Grado en Ingeniería Informática

Práctica 2:
Sistema de Ficheros

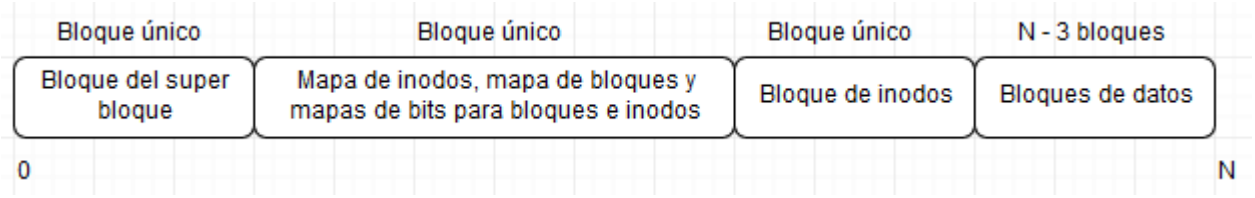
Diseño de sistemas operativos

Autores:

Alberto Gómez Aparicio (100363805)
Juan José Garzón Luján (100363861)
Carlos García Cañibano (100363813)

Índice de contenidos

Diseño detallado del sistema de ficheros



Este diagrama superior indica la repartición por ocupación de bloques del espacio que ocuparán todas las estructuras básicas del sistema de ficheros. El último bloque de datos indica la posición inicial y su rango de crecimiento. Cabe destacar que hemos intentado cumplir con el requisito de optimizar al máximo los recursos de espacio del sistema estableciendo un compromiso con el respeto de la semántica. Es decir, los bloques están distribuidos en primer lugar de manera lógica y, en segundo lugar, de manera óptima. Hemos intentado que nuestro sistema de ficheros se comporte de forma parecida al de Linux y por este motivo utilizamos uno de los cuarenta ficheros máximos para crear la raíz "/".

Para poder implementar el sistema de ficheros hemos necesitado estructuras de datos que vamos a clasificar por ubicación dentro del sistema (disco o memoria). Son las siguientes:

I. Estructuras de disco necesarias para el sistema de ficheros:

El super bloque contendrá los siguientes campos: **numeroMagico**, **numeroBloquesMapaInodos**, **numeroBloquesMapaDatos**, **numeroInodos**, **primerInodo**, **numeroBloquesDatos**, **primerBloqueDatos**, **tamanoDispositivo**.

Hemos estimado oportuno recoger estos datos para el correcto manejo del sistema de ficheros. El super bloque es una estructura importante que recoge la información de montaje esencial del sistema de ficheros. También sirve de orientación para la implementación de las funciones necesarias para su manejo.

Para la estructura que representa al inodo hemos decidido recoger los siguientes datos: **tipo**, **nombre**, **tamaño**, **bloqueDirecto**.

El tipo nos ayudará a distinguir entre fichero o directorio, el nombre identificará al fichero dentro de una ruta, el tamaño informará acerca de la ocupación de datos y el bloque directo la asociación entre inodo y bloque de datos. Pensamos que con estos datos la estructura de inodo en disco queda completa y está correctamente definida.

Para los mapas de bits hemos decidido utilizar dos estructuras donde tanto inodos como bloques de datos tendrán su mapa de ocupación de un máximo de 40 coincidencias para cumplir con los requisitos de especificación del sistema.

II. Estructuras de memoria necesarias para el sistema de ficheros:

Para realizar un correcto manejo de los inodos cuando se encuentran en memoria y permitir una mayor funcionalidad con el manejo de estos, hemos creído oportuno realizar un encapsulamiento de la estructura de disco que añada los campos **posición**, **estado** y un **puntero a la estructura de disco**. De esta manera podemos distinguir los diferentes estados en los que puede encontrarse un fichero

(abierto o cerrado) y donde se dejó la última operación de lectura/escritura. Para este diseño, no necesitamos contemplar más situaciones.

