Universidad de Murcia
Facultad de Informática
Diseño y Estructura Interna de un SO
Juan Piernas, Diego Sevilla
5º de Ingeniería Informática

Dedupfs
Sistema de ficheros con deduplicación basado en FUSE

Convocatoria de Junio, curso 2014-2015

Juan José Andreu Blázquez
<juanjose.andreu@um.es>
1 de junio de 2015

Índice

1. Resumen del trabajo realizado

El objetivo de este proyecto de programación práctico ha sido desarrollar un sistema de ficheros con FUSE capaz de detectar cuándo se almacenan ficheros idénticos, y dejar solo una copia de los datos en el sistema de ficheros, haciendo que el resto de ficheros sean simples apuntadores a la copia que contiene los datos. Esto permite que, en sistemas de ficheros donde hay muchos ficheros iguales, se ahorre una cantidad importante de espacio en disco.

Esta deduplicación de ficheros idénticos, se hace de forma transparente al usuario del sistema de ficheros, por lo que se siguen viendo ambos archivos como archivos diferentes, y en caso de modificarse uno de ellos, se volverán a separar y quedarán como dos ficheros diferentes. El usuario no llega a percibir estos cambios, y él utiliza su sistema de ficheros de forma normal, pero se ahorra espacio si tiene muchos archivos iguales. Para detectar si dos ficheros son idénticos, se ha utilizado la función SHA1 para calcular la clave de dispersión del archivo. Con esta clave se almacenan los datos en un directorio del sf, con un fichero que tiene como nombre el código hexadecimal del hash, y se deja el fichero original vacío como un marcador. En una base de datos interna, se almacenan una relación de cada nombre de fichero y qué clave de dispersión le corresponde, además de algunos parámetros adicionales que se indican más adelante. Esto permite saber dónde se encuentran realmente los datos de cada fichero, y se pueden mover fácilmente de un lugar a otro cuando se modifican y se recalcula su clave de dispersión.

El cálculo de la clave de dispersión es algo costoso, puesto que implica leer el fichero completo, y realizar cálculos sobre los datos del fichero para obtener la clave. Debido a esto, es necesario buscar el momento más adecuado para calcular la clave y evitar un impacto considerable sobre la velocidad del sistema de ficheros. Este momento adecuado se produce cuando un archivo ha sido abierto para escribir en él por uno o más procesos, se ha escrito en él, y lo cierra el último de los procesos que lo tenía abierto. Entonces se recalcula la clave de dispersión y, si ésta ha cambiado, se mueven los datos al fichero que corresponda a esa clave.

Además del deduplicado de archivos, también es necesario asegurarse de que otro tipo de funcionalidades típicas de un SF en UNIX están correctamente implementadas. Para asegurar que el funcionamiento de los enlaces físicos permanece intacto en nuestro SF, se ha utilizado una tabla adicional en la base de datos que almacena para cada enlace, cuál es el archivo que posee los contenidos. En cuanto a los enlaces simbólicos, no es necesario llevar a cabo ninguna acción especial para que funcionen de forma normal.

Por último, los permisos y el resto de atributos que tiene un archivo, se almacenan en el archivo vacío que se deja de marcador, a excepción del tamaño de archivo, que también se almacena en la base de datos junto a cada archivo para facilitar el acceso posterior a esta información.

2. Glosario de conceptos

A lo largo de esta memoria explicativa del desarrollo de la práctica propuesta, se utilizan una serie de términos cuyo significado se explica a continuación:

- **Hash/Clave de dispersión**: Clave de 20 bytes generada a partir de un archivo mediante la función criptográfica SHA1 y que se utiliza como identificador del contenido de un archivo.
- **Deduplicar/deduplicado**: Cuando se habla de deduplicar un archivo, o de uno deduplicado, se hace referencia a la acción de coger dos archivos idénticos que están duplicados en el sf, y dejar solo uno conteniendo los datos.
- **Marcador**: Un marcador es el archivo vacío con el nombre original que se deja en el lugar donde se creó un fichero en primer lugar.
- **Archivo de datos**: Archivo que se crea en una carpeta oculta del sistema de ficheros y que contiene los datos a los que apuntan los marcadores. Su nombre se corresponde con el hash generado por los datos contenidos en él.
- **Reduplicar**: Este término se refiere a un archivo que se vuelve a duplicar cuando estaba deduplicado, porque sus contenidos han cambiado.
- **Punto de montaje**: Es el punto del árbol de ficheros en el que se monta el sistema de ficheros dedupfs, y desde donde se interactúa con él. En este directorio no se almacena información, si no que sirve de lugar de acceso a los datos.
- **Directorio subyacente/raíz**: Corresponde con el subárbol de ficheros donde se almacenan realmente los datos del sistema de ficheros dedupfs. Si se accede a los datos a través de este directorio, en puesto de a través del punto de montaje, sólo se verán los marcadores vacíos, y no se podrá acceder a sus datos.

3. Descripción del sistema de ficheros

El diseño e implementación del sistema de ficheros *Dedupfs* se ha llevado a cabo teniendo en cuenta desde el primer momento cuáles eran los principales retos a tener en cuenta para lograr la deduplicación totalmente transparente al usuario.

- Es necesario conocer qué ficheros están deduplicados y cuáles no, y dónde se encuentran sus datos, para ello se utiliza una base de datos sqlite que se almacena en la dirección `.dedupfs/dedupfs.db`.
- Es necesario que los archivos deduplicados permanezcan independientes de cara al usuario del sistema de ficheros, y que los camios producidos en uno de ellos no se propaguen a otros. Además, sus permisos, tiempos de modificación, y otros atributos, deben permanecer independientes. Por este motivo se ha optado por dejar un marcador vacío en el lugar de cada archivo, donde se conservan sus permisos y otros atributos, y almacenar los datos en una subcarpeta del directorio `.dedupfs/data/` con el hash de los datos como nombre de archivo.
- Los enlaces físicos sirven para designar al mismo fichero con diferentes nombres desde el punto de vista del usuario. Es necesario tratarlo de la forma adecuada, y evita tratar enlaces físicos como si fueran archivos deduplicados. Para ello se ha creado una segunda tabla en la base de datos, que contabiliza estos enlaces y permite gestionar los archivos.
- Para evitar realizar el cálculo de hash cuando un archivo todavía puede ser modificado, lo más adecuado es hacerlo solamente cuando un fichero haya sido modificado, y deja de ser utilizado. Para ello se almacena en memoria un mapa de archivos abiertos para escritura, se marca si son modificados, y se computa su hash cuando los cierra el último proceso que los tenga abiertos.

Todo lo anterior se ha implementado modificando las funciones principales del ejemplo `bbfs.c`, copiándolas en otro llamado `dedupfs.c`, y añadiendo algunas nuevas.

También, se ha incluido un módulo llamado `database` que se encarga de gestionar toda la comunicación del módulo principal con las distintas tablas de la base de datos y con el mapa de ficheros abiertos (que también ha sido implementado como una base de datos sqlite en memoria).

Se ha mantenido y utilizado, por conveniencia, el módulo `log` para servir de ayuda en la depuración durante el desarrollo del programa. Aunque se ha definido un flag del preprocesador que anula el uso de este sistema de log una vez que el programa ha sido finalizado, puesto que el flujo de informaciónn generado por éste era demasiado alto.

Finalmente, se ha ampliado el fichero de cabecera `params.h` con la definición de algunas estructuras de datos, y la inclusión de cabeceras nuevas que han sido necesarias.

3.1. Estructuras de datos

Como se ha indicado en el apartado anterior, las estructuras de datos utilizadas en el proyecto se han basado en bases de datos sqlite3, tanto para el almacenamiento de la información de archivos deduplicados y sus hashes, como para el mapa de archivos abiertos en memoria. Incluyendo también los enlaces físicos.

- **Base de datos de archivos**: La base de datos de archivos está formada por dos tablas, y se almacena en `.dedupfs/dedupfs.db` dentro del directorio subyacente. La primera tabla, la tabla de archivos *(files)*, contiene una entrada por cada archivo de datos almacenado en nuestro SF. En dicha entrada se almacena la siguiente información:

  - Path: El path del archivo lógico. Que se dejará como marcador.
  - Shasum: El hash SHA1 de los datos del archivo, almacenado en texto con representación hexadecimal.
  - Datapath: El path donde se almacenan físicamente los datos del archivo en el SF.
  - Size: El tamaño en bytes del archivo, para acceder a esta información cómodamente, puesto que no está disponible en le marcador.
  - Deduplicados: La cuenta de veces que están deduplicados los datos del archivo, empieza en 0 para archivos no deduplicados.

  Con esta información se pueden realizar fácilmente todas las operaciones necesarias para deduplicar y reduplicar los archivos, y saber en qué estado está cada uno en cada momento.

  En esta base de datos de datos también se guarda la tabla de enlaces físicos, llamada *links*. En ella se guarda para cada enlace físico, cuál es el archivo que se debe buscar en la tabla de archivos para encontrar sus datos.

  - Linkpath: El path del enlace físico.
  - Originalpath: El path del archivo cuya entrada de la tabla de archivos posee toda la información.

- **Mapa de ficheros abiertos**: En esta estructura se guarda qué archivos han sido abiertos para escritura, de forma que se pueda saber cuando se cierran, si han sido modificados y si es el último descriptor siendo cerrado de ese archivo. Este mapa se ha establecido como una base de datos sqlite residente en memoria, ya que solo es necesario durante la ejecución del SF. Cada entrada de la tabla contiene la siguiente información:

  - fh: Descriptor de ficheros utilizado para abrir este archivo.
  - path: Path del archivo lógico que está abierto para escritura.
  - deduplicado: Si el archivo está deduplicado o no.
  - modificado: Indicador de si el archivo ha sido modificado o no. Un archivo puede abrirse para escritura, y finalmente no ser modificado.

Con estas tres tablas, dedupfs tiene toda la información necesaria de cada archivo para realizar cualquier operación que se le solicite.

En el fichero de cabecera `param.h` se han incluido dos *struct* que se corresponden con las entradas en la base de de datos de archivos, y en el mapa de ficheros abiertos en escritura.

3.2. Implementación de la funcionalidad

La funcionalidad completa del sistema de ficheros dedupfs dista bastante de ser trivial en muchas de las funciones que ha habido que reimplementar. Para explicar el funcionamiento de las nuevas funciones implementadas, vamos a seguir el flujo de ejeución, explicando cada una de las partes implicadas en él.

3.2.1. Apertura, lectura, escritura, y cierre de un fichero

Estas cuatro operaciones componen la funcionalidad fundamental del sistema de ficheros. Pasamos a explicar el comportamiento de dedupfs en estas operaciones.

Apertura: Cuando se produce la apertura de un archivo, dedupfs recibe una llamada a la función **bb_open** con el path del archivo, y los indicadores de apertura correspondientes. Lo primero que hace el SF en este caso, es comprobar que puede abrir el marcador. Si no hay ningún problema al acceder al marcador, entonces se abre el archivo de datos: Se mira si es un enlace duro, y de serlo, se obtiene el archivo al que apunta. Tras esto, se busca este archivo en la base de datos. En caso de estar en la bd, se coge el path de su archivo de datos. Si está deduplicado y se abre en escritura, es necesario abrir una copia, para evitar contaminar el archivo de datos deduplicado (se realiza esta copia si es el primer open de este archivo), esta copia se llama idéntica al archivo de datos, sólo que añadiendo una 'w' al final. Si no estuviera en la base de datos, simplemente se utilizará el marcador. Finalmente se abre el path que corresponda, y una vez abierto y si se hace en escritura, se introduce en el mapa de archivos abiertos para escritura.

La creación de un archivo nuevo puede verse como un caso particular de apertura en el que se abre un archivo nuevo para escribir en él. En este caso, cuando se recibe una llamada a **bb_create**, simplemente se abre el archivo y se introduce en el mapa de archivos abiertos.

Lectura y escritura: En el caso de la lectura de un fichero, no ha hecho falta realizar ningún cambio a la función **bb_read**, puesto que se trabaja con el descriptor de ficheros. Por otra parte, en la función **bb_write** sólo se ha hecho un cambio: cuando se escribe en un fichero, éste se establece como modificado en el mapa de ficheros abiertos.

Cierre: El cierre de un fichero que estaba abierto se lleva a cabo en la función **bb_release**. Ésta es una de las funciones más complejas del sistema de ficheros, ya que es aquí donde se ha implementado toda la lógica de la deduplicación de ficheros, al ser el cierre de un archivo modificado el mejor momento para recalcular su hash y comprobar si pasa a estar deduplicado, o si debe reduplicarse.
Lo primero que se hace en esta función, es cerrar el descriptor abierto y ejecutar *utime* sobre el marcador, para actualizar su tiempo de modificación. Tras esto, se comprueba que el fichero se encuentre en el mapa de archivos abiertos para escritura, haya sido modificado, y sea este descriptor el último que queda abierto. De no ser así, no habría nada más que hacer. Si lo anterior se cumple, el archivo se había modificado, y será necesario calcular su hash, comprobar si es un fichero enlazado físicamente, y si está en la base de datos. Nos encontramos con dos escenarios bien diferenciados:

- Si no está en la tabla de ficheros, significa que se trata de un archivo nuevo, o que anteriormente estaba vacío. Se calcula su hash y tamaño, y se mira si ya está ese hash en la BD. En caso positivo, sólo hay que introducir una nueva entrada en la base de datos, incrementar el contador de deduplicados en cada entrada con ese hash, y truncar el archivo original para dejarlo como marcador. En caso negativo, se trata del único archivo con ese hash, y es necesario moverlo al directorio interno donde se almacenan los archivos de datos. Para ello hace falta calcular y crear el subdirectorio donde se almacena el hash (de la forma `/.dedupfs/data/X/Y/hash` donde X e Y son el primer y segundo caracter del hash), moverlo ahí, y recrear el marcador en el lugar original, conservando los mismos permisos que tenía. Finalmente, se añade en la base de datos.

- Si está en la tabla de ficheros, significa que hemos modificado un archivo que ya existe. Se calcula su hash y tamaño. Si el nuevo tamaño es cero, se saca de la base de datos y deja sólo el marcador, en caso de que esté deduplicado, se decrementa el número de deuplicados para ese hash, si no lo está, se elimina el archivo de ese hash.
  Si el nuevo tamaño es mayor de cero, se comprueba si el nuevo hash y el antiguo de este archivo difieren. Si no difieren, no hay nada que hacer. Si lo hacen,debe insertarse en la base de datos con el nuevo hash y path de datos, y debe moverse a la localización del nuevo hash, a no ser que ese hash ya esté presente en el sistema, en cuyo caso se elimina este archivo y incrementa el número de deuplicados para el nuevo hash. Por último, se decrementa el número de deduplicados para el antiguo hash.
  Destacar que, puesto que si está deduplicado el archivo se está trabajando con una copia, no es necesario contemplar la conservación del archivo de datos en el lugar original. Si se elimina o se mueve y está deduplicado, se elimina o se mueve la copia, y si no está deduplicado, se actúa sobre el archivo de datos.

