Grado en Ingeniería Informática

Diseño de Sistemas Operativos

Curso 2018-2019

Grupo 83

Práctica – 2

SISTEMA DE FICHEROS

David Maroto Gómez – 100363850 – 100363850@alumnos.uc3m.es

Sabrina Riesgo Reyes – 100363834 – 100363834@alumnos.uc3m.es

Rafael Rus Rus – 100363907 – 100363907@alumnos.uc3m.es

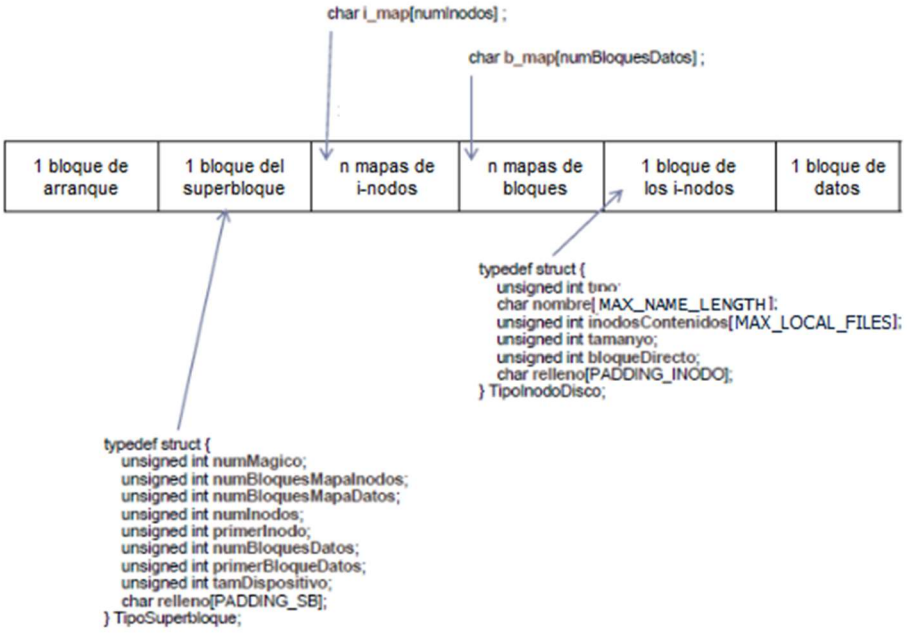
ÍNDICE

1. INTRODUCCIÓN

El contenido de este documento pretende analizar los resultados obtenidos durante la realización de la práctica 2 de la asignatura consistente en desarrollar un nuevo sistema de ficheros simplificado en espacio de usuario utilizando el lenguaje de programación C sobre la máquina virtual, cuyo almacenamiento estará respaldado por un fichero sobre el sistema operativo en el que se ejecute el programa.
Para la elaboración de esta práctica se ha utilizado el material dado por los profesores donde encontrábamos la especificación de la funcionalidad (constituido por las funciones que serán implementadas), así como los conceptos vistos en la asignatura que nos han ayudado a diseñar la arquitectura del sistema de ficheros, sus estructuras de control y los algoritmos necesarios para asegurar el cumplimiento de todos los requisitos pedidos en el enunciado de la práctica.
En los siguientes apartados de este documento se comentarán también las decisiones tomadas en el diseño y su implementación, así como la batería de pruebas realizadas para satisfacer el funcionamiento deseado. Por último, finalizaremos con las conclusiones sacadas de este trabajo, así como las dificultades encontradas.

2. DISEÑO DETALLADO DEL SISTEMA DE FICHEROS

A continuación, vamos a proceder con la explicación del diseño detallado del sistema de ficheros que hemos implementado, la cual consta de las siguientes partes: arquitectura, estructuras de control (i-nodos, superbloques y descriptores de fichero), y algoritmos de bajo y alto nivel del sistema de archivos vistos en la asignatura y necesarios para afirmar el cumplimiento de todos los requisitos y características. Para ello, explicaremos cuáles son los pasos del diseño que hemos realizado y llevado a cabo, así como su justificación y planteamiento.
Vamos a empezar detallando cómo hemos diseñado la organización en disco y cómo son las estructuras de control que hemos realizado e implementado (la mayor parte está en el fichero *include/metadata.h* tal y como se nos pedía en el enunciado de la práctica).
Nuestra propuesta de diseño de estructura en disco es la siguiente:



- 1 bloque o registro de arranque principal, que consiste en un sector dentro de un disco duro que contiene el código de arranque del sistema de ficheros.
- 1 bloque del superbloque, que consiste en un struct TipoSuperbloque definido dentro de *include/metadata.h* y que está formado por los siguientes atributos: numMagico (número mágico del superbloque, que será 1234); numBloquesMapaInodos (número de bloques del mapa inodos); numBloquesMapaDatos (número de bloques del mapa datos); numIN odos (número de i-nodos en el dispositivo, al cual vamos a dar un valor MAX_FILES de 40 cumpliendo así con el requisito **NF1** en el que se nos pedía que el máximo numero de ficheros en el sistema de ficheros no será nunca mayor de 40); primerInodo (número bloque del primer i-nodo del dispositivo, es decir, el i-nodo raíz); numBloquesDatos (es el número de bloques de datos en el dispositivo, al cual vamos a dar un valor de MAX_ FILES de 40 también cumpliendo con el requisito **NF1**); primerBloqueDatos (número de bloques del primer bloque de datos, al cual vamos a dar un valor de 2); tamDispositivo (tamaño total del dispositivo en bytes y que será dado mediante la función mkFS); y el atributo relleno (que consiste en el campo de relleno para completar un bloque y al cual le hemos dado un valor de PADDING_SB igual a 2016, resultado de restar 2048 - 32 que son los bytes que ocupan el resto de los atributos).

  Este bloque del superbloque ocupa una capacidad máxima de 2048 bytes, por lo que cumple con el requisito **NF7** en el que se nos pedía que el tamaño del bloque del sistema fuera de 2048 bytes.
- n mapas de i-nodos (i_map[numInodos]), que consisten cada uno en un array de chars con máxima capacidad para 40 i-nodos, y que están definidos en el fichero *metadata.h* para ser utilizados por el bloque del superbloque como almacenamiento de i-nodos para saber lo que está en uso y no.
- n mapa de bloques (b_map[numBloquesDatos]), que consisten cada uno en un array de chars con máxima capacidad para 40 superbloques, y que están definidos en el fichero *metadata.h* para ser utilizados por el bloque del superbloque como almacenamiento de superbloques para saber lo que está en uso y no.
- 1 bloque de los i-nodos, que consiste en un struct TipoInodoDisco definido dentro de *in clude/metadata.h* y que está formado por los siguientes atributos: tipo (especifica si el i-nodo es un fichero con un 0, o si es un directorio con un 1); nombre (atributo que sirve para almacenar el nombre de un fichero o directorio, y que tiene un tamaño MAX_NAME _LENGTH igual a 132, de esta forma especificamos que se cumple el requisito **NF4** en el que se nos pedía que la longitud máxima de la ruta completa de un fichero incluyendo su nombre será de 132 caracteres, incluyendo los separadores '/', y será de 99 dígitos en el caso de un directorio - luego nos aseguramos de que el atributo nombre tenga un tamaño máximo de 132 caracteres, independientemente de si es un directorio o bien un fichero); inodosContenidos (lista de los i-nodos del directorio, con una capacidad máxima de MAX_LOCAL_FILES igual a 10, cumpliéndose así el requisito **NF2** en el que nos pedía que el máximo número de elementos en un directorio no será nunca mayor de 10); el atributo tamanyo (tamaño actual del fichero en bytes que inicialmente es 0); el atributo bloqueDirecto (que especifica el numero del bloque directo al que hace referencia); el atributo relleno (que consiste en el campo de relleno para completar un i-nodo y al cual le hemos dado un valor de PADDING_INODO igual a 1964, resultado de restar 2048 - 84 que son los bytes que ocupan el resto de los atributos); y el atributo numElementos (que indica el número de elementos asociados a ese i-nodo).

  Este bloque de los i-nodos ocupa una capacidad máxima de 2048 bytes, por lo que cumple con el requisito **NF6** en el que se nos pedía que el tamaño máximo de un fichero será de un bloque (es decir de 2048 bytes).

- n bloques de datos, utilizados por el resto de estructuras de control.

Como podemos observar, en el diseño de la estructura en disco elaborado también se ha tenido en cuenta el cumplimiento de los requisitos **NF10** (en el que se nos exigía que en el programa cliente debería minimizarse el espacio en disco del tamaño de todos los metadatos del sistema, en el cual ningún elemento de control excede los 2048 bytes) y **NF11** (en el que se nos pedía que aprovechásemos al máximo los recursos disponibles en el metadata).

