2020

lobo7 lobato diaz

Alberto Lobato Díaz, 100383508 Alejandro Massotti López, 100383565 Javier Fernández Velasco, 100383392,

Práctica 2 Sistemas de ficheros

CONTENIDO

[INTRODUCCIÓN 1](#_gjdgxs)

[DISEÑO DEL SISTEMA DE FICHEROS 1](#_30j0zll)

[DESCRIPCIÓN DE FUNCIONALIDADES PRINCIPALES 2](#_1fob9te)

[mkFS() 2](#_3znysh7)

[mountFS() 2](#_2et92p0)

[unmountFS() 2](#_tyjcwt)

[ifree() 3](#_3dy6vkm)

[ialloc() 3](#_1t3h5sf)

[alloc() 3](#_4d34og8)

[fileExist() 3](#_2s8eyo1)

[isNodeFull() 3](#_17dp8vu)

[isBlocksFull() 3](#_3rdcrjn)

[createFile() 3](#_26in1rg)

[busca\_inodo() 4](#_lnxbz9)

[removeFile() 4](#_35nkun2)

[openFile() 4](#_1ksv4uv)

[closeFile() 4](#_44sinio)

[readFile() 4](#_2jxsxqh)

[writeFile() 4](#_z337ya)

[lseekFile() 5](#_3j2qqm3)

[checkFile() 5](#_1y810tw)

[includeIntegrity() 5](#_4i7ojhp)

[openFileIntegrity() 5](#_2xcytpi)

[closeFileIntegrity() 6](#_1ci93xb)

[createLn() 6](#_3whwml4)

[removeLn() 6](#_2bn6wsx)

[fullInodeMap() 6](#_qsh70q)

[fullBlockMap() 6](#_3as4poj)

[printSystem() 6](#_1pxezwc)

[DISEÑO DE PRUEBAS 6](#_49x2ik5)

[F1: Funcionalidades principales del sistema de ficheros 7](#_2p2csry)

[NF1: Comprobación de requisitos no funcionales 13](#_147n2zr)

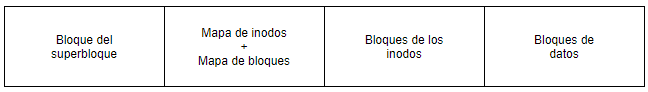
[CONCLUSIONES 14](#_3o7alnk)

# INTRODUCCIÓN

Esta segunda práctica tiene como objetivo implementar un sistema de ficheros en espacio de usuario usando el lenguaje de programación C. A través de esta práctica se buscan asentar los conocimientos obtenidos a lo largo de la asignatura. En este documento se detallan los siguientes apartados:

* Diseño del sistema de ficheros, incluyendo las estructuras de i-nodo y superbloque.
* Descripción de las funcionalidades, indicando el funcionamiento y desarrollo de los métodos principales y auxiliares.
* Diseño y realización de pruebas, detallando el resultado de cada una de las pruebas desarrolladas en el fichero “test.c”.

# DISEÑO DEL SISTEMA DE FICHEROS



El diseño del sistema de ficheros cuenta con un superbloque, que ocupa un bloque de almacenamiento, un bloque que contiene el mapa de inodos y el mapa de bloques de datos libres, almacenados juntos para la optimización de almacenamiento, 3 bloques que contienen la información de los 48 inodos y el resto de los bloques serán de tipo bloques de datos.

A continuación, se procede a la explicación de las estructuras nombradas anteriormente:

El superbloque estará formado por:

* Un número mágico como identificador del sistema de ficheros.
* El número de bloques que ocupará los mapas de inodos y de bloques, los cuales los hemos unificado en un bloque, para optimizar el espacio.
* El número de inodos del sistema de ficheros, siendo 48.
* El número del bloque donde se encontrará el primer i-nodo, el cual será el 2, ya que el bloque 0 estará reservado para el superbloque y el 1 para los mapas de bits.
* El número de bloques de datos, calculado en el mkFS mediante la siguiente fórmula DEVICE\_SIZE / BLOCK\_SIZE - 2 - (lo que ocupa los bloques de inodos = 3).
* El primer bloque de datos, posicionado en el bloque 5 (ya que se empieza en el bloque 0), ya que los dos primeros bloques son para el superbloque y los mapas de bits y los 3 siguientes para los inodos.
* El tamaño del dispositivo, el cual será el DEVICE\_SIZE.
* El padding añadido para completar un bloque, por lo que será un padding de 2020 bytes, formado por un vector de caracteres de tamaño 2020.

La estructura que contiene la información de los mapas está formada por:

* El mapa de inodos libres, para su representación se ha utilizado un vector de caracteres con 6 posiciones. Esto es debido a que con 6 posiciones de array de caracteres se pueden representar 48 bits, que es el número de inodos con el que cuenta el sistema.
* El mapa de bloques de datos libres, para su representación se ha utilizado un vector de 38 caracteres obtenido mediante la fórmula [300 (máximo teórico de tamaño del sistema) / 8 (bits en 1 byte) = 38]
* El relleno añadido para completar un bloque es de 2004 bytes, por lo que se usa un vector de caracteres con 2004 posiciones.

La estructura que contiene la información de los inodos está compuesta por:

* El tipo de archivo, diferenciando los ficheros, con un 1, de los enlaces blandos, con un 0. No será necesario diferenciar los directorios ya que no será necesaria su implementación.
* La posición en bytes desde el inicio del archivo, inicialmente esta posición se encontrará al inicio del fichero (posición 0).
* Un marcador para comprobar si el fichero está abierto (open = 1) o cerrado (open = 0).
* La integridad correspondiente a un i-nodo, inicialmente será 0. Además, dicha integridad será calculada al cerrar su i-nodo correspondiente.
* El nombre del fichero o enlace blando asociado al i-nodo correspondiente. La longitud máxima será de 32 caracteres, aunque el vector de caracteres utilizados tendrá un tamaño de 33 caracteres para guardar el final de cadena \0.
* Los inodos contenidos. Al tener un tamaño máximo de 10KiB, podemos observar que al dividirlo entre el tamaño de un bloque (2048 bytes), podremos crear un vector de números enteros de longitud 5. En cada posición se pondrá el bloque que está usando el archivo. Si es un enlace simbólico, inodosContenidos[0] corresponderá al i-nodo al que apunta.
* El tamaño que ocupará el i-nodo, siendo un máximo de 10240 bytes.
* La cantidad de bloques que ocupan los datos del archivo del i-nodo.
* El relleno añadido para completar un bloque es de 47 bytes, por lo que se usa un vector de caracteres con 47 posiciones.

