

**操作系统课程设计报告**

**设计题目**  兼容Unix/Linux命令接口1

**学生姓名**  侯腾跃

**学 号**  2022217477

**专业班级**  计算机科学与技术22-4班

**指导教师**  田卫东

**完成日期**  2024/12/29

**合肥工业大学 计算机与信息学院**

目录

[操作系统课程设计报告 1](#_Toc11511)

[1. 课程设计任务、要求、目的 2](#_Toc30009)

[1.1课程设计任务 2](#_Toc22403)

[1.2课程设计目的和要求 3](#_Toc14244)

[2.开发环境.........................................................................................................................................3](#_Toc6580)

[3.相关原理及算法.............................................................................................................................3](#_Toc26075)

[3.1 说明文档： 3](#_Toc32079)

[3.2 全局变量： 4](#_Toc2668)

[3.3 主要函数算法声明： 4](#_Toc21276)

[4. 系统结构和主要的算法设计思路 5](#_Toc10660)

[4.1 程序流程图 5](#_Toc21046)

[4.2 算法设计 5](#_Toc23085)

[5. 程序实现——主要数据结构 12](#_Toc27315)

[5.1 全局缓冲区和路径变量 12](#_Toc30611)

[5.2 命令参数数组 12](#_Toc11917)

[5.3 输入缓冲区 13](#_Toc26110)

[5.4 索引变量 13](#_Toc2818)

[5.5 字符变量 13](#_Toc22686)

[5.6 数据结构的作用和交互 13](#_Toc7618)

[6. 程序实现——程序实现细节描述 13](#_Toc22687)

[6.1在主函数引入头文件SO.h 13](#_Toc3226)

[6.2全局变量声明 14](#_Toc30883)

[6.3 命令函数定义 14](#_Toc15240)

[6.4 主函数实现 14](#_Toc16652)

[6.5 主函数逻辑实现 15](#_Toc23639)

[7. 程序运行的主要界面和实验结果截图 16](#_Toc929)

[8. 总结和感想体会 19](#_Toc27700)

[附录1：程序清单(部分) 21](#_Toc31058)

## 1. 课程设计任务、要求、目的

### 1.1课程设计任务

操作系统接口：兼容Unix/Linux命令接口1。（1人，难度：4）

1.为Windows操作系统建立一个兼容Unix/Linux命令的命令接口；

2.实现关于文件系统的操作命令，包括ls,cat,cp,rm,mkdir,rmdir,cd,sort 等，命

令的内容与详细格式请查阅Unix/Linux命令手册；

3.可以字符形式接收命令，执行命令，然后显示命令执行结果；

### 1.2课程设计目的和要求

操作系统课程设计涉及大量的内核代码编写，原则上只允许使用系统级程序设计语言C语言或者汇编语言。主要的操作系统代码部分都使用C语言（不建议使用C++语言），部分设计关键性能和直接与硬件打交道的部分，建议使用汇编语言实现。 每个完成的课程设计包含两个代码组成部分：(1) 操作系统代码；(2) 演示代码；

**操作系统代码**

操作系统代码部分，完成课程设计的主体设计内容，需要根据操作系统内核原理来进行

数据结构的设计和算法的设计与实现。

这一部分设计内容以一系列函数的形式表现出来，设计的最终结果，就是一系列的.C

文件，其中定义和实现所有完成设计题目功能的代码。一般将这部分的设计组织成一个

Project，Project的目标不是生成应用程序，而是生成一个.lib。 这部分设计强调按照操作系统原理课程的要求，设计和实现相关的数据结构。数据结构需要精心设计，要求紧凑、节省空间。算法的设计也以操作系统原理课程讲授的标准算法为基准。

**演示代码**

演示代码是展示和验证上述“1.1.1 操作系统代码”中所设计的代码的有效性而额外编

写的演示程序。需要通过将上述函数有机地按一定顺序调用，以展示操作系统较大模块的功

能和运行过程。 演示代码通常带有UI，可以用标准的Windows/Linux窗口作为UI，也可以使用字符命令作为展示窗口，要求功能展示完整，操作简便。 这部分代码，通常也是以一系列的.C/.CPP文件的形式出现，并组织成一个Project这个Project的链接结果，是一个应用程序，装入模块。例如Windows环境下的.exe。

## 2. 开发环境

开发工具：CLion 2024.3.1.1

编译器：mingw64

编程语言：C语言

## 3. 相关原理及算法

### 3.1 说明文档：

#define PRINTLSHELP "用法：ls [选项]\n\

列出文件信息(默认为当前目录)\n\

ls无参 显示一个目录中的文件和子目录\n\

ls -l 每行仅显示一个目录或名称\n\

ls --help 显示此帮助信息并离开"

#define PRINTMKDIRHELP "用法：mkdir 目录\n\

若目录不存在则创建目录\n\

mkdir --help 显示此帮助信息并离开"

#define PRINTRMDIRHELP "用法：rmdir 目录\n\

若目录没有数据则删除目录\n\

rmdir --help 显示此帮助信息并离开"

#define PRINTCPHELP "用法：cp 来源文件 目的文件\n\

将来源文件复制到目的文件\n\

cp --help 显示此帮助信息并离开"

#define PRINTCATHELP "用法：cat 文件\n\

或：cat 文件1 >> 文件2\n\

显示文件内容或连接两个文件，并将结果输出到显示屏上\n\

cat 文件 将文件的内容输出到显示屏上\n\

cat 文件1 >> 文件2 将文件1连接到文件2的结尾，并输出到显示屏上\n\

cat --help 显示此帮助信息并离开"

#define PRINTSORTHELP "用法：sort 文件\n\

将文本文件内容加以排序\n\

sort --help 显示此帮助信息并离开"

### 3.2 全局变量：

buf用于缓存数据，3个path用于保存路径

char buf[100], path[100], path1[100], path2[100];

