

《操作系统综合实验》

实验报告

02 多进程编程

姓名 王栗政
班级 2018039
学号 20179100018
教师 冯鹏斌

实验日期 2022.4.9

一、实验题目

Project 1—UNIX Shell and History Feature

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. This project can be completed on any Linux, UNIX, or Mac OS X system. A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c`. (This command displays the file `prog.c` on the terminal using the UNIX `cat` command.)

```
osh> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.10. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (`&`) at the end of the command. Thus, if we rewrite the above command as

```
osh> cat prog.c
```

the parent and child processes will run concurrently.

The separate child process is created using the `fork()` system call, and the user's command is executed using one of the system calls in the `exec()` family (as described in Section 3.3.1).

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. The `main()` function presents the prompt `osh->` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should run` equals 1; when the user enters `exit` at the prompt, your program will set `should run` to 0 and terminate.

This project is organized into two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to allow a history feature.

二、相关原理与知识

1.C语言控制终端字符颜色

`\033[`字符的显示方式;字符的颜色;字符的背景颜色`m` 需要显示的字符 `\033[m`

其中字符的显示方式、字符的颜色和字符的背景颜色都用数字来表示，不同的数字代表不同的意思，各个数字代表的含义如下：

字符的显示方式:0 (默认值)、1 (加粗)、2 (灰显)、3 (斜体)、4 (下划线)、5 (慢闪烁)、6 (快闪烁)、7 (反显)、8 (隐藏)、9 (删除线)

字符的颜色:30 (黑色)、31 (红色)、32 (绿色)、33 (黄色)、34 (深蓝色)、35 (紫色)、36 (浅蓝色)、37 (白色)

字符的背景颜色:40 (黑色)、41 (红色)、42 (绿色)、43 (黄色)、44 (蓝色)、45 (紫色)、46 (浅蓝色)、47 (白色)

- 每个数字后面都有一个括号，括号中的内容表示该数值对字符的控制形式。比如字符的颜色这一项中，31 (红色) 表示使用红色来显示字符，我们会在终端中看到printf输出红色的字符。
- 在字符的显示方式中没有数值5、6和8。因为这几个数值控制的显示方式和默认的显示方式十分类似，所以我没有单独来说明它们代表的显示方式。
- 字符的颜色及其背景颜色都是从数值x0到x7，只不过这个x的数值一个是3，另外一个为4。数值x7以后的数值没有特别的意义。

示例:



2. 获取提示符的相关函数

```
#include<unistd.h>
#include<sys/types.h>
#include<pwd.h>

uid_t getuid(void) //用来取得执行目前进程的用户识别码。
struct passwd *getpwuid(uid_t uid); //通过用户的uid查找用户的passwd数据
//The passwd structure is defined in <pwd.h> as follows:
struct passwd {
    char *pw_name; /*user name */
    char *pw_passwd; /*user password */
    uid_t pw_uid; /*user id */
    gid_t pw_gid; /*group id */
    char *pw_gecos; /*user real name */
    char *pw_dir; /*home directory */
    char *pw_shell; /*shell program */
};
int gethostname(char *name, size_t len); //获取本地主机的标准主机名
char *getcwd(char *buf, size_t size); //将当前的工作目录绝对路径复制到参数buf 所指的内存空间, 参数size 为buf 的空间大小。
```

3. exec函数族

/*exec函数族共有6种不同形式的函数。这6个函数可以划分为两组：

(1)execl、execle和execlp。

(2)execv、execve和execvp。

这两组函数的不同在于exec后的第一个字符，第一组是l，在此称，为execl系列；第二组是v，在此称为execv系列。这里的l是list(列表)的意思，表示execl系列函数需要将每个命令行参数作为函数的参数进行传递；而v是vector(矢量)的意思，表示execv系列函数将所有函数包装到一个矢量数组中传递即可。*/