3.2.2. Atributos de fichero

Para obtener los atributos del fichero se utilizan las funciones **bb_getattr** y **bb_fgetattr**. La primera utiliza el path simplemente, y la segunda utiliza el descriptor de fichero de un

archivo abierto. En estas funciones ha sido necesario hacer un pequeño cambio, ya que el archivo descrito por el path es sólo un marcador en el sistema dedupfs. Este marcador almacena todos los atributos del fichero excepto el tamaño, ya que está vacío. Por lo tanto estas funciones deben comprobar si el fichero está en la base de datos, y si existe una entrada, tomar el tamaño de ahí y sustituir el que han obtenido del marcador.

3.2.3. Truncado de archivos

Para modificar el tamaño de un archivo se dispone de las funciones **bb_truncate** y **bb_ftruncate**. Al igual que en el caso anterior, la primera trabaja con el path de un archivo, mientras que la otra lo hace con el descriptor de fichero abierto. En este caso, ambas no funcionan igual. La función ftruncate que funciona con un descriptor de fichero abierto, puede verse como una modificación de un fichero abierto en escritura, por lo que simplemente se marca el fichero como modificado en el mapa de ficheros abiertos. En el caso de truncate, se utiliza el path, y se puede ver como una apertura de archivo, modificación de su tamaño, y cierre. Lo que se ha hecho en este caso, es utilizar las funciones de apertura, truncamiento y cierre ya implementadas, de forma que no hubiera que repetir el cálculo de su nuevo hash y el tratamiento adecuado.

3.2.4. Renombrado

El renombrado de un archivo mueve un archivo desde un punto del sistema de ficheros hasta otro diferente, sin afectar a sus datos. Esto lo hace la función **bb_rename**. Para lograr esta funcionalidad de forma completa en dedupfs, solamente ha sido necesario mover el marcador, y en caso de que esa operación sea exitosa, cambiar toda referencia al path anterior en la base de datos por el path nuevo.

3.2.5. Enlaces físicos y borrado

Los enlaces físicos se crean mediante llamadas a **bb_link**. En esta función simplemente ha sido necesario incluir una llamada a una función del módulo de la base de datos que añade una entrada en la tabla de enlaces duros.
El borrado de archivos, se lleva a cabo mediante **bb_unlink**. En este caso sí ha hecho falta tener en cuenta algunas cuestiones. Además de eliminar el marcador, hay que ver si está en la base de datos.Si está, puede ser que haya que eliminar su entrada, o, en caso de estar enlazado, deberá heredarla uno de los enlaces de la tabla de enlaces. Si se elimina su entrada, es necesario ver si estaba deduplicado, y decrementar el contador de deduplicados, o eliminar su archivo de datos si no lo estaba. Por último, en caso de no encontrarse en la tabla de ficheros, puede que esté en la de enlaces como enlace, y se elimina de ahí.

3.2.6. Inicialización y destrucción del SF

La biblioteca de FUSE proporciona dos funciones de inicialización y destrucción de las estructuras de datos necesarias, que son llamadas cuando se produce el arranque y parada del sistema de ficheros. Estas funciones son **bb_init** y **bb_destroy**. En la primera se lleva a cabo toda la inicialización del sistema de ficheros, de las tablas y bases de datos, y del mapa de ficheros abiertos. Se comprueba si el sistema se está montando por primera vez, y en caso de ser así, se crean las carpetas de datos, la base de datos, y se guarda un puntero en la estructura proporcionada por fuse para poder acceder a estas bases de datos más adelante. Después, cuando el sistema de ficheros se cierra, se recibe una llamada a la segunda función, que se encarga de cerrar las conexiones a las bases de datos.

3.2.7. Funciones auxiliares

Además de las funciones proporcionadas por FUSE, se han añadido algunas más para realizar ciertas tareas requeridas por una o varias operaciones. Se describen a continuación.

check_conf_dir: Esta función se utiliza previamente a la llamada a la función principal de FUSE, y se encarga de comprobar que el sistema de ficheros tenga acceso con los permisos adecuados, y si no existen los subdirectorios ocultos de dedupfs, los crea.

calcular_hash: Es la encargada de leer los datos de un fichero y calcular su hash utilizando la función SHA1. También registra el tamaño del fichero leído, y devuelve tanto el hash en forma de cadena de caracteres hexadecimales, como el tamaño del fichero cuyo hash ha sido calculado. Para el cálculo del hash, se basa en la biblioteca *openssl*

preprara_datapath: Esta función realiza dos tareas simultáneas: Por un lado, dado un hash, devuelve el path donde se tiene que almacenar dentro del directorio de datos; por otro lado, se encarga de asegurar que el subdirectorio en el que se va a almacenar el hash esté creado. El subdirectorio en el que se guarda un hash es de la forma `.dedupfs/data/x/y/` donde x e y son los dos primeros caracteres del hash a almacenar. Esto se hace así con el fin de evitar directorios enormes si hay muchos archivos.

copiar: Requiere poca explicación, se le proporcionan dos paths como parámetro, y copia el contenido del primero en el segundo. Es utilizada cuando se tiene que abrir una copia de un archivo deduplicado para un proceso que va a escribir en él.

3.2.8. Módulo de acceso a las BBDD

Como se ha explicado anteriormente, para gestionar los archivos deduplicados, enlaces físicos y archivos abiertos para escritura, se utilizan bases de datos sqlite3. Toda la funcionalidad relacionada con la apertura, consultas y cierre de estas BBDD se ha encapsulado en un módulo independiente compuesto por los archivos `database.c` y `database.h`. Dentro de este módulo se han creado las funciones que realizan toda la interacción con la base de datos.

Funciones de la base de datos: La base de datos principal de dedupfs es la encargada de almacenar la tabla de ficheros y sus hashes, y la de enlaces físicos. Las funciones que interactúan con esta base de datos comienzan todas con el prefijo *db_*. Las describimos a continuación:

- **db_open**: Abre la base de datos .dedupfs (creándola si no existía previamente) y crea las dos tablas necesarios si no estaban creadas previamente. Devuelve un puntero al objeto sqlite3 que el módulo principal guarda en los datos privados de FUSE, y que se utiliza posteriormente en el resto de consultas.
- **db_close**: Cierra la conexión a la base de datos creada en el paso anterior.
- **db_insertar**: Introduce una entrada en la base de datos, sustituyéndola por la anterior si existiera ya una con el mismo path.
- **db_get**: Recupera de la base de datos la entrada del argumento path, y la devuelve en el puntero que se le pasa como parámetro. Devuelve un 1 si ha encontrado la entrada, y un 0 si no.
- **db_get_datapath_hash**: Recibe un hash como argumento, y devuelve por parámetros el datapath de ese hash, así como la cuenta de deduplicados que tiene. Devuelve 1 si ha encontrado el hash, y 0 si no.
- **db_incrementar_duplicados**: Incrementa en 1 la cuenta de duplicados para cada entrada de la bd que coincida con ese hash.
- **db_decrementar_duplicados**: Realiza la operación inversa a la anterior.
- **db_eliminar**: Elimina la entrada que corresponde con path de la base de datos.
- **db_addLink**: Inserta una entrada en la tabla de enlaces físicos.
- **db_getLinkPath**: Dado un path, recupera de la tabla de enlaces el path donde se encuentra su información en la otra tabla.
- **db_unlink**: Elimina la entrada que corresponde con path de la tabla de enlaces.

- **db_link_get_heredero**: Cuando se elimina un archivo de la tabla de ficheros, puede que tuviera un enlace físico. Esta función busca en la tabla de enlaces si el path proporcionado tenía algún enlace, y en caso de ser así, lo devuelve para ser el heredero del original. Esta función devuelve 1 si se encuentra heredero, y 0 si no.
- **db_link_heredar**: Recibe un path, y el heredero por el que se debe sustituir, y sustituye toda ocurrencia del primero por el segundo en las dos tablas (elimina también su propia entrada de la tabla de enlaces si se ha cambiado, puesto que no tiene sentido apuntarse a sí mismo).
- **db_rename**: Se encarga de sustituir un path por otro en las tablas de la base de datos cuando se produce un renombrado de un archivo. Se diferencia de la función anterior en que ésta no puede producir el borrado de ninguna entrada.

Funciones del mapa de archivos abiertos: El mapa de archivos abiertos para escritura se gestiona como una base de datos en memoria, por lo que tiene también sus correspondientes funciones de acceso y consulta en este módulo:

- **map_open**: Se encarga de abrir la base de datos en memoria y devolver la referencia a la conexión con la BD
- **map_close**: Cierra la conexión abierta en la función anterior.
- **map_add**: Inserta una entrada en el mapa, correspondiente al fichero que ha sido abierto para escritura.
- **map_extract**: Extrae la entrada correspondiente al descriptor de ficheros proporcionado, se le proporciona un argumento entero *eliminar* que si está a true, indica que también debe eliminarse dicha entrada. La función devuelve por parámetro la entrada recuperada, y 1 si se ha encontrado dicha entrada, o 0 si no se ha encontrado.
- **map_count**: Dado un path, devuelve el número de veces que está abierto para escritura.
- **map_set_modificado**: Establece el fichero como modificado en la entrada correspondiente al argumento proporcionado.

4. Manual de uso

El uso del sistema de ficheros dedupfs es bastante sencillo. Los requisitos necesarios para poder compilar y ejecutar dedupfs son tener instalados los paquetes de desarrollo de fuse, openssl, y sqlite3. En ubuntu 14.04 esto se consigue instalando los paquetes `libssl-dev pkg-config libfuse2 libfuse-dev sqlite3 libsqlite3-dev`. Una vez hecho esto, sólo queda situarse dentro del directorio de dedupfs, y ejecutar el comando `make`, para generar el ejecutable.

Si todo ha ido bien, obtendremos un ejecutable `dedupfs` que nos permitirá montar nuestro sistema de ficheros. Para montarlo, solamente necesitamos un punto de montaje, que puede ser cualquier directorio vacío, y un directorio subyacente o raíz en el que se almacenarán los archivos de nuestro sistema de ficheros. Este directorio raíz debe estar vacío la primera vez que se utilice dedupfs. El comando para montar nuestro sistema de ficheros es:

```
./dedupfs -s directorio/raiz punto/de/montaje
```

Una vez que ya tenemos montado el sistema de ficheros, podemos trabajar tranquilamente sobre el punto de montaje y dedupfs se encargará de todo el trabajo para analizar y deduplicar los ficheros que escribamos en él. Cuando se ha terminado de trabajar con el sistema de ficheros, se puede desmontar mediante el comando `fusermount -uz punto/de/montaje` y los archivos dejarán de estar accesibles hasta que vuelva a montarse.

5. Pruebas del sistema de ficheros

5.1. Prueba de funcionamiento

Para demostrar el funcionamiento del sistema de ficheros dedupfs, vamos a realizar unas pruebas, escribiendo archivos en el sistema de ficheros, y realizando cambios en éstos que los convierten en copias de otros ya existentes, analizando enlaces físicos, rendimiento sobre permisos, etc.

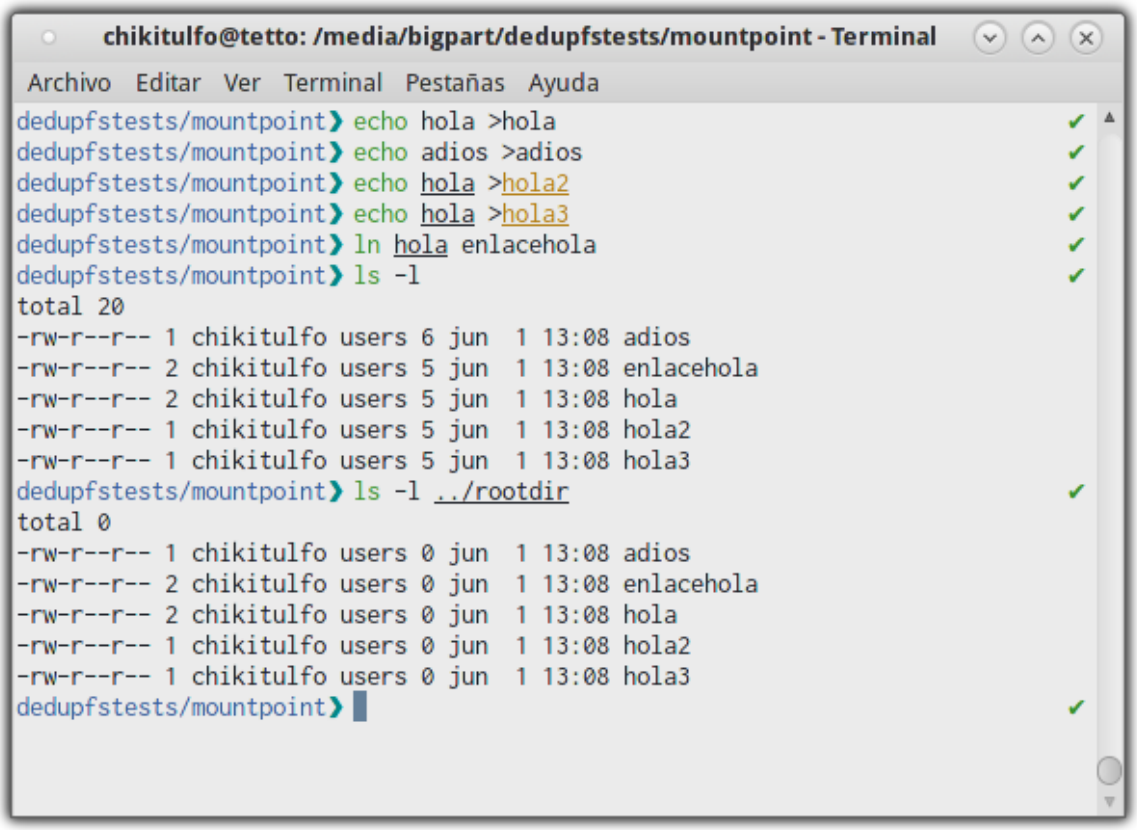

Figura 1: Deduplicado de archivos

En la figura 1 se han creado 3 archivos con el mismo contenido (“hola”), uno más diferente (“adios”), y un enlace a uno de esos archivos. Se puede observar el resultado de la ejecución de `ls -l` sobre el directorio. Vemos cómo todos los archivos son detectados como archivos independientes entre sí, a excepción de los que están físicamente enlazados, cuya cuenta de enlaces indica 2. Se puede ver también, que al ejecutar de nuevo textttls -l en el directorio raíz, el tamaño de todos estos archivos es 0, puesto que son meros marcadores, y su contenido se encuentra almacenado en un directorio oculto. Si el sistema de ficheros funciona adecuadamente, solamente debería haber quedado una copia del contenido del fichero hola, y una copia del contenido de adios.

