**北 京 林 业 大 学**

**2023 — 2024学年第 2 学期 操作系统A 实习报告书**

实习步骤、方法、技术方案、结果、分析：

目录

[一、 文件系统格式 3](#_Toc169641067)

[二、 功能实现： 4](#_Toc169641068)

[1. 登录逻辑的实现： 4](#_Toc169641069)

[2. 注册的相关逻辑： 6](#_Toc169641070)

[3. 文件类命令： 7](#_Toc169641071)

[3.1create创建文件： 7](#_Toc169641072)

[3.2delete删除文件 10](#_Toc169641073)

[3.3 open 打开文件 11](#_Toc169641074)

[3.4 close 关闭文件： 14](#_Toc169641075)

[3.5 read 读文件： 15](#_Toc169641076)

[3.6 write写文件 17](#_Toc169641077)

[3.7 head –num 显示文件的前num行 20](#_Toc169641078)

[3.8 tail -num 显示文件尾巴上的num行 21](#_Toc169641079)

[3.9 lseek 文件读写指针 22](#_Toc169641080)

[3.10 flock 文件加锁，需要实现加锁和解锁功能 24](#_Toc169641081)

[3.11 move 移动文件 26](#_Toc169641082)

[3.12 copy 拷贝文件 29](#_Toc169641083)

[4. 目录类命令： 32](#_Toc169641084)

[4.1 cd 进入目录 32](#_Toc169641085)

[4.2 dir 显示当前目录 35](#_Toc169641086)

[4.3 mkdir 创建目录 36](#_Toc169641087)

[4.4 rmdir 删除目录 38](#_Toc169641088)

[5. 导入文件 import 39](#_Toc169641089)

[6. 导出文件 export 41](#_Toc169641090)

[7. 多线程实现： 42](#_Toc169641091)

[三、 流程图 44](#_Toc169641092)

[1. 总体的流程图 44](#_Toc169641093)

[2. 前端后端流程图 45](#_Toc169641094)

[四、 主要数据结构、函数说明： 45](#_Toc169641095)

[1. Fcb： 45](#_Toc169641096)

[2. User 46](#_Toc169641097)

[3. 全局变量： 46](#_Toc169641098)

[4. 底层函数实现 47](#_Toc169641099)

[4.1首先是初始化相关的数组和变量 47](#_Toc169641100)

[4.2创建文件系统 47](#_Toc169641101)

[4.3 保存用户数据 49](#_Toc169641102)

[4.4 保存文件分配表数据 50](#_Toc169641103)

[4.5 保存位示图 50](#_Toc169641104)

[4.6 更新数据(update函数) 51](#_Toc169641105)

[4.7 保存文件系统 53](#_Toc169641106)

[五、 实验结果截图： 54](#_Toc169641107)

[1. 登录界面： 54](#_Toc169641108)

[2. 注册逻辑： 55](#_Toc169641109)

[3. Create指令： 55](#_Toc169641110)

[4. Delete命令： 55](#_Toc169641111)

[5. Open命令： 55](#_Toc169641112)

[6. Close命令： 56](#_Toc169641113)

[7. Read命令： 56](#_Toc169641114)

[8. Write命令（覆盖写）： 56](#_Toc169641115)

[9. Write命令（追加写）： 57](#_Toc169641116)

[10. Head-num读取文件前num行 57](#_Toc169641117)

[11. Tail-num，读取文件后num行 58](#_Toc169641118)

[12. Lseek读写指针： 58](#_Toc169641119)

[13. Flock，实现对文件的锁定 59](#_Toc169641120)

[14. Dir显示当前目录： 59](#_Toc169641121)

[15. mkdir创建目录： 59](#_Toc169641122)

[16. Cd进入目录： 60](#_Toc169641123)

[17. rmdir删除目录： 60](#_Toc169641124)

[18. Move 移动文件： 60](#_Toc169641125)

[19. Copy 复制文件： 61](#_Toc169641126)

[20. import导入文件: 61](#_Toc169641127)

[21. export导出文件： 61](#_Toc169641128)

[22. 多线程执行： 62](#_Toc169641129)

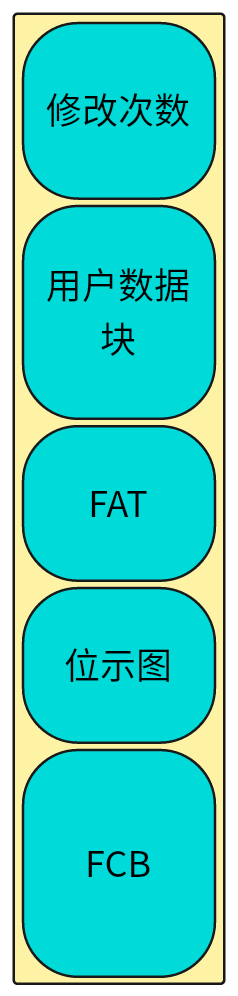
[六、 实验分析及感受 63](#_Toc169641130)

# 文件系统格式

使用txt 文件模拟为系统硬盘，**以二进制的形式**写入和读取文件，划分总容量为10MB，其中前1MB 为系统目录部分，后9MB 空间用于存放文件。



目录空间划分如下：



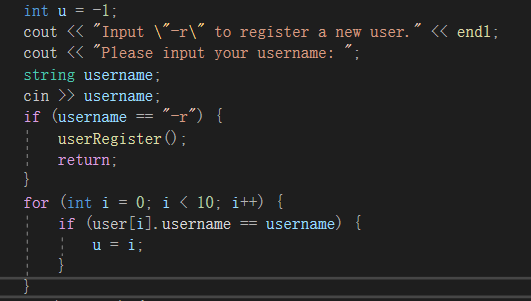
# 功能实现：

## 登录逻辑的实现：

首先是在run函数中检查是否登录，如果没有登录的话，执行登录的函数userLogin，其具体实现的逻辑如下：

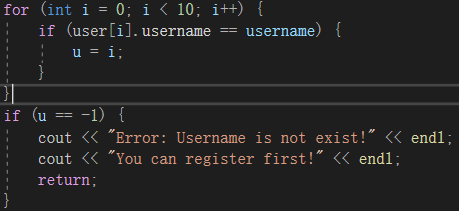
**提示用户输入用户名：**

* 提示用户输入用户名。如果用户输入 -r，调用注册函数 userRegister。



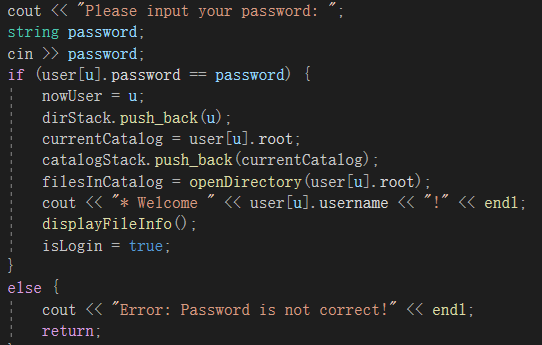
**验证用户名是否存在：**

* 遍历用户数组 user，检查输入的用户名是否存在。
* 如果用户名不存在，输出错误信息并提示用户可以注册。



**提示用户输入密码：**

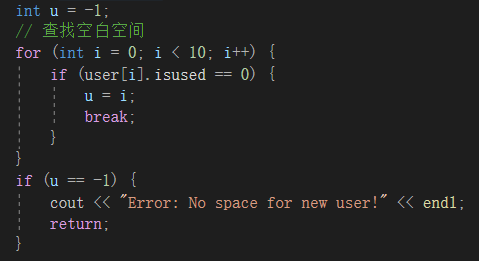
* 提示用户输入密码，并验证密码是否正确。
* 如果密码正确，设置当前登录用户 nowUser，初始化用户的目录信息，并显示欢迎信息和当前目录的文件信息。
* 如果密码不正确，输出错误信息。



## 注册的相关逻辑：

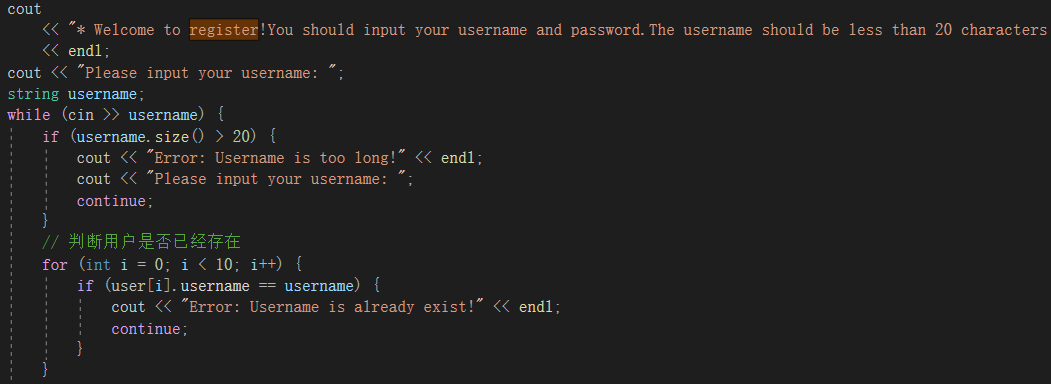
**查找空白空间：**

* 遍历用户数组 user，查找第一个未使用的位置 u。
* 如果没有空白空间，输出错误信息并返回。



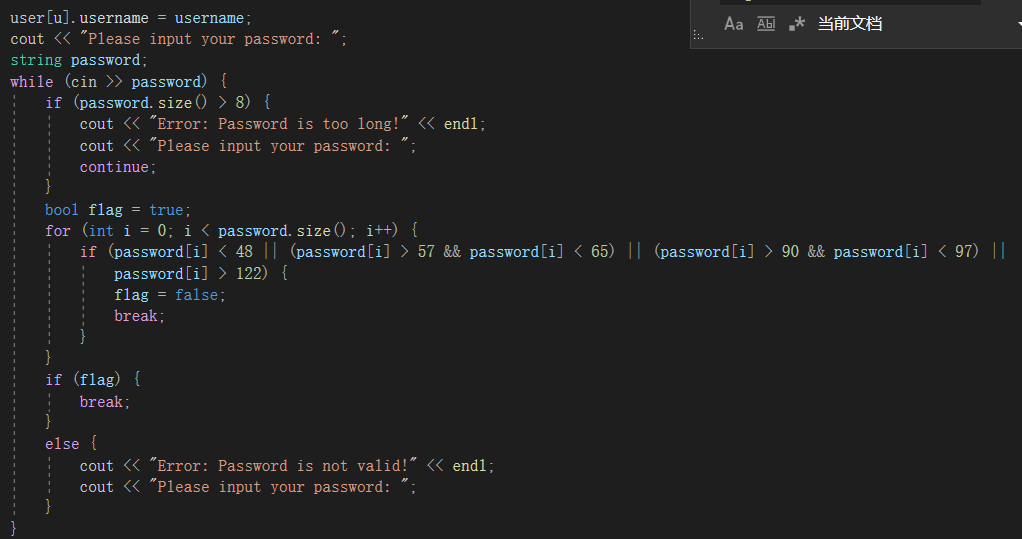
**提示用户输入用户名：**

* 提示用户输入用户名，并进行合法性检查。
* 检查用户名长度是否超过 20 字符。
* 检查用户名是否已存在。
* 检查用户名是否仅包含字母和数字。



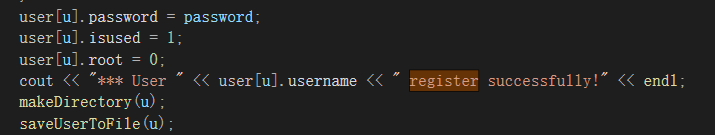
**提示用户输入密码：**

* 提示用户输入密码，并进行合法性检查。
* 检查密码长度是否超过 8 字符。
* 检查密码是否仅包含字母和数字。



**注册新用户：**

* 设置新用户的用户名、密码和根目录信息。
* 创建用户的根目录并保存用户信息。



## 文件类命令：

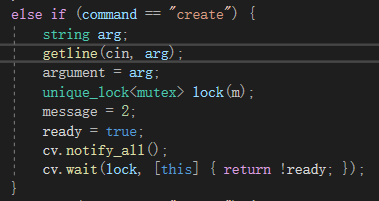
### 3.1create创建文件：

用法：在命令中输入create以及文件名

功能：创建一个文件。

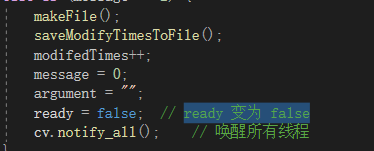
实现的具体步骤：

首先在run函数中，检测输入的字符串是否为“create”，如果是的话，将传递的变量message变成2，唤醒所有线程，执行kernel中的相关函数。



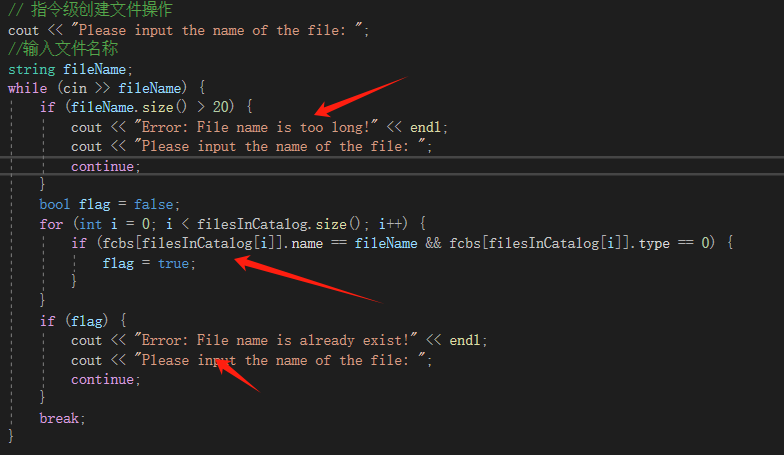
然后在kernel函数中，调用makeFile和saveModifyTimesToFile函数实现创建文件的功能。

