Names: Jace Johnson, Xiaohu Luo

Professor: Dr. Stephen W. Turner

Course: CSC 377 section 01

Introduction

This project focuses on the implementation of a simplified file management system that emulates key functionalities of an operating system's file system. The system provides core features such as file creation, deletion, reading, and writing, along with robust directory management. In addition, it supports keyword and file-type-based search operations, simulates device I/O operations, and includes basic error handling for common issues such as missing files or unauthorized access.

The implemented components reflect essential elements of modern file systems, offering a hands-on understanding of how operating systems manage and interact with files, directories, and I/O devices. Commands such as New, Del, View, Edit, Makedir, Changedir, Removedir, and Exit enable users to interact with the file system in a structured and user-friendly manner.

Algorithms

- File Search
  - Iterates over files in the directory and checks for a substring match
  - Simple linear search (brute-force) using string::find
- File Grouping
  - Uses map<string, vector<string>> to categorize files by extension
  - Files are sorted implicitly by key insertion order
- Command Parsing
  - Input is split into a command and an argument using istringstream

- ➢ Leading spaces in the argument are trimmed manually with a lambda-based find_if

- ❖ Device State Management
  - ➢ Devices toggle state using the boolean map and check status before communicating.

Code Design

File Management:

The file management module is responsible for handling file operations:

- **File Creation (New filename)**: Creates a new file with the given name. If a file with the same name already exists in the current directory, the system returns an error.

```
// Command: New <filename>
// Creates a new file if it doesn't already exist
void createFile(const string& filename) {
    // Check if the file already exists using an input file stream (ifstream)
    ifstream checkFile(filename);
    if (checkFile.good()) {
        cerr << "Error: File '" << filename << "' already exists." << endl;
        checkFile.close();
        return;
    }
    // If the file does not exist, proceed to create it using ofstream (output stream)
    ofstream file(filename);
    // Check if the file was created successfully
    if (file) {
        cout << "File '" << filename << "' created successfully." << endl;
    } else {
        cerr << "Failed to create file." << endl;
    }
    // Close the file stream (important to release system resources)
    file.close();
}
```

- **File Deletion (Del filename)**: Deletes an existing file. If the file does not exist, an appropriate error message is displayed.

```
// Command: Del <filename>
// Deletes the specified file if it exists
void deleteFile(const string& filename) {
    // Step 1: Check if the file exists using access()
    // F_OK checks for the file's existence (0 = exists, -1 = does not exist)
    if (access(filename.c_str(), F_OK) != 0) {
        cerr << "Error: File '" << filename << "' does not exist." << endl;
        return;
    }

    // Step 2: Try to remove the file using remove()
    // remove() returns 0 on success, non-zero on failure
    if (remove(filename.c_str()) == 0) {
        cout << "File '" << filename << "' deleted successfully." << endl;
    } else {
        perror("Error deleting file");
    }
}
```

- **File Reading (View filename)**: Allows the user to view the contents of a file.

```
// Command: View <filename>
// Displays the content of the file line by line
void viewFile(const string& filename) {
    // Try to open file
    ifstream file(filename);
    // Check if the file was opened successfully
    if (!file) {
        cerr << "Error: Cannot open file '" << filename << "' for reading." << endl;
        return;
    }

    // If the file is open, read it line by line and print to the console
    string line;
    cout << "Contents of '" << filename << "':\n";
    while (getline(file, line)) {
        cout << line << endl;
    }
    // Close the file stream to release system resources
    file.close();
}
```

- **File Writing (Edit filename)**: Enables users to write or append content to an existing file.

```cpp
// Command: Edit <filename>
// Opens the file and appends lines until ":wq" is typed
void editFile(const string& filename) {
    // Open the file in append mode using ofstream
    // ios::app ensures new content is added at the end without overwriting existing content
    ofstream file(filename, ios::app);

    if (!file) {
        cerr << "Error: Cannot open file '" << filename << "' for writing." << endl;
        return;
    }

    cout << "Enter text to append to '" << filename << "' (type ':wq' to save and quit):" << endl;

    string line;
    // Read input from user line-by-line
    while (getline(cin, line)) {
        // If the user types ":wq", exit the editing loop
        if (line == ":wq") break;
        // Otherwise, write the line to the file and move to the next line
        file << line << endl;
    }

    // Close the file to save changes and release resources
    file.close();

    cout << "Changes saved to '" << filename << "'." << endl;
}
```