```
int execl(const char * path, const char * arg, ...);
int execle(const char * path, const char * arg, char * const envp[]);
int execlp(const char * file, const char * arg, ...);
int execv(const char * path, char * const argv[]);
int execve(const char * path, char * const argv[], char * const envp[]);
int execvp(const char * file, char * const argv[]);
/*参数说明：
```

path：要执行的程序路径。可以是绝对路径或者是相对路径。在execv、execve、execl和execle这4个函数中，使用带路径名的文件名作为参数。

file：要执行的程序名称。如果该参数中包含“/”字符，则视为路径名直接执行；否则视为单独的文件名，系统将根据PATH环境变量指定的路径顺序搜索指定的文件。

argv：命令行参数的矢量数组。

envp：带有该参数的exec函数可以在调用时指定一个环境变量数组。其他不带该参数的exec函数则使用调用进程的环境变量。

arg：程序的第0个参数，即程序名自身。相当于argv[0]。

...: 命令行参数列表。调用相应程序时有多少命令行参数，就需要有多少个输入参数项。注意：在使用此类函数时，在所有命令行参数的最后应该增加一个空的参数项(NULL)，表明命令行参数结束。

返回值：-1表明调用exec失败，无返回表明调用成功。 [1] */

4.一些最长限制

命令行的长度限制 2097152 //可以用getconf ARG_MAX 获取

5.strtok函数用法

```
char *strtok(char s[], const char *delim);
```

//分解字符串为一组字符串。s为要分解的字符串，delim为分隔符字符串（如果传入字符串，则传入的字符串中每个字符均为分隔符）。首次调用时，s指向要分解的字符串，之后再次调用要把s设成NULL。

//这个函数的原理是将找到的delim分隔字符换成'\0'

//需要注意的是s需要是可改变的量，这就涉及到C内存分配的知识了，在网上看到有人说只能使用字符数组，那是不正确的，如果是字符串常量会存在常量区，无法进行修改，所以字符串常量不能作为第一个参数，但是可以使用数组或者指向堆栈区的指针，例如提前声明的指针，malloc分配空间，之后用scanf填入，此时字符串存入的还是堆区，可修改，但是如果分配空间后使用它指向常量区，那仍然不可修改

//例如：

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
    char *char_changeable;
    char *char_unchangeable = "I'm unchangeable";

    char_changeable = (char *) malloc(sizeof(char) * 100);
    // char_unchangeable[5] = '2';
    scanf("%s",char_changeable);
    char_changeable[5] = '2';
    // printf("%s",char_unchangeable);
    printf("%s",char_changeable);
}
```

//本题中我使用的是用malloc分配的空间存储的，虽然不是字符数组，但仍属于可以修改的量

三、实验过程

实验思路：启动程序后首先和shell一样给出带有颜色的提示信息（对于不同用户和root用户进行区分），之后对于cd、history、exit函数进行重写，通过exec族函数实现剩余的功能。每次读取用户的命令首先进行解析，将其通过空格进行分隔。每次使用完要对使用过的空间进行清理。

1.提示信息

为了实现和真正的shell一模一样的提示信息，我们使用了上面涉及到的获取系统信息的函数，并对普通用户和root用户提示信息颜色进行了区分。

```
uid_t uid;
int user_path_len;
char host_name[256]; //linux主机名最长255个字符
char current_user_path[256]; //用于获取当前目录的user_path
char current_path[512];
struct passwd * user_info = NULL;

//获取提示信息:user\@hostname:current_path$, 并且在第一次输出时, 给出更多的提示信息
void get_Prompt(void){
    uid = getuid(); //unistd.h
    user_info = getpwuid(uid);
    user_path_len = strlen(user_info->pw_dir);
    if(gethostname(host_name, 256)) //函数成功返回0, 失败返回-1, 错误号存在外部变量errno中
    {
        printf("\033[1;31m Error:Unable to get the hostname, something must be
wrong.\033[m\n");
        exit(-1);
    }
    if(!getcwd(current_path, 512)) //函数成功current_path的内存空间, 失败返回NULL
    {
        printf("\033[1;31m Error:Unable to get the current path, something must be
wrong.\033[m\n");
        exit(-1);
    }

    if(uid == 0) //root_uid == 0, root用户终端无颜色
    {
        printf("%s", user_info->pw_name);
        printf("@");
        printf("%s", host_name);
        printf(":");
        if(strlen(current_path) > user_path_len)
        {
            memcpy(current_user_path, current_path, user_path_len);
            current_user_path[user_path_len] = '\0';
            if(!strcmp(current_user_path, user_info->pw_dir))
            {
                printf("~");
                printf("%s", &current_path[user_path_len]);
            }
        }
    }
}
```

```

        else
            printf("%s",current_path);
    }
    else
    {
        printf("%s",current_path);
    }
    printf("# ");
}
else
{
    printf("\033[1;32m");
    printf("%s",user_info->pw_name);
    printf("@");
    printf("%s",host_name);
    printf("\033[m");
    printf(":");
    printf("\033[1;34m");
    if(strlen(current_path) > user_path_len)
    {
        memcpy(current_user_path, current_path, user_path_len);
        current_user_path[user_path_len] = '\0';
        if(!strcmp(current_user_path, user_info->pw_dir))
        {
            printf("~");
            printf("%s",&current_path[user_path_len]);
        }
        else
            printf("%s",current_path);
    }
    else
    {
        printf("%s",current_path);
    }
    printf("\033[m");
    printf("$ ");
}
return;
}

