DISEÑO DE SISTEMAS OPERATIVOS

Sistema de Ficheros

abril 2020

Pablo Escrivá Gallardo 100348802 - 100348802@alumnos.uc3m.es

Cristian K. Gómez López - 100349207 - 100349207@alumnos.uc3m.es

ÍNDICE DE CONTENIDOS

Introducción.

Presentamos en esta memoria una solución simple a la creación de un sistema de ficheros, simulando un disco mediante un archivo de texto. Esta práctica forma parte de la segunda y última entrega práctica de la asignatura de Diseño de Sistemas Operativos. La estructura que daremos a nuestra memoria pasará por una descripción de alto nivel

Descripción de alto nivel de la funcionalidad principal.

El sistema de ficheros que planteamos está basado en los SSFF tipo *nix ya que usamos una estructuración basada en inodos y que describiremos más adelante en detalle. También es importante aclarar que nuestro sistema de ficheros está limitado a un tamaño máximo de fichero mucho más reducido de lo que soportaría por ejemplo NTFS, cierto número de ficheros soportados y otras características que hacen que sea un SF muy simplificado que se centra en la estructuración y lógica de las operaciones sobre el sistema de ficheros.

Diseño detallado del sistema de ficheros

Supuestos

En los requisitos proporcionados en el guión de la práctica no se mencionan algunos aspectos que son muy importantes a la hora de diseñar nuestro sistema de ficheros y que por lo tanto los hemos definido por nuestra cuenta como decisiones de diseño:

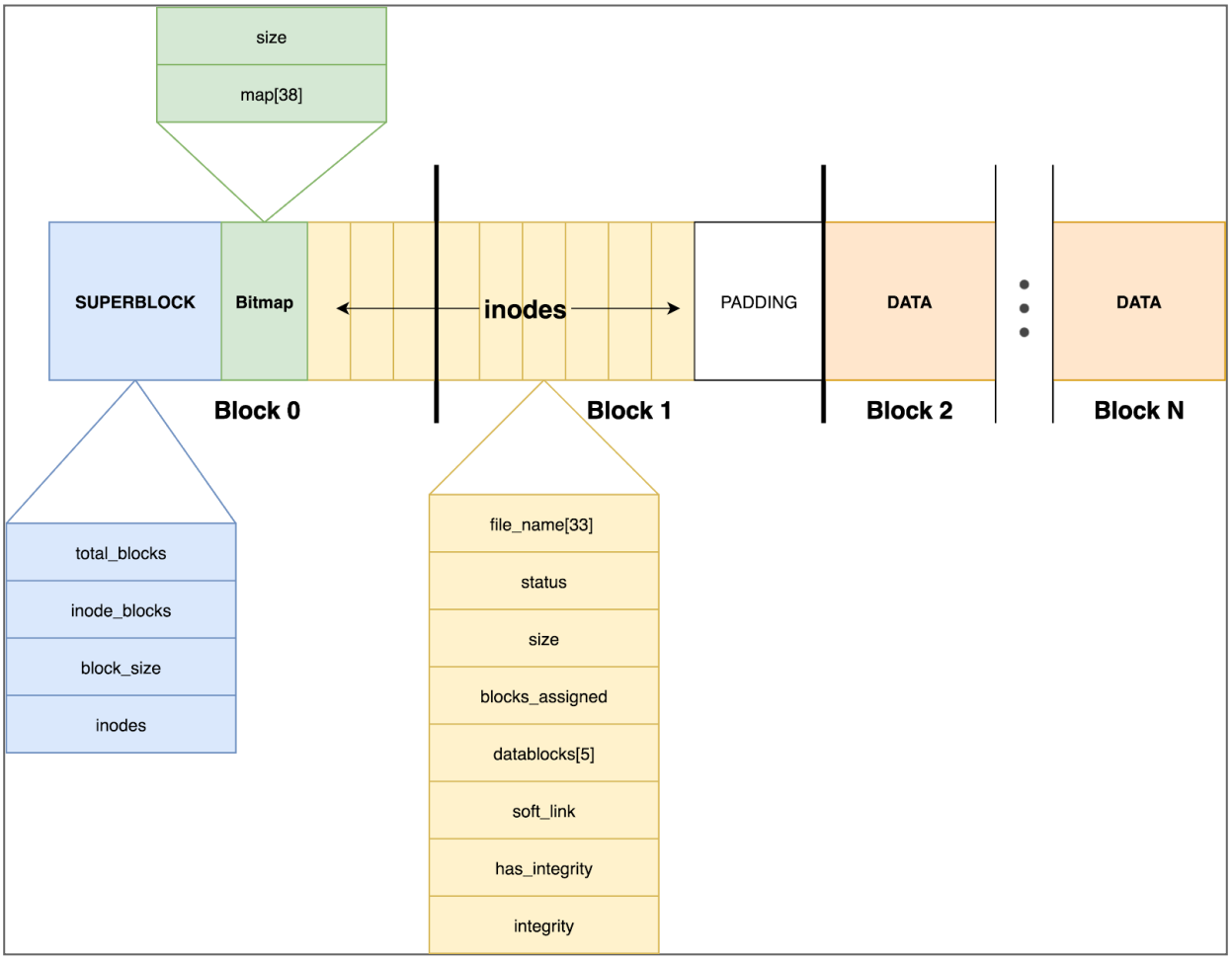
- El número máximo de ficheros abiertos en un momento determinado es 48. El motivo de elegir este número es para permitir a un usuario poder abrir el número máximo de ficheros que puede almacenar nuestro SF.
- En la función lseek no permitimos al usuario cambiar el offset de un fichero fuera de los límites del tamaño actual de éste (en sistemas de ficheros tipo UNIX/Linux se permite añadiendo caracteres nulos en los nuevos bytes que se generan).
- Cuando creamos un enlace blando a un fichero usamos un i-nodo, manteniendo su información por defecto y cambiando su variable soft_link de -1 al índice del i-nodo del fichero original. De esta manera cuando accedemos a un i-nodo que tenga un valor distinto a -1 en soft_link sabemos que se trata de un enlace blando y que el resto de variables del i-nodo no contienen información útil.
- Cuando se elimina un fichero no borramos el contenido de los bloques que ocupaba previamente, simplemente marcamos que esos bloques están libres (FREE) en el bitmap de manera que se pueda almacenar los datos de otro fichero sobreescribidolos.

