South China University of Technology

《Operating System》 Project Report

Experiment Title: Index-node-based File System Design

Name: 向天翼 Student ID: 201836020389

Name: 宋浩瑞 Student ID: 201830581404

Class: 18计算机全英联合班 Group: 1

Partner:

Advisor: 钟竞辉

|  |
| --- |
| **Description** |
| **[Objective and Requirement]**  **Objective:** Design and implement a real file management system based on index-node architecture.  **Requirements:**  1． Review Unix file system design and index-node usage.  2． Design and implement an index-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  **[Environment]**  Windows 10, Visual Studio 2019 |
| **Content** |
| 1. **Architecture introduction**   According to the project requirement, we totally have 16 MB space and it’s divided into blocks with block size 1024 B. So, we distribute the 16384 blocks to four parts, which is showed in Figure 1. Here is the detail of each part:   1. **Super block:** Super block occupy the first block of this disk and contains the control information of the disk structure. The first 8 B of super block is the ‘magic number’, which indicate that the disk is valid. The control information including total and free index-node number, total and free blocks number, the size of block, index-node, and total disk. It also contains the start address of super block, the index-node blocks and data blocks area. Moreover, the index-node bitmap and current free block pointer is also stored in super block. 2. **Index-node blocks:** In our design, we totally have 4,096 index-nodes. Each index-node occupies 128 B of space. So, we should allocate blocks for storing index-node. This means that the total amount of all directories and files in the file system is up to 4,096. The index-node not only stores the size and creation, modification and access time of the file, but also stores the index-node number of the directory to which the file belongs and the address of the file block. Since the directory. Because the directory is also treated as a special file in this system, the index-node also stores the type of the file to identify whether the index-node points to a directory or a file. 3. **Free block address blocks:** In order to manage the free blocks in the disk, we need to use some blocks to store the addresses of the free blocks in the current disk. Initially, 47 blocks are used to store all the free addresses. As disk usage increases, the number of these address storage blocks will gradually decrease. The detail of allocating data blocks and setting them free are discussed in next part. 4. **Free blocks:** These blocks can be allocated to store directory or files.     Fig.1 The whole structure of our file system (not draw to scale)   1. **Details of system design**   **2.1 System hierarchy**  We build a four-layer model to manage the disk and lower layers provide interface for higher layers to implement its corresponding functions, which are showed in Figure 2. From bottom to top, the layers are:   1. Basic disk operation layer: This layer mainly implements the most basic reading and writing of the disk and the locating operation of the disk read and write pointer 2. Block and index-node operation layer: This layer implements the read and write, allocate and release operations of blocks and index-node. It is also responsible for reading and writing indirect block, updating super block and formatting the disk. 3. File and directory operation layer: This layer mainly implements some basic operations for files or directories, including reading and writing directories, allocating space for new files or directories, or releasing space when files or directories are deleted. 4. User operation layer: Functions in this layer mainly implement the creating, deleting, changing current directory and copy operations for user by calling the functions in the file and directory operation layer.     Fig.2 The hierarchy design of our system  The following sub-sections will given the implementation detail for each layers.  **2.2 Basic disk layer**  In this layer, we implement bytes-based operations on the disk, including five functions which is showed in Figure 3. These five functions actually call the build-in function of C++ to implement operations on files. But considering that the file operation functions provided by C++ will not print error messages when an error occurs in the operation, we encapsulated these functions to get these five functions.    Fig.3 Functions in basic disk layer  Table.1 Functions description of basic disk layer   |  |  | | --- | --- | | **Function Name** | **Description** | | fileSeek | Locate file operation pointer to given position | | fileOpen | Open a disk file | | fileRead | Read bytes from disk file to the buffer | | fileWrite | Write bytes to disk file from the buffer | | filePutCharacter | Write a character at the location of the file pointer (used to create empty file with given size) |   **2.3 Block and index-node operation layer**  In this layer, we implement the block and index-node oriented operations, which are disscused as follow.  **2.3.1 Block operation**  After we implemented the disk driver, we need to implement disk block manager.  The Address is 24 bits (3B). The first 14 bits locate the blocks and the last 10 bits represent the offset position within a block. The layout of the disk address is as follows.    