Assignment 2: In-Memory (Tree-Based) File System

Due: May 16, 2025 23:55 (Friday)

Objectives

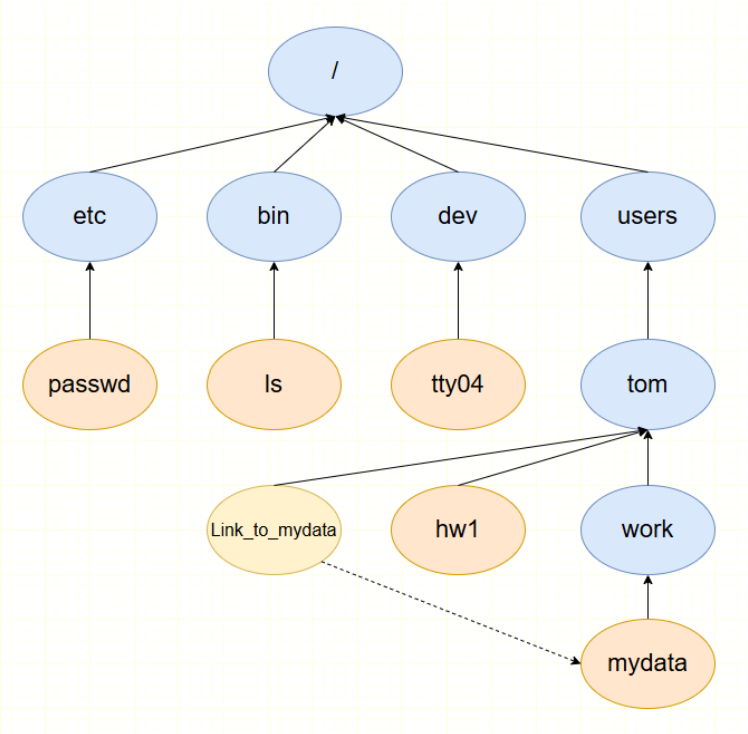

Figure 1: File system hierarchy

Objectives

This assignment aims to develop students' understanding of file system concepts through practical implementation. Students will:

- Master object-oriented design principles by implementing a polymorphic tree structure with distinct node types (files, folders, links) and applying inheritance, encapsulation, and proper memory management
- Develop core file system algorithms including path resolution, tree traversal, and symbolic link handling that mirror real-world UNIX file system operations
- Implement comprehensive file manipulation operations such as creating, reading, copying, moving, and deleting nodes within the hierarchical structure while ensuring proper error handling
- Apply advanced C++ programming techniques including deep copying of complex structures, proper resource management with constructors/destructors, and polymorphic behavior through virtual functions

I. Assignment Overview

Students will implement an abstract file system in C++ using a tree structure. Each node in the tree is one of:

- **File**: holds a `std::string content`.
- **Folder**: contains any number of child `Node*`.
- **Link**: symbolic reference to another `Node*` (file, folder, or link).

II. Core Classes & Headers

1. Node (abstract base)

The `Node` class is an abstract base class that represents a generic element in the file system hierarchy. Each node has a name and an optional pointer to its parent. Derived classes such as `File`, `Folder`, and `Link` inherit from `Node` and share its common properties.

1.1 Node(const std::string& name, Node* parent = nullptr);

The constructor initializes the node's name and parent. You can see that the `name` is the name of the node, and `parent` points to the parent folder in the hierarchy.

1.2 virtual ~Node();

Virtual destructor to ensure proper cleanup.

1.3 const std::string& get_name() const;

You should return the name of the node.

1.4 Node* get_parent() const;

You should return a pointer to the node's parent.

1.5 void set_parent(Node* parent);

You should assign the specified parent node as the parent of the current node.

1.6 void set_name(const std::string &name);

You should set the name of the current node to the given string name.

1.7 virtual NodeType node_type() const=0;

Pure virtual function that you should implement in derived classes to identify the node type (e.g., File, Folder, Link).

1.8 virtual Node* clone(Node* parent = nullptr) const = 0;

Pure virtual method for deep copying nodes.

2. File

The `File` class is a subclass of `Node` and represents a file in the virtual file system. It stores textual content (as a private member) and allows for reading and writing operations. Each `File` object has a name, a parent folder, and content.

2.1 File(const std::string& name, Node* parent = nullptr, const std::string& content = "");

This is the constructor for the `File` class. Initialize a file with a given name, an optional parent node, and an optional initial content. It will be used to create and insert file objects into the file system tree. You should make necessary initializations in this function.

2.2 ~File();

Destructor to ensure proper cleanup when a file is deleted.

2.3 NodeType node_type() const override;

You should return the type of node. It helps to distinguish between other node types during operations.

2.4 Node* clone(Node* parent = nullpt) const override;

In this method you should create and return a deep copy of the file, including its content. You can use this method to copy a file with the same name and content, and assign the given parent.

2.5 const std::string& read_content() const;

You should return a constant reference to the file content, allowing read-only access.

