MASTER OF COMPUTER APPLICATION

DAYANANDA SAGAR ACADEMY

OF

TECHNOLOGY AND MANAGEMENT

PROJECT ON

"Hierarchical file system using a Tree data structure"

UNDER THE GUIDENCE AND SUPERVISION OF

Dr. D.R Enoch Arulprakash, MCA

Asst Prof of Computer Science Department

Submitted by

NAMES	USN
KHAZI PARVEEZ	1DT24MC042
LOKESH M	1DT24MC050
LATHESHA A N	1DT24MC047

INDEX

1) Abstract

This project implements a hierarchical file system using C programming, allowing users to perform fundamental file management operations. The system supports creating, deleting, and displaying directories and files, following a structured approach similar to real-world file systems. The implementation utilizes structures (structs) in C to define directories and files, ensuring an organized and efficient representation of the file system.

Key functionalities include the ability to create directories within a parent directory, add files to directories, delete directories and files, and display the directory hierarchy. The system employs dynamic memory allocation to handle variable-sized directory and file storage efficiently. Recursive functions are used for directory traversal, enabling seamless navigation through the hierarchical structure.

The directory and file management operations are executed through a menu-driven interface, allowing users to interact with the system dynamically. The project includes error handling mechanisms to ensure robustness, such as checking for the existence of parent directories before creating new directories or files and verifying the existence of files before deletion.

The directory visualization function provides a structured representation of the hierarchical organization, enhancing usability. Additionally, memory management techniques such as reallocation (realloc) and deallocation (free) help optimize resource usage, ensuring that storage is dynamically adjusted based on user operations.

This project serves as an educational tool for understanding file system design, memory management, recursion, and data structures in C programming. It provides a simplified yet practical demonstration of file system functionalities commonly found in operating systems, making it valuable for students and professionals interested in systems programming and data organization.

2) Introduction

A file system is an essential part of any operating system, organizing and managing stored data. This project aims to simulate basic file system operations such as creating, deleting, and displaying files and directories. Implementing these functionalities using data structures like trees and linked lists provides insight into how real-world file systems operate.

Significance of the Project

- File management is crucial for operating systems, databases, and cloud storage solutions.
- Understanding file organization principles helps in designing efficient storage systems.
- Learning dynamic memory allocation enhances coding efficiency and performance.

Real-World Applications

- Operating System file management (Linux/Windows file systems)
- Cloud storage solutions (Google Drive, OneDrive)
- Embedded systems requiring efficient storage management

3) Problem Definition

Problem Statement

The objective is to design a hierarchical file system that enables users to:

- Create and delete directories.
- Add and remove files from directories.
- Display the entire file system structure in a tree format.

Constraints and Assumptions

- The system operates in-memory, meaning data is not permanently stored.

- The maximum number of files or directories is limited by system memory.
- Each file and directory have a unique name within its parent directory.

Input and Expected Output

- **Input:** User commands to create/delete directories or files.
- **Expected Output:** The file structure is updated accordingly after each operation.

4) Algorithm Design: -

Approach

- **Use of Structures:**
    - struct Directories: Represents directories, stores subdirectories and files.
    - struct File: Represents individual files.
- **Dynamic Memory Allocation:**
    - malloc and realloc are used for efficient memory handling.
- **Recursive Traversal:**
    - Used for tree structure representation.
- **Menu-driven User Interaction:**
    - Enables the user to interact with the file system conveniently.

5) Pseudocode: -

```
Function createNode(name):
   Allocate memory for new directory
   Initialize name, files, subdirectories
   Return new directory

Function createFileNode(name):
   Allocate memory for new file
   Assign name
   Return file node

Function findDirectory(root, name):
   If root matches name, return root
   For each subdirectory in root:
      Recursively search
   Return NULL if not found

Function createDirectory(root, parentName, dirName):
   Find parent directory
   Allocate memory and append new directory

Function printTree(dir, level):
   Print directory name
   Print file names
   Recursively print subdirectories

Function createFile(root, dir, filename):
   Find directory
   Allocate memory and append file

Function deleteFile(dir, filename):
   Locate file
   Remove and reallocate memory

Function deleteDirectory(root, dirname):
   Locate directory
   Free memory recursively
   Adjust pointers
```

5) Implementation

Programming Language and Environment

- **Language:** C
- **Compiler:** GCC
- **IDE:** Code::Dev++ / Visual Studio Code
- **System:** Windows/Linux

