Práctica 2

Sistema de ficheros

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.

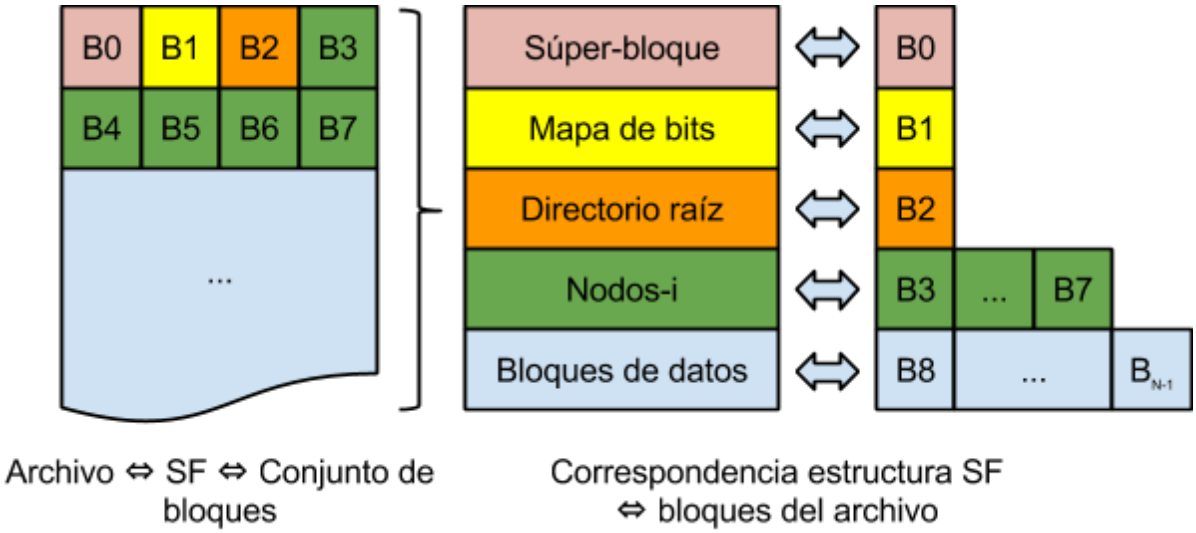
Introducción

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).

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: (1) el super-bloque, que almacena información genérica, (2) el mapa de bits que indica qué bloques están libres u ocupados, (3) la información del directorio raíz, (4) los nodos-i, y finalmente (5) los datos vinculados a los nodos-i. Una posible estructura C para gestionar todo esto podría ser como sigue:

```c
#define BIT unsigned
#define TAM_BLOQUE_BYTES 4096
#define NUM_BITS (TAM_BLOQUE_BYTES/sizeof(BIT))
typedef struct MiSistemaDeFicheros {
  int fdDiscoVirtual;              // Descriptor del fichero que almacena 'miSF'
  EstructuraSuperBloque superBloque;   // Superbloque
  BIT mapaDeBits[NUM_BITS];            // Mapa de bits
  EstructuraDirectorio directorio;     // Directorio raíz
  EstructuraNodoI nodosI[MAX_NODOSI];  // Array de punteros a Nodos-i
  int numNodosLibres;                  // Número de nodos-i libres
} MiSistemaDeFicheros;
```

donde:

- *fdDiscoVirtual* es el identificador del fichero regular abierto que almacenará el SF en el disco duro.
- La estructura *superBloque* 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:

```c
typedef struct EstructuraSuperBloque {
  time_t fechaCreacion;     // Fecha en la que se creó el sistema de ficheros
  int tamDiscoEnBloques;    // Núm. de bloques en disco
  int numBloquesLibres;     // Núm. de bloques libres
  int tamBloque;            // Tamaño de bloque
  int maxTamNombreArchivo;  // Tamaño máx. de nombre de archivo
  int maxBloquesPorArchivo; // Tamaño máx. de bloques por archivo
} EstructuraSuperBloque;
```

