Práctica 2

Sistema de ficheros

2.1. Objetivos

Comprender las llamadas al sistema y funciones en GNU/Linux para manejo de ficheros y directorios. Entender cómo se realiza la gestión de un sistema de ficheros.

2.2. Introducción

Esta práctica se centra en el Sistema de Ficheros (SF), su estructura básica en Linux así como la creación de un sistema de ficheros propio, implementando un conjunto de funciones que doten a este sistema de una funcionalidad mínima. Se usarán fundamentalmente llamadas al sistema (consultar man 2 syscalls, por ejemplo) y funciones de biblioteca (consultar man 3 strcmp, por ejemplo).

Para ello usaremos la biblioteca FUSE[1] (Filesystem in Userspace) que nos proporciona una API sencilla que nos permite implementar sistemas de ficheros en espacio de usuario y sin

[1] `http://libfuse.github.io/doxygen`

la necesidad de privilegios especiales (simplemente la inclusión del usuario en el grupo del sistema `fuse`)

2.2.1. Recuerda: gestión de errores

El chequeo y la gestión de errores es tema de gran importancia en cualquier circunstancia y también por lo que se refiere al uso de llamadas al sistema. En este caso el error se manifiesta mediante un determinado valor de retorno al efectuar la llamada, típicamente el valor entero `-1`, y se detalla mediante una variable especial, `errno` (algunas funciones de biblioteca recurren a un mecanismo semejante). Esta variable está definida en `<errno.h>` como:

```
extern int errno;
```

Su valor sólo es válido inmediatamente después de que una llamada dé error (devuelva -1) ya que es legal que la variable sea modificada durante la ejecución con éxito de una llamada al sistema. Conviene aclarar además que en programas de un solo thread, `errno` es una variable global; en caso de un proceso multithread se emplea un `errno` local por cada thread, para evitar problemas de coherencia.

El valor de `errno` puede ser leído o escrito directamente; se corresponde con una descripción textual de un error específico que está documentado en `<asm/errno.h>`. La biblioteca de C proporciona una serie de funciones útiles para traducir el valor de `errno` por su representación textual. Las principales funciones son:

```
#include <stdio.h>
void perror(const char *str); // Muestra el texto asociado a errno
                              // junto con la cadena indicada como
                              // argumento
#include <string.h>
char *strerror(int errnum);   // Devuelve el texto explicativo del
                              // error errnum
```

2.3. Manejo de ficheros

El manejo de ficheros en Linux se hace típicamente de dos formas: (1) mediante llamadas al sistema conocido como acceso de bajo nivel, y (2) mediante funciones de la biblioteca estándar de entrada/salida (Standard I/O). En esta práctica vamos a manejar ficheros usando llamadas de bajo nivel. Será de especial ayuda consultar las páginas de manual de `stat` (2), `fstat`, `lseek`, `read` (2) y `write` (2).

2.3.1. Acceso de bajo nivel a ficheros

El acceso de bajo nivel emplea las siguientes llamadas representativas (consultar `man 2 nombre_llamada` para más información):

- Creación, eliminación y cambio de nombre: `creat()`, `unlink()`, `rename()`
- Apertura y cierre: `open()`, `close()`
- Lectura y escritura: `read()`, `write()`
- Miscelánea: `dup()`, `dup2()`, `lseek()`, `umask()`, `truncate()`, `ftruncate()`, `fcntl()`, `ioctl()`, `fsync()`, etc.

Estas llamadas se caracterizan por representar internamente el fichero de trabajo mediante un descriptor de fichero (tipo `int`) y considerar un fichero como una secuencia de bytes cuyo índice es su posición en el fichero. Un fichero abierto está caracterizado por: su descriptor, su posición actual en la secuencia de bytes y el tipo de apertura vigente.

Habitualmente existen 3 ficheros predeterminados abiertos y se suelen corresponder con la entrada y salida por el terminal asignado a la sesión:

- Entrada estándar `STDIN_FILENO` (fd=0; acceso de sólo-lectura),
- Salida estándar `STDOUT_FILENO` (fd=1, acceso de sólo-escritura)
- Salida de error `STDERR_FILENO` (fd=2, acceso de sólo-escritura).

