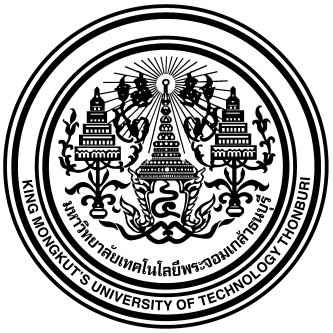

FILE SYSTEM SIMULATION

MR. THIRA RUNGRUANGKASED
MR. PAPANGKORN JITVOOTTIKRAI
MR. PHACHARAWAT EAKGAWATPHOKHIN

A PROJECT SUBMITTED IN PROGRAMMING WITH DATA STRUCTURES
CPE 112 OF THE REQUIREMENTS FOR
THE DEGREE OF BACHELOR OF ENGINEERING(COMPUTER ENGINEERING)
FACULTY OF ENGINEERING
KING MONGKUT'S UNIVERSITY OF TECHNOLOGY THONBURI 2024

Project Title	File System Simulation
Credits	3
Member(s)	Mr. Thira Rungruangkased Mr. Papangkorn Jitvoottikrai Mr. Phacharawat Eakgawatphokhin
Project Advisor	Prof. Natasha Dejdumrong, Proj. Naveed Sultan, Prof. Aye Hninn Khine
Program	Bachelor of Engineering
Field of Study	Computer Engineering
Department	Computer Engineering
Faculty	Engineering
Academic Year	2024

Abstract

This project is about simulating a file system using data structures in C language. A file system is a method of organizing, storing and accessing data on a computer. In this simulation, we use data structures such as linked lists and trees to manage files and folders. This program will allow users to create files, delete files, edit files, Navigating files, as well as directory management just like a computer system. The objective of this project is to increase understanding of how file systems operate internally and demonstrate how data structures can be applied to solve real-life problems. It helps to improve programming logic, problem solving skills, understanding of data structures, and understanding of computer management.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 Problem Statement and Approach

In real-world operating systems, file management is a critical feature that enables users to interact with data through the creation, manipulation, and deletion of files. However, implementing a file system from scratch is complex and deeply tied to OS internals.

File systems are a core function of every operating system. They enable users to store, retrieve, and organize data efficiently. Internally, file systems rely on data structures like trees for hierarchical organization and hash maps for fast access and lookup operations.

This project simulates a file system at a conceptual level using the C programming language. The approach involves building a command-line interface (CLI) that supports core file system commands — such as creating, renaming, deleting, and navigating directories. It uses trees to represent the directory structure and hash maps for optimizing search operations.

1.2 Objective

- Simulate a basic file system management and folder operations.
- Implement directory navigation and file searching using structured data types.
- Enhance performance with hash maps and caching strategies.
- Provide an intuitive command-line interface for user interaction.
- Strengthen understanding of file systems, tree traversal, and memory management
- Gain experiences with C programming and string handling

1.3 Expected outcome

- A fully functioning CLI-based file system simulation.
- Users can interact with a tree-structured directory containing files and folders.
- Operations such as create, delete, copy, move, rename, search, and list are supported.
- The system will optimize performance with hashing and caching techniques.
- The final system will demonstrate modular, maintainable code using C.

CHAPTER 2 SOLUTION WITH FUNCTIONALITIES

2.1 File/Folder Management

- create: Create a new file or folder in the current directory.
- rename: Change the name of a file or folder.
- delete: Remove a file or folder permanently.
- move: Move an item to another folder.
- copy: Duplicate a file or folder.
- write: Modify contents of a file.
- view: Show contents of a file.

2.2 Directory Navigation

- cd: Change the current directory.
- ls: List contents of the current directory.
- pwd: Show the full path to the current directory.
- home: Return to the root directory.

2.3 Searching

- search: Locate files/folders by name.
- filter: Show only files or only folders.
- goto: Jump to the path of a selected file or folder.

2.4 Optimizing

- Hashing: Use a hash map for quick file/folder lookup inside directories.

2.5 User Interface

- Command-Line Interface (CLI) with responsive feedback.
- Help tool that explains all available commands.

