Universidad Carlos III de Madrid

Diseño de Sistemas Operativos

Ingeniería Informática

Grupo 83

Práctica 3: Sistema de ficheros

Autores:

Daniel Medina García	NIA 100316850
Alejandro Rodríguez Salamanca	NIA 100315874
Diego Vicente Martín	NIA 100317150

30 de abril de 2016

Índice

1. Descripción del código

1.1. Descripción del Código Inicial

Las funciones implementadas en el código inicial vienen concretadas en el propio enunciado de la práctica, si bien cabe mencionar brevemente la estructura del fichero que vamos a modificar, i.e. `filesystem.c`. Este archivo contiene las funciones que debemos implementar con el cuerpo vacío, y una clara descripción de lo que debe devolver cada una de ellas.

1.2. Decisiones de diseño e implementación

En este punto obviaremos descripciones detalladas del código, para centrarnos en las decisiones de diseño y estabilidad de nuestro código. Si bien el código inicial se dividía en tan solo tres secciones que detallaremos más adelante, aquí también explicaremos el diseño de las estructuras de datos utilizadas, optimizadas tanto como supimos para aprovechar al máximo los recursos de los que se contaban.

1.2.1. Estructuras de datos

Las estructuras de datos utilizadas en el sistema de ficheros deben de ser optimizadas lo más posible para evitar malgasto de recursos mientras se cumplan todas las funcionalidades acordadas.

En nuestra implementación, utilizamos estructuras de datos para definir el superbloque, los inodos y las etiquetas.

- **superblock** : el superbloque es el primer bloque del dispositivo de almacenamiento. En él se almacenan los metadatos del sistema de ficheros como son:
  - `byte numFiles` indica el número de ficheros existentes, equivalente también al número de inodos. Puesto que en nuestro sistema de ficheros no existe la opción de eliminar, los ficheros se escribirán secuencialmente, y este número determinará la posición del último fichero creado en el array de inodos.
  - `byte maxNumFiles` limita el número de ficheros que se pueden almacenar en el sistema de ficheros. Su valor viene dado por el parámetro del mismo nombre que se recibe en `mkFS`. El máximo valor que puede tomar este atributo es `MAX_FILES`, cuyo valor es 50.
  - `tag tags[MAX_TAGS]` es un array de la siguiente estructura de datos que comentaremos, `tag` y contiene todas las etiquetas aplicadas a los ficheros. Se almacenan así las etiquetas de forma unívoca aquí y se referencian a través de su `id` para optimizar espacio al evitar redundancia cuando asignamos a varios archivos la misma etiqueta, a la vez que limitamos el número máximo de etiquetas como es requerido en el enunciado. `MAX_TAGS` es una constante declarada al inicio de `filesystem.c` cuyo valor es 30.
  - Por último, el superbloque contiene un bitmap `byte tagmap[MAX_TAGS]` para llevar la cuenta de los huecos disponibles en el array de etiquetas, agilizando el acceso a las mismas. Los elementos de este array pueden contener dos distintos valores:

    - `FREE_TAG` - Indica que en esa posición del array de tags se puede escribir una nueva etiqueta, pues está libre. Su valor es de 0.
    - `OCCUPIED_TAG` - Indica que en esa posición del array de tags no se puede escribir una nueva etiqueta, pues ya está ocupada. Su valor es de 1.

- **tag** : las etiquetas son elementos aplicables a un archivo por los cuales se pueden buscar dentro del sistema de ficheros. Cada etiqueta puede ser aplicada a uno o más ficheros, tienen un numero máximo de caracteres y en ningún caso podrá haber más de un número determinado de etiquetas en el sistema de ficheros. Dadas estas restricciones, implementamos una estructura que llevase cuenta de cada etiqueta así como del número de archivos referenciables por ella. De esta forma, el contenido de cada estructura consta de:

    - `char name[MAX_TAG_NAME]`, *string* con el contenido de la etiqueta en cuestión. El valor de la constante `MAX_TAG_NAME` es 32.
    - `byte counter` es un contador de referencias a esta etiqueta. Cada vez que un archivo sea etiquetado, este contador subirá en una unidad. De esta forma, cuando una etiqueta se deje de usar, el valor del contador será 0, y podremos marcar la posición en la que se encuentra en el array de tags con `FREE_TAG` en `tagmap`, para indicar al sistema de ficheros que puede sobreescribir esa posición con una nueva etiqueta.