- *mapaDeBits* 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_TAM_NOMBRE_ARCHIVO 15
#define MAX_ARCHIVOS_POR_DIRECTORIO 100
typedef struct EstructuraDirectorio {
  int numArchivos;                                        // Núm. archivos
  EstructuraArchivo archivos[MAX_ARCHIVOS_POR_DIRECTORIO]; // Archivos
} EstructuraDirectorio;

typedef struct EstructuraArchivo {
  int  idxNodoI;                                  // Nodo-i asociado
  char nombreArchivo[MAX_TAM_NOMBRE_ARCHIVO+1]; // Nombre archivo
  int libre;                                      // Archivo libre
} EstructuraArchivo;
```

- nodosI 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_NODOS_I` que ya pueden definirse. Estas definiciones y la estructura que define el nodo-i podrían ser como sigue:

```
#define MAX_BLOQUES_POR_ARCHIVO 100
typedef struct EstructuraNodoI {
  int numBloques;                          // Núm. bloques
  int tamArchivo;                          // Tamaño archivo
  time_t tiempoModificado;                 // Tiempo de modificación
  int idxBloques[MAX_BLOQUES_POR_ARCHIVO]; // Bloques
  int libre;                               // Nodo libre
} EstructuraNodoI;
#define NODOSI_POR_BLOQUE (TAM_BLOQUE_BYTES/sizeof(EstructuraNodoI))
#define MAX_NODOSI (NODOSI_POR_BLOQUE * MAX_BLOQUES_CON_NODOSI)
```

- Finalmente, *numNodosLibres* 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 $100 \times tam.\ bloque$. 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.

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`.

myMkfs

```
int myMkfs(MiSistemaDeFicheros* miSistemaDeFicheros, int tamDisco, char*
nombreArchivo)
```

Esta función crea un SF de `tamDisco` bytes y almacena su contenido en el archivo `nombreArchivo`.

Además, esta función inicializa todas las estructuras de datos mencionadas anteriormente. Se proporciona ya implementada en `myFS.c`.

fuse_main

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

Esta función se encargada 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:
  - *-d*: habilitar salida de depuración de FUSE (implica -f)
  - *-f*: trabajar en primer plano
  - *-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     = mi_getattr,   // Obtener atributos de un fichero
       .readdir     = mi_readdir,   // Leer entradas del directorio
       .truncate    = mi_truncate,  // Modificar el tamaño de un fichero
       .open        = mi_open,      // Abrir un fichero
       .write       = mi_write,     // Escribir datos en un fichero abierto
       .release     = mi_release,   // Cerrar un fichero abierto
       .mknod       = mi_mknod,     // Crear un fichero nuevo
};
```

`http://fuse.sourceforge.net/doxygen/structfuse__operations.html`

- **user_data:** argumentos asignados al contexto durante la inicialización de FUSE

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.

Ejemplo

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

```
$ ./sf-fuse -t 2097152 -a disco-virtual -f '-d -s punto-montaje'
```

Si todo ha ido bien, obtendremos la siguiente salida:

```
SF: disco-virtual, 2097152 B (4096 B/bloque), 512 bloques
1 bloque para SUPERBLOQUE (32 B)
1 bloque para MAPA DE BITS, que cubre 1024 bloques, 4194304 B
1 bloque para DIRECTORIO (2404 B)
5 bloques para nodos-i (a 424 B/nodo-i, 45 nodos-i)
504 bloques para datos (2064384 B)
¡Formato completado!
Sistema de ficheros disponible
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: 7.17
flags=0x0000047b
max_readahead=0x00020000
```

```
   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
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
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_miFS$ ls -la punto-montaje
total 2
drwxr-xr-x 2 usuarioso usuarioso    0 oct 23 14:42 .
drwxr-xr-x 1 usuarioso usuarioso 4096 oct 23 14:42 ..
```

Mientas 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 `fichero1.txt` con contenido *"fichero 1"* y `fichero2.txt` con el texto *"este es el fichero 2"*, de manera que al listar el

contenido del directorio obtendremos:

```
usuarioso@FUSE_miFS$ ls -la punto-montaje
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 fichero1.txt
-rw-r--r-- 1 usuarioso usuarioso   21 oct 23 14:55 fichero2.txt
```