增加修改的次数，同时ready 变为 false，唤醒所有线程。

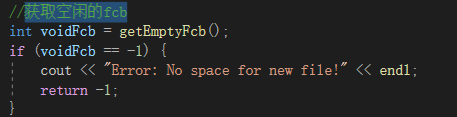


首先要求用户输入新创建的文件名称：

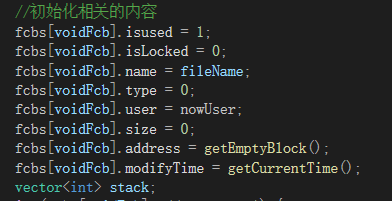
如果输入名称太长，或者是有重名的文件，那就提示用户重新输入。



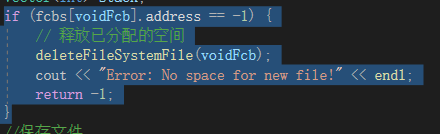
从现有的空间中获取空闲的fcb块，



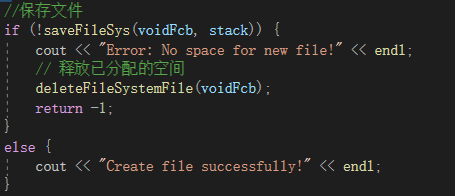
初始化这个块的相关内容：



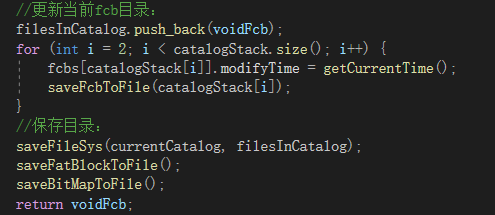
如果这个块的地址已经被占据，那么，就释放掉已经分配的空间：



保存文件：



更新当前的fcb目录并且保存目录。



### 3.2delete删除文件

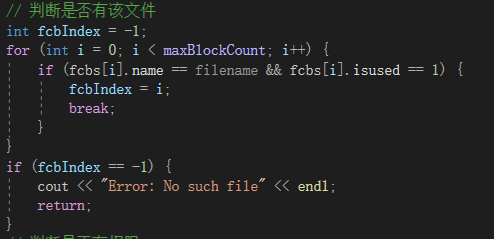
首先是从用户输入的字符串中读取，是否有delete的字符：

如果有，去除这个字符串开头部分的空格，调用delete函数执行删除文件的操作：



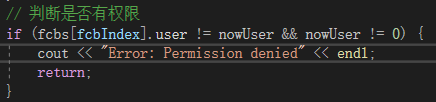
**查找文件：**

* 遍历文件控制块数组 fcbs，找到与指定文件名匹配且标记为已使用的文件。
* 如果找到，保存文件控制块的索引；如果没有找到，输出错误信息并返回。



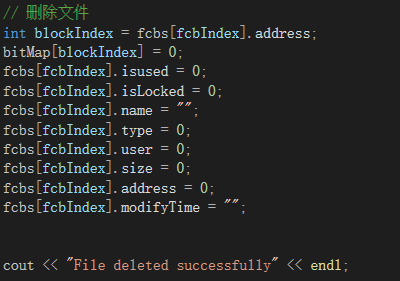
**权限检查：**

* 检查当前用户是否有权限删除文件。如果当前用户不是文件的所有者且不是超级用户，输出错误信息并返回。



**删除文件：**

* 将文件对应的位图块标记为未使用。
* 重置文件控制块的所有属性，表示该文件控制块未被使用。
* 输出文件删除成功的信息。



### open 打开文件

首先是从用户输入的字符串中读取，是否有open的字符：

如果有，去除这个字符串开头部分的空格，将其设置为参数，调用openFileMode函数执行打开文件的操作：



然后执行这个函数。

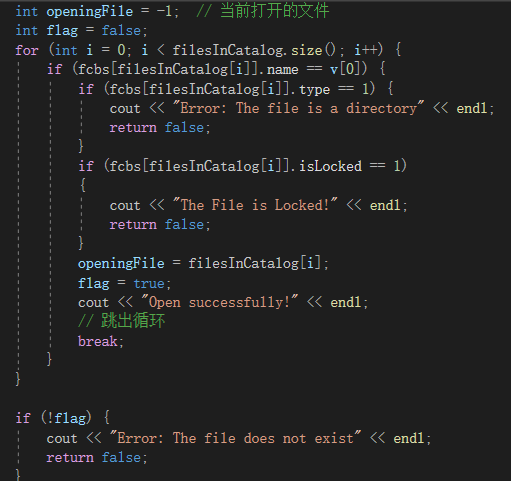
1. **首先解析参数：**

* 将 arg 字符串通过空格拆分成一个字符串向量 v，其中第一个元素是文件名。



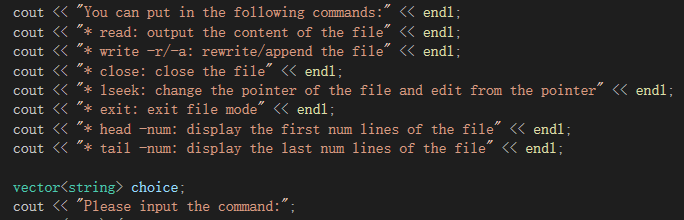
1. **查找文件：**

* 遍历目录中的文件列表 filesInCatalog，找到与指定文件名匹配且不是目录的文件控制块 fcbs。
* 如果找到文件且文件未被锁定，则标记为打开并输出打开成功的信息。
* 如果未找到文件，输出错误信息并返回 false。

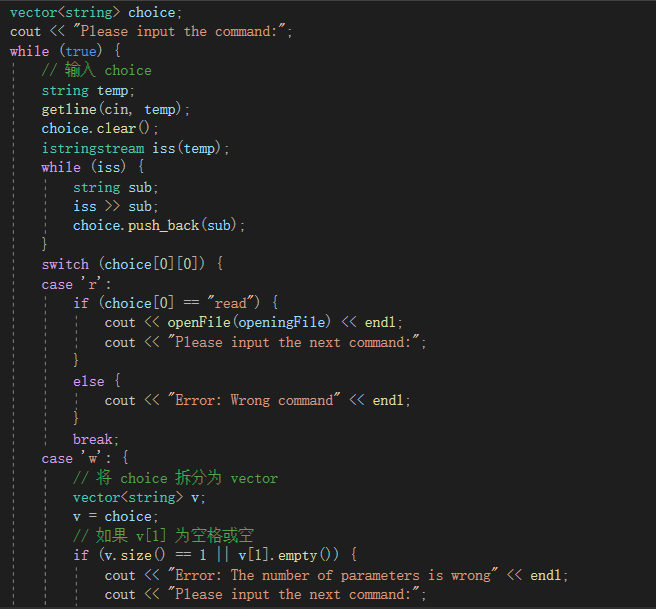


1. **命令模式：**

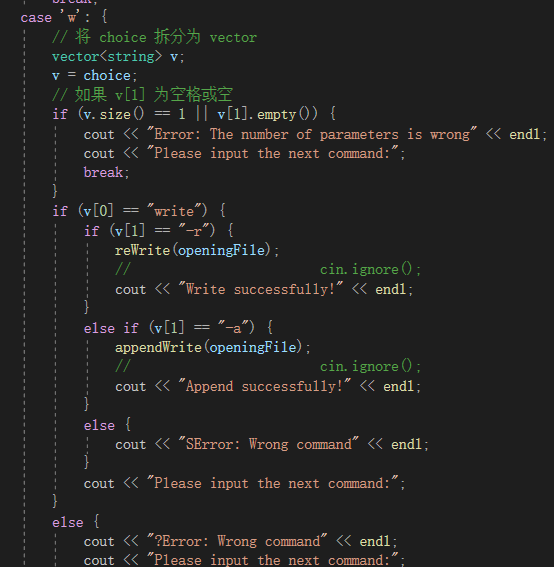
* 提供用户一系列可以对文件进行操作的命令，如读取、写入、关闭等。



* 使用 while 循环不断读取用户输入的命令，根据不同命令执行相应的文件操作。



* 对于每个命令，首先检查命令格式是否正确，然后执行相应的操作，并提示用户输入下一条命令。

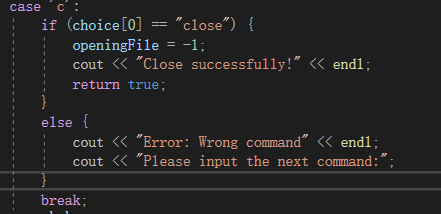


**具体的每个函数实现的功能和命令在如下的菜单中会显示！**

### close 关闭文件：

在open已经打开文件的模式下运行：

如果检测到输入的事close这个字符串，那么返回true，该文件系统退回上一层目录。



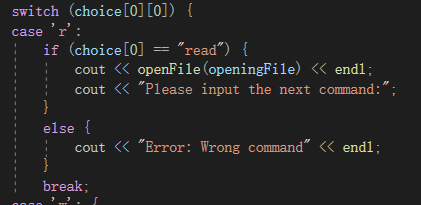
如果输入的不是close，则显示用户输入错误，请求用户重新输入！

### 3.5 read 读文件：

在open已经打开文件的模式下运行：

如果检测到输入的字符串是read，则调用openFile函数，读取其中的内容。

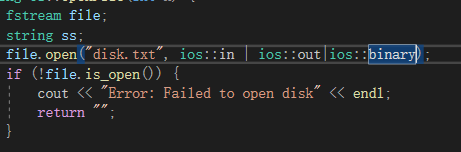
否则会输出错误的指令，要求用户重新输入。



OpenFile函数的实现步骤：

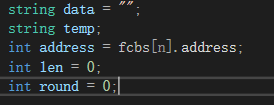
**打开磁盘文件：**

使用 fstream 打开 disk.txt 文件进行读写操作。如果打开失败， 输出错误信息并返回空字符串。



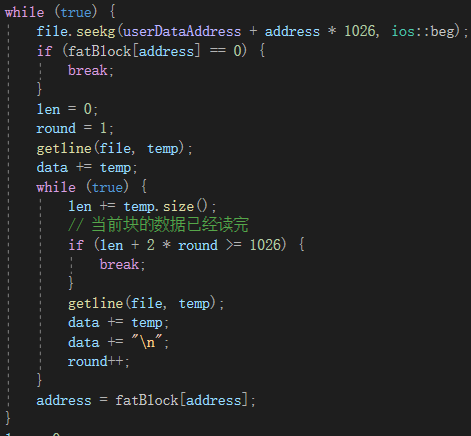
**初始化变量：**

* data 用于存储读取到的文件内容。
* temp 用于存储每次从文件中读取的一行数据。
* address 存储当前读取块的地址，初始值为文件控制块中存储的文件起始地址。
* len 和 round 用于计算当前块中已经读取的数据长度和读取的行数。



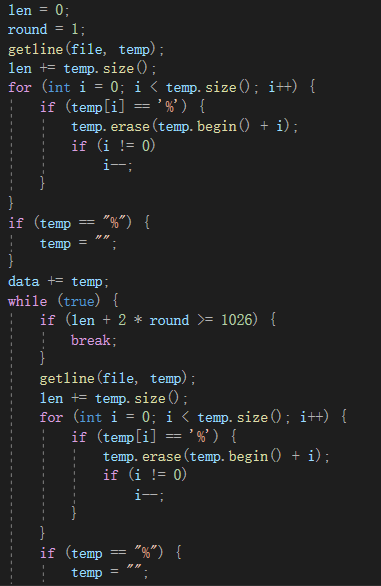
**读取文件内容：**

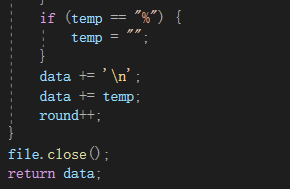
* 循环读取文件块数据，直到 FAT 表中对应的地址为 0（即文件结束）。
* 将当前块的内容读取并拼接到 data 中。
* 每块数据的读取以 1026 字节为限，逐行读取并累积长度，直到达到该块的大小限制。



**处理读取数据：**

* 移除数据中的特殊字符 %。
* 拼接读取的数据并返回。



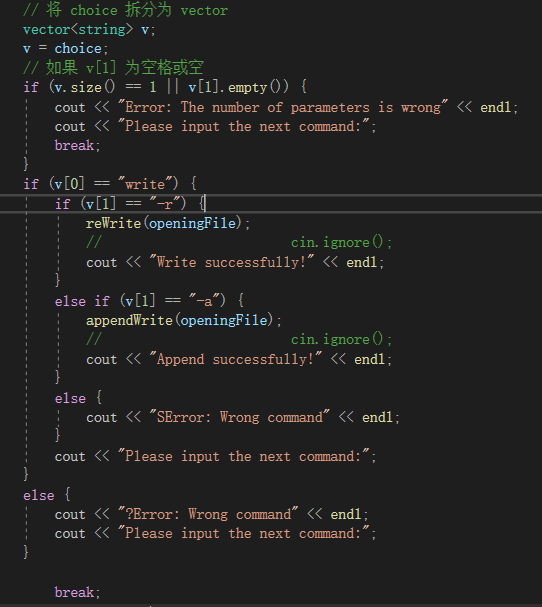


### 3.6 write写文件

在open已经打开文件的模式下运行：

如果检测到输入的字符串是write，则再次识别后面的字符串是-r还是-a，如果是-r，则执行重写的操作，如果是-a则执行追加写的操作。

否则会输出错误的指令，要求用户重新输入。

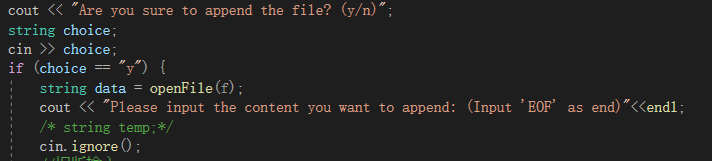


#### 3.6.1重写功能实现：

如果用户输入的是write -r，那么则执行覆盖写操作，执行rewrite函数。

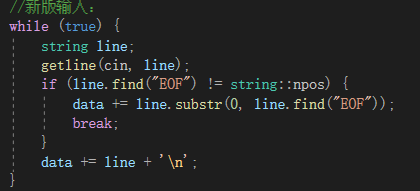
**用户确认：**

* 提示用户确认是否重写文件。如果用户选择 'y'，继续执行；如果选择 'n'，返回 false；否则输出错误信息并返回 false。



**输入新内容：**

* 提示用户输入新的文件内容，并以 "EOF" 作为输入结束标志。
* 使用 getline 循环读取用户输入的每一行，并将其拼接到 data 字符串中。如果检测到 "EOF"，截取 "EOF" 之前的内容并结束输入。



