OS Lab # 6
File System Implementation

Submitted by:
Muhammad Waleed Abdullah (348883)
Muhammad Taimoor Zaeem (337255)

System design:

The system has been implemented in python; the data has been made persistent using the pickle module. I went with an object-oriented approach to implement the filesystem.

Data structures:

The main class is the FileSystem class which communicates with DirectoryNode and FileNode classes. The main file only communicates with the FileSystem object which allows for great abstraction and ease. Some screenshots have been provided below to understand the attributes.

I have also maintained the blocks as a data structure which has two attributes {usedSize: -, data: ‘’ }, the block list contains nodes based on this dictionary which stored the data. The nodes stored in the directories children attribute are also stored in an additional data structure in the form {nodeName: --, nodeType, node: ‘’ }, this is to maintain some metadata about the node so that it can be used in different operations.

The nodes in the dir.children attribute are stored as follows:

```
def createFile(self, fileName):
    if self.checkIfNameExists(fileName):
        return "FileName already exists"
    self.children.append(
        {'nodeName': fileName, 'node': FileNode
        (fileName), 'nodeType': FILE})
    return "Added File"

def createDirectory(self, dirName):
    if self.checkIfNameExists(dirName):
        return "Directory already exists"
    self.children.append(
        {'nodeName': dirName, 'node': DirectoryNode
        (dirName), 'nodeType': DIR})
    self.children[-1]['node'].parent = self
    return 'Added directory'
```

```
TOTAL_BLOCKS = TOTAL_MEM // BLOCK_SIZE


class FileSystem:

    def __init__(self):
        self.root = DirectoryNode('root')
        self.currentDir = self.root
        self.blocks = [{'usedSize': 0, 'data': ''}
                       for _ in range(TOTAL_BLOCKS)]
        # 1 means the block is occupied and 0 means
        that it is free
        self.freeBlocks = [0] * TOTAL_BLOCKS
        self.numFreeBlocks = TOTAL_BLOCKS
        # idx 0 indicates fileNode, idx 1 indicates
        the mode
        self.currentFile = (None, None)
```

```
class FileNode:
    def __init__(self, name):
        self.name = name
        self.dataPointers = []  # holds indices to the data blocks assigned to the file

```

```
class DirectoryNode:

    def __init__(self, name):
        self.name = name
        self.children = []
        self.parent = None
```

Directory Structure:

The directory structure of the system is in a tree format, there is one root node, and that node can have n children and the n children can have further n children and so on. The children can be fileNodes or directoryNodes. There cannot be 2 children with the same name in one level of the tree. The tree is acyclic so an infinite loop problem will not occur in the system.

Block system:

I have implemented block system to store data in files. The fileNode only holds the filename and the pointers to the blocks where its data is stored. This means even If we move the file from one location to another, we won't have to move the data, only the fileNode.

The block allocation is indexed, and a freeblockList is maintained to check which blocks are free, whenever we enter data into a file the freeBlockList is traversed to find a freeBlock until all the data has been written to the blocks. If enough blocks are not available for the data, then the data is not written to the file, this is checked without having to traverse the freeBlockList, because an attribute called numFreeBlocks has been maintained which is incremented / decremented when a block is allocated / deallocated. When we delete a fileNode in the system we also deallocate its blocks so that they can be used for another file.

Operations:

All the methods that were required to be implemented in the lab document have been implemented, an interactive terminal has also been created to show the user which file is open and in which directory the user is currently in.

Limitations:

Currently, we can only work with relative paths within the current directory in the system, the ability to use relative paths which extend to outside the directory and absolute paths can be implemented pretty easily, by using the split('/') method in python and defining a method in the FileSystem class which gets the destination node, but I am not implementing it right now due to time constraints.

Final remarks:

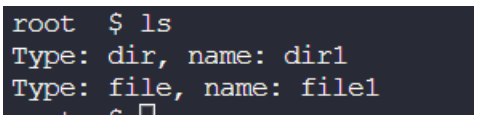
Other than that the system is fully functional we can create, delete, edit, read , move, truncate files. We can also move data within the file. This was done using clever string slicing and allocation according to the block size. The block size and the memory size can also be changed, which means this fileSystem class can be used with memory of any size

After the user shuts the system down the data is written to a .dat file which loads up when the user opens the program again.

User guide:

The user can enter multiple commands to interact with the terminal the terminal also shows the current directory and the current open file and the mode. If any is open, the user can enter shutdown to exit the terminal.