# DESCRIPCIÓN DE FUNCIONALIDADES PRINCIPALES

En este apartado se detalla el funcionamiento de los métodos principales para la correcta implementación del sistema de ficheros. Las funciones implementadas son las siguientes, y se encuentran en el fichero “filesystem.c”.

## mkFS()

Esta función genera el sistema de ficheros diseñado, teniendo en cuenta los metadatos introducidos. Para ello, se inicializan los metadatos a los valores por defecto del superbloque, los i-nodos y los mapas de bits. Para inicializar correctamente el sistema, se recorren todos los i-nodos, inicializando todos sus atributos. También se recorren los mapas de bits, para inicializar tanto el mapa de bloques, como el mapa de i-nodos a 0. Una vez inicializados todos los atributos, y añadido el relleno al superbloque, se escribe el sistema de ficheros del buffer al dispositivo. La función devuelve 0 en caso de ejecutarse correctamente, y -1 en caso contrario.

## mountFS()

El objetivo de esta función es el de montar el sistema de ficheros creado en el dispositivo disk.dat. Para ello, se procede al uso de la función bread, que lee un bloque del dispositivo y lo almacena en buffer. De esta forma, se almacena en buffer el superbloque, el mapa de bits y los i-nodos. La función devuelve 0 en caso de ejecutarse correctamente y -1 en caso contrario.

## unmountFS()

Esta función tiene como objetivo desmontar el sistema de ficheros situado en el dispositivo disk.dat. De esta forma, se liberan las variables y estructuras utilizadas para la creación del sistema de ficheros. Para ello se procede al uso de la función bwrite, el cual escribe un bloque del buffer en el dispositivo. De esta forma escribimos tanto el superbloque, como el mapa de bits y los i-nodos. La función devuelve 0 en caso de ejecutarse correctamente y -1 en caso contrario.

## ifree()

Esta función auxiliar comprueba la validez del i-nodo, es decir, si el i-nodo introducido por parámetro pertenece al superbloque. Si es el caso, se procede a liberar el i-nodo a través del mapa de bits de i-nodos, cambiando el valor de su posición en el mapa a 0.

## ialloc()

Esta función auxiliar tiene como objetivo buscar un i-nodo libre en el mapa de bits. Para ello, la función recorre el mapa de bits de i-nodos. Cuando encuentre un i-nodo sin ocupar, es decir, cuando se encuentre una posición en el mapa de i-nodos con el valor de bit 0, se establece el valor de ese bit a 1, cambiando el estado del i-nodo a ocupado, y actualizando sus atributos a sus valores por defecto. La función devuelve el id del i-nodo si se ha ejecutado correctamente, y -1 en caso de error.

## alloc()

Esta función es equivalente a la anterior, con la diferencia de que se busca un bloque disponible en el mapa de bloques. Cuando se encuentra un bloque disponible, con el valor a 0, se cambia el valor a 1 indicando que el bloque seleccionado está ocupado. Asimismo, se genera el bloque y se escribe del buffer al dispositivo.

## fileExist()

Esta función auxiliar tiene como objetivo comprobar que el nombre del archivo/enlace introducido por parámetro ya exista en el sistema de ficheros. Para ello, se recorren todos los i-nodos comprobando el atributo nombre. En caso de encontrar un nombre igual al nombre de alguno de los i-nodos, la función devolverá -1. En caso de no encontrar alguna coincidencia, devolverá 0.

## isNodeFull()

Este método auxiliar comprueba que no estén todos los i-nodos ocupados. Para ello, recorre todo el mapa de bits de i-nodos. Si encuentra alguna posición con el valor de bit 0, la función devolverá 0. Si el valor de todas las posiciones equivale a 1, significará que todos los i-nodos están ocupados. De esta forma, este método devolverá -2.

## isBlocksFull()

Este método es similar al anterior, con la diferencia de que se recorre el mapa de bits de bloque. En caso de que no se encuentre ningún bloque sin ocupar, el método devolverá -2. Si hay bloques sin ocupar, devolverá 0.

## createFile()

Esta es la función principal para crear un archivo en el sistema de ficheros. Para ejecutar la función, es necesario comprobar anteriormente que el archivo a crear no haya sido creado anteriormente en el sistema de ficheros, a través del nombre en el método auxiliar fileExist() detallado anteriormente. También es necesario comprobar que hay espacio en sistema de ficheros, a través de las funciones isNodeFull() y isBlocksFull(). Una vez comprobado, obtenemos el identificador del i-nodo libre encontrado a través de la función ialloc(). También obtenemos el bloque disponible a través de la función alloc(). Al tratarse de un archivo, se actualizan los atributos del i-nodo. Se actualiza el atributo “tipo” a 1 al tratarse de un archivo, se inicializan a 0 los atributos de “posición”, “abierto” y “tamaño”, y se actualizan los atributos de ”i-nodos contenidos” y “bloques ocupados”. De esta forma se crea un archivo vacío en el sistema de ficheros. La función devuelve 0 si se ha ejecutado correctamente, -1 si el archivo ya se había creado anteriormente, y -2 en otro caso de error.

## busca\_inodo()

Esta función auxiliar tiene como objetivo buscar un i-nodo por su nombre. La función devuelve el i-nodo en caso de haberlo encontrado, y -1 en caso de no encontrar ningún i-nodo con el nombre introducido por parámetro.

## removeFile()

Esta es la función principal para eliminar un archivo del sistema de ficheros. Para ello, es necesario buscar el i-nodo con el nombre introducido por parámetro, a través de la función auxiliar busca\_inodo(). Si no se encuentra el archivo, esta función devolverá -1. Si el archivo encontrado es un enlace simbólico, la función devolverá -2. Si el archivo tiene enlaces simbólicos que apuntan a él, se elimina el archivo y los enlaces simbólicos. Para eliminar el archivo, se modifica el valor del bit del i-nodo en el mapa de bits a 0, y se vuelven a actualizar los atributos del i-nodo a los valores por defecto.

