

1. Introduction

Background/ Target

The goal of this project is to design a medical information management system with client-server separation based on the I-Node file system.

Functions Implemented

The development of this project mainly includes three parts: I-Node file system development, medical information management system interface development, and client-server development based on SOCKCT.

The functions implemented in the I-Node file system include:

- Creating/deleting/displaying directories
- Creating/deleting/reading/writing/appending files
- Persistent storage of file systems
- Storage/use/list of available backup files
- Open/use/list of available snapshots of file snapshots
- Changing file permissions
- Changing file owners
- Changing file user groups

The medical information management system interface includes:

- User login
- Creating/reading/writing/deleting/displaying patient records

- Doctors publishing/cancelling/reading/displaying appointments
- Patients making/deleting/viewing available appointments
- Administrators create users/delete users/change user passwords

SOCKCT's client-server functions include:

- Using sockets to receive/send information on the client and server
- Calling the medical information management system interface
- Calling the file system interface
- Create threads to meet the needs of multiple clients accessing at the same time
- Using semaphores to complete mutual exclusion/synchronization operations to ensure the correctness of the file system (reader-writer problem)

Additional technical details:

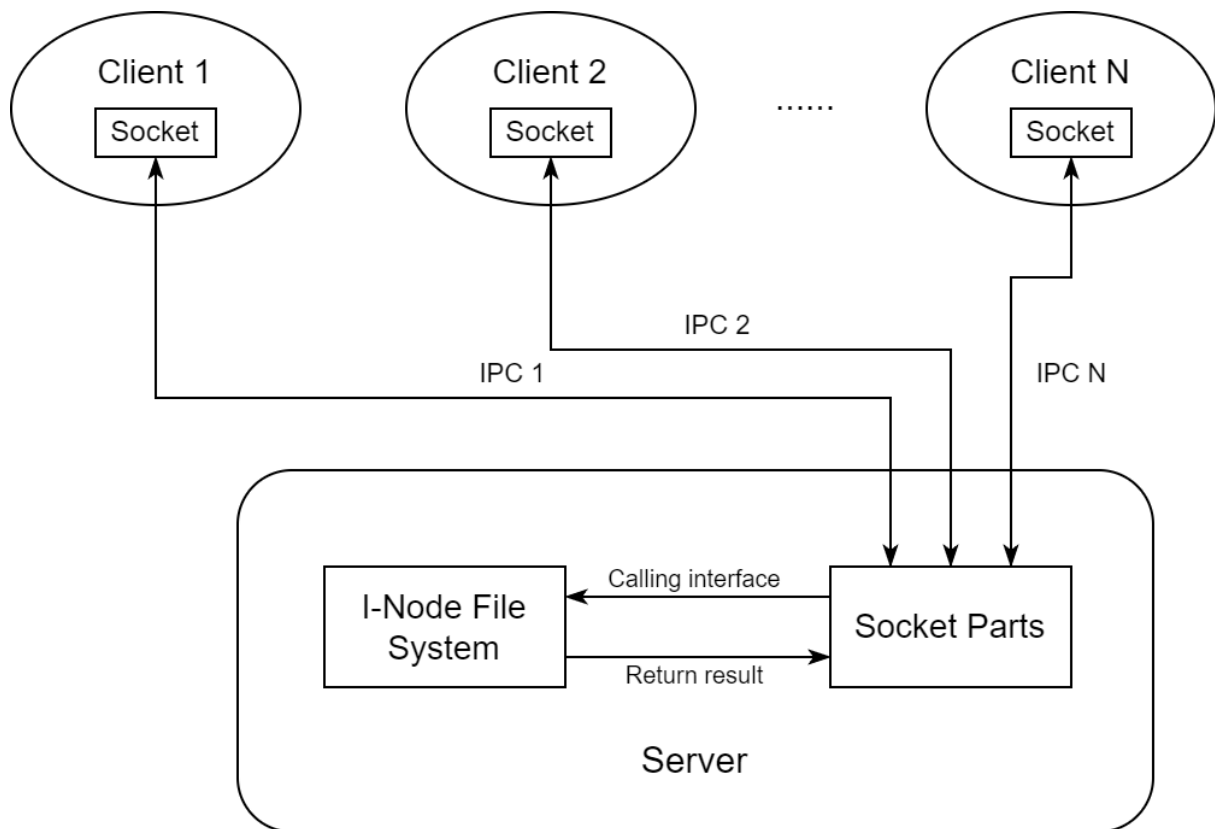
- Using indirect pointers to manage I-Node Limitations

2. Implementation Details

The project can be divided into three main implementation parts:

1. I-Node file system implementation
2. Medical data management system interface implementation
3. Server-client implementation

The architecture diagram of the entire system is as follows.



2.1 I-Node file system implementation

The implementation of the I-Node file system is encapsulated in `filesystem.h` and `filesystem.cpp`. Next, I will explain the specific implementation of the file system for this task.

2.1.1 Core data structures

The I-Node file system as a whole consists of two core data structures:

(1) I-Node (index node): describes the metadata of the file/directory

```
// 该文件系统单个文件的directBlock有4个，最多支持12个singleIndirectBlock。
// 单个文件最多16个blocks，大小为16*1023byte。
struct Inode {
    int blockIDs[BLOCKS_PER_INODE];    //block编号
    int singleIndirectBlock = -1;      // 单级间接块指针(指向一个块用于存储额外的块地址)
    int blockNum;                      //所拥有块数量
    int inodeID;                       // Inode 编号
    int fileType;                      // 文件类型（普通文件1/目录0），-1的时候表示空闲可用
    unsigned int fileSize;             // 文件大小32位
    time_t createTime;                // 文件创建时间
    time_t modifiedTime;              // 文件最后修改时间
    char permission[2] = { 'N', 'R' }; // [0]是一般用户（除创建者和用户组）权限（一般为N），[1]是用户组的权限（eg,W,R,N）
    char creator[MAX_LENGTH_USERNAME]; //创建者名字
    int groupNum;                      //用户组内数量
    char group[MAX_GROUP_NUM][MAX_LENGTH_USERNAME] = {}; //用户组（'lihua','zhangsan'）

    //文件快照
    bool FS = 0; //是否开启
    char FSoperation[MAX_FS] = { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' }; // W/A(WA)
    char FSsize[MAX_FS][64];
};
```

- `blockIDs[BLOCKS PER INODE]:`

Stores the block number where the data is actually stored, up to 4 direct blocks.

- `singleIndirectBlock`:

Block ID of a single-level indirect block; if 4 direct blocks are not enough, additional data block numbers can be stored in this indirect block.

- fileType:

distinguish between ordinary files and directories.

- permission:

simple permission control, for ordinary users and user groups.

- Snapshot function (FS, FSoperation, FStime):

Whether to enable snapshots, and snapshot operation records and

timestamps.

