Universidad Carlos III

Curso Diseño de Sistemas Operativos 2019-20

Práctica 2

Apellidos, Nombre	**NIA**	**Correo**
Parrado Arribas, Alejandro	100383453	100383453@alumnos.uc3m.es
Sanz Gómez, Adrián	100383473	100383473@alumnos.uc3m.es
Vinagre Blanco, Andrés	100383414	100383414@alumnos.uc3m.es

Índice

1 Introducción

En este documento se redactará el desarrollo, diseño e implementación de la práctica 2 **Sistema de Ficheros** de la asignatura Desarrollo de Sistemas Operativos.

El objetivo de esta práctica es ver de **una manera interactiva y a menor escala** como se implementan los sistemas de ficheros dentro de dispositivos como discos duros. Para ello se crearán estructuras auxiliares que ayuden a la manipulación del mismo, así como a la creación y modificación de archivos incluidos en el sistema.

Para ello, primeramente, se detallará todo el **diseño del sistema,** incluyendo estructuras y funciones, seguido de una batería de pruebas para comprobar que se han contemplado todos los requisitos funcionales y no funcionales del enunciado y finalmente una conclusión, donde se redactará los problemas encontrados y la opinión personal.

2 Diseño detallado del sistema de ficheros.

2.1 Estructuras de datos necesarias para el sistema de ficheros.

Para la implementación del sistema de ficheros, se han necesitado las siguientes estructuras de datos.

- **Superbloque:** se dispone de una estructura de datos superbloque, que es donde se encuentra toda la **información del sistema** de ficheros. Está formada por el número mágico, el número de bloques que ocupa el mapa inodos, el número de bloques que ocupa el mapa de datos, el número de inodos del sistema de ficheros, el número de bloque donde está el primer inodo, el número de bloque donde está el primer bloque de datos, el tamaño del dispositivo y el propio padding de este superbloque.

- **Inodo:** esta estructura guarda **metadatos de cada inodo**, como tipo (puede ser fichero o enlace simbólico), nombre (que será el nombre de cada fichero o enlace del sistema con un tamaño máximo de 32), tamaño actual del fichero, array de bloques directos, que es un array formado por 5 posiciones para cada bloque donde inicialmente están todas a -1, ya que no hay datos en el fichero y si hay datos este array contendrá el primer bloque libre. Además, contiene un array de firmas para cada bloque de datos del fichero, por lo que cada bloque de datos tendrá una firma asociada, ya que el cálculo de la misma se hace por bloques.

- **Inodo Extra:** es una estructura auxiliar inodo que contiene **metainformación** que **no** hace falta que **se guarde en disco** ya que es sobre el estado del fichero en ese momento. Por lo que tenemos varios campos: abierto (0 si el fichero cerrado, 1 si abierto), posición (es el puntero de posición escritura/lectura del propio fichero), integridad (0 si no la tiene, 1 si la tiene).

- **Imap:** mapa de bits de inodos, donde 0 significa inodo libre y 1 ocupado.

- **Bmap:** mapa de bits de bloques, donde 0 significa bloque libre y 1 ocupado.

Además, se han definido las siguientes constantes:

- **BLOCK_SIZE: 2048.** Es el tamaño de cada bloque.

- **DISK:** “disk.dat”. El fichero disco.

- **T_FICHERO y T_ENLACE_S**, 1 y 2, respectivamente, indica si un inodo es un fichero o es un enlace simbólico.

Y una variable global **is_mount** para indicar si el sistema de ficheros está montado (1) o no (0).

Para iniciar el sistema de ficheros, se ha creado un vector de tamaño 1 que contiene el **superbloque** y también se han creado **dos vectores** de tipo inodo e inodo extra, respectivamente, ambos de tamaño 48.

