**基于C语言实现**

**Linux自定义shell**

**设计报告**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 小组成员 | 林浩 | 钟在东 | 蒋盈楠 | 丘炜林 |
| 成员学号 | 21312596 | 21312783 | 21312649 | 21312615 |
| 学 院 | 网络空间安全学院 | | | |
| 完成时间 | 2024年1月5日 | | | |

目录

一、项目设计目的 ..3

二、项目预期计划 3

三、项目具体实现 4

3.1、shell整体框架 4

3.1.1、指令输入基本处理 3

3.1.2、指令函数调用 6

3.1.3、外部指令调用逻辑 7

3.2、shell内部指令实现 7

3.2.1、help指令 7

3.2.2、cd指令 10

3.2.3、pwd指令 13

3.2.4、history指令 13

3.2.5、exit指令 14

3.3、shell外部指令实现 15

3.3.1、tree指令 15

3.3.2、ps指令 18

3.3.3、cp指令 24

3.3.4、mv指令 30

3.3.5、ls指令 32

3.3.6、rm指令 37

四、项目总结 42

一、 项目设计目的

1.1 通过设计一个自定义的 Shell 解释器，加深对 Linux 系统的底层系统调用和解释器工作原理的理解。

1.2 综合运用操作系统原理中所学的知识和 Linux 系统实验中锻炼的动手

能力完成实验。

二、 项目预期计划

2.1（基础）编写程序实现一个自定义的 Linux 的 Shell 命令解释器，要求实现以下基本功能：

**·**命令帮助信息：类似于help 命令或man 命令。该功能提供一个命令，用于显示设计的自定义Shell 解释器支持的所有命令及其用法。用户可以随时调用此命令，以获取对其他命令的简要描述和正确使用方式的指导。

**·**显示指定目录文件：类似于 ls 命令。该功能提供一个命令，用于列出指定目录下的所有文件和文件夹。用户可以通过这个命令查看当前工作目录中的内容。

**·**切换工作目录：类似于cd 命令。该功能提供一个命令，允许用户改变当前工作目录，切换到指定的路径。

**·**复制文件、文件夹：类似于 cp 命令。该功能提供一个命令，允许用户将

源文件或目录复制到指定的目标文件或目录中。

**·**移动文件、文件夹：类似于 mv 命令。该功能提供一个命令，用户可以调用它来对文件或目录重新命名，或者将文件从一个目录移到另一个目录中。

**·**删除文件、文件夹：类似于 rm 命令。该功能提供一个命令，用于删除文

件或目录，也可以将某个目录及其下属的所有文件及其子目录递归删除。

**·**显示工作目录：类似于pwd 命令。该功能提供一个命令，用于显示当前工作目录的路径。用户可以通过调用此命令立刻获取目前所在的工作目录的绝对路径名称。

**·**显示输入命令历史：类似于 history 命令。该功能提供一个命令，用户可

以调用它来显示之前输入的命令历史记录，方便回顾和重用命令。

**·**显示进程信息：类似于ps 命令。该功能提供一个命令，用户可以调用它

来获取有关系统中运行进程的详细信息，例如进程 ID、状态和资源使用情况等。

**·**显示目录结构：类似于 tree 命令。该功能提供一个命令，以树状结构显

示目录的内容。执行命令会列出指定目录下的所有文件，包括子目录里的文件。

**·**输入输出重定向： 允许用户使用符号将命令的输入或输出重定向到文件，实现更灵活的输入输出控制。

**·**管道：类似于[ | ]操作符，该功能允许用户将一个命令的输出作为另一个命令的输入，实现命令之间的数据流传递。

**·**后台运行程序：类似于[&]操作符，该功能使得某个程序在后台运行，允许用户继续输入其他命令而不阻塞 Shell。

2.2（进阶）在完成上述基础功能后，进一步了解Shell解释器的底层设计，尝试在自定义的Shell 解释器中实现更复杂的功能，如外部指令调用等。

2.3 对实现的功能逐一进行测试，确保程序的完整性和正确性。

三、 项目具体实现

3.1 Shell整体框架

框架在整体上负责指令输入处理以及指令函数的调用。同时，内部指令以及重定向，管道，后台功能是作为函数放在shell同一个代码文件下。他们虽然与框架在同一份代码，但在逻辑上不属于框架部分。

现对框架的主要实现做详细介绍。

**3.1.1 指令输入基本处理**

首先Shell的整体逻辑是不断获取指令的输入并做相应的指令处理。那么Shell的main主函数很好理解，就是一个不断接收输入的循环函数，直到遇到错误或者退出。我们将输入的指令首先当作一个字符串，调用自定义get\_input函数把指令存在全局变量buf，然后运行parse(buf)进行指令分割，分割指令后运行do\_cmd(argc, argv)调用指令对应的函数。

|  |
| --- |
| while(1) {  // 输入字符存入buf数组, 如果输入字符数为0, 则跳过此次循环  if (get\_input(buf) == 0)  continue;  // strcpy(history[commandNum++], buf);  strcpy(backupBuf, buf);  parse(buf);  do\_cmd(argc, argv);  argc = 0;  } |

get\_input函数在接受输入的同时负责输出shell的信息提示，比如当前路径和shell的提示符：



这部分通过全局变量curPath存储当前目录路径，（在未实现自定义的pwd内置函数之前，通过unistd.h的getcwd函数获得当前目录路径）。

|  |
| --- |
| char curPath[BUFFSIZE];  getcwd(curPath, sizeof(curPath));  char now[2 \* BUFFSIZE] = "[myshell]:";  strcat(now, curPath);  strcat(now, "$ "); |

而在具体处理输入问题上，为了方便实现光标移动以及上下文指令的调用（这些涉及到键盘鼠标与屏幕的交互处理，实现较为复杂），我们调用封装好的readline库来处理我们的具体输入。同时我们用readline库的add\_history函数把输入过的指令存储在库指定的目录。

|  |
| --- |
| char \*input = readline(now);  if (!input) {  return 0; // EOF or error  }  add\_history(input); |

Parse函数负责对指令分割，因为指令输入是一个字符串比如“ls -a” 有两个参数一个ls一个-a，只有分割指令获得相应的参数之后才能方便进一步指令调用。

其中用到几个关键的全局变量来存储参数和参数数量。Command二维字符数组、argv字符指针数组，argc整型数字变量。

|  |
| --- |
| int argc; // 有效参数个数  char\* argv[MAX\_CMD]; // 参数指针数组  char command[MAX\_CMD][MAX\_CMD\_LEN]; // 参数数组 |

例如ls -a -l 那么argc为3，command[0]是ls，command[1]是-a，command[0][0]是l，command[0][1]是s，argv[0]是指向command[0]的指针，也就是ls。至于为什么要多设计一个argv指针数组，是为了后续指令函数的参数读取，用指针更方便（直接对变量本体操作）。

Parse的指令分割很简单直观，就靠字符串的空格来划分参数。然后分别把几个字符串存到以上的几个全局变量。

3.1.2 指令函数调用

接着就是do\_cmd(argc, argv)函数，它首先遍历参数查看是否有重定向、管道、后台命令的指定功能的特殊符号。比如>，｜，&。如果存在则需要跳转到对应的自定义函数进行处理。

在遍历检查完，确保参数没有指定功能的特殊符号之后，我们就将首先在内部指令数组里遍历检索第一个参数，查看其是否为内部指令函数。

我们定义的内部指令结构体数组很关键，它决定了内部指令调度的简洁性和整体框架的代码布局的有序。

首先定义cmd\_map结构体，其中包含指令名称和指令的函数：

|  |
| --- |
| typedef struct {  char\* cmd;  cmd\_handler handler;  } cmd\_map; |

那么我们维护一个cmd\_map的结构体数组，我们要实现的内部指令函数都放在里面以供查询和调用：

|  |
| --- |
| cmd\_map commands[] = { // 命令映射表 注意区分command[][]和commands[],  {"cd", handle\_cd},  {"history", handle\_history},  {"exit", handle\_exit},  {"help", handle\_help},  {"pwd", handle\_pwd},  {"top",handle\_top},  // ... 其他命令  {NULL, NULL} // 结尾哨兵  }; |

那么很自然的，我们检索第一个参数的时候只需要将其在数组里遍历比较每一个结构体的第一个元素即可。匹配成功则调用对应函数。也很自然的，我们的内部指令函数都统一用handle\_前缀命名，方便管理。

3.1.3外部指令调用逻辑

如果内部指令数组没有匹配成功，我们尝试检索我们实现的外部指令。我们的外部指令调用函数execute\_external\_command(char\* argv[])同样根据第一个参数来检索外部指令。