Estructuras de datos

Los metadatos de nuestro sistema de ficheros se almacenan de forma persistente en los primeros dos bloques del disco.

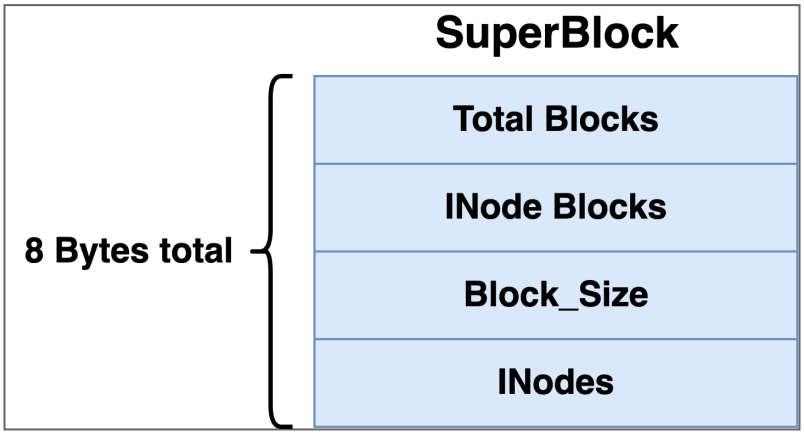
El primer bloque contiene el superbloque con la información sobre el sistema de ficheros, un mapa de bits que indica el estado de cada bloque de datos y una parte del total de i-nodos.

El segundo bloque de metadatos contiene únicamente el resto de i-nodos que no se han podido almacenar en el primer bloque. El resto de bloque tiene un padding que nos permite identificar de manera visual en que byte comienzan los bloques de datos.

