Simulación de un Sistema de Archivos en C

Pablo Mesén Alvarado y Luis Urbina Salazar

Principios de Sistemas Operativos – Instituto Tecnológico de Costa Rica

1. Introducción

El presente documento describe el diseño e implementación de un **sistema de archivos simulado en memoria** desarrollado en el lenguaje C, como parte del curso *Principios de Sistemas Operativos*.

El objetivo principal es modelar, de forma sencilla, cómo un sistema de archivos administra bloques de datos, mantiene una tabla de archivos (directorio raíz) y ofrece operaciones básicas como `CREATE`, `WRITE`, `READ`, `DELETE` y `LIST`. El programa lee un archivo de texto con instrucciones y ejecuta cada operación sobre una estructura de datos que representa un "disco" en memoria.

2. Descripción general del sistema

El sistema de archivos se implementa completamente en memoria, sin acceso al disco real. Para lograrlo se definen:

- Un arreglo bidimensional que simula los bloques de datos del disco.
- Una tabla de archivos que actúa como directorio raíz.
- Un arreglo auxiliar para marcar qué bloques están libres u ocupados.

El flujo general de ejecución es el siguiente:

1. El programa se ejecuta recibiendo como argumento la ruta de un archivo de texto.
2. La función `read_test_file()` abre el archivo y lee línea por línea.
3. Cada línea se interpreta como una instrucción (`CREATE`, `WRITE`, `READ`, `DELETE`, `LIST` o comentario).
4. Para cada instrucción se llama a la función correspondiente que modifica o consulta las estructuras internas del sistema de archivos.

De esta forma se separa claramente la *entrada* (archivo de comandos) de la *lógica* de manejo de archivos simulados.

3. Estructuras de datos utilizadas

A continuación se describen brevemente las estructuras de datos centrales del diseño.

3.1. Bloques de datos

El "disco" se modela mediante:

```
#define BLOCK_SIZE 512
#define NUM_BLOCKS /* valor definido en includes.h */

unsigned char data_blocks[NUM_BLOCKS][BLOCK_SIZE];
```

Cada fila de `data_blocks` representa un bloque físico de tamaño fijo `BLOCK_SIZE` bytes. Todo el contenido de los archivos se almacena en este arreglo.

3.2. Mapa de bloques usados

Para saber qué bloques están libres u ocupados se utiliza:

```
int blocks_used[NUM_BLOCKS]; // 0 = libre, 1 = ocupado
```

Esta estructura funciona como un bitmap simplificado. Las funciones auxiliares `count_free_blocks()` y `allocate_blocks()` utilizan este arreglo para contar bloques libres y reservarlos cuando se crea un archivo.

3.3. Tabla de archivos (directorio raíz)

Cada archivo se representa mediante la estructura `FileEntry` (definida en `includes.h`):

```
typedef struct {
    char name[MAX_FILENAME];
    int size;           // tama o l gico (bytes escritos)
    int max_size;       // capacidad m xima reservada
    int blocks[MAX_BLOCKS_PER_FILE]; // bloques f sicos asignados
    int num_blocks;     // cantidad de bloques asignados
    int used;           // 0 = entrada libre, 1 = archivo v lido
} FileEntry;

FileEntry file_table[MAX_FILES];
```

Esta tabla cumple el rol de **directorio raíz**. No se manejan subdirectorios; todos los archivos se almacenan en una lista plana. Cada entrada indica el nombre del archivo, su tamaño real, su tamaño máximo y los índices de los bloques físicos donde se encuentra su contenido.

La decisión de almacenar la lista de bloques en un arreglo por archivo permite manejar archivos cuyos bloques **no son contiguos** en el disco simulado.

4. Principales decisiones de diseño

4.1. Asignación no contigua de bloques

En lugar de requerir que los bloques de un archivo sean contiguos en el arreglo `data_blocks`, se eligió una asignación no contigua. Cada `FileEntry` mantiene un arreglo `blocks[]` con los índices de bloques físicos asignados, en orden lógico.

