Práctica 2: Sistema de ficheros

EPS

Ingeniería Informática

Diseño de Sistemas Operativos

Santiago Ramos Sevillano NIA 100383401 Gr. 83
100383401@alumnos.uc3m.es
Iván Valbuena Gálvez NIA 100383375 Gr. 83
100383375@alumnos.uc3m.es
Lucía Ruz Sáez NIA 100363940 Gr. 83
100363940@alumnos.uc3m.es

Santiago Ramos Sevillano NIA 100383401
Iván Valbuena Gálvez NIA 100383375
Lucía Ruz Sáez NIA 100363940

uc3m | Universidad Carlos III de Madrid

Índice

Índice de tablas

Santiago Ramos Sevillano NIA 100383401
Iván Valbuena Gálvez NIA 100383375
Lucía Ruz Sáez NIA 100363940

uc3m | Universidad Carlos III de Madrid

1 Introducción

En esta práctica se va a desarrollar un sistemas de ficheros con algunas de las funcionalidades básicas especificadas en el enunciado. Para ello, se procede a diseñar la arquitectura del sistema de ficheros, implementar su diseño en el lenguaje de programación C y crear un plan de pruebas.

En los siguientes puntos se justificarán detalladamente todas las decisiones de diseño tomadas por el equipo. En el apartado de “Plan de Validación” se expondrán todos aquellos requisitos expuestos en el enunciado frente a la prueba realizada para comprobar su correcto funcionamiento.

Por último, se ha incluido un apartado de conclusiones donde se expondrán los problemas encontrados a lo largo de la práctica, tanto en el diseño como en la implementacion del codigo, como aquellas conclusiones a las cuales se ha llegado.

2 Diseño de sistema de ficheros

A continuación se definen todas las decisiones de diseño tomadas por el equipo en los tres apartados principales de la práctica.

2.1 Diseño de organización en disco

La estructura del disco será la siguiente:

- El primer bloque viene ocupado en su totalidad por el superbloque, con todos los datos que tiene su estructura (como mencionamos en el siguiente punto). Cabe destacar que el superbloque contiene los mapas de inodos y mapas de bloques, que se usarán para mantener un control de los mismos, además del resto de información importante a la hora de interaccionar con el sistema de ficheros
- El segundo y tercer bloque corresponderá a los necesarios para los inodos. En este caso, y con los cálculos que mostramos posteriormente en la estructura de los inodos, un solo bloque no es suficiente para almacenar todos los 48 inodos, por lo que tendremos 24 en cada uno de los dos bloques
- Tras estos bloques de metadatos en la cabecera vendrán todos los bloques de datos que almacenarán la información del sistema de ficheros.

Cabe destacar que el máximo número de bloques con los que cuenta el sistema vienen determinados por el tamaño del disco / tamaño del bloque, de los cuales los tres primeros conforman la cabecera y el resto serán bloques de datos.

2.1.1 Estructuras en memoria

Se han implementado 3 estructuras de datos:

- **Tipo SuperBloque:**
  - **uint32_t** numMagico: número mágico elegido para nuestro sistema de ficheros
  - **uint8_t** numInodos: número de inodos que tiene el sistema (48 en este caso)
  - **uint8_t** primerInodo: número que corresponde al primer inodo
  - **uint8_t** numBloquesInodos: número de bloques de inodos (2 en este caso)
  - **uint8_t** numBloqueDatos: máximo total de bloques de datos
  - **uint8_t** primerBloqueDatos: número correspondiente al primer bloque de datos
  - **uint32_t** tamDispositivo: tamaño en bytes del dispositivo

- **char** mapaBloques: mapa que indica si los bloques se encuentran vacíos. Si están en uso se representa con un 1, mientras que si están libres con un 0
- **uint32_t** mapaInodos: mapa que indica si los inodos se encuentran en uso o no. 0 para aquellos que están libres, 1 para los usados.
- **uint32_t** relleno: relleno para completar el tamaño del superbloque.

Nota: hemos observado, tanto con cálculos a mano como con el uso de sizeof para los diferentes parámetros que el tamaño de la estructura (sin contar relleno) es de 301 bytes. No obstante, al imprimir el tamaño de la estructura en su conjunto con el relleno correspondiente para llegar a 2048 (que serían 1747 bytes) hemos observado que nos mostraba un tamaño de 2052 bytes, por lo que hemos tenido que reducir el tamaño del relleno a 1743.

- **TipoInodo:**
  - **uint8_t** tipo: indica si el fichero asociado es un fichero (tomará el valor de 1) o un enlace blando (tomará el valor de 0).
  - **char** nombre: indica el nombre del fichero o enlace al que corresponde.
  - **char** nombreEnlace: en caso de ser tipo enlace almacena el nombre del fichero al que apunta.
  - **uint16_t** tamano: indica el tamaño actual del fichero asociado en bytes.
  - **uint8_t** bloqueDirecto[]: Array que guarda los bloques que componen el fichero asociado.
  - **uint32_t** integridad: número resultado de sumar las integridades de cada bloque del fichero asociado, indica la integridad del fichero.