- **List files (Filelist)**: List the contents of the directory and sort by the file type.

```cpp
// Command: Filelist
// Lists all files in the current directory, grouped by file extension
void listFiles() {
    // Open the current directory using opendir(".")
    DIR* dir = opendir(".");
    struct dirent* entry;// Struct to hold directory entry info

    // Check if the directory was opened successfully
    if (!dir) {
        perror("Could not open current directory");
        return;
    }

    // Create a map to store files grouped by extension
    // Key   = file extension (e.g., "txt", "cpp")
    // Value = vector of filenames with that extension
    map<string, vector<string>> fileMap;

    // Iterate through directory entries using readdir()
    while ((entry = readdir(dir)) != nullptr) {
        string name = entry->d_name;
        if (name == "." || name == "..") continue;

        // Find the last '.' in the filename to get the file extension
        size_t pos = name.find_last_of(".");

        // If there's a '.' found, extract extension after it
        // Otherwise, use "no_extension" as the key
        string ext = (pos != string::npos) ? name.substr(pos + 1) : "no_extension";

        // Add the filename to the corresponding extension group
        fileMap[ext].push_back(name);
    }

    // Close the directory stream after reading all entries
    closedir(dir);

    // Print the grouped file list
    cout << "Files grouped by file type:\n";
    for (const auto& pair : fileMap) {
        cout << "\n[" << pair.first << "]\n";// Print extension group
        for (const string& file : pair.second) {
            cout << "  " << file << endl;// Print each file under that group
        }
    }
}
```

Directory Management:

The system supports a hierarchical directory structure to organize files efficiently:

- **Create Directory (Makedir)**: Adds a new directory within the current directory.

```
// Command: Makedir <dirname>
// Creates a new directory with default permissions
void makeDirectory(const string& dirname) {

    // Attempt to create the directory with permission mode 0755
    // 0755 means: Owner can read/write/execute; Group and Others can read/execute
    if (mkdir(dirname.c_str(), 0755) == 0) {
        cout << "Directory '" << dirname << "' created." << endl;
    } else {
        perror("Failed to create directory");
    }
}
```

- **Change Directory (Changedir)**: Navigates into a specified subdirectory.

```
// Command: Changedir <dirname>
// Changes current working directory
void changeDirectory(const string& dirname) {
    // If empty or matches originalDir, go back
    if (dirname.empty() || dirname == originalDir) {
        if (chdir(originalDir.c_str()) == 0) {
            // If successful, get and display the current working directory
            char cwd[1024];
            getcwd(cwd, sizeof(cwd));
            cout << "Returned to original directory: " << cwd << endl;
        } else {
            perror("Failed to return to original directory");
        }
        return;
    }

    // Otherwise, try changing to specified directory
    if (chdir(dirname.c_str()) == 0) {
        // On success, get and display the new working directory
        char cwd[1024];
        getcwd(cwd, sizeof(cwd));
        cout << "Changed to directory: " << cwd << endl;
    } else {
        perror("Failed to change directory");
    }
}
```

- **Remove Directory (Removedir)**: Deletes a directory if it is empty.

```cpp
// Command: Removedir <dirname>
// Deletes an empty directory
void removeDirectory(const string& dirname) {
    // Attempt to remove the directory using rmdir (only works if directory is empty)
    if (rmdir(dirname.c_str()) == 0) {
        cout << "Directory '" << dirname << "' removed." << endl;
    } else {
        perror("Failed to remove directory");
    }
}
```

- **Exit (Exit)**: Exits the file system simulation.

```cpp
        } else if (command == "Exit" || command == "exit") {
            cout << "Exiting... Goodbye!" << endl;
            break;
```

Search Functionality:

To enhance usability, the system supports:

- **Keyword Search**: Locate files by searching for specific keywords in filenames or file contents.
- **File Type Search**: Search for files based on extensions such as .txt, .cpp, or .pdf, or by file type category (e.g., document, image, text).

```cpp
// Command: Search <keyword>
// Searches for files that contain the keyword in their names
void searchFiles(const string& keyword) {
    // Open the current directory
    DIR* dir = opendir(".");
    // Check if directory was opened successfully
    if (dir == nullptr) {
        perror("Could not open current directory");
        return;
    }

    bool found = false;// Flag to check if any matching file is found
    struct dirent* entry;// Pointer to hold directory entry data

    // Iterate over all entries in the directory
    while ((entry = readdir(dir)) != nullptr) {
        if (string(entry->d_name).find(keyword) != string::npos) {
            cout << entry->d_name << endl;
            found = true; // Set the flag to true as a match is found
        }
    }

    if (!found) cout << "No matching files found." << endl;

    closedir(dir);
}
```

I/O Handling:

Simulated device communication is achieved through I/O handling mechanisms. This includes basic interrupt handling to model how real-world systems interact with hardware devices asynchronously.

Activate device

```
// Command: Device <deviceName>
// Simulates communication with a device and triggers interrupt
void communicateWithDevice(const string& device) {
    // Check if the device is already active
    if (deviceActive[device]) {
        cout << "Communicating with " << device << "...\n";
        simulateInterrupt(device);
    } else {
        cout << "Device '" << device << "' not active. Initializing...\n";
        deviceActive[device] = true;
    }
}
```

Kill device

```
// Command: Kill <deviceName>
// Turns off communication with the specified device
void killDevice(const string& device) {
    if (deviceActive[device]) {
        cout << "Killing communication with device '" << device << "'...\n";
        deviceActive[device] = false;
    } else {
        cout << "Device '" << device << "' is already inactive.\n";
    }
}
```

Error Handling:

The system includes comprehensive error detection and reporting for scenarios such as:

- Attempting to access a non-existent file
- Trying to delete non-empty directories)
- Trying to create a directory or file that already exists

Main Program

```
// Main command loop
int main() {
    // Store the original working directory
    char cwd[1024];
    getcwd(cwd, sizeof(cwd));
    originalDir = cwd;

    string input;
    cout << "Enhanced File System + Device & Directory Management (Linux)\n";
    cout << "Commands: New, Del, View, Edit, Filelist, Search, Device <dev>, Kill <dev>\n";
    cout << "          Makedir, Changedir, Removedir, Exit\n";

    while (true) {
        cout << "\n> ";
        getline(cin, input);// Read full line of user input

        istringstream iss(input);
        string command;
        string argument;

        iss >> command;// Extract the command keyword
        getline(iss, argument);// Extract the rest of the input as an argument

        // Trim leading spaces from argument
        argument.erase(argument.begin(), find_if(argument.begin(), argument.end(), [](unsigned char ch) {
            return !isspace(ch);
        }));
```

```
        // Command Dispatcher
        if (command == "New") {
            createFile(argument);
        } else if (command == "Del") {
            deleteFile(argument);
        } else if (command == "View") {
            viewFile(argument);
        } else if (command == "Edit") {
            editFile(argument);
        } else if (command == "Filelist") {
            listFiles();
        } else if (command == "Search") {
            searchFiles(argument);
        } else if (command == "Device") {
            communicateWithDevice(argument);
        } else if (command == "Kill") {
            killDevice(argument);
        } else if (command == "Makedir") {
            makeDirectory(argument);
        } else if (command == "Changedir") {
            changeDirectory(argument);
        } else if (command == "Removedir") {
            removeDirectory(argument);
        } else if (command == "Exit" || command == "exit") {
            cout << "Exiting... Goodbye!" << endl;
            break;
        } else {
            // Handle invalid command
            cout << "Invalid command. Try: New, Del, View, Edit, Filelist, Search, Device, Kill, Makedir, Changedir, Removedir, Exit" << endl;
        }
    }

    return 0;
}
```

Challenges

One of the main challenges we faced was ensuring proper error handling when trying to create a file that already exists with the same name, or trying to run a command on a file that does not exist. Initially, creating a new file with the same name would overwrite the previous file without warning, and simple errors like a file not existing would terminate the program.

