|  |
| --- |
| U. Carlos III de Madrid  Aulas Informáticas Generales |
| Diseño de Sistemas Operativos |
| Práctica 3: Sistema de Ficheros |
|  |
|  |
|  |

|  |
| --- |
|  |

Javier García Lozano – 100304063 – [100304063@alumnos.uc3m.es](mailto:100304063@alumnos.uc3m.es)

Jesús Núñez Cedillo – 100304139 – [100304139@alumnos.uc3m.es](mailto:100304139@alumnos.uc3m.es)

Álvaro Romo Herrero – 100304120 – [100304120@alumnos.uc3m.es](mailto:100304120@alumnos.uc3m.es)

# Índice

[Introducción 3](#_Toc418501014)

[Planteamiento del diseño 4](#_Toc418501015)

[El superbloque 5](#_Toc418501016)

[El bloque de i-nodos 6](#_Toc418501017)

[Los snapshots 8](#_Toc418501018)

[Diseño de las funcionalidades requeridas 11](#_Toc418501019)

[Formateo, montaje y desmontaje 11](#_Toc418501020)

[Creación, apertura y cerrado de ficheros 11](#_Toc418501021)

[Lectura, escritura y reposicionamiento de puntero 12](#_Toc418501022)

[Deshacer cambios 13](#_Toc418501023)

[Implementación 14](#_Toc418501024)

[mfFS 14](#_Toc418501025)

[umountFS 15](#_Toc418501026)

[mountFS 15](#_Toc418501027)

[undoFS 16](#_Toc418501028)

[creatFS 17](#_Toc418501029)

[openFS 17](#_Toc418501030)

[closeFS 18](#_Toc418501031)

[readFS 18](#_Toc418501032)

[writeFS 18](#_Toc418501033)

[lseekFS 19](#_Toc418501034)

[snapshot.c 20](#_Toc418501035)

[Batería de pruebas 21](#_Toc418501036)

[Prueba 1. Creación, apertura y cierre de ficheros 21](#_Toc418501037)

[Prueba 2. Creación máxima de ficheros y undoFS. 22](#_Toc418501038)

[Prueba 3. Revertir escrituras. 23](#_Toc418501039)

[Prueba 4. Probar función writeFS. 24](#_Toc418501040)

[Prueba 5. Probar función writeFS y readFS. 25](#_Toc418501041)

[Prueba 6. Persistencia de datos en el disco 26](#_Toc418501042)

[Conclusiones 27](#_Toc418501043)

# Índice de ilustraciones

[Ilustración 1. Concepto a implementar. 4](#_Toc418500550)

[Ilustración 2. Concepto del superbloque y su relación con los i-nodos. 6](#_Toc418500551)

[Ilustración 3. Concepto del bloque de i-nodos y su relación con los bloques de datos. 8](#_Toc418500552)

[Ilustración 4. Concepto de los snapshots. 10](#_Toc418500553)

[Ilustración 5. Traza de la prueba original. 21](file:///C:\Users\c0a00217.AIG.001\Desktop\Memoria%20v0.7.docx#_Toc418500554)

[Ilustración 6. Creación máxima de ficheros y undo. 22](file:///C:\Users\c0a00217.AIG.001\Desktop\Memoria%20v0.7.docx#_Toc418500555)

[Ilustración 7. Probar la función undoFS 23](#_Toc418500556)

[Ilustración 8. Probar función writeFS 24](#_Toc418500557)

[Ilustración 9. Probar la función writeFS y readFS 25](#_Toc418500558)

[Ilustración 10. Traza tras la primera ejecución. 26](#_Toc418500559)

[Ilustración 11. Traza tras la segunda ejecución. 26](#_Toc418500560)

# Introducción

El objetivo de esta práctica es implementar **un sistema de ficheros simulado** para un dispositivo pequeño, de entre 320KB hasta 500KB. En el sistema, se podrán crear ficheros y leer y escribir sobre ellos. Las especificaciones que deberá cumplir el sistema son los siguientes:

* El número máximo de ficheros se podrá especificar por el usuario, pero nunca superando los 50 ficheros.
* El nombre de un fichero tendrá un tamaño máximo de 64 bytes.
* Los bloques serán de 4KB.
* El tamaño máximo de fichero será de 4KB, lo que implica que cada fichero debe caber en un único bloque de datos.
* Se podrán deshacer hasta diez cambios de creación de ficheros y escritura (snapshots), siempre y cuando los ficheros estén cerrados.
* Tanto los metadatos como los snapshots deberán ser persistentes entre sesiones.
* No se utilizarán directorios.

En el proyecto inicial se ha incorporado el fichero **filesystem.c** para implementar todas las funcionalidades requeridas. También se adjunta el fichero **ufs.c** para utilizar las funciones de acceso a disco, ya definidas previamente.

Si bien se conocen los objetivos que se deben cumplir, no se dispone de un modelo de sistema de fichero en donde podamos comenzar a programar. Por tanto, queda a nuestra elección cómo será y cómo funcionará el sistema de ficheros.

Por tanto, el procedimiento que hemos seguido es el siguiente:

* **Planteamiento del diseño del sistema de ficheros**, en donde definiremos cómo organizaremos los datos y justificaremos decisiones que no vienen explicadas explícitamente en el enunciado.
* **Implementación**, donde se explica cómo se han programado cada una de las funciones de filesystem.c (y otras auxiliares) a partir del diseño previo.
* **Pruebas**, en donde se verifica la efectividad del sistema de ficheros.

# Planteamiento del diseño

Una vez leído y analizado el enunciado de la práctica, hemos realizado un estudio para averiguar qué modelo de sistema de ficheros sería el más apropiado para el proyecto. Al final, hemos decidido emplear un modelo inspirado en UNIX pero con varias simplificaciones y modificaciones para adaptarse mejor a las funcionalidades requeridas.

Nuestro sistema de ficheros contará con un total de 12 bloques destinados a los metadatos, mientras que el resto estarán dedicados a los datos que se escriban en los ficheros.

* El bloque 0 es el **superbloque**, en donde se guardarán información general sobre todo el sistema.
* El bloque 1 es el **bloque de i-nodos**, donde se guardarán la información específica de cada fichero.
* Los bloques 2 a 11 (ambos inclusive) serán los bloques destinados al **almacenamiento de datos de un snapshot generado por escritura de fichero**.
* A partir del bloque 12 se guardarán **los datos de cada fichero**.

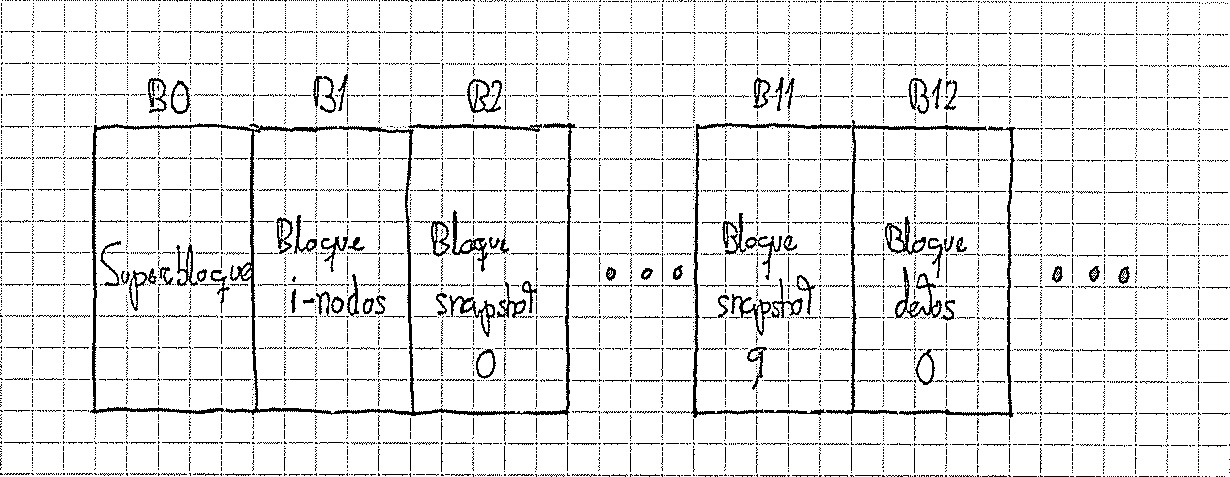


Ilustración 1. Concepto a implementar.