(2) Block (data block): stores the file or directory content

I. Directory Block

```
struct DirectoryBlock {
    char fileName[ENTRY_NUMBER][MAX_LENGTH_FILENAME]; //32*28
    int inodeID[ENTRY_NUMBER]; //32字节*4

    // 构造函数
    DirectoryBlock() {
        // 初始化 fileName 为 "nan" 且 inodeID 为 -1
        for (int i = 0; i < ENTRY_NUMBER; ++i) {
            strcpy_s(fileName[i], "nan"); // 用 "nan" 填充 fileName
            inodeID[i] = -1; // 将 inodeID 设置为 -1
        }
    }
};
```

For directory-type inodes, their storage content is placed in the corresponding DirectoryBlock. A directory block can store multiple entries, each of which stores:

- fileName[i]: the name of the subfile/subdirectory;
- inodeID[i]: the inode number of the corresponding subfile or

subdirectory;

When accessing the path "/root/code", the inode corresponding to the root will be found first, and the block storing the DirectoryBlock will be found through its blockIDs, and then the entry named "code" will be traversed in the DirectoryBlock, and the corresponding inodeID will be obtained, and the next layer will be moved down.

II. File Block

```
struct FileBlock {
    char content[BLOCK_SIZE];
    // 一个文件中最多可以写1023个字符，最后写终止符
};
```

The actual data of a normal file is stored in these blocks.

It is worth noting that the size of both file blocks and directory blocks is 1024 bytes, which ensures that the file system can assign arbitrary block numbers to these two blocks.

Since blockMem in this code is declared as DirectoryBlock, if we want to read and write text, we can simply reinterpret_cast or otherwise force this block to be FileBlock to handle its content.

(3) Bitmap

```
char inodeBitmap[BLOCK_NUMBER / 8]; // inode的使用位图
char blockBitmap[BLOCK_NUMBER / 8]; // block的使用位图
```

In order to quickly find free inodes or blocks, a bitmap is used for management:

- inodeBitmap[i] == 1 means that the i-th inode is occupied.
- blockBitmap[i] == 1 means that the i-th block is occupied.

When allocating, the first 0 position will be found and set to 1; when releasing, it will be set back to 0.

2.1.2 Core data

```
//数据成员
Inode inodeMem[INODE_NUMBER]; // 存储所有的inode
DirectoryBlock blockMem[BLOCK_NUMBER]; // 存储所有block,

#define INODE_NUMBER 1024 //一共1024个inode
#define BLOCK_NUMBER 1024*4 //一共1024*4个块
```

The core data of the file system is stored in two arrays, one is the inode array and the other is the block array. There are a total of 1024 allocatable inodes and 4096 allocatable blocks.

2.1.3 Core function

(1) findInodeByPath

```
Inode* fileSystem::findInodeByPath(const char* path) {
    // 将路径拆分为多个部分
    std::vector<std::string> pathParts = splitPath(path);

    Inode* currentInode = &inodeMem[0]; // 从根目录开始, 根目录的 inode 是 0

    for (const auto& part : pathParts) {
        bool found = false;

        // 查找当前目录中的文件/子目录

        DirectoryBlock* currentDirBlock = reinterpret_cast<DirectoryBlock*>(&blockMem[currentInode->blockIDs[0]]);

        // 遍历当前目录块中的所有文件名
        for (int i = 0; i < ENTRY_NUMBER; ++i) {
            if (currentDirBlock->inodeID[i] != -1 && strcmp(currentDirBlock->fileName[i], part.c_str()) == 0) {
                // 找到匹配的文件/目录
                currentInode = &inodeMem[currentDirBlock->inodeID[i]];
                found = true;
                break;
            }
        }

        if (!found) {
            return nullptr; // 如果没有找到路径部分, 返回空
        }
    }

    return currentInode; // 返回找到的 inode
}
```

First, split the input path (e.g. "/root/folder/file") into a string array

(e.g. ["root", "folder", "file"]) according to the path separator.

Start from the root node (inodeID=0) and treat it as the inode of the current directory.

For each path part, traverse all subitems in the DirectoryBlock of the current directory, compare the file names, and update the current inode if a match is found; if not, return null (indicating that a part of the path does not exist).

(2) createFile/ createDirectory

```
bool FileSystem::createFile(const char* path, const char* user) {
    // 1. 确保路径有效
    if (path == nullptr || strlen(path) == 0) {
        return false; // 路径不能为空
    }
    // 2. 获取路径的各个部分
    std::vector<std::string> pathParts = splitPath(path);
    if (pathParts.empty()) {
        return false; // 空路径
    }
    // 3. 获取父路径 (除了最后一个部分)
    std::string parentPath = "";
    for (size_t i = 0; i < pathParts.size() - 1; ++i) {
        parentPath += "/" + pathParts[i]; // 将除最后一个元素外的路径拼接成父路径
    }
    // 4. 获取新文件的名字 (最后一个部分)
    std::string newFileName = pathParts.back();
    // 5. 查找父目录的 inode
    Inode* parentInode = findInodeByPath(parentPath.c_str());
    if (!parentInode) {
        return false; // 父目录不存在
    }
    // 6. 确保父目录是有效目录
    if (parentInode->fileType != 0) {
        return false; // 父 inode 不是目录
    }
    // 7. 检查是否已有同名的文件
    DirectoryBlock* parentDirBlock = reinterpret_cast<DirectoryBlock*>(&blockMem[parentInode->blockIDs[0]]);
    for (int i = 0; i < ENTRY_NUMBER; ++i) {
        if (parentDirBlock->inodeID[i] != -1 && strcmp(parentDirBlock->fileName[i],
            newFileName.c_str()) == 0) {
            return false; // 文件名已存在
        }
    }
    // 8. 在父目录中创建新文件
    bool fileCreated = false;
    for (int i = 0; i < ENTRY_NUMBER; ++i) {
        if (parentDirBlock->inodeID[i] == -1) { // 找到一个空位置
            // 调用 allocateInode 来分配一个新的 inode
            int newInodeID = allocateInode();
            if (newInodeID == -1) {
                return false; // 没有空闲的 inode
            }

            // 创建新文件的 inode 信息
            Inode* newInode = &inodeMem[newInodeID];
            newInode->fileType = 1; // 文件类型设置为 1
            newInode->fileSize = 0;
            newInode->createdTime = time(nullptr);
            newInode->modifiedTime = time(nullptr);
            strcpy_s(newInode->creator, user); // 设置创建者为传入的用户
            newInode->permission[0] = N; // 设置其他用户无权限
            newInode->permission[1] = W; // 设置用户组权限为读写
            newInode->blockNum = 4;

            // 为新文件分配四个块并清空它们
            for (int j = 0; j < 4; ++j) {
                int newBlockID = allocateBlock();
                if (newBlockID == -1) {
                    // 如果无法分配足够的块, 需要回收已分配的资源
                    for (int k = 0; k < j; ++k) { // 回收之前已分配的块
                        freeBlock(newInode->blockIDs[k]);
                        newInode->blockIDs[k] = -1;
                    }
                    freeInode(newInodeID); // 回收inode
                    return false; // 返回失败
                }
                newInode->blockIDs[j] = newBlockID; // 将新分配的块ID保存到inode
                FileBlock* newBlock = reinterpret_cast<FileBlock*>(&blockMem[newBlockID]);
                memset(newBlock->content, 0, BLOCK_SIZE); // 清空新分配的块
            }

            // 在父目录中添加新文件
            parentDirBlock->inodeID[i] = newInodeID;
            strcpy_s(parentDirBlock->fileName[i], newFileName.c_str());

            fileCreated = true;
            break;
        }
    }

    if (!fileCreated) {
        return false; // 父目录没有空余位置
    }
    return true;
}
```


After receiving the file path and user, this function will first check whether the path is legal and resolve it into the parent directory path and the new file name.