**保存文件内容：**

* 调用 saveFileSys 函数将新的内容保存到文件系统中。如果保存失败，输出错误信息并返回 false。

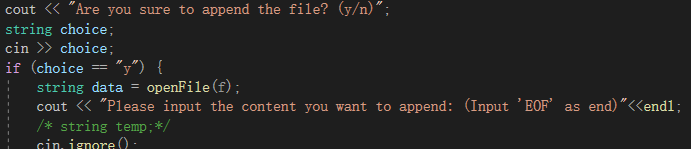


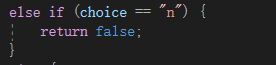
#### 3.6.2追加写功能实现

如果用户输入的是write -a，那么则执行追加写操作，执行rewrite函数。

**用户确认：**

* 提示用户确认是否追加内容。如果用户选择 'y'，继续执行；如果选择 'n'，返回 false；否则输出错误信息并返回 false。





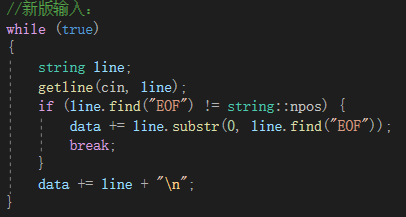
**读取现有内容：**

* 使用 openFile 函数读取文件的现有内容，并将其存储在 data 字符串中。



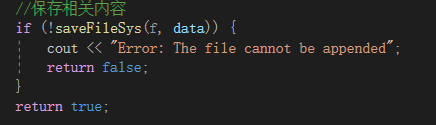
**输入新内容：**

* 提示用户输入要追加的内容，并以 "EOF" 作为输入结束标志。
* 使用 getline 循环读取用户输入的每一行，并将其拼接到 data 字符串中。如果检测到 "EOF"，截取 "EOF" 之前的内容并结束输入。