2.2 Descripción de las funciones auxiliares:

Para la implementación del sistema de ficheros se ha incluido las siguientes funciones auxiliares:

- **metadata_setDefault:** función que restablece los metadatos del sistema de ficheros y los guarda a discos. Se inicializa el **número mágico**, 0x12345, el **número de inodos** (48), el **número de bloques** que ocupa de mapa de datos a 1, el de inodos a 1, el **primer inodo** en el bloque 3, el **número de bloques de datos** (240, puesto que hay 48 inodos que tienen como máximo 5 bloques directos a datos, por lo que 48*5 = 240), **primer bloque de datos** (1+1+1+48) y **tamaño del dispositivo**. Además, se inicializan **los mapas de bits de inodos** y **bloques** a 0 y se inician todos **los inodos** a 0.

- **metadata_writeToDisk:** escribe todos los metadatos a disco. El **superbloque, los bloques correspondientes al mapa de inodos** y al **mapa de bloques** y también escribe toda la información de cada inodo en los bloques de discos que ocupen los 48 inodos.

- **metadata_readFromDisk:** lee todos los metadatos de disco explicados en la función anterior y los escribe de nuevo en las estructuras.

- **namei:** función que **busca el inodo** con el nombre introducido por parámetro. Lo que hace el atributo Tipo es permitir al usuario poder solicitar el inodo de un fichero o de un enlace. Se **contemplan 4 casos**, el primero que el usuario solicite el **inodo de un enlace**, en ese caso se devuelve el inodo del enlace; el segundo que el usuario solicite **el inodo de una fichero** pero le pasen el **nombre de un enlace**, la función devolverá el inodo del fichero al que apunta el enlace; el tercero se solicita **el inodo de un fichero**, el programa devuelve el inodo del fichero; y por último en caso de que **no exista** ningún inodo por el nombre introducido, devuelve un error -1.

- **ialloc:** busca el **primer inodo libre** del sistema de ficheros consultando el mapa de bits de inodos y en el caso de encontrar un inodo libre lo **ocupa**, establece valores

por defecto y devuelve el identificador del inodo (de 0 a 47). En el caso de no encontrar ningún inodo libre devuelve -1.

- **alloc:** funciona de forma similar al ialloc, solo que busca el **primer bloque de datos libre, lo ocupa**, inicializa a 0 el bloque entero y lo escribe a disco y devuelve el identificador de bloque. En el caso de no haber ningún bloque libre retorna -1.

- **ifree: libera un inodo** cuyo identificador se pasa por parámetro, por lo que se cambia el bit de ocupación de ese inodo en el mapa de bits de 1 a 0. Devuelve error en el caso de que el id del inodo no se salga del rango de los inodos disponibles del sistema de ficheros (0 a 47).

- **bfree:** funciona de forma similar al **ifree** pero con los bloques, cambia el **bit de ocupación** del bloque introducido por parámetro de 1 a 0. Devuelve el error en el caso de que se dé un número de bloques inválido en el sistema de ficheros (recordar que tiene 240 bloques de datos).

- **bitmap_print(char *bitmap_, int size):** permite imprimir por pantalla el mapa de inodos y de bloques, dependiendo de que le pases por parámetro. Esta función es útil para la comprobación de errores.

- **sblock_print():** imprime el superbloque asociado al sistema de ficheros. Esta función al igual que el anterior sirve para comprobar errores.

2.3 Descripción de las funciones implementadas

NOTA: los tests asociados a cada función se encuentran en la batería de pruebas

- **mkFS(long deviceSize):** crea la estructura de almacenamiento del dispositivo. Genera el sistema de ficheros que se ha diseñado para la realización de la práctica. Para ello utiliza **metadata_setDefault**, asigna el tamaño al dispositivo, escribe en disco con **metadata_writeToDisk** e inicializa los bloques.