- **inodo** : Cada fichero será accesible a través de un inodo que contiene los metadatos del fichero:

    - `char name[MAX_NAME_LENGTH]` es una *string* que almacena el nombre del archivo dentro del sistema de ficheros. La longitud máxima del nombre de un fichero puede ser 64.
    - `byte state` es el atributo por el cuál identificamos un fichero como abierto (algún proceso está accediendo a su contenido) o cerrado (cuando `state` sea igual a 0).
    - `short fileSize` contiene el tamaño ocupado por el fichero.
    - `unsigned short offset` indica la posición de lectura/escritura actual del fichero.
    - `byte tagId[MAX_TAGS_FILE]` almacena los índices de las etiquetas que se han asignado a este fichero de forma que podemos acceder rápidamente a ellas a través del array que las contiene en el superbloque. Los elementos de este array pueden tomar dos valores:
        - El index en el array de tags de la etiqueta asignada al fichero.
        - `NO_TAG` - Constante cuyo valor es 255 que indica que esa posición no tiene una etiqueta asignada. Se ha elegido el valor de 255 porque era necesario que éste valor fuese mayor que el número máximo de etiquetas, `MAX_TAGS`.

Como se puede observar, dado el reducido tamaño del sistema a implementar definimos un nuevo tipo de datos `byte` que utilizamos en lugar del clásico `int` reduciendo considerablemente el tamaño del superbloque y los inodos. Este nuevo tipo de dato consiste en un `char`

renombrado como `typedef unsigned char byte` para poder comprender más facilmente el propósito de los cambos donde se usa: ocupar un byte.

1.2.2. Gestión del sistema de datos

Una vez explicadas las estructuras de datos que utiliza nuestro sistema de ficheros, pasamos a comentar los aspectos más delicados de las funciones implementadas. En esta primera parte del código tenemos las funciones encargadas de la gestión del sistema de datos, tanto su inicialización como su montaje.

`mkFS()` es la función encargada de inicializar el sistema de ficheros. Toma como parámetros el tamaño total del sistema de ficheros así como el número total de archivos que puede contener. Existen unos límites para ambos, y es aquí donde se comprueban los valores que recibe la función. Más adelante se procede a la inicialización del superbloque `sblock`, asignando los valores iniciales a sus atributos. Hecho esto, se formatea el disco escribiendo ceros todos los bloques del dispositivo para comenzar a escribir en él. Los dos primerso bloques se rellenan aquí con los metadatos (superbloque e inodos) tras los cuales se podrán añadir más adelante los bloques de datos.

`mountFS()` realiza el montaje del sistema de ficheros, es decir, carga en memoria los dos primeros bloques del dispositivo (el superbloque y los inodos), que se encuentran escritos en el disco, para hacer posible la utilización del dispositivo.

`umountFS()`, al contrario que la función anterior, realiza la operación contraria y escribe en el disco los metadatos del sistema de ficheros después de cerrar todos los ficheros abiertos. Cabe destacar que, para evitar la pérdida de metadatos ante una extracción no segura (en este caso hablaríamos de matar el proceso de test antes de hacer el `umount()`), cada vez que hay un cambio en los metadatos, ya sea porque se ha escrito, creado, o cambiado la posición del puntero de lectura, éstos son escritos en disco.