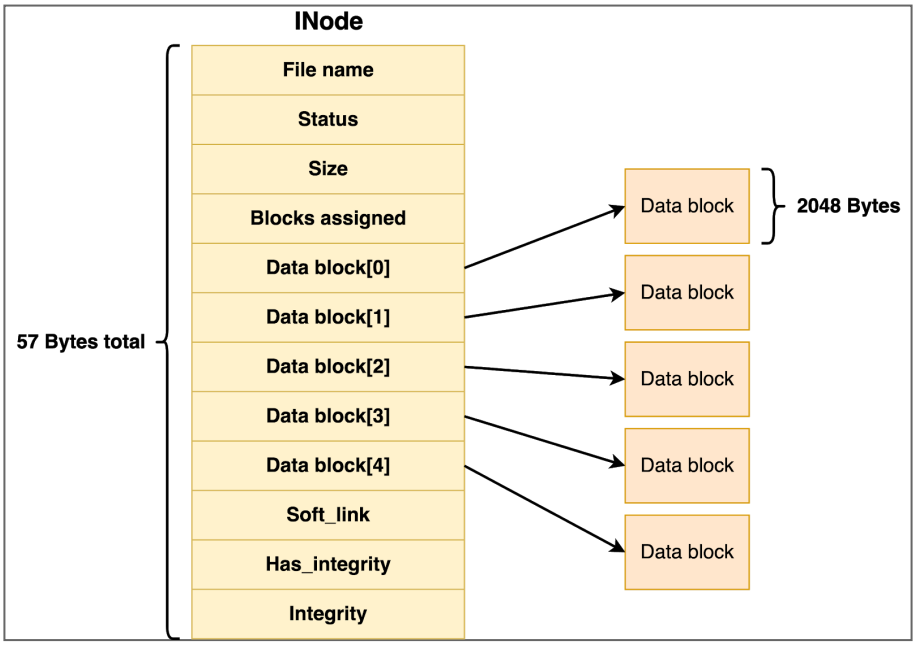


Superbloque

El superbloque contiene constantes necesarias para poder hacer uso de nuestro sistema de ficheros. De esta manera el sistema operativo puede conocer el número de bloques que contiene el volumen, conocer el tamaño de los bloques de disco y el número de i-nodos totales a los que se puede acceder.



Inodos



Los inodos constituyen la base de nuestro SSFF.

No implementar entradas de directorio nos ha llevado a tomar la siguientes decisiones:

- Los nombres de los archivos, ya sean archivos normales o enlaces simbólicos son almacenados dentro de la estructura de los propios inodos.
- La búsqueda de archivos por nombre para las distintas operaciones que así lo requieran se hará de manera secuencial sobre cada uno de los inodos en disco.
- Requerimos solamente 48 inodos siguiendo el NF2, sin necesidad de tener un directorio raíz para optimizar el espacio en disco aprovechándose de la decisión comentada antes de no implementar directorios

1. *Filename*: El NF2 especifica que el tamaño máximo de nombre de fichero es de 32 caracteres, hemos supuesto que esto es incluyendo el caracter nulo, por lo que la longitud de filename es de 32 caracteres.
2. *Status*: Nos indica si el inodo ha sido usado o no, no olvidemos que es la única forma que tenemos para determinar esto, ya que solo usaremos un mapa de bits para identificar bloques libres y ocupados

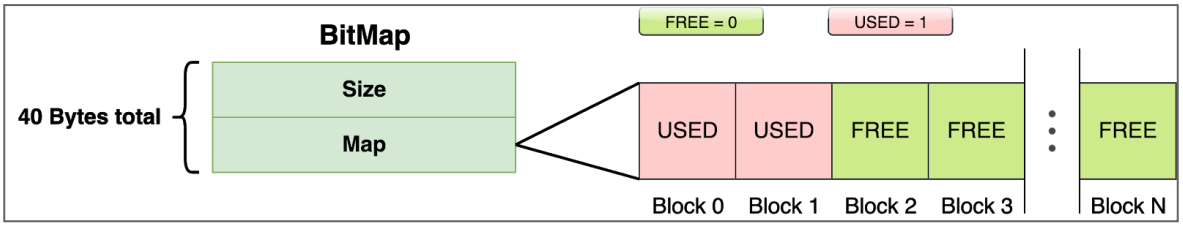
3. *Size*: Guardamos aquí el tamaño del archivo (no confundir con el offset, que se guardará en la tabla de archivos abiertos filetable).
4. *Blocks assigned*: Se refiere a los bloques que un archivo lleva asignados. Aunque parezca un atributo redundante del inodo ya que podríamos derivarlo viendo cuales de los bloques tienen números de bloque asignados, vimos que era de de gran ayuda en las funciones de lectura y escritura tener esa información de antemano para evitar los cálculos de derivación.
5. *Data_blocks*: Podemos apreciar el la figura esquemático de un inodo que el direccionamiento que se hace, es siempre de forma directa. Hicimos esto para simplificar las funciones de lectura y escritura aprovechándose del NF3 que indicaba que los ficheros tendrían un máximo de 10 KiB, o lo que es equivalente: 5 bloques de 2048 Bytes cada uno.
6. *Soft_link*: El valor de este campo es -1, cuando el inodo no representa un soft link. En el caso de que este sea un soft link lo que contiene este campo es el índice de inodo donde se encuentra la información del archivo referenciado, es decir el inodo que contiene el conjunto de bloques de datos que representa la información del archivo.

