操作系统专题实践实验三

Shell的实现

71121123 肖以成 2023/12/12

实验目的

通过实验了解Shell实现机制

实验内容

实现具有管道、重定向功能的shell，能够执行一些简单的基本命令，如进程执行、列目录等

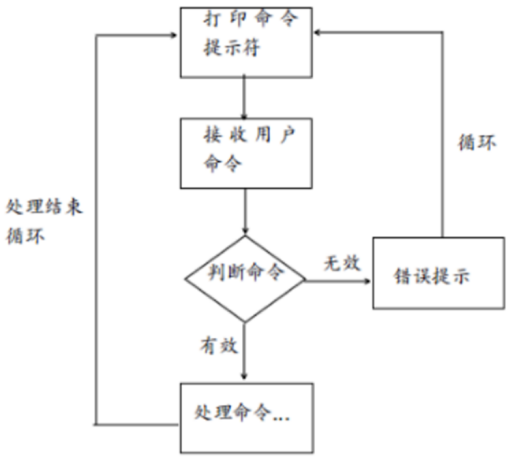
具体要求

1. 设计一个C语言程序，完成最基本的shell角色：给出命令行提示符、能够逐次接受命令；
   对于命令分成三种:

- 内部命令（例如help命令、exit命令等）
- 外部命令（常见的ls等，以及其他磁盘上的可执行程序HelloWrold等）无效
- 命令（不是上述二种命令）

2. 具有支持管道的功能，即在shell中输入诸如 “dir | more”能够执行dir命令并将其输出通过管道将其输入传送给more。
3. 具有支持重定向的功能，即在shell中输入诸如“dir > direct.txt”能够执行dir命令并将结果输出到direct.txt
4. 将上述步骤直接合并完成

设计思路



1. 启动后常驻在终端，通过读入用户输入的指令识别语义，并调用相应函数进行处理，因此可以设计为一个while(1)的死循环，使程序反复等待用户输入指令。过程中首先需要在终端输出当前用户和当前目录，这两者可以通过调用系统调用获取。
2. 用户输入指令后，可能为单独的诸如cd、内部指令，因此首先判断是否为内部指令，是的单独执行。
3. 判断是否存在重定向及管道格式。所以，首先，需要找到用户命令中所有的“|”符号，将其进行分割，按顺序逐个处理，并将上一个的输出作为下一个独立命令函数中的输入通过管道传输。除了管道之外，指令中还存在重定向，所以对于独立的指令，需要判断指令中是否有“>”或“<”符号。如果有的话，则还要将指令的输出重定向到文件。
4. 在对指令分割完成后，分析指令本身语义。一个独立指令应包含两个部分：指令本身和指令的参数。在处理时首先使用strtok根据指令的前几个字母识别出指令内容（如果没有找到匹配项则认为指令错误），然后基于空格和“-”将参数分开。

依赖

安装readline.h的库以实现功能拓展

```
yum install libtermcap-devel
    ncurses-devel
    libevent-devel
    readline-devel
```

编译时需额外添加-lreadline的参数

源程序

1. 主循环

利用getcwd函数获取当前路径，利用strcat函数拼接shell名称、当前路径及$分隔符。通过readline来获取命令行输入，当命令不为内部命令时，生成子进程，调用pipel函数。

```
while (1) {
        getcwd(buf, sizeof(buf));
        if (strcmp(tmp, buf) == 0) {
                memset(buf, 0, sizeof(buf));
                buf[0] = '~';
                buf[1] = '\0';
        }
        sprintf (prompt, "%s:%s$" , username, buf);
        char *t;
        t=readline (prompt);
    ...
    if ((pid = fork ()) == 0) {
                pipel(order[i]);
        }
        else if (back == 0){
                currentpid=pid;
                waitpid (pid, &status, WUNTRACED);
        }
}
```

1. pipel部分处理

pipel函数首先利用strtok函数判断是否存在“|”分隔符，若存在则说明出现管道命令。strtok函数会将分隔符前的参数作为char* 类型返回，本函数用order接受。再次调用strtok函数时，第一个参数需设置为NULL，使得该函数默认使用上一次未分割完的字符串继续分割。char* trim (const char *str)为自定义函数，用于去除命令首尾的空格以免解析时产生干扰。

- pipel函数：

```
int pipel(char *cmd) {
        char *trim (const char *str);
        int redirect (char *cmd);
        int fd[2], status, pid;
        char *order, *other;
        order = trim( strtok(cmd, "|"));
        other = trim( strtok (NULL, "" ));
        if (!other)
                redirect(order);
        else {
                pipe(&fd[0]);
                if ((pid = fork ()) == 0) {
                        close(fd[0]);
                        close(STD_OUT);
                        dup(fd[1]);
                        close(fd[1]);
                        redirect(order);
                } else {
                        close(fd[1]);
                        close(STD_IN);
                        dup(fd[0]);
                        close(fd[0]);
                        waitpid(pid, &status, 0);
                        pipel(other);
                }
        }
        return 1;
}
```

- trim函数:

```
char *trim (const char *str) {
        int i, j, k;
        char *order;
        if (str == NULL)
                return NULL;
        for (i = 0; i < strlen(str); i++)
                if (str[i] != ' ')
                        break;
        for (j = strlen(str) - 1; j > -1; j--)
                if (str[j] != ' ')
                        break;
        if (i <= j) {
                if ((order = ( char *) malloc((j - i + 2) * ( sizeof(char)))) == 0) {
                        fprintf (stderr, "#error: can't malloc enough space\n" );
                        return NULL;
                }
                for (k = 0; k < j - i + 1; k++)
                        order[k] = str[k + i];
                order[k] = '\0';
                return order;
        } else
        return NULL;
}
```

