

PRÁCTICA 2

Creación del sistema de ficheros ASSOOFS

Ampliación de Sistemas Operativos (ASSOO)
Grado en Ingeniería Informática

Universidad de León

Distributed under: Creative Commons Attribution-ShareAlike 4.0 International



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 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

- Creating Linux virtual filesystems. <https://lwn.net/Articles/57369/>
- 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>
- *libfs* stuff: <https://elixir.bootlin.com/linux/latest/source>

Índice

1. Creación de un módulo básico	3
2. Implementación de ASSOOFs	4
2.1. Estructuras de datos necesarias	4
2.2. Implementación un programa que permita formatear dispositivos de bloques como ASSOOFs	5
2.3. Implementación de un módulo para ASSOOFs	9
2.3.1. Inicializar y registrar el nuevo sistema de ficheros en el kernel	12
2.3.2. Implementar una función que permita montar dispositivos con el nuevo sistema de ficheros: <code>assoofs_mount</code>	12
2.3.3. Implementar una función para inicializar el superbloque: <code>assoofs_fill_super</code>	12
2.3.4. Declarar una estructura e implementar funciones para manejar inodos	15
2.3.5. Declarar una estructura e implementar funciones para manejar archivos y directorios	21
3. Prueba tu práctica	24
3.1. Compilación estructura básica	25
3.2. Implementadas <code>assoofs_fill_super</code> y <code>assoofs_get_inode_info</code>	25
3.3. Implementadas <code>assoofs_lookup</code> y <code>assoofs_get_inode</code>	26
3.4. Implementadas <code>assoofs_create</code> , <code>assoofs_sb_get_a_freeinode</code> , <code>assoofs_sb_get_a_freeblock</code> , <code>assoofs_save_sb_info</code> y <code>assoofs_add_inode_info</code>	26
3.5. Implementadas <code>assoofs_create</code> y <code>assoofs_mkdir</code> con todas sus funciones auxiliares	26
3.6. Implementadas <code>assoofs_remove</code> y sus funciones auxiliares	27
3.7. Parte básica	28
A. Operaciones binarias sobre <code>free_blocks</code>	30
B. Caché de inodos	30
C. Uso de semáforos para bloquear recursos compartidos	31
D. Mover archivos	31

1. Creación de un módulo básico

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
6 MODULE_LICENSE("GPL");
7 MODULE_AUTHOR("Angel Manuel Guerrero Higuera");
8
9 static int __init init_hello(void)
10 {
11     printk(KERN_INFO "Hello world\n");
12     return 0;
13 }
14
15 static void __exit cleanup_hello(void)
16 {
17     printk(KERN_INFO "Goodbye world\n");
18 }
19
20 module_init(init_hello);
21 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:

```
1 obj-m += helloWorldModule.o
2
3 all: ko
4
5 ko:
6     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
7
8 clean:
9     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.

```
1 # ls
2 helloWorldModule.c Makefile
3 # make
4 make -C /lib/modules/3.13.0-86-generic/build M=/root modules
5 make[1]: se ingresa al directorio "/usr/src/linux-headers-3.13.0-86-generic"
6 CC [M] /root/helloWorldModule.o
7 Building modules, stage 2.
8 MODPOST 1 modules
9 CC /root/helloWorldModule.mod.o
10 LD [M] /root/helloWorldModule.ko
11 make[1]: se sale del directorio "/usr/src/linux-headers-3.13.0-86-generic"
12 # insmod helloWorldModule.ko
13 # dmesg
14 ...
15 [ 2424.977652] Hello world
16 # rmmod helloWorldModule
17 # dmesg
18 ...
19 [ 2424.977652] Hello world
20 [ 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 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:

```
1 #define ASSOOFS_MAGIC 0x20200406
2 #define ASSOOFS_DEFAULT_BLOCK_SIZE 4096
3 #define ASSOOFS_FILENAME_MAXLEN 255
4 #define ASSOOFS_LAST_RESERVED_BLOCK ASSOOFS_ROOTDIR_BLOCK_NUMBER
5 #define ASSOOFS_LAST_RESERVED_INODE ASSOOFS_ROOTDIR_INODE_NUMBER
6 const int ASSOOFS_TRUE = 1;
7 const int ASSOOFS_FALSE = 0;
8 const int ASSOOFS_SUPERBLOCK_BLOCK_NUMBER = 0;
9 const int ASSOOFS_INODESTORE_BLOCK_NUMBER = 1;
10 const int ASSOOFS_ROOTDIR_BLOCK_NUMBER = 2;
11 const int ASSOOFS_ROOTDIR_INODE_NUMBER = 0;
12 const int ASSOOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED = 64;
13
14 struct assoofs_super_block_info {
15     uint64_t version;
16     uint64_t magic;
17     uint64_t block_size;
18     uint64_t inodes_count;
19     uint64_t free_blocks;
20     uint64_t free_inodes;
21     char padding[4048];
22 };
23
24 struct assoofs_dir_record_entry {
25     char filename[ASSOOFS_FILENAME_MAXLEN];
26     uint64_t inode_no;
27     uint64_t entry_removed;
28 };
29
30
31 struct assoofs_inode_info {
32     mode_t mode;
33     uint64_t inode_no;
34     uint64_t data_block_number;
35
36     union {
37         uint64_t file_size;
38         uint64_t dir_children_count;
39     };
40 };
```

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 el programa `mkassoofs.c`, cuyo código se muestra a continuación:

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <stdint.h>
7 #include <stdlib.h>
8 #include <string.h>
9 #include "assoofs.h"
10
11 #define WELCOMEFIL_DATABLOCK_NUMBER (ASSOofs_LAST_RESERVED_BLOCK + 1)
12 #define WELCOMEFIL_INODE_NUMBER (ASSOofs_LAST_RESERVED_INODE + 1)
13
14 static int write_superblock(int fd) {
15     struct assoofs_super_block_info sb = {
16         .version = 1,
17         .magic = ASSOofs_MAGIC,
18         .block_size = ASSOofs_DEFAULT_BLOCK_SIZE,
19         .inodes_count = 2, // DIRECTORIO RAIZ y README.txt
20         .free_blocks = (15), // 1111 En binario
21         .free_inodes = (3), // 11 En binario
22     };
23     ssize_t ret;
24
25     ret = write(fd, &sb, sizeof(sb));
26     if (ret != ASSOofs_DEFAULT_BLOCK_SIZE) {
27         printf("Bytes written [%d] are not equal to the default block size.\n", (int)ret);
28         return -1;
29     }
30
31     printf("Super block written succesfully.\n");
32     return 0;
33 }
34
35 static int write_root_inode(int fd) {
36     ssize_t ret;
37
38     struct assoofs_inode_info root_inode;
39
40     root_inode.mode = S_IFDIR;
41     root_inode.inode_no = ASSOofs_ROOTDIR_INODE_NUMBER;
42     root_inode.data_block_number = ASSOofs_ROOTDIR_BLOCK_NUMBER;
43     root_inode.dir_children_count = 1;
44
45     ret = write(fd, &root_inode, sizeof(root_inode));
46
47     if (ret != sizeof(root_inode)) {
48         printf("The inode store was not written properly.\n");
49         return -1;
50     }
51
52     printf("root directory inode written succesfully.\n");
53     return 0;
54 }
55
56 static int write_welcome_inode(int fd, const struct assoofs_inode_info *i) {
57     off_t nbytes;
58     ssize_t ret;
59
60     ret = write(fd, i, sizeof(*i));
61     if (ret != sizeof(*i)) {
62         printf("The welcomefile inode was not written properly.\n");
63         return -1;
64     }
65     printf("welcomefile inode written succesfully.\n");
66
67     nbytes = ASSOofs_DEFAULT_BLOCK_SIZE - (sizeof(*i) * 2);
68     ret = lseek(fd, nbytes, SEEK_CUR);
69     if (ret == (off_t)-1) {
70         printf("The padding bytes are not written properly.\n");
71         return -1;
72     }
73 }
```

