UNIX_myshell实验报告

一、myshell功能概述

myshell 是使用c语言编写的可以在Linux系统下运行的shell程序，该程序能够在Ubuntu中运行。其主要功能如下：

1. 可以运行**带参数和不带参数**的命令
2. 每一行可支持指令总长度不超过**256**个
3. myshell支持标准I/O重定向，可以通过管道连接**多个**命令（管道符号和重定向符号与指令之间可以不用空格连接）
4. myshell支持指令**后台运行**
5. myshell支持使用`cd`命令切换工作路径
6. myshell支持使用`history`命令查看历史指令
7. 项目文件夹里有Makefile文件，通过在目录下执行make指令可以生成myshell.o可执行文件，运行该文件即可启动myshell程序，执行make clean指令即可删除该可执行程序
8. myshell支持使用**exit**或**logout**指令退出

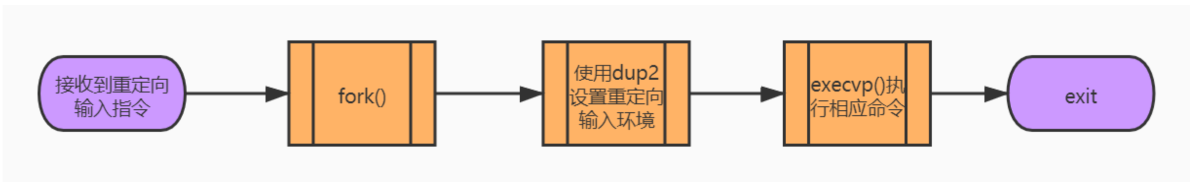
二、功能实现概述

1. 功能实现所需的系统调用

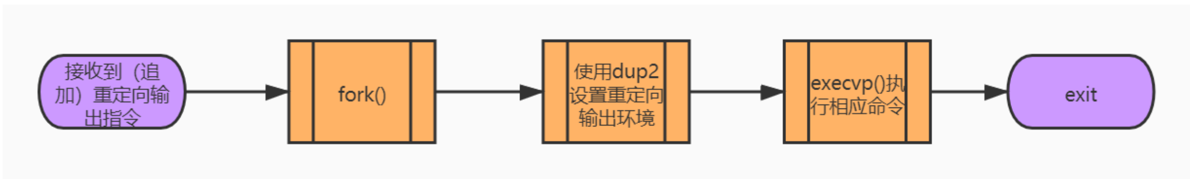
实现功能	实现函数	系统调用
cd命令（切换工作目录）	int dealCd(int argc)	getcwd(), chdir()
history命令（查看过去输入指令）	int getHistory()	/
输入重定向（<）	void deal_with_command(int argcount, ...)	fork(), dup2(), open(), execvp(), exit()，close()
（追加）输出重定向(>/>>)	void deal_with_command(int argcount, ...)	fork(), dup2(), open(), execvp(),exit(), close()
管道(|)	void split_command(int argct, char argl`[100][BUFSIZE]`)	fork(), dup2(), open(),write(), exit(), close()
指令后台运行（&）	void split_command(int argct, char argl`[100][BUFSIZE]`)	fork(), waitpid(), exit()
退出shell	int main()	exit()

