La Salle Campus Barcelona

Sistemes Operativos Avanzados

Ragnarök

Sistemas de Ficheros

Gabriel Cammany Ruiz

Daniel Ortiz Iriarte

ls30652

daniel.ortiz.2015

22 de mayo de 2018

Índice

1. Requisitos

Esta practica tiene como objetivo aumentar los conocimientos de los alumnos en referencia a los sistemas de ficheros, en este caso con `FAT32` y `EXT4`.

Se separa en diversas fases que ayudan al desarrollo del entendimiento de los diferentes sistemas, empezando por una fase sencilla de detección los diferentes tipos que existen, dentro de un rango limitado.

Continuando por la investigación de la estructuración interna de cada *filesystem*, desarrollando el proyecto para la búsqueda recursiva de ficheros de cada sistema y finalmente acabando con modificaciones de permisos y fechas de creación.

El objetivo por lo tanto, es mostrar al alumno como y de que manera se organizan dos de los sistemas de ficheros mas usados a día de hoy, como continuación de la asignatura de Sistemas Operativos.

2. Diseño

El proyecto se divide en dos módulos en los cuales se implementan todas las funcionalidades. Cabe recalcar que la gran mayoría de estas están implementadas en el segundo mientras que el primero carga con menos trabajo. A partir de esto, se establece la siguiente estructuración a nivel de ficheros.

```
Ragnarok
├── include
│   ├── info.h
│   └── search.h
├── src
│   ├── info.c
│   └── search.c
└── ragnarok.c
```

2.1. info.h/info.c

En este módulo es donde se implementa las funcionalidades de la fase 1 así como otras "auxiliares" para el resto de fases. Por ejemplo, se cuenta con las siguientes funciones:

```
int detecta_tipo();
```

Una vez abierto el volumen, esta función dice qué tipo de volumen es. `FAT32`, `EXT4`, `FAT16`, etc.

```
void ext4_info(); void fat32_info();
```

Para un volumen `EXT4` o `FAT32`, muestra la información, como el nombre del *filesystem*, número de inodos libres, última verificación, etc.

```
void ext4_get_structure();  void fat32_get_structure();
```

Estas dos funciones son bastante importantes, ya que permite rellenar la estructura de datos principal donde se encuentra la información del filesystem a utilizar. Es decir, antes de realizar una búsqueda recursiva o mostrar un fichero, se llama siempre a esta función para extraer del *Superblock* y del *Block Directory*, en el caso de `EXT4`, y Volume ID para `FAT32`, los datos necesarios para poder ejecutar la comanda. Como viene a ser, tamaño de bloque, dirección tabla de *inodes*, etc.

Destacar en ambos sistemas cálculos necesarios que nos ayudan en todo el proceso de llamadas recursivas.

FAT32

$$fat32.fat\_location = fat32.bytes\_per\_sector \times fat32.reserved\_sectors$$

$$fat32.first\_cluster = fat32.fat\_location + fat32.sectors\_per\_fat \times fat32.bytes\_per\_sector \times 2$$

Para el primer *cluster*, vamos a necesitar multiplicar por dos el valor del tamaño de este, para saltarnos los dos primeros inválidos.

EXT4

$$ext4.inode.table\_loc = (((uint64\_t)\ (high))\ << 32) \mid low;$$

Siendo *high* y *low* los valores que estan en el *directory entry*.

2.2. search.h/search.c

Por otra parte, este se trata de la parte troncal del proyecto. Lo podríamos dividir en dos subapartados. Para cada uno de estos, tenemos funciones recursivas para la muestra del contenido como para la búsqueda.

2.2.1. EXT4

Primero de todo comentar la estructura principal de `EXT4`, con la cual hemos trabajado para resolver la busqueda de ficheros en este tipo de *filesystem*.

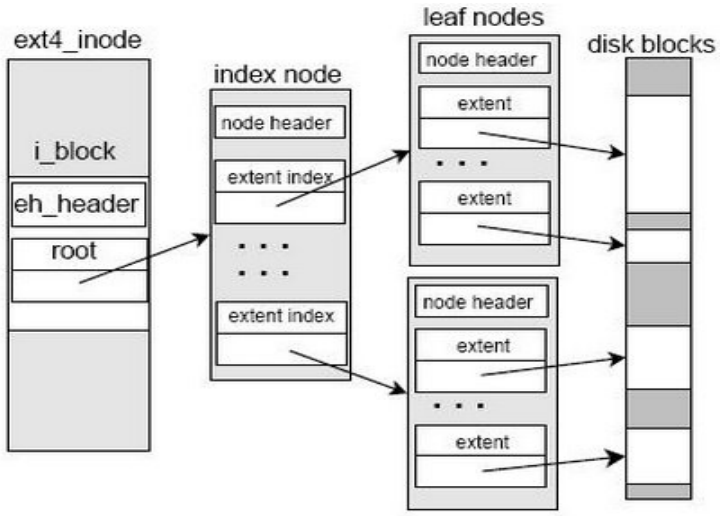

Figura 1: Estructura del *extent tree* de `EXT4`

A partir de la figura anterior, las siguientes funciones se dedican a ejecutar cada uno de los apartados. Con el orden de flechas, de izquierda a derecha ejecutamos las funciones a continuación. Como se ejecuta de manera recursiva, una vez llegadas a la última de `deepsearch_leaf_ext4`, volvemos a la inicial `_ext4`. Recalcar que cada una de estas funciones tienen un *scope* privado, ya que la interfície de este modulo es bastante limitada.

```
uint32_t _ext4(char show, char *name, uint32_t inode);
```