```

74     printf("inode store padding bytes (after two inodes) written sucessfully.\n");
75     return 0;
76 }
77
78 int write_dirent(int fd, const struct assoofs_dir_record_entry *record) {
79     ssize_t nbytes = sizeof(*record), ret;
80
81     ret = write(fd, record, nbytes);
82     if (ret != nbytes) {
83         printf("Writing the rootdirectory datablock (name+inode_no pair for welcomefile) has failed.\n");
84         return -1;
85     }
86     printf("root directory datablocks (name+inode_no pair for welcomefile) written succesfully.\n");
87
88     nbytes = ASSOOFS_DEFAULT_BLOCK_SIZE - sizeof(*record);
89     ret = lseek(fd, nbytes, SEEK_CUR);
90     if (ret == (off_t)-1) {
91         printf("Writing the padding for rootdirectory children datablock has failed.\n");
92         return -1;
93     }
94     printf("Padding after the rootdirectory children written succesfully.\n");
95     return 0;
96 }
97
98 int write_block(int fd, char *block, size_t len) {
99     ssize_t ret;
100
101     ret = write(fd, block, len);
102     if (ret != len) {
103         printf("Writing file body has failed.\n");
104         return -1;
105     }
106     printf("block has been written succesfully.\n");
107     return 0;
108 }
109
110 int main(int argc, char *argv[])
111 {
112     int fd;
113     ssize_t ret;
114     char welcomefile_body[] = "Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.\n";
115
116     struct assoofs_inode_info welcome = {
117         .mode = S_IFREG,
118         .inode_no = WELCOMEFILE_INODE_NUMBER,
119         .data_block_number = WELCOMEFILE_DATABLOCK_NUMBER,
120         .file_size = sizeof(welcomefile_body),
121     };
122
123     struct assoofs_dir_record_entry record = {
124         .filename = "README.txt",
125         .inode_no = WELCOMEFILE_INODE_NUMBER,
126         .entry_removed = ASSOOFS_FALSE,
127     };
128
129     if (argc != 2) {
130         printf("Usage: mkassoofs <device>\n");
131         return -1;
132     }
133
134     fd = open(argv[1], O_RDWR);
135     if (fd == -1) {
136         perror("Error opening the device");
137         return -1;
138     }
139
140     ret = 1;
141     do {
142         if (write_superblock(fd))
143             break;
144
145         if (write_root_inode(fd))
146             break;
147
148         if (write_welcome_inode(fd, &welcome))
149             break;
150
151         if (write_dirent(fd, &record))
152             break;

```

```

153         if (write_block(fd, welcomefile_body, welcome.file_size))
154             break;
155     }
156     ret = 0;
157 } while (0);
158
159 close(fd);
160 return ret;
161 }
162

```

La función `main`, después de inicializar algunas variables (`welcome`, `welcome_body` y `record`) y de abrir el dispositivo indicado como segundo argumento en la línea de comandos (`open(argv[1], O_RDWR)`), ejecuta de manera secuencial las siguientes funciones:

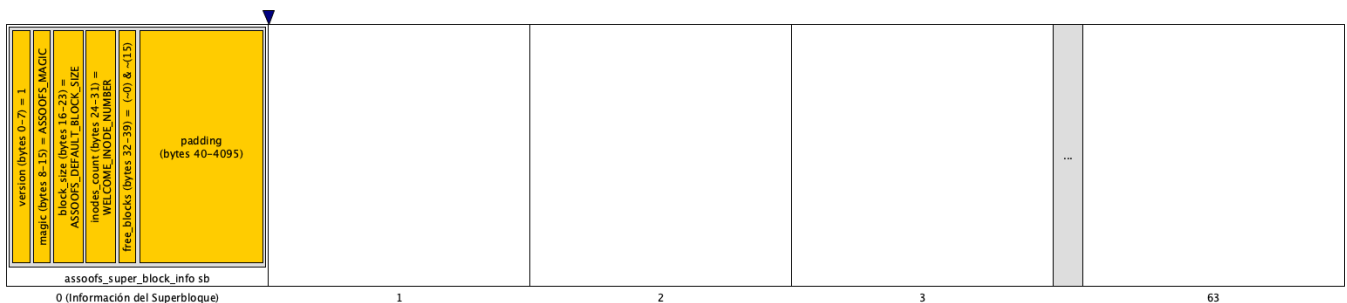
1. Primero, la función `write_superblock` escribe el superbloque en el primer bloque de disco (bloque 0).

```

1 static int write_superblock(int fd) {
2     struct assoofs_super_block_info sb = {
3         .version = 1,
4         .magic = ASSOOFS_MAGIC,
5         .block_size = ASSOOFS_DEFAULT_BLOCK_SIZE,
6         .inodes_count = 2,
7         .free_blocks = (15), // 1111 En binario
8         .free_inodes = (3), // 11 En binario
9     };
10    ssize_t ret;
11
12    ret = write(fd, &sb, sizeof(sb));
13    if (ret != ASSOOFS_DEFAULT_BLOCK_SIZE) {
14        printf("Bytes written [%d] are not equal to the default block size.\n", (int)ret);
15        return -1;
16    }
17
18    printf("Super block written succesfully.\n");
19    return 0;
20 }

```

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



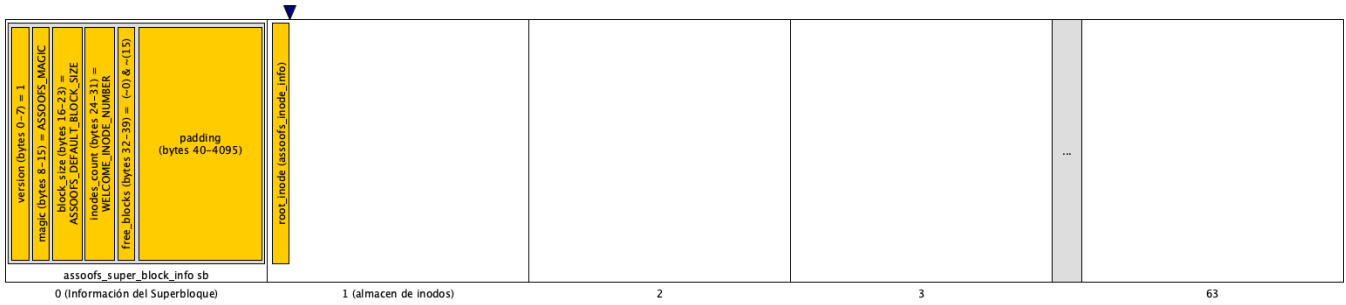
2. Después, `write_root_inode` guarda el inodo del directorio raíz en el almacén de inodos (bloque 1).

```

1 static int write_root_inode(int fd) {
2     ssize_t ret;
3
4     struct assoofs_inode_info root_inode;
5
6     root_inode.mode = S_IFDIR;
7     root_inode.inode_no = ASSOOFS_ROOTDIR_INODE_NUMBER;
8     root_inode.data_block_number = ASSOOFS_ROOTDIR_BLOCK_NUMBER;
9     root_inode.dir_children_count = 1;
10
11    ret = write(fd, &root_inode, sizeof(root_inode));
12
13    if (ret != sizeof(root_inode)) {
14        printf("The inode store was not written properly.\n");
15        return -1;
16    }
17
18    printf("root directory inode written succesfully.\n");
19    return 0;
20 }

```

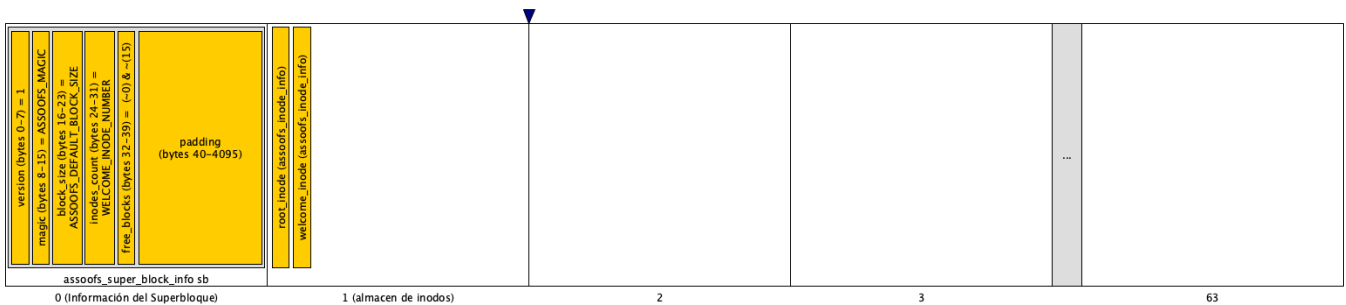
La siguiente figura muestra el contenido de un dispositivo de bloques con formato ASSOOFS después de ejecutar `write_root_inode`.



3. A continuación, `write_welcome_inode` guarda el inodo del fichero README.txt en el almacén de inodos (bloque 1).

```
1 static int write_welcome_inode(int fd, const struct assoofs_inode_info *i) {
2     off_t nbytes;
3     ssize_t ret;
4
5     ret = write(fd, i, sizeof(*i));
6     if (ret != sizeof(*i)) {
7         printf("The welcomefile inode was not written properly.\n");
8         return -1;
9     }
10    printf("welcomefile inode written succesfully.\n");
11
12    nbytes = ASSOOFS_DEFAULT_BLOCK_SIZE - (sizeof(*i) * 2);
13    ret = lseek(fd, nbytes, SEEK_CUR);
14    if (ret == (off_t)-1) {
15        printf("The padding bytes are not written properly.\n");
16        return -1;
17    }
18
19    printf("inode store padding bytes (after two inodes) written sucessfully.\n");
20    return 0;
21 }
```

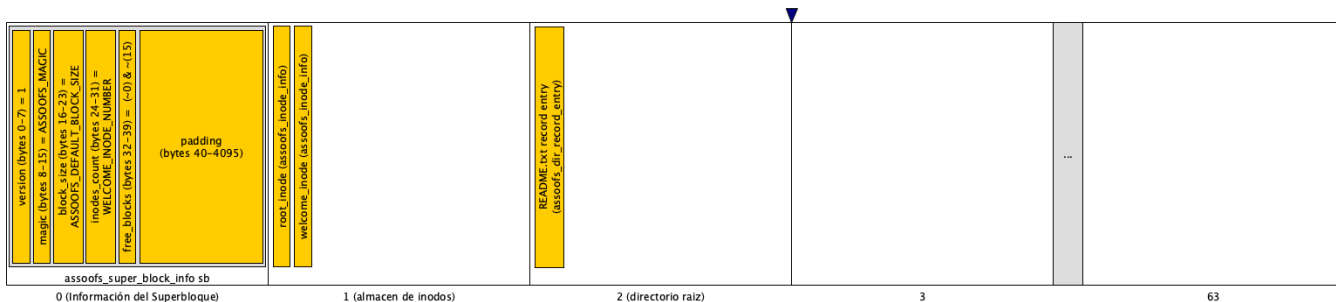
La siguiente figura muestra el contenido de un dispositivo de bloques con formato ASSOOFS después de ejecutar `write_welcome_inode`.



4. Después, `write_dirent` guarda una entrada (`<nombre, número de inodo>`) para el fichero README.txt en el bloque que almacena las entradas del directorio raíz (bloque 2).

