**PRÁCTICA 2: Planificación de procesos**

Diseño de Sistemas Operativos



Luis Cabrero García

100330021

Javier Bezares Sanfelip

100317428

Jorge Hevia Moreno

100317565

Grupo 80, 3º G. Ingeniería Informática

# Índice

[Índice 1](#_Toc481094455)

[Introducción 2](#_Toc481094456)

[Diseño del Sistema de Ficheros 3](#_Toc481094457)

[Estructura de SuperBloque: 3](#_Toc481094458)

[Estructura de iNodo: 4](#_Toc481094459)

[Diseño del SuperBloque. 4](#_Toc481094460)

[Diseño del inodo. 4](#_Toc481094461)

[Descripción de las funcionalidades implementadas 5](#_Toc481094462)

[mKFS 5](#_Toc481094463)

[mountFS 5](#_Toc481094464)

[unmountFS 5](#_Toc481094465)

[createFile 5](#_Toc481094466)

[removeFile 6](#_Toc481094467)

[openFile 6](#_Toc481094468)

[closeFile 6](#_Toc481094469)

[readFile 6](#_Toc481094470)

[writeFile 6](#_Toc481094471)

[lseekFile 7](#_Toc481094472)

[checkFS 7](#_Toc481094473)

[checkFile 7](#_Toc481094474)

[Diseño de plan de pruebas 8](#_Toc481094475)

[Flujos de programa con casos de éxito 8](#_Toc481094476)

[Flujos de programa con casos de error 10](#_Toc481094477)

[Conclusiones. 11](#_Toc481094478)

# Introducción

En esta segunda práctica de la asignatura de Diseño de Sistemas Operativos, debíamos ser capaces de diseñar e implementar un sistema de ficheros simulado, para un dispositivo pequeño (de entre 50 KiB y 100 KiB), de tal forma que se puedan crear archivos, leerlos, escribir sobre ellos, etc. Este sistema de ficheros estaría sujeto a una serie de especificaciones de diseño que se deben cumplir, tamaño de bloque, tamaño máximo de archivo, etc. El diseño de este sistema de ficheros estará limitado al uso de archivos, excluyendo de nuestro sistema los directorios.

La estructura del documento es la siguiente: primero plantearemos una solución al diseño pedido, de tal forma que explicaremos las decisiones de diseño que tomemos y las justificaremos, posteriormente, realizaremos una explicación de las funcionalidades implementadas teniendo en cuenta el diseño, una sección del diseño del plan de pruebas y, por último, una sección de conclusiones, en la que se describen problemas encontrados y sus soluciones.

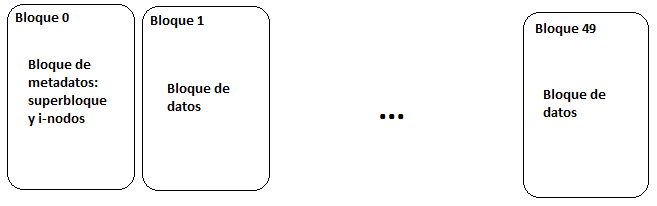
# Diseño del Sistema de Ficheros

En esta sección se comentará el diseño en detalle que utilizaremos como base para la implementación, incluyendo las decisiones de diseño que hemos tomado ante situaciones que no estaban definidas en el enunciado de la práctica y las estructuras de datos que hemos utilizado para guardar los metadatos. Explicaremos también la gestión que realizamos sobre los datos y analizaremos los requisitos arquitectónicos justificando nuestro diseño y explicando las optimizaciones que hemos llevado a cabo.

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

Los requisitos arquitectónicos son los siguientes:

1. El **máximo número de ficheros** en el sistema operativo es de 64.
2. La longitud máxima de un nombre de fichero es de 32 caracteres (32 bytes).
3. El **tamaño máximo de un fichero** y el **tamaño de bloque** es de 2KB. Almacenaremos un fichero por cada bloque ya que ocupan el mismo espacio en caso de que el fichero esté lleno.
4. Los metadatos deben persistir entre operaciones de montaje y desmontaje.
5. El sistema de ficheros será usado en **discos** de entre 50KB y 100KB. Dado que el tamaño de bloque es de 2KB, el número máximo de bloques que puede haber en 100KB es 50. Teniendo en cuenta esto y que al menos un bloque será utilizado para meta datos, el **número máximo de ficheros es 49**.