Then, it will find the Inode of the parent directory through `findInodeByPath` and confirm that it is a directory and that there is no file with the same name. If the conditions are met, it will find a free entry in the directory and allocate a new Inode.

It will call `allocateInode()` to obtain an available index node and set the file meta information (such as type, permissions, creation time, etc.).

Then, it will try to allocate data blocks according to a fixed number (4 in this case). If the allocation fails, it will release the allocated resources and return false. If successful, it will update the Inode ID and file name of the new file to the directory block of the parent directory. Finally, it will return true to indicate successful creation.

Note: `createDirectory()` function implementation is basically the same, except that the created block is `DirectoryBlock`, which is used to store subsequent directories.

(2) deleteFile/deleteDirectory

```
bool fileSystem::deleteFile(const char* path, const char* user) {
    // 1. 确保路径有效
    if (path == nullptr || strlen(path) == 0) {
        return false; // 路径不能为空
    }
    // 2. 查找要删除的文件的 inode
    Inode* targetInode = findInodeByPath(path);
    if (!targetInode) {
        return false; // 文件不存在
    }
    // 3. 确保目标是一个文件
    if (targetInode->fileType != 1) {
        return false; // 目标不是一个文件
    }
    // 4. 检查权限
    if (strcmp(targetInode->creator, user) != 0) {
        return false; // 用户无权删除该文件
    }
    // 5. 查找并更新父目录的 DirectoryBlock
    std::vector<std::string> pathParts = splitPath(path);
    std::string parentPath = "";
    for (size_t i = 0; i < pathParts.size() - 1; ++i) {
        parentPath += "/" + pathParts[i];
    }
    std::string targetFileName = pathParts.back();

    Inode* parentInode = findInodeByPath(parentPath.c_str());
    if (!parentInode) {
        return false; // 父目录不存在
    }
    DirectoryBlock* parentDirBlock = reinterpret_cast<DirectoryBlock*>(&blockMem[parentInode->blockIDs[0]]);
    for (int i = 0; i < ENTRY_NUMBER; ++i) {
        if (strcmp(parentDirBlock->fileName[i], targetFileName.c_str()) == 0) {
            parentDirBlock->inodeID[i] = -1;
            strcpy_s(parentDirBlock->fileName[i], "nan"); // 标记为"nan"或其他空标记
            break;
        }
    }
    // 6. 释放 inode 和相关的块
    // 释放直接分配的块
    for (int i = 0; i < 4; ++i) { // 每个文件最多分配了4个直接块
        if (targetInode->blockIDs[i] != -1) {
            freeBlock(targetInode->blockIDs[i]);
            targetInode->blockIDs[i] = -1;
        }
    }
    // 如果存在单级间接块, 释放其中的块
    if (targetInode->singleIndirectBlock != -1) {
        FileBlock* indirectBlock = reinterpret_cast<FileBlock*>(&blockMem[targetInode->singleIndirectBlock]);
        // 释放间接块中存储的每个块ID
        for (int i = 0; i < (targetInode->blockNum - 4); ++i) {
            int blockID;
            memcpy(&blockID, &indirectBlock->content[i * sizeof(int)], sizeof(int));
            if (blockID != -1) {
                freeBlock(blockID); // 释放间接块指向的块
            }
        }
        // 释放间接块本身
        freeBlock(targetInode->singleIndirectBlock);
        targetInode->singleIndirectBlock = -1;
    }
    // 释放 inode
    freeInode(targetInode->inodeID);
    return true;
}
```

After receiving the file path and user, this function first verifies the legitimacy of the path and obtains the Inode of the target file through `findInodeByPath()` to confirm that it is indeed a file and was created by the current user; it then obtains the Inode of the parent directory by parsing the path, clears the entry corresponding to the file in the parent directory block (sets `inodeID[i] = -1`, and the file name is "nan"), then releases all blocks of the file (including four direct blocks and possible single-level indirect blocks) and calls `freeInode` to reclaim the Inode used by the file, and finally returns `true` to indicate that the file was successfully deleted.

Note: `deleteDirectory()` function implementation is basically the same, except that the delete block is `DirectoryBlock`, which is used to store subsequent directories.

(3)writeFile

Pseudocode:

function writeFile(path, user, context, flag):

1. Validate the path (non-null and not empty).
2. Locate the file's inode via findInodeByPath(path).
3. Check if user has write permission or is admin.
4. Calculate total blocks (direct + possibly indirect):
 - Collect block IDs from fileInode.blockIDs (up to 4).
 - If singleIndirectBlock exists, load additional block IDs.
5. If new data size exceeds current capacity:
 - Determine how many extra blocks are needed (expansion).
 - If fileInode.singleIndirectBlock is -1, allocate it.
 - Add newly allocated blocks into singleIndirectBlock.
6. Clear content of all involved blocks (memset to 0).
7. Write data from context to these blocks and then update information in fileInode.
8. If file snapshots (FS) are enabled and flag == 1:
 - Record operation/time in FS arrays (FSoperation, FSime).
 - Create snapshot file under "/FS/filename/timestamp.txt".
 - Recursively write the same content into the snapshot file.
9. Return true if successful.

This function first ensures that the target path and the amount of data to be written are legal, then locates the file's Inode and verifies the write permission; if expansion is required, new blocks are allocated and single-level indirect blocks are updated or created, and finally all used blocks are cleared and data is written block by block to update the file's meta information; if snapshots are enabled for the file and the current write operation is marked as normal write (flag=1), a snapshot is created at the "/FS/target file name/timestamp.txt" location and all the same contents are written.

(4) writeAppendFile

Pseudocode:

function writeAppendFile(path, user, context):

1. Check if path is valid (non-null, non-empty).
2. Find the file's inode via findInodeByPath(path).
 - If fileInode is null => return false.
 - If fileInode.fileType != 1 => return false (not a regular file).
3. Check write permissions:
 - Either the file creator == user
 - Or fileInode.permission[1] == 'W'
 - Or user is admin
 - Otherwise => return false
4. If file snapshots (fileInode.FS == 1):
 - Generate timestamp = getCurrentDateTime()
 - Insert a record (time, 'A') in fileInode.FS time / FSoperation
 - If no free snapshot slot => log message and continue
 - Construct snapshot file path "/FS/filename/timestamp.txt"
 - createFile(snapshotFilePath, user)
 - writeFile(snapshotFilePath, user, context, 1) => incremental content
5. existingContent = readFile(path, user) // Read original file content
6. newContent = existingContent + context // Append
7. return writeFile(path, user, newContent, 0) // Re-write with updated content

This function will first verify whether the path and file exist, whether the file type is correct, and whether the user has write permission; if the snapshot function is enabled for the file, it will first create a snapshot file with a timestamp for the newly added content, then read the original file content and append the new content to the end, and finally call writeFile to write the merged complete content back to the target file, thereby realizing additional writing to the file.

