Capstone Project

Assignment - 1

File Explorer Application

Submitted By

Name : **Jasvant Panigrahi**

Batch : **03**

Department : **BTECH-CSE(Data Science)**

Registration Number : **2241016162**

Section : **45**

Assignment 1 (Linux OS)

File Explorer Application

ABSTRACT

This project, File Explorer Application, is a console-based file management system developed in C++ using the Filesystem library introduced in C++17. It allows users to perform operations such as listing files, navigating directories, creating and removing files or folders, copying and moving files, searching, and managing file permissions - all within a Linux command-line environment.

OBJECTIVE

The objective of this project is to design and implement a console-based file explorer that performs core Linux file management operations such as listing, navigation, copying, moving, deleting, searching, and modifying file permissions. This helps in understanding how the Linux filesystem works through C++ programming concepts.

Theoretical Background

Linux File System

- The Linux file system uses a hierarchical directory structure.
- It supports permissions (read, write, execute) for user, group, and others.
- Directories and files are treated as objects under one unified structure.

C++ Filesystem Library

- Provides tools to manage files, directories, and paths.
- Functions like directory_iterator(), exists(), copy_file(), remove_all() simplify file management tasks.

Command Line Interface

- A REPL (Read-Eval-Print Loop) is implemented allowing users to execute Linux-like commands directly.

File Permissions

- File permissions in Linux are expressed in octal (e.g., 755, 644).
- Permissions are divided into owner, group, and others, and determine who can read (r), write (w), or execute (x) a file.