Después de la propuesta de diseño de estructura en disco elaborada anteriormente, se van a crear las variables para las estructuras en memoria. En este caso, vamos a tener en cuenta dos tipos de información:

- Información leída del disco, constituida por la arquitectura que va a ser utilizada en el programa cliente, que son el mapa de i-nodos (i_map[numInodos]); el mapa de bloques (b_map[numBloquesDatos]); un array de bloques del superbloque, que está inicializado en el *filesystem.c*, va a ser nombrado como sbloques y tendrá una capacidad de 1; y un array de bloques de los i-nodos inicializado en *filesystem.c*, que va a ser nombrado como inodos y que tendrá una capacidad máxima de numInodos igual a 40.
- Información extra de apoyo, constituida por la arquitectura extra que se va utilizar para complementar el programa cliente con la información leída del disco, que es un struct nuevo denominado inodos_x con una capacidad máxima de numInodos igual a 40 y que tiene los atributos puntero y abierto.

Tras ello, procedemos a describir el diseño de las rutinas de gestión, que sería la lectura y escritura en disco de las estructuras en memoria, es decir, el conjunto de algoritmos de bajo y alto nivel del sistema de archivos. Estas serían las funciones ialloc, alloc, ifree Aux, freeAux, namei, bmap y syncAux.

Función ialloc: función que busca un i-nodo libre, actualiza los valores por defecto en el i-nodo y devuelve el identificador de i-nodo. Necesario para las operaciones de lectura y escritura del sistema de ficheros.

Función alloc: función que busca un bloque libre, actualiza los valores por defecto en el bloque y devuelve el identificador de i-nodo. Necesario para las operaciones de lectura y escritura del sistema de ficheros.

Función ifreeAux: función que comprueba la validez del i-nodo inodo_id y luego lo libera.

Función freeAux: función que comprueba la validez del bloque inodo_id y luego lo libera.

Función namei: función que comprueba que el nombre de un i-nodo concuerda con uno solicitado por la función.

Función bmap: función que comprueba la validez del i-nodo inodo_id y va a devolver el bloque de datos asociado.

Función syncAux: función que se encarga de escribir los valores por defecto al disco.

Por último, vamos a describir el diseño de las llamadas del sistema de ficheros compuesto por las siguientes funciones:

- Función mkFS: genera la estructura del sistema de ficheros diseñada en el dispositivo de almacenamiento. Para ello, primero comprueba que el tamaño del disco en bytes dado como argumento este entre 50 KiB y 10 MiB (requisito **NF9**). A continuación, se inicializa el numMagico, el numInodos, el numBloquesDatos, tamDispositivo, primerBloqueDatos y el totalElementos, que serán utilizados por todas las llamadas del sistema de ficheros.

Luego se inicializan el mapas de i-nodos (i_map[numInodos]) y el mapa de bloques (b_map[numBloquesDatos]), de manera que tenemos un array de i-nodos libre y un array de bloques libres. Posteriormente, inicializaremos el contenido de los i-nodos a -1, para así no tener problemas con el contenido de los i-nodos y el identificador a 0. Tras ello, con la función syncAux, escribimos los valores por defecto al disco. Esta función cumple con el requisito **F1.1** y con el requisito **F6** (el cual nos pide que el sistema de ficheros puede ser creado en particiones del dispositivo más pequeñas que su tamaño máximo).