   Algo importante a mencionar sobre nuestro diseño de los enlaces simbólico en nuestro sistema de ficheros es el siguiente aspecto: Hemos permitido crear **enlaces simbólicos de un solo nivel**, o dicho de otra manera, no hemos permitido realizar realizar un enlace simbólico que apunte a un enlace simbólico ya existente. Hemos hecho esto con intención de simplificar nuestro diseño cumpliendo con la funcionalidad pedida y evitar problemas indeseados como bucles infinitos en acceso a archivos.
7. *Has_integrity*: Este es un campo binario que indica si los datos del fichero del inodo en cuestión tienen integridad o no. En el caso de un enlace simbólico este valor estará puesto a 1(verdadero, tiene integridad) en el archivo original.
8. *integrity*: un campo que usamos para almacenar el resultado de la función CRC 32 bits, que nos da el checksum al proporcionarle el buffer con el contenido de los bloques de datos de nuestro fichero.

Mapa de Bits

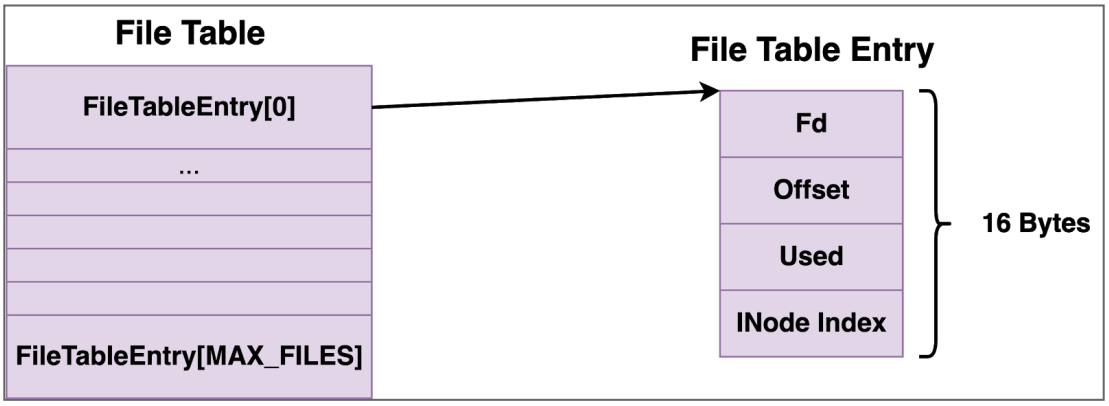
Utilizamos un mapa de bits para poder acceder en todo momento al estado de cada bloque en disco, usando las constantes FREE (0) o USED (1). Cuando necesitamos asignar o ampliar el espacio necesario para un fichero recorremos bit a bit el mapa y usamos el primer bloque libre, de manera que priorizamos los bloques con un índice menor y concentramos los bloques usados al principio.

Almacenamos el tamaño total del bitmap para poder recorrer el mapa de manera sencilla y evitar que se asignen bloques inexistentes dado que el mapa puede tener más bits que bloques (esto es debido a que lo construimos usando arrays de caracteres y por tanto tiene que ser un múltiplo de 8).



FileTable

El file table es una estructura de memoria que no se almacena en disco. Es la estructura que usa el sistema operativo para poder almacenar los ficheros abiertos en un momento determinado y poder acceder a su i-nodo, además de guardar el offset de dicho fichero.



Algoritmos y optimizaciones

Es este apartado nos centraremos solamente en las funciones que hemos implementado que consideramos tienen mayor carga algorítmica, ya que algunas de ellas como son por ejemplo *lseekFile* se limitan a cambiar un parámetro en la *filetable*. También otras como createFile, openFile o closeFile resultan bastante triviales ya que es son algoritmos bastante lineales y de reseteo de estructuras de datos muy sencillas.

writeFile

Tal vez el algoritmo implementado para escribir en disco sea el que más tiempo nos ha tomado pensar para no sobrecargarlo haciéndolo muy lento y para cumplir los requisitos pedidos. Nuestro algoritmo es el siguiente:

1. Comprobamos que el *fd* proporcionado es válido (existente en la *FileTable*).
    a. si no lo encontramos devolvemos error
2. Mientras tengamos todavía bytes que escribir:
    a. Rellenamos un buffer con el carácter '_' para ver claramente en el fichero disk.dat los bloques en los que se ha escrito.
    b. Si la escritura se va a realizar en el último bloque para tratarlo de manera especial:
        i. Calculamos cuántos bytes han sido ya escritos en el bloque.
        ii. Guardamos el mínimo entre el espacio restante en el bloque y los bytes que quedan por escribir

iii. Escribimos los bytes desde el offset que tengamos en ese bloque(nótese que cada vez que escribimos en disco, incrementamos el offset)

c. Si la escritura no se va a realizar aún en el último bloque comprobamos si el bloque actual ha sido reservado:
   i. Si no ha sido reservado, lo reservamos y escribimos en nuestro buffer incluso hasta 2048 bytes (nótese que si necesitaremos menos de 2048 tendríamos '_' en el resto de nuestro bloque de buffer')
   ii. Si ya ha sido reservado calculamos cuánto espacio tenemos libre en el bloque actual y tenemos en cuenta ese espacio como longitud máxima de escritura para la iteración

