PRACTICA 2 – SISTEMA DE FICHEROS

DISEÑO DE SISTEMAS OPERATIVOS



ADRIÁN PRIETO CAZALLAS G80 100330520

MIGUEL ROZAS RIQUELME G80 100330321

RAZVAN ALEXANDRU TANASE G80 100330641

**INDICE**

[1. INTRODUCCIÓN 2](#_Toc481095059)

[2. DISEÑO DETALLADO DEL SISTEMA DE FICHEROS 3](#_Toc481095060)

[2.1. mkFS 4](#_Toc481095061)

[2.2. mountFS/unmountFS 4](#_Toc481095062)

[2.3. createFile/removeFile 4](#_Toc481095063)

[2.4. openFile/closeFile 5](#_Toc481095064)

[2.5. readFile/writeFile 5](#_Toc481095065)

[2.6. lseekFile 5](#_Toc481095066)

[2.7. checkFS 6](#_Toc481095067)

[2.8. checkFile 6](#_Toc481095068)

[2.9. posFD 6](#_Toc481095069)

[3. DESCRIPCIÓN DE ALTO NIVEL DE LA FUNCIONALIDAD PRINCIPAL 7](#_Toc481095070)

[4. PRUEBAS 8](#_Toc481095071)

[5. CONCLUSIÓN 9](#_Toc481095072)

# **INTRODUCCIÓN**

En esta práctica, debemos diseñar e implementar un sistema de ficheros para manejar un dispositivo de almacenamiento virtual (disk.dat). Para ello, se generarán tanto la arquitectura del sistema de ficheros, sus estructuras de control (p. ej. i-nodos, superbloque o descriptores de fichero) y los algoritmos necesarios para asegurar el cumplimiento de todos los requisitos y características especificados en el enunciado de la práctica, y después se implementará usando el lenguaje de programación C11.

Un sistema de ficheros es un elemento del Sistema Operativo cuyo objetivo es incorporar una funcionalidad añadida para facilitar el manejo del almacenamiento secundario, lo cual permitirá tanto para reducir el tiempo que tarda el sistema en coger información de la memoria física, reduciendo por tanto el tiempo de ejecución; como para reducir el tamaño de archivos y ficheros en memoria, mejorando por tanto la eficiencia de capacidad de disco. Para que pueda cumplir todas estas funciones, deberá cumplir los siguientes puntos:

* Ser independiente del disco duro, es decir, no pertenecer al mismo y que sirva para la gestión de cualquier disco duro.
* Ofrecer una visión lógica unificada para usuarios y programas (incluyendo el propio sistema operativo).
* Ser suficientemente simple pero completa como para valer para la mayoría: ficheros, directorios, etc.

# **DISEÑO DETALLADO DEL SISTEMA DE FICHEROS**

Un sistema de ficheros está compuesto principalmente de dos tipos de estructuras: metadatos, que contiene información acerca de la distribución de los bloques en memoria secundaria (superbloque, mapa de i-nodos, mapa de bloques…); y datos, que contendrán la información propia de un fichero o directorio.

Es importante que el diseño del sistema se realice buscando minimizar el impacto en el rendimiento y en el espacio para metadatos, es decir, que los metadatos ocupen el mínimo espacio posible para maximizar el espacio posible para datos.

El sistema de ficheros se distribuirá de la siguiente forma:

1 bloque n bloques 1 bloque/i-nodo n bloques

|  |  |  |  |
| --- | --- | --- | --- |
| Bloque del superbloque | Mapa de  i-nodos | Bloques de los  i-nodos | Bloques de datos |

* Superbloque: El superbloque contiene la información de:
  1. Tamaño del sistema de archivos: 100352 Bytes (49[[1]](#footnote-1) ficheros \* 2048 Bytes).
  2. Tamaño de cada bloque: 2048 Bytes.
  3. Tamaño del mapa de i-nodos: array de enteros de tamaño 49 Bytes.
  4. Número de i-nodo.
* Mapa de i-nodos: esta lista tiene una casilla (entrada) por cada fichero que se guarde en el sistema, donde se guarda la descripción del fichero.
* Bloques de i-nodos: contiene la información de:
  1. Nombre del fichero: array de caracteres de como máximo 32 Bytes.
  2. Tamaño actual del fichero: 4 Bytes
  3. Número de bloque: 1 Byte
  4. Descriptor del fichero: 1 Byte
  5. Estado del fichero: 1 Byte