### 3.3 主要函数算法声明：

void cd\_cmd(char\* route);改变当前工作目录。

void ls\_cmd(char\* array);列出当前目录下的文件和子目录。

void mkdir\_cmd(char\* array);创建新目录。

void rmdir\_cmd(char\* array);删除空目录。

void cp\_cmd(char\* array1, char\* array2);复制文件。

void cat\_cmd1(char\* array);显示文件内容或将一个文件的内容追加到另一个文件。

void cat\_cmd2(char \*array1, char \*array2);

void cmd\_sort(char \*array);对文件中的行进行排序。

## 系统结构和主要的算法设计思路

### 4.1 程序流程图

命令是否正确确

是否继续

文件内容排序

创建目录命令

删除文件命令

改变工作目录

显示文本命令

复制文件令命令

列出目录下文件

执行相关操作

程序结束

输入操作命令

开始程序

否

否

是

是

### 4.2 算法设计

**（1）void cd\_cmd(char\* route)**

尝试将当前工作目录更改为传入的路径 route。如果更改失败（路径不存在），打印错误信息并显示当前路径提示符。

if (!SetCurrentDirectory(route)) {

printf("路径不存在!\n");

printf("%s>", path);

}

如果更改成功，更新当前路径并显示新的路径提示符。

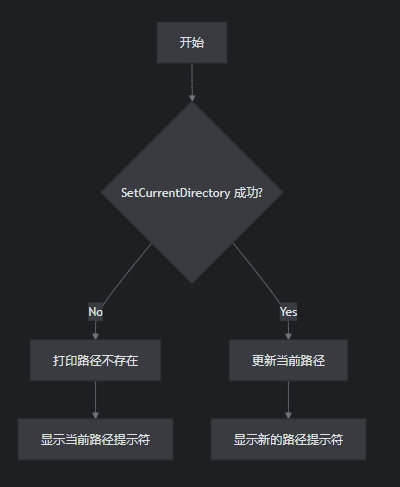
else {

SetCurrentDirectory(route);

GetCurrentDirectory(100, path);

printf("%s>", path);

}



**（2）void ls\_cmd(char\* array)**

strcat(path1, "\\\*.\*"); //获取当前工作目录并添加通配符 \*.\*。

分配内存给 WIN32\_FIND\_DATA 结构体指针 path2WIN32\_FIND\_DATA\* path2 = (WIN32\_FIND\_DATA\*)malloc(sizeof(WIN32\_FIND\_DATA));

然后使用FindFirstFile(path1, path2))和 FindNextFile(handle, path2) 遍历目录中的文件。

对每个文件获取其最后修改时间，并根据传入的命令参数决定输出格式：

HANDLE hDir = CreateFile(path2->cFileName, GENERIC\_READ, FILE\_SHARE\_READ | FILE\_SHARE\_DELETE,NULL, OPEN\_EXISTING, FILE\_FLAG\_BACKUP\_SEMANTICS, NULL);

if (GetFileTime(hDir, &lpCreationTime, &lpLastAccessTime, &lpLastWriteTime)) {

FILETIME ftime;

FileTimeToLocalFileTime(&lpLastWriteTime, &ftime); // 转换成本地时间

FileTimeToSystemTime(&ftime, &stime); // 转换成系统时间格式

}

如果传入参数为 ls，则以多列形式输出文件名。

if (strcmp(array, "ls") == 0) {

printf("%-20s", path2->cFileName);

if (filenum % 3 == 2) {

printf("\n");

}

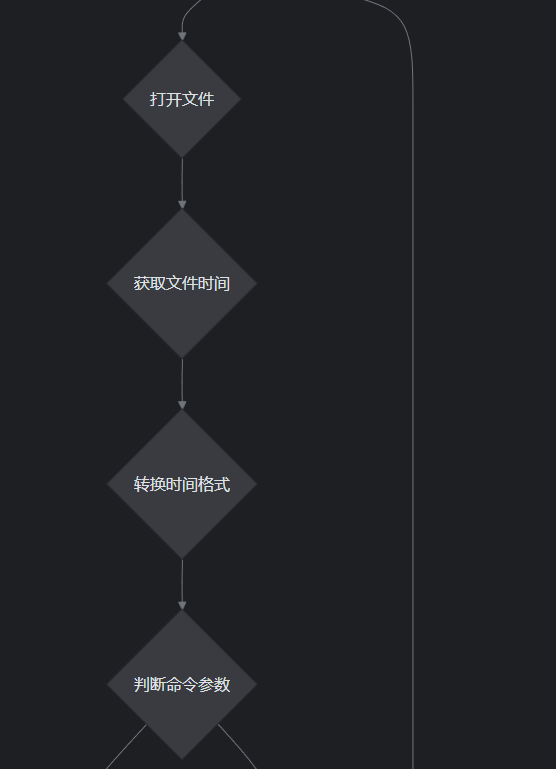
filenum++;

}

否则，输出文件的最后修改时间和文件名。

else {

printf("%04d-%02d-%02d %02d:%02d %s\n", stime.wYear, stime.wMonth, stime.wDay, stime.wHour, stime.wMinute, path2->cFileName);}



**（3）void mkdir\_cmd(char\* array)**

获取当前工作目录并拼接传入的目录名。

检查该目录是否已存在：

如果存在，输出提示信息并返回当前目录路径。

if (hFind != INVALID\_HANDLE\_VALUE) {

printf("该目录已存在，创建失败!\n");

GetCurrentDirectory(100, path);

printf("%s>", path);

}

如果不存在，创建新目录并返回当前目录路径。

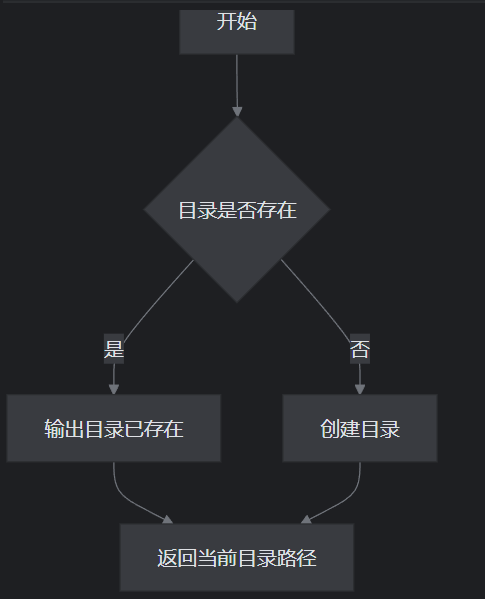
else {

CreateDirectory(array, NULL);