Nota: al igual que ocurre con el superbloque obtenemos un tamaño ligeramente superior para cada estructura de tipo inodo, comparado con los cálculos realizados manualmente. En este caso obtenemos que cada inodo debería ocupar 76 bytes, pero haciendo un sizeof de un inodo ocupa 80. Aún así, como queríamos meter 24 inodos por bloque y 80 * 24 = 1920 que es menor que el tamaño del bloque, seguimos teniendo esa misma estructura.

Además de las dos estructuras presentadas, las cuáles se irán volcando y leyendo de disco, hemos utilizado otra estructura y una variable adicional, que no son escritas en disco pero nos permiten un control de los ficheros abiertos y cerrados y sus respectivos punteros y del montaje y desmontaje del dispositivo. Estas son:

- **TipoInodo_x:**
  - **uint16_t** posicion: indica la posición del puntero para lectura o escritura.
  - **uint8_t** abierto: indica si el fichero asociado está abierto (tomará el valor de 1) o no (tomará el valor de 0).
- int montar = 0 : variable que permite controlar si el dispositivo está montado o no.

Podemos observar que se han utilizado tipos diferentes realizando cálculos de tamaño máximo según el atributo, así, hemos establecido el menor número de bytes posibles para cada uno de ellos minimizando el espacio ocupado.

2.1.2 Diseño de rutinas de gestión

- **ialloc (void):** Función encargada de asignar un inodo como ocupado en el mapa de inodos o indicar cuando no queden inodos disponibles
  - Recorremos todos los inodos:
    - Si encontramos un inodo libre:
      - Cambiamos estado del inodo en el mapa de inodos a ocupado, le ponemos los valores a 0 al inodo y devolvemos el identificador del inodo

- Si no hemos encontrado un inodo libre, retornamos -1.

- **alloc (void):** Función encargada de asignar un bloque como ocupado en el mapa de bloques o indicar cuando no queden bloques disponibles
  - Recorremos todos los bloques de datos:
    - Si encontramos uno libre: cambiamos el estado del bloque en el mapa de bloques a ocupado, le ponemos los valores a al bloque de datos, bwrite para escribir a dispositivo y retornamos el identificador del bloque.
  - En caso de no encontrar un bloque libre, devolvemos -1.
- **ifree (int inodo_id:** Función encargada de liberar un inodo.
  - Si el identificador del inodo no es válido, retornamos -1
  - Si es válido, liberamos el inodo.
  - retornamos 0
- **bfree (int block_id:** Función encargada de liberar un bloque
  - Si el identificador del inodo no es válido, retornamos -1
  - Si es válido, liberamos el bloque.
  - retornamos 0
- **bmap (int inodo_id, int offset**: Función encargada de devolver el bloque de datos asociado al id y offset recibidos
  - Calculamos el índice del bloque en que estamos dentro del array “bloqueDirecto[ ]” de ese fichero.
  - Si es mayor a 4, devolvemos -1 ya que el tamaño máximo de un fichero es de 5 bloques (0, 1, 2, 3 y 4).
  - Devolvemos el bloque de datos correspondiente a ese bloque directo.
- **namei (char *fname):** Función encargada de comprobar que existe un inodo en el sistema con el nombre pasado por parámetro
  - Recorremos todos los inodos:
    - Si encontramos un fichero cuyo nombre coincida con fname:
      - Devolvemos el identificador del fichero
  - Retornamos -1.
- **metadata_fromDiskToMemory (void):** encargada de transferir los metadatos de disco a memoria.
  - Lee la información del superbloque y la almacena en un buffer
  - Actualiza la información en memoria del superbloque con lo leído del disco
  - Para cada bloque de inodos (2 en total)
    - Lee los 24 inodos del bloque correspondiente del disco
    - Actualiza la información en memoria de esos inodos
  - Devolvemos 0 para indicar que todo ha ido bien.
- **metadata_fromMemoryToDisk (void):** encargada de transferir los metadatos de memoria a disco.
  - Ponemos a 0 todos los valores de un buffer
  - Modifica el buffer con la información del superbloque
  - Escribe el superbloque a disco
  - Para el número de bloques de inodos (2 en total): vuelve a poner el buffer a 0, lo modifica con los 24 inodos del bloque en el que esté y escribe en el disco los inodos en su posición correspondiente
  - Retornamos 0 para indicar que todo ha ido bien.

2.1.3 Decisiones de diseño en cuanto a la funcionalidad

En este apartado vamos a detallar las decisiones de diseño que hemos llevado a cabo para la resolución de la práctica en cuanto a la funcionalidad de la misma.

