FILE MANAGEMENT SYSTEM SIMULATION

A PROJECT BY

Damian Rozycki (Ideas, System Design, Implementation)

Asmere Duressa (Ideas, Testing, User Guide)

Anh Phan (Ideas, Writing Report, Analysis)

Kevin Jones (Ideas, Testing, User Guide)

Fall 2024

Towson University

Professor Ramesh K. Karne

COSC 439.001 - OPERATING SYSTEMS

1. Introduction

File management is a crucial aspect of operating systems, responsible for organizing and storing data efficiently. Understanding file system operations and their implementation is essential for system developers and administrators. In this project, we implement a simulation of a file management system to explore various file system functionalities and their interactions.

This paper presents the design, implementation, and simulation of a file management system (FMS) as part of an operating system course project. The FMS is developed in Java and simulates essential functionalities of a file system, including directory management, file creation, deletion, copying, moving, and dynamic resizing. The system also incorporates disk management, allocation, and retrieval of file metadata. The project aims to provide a hands-on understanding of file system operations and their underlying mechanisms.

2. System Design

Our file management system comprises several key components:

2.1 Block

The `**Block**` class represents the basic unit of data storage, with a fixed size of 1024 bytes. It encapsulates data manipulation methods and ensures data integrity through clear boundaries.

List of methods in the `**Block**` class:

1. Constructor: Block()
2. getData()
3. setData(byte[] data)
4. getBlockSize()
5. clearData()

2.2 SuperBlock

The `**SuperBlock**` class represents the superblock of the file system, containing metadata about the file system such as the total number of blocks, free blocks, and block size. It provides methods for initializing the file system and updating disk usage information.

List of methods in the `**SuperBlock**` class:

1. Constructor: SuperBlock()
2. getTotalBlocks()
3. allocateBlock()
4. freeBlock(int blockNumber)
5. findInode(String fileName, String directoryName)
6. listInodes(String directoryName)
7. allocateInode(String fileName, int fileSizeInKB)
8. releaseInode(String fileName, String directoryName)
9. allocateBlocks(int requiredBlocks)
10. freeBlocks(int startingBlock, int numberOfBlocks)
11. getInodes()
12. countFreeBlocks()
13. countUsedInodes()

14. listFileSystem()
15. moveInode(Inode inode, String newFileName, String newDirectoryName)

2.3 Directory

The `**Directory**` class manages directories within the file system, facilitating operations such as creation, deletion, and navigation. It also supports file management within directories, including creation, deletion, and copying.

List of methods in the `**Directory**` class:

1. Constructor: Directory(String name, SuperBlock superBlock, Directory parent)
2. getSuperBlock()
3. getSubDirectories()
4. getName()
5. createDirectory(String dirName)
6. deleteDirectory(String dirName)
7. deleteAllContents()
8. deleteAllFiles()
9. moveFile(String sourceFileName, String destPath, Directory currentDirectory, Disk disk)
10. findDirectory(String path, Directory rootDirectory, Directory currentDirectory)
11. createFile(String fileName, int size)
12. deleteFile(String fileName)
13. listContents()
14. copyFile(String sourceFileName, String destPath, Disk disk)
15. generateUniqueFileName(String baseName, Directory destDir)
16. copyFileToDirectory(Inode sourceInode, String newFileName, Directory destDir, Disk disk)
17. getParentDirectory()

2.4 Disk

The `**Disk**` class simulates the disk storage medium and manages block allocation. It provides methods for reading, writing, and deleting files, as well as retrieving disk usage information.

List of methods in the `**Disk**` class:

1. Constructor: Disk()
2. allocateBlocks(int size)
3. markBlocksUsed(int start, int size, boolean used)
4. areBlocksFree(int start, int size)
5. freeBlocks(int start, int size)
6. getSuperBlock()
7. writeFile(String command)
8. deleteFile(String fileName)
9. readFile(String fileName)
10. listFiles()
11. printFileInfo(String fileName, Directory currentDirectory)
12. displayDiskInfo()
13. getRootDirectory()
14. listFileSystem()

2.5 Inode

The `**Inode**` class stores metadata for each file, including file name, size, block allocation, and last modified time. It facilitates file creation, deletion, and modification operations.