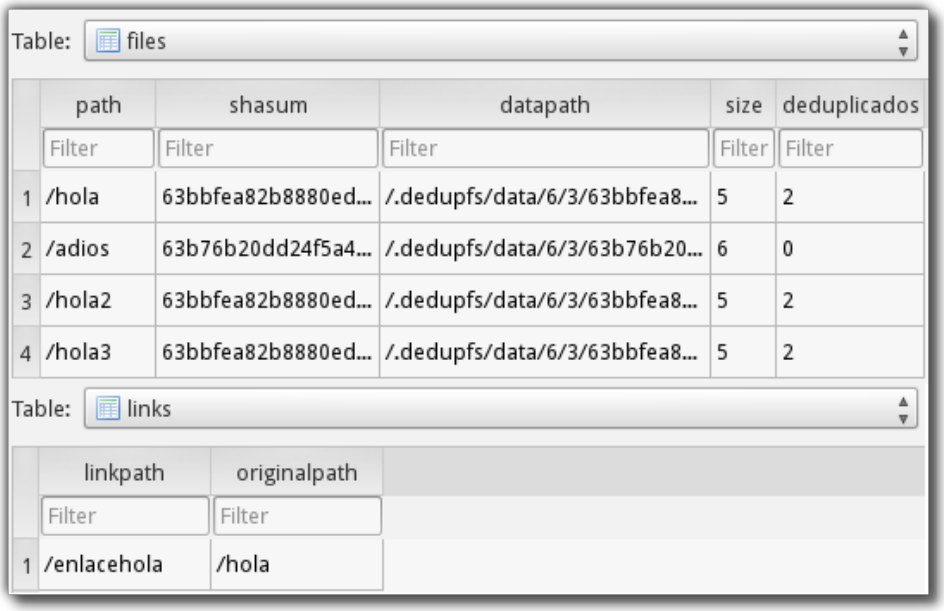

Table: files

	path	shasum	datapath	size	deduplicados
	Filter	Filter	Filter	Filter	Filter
1	/hola	63bbfea82b8880ed...	/.dedupfs/data/6/3/63bbfea8...	5	2
2	/adios	63b76b20dd24f5a4...	/.dedupfs/data/6/3/63b76b20...	6	0
3	/hola2	63bbfea82b8880ed...	/.dedupfs/data/6/3/63bbfea8...	5	2
4	/hola3	63bbfea82b8880ed...	/.dedupfs/data/6/3/63bbfea8...	5	2

Table: links

	linkpath	originalpath
	Filter	Filter
1	/enlacehola	/hola

Figura 2: Contenidos de la BD tras deduplicado

En la figura 2 tenemos una captura del contenido de la base de datos tras ejecutar los comandos anteriores. En ella podemos observar que en la tabla *files* tenemos una entrada por cada fichero que se ha creado. En aquellos que se han creado co el mismo contenido, se observa que el shasum es el mismo, y que la cuenta de deduplicados asciende a 2, ya que se han producido 2 deduplicaciones. También se ve como el enlace *enlacehola* no se encuentra en la tabla *files* si no en la de *links*, apuntando al archivo original, hola.

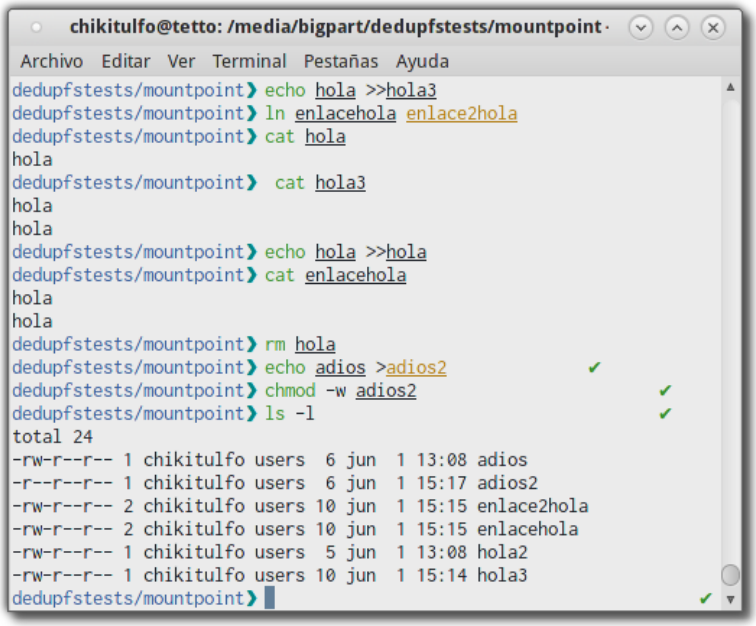

Figura 3: Captura de tests adicionales

En la figura 3 se ejecutan otra serie de comandos que permiten comprobar la evolución del sistema de ficheros al deduplicar y reduplicar más archivos. En este caso vemos que se añade otra palabra al archivo *hola3*, lo que causa que se reduplique, se crea otro enlace a partir de enlacehola, y se comprueba que los contenidos de *hola* y *hola3* han cambiado. Se escribe en *hola* otra palabra, dando lugar a una nueva deduplicación de estos dos últimos archivos. Después se lee el archivo *enlacehola*, que al ser un enlace físico a *hola* ve los mismos contenidos que éste. Después se borra el fichero hola, con lo que uno de sus enlaces deberá heredar sus entradas de las tablas de archivos y enlaces físicos. Se crea un nuevo archivo *adios2* que se deduplica con los contenidos de *adios*, y se le retira el permiso de escritura a ese archivo. Por último se ejecuta un listado de los archivos del directorio, donde se pueden observar los cambios acontecidos.

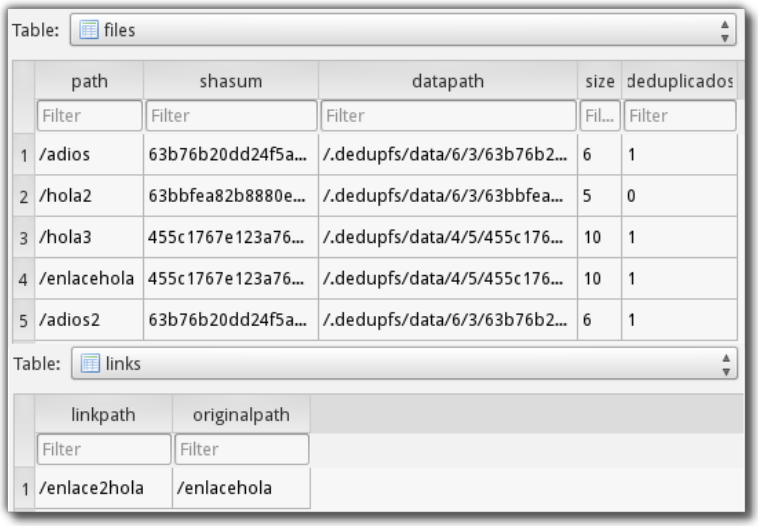
Table: files

	path	shasum	datapath	size	deduplicados
	Filter	Filter	Filter	Fil...	Filter
1	/adios	63b76b20dd24f5a...	/.dedupfs/data/6/3/63b76b2...	6	1
2	/hola2	63bbfea82b8880e...	/.dedupfs/data/6/3/63bbfea...	5	0
3	/hola3	455c1767e123a76...	/.dedupfs/data/4/5/455c176...	10	1
4	/enlacehola	455c1767e123a76...	/.dedupfs/data/4/5/455c176...	10	1
5	/adios2	63b76b20dd24f5a...	/.dedupfs/data/6/3/63b76b2...	6	1

Table: links

	linkpath	originalpath
	Filter	Filter
1	/enlace2hola	/enlacehola

Figura 4: Aspecto de la BD tras los tests en figura 3

En la figura 4 se nos muestra el estado de la base de datos tras los cambios descritos en el párrafo anterior. Se puede ver que *adios* y *adios2* están ahora deduplicados, que *hola* ya no está, y en su lugar podemos encontrar uno de sus enlaces: *enlacehola*, el cual se encuentra deduplicado con *hola3*. Se observa también que *hola2*, que no ha sido modificado desde el ejemplo anterior, sigue ocupando 5 bytes y ya no está deduplicado. Por último, en la tabla de enlaces vemos que la entrada de *enlacehola* ya no está, al haber heredado éste el nombre de *hola*, y que *enlace2hola* tiene una entrada apuntando al primer enlace que se creó.

Los ejemplos anteriores, aunque no son muy extensos, permiten observar toda la funcionalidad que proporciona el sistema de ficheros dedupfs. Y que éste es capaz que ofrecer una visión transparente de los ficheros al usuario, mientras que éstos se almacenan ocupando un menor espacio.

5.2. Estudio de rendimiento

Como se dijo en la introducción, el sistema de ficheros dedupfs permite ahorrar bastante espacio, pero es importante conocer cuáles son sus puntos fuertes y cuáles sus puntos débiles. Este sistema de ficheros se apoya en dejar ficheros vacíos como marcadores, y en usar una base de datos para almacenar la clave de dispersión de cada archivo y el lugar donde están almacenados los datos. El uso adicional de espacio para estos metadatos no es muy alto, pero puede llegar a ocasionar una mayor ocupación de espacio en disco en algunos escenarios.

En un escenario donde haya muchos archivos de tamaño relativamente pequeño, y no haya casi duplicados, este sistema de ficheros no conseguirá mejora de uso de espacio, e incluso puede llegar a ocupar más espacio que uno sin deduplicación. En un escenario más heterogéneo donde se encuentran archivos de diversos tamaños y existen deduplicados, dedupfs probablemente consiga una mejora de ocupación de espacio, aunque sea modesta. Aunque sin duda el caso más favorable para nuestro sistema de ficheros es aquel donde exista un buen número de archivos duplicados, como por ejemplo un directorio donde se almacenan copias de seguridad, etc.

Para obtener una medida real de las diferencias de rendimiento del sistema de ficheros, se ha realizado un estudio con tres escenarios diferentes con los que comparar el uso de espacio en el sistema de ficheros entre dedupfs, y un sistema de ficheros sin deduplicación (*ext4* en este caso):

- **Código fuente del kernel linux**: El primer escenario es el código fuente completo del núcleo de linux 4.0.4. Este caso parece a priori el más desfavorable para dedupfs. Es un directorio con 48.948 archivos, que ocupa 648MiB en su versión sin deduplicación, y donde cabe esperar que el número de archivos duplicados no sea muy alto.
- **Subdirectorio del usuario**: El segundo escenario es el subdirectorio del usuario (`/home`). En este directorio existen 16.964 archivos, con un tamaño total de 938MiB. En él se encuentran muchos archivos de configuración, de pequeño tamaño, así como archivos multimedia y de datos de mayor tamaño. Desconocemos a priori cuál será el número de duplicados.
- **Directorio de copias de seguridad**: El último escenario es el que parece más favorable a dedupfs. Se trata de un directorio de copias de seguridad donde se encuentran 5 copias de seguridad de un directorio de datos de un servidor de juegos. En este directorio, aproximadamente el 75 % de los archivos es de pequeño tamaño (menores a 1KiB) pero también existen otros de tamaño considerablemente mayor. Al tratarse de 5 copias, es de esperar que haya un buen número de archivos duplicados, aunque los datos almacenados en él son bastante cambiantes, y podría no ser así. Las 5 copias suman 1374 archivos para un total de 1056MiB sin deduplicación.

Para llevar a cabo la comparación, se ha contabilizado el número de archivos existentes y tamaño en disco en la versión sin deduplicar, frente al número de archivos total y tamaño en

disco del directorio raíz en dedupfs (esto contabiliza como ficheros tanto los marcadores como los de datos). Además, se ha observado para cada caso qué porcentaje de las entradas totales de la base de datos está deduplicada al menos una vez.

	No dedup		Dedupfs			
	Archivos	MiB	Archivos	MiB	Duplicados	Mejora
Núcleo Linux	48948	648	97609	660	0,6 %	-1,85 %
Directorio /home	16964	938	32876	925	3,4 %	1,38 %
Copias seguridad	1374	1056	1908	753	70 %	28,7 %

Los resultados obtenidos tras estudiar el rendimiento del sistema de ficheros se aproximan bastante a lo que cabía esperar. En el escenario del código fuente del kernel linux, el sistema dedupfs necesita más espacio que la versión sin deduplicación. Esto es debido a la gran cantidad de ficheros que se crean como marcador, y a que existen muy pocos deduplicados.

En el escenario del directorio del usuario, se observa una mejora casi despreciable, puesto que, de un modo parecido al primer escenario, existe una cantidad muy alta de ficheros pequeños, y esta provoca que se generen muchos marcadores sin obtener casi mejora. Pese a todo, y aunque hay una cantidad muy pequeña de archivos duplicados, la mejora obtenida al deduplicar estos archivos, proporciona un ahorro de espacio suficiente para solventar el anterior problema.

El último escenario ha sido el más favorable, como esperábamos. Con un ahorro de espacio de un 28,7 %, se trata de un éxito absoluto para dedupfs. Al tratarse de copias de seguridad, había una gran cantidad de archivos duplicados. De un total de 1354 entradas en la base de datos, el 70 % estaban deduplicados, quedando 533 archivos de datos diferentes que contenían información única. Sin duda merece la pena utilizar sistemas de ficheros de este tipo en escenarios donde hay archivos duplicados.

Si equilibramos los tres escenarios estudiados, podemos observar que aunque en uno de ellos dedupfs tiene peor desempeño, con la mejora que se consigue en los otros dos se compensa sobradamente, y utilizar este sistema de ficheros en un sistema que aglutine a los tres tipos de escenarios sería ventajoso en cuanto al ahorro de espacio.