- El número de bloques dependerá del tamaño del dispositivo, siendo los tres primeros para metadatos y el resto para datos. El número máximo de bloques de datos será de 240 (puesto que es lo necesario para los 48 ficheros máximos). En caso de no poder contar con 240 bloques de datos el array del mapa de bloques no hará uso de las últimas posiciones, pues seguirá teniendo 240 posiciones.
- La información en memoria se volcará al disco en el **mkFS**, **unmountFS** y en las funciones de **close** y **remove** tanto con como sin integridad.
- No se podrá realizar ninguna función aparte de **mkFS** sin montar el dispositivo antes.
- No se podrá desmontar el dispositivo si hay algún fichero abierto
- Al crear un nuevo fichero solamente se asignará su primer bloque directo, el resto tendrán un valor fuera del rango (255) y serán asignados cuando se les dé uso
- Al eliminar un fichero o enlace se borrará su información e inodos asociados además de liberar sus inodos y bloques
- Los ficheros con integridad podrán ser abiertos y cerrados exclusivamente con las funciones de integridad, evitando tener un fichero con integridad y no hacer uso de las comprobaciones pertinentes para comprobar si está corrupto por hacer un **openFile** sin integridad. Por otro lado los ficheros sin integridad sólo podrán ser abiertos y cerrados con las funciones sin integridad.
- Si se realiza un **openFile** de un enlace se abrirá su fichero asociado. Si tiene asociado otro enlace se devuelve un error.
- No se podrán abrir enlaces con la función de **openFileIntegrity**. Además, no se podrá realizar un **close** de ninguno de los tipos de un enlace.
- Solo se podrá leer o escribir en ficheros abiertos.
- La integridad irá por fichero y se calculará como la suma de las integridades obtenida de aplicar CRC a cada uno de sus bloques
- Haremos uso de CRC32 para cálculos de integridad
- La operación de abrir con integridad comprobará la integridad del fichero y la de cerrar actualizará el valor
- No se podrán crear enlaces a enlaces para evitar bucles
- Hemos decidido no poner padding en los inodos porque tenemos control de cuánto ocupan en cada uno de los bloques y queríamos evitar el mismo error que nos pasó en el superbloque con el tamaño.
- Se podrán crear enlaces a ficheros con ellos abiertos o cerrados.
- Se podrá eliminar un fichero o enlace con las funciones de **removeFile** y **removeLn** estén abiertos o cerrados.
- Al cerrar un fichero no se actualiza su puntero de posición, será al abrirlo, tal y como se especifica en el enunciado, cuando se ponga al principio.

2.2 Funcionalidad básica

2.2.1 Gestión del dispositivo

- **mkFS (long deviceSize):** Esta es la primera función a ejecutar y será la encargada de inicializar las diferentes estructuras que hemos definido. En primera instancia, solo una parte del superbloque será lo único que contendrá información distinta de 0, puesto que los mapas de inodos, mapas de bloques y los diferentes bloques de inodos y bloques de datos empiezan a 0.
  - Si deviceSize es menor que el mínimo devuelve -1
  - Si deviceSize es mayor que el máximo devuelve -1
  - Asigna todos los valores a la estructura del superbloque

- Para el número de inodos: mapaInodos[i] = 0
- Para el número de bloques de datos: mapaBloques[i] = 0
- Para el número de inodos: pone a 0 la estructura de cada inodo
- Llamada a **metadata_fromMemoryToDisk (void)**
- Para el número de bloques de datos: pone los bloques de datos a 0
- Devuelve 0

- **mountFS (void):** Tras realizar un mkFS se debe ejecutar esta función, puesto que será la encargada de montar el dispositivo **disk.dat** para poder trabajar con él.
  - Si ya está montado devuelve -1
  - Llamada a **metadata_fromDiskToMemory (void)**
  - Si el número mágico no coincide con el marcado en la estructura devuelve -1
  - Marca el dispositivo como montado (montar = 1)
  - Devuelve 0
- **umountFS (void):** Al contrario que la función **mountFS**, la función **unmountFS** es la encargada de desmontar el dispositivo y liberar las estructuras y variables utilizadas. Además, no será posible desmontar el dispositivo si hay algún fichero abierto por lo que devolverá un error.
  - Si no está montado devuelve -1
  - Para el número de inodos: Si hay alguno abierto devuelve -1
  - Llamada a **metadata_fromMemoryToDisk (void)**
  - Marca el dispositivo como desmontado (montar = 0)
  - Devuelve 0