List of methods in the `**Inode**` class:

1. Constructor: Inode()
2. allocateBlocks(int startingBlock, int blocksAllocated)
3. clearBlockAllocation()
4. getName()
5. setName(String name)
6. getSize()
7. setSize(int size)
8. getStartingBlock()
9. getBlocksAllocated()
10. getLastModifiedTime()
11. isUsed()
12. setUsed(boolean used)
13. getDirectoryName()
14. setDirectoryName(String directoryName)

2.6 FileSystemCLI

The `**FileSystemCLI**` class implements a command-line interface for interacting with the file management system. It allows users to execute commands to perform various file and directory operations.

List of methods in the `**FileSystemCLI**` class:

1. Constructor: FileSystemCLI()
2. start()
3. processInput(String inputLine)
4. changeDirectory(String directoryName)
5. clearScreen()
6. printWorkingDirectory()
7. printHelp()
8. main(String[] args)

2.7 ScriptRunner

The `**ScriptRunner**` class facilitates the execution of commands stored in a script file within the file system simulator. It allows for batch processing of file system operations by reading each command from the file, ignoring comments, and passing them to the `**FileSystemCLI**` object for execution.

List of methods in the `**ScriptRunner**` class:

1. ScriptRunner(FileSystemCLI fileSystemCLI)
2. runScriptFromFile()
3. runScript(String filePath)

3. Implementation

3.1 Functionality

1. **Directory Management:**
- Creating, deleting, and listing directories.
- Use system calls or library functions to interact with the underlying file system to perform these operations.

2. **File Management:**
- Creating, deleting, copying, moving, and resizing files.
- Handle file operations using appropriate system calls or file manipulation functions provided by the programming language or operating system.

3. **Disk Management**:
- Display disk information, including total space, used space, and free space.
- Calculate disk space usage based on the allocation of blocks to files.

4. **Block Management**:
- We manage blocks using a bitmap or a simple Boolean array to indicate whether each block is free or used.
- This will help in efficiently finding the first available block(s) during file creation.

5. **Contiguous Block Allocation**:
- When a file is created, it's allocated blocks contiguously from the first available free block.
- This ensures that the file's data is stored contiguously, simplifying access, and reducing fragmentation.

6. **Updating on File Deletion**:
- Upon deleting a file, the blocks it occupied will be marked as free.
- For contiguous allocation, we won't immediately shift other files' blocks to fill the gap; however, future files will use the earliest available blocks, ensuring space is used efficiently over time.

7. **Displaying Block Usage**:
- To display the blocks used by a file, we simply list the contiguous of blocks it occupies.
- For a visual representation of the entire disk's block usage, we iterate through the block management structure and display the status of each block.

3.2 List of available commands

1. **ls** - List directory contents
2. **cd <dirName>** - Change directory
3. **pwd** - Print working directory
4. **mkfile <fileName> <sizeInKB>** - Create a new file
5. **rmfile <fileName>** - Remove a file
6. **mkdir <dirName>** - Create a new directory
7. **rmdir <dirName>** - Remove a directory and its contents
8. **cpfile <fileName> <destinationDiPathr>** - Copy a file to a specified destination.
9. **mvfile <fileName> <destinationDirPath>** - Move a file to a specified destination.
10. **diskinfo** - Display disk information
11. **fileinfo <fileName>** - Display information about a file
12. **showsystem** - Show the file system's block allocation as an array
13. **writefile -a <fileName> <sizeChangeInKB>** - Append to a file
14. **writefile -r <fileName> <sizeChangeInKB>** - Reduce a file size
15. **readfile <fileName>** - Read a file

16. **runscript** - Run a script file with a list of commands
17. **clear** - Clear the screen
18. **exit** - Exit the simulator