GetCurrentDirectory(100, path);

printf("%s>", path);

}



**（4）void rmdir\_cmd(char\* array)**

获取当前工作目录，并拼接传入的目录名。

使用 FindFirstFile 检查该目录是否存在。

如果目录不存在，输出错误信息并返回当前工作目录。

if (hFind == INVALID\_HANDLE\_VALUE) {

printf("该目录不存在，删除失败!\n");

GetCurrentDirectory(100, path);

printf("%s>", path);}

如果目录存在，调用 RemoveDirectory 删除该目录，并返回当前工作目录。

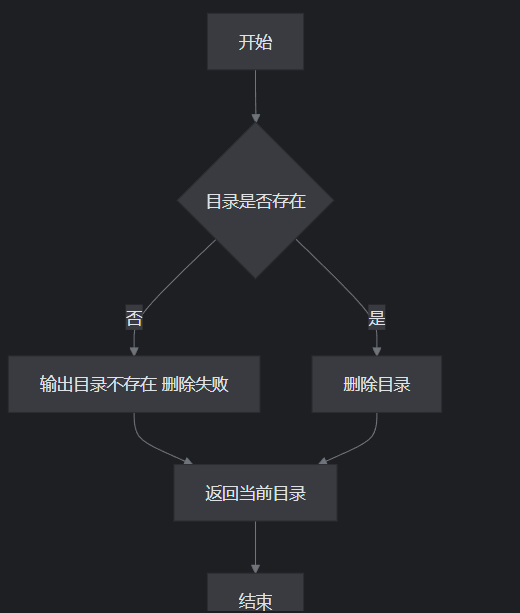
else {

RemoveDirectory(array);

GetCurrentDirectory(100, path);

printf("%s>", path);

}



**（5）void cp\_cmd(char\* array1, char\* array2)**

获取当前工作目录并构建源文件和目标文件的完整路径。

使用 CopyFile 函数进行文件复制。

if (!CopyFile(fullSourcePath, fullDestPath, FALSE)) {

如果复制失败，输出错误信息；

printf("复制失败，请检查路径和文件是否存在！错误代码: %lu\n", GetLastError());

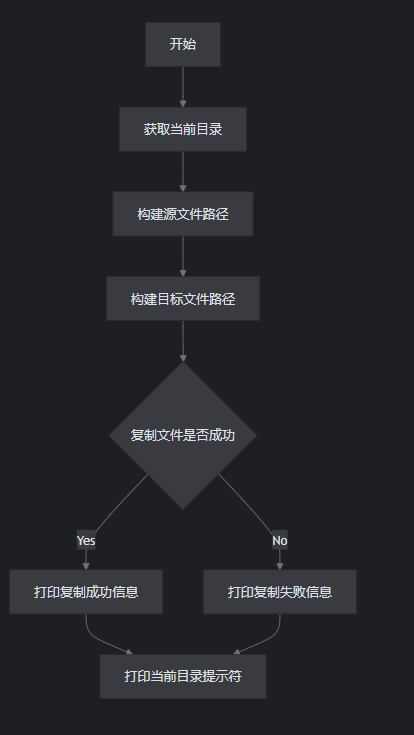
}

如果成功，输出成功信息。

else {

printf("文件复制成功: %s -> %s\n", fullSourcePath, fullDestPath);

}



1. **void cat\_cmd1(char\* array)**

该函数用于显示文件内容

获取当前目录路径并构造完整文件路径。

打开文件，若失败则提示错误信息。

if (!infile) {

printf("无法打开文件%s\n", path);

GetCurrentDirectory(100, path);

printf("%s>", path);

}

若成功打开文件，则逐行读取并打印文件内容，最后关闭文件。

else {

while (fgets(ch, 1000, infile)) {

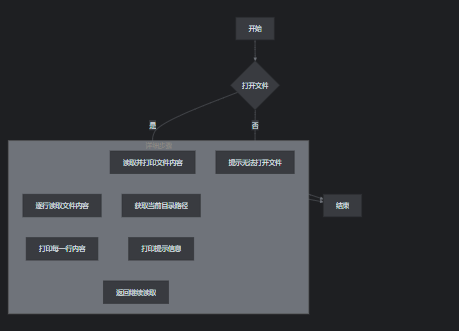
printf("%s", ch);

}

GetCurrentDirectory(100, path);

printf("%s>", path);

}



**void cat\_cmd2(char \*array1, char \*array2)**

该函数用于将一个文件的内容追加到另一个文件中，并显示追加后的文件内容：

获取当前目录路径并分别构造输入和输出文件的完整路径。

FILE \*infile1 = fopen(path1, "r");

if (!infile1) {

//打开输入文件，若失败则提示错误信息。

printf("无法打开文件%s\n", path1);

GetCurrentDirectory(100, path1);

printf("%s>", path1);

}

若成功打开输入文件，则以追加模式打开输出文件，若失败则提示错误信息。

else {

FILE \*outfile2 = fopen(path2, "a"); // 以追加模式打开输出文件

if (!outfile2) {

printf("无法打开文件%s\n", path2);

GetCurrentDirectory(100, path2);

printf("%s>", path2);

}

else {

while (fgets(ch, 1000, infile1) != NULL) { // 读取输入文件并写入输出文件

fputs(ch, outfile2);

}

最后关闭所有文件。

**（7）void cmd\_sort(char \*array)**

获取文件路径：通过当前目录和传入的文件名拼接成完整路径。

打开文件并读取内容：如果文件存在，则逐行读取并记录行数。

冒泡排序：对读取的行进行字典顺序排序。

for (int x = 0; x < lnum - 1; x++) {

min = x;

for (int y = x + 1; y < lnum; y++) {

if (strcmp(line[y], line[min]) < 0) {

min = y;