3. redirect处理

redirect函数首先根据<、>存在的位置和个数判断当前重定向操作属于那种类型。

> type 2：无任何<、>，不进行任何重定向操作，直接执行命令；
> type 3：有一个<，设置 < 后文件为重定向输入文件；
> type 4：有一个>，设置 > 后文件为重定向输出文件；
> type 5：依次有一个>、<，设置 > 后为重定向输出文件，< 后为重定向输入文件；
> type 6：依次有一个<、>，设置 < 后未重定向输入文件，> 后未重定向输出文件。

然后获取用户指令的第一个参数 (如cat)，并用自定义函数 if_exist函数获取命令所在的地址，方便后续使用exec族函数调用。这时再根据 type 获取输入输出文件名。若为输入重定向，先关闭STD_ID（互斥），之后调用dup将写入重定向到infile文件，最后关闭STD_OUT；对于输出重定向，进行类似的操作。重定向完成后，最后调用exec族函数执行命令。

- redirect函数

```
int redirect(char *cmd) {
        char *trim (const char *str);
        void do_cd(char *argv[]);
        char *order = trim(cmd), *order_path, *real_order;
        char *infile, *outfile, *arg[MAXPARA], *buffer;
        int i, type = 2, fd_out, fd_in;
        for (i = 0; i < strlen(cmd); i++) {
                if (cmd[i] == '<')
                        type++;
                if (cmd[i] == '>')
                        type = type * 2;
        }
        if (type == 3 || type == 6)
                real_order = trim( strtok (cmd, "<" ));
        else if (type == 4 || type == 5)
                real_order = trim( strtok (cmd, ">" ));
        else if (type == 2)
                real_order = trim(cmd);
        else {
                fprintf(stderr, "#error: bad redirection form\n" );
                return -1;
        }
        arg[0] = trim(strtok(real_order, " " ));
        for (i = 1; (arg[i] = trim( strtok (NULL, " " ))) != NULL; i++);
        if (strcmp (arg[0], "history" ) == 0) {
                while (head->next != NULL) {
                        printf ("id:%d %s\n" , head->id , head->cmd);
                        head = head->next;
                }
                exit(1);
                return 1;
        }
        if (strcmp (arg[0], "jobs" ) == 0) {
                int i = 1;
                for (; i < MAX_BACK_JOBS_NUM; i++) {
                        if (back_jobs[i] != NULL)
                                printf ("[%d] %d %s\t\t\t\t%s\n" , i, back_jobs[i]-> pid, st[ba
                }
                exit(1);
                return 1;
        }
        if (strcmp (arg[0], "help" ) == 0) {
                do_help(arg[1]);
```

```
        exit(1);
        return 1;
}
if (strcmp (arg[0], "pwd" ) == 0) {
        do_pwd();
        exit(1);
        return 1;
}
if ((order_path = if_exist(arg[0])) == NULL) {
        fprintf (stderr, "#error: this command doesn't exist\n" );
        exit(1);
        return -1;
}
switch (type) {
        case 2:
                break;
        case 3:
                buffer = strtok (order, "<" );
                infile = trim( strtok (NULL, "" ));
                break;
        case 4:
                buffer = strtok (order, ">" );
                outfile = trim( strtok(NULL, "" ));
                break;
        case 5:
                buffer = strtok (order, ">" );
                outfile = trim( strtok(NULL, "<" ));
                infile = trim( strtok (NULL, "" ));
                break;
        case 6:
                buffer = strtok (order, "<" );
                infile = trim( strtok (NULL, ">" ));
                outfile = trim( strtok(NULL, "" ));
                break;
        default:
                return -1;
}
if (type == 4 || type == 5 || type == 6) {
        if ((fd_out = creat(outfile, 0755)) == -1) {
                fprintf (stderr, "#error: redirect standard out error\n" );
                return -1;
        }
        close(STD_OUT);
```

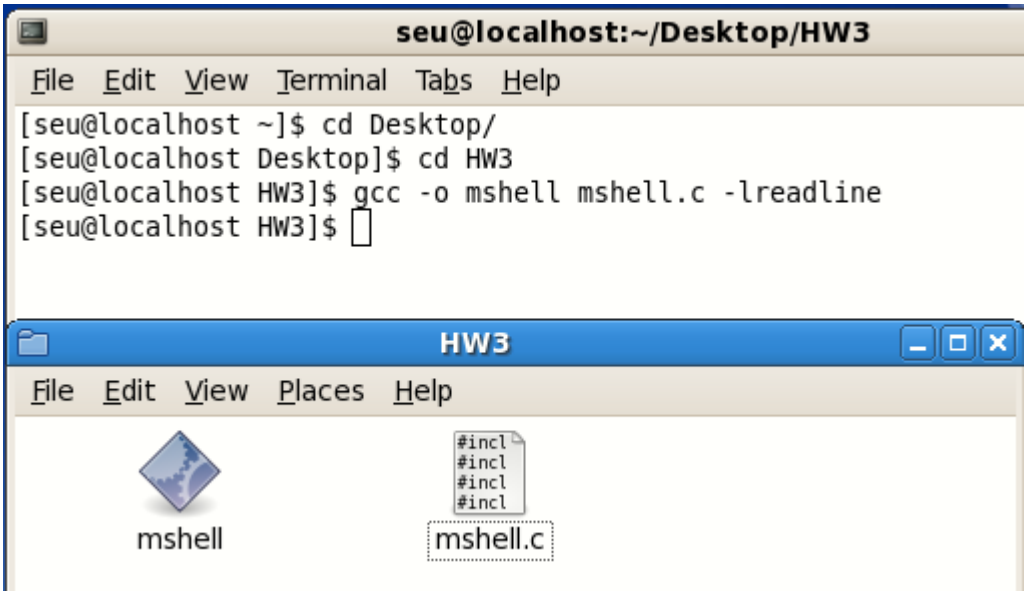
```
                dup(fd_out);
                close(fd_out);
        }
        if (type == 3 || type == 5 || type == 6) {
                if ((fd_in = open(infile, O_RDONLY, S_IRUSR | S_IWUSR)) == -1) {
                        fprintf (stderr, "#error: can't open inputfile '%s'\n" , infile);
                        return -1;
                }
                close(STD_IN);
                dup(fd_in);
                close(fd_in);
        }
        execv(order_path, arg);
        exit(0);
        return 1;
}
```