- **mountFS(void):** monta el sistema de ficheros en un dispositivo simulado. Se comprueba previamente que la variable global is_mount tenga valor 0, para asegurar que no hay ningún sistema de ficheros ya creado. La variable is_mount toma el valor 1. Para montar utiliza **metadata_readFromDisk.**

- **unmountFS(void):** para desmontar un sistema de ficheros se comprueba previamente que no haya ningún fichero abierto y que no esté ya desmontado. La variable global is_mount se actualiza a valor 0. Para montar utiliza **metadata_writeToDisk.**

- **createFile(char *fileName):** esta función **permite crear los ficheros**. Se comprueba previamente que el nombre del fichero no tenga sobrepase los 32 caracteres y no coincida con el nombre de otro fichero ya que creado. Se crea variable inodo id, y bloque id para el primer bloque de datos. En el caso de que no haya bloques disponibles, **se libera el inodo** ocupado con anterioridad. Además, como es de tipo fichero se establece el campo “tipo” a la **constante T_FICHERO.**

- **removeFile (char *fileName):** para eliminar un fichero es necesario localizar el Inodo asociado, liberar el inodo y los bloques de datos asociados a él. También es necesario controlar que el fichero exista.

- **openFile (char *fileName):** al abrir un fichero es necesario primero **localizar Inodo del fichero asociado** y actualizar los campos de inodos_x abierto a 1 y posición a 0. Para cumplir con los requisitos no funcionales, si el fichero tiene integridad, devuelve error ya que no puede abrirse con esta función.

- **closeFile (int fileDescriptor):** a la hora de cerrar un fichero hay que comprobar que esté abierto y que el descriptor de fichero sea válido. **No puede cerrarse un fichero que está cerrado**. Además, si ese fichero tiene **integridad**, se devuelve un error ya que no puede cerrarse con esta función.

- **bmap (int inodo_id, int offset):** te devuelve el bloque donde está el puntero de posición de un Inodo. Se verifica que el bloque resultante no esté fuera del rango permitido.

- **readFile (int fileDescriptor, void *buffer, int numBytes):** esta función permite leer de un fichero representado por *fileDescriptor* un número de bytes definido por *numBytes* y escribirlo en un *buffer.*

  Para ello, primero se crea un array auxiliar con un tamaño total del máximo permitido para un inodo (5 bloques de datos). Si el fichero está cerrado devuelve -1, sino calcula si el número de bytes a leer es mayor que el del fichero, en cuyo caso se ajustan los bytes al máximo.

  Si el número de bytes es negativo dará error y si es 0 significa que el puntero de lectura/escritura está al final del fichero por lo que retornará 0 bytes leídos.

  A partir de aquí entra la variable *bytesLeft,* que se irá actualizando para ver cuantos bytes quedan por leer. A continuación, se calcula el bloque que tiene que leer a través de la función **bmap** y entra en un bucle que utiliza **bread** para leer el bloque y **bmap** para el siguiente bloque en caso de ser necesario.

  Después utiliza **memmove** para copiar lo leído en el buffer y finalmente actualiza posición y retorna el número de bytes leídos.

- **writeFile (int fileDescriptor, void *buffer, int numBytes):** esta función permite escribir en un fichero representado por *fileDescriptor* un número de bytes *numBytes* el contenido de un *buffer*.

  Su estructura es similar a la usada en el **readFile.** Primero se crean todas las variables que se van a usar que las comentaremos según se vayan usando.

  Si el fichero está abierto retorna -1, si el puntero de posición está al final de archivo en su máxima capacidad se retorna 0, así como los bytes a copia *size* es 0.

  Si *numBytes* es mayor que el tamaño del buffer significa que se va a escribir el buffer entero, así que se actualiza *bytesLeft* que es la variable que indica los bytes que faltan

por escribir. A continuación comienza el bucle de escritura mientra falten bytes por escribir.