}

}

if (min != x) {

strcpy(tmp, line[x]);

strcpy(line[x], line[min]);

strcpy(line[min], tmp);

}

}

写回文件：将排序后的行重新写入文件。

FILE \*outfile = fopen(path, "w");

if (!outfile) {

printf("文件打开失败\n");

}

else {

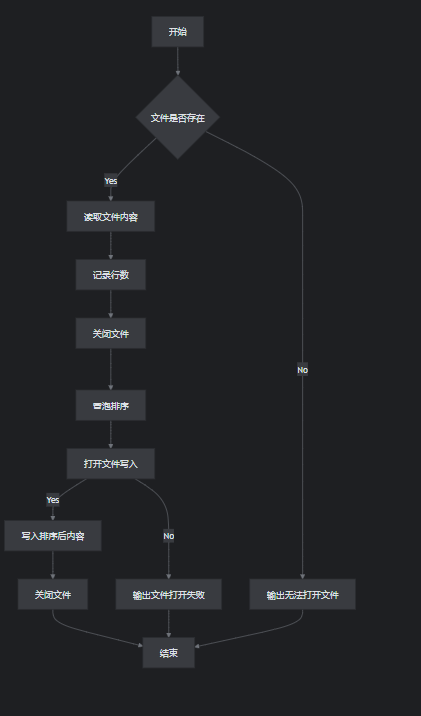
for (int c = 0; c < lnum; c++) {

fprintf(outfile, "%s\n", line[c]);

}

fclose(outfile);

}



## 5. 程序实现——主要数据结构

### 5.1 全局缓冲区和路径变量

char buf[100], path[100], path1[100], path2[100];

buf：用于存储用户输入的命令行内容。它是一个固定大小的字符数组，最大长度为100个字符。

path、path1、path2：用于存储当前工作目录或文件路径。这些变量在不同函数中被用来构建完整的文件路径。

### 5.2 命令参数数组

char\* arg[4];

arg：这是一个指针数组，用于存储解析后的命令及其参数。每个元素指向一个字符串（命令或参数）。`arg[0]`通常是命令名，后续元素是命令的参数。

### 5.3 输入缓冲区

char\* input = (char\*)malloc(input\_len + 1);

input：这是一个动态分配的字符数组，用于存储从`buf`复制过来的命令行内容。它的大小根据实际输入长度动态调整，确保能够容纳用户输入的所有字符。

### 5.4 索引变量

int i, j, k, input\_len;

i：用于遍历用户输入的命令字符串。

j：用于记录当前解析到的命令或参数的长度。

k：用于记录已经解析出的命令或参数的数量。

input\_len：记录用户输入命令的实际长度。

### 5.5 字符变量

char c;

c：用于逐个读取用户输入的字符，并进行初步处理（如去除空格和制表符）。

### 5.6 数据结构的作用和交互

命令解析流程：用户输入命令后，程序通过`getchar()`逐个读取字符，直到遇到换行符`\n`为止。这些字符被存储在`buf`中，然后复制到动态分配的`input`中。接下来，程序遍历`input`，将命令和参数分离出来，并存储到`arg`数组中。每个参数都被分配独立的内存空间，并通过`strcpy`复制内容。

命令执行流程：解析完成后，程序根据`arg[0]`中的命令名调用相应的命令处理函数（如`cd\_cmd`、`ls\_cmd`等）。每个命令处理函数会根据需要使用`arg`数组中的参数来执行具体的操作。

内存管理：每次解析完命令后，程序会释放动态分配的内存（如`input`和`arg`中的各个参数），以避免内存泄漏。

## 程序实现——程序实现细节描述

### 6.1在主函数引入头文件SO.h

#include "OS.h"

### **6.2**全局变量声明

char buf[100], path[100], path1[100], path2[100];

### 6.3 命令函数定义

cd\_cmd：改变当前工作目录。

ls\_cmd：列出当前目录下的文件和子目录。

mkdir\_cmd：创建新目录。

rmdir\_cmd：删除空目录。

cp\_cmd：复制文件。

cat\_cmd1 和 cat\_cmd2：显示文件内容或将一个文件的内容追加到另一个文件。

cmd\_sort：对文件中的行进行排序。

### 6.4 主函数实现

初始化：

SetCurrentDirectoryA("C:\\"); //设置默认工作目录为C:\。

char path[100];

GetCurrentDirectoryA(100, path);

printf("%s>", path);// 获取并打印当前工作目录

命令循环：

使用while (1)进入无限循环，等待用户输入命令。

用户输入命令后，程序会解析命令行参数，并根据第一个参数调用相应的命令处理函数。

命令解析：

将用户输入的命令分割成多个参数（如命令名、文件名等）。

char\* input = (char\*)malloc(input\_len + 1);

