Canned File System

Team Information

Sumatra Dhimoyee, U26967910, sumatrad

Emre Karabay, U73342696, karabay

Eric Xie, U15267667, ericx

Lisa Korver, U85983052, lkorver

Abstract

Our goal with this project was to implement in C++ a canned file storage system. The idea is that a user could upload files, which would then be separated into chunks, saved on a disk, and later retrieved. The primary data structure used is an N-node tree similar to Linux that follows the structure of the folders and files in the file system. We additionally use a hashmap to store chunk information for each file. We allow chunk sharing between multiple files to store data between files efficiently. Each leaf node of that tree represents a file and contains all of the metadata about that file. When a file is uploaded to the system, it is divided into 4KB chunks that are then stored separately on disk, and the address of these chunks and the total size of the file are stored in the tree node for that file. We have implemented functions to create, find, delete, update, list, and copy files, as well as more described in the next section. Our implementation is an example of how the functionality and user interface of the file system would work, but it is not currently connected to any lasting storage. We plan to extend this work to create a distributed file system with concurrency and persistent memory.

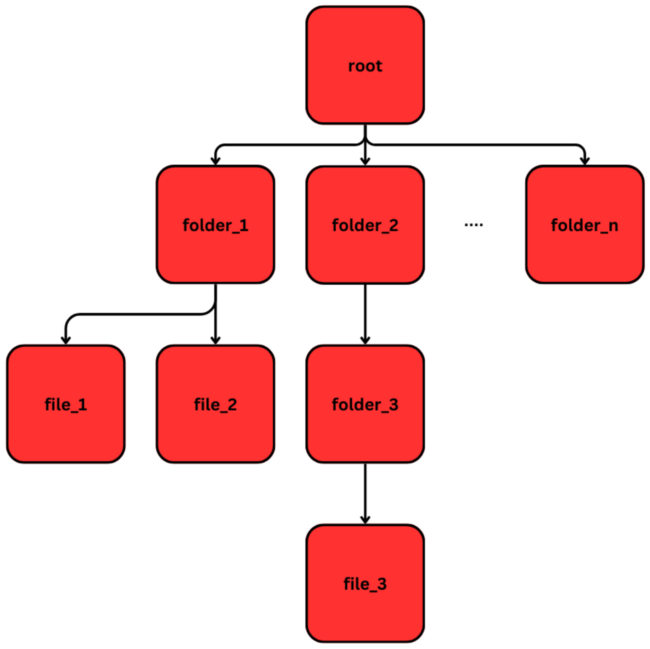


Figure 1: File Tree Diagram

Functions

searchByName: This function takes an input file name and searches the existing file tree using a depth-first search algorithm. It returns the path to the file from the root node. The find function is used in many of our other functions first to locate where the file is. The time complexity for this function is O(n) as the tree is not balanced, where n equals the number of nodes in the tree.

storeFile: This function loads a new file into the storage system from the directory that the program is running in. It takes as an input the name of the file being uploaded. It will create a new tree node to represent the file, store the respective metadata related to the file, divide the file into chunks, and store the chunk in a folder (which we plan to mount on a remote NVMe for faster storage). The time complexity for this function is O(n+c), where n equals the number of nodes in the tree and c equals the number of chunks.

splitFile: This function is used to split the file into 4KB chunks and copy those chunks into the disk. The inputs are the file's name and the tree node for that file, and it saves chunk-related metadata (chunk address and identifier) on the tree node so that they can be accessed later. The time complexity for this function is O(c), where c equals the number of chunks.

moveFile: This function moves a file from one folder to another. The inputs are the target file and the destination folder names. It searches the nodes for the file and folder and then updates the parent/children relationships within the tree. The time complexity for this function is O(n+c), where n equals the number of nodes in the tree and c equals the number of chunks.

deleteFile: This function takes the file name as input, finds that file in the tree using the name, and removes the tree node and related metadata. It checks the chunk hashmap to see if the chunks related to this file are shared with any other files. If the chunk is not shared, it will delete the chunk and free up the space, but if it is shared, the values of those chunks in the hashmap are decremented to indicate that this file no longer references those chunks. The time complexity for this function is O(n+c), where n equals the number of nodes in the tree and c equals the number of chunks.

copyFile: This function will create a copy of an existing file under a new name. It uses a Copy-on-Write mechanism, meaning only the file metadata is copied when the copy function is called the first time (e.g., the pointers to the chunks). However, the data is not copied until one of the files is modified. We update the chunk hash map to indicate that other files are also using those chunks. The time complexity for this function is O(n+c), where n equals the number of nodes in the tree and c equals the number of chunks.

getFile: This function takes the filename as the input and searches the tree to find the tree node corresponding to that specific file. It uses the chunk metadata to locate the chunks for those files, fetches them, and combines them using the mergeChunks function to recreate and return that file to the user. The time complexity for this function is O(n), where n equals the number of nodes in the tree and c equals the number of chunks.

mergeChunks: This function uses chunk information stored in TreeNode to fetch chunks corresponding to the file and reconstruct that file for the user. The time complexity for this function is O(c), where c equals the number of chunks.
updateFile: This function will overwrite an existing file. It takes as input the file name and will find that file in the tree and update or create new chunks, as well as update the metadata information of the file.