2. 功能实现流程图

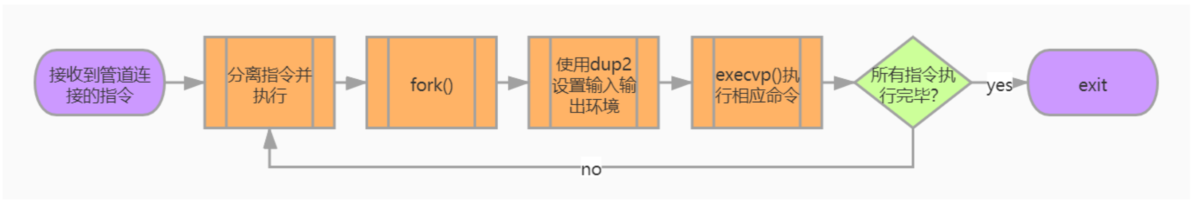
1. 输入重定向实现



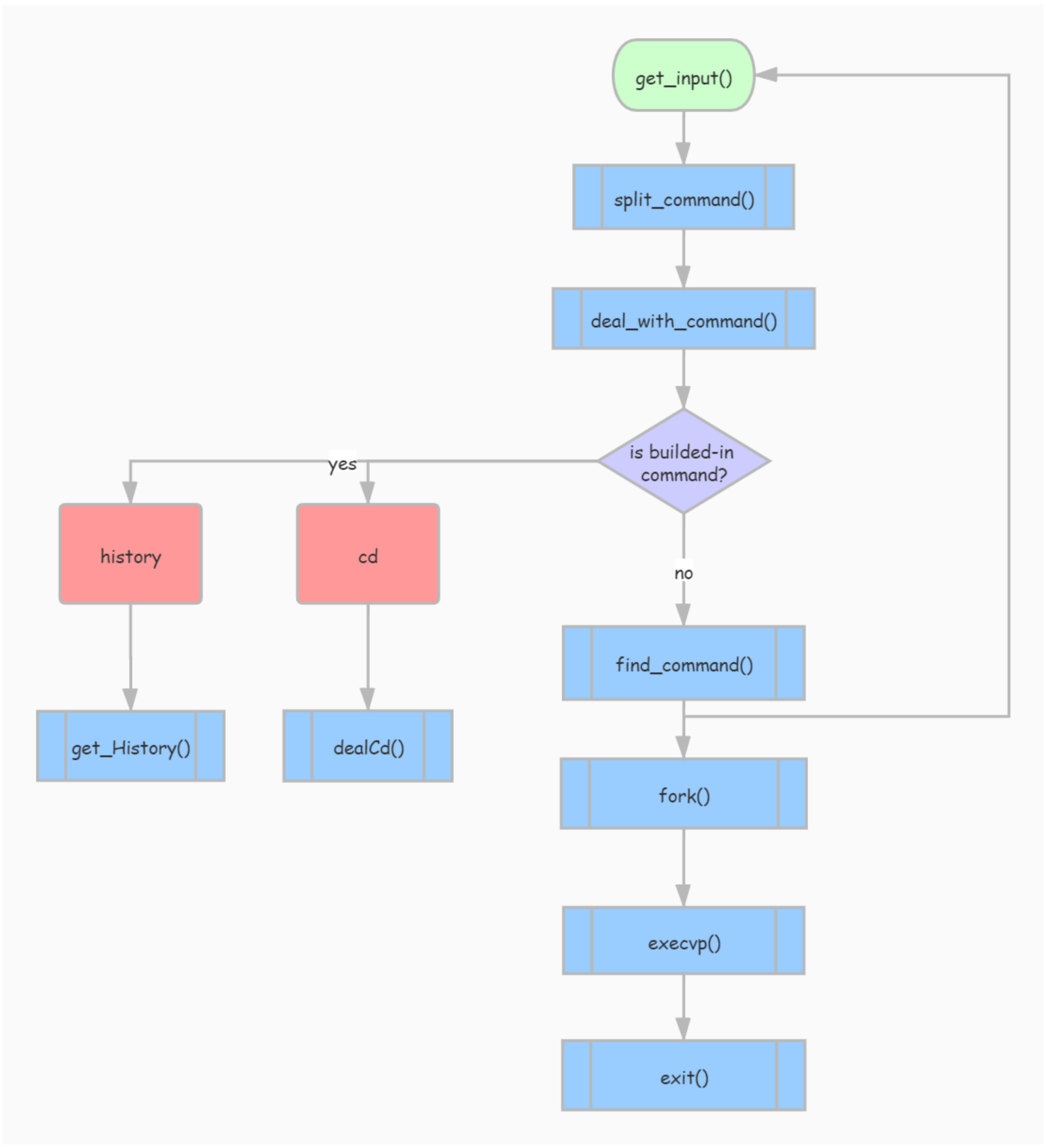
2. 输出重定向实现



3. 管道实现



3. 指令的执行



4. 功能的实现

1. cd命令实现

   使用chdir系统调用切换当前进程工作目录，并使用getcwd系统调用获取当前工作目录（**一定要在父进程中而不能在子进程中！**）

2. 管道和重定向的实现

   在实现重定向时，使用open系统调用打开重定向的目标文件，并使用dup2系统调用将其与标准输入或标准输出等相连接，执行指令。

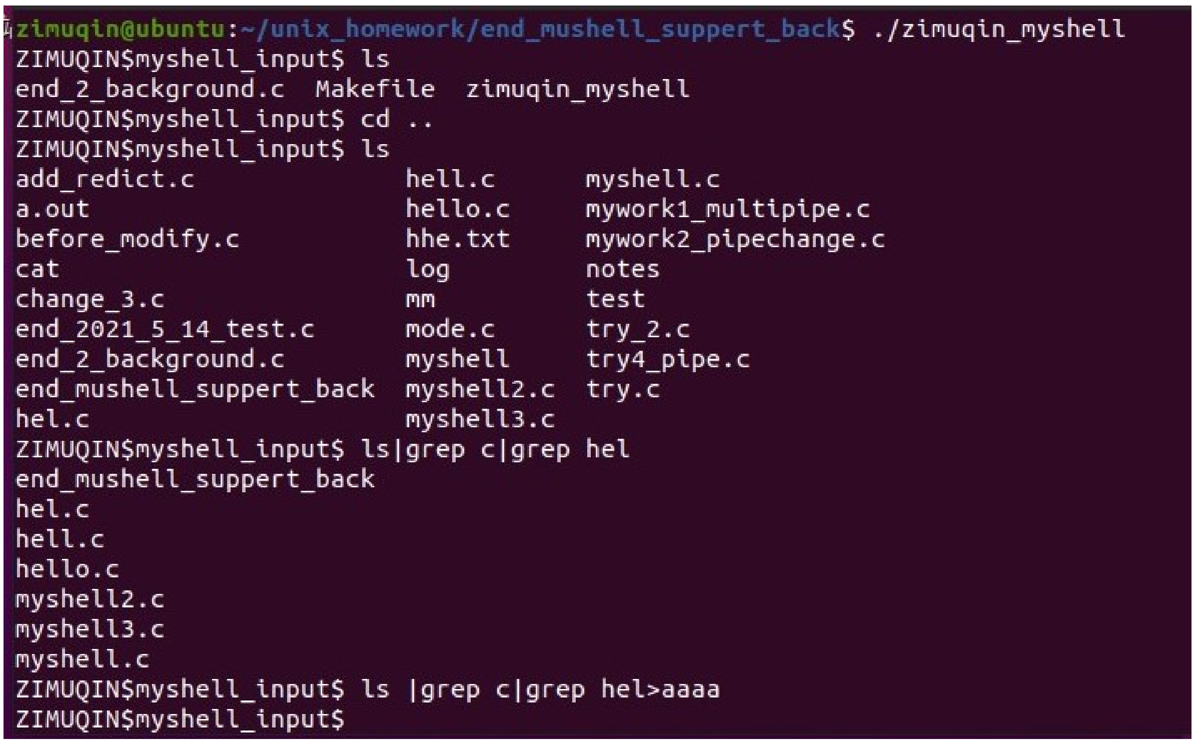
   在实现管道时，分别在tmp文件夹下使用open打开youdontknowfile和youdontknowfiile1两个文件，使用dup2系统调用将其分别与标准输入和标准输出相连接，将其作为输入的来源或输出的目的，当多个管道连接时，分别从这两个文件进行输入输出。

3. 指令后台运行的实现

在执行split_command()函数时会重新开启一个子进程，在未获取到'&'时，父进程会等待子进程正确结束后返回，shell重新运行，用户需要等到指令执行结束后才能够重新输入指令；如果在输入指令中含有'&'时，父进程不会等待子进程运行结束，而会直接返回，此时用户可以在上一条指令未执行结束时输入指令。

三、shell运行示例

如图，该shell能够在Ubuntu环境下运行，部分指令结果如下：



```
zimuqin@ubuntu:~/unix_homework/end_mushell_suppert_back$ ./zimuqin_myshell
ZIMUQIN$myshell_input$ ls
end_2_background.c  Makefile  zimuqin_myshell
ZIMUQIN$myshell_input$ cd ..
ZIMUQIN$myshell_input$ ls
add_redict.c              hell.c      myshell.c
a.out                     hello.c     mywork1_multipipe.c
before_modify.c           hhe.txt     mywork2_pipechange.c
cat                       log         notes
change_3.c                mm          test
end_2021_5_14_test.c      mode.c      try_2.c
end_2_background.c        myshell     try4_pipe.c
end_mushell_suppert_back  myshell2.c  try.c
hel.c                     myshell3.c
ZIMUQIN$myshell_input$ ls|grep c|grep hel
end_mushell_suppert_back
hel.c
hell.c
hello.c
myshell2.c
myshell3.c
myshell.c
ZIMUQIN$myshell_input$ ls |grep c|grep hel>aaaa
ZIMUQIN$myshell_input$
```