strcpy(input, buf); // 将命令从缓存复制到input中

根据命令名调用对应的处理函数。

其中命令解析具体描述：

主要函数实现后，为了能让Windows系统识别linux的命令，必须把把命令解析成命令+[参数]的形式。过程如下

初始化：将 arg 数组的两个元素置为 NULL。

读取命令：跳过空白字符和制表符，直到遇到非空字符或换行符。

处理输入：读取用户输入并存储在 buf 中，直到遇到换行符。

分配内存：为输入命令分配内存，并将其复制到 input 中。

解析命令：将命令解析为命令和参数的形式，去除多余的空格，并将每个部分存储在 arg 数组中。

### 6.5 主函数逻辑实现

在此不赘述代码实现，仅说明具体逻辑

**ls命令**

没有参数 -> 调用ls\_cmd函数列出当前目录内容。

有参数 -> 如果参数是--help，打印帮助信息。

如果参数是-l，调用ls\_cmd函数列出详细信息。

否则，打印无效命令提示。

最后释放分配的内存并继续执行。

**cd命令**

检查输入命令是否为 cd。如果是 cd 命令则跳过命令前的空格。

没有参数 -> 将当前目录设置为 C:\ 并打印提示符。

有参数 -> 提取参数并调用 cd\_cmd 函数更改目录。

最后释放分配的内存并重置相关变量。

**cat命令**

检查是否有参数：如果cat命令没有参数，提示用户缺少文件参数，并显示当前目录。

单个参数处理：

如果参数是--help，打印帮助信息。

否则调用cat\_cmd1函数处理单个文件参数。

多个参数处理：

如果有三个参数且第二个参数为>>，调用cat\_cmd2函数处理追加文件内容。

其他情况提示无效命令。

内存管理：使用malloc分配内存并及时释放。

**cp命令**

检查输入参数：判断用户输入的命令是否为cp，并处理不同情况。

无参数处理：如果cp命令后没有参数，提示用户缺少文件参数，并显示当前目录路径。

单参数处理：如果只有一个参数，检查是否为--help，否则提示缺少目的文件。

双参数处理：如果有两个参数，分配内存并复制文件名，调用cp\_cmd函数执行复制操作，最后释放分配的内存。

**mkdir命令**

检查命令是否为 mkdir：通过 strcmp 检查输入命令是否为 mkdir。

跳过空格：跳过命令后的空格字符。

无参数处理：如果 mkdir 后没有参数，提示用户参数太少，并显示当前目录提示符。

获取参数：如果有参数，提取并存储参数到 arg[1] 中。

帮助信息：如果参数是 --help，打印帮助信息并显示当前目录提示符。

