Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Administración de Empresas

Práctica 2: Sistema de Ficheros

Diseño de Sistemas Operativos

Eddy N. Almuiña Hernández

Versión 1.3

Índice

1. Introducción

El objetivo de esta segunda práctica es desarrollar un sistema de ficheros simplificado en espacio de usuario utilizando el lenguaje de programación C (ISO C11) sobre la máquina virtual provista para los cuadernos de práctica y/o *guernika*, cuyo almacenamiento estará respaldado por un fichero sobre el sistema operativo en el que se ejecute el programa.

Con esta práctica, se espera que el/la estudiante:

1. Diseñe 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 este documento.

2. Implemente el diseño anterior en el lenguaje de programación C, y desarrolle programas clientes teniendo en cuenta la interfaz del sistema para acceder y modificar sus ficheros.

3. Justifique el diseño y la implementación del sistema de ficheros, ajustándose a los conceptos teóricos adquiridos durante el curso. Esto es particularmente crítico para comportamientos indefinidos o dependientes de la implementación.

4. Cree un plan de pruebas para validar el diseño resultante frente a los requisitos pedidos.

5. Resuma y discuta los puntos anteriores de manera escrita en una memoria.

Todos estos elementos serán tenidos en cuenta en la evaluación de la práctica.

2. Evaluación

La calificación final de esta práctica viene determinada por los siguientes aspectos:

- **Diseño** (*4 puntos*).
- **Plan de validación** (*2 punto*).
- **Implementación** (*4 puntos*)

3. Código inicial

El fichero comprimido *dssoo_fs.zip* contiene el código inicial para poder comenzar. Una vez descomprimido, se puede encontrar lo siguiente:

- **Ficheros que NO DEBEN ser modificados**

  1. `include/blocks_cache.h`: declaración de las funciones para leer y escribir bloques de disco. En concreto, este fichero incluye las funciones `bread` y `bwrite` para leer y escribir, respectivamente, un bloque completo del disco dado un número de bloque. Está estrictamente **PROHIBIDO** acceder al disco de una forma diferente.
  2. `blocks_cache.c`: implementación de las funciones para leer y escribir bloques de disco.
  3. `include/filesystem.h`: declaración de las *únicas* funciones que representan la interfaz del sistema de ficheros. Estas funciones deben ser implementadas en el fichero `filesystem.c`.
  4. `create_disk.c`: permite crear un fichero `disk.dat` a partir de un tamaño dado.
  5. `Makefile`: se utiliza para compilar los distintos componentes. Mediante el comando `make` se compila y con el comando `make clean` se eliminan los ficheros compilados.

- **Ficheros a completar por el/la estudiante**

  1. `autores.txt`: fichero con los NIA de cada uno de los integrantes del grupo.
  2. `filesystem.c`: implementación de las funciones que representan la interfaz del sistema de ficheros. El estudiante debe utilizar este fichero para desarrollar el sistema de ficheros pedido. Cualquier función adicional desarrollada se debe implementar en este fichero, y ha de ser exclusivamente interna.
  3. `include/metadata.h`: definición de las estructuras, tipos de datos y constantes definidas por el/la estudiante para poder implementar el sistema de ficheros.

     Adicionalmente, se proporcionan las funciones `bitmap_getbit()` y `bitmap_setbit()` para manejar arrays de bit de longitud arbitraria, ya que serán útiles para realizar el seguimiento de bloques ocupados.

     `bitmap_getbit(bitmap_, i_)`: obtiene el estado del bit i-ésimo en el mapa de bits referenciado por `bitmap_` (de tipo `char *`).

     `bitmap_setbit(bitmap_, i_, val_)`: establece el estado del bit i-ésimo en el mapa de bits referenciado por `bitmap_` (de tipo `char *`) a `val_`.