2.2.2 Gestión de ficheros

- **createFile (char *nombre):** Esta función se encarga de realizar las comprobaciones y hacer todas las modificaciones que permitan crear un nuevo fichero en el sistema.
  - Si la longitud del nombre no está entre 1 y 32 devuelve -1
  - Si no está montado devuelve -1
  - Llamada a **namei** y, si ya existe un fichero con ese nombre devuelve -1
  - Llamada a **ialloc** para obtener un inodo, si no quedan inodos devuelve -1
  - Llamada a **alloc** para obtener un bloque libre
    - Si no hay bloques libres libera el inodo obtenido y devuelve -1
  - Asigna los valores necesarios a la estructura del inodo

Observaciones: podemos comprobar que en esta función solamente se asigna el primer bloque directo del inodo, el resto los pone a un valor no válido (en este caso 255 que es mayor que el máximo de bloques de datos (240)). El resto de bloques directos irán siendo asignados según se necesite

- **removeFile (char *nombre):** Esta función se encarga de realizar las comprobaciones y hacer todas las modificaciones que permitan borrar un fichero en el sistema.
  - Si no está montado devuelve -1
  - Llamada a **namei** y, si no existe un fichero con ese nombre devuelve -1
  - Para el número de bloques directos del fichero (5)
    - Llamada a **bfree** para liberar los bloques directos
  - Para el número de bloques directos del fichero (5)
    - Pone a 0 los bloques de datos asociados al fichero
  - Pone a 0 la información del inodo asociado al fichero
  - Libera el inodo con una llamada **ifree**
  - Llamada a **metadata_fromMemoryToDisk (void)**

- Devuelve 0

- **openFile (char *nombre):** Esta función será la encargada de abrir un fichero existente en el sistema de ficheros y de posicionar el puntero de lectura o escritura al inicio del mismo.
  - Si no está montado devuelve -2
  - Llamada a **namei** y, si no existe un fichero con ese nombre devuelve -1
  - Si ya está abierto devuelve -2
  - Si tiene integridad devuelve -2 (se debe abrir con integridad)
  - Si el inodo es de tipo enlace: llama a namei para obtener el inodo del objeto al que apunta, si es de tipo enlace o no existe devuelve -2 y llama a openFile con el nombre del fichero asociado.
  - Actualiza la posición del puntero y marca como abierto
  - Devuelve el descriptor del fichero abierto
- **closeFile (int descriptor):** encargada de cerrar un fichero abierto
  - Si no está montado devuelve -1
  - Si el descriptor está fuera del número de inodos (< 0 o > 48) devuelve -1
  - Si no está abierto devuelve -1
  - Si tiene integridad devuelve -1 (se debe cerrar con integridad)
  - Si es de tipo enlace devuelve -1
  - Marca como cerrado y llama a **metadata_fromMemoryToDisk (void)**
  - Devuelve 0

2.2.3 Interacción con ficheros

- **readFile (int descriptor, char *buffer, int size):** Esta es la función encargada de leer tanta información de un fichero como nos pidan y almacenarla en un buffer
  - Si no está montado devuelve -1
  - Si el descriptor está fuera del número de inodos (< 0 o > 48) devuelve -1
  - Si no está abierto devuelve -1
  - Si es de tipo enlace devuelve -1
  - Si la posición del puntero más el tamaño a leer excede el tamaño actual del fichero: el nuevo tamaño a leer es la resta entre el final de fichero y la posición del puntero
  - Si el tamaño a leer es 0 devuelve 0
  - Mientras que la posición del puntero más el tamaño a leer sea mayor que lo que queda por leer en el bloque en el que está el puntero: obtenemos el bloque actual y lo leemos de disco, guardamos la información en el buffer proporcionado y actualiza todo lo necesario para seguir con el bucle y devolver el resultado al final
  - Mismos pasos que el bucle pero cuando no entra en el mismo
  - Devuelve la cantidad de bytes leídos
- **writeFile (int descriptor, char *buffer, int size)**: Esta función es la encargada de escribir en un fichero la información almacenada en un buffer.
  - Si no está montado devuelve -1
  - Si el descriptor está fuera del número de inodos (< 0 o > 48) devuelve -1
  - Si no está abierto devuelve -1
  - Si es de tipo enlace devuelve -1
  - Si la posición del puntero más el tamaño a leer excede el tamaño máximo del fichero: el nuevo tamaño a escribir es la resta entre el tamaño máximo del fichero y la posición del puntero
  - Si el tamaño a escribir es 0 devuelve 0
  - Mientras que la posición del puntero más el tamaño a escribir sea mayor que el hueco que queda para escribir en el bloque en el que está el puntero

        - Obtenemos el bloque actual y se lo asignamos al inodo como el siguiente bloque directo (solo si no está asignado, es decir, vale 255)
        - Leemos el bloque de disco a un buffer auxiliar
        - Modificamos el buffer auxiliar con la nueva información a escribir
        - Escribe a disco el bloque modificado
        - Actualiza todo lo necesario para seguir con el bucle y devolver el resultado al final
    - Mismos pasos que el bucle pero cuando no entra en el mismo
    - Devuelve la cantidad de bytes escritos