Donde Superbloque, mapa de i-nodos y bloques de i-nodos pertenecen a los metadatos, mientras que los datos estarán contenidos en los bloques de datos. Todos estos datos podemos encontrarlos en **include/metadata.h** donde están definidas todas las variables, estructuras, ... del diseño de nuestro sistema de ficheros.

Este diseño y los pesos de cada elemento en Bytes son así porque el objetivo es que los metadatos ocupen el mínimo espacio posible para poder tener así más espacio para los datos de los ficheros. Para ello hemos usado los tipos especiales de variable (uint8\_t p.ej) para reducir al mínimo posible de espacio que necesita una variable para cumplir su función en los metadatos.

Además de las distintas estructuras para montar la arquitectura del sistema de ficheros, se necesita de funciones para poder manejar dicho sistema, tanto a nivel del propio Sistema de Ficheros como de creación y tratamiento de ficheros como de control de integridad (muchos métodos hacen las mismas funcionalidades que las llamadas al sistema de UNIX (creat, open, read, write…)). Las funciones que creemos necesarias para nuestro sistema de ficheros son las siguientes.

## mkFS

mkFS (long deviceSize): Se genera la estructura del sistema de ficheros que hemos diseñado anteriormente, es decir, se generan los metadatos necesarios para inicializar nuestro Sistema de Ficheros y se ponen a 0. Con esto “limpiamos” (en realidad no es así, ya que los datos siguen con sus datos anteriores, solo se tocan los metadatos) el disco y lo preparamos para que empiece a funcionar.

Para ello lo primero que hemos de comprobar es que el tamaño del disco esté dentro de los valores permitidos, en este caso entre 50KB y 100KB. Después generaremos el superbloque y el mapa de i-nodos (inicializando cada i-nodo a 0 con un bucle), y finalmente hacemos un desmontaje de los metadatos del disco (unmountFS()).

## mountFS/unmountFS

mountFS (void): Esta función monta los metadatos del dispositivo simulado –disk.dat– dentro del Sistema de Ficheros que hemos creado antes, por lo que tiene que asignar y configurar todas las estructuras y variables necesarias para utilizar el sistema de ficheros.

unmountFS(void): Al contrario que mount, unmount se encargará de desmontar los metadatos del dispositivo simulado y liberar las estructuras que habíamos usado en el Sistema de Ficheros.

## createFile/removeFile

createFile(char \*fileName): Esta función actuará como llamada al sistema y se encargará de realizar todas las acciones necesarias para crear un fichero nuevo vacío.

Primero comprobaremos si existe un fichero con el nombre que se ha introducido como entrada (si es así se devuelve -1). En caso de que no sea así, entonces reservamos un bloque que esté libre, y escribiremos en ese bloque el nombre del fichero, su tamaño (de momento 0) y su descriptor de fichero (al no estar abierto el fichero, 0) y se devuelve que ha funcionado correctamente.

removeFile(char \*fileName): Esta función actuará como llamada al sistema y se encargará de borrar un fichero existente en memoria secundaria.

## openFile/closeFile

OpenFile(char \*fileName): Esta función actuará como llamada al sistema y se encargará de abrir un fichero, es decir, darle un descriptor de fichero y traerlo (en caso necesario) de memoria secundaria.

Lo primero que hacemos es mirar si el fichero que entra como entrada existe en el sistema de ficheros (recorremos el mapa de i-nodos buscando el fileName). Si no lo encuentra entonces devolvemos que ese fichero no existe. Si lo encontramos, entonces le damos un descriptor de fichero y su bloque de memoria secundaria pasa a estar libre (hemos cogido el bloque).

closeFile(int fileDescriptor): Esta función actuará como llamada al sistema y se encargará de cerrar un fichero.

