo `i_private`. Esta operación se realiza en más situaciones, por tanto, es interesante definir una función auxiliar para ello: `assoofs_get_inode_info`.

Por último tenemos que introducir el nuevo inodo en el árbol de inodos. Hay dos formas de hacer esto: la primera es solamente para el directorio raíz y sólo hay que hacerlo una vez; la segunda es para el resto de inodos.

1. Cuando el nuevo inodo 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.

F. Reserva de memoria para los datos persistentes de un inodo

La función `new_inode` vista en el apéndice E permite reservar memoria e inicializar un `struct inode`. Sin embargo, en alguna parte de nuestro programa necesitaremos reservar memoria para un `struct assoofs_inode_info`. En el kernel no podemos utilizar `malloc`, en su lugar usaremos `kmalloc` cuyo prototipo se muestra a continuación:

```
void *kmalloc(size_t size, gfp_t flags);
```

Para reservar memoria para un `struct assoofs_inode_info` lo haremos como sigue, utilizando el flag `GFP_KERNEL`:

```
struct assoofs_inide_info *inode_info;
inode_info = kmalloc(sizeof(struct assoofs_inode_info), GFP_KERNEL);
```

En caso de usar una caché de inodos, utilizaremos `kmem_cache_alloc` en lugar de `kmalloc`. Para más detalles ver el apéndice G.

G. 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 mófulo 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);
}
```

H. Uso de semáforos para bloquear recursos compartidos

Algunos recursos compartidos deben protegerse de accesos concurrentes. Cómo mínimo, el superbloque y el almacen de inodos. Para declarar un semáforo mutex nuevo utilizaremos la macro `DEFINE_MUTEX` como sigue:

```
static DEFINE_MUTEX(assoofs_sb_lock);
```

Para bloquear el mutex usaremos la función `mutex_lock_interruptible`:

```
mutex_lock_interruptible(&assoofs_sb_lock);
```

Para desbloquearlo, usaremos `mutex_unlock`:

```
mutex_unlock(&assoofs_sb_lock);
```