```

2.命令解析

由于存在输入多个空格的情况，所以我选择使用 `strtok()` 函数实现对于命令的分割，以 `' '` 作为分隔符

```

void analyse_Command(void)
{
    args_count = 0;
    args[args_count] = strtok(command, " ");
    char * ptr;
    while((ptr = strtok(NULL, " ")))
    {
        args_count++;
        args[args_count] = ptr;
        if(args_count + 1 == ARG_MAXM) //最多有256个参数
            break;
    }
}

```

3.历史命令的记录

在shell启动伊始，我们对于 `history` 数组进行初始化，每一个指针指向一个256char的空间,并对malloc是否成功进行检查

```

//初始化history数组
void init_history(){
    for(int i=0;i<10;i++){
        history[i] = (char *) malloc(sizeof(char)*SIZE_UNIT);
        check_malloc(history[i]);
    }
}

//检查malloc是否成功
void check_malloc(char *s){
    if(s == NULL){
        printf("\033[1;31mError: failed to mall!\033[m");
        exit(1);
    }
}

```

在程序运行对history记录时，我们根据需要对与history空间进行扩充

```

//记录history
void history_record(){
    if(count2 > 1){
        new_history = (char *) malloc(sizeof(char)*(SIZE_UNIT * count2));
        free(history[his_count]);
        history[his_count] = new_history;
    }
    memset(history[his_count], '\0', strlen(history[his_count]));
    memcpy(history[his_count], command, strlen(command));
    his_count++;
    if(his_full)

```



```

{
    his_ori++;
    his_ori %= 10;
}
if(his_count == 10)
{
    his_count = 0;
    his_full = 1;
}
}

```

4.程序执行

对于 `cd` (改变当前工作目录)、`history` (查看历史命令)、`exit` (退出程序) 命令进行重写, 其他命令直接可以调用 `execvp()` 执行, 并对于一些特殊指令进行识别, `&`、`!`、`1`、`!!` 等

- `cd` 命令可以直接调用 `chdir()` 函数改变当前工作目录, 需要注意的是对于字符串 `"~"` 我们应当单独解析——将其替换为用户工作目录后再进行字符串拼接
- `history` 命令则需要我们预先有一个储存历史命令的缓冲区, 同时当历史记录达到上限时我们应当进行清除, 这里我们选择模拟一个循环链表以在历史命令满之后每次输入命令时都会去除现存的最早的命令
- `exit` 命令则只需要在主进程中识别到该字符串时直接调用 `exit()` 即可

```

void exe(void){
    analyse_Command();
    if(!strcmp(args[0],"exit")){
        printf("
        ****      ***          ***          \n\
        *****      ***      ****          \n\
        *** ***      ***      ***** *** ***** \n\
        ***      ***** ***          *** *** *** \n\
        ***      ***** ***      *** *** *** \n\
        ***          ***      *****      *** *** \n\
        \033[1;31mExit the Noir_SHELL,THANK YOU FOR YOUR USE\n\033[m");
        exit(0);
    }
    if (!strcmp(args[0],"history"))
    {
        int i,j;
        if(his_full == 0){
            i = his_count;
            j = (his_ori+his_count-1) % 10;
        }else{
            i = 10;
            j = (his_ori-1+10) % 10;
        }
        do
        {

```

```

        printf("%d %s\n", i, history[j]);
        i--;
        j = (j + 10 - 1) % 10;
    } while (i >= 1);
    return;
}
if(!strcmp(args[0], "cd"))
{
    if(args_count > 2 || args_count == 0)
        printf("\033[1;31m Error: arguments wrong\033[m \n");
    else
    {
        if(args[1][0] == '~')
        {

            dir = malloc(strlen(args[1]) + strlen(user_info->pw_dir));
            strcpy(dir, user_info->pw_dir);
            strncat(dir, args[1] + 1, strlen(args[1]) - 1);
            chdir(dir);
            free(dir);
            // free(user_info);
        }
        else
            chdir(args[1]);
        //memset(args, '\0', 256);
    }
    return ;
}
run_on_Child();
return;
}