Para ello primero actualiza *size* con *bytesLeft* y saca el bloque al que hay que escribir con **bmap**. Si *b_id,* que indica el bloque, es -2 significa que el archive está lleno y si es -1 que no hay bloques libres, por lo que hay que vincular otro al inodo. Si este es el caso se llama a **alloc** y a través de *b_log_aux* que indica el bloque lógico se saca *b_aux* que será el nuevo *b_id.*

Una vez hecho esto se realiza **memmove** para copiar en el buffer auxiliar *b* y se escribe el bloque en el disco a través de **bwrite**. Finalmente se actualiza posición, el tamaño del inodo y los bytes que quedan por escribir, así como los bytes escritos totales *bytesWr,* que es el valor de retorno*.*

- **lseekFile (int fileDescriptor, long offset, int whence):** está función modifica el puntero de posición lectura/escritura de un fichero representado por *fileDescriptor,* a una posición *whence* con un *offset* en un caso determinado.

  Para ello primero comprueba que el fichero sea válido y este abierto el fichero. Después comprueba el valor del *whence,* si es **BEGIN** simplemente se ajusta el puntero al principio; si es **END** se ajusta al final teniendo en cuenta la cabecera; y si es **CUR** se actualiza sumando offset, comprobando los límites ya actualizando el puntero a ellos si se sobrepasa.

- **checkFile (char *fileName):** comprueba la integridad de un fichero, por lo que es necesario el id del inodo asociado a fileName y el id de cada bloque de datos del fichero.

  Para ello, **se calcula la firma de cada bloque de datos** del fichero con el algoritmo CRC32 y se compara con la firma que hay en el fichero. De forma que, si una firma recién calculada no coincide con la que ya está, el fichero está corrupto, por lo que la función devuelve -1. Devuelve -2 en el caso de que la función no tuviera integridad (por lo que no puede checkearse), el parámetro fileName exceda los 32 caracteres y si el fichero está abierto.

- **includeIntegrity (char *fileName):** función que se encarga de **incluir integridad** a un fichero. Para eso obtiene el id del inodo asociado a ese fichero y para cada bloque de fichero con datos, se calcula una firma que se introduce en el array de firmas de integridad definido en la struct de inodo. Además, como se ha introducido la integridad, se pone en el vector inodos_x para ese fichero el campo integridad a 1.

- **openFileIntegrity (char *fileName):** para la apertura de un fichero con integridad, se obtiene el **id del inodo asociado** a ese nombre de fichero y se devuelve -1 en el caso de que el parámetro fileName sobre pase los 32 caracteres o el fichero no exista, -3 si el campo de integridad del vector inodos_x del inodo obtenido está a 0 (no se ha calculado integridad), y -2 si la integridad del fichero no es correcta, por lo que **se llama a la función *checkFile*** y en el caso de que el fichero esté corrupto se devuelve -2. Además, pone el puntero de posición a 0 y marca ese fichero como abierto (vector inodos_x).

- **closeFileIntegrity (int fileDescriptor):** para cerrar un fichero con integridad es necesario **actualizar la firma de cada bloque asociado** al fichero con el algoritmo **CRC32** por lo que, se accede al vector firmas de integridad de ese inodo y se actualiza la firma para cada bloque de datos del fichero en el vector de firmas del inodo. Se devuelve -1 si descriptor del fichero no existe en el sistema, si ya está cerrado o si no tiene integridad y quiere cerrarse con esta función. Devuelve 0 si la ejecución está abierta y el campo abierto se pone a 0 ya que el fichero se ha cerrado.

- **createLn(char *fileName, char *linkName):** Esta función se encarga de crear un **enlace simbólico**, para ello **localiza un inodo libre y el inodo del fichero** al que tiene que apuntar. Como decisión de diseño se ha utilizado el mismo *struct* de los inodos, se ha utilizado la **primera posición de bloque directo** para guardar el inodo del fichero al que apunta porque son unos campos que un enlace simbólico no utiliza ya que no contiene datos.

- **removeLn(char *linkName):** para eliminar un enlace simbólico se localiza el inodo del enlace y se liberan los bloques asociados al enlace simbólico así como el inodo.

