Instituto Tecnológico de Costa Rica
Campus Tecnológico Central Cartago
Escuela de Ingeniería en Computación

Sistema de Archivos Simple

Principios de Sistemas Operativos - Grupo 2
Prof. Kenneth Roberto Obando Rodriguez

Daniel Granados Retana, carné 2022104692
Diego Manuel Granados Retana, carné 2022158363
David Fernández Salas, carné 2022045079

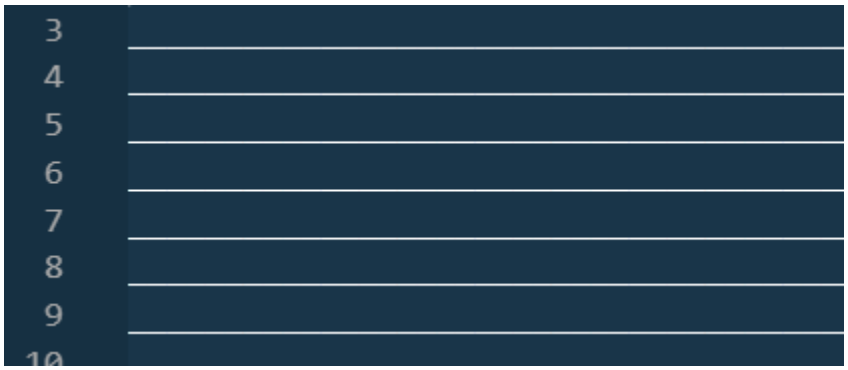
30 de Octubre del 2024
IIS 2024

Introducción

Este proyecto consiste en implementar un sistema de archivos simple en C que simula las operaciones básicas de un sistema real, como crear, escribir, leer, eliminar archivos y listar el contenido de un directorio. Mediante el uso de una tabla de archivos y bloques de datos simulados, el sistema permitirá gestionar espacio de almacenamiento, seguimiento de archivos, y manejo de errores como falta de espacio y accesos inválidos. Se implementó un sistema de archivos planos, donde todos los archivos están contenidos en un directorio de un nivel. Es decir, no hay subdirectorios.

La principal decisión tomada para el diseño del programa fue el uso de la asignación enlazada, donde los bloques tienen un puntero al final que apunta hacia el siguiente bloque del archivo. Esto permite evitar la fragmentación externa ya que, al poder tener el archivo particionado por el disco, se pueden aprovechar todos los bloques disponibles sin tomar en cuenta el tamaño del hueco. Para el manejo de los bloques libres, se utilizó una lista enlazada. Así, solo se toma la cabeza de la lista cuando se necesita un bloque adicional. No obstante, la asignación enlazada dificulta el acceso directo debido a que habría que pasar por todos los bloques anteriores para llegar al bloque “n”. Para prevenir esto, se usó un File Allocation Table (FAT), que es un arreglo del tamaño de la cantidad de bloques del disco donde cada celda representa un bloque y contiene un puntero al siguiente bloque del archivo. Así, para llegar a un cierto bloque, se opera sobre el FAT y no hace falta leer todos los bloques anteriores.

Cabe destacar que para representar que un espacio del disco está vacío o que el puntero es nulo, se usó la cadena de “__”, o dos barras bajas. Esto permite representar que el disco está vacío de una manera más visual:



Cada línea representa un bloque, por lo que tienen una longitud de 512 bytes. Al usar la asignación enlazada, los últimos 2 bytes del bloque son un puntero al siguiente bloque del archivo.

Enlace a video explicativo

https://youtu.be/3_OtC9KJpnc

Ejecución del programa

El método para ejecutar el programa es el siguiente:

1. Primero, se compila el archivo main.c:

```
gcc main.c -o main.exe
```

2. Segundo, se ejecuta en la línea de comandos, especificando el archivo de instrucciones a correr en la línea de comandos:

```
./main.exe pruebas.txt
```

Se realizaron tres archivos de pruebas:

1. pruebas.txt: Es un archivo con una gran cantidad de pruebas para probar diferentes aspectos.
2. pruebaPersistente1.txt: Es un archivo de pruebas que crea un archivo en el disco y escribo datos que abarcan varios bloques.
3. pruebaPersistente2.txt: Es un archivo que verifica que la escritura realizada en pruebaPersistente1.txt haya quedado persistente en el disco. También valida que la lectura a través de bloques se haya realizado correctamente.

