Sistema de Archivos FAT32 mediante FUSE (Junio 2020)

Rodrigo Díaz Rodríguez, Anastasiya Oxenyuk

***Resumen*—El proyecto se centra en un Sistema de Archivos tipo Fat para Linux con base en el módulo FUSE. Sus funciones principales serán el control de archivos y directorios, tanto como su almacenamiento dentro de la Fat por medio de funciones de la API de FUSE.**

I. Introducción

El proyecto concibe la idea de un sistema de archivos de usuario basada en FAT32.
Esta idea es originaria del profesor Andrés Rodríguez Moreno, el cuál además de ofrecer su idea puso a nuestra disposición un código base sobre el cuál trabajar . Usaremos la librería de FUSE para controlar el sistema de archivos, e imágenes FAT32 en las que se encontraran registrada toda la información del sistema.

II. Especificaciones

Para el Proyecto se ha usado principalmente la librería de FUSE. FUSE es acrónimo de Sistema de Archivos en espacio de usuario, ¿esto que significa? Nos permite crear sistemas de archivos propios sin editar el código del kernel, acciones normalmente solo disponibles para el usuario root. Esto se logra mientras ejecutas el código de fuse desde un espacio de usuario, mientras que el módulo de FUSE realiza un "puente" hacia las interfaces del kernel.

Para implementar un nuevo sistema de archivos, se debe escribir un programa de controlador vinculado a la biblioteca libfuse suministrada. El objetivo principal de este programa es especificar cómo debe responder el sistema de archivos a las solicitudes de lectura / escritura . El programa también se utiliza para montar el nuevo sistema de archivos. Si un usuario ahora emite solicitudes de lectura / escritura para este sistema de archivos recién montado, el núcleo reenvía estas solicitudes de Entrada / Salida al controlador y luego envía la respuesta del controlador al usuario.

Las operaciones de FUSE son muy similares a las originales de Unix, existen muchos métodos que nos permite variar y personalizar nuestro sistema de archivos. Estos métodos por lo general son opcionales, aunque hay alguno que otro que es obligatorio al ser esencialmente necesarias para el funcionamiento del sistema.

Descripción de las operaciones usadas:

"init": Inicializa el sistema de archivos, además sirve para configurar este sistema asignando datos a estructuras.

"getattr" Llamado por el sistema cuando este intente obtener los atributos de los archivos o directorios. Los datos del guardan en la estructura "stat".

"readdir" Devuelve al usuario que llama todas las entradas del directorio excepto las especiales. Es esencial para un sistema de archivos.

"open" Encuentra la entrada de directorio de un fichero, y abre el fichero actuando en consecuencia de la verificación de sus identificadores y permisos.

"read" Lee los datos de un archivo pasado por "path" previamente abierto. Devolviendo el mismo nº de bytes al de los solicitados a partir del "offset" dado.

"write" Escribe la información recibida desde la consola en un archivo previamente abierto, teniendo en consideración los mismos parámetros que read.

"truncate" Cambia el tamaño del fichero al tamaño dado por parámetro.

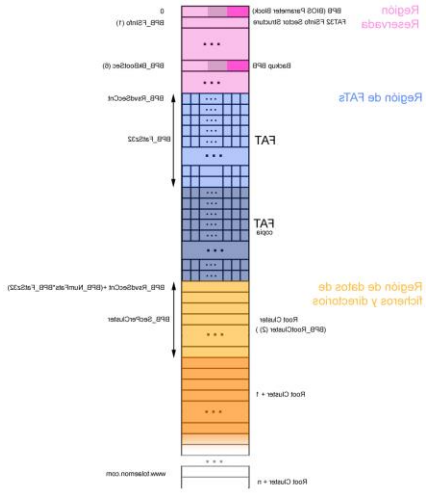
"rmdir" Elimina el directorio pasado por parámetro. Como precondición el directorio debe estar vació.

"unlink" Elimina el fichero pasado por parámetro.

Puesto que nuestro proyecto se basa en almacenar la información en una FAT32 además utilizamos una imagen la cual editamos y leemos a tiempo real.

La FAT32 es un espacio físico de almacenamiento que se organiza en 3 partes centrales:

.



Región reservada: Información fija que indica el funcionamiento de la Fat. Y almacena información relevante de esta.

Región de las Fats: FAT (Tabla de asignación de archivos) proporciona una tabla de entradas de los clústers en los que se almacenan los archivos y directorios. El sistema de archivos se encarga de recorrer la FAT, en búsqueda de los sucesivos clústers hasta llegar al final de archivo.