4. Analysis

4.1 Delays and Limits

FUNCTIONALITY	DELAY	LIMIT
Allocation and Deallocation of Blocks	The process of allocating and deallocating blocks in the disk introduces delays when the disk is nearly full and needs to search for contiguous free blocks.	The total number of blocks available on the disk (128 blocks) imposes a limit on the maximum size of files that can be stored.
File Creation and Deletion	Creating and deleting files involve operations like allocating and deallocating inodes and blocks, resulting in delays, especially in larger file systems.	The maximum number of files (16 files) that can be created in the file system imposes a limit on the scalability of the system.
File Modification (Append/Reduce)	Modifying file sizes involves allocating or deallocating blocks so it could introduce delays if the file needs to be relocated to accommodate the new size.	The maximum file size of 8 KB imposes a limit on the size to which a file can be extended.
Directory Operations (Creation, Deletion, Navigation)	Directory operations might introduce delays, especially when navigating through deep directory structures or when directories contain a large number of files.	The maximum depth of the directory structure and the maximum number of files per directory are implicit limits that might affect performance and organization.

4.2 Block Allocation - Contiguous vs Linked vs Indexed

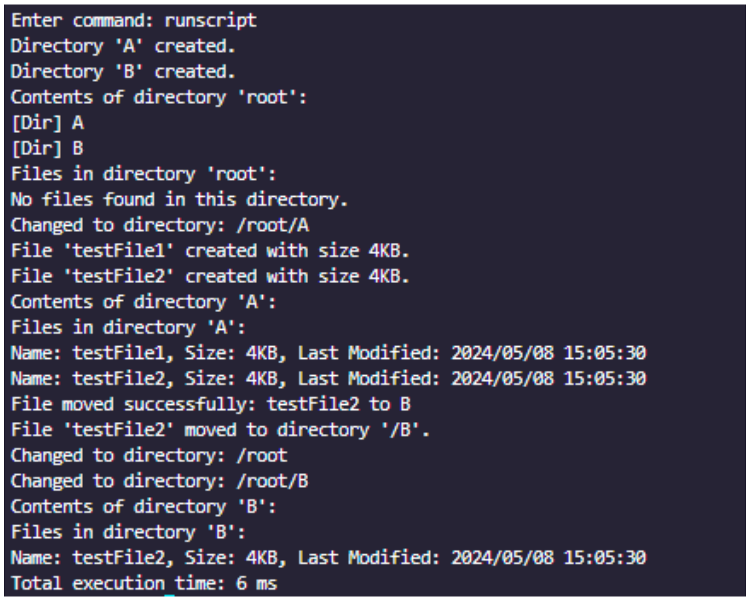
In our simulator, we use **Contiguous Allocation**, where each file occupies a contiguous block of disk space. This method simplifies file access by storing data blocks sequentially, facilitating efficient reads and writes. However, it can lead to external fragmentation over time, impacting space utilization. On the other hand, **Linked Allocation** represents files as linked lists of disk blocks, offering flexibility for variable-sized files. But accessing files may be slower due to multiple pointers, and disk locality can suffer. Finally, **Indexed Allocation** is another different method which assigns each file an index block for direct access to any block, ensuring efficient random access. However, it requires additional space for the index block and has scalability limitations. Each method has trade-offs in access efficiency, space utilization, and complexity, influenced by system requirements and storage characteristics.

ASPECT	CONTIGUOUS	LINKED	INDEXED
Description	Assigns each file contiguous blocks on the disk.	Represents files as linked lists of disk blocks.	Utilizes an index block containing pointers to file blocks.
File Access Efficiency	Efficient sequential access due to contiguous blocks.	May require following multiple pointers, leading to longer access times.	Efficient random access through the index block.

Fragmentation	External fragmentation may occur over time.	No external fragmentation as each block is only used for a single file.	No external fragmentation as each block is only used for a single file.
Space Utilization	May lead to poor space utilization due to wasted space when files are not perfectly sized.	Flexible space utilization as files can grow dynamically without contiguous blocks.	Requires additional space for the index block, impacting overall space utilization.
Disk Locality	N/A	Poor disk locality may occur due to scattered block locations.	N/A
Complexity	Relatively simple implementation.	Requires additional bookkeeping for maintaining pointers.	Requires additional space and complexity for the index block.
Scalability	Limited scalability due to potential fragmentation and wasted space.	Limited scalability due to potential scattered block locations.	Limited scalability due to the size of the index block.