- Función mountFS: monta el dispositivo simulado *disk.dat*, de manera que esta función se encarga de asignar y configurar todas las estructuras y variables necesarias utilizadas en el sistema de ficheros. Así pues, lo primero que hace es leer el bloque 1 de disco en sbloques[0] teniendo en cuenta el posible fallo. Luego, lee los bloques para el mapa de i-nodos y lee los bloques para el mapa de bloques de datos. Por último, lee los i-nodos a memoria montando la información en el dispositivo simulado. Si cualquiera de los bread ejecutados anteriormente devolviera -1, significa que no se monta en el *disk.dat*; pero si devuelve 0, la ejecución ha sido correcta. Esta función cumple con el requisito **F1.2** y valida el requisito **NF8**, en el que se nos pedía que los metadatos del sistema debieran persistir entre operaciones de desmontaje y montaje.
- Función unmountFS: desmonta el dispositivo simulado *disk.dat*, de manera que libera todas las estructuras y variables utilizadas por el sistema de ficheros. Simplemente, se asegura de que todos los ficheros están correctamente cerrados, y escribe a disco los metadatos utilizando la función syncAux. Esta función cumple con el requisito **F1.3** y valida el requisito **NF8**, en el que se nos pedía que los metadatos del sistema debieran persistir entre operaciones de desmontaje y montaje.
- Función createFile: realiza todos los cambios necesarios en el sistema de ficheros para crear un nuevo fichero vacío. Primero comprueba si no ha sobrepasado el número límite de ficheros que se pueden crear (40), y a continuación comprueba a partir de la ruta de entrada que solo tiene 3 niveles (de esta forma se cumple con el requisito **NF5** en el que se nos pedía que la profundidad máxima de una jerarquía de directorios no superará 3 niveles). Dentro de esta comprobación, también se verifica el requisito **NF3** en el que se exigía que el nombre de un fichero o directorio sólo podrá tener como máximo una longitud de 32 caracteres. También se comprobarán a continuación que los directorios existen, que están dentro del nodo raíz correspondiente (dentro de su padre dado en la ruta de forma correcta) y verifica que no se pueda crear un fichero dentro de otro fichero. Además se verificará en esta función el requisito **F7** y **F8**, donde se nos pedía utilizar el caracter '/' como separador en un ruta que no podrá aparecer en el nombre de ningún fichero ni directorio, y se considerará la ruta del directorio raíz como '/'. Después de todo esto, se comprobará que no está repetido con la función namei, y si todo va bien, ejecuta un ialloc y un alloc, para comprobar el identificador. Tras ello, realiza los cambios en el diseño correspondientes con la creación y se asume que está cerrado al crearlo, al igual que se asume que su puntero es 0. Por último, se cumple con el requisito **F1.4**.
- Función removeFile: realiza todos los cambios necesarios en el sistema de ficheros para eliminar un fichero. Primero comprueba que el totalElementos no sea menor o igual que 0, ya que supuestamente se ha creado el fichero. Luego comprueba los requisitos **NF5** y **NF3** de la misma forma que la función createFile, y tras ello comprueba con la función namei si el fichero no existe en el sistema de ficheros (a lo cual devuelve un -1). Luego libera el fichero y resta uno al totalElementos. Se cumple pues el requisito **F1.5**.
- Función openFile: abre un fichero existente en el sistema de ficheros e inicia su puntero de puntero de posición al principio del fichero (cumple pues con el requisito **F2** que nos pide reiniciar al principio el puntero de posición cada vez que un fichero es abierto). En

primer lugar comprueba los requisitos **NF5** y **NF3** (al igual que la función removeFile) y tras ello busca el i-nodo asociado al nombre puesto en el path como argumento con la función namei. Por último comprueba que el fichero no pueda ser abierto porque no va a existir en el sistema de ficheros, y controlamos que no esté abierto (ya que daría fallo la función) y si todo va bien, devuelve 0. Se cumple pues el requisito **F1.6**.