'ls': the ls command can be used to display all the files and folders in the current dir



'cd': the user can use the cd command to move between directories and move up to the parent, currently you cannot provide path to move, only the directories in the current dirs. Children can be moved to using cd, cd .. can be used to move back to the parent

'mkdir': mkdir dirName can be used to create a direcetory

‘create’: create filename can be used to create a file

‘delete’: delete can be used to delete a dir or a file

‘open’: open can be used to open a file, open file mode, default mode is r

‘close’: close filename, can be used to close an open file

‘write_to_file’: write_to_file writeAt text

‘read_from_file’: read_from_file start size, it returns the data in the file, start and size are optional

‘move_within_file’: move_within_file start, size, target, all parameters are required

‘truncate_file’: truncates the current open file

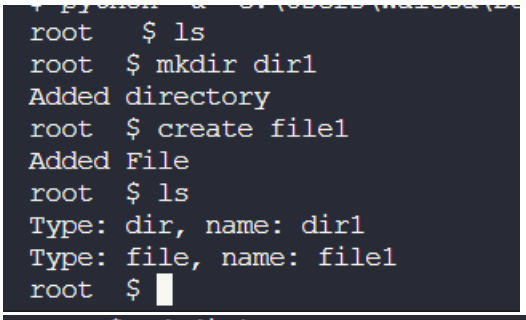
‘show_map’: shows the memory map of the current state of the filesystem

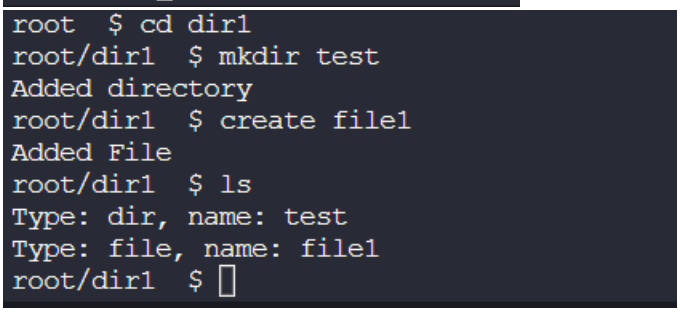
‘move’: moves a file from the current dir to some dest dir which is also in the current dir

‘shutdown’: shutsdown the system

Output Screenshots and usage of above commands:

Ls:





Cd:

```
root/dir1  $ cd ..
root  $ open file1 rw
File opened
root , currFile: file1 mode: rw $
```

Open, write_to_file, move_within_file, truncate_file, close:

```
root/dir1  $ cd ..
root  $ open file1 rw
File opened
root , currFile: file1 mode: rw $ write_to_file Hello world this is a
test to submit the document
successfuly written
root , currFile: file1 mode: rw $ read_from_file
Hello world this is a test to submit the document
root , currFile: file1 mode: rw $
```

```
root , currFile: file1 mode: rw $ move_within_file 0 3 10
Moved Data
root , currFile: file1 mode: rw $ read_from_file
lo worlHeld this is a test to submit the document
root , currFile: file1 mode: rw $
```

```
root , currFile: file1 mode: rw $ truncate_file
truncated File
root , currFile: file1 mode: rw $ read_from_file
**starting location outside file**
root , currFile: file1 mode: rw $
```

It says starting location outside file because there is no data in the file

```
root , currFile: file1 mode: rw $ close file1
Closed File
root  $
```

Show_map:

```
root   $ show_map
 root
  dir->dir1
   dir->test
   file->file1
  file->file1
root  $
```

```
root/dir1/test  $ show_map
 root
  dir->dir1
   dir->test
    dir->helloDir
   file->file1
  file->file1
  dir->dir2
   dir->dir23
   file->file4
root/dir1/test  $
```

Move:

```
Type: dir, name: dir23
Type: file, name: file4
Type: file, name: file1
root/dir2  $ move file1 root
Moved
root/dir2  $ ls
Type: dir, name: dir23
Type: file, name: file4
root/dir2  $ cd ..
```

```
Type: dir, name: dir1
Type: dir, name: dir2
Type: file, name: file1
root  $ move file1 dir2
Moved
root  $ cd dir2
root/dir2  $ ls
Type: dir, name: dir23
Type: file, name: file4
Type: file, name: file1
root/dir2  $
```

References:

The only references were the requirements in the lab documents and the class lecture slides to check different ways to implement blocks.