Primero conseguimos su fd, comprobamos que ese fd es correcto y cambia el estado del fichero a cerrado, cambiando su fd a 0.

## readFile/writeFile

readFile(int fileDescriptor, void \*buffer, int numBytes): Esta función actuará como llamada al sistema y se encargará de leer un fichero, es decir, coger los bytes que tenía el fichero y pasarlos a un buffer.

Para ello debemos conseguir primero el fd, y después buscaremos en memoria y haremos un bread (comprobando antes que los parámetros de entrada son correctos), almacenando lo que había en el bloque en un buffer, y devolveremos el número de bytes leídos en la función.

writeFile(int fileDescriptor, void \*buffer, int numBytes): Esta función actuará como llamada al sistema y se encargará de escribir un fichero, es decir, coger los bytes de un fichero o de un buffer e introducirlos en el mismo fichero u otro.

Para ello debemos conseguir primero el fd, y después buscaremos en memoria el bloque que le corresponde y haremos un bwrite (comprobando antes que los parámetros de entrada son correctos), almacenando lo que había en el buffer en el bloque determinado anteriormente, y devolveremos el número de bytes leídos en la función.

## lseekFile

lseekFile(int fileDescriptor, int whence, long offset): modifica el valor del puntero de posición de un fichero.

Para ello debemos conseguir primero el fd, y después posicionaremos el puntero donde nos indique la variable whence, ya sea en FS\_SEEK\_END, FS\_SEEK\_BEGIN o FS\_SEEK\_CUR, y después devolveremos que la función se ha realizado correctamente.

## checkFS

checkFS(void): verifica la integridad de los metadatos del sistema de ficheros.

## checkFile

checkFile(char \*fileName): verifica la integridad de los datos asociados a un fichero especificado.

## posFD

posFD(int fd): este método lo hemos introducido en auxiliary.h. Este método sirve para buscar un fichero abierto por su descriptor. Lo usamos principalmente en las funciones de interacción de ficheros salvo el open, porque nos optimiza la búsqueda de esos ficheros y, por lo tanto, el Sistema de Ficheros.

Hemos decidido añadir esta función auxiliar, con la finalidad de facilitar la búsqueda de ficheros, por su descriptor de ficheros. Hemos utilizado esta función exactamente en las funciones del sistema: closeFile, readFile, writeFile y lseekFile.

La función que tiene en todas estas funciones es buscar el fichero, en el momento que el fichero no se encuentre, devolverá un error. Su función exactamente en la función closeFile es identificar el descriptor de fichero. De manera que cuando identifique el descriptor si el estado de éste, es CLOSED, retornará un error, ya que no se puede cerrar un fichero, que ya está cerrado. En el caso de que el estado del fichero a cerrar sea correcto ‘OPENED’ éste se cerrará y su descriptor de fichero se pondrá a cero.

# **DESCRIPCIÓN DE ALTO NIVEL DE LA FUNCIONALIDAD PRINCIPAL**

Para poder usar el Sistema de ficheros, primero deberemos hacer un mkfs para inicializarlo y preparar las estructuras. Después, se montarán los metadatos del disco virtual con el mountFS(). Posteriormente ya se podrán realizar todas las funcionalidades del sistema de ficheros, ya sea para gestión de ficheros como interacción con ficheros. Siempre que se haga una lectura de bloques del disco, se cargará el bloque en memoria, y luego, cuando se escriba de vuelta el fichero, se modificarán tanto los datos del bloque donde se encontraba el fichero como los metadatos del sistema.

También se podrán realizar controles de integridad del disco, es decir, comprobar si algún dato o metadato del sistema se ha visto corrompido ya sea por un error en la escritura como una alteración externa. Para ello usará las funciones CRC, que son funciones de tipo Hash cuya principal función es realizar una modificación en el nombre del fichero y sus datos para protegerlos. Si se desea comprobar la integridad de un metadato o un fichero, se realizará el CRC del fichero y se comprobará con el CRC de ese fichero que tiene guardado el sistema. Si ambas cadenas no son iguales, querrá decir que el fichero o metadato ha sido corrompido. Por último, cuando se vaya a dejar de utilizar el Sistema de ficheros, se realizará un unmount() para guardar los metadatos en el disco