- **lseekFile (int descriptor, long offset, int whence):** Esta función cambia valor del puntero de posición de un fichero.
    - Comprobamos que el dispositivo está montado, que el "descriptor" es válido y que es de un fichero y no es de un enlace: si no se cumple alguna de estas condiciones, devolvemos -1 indicando un error
    - Si "whence" es FS_SEEK_BEGIN:
        - Ajustamos el puntero de posición al inicio y retornamos 0
    - Si "whence" es FS_SEEK_END:
        - Ajustamos el puntero de posición al final del fichero y retornamos 0
    - Si "whence" es FS_SEEK_CUR:
        - Comprobamos que la posición a la que queremos llegar es válida, en caso contrario devolvemos -1
        - Si offset <= 0 y acaba en una posición válida: vamos hacia atrás tantos bytes como diga offset y retornamos 0 para indicar que todo ha ido bien
        - Si offset > 0: vamos hacia adelante tantos bytes como diga offset y retornamos 0 para indicar que todo ha ido bien
    - Retornamos -1 ya que si llegamos hasta aquí es porque ha habido un error.

2.3 Implementación de integridad

- **checkFile (char * fileName):** Esta función comprueba la integridad de un fichero que se le pasa por parámetro.
    - Comprobamos que existe un fichero cuyo nombre sea fileName, que sea un fichero y no un enlace (ya que los enlaces no tienen integridad) y que el dispositivo está montado
    - Si no se cumple alguna de las condiciones: retornamos -2
    - Nos guardamos la integridad del fichero contenida en el atributo "integridad" en "hashFichParam" y creamos "hashFich" para almacenar la integridad total del fichero.
    - Para cada bloque del fichero: leemos el bloque, calculamos la integridad del bloque llamando a "CRC32", se la sumamos a "hashFich" y si se ha producido un error al llamar a "CRC32", devolvemos -2;
    - Si las integridades coinciden: devolvemos 0
    - Si no coinciden: retornamos -1
    - Devolvemos -2
- **includeIntegrity (char * fileName:** añade integridad al fichero que se la pasa por parámetro.
    - Si el dispositivo no está montado: devolvemos -2
    - Si no existe el fichero devuelve -1
    - Si el fichero ya tiene integridad o es un enlace en vez de un fichero: devolvemos -2
    - Calculamos la integridad del fichero bloque por bloque igual que se hace en la función **checkFile** solo que ahora, como hacemos uso de los 5 bloques directos del fichero y puede haber alguno sin asignar aún, en caso de que el bloque directo tenga valor 255 hace llamada a **alloc** y asigna el bloque obtenido al inodo.

- Agregamos la integridad al atributo “integridad” del fichero.
- Devuelve 0

- **openFileIntegrity (char * fileName):** Esta función se encarga de abrir aquellos ficheros que cuentan con integridad, comprobando la misma
  - Si el dispositivo no está montado devuelve -3
  - Si el fichero no existe devuelve -1
  - Si el fichero está abierto o no tiene integridad o es un enlace: devolvemos -3
  - Cogemos la integridad del fichero llamando a **checkFile**
  - Si el fichero está corrupto: devolvemos -2
  - Si ha habido algún error al comprobar la integridad: retornamos -3
  - Establecemos el puntero de lectura y escritura a 0
  - Cambiamos su estado a abierto.
  - Devolvemos el descriptor del fichero
- **closeFileIntegrity (int fileDescriptor):** Esta función se encarga de cerrar ficheros con integridad y de actualizar su valor de integridad con los nuevos datos
  - Comprobamos si el dispositivo está montado, si “fileDescriptor” es válido, si el fichero está abierto, si tiene integridad y si es un fichero y no un enlace:
    - Si no se cumple alguna de estas condiciones, retornamos -1
  - Calculamos la integridad de la misma forma que en **checkFile**
  - Guardamos la integridad del fichero en su atributo “integridad”.
  - Cambiamos su estado a “cerrado”.
  - Guardamos los cambios en disco con una llamada a **metadata_fromMemoryToDisk (void)**.
  - retornamos 0 para indicar que todo ha ido bien.