Región de datos de directorio y fichero: Zona donde se almacena el contenido de los directorios y archivos. Se podrá añadir información siempre que se dispongan suficientes clústeres libres. Cada clúster se encuentra enlazado con el siguiente por medio de punteros. Si un clúster no se ocupa completamente, el espacio restante se desperdicia.
.

III. Manejo de la Aplicación

El manejo de nuestro Sistema de Archivos tiene la finalidad de ser lo más parecido al universal UNIX.

Lo primero será montar el Sistema de Archivos. Una vez dentro el usuario podrá realizar las clásicas acciones referentes a los archivos y directorios. Tales como:

Trasladarnos por los distintos directorios posibles (comando cd), ver el contenido de dichos directorios (comando ls) además de sus distintos atributos (comando ls -l).

El usuario tendrá la oportunidad de interactuar con los ficheros existentes, editando su contenido (comando echo, editor vi). Este nuevo contenido se vera reflejado en los atributos del fichero.

Siempre podrá visualizar su contenido actual (comando cat). Y a gusto eliminar ficheros o directorios (previamente vacíos) (comandos unlink, rmdir).

Por último, el usuario saldrá del sistema de ficheros y podrá desmontarlo.

Toda la información de este guardada en una Imagen.

IV. Funcionamiento Interno

Vamos a describir cómo es la aplicación desde dentro, es decir, su código componente.

Tenemos el archivo de cabecera *fat32.h* en el que tenemos declaradas las distintas estructuras, también disponemos del archivo de mayor importancia *fat32.c* que contiene la mayor parte del código. Por último, *Makefile* el cual permitirá al usuario montar y desmontar el sistema de archivos haciéndose uso de los ficheros: *fat32, fat32.o*, y *file.img*(imagen de la Fat en la que se encuentran guardados los datos).

fat32.h

`struct bios_param_block:` Contiene la información elemental del dispositivo FAT.
`struct fs_information_sector:` Estructura auxiliar que dispone información actualizada sobre la FAT (pj. primer clúster libre).
`struct directory_entry:` Guarda la información relevante y útil sobre el archivo o directorio.
`struct long_filename_entry:` Permite guardar los nombres de directorio o fichero mayores de 11 bytes.
`Struct structura_mis_datos:` Utiliza las estructuras definidas anteriormente para almacenar la información contenida en la imagen.

También en esta cabecera se incluyen varias constantes que se usaran en el código.

fat32.c

Lo primero que hacemos en el código es incluir los archivos de cabecera que usaremos, como parte de nuestro proyecto remarco la de *fuse.h* que nos permite usar sus operaciones.

La función principal del ***main*** de este fichero es abrir la imagen FAT que pasaremos como parámetro gracias al Makefile. Se controlarán el posible error de apertura de la imagen.

A continuación, definiremos las funciones necesarias para que el sistema de ficheros funcione correctamente. Y las complementarías para poder obtener y guardar la información de la imagen de la FAT.

FAT32_init

Aquí es donde a partir del *file descriptor* se cargan los datos de la imagen en la anteriormente nombrada estructura de *structura_mis_datos.*

FAT32_readdir

Esta función se encarga de la visualización de los ficheros/directorios que se encuentran dentro del directorio pasado por *path*.

Concretamente obtenemos la entrada del directorio (estructura *directory_entry*) del path pasado por parámetro mediante el método aux. *encuentra_entrada(),* posteriormente calcularemos el primer clúster en el cual se encuentra su información mediante las variables de la estructura *directory_entry: First_Cluster_Low* y *First_Cluster_High.*
A continuación, se leen todas las entradas de directorio en todos los clústers correspondientes al *path.* Cada vez que encontremos una entrada de directorio válida (tipo archivo/directorio no borrado) meteremos su nombre de entrada en la variable *buf.*

FAT32_geattr

La tarea principal de esta operación es declarar los atributos de los posibles ficheros/archivos del sistema de archivos, entre ellos indicar de que tipo son, fecha, permisos.

La manera en la que se logra dicha tarea:

Mediante el *path* comprobamos si es el directorio raíz, si es así le indicamos de permisos específicos 0755(el propietario tiene todos los permisos, mientras que los demás de lectura y ejecución). Si no es directorio raíz deberemos averiguar que tipo de entrada es, esto lo logramos con el método aux. *encuentra_entrada()* y su variable *Attributes* que indicara si es un directorio (Permisos 0555) o archivo. Si es un archivo, FAT32 solo soporta dos modos de permisos **1**: solo lectura y ejecución 0555 **2**: todo permitido 0777.

A parte de los permisos se rellenarán otros atributos de entrada como tamaño, ids, fechas (usamos la función *conv_time()*).