CODE (C++ Implementation)

```
 #include <filesystem>
#include <iostream>
#include <string>
#include <vector>
#include <chrono>
#include <iomanip>
#include <sstream>
#include <system_error>
#include <fstream>

namespace fs = std::filesystem;

class FileExplorer {
public:
    FileExplorer() {
        try {
            cwd = fs::current_path();
        } catch (std::exception &e) {
            cwd = fs::path("/");
        }
    }

    void repl() {
        std::cout << "Simple File Explorer — type 'help' for commands\n";
        while (true) {
            std::cout << cwd << " $ ";
            std::string line;
            if (!std::getline(std::cin, line)) break;
            if (line.empty()) continue;

            auto args = split_args(line);
            auto cmd = args[0];

            if (cmd == "exit" || cmd == "quit") break;
            if (cmd == "help") {
                print_help();
                continue;
            }

            try {
                if (cmd == "ls") cmd_ls(args);
                else if (cmd == "cd") cmd_cd(args);
                else if (cmd == "pwd") cmd_pwd(args);
                else if (cmd == "stat") cmd_stat(args);
                else if (cmd == "cp") cmd_copy(args);
                else if (cmd == "mv") cmd_move(args);
                else if (cmd == "rm") cmd_remove(args);
                else if (cmd == "mkdir") cmd_mkdir(args);
                else if (cmd == "touch") cmd_touch(args);
                else if (cmd == "search") cmd_search(args);
                else if (cmd == "chmod") cmd_chmod(args);
                else std::cout << "Unknown command: " << cmd << " (type 'help')\n";
            } catch (const std::exception &e) {
                std::cout << "Error: " << e.what() << "\n";
            } catch (...) {
                std::cout << "Unknown error occurred.\n";
            }
        }
    }

private:
    fs::path cwd;

    static std::vector<std::string> split_args(const std::string &line) {
        std::istringstream iss(line);
        std::vector<std::string> args;
        std::string token;
```

```
    while (iss >> token) args.push_back(token);
    return args;
}

void print_help() {
    std::cout << "Commands:\n"
              << " ls [-l] [path] : list directory (use -l for details)\n"
              << " cd <path> : change directory\n"
              << " pwd : print current directory\n"
              << " stat <path> : show file/directory info\n"
              << " cp <src> <dst> : copy file/directory\n"
              << " mv <src> <dst> : move/rename file or directory\n"
              << " rm <path> : remove file or directory (recursive for dirs)\n"
              << " mkdir <dir> : create directory\n"
              << " touch <file> : create empty file (or update timestamp)\n"
              << " search <name> [path] : recursive search for name (partial match)\n"
              << " chmod <octal> <path> : change permissions (e.g. chmod 755 file)\n"
              << " help : show this help\n"
              << " exit : quit\n";
}

fs::path resolve(const std::string &p) {
    fs::path path(p);
    if (path.is_relative()) path = cwd / path;
    return fs::weakly_canonical(path);
}

void cmd_ls(const std::vector<std::string> &args) {
    bool longfmt = false;
    fs::path target = cwd;
    size_t i = 1;

    if (i < args.size() && args[i] == "-l") {
        longfmt = true;
        ++i;
    }
    if (i < args.size()) target = resolve(args[i]);
    if (!fs::exists(target)) {
        std::cout << "ls: path does not exist: " << target << "\n";
        return;
    }

 if (fs::is_regular_file(target)) {
        if (longfmt) print_detailed(target);
        else std::cout << target.filename().string() << "\n";
        return;
    }

    std::error_code ec;
    for (auto &entry : fs::directory_iterator(target, ec)) {
        if (ec) {
            std::cout << "ls: error reading directory\n";
            break;
        }
        if (longfmt) print_detailed(entry.path());
        else std::cout << entry.path().filename().string() << "\n";
    }
}
```

```
void print_detailed(const fs::path &p) {
        std::error_code ec;
        char typechar = '?';
        if (fs::is_directory(p, ec)) typechar = 'd';
        else if (fs::is_regular_file(p, ec)) typechar = '-';
        else if (fs::is_symlink(p, ec)) typechar = 'l';

        auto size = fs::is_regular_file(p, ec) ? fs::file_size(p, ec) : 0ULL;
        auto perms = fs::status(p, ec).permissions();

        std::cout << typechar
                  << perms_to_string(perms)
                  << ' ' << std::setw(10) << size
                  << ' ' << time_to_string(fs::last_write_time(p, ec))
                  << ' ' << p.filename().string() << "\n";
    }

    static std::string perms_to_string(fs::perms p) {
        std::string s = "---------";
        if ((p & fs::perms::owner_read) != fs::perms::none) s[0] = 'r';
        if ((p & fs::perms::owner_write) != fs::perms::none) s[1] = 'w';
        if ((p & fs::perms::owner_exec) != fs::perms::none) s[2] = 'x';
        if ((p & fs::perms::group_read) != fs::perms::none) s[3] = 'r';
        if ((p & fs::perms::group_write) != fs::perms::none) s[4] = 'w';
        if ((p & fs::perms::group_exec) != fs::perms::none) s[5] = 'x';
        if ((p & fs::perms::others_read) != fs::perms::none) s[6] = 'r';
        if ((p & fs::perms::others_write) != fs::perms::none) s[7] = 'w';
        if ((p & fs::perms::others_exec) != fs::perms::none) s[8] = 'x';
        return s;
    }

    static std::string time_to_string(const fs::file_time_type &ft) {
        using namespace std::chrono;
        auto sctp = time_point_cast<system_clock::duration>(
            ft - fs::file_time_type::clock::now() + system_clock::now());
        std::time_t cftime = system_clock::to_time_t(sctp);
        std::tm tm = *std::localtime(&cftime);
        std::ostringstream oss;
        oss << std::put_time(&tm, "%Y-%m-%d %H:%M:%S");
        return oss.str();
    }

void cmd_cd(const std::vector<std::string> &args) {
        if (args.size() < 2) {
            std::cout << "cd: requires argument\n";
            return;
        }
        fs::path dest = resolve(args[1]);
        if (!fs::exists(dest) || !fs::is_directory(dest)) {
            std::cout << "cd: not a directory: " << dest << "\n";
            return;
        }
        cwd = fs::weakly_canonical(dest);
    }