CHAPTER 3 CODE WALKTHROUGH

3.1 Main

Handles program startup, input loop, and termination. It initializes the file system, loads saved data, and continually reads user commands until user exit.

```c
#include <stdio.h>
#include <string.h>
#include "module/create.h"
#include "module/interface.h"
#include "module/command.h"
#include "module/struct.h"
#include "module/loader.h"
#include "module/saver.h"

int main(void){
    node head;
    struct_init(&head);
    loader_main(&head);
    node* current = &head;
    char path[50];

    printf("File System [Version 0.0.0.1]\n");
    while(1){
        struct_buildPath(&head, current, path);
        int r = interface_input(path, &current);
        if(r == -1) break;
        else if(r == 3) current = &head;
        else if(r == 4) create_folder(current);
        else if(r == 5) create_file(current);
    }
    saver_main(&head);
    return 0;
}
```

3.2 Interface

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "command.h"
#include "struct.h"
#include "rename.h"
#include "delete.h"
#include "copy.h"
#include "move.h"
#include "file.h"

void interface_cleancomm(char* comm){
    for(int i = 0; i < strlen(comm); i++){
        comm[i] = tolower(comm[i]);
    }
}

void interface_list(node* current){
    if(current->child == NULL) return;

    node* temp = current->child;
    while (temp != NULL) {
        printf("%s\n", temp->value);
        temp = temp->next;
    }
}
```

```c
int interface_input(char* path, node** current){
    char comm[10];
    printf("%s> ", path);
    scanf("%s", comm);
    interface_cleancomm(comm);
    int r = command_read(comm);

    if(r == 1) printf("Path\n----\n%s\n", path);
    else if(r == 2){
        printf("mkdir\tCreate folder.\n");
        printf("touch\tCreate file.\n");
        printf("mv\tRename directory.\n");
        printf("rm\t Remove directory.\n");
        printf("mv\tMove directory.\n");
        printf("cp\tDuplicate directory.\n");
        printf("nano\tEdit file.\n");
        printf("cat\tView file content.\n");
        printf("cd\tChange the current directory.\n");
        printf("ls\tDisplay the current directory.\n");
        printf("pwd\tDisplay the current directory.\n");
        printf("home\tQuickly return to root directory.\n");
        printf("exit\tTerminate program.\n");
    }else if(r == 6) interface_list(*current);
    else if(r == 7) command_navigate(current);
    else if(r == 8) rename_main(*current);
    else if(r == 9) delete_main(*current);
    else if(r == 10) copy_main(*current);
    else if(r == 11) move_main(*current);
    else if(r == 12) file_view(*current);
    else if(r == 13) file_edit(*current);
    return r;
}
```

Provides a command-line interface (CLI) where users type commands to interact with the file system. It shows prompts, reads input, and provides feedback.

3.3 Struct

Defines the main data structures such as FileNode, representing files and folders, and includes fields like name, type, parent, children, and content.

```c
#include "struct.h"
#include <string.h>

void struct_init(node* head){
    strcpy(head->value, "Home");
    head->isFolder = 1;
    head->child = NULL;
    head->next = NULL;
    head->content = NULL;
    head->parent = NULL;
}

int findPath(node* current, node* target, char* path) {
    if (current == NULL) return 0;
    strcat(path, current->value);

    if (current == target) return 1;

    if (current->child) {
        strcat(path, "/");
        if (findPath(current->child, target, path)) return 1;
        path[strlen(path) - 1] = '\0';
    }
    if (current->next) {
        int len = strlen(current->value);
        path[strlen(path) - len] = '\0';
        if (findPath(current->next, target, path)) return 1;
    }
    return 0;
}

void struct_buildPath(node* root, node* target, char* path) {
    path[0] = '\0';
    if (!findPath(root, target, path)) strcpy(path, "Node not found");
}
```

3.4 File

```c
#include <stdio.h>
#include <string.h>
#include "struct.h"

void file_view(node* current){
    char name[30];
    scanf("%s", name);

    node* temp = current->child;
    while (temp != NULL) {
        if (!temp->isFolder && strcmp(temp->value, name) == 0) {
            printf("%s\n",temp->content);
            return;
        }
        temp = temp->next;
    }

    printf("%s not found.\n", name);
}
```

```c
void file_edit(node* current) {
    char name[30];
    scanf("%s", name);
    getchar();

    node* temp = current->child;
    while (temp != NULL) {
        if (!temp->isFolder && strcmp(temp->value, name) == 0) {
            printf("Enter new content for %s (end with a single line containing only `::end`)\n", name);

            char buffer[1000] = "";
            char line[256];
            while (1) {
                fgets(line, sizeof(line), stdin);
                if (strcmp(line, "::end\n") == 0) break;
                strcat(buffer, line);
            }

            strcpy(temp->content, buffer);
            printf("%s updated.\n", name);
            return;
        }
        temp = temp->next;
    }

    printf("%s not found.\n", name);
}
```

Handles file-related operations including reading, writing, and displaying file content. It manages data stored within FileNode structures for file-type nodes.

3.5 Command