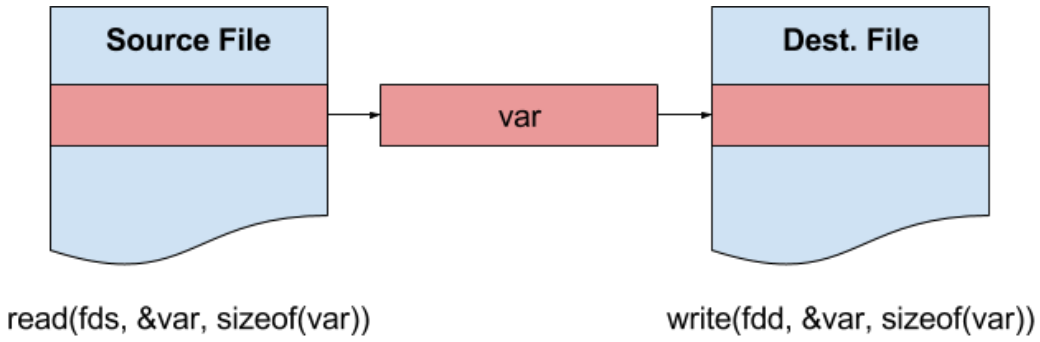


Figura 2.1: Copia de ficheros usando llamadas al sistema.

Ejemplo 1: Implementar un programa que copie el contenido de un fichero en otro (ver Figura 2.1).

Sintaxis: `copy source_file dest_file [BLOCKSIZ]`

Operación: El programa copia el contenido del fichero `source_file` sobre el fichero `dest_file`. La copia se realiza mediante transferencias de `BLOCKSIZ` bytes que opcionalmente puede ser especificada como argumento.

Ejercicio 1: Comparar el tiempo de ejecución de la copia de un fichero de gran tamaño (> 1MB) para diferentes valores de `BLOCKSIZ`. Explicar los valores obtenidos. Para ello ejecutar el programa con una orden como la siguiente:

```
time ./copy example1.pdf example1_1024.pdf 1024
```

El comando `time` muestra en pantalla el tiempo de ejecución del programa que se le pasa como argumento desglosado en tres valores: tiempo de CPU en modo usuario, tiempo de CPU en modo sistema y tiempo real transcurrido.

2.3.2. La biblioteca de E/S estándar (`stdio`)

La biblioteca de E/S estándar proporciona una interfaz sencilla para controlar la E/S en C. El descriptor de fichero para las funciones `stdio` no es un `int` sino un tipo (`FILE*`) donde `FILE` es una estructura de datos que encapsula, entre otros parámetros, la información de bajo nivel de acceso (descriptor entero, tipo de apertura, ...).

Correspondiendo a los descriptores estándar de bajo nivel, disponemos habitualmente de tres descriptores estándar `stdio` que son:

- Entrada estándar **`stdin`** (relacionado con el descriptor `0`),
- Salida estándar **`stdout`** (relacionado con el descriptor `1`)
- Salida error **`stderr`** (relacionado con el descriptor `2`).

Algunas funciones `stdio` habituales son, por ejemplo, **fopen()**, **fclose()**, **fread()**, **fscanf()**, **fwrite()**, **fprintf()**, **fseek()**, etc.