## openFile()

Esta es la función principal para abrir un archivo creado en el sistema de ficheros. Para ello comprueba primero que el archivo que se quiere abrir haya sido creado. En caso contrario la función devolverá -1. A continuación, se busca el i-nodo donde esté situado el archivo. Si el i-nodo está fuera de rango o ya está abierto, la función devolverá -2. Si el archivo es un enlace simbólico, se actualiza el atributo “abierto” a 1 tanto del fichero al que referencia como del enlace simbólico. En caso contrario solo se actualiza el atributo del fichero. En ambos casos, la posición se actualiza al inicio del archivo, es decir, a 0. En caso de ejecutarse correctamente, la función devolverá el i-nodo del archivo abierto.

## closeFile()

Esta es la función principal para cerrar un archivo creado en el sistema de ficheros. Para ello comprueba primero que el descriptor de fichero se encuentre en el rango de i-nodos. En caso contrario la función devolverá -1. A continuación, se comprueba que el i-nodo no se encuentre ya cerrado. Si el i-nodo pasado por parámetro ya estaba cerrado anteriormente, la función devolverá -1. Si el archivo a cerrar es un enlace simbólico, se cierra también el fichero original, pero solo si se ha abierto a través del enlace simbólico. Si el archivo a cerrar es un fichero, simplemente actualiza el valor de “abierto” a 0. En ambos casos, se actualiza el valor de la posición a 0. Si la función se ha ejecutado correctamente, devolverá 0.

## readFile()

Función encargada de la lectura de un archivo. Primeramente, se comprueba si el archivo sobre el que se desea leer es un enlace simbólico, en ese caso se modifica el descriptor de fichero al referenciado por dicho enlace blando. Seguidamente, se comprueba si el fichero referenciado ha sido abierto previamente, en caso contrario la función retorna -1. A continuación se verifica si el i-nodo sobre el que se quiere leer tiene un fichero asociado, en caso contrario se retorna -1. Posteriormente, se ajusta el número de caracteres a leer al tamaño máximo del fichero y se calcula el número de bloques que se van a leer.  Ulteriormente, se accede al bucle for y se calcula el bloque asociado a la posición del fichero y mediante sentencias condicionales se identifica si el bloque de datos a leer es el calculado anteriormente, uno intermedio o el bloque de datos en el que finaliza la lectura. En el primero de los casos se calcula la longitud a leer, y se procede a ello. Para los otros casos condicionales el funcionamiento es similar, pero teniendo en cuenta los caracteres restantes de lectura. Finalmente se actualiza la posición del puntero de lectura retornando el número de caracteres leídos.

## writeFile()

Función encargada de la escritura de un archivo. Primeramente, se comprueba si el archivo sobre el que se desea escribir es un enlace simbólico, en ese caso se modifica el descriptor de fichero al referenciado por dicho enlace blando. Seguidamente, se comprueba si el fichero referenciado ha sido abierto previamente, en caso contrario la función retorna -1. A continuación se verifica si el i-nodo sobre el que se quiere escribir tiene un fichero asociado, en caso contrario se retorna -1. Posteriormente, se ajusta el número de caracteres a escribir al tamaño máximo del fichero, se calcula el número de bloques que se van a escribir y se procede a eliminar bloques que ya no van a ser usados, ya que cuando se escribe se sobrescribe lo que se encuentre después de la posición actual. Ulteriormente, se accede al bucle for y se calcula el bloque asociado a la posición del fichero y mediante sentencias condicionales se identifica si el bloque de datos a escribir es el calculado anteriormente, uno intermedio o el bloque de datos en el que finaliza la escritura. En el primero de los casos se calcula la longitud a escribir, se lee lo existente anteriormente en el bloque, se procede a la escritura teniendo en cuenta la posición actual y finalmente se escribe lo modificado de nuevo en su bloque correspondiente. Para los otros casos condicionales el funcionamiento es similar, pero teniendo en cuenta los caracteres restantes de escritura. Finalmente se actualiza la posición del puntero de lectura y el tamaño del i-nodo retornando el número de caracteres escritos.

## lseekFile()

Función encargada de modificar el puntero de lectura y escritura de un fichero. Primeramente, se comprueba que el descriptor de fichero introducido por parámetro sea correcto, que el valor de whence introducido sea correcto y que el i-nodo referenciado exista, en caso de que no se cumpla ninguna de estas condiciones se procede a retornar -1. A continuación, si el descriptor de fichero introducido pertenece a un enlace simbólico dicho descriptor se cambia por el referenciado por el enlace blando. Seguidamente se comprueba la operación a realizar mediante secuencias condicionales. En el caso de que el valor introducido sea SEEK\_CUR/0 se comprueba que el offset introducido viole los límites de fichero establecidos, si lo hiciese se retorna -1, en caso contrario se actualiza la posición del puntero de lectura y escritura del fichero sumando el offset.  En caso en el que el valor introducido sea SEEK\_END/1 se modifica el puntero al final del fichero escrito. Finalmente, en caso en el que el valor introducido para whence sea SEEK\_BEGIN/2 se modifica el puntero de lectura y escritura al inicio del archivo. Si todos estos casos se han realizado sin errores retornan 0, en caso contrario -1.

## checkFile()

Función encargada de comprobar la integridad de un fichero introducido por parámetro. Para ello se realizarán comprobaciones básicas previas, como que el fichero exista. Posteriormente se abrirá el fichero, si ya se encontraba abierto, se busca su descriptor correspondiente, se hará un backup de la posición para así poder resetearla a 0 y se leerá con readFile para almacenarlo en un buffer. Si el fichero se encontraba cerrado, se abrirá, se leerá y será guardado igualmente en el buffer y se cerrará. Una vez leído el archivo, se pasará dicho buffer a CRC32, comparando dicho valor con la integridad del i-nodo para así saber si se encuentra corrupto o no.

## includeIntegrity()

Función encargada de asignar la integridad de un fichero introducido por parámetro. Para ello se realizarán comprobaciones básicas previas, como que el fichero exista. Posteriormente se abrirá el fichero, si ya se encontraba abierto, se busca su descriptor correspondiente, se hará un backup de la posición para así poder resetearla a 0 y se leerá con readFile para almacenarlo en un buffer. Si el fichero se encontraba cerrado, se abrirá, se leerá y será guardado igualmente en el buffer y se cerrará. Una vez leído el archivo, se pasará dicho buffer a CRC32 y el resultado se sobrescribirá en la integridad del fichero correspondiente.