```c
#include <stdio.h>
#include <string.h>
#include "struct.h"

void command_navigate(node** current) {
    char name[30];
    scanf("%s", name);

    if (strcmp(name, "..") == 0) {
        if ((*current)->parent != NULL) {
            *current = (*current)->parent;
        }
        return;
    }

    node* temp = (*current)->child;
    while (temp != NULL) {
        if (temp->isFolder && strcmp(temp->value, name) == 0) {
            *current = temp;
            return;
        }
        temp = temp->next;
    }
}
```

```c
int command_read(char* comm){
    if(strcmp(comm, "exit") == 0){
        return -1;
    }else if(strcmp(comm, "pwd") == 0){
        return 1;
    }else if(strcmp(comm, "help") == 0){
        return 2;
    }else if(strcmp(comm, "home") == 0){
        return 3;
    }else if(strcmp(comm, "mkdir") == 0){
        return 4;
    }else if(strcmp(comm, "touch") == 0){
        return 5;
    }else if(strcmp(comm, "ls") == 0){
        return 6;
    }else if(strcmp(comm, "cd") == 0){
        return 7;
    }else if(strcmp(comm, "rn") == 0){
        return 8;
    }else if(strcmp(comm, "rm") == 0){
        return 9;
    }else if(strcmp(comm, "cp") == 0){
        return 10;
    }else if(strcmp(comm, "mv") == 0){
        return 11;
    }else if(strcmp(comm, "cat") == 0){
        return 12;
    }else if(strcmp(comm, "nano") == 0){
        return 13;
    }
    return 0;
}
```

Analyze user input and split it into commands, also navigate the request to the appropriate function. It acts as the navigator for all CLI commands.

3.6 Create

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "struct.h"

void create_folder(node* current) {
    getchar();
    char name[100];
    fgets(name, sizeof(name), stdin);
    name[strcspn(name, "\n")] = 0;

    node* newNode = (node*)malloc(sizeof(node));
    strcpy(newNode->value, name);
    newNode->isFolder = 1;
    newNode->child = NULL;
    newNode->next = NULL;
    newNode->parent = current;

    if (current->child == NULL) {
        current->child = newNode;
    } else {
        node* temp = current->child;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }

    printf("Folder %s created.\n", name);
}
```

```c
void create_file(node* current) {
    getchar();
    char name[100];
    fgets(name, sizeof(name), stdin);
    name[strcspn(name, "\n")] = 0;

    node* newNode = (node*)malloc(sizeof(node));
    strcpy(newNode->value, name);
    newNode->isFolder = 0;
    newNode->child = NULL;
    newNode->next = NULL;
    newNode->parent = current;

    if (current->child == NULL) {
        current->child = newNode;
    } else {
        node* temp = current->child;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }

    printf("File %s created.\n", name);
}
```

Implements the create command to add new files or folders to the current directory. It allocates memory and updates the tree structure.

3.7 Delete

Implements the delete command to remove files or folders. If a folder is deleted, its contents will be removed from memory and the tree.

```c
#include "struct.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void delete_node(node* target) {
    if (target == NULL) return;
    delete_node(target->child);
    delete_node(target->next);
    free(target);
}

void delete_main(node* current) {
    char dest[30];
    scanf("%s", dest);

    if (current == NULL) return;

    node* prev = NULL;
    node* temp = current->child;

    while (temp != NULL) {
        if (strcmp(temp->value, dest) == 0) {
            if (prev == NULL) current->child = temp->next;
            else prev->next = temp->next;

            delete_node(temp);
            printf("Deleted %s\n", dest, current->value);
            return;
        }
        prev = temp;
        temp = temp->next;
    }
}
```