2.4 Enlaces simbólicos

- **createLn (char * fileName, char * linkName):** crea un enlace simbólico y lo asocia a un fichero.
  - Comprobamos que tanto “fileName”, como “linkName” sean válidos, si no lo son, devolvemos -1
  - Comprobamos que el sistema está montado, si no lo está, devolvemos -1
  - Obtenemos los identificadores de los inodos asociados al fichero y al enlace
  - Comprobamos que el fichero exista y el enlace no, si el fichero no existe y/o ya hay un enlace cuyo nombre en “linkName”, devolvemos -1
  - Le asignamos un inodo al nuevo enlace con **ialloc**
  - Asignamos los valores correspondientes a cada atributo de la estructura del inodo.
  - Devuelve 0
- **removeLn (char * linkName):** se encarga de borrar un enlace simbólico creado
  - Comprobamos que el dispositivo está montado y que existe un enlace cuyo nombre es “linkName”, en caso contrario, devolvemos -1
  - Ponemos todos los valores del inodo a 0
  - Liberamos el inodo
  - Guardamos las modificaciones en disco con una llamada a **metadata_fromMemoryToDisk (void)**
  - Retornamos 0 para indicar que todo ha ido bien.

3 Plan de validación

En primer lugar, se va a presentar la prueba más grande de todas las que vamos a realizar. Esta tiene como objetivo hacer una ejecución completa, utilizando todas las funciones realizadas, para posteriormente poder

hacer pruebas más específicas probando funcionamientos concretos de cada función de manera individual o en grupos pequeños. Por lo tanto, es una prueba general que comprobará gran parte de las funcionalidades pedidas. Se va a dividir en varias fases, para mayor claridad, que se desarrollarán en las filas de la siguiente tabla:

ID	Objetivo de la prueba	Prueba realizada	Salida esperada
F1	El objetivo es comprobar que la función **mkfs** se realiza correctamente	Crear un sistema de ficheros (**mkFS**) en un disco entre 460 KiB y 600 KiB	0
F2	Con esta prueba probamos tanto que el **mkFS** ha introducido bien los datos (pues realiza alguna comprobación) como que se monta el dispositivo	Montar un fichero no montado (**mountFS**)	0
F3	En esta fase se comprueban varias funciones como que se crea el fichero, se abre, se escribe y lee bien (para lo que se comparará lo escrito con lo que almacene **read** en el buffer) y que se mueve correctamente el puntero	Crear un fichero (**createFile**) con longitud en el nombre inferior a 32 caracteres y con un tamaño inferior a 10 KiB. Se llamará “prueba1”	0
		Abrir el fichero “prueba1” (**OpenFile**)	0
		Escribir desde el principio en el fichero ya abierto con (**writeFile**) la cadena 12345	5 bytes escritos
		Modificar el puntero de posición de un fichero con (**lseekFile**) con whence = FS_SEEK_BEGIN	0
		Leer con (**readFile**) todo el fichero y comprobar que se lee la cadena 12345	5 bytes leídos
F4	Esta fase trata de comprobar que el resto de posibilidades de **lseek** (mandar al final del fichero o mover según el offset) funcionan correctamente. Para ello se moverá el puntero a diferentes posiciones leyendo parte del contenido	Modificar el puntero de posición de un fichero con (**lseekFile**) con whence = FS_SEEK_CUR y offset -3	0
		Leer con (**readFile**) y comprobar que se lee la cadena 345	3 bytes leídos
		Modificar el puntero de posición de un fichero con (**lseekFile**) con whence = FS_SEEK_BEGIN	0
		Leer con (**readFile**) y comprobar que no se lee nada y se devuelve 0	0 bytes leídos

Tabla 1: Prueba general, fases 1 - 4

ID	Objetivo de la prueba	Prueba realizada	Salida esperada
F5	En esta fase se comprobará que tanto la función **write** como **read** son capaces de escribir y leer más de un bloque respectivamente, para lo que se escribirá y leerá una cadena más grande	Escribir una cadena de 5800 caracteres en el fichero para que ocupe más de un bloque, se deberá escribir después de la cadena 12345	5800 bytes escritos
		Modificar el puntero con whence = FS_SEEK_BEGIN y leer el fichero entero que debería ocupar más de un bloque y tener las dos cadenas escritas, 12345 y la de 5800 caracteres seguidas	5805 bytes leídos
		Cerrar el fichero abierto (**closeFile**)	0
F6	En esta fase se realizarán pruebas para comprobar todas las funciones de integridad.	Creamos un nuevo fichero llamado “prueba1_2”	0
		Incluir integridad al fichero recién creado (**includeIntegrity**)	0
		Comprobar la integridad del fichero existente y cerrado (**checkFile**)	0
		Abrir un fichero con integridad con (**openFileIntegrity**).	0 bytes leídos
		Cerrar con (**closeFileIntegrity**) y actualizar metadatos.	0
F7	En esta fase se comprueban todas las funcionalidades relacionadas con los enlaces simbólicos. Además, se comprobará que se puede abrir un fichero desde su enlace	Crear enlace “enlace1” apuntando al fichero “prueba1” existente (**createLn**)	0
		Abrir un fichero existente desde su enlace asociado (**OpenFile**) y guardar el descriptor devuelto en una variable a la que llamaremos “descriptor”	0 (descriptor del fichero)
		Leer los dos primeros caracteres, debería leer la cadena 12 (usaremos la variable descriptor para hacer llamar a la función **readFile**)	2 bytes leídos
		Cerrar el fichero con el descriptor que obtuvimos del openFile anterior (**closeFile**).	0
		Eliminar enlace simbólico existente (**removeLn**)	0
F8	Esta última fase comprueba que se puede eliminar un fichero sin cerrarlo, para lo que precisa de volver a abrir “prueba1”, y que el **unmountFS** funciona. Al eliminar el fichero sin cerrarlo debe marcarlo como cerrado, pues si no daría un error al desmontar el dispositivo	Abrir el fichero “prueba1” de nuevo	0 (descriptor)
		Eliminar fichero “prueba1” (**removeFile**) sin cerrarlo	0
		Desmontar un dispositivo montado (función **unmountFS**)	0