III. Diseño de las rutinas de gestión:

Para la reserva de inodos hacemos uso de la función **ialloc** y su esquema básico de funcionamiento es el siguiente:

- ➔ Buscar en el mapa de bits de inodos con límite del número de ficheros declarado en super bloque una entrada que tenga valor nulo.
- ➔ Cuando encontramos el espacio en el mapa de bits, lo cambiamos a ocupado y recordamos el índice de encuentro, que será la representación numérica del inodo libre.
- ➔ Se actualiza el mapa de sincronización para aportar coherencia global al sistema.
- ➔ Se controlar finalmente si se ha encontrado un espacio libre para el inodo en cuestión y, en caso contrario, se retorna un código de error.

Para liberar los inodos hacemos uso de la instrucción **ifree** con un funcionamiento resumido explicado a continuación:

- ➔ Controlamos los límites de índice de inodo que se hayan podido reservar para detectar errores.
- ➔ Obtenemos el índice de bits dentro del mapa de bits de inodos.
- ➔ Ponemos a 0 ese bit.
- ➔ Sincronizamos los cambios con las estructuras involucradas.

En este caso buscamos un bloque libre. Para este fin utilizaremos la función **balloc**:

- ➔ Buscamos en el mapa de bits de bloques una posición vacía.
- ➔ Si encontramos una posición a 0, ya tenemos la reserva de nuestro nuevo bloque de datos.
- ➔ Sincronizamos cambios con las estructuras involucradas.
- ➔ Devolvemos el índice de bloque reservado utilizando como apoyo el indicador de primer bloque de datos que almacena el super bloque.
- ➔ Controlamos el posible error de no encontrar un bloque libre y lo comunicamos.

Liberando un bloque recurrimos a la función **bfree**:

- ➔ Controlamos los limites de índice de bloque que se hayan podido reservar para detectar errores.
- ➔ Se reserva un buffer de tamaño del bloque para ir liberando los datos que pueda tener con la función proporcionada **bwrite**.
- ➔ Localizamos en el mapa de bits de bloques de datos la posición que se correspondía con el bloque liberado y la ponemos a 0.
- ➔ Sincronizamos cambios con las estructuras involucradas.

Para conocer mayores detalles acerca de la implementación de estas funciones gestión, les recomendamos consultar directamente los ficheros de código fuente puesto que, para cumplir con los requisitos de documentación, no se ha añadido código fuente, si no un resumen esquemático del funcionamiento.

IV. Diseño de las llamadas que componen el sistema de ficheros

Antes de explicar las llamadas, cabe destacar una función de nuestro sistema de ficheros; utilizando un **mapa de sincronización** de bits conoceremos el estado de un bloque que esté en memoria, si el bit de una posición es 0 tiene los mismos datos que el disco, en caso contrario, si hay un 1, significa que se ha actualizado la memoria y habrá que actualizar el bloque de esa posición.

Todas las funciones no explicadas aquí y que se consideran auxiliares se explican en el punto V. (lsInodo, trocearRuta, etc…) Consultar el índice para más detalles.

En el caso de implementar el creado del sistema de ficheros en el sistema de almacenamiento explicamos la implementación de la función **mkFS** de la siguiente manera:

- ➔ En primer lugar, comprobamos que el tamaño del almacenamiento es adecuado dentro de los requisitos del sistema. Se comprueba también la existencia del archivo y los criterios de tamaño mínimos y máximos. También miramos si se ha montado con anterioridad el sistema para, antes de volver a crear, desmontar el anterior.
- ➔ Formateamos los bloques del dispositivo asignados al sistema de ficheros para evitar corrupción de datos con sistemas anteriores.
- ➔ Inicializamos la estructura del super bloque con los datos necesarios según el diseño. Estos datos nos serán necesarios para gestionar el sistema más adelante. Se crean también las estructuras en disco de mapas de bits y los inodos de disco.
- ➔ La última operación sería sincronizar con el disco todas las estructuras para asegurar su persistencia física.

En la situación de montaje del sistema de ficheros, rescatamos las estructuras de disco almacenadas para instanciarlas en memoria, ampliar su funcionalidad con estructuras adicionales y poner en funcionamiento el mismo. La función **mountFS** llamada a este cometido funciona de la siguiente manera:

- ➔ Sincronizamos la memoria y creamos los inodos de memoria basados en los inodos creados en disco. Para esto añadimos metadatos adicionales como la posición del puntero y el estado del fichero dentro del sistema.
- ➔ Reservamos los bytes necesarios para el uso de los mapas de bits de inodos y bloques con un valor nulo inicial.