Code Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100

// files structure
typedef struct File
{
   char fname[MAX];
} File;

// directories structure
typedef struct Directories
{
   char name[MAX];
   struct Directories *directories;
   File *files;
   int fsize;
   int dsize;
} Directories;

// Function to create a new directory node with 1 parameter {Dirname}
Directories *createNode(const char *name)
{
   Directories *newDir = (Directories *)malloc(sizeof(Directories));
   strncpy(newDir->name, name, MAX);
```

```
   newDir->directories = NULL;
   newDir->files = NULL;
   newDir->fsize = 0;
   newDir->dsize = 0;
   return newDir;
}

// function to create a new file node with 1 parameter{fileName}
File *createFileNode(const char *name)
{
   File newFile = (*File)malloc(sizeof(File));
   strncpy(newFile->fname, name, MAX);
   return newFile;
}

// Function to find a directory by name with 2 parameters {root dir
address}{fileName}
Directories *findDirectory(Directories *root, const char *name)
{
   int i;
   if (strcmp(root->name, name) == 0)
   {
      return root;
   }
   for (i = 0; i < root->dsize; ++i)
   {
      Directories *result = findDirectory(&root->directories[i], name);
      if (result != NULL)
      {
         return result;
      }
   }
   return NULL;
}

// Function to create a directory with 3 parameters {fileSystem Root file
address}{parentDir name}{newdir name}
void createDirectory(Directories *root, const char *parentName, const char
*dirName)
{
```

```
   Directories *parentDir = findDirectory(root, parentName);
   if (parentDir == NULL)
   {
      printf("Parent directory not found.\n");
      return;
   }
   parentDir->directories = (Directories *)realloc(parentDir->directories,
(parentDir->dsize + 1) * sizeof(Directories));
   parentDir->directories[parentDir->dsize++] = *createNode(dirName);
   printf("\n \t Directory created sucessfully\n");
}

// Function to print the directory tree with 2 parameters {root address}{level
for space}
void printTree(const Directories *dir, int level)
{
   int i, j;
   for (i = 0; i < level; ++i)
   {
      printf("  ");
   }
   printf("D: %s\n", dir->name);

   for (i = 0; i < dir->fsize; ++i)
   {
      for (j = 0; j < level + 1; ++j)
      {
         printf("   ");
      }
      printf("F: %s\n", dir->files[i].fname);
   }

   for (i = 0; i < dir->dsize; ++i)
   {
      printTree(&dir->directories[i], level + 1);
   }
}

// Function to creating file with 3 parameters {root
address}{dirName}{fileNmae}
```

```
void createFile(Directories *root, char *dir, char *f_name)
{
   Directories *parentDir = findDirectory(root, dir);
   if (parentDir == NULL)
   {
      printf("Parent directory not found .\n");
      return;
   }
   parentDir->files = (File *)realloc(parentDir->files, (parentDir->fsize + 1) *
sizeof(File));
   parentDir->files[parentDir->fsize++] = *createFileNode(f_name);
   printf("\n\n\t File created sucessfully \n");
}

// Function to delete file with 2 parameters {dirName}{fileName}
int deleteFile(Directories *dir, const char *fname)
{
   int index = -1, i, flag = 0;
   for (i = 0; i < dir->fsize; ++i)
   {
      if (strcmp(dir->files[i].fname, fname) == 0)
      {
         index = i;
         break;
      }
   }
   if (index != -1)
   {
      // Shift files to fill the gap
      for (i = index; i < dir->fsize - 1; ++i)
      {
         dir->files[i] = dir->files[i + 1];
      }
      dir->fsize--;
      dir->files = (File *)realloc(dir->files, dir->fsize * sizeof(File));
      flag = 1;
      return flag;
   }
   return flag;
}
```

```
// Function to delete directorie with 2 parameters {root address}{dirName}
int deleteDirectory(Directories *root, const char *dirName)
{
   Directories *parentDir = NULL;
   int index = -1, i, flag = 0;
   for (i = 0; i < root->dsize; ++i)
   {
      if (strcmp(root->directories[i].name, dirName) == 0)
      {
         parentDir = root;
         index = i;
         break;
      }
      else
      {
         deleteDirectory(&root->directories[i], dirName);
      }
   }
   if (parentDir != NULL && index != -1)
   {
      // Free memory allocated for the directory and its contents
      free(parentDir->directories[index].files);
      free(parentDir->directories[index].directories);