Tabla 2: Prueba general, fases 5 - 8

Como hemos podido observar, esta prueba no busca forzar los casos límites sino comprobar que todas las funciones se realizan correctamente en una ejecución normal, lo cual hace. Además, se ha comprobado con un editor hexadecimal que todas aquellas funciones que deben volcar su información al disco lo hacen. Por último, se ha comprobado con el mismo editor que las funciones mountFS y unmountFS no modifican los metadatos y que además se cumple con todos los requisitos propuesto

ID	Objetivo de la prueba	Prueba realizada	Salida esperada
P1	Esta prueba comprueba que, si el puntero de lectura/escritura está al final del fichero, no se puede escribir nada más en él.	Escribir en un fichero cuyo puntero de escritura está al final (writeFile) .	0 (no ha escrito nada)
P2	Esta prueba comprueba que no se pueden crear sistemas de ficheros cuyo tamaño sea mayor a 600 KiB o menor a 460 KiB	Crear un sistema de ficheros (**mkFS**) en un disco mayor de 600 KiB	-1
		Crear un sistema de ficheros (**mkFS**) en un disco menor de 460 KiB	-1
P3	En esta prueba se comprueba que no se pueda montar un dispositivo ya montado ni desmontar un dispositivo que no esté montado	Volver a montar el dispositivo con (**mountFS**)	-1
		Volver a desmontar el dispositivo (**unmountFS**)	-1
P4	En esta prueba se busca comprobar que se detectan los fallos posibles que se pueden dar al crear un fichero y al eliminarlo.	Crear un fichero con el dispositivo sin montar (**createFile**)	-2
		Eliminar un fichero con el dispositivo sin montar (**removeFile**) y montar un dispositivo con (**mountFS**) para los siguientes pasos)	-2
		Crear un fichero (**createFile**) con longitud en el nombre superior a 32 caracteres	-2
		Crear un fichero (**createFile**) con longitud en el nombre de 0 caracteres	-2
		Crear más de 48 ficheros en total con la función (**createFile**) para comprobar que el 49 no se crea	-2
		Crear un fichero que ya existe (**createFile**)	-1
		Eliminar fichero no existente (**removeFile**)	-1
P5	En esta prueba se busca comprobar que se detectan los fallos posibles que se pueden dar al abrir un fichero sin integridad y al cerrarlo. Además, se comprobarán los fallos que se deben dar en el funcionamiento de apertura de los ficheros a través de enlaces.	Abrir un fichero que no existe (**OpenFile**)	-1
		Abrir un fichero abierto (**OpenFile**)	-2
		Incluir integridad a un fichero (includeIntegrity) y abrirlo sin integridad (**OpenFile**)	-2
		Abrir un enlace (**OpenFile**) que apunta a un fichero eliminado.	-2