这里需要说明我们外部指令的实现和调用方式。我们将每一条外部指令通过编译成c程序存储在我们指定外部指令目录：

|  |
| --- |
| #define MY\_COMMANDS\_DIR "./mycommands/" |

那么我们检索外部指令时候就只需要在该目录下检索指令对应的程序，如果我们需要新增或修改外部指令也只需要在该目录下增加或修改程序即可，非常方便。

既然是外部指令，我们不自然能把指令当作shell代码程序下的一个函数，而是要当做外部的一个程序。那么我们就需要创建Linux子进程来执行程序。通过pid\_t pid = fork();创建子进程，创建成功则使用unistd.h 库的execv(command\_path, argv)来调用相应的程序。这里的command\_path是通过字符串拼接（前面提到的MY\_COMMANDS\_DIR+第一个参数比如ls）得到的程序路径。而我们的外部指令程序的命名就要根据指令来命名，否则execv就找不到这个程序。

3.2 Shell内部指令实现

**3.2.1 help指令**

该指令类似于Linux中的help命令，用于显示设计的自定义Shell解释器支持的所有内部指令及其用法。用户可以随时调用此命令，以获取对其他命令的

简要描述和正确使用方法的指导。

用if-else语句判断用户需要显示的指令，然后打印出指令的相关信息。

如果不输入参数，则输出所有内部指令。

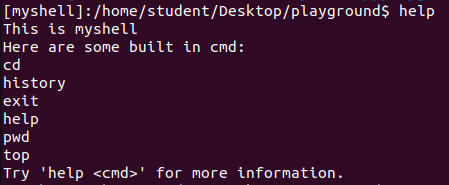
如果参数为某个命令，则输出该命令的用法。

主要代码如下：

|  |
| --- |
| // 不输入指定命令，则输出所有命令      if (argc == 1){          printf("This is myshell\n");          printf("Here are some built in cmd:\n");          for (int i = 0; i < sizeof(commands) / (sizeof(commands[0].cmd) + sizeof(commands[0].handler)) - 1; i++){              printf("%s\n", commands[i].cmd);          }          printf("Try 'help <cmd>' for more information.\n");      }      // 查看指定命令的帮助      else if (argc == 2){          // 查看cd命令的帮助手册          if (strcmp("cd", argv[1]) == 0){              printf("cd: Change the shell working directory.\n");              printf("   Options:\n");              printf("      'cd':     change the current directory to home\n");              printf("    'cd ~':     same as 'cd'\n");              printf("'cd [dir]':     change the current direcrtory to dir, the dir can be absolute path relative path\n");              printf("   'cd ..':     change to the parent directory\n");              printf("    'cd -':     change to the last directory you visited\n");          }          // 查看history命令的帮助手册          else if (strcmp("history", argv[1]) == 0) {              printf("history: Review the commands used earlier.\n");              printf("      Options:\n");              printf("    'history':     review the all commands\n");              printf("'history [n]':     review the last n commands\n");          }          // 查看exit命令的帮助手册          else if (strcmp("exit", argv[1]) == 0) {              printf("exit: Exit the shell.\n");              printf("  Options:\n");              printf("   'exit':     exit the shell with status of the last command executed\n");          }          // 查看top命令的帮助手册            else if (strcmp("top", argv[1]) == 0) {              printf("top: Display Linux processes.\n");              printf("   Options:\n");              printf("  'top':     Display Linux processes and all related information\n");          }            // 查看help命令的帮助手册          else if (strcmp("help", argv[1]) == 0) {              printf("help: Display information about builtin commands.\n");              printf("    Options:\n");              printf("     'help':     display all builtin commands\n");              printf("'help <cmd>:     display information of the given command\n'");          }          // 查看pwd命令的帮助手册          else if (strcmp("pwd", argv[1]) == 0) {              printf("pwd: Print the name of the current working directory.\n");              printf("    Options:\n");              printf("   'pwd -L':     print the value of $PWD if it names the current working directory\n");              printf("   'pwd -P':     print the physical directory, without any symbolic links\n");          }      } |

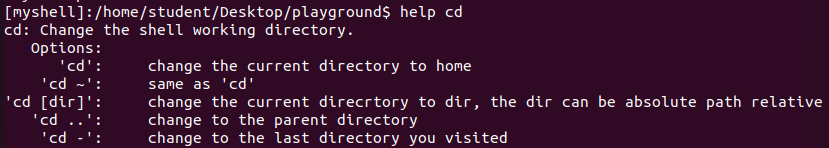
下面是具体的测试结果截图：

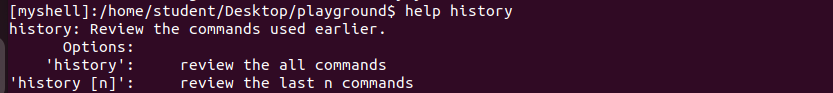
①help

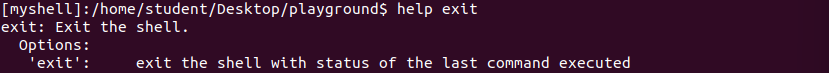


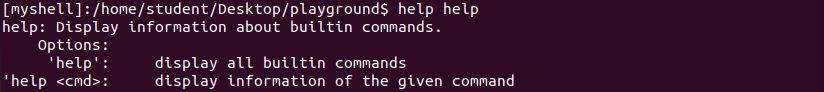
不输入指定命令，则输出所有内置命令。

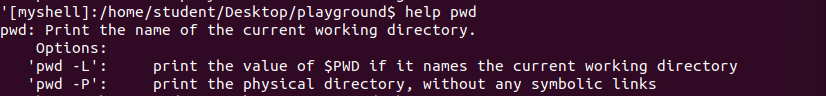
②help cd/history/exit/help/pwd/top

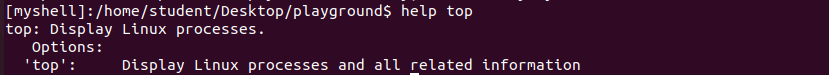












**3.2.2 cd指令**

该指令允许用户改变当前工作目录，切换到指定的路径。

使用chdir函数切换路径。当需要返回主目录时，通过环境变量获取主目录地址，然后用chdir更新到主目录。为了实现返回上一个访问的目录的功能，我们定义了一个全局变量存储上一个访问的目录，并在每次更新访问的目录时更新这个全局变量。

主要代码如下：

|  |
| --- |
| static char prevPath[BUFFSIZE] = "";  // 用于存储上一个工作目录的路径      char curPath[BUFFSIZE];  // 用于存储当前工作目录的路径      // 获取当前工作目录      if (getcwd(curPath, sizeof(curPath)) == NULL) {          perror("getcwd");          return 0; // 获取当前工作目录失败      }      // 'cd -' 返回上一个访问的目录      if (argc == 2 && strcmp(argv[1], "-") == 0) {            if (strlen(prevPath) == 0) {              printf("没有记录的上一个工作目录\n");              return 0; // 如果没有记录的上一个目录          }          // 切换到上一个工作目录          if (chdir(prevPath) != 0) {              perror("chdir");              return 0; // 执行失败          }          // 更新上一个工作目录信息          strcpy(prevPath, curPath);          return 1; // 执行成功      }      // 'cd '或'cd ~' 返回主目录      else if ((argc == 1) | (argc == 2 && strcmp(argv[1], "~") == 0))      {          // 主目录通常由环境变量指定          // 因此通过环境变量获取主目录          char\* home\_dir = getenv("HOME");          if (home\_dir == NULL){              printf("Failed to get home directory.\n");          } else {              printf("Home directory: %s\n", home\_dir);          }          // 更新到主目录          chdir(home\_dir);  // 更新上一个工作目录信息          strcpy(prevPath, curPath);          // 获取当前目录          getcwd(curPath, sizeof(curPath));  }      // 'cd <路径>' 根据相对路径或绝对路径切换      else if (argc == 2) {            if (chdir(argv[1]) != 0) {              perror("chdir");              return 0; // 执行失败          }          // 更新上一个工作目录信息          strcpy(prevPath, curPath);          // 获取新的工作目录          if (getcwd(curPath, sizeof(curPath)) == NULL) {              perror("getcwd");              return 0; // 获取新工作目录失败          }          return 1; // 执行成功      }      else {          printf("用法错误: cd <目录>\n");          return 0; // 参数错误      } |

下面是具体的测试结果截图：

①cd testcd

打开当前目录中的一个文件夹testcd。



cd ..