6. Anexo: Listado del código fuente

6.1. dedupfs.c

```
1
#include "params.h"

#include <fuse.h>
#include <ctype.h>
6 #include <dirent.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
11 #include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/xattr.h>
#include <openssl/evp.h>
16

#include "log.h"
#include "database.h"

21

// Report errors to logfile and give -errno to caller
static int bb_error(char *str)
{
  int ret = -errno;
26

  log_msg("    ERROR %s: %s\n", str, strerror(errno));

  return ret;
}
31

/**
 * Calcula el sha1sum de path y lo devuelve como cadena en outHashString
 */
static void calcular_hash(const char * path, char * outHashString,
   unsigned int * filesize){
36  ssize_t leidos;
  unsigned int sizedigest, i;
  int fd;
  unsigned char hash[20];
  char buffer[4096], *outptr;
41

  if ( (fd = open(path,O_RDONLY)) == -1){
    bb_error("[calcular_hash]open");
  }
  *filesize = 0;
```

```
46      EVP_MD_CTX *context = EVP_MD_CTX_create();
        EVP_DigestInit_ex(context, EVP_sha1(), NULL);
        //Leer desde el principio
        if(lseek(fd, 0, SEEK_SET)<0){
          bb_error("[calcular_hash]lseek");
51      }
        leidos = read(fd,buffer,4096);
        if(leidos<0){
          bb_error("[calcular_hash]read");
        }
56      else{
          *filesize += leidos;
          EVP_DigestUpdate(context,buffer,leidos);
          while (leidos == 4096){
            leidos = read(fd,buffer,4096);
61          *filesize += leidos;
            EVP_DigestUpdate(context,buffer,leidos);
          }
        }
        close(fd);
66      EVP_DigestFinal_ex(context,hash,&sizedigest);
        EVP_MD_CTX_destroy(context);

        //Introducirlo en la cadena outHashString
        outptr = outHashString;
71      for(i=0; i<sizedigest; i++){
          //log_msg("%02x", hash[i]);
          sprintf(outptr,"%02x", hash[i]);
          outptr+=2;
        }
76      outptr[0]=0; //Terminar la cadena en 0
      }

      /**
       * Función encargada de asegurar que los directorios necesarios para
          guardar
81    * los datos en el directorio de datos están creados.
       * El path donde se almacenen será .dedupfs/data/X/Y/HASH donde X e Y son
       * los dos primeros caracteres del hash.
       * Devuelve en datapath la ruta donde se almacenan los datos de ese hash.
       * (Relativa a la raiz del sf fuse)
86    */
      static void preparar_datapath(char * hash, char * datapath) {
        int rv;
        log_msg("    preparar_datapath(%s)\n",hash);

91      sprintf(datapath, "%s/.dedupfs/data/%c/", BB_DATA->rootdir, hash[0]);
        rv = mkdir(datapath, 0777);   //Intentamos crear X, y asumimos el error
           EEXIST
        if(rv != 0 && errno != EEXIST)
          bb_error("mkdir: ");
        sprintf(datapath, "%s%c/", datapath, hash[1]);
96      rv = mkdir(datapath, 0777);   //Intentamos crear Y, y asumimos el error
```

```
      EEXIST
    if(rv != 0 && errno != EEXIST)
      bb_error("mkdir: ");
    //Devolver sin rootdir
    sprintf(datapath, "/.dedupfs/data/%c/%c/%s", hash[0], hash[1], hash);
}

/**
 * Copia un archivo en otro.
 * Utilizado en las copias de escritura de archivos deduplicados
 */
static void copiar ( const char * src, const char * dst){
  unsigned char buffer[4096];
  int fdsrc, fddst, leidos, escritos;
  log_msg("    copiar( %s ;\n"
      "            %s\n", src, dst);
  if ( (fdsrc = open(src,O_RDONLY)) == -1){
    bb_error("[copiar]open src");
  }
  else if ((fddst = creat(dst,0777)) == -1){
    bb_error("[copiar]create dst");
  }
  else {
    leidos = read(fdsrc,buffer,4096);
    escritos = write(fddst,buffer,leidos);
    if(leidos<0 || escritos<0){
      bb_error("[copiar]read/write");}
    else{
      while (leidos == 4096){
        leidos = read(fdsrc,buffer,4096);
        escritos = write(fddst,buffer,leidos);
        if(leidos<0 || escritos<0){
          bb_error("[copiar]read/write");}
      }
    }
  }
  close(fdsrc);
  close(fddst);
}

//  All the paths I see are relative to the root of the mounted
//  filesystem.  In order to get to the underlying filesystem, I need to
//  have the mountpoint.  I'll save it away early on in main(), and then
//  whenever I need a path for something I'll call this to construct
//  it.
static void bb_fullpath(char fpath[PATH_MAX], const char *path)
{
  strcpy(fpath, BB_DATA->rootdir);
  strncat(fpath, path, PATH_MAX); // ridiculously long paths will
    // break here

  log_msg("    bb_fullpath:  rootdir = \"%s\", path = \"%s\", fpath = \"%s
      \"\n",
```

```
    BB_DATA->rootdir, path, fpath);
}

/** Get file attributes.
 *
 * Similar to stat().  The 'st_dev' and 'st_blksize' fields are
 * ignored.  The 'st_ino' field is ignored except if the 'use_ino'
 * mount option is given.
 */
int bb_getattr(const char *path, struct stat *statbuf)
{
  int retstat = 0;
  char fpath[PATH_MAX];
  struct db_entry entradabd;

  log_msg("\nbb_getattr(path=\"%s\", statbuf=0x%08x)\n",
  path, statbuf);
  bb_fullpath(fpath, path);

  retstat = lstat(fpath, statbuf);
  if (retstat != 0)
    retstat = bb_error("bb_getattr lstat");

  // Desreferenciamos path si es un enlace duro
  if(db_getLinkPath(path, fpath)) {
    //Si es true se ha desreferenciado y guardado en fpath.
    //Cambiamos el puntero de path a fpath
    path=fpath;
  }
  //Si el archivo está en la bd, el tamaño se toma de ahí
  if( db_get(path,&entradabd)){
    statbuf->st_size = entradabd.size;
    //Necesario incluir st_blocks? Lo incluimos
    //damos por supuesto tamaño de bloque de 4096
    statbuf->st_blocks = (entradabd.size / 4096 + (entradabd.size % 4096 >
        0))*8;
  }
  log_stat(statbuf);

  return retstat;
}

/** Read the target of a symbolic link
 *
 * The buffer should be filled with a null terminated string.  The
 * buffer size argument includes the space for the terminating
 * null character.  If the linkname is too long to fit in the
 * buffer, it should be truncated.  The return value should be 0
 * for success.
 */
// Note the system readlink() will truncate and lose the terminating
// null.  So, the size passed to to the system readlink() must be one
// less than the size passed to bb_readlink()
```

```
  // bb_readlink() code by Bernardo F Costa (thanks!)
201 int bb_readlink(const char *path, char *link, size_t size)
  {
    int retstat = 0;
    char fpath[PATH_MAX];

206   log_msg("bb_readlink(path=\"%s\", link=\"%s\", size=%d)\n",
        path, link, size);
    bb_fullpath(fpath, path);

    retstat = readlink(fpath, link, size - 1);
211   log_msg("    link: %s", link);
    if (retstat < 0)
    retstat = bb_error("bb_readlink readlink");
    else  {
    link[retstat] = '\0';
216 retstat = 0;
    }

    return retstat;
  }
221
  /** Create a file node
   *
   * There is no create() operation, mknod() will be called for
   * creation of all non-directory, non-symlink nodes.
226 */
  // shouldn't that comment be "if" there is no.... ?
  int bb_mknod(const char *path, mode_t mode, dev_t dev)
  {
    int retstat = 0;
231 char fpath[PATH_MAX];

    log_msg("\nbb_mknod(path=\"%s\", mode=0%3o, dev=%lld)\n",
      path, mode, dev);
    bb_fullpath(fpath, path);
236
    // On Linux this could just be 'mknod(path, mode, rdev)' but this
    //  is more portable
    if (S_ISREG(mode)) {
      retstat = open(fpath, O_CREAT | O_EXCL | O_WRONLY, mode);
241 if (retstat < 0)
      retstat = bb_error("bb_mknod open");
      else {
      retstat = close(retstat);
      if (retstat < 0)
246   retstat = bb_error("bb_mknod close");
    }
    } else
    if (S_ISFIFO(mode)) {
      retstat = mkfifo(fpath, mode);
251   if (retstat < 0)
      retstat = bb_error("bb_mknod mkfifo");
```

```
    } else {
      retstat = mknod(fpath, mode, dev);
      if (retstat < 0)
      retstat = bb_error("bb_mknod mknod");
    }

    return retstat;
}

/** Create a directory */
int bb_mkdir(const char *path, mode_t mode)
{
    int retstat = 0;
    char fpath[PATH_MAX];

    log_msg("\nbb_mkdir(path=\"%s\", mode=0%3o)\n",
      path, mode);
    bb_fullpath(fpath, path);

    retstat = mkdir(fpath, mode);
    if (retstat < 0)
    retstat = bb_error("bb_mkdir mkdir");

    return retstat;
}

/** Remove a directory */
int bb_rmdir(const char *path)
{
    int retstat = 0;
    char fpath[PATH_MAX];

    log_msg("bb_rmdir(path=\"%s\")\n",
      path);
    bb_fullpath(fpath, path);

    retstat = rmdir(fpath);
    if (retstat < 0)
    retstat = bb_error("bb_rmdir rmdir");

    return retstat;
}

/** Create a symbolic link */
// The parameters here are a little bit confusing, but do correspond
// to the symlink() system call.  The 'path' is where the link points,
// while the 'link' is the link itself.  So we need to leave the path
// unaltered, but insert the link into the mounted directory.
int bb_symlink(const char *path, const char *link)
{
    int retstat = 0;
    char flink[PATH_MAX];
```

```
306     log_msg("\nbb_symlink(path=\"%s\", link=\"%s\")\n",
          path, link);
        bb_fullpath(flink, link);

        retstat = symlink(path, flink);
311     if (retstat < 0)
        retstat = bb_error("bb_symlink symlink");

        return retstat;
      }
316
      /** Rename a file */
      // both path and newpath are fs-relative
      int bb_rename(const char *path, const char *newpath)
      {
321     int retstat = 0;
        char fpath[PATH_MAX];
        char fnewpath[PATH_MAX];

        log_msg("\nbb_rename(fpath=\"%s\", newpath=\"%s\")\n",
326       path, newpath);
        bb_fullpath(fpath, path);
        bb_fullpath(fnewpath, newpath);

        retstat = rename(fpath, fnewpath);
331     if (retstat < 0)
          retstat = bb_error("bb_rename rename");
        else {
          //Si se renombra con éxito, cambiar en la base de datos
          db_rename(path, newpath);
336     }
        return retstat;
      }

      /** Create a hard link to a file */
341   int bb_link(const char *path, const char *newpath)
      {
        int retstat = 0;
        char fpath[PATH_MAX], fnewpath[PATH_MAX];

346     log_msg("\nbb_link(path=\"%s\", newpath=\"%s\")\n",
          path, newpath);
        bb_fullpath(fpath, path);
        bb_fullpath(fnewpath, newpath);

351     retstat = link(fpath, fnewpath);
        if (retstat < 0)
          retstat = bb_error("bb_link link");
        else // Se inserta en la tabla de enlaces
          db_addLink(path, newpath);
356
        return retstat;
      }
```

```
/** Remove a file */
int bb_unlink(const char *path)
{
  int retstat = 0;
  char fpath[PATH_MAX];


  log_msg("bb_unlink(path=\"%s\")\n",
    path);

  bb_fullpath(fpath, path);

  retstat = unlink(fpath);
  if (retstat < 0){
    retstat = bb_error("bb_unlink unlink");
  }
  else {
    struct db_entry entradadb;
    if (db_get(path, &entradadb)){
      char * heredero = malloc(PATH_MAX);
      // Si está en la tabla de enlaces, su entrada de la tabla de enlaces
      //  debe heredarla otro, y no se modifican los datos
      if (db_link_get_heredero(path, heredero)){
        db_link_heredar(path, heredero);
      } else {
        db_eliminar(path);
        // Si no está deduplicado se eliminan los datos
        if (entradadb.deduplicados == 0) {
          bb_fullpath(fpath,entradadb.datapath);
          unlink(fpath);
        } else { // Si está deduplicado se decrementa el contador
          db_decrementar_duplicados(entradadb.sha1sum);
        }
      }
      free(heredero);
    } else {
      //Si no está en la tabla de datos, se elimina de la de enlaces
      // puesto que puede estar como enlace
      db_unlink(path);
    }
  }

  return retstat;
}

/** Change the permission bits of a file */
int bb_chmod(const char *path, mode_t mode)
{
  int retstat = 0;
  char fpath[PATH_MAX];

  log_msg("\nbb_chmod(fpath=\"%s\", mode=0%03o)\n",
```

```
    path, mode);
  bb_fullpath(fpath, path);

  retstat = chmod(fpath, mode);
  if (retstat < 0)
  retstat = bb_error("bb_chmod chmod");

  return retstat;
}

/** Change the owner and group of a file */
int bb_chown(const char *path, uid_t uid, gid_t gid)

{
  int retstat = 0;
  char fpath[PATH_MAX];

  log_msg("\nbb_chown(path=\"%s\", uid=%d, gid=%d)\n",
    path, uid, gid);
  bb_fullpath(fpath, path);

  retstat = chown(fpath, uid, gid);
  if (retstat < 0)
  retstat = bb_error("bb_chown chown");

  return retstat;
}

/** Change the access and/or modification times of a file */
/* note -- I'll want to change this as soon as 2.6 is in debian testing */
int bb_utime(const char *path, struct utimbuf *ubuf)
{
  int retstat = 0;
  char fpath[PATH_MAX];

  log_msg("\nbb_utime(path=\"%s\", ubuf=0x%08x)\n",
    path, ubuf);
  bb_fullpath(fpath, path);

  retstat = utime(fpath, ubuf);
  if (retstat < 0)
    retstat = bb_error("bb_utime utime");

  return retstat;
}

/** File open operation
 *
 * No creation, or truncation flags (O_CREAT, O_EXCL, O_TRUNC)
 * will be passed to open().  Open should check if the operation
 * is permitted for the given flags.  Optionally open may also
 * return an arbitrary filehandle in the fuse_file_info structure,
 * which will be passed to all file operations.
```

```
 *
 * Changed in version 2.2
 */
int bb_open(const char *path, struct fuse_file_info *fi)
{
  int fd;
  char fpath[PATH_MAX], dereferecedp[PATH_MAX];
  struct db_entry entradadb;
  int retstat = 0;
  char escritura = 0;

  log_msg("\nbb_open(path\"%s\", fi=0x%08x)\n",
    path, fi);

  //Comprobamos si se puede abrir el marcador
  bb_fullpath(fpath,path);
  fd=open(fpath, fi->flags);
  if( fd == -1){
    //No se abre.
    retstat = bb_error("bb_open open");
    return retstat;
  } else {
    //Se ha abierto adecuadamente, continuamos
    close(fd);
    fd=0;
    retstat=0;
  }

  if ((fi->flags&O_RDWR) == O_RDWR || (fi->flags&O_WRONLY) == O_WRONLY){
    escritura = 1;
  }

  //Obtener auténtico path de datos si es un enlace.
  if ( db_getLinkPath(path, dereferecedp)){
    //Se ha obtenido el path original.
    path = dereferecedp;
  }

  if (db_get(path, &entradadb)) {
    bb_fullpath(fpath, entradadb.datapath);
    //Si está deduplicado y va a escribirse, hay que usar una copia.
    if(escritura && entradadb.deduplicados > 0) {
      //Se hace la copia si es el primero en abrirse
      if (map_count(path) == 0){
        char * oldfpath = strdup(fpath);
        strcat(fpath, "w"); //Las copias de trabajo tienen una "w" al
           final
        copiar(oldfpath, fpath);
        free(oldfpath);
      } else {
        strcat(fpath, "w");
      }
    }
```

```
    }
    else {
      bb_fullpath(fpath, path);
      entradadb.deduplicados = 0; //para incluirlo en el mapa
    }
    log_msg("    open(%s)\n", fpath);
    fd = open(fpath, fi->flags);
    if (fd < 0)
    retstat = bb_error("bb_open open");

    fi->fh = fd;
    log_fi(fi);
    // Si se abre para escritura, añadir al mapa
    if (escritura){
      map_add(fi->fh, path, entradadb.deduplicados, 0);
    }

    return retstat;
}

/** Read data from an open file
 *
 * Read should return exactly the number of bytes requested except
 * on EOF or error, otherwise the rest of the data will be
 * substituted with zeroes.  An exception to this is when the
 * 'direct_io' mount option is specified, in which case the return
 * value of the read system call will reflect the return value of
 * this operation.
 *
 * Changed in version 2.2
 */
// I don't fully understand the documentation above -- it doesn't
// match the documentation for the read() system call which says it
// can return with anything up to the amount of data requested. nor
// with the fusexmp code which returns the amount of data also
// returned by read.
int bb_read(const char *path, char *buf, size_t size, off_t offset, struct
    fuse_file_info *fi)
{
    int retstat = 0;

    log_msg("\nbb_read(path=\"%s\", buf=0x%08x, size=%d, offset=%lld, fi=0x
       %08x)\n",
      path, buf, size, offset, fi);
    // no need to get fpath on this one, since I work from fi->fh not the
       path
    log_fi(fi);

    retstat = pread(fi->fh, buf, size, offset);
    if (retstat < 0)
    retstat = bb_error("bb_read read");

    return retstat;
```

```
}

/** Write data to an open file
 *
 * Write should return exactly the number of bytes requested
 * except on error.  An exception to this is when the 'direct_io'
 * mount option is specified (see read operation).
 *
 * Changed in version 2.2
 */
// As  with read(), the documentation above is inconsistent with the
// documentation for the write() system call.
int bb_write(const char *path, const char *buf, size_t size, off_t offset,
     struct fuse_file_info *fi)
{
  int retstat = 0;

  log_msg("\nbb_write(path=\"%s\", buf=0x%08x, size=%d, offset=%lld, fi=0x
     %08x)\n",
    path, buf, size, offset, fi
    );
  // no need to get fpath on this one, since I work from fi->fh not the
     path
  log_fi(fi);

  retstat = pwrite(fi->fh, buf, size, offset);
  if (retstat < 0)
  retstat = bb_error("bb_write pwrite");

  // Marcamos el archivo como modificado
  if(size >0){
    map_set_modificado(fi->fh);
  }

  return retstat;
}

/** Get file system statistics
 *
 * The 'f_frsize', 'f_favail', 'f_fsid' and 'f_flag' fields are ignored
 *
 * Replaced 'struct statfs' parameter with 'struct statvfs' in
 * version 2.5
 */
int bb_statfs(const char *path, struct statvfs *statv)
{
  int retstat = 0;
  char fpath[PATH_MAX];

  log_msg("\nbb_statfs(path=\"%s\", statv=0x%08x)\n",
    path, statv);
  bb_fullpath(fpath, path);
```

```
    // get stats for underlying filesystem
    retstat = statvfs(fpath, statv);
    if (retstat < 0)
    retstat = bb_error("bb_statfs statvfs");

    log_statvfs(statv);

    return retstat;
}

/** Possibly flush cached data
 *
 * BIG NOTE: This is not equivalent to fsync().  It's not a
 * request to sync dirty data.
 *
 * Flush is called on each close() of a file descriptor.  So if a
 * filesystem wants to return write errors in close() and the file
 * has cached dirty data, this is a good place to write back data
 * and return any errors.  Since many applications ignore close()
 * errors this is not always useful.
 *
 * NOTE: The flush() method may be called more than once for each
 * open().  This happens if more than one file descriptor refers
 * to an opened file due to dup(), dup2() or fork() calls.  It is
 * not possible to determine if a flush is final, so each flush
 * should be treated equally.  Multiple write-flush sequences are
 * relatively rare, so this shouldn't be a problem.
 *
 * Filesystems shouldn't assume that flush will always be called
 * after some writes, or that if will be called at all.
 *
 * Changed in version 2.2
 */
int bb_flush(const char *path, struct fuse_file_info *fi)
{
    int retstat = 0;

    log_msg("\nbb_flush(path=\"%s\", fi=0x%08x)\n", path, fi);
    // no need to get fpath on this one, since I work from fi->fh not the
       path
    log_fi(fi);

    return retstat;
}

/** Release an open file
 *
 * Release is called when there are no more references to an open
 * file: all file descriptors are closed and all memory mappings
 * are unmapped.
 *
 * For every open() call there will be exactly one release() call
 * with the same flags and file descriptor.  It is possible to
```

```
 * have a file opened more than once, in which case only the last
 * release will mean, that no more reads/writes will happen on the
 * file.  The return value of release is ignored.
 *
 * Changed in version 2.2
 */
int bb_release(const char *path, struct fuse_file_info *fi)
{
  int retstat = 0;
  struct map_entry entradamap;
  char nuevohash[41];
  log_msg("\nbb_release(path=\"%s\", fi=0x%08x)\n",
  path, fi);
  log_fi(fi);
  // Cerramos el archivo, y después vemos.
  retstat = close(fi->fh);
  //Establecemos el tiempo de modificación en el marcador
  char *fpath = malloc(PATH_MAX);
  bb_fullpath(fpath, path);
  utime(fpath, NULL);
  free(fpath);
  // Obtenemos la información del mapa de ficheros abiertos en escritura
  if (map_extract(fi->fh, &entradamap, 1)  //Si está en el mapa
      && entradamap.modificado     //y ha sido modificado
      && map_count(entradamap.path)<1){//y era el último abierto
    unsigned int size;
    struct db_entry entradadb;
    char truepath[PATH_MAX];
    char truedatapath[PATH_MAX];
    char dereferencedpath[PATH_MAX];

    log_msg("    Abierto en escritura, modificado y último.\n");

    //Si es un archivo enlazado, path contiene la direccion del enlace,
    // mientras que la tabla de deduplicados está guardada con el path
    // del que se enlaza. Hay que cambiar eso.