     A continuación se muestra un ejemplo de uso:

     ```
     char bitmap[2];  // array de 16 bits,
     int val = bitmap_getbit(bitmap, 7);
     printf("%d\n", val); // Valor del bit 7 = 0
     bitmap_setbit(bitmap, 7, 1);
     val = bitmap_getbit(bitmap, 7);
     printf("%d\n", val); // Valor del bit 7 = 1
     ```

  4. `include/auxiliary.h`: declaración de las funciones auxiliares que sirvan para complementar las funciones principales declaradas en el fichero `filesystem.h`.

Estas funciones deben ser implementadas en el fichero `filesystem.c`, y no podrán ser utilizadas fuera del mismo (i.e. no se pueden utilizar para ampliar la interfaz del sistema de ficheros).

5. `test.c`: incluye un conjunto mínimo de pruebas para poder comprobar algunas características del sistema. Este fichero debe ampliarse para que incluya las pruebas creadas por el estudiante y así poder validar las características del sistema de ficheros implementado de forma exhaustiva, y además poder comprobar todos los posibles errores detallados en la interfaz.

Está estrictamente **PROHIBIDO** modificar la firma de las funciones de los ficheros `blocks_cache.h` y `filesystem.h` (p. ej. nombre, parámetros, tipo del valor de retorno).

La documentación interna de los ficheros de cabecera aporta información detallada sobre los valores devueltos por las funciones proporcionadas. El estudiante debe consultar esta información antes de comenzar a implementar su diseño.

4. Especificación de la funcionalidad

El estudiante debe diseñar e implementar desde cero un sistema de ficheros capaz de gestionar un dispositivo de almacenamiento emulado (`disk.dat`).

Los siguientes apartados definen en detalle la interfaz cliente de toda la funcionalidad pedida. Estas funciones están declaradas en el fichero `filesystem.h`, y deben ser implementadas dentro del fichero `filesystem.c`.

4.1. Gestión del dispositivo

```
int mkFS(long deviceSize)
```

- **Comportamiento:** genera la estructura del sistema de ficheros diseñada por el/la estudiante en el dispositivo de almacenamiento.
- **Parámetros:**
  - `deviceSize` – Tamaño del disco a dar formato, en *bytes*.
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – En caso de error. Intentar crear un sistema de ficheros que exceda los límites de capacidad de almacenamiento del dispositivo se considera un error.

```
int mountFS(void)
```

- **Comportamiento:** esta es la primera operación del sistema de ficheros a ejecutar por un programa cliente para así poder interactuar con los ficheros. Esta función monta el dispositivo simulado –*disk.dat*–, por lo que tiene que asignar y configurar todas las estructuras y variables necesarias para utilizar el sistema de ficheros.
- **Parámetros:** Ninguno.
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – En caso de error.

```
int unmountFS(void)
```

- **Comportamiento:** esta es la última operación del sistema de ficheros a ejecutar por un programa cliente, ya que desmonta el dispositivo simulado *disk.dat*. Esta función libera todas las estructuras y variables utilizadas por el sistema de ficheros.
- **Parámetros:** Ninguno.
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – En caso de error.

4.2. Gestión de ficheros

```
int createFile(char *path)
```

- **Comportamiento:** realiza todos los cambios necesarios en el sistema de ficheros para crear un nuevo fichero vacío.
- **Parámetros:**
  - `path` – Nombre y ruta del fichero a crear (ej. `/Movies/Mission_Impossible/poster"`).
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El fichero no puede ser creado porque ya existe en el sistema de ficheros.
  - `-2` – En caso de que exista cualquier otro error.

```
int removeFile(char *path)
```

- **Comportamiento:** realiza todos los cambios necesarios en el sistema de ficheros para eliminar un fichero.
- **Parámetros:**
  - `path` – Nombre y ruta del fichero a borrar (ej. `/Movies/Mission_Impossible/intro_song"`).
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El fichero no puede ser borrado porque no existe en el sistema de ficheros.
  - `-2` – En caso de que exista cualquier otro error.

```
int openFile(char *path)
```