打开当前目录的上一级目录。



cd testcd/testcd1

打开当前目录文件夹中的一个文件夹testcd1。



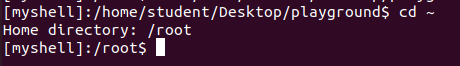
cd /home/student/Desktop/playground

使用绝对路径切换目录。



②cd ~

返回主目录，并打印主目录的位置。



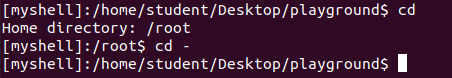
cd

返回主目录，并打印主目录的位置。



③cd -

返回上一个访问的目录。



**3.2.3 pwd指令**

该指令实现了一个简单的pwd命令，用于显示当前工作目录的路径，用户可以通过调用此命令立刻获取目前所在的工作目录的绝对路径。这个指令的实现是通过调用getcwd()函数实现的。

主要代码如下：

|  |
| --- |
| if (getcwd(curPath, sizeof(curPath)) != NULL){          printf("%s\n", curPath);      } else {          perror("getcwd() error");      } |

下面是具体的测试结果截图：

①pwd



**3.2.4 history指令**

该指令用于显示之前输入的命令历史记录，方便回顾和重用命令。该指令通过指针来获取Linux中的history\_list()，再根据用户的需求输出对应的历史记录。

①如果用户没有指定输出历史记录的数量，则将从开启这个Shell开始的历史记录都输出。

②如果用户指定了输出历史记录的数量，则输出最近使用的n条指令。

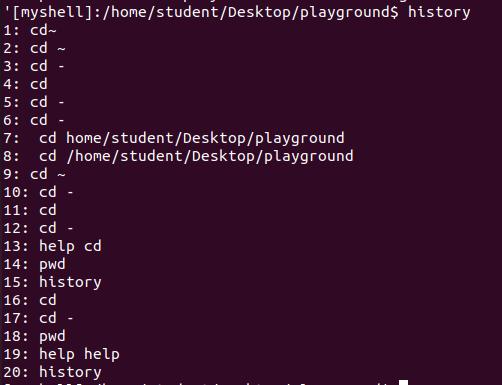
主要代码如下：

|  |
| --- |
| int history\_count = 0;      HIST\_ENTRY \*\*the\_list = history\_list();      if (the\_list) {          while (the\_list[history\_count]) {              ++history\_count;          }      }      int n = history\_count; // 默认显示所有历史记录      // 检查是否提供了参数，并且是有效数字      if (argc > 1) {          int requested = atoi(argv[1]);          if (requested > 0 && requested < n) {              n = requested;          }      }      // 显示历史记录      for (int i = history\_count - n; i < history\_count; i++) {          printf("%d: %s\n", i + history\_base, the\_list[i]->line);      } |

下面是具体的测试结果截图：

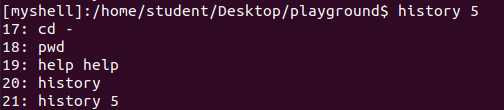
①history

输出从开启Shell开始的所有历史记录。



②history 5

输出最近的5条历史记录。



**3.2.5 exit指令**

该指令实现了一个简单的exit指令，通过调用exit()函数退出myshell程序，可以指定exit的状态，默认exit状态参数为0。

主要代码如下：

|  |
| --- |
| int exit\_status = 0;      if (argc > 1) {          exit\_status = atoi(argv[1]);      }      // 打印退出消息（如果需要）      printf("Exiting shell with status %d\n", exit\_status);      // 退出程序      exit(exit\_status); |

下面是具体的测试结果截图：

①exit

退出shell程序并打印退出信息，返回Linux系统的shell。



3.3 Shell外部指令实现

**3.3.1 tree指令**

tree指令主要实现了这几个功能：**1**、**tree**不加任何后缀，只输出当前目录下的文件树状结构，**2**、**tree -f**输出完整的路径名字，**3**、**tree -d**只输出目录不输出文件，**4**、**tree -L level**目录展示第level级，**5**、**tree dir**输出指定目录dir下的树状结构。同时，上面的指令可以叠加运行，如**tree -f -d -L 1 dir**，实现完整的输出。下面是具体的代码实现流程：

首先定义了一系列**全局变量**来控制程序的行为和存储关键信息。这些变量在程序的不同部分被访问和修改，使得程序可以根据用户的输入和程序的执行状态改变其行为。

|  |
| --- |
| int only\_directories = 0; //标记是否只显示目录  //定义两个全局变量来存储文件夹和文件的数量  int dir\_count = 0;  int file\_count = 0;  int full\_paths = 0; // 0 表示不显示完整路径，1 表示显示完整路径  int max\_level = -1; // 如果这个值为-1，则没有层级限制 |

为了给程序提供一种统一和有效的方式来处理运行时遇到的错误，我们实现了一个自定义的错误处理函数 **my\_error**。函数使用**perror**标准库函数输出传递给它的错误消息。在打印错误消息后，函数调用**exit**函数并传递状态码1，用于指示程序因错误而非正常结束

|  |
| --- |
| // 自定义错误处理函数  void my\_error(const char \*strerr)  {  perror(strerr);  exit(1);} |

为了提高tree的输出可读性，我们在自定义引入了 **ANSI** 颜色代码。

**#define COLOR\_DIR "\x1b[34m"**：定义蓝色输出，用于目录名。这里**\x1b[34m** 是 **ANSI** 转义序列，用于设置后续文本的颜色为蓝色。

**#define COLOR\_RESET "\x1b[0m"**：定义重置颜色的代码。它用于在输出了带颜色的文本之后恢复到默认颜色，避免颜色设置影响到后续的文本输出。

这些宏在打印目录名时被使用。具体来说，当程序打印一个目录名时，它首先使用 **COLOR\_DIR** 宏来设置文本颜色为蓝色，然后打印目录名，最后使用 **COLOR\_RESET** 宏来重置文本颜色。

|  |
| --- |
| // 定义ANSI颜色代码  #define COLOR\_DIR "\x1b[34m" // 蓝色  #define COLOR\_RESET "\x1b[0m" // 重置颜色 |

然后是tree的核心部分**PrintDirentStruct**函数，该函数接收路径名和层级两个变量，负责遍历目录并根据设定的选项（如是否仅显示目录、是否显示完整路径）打印出目录结构。

**·目录遍历循环：**while((p\_dirent = readdir(p\_dir)) != NULL)循环使用 **readdir** 函数逐个读取目录项，进行处理，直到所有项都被读取完毕。

**·打印目录结构**：for(int i = 0; i < level; i++) { ... }，根据目录的层级 level 打印出树状结构前的缩进；printf("|--- ")，在每个文件或目录名前打印，表示树状结构的一部分。

**·完整路径处理**：if (full\_paths) { ... }，如果设置了显示完整路径，使用 snprintf 函数构建完整的文件或目录路径。

**·目录和文件处理：**使用 if(p\_dirent->d\_type == DT\_DIR) { ... } 来判断当前项是否为目录，如果是目录，计数，递归调用 **PrintDirentStruct** 函数来处理子目录。如果是文件，根据only\_directories是否为1来决定要不要打印文件和计数。

|  |
| --- |
| void PrintDirentStruct(char direntName[], int level)  {  if (max\_level != -1 && level >= max\_level) {  return; // 如果达到了最大层级，就不再递归  }  DIR \*p\_dir = NULL;  struct dirent \*p\_dirent = NULL;  p\_dir = opendir(direntName);  while((p\_dirent = readdir(p\_dir)) != NULL)  {  char \*backupDirName = NULL;  for(int i = 0; i < level; i++)  {  printf("|");  printf(" ");  }  printf("|--- "); // 在每个文件或目录名前打印，表示树状结构的一部分  char fullPath[1024];  if (full\_paths) {  snprintf(fullPath, sizeof(fullPath), "%s/%s", direntName, p\_dirent->d\_name);  //显示完整路径  }  if(p\_dirent->d\_type == DT\_DIR)// 判断是否为目录  {  dir\_count++;  if (full\_paths)  {  printf("%s%s%s\n", COLOR\_DIR, fullPath, COLOR\_RESET);  }  else  {  printf("%s%s%s\n", COLOR\_DIR, p\_dirent->d\_name, COLOR\_RESET);  }  int curDirentNameLen = strlen(direntName) + 1;  backupDirName = (char \*)malloc(curDirentNameLen);  memset(backupDirName, 0, curDirentNameLen); //维护需要递归的路径  memcpy(backupDirName, direntName, curDirentNameLen);  strcat(direntName, "/");  strcat(direntName, p\_dirent->d\_name);  PrintDirentStruct(direntName, level + 1);//递归调用函数来处理子目录  memcpy(direntName, backupDirName, curDirentNameLen);  free(backupDirName);  backupDirName = NULL;  }  else if(only\_directories == 0)  {  file\_count++;  if (full\_paths)  {  printf("%s\n", fullPath);  }  else  {  printf("%s\n", p\_dirent->d\_name);  }  }  }  closedir(p\_dir);} |