      // Shift directories to fill the gap
      for (i = index; i < parentDir->dsize - 1; ++i)
      {
         parentDir->directories[i] = parentDir->directories[i + 1];
      }
      parentDir->dsize--;
      parentDir->directories = (Directories *)realloc(parentDir->directories,
parentDir->dsize * sizeof(Directories));
      flag = 1;
      return flag;
   }
}

// main function
int main()
```

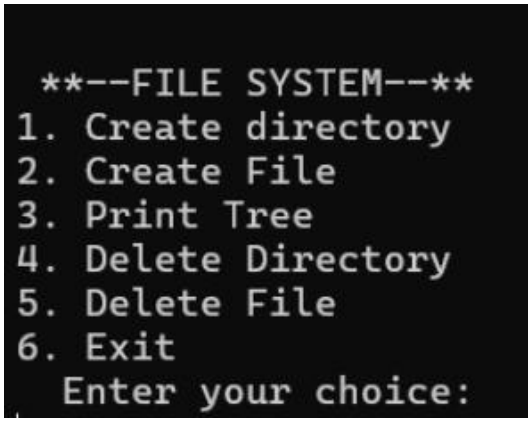
```
{
   char name[MAX], dirname[MAX];
   int ch;
   Directories *root = createNode("root");
   do
   {
      printf("\n *--FILE SYSTEM--*\n");
      printf("1. Create directory\n");
      printf("2. Create File\n");
      printf("3. Print Tree\n");
      printf("4. Delete Directory\n");
      printf("5. Delete File\n");
      printf("6. Exit \n");
      printf("  Enter your choice: \n");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         printf("Enter the directory name: \n");
         scanf("%s", name);
         printf("Enter the parent name: \n");
         scanf("%s", dirname);
         createDirectory(root, dirname, name);
         break;
      case 2:
         printf("enter the file name \n");
         scanf("%s", name);
         printf("enter the directori name \n");
         scanf("%s", dirname);
         createFile(root, dirname, name);
         break;
      case 3:
         printTree(root, 0);
         break;
      case 4:
         printf("\n enter the directory name \n");
         scanf("%s", dirname);
         if (deleteDirectory(root, dirname))
            printf("%s is deleted sucessfully deleted \n", dirname);
         else
```

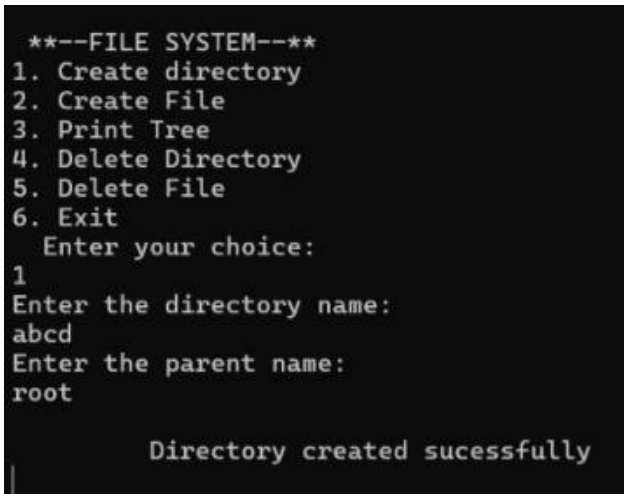
```
            printf("%s is not found \n", dirname);
         break;
      case 5:
         printf("\n enter the directory name \n");
         scanf("%s", dirname);
         Directories *dir = findDirectory(root, dirname);
         if (dir)
         {
            printf("\n enter the file name to be delete \n");
            scanf("%s", name);
            if (deleteFile(dir, name))
               printf("\n %s is deleted sucessfully \n ", name);
            else
               printf("\n No such file found with name %s \n ", name);
         }
         else
         {
            printf("\n no such directory with name %s \n", dirname);
         }
         break;
      case 6:
         printf("Exiting...\n");
         break;
      default:
         printf("Invalid choice\n");
         break;
      }
      getchar();
   } while (ch != 6);
   return 0;
}
```

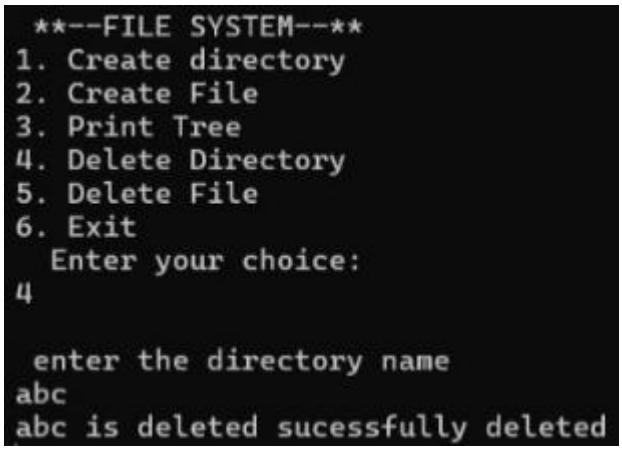
6) Test Cases and Results

Test Case No.	Input	Expected Output	Actual Output	Status
1	Create directory A in root	Directory created successfully	Directory created successfully	Pass
2	Create directory Inside A directory	Directory created successfully	Directory created successfully	Pass
3	Create file F1 in A	File created successfully	File created successfully	Pass
4	Delete file F1 from A	File deleted successfully	File deleted successfully	Pass
5	Delete directory A	Directory deleted successfully	Directory deleted successfully	Pass

based on problem definition each operation input and output screenshots are given below:-







