实验四——简单文件系统

- **1120211231 李卓**

1. 实验内容

通过对具体的文件存储空间的管理、文件的物理结构、目录结构和文件操作的实现，加深对文件系统内部功能和实现过程的理解。

> 实验全部内容已同步Github仓库：FileSystem

2. 程序中使用的数据结构及符号说明

2.1 虚拟磁盘

2.1.1 虚拟磁盘参数定义

在虚拟磁盘部分，使用宏定义定义了系统的一些参数：

```
/*-------------------文件系统常量-------------------*/
// 版本号
#define VERSION "1.0"
// 作者
#define AUTHOR "Li Zhuo"
// 邮箱
#define EMAIL "Lz1958455046@outlook.com or 1120211231@bit.edu.cn"
// 数据块大小
#define BLOCK_SIZE 512
// inode节点大小
#define INODE_SIZE 128
// 数据块指针数目
#define BLOCK_POINTER_NUM 12
// 目录项名称最大长度
#define MAX_DIRITEM_NAME_LEN 28
// 虚拟磁盘大小，即3.2MB
#define MAX_DISK_SIZE (320*1000)
// BLOCK数目，暂定为1024
#define BLOCK_NUM 1024
// inode节点数目，暂定为512
#define MAX_INODE_NUM 512
// 每个块组的块数
#define BLOCK_GROUP_SIZE 32
#define FREE_BLOCK_STACK_SIZE 64
// inode缓存大小
#define MAX_INODE_CACHE_SIZE 128

/*-------------------文件系统目录项/INode常量-------------------*/

// 文件类型
#define FILE_TYPE 00000
// 目录类型
#define DIR_TYPE 01000
// 链接类型
#define LINK_TYPE 02000
// 本用户读权限
#define OWNER_R 4<<6
//本用户写权限
#define OWNER_W 2<<6
// 本用户执行权限
#define OWNER_X 1<<6
// 组用户读权限
#define GROUP_R 4<<3
// 组用户写权限
#define GROUP_W 2<<3
// 组用户执行权限
#define GROUP_X 1<<3
// 其他用户读权限
#define OTHERS_R        4
// 其他用户写权限
#define OTHERS_W        2
// 其他用户执行权限
#define OTHERS_X        1
// 默认权限：文件，即本用户和组用户有读写权限，其他用户有读权限
#define DEFAULT_FILE_MODE 0777
// 默认权限：目录，即本用户有读写执行权限，其他用户和组用户有读和执行权限
#define DEFAULT_DIR_MODE 0777
```

```
/*------------------文件系统用户常量------------------*/
/* 用户 */
// 用户名最大长度
#define MAX_USER_NAME_LEN 20
// 用户组名最大长度
#define MAX_GROUP_NAME_LEN 20
// 用户密码最大长度
#define MAX_PASSWORD_LEN 20
// 用户状态
#define MAX_USER_STATE 1
// 主机名最大长度
#define MAX_HOST_NAME_LEN 32
```

在以上的宏定义中，分为文件系统常量、文件系统目录项/INode常量和文件系统用户常量三部分。
在**文件系统部分**
，主要的宏定义是对文件系统的一些参数进行定义，如数据块大小、inode节点大小、数据块指针数目、目录项名称最大长度、虚拟磁盘大小、BLOCK数目、inode节点数目、每个块组的块数、FREE_BLOCK_STACK_SIZE、inode缓存大小等。
而在文件系统**目录项/INode常量**
部分，主要是对文件类型、目录类型、链接类型、权限等进行定义。类型与权限采用了八进制的形式进行定义，如文件类型为00000，目录类型为01000，链接类型为02000，本用户读权限为4<<
6，本用户写权限为2<<6，本用户执行权限为1<<6，组用户读权限为4<<3，组用户写权限为2<<3，组用户执行权限为1<<
3，其他用户读权限为4，其他用户写权限为2，其他用户执行权限为1，文件的默认权限为0777，目录的默认权限为0777。
在文件系统**用户常量**部分，主要是对用户、用户组、用户密码、用户状态、主机名等进行定义，如用户名最大长度为20，用户组名最大长度为20，用户密码最大长度为20，主机名最大长度为32。

除此之外，也采用全局变量的形式定义了一些参数：

```
/* 全局变量 */
// 超级块开始位置,占一个磁盘块
const int superBlockStartPos = 0;
// inode位图开始位置,位图占两个磁盘块
const int iNodeBitmapStartPos = 1 * BLOCK_SIZE;
// 数据块位图开始位置，数据块位图占20个磁盘块
const int blockBitmapStartPos = iNodeBitmapStartPos + 2 * BLOCK_SIZE;
// inode节点开始位置,占 INODE_NUM/(BLOCK_SIZE/INODE_SIZE) 个磁盘块即 129 个磁盘块
const int iNodeStartPos = blockBitmapStartPos + 20 * BLOCK_SIZE;
// 数据块开始位置，占 BLOCK_NUM 个磁盘块
const int blockStartPos = iNodeStartPos + (MAX_INODE_NUM * INODE_SIZE) / BLOCK_SIZE + 1;
// 虚拟磁盘文件大小
const int diskBlockSize = blockStartPos + BLOCK_NUM * BLOCK_SIZE;
// 单个虚拟磁盘文件文件最大大小
const int maxFileSize = 9 * BLOCK_SIZE + BLOCK_SIZE / sizeof(int) * BLOCK_SIZE + BLOCK_SIZE / sizeof(int) * BLOCK_SIZE *
    BLOCK_SIZE / sizeof(int);
// 虚拟磁盘缓冲区，初始为最大缓冲区大小
static char diskBuffer[MAX_DISK_SIZE];
```

在以上的全局变量中，定义了超级块开始位置、inode位图开始位置、数据块位图开始位置、inode节点开始位置、数据块开始位置、虚拟磁盘文件大小、单个虚拟磁盘文件最大大小、虚拟磁盘缓冲区等参数。

2.1.2 虚拟磁盘数据结构

在虚拟磁盘数据结构部分，定义了虚拟磁盘的数据结构：

超级块

```
struct SuperBlock
{
    /* 节点数目 */
    // inode节点数目，最多65535个
    unsigned short iNodeNum;
    // 数据块数目，最多4294967295个
    unsigned int blockNum;
    // 空闲inode节点数目
    unsigned short freeINodeNum;
    // 空闲数据块数目
    unsigned int freeBlockNum;

    /* 空闲块堆栈 */

    // 空闲块堆栈
    int freeBlockStack[BLOCK_GROUP_SIZE];
    // 堆栈顶指针
    int freeBlockAddr;

    /* 大小 */

    // 磁盘块大小
    unsigned short blockSize;
    // inode节点大小
    unsigned short iNodeSize;
    // 超级块大小
    unsigned short superBlockSize;
    // 每个块组的块数
    unsigned short blockGroupSize;

    /* 磁盘分布（各个区块在虚拟磁盘中的位置） */

    // 超级块位置
    int superBlockPos;
    // inode位图位置
    int iNodeBitmapPos;
    // 数据块位图位置
    int blockBitmapPos;
    // inode节点起始位置
    int iNodeStartPos;
    // 数据块起始位置
    int blockStartPos;
};
```

超级块将被存储在虚拟磁盘的第一个磁盘块中，用于记录文件系统的基本信息，如inode节点数目、数据块数目、空闲inode节点数目、空闲数据块数目、空闲块堆栈、大小、磁盘分布等参数。
而在载入时，超级块将被载入到内存中，用于文件系统的操作。

inode节点

```
struct INode
{
    // inode节点号
    unsigned short iNodeNo;
    // 文件类型与存取权限 采用八进制表示 例如：0755表示文件类型为普通文件，所有者有读写执行权限，组用户和其他用户有读和执行权限
    unsigned short iNodeMode;
    // 链接数
    unsigned short iNodeLink;
    // 文件所有者，字符串
    char iNodeOwner[MAX_USER_NAME_LEN];
    // 文件所属组，字符串
    char iNodeGroup[MAX_GROUP_NAME_LEN];
    // 文件大小
    unsigned int iNodeSize;
    // 文件创建时间，时间戳
    time_t iNodeCreateTime;
    // 文件修改时间，时间戳
    time_t iNodeModifyTime;
    // 文件访问时间，时间戳
    time_t iNodeAccessTime;
    // 文件数据块指针，9个直接指针，1个一级间接指针，1个二级间接指针，1个三级间接指针
    int iNodeBlockPointer[BLOCK_POINTER_NUM];
};
```