下面是主函数部分，它主要负责处理命令行参数、调用具体的执行函数以及进行输出，是程序的主要入口点，决定了程序的行为和输出。

它首先定义一个字符数组 **direntName** 来存储目录名，并用 **memset** 将其初始化为零。接着，函数处理命令行参数以配置程序的行为。

|  |
| --- |
| char direntName[256];  memset(direntName, 0, sizeof(direntName)); |

接着，函数处理命令行参数以配置程序的行为。对输入的参数进行处理，根据不同的输入来决定输出的结构。

·**无参数情况 (**argc == 1**)**:当没有提供任何额外参数时（只有程序名），默认目录设置为当前目录（.）

·**有参数情况 (**argc >= 2): 当提供了一个或多个参数时，程序通过遍历这些参数来设置不同的操作模式：**-d**参数将**only\_directories**设置为1，使程序仅显示目录；**-f** 参数将 **full\_paths** 设置为 1，使程序显示文件和目录的完整路径；-L参数跟随一个数字，用于设置目录显示的最大深度，将数字通过atoi函数赋值给**max\_level**。如果参数不是上述选项之一，它被视为目录名并附加到 **direntName。**

|  |
| --- |
| if(argc == 1)  {  direntName[0] = '.'; // 没有参数，路径设置为当前目录(.)  }  else if(argc >= 2)  {  for (int i = 1; i < argc; i++)  {  if (strcmp(argv[i], "-d") == 0)  {  only\_directories = 1;  }  else if(strcmp(argv[i], "-f") == 0)  {  full\_paths = 1;  }  else if(strcmp(argv[i], "-L") == 0 && i + 1 < argc)  {  max\_level = atoi(argv[++i]);  }  else  {  strcat(direntName, argv[i]);  }  } |

下面代码的功能是尝试打开指定的目录，然后调用**PrintDirentStruct**函数实现tree函数的功能，最后将统计到的文件数目和目录数目进行输出。

·**打开目录与错误处理：**使用 **opendir** 函数尝试打开由 **direntName** 指定的目录。**opendir** 返回一个 **DIR** 类型的指针，指向目录流。如果**opendir** 返回 **NULL**，则表明无法打开目录，可能的原因是目录不存在或者是权限不够，这时调用错误处理函数**my\_error**进行处理。

·**目录遍历：**将**direntName** 作为参数传递给 **PrintDirentStruct** 函数，然后开始递归地遍历目录结构。这是程序的核心功能，它详细地打印出目录和（如果不是仅限目录模式）文件的树状结构。

·**结果统计与输出**：在遍历完成后，程序使用 **printf** 函数输出统计的总目录数（**dir\_count**）。如果不是仅显示目录的模式（**only\_directorie**不为 1），程序还会输出统计的总文件数（**file\_count**）。

|  |
| --- |
| DIR \*a\_dir = NULL;  a\_dir = opendir(direntName);  if(a\_dir == NULL)  {  printf("error");  my\_error("opendir error");  }  printf("%s\n", direntName); // 先输出要查看的目录名字，作为树的根。  PrintDirentStruct(direntName, 0);  printf("Total directories: %d\n", dir\_count); // 输出目录数目  if (!only\_directories) {  printf("Total files: %d\n", file\_count); // 输出文件的数目  } |

**3.3.2 ps指令**

ps指令主要实现了这几个功能：1、ps，基础功能，只输出与当前终端有关的进程信息，2、ps -u，输出进程有关的用户信息，3、ps -e，显示所有进程，不局限于本终端，4、ps -l，进一步显示详细信息，如ppid等信息。上面的各种后缀可以组合在一起，如ps -u -l -e显示最完整的进程信息。

下面是具体的代码展示：

函数 **get\_username** 的目的是根据用户ID（UID）获取相应的用户名。通过调用 **getpwuid**来查找对应UID 的一些信息，如用户名、密码等。如果**pw**不是空的，则返回**pw->pw\_name**，即用户的名称。

主要作用是根据进程的UID来得到相应的用户名。

|  |
| --- |
| char\* get\_username(uid\_t uid) {  struct passwd \*pw = getpwuid(uid);  return (pw != NULL) ? pw->pw\_name : "UNKNOWN";  } |

函数 **get\_tty\_name** 的目的是获取与特定进程PID关联的终端（TTY）的名称。根据进程的**pid**信息通过检查 **/proc/[pid]/fd** 目录来寻找与 TTY 关联的文件描述符。

打开**/proc/[pid]/fd**，然后循环读取检查每个条目是否为符号链接（**DT\_LNK**），因为有效的 TTY 文件描述符通常是符号链接。若是，则打开符号连接，使用 **ttyname\_r** 函数尝试获取关联的TTY名称，然后将其复制到**tty\_name**参数中。

|  |
| --- |
| int get\_tty\_name(int pid, char \*tty\_name, size\_t tty\_name\_size) {  char fd\_path[256];  sprintf(fd\_path, "/proc/%d/fd", pid);  DIR \*fd\_dir = opendir(fd\_path);  if (fd\_dir == NULL) {  return -1; // 无法打开文件  }  struct dirent \*fd\_entry;  while ((fd\_entry = readdir(fd\_dir)) != NULL) {  if (fd\_entry->d\_type != DT\_LNK) {// 判断是否为符号连接  continue; // 不是符号连接，继续往下读  }  char link\_path[256];  sprintf(link\_path, "%s/%s", fd\_path, fd\_entry->d\_name);  int fd = open(link\_path, O\_RDONLY);  if (fd == -1) {  continue;  }  if (ttyname\_r(fd, tty\_name, tty\_name\_size) == 0) {// 获取TTY  close(fd);  closedir(fd\_dir);  return 0; // 找到终端  }  close(fd);  }  closedir(fd\_dir);  return -1; // 没找到终端信息  } |

这段代码中的 **display\_process\_info** 函数是一个用于展示系统中特定进程的详细信息的功能。它利用了 Linux 系统中的 **/proc** 文件系统，这是一个伪文件系统，通常被用于提供对内核和进程信息的访问接口。该函数通过读取和解析位于 **/proc** 下的特定文件，能够提取出关于特定进程的各种信息。

·**读取和解析/proc/[pid]/stat文件：**/proc/[pid]/stat这个文件包含了进程的基本信息，如进程ID（PID）、进程名称（命令名）、父进程ID（PPID）等。使用格式化输入（fscanf）来提取特定的信息，比如PID和PPID。

·**读取和解析/proc/[pid]/status文件：**/proc/[pid]/status 文件提供了更为详细的进程信息，包括进程的用户ID（UID）函数解析这个文件以获取进程的 UID。获取 UID 后，通过调用前面介绍过的函数 **get\_username** 将 UID 转换为我们比较熟悉的用户名。

**·获取进程的终端信息：**同样调用前面的**get\_tty\_name**函数，传入进程的PID信息，将返回值赋给变量**tty\_name。**然后与当前终端的值进行比较，如果不相等且没有要求要输出全部进程，则直接**return**；否则输出进程信息。

·**定制输出内容**：函数根据传入的参数（show\_user\_info、show\_details 和 show\_all）定制输出内容，这些参数允许用户选择是否展示用户信息、详细信息，或者是否限制输出到当前终端的进程。

|  |
| --- |
| void display\_process\_info(int pid, int show\_user\_info, int show\_details, int show\_all) {  char stat\_path[256];  char status\_path[256];  // Paths for stat and status files  sprintf(stat\_path, "/proc/%d/stat", pid);  sprintf(status\_path, "/proc/%d/status", pid);  // Open stat file  FILE \*stat\_file = fopen(stat\_path, "r");  if (stat\_file == NULL) {  perror("fopen");  return;  }  // Read process information from stat file  int process\_pid, ppid;  char process\_comm[256];  fscanf(stat\_file, "%d %s %\*c %d", &process\_pid, process\_comm, &ppid);  fclose(stat\_file);  // Open status file  FILE \*status\_file = fopen(status\_path, "r");  if (status\_file == NULL) {  perror("fopen");  return;  }  // Read user information from status file  uid\_t uid;  while (fscanf(status\_file, "Uid: %u", &uid) != 1) {  while (fgetc(status\_file) != '\n'); // Skip rest of the line  }  fclose(status\_file);  // Get username from UID  char \*user = get\_username(uid);  // Get tty name  char tty\_name[256] = "?";  get\_tty\_name(pid, tty\_name, sizeof(tty\_name));  if(strcmp(tty\_name, "/dev/pts/0") != 0&&show\_all)  {  return;  }  } |

