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

Diseño de Sistemas Operativos

Práctica 2:
Sistema de Ficheros

Diseño de Sistemas Operativos

Elías del Pozo Puñal
Cristhian Martínez Rendón
José Rivadeneira López-Bravo

8, Abril, 2020

Contents

1 Introducción **2**

2 Evaluación **3**

3 Código inicial **4**

4 Especificación de la funcionalidad **7**
4.1 Funcionalidad básica 7
4.1.1 Gestión del dispositivo 7
4.1.2 Gestión de ficheros 8
4.1.3 Interacción con ficheros 10
4.2 Implementación de integridad 11
4.3 Enlaces simbólicos 13
4.4 Comportamiento no definido 14

5 Especificación de requisitos **15**
5.1 Requisitos funcionales 15
5.2 Requisitos no funcionales 16
5.3 Requisitos de implementación 16
5.4 Requisitos de documentación 17
5.5 Requisitos de entrega 18

6 Otras consideraciones **19**

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 validad 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:

1. **Memoria** (*3 puntos*)
    - **Diseño del sistema de ficheros:** (*2 puntos*)
    - **Plan de validación** (*1 puntos*)
2. **Implementación** (*7 puntos*)
    - **Primer apartado:** (*4 puntos*)
    - **Segundo apartado:** (*2 puntos*)
    - **Tercer apartado:** (*1 punto*)

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. **filesystem/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. **filesystem/blocks_cache.c**: implementación de las funciones para leer y escribir bloques de disco.
  3. **filesystem/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. **zlib/crc32.c**: implementación de la función crc32 necesaria para el cálculo de los crc para comprobar la integridad del sistema de ficheros. Esta hace uso de diferentes funciones que se encuentran definidas dentro de la lbreria de compresión denominada zlib.
  5. **zlib/crc32.h**: Contiene las tablas para el cálculo de los crc.
  6. **zlib/zlib.h**: Define la cabecera de la función crc32, la cual es utilizada para el cálculo de crc en este sistema de ficheros.
  7. **zlib/zutil.h**: Define diferentes macros y constantes necesarias para el cálculo de los crc.
  8. **filesystem/crc.h**: declaración de las funciones para el cálculo de la integridad de los bloques.Este fichero contiene la definición de tres funciones una para el cálculo de un crc de 16 bits, otra para el cálculo de un crc de 32 bits y otra para uno de 64 bits.
  9. **filesystem/crc.c**: implemetación de las funciones para el cálculo de los crc.

     Para el cálculo de los crc de los ficheros se debe hacer uso de la función (**uint32_t CRC32(const unsigned char* buffer, unsigned int length)**) definida dentro del fichero crc.h.
     A continuación se muestra un ejemplo de uso:

     ```
     char *buffer = "Esto es una prueba\n";
     uint32_t val = CRC32(buffer, strlen(buffer));
     printf("%x\n", val);    // CRC de 32 bits
     ```

  10. **create_disk.c**: crea el fichero `disk.dat` con diferentes tamaños.
  11. **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/filesystem.c**: implementación de las funciones que representan la interfaz del sistema de ficheros. El/La 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. **filesystem/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_ (del 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 de las características del sistema. Este fichero debe ampliarse para que incluya las pruebas creadas por el/la 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, crc.h y filesystem.h (p. ej. nombre, parámetros, tipo de valor de retorno).

La documentación interna de los ficheros de cabecera aporta información detallada sobre los valores devueltos por las funciones proporcionadas. La/El

estudiante debe consultar esta información antes de comenzar a implementar su diseño.

4 Especificación de la funcionalidad

El desarrollo de esta práctica estará divido en tres partes incrementales, por lo que, se recomienda resolver previamente las partes anteriores antes de pasar a la siguiente.

La/El estudiante debe diseñar e implementar desde cero un sistema de ficheros capaz de gestionar un dispositivo de almacenamiento emulado (**disk.dat**), para ello tendrá que crear y dar soporte a las operaciones básicas de un sistema de ficheros(`mount, open, read, write, etc.`).

En segundo lugar, la/el estudiante añadirá mecanismos de integridad de datos para detectar corrupción de datos a nivel de fichero. Esta funcionalidad extra es clave en los sistemas de ficheros, ya que a lo largo del ciclo de vida del sistema de ficheros pueden ocurrir fallos de hardware o corrupción de información. Para llevar a cabo esta tarea, la/el estudiante utilizará la funcionalidad CRC que se ofrece en el código base.

Para finalizar, el/la estudiante deberá añadir la funcionalidad para crear y eli-minar enlaces blandos.

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 Funcionalidad básica

La primera parte de la práctica consiste en la realización de la funcionalidad que permita la gestión del dispositivo (descrito en la sección 4.1.1). En segundo lugar la/el estudiante, debe implementar las funciones que permite la gestión de los ficheros dentro del disco (descrito en la sección 4.1.2). Y para finalizar debe realizar las funciones que permiten la interacción con los ficheros (descrito en la sección 4.1.3).