为了保持128字节的大小，inode节点中的文件名、文件类型、文件权限、文件大小、文件创建时间、文件修改时间、文件访问时间、文件数据块指针等参数都被定义为了固定大小的数据类型，如文件名为28字节的字符串、文件类型为2字节的无符号短整型、文件权限为2字节的无符号短整型、文件大小为4字节的无符号整型、文件创建时间、文件修改时间、文件访问时间为8字节的时间戳、文件数据块指针为48字节的整型数组。

目录项

```
struct DirItem
{
    // 目录项名
    char itemName[MAX_DIRITEM_NAME_LEN];
    // inode地址
    int iNodeAddr;
};
```

目录项的大小为32字节，其中包括了目录项名和inode地址。在一个磁盘块中，可以存储16个目录项。

空闲目录项索引

```
struct FreeDirItemIndex
{
    // 目录项索引
    int dirItemIndex;
    // 目录项内的索引
    int dirItemInnerIndex;
};
```

该数据结构用于记录空闲目录项的索引，包括目录项索引和目录项内的索引。

2.2 INode缓存

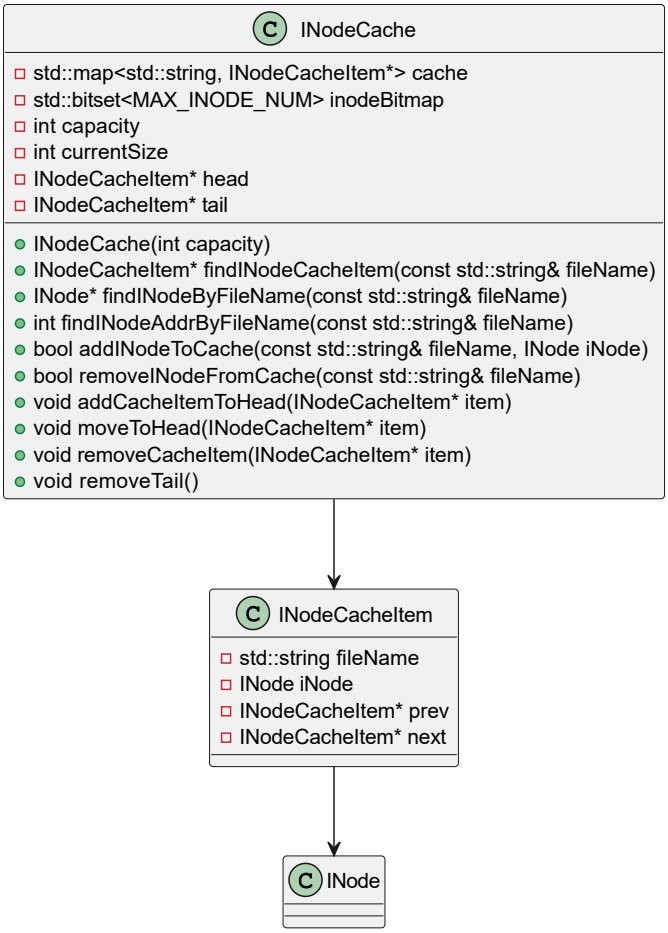
```
struct INodeCacheItem
{
    // file name,  key
    std::string fileName;
    // inode节点,  value
    INode iNode;
    // 前一个节点
    INodeCacheItem* prev;
    // 后一个节点
    INodeCacheItem* next;
};
```

INode缓存采用了LRU算法，用于缓存inode节点，减少对磁盘的访问次数。在本系统中采用了双向链表的形式进行存储。

```
class INodeCache
{
public:
    // INode缓存，使用哈希表存储
    std::map<std::string, INodeCacheItem*> cache;
    // 文件描述符的位图，用于查找空闲的文件描述符
    std::bitset<MAX_INODE_NUM> inodeBitmap;

    // 缓存大小
    int capacity;
    // 当前缓存大小
    int currentSize;

    // 头指针
    INodeCacheItem* head;
    // 尾指针
    INodeCacheItem* tail;

    explicit INodeCache(int capacity);
    INodeCacheItem* findINodeCacheItem(const std::string& fileName);
    INode* findINodeByFileName(const std::string& fileName);
    int findINodeAddrByFileName(const std::string& fileName);
    bool addINodeToCache(const std::string& fileName, INode iNode);
    bool removeINodeFromCache(const std::string& fileName);
    void addCacheItemToHead(INodeCacheItem* item);
    void moveToHead(INodeCacheItem* item);
    void removeCacheItem(INodeCacheItem* item);
    void removeTail();
};
```

以上是INode缓存的数据结构，其中包括了INode缓存、文件描述符的位图、缓存大小、当前缓存大小、头指针、尾指针等参数。在INodeCache类中，定义了一些方法，如通过文件名查找INode节点、添加INode节点到缓存、从缓存中删除INode节点等。

而以上类的类图如下:



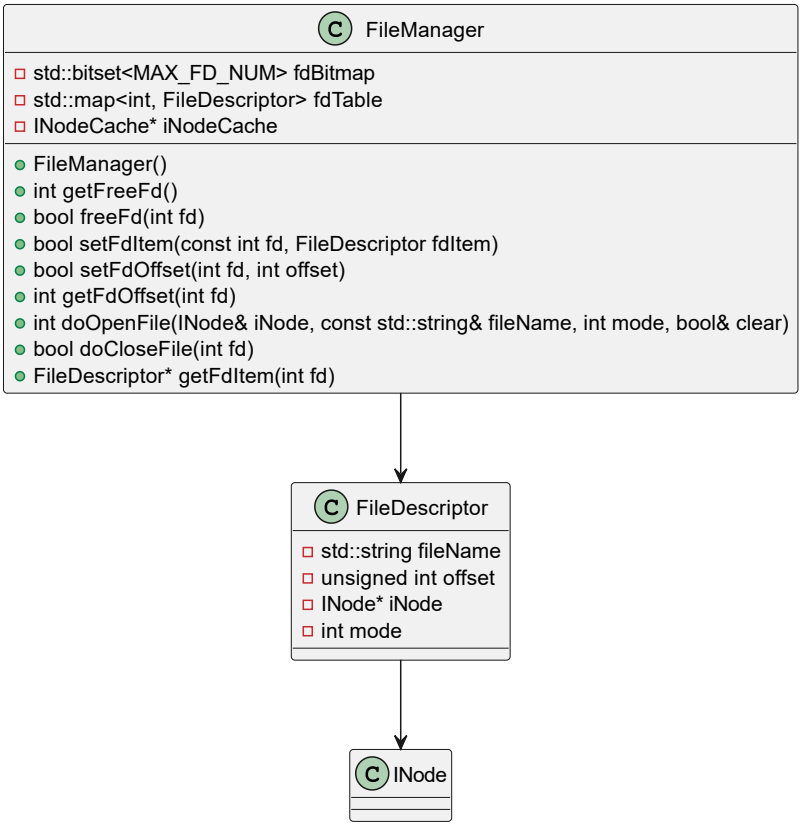
2.3 文件描述符管理

定义了如下的常量与数据结构：

```
/*-------------------文件描述符-------------------*/
// 最大文件描述符数目
#define MAX_FD_NUM 128
// 文件打开模式，采用二进制描述，在模式控制时使用位运算
#define MODE_R 0b001
#define MODE_W 0b010
// 是否为追加模式
#define MODE_A 0b100
// 默认文件打开模式
#define MODE_DEFAULT MODE_R

/**
 * 文件描述句柄
 * 当前文件偏移量（调用read()和write()时更新，或使用lseek()直接修改）
 * 对应的INode引用
 * 文件打开模式 r/w/rw
 */
struct FileDescriptor
{
    // 文件名
    std::string fileName;
    // 当前文件偏移量
    unsigned int offset;
    // INode引用
    INode* iNode;
    // 文件打开模式
    int mode;
};

/**
 * 文件管理类
 * 用于管理文件系统中的文件文件描述符和文件缓存的对应关系
 */
class FileManager
{
public:
    // 文件描述符的位图，用于查找空闲的文件描述符
    std::bitset<MAX_FD_NUM> fdBitmap;
    // 文件描述符表，用于查找文件描述符对应的文件句柄
    std::map<int, FileDescriptor> fdTable;
    // 文件缓存，用于缓存文件内容
    INodeCache* iNodeCache;
    FileManager();
    int getFreeFd();
    bool freeFd(int fd);
    bool setFdItem(const int fd, FileDescriptor fdItem);
    bool setFdOffset(int fd, int offset);
    int getFdOffset(int fd);
    int doOpenFile(INode& iNode, const std::string& fileName, int mode, bool& clear);
    bool doCloseFile(int fd);
    FileDescriptor* getFdItem(int fd);
};
```

在文件描述符管理部分，定义了最大文件描述符数目、文件打开模式、是否为追加模式、默认文件打开模式等常量。在数据结构部分，定义了文件描述句柄和文件管理类。文件描述句柄包括了文件名、当前文件偏移量、INode引用、文件打开模式等参数。文件管理类包括了文件描述符的位图、文件描述符表、文件缓存等参数。在文件管理类中，定义了一些方法，如获取空闲文件描述符、释放文件描述符、设置文件描述符、设置文件偏移量、获取文件偏移量、打开文件、关闭文件、获取文件描述符等。

以上的类的类图如下：



2.4 文件系统帮助类

仅为一个静态类，用于提供一些文件系统操作的帮助函数。

```
/**
* 帮助文档类
* @brief 显示帮助文档
*/
class Helper {
public:
    /**
    * 显示帮助文档
    */
    static void showHelpInformation(const std::string& command);
private:
    // new命令帮助信息
    static void showNewHelp();
    // sys命令帮助信息
    static void showSysHelp();
    // cd命令帮助信息
    static void showCdHelp();
    // ls命令帮助信息
    static void showLsHelp();
    // ...
};
```

以上的类中，具体的方法帮助暂未实现，仅为一个静态类。但实现了总的帮助文档的显示。

2.5 文件系统类

文件系统类是整个文件系统的核心类，用于实现文件系统的基本操作。

首先定义了一个 `UserState` 结构体，用于表示用户的状态：

```
/**
 * 用户状态
 */
struct UserState
{
    // 是否已经登录
    bool isLogin;
    // 用户名
    char userName[MAX_USER_NAME_LEN];
    // 用户所在组
    char userGroup[MAX_GROUP_NAME_LEN];
    // 用户密码
    char userPassword[MAX_PASSWORD_LEN];
    // 用户ID
    unsigned short userID;
    // 用户组ID
    unsigned short userGroupID;
};
```

然后定义了文件系统类：

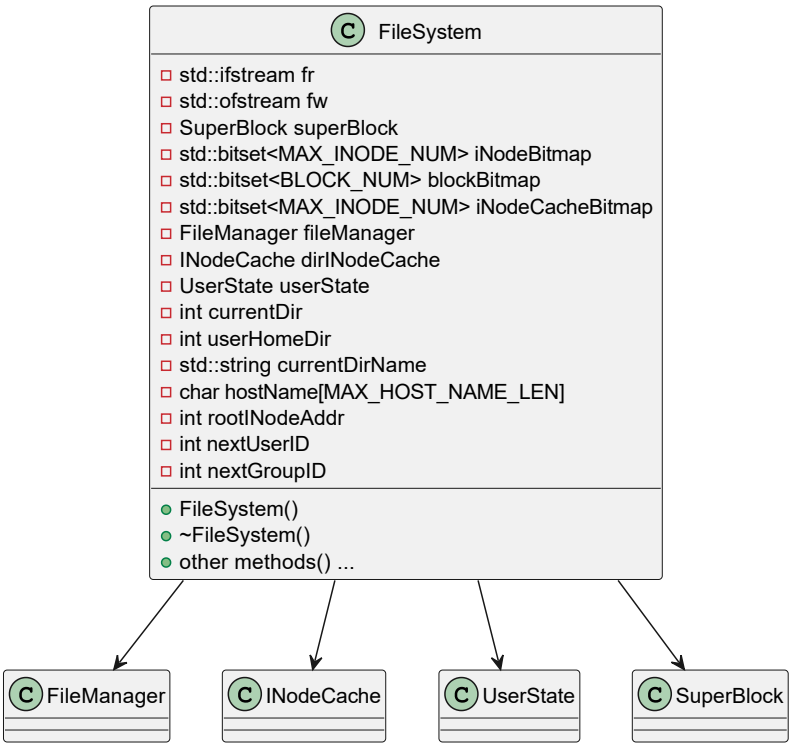
```
class FileSystem {
public:
    // 构造函数
    FileSystem();
    // 析构函数
    ~FileSystem();
    void printBuffer();
    void readSuperBlock();
    bool installed;

    /*-------------------文件系统基本操作-------------------*/

    void getHostName();
    int allocateINode();
    bool freeINode(int iNodeAddr);
    int allocateBlock();
    bool freeBlock(int blockAddr);
    bool formatFileSystem();
    void initSystemConfig();
    void createFileSystem(const std::string& systemName);
    bool installFileSystem();
    void uninstall();
    void load(const std::string &filename);
    void save();
    void executeInFS(const std::string &command);
    void printSuperBlock();
    void printSystemInfo();
    void printUserAndHostName();
    void clearScreen();
    std::string getAbsolutePath(const std::string& dirName, const std::string& INodeName);
    INode* findINodeInDir(INode& dirINode, const std::string& dirName, const std::string& INodeName, int type);
    FreeDirItemIndex findFreeDirItem(const INode& dirINode, const std::string& itemName, int itemType);
    FreeDirItemIndex findFreeDirItem(const INode& dirINode);
    void clearINodeBlocks(INode& iNode);
    int calculatePermission(const INode& iNode);
    INode* findINodeInCache(const std::string& absolutePath, int type);

    /*-------------------文件系统文件操作-------------------*/

    int openFile(std::string& dirName, int dirAddr, std::string& fileName, int mode);
    void openWithFilename(const std::string& arg);
    bool closeWithFd(int fd);
    void closeFile(const std::string& arg);
    void seekWithFd(const std::string& arg);
    bool createFileHelper(const std::string& fileName, int dirINodeAddr, char* content, unsigned int size);
    void createFile(const std::string& arg);
    bool sysReadFile(INode& iNode, unsigned int offset, unsigned int size, char* content);
    bool readWithFd(int fd, char* content, unsigned int size);
    void readFile(const std::string& arg);
    void catFile(const std::string& arg);
    bool sysWriteFile(INode& iNode, const char* content, unsigned int size, unsigned int offset);
    bool writeWithFd(int fd, const char* content, unsigned int size);
    void writeFile(const std::string& arg);
    void deleteFile(const std::string& arg);
    void echoFile(const std::string& arg);

    /*-------------------文件系统目录操作-------------------*/
```

```
    void makeDir(const std::string& arg);
    bool mkdirHelper(bool pFlag, const std::string& dirName, int inodeAddr);
    bool rmdirHelper(bool ignore, bool pFlag, const std::string& dirName, int inodeAddr);
    void removeDir(const std::string& arg);
    void listDir(const std::string& arg);
    void listDirByINode(int inodeAddr, int lsMode);
    bool changeDir(const std::string& arg, int& currentDir, std::string& currentDirName);
    void printCurrentDir();

    /*-------------------文件系统用户操作-------------------*/

    void login(const std::string &userName, const std::string &password);
    void logout();
    void createUser(const std::string &userName, const std::string &password);
    void deleteUser(const std::string &userName);
    void changePassword(const std::string &userName, const std::string &password);
    void listUser();
    void listGroup();

    /*-------------------文件系统权限操作-------------------*/

    void changeMode(const std::string &filename, const std::string &mode);
    void changeOwner(const std::string &filename, const std::string &owner);
    void changeGroup(const std::string &filename, const std::string &group);

    /*-------------------文件系统其他操作-------------------*/

    void viEditor(char* str, int inodeAddr, unsigned int size);
    void vi(const std::string& arg);

private:
    /*-------------------系统-------------------*/
    // 读系统文件的指针
    std::ifstream fr;
    // 写系统文件的指针
    std::ofstream fw;
    // 超级块
    SuperBlock superBlock{};
    // inode位图
    std::bitset<MAX_INODE_NUM> iNodeBitmap;
    // 数据块位图
    std::bitset<BLOCK_NUM> blockBitmap;
    // 判断inode缓存是否与磁盘同步的位图
    std::bitset<MAX_INODE_NUM> iNodeCacheBitmap;
    // FileManager类
    FileManager fileManager;
    // // 文件inode缓存
    // INodeCache fileINodeCache;
    // 目录inode缓存
    INodeCache dirINodeCache;

    /*-------------------用户-------------------*/
    // 用户状态
    UserState userState{};
    // 当前目录
    int currentDir{};
    // 当前用户主目录
```

```
    int userHomeDir;
    // 当前目录名
    std::string currentDirName;
    // 当前主机名
    char hostName[MAX_HOST_NAME_LEN]{};
    // 根目录inode节点
    int rootINodeAddr{};
    // 下一个要被分配的用户标识
    int nextUserID;
    // 下一个要被分配的组标识
    int nextGroupID;

};
```