Note: Under the file snapshot function, a flag is introduced to provide a convenient interface for the subsequent writeAppendFile.

(5) readFile

Pseudocode:

function readFile(path, user):

1. Check if path is valid (non-null, non-empty).
 - If path is null or empty => log "Error: Path is invalid." and return "".
2. Find the file's inode via findInodeByPath(path).
 - If fileInode is null => log "Error: File does not exist." and return "".
 - If fileInode.fileSize == 0 => return "" (file is empty).
 - If fileInode.fileType != 1 => log "Error" and return "".
3. Check read permissions:
 - If fileInode.creator == user OR user is admin OR user is in the file's group and fileInode.permission[1] != 'N'.
 - If not => log "Error" and return "".
4. Collect block IDs:
 - Initialize blockIDs[16] with -1.
 - Collect up to 4 direct block IDs from fileInode.blockIDs.
 - If fileInode.singleIndirectBlock exists:
 - Load IDs from the single indirect block into blockIDs.
5. Read data from blocks:
 - Initialize content as an empty string.
 - For each valid blockID in blockIDs:
 - Skip if blockID == -1.
 - Read block content and append to content.
6. Return content:
 - Return the concatenated string from all blocks.

Based on the passed file path and user identity, this function first confirms whether the path is valid, whether the target file Inode can be located, and whether the file type and permissions meet the reading requirements. Then, by collecting the block numbers of all direct blocks and single-level indirect blocks, the corresponding block contents are read sequentially and assembled into a string stream, and finally the complete file contents are returned.

(6) useFileSnapshots

Pseudocode:

function useFileSnapshots(filePath, time, operatorName):

1. Find the file's inode via findInodeByPath(filePath).
 - If fileInode is null: log "Error" and return false.
2. Verify operator permissions:
 - If operatorName is admin OR fileInode.creator == operatorName:
3. Extract the target file name from filePath:
 - Split filePath into pathParts.
 - targetFileName = last element of pathParts.
4. Locate the snapshot corresponding to time:
 - Initialize op = -1.
 - For i in range(0, MAX_FS):
 - If fileInode.FStime[i] == time:
 - Set op = i.
 - Break.
 - If op == -1:
 - log "Error" and return false.
5. Find the most recent 'W' operation before or at op:
 - Initialize newW = -1.
 - For j in range(op, -1, -1):
 - If fileInode.FSoperation[j] == 'W':
 - Set newW = j.
 - Break.
6. Accumulate content from snapshots between newW and op:
 - Initialize context = "".
 - For z in range(newW, op + 1):
 - time_ = fileInode.FStime[z].
 - Construct snapshotFilePath
 - Append readFile(snapshotFilePath, operatorName) to context.
7. Write accumulated content to the original file:
 - Return writeFile(filePath, operatorName, context, 0).

This function finds the target timestamp and the most recent complete write operation (W) through the snapshot record, then gradually reads all incremental content from the write operation to the target timestamp, splices them into complete data, and writes these contents back to the source file.

(7) saveBackUp/loadBackUp

```
bool FileSystem::saveBackUp() {
    // 获取当前时间戳作为备份文件名
    auto now = std::chrono::system_clock::now();
    auto time_t_now = std::chrono::system_clock::to_time_t(now);
    std::string backupFileName = "backup/" + std::to_string(time_t_now) + ".dat";

    // 打开文件并保存文件系统
    std::ofstream backupFile(backupFileName, std::ios::binary);
    if (!backupFile.is_open()) {
        std::cerr << "Failed to open backup file for writing." << std::endl;
        return false;
    }

    // 保存 inode 数据
    backupFile.write(reinterpret_cast<char*>(inodeMem), sizeof(inodeMem));
    backupFile.write(inodeBitmap, sizeof(inodeBitmap));
    backupFile.write(blockBitmap, sizeof(blockBitmap));
    backupFile.write(reinterpret_cast<char*>(blockMem), sizeof(blockMem));

    backupFile.close();
    std::cout << "Backup saved to " << backupFileName << std::endl;

    // 更新备份文件名列表
    updateBackupFileList();

    return true;
}

bool FileSystem::loadBackUp(const std::string& filePath) {
    std::ifstream backupFile(filePath, std::ios::binary);
    if (!backupFile.is_open()) {
        std::cerr << "Failed to open backup file for loading." << std::endl;
        return false;
    }

    // 加载 inode 数据
    backupFile.read(reinterpret_cast<char*>(inodeMem), sizeof(inodeMem));
    backupFile.read(inodeBitmap, sizeof(inodeBitmap));
    backupFile.read(blockBitmap, sizeof(blockBitmap));
    backupFile.read(reinterpret_cast<char*>(blockMem), sizeof(blockMem));

    backupFile.close();
    std::cout << "File system loaded from " << filePath << std::endl;

    return true;
}
```

saveBackUp function generates the current timestamp as the backup file name, opens the backup file, saves the key data of the file system (`inodeMem`, `inodeBitmap`, `blockBitmap` and `blockMem`) in binary form to the file, closes the file and updates the backup file name list after completion, and finally returns `true` to indicate that the backup is successful. If the file cannot be opened, it outputs an error message and returns `false`.

Note: loadBackUp is very similar.

(8) displayDirectory

```
std::string fileSystem::displayDirectory(const char* path) {  
    // 1. 查找路径对应的 Inode  
    Inode* inode = findInodeByPath(path);  
    if (!inode) {  
        return "Directory not found.\n"; // 路径不存在  
    }  
  
    // 2. 确保路径是一个目录  
    if (inode->fileType != 0) {  
        return "The specified path is not a directory.\n";  
    }  
  
    // 3. 获取目录块  
    DirectoryBlock* dirBlock = reinterpret_cast<DirectoryBlock*>(&blockMem[inode->blockIDs[0]]);  
  
    // 4. 构建目录内容字符串  
    std::string directoryListing = "Directory contents of " + std::string(path) + ":\n";  
    for (int i = 0; i < ENTRY_NUMBER; ++i) {  
        if (dirBlock->inodeID[i] != -1) { // 如果该目录项存在  
            directoryListing += "- " + std::string(dirBlock->fileName[i]) + "\n";  
        }  
    }  
  
    return directoryListing;  
}
```

This function searches for the corresponding Inode by path, verifies whether the path exists and is a directory, then reads the directory block content, adds all valid file names in the directory to a string one by one, and finally returns a list of the directory contents. If the path does not exist or is not a directory, a corresponding error message is returned.

(9) changeFilePermission/changeFileOwner/adjustUserGroup

Use the findInodeByPath function to find the target inode node and then modify the corresponding data.

2.2 Medical data management system interface implementation

2.2.1 File system architecture

```
-/  
  - user  
    - admin.txt  
    - doctor.txt  
    - patient.txt  
  - records  
    - patientID_1.txt  
    - patientID_2.txt  
    ...  
  - appointments  
    - doctor_id-time.txt  
  - FS  
    - filename  
      - time_1.txt
```

Based on the I-Node file system that has just been implemented, the upper-level medical information management system is designed as above.