最后是主函数部分，主要负责处理命令行参数、调用具体的执行函数以及进行输出，决定了程序的行为和输出

**·解析命令行参数：**使用 getopt 函数解析命令行参数，以决定程序的输出格式。-l：显示详细信息（设置 show\_details 标志），-u：显示用户信息（设置 show\_user\_info 标志），-e：显示所有进程（更改 show\_all 标志）。

**·调用函数显示进程信息：**遍历/proc目录，检查每个条目是否为目录，并且其名称是否为数字（代表PID），对于每个符合条件的目录（即进程），调用 **display\_process\_info** 函数显示相关信息。这个函数根据设置的标志决定具体显示哪些信息。

|  |
| --- |
| int main(int argc, char \*argv[]) {  DIR \*dir;  struct dirent \*entry;  int show\_user\_info = 0; // Flag to determine whether to display user info  int show\_details = 0; // Flag to determine whether to display detailed info  int show\_all = 1;  // Parse command line options  int opt;  while ((opt = getopt(argc, argv, "lue")) != -1) {  switch (opt) {  case 'l':  show\_details = 1;  break;  case 'u':  show\_user\_info = 1;  break;  case 'e':  show\_all = 0;  break;  default:  exit(EXIT\_FAILURE);  }  }  // Open the /proc directory  dir = opendir("/proc");  if (dir == NULL) {  perror("opendir");  exit(EXIT\_FAILURE);  }  // Read the contents of the /proc directory  while ((entry = readdir(dir)) != NULL) {  // Check if the entry is a directory and its name is a number (pid)  if (entry->d\_type == DT\_DIR && atoi(entry->d\_name) > 0) {  display\_process\_info(atoi(entry->d\_name), show\_user\_info, show\_details, show\_all);  }  }  // Close the /proc directory  closedir(dir);  return 0;  } |

**3.3.3 cp指令**

该指令实现了对一个文件和文件夹复制，可以通过加入 -r 参数的方法来选择是复制文件还是文件夹。

代码首先定义了一些常量和函数，包括is\_dir函数用于判断是否是目录，copy\_file函数用于复制文件，copy\_dir函数用于复制文件夹，以及getFileName FromPath函数用于从路径中获取文件名。

在main函数中，首先检查命令行参数的数量，如果参数数量不正确，则输出错误信息并退出程序。

如果参数数量为3，表示需要复制单个文件。程序会以只读方式打开源文件和以写入、创建、截断方式打开目标文件，然后循环读取源文件的内容，并将读取的内容写入目标文件，直到读取到文件尾。

如果参数数量为4，并且此时加入的参数为 -r 则表示需要复制整个文件夹。程序首先判断源文件夹是否存在，如果不存在则输出错误信息并退出程序。然后创建目标文件夹，然后调用copy\_dir函数递归地复制文件夹中的所有文件和子文件夹。

其中一些重要的实现函数如下所示：

is\_dir函数为：

|  |
| --- |
| // 判断是否是目录，是返回1，否则返回0  int is\_dir(char \*path)  {      struct stat st;      stat(path, &st);      if (S\_ISDIR(st.st\_mode))          return 1;      else          return 0;  } |

copy\_file函数为：

|  |
| --- |
| // 复制文件，成功返回0，否则返回-1  int copy\_file(char \*src\_path, char \*dest\_path)  {      int src\_fd, dest\_fd; // 文件描述符      char buf[BUF\_SIZE];  // 缓冲区      int count;           // 读写字节数      // 打开源文件和目标文件      src\_fd = open(src\_path, O\_RDONLY);      if (src\_fd == -1)      {          perror("open src file error");          return -1;      }      dest\_fd = open(dest\_path, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);      if (dest\_fd == -1)      {          perror("open dest file error");          close(src\_fd);          return -1;      }      // 循环读写数据      while ((count = read(src\_fd, buf, BUF\_SIZE)) > 0)      {          if (write(dest\_fd, buf, count) != count)          {              perror("write error");              close(src\_fd);              close(dest\_fd);              return -1;          }      }      if (count == -1)      {          perror("read error");          close(src\_fd);          close(dest\_fd);          return -1;      }      // 关闭文件      close(src\_fd);      close(dest\_fd);      return 0;  } |

copy\_dir函数为：

|  |
| --- |
| // 复制文件夹，成功返回0，否则返回-1  void copy\_dir(char \*src, char \*dst) {      DIR \*dirp;      struct dirent \*dp;      struct stat st;      char src\_path[PATH\_MAX];      char dst\_path[PATH\_MAX];      dirp = opendir(src);      if (dirp == NULL) {          perror("opendir src");          exit(1);      }      // 创建目标目录，如果目录已存在则不创建      if (mkdir(dst, 0755) == -1 && errno != EEXIST) {          perror("mkdir dst");          exit(1);      }      while ((dp = readdir(dirp)) != NULL) {          if (strcmp(dp->d\_name, ".") == 0 || strcmp(dp->d\_name, "..") == 0) {              continue;          }          sprintf(src\_path, "%s/%s", src, dp->d\_name);          sprintf(dst\_path, "%s/%s", dst, dp->d\_name);          if (stat(src\_path, &st) == -1) {              perror("stat src");              exit(1);          }          if (S\_ISDIR(st.st\_mode)) {              copy\_dir(src\_path, dst\_path);          } else {              copy\_file(src\_path, dst\_path);          }      }      closedir(dirp);  } |

1. getFileName FromPath函数为：

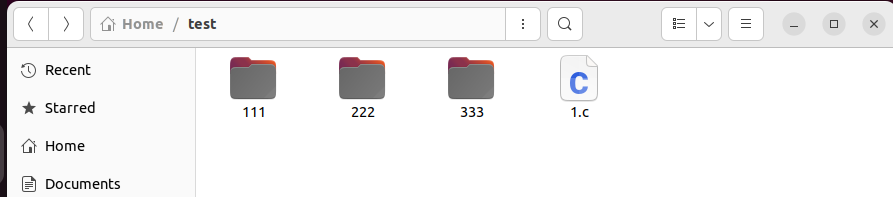
|  |
| --- |
| // 获取目录  char\* getFileNameFromPath(char\* path) {      int len = strlen(path);      if (len == 0) {          return NULL;  // 如果路径为空，则返回空指针      }      // 去除路径末尾的斜杠      if (path[len - 1] == '/') {          path[len - 1] = '\0';          len--;      }      // 查找最后一个斜杠的位置      char\* lastSlash = strrchr(path, '/');      if (lastSlash == NULL) {          return path;  // 如果没有斜杠，则整个路径就是文件名      }      else {          return lastSlash + 1;  // 返回斜杠后面的字符串作为文件名      }  } |