3.8 Copy

```c
#include "struct.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

node* copy_node(node* original) {
    if (original == NULL) return NULL;

    node* copy = malloc(sizeof(node));
    if (!copy) return NULL;

    strcpy(copy->value, original->value);
    copy->isFolder = original->isFolder;
    copy->parent = NULL;
    copy->child = copy_node(original->child);

    node* temp = copy->child;
    while (temp != NULL) {
        temp->parent = copy;
        temp = temp->next;
    }

    copy->next = NULL;
    return copy;
}
```

```c
void copy_main(node* current) {
    char src[30], dest[30];
    scanf("%s %s", src, dest);

    if (current == NULL) return;

    node* temp = current->child;

    while (temp != NULL) {
        if (strcmp(temp->value, src) == 0) {
            node* new_node = copy_node(temp);
            strcpy(new_node->value, dest);
            new_node->next = temp->next;
            temp->next = new_node;
            new_node->parent = current;
            printf("Copied '%s' as '%s'\n", src, dest);
            return;
        }
        temp = temp->next;
    }

    printf("Item '%s' not found.\n", src);
}
```

Handles the copy command. It duplicates files or entire directories, preserving structure and contents then adds them to the target location.

3.9 Move

```c
#include <string.h>
#include "struct.h"
#include <stdio.h>

node* find_node_by_path(node** start, const char* path) {
    node* curr = *start;
    char temp[100];
    strcpy(temp, path);

    char* token = strtok(temp, "/");
    while (token != NULL && curr) {
        node* child = curr->child;
        while (child) {
            if (strcmp(child->value, token) == 0) {
                curr = child;
                break;
            }
            child = child->next;
        }
        if (!child) return NULL;
        token = strtok(NULL, "/");
    }
    return curr;
}
```

```c
void move_main(node* current) {
    char src_name[50], dest_path[100];
    scanf("%s %s", src_name, dest_path);

    if (!current) return;

    node* prev = NULL;
    node* src_node = current->child;

    while (src_node && strcmp(src_node->value, src_name) != 0) {
        prev = src_node;
        src_node = src_node->next;
    }

    if (!src_node) {
        printf("Item not found\n", src_name, current->value);
        return;
    }

    if (prev == NULL) current->child = src_node->next;
    else prev->next = src_node->next;

    node* dest_parent = current;
    node* dest_node = find_node_by_path(&dest_parent, dest_path);

    if (!dest_node || !dest_node->isFolder) {
        printf("Destination not found.\n", dest_path);
        src_node->next = current->child;
        current->child = src_node;
        return;
    }

    src_node->next = dest_node->child;
    dest_node->child = src_node;
    src_node->parent = dest_node;

    printf("Moved %s to %s\n", src_name, dest_node->value);
}
```

Implements the move command, which relocates a file
or folder to a different directory by updating parent-child relationships in the tree.

3.10 Loader

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "struct.h"

void create_node_from_path(node* root, const char* type, const char* path) {
    char pathCopy[256];
    strcpy(pathCopy, path);
    char* token = strtok(pathCopy, "/");

    node* current = root;
    while (token != NULL) {
        node* temp = current->child;
        int found = 0;

        while (temp) {
            if (strcmp(temp->value, token) == 0) {
                current = temp;
                found = 1;
                break;
            }
            temp = temp->next;
        }

        if (!found) {
            node* newNode = malloc(sizeof(node));
            strcpy(newNode->value, token);
            newNode->isFolder = 1;
            newNode->content = NULL;
            newNode->child = NULL;
            newNode->next = NULL;
            newNode->parent = current;

            if (!current->child) {
                current->child = newNode;
            } else {
                node* last = current->child;
                while (last->next) last = last->next;
                last->next = newNode;
            }

            current = newNode;
        }

        token = strtok(NULL, "/");
    }
    current->isFolder = (strcmp(type, "FOLDER") == 0) ? 1 : 0;
}
```

```c
void load_filesystem(node* root) {
    FILE* fp = fopen("./data/system.txt", "r");
    if (!fp){
        return;
    }

    char type[10], path[256];
    while (fscanf(fp, "%s %[^\n]", type, path) == 2) {
        create_node_from_path(root, type, path);
    }

    fclose(fp);
}

node* loader_find_node_by_path(node* root, const char* path) {
    char pathCopy[256];
    strcpy(pathCopy, path);
    char* token = strtok(pathCopy, "/");

    node* current = root;
    while (token && current) {
        node* temp = current->child;
        current = NULL;

        while (temp) {
            if (strcmp(temp->value, token) == 0) {
                current = temp;
                break;
            }
            temp = temp->next;
        }

        token = strtok(NULL, "/");
    }

    return current;
}
```

Reads previously saved file system data from disk on program start. It reconstructs the tree structure so users can resume where they left off.