When a file is being modified, the function first must check if those chunks belong to only one file. If it belongs to more than one file, then at that point, the data will be copied to another set of chunks, and those chunks will be modified. The new chunk details will be stored in the tree node. The value here is that we save memory by not having unnecessary copies of identical data. The time complexity for this function is O(n+c), where n equals the number of nodes in the tree and c equals the number of chunks.

printTree: This function will print out all the files in the tree. It uses the search function (a depth-first search) and will print out the names of the files and folders as it traverses through each node. The time complexity for this function is O(n), where n equals the number of nodes in the tree.

writeTreeFile: This function writes the relevant information stored in the tree needed to reconstruct it including number of chunks, the map of chunk references, and TreeNode member variables. The time complexity for this function is O(n+c), where n equals the number of nodes in the tree and c equals the number of chunks.

readTreeFile: This function reads from a data file to reconstruct the tree. The time complexity for this function is O(n^2 + c), where n equals the number of nodes in the tree and c equals the number of chunks.

Instructions

Compile the NfileTree.cpp file using the provided makefile by simply navigating to the directory canned-file-share and typing in the terminal:

```
>> make
```

We have provided an example text file input.txt with a set of commands and three files with sample text to be uploaded. Run the resulting executable file by running the following in the terminal in the directory where NFileTree.cpp was compiled.

```
>>./NFileTree < input.txt
```

Interact with the file management system by writing commands in the virtual command line interface (CLI) followed by the necessary arguments. Running the executable on its own will print the following prompt:

```
>>./NFileTree
 Enter the command:
```

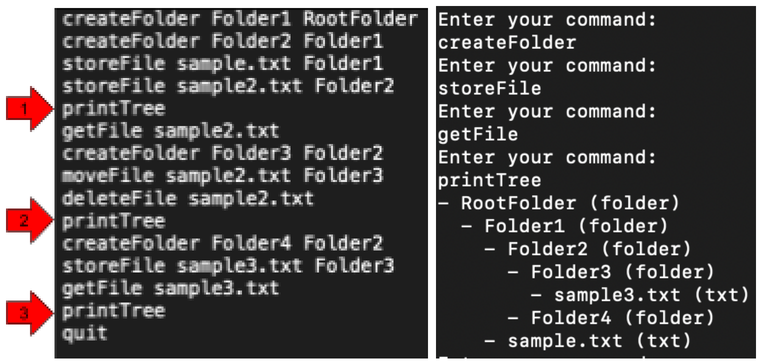
After which, use our functionalities following the format shown below.

```
>> moveFile [name of the file to be moved] [name of the folder to move the file to]
>> copyFile [name of the file to be copied] [name of the new file] [name of the folder to copy new file into]
>> deleteFile [name of the file to be deleted]
>> createFolder [new folder name] [location to create the folder in (name of folder)]
>> storeFile [file to be stored] [location to store the file in (name of folder)]
>> printTree
>> quit
```

Sample Results/Discussion

In the current implementation, after each function, we can check the state of the file system using the printTree function to show the current nodes in the tree, and the chunks of data can be found in the “store” folder. After each storeFile function, we can see an increase in the number of chunks stored in the store folder (the value can be calculated by dividing the file size by chunk size which is 4KB), and after deleteFile, a decrease in the number of chunks. We can see after we retrieve the file, the file content is the same along with the size of the original file.

The results of running the input.txt file command are shown in Figure 2.