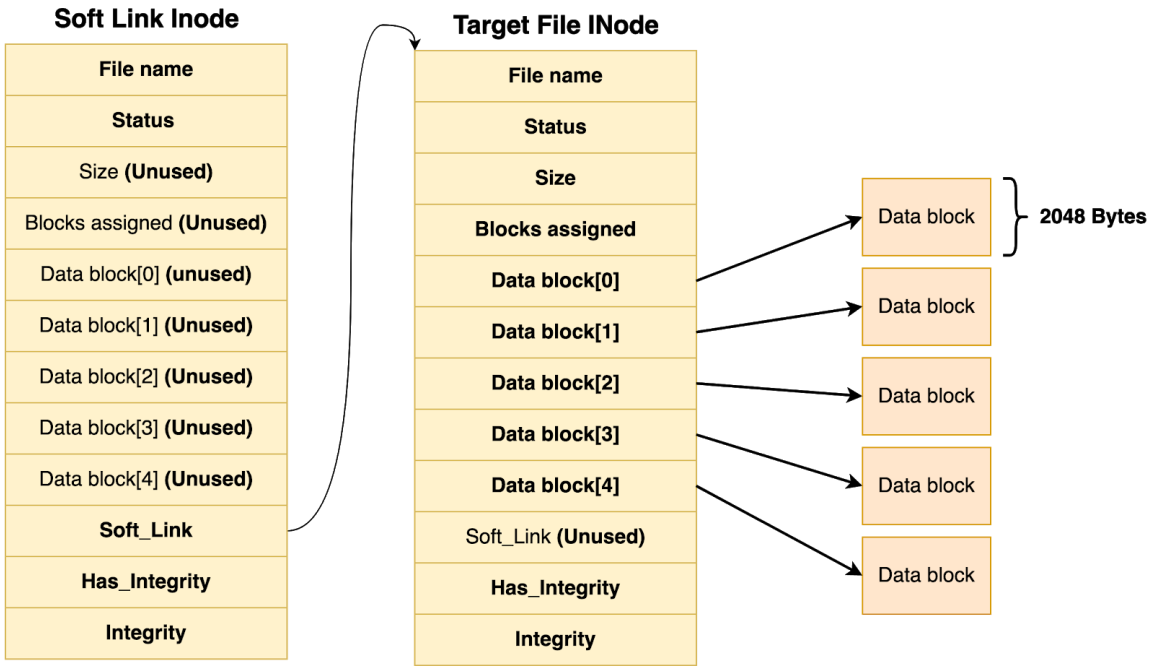
d. escribimos lo que tengamos en nuestro buffer en disco

3. Al acabar devolvemos los bytes escritos con éxito e incrementamos el tamaño de nuestro archivo en el i-nodo correspondiente. Nótese que el offset se ha ido autoincrementando en cada iteración del paso 2 del algoritmo.

readFile

1. Comprobamos que el descriptor de fichero existe.
2. Comprobamos si el fichero está vacío o si el offset está al final del fichero, en cualquiera de los dos casos devolvemos 0 (no vamos a poder leer ningún byte) y no escribimos nada en el buffer.
3. Para simplificar el algoritmo, leemos la totalidad de bloques que ocupa la información del fichero. Para ello seguimos los creamos un buffer temporal que pueda contener todos los bytes del tamaño del fichero.
4. Leemos bloque a bloque y almacenamos en nuestro buffer temporal.
5. Calculamos el primer y último byte que componen el rango del offset y el argumento numBytes.
6. Copiamos este rango de bytes del buffer temporal al buffer proporcionado como parámetro y devolvemos el número de bytes que hemos escrito.