    if ( db_getLinkPath(path, dereferencedpath)){
      //Se ha obtenido el path original.
      path = dereferencedpath;
    }

    //Si se encuentra en la base de datos, el archivo no es nuevo
    if( db_get(path,&entradadb)) {
      //El archivo no es nuevo, se ha modificado
      if(entradamap.deduplicados>0) {
        // Si estaba deduplicado, es una copia para escritura.
        bb_fullpath(truedatapath, strcat(entradadb.datapath,"w"));
      } else {
        bb_fullpath(truedatapath, entradadb.datapath);
      }
      //Calculamos el hash
      calcular_hash(truedatapath, nuevohash, &size);
```

```
        log_msg("    NewHash: %s\n", nuevohash);
        log_msg("    OldHash: %s\n", entradadb.sha1sum);
        // Había una entrada en la bd. Comparar hashes y tamaño
        if (size == 0){
          // Si el tamaño es 0, se saca de la bd y punto.
          db_eliminar(path);
          if(entradadb.deduplicados==0){
            // Borrar los datos también, es el último.
            unlink(truedatapath);
          } else {
            // No es el último, decrementar
            db_decrementar_duplicados(entradadb.sha1sum);
          }
        }
        else if (strcmp(entradadb.sha1sum,nuevohash)) {
          // Si el tamaño no es 0, y los hashes son diferentes
          if (db_get_datapath_hash(nuevohash, entradadb.datapath,&(entradadb
              .deduplicados))){
            // El nuevo hash ya está presente, eliminar
            unlink(truedatapath);
            db_insertar(path, nuevohash, entradadb.datapath, size, entradadb
                .deduplicados);
            db_incrementar_duplicados(nuevohash);
          }else {
            //Es el primero, mover(si está deduplicado estamos moviendo la
                copia)
            preparar_datapath(nuevohash, entradadb.datapath);
            char * newdatapath = truepath; //newdatapath apunta a truepath
                pero se llama diferente por claridad
            bb_fullpath(newdatapath, entradadb.datapath);
            if (rename(truedatapath,newdatapath)) {
              log_msg("    rename(%s,%s)",truedatapath,newdatapath);
              bb_error("rename");}
            db_insertar(path, nuevohash, entradadb.datapath, size, 0);
          }
          db_decrementar_duplicados(entradadb.sha1sum);
        }
      } else {
        bb_fullpath(truepath, path);
        //Calculamos el hash
        calcular_hash(truepath, nuevohash, &size);
        log_msg("    NewHash: %s\n", nuevohash);
        //No había entrada con este path, se inserta y nada más
        if (size > 0){ //Si el tamaño es mayor que 0, hay que insertar
          if (db_get_datapath_hash(nuevohash, entradadb.datapath, &(
              entradadb.deduplicados))){
            //Si el hash ya está en la bd, añadimos este archivo
            db_insertar(path, nuevohash, entradadb.datapath, size, entradadb
                .deduplicados);
            db_incrementar_duplicados(nuevohash);
            //Truncamos el archivo en el lugar original
            truncate(truepath,0);
          } else { //Único archivo con este hash
```

```
            //Mover a la dirección del hash. Hay que dejar un archivo vacío.
771         //conservamos los permisos originales
            int fd ;
            struct stat *prestat = malloc(sizeof(struct stat));
            if (stat(truepath,prestat)) {
              bb_error("stat al mover");}
776         //obtenemos el path donde se van a almacenar los datos
            preparar_datapath(nuevohash, entradadb.datapath);
            bb_fullpath(truedatapath,entradadb.datapath);
            //movemos los datos a la carpeta de datos
            if (rename(truepath, truedatapath)) {
781           bb_error("rename");}
            //Volvemos a crear el archivo del path original, con sus
               permisos
            fd = creat(truepath,prestat->st_mode);
            if (fd == -1) {
              bb_error("Create tras mover a datadir");}
786         close(fd);
            free(prestat);
            //insertamos la entrada en la base de datos
            db_insertar(path, nuevohash, entradadb.datapath, size, 0);
          }
791     }
      }
    }
    return retstat;
  }
796
  /** Synchronize file contents
   *
   * If the datasync parameter is non-zero, then only the user data
   * should be flushed, not the meta data.
801 *
   * Changed in version 2.2
   */
  int bb_fsync(const char *path, int datasync, struct fuse_file_info *fi)
  {
806 int retstat = 0;

    log_msg("\nbb_fsync(path=\"%s\", datasync=%d, fi=0x%08x)\n",
      path, datasync, fi);
    log_fi(fi);
811
    if (datasync)
    retstat = fdatasync(fi->fh);
    else
    retstat = fsync(fi->fh);
816
    if (retstat < 0)
    bb_error("bb_fsync fsync");

    return retstat;
821 }
```

```
/** Set extended attributes */
int bb_setxattr(const char *path, const char *name, const char *value,
    size_t size, int flags)
{
  int retstat = 0;
  char fpath[PATH_MAX];

  log_msg("\nbb_setxattr(path=\"%s\", name=\"%s\", value=\"%s\", size=%d,
      flags=0x%08x)\n",
    path, name, value, size, flags);
  bb_fullpath(fpath, path);

  retstat = lsetxattr(fpath, name, value, size, flags);
  if (retstat < 0)
  retstat = bb_error("bb_setxattr lsetxattr");

  return retstat;
}

/** Get extended attributes */
int bb_getxattr(const char *path, const char *name, char *value, size_t
    size)
{
  int retstat = 0;
  char fpath[PATH_MAX];

  log_msg("\nbb_getxattr(path = \"%s\", name = \"%s\", value = 0x%08x,
      size = %d)\n",
    path, name, value, size);
  bb_fullpath(fpath, path);

  retstat = lgetxattr(fpath, name, value, size);
  if (retstat < 0)
  retstat = bb_error("bb_getxattr lgetxattr");
  else
  log_msg("    value = \"%s\"\n", value);

  return retstat;
}

/** List extended attributes */
int bb_listxattr(const char *path, char *list, size_t size)
{
  int retstat = 0;
  char fpath[PATH_MAX];
  char *ptr;

  log_msg("bb_listxattr(path=\"%s\", list=0x%08x, size=%d)\n",
    path, list, size
    );
  bb_fullpath(fpath, path);
```

```
871     retstat = llistxattr(fpath, list, size);
        if (retstat < 0)
        retstat = bb_error("bb_listxattr llistxattr");

        log_msg("    returned attributes (length %d):\n", retstat);
876     for (ptr = list; ptr < list + retstat; ptr += strlen(ptr)+1)
        log_msg("    \"%s\"\n", ptr);

        return retstat;
      }
881
      /** Remove extended attributes */
      int bb_removexattr(const char *path, const char *name)
      {
        int retstat = 0;
886     char fpath[PATH_MAX];

        log_msg("\nbb_removexattr(path=\"%s\", name=\"%s\")\n",
          path, name);
        bb_fullpath(fpath, path);
891
        retstat = lremovexattr(fpath, name);
        if (retstat < 0)
        retstat = bb_error("bb_removexattr lrmovexattr");

896     return retstat;
      }

      /** Open directory
       *
901    * This method should check if the open operation is permitted for
       * this  directory
       *
       * Introduced in version 2.3
       */
906   int bb_opendir(const char *path, struct fuse_file_info *fi)
      {
        DIR *dp;
        int retstat = 0;
        char fpath[PATH_MAX];
911
        log_msg("\nbb_opendir(path=\"%s\", fi=0x%08x)\n",
          path, fi);
        bb_fullpath(fpath, path);

916     dp = opendir(fpath);
        if (dp == NULL)
        retstat = bb_error("bb_opendir opendir");

        fi->fh = (intptr_t) dp;
921
        log_fi(fi);
```

```
    return retstat;
}

/** Read directory
 *
 * This supersedes the old getdir() interface.  New applications
 * should use this.
 *
 * The filesystem may choose between two modes of operation:
 *
 * 1) The readdir implementation ignores the offset parameter, and
 * passes zero to the filler function's offset.  The filler
 * function will not return '1' (unless an error happens), so the
 * whole directory is read in a single readdir operation.  This
 * works just like the old getdir() method.
 *
 * 2) The readdir implementation keeps track of the offsets of the
 * directory entries.  It uses the offset parameter and always
 * passes non-zero offset to the filler function.  When the buffer
 * is full (or an error happens) the filler function will return
 * '1'.
 *
 * Introduced in version 2.3
 */
int bb_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t
    offset,
        struct fuse_file_info *fi)
{
    int retstat = 0;
    DIR *dp;
    struct dirent *de;

    log_msg("\nbb_readdir(path=\"%s\", buf=0x%08x, filler=0x%08x, offset=%
        lld, fi=0x%08x)\n",
      path, buf, filler, offset, fi);
    // once again, no need for fullpath -- but note that I need to cast fi->
        fh
    dp = (DIR *) (uintptr_t) fi->fh;

    // Every directory contains at least two entries: . and ..  If my
    // first call to the system readdir() returns NULL I've got an
    // error; near as I can tell, that's the only condition under
    // which I can get an error from readdir()
    de = readdir(dp);
    if (de == 0) {
    retstat = bb_error("bb_readdir readdir");
    return retstat;
    }

    // This will copy the entire directory into the buffer.  The loop exits
    // when either the system readdir() returns NULL, or filler()
    // returns something non-zero.  The first case just means I've
    // read the whole directory; the second means the buffer is full.
```

```
    do {
    log_msg("calling filler with name %s\n", de->d_name);
    if (filler(buf, de->d_name, NULL, 0) != 0) {
      log_msg("    ERROR bb_readdir filler:  buffer full");
      return -ENOMEM;
    }
    } while ((de = readdir(dp)) != NULL);

    log_fi(fi);

    return retstat;
}

/** Release directory
 *
 * Introduced in version 2.3
 */
int bb_releasedir(const char *path, struct fuse_file_info *fi)
{
    int retstat = 0;

    log_msg("\nbb_releasedir(path=\"%s\", fi=0x%08x)\n",
      path, fi);
    log_fi(fi);

    closedir((DIR *) (uintptr_t) fi->fh);

    return retstat;
}

/** Synchronize directory contents
 *
 * If the datasync parameter is non-zero, then only the user data
 * should be flushed, not the meta data
 *
 * Introduced in version 2.3
 */
// when exactly is this called?  when a user calls fsync and it
// happens to be a directory? ???
int bb_fsyncdir(const char *path, int datasync, struct fuse_file_info *fi)
{
    int retstat = 0;

    log_msg("\nbb_fsyncdir(path=\"%s\", datasync=%d, fi=0x%08x)\n",
      path, datasync, fi);
    log_fi(fi);

    return retstat;
}

/**
 * Initialize filesystem
 *
```

```
 * The return value will passed in the private_data field of
 * fuse_context to all file operations and as a parameter to the
 * destroy() method.
 *
 * Introduced in version 2.3
 * Changed in version 2.6
 */
// Undocumented but extraordinarily useful fact:  the fuse_context is
// set up before this function is called, and
// fuse_get_context()->private_data returns the user_data passed to
// fuse_main().  Really seems like either it should be a third
// parameter coming in here, or else the fact should be documented
// (and this might as well return void, as it did in older versions of
// FUSE).
void *bb_init(struct fuse_conn_info *conn)
{
  char * dbpath;
  struct bb_state *bb_data = BB_DATA;

  log_msg("\nbb_init()\n");
  //Abrimos la base de datos
  dbpath = malloc(sizeof(char)*PATH_MAX);
  bb_fullpath(dbpath,"/.dedupfs/dedupfs.db");
  bb_data->db = db_open(dbpath);
  free(dbpath);
  bb_data->mapopenw = map_open();
  return bb_data;
}

/**
 * Clean up filesystem
 *
 * Called on filesystem exit.
 *
 * Introduced in version 2.3
 */
void bb_destroy(void *userdata)
{
  //Cerramos las bd
  db_close(BB_DATA->db);
  map_close(BB_DATA->mapopenw);
  log_msg("\nbb_destroy(userdata=0x%08x)\n", userdata);
}

/**
 * Check file access permissions
 *
 * This will be called for the access() system call.  If the
 * 'default_permissions' mount option is given, this method is not
 * called.
 *
 * This method is not called under Linux kernel versions 2.4.x
 *
```

```
 * Introduced in version 2.5
 */
int bb_access(const char *path, int mask)
{
  int retstat = 0;
  char fpath[PATH_MAX];

  log_msg("\nbb_access(path=\"%s\", mask=0%o)\n",
    path, mask);
  bb_fullpath(fpath, path);

  retstat = access(fpath, mask);

  if (retstat < 0)
  retstat = bb_error("bb_access access");

  return retstat;
}

/**
 * Create and open a file
 *
 * If the file does not exist, first create it with the specified
 * mode, and then open it.
 *
 * If this method is not implemented or under Linux kernel
 * versions earlier than 2.6.15, the mknod() and open() methods
 * will be called instead.
 *
 * Introduced in version 2.5
 */
int bb_create(const char *path, mode_t mode, struct fuse_file_info *fi)
{
  int retstat = 0;
  char fpath[PATH_MAX];
  int fd;

  log_msg("\nbb_create(path=\"%s\", mode=0%03o, fi=0x%08x)\n",
      path, mode, fi);
  bb_fullpath(fpath, path);

  fd = creat(fpath, mode);
  if (fd < 0)
    retstat = bb_error("bb_create creat");

  fi->fh = fd;

  log_fi(fi);

  // Se da por supuesto que una llamada a create equivale a abrir
  // para escritura, lo introducimos en el mapa.
  map_add(fi->fh, path, 0, 0);
```

```
    return retstat;
}

/**
 * Change the size of an open file
 *
 * This method is called instead of the truncate() method if the
 * truncation was invoked from an ftruncate() system call.
 *
 * If this method is not implemented or under Linux kernel
 * versions earlier than 2.6.15, the truncate() method will be
 * called instead.
 *
 * Introduced in version 2.5
 */
int bb_ftruncate(const char *path, off_t offset, struct fuse_file_info *fi
    )
{
    int retstat = 0;

    log_msg("\nbb_ftruncate(path=\"%s\", offset=%lld, fi=0x%08x)\n",
      path, offset, fi);
    log_fi(fi);

    //Establecer como modificado este descriptor
    map_set_modificado(fi->fh);

    retstat = ftruncate(fi->fh, offset);
    if (retstat < 0)
    retstat = bb_error("bb_ftruncate ftruncate");

    return retstat;
}

/** Change the size of a file */
int bb_truncate(const char *path, off_t newsize)
{
    int retstat = 0;

    log_msg("\nbb_truncate(path=\"%s\", newsize=%lld)\n",
      path, newsize);

    // Truncate puede verse como una apertura de archivo,
    // truncamiento hasta el tamaño adecuado (modificación),
    // y cierre con todas las implicaciones que tiene el cierre.
    // Se usa bb_open, bb_ftruncate y bb_release para ello.
    struct fuse_file_info *fi = malloc(sizeof(struct fuse_file_info));
    fi->flags = O_RDWR; //Para acceder en escritura
    retstat = bb_open(path, fi);
    if (retstat == 0) {
      //Se ha abierto adecuadamente, truncamos
      retstat = bb_ftruncate(path,newsize,fi);
      if (retstat == 0) {
```

```
        //Se ha truncado adecuadamente, cerramos
        retstat = bb_release(path, fi);
      }
    }
    free(fi);

    if (retstat < 0) {
      bb_error("bb_truncate truncate");
      close(fi->fh);
    }

    return retstat;
}

/**
 * Get attributes from an open file
 *
 * This method is called instead of the getattr() method if the
 * file information is available.
 *
 * Currently this is only called after the create() method if that
 * is implemented (see above).  Later it may be called for
 * invocations of fstat() too.
 *
 * Introduced in version 2.5
 */
// Since it's currently only called after bb_create(), and bb_create()
// opens the file, I ought to be able to just use the fd and ignore
// the path...
int bb_fgetattr(const char *path, struct stat *statbuf, struct
    fuse_file_info *fi)
{
    int retstat = 0;
    struct db_entry entradabd;
    char fpath[PATH_MAX];
    log_msg("\nbb_fgetattr(path=\"%s\", statbuf=0x%08x, fi=0x%08x)\n",
      path, statbuf, fi);
    log_fi(fi);

    //Ahora mismo recibimos el stat del deduplicado.
    //Se llama a getattr desde aquí, y se evita
    retstat = fstat(fi->fh, statbuf);
    if (retstat < 0) {
      retstat = bb_error("bb_fgetattr fstat");
      return retstat;
    }
    // Desreferenciamos path si es un enlace duro
    if(db_getLinkPath(path, fpath)) {
      //Si es true se ha desreferenciado y guardado en fpath.
      //Cambiamos el puntero de path a fpath
      path=fpath;
    }
    //Si el archivo está en la bd, el tamaño se toma de ahí
```

```
    if( db_get(path,&entradabd)){
      statbuf->st_size = entradabd.size;
      //Necesario incluir st_blocks? Lo incluimos
      //damos por supuesto tamaño de bloque de 4096
      statbuf->st_blocks = (entradabd.size / 4096 + (entradabd.size % 4096 >
          0))*8;
    }

    log_stat(statbuf);

    return retstat;
}

struct fuse_operations bb_oper = {
    .getattr = bb_getattr,
    .readlink = bb_readlink,
    // .getdir = NULL,
    .mknod = bb_mknod,
    .mkdir = bb_mkdir,
    .rmdir = bb_rmdir,
    .symlink = bb_symlink,
    .rename = bb_rename,
    .link = bb_link,
    .unlink = bb_unlink,
    .chmod = bb_chmod,
    .chown = bb_chown,
    .truncate = bb_truncate,
    .utime = bb_utime,
    .open = bb_open,
    .read = bb_read,
    .write = bb_write,
    .statfs = bb_statfs,
    .flush = bb_flush,
    .release = bb_release,
    .fsync = bb_fsync,
    .setxattr = bb_setxattr,
    .getxattr = bb_getxattr,
    .listxattr = bb_listxattr,
    .removexattr = bb_removexattr,
    .opendir = bb_opendir,
    .readdir = bb_readdir,
    .releasedir = bb_releasedir,
    .fsyncdir = bb_fsyncdir,
    .init = bb_init,
    .destroy = bb_destroy,
    .access = bb_access,
    .create = bb_create,
    .ftruncate = bb_ftruncate,
    .fgetattr = bb_fgetattr
};

void bb_usage()
{
```

```
    fprintf(stderr, "usage:  dedupfs [FUSE and mount options] rootDir
        mountPoint\n");
    abort();
}

/**
 * Comprobar si existe el directorio .dedupdir en el árbol de directorios
 * subyacente, y crearlo si no existe.
 */
static void check_conf_dir(char rootdir[PATH_MAX]){
    struct stat s;
    char * dedupdir = malloc(sizeof(char) * PATH_MAX);
    strcpy(dedupdir, rootdir);
    strncat(dedupdir, "/.dedupfs", PATH_MAX);
    int err = stat(dedupdir, &s);
    if(-1 == err) {
        if(ENOENT == errno) {
        /* No existe, se crea */
            mkdir(dedupdir, 0777);
            // El de los datos también
            strcat(dedupdir,"/data");
            mkdir(dedupdir, 0777);
        } else {
        perror("Stat error al acceder al directorio \".dedupfs\"");
        exit(1);
        }
    } else {
        if(S_ISDIR(s.st_mode)) {
        /* Directorio, comprobamos permisos */
            if (!(s.st_mode & S_IRWXU)) {
                perror("\".dedupfs\" debe tener permisos rwx");
                exit(1);
            }
        } else {
        /* exists but is no dir */
        perror("\".dedupfs\" debe ser un directorio o no existir");
        exit(1);
        }
    }
    free(dedupdir);
}

int main(int argc, char *argv[])
{
    int fuse_stat;
    struct bb_state *bb_data;

    // asegurarse de que no se ejecuta como root
    if ((getuid() == 0) || (geteuid() == 0)) {
    fprintf(stderr, "Running BBFS as root opens unnacceptable security holes
        \n");
    return 1;
    }
```

```
1341    // Perform some sanity checking on the command line:  make sure
        // there are enough arguments, and that neither of the last two
        // start with a hyphen (this will break if you actually have a
        // rootpoint or mountpoint whose name starts with a hyphen, but so
        // will a zillion other programs)
1346    if ((argc < 3) || (argv[argc-2][0] == '-') || (argv[argc-1][0] == '-'))
        bb_usage();

        bb_data = malloc(sizeof(struct bb_state));
        if (bb_data == NULL) {
1351    perror("main calloc");
        abort();
        }

        // Pull the rootdir out of the argument list and save it in my
1356    // internal data
        bb_data->rootdir = realpath(argv[argc-2], NULL);
        argv[argc-2] = argv[argc-1];
        argv[argc-1] = NULL;
        argc--;
1361
        bb_data->logfile = log_open("dedupfs.log");

        //Comprobar si .dedupfs existe y que sea legible.
        check_conf_dir(bb_data->rootdir);
1366    //TODO se puede hacer que check_conf devuelva un valor si existen
           archivos
        //  en el rootdir, y que dichos archivos se transfieran tras llamar a
           fuse
        //  para ello, fuse_main debe volver tras la llamada(lo hace?)

        // turn over control to fuse
1371    fprintf(stderr, "about to call fuse_main\n");
        fuse_stat = fuse_main(argc, argv, &bb_oper, bb_data);
        fprintf(stderr, "fuse_main returned %d\n", fuse_stat);

        return fuse_stat;
1376  }
```