**保存文件内容：**

* 调用 saveFileSys 函数将追加后的内容保存到文件系统中。如果保存失败，输出错误信息并返回 false。

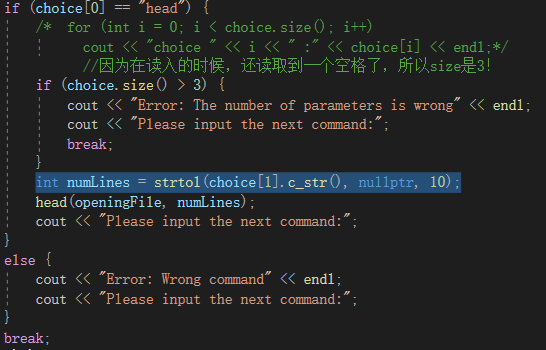


### 3.7 head –num 显示文件的前num行

在open已经打开文件的模式下运行：

如果检测到输入的字符串是head，则识别命令字符串的size是否为3！如果不为3，则会显示输入的参数有误，要求用户重新输入。

同时将字符串中的空格去掉，将字符串中的数字变为int类型，作为参数传递给head函数。



Head函数具体实现如下：

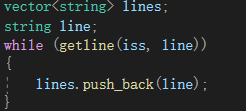
**读取文件内容：**

* 使用 openFile 函数读取文件内容，并将其存储在字符串 data 中。



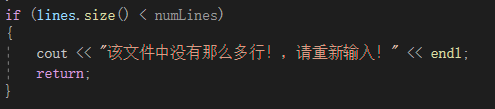
**分割文件内容为行：**

* 使用 istringstream 将 data 分割成多行，并将每一行存储到 vector<string> 类型的 lines 中。



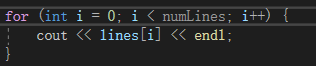
**检查行数是否足够：**

* 检查文件中的行数是否少于指定的 numLines，如果少于，输出错误信息并返回。



**输出前 numLines 行：**

* 循环输出 lines 中的前 numLines 行。

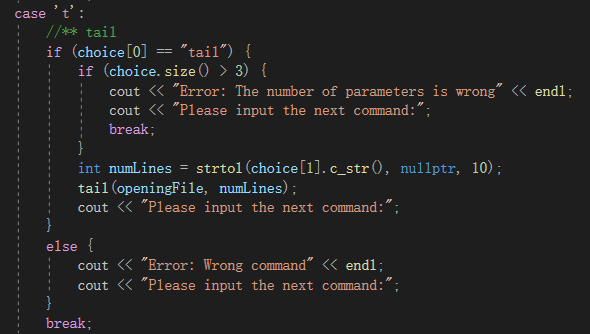


### 3.8 tail -num 显示文件尾巴上的num行

在open已经打开文件的模式下运行：

如果检测到输入的字符串是tail，则识别命令字符串的size是否为3！如果不为3，则会显示输入的参数有误，要求用户重新输入。

同时将字符串中的空格去掉，将字符串中的数字变为int类型，作为参数传递给tail函数。



tail函数实现逻辑：

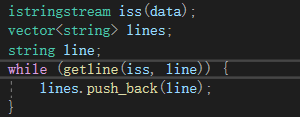
**读取文件内容：**

* 使用 openFile 函数读取文件内容，并将其存储在字符串 data 中。



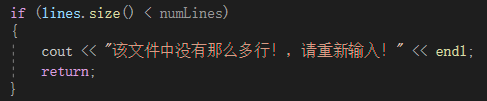
**分割文件内容为行：**

* 使用 istringstream 将 data 分割成多行，并将每一行存储到 vector<string> 类型的 lines 中。



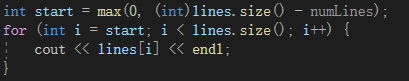
**检查行数是否足够：**

* 检查文件中的行数是否少于指定的 numLines，如果少于，输出错误信息并返回。



**输出最后 numLines 行：**

* 计算开始输出的行的索引，并输出从该索引开始的行。

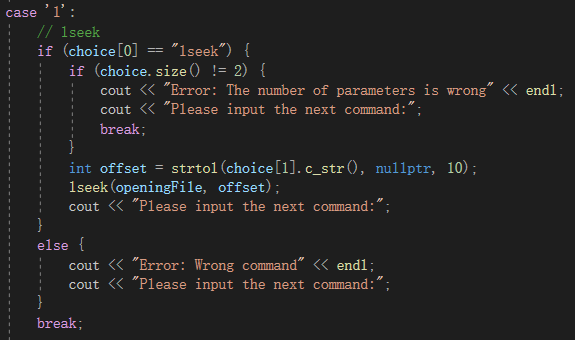


### 3.9 lseek 文件读写指针

在open已经打开文件的模式下运行：

如果检测到这个字符串的size的值不等于2，那么认为输入的函数参数有问题，要求用户重新输入。

否则，将输入的第一个参数转化为int类型的值，然后作为参数，调用lseek函数实现指针的移动操作。



lseek函数实现的具体步骤：

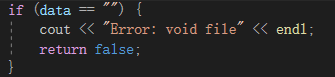
**读取文件内容：**

* 使用 openFile 函数读取文件内容，并将其存储在字符串 data 中。



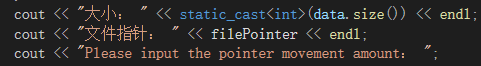
**验证文件内容：**

* 如果文件内容为空，输出错误信息并返回 false。



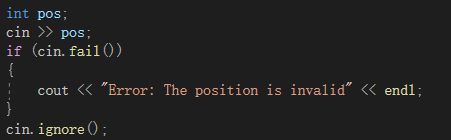
**显示文件大小和当前文件指针：**

* 输出文件大小和当前文件指针位置。



**获取指针移动量：**

* 提示用户输入指针移动量，并验证输入是否有效。
* 计算新的文件指针位置，并处理指针移动越界的情况。



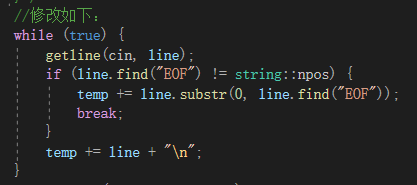
**更新文件指针：**

* 根据用户输入的移动量更新文件指针位置。



**获取新内容：**

* 提示用户输入新的内容，并检测到 "EOF" 结束输入。



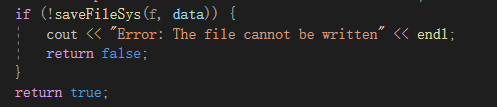
**插入新内容：**

* 将新内容插入到文件内容的指定位置，并显示更新后的内容。



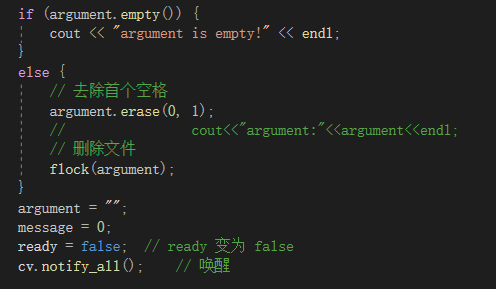
**保存更新后的文件内容：**

* 调用 saveFileSys 函数保存文件内容，如果保存失败，输出错误信息并返回 false。



### 3.10 flock 文件加锁，需要实现加锁和解锁功能

首先是检测输入的字符串中是否有flock 的字符串，如果有的话，将其处理，去掉flock和空格，将这个字符串作为参数传递给flock函数，执行上锁和解锁的操作。



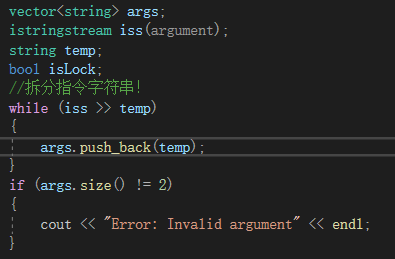
flock函数具体的执行如下：

**解析参数：**

* 将 argument 字符串通过空格拆分成一个字符串向量 args，其中第一个元素是锁定/解锁指令，第二个元素是文件名。

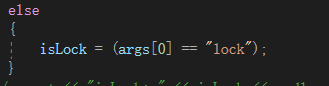
**验证参数数量：**

* 检查 args 的大小是否为 2。如果不是，输出错误信息并返回 false。



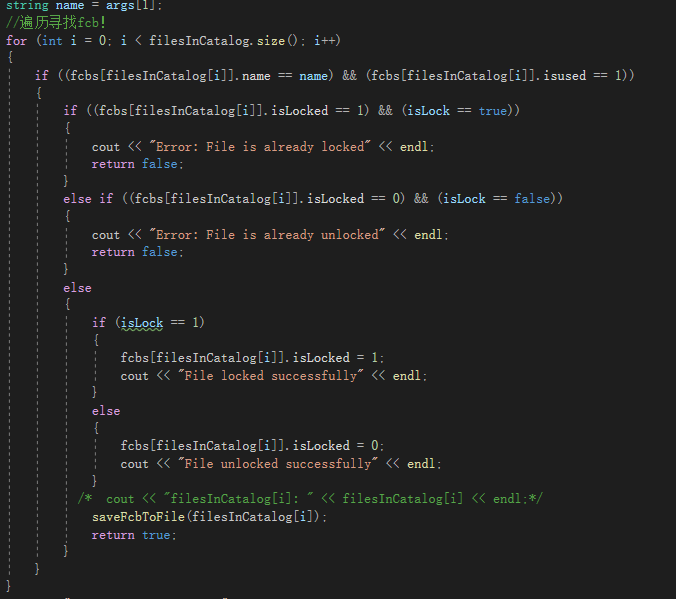
**确定锁定状态：**

* 根据指令（"lock" 或 "unlock"）设置 isLock 布尔变量。



**查找文件：**

* 遍历目录中的文件列表 filesInCatalog，找到与指定文件名匹配且已使用的文件控制块 fcbs。

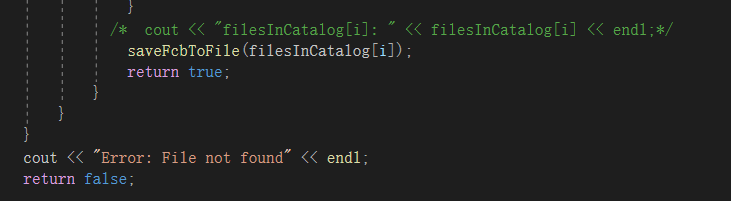


**检查并执行锁定/解锁操作：**

* 如果文件已经处于指定的锁定/解锁状态，输出错误信息并返回 false。
* 否则，根据 isLock 的值更新文件的锁定状态，并输出成功信息。

**保存文件控制块状态：**

* 调用 saveFcbToFile 函数保存文件控制块状态。



### 3.11 move 移动文件

首先是从用户输入的字符串中读取，是否有move的字符：

如果有，对输入的字符串进行拆分，使其分解出来移动的文件名称和移动的路径。

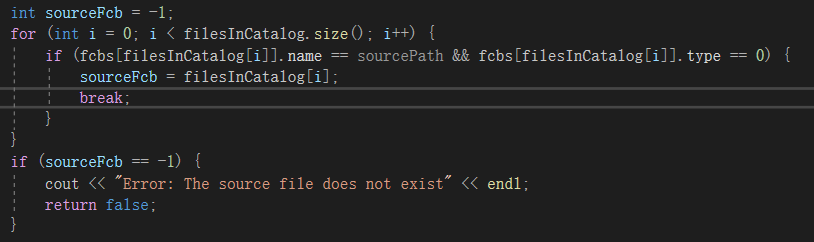
将这两个字符串作为参数，传给moveFile函数。



moveFile的具体实现步骤：

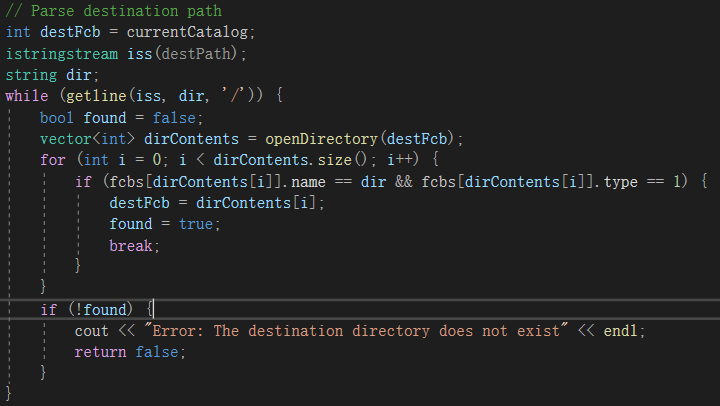
**查找源文件：**

* 遍历 filesInCatalog，找到与 sourcePath 匹配的文件控制块 fcbs。



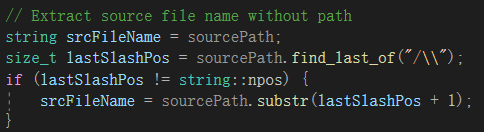
**解析目标路径：**

* 使用 istringstream 将 destPath 分割成目录部分，并依次检查每个目录是否存在。如果某个目录不存在，输出错误信息并返回 false。



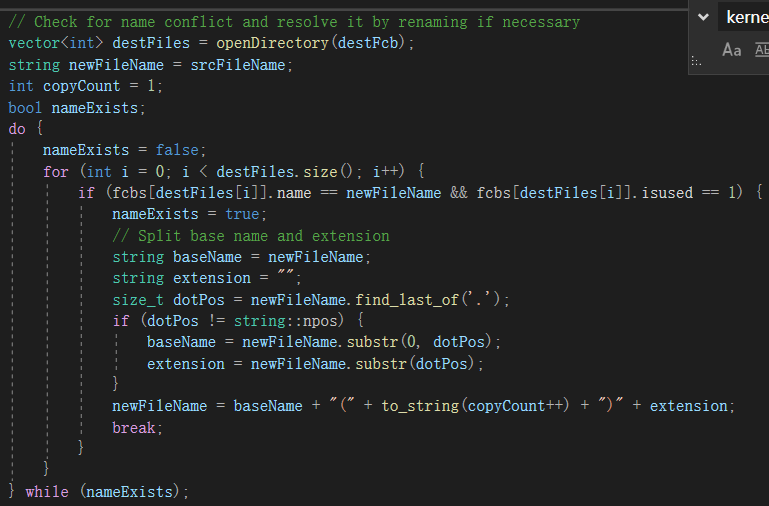
**提取源文件名：**

* 从 sourcePath 中提取文件名。



**检查并解决命名冲突：**

* 遍历目标目录中的文件，如果发现名称冲突，通过添加计数器解决冲突。



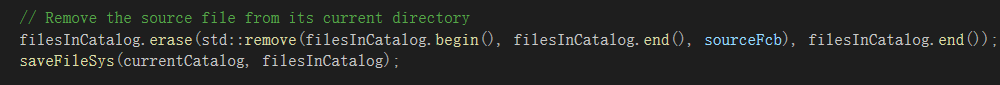
**更新文件控制块的名称：**

* 将文件控制块中的文件名更新为新的文件名。



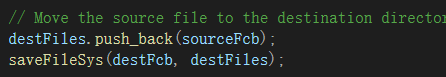
**从源目录中删除文件：**

* 从 filesInCatalog 中删除源文件控制块的索引，并保存目录信息。



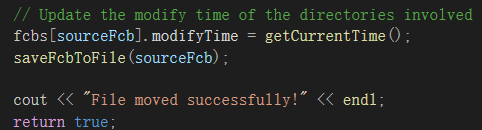
**将文件移动到目标目录：**

* 将源文件控制块的索引添加到目标目录，并保存目录信息。



**更新修改时间：**

* 更新文件控制块的修改时间并保存。



### 3.12 copy 拷贝文件

首先是从用户输入的字符串中读取，是否有copy的字符：

如果有，对输入的字符串进行拆分，去掉除了copy以外的输入的字符串中首个空格。

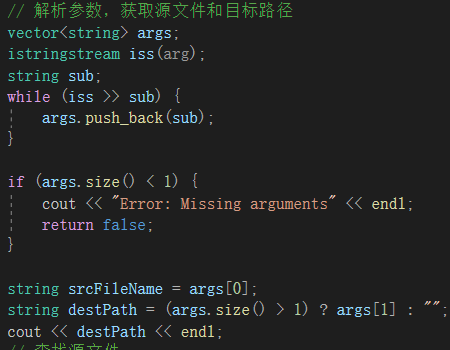
然后将这个字符串作为参数传递，调用copyFile实现拷贝文件的功能。



copyFile函数的实现步骤如下：

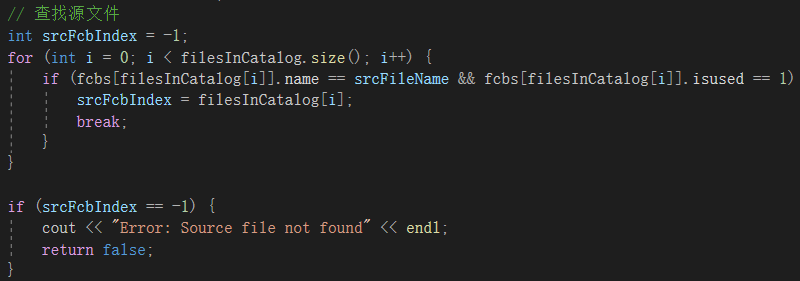
**解析参数：**

* 将 arg 字符串通过空格拆分成 args 向量，其中第一个元素是源文件名，第二个元素是目标路径（如果有）。



**查找源文件：**

* 遍历 filesInCatalog，找到与 srcFileName 匹配且标记为已使用的文件控制块 fcbs。



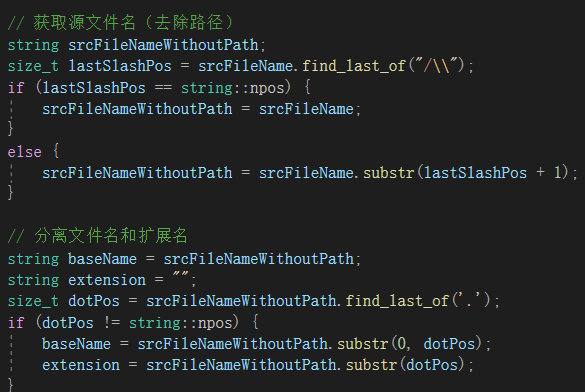
**读取源文件内容：**

* 使用 openFile 函数读取源文件内容，并将其存储在字符串 fileContent 中。



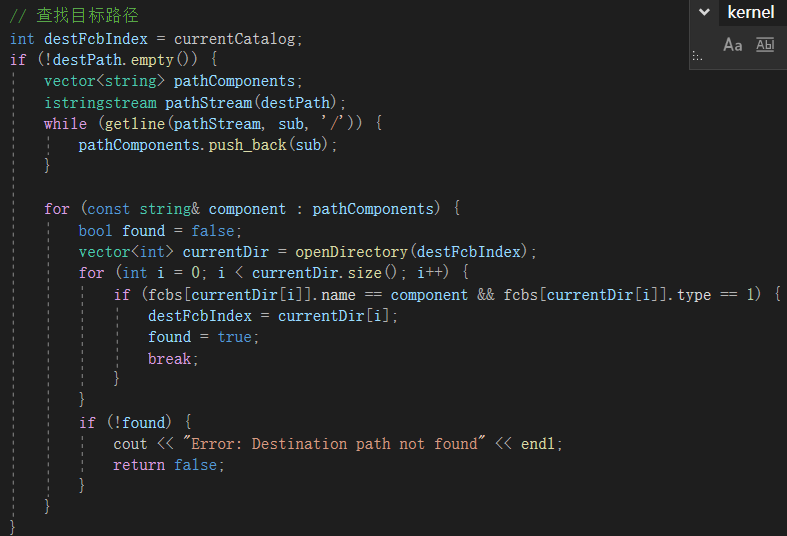
**解析源文件名：**

* 提取源文件名中的基本名和扩展名。



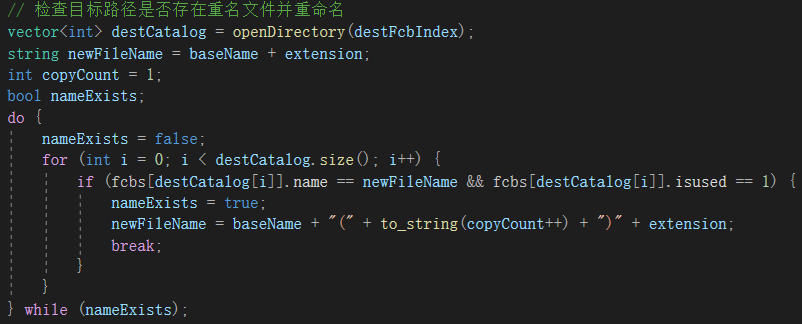
**查找目标路径：**

* 如果提供了目标路径，解析路径并遍历每个目录部分，检查目录是否存在。



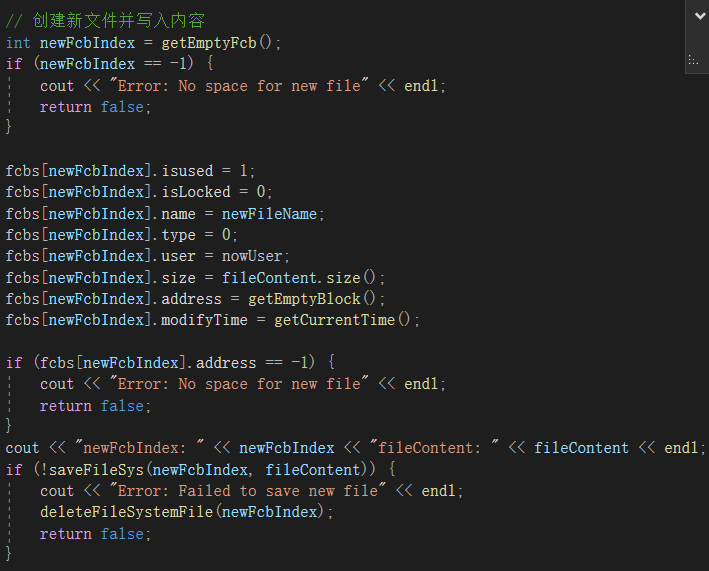
**检查并解决命名冲突：**

* 在目标目录中检查是否存在同名文件，并通过添加计数器解决命名冲突。



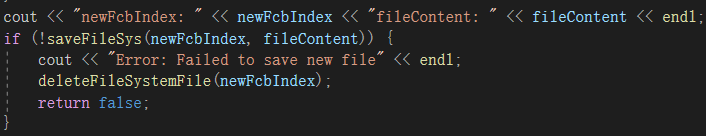
**创建新文件控制块：**

* 获取一个空闲的文件控制块 newFcbIndex，并将文件控制块的属性设置为新文件的信息。



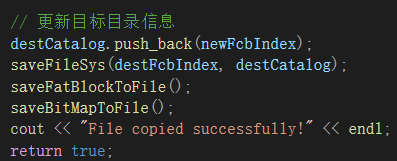
**写入文件内容：**

* 调用 saveFileSys 函数将新文件内容保存到文件系统中。如果保存失败，删除新文件并返回 false。



**更新目标目录信息：**

* 将新文件控制块的索引添加到目标目录，并保存目录信息。

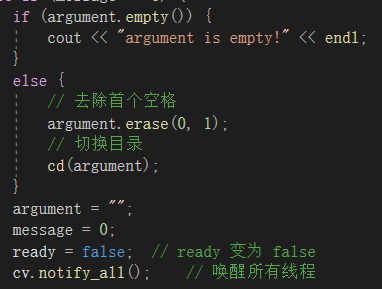


## 目录类命令：

### 4.1 cd 进入目录

首先是从用户输入的字符串中读取，是否有cd的字符：

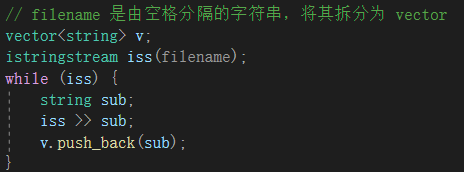
如果有，去除这个字符串开头部分的空格，调用cd函数执行删除文件的操作：将用户输入的剩余的字符串作为参数传递。



cd函数实现的具体步骤如下：

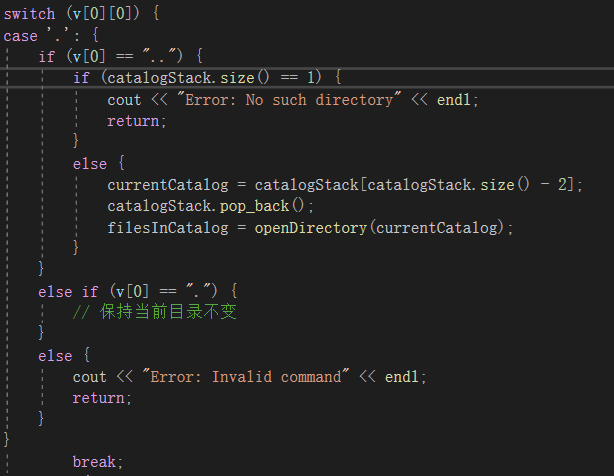
**解析参数：**

* 将输入的 filename 字符串通过空格拆分成 v 向量。



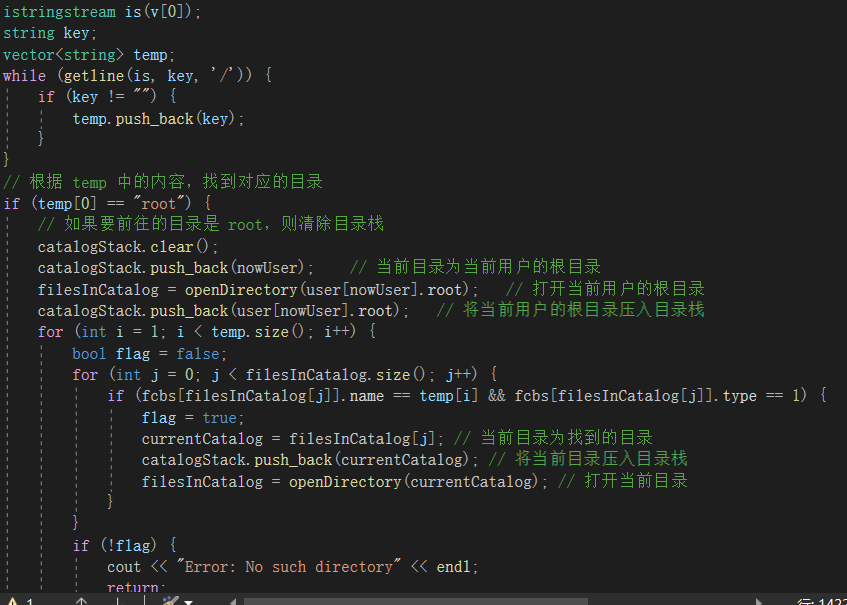
**处理特殊目录命令：**

* 如果输入为 .. ，则返回上一级目录。
* 如果输入为 . ，则保持当前目录不变。
* 如果输入不为上述特殊命令，解析路径并导航到目标目录。



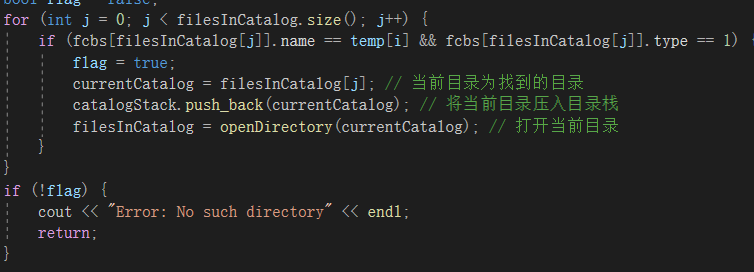
**解析路径并导航：**

* 解析路径字符串，将路径分割为各级目录。
* 如果路径起始为 root，则从根目录开始导航。
* 否则，从当前目录开始导航。



**检查并更新目录：**

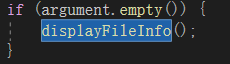
* 遍历每一级目录，检查目录是否存在且为有效目录。
* 如果目录存在，更新当前目录和目录栈。
* 如果目录不存在，输出错误信息并返回。