5. Measurements

Our `**ScriptRunner**` class enables the execution of commands stored in a script file within the file system simulator. Additionally, it provides functionality to measure and display the total execution time of the script. Particularly, the method **runScript(String filePath)** reads each line from the script file, ignoring lines starting with '#'. It then processes each non-comment line using the `**FileSystemCLI**` object. After execution, it calculates and prints the total execution time in milliseconds.

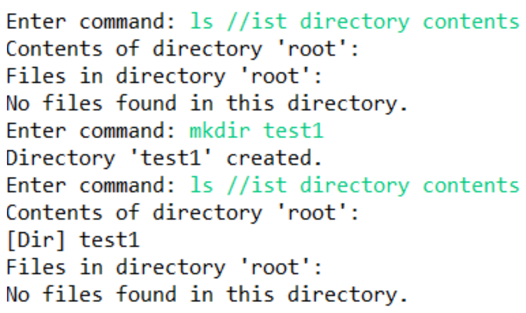


```
Enter command: runscript
Directory 'A' created.
Directory 'B' created.
Contents of directory 'root':
[Dir] A
[Dir] B
Files in directory 'root':
No files found in this directory.
Changed to directory: /root/A
File 'testFile1' created with size 4KB.
File 'testFile2' created with size 4KB.
Contents of directory 'A':
Files in directory 'A':
Name: testFile1, Size: 4KB, Last Modified: 2024/05/08 15:05:30
Name: testFile2, Size: 4KB, Last Modified: 2024/05/08 15:05:30
File moved successfully: testFile2 to B
File 'testFile2' moved to directory '/B'.
Changed to directory: /root
Changed to directory: /root/B
Contents of directory 'B':
Files in directory 'B':
Name: testFile2, Size: 4KB, Last Modified: 2024/05/08 15:05:30
Total execution time: 6 ms
```

6. User Guide

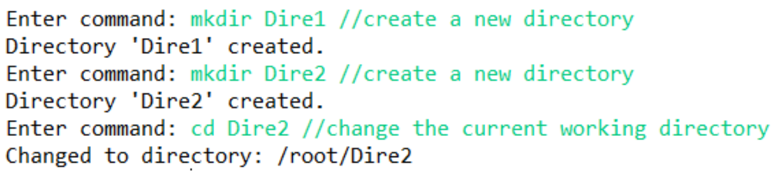
We simulate the file management system by executing a series of commands through the command-line interface. Each command corresponds to a specific file or directory operation.

1. **ls** - List directory contents



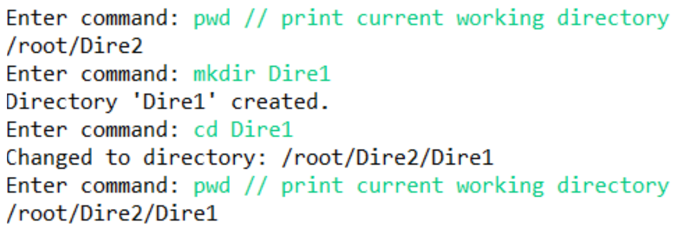
```
Enter command: ls //ist directory contents
Contents of directory 'root':
Files in directory 'root':
No files found in this directory.
Enter command: mkdir test1
Directory 'test1' created.
Enter command: ls //ist directory contents
Contents of directory 'root':
[Dir] test1
Files in directory 'root':
No files found in this directory.
```

2. **cd <dirName>** - Change directory



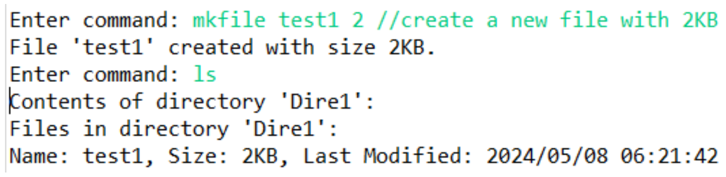
```
Enter command: mkdir Dire1 //create a new directory
Directory 'Dire1' created.
Enter command: mkdir Dire2 //create a new directory
Directory 'Dire2' created.
Enter command: cd Dire2 //change the current working directory
Changed to directory: /root/Dire2
```