执行 mkdir：如果不是 --help，调用 mkdir\_cmd 函数创建目录。

内存释放：释放动态分配的内存。

**rmdir命令**

判断命令是否为rmdir：通过strcmp函数检查输入命令是否为rmdir。

跳过空格：从输入字符串中跳过所有空格字符，直到找到实际参数。

检查是否有参数：

没有参数 -> 提示用户参数太少，并打印帮助信息。

有参数 -> 进一步处理：

参数是--help -> 打印帮助信息。

否则 -> 调用rmdir\_cmd函数删除指定目录。

内存管理：释放动态分配的内存。

**sort命令**

判断是否为sort命令：通过strcmp检查输入的第一个参数是否为sort。

跳过空格：跳过命令后的空格，直到遇到非空格字符。

检查是否有参数：

如果没有参数，提示无效命令并打印帮助信息。

如果有参数，提取参数并进一步处理。

处理参数：

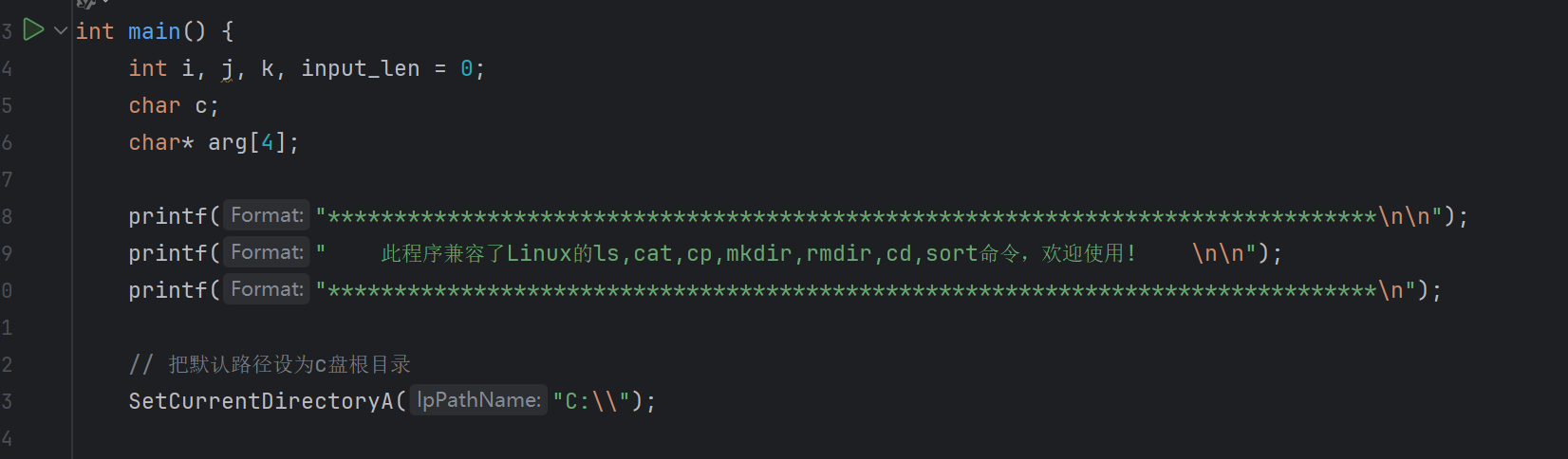
如果参数是--help，打印帮助信息。

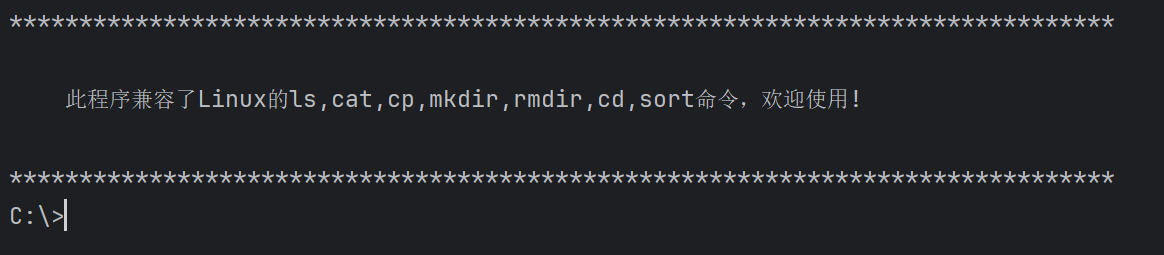
否则调用cmd\_sort函数处理排序逻辑。

释放内存：释放分配的内存并继续循环。

## 程序运行的主要界面和实验结果截图

**主函数输出部分**

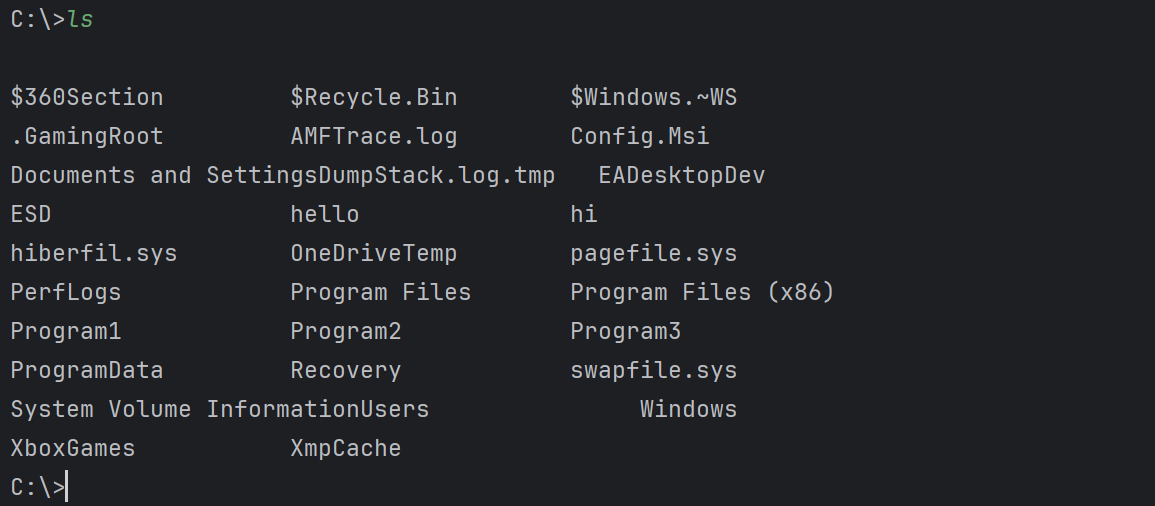




**运行输出**

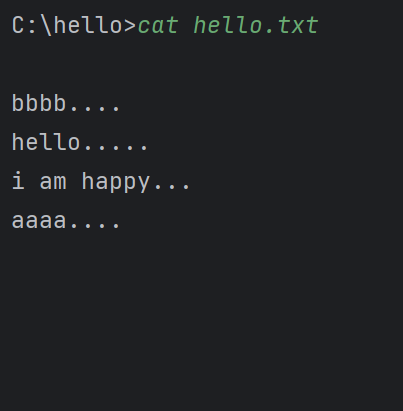
**ls命令**

列出当前目录下的文件和子目录：



**cat命令**

显示文件内容：



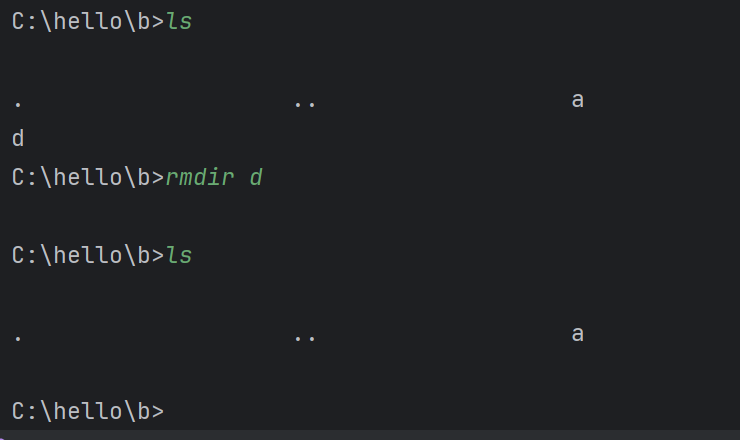
**mkdir命令**

创建新目录：



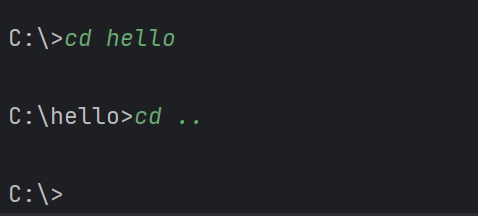
**rmdir命令**

删除空目录：



**cd命令**

改变当前工作目录：



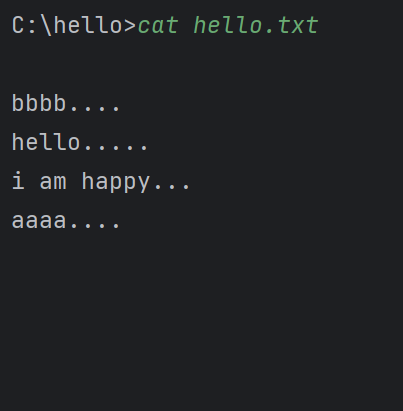
**cp命令**

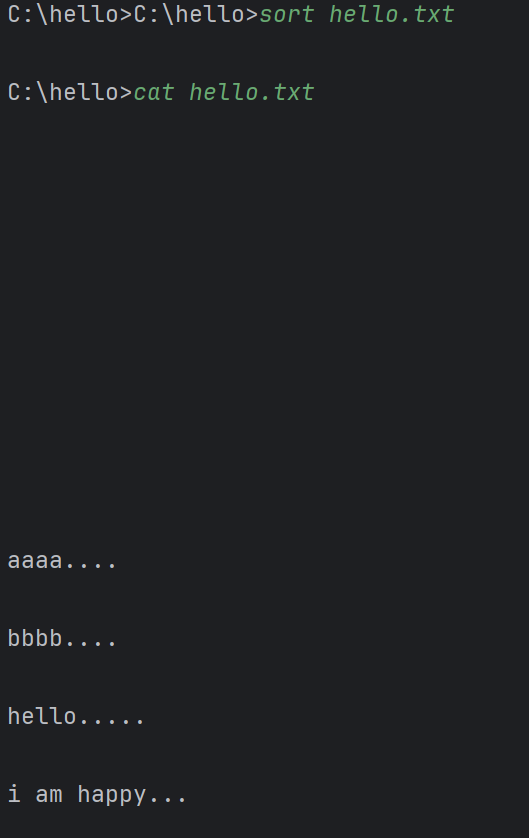
复制文件：



**sort命令**

对文件中的行进行排序：





## 总结和感想体会

1. 深入理解操作系统命令的实现原理

通过为Windows设计兼容Linux命令的接口，我深入理解了操作系统命令的底层实现原理。每个命令的背后都涉及到文件系统、内存管理、进程控制等核心操作系统的概念。例如：

