Welcome to PA-1 Learning Journey! 🚀

ⓘ **Welcome!**

👋 Welcome to the **Advanced Programming** Spring 2025 course PA-1 project!
😊 This document serves as the lab manual to help you successfully complete this PA project.
😎 Through this PA course, you will learn and master the following:

- 🎯 Deepen your understanding of **object-oriented** programming concepts.
- 💡 Master the design and implementation of C++ classes.
- 🔧 Learn to build a complete software system.

⚠ **Warning**

The `PA-1` project strictly prohibits plagiarism, including **direct copying, using someone else's code without proper attribution, or failing to cite sources**. Plagiarism may result in a zero score, academic penalties, or more severe consequences. To avoid plagiarism, please complete the project independently, properly cite any external sources used, and comply with open-source licenses.

✓ **Let's Start Now!**

🎉 Let's dive into the exciting world of **Virtual File Systems**!

PA-1: Virtual File System

ⓘ **Virtual File System with Command Line Interface (VFS)** 🌐

In this project, you will build a **Virtual File System (VFS)** that simulates real-world file management. The system will include:

- **FileSystem Class**: Manages files and directories.
- **ClientInterface Class**: Simulates user command-line interactions.
- **VFS**: Integrates and manages the entire system.

0. Background

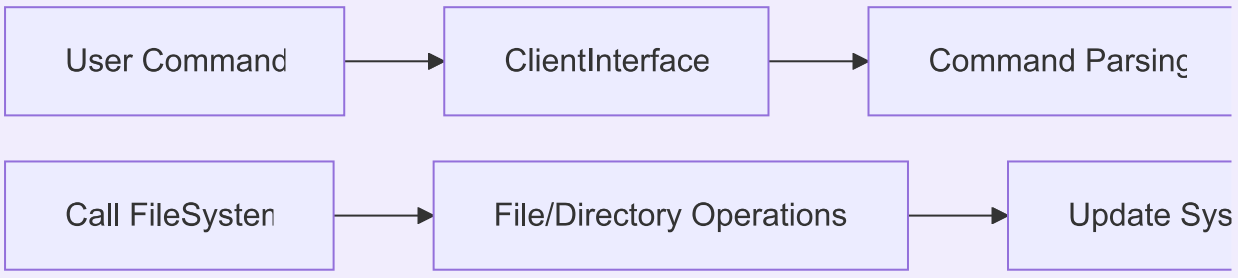
What is a Virtual File System (VFS)?

A **Virtual File System (VFS)** is a software implementation that simulates a real file system. Using object-oriented design, VFS provides basic file and directory management functionalities, helping you understand core file system concepts and object-oriented programming principles.

How VFS Works

1. **File Abstraction**: Files and directories are abstracted into classes (`File`, `Directory`) using a base class `FileObj` for inheritance and polymorphism.
2. **Directory Organization**: The `FileSystem` class organizes files and directories, supporting flexible path navigation.
3. **User Interaction**: The `ClientInterface` provides a command-line interface with Unix-style file operations.
4. **System Integration**: The `VFS` integrates `FileSystem` and `ClientInterface` for unified management.

System Flowchart



1. Core Components

Key Components

Component	Description
`ClientInterface`	Handles command parsing and user interaction.
`FileSystem`	Implements file and directory management.

Component	Description
`FileObj and Derived Classes`	Provides concrete implementations for files and directories.
`VFS`	Integrates all components for unified management.

Project Task

Use **object-oriented programming** to implement the three core components and build a complete **VFS** system.

---

2. Learning Objectives

What You Will Learn

Objective Type	Details
📚 **Class Design**	Master C++ class design and implementation.
🧩 **Inheritance & Polymorphism**	Understand and apply inheritance and polymorphism.
🧠 **System Building**	Use composition and nested classes to build complex systems.
🔒 **Encapsulation**	Implement access control and encapsulation.

---

3. Prerequisites

What You Should Know

Before starting PA-1, familiarize yourself with the following:

- **Data Structures**:
  - Arrays: `vector<T>` and its methods (`push(T)` , `pop()`).
  - Dictionaries: `unordered_map<key, value>` and CRUD operations.
  - Sets: `set<T>` and its methods (`find(T)` , `insert(T)`).
- **Data Types**: `enum` , `size_t` .
- **Strings**: `string` .
- **Input/Output Streams**: `istringstream` .

- **OOP Principles**: Inheritance, polymorphism, encapsulation, and abstraction.

💡 **Pro Tip**: Always think one level of abstraction ahead during design!