```
1 int write_dirent(int fd, const struct assoofs_dir_record_entry *record) {
2     ssize_t nbytes = sizeof(*record), ret;
3
4     ret = write(fd, record, nbytes);
5     if (ret != nbytes) {
6         printf("Writing the rootdirectory datablock (name+inode_no pair for welcomefile) has failed.\n");
7         return -1;
8     }
9     printf("root directory datablocks (name+inode_no pair for welcomefile) written succesfully.\n");
10
11    nbytes = ASSOOFS_DEFAULT_BLOCK_SIZE - sizeof(*record);
12    ret = lseek(fd, nbytes, SEEK_CUR);
13    if (ret == (off_t)-1) {
14        printf("Writing the padding for rootdirectory children datablock has failed.\n");
15        return -1;
16    }
17    printf("Padding after the rootdirectory children written succesfully.\n");
18    return 0;
19 }
```

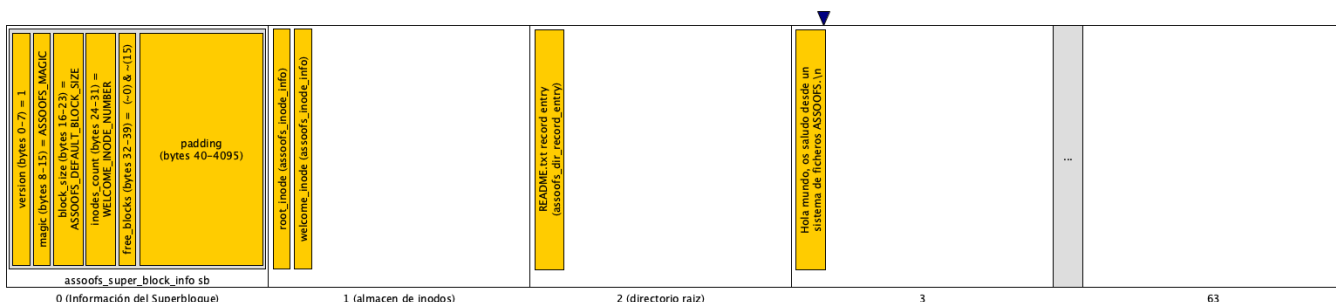

La siguiente figura muestra el contenido de un dispositivo de bloques con formato ASSOOFS después de ejecutar `write_dirent`.



5. Por último, `write_block` escribe un mensaje en el bloque que almacena los contenidos del fichero README.txt (bloque 3).

```
1 int write_block(int fd, char *block, size_t len) {
2     ssize_t ret;
3
4     ret = write(fd, block, len);
5     if (ret != len) {
6         printf("Writing file body has failed.\n");
7         return -1;
8     }
9     printf("block has been written succesfully.\n");
10    return 0;
11 }
```

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



2.3. Implementación de un módulo para ASSOOFS

El desarrollo de la práctica se realizará sobre la distribución Linux Ubuntu 24.04 con versión del kernel 6.11.0-17-generic., en este enlace se encuentra una OVA de VirtualBox con la configuración necesaria para poder empezar a trabajar. La contraseña es: **osboxes.org**. Se recomienda no desarrollar dentro de la máquina virtual, solamente utilizarla para probar el código desarrollado. Puedes crear una carpeta compartida entre tu host y la maquina virtual de forma que puedes desarrollar en tu host y utilizar la máquina virtual únicamente para probar el código.

Para implementar un sistema de ficheros básico es necesario seguir los pasos que se enumeran a continuación. Estas funcionalidades suponen los **requisitos básicos** que debe cumplir un sistema de ficheros ASSOOFS (8 puntos):

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

Además de las funcionalidades básicas enumeradas, para obtener la máxima puntuación es necesario implementar las siguientes funcionalidades **avanzadas** (2 puntos):

- Mantener una cache de inodos. Ver Anexo B.

- Utilizar semáforos para acceder a las estructuras principales. Ver Anexo C.
- Mover ficheros con el comando mv. Ver Anexo D.

El detalle de cada paso se describe en los siguientes sub-apartados. Cómo base podemos utilizar la estructura básica del módulo con las funciones principales de `assoofs.c`, cuyo código se muestra a a continuación (disponible en el moodle de la asignatura). Además, en la Sección 3 se presenta de forma genérica cual es el proceso de compilación de la solución y el proceso de creación, formateado y montaje de un dispositivo ASSOOFS. En esa misma Sección se ha añadido una serie de posibles pruebas que se pueden ir realizando en cada una de las sesiones de prácticas para asegurar que el desarrollo se está realizando de forma correcta hasta este momento. Para saber cuando es necesario hacer alguna de estas comprobaciones lo

identificaremos a lo largo de la práctica de la siguiente forma:  **CHECKPOINT**

```

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 <linux/buffer_head.h>    /* buffer_head */
6  #include <linux/slab.h>            /* kmem_cache */
7  #include "assoofs.h"
8
9  MODULE_LICENSE("GPL");
10
11 /*
12  * Prototipos de funciones
13  */
14 static struct dentry *assoofs_mount(struct file_system_type *fs_type, int flags, const char *dev_name, void *data);
15 int assoofs_fill_super(struct super_block *sb, void *data, int silent);
16 ssize_t assoofs_read(struct file * filp, char __user * buf, size_t len, loff_t * ppos);
17 ssize_t assoofs_write(struct file * filp, const char __user * buf, size_t len, loff_t * ppos);
18 static int assoofs_iterate(struct file *filp, struct dir_context *ctx);
19 static int assoofs_create(struct mnt_idmap *idmap, struct inode *dir, struct dentry *dentry, umode_t mode, bool excl);
20 struct dentry *assoofs_lookup(struct inode *parent_inode, struct dentry *child_dentry, unsigned int flags);
21 static int assoofs_mkdir(struct mnt_idmap *idmap, struct inode *dir, struct dentry *dentry, umode_t mode);
22 static int assoofs_remove(struct inode *dir, struct dentry *dentry);
23
24 /*
25  * Estructuras de datos necesarias
26  */
27
28 // Definicion del tipo de sistema de archivos assoofs
29 static struct file_system_type assoofs_type = {
30     .owner      = THIS_MODULE,
31     .name       = "assoofs",
32     .mount      = assoofs_mount,
33     .kill_sb    = kill_block_super,
34 };
35
36 // Operaciones sobre ficheros
37 const struct file_operations assoofs_file_operations = {
38     .read = assoofs_read,
39     .write = assoofs_write,
40 };
41
42 // Operaciones sobre directorios
43 const struct file_operations assoofs_dir_operations = {
44     .owner = THIS_MODULE,
45     .iterate_shared = assoofs_iterate,
46 };
47
48 // Operaciones sobre inodos
49 static struct inode_operations assoofs_inode_ops = {
50     .create = assoofs_create,
51     .lookup = assoofs_lookup,
52     .mkdir = assoofs_mkdir,
53     .unlink = assoofs_remove,
54     .rmdir = assoofs_remove,
55 };
56
57 // Operaciones sobre el superbloque
58 static const struct super_operations assoofs_sops = {
59     .drop_inode = generic_delete_inode,
60 };
61
62 /*
63  * Funciones que realizan operaciones sobre ficheros
64  */
65
66 ssize_t assoofs_read(struct file * filp, char __user * buf, size_t len, loff_t * ppos) {
67     printk(KERN_INFO "Read request\n");
68 }

```