- **Comportamiento:** abre un fichero existente en el sistema de ficheros e inicializa su puntero de posición al principio del fichero.
- **Parámetros:**
  - `path` – Nombre y ruta del fichero a abrir (ej. `/Movies/Mission_Impossible/poster2"`).
- **Valor de retorno:**
  - Descriptor del fichero abierto.
  - `-1` – El fichero no puede ser abierto porque no existe en el sistema de ficheros.
  - `-2` – En caso de que exista cualquier otro error.

```
int closeFile(int fileDescriptor)
```

- **Comportamiento:** cierra un fichero abierto.
- **Parámetros:**
  - `fileDescriptor` – Descriptor del fichero a cerrar.
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – En caso de error.

4.3. Interacción con los ficheros

```
int readFile(int fileDescriptor, void *buffer, int numBytes)
```

- **Comportamiento:** lee tantos bytes como se indiquen de un fichero representado por su descriptor, comenzando desde su puntero de posición y alojando los bytes leídos en un buffer proporcionado. Esta función incrementa el puntero de posición del fichero tantos bytes como se hayan leído correctamente.
- **Parámetros:**
  - `fileDescriptor` – Descriptor del fichero del que se quiere leer.
  - `buffer` – Buffer que almacenará los bytes leídos tras ejecutar la función.
  - `numBytes` – Número de bytes a leer del fichero.
- **Valor de retorno:**
  - Número de bytes leídos correctamente. Hay que tener en cuenta que si el número de bytes que se pueden leer del fichero es menor que el parámetro `numBytes`, la función debe devolver el número de bytes realmente leídos. En particular, leer cuando el puntero de posición del fichero está al final de este (no hay más bytes para leer), debe devolver un valor de 0.
  - `-1` – En caso de error.

```
int writeFile(int fileDescriptor, void *buffer, int numBytes)
```

- **Comportamiento:** modifica tantos bytes como se indiquen de un fichero representado por su descriptor, comenzando desde su puntero de posición y escribiendo en el fichero el contenido de un buffer proporcionado. Esta función incrementa el puntero de posición del fichero tras la escritura tantos bytes como se hayan escrito.
- **Parámetros:**
  - `fileDescriptor` – Descriptor del fichero en el que se quiere escribir.
  - `buffer` – Buffer con los datos a ser escritos en el fichero.
  - `numBytes` – Número de bytes a escribir del buffer.

- **Valor de retorno:**
  - Número de bytes escritos correctamente. Intentar escribir pasado el máximo tamaño de fichero no se considerará un error, y siempre se devolverá el número de bytes realmente escritos en el fichero. Por lo tanto, intentar escribir cuando el puntero de posición del fichero está al final de este, debe devolver un valor de 0.
  - `-1` – En caso de error.

```
int lseekFile(int fileDescriptor, int whence, long offset)
```

- **Comportamiento:** modifica el valor del puntero de posición de un fichero.
- **Parámetros:**
  - `fileDescriptor` – Descriptor del fichero.
  - `whence` – Constante de referencia para la operación de modificación del puntero de posición. Puede valer:
    - `FS_SEEK_CUR` – Posición actual del puntero de posición.
    - `FS_SEEK_BEGIN` – Comienzo del fichero.
    - `FS_SEEK_END` – Final del fichero.
  - `offset` – Número de bytes a desplazar el puntero de posición actual del fichero si la constante `whence` tiene el valor `FS_SEEK_CUR`. Este valor puede ser positivo o negativo, pero nunca debe posicionar el puntero de posición fuera de los límites del fichero. Si `whence` tiene como valor `FS_SEEK_BEGIN` o `FS_SEEK_END`, el puntero de posición del fichero debe modificarse al principio o al final del fichero, respectivamente (sin tener en cuenta el parámetro `offset`).
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – En caso de error.