3. **pwd** - Print working directory



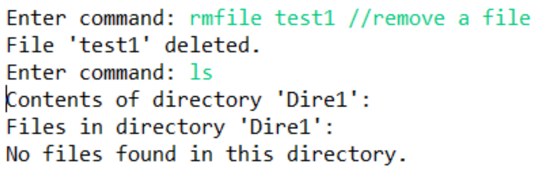
```
Enter command: pwd // print current working directory
/root/Dire2
Enter command: mkdir Dire1
Directory 'Dire1' created.
Enter command: cd Dire1
Changed to directory: /root/Dire2/Dire1
Enter command: pwd // print current working directory
/root/Dire2/Dire1
```

4. **mkfile <fileName> <sizeInKB>** - Create a new file



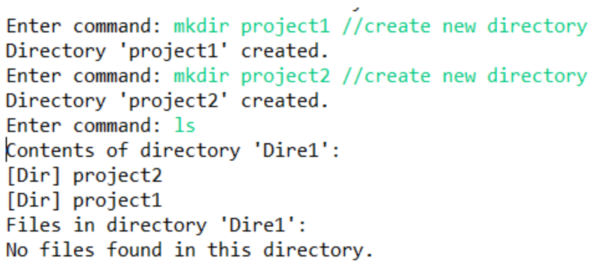
```
Enter command: mkfile test1 2 //create a new file with 2KB
File 'test1' created with size 2KB.
Enter command: ls
Contents of directory 'Dire1':
Files in directory 'Dire1':
Name: test1, Size: 2KB, Last Modified: 2024/05/08 06:21:42
```

5. **rmfile <fileName>** - Remove a file



```
Enter command: rmfile test1 //remove a file
File 'test1' deleted.
Enter command: ls
Contents of directory 'Dire1':
Files in directory 'Dire1':
No files found in this directory.
```

6. **mkdir <dirName>** - Create a new directory



```
Enter command: mkdir project1 //create new directory
Directory 'project1' created.
Enter command: mkdir project2 //create new directory
Directory 'project2' created.
Enter command: ls
Contents of directory 'Dire1':
[Dir] project2
[Dir] project1
Files in directory 'Dire1':
No files found in this directory.
```

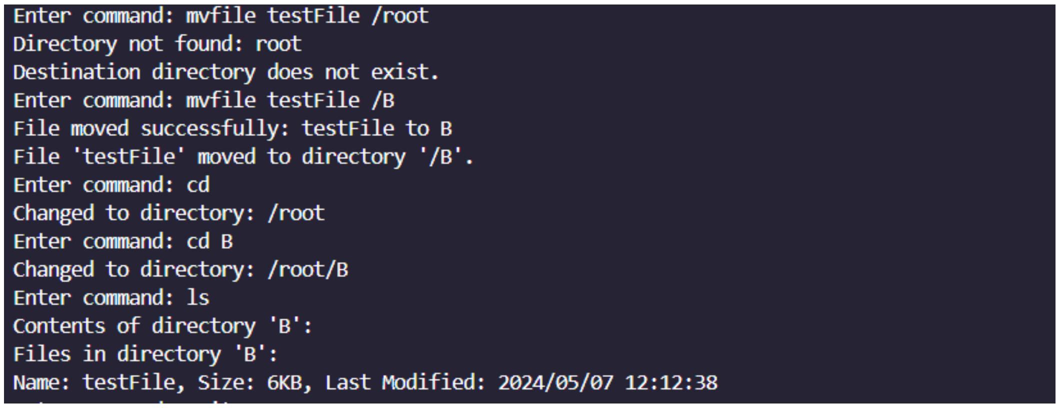
7. **rmdir <dirName>** - Remove a directory and its contents

```
Enter command: mkdir project1 /create new directory
Error: Directory 'project1' already exists.
Enter command: mkdir project2 /create new directory
Directory 'project2' created.
Enter command: rmdir project2 //Remove a directory and its contents
Directory 'project2' deleted.
Enter command: rmdir project1 //Remove a directory and its contents
Directory 'project1' deleted.
```

8. **cpfile <fileName> <destinationDirPath>** - Copy a file to a specified destination.