- Función closeFile: cierra el fichero abierto. Comprueba que el descriptor introducido es válido y que está abierto (que en cuyo caso lo cerramos y el puntero lo devolvemos a 0 tal y como se indica en el enunciado de la práctica). Si el fichero no existe o está cerrado devuelve un error -1. Se cumple pues el requisito **F1.7**.
- Función readFile: lee tantos bytes como se indiquen de un fichero representado por su descriptor, comenzando desde su puntero de posición y alojando los bytes leídos en un buffer (declarado en el fichero *test.c*) proporcionado. Además, esta función incrementa el puntero de posición del fichero tantos bytes como se hayan leído correctamente. Así pues comienza comprobando que al aplicar numBytes no se exceda de los límites de su tamaño máximo o sea 0. Luego se tiene en cuenta lo siguiente: cuando el número total de bytes que se pueden leer del fichero es menor que el parámetro numBytes, readFile devuelve el número de bytes realmente leídos; cuando el puntero de posición del fichero está al final de este (no hay más bytes para leer) o al principio (no hay nada escrito en el fichero indicado por el fileDescriptor) devuelve un 0; y en caso de fallo devuelve un -1. Esta función cumple con los requisitos **F1.8** y **F4** (donde se nos pide que el sistema pueda leer el contenido completo de un fichero utilizando varias operaciones).
- Función writeFile: modifica tantos bytes como se indiquen de un fichero representado por su descriptor, comenzando desde su puntero de posición y escribiendo en el fichero el contenido de un buffer proporcionado. Además, incrementa el puntero de posición del fichero tras la escritura tantos bytes como se hayan escrito. Su funcionamiento es muy parecido al de readFile, al igual que las comprobaciones. Esta función cumple con los requisitos **F1.9**, **F3** (donde se nos pedía que los metadatos del sistema se actualizasen después de cada operación de escritura para que reflejen adecuadamente los cambios realizados) y **F5** (donde se nos pedía poder modificar parte del contenido de un fichero utilizando operaciones write).
- Función lseekFile: modifica el valor del puntero de posición de un fichero dependiendo de las constantes de referencia whence (que puede ser o FS_SEEK_CUR o FS_SEEK_BE GIN o FS_SEEK_END) y del número de bytes a desplazar el puntero de posición actual del fichero. Cumple con el requisito **F.10**.
- Función mkDir: realiza todos los cambios necesarios en el sistema de ficheros para crear un nuevo directorio vacío. Realiza las mismas comprobaciones iniciales que la función createFile, y luego comprueba que no está repetido el directorio y luego ejecuta todos los cambios necesarios para crear el nuevo directorio. Cumple con el requisito **F.11**.
- Función rmDir: realiza todos los cambios necesarios en el sistema de ficheros para borrar un directorio y su contenido. El código es bastante similar al de la función mkDir. Cumple con el requisito **F.12**.
- Función lsDir: carga los i-nodos y nombres de los elementos de un directorio. Cumple con el requisito **F.13**.

3. BATERÍA DE PRUEBAS

En este apartado se van a describir todas las baterías de pruebas utilizadas y todos los resultados obtenidos.

3.1. FUNCIÓN MKFS

Objetivo	Procedimiento	Entrada	Salida esperada
Comprobar que el programa intenta crear un sistema de ficheros que exceda los límites de capacidad de almacenamiento del dispositivo (requisito NF9 - el sistema de ficheros será usado en discos de 50 KiB a 10 MiB)	Ponemos un device_size menor de 50 KB en el fichero test.c	mkfs(15000)	return -1 Resultado de la prueba es *FAILED.* *Prueba superada*
Comprobar que el programa intenta crear un sistema de ficheros que exceda los límites de capacidad de almacenamiento del dispositivo (requisito NF9 - el sistema de ficheros será usado en discos de 50 KiB a 10 MiB)	Ponemos un device_size mayor de 10 MB en el fichero test.c	mkfs(9000000000)	return -1 Resultado de la prueba es *FAILED.* *Prueba superada*
Comprobar que el programa genera la estructura del sistema de ficheros diseñada en el dispositivo de almacenamiento	Introducimos un device_size dentro del rango y llamamos a la función mkfs()	mkfs(60000)	return 0 Resultado de la prueba es *SUCCESS* Se genera la estructura del sistema de ficheros Prueba superada

3.2. FUNCIÓN MOUNTFS

Objetivo	Procedimiento	Entrada	Salida esperada
Comprobar que se carga el sistema de archivos almacenado en el disco	Llamar a la función mountfs() después de realizar la operación mkfs() que crea el sistema.	mountfs()	Realizamos un print para comprobar que se cargaban los ficheros. SUCCESS Prueba superada
Comprobar que no se carga el sistema de archivos si antes no se ha creado	Llamar a la función mountfs() antes de realizar la operación mkfs() que crea el sistema.	mountfs()	FAILED Prueba superada

3.3. FUNCIÓN UNMOUNTFS

Objetivo	Procedimiento	Entrada	Salida esperada
Comprobar que el sistema de archivos se almacena en el disco	Llamar a la función unmountfs() después de realizar la operación mkfs() que crea el sistema. Después del unmount hacemos un mount para comprobar que se guarda correctamente.	unmountfs()	Realizamos un print para comprobar que se cargaban los ficheros. SUCCESS Prueba superada