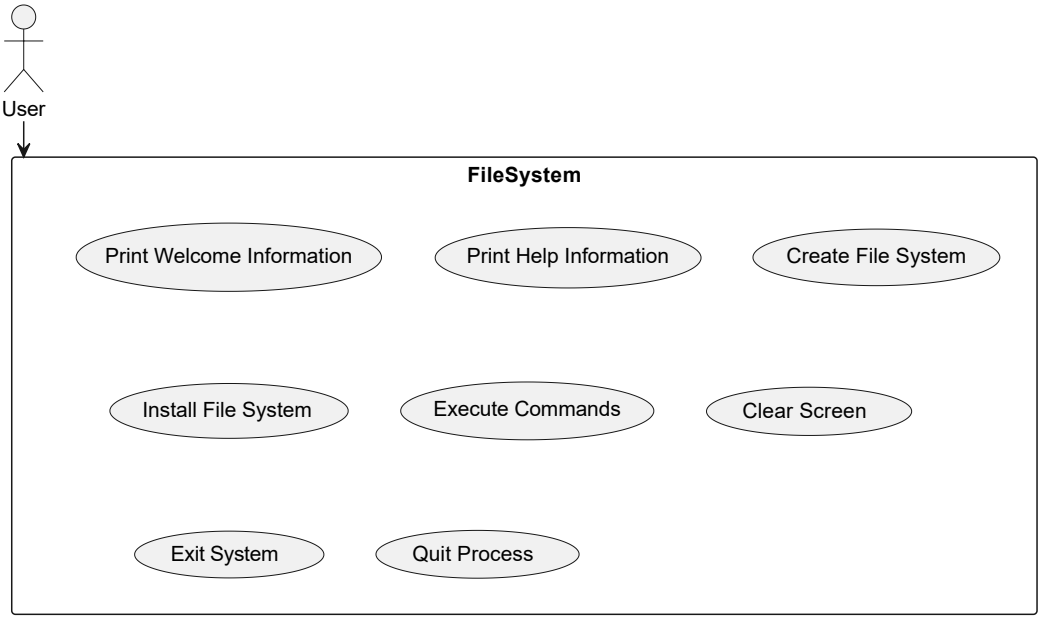
在文件系统类中，定义了一些基本的文件系统操作、文件操作、目录操作、用户操作、权限操作、其他操作等方法。在文件系统类中，定义了一些私有的参数，如读系统文件的指针、写系统文件的指针、超级块、inode位图、数据块位图、判断inode缓存是否与磁盘同步的位图、FileManager类、目录inode缓存、用户状态、当前目录、当前用户主目录、当前目录名、当前主机名、根目录inode节点、下一个要被分配的用户标识、下一个要被分配的组标识等参数。

以上类的类图如下：



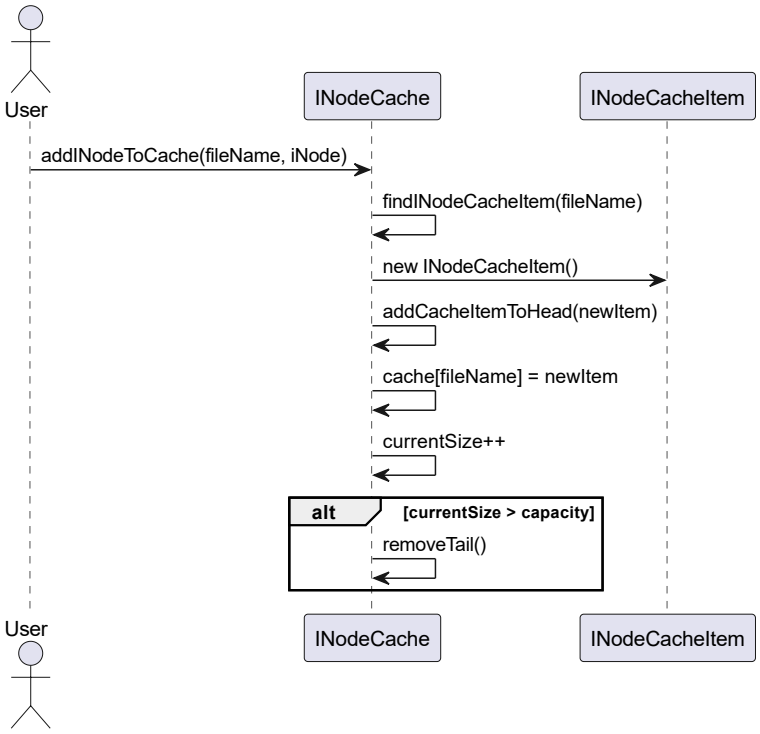
3. 模块设计与处理流程

用户交互逻辑：



3.1 INode缓存

在缓存模块中，采用了LRU算法，用于缓存inode节点，减少对磁盘的访问次数。在本系统中采用了双向链表的形式进行存储。



以上是INode缓存添加INode节点的序列图。

而对于查找INode节点，采用了哈希表的形式进行存储，通过文件名查找INode节点。
另外，在删除节点时，则是查找到节点后，将节点从链表中删除，并释放内存。

3.2 文件描述符管理

在文件描述符管理模块，主要涉及到文件描述符的分配、释放、设置、打开、关闭、读写等操作。重点在于文件描述符的分配与文件打开操作。

在分配时，会去位图中查找最小的空闲文件描述符，然后将其分配给用户。

而在打开文件时，会首先申请一个文件描述符，然后新建一个文件描述句柄，将文件描述符与文件描述句柄进行绑定，根据权限设置句柄的偏移量、mode等属性，最后将文件描述句柄存入文件描述符表中。