## openFileIntegrity()

Función encargada de abrir un fichero, comprobando que no está corrupto, para ello usará el mismo formato que openFile con su comprobación correspondiente con CheckFile. Además, si la integridad no ha sido calculada previamente no se podrá utilizar esta función, devolviendo un -3.

## closeFileIntegrity()

Función encargada de cerrar un fichero, añadiendo su nueva integridad, para ello usará el mismo formato que closeFile con su asignación correspondiente con includeIntegrity. Además, si el fichero no ha sido abierto previamente con openFileIntegrity no se podrá utilizar esta función, devolviendo un -3.

## createLn()

Esta es la función principal para crear un enlace simbólico a un archivo existente. Para ello, se comprueba que el fichero introducido por parámetro se encuentra en el sistema de ficheros. En caso contrario la función devuelve -1. También se comprueba que el nombre del enlace simbólico no esté ya en el sistema de ficheros. En caso contrario devolverá -1. Si no hay espacio en el mapa de i-nodos o bloques, la función devolverá -2. En caso de haber espacio, se obtiene el nuevo i-nodo y el bloque asignado para el enlace simbólico. Se comprueba si el enlace apunta a otro enlace. En caso afirmativo, la función devolverá -2. Por último, se busca el fichero al que referencia el enlace para actualizar los atributos del enlace simbólico. Para el enlace simbólico, se actualiza el atributo “tipo” a 0, así como la posición, el tamaño y el número de bloques ocupados. En i-nodos contenidos, se almacena el i-nodo del fichero al que referencia el enlace. La función devuelve el i-nodo del enlace en caso de que se haya ejecutado correctamente.

## removeLn()

Esta es la función principal para eliminar un enlace simbólico del sistema de ficheros. Para ello, se busca el i-nodo que se quiere eliminar del sistema de ficheros. En caso de que no se encuentre, la función devolverá -1. Si se ha encontrado, se comprueba que el i-nodo a eliminar es un enlace simbólico. En caso contrario, la función devolverá -2. Se actualiza el mapa de bits de i-nodos a 0 y se actualiza el valor de los atributos del i-nodo por defecto. La función devuelve 0 en caso de ejecutarse correctamente.

## fullInodeMap()

Esta función auxiliar ha sido implementada para la realización de algunas pruebas que se detallarán a continuación. Su objetivo es el de llenar el mapa de i-nodos, para comprobar el correcto funcionamiento de las funcionalidades principales.

## fullBlockMap()

Esta función auxiliar ha sido implementada para la realización de algunas pruebas que se detallarán a continuación. Su objetivo es el de llenar el mapa de bloques, para comprobar el correcto funcionamiento de las funcionalidades principales.

## printSystem()

Esta función auxiliar ha sido implementada para comprobar el funcionamiento del sistema a lo largo de la realización de pruebas del sistema. La función imprime el mapa de i-nodos y el mapa de bloques, así como los i-nodos con el valor de los atributos de cada uno de ellos.

# DISEÑO DE PRUEBAS

En este apartado se detallan las pruebas realizadas a lo largo de la implementación del sistema de ficheros. Los identificadores de cada prueba están relacionados con los requisitos proporcionados en el enunciado. Todas estas pruebas están implementadas en el fichero “test.c”.

## F1: Funcionalidades principales del sistema de ficheros

|  |  |
| --- | --- |
| **F1.1 (función mkFS)** | |
| DESCRIPCIÓN | Prueba básica de creación de un sistema de ficheros. |
| MÉTODO | Para comprobar la correcta ejecución del método, se ha creado un método auxiliar que imprime todos los datos del sistema de ficheros (printSystem). Como resultado, se debe inicializar el sistema de ficheros con los metadatos introducidos en el código. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 0, open 0, integridad 0, nombre /test1.txt, inodos Contenidos [0,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 1: tipo 1, pos 0, open 0, integridad 0, nombre /test2.txt, inodos Contenidos [1,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 2: tipo 1, pos 0, open 0, integridad 0, nombre /test3.txt, inodos Contenidos [2,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 3: tipo 1, pos 0, open 0, integridad 0, nombre /test4.txt, inodos Contenidos [3,-1,-1,-1,-1], tamaño 0, bloques ocupados 1 |

|  |  |
| --- | --- |
| **F1.2 (función mountFS)** | |
| F1.2.1 | |
| DESCRIPCIÓN | Prueba básica para montar un sistema de ficheros. |
| MÉTODO | Para comprobar la correcta funcionalidad de la función mountFS, se crea el disco antes de ejecutar el test, introduciendo ./create\_disk 300. De esta forma, se genera el dispositivo disk.dat. |
| RESULTADO OBTENIDO | A través del comando hexdump -C disk.dat, se puede observar que el dispositivo ha sido montado correctamente |
| F1.2.2 | |
| DESCRIPCIÓN | Prueba a montar el sistema de ficheros sin crearlo anteriormente |
| MÉTODO | Para ello procedemos a realizar la misma prueba que en el apartado F1.2.1, con la diferencia de no ejecutar la función mkFS. |
| RESULTADO OBTENIDO | Al no haberse creado el sistema de ficheros, el comando hexdump -C disk.dat, devolverá el sistema de ficheros montado incorrectamente en el sistema. |

|  |  |
| --- | --- |
| **F1.3 (función unmountFS)** | |
| F1.3.1 | |
| DESCRIPCIÓN | Prueba básica para desmontar el sistema de ficheros. |
| MÉTODO | Para la ejecución de esta prueba, se crea y se monta el sistema de ficheros previamente, a través de las funciones mkFS y mountFS. |
| RESULTADO OBTENIDO | La función unmountFS se ejecutará correctamente, ya que el sistema de ficheros se ha creado y montado previamente en el dispositivo disk.dat, por lo que no resulta ningún problema desmontar el sistema del dispositivo. |
| F1.3.2 y F1.3.3 | |
| DESCRIPCIÓN | Pruebas para desmontar el sistema de ficheros, sin haberse creado previamente, y sin haberse montado previamente. |
| MÉTODO | Para la ejecución de esta prueba, no se accede a la función mkFS anteriormente, ni a la función mountFS, por lo que no se crea ni se monta el sistema de ficheros. |
| RESULTADO OBTENIDO | Como es de esperar, la función unmountFS dará un error, ya que el sistema de ficheros no ha sido creado, resultado imposible incluso montar el sistema de ficheros. |