createLn



1. Comprobamos si el nombre proporcionado de destino de nuestro soft_link, es decir *filename* ya está registrado para otro archivo, si es así abortamos la creación del enlace simbólico
2. Solo permitiremos links directos a archivos originales propiamente dichos, es decir un enlace simbólico nunca podrá apuntar a otro enlace simbólico. Con lo cual si comprobamos que el target de nuestro link es otro enlace simbólico abortamos la operación.
3. Comprobamos que el nombre del link está disponible y no ha sido usado ya por otro fichero. Si hubiese sido ya usado, abortamos también la operación.
4. Se busca un i-nodo libre donde localizamos nuestro enlace simbólico, si no lo hubiese abortamos la operación
5. Llegamos a este punto tenemos ya un inodo libre para nuestro enlace simbólico.
    a. Marcamos nuestro inodo como enlace simbólico cambiando el valor de *soft_link* al valor del inodo destino que encontramos indirectamente en el paso 1. De esta manera ya podremos identificar en el inodo que se trata de un enlace simbólico y podremos acceder a su información por medio del inodo del fichero destino

removeLn

1. Comprobamos si el nombre de enlace simbólico proporcionado existe en el SF. Si no existiese abortamos la operación de borrado devolviendo error.
2. Existiendo el nombre, vemos si ese nombre se refiere a un enlace simbólico y no a un fichero “normal”. Si fuese un fichero normal abortamos la operación de borrado.
3. Una vez habiendo verificado que se trata de un enlace simbólico, simplemente reseteamos los valores del inodo y lo marcamos como libre para habilitar su uso.

Plan de pruebas

Nombre	Objetivo	Procedimiento	Entrada	Salida esperada
F1.1 & F8 mkFS	Probar la correcta funcionalidad de la creación de las estructuras en disco.	Ejecutamos mkFS escribiendo todas las estructuras esenciales en disco.	Tamaño de volúmen (device size)	Valor de retorno 0 indicando que se han creado las estructuras esenciales en disco sin errores.
F1.2- mountFS	Probar la correcta lectura de estructuras de datos de disco y su volcado en mem.	Ejecutamos mountFS leyendo las estructuras de disco y volcándolas en memoria.	N/A	Valor de retorno 0 indicando que se han creado las estructuras de datos en memoria.
F1.3- unMountFS	Comprobar que las estructuras se escriben correctamente en disco.	Ejecutamos unmountFS.	N/A	Valor de retorno 0 indicando que se han escrito las estructuras de datos en el disco.
F1.4- createFile	Comprobar que se crea correctamente el fichero.	Ejecutamos createFile.	Nombre del fichero válido.	Valor de retorno 0 indicando que se ha creado el fichero.
F1.5- fileWasDeleted	Comprobar que se borra correctamente el fichero.	Ejecutamos removeFile.	Nombre del fichero válido.	Valor de retorno 0 indicando que se ha borrado el fichero.
F1.6- testOpenFile	Comprobar que se abre correctamente el fichero.	Creamos un fichero, ejecutamos openFile sobre éste.	Nombre del fichero creado.	Valor de retorno 0 indicando que se ha abierto el fichero.
F1.7- testCloseFile	Comprobar que se elimina correctamente el fichero.	Creamos un fichero, ejecutamos openFile y closeFile	El descriptor de fichero proporcionado por openFile.	Valor de retorno 0 indicando que se ha cerrado el fichero.

F1.8(1) & F2 testReadFile NoOffset	Comprobar que podemos hacer una lectura correcta dentro de un fichero recién abierto (offset = 0)	Creamos un fichero, lo abrimos y escribimos en el 212 bytes. Lo cerramos y lo volvemos a abrir. Luego leemos y comprobamos que obtenemos lo mismo que previamente hemos escrito.	descriptor de fichero válido del archivo escrito	Valor de retorno 0 de la función *strcmp* sobre los dos strings, el inicialmente escrito y el posteriormente leído de disco.
F1.8(2) testReadFile OffsetAtEnd	Comprobar que se leen 0 bytes de un fichero con el offset al final de éste.	Se crea un fichero, se escribe contenido en el, se lee el fichero.	El fd, un buffer y un tamaño > 0 para leer.	La función readFile devuelve 0 porque no ha podido leer más allá del tamaño actual del fichero.
F1.8(3) & F6 testReadFile5 Blocks	Comprobar que podemos leer más de un bloque usando varias operaciones de lectura	Se crea un fichero. Abrimos el fichero y escribimos en el 10240 bytes. Después lo leemos y comprobamos que lo leído coincide con lo escrito	descriptor del fichero a leer, y buffer donde se vuelca el resultado	Valor de retorno 0 de la función *strcmp* sobre los dos strings, el inicialmente escrito y el posteriormente leído de disco.
F1.9(1) testWriteLess ThanBlock	Comprobar que se puede escribir menos que el tamaño de un bloque de disco	Se crea el fichero, se escribe un número de bytes inferior al tamaño de bloque.	El fd y un número de bytes a escribir < tamaño de un bloque	WriteFile devuelve correctamente un número de bytes escritos < tamaño de un bloque.
F1.9(2) & F7 & NF3 testWriteMore ThanBlock	Comprobar que se puede escribir más que el tamaño de un bloque de disco.	se crea un fichero, se escriben varios bloques de disco.	El fd y un número de bytes a escribir > tamaño de un bloque	WriteFile devuelve correctamente un número de bytes escritos > tamaño de un bloque.
F1.10(1) testLseekEnd	Comprobar que lseek cumple con la opción SEEK_END	Se crea un fichero, escribimos 10 bytes, cerramos el fichero y lo volvemos a abrir para llevar el offset al inicio. realizamos un lseek con SEEK_END	descriptor de fichero y opción SEEK_END (parámetro offset es ignorado)	Comprobamos que el valor de retorno es 0 indicando que se ha realizado el seek end con éxito

