




4 DE MAYO DE 2015

DISEÑO DE SISTEMAS OPERATIVOS

SISTEMA DE FICHEROS. PRÁCTICA 3.

Carlos Contreras Sanz	100303562
Alejandro García-Cantarero Alañón	100307006
Álvaro Gómez Ramos	100307009



Contenido

1.	Diseño detallado del sistema de ficheros implementado	3
1.1.	Inodos	3
1.2.	Superbloque	4
1.3.	Consideraciones sobre el modelo elegido	5
1.4.	Funcionamiento general	6
1.4.1.	Snapshots	6
1.4.2.	trickUsuario	7
2.	Descripción del código	8
2.1.	Filesystem.c	8
2.1.1.	mkFS()	8
2.1.2.	mountFS()	9
2.1.3.	unmountFS()	9
2.1.4.	undoFS()	9
2.1.5.	creatFS()	10
2.1.6.	openFS()	10
2.1.7.	closeFS()	10
2.1.8.	closeFS()	10
2.1.9.	writeFS()	11
2.1.10.	lseekFS()	11
2.1.11.	actualizaBitmapCreate()	11
2.1.12.	actualizaBitmapUndo()	12
3.	Batería de pruebas	13
3.1.	mkFS	13
3.1.1.	mkFS con parámetros válidos	13
3.1.2.	mkFS con parámetros inválidos	13
3.2.	mountFS	13
3.2.1.	mountFS válido	13
3.2.2.	mountFS no válido	14
3.3.	creatFS	14
3.3.1.	creatFS válido	14
3.3.2.	creatFS repetido	14
3.3.3.	Numero de creaciones superior a máximo de ficheros	15
3.4.	openFS	15
3.4.1.	openFS válido	15

3.4.2.	openFS en fichero ya abierto	16
3.4.3.	openFS a un archivo inexistente	16
3.5.	readFS.....	16
3.5.1.	readFS válido	16
3.5.2.	readFS en fichero cerrado	17
3.6.	writeFS.....	17
3.6.1.	writeFS válido	17
3.6.2.	writeFS en fichero cerrado	18
3.7.	closeFS.....	18
3.7.1.	closeFS válido	18
3.7.2.	closeFS en fichero cerrado	19
3.8.	lseekFS.....	19
3.8.1.	lseekFS válido	19
3.8.2.	lseekFS en fichero cerrado	20
3.8.3.	lseekFS con parámetro erróneo	20
3.9.	umountFS	20
3.9.1.	umountFS válido.....	20
3.10.	undoFS.....	21
3.10.1.	undoFS múltiple válido.....	21
3.10.2.	Más peticiones undo que undo disponibles	21
3.10.3.	Undo post umount	22
4.	Conclusiones.....	22

Este documento presenta y describe la solución implementada por los autores del documento (Carlos, Alejandro y Álvaro) para la tercera práctica de la asignatura de Diseño de Sistemas Operativos.

En ella se pedía que dado el código de inicio proporcionado por los profesores de la asignatura, que simulaba de forma virtual un dispositivo de almacenamiento, se desarrollase un sistema de ficheros que cumpliese una serie de características concretas.

Por tanto el trabajo ha consistido en el diseño de estructuras de control y datos necesarios para nuestro sistema de ficheros, así como el desarrollo de las funcionalidades requeridas del mismo.

Toda la implementación realizada se ha llevado a cabo en el archivo filesystem.c.

1. Diseño detallado del sistema de ficheros implementado

Para empezar, se decidió partir de un sistema con un superbloque y una serie de inodos.

La razón de esto es que el superbloque tan sólo ocupa un bloque, y antes de intentar comprimir toda la información (de inodos + del sistema de ficheros) en un solo bloque, por simplicidad del diseño se ha elegido “perder” un bloque para datos generales (superbloque) y otro para los inodos.

Para el diseño de las estructuras de datos y control partimos de la base de que se tenían bloques de 4096 bytes, tal y como se indicaba en el enunciado de la práctica, de forma que esa cantidad de datos era la que se podía guardar en un bloque.

Además, se ha de tener en cuenta durante la implementación, que la solución dada para las snapshots ha consistido en copiar el inodo y el bloque de datos de aquello que debemos de guardar para snapshot y tenerlo presente en disco mientras sea necesario, por lo que siempre necesitaremos 10 inodos y 10 bloques de datos extras, además de los que nos solicite el usuario.

También, tener en cuenta que se hace uso del bloque 0 del “disco” para el superbloque y del 1 para el bloque de inodos.

1.1. Inodos

Para guardar los inodos se ha hecho uso de un solo bloque de “disco”, es decir, de 4096 bytes.

Se ha decidido guardar la mínima información necesaria de cada bloque/archivo en cada uno de estos inodos. Cada inodo por tanto está compuesto por:

- **char nombre [64]:** Un array de 64 char, para guardar el nombre del fichero que representa ese inodo (no se hace uso de \0 para terminar la cadena de texto, para economizar espacio, de forma que se tendrá en cuenta esta implementación a la hora de usar las cadenas en el resto de la solución).
- **unsigned char bloqueDato:** Un dato de tipo char que se usa como indicativo de en qué bloque están los datos de éste inodo. Éste tipo char se usa por tanto como número, entre 0 y 255 (char es de tamaño byte, o lo que es lo mismo 2^8), siendo un rango más que suficiente para representar los 50 archivos máximos más los 10 necesarios para las snapshots.

De esta forma, tenemos que para guardar todos los posibles archivos se necesitan 50 inodos (uno por cada posible archivo) más 10 inodos para guardar las 10 snapshots a las que hay que dar soporte.

De esto forma tenemos un array de 60 posiciones de tipo estructura inodo (con los datos antes descritos) en un bloque de 4096 bytes.

$$60 \text{ posiciones} * (64 \text{ char del nombre} + 1 \text{ char para el numero de bloque}) \\ = 3900 \text{ bytes}$$

Como vemos, de los 4096 bytes libres en cada bloque, nosotros tendremos sin ocupar 196 bytes, incluso aunque nos pidan crear un sistema de archivos con el máximo número posible de ficheros (50).