# **PRUEBAS**

|  |  |  |  |
| --- | --- | --- | --- |
| PRUEBA REALIZADA | DESCRIPCION | RESULTADO ESPERADO | RESULTADO OBTENIDO |
| mkFS – TEST | La prueba que vamos a realizar, para este bloque de pruebas. Es una prueba básica en la que el resultado será el esperado cuando la función retorne el valor esperado. Si no devuelve (por lo general, return 0) la prueba fallará. | SUCCESS | **SUCCESS** |
| mountFS – TEST | SUCCESS | FAILED |
| unmountFS – TEST | SUCCESS | FAILED |
| createFile – TEST | SUCCESS | **SUCCESS** |
| RemoveFile – TEST | SUCCESS | FAILED |
| openFile – TEST | SUCCESS | **SUCCESS** |
| closeFile – TEST | SUCCESS | **SUCCESS** |
| readFile – TEST | SUCCESS | **SUCCESS** |
| writeFile – TEST | SUCCESS | **SUCCESS** |
| lseekFile – TEST | SUCCESS | **SUCCESS** |
| checkFS – TEST | SUCCESS | FAILED |
| checkFile – TEST | SUCCESS | FAILED |

|  |  |  |  |
| --- | --- | --- | --- |
| PRUEBA REALIZADA | DESCRIPCION | RESULTADO ESPERADO | RESULTADO OBTENIDO |
| openFile + closeFile | Se van a intentar combinar las funciones openFile y closeFile. De tal manera que nada más se abra el fichero, éste se cerrará, pasando su estado “openF” a CLOSED. | SUCCESS | **SUCCESS** |
| createFile + removeFile | Se combinarán las funciones createFile y removeFile. De tal manera que nada más se cree el fichero, éste se eliminará. | SUCCESS | FAILED |
| readFile + writeFile | Se combinarán las funciones readFile y writeFile. De tal manera que nada más se lea el fichero, éste se escribirá. | SUCCESS | FAILED |

# **CONCLUSIÓN**

A la hora de realizar la práctica, nos hemos encontrado con múltiples problemas, tanto a la hora de diseñar el sistema de ficheros como al implementarlo, debido en cierta medida a la dificultad de la práctica, que era bastante alto, pero principalmente porque estos conocimientos son nuevos para nosotros, por lo que ha habido aspectos a nivel teórico que no hemos podido solventar, provocando que haya elementos cuyas justificaciones sean escuetas o incluso incompletas.

No fue hasta una semana antes de la entrega que nos dimos cuenta gracias a los profesores que había elementos que se podían omitir de un sistema de ficheros real de un sistema operativo, y que los métodos necesitaban de ciertas estructuras para poder funcionar.

Queremos destacar los 3 miembros del equipo, que en NINGÚN momento ha sido culpa, de ninguno de los dos profesores de prácticas, Manuel y Saúl. Es más, han sido de gran ayuda en la práctica. También queremos decir que sin la clase de dudas de la práctica del 21 de abril no sabemos que habríamos entregado hoy. Por la cantidad de trabajo acumulado en todas las asignaturas y el bajo nivel de aprendizaje teórico en esta asignatura, nos hemos encontrado con menos de una semana para hacer la práctica, mezclada con más trabajos de otras asignaturas y la dificultad de juntarnos los 3 compañeros al pertenecer a distintas especialidades de la carrera y no coincidir en horarios.

Aun así, hemos presentado, comentado y realizado lo máximo posible que hemos podido hacer de la práctica. Si tuviéramos que quedarnos con los puntos a favor de esta práctica y su utilidad, ha sido, la gran disponibilidad y esfuerzo de los profesores de prácticas por enseñarnos como funciona un sistema de ficheros, desde las funciones del sistema, hasta diferenciar los datos, de los metadatos. El punto en contra de la práctica, quizá hemos tenido poco tiempo para poder completar esta práctica en condiciones.

1. 100KB/2048B – 1 = 49; donde 100KB es el tamaño máximo del disco, y 2048 el tamaño máximo de un fichero. [↑](#footnote-ref-1)