Junto con el esqueleto de la práctica se proporciona un ejecutable, `audita`, 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 las 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_miFS$ ./audita disco-virtual
*************************** SUPERBLOQUE *****
Fecha de creación: Sun Oct 26 10:33:17 2014
Tam Super Bloque: 32
Tam Directorio 2404
Tam NodoI 424
Tam Disco (bloques) 512
Num bloques libres 502
Tam Bloque (bytes) 4096
Max. Tam nombre archivo 15
Max bloques por archivo 100
*****************************
*************************CHEQUEO ESPACIO LIBRE******
Espacio libre consistente:Bloques libres: 502 (por mapa de bits) . 502 (por superbloque
*****************************
*************************Recuento de ficheros ******
Nodos i ocupados: 2. Libres 43. Libres pero SIN FREE: 0
Ficheros según campo numArchivos 2. Según array de archivos 2
*****************************
************************* Comprobacion bloques ocupados ******
Bloques de datos ocupados. Por mapa de bits 2, Por nodos-i 2
*****************************
************************* Listado de ficheros ******
Nombre Tam(bloq) Tam(bytes) Fecha
fichero1.txt 4096 10 9/26 10:33
fichero2.txt 4096 21 9/26 10:33
*****************************
```

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

```
usuarioso@FUSE_miFS$ cat punto-montaje/fichero1.txt
cat: punto-montaje/fichero1.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_miFS$ hexdump disco-virtual -C -s 32768 -n 10
00008000  66 69 63 68 65 72 6f 20  31 0a                    |fichero 1.|
```

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

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

Parte obligatoria

1. Implementa primero la operación para borrar ficheros asociada al miembro `unlink` de la estructura `fuse_operations`. Comprueba que borra adecuadamente con la herramienta `AuditaDisco`.

2. Implementa ahora la operación para leer ficheros asociada al miembro `read` de la estructura `fuse_operations`.
3. Desarrolla un script que realice las siguientes operaciones sobre el sistema de ficheros:
    a. Copie dos ficheros de texto que ocupen más de un boque (por ejemplo fuseLib.c y myFS.h) a nuestro SF y a un directorio temporal, por ejemplo `./copiasTemporales`
    b. Audite el disco y haga un `diff` entre los ficheros originales y los copiados en el SF
    c. Trunque el primer fichero (man truncate) en `copiasTemporales` y en nuestro SF de manera que ocupe ocupe un bloque de datos menos.
    d. Audite el disco y haga un `diff` entre el fichero original y el truncado.
    e. Copie un tercer fichero de texto a nuestro SF.
    f. Audite el disco y haga un `diff` entre el fichero original y el copiado en el SF
    g. Trunque el segundo fichero en `copiasTemporales` y en nuestro SF haciendo que ocupe algún bloque de datos más.
    h. 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 punto-montaje
```

Parte extra

Se quiere dar soporte a los enlaces simbólicos:

`ln -s archivo enlace`: Crea un enlace simbólico llamado `enlace` que apunta al fichero `archivo`.

Para poder realizar este apartado se puede, por ejemplo, modificar la estructura `EstructuraNodoI` para que contenga un campo entero llamado `tipo` que identifique el tipo de nodo-i, que podrá ser bien un fichero regular (`tipo=0`) o bien un enlace simbólico (`tipo=1`). Para crear un enlace simbólico se debe implementar la función **`int`** `fuse_operations::sym_link(`**`const char*, const char*`**`);`

Para leer un enlace simbólico se debe completar adecuadamente la función `getattr`, actualizando la estructura `stat` según el campo `tipo` mencionado anteriormente

```
stbuf->st_mode = S_IFLNK | 0644;
```

Además, se debe implementar la función **`int`** `fuse_operations::readlink(`**`const char*, char*`**`, size_t);`

Funciones útiles para el desarrollo de la práctica

Llamadas al sistema

`open, close, read, write, sync, lseek, stat, time`

Funciones de biblioteca

`localtime, strcmp, strlen`