6.2. database.h

```
        #ifndef _DATABASE_H_
4       #define _DATABASE_H_
        #include <stdio.h>
        #include <stdlib.h>
        #include <sqlite3.h>
```

```
#include <limits.h>
#include "params.h"


sqlite3 * db_open (const char *path);
int db_close (sqlite3 * db);
int db_insertar(const char *path, const char *shasum, const char *datapath
    , unsigned int size, int deduplicados);
int db_get(const char *path, struct db_entry *entrada);
int db_get_datapath_hash(const char * hash, char * path, int *
   deduplicados);
void db_incrementar_duplicados(const char * hash);
void db_decrementar_duplicados(const char * hash);
int db_eliminar(const char * path);

int db_addLink(const char * enlace, const char * pathoriginal);
int db_getLinkPath(const char * enlace, char * pathoriginal);
int db_unlink(const char * path);
int db_link_get_heredero(const char * path, char *heredero);
int db_link_heredar(const char *path, const char *heredero);

void db_rename(const char *path, const char *newpath);

sqlite3 * map_open ();
int map_close();
int map_add(unsigned long long int fh, const char * path, int deduplicado,
     char modificado);
int map_extract(unsigned long long int fh, struct map_entry * entrada, int
     eliminar);
int map_count(const char * path);
void map_set_modificado(unsigned long long int fh);

#endif
```