4.4. Gestión de directorios

```
int mkDir(char *path)
```

- **Comportamiento:** realiza todos los cambios necesarios en el sistema de ficheros para crear un nuevo directorio vacío.
- **Parámetros:**
  - `path` – Ruta y nombre del directorio a crear (ej. `/Movies/Mission_Impossible"`)
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El directorio no puede ser creado porque ya existe en el sistema de ficheros.
  - `-2` – En caso de que exista cualquier otro error.

```
int rmDir(char *path)
```

- **Comportamiento:** realiza todos los cambios necesarios en el sistema de ficheros para eliminar un directorio y su contenido.
- **Parámetros:**
  - `path` – Ruta y nombre del directorio a borrar (ej. `/Movies/Mission_Impossible"`)
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El directorio no puede ser borrado porque no existe en el sistema de ficheros.
  - `-2` – En caso de que exista cualquier otro error.

4.5. Interacción con los directorios

```
int lsDir(char *path, int inodesDir[10], char namesDir[10][33])
```

- **Comportamiento:** carga los inodos y nombres de los elementos de un directorio.
- **Parámetros:**
  - `path` – Ruta y nombre del directorio (ej. `/Movies/Mission_Impossible"`)
  - `inodesDir` – Array que la función deberá completar con los inodos de los elementos contenidos en el directorio. Las entradas no usadas se completarán con -1.
  - `namesDir` – Array con los nombres de los elementos contenidos en el directorio, siguiendo el mismo orden que en `inodesDir`.
- **Valor de retorno:**
  - Cantidad de elementos presentes en el directorio.
  - `-1` – El directorio no puede ser leído porque no existe en el sistema de ficheros.
  - `-2` – En caso de que exista cualquier otro error.

4.6. Comportamientos no definidos

Cualquier comportamiento o característica arquitectónica no especificada en este documento está sujeta al propio juicio de el/la estudiante, por lo que será considerada una *decisión de diseño*. Dado que el objetivo de esta práctica es diseñar un sistema de ficheros, las decisiones de diseño tienen un papel fundamental en la adquisición de conocimientos de el/la estudiante sobre el tema.

El/La estudiante debe indicar **de manera clara y detallada** aquellas características que diseñe para resolver *comportamientos indefinidos* o *dependientes de la implementación* (p. ej. aspectos que estén fuera del ámbito de este documento, pero a su vez no estén en conflicto con la especificación de los requisitos definida en la Sección 5). Por lo tanto, el/la estudiante debe describir todos los problemas encontrados o ambigüedades, y justificar las soluciones seleccionadas.

5. Especificación de requisitos

El estudiante debe diseñar e implementar el sistema de ficheros para conseguir la funcionalidad descrita en la Sección 4. Además, el estudiante deberá cumplir con los requisitos definidos en esta sección. Se recomienda encarecidamente que los estudiantes verifiquen el cumplimiento de los mismos.

5.1. Requisitos funcionales

F1 El sistema de ficheros soportará las siguientes funcionalidades principales:

- **F1.1** Crear un sistema de ficheros (función `mkFS`).
- **F1.2** Montar un sistema de ficheros (función `mountFS`).
- **F1.3** Desmontar un sistema de ficheros `unmountFS`).
- **F1.4** Crear un fichero dentro del sistema de ficheros (función `createFile`).
- **F1.5** Eliminar un fichero existente dentro del sistema de ficheros (función `removeFile`).
- **F1.6** Abrir un fichero existente (función `openFile`).
- **F1.7** Cerrar un fichero abierto (función `closeFile`).
- **F1.8** Leer de un fichero abierto (función `readFile`).
- **F1.9** Escribir en un fichero abierto (función `writeFile`).
- **F1.10** Modificar el puntero de posición de un fichero (función `lseekFile`).
- **F1.11** Crear un directorio dentro del sistema de ficheros (función `mkDir`).
- **F1.12** Eliminar un directorio existente dentro del sistema de ficheros (función `rmDir`).
- **F1.13** Listar el contenido de un directorio existente (función `lsDir`).