Esta es la principal función recursiva. Se llama cada vez que se quiere consultar un *inode*. Su principal objetivo es extraer el `eh_header` y decidir cual de las dos funciones a continuación tiene que llamar. Dado que también es la primera que se llama, también se utiliza para llamar a las funciones que se encargan de realizar la parte de muestra del contenido de un fichero. Se trataría de la primera columna de la figura 1. Para poder extraer el `eh_header` utiliza el resultado de la siguiente formula y utiliza la función *lseek* para acceder a esta. Restamos menos uno al valor de *inode*, ya que el primero no existe.

$$new\_position = (ext4.inode.table\_loc \times ext4.block.size) + (ext4.inode.size \times (inode - 1)) + 0x28$$

```
uint32_t deepsearch_tree_ext4(char *name, uint16_t eh_entries);
```

Como ya hemos comentado, esta función es llamada dependiendo del valor de `eh_depth` que tiene el `eh_header` que se ha leído anteriormente. Dado que esta se encarga de tratar todos las *entries* como nodos raíz, extraerá la posición del bloque de cada entrada que contiene el siguiente nivel del árbol, moverá el *file descriptor* a la nueva posición y leerá el `eh_header` de nuevo, así recursivamente llamándose a si misma, hasta encontrar el valor de `eh_depth` igual a cero. En este caso, llamará a la siguiente función. A continuación añadimos el calculo realizado para mover el *file descriptor* a la nueva posición de la hoja del árbol, se utiliza tanto para esta función como para la siguiente.

$$block\_direction = ((block\_direction \mid leaf[i].ei\_leaf\_hi) \ << \ 32) \mid leaf[i].ei\_leaf\_lo;$$

```
uint32_t deepsearch_leaf_ext4(char *name, uint16_t eh_entries);
```

Podríamos considerar esta función como la última en el árbol de llamadas recursivas, ya que esta será la encargada de finalizar la ejecución del programa en caso de encontrar el fichero, o devolver el valor de fichero no encontrado y finalizar la búsqueda del fichero en el directorio actual.

Está centrada en trabajar en bloques con la estructura de datos `ext4_dir_entry`. Ya que, en los únicos tipos de *inodes* que vamos a ejecutar esta función es en directorios, para los ficheros simplemente nos vale con comprobar una vez en el directorio actual, que el nombre de este no sea igual al que buscamos. Pero nunca llegamos a visitar el *inode* y su *extent tree* fuera del apartado de muestra de contenido.

Llegados a cada entrada de tipo `ext4_dir_entry` con el `file_type` indicando que es un directorio, volvemos a llamar a la función `_ext4`, comentada anteriormente, con el nuevo *inode* de la entrada actual. Siempre teniendo en cuenta que no entraremos para los casos de directorios "." o "..".

```
void deepshow_tree_ext4(uint16_t eh_entries);
void deepshow_leaf_ext4(uint16_t eh_entries);
```

Para acabar, ambas funciones realizan el mismo trabajo que las dos anteriores. Con la diferencia que la recursividad va entre ambas en vez de tener una tercera función. Para el caso de `deepshow_tree_ext4`, su único trabajo es extraer del `eh_header` el `eh_depth` y decidir si llamarse a si misma o a `deepshow_leaf_ext4`, queriendo significar que ya ha encontrado contenido.

Por otra parte, la función comentada anteriormente se encargará a partir del `ee_len`, ir mostrando por pantalla carácter a carácter hasta que llegue al limite o se encuentre un byte igual a cero. A partir de ese punto finalizará su ejecución y en caso de estar en un árbol con más de un nivel continuará mostrando otras ramas.

2.2.2. FAT32

Una vez explicado el diseño que hemos establecido para `EXT4`, `FAT32` es un sistema con algunas características similares pero con unas estructuras de datos más sencillas. Primero de todo, como ya hemos hecho con el sistema de ficheros anterior, mostramos la vista general de su funcionamiento, para así desarrollar una mejor explicación de las funciones creadas. [1]