- if_exist函数

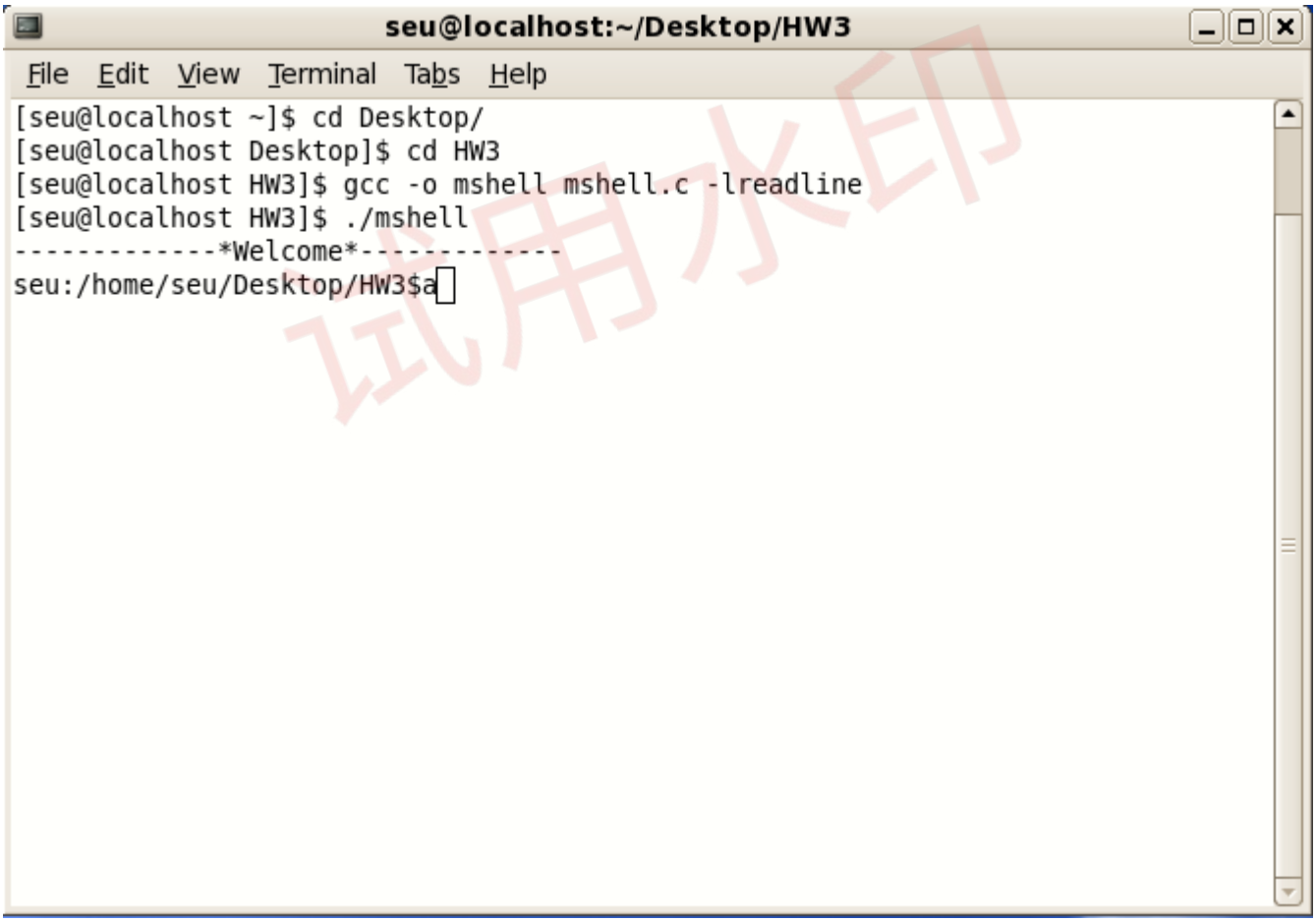
```
char *if_exist(char *order) {
        char *all_path, *p, *path, *buffer;
        int len;
        all_path = getenv("PATH");
        buffer = trim(all_path);
        len = strlen(all_path) + strlen(order);
        if ((path = (char *) malloc(len * ( sizeof(char)))) == 0) {
                fprintf (stderr, "#error: can't malloc enough space for buffer\n" );
                return NULL;
        }
        p = strtok (buffer, ":" );
        while (p) {
                strcat(strcat(strcpy(path, p), "/" ), order);
                if (access(path, F_OK) == 0) {
                        return path;
                }
                p = strtok (NULL, ":" );
        }
        strcpy(path, order);
        if (access(path, F_OK) == 0)
                return path;
        return NULL;
}
```