Como podemos apreciar, al tener sólo 12 bloques de metadatos no corremos ningún riesgo de memoria insuficiente al hacer un formateo. En el peor caso posible (un dispositivo de 320KB) dispondríamos de 80 bloques, por lo que siempre nos cabrán los metadatos y al menos un bloque por fichero.

Además, uno de nuestros objetivos de diseño ha sido mantener este reparto de datos de manera estática, para asegurarse de que nunca nos faltará espacio para guardar los metadatos.

## El superbloque

Como hemos mencionado antes, el superbloque contiene información sobre el estado general del sistema de ficheros. Tiene dos objetivos principales:

* Controlar el número y estado de los i-nodos del sistema
* Almacenar el estado de los snapshots

Los datos del superbloque se actualizarán cada vez que se produzca un cambio en el estado de creación/destrucción de ficheros y cuando se disponga de los recursos para volver a un estado anterior del sistema. Cada vez que se “desmonte” el sistema de ficheros, la información del superbloque se almacenará en disco, para que en sesiones sucesivas se pueda recuperar el estado previo.

En concreto, el bloque contará con los siguientes parámetros:

* **inodeNumber**. Número de i-nodos que podremos procesar y, por tanto, cuántos ficheros podrán entrar en el sistema. El usuario podrá especificar el valor de esta variable al realizar un formateo.
* **deviceSize**. Tamaño del dispositivo en bytes. El usuario podrá especificar este valor durante el formateo.
* **iMap**. Array de 50 integers que determina la situación de los i-nodos. Un valor 0 indica que hay un espacio disponible, mientras que 1 determina si hay un i-nodo creado en esa posición. Cada una de las posiciones del array se actualiza cuando el i-nodo que representa sufre una modificación.
* **snapshots**. Una estructura que define la situación actual de los snapshots. Se definirá con más detalle en las siguientes secciones.

Como se puede observar, el sistema de ficheros **no dispone de un mapa de bloques**. Hemos decidido no utilizar este elemento porque no será necesario en la gestión. De acuerdo a las especificaciones del enunciado, el tamaño máximo de cada fichero es de 4KB, que coincide con el tamaño máximo de bloque. Esto implica que nunca vamos a necesitar más de un bloque para procesar un fichero, por lo que el i-nodo asociado a él será suficiente para realizar todas las funcionalidades a nivel de fichero.

Además, ya que siempre habrá un total de 12 bloques de metadatos, hemos decidido dejar como constantes de programa valores como el primer bloque de datos o el primer bloque de snapshots en lugar de incluirlos como variables en el superbloque.

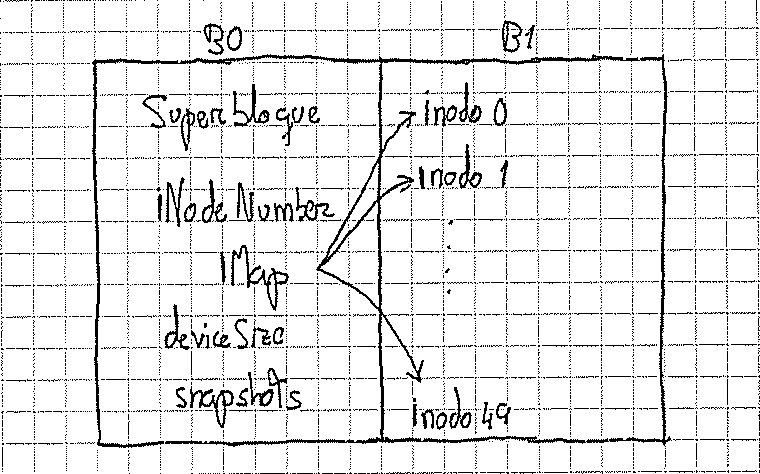


Ilustración 2. Concepto del superbloque y su relación con los i-nodos.

## El bloque de i-nodos

En este bloque se almacena la información de los i-nodos que han sido activados en el sistema de ficheros. Su principal objetivo será controlar el estado de los ficheros y facilitar toda la información que sea necesaria cuando se realiza una operación a nivel de fichero. Este bloque de datos también se escribirá en el disco cada vez que se realiza una operación de desmontaje.

Cada i-nodo almacenará la siguiente información:

* **fileSize**. Tamaño actual del fichero que representa. Tendrá un valor de 0 a 4096 bytes.
* **fileDesc**. Un descriptor de fichero que se ha facilitado al usuario. En esta versión del sistema de fichero el valor corresponde a la posición que ocupa el i-nodo en el bloque.
* **pointer**. Posición del puntero del fichero para las operaciones de lectura y escritura.
* **name**. Array de char que especifica el nombre del fichero. El nombre no deberá exceder de 64 bytes.
* **isOpen**. Un char que determina si el fichero está abierto o no. Toma los valores ‘0’ (cerrado) y ‘1’ (abierto).
* **blockNumber**. Número del bloque en donde se encuentra los datos del fichero. Será un valor mayor o igual a 12.

Los atributos del i-nodo han sido seleccionados cuidadosamente para asegurarnos de que se dispone de toda la información necesaria y, a la vez, permitir que todos los i-nodos quepan en el mismo bloque. Hemos podido comprobar que un i-nodo ocupa un total de 80 bytes, por lo que se utilizan un total de 4000 bytes en este bloque y se quedan un total de 96 bytes como padding.

Entre los atributos y características de cada i-nodo, se han decidido descartar el uso de bloques indirectos porque el sistema de fichero no aceptará el uso de directorios. Por tanto, sólo dispondremos de un bloque directo (definido por *blockNumber*).

Además, también se puede apreciar que la información relevante de acuerdo al descriptor de fichero o si está abierto o no también se almacenará en disco al desmontar el sistema. Si bien este tipo de información no debería ser persistente en disco en diseños más complejos, hemos decidido dejarlo así porque estamos trabajando con un sistema más sencillo e intentar corregirlo implicaría tener que reservar más metadatos en otros bloques del sistema y acabaríamos corriendo el riesgo de quedarnos sin espacio.