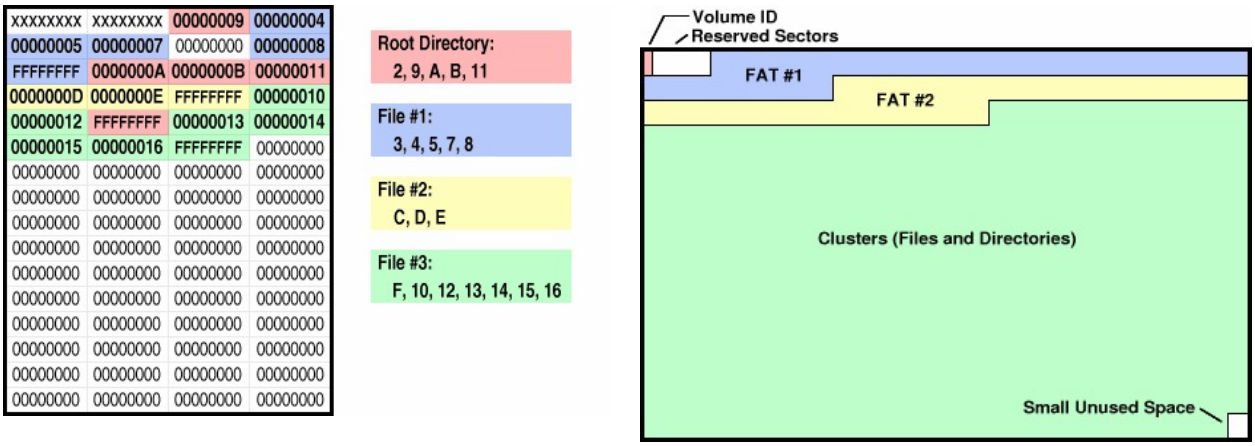

Figura 2: Vista general de la estructura del sistema de ficheros `FAT32`

Para poder encontrar un fichero, primero de todo es importante saberse ubicarse en el sistema de ficheros. Su estructura, así como en `EXT4`, se apoya en un sistema que la recursividad es la opción más indicada para su consulta. Tiene como en el caso anterior, una tabla de datos donde gestiona la información. Esta, llamada *Fat allocation table*, se encargara de gestionar que *clusters* pertenecen a quien.

```
off_t _fat32(char show, char *name);
```

Por esta razón, la primera función `_fat32`, se encargara de gestionar la llamada a la única función recursiva de búsqueda que hay para este sistema de ficheros, `deepsearch_fat32`. Así como mostrar el error pertinente o llamar a las funciones de muestra de información del fichero en caso de recibir el parámetro *-info*.

[1] https://www.pjrc.com/tech/8051/ide/fat32.html

```
off_t deepsearch_fat32(char *name, uint32_t position);
```

Como ya hemos comentado anteriormente, esta función tiene como objetivo recorrer todo los elementos que contiene la *Fat allocation table* para consultar sus contenidos y en este caso, buscar el fichero o mostrar el contenido de este.

Por lo tanto, irá ejecutando de manera recursiva los siguientes pasos:

1. Ir directamente al *cluster* de la primera posición, en la primera llamada, será el del directorio raíz.
2. Por cada elemento de tamaño `fat32_directory` (32 bytes) consultará primero su atributo, para ver si se trata de un *longname* o de una entrada correcta de directorio.
3. Si se trata de un *longname* leerá de manera iterativa tantas veces como el byte de `Sequence Number` indique, en caso contrario continuará con las siguientes comprobaciones.
4. Consultará que sea un fichero o directorio, sus atributos de "oculto" y en caso de directorio calcular el *cluster* siguiente y volverse a llamar con la nueva posición, volviendo de nuevo al primer paso.

Dado que en este caso no tenemos un identificador fácil de retornar donde haya los *metadatos* del fichero, retornamos directamente el *offset* del fichero donde se encuentra su `fat32_directory` con los atributos etc. De esta forma, si lo hemos encontrado y queremos extraer los datos de tamaño o cambiar sus permisos, sabemos directamente que tenemos que ir a esa posición y cambiar o leer la estructura pertinente. Comentar también, que para poder consultar el *cluster* una vez leída cada entrada, utilizamos la siguiente formula:

$$cluster\_position = (fat32.first\_cluster + ((fat32.bytes\_per\_sector \times fat32.sectors\_per\_cluster) \times position))$$

Juntamente con una constante, que nos sirve para juntar `fat32_dir.cluster_high` y `fat32_dir.cluster_low`, de 16 y 32 bits respectivamente.

```
    #define CLUSTER(h, l) (((((uint32_t) (h)) << 16) | ((l))) - 2)
```

```
void deepshow_fat32(off_t off);
```

Como en el sistema de ficheros anterior, esta función se encargará, de manera muy parecida a la anterior, consultar los datos del fichero y mostrarlos por pantalla. La diferencia de la anterior, que cada vez que accedemos a cada *cluster*, sabemos que hay datos directo por lo tanto no tenemos que consultar ningún tipo de atributo o estructura especial. Por otra parte, sabemos hasta qué limite tenemos que mostrar, primero cada vez que leemos el *cluster* a consultar, miramos su valor de la `FAT` y a la vez, dado que al buscar un fichero hemos retornado su offset con la información de este, sabemos lo que ocupa en bytes.

```
char *convert_UCS2_ASCII(uint16_t *in, char *out, int size);
```

Para acabar, para poder gestionar bien los *longnames*, hemos tenido que recortar la representación de los caracteres en `UCS-2` a `ASCII`, representados de 2 a 1 byte. Ya que los nombres de búsqueda se han introducido con una representación de un *byte* por carácter.