F2 Cada vez que un fichero es abierto, su puntero de posición debe reiniciarse al principio del fichero.

F3 Los metadatos del sistema deben actualizarse después de cada operación de escritura para que reflejen adecuadamente las modificaciones realizadas.

F4 Se podrá leer el contenido completo de un fichero utilizando varias operaciones `read`.

F5 Se podrá modificar parte del contenido de un fichero utilizando operaciones `write`.

F6 El sistema de ficheros podrá ser creado en particiones del dispositivo más pequeñas que su tamaño máximo.

F7 Se utilizará el carácter `'/'` como separador en una ruta, y este no puede aparecer en el nombre de ningún fichero ni directorio.

F8 Se considera la ruta del directorio raíz como `'/'`, y en el mismo pueden aparecer tanto ficheros (ej. `'/foto1'`) como otros directorios, que a su vez podrán estar anidados (ej. `'/music/soundtracks/'`).

5.2. Requisitos no funcionales

NF1 El máximo número de ficheros en el sistema de ficheros no será nunca mayor de 40.

NF2 El máximo número de elementos (ficheros y directorios en total) en un directorio no será nunca mayor de 10.

NF3 El nombre de un fichero o directorio podrá tener como máximo una longitud de 32 caracteres.

NF4 La longitud máxima de la ruta completa (incluyendo su nombre) de un fichero será de 132 caracteres, incluyendo los separadores (`'/'`). En el caso de un directorio será de 99 caracteres, incluyendo los separadores (`'/'`).

NF5 La profundidad máxima de una jerarquía de directorios no superará 3 niveles.

NF6 El tamaño máximo de un fichero será de un bloque.

NF7 El tamaño de bloque del sistema será de 2048 bytes.

NF8 Los metadatos del sistema deben persistir entre operaciones de `desmontaje` y `montaje`.

NF9 El sistema de ficheros será usado en discos de 50 KiB a 10 MiB.

NF10 Deberá minimizarse el espacio en disco del tamaño de los metadatos del sistema.

NF11 Deberán aprovecharse al máximo los recursos disponibles.

5.3. Requisitos de implementación

I1 El código enviado debe funcionar en el servidor *guernika* y en las máquinas virtuales dadas para los cuadernos de prácticas. El/la estudiante debe compilar y probar su código en uno de ellos para el correcto desarrollo de la práctica[1].

I2 El código debe compilar con los flags `-Werror` y `-Wall` para que la práctica sea evaluada. Los programas que no compilen tendrán una calificación de cero.

I3 Los ficheros `blocks_cache.h`, `blocks_cache.c`, `filesystem.h`, `create_disk.c` y `Makefile` no pueden ser modificados bajo ningún concepto[2].

I4 El dispositivo de almacenamiento será emulado mediante un fichero con nombre `disk.dat`. El uso de cualquier otro fichero está terminantemente prohibido.

I5 La implementación del sistema de ficheros deberá ajustarse a la interfaz cliente definida en la Sección 4.

I7 El estudiante solo declarará funciones auxiliares en el fichero de cabecera `auxiliary.h`.

I8 Tanto las funciones de la interfaz del sistema de ficheros como las funciones auxiliares definidas por el/la estudiante deberán implementarse en el fichero `filesystem.c`.

I9 El estudiante declarará tipos de datos auxiliares, constantes y estructuras en el fichero de cabecera `metadata.h`.

[1] Si hay algún problema con este aspecto, se recomienda ponerse en contacto con su profesor de prácticas.

[2] Se recomienda contactar con el profesor de prácticas en el caso de que se encuentre cualquier posible bug o problema en estos ficheros **ANTES** de que los estudiantes los modifiquen por ellos mismos.