F1.10(2) testLseekBegin	Comprobar que lseek mueve el offset al primer byte del fichero.	Se crea un fichero, se escriben 10 bytes dentro, se ejecuta lseek	fd y opción FS_SEEK_END	Comprobamos que el valor de retorno es 0 indicando que se ha realizado el seek end con éxito
F1.10(3) testLseekAfterSize	Comprobar que no permitimos llevar el puntero más allá del size del fichero	Creamos y escribimos un fichero de 10 Bytes. Teniendo el puntero al final del fichero, intentamos mover un byte más allá	descriptor de fichero del archivo, 1 byte en offset y opcion SEEK_CUR	Comprobamos que el resultado devuelto es -1 indicando que se ha controlado con éxito la situación límite de movimiento del cursor más allá del final del fichero
F1.10(4) testLseekBelow0	Comprobar que no se puede mover el offset a un valor negativo	Se crea un fichero, se escriben 10 bytes dentro, se ejecuta lseek	fd y opción FS_SEEK_CUR y un offset de -11	Comprobamos que el valor de retorno es -1 indicando que no se ha podido mover el offset a un valor negativo.
F1.11 testCheckFileIntegrity	Comprobar que se comprueba correctamente la integridad de un fichero	Se crea un fichero con integridad y se llama a la función checkIntegrity	Nombre del fichero con integridad	Comprobamos que checkFile devuelve 0 indicando que la integridad es correcta.
F1.12 testIncludeIntegrity	Comprobar que se puede añadir integridad de manera correcta a un fichero que no la tenía previamente	Se crea un fichero, se escribe en él, se cierra este fichero y se incluye la integridad sobre el mismo.	el nombre del fichero	Comprobamos que el valor de retorno es 0, comprobando así que se ha calculado y puesto el valor de integridad en el inodo asociado al nombre del fichero
F1.13 testOpenFIleIntegrity	Comprobar que se puede abrir un fichero con integridad correctamente.	Se crea un fichero con integridad y se intenta abrir con la función openFileIntegrity()	El nombre del fichero con integridad.	Comprobamos que la función openFileIntegrity devuelve 0 indicando que se ha podido abrir y comprobar la integridad correctamente.

F1.14 & NF12 testCloseFileIntegrity	Comprobar que se puede cerrar un fichero con integridad correctamente.	Se crea un fichero con integridad, se abre con integridad y se intenta cerrar con la función closeFileIntegrity()	El descriptor de fichero del fichero con integridad.	Comprobamos que la función closeFileIntegrity devuelve 0 indicando que se ha podido cerrar y comprobar la integridad correctamente.
testCheckFileAlreadyOpen	Comprobar que no se permite realizar la operación checkFile() cuando un fichero ya estaba abierto	Se crea un fichero, se escribe en él y se deja abierto a propósito para comprobar que checkFile tiene en cuenta que el fichero que se va a comprobar ha sido ya abierto y no siga su operación	el nombre del archivo	Comprobamos que el código de retorno de checkFile es -2, dando a conocer que se ha capturada el error de un fichero ya abierto.
testOpenFileIntegrityLink	Comprobar que se puede abrir un fichero con integridad usando un soft link.	Se crea un fichero y se escribe en él. Se le añade integridad, se cierra y se crea un link al fichero. Se llama a openFileIntegrity().	El nombre del link al fichero original con integridad.	Comprobamos que openFileIntegrity() no devuelve ningún tipo de error dado que ha encontrado el fichero original.
NF11 testCloseFileWithIntegrity	Comprobar que closeFile() no puede cerrar un fichero con integridad.	Se crea un fichero con integridad, se abre y se llama a closeFile()	El descriptor de fichero que tiene integridad.	closeFile() devuelve un error dado que no puede cerrar un fichero con integridad.
NF8 testOpenFileWithIntegrity	Comprobar que openFile() no puede abrir un fichero con integridad.	Se crea un fichero con integridad y se abre con openFile()	El nombre del fichero que tiene integridad.	openFile() devuelve -2 indicando que no puede abrir un fichero con integridad.
F1.15 testCreateLn	Comprobar que se puede crear un enlace blando.	Se crea un fichero, se escribe en él y se crea un enlace blando, se usa readfile sobre el enlace blando	el fd devuelto por openFile sobre el enlace blando	Se lee correctamente el contenido del fichero original usando el enlace blando.