3. Descripción de alto nivel de la funcionalidad principal.

El **sistema de ficheros** que implementa el disco permite crear un **máximo de 48 ficheros** o enlaces a esos ficheros. Por lo tanto, podrás acceder a tus ficheros desde casi cualquier parte. Una característica esencial es la **seguridad**, queremos que todos tus ficheros estén a salvo de malware que quiera corromperlo, por lo que permitimos abrir, cerrar y añadir a todos tus **ficheros con integridad**, por lo que podrás detectar de forma fácil modificaciones realizadas por terceros a tus datos, gracias al cálculo de firmas realizado de forma unívoca para los datos.

4. Comportamiento no definido.

En este apartado se procede a explicar algunas decisiones de diseño tomadas por nosotros sobre este sistema de ficheros.

Por un lado, como cada **inodo / fichero tiene un puntero de posición asociado,** hay situaciones en las que no se nos ha indicado si había que restablecerlo al principio o no, por tanto, hemos decidido mantener el puntero de posición donde se encontraba en el momento de llamar a una función, esto sucede en las funciones ***closeFileIntegrity, includeIntegrity, checkFile*** ya que se necesita que el puntero de posición se mueva para actualizar, añadir o comprobar integridad, respectivamente, por lo que, al final de cada función, se **restaura al estado inicial** antes de la ejecución de estas funciones.

Por otro lado, para todas las funciones relacionadas con integridad sobre los datos, se ha decidido optar por el **algoritmo CRC32**, ya que para nuestros ficheros el CRC16 podría quedarse corto y el CRC64 ser demasiado seguro en exceso. El CRC32 devuelve una **firma de 32 caracteres** y el cálculo de ella se ha realizado para cada bloque de datos del fichero,

por lo que cada bloque directo de datos asociado al fichero tiene una firma que se guarda en el vector **"firmasIntegridad"** del inodo del fichero.

También se ha considerado que, a la hora de **crear un fichero**, este no se abrirá, sino que simplemente será creado.

Otra decisión de diseño que se ha decidido tomar es que **como máximo puede haber 48 ficheros y/o enlaces** en nuestro sistema de ficheros. Podría haberse tomado la decisión de que, si hay 48 ficheros, podría haber hasta 48 enlaces a ellos, pero, por simplicidad, solo se permite un máximo de 48 ficheros y/o enlaces en nuestro sistema.

Para evitar bucles relacionados con los **enlaces simbólicos**, se ha decidido que cuando el usuario intente crear un enlace a otro enlace que apunta a un fichero, este nuevo enlace va a directamente apuntar al fichero. El usuario quiere E2 → E1 → F, entonces el sistema internamente, a la hora de la creación de E2 dará como resultado **E2 → F** y por otro lado seguiremos teniendo **E1 → F**. Gran parte del funcionamiento de este sistema está en **namei.**

Como decisión de diseño, se reutiliza la estructura de los inodos para los enlaces simbólicos. Se usa la posición 0 de bloque directo para guardar el inodo del fichero al que apunta el enlace.

4. Comprobación de requisitos.

Requisitos no funcionales

Identificador	Se cumple	Tests Asociados (test.c)
F1.1	**Si**	**00, 01**
F1.2	**Si**	**02**
F1.3	**Si**	**22, 26**
F1.4	**Si**	**03**
F1.5	**Si**	**25, 25**
F1.7	**Si**	**13**
F1.8	**Si**	**6, 8 y 10**
F1.9	**Si**	**5, 11, 12, 14, 21**
F1.10	**Si**	**7 y 9**
F1.10	**Si**	**11, 19, 22**
F1.12	**Si**	**15**
F1.13	**Si**	**18**

F1.14	**Si**	**22**
F1.15	**Si**	**26, 27, 28, 29**
F1.16	**Si**	**31**
F2	**Si**	**Ver código de openFile y openFileIntegrity**
F3	**Si**	**Ver código closeFile y closeFileIntegrity**
F4	**Si**	**Struct inodo no tiene inodos contenidos (no tiene directorios)**
F5	**Si**	**En openFileIntegrity se llama a checkFile**
F6	**Si**	**11**
F7	**Si**	**12, 21**
F8	**Si**	**01**

Requisitos no funcionales.