5.4. Requisitos de documentación

D1 Cada función en el código debe estar correctamente comentada, enfatizando los detalles de implementación más complejos y los posibles comportamientos definidos por el estudiante. Los programas sin comentar serán penalizados en la calificación.

D2 La memoria debe incluir, al menos:

- **D2.1** Portada con el nombre de los autores y NIA.
- **D2.2** Índice de contenidos.
- **D2.3** Diseño detallado del sistema de ficheros, incluyendo descripción de alto nivel de la funcionalidad principal, supuestos, estructuras de datos, algoritmos y optimizaciones.
- **D2.4** Diseño de un plan de pruebas para verificar que una implementación se ajusta al diseño del sistema de ficheros previamente propuesto.
- **D2.5** Conclusiones, describiendo los principales problemas encontrados y cómo han sido resueltos. Opcionalmente, se pueden incluir conclusiones personales.

D3 *Toda* decisión de diseño debe ser contextualizada y justificada según los conceptos teóricos de la asignatura.

D4 No se debe incluir ningún código fuente en la memoria.

D5 No se deben incluir capturas de pantalla de código en la memoria.

D6 Cada caso de prueba debe especificar su objetivo, procedimiento, entrada y salida esperada. Opcionalmente, el/la estudiante puede indicar si su implementación pasa o no cada caso de prueba.

D7 Todas las páginas deben estar numeradas, excepto la portada.

D8 El texto debe estar justificado.

D9 La memoria no debe superar las 15 páginas, incluyendo portada, índice, figuras, tablas y referencias. Cualquier contenido que supere las 15 páginas será ignorado.

5.5. Requisitos de entrega

E1 La práctica debe enviarse a través de Aula Global. La fecha límite estará indicada en el entregador y notificada convenientemente. No se permite la entrega de la práctica por correo electrónico sin autorización previa.

E2 El código y la memoria deben entregarse por separado.

E3 La memoria debe entregarse mediante Turnitin en formato PDF. Solo memorias en PDF serán corregidas.

E4 La entrega deberá ajustarse a los siguientes nombres, siendo AAAAAAAAA, BBBBBBBBB y CCCCCCCCC los NIA de los tres integrantes del grupo:

- **E4.1** Memoria: `dssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCCC.pdf`
- **E4.2** Código: `dssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.zip`

E5 El fichero comprimido (.zip) debe contar con los siguientes ficheros:

E5.1 Implementación del sistema de ficheros: `filesystem.c`

E5.2 Implementación del plan de pruebas: `test.c`

E5.3 Carpeta completa con todos los ficheros de cabecera: `include`

E6 Los grupos deben estar formados por un máximo de tres estudiantes.

E7 El cumplimiento de los requisitos debe haberse verificado antes del envío.

6. Otras consideraciones

Además, los siguientes aspectos deben ser considerados por parte de el/la estudiante:

- Dos sistemas de ficheros que pasen las mismas pruebas no es seguro que tengan la misma nota. Dependerá en gran medida del diseño realizado.
- La memoria es una parte importante de la nota de la práctica, ya que es donde se describe el diseño realizado. No se debe descuidar la calidad de la misma.
- Se recomienda evitar pruebas duplicadas que tengan como objetivo evaluar partes de código con similares parámetros de entrada. El plan de pruebas se evaluará dependiendo de su alcance, no del número de pruebas.
- Que la práctica compile sin errores y sin avisos no garantiza que la implementación cumpla con los requisitos funcionales. Se recomienda probar y depurar el código para asegurar su correcto funcionamiento.
- Las prácticas enviadas deben contener trabajo original. En caso de que se detecte un caso de copia entre dos prácticas, los miembros de ambos grupos suspenderán la evaluación continua, además de que se aplicarán los procedimientos administrativos correspondientes a mala conducta.
- Se puede utilizar el siguiente comando para crear el fichero comprimido con el código fuente:

```
$ zip -r dssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCCC.zip .
```