2.4. Administración de ficheros

Los atributos de cada fichero en UNIX están contenidos en un *nodo-i*. El contenido de los *nodos-i* puede ser consultado, al menos parcialmente, con las llamadas `stat()`, `fstat()` o `lstat()`. La información obtenida es la que se describe en la estructura `struct stat` declarada en `<sys/stat.h>` (consultar `man 2 stat`).

```
struct stat {
  dev_t    st_dev;           /* dispositivo ('major' y 'minor')
                              * del sistema de ficheros */
  ino_t    st_ino;           /* número de nodoi */
  mode_t   st_mode;          /* tipo de fichero y permisos */
  link_t   st_nlink;         /* número de enlaces rígidos */
  uid_t    st_uid;           /* ID del usuario propietario */
  gid_t    st_gid;           /* ID del grupo propietario */
  dev_t    st_rdev;          /* dispositivo ('major' y 'minor'
                              * si el fichero es "especial") */
  off_t    st_size;          /* tamaño total, en bytes */
  unsigned long st_blksize; /* tamaño de bloque para la E/S
                              * del sistema de ficheros */
  unsigned long st_blocks;  /* número de bloques asignados */
  time_t   st_atime;         /* fecha de último acceso */
  time_t   st_mtime;         /* fecha de última modificación */
  time_t   st_ctime;         /* fecha de último cambio de estado */
};
```

Algunas llamadas que permiten cambiar o consultar algunos de los atributos de un fichero son:

- Consultar la posibilidad de acceder a un fichero (examinar los permisos): `access()`
- Modificación de los permisos y propietarios (usuario y grupo): `chmod()`, `fchmod()`, `chown()`, `fchown()`, `lchown()`
- Modificación de las fechas asociadas de acceso y modificación: `utime()`

Ejemplo 2: Emular el comportamiento de la orden `stat` de Linux que muestra los atributos de un fichero.

Sintaxis: `status file1 file2 ...`

Operación: Este programa consulta con la llamada `stat` los atributos del *nodo-i* correspondiente a cada uno de los ficheros especificados como argumento y muestra por la salida estándar una relación de todos los atributos que la llamada proporciona.

Ejercicio 2: En el ejemplo 2, si el fichero es un enlace simbólico, los atributos mostrados se refieren al *nodo-i* del fichero al que apunta el enlace. Incorporar una opción (`-L`) al programa anterior para que, cuando se especifique, haga que la consulta de atributos para un enlace simbólico se refiera al fichero apuntado, y cuando no se dé tal opción, se refiera a los atributos del propio enlace.

2.5. Manejo de directorios

En UNIX se distinguen 7 tipos de ficheros: ordinarios, directorios, FIFOS, dispositivos de caracteres, dispositivos de bloques, enlaces simbólicos y sockets.

La creación de cada uno de ellos se realiza empleando valores específicos en los argumentos de ciertas funciones:

- Ficheros ordinarios (altas y bajas en directorios): `mknod()`, `open()`, `link()`
- Enlaces simbólicos: `symlink()`, `readlink()`
- Dispositivos: `mknod()`
- FIFOS: `mknod()`
- Sockets: `socket()`

Los directorios son ficheros cuya creación, eliminación y acceso se efectúa con operaciones particulares, diferentes a las de los ficheros ordinarios: `mkdir()`, `rmdir()`, `chdir()`, `getcwd()`

Las funciones que se aplican para actuar sobre directorios (abrir, cerrar y recorrer) están definidas en `<dirent.h>` y básicamente son: `opendir()`, `closedir()`, `readdir()`, `seekdir()`, `telldir()` y `rewinddir()`.

Estas funciones hacen uso de los siguientes tipos:

- **DIR** (descriptor de directorio): tiene una definición opaca al usuario
- **struct dirent** (campos relevantes de cada entrada individual en un directorio):

```
struct dirent {
  ino_t     d_ino;     /* Número de nodo-i */
  off_t     d_off;     /* Desplazamiento */
  ushort    d_reclen;  /* Longitud del registro */
  char*     d_name;    /* Nombre de fichero */
};
```

Ejemplo 3: Calcular, dentro de un subconjunto de la jerarquía de ficheros, la distribución de ficheros según tamaños y la distribución de directorios según número de entradas.

Sintaxis: `distribution [-t|-n] directory`

Operación: Este programa recorre los ficheros y directorios del esquema jerárquico de una instalación UNIX tomando como punto de partida o raíz el directorio especificado como argumento, y va generando los datos de sendos histogramas de número de ficheros según tamaños (opción `-t`) y/o de número de directorios según entradas que contienen (opción `-n`). Si no especifica ninguna opción se aplica la opción `-t` por defecto. Para ello, el programa lee recursivamente las entradas de los directorios que se encuentra.