---

4. Project Structure

ⓘ **FileSimulator Project Tree**

```
FileSimulator/
├── CMakeLists.txt
├── include/
│   ├── FileObj.h
│   ├── Directory.h
│   ├── FileSystem.h
│   ├── VFS.h
│   └── ClientInterface.h
├── src/
│   ├── main.cpp
│   ├── FileObj.cpp
│   ├── Directory.cpp
│   ├── FileSystem.cpp
│   ├── VFS.cpp
│   └── ClientInterface.cpp
└── README.md
```

🎯 5. Experimental Guidelines

5-0. InodeFactory Class

ⓘ **InodeFactory Class (`include/InodeFactory.h`)**

cpp

```cpp
class InodeFactory {
public:
    static uint64_t generateInode() {
        static uint64_t nextInode = 1;
        return nextInode++;
    }
};
```

- In the VFS, all system objects (`File` and `Directory`) have a unique `inode` identifier of type `uint64_t` (unsigned long long).
- We provide the `generateInode()` API for you to use. By default, the root directory `/` has `inode = 1`.
- Any newly created file or directory will have `inode >= 2`. (Note: The VFS does not recycle `inode` values, so they only increase and never repeat.)

5-1. FileObj (Base Class)

FileObj Class (`include/FileObj.h`)

```
class FileObj {
    string name;     // File/Directory name
    string path;     // Absolute path
    string type;     // "file" or "directory"
    string owner;    // Owner
    uint64_t inode;  // Unique identifier
    FileObj* parent; // Parent directory
};
```

- **Attributes**:
  - `inode` is generated by `InodeFactory`.
  - `parent` points to the parent directory (for the root directory `/`, `parent` is `nullptr`).
- We have implemented the constructor for `FileObj`. You can use it as a reference for implementing constructors in other classes.

5-2. File Class

File Class (`include/File.h`)

```
class File : public FileObj {
protected:
    string content; // Content of the file
};
```

- The `File` class adds a `content` attribute to `FileObj`.
- **Methods to Implement**:

  ```
  File(const string& name, const string& type, const string& owner,
       const uint64_t& inode, FileObj* parent);

  virtual string read() const;             // Read content
  virtual bool write(const string &data); // Write content (append to
  `content`)
  virtual string getContent() const;       // Get content
  ```

- Refer to the framework code for detailed TODOs.

5-3. Directory Class

ⓘ Directory Class (`src/Directory.cpp`)

```
class Directory : public FileObj {
    std::unordered_map<uint64_t, FileObj*> children; // Store child
objects
};
```

- **Attributes**:
  - `children`: An `unordered_map` that maps `inode` to `FileObj*` pointers (stores all child items in the directory).
- **Methods to Implement**:

  ```
  Directory(const string& name, const string& owner,
            const uint64_t& inode, FileObj* parent);

  bool add(FileObj* child);              // Add a child node
  bool remove(uint64_t inode);           // Remove a file node
  bool removeDir(uint64_t inode);        // Recursively remove a
  directory node
  FileObj* getChild(uint64_t inode);     // Get a specific child node
  std::vector<FileObj*> getAll() const; // Get all child nodes
  size_t getCount() const;               // Get the number of child
  nodes
  ```

```
bool isEmpty() const;                  // Check if the directory is
empty
```

- **Hint**: Use `inode` as the unique identifier for removing nodes. Refer to the framework code for detailed TODOs.

5-4. FileSystem (Core Functionality)

ⓘ FileSystem Class (`include/FileSystem.h`)

```
class FileSystem {
    Directory* root;     // Root directory
    Directory* cur;      // Current directory
    string username;     // Current user
    std::set<string> users;  // All users
    std::unordered_map<string, uint64_t> config_table;  // Path to
inode mapping
};
```

- **Attributes**:
    - `root` : The root directory of the file system.
    - `cur` : The current directory.
    - `username` : The current user.
    - `users` : A set of all registered users.
    - `config_table` : Maps an object's **absolute path + type** to its `inode` .
- **Methods to Implement**:

```
// Directory navigation
changeDir(const uint64_t& inode);      // Change current directory
getCurrentPath() const;                // Get current path
resolvePath(const string& path);       // Resolve path

// File operations
createFile(const string& name);        // Create a file
deleteFile(const string& name, const string& user);  // Delete a
file

// Directory operations
createDir(const string& name);         // Create a directory
deleteDir(const string& name, const string& user, bool recursive);
// Delete a directory
// Search and user management
search(const string& name, const string& type);        // Search for
a file or directory
setUser(const string& username);                       // Set the
current user
hasUser(const string& username);                       // Check if a
user exists
registerUser(const string& username);                  // Register a
new user
```

- **Hints**:
    1. No permission control is needed for creating files/directories, but deletion requires permission (only the `owner` or `root` user can delete).
    2. Use `resolvePath()` in `changeDir()`. Parse paths using `strtok()` or `istringstream`.
    3. Maintain both `children` and `config_table` when adding or removing items.
    4. Different users share the same file system. Manage the `users` set using `insert()` or `remove()`.