Para guardar estos inodos en disco, se crea un array de 60 posiciones, teniendo en cada una de ellas una estructura con los dos datos antes descritos. Después haciendo uso de memcpy escribimos en el bloque 1 del disco el array de inodos (habiendo reseteado el bloque a 0 antes).

1.2. Superbloque

Este superbloque se ha definido como una estructura, que se guardará en el bloque 0 del disco. Esta estructura tiene los siguientes atributos:

- **unsigned int numeroMagico**: Número usado para reconocer si (al montar un sistema de archivos) se trata del nuestro o no, de forma que sabremos si seremos capaces de leerlo o no.
- **unsigned int numeroMaximoFicheros**: Número máximo de ficheros que podrá almacenar el disco (dado el enunciado y nuestra implementación, vendrá dado por ser el límite impuesto en el mkFS()).
- **unsigned int numeroDeFicheros**: Número de ficheros creados en el disco actualmente, sin contar snapshots. Con esto podremos llevar el control de cuando ficheros creados tenemos, para dar error cuando se intenten crear más de los posibles.
- **unsigned int sizeDispositivo**: Tamaño del dispositivo en bytes.
- **unsigned int numeroBloquesDatos**: Número de bloques del disco que serán destinados a datos. Puede representar la memoria disponible.
- **unsigned int primerBloqueDatos**: Número que indica en que bloque comienzan los bloques de datos.
- **unsigned int bloqueInodos**: Número que indica en que numero de bloque se encuentran los inodos.
- **char mapaInodos [60]**: Array que representa los 60 inodos del sistema, y se usará para ver cuáles hay libres (estarán como 'L'), para llevar el control de las snapshots (marcadas de '0' a '9') y para ver cuales ya tienen sus datos permanentes (marcados con 'P').
- **char mapaDatos[60]**: Array que representa los 60 bloques de datos, y en que se llevara en control de cual está ocupado (1) y cual no(0).
- **char trickUsuario[60]**: Array que se usara para calcular, de forma interna, el bloque correcto, dado un descriptor que nos pasa el usuario. Se usa, dad nuestra implementación, en los casos en los que se realizan varias acciones sobre un mismo archivo, ya que con cada nueva acción se hace copia de inodo y bloque asociados (para las snapshots), y el número de bloque (e inodo) al que se debe acceder cambia).
- **TipoInfoInodo inodoInfo [60]**: Array de tipo estructura que contiene un int para el tamaño del dato de cada bloque, y otro int como puntero de posición de escritura/lectura dentro del bloque. Estos datos son necesarios a la hora de leer y escribir en los bloques/archivos.

De esta forma guardamos en este superbloque la información necesaria para el sistema de archivos (por ejemplo el número mágico, para reconocer y verificar el sistema de archivos), guardamos también datos para el control de las snapshots y otros datos de utilidad cuyo uso será explicado en el apartado de descripción del código.

1.3. Consideraciones sobre el modelo elegido

Se ha elegido este modelo de datos y estructuras de control, ya que es el que se ha considerado más adecuado.

Somos plenamente conscientes de que incluso sobre ésta solución hay varias mejoras posibles (si atendemos solamente a la optimización del espacio), que podrían incluir:

- **unsigned int numeroMagico:** Si es necesario, ocupa *4 bytes*.
- **unsigned int numeroMaximoFicheros:** No es necesario, si creásemos sólo los inodos necesarios no haría falta, sabiendo número de inodos, sabemos número de ficheros.
- **unsigned int numeroDeFicheros:** No es necesario, basta con recorrer el array de mapalnodos para saber si hay alguno libre o no.
- **unsigned int sizeDispositivo:** No es necesario, ya que no se usa.
- **unsigned int numeroBloquesDatos:** No es necesario, ya que siempre es fijo en nuestra implementación.
- **unsigned int primerBloqueDatos:** No es necesario, ya que siempre es fijo en nuestra implementación.
- **unsigned int bloqueInodos:** No es necesario, ya que siempre es fijo en nuestra implementación.
- **char mapaInodos [60]:** Si es necesario, para controlar ocupado/libre, y las snapshots. Ocuparía *60 bytes*.
- **char mapaDatos[60]:** Dado que en esta solución un inodo y un bloque de datos son iguales, no sería necesario.
- **char trickUsuario[60]:** Si sería necesario, pero quizá implementado de otra forma, y solo de longitud 10 (10 snapshots, 10 cambios de descriptor real como máximo) ocuparía *10 bytes*. Incluso puede ser algo de la sesión, y no estar en el superbloque.
- **TipoInfoInodo inodoInfo [60]:** Sería necesario conocer el tamaño de los archivos, pero no es necesario conocer el puntero de lectura/escritura, si consideramos que no es persistente entre sesiones. Además para guardar tamaño de archivo, basta con usar datos de tipo short, usando sólo 2 bytes, en lugar de 4. Por tanto ocuparía 120 bytes.

De esta forma, podríamos contener todo (datos de “superbloque” más inodos) en un solo bloque, ya que ahora el “superbloque” solo ocuparía $4+60+10+120 = 194 \text{ bytes}$, y recordemos que había 196 bytes libres en el bloque de inodos.

Aun siendo conscientes de esto, hemos preferido hacer uso de un bloque en exclusiva para el superbloque, primero porque a pesar de no ser necesarios todos los datos es más visual, y segundo porque por mucho que sea nuestra propia implementación, el hecho de “perder” un bloque para el superbloque no es algo que tenga demasiada importancia, ya que dados los parámetros del guion de la práctica, siempre nos es posible crear el número máximo de ficheros (que en ningún caso será mayor de 50), y consideramos que esta es una mejor práctica.