3. Estructuras de datos

Las únicas estructuras de datos empleadas en esta práctica han sido "registros", o `structs`, que proporciona el lenguaje de programación `C`. A continuación se expone cada uno de ellos y cuál es su finalidad.

Nota: todas estas estructuras están acompañadas del atributo `__attribute__((packed))` para que el compilador haga todos sus campos contiguos en memoria RAM.

3.1. EXT4

```
typedef struct {
    uint32_t size;
    uint32_t count;
    uint32_t free_count;
} Block;
```

```
typedef struct {
    uint64_t table_loc;
    uint16_t size;
    uint32_t count;
    uint32_t free_count;
} Inode;
```

```
typedef struct {
    Block block;
    Inode inode;
    uint16_t group_descriptor_size;
} EXT4_info;
```

Estas tres estructuras son el "building block" de la práctica. Como se ha dicho anteriormente, la función `ext4_get_structure()` extrae la información del volumen y la coloca en una variable global cuyo tipo es **EXT4_info**. De esta manera y siempre que sea necesario, se puede consultar el contenido de esta variable para tener cualquier tipo de información necesaria.

```
typedef struct {
    uint16_t eh_magic;
    uint16_t eh_entries;
    uint16_t eh_max;
    uint16_t eh_depth;
    uint32_t eh_generation;
} ext4_extent_header;
```

Estructura bastante utilizada durante todo el proceso de búsqueda. Se trata del conocido *header* del *extent tree*. De esta manera, cada vez que consultábamos el nodo padre del árbol o bien el bloque donde apunta un nodo raíz, siempre leíamos esta estructura para decidir qué camino a seguir dependiendo de sus campos.

```
typedef struct {
    uint32_t ei_block;
    uint32_t ei_leaf_lo;
    uint16_t ei_leaf_hi;
    uint16_t ei_unused;
} ext4_extent_idx;
```

```
typedef struct {
    uint32_t ee_block;
    uint16_t ee_len;
    uint16_t ee_start_hi;
    uint32_t ee_start_lo;
} ext4_extent;
```

Estas dos estructuras, las hemos utilizar para leer los diferentes tipos de nodos del *extent tree* de cada *inode*. Por una lado tenemos en la izquierda la estructura propia de un nodo raiz y por consiguiente, en la derecha la del nodo hoja.

La razón principal en definirlas fue para aumentar la simplicidad de la lectura de las entradas en el momento de consultarlas. En vez de tener que ir leyendo en disco, decidimos una vez teníamos el `ext4_extent_header` y por consiguiente el campo `eh_entries`, pedir memoria para tantas entradas de tipo raíz o hoja, dependiendo del `eh_depth`, e iterar en cada una de ellas.

```
typedef struct {
    uint32_t inode;
    uint16_t rec_len;
    uint8_t name_len;
    uint8_t file_type;
} ext4_dir_entry_2;
```

Entrada de una carpeta o un fichero dentro del bloque que está apuntando el nodo hoja de un *inode* de tipo directorio. A diferencia de la documentación, no añadimos el campo del nombre. La razón principal fue su carácter dinámico que no favorecia añadirlo dentro del *struct*.

3.2. FAT32

```
typedef struct {
    uint16_t bytes_per_sector;
    uint8_t sectors_per_cluster;
    uint16_t reserved_sectors;
    uint32_t sectors_per_fat;
    uint32_t root_first_cluster;
    uint32_t fat_location;
    uint32_t first_cluster;
} FAT32_info;
```

Es el equivalente de `EXT4_info` para `FAT32`. Esta estructura se llena con `fat32_get_structure()`. También existe una variable global con este tipo para ser consultada en cualquier momento.

```
typedef struct {
    char short_name[8];
    char file_extension[3];
    uint8_t attribute;
    uint8_t user_attribute;
    uint8_t created_time_ms;
    uint16_t created_time;
    uint16_t created_date;
    uint16_t access_date;
    uint16_t cluster_high;
    uint16_t last_modification_time;
    uint16_t last_modification_date;
    uint16_t cluster_low;
    uint32_t size;
} fat32_directory;
```

Así como en el anterior sistema de ficheros hemos comentado el gran uso de las estructuras de tipo `ext4_dir_entry_2`, en este caso actua de la misma manera. A partir de estos 32 bytes, nos movemos por todo el contenido del *cluster* para definir las acciones que tomar. Como campos más usados, el `attribute` y `short_name` están en las primeras posiciones. Gracias a estos conocemos atributos simples de lectura, escritura, si lo que estamos leyendo es una entrada `fat32_vfat` o si ya hemos acabado de leer toda el *array*.

```
typedef struct {
    uint8_t sequence;
    uint16_t name[5];
    uint8_t attribute;
    uint8_t type;
    uint8_t checksum;
    uint16_t name2[6];
    uint16_t first_cluster;
    uint16_t name3[2];
} fat32_vfat;
```

Únicamente se usa para los *longname* de una entrada. Los campos más importante de esta estructura son `name`, `name2` y `name3` dado que juntos forman el nombre entero de la entrada. Es importante destacar que los caracteres codificados en 16 bits (UCS-2). Sin embargo, cuando se va a usar este nombre, se convierte a un *string* donde cada carácter ocupa 8 bits.