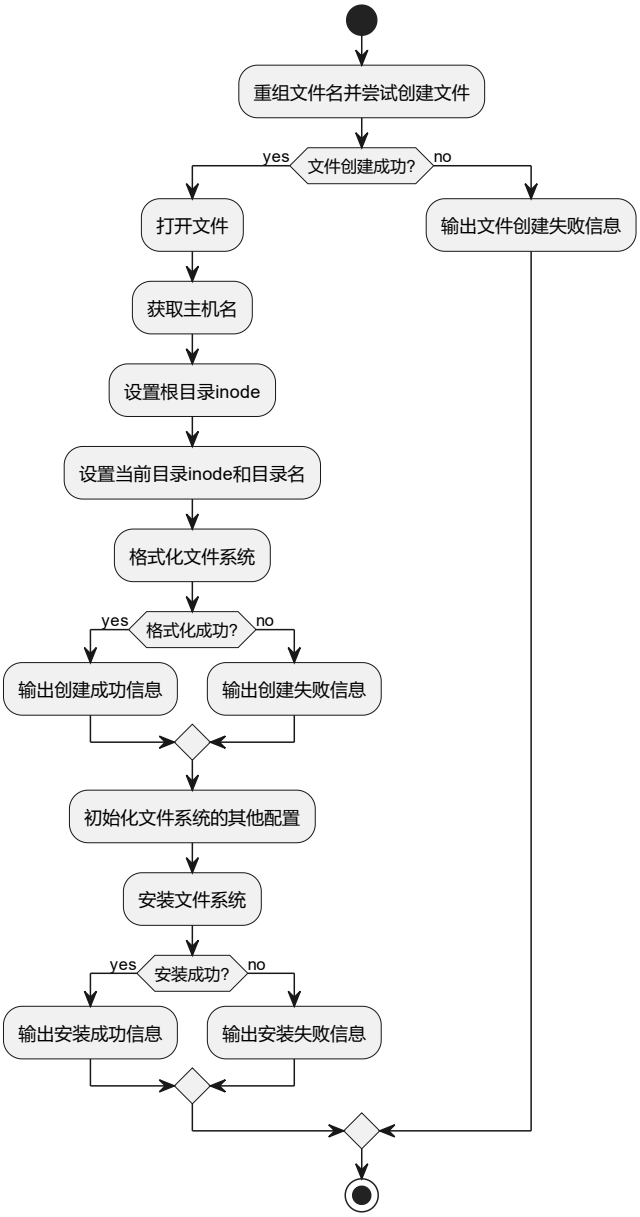
3.3 文件系统模块

3.3.1 文件系统新建与载入

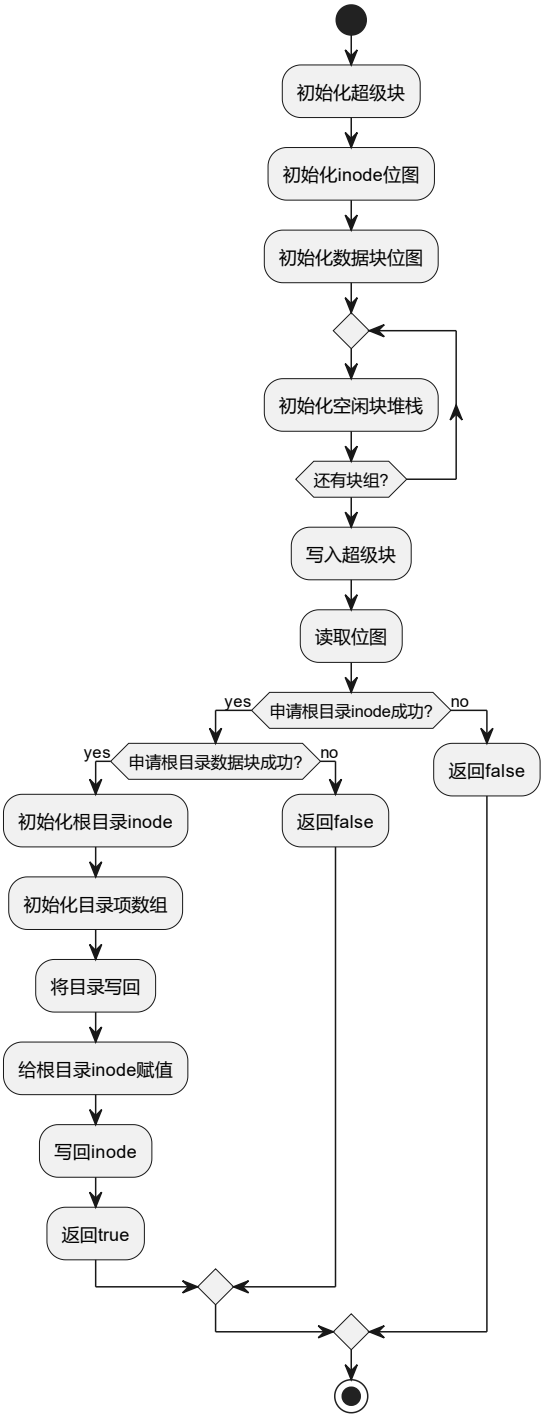
在文件系统新建与载入模块中，主要涉及到文件系统的新建、载入、格式化、安装、保存、卸载等操作。