下面是具体的测试结果截图：

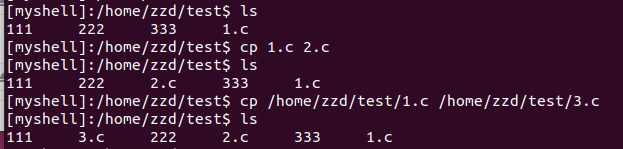
①cp <file> <destination + new\_name>

eg. cp 1.c 2.c 表示将当前目录下的1.c文件复制到当前目录，并且复制后的文件命名为2.c

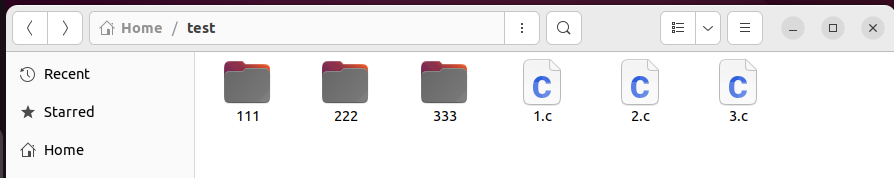
复制前目录为：



下面执行了两次cp命令，分别使用了相对路径和绝对路径，结果表示都成功复制。



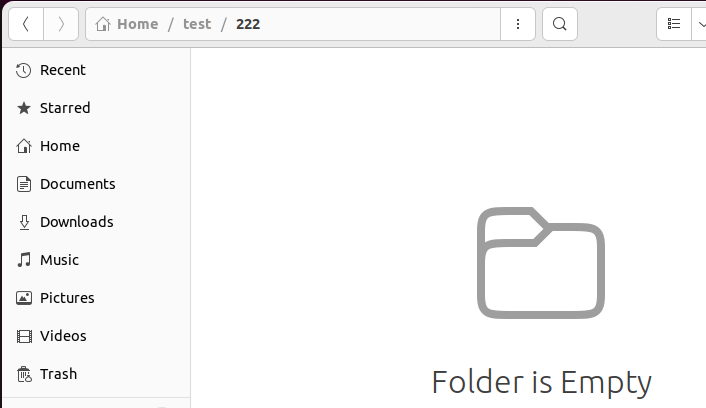
复制后目录为：



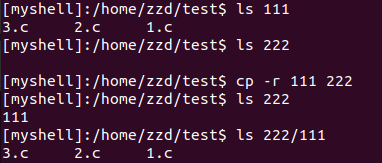
②cp -r <dir> <destination>

eg. cp -r 111 222 表示把当前目录下的111文件夹整个复制到当前目录下的222文件夹中。

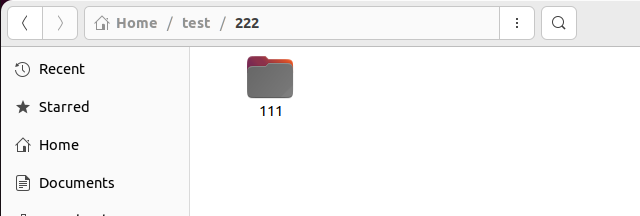
复制前222文件夹为：



使用相对路径进行复制，并且用于复制的111文件夹不是空的，通过ls指令检查后可以看见，将111文件夹复制到222文件夹后，原本111文件夹中的所有文件也被复制过去。下面是执行cp -r 指令，使用相对路径：

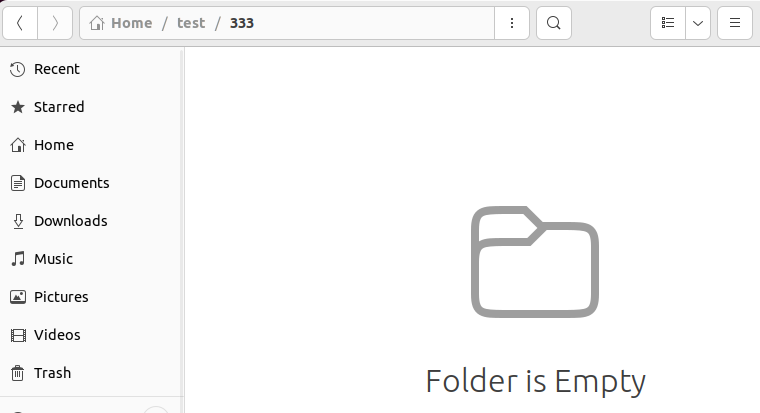


复制后222文件夹为：

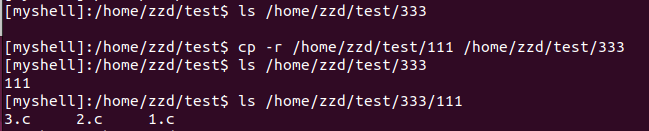


下面是使用绝对路径进行复制，将111文件夹复制到一个空的333文件夹中，通过结果表示是成功完成复制功能。

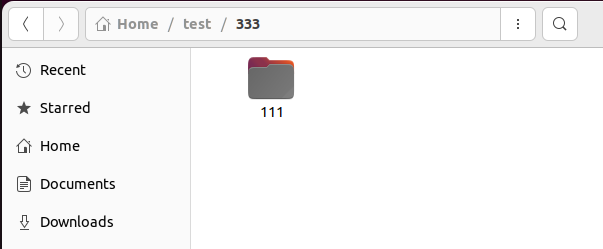
复制前333文件夹为：



执行cp -r 指令，使用绝对路径：



复制后333文件夹为：



**3.3.4 mv指令**

该指令实现了一个简单的文件及文件夹移动和重命名功能。为了更直观的体现出重命名功能，在对文件进行移动后会有一个提示信息，会让用户选择<y/n>，此时根据用户自己的选择来决定是否对文件进行重命名。

在main函数中，首先检查命令行参数的数量，如果参数数量不正确，则输出使用方法信息并退出程序。

然后，代码获取命令行参数中的源文件路径和目标路径，并使用basename函数获取源文件的名称。

接下来，代码根据用户的选择进行不同的操作：

①如果用户选择重命名文件，代码会要求用户输入新的文件名，并根据新的文件名构造新的目标路径。然后使用rename函数将源文件移动到新的目标路径，并输出移动和重命名成功的消息。

②如果用户选择不重命名文件，代码会直接使用原始的目标路径，即将源文件移动到目标路径，并输出移动成功的消息。

如果文件移动和重命名操作成功，程序返回0；否则，程序返回1，并输出相应的错误消息。

其中一些重要的实现函数如下所示：

mv.c整体代码为：

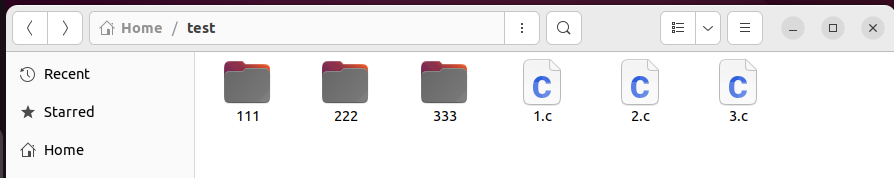
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <libgen.h>  #include <string.h>  int main(int argc, char \*argv[]) {      if (argc != 3) {          printf("Usage: %s <source> <destination>\n", argv[0]);          return 1;      }      char \*source = argv[1];      char \*destination = argv[2];      char \*source\_name = basename(source); // 获取源文件的名称      // 构造目标路径      char \*new\_destination = malloc(strlen(destination) + strlen(source\_name) + 2);      sprintf(new\_destination, "%s/%s", destination, source\_name);      char choice; // 存储用户选择      printf("Do you want to rename the file? (y/n): ");      scanf(" %c", &choice);      if (choice == 'y' || choice == 'Y') {          char new\_name[256];          printf("Enter the new name: ");          scanf("%255s", new\_name);          char \*new\_destination\_with\_name = malloc(strlen(destination) + strlen(new\_name) + 2);          sprintf(new\_destination\_with\_name, "%s/%s", destination, new\_name);          if (rename(source, new\_destination\_with\_name) == 0) {              printf("Moved and renamed successfully.\n");              return 0;          } else {              perror("Error moving and renaming file");              return 1;          }      } else {          if (rename(source, new\_destination) == 0) {              printf("Moved successfully.\n");              return 0;          } else {              perror("Error moving file");              return 1;          }      }  } |

下面是具体的测试结果截图：

①mv <file> <destination> (ask rename 'y/n')

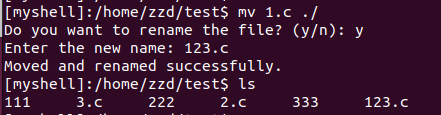
eg. mv 1.c ./ 表示把当前目录下的1.c文件移动到当前目录中，此时会有一个询问信息，如果选择重命名则需要用户输入一个新名字，如123.c，则会进行移动功能，并且进行重命名。

移动前目录为：

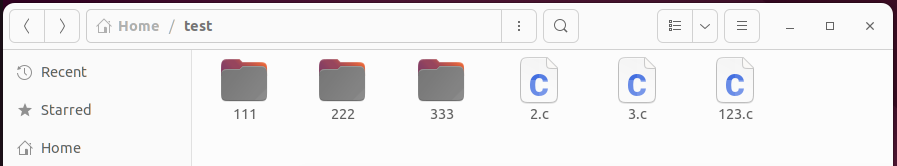


使用相对路径进行移动并且重命名，通过使用ls指令及可视化界面，可以看到成功地将1.c文件移动到当前目录中，并且重命名为123.c

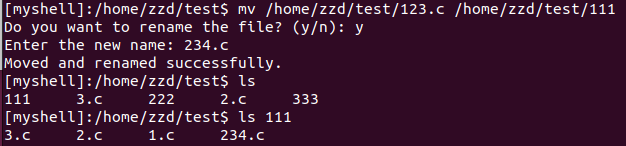
执行mv指令，使用相对路径：



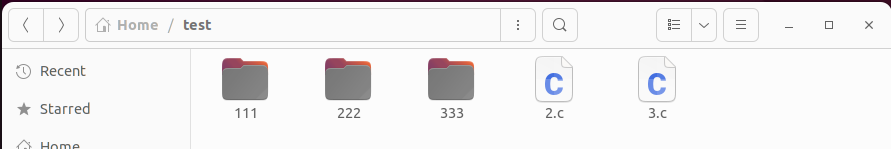
移动后目录为：



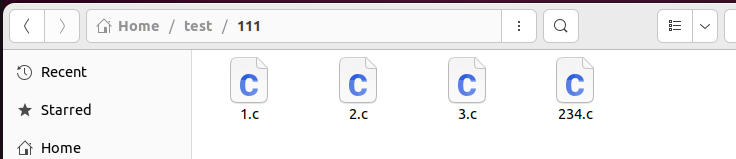
使用绝对路径进行移动并且重命名，通过使用ls指令及可视化界面，可以看到成功地将test目录中的123.c文件移动到111文件夹中，并且重命名为234.c。执行mv指令，使用绝对路径：