4. Pruebas realizadas

Para poder confirmar el buen funcionamiento en la mayoría de casos del correcto funcionamiento del proyecto en el momento de mostrar, buscar y modificar los permisos de los ficheros, se ha tenido que realizar un número elevado de pruebas para satisfacer los requerimientos.

Separaremos las pruebas para las diferentes fases con la explicación de la prueba y los recursos utilizados para estas.

4.1. Fase 1

Primero de todo, para la primera fase de detectar el tipo de *File System*, los principales recursos han sido los volúmenes de prueba, no hacía falta unos en particular, sino uno de cada tipo, para corroborar el buen funcionamiento de la fase.

Era importante asegurarnos que las pruebas se realizaban correctamente ya que esta fase sería importante para las siguientes con el fin de detectar bien el sistema de ficheros y ejecutar la estrategia pertinente.

4.2. Fase 2/3

Una vez que ya teníamos la fase acabada, empezamos a desarrollar las fases consecuentes, las cuales dado que pasamos directamente a la fase 3, las hemos tratado como iguales. Primero de todo, se utilizo los volúmenes básicos de *Volume_1_1024_Block_Size.ext4* y *Volume_1.fat32*.

Estos dos volúmenes tienen la misma estructura de ficheros, por lo que simplificaba la comprobación de la estructura de los volúmenes a través de las pruebas para ambos. Llegados al punto del buen funcionamiento para ambos, fuimos utilizando diferentes volúmenes con caracteristicas particulares para cada sistema.

En el caso de `EXT4`, el uso de volúmenes con *Block Size* mayor a 1024, nos ayudó a solucionar problemas con la lectura de la información, ya que en casos de tamaños mayores a 1024, los *offsets* son diferentes. Y para `FAT32`, en volúmenes con nombres bastante largos como el mismo del *checkpoint*, nos ayudó a solucionar problemas de la lectura secuencial de estos.

4.3. Fase 4

La siguiente fase, separa el nivel de dificultad de generar pruebas para los dos sistemas. Por la estructuración interna de `EXT4`, para que un *inode* tenga nodos raíz en su *extent tree*, se necesita que el fichero en cuestión que gestiona este sea de grandes dimensiones. Para poder conseguirlo, se ha necesitado hacer uso de discos `USB`, pequeño programa encargado de escribir en un fichero de texto y con la comanda `disk dump` de *linux*, se ha generado el volumen necesario. En nuestro caso, bastó para generar un fichero de 2GBytes dentro de un sistema de ficheros `EXT4` con el *inode* conteniendo un nodo raíz y este gestionando 16 nodos hoja.

```
Level Entries          Logical          Physical Length Flags
 0/ 1   1/  1      0 - 488281  33247           488282
 1/ 1   1/ 16      0 -  32767  34816 -  67583  32768
 1/ 1   2/ 16  32768 -  63487  67584 -  98303  30720
 1/ 1   3/ 16  63488 -  96255 100352 - 133119  32768
 1/ 1   4/ 16  96256 - 126975 133120 - 163839  30720
 1/ 1   5/ 16 126976 - 159743 165888 - 198655  32768
 1/ 1   6/ 16 159744 - 190463 198656 - 229375  30720
 1/ 1   7/ 16 190464 - 223231 231424 - 264191  32768
 1/ 1   8/ 16 223232 - 253951 264192 - 294911  30720
 1/ 1   9/ 16 253952 - 286719 296960 - 329727  32768
 1/ 1  10/ 16 286720 - 319487 329728 - 362495  32768
 1/ 1  11/ 16 319488 - 352255 362496 - 395263  32768
 1/ 1  12/ 16 352256 - 382975 395264 - 425983  30720
 1/ 1  13/ 16 382976 - 415743 442368 - 475135  32768
 1/ 1  14/ 16 415744 - 448511 475136 - 507903  32768
 1/ 1  15/ 16 448512 - 464895 507904 - 524287  16384
 1/ 1  16/ 16 464896 - 488281 557056 - 580441  23386
~
```

```
Inode: 12   Type: regular    Mode:  0664   Flags: 0x80000
Generation: 848584765    Version: 0x00000000:00000001
User:  1000   Group:  1000   Project:     0   Size: 2000000031
File ACL: 0
Links: 1   Blockcount: 3906264
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x5afe95f7:94624a68 -- Fri May 18 10:59:35 2018
 atime: 0x5afe95f7:7888fea0 -- Fri May 18 10:59:35 2018
 mtime: 0x5afe9453:56683d40 -- Fri May 18 10:52:35 2018
crtime: 0x5afe954c:788900a8 -- Fri May 18 10:56:44 2018
Size of extra inode fields: 32
Inode checksum: 0x74d9205a
EXTENTS:
(ETB0):33247, (0-32767):34816-67583, (32768-63487):67584-98303, (63488-9625
5):100352-133119, (96256-126975):133120-163839, (126976-159743):165888-1986
55, (159744-190463):198656-229375, (190464-223231):231424-264191, (223232-2
53951):264192-294911, (253952-286719):296960-329727, (286720-319487):329728
-362495, (319488-352255):362496-395263, (352256-382975):395264-425983, (382
976-415743):442368-475135, (415744-448511):475136-507903, (448512-464895):5
07904-524287, (464896-488281):557056-580441
```