Teniendo en cuenta estos requisitos arquitectónicos, las estructuras que utilizaremos para almacenar los metadatos serán las siguientes:

## Estructura de SuperBloque:

|  |  |  |  |
| --- | --- | --- | --- |
| Tipo de atributo | Atributo | Descripción | Tamaño |
| uint8\_t | iNodeNum | Número de inodos. | 1 byte |
| uint16\_t | size | Tamaño del disco. | 2 bytes |
| uint8\_t | InodeMap | Mapa de inodos indicando si están siendo usados (1) o no (0). | 49 bytes |
| Inode | inodeArray | Array de estructuras inode en el que se almacenan los datos de cada inodo. | 1960 bytes |

## Estructura de iNodo:

|  |  |  |  |
| --- | --- | --- | --- |
| Tipo de atributo | Atributo | Descripción | Tamaño |
| uint16\_t | crc | Almacena el crc de un fichero. | 2 bytes |
| uint16\_t | fileSize | Tamaño de un fichero. | 2 bytes |
| uint8\_t | fileDescriptor | Descriptor de fichero. | 1 byte |
| uint16\_t | filePointer | Puntero de lectura/escritura. | 2 bytes |
| char\* | name | Nombre de archivo. | 32 bytes |
| uint8\_t | open | Indica si el archivo está abierto (1) o cerrado (0). | 1 byte |

Nuestro sistema de ficheros únicamente contará con un bloque destinado para el almacenamiento de metadatos:

* En total, la estructura de inodo ocupa 40 bytes.
* El array de inodos, ocupa 40\*49=1960 bytes.
* Por tanto, el espacio total necesario para almacenar las dos estructuras de metadatos es de 2012 bytes. Siendo el tamaño de bloque de 2048 bytes, entonces tenemos un padding de 36 bytes.

Esto es una optimización muy importante en cuanto al uso de variables para que los metadatos puedan ser almacenados en un solo bloque (el bloque 0). El resto de bloques, del 1 al 49, son bloques de datos.

## Diseño del SuperBloque.

El superbloque es una estructura que sirve para almacenar información sobre el sistema de ficheros. Está destinado a gestionar los inodos y el tamaño del disco que se creará, y comprobar la integridad de los datos.

Los datos almacenados en la estructura de superbloque se actualizarán cada vez que se produzca una creación o destrucción de un fichero. Cuando se desmonte el sistema de ficheros, deberemos almacenar en disco toda la información que lo contenga para satisfacer el requisito que indica que dicha información debe persistir en operaciones de montaje y desmontaje.

El sistema de ficheros no contiene un mapa de bloques en el superbloque, dado que no es necesario, si un bloque está ocupado se indicará mediante el mapa de inodos, dado que un fichero ocupa un bloque.

## Diseño del inodo.

El inodo es una estructura para almacenar información sobre un fichero. Controla el estado de dicho fichero y contiene toda la información necesaria para realizar una operación a nivel de fichero. Los datos de inodos también se escribirán en disco cuando se realice una operación de desmontaje dado que son metadatos que deben persistir.

Al no tener que implementar directorios en nuestro sistema de ficheros, no tendremos que preocuparnos por el uso de bloques indirectos. También almacenaremos la información sobre si un fichero está abierto o cerrado en operaciones de desmontaje, datos que no deberían persistir dado que cuando se desmonta no deberían conservarse esos datos, pero esto implicaría complicaciones en la implementación y la necesidad de utilización de otras estructuras auxiliares que nos implicarían utilizar más tamaño para metadatos, por lo que hemos preferido simplificar y optar por la optimización del sistema.

# Descripción de las funcionalidades implementadas

## mKFS

Esta función genera la estructura de sistema de ficheros en un disco, los metadatos necesarios para gestionar el sistema de archivos y desmonta el disco. Para ello primero se comprueba que el tamaño del disco esté entre 50KiB y 100KiB. Hallamos el número de inodos en función del tamaño del dispositivo redondeando un bloque hacia arriba para obtener el número de bloques necesario para almacenamiento. El tamaño se recalcula en función del número de inodos. Se inicializan todas las variables de los inodos a cero y se desmonta el disco. Si no ha sucedido ningún error entonces retornamos 0 (éxito).