实验结果

1. 编译



2. 启动shell



3. 运行命令

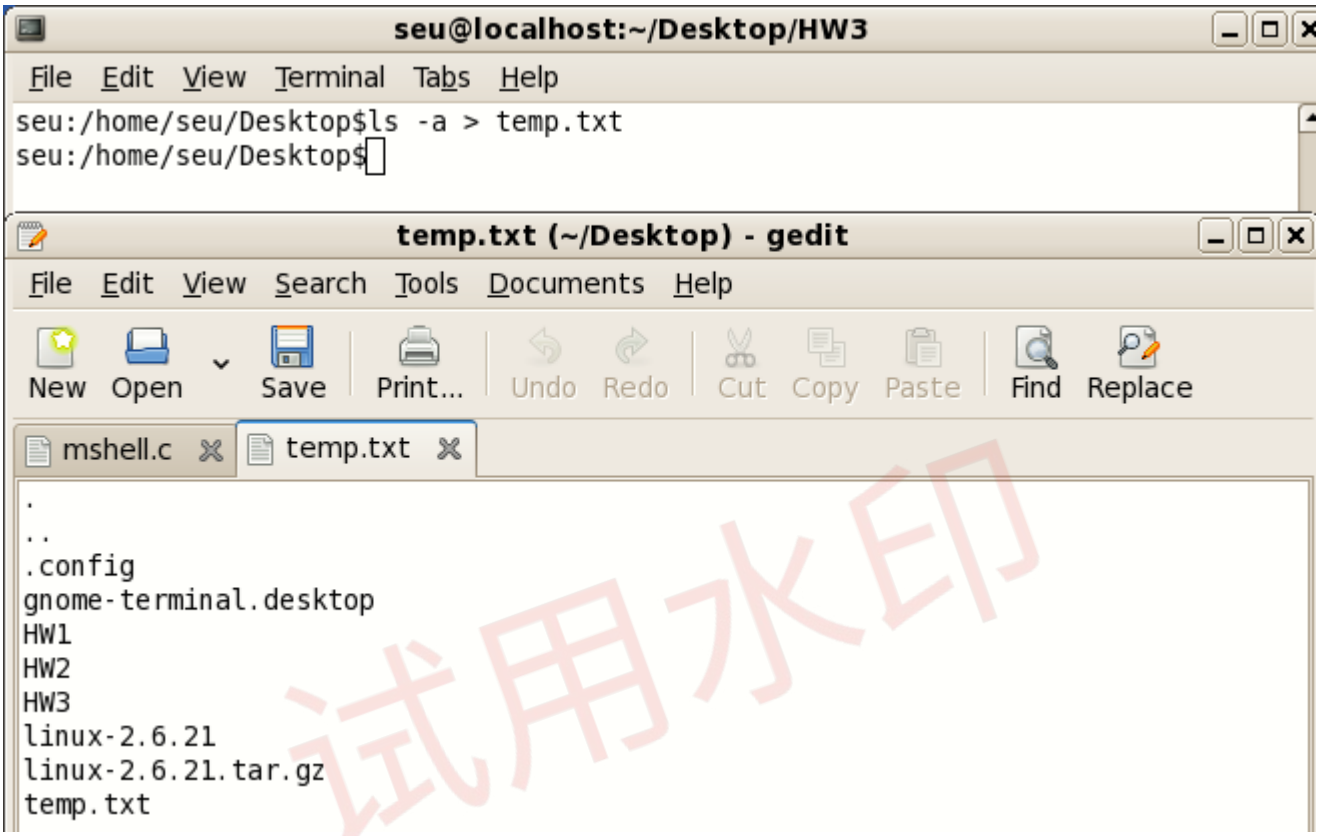
- ls

```
-------------*Welcome*-------------
seu:/home/seu/Desktop/HW3$ls -a
.  ..  mshell  mshell.c  mshell.c~
seu:/home/seu/Desktop/HW3$
```

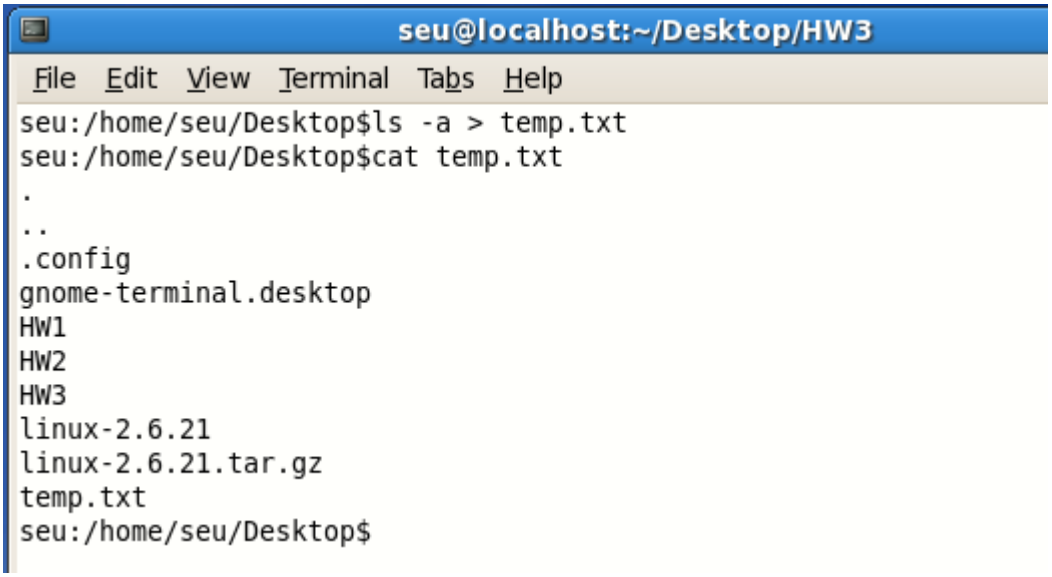
- cd

```
seu:/home/seu/Desktop$cd HW3
seu:/home/seu/Desktop/HW3$
```