En los archivos de prueba, si una línea empieza con “+”, se imprime a la salida estándar. Si más bien empieza con “#”, es como si fuera un comentario.

Diseño del programa

El programa cuenta con las siguientes variables globales las cuales permiten llevar el control de los tamaños permitidos para ciertas estructuras

```
#define DISK_SIZE 1048576
#define BLOCK_SIZE 510
#define MAX_FILES 100
```

```
#define NUM_BLOCKS 2048
#define MAX_LINE_LENGTH 1024
```

DISK_SIZE: Tiene guarda la cantidad exacta de 1mb de memoria.
BLOCK_SIZE: Mide 510 porque los próximos 2 bytes se usan para guardar la dirección del próximo bloque, así todos los bloques tienen el tamaño de 512 bytes.
MAX_FILES: Permite tener un máximo de 100 archivos
NUM_BLOCKS: Indica la cantidad máxima posible de bloques. Multiplicar este número por el tamaño de cada bloque (510 bytes) da 1,044,480, siendo este el tamaño máximo de un archivo.
MAX_LINE_LENGTH: Longitud máxima de una línea, siendo dos bloques o 1Kb.
También, se definen las siguientes dos constantes para restringir la longitud del nombre de los archivos y la cantidad que se puede escribir a la vez:

```
#define MAX_FILENAME 50
#define MAX_CONTENT_SIZE 1000
```

Estructuras necesarias

Las siguientes son *structs* de C que se usaron para representar diferentes estructuras de datos. El principal tipo de estructura de datos que se usó fue una lista doblemente enlazada.
File: Tiene el nombre, tamaño, primer bloque, y su archivo anterior y próximo. Es un nodo de la lista de archivos.
FileList: Contiene una lista de los archivos con cantidad y tamaño.
Block: Contiene su dirección y el bloque anterior y próximo. Es un nodo de la lista de archivos.
BlockList: Contiene la lista de los bloques con la cantidad siendo doblemente enlazada.

Se crean 2 instancias de “FileList”: el directorio con todos los archivos y la lista de archivos abiertos.

Se crean 3 instancias de “FILE”, que es un estructura de la librería estándar de C que contiene el handle para un archivo abierto por el sistema operativo: el disco

donde se escribe, el FATFile que cuenta con la dirección de los bloques y directoryFile para manejar los datos del directorio.

Finalmente, se crea una instancia de BlockList, la cual va a ser una lista para almacenar los bloques libres.

Cada una de estas estructuras de datos tiene su método de creación, add y delete.

Estructura FAT

La File Allocation Table (FAT) es una estructura de datos usada en sistemas de archivos que funciona como un "índice" de bloques de datos. La FAT mapea el uso de cada bloque en el disco, indicando si un bloque está libre, ocupado, o si es parte de una cadena que representa un archivo. En nuestra implementación, incorporamos el uso de esta tabla porque simplifica la forma de acceder a los bloques de información de manera directa. Se evita tener que leer todos los bloques anteriores para llegar a un bloque específico, ya que la FAT brinda una forma ordenada de conocer la secuencia de bloques de un archivo. De esta forma el uso de la FAT permite gestionar el espacio del disco virtual, permitiendo saber qué bloques están disponibles o ya ocupados por datos de archivos.