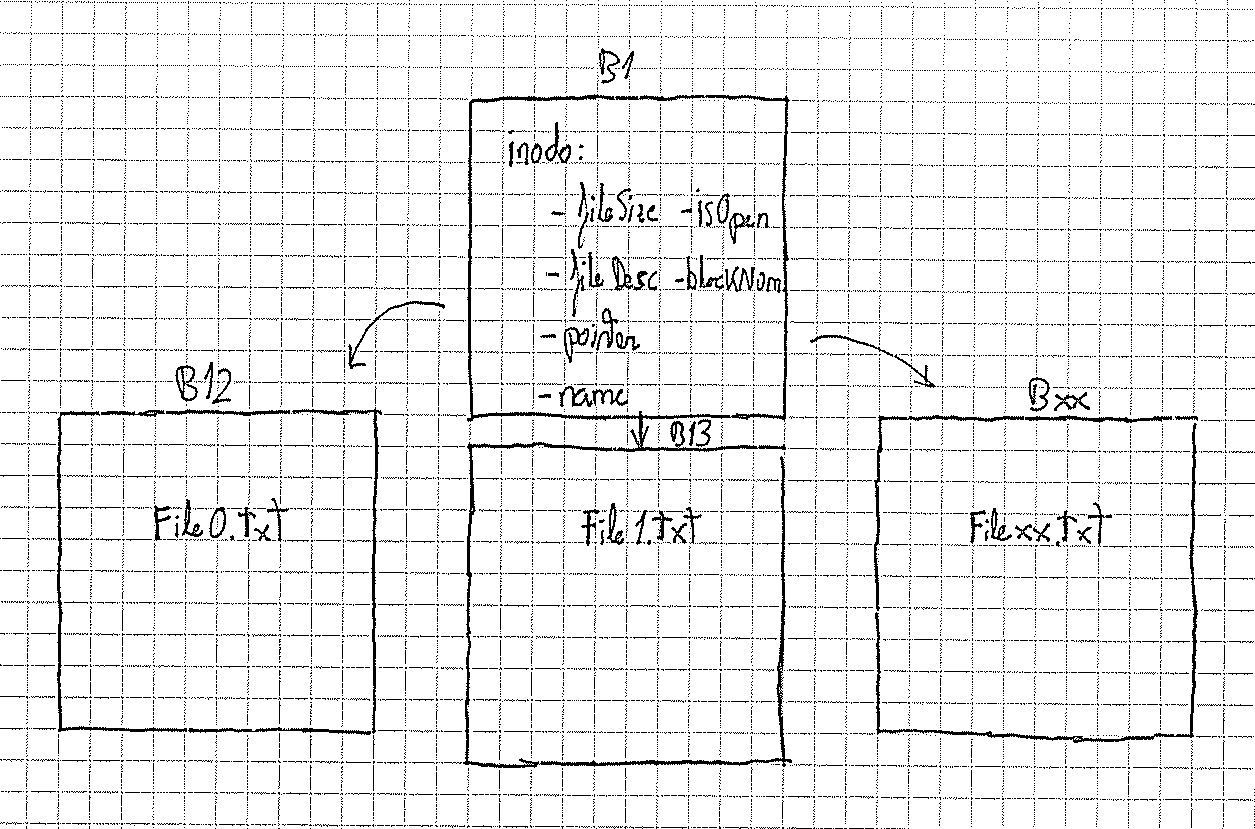


Ilustración 3. Concepto del bloque de i-nodos y su relación con los bloques de datos.

## Los snapshots

Los snapshots permiten deshacer cambios que se hayan realizado previamente en el sistema de ficheros. Los hechos que el usuario podrá deshacer son:

* Creación de un fichero.
* Escritura de un fichero.

Como habíamos definido antes, la información de estado de se encuentra en el superbloque como *snapshots*. La variable es una estructura *SList* definida en un fichero aparte llamado **snapshot.c**. Dicha estructura contendrá las siguientes variables:

* **snapshotList**. Array de 10 posiciones que contiene estructuras *snapshot*, que definiremos más adelante.
* **size**. Número de snapshots disponibles.
* **pointer**. Puntero que apunta a una posición del array **snapshotList** que será modificada si se añade un nuevo snapshot.

A su vez, la estructura *snapshot* dispone de una serie de variables para almacenar el estado previo:

* **operation**. Indica que operación se va a revertir. 0 es “creación de fichero” y 1 es “escritura de fichero”.
* **affected.** Indica qué i-nodo está siendo afectado.
* **block.** Bloque en donde se sitúa la información del fichero antes de una operación de escritura. Su valor estará entre 2 y 11, pero valdrá 0 si la operación es de creación de fichero.
* **fileSize.** Tamaño del fichero antes de la escritura. La variable sólo se utiliza si la operación es de escritura.
* **prevPointer**. Posición del puntero del fichero antes de la escritura. La variable sólo se utiliza si la operación es de escritura.

Los snapshots se implementan como una lista circular: el puntero de SList avanzará por las posiciones del array, y cada vez que se llegue al final del array se volverá a la posición 0 y se sobrescribirán a partir de ahí. Los snapshots se borrarán haciendo el proceso inverso: el puntero avanzará hacia atrás, y cada vez que volvamos a la primera posición se moverá hasta el final del array.

Cada snapshot de operación de escritura apuntará al bloque que le corresponde en orden. Por ejemplo, el snapshot que se sitúe en la posición 0 del array tendrá a su disposición el bloque 2, el 1 tendrá el bloque 3, y así sucesivamente hasta llegar al último, que tendrá el bloque 11.

Con este algoritmo cumpliremos dos objetivos de diseño: no sólo limitamos el número de snapshots a 10, sino garantizamos que podremos revertir los cambios más recientes y en el orden inverso en el que se generaron.

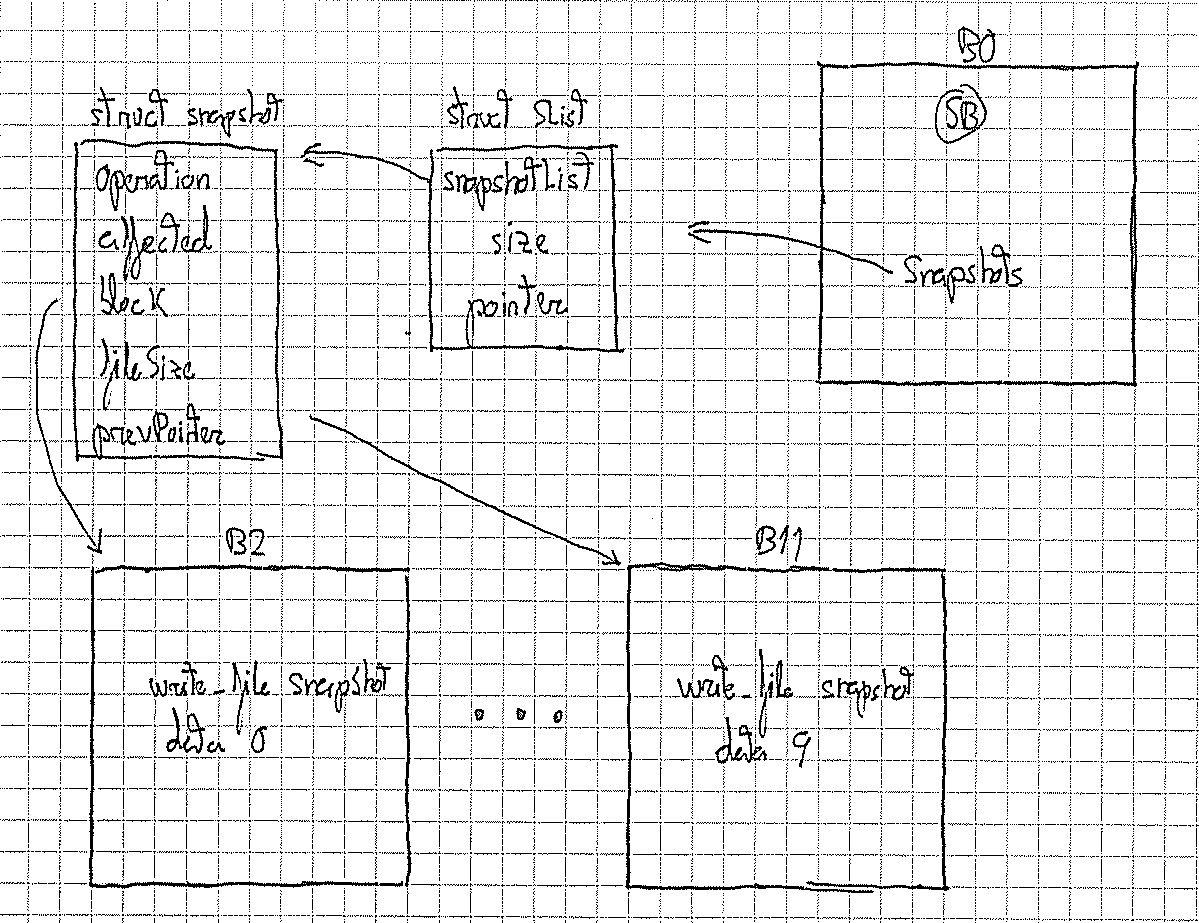


Ilustración 4. Concepto de los snapshots.

# Diseño de las funcionalidades requeridas

A partir de todas las estructuras de datos y variables definidas en la sección anterior, podremos empezar a diseñar las funcionalidades requeridas.

