South China University of Technology

《Operating System Course Design》Report

Experiment Title： Operating System Course Design

Name： Tohsaka-Sakura Student ID：

Name： 不会起名 Student ID：

Name： ShockWithAwe Student ID：

Name： SwordRain Student ID：

Class： 22计科全英创新班 Group： Chinese 5

Teacher： 钟竞辉

|  |
| --- |
| **Description** |
| 【Objective and Requirement】   1. Conduct an in-depth research into the core design principles of Unix-like file systems, focusing specifically on: The structure and role of i-nodes and their relationship with file contents; Methods for allocating and managing Data Blocks; The implementation of Directories (as special files); The role and information contained within the Superblock; Mechanisms for file access Permissions (simplified). 2. Design and implement an i-node-based Unix-style file system. 3. Implement basic functionalities specified in the following section. 4. The task needs to be completed using C++ ONLY   The following functions are required in your file system:   1. Allocate **16MB** space in memory as the storage for your file system. The space is divided into blocks with block size 1KB   Assume **address length is 24-bit，**please design your virtual address structure.  Design what information should be contained in an i-node  The i-node should support **10 direct block addresses, and** **one indirect block address**   1. The first a few blocks can be used for storing the i-nodes, and the first i-node can be used for the root directory (/).   (You can design the structure as you like, as long as it is reasonable and well  explained in your report.)   1. Using random strings to fill the files you created. It means you just need to specify the file size (in KB) and path+name. 2. Following commands should be supported in your system：    1. A welcome message with the group info (names and IDs) when the system is launched. It is also the claim of your ‘copyright’    2. Create a file：**createFile** **fileName fileSize**   i.e.：createFile /dir1/myFile 10 (in KB)  **if fileSiz > max file size, print out an error message**.   * 1. Delete a file：**deleteFile** filename   i.e.：deleteFile /dir1/myFile   * 1. Create a directory：**createDir**   i.e.：createDir /dir1/sub1 (should support nested-directory)   * 1. Delete a directory：**deleteDir**   i.e.: deleteDir /dir1/sub1 （The current working directory is not allowed to be deleted）   * 1. Change current working direcotry：**changeDir**   i.e.: changeDir /dir2   * 1. List all the files and sub-directories under current working directory：**dir**   You **also need to list at least two file attributes**. (i.e. file size, time created, etc.)   * 1. Copy a file : **cp**   i.e.: cp file1 file2   * 1. Display the usage of storage space：**sum**   Display the usage of the 16MB space. You need to list how many blocks are used and how many blocks are unused.   * 1. Print out the file contents: **cat**   Print out the contents of the file on the terminal  i.e: cat /dir1/file1   * 1. You are NOT required to implement the function of Login!  1. **Loading and exiting**: exit the program and release all the memory occupied, but the contents of the memory should be stored on your disk for re-loading；   【Environment】  Operating System：Linux. |
| **Content** |
| ①System Design In our design, we utilize a file on the computer to simulate the hard drive of our file system. We made the following assumptions:   |  |  | | --- | --- | | **Structure** | **Size(KB)** | | Block | 1 | | Super Block | 1 | | Inode Bitmap | 1 | | Inodes | 32 | | Data Blocks | 16349 |   The total space occupied is 16384KB which is 16MB.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Blank Block** | **SuperBlock** | **Inode Bitmap** | **Inodes** | **Data Blocks** | | 1KB | 1KB | 1KB | 32KB | 16349KB |   The super block contains the information of the number of Inodes and blocks in our file system and it contains a array the size of which is 300 called special stack. The array will use the group linking method to store the information of the using of the data blocks.  Inode bitmaps used to keep track of which inodes are used by a linear search.  Inodes structure is used to store all the Inodes of file and directories. It contains mainly 10 direct address and one indirect address. We use an array to store the serial number of blocks to represent the data block address. For the indirect address, we assume it points a data block and the data block will contain an array which has 256(1KB / sizeof(int)) integer number which points the data blocks.    In our file system, the location of file and directory both are the data block. The directory contains the filename or the subdirectory name and the inodes number correspondingly. This also includes the own directory and the parent directory. And the root directory is located at the start of the data block. So to find a any file, get the root directory and find the Inode number of the subdirectory, then get the Inode, and from the Inode get the subdirectory structure iteratively.    To achieve fast allocation and recycling of blocks and good management of blocks, we use the group chaining method. The basic idea is as follows:  In a data block, not all blocks are stored as data blocks; some are used to hold a special stack. In our file system, one block every 299 blocks is used to hold the special stack. The purpose of the special stack is to record the usage of the previous 299 blocks in its position. This stack also includes a data to record the position of the next stack. When the blocks in this stack are used up, it will jump to the next stack position to update the current stack and start the new block. To keep track of stack usage, the superblock also contains a special stack pointer to keep track of stack usage.   ②Function **1. void find\_free\_block(unsigned int& inode\_number)**  The purpose of this function is to find the available blocks of the current filesystem and obtain their sequence number. A list of free blocks is obtained using the stack in the superblock as well as the stack stored in the Data Block. The position of the available blocks of the current stack is obtained by means of the stack pointer in the superblock. When a stack is used up, the superblock uses the stack pointer to get the location of the next stack (as shown in the architecture), and then reads the next stack into the superblock, which is used as a stack to find free blocks.  **2. void recycle\_block(unsigned int& inode\_number)**  The purpose of this function is to reclaim usable blocks from a file or directory. When the block sequence number that should be recycled is obtained from the file or directory, the stack in the superblock is first searched to see if there is room left, the sequence number is stored and the stack pointer is moved back. If there is not enough room left, the stack in the superblock is updated by the current stack position minus 300, the next stack position minus 600 (as shown in the architecture diagram). At the same time, the stack in the superblock is written to block. The stack is updated before the sequence number is put in and the stack pointer is moved back.  **3. bool Format()**  Formats the file system, creating the superblock, bitmap, root directory, and system file.  **4. bool Mount()**  Mounts the file system by loading the superblock, inode bitmap, and current directory from file.  **5. string generateRandomContent(int totalSize)**  Generates a random English string of a given size for populating file content.  **6. bool ResolvePath(const string& path, string& filename, directory& targetDir, int& dirInodeID)**  The purpose of this function is to parse the path entered by the user to obtain the target directory, the inodeID of the target directory, and the file name.  **7. bool CreateFile(const string& path, int blockCount)**  Creates a new file at the specified path, allocating a given number of blocks with random English characters.  **8. bool DeleteFile(const string& path)**  Deletes a file and releases its inode and all associated blocks.  **9. bool Copy(const string& srcPath, const string& destPath)**  Copies a file from source to destination, creating a new inode and block allocations.  **10. inode\* OpenFile(char\* filepath)**  Since users may provide absolute or relative paths to read file content, we need to determine the type of path entered by the user to decide whether to start the search from the root directory or the current directory. The determination is made as follows: if the path starts with “/”, it is considered an absolute path; if it starts with “.”, the search should begin from the current directory or its parent directory; if the path does not start with either “/” or “.”, the search defaults to starting from the current directory. After obtaining the inode of the starting directory for the search, we traverse the user-provided path to locate the target file.  If no matching file is found, an error message is displayed, indicating that the target file was not found. If a matching file is found, we need to determine whether it is a directory. If the target file is successfully located, a pointer to its inode is returned to facilitate subsequent reading and printing of the file content.  **11. void PrintFile(inode& ifile)**  The function PrintFile is responsible for printing out the file content. Based on the inode of the target file returned by the OpenFile function, we calculate the number of blocks the file occupies using the file size information contained in the inode.  If the file occupies no more than 10 blocks, it means that the file does not use indirect block address. In this case, we simply retrieve the content stored in each block of the file according to the direct addresses recorded in the file’s inode and store it in a “stack”. The content is then printed line by line, with each line containing 128 bytes for easy viewing.  If the file occupies more than 10 blocks, the processing of the first ten blocks is the same as before. For the subsequent blocks, we first locate the block storing the indirect addresses and read the addresses of the file blocks stored therein. We then find the data blocks based on these indirect addresses and print their content, a process similar to that for direct addresses.  **12. bool OpenDir\_Specify(const char\* dirname)**  Opens a specified subdirectory and updates the current directory pointer. The OpenDir\_Specify() is used to open the specified subdirectory under the current directory.  ① Parameter verification. This stage will detect the case that the directory name is empty or too long (the limit of the global variable FILE\_NAME\_LENGTH). If an exception is found, false will be returned.  ② Directory entry lookup. This stage will traverse the DIRECTORY\_NUM entries of the current directory. When a file with the same name is encountered, skip directly (only the directory type is processed).  ③ Directory switch. First, the function will read the target directory data block to new\_current\_dir (that is, the directory to jump to). Next, for different types of directories, there will be different operations to update the current directory and absolute path. Specifically, for ".", there is no operation; For "..", the path pointer will decrease by 1 (that is, the fallback level); For normal directories, the function will write the absolute path array ab\_dir and increase the path pointer.  **13. bool OpenDir(const char\* dirname)**  Navigates into directories step-by-step based on a full path (absolute or relative). The OpenDir() is used to handle multi-level paths (including absolute paths). The OpenDir\_Specify() function in part 12 will be called during use.  ① Backup and error recovery. In this phase, the current directory state will be backed up for rollback in case of subsequent errors. When any subdirectory fails to open, the function will restore the original directory and path pointer, and re read the original directory data in the disk.  ② Path resolution. Use the strtok() function to split the path according to "/". For absolute paths(when the input string starts with "/"), it will be reset to the root directory.  ③ Level by level directory switching. Call OpenDir\_Specify() to enter the subdirectory.  **14. bool MakeDir\_Specify(const char\* dirname)**  Creates a new subdirectory in the current directory. MakeDir\_Specify() is used to create a single level directory.  ① Compliance check. This part mainly checks whether the current state of this function conform to the settings of the current operating system. Therefore, the compliance check at this stage includes: 1) parameter check: check the directory name and the length; 2) Space check: check the free resources of the super block, and call the find\_free\_block() function to get the free block; 3) Duplicate name check: traverse the current directory entry(allow the directory to have the same name as the file); 4) Directory item upper limit check: check the number of directory items.  ② Create directory. There are two aspects involved in creating a new directory. On the one hand, you need to create a new inode and set the corresponding directory metadata: then write to the disk inode area; On the other hand, create a directory file and initialize the directory structure containing "." (current directory) and ".." (parent directory).  ③ Update status. Finally, the function needs to modify the update status. First, modify the inode bitmap and write to the disk; Next, update the parent directory entry; Then, the link count of the parent directory is updated recursively; Finally, the superblock free inode count is updated.  **15. bool MakeDir(const char\* dirname)**  Creates a nested directory structure, including intermediate levels. MakeDir() is used to handle multi-level paths (including absolute paths), and supports error recovery. MakeDir\_Specify() function will be called during use.  ① Path resolution. In this stage, the path will be divided into hierarchical arrays. When the absolute path is determined, it will be switched to the root directory for processing.  ② Create directory level by level. First of all, the function traverses the hierarchy and performs sequential processing on the names of each hierarchy after path decomposition: from the first hierarchy to the next; The last level is the name of the directory to be created; The intermediate level is the transition directory of the path. Next, the same name item detection will be performed, that is, the file system entity with the same name as the current level will be retrieved in the current working directory. When an entity with the same name is detected, if it is a directory type and not the end level, it will switch to the directory to continue processing the subsequent levels; If it is a directory type and is the end level, the trigger directory already has an error; If it is a file type, skip. In addition, if a directory with the same name is not detected, the directory creation process will be directly triggered. Finally, the function will enter the process of dynamic directory creation. Specifically, the function will call the MakeDir\_Specify() function to build the directory of this level, enter the new directory immediately after the creation is successful, and continue to process the subsequent levels based on the new directory.  ③ Backup and error recovery. The function will back up the current directory state at the beginning and restore the backed up directory state at the return value. When the creation of any level fails, it will immediately terminate and roll back to the initial directory state.  **16. bool isAncestor(const char\* ancestor, const char\* descendant)**  Checks if one path is an ancestor of another path. Specifically, to prevent deleting the current directory or its ancestor directory, an auxiliary function isAncestor() is required before the RemoveDir() functio, which is used to check the ancestor or equality relationship of the path by comparing whether the path prefix matches and verifying that the suffix is "/" or '\0'. In the process, the RemoveDir() function will call the isAncestor() function to prevent the deletion of the current directory or its ancestor directory.  **17. bool RemoveDir(const char\* dirname\_input)**  Recursively deletes a directory and all its contents, with path safety validation.  ① Path processing. The function will build the current absolute path and then parse the target absolute path. On this basis, the function will parse the parent directory.  ② Delete operation. First, linearly search the directory entries and empty the file name and inode index. Then, the hard link count is used to determine whether to delete the subitems of the target directory. Finally, if it is determined that there are no other links in the previous step, the children will be deleted recursively through DFS.  ③ Resource recovery and renewal. The final stage will reclaim data blocks, empty the corresponding inode disk space, and update inode bitmaps and superblocks.  **18. void List()**  Lists detailed information for all files and subdirectories in the current directory.  **19. bool filePermission(const char\* filepath)**  We simply classify file access permissions into two categories: system files and user files. System files are created during the program initialization phase. Users are not allowed to delete or copy them and can only view their content. User files, on the other hand, can only be created by users, and users have full permissions over them. Therefore, when a user needs to delete or copy a file, we call the filePermission function to check whether the file is a system file. If it is, an error message is output to inform the user that the system file cannot be modified, and the corresponding operation instruction is not executed.  For the implementation of the filePermission function, we first need to find the inode of the corresponding file based on the input path. If the file is successfully found, we read its inode and check the “permission” attribute stored in it to determine whether the file is a system file or a user file. If it is determined to be a system file, we output a message to inform the user that the system file cannot be modified.  **20. void CommParser(inode\*& currentInode)**  Parses and executes command-line input, providing a shell-like user interface.  **21. int main()**  Program entry point; mounts or formats the system and launches the command parser. ③Experiment The correspondence between instructions in the program and instructions in the task book is:   |  |  | | --- | --- | | **Our Code** | **Task Requirement** | | exit | exit | | help | None | | ls | dir | | cd + dirname | changeDir + dirname | | mkdir + dirpath | createDir + dirpath | | rmdir + dirpath | deleteDir + dirpath | | create + filename + file size | createFile + fileName + fileSize | | cat + filepath | cat + filepath | | cp + sourcefilepath + targetfilepath | cp + file1 + file2 | | rm + filename | deleteFile + filename | | info | sum |   On Linux systems, run the command:  *g++ -o OS\_Course\_Design OS\_Course\_Design. cpp && ./OS\_Course\_Design*  in the directory where the OS\_Course\_Design. cpp file is located to compile and run the program. Because this is the first time running the program, it will automatically generate a stored file.  1749798373057  Enter the command 'help' to get assistance.  da49dfcaad93fd98b238b80a0013d4f  Test permission restriction: Users can only read system files but cannot delete or copy them.  9fc1903e5f89cfd0ad1263659f640f7  Enter the command 'info' to retrieve system information. The info command displays the usage of each block in the block stack, as well as the total number of blocks, free blocks, total inodes, and free inodes in the system.  c61708c67f9deff7fa2e98f445602c2  Create directories using relative and absolute paths.  deb9262061fc8adf7ecf8a9228cae90  Enter the command 'info' to determine if inodes and blocks are being used correctly. Created 4 directories consuming 4 inodes and 4 blocks.  431bba20523c1a43ca8c7627cc3bf23  Test using relative paths to create and delete files. The size of the file cannot exceed the maximum file size and the remaining number of blocks, and if there is a file with the same name, it will not be created.  575416636bc5f3b9655fa9091a9c332  Test using absolute paths to create and delete files.  190527b3bfb2bfe54204826058a16ec  Test using relative path to copy files. When the source file does not exist and there is a file with the same name in the target directory, the new file will not be copied over. The cat results indicate that a.txt and c.txt have the same content.  c62f63ba7eb59295625e14ff6af0a05  IMG_256  Test using absolute path to copy files.  IMG_256  Test the file saving status after exiting the program. When the program is opened for the second time, it will read the stored file.    Test relative and absolute path deletion directory.  84ecfc021086e532fd26a250011bdf0  **④Member Contributions**  In this course design, our group jointly completed the system design (including super blocks, files, inodes, etc.) and calculated the correct constants. After the overall code was completed, the four of us worked together to deal with bugs. Here are the unique contributions of each of our members:  Tohsaka-Sakura:  Designed the overall architecture of the filesystem, including the design of Inodes and data Block areas, and used Inodes to link the various parts of the filesystem such as files and directories. At the same time, the find\_free\_block(), recycle\_block() and Format() functions are designed, which are the lowest and one of the most important parts of the whole file system. The group linking method of blocks provides an efficient way for other functions to obtain and reclaim free blocks.  不会起名:  In this course project, I am primarily responsible for implementing two functions in this file system: file content output and file permission handling (filePermission()). File content output is divided into two steps: reading the file inode (OpenFile()) and reading & outputting file content (PrintFile()). Both OpenFile() and filePermission() involve path parsing and alternating inode - directory searches. PrintFile() includes locating file content through direct and indirect addresses stored in inode.  ShockWithAwe:  In this course project on operating systems, I was responsible for implementing six core components of the part of the file system that involves directory operation, including OpenDir(), OpenDir\_Specify, MakeDir(), MakeDir\_Specify(), isAncestor() and RemoveDir(). The part about directory is an important part of the whole file system (inode, directory and bitmap are used), including directory jump, creation, deletion and several other functions.  SwordRain:  In this course project on operating systems, I was responsible for implementing three core components of the file system: CreateFile, DeleteFile, and Copy. These features are among the most fundamental and essential operations in any file system, requiring precise handling of inode allocation/release, block management, path resolution and directory structure updates. |
| **Conclusion** |
| In this experiment, we successfully designed and implemented a simplified Unix-like file system using C++. Through defining structures such as superblocks, inodes, directories, and data blocks, we realized fundamental functionalities including file creation, deletion, copying, reading, directory operations, and permission management.  Throughout the development, we deepened our understanding of key file system concepts such as block allocation, inode indexing (including direct and indirect addressing), directory structure, and file permissions. We also encountered and resolved practical issues such as file path resolution, recursive directory operations, and memory consistency.  This project not only reinforced theoretical knowledge from operating systems courses but also improved our programming skills in system-level development. It laid a solid foundation for further exploration into advanced topics such as journaling, virtual file systems, and multi-user management. |
| **Teacher’s Comments and Score** |
|  |
| Comment：  Score：    Signature：钟竞辉                                                 Date： |