## mountFS

Esta función monta la estructura que hemos generado en el mKFS. Lo primero que hacemos es comprobar la integridad de los datos de todos los bloques (todo el sistema de ficheros). Una vez que comprueba que no está corrupto el sistema de archivos, leemos el primer bloque del disco y lo almacenamos en la estructura de superbloque, indicando que estamos guardando en el bloque 0. Realiza comprobación de errores y si todo va bien retorna 0.

## unmountFS

Escribe los datos de la estructura de superbloque en el disco. Realizamos una comprobación de errores y retornamos 0 si no hay ningún error.

## createFile

Esta función crea un archivo en el sistema de ficheros comprobando previamente que ese fichero no exista en el sistema. Generamos una variable que inicializamos a -1 que indicará la posición en la que vamos a encontrar una posición vacía en nuestros bloques. Recorremos con un bucle el array de inodos comprobando que el nombre de nuestro nuevo archivo no coincida con los nombres que vamos encontrando en el sistema de archivos. Si hay espacio en disco (hay una posición vacía) salimos, y si hemos recorrido todo el bucle y no hemos encontrado una posición vacía entonces retornamos -2. Una vez se han comprobado todos los posibles errores copiamos el nombre del archivo en la variable name del inodo, ponemos el open a 0 indicando que el archivo no está abierto. Ponemos el descriptor de fichero a la posición del bloque sumando 1 (porque el bloque 0 es el bloque de metadatos), indicamos que el tamaño es 0, porque no contiene nada, el puntero lo ponemos en 0 y en el mapa de inodos, indicamos que el bloque está siendo utilizado. Hacemos un block write, escribiendo en la imagen del dispositivo en el bloque que hemos encontrado espacio ponemos un buffer vacío de tamaño 2048 bytes.

## removeFile

Esta función sirve para borrar un archivo del sistema de ficheros. Buscamos en el array de inodos, y comprobamos el nombre del archivo con el nombre que haya almacenado en el inodo. Si coincide y el archivo está cerrado, entonces reinicializa todos los atributos de metadatos a -1, ponemos a 0 su posición en el mapa de inodos y el nombre es eliminado mediante memset(). Realizamos control de errores y si todo va bien retornamos 0.

## openFile

Se encarga de abrir un archivo dado su nombre. Primero comprobamos la integridad del archivo, después buscamos en los inodos uno que contenga el mismo nombre que el nombre pasado por parámetro, comprueba que no esté ya abierto, cambia el atributo de abierto a 1, el puntero de fichero lo pone a 0 (el inicio) y posteriormente retornamos el descriptor de fichero.

## closeFile

Se encarga de cerrar un fichero dado su descriptor de fichero o número de bloque. En primer lugar, se obtiene la posición en la que está el fichero y en su inodo ponemos el atributo open a 0. Retornamos 0 si todo ha ido bien.

## readFile

Se encarga de realizar un volcado del contenido del fichero identificado por el descriptor de fichero pasado por parámetro, en un buffer que se le pasa por parámetro y de una cantidad de bytes indicados. Si el fichero no se encuentra abierto, no podemos leer y retornamos -1. Primero obtenemos la posición en la que se encuentra el fichero identificado por ese descriptor de fichero y comprobamos que realmente existe, retornando -1 en caso contrario. Realizamos un control de errores sobre los parámetros recibidos, por ejemplo, el número de bytes a leer no puede ser un número igual a 0 o negativo, y si tratamos de leer en el final del fichero, retornamos 0 (no estaríamos leyendo absolutamente nada). Si el puntero de fichero en el estado en el que se encuentra sumándole los bytes que deseamos leer es mayor que el tamaño del fichero, entonces únicamente leeremos la diferencia entre el tamaño del fichero y la posición actual del fichero. Leemos el fichero y lo volcamos en el buffer mediante un block read y establecemos el puntero de fichero sumándole el número de bytes que finalmente se han leído. Si todo ha ido bien, retornamos el número de bytes realmente leidos.