- 重定向



- cat



- grep + 管道

```
seu:/home/seu/Desktop$ls | grep temp | grep .txt
temp.txt
seu:/home/seu/Desktop$
```

- 运行外部命令

```
seu:/home/seu/Desktop$./hello
hello world!
seu:/home/seu/Desktop$
```

> hello.c内容如下：
>
> ```c
> #include <stdio.h>
>
> int main(int argc,char* argv[])
> {
>         printf("hello world!\n");
>         return 0;
> }
> ```

实验体会

本次实验手工实现了一个简单的shell，加深了Linux编程的理解，深入理解了重定向、管道等概念，也基本了解了Shell实现的部分实施方法~

其他

添加的readline.h需要额外的配置，这里均按照对应教程编写的代码。
教程地址：

```
https://tiswww.case.edu/php/chet/readline/rltop.html
https://www.360docs.net/doc/f57644068.html
https://cnblogs.com/LiuYanYGZ/p/14806139.html
```

- 采用的数据结构

```
// help指令的构建
struct HELP_DOC {
        char *usage[lengthOfBUILTIN_COMMANDS];
        char *info[lengthOfBUILTIN_COMMANDS];
};
// 记录jobs后台存储
typedef struct BACK_JOBS {
        pid_t pid ;
        char * cmd;
        int status;
}BACK_JOB;
// 用于Command的history存储
typedef struct Node {
        int id;
        char cmd[100];
        struct Node * next;
} NODE;
```

- 通过以下代码进行初始化

```
char *command_generator (const char *text, int state) {
        char *name;
        static int list_index, len;
        if (!state) {
                list_index = 0;
                len = strlen (text);
        }
        while (name = commands[list_index]) {
                list_index++;
                if (strncmp(name, text, len) == 0)
                        return (strdup (name));
        }
        return ((char *)NULL);
}

char **command_completion(const char *text, int start, int end) {
        char **matches = NULL;
        if (start == 0)
                matches = rl_completion_matches (text, command_generator);
        return (matches);
}

void initialize_readline() {
        rl_attempted_completion_function = (CPPFunction*)command_completion;
}

void initWithHelpDoc (struct HELP_DOC *help_doc) {
        help_doc-> usage[EXIT] = "exit: exit" ;
        help_doc-> info[EXIT] = "Exit the shell." ;
        help_doc-> usage[CD] = "cd: cd [dir]" ;
        help_doc-> info[CD] = "\n\tThe default DIR is the value of the HOME shell variable." ;
        help_doc-> usage[HISTORY] = "history: history [-c] [-s num]" ;
        help_doc-> info[HISTORY] = "\n\tentry with a `*'. \n\t \n\t -s num\tsize of the history
        help_doc-> usage[PWD] = "pwd: pwd" ;
        help_doc-> info[PWD] = "Print the name of the current working directory." ;
        help_doc-> usage[HELP] = "help: help [pattern ...]" ;
        help_doc-> info[HELP] = "\n\t PATTERN Pattern specifiying a help topic" ;
        help_doc-> usage[JOBS] = "jobs: jobs" ;
        help_doc-> info[JOBS] = "\n\tLists the active jobs. JOBSPEC restricts output to that jol
}
```

源代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <grp.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <readline/readline.h>
#include <readline/history.h>

#define STD_IN 0
#define STD_OUT 1
#define MAXORD 20
#define MAXPARA 8
#define MAX_BACK_JOBS_NUM 20
#define SIGSTCP 20
#define lengthOfBUILTIN_COMMANDS 10

static int back_jobs_ptr = 0;
char *st[] = { "DONE" , "RUNNING" , "STOPPED" };
// current shell pid
int currentpid;
int isCtrlz;
enum BUILTIN_COMMANDS { NO_SUCH_BUILTIN =0, EXIT, CD, HISTORY, DO_HIS_CMD , PWD , KILL , HELP ,
//for tab completion
char *commands[] = { "cd", "cp", "chmod", "exit",
"mv", "man", "rm", "rmdir", "vi", "bg", "fg", "grep", "ls", "cat",
"history", "help", "jobs", "kill", "ps", "pwd"};

struct HELP_DOC {
        char * usage[lengthOfBUILTIN_COMMANDS];
        char *info [lengthOfBUILTIN_COMMANDS];
};

char *trim (const char *str);
int is_back(char *order);
int pipel(char *cmd);
```

```
char *if_exist(char *order);
void handle_sigchld(int s);
void handle_sigint(int s);
void handle_sigstcp(int s);
void initWithHelpDoc (struct HELP_DOC*);
void do_help(char *);
void do_pwd();
// deamon process
typedef struct BACK_JOBS {
        pid_t pid ;
        char * cmd;
        int status;
}BACK_JOB;

typedef struct Node {
        int id;
        char cmd[100];
        struct Node * next;
} NODE;

struct BACK_JOBS *back_jobs[MAX_BACK_JOBS_NUM];
static NODE *head;
struct HELP_DOC * help_doc;

int pipel(char *cmd) {
        char *trim (const char *str);
        int redirect (char *cmd);
        int fd[2], status, pid;
        char *order, *other;
        order = trim( strtok(cmd, "|"));
        other = trim( strtok (NULL, "" ));
        if (!other)
                redirect(order);
        else {
                pipe(&fd[0]);
                if ((pid = fork ()) == 0) {
                        close(fd[0]);
                        close(STD_OUT);
                        dup(fd[1]);
                        close(fd[1]);
                        redirect(order);
                } else {
                        close(fd[1]);
```