Identificador	Se cumple	Pruebas asociadas
NF1	**Si**	**Strlen(inodos[])=48**
NF2	**Si**	**29**
NF3	**Si**	**5 bloques de datos * 2048 bytes**
NF4	**Si**	**Definido como constante**
NF5	**Si**	**No se resetean en mount y unmount**
NF6	**Si**	**00**

NF7	**Si**	**Las estructuras solo tienen los campos necesarios**
NF8	**Si**	**Código de openFile**
NF9	**Si**	**Código de closeFileIntegrity**
NF10	**Si**	**Código de openFileIntegrity**
NF11	**Si**	**Código de closeFile**
NF12	**Si**	**Código de closeFileIntegrity**

3 Batería de Pruebas

Debido a que muchas funciones y pruebas dependen de otras para poder funcionar (integridad, lectura, escritura, etc) se han ordenado las pruebas **en 3 bloques de ejecución**.

Estos bloques están representados en la salida por terminal por el título **TEST**, haciendo referencia a las pruebas que trabajan con un archivo sin integridad; **TEST_INTEGRITY,** a las que trabajan con archivos con integridad y **TEST_LINK**, que trabaja con enlaces simbólicos.

Algunas de las pruebas se usan como pasos intermedios para otras pruebas, como por ejemplo usar el **read** para comprobar el **write** o usar el **lseek** antes de usar el **write**. Todo esto está comentado en el código del test.

En el **procedimiento** indicado en la tabla se indican los pasos en **caso de que la prueba fuera individual**, por ejemplo, una prueba de **lseek** tendrá un procedimiento que implique crear el fichero, abrirlo, etc, sin embargo, la prueba como tal en el código es solo usar la función dado que pruebas anteriores ya han creado y abierto el fichero.

ID	Objetivo	Procedimiento	Entrada	Salida esperada
00	Crear el sistema de ficheros fuera de rango.	Llamar a **mkfs** con más de 600 KB	La función **mkfs** con un parámetro de más de 600 KB	No se crea el sistema
01	Crear el sistema de ficheros	Llamar a **mkfs**	La función **mkfs**	Se crea el sistema

02	Montar el sistema de ficheros	Crear un sistema de ficheros con **mkFS** Montar el sistema de ficheros **mount**	La función **mount**	Se monta el sistema de ficheros
03	Crear un archivo	Crear un archivo con **createFile**	La función **createFile**	Se crea el archivo
04	Abrir el archivo	Crear un archivo con **createFile** Abrir el archivo **openFile**	La función **openFile** sobre el archivo creado	Se abre el archivo
05	Escribir en el archivo	Crear un archivo con **createFile** Abrir el archivo **openFile** Escribir en el archivo **writeFile**	Un buffer con “hola mundo” y la función **writeFile** con un size de 7.	Se escribe en el fichero “hola mu”
06	Leer el archivo con el puntero al final	Crear un archivo con **createFile** Abrir el archivo con **openFile** Escribir en el archivo con **writeFile** Leer el archivo con **readFile**	El archivo después de haber escrito y la función **readFile**	Se leen 0 bytes porque el puntero está al final.
07	Mover el puntero del fichero al inicio	Crear un archivo con **createFile** Abrir el archivo con **createFile** Escribir en el archivo con **writeFile** Mover puntero con **lseek** al inicio	Un archivo con el puntero al final tras escribir y la función **lseek** con parámetro *BEGIN*	El archivo con el puntero al principio
08	Leer el archivo con el puntero al principio	Crear un archivo con **createFile** Abrir el archivo con **openFile** Mover el puntero con **lseek** al inicio	Un archivo con el puntero al principio que tiene escrito “hola mu” y la función **readFile** con *size* 10	Se leen 7 bytes ya que es todo el contenido del archivo