1.4. Funcionamiento general

En este apartado se pretende explicar el funcionamiento de las partes de la implementación que consideramos más peculiares, y de más difícil comprensión.

El primer comentario sería que en la implementación realizada siempre tenemos los 50+10 inodos creados, y sus estructuras de datos, estén o no llenas, y controlamos el número de archivos creados y el máximo permitido con los metadatos del superbloque.

Además, antes de pasar a la explicación de lo más raro de nuestra implementación, destacar que entre los 60 inodos/archivos máximos, guardamos los 10 de snapshot, con la única distinción de los arrays `mapalnodos` y `trickUsuario`.

Tras leer las explicaciones de más abajo, se verá que al tener snapshots y datos mezclados, puede parecer un poco complejo y propenso a errores, pero no es así, y en todo momento somos conscientes de que archivo es el que debemos devolver al usuario y sobre el que tenemos que trabajar. De esta forma para cada nueva acción deberemos crear nuevos inodos/datos y actualizar un par de arrays.

Las partes descritas en este apartado son la implementación de las snapshots y el uso del `trickUsuario`.

1.4.1. Snapshots

El funcionamiento de las snapshots es el siguiente:

Cada vez que se produce un write se crea una copia del inodo con la información más actual de ese archivo, y a partir de ese momento se trabaja sobre la nueva copia, y si se produjese otro write, se haría la copia sobre este último.

Nuestra implementación hace uso del array `mapalnodos` para controlar las snapshots (además de cuales inodos hay libres, y cuales ya son permanentes y no puede deshacerse esa acción).

En éste array guardamos en cada posición un carácter, que representa el estado del inodo de esa posición, de forma que `mapalnodos[5]='L'`, quiere decir que el inodo 5 (el mismo inodo en la posición 5 del array de inodos contenido en el bloque 1 del disco, y en el array `inodos` de memoria) está libre, y puede ser usado.

Haciendo uso de ese array implementamos las snapshots de manera que al escribir o crear algo nuevo se guarda esa operación con un nuevo inodo y un nuevo bloque de datos. Ese inodo usado para la nueva operación recibe un '0' en el array `mapalnodos`.

Si yo realizo otra operación similar (otro write del mismo archivo), la posición de `mapalnodos` que tuviese un '0' se convertiría en un '1', para indicar que ya no es la última operación que se ha realizado, y la que es realmente la última tendría un '0' en su posición del array `mapalnodos`. Esto será así hasta llegar a '9'.

Si nos encontramos con algo a '9', lo que deberemos hacer es eliminar todo lo que tenga un mismo nombre y estado 'P', o permanente, para poder pasar el '9' a ser la única copia permanente de ese archivo, y de esa forma no agotar la memoria.

De forma que hacer un undo, será realmente eliminar todos los datos relacionados con el inodo cuya posición del array `mapalnodos` tengan un '0'. Si se trataba de una escritura, tendremos otro

inodo con el mismo nombre, y si era un create, desaparecerá el único inodo cuyo nombre era el de ese archivo.

1.4.2. `trickUsuario`

Este es un array presente en el superbloque, y por tanto persistente. Tiene una longitud de 60 posiciones (el máximo número de archivos que serán creados) y guarda caracteres, que en realidad usaremos como bytes (como números).

En cada una de sus posiciones se guarda el inodo que realmente corresponde a un determinado descriptor.

Es decir, al montar el disco y abrir un fichero, se da un descriptor al usuario. Este descriptor será durante toda esa sesión el inodo que contenía los datos más actuales de un archivo en el momento de hacer el open.

Supongamos que creamos un archivo y se le asigna el inodo 12, entonces `trickUsuario` tendrá en la posición 12, un 12. Al abrir el fichero, al usuario se le devolverá como descriptor un 12. Si justo después se hace un write, por el funcionamiento de las snapshots se pasarán los datos más actuales a, por ejemplo, el inodo 15. En ese caso, la versión más reciente del fichero sobre el que estamos tratando, estará en el inodo 15, y el array `trickUsuario` se actualiza escribiendo un 15 en su posición 12, y un 15 en su posición 15.

De esta forma, cada vez que el usuario haga algo sobre el descriptor 12, nosotros recorreremos el array `trickUsuario` hasta obtener en qué posición el índice es igual que lo que guarda.

Si con la situación actual se desmontase el disco, montase y volviese a abrir el fichero, el descriptor devuelto sería el más actual en ese momento, es decir el 15.

No presenta un problema con las snapshots ya que claramente se especifica que deben hacerse con todos los archivos cerrados, por lo que el usuario no tendría ningún descriptor.

Además, de esta forma, se llevan de forma paralela (pero incompleta) las snapshots, y accedemos rápidamente a la última versión de un archivo, sin tener que consultar por un lado al `mapaInodos` y por otro el nombre de los archivos, para encontrar el archivo correcto.

Si se hiciese un undo, siguiendo con este ejemplo, la posición 15 pasaría a borrarse, y la posición 12 pasaría a ser 12 otra vez.

2. Descripción del código

En este apartado se procederá a describir en detalle la implementación de las distintas funciones de la solución.

2.1. Filesystem.c

Es el único archivo que se ha modificado en esta solución. En él se encuentran desarrolladas las funciones dadas en el enunciado, y además, como parte de nuestra solución se hace uso de los siguientes datos globales:

- **static TipoSuperBloque sb**: Estructura a la que se copia el superbloque al montar el dispositivo. Se trabaja con estos datos, hasta el momento de desmontar el dispositivo, es entonces cuando se sobrescribe el superbloque de disco por éste.
- **static TipoInodo inodos[60]**: De la misma forma que con el superbloque, array de inodos para trabajar con ellos en memoria.
- **static TipoDescriptor descFichero [60]**: Array que almacena el estado de los ficheros (abierto con un 1 y cerrado con un 0).