Esta decisión simplifica la reutilización de espacios libres: un archivo puede usar cualquier bloque que esté disponible, sin necesidad de mover datos ni compactar.

4.2. Separación entre datos y metadatos

El diseño separa claramente:

- **Datos**: se almacenan exclusivamente en `data_blocks`.
- **Metadatos**: nombre, tamaño, lista de bloques y estado del archivo se guardan en `file_table`.

Esta separación refleja la organización de sistemas de archivos reales, donde los inodos o estructuras similares almacenan metadatos, mientras que los datos residen en bloques independientes.

4.3. Procesamiento basado en archivo de comandos

En lugar de implementar un *shell interactivo*, el programa recibe un archivo de texto con comandos. Esta elección facilita la prueba y validación del sistema, ya que se pueden diseñar distintos casos de prueba simplemente editando archivos en `test_files/`.

4.4. Validación y manejo de errores

Cada operación valida cuidadosamente que:

- El archivo exista (o no exista, según la operación).
- El nombre sea válido y no se repita.
- Haya suficientes bloques libres para un `CREATE`.
- Los `offsets` y tamaños de lectura/escritura estén dentro del rango permitido.

Aunque se trata de una simulación sencilla, se buscó que el sistema se comporte de manera robusta ante casos de error comunes.

5. Funcionamiento de las operaciones principales

A continuación se resume el comportamiento de las funciones implementadas en `functions.c`.

5.1. `create_file()`

- Verifica que el nombre no sea vacío ni exceda `MAX_FILENAME`.
- Comprueba que no exista ya un archivo con ese nombre.
- Busca una entrada libre en `file_table`.
- Calcula la cantidad de bloques necesarios según el tamaño solicitado.
- Revisa si hay suficientes bloques libres.
- Llama a `allocate_blocks()` para reservarlos.
- Inicializa la entrada de archivo con nombre, tamaño máximo, número de bloques y marca `used = 1`.

5.2. `write_file()`

- Busca el archivo en la tabla.
- Verifica que el `offset` y la longitud de los datos no superen `max_size`.
- Recorre la cadena a escribir y, para cada byte, calcula:
  - Posición global dentro del archivo.
  - Índice lógico de bloque (`block_index`).
  - Desplazamiento interno dentro del bloque.
- Usa `f->blocks[block_index]` para obtener el bloque físico correspondiente y escribe en `data_blocks`.
- Actualiza el tamaño lógico `f->size` si es necesario.

5.3. `read_file()`

- Localiza el archivo por nombre.
- Valida `offset` y `size` contra los límites del archivo.
- Reserva un buffer temporal en memoria dinámica.
- Copia, byte a byte, desde los bloques físicos hacia el buffer, siguiendo el mismo cálculo de posición lógica que `write_file()`.
- Imprime el contenido leído por pantalla.

5.4. `delete_file()`

- Localiza la entrada del archivo en la tabla.
- Recorre `f->blocks[]` y marca cada bloque como libre en `blocks_used`.
- Opcionalmente limpia el contenido de `data_blocks`.
- Reinicia los campos de la estructura y marca `used = 0`.

5.5. `list_files()`

- Recorre la tabla de archivos.
- Para cada entrada en uso imprime el nombre del archivo y su tamaño.
- Si no hay archivos, muestra un mensaje indicando que el directorio está vacío.

6. Conclusiones

El sistema de archivos simulado permite visualizar de manera concreta cómo un sistema operativo podría gestionar archivos sobre un conjunto de bloques físicos, incluso cuando estos no son contiguos.

A pesar de ser una implementación simplificada, el ejercicio ayuda a comprender conceptos fundamentales como:

- La diferencia entre datos y metadatos.
- La importancia de la gestión de bloques libres.
- La validación de rangos en operaciones de lectura y escritura.
- El impacto de las decisiones de diseño (por ejemplo, asignación no contigua) sobre la flexibilidad del sistema.

Este tipo de simulaciones constituye una base importante para el estudio de sistemas de archivos reales y otros componentes de sistemas operativos.