```

由于 `exec` 函数族执行成功不会返回，若执行失败，则会返回-1，并将失败原因存入 `errno`，所以我选择在返回值为-1时，给出执行错误的语句，并通过 `strerror(errno)` 读出错误信息

```

void run_on_Child()
{
    int pid = fork();
    errno = 0;
    if(pid < 0) // failed to fork a new thread
        printf("\033[1;31mError: Unable to fork the child, inner error.\033[m");
    else if(pid == 0) // the child thread
    {

        // printf("ls:%s", args[1]);
        int n = execvp(args[0], args);
        // `execlp()` 函数如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。
        printf("\033[1;31mError: unable to execute the programme: %s. something's
wrong.\n%s\033[m\n", args[0], strerror(errno));
    }
}

```

```

        exit(0);
    }
    else // the parent thread
        if(flag == FLAG_NO_BACKGROUND)
            wait(NULL); //waiting for the child to exit
    }
    //命令解析与执行

```

其中获取指令 `command` 时, 为了实现和 `ubuntu 20.04` 一样的最长长度, 我对其进行了动态实现, 每一个单元为 256 个字节, 每次输入一旦多于 256 个字节, 就再增加一个单元。

之后对于 `!!` 和 `!n` 进行了实现, 首先读取 `command` 中的前两个字符进行判断是否要执行 `history` 中的指令, 最后通过判断 `command` 中最后一个字符是否为 `&` 判断是否要后台运行, 完成执行之后最终释放 `command` 的空间。

```

//获取指令, 分割指令和参数, 识别history和!!
int get_command(void){
    count1=0,count2=1,count3=0;
    flag = FLAG_NO_BACKGROUND;
    command =(char*) malloc(sizeof(char)*SIZE_UNIT); //动态实现command, 因为ubuntu20.04命令
    行最长为2097152, 有点太长了
    check_malloc(command);
    memset(command, '\0', SIZE_UNIT);
    while((c = getchar()) != '\n')
    {
        if(count1 == count2 * SIZE_UNIT )
        {
            count2++;
            if(count2 * SIZE_UNIT >2097152)
            {
                printf("\033[1;31mError: command too long!\n\033[m");
                exit(1);
            }
            new_command = (char*) malloc(sizeof(char)*(SIZE_UNIT * count2));
            check_malloc(new_command);
            memcpy(new_command,command,(count2-1) * SIZE_UNIT);
            free(command);
            command = new_command;
        }
        command[count1++] = c;
    }
    command[count1] = '\0';
    if(count1 == 0){
        flag = NO_INPUT;
    }else{
        if(command[0] == '!'){
            if(command[1] == '!'){
                if(!his_full && his_count == 0)
                {