以下采用了状态图的形式进行展示处理流程：

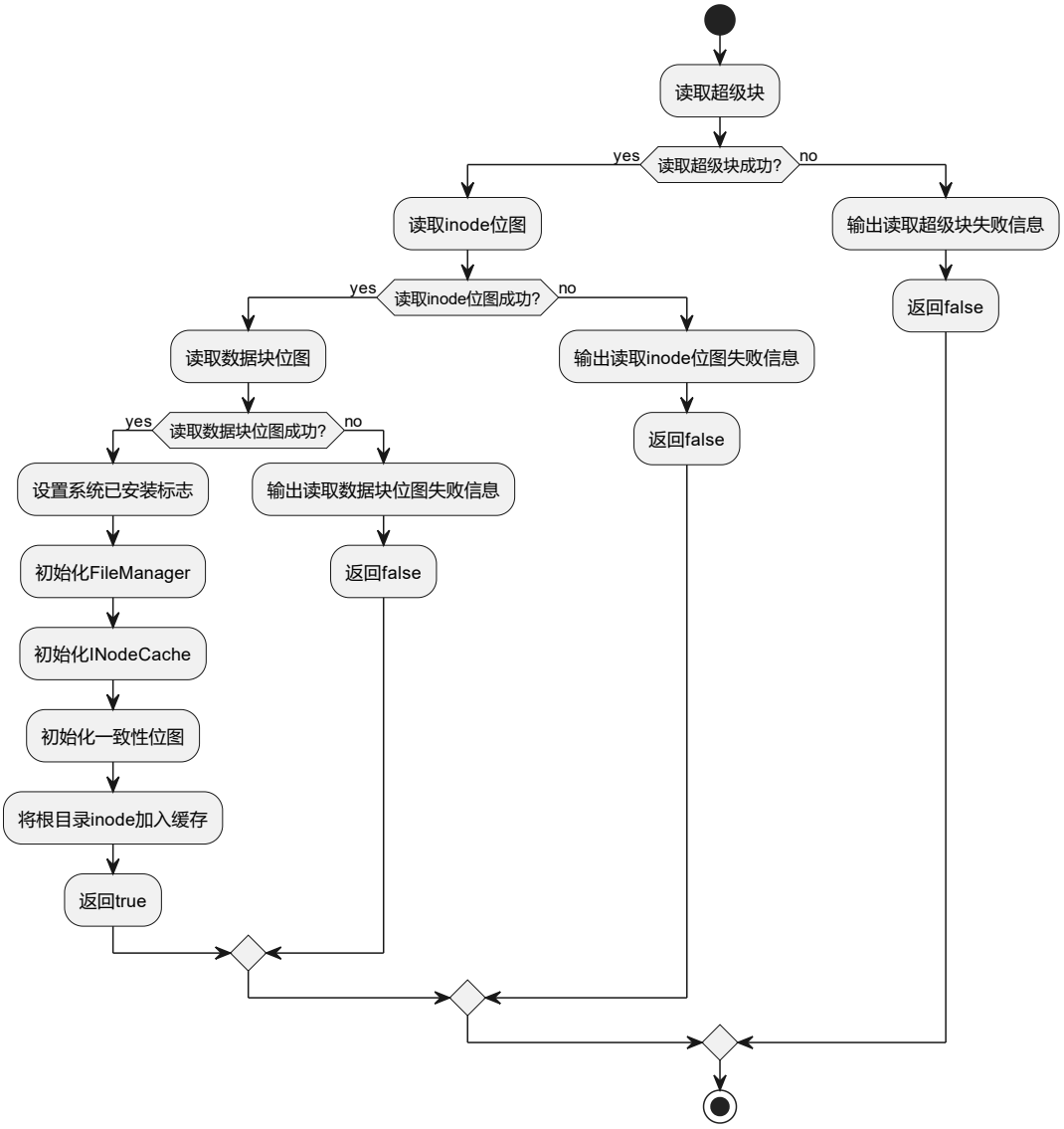
新建：——对应new命令



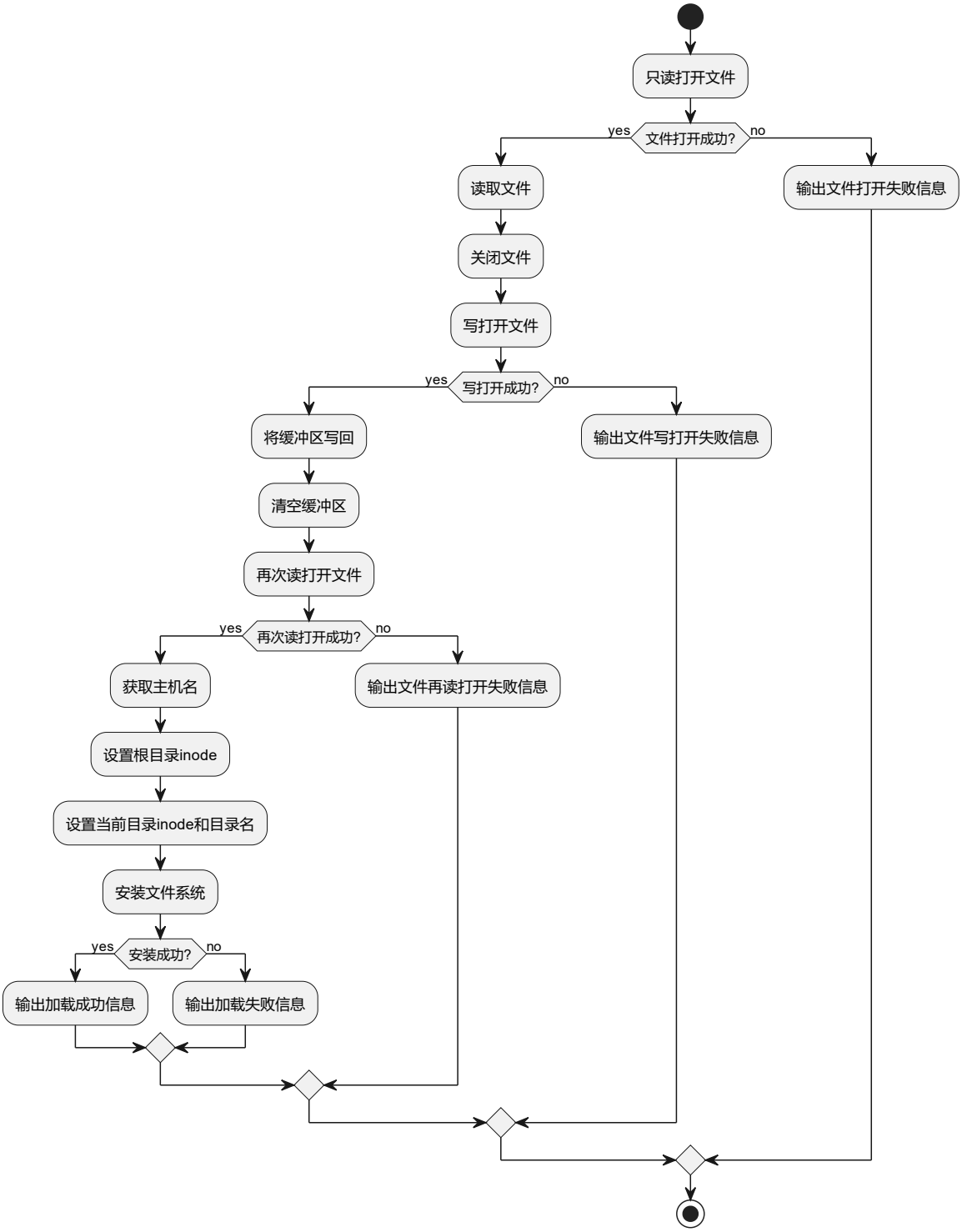
格式化文件系统：


初始化超级块
初始化inode位图
初始化数据块位图
初始化空闲块堆栈
还有块组?
写入超级块
读取位图
yes
申请根目录inode成功?
no
返回false
yes
申请根目录数据块成功?
no
初始化根目录inode
返回false
初始化目录项数组
将目录写回
给根目录inode赋值
写回inode
返回true

安装：


读取超级块
读取超级块成功?
yes
no
读取inode位图
输出读取超级块失败信息
返回false
读取inode位图成功?
yes
no
读取数据块位图
输出读取inode位图失败信息
返回false
读取数据块位图成功?
yes
no
设置系统已安装标志
输出读取数据块位图失败信息
初始化FileManager
返回false
初始化INodeCache
初始化一致性位图
将根目录inode加入缓存
返回true