Ejercicio 3: Mostrar los resultados de aplicar el programa con ambas opciones al directorio `$HOME` del usuario.

2.6. Tiempos

Por lo general, las dos principales medidas de tiempo son:

Fecha o tiempo real

Tiempo transcurrido desde el inicio de los tiempos o *epoch*, que la mayor parte de los UNIX sitúan en el 1 de Enero de 1970 a las 00:00:00 GMT. Se obtiene mediante las llamadas `time()` y `gettimeofday()` y se puede dar formato mediante las funciones: `localtime()`, `gmtime()`, `asctime()`, `ctime()`, `mktime()`, `strftime()`, `strptime()`.

```
// SYNOPSIS
#include <time.h>
time_t time(time_t *tp);
struct tm *localtime(const time_t *tp);
size_t strftime(char *s, size_t max, const char *fmt; const struct tm *tmp)
```

```
// SYNOPSIS
#include <sys/time.h>
#include <unistd.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv,const struct timezone *tz);
```

El parámetro `timezone` no debe utilizarse (puntero a `NULL`).

Tiempo de CPU

Tiempo consumido por la ejecución de un programa (usuario y/o sistema). Se obtiene mediante las llamadas `times()` y `getrusage()`, o mediante la función `clock()`.

```
// SYNOPSIS
#include <systime.h>
clock_t times(struct tms *buf);
```

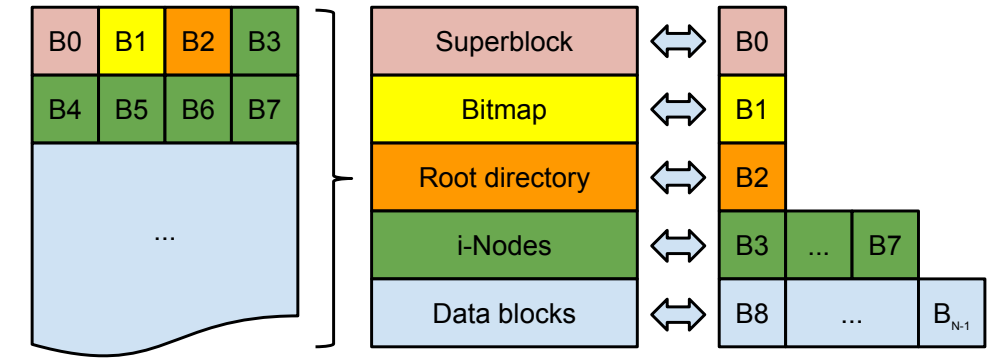
```
// SYNOPSIS
#include <time.h>
clock_t clock(void);

// SYNOPSIS
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
int getrusage (int who, struct rusage *usage);
```

Otros comandos

Otros comandos y funciones relacionados con la gestión de tiempos son: `date`, `sleep()`, `usleep()`, `nanosleep()`, `sleep`, `getitimer()`, `setitimer()`

2.7. Desarrollo de la práctica

En esta práctica implementaremos un mini-sistema de ficheros tipo UNIX. Normalmente un Sistema de Ficheros (SF) se crea al formatear una partición, sin embargo para nosotros resultará más sencillo usar un fichero regular de tamaño fijo para emular una partición y darle el formato deseado. Para ello dividiremos el fichero en **N** bloques de tamaño predefinido (clusters de por ejemplo 4 KiB). Cada bloque del fichero estará destinado a almacenar cierta información (metainformación o datos). Una vez formateado el fichero de acuerdo a nuestro SF propio, almacenaremos, modificaremos y eliminaremos archivos de este SF. La correspondencia entre el SF y su archivo regular de nuestro SO se ilustra conceptualmente en la siguiente figura:



Como se puede apreciar en la Figura, el SF viene definido por cinco partes bien diferenciadas:

- **Super-bloque**: almacena información genérica del SF.
- **Mapa de bits**: indica qué bloques están libres u ocupados.
- **Directorio raíz**: listado de ficheros del directorio raíz.