四、源代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <dirent.h>

#define normal 0  //其他命令
#define out_redirect 1  //输出重定向
#define in_redirect 2  //输入重定向
#define have_pipe 3  //含有管道的命令
#define out_redict 4  //不覆盖的输出重定向
#define BUFSIZE 256

int argc;
char argv[100][BUFSIZE];
char **arg = NULL;
char buf[BUFSIZE];
char buff[BUFSIZE];
char *current;
char history[500][BUFSIZE];
int commandNum;

int get_input(char buf[]);
void explain_input(char buf[], int *argc, char argv[100][BUFSIZE]);
void deal_with_command(int argcount, char arglist[100][BUFSIZE], int isstart,
int isend);
int find_command (char *command);
int dealCd(int argc);
int getHistory();
void split_command(int argct, char argl[100][BUFSIZE]);

int main()
{
    int i;
    commandNum = 0;
    while(1) {
        argc = 0;
        memset(buf, 0, BUFSIZE);
        printf("ZIMUQIN$myshell_input$ ");
        get_input(buf);

        //exit the shell
        if(strcmp(buf,"exit\n") == 0 || strcmp(buf,"logout\n") == 0)
            break;
        for (i=0; i < 100; i++)
        {
            argv[i][0]='\0';
        }
        argc = 0;
        explain_input(buf, &argc, argv);
        split_command(argc, argv);
    }
    exit(0);
}

//接受并处理输入的函数qaq（保证重定向和管道前后都有空格）

int get_input(char buf[]) {
    memset(buff, 0, BUFSIZE);
    fgets(buff, BUFSIZE, stdin);
    strcpy(history[commandNum++],buff);
    int i = 0;
    int j = 0;
    while (buff[i] != '\n') {
        //处理输入输出重定向时的问题
        if (buff[i] != '<' && buff[i] != '>' && buff[i] != '|') {
            buf[j] = buff[i];
            j = j + 1;
        }
        else {
            if (buff[i-1] == '<' || buff[i-1] == '>' || buff[i-1] == ' ') {
                buf[j] = buff[i];
                j = j + 1;
            }
            else {
                buf[j] = ' ';
                j = j + 1;
                buf[j] = buff[i];
                j = j + 1;
            }
            if (buff[i+1] == '<' || buff[i+1] == '>' || buff[i+1] == ' ') {

            }
            else {
                buf[j] = ' ';
                j = j + 1;
            }
        }
        i ++;
    }
    buf[j] = '\n';
    j = j + 1;
    buf[j] = '\0';
    return j;
}

void explain_input(char buf[], int *argc, char argv[100][BUFSIZE])
{
    char *p = buf;
    char *q = buf;
    int number = 0;

    while (1) {
        if ( p[0] == '\n' )
            break;

        if ( p[0] == ' '  )
            p++;
        else {
            q = p;
            number = 0;
            while( (q[0]!=' ') && (q[0]!='\n') ) {
                number++;
                q++;
            }
            strncpy(argv[*argc], p, number+1);
            argv[*argc][number] = '\0';
            *argc = *argc + 1;
            p = q;
        }
    }
}

void split_command(int argct, char argl[100][BUFSIZE]) {

    int i = 0;
    int now_cmd = 0;
    int background = 0;
    int status;
    char argnext[100][BUFSIZE];
    int argcnext;
    int isstart = 0;
    pid_t pid44;
```