移动后test目录为：



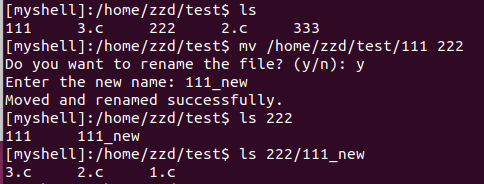
移动后111目录为：



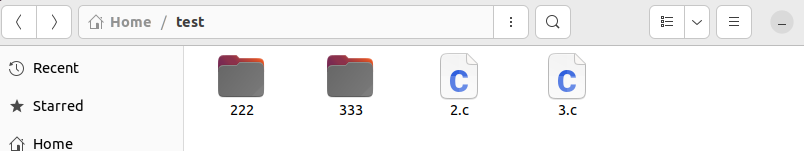
②mv <dir> <destination> (ask rename 'y/n')

eg. mv /home/zzd/test/111 222 表示把/home/zzd/test/111文件夹移动到当前目录中的222文件夹中，此时会有一个询问信息，如果选择重命名则需要用户输入一个新名字，如111\_new，则会进行移动功能，并且进行重命名。

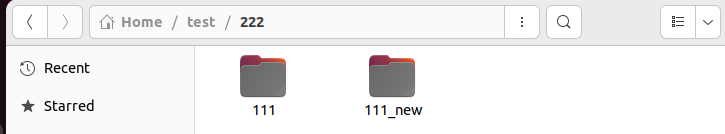
使用绝对路径和相对路径进行移动文件夹并且重命名，通过使用ls指令及可视化界面，可以看到成功地将test目录中的111文件夹移动到当前目录中的222文件夹中，并且重命名为111\_new，并且通过ls指令查看111\_new文件夹，发现成功将文件夹中的子文件也移动过去。执行mv指令移动文件夹，包含相对路径和绝对路径：



移动后test目录为：



移动后222目录为：



**3.3.5 ls指令**

该指令实现了一个简单的ls命令，可以列出指定目录下的文件和文件夹。

代码首先通过opendir函数打开指定目录（默认为当前目录），然后使用readdir函数读取目录下的文件和文件夹。

然后代码根据输入命令的参数的不同情况进行不同的处理：

ls：如果没有传递任何参数，则默认列出当前目录下的文件和文件夹，但会过滤掉以.开头的隐藏文件。

ls dir or ls -a ：如果传递了一个参数，则判断该参数是目录名还是 -a。如果是 -a ，则列出当前目录下的文件和文件夹，包括隐藏文件；如果是目录名，则列出该目录下的文件和文件夹，并且会过滤掉隐藏文件。

ls -a dir or ls dir -a ：如果传递了两个参数，则判断参数中是否有-a选项。如果有-a参数，则另一个参数就是目录名，列出目录下的所有文件和文件夹，包括隐藏文件。

最后，程序会输出列出的文件和文件夹名。

其中一些重要的实现函数如下所示：

ls.c整体代码为：

|  |
| --- |
| #include <stdio.h>  #include <dirent.h>  #include <string.h>  // 实现了ls、ls dir 、ls -a、 ls -a dir、 ls dir -a  int main(int argc, char\* argv[]) {      DIR \*d;      struct dirent \*dir;      const char \*path = ".";  // 默认目录是当前目录      int point;  // 用于判断是否含有 -a 参数      // 检查是否有参数传递给程序      if (argv[1] == NULL ) {    // ls 指令          d = opendir(path);          while ((dir = readdir(d)) != NULL) {              if (dir->d\_name[0] != '.') {  // 简单过滤掉隐藏文件                  printf("%s \t", dir->d\_name);              }          }      }      if(argv[1] != NULL )  //  ls dir or ls -a  or  ls -a dir or ls dir -a      {          if(argc == 3 ) { // ls -a dir or ls dir -a              if(strcmp(argv[1],"-a") == 0 || strcmp(argv[2],"-a") == 0){                  point = 1;                  if(strcmp(argv[1],"-a") == 0)                      path = argv[2];                  else                      path = argv[1];              }          }          if(argc == 2){              if(strcmp(argv[1],"-a") != 0){                  path = argv[1];                  point = 2;              }              else{                  path = ".";                  point = 1;              }          }          d = opendir(path);          if (d == NULL) {                  printf("无法打开目录: %s", path);                  return 1;              }          while ((dir = readdir(d)) != NULL){              if(point == 1){                  printf("%s \t", dir->d\_name);              }              else {                  if (dir->d\_name[0] != '.') {  // 简单过滤掉隐藏文件                      printf("%s \t", dir->d\_name);                  }              }          }      }      printf("\n");  //简单的换行，用于格式好看      closedir(d);      return 0;  } |

下面是具体的测试结果截图：

①ls

表示列出当前目录的文件和文件夹，但会过滤掉隐藏文件。执行ls指令：

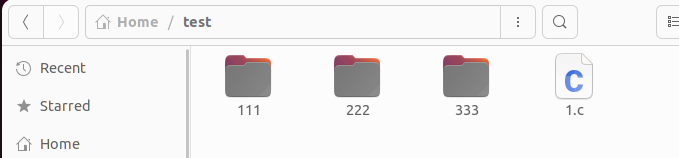


②ls -a

表示列出当前目录的文件和文件夹，包括隐藏文件。执行ls -a 指令：

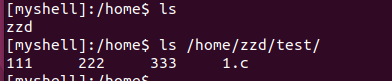


test目录可视化界面为：



③ls <dir>

eg. Ls /home/zzd/test/ 表示列出/home/zzd/test/目录的文件和文件夹，但会过滤掉隐藏文件。执行ls dir 指令：



④ls <dir> -a or ls -a <dir>

eg. Ls /home/zzd/test/ -a 表示列出/home/zzd/test/目录的文件和文件夹，包括隐藏文件。执行ls dir -a指令以及ls -a dir指令：





**3.3.6 rm指令**

该指令实现了一个简化版的rm命令，用于删除文件和目录。

会根据输入的命令行参数的不同情况进行不同的处理：

①如果argv[1]不以"-"开头，则表示此时命令为没有加option，故该输入的命令从argv[1]开始就是要删除的文件的路径，但是此时只能删除文件。然后再依次处理每个文件路径：首先检查文件是否存在，并确保它是一个普通文件而不是目录。然后，代码会询问用户是否要删除文件，如果用户确认，则使用remove函数删除文件。

②如果argv[1]以"-"开头，则根据不同的选项进行处理：

如果选项是"-f"或"-F"，则直接删除指定的文件。代码依次处理每个文件路径，确保它们是普通文件而不是目录，然后使用remove函数删除文件。

如果选项是"-r"或"-R"，则询问用户是否要删除指定的目录及其内容。代码依次处理每个目录路径，询问用户是否要删除目录，如果用户确认，则调用removeDirectory函数递归删除目录及其内容。removeDirectory函数会先递归删除目录下的所有文件，然后再删除目录本身。

如果选项是"-rf"，则直接删除指定的目录及其内容。代码依次处理每个目录路径，调用removeDirectory函数递归删除目录及其内容。

最终，无论是删除文件还是删除目录，如果操作成功，代码会输出相应的成功删除消息。如果出现错误，代码会使用perror函数输出错误消息。

其中一些重要的实现函数如下所示：

removeDirectory函数为：

|  |
| --- |
| void removeDirectory(const char \*dir\_path) {      DIR \*dir = opendir(dir\_path);      if (dir == NULL) {          perror("Error opening directory");          return;      }      struct dirent \*entry;      char path[PATH\_MAX];      while ((entry = readdir(dir)) != NULL) {          if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0) {              continue;          }          snprintf(path, sizeof(path), "%s/%s", dir\_path, entry->d\_name);          if (entry->d\_type == DT\_DIR) {              removeDirectory(path); // 递归删除子目录          } else {              if (unlink(path) != 0) {                  perror("Error deleting file");              }          }      }      closedir(dir);      if (rmdir(dir\_path) != 0) {          perror("Error deleting directory");  }  } |