- **nodos-i**: estructura de tamaño fijo con los nodos-i.
- **Datos**: datos vinculados a los nodos-i

Una posible estructura C para gestionar todo esto podría ser como sigue:

```
#define BIT unsigned
#define BLOCK_SIZE_BYTES 4096
#define NUM_BITS (BLOCK_SIZE_BYTES/sizeof(BIT))
typedef struct MyFileSystemStructure {
        int fdVirtualDisk;              // File descriptor associated with the file
                                        // where the FS is stored
        SuperBlockStruct superBlock;    // Superblock
        BIT bitMap[NUM_BITS];           // Bitmap
        DirectoryStruct directory;      // Root directory
        NodeStruct* nodes[MAX_NODES];   // Array of pointers to inodes
        int numFreeNodes;               // # of available inodes
} MyFileSystem;
```

donde:

- *fdVirtualDisk*: es el identificador del fichero regular abierto que almacenará el SF en el disco duro.
- *superBlock*: es una estructura que se almacena en el bloque 0 del archivo, y contiene información global del SF: número total de bloques de datos libres, tamaño total del sistema de ficheros, etc. Una posible estructura para almacenar esta información podría ser la siguiente:

```
typedef struct SuperBlockStructure {
        time_t creationTime;    // Creation time
        int diskSizeInBlocks;   // # blocks in disk
        int numOfFreeBlocks;    // # of available blocks
        int blockSize;          // Block size
        int maxLenFileName;     // Max. length of a file name
        int maxBlocksPerFile;   // Max. number of blocks per file
} SuperBlockStruct;
```

- *bitMap*: es un array de 0-1. Se almacena en el segundo bloque del archivo de respaldo y contiene información acerca de los bloques libres (0) y ocupados (1) del sistema.
- Tendremos un único directorio. La estructura directorio almacena el número de archivos que contiene y la información relativa a cada archivo que es: el nodo-i al que está vinculado, el nombre del archivo, y una variable lógica que indica si el archivo está libre o no. Definiremos una longitud máxima para el nombre del archivo de 15 caracteres, y como mucho, que un directorio pueda contener hasta 100 archivos. Toda la información el directorio se almacena en el tercer bloque del archivo. La estructuras que definen el directorio podrían ser las siguientes:

```
#define MAX_FILES_PER_DIRECTORY 100
#define MAX_LEN_FILE_NAME 15
typedef struct DirectoryStructure {
        int numFiles;                           // Number of directory entries
        FileStruct files[MAX_FILES_PER_DIRECTORY];      // Directory entries
} DirectoryStruct;

typedef struct FileStructure {
        int  nodeIdx;                       // Associated inode
        char fileName[MAX_LEN_FILE_NAME + 1]; // File name
        BOOLEAN freeFile;                   // Free directory entry
} FileStruct;
```

- *nodes*: son los nodos-i del SF. Cada nodo-i almacena información tal como el número de bloques que ocupa el archivo asociado, el tamaño del archivo asociado, la fecha/hora en la que fue creado o modificado, los índices de los bloques del SF donde están los datos del archivo asociado, y si es un nodo-i libre o no. Todos los nodos-i se guardarán en 5 bloques del SF. No obstante, aún nos quedan algunas constantes por definir, como `MAX_NODES` que ya pueden definirse. Estas definiciones y la estructura que define el nodo-i podrían ser como sigue:

```
#define MAX_BLOCKS_PER_FILE 100
typedef struct NodeStructure {
        int numBlocks;                          // Num blocks
        int fileSize;                           // File size
        time_t modificationTime;                // Modification time
        DISK_LBA blocks[MAX_BLOCKS_PER_FILE];   // Blocks
        BOOLEAN freeNode;                       // If the node is available
} NodeStruct;
#define NODES_PER_BLOCK (BLOCK_SIZE_BYTES/sizeof(NodeStruct))
#define MAX_NODES (NODES_PER_BLOCK * MAX_BLOCKS_WITH_NODES)
```

- *numFreeNodes* almacena la cantidad de nodos-i libres que quedan en el SF.