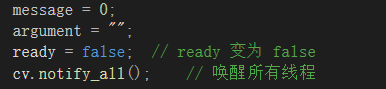


### 4.2 dir 显示当前目录

首先是从用户输入的字符串中读取，是否有dir的字符：

如果有，直接调用displayFileInfo函数，显示当前目录下的文件列表。





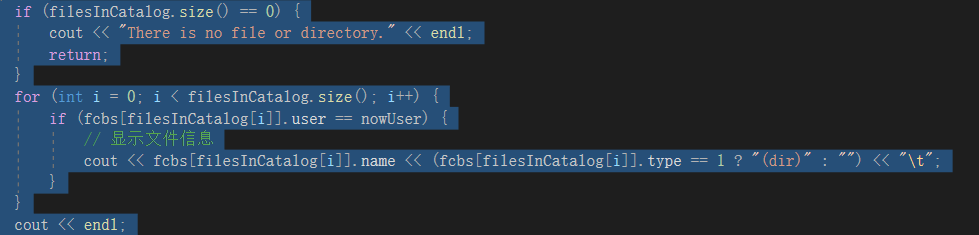
displayFileInfo函数的具体实现步骤：

**检查当前目录是否为空：**

* 如果 filesInCatalog 为空，输出 "There is no file or directory." 并返回。

**遍历当前目录文件：**

* 遍历 filesInCatalog，如果文件或目录的所有者是当前用户，显示其名称和类型（目录标记为 "(dir)"）。

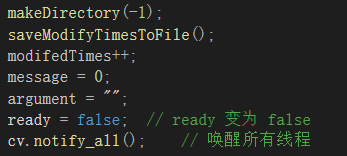


### mkdir 创建目录

首先是从用户输入的字符串中读取，是否有mkdir的字符：

如果有，直接调用makeDirectory()和saveModifyTimesToFile()函数。

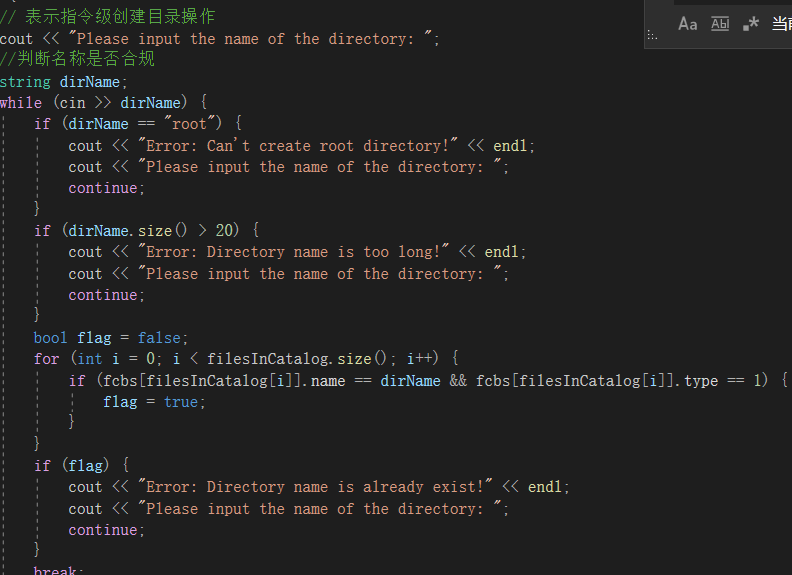
在调用makeDirectory()函数的时候，以-1作为参数。



makeDirectory()函数的实现步骤：

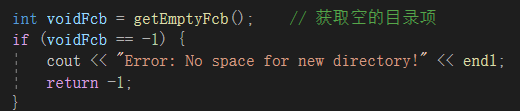
**获取用户输入的目录名称：**

* 提示用户输入目录名称，并进行合法性检查。
* 检查目录名称是否为 "root" 或长度是否超过 20 个字符。
* 检查当前目录下是否存在同名目录。



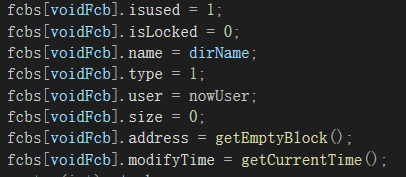
**获取空的文件控制块（FCB）：**

* 调用 getEmptyFcb 获取一个空闲的文件控制块索引。
* 如果没有空闲的文件控制块，输出错误信息并返回。



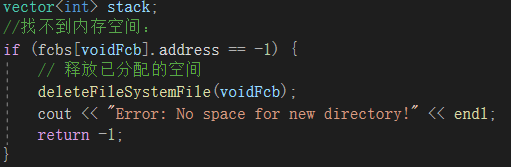
**初始化目录的文件控制块：**

* 设置文件控制块的属性（如名称、类型、用户、大小、地址、修改时间等）。



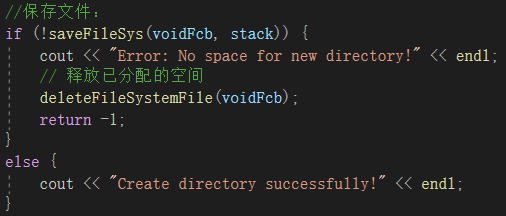
**获取空闲的磁盘块：**

* 调用 getEmptyBlock 获取一个空闲的磁盘块地址。
* 如果没有空闲的磁盘块，释放已分配的文件控制块并输出错误信息。



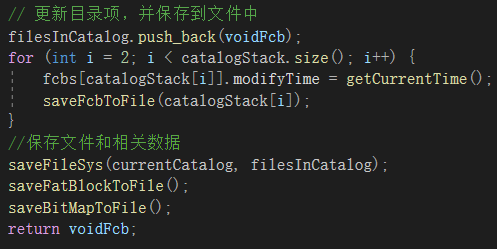
**保存新目录的信息：**

* 将新目录的信息保存到文件系统中。
* 如果保存失败，释放已分配的资源并输出错误信息。



**更新当前目录的信息：**

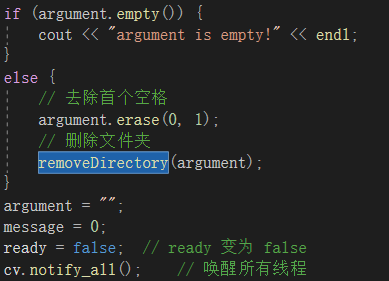
* 将新目录添加到当前目录的目录项中。
* 更新当前目录及其父目录的修改时间并保存。



### 4.4 rmdir 删除目录

首先是从用户输入的字符串中读取，是否有rmdir的字符：

如果在rmdir字符串之后还有参数，则将位于用户输入的字符串的首个空格去掉，将剩下的内容作为参数传递给removeDirectory这个函数去执行。



removeDirectory这个函数实现的具体步骤如下：

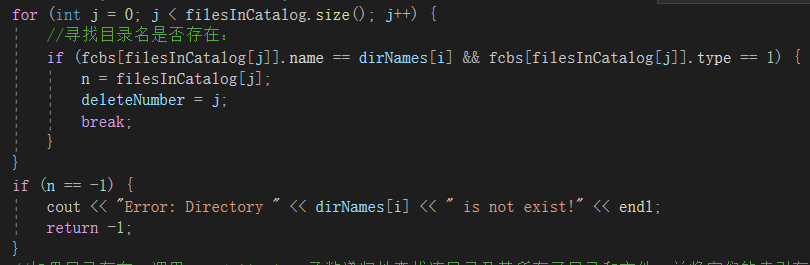
**解析目录名：**

* 将输入的目录名字符串通过空格分隔并转换为字符串向量 dirNames。



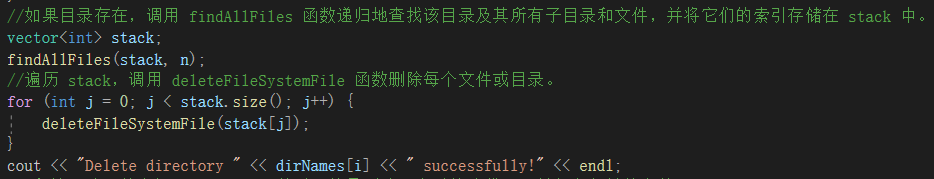
**检查并删除目录：**

* 遍历 dirNames，检查每个目录是否存在于当前目录中。
* 如果目录存在，递归删除该目录及其所有子目录和文件。
* 如果目录不存在，输出错误信息并返回。



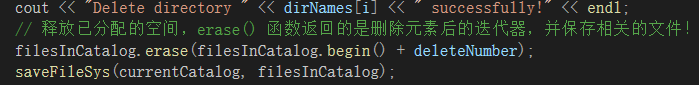
**递归删除目录及其子目录和文件：**

* 使用 findAllFiles 函数递归查找目录及其所有子目录和文件，并将它们的索引存储在 stack 中。
* 遍历 stack，调用 deleteFileSystemFile 函数删除每个文件或目录。



**更新目录信息：**

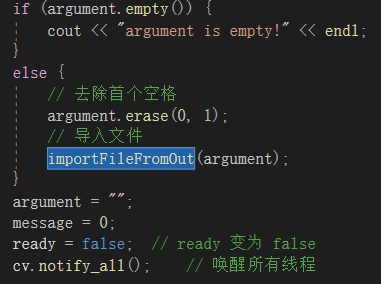
* 删除成功后，从当前目录的文件列表中移除该目录，并保存目录信息。



## 导入文件 import

首先是从用户输入的字符串中读取，是否有import的字符：

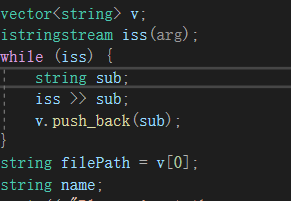
如果有，去除这个字符串开头部分的空格，将其设置为参数，调用importFileFromOut函数执行打开文件的操作：



函数的实现步骤如下：

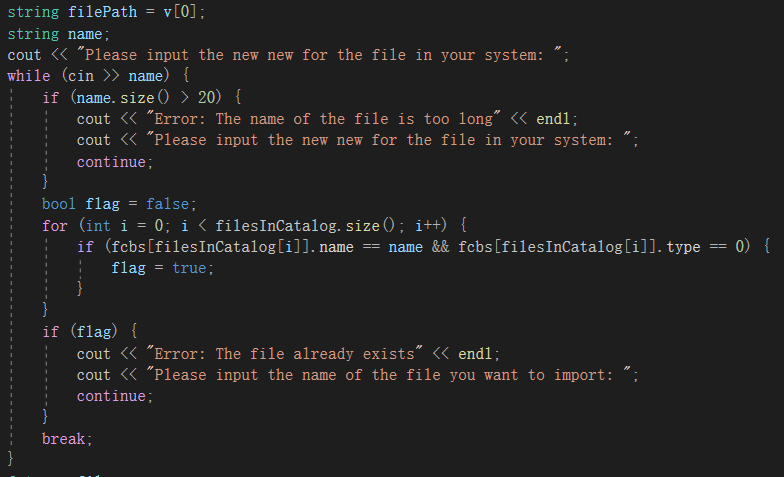
**解析参数：**

* 将 arg 字符串通过空格拆分成 v 向量，其中第一个元素是源文件路径。



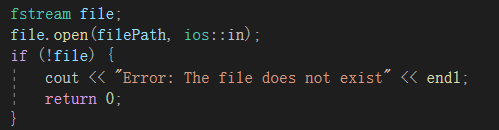
**获取新文件名：**

* 提示用户输入新文件名，并进行合法性检查（长度是否超过 20 字符，是否与当前目录下已有文件重名）。



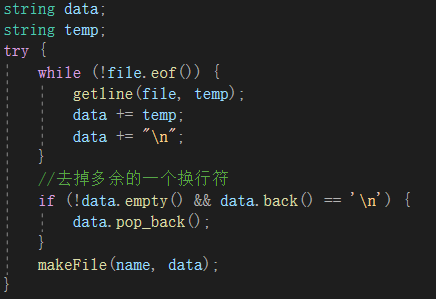
**打开外部文件：**

* 尝试打开外部文件进行读取，如果文件不存在，输出错误信息并返回。



**读取文件内容：**

* 逐行读取文件内容，并将其存储在 data 字符串中。
* 去掉读取内容中的最后一个多余换行符。



**创建新文件并保存内容：**

* 调用 makeFile 函数在当前文件系统中创建新文件并保存读取的内容。



## 导出文件 export

首先是从用户输入的字符串中读取，是否有export的字符：

如果有，去除这个字符串开头部分的空格，将其设置为参数，调用exportFileToOut函数执行打开文件的操作：



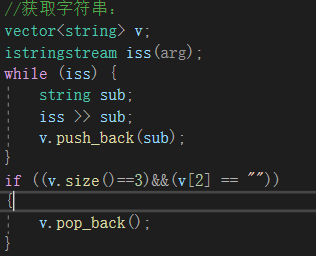
exportFileToOut函数的具体实现步骤如下：

**解析参数：**

* 将 arg 字符串通过空格拆分成 v 向量，其中第一个元素是源文件名，第二个元素是目标路径（如果有）。

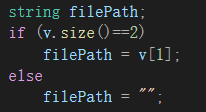
**检查参数数量：**

* 如果 v 的大小超过 2，则输出错误信息并返回。



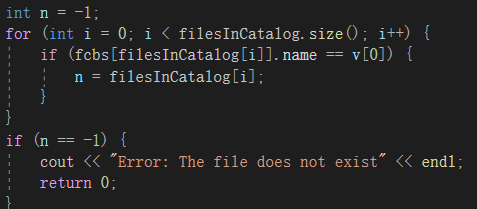
**获取目标文件路径：**

* 如果提供了目标路径，则使用该路径；否则，使用默认路径。



**查找源文件：**

* 遍历 filesInCatalog，找到与 v[0] 匹配的文件控制块 fcbs。

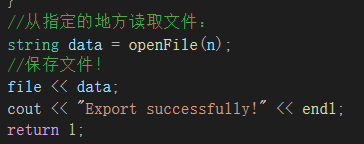


**读取源文件内容：**

* 调用 openFile 函数读取源文件内容，并将其存储在字符串 data 中。

**创建外部文件并保存内容：**

* 在外部文件系统中创建一个新文件，并将读取的内容写入该文件中。



## 多线程实现：

首先是我设计的函数框架就是为了多线程服务。几个主函数如下：

1. **main() 函数：**

* 创建两个线程 t1 和 t2，分别执行 os 类的成员函数 run() 和 kernel()。
* 使用 t1.join() 和 t2.join() 等待两个线程执行完毕，并返回 0 结束程序。