1.2.3. Lectura y escritura

Las siguientes funciones son las encargadas de las interacciones de los procesos externos con el sistema de ficheros. Éstas posibilitan la edición del contenido del sistema de ficheros al crear (`creatFS()`), leer (`readFS()` y modificiarlos (`writeFS()`). Un proceso podrá interactuar sólo con ficheros que tenga abiertos (`openFS()`), los cuales podrá recorrer para acceder a la parte del fichero que desee (`lseekFS()`) sin tener que leer todo el fichero completo. Al finalizar las operaciones que el proceso quiere realizar con el archivo, debe liberar el recurso mediante `closeFS()`. A continuación describimos el diseño de las funciones mencionadas:

- `creatFS()` : antes de crear un nuevo fichero, se deben comprobar que:
  - El nombre del fichero que se desea crear se encuentra dentro del límite que establece `MAX_NAME_LENGTH`.
  - Quedan descriptores de fichero libres en el disco.

- El archivo no existe ya en el disco.

Una vez aseguramos que el archivo puede ser creado, inicializamos un nuevo inodo para el fichero vacío, y lo insertamos en el array en memoria de inodos. Este nuevo inodo se creará cerrado (`state = STATE_CLOSED`), tendrá tanto el tamaño (`filesize`) como el puntero de lectura/escritura (`offset`) igual a cero y no tendrá etiquetas. Finalmente, se actualizan el campo del superbloque `sblock` que contiene el número de ficheros existentes.

- `readFS()` es capaz de extraer un número dado de bytes de un fichero también dado. Para ello, comprueba que los parámetros sean legales (i.e. que existe el fichero y que se trate de leer un número de bytes menor al tamaño máximo permitido del fichero) y, tras el visto bueno, devuelve el número de bytes que ha leído en realidad almacenando lo leído en el buffer pasado como parámetro a la función. De esta forma, si se lee desde el puntero de lectura hasta el final del archivo y no se alcanza el tamaño pedido, queda constancia de ello.

- Por el contrario, `writeFS()` se encarga de modificar el contenido de un archivo. Para esta operación se requieren como precondiciones:

  - El archivo en el que se quiere escribir está abierto.
  - El tamaño del contenido a escribir no sobrepasa el tamaño máximo de un fichero en el sistema.
  - El archivo tiene el puntero de lectura/escritura `offset` al final del archivo (`EOF`). De otra forma, se devuelve 0 para indicar que no se pudo escribir nada por fin de archivo.
  - Desde el puntero de lectura/escritura actual hasta el final del archivo hay al menos el número de bytes que se quieren escribir (se calcula la diferencia entre el tamaño máximo del fichero y la posición del puntero).

  Cerciorada la posibilidad de escribir en el fichero sin problemas, se procede a modificar el contenido del buffer en memoria que contiene el contenido del fichero en disco, para posteriormente volver a escribir el bloque ya modificado en el dispositivo. En caso de que se haya modificado el tamaño del archivo en disco, se modifica el inodo correspondiente para reflejar esta expansión.

- `lseekFS()` es la función que permite el desplazamiento del puntero de lectura/escritura de un archivo sin el acceso al contenido del mismo. Tras comprobar que el fichero al que se trata de acceder existe y está abierto, se procede a mover dicho puntero en función del parámetro `whence`:

  - Si es igual a `FS_SEEK_END` el puntero se mueve al final del archivo.
  - Si es igual a `FS_SEEK_BEGIN` el puntero se mueve al principio del archivo.
  - Si es igual a `FS_SEEK_SET` el puntero se mueve a la posición indicada por el tercer parámetro recibido, `offset`, siempre y cuando esta sea una posición legal.

- `openFS()` permite a un proceso el acceso a un fichero al abrirlo. Para ello, busca el fichero por nombre en el sistema de ficheros (previa comprobación de la legalidad del nombre) iterando sobre los inodos. Una vez encontrado, esta función devuleve el descriptor de fichero correspondiente.

- `closeFS()` realiza la acción contraria a la función anterior. De esta forma, libera los recursos del archivo abierto tras comprobar que el descriptor de fichero pasado como argumento es correcto y señala a un fichero abierto.