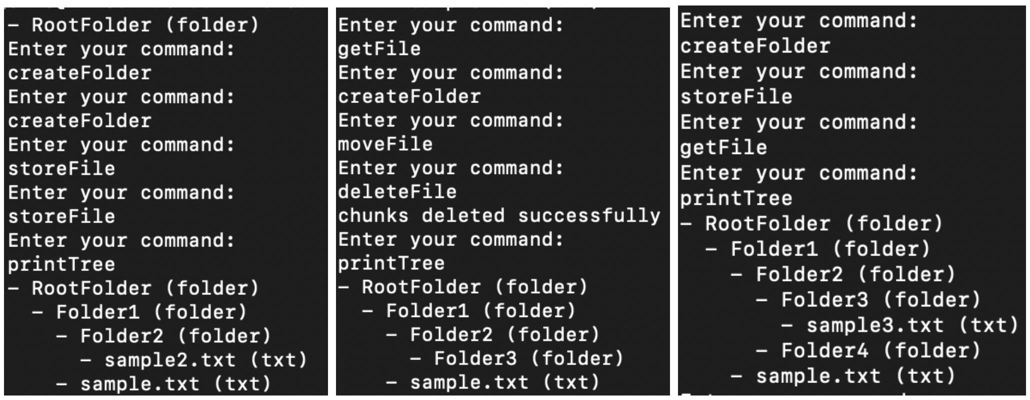


```
createFolder Folder1 RootFolder
createFolder Folder2 Folder1
storeFile sample.txt Folder1
storeFile sample2.txt Folder2
printTree
getFile sample2.txt
createFolder Folder3 Folder2
moveFile sample2.txt Folder3
deleteFile sample2.txt
printTree
createFolder Folder4 Folder2
storeFile sample3.txt Folder3
getFile sample3.txt
printTree
quit
```

```
Enter your command:
createFolder
Enter your command:
storeFile
Enter your command:
getFile
Enter your command:
printTree
- RootFolder (folder)
  - Folder1 (folder)
    - Folder2 (folder)
      - Folder3 (folder)
        - sample3.txt (txt)
      - Folder4 (folder)
    - sample.txt (txt)
```

Figure 2: List of commands included in input.txt (left) and resulting tree (right)

Taking a closer look at the steps, Figure 3 shows the original tree, and the resulting printTree output after some set of commands are completed.



```
- RootFolder (folder)
Enter your command:
createFolder
Enter your command:
createFolder
Enter your command:
storeFile
Enter your command:
storeFile
Enter your command:
printTree
- RootFolder (folder)
  - Folder1 (folder)
    - Folder2 (folder)
      - sample2.txt (txt)
    - sample.txt (txt)
```

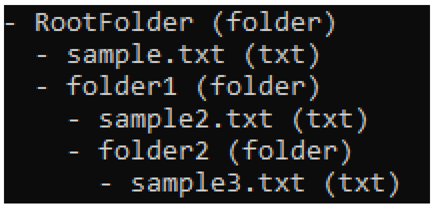
```
Enter your command:
getFile
Enter your command:
createFolder
Enter your command:
moveFile
Enter your command:
deleteFile
chunks deleted successfully
Enter your command:
printTree
- RootFolder (folder)
  - Folder1 (folder)
    - Folder2 (folder)
      - Folder3 (folder)
    - sample.txt (txt)
```

```
Enter your command:
createFolder
Enter your command:
storeFile
Enter your command:
getFile
Enter your command:
printTree
- RootFolder (folder)
  - Folder1 (folder)
    - Folder2 (folder)
      - Folder3 (folder)
        - sample3.txt (txt)
      - Folder4 (folder)
    - sample.txt (txt)
```

Figure 3 left: printTree (red arrow 1 in figure 2) after first 4 commands are run
Figure 3 center: printTree (red arrow 2 in figure 2) after next 4 commands are run
Figure 3 right: printTree (red arrow 3 in figure 2) after last 3 commands are run

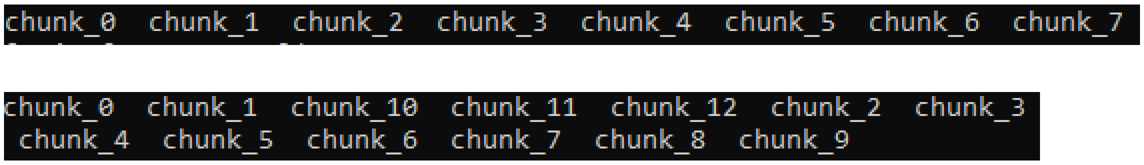
Below we have given a few examples of the commands and how the file system appears before and after each command is executed:

Command (storeFile sample3.txt in folder2):



```
- RootFolder (folder)
  - sample.txt (txt)
  - folder1 (folder)
    - sample2.txt (txt)
    - folder2 (folder)
      - sample3.txt (txt)
```

Figure4 left: printTree before the storeFile command
Figure4 right: printTree after the storeFile command

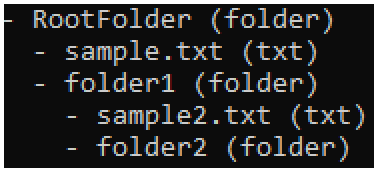


```
chunk_0  chunk_1  chunk_2  chunk_3  chunk_4  chunk_5  chunk_6  chunk_7
```

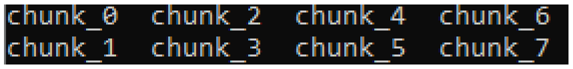
```
chunk_0  chunk_1  chunk_10  chunk_11  chunk_12  chunk_2  chunk_3
 chunk_4  chunk_5  chunk_6  chunk_7  chunk_8  chunk_9
```