载入：——对应sfs命令

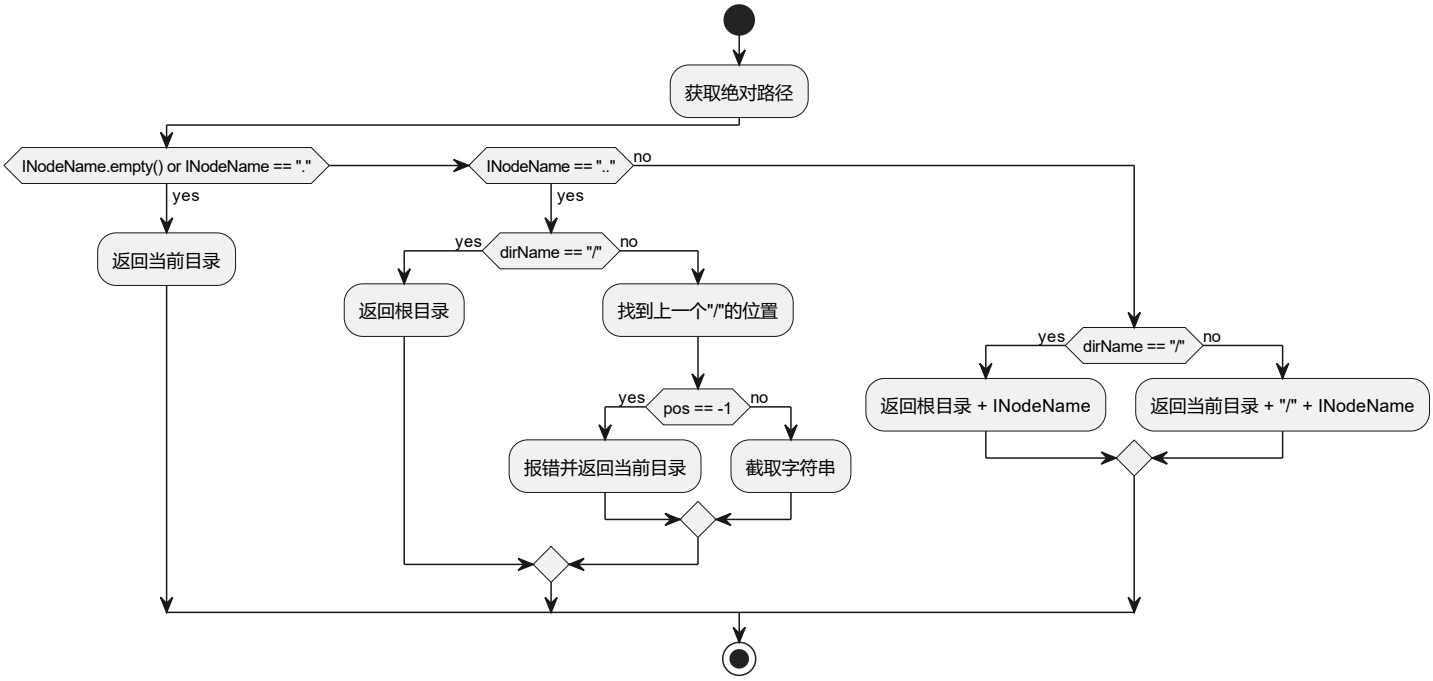


3.3.2 文件系统基础操作

获取绝对路径 - `getAbsolutePath`

该函数通过给定的当前目录和inode名称，返回绝对路径。

- **输入为空或为"."**：返回当前目录。
- **输入为"..."**：处理父目录。如果当前目录为根目录，返回根目录。否则，找到上一个"/"的位置并截取字符串。
- **其他情况**：根据当前目录是否为根目录，拼接并返回完整路径。



分配inode - `allocateINode`

该函数在inode位图中找到一个空闲的inode，并将其分配给用户，更新inode位图。

- **无空闲inode**：报错并返回-1。
- **顺序查找inode位图**：找到第一个空闲inode，更新位图和超级块，将结果写入文件，清空缓冲区并返回inode地址。

释放inode - `freeINode`

该函数释放给定地址的inode。

- **地址非法**：报错并返回false。
- **清空inode**：更新位图和超级块，将结果写入文件，清空缓冲区并返回true。

分配数据块 - `allocateBlock`

该函数从空闲块堆栈中分配一个数据块。

- **无空闲数据块**：报错并返回-1。
- **计算当前栈对应组的栈顶**：如果当前是栈底，返回并更新栈顶。否则，返回栈顶并更新超级块和数据块位图。

释放数据块 - `freeBlock`

该函数释放给定地址的数据块。

- **地址非法或块已空闲**：报错并返回false。
- **清空数据块**：根据情况更新栈顶，将块放到栈顶，更新超级块和数据块位图，写入文件并清空缓冲区。

查找缓存中的inode - `findINodeInCache`

这个方法根据绝对路径和类型查找缓存中的inode。

1. **输入参数**：
   - `absolutePath`： inode的绝对路径。
   - `type`： inode的类型（目录或文件）。
2. **流程**：
   - 根据类型判断是目录还是文件。
   - 在对应的缓存中查找inode。

- 检查inode是否被释放。如果被释放，则从缓存中移除，并返回 `nullptr` 。
- 检查inode在缓存中的一致性。如果不一致，则从磁盘中读取inode，并更新缓存。
- 返回找到的inode或 `nullptr` 。

根据目录INode查找目录下指定的文件/目录INode - `findINodeInDir`

这个方法根据目录INode查找指定文件或目录的inode。

1. **输入参数**:
   - `dirINode`： 目录的INode。
   - `dirName`： 目录名。
   - `INodeName`： 需要查找的文件或目录名。
   - `type`： inode的类型（目录或文件）。
2. **流程**:
   - 检查输入的目录是否为合法目录。
   - 获取目标项的绝对路径，并在缓存中查找inode。
   - 如果缓存中找到，则返回该inode。
   - 读取目录项并遍历，查找目标项的inode。
   - 如果找到匹配项，则检查类型是否匹配，匹配则返回该inode并更新缓存。
   - 如果没有找到，返回 `nullptr` 。

查找目录下空闲的目录项并检查重名 - `findFreeDirItem`

这个方法根据目录INode查找空闲的目录项，并检查是否有重名目录项。

1. **输入参数**:
   - `dirINode`： 目录的INode。
   - `itemName`： 目录项名。
   - `itemType`： 目录项类型。
2. **流程**:
   - 检查输入的目录是否为合法目录，并检查权限。
   - 读取目录项并遍历，查找空闲的目录项。
   - 如果找到空闲项，则返回其索引。
   - 如果需要检查重名，则继续查找，并在找到重名项时返回错误。
   - 如果没有找到空闲项，则返回索引。

重载方法： `findFreeDirItem`

该方法是无参数 `itemName` 和 `itemType` 的重载版本，直接调用带参数的版本查找空闲的目录项。

3.3.3 文件系统目录操作

目录操作主要包括以下内容：

1. **改变当前目录 (`changeDir`)**
2. **创建目录 (`makeDir` & `mkdirHelper`)**
3. **删除目录 (`rmdirHelper`)**

重点解析：改变当前目录 (`changeDir`)

1. 函数功能
   - **功能**：该函数实现了更改当前工作目录的功能。
   - **参数**:
     - `arg`：要进入的目录路径。
     - `currentDir`：当前目录的 iNode 地址。
     - `currentDirName`：当前目录的名称。

以下为函数的主要流程：

- **参数为空或为根目录**：若 `arg` 为空或为 `~`，则切换到用户主目录或根目录。

```
if (arg.empty() || arg == "~") {
  if (userHomeDir == -1) {
    currentDir = rootINodeAddr;
    currentDirName = "/";
  } else {
    currentDir = userHomeDir;
    currentDirName = "/home/" + std::string(userState.userName);
  }
  return true;
}
```

- **绝对路径**：若 `arg` 以 `/` 开头，则认为是绝对路径，设置当前目录为根目录后递归查找目标路径。

```
if (arg.at(0) == '/') {
  currentDir = rootINodeAddr;
  currentDirName = "/";
  std::string remainPath = arg.substr(1);
  return changeDir(remainPath, currentDir, currentDirName);
}
```

- **相对路径**：解析路径，提取第一个目录名和剩余路径。

```
std::string dirName;
std::string remainPath;
if (arg.find('/') != std::string::npos) {
  int pos = arg.find('/');
  dirName = arg.substr(0, pos);
  remainPath = arg.substr(pos + 1);
} else {
  dirName = arg;
  remainPath = "";
}
```

- **目录名长度检查**：检查目录名长度是否合法。
- **获取当前目录的 iNode**：从文件中读取当前目录的 iNode。
- **查找下一级目录的 iNode**：使用 `findINodeInDir` 函数在当前目录下查找指定名称的目录项。
- **权限检查和更新目录路径**：若找到目录项并有权限，则更新当前目录 iNode 地址和名称。若有剩余路径，则递归调用 `changeDir`。

```
    if (nextINode != nullptr) {
      if (((nextINode->iNodeMode >> accessPermission >> 2) & 1) == 0 && strcmp(userState.userName, "root") != 0) {
        std::cerr << "Error: Permission denied." << std::endl;
        return false;
      }
      currentDir = superBlock.iNodeStartPos + nextINode->iNodeNo * superBlock.iNodeSize;
      if (strcmp(dirName.c_str(), ".") == 0) {
        // currentDirName = currentDirName;
      } else if (strcmp(dirName.c_str(), "..") == 0) {
        if (currentDirName == "/") {
          currentDirName = "/";
        } else {
          int pos = currentDirName.find_last_of('/');
          if (pos == 0) {
            currentDirName = "/";
          } else {
            currentDirName = currentDirName.substr(0, pos);
          }
        }
      } else {
        if (currentDirName == "/") {
          currentDirName += dirName;
        } else {
          currentDirName += "/" + dirName;
        }
      }
      if (!remainPath.empty()) {
        return changeDir(remainPath, currentDir, currentDirName);
      }
      return true;
    }
```