|  |  |
| --- | --- |
| **F1.4 (función createFile)** | |
| F1.4.1 | |
| DESCRIPCIÓN | Prueba básica de creación de archivos. |
| MÉTODO | Para la ejecución de este test, se ha procedido a crear 4 archivos de nombres distintos. De esta forma se crearán los 4 archivos a través de la función createFile. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 0, open 0, integridad 0, nombre /test1.txt, inodos Contenidos [0,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 1: tipo 1, pos 0, open 0, integridad 0, nombre /test2.txt, inodos Contenidos [1,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 2: tipo 1, pos 0, open 0, integridad 0, nombre /test3.txt, inodos Contenidos [2,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 3: tipo 1, pos 0, open 0, integridad 0, nombre /test4.txt, inodos Contenidos [3,-1,-1,-1,-1], tamaño 0, bloques ocupados 1 |
| F1.4.2 | |
| DESCRIPCIÓN | Prueba en la que se crea un archivo con nombre repetido. |
| MÉTODO | En esta prueba se crean 4 archivos, 2 con el mismo nombre, por lo que no se deberá de crear el archivo con el nombre repetido. |
| RESULTADO OBTENIDO | El resultado obtenido es parecido al del apartado F1.4.1, con la diferencia de que hay un archivo que no se ha creado, el archivo con el nombre repetido. |
| F1.4.3 y F1.4.4 | |
| DESCRIPCIÓN | Pruebas en la cual se intenta crear un archivo sin nodos disponibles, y sin bloques disponibles. |
| MÉTODO | Para la ejecución de estas pruebas, necesitamos las funciones auxiliares fullInodeMap() y fullBlockMap(). Con esta función podemos probar que no se crea un archivo si el mapa de i-nodos está lleno. |
| RESULTADO OBTENIDO | La función createFile devuelve -2, esto significa que no hay espacio suficiente en el sistema de ficheros, por lo que el archivo no se crea. |

|  |  |
| --- | --- |
| **F1.5 (función removeFile)** | |
| F1.5.1 | |
| DESCRIPCIÓN | Prueba básica de eliminación de un archivo existente. |
| MÉTODO | Para la ejecución de esta prueba, se han creado varios archivos, y se ha procedido a eliminar el primero de ellos a través de la función removeFile. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 0, open 0, integridad 0, nombre, inodos Contenidos [-1,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 1: tipo 1, pos 0, open 0, integridad 0, nombre /test2.txt, inodos Contenidos [1,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 2: tipo 1, pos 0, open 0, integridad 0, nombre /test3.txt, inodos Contenidos [2,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 3: tipo 1, pos 0, open 0, integridad 0, nombre /test4.txt, inodos Contenidos [3,-1,-1,-1,-1], tamaño 0, bloques ocupados 1 |
| F1.5.2 | |
| DESCRIPCIÓN | Prueba de eliminación de un archivo no creado anteriormente. |
| MÉTODO | Para la ejecución de esta prueba, se han creado varios archivos, y se ha llamado a la función removeFile de un archivo no creado. |
| RESULTADO OBTENIDO | Como se esperaba, el resultado de la prueba ha sido satisfactorio, debido a que la función ha devuelto -1 debido a que no ha encontrado el archivo. |
| F1.5.3 | |
| DESCRIPCIÓN | Prueba de eliminación de un enlace simbólico a través de la función removeFile. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un archivo y un enlace simbólico que referencia a ese archivo. Se llama a la función removeFile metiendo por parámetro el enlace creado. |
| RESULTADO OBTENIDO | Como se esperaba, la función removeFile devuelve -2, eso significa que se ha intentado eliminar un enlace simbólico a través del método de eliminación de archivos, por lo que el enlace simbólico indicado por parámetro no se eliminará. |
| F1.5.4 | |
| DESCRIPCIÓN | Prueba que comprueba la eliminación de un archivo, eliminando también los enlaces simbólicos que le apuntaban. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un archivo, además de dos enlaces simbólicos que apuntan a dicho archivo. Se llama a la función eliminando el archivo. |
| RESULTADO OBTENIDO | Como se comprueba a la salida de la ejecución a través de printSystem, se ha eliminado tanto el archivo como los enlaces simbólicos que le referenciaban, por lo que el resultado de esta prueba es satisfactorio. |

|  |  |
| --- | --- |
| **F1.6 (función openFile)** | |
| F1.6.1 | |
| DESCRIPCIÓN | Prueba básica para abrir un archivo existente. |
| MÉTODO | Para la ejecución de esta prueba, se crean dos archivos distintos, y se llama a la función openFile en ambos. Como resultado, los dos archivos creados deben abrirse sin problema. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 0, open 1, integridad 0, nombre /test1.txt, inodos Contenidos [0,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 1: tipo 1, pos 0, open 1, integridad 0, nombre /test2.txt, inodos Contenidos [1,-1,-1,-1,-1], tamaño 0, bloques ocupados 1 |
| F1.6.2 | |
| DESCRIPCIÓN | Prueba para abrir un archivo ya abierto anteriormente. |
| MÉTODO | Para ejecutar esta prueba, se ha creado un archivo y se ha abierto dos veces. |
| RESULTADO OBTENIDO | Como se puede comprobar, el archivo solo se ha abierto una vez, ya que se ha ejecutado el primer openFile correctamente, y el segundo openFile ha devuelto -2, ya que el archivo se encontraba abierto. |
| F1.6.3 | |
| DESCRIPCIÓN | Prueba para abrir un archivo no creado anteriormente. |
| MÉTODO | Para la ejecución de esta prueba, simplemente no se ha creado ningún fichero, por lo que se llama a la función openFile con un archivo arbitrario e inexistente. |
| RESULTADO OBTENIDO | Como era de esperar, no se puede abrir un archivo que no se encuentre en el sistema de ficheros, por lo que la función devuelve -1. |
| F1.6.4 | |
| DESCRIPCIÓN | Prueba para abrir un fichero a través de un enlace simbólico |
| MÉTODO | Para la ejecución de la prueba, se crea un archivo y un enlace simbólico que apunte a ese archivo. Se llama a la función openFile a través del enlace simbólico. |
| RESULTADO OBTENIDO | El resultado obtenido es que tanto el fichero como el enlace simbólico se encuentran abiertos por la función openFile, por lo que la prueba ha dado un resultado satisfactorio. |