## Formateo, montaje y desmontaje

Cuando el usuario realiza un formateo, se deberán restaurar todos los datos del sistema a su valor inicial. Esto implica que tanto el superbloque como el bloque de i-nodos serán reiniciados, por lo que se perderán todos los datos que hubiese antes de formatear.

Para realizar un formateo, hemos optado por realizar un borrado lógico: los datos del superbloque y de los i-nodos se podrán a sus valores por defecto pero no se eliminará los datos de los bloques de datos en el disco. Sin embargo, ya que los metadatos ya no tienen información de que los datos del fichero existen, ya no se pueden acceder a ellos. De esta manera ganamos en rendimiento mientras que nuestra implementación se adapta a las condiciones impuestas en el enunciado.

La operación de desmontaje permite escribir los metadatos que están en memoria al disco, de manera que cuando se realiza una operación de montaje, se mantenga el mismo estado en el estábamos en la última sesión. En estas operaciones los bloques que se envían/recuperan del disco son los bloques 0 y 1. Como veremos en las operaciones de deshacer cambios, no será necesario cargar los bloques 2 a 11 porque ya están escritos en el disco y podremos acceder a ellos con el puntero de los snapshots.

Tal y como está implementado nuestro sistema de ficheros, es altamente recomendable realizar el desmontaje antes de salir del programa, sobre todo si se ha escrito en ficheros, para evitar pérdidas de datos e inconsistencias en el sistema.

## Creación, apertura y cerrado de ficheros

Para crear un fichero, será necesario asegurarse de que hay espacio disponible (es decir, si hay i-nodos disponibles). También se comprobará si no hay un fichero con el mismo nombre (no podrá haber dos con el mismo nombre). Si se cumplen estas condiciones, se procederá a crear un nuevo i-nodo para el fichero, poniendo sus valores por defecto y asignando un bloque de memoria para escribir en él. En esta situación se procederá a realizar un borrado físico del bloque afectado para asegurarnos de que no queda información de estados anteriores. El superbloque y el bloque de i-nodos se actualizarán en memoria (pero no en disco: hay que desmontar para que los cambios sean persistentes). La única manera de borrar un fichero será deshaciendo el cambio o bien formateando el disco.

Las operaciones de apertura se efectuarán si el fichero que se quiere abrir ha sido creado en el sistema. En este caso, el i-nodo estará en modo de “fichero abierto” y se le asignará un descriptor de fichero, que también se le proporcionará al usuario. No se permitirá abrir más de una vez un fichero, con el objetivo de asegurarse de que hay espacio suficiente para procesar todos los demás ficheros.

Las operaciones de cerrado pondrán el i-nodo correspondiente en estado de “cerrado” y se establecerá su descriptor de fichero a 0.

## Lectura, escritura y reposicionamiento de puntero

Para cualquiera de estas operaciones, será necesario que el fichero a procesar tenga un descriptor de fichero y, por tanto, esté abierto.

En una operación de lectura, se procederá a utilizar la función **bread**() (ofrecida por ufs.h) para obtener el bloque determinado por el i-nodo del fichero. Después, se procederá a almacenar en el buffer de destino el número de bytes solicitado, empezando desde la posición en donde se situaba el puntero. No se realizará una lectura si el puntero está al final del fichero, y se leerán menos bytes si el número solicitado supera el número de bytes que quedan por leer.

En una operación de escritura, se utilizará bread() para obtener el bloque afectado y se realizará la operación escribiendo los datos a partir de la posición del fichero. Si el puntero no estaba al final del fichero se podrán sobrescribir bytes. Una vez realizada la operación se calculará el tamaño del fichero para actualizarlo en el i-nodo. También se creará un snapshot de la situación previa. No se realizará una escritura si el puntero se sitúa en la posición 4096, y si la escritura supera el tamaño máximo se escribirá todo lo que se pueda y se perderá el resto.

Se debe tener en cuenta de que el puntero del i-nodo se utiliza tanto en las operaciones de lectura como de escritura. Se permitirá colocar el puntero en la posición deseada mediante una función definida en filesystem.h. El usuario podrá cambiar la posición del puntero siempre y cuando no quede fuera del fichero.

## Deshacer cambios

Las condiciones fundamentales para realizar una operación de deshacer cambios serán que todos los ficheros estén cerrados y que haya algún snapshot disponible. Se podrán realizar dos tipos de cambio:

* Si es una operación de creación de ficheros, se procederá a borrar la referencia del i-nodo en el mapa de i-nodos del superbloque.
* Si es una operación de escritura, se recuperará el bloque del snapshot correspondiente para luego ser escrito en el bloque del i-nodo afectado, y se calculará su nuevo tamaño de fichero y puntero.

Como definimos antes, en los snapshots de creación de ficheros se utilizarán sólo el atributo *affected*, y se utilizarán el resto en una operación de escritura.

# Implementación

A partir del diseño realizado, hemos implementado las funciones requeridas de filesystem.c con nuestras propias estructuras de datos. También se han utilizado funciones auxiliares y ficheros aparte, que también serán definidos.

En primer lugar vamos a hablar de los métodos auxiliares que hemos implementado y las variables globales:

* **Variables**
  + **inodes:** es una variable de tipo inodeList y es una estructura donde almacenaremos el bloque de i-nodos que se encuentra en disco. Los datos almacenados en esta variable siempre serán persistentes con el disco (mientras se utilice correctamente la función umountFS()).
  + **SB**: es una variable de tipo SuperBlock y es una estructura donde almacenaremos el superbloque que se encuentra en disco. Los datos almacenados en esta variable siempre serán persistentes con el disco (mientras se utilice correctamente la función umountFS()).
* **Funciones**
  + **getPosition**: dado un descriptor de fichero pasado por parámetro busca la posición del i-nodo dentro de la estructura inodes. Recorremos esta estructura y si encontramos un nodo que tenga el mismo descriptor de fichero salimos del bucle y almacenamos en qué posición lo hemos encontrado. Devolvemos un -1 si no hemos encontrado ningún fichero con el mismo descriptor de fichero y la posición del fichero en el caso de que si lo hayamos encontrado.
  + **print:** accedemos a SB e imprimimos el número de i-nodos, el tamaño del disco, y todo el mapa de i-nodos.

## mfFS

Esta función es la encargada de formatear nuestro sistema. Recibe dos parámetros de entrada: un número máximos de ficheros y el tamaño en bytes de nuestro sistema.

Lo primero que debemos hacer es comprobar si estos parámetros cumplen con las especificaciones que se nos pedían en el enunciado. El número deberá ser menor o igual que 50 y mayor o igual que 1 y el tamaño de nuestro sistema deberá ser mayor o igual 327680 Bytes o menor o igual a 512000 Bytes. Si no se cumple algunas de estas restricciones se generará un mensaje de error y la función terminará y devolverá -1.

Si no se ha producido ningún error, debemos modificar los metadatos. En el superbloque pondremos el nuevo tamaño de nuestro sistema y el número de ficheros que se nos facilitó como argumentos. Además debemos poner el todo el iMap con ceros porque después del formateo no habrá ningún i-nodo ocupado. El tamaño del snapshot y su puntero deberá ser igual a 0 porque después del formateo no tendremos al opción de deshacer la acción de un fichero porque no habrá ninguno creado. Reiniciamos todos los i-nodos para borrar los datos de los ficheros que teníamos creados, invocamos la función umountFS que se encargará del desmontaje del sistema e imprimiremos un mensaje informando de que todos se ha realizado correctamente.

## umountFS

Esta función se encargará del desmontaje del sistema. Las modificaciones que habíamos hecho antes en mfFS solo estaban reflejadas en memoria. Para hacerlas persistentes con el disco esta función utilizara bwrite para escribir el nuevo superbloque en la el bloque 0 del disco y el bloque de i-nodos en el bloque 1 del disco. No escribimos el bloque de snapshots porque ya se escriben en la función writeFS. Se imprime un mensaje para informar de que la operación se ha realizado con éxito y la función devuelve un 0 debido a que es un aviso y no un error.