Con la especificación anterior, podemos resumir las simplificaciones que se llevan a cabo:

- Nuestro sistema de ficheros no necesita soporte para directorios multinivel. Hay un solo directorio en nuestro sistema. Este directorio contiene como mucho 100 archivos.
- Cada archivo contiene a lo sumo 100 bloques de datos. Por lo tanto, el tamaño de cada archivo es menor que 100 tam. bloque. Podemos definir el tamaño de bloque por nuestra cuenta (4 KiB en el código anterior).
- Los nodos-i en nuestro sistema contienen a lo sumo 100 punteros directos a bloques de datos. No se requieren punteros indirectos.

2.8. Creación del SF

Se proporciona, como esqueleto de la práctica, una función `main` que parsea los parámetros de entrada, inicializa el sistema de ficheros empleando la función `myMkfs`, llama a la función `fuse_main` y finaliza liberando la memoria reservada a través de `myFree`.

2.8.1. `myMkfs`

```
int myMkfs(MyFileSystem *myFileSystem, int diskSize, char *backupFileName);
```

Esta función crea un SF de `diskSize` bytes y almacena su contenido en el archivo `backupFileName`. Además, esta función inicializa todas las estructuras de datos mencionadas anteriormente. Se proporciona ya implementada en `myFS.c`.

2.8.2. `fuse_main`

```
int fuse_main(int argc,char* argv[],struct fuse_operations* op,void* user_data )
```

Esta función se encarga de montar el SF usando la librería FUSE y recibe los siguientes parámetros:

- `argc`: número de argumentos para el montaje.

- `argv`: argumentos de montaje. Los más relevantes para nosotros son:
  - `-f`: trabajar en primer plano
  - `-d`: habilitar salida de depuración de FUSE (implica -f)
  - `-s`: deshabilita multi-hilo (facilita depuración)
  - `directorio`: punto de montaje de FUSE
- `op`: estructura con punteros a las operaciones soportadas por nuestro SF. En nuestro caso están implementadas las siguientes operaciones:

```
struct fuse_operations myFS_operations = {
        .getattr        = my_getattr,    // Retrieve attributes from a file
        .readdir        = my_readdir,    // Read directory entries
        .truncate       = my_truncate,   // Modify the size of a file
        .open           = my_open,       // Open a file
        .write          = my_write,      // Write data into a file already open
        .release        = my_release,    // Close an open file
        .mknod          = my_mknod,      // Create a new file
};
```

  Se puede consultar la documentación y el prototipo de las operaciones soportadas en el manual de FUSE apartado `fuse_operations Struct Reference`[2].
- `user_data`: argumentos extra durante la inicialización (no implementado).

La función `fuse_main` se mantiene a la espera de llamadas a nuestro SF e invoca a las operaciones registradas. Podemos finalizar la función con `Control+C`.

2.8.3. Ejemplo

Con el siguiente comando creamos un sistema de ficheros propio de tamaño 2097152 dentro de un fichero regular llamado `virtual-disk` y le decimos a FUSE que lo monte en el directorio `mount-point`, que deberá existir y estar vacío:

```
$ ./fs-fuse -t 2097152 -a virtual-disk -f '-d -s mount-point'
```

FUSE montará el sistema de ficheros en el directorio `mount-point`, que debe existir y estar vacío. Si todo ha ido bien, obtendremos la siguiente salida:

```
SF: virtual-disk, 2097152 B (4096 B/block), 512 blocks
1 block for SUPERBLOCK (32 B)
1 block for BITMAP, covering 1024 blocks, 4194304 B
1 block for DIRECTORY (2404 B)
5 blocks for inodes (424 B/inode, 45 inodes)
504 blocks for data (2064384 B)
Formatting completed!
File system available
FUSE library version: 2.9.0
nullpath_ok: 0
nopath: 0
utime_omit_ok: 0
unique: 1, opcode: INIT (26), nodeid: 0, insize: 56, pid: 0
INIT: 7.22
flags=0x0000f7fb
max_readahead=0x00020000
```