2.6 void write_content(const std::string& data);

You should replace the current file content with the new `data` string. This method replaces the private content of the File by replacing it with a new string.

3. Folder

The `Folder` class is a subclass of `Node` and represents a directory in the file system tree. It can contain other nodes such as files, folders, and links as its children. It manages its children using a `std::vector` and supports adding, removing, searching, and listing child nodes.

3.1 Folder(const std::string& name, Node* parent = nullptr);

This constructor creates a folder node with a specified name and parent.

3.2 ~Folder();

This is the destructor for the `Folder` class. You should ensure that all child nodes owned by this folder are properly deleted to avoid memory leaks.

3.3 NodeType node_type() const override;

You should return the type of node.

3.4 Node* clone(Node* parent = nullptr) const override;

You should create a deep copy of the folder along with all its child nodes. This method can be used for operations that involve duplicating folder structures. The `parent` parameter specifies the parent node for the cloned folder. It returns a pointer to the newly created Folder clone.

3.5 Node* add_file(const std::string &name, const std::string &content);

Creates a new `File` node with the specified name and content, and adds it to the folder's `children_` vector. Before insertion, you should check that the name is not empty and that no other child with the same name and node type exists in the folder. If either condition fails, an error is thrown.

3.6 Node* add_folder(const std::string& name);

Creates a new empty `Folder` node with the given name and adds it as a child to the parent folder. Before adding to the folder, you should check that the name of the folder is not empty and a child with the same name and type does not already exists. If either condition fails, an error is thrown.

3.7 Node* add_link(const std::string& name, Node* target);

Creates a new `Link` node that points to the specified node (`target`) and adds it as a child to the folder. Before adding the link, you should check that the name of the link is not empty, the `target` is not `nullptr`, and no child with the same name and type already exists. If any of these conditions fail, an error is thrown.

3.8 void add_child(Node* child);

Adds the given child node to the folder's list of children.

3.9 Node* remove_child(const std::string& name);

In this method, you should remove a child node with the specified name from the folder's `children_` vector. If a matching node is found and removed, a pointer to it should be returned. If no child with the given name exists, it should throw an error.

3.10 Node* find_child(const std::string& name) const;

You should search for a child node by name and return a pointer to it if found. Return `nullptr` if no matching child exists.

3.11 std::vector<std::string>list_children() const;

You should return the list of the names of all child nodes contained within the folder.

4. Link

The `Link` class is a subclass of `Node` and represents symbolic links in the file system. A link does not store its own content, but instead points to another node (either a file, folder, or another link). In addition to a name and a pointer to its parent node, it maintains a pointer to its target node.

4.1 Link(const std::string& name, Node parent, Node* target);

This constructor initializes a `Link` object with the specified name, parent folder, and target node.

4.2 ~Link();

This is the destructor for the `Link` class for proper cleanup. Note that the link does not own its target node so, it does not delete it.

4.3 NodeType node_type() const override;

You should return the type of node.

4.4 Node* clone(Node* parent = nullptr) const override;

In this method, you should create a clone of the `Link` node, pointing to the same target as the original. It returns the pointer to the newly created Link clone.

4.5 Node* get_target() const;

You should get the target node that the link points to and return the pointer to the target node.

5. FileSystem

The FileSystem class provides a complete interface for managing an in-memory file system, including operations like creating, moving, copying, and removing nodes, as well as mounting and unmounting external file systems.

5.1 FileSystem();

This is the default constructor. You should initialize the file system with an empty root folder named "/".

5.2 FileSystem(const FileSystem& other);

This is the copy constructor. You should create a deep copy of the given FileSystem object, duplicating the entire internal tree structure rooted at the original file system's root folder. Note that mounts are not copied as they reference external filesystem objects.

5.3 FileSystem& operator=(const FileSystem& other);

This is the assignment operator. You should create a deep copy of another FileSystem object and assign it to this object. Note that mounts are not copied as they reference external filesystem objects.

5.4 ~FileSystem();

This is the destructor, it deletes all nodes in the file system and frees their memory.

5.5 Node* resolve_path(const std::string& path) const;

In this method, you should traverse the file system tree following the given absolute path using '/' as the seperator. Your method should handle navigation through links as well. Return the node at the target path, or nullptr if the path is invalid.

5.6 void recursive_delete(Node* node);

You should recursively delete a node and all its childeren (if any) and free the allocated memory for the node and its childeren.

5.7 void create_file(const std::string& path, const std::string& content = "");

In this method, you should create a new file with the given content at the specified path. If the parent folder does not exist, a "No such file or directory." error is thrown. If a file with the same name already exists in the parent directory, a "A file with this name already exists." error is thrown. Otherwise, it creates the new file.

5.8 void create_folder(const std::string& path);

You should create a new folder at the specified path. It extracts the parent directory and folder name from the path. If the parent directory does not exist, it throws a "No such file or directory." error. If a folder with the same name already exists in the parent, a "A folder with this name already exists." error is thrown. Otherwise, the new folder is added to the parent directory.