5-5. ClientInterface (User Interface)

ⓘ **ClientInterface Class (`include/ClientInterface.h`)**

```
class ClientInterface {
    FileSystem* filesystem;  // FileSystem instance
    string username;         // Current user
};
```

- **Design**: The `ClientInterface` class follows the **Visitor Pattern**. Users interact with the file system through this interface without directly embedding it.
- **Methods to Implement**:

```
// Command processing
parseCommand(const string& cmdLine);    // Parse command line
execueCommand(const vector<string>& cmd);   // Execute command
processCommand(const string& cmdLine);      // Process command

// File operations
createFile(const string& name);            // Create a file
deleteFile(const string& name);            // Delete a file
readFile(const string& name);              // Read a file
```

```
writeFile(const string& name, const string& data);  // Write to a
file

// Directory operations
createDir(const string& name);            // Create a directory
deleteDir(const string& name, bool recursive);  // Delete a
directory
changeDir(const string& path);            // Change directory
listCurrentDir();                         // List directory contents
getCurrentPath() const;                   // Get current path

// Other commands
showHelp() const;                                  // Show help
information
search(const string& name, const string& type);    // Search for a
file or directory
```

- **Command Processing**:
  - `parseCommand`: Splits the command line into tokens.
  - `execueCommand`: Executes the command based on the first token (`cmd[0]`).
  - `processCommand`: Combines `parseCommand` and `execueCommand`.

5-6. Commands to Implement

Supported Commands

```
# File operations
create <filename...>      # Create one or more files
delete <filename...>      # Delete one or more files
read <filename...>        # Read one or more files
write <filename> <text>   # Write to a file (supports multi-line text
and escape characters)

# Directory operations
mkdir <dirname>          # Create a directory
rmdir [-r] <dirname>     # Delete a directory (-r for recursive
deletion)
cd <path>                # Change directory (supports relative and
absolute paths)
ls                       # List directory contents
pwd                      # Print current working directory
clear                    # Clear the terminal
```

```
# System commands
help                       # Show help information
exit                       # Log out
quit                       # Exit the system
```

5-7. Notes and Warnings

Error Handling

- Handle errors such as:
  - File already exists
  - Path does not exist
  - Insufficient permissions
  - Invalid parameters

User Management

- User creation
- Permission verification
- Session management

File Operations

- Handle empty files
- Recursive deletion
- Path resolution
- Permission verification

Memory Management

- Properly release objects
- Avoid memory leaks
- Prevent dangling pointers

🚀 6. Build and Execution

6-1. Windows Environment

Windows Build and Execution

1. **Generate the Project**:

```
# Run in the project root directory
mkdir build
cd build
cmake ..
cmake --build .
```

2. **Run the Program**:

```
# In the build/bin or build/bin/Release directory
FileSimulator.exe
```

6-2. Linux Environment

Linux Build and Execution

1. **Install Dependencies**:

   ```
   # Ubuntu/Debian
   sudo apt-get install build-essential cmake

   # CentOS/RHEL
   sudo yum groupinstall "Development Tools"
   sudo yum install cmake
   ```

2. **Generate and Compile**:

   ```
   mkdir build
   cd build
   cmake ..
   make
   ```

3. **Run the Program**:

```
# In the build/bin directory
./FileSimulator
```

6-3. Common Issues and Troubleshooting

⚠ Common Issues

1. **CMake Errors**:
   - Ensure CMake version >= 3.10.
   - Verify that the compiler is correctly installed.
   - Ensure environment variables are properly set.
2. **Compilation Errors**:
   - Check for missing dependencies.
   - Ensure the correct generator is used.
   - Review detailed compilation logs.
3. **Runtime Errors**:
   - Ensure the program is run from the correct directory.
   - Check for missing dynamic libraries.
   - Review the program's error output.

7. Example Code

☰ Example Usage

```
# Start the system
$ ./FileSimulator
File System Simulator Started
Please login with your username
Login: root

# Show help
root@FileSimulator:/$ help
Available commands:
  create <filename...>        - Create one or more new files
  delete <filename...>        - Delete one or more files
  read <filename...>          - Read content from one or more files
  write <filename> <text>   - Write text to file (supports '\\n' for
newline)
  mkdir <dirname>           - Create a new directory
  rmdir [-r] <dirname>      - Remove directory (-r for recursive
deletion)
  cd <path>                - Change directory (supports
relative/absolute paths)
  ls                       - List current directory contents
  pwd                      - Show current working directory
  whoami                   - Show current user name
  clear                    - Clear current command line
  help                     - Show this help message
  exit                     - Logout current user
  quit                     - Exit program

# Some Basic operations
root@FileSimulator:/$ mkdir docs
root@FileSimulator:/$ cd docs
root@FileSimulator:/docs$ create readme1 readme2
root@FileSimulator:/docs$ write readme1 "This is a test file."
root@FileSimulator:/docs$ write readme2 "Hello PA-1"
root@FileSimulator:/docs$ read readme1 readme2
=== readm1 ===
This is a test file.
=== readm2 ===
Hello PA-1
root@FileSimulator:/docs$ pwd
/docs
root@FileSimulator:/docs$ ls
readme1
readme2
root@FileSimulator:/docs$ exit

# exit the program
User Login (Please input who you are):
exit
Bye!
```

Friendly Tips

- **If you encounter any issues during the experiment, please contact the teaching assistant promptly.**
- **Thank you for reading. Good luck with your PA-1 experiment!**