```

67     return 0;
68 }
69
70 ssize_t assoofs_write(struct file * filp, const char __user * buf, size_t len, loff_t * ppos) {
71     printk(KERN_INFO "Write request\n");
72     return 0;
73 }
74
75 /*
76  * Funciones que realizan operaciones sobre directorios
77  */
78
79 static int assoofs_iterate(struct file *filp, struct dir_context *ctx) {
80     printk(KERN_INFO "Iterate request\n");
81     return 0;
82 }
83
84 /*
85  * Funciones que realizan operaciones sobre inodos
86  */
87 struct dentry *assoofs_lookup(struct inode *parent_inode, struct dentry *child_dentry, unsigned int flags) {
88     printk(KERN_INFO "Lookup request\n");
89     return NULL;
90 }
91
92
93 static int assoofs_create(struct mnt_idmap *idmap, struct inode *dir, struct dentry *dentry, umode_t mode, bool excl) {
94     printk(KERN_INFO "New file request\n");
95     return 0;
96 }
97
98 static int assoofs_mkdir(struct mnt_idmap *idmap, struct inode *dir, struct dentry *dentry, umode_t mode) {
99     printk(KERN_INFO "New directory request\n");
100    return 0;
101 }
102
103 static int assoofs_remove(struct inode *dir, struct dentry *dentry){
104     printk(KERN_INFO "assoofs_remove request\n");
105     return 0;
106 }
107
108 /*
109  * Inicialización del superbloque
110  */
111 int assoofs_fill_super(struct super_block *sb, void *data, int silent) {
112     printk(KERN_INFO "assoofs_fill_super request\n");
113     // 1.- Leer la información persistente del superbloque del dispositivo de bloques
114     // 2.- Comprobar los parámetros del superbloque
115     // 3.- Escribir la información persistente leída del dispositivo de bloques en el superbloque sb, incluido el campo s_op
116     //    con las operaciones que soporta.
117     // 4.- Crear el inodo raíz y asignarle operaciones sobre inodos (i_op) y sobre directorios (i_fop)
118     return 0;
119 }
120
121 /*
122  * Montaje de dispositivos assoofs
123  */
124 static struct dentry *assoofs_mount(struct file_system_type *fs_type, int flags, const char *dev_name, void *data) {
125     struct dentry *ret;
126     printk(KERN_INFO "assoofs_mount request\n");
127     ret = mount_bdev(fs_type, flags, dev_name, data, assoofs_fill_super);
128     // Control de errores a partir del valor de retorno. En este caso se puede utilizar la macro IS_ERR: if (IS_ERR(ret)) ...
129     return ret;
130 }
131
132
133 static int __init assoofs_init(void) {
134     int ret;
135     printk(KERN_INFO "assoofs_init request\n");
136     ret = register_filesystem(&assoofs_type);
137     // Control de errores a partir del valor de retorno
138     return ret;
139 }
140
141 static void __exit assoofs_exit(void) {
142     int ret;
143     printk(KERN_INFO "assoofs_exit request\n");
144     ret = unregister_filesystem(&assoofs_type);

```

```

145 // Control de errores a partir del valor de retorno
146 }
147
148 module_init(assoofs_init);
149 module_exit(assoofs_exit);

```

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:

```

1 extern int register_filesystem(struct file_system_type *);
2 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:

```

1 static struct file_system_type assoofs_type = {
2     .owner      = THIS_MODULE,
3     .name       = "assoofs",
4     .mount      = assoofs_mount,
5     .kill_sb    = kill_block_super,
6 };

```

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:

```

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

```


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

```

1 extern struct dentry *mount_bdev(struct file_system_type *fs_type,
2     int flags, const char *dev_name, void *data,
3     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`.

 **CHECKPOINT** Una vez llegados a este punto, podemos probar nuestra práctica siguiendo los comandos que se proponen en la Sección 3.1.

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

El prototipo de `assoofs_fill_super` es el siguiente:

```

1 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. En nuestro caso la información del superbloque está en el bloque 0.

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`:

```

1 static inline struct buffer_head *sb_bread(struct super_block *sb, sector_t block)
2 {
3     return __bread(sb->s_bdev, block, sb->s_blocksize);
4 }

```

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:

```

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

```

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

```

1 brelse(bh);

```

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 *ASSOOF_MAGIC* definido en *assoofs.h* al campo *s_magic* del superbloque *sb*.
- Asignaremos el tamaño de bloque *ASSOOF_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:

```

1 static const struct super_operations assoofs_sops = {
2     .drop_inode     = generic_delete_inode,
3 };

```

- 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 punto 1) en el campo *s_fs_info* del superbloque *sb*.

4. Crear el inodo raíz. Es necesario seguir los siguientes pasos:

- 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.

```

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

```

new_inode reserva la memoria necesaria e inicializa una variable de tipo *struct inode*:

```

1 struct inode *root_inode;
2 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:

```

1 extern void inode_init_owner(struct mnt_idmap *idmap, struct inode *inode, const struct inode *dir, mode_t mode);

```

Y que se invoca como sigue:

```

1 inode_init_owner(&nop_mnt_idmap, root_inode, NULL, S_IFDIR); // S_IFDIR para directorios, S_IFREG para ficheros.

```

El primer argumento, es una variable global definida en el kernel para gestionar la traducción de identificadores de usuario dentro del espacio de nombres del sistemas de archivos montado. El segundo argumento es el inodo raíz. El tercer 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. Por último, el cuarto argumento permite definir si estamos creando un inodo para un directorio o para un fichero.

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. 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. Además, en el campo *i_private* guardaremos un *struct assoofs_inode_info* con la información persistente del inodo.

```

1 root_inode->i_ino = ASSOOF_ROOTDIR_INODE_NUMBER; // número de inodo
2 root_inode->i_sb = sb; // puntero al superbloque
3 root_inode->i_op = &assoofs_inode_ops; // dirección de una variable de tipo struct inode_operations previamente declarada
4 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.

```

```

5 struct timespec64 ts = current_time(root_inode); // fechas.
6 inode_set_ctime(root_inode, ts.tv_sec, ts.tv_nsec);
7 inode_set_mtime(root_inode, ts.tv_sec, ts.tv_nsec);
8 inode_set_atime(root_inode, ts.tv_sec, ts.tv_nsec);
9 root_inode->i_private = assoofs_get_inode_info(sb, ASSOOFS_ROOTDIR_INODE_NUMBER); // Información persistente del
    inode

```

Guardaremos la información persistente del inodo 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`.

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.

- a) 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:

```

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

```

Y se invoca como sigue:

```

1 sb->s_root = d_make_root(root_inode);

```

- b) 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:

```

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

```

Y se invoca como sigue:

```

1 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.

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:

```

1 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:

```

1 struct assoofs_inode_info *inode_info = NULL;
2 struct buffer_head *bh;
3
4 bh = sb_bread(sb, ASSOOFS_INODESTORE_BLOCK_NUMBER);
5 inode_info = (struct assoofs_inode_info *)bh->b_data;

```

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

```

1 struct assoofs_super_block_info *afs_sb = sb->s_fs_info;
2 struct assoofs_inode_info *buffer = NULL;
3 int i;
4 for (i = 0; i < afs_sb->inodes_count; i++) {
5     if (inode_info->inode_no == inode_no) {
6         buffer = kmalloc(sizeof(struct assoofs_inode_info), GFP_KERNEL);
7         memcpy(buffer, inode_info, sizeof(*buffer));
8         break;
9     }
10    inode_info++;
11 }

```

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

```

1 brelse(bh);
2 return buffer;

```



CHECKPOINT Una vez llegados a este punto, podemos probar nuestra práctica siguiendo los comandos que se proponen en la Sección 3.2.

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:

```
1 static struct inode_operations assoofs_inode_ops = {
2     .lookup = assoofs_lookup,
3     .create = assoofs_create,
4     .mkdir = assoofs_mkdir,
5     .unlink = assoofs_remove,
6     .rmdir = assoofs_remove,
7 };
```

Es necesario implementar las funciones para cada operación. Los siguientes sub-apartados explican los pasos a seguir para implementar 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 concreto (`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:

```
1 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 sirve para definir una serie de flags pero 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`.

```
1 struct assoofs_inode_info *parent_info = parent_inode->i_private;
2 struct super_block *sb = parent_inode->i_sb;
3 struct buffer_head *bh;
4 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 y comprobamos que no esté marcado como eliminado. Si se localiza la entrada, entonces tenemos que construir el inodo correspondiente (*).

```
1 struct assoofs_dir_record_entry *record;
2 record = (struct assoofs_dir_record_entry *)bh->b_data;
3 for (i=0; i < parent_info->dir_children_count; i++) {
4     if (!strcmp(record->filename, child_dentry->d_name.name) && record->entry_removed == ASSOOFS_FALSE) {
5         struct inode *inode = assoofs_get_inode(sb, record->inode_no); // Función auxiliar que obtiene la información
6         de un inodo a partir de su número de inodo.
7         inode_init_owner(&nop_mnt_idmap, inode, parent_inode, ((struct assoofs_inode_info *)inode->i_private)->mode);
8         d_add(child_dentry, inode);
9         return NULL;
10    }
11    record++;
12 }
```

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:

```
1 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` (antes utilizada (*) en la página 13). Asignar valores a los campos `i_ino`, `i_sb`, `i_op`, `i_fop` 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_ISDIR` y `S_ISREG`:


```

1 struct inode *inode;
2 inode_info = assoofs_get_inode_info(sb, ino);
3
4 if (S_ISDIR(inode_info->mode))
5     inode->i_fop = &assoofs_dir_operations;
6 else if (S_ISREG(inode_info->mode))
7     inode->i_fop = &assoofs_file_operations;
8 else
9     printk(KERN_ERR "Unknown inode type. Neither a directory nor a file.");

```

- Usaremos las funciones `inode_set_ctime()`, `inode_set_mtime()` y `inode_set_atime()` para asignar la fecha de creación, acceso y modificación del nuevo inodo. El prototipo de las funciones son los siguientes:


```

1 struct timespec64 inode_set_ctime(struct inode *inode, time64_t sec, long nsec);
2 struct timespec64 inode_set_mtime(struct inode *inode, time64_t sec, long nsec);
3 struct timespec64 inode_set_atime(struct inode *inode, time64_t sec, long nsec);

```

- 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.