Esta implementación de la FAT es ligeramente diferente a la que presenta el libro. La versión presentada por Silberschatz et al. (2018) explica que para el último bloque de un archivo, se usa un valor especial de “End-of-File”, y para los bloques que están vacíos, se usa el valor de 0. Así, se puede obtener cuáles bloques están disponibles con solo la FAT. Nosotros la implementamos tal que ambos, un bloque vacío y uno que representa el fin de un archivo, contienen el “puntero nulo” para nuestro sistema: “5F5F” o “__”. Esto es posible ya que la gestión de los bloques libres se realiza con una lista enlazada específica, por lo que no es necesaria la distinción entre estos dos casos.

Operación Create

La operación de crear un archivo recibe por parámetro el nombre y el tamaño, donde la primera tarea es encontrar espacio libre en el disco.

```
if (size > (freeBlocks->quantity * BLOCK_SIZE) || directory->quantity
== MAX_FILES || (directory->size + size) > DISK_SIZE || freeBlock ==
NULL)
```

Si tiene suficiente espacio, la siguiente verificación que realiza es comprobar si existe otro archivo con el mismo nombre. Si tiene un nombre único, se crea una referencia para el archivo que incluye el nombre, tamaño y ubicación del bloque con base en la FAT.

```
struct File *newNode = createFileRef(name, size,freeBlock->location);
```

Después se asignan los bloques y se va actualizando el FAT donde va utilizando memoria y si necesita más toma otro bloque de memoria.

```
while (freeBlock != NULL && size > 0)
{
    size -= BLOCK_SIZE;
    if (size > 0)
    {
        FAT[freeBlock->location] = freeBlock->next->location;
        writeFAT(freeBlock->location, freeBlock->next->location);
    }
    else
    {
        FAT[freeBlock->location] = 0x5F5F;
        writeFAT(freeBlock->location, 0x5F5F);
    }
}
```

El próximo paso es calcular la posición en el archivo Disk donde se debe de escribir la entrada FAT correspondiente, la cual se calcula con la siguiente instrucción:

```
  nextBlockPointer = (freeBlock->location * BLOCK_SIZE + sizeof(uint16_t) *
freeBlock->location + freeBlock->location) + BLOCK_SIZE;
```

Esta operación lo que hace es apuntar a la posición donde se empieza a escribir el puntero al siguiente bloque. Es decir, queda en el byte 511 del bloque para escribir el puntero de 2 bytes.

Finalmente se eliminan los bloques usados de la lista de bloques libres. Después, se agrega el nuevo archivo al directorio actualizando datos de control como la cantidad de archivos y la cantidad de bytes usados.

Operación Escribir

Recibe por parámetro el nombre, offset y la información a escribir. Funciona similar a la función anterior porque primeramente recorre el directorio para encontrar el archivo con base en el nombre. Seguidamente, se hacen ciertas validaciones para conocer si se puede realizar la escritura. Después de esto, se busca el bloque y se ajusta el offset correspondiente. Cuando se encuentra el bloque, se calcula el offset para escribir en el punto exacto donde byteCount representa la cantidad de datos que se pueden escribir en el bloque.

```
int block = temp->firstBlock;
int byteCount = BLOCK_SIZE;
int blockOffset = 0;

while (block != 0x5F5F)
{
    if (byteCount >= offset)
    {
        blockOffset = BLOCK_SIZE - (byteCount - offset);
        byteCount -= offset;
        break;
    }
    block = FAT[block];
    byteCount += BLOCK_SIZE;
}
```

Justo después de validar que si existen suficientes bloques necesarios se empieza con la escritura en disco donde se calcula el desplazamiento en el disco y mediante un puntero se actualiza la información moviéndose según la cantidad de

datos escrita. Finalmente se actualiza el bloque, se configura la cantidad restante de datos y se pasa al siguiente bloque de memoria.

```
int blockPosition = (block * BLOCK_SIZE + sizeof(uint16_t) * block + block) + blockOffset;
int dataPosition = 0;

while (dataSize > 0 && block != 0x5F5F)
{
    if (fseek(disk, blockPosition, SEEK_SET) != 0)
    {
        perror("Error seeking in file");
        exit(EXIT_FAILURE);
    }
    if (fwrite(data + dataPosition, sizeof(char), byteCount, disk) != byteCount)
    {
        perror("Error writing to file");
        exit(EXIT_FAILURE);
    }
```