El desmontaje del sistema de ficheros asegura que el estado actual del sistema de ficheros es salvado para futuras ejecuciones. La función responsable de esto es **unmountFS** y funciona así:

- ➔ Sincronizamos con el disco todas las estructuras de memoria para asegurar la persistencia de estas si han sido modificadas gracias al mapa de sincronización.
- ➔ Restauramos el estado de los ficheros al desmontar el sistema de ficheros.
- ➔ Liberamos todos los recursos reservados como el super bloque, mapas de bits, inodos de disco, inodos de memoria, etc…

Para el creado de ficheros y directorios se utilizan las funciones **createFile** y **mkDir**. Sin embargo, como pensamos que comparten gran parte del comportamiento, hemos externalizado su funcionamiento a una función de apoyo llamada **crearFichero(char *path, int tipo)**. De esta manera con solo la especificación por parámetro del tipo (FICHERO = 0, DIRECTORIO = 1) podemos diferenciar los casos de uso y proponer un modelo menos redundante. El funcionamiento esquemático de esta función sería el siguiente:

- ➔ Comprobar que la ruta es correcta en términos de formato y de longitud/profundidad máxima.
- ➔ Gracias a la función **infoFichero** somos capaces de saber si el archivo ya existe o si la ruta es válida. También comprobamos si es posible reservar inodo y bloque.
- ➔ Asignamos al nuevo fichero su tipo. FICHERO o DIRECTORIO, así como el inodo asociado, bloque directo y tamaño.
- ➔ Contemplamos un caso especial para el primer directorio de todos "/". Donde siempre se asignan los mismos datos.
- ➔ Si no estamos tratando con el fichero raíz y es un directorio, tenemos que modificar su bloque para poder añadir las rutas "." y ".." que corresponden con él mismo y una referencia a su padre. También hay que cambiar el bloque del padre para añadir la entrada de este nuevo hijo con el formato "<numero de inodo><espacio en blanco><nombre del fichero><\n>". Si solo es un fichero, solo hacemos esto último.
- ➔ Por último, modificamos los metadatos de las estructuras involucradas y actualizamos el mapa de sincronización.

Para el borrado de ficheros y directorios se utilizan las funciones **removeFile** y **rmDir**. El razonamiento es el mismo al de arriba. Utilizamos la función auxiliar **eliminarFichero(char *path, int tipo)**. El funcionamiento esquemático de la función sería el siguiente:

- ➔ Comprobar que la ruta es correcta en términos de formato y de longitud/profundidad máxima.
- ➔ Gracias a la función **infoFichero** somos capaces de saber si el fichero existe, si la ruta es válida o si el tipo de borrado corresponde con el tipo de fichero puesto que no se realizan las mismas operaciones exactamente.
- ➔ Si es un directorio, entonces también tenemos que borrar todo su contenido y activamos la recursividad de la función con ayuda de la función **lsInodo**.
- ➔ Después de cualquier caso de borrado tenemos que actualizar los bloques del padre para eliminar las entradas del hijo y liberar los recursos asignados al fichero en términos de bloque e inodo.
- ➔ Por último, modificamos los metadatos de las estructuras involucradas.

En el caso de apertura del fichero, la encargada es la función **openFile** que sigue un funcionamiento de este estilo. También cabe destacar que utilizamos un array de apoyo llamado **estadoFicheros** que se encarga de repartir los descriptores de fichero, el índice es el descriptor y el número almacenado es el inodo asociado:

- ➔ Comprobamos que la ruta es correcta en términos de formato, longitud/profundidad máxima, existencia del fichero y tipo, ya que solo puede ser abierto un fichero que no sea directorio.
- ➔ Se recorre el array auxiliar **estadoFicheros** buscando una posición vacía para que se asigne el índice al fichero que solicita ser abierto. Si no es posible, la operación de apertura falla.
- ➔ Se actualizan los metadatos del inodo asociado, se cambia su estado para que figure como abierto y se actualiza la posición de su puntero a 0 para cumplir con los requisitos de diseño.