```
                        close(STD_IN);
                        dup(fd[0]);
                        close(fd[0]);
                        waitpid(pid, &status, 0);
                        pipel(other);
                }
        }
        return 1;
}

int redirect(char *cmd) {
        char *trim (const char *str);
        void do_cd(char *argv[]);
        char *order = trim(cmd), *order_path, *real_order;
        char *infile, *outfile, *arg[MAXPARA], *buffer;
        int i, type = 2, fd_out, fd_in;
        for (i = 0; i < strlen(cmd); i++) {
                if (cmd[i] == '<')
                        type++;
                if (cmd[i] == '>')
                        type = type * 2;
        }
        if (type == 3 || type == 6)
                real_order = trim( strtok (cmd, "<" ));
        else if (type == 4 || type == 5)
                real_order = trim( strtok (cmd, ">" ));
        else if (type == 2)
                real_order = trim(cmd);
        else {
                fprintf(stderr, "#error: bad redirection form\n" );
                return -1;
        }
        arg[0] = trim(strtok(real_order, " " ));
        for (i = 1; (arg[i] = trim( strtok (NULL, " " ))) != NULL; i++);
        if (strcmp (arg[0], "history" ) == 0) {
                while (head->next != NULL) {
                        printf ("id:%d %s\n" , head->id , head->cmd);
                        head = head->next;
                }
                exit(1);
                return 1;
        }
        if (strcmp (arg[0], "jobs" ) == 0) {
```

```
        int i = 1;
        for (; i < MAX_BACK_JOBS_NUM; i++) {
                if (back_jobs[i] != NULL)
                        printf ("[%d] %d %s\t\t\t\t%s\n" , i, back_jobs[i]-> pid, st[bac
        }
        exit(1);
        return 1;
}
if (strcmp (arg[0], "help" ) == 0) {
        do_help(arg[1]);
        exit(1);
        return 1;
}
if (strcmp (arg[0], "pwd" ) == 0) {
        do_pwd();
        exit(1);
        return 1;
}
if ((order_path = if_exist(arg[0])) == NULL) {
        fprintf (stderr, "#error: this command doesn't exist\n" );
        exit(1);
        return -1;
}
switch (type) {
        case 2:
                break;
        case 3:
                buffer = strtok (order, "<" );
                infile = trim( strtok (NULL, "" ));
                break;
        case 4:
                buffer = strtok (order, ">" );
                outfile = trim( strtok(NULL, "" ));
                break;
        case 5:
                buffer = strtok (order, ">" );
                outfile = trim( strtok(NULL, "<" ));
                infile = trim( strtok (NULL, "" ));
                break;
        case 6:
                buffer = strtok (order, "<" );
                infile = trim( strtok (NULL, ">" ));
                outfile = trim( strtok(NULL, "" ));
```

```
                        break;
                default:
                        return -1;
        }
        if (type == 4 || type == 5 || type == 6) {
                if ((fd_out = creat(outfile, 0755)) == -1) {
                        fprintf (stderr, "#error: redirect standard out error\n" );
                        return -1;
                }
                close(STD_OUT);
                dup(fd_out);
                close(fd_out);
        }
        if (type == 3 || type == 5 || type == 6) {
                if ((fd_in = open(infile, O_RDONLY, S_IRUSR | S_IWUSR)) == -1) {
                        fprintf (stderr, "#error: can't open inputfile '%s'\n" , infile);
                        return -1;
                }
                close(STD_IN);
                dup(fd_in);
                close(fd_in);
        }
        execv(order_path, arg);
        exit(0);
        return 1;
}

int is_back(char *order) {
        int len = strlen(order);
        if (order[len - 1] == '&' ) {
                order[len] = '\0';
                return 1;
        }
        else
                return 0;
}

void do_cd(char *argv[]) {
        if (argv[1] != NULL) {
                if (chdir(argv[1]) < 0) {
                        switch (errno) {
                                case ENOENT:
                                        fprintf (stderr, "#error: directory can't be found\n" );
```

```
                                        break ;
                                case ENOTDIR:
                                        fprintf (stderr, "#error: this is not a directory name\r
                                        break ;
                                case EACCES:
                                        fprintf (stderr, "#error: you have no right to access\n"
                                        break ;
                                default:
                                        fprintf (stderr, "#error: unknown error\n" );
                        }
                }
        }
}

void do_help(char *argv){
        int i;
        if(argv==NULL){
                i=HELP ;
        }
        else if(strcmp(argv, "cd" )==0){
                i=CD ;
        } else if(strcmp(argv, "exit" )==0){
                i=EXIT ;
        } else if(strcmp(argv, "history" )==0){
                i=HISTORY ;
        } else if(strcmp(argv, "pwd" )==0){
                i=PWD ;
        } else if(strcmp(argv, "help" )==0){
                i=HELP ;
        } else if(strcmp(argv, "jobs" )==0){
                i=JOBS;
        }
        printf ("%s\n" ,help_doc-> usage[i]);
        printf ("%s\n" ,help_doc-> info [i]);
        return ;
}

char *if_exist(char *order) {
        char *all_path, *p, *path, *buffer;
        int len;
        all_path = getenv("PATH" );
        buffer = trim(all_path);
        len = strlen(all_path) + strlen(order);
```