1.2.4. Gestión de etiquetas

La última funcionalidad a comentar es la gestión de las etiquetas. Un archivo puede tener tantas etiquetas como limite el parámetro `MAX_TAGS_FILE`, mientras que la suma total de las distintas etiquetas aplicadas a los archivos del conjunto del sistema no exceda lo indicado en `MAX_TAGS`. Para la edición del etiquetado, utilizamos las siguientes funciones:

- `tagFS()` se encarga de la inclusión de nuevas etiquetas. Con un descriptor de fichero válido (existente y referente a un fichero abierto) etiqueta el fichero con el *string* pasado como segundo argumento.

    - Si la etiqueta no existía en la base de datos del superbloque previamente, esta función llama a `addTagToSystem()` para añadirla. Esta función auxiliar se encarga de actualizar tanto el array de etiquetas del superbloque como el bitmap de existencia de etiquetas. Si la nueva etiqueta no tiene espacio, se devuelve error.
    - Una vez cerciorada la existencia de la etiqueta en la lista, se realiza el etiquetado con `addTagToFile()`, que busca el primer hueco libre en las etiquetas del inodo del fichero para guardar aquí la nueva. Si no hay hueco, se devuelve error.

- `untagFS()` deshace la tarea de la función anterior. Tras comprobar la adecuación de los parámetros pasados, busca la etiqueta en el fichero correspondiente y la elimina. En caso de que este archivo fuese el único utilizando dicha etiqueta, ésta se elimina también del superbloque para dejar espacio a nuevas etiquetas.

- `listFS()` permite listar los archivos que hayan sido etiquetados con un nombre concreto. Esta función compara el índice de la etiqueta pasada como atributo con todos aquellos contenidos en los arrays de etiquetas de los ficheros, devolviendo la lista de ficheros etiquetados en el buffer pasado en forma de puntero y el número de ficheros encontrados como valor. En caso de no encontrar coincidencias, devuelve error.

Al modificar los metadatos del sistema de ficheros, estas funciones terminan acutalizando los metadatos en disco para asegurar la persistencia de éstos en caso de extracción insegura del dispositivo, i.e. la terminación inesperada del proceso con el disco aún montado.

2. Batería de Pruebas y Análisis de Resultados.

A la hora de realizar las pruebas pertinentes en el sistema de ficheros, hemos decidido probar todas las funciones que se ofrecen al usuario, comprobando su funcionamiento tanto en casos correctos y forzando distintos tipos de errores y casos extremos para comprobar cómo el sistema los maneja. Con ello buscamos conseguir la máxima cobertura del código, usando la interfaz del usuario para realizar tests más completos e intuitivos. Aunque al principio parezca una desventaja frente a tests realizados a más bajo nivel (y por tanto más atómicos y fáciles de depurar), los tests realizados ejecutan casos muy específicos, en los que es fácil seguir el flujo del programa en cada uno de los tests.

Los test que realizaremos se dividen en 4 grandes categorías: Tests del sistema de archivos, tests de operaciones con archivos, tests de etiquetas y tests de persistencia. En el primer grupo podemos encontrar:

1. **Creación del sistema de archivos**: usar `mkFS()` para crear el sistema de archivo con valores correctos. Como errores, probamos a crear un sistema de archivos más grande que el almacenamiento disponible en el disco.

2. **Montar el sistema de archivos**: usar el método `mountFS()` para montar el sistema de archivos.

3. **Desmontar el sistema de archivos**: usar el método `umountFS()` para desmontar el sistema de archivos.

Como tests a realizar sobre las operaciones con archivos, tenemos:

1. **Crear un archivo**: usar `creatFS()` para crear un archivo en el sistema, usando valores correctos. Se comprueba el error devuelto cuando se intenta crear un archivo ya creado, un archivo con un nombre más largo de 64 caracteres, cuando se crea un archivo sin nombre y cuando se crean más archivos de los que debe aceptar el sistema de archivos.

2. **Abrir un archivo**: Se utiliza `openFS()` para comprobar el descriptor de archivo devuelto por el sistema. Debemos comprobar el error devuelto al intentar abrir un archivo que no existe.

3. **Cerrar un archivo**: Se utiliza el método `closeFS()` para cerrar el descriptor de un archivo previamente abierto. Debe devolver un error cuando se cierra un descriptor que no existe o cuando se cierra un archivo que no estaba abierto.