Para el cerrado de un fichero usamos la función **closeFile** con el apoyo del array **estadoFicheros**. El funcionamiento es el siguiente:

- ➔ Primero comprobamos si el descriptor de fichero proporcionado es válido dentro de los límites de ficheros existentes. También nos fijamos si el estado de fichero está necesariamente abierto, puesto que un fichero cerrado no puede ser cerrado nuevamente.
- ➔ Si cumplimos estas condiciones, lo único que hacemos es actualizarlos metadatos del inodo en memoria para cambiar su estado a cerrado.

- ➔ Modificamos **estadoFicheros** para liberar el índice del descriptor de fichero asignado con el valor del inodo, poniendo su posición a cero.

Leyendo un archivo tenemos la función **readFile** que sigue este funcionamiento:

- ➔ Comprobamos que el número de bytes para leer no es negativo y que el descriptor de fichero exista.
- ➔ Buscamos el inodo asociado gracias al array de apoyo **estadoFicheros**. De esta manera obtenemos la posición de su puntero. Así comprobamos que el puntero no está en el final del fichero y que la lectura pueda ser posible.
- ➔ Recuperamos de disco el bloque asociado al fichero que se pretende leer.
- ➔ Copiamos en el búfer de lectura proporcionado por parámetro los datos asociados a la lectura.
- ➔ Reportamos los bytes leídos correctamente del fichero.

Para el caso de escritura de un fichero utilizamos **writeFile** que funciona así:

- ➔ Comprobamos que el número de bytes para escribir no es negativo y que el descriptor de fichero exista.
- ➔ Buscamos el inodo asociado gracias al array de apoyo **estadoFicheros**. De esta manera obtenemos la posición de su puntero que no sirve para saber a partir de donde escribimos y hasta donde podemos hacerlo.
- ➔ Leemos el bloque de fichero asociado y escribimos a partir de la posición permitida para no sobrescribir los datos anteriormente asociados al fichero.
- ➔ Modificamos el puntero del fichero en función de lo verdaderamente escrito y no de lo que se esperaba haber escrito. También se hace con el tamaño.
- ➔ Se informa del número de bytes escritos correctamente.

En la modificación del puntero de posición del fichero tenemos la siguiente función **lseekFile** con un funcionamiento como el que sigue:

- ➔ Comprobamos que el descriptor de fichero sea correcto dentro de los límites de ficheros.
- ➔ Con un switch distinguimos entre mover el puntero al final, al principio o a cualquier parte del documento según el offset.
- ➔ Informamos según el éxito de la operación.

En el caso de listar los elementos de un directorio utilizamos la función **lsDir**. Destacamos que tenemos que utilizar la función de apoyo **lsDirAuxiliar** para aplicar la recursividad al supuesto. En este caso la ejecución de ambas funciones es la siguiente:

- ➔ Comprobamos la ruta para evitar que sea inválida en los términos descritos en otros casos. Ruta no vacía, por ejemplo.
- ➔ Como la signatura de la función espera recibir los índices de los diez inodos máximos encontrados con sus nombres, pero nuestro diseño contempla dos más adicionales como son “.” y “..”. De esta manera mapeamos el resultado de la función auxiliar con la principal.
- ➔ La función auxiliar observa el criterio de parada recursivo de que la ruta se encuentre vacía después de inspeccionar individualmente cada nivel de profundidad y también de que nos encontremos con el caso especial del directorio raíz “/”. En ese caso contamos los elementos retornados y retornamos.

- ➔ Mientras la condición de parada no se dé, recortamos la ruta en un nivel de profundidad con la función **trocearRuta** y listamos el inodo extraído con **lsInodo**. Así peinamos ese nivel entero con todos sus inodos.
- ➔ En el siguiente paso comparamos el contenido del inodo con el fragmento de la ruta extraído para poder buscar el siguiente paso y comprobar siempre que sea un directorio para no fallar.
- ➔ Repetimos el procedimiento constantemente con una llamada recursiva hasta deshacer la ruta entera.
- ➔ Retomamos el control a la función principal que se encargará de devolver el número de elementos encontrados en el directorio pedido e imprimirá sus nombres.