4.1.1 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.

4.1.2 Gestión de ficheros

```
int createFile(char *fileName)
```

- **Comportamiento:** realiza todos los cambios necesarios en el sistema de ficheros para crear un nuevo fichero vacío.
- **Parámetros:**
  - `filename` – Nombre del fichero a crear.
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El fichero no ha podido ser creado porque ya existe en el sistema de ficheros.
  - `-2` – En caso de otros errores.

```
int removeFile(char *fileName)
```

- **Comportamiento:** realiza todos los cambios necesarios en el sistema de ficheros para eliminar un fichero.
- **Parámetros:**
  - `filename` – Nombre del fichero a eliminar.
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El fichero no ha podido ser eliminado porque no existe en el sistema de ficheros.
  - `-2` – En caso de otros errores.

```
int openFile(char *fileName)
```

- **Comportamiento:** Abre un fichero existente en el sistema de ficheros e inicializa su puntero de posición al principio del fichero.
- **Parámetros:**
  - `filename` – Nombre del fichero a abrir.
- **Valor de retorno:**
  - El descriptor de fichero del fichero abierto.
  - `-1` – El fichero no ha podido ser abierto porque no existe en el sistema de ficheros.
  - `-2` – En caso de otros errores.

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

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

4.1.3 Interacción con 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 escritor en el fichero.
  - `numBytes` – Número de bytes a escribir del buffer.
- **Valor de retorno:**

- Número de bytes escritos correctamente. Si, por ejemplo, quedan 150 bytes por escribir para llegar al tamaño máximo del fichero y se quieren escribir 400 bytes, se devolverá 150 bytes, que son los que se han escrito hasta llegar al máximo del fichero. En el caso de que el puntero de posición esté al final del máximo número de bytes del tamaño máximo del fichero se devolverá 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 de 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.
  - **NOTA:** Al contrario que POSIX, 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, ya que no se debe tener en cuenta el parámetro `offset`).
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – En caso de error.

4.2 Implementación de integridad

Con el objetivo de comprobar que un fichero no se ha corrompido, o de que un virus nos lo haya modificado se disponen de las siguientes funciones que permiten comprobar la integridad de los diferentes ficheros. La comprobación de la integridad de los ficheros solo se puede hacer sobre ficheros que ya existan en el sistema. Para ello, se usarán las funciones proporcionadas de CRC.

```
int checkFile(char *fileName)
```

- **Comportamiento:** Verifica la integridad del fichero.
- **Parámetros:**
  - `fileName` – Nombre del fichero a verificar.
- **Return:**
  - `0` – La ejecución ha sido un éxito y el fichero es correcto.
  - `-1` – La ejecución ha sido un éxito pero el fichero está corrupto.
  - `-2` – En caso de error. Intentar ejecutar esta función cuando el fichero está abierto se considera error.

```
int includeIntegrity(char *fileName)
```

- **Comportamiento:** Añade integridad a un fichero que no tuviese anteriormente.
- **Parámetros:**
  - `fileName` – Nombre del fichero a añadir integridad utilizando CRC.
- **Return:**
  - `0` – La ejecución ha sido un éxito.
  - `-1` – Fichero no existe.
  - `-2` – Otros errores, por ejemplo, el fichero ya tiene integridad.

```
int openFileIntegrity(char *fileName)
```

- **Comportamiento:** Abre un fichero existente en el sistema de ficheros e inicializa su puntero de posición al principio del fichero, esta función debe comprobar la integridad del fichero, para ello llamará a la función `checkFile`. Esta función es igual a la función `openFile` pero se le añade comprobación de integridad.
- **Parámetros:**
  - `filename` – Nombre del fichero a abrir y a comprobar la integridad.
- **Valor de retorno:**
  - El descriptor de fichero del fichero abierto si la integridad es correcta.
  - `-1` – El fichero no ha podido ser abierto porque no existe en el sistema de ficheros.
  - `-2` – Fichero corrupto.
  - `-3` – Otros errores, por ejemplo, el fichero no tiene integridad (nunca se le ha calculado).

```
int closeFileIntegrity(int fileDescriptor)
```

- **Comportamiento:** Cierra un fichero abierto y actualiza el crc del mismo.
- **Parámetros:**
  - `fileDescriptor` – Descriptor de fichero del fichero a cerrar.
- **Valor de retorno:**
  - `0` – La ejecución es correcta y el crc se ha actualizado correctamente.
  - `-1` – En caso de error.

4.3 Enlaces simbólicos

Para finalizar en este sistemas de ficheros también queremos implementar los enlaces blandos para tener una alternativa de acceso a los ficheros ya creados y facilitar su uso, para ello se pide que se creen las siguientes funciones.

```
int createLn(char *fileName, char *linkName)
```

- **Comportamiento:** Crea un enlace simbólico a un fichero existente en el sistema de ficheros.
- **Parámetros:**
  - `fileName` – Nombre del fichero a enlazar.
  - `linkName` – Nombre del enlace simbólico.
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El fichero no ha podido ser enlazado porque no existe en el sistema de ficheros.
  - `-2` – En otro caso de error.

```
int removeLn(char *linkName)
```

- **Comportamiento:** Elimina un enlace simbólico del sistema de ficheros.
- **Parámetros:**
  - `linkName` – Nombre del enlace simbólico
- **Valor de retorno:**
  - `0` – La ejecución es correcta.
  - `-1` – El enlace no ha podido ser eliminado porque no existe en el sistema de ficheros.
  - `-2` – En otro caso de error.

