CSCI 2270: Data Structures - Project Specifications fileSim (A File System Simulator)

Due December 8, 11:59 PM MDT

Interview Grading December 9-12, by appointment

1 File System Simulator

A file system is an essential part of any operating system. It governs how files and directories are organized within the computer so they can be accessed, modified, executed, and moved on a storage device such as an SSD or a hard drive. For this project, we aim to design a file system simulator that mimics a real-world file system. We use a Tree data structure to represent the overall file system, where each node (Inode), corresponds to a file or a directory. This document provides the minimal capability your tool should have, but you are encouraged to improve upon it to simulate the more sophisticated aspects of a real-world file system. The project is organized into two parts: Core Features and Efficient Search.

This project is intended to be more open-ended than the typical CSCI 2270 course assignments. That being said, there are some implementation requirements listed below. Your project will first be graded from the user interface perspective, where you will be asked to present the required functionality. Once the functionality is verified, the grader will look at the implementation to confirm that the correct data structures are used in the C++ code.

2 User Interface

2.1 Interface

The user is to interact with the program via the command line. The program should run indefinitely until the user decides to quit. When the user launches `run_app_1`. The menu should look similar to the one shown below. If the user selects the help option, they can see the list of valid commands. Your objective is to implement the program for those commands. You may use any of the provided starter code in `main_1.cpp`.

```
Welcome to fileSim! Type ‘help‘ to view commands.
> help
```

```
Valid commands: quit, touch, edit, mkdir, ls, tree, rm, cd, mv, pwd, stat,
    cat, search
```

3 Implementation

3.1 Overview

The implementation needs to be in C++ and include classes defined as given in the starter code. Additional classes, structs, and functions can be added as needed. The `fileSim` class needs to contain methods to implement all the functionality listed below. The File System is to be represented using an N-ary tree, composed of file-system objects referred to as inodes. An N-ary tree is a hierarchical tree data structure, but unlike a Binary Search Tree it is not ordered, and a node can have any number of children. In our tree, each Inode is to have a single parent pointer, and only the root node's parent is NULL. Every Inode represents either a file or a directory (so the root Inode is a directory node).

The Inode is to be implemented as a C++ struct, as shown below. You may change it to suit your particular implementations, but the core functionality cannot be changed. Also, we show the struct definition for the Hash node, which is described for the search feature 13.

```
struct Inode {
    std::string name;
    std::string fileData;
    bool directory = false;
    int fileSize = 1; // assume every file is at least 1 byte in size
    std::time_t creationTime; // include the ctime library to use this type
    std::time_t modTime;
    Inode* parentNode;
    std::vector<Inode*> childNodes;
};

struct HashNode // hash node datastructure{
    std::string key;
    Inode *iptr;
    HashNode *next;
};
```

The C++ class definition should look like this. Again, all the methods related to the Hash Table are for the Efficient Search part of the project, as described for the search feature 13.

```
class FileSim {
  private:
    Inode *root;
    Inode *currentDir;
    int diskSize;
    int currentUsedDiskMemory;

    int makeNode(std::string name, bool isDirectory);
    Inode* pathToNode(std::string path);
    std::string nodeToPath(Inode *node);

    // Hash Table # of buckets (linked lists)
    int tableSize;
    // Pointer to an array containing buckets
    HashNode **table;
    // Create a new hash record
    HashNode *createNode(std::string key, Inode *_iptr, HashNode *next);
    // Hash table - insert record
    bool insertItem(std::string _key, Inode *_iptr);
    // hash function to map values to key
    unsigned int hashFunction(std::string _key);
    // Hash search
    HashNode *searchItem(std::string key);

  public:
    // FileSim Features; see writeup for specifics
    FileSim(int _diskSize);
    ~FileSim();

    int touch(std::string fileName);
    int mkdir(std::string dirName);
    void ls();
    void pwd();
    void tree();

    int cat(std::string path);
    int stat(std::string path);
    int edit(std::string path, std::string newValue);
    int cd(std::string dirPath);
    int rm(std::string path, bool recursive);
    int mv(std::string originPath, std::string destPath);
};
```

3.2 Initialization

When the program is launched, a prompt should greet the user and ask them for the size of the disk they wish to simulate. Then, a new instance of your tree gets instantiated, with a root node. The root node's parent pointer should be NULL and stay NULL, while the vector of children nodes should be empty.

3.3 Feature Requirements

Now let's talk about the functionality of the user features. Each of the features should be implemented as its own member function belonging to the `FileSim` class. Any operation that adds a new directory, file, or changes a file size needs to update the `currenntUsedDiskMemory` class member and compare it against the `diskSize`. If the disk size is exceeded, the given operation cannot be completed and an appropriate error message should be displayed.

