**PRÁCTICA 2:  
 SISTEMA DE FICHEROS**

Diseño de Sistemas Operativos



Pedro de la Torre Diaz;100330636

Grupo 80,

# **Índice**

[**Índice** 1](#_Toc7719658)

[**Introducción a la práctica** 2](#_Toc7719659)

[**Diseño** 2](#_Toc7719660)

[**SuperBloque Diseño:** 4](#_Toc7719661)

[**Inodo Diseño:** 4](#_Toc7719662)

[**Funciones implementadas en la práctica:** 5](#_Toc7719663)

[**mKFS** 5](#_Toc7719664)

[**mountFS** 5](#_Toc7719665)

[**unmountFS** 5](#_Toc7719666)

[**createFile** 5](#_Toc7719667)

[**removeFile** 6](#_Toc7719668)

[**openFile** 6](#_Toc7719669)

[**closeFile** 6](#_Toc7719670)

[**readFile** 6](#_Toc7719671)

[**writeFile** 7](#_Toc7719672)

[**lseekFile** 7](#_Toc7719673)

[**mkDir** 7](#_Toc7719674)

[**rmDir** 7](#_Toc7719675)

[**lsDir** 8](#_Toc7719676)

[**Plan de pruebas** 8](#_Toc7719677)

[**Conclusion** 10](#_Toc7719678)

# **Introducción a la práctica**

El objetivo de esta segunda práctica, se nos pide desarrollar un sistema de ficheros simplificado en lenguaje de programación C (ISO C11) sobre nuestra máquina de Guernica.

Esta práctica tiene el fin de diseñar la arquitectura de un sistema de ficheros. Controlando su estructura como los INodos, supebloque, o descriptores del fichero. Desarrollando los algoritmos necesarios para su funcionamiento.

Debemos justificar el diseño desarrollado y la implementación del mismo.

Crearemos un plan de pruebas para probar la implementación del mismo, para poder validar lo realizado.

# **Diseño**

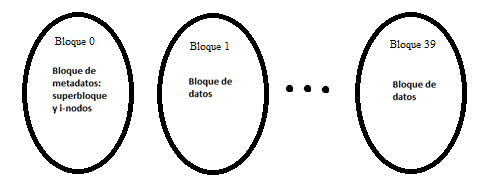
Vamos a comentar el diseño que hemos decidido realizar para la práctica, basándonos en lo dado en el enunciado y en las decisiones tomadas para su correcto funcionamiento, asi como las estructuras referente a datos que hemos implementado teniendo en cuenta los requisitos impuestos en el enunciado.

Utilizamos Linux (en nuestro caso la máquina virtual de Guernika) que está basado en un sistema de ficheros de tipo UNIX, dándole nosotros una seria de cambios y modificaciones.

Se empleará un sistema de ficheros basado en el sistema de ficheros UNIX, pero con modificaciones y optimizaciones que vamos a describir a continuación.

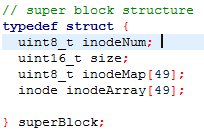
Se nos pide una serie de requisitos no funcionales:

* El número máximo de ficheros en el sistema será 40.
* El nombre de un fichero o directorio solo podrá tener como máximo 32 caracteres.
* El tamaño máximo de un fichero será de 1 bloque.
* Cada bloque del sistema tendrá un tamaño de 2048 bytes. Al igual que los ficheros.
* Los metadatos del sistema persistirán en las operaciones de desmontaje y montaje.
* El sistema de ficheros será usado en discos de 50 KiB a 10 MiB.
* Deberemos aprovechar al máximo los recursos disponibles.
* Debemos aprovechar al máximo los recursos disponibles.



Creamos una serie de estructuras de tipo STRUCT para el correcto funcionamiento de nuestro sistema:

**SUPERBLOQUE:**



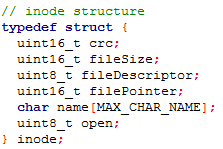
inodeNum🡪 Son los números de INodos; tamaño 1 B.

size🡪 Tamaño; tamaño 2 B;

inodeMao[49] 🡪mapa de los inodos; tamaño 49 B.

inideArray[49] 🡪 Array de inodos; tamaño 1960 B.

**INODO:**



Crc 🡪 crc fichero; tamaño 2 B.

fileSize 🡪tamaño fichero; tamaño 2 B.

fileDescriptor 🡪descripción; tamaño 1 B.

filePointer 🡪puntero; tamaño 2 B.

name 🡪nombre; tamaño 32 B.

open 🡪abierto o cerrado (1/0); tamaño 1 B.

El sistema que hemos creado solo tendrá un bloque de metadatos, que será el primero.