De esta forma se trabajara con el superbloque y los inodos en memoria, y se guardaran en disco solo cuando se desmonte el dispositivo (y se cumple el requisito de guardar los metadatos de forma persistente). También tenemos el array para guardar el estado (abierto o cerrado) de los archivos, para controlar cuando se puede hacer el undoFS() (de acuerdo al anunciado, solo si todos los archivos están cerrados).

A continuación se analizarán una por una las distintas funciones presentes en el archivo:

2.1.1. mkFS()

Ésta es la función encargada de crear el disco, para ello recibe dos parámetros, maxNumFiles que indica el número máximo de ficheros que podrá contener el dispositivo, y deviceSize para indicar en bytes el tamaño del disco.

Lo primero que se comprueba en este caso es que el tamaño solicitado se encuentra dentro del rango valido (>320KB y <500KB), y que el número de ficheros máximo que deberá tener el dispositivo esta entre 1 y 50. Cabe destacar que dada nuestra implementación siempre seremos capaces de dar soporte al número de archivos que nos soliciten dentro del límite del enunciado, ya que en el peor de los casos tendremos que 320KB dan para 80 bloques de 4096 bytes, por lo que nosotros usaríamos 2 de metadatos, 10 de snapshot y 50 para datos (el máximo), sobrando 12 bloques.

Una vez se han verificado los parámetros se procede a crear el superbloque se inicializan los datos:

El número mágico es el definido por nosotros como constante, por ejemplo. Destacar que el mapa de inodos se inicializa a 'L', para marcar que todas las posiciones están libres, de la misma forma que se inicializa a 0 el mapa de datos o el trickUsuario. El array de información de inodos (con tamaño y posición de lectura/escritura tiene a 0 también cada uno de sus datos).

Tras eso nos aseguramos de que en el bloque 0 solo haya ceros y escribimos el superbloque al comienzo del disco (bloque 0).

Tras eso se hace lo propia creando el array de inodos, llenando sus datos inicialmente a 0, vacios, y se escribe en el bloque 1 del disco (habiendo reseteado dicho bloque antes).

En todo momento se hacen las comprobaciones necesarias del correcto funcionamiento del proceso, retornando 0 en caso satisfactorio o -1 en caso contrario.

Algo que no se ha tenido en cuenta (ya que no se pide) es la comprobación de que no se puede hacer varias veces mkFS() sobre el mismo archivo, pero bastaría con leer el bloque 0 y comprobar que donde debería estar el número mágico, hay algo diferente de nuestro número esperado. Esto querría decir que sea lo que sea lo que hubiese ahí, no sería nuestro sistema de ficheros, y por tanto podríamos crearlo.

2.1.2. mountFS()

Esta función se encarga de montar un disco ya creado, que implementa nuestro sistema de archivos, para su uso.

Lo primero es leer el bloque 0 de dicho “disco”. De lo leído, obtenemos la posición donde nosotros hubiésemos guardado el número mágico, y comparamos lo leído con lo esperado. Si no fuese así devolveríamos -1 (al igual que si hubiese habido problemas con la lectura).

Si ese fuese efectivamente nuestro sistema de archivos, leeríamos el bloque 1 (el 0 del superbloque ya lo habríamos leído antes), y guardaríamos ambos bloques en memoria para trabajar con ellos mientras tuviésemos el disco montado.

Si todo se ha ejecutado correctamente devolveremos 0.

2.1.3. unmountFS()

En este caso se realiza la operación opuesta a la de la función anterior. Ahora se copian de memoria a disco, tanto el superbloque como el array de inodos.

Si todo se ha ejecutado correctamente devolveremos un 0.

Algo que no se contempla en la solución es desmontar el disco sólo si todos los archivos han sido cerrados, pero como no se pide, no se ha implementado. Si se desease incluir dicha funcionalidad, bastaría con recorrer el array descFichero, asegurándonos de que no hubiese ningún 1 en sus posiciones, y permitiendo desmontar en disco sólo en ese caso.

Tampoco se comprueba que se desmonte un disco montado (es decir, se pueden hacer, por ejemplo, dos unmountFS() seguidos). Esto no se ha tenido en cuenta porque no era una funcionalidad requerida, pero basta con tener una variable global en la que se guarde si el disco está montado o no.

2.1.4. undoFS()

Esta es la función encargada de deshacer las operaciones, en orden inverso, hasta un máximo de 10.

Esta función comienza comprobando que están todos los archivos cerrados, recorriendo el array de descFichero y viendo que todas sus posiciones están a 0.

Si esto es así encuentra en el array de mapalNodos el índice del inodo que se debe eliminar (en caso de ser posible, si no, retorna -1). Una vez obtenido cual es el inodo que debemos borrar, procedemos a hacerlo: escribimos en mapalNodos que está libre ('L'), decimos en su mapa de datos que está libre (0), decimos en inodoInfo que ocupa 0, y que su posición de lectura/escritura es 0.

Además se actualiza el array de `trickUsuario`, que recordemos era aquel que encadenaba punteros. Se actualiza de forma que la posición de `trickUsuario` que “apuntaba” al inodo que se ha borrado apunte ahora a ‘0’, y la que apuntaba a esa apunte a sí misma, de forma que esa se convierta en la última versión de ese archivo (ver explicación del punto 1.4.1).

Si se ha deshecho un create (se sabe porque no había nadie en `trickUsuario` que apuntase a esa posición) se decrementan el número de ficheros.

Por último se llama a `actualizaBitapUndo()` en el return, devolviendo el número de undo que se pueden hacer.

2.1.5. `creatFS()`

Esta función es la encargada de crear archivos. Para ello primero comprueba que se hay espacio disponible, y que el nombre del archivo (que recibe por parámetros) no es excesivo (devolviendo -1 si alguna de estas cosas sucediesen). Además comprueba que no haya ningún fichero con el mismo nombre (devolviendo 1 si sucediese).