```

```

        puts("\033[1;31mError: No available command, history is
empty.\n\033[m");
        flag = NO_INPUT;
    }
    count3 = sizeof(history[((his_count + 10 - 1) % 10)]+count1);
    tmp = (char *) malloc(sizeof(count3));
    strcpy(tmp, history[((his_count + 10 - 1) % 10)]);
    strncat(tmp, command + 2, count1);
    strcpy(command,tmp); //这里可能会有溢出, 但是处理起来太麻烦了, 先放着吧
    free(tmp);
    printf("%s\n", command);
} else if (command[1] >= '0' && command[2] <= '9') {
    int his = atoi(command + 1);
    if (his <= 0 || his > 10 || history[his-1][0] == '\0')
    {
        puts("\033[31m\033[1mError: No available command, invalid history
index.\033[0m");
        return NO_INPUT;
    }
    count3 = sizeof(history[((his_ori+his+ 10 - 1) % 10)]+count1);
    tmp = (char *) malloc(sizeof(count3));
    strcpy(tmp, history[((his_ori+his+ 10 - 1) % 10)]);
    strncat(tmp, command + 2, count1);
    strcpy(command,tmp); //这里可能会有溢出, 但是处理起来太麻烦了, 先放着吧
    free(tmp);
    printf("%s\n", command);
}

}

}

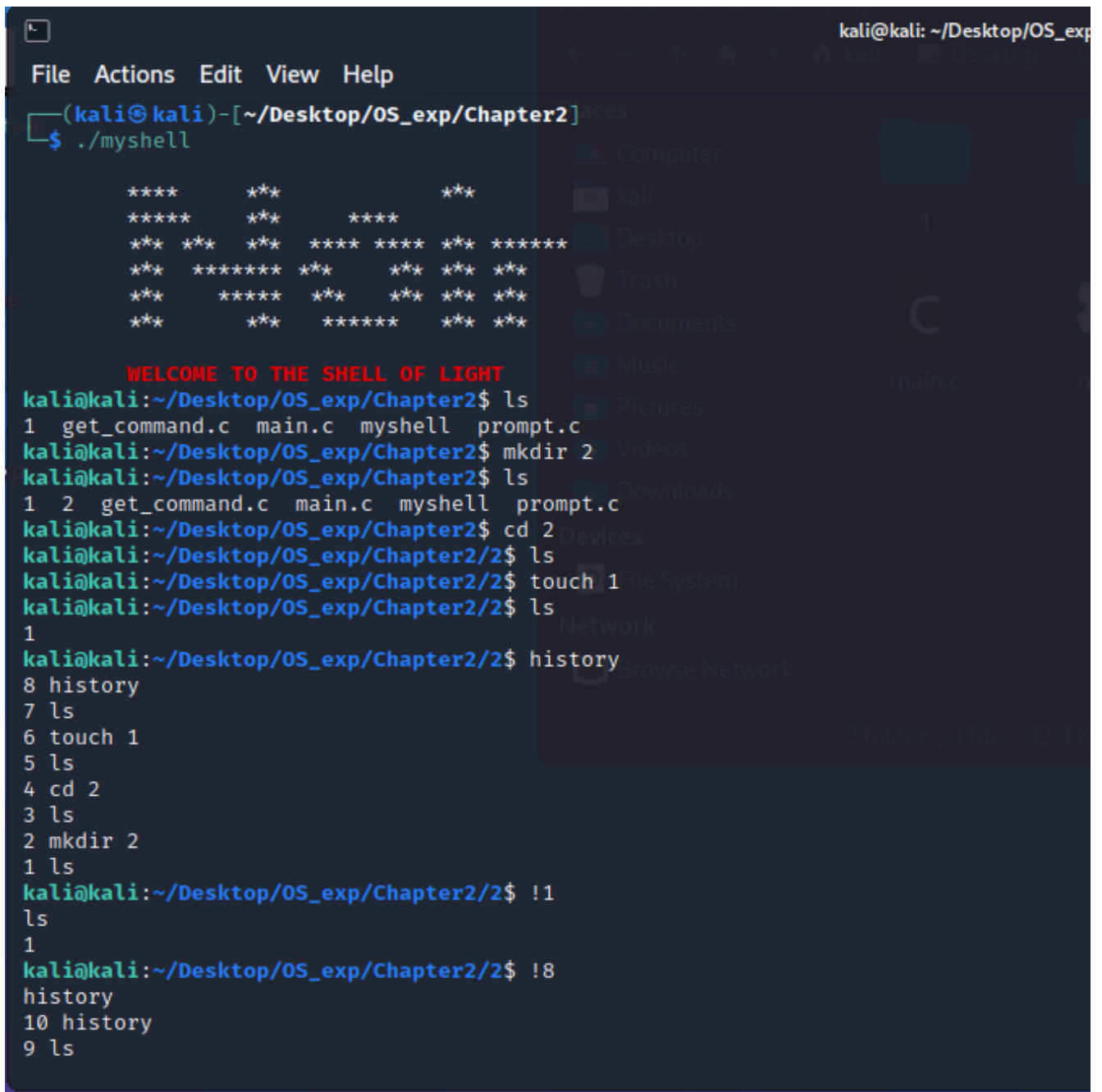
history_record();

if (command[count1 - 1] == '&')
{
    command[count1 - 1] = '\0';
    flag = FLAG_BACKGROUND;
}
if (flag == NO_INPUT)
{
    printf("\033[1;31mError: NO INPUT \n\033[m");
    return 0;
}
exe();
free(command);
return 0;
}