Tabla 3: Pruebas específicas, pruebas 1 - 5

ID	Objetivo de la prueba	Prueba realizada	Salida esperada
P5	Esta es la continuación de la prueba 5, por tanto el objetivo es el mismo.	Cerrar un fichero inexistente (**closeFile**)	-1
		Cerrar un fichero que ya está cerrado (**closeFile**)	-1
		Abrir fichero con (**openFileIntegrity**) y cerrar con (**closeFile**)	-1
P6	En esta prueba se busca comprobar que se detectan los fallos posibles que se pueden dar al leer un fichero y al escribir en él.	Leer fichero inexistente (**readFile**).	-1
		Leer fichero cerrado con (**readFile**).	-1
		Escribir fichero cerrado con (**writeFile**)	-1
P7	En esta prueba se busca comprobar que se detectan los fallos posibles que se pueden dar al modificar el puntero de lectura/escritura un fichero.	Modificar el puntero desde la posición actual hasta una posición fuera del fichero por debajo (posición - offset < posición inicial del fichero) (**lseekFile**).	-1
		Modificar el puntero desde la posición actual hasta una posición fuera del fichero por arriba (posición + offset > tamaño fichero) (lseekFile).	-1
		Modificar el puntero de posición de un fichero inexistente (**lseekFile**).	-1
P8	En esta prueba se busca comprobar que se detectan los fallos posibles que se pueden dar al chequear e incluir la integridad de un fichero.	Comprobar la integridad de un fichero inexistente (**checkFile**)	-2
		Comprobar la integridad de un fichero abierto (**checkFile**)	-2
		Comprobar la integridad de un fichero que no tenga integridad (**checkFile**)	-2
		Incluir integridad a un fichero inexistente (**includeIntegrity**)	-1
		Incluir integridad a un fichero que ya tiene integridad (**includeIntegrity**)	-2
P9	En esta prueba se busca comprobar que se detectan los fallos posibles que se pueden dar al abrir y cerrar con integridad un fichero.	Abrir un fichero inexistente (openFileIntegrity)	-1
		Abrir un fichero abierto (**openFileIntegrity**)	-3
		Abrir un fichero sin integridad con (**openFileIntegrity**)	-3
		Abrir un enlace (**openFileIntegrity**)	-3

Tabla 4: Pruebas específicas, pruebas 5 - 9

ID	Objetivo de la prueba	Prueba realizada	Salida esperada
P9	Esta es la continuación de la prueba 9, por tanto el objetivo es el mismo.	Cerrar fichero inexistente (**closeFileIntegrity**)	-1
		Cerrar fichero cerrado (**closeFileIntegrity**)	-1
		Abrir fichero con (**openFile**) y cerrar con (**closeFileIntegrity**)	-1
P10	En esta fase se busca comprobar que se detectan los fallos posibles que se pueden dar al crear un enlace blando y al eliminarlo.	Crear enlace simbólico con (**createLn**) con una longitud superior a 32	-2
		Crear enlace apuntando a un fichero inexistente (**createLn**)	-1
		Crear un enlace ya existente (**createLn**)	-2
		Crear un enlace a otro enlace (**createLn**)	-2
		Eliminar enlace simbólico no existente (**removeLn**)	-1

Tabla 5: Pruebas específicas, pruebas 9 - 10

Podemos observar que las pruebas se han definido de tal manera que cada una de las filas de la tabla agrupa una serie de funciones de características similares, tratando de hacer pruebas más complejas y evitando probar lo mismo varias veces. Además, existen ciertas comprobaciones, como que no se puede ejecutar la mayor parte de las funciones sin montar el dispositivo (las únicas posibles son el **mkFS** y el propio **mountFS**) que se realizan en el código para cada una de las funciones. No obstante, al ser la misma comprobación para todas ellas, no vamos a realizar una prueba aparte para cada una por lo que solo se ha realizado en algunas de ellas en las pruebas.

Nota: entre prueba y prueba es posible que se ejecuten funciones extra, como **remove**, **unmount**, **etc** para restablecer el sistema y volver a disponer de los recursos, puesto que algunas pruebas, como la de comprobar que no se pueden crear más de 48 ficheros, requieren de tener todos los inodos disponibles. Además, algunas pruebas usarán funciones no especificadas en la tabla, por poner un ejemplo, crear un fichero antes de abrirlo dos veces para probar que no se puede a la segunda, puesto que en este caso la prueba no pretende comprobar la funcionalidad de crear fichero sino de abrirlo.

4 Conclusiones

En primer lugar, comentar que la realización de la práctica nos ha servido para entender mucho mejor cómo funciona un sistema de ficheros de manera muy reducida. Además, el documento nanoFS y las diapositivas proporcionadas han sido de gran ayuda a la hora de realizar la práctica, pues es cierto que al principio estábamos un poco perdidos y no sabíamos ni por dónde empezar. Aún así, pensamos que es una práctica que está muy bien equilibrada en cuanto a contenido y tiempo.

En cuanto a los problemas encontrados, principalmente venían por no estar acostumbrados a tocar la información del disco e ir sacando y volcando información en él, lo que ha provocado que, en unos primeros intentos, quisiéramos manejar la información sin realizar ninguna de esas dos acciones y, por consiguiente, unos quebraderos de cabeza para ver qué estaba mal. Esto, unido a la dificultad de los punteros y a las decisiones de diseño que hemos tenido que ir tomando, las cuales aparecen sobre la marcha y nos llevan a tener que cambiar partes del código, han sido los principales impedimentos.

Por último, agradecer el esfuerzo realizado, puesto que con la situación actual entendemos que es difícil responder a todos los correos. A pesar de esta dificultad, no hemos encontrado mucho problema pues la mayoría de los correos se respondían con rapidez pero, de todas formas, las clases de esta forma dificultan en gran medida seguir el ritmo de las prácticas.