```
        if ((path = ( char *) malloc(len * ( sizeof(char)))) == 0) {
                fprintf (stderr, "#error: can't malloc enough space for buffer\n" );
                return NULL;
        }
        p = strtok (buffer, ":" );
        while (p) {
                strcat(strcat(strcpy(path, p), "/" ), order);
                if (access(path, F_OK) == 0) {
                        return path;
                }
                p = strtok (NULL, ":" );
        }
        strcpy(path, order);
        if (access(path, F_OK) == 0)
                return path;
        return NULL;
}

char *trim (const char *str) {
        int i, j, k;
        char *order;
        if (str == NULL)
                return NULL;
        for (i = 0; i < strlen(str); i++)
                if (str[i] != ' ' && str[i] != ' ')
                        break;
        for (j = strlen(str) - 1; j > -1; j--)
                if (str[j] != ' ' && str[j] != ' ')
                        break;
        if (i <= j) {
                if ((order = ( char *) malloc((j - i + 2) * ( sizeof(char)))) == 0) {
                        fprintf (stderr, "#error: can't malloc enough space\n" );
                        return NULL;
                }
                for (k = 0; k < j - i + 1; k++)
                        order[k] = str[k + i];
                order[k] = '\0';
                return order;
        } else
        return NULL;
}

void handle_sigchld(int s) {
```

```
/* execute non-blocking waitpid, loop because we may only receive
* a single signal if multiple processes exit around the same time.
*/
// printf("recieve %d pid %d.\n",s,currentpid);
        int i=1;
        if(isCtrlz==0){
                for (; i < MAX_BACK_JOBS_NUM; i++) {
                        if (back_jobs[i] == NULL)
                                continue;
                        if (back_jobs[i]-> pid == currentpid) {
                                back_jobs[i]-> status = 0;
                                break;
                        }
                }
        } else{
                isCtrlz=0;
        }
        pid_t pid;
        while ((pid = waitpid (0, NULL, WNOHANG)) > 0) {
                int i = 1;
                for (; i < MAX_BACK_JOBS_NUM; i++) {
                        if (back_jobs[i] == NULL)
                                continue;
                        if (back_jobs[i]-> pid == pid) {
                                back_jobs[i]-> status = 0;
                                break ;
                        }
                }
        }
}

void handle_sigint(int s) {
        return;
}

void handle_sigstcp(int s) {
        int i = 1;
        isCtrlz = 1; // tell if it is a combination of Ctrl and Z
        int flag=0;
        printf ("\n");
        for (; i < MAX_BACK_JOBS_NUM; i++) {
                if (back_jobs[i] == NULL)
                        continue;
```

```
                if (back_jobs[i]-> pid == currentpid) {
                        back_jobs[i]-> status = 2;
                        flag=1;
                        break ;
                }
        }
        if(flag==0){
                back_jobs_ptr++;
                back_jobs[back_jobs_ptr] = ( struct BACK_JOBS *) malloc(
                sizeof(struct BACK_JOBS *));
                back_jobs[back_jobs_ptr]-> pid = currentpid;
                back_jobs[back_jobs_ptr]-> cmd = (char *) malloc(100);
                strcpy(back_jobs[back_jobs_ptr]-> cmd, "This is a stop process." );
                back_jobs[back_jobs_ptr]-> status = 2;
                printf ("[%d] %d %s\t\t\t\t%s\n" , back_jobs_ptr,
                back_jobs[back_jobs_ptr]-> pid ,
                st[back_jobs[back_jobs_ptr]-> status],
                back_jobs[back_jobs_ptr]-> cmd);
        }
        return;
}

void initWithHelpDoc (struct HELP_DOC *help_doc) {
        help_doc-> usage[EXIT] = "exit: exit" ;
        help_doc-> info[EXIT] = "Exit the shell." ;
        help_doc-> usage[CD] = "cd: cd [dir]" ;
        help_doc-> info[CD] = "\n\tThe default DIR is the value of the HOME shell variable." ;
        help_doc-> usage[HISTORY] = "history: history [-c] [-s num]" ;
        help_doc-> info[HISTORY] = "\n\tentry with a `*'. \n\t \n\t -s num\tsize of the history
        help_doc-> usage[PWD] = "pwd: pwd" ;
        help_doc-> info[PWD] = "Print the name of the current working directory." ;
        help_doc-> usage[HELP] = "help: help [pattern ...]" ;
        help_doc-> info[HELP] = "\n\t PATTERN Pattern specifiying a help topic" ;
        help_doc-> usage[JOBS] = "jobs: jobs" ;
        help_doc-> info[JOBS] = "\n\tLists the active jobs. JOBSPEC restricts output to that jol
}

void do_pwd() {
        char dirname[100];
        if(getcwd(dirname, 99) == NULL) {
                fprintf (stderr,"getcwd error\n" );
        }
        else {
```

```
                printf ("%s \n" ,dirname);
        }
}

char *command_generator (const char *text, int state) {
        char *name;
        static int list_index, len;
        if (!state) {
                list_index = 0;
                len = strlen (text);
        }
        while (name = commands[list_index]) {
                list_index++;
                if (strncmp(name, text, len) == 0)
                        return (strdup (name));
        }
        return ((char *)NULL);
}

char **command_completion(const char *text, int start, int end) {
        char **matches = NULL;
        if (start == 0)
                matches = rl_completion_matches (text, command_generator);
        return (matches);
}

void initialize_readline() {
        rl_attempted_completion_function = (CPPFunction*)command_completion;
}

int main (void){
        signal(SIGCHLD, handle_sigchld);
        signal(SIGINT,handle_sigint);
        signal(SIGSTCP,handle_sigstcp);
        char all_order[100], *order[MAXORD];
        int i, pid, status, number = 1, back, historyid = 1;
        char buf[80],prompt[100];
        char tmp[80];
        memset(tmp, 0, sizeof(tmp));
        char *username, *arg[MAXPARA];
        struct group *data;
        head = (NODE *) malloc(sizeof(NODE ));
        strcat(head->cmd, "intial" );
```