(1) /user

There are three files in this directory, namely admin.txt, doctor.txt and patient.txt. These files contain the accounts and passwords of the corresponding types of users.

(2) /records

This directory stores the patient's record files. When a patient user is created, the system automatically creates a file named after their user name in this directory.

(3) /appointments

This directory stores the appointment files created by the doctor. The appointment file is named with the doctor's username plus the appointment time.

(4) /FS

The directory stores the file snapshots of the corresponding files. After the file snapshot function of a file is enabled, the system creates a subdirectory with the same name as the target file name in the /FS directory. The snapshots of the file will then be stored in this subdirectory and named after the time the snapshot was created.

2.2.2 Medical data management system interface function

(1) createUser/ deleteUser

Create a user: Find a blank line in the file of the corresponding target type in the /user directory, and then write it in the format of username + "," + password + "\n".

Delete user: Directly find the target user line by line from the file of the corresponding target type in the /user directory and delete the line.

(2) userLogin

Read directly from the file of the corresponding target type in the /user directory line by line. If the account and password match, it succeeds.

(3) changePassword

Directly find the target user line by line from the file of the

corresponding target type in the /user directory and write it in the format of username + "," + newpassword + "\n".

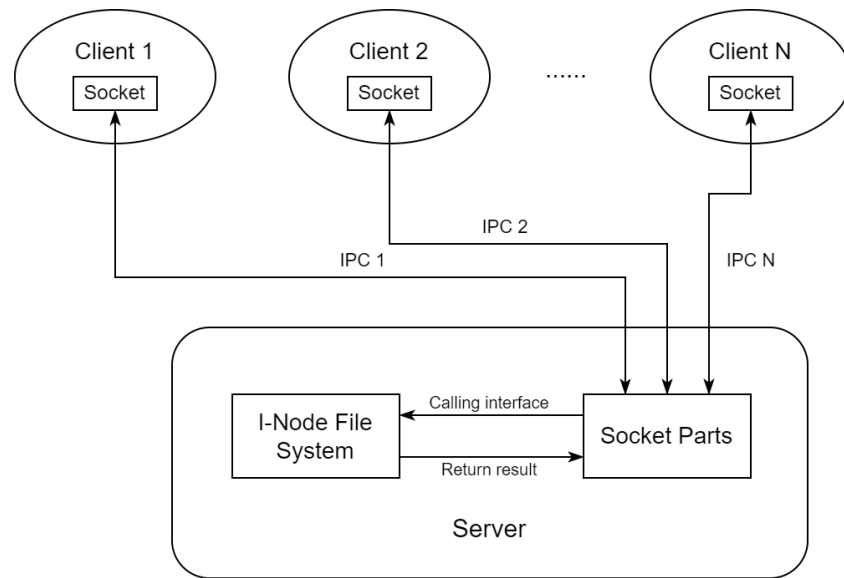
(4) creatRecords/deleteRecords/writeAppendRecords/readRecords
/listRecords

After finding the file path of the target record (/records/patient_1.txt) according to the system design, only need to simply call the corresponding file system interface designed in the previous article.

(5) releaseAppointments/revocationAppointments/writeAppointments
/deleteAppointments/readAppointments/listAppointments

After finding the file path of the target appointment (/appointments/doctor_id-time.txt) according to the system design, only need to simply call the corresponding file system interface designed in the previous article.

2.3 Server-client implementation



In the design of this task, the server maintains a file system instance and communicates with the client through a socket. The content of the communication is the requested operation and the parameters of the call.

To ensure the stability of the file system (reader-writer problem), semaphores are used to complete synchronization and mutual exclusion operations.

In this design, for every client connection, the server will construct a new thread to provide services.

So I will explain it from the following three aspects:

- IPC with Socket
- Reader-writer problem
- Creating threads

2.3.1 IPC with Socket

```
else if (command == "1. create Directory") {  
    reply = "plz enter the path";  
    send(clientSocket, reply.c_str(), reply.length(), 0);  
    bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);  
    std::string path(buffer, bytesRead);  
  
    if (fs.createDirectory(path.c_str(), username.c_str())) {  
        reply = "successfully create Directory: " + path;  
        send(clientSocket, reply.c_str(), reply.length(), 0);  
    }  
    else {  
        reply = "fail to create Directory: " + path;  
        send(clientSocket, reply.c_str(), reply.length(), 0);  
    }  
}
```

Server code

```
case 1:  
    command = "1. create Directory";  
    send(clientSocket, command.c_str(), command.length(), 0);  
  
    //接收 "plz enter the path"  
    bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);  
    if (bytesRead > 0)  
    {  
        std::cout << std::string(buffer, bytesRead) << std::endl;  
    }  
    cin >> path_1;  
    send(clientSocket, path_1.c_str(), path_1.length(), 0);  
  
    //接收 "successfully create Directory:"/"fail to create Directory:"  
    bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);  
    if (bytesRead > 0) {  
        std::cout << std::string(buffer, bytesRead) << std::endl;  
    }  
  
    break;
```

Client code

In the sample code above, the client sends the command "1. create Directory" to the server through the socket. After receiving it, the server returns the prompt "plz enter the path". The client then sends the directory path entered by the user. The server calls `fs.createDirectory` to perform the creation operation and returns the result (success or failure message) to the client through the socket, completing the communication and operation.

2.3.2 Reader-writer problem

In order to ensure the stability of the file system, file system operations should be divided into two categories, read and write, and restricted. Multiple processes can read content at the same time, but when writing, it is necessary to ensure that only one process is performing the operation.

Some read operation functions in this file system:

- readFile()
- displayDirectory()

Some write operation functions in this file system:

- writeFile()
- writeAppendFile()
- loadBackUp()

In order to solve this problem, semaphores are introduced.

```
//并发控制
std::mutex FS; // 用于文件系统操作的互斥锁
std::mutex RC; // 用于互斥访问
int rc = 0; //读者人数

//信号量，读者写者问题
void lockFS(); // 加锁
void unlockFS(); // 解锁

void lockRC(); // 加锁
void unlockRC(); // 解锁
```

Each time a write operation is called, code for requesting a write lock and a write unlock is added above and below it.

```
fs.lockFS();
```

```
writeOperation();
```

```
fs.unlockFS();
```

Example:

```
// 5. 导入备份
else if (command == "5. load backup") {

    reply = "plz enter backup file path";
    send(clientSocket, reply.c_str(), reply.length(), 0);
    bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);
    std::string backup_file_path(buffer, bytesRead);

    //此处后面需要加锁

    //写者请求锁
    fs.lockFS();

    if (fs.loadBackUp(backup_file_path)) {
        reply = "successfully load a backup:" + backup_file_path;
        send(clientSocket, reply.c_str(), reply.length(), 0);
    }
    else {
        reply = "fail to load a backup:" + backup_file_path;
        send(clientSocket, reply.c_str(), reply.length(), 0);
    }

    //写者归还锁
    fs.unlockFS();
}
```