3.4. FUNCIÓN CREATEFILE

Objetivo	Procedimiento	Entrada	Salida esperada
Crear un fichero en el directorio raíz	Llamamos a la función creatFile después de crear el sistema de ficheros y le pasamos por parámetros la ruta de un fichero	createFile(fichero.txt)	Hacemos un print para comprobar que se crea el archivo. Success Prueba superada
Crear un fichero en un directorio	Llamamos a la función creatFile después de crear el sistema de ficheros y le pasamos por parámetros la ruta de un fichero	createFile(home/Documentos/fichero.txt)	Hacemos un print para comprobar que se crea el archivo. Success Prueba superada
Crear un fichero en un directorio que no existe	Llamamos a la función creatFile después de crear el sistema de ficheros y le pasamos por parámetros la ruta de un fichero con un directorio no existente	createFile(home/fichero.txt)	"ERROR: No se puede crear el fichero" FAILED Prueba superada
Crear un fichero con un nombre mayor de 32	Llamamos a la función creatFile después de crear el sistema de ficheros y le pasamos por parámetros la ruta de un fichero con un nombre mayor de 32 bytes	createFile(ficherosficherosficherosficherosficheros.txt)	"ERROR: El fichero tiene más de 32 bytes" FAILED Prueba superada
Crear un fichero dentro de un fichero	Llamamos a la función creatFile después de crear el sistema de ficheros y le pasamos por parámetros la ruta	createFile(fichero.txt/fichero1.txt)	ERROR FAILED Prueba superada

	de un fichero para crearse dentro de otro fichero		
Fichero ya existente	Llamamos a la función creatFile después de crear un fichero con el mismo nombre	createFile(fichero.txt)	ERROR: El fichero ya existe FAILED Prueba superada

3.5. FUNCIÓN REMOVEFILE

Objetivo	Procedimiento	Entrada	Salida esperada
Borrar fichero con el sistema vacío	Llamamos a la función removeFile() sin ningún fichero en el sistema	removeFile (Fichero.txt)	"No hay elementos" FAILED Prueba superada
Borrar fichero	Llamamos a la función removeFile() pasándole la ruta de un fichero existente	removeFile (Fichero.txt)	SUCCESS Prueba superada
Borrar fichero inexiste	Llamamos a la función removeFile() pasándole la ruta de un fichero existente	removeFile (Fichero1.txt)	ERROR "El fichero no existe" FAILED Prueba superada

3.6. FUNCIÓN OPENFILE

Objetivo	Procedimiento	Entrada	Salida esperada
Abrir un fichero existente	Llamamos a la función openFile() pasándole un fichero que existe en el sistema	openFile(fichero.txt)	SUCCESS Prueba superada
Abrir un fichero inexistente	Llamamos a la función openFile() pasándole un fichero que no existe en el sistema	openFile(fichero1.txt)	ERROR FAILED Prueba superada

3.7. FUNCIÓN CLOSEFILE

Objetivo	Procedimiento	Entrada	Salida esperada
Cerrar un fichero existente	Llamamos a la función closeFile() pasándole un descriptor de un	closeFile(0)	SUCCESS Prueba superada

	fichero abierto previamente		
Cerrar un fichero inexistente	Llamamos a la función closeFile() pasándole un descriptor fichero que no existe en el sistema	closeFile(8)	ERROR: "No existe el fichero o está cerrado" FAILED Prueba superada

3.8. FUNCIÓN READFILE

Objetivo	Procedimiento	Entrada	Salida esperada
Leer un fichero previamente creado, abierto y escrito	Llamar a la función readFile() sobre un fichero abierto del sistema pasándole por parámetros el descriptor del fichero, el buffer donde queremos que lea y el número de bytes que queremos leer	readFile(0, buffer, 10)	SUCCESS Prueba superada
Leer un fichero cerrado	Llamar a la función readFile() sobre un fichero cerrado del sistema pasándole por parámetros el descriptor del fichero, el buffer donde queremos que lea y el número de bytes que queremos leer	readFile(0, buffer, 10)	ERROR: "El fichero está cerrado" FAILED Prueba superada

3.9. FUNCIÓN WRITEFILE

Objetivo	Procedimiento	Entrada	Salida esperada
Escribir un fichero previamente creado y abierto	Llamar a la función writeFile() sobre un fichero abierto del sistema pasándole por parámetros el descriptor del fichero, el buffer que queremos escribir y número de bytes que queremos escribir	writeFile(0, "Hola mundo", 10)	SUCCESS Prueba superada

Escribir un fichero previamente creado y cerrado	Llamar a la función writeFile() sobre un fichero cerrado del sistema pasándole por parámetros el descriptor del fichero, el buffer donde queremos que lea y el número de bytes que queremos leer	writeFile(0, buffer, 10)	ERROR: "El fichero está cerrado" FAILED Prueba superada