|  |  |
| --- | --- |
| **F1.7 (función closeFile)** | |
| F1.7.1 | |
| DESCRIPCIÓN | Prueba básica para cerrar un descriptor de fichero. |
| MÉTODO | Para proceder con esta prueba, se ha creado un archivo, se ha abierto, y finalmente se ha cerrado llamando a la función closeFile. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 0, open 0, integridad 0, nombre /test1.txt, inodos Contenidos [0,-1,-1,-1,-1], tamaño 0, bloques ocupados 1 |
| F1.7.2 | |
| DESCRIPCIÓN | Prueba para cerrar un descriptor de fichero ya cerrado o no creado. |
| MÉTODO | Para la ejecución de esta prueba, se ha llamado a la función closeFile sin haber creado ningún fichero anteriormente. |
| RESULTADO OBTENIDO | Como era de esperar, la función devuelve -1, ya que no se encuentra inicializado el descriptor de fichero, es decir, el valor de “open” es 0. |
| F1.7.3 | |
| DESCRIPCIÓN | Prueba para cerrar descriptor de fichero fuera de rango. |
| MÉTODO | Para esta prueba, se intenta cerrar el descriptor de fichero fuera del rango del sistema de ficheros. |
| RESULTADO OBTENIDO | Como resultado, la función closeFile devuelve -1, debido a que el i-nodo que se quiere cerrar está fuera de rango. La prueba ha resultado satisfactoria. |
| F1.7.4 | |
| DESCRIPCIÓN | Prueba para cerrar descriptor de fichero a través de enlace simbólico. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un archivo y un enlace simbólico que referencia al archivo. Posteriormente, se ha abierto el enlace a través de openFile. Por último, se ha cerrado el enlace a través de closeFile. |
| RESULTADO OBTENIDO | Se puede observar a través de la función printSystem(), que se ha cerrado tanto el archivo como el enlace simbólico. Esto quiere decir que la prueba ha sido superada. |

|  |  |
| --- | --- |
| **F1.8 y F1.9 (función readFile y writeFile)** | |
| F1.8.1 y F1.9.1 | |
| DESCRIPCIÓN | Pruebas básicas de lectura y escritura de un fichero abierto. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un fichero y se ha abierto con la función openFile. Posteriormente se ha escrito en el fichero a través de la función writeFile, y se ha leído el texto con readFile. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 18, open 1, integridad 0, nombre /test1.txt, inodos Contenidos [0,-1,-1,-1,-1], tamaño 19, bloques ocupados 1  A través del comando hexdump -C disk.dat, se obtiene el bloque de datos actualizado. |
| F1.8.2, F1.9.2, F1.8.3 y F1.9.3 | |
| DESCRIPCIÓN | Pruebas para intentar leer y escribir un fichero que no ha sido abierto anteriormente o no existe. |
| MÉTODO | Para la ejecución de las dos primeras pruebas, se ha creado un fichero, pero no se ha abierto para la lectura y escritura. Para las dos segundas, no se ha creado ningún archivo. |
| RESULTADO OBTENIDO | Como era de esperar, las funciones readFile y writeFile necesitan que el fichero esté abierto, por lo que ambas devolverán -1 y no se ejecutarán correctamente. Asimismo, si el i-nodo al que acceden las funciones no está inicializado por ningún archivo, estas funciones también devolverán -1. |
| F1.8.4 y F1.9.4 | |
| DESCRIPCIÓN | Prueba en la que se lee y escribe en un fichero que ocupa varios bloques. |
| MÉTODO | Para estas pruebas, se ha creado un fichero y se ha abierto a través de la función openFile. Además, se ha escrito una cadena de caracteres extensa para que el contenido del fichero ocupe varios bloques. |
| RESULTADO OBTENIDO | Utilizando el comando hexdump -C disk.dat, podemos observar que, tanto la lectura como la escritura del fichero se han ejecutado correctamente. |
| F1.8.5 y F1.9.5 | |
| DESCRIPCIÓN | Prueba en la que se lee y se escribe varias veces el mismo fichero. |
| MÉTODO | Para la ejecución de estas pruebas, se ha creado un fichero, y posteriormente se ha llamado a la función writeFile y readFile varias veces. |
| RESULTADO OBTENIDO | Como podemos comprobar, tanto las lecturas como las escrituras se ejecutan correctamente, ya que no se sobrescriben unas con otras. |
| F1.8.6, F1.9.6, F1.8.7 y F1.9.7 | |
| DESCRIPCIÓN | Pruebas en las que se lee y escribe en varios ficheros abiertos, diferentes cadenas de caracteres. |
| MÉTODO | Para la ejecución de estas pruebas, se han creado varios ficheros, y se han leído y escrito diferentes cadenas de caracteres en cada uno de ellos. Para las pruebas F1.8.7 y F1.9.7, la primera cadena de caracteres ocupa un bloque. |
| RESULTADO OBTENIDO | Como podemos comprobar, tanto las lecturas como las escrituras se ejecutan correctamente. |

|  |  |
| --- | --- |
| **F1.10 (función lseekFile)** | |
| F1.10.1, F1.10.2, F1.10.3 | |
| DESCRIPCIÓN | Pruebas básicas de lseekFile, para los 3 casos disponibles. |
| MÉTODO | Para la ejecución de estas pruebas, se ha creado un fichero, se ha abierto con la función openFile, y se ha escrito el contenido del buffer con la función writeFile. La primera prueba ejecuta FS\_SEEK\_CUR, la segunda FS\_SEEK\_END, y la tercera FS\_SEEK\_BEGIN. |
| RESULTADO OBTENIDO | Como se puede comprobar en los atributos del i-nodo, el valor del puntero de posición se actualiza según el lseekFile ejecutado. |
| F1.10.4, F1.10.5, F1.10.6 | |
| DESCRIPCIÓN | Pruebas de ejecución de lseekFile con datos introducidos incorrectamente, e i-nodos inexistentes. |
| MÉTODO | Para la ejecución de las dos primeras pruebas, se ha creado un fichero, se ha abierto y se ha escrito el contenido del buffer. Para la segunda, no se ha creado ningún fichero. |
| RESULTADO OBTENIDO | Para todos los casos, la función lseekFile devolverá -1, ya que la función comprueba si el fichero existe, y si los datos introducidos por parámetro son correctos. |