```

四、实验结果分析

1. 普通用户登陆界面



```
kali@kali: ~/Desktop/OS_exp
File Actions Edit View Help
(kali@kali)-[~/Desktop/OS_exp/Chapter2]
$ ./myshell

****      ***      ***
*****    ***      ****
*** **    ***      ****  ****  ***  *****
***  *****  ***      ***  ***  ***
***      *****  ***      ***  ***  ***
***      ***      *****  ***  ***

WELCOME TO THE SHELL OF LIGHT
kali@kali:~/Desktop/OS_exp/Chapter2$ ls
1 get_command.c main.c myshell prompt.c
kali@kali:~/Desktop/OS_exp/Chapter2$ mkdir 2
kali@kali:~/Desktop/OS_exp/Chapter2$ ls
1 2 get_command.c main.c myshell prompt.c
kali@kali:~/Desktop/OS_exp/Chapter2$ cd 2
kali@kali:~/Desktop/OS_exp/Chapter2/2$ ls
1
kali@kali:~/Desktop/OS_exp/Chapter2/2$ touch 1
kali@kali:~/Desktop/OS_exp/Chapter2/2$ ls
1
kali@kali:~/Desktop/OS_exp/Chapter2/2$ history
8 history
7 ls
6 touch 1
5 ls
4 cd 2
3 ls
2 mkdir 2
1 ls
kali@kali:~/Desktop/OS_exp/Chapter2/2$ !1
ls
1
kali@kali:~/Desktop/OS_exp/Chapter2/2$ !8
history
10 history
9 ls
```

2. root用户登陆界面


```

(kali㉿kali)-[~/Desktop/OS_exp/Chapter2]
$ ./myshell

      ****      ***      ***
      *****      ***      *****
      ***  ***  ***  ****  *****  ***  *****
      ***  ****  ***  ***  ***  ***  ***
      ***  ****  ***  ***  ***  ***  ***
      ***      ***  ****  ***  ***

      WELCOME TO THE SHELL OF LIGHT
kali@kali:~/Desktop/OS_exp/Chapter2$ sleep 10 &
[623494]
kali@kali:~/Desktop/OS_exp/Chapter2$ sleep 10

```

五、问题总结

1. cd无法正常使用的问题

由于本程序中输入 `command` 后，将 `command` 用 `strtok` 进行分割，导致后面有一些 `argument` 在后续使用时候没有进行清除，导致后续 `execvp` 时仍会输入后续参数，导致无法正常运行，每次使用完之后将 `args` 全部置为 `\0` 即可

2. 在前面使用 `mkdir` 之后后续 `ls` 无法正常使用

这个问题原因我找了好久，因为报错是 `ls: can't access ''` 导致我一度认为是 `mkdir` 系统调用的问题，后来经过不断的测试修改，发现如果使用 `echo` 输入之间多个空格，也会导致这种原因，意识到 `strtok` 会将所有空格进行划分，如果对此改进就需要重写 `echo` 命令，还要实现 `>` 功能，过于繁琐，就没有进行修缮。但是这让我发现了 `ls: can't access ''` 的原因，还是 `args` 字符串数组的问题，具体应该是每次用完没有对字符串数组进行清理，之后后面就会多结果为 `''` 的参数导致报错，只需要在每次使用 `args` 字符串数组之前进行 `memset(args, '\0', 256)` 即可

六、源代码

1. prompt.c //用于给出提示信息

```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <pwd.h>

uid_t uid;
int user_path_len;
char host_name[256]; //linux主机名最长255个字符
char current_user_path[256]; //用于获取当前目录的user_path
char current_path[512];

```

```

struct passwd * user_info = NULL;

//获取提示信息:user\@hostname:current_path$, 并且在第一次输出时, 给出更多的提示信息
void get_Prompt(void){
    uid = getuid();//unistd.h
    user_info = getpwuid(uid);
    user_path_len = strlen(user_info->pw_dir);
    if(gethostname(host_name,256))//函数成功返回0, 失败返回-1, 错误号存在外部变量errno中
    {
        printf("\033[1;31m Error:Unable to get the hostname, something must be
wrong.\033[m\n");
        exit(-1);
    }
    if(!getcwd(current_path,512))//函数成功current_path的内存空间,失败返回NULL
    {
        printf("\033[1;31m Error:Unable to get the current path, something must be
wrong.\033[m\n");
        exit(-1);
    }

    if(uid == 0)//root_uid == 0, root用户终端无颜色
    {
        printf("%s",user_info->pw_name);
        printf("@");
        printf("%s",host_name);
        printf(":");
        if(strlen(current_path) > user_path_len)
        {
            memcpy(current_user_path, current_path, user_path_len);
            current_user_path[user_path_len] = '\0';
            if(!strcmp(current_user_path, user_info->pw_dir))
            {
                printf("~");
                printf("%s",&current_path[user_path_len]);
            }
            else
                printf("%s",current_path);
        }
        else
        {
            printf("%s",current_path);
        }
        printf("# ");
    }
    else
    {
        printf("\033[1;32m");
        printf("%s",user_info->pw_name);
    }
}