Si todo es correcto, se obtiene el primer inodo libre (cuya posición en `mapaInodos` esta a ‘L’)

En ese momento se inicializan todos los datos relacionados con el inodo (`mapaInodos`, `mapaDatos`,..., y se llama a `actualizaBitmapCreate()` para las snapshots), numero de ficheros y se escriben en disco.

Se devuelve 0 si todo es correcto.

2.1.6. `openFS()`

Esta función es la encargada de devolver un descriptor de fichero, dado un nombre de fichero.

Lo primero que se comprueba es que el archivo esté presente. Si no es así se devuelve -1. Si está presente se obtiene el inodo con la última versión del archivo (usando el array `trickUsuario`). Nos fijamos en que ese archivo no esté abierto ya (en cuyo caso devolveríamos -2) y devolvemos su descriptor, además de cambiar su estado a abierto.

2.1.7. `closeFS()`

Esta es la función encargada de cerrar el fichero cuyo descriptor se recibe por parámetros.

Para ello se obtiene el descriptor más reciente de ese archivo y se cierra, devolviendo -1 (si ya lo estuviese, devolveríamos -1).

2.1.8. `closeFS()`

En esta función se lee el número de bytes solicitado de un descriptor de fichero, y se guarda en un buffer.

Para ello lo primero es comprobar cuál es el inodo que guarda la información más actual de ese fichero (consultando `trickUsuario`), y comprobando que está abierto el fichero (devolveríamos -1 en caso contrario).

Después con `bread()` leemos del disco, el bloque del inodo con los datos más actuales, y lo dejamos en un array de char de tamaño 4096 bytes, devolviendo el número de bytes leídos, y actualizando puntero de lectura/escritura.

2.1.9. writeFS()

Esta es la función encargada de implementar las escrituras a disco (sobre un archivo concreto). Para ello se le pasa un descriptor de fichero (del archivo sobre el que se va a escribir), un puntero a los datos a escribir, y el tamaño de datos a escribir.

Lo primero (tras comprobar que efectivamente ese descriptor se corresponde con un inodo con datos), es encontrar cual es el inodo con los datos más actuales de ese archivo (recordemos el funcionamiento de las snapshots y del trickUsuario, descritos anteriormente).

Una vez lo tenemos, hallamos el primer inodo libre, y sobre ese inodo y su bloque de datos escribimos una copia del inodo y datos sobre los que vamos a escribir.

Después (y tras actualizar el mapa de inodos con `actualizaBitmapCreate()`), sobre esa copia es sobre la que modificaremos los datos.

Escribimos entonces el número de bytes que podamos (hasta el máximo que nos dan por parámetros), actualizamos el dato del tamaño y el puntero de lectura/escritura (si no hubiese habido espacio restante, no se hubiese escrito nada).

Devolvemos, si todo ha ido bien, el número de bytes que se han escrito, si no un -1.

Cabe destacar que no se ha implementado la comprobación de que un archivo deba estar abierto/cerrado para escribir en el (no se pedía en el enunciado), pero bastaría hacerlo de la misma forma que se hace en el `read`.

2.1.10. lseekFS()

En este caso se implementa la función que permite mover el puntero de lectura/escritura de un fichero. Para ello se reciben por parámetros el descriptor de fichero, el nuevo valor del puntero, y el tipo de posicionamiento.

Lo primero es, al igual que en otras ocasiones obtener el inodo con los datos de fichero más actuales, comprobando que efectivamente hay un archivo que se corresponde con ese descriptor.

Tras eso, simplemente se actualiza el atributo de puntero del array `inodoInfo` (presente en el superbloque), en función de lo recibido por parámetros.

Si todo ha salido bien, se devuelve el nuevo puntero, si no, se devuelve -1.

Nótese que los `lseekFS()` no se deshacen, ya que no se pide en el enunciado.

2.1.11. actualizaBitmapCreate()

Esta función es artificial, creada por nosotros, y se usa para actualizar correctamente el `mapaInodos`, para llevar la cuenta de snapshots y orden (al incrementar snapshots, es decir, al leer y escribir).

Su funcionamiento ya ha sido brevemente esbozado en la explicación sobre snapshots, y básicamente consiste en incrementar todos aquellos cuyo valor sea '8' o inferior, para dejar el '0' libre.

En caso de que haya uno con valor '9', lo que se hace es buscar todos aquel inodo que contenga una versión permanente ('P') del mismo archivo, y en caso de existir se borra. De esta forma se deja tan solo una versión de cada archivo de forma permanente en memoria, y nos aseguramos

de tener siempre disponibles los bloques de datos para archivos, y de no ocupar espacio innecesariamente.

Se borra el que se encuentra como 'P' (con el mismo nombre de archivo, que es identificador univoco) ya que si tengo uno como 'P' y otro como '9', quiere decir que ese '9' es una versión más reciente, y es en realidad a la que yo estaría accediendo al hacer read/write/lseek, en lugar de a la que fuese 'P'.

Devuelve el número de snapshots almacenadas.

2.1.12. `actualizaBitmapUndo()`

Esta última función se usa para decrementar los números que guarda el array `mapalnodos` (se produce al hacer `undo`).

De esta forma, el '0' sería el que se desharía, y los demás, pasarían a estar más cerca del '0' (el '9' pasaría a ser el '8', etc.).

Se usa solo al hacer `undo`, y “deja” una posición de snapshot libre, de forma que una escritura tras un `undo`, nunca producirá el borrado que se comentaba en la función anterior, ya que no habrá nada con índice '9'.

Devuelve el número de snapshots almacenadas.

3. Batería de pruebas

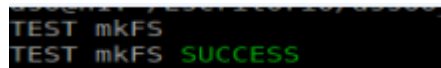
En las siguientes pruebas comprobaremos que la funcionalidad de las pruebas se ciñen a los requisitos dados por los profesores en la memoria disponible.