Figure 5 top: chunks directory before storeFile command
Figure 5 bottom: chunks directory after storeFile command

Command (deleteFile sample3.txt):



```
- RootFolder (folder)
  - sample.txt (txt)
  - folder1 (folder)
    - sample2.txt (txt)
    - folder2 (folder)
```

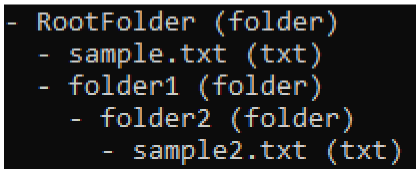


```
chunk_0  chunk_2  chunk_4  chunk_6
chunk_1  chunk_3  chunk_5  chunk_7
```

Figure 6: printTree (left) and chunks directory (right) after deleting sample3.txt

---

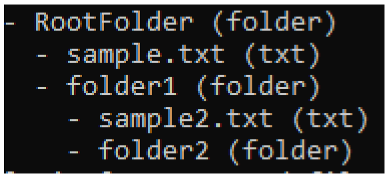
Command (moveFile sample3.txt into folder2):



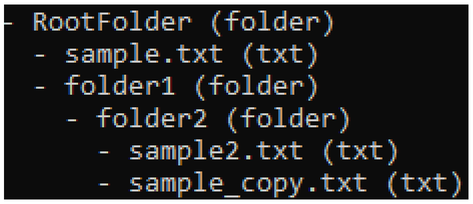
```
- RootFolder (folder)
  - sample.txt (txt)
  - folder1 (folder)
    - folder2 (folder)
      - sample2.txt (txt)
```

Figure 7: printTree after moving sample3.txt from folder1 to folder2

Command (copyFile sample.txt sample_copy.txt folder2):



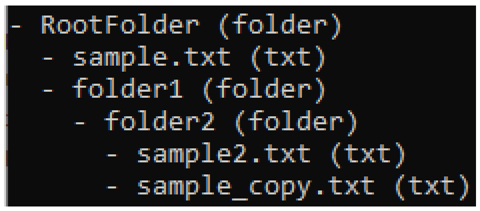
```
- RootFolder (folder)
  - sample.txt (txt)
  - folder1 (folder)
    - sample2.txt (txt)
    - folder2 (folder)
```



```
- RootFolder (folder)
  - sample.txt (txt)
  - folder1 (folder)
    - folder2 (folder)
      - sample2.txt (txt)
      - sample_copy.txt (txt)
```

Figure 8 left: printTree before copying sample.txt into folder2
Figure 8 right: printTree after copying sample.txt into folder2

Command (readTreeFile):

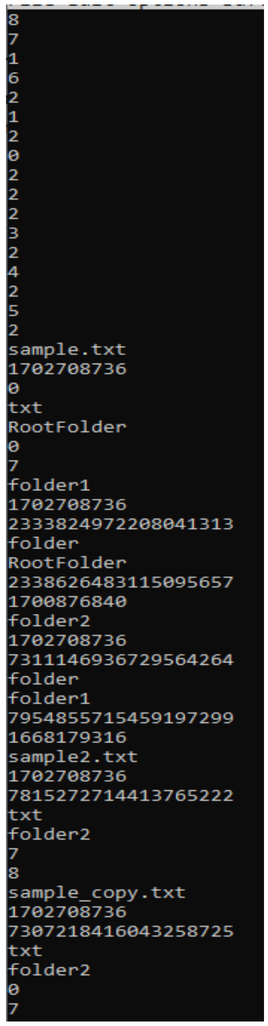


```
- RootFolder (folder)
  - sample.txt (txt)
  - folder1 (folder)
    - folder2 (folder)
      - sample2.txt (txt)
      - sample_copy.txt (txt)
```

Figure 9: fileTree after reading the data file to reconstruct tree

---

Command (writeTreeFile):



```
8
7
1
6
2
1
2
0
2
2
2
3
2
4
2
5
2
sample.txt
1702708736
0
txt
RootFolder
0
7
folder1
1702708736
2333824972208041313
folder
RootFolder
2338626483115095657
1700876840
folder2
1702708736
7311146936729564264
folder
folder1
7954855715459197299
1668179316
sample2.txt
1702708736
7815272714413765222
txt
folder2
7
8
sample_copy.txt
1702708736
7307218416043258725
txt
folder2
0
7
```

Figure 10: Custom data file storing FileTree information

References

[1] “Example of a File System Server in C++ - Ice,” *Ice Architecture*. https://doc.zeroc.com/ice/3.6/language-mappings/c++-mapping/server-side-slice-to-c++-mapping/example-of-a-file-system-server-in-c++.

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, Operating systems : three easy pieces. Madison: Arpaci-Dusseau Books, 2018.