4.4 Comportamiento no definido

El/La estudiante debe tener presente que se puede acceder a todos los ficheros, ya sea para lectura, escritura, comprobación de integridad, usando el fichero original o usando los enlaces simbólicos que se hayan creado. 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/la estudiante debe diseñar e implementar el sistema de ficheros para conseguir la funcionalidad descrita en la Sección 4. Además, el/la estudiante deberá cumplir con los requisitos definidos en esta sección. Se recomienda encarecidamente que los/las 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 (función `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** Comprobar la integridad de un fichero existente (función `checkFile`).
- **F1.12** Añade integridad a un fichero existente (función `includeIntegrity`).
- **F1.13** Apertura de un fichero comprobando su integridad (función `openFileIntegrity`).
- **F1.14** Cierre de un fichero actualizando el crc del mismo (función `closeFileIntegrity`).
- **F1.15** Crear un enlace simbólico (función `createLn`).
- **F1.16** Eliminar un enlace simbólico (función `removeLn`).

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 cierre de fichero para que reflejen adecuadamente las modificaciones realizadas. Los metadatos relacionados con la integridad solamente se actualizarán en la parte correspondiente a la integridad.

F4 El sistema de ficheros no implementará directorios..

F5 La integridad de los ficheros se debe comprobar, al menos, en operaciones de `apertura`.

F6 Se podrá leer el contenido completo de un fichero utilizando varias operaciones `read`.

F7 Deberá poder extenderse la capacidad de un fichero mediante bloques extra utilizando operaciones `write`.

F8 El sistema de ficheros podrá ser creado en particiones del dispositivo más pequeñas que su tamaño máximo.

5.2 Requisitos no funcionales

NF1 El máximo número de ficheros en el sistema de ficheros no será nunca mayor de 48.

NF2 La longitud máxima de un nombre de fichero y enlace simbólico será de 32 caracteres.

NF3 El tamaño máximo de un fichero será de 10KiB.

NF4 El tamaño de bloque del sistema será de 2048 bytes.

NF5 Los metadatos del sistema deben persistir entre operaciones de `desmontaje` y `montaje`

NF6 El sistema de ficheros será usado en discos de 460 KiB a 600 KiB.

NF7 Deberán aprovecharse al máximo los recursos disponibles. Por ejemplo, deberá minimizarse el espacio en disco del tamaño de los metadatos del sistema.

NF8 Un fichero sin integridad solo debe poderse abrir con la función de abrir sin integridad.

NF9 La integridad de un fichero se calculará siempre en el cierre de este (Función closeFileIntegrity).

NF10 Si se quiere abrir un fichero sin integridad con las funciones de integridad, deberá añadirse su integridad previamente (función `includeIntegrity`).

NF11 Un fichero no se puede abrir con la función de integridad (openFileIntegrity) y cerrarse sin integridad (closeFile).

NF12 Un fichero no se puede abrir con la función sin integridad (openFile) y cerrarse con ella (closeFileIntegrity).

5.3 Requisitos de implementación

I1 El código enviado debe funcionar en el servidor `guernika` y/o en la máquina virtual proporcionada en AG. El/la estudiante debe compilar y probar su código en `guernika` para el correcto desarrollo de la práctica [1].

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

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`, `Makefile`, `crc.h`, y `crc.c`, al igual que los ficheros incluidos en la capeta `zlib`, 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.

I6 La integridad de los datos se comprobará mediante la funcionalidad CRC proporcionada en el fichero crc.c.

I7 El/la 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/la estudiante declarará tipos de datos auxiliares, constantes y estructuras en el fichero de cabecera metadata.h.

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/la estudiante. Los programas sin comentar serán penalizados en la calificación.

D2 La memoria debe incluir, como mínimo:

- **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 supuestos, estructuras de datos, algoritmos y optimizaciones.
- **D2.4** Descripción de alto nivel de la funcionalidad principal.
- **D2.5** Diseño de un plan de pruebas para verificar que una implementación se ajusta al diseño del sistema de ficheros previamente propuesto.

[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/las estudiantes los modifiquen por ellos mismos.

D2.6 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. Si se incluye, será ignorado.

D5 No se deben incluir capturas de pantalla de código en la memoria. Si se incluyen, serán ignoradas.

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 y con tamaño de letra 12.

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. El texto de la memoria deberá ser seleccionable.

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.

6 Otras consideraciones

Además, los siguientes aspectos deben ser considerados por parte del 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 -9 -r dssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCCC filesystem.c
test.c include/*
```