```c
    if (strcmp(argv[0],"cd") == 0) {
        int res = dealCd(argc);
        if (!res) {
            printf("wrong input!\n");
         }
        return;
    }

    for (i = 0; i < argct; i++) {
        if (strcmp(argl[i], "&") == 0 && i == argct - 1) {
            background = 1;
            argct = argct - 1;
        }
        else if (strcmp(argl[i], "&") == 0){
            printf("wrong command!\n");
            return;
        }
    }

    pid44 = fork();
    if (pid44 == 0) {
        for (i = 0; i < argct; i++) {

            if (strcmp(argl[i], "|") == 0) {
                int j;
                argcnext = 0;
                int flag = 0;
                for (j = now_cmd; j < i; j++) {
                    if (strcmp(argl[i], "<") == 0) {
                        flag = 1;
                    }
                    strcpy(argnext[argcnext],argl[j]);
                    argcnext++;
                }
                if (now_cmd == 0) {
                    isstart = 1;
                }
                else {
                    isstart = 0;
                }
                now_cmd = i + 1;
                deal_with_command(argcnext, argnext, isstart, 0);
                //if (flag == 0) {
                int fd2, fd3;
                fd3 = open("/tmp/youdonotknowfile1",O_RDONLY);
                fd2 = open("/tmp/youdonotknowfile",
                        O_WRONLY|O_CREAT|O_TRUNC,0644);
                char buffer[1024] = {0};
                    char *ptr;
                int count;
                while (count = read(fd3, buffer, 1024))
                {
                    ptr = buffer;
                    write(fd2, ptr, count);
                    memset(buffer, 0, 1024);
                }
                //}
            }
        }

        int j;
        argcnext = 0;
        for (j = now_cmd; j < argct; j++) {
            strcpy(argnext[argcnext],argl[j]);
            argcnext++;
        }
        if (now_cmd == 0) {
            isstart = 1;
        }
        else {
            isstart = 0;
        }
        deal_with_command(argcnext, argnext, isstart, 1);

        exit(0);
    }
    if ( background == 1 ) {
        printf("[process id %d]\n", pid44);
        return ;
    }

    /* 父进程等待子进程结束 */
    if (waitpid (pid44, &status,0) == -1)
        printf("wait for child process error\n");
}


void deal_with_command(int argcount, char arglist[100][BUFSIZE], int isstart,
int isend)
{
    int flag = 0;
    int how = 0;         //运行方式, (重定向或者没有)
    int background = 0; //命令是否需要在后台运行
    int status;
    int i;
    int fd;
    int fd2;
    int fd3;
    char* argpp[argcount+1];
    char* argnext[argcount+1];
    char* file;
    pid_t pid;

    //get the command
    for (i=0; i < argcount; i++) {
        argpp[i] = (char *) arglist[i];
    }
    argpp[argcount] = NULL; //最后一个是空

    //whether background
    for (i=0; i < argcount; i++) {
        if (strncmp(argpp[i], "&",1) == 0) {
            if (i == argcount-1) {
                background = 1;
                argpp[argcount-1] = NULL;
                break;
            }
            else {
                printf("wrong command!\n");
                return ;
            }
        }
    }

    for (i=0; argpp[i]!=NULL; i++) {
        if (strcmp(argpp[i], ">") == 0 ) {
            flag++;
            how = out_redirect;
            if (argpp[i+1] == NULL)
                flag++;
        }
        if (strcmp(argpp[i],"<") == 0 ) {
            flag++;
            how = in_redirect;
            if(i == 0)
                flag++;
        }
        if (strcmp(argpp[i],">>") == 0 ) {
            flag++;
            how = out_redict;
            if(i == 0)
                flag++;
        }
    }


    if (flag > 1) {
        printf("wrong command!\n");
        return;
    }

    if (how == out_redirect || how == out_redict) {  //命令含有重定向输出
        for (i=0; argpp[i] != NULL; i++) {
            if (strcmp(argpp[i],">")==0 || strcmp(argpp[i], ">>") == 0) {
                file   = argpp[i+1];
                argpp[i] = NULL;
            }
        }
    }