1. **os::run() 函数：**

* 接收用户输入和处理命令的线程函数。
* 在无限循环中，不断接收用户命令并执行相应操作。
* 如果用户未登录，则调用 userLogin() 进行验证。
* 如果用户已登录，则输出命令提示符，接收并处理用户命令。

1. **os::kernel() 函数：**

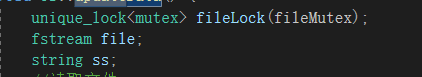
* 处理后台维护虚拟磁盘驱动器操作的线程函数。
* 在无限循环中，通过 cv.wait() 等待命令。
* 一旦有命令到来，检查并更新数据，根据 message 变量执行相应操作。
* 操作完成后，更新状态变量，并通过 cv.notify\_all() 唤醒等待的线程。

通过使用多线程，可以实现系统的并发执行和用户交互。一个线程负责接收用户的命令，并将命令转换为消息，通知后台维护虚拟磁盘驱动器的线程进行相应操作。这样可以提高系统的响应性能，并实现多用户同时访问系统的能力。

为了避免多个线程同时访问文件时的冲突，本系统采用互斥锁机制来实现线程间的同步和互斥访问。通过定义了一个名为fileMutex 的互斥锁对象，并在涉及对文件的操作代码块中使用该锁来确保同一时间只有一个线程可以访问文件。

1. **设置锁：**

* 为避免线程冲突，采用互斥锁机制实现同步和互斥访问。
* 定义互斥锁对象 fileMutex 确保文件读取时不同函数执行不冲突。
* 使用 mutex m 实现每个命令在一次循环中只能执行一个。



1. **condition\_variable cv:**

* 是 C++11 标准库中的同步原语，用于线程之间的通信。

1. **cv.notify\_all():**

* 唤醒所有等待 cv 的线程，通知某个条件发生变化（ready 变为 true）。

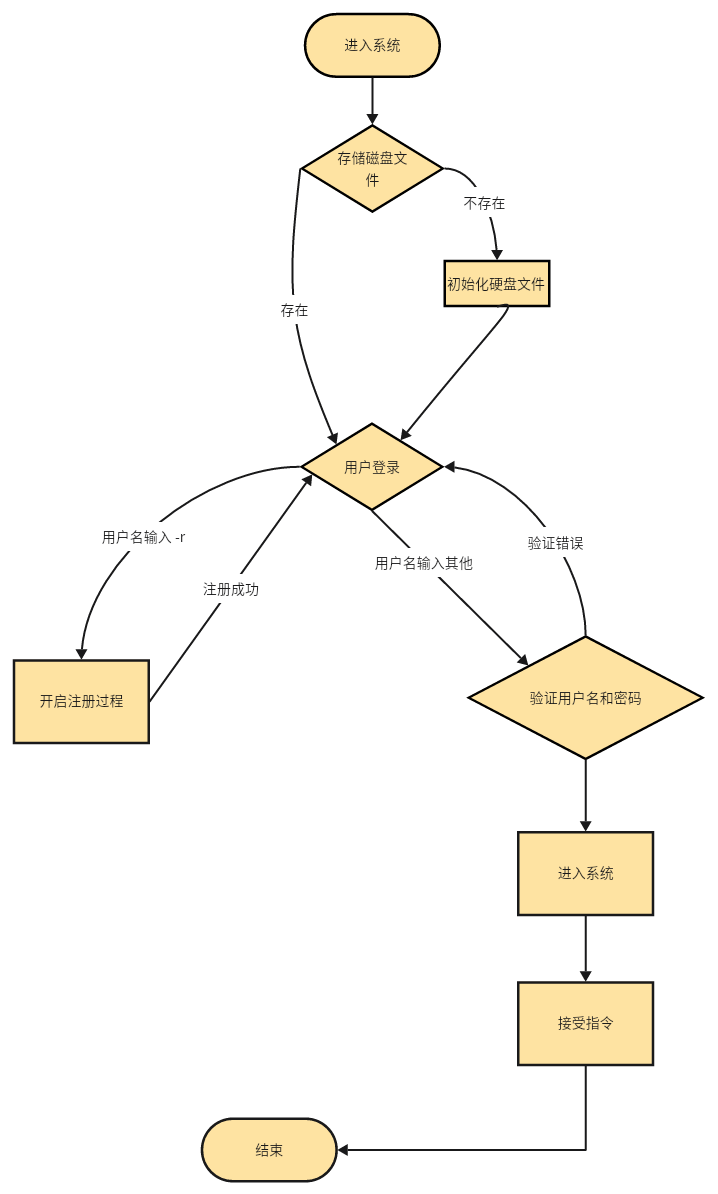
1. **cv.wait(lock, [this] { return !ready; }):**

* 当前线程等待，直到 ready 变为 false。
* 确保线程在等待时持有互斥量 m。
* Lambda 表达式检查条件是否满足，ready 为 false 时等待结束。



# 流程图

## 总体的流程图

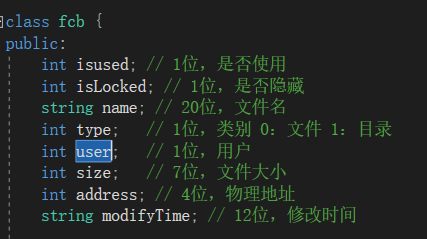


## 前端后端流程图



# 主要数据结构、函数说明：

## Fcb：



FCB（File Control Block，文件控制块）是操作系统中用于管理文件

的数据结构。下面是对FCB 信息的描述：

- isused：表示该FCB 是否被使用，用一个位来表示，1 表示被使用，

0 表示未使用。（用于在之后回收内存）

- isLocked：表示该文件是否被锁定，同样用一个位来表示，1 表示锁定，0 表示不锁定。

- name：文件名，长度为20 位，用于标识文件的名称。

- type：表示文件的类型，用一个位来表示，0 表示文件，1 表示目

录。

- user：表示文件所属的用户。

- size：表示文件的大小，占据7 位，用于记录文件的字节数。

- address：表示文件在磁盘上的物理地址，占据4 位，用于记录文件

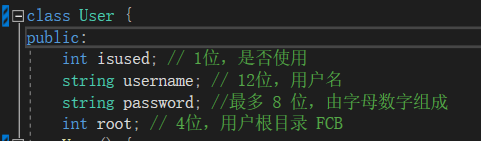
所在的磁盘块号。

- modifyTime：表示文件的修改时间，长度为12 位，用于记录文件

的最后修改时间。

FCB（文件控制块）的功能是保存与文件相关的信息，包括文件名、类型、大小以及物理地址等。操作系统通过访问和操作FCB，实现对文件的读取、写入和删除等操作。

## User



User 类是用于表示系统中的用户信息的数据结构。下面是对User

信息的描述：

- isused：表示该用户是否被使用，用一个位来表示，1 表示被使用，

0 表示未使用。这个是用于检测是否用户数量超标的！

- username：用户名，长度为12 位，用于标识用户的名称。

- password：密码，最多8 位，由字母和数字组成，用于用户进行身

份验证。

- root：用户根目录的FCB（File Control Block），占据4 位，用于记

录用户的根目录在文件系统中的位置。

User类的构造函数用于初始化用户对象的初始值。默认情况下，isused、username、password和root的初始值分别设置为0、000000000000、00000000和0。通过修改这些属性的值，可以管理和跟踪用户的使用状态、身份信息以及用户根目录的位置。

在系统中，User对象用于表示每个用户的信息，包括用户名、密码以及用户根目录的位置。这些信息对于系统的安全性和文件管理至关重要。通过访问和操作User对象，可以进行用户身份验证和管理用户根目录。

## 全局变量：

- userDataAddress: 存储用户数据区的起始地址。

- maxBlockCount: 定义了文件系统中的最大块数。

- maxUserCount: 定义了系统中允许的最大用户数。

- modifedTimesLength: 指定了修改次数的长度。

- userLength: 指定了用户数据的长度，用于存储用户信息。

- fatLength: 指定了文件分配表（FAT）的长度。

- bitMapLength: 指定了位图的长度。

- catalogStack: 使用栈结构存储目录号。

- currentCatalog: 存储当前目录的文件控制块（FCB）号。

- filesInCatalog: 存储当前目录下的文件号列表。

- user: 指向用户对象的指针。

- nowUser: 存储当前用户的索引。

- fileMutex: 用于文件操作的互斥锁。

- m: 用于其他操作的互斥锁。

- cmd: 存储命令行线程的ID。

- kernels: 存储内核线程的ID。

- modifedTimes: 记录修改次数。

- message: 用于标识指令。

- argument: 存储指令的参数。

- fatBlock: 指向文件分配表块的指针。

- bitMap: 指向位图的指针。

- fcbs: 指向文件控制块数组的指针。

- isLogin: 标识是否已登录。

这些全局变量用于存储和共享相关的状态和数据。它们被多个函数和线程使用，以实现系统的功能和交互。

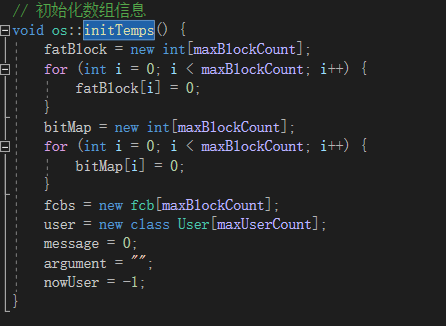
## 底层函数实现

### 4.1首先是初始化相关的数组和变量

1. 初始化一些必要的变量：

* fatBlock（文件分配表）
* bitMap（位图）
* fcbs（文件控制块数组）

1. 将fatBlock和bitMap初始化为全零。
2. 将fcbs和user初始化为对应的数据结构。
3. 将Message、argument、nowUser这些变量都赋值为-1，以标志为没有有效值。



### 4.2创建文件系统

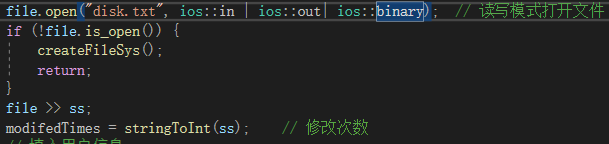
该函数用于创建文件系统，并将初始化的文件系统信息写入磁盘文件disk.txt 中。

**加锁文件系统：**

* 使用 unique\_lock 和 fileMutex 互斥锁确保文件系统的线程安全。

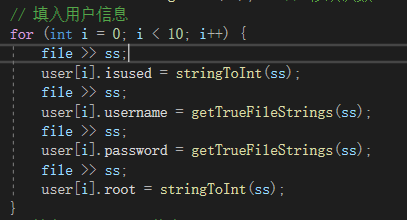
**尝试打开文件系统存储文件：**

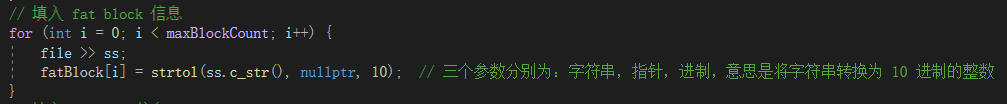
* 尝试以读写模式打开 disk.txt 文件。
* 如果文件不存在，调用 createFileSys 函数创建新的文件系统并返回。

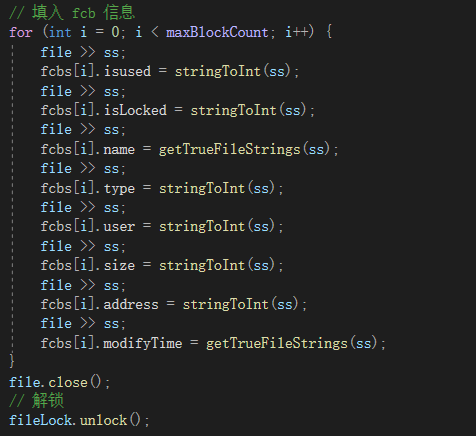


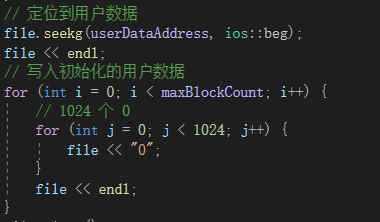
**读取文件内容并初始化文件系统：**

* 读取修改次数 modifedTimes。
* 读取用户信息并填充用户数组 user。
* 读取 FAT block 信息并填充 fatBlock 数组。
* 读取 bit map 信息并填充 bitMap 数组。
* 读取 FCB 信息并填充 fcbs 数组。



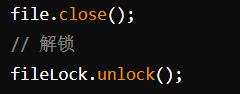




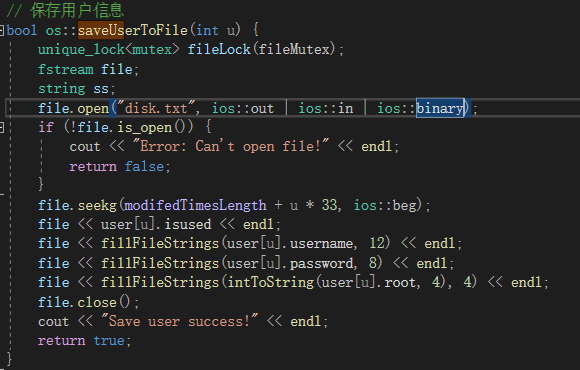


**关闭文件并解锁：**

* 关闭文件并释放互斥锁。



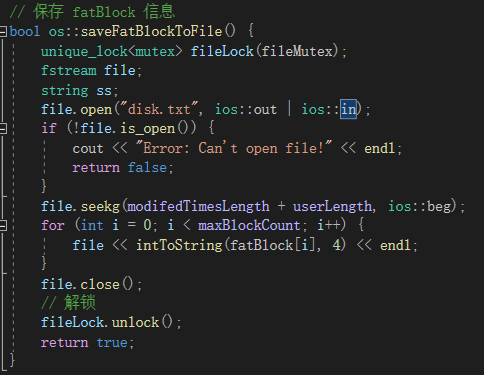
### 4.3 保存用户数据

* 该函数的作用是将指定用户的信息保存到磁盘文件中。
* 创建文件流对象并打开名为 disk.txt 的文件。
* 将文件指针定位到用户信息的位置，该位置可以通过计算得到。
* 将指定用户的信息依次写入文件，包括 isused（是否被使用）、username（用户名）、password（密码）和 root（根目录索引）。