1. rm -f 参数的具体实现部分为：

|  |
| --- |
| // 参数为：-f or -F （直接删除文件）          if(strcmp(argv[1], "-f") == 0 || strcmp(argv[1], "-F") == 0){              // rm -f <file1> <file2> <file3>              for (int i = 2; i < argc; i++) {                  char\* file\_path = argv[i];                  struct stat st;                  if (stat(file\_path, &st) != 0) {                      perror("Error");                      return 1;                  }                  if (!S\_ISREG(st.st\_mode)) { // 确保无法删除目录                      printf("rm: cannot remove '%s': Is a directory\n", file\_path);                      continue;                  }                  // 使用remove函数删除文件                  if (remove(file\_path) == 0) {                      printf("File '%s' successfully deleted.\n", file\_path);                  }                  else                      perror("Error");              } |

rm -r参数的具体实现部分为：

|  |
| --- |
| // 参数为：-r or -R （询问确认）          if(strcmp(argv[1], "-r") == 0 || strcmp(argv[1], "-R") == 0){              for (int i = 2; i < argc; i++) {                  char\* dir\_path = argv[i];                  printf("rm: remove directory '%s' ? (y/n): ", dir\_path);                  char choice;                  scanf(" %c", &choice);  // 注意空格，以避免读取之前的换行符                  if (choice == 'y' || choice == 'Y') {                      removeDirectory(dir\_path);                      printf("directory '%s' successfully deleted.\n", dir\_path);                  }              }          } |

rm -rf 参数的具体实现部分为：

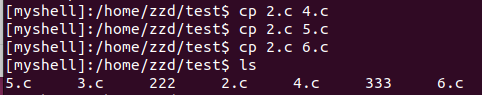
|  |
| --- |
| // 参数为: -rf  （直接删除目录）          if(strcmp(argv[1], "-rf") == 0 ){              for (int i = 2; i < argc; i++) {                  char\* dir\_path = argv[i];                  removeDirectory(dir\_path);                  printf("directory '%s' successfully deleted.\n", dir\_path);              }          } |

下面是具体的测试结果截图：

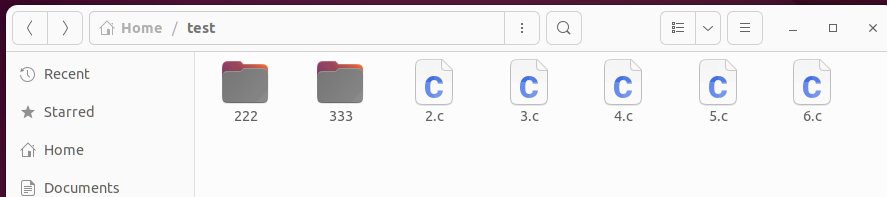
①rm <file1> <file2> ...

eg. rm 2.c 3.c /home/zzd/test/4.c 表示将三个文件删除，这里使用了相对路径和绝对路径，用于检测路径的正确性。使用该命令时会对每一条删除指令进行确认，当用户确定要删除文件时，才会执行删除指令。

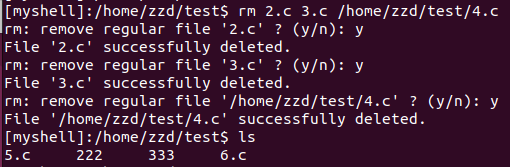
首先使用cp指令复制一些文件用于后续的测试。



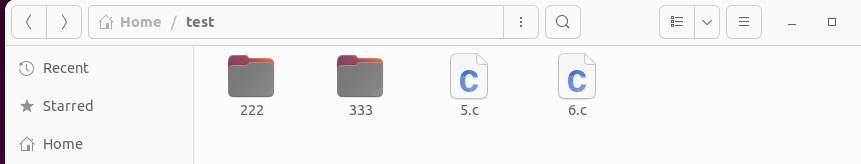
删除前test目录为：



通过使用ls指令检查，该删除指令可以成功同时完成对多个文件的删除，相对路径和绝对路径都可以成功。执行rm指令，包含相对路径和绝对路径：

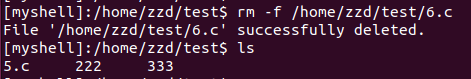


删除后test目录为：

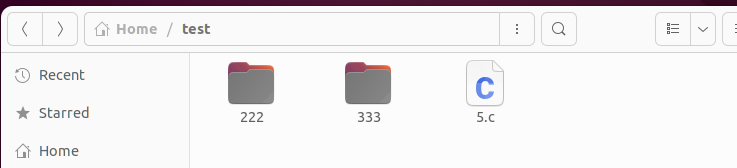


②rm -f <file1> <file2> ...

eg. rm -f /home/zzd/test/6.c 表示直接将该目录下的6.c文件删除掉，不会给出询问信息。执行rm -f指令：



删除后test目录为：

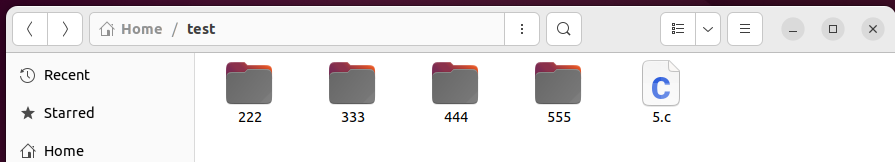


③rm -r <dir1> <dir2> ...

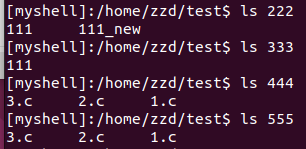
eg. rm -r 333 /home/zzd/test/444 表示将两个文件夹删除，这里使用了相对路径和绝对路径，用于检测路径的正确性。使用该命令时会对每一条删除指令进行确认，当用户确定要删除文件夹时，才会执行删除指令。

首先在test目录中创建444、555两个文件夹，并且在这两个文件夹中都放入三个文件，以保证这两个文件夹不是空的，用于后续的删除测试。

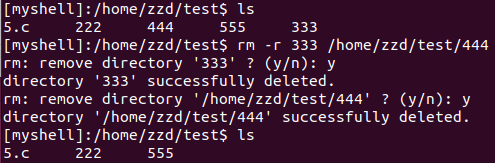
删除前test目录为：



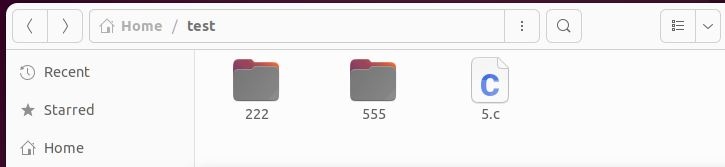
首先检查文件夹都不是空的：



使用该指令进行测试，并且通过ls指令检查，该删除指令可以成功同时完成对多个文件夹的删除，相对路径和绝对路径都可以成功。执行rm -r指令，包含相对路径和绝对路径：

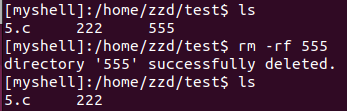


删除后test目录为：

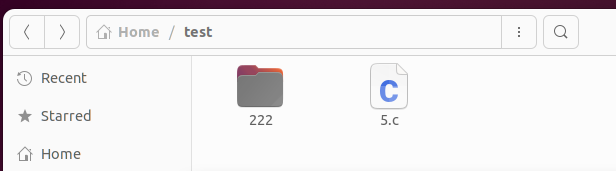


④rm -rf <dir1> <dir2> ...

eg. rm -rf 555 表示直接将该目录下的555文件夹删除掉，不会给出询问信息。执行rm -rf指令：



删除后test目录为：



四、 项目总结

本次实验旨在使用C语言创建自定义Linux shell，以加深对Linux系统底层系统调用和解释器工作原理的理解。项目包括实现shell解释器的基本和高级功能，如命令帮助、文件目录显示、工作目录切换、文件和文件夹复制、移动、删除、命令历史显示、进程信息显示、目录结构可视化、输入输出重定向、管道以及后台程序运行等。

团队成功实现了这些功能，采用了结构化的方法，重点关注内部命令函数、重定向、管道和后台进程，所有这些都整合到shell的主代码文件中。shell的整体框架涉及处理命令输入并调用适当的函数执行。通过对每个实现的功能进行测试，确保了程序的完整性和正确性。