User Guide

1 User Interface Introduction

The C++ interface consists of two classes: `FileSystem` and `File`. A single `FileSystem` is required to access all of the files in the system. It provides the majority of the user interface and manages the "disk" file and working directory:

```
class FileSystem {
    disk_s  m_disk;
    inode_s m_dir;
public: // Disk RAII
    FileSystem(const std::string& disk_file_name="disk.dat");
    ~FileSystem(); // Destructor declaration
public: // Target Interface
    // File Interface
    File open(const std::string&, const uint8_t&);
    void create_file(const std::string&);
    void delete_file(const std::string&);
    void move_file(const std::string&, const std::string&);
    % Directory Interface
    void ls_dir();
    void mk_dir(const std::string&);
    void ch_dir(const std::string&);
};
```

The `FileSystem` object can also be used to retrieve `File` objects, each of which represents an interface to an open file (for reading, writing or both). Files "on disk" can be opened via the `FileSystem::open()` method, which returns a `File` object. The opened file will be automatically closed once this object is destroyed.

```
class File{
        file_s m_file;
        uint32_t end;
public: // File RAII
        File(file_s);
       ~File();
public: // Read / Write Interface
        void write(const std::string&);
```

```
        void write(const std::string&,const uint32_t&);
        std::string read(const uint32_t&,const uint32_t&);
        std::string read();
};
```

2 Files & Directories

2.1 Creation & Deletion

Files and directories can be created by providing their path relative to the current working directory (initially, the first directory read from the disk). This is done by passing the path as an `std::string` object or string literal to the `FileSystem::create_file` and `FileSystem::mk_dir` functions for files and directories, respectively.

Although files and directories are created using different `FileSystem` methods, it is important to note that they are treated mostly the same by other parts of the system - Directories are just a special type of file. File deletion is one example of this: both files and directories are deleted using the same method.

```
#include "filesystem.hpp"
int main(void) {
    FileSystem fs("data.disk");
    // Files & Directories Are Created Differently
    fs.mk_dir("Directory");
    fs.create_file("File");
    // But Deleted The Same Way
    fs.delete_file("Directory");
    fs.delete_file("File");
}
```

2.2 The Working Directory

Almost every `FileSystem` method expects a pathname as an argument. These are all formatted relative to the current working directory, which is expected to be the Root initially (absolute paths are not supported). The `FileSystem` class provides three methods to deal with directories. The first, seen in the previous section, is `FileSystem::mk_dir`, which handles directory creation. The second is `FileSystem::ls_dir`, which lists out the entries in the working directory, along with their Inode numbers (for clarity). The third method is `FileSystem::ch_dir`, which takes the path of a valid directory relative to the current root, and sets that directory as the new working directory.

Each directory also contains a ".." entry, which refers to it's parent. This entry works as you might expect, allowing us to return to the old working directory with another call to `FileSystem::ch_dir`.

```
#include "filesystem.hpp"
int main(void) {
    FileSystem fs("data.disk");
    fs.mk_dir("NewDir");
    fs.ls_dir();     // "NewDir" And ".." Are The Only Expected Entries Here
    fs.ch_dir("NewDir");
    fs.ls_dir();     // This Time, We Expect No Entries
}
```

Note in the example above that the Inode number (in parentheses next to each entry) is the same for the working directory in the second `fs.ls_dir()` call as it was for the `NewDir` in the first. This is an indication that things are working as expected, since it tells us that the working directory has in fact changed, even if we can't see its name.

2.3 Directory Structure & Manipulation

The hierarchical structure of the file / directory tree is not fixed. It can be manipulated by moving any of its entries around different branches. The `FileSystem::move` method allows us to do this very easily. It takes in two relative paths instead of one: the "source path" followed by the "destination path". So `fs.move("Docs/A.txt","Dir/A.txt")` would move file `A.txt` from directory `Docs` into directory `Dir`.

If the filenames specified in the "source" and "destination" fields are different, the file will be renamed. This allows us to use the `FileSystem::move` method to rename a file without moving it, by providing a source and deestination file in the same directory.

```
#include "filesystem.hpp"
int main(void) {
    FileSystem fs("data.disk");
    fs.mkdir("Dir");
    fs.create_file("A");
    fs.ls_dir();     // Expect "Dir", "A", and ".."
    fs.move("A","Dir/B");
    fs.ch_dir("Dir");    // Now Just "B" and ".."
    fs.ls_dir();
}
```

Similarly to the last example, we can see that files A and B have the same I-Number. This confirms that they are in fact the same file, just with a different name. Because we haven't changed anything about the file itself, the `FileSystem::move` method will work with directories as well, without damaging their entries.

It should also be noted that when the destination filename is already taken, `FileSystem::move` will delete the target file, and replace it entirely with the newly moved.

3 File Handles & Read / Write

3.1 Opening Files

While the `FileSystem` class handles the management of the disk and directory structure, `File` objects are used to allow the user to manipulate the contents of a file directly. `File` objects are not instantiated directly by the user, but are instead obtained through the `FileSystem` class's public interface. Specifically the `FileSystem::open` method, which accepts a path to a valid file or directory, relative to the current working directory, and a some flags specifying the permissions (read/write/etc.) that the newly created file handle will receive. `File` class objects manage the underling file descriptor on their own, so we don't need to worry about manually closing the file.

3.2 File Reading

Once a `File` object is obtained, we can use its public interface to perform read and write operations on the file (assuming the `FileSystem::open` method was given the appropriate permission flags). The two functions provided by the `File` interface are `File::read` and `File::write`, which are nearly self-explanatory. Each of these has two overloads.

The first version of `File::read` takes no arguments, and simply returns the entire content of the file. This version does not move the iterator, so it will not interrupt the flow of other read / write operations. The second version accepts two 32-bit unsigned integer parameters: the first indicating an offset into the file to start reading from, and the second specifying a length to read. Both represent a number of bytes.

3.3 File Writing

The final method provided by the C++ Interface is `File::write` which, as previously mentioned, has two overloads: one which takes an `std::string` object or reference (or a string literal) and another which takes the same, along with an unsigned 8-bit integer (or a reference to one). The integer value will once again be interpreted as a byte-offset into the file's data section, and the entirety of the first argument (buffer) will be read into that offset. Note that this will overwrite any content that was previously stored at and around that position.

```
#include "filesystem.hpp"
int main(void) {
    FileSystem fs("data.disk");
    fs.create_file("hello.txt");
    File hello = fs.open(
        "hello.txt",
        FILE_READABLE_BIT |
        FILE_WRITABLE_BIT
    );
```

```
    hello.write("Hello, World!");
    std::cout << hello.read();
    fs.ls_dir();
}
```

In this example, we can see a few features of the interface. First, note that there is no matching `close(...)` call made by the user - that's handled by the `File hello` object itself once it exits the scope. Second, the fs.open(...) call's second argument is specified using two macros (`FILE_READABLE_BIT` and `FILE_WRITABLE_BIT`), which are passed together as one using the bit-wise `|` operator. This operator is needed if we want to perform both read and write operations through the newly created file handle. Otherwise, only one may be specified. Optionally, neither may be specified by passing `0` to the second argument, though this will predictably result in a file handle with which we can perform no useful operations.