6.3. database.c

```
/*
  Módulo de dedupfs encargado de la interacción con
  la base de datos sqlite utilizada.
*/
#define _GNU_SOURCE

#include "log.h"
#include "database.h"

#include <string.h>
```

```
14  sqlite3 * db_open (const char *path) {
      sqlite3 *db;
      int rc;
      char *dbErrMsg = 0;

19    rc = sqlite3_open(path, &db);
      if( rc ){
        log_msg("      Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        abort();
24    }
      // Crear las tablas si no están creadas
      char * peticion =
          "CREATE TABLE IF NOT EXISTS files("
            "path TEXT PRIMARY KEY ON CONFLICT FAIL,"
29          "shasum TEXT NOT NULL ON CONFLICT FAIL,"
            "datapath TEXT NOT NULL ON CONFLICT FAIL,"
            "size INTEGER NOT NULL ON CONFLICT FAIL,"
            "deduplicados INTEGER NOT NULL ON CONFLICT FAIL"
          ");"
34        "CREATE TABLE IF NOT EXISTS links("
            "linkpath TEXT PRIMARY KEY ON CONFLICT FAIL,"
            "originalpath TEXT NOT NULL ON CONFLICT FAIL"
          ");";
      rc = sqlite3_exec(db, peticion, /*callback*/NULL, 0, &dbErrMsg);
39    if( rc!=SQLITE_OK ){
        log_msg("      Error accediendo a la base de datos\n"
            "      SQL error: %s\n", dbErrMsg);
        sqlite3_free(dbErrMsg);
        abort();
44    }
      return db;
    }

    int db_close (sqlite3 *db) {
49    return sqlite3_close(db);
    }
    /**
     * Se introduce esta entrada en la base de datos de ficheros. Si ya existe
         se reemplaza
     */
54  int db_insertar(const char * path, const char * shasum, const char *
       datapath, unsigned int size, int deduplicados){
      int rc;
      static sqlite3_stmt * stmt;
      sqlite3 * mapa = BB_DATA->db;
      sqlite3_prepare_v2(mapa,"INSERT OR REPLACE INTO files VALUES ( ?1, ?2,
          ?3, ?4, ?5)", -1,&stmt,NULL);
59    log_msg("    db_insertar(%s, %s, %s, %u, %d)\n", path, shasum, datapath,
          size, deduplicados);
      sqlite3_bind_text(stmt,1,path,-1,SQLITE_STATIC);
      sqlite3_bind_text(stmt,2,shasum,-1,SQLITE_STATIC);
      sqlite3_bind_text(stmt,3,datapath,-1,SQLITE_STATIC);
```

```
    sqlite3_bind_int(stmt, 4, size);
    sqlite3_bind_int(stmt, 5, deduplicados);

    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
      log_msg("      ERROR inserting data: %s\n", sqlite3_errmsg(mapa));
      return 0;
    }
    sqlite3_finalize(stmt);
    return 1;
}

/**
 * Recupera la entrada correspondiente a path.
 * Devuelve 1 si se ha recuperado, 0 en caso contrario.
 */
int db_get(const char * path, struct db_entry *entrada){
    static sqlite3_stmt *stmt;
    sqlite3 * mapa = BB_DATA->db;
    int rc,found=0;

    sqlite3_prepare_v2(mapa,"SELECT * FROM files WHERE (path = ?1)", -1,&
        stmt,NULL);
    sqlite3_bind_text(stmt,1,path,-1,SQLITE_STATIC);
    log_msg("    db_get(%s)\n", path);
    rc = sqlite3_step(stmt);
    log_msg("      rc_code= %d\n", rc);
    if (rc == SQLITE_ROW) {//Devuelve una fila. recuperamos los valores
      strcpy (entrada->path, (char *)sqlite3_column_text(stmt,0));
      strcpy (entrada->sha1sum, (char *)sqlite3_column_text(stmt,1));
      strcpy (entrada->datapath, (char *)sqlite3_column_text(stmt,2));
      entrada->size = sqlite3_column_int(stmt,3);
      entrada->deduplicados = sqlite3_column_int(stmt,4);
      log_msg("      Encontrada: %s, %s, %s, %u, %d\n", entrada->path,
          entrada->sha1sum,
           entrada->datapath, entrada->size, entrada->deduplicados);
      found=1;
    }
    else if (rc != SQLITE_DONE) {
      log_msg("       ERROR recuperando del mapa: %s\n", sqlite3_errmsg(mapa
          ));
    }
    sqlite3_finalize(stmt);
    return found;
}

/**
 * Recupera el datapath de un hash dado, y la cuenta de deduplicados.
 * Devuelve 1 si se ha encontrado, y 0 en caso contrario.
 */
int db_get_datapath_hash(const char * hash, char * path, int *
    deduplicados) {
    static sqlite3_stmt *stmt;
```

```
    sqlite3 * mapa = BB_DATA->db;
    int rc,found=0;

    sqlite3_prepare_v2(mapa,"SELECT datapath,deduplicados FROM files WHERE (
        shasum = ?1)",
              -1,&stmt,NULL);
    sqlite3_bind_text(stmt,1,hash,-1,SQLITE_STATIC);
    log_msg("    db_get_datapath_hash(%s)\n", hash);
    rc = sqlite3_step(stmt);
    log_msg("      rc_code= %d\n", rc);
    if (rc == SQLITE_ROW) {//Devuelve una fila. recuperamos los valores
      strcpy (path, (char *)sqlite3_column_text(stmt,0));
      *deduplicados = sqlite3_column_int(stmt,1);
      log_msg("      Encontrada: %s, %d\n", path,*deduplicados);
      found=1;
    }
    else if (rc != SQLITE_DONE) {
      log_msg("       ERROR recuperando: %s\n", sqlite3_errmsg(mapa));
    }
    sqlite3_finalize(stmt);
    return found;
  }

  /**
   * Se incrementa la cuenta de duplicados en cada entrada que coincida con
      el hash.
   */
  void db_incrementar_duplicados(const char * hash){
    static sqlite3_stmt *stmt;
    sqlite3 * mapa = BB_DATA->db;
    int rc;
    log_msg("    db_incrementar_duplicados(%s)\n", hash);
    sqlite3_prepare_v2(mapa,"UPDATE files SET deduplicados = deduplicados+1
        "
        "WHERE (shasum = ?1)", -1,&stmt,NULL);
    sqlite3_bind_text(stmt,1,hash,-1,SQLITE_STATIC);
    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
        log_msg("       ERROR incrementando duplicados: %s\n",
           sqlite3_errmsg(mapa));
    }
    sqlite3_finalize(stmt);
  }

  /**
   * Se decrementa la cuenta de duplicados en cada entrada que coincida con
      el hash.
   */
  void db_decrementar_duplicados(const char * hash){
    static sqlite3_stmt *stmt;
    sqlite3 * mapa = BB_DATA->db;
    int rc;
    log_msg("    db_decrementar_duplicados(%s)\n", hash);
```

```
    sqlite3_prepare_v2(mapa,"UPDATE files SET deduplicados = deduplicados-1
        "
         "WHERE (shasum = ?1)", -1,&stmt,NULL);
    sqlite3_bind_text(stmt,1,hash,-1,SQLITE_STATIC);
    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
         log_msg("       ERROR decrementando duplicados: %s\n",
            sqlite3_errmsg(mapa));
    }
    sqlite3_finalize(stmt);
}

/**
 * Elimina una entrada de la base de datos.
 */
int db_eliminar(const char * path){
    static sqlite3_stmt *stmt;
    sqlite3 * mapa = BB_DATA->db;
    int rc;
    log_msg("    db_eliminar(%s)\n",path);
    sqlite3_prepare_v2(mapa,"DELETE FROM files WHERE (path = ?1)", -1,&stmt,
        NULL);
    sqlite3_bind_text(stmt,1,path,-1,SQLITE_STATIC);
    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
         log_msg("       ERROR eliminando en db: %s\n", sqlite3_errmsg(mapa))
            ;
         rc = 0;
    }
    sqlite3_finalize(stmt);
    rc=1;
    return rc;
}

/**
 * Añade una entrada a la tabla de enlaces físicos, enlace es el path
     enlazado,
 * pathoriginal es el path del archivo creado originalmente.
 * Los enlaces físicos no hacen distinción entre enlace y enlazado, aquí
     es
 * necesario puesto que en la tabla de archivos deduplicados utilizamos el
     path
 * del archivo original.
 * También se desreferencia hasta apuntar al último nodo
 * Devuelve true  si se ha insertado, false si no.
 */
int db_addLink(const char * pathoriginal, const char * enlace){
    int rc, retval=1;
    static sqlite3_stmt * stmt;
    sqlite3 * db = BB_DATA->db;
    char * desreferenciado = strdup(pathoriginal);
    char * desreferenciar = strdup(pathoriginal);
    char * auxptr;
```

```
    //Desreferenciamos el pathoriginal en caso de que sea necesario.
    log_msg("    db_addlink(%s, %s)\n", pathoriginal, enlace);
    while(db_getLinkPath(desreferenciar,desreferenciado)) {
      auxptr = desreferenciado;
      desreferenciado = desreferenciar;
      desreferenciar = auxptr;
      log_msg("        derp\n");
    }

    sqlite3_prepare_v2(db,"INSERT INTO links VALUES ( ?1, ?2)", -1,&stmt,
       NULL);
    log_msg("      desreferenciado: %s\n", desreferenciar);
    sqlite3_bind_text(stmt,1,enlace,-1,SQLITE_STATIC);
    sqlite3_bind_text(stmt,2,desreferenciar,-1,SQLITE_STATIC);

    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
      log_msg("      ERROR: %s\n", sqlite3_errmsg(db));
      retval = 0;
    }
    sqlite3_finalize(stmt);
    free(desreferenciado);
    free(desreferenciar);
    return retval;
  }

  /**
   * Recupera el path del archivo original al que apunta enlace, y lo guarda
       en
   * pathoriginal.
   * Devuelve true si se ha recuperado, false si no.
   */
  int db_getLinkPath(const char * enlace, char * pathoriginal) {
    static sqlite3_stmt *stmt;
    sqlite3 * db = BB_DATA->db;
    int rc,found=0;

    sqlite3_prepare_v2(db,"SELECT originalpath FROM links WHERE (linkpath =
       ?1)", -1,&stmt,NULL);
    sqlite3_bind_text(stmt,1,enlace,-1,SQLITE_STATIC);
    log_msg("    db_getLinkPath(%s)\n", enlace);
    rc = sqlite3_step(stmt);
    log_msg("      rc_code= %d\n", rc);
    if (rc == SQLITE_ROW) {//Devuelve una fila. recuperamos los valores
      strcpy (pathoriginal, (char *)sqlite3_column_text(stmt,0));

      log_msg("      Encontrada: %s, %s\n", enlace, pathoriginal);
      found=1;
    }
    else if (rc != SQLITE_DONE) {
      log_msg("       ERROR recuperando: %s\n", sqlite3_errmsg(db));
    }
    sqlite3_finalize(stmt);
```

```
  return found;
}

/*
 * Elimina el enlace indicado por path.
 * Devuelve true si se ha eliminado, false en caso contrario.
 */
int db_unlink(const char * path) {
  int rc;
  static sqlite3_stmt * stmt;
  sqlite3 * db = BB_DATA->db;
  log_msg("    db_unlink(%s)\n",path);

  // Probamos a eliminar una entrada con linkpath = path
  sqlite3_prepare_v2(db,"DELETE fom links where linkpath = ?1", -1,&stmt,
     NULL);
  sqlite3_bind_text(stmt,1,path,-1,SQLITE_STATIC);
  rc = sqlite3_step(stmt);
  if (rc != SQLITE_DONE) {
    log_msg("      ERROR: %s\n", sqlite3_errmsg(db));
    return 0;
  }
  if (sqlite3_changes(db)==0) {
    return 0;
  }
  return 1;
}

/**
 * Busca en la tabla de enlaces uno que haga referencia a path y lo
    devuelve
 * en heredero. Se utiliza cuando es necesario eliminar la referencia de
    la
 * tabla de archivos, y hay que mirar si hay un heredero.
 * Devuelve true si ha encontrado heredero, y false en caso contrario
 */
int db_link_get_heredero(const char * path, char *heredero){
  static sqlite3_stmt *stmt;
  sqlite3 * db = BB_DATA->db;
  int rc,found=0;

  sqlite3_prepare_v2(db,"select linkpath from links where originalpath =
     ?1 limit 1", -1,&stmt,NULL);
  sqlite3_bind_text(stmt,1,path,-1,SQLITE_STATIC);
  log_msg("    db_linkget_heredero(%s)\n", path);
  rc = sqlite3_step(stmt);
  log_msg("      rc_code= %d\n", rc);
  if (rc == SQLITE_ROW) {//Devuelve una fila. recuperamos los valores
    strcpy (heredero, (char *)sqlite3_column_text(stmt,0));
    log_msg("      Encontrada: %s, %s\n", path, heredero);
    found=1;
  }
  else if (rc != SQLITE_DONE) {
```

```
    log_msg("       ERROR recuperando: %s\n", sqlite3_errmsg(db));
  }
  sqlite3_finalize(stmt);
  return found;
}

/**
 * Modifica todas las referencias que apunten a path en la tabla de
    enlaces,
 * y las sustituye por heredero.
 * Modifica la referencia con 'path' == path en la tabla de ficheros, en
    caso
 * de haberla.
 * Devuelve true si se han realizado cambios, false si no.
 */
int db_link_heredar(const char *path, const char *heredero){
  int rc, cambios=0;
  sqlite3 * db = BB_DATA->db;
  log_msg("    db_link_heredar(%s, %s); ",path, heredero);
  //Se modifican las referencias a path por heredero, y se elimina en caso
      de
  // que haya una tras la modificación que se referencie a sí misma.
  char *dbErrMsg=0, *peticion;
  asprintf(&peticion,"UPDATE files SET path = '%s' WHERE path = '%s' ;"
      "UPDATE links SET originalpath = '%s' WHERE originalpath = '%s' ;"
      "DELETE from links where linkpath = originalpath;",
      heredero,path,heredero,path);
  rc = sqlite3_exec(db, peticion, /*callback*/NULL, 0, &dbErrMsg);
  free(peticion);
  cambios += sqlite3_changes(db);
  if( rc!=SQLITE_OK ){
    log_msg("      Error accediendo a la base de datos: %s\n", dbErrMsg);
    sqlite3_free(dbErrMsg);
  }
  log_msg("cambios: %d", cambios);
  if ( cambios==0){
    return 0;
  }
  return 1;
}

/**
 * Renombra todas las referencias de path a newpath en ambas tablas
 * de la base de datos
 */
void db_rename(const char *path, const char *newpath){
  int rc, cambios=0;
  sqlite3 * db = BB_DATA->db;
  log_msg("    db_rename(%s, %s); ",path, newpath);
  //Se modifican las referencias a path por newpath en las tablas.
  char *dbErrMsg=0, *peticion;
  asprintf(&peticion,"UPDATE files SET path = '%s' WHERE path = '%s' ;"
      "UPDATE links SET originalpath = '%s' WHERE originalpath = '%s' ;"
```

```
        "UPDATE links SET linkpath = '%s' WHERE linkpath = '%s';",
        newpath,path,
        newpath,path,
        newpath,path);
359   rc = sqlite3_exec(db, peticion, /*callback*/NULL, 0, &dbErrMsg);
      free(peticion);
      cambios += sqlite3_changes(db);
      if( rc!=SQLITE_OK ){
        log_msg("      Error accediendo a la base de datos: %s\n", dbErrMsg);
364     sqlite3_free(dbErrMsg);
      }
      log_msg("cambios: %d", cambios);
    }

369

    /**
     * El mapa de ficheros abiertos se guarda en una base de datos en memoria
     * Para cada fichero, guarda cuantas veces está abierto para escritura,
     * y si ha sido modificado.
374  */
    sqlite3 * map_open() {
      sqlite3 * mapa;
      int rc;
      char * dbErrMsg = 0;
379   rc = sqlite3_open(":memory:", &mapa);
      if( rc ){
        log_msg("      Can't open database: %s\n", sqlite3_errmsg(mapa));
        sqlite3_close(mapa);
        abort();
384   }
      log_msg("      bd mem_mapa() creada\n");
      // Crear la tabla si no está creada
      char * peticion =
          "CREATE TABLE IF NOT EXISTS mapa("
389         "fh INTEGER PRIMARY KEY ON CONFLICT FAIL,"
            "path TEXT NOT NULL ON CONFLICT FAIL,"
            "deduplicado INTEGER DEFAULT 0,"
            "modificado INTEGER DEFAULT 0"
            ");";
394   rc = sqlite3_exec(mapa, peticion, /*callback*/NULL, 0, &dbErrMsg);
      if( rc!=SQLITE_OK ){
        log_msg("      Error accediendo a la base de datos: %s\n", dbErrMsg);
        sqlite3_free(dbErrMsg);
        abort();
399   }
      return mapa;
    }

    int map_close () {
404   return sqlite3_close(BB_DATA->mapopenw);
    }

    /**
```

```
 * Se inserta un nuevo descriptor de fichero en el mapa de ficheros
    abiertos.
 */
int map_add(unsigned long long int fh,const char * path, int deduplicado,
   char modificado){
  int rc;
  static sqlite3_stmt * stmt;
  sqlite3 * mapa = BB_DATA->mapopenw;
  sqlite3_prepare_v2(mapa,"INSERT INTO mapa VALUES ( ?1, ?2, ?3, ?4)",
     -1,&stmt,NULL);
  log_msg("    map_add(%llu, %s, %d, %d)\n", fh, path, deduplicado,
     modificado);
  sqlite3_bind_int64(stmt,1,fh);
  sqlite3_bind_text(stmt,2,path,-1,SQLITE_STATIC);
  sqlite3_bind_int(stmt, 3, deduplicado);
  sqlite3_bind_int(stmt, 4, modificado);

  rc = sqlite3_step(stmt);
  if (rc != SQLITE_DONE) {
    log_msg("      ERROR inserting data: %s\n", sqlite3_errmsg(mapa));
    return 0;
  }
  sqlite3_finalize(stmt);
  return 1;
}
/**
 * Se recupera la entrada del descriptor fh si existe, y se guarda en
    entrada.
 * Si eliminar es true, además se elimina de la tabla en caso de existir.
 * Devuelve 1 si se ha recuperado, 0 si no
 */
int map_extract(unsigned long long int fh, struct map_entry * entrada, int
    eliminar){
  static sqlite3_stmt *stmt_select, *stmt_delete;
  sqlite3 * mapa = BB_DATA->mapopenw;
  int rc,found=0;

  sqlite3_prepare_v2(mapa,"SELECT * FROM mapa WHERE (fh = ?1)", -1,&
     stmt_select,NULL);
  sqlite3_bind_int64(stmt_select,1,fh);
  log_msg("    map_extract(%llu, %d)\n", fh, eliminar);
  rc = sqlite3_step(stmt_select);
  log_msg("      rc_code= %d\n", rc);
  if (rc == SQLITE_ROW) {//Devuelve una fila. recuperamos los valores
    entrada->fh=sqlite3_column_int64(stmt_select,0);
    strcpy (entrada->path, (char *)sqlite3_column_text(stmt_select,1));
    entrada->deduplicados = sqlite3_column_int(stmt_select,2);
    entrada->modificado = sqlite3_column_int(stmt_select,3);
    log_msg("      Encontrada fila: %llu, %s, %d, %d\n",
        entrada->fh, entrada->path, entrada->deduplicados, entrada->
           modificado);
    if (eliminar){ //Eliminamos si se solicita
      sqlite3_prepare_v2(mapa,"DELETE FROM mapa WHERE (fh = ?1)", -1,&
```

```
          stmt_delete,NULL);
      sqlite3_bind_int64(stmt_delete,1,fh);
      rc = sqlite3_step(stmt_delete);
      if (rc != SQLITE_DONE) {
          log_msg("       ERROR eliminando en mapa: %s\n", sqlite3_errmsg(
             mapa));
      }
      sqlite3_finalize(stmt_delete);
    }
    found=1;
  }
  else if (rc != SQLITE_DONE) {
    log_msg("       ERROR recuperando del mapa: %s\n", sqlite3_errmsg(mapa
       ));
  }
  sqlite3_finalize(stmt_select);
  return found;
}

/**
 * Devuelve el número de entradas en el mapa para una misma ruta de
    archivo
 */
int map_count(const char * path){
  int rc,count;
  static sqlite3_stmt *stmt_count;
  sqlite3 * mapa = BB_DATA->mapopenw;
  sqlite3_prepare_v2(mapa,"SELECT COUNT(fh) FROM mapa WHERE (path = ?1)",
     -1,&stmt_count,NULL);
  sqlite3_bind_text(stmt_count,1,path,-1,SQLITE_STATIC);
  rc = sqlite3_step(stmt_count);
  if (rc == SQLITE_ROW) {//Devuelve una fila. recuperamos el valor
    count = sqlite3_column_int(stmt_count,0);
    log_msg("    map_count(%s) = %d\n", path, count);
  }
  else if (rc != SQLITE_DONE) {
    log_msg("       ERROR map_count(): %s\n", sqlite3_errmsg(mapa));
    count=-1;
  }
  sqlite3_finalize(stmt_count);
  return count;
}

/**
 *  Actualiza las entradas del mapa que coincidan con path,
 *  activando el flag modificado
 */
void map_set_modificado(unsigned long long int fh){
  int rc;
  char *dbErrMsg, *peticion;
  sqlite3 * mapa = BB_DATA->mapopenw;
  asprintf(&peticion, "UPDATE mapa SET modificado = 1 WHERE fh=%llu", fh);
  rc = sqlite3_exec(mapa, peticion, /*callback*/NULL, 0, &dbErrMsg);
```

```
    log_msg("    map_set_modificado(%llu)\n",fh);
    free(peticion);
    if( rc!=SQLITE_OK ){
      log_msg("      Error accediendo a la base de datos: %s\n", dbErrMsg);
      sqlite3_free(dbErrMsg);
    }
  }
```