cd 命令：需要调用 SetCurrentDirectory 和 GetCurrentDirectory 函数来改变和获取当前工作目录。这让我了解到Windows API如何管理文件路径和目录结构。

ls 命令：使用 FindFirstFile 和 FindNextFile 来遍历目录中的文件，并通过 GetFileTime 获取文件的时间戳信息。这让我明白了文件元数据的获取方式以及文件时间的转换。

mkdir 和 rmdir 命令：分别使用 CreateDirectory 和 RemoveDirectory 来创建和删除目录。这让我了解了Windows文件系统的目录管理机制。

2. 提升跨平台编程能力

在为Windows实现这些命令时，我深刻体会到了跨平台编程的挑战和乐趣。虽然Windows和Linux在API层面存在差异，但通过合理的设计和封装，可以实现功能相似的命令。例如：Windows使用反斜杠（\）作为路径分隔符，而Linux使用正斜杠（/）。为了确保代码的可移植性，我在代码中使用了统一的路径拼接方式，并在必要时进行路径格式的转换。Windows和Linux在文件操作上有一些细微的差异，如文件权限、符号链接等。通过学习Windows API，我掌握了如何在Windows上模拟Linux的文件操作行为。

3. 熟悉Windows API的使用

这次项目让我对Windows API有了更深入的了解。Windows API提供了丰富的函数库，涵盖了从文件系统到网络通信的各种功能。通过实际编写代码，我熟悉了许多常用的API函数，如：

文件和目录操作：CreateFile, FindFirstFile, FindNextFile, RemoveDirectory, CreateDirectory 等。

文件属性和时间：GetFileTime, FileTimeToLocalFileTime, FileTimeToSystemTime 等。

内存管理：malloc, free 等动态内存分配函数。

4. 掌握命令行解析技巧

在实现这些命令的过程中，我学会了如何解析用户输入的命令行参数。命令行解析是命令行工具的核心功能之一，它决定了程序如何根据用户的输入执行相应的操作。通过编写代码，我掌握了以下技巧：

分割命令行参数：使用空格或特定字符（如 - 或 --）将命令行参数分割成多个部分。

参数验证：检查用户输入的参数是否合法，如文件是否存在、路径是否正确等。

帮助信息：为每个命令提供详细的帮助信息，方便用户理解和使用。

5. 提高代码复用性和模块化设计

为了使代码更加简洁和易于维护，我采用了模块化设计的思想。每个命令都被封装成一个独立的函数，这样不仅可以提高代码的复用性，还可以方便地进行单元测试和调试。例如：

cd\_cmd：负责处理 cd 命令的逻辑。

ls\_cmd：负责处理 ls 命令的逻辑。

mkdir\_cmd 和 rmdir\_cmd：分别负责处理 mkdir 和 rmdir 命令的逻辑。

这种模块化设计不仅提高了代码的可读性和可维护性，还使得后续的功能扩展变得更加容易。

6. 学会处理错误和异常情况

在实现这些命令的过程中，我遇到了许多错误和异常情况，如文件不存在、权限不足、内存分配失败等。通过不断调试和优化代码，我学会了如何优雅地处理这些异常情况。例如：

错误提示：当命令执行失败时，提供清晰的错误提示信息，帮助用户快速定位问题。

容错机制：在某些情况下，即使命令执行失败，也不应导致程序崩溃。通过合理的错误处理机制，可以确保程序的稳定性。

7. 提升用户体验

最后，通过实现这些命令，我更加注重用户体验。一个好的命令行工具不仅要功能强大，还要易于使用。因此，在设计过程中，我特别关注以下几点，尽量保持命令语法的简洁性，避免过于复杂的参数组合。无论是成功还是失败的操作，都应该提供清晰的提示信息，帮助用户理解当前的状态。

通过为Windows设计兼容Linux命令的接口，我不仅提升了编程技能，还对操作系统的工作原理有了更深入的理解。这次项目让我意识到，跨平台编程虽然充满挑战，但也充满了无限的可能性。

## 附录1：程序清单(部分)

char buf[100], path[100], path1[100], path2[100]; // 全局变量，buf用于缓存数据，3个path用于保存路径

/\*cd命令\*/

void cd\_cmd(char\* route) {

if (!SetCurrentDirectory(route)) {

printf("路径不存在!\n");

printf("%s>", path);

}

else {

SetCurrentDirectory(route);

GetCurrentDirectory(100, path);

printf("%s>", path);

}

}

/\*ls命令\*/

void ls\_cmd(char\* array) {

HANDLE handle;

int filenum = 0;

FILETIME lpCreationTime, lpLastAccessTime, lpLastWriteTime;

SYSTEMTIME stime;

WIN32\_FIND\_DATA\* path2 = (WIN32\_FIND\_DATA\*)malloc(sizeof(WIN32\_FIND\_DATA));

if (!GetCurrentDirectory(50, path1)) {

printf("路径不存在!\n");

return;

}

strcat(path1, "\\\*.\*"); // DOS通配符

if ((handle = FindFirstFile(path1, path2)) != INVALID\_HANDLE\_VALUE) {

do {

HANDLE hDir = CreateFile(path2->cFileName, GENERIC\_READ, FILE\_SHARE\_READ | FILE\_SHARE\_DELETE,

NULL, OPEN\_EXISTING, FILE\_FLAG\_BACKUP\_SEMANTICS, NULL);

if (GetFileTime(hDir, &lpCreationTime, &lpLastAccessTime, &lpLastWriteTime)) {

FILETIME ftime;