## mountFS

Esta función se encargará del montaje del sistema. Leemos los superdatos que hay en el disco y los pasamos a las estructuras que tenemos creadas en memoria. Con la función bread leemos el bloque 0 del disco y almacenamos lo que hemos leído en SB, que es una variable global de tipo SuperBlock. De la misma forma, leemos con bread el bloque 1 y almacenamos lo que hemos leído en inodes, que es una variable global de tipo inodeList.

Por último imprimimos un mensaje informando de que la operación se ha realizado con éxito y la función devuelve un 0.

## undoFS

Esta función se encargará de revertir cambios en ficheros. Antes de realizar ningún cambio en los ficheros deberemos hacer una serie de comprobaciones. En primer lugar accedemos al tamaño de snapshoot en el superbloque que aumenta en 1 cada vez que se realiza alguna modificación o creación de un fichero. Si este tamaño es 0 quiere decir que no hay cambios recientes, por lo que no se puede hacer undo. También debemos recorrer el mapa de indos para comprobar si hay algún fichero abierto. En caso de haber un fichero abierto tampoco se podrá hacer undo. Si se da alguna de estas situaciones se imprimirá un mensaje de error y la función terminará devolviendo un -1.

Si no ha habido ningún error, reservamos memoria y recuperamos el último snapshot almacenado en la lista de snapshot. Para ello llamamos a la función delete que eliminará el snapshot de la lista y lo guardará en nuestra variable. Si esta función devuelve un -1 significa que debemos terminar la función y devolver -1 (aunque este comportamiento no debería suceder nunca). Solo debemos modificar los ficheros actuales si la operación que se hizo fue una creación o una escritura.

Si la operación fue de creación, accedemos a la posición del iMap donde nos creamos el fichero y lo ponemos a 0 para que el i-nodo vuelva a estar libre. Imprimimos un mensaje informando que la operación se ha realizado con éxito y devolvemos cuántos undos podemos hacer más.

Si la operación fue de escritura leemos el bloque del disco que nos indica el atributo bloque del snapshot que hemos obtenido anteriormente de la lista de snapshots. Este bloque tendrá los datos del bloque afectado antes de la escritura. Después de almacenar lo que hemos leído en este bloque escribimos con bwrite en el bloque aceptado y así deshacemos la escritura. Además deberemos restablecer el puntero y el tamaño de fichero. Para ello cogeremos los atributos prevPointer y fileSize almacenados en la estructura snapshot que hemos obtenido de la lista de snapshots. Imprimimos un mensaje informando que la operación se ha realizado con éxito y devolvemos cuantos undos podemos hacer más.

Si la operación que tenia almacenada en el snapshot que hemos obtenido de la lista de snapshots no es ni de creación ni de escritura lanzaremos un mensaje de error y devolveremos un -1, si bien este comportamiento no debería producirse en ningún caso.

## creatFS

Esta función será la encargada de crear los ficheros. Como es habitual, debemos hacer una serie de comprobaciones antes de realizar la creación del fichero. En primer lugar debemos comprobar si hay espacio para la creación de un nuevo fichero. Para ello nos recorremos todo el iMap y si encontramos algún 0 podremos introducir el i-nodo en esta posición. En el caso contrario imprimiremos un mensaje informando del error y devolveremos un -1.

Además debemos comprobar si ya hay un fichero creado con el mismo nombre. Para ello nos recorremos todos los i-nodos comprobados si hay alguno con el mismo nombre y si iMap dice que está ocupado. Esto lo hacemos porque puede que hayamos creado un fichero pero al hacer un undo lo hayamos eliminado. Cuando esto pasa, undo no borra todo los datos del i-nodo solo cambia el iMap por ello la única forma de contemplar este caso es ver el valor del iMap. Aunque encontremos un i-nodo con el mismo nombre de fichero pero el iMap nos diga que está desocupado lo deberemos crear igual porque ha ocurrido el caso que hemos mencionado anteriormente. Si se ha encontrado un nodo ocupado y con el mismo nombre debemos imprimir un mensaje avisando de lo ocurrido y la función devolverá un 1.

Actualizamos todos los datos necesarios en el i-nodo correspondiente y escribimos en disco con la función bwrite. Además debemos modificar el iMap para reflejar que este i-nodo estará ocupado ahora y debemos añadir un snapshot a la lista de snapshot que refleje la acción que hemos realizado. Imprimimos un mensaje informando de que la operación se ha realizado con éxito y devolvemos un 0.

## openFS

Esta función se encargará de abrir un fichero. Con el nombre del fichero que recibimos por parámetro recorremos todos los i-nodos para ver si existe este fichero y si está ocupado. Esto lo hacemos por el mismo motivo que en la función creatFS. Si no lo encontramos devolvemos un -1 y un mensaje de error. Hacemos el mismo procedimiento si el fichero está ya abierto aunque devolveremos un 0 en lugar de un -1 debido a que es un aviso y no un error.

Si no se ha producido ninguno de los casos mencionados anteriormente procedemos a modificar el i-nodo. El puntero será 0, la variable isOpen reflejará que el fichero está abierto y le asignaremos un descriptor que será la posición del i-nodo más 1 para que nunca sea 0 el descriptor de fichero. Imprimimos un mensaje de que la operación se ha realizado con éxito y devolvemos el descriptor de fichero.

## closeFS

Esta función se encargará de cerrar un fichero. Con la función getPosition que hemos descrito al comienzo de este apartado obtendremos la posición del i-nodo dado el descriptor del fichero que se nos ha pasado por parámetros. Si esta función devuelve -1 significa que no hay ningún fichero con este descriptor de fichero, lanzaremos un mensaje de error y devolveremos -1.

Si la función no nos devuelve -1 habremos obtenido la posición del i-nodo. Modificaremos los atributos isOpen y fileDescp que tomará el valor 0 debido a que nunca generamos descriptores menores que 0 y será un indicador de que el fichero no está abierto.

## readFS

Esta función se encarga de leer un fichero. Como siempre tendremos que hacer unas comprobaciones iniciales antes de leer el fichero.

Con la función getPosition que hemos descrito al comienzo de este apartado obtendremos la posición del i-nodo dado el descriptor del fichero que se nos ha pasado por parámetro. Si esta función devuelve -1 significa que no hay ningún fichero con este descriptor de fichero, lanzaremos un mensaje de error y devolveremos -1. Si el puntero del fichero está al final del fichero se devuelve un mensaje con un aviso y se devuelve un 0. Por último si se intenta leer 0 o menos bytes un mensaje de error y devolveremos -1.

Si no se ha producido ninguno de estos casos calculamos los bytes que vamos a leer y si el número de bytes sobrepasa el límite del fichero leeremos hasta el final de este. Después leemos el fichero con bread y pasamos los datos al buffer que nos ha pasado por parámetro con la función memmove. Actualizamos el puntero del fichero y devolvemos el número de bytes reales que hemos leído.

## writeFS

Esta función se encargará de escribir un fichero. Con la función getPosition que hemos descrito al comienzo de este apartado obtendremos la posición del i-nodo dado el descriptor del fichero que se nos ha pasado por parámetro. Si esta función devuelve -1 significa que no hay ningún fichero con este descriptor de fichero, lanzaremos un mensaje de error y devolveremos -1. Si el puntero del fichero está al final del fichero se devuelve un mensaje con un aviso y se devuelve un 0. Por último si se intenta leer 0 o menos bytes un mensaje de error y devolveremos -1.

Como es una escritura deberemos generar un snapshot que recoja los datos que vamos a modificar. Leemos el bloque actual y lo escribimos en el bloque libre que nos indica el puntero de snapshot que se encuentra en el superBloque. Además debemos de tipo snapshot que recoja lo que hemos hecho y añadirla a la lista de snapshot. De esta forma tendremos almacenado el fichero sin modificar por si debemos hacer un undo.