El tamaño del inodo será de 40 B, mientras que el arrayInodos[40] tendrá un tamaño de 40\*49=1960 B.

Este diseño hace que tengamos un padding muy pequeño.

## **SuperBloque Diseño:**

El superbloque sirve para toda la gestión de inodos, que lo realiza a mediante los datos de su propia estructura.

Los datos de su estructura se modifican cada vez que se produce una creación o destrucción de ficheros. Al finalizar, volvamos todos los datos comprobando que no se corrompan. Mediante el desmontaje y montaje de discos.

Tenemos que tener en cuenta que cada fichero ocupa un bloque, por lo que facilita la gestion y no hace falta un mapa de bloques.

## **Inodo Diseño:**

El Inodo sirve para almacenar información, es una estructura de tipo STRUCT.

Gestiona la información que se encuentra en el INodo, y puede gestionar la información dentro del mismo. Esta información también se volcara en disco en las operación es de desmontaje y montaje.

Al ocupar lo mismo que un fichero, no tenemos que gestionar los bloques indirectos, en el caso que fuese necesario. La información de dicha estructura ha sido descrita anteriormente.

# **Funciones implementadas en la práctica:**

## **mKFS**

La función “int mkFS(long deviceSize) “ genera una estructura del sistema de ficheros, dentro de nuestro dispositivo de almacenamiento.

Recibe por parámetros el tamaño del disco a dar formato (B), devuelve 0 si es correcta la ejecución y -1 si es errónea.

El tamaño debe estar comprendido entre 50KiB y 100KiB.

El tamaño lo calculamos en función de los inodos. Inicializamos a 0 todos los inodos.

## **mountFS**

En esta función “int mountFS(void)” en la que una operación del sistema de ficheros es ejecutada por el programa cliente, para poder interactuar con los ficheros, montamos el dispositivo, por lo que asignamos y configuramos las estructuras necesarias para poder utilizar los ficheros.

Primero comprueba que el sistema este integro, primero utilizamos el superbloque.

## **unmountFS**

“int unmountFS(void)” Esta es la última operación del sistema de ficheros, en la que desmontamos el dispositivo, y se liberan todas las estructuras y variables. Devolvemos 0 si es correcto, -1 si es erróneo.

## **createFile**

En la función “int createFile(char \*path)” creamos un nuevo fichero vacío, realizando todos los cambios necesarios en el sistema.

Primero comprobamos que no existe dicho fichero, inicializamos la posición que va a tener nuestro fichero, que será -1 para saber la posición libre dentro de nuestro sistema de bloques, recorremos el array comprobando el nombre, si encontramos dentro del bucle una posición vacía salimos, y en el caso que recorramos todo nuestro bucle y no encontremos posición vacía de devuelve un -2 indicándolo.

Introducimos en el name el nombre del archivo que estamos creando. Lo ponemos a 0 en la variable open de la estructura, cambiamos la descripción a 1, tamaño a 0 y puntero a 0. Así podemos saber que el bloque ya se está utilizando. Y escribimos con write.

## **removeFile**

La función de remove la utilizamos para borrar ficheros. Lo localizamos dentro de nuestro array de estructuras, comprobamos el nombre mediante el path, y lo eliminamos poniendo a -1 sus metadatos, y ponemos a 0 su posición. Realizamos todos los cambios necesarios en el sistema de ficheros para eliminarlo correctamente. Devolvemos 0, en el caso de que no pueda ser borrado devolvemos -1 y si ocurre cualquier otro error devolvemos -2.

## **openFile**

Abrimos un fichero existente dentro de nuestro sistema de ficheros utilizando la path que se le pasa a la función para saber el nombre.

Comprobamos que el archivo exista buscando dentro de nuestro inodos por nombre, comprobamos que no se encuentre en ese momento abierto, y cambiamos abierto=1, puntero=0, y devolvemos el descriptor del fichero mediante return, -1 si no puede ser abierto por que no existe, -2 si ocurre cualquier otro error.

## **closeFile**

Esta función cierra un fichero abierto pasándole el descriptor del mismo, buscamos cogemos el lugar en donde se encuentra y modificamos open=0. Devolvemos 0 si todo salio bien o -1 en caso de error.

## **readFile**

En la función de leer Fichero le pasamos el descriptor, el buffer y el número de bytes. Lo que realizamos es volcar todos los bytes que le indicamos dentro del descriptor del fichero en el buffer que se pasa a la función. En el caso que el fichero no esté abierto, devolvemos -1, porque no podemos realizar la lectura.