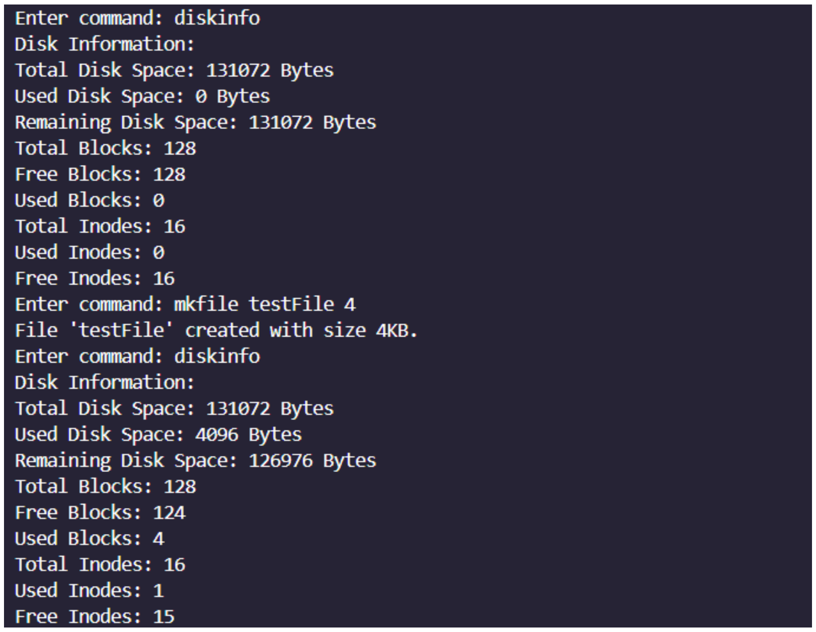
```
Enter command: mkfile test1 2
File 'test1' created with size 2KB.
Enter command: mkfile test2 4 //create a new file with 4KB
File 'test2' created with size 4KB.
Enter command: cpfile test2 /Dire1/Dire2 //copy the file in to another directory
File 'test2(1)' copied to 'Dire2'.
```

9. **mvfile <fileName> <destinationDirPath>** - Move a file to a specified destination.



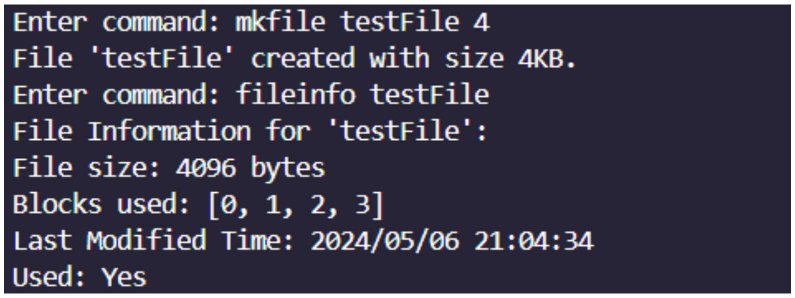
```
Enter command: mvfile testFile /root
Directory not found: root
Destination directory does not exist.
Enter command: mvfile testFile /B
File moved successfully: testFile to B
File 'testFile' moved to directory '/B'.
Enter command: cd
Changed to directory: /root
Enter command: cd B
Changed to directory: /root/B
Enter command: ls
Contents of directory 'B':
Files in directory 'B':
Name: testFile, Size: 6KB, Last Modified: 2024/05/07 12:12:38
```