Fig.HH Disk address layout  After we partitioned the disk block storage, it’s time to designed the free block management algorithm. We employed dynamic doubly linked list blocks to store the free block addresses. We maintained or regarded it as a stack, with a free pointer pointing to the top of the stack which contains a free block address. The structure of the doubly linked list is demonstrated in the following picture. In this example, there are 4 disk blocks used to store the free block addresses. The block addresses of the four blocks are “addr”, “free addr1”, “free addr2”, “free addr3” respectively, the reason of which will be explained later in the special case part. The first address in each block except the first one is the address of the previous block. Since the first block does not have a previous block, the first address in the first block then points to the block where it is located. The free pointer points to the last free address, which is at the top of the free pointer stack. When other function requires a new free block, the disk block manager can directly return the address pointed by the free pointer and move the free pointer upward for one step. Or when other function wants to release a block, it just needs to pass the block address to disk block manager and then the disk block manager will add the free address to the list and move the free pointer downward for one step. Notice that the value of free pointer is stored in the super block, so whenever the free pointer changes, it need to update the value in the super block and write it to the disk.    Fig.HH Structure of linked list blocks  Besides, there some special cases for releasing and allocation, which is discussed in following paragraphs.  **Special cases of releasing:** when the last block storing addresses is full while another block is waiting to be released. This case is demonstrated in the following pictures. All the three blocks are full of free block addresses. We denote the incoming new free block address as “new free addr”. At this time, the block with address “free addr3” is definitely free. Therefore, we take that block as a new block to contain the incoming address to be released. Finally the “free addr3” is converted to a pointer to the new block and the whole list is still a doubly linked list.    Fig.HH Special case of releasing (before releasing)    Fig.HH Special case of releasing (after releasing)  **Special case of allocation:** when the free pointer points to the first address in a block and other function requires a free block. This case is demonstrated in the following pictures. The disk block manager will first check whether the first address in the block points to the block where it locates so as to check whether still free block left. If it does not, the manager will move the free pointer to the last address in the previous block. At this time, the block “free addr3” is free, so it can be directly allocated to the function which requires a block. Finally, move the free pointer upward for one step.    Fig.HH Special case of allocation  (Step.1: the free pointer points to the first address in a block)    Fig.HH Special case of allocation (Step 2: addr3 is added to the list)    Fig.HH Special case of allocation  (Step 3: addr3 is returned, and block on addr3 is allocated)  Unlike most methodologies, we did not introduce a bit-map to maintain the free blocks because our dynamic doubly linked list is efficient enough. Both “free” and “allocate” operations are of complexity. And it only needs to traverse the list to count how many blocks are used, which is of complexity and still fast to operate. Moreover, it also occupies less space than bit-map. When the majority of blocks are used, then the size of the list is small. When almost all the blocks are not free, the number of blocks used by the linked list is only one.  Based on the methodology, we define a block manager class which is showed in following figure. The *alloc* and *free* functions is responsible for allocating and releasing data block based on the methodology we discussed above. And the *initialize* function is used to initialize the linked-list during the formatting process.    Fig.HH Implementation of disk block manager class  Once the *alloc* and *free* is called, since the free pointer will be update and points to the new free block address, we need to update the free block number and free pointer in the super block. So we further more encapsulate these two functions in super block, which is showed in following figure.  In the two functions we encapsulate, when successfully calling *alloc* and *free*, we update the free pointer and free block number in the super block. And then write the latest super block to the disk. When applying for data blocks later, we will directly use these encapsulated allocating and release functions in the Disk class.    Fig.HH Encapsulated allocating and releasing function for data blocks  **2.3.2 Index-node operation**  Index-node operations involve allocating, freeing, reading and updating operation, which is showed in Fig. HH Since we totally have 4,096 index-node and a character occupies 8 bits in C++, we store an characters array with size in super block to identify the usage of each index-node. Each bit in the array corresponds to each index-node one-to-one , and 0 means availabel and 1 means used. So, we can use bit operation (not, and, or) to read and modify the usage of each index-node.  When allocating a new index-node, the program will do the linear search on the index-node usage array in the super block. If a 0 is found then the given file size, parent index-node ID, block address and file type (file or directory) will be used to create a index-node object and write it to the index-node blocks area according to the position the 0 found.  The releasing, reading and writing operation is very simple. For releasing, we only need to modify the corresponding bit in the usage array. For reading and writing, we can use the index-node ID to locate the disk pointer to the correct position and then read or write a index-node object.    