Each time a read operation is called, code for requesting a read lock and a read unlock is added above and below it.

```
fs.lockRC();
```

```
if (fs.rc == 0) {
```

```
    fs.lockFS();
```

```
}
```

```
fs.rc++;
```

```
fs.unlockRC();
```

```
readOperation()
```

```
fs.lockRC();
```

```
fs.rc--;
```

```
if (fs.rc == 0) {
```

```
    fs.unlockFS();
```

```
}
```

```
fs.unlockRC();
```


Example:

```
// 4. 备份目录
else if (command == "4. list backup") {

    //此处后面需要加锁

    //读者获得锁
    fs.lockRC();
    if (fs.rc == 0) {
        fs.lockFS();
    }
    fs.rc++;
    fs.unlockRC();

    if (fs.listBackUp().empty() ){
        reply = "backup list is empty";
        send(clientSocket, reply.c_str(), reply.length(), 0);
    }
    else {
        reply = fs.listBackUp();
        send(clientSocket, reply.c_str(), reply.length(), 0);
    }

    //读者归还锁
    fs.lockRC();
    fs.rc--;
    if (fs.rc == 0) {
        fs.unlockFS();
    }
    fs.unlockRC();
}
```

2.2.3 Creating threads

The server code uses ``accept`` to wait for client connections. Whenever a client connection is successfully received, a new thread is immediately created, the ``handleClient`` function is called to process the client's request, and the thread object is stored in the ``threads`` container, so that each client connection corresponds to an independent thread for concurrent processing.



```
while (true) {
    socklen_t clientAddrSize = sizeof(clientAddr);
    clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr, &clientAddrSize);
    if (clientSocket == -1) {
        std::cerr << "Error accepting client connection" << std::endl;
        continue;
    }

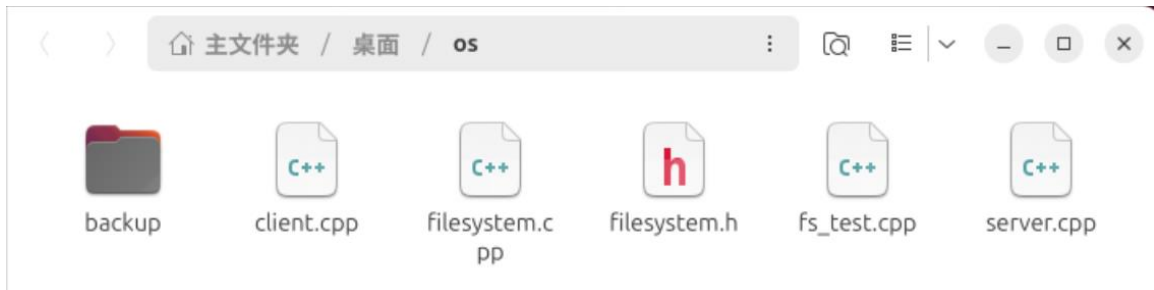
    std::cout << "Accepted a client connection" << std::endl;

    // a client -> a threads
    threads.push_back(std::thread(handleClient, clientSocket));
}
```

3. Execution Instruction

Source file directory:

- filesystem.h
- filesystem.cpp
- fs_test.cpp
- server.cpp
- client.cpp

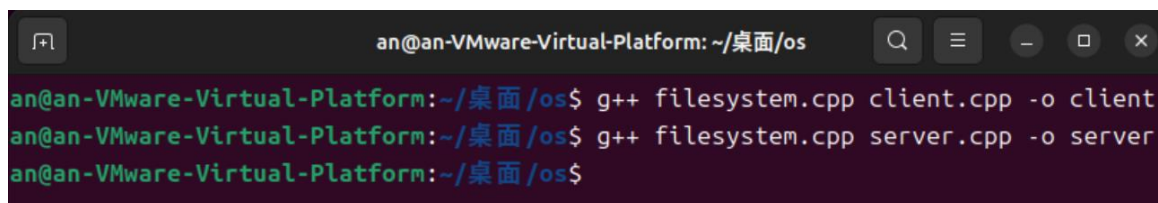


(1) Open the terminal in the source file directory

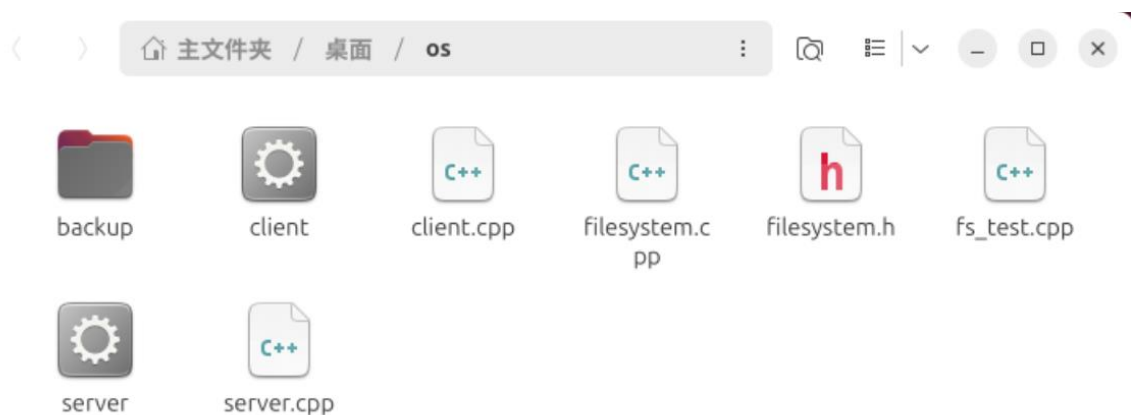
(2) Execute the code in sequence:

```
g++ filesystem.cpp client.cpp -o client
```

```
g++ filesystem.cpp server.cpp -o server
```



Get the compiled client and server executable files.



(3) Then execute in a terminal:

```
./server
```

Then enter the port number you want to use.

```
an@an-VMware-Virtual-Platform:~/桌面/os$ ./server
Error: Unable to open file for loading the file system.
system init the first time
Directory contents of /:
- /
- user
- records
- appointments
- FS

Enter a port number
8080
```

(4) Open one or more terminals and enter:

```
./client
```

Then enter the port number you use in the server.

```
an@an-VMware-Virtual-Platform:~/桌面/os$ ./client
Enter a port number
8080
```

(5) Perform management system or file system operations

```
Medical Data Management System
1. Patient log in
2. Doctor log in
3. Administrator log in
4. File System Mode
0. Exit System
```

4. Result Presentation

4.1 I-Node file system

(1) Press 4 to enter the file system operation, enter the user name

“admin” to use administrator privileges

```
Medical Data Management System
1. Patient log in
2. Doctor log in
3. Administrator log in
4. File System Mode
0. Exit System

4
plz enter user name
admin
```

Enter file system operations

```
welcome to file system mode
-----FileSystem-----
1. create Directory
2. delete Directory
3. display Directory
4. create File
5. write File
6. read File
7. delete File
8. change File Permission
9. change File Owner
10. adjust User Group
11. write appened File
12. enable File Snapshot
13. list File Snapshot
14. use File Snapshots
0. back to upper directory
-----
```

(2) Display directory

Type 3 and select the path "/" to view the root directory.

```
3
plz enter the path
/
Directory contents of /:
- /
- user
- records
- appointments
- FS
```

(3) Create directory

Press 1 to create the folder /test.

```
1
plz enter the path
/test
successfully create Directory: /test
```

Type 3 and select the path "/" to view the root directory.

```
3
plz enter the path
/
Directory contents of /:
- /
- user
- records
- appointments
- FS
- test
```

Create Success.