## writeFile

Esta función sirve para escribir el contenido de un buffer pasado por parámetro en un fichero identificado por su descriptor de fichero en una cantidad de número de bytes pasada por parámetro. Si el fichero no está abierto retornamos -1. Obtenemos la posición en la que se encuentra el fichero realizando los mismos controles de error que en la función anterior. Si el número de bytes que pretendemos escribir supera a la diferencia que existe entre el tamaño máximo del archivo y el puntero de lectura y escritura, entonces solamente escribiremos dicha diferencia hasta llenar el archivo. Lo primero que hacemos es leer el bloque volcándolo en un buffer auxiliar y posteriormente concatenamos lo que había en el bloque (en el fichero) con lo que queríamos escribir. Cuando tenemos un bloque con todo el nuevo contenido del fichero, hacemos un block write para guardarlo en el mismo bloque, modificando la integridad del fichero. Indicamos el nuevo tamaño del fichero y retornamos el número de bytes que finalmente consiguieron ser escritos.

## lseekFile

Se encarga de desplazar el puntero de fichero de lectura y escritura dependiendo de los parámetros “whence” y “offset”. Primero obtenemos la posición en la que se encuentra el fichero dado su descriptor de fichero, y realizamos el mismo control de errores sobre la posición que en read y write. Si se trata de SEEK\_END, entonces ponemos el puntero de fichero al final del archivo y retornamos 0. Si se trata de SEEK\_BEGIN, ponemos el puntero de fichero al comienzo del fichero y retornamos 0. En caso contrario, si se trata de SEEK\_CUR, si el offset es negativo o es mayor que el tamaño del archivo retornamos error, si no, establecemos el puntero de fichero a la posición que indique el offset y retornamos 0. En caso contrario, se trata de un error y retornamos -1.

## checkFS

Se encarga de verificar la integridad de todo el sistema de ficheros, es decir, de todos los bloques en uso. En primer lugar, se recorren todos los inodos y en cada inodo se comprueba la integridad del bloque, si no hay ningún fallo se devuelve un 0 y si se produce algún fallo se devuelve -1.

## checkFile

Esta función se encarga de comprobar la integridad de un fichero de nuestro sistema de ficheros. Primero buscamos el nombre del archivo que queremos verificar y después realizamos la operación sobre ese archivo. En la función anterior se realiza la verificación sobre todos los ficheros y en esta sobre un solo fichero.

# Diseño de plan de pruebas

El diseño del plan de pruebas se realizaba en el fichero test.c, implementando una serie de pruebas que consideramos necesarias y suficientes para demostrar el correcto funcionamiento de nuestro sistema de ficheros y la consistencia entre funciones. Estas pruebas se basan a nivel general en montar y desmontar el sistema de ficheros, crearlo, crear un fichero, abrirlo, leerlo, escribir en él, cerrarlo y borrarlo. Lo que hacemos en estas pruebas, es introducir unos parámetros a nuestras funciones, suponiendo un retorno determinado y comprobando que el retorno obtenido coincide con el retorno esperado. También hacemos pruebas para probar determinados flujos de programa en los que nuestras funciones retornan errores, como por ejemplo en el caso en el que se pretende escribir en un archivo determinado antes de haberlo abierto. Las pruebas como hemos comentado anteriormente se dividen en el análisis de los casos de error y el análisis de los casos de éxito, comparando los resultados.

Utilizaremos una plantilla basada en una tabla con el identificador de la prueba, la descripción de la prueba, la función que se prueba, resultado esperado y resultado obtenido.