Fig. HH Index-node operation functions    Fig. HH Implementation of index-node class  Table.2 Functions description of index-node operation   |  |  | | --- | --- | | **Function Name** | **Description** | | allocateNewInode | Allocate a new index-node and store the given file information. The function return the index-node ID number. | | freeInode | Release a used index-node according to the index-node ID number. | | loadInode | Load a index-node from disk according to the index-node ID number. | | writeInode | Update a index-node based on a index-node object. |   **2.3.3 Disk loading and formatting**  When the system starts, if the disk file already exists, we first read the first 8 bytes of the file and compare it with the preset magic number. If the two are the same, it means that the file is a legal disk file and can be read into the super block and the root directory.  If the disk file is not exists, we first create a file with size 16MB and filling with all zero using *filePutCharacter* function, and write the magic number to the start of the disk file. Then we call the *initialize* function of DiskblockManager to initialize all the free block address blocks and free pointer, and update them to the super block object. Finally, we use the corresponding allocation function of data block and index-node to initialize the root directory and set current index-node to the root directory. This process is showed in following figure.  Considering the efficiency of reading and writing files, here we store the created file operation object, so that this object can be called directly when the file is read and written later, so that it is not necessary to open the file.    Fig.HH Implementation of loading disk  **2.3.4 Data and indirect address blocks reading and writing**  In our system, we design two classes, Diskblock and IndirectDiskblock, to manager the data block and indirect block respectively. The Diskblock class has a buffer, we can use *load* and *write* function to read from or write to the disk block according to the given absolute address.  For indirect disk block, since the length of our address is 24 bits = 3 B, the maximum number of addresses that one block can store is The length of address array of IndirectDiskblock is 341 and the size of IndirectDiskblock is 1023, so we can directly store the array to the disk. The *load* and *write* function is used to read the address array from or write the address array to the disk according to given address.  The following figure shows the data structure of these two classes.    Fig.HH Data structure to manager disk block and indirect disk block  **2.4 Block and index-node operation layer**  **2.4.1 Directory in our system**  In our system, the directory is actually a table, which records the index-node ID and name of all its child file or directory. It should also record itself and its parent directory. Specially, the parent of root directory is the root directory itself.  Therefore, we design a Directory structure with a vector to manage the directory in our file system. Each element of the vector is a fileEntry object. fileEntry can be seen as a row of the table, recording the index-node ID and name of the file. The size of a fileEntry object is set to 64 and maximum file name length is 62. So a block can store 128 fileEntry object in maximum.    Fig.HH Data structure of directory  **2.4.2 Read/Write directory file from/to disk**  Since directories are also treated as files in this system, the size of each directory file depends on the length of its file table. For example, a newly created directory initially contains only two fileEntry, one representing itself and the other representing its parent directory, so the size of the directory file is 128 B. The directory file may also need to use indirect blocks because it is too large, so When reading and writing catalog files, you also need to consider whether indirect blocks are used.    Fig. HH Implementation of read/write directory file from/to disk  In this system, we use the size of the directory file to calculate the block occupied by the directory file. If only the direct block is occupied, we directly read each fileEntry from the direct block. If the indirect block is occupied, after we read the direct block, we need to read the indirect block, then read the remaining fileEntry through the address in the indirect block, and then read all the fileEntry. The two functions is showed in Figure HH.  **2.4.2 Allocate/Release resource to file and directory**  If user wants to create or delete a file or a directory, the program will correspondingly allocate and release the index-node and blocks. However, considering that users may involve multiple levels of directories or recursive deletion when users create and delete files, we only consider the simplest case at this level, that is, the index-node of a given parent directory, and we need to provide The created file or directory allocates space. Or given an empty directory or index-node of a file, we need to release the block and index-node occupied by the corresponding file and directory according to the information in it. We design three functions to implement this requirement, which is showed in following figure. The details of these three functions is discuss in Table 3.    Fig.HH Functions to allocate/release resource of file and directory  Table.3 Functions description of allocate/release for file/directory operation   |  |  | | --- | --- | | **Function Name** | **Description** | | allocateResourceForNewDirectory | Allocate a new index-node and a new block for the directory. The corresponding initial file entries (including itself and its parent) will be written to the new block. The new file index-node ID will be returned. | | allocateResourceForNewFile | Based on the given file size and parent index-node, this function will allocate a new index-node and several blocks for the new file. The blocks requirment is calculated from the file size. The file content will be randomly filled and then written to the blocks. The new directory index-node ID will be returned. | | deleteFile | Based on the given file size, this function will release all the blocks that the file or directory have occupied and then release the index-node of this file. Then the function will delete the corresponding file entry in the parent directory. If all the operations are successful, the function will return a true value. |   **2.4 User layer**  In this layer, we need to implement some specific functions for user including create, delete and copy file or directory, print a file or change working directory and so on.  **2.4.1 Create file**  User can input command *mkfile* or *createFile* to activate the operation. The parameters are the path to the new file and the size of the file. After checking the parameters, it’s required to locate the parent directory of the file. If any of the directories along the file path does not exist, then new directory with the specified name is created. After all the directories along the path fulfill the requirements, it’s time to create the file. Before creating the file, free blocks and index-nodes should be checked. If there are not enough free blocks or spared index-nodes, the creation should be ceased. If passing the space checking, then creation will begin. The creation operation is divided to two procedures. The first one is to create file under a new index-node. This procedure includes allocating enough free blocks, initialize the file with random string of the specified size, and filling in the information of the file to the index-node. The second procedure is to add a file entry to the parent directory. The apropos code is as follows.    Fig.HH Implementation of create file  **2.4.2 Create directory**  User can use *mkdir* or *createDir* to create a new directory under current working directory. We first divide the path entered by the user according to slash, and look up from the current directory level by level through the results of the division. If there is a corresponding directory, continue to search, otherwise create a directory. Before creating a directory, we first check whether the number of index-nodes and blocks is sufficient. If the conditions are met, we first allocate index-nodes and blocks to the new directory by calling the interface, and add a corresponding fileEntry to the parent directory to achieve the creation of the directory.    Fig.HH Implementation of create directory  **2.4.3 Print out file content**  User can input cat and the path to the file to execute the operation. The first task is to obtain the index-node of the file. Then it’s required to calculate the number of direct and indirect blocks used according to the file size recorded in the index-node. Then get the addresses of the blocks containing data and print the contents of the blocks in order. The core code is as follows.    Fig.HH Implementation of file content printing function  **2.4.4 Copy file or directory**  The format of this command is “cp source\_path target\_path”. According to the source path to the file/directory, we can calculate the blocks and index-nodes occupied by that file/directory. Before copying, the first task is to check whether there are enough free blocks and free index-nodes. If space requirement is satisfied, then invoke the “copy” function. The “copy” function is a recurrent function receiving the source index-node, the new file/directory name, and the target parent directory index-node. If the source index-node represents a file, then copy the content to the new index-node created under the target directory. If the source index-node represents a directory, then create a new directory under the target directory and apply “copy” to all of child index-nodes under the source directory to the newly created directory. Copying a file to a target directory is similar to creating a file except that one should copy the content to fill the blocks other than randomly initializing the content.    Fig.HH Implementation of copying file or directory  **2.4.5 Delete file**  Users can use *rmdir* or *deleteFile* to delete a file from disk. Then we locate the index-node of the file from given path and call *deleteFile* function to delete the file according to the index-node, which is showed in following figure. This process is quite simple since only one file need to be deleted.    Fig.HH Implementation of file delete function  **2.4.6 Delete directory**  Delete a directory is more complex than delete a single file. Since all the child file and directory under the directory should be deleted. Here we use Depth First Search (DFS) algorithm to recursively delete a directory. Before deleting, we first check whether the target is a directory and whether the target is current working directory. If the command from user pass this check, we use a function to recursively delete the directory, which is showed in Figure HH.  This function first check whether the directory that the index-node points to is an empty directory. If it does, then we directly remove this directory and return. Otherwise, we will load all the entries from the table of the directoy. For each entry, if it’s a file, then we use *deleteFile* function to delete it. Otherwise, we call the function *recursiveDeleteDirectory* to recursively process the child directory. After all the children are deleted, we use *deleteFile* function to delete current directory itself.    Fig.HH Recursively delete a directory    Fig.HH Implementation of delete a directory  **2.4.7 Change current working direcotry**  Users can input *cd* or *changeDir* to change current working directory. We only need to locate the new index-node according to the path given by user, and set current index-node to the new index-node, then it’s done. Figure HH shows the detail of locate an index-node from given path.    