5.9 void create_link(const std::string& path, const std::string& target_path);

You should create a symbolic link at the specified path, pointing to the target path. If the parent directory does not exist, it throws a "No such file or directory." error. If the target node does not exist, it throws a "Target node not found." error. If a link with the same name already exists in the parent directory, it throws a "A link with this name already exists." error. Otherwise, it adds the new link to the parent folder.

5.10 bool remove_node(const std::string& path);

You should remove the node at the given path from the file system, including all of its children (if it is a folder). Note that the root cannot be removed. If the node was successfully removed return true, otherwise return false.

5.11 void move_node(const std::string& src_path, const std::string& dest_path);

In this method you should move a node from src_path to dest_path. It first resolves the source node and destination parent folder. If the source node does not throw an error. If the destination parent folder is not found, an error is thrown. Otherwise, remove the node from its old parent, updates its name and parent, and adds it to the new destination folder.

5.12 void copy_node(const std::string& src_path, const std::string& dest_path);

You should copy a node from source path to the destination path. You should first resolve the source node and the destination folder.If either is not found throw an error. The source node should be cloned and added to the destination folder with its new name. If a node with the same name and type already exists at the destination, it should be removed before copying.

5.13 Node* find_node(const std::string& path) const;

You should find a node at the specified path. Return a pointer to the node if it is valid, otherwise return nullptr.

5.14 std::vector<std::string >list_directory(const std::string& path) const;

You should return a vector containing the names of all the items inside a directory. If the directory is empty return an empty vector.

5.15 void print_tree(const std::string& path = "/", int indent = 0) const;

You should print the file system tree starting from the specified path. It recursively prints the tree structure of the file system with indentation. Note that each level in the directory tree is indented by 2 spaces. The print_tree function uses the indent parameter to control this spacing. Example output can be as follows:

```
/
  home
    user
      file.txt
```

III. Student Deliverables & Required Implementations

1. Implement all public member functions in the headers (constructors, destructors, clone, helpers).
2. Ensure deep-copy semantics for copy constructor/operator= and proper memory cleanup in destructors.
3. Path semantics:
   - Absolute UNIX-style paths (e.g. `/a/b/c`).
   - Auto-create intermediate folders on create operations.
   - Throw `std::invalid_argument` for invalid paths or type mismatches.
4. Required operations:
   - File: `read_content()`, `write_content()`.
   - Folder: `add_file()`, `add_folder()`, `add_link()`, `add_child()`, `remove_child()`, `find_child()`, `list_children()`.
   - Link: `get_target()`.

- FileSystem: `resolve_path()`, `recursive_delete()`, `create_file()`, `create_folder()`, `create_link()` `remove_node()`, `move_node()`, `copy_node()`, `find_node()`, `list_directory()`, `print_tree()`.

5. All folder-level operations must use the public helper methods (no direct access to children vector).
6. Covere edge cases such as invalid paths, double slash handling, moving/copying to a non-existent location, creating a duplicate node, removing non-existent node, etc.).

IV. Programming Tasks

Implement the following methods in `sneaky_case` exactly as declared.

A. Node Class

1. Constructor & Destructor
   - `Node(const std::string& name, Node* parent = nullptr)`
   - `virtual  Node()`
2. Accessors: `get_name()`, `get_parent()`.
3. Helpers: `set_name()`, `set_parent()`
4. Abstract: leave `node_type()` and `clone()` pure virtual.

B. File Class

1. `node_type()` returns `NodeType::File`.
2. `clone()` returns a new `File(name_, nullptr, content_)`.
3. Helpers: `read_content()`, `write_content(const std::string& data)`.

C. Folder Class

1. Destructor: delete all `children_`.
2. `node_type()` returns `NodeType::Folder`.
3. `clone()`: deep-copy children, setting new parent.
4. Helpers: `add_file`, `add_folder`, `add_link`, `add_child`, `remove_child`, `find_child`, `list_children`.

D. Link Class

A `Link` is a node that acts as a shortcut or symbolic reference to another existing node (file, folder, or another link) in the file system. It does not store its own data but points to another node.

1. `node_type()` returns `NodeType::Link`.
2. `clone()` returns new `Link(name_, nullptr, target_)`.
3. Accessors: `get_target()`.

E. FileSystem Class

1. `resolve_path(path)`: support "/" seperator.
2. `recursive_delete(node)`: delete subtree.
3. `create_file`, `create_folder`, `create_link`.
4. `remove_node(path)`.
5. `move_node(src, dest)`.
6. `copy_node(src, dest)`.
7. `find_node(path)`.
8. `list_directory(path)`.
9. `print_tree(path, indent)`.

Submission

Submit all the `.cpp` files —i.e. `Node.cpp`, `File.cpp`, `Folder.cpp`, `Link.cpp`, and `FileSystem.cpp`— in a single archive via the odtuclass course page. Alternatively, `save` button in the **VPL** environment handles the submission automatically. Only the last submission before the deadline will be graded.

Good luck!