解析目录项操作重点

- **路径解析**：无论是绝对路径还是相对路径，首先要解析出需要匹配的目录名和剩余路径。
- **权限检查**：确保用户对要进入的目录具有权限。
- **递归处理**：若有剩余路径，则递归调用自身处理剩余路径。
- **路径更新**：成功进入目录后，更新当前目录的 iNode 地址和名称。

其他目录操作

1. 创建目录（`makeDir` & `mkdirHelper`）

- **功能**：在指定路径创建目录，支持 `-p` 参数递归创建父级目录。
- **参数检查**：包括路径合法性、权限检查等。
- **目录项创建**：找到空闲的目录项位置，分配 iNode 和数据块，并更新目录项和 iNode。

2. 删除目录（`rmdirHelper`）

- **功能**：递归删除指定目录。
- **参数检查**：包括路径合法性、权限检查等。
- **目录项删除**：查找目录项并删除，释放 iNode 和数据块，并更新父级目录的目录项和 iNode。

在目录操作这一部分：

- **路径解析和递归处理**是目录操作的核心逻辑。

- **权限检查**和**错误处理**确保操作的安全性和可靠性。
- **更新目录路径和 iNode**是目录操作成功的重要步骤。

3.3.4 文件系统文件操作

首先，对文件操作的文件打开、文件关闭、文件读取、文件写入等操作进行了设计。

这段代码实现了基本的文件操作，包括打开、关闭、写入文件以及设置文件指针位置。让我们逐步理解这些操作的具体实现细节：

打开文件（`openFile`）

1. **读取目录项**：
   - 从上级目录读取 `inode`。
   - 调用 `findINodeInDir` 找到文件的 `inode`。
2. **权限检查**：
   - 计算访问权限，检查是否有读取或写入权限。
   - 如果文件不存在且模式为只读，返回错误。
   - 如果没有写入权限，返回错误。
3. **文件创建**：
   - 如果文件不存在且有写入权限，则调用 `createFileHelper` 创建文件。
   - 重新读取文件的 `inode`。
4. **权限检查（读写）**：
   - 检查模式中的读写权限，如果没有相应权限，返回错误。
5. **调用 `FileManager`**：
   - 调用 `fileManager.doOpenFile` 打开文件。
   - 如果 `clear` 为真，则清空文件并重新分配数据块，更新 `inode`。

解析 `open` 命令（`openWithFilename`）

- 解析 `open` 命令的参数，支持设置模式（r、w、a）。
- 根据文件路径，找到文件所在目录并调用 `openFile` 打开文件。
- 打开成功后，返回文件描述符。

关闭文件（`closeWithFd`）

- 调用 `fileManager.doCloseFile` 关闭文件，返回操作结果。

解析 `close` 命令（`closeFile`）

- 解析 `close` 命令参数，获取文件描述符。
- 调用 `closeWithFd` 关闭文件。

设置文件指针位置（`seekWithFd`）

- 解析 `seek` 命令参数，获取文件描述符和偏移量。
- 调用 `fileManager.setFdOffset` 设置文件指针位置，返回操作结果。

写入文件（`sysWriteFile`）

- 计算 `inode` 地址，检查写入权限。
- 检查偏移量和写入内容大小。
- 计算起始的磁盘块号，并依次写入数据。
- 如果超出直接块范围，使用二级索引。
- 更新 `inode` 信息并写回磁盘。

通过文件描述符写入文件（`writeWithFd`）

- 获取文件描述符，检查是否为写入模式。
- 调用 `sysWriteFile` 写入数据。

- 更新文件描述符的偏移量。

解析 `write` 命令（`writeFile`）

- 解析 `write` 命令参数，获取文件描述符和写入内容。
- 调用 `writeWithFd` 写入数据，返回操作结果。

解析 `create` 命令（`createFile`）

- 解析 `createFile` 命令参数，获取文件名和文件内容。
- 如果文件名是绝对路径，从根目录开始查找；如果是相对路径，从当前目录开始查找。
- 使用 `changeDir` 函数找到文件的 inode 并切换到文件所在目录。
- 调用 `createFileHelper` 创建文件并返回操作结果。

`sysReadFile`

- 检查读取大小是否为 0，如是则直接返回成功。
- 检查偏移量和读取大小是否超出文件大小，如超出则报错。
- 计算起始磁盘块和偏移量，每次读取一个磁盘块的数据，直到读取完指定大小的数据。
- 如果读取过程中遇到错误，则报错并返回失败。

readWithFd`

- 调用 `FileManager` 获取文件描述符对应的文件描述信息。
- 检查文件是否打开，是否有读取权限。
- 获取文件的 inode，判断读取权限。
- 调用 `sysReadFile` 从文件中读取数据，并更新文件描述符的偏移量。

解析 `read` 命令（`readFile`）

- 解析 `readFile` 命令参数，获取文件描述符和读取大小。
- 检查参数合法性，解析文件描述符和读取大小。
- 调用 `readWithFd` 从文件中读取数据并输出。

解析 `cat` 命令（`catFile`）

- 解析 `cat` 命令参数，获取文件名、读取大小和起始位置。
- 使用 `changeDir` 函数找到文件的 inode 并切换到文件所在目录。
- 打开文件，如果未指定读取大小，则默认读取文件剩余部分。
- 调用 `readWithFd` 从文件中读取数据并输出。
- 关闭文件。

解析 `delete` 命令（`deleteFile`）

- 解析 `delete` 命令参数，获取文件名。
- 使用 `changeDir` 函数找到文件的 inode 并切换到文件所在目录。
- 遍历目录项，找到文件 inode，判断是否有删除权限。
- 如果是文件，释放 inode 和磁盘块，清空目录项，更新链接数，并写回 inode。

解析 `echo` 命令（`echoFile`）

- 解析 `echo` 命令参数，获取文件名和写入内容。
- 使用 `changeDir` 函数找到文件的 inode 并切换到文件所在目录。
- 打开文件，调用 `writeWithFd` 写入数据并返回操作结果。
- 关闭文件。

4. 测试

本次实验中，主要采用单元测试与系统测试的方式进行测试。

4.1 单元测试

在单元测试中，主要对文件系统的虚拟磁盘读写、inode块与数据块的分配与释放、文件描述符的分配与释放以及inode缓存进行了测试。

4.2 系统测试

在系统测试中，主要采用了黑盒测试的方式进行测试，对文件系统的新建、载入、格式化、安装、保存、卸载等本次实验所要求的功能进行了测试。

测试的方式与命令参照 使用说明 部分。

4.3 问题与解决

在实验过程中，主要遇到了以下问题：

- 在文件inode缓存时没有初始化以及没有及时更新缓存一致性位图，导致缓存中的inode与磁盘中的inode不一致，从而出现了一些问题。
- 目录解析过程中，对于绝对路径的拼接错误，导致了目录解析错误。

解决方式主要是通过调试程序，查看输出信息，定位问题所在，然后进行修改。

5. 总结

本次实验主要实现了一个简单的文件系统，包括了文件系统的新建、载入、格式化、安装、保存、卸载等操作，以及文件系统的文件操作、目录操作、用户操作、权限操作等功能。在实验过程中，主要遇到了一些问题，如文件系统的新建与载入、文件描述符的分配与释放、inode缓存的实现等问题。通过调试程序，查看输出信息，定位问题所在，然后进行修改，最终实现了一个简单的文件系统。

本次实验的主要收获有：

1. 熟悉了文件系统的基本组成，包括超级块、inode节点、数据块、目录项等，并基于这些组成实现了文件系统的新建、载入、格式化、安装、保存、卸载等操作，以及一些其他的基础操作例如inode块与数据块的分配与释放、查找空闲inode等。
2. 熟悉了文件系统的文件与目录的实现，包括目录项的增删改查，索引的实现、文件的描述符、打开文件表、文件指针等。
3. 在基本的文件系统操作的基础上，附加了使用LRU算法实现的inode缓存，添加了必要的帮助信息，实现了简单的命令行接口。