V. Especificación de funciones auxiliares necesarias para el funcionamiento del sistema

Hemos creado la función **comprobarRuta** para comprobar que se respeta correctamente el formato de la ruta. Esto es, supuestas rutas con dos barras seguidas por ejemplo "//", vacías y otro tipo de errores. Básicamente troceamos la cadena de la ruta por bytes y comprobamos por segmentos la integridad de esta. También nos es útil para comprobar los requisitos de tamaño en el nombre de los ficheros y de rutas completas.

La función **bloqueModificado** se encarga de actualizar un bit a 1 del mapa de bits de sincronización para que al desmontar solo se guarde en disco aquello que ha sido modificado. Básicamente, el mapa de bits de sincronización sirve para evitar una escritura de metadatos que no han sido modificados durante el uso del sistema de ficheros.

Para poder sincronizar las estructuras de memoria con las de disco utilizamos la función **sincronizarDisco**. Escribimos en el fichero que guarda la información del sistema de ficheros todas las estructuras de memoria cuya información es necesaria para hacer posible la persistencia del estado global del sistema de ficheros. El mismo caso se utiliza para **sincronizarMemoria** para el caso inverso de disco a memoria. Para estos casos, también se utiliza el mapa de bits de sincronización.

La función **lsInodo** lista todo el contenido de un inodo individual y ayuda a diversas funciones como por ejemplo **lsDir**. Su funcionamiento es el de recorrer y consultar las estructuras existentes para componer un resultado completo y de uso útil.

En el caso de **trocearRuta** lo que hacemos es quitar un nivel de profundidad a la ruta actual para hacer posible la resolución iterativa de esta en profundidad. Devolvemos una cadena con la misma ruta sin el nivel superior y además el segmento aislado separado de la ruta inicial.

Finalmente, **infoFichero** devuelve la ruta del fichero padre respecto del fichero indicado en la ruta introducida por parámetro. También devolvemos el índice del padre y el índice del hijo en términos de identificación del inodo.

Nótese que la funcionalidad profunda no se encuentra detallada en esta parte del documento por escasez de número de páginas y por el requisito presente de no incluir código fuente. Sin embargo, les invitamos a conocer la codificación directamente para desvelar los detalles de las descripciones aportadas anteriormente en cada uno de los campos provistos.

VI. Estudio de los casos de prueba necesarios para justificar la funcionalidad esperada

En este apartado estudiaremos los requisitos aportados en el enunciado de la práctica y los agruparemos y/o examinaremos en pruebas aisladas y conjuntas que garantizarán su correcta

cobertura. Los casos de prueba que hemos identificado son los siguientes y se dividen en entrada, salida, comprobaciones, salida e interpretación de los resultados obtenidos:

Entrada	Comprobaciones (Requisitos)	Salida	Interpretación
mkFS() mountFS() unmountFS()	Comprobamos que se genera bien nuestro sistema de ficheros antes de añadir ficheros, directorios… (F1.1, F1.2, F1.2)	Todo se realizó correctamente.	La salida es la esperada, ya que estas operaciones deben hacerse con corrección para la creación correctamente del sistema de ficheros.
createfile(“/”)	No se puede crear ficheros sin nombre, ni con “/” en él. (F7)	No se realizó la creación.	La salida es correcta, ya que no tiene nombre el fichero y no se puede crear un fichero sin nombre ni con “/”.
mkDir(“/”)	No se puede crear directorios sin nombres, ni con “/” en él. (F7)	No se realizó la creación	La salida es correcta, ya que no tiene nombre el directorio y no se puede crear un directorio sin nombre ni con “/”.
mkDir("/a") mkDir("/a/aa") createFile("/a/af") mkDir("/a/aa/aaa") createFile("/a/aa/aaf") mkDir("/b") createFile("/f")	Probamos a crear varios directorios, varios ficheros y en diferentes directores. (F1.4, F1.11, F8).	Se crean todos los ficheros y directorios correctamente.	La salida es correcta, ya que no existía ningún directorio o fichero con esos nombres antes y el tamaño del nombre es válido.