```c
void load_file_content(node* root) {
    FILE* fp = fopen("./data/content.txt", "r");
    if (!fp) return;

    char line[512], path[256], content[10000] = "";
    node* target = NULL;
    int reading = 0;

    while (fgets(line, sizeof(line), fp)) {
        if (strncmp(line, ":::", 3) == 0) {
            if (target && !target->isFolder) {
                target->content = strdup(content);
            }

            strcpy(path, line + 3);
            path[strcspn(path, "\n")] = 0;
            target = loader_find_node_by_path(root, path);
            content[0] = '\0';
            reading = 1;
        } else if (strncmp(line, "---END---", 9) == 0) {
            if (target && !target->isFolder) {
                target->content = strdup(content);
            }
            reading = 0;
            content[0] = '\0';
            target = NULL;
        } else if (reading && target && !target->isFolder) {
            strcat(content, line);
        }
    }

    fclose(fp);
}

void loader_main(node* root) {
    load_filesystem(root);
    load_file_content(root);
}
```

3.11 Rename

Implements the rename command. Users can change the name of a file or folder. It updates the node's name while preserving its content and position.

FileSystem-main > module > rename.c > ...

```c
#include <string.h>
#include <stdio.h>
#include "struct.h"

void rename_main(node* file) {
    char dest[30], name[30];
    scanf("%s %s", dest, name);

    node* temp = file->child;
    while (temp != NULL) {
        if (strcmp(temp->value, dest) == 0) {
            strcpy(temp->value, name);
            printf("Renamed %s to %s\n", dest, name);
            return;
        }
        temp = temp->next;
    }
}
```

3.12 Saver

Saves the current file system state to disk before the program exits. It serializes the tree structure so it can be loaded again later.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "struct.h"

void save_file_content(node* current, FILE* content_file) {
    if (current->isFolder) return;
    fprintf(content_file, ":::%s\n", current->value);
    if (current->content != NULL) fprintf(content_file, "%s\n", current->content);
    fprintf(content_file, "---END---\n\n");
}

void save_filesystem_structure(node* current, FILE* structure_file) {
    if (current == NULL) return;
    if (current->isFolder) fprintf(structure_file, "FOLDER %s\n", current->value);
    else fprintf(structure_file, "FILE %s\n", current->value);
    if (current->child != NULL) save_filesystem_structure(current->child, structure_file);
    if (current->next != NULL) save_filesystem_structure(current->next, structure_file);
}

void saver_main(node* root) {
    FILE* structure_file = fopen("./data/system.txt", "w");
    FILE* content_file = fopen("./data/content.txt", "w");

    if (structure_file == NULL || content_file == NULL) return;

    save_filesystem_structure(root, structure_file);

    node* temp = root->child;
    while (temp != NULL) {
        save_file_content(temp, content_file);
        temp = temp->next;
    }

    fclose(structure_file);
    fclose(content_file);
}
```

CHAPTER 4 TIME COMPLEXITY ANALYSIS

Creating item - Average of **O(n)** to put item in the tail of linked list.
Renaming item - Require **O(n)** times to rename specific items.
Deleting item - Normally takes **O(n)** to find that specific item.
Moving item - Require **O(d*k)** where d is the depth of the tree, k is the number of childrens.
Copying item - Takes **O(c+n)** where c is number of children, n is total nodes in subtree.
Writing files - Uses average on **O(n)** to find nodes to rewrite.
Reading files - Uses average of **O(n)** to traverse the linked list.
Navigation through Directories - Require **O(n)** to find specific directories.
Search for Files or Folders - With a hash table, time complexity reduces to **O(1)**.

CHAPTER 5 TEAM MEMBER RESPONSIBILITIES

Thira Rungruangkased - File and folder operations, hashing functions
Papangkorn Jitvoottikrai - Command parsing, CLI interaction, navigation logic
Phacharawat Eakgawatphokhin - Search system, quality testing, project management