FileTimeToLocalFileTime(&lpLastWriteTime, &ftime); // 转换成本地时间

FileTimeToSystemTime(&ftime, &stime); // 转换成系统时间格式

}

// printf("%04d-%02d-%02d %02d:%02d %s\n", stime.wYear, stime.wMonth, stime.wDay, stime.wHour, stime.wMinute, path2->cFileName);

if (strcmp(array, "ls") == 0) {

printf("%-20s", path2->cFileName);

if (filenum % 3 == 2) {

printf("\n");

}

filenum++;

}

else {

printf("%04d-%02d-%02d %02d:%02d %s\n", stime.wYear, stime.wMonth, stime.wDay, stime.wHour, stime.wMinute, path2->cFileName);

}

CloseHandle(hDir);

} while (FindNextFile(handle, path2));

}

FindClose(handle);

printf("\n");

GetCurrentDirectory(100, path1);

printf("%s>", path1);

}

/\*mkdir命令\*/

void mkdir\_cmd(char\* array) {

GetCurrentDirectory(100, path);

strcat(path, "\\");

strcat(path, array);

WIN32\_FIND\_DATA wfd;

HANDLE hFind;

hFind = FindFirstFile(path, &wfd);

if (hFind != INVALID\_HANDLE\_VALUE) {

printf("该目录已存在，创建失败!\n");

GetCurrentDirectory(100, path);

printf("%s>", path);

}

else {

CreateDirectory(array, NULL);

GetCurrentDirectory(100, path);

printf("%s>", path);

}

}

/\*rmdir命令\*/

void rmdir\_cmd(char\* array) {

GetCurrentDirectory(100, path);

strcat(path, "\\");

strcat(path, array);

WIN32\_FIND\_DATA wfd;

HANDLE hFind;

hFind = FindFirstFile(path, &wfd);

if (hFind == INVALID\_HANDLE\_VALUE) {

printf("该目录不存在，删除失败!\n");

GetCurrentDirectory(100, path);

printf("%s>", path);

}

else {

RemoveDirectory(array);

GetCurrentDirectory(100, path);

printf("%s>", path);

}

}

/\*cp命令\*/

void cp\_cmd(char\* array1, char\* array2) {

char fullSourcePath[100], fullDestPath[100];

GetCurrentDirectory(100, fullSourcePath);

strcat(fullSourcePath, "\\");

strcat(fullSourcePath, array1);

GetCurrentDirectory(100, fullDestPath);

strcat(fullDestPath, "\\");

strcat(fullDestPath, array2);

if (!CopyFile(fullSourcePath, fullDestPath, FALSE)) {

printf("复制失败，请检查路径和文件是否存在！错误代码: %lu\n", GetLastError());

} else {

printf("文件复制成功: %s -> %s\n", fullSourcePath, fullDestPath);

}

GetCurrentDirectory(100, path);

printf("%s>", path);

}

/\*cat命令\*/

void cat\_cmd1(char\* array) { // 用于显示文件

char ch[1000];

GetCurrentDirectory(100, path);

strcat(path, "\\");

strcat(path, array);

FILE\* infile = fopen(path, "r");

if (!infile) {

printf("无法打开文件%s\n", path);

GetCurrentDirectory(100, path);

printf("%s>", path);

}

else {

while (fgets(ch, 1000, infile)) {

printf("%s", ch);

}

GetCurrentDirectory(100, path);

printf("%s>", path);

}

fclose(infile);

}

void cat\_cmd2(char \*array1, char \*array2) {

char ch[1000];

GetCurrentDirectory(100, path); // 保存cat操作之前的路径

strcpy(path1, path); // 复制当前路径到path1

strcat(path1, "\\");

strcpy(path2, path); // 复制当前路径到path2

strcat(path2, "\\");

strcat(path1, array1); // 构造输入文件的完整路径

strcat(path2, array2); // 构造输出文件的完整路径

FILE \*infile1 = fopen(path1, "r");

if (!infile1) {

printf("无法打开文件%s\n", path1);

GetCurrentDirectory(100, path1);

printf("%s>", path1);

} else {

FILE \*outfile2 = fopen(path2, "a"); // 以追加模式打开输出文件

if (!outfile2) {

printf("无法打开文件%s\n", path2);

GetCurrentDirectory(100, path2);

printf("%s>", path2);

} else {

while (fgets(ch, 1000, infile1) != NULL) { // 读取输入文件并写入输出文件

fputs(ch, outfile2);

}

fclose(infile1);

fclose(outfile2);

FILE \*infile2 = fopen(path2, "r"); // 重新打开输出文件以读取内容

if (infile2) {

while (fgets(ch, 1000, infile2) != NULL) { // 读取并打印输出文件的内容

printf("%s", ch);

}

fclose(infile2);

}

}

}

printf("%s>", path);

}

/\*sort命令\*/

void cmd\_sort(char \*array) {

int lnum = 0, min;

char line[100][100], tmp[100]; // 行数、每行字符数

// 获取当前目录并拼接文件名

GetCurrentDirectory(100, path);

strcat(path, "\\");

strcat(path, array);

FILE \*infile = fopen(path, "r");

if (!infile) {

printf("无法打开文件%s\n", array);

GetCurrentDirectory(100, path);

printf("%s>", path);

} else {

// 读取文件内容并记录行数

while (fgets(line[lnum], 100, infile) != NULL) {

lnum++; // 记录总行数

}

fclose(infile);

// 简单的冒泡排序

for (int x = 0; x < lnum - 1; x++) {

min = x;

for (int y = x + 1; y < lnum; y++) {

if (strcmp(line[y], line[min]) < 0) {

min = y;

}

}

if (min != x) {

strcpy(tmp, line[x]);

strcpy(line[x], line[min]);

strcpy(line[min], tmp);

}

}

// 将排序后的内容写回文件

FILE \*outfile = fopen(path, "w");

if (!outfile) {

printf("文件打开失败\n");

} else {

for (int c = 0; c < lnum; c++) {

fprintf(outfile, "%s\n", line[c]);

}

fclose(outfile);

}

GetCurrentDirectory(100, path);

printf("%s>", path);

}

}