3.1. mkFS

En estas primeras pruebas comprobaremos que el comportamiento del *mkFS* es el correcto. Para ello realizaremos varias pruebas: una primera en la que introduciremos los parámetros correctamente, y se ejecute; y otras en las que los parámetros introducidos no serán correctos, y, por tanto, no se permitirá su ejecución.

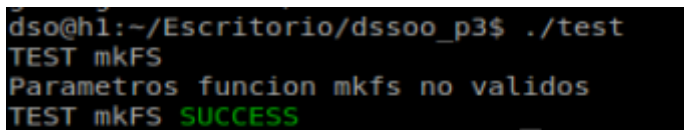
3.1.1. *mkFS* con parámetros válidos

En la siguiente prueba modificaremos el archivo *test.c*, de modo que realizaremos una simple llamada al método *mkFS*, introduciendo como parámetros unos valores válidos, entre 0 y 51 (ambos no incluidos), respecto al número de ficheros máximo, y un tamaño de memoria entre 327680 y 512000 bytes. En el caso que no devuelva el valor esperado (0, ya que se debe realizar bien), se imprimirá un *fail* en rojo. Si va bien, un *SUCCESS* en verde. Como podemos observar en la captura de la ejecución de la prueba, ha ido bien, por lo que se pasa esta prueba.



3.1.2. *mkFS* con parámetros inválidos

En estas pruebas modificaremos el archivo *test.c*, de modo que realizaremos de nuevo una llamada al método *mkFS*, introduciendo como parámetros unos valores inválidos, con los valores 0 y 51 respecto al número de ficheros máximo, y un tamaño de memoria de 327679 y 512001 bytes. En el caso que no devuelva el valor esperado (-1, ya que debe saltar error), se imprimirá un *fail* en rojo. Si va bien, un *SUCCESS* en verde. Como podemos observar en la captura de la



ejecución de la prueba, ha ido bien, por lo que se pasa esta prueba. Adjuntamos solo el resultado de una de las pruebas, pues es el mismo en todas las restantes.

3.2. mountFS

En las siguientes pruebas comprobaremos que el comportamiento del *mountFS* es el correcto. Para ello realizaremos varias pruebas: una primera en la que introduciremos los parámetros correctamente, y se ejecute; y otra en la que no se realizará correctamente la llamada al método *mkFS*, y, por tanto, no se permitirá la correcta ejecución del *mountFS*.

Cabe destacar la posibilidad de añadir otra prueba, la cual sería usar un sistema de ficheros no válido, pero no es posible la realización de esa prueba con el material dado.

3.2.1. *mountFS* válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0, indicando su correcta ejecución, se imprimirá una traza de *SUCCESS* verdes.

Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
```

3.2.2. mountFS no válido

Esta otra prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros inválidos, de modo que no será posible su ejecución, y posteriormente se llame a *mountFS*; y no se pueda montar el archivo *disk.dat*. Es necesario que, antes de realizarla, se vacíe el archivo *.dat* anterior, porque se mantienen en él los datos de las anteriores pruebas, falseando los resultados de esta. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo -1, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
Parametros funcion mkfs no validos
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
```

Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

3.3. creatFS

En las siguientes pruebas comprobaremos que el comportamiento del *creatFS* es el correcto. Para ello realizaremos varias pruebas: una primera en la que introduciremos los parámetros correctamente, y se ejecute; otra en la que se intentará crear un fichero dos veces, y no nos debe dejar crearlo; y, por último, realizar un número de creaciones de ficheros superior al máximo de ficheros estipulados en el *mkFS*.

3.3.1. creatFS válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro "test.txt", el cual deberá crear un fichero llamado "test.txt". Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0, indicando su correcta ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

```
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
```

3.3.2. creatFS repetido

Esta prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST creatFS
Ya existe un fichero con el mismo nombre
TEST creatFS SUCCESS
```

método *creatFS*, pasando como parámetro "test.txt", el cual deberá crear un fichero llamado "test.txt". Por último, volveremos a llamar al método *creatFS* con el mismo parámetro. Si en cualquier caso devuelven un valor distinto al

esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0 excepto en el último caso, el cual deberá devolver 1, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

3.3.3. Numero de creaciones superior a máximo de ficheros

Por último, modificaremos el archivo *test.c*, de manera que primero llame al *mkFS*, estableciendo como máximo 3 ficheros posibles en nuestro sistema, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS* en 4 ocasiones, pasando como parámetros nombres distintos de ficheros. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0, excepto el último fichero, el cual deberá devolver -1, ya que se ha sobrepasado el número máximo de ficheros, indicando su correcta ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente. Hemos añadido unas trazas para comprobar que en los 3 primeros casos se ha creado correctamente, y en el cuarto no ha sido creado nada.

```
dso@h1:~/Escritorio/dss00_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
creado con exito
TEST creatFS SUCCESS
TEST creatFS
creado con exito
TEST creatFS SUCCESS
TEST creatFS
creado con exito
TEST creatFS SUCCESS
TEST creatFS
```

3.4. openFS

En las siguientes pruebas comprobaremos que el comportamiento del *openFS* es el correcto. Para ello realizaremos varias pruebas: una primera en la que introduciremos los parámetros correctamente, y se ejecute; otra en la que se intentará abrir un fichero dos veces, y no nos debe dejar abrirlo la segunda vez; y por último realizar un *openFS* a un fichero que no existe en nuestro sistema.

3.4.1. openFS válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro "test.txt". Por último, llamaremos al método *openFS*, pasando como parámetro el mismo que hemos pasado al *creat*; abriendo el archivo creado anteriormente. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0, a excepción del último, que devolverá el descriptor del fichero abierto, lo que indicaría su correcta ejecución; se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente

```
dso@h1:~/Escritorio/dss00_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
```


3.4.2. openFS en fichero ya abierto

Esta prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro "test.txt", y llamaremos al método *openFS*, pasando por parámetro el mismo parámetro del *creatFS*, que abrirá ese fichero. Por último, volveremos a llamar al método *openFS* con el mismo parámetro. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0 excepto en el

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
Fichero ya abierto
TEST openFS SUCCESS
```

último caso, el cual deberá devolver -2, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente. Hemos añadido una traza que indique el error.