Operación Leer

Para la lectura se recibe por parámetro el nombre del archivo offset y tamaño de cuánto se quiere leer. Se empieza igual que los casos anteriores: recorriendo el directorio para encontrar el archivo. Se realizan ciertas validaciones con el tamaño y archivo para evitar errores. Se navega hasta el bloque correcto de la misma forma de la función anterior. Cuando se encuentra el bloque correcto, se realizan más validaciones para conocer que no haya errores y si las pasa, se empieza con la lectura del archivo.
En este caso se itera con base en los bloques, donde a partir del block Offset, se obtienen los caracteres que se van leyendo en el disco. Cuando se lee todo el bloque se pasa el siguiente hasta que el tamaño solicitado se cumpla.

```
while (size > 0 && block != 0x5F5F)
{
    int blockPosition = block * BLOCK_SIZE + sizeof(uint16_t) * block + block + blockOffset;
```

```
    if (fseek(disk, blockPosition, SEEK_SET) != 0)
    {
        perror("Error al buscar en el archivo");
        return;
    }
    for (int i = blockOffset; i < BLOCK_SIZE && size > 0; i++)
    {
        int ch = fgetc(disk);
        if (ch == EOF)
        {
            return;
        }
        putchar(ch);
        size--;
    }
```

Operación borrar

Esta función recibe por parámetro solo el nombre del archivo a borrar, donde lo primero que hace es encontrar el nombre del archivo en el directorio. Al encontrar el archivo va liberando todos los bloques de memoria que tiene asignados en la FAT y los va añadiendo a la lista de bloques libres.

```
uint16_t block = temp->firstBlock;
while (block != 0x5F5F)
{
    struct Block *newBlock = createBlockRef(block);
    addBlockToList(newBlock);
    uint16_t oldBlock = block;
    block = FAT[block];
    FAT[oldBlock] = 0x5F5F;
    writeFAT(oldBlock, 0x5F5F);
}
```

Ahora, al liberar los bloques y borrar el archivo, se debe actualizar el directorio para eliminarlo. Se copia la información en otro archivo nuevo, exceptuando el archivo a borrar.

```
deleteFileFromList(temp);
FILE *tempFile = fopen("temp.txt", "w");
if (tempFile == NULL)
{
    perror("Error opening temporary file");
    return;
}

char line[MAX_LINE_LENGTH];
while (fgets(line, MAX_LINE_LENGTH, directoryFile) != NULL)
{
    if (strstr(line, name) == NULL)
    {
        fputs(line, tempFile);
    }
}
```

Finalmente se cierran ambos archivos, se elimina el directorio original y se sustituye por el nuevo. Además, se reabre en modo “r+” para posibles lecturas y escrituras.

```
fclose(directoryFile);
fclose(tempFile);
remove("directory.txt");
rename("temp.txt", "directory.txt");
directoryFile = fopen("directory.txt", "r+");
if (directoryFile == NULL)
{
    perror("Error opening file");
    return;
}
```

Operación List

La función listFiles se encarga de mostrar el contenido de un directorio en un sistema de archivos virtual. Primero, verifica si hay archivos en el directorio, y si no hay termina la ejecución. Si hay, recorre los que existen y va imprimiendo su nombre, tamaño, y el primer bloque asignado.

```
if (directory->head == NULL)
    {
        printf("No hay archivos en el directorio.\n");
        return;
    }
    printf("Directorio: \n");
    struct File *temp = directory->head;
    while (temp != NULL)
    {
        printf("Nombre: %s, Tamano: %d bytes, Primer bloque: %d\n", temp->name,
temp->size, temp->firstBlock);
        temp = temp->next;
    }
```

Referencias

A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, 10e*. John Wiley & Sons, 2018.