mkDir("/aaaaaaaaaa aaaaaaaaaaaaaaa aaaaaaaa")	El nombre de un fichero o directorio podrá tener como máximo una longitud de 32 caracteres. (NF3)	No se realizó la creación.	La salida es correcta, ya que el nombre del directorio tiene 33 caracteres.
createFile("/aaaaaaaaaa aaaaaaaaaaaaaaa aaaaaaaa")	El nombre de un fichero o directorio podrá tener como máximo una longitud de 32 caracteres. (NF3)	No se realizó la creación.	La salida es correcta, ya que el nombre del fichero tiene 33 caracteres.
mkDir("/aaaaaaaaaa aaaaaaaaaaaaaaa aaaaaaa")	El nombre de un fichero o directorio podrá tener como máximo una longitud de 32 caracteres. (F1.11, NF3, F8)	Se realizó la creación.	La salida es correcta, ya que el nombre del directorio tiene 32 caracteres.
createFile("/aaaaaaaaaa aaaaaaaaaaaaaaa aaaaaaa")	El nombre de un fichero o directorio podrá tener como máximo una longitud de 32 caracteres. (F1.4, NF3, F8)	Se realizó la creación.	La salida es correcta, ya que el nombre del fichero tiene 32 caracteres.
Crear un directorio de 99 caracteres de ruta completa y dentro de él un fichero con 132 caracteres de ruta completa.	El tamaño máximo de una ruta completa de un fichero es de 99 caracteres y de un fichero 132. (F1.4, F1.11, NF4)	Se realizó la creación.	La salida es correcta, ya que las dos rutas están dentro de su tamaño permitido.
Crear un directorio con más de 99 caracteres de ruta completa.	El tamaño máximo de una ruta completa de un fichero es de 99 caracteres y de un fichero 132. (NF4)	No se realizó la creación.	La salida es correcta, ya que la ruta para un directorio supera su tamaño máximo.

Crear un fichero con más de 132 caracteres de ruta completa.	El tamaño máximo de una ruta completa de un fichero es de 99 caracteres y de un fichero 132. (NF4)	No se realizó la creación.	La salida es correcta, ya que la ruta para un fichero supera su tamaño máximo.
mkDir("/a") mkDir("/a/aa") createFile("/a/af") mkDir("/a/aa/aaa") createFile("/a/aa/aaf") mkDir("/b") createFile("/f") createFile("/a/aa/aaa/f") mkDir("/a/aa/aaa/aaaa")	Intentamos crear un directorio de profundidad 4. (NF5)	No se realizó la creación del directorio de profundidad 4 pero si todo lo anterior.	La salida es la correcta, ya que puedes tener directorios con profundidad menor o igual a 3, y en el caso de tener un directorio de profundidad 3, en él se pueden crear ficheros.
Un bucle para crear más de 10 ficheros dentro del directorio "/".	El número de elementos máximo dentro de un directorio no puede superar a 10. (F1.4, F8, NF2)	Se crearon los 10 primeros ficheros correctamente.	La salida es la correcta ya que te permite crear 10 ficheros en un mismo directorio, pero no más.
Crear 40 archivos (ficheros y directorios).	El máximo número de ficheros en el sistema de ficheros no será nunca mayor de 40. (F1.4, NF1)	Se crean 39 archivos correctamente, al llegar al 40 no te lo puede crear.	La salida es la esperada, ya que al hacer el mkFS, hemos creado ya el directorio "/", entonces solo se podrían 39 archivos más, en total son 40 archivos.
Crear 40 archivos (ficheros y directorios).	El máximo número de ficheros en el sistema de ficheros no será nunca mayor de 40. (NF1)	Se crean los 39 archivos correctamente.	La salida es la esperada, ya que al hacer el mkFS, hemos creado ya el directorio "/", entonces solo se podrían 39 archivos más, en total son 40 archivos.
Partiendo que tenemos la estructura básica. (Explicada la estructura después de la tabla) rmDir("/a/aa/aaa") removeFile("/a/aa/aaf") rmDir("/a") removeFile("/f") createFile("/f") rmDir(/f)	Eliminar directorio y ficheros en diferentes longitudes de ruta y que se invoca bien la función para borrar cada elemento. (F1.5, F1.11,	Se eliminaron bien todas las operaciones excepto el último ya que es un fichero y no un directorio.	La salida es la correcta, nos debe dejar los directorios si existen y si se invoca bien su función de borrada, dependiendo de si es directorio o fichero.