```



```

    printf("@");
    printf("%s",host_name);
    printf("\033[m");
    printf(":");
    printf("\033[1;34m");
    if(strlen(current_path) > user_path_len)
    {
        memcpy(current_user_path, current_path, user_path_len);
        current_user_path[user_path_len] = '\0';
        if(!strcmp(current_user_path, user_info->pw_dir))
        {
            printf("~");
            printf("%s",&current_path[user_path_len]);
        }
        else
            printf("%s",current_path);
    }
    else
    {
        printf("%s",current_path);
    }
    printf("\033[m");
    printf("$ ");
}
return;
}

```

2. get_command.c //用于获取命令，分析并执行

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/wait.h>
#include"prompt.c"
#include<errno.h>

#define SIZE_UNIT 256
#define NO_INPUT 1
#define FLAG_BACKGROUND 2
#define FLAG_NO_BACKGROUND 3
#define ARG_MAXM 0x100
pid_t pid;

char *command;
char *new_command;
int count1,count2,count3;
int his_count = 0,his_ori = 0,his_full=0;

```

```

char c;//逐字符读取的缓冲
static char *history[10];
char *new_history;
int flag;
char *tmp;
char * args[0x100];
int args_count ;
char *dir;
struct passwd * user_infol = NULL;

//检查malloc是否成功
void check_malloc(char *s){
    if(s == NULL){
        printf("\033[1;31mError: failed to mall!\033[m");
        exit(1);
    }
}

void analyse_Command(void)
{
    memset(args, '\0', 256);
    args_count = 0;
    args[args_count] = (char *) malloc(sizeof(char) * 256);
    args[args_count] = strtok(command, " ");
    char * ptr;
    while((ptr = strtok(NULL, " ")))
    {
        args_count++;
        args[args_count] = (char *) malloc(sizeof(char) * 256);
        args[args_count] = ptr;
        if(args_count + 1 == ARG_MAXM)//最多有256个参数
            break;
    }
}

void run_on_Child()
{
    if(flag == FLAG_NO_BACKGROUND){
        int pid = fork();
        errno = 0;
        if(pid < 0) // failed to fork a new thread
            printf("\033[1;31mError: Unable to fork the child, inner error.\033[m");
        else if(pid == 0) // the child thread
        {
            // printf("ls:%s",args[1]);
            int n = execvp(args[0], args);
        }
    }
}

```

```

        // `execlp()` 函数如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno
        中。

        printf("\033[1;31mError: unable to execute the programme: %s. something\'s
wrong.\n%s\033[m\n", args[0],strerror(errno));
        exit(0);
    }else{
        wait(NULL);
    }

}
else{
    int pid = fork();
    errno = 0;
    if(pid < 0) // failed to fork a new thread
        printf("\033[1;31mError: Unable to fork the child, inner error.\033[m");
    else if(pid == 0) // the child thread
    {
        // printf("ls:%s",args[1]);
        int n = execvp(args[0], args);
        // `execlp()` 函数如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno
        中。

        printf("\033[1;31mError: unable to execute the programme: %s. something\'s
wrong.\n%s\033[m\n", args[0],strerror(errno));
        exit(0);
    }else{
        printf("[%d]\n",pid);
    }
}
}

//初始化history数组
void init_history(){
    for(int i=0;i<10;i++){
        history[i] = (char *) malloc(sizeof(char)*SIZE_UNIT);
        check_malloc(history[i]);
    }
}

//记录history
void history_record(){
    if(count2 > 1){
        new_history = (char *) malloc(sizeof(char)*(SIZE_UNIT * count2));
        free(history[his_count]);
        history[his_count] = new_history;
    }
    memset(history[his_count], '\0', strlen(history[his_count]));
    memcpy(history[his_count], command, strlen(command));
    his_count++;
    if(his_full)

```

```

{
    his_ori++;
    his_ori %= 10;
}
if(his_count == 10)
{
    his_count = 0;
    his_full = 1;
}
}

```

//命令解析与执行