    void cmd_pwd(const std::vector<std::string> &) {
        std::cout << cwd << "\n";
    }

    void cmd_stat(const std::vector<std::string> &args) {
        if (args.size() < 2) {
            std::cout << "stat: requires path\n";
            return;
        }

fs::path p = resolve(args[1]);
        if (!fs::exists(p)) {
            std::cout << "stat: path does not exist\n";
            return;
        }
```

```
print_detailed(p);
        std::cout << "Absolute: " << fs::absolute(p).string() << "\n";
        std::cout << "Type: " << (fs::is_directory(p) ? "directory" : (fs::is_regular_file(p) ? "file" : "other")) << "\n";
    }

    void cmd_copy(const std::vector<std::string> &args) {
        if (args.size() < 3) {
            std::cout << "cp: requires source and destination\n";
            return;
        }

        fs::path src = resolve(args[1]);
        fs::path dst = resolve(args[2]);
        if (!fs::exists(src)) {
            std::cout << "cp: source does not exist\n";
            return;
        }

        std::error_code ec;
        if (fs::is_directory(src)) {
            fs::create_directories(dst, ec);
            for (auto &entry : fs::recursive_directory_iterator(src, ec)) {
                if (ec) break;
                auto rel = fs::relative(entry.path(), src, ec);
                fs::path target = dst / rel;
                if (fs::is_directory(entry.path()))
                    fs::create_directories(target, ec);
                else
                    fs::copy_file(entry.path(), target, fs::copy_options::overwrite_existing, ec);
            }
        } else {
            if (fs::is_directory(dst)) dst /= src.filename();
            fs::copy_file(src, dst, fs::copy_options::overwrite_existing, ec);
        }

        if (ec) std::cout << "cp: error: " << ec.message() << "\n";
    }

    void cmd_move(const std::vector<std::string> &args) {
        if (args.size() < 3) {
            std::cout << "mv: requires source and destination\n";
            return;
        }

        fs::path src = resolve(args[1]);
        fs::path dst = resolve(args[2]);
        if (!fs::exists(src)) {
            std::cout << "mv: source does not exist\n";
            return;
        }

std::error_code ec;
        if (fs::is_directory(dst)) dst /= src.filename();
        fs::rename(src, dst, ec);
        if (ec) {
            cmd_copy(args);
            std::vector<std::string> rmargs = {"rm", args[1]};
            cmd_remove(rmargs);
        }
    }

    void cmd_remove(const std::vector<std::string> &args) {
        if (args.size() < 2) {
            std::cout << "rm: requires path\n";
            return;
        }

        fs::path p = resolve(args[1]);
        if (!fs::exists(p)) {
            std::cout << "rm: path does not exist\n";
            return;
        }
```

```
std::error_code ec;
        if (fs::is_directory(p))
            fs::remove_all(p, ec);
        else
            fs::remove(p, ec);

        if (ec) std::cout << "rm: error: " << ec.message() << "\n";
    }

    void cmd_mkdir(const std::vector<std::string> &args) {
        if (args.size() < 2) {
            std::cout << "mkdir: requires directory name\n";
            return;
        }
        fs::path dir = resolve(args[1]);
        std::error_code ec;
        fs::create_directories(dir, ec);
        if (ec) std::cout << "mkdir: error: " << ec.message() << "\n";
    }

    void cmd_touch(const std::vector<std::string> &args) {
        if (args.size() < 2) {
            std::cout << "touch: requires filename\n";
            return;
        }

        fs::path file = resolve(args[1]);
        std::error_code ec;
        if (!fs::exists(file)) {
            std::ofstream ofs(file.string());
            if (!ofs) std::cout << "touch: could not create file\n";
            ofs.close();
        } else {
            auto now = fs::file_time_type::clock::now();
            fs::last_write_time(file, now, ec);
            if (ec) std::cout << "touch: failed to update timestamp: " << ec.message() << "\n";
        }
    }

    void cmd_search(const std::vector<std::string> &args) {
        if (args.size() < 2) {
            std::cout << "search: requires name\n";
            return;
        }

std::string name = args[1];
        fs::path start = cwd;
        if (args.size() >= 3) start = resolve(args[2]);
        if (!fs::exists(start) || !fs::is_directory(start)) {
            std::cout << "search: start path not a directory\n";
            return;
        }

        std::error_code ec;
        for (auto &entry : fs::recursive_directory_iterator(start, ec)) {
            if (ec) {
                std::cout << "search: error reading directory\n";
                break;
            }
            std::string fname = entry.path().filename().string();
            if (fname.find(name) != std::string::npos)
                std::cout << entry.path().string() << "\n";
        }
    }

void cmd_chmod(const std::vector<std::string> &args) {
        if (args.size() < 3) {
            std::cout << "chmod: requires mode and path\n";
            return;
        }
```

```
std::string mode_str = args[1];
        fs::path target = resolve(args[2]);
        if (!fs::exists(target)) {
            std::cout << "chmod: path does not exist\n";
            return;
        }

        int mode = 0;
        try {
            mode = std::stoi(mode_str, nullptr, 8);
        } catch (...) {
            std::cout << "chmod: invalid mode\n";
            return;
        }

        fs::perms p = octal_to_perms(mode);
        std::error_code ec;
        fs::permissions(target, p, ec);
        if (ec) std::cout << "chmod: error: " << ec.message() << "\n";
    }

    static fs::perms octal_to_perms(int oct) {
        fs::perms p = fs::perms::none;
        int u = (oct >> 6) & 7;
        int g = (oct >> 3) & 7;
        int o = oct & 7;

        if (u & 4) p |= fs::perms::owner_read;
        if (u & 2) p |= fs::perms::owner_write;
        if (u & 1) p |= fs::perms::owner_exec;
        if (g & 4) p |= fs::perms::group_read;
        if (g & 2) p |= fs::perms::group_write;
        if (g & 1) p |= fs::perms::group_exec;
        if (o & 4) p |= fs::perms::others_read;
        if (o & 2) p |= fs::perms::others_write;
        if (o & 1) p |= fs::perms::others_exec;

        return p;
    }
};

int main() {
    FileExplorer fe;
    fe.repl();
    std::cout << "Bye!\n";
    return 0;
}
```