		Leer el archivo con readFile		
09	Mover el puntero del fichero al medio	Crear un archivo con **createFile** Abrir el archivo con **openFile** Mover el puntero con **lseek** usando la posición actual y la variable *offset*.	Un archivo con el puntero al final y **lseek** con parámetro *CUR* y *offset* -4	Se mueve el puntero a la posición 3, que corresponde que la letra “a” de “hola mu”.
10	Leer el archivo con el puntero en medio	Crear un archivo con **createFile** Abrir el archivo con **openFile** Mover el puntero con **lseek** usando la posición actual y la variable *offset*. Leer el archivo con **readFile**	Un archivo con el puntero en la tercera posición y la función **readFile** con un *size* de 2.	Se lee la cadena “a “ del archivo.
11	Sobrescribir una parte del fichero	Crear el archivo con **createFile** Abrir el archivo con **openFile** Escribir en el archivo con **writeFile** Mover el puntero con **lseek** al inicio Escribir en el archivo de nuevo con **writeFile** Comprobar el resultado con **readFile**	El archivo con el puntero en la posición 5, se pasa la posición al principio y se tiene un buffer “SSS”. La función **writeFile** con *size* 2 y la función **readFile** con *size* 3.	Se escribe en el archivo, quedando una cadena “SSla mu” y se lee “SSl”.
12	Escribir en el archivo más de 5 bloques	Crear el archivo con **createFile** Abrir el archivo con **openFile** Crear un buffer con más de 5 bloques	Un buffer con mas de 12000 bytes. La función **writeFile** con *size* 11000	Se escriben todos los bloques de datos del archivo y devuelve 10233 que son los que

		Escribir en el archivo el buffer con **writeFile**		quedaban por escribir.
13	Cerrar el archivo	Crear el archivo con **createFile** Abrir el archivo con **openFile** Cerrar el archivo con **closeFile**	El archivo de las pruebas anteriores abierto	Se cierra el archivo correctamente
14	Escribir en el archivo cerrado	Crear el archivo con **createFile** Abrir el archivo con **openFile** Cerrar el archivo con **closeFile** Escribir en el archivo con **writeFile**	El archivo cerrado anterior	Devuelve -1 ya que no se puede escribir si está cerrado
15	Eliminar el archivo	Crear el archivo con **createFile** Llamar a **removeFile**	El archivo cerrado	Se elimina el archivo
16	Incluir integridad a un fichero	Crear un fichero con **createFile** Llamar a la función **includeIntegrity** con un archivo existente	Se pasa ./test_int.txt a la función **openFile** y después a **includeIntegrity**	Se incluye integridad en el archivo creado
17	Abrir un fichero con integridad con **openFile**	Seleccionar un fichero al que hayamos incluido integridad Abrir con **openFile**	El archivo creado en la prueba anterior y la función **openFile**	Devuelve -2 ya que tiene integridad
18	Comprobar la integridad de un archivo antes de modificarlo	Crear un archivo con integridad Se llama a la función **checkFile**	El archivo de la prueba anterior y la función **checkFile**	**checkFile** devuelve 0 ya que no ha sido modificado
19	Abrir un fichero con **openFileIntegrity**	Crear un archivo con integridad Llamar a la función **openFileIntegrity**	El archivo creado en la prueba 17 y la función **openFileIntegrity**	Se devuelve 0 porque se abre correctamente