10. **diskinfo** - Display disk information



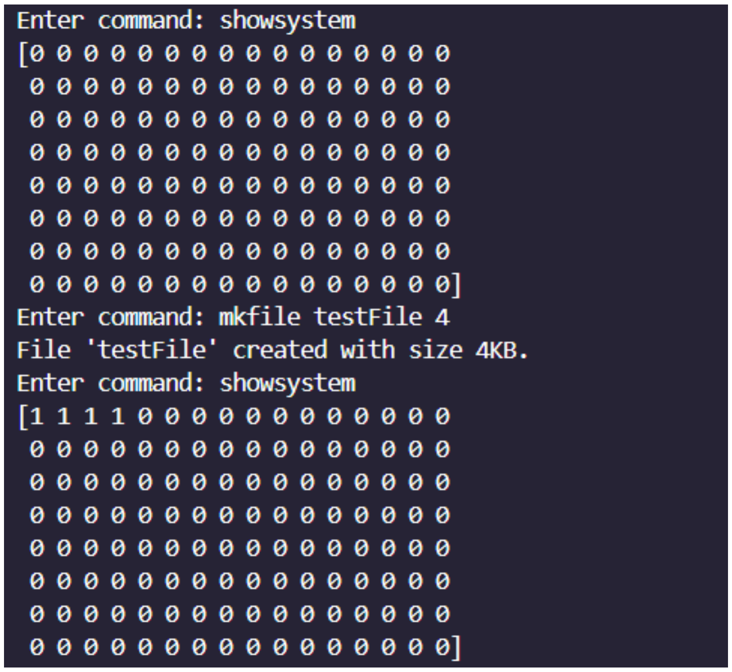
```
Enter command: diskinfo
Disk Information:
Total Disk Space: 131072 Bytes
Used Disk Space: 0 Bytes
Remaining Disk Space: 131072 Bytes
Total Blocks: 128
Free Blocks: 128
Used Blocks: 0
Total Inodes: 16
Used Inodes: 0
Free Inodes: 16
Enter command: mkfile testFile 4
File 'testFile' created with size 4KB.
Enter command: diskinfo
Disk Information:
Total Disk Space: 131072 Bytes
Used Disk Space: 4096 Bytes
Remaining Disk Space: 126976 Bytes
Total Blocks: 128
Free Blocks: 124
Used Blocks: 4
Total Inodes: 16
Used Inodes: 1
Free Inodes: 15
```

11. **fileinfo <fileName>** - Display information about a file



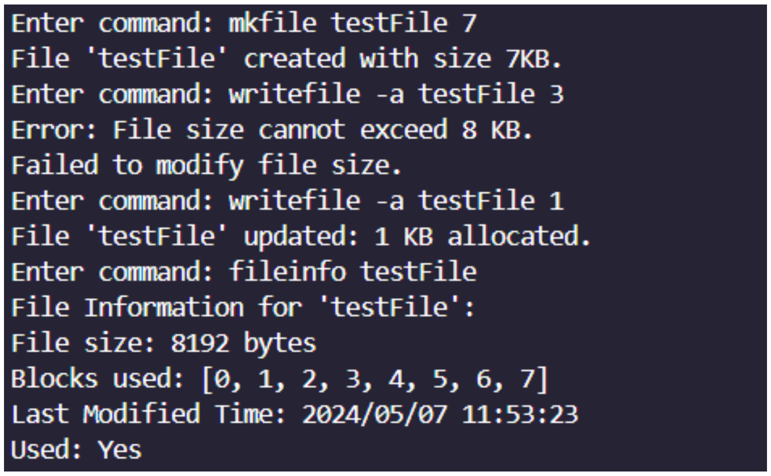
```
Enter command: mkfile testFile 4
File 'testFile' created with size 4KB.
Enter command: fileinfo testFile
File Information for 'testFile':
File size: 4096 bytes
Blocks used: [0, 1, 2, 3]
Last Modified Time: 2024/05/06 21:04:34
Used: Yes
```