[2] http://libfuse.github.io/doxygen/structfuse__operations.html

```
   INIT: 7.18
   flags=0x00000011
   max_readahead=0x00020000
   max_write=0x00020000
   max_background=0
   congestion_threshold=0
   unique: 1, success, outsize: 40
```

Se puede comprobar cómo se ha formateado el disco virtual reservando 1 bloque para el superbloque, otro para el mapa de bits, otro para el directorio raíz, 5 para los nodos-i y el resto, 504, para datos. Seguidamente se monta el sistema de ficheros y, dependiendo del sistema operativo, comenzarán a aparecer mensajes de depuración:

```
unique: 2, opcode: LOOKUP (1), nodeid: 1, insize: 47, pid: 4535
LOOKUP /.Trash
getattr /.Trash
--->>>my_getattr: path /.Trash
   unique: 2, error: -2 (No such file or directory), outsize: 16
unique: 3, opcode: LOOKUP (1), nodeid: 1, insize: 52, pid: 4535
LOOKUP /.Trash-1000
getattr /.Trash-1000
--->>>my_getattr: path /.Trash-1000
   unique: 3, error: -2 (No such file or directory), outsize: 16
```

Cada llamada al SF comienza con unique más un número que va incrementado, en el ejemplo anterior se vemos cómo al montar la unidad se consultan los atributos de .Trash y .Trash-1000 (directorios usados como papelera de reciclaje) y cómo la respuesta del SF es "No such file or directory" para ambos, ya que no existen. Si ahora hacemos un ls del punto de montaje en otro terminal obtenemos la siguiente salida:

```
usuarioso@FUSE_myFS$ ls -la mount-point
total 2
drwxr-xr-x 2 usuarioso usuarioso   0 oct 23 14:42 .
drwxr-xr-x 1 usuarioso usuarioso 4096 oct 23 14:42 ..
```

Mientras que FUSE nos mostrará las llamadas que se han realizado:

```
unique: 4, opcode: GETATTR (3), nodeid: 1, insize: 56
getattr /
--->>>my_getattr: path /
   unique: 4, success, outsize: 120
unique: 5, opcode: GETXATTR (22), nodeid: 1, insize: 65
   unique: 5, error: -38 (Function not implemented), outsize: 16
unique: 6, opcode: OPENDIR (27), nodeid: 1, insize: 48
   unique: 6, success, outsize: 32
unique: 7, opcode: READDIR (28), nodeid: 1, insize: 80
readdir[0] from 0
--->>>my_readdir: path /, offset 0
   unique: 7, success, outsize: 80
unique: 8, opcode: GETATTR (3), nodeid: 1, insize: 56
getattr /
--->>>my_getattr: path /
   unique: 8, success, outsize: 120
unique: 9, opcode: READDIR (28), nodeid: 1, insize: 80
   unique: 9, success, outsize: 16
unique: 10, opcode: RELEASEDIR (29), nodeid: 1, insize: 64
   unique: 10, success, outsize: 16
```

Se han obtenido los atributos del directorio raíz y se ha leído su contenido. Si ejecutamos el programa `test1.sh` se crearán 2 ficheros dentro de nuestro SF `file1.txt` con contenido

"`file 1`" y `file2.txt` con el texto "`this is file 2`", de manera que al listar el contenido del directorio obtendremos:

```
usuarioso@FUSE_myFS$ ls -la mount-point
total 2
drwxr-xr-x 2 usuarioso usuarioso 0 oct 23 14:42 .
drwxr-xr-x 1 usuarioso usuarioso 4096 oct 23 14:55 ..
-rw-r--r-- 1 usuarioso usuarioso   10 oct 23 14:55 file1.txt
-rw-r--r-- 1 usuarioso usuarioso   21 oct 23 14:55 file2.txt
```