Figura 3: Estructura del *extent tree* del fichero y sus *metadatos*

Por consecuente, la generación de métodos de prueba para este sistema de ficheros ha tenido mayor dificultad para `EXT4` que para `FAT32`, el cual, con llenar más de un *cluster* entero de información y mostrarlo correctamente ya era suficiente para validar el funcionamiento.

4.4. Fase 5

Para terminar, la última fase, el método de pruebas ha sido el más simple de los anteriores. Dado que simplemente es cambiar un byte dentro dentro de la estructura del *inode* o del `fat32_directory`, con leer lo que vamos a sobre-escribir y lo que hemos sobre-escrito, nos ha servido para corroborar el buen funcionamiento. Por requerimientos del enunciado, no se ha procedido a montar el volumen de nuevo para comprobarlo, ya que para este caso, deberíamos recalcular el *checksum* del *inode* en cuestión.

5. Problemas observados

5.1. Fase 1

Esta fase no supuso problemas grandes aparte del primer "choque con la realidad" que consistía en empezar a entender los diferentes sistemas de ficheros y su manera de organizarse. El único cambio que tuvimos que realizar más adelante en referencia a esta fase fue en la manera de detectar sistemas de ficheros `FAT32`. La razón fue por el hecho de consultar una posición de datos que era susceptible a no funcionar bien en el proceso de diferenciar entre `FAT32` y `FAT16`.

5.2. Fase 2/3

Ya entrando en la parte troncal del proyecto. Durante estas fases tuvimos una serie de problemas tanto para `EXT4` como `FAT32`.

5.2.1. EXT4

Este sistema de ficheros tiene una estructura muy organizada. Por esta razón, su comprensión desde un buen inicio puede ser complicada, ya que la estructura de árbol, *inode* y la manera de organizar cada *block group* no es trivial. Partiendo de esta base, el desarrollo de estas dos fases en su buen inicio fue forzado, tuvimos que entender bien como funcionaba cada elemento y comprobar, sobretodo mediante el editor de ficheros por comanda `VIM`, que las suposiciones que teníamos de ubicación y estructuración tenían sentido. Generando los siguientes problemas:

- Problemas para encontrar el *inodes* en la *inode table*. Principalmente fue debido a una falta de comprensión del *File System*, dado que su solución fue establecer que todos los números de *inodes* leidos en cada `ext4_dir_entry_2`, se debería restar dos, ya que los dos primeros no existen.
- Falta de comprobaciones en el momento de leer un bloque que estemos en el final. Ocasionando lecturas indebidas de otros bloques y bucles infinitos.
- Cuando leíamos estructuras de tipo `ext4_dir_entry_2`, nos encontrábamos que el campo de `name_len` más todos los bytes sumados de la estructura no equivalían a `rec_len`, provocando un desplazamiento de los datos leídos y por lo tanto, lectura de datos incorrecta.
- Parecido al caso anterior, el campo `rec_len` de la estructura `ext4_dir_entry_2` en ocasiones daba valores fuera de lo común. Por lo tanto la solución fue establecer que la lectura de cada elemento de la estructura fuese una combinada entre este y `name_len`, comprobando siempre que no excediese el máximo de `EXT4_NAME_LEN` más los ocho bytes de los otros campos.

5.2.2. FAT32

A diferencia del anterior sistema de ficheros comentado, este consiste en una estructura bastante más simplificada sin tantos *metadatos*. Haciendo que su comprensión y ejecución sea más simple y directa. Aún así, no deja de tener sus complicaciones y por lo tanto, problemas a la hora de implementar la búsqueda de ficheros con este.

1. El principal problema, seguramente la parte más confusa, es en referencia a los *longnames*. La estructuración de este sistema establece dos tipos de nombres, separados en dos estructuras diferentes. El problema recae en que estas están juntas en memorias, mezcladas entre cada entrada haciendo que se tenga que tener una buena comprensión de los campos y sus valores para saber diferenciar entre una y otra. Ocasionándonos problemas a la hora de saber en que momento leer que, en un primer momento.

2. Una vez que supimos que estructura recae en cada sitio a la hora de su lectura, leer correctamente el nombre del fichero separada en estas estructuras conocidas como `VFAT long file names`, fue un poco confuso. En primera instancia por el hecho que el nombre está representado en un formato de caracteres llamado `UCS-2`. Organizado en dos bytes en vez de un único típico en `ASCII`. Por lo tanto, como solución fue extraer la parte inferior de cada representación del carácter y descartar la parte de arriba.