12. **showsystem** - Show the file system's block allocation as an array



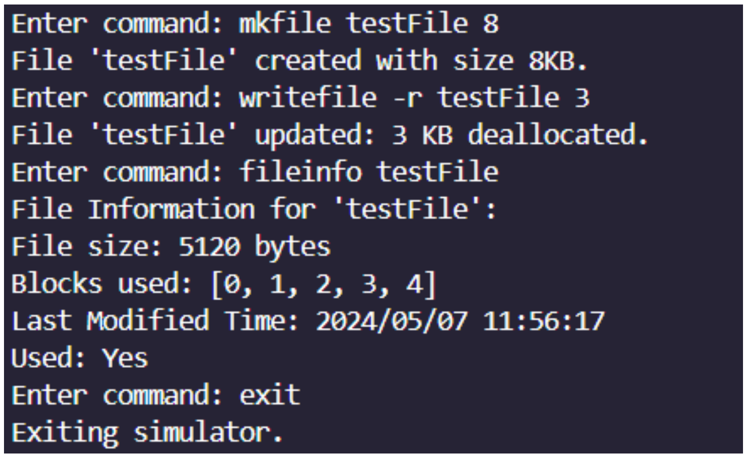
```
Enter command: showsystem
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
Enter command: mkfile testFile 4
File 'testFile' created with size 4KB.
Enter command: showsystem
[1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

13. **writefile -a <fileName> <sizeChangeInKB>** - Append to a file



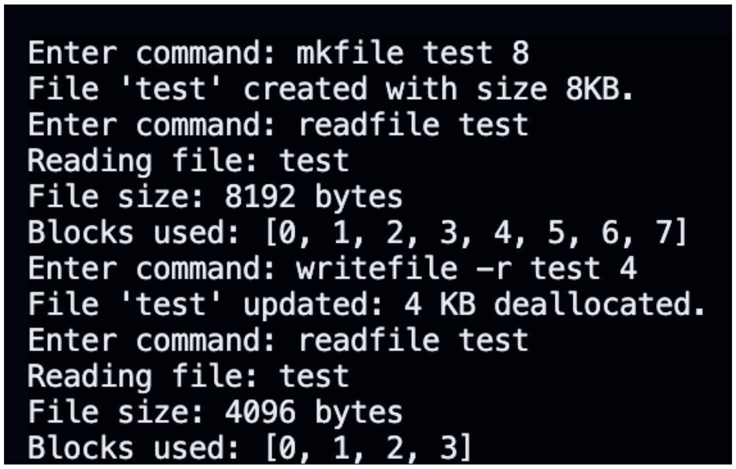
```
Enter command: mkfile testFile 7
File 'testFile' created with size 7KB.
Enter command: writefile -a testFile 3
Error: File size cannot exceed 8 KB.
Failed to modify file size.
Enter command: writefile -a testFile 1
File 'testFile' updated: 1 KB allocated.
Enter command: fileinfo testFile
File Information for 'testFile':
File size: 8192 bytes
Blocks used: [0, 1, 2, 3, 4, 5, 6, 7]
Last Modified Time: 2024/05/07 11:53:23
Used: Yes
```

14. **writefile -r <fileName> <sizeChangeInKB>** - Reduce a file size



```
Enter command: mkfile testFile 8
File 'testFile' created with size 8KB.
Enter command: writefile -r testFile 3
File 'testFile' updated: 3 KB deallocated.
Enter command: fileinfo testFile
File Information for 'testFile':
File size: 5120 bytes
Blocks used: [0, 1, 2, 3, 4]
Last Modified Time: 2024/05/07 11:56:17
Used: Yes
Enter command: exit
Exiting simulator.
```

15. **readfile <fileName>** - Read a file



```
Enter command: mkfile test 8
File 'test' created with size 8KB.
Enter command: readfile test
Reading file: test
File size: 8192 bytes
Blocks used: [0, 1, 2, 3, 4, 5, 6, 7]
Enter command: writefile -r test 4
File 'test' updated: 4 KB deallocated.
Enter command: readfile test
Reading file: test
File size: 4096 bytes
Blocks used: [0, 1, 2, 3]
```

16. **runscript** - Run a script file with a list of commands [See Section 5 - Measurements]
17. **clear** - Clear the screen
18. **exit** - Exit the simulator

7. Conclusion

In conclusion, the project provides a comprehensive simulation of a file management system, offering insights into the underlying mechanisms of file system operations. Through practical implementation and testing, we gain a deeper understanding of file management concepts and their implementation in real-world systems.

8. References

Ttsugriy. "TTSUGRIY/File-System-Simulator: The File System Simulator Shows the Inner Workings of a Unix v7 File System." *GitHub*, github.com/ttsugriy/File-System-Simulator.