Project 1 Report
Name: Fanyue Zhu Student id: 522031910547

1 Introduction

The aim of this project is to design and implement a disk-like secondary storage server and a base filesystem as a client, with the secondary storage server providing the disk services and the base filesystem using those services. A special feature of the project is that the communication mechanism between the two is implemented through a socket API.
In the first part of the project, we will design and implement a basic disk-based secondary storage server. The main function of this server is to provide data storage and retrieval services, similar to the hard disc storage of a computer.
In the second part of the project, we will design and implement a basic file system as a client, which will use the disk services provided by the above server. The design of this file system will follow some basic file system principles and will be able to perform operations such as file creation, reading, writing and deletion.
In addition, we will study and learn to use the sockets API, which will be used to provide a communication mechanism that will enable client processes to communicate with the file system, and the file system to communicate with the disk storage server.
Overall, our goal is to gain an in-depth understanding of the design and implementation of storage servers and the use of socket APIs through this project, so that we can master the basic working principles of file systems and data storage techniques.

2 Step1: Design a basic disk storage system

2.1 Objectives

1. Implement an Internet-domain socket server, a simulation of a physical disk. The simulated disk is organized by cylinder and sector.
2. Include the seek time in the simulation to account for track-to-track time.
3. Store the actual data in a real disk file.
4. Process incoming client data and protocols.

2.2 Implementation

2.2.1 Disk Server

1. Create or open the 'disk' file, stretch it and mmap it.
2. Protocols:
   (a) **I**: Send the parameters to client.
   (b) **R c s**: Read from the designated position and send the data to client.
   (c) **W c s data**: Write the data to specified position.

2.2.2 Disk Client

Connect to the server using socket, and print the info received from it.

3 Step2: Design a basic file system

3.1 Objectives

1. Implement an inode file system. The file system should provide operations including: initialize the file system, create a file, read data from a file, write the given data to a file, append data to a file, remove a file, create directories
2. Free space management, involving maintaining a list of free blocks available on the disk.
3. The file system needs to support files of at least 16 KB in size, and filenames must be at least 8 Bytes long.
4. Implement this file system server as another UNIX-domain socket server. On one hand, this program will be a server for one UNIX-domain socket; on the other hand, it will be a client to the disk server UNIX-domain socket from the previous parts, as Figure 1 shown below.

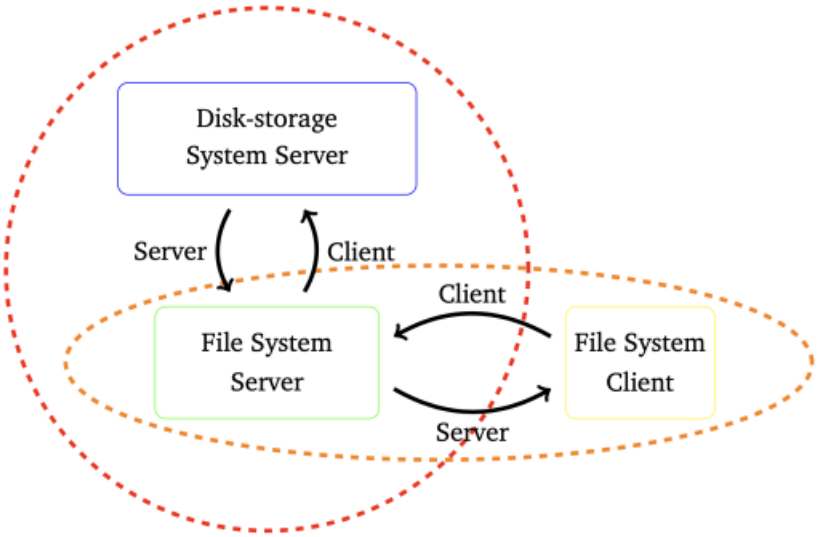

Figure 1: The File System Architecture

3.2 Implementation

3.2.1 Structure

I designed an Inode file system borrowing the Unix system. First several blocks serve as the system info blocks, but it's not actually used in this step. The following 256 blocks store the Inode table, and the rest are the data blocks.The system storage architecture is as Figure 2 shown below.
My inode has 64 bytes, containing its type, creation and modification time, size, linkscount, parent and current available status, along with 8 direct blocks and 1 indirect block. Its structure is shown as Figure 3 below.