3.4.3. openFS a un archivo inexistente

Por último, modificaremos el archivo *test.c*, de manera que primero llame al *mkFS*, estableciendo como máximo 3 ficheros posibles en nuestro sistema, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *openFS*, pasando como parámetro el nombre de un fichero que no existe. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0, excepto el último fichero, el cual deberá devolver -1, ya que no se ha encontrado el fichero a abrir, indicando su correcta ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST openFS
TEST openFS SUCCESS
```

3.5. readFS

En las siguientes pruebas comprobaremos que el comportamiento del *readFS* es el correcto. Para ello realizaremos varias pruebas: una primera en la que introduciremos los parámetros correctamente, y se ejecute; y otra en la que se intentará leer de un fichero el cual no ha sido abierto anteriormente.

3.5.1. readFS válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro "test.txt" y llamaremos al método *openFS*, pasando como parámetro el mismo que hemos pasado al creat. Por último, llamaremos al método *readFS*,

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
TEST readFS
TEST readFS SUCCESS
```

pasando por parámetro el descriptor devuelto por el `open`, el buffer en el que queremos guardar los datos leídos, y el número de datos que queremos leer. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo el número de bytes leídos devueltos en la llamada *readFS*, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente

3.5.2. *readFS* en fichero cerrado

Esta prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro “test.txt”, y llamaremos al método *readFS*, pasando por parámetro un descriptor cualquiera. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0 excepto en el último caso,

```
dso@hl:~/Escritorio/dss00_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST readFS
TEST readFS SUCCESS
```

el cual deberá devolver -1, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

3.6. *writeFS*

En las siguientes pruebas comprobaremos que el comportamiento del *writeFS* es el correcto. Para ello realizaremos las mismas pruebas que las realizadas en el *read*: una primera en la que introduciremos los parámetros correctamente, y se ejecute; y otra en la que se intentará escribir en un fichero el cual no ha sido abierto anteriormente.

3.6.1. *writeFS* válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro “test.txt” y llamaremos al método *openFS*, pasando como parámetro el mismo que hemos pasado al *creat*. Por último, llamaremos al método *writeFS*, pasando por parámetro el descriptor devuelto por el *open*, el buffer del que queremos escribir los datos, y el número de datos que queremos escribir. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo el número de bytes escritos devueltos en la llamada *writeFS*, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente

```
dso@hl:~/Escritorio/dss00_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
TEST writeFS
TEST writeFS SUCCESS
```

3.6.2. writeFS en fichero cerrado

Esta prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro “test.txt”, y llamaremos al método *writeFS*, pasando por parámetro un descriptor cualquiera. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo 0 excepto en el último caso,

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST writeFS
TEST writeFS SUCCESS
```

el cual deberá devolver -1, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

3.7. closeFS

En las siguientes pruebas comprobaremos que el comportamiento del *closeFS* es el correcto. Para ello realizaremos las siguientes pruebas: una primera en la que introduciremos los parámetros correctamente, y se ejecute; y otra en la que se intentará cerrar un fichero el cual ya ha sido cerrado anteriormente.

3.7.1. closeFS válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro “test.txt” y llamaremos al método *openFS*, pasando como parámetro el mismo que hemos pasado al creat. Por último, llamaremos al método *closeFS*, pasando por parámetro el descriptor devuelto por el open, ya que es el que queremos cerrar. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo.

En caso que devuelva todo correctamente, incluyendo el 0 de la correcta ejecución del *closeFS*, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente

```
dso@hl:~/Escritorio/dssoo_p3$ ./tes
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
TEST closeFS
TEST closeFS SUCCESS
```

3.7.2. closeFS en fichero cerrado

Esta prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro "test.txt". Acto seguido procederemos a abrirlo, y llamaremos al método *closeFS*, pasando por parámetro el descriptor anterior. Por último, volveremos a llamar al *closeFS* pasando el mismo descriptor, y no nos debería dejar cerrarlo, pues ya está cerrado. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente incluyendo en el último caso, el cual deberá devolver -1, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente. Hemos añadido una traza para demostrar la ejecución.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
TEST closeFS
TEST closeFS SUCCESS
TEST closeFS
Descriptor de fichero ya esta cerrado
TEST closeFS SUCCESS
```

3.8. lseekFS

En las siguientes pruebas comprobaremos que el comportamiento del *lseekFS* es el correcto. Para ello realizaremos las siguientes pruebas: una primera en la que introduciremos los parámetros correctamente, y se ejecute; otra en la que se intentará realizar en un fichero el cual no ha sido abierto, y otra en la que el puntero apuntará a una posición errónea.

3.8.1. lseekFS válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro "test.txt" y llamaremos al método *openFS*, pasando como parámetro el mismo que hemos pasado al creat. Por último, realizaremos una escritura en el fichero abierto anteriormente, la cual modificará la posición del puntero del fichero, y llamaremos al método *lseekFS*, pasando por parámetro el descriptor del fichero, la posición a la que queremos dejar el puntero, y el modo en el que queremos realizar el lseek. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo el valor de la posición del puntero actualizada de la correcta ejecución del *lseekFS*, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
TEST writeFS
TEST writeFS SUCCESS
TEST lseekFS
TEST lseekFS SUCCESS
```

3.8.2. lseekFS en fichero cerrado

Esta prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro “test.txt”, y llamaremos al método *openFS*, pasando por parámetro el descriptor anterior. Por último, llamaremos al *closeFS* pasando el mismo descriptor, y finalmente al *lseekFS* con el descriptor anterior, el cual ya está cerrado y no se nos debe permitir realizar esa acción. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo en el último caso, el cual deberá devolver -1, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente. Hemos añadido una traza para demostrar la ejecución.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
TEST closeFS
TEST closeFS SUCCESS
TEST lseekFS
Descriptor de fichero no abierto
TEST lseekFS SUCCESS
```

