South China University of Technology

《Operating System Course Design》Report

Experiment Title： Operating System Course Design

Name：Telanchakorn Thangmisri Student ID：202269990037

Name：Taha Murteza Morkoc Student ID：202269990134

Name：Awab Student ID：2022699900

Class：CST Group： Foreign 1

Teacher： 钟竞辉

|  |
| --- |
| **Description** |
| 【Objective and Requirement】  The purpose of this project was to implement a Unix-style file system which mimics some basic features of a conventional hierarchical file system. This implementation affords a very good overview of file system internals and gives us hands-on experience with concepts such as block allocation, inode management as well as integrating the disk data into directory structures. Implementation System has been implemented in C++, and it is in memory simulation, and it also includes persistent storage, where simulation is being persist on the disk in binary file named as **filesystem.dat**. The file system was designed with specific parameters to balance functionality and simplicity:   * Block size of 1KB (1024 bytes) * Total storage capacity of 16MB (16,384 blocks) * Support for up to 1,024 inodes * Maximum filename length of 28 characters   【Environment】  Operating System：Windows 11 |
| **Content** |
| File System Architecture File system architecture the file system is laid out in accordance with the classic Unix style, with storage being organized into a super-block, inode table, and data blocks. The superblock contains critical informations about the state of the file system such as block size and free block count and inode informations. All files and directories metadata are stored in inode tables, and the true content of the files and directories is stored on the data blocks.  **Key Data Structures**    **Code Implementation and Explanation**  **1. Block Allocation and Management**    This function will claim a block from the file system as free space for storing file or directory data. The function begins by testing for the existence of any free blocks by looking at the **freeBlocks** field of the superblock. If there are free blocks available, it takes the block number of the first free block from **firstFreeBlock** in the superblock.  The free blocks are kept in a linked list, where each free block has a block number of the next free block. Function updates the **firstFreeBlock** pointer so that it points at the next free block in the list, decrements the free block counter and then zeros the block. This guarantees that there will be no remnants in the fresh block.  **2. Path Resolution and Inode Finding**      This all-important function maps the file or directory pathname to its respective inode number. The first thing it looks for is absolute paths ('/') and dot-based paths. For an absolute path, it starts at the root directory (inode 0), for a relative path from the current directory.  It then explodes the path on '/' and traverses the directory tree with a loop. For each element, it checks whether the current inode is a directory, and if it is will do a read request in order to find the entry of the directory. Once it's found, it updates the inode number and moves on to the next component. If any component does not exist or is not a directory during traversal, the function returns an invalid inode number.     This is one of the basic functions of the file system because it is the functionality directly responsible for allowing you to move into a series of nested directories and is a precondition for all operations that require searching for a file or directory.  **3. Directory Creation and Management**      This method creates a directory in the file system. It then divides the input path into the parent directory and the new directory name. It then gumshoes back to the parent directory's inode with the **findInode** function.  Once it is verified that there is such parent and it is a directory, it verifies that there is no other file or directory with the same name there, assigns the parent reference to the respective slot, and updates the name. When it doesn’t, it would create a new inode for the directory node, sets its type to directory, and fills out its metadata (creation time, modification time, link count).  Special Domains The function then takes a date block for directory entries and creates the special entries "." (pointing to the directory itself) and “..” (pointing to parent directory). It also creates an entry for this new directory in the parent directory and updates the parent’s link count and modification time.  This implementation will take care of the directory hierarchy and the parent-child relation of directory as necessary for traversing the file system.  **4. File Creation and Writing**    This function creates a file with optional size. It starts with parsing command arguments, that contain filepath and size if set. It slashes the parrent directory path and the file name, just like the directory creation function.  The function verifies if a file with the same name exists. If it is, and is a regular file (not a directory), then the function reclaims it, freeing all of its data blocks and resetting its size. If the file does not exist, the function creates a new inode, makes the inode as regular file and initializes the metadata information of the file.  