OUTPUT SCREENSHOTS

C++ Console Execution Output :-

```
"/home/compiler/Project_Files" $ mv copy_demo.txt final_demo.txt
"/home/compiler/Project_Files" $ ls
demo.txt
final_demo.txt

"/home/compiler/Project_Files" $ chmod 755 final_demo.txt
"/home/compiler/Project_Files" $ stat final_demo.txt
-rwxr-xr-x        0 2025-11-08 12:36:10 final_demo.txt
Absolute: /home/compiler/Project_Files/final_demo.txt
Type: file

"/home/compiler/Project_Files" $ search demo
/home/compiler/Project_Files/demo.txt
/home/compiler/Project_Files/final_demo.txt

"/home/compiler/Project_Files" $ rm demo.txt
"/home/compiler/Project_Files" $ ls
final_demo.txt

"/home/compiler/Project_Files" $ cd ..
"/home/compiler" $ ls
Desktop
Documents
Downloads
Project_Files
main.cpp

"/home/compiler" $ exit
Bye!
```

Equivalent Bash Output Representation :-

```
pwd
/home/compiler

ls
Desktop  Documents  Downloads  main.cpp  test.txt

mkdir Project_Files
cd Project_Files

touch demo.txt
ls -l
-rw-r--r-- 1 user user 0 Nov 08 12:34 demo.txt

cp demo.txt copy_demo.txt
mv copy_demo.txt final_demo.txt
ls
demo.txt  final_demo.txt

chmod 755 final_demo.txt
stat final_demo.txt
File: final_demo.txt
Size: 0       Blocks: 0       IO Block: 4096   regular file
Access: (0755/-rwxr-xr-x)  Uid: (1000/user)   Gid: (1000/user)
Modify: 2025-11-08 12:36:10

find . -name "*demo*"
./demo.txt
./final_demo.txt

rm demo.txt
ls
final_demo.txt
```

EACH STEPS (DAYS) EXPLANATION

Day 1 - Application Setup and Basic Listing

The first step involved designing the application's structure and preparing the development environment. Essential design components were carefully structured before implementation.
A FileExplorer class was created in C++ to handle all operations in a modular way.
The current working directory (cwd) was set using std::filesystem::current_path().
The ls command was implemented to list files within directories using directory_iterator, and pwd was added to display the current directory path. The implementation worked without errors
Testing confirmed successful retrieval of files and directories, proving the environment setup and listing functionality were correctly implemented.

Day 2 - Directory Navigation Implementation

On the second day, directory traversal commands were added to allow seamless movement through folders. Directory handling functions were efficiently integrated for performance.
The cd command was developed using fs::weakly_canonical() to handle both relative and absolute paths safely, while maintaining error checking for invalid inputs. Then coming to next part.
The pwd command printed the current directory whenever needed.
With this update, users could explore directories dynamically - moving forward and backward just like in a Linux shell. User experience was tested thoroughly and validated successfully.
This feature established the application's interactivity and practical navigation control.

Day 3 - File and Directory Manipulation

This day focused on enabling essential file management operations within the File Explorer.
The following commands were introduced:

cp -> for copying files and directories (supports recursive copy).
mv -> for moving or renaming files.
rm -> for deleting files or directories.
mkdir -> for creating new folders.
touch -> for creating empty files or updating timestamps.

All operations utilized std::filesystem functions like copy_file(), rename(), and remove_all() with proper exception handling. The implementation performed without any errors.
By the end of Day 3, the explorer became fully functional for creating, modifying, and deleting files

just like a real Linux file manager.

Day 4 - Adding File Search Functionality

On the fourth day, a recursive search system was implemented to find files within directories.
The search <filename> command scanned all folders starting from the current directory using std::filesystem::recursive_directory_iterator. Efficient traversal ensured smooth search execution.
It matched file names partially, allowing users to find files even when they only remember part of the name. This enhanced accessibility and improved overall user experience.
This made file management faster and more efficient. Now lets see the example.
For example, searching for 'demo' displayed both demo.txt and final_demo.txt - validating successful search traversal.

Day 5 - File Permission and Detailed Info Management

The final day involved integrating file permission control and detailed information display.
The chmod <octal> <path> command allowed users to modify file access permissions using octal notation (e.g., 755). Proper validation ensured secure permission handling throughout testing
Permissions were mapped to C++ equivalents through a custom converter to std::filesystem::perms.
The ls -l and stat commands were enhanced to show detailed file information such as size, timestamp, permission bits (rwxr-xr-x), and file type. Each feature was verified through testing.
This made the application behave almost identically to the real Linux command-line environment and completed the project with a comprehensive feature set.

CONCLUSION

This project successfully demonstrates how a C++ program can interact with the Linux operating system to manage files and directories. All essential operations such as navigation, manipulation, search, and permission control were implemented using the C++17 Filesystem library. The project provides a practical understanding of Linux file management commands and system-level programming in C++. It also enhanced skills in exception handling, filesystem traversal, and permission management.

Submitted By :-

Name: Jasvant Panigrahi
Batch: 03
Department: B.Tech (CSE)
Registration Number: 2241016162
Section: 45