Después, tendremos que calcular los bytes que vamos a escribir y si el número de bytes sobrepasa el límite del fichero escribiremos hasta el final de este. Cuando hayamos calculado el número de bytes que tenemos que leer debemos leer el bloque que vamos a escribir, modificar el bloque para escribir lo que hay en el buffer (con la función memcopy) y volver a escribir el buffer en el disco. Actualizamos el puntero del fichero que hemos escrito, actualizamos el tamaño del fichero, imprimimos un mensaje de que la operación se ha realizado con éxito y devolveremos el número de bytes escritos.

## lseekFS

Esta función será la encargada del reposicionamiento del puntero de un fichero. Con la función getPosition que hemos descrito al comienzo de este apartado obtendremos la posición del i-nodo dado el descriptor del fichero que se nos ha pasado por parámetro. Si esta función devuelve -1 significa que no hay ningún fichero con este descriptor de fichero, por lo que lanzaremos un mensaje de error y devolveremos -1.

Si no se da este caso se lee que nos ha pasado la función como parámetro. Si el parámetro whence es FS\_SEEK\_END, se sitúa el puntero al final del fichero ignorando el segundo parámetro.

Si whence es FS\_SEEK\_SET tendremos que leer el segundo parámetro (offset). Pero si offeset es negativo o mayor que el tamaño del fichero mandaremos un mensaje de error y devolveremos un -1 debido a que no se puede situar el puntero del fichero fuera del tamaño del fichero. Si no se da ninguno de estos casos se coloca el puntero del fichero donde el parámetro offset nos especifica.

Por último si whence no es ninguno de los valores mencionados anteriormente fichero mandaremos un mensaje de error y devolveremos un -1.

## snapshot.c

Por último vamos a definir como hemos implementado la lista de snapshots:

SList es una lista cíclica de tamaño 10. Esta lista tendrá dos atributos: size que indica cuántos snapshots guarda actualmente la lista y otro atributo llamado pointer que indica el siguiente vacio donde se puede escribir (o sobrescribir). La idea es la siguiente, cuando hayamos almacenado 10 snapshot en la lista nuestra lista permitirá sobrescribir el snapshot más antiguo para remplazarlo por el nuevo. Slist contará con dos métodos:

* **add**: función que permite añadir snapshots a la lista. Si el puntero no es 9 (el máximo de snapshot que se puede almacenar ya que el puntero toma valores de 0 a 9) se añadirá el snapshot en la posición que indica el puntero, se incrementará el puntero y si el tamaño no supera o es igual el máximo se aumentará. SI el puntero es 9 debemos escribir el snapshot al final de la lista y el puntero ahora será 0 para que el siguiente elemento se añada en la posición del elemento que lleva más tiempo en la lista. Si el tamaño no supera o es igual el máximo se aumentará.
* **delete**: borrará el elemento añadido más recientemente en la lista y se devolverá este snapshot en el buffer pasado por parámetro. Si el tamaño de la lista es 0 se devolverá un -1. Si el puntero es 0 se cambiará su valor a la última posición de la lista. Si no se da este caso solo se reducirá el puntero en 1. Se tomará el snapshot de este puntero y se pasará al buffer que se nos ha pasado por parámetro. Por último se reduce el tamaño de la lista en 1.

# Batería de pruebas

A continuación se adjuntan las pruebas que hemos realizado para comprobar el funcionamiento del sistema de ficheros. En cada prueba se define:

* El estado inicial del sistema de ficheros.
* El objetivo de la prueba.
* Un resumen de la traza con los aspectos más importantes o en su defecto una imagen.

## Prueba 1. Creación, apertura y cierre de ficheros

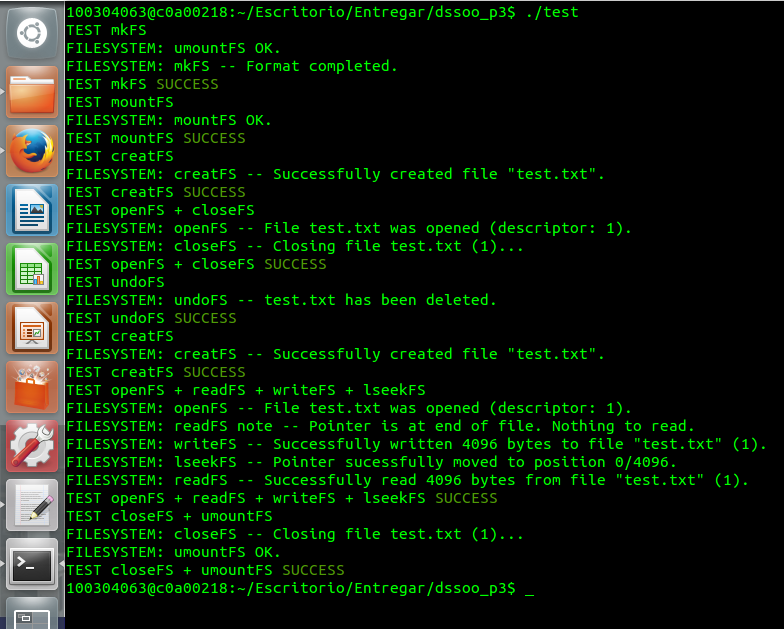
* **Estado inicial del sistema de ficheros**: El sistema de ficheros inicialmente no se encuentra montado.
* **Objetivo**: El objetivo de la prueba es la creación de ficheros y la comprobación de que las principales funcionalidades del sistema de ficheros tales como openFS, closeFS, readFS, writeFS, lseekFS, undoFS, mountFS y unmountFS, funcionan correctamente. Lo primero que hacemos es montar el sistema de ficheros, hacemos las pruebas para comprobar las funcionalidades y finalmente desmontamos el sistema de ficheros. Esta prueba es la que venía por defecto en el proyecto inicial.

Ilustración 5. Traza de la prueba original.

## Prueba 2. Creación máxima de ficheros y undoFS.

* **Estado inicial del sistema de ficheros**: El sistema de ficheros inicialmente no se encuentra montado.
* **Objetivo**: El objetivo de la prueba es crear más ficheros de los que podemos crear y hacer más undo de los que se pueden hacer. El sistema de ficheros permite la creación de 12 ficheros, nosotros creamos 13 para ver si después de crear 12, cuando vayamos a crear el 13, nos mande un mensaje de error y no lo cree. Lo mismo pasa con la función undo, solo podemos usarla 10 veces, la usamos 11 para ver si muestra mensaje de error.

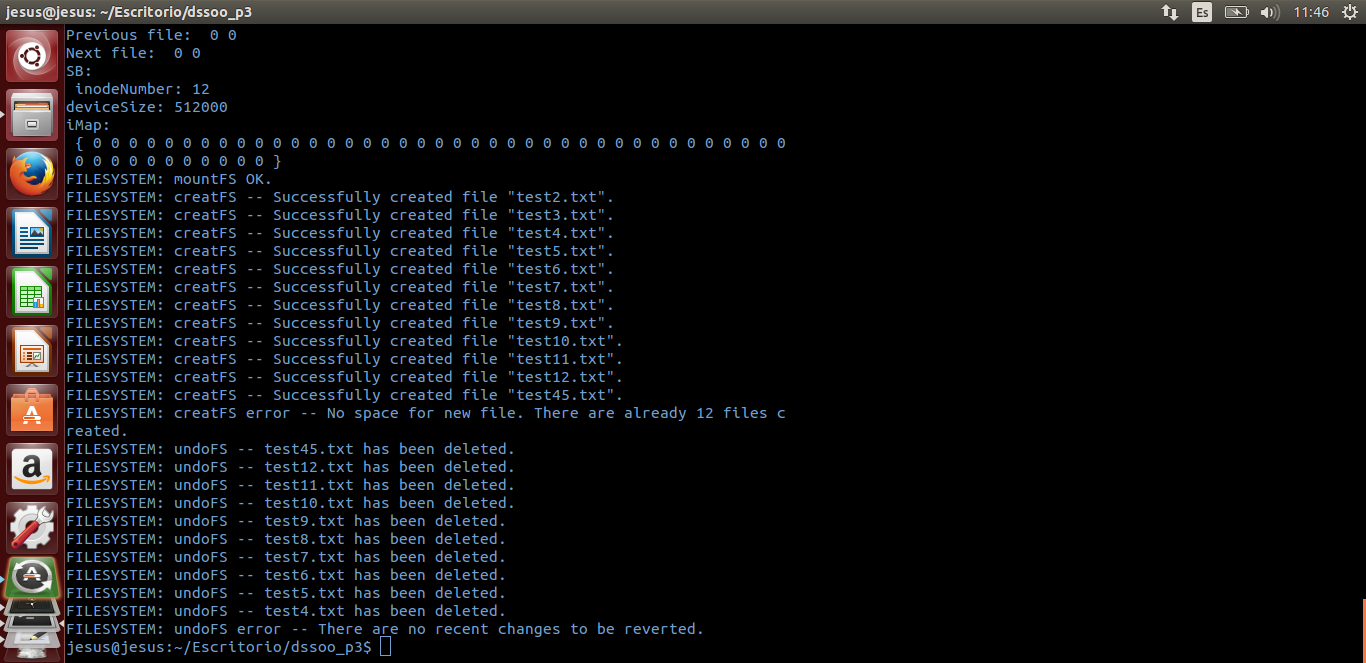


Ilustración 6. Creación máxima de ficheros y undo.