(4) Delete directory

Press 2 to delete the folder /test.

```
2
plz enter the path
/test
successfully delete Directory: /test
```

Type 3 and select the path "/" to view the root directory.

```
3
plz enter the path
/
Directory contents of /:
- /
- user
- records
- appointments
- FS
```

Delete Success.

(5) Create file

Press 4 to create /test.txt

```
4
plz enter the path
/test.txt
successfully create File: /test.txt
File System
```

Type 3 and select the path "/" to view the root directory

```
3
plz enter the path
/
Directory contents of /:
- /
- user
- records
- appointments
- FS
- test.txt
```

(6) Write file

Press 5 to write /test.txt with context "this is a test"

```
5
plz enter path
/test.txt
plz enter context(type '-1' to finish)
this is a test
-1
successfully write file/test.txt
```

(7) Read file

Press 6 to read /test.txt with result "this is a test"

```
6
plz enter path
/test.txt

this is a test
```

(8) Open file Snapshot

Press 12 to open file Snapshot on /test.txt

```
12
plz enter path
/test.txt
File Snapshot open: /test.txt
```

(9) Write Append

Press 11 to write append /test.txt with context “this is test 2”

```
11
plz enter path
/test.txt
plz enter context(type '-1' to finish)
this is test 2
-1
successfully write append file/test.txt
```

Press 11 to write append /test.txt with context “this is test 3”

```
11
plz enter path
/test.txt
plz enter context(type '-1' to finish)
this is test 3
-1
successfully write append file/test.txt
```

Press 6 to read /test.txt

```
6
plz enter path
/test.txt

this is a test

this is test 2

this is test 3
```

Write Append Success

(10)List File Snapshot

Press 13 to list File Snapshot of file /test.txt

```
13
plz enter path
/test.txt
Directory contents of /FS/test.txt:
- 2024-12-24_15:53:54.txt
- 2024-12-24_15:54:21.txt
- 2024-12-24_15:55:5.txt
```

There are three snapshot files. The first one is the full snapshot saved when the file snapshot function is enabled, and the last two are incremental snapshots recorded in the append operation.

(11)Use File Snapshot

Press 14 to use File Snapshot of file /test.txt with time 2

```
14
plz enter path
/test.txt
plz enter time
2024-12-24_15:54:21
File Snapshot imply succussful: /test.txt
```

Read /test.txt again

```
6
plz enter path
/test.txt

this is a test

this is test 2
```

Use File Snapshot Success.

(12)Change File Permission

Create PATIENT user *p1* in ADMIN mode.

```
1
plz enter new user type : admin -- 0, doctor -- 1, patient -- 2
2
plz enter username
p1
plz enter user password
123456
successfully create a new user:p1
```

Use *p1* to enter File System Mode

```
4
plz enter user name
p1
welcome to file system mode
```

Try to read /test.txt

```
6
plz enter path
/test.txt
this file is empty
```

Result is empty, because *p1* permission denied. Server show error.

```
Error: User does not have read permission.
```

Now use *admin* to enter File System Mode and change file permission.

Enter 8 to change file permission to R|R.

```
8
plz enter the path
/test.txt
plz enter the generalPermission: N - No Permission, R - Reading Permission, W - Writing Permission
R
plz enter the groupPermission: N - No Permission, R - Reading Permission, W - Writing Permission
R
successfully change File Permission: generalPermission - R| groupPermission - R
```

Now use *p1* to read /test.txt again

```
6
plz enter path
/test.txt

this is a test

this is test 2
```

No problem.

(13)Change File Owner

Enter 9 to change file owner of /test.txt from “admin” to “p1”.

```
9
plz enter the path
/test.txt
plz enter the new owner
p1
successfully change File Owner
```

As owner of this file, p1 can read or write this file.

(14)Adjust User Group

Enter 10 to add “p1” into the user group of /test.txt. (if enter 0 will delete “p1” in the user group of /test.txt)

```
10
plz enter the path
/test.txt
plz enter the targetUsername
p1
plz enter 1 to add User, enter 0 to remove User
1
successfully adjust User Group
```

Now, ‘p1’ can use the permissions of the user group specified in the file instead of general user permissions.

4.2 Medical information management system

4.2.1 ADMIN

(1) Press 3 to enter the admin log in, enter the username “admin”,
enter the password “123456”.

Enter the admin operation.

```
-----AdministratorMode-----  
1. create user  
2. delete user  
3. create backup  
4. list backup  
5. load backup  
6. change user password  
0. back to upper directory  
-----  
█
```

(2) Create user

Enter 1 to create two patient(p1,p2) and one doctor(d1).

```
1  
plz enter new user type : admin -- 0, doctor -- 1, patient -- 2  
2  
plz enter username  
p1  
plz enter user password  
123456  
successfully create a new user:p1
```

```
1  
plz enter new user type : admin -- 0, doctor -- 1, patient -- 2  
2  
plz enter username  
p2  
plz enter user password  
123456  
successfully create a new user:p2
```

```
1  
plz enter new user type : admin -- 0, doctor -- 1, patient -- 2  
1  
plz enter username  
d1  
plz enter user password  
123456  
successfully create a new user:d1
```

(3) Change user password

Enter 6 to change “p2” password.

```
6
plz enter user type : admin -- 0, doctor -- 1, patient -- 2
2
plz enter username
p2
plz enter user new password
123
successfully change user password: p2
```

The patient Mode can be used for login verification to prove that the p2 password has been changed

(4) Delete user

Enter 2 to delete “p2”.

```
2
plz enter delete user type : admin -- 0, doctor -- 1, patient -- 2
2
plz enter delete username
p2
successfully delete user:p2
```

(5) Create backup


Enter 3 to create a backup.

```
3
successfully create a backup:
```

(6) List backup

Enter 4 to see available backup.

```
4
Backup files:
backup/1735045329.dat
```



The screenshot shows a file explorer window with a breadcrumb path: 主文件夹 / 桌面 / os / backup. Below the path, there is a file icon with a binary code pattern (0100, 0010, 1001, 0110) and the filename 1735045329.dat.

(7) Load backup

Now delete user p1.

```
2
plz enter delete user type : admin -- 0, doctor -- 1, patient -- 2
2
plz enter delete username
p1
successfully delete user:p1
```

Then load the backup.

```
5
plz enter backup file path
backup/1735045329.dat
successfully load a backup:backup/1735045329.dat
```

The patient Mode can be used for login verification to prove that the user “p1” back.