 **CHECKPOINT** Una vez llegados a este punto, podemos probar nuestra práctica siguiendo los comandos que se proponen en la Sección 3.3.

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

```

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

```

El primer argumento permite realizar la gestión de identificadores de usuario dentro del espacio de nombres del sistemas de archivos montado. El segundo parámetro es el inodo del directorio dónde se pretende crear el archivo al que apunta el nuevo inodo. El tercer parámetro representa la entrada en el directorio padre del nuevo archivo (de aquí sacaremos el nombre). El cuarto 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 descritos anteriormente (*) en la página 13**, teniendo en cuenta las siguientes consideraciones adicionales:

- El número del nuevo inodo se asigna a partir del mapa de bits del superbloque que nos permite conocer que inodos están disponibles y cuales no. Vamos a definir una función auxiliar para ello para llevar a cabo esta operación: `assoofs_sb_get_a_freeinode`. `assoofs_sb_get_a_freeinode` a su vez, tendrá que actualizar la información persistente del superbloque, en concreto el valor del campo `free_inodes`. Esta operación también se repite en más lugares por lo que definiremos una función auxiliar: `assoofs_save_sb_info`.

```

1 struct inode *inode;
2 sb = dir->i_sb;
3
4 inode = new_inode(sb);
5 inode->i_sb = sb;
6 struct timespec64 ts = current_time(inode);
7 inode_set_ctime(inode, ts.tv_sec, ts.tv_nsec);
8 inode_set_mtime(inode, ts.tv_sec, ts.tv_nsec);
9 inode_set_atime(inode, ts.tv_sec, ts.tv_nsec);
10
11 inode->i_op = &assoofs_inode_ops;
12 assoofs_sb_get_a_freeinode(sb, &inode->i_ino); // Obtenemos el numero de inodo

```

- 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:

```

1 struct assoofs_inode_info *inode_info;
2 inode_info = kmalloc(sizeof(struct assoofs_inode_info), GFP_KERNEL);
3 inode_info->inode_no = inode->i_ino;
4 inode_info->mode = mode; // El segundo mode me llega como argumento
5 inode_info->file_size = 0;
6 inode->i_private = inode_info;

```

- Para las operaciones sobre ficheros utilizaremos `assoofs_file_operations`.

```

1 inode->i_fop=&assoofs_file_operations;

```

- Asignamos propietario y permisos, y guardamos el nuevo inodo en el árbol de directorios.


```

1 inode_init_owner(&nop_mnt_idmap, inode, dir, mode);
2 d_add(dentry, inode);

```

- Hay que asignarle un bloque al nuevo inodo, por lo que habrá que consultar el mapa de bits del superbloque que nos permite conocer que bloques están libres. 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`, por lo que hará uso de la función auxiliar previamente creada `assoofs_save_sb_info`.

```

1 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`.

```

1 assoofs_add_inode_info(sb, inode_info);

```

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

```

1 struct assoofs_inode_info *parent_inode_info;
2 struct assoofs_dir_record_entry *dir_contents;
3
4 parent_inode_info = dir->i_private;
5 bh = sb_bread(sb, parent_inode_info->data_block_number);
6
7 dir_contents = (struct assoofs_dir_record_entry *)bh->b_data;
8 dir_contents += parent_inode_info->dir_children_count;
9 dir_contents->inode_no = inode_info->inode_no; // inode_info es la información persistente del inodo creado en el paso 2.
10
11 strcpy(dir_contents->filename, dentry->d_name.name);
12 mark_buffer_dirty(bh);
13 sync_dirty_buffer(bh);
14 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 almacén y localizar dicho inodo, para ello se recomienda definir otra función auxiliar: `assoofs_search_inode_info`.

```

1 parent_inode_info->dir_children_count++;
2 assoofs_save_inode_info(sb, parent_inode_info);

```

4. Por último devolvemos el valor cero para indicar que todo ha ido bien.

```

1 return 0;

```

assoofs_sb_get_a_freeinode Esta función permite obtener un inodo libre, su prototipo es el siguiente:

```

1 int assoofs_sb_get_a_freeinode(struct super_block *sb, unsigned long *inode)

```

`assoofs_sb_get_a_freeinode` debe realizar las siguientes operaciones:

1. Obtenemos la información persistente del superbloque que previamente habíamos guardado en el campo `s.fs_info`:

```

1 struct assoofs_super_block_info *assoofs_sb = sb->s_fs_info;

```

2. Recorrer el mapa de bits `free_inodes` en busca de un inodo libre (bit = 0):

```

1 for (i = 1; i < ASSOOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED; i++){
2     if (!(assoofs_sb->free_inodes) & (1 << i)){
3         break;
4     }
5 }
6 *inode = i;

```

Antes de asignar el valor, conviene comprobar que no hemos alcanzado el número máximo de objetos que soporta el sistema de ficheros (`ASSOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED`).

3. Por último, hay que actualizar el valor `free_inodes` y guardar los cambios en el superbloque. Devolvemos 0 si todo ha ido bien.

```

1 assoofs_sb->free_inodes |= (1 << i);
2 assoofs_save_sb_info(sb);
3 return 0;

```

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

```
1 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 sobrescribir el campo `b_data` con la información en memoria:

```
1 struct buffer_head *bh;  
2 struct assoofs_super_block_info *sb = vsb->s_fs_info; // Información persistente del superbloque en memoria  
3 bh = sb_bread(vsb, ASSOOFS_SUPERBLOCK_BLOCK_NUMBER);  
4 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:

```
1 mark_buffer_dirty(bh);  
2 sync_dirty_buffer(bh);  
3 brelse(bh);
```

assoofs_sb_get_a_freeblock Esta función auxiliar nos permitirá obtener un bloque libre:

```
1 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`:

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

- Recorremos el mapa de bits en busca de un bloque libre (bit = 0):

```
1 int i;  
2 for (i = 2; i < ASSOOFS_MAX_FILESYSTEM_OBJECTS_SUPPORTED; i++)  
3     if (~(assoofs_sb->free_blocks) & (1 << i))  
4         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  
5  
6 *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 que no hemos alcanzado el número máximo de objetos en un sistema assoofs (`ASSOOF_MAX_FILESYSTEM_OBJECTS_SUPPORTED`).

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

```
1 assoofs_sb->free_blocks |= (1 << i);  
2 assoofs_save_sb_info(sb);  
3 return 0;
```

assoofs_add_inode_info Esta función auxiliar nos permitirá guardar en disco la información persistente de un inodo nuevo:

```
1 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.

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

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

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

- Marcar el bloque como sucio y sincronizar.

```
1 mark_buffer_dirty(bh);  
2 sync_dirty_buffer(bh);
```

- Si el número de inodo asociado es igual o mayor que el contador de inodos del superbloque tenemos que actualizarlo y guardar los cambios. Si no se cumple esta condición no es necesario, significa que al nuevo inodo se le ha asignado el identificador de un inodo que ha sido eliminado previamente.

```
1 if (assoofs_sb->inodes_count <= inode->inode_no){
2     assoofs_sb->inodes_count++;
3     assoofs_save_sb_info(sb);
4 }
```

 **CHECKPOINT** Una vez llegados a este punto, podemos probar nuestra práctica siguiendo los comandos que se proponen en la Sección 3.4.

assoofs_save_inode_info Esta función auxiliar nos permitirá actualizar en disco la información persistente de un inodo:

```
1 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.

```
1 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.

```
1 memcpy(inode_pos, inode_info, sizeof(*inode_pos));
2 mark_buffer_dirty(bh);
3 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:

```
1 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 del almacen, hasta encontrar los datos del inodo `search` o hasta el final del almacén.

```
1 uint64_t count = 0;
2 while (start->inode_no != search->inode_no && count < ((struct assoofs_super_block_info *)sb->s_fs_info)->inodes_count) {
3     count++;
4     start++;
5 }
6
7 if (start->inode_no == search->inode_no)
8     return start;
9 else
10    return NULL;
```

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

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

El primer argumento permite realizar la gestión de identificadores de usuario dentro del espacio de nombres del sistemas de archivos montado. El segundo parámetro es el inodo del directorio dónde se pretende crear el directorio al que apunta el nuevo inodo. El tercer parámetro representa la entrada en el directorio padre del nuevo directorio (de aquí sacaremos el nombre). El cuarto 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:

```
1 inode->i_fop=&assoofs_dir_operations;
2 inode_info->dir_children_count = 0;
3 inode_info->mode = S_IFDIR | mode;
4 inode_init_owner(&nop_mnt_idmap, inode, dir, inode_info->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`.