20	Comprobar la integridad de un archivo nada más abierto	Abrir archivo con **openFileIntegrity** Llamar a **checkFile**	El archivo de la prueba anterior y a función **checkFile.**	Devuelve -2 ya que no se puede hacer con el archivo abierto
21	Llenar un bloque de datos	Crear archivo con **createFile** Abrir con **openFile** Escribir suficientes bytes con **writeFile**	El archivo anterior, un buffer con 2229 bytes y la función **writeFile** con *size* 2200	Se llena un bloque de datos y se le asigna otro al inodo
22	Desmontar el sistema de ficheros con un fichero abierto	Crear un sistema de ficheros con **mkFs** Montar el sistema de ficheros con **mount** Crear y abrir el archivo con **createFile y openFile** Desmontar el sistema de ficheros con **unmountFS**	La función **unmountFS**	Da error ya que hay un fichero abierto
23	Cerrar un fichero con **closeFileIntegrity**	Crear un archivo con integridad Abrir un archivo con **openFileIntegrity** Cerrar el archivo con **closeFileIntegrity**	El archivo de las pruebas anteriores y la función **closeFileIntegrity**	Se cierra correctamente el archivo
24	Comprobar la integridad de un archivo después de modificarlo	Se modifica un archivo con **writeFile** (test 21) Se cierra el archivo Se comprueba la integridad con **checkFile**	El fichero sobre el que se había escrito cerrado y la función **checkFile**	La integridad es correcta porque se cerró con integridad
25	Eliminar un archivo con integridad	Se crea un archivo con **createFile** Se incluye integridad con **IncludeIntegrity** Se llama la función **removeFile**	El archivo creado anteriormente “/test_int.txt” con integridad incluida.	Se elimina del sistema

26	Crear un enlace simbólico a un fichero que existe	Crear un archivo con **createFile** Abrir el archivo con **openFile** Crear un enlace simbólico con **createLn**	La función **createLn** sobre el fichero creado. ES1 → Fichero	Enlace simbólico creado
27	Crear un enlace simbólico a un fichero que no existe	Crear un enlace simbólico con **createLn**	La función **createLn** sobre el fichero **no** creado. ES1 → …	ERROR
28	Crear un enlace simbólico a otro enlace	Crear el archivo con **createFile** Abrir el archivo con **openFile** Crear un enlace simbólico con **createLn** Crear un enlace simbólico con **createLn** de nuevo	La función **createLn** sobre un enlace creado anteriormente. ES2 → ES1	Enlace simbólico creado
29	Crear un fichero con el nombre muy largo	Crear un archivo con **createFile** Abrir el archivo con **openFile** Crear un enlace simbólico de nombre >32 caracteres	La función **createLn** con el nombre del enlace con más de 32 char .	ERROR
30	Escribir en un fichero accediendo a través de un enlace simbólico	Crear un archivo con **createFile** Abrir el archivo con **openFile** Crear un enlace simbólico con **createLn** Escribir en el archivo con **writeFile**	Hacer **openFile** al enlace y hacer **writeFile** con un buffer “hola mundo” y size de 7.	Escritura realizada
31	Eliminar los enlaces simbólicos	Crear un archivo con **createFile**	La función **removeLn** sobre los enlaces	Enlace simbólico eliminado

		Crear un enlace simbólico con **createLn** Eliminar el enlace simbólico con **removeLn**	creados previamente	
32	Desmontar el sistema de ficheros	Crear un sistema de ficheros con **mkFs** Montar el sistema de ficheros con **mount** Desmontar el sistema de ficheros con **unmount**	La función **unmountFS**	Sistema de ficheros desmontado

4 Conclusión

Uno de los problemas encontrados ha sido en la función **write** que no actualizaba el puntero bien ni el tamaño a escribir en diversos bloques del fichero. Tras varios intentos y diferentes pruebas conseguimos ver los errores y solucionarlos.

En **las funciones de integridad**, pensamos inicialmente en pasar todo el fichero y calcular la integridad para el fichero completo. Sin embargo, por facilidad hemos decidido calcular la integridad por bloques.

También, no sabíamos cómo abordar los **enlaces simbólicos**, si añadir un nuevo campo al *struct* inodo que indicara el descriptor del fichero al que apunta o bien introducir ese descriptor apuntado en el bloque de datos. Optamos por esta última opción.

Personalmente consideramos que es una de las prácticas más útiles hasta ahora ya que nos ayuda a entender como funciona por dentro un sistema de ficheros que utilizamos diariamente.

El código proporcionado por los profesores, así como la ayuda recibida en clases y por correo electrónico ha sido de gran ayuda para la práctica a pesar de la situación actual.