   Llegados a este punto, juntar este *array* de estructuras organizadas del final al principio también tuvo su complicación para que funcionase a la perfección.

3. Finalmente, por la estructura partida de los *longnames*, usando memoria dinámica nos generaba errores al limpiar la memoria. Pasándonos al uso de memoria estática, dado que no había limitación de máximo uso de memoria.

5.3. Fase 4

Llegados a este punto después de haber hecho las pruebas necesarias con la fase 2 y 3, la implementación de esta fue bastante rápida. Sin embargo, hubieron momentos de estancamiento provocados por pequeños errores al definir limites y maneras de mostrar el contenido. A diferencia de las anteriores fases que las hemos separado para cada sistema de ficheros, en este caso los problemas se comparten mayormente.

- En el momento de mostrar el contenido, nos encontramos que aunque el fichero indicaba que habían mas de 0 bytes de contenido, cuando mostrábamos el contenido, este no eran caracteres *printables*. Gracias al uso de la comanda `cat` de *linux*, pudimos corroborar que este no tenía el mismo problema. Para solucionarlo establecimos que al detectar el carácter nulo (`'\0'`), se dejase de mostrar el contenido, evitando casos de overflow o errores de calculo de tamaño.
- En el caso de `EXT4`, establecer el limite de muestra de los caracteres a partir del tamaño establecido en el *inode* nos generó problemas. Decidiéndonos por establecer el limite a partir del campo `ee_len` por el tamaño de cada bloque, de cada nodo hoja.

5.4. Fase 5

Finalmente, para esta fase, el problema principal fue causado en `EXT4`. En el momento de cambiar los permisos de un fichero y montar el volumen para verificar que se han cambiado correctamente, aparecía lo siguiente al ejecutar `ls -l`:

Figura 4: Salida de `ls -l`

Después de este resultado y haber consultado la documentación, por una parte puede ser causado por el *Journal* de `EXT4` y juntamente con el *checksum* del *inode*. Ya que este deja de ser correcto al realizar este cambio. Al comentar la situación con un monitor de prácticas de la asignatura se concluyó que no era necesario arreglar esto.

Por otra parte, para ambos sistemas de ficheros nos encontramos que inicialmente (desde la fase 1) se abría el fichero (volumen) con permisos únicamente de lectura. Debido a esto, cuando se realizaba una escritura con la función `write()` y luego se intentaba leer lo que se había escrito, se apreciaba que el dato *original* no había cambiado. Este problema fue solucionado cambiando la forma de abrir el fichero con `open()`. En lugar de abrirlo en modo `O_RDONLY` pasó a abrirse en modo `O_RDWR` de tal manera que se permitía leer y escribir en el volumen.

6. Estimación temporal

Durante el desarrollo de esta practica, se han empleado **39 horas** en total. A continuación exponemos de forma visual las horas dedicadas para cada parte de este.

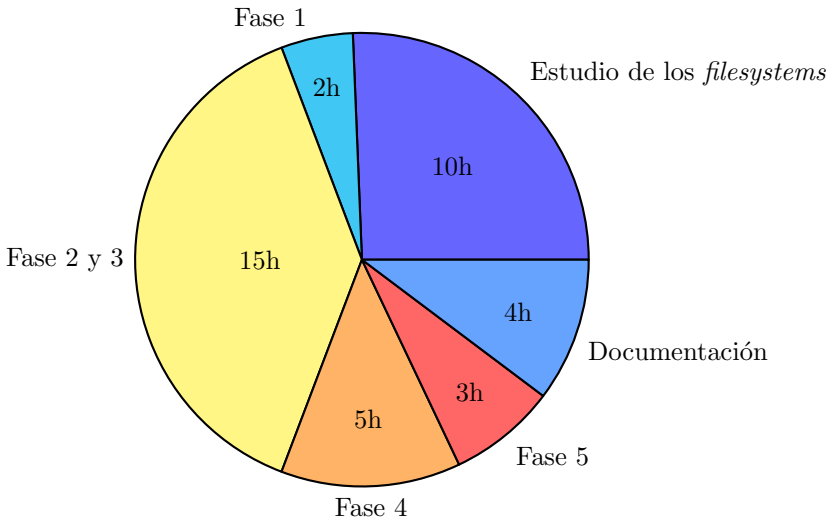

Figura 5: Representación en forma de diagrama circular de la distribución del tiempo

Tal como ya hemos comentado con anterioridad, para no tener que realizar doble trabajo, la fase 2 y 3 se juntaron en una única y se realizaron al mismo tiempo. Junto con la fase 4 fueron las partes de la práctica a las que más tiempo se le invirtieron (un total de 20 horas).

7. Conclusiones

Llegados a la finalización de esta practica, se puede confirmar que se trata de una continuación totalmente necesaria de la asignatura de Sistemas Operativos. No solo por el uso del lenguaje `C`, sino porque contiene la misma filosofía de entender de que manera se gestiona los recursos y qué posibilidades existen con los sistemas actuales.