Ojo en `assoofs_create` y `assoofs_mkdir` estamos añadiendo siempre las entradas de directorio al final. Podríamos reutilizar aquellas que tienen la variable `entry_removed == ASSOFS_TRUE`, pero en la implementación de ASSOFS no vamos a llegar a ocupar todo el bloque (4096 bits) por lo que no es necesario implementarlo.

✔ **CHECKPOINT** Una vez llegados a este punto, podemos probar nuestra práctica siguiendo los comandos que se proponen en la Sección 3.5.

assoofs_remove Esta función permite eliminar inodos de archivos. Su prototipo es el siguiente:

```
1 static int assoofs_remove(struct inode *dir, struct dentry *dentry);
```

`assoofs_remove` tiene que realizar las siguientes tareas:

1. Obtener el superbloque, el inodo del archivo que vamos a eliminar, la información persistente del inodo que vamos a eliminar y la información persistente del inodo que contiene el archivo:

```
1 struct super_block *sb = dir->i_sb;
2 struct inode *inode_remove = dentry->d_inode;
3 struct assoofs_inode_info *inode_info_remove = inode_remove->i_private;
4 struct assoofs_inode_info *parent_inode_info = dir->i_private;
```

2. Marcar la entrada de directorio en el bloque que contiene la información del directorio que contiene el archivo como libre variable `entry_removed = ASSOFS_TRUE`.

```
1 bh = sb_bread(sb, parent_inode_info->data_block_number);
2 dir_contents = (struct assoofs_dir_record_entry*)bh->b_data;
3 for(i = 0; i < parent_inode_info->dir_children_count; i++){
4     if (!strcmp(dir_contents->filename, dentry->d_name.name) && dir_contents->inode_no == inode_remove->i_ino){
5         printk(KERN_INFO "Found dir_record_entry to remove: %s\n", dir_contents->filename);
6         dir_contents->entry_removed = ASSOFS_TRUE;
7         break;
8     }
9     dir_contents++;
10 }
11 mark_buffer_dirty(bh);
12 sync_dirty_buffer(bh);
13 brelse(bh);
```

3. Actualizar el mapa de bits `free_inodes` del superbloque indicando que el inodo que antes ocupaba el archivo está libre. Para ello vamos a crear una función auxiliar `assoofs_sb_set_a_freeinode` a la cual le indicaremos el número de inodo que queremos marcar como libre.

```
1 assoofs_sb_set_a_freeinode(sb, inode_info_remove->inode_no);
```

4. Actualizar el mapa de bits `free_blocks` del superbloque indicando que el bloque que antes estaba ocupado ahora se encuentra libre. Para ello vamos a crear una función auxiliar `assoofs_sb_set_a_freeblock` a la cual le indicaremos el número de bloque que queremos marcar como libre.

```
1 assoofs_sb_set_a_freeblock(sb, inode_info_remove->data_block_number);
```

5. Devolvemos el valor 0 para indicar que todo ha ido bien.

```
1 return 0;
```

assoofs_sb_set_a_freeinode Esta función permite actualizar el campo `free_inodes` del superbloque. Su prototipo es el siguiente:

```
1 int assoofs_sb_set_a_freeinode(struct super_block *sb, uint64_t inode_no);
```

`assoofs_sb_set_a_freeinode` debe realizar las siguientes operaciones:

1. Sacar la información persistente del superbloque.

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

2. Actualizar el campo `free_inodes` indicando el inodo que pasa a estar libre. Y actualizar la información persistente del superbloque en disco:

```
1 struct assoofs_super_block_info *assoofs_sb = sb->s_fs_info;
2 assoofs_sb->free_inodes &= ~(1 << inode_no);
3 assoofs_save_sb_info(sb);
4 return 0;
```

assoofs_sb_set_a_freeblock Esta función permite actualizar el campo `free.blocks` del superbloque. Su prototipo es el siguiente:

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


`assoofs_sb_set_a_freeblock` debe realizar las siguientes operaciones:

1. Sacar la información persistente del superbloque.

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

2. Actualizar el campo `free.blocks` indicando el bloque que pasa a estar libre. Y actualizar la información persistente del superbloque en disco:

```
1 struct assoofs_super_block_info *assoofs_sb = sb->s_fs_info;
2 assoofs_sb->free_blocks &= ~(1 << block);
3 assoofs_save_sb_info(sb);
4 return 0;
```

 **CHECKPOINT** Una vez llegados a este punto, podemos probar nuestra práctica siguiendo los comandos que se proponen en la Sección 3.6.

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:

```
1 const struct file_operations assoofs_dir_operations = {
2     .owner = THIS_MODULE,
3     .iterate_shared = assoofs_iterate,
4 };
```

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

```
1 const struct file_operations assoofs_file_operations = {
2     .read = assoofs_read,
3     .write = assoofs_write,
4 };
```

Es necesario implementar las funciones para cada operación. Los siguientes sub-apartados explican los pasos a seguir con cada una.

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. El prototipo de `assoofs_iterate` es el siguiente:

```
1 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`:

```
1 struct inode *inode;
2 struct super_block *sb;
3 struct assoofs_inode_info *inode_info;
4
5 inode = filp->f_path.dentry->d_inode;
6 sb = inode->i_sb;
7 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:

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

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

```
1 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 (si no ha sido eliminada) inicializamos el contexto `ctx`:

```

1 struct buffer_head *bh;
2 bh = sb_bread(sb, inode_info->data_block_number);
3 record = (struct assoofs_dir_record_entry *)bh->b_data;
4 for (i = 0; i < inode_info->dir_children_count; i++) {
5     if(record->entry_removed == ASSOOFS_FALSE){
6         dir_emit(ctx, record->filename, ASSOOFS_FILENAME_MAXLEN, record->inode_no, DT_UNKNOWN);
7         ctx->pos += sizeof(struct assoofs_dir_record_entry);
8     }
9     record++;
10 }
11 brelse(bh);
12 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 siguiente:

```

1 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 copiarán 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`:

```

1 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:

```

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

```

- Acceder al contenido del fichero.

```

1 struct buffer_head *bh;
2 char *buffer;
3
4 bh = sb_bread(filp->f_path.dentry->d_inode->i_sb, inode_info->data_block_number);
5 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`:

```

1 int nbytes;
2 buffer += *ppos; // Incrementamos el buffer para que lea a partir de donde se quedo
3 nbytes = min((size_t) inode_info->file_size - (size_t) *ppos, len); // Hay que comparar len con el tamaño del fichero
4 // menos los bytes leídos hasta el momento, por si llegamos al final del fichero
5 copy_to_user(buf, buffer, nbytes);

```

La función `copy_to_user` devuelve cero en caso de que haya ido todo bien, y el número de bytes que no ha podido leer en caso de error. Es necesario controlar el valor que devuelve.

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

```

1 *ppos += nbytes;
2 return nbytes;

```

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

```

1 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` con algunas modificaciones:

- Comprobar que el valor de `ppos` mas la longitud de los datos que queremos escribir en el archivo, no sea superior al tamaño máximo del bloque:

```

1 if (*ppos + len >= ASSOFS_DEFAULT_BLOCK_SIZE){
2     printk(KERN_ERR "No hay suficiente espacio en el disco para escribir.\n");
3     return -ENOSPC;
4 }

```

- Escribir en el fichero los datos obtenidos de `buf` mediante `copy_from_user`:

```

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

```

La función `copy_from_user` devuelve cero en caso de que haya ido todo bien, y el número de bytes que no ha podido escribir en caso de error. Es necesario controlar el valor que devuelve.

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

```

1 *ppos+=len;
2 mark_buffer_dirty(bh);
3 sync_dirty_buffer(bh);

```

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

```

1 inode_info->file_size = *ppos;
2 assoofs_save_inode_info(sb, inode_info);
3 return len;

```



CHECKPOINT Una vez llegados a este punto, hemos finalizado la parte básica de la práctica, la podremos probar, siguiendo los comandos que se proponen en la Sección 3.7.