When a size is provided, it assigns data blocks of that size and may zero fill (the size would be 1 for zero fill) the blocks as required. This makes it straightforward to generate test files of a fixed size. The function also transverses direct blocks (for small files) and indirect blocks (for big files), and so on, so it illustrates the strategy to allocate the block at different levels as it happens in Unix-like file systems.  **5. Command Interface and User Interaction**    It has to be this way, because this is the definition of the primary interface on the file system. It is a prompt which shows currently working directory and your cursor is waiting for what you want to type. The parsed command name and arguments are passed to the function, which calls one of those handlers directly from the bot.  The shell supports both traditional Unix commands(**Is**, **cd**, **mkdir**, **rm**, **touch**, **cp**, **cat**, **sum** ) It also supports special things like **cd..** without space and **cd/path** where the path directly follows the command.  When it goes away, it calls **saveFileSystem()** to save the file system to disk so that changes are preserved across multiple invocations of the program. This interface is a simple and obvious way for a user to communicate with the file system and represents a fundamental example of the way the commands a user types are interpreted and executed by command-line shells.  **Special Features and Challenges**  **Handling Special Command Formats**  A key usability enhancement in this implementation is the support for shorthand command formats like **cd..** (without space) and **cd/path.** This is achieved by special case handling in the command parsing logic:    This is friendlier and less confounding, more like actual command-line interfaces people use in the real world.  **How to use:**   1. Touch - Create a new file      1. rm - Remove a file      1. mkdir - Create a new directory      1. rmdir - Remove a directory (must be empty)      1. cd - Change directory:   *cd documents*  *cd .. (go up one level)*  *cd / (go to root)*     1. ls - List directory contents      1. cp - Copy a file      1. cat - Display file contents      1. sum - Show file system usage summary      1. exit - exit and save     **Contribution**  Telanchakorn Thangmisri 202269990037 CST   1. **File System Initialization and Memory Management**    * Created **initializeFileSystem()** method to create the initial file system layout    * Designed the memory management for the in-memory file system representation    * Implemented the persistence mechanism to store and restore the file system state to disk 2. **Block and Inode Management**    * Implemented data block management functions **allocateBlock()** and **freeBlock()**    * Implemented **allocateInode()** and **freeInode()** functions for inodes allocation    * Implemented the free block list and free inode list management logic. 3. **Path Resolution and Navigation**    * Added the function **findInode()** for resolving file and directory paths    * Wrote the directory traversal logic to walk the file system directory structure    * Implemented the logic for path parsing to support absolute and relative paths 4. **File Operations**    * Added file reading and writing functions    * Coded the handling of the indirect block for big files.    * I have also designed the file size have block allocation policy.   Taha Morkoc 202269990134 CST   1. **Command Line Interface**    * Implemented the **run()** function that provides the main command-line interface    * Implemented the code to parse commands for command names and arguments    * Implemented special command handling for commands such as **cd..** and **cd/path**. 2. **Directory Management Commands**    * Added **mkdir()** function for making directories    * Implemented **rmdir()** for deleting directories    * Added **cp()** for navigating directories    * Added **ls()** to list a directory contents 3. **File Management Commands**    * Implemented **touch()**which is file creation that optionally takes a size    * Implemented **rm()**and file deletion    * Added **cat()** to show file content    * Implemented **cp()** for copying files 4. **System Information and Utilities**    * Added **sum()** to display storage usage.    * Added support for the help system so that users would have a guide to commands open to them.   Awab 202269990 CST   1. **Attending** **to the description** 2. **Making the report** |
| **Conclusion** |
| This Unix-like file system is a simplified version of file system principles and functions. It serves as a tangible example to learn how today's file system handle storage devices, store hierarchical directories, and run this in basic operation. The system effectively compromises between simplicity and quality, providing a useful didactic tool.  Key achievements include:   * Setup an operational hierarchical directory structure * Basic file and directory operations support * Permanent storage for preservation of the file system state between sessions * A friendly command-line interface, including lots of familiar Unix-like commands * Resolved major usability problems for some commands such as **cd..**, **cd/path**, **rmdir**, **cp** and **etc.**   This model can be used in teaching to illustrate how file system is organised, block allocation algorithm, metadata management, and how directory structure is updated. Although already usable, the system might be improved by adding file permissions, symbolic links, journaling, and defragmentation. |
| **Teacher’s Comments and Score** |
|  |
| Comment：  Score：    Signature：钟竞辉                                                 Date： |