En caso de que no se encuentre una entrada de directorio correspondiente al *path* daremos error.

FAT32_open

Esta función es llamada cada vez que el sistema quiera realizar acciones sobre un archivo, pues antes que nada habrá que abrir y comprobar que puede realizar dichas acciones.
Para ello, encontramos la entrada correspondiente al *path* mediante el método aux. *encuentra_entrada*() y comprueba su variable *Attributes,* dependiendo de esta se permitirán unas u otras operaciones. Además, guardaremos el primer clúster en el que se encuentra la información del archivo en la variable *fi->fh* .

FAT32_read

Esta operación se ocupa de leer dentro del *buf* pasado como parámetro la información del archivo que se encuentre en la *path* previamente abierto. Con ese fin tomamos el clúster guardado en la variable *fi->fh, y* teniendo en cuenta el offset calculamos a partir de donde tenemos que leer la información.
Estaremos leyendo hasta completar el tamaño deseado.

FAT32_write

Esta operación consta de tres partes.

En la primera parte: Primero de todo encontramos la posición del clúster correspondiente al *path* mediante el método aux. *encuentrapos()* y le sumamos 28 bytes.
A partir de esta nueva posición leemos los siguientes 4 bytes que indicaran el tamaño del archivo. Como excepción en caso de que vayamos a escribir en mitad de la información existente, le tenemos que quitar el *offset.*
A continuación, le sumaremos el *size* , tamaño de lo que tenemos que escribir. Una vez hechas todas las modificaciones de tamaño sobrescribiremos este, en la posición anteriormente obtenida, para indicar el nuevo tamaño de archivo.

En la segunda parte cambiaremos la FAT. Primeo calculamos el nº de clústers necesarios para el actual tamaño del archivo. Recorreremos los clústers anteriormente reservados mediante la función aux. *readFAT().* Si nos encontramos un clúster que indica *FIN* y todavía nos faltan por reservar clústers, buscaremos el siguiente clúster libre mediante el método aux. *encuentraPosLibre()*. Tendremos que señalar este nuevo clúster y añadirlo a la FAT. En caso de que hayamos terminado de reservar los clústers sin encontrar *FIN*, tendremos que inscribirlo en el clúster acompañado de ceros en el espacio sobrante mediante la función *writeFAT().* Por último, en caso de terminar de reservar los clústeres y existe una marca de *FIN* no tocaremos la FAT.

La tercera parte consiste en escribir la información dentro de los clústers. Tenemos que encontrar el clúster a partir de cual debemos de escribir indicado por el *offset*. Una vez encontrado, escribiremos los datos que nos permita su capacidad. A continuación, pasaremos al siguiente clúster reservado. Repetimos la acción hasta completar el tamaño deseado de la información.

FAT32_truncate()

Busca la posición en la que se encuentra la entrada correspondiente al *path* mediante la función aux. *encuentrapos().* A esta entrada de directorio le cambiaremos el tamaño por el deseado.

FAT32_rmdir()

Encontramos la posición en la que se encuentra el directorio pasado por *path* con el método *encuentrapos(),* a partir de esa entrada de directorio encontramos el clúster en el que se encuentra su información.

El procedimiento aplicado ahora es prácticamente igual que en FAT32_readdir. Cada vez que encontremos una entrada válida comprobaremos si es de tipo ‘ **.** ‘ (directorio actual) o ‘ **..** ’ (directorio padre), en caso de encontrar uno diferente devolveremos error (es decir el directorio a eliminar no está

vacío).

Una vez comprobado que el directorio se encuentre vació, escribiremos en todos sus clústers de información, ceros para indicar que el clúster vuelve a estar libre. Posteriormente situamos en el primer byte de entrada del directorio el valor de *ATTR_DELETED* que indica que es una entrada borrada.

FAT32_unlink

Tal y como anteriormente, encontramos la posición en la que se encuentra el fichero pasado por *path*, y a partir del cual encontramos el clúster donde se encuentre su información.

Modificamos la información de su clústers a 0, indicando que vuelven a estar vacíos. Ponemos el primer byte de su entrada con el valor de *ATTR_DELETED* es decir entrada borrada.

Funciones Auxiliares

readFAT()

Lee la entrada de la FAT de la posición del clúster indicado por parámetro. Devuelve un entero *data* que contiene lo leído.

writeFAT()

Escribe *data* en el clúster indicado por la variable *lugar*, tanto en la FAT como en la copia de esta**.**

readCLUSTER()

Lee toda la información del clúster indicado. Esta información se guarda en el *buffer.*

entrada_vacia()

Comprueba si la entrada concreta se encuentra vacía o no.

get_long_filename()

Busca en todas las entradas largas asociadas a un *directory_entry*, y con ellas compone el nombre completo de la entrada de directorio.

encuentra_entrada()

El objetivo de este método es devolver la copia de la entrada de directorio adjunta al path pasado por parámetro.
Funciona del siguiente modo:

Primero de todo analizara el *path*, guardando los directorios y subdirectorios separador por el símbolo'/' en la variable *token*.