3. Prueba tu práctica

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

- La implementación del módulo está en el 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.

```
1 obj-m := assoofs.o
2
3 all: ko mkassoofs
4
5 ko:
6     make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
7
8 mkassoofs_SOURCES:
9     mkassoofs.c assoofs.h
10
11 clean:
12     make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
13     rm mkassoofs
```

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

1. Iniciar una sesión como el usuario root:

```
1 $ sudo su
```

2. Compilar:

```
1 # make
2 make -C /lib/modules/6.11.0-17-generic/build M=/home/osboxes/ASO/assoofs_base modules
3 make[1]: Entering directory '/usr/src/linux-headers-6.11.0-17-generic'
4 warning: the compiler differs from the one used to build the kernel
5   The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
6   You are using:          gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
7   MODPOST /home/osboxes/ASO/assoofs_base/Module.symvers
8   CC [M]  /home/osboxes/ASO/assoofs_base/assoofs.mod.o
9   LD [M]  /home/osboxes/ASO/assoofs_base/assoofs.ko
10  BTF [M]  /home/osboxes/ASO/assoofs_base/assoofs.ko
11 Skipping BTF generation for /home/osboxes/ASO/assoofs_base/assoofs.ko due to unavailability of vmlinux
12 make[1]: Leaving directory '/usr/src/linux-headers-6.11.0-17-generic'
13 cc      mkassoofs.c  -o mkassoofs
```

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

```
1 # dd bs=4096 count=100 if=/dev/zero of=image
2 100+0 records in
3 100+0 records out
4 409600 bytes (410 kB, 400 KiB) copied, 0.0153924 s, 26.6 MB/s
```

4. Formatear la imagen creada en el punto anterior en formato *assoofs*:

```
1 # ./mkassoofs image
2 Super block written succesfully.
3 root directory inode written succesfully.
4 welcomefile inode written succesfully.
5 inode store padding bytes (after two inodes) written sucessfully.
6 root directory datablocks (name+inode_no pair for welcomefile) written succesfully.
7 Padding after the rootdirectory children written succesfully.
8 block has been written succesfully.
```

5. Insertar el módulo en el kernel:

```
1 # insmod assoofs.ko
```

6. Crear un punto de montaje:

```
1 # mkdir mnt
```


7. Montamos la imagen creada en el punto de montaje.

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

8. Eliminar el módulo del kernel:

```
1 # rmmmod assoofs
```

9. Comprobar los mensajes del kernel:

```
1 # dmesg
2 ...
3 [ 81.424056] assoofs_init request
4 ...
5 [ 84.699090] assoofs_exit request
```

10. Eliminar los archivos generados durante la compilación y eliminar la imagen creada:

```
1 # make clean
2 # rm image
```

3.1. Compilación estructura básica

Se puede observar cómo después de insertar el módulo `assoofs.ko` en el kernel y posteriormente eliminarlo, la salida que nos proporciona `dmesg` es similar a la siguiente:

```
1 # make
2 # insmod assoofs.ko
3 # rmmmod assoofs
4 # dmesg
5 [ 2424.977652] assoofs_init request
6 [ 2498.313133] assoofs_exit request
```

3.2. Implementadas `assoofs_fill_super` y `assoofs_get_inode_info`

Al ejecutar el comando `ls mnt` no mostrará nada. Y al ejecutar el comando `cat mnt/README.txt` nos dirá que el archivo no existe. Eso no significa que la práctica no funcione, simplemente aún no hemos implementado esas funcionalidades en nuestro sistema de ficheros. Lo que nos interesa es que después de ejecutar la serie de comandos propuestos, la salida de `dmesg` sea igual (o similar a la que se muestra si habéis introducido más trazas en vuestro código).

```
1 # make
2 ...
3 # dd bs=4096 count=100 if=/dev/zero of=image
4 100+0 records in
5 100+0 records out
6 409600 bytes (410 kB, 400 KiB) copied, 0.0128624 s, 31.8 MB/s
7 # ./mkassoofs image
8 Super block written succesfully.
9 root directory inode written succesfully.
10 welcomefile inode written succesfully.
11 inode store padding bytes (after two inodes) written successfully.
12 root directory datablocks (name+inode_no pair for welcomefile) written succesfully.
13 Padding after the rootdirectory children written successfully.
14 block has been written succesfully.
15 # insmod assoofs.ko
16 # mkdir mnt
17 # mount -o loop -t assoofs image mnt
18 # ls mnt/
19 # cat mnt/README.txt
20 cat: mnt/README.txt: No such file or directory
21 # umount mnt/
22 # rmmmod assoofs
23 # dmesg
24 ...
25 [ 5560.058547] assoofs_init request
26 [ 5565.916948] assoofs_mount request
27 [ 5565.916971] assoofs_fill_super request
28 [ 5565.917130] assoofs_get_inode_info
29 [ 5573.156893] Iterate request
30 [ 5583.448024] Lookup request
31 [ 5598.676080] assoofs_exit request
```

3.3. Implementadas assoofs_lookup y assoofs_get_inode

Al ejecutar los comandos "ls mnt" y "cat mnt/README.txt" no va a mostrar nada. Eso no significa que la práctica no funciona, simplemente aún no hemos implementado esas funcionalidades en nuestro sistema de ficheros. Lo que nos interesa es que después de ejecutar la serie de comandos propuestos, la salida de "dmesg" sea igual (o similar si habéis introducido más trazas en vuestro código) a la siguiente:

```
1 # make
2 ...
3 # dd bs=4096 count=100 if=/dev/zero of=image
4 ...
5 # ./mkassoofs image
6 ...
7 # insmod assoofs.ko
8 # mkdir mnt
9 # mount -o loop -t assoofs image mnt
10 # ls mnt
11 # cat mnt/README.txt
12 # umount mnt/
13 # rmmod assoofs
14 # dmesg
15 [ 9420.891980] assoofs_init request
16 [ 9427.653487] assoofs_mount request
17 [ 9427.653506] assoofs_fill_super request
18 [ 9427.653668] assoofs_get_inode_info
19 [ 9429.430502] Iterate request
20 [ 9436.796311] Lookup request
21 [ 9436.796337] assoofs_get_inode
22 [ 9436.796338] assoofs_get_inode_info
23 [ 9436.796362] Read request
24 [ 9447.185458] assoofs_exit request
```

3.4. Implementadas assoofs_create, assoofs_sb_get_a_freeinode, assoofs_sb_get_a_freeblock, assoofs_save_sb_info y assoofs_add_inode_info

Después de ejecutar la serie de comandos propuestos, la salida de "dmesg" debe ser igual (o similar si habéis introducido más trazas en vuestro código) a la siguiente:

```
1 # make
2 ...
3 # dd bs=4096 count=100 if=/dev/zero of=image
4 ...
5 # ./mkassoofs image
6 ...
7 # insmod assoofs.ko
8 # mkdir mnt
9 # mount -o loop -t assoofs image mnt
10 # touch mnt/hola.txt
11 # umount mnt/
12 # rmmod assoofs
13 # dmesg
14 [11930.164393] assoofs_init request
15 [11938.814595] assoofs_mount request
16 [11938.814612] assoofs_fill_super request
17 [11938.814816] assoofs_get_inode_info
18 [11948.560099] Lookup request
19 [ 152.726885] No inode found for the filename {hola.txt}
20 [ 152.726890] assoofs_create New file request
21 [ 152.726894] assoofs_sb_get_a_freeinode request
22 [ 152.726896] Free inode number 2 is free
23 [ 152.726897] assoofs_save_sb_info request
24 [ 152.727218] assoofs_sb_get_a_freeblock request
25 [ 152.727219] Free block number 4 is free
26 [ 152.727220] assoofs_save_sb_info request
27 [ 152.727738] assoofs_add_inode_info request
28 [ 152.728285] assoofs_save_sb_info request
29 [ 152.729353] assoofs_save_inode_info request
30 [ 162.193905] assoofs_exit request
```

3.5. Implementadas assoofs_create y assoofs_mkdir con todas sus funciones auxiliares

Después de ejecutar la serie de comandos propuestos, la salida de "dmesg" debe ser igual a la indicar (o similar si habéis introducido más trazas en vuestro código).

```

1 # make
2 # dd bs=4096 count=100 if=/dev/zero of=image
3 # ./mkassooofs image
4 # insmod assoofs.ko
5 # mkdir mnt
6 # mount -o loop -t assoofs image mnt
7 # touch mnt/hola.txt
8 # mkdir mnt/carpeta
9 # dmesg
10 ...
11 [ 758.430644] assoofs_init request
12 [ 771.378249] assoofs_mount request
13 [ 771.378272] assoofs_fill_super request
14 [ 771.378656] assoofs_get_inode_info
15 [ 782.148085] Iterate request
16 [ 784.759486] Lookup request
17 [ 784.759508] No inode found for the filename {hola.txt}
18 [ 784.759513] assoofs_create New file request
19 [ 784.759517] assoofs_sb_get_a_freeinode request
20 [ 784.759518] Free inode number 2 is free
21 [ 784.759520] assoofs_save_sb_info request
22 [ 784.760179] assoofs_sb_get_a_freeblock request
23 [ 784.760181] Free block number 4 is free
24 [ 784.760182] assoofs_save_sb_info request
25 [ 784.760804] assoofs_add_inode_info request
26 [ 784.761177] assoofs_save_sb_info request
27 [ 784.761624] assoofs_save_inode_info request
28 [ 784.761628] assoofs_search_inode_info request
29 [ 797.455979] Lookup request
30 [ 797.455989] No inode found for the filename {carpeta}
31 [ 797.455994] New directory request
32 [ 797.455997] assoofs_sb_get_a_freeinode request
33 [ 797.455999] Free inode number 3 is free
34 [ 797.456000] assoofs_save_sb_info request
35 [ 797.456541] assoofs_sb_get_a_freeblock request
36 [ 797.456542] Free block number 5 is free
37 [ 797.456544] assoofs_save_sb_info request
38 [ 797.456922] assoofs_add_inode_info request
39 [ 797.457311] assoofs_save_sb_info request
40 [ 797.458015] assoofs_save_inode_info request
41 [ 797.458019] assoofs_search_inode_info request
42 # umount mnt/
43 # mount -o loop -t assoofs image mnt
44 # ls mnt/hola.txt
45 # ls mnt/carpeta
46 # umount mnt/
47 # rmmod assoofs
48 # dmesg
49 [ 856.959894] assoofs_mount request
50 [ 856.959915] assoofs_fill_super request
51 [ 856.960042] assoofs_get_inode_info
52 [ 859.349962] Iterate request
53 [ 863.014577] Lookup request
54 [ 863.014594] assoofs_get_inode
55 [ 863.014595] assoofs_get_inode_info
56 [ 863.014601] Have file: hola.txt, ino=2
57 [ 867.987350] Lookup request
58 [ 867.987361] assoofs_get_inode
59 [ 867.987362] assoofs_get_inode_info
60 [ 867.987366] Have file: carpeta, ino=3
61 [ 867.988004] Iterate request
62 [ 874.358282] assoofs_exit request