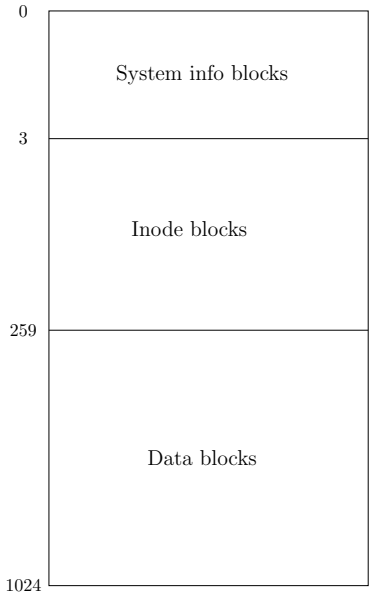

Figure 2: System Storage Architecture

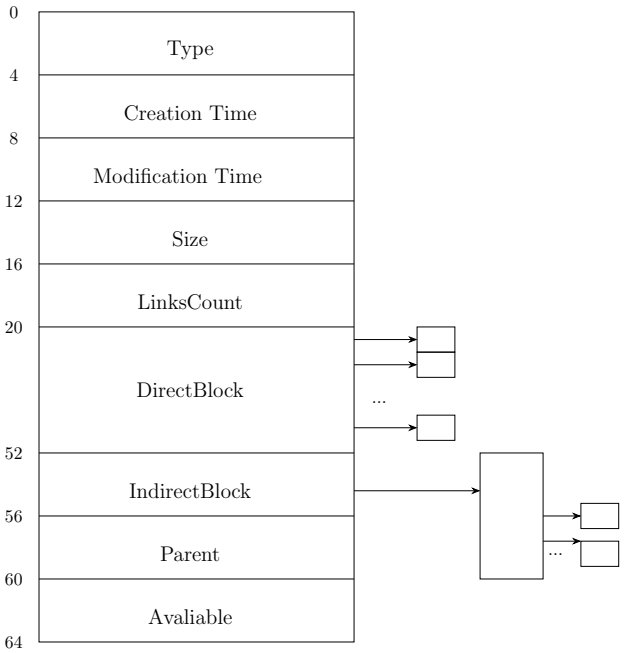

Figure 3: Inode Structure

3.2.2 Functions

1. Inode operations:

   a) `Inode_init()`: Initialize a new allocated inode.

   b) `Inode_format()`: Format the InodeTable.

   c) `Inode_alloc()`: Alloc an available inode.

   d) `Inode_write()`: Write the Inode info into blocks.

   e) `Inode_free()`: Free an Inode.

   f) `Inode_add()`: Add a file or directory to current path.

   g) `Inode_removeFile()`: Remove file from current path.

   h) `Inode_removeDir()`: Remove directory from current path, only if the directory is empty.

   i) `Inode_listFiles()`: List all files and directories of current path.

   j) `Inode_search()`: Search a specified file or directory in this path.

   k) `Inode_write_file()`: Write data into file.

   l) `Inode_read()`: Read data from file.

   m) `Inode_delete()`: Delete data from file.

2. Block operations:

   a) `Block_format()`: Format the BlockTable.

   b) `Block_read()`: Read data from disk block.

   c) `Block_write()`: Write data into disk block.

   d) `Block_alloc()`: Alloc an available block.

   e) `Block_free()`: Free a block.

3. Directory items:

   For a directory inode, only its direct block is effective. Define a new type named Directory_item to store the content its contains.

```
typedef struct Directory_item
{
    char Name[MAX_NAME];
    int Inode;
    uint8_t Available;
    uint8_t Type;
} Directory;//32bytes

```

   Thus, a directory inode can link to 64 files or directories at most.

4. Protocols:

   (a) **f**: Call `Block_format()` and `Inode_format`, then set current path root.

   (b) **mk f**: Call `Inode_add()` with the type parameter set as file(=0).

   (c) **mkdir d**: Call `Inode_add()` with the type parameter set as directory(=1).

   (d) **rm f**: Call `Inode_removeFile()`.

(e) **cd path**: Get the path using `strtok()`, then search it in the current directory and change current inode.

(f) **rmdir d**: Call`Inode_removeDir()`.

(g) **ls**: Call `Inode_listFiles()`.

(h) **cat f**: Call `Inode_read()` and read all data from the file.

(i) **w f l data**: Call `Inode_write_file()` and overwrite the data from 0^{th} pos.

(j) **i f pos l data**: Read the data from pos^{th} to the end to buffer and delete them in file. Then write the inserted data to the end of the file, then the buffer back.