Posteriormente comprobamos si nos encontramos en la raíz es decir que el path sea igual '/'. Si es así devolveremos el root clúster.

En cualquier otro caso, se recorrerán todas las entradas de los clústers. Cada vez que se encuentre una entrada de directorio calcularemos su nombre completo con el método aux. *get_long_filename()*, comprobando si coincide con *token.*
En caso de coincidencia y siendo el último *token*, significa que hemos encontrado nuestro destino de *path,* por lo consiguiente devolvemos la copia de la entrada directorio encontrada.
Por otro lado, en caso de coincidencia y quedando más *tokens* por recorrer, en caso de ser un fichero nuestra posición actual se devolverá un *NULL* indicando error. En caso de directorio, continuaremos el recorrido, trasladándonos al clúster que nos indica la entrada de directorio actual, hasta encontrar el *path* deseado o devolver error.

Cada vez que se termina la lectura de las entradas existentes en un clúster, buscaremos el siguiente clúster dentro de la FAT, para ello invocamos el método aux. *readFAT().*

Posibles errores: tropezar con una entrada vacía, en caso de agotamiento de clústers por leer.

encuentrapos()

Función similar a *encuentra_entrada(),* a diferencia de la primera, esta devuelve la posición dentro de la imagen en la que se encuentra la entrada.

conv_time()

Función que convierte la fecha al formato deseado.

encuentraPosLibre()

Comprueba uno a uno todos los clústers de la FAT devolviendo la primera posición del clúster libre.

V. Errores Encontrados

Nuestros principales problemas se han debido al uso de la FAT32, nos hemos tenido que enfrentar a la comprensión correcta de su funcionamiento interno y entendimiento del significado de cada parte de la imagen. También al usar de base un código externo debíamos deducir como interactúa este con la imagen de la FAT32, de qué manera implementa sus funciones de guardado y extracción de datos.

Nos hemos encontrado errores concretos tales como:

Si un fichero tiene de nombre únicamente números se producen fallos en la imagen, con la intervención de FUSE no se procesan de manera correcta.

En un principio al extraer el primer byte de una entrada de directorio que se encontraba borrada devolvía un valor negativo. Incorrecto y rotundamente diferente al que se suponía que devolvía.

En cuanto a FUSE particularmente:

Si concebíamos por error un bucle infinito, no teníamos modo de cancelarlo. Producía errores en el sistema, impidiéndonos desmontar el sistema de archivos o simplemente borrarlo. Estábamos obligados a borrar el proyecto entero y recuperar versiones anteriores.

Las funciones iniciales existentes en el código externo no funcionaban de la manera correcta o deseada. Debíamos encontrar donde se originaba el error y el modo de corregirlo.

Finalmente hemos tenido problemas en la implementación de las funciones de FUSE mkdir y mknod encargadas de la creación. Para hacer su implementación habría que modificar mucho código y no veíamos la manera correcta de hacerlo sin generar en el camino montones de fallos. Por lo tanto, hemos decidido no implementarlas.

VI. Posibles Mejoras

En un principio las funciones que tenemos descritas funcionan de manera correcta, es verdad que siempre es posible optimizarlas, pero actualmente no vemos la necesidad obligada.

Si pudiéramos continuar con el proyecto y seguir investigando, nos gustaría poder completarlo con más funciones y acciones a posibles a realizar por el usuario. Tales como las anteriormente comentadas de **creación de ficheros/archivos**.
Otra como la función de FUSE **rename** que permitiría cambiar de nombre un fichero.

Referencias

http://www.tolaemon.com/docs/fat32.htm
//introducción a FAT32
http://libfuse.github.io/doxygen/structfuse__operations.html#a79cc0238ae11defe11755c825d51ca93
//Referencias de las operaciones de FUSE
https://www.man7.org/index.html
//referencias a funciones c Linux
https://bitbucket.org/DSO2014/fuse_fat32/src/master/
//Código externo, autor Andrés Rodríguez Moreno

Todo el código, e imágenes que hemos usado en nuestro proyecto se encuentran en el github
https://github.com/rodriwito/ProyectoFuseFat32

Autores

Rodrigo Díaz Rodríguez

Anastasiya Oxenyuk