Cogemos la posición del fichero, basándonos en el descriptor del mismo, comprobando que esta, el número de bytes que leemos no puede ser inferior a 1, comprobando esto dentro de la lectura del fichero, sino devolvemos -1, en el caso que intentemos leer más que el tamaño del fichero, solo devolvemos desde nos encontramos hasta el final del mismo. Volcamos los datos leídos al buffer, y en el caso que salga la lectura todo bien , devolvemos el número de bytes que hemos leído realmente, en el caso que no leamos nada devolvemos 0.

## **writeFile**

La función de escritura sobre ficheros, modifica tantos bytes como se indiquen. Escribimos los datos del buffer en el descriptor del fichero, indicando los números de bytes, los cuales recivimos por parámetro, Buscamos la posición de ese fichero de la misma manera que la función anterior de lectura, al igual que el control que realizamos sobre los errores. Si se pretende escribir mas bytes que lso que se pueden llegar a escribir, entonces solo realizamos la escritura hasta llenar el fichero. Usamos el buffer para realizar la escritura en el fichero, al finalizar indicamos el nuevo tamaño que tiene nuestro fichero.

Devolvemos mediante return 0 si al ejecution fue correcta y -1 si no lo fue.

## **lseekFile**

Esta función modifica el valor del puntero de posición de un archivo. Desplazamos el puntero de lectura y escritura. Cogemos la posición que tiene el fichero mediante el descriptor que cogemos mediante parámetros. Utilizamos whence como constante de referencia para la operación de modificación del puntero de posición, En el caso de que sea SEEK\_END colocamos el puntero al final del archivo, si es FS\_SEEK\_BEGIN lo ponemos al comienzo, si es FS\_SEEK\_CUR colocamos el puntero donde nos indique el offset, comprobando que no sea negativo o sea mayor que el fichero.

Si la ejecución es correcta, devolvemos 0, en caso contrario -1.

## **mkDir**

Esta función crea un nuevo directorio vacío, pasándole por parámetro la path donde ira dicho directorio.

La path será la ruta del directorio, y devolvemos mediante return un 0 si salió todo bien, -1 en caso de que ya exista el directorio y -2 por cualquier otro error dentro de la creación del directorio.

## **rmDir**

En esta función lo que realizamos en la eliminación de un directorio, el cual pasamos por parámetro mediante la path. Realizando también la eliminación de su contenido.

Devolvemos 0 en el caso de que sea todo correcto, -1 si no podemos leerlo o no existe, y -2 por otro error.

## **lsDir**

Esta función carga los inodos y los nombres de los elementos de un directorio, pasándole por parámetros la path, “inodesDir” que es un array que la función debe completar con los inodos de los elementos del directorio, las entradas no usadas se complementan con -1. Y “namesDir” array con lso nombres de los elementos contenidos en el directorio siguiendo un mismo orden.

Devolvemos la cantidad de elementos presentes en el directorio, -1 si el directorio no puede ser leído, -2 por cualquier otro error.

# **Plan de pruebas**

EL fichero “test.c” contiene nuestras pruebas realizadas, las cuales consideramos necesarias para demostrar el funcionamiento de nuestras funciones y de nuestro sistema.

|  |  |
| --- | --- |
| Descripción | Evalúa la creación del sistema de fichero pasándole el tamaño mínimo de disco. |
| Función | mkFS |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Evalúa el montaje del sistema de ficheros una vez creado. |
| Función | mountFS |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Evalúa la creación de un fichero llamándolo test.txt |
| Función | createFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Evalúa el desmontaje del sistema de ficheros una vez creado. |
| Función | unmountFS |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Evalúa el cierre de un fichero, recibe un descriptor donde hay un fichero para cerrar. |
| Función | closeFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Evalúa la apertura de un fichero, recibiendo el path, en este caso “test.txt” |
| Función | openFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Evalúa la escritura de un testo que se pasa por parámetros, al igual que la posición del fichero, y cantidad que se quiere escribir. |
| Función | writeFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Evalúa la lectura de un fichero, pasándole la cantidad a leer, posición de fichero y buffer. |
| Función | readFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Modifica el valor del puntero pasándole el descriptor, posición inicial, numero de bytes a desplazar. |
| Función | lseekFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Descripción | Elimina el fichero “test.txt” |
| Función | removeFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

# **Conclusion**

Como conclusiones podemos sacar que esta práctica nos ha servido bastante para saber mejor como es el funcionamiento de los ficheros dentro de nuestro sistema y poder comprobar como es la implementación del mismo, así como su funcionamiento.

Hemos mejorado conceptos muy importantes referentes a desmontajes y montajes. Hemos aprendido mucho referente a la implementación de la teoría de la asignatura, asi como la resolución de problemas respecto al sistema de ficheros.