    if (how == in_redirect) {    //含有输入重定向
        for (i=0; argpp[i] != NULL; i++) {
            if (strcmp (argpp[i],"<") == 0) {
                file   = argpp[i+1];
                argpp[i] = NULL;
            }
        }
    }
    if ( (pid = fork()) < 0 ) {
        printf("error! fork failed!\n");
        return;
    }
    switch(how) {
        case 0:
            //让子进程执行
            if (pid == 0) {
                if (!(find_command(argpp[0])) ) {
                    printf("%s : command not found1\n", argpp[0]);
                    exit (0);
                }
                fd2 = open("/tmp/youdonotknowfile",O_RDONLY);
                if (isstart ==  0) {
                    dup2(fd2,0);
                }
                fd3 = open("/tmp/youdonotknowfile1",
                        O_WRONLY|O_CREAT|O_TRUNC,0644);
                if (isend == 0) {
                    dup2(fd3, 1);
                }
                execvp(argpp[0], argpp);
                close(fd2);
                close(fd3);
                exit(0);
            }
            break;
        case 1:
            //输出重定向
            if (pid == 0) {
                if ( !(find_command(argpp[0])) ) {
                    printf("%s : command not found\n",argpp[0]);
                    exit(0);
                }
                fd2 = open("/tmp/youdonotknowfile",O_RDONLY);
                if (isstart ==  0) {
                    dup2(fd2,0);
                }
                fd = open(file,O_RDWR|O_CREAT|O_TRUNC,0644);
                dup2(fd,1);
                execvp(argpp[0],argpp);
                exit(0);
            }
            break;
        case 2:
            //输入重定向
            if (pid == 0) {
                if ( !(find_command (argpp[0])) ) {
                    printf("%s : command not found\n",argpp[0]);
                    exit(0);
                }
                fd2 = open("/tmp/youdonotknowfile1",
                        O_WRONLY|O_CREAT|O_TRUNC,0644);
                if (isend == 0) {
                    dup2(fd2, 1);
                }
                fd = open(file,O_RDONLY);
                dup2(fd,0);
                execvp(argpp[0],argpp);
                exit(0);
            }
            break;
        case 4:
            //追加输出重定向
            if (pid == 0) {
                if ( !(find_command(argpp[0])) ) {
                    printf("%s : command not found\n",argpp[0]);
                    exit(0);
                }
                fd2 = open("/tmp/youdonotknowfile",O_RDONLY);
                if (isstart ==  0) {
                    dup2(fd2,0);
                }
                fd = open(file, O_WRONLY|O_APPEND|O_CREAT|O_APPEND, 7777);
                dup2(fd,1);
                execvp(argpp[0],argpp);
                exit(0);
            }
            break;
        default:
            break;
    }

    //等待子进程结束后返回
    if (waitpid (pid, &status,0) == -1)
        printf("wait for child process error\n");
}

//找到执行命令的程序
int find_command (char *command)
{
    DIR*             dp;
    struct dirent*   dirp;
    char*            path[] = { "./", "/bin", "/usr/bin", NULL};

    if (strcmp(argv[0], "history") == 0) {
        getHistory();
        return 1;
    }

    /* 使当前目录下的程序可以被运行，如命令"./fork"可以被正确解释和执行 */
    if( strncmp(command,"./",2) == 0 )
        command = command + 2;

    /* 分别在当前目录、/bin和/usr/bin目录查找要可执行程序 */
    int i = 0;
    while (path[i] != NULL) {
        if ( (dp = opendir(path[i]) ) == NULL)
            printf ("can not open /bin \n");
        while ( (dirp = readdir(dp)) != NULL) {
            if (strcmp(dirp->d_name,command) == 0) {
                closedir(dp);
                return 1;
            }
        }
        closedir (dp);
        i++;
    }
    return 0;
}
```

```c
int dealCd(int argc) {
    int result = 1;
    if (argc != 2) {
        printf("the command is wrong:please input 'cd dir'\n");
    }
    else {
        int ret = chdir(argv[1]);
        if (ret != 0) {
            return 0;
        }
    }
    if (result) {
        char* res = getcwd(current, BUFSIZE);
        if (res == NULL) {
            printf("wrong path! please enter a existed directory\n");
        }
        return result;
    }
    return 0;
}

int getHistory() {
    int n;
    int i;
    if (argc == 2) {
        n = atoi(argv[1]);
    }
    else if (argc == 1) {
        n = 10;
    }
    else {
        printf("wrong input: please enter history [commandsNum]\n");
        return 0;
    }

    for (i = 1; i <= n && commandNum - i >= 0; i++) {
        printf("%d\t%s\n", commandNum - i, history[commandNum - i]);
    }
    return 0;
}
```