Junto con el esqueleto de la práctica se proporciona un ejecutable, `my-fsck`, encargado de chequear la consistencia del sistema de ficheros (semejante al comando `chkdsk` de Windows o `fsck` en linux). Este auditor calcula de dos formas distintas el espacio libre del SF, el número de ficheros presentes y los bloque ocupados aprovechando la redundancia de información y presenta un informe detallado:

```
usuarioso@FUSE_myFS$ ./my-fsck virtual-disk
*************************** SUPERBLOCK *****
Creation time: Sun Jul 12 12:13:47 2015
Super block size: 32
Directory size  2404
Node size 424
Disk size (blocks) 512
Num. of free blocks 502
Block size (bytes) 4096
Max. Length file name 15
Max. blocks/file 100
*****************************

*************************CHECKING FREE SPACE******
Free space consistent:Free blocks: 502 (in bitmap) . 502 (in super-block)
*****************************

*************************Number of files ******
inodes occupied: 2. Free 43. Free WITHOUT FREE: 0
Files using numFiles attribute 2. In directory's file array 2
*****************************

************************* Checking occupied blocks ******
Data blocks occupied. Using bitmap 2,  Using inodes 2
*****************************

************************* File list ******
Name            Siz(block)      Siz(bytes)      Time
file1.txt       4096            7               6/12    12:13
file2.txt       4096            15              6/12    12:13
*****************************
```

Como no está implementada la lectura en nuestro sistema de ficheros, si intentamos leer los datos del fichero 1 obtendremos error:

```
usuarioso@FUSE_myFS$ cat mount-point/file1.txt
cat: mount-point/file1.txt: Función no implementada
```

pero sabiendo que el fichero 1 se creó el primero y que asignamos los bloques de datos por orden y a partir del octavo, podremos buscar el contenido del fichero en el disco virtual de la siguiente manera:

```
usuarioso@FUSE_myFS$ hexdump virtual-disk -C -s 32768 -n 7
00008000  66 69 6c 65 20 31 0a                              |file 1.|
00008007
```

donde `0x0a` corresponde con la secuencia de escape `\n`.

Ejercicio 4: ¿Dónde están y cómo podemos obtener los datos correspondientes al fichero 2 usando `hexdump`? consulta el manual de `hexdump` para ello.

2.9. Parte obligatoria

1. Implemente la operación para borrar ficheros e inclúyala en la estructura `fuse_operations`. Consulte el manual de FUSE para ver el prototipo de la función `unlink`[3]. Compruebe que borra adecuadamente con la herramienta `my-fsck`.
2. Implemente la operación para leer datos de los ficheros asociada al miembro `read`[4] de la estructura `fuse_operations`. Dese cuenta de que, según el manual, la función debe retornar tantos bytes como se le solicitan siempre y cuando los haya, ya que en caso contrario serán rellenados con ceros (no se especifica `direct_io` en el montaje de la unidad).
3. Desarrolle un `script` que realice las siguientes operaciones sobre el sistema de ficheros:
   a) Copie dos ficheros de texto que ocupen más de un bloque (por ejemplo fuseLib.c y myFS.h) a nuestro SF y a un directorio temporal, por ejemplo `./temp`
   b) Audite el disco y haga un diff entre los ficheros originales y los copiados en el SF Trunque el primer fichero (man truncate) en copiasTemporales y en nuestro SF de manera que ocupe ocupe un bloque de datos menos.
   c) Audite el disco y haga un `diff` entre el fichero original y el truncado.
   d) Copie un tercer fichero de texto a nuestro SF.
   e) Audite el disco y haga un `diff` entre el fichero original y el copiado en el SF
   f) Trunque el segundo fichero en copiasTemporales y en nuestro SF haciendo que ocupe algún bloque de datos más.
   g) Audite el disco y haga un `diff` entre el fichero original y el truncado.

Nota: si durante las pruebas de depuración finalizamos el programa de manera abrupta, el punto de montaje puede quedar bloqueado por FUSE, para desmontarlo en línea de comando podemos utilizar la siguiente orden:

```
$ fusermount -u mount-point
```

[3] `http://libfuse.github.io/doxygen/structfuse_operations.html#a8bf63301a9d6e94311fa10480993801e`
[4] `http://libfuse.github.io/doxygen/structfuse_operations.html#a2a1c6b4ce1845de56863f8b7939501b5`