La prueba ha obtenido los siguientes resultados:

* Se han creado 12 ficheros correctamente. El decimotercero no se puede crear debido a que se ha llegado al límite de espacio.
* Al utilizar undoFS() diez veces, se han eliminado los ficheros en orden inverso a su creación. Utilizar la función undoFS() por undécima vez emite un mensaje de error.

## Prueba 3. Revertir escrituras.

* **Estado inicial del sistema de ficheros**: El sistema de ficheros inicialmente no se encuentra montado.
* **Objetivo**: El objetivo de esta prueba es usar la función undo en situaciones de escritura y ver que funciona correctamente. Para ello, montamos el sistema de ficheros y creamos un fichero y hemos hecho operaciones de escritura y lectura. Hemos cerrado el fichero y usado la función undo, comprobamos que funciona correctamente.

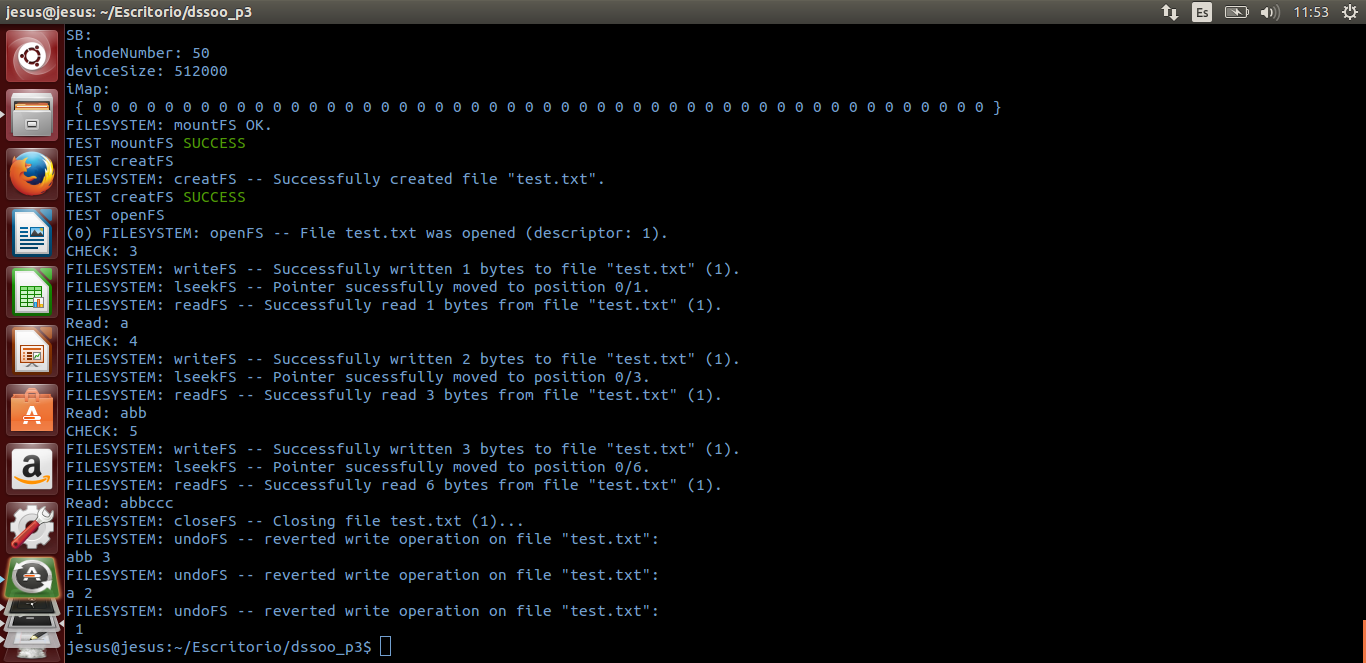


Ilustración 7. Probar la función undoFS

La prueba ha obtenido los siguientes resultados:

* Las escrituras actualizan correctamente el bloque de datos.
* Al realizar la función undoFS() se vuelve al estado previo del bloque.

Notas:

* Tras realizar undoFS() se imprime por pantalla el texto del bloque y un número. Dicho número indica cuantos snapshots se pueden utilizar.
* El mensaje “CHECK: X” (X siendo un número) indica el bloque de snapshot en donde se está escribiendo.

## Prueba 4. Probar función writeFS.

* **Estado inicial del sistema de ficheros**: El sistema de ficheros inicialmente no se encuentra montado.
* **Objetivo**: El objetivo de esta prueba es usar las funciones writeFS. Montamos el sistema de ficheros, creamos un fichero y hacemos operaciones de lectura y escritura. Comprobamos que hace correctamente las escrituras.

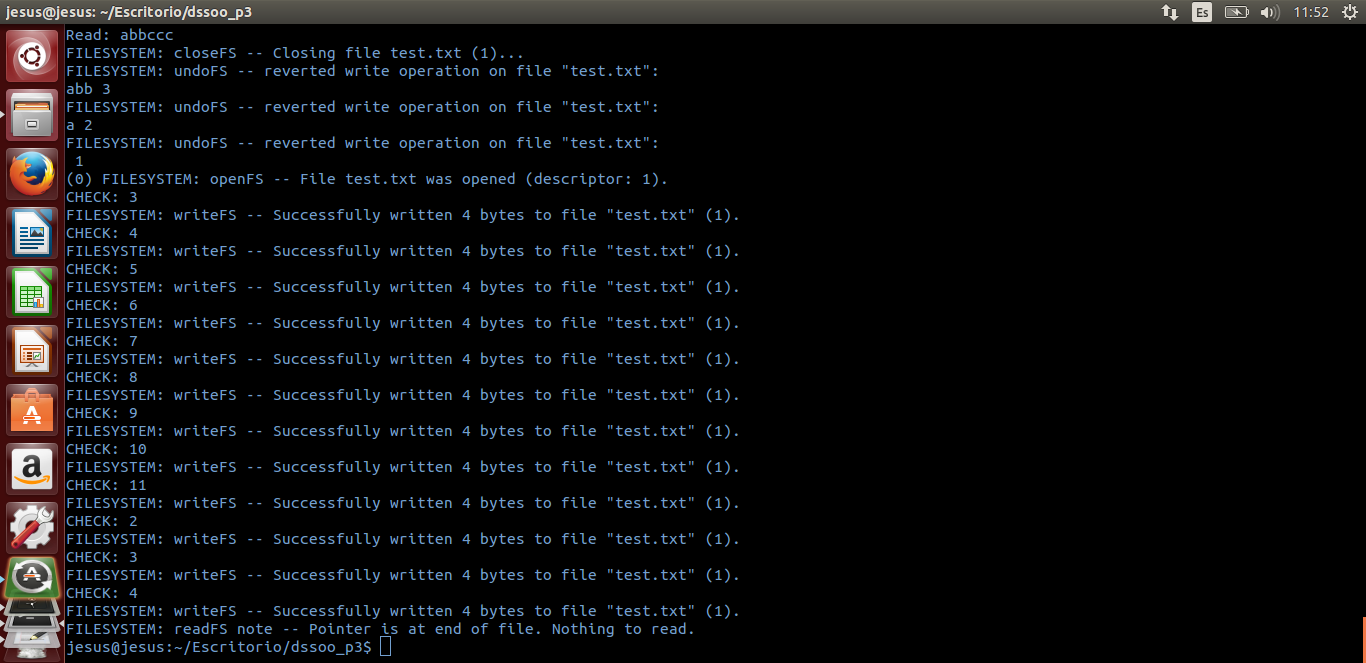


Ilustración 8. Probar función writeFS

La prueba ha obtenido los siguientes resultados:

* Los datos a almacenar en los snapshots van a los bloques correctos.
* El puntero del fichero se actualiza correctamente con cada escritura.