|  |  |
| --- | --- |
| **F1.11 (función checkFile)** | |
| F1.11.1 | |
| DESCRIPCIÓN | Prueba que verifica la integridad de un fichero no existente |
| MÉTODO | Se realiza checkFile a un fichero no existente |
| RESULTADO OBTENIDO | La función devuelve -2 para reflejar el error |
| F1.11.2 y F1.11.3 | |
| DESCRIPCIÓN | Prueba de comprobar integridad a un fichero al que previamente se le ha asignado integridad. Se hace una comprobación dejando el fichero abierto y otra con el fichero cerrado. |
| MÉTODO | Escribir un texto de prueba en el archivo, calcular su integridad y posteriormente realizar el checkFile. |
| RESULTADO OBTENIDO | El resultado obtenido debe de devolver 0, ya que los archivos no están corruptos. Además tendrán que dejar el archivo igual que como entró a la función checkFile (cerrado o abierto y con el mismo puntero) |
| F1.11.4 y F1.11.5 | |
| DESCRIPCIÓN | Prueba de comprobar integridad a un fichero al que previamente se le ha asignado integridad y posteriormente se le ha cambiado el buffer. Se hace una comprobación dejando el fichero abierto y otra con el fichero cerrado. |
| MÉTODO | Escribir un texto de prueba en el archivo, calcular su integridad, volver a escribir en el archivo y posteriormente realizar el checkFile. |
| RESULTADO OBTENIDO | El resultado obtenido debe de devolver -1, ya que los archivos están corruptos. Además tendrán que dejar el archivo igual que como entró a la función checkFile (cerrado o abierto y con el mismo puntero) |

|  |  |
| --- | --- |
| **F1.12 (función includeIntegrity)** | |
| F1.12.1, F1.12.2 y F1.12.3 | |
| DESCRIPCIÓN | Pruebas para comprobar el añadir integridad a un fichero |
| MÉTODO | Para la ejecución de esta prueba, se han realizado todas los casos de que el fichero a añadir integridad no exista, que no esté abierto y que se encuentre abierto. |
| RESULTADO OBTENIDO | En todas las pruebas hemos obtenido el resultado esperado, dando un error negativo en las pruebas donde se intenta añadir integridad a un fichero no abierto y no existente y un resultado positivo en la prueba restante. |

|  |  |
| --- | --- |
| **F1.13 (función OpenFileIntegrity)** | |
| F1.13.1, F1.13.2 y F1.13.3 | |
| DESCRIPCIÓN | Pruebas que verifican la detección de los errores de que no exista integridad previamente, que no exista el archivo y que el archivo ya se encuentre abierto. |
| MÉTODO | Se realiza OpenFileIntegrity para los casos anteriores |
| RESULTADO OBTENIDO | La función devuelve valores negativos en todos estos casos para indicar el error. |
| F1.13.4 y F1.13.5 | |
| DESCRIPCIÓN | Pruebas que verifican el correcto funcionamiento de la función a través de tanto enlaces simbólicos como de acceso normal. |
| MÉTODO | Escribir un texto de prueba en el archivo, calcular su integridad, cerrar el archivo y  realizar el OpenFileIntegrity en ambos casos. |
| RESULTADO OBTENIDO | El resultado obtenido debe devolver 0, ya que los archivos tienen el formato correcto. Al finalizar estos archivos tendrán su integridad comprobada. |
| F1.13.6 | |
| DESCRIPCIÓN | Prueba que verifica que si la integridad de un archivo varía el OpenFileIntegrity  detectará este cambio. |
| MÉTODO | Escribir un texto de prueba en el archivo, calcular su integridad, cerrar el archivo , calcular la integridad, volver a escribir en él y realizar el OpenFileIntegrity |
| RESULTADO OBTENIDO | El resultado obtenido debe devolver -2, ya se ha escrito después de haber calculado la integridad y por tanto  OpenFileIntegrity lo detectará. |

|  |  |
| --- | --- |
| **F1.14 (función closeFileIntegrity)** | |
| F1.14.1 | |
| DESCRIPCIÓN | Prueba para cerrar un descriptor de fichero y calcular su integridad. |
| MÉTODO | Para esta prueba, se ha creado y abierto un fichero. Posteriormente se ha escrito el contenido del buffer en el fichero. Por último, se ha cerrado el fichero con closeFile. Se ha incluido la integridad del fichero y se ha llamado a la función openFileIntegrity. |
| RESULTADO OBTENIDO | Como resultado, al llamar a la función closeFileIntegrity, el archivo se cerrará correctamente. |
| F1.14.2 y F1.14.3 | |
| DESCRIPCIÓN | Pruebas para cerrar un descriptor de fichero ya cerrado, no inicializado o fuera de rango. |
| MÉTODO | Para la ejecución de estas pruebas, no se ha creado ningún fichero. Simplemente se ha llamado a la función closeFileIntegrity desde un i-nodo sin inicializar, y desde un i-nodo fuera del sistema de ficheros. |
| RESULTADO OBTENIDO | Como era de esperar, la función closeFileIntegrity devolverá -1, debido a que no existe el fichero, o el i-nodo se encuentra fuera de rango. |
| F1.14.4 | |
| DESCRIPCIÓN | Prueba en la cual se cierra un archivo a través del enlace simbólico mediante integridad. |
| MÉTODO | Para la ejecución de esta prueba se ha creado un fichero y un enlace simbólico que referencia al fichero. A continuación, se ha abierto el fichero a través del enlace y se ha escrito el contenido del buffer. Se ha cerrado el fichero y se ha introducido la integridad. Por último se ha abierto el fichero con integridad a través del enlace. |
| RESULTADO OBTENIDO | Como resultado, al abrir el fichero con integridad a partir del enlace, la función closeFileIntegrity se ejecutará correctamente. |