Another major challenge was implementing the directory commands. When we first created our program, we assumed that it would be in one directory, so when we got to creating the directory commands, it raised several issues and questions about how we wanted our program to behave. This led to us having to rewrite many of our functions to align with the new behavior. We also had a problem with our change directory command not allowing us to back out of a directory without typing the full path to the directory.

Testing

We tested each command in the system, along with the corresponding fail cases, to ensure that appropriate error messages were displayed when invalid input was provided. This included testing file creation, deletion, viewing, editing, device communication, and directory management. We also tested the overall system to verify that it met the functionality outlined in our project proposal. Additionally, we created test scripts to generate a large number of files, which were useful for testing the Search command.

Feature	**Test Case**	**Expected Result**
File creation	New test.txt	**Output:** File 'test.txt' created successfully. **Fail Output:** Error: File 'test.txt' already exists.
File deletion	Del test.txt	**Output:** File 'test.txt' deleted successfully. **Fail Output:** Error: File 'test.txt' does not exist.
File viewing	View test.txt	**Output:**

		Contents of 'test.txt': **Fail Output:** Error: Cannot open file 'test.txt' for reading.
File editing	Edit test.txt	**Output:** Enter text to append to 'test.txt' (type ':wq' to save and quit): **Fail Case:** If test.txt does not exist, it will create a new file and edit into it similar to nano.
File search	Search test .txt	**Output:** test.txt (if a specific name is provided, it will only show one file) **Fail Output:** No matching files found.
Make directory	Makedir testdir	**Output:** Directory 'testdir' created. **Fail Output:** Failed to create directory: File exists
Change directory	Changedir testdir	**Output:** Changed to directory: /home/administrator/Project/testdir **Fail Output:** Failed to change directory: No such file or

		directory
Remove directory	Removedir testdir	**Output:** Directory 'testdir' removed. **Fail Output:** Failed to remove directory: No such file or directory
Activate device	Device printer	**Output:** Device 'printer' not active. Initializing... **Output (if device is active):** Communicating with printer... [INTERRUPT] Received from device: printer
Kill device	Kill printer	**Output:** Killing communication with device 'printer'... **Fail Output:** Device 'printer' is already inactive.
List directory	Filelist	**Output:** Files grouped by file type: [cpp] file_management.cpp file_management_new.cpp [no_extension] file_management file_management_new

		[sh] generate_test_files.sh delete_test_files.sh [txt] test.txt

Test Scripts used

This is a script that we have written to generate test files.

```
  GNU nano 5.6.1                          generate_test_files.sh
#!/bin/bash

# Number of files to create
NUM_FILES=100

# File extensions to randomly pick from
EXTENSIONS=("pdf" "doc" "docx" "png" "cpp")

echo "Generating $NUM_FILES random files..."

for ((i=1; i<=NUM_FILES; i++)); do
    EXT=${EXTENSIONS[$RANDOM % ${#EXTENSIONS[@]}]}
    FILENAME="testfile_$RANDOM.$EXT"
    touch "$FILENAME"
    echo "Created: $FILENAME"
done
```

This is a script that we have written to remove all the test files generated by the test file script.

```
  GNU nano 5.6.1                          delete_test_files.sh
#!/bin/bash

echo "Deleting generated test files..."

# You can delete only files that match a certain pattern
rm -v testfile_*.pdf testfile_*.doc testfile_*.docx testfile_*.png testfile_*.cpp

echo "Cleanup complete."
```

Conclusion

We successfully implemented key components such as file management, device communication, and directory operations. We overcame several challenges, particularly with error handling and directory navigation. We ensured that the system behaved as expected by testing all of the commands with the help of a testing script that created files to search through.

Additionally, we designed a file listing function that enables users to sort and display files within a directory. This functionality not only improves the organization and accessibility of file data but also enhances the overall user experience with a file system. This project gave us a deeper understanding of how users interact with operating systems and strengthened our skills in system-level programming, debugging, and design.