## Prueba 5. Probar función writeFS y readFS.

* **Estado inicial del sistema de ficheros**: El sistema de ficheros inicialmente no se encuentra montado.
* **Objetivo**: El objetivo de esta prueba es usar las funciones writeFS y readFS. Montamos el sistema de ficheros, creamos un fichero y hacemos operaciones de lectura y escritura. Después de una escritura, colocamos el puntero en mitad del fichero y volvemos a hacer una escritura, comprobamos que ejecuta dicha escritura y machaca los datos que había en las posiciones ocupadas.

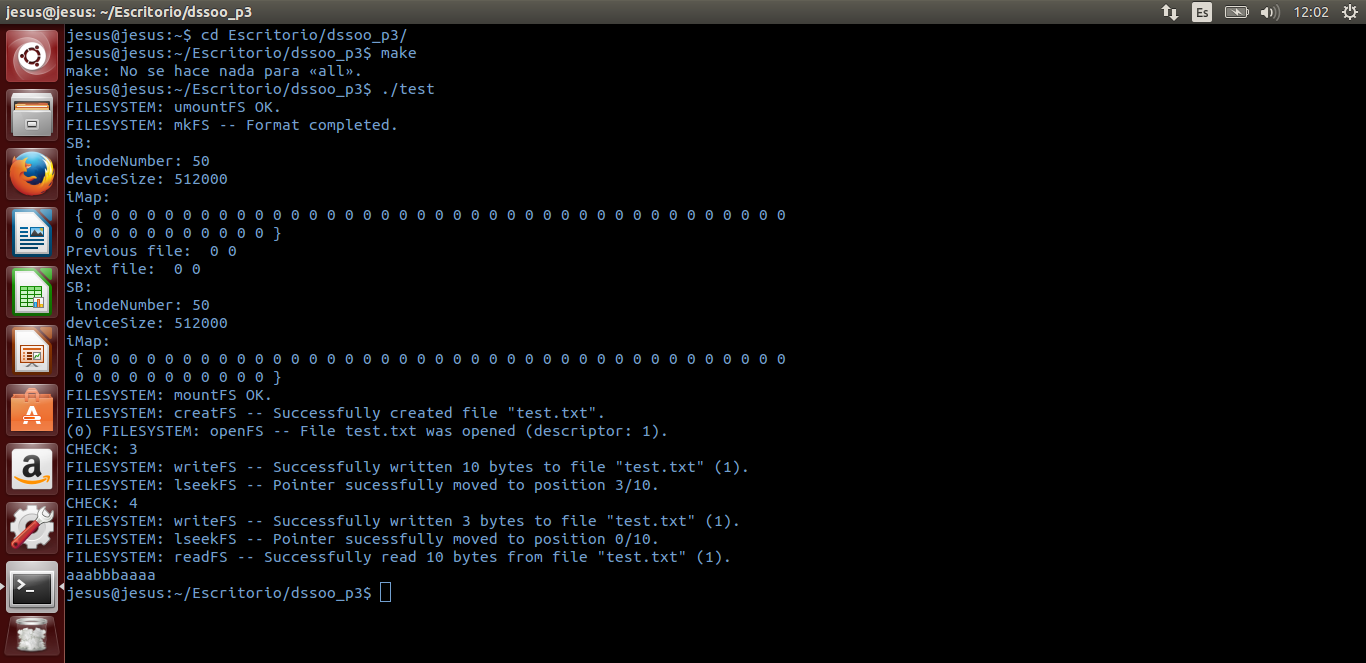


Ilustración 9. Probar la función writeFS y readFS

La prueba ha obtenido los siguientes resultados:

* Si se realiza una escritura y el puntero no está al final del fichero, se sobrescriben los bytes correspondientes y el tamaño de fichero sigue consistente.

## Prueba 6. Persistencia de datos en el disco

* **Estado inicial del sistema de ficheros**: El sistema de ficheros inicialmente no se encuentra montado.
* **Estado tras primera ejecución**: El sistema de ficheros ha creado dos ficheros y se han escrito en ellos.
* **Objetivo**: verificar la consistencia de datos en el sistema. Para ello se realizarán dos ejecuciones: en la primera se escribirán en dos ficheros y en la segunda se leerán los datos.

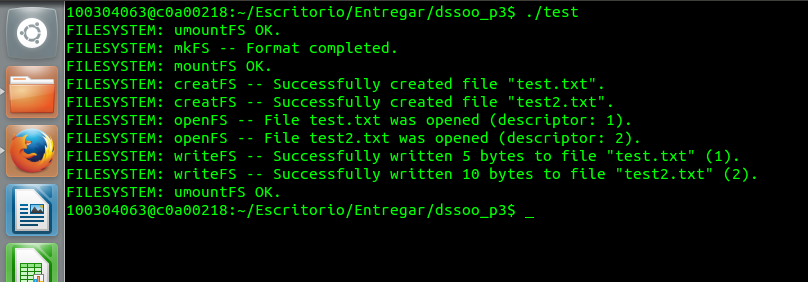


Ilustración 10. Traza tras la primera ejecución.

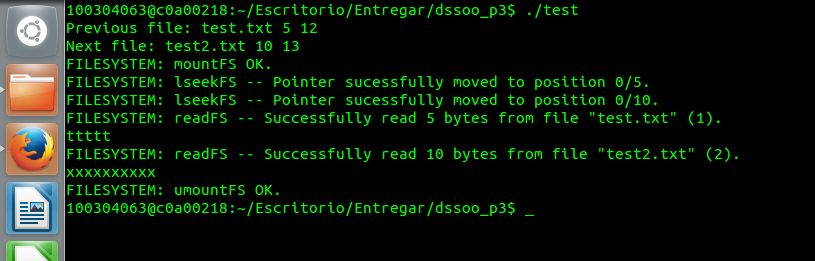


Ilustración 11. Traza tras la segunda ejecución.

Notas:

* En mountFS() se ha incluido trazas adicionales para comprobar que los ficheros aún existen. El formato es: <nombreFichero>, <tamañoFichero>, <bloqueDisco>.

# Conclusiones

Esta práctica nos ha servido de gran utilidad para comprender el funcionamiento de un sistema de ficheros y cómo trabajar a nivel de bloque y ficheros. Esta práctica ha tenido además la particularidad de que no disponíamos de ninguna base para implementar el sistema, por lo que hemos ver todo el proceso de diseño, que es el aspecto fundamental de la asignatura de Diseño de Sistemas Operativos.

Al principio nos sentíamos “abrumados” por la gran dificultad del proyecto, además de las limitaciones de tiempo que hemos tenido debido a otras asignaturas. Sin embargo, una vez que empezamos a plantear cómo debería funcionar el sistema de ficheros y qué estructuras de datos emplearía, empezamos a comprender cuál era el objetivo de cada función y pudimos asociar cada estructura implementada a cada una de las funcionalidades.

Por tanto, tenemos una valoración muy positiva de esta práctica. Si bien es muy difícil de entender y poner en marcha al principio, al final hemos aprendido mucho a diseñar nuevas funcionalidades para un sistema operativo.