Grado en Ingeniería Informática

Diseño de sistemas operativos

Práctica 2:

Sistema de ficheros

Javier Mora Argumánez (100383465)

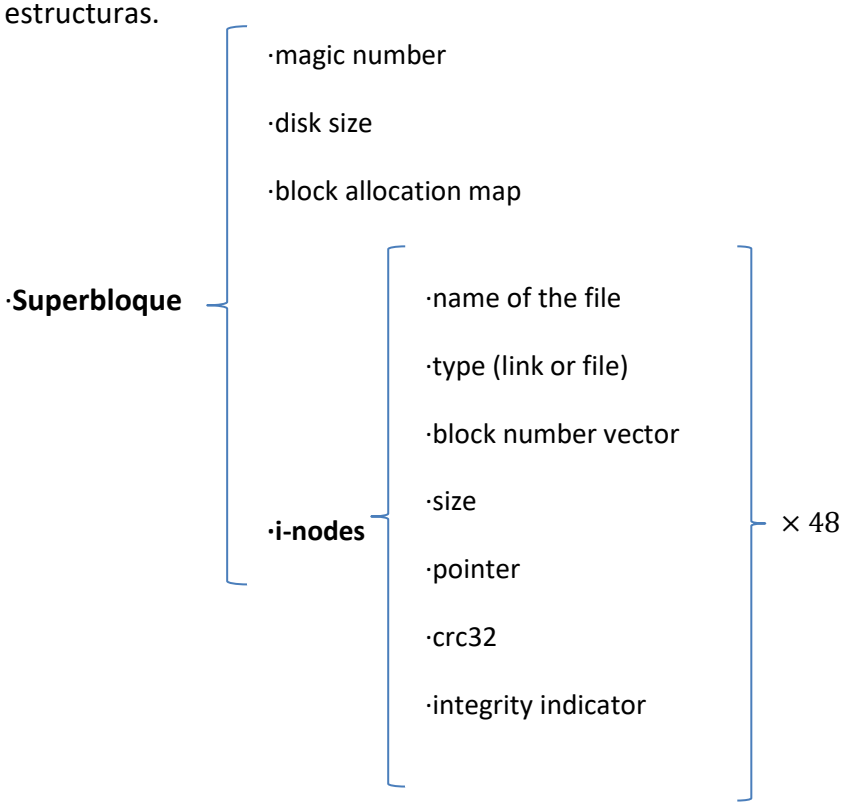
Luis Daniel Sánchez Leva (100383388)

Tabla de contenido

·Diseño del sistema de ficheros

·Estructuras de datos

Las principales estructuras de datos utilizadas son el **superbloque** y los **i-nodos**. Estos últimos contienen la información de cada uno de los archivos, y están contenidos en un vector que forma parte del superbloque. En el siguiente esquema se puede observar de manera concisa cómo están constituidas estas estructuras.



Cada una de estas propiedades está basada en distintos tipos de datos y contiene diferente información:

- **Magic number:** Se trata de un número de identificación para el sistema de ficheros diseñado, que permite a sistemas de montado reconocer este sistema, sabiendo así cómo interpretar el superbloque. Es de tipo *uint16_t.*
- **Disk size:** Tamaño del disco. Es de tipo *long*.
- **Block allocation map:** Se trata de un vector de booleanos que indica cuando un bloque está en uso, o si está libre. Su existencia permite que

no sea necesario borrar el contenido de los bloques cuando se borra un archivo, simplemente pudiendo marcar los bloques de este como libres.

- **I-nodes:** Cada uno de ellos representa un archivo o enlace blando. Contiene la información de este.
- **Name of the file**: Contiene el nombre del archivo. Es de tipo char *.
- **Type:** Define si se trata de un archivo o de un enlace blando. Es de tipo uint8_t.
- **Block number vector:** Se trata de un vector de int, que contiene el número de los bloques en los que se distribuye un fichero. Al ser un sistema en el que los bloques de un archivo en concreto no son contiguos, es necesario almacenar los cinco posibles bloques. Cuando un elemento del vector no corresponde a ningún bloque, se inicializa en 255.
- **Size:** Contiene el tamaño del fichero en bytes. Es de tipo uint16_t.
- **Pointer:** Contiene el byte del archivo al que se está apuntando. Al abrir un archivo, se coloca al principio de este. Leer o escribir mueven el puntero una cantidad de bytes igual a los leídos o escritos. Puede también ser modificado mediante la función lseekFile. Es de tipo uint16_t.
- **Crc32:** Contiene el identificador del archivo para la verificación por redundancia cíclica. Esta inicializado de manera normal en 0 (ya que no se puede añadir integridad a un archivo vacío) y cambia cuando se ejecuta la función includeIntegrity sobre dicho archivo. Este parámetro se contrasta con el resultado de calcular el crc al leer un archivo cuando este se abre con integridad, y se actualiza cuando el archivo se cierra con integridad.
- **Integrity indicator:** Se trata de un indicador de tipo int, una variable que controla si un archivo ha sido abierto con integridad. Su utilidad es la de asegurar que se cumplen los requisitos NF11 y NF12.