|  |  |
| --- | --- |
| **F1.15 (función createLn)** | |
| F1.15.1 | |
| DESCRIPCIÓN | Prueba básica de creación de un enlace simbólico de un fichero. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un fichero y un enlace simbólico que apunta al fichero. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 0, open 0, integridad 0, nombre /test1.txt, inodos Contenidos [0,-1,-1,-1,-1], tamaño 0, bloques ocupados 1  Inodo 1: tipo 0, pos 0, open 0, integridad 0, nombre enlace1, inodos Contenidos [0,-1,-1,-1,-1], tamaño 0, bloques ocupados 0 |
| F1.15.2 | |
| DESCRIPCIÓN | Prueba de creación de un enlace simbólico de un fichero inexistente. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un enlace simbólico que apunta a un fichero sin crear. |
| RESULTADO OBTENIDO | Como resultado, la función createLn devuelve -1, debido a que el fichero al que se referencia es inexistente. Debido a ello, el enlace no se crea. |
| F1.15.3 y F1.15.4 | |
| DESCRIPCIÓN | Prueba de creación de un enlace simbólico sin i-nodos disponibles y sin bloques disponibles. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un fichero y un enlace simbólico que apunta al fichero. Además se ha llamado a la función fullInodeMap y fullBlockMap. |
| RESULTADO OBTENIDO | Al igual que con createFile, si el mapa de i-nodos está lleno, o el mapa de bloques está lleno, la función devolverá -1, por lo que no se creará el enlace simbólico. |
| F1.15.5 | |
| DESCRIPCIÓN | Prueba de creación de un enlace simbólico apuntando a otro enlace. |
| MÉTODO | Para la ejecución de esta prueba, se crea un archivo y un enlace que apunta a ese archivo. Además, se crea un enlace que apunta al enlace creado anteriormente. |
| RESULTADO OBTENIDO | Como era de esperar, la función devuelve -2, ya que no es posible crear un enlace que apunte a otro enlace. De esta forma, la prueba realizada ha resultado satisfactoria. |
| F1.15.6 | |
| DESCRIPCIÓN | Prueba en la cual se repite el nombre del enlace que se desea crear. |
| MÉTODO | Para esta prueba, se crean varios ficheros con varios enlaces simbólicos, pero estos enlaces contienen el mismo nombre. |
| RESULTADO OBTENIDO | Los enlaces con nombres repetidos no se crean, ya que la función devuelve -1, por lo que el resultado de esta prueba es satisfactorio. |
| F1.15.7 y F 1.15.8 | |
| DESCRIPCIÓN | Prueba de lectura y escritura en un enlace simbólico. |
| MÉTODO | Para estas pruebas, se crean un archivo y un enlace simbólico que apunta a ese archivo. Al leer y escribir a través del enlace simbólico, se debe acceder al fichero. |
| RESULTADO OBTENIDO | Como se puede comprobar, se actualizan los atributos del fichero y no del enlace simbólico, por lo que la prueba indica una correcta implementación. |
| F1.15.9 | |
| DESCRIPCIÓN | Prueba de lseekFile en un enlace simbólico. |
| MÉTODO | Para esta prueba, se crea un fichero y un enlace simbólico que apunta al fichero. Al acceder a lseekFile a través de un enlace simbólico, se debe actualizar la posición del fichero al que apunta. |
| RESULTADO OBTENIDO | El resultado obtenido es equivalente a que el fichero al que apunta el enlace simbólico ha sido actualizado, por lo que la función ha sido implementada correctamente. |

|  |  |
| --- | --- |
| **F1.16 (función removeLn)** | |
| F1.16.1 | |
| DESCRIPCIÓN | Prueba básica de eliminación de un enlace simbólico de un archivo existente. |
| MÉTODO | Para esta prueba, se ha creado un fichero y un enlace que referencia a dicho fichero. Por último se ha llamado a la función removeLn con el enlace creado. |
| RESULTADO OBTENIDO | Inodo 0: tipo 1, pos 0, open 0, integridad 0, nombre /test1.txt, inodos Contenidos [0,-1,-1,-1,-1], tamaño 0, bloques ocupados 1 |
| F1.16.2 | |
| DESCRIPCIÓN | Prueba de eliminación de un enlace inexistente. |
| MÉTODO | Para la ejecución de esta prueba se han creado un fichero y un enlace que referencia al fichero. Por último, se ha llamado a la función removeLn pasando por parámetro un enlace simbólico que no se ha creado. |
| RESULTADO OBTENIDO | Como resultado, la función removeLn devolverá -1, ya que la función no ha encontrado el i-nodo del enlace, que es inexistente. Por lo tanto, la prueba ha sido superada satisfactoriamente. |
| F1.16.3 | |
| DESCRIPCIÓN | Prueba de eliminación de un archivo a través de la función removeLn. |
| MÉTODO | Para la ejecución de esta prueba, se ha creado un fichero y un enlace a ese fichero. Por último, se ha llamado a la función removeLn intentando eliminar el fichero. |
| RESULTADO OBTENIDO | El resultado obtenido es un error de la función removeLn, como era de esperar, ya que no se puede eliminar un fichero con la función removeLn, que solo sirve para eliminar enlaces simbólicos. |

## NF1: Comprobación de requisitos no funcionales

|  |  |
| --- | --- |
| **NF (Comprobación entre diferentes opens y closes)** | |
| NF10.1, NF10.2, NF11.1, NF11.2, NF12.1, NF12.2 | |
| DESCRIPCIÓN | Pruebas para comprobar que un archivo no puede ser ni abierto ni cerrado con un open/close que no tengan ambos integridad o que no las tengan, por lo que por ejemplo, no se puede mezclar openFile y closeFileIntegrity. |
| MÉTODO | Para la ejecución de esta prueba, se han realizado todas las combinaciones posibles de los diferentes tipos de close y open para comprobar su correcto funcionamiento. |
| RESULTADO OBTENIDO | En todas las pruebas hemos obtenido el resultado esperado, dando un error negativo en las pruebas donde se mezclan open y close con integridad y sin ella y dando un resultado positivo cuando no se mezclan. |

# CONCLUSIONES

En general, esta práctica nos ha servido para reforzar los conocimientos obtenidos en la asignatura, concretamente en los apartados de sistemas de ficheros y memoria, así como para comprender mejor la estructura y el funcionamiento del sistema operativo.

Hemos tenido varios problemas, principalmente a la hora de implementar la función WriteFile() debido a que por un error de punteros la escritura resultante estaba formada por numerosas impresiones por pantalla realizadas a lo largo del código.

Estos problemas se han resuelto repasando las transparencias y los pseudocódigos de los ejercicios resueltos, junto al repaso de dudas de las últimas clases prácticas.

Por último, no tenemos ninguna queja con la práctica, ya que hemos comprendido mejor el funcionamiento del sistema operativo, así como la implementación de diversos sistemas de ficheros en lenguaje C.