1. **quit** - exit the menu loop and quit the program

2. **touch** - check to see if the new file does not cause the total data usage to exceed the disk capacity, in which case do not insert the new Inode and return an error message to the user. Check to see if a file by the same name already exists in the current directory, in which case also do not insert the new Inode and return an error message to the user. Otherwise, create a new Inode that is a file type (directory = false). Ask the user to enter the file name. Add an entry to the hash table (more on this below). Set the `fileSize` to one (assume that every file requires 1 byte of metadata). Update the `creationTime` member with the current time. Insert the Inode into the current directory.

3. **edit** - ask the user which file to edit. Check if a corresponding Inode exists in the current directory, and only allow editing if it is a file type (i.e. cannot edit a directory Inode type). Ask the user to enter a string to override the string in the corresponding `fileData` Inode member. The new file size is the length of the `fileData` string size plus one. Ensure that this does not cause the disk capacity to overflow, and update the corresponding `fileSize`.

4. **mkdir** - create a new Inode that is a directory type. Insert the Inode into the current directory. Since even a directory type has a size of 1 byte, the same check for exceeding the disk capacity should be performed as for the edit function.

5. **ls** - list the contents of the current directory, including all the files and subdirectories.

6. **tree** - starting with the current directory, print the entire descendant subtree of files and subdirectories. Make sure the print is legible.

7. **rm** - remove file or directory from the current directory. If the user chooses to delete an entire subdirectory, prompt them to ensure that is their intention.

8. **cd** - change the directory by traversing the tree structure and updating the `currentDir` pointer.
9. **mv** - to mimic the move command in Linux, this feature can be used to either change the name of an Inode or to change its location. It should take in two strings, which are either the name of the existing file and the new file name, or the absolute path of an existing file and the absolute path of the file's new location. (Note: by absolute path, we mean the path from the root directory to the given file.)
10. **pwd** - from the current directory, print the path to the root.
11. **stat** - ask the user for a file name in the current directory and print all the info from the file including the file size, file type, creation and modified times, and its parent directory.
12. **cat** - ask the user for a file name in the current directory and print the `fileData` contents.
13. **search** - prompt the user for a search input. The input can either be a file or a directory name. If one or more files with the name exist in the file system, print the path from the root to the given file(s).

    The search needs to utilize a Hash Table. The hash function implementation can be based on the basic division method, with open addressing, and linear probing as the collision resolution method. Make the initial table size small (m = 5). The Hash Table should allow for duplicate entries, as files with the same names can exist in multiple directories. You will need to go back to update your `touch` function such that a new hash record gets added every time a user creates a new file. Likewise, you will need to update `rm` and `mv` to update the Hash Table accordingly.

3.4 Search Optimization

For this project phase, create a new branch in GitHub, so that the core functionality from the previous phase will be preserved.

Now, your goal will be to observe the performance of the Hash Table while experimenting with different designs and parameters.

3.4.1 Collisions

A collision is when two keys hash to the same table index. In your hash table, add functionality to allow for counting the number of collisions.

3.4.2 Testing

Create a set of test cases in the test suite located in the `tests` directory of your repository. Start with a test that creates 100 directories, each containing one unique file. Then, you can run this test as you adjust the Hash Table configuration.

3.4.3 Hash Table Variation

Record the number of collisions from the base version of your Hash Table. Next, try the following configurations, running the same test every time.

- Open addressing, linear probing.
- Open addressing, quadratic probing.
- Chaining with a linked list.

Organize your implementation so that you can show the results from all the different configurations (i.e. make sure you do not lose a configuration as you go to implement a new one.)

4 Project Submission and Grading

4.1 Deliverables

For your project to be graded, it must written in C++ and compile with the provided Makefile in the CSEL Linux terminal. Please note that a project that does not compile will not be graded and will **result in a grade of 0.** Furthermore if the submitted program compiles but crashes while testing any of the features, the team will not receive any points for those features.

4.2 Teams, Grading, and Submission

You can work individually or in a group of up to two students. Either way, a single GitHub submission is to be made. You are encouraged to frequently push changes to your GitHub repo to prevent loss of work. The final submission should contain your source code files with the functioning class-based implementation of the `fileSim` program and the Hash Table files. You are provided with a set of test cases in the `test` directory. Your program needs to pass all of them. However, your overall grade will not be solely based on passing the test cases. A grading rubric will be released on Canvas later on. **The last commit you make to your GitHub repository before the deadline will be used for grading.**