```

3.6. Implementadas assoofs_remove y sus funciones auxiliares

```

1 # make
2 # dd bs=4096 count=100 if=/dev/zero of=image
3 # ./mkassooofs image
4 # insmod assoofs.ko
5 # mkdir mnt
6 # mount -o loop -t assoofs image mnt
7 # touch mnt/hola.txt
8 # mkdir mnt/carpeta
9 # rm mnt/hola.txt
10 # rmdir mnt/carpeta
11 # umount mnt/
12 # rmmod assoofs

```

```

13 # dmesg
14 ...
15 [ 1153.962903] assoofs_init request
16 [ 1168.706661] assoofs_mount request
17 [ 1168.706699] assoofs_fill_super request
18 [ 1168.707123] assoofs_get_inode_info
19 [ 1203.643959] Lookup request
20 [ 1203.643964] Lookup in: ino = 0, b=2
21 [ 1203.643974] No inode found for the filename {hola.txt}
22 [ 1203.643979] assoofs_create New file request
23 [ 1203.643983] assoofs_sb_get_a_freeinode request
24 [ 1203.643984] Free inode number 2 is free
25 [ 1203.643985] assoofs_save_sb_info request
26 [ 1203.644226] assoofs_sb_get_a_freeblock request
27 [ 1203.644230] Free block number 4 is free
28 [ 1203.644231] assoofs_save_sb_info request
29 [ 1203.644631] assoofs_add_inode_info request
30 [ 1203.644760] assoofs_save_sb_info request
31 [ 1203.644890] assoofs_save_inode_info request
32 [ 1203.644891] assoofs_search_inode_info request
33 [ 1213.153950] Lookup request
34 [ 1213.153955] Lookup in: ino = 0, b=2
35 [ 1213.153960] No inode found for the filename {carpeta}
36 [ 1213.153965] New directory request
37 [ 1213.153969] assoofs_sb_get_a_freeinode request
38 [ 1213.153970] Free inode number 3 is free
39 [ 1213.153971] assoofs_save_sb_info request
40 [ 1213.154558] assoofs_sb_get_a_freeblock request
41 [ 1213.154559] Free block number 5 is free
42 [ 1213.154560] assoofs_save_sb_info request
43 [ 1213.154992] assoofs_add_inode_info request
44 [ 1213.155339] assoofs_save_sb_info request
45 [ 1213.155894] assoofs_save_inode_info request
46 [ 1213.155898] assoofs_search_inode_info request
47 [ 1218.733916] Iterate request
48 [ 1219.416350] Iterate request
49 [ 1221.825153] assoofs_remove
50 [ 1221.825162] Found dir_record_entry to remove: hola.txt
51 [ 1221.825660] assoofs_sb_set_a_freeinode request
52 [ 1221.825664] El inodo que eliminamos es: 2
53 [ 1221.825666] assoofs_save_sb_info request
54 [ 1221.825801] El bloque eliminamos es: 4
55 [ 1221.825803] assoofs_save_sb_info request
56 [ 1228.528541] assoofs_remove
57 [ 1228.528547] Found dir_record_entry to remove: carpeta
58 [ 1228.529190] assoofs_sb_set_a_freeinode request
59 [ 1228.529207] El inodo que eliminamos es: 3
60 [ 1228.529209] assoofs_save_sb_info request
61 [ 1228.529680] El bloque eliminamos es: 5
62 [ 1228.529684] assoofs_save_sb_info request
63 [ 1234.265366] assoofs_exit request

```

3.7. Parte básica

Para comprobar que el sistema de ficheros se comporta como esperamos, puedes seguir los siguientes pasos:

```

1 # make
2 # dd bs=4096 count=100 if=/dev/zero of=image
3 # ./mkassooofs image
4 # insmod assoofs.ko
5 # mkdir mnt
6 # mount -o loop -t assoofs image mnt
7 # cd mnt/
8 # ls
9 README.txt
10 # cat README.txt
11 Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.
12 # cp README.txt README.txt.bak
13 # ls
14 README.txt README.txt.bak
15 # cat README.txt.bak
16 Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.
17 # mkdir tmp
18 # ls
19 README.txt README.txt.bak tmp
20 # cp README.txt tmp/HOLA
21 # cat tmp/HOLA
22 Hola mundo, os saludo desde un sistema de ficheros ASSOOFS.

```

```
23 # cd ..
24 # umount mnt/
25 # mount -o loop -t assoofs image mnt/
26 # cd mnt/
27 # ls -l
28 total 0
29 ----- 1 root root 0 Feb 24 07:38 README.txt
30 ----- 1 root root 0 Feb 24 07:38 README.txt.bak
31 drwxr-xr-x 1 root root 0 Feb 24 07:38 tmp
32 # rm tmp/HOLA
33 # rmdir tmp/
34 # cd ..
35 # umount mnt/
36 # mount -o loop -t assoofs image mnt
37 # cd mnt/
38 # ls -l
39 total 0
40 ----- 1 root root 0 Feb 24 07:39 README.txt
41 ----- 1 root root 0 Feb 24 07:39 README.txt.bak
42 # cd ..
43 # umount mnt/
44 # rmdir assoofs
```

A. Operaciones binarias sobre free_blocks

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

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <limits.h>
4 #include <stdint.h>
5 #define pbit(v, ds)  !((v) & 1 << (ds))
6
7 void binary(int v) {
8     int i = 32;
9
10    while(i-->0) putchar(pbit(v, i) + '0');
11 }
12
13 int main() {
14     int i;
15
16     uint64_t bloques = (15); // Valor 15 (1111)
17     printf("Bloques ---> valor 15 (1111) \n en Binario = ");
18     binary(bloques);
19     printf("\n");
20
21     printf("Complemento a 1 de Bloques ---> ~(Bloques) \n en Binario =");
22     binary(~(bloques));
23     printf("\n");
24
25     for (i=2; i<32; i++) {
26         printf("Esta libre el bloque %d?\n", i);
27
28         printf("      ~(Bloques): ");
29         binary(~(bloques));
30         printf("\n");
31         printf("      1 << %2d: ", i);
32         binary(1 << i);
33         printf("\n");
34         printf("      ~(Bloques) & 1 << %2d: ", i);
35         binary(~(bloques) & 1 << i);
36         if ((~(bloques) & 1 << i) > 0){
37             printf(" -> El bloque %d SI esta libre\n\n", i);
38             printf("Modificamos Bloques para que el bloque %d pase a estar ocupado\n", i);
39             printf("      Bloques: ");
40             binary(bloques);
41             printf("\n");
42             printf("      (1 << %2d): ", i);
43             binary(1 << i);
44             printf("\n");
45             printf("Bloques | (1 << %2d): ", i);
46             bloques |= (1 << i);
47             binary(bloques);
48             printf("\n\n");
49         }else{
50             printf(" -> El bloque %d NO esta libre\n\n", i);
51         }
52     }
53
54     return 0;
55 }
```

B. Caché de inodos

La función `new_inode` vista en la página 13 (*) 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:

```
1 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`:

```
1 struct assoofs_inode_info *inode_info;
2 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`.

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

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

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

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

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

```
1 kmem_cache_destroy(assoofs_inode_cache);
```

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

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

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

```
1 static const struct super_operations assoofs_sops = {
2     .drop_inode    = generic_delete_inode,
3 };
```

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:

```
1 int assoofs_destroy_inode(struct inode *inode) {
2     struct assoofs_inode *inode_info = inode->i_private;
3     printk(KERN_INFO "Freeing private data of inode %p (%lu)\n", inode_info, inode->i_ino);
4     kmem_cache_free(assoofs_inode_cache, inode_info);
5     return 0;
6 }
```

C. Uso de semáforos para bloquear recursos compartidos

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

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

Para bloquear el mutex usaremos la función `mutex_lock_interruptible`:

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

Para desbloquearlo, usaremos `mutex_unlock`:

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

D. Mover archivos

Para implementar la funcionalidad de mover ficheros y carpetas únicamente es necesario crear una función llamada `assoofs_move`. Esta función a su vez llamará a funciones que ya tenemos definidas dentro de nuestro módulo. Para que esta función sea invocada, en la estructura `inode_operations` será necesario indicar una nueva operación que puede realizarse sobre los inodos.