```

void exe(void){
    analyse_Command();
    if(!strcmp(args[0], "exit")){
        printf("
        ****      ***          ***          \n\
        *****      ***      ****          \n\
        *** ***      ***      **** **** *** ***** \n\
        ***      ***** ***      *** *** *** \n\
        ***      *****      ***      *** *** *** \n\
        ***      ***      *****      *** *** \n\
        \033[1;31mExit the Noir_SHELL, THANK YOU FOR YOUR USE\n\033[m");
        exit(0);
    }
    if (!strcmp(args[0], "history"))
    {
        int i,j;
        if(his_full == 0){
            i = his_count;
            j = (his_ori+his_count-1) % 10;
        }else{
            i = 10;
            j = (his_ori-1+10) % 10;
        }

        do
        {
            printf("%d %s\n", i, history[j]);
            i--;
            j = (j + 10 - 1) % 10;
        } while (i>=1);
        return;
    }
    if(!strcmp(args[0], "cd"))
    {
        if(args_count > 2 || args_count == 0)

```

```

    printf("\033[1;31m Error: arguments wrong\033[m \n");
else
{
    if(args[1][0] == '~')
    {
        dir = malloc(strlen(args[1]) + strlen(user_info->pw_dir));
        strcpy(dir, user_info->pw_dir);
        strncat(dir, args[1] + 1, strlen(args[1]) - 1);
        chdir(dir);
        free(dir);
        // free(user_info);
    }
    else
        chdir(args[1]);
    //memset(args, '\0', 256);
}
return ;
}

run_on_Child();
return;
}

//获取指令，分割指令和参数，识别history和!!
int get_command(void){
    count1=0,count2=1,count3=0;
    flag = FLAG_NO_BACKGROUND;
    command =(char*) malloc(sizeof(char)*SIZE_UNIT); //动态实现command，因为ubuntu20.04命令
    行最长为2097152，有点太长了
    check_malloc(command);
    memset(command, '\0', SIZE_UNIT);
    while((c = getchar()) != '\n')
    {
        if(count1 == count2 * SIZE_UNIT )
        {
            count2++;
            if(count2 * SIZE_UNIT >2097152)
            {
                printf("\033[1;31mError: command too long!\n\033[m");
                exit(1);
            }
            new_command = (char*) malloc(sizeof(char)*(SIZE_UNIT * count2));
            check_malloc(new_command);
            memcpy(new_command,command,(count2-1) * SIZE_UNIT);
            free(command);
            command = new_command;
        }
        command[count1++] = c;
    }
}

```

```

command[count1] = '\0';
if(count1 == 0){
    flag = NO_INPUT;
}else{
    if(command[0] == '!'){
        if(command[1] == '!'){
            if(!his_full && his_count == 0)
            {
                puts("\033[1;31mError: No available command, history is
empty.\n\033[m");
                flag = NO_INPUT;
            }
            count3 = sizeof(history[((his_count + 10 - 1) % 10)])+count1;
            tmp = (char *) malloc(sizeof(count3));
            strcpy(tmp, history[((his_count + 10 - 1) % 10)]);
            strncat(tmp, command + 2, count1);
            strcpy(command,tmp);
            free(tmp);
            printf("%s\n", command);
        }else if(command[1]>='0' && command[2]<='9'){
            int his = atoi(command + 1);
            if (his <= 0 || his > 10 || history[his-1][0]=='\0')
            {
                puts("\033[31m\033[1mError: No available command, invalid history
index.\033[0m");
                return NO_INPUT;
            }
            count3 = sizeof(history[((his_ori+his+ 10 - 1) % 10)])+count1;
            tmp = (char *) malloc(sizeof(count3));
            strcpy(tmp, history[((his_ori+his+ 10 - 1) % 10)]);
            strncat(tmp, command + 2, count1);
            strcpy(command,tmp);
            free(tmp);
            printf("%s\n", command);
        }
    }
}

history_record();
if(command[count1 - 1] == '&')
{
    command[count1 - 1] = '\0';
    flag = FLAG_BACKGROUND;
}
if (flag == NO_INPUT)
{
    printf("\033[1;31mError: NO INPUT \n\033[m");
}

```

```

        return 0;
    }
    exe();
    free(command);
    return 0;
}

```

3. main.c //主程序部分

```

#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include"get_command.c"
#define SHOULD_RUN 1

int main(void){
    printf("
        ****      ***          ***          \n\
        *****      ***      ****          \n\
        ***  ***  ***  *****  ***  ***** \n\
        ***  *****  ***      ***  ***  *** \n\
        ***      *****  ***      ***  ***  *** \n\
        ***          ***      *****      ***  *** \n\
        \033[1;31mWELCOME TO THE SHELL OF LIGHT\033[m\n");
    init_history();
    while(SHOULD_RUN){
        get_Prompt();
        get_command();
    }
    return 0;
}

```