3.8.3. lseekFS con parámetro erróneo

Esta última prueba consistirá en modificar el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación llamaremos al método *creatFS*, pasando como parámetro “test.txt”, y llamaremos al método *openFS*, pasando por parámetro el descriptor anterior. Por último, llamaremos al *lseekFS* con el descriptor anterior y pediremos que nos sitúe el puntero en la posición -1, acción que no se nos debería permitir realizar. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo en el último caso, el cual deberá devolver -1, indicando su fallida ejecución, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST openFS
TEST openFS SUCCESS
TEST lseekFS
TEST lseekFS SUCCESS
```

3.9. umountFS

En las siguientes pruebas comprobaremos que el comportamiento del *umountFS* es el correcto. Para ello realizaremos la siguiente prueba: introduciremos los parámetros correctamente, y se ejecutará correctamente.

3.9.1. umountFS válido

En esta prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. Por último, realizaremos una llamada al método *umountFS*, el cual deberá desmontar el *disk.dat*. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo 0 de la correcta realización del umount, se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST umountFS
TEST umountFS SUCCESS
```

3.10. undoFS

En estas últimas pruebas simples comprobaremos que el comportamiento del *undoFS* es el correcto. Para ello realizaremos las siguientes pruebas: comprobaremos el funcionamiento con múltiples llamadas al *undo*, el funcionamiento si realizamos más *undo* de los disponibles, y que si cerramos el sistema, y lo volvemos a abrir, se mantienen los *undo*.

3.10.1. undoFS múltiple válido

En esta primera prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación, crearemos 3 ficheros y, para finalizar, realizaremos 3 llamadas al método *undoFS*, el cual deberá deshacer la última acción realizada. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo el valor que devuelve el *undo* (que representa el número de *undo* restantes disponibles), se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente. Hemos añadido las trazas, tanto de que se deshace, como de la cantidad de *undo* restantes, para observar mejor el funcionamiento.

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST undoFS
deshago create
Valor de ret 2
TEST undoFS SUCCESS
deshago create
Valor de ret 1
TEST undoFS SUCCESS
deshago create
Valor de ret 0
TEST undoFS SUCCESS
```

3.10.2. Más peticiones undo que undo disponibles

En esta segunda prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación, crearemos 3

```
dso@hl:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST undoFS
deshago create
Valor de ret 2
TEST undoFS SUCCESS
deshago create
Valor de ret 1
TEST undoFS SUCCESS
deshago create
Valor de ret 0
TEST undoFS SUCCESS
No existen cambios a deshacer
Valor de ret -1
TEST undoFS SUCCESS
TEST umountFS SUCCESS
```

ficheros y, para finalizar, realizaremos 4 llamadas al método *undoFS*, el cual deberá deshacer la última acción realizada, excepto la última, la cual no se podrá realizar ya que no hay más *undo* disponibles. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo el valor que devuelve el *undo* (que representa el número de *undo* restantes disponibles), que en el último caso será -1, pues ya no quedan disponibles; se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente. Hemos añadido las trazas, tanto de que se deshace, como de la cantidad de *undo* restantes, para observar mejor el funcionamiento.

3.10.3. Undo post umount

En esta última prueba modificaremos el archivo *test.c*, de manera que primero llame al *mkFS* con unos parámetros válidos, de modo que se ejecute correctamente, y posteriormente se llame a *mountFS*; y este método monte correctamente el *disk.dat*. A continuación, crearemos un fichero y desmontaremos, para volver a montar, el *.dat*. Para finalizar, realizaremos una llamada al método *undoFS*, el cual deberá deshacer la última acción realizada, en este caso antes del unmount. Si en cualquier caso devuelven un valor distinto al esperado, se imprime un *FAIL* en rojo. En caso que devuelva todo correctamente, incluyendo el valor que devuelve el *undo* (que representa el número de *undo* restantes disponibles); se imprimirá una traza de *SUCCESS* verdes. Como podemos observar en la siguiente captura, el código ha realizado su función correctamente, pasando la prueba satisfactoriamente. Hemos añadido las trazas de la acción que deshace para comprobar que es la anterior al umount.

```
iso@hl1:~/Escritorio/dssoo_p3$ ./test
TEST mkFS
TEST mkFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST creatFS
TEST creatFS SUCCESS
TEST umountFS
TEST umountFS SUCCESS
TEST mountFS
TEST mountFS SUCCESS
TEST undoFS
deshago create
TEST undoFS SUCCESS
```

4. Conclusiones

Ésta ha sido una práctica, a nuestro modo de ver, con numerosos detalles, y numerosas soluciones posibles, no todas ellas óptimas en términos de espacio (de hecho la nuestra somos conscientes de que es mejorable), pero con las que se puede llegar a implementar la funcionalidad requerida.

Nuestra solución, pese a no ser la mejor optimizada (“desperdiciamos” un bloque), es la que más nos ha gustado, y la que nos ha parecido que implementaba una mejor practica a la hora de programar usando el superbloque.

Hasta llegar a la solución que hemos finalmente implementado, hemos pasado por varios planteamientos teóricos, y hemos visto cambiada la solución varias ocasiones por esos “detalles” que no habíamos tenido en cuenta, pero que si eran necesarios.

Dado que el modo de acercarnos a la práctica ha sido, primero tener claro que queremos hacer, y como queremos hacerlo, a la hora de programar el único problema que nos ha surgido ha sido a la hora de escribir estructuras en disco, y de leer estructuras.

Ese pequeño problema fue resuelto gracias a los correos con los profesores, y el resto de la programación no supuso mayor problema, más allá de los típicos de depuración o despistes varios.

En general, una práctica divertida y con la que hemos disfrutado.