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 i-node usage.  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  **[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 operation**  **2.3.4 Data and indirect address blocks reading and writing** |
| **Summary** |
|  |
| **Comment of Advisor and Grade** |
| Comment：  Grade：           Sign of Advisor：                                                 Date： |