```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
1
Enter the directory name:
efgh
Enter the parent name:
root

         Directory created sucessfully
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
1
Enter the directory name:
abc
Enter the parent name:
abcd

         Directory created sucessfully
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
2
enter the file name
wordslist.txt
enter the directori name
abc


         File created sucessfully
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
2
enter the file name
nameslist.txt
enter the directori name
abcd


         File created sucessfully
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
3
D: root
  D: abcd
      F: nameslist.txt
    D: abc
        F: wordslist.txt
  D: efgh
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
3
D: root
  D: abcd
      F: nameslist.txt
  D: efgh
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
5

 enter the directory name
abcd

 enter the file name to be delete
nameslist.txt

 nameslist.txt is deleted sucessfully
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
3
D: root
  D: abcd
  D: efgh
```

```
 **--FILE SYSTEM--**
1. Create directory
2. Create File
3. Print Tree
4. Delete Directory
5. Delete File
6. Exit
  Enter your choice:
6
Exiting...

--------------------------------
Process exited after 627.2 seconds with return value 0
Press any key to continue . . .
```

7) Observations and Challenges:

This project successfully implements a hierarchical file system using C programming**,** enabling essential file operations such as creating, deleting, and displaying directories and files. The system utilizes structs to represent directories and files, ensuring an organized and structured approach to data storage. Dynamic memory allocation is employed to manage directories and files efficiently, allowing for flexible storage adjustments. The recursive traversal method enhances the system's ability to navigate through nested directories seamlessly.

A menu-driven interface facilitates user interaction, making it easier to create and manage the file system dynamically. Additionally, error handling mechanisms ensure system stability by preventing invalid operations, such as deleting non-existent directories or files. The directory visualization function effectively represents the hierarchical structure, providing a clear and intuitive view of the system's organization.

Challenges Faced:

a. Recursive Deletion Complexity:
   - Deleting directories with multiple nested subdirectories required recursive traversal, making memory cleanup more complex.
   - Ensuring that all associated files and subdirectories were freed correctly was essential to avoid memory leaks.

b. Directory Search Efficiency:
   - Locating a specific directory within the hierarchy required deep traversal, impacting performance for large directory structures.
   - The current array-based approach has limitations in scalability and efficiency for large file systems.

c. Handling User Input and Preventing Errors:
   - Preventing segmentation faults due to invalid memory access required rigorous input validation.
   - The system had to ensure users did not attempt duplicate operations (e.g., creating files/directories with the same name).

8) Conclusion:

This project successfully implements a hierarchical file system in C programming, demonstrating essential file management operations such as creating, deleting, and displaying directories and files. By utilizing structs, dynamic memory allocation, and recursion, the project efficiently manages data organization and provides a structured representation of the file system.

The use of recursive directory traversal allows seamless navigation, while memory management techniques ensure efficient resource utilization. The menu-driven interface enhances user interaction, making the system intuitive and functional. Error handling mechanisms further improve reliability by preventing invalid operations.

This project provides a strong foundation for understanding file system architecture, data structures, and memory management techniques. It serves as a practical learning tool for students and professionals interested in systems programming and operating system development. Future improvements may include implementing file read/write operations, permission management, and persistent storage to further extend its capabilities.

Overall, this project highlights the core principles of file system management, offering insights into how modern operating systems handle hierarchical data structures efficiently.