4.2.2 DOCTOR

Now enter 2 to use doctor mode.

```
Medical Data Management System
1. Patient log in
2. Doctor log in
3. Administrator log in
4. File System Mode
0. Exit System

2
plz enter user name
d1
plz enter password
123456
```

```
-----DoctorMode-----
1. list records
2. write append records
3. read records
4. release Appointments
5. revocation Appointments
6. list Appointments
7. read Appointments
0. back to upper directory
-----
```

(1) List record

Enter 1.

```
1
Directory contents of /records/:
- p1.txt
```

When a user creates a patient, the system automatically creates a record file for them in the /records directory.

(2) Write append record

Enter 2 to write “p1” record with context “something bad!”

```
2
plz enter patient username
p1
plz enter context(type '-1' to finish)
something bad!
-1
successfully write Append Records
```

(3) Read record

Enter 3 to read record.

```
3
plz enter records Name
p1.txt

2024-12-24_21:18:58
d1:

something bad!
```

The result includes the write time, doctor name and the context.

(4) Release Appointments

Enter 4 to release a appointment in 12/24/2024.

```
4
plz enter year
2024
plz enter month
12
plz enter day
24
successfully release Appointments
```

(5) List Appointments

Enter 6 to list appointments.

```
6
Directory contents of /appointments/:
- d1-2024-12-24.txt
```

(6) Read Appointments

After patient p1's appointment.

Enter 7 to read.

```
7
plz enter year
2024
plz enter month
12
plz enter day
24

2024-12-24 appointment
p1
```

The result includes patient “p1”.

(7) Revocation Appointments

Enter 5 to revocation appointments.

```
5
plz enter year
2024
plz enter month
12
plz enter day
24
successfully revocation Appointments
```

Then list appointments.

```
6
Directory contents of /appointments/:
```

This is empty now.

4.2.3 PATIENT

(1) Enter 1 to come in patient mode

```
Medical Data Management System
1. Patient log in
2. Doctor log in
3. Administrator log in
4. File System Mode
0. Exit System

1
plz enter user name
pl
plz enter password
1
```

(2) Read record

Enter 1 to read record.

```
1

2024-12-24_21:38:13
d1:

something bad!
```

(3) List all appointments

Enter 2 to list.

```
2
Directory contents of /appointments/:
- d1-2024-12-24.txt
```

(4) Make my appointments

Enter 3 to make.

```
3
plz enter doctor name
d1
plz enter year
2024
plz enter month
12
plz enter day
24
Appointments write succssful, plz remember you appointment time.
```

After make appointment, the doctor can find in the appointment file.

(4) Delete my appointments

Enter 4 to delete.

```
4
plz enter doctor name
d1
plz enter year
2024
plz enter month
12
plz enter day
24
Appointments delete succssful
```

The result can be check in doctor mode.

```

7
plz enter year
2024
plz enter month
12
plz enter day
24

this Appointments is empty

```

After delete the appointment file is empty.

4.2.4 EXIT

Enter 0 to exit whole system.

```

Medical Data Management System
1. Patient log in
2. Doctor log in
3. Administrator log in
4. File System Mode
0. Exit System

0

```

The filesystem.dat file will generate to keep the state in file system.



The next time the system starts, it will read the contents of the filesystem.dat file.

4.3 SOCKCT's client-server functions

In order to reflect the mutually exclusive access in the file system, I request a read lock when entering the file system mode. (This is actually a bad design, just to reflect the mutually exclusive access. A better design is to request a read lock or a write lock every time you need to call the file system interface, which is what I do in other parts of the code)

In fact, when the number of clients is small, the probability of simultaneous access is very low, which is not conducive to demonstrating the idea of mutually exclusive access.

If two client processes want to access the file system mode at the same time, the one that enters later will sleep because it cannot obtain the lock, until the former actively exits the mode, and the latter will be awakened and enter.

```
Medical Data Management System
1. Patient log in
2. Doctor log in
3. Administrator log in
4. File System Mode
0. Exit System

4
plz enter user name
admin
welcome to file system mode
-----FileSystem-----
1. create Directory
2. delete Directory
3. display Directory
4. create File
5. write File
```

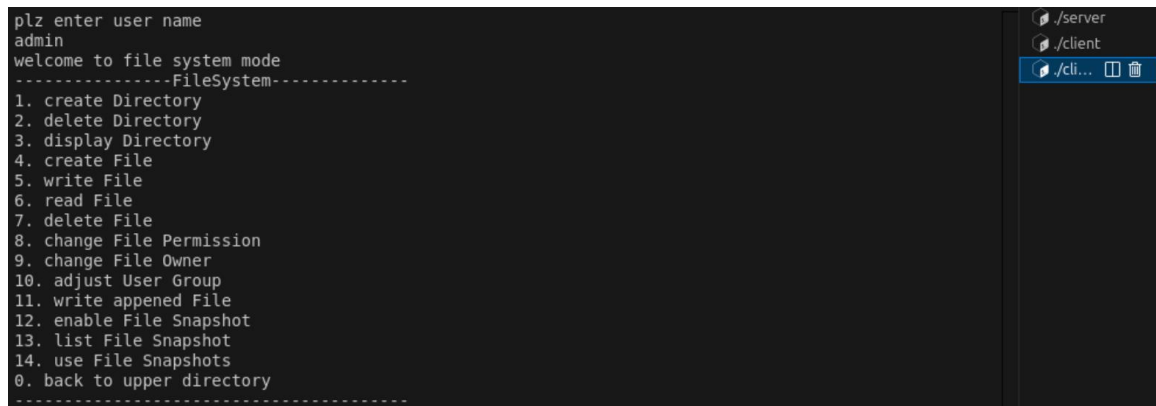
Client 1 enters file system mode first

```
Medical Data Management System
1. Patient log in
2. Doctor log in
3. Administrator log in
4. File System Mode
0. Exit System

4
plz enter user name
admin

```

Client 2 sleep to wait



```
plz enter user name
admin
welcome to file system mode
-----FileSystem-----
1. create Directory
2. delete Directory
3. display Directory
4. create File
5. write File
6. read File
7. delete File
8. change File Permission
9. change File Owner
10. adjust User Group
11. write appened File
12. enable File Snapshot
13. list File Snapshot
14. use File Snapshots
0. back to upper directory
-----
```

The sidebar on the right contains icons for `/server`, `/client`, and `/cli...`.

After Client 1 exit file system mode, Client 2 enters

4.4 Managing I-Node Limitations

In this system, each I-Node has four direct blocks, each block size is 1024. The maximum number of indirect blocks is 12. Therefore, a file has a maximum of 16 blocks. To verify the correctness of the code, we write 3000 "a", "b", "c" characters into the file three times respectively.



```
fileSystem fs;

fs.createFile("/test.txt", "admin");

string a;
a.append(3000, 'a');
string b;
b.append(3000, 'b');
string c;
c.append(3000, 'c');

fs.writeFile("test.txt", "admin", a, 1);
fs.writeAppendFile("test.txt", "admin", b);
fs.writeAppendFile("test.txt", "admin", c);
cout << fs.readFile("/test.txt", "admin") << endl;
```

Directly call the file system

Operation Results:

[illegible]

The write is successful, indicating that there is no problem with the indirect block design.