Centrada en los sistemas de ficheros, esta practica ha aportado un conocimiento, hasta día de hoy desconocido, de la manera que se estructuran los datos en disco. Aumentando aún más la visión de que a vista de un programador de sistemas operativos, todo no deja de ser bytes, siendo el fichero que sea. Y por consiguiente, es la manera que uno mismo trata los datos que definen que usos tienen y que interprete es el encargado de gestionarlos.

En la primera fase pudimos ver un inicio de este concepto, pudiendo separar y detectar que tipos de sistemas leemos. Y aunque en su momento nos pareció una labor complicada, no deja de ser leer unos bytes determinados en una posición determinada y el único objetivo, es esperar que esos tengan un valor concreto. Tan simple como eso.

Acabada la primera toma de contacto, el camino a seguir aunque más difícil, seguía en el mismo concepto y por lo tanto, la visión que teníamos ya estaba moldeada y preparada para afrontarlo. Decidimos desde el primer momento, juntar las dos fases siguientes, ya que seguían el mismo objetivo pero con la diferencia que una de ellas se quedaba atrás en funcionalidades.

El momento de desarrollo de estas dos fases, fue el punto culminante de entendimiento y comprensión de los dos sistemas. Pudimos ver el gran tiempo dedicado a través de los años, de los creadores de ambos, `EXT4`: *Mingming Cao*, *Andreas Dilger*, *Alex Zhuravlev*, entre otros y `FAT32`, sin autores como tales pero con multinacionales siendo participes en el desarrollo como, Microsoft, NCR, IBM, Compaq, etc. para poder traer a la industria sistemas fiables que se pueda depender para el uso general.

Así pues, tanto para la fase 2 como 3, desarrollamos la capacidad de poder discutir y razonar la manera que teníamos que afrontar nos a los diferentes problemas que nos iban surgiendo para cada sistema de ficheros. Siendo de gran ayuda para la implementación de las siguientes fases, ya que llegados a este punto, su implementación era dedicarle horas de programación y no tanto de investigación.

Para culminar esta practica, podemos decir que tanto el objetivo como la manera a la hora de desarrollarla ha sido totalmente ajustada a nuestros intereses. Como hemos comentado anteriormente, el hecho que se haya establecido este proyecto como una continuación de lo que hemos aprendido en sistemas operativos, hace que la visión creada al largo de este año mediante estas dos asignaturas, nos aporte un nivel técnico superior del funcionamiento que tienen los sistemas que utilizamos día a día. Formándonos para que podamos seleccionar en un futuro la rama de la informática que más nos guste, teniendo una buena base con la que basar nuestras decisiones.

8. Valoraciones

Para el desarrollo de esta practica, uno de los aspectos que más nos ha parecido más acertado es la manera de trabajo que se ha planteado. Tal como se comenta en el enunciado, la conocida metodología *on demand*, ha hecho que podamos descubrir por nuestra propia cuenta y por consiguiente, incrementar nuestra habilidad resolutiva. El hecho que hayamos tenido que dedicarle horas para ampliar nuestro conocimiento e intentar entender la manera que ambos sistemas de ficheros se gestionan internamente, ha hecho que no pase como una practica totalmente desapercibida la cual no tiene aportaciones finales después de implementarla.

Por otra parte, se agradece mucho el hecho de que se hayan dejado todas las clases después del punto de control para trabajar en la práctica. De esta manera se garantizaba que ambos integrantes del grupo trabajaban al mismo tiempo y a la vez, creaba entornos de discusión sobre mejores maneras de afrontar los distintos problemas que cada grupo tenia.

Finalmente, como aporte final, se ha valorado el buen uso de del repositorio de *GitHub* con el uso de los *issues*, lo cual llegó a ser realmente útil (aunque no hayan sido usados al cien por cien), ya que ha aportado una introducción a la actualidad en el mundo del desarrollo de software.

Índice de figuras

Referencias

[1] Paul, "Understanding fat32 filesystems," 2005. [Online]. Available: https://www.pjrc.com/tech/8051/ide/fat32.html

[2] "struct tm - c++ reference," 2017. [Online]. Available: http://www.cplusplus.com/reference/ctime/tm/

[3] Aparci-Desseau, "File intro," 2014. [Online]. Available: http://pages.cs.wisc.edu/ remzi/OSTEP/file-intro.pdf

[4] ——, "File system implementation," 2014. [Online]. Available: http://pages.cs.wisc.edu/ remzi/OSTEP/file-implementation.pdf

[5] J. Hopkins, "An analysis of ext4 for digital forensics," 2012. [Online]. Available: www.elsevier.com/locate/diin

[6] A. Mathur, "The new ext4 filesystem: current status and future plans," 2007. [Online]. Available: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf

[7] "Ext4 disk layout - ext4," 2018. [Online]. Available: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

[8] "Design of the fat file systems," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system

[9] "File allocation table," 2018. [Online]. Available: https://en.wikipedia.org/wiki/File_Allocation_Table

[10] A. Mathur, "The new ext4 filesystem: current status and future plans," 2007. [Online]. Available: https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system