```
data = getgrgid(getgid());
username = data->gr_name;
strcat(tmp, "/home/" );
strcat(tmp, username);
//help doc
help_doc = ( struct HELP_DOC *) malloc(20* sizeof(struct HELP_DOC *));
initialize_readline();
initWithHelpDoc(help_doc);
printf ("-------------*Welcome*-------------\n" );
while (1) {
        getcwd(buf, sizeof(buf));
        if (strcmp(tmp, buf) == 0) {
                memset(buf, 0, sizeof(buf));
                buf[0] = '~';
                buf[1] = '\0';
        }
        sprintf (prompt, "%s:%s$" , username, buf);
        char *t;
        t=readline (prompt);
        sprintf (all_order, "%s" , t);
        if (all_order == NULL || trim(all_order) == NULL)
                continue;
        // store history
        NODE *next;
        next = (NODE *) malloc(sizeof(NODE ));
        next->id = historyid++;
        strcpy(next-> cmd, trim(all_order));
        next->next = head;
        head = next;
        add_history(all_order);
        back = is_back(trim(all_order));
        if (strcmp(trim(all_order), "exit" ) == 0) {
                int i = 1;
                int e = 1;
                for (; i < MAX_BACK_JOBS_NUM; i++) {
                        if (back_jobs[i] != NULL) {
                                if (back_jobs[i]-> status != 0) {
                                        e = 0;
                                        break;
                                }
                        }
                }
                if (e == 0) {
```

```
                fprintf (stdout, "Some jobs are undone, please stop them first.'
                fprintf (stdout, "Type \"jobs\" to see them.\n" );
                continue;
        }
        else {
                printf ("-------------*Goodbye*-------------\n" );
                exit(-1);
        }
}
if (trim(all_order)[0] == '!'&&trim(all_order)[1]== '-') {
        int i;
        sscanf(trim(all_order), "!-%d" , &i);
        if(i<0)
                fprintf (stderr,"!# # must be a negative number." );
        NODE * p;
        p=head;
        int j,flag=0;
        for(j=0;j<i;j++){
                p=p-> next;
                if(p==NULL){
                        fprintf (stderr,"!# # must be a less than history numbe
                        flag=1;
                }
        }
        if(flag==1)
                continue;
        strcpy(all_order,p-> cmd);
        historyid=historyid-1;
        head=head->next;
}
else if(trim(all_order)[0] == '!'){
        int i;
        sscanf(trim(all_order), "!%d" , &i);
        if(i<0) fprintf (stderr,"!# # must be a negative number." );
        i=historyid-i-1;
        NODE * p;
        p=head;
        int j,flag=0;
        for(j=0;j<i;j++){
                p=p-> next;
                if(p==NULL){
                        fprintf (stderr,"!# # must be a less than history numbe
                        flag=1;
```

```
                }
        }
        if(flag==1) continue;
        strcpy(all_order,p-> cmd);
        historyid=historyid-1;
        head=head->next;
}
if(trim(all_order)[0]== 'f' &&trim(all_order)[1]== 'g'){
        int i;
        sscanf(trim(all_order), "fg%d" , &i);
        if(back_jobs[i]-> status==0){
                continue;
        }
        currentpid=back_jobs[i]-> pid ;
        kill (back_jobs[i]-> pid ,SIGCONT);
        back_jobs[i]-> status=1;
        waitpid (back_jobs[i]-> pid, &status, WUNTRACED);
        continue;
}
if(trim(all_order)[0]== 'b'&&trim(all_order)[1]== 'g'){
        int i;
        sscanf(trim(all_order), "bg%d" , &i);
        if(back_jobs[i]-> status==0){
                printf ("It is already done.\n" );
                continue;
        }
        currentpid=0;
        kill (back_jobs[i]-> pid ,SIGCONT);
        back_jobs[i]-> status=1;
        continue;
}
arg[0] = trim( strtok (trim(all_order), " " ));
for (i = 1; (arg[i] = trim( strtok (NULL, " " ))) != NULL; i++);
if (strcmp(arg[0], "cd" ) == 0) {
        do_cd(arg);
        continue;
}
order[0] = trim( strtok (all_order, "&" ));
for (i = 1; (order[i] = trim( strtok (NULL, "&" ))) != NULL; i++)
        number++;
for (i = 0; i < number - 1; i++) {
        if (fork () == 0) {
                pipel(order[i]);
```

```
                        }
                }
                // not deamon
                if ((pid = fork ()) == 0) {
                        pipel(order[i]);
                }
                else if (back == 0){
                        currentpid=pid;
                        waitpid (pid, &status, WUNTRACED);
                }
                else {
                        back_jobs_ptr++;
                        back_jobs[back_jobs_ptr] = ( struct BACK_JOBS *) malloc(
                        sizeof(struct BACK_JOBS *));
                        back_jobs[back_jobs_ptr]-> pid = pid;
                        back_jobs[back_jobs_ptr]-> cmd = (char *) malloc(100);
                        strcpy(back_jobs[back_jobs_ptr]-> cmd, all_order);
                        back_jobs[back_jobs_ptr]-> status = 1;
                        printf ("[%d] %d %s\t\t\t\t%s\n" , back_jobs_ptr,back_jobs[back_jobs_ptr
                }
                number = 1;
        }
        return 1;
}
```