(k) **d f pos l**: Read the data from $(pos + l)^{th}$ to the end to buffer. Then delete data from pos^{th} , and write the buffer back to the end of file.

4 Step3: Support multiple users in the file system

4.1 Objectives

1. Implement a multi-user system.
2. Design a data structure to store the user information, which should be stored in the file system, instead of memory.
3. Provide approaches and interfaces to log into your file system, create new users, delete users, etc.
4. Design a file access protocol to control file read/write permissions.

4.2 Implementation

4.2.1 Structure

On the basis of step 2, I modified the system storage architecture, using the first two blocks for user information, and the left one to store system info, such as whether it's be initialized and formatted. The modified architecture is shown as Figure 4 below.
The inode structure was also updated, changing the creation and modification time to its owner and permission to different users while let its size unchanged. The updated structure is shown as Figure 5.

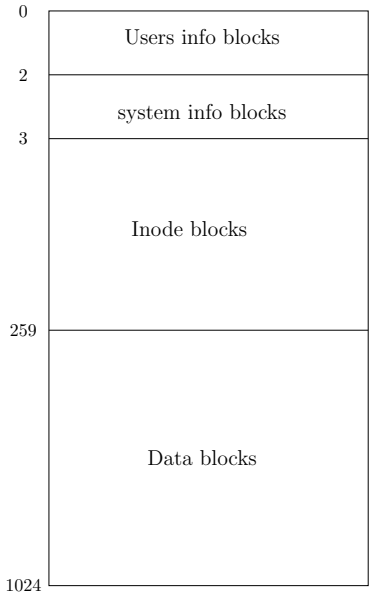

Figure 4: Updated System Storage Architecture

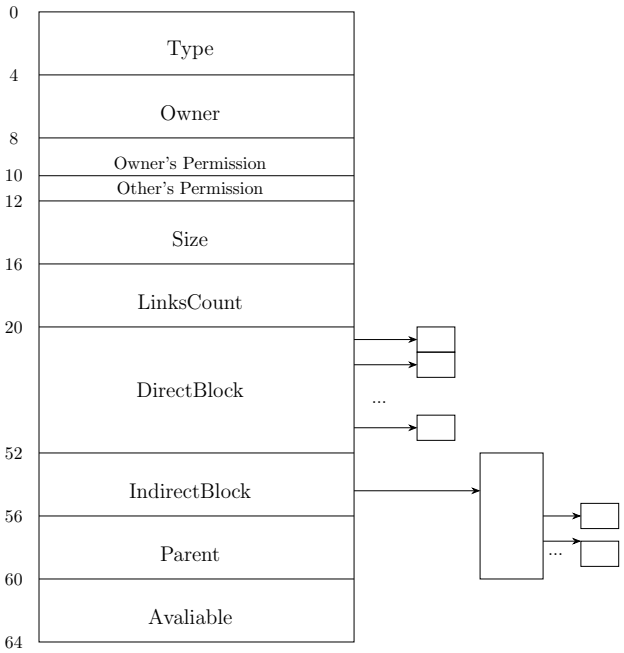

Figure 5: Updated Inode Structure

4.2.2 Functions

1. Modified the operations of Inode consedering user info, such as access to files and so on.
2. Define a new type called Users which stores the users' info.

```
typedef struct User
{
    char username[8];
    char password[16];
    uint16_t uid;
    uint16_t inode;
    uint16_t current_inode;
    uint16_t last_login;
} Users;//32bytes

```

3. Users operations:
   a) `User_login()`: Detect whether the incoming username and password can be matched with existing users.
   b) `User_create()`: Create a new user.
   c) `User_format()`: Format UserTable.
   d) `User_write()`: Write user info into blocks.
4. Interfaces: In file system client, detect whether there's a user logged in. If so, continue its command collect and transfer. If not, print the leading word and ask the user to log in or sign up.
5. File access permission: When a new file or directory is created, its permission for owner is write and read, while for other is read only. **root** has the supreme authority and is able to access every file in the system.
6. New command added:
   (a) **chmod f owner/other mode**: Check the permission. Only its owner or write and read users is able to change its mode. Then set the access mode as parameter.
   (b) **l username password**: Call `User_login()`.
   (c) **s username password**: Call `User_signup()`.
   (d) **logout**: Current user set as root.

5 Step4: Optional extensions

I've tried to implement multi client and solve the read and write conflict using mutex lock, but due to time limit, I didn't have time to finish this part.

6 Acknowledge

Thanks to teacher and Ta for their altruistic help. And thanks to my friend who assist me solving problem.