4. **Escribir un archivo**: Se utiliza `writeFS()` para escribir un archivo usando un descriptor. Se debe probar también el comportamiento del sistema al intentar escribir más de un bloque.

5. **Leer un archivo**: Se utiliza `readFS()` para leer un archivo usando su descriptor. Para probar errores se fuerza el método leyendo tanto fuera del archivo como pasando un parámetro negativo.

6. **Modificar el offset**: Se utiliza `lseekFS()` para mover la posición de un puntero en el descriptor de un archivo. Para ello, se comprueba que funciona correctamente con cualquiera de los modos para ir al principio, al final y el movimiento con offset desde la posición actual. También comprobamos que el método no acepta offsets negativos y que no se desplaza fuera del archivo.

Para probar las etiquetas del sistema de archivos:

1. **Crear una etiqueta**: usar `tagFS()` para crear una etiqueta, asignándola al archivo que se pasa como parámetro al método. De esta manera, el archivo queda etiquetado. Para probar los errores de este método, debemos intentar etiquetar un archivo ya etiquetado con la etiqueta, usar una etiqueta de más de 32 caracteres, e intentar etiquetar un archivo que esté abierto.

2. **Añadir varias etiquetas**: usar `tagFS()` para asociar un mismo archivo 1, 2 o 3 etiquetas. Para probar errores en este concepto debemos añadir más de 3 etiquetas a un archivo, crear más de 30 etiquetas, y comprobar que se pueden reescribir las etiquetas de forma correcta una vez se llega al límite (usando `untagFS()`).

3. **Búsqueda por etiqueta**: usar `listFS()` para realizar una búsqueda por etiqueta. Para ello, crearemos múltiples archivos y etiquetas, para de esta manera controlar los resultados que debe producir la búsqueda y contrastarlos con los que devuelve el método. De igual manera, probamos el error deveulto por el método al intentar buscar por una etiqueta que no existe.

4. **Desetiquetar archivos**: usando el método `untagFS()`, se comprueba que un archivo se pueda desetiquetar correctamente. Además de ello, probamos el comportamiento del método al desetiquetar un archivo que tenía esa etiqueta y a desetiquetar un archivo que no existe.

5. **Búsqueda después de desetiquetar**: se comprueba que las etiquetas se eliminan correctamente de los archivos y que no figuran en una búsqueda por etiqueta si previamente han sido desetiquetados. También se prueba a desetiquetar todos los archivos de una etiqueta y comprobar que no queda ningún tipo de metadato residual.

Por último, la prueba de persistencia consiste en crear un sistema de archivos de forma correcta, así como archivos de ejemplo y escribir en ellos Además de esto, se etiquetan varios archivos para probar la persistencia de metadatos. Una vez realizadas estas operaciones, se desmonta el sistema de archivos. Al volver a montarse, la persistencia de los archivos y los metadatos es comprobada abriendo los archivos y consultando los metadatos. Si todo sigue tal y como se dejó antes de ser desmontado, el test ha sido superado correctamente. Además, cabe destacar que nuestro sistema es capaz de guardar los metadatos incluso si el sistema de archivo no ha sido desmontado correctamente.