### 4.4 保存文件分配表数据

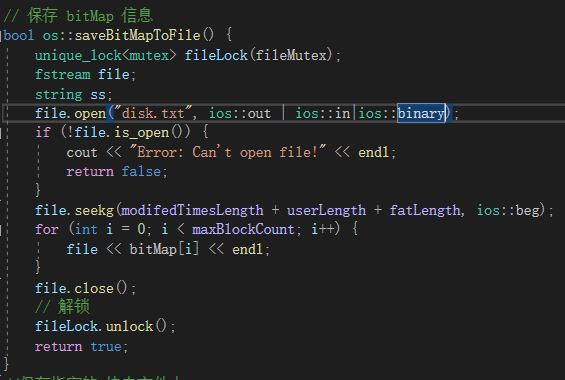
该函数的作用是将文件分配表（fatBlock）信息保存到磁盘文件中。

将文件指针定位到文件分配表信息的位置，该位置可以通过计算得到。循环遍历文件分配表数组，并将每个块的文件分配表信息依次写入文件。



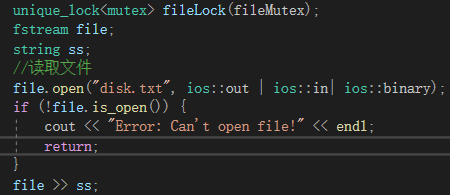
### 4.5 保存位示图

* 该函数的作用是将文件分配表（fatBlock）信息保存到磁盘文件中。
* 将文件指针定位到文件分配表信息的位置，该位置可以通过计算得到。
* 循环遍历文件分配表数组，并将每个块的文件分配表信息依次写入文件。



### 4.6 更新数据(update函数)

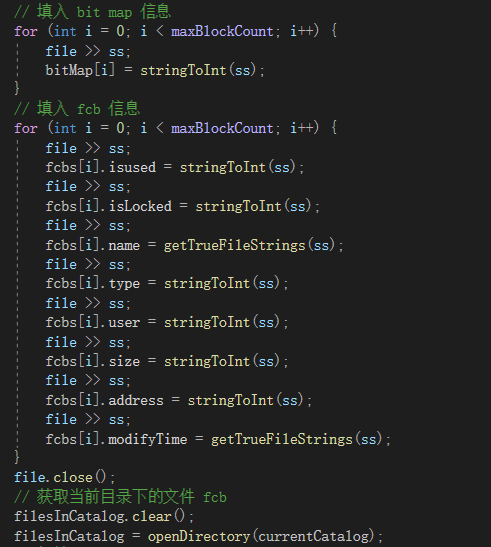
* 创建文件流对象并打开名为disk.txt的文件，以读写模式打开。



* 从文件中读取修改次数并将其转换为整数，赋值给变量modifiedTimes，表示文件系统的修改次数。
* 循环10次，读取文件中的用户信息，并填充到相应的用户数据结构中。依次读取isused（是否被使用）、username（用户名）、password（密码）和root（根目录索引）。



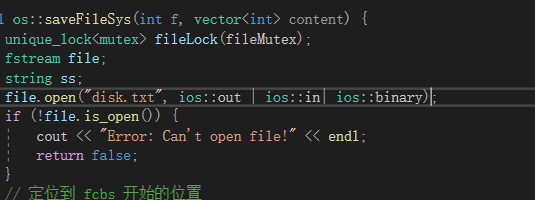
* 循环`maxBlockCount` 次，读取文件中的文件分配表信息，并将其转换为整数类型，存储到相应的`fatBlock` 数组中。
* 循环`maxBlockCount` 次，读取文件中的位图信息，并将其转换为整数类型，存储到相应的`bitMap` 数组中。
* 循环`maxBlockCount` 次，读取文件中的文件控制块（FCB）信息，并填充到相应的FCB 数据结构中。



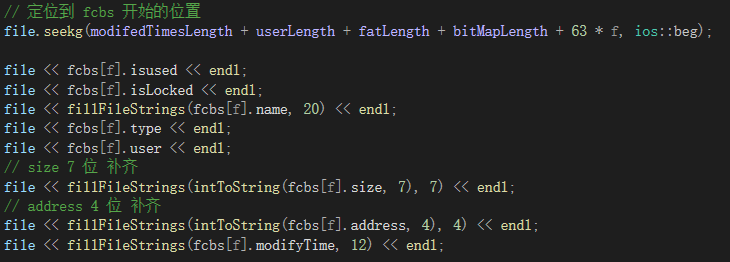
### 4.7 保存文件系统

创建saveFileSys函数接受一个文件描述符和一个整数向量作为输入，将整数向量中的内容保存到文件系统中对应的文件中。具体步骤如下：

1. 首先获取文件锁。
2. 打开文件系统文件，如果打开失败则返回false。



1. 将文件指针定位到文件控制块（FCB）所在的位置，并将FCB的各个属性写入文件。

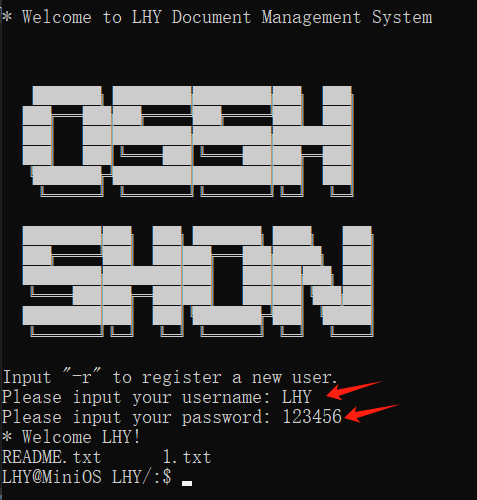


1. 将文件指针定位到用户数据块的开始位置。
2. 将整数向量中的内容写入文件。
3. 如果需要，创建新的数据块并将其链接到文件的末尾。
4. 释放文件锁。
5. 将文件分配表和位图保存到文件系统文件中。
6. 如果整个过程中没有出现错误，则返回true。