F1.16 testRemoveLn	Comprobar que se puede eliminar un enlace simbólico	Se crea un fichero en el cual se escriben 10 bytes. se cierra el fichero y se crea un enlace simbólico. Se ejecuta posteriormente removeLn	el nombre del enlace simbólico	Se comprueba que el valor devuelto es 0, certificando así que el enlace simbólico ha sido eliminado
NF1 testCreateMoreThanMaxFiles	comprobar que el SF no permite crear más de 48 archivos	se crean 48 archivos y se añade un último archivo (n 49)	nombres de forma secuencial como: archivo1.txt archivo2.txt.. archivoN.txt	Se comprueba que los primeros 48 archivos son creados satisfactoriamente y que el último archivo n49, devuelve un error en su creación, al haber excedido el número de archivos permitidos en el sistema.
NF2 createFileWithNameTooLong	Comprobar que la longitud máxima del nombre de un fichero son 32 caracteres.	Se intenta crear un fichero con un nombre de más de 32 caracteres.	nombre del fichero mayor que 32 caracteres.	Se comprueba que el código de error devuelto es -2, indicando que se ha tenido en cuenta la limitación del nombre a 32 caracteres
NF6(1) testMakeFileDeviceTooSmall	Comprobar que el límite mínimo de tamaño de volúmen es respetado en la creación del SF	Se intenta realizar la operación mkFS con un tamaño de dispositivo menor a 460KiB	tamaño del volúmen	Se comprueba que se obtiene el código de error de retorno -1
NF6(2) testMakeFileDeviceTooBig	Comprobar que el límite máximo de tamaño de volúmen es respetado en la creación del SF	Se intenta realizar la operación mkFS con un tamaño de dispositivo mayor a 600 KiB	tamaño del volúmen	Se comprueba que se obtiene el código de error de retorno -1

NF10 testOpenFileIntegrityNoIntegrity	Comprobar que no sea puede abrir un fichero que no tenga integridad con la función de abrir ficheros con integridad (openFileIntegrity)	Se crea un fichero, se cierra el fichero sin añadirle integridad y luego se intenta abrirlo con integridad	nombre del fichero	Se comprueba que el valor de retorno es -3, indicando que no se ha podido abrir un fichero sin integridad mediante el uso de la función: openFileIntegrity().

Conclusiones

Principales problemas encontrados y sus soluciones

- **Problema**: Tuvimos que cancelar nuestro primer intento de implementación de writeFile, debido a que intentábamos hacerlo demasiado genérico
    - **Solución**: Tomamos la estrategia de granularizar un poco más los casos dentro de la escritura en disco de manera que pudiesemos tener más control sobre nuestras depuraciones.
- **Problema**: Tuvimos que cancelar nuestro primer intento de creación en disco de nuestras estructuras de datos para que cupiese todo en un bloque, pero vimos que a medida que íbamos avanzando en la práctica cada vez que necesitábamos un solo dato más en un inodo debíamos multiplicarlo su tamaño por 48, ya que ese dato aparecería en cada uno de nuestros 48 i-nodos.
    - **Solución**: Decidimos primero intentar optimizar el tamaño que ocupaban los inodos incluso a nivel de bit, pero aún así resultaba imposible que entrasen en el primer bloque junto al superbloque y el bitmap. Finalmente tomamos la decisión de usar los dos primeros bloques para metadatos.
- **Problema**: Tuvimos un error de concepto al pensar que la indicación en el NF10 de que el include integrity debía hacer “previamente” al *openFileIntegrity*, ya que pensamos que se debía hacer dentro de la propia función *openFileIntegrity*.
    - **Solución**: Simplemente cambiamos la lógica de nuestras funciones de integridad de manera que *openFileIntegrity* devolviese error si se le pasaba un fichero sin integridad, obligando así a que fuese el propio usuario quien incluyese la integridad en los ficheros.