3.10. FUNCIÓN LSEEKFILE

Objetivo	Procedimiento	Entrada	Salida esperada
Mover el puntero de un fichero creado y abierto a la posición indicada por el whence y el offset	Llamar a la función lseekFile() sobre un fichero abierto del sistema pasándole por parámetros el descriptor del fichero, el whence y los bytes de desplazamiento. Comprobamos el resultado imprimiendo la posición del puntero	lseekFile(0, FS_SEEK _BEGIN , 3)	SUCCESS Imprima 3 Prueba superada
Mover el puntero de un fichero creado y cerrado a la posición indicada por el whence y el offset	Llamar a la función lseekFile() sobre un fichero cerrado del sistema pasándole por parámetros el descriptor del fichero, el whence y los bytes de desplazamiento. Comprobamos el resultado imprimiendo la posición del puntero	lseekFile(0, FS_SEEK _BEGIN , 3)	ERROR: El fichero está cerrado. FAILED Prueba superada

3.11. FUNCIÓN MKDIR

Objetivo	Procedimiento	Entrada	Salida esperada
Crear un directorio en el sistema de ficheros dentro de un directorio existente	Llamar a la función mkdir() indicando la ruta del directorio que deseamos crear.	mkdir (home/Documentos)	SUCCESS Prueba superada

	Comprobamos el resultado imprimiendo los inodos		
Crear un directorio ya existente en el sistema de ficheros	Llamar a la función mkdir() indicando la ruta del directorio que deseamos crear. Comprobamos el resultado imprimiendo los inodos	mkdir (home)	ERROR: "El directorio ya existe" FAILED Prueba superada
Crear un directorio dentro de un directorio que no existe	Llamar a la función mkdir() indicando la ruta del directorio que deseamos crear. Comprobamos el resultado imprimiendo los inodos	mkdir (pepe/Documentos)	ERROR: "No se puede crear el directorio" FAILED Prueba superada
Crear un directorio dentro de un fichero que existe	Llamar a la función mkdir() indicando la ruta del directorio que deseamos crear. Comprobamos el resultado imprimiendo los inodos	mkdir (fichero.txt/Programa)	ERROR: "No se puede crear un directorio dentro de un fichero" FAILED Prueba superada
Crear un directorio con más de 3 niveles	Llamar a la función mkdir() indicando la ruta del directorio que deseamos crear. Comprobamos el resultado imprimiendo los inodos	mkdir (home/Documentos/Programas/Pepe)	ERROR: "Máximo 3 niveles" FAILED Prueba superada

3.12. FUNCIÓN RMDIR

Objetivo	Procedimiento	Entrada	Salida esperada
Borrar un directorio existente y todos los elementos en su interior	Llamar a la función rmDir() indicando la ruta del directorio que deseamos borrar. Comprobamos el resultado imprimiendo los inodos	rmDir (home/Documentos)	Debería borrarse home/Documentos y todo lo que hay dentro de ese directorio. SUCCESS Prueba no superada. Solo se borra el directorio pero no los elementos de su interior

3.13. FUNCIÓN LSDIR

Objetivo	Procedimiento	Entrada	Salida esperada
Mostrar los elementos dentro de un directorio	Llamar a la función lsDir() indicando la ruta del directorio que deseamos ver. Comprobamos el resultado imprimiendo los inodos	lsDir(home, inodesDir, namesDir)	2 Documentos SUCCESS Prueba no superada. No conseguimos mostrar los elementos dentro del directorio.

4. CONCLUSIONES Y PROBLEMAS ENCONTRADOS

Con el desarrollo de esta segunda práctica hemos comprendido con mayor claridad el funcionamiento de la arquitectura de un sistema de ficheros, así como su implementación en el lenguaje de programación C para desarrollar programas clientes teniendo en cuenta la interfaz del sistema para acceder y modificar sus ficheros.
En cuanto a los problemas encontrados, nos ha resultado una práctica bastante difícil de realizar debido a que no sabíamos cómo interpretar los resultados de la implementación de las funciones en el *disk.dat*, y tardamos mucho tiempo en comprenderlo. Pensábamos también que nuestro diseño del sistema no era correcto, así que tuvimos que cambiarlo varias veces hasta dar con el propuesto en el punto 2 (el cual funciona y cumple con los requisitos propuestos).
Las funciones rmDir y lsDir no dieron tiempo a hacerlas, sin embargo, el código compila sin errores y el código está más o menos bien.