6.4. log.h

```
  /*
    Copyright (C) 2012 Joseph J. Pfeiffer, Jr., Ph.D. <pfeiffer@cs.nmsu.edu>

    This program can be distributed under the terms of the GNU GPLv3.
    See the file COPYING.
  */

  #ifndef _LOG_H_
  #define _LOG_H_

  #include "params.h"
  #include <fuse.h>
  #include <stdarg.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <unistd.h>
  #include <stdio.h>
  #include <sys/types.h>
  #include <sys/stat.h>



  //  macro to log fields in structs.
  #define log_struct(st, field, format, typecast) \
    log_msg("    " #field " = " #format "\n", typecast st->field)

  FILE *log_open(char path[PATH_MAX]);
  void log_fi (struct fuse_file_info *fi);
  void log_stat(struct stat *si);
  void log_statvfs(struct statvfs *sv);
  void log_utime(struct utimbuf *buf);

  void log_msg(const char *format, ...);
  #endif
```

6.5. log.c

```
/*
  Copyright (C) 2012 Joseph J. Pfeiffer, Jr., Ph.D. <pfeiffer@cs.nmsu.edu>

  This program can be distributed under the terms of the GNU GPLv3.
  See the file COPYING.

  Since the point of this filesystem is to learn FUSE and its
  datastructures, I want to see *everything* that happens related to
  its data structures.  This file contains macros and functions to
  accomplish this.
*/

#include "params.h"
#include "log.h"

FILE *log_open(char path[PATH_MAX])
{
#ifdef NOLOG
  return NULL;
#endif
  FILE *logfile;

    // very first thing, open up the logfile and mark that we got in
    // here.  If we can't open the logfile, we're dead.
    logfile = fopen(path, "w");
    if (logfile == NULL) {
  perror("logfile");
  exit(EXIT_FAILURE);
    }

    // set logfile to line buffering
    setvbuf(logfile, NULL, _IOLBF, 0);

    return logfile;
}

void log_msg(const char *format, ...)
{
#ifdef NOLOG
  return ;
#endif
  va_list ap;
    va_start(ap, format);

    vfprintf(BB_DATA->logfile, format, ap);
}

// struct fuse_file_info keeps information about files (surprise!).
// This dumps all the information in a struct fuse_file_info.  The struct
// definition, and comments, come from /usr/include/fuse/fuse_common.h
```

```
51  // Duplicated here for convenience.
    void log_fi (struct fuse_file_info *fi)
    {
    #ifdef NOLOG
      return ;
56  #endif
      /** Open flags.  Available in open() and release() */
        //  int flags;
      log_struct(fi, flags, 0x%08x, );

61      /** Old file handle, don't use */
        //  unsigned long fh_old;
      log_struct(fi, fh_old, 0x%08lx,  );

        /** In case of a write operation indicates if this was caused by a
66          writepage */
        //  int writepage;
      log_struct(fi, writepage, %d, );

        /** Can be filled in by open, to use direct I/O on this file.
71          Introduced in version 2.4 */
        //  unsigned int keep_cache : 1;
      log_struct(fi, direct_io, %d, );

        /** Can be filled in by open, to indicate, that cached file data
76          need not be invalidated.  Introduced in version 2.4 */
        //  unsigned int flush : 1;
      log_struct(fi, keep_cache, %d, );

        /** Padding.  Do not use*/
81      //  unsigned int padding : 29;

        /** File handle.  May be filled in by filesystem in open().
            Available in all other file operations */
        //  uint64_t fh;
86    log_struct(fi, fh, 0x%016llx,  );

        /** Lock owner id.  Available in locking operations and flush */
        //  uint64_t lock_owner;
      log_struct(fi, lock_owner, 0x%016llx, );
91  };

    // This dumps the info from a struct stat.  The struct is defined in
    // <bits/stat.h>; this is indirectly included from <fcntl.h>
    void log_stat(struct stat *si)
96  {
    #ifdef NOLOG
      return ;
    #endif
      //  dev_t     st_dev;     /* ID of device containing file */
101   log_struct(si, st_dev, %lld, );

        //  ino_t     st_ino;     /* inode number */
```

```
    log_struct(si, st_ino, %lld, );

106   //  mode_t    st_mode;    /* protection */
    log_struct(si, st_mode, 0%o, );

    //  nlink_t   st_nlink;   /* number of hard links */
    log_struct(si, st_nlink, %d, );
111
    //  uid_t     st_uid;     /* user ID of owner */
    log_struct(si, st_uid, %d, );

    //  gid_t     st_gid;     /* group ID of owner */
116 log_struct(si, st_gid, %d, );

    //  dev_t     st_rdev;    /* device ID (if special file) */
    log_struct(si, st_rdev, %lld,  );

121 //  off_t     st_size;    /* total size, in bytes */
    log_struct(si, st_size, %lld,  );

    //  blksize_t st_blksize; /* blocksize for filesystem I/O */
    log_struct(si, st_blksize, %ld,  );
126
    //  blkcnt_t  st_blocks;  /* number of blocks allocated */
    log_struct(si, st_blocks, %lld,  );

    //  time_t    st_atime;   /* time of last access */
131 log_struct(si, st_atime, 0x%08lx, );

    //  time_t    st_mtime;   /* time of last modification */
    log_struct(si, st_mtime, 0x%08lx, );

136 //  time_t    st_ctime;   /* time of last status change */
    log_struct(si, st_ctime, 0x%08lx, );

}

141 void log_statvfs(struct statvfs *sv)
{
#ifdef NOLOG
  return ;
#endif
146 //  unsigned long  f_bsize;    /* file system block size */
    log_struct(sv, f_bsize, %ld, );

    //  unsigned long  f_frsize;   /* fragment size */
    log_struct(sv, f_frsize, %ld, );
151
    //  fsblkcnt_t     f_blocks;   /* size of fs in f_frsize units */
    log_struct(sv, f_blocks, %lld, );

    //  fsblkcnt_t     f_bfree;    /* # free blocks */
156 log_struct(sv, f_bfree, %lld, );
```

```
    //  fsblkcnt_t     f_bavail;   /* # free blocks for non-root */
  log_struct(sv, f_bavail, %lld, );

    //  fsfilcnt_t     f_files;    /* # inodes */
  log_struct(sv, f_files, %lld, );

    //  fsfilcnt_t     f_ffree;    /* # free inodes */
  log_struct(sv, f_ffree, %lld, );

    //  fsfilcnt_t     f_favail;   /* # free inodes for non-root */
  log_struct(sv, f_favail, %lld, );

    //  unsigned long  f_fsid;     /* file system ID */
  log_struct(sv, f_fsid, %ld, );

    //  unsigned long  f_flag;     /* mount flags */
  log_struct(sv, f_flag, 0x%08lx, );

    //  unsigned long  f_namemax;  /* maximum filename length */
  log_struct(sv, f_namemax, %ld, );

}

void log_utime(struct utimbuf *buf)
{
#ifdef NOLOG
  return ;
#endif
  //    time_t actime;
  log_struct(buf, actime, 0x%08lx, );

  //    time_t modtime;
  log_struct(buf, modtime, 0x%08lx, );
}
```

6.6. params.h

```
/*
  Copyright (C) 2012 Joseph J. Pfeiffer, Jr., Ph.D. <pfeiffer@cs.nmsu.edu>

  This program can be distributed under the terms of the GNU GPLv3.
  See the file COPYING.

  There are a couple of symbols that need to be #defined before
  #including all the headers.
*/

#ifndef _PARAMS_H_
```

```
#define _PARAMS_H_


//No utilizar el log
#define NOLOG
// The FUSE API has been changed a number of times.  So, our code
// needs to define the version of the API that we assume.  As of this
// writing, the most current API version is 26
#define FUSE_USE_VERSION 26

// need this to get pwrite().  I have to use setvbuf() instead of
// setlinebuf() later in consequence.
#define _XOPEN_SOURCE 500

// maintain bbfs state in here
#include <limits.h>
#include <stdio.h>

struct bb_state {
  FILE *logfile;
  char *rootdir;
  void *db; // Base de datos que almacena la información de archivos.
  void *mapopenw; // Mapa que gestiona los archivos abiertos para
     escritura.
};

//Entrada de la base de datos
struct db_entry {
  char path[PATH_MAX];
  char sha1sum[41];
  char datapath[PATH_MAX];
  unsigned int size;
  int deduplicados;
};
//Entrada en el mapa de ficheros abiertos en escritura
struct map_entry {
  unsigned long long int fh;
  char path[PATH_MAX];
  int deduplicados;
  char modificado;
};

#define BB_DATA ((struct bb_state *) fuse_get_context()->private_data)

#endif
```

6.7. Makefile

```
LDFLAGS=`pkg-config fuse openssl sqlite3 --libs`

dedupfs : dedupfs.o log.o database.o
	gcc -g -o dedupfs dedupfs.o log.o database.o $(LDFLAGS)

dedupfs.o : dedupfs.c database.h log.h params.h
	gcc -g -Wall `pkg-config fuse --cflags` -c dedupfs.c

database.o : database.c database.h log.h params.h
	gcc -g -Wall `pkg-config fuse --cflags` -c database.c

bbfs : bbfs.o log.o
	gcc -g -o bbfs bbfs.o log.o `pkg-config fuse --libs`

bbfs.o :  bbfs.c log.h params.h
	gcc -g -Wall `pkg-config fuse --cflags` -c bbfs.c

log.o : log.c log.h params.h
	gcc -g -Wall `pkg-config fuse --cflags` -c log.c

clean:
	rm -f dedupfs *.o
```