## Flujos de programa con casos de éxito

|  |  |
| --- | --- |
| Identificador de la prueba | CP-01 |
| Descripción de la prueba | La prueba evalúa la creación del sistema de fichero pasándole el tamaño mínimo de disco. |
| Función que se prueba | mkFS |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-02 |
| Descripción de la prueba | La prueba evalúa el montaje del sistema de ficheros una vez se ha creado el sistema de ficheros. |
| Función que se prueba | mountFS |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-03 |
| Descripción de la prueba | La prueba evalúa la creación de ficheros creando un fichero test.txt. |
| Función que se prueba | createFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-04 |
| Descripción de la prueba | La prueba evalúa la apertura del fichero test.txt creado anteriormente. |
| Función que se prueba | openFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-05 |
| Descripción de la prueba | La prueba evalúa la escritura de “hola mundo” en el fichero abierto anteriormente. |
| Función que se prueba | writeFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-06 |
| Descripción de la prueba | La prueba evalúa la lectura de “hola mundo.” en el fichero abierto anteriormente. |
| Función que se prueba | writeFile |
| Resultado esperado | Distinto de -1 |
| Resultado obtenido | Distinto de -1 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-07 |
| Descripción de la prueba | La prueba evalúa la lectura de “hola mundo.” en el fichero abierto anteriormente. |
| Función que se prueba | readFile |
| Resultado esperado | Distinto de -1 |
| Resultado obtenido | Distinto de -1 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-08 |
| Descripción de la prueba | La prueba evalúa el movimiento del puntero de lectura y escritura al comienzo del fichero. |
| Función que se prueba | lseekFile |
| Resultado esperado | Distinto de -1 |
| Resultado obtenido | Distinto de -1 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-09 |
| Descripción de la prueba | La prueba evalúa el cerrar el fichero abierto previamente. |
| Función que se prueba | closeFile |
| Resultado esperado | Distinto de -1 |
| Resultado obtenido | Distinto de -1 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-10 |
| Descripción de la prueba | La prueba evalúa el borrado del archivo test.txt |
| Función que se prueba | removeFile |
| Resultado esperado | Distinto de -1 y -2 |
| Resultado obtenido | Distinto de -1 y -2 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-11 |
| Descripción de la prueba | La prueba evalúa el desmontaje del sistema de ficheros. |
| Función que se prueba | unmountFile |
| Resultado esperado | 0 |
| Resultado obtenido | 0 |

## Flujos de programa con casos de error

|  |  |
| --- | --- |
| Identificador de la prueba | CP-12 |
| Descripción de la prueba | La prueba evalúa la creación del sistema de fichero pasándole un tamaño demasiado pequeño. |
| Función que se prueba | mkFS |
| Resultado esperado | -1 |
| Resultado obtenido | -1 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-13 |
| Descripción de la prueba | La prueba evalúa la creación del sistema de fichero pasándole un tamaño demasiado grande. |
| Función que se prueba | mkFS |
| Resultado esperado | -1 |
| Resultado obtenido | -1 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-14 |
| Descripción de la prueba | La prueba evalúa la creación de un fichero por duplicado. |
| Función que se prueba | createFile |
| Resultado esperado | -1 |
| Resultado obtenido | -1 |

|  |  |
| --- | --- |
| Identificador de la prueba | CP-15 |
| Descripción de la prueba | La prueba evalúa la creación de un fichero cuando ya no queda más espacio. |
| Función que se prueba | createFile |
| Resultado esperado | -2 |
| Resultado obtenido | -2 |

# Conclusiones.

Consideramos que la práctica nos ha ayudado bastante a aprender el funcionamiento del sistema de ficheros y ver una implementación aproximada de las funciones de tratamiento de ficheros y del sistema. Hemos aprendido conceptos como el de montaje, desmontaje, hemos visto temas de integridad de datos y consideramos que la práctica ha sido un buen aprendizaje y que estaban claros los objetivos a alcanzar gracias a una estructura clara de requisitos.

Hemos tenido varios problemas:

* A la hora de intentar implementar sobretodo las funciones de comprobación de integridad de los datos, nos ha costado bastante entender su funcionamiento y que debíamos hacer en parte también porque consideramos que faltaban comentarios en el código base de la parte de crc.
* A la hora de optimizar el diseño nos ha costado bastante entender que podíamos utilizar variables del tipo “uint” para expresar los mismos valores enteros, pero optimizando al máximo el espacio para que entrasen todos los metadatos en un solo bloque.
* No entendíamos bien las estructuras necesarias que teníamos que implementar en metadata.h para que nuestro filesystem dispusiese de la información necesaria. Y hemos tenido que recurrir a bibliografía para recordar las estructuras de inodos y todo el tema de teoría asociado.

Por último, consideramos que esta práctica nos ha ayudado a entender los ejercicios sobre ficheros y que nos ayudará al fin y al cabo para afrontar el examen final de la asignatura.