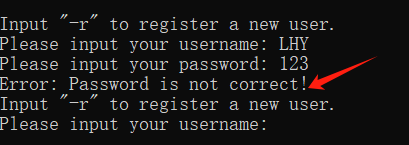
# 实验结果截图：

## 登录界面：

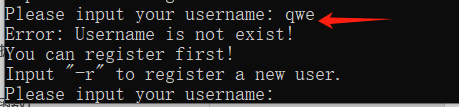


成功登录。

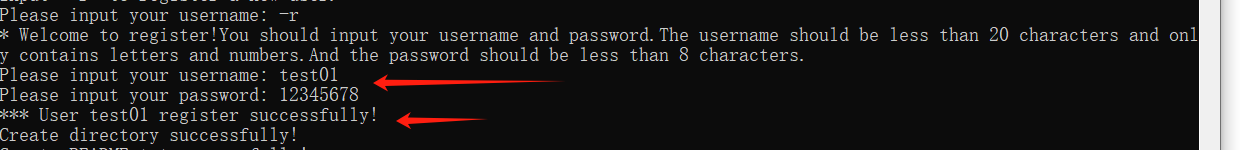
如果密码输入错误，会要求用户重新输入用户和密码



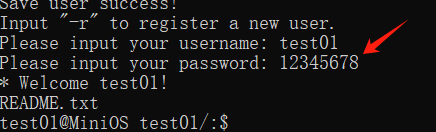
如果用户输入的用户名错误，则会提示用户名不存在！



## 注册逻辑：

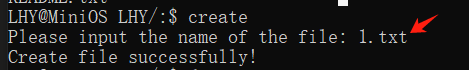


如果新注册一个用户名，显示如上，会要求输入新的用户名和密码，注册成功后会显示注册成功。



成功登录。

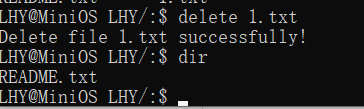
## Create指令：



首先输入create字符串，然后输入创建文件的名称：



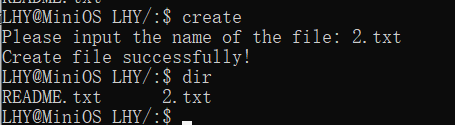
## Delete命令：



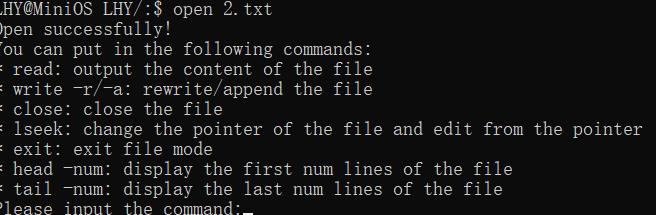
删除1.txt之后，dir中就只有一开始创建的文件了。

## Open命令：

我们再次新创建一个文件2.txt。

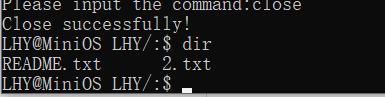


打开它，然后我们可以进行各种各样的操作了。



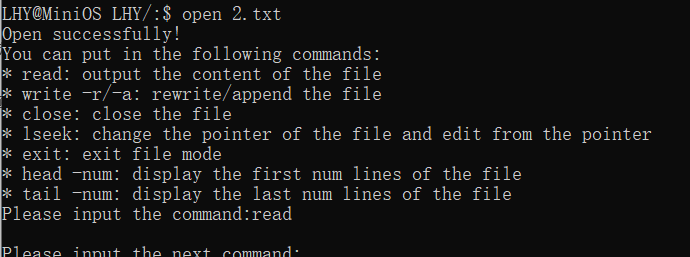
## Close命令：

我可以输入close，然后这个系统就退回到文件目录的界面中：



## Read命令：

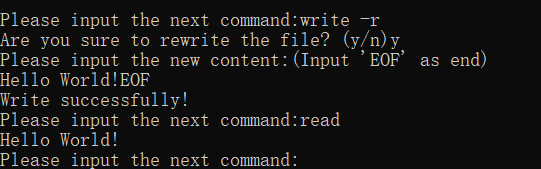
再接着打开文件，执行read命令。



因为新创建的事空的文件，所以read之后没有任何显示

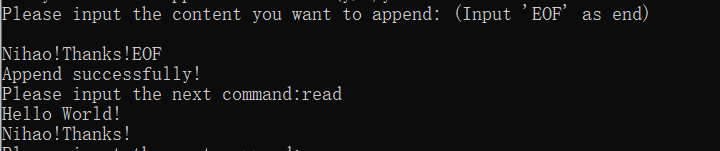
## Write命令（覆盖写）：

执行覆盖写命令。首先询问用户是否真的要覆盖写，输入的字符串以EOF结尾！



成功写入。

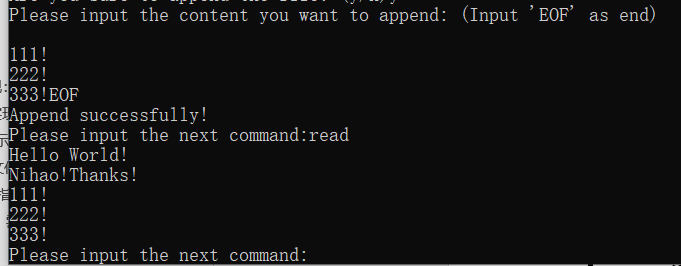
## Write命令（追加写）：



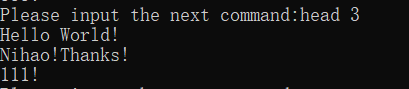
先输入空格之后，再进行写入相关的字符串，字符串会显示在第二行中。

## Head-num读取文件前num行

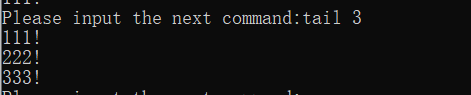
我们又追加写了一些字符串



我们可以输入head 3，显示前3行的内容：



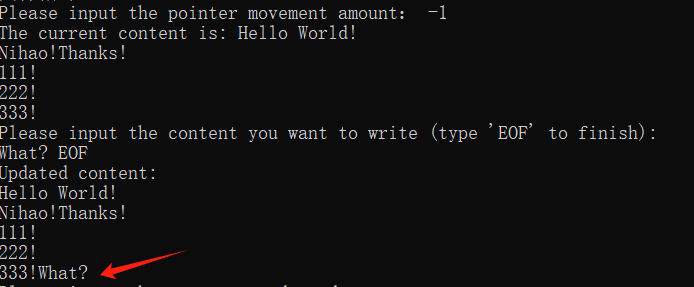
## Tail-num，读取文件后num行

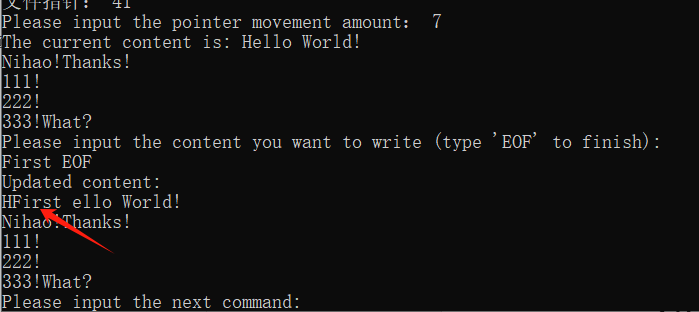


显示的是文件的后3行内容。

## Lseek读写指针：

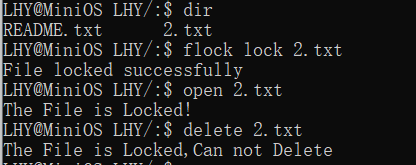
首先输入lseek指令，然后再输入-1，让指针指向文件的末尾，然后插入字符串：



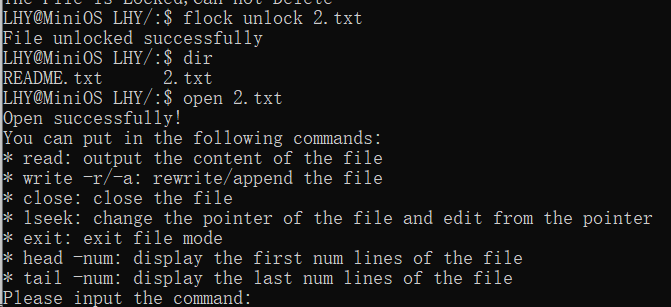


移动指针，在文件开头之后的一个字符中插入输入的字符。

## Flock，实现对文件的锁定



当我锁定文件的时候，无法打开文件和删除文件，然后再解锁：

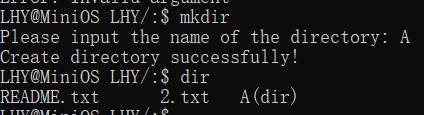


Delete也同样可以删除。

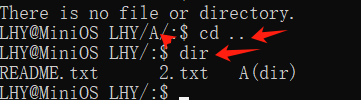
## Dir显示当前目录：



## mkdir创建目录：

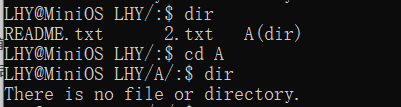


创建目录成功



我们在A中再创建一个菜单B。

## Cd进入目录：



我们进入新创建的A目录，里面什么都没有

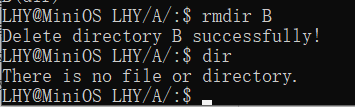
我们返回上一层菜单。

从b中返回根目录：

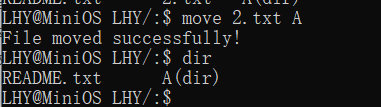


## rmdir删除目录：

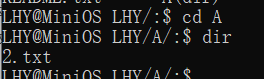
我们删除A下的B目录：



## Move 移动文件：

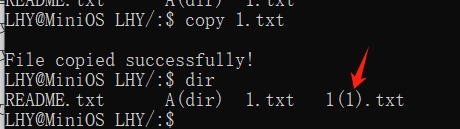


将2.txt移动到A目录下。



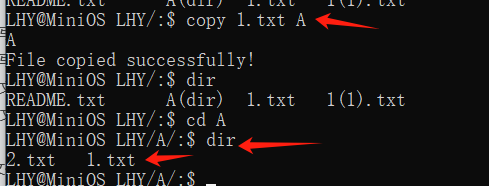
确实是在A目录下！

## Copy 复制文件：

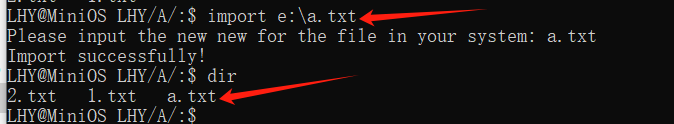


如果不输入路径，则拷贝到这个目录路径下，会修改这个文件的名称。

如果移动到其他路径下：



## import导入文件:

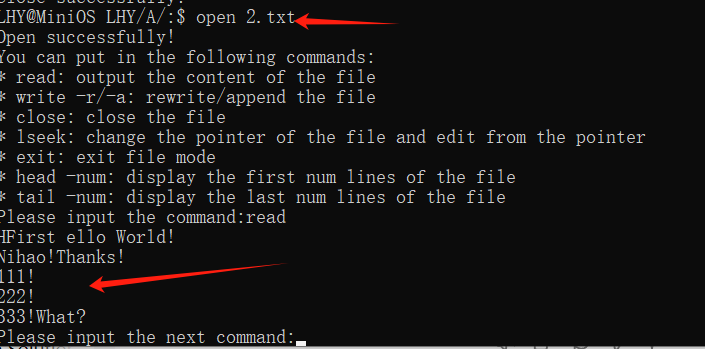


从e盘中导入文件。

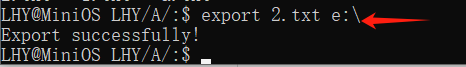


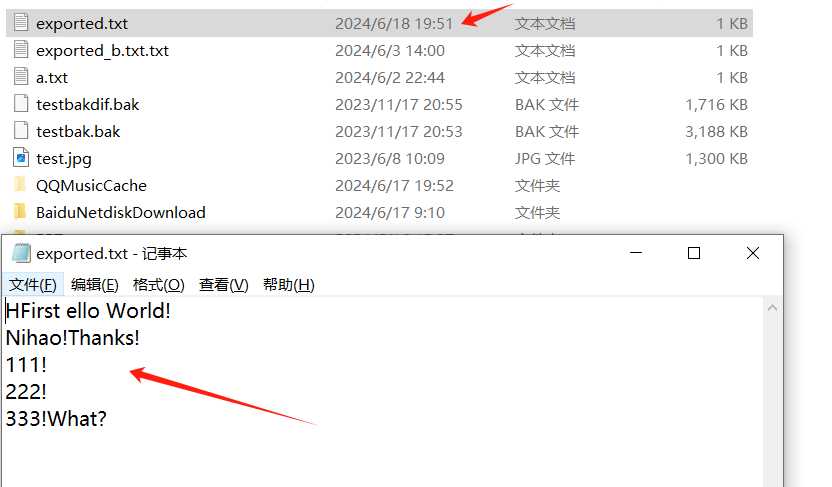
## export导出文件：

将刚刚创建的2.txt文件导出：



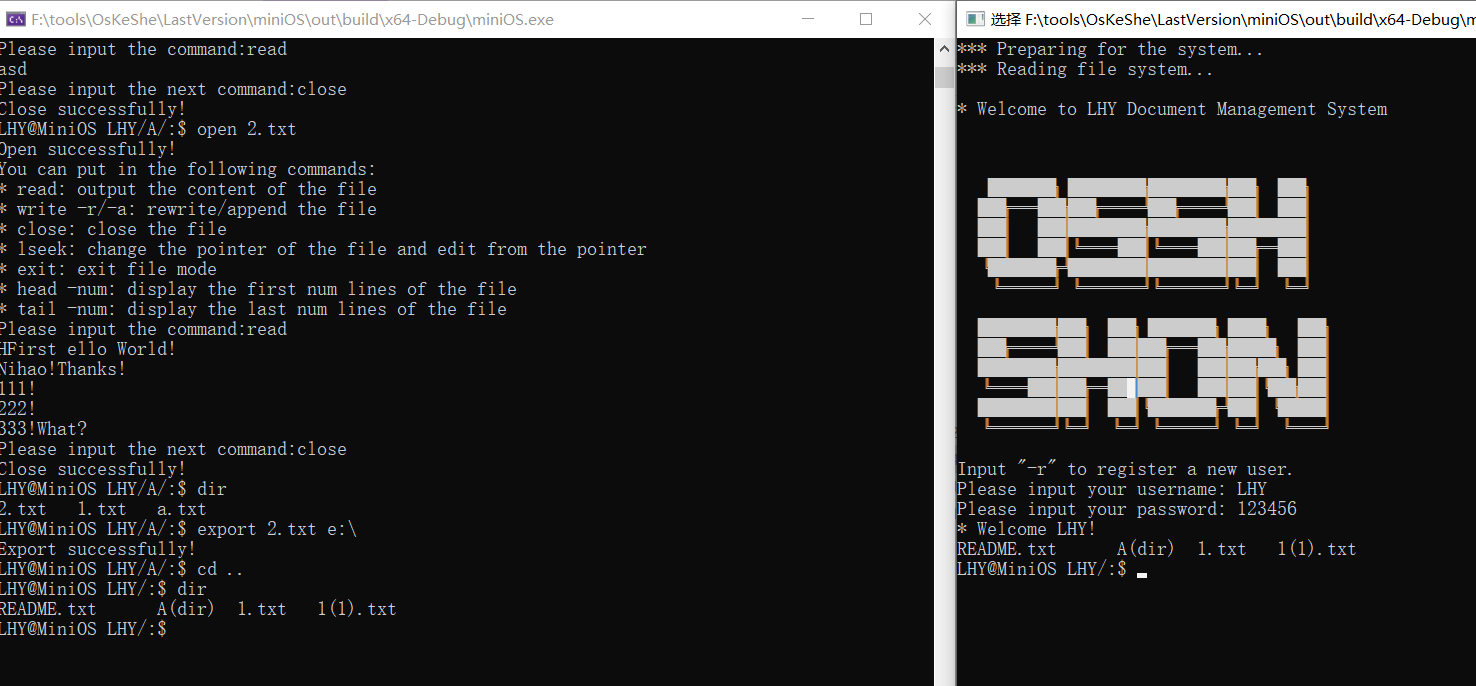
导出到e盘



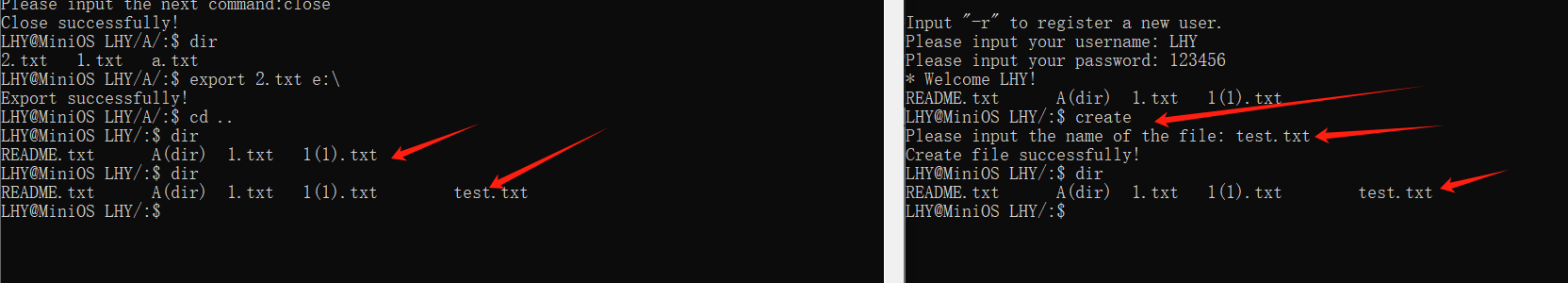


## 多线程执行：

打开两个窗口：



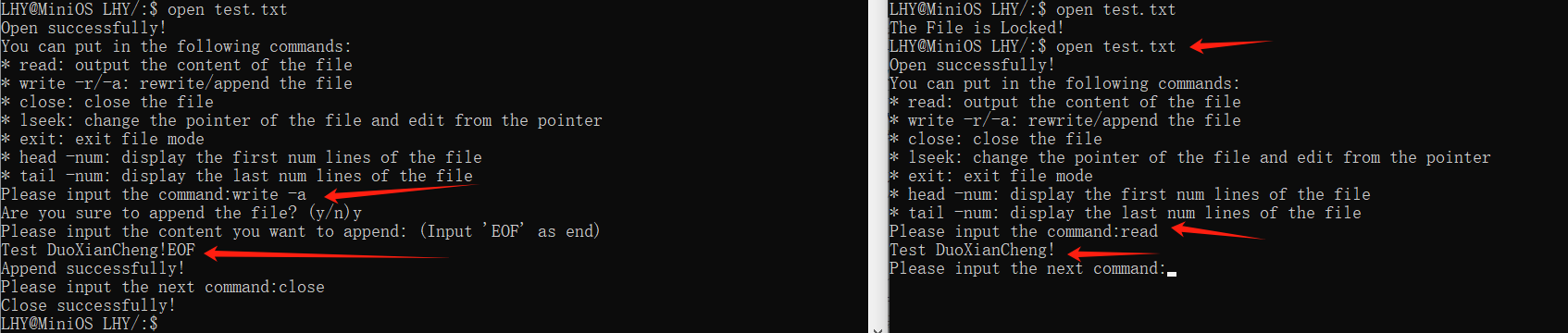
比如说在一个里面创建一个新的文件，另外一个会自动更新：



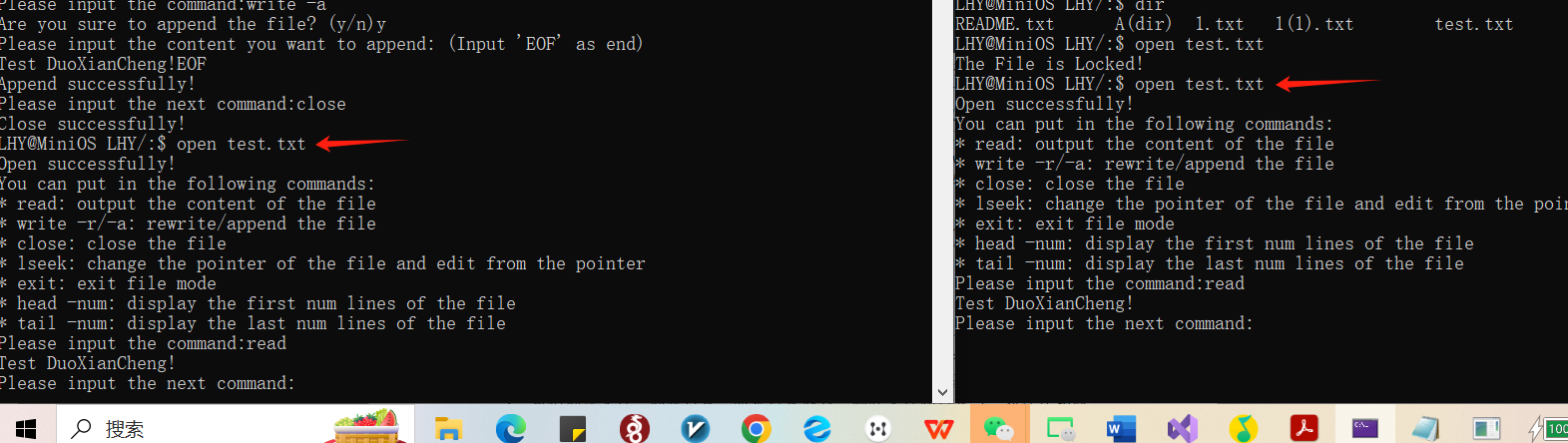
比如说在一个程序中将其锁定之后，另外一个程序中无法打开：



在一个程序中对一个文件写入东西之后，另外一个文件中可以正确的读出：



他们也可以同时打开文件，进行操作：



证明这是一个多线程的程序。

# 实验分析及感受

在完成这次课程设计的过程中，我对文件系统的设计和基本命令的实现有了更加深刻的认识。通过模拟文件系统的操作，我深入理解了文件控制块（FCB）的重要性以及它所保存的关键文件信息。此外，我还掌握了创建文件、目录，以及删除文件和目录等基本文件系统操作的设计与实现。这不仅加深了我对操作系统结构的理解，也增强了我在多线程编程方面的能力。

在实践测试中，我利用多线程技术，成功实现了系统的并发执行和良好的用户交互体验。通过精心设计线程的管理，我提高了系统的响应速度并支持了多用户的同时访问。此外，我还学习了如何运用互斥锁来确保线程间对文件操作的同步和互斥，有效避免了多线程访问冲突和数据不一致的问题。

在撰写设计报告的过程中，我也意识到了条理清晰的结构和细致的说明对于表达设计思路和实现方法的重要性。这有助于使读者更加容易理解和使用该系统。

总之，通过这次课程设计，我不仅加深了对文件系统和多线程编程的理解，还提高了自己的实践操作能力。我认识到，设计和实现一个高效可靠的文件系统是一项充满挑战的工作，它需要对多种因素和可能性进行综合考虑。我相信，通过不断的学习和实践，我将能在这一领域取得更进一步的进展。