En total, estas estructuras de datos dotan al superbloque de un tamaño menor de 2048 bytes, permitiendo que este se escriba en un solo bloque del disco. Teniendo esto en cuenta, junto con el hecho de que los bloques ocupan 2048 bytes, y que el tamaño máximo de los archivos es de 48KiB, el tamaño del disco debe ser de 241 bloques (. /create_disk 241).

Además de estas estructuras, se han creado también una estructura adicional en memoria:

- **File state vector:** Contiene el estado (abierto o cerrado) de cada uno de los i-nodos. No fue necesario incluirlo en el superbloque, ya que sólo es relevante mientras el sistema de archivos corra, y puede inicializarse cada vez que este arranque.

·Algoritmos

Debido al alcance del proyecto, se detallará en esta sección una explicación de muy alto nivel de cada una de las funcionalidades del sistema de ficheros:

- **makeFS:** Se trata de la función que inicializa las variables del sistema de archivos. Una vez ha asignado los valores por defecto de cada una de ellas, llama a la función **unmountFS** para escribir este superbloque inicializado a disco.
- **mountFS:** Montar el sistema de archivos, es decir, leer el superbloque del disco a memoria.
- **unmountFS:** Desmontar el sistema de archivos, es decir, escribir el superbloque de memoria al disco.
- **getFreeBlock:** Se trata de una función auxiliar que itera por el mapa de asignación de bloques, devuelve el primer bloque libre y lo etiqueta como ocupado.
- **createFile:** Itera a través de los i-nodos hasta encontrar uno libre. En este caso, un i-nodo libre se reconoce como uno cuyo nombre está vacío, dado que no hay directorios y no es posible crear archivos sin nombre. Esta decisión hace el sistema más sencillo, pues la necesidad de incluir un mapa de i-nodos haría que el superbloque excediese los 2048 bytes y tuviera que ser escrito a lo largo de dos bloques. Una vez se localiza un i-nodo libre, se inicializa el primer bloque del archivo, así como el nombre de este.
- **removeFile:** Itera a través de los i-nodos hasta encontrar el archivo de nombre especificado. A continuación, devuelve todos los campos del i-nodo a sus valores por defecto y libera los bloques que estuviese utilizando este archivo. Estos bloques no son borrados, simplemente se marcan como libres y otro archivo puede escribir por encima de ellos. Dado que no es posible que se lea un archivo más allá de su tamaño total, nunca se accede al contenido que pueda quedar en el bloque. Esta decisión optimiza el uso de recursos.
- **openFile:** Itera a través de los i-nodos hasta encontrar el fichero especificado, lo marca como abierto en el vector de estado del fichero, y

devuelve su descriptor. Si se trata de un enlace simbólico, marca y devuelve el descriptor del archivo al que este enlaza.

- **closeFile:** Marca el archivo especificado como cerrado y actualiza el superbloque en disco mediante una operación unmount.
- **readFile**: Lee un fichero abierto, copiando un número de bytes dado a un buffer, empezando desde el puntero de dicho fichero. Debido a que la única manera de acceder al disco es mediante la operación bread, es necesario calcular el offset del puntero respecto al principio del bloque, así como el bloque desde el que se empieza a leer. Una vez hecho esto, se leen todos los bloques a lo largo de los que se distribuya el archivo, y luego se utilizan operaciones memcpy para copiar al buffer dado por el usuario únicamente la información deseada. En caso de que el archivo no llegue hasta el final de los bytes dados, lee hasta donde sea posible. Devuelve el número de bytes leídos con éxito.
- **writeFile**: Escribe un número dado de bytes desde un buffer a un fichero abierto, empezando desde su puntero. Debido a que la única manera de acceder al disco es mediante la operación bwrite, es necesario calcular el offset del puntero respecto al principio del bloque, así como el bloque desde el que se empieza a escribir. Una vez hecho esto, se leen los bloques sobre los que se vaya a escribir, y se utiliza el offset calculado para modificar sólo la parte especificada. Finalmente, se escriben los bloques modificados al disco. En caso de que lo que se intenta escribir no quepa en el fichero, escribe hasta donde sea posible. Devuelve el número de bytes escritos con éxito.
- **lseekFile:** Mueve el puntero de un archivo. Puede moverlo al principio de este, a su final o desplazarlo un offset desde su posición actual.
- **checkFile:** Dado un nombre de fichero, comprueba que sea válido para crear un fichero con dicho nombre.
- **includeIntegrity:** Genera el identificador de verificación de redundancia cíclica de un archivo que no esté vacío.
- **openIntegrity:** Abre un fichero con identificador de integridad, comprobando que si este se computa de nuevo, coincide con el almacenado, evitando abrir ficheros corruptos.
- **closeIntegrity:** Cierra un fichero que haya sido abierto mediante la función **openIntegrity** y actualiza su identificador de integridad.
- **createLn:** Crea un enlace simbólico blando a otro archivo.
- **removeLn:** Elimina un enlace simbólico blando.