Fig.HH Implementation of locating an index-node  **2.4.8 Print disk usage**  Users can input *sum* to check out the usage of disk blocks, index-nodes and total space.  **Calculate usage of total space:** We can also use a DFS to the directory tree to calculate the total occupied space, which is showed in Figure HH. If current index-node points to a empty directory or a file, we directly return its size. Otherwise we recursively calculate all its children size and them to the summation result.    Fig.HH Calculation of total space usage  **Calculate usage of index-nodes:** The free index-node number is store in super block, so we can directly read it.  **Calculation of free blocks:** Given the free block pointer, we can locate the last block in the linked list, meaning that we can get the pointer to the previous block. Through traversing on the pointers, we can count the number of blocks used to containing free block addresses. All the blocks contain (number of addresses - 2) addresses to free blocks other than the last block. So there are (number of addresses - 2 ) \* (number of blocks - 1) addresses pointing to free blocks. Finally, add the offset in the last block to that number and we can get the number of free blocks. This calculation detail is showed in Figure HH.    Fig.HH Source code of calculating free blocks  **2.4.9 List directory**  Users can input *dir* to get all the directory and files under current directory. This function is very easy to implement. We first load all the file entries and then for each entry, we load the corresponding index-node to get the file attribute and print them on the screen, which is showed in Fig.HH.    Fig.HH Implementation of listing directory   1. **Demonstration**   Here is the demonstration of our file system. When you run our file system for the first time, it will create a brand-new disk file which have only a root directory.     1. *help* command:   User can type *help* to get instructions of our file system. You can follow the guidance of how to use each command.     1. Create and cat file:   You can use *mkfile* or *createFile* command to create a file. Remember to specify the size of the file. If you want to read the content of the file, then use the “cat” command. Here I make a helloworld.txt with 10KB under root directory. Then I output the blocks and index-nodes usage using *sum* command. We can see that index-node usage number is 2. I use *dir* command to check out the content of the directory. We can see that a new file named helloworld.txt appears. I used *cat* command to see the content, then it output the content of the file (too long don’t show completely).         1. Create a directory:   User can use *mkdir* or *createDir* to create a new directory. Notice that this command also supports nested creation. Continuing the above state, I make a new directory named *shr* under root. We can see the directory created successfully. Then I create a nest directory named *xty/hhh* then we can see that the two directories created successfully (*cd* is the command to change the working directory).      Moreover, we can also create nested file. Here I create a file named “nestfile.txt” under a nonexistent directory “testnest”. When I “cd” to “testnest” and use “dir”, we can see that “nestfile.txt” is created successfully.     1. Copy file/directory to another directory:   User can use “cp” command to copy a file or a directory to a directory. Notice that user need to specify the target file/directory name. Here I copy the directory “xty” to a new directory. Before copying I print out the space and index-node usage. Then I copy it to a new directory named “testcp” under root directory. Notice that “xty” occupies 4 index-nodes and 8 blocks. After copying I let it print out the blocks and index-nodes usage again, which printed correctly. Then I enter the “testcp” directory to check out whether all the directories and files are copied correctly. Finally I cat the two “nestfile.txt” to checkout whether the contents are identical. Eventually, it turns out that the copying operation executes successfully.           1. Delete file/directory:   User can delete file or directory using “rmfile” and “rmdir” respectively. Before deleting I print out the blocks and index-nodes usage first. And then I deleted file “helloworld.txt”. Then I print the blocks and index-nodes usage again. We can see that the blocks and index-nodes are released successfully.    Then I delete the directory “testcp”. We can see that all the information is correct.     1. **Individual contributions**   **Haorui Song**: I am responsible for the disk blocks management. I designed the structure of the dynamic doubly linked list and also the free and allocate function. I encapsulated those functions to the disk block manager which provides interface to the upper layer and other functions whenever they want to require or release a block. I also implemented the file creating, file content printing and disk usage calculating functions.  **Tianyi Xiang**: |
| **Summary** |
| Through this project, we have a deeper understanding of how the UNIX file system works. We also have learned how to effectively manage disk storage. By implementing the function of the file system using C++, we get clearer about the structure and organization of the index node and how to operate on the index node.  During this period, we also experienced some difficulties, such as ignoring the duplication of directories, stack overflow caused by unreasonable recursive algorithm design, and whether to use bitmap method or linked list method when discussing system architecture at the beginning, but these problems were finally solved by us. |
| **Comment of Advisor and Grade** |
| Comment：  Grade：           Sign of Advisor：                                                 Date： |