estructuraPrueba() testsEscribir() lecturaConPocoTamano()	Creamos nuestra estructura básica, abrimos, cerramos leemos, usamos "lseek", y escribimos en el fichero "/". Verificamos que se cumplen todos los requisitos con las funciones de escribir, leer y de la función "lseek". Aparte de ver que el tamaño nunca supera el tamaño de un bloque.(F1.4, F1.6, F1.7, F1.8, F1.10, F1.11, F2, F3, F4, F5, NF6, NF7).	Realiza todas las operaciones correctamente.	La salida es la correcta, ya que estamos comprobando que los requisitos funcionales y no funcionales se cumple.
mkDir("/a") mkDir("/a")	No pueden existir dos directorios con el mismo nombre en un mismo directorio.	Se realiza el primer directorio, el segundo ya que ya hay un archivo en ese directorio con el mismo nombre.	La salida es la correcta, ya que no te debe permitir crear archivos con nombres que ya existen en un directorio.
createFile("/a") createFile ("/a")	No pueden existir dos ficheros con el mismo nombre en un mismo directorio.	Se realiza el primer fichero, el segundo ya que ya hay un archivo en ese directorio con el mismo nombre.	La salida es la correcta, ya que no te debe permitir crear archivos con nombres que ya existen en un directorio.
Partiendo que tenemos la estructura básica. lsDir("/") lsDir ("/a") lsDir ("/a/aa") lsDir ("/f")	Listamos contenidos de distintos directorios (F1.13)	Lista los contenidos de los directorios correctamente si existen.	La salida es la correcta, muestra bien los archivos que están dentro de cada directorio si existen. Si es fichero o no existe el directorio, no puede completar la operación.

mkFS() mountFS() unmount() mountFS() unmount() *Se puede hacer en diferentes ejecuciones porque los metadatos son persistentes	Creación del FS y desmontar para que se guarden los metadatos (NF8)	Todo se realizó correctamente	En la 2ª parte de la ejecución tenemos los metadatos sin haber hecho mkFS()
create_disk 40 mkFS(30 * BLOCK_SIZE)	Tener un archivo con 40 bloques y crear nuestro SF de 30 (F6)	Todo se realizó correctamente	Al dejar los últimos 10 libres, nunca se usarán
Sin haber hecho posteriormente mkFS(): mountFS() unmount()	Montar el FS sin haberlo creado antes	Todo se realizó correctamente	El mount() da el error porque no se ha hecho mkFS

En todas pruebas que no aparece mkFS(), mountFS() y unmountFS(), están incluidas esas operaciones, para poder realizar las pruebas explicadas.

Con el término de "estructura básica" nos referimos a lo que creamos en la cuarta prueba de la tabla (se crea con el método estructuraPrueba()). Los elementos que terminan en f son ficheros, el resto directorios:

```
/
        a
                aa
                        aaa
                        aaf
                af
        b
        f
```

VII. Conclusiones técnicas y problemas encontrados

Entre los principales problemas encontrados podemos destacar el manejo de los mapas de bits. No estábamos familiarizados con este tipo de operaciones. También las funciones recursivas presentaron problemas de entendimiento y depuración para los casos de listar un directorio y de borrarlo junto con todo su contenido. No obstante, descubrimos que con el uso de funciones auxiliares podíamos facilitar nuestras tareas, modular las grandes operaciones y externalizar responsabilidades. De esta manera fue posible depurar y diseñar más eficientemente, así como mejorar la legibilidad del proyecto final.

Personalmente, la práctica nos ha ayudado a comprender en profundidad como funciona un sistema de ficheros manejando sus estructuras de datos básicas a la vez que incrementábamos su funcionalidad. Nos ha parecido interesante y desafiante a partes iguales en algunos momentos determinados.