·Plan de pruebas

·Consideraciones para el plan de pruebas

El plan de pruebas se ha diseñado intentando ajustarse a las indicaciones dadas en el enunciado de la práctica. Es por esto que, aunque hemos intentado ser lo más rigurosos posibles, hemos omitido tests como los de boundary values dado que se recomienda evitar pruebas duplicadas que tengan como objetivo evaluar partes de código con similares parámetros de entrada. A pesar de esto, se han examinado numerosos casos de errores para asegurar la correcta implementación del código.

Además de las pruebas especificadas en la siguiente sección, se han comprobado las funciones de lectura y escritura mediante el uso de hexedit para leer los bytes del disco y compararlos con los de los buffers.

·Pruebas realizadas

Test	Descripción	Retorno esperado
mkFs under min size	Se ejecuta mkFs con un tamaño de disco menor que el mínimo	-1
mkFs over max size	Se ejecuta mkFs con un tamaño de disco mayor que el máximo	-1
mkFS	Se ejecuta mkFs con un tamaño de disco válido	0
mountFS	Se ejecuta mountFs	0
unmountFS	Se ejecuta unmountFS	0
createFile empty name	Se ejecuta createFile con el parámetro “”	-2
createFile name too long	Se ejecuta createFile con un nombre de fichero demasiado largo	-2
createFile	Se ejecuta createFile con un nombre de fichero válido	0
writeFile on a closed file	Se ejecuta writeFile sobre un fichero cerrado	-1
openFile	Se ejecuta openFile sobre el fichero creado anteriormente	Descriptor del fichero
writeFile	Se desplaza el puntero del archivo abierto en un offset de 10 bytes, y se escribe el buffer **A** en este. Este archivo fue además comprobado en hexedit.	Número de bytes escritos.
readFile	Se desplaza el puntero del archivo abierto en un offset de 10 bytes desde su inicio, y se lee su contenido al buffer **B.**	Número de bytes leídos
read-write consistency.	Se realiza un memcmp de los buffers **A** y **B**.	0
closeFile	Se ejecuta closeFile sobre el archivo abierto	0
includeIntegrity	Se ejecuta includeIntegrity sobre el archivo cerrado	0
openFileIntegrity	Se ejecuta openFileIntegrity sobre un archivo íntegro	Descriptor del fichero
openFileIntegrity on corrupted file	Se “corrompe” un archivo, y se ejecuta openFileIntegrity sobre él.	-2
openFileIntegrity on non-included file	Se ejecuta openFileIntegrity sobre un fichero sobre el cuál no se ha ejecutado includeIntegrity previamente.	-1
includeIntegrity on empty file	Se ejecuta includeIntegrity sobre un fichero vacío.	-2
closeFileintegrity	Se ejecuta closeFileIntegrity en un archivo abierto con integridad	0

createLn	Se ejecuta createLn sobre un archivo, con un nombre de enlace válido	0
openFile on link	Se ejecuta openFile sobre el enlace creado	Descriptor del archivo al que apunta
closeFile on linked file	Se ejecuta closeFile sobre el fichero cuyo link ha sido abierto	0
openFile on linked file	Se ejecuta openFile en un fichero cuyo link ha sido cerrado	Descriptor del archivo
removeFile on open file	Se ejecuta removeFile sobre un fichero abierto	-1
removeFile	Se ejecuta removeFile sobre un fichero cerrado	0
removeFile on removed file	Se ejecuta removeFile sobre un fichero borrado	-1
removeLn	Se ejecuta removeLn en un enlace	0

Todas estas pruebas han sido implementadas en el fichero test.c y probadas en guernika. Todas ellas han sido pasadas con éxito.