Los resultados arrojados por la ejecución de los test son los siguientes:

```
1. FYLESYSTEM TESTS:
    1.1. TEST mkFS - correct creation: SUCCESS
        1.1.1 TEST mkFS - not enough storage: SUCCESS
    1.2. TEST mountFS - correct mount: SUCCESS
    1.3. TEST umountFS - correct unmount: SUCCESS

2. FILES TESTS:
    2.1. TEST creatFS - correct creation: SUCCESS
        2.1.1. TEST creatFS - already created: SUCCESS
        2.1.2. TEST creatFS - Name is too long: SUCCESS
        2.1.3. TEST creatFS - There is no name: SUCCESS
        2.1.4. TEST creatFS - More files than possible: SUCCESS
    2.2. TEST openFS - correct value: SUCCESS
        2.2.2. TEST openFS - no file: SUCCESS
    2.3. TEST closeFS - correct value: SUCCESS
        2.3.1. TEST closeFS - already closed: SUCCESS
        2.3.2. TEST closeFS - no file: SUCCESS
    2.4. TEST writeFS - correct value: SUCCESS
        2.4.1. TEST writeFS - write more than a BLOCK_SIZE: SUCCESS
    2.5. TEST readFS - correct value: SUCCESS
        2.5.1. TEST readFS - out of the file: SUCCESS
        2.5.2. TEST readFS - out of the file negative: SUCCESS
    2.6. TEST lseekFS:
        2.6.1. TEST lseekFS begin - correct: negative: SUCCESS
        2.6.2. TEST lseekFS end - correct: negative: SUCCESS
        2.6.3. TEST lseekFS set - correct: negative: SUCCESS
        2.6.4. TEST lseekFS set - incorrect: negative: SUCCESS
        2.6.5. TEST lseekFS set - negative: negative: SUCCESS

3. TAGS TESTS:
    3.1. TEST tagFS - correct creation: SUCCESS
        3.1.1. TEST tagFS - already tagged: SUCCESS
        3.1.2. TEST tagFS - tag too long: SUCCESS
        3.1.3. TEST tagFS - tagging an open file: SUCCESS
    3.2. TEST tagFS - more than 1 tag in a file: SUCCESS
        3.2.1. TEST tagFS - more than 3 tags in a file: SUCCESS
        3.2.2. TEST tagFS - more than 30 tags: SUCCESS
        3.2.3. TEST tagFS - rewriting tags: SUCCESS
    3.3. TEST listFS - returns correct: SUCCESS
        3.3.1. TEST listFS - inexistent tag: SUCCESS
    3.4. TEST untagFS - correct value: SUCCESS
        3.4.1. TEST untagFS - file is not tagged: SUCCESS
        3.4.2. TEST untagFS - file does not exist: SUCCESS
    3.5. TEST listFS + untagFS - returns correct: SUCCESS
        3.5.1. TEST listFS + untagFS all files: SUCCESS
```

```
4. PERSISTENCE TESTS:
    4.1. Correctly written: SUCCESS
    Unmounting... SUCCESS
    4.2 Correctly opened again: SUCCESS
    4.3 Metadata persistence: SUCCESS
```

En él, podemos ver cómo tanto los tests de correcto funcionamiento (X.Y) como los tests de casos extremos y errores (X.Y.Z) se superan correctamente. Con esta comprobación, consideramos que la cobertura del código es suficiente y el funcionamiento del sistema de archivos es el correcto.

3. Conclusiones, Problemas Encontrados y Opiniones Personales

A la hora de hacer esta práctica, la principal dificultad que nos hemos encontrado ha sido darnos cuenta de que los tests que se nos han proporcionado contenían errores. Mientras desarrollábamos el sistema de ficheros, cada vez que completábamos una nueva funcionalidad, ejecutábamos los test, y en alguna ocasión éstos mostraban un error, como en el caso del test de `FS_SEEK_SET`, lo que nos hacía pensar que nuestro código estaba mal. Hacer los tests requiere poner en práctica las distintas características de código, algo que no siempre es fácil de ver una vez se ejecutan las pruebas, y además no cerciorarse de que todos los aspectos funcionan bien pueden implicar arrastrar un fallo toda la práctica. Otro de los problemas que nos hemos encontrado ha sido el hecho de que el enunciado de la práctica no estaba del todo claro. Es algo que se solucionó en la clase de dudas de la práctica, pero sin embargo no deja de ser un problema tener que esperar a esas clases para poder estar seguro de que se está avanzando en la dirección correcta con la práctica.

Pese a todo lo comentado, la orientación de la práctica en sí nos parece adecuada. Tras pulir esos incómodos detalles nos parece que a través de ella hemos entendido cómo funciona un sistema de ficheros. No tuvimos problema con el tiempo disponible para la realización de la práctica, y agradecimos el apoyo del profesorado contestando nuestras dudas.