



東南大學  
SOUTHEAST UNIVERSITY

## 操作系统实践实验报告

姓名 李宗辉

学号 61522122

教师 张柏礼

2024 年 12 月 21 日

# 目录

<b>1</b>	<b>Shell 的实现</b>	<b>3</b>
1.1	Shell 的基本流程	3
1.2	声明的变量与函数	4
1.3	打印命令行提示符	5
1.4	获取命令行并且分割	5
1.5	基本命令的执行	6
1.5.1	外部命令执行	6
1.5.2	内部命令执行	7
1.6	管道功能的实现	9
1.6.1	单个管道的实现	9
1.6.2	多重管道的实现	11
1.7	重定向功能实现	12
1.8	main 函数	15
<b>2</b>	<b>Shell 的测试</b>	<b>16</b>
2.1	Shell 的基本形式	16
2.2	基本命令测试	16
2.2.1	外部命令测试	16
2.2.2	内部命令测试	16
2.3	管道功能测试	16
2.4	重定向功能测试	18
<b>3</b>	<b>实验疑难与体会</b>	<b>18</b>
3.1	实验疑难	18
3.2	实验体会	19
<b>4</b>	<b>完整源代码</b>	<b>19</b>
4.1	代码地址	19
4.2	源代码	19

# 1 Shell 的实现

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。具体要求如下：

- 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，包括：
  - 内部命令（例如 help 命令、exit 命令等）
  - 外部命令（常见的 ls、cp 等，以及其他磁盘上的可执行程序 HelloWorld 等）
  - 无效命令（不是上述二种命令）
- 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。
- 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
- 将上述步骤直接合并完成

## 1.1 Shell 的基本流程

Shell 的基本流程主要是在一个 while(true) 的循环中，首先打印出仿照终端的命令行提示符，然后不断地接受用户输入的命令并进行分解，接着判断命令是否有效，如果有效，则根据命令的类型，执行相应的操作。如图 1 所示。

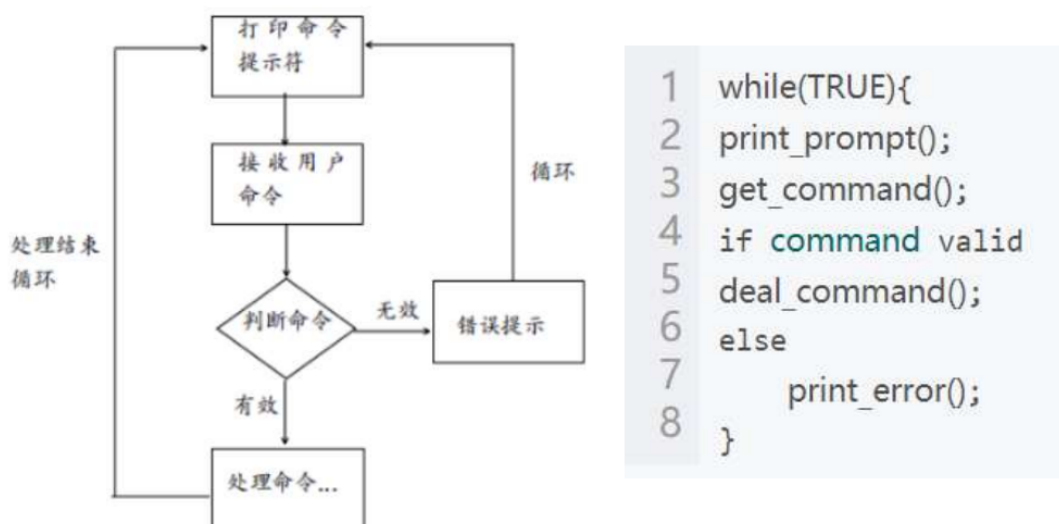


图 1: Shell 的基本流程

## 1.2 声明的变量与函数

本实验定义的常量和全局变量包括：

```

1 #define LINE_SIZE 1024 // 命令行最大长度
2 #define ARGV_SIZE 32   // 命令行参数个数最大个数
3 #define EXIT_CODE 55   // 设定的退出代码
4
5 int lastcode = 0; // 最后执行命令的返回值
6 char pwd[LINE_SIZE]; // 当前工作目录
7 char hostname[64]; // 主机名
8 char commandline[LINE_SIZE]; // 存放命令行
9 char *argv[ARGV_SIZE]; // 存放命令行参数，字符指针数组
10 char *internalCommands[] = {"exit", "cd", "help", "echo", NULL}; // 内部命令列表

```

本实验使用的函数包括：

```

1 const char* getusername(); // 获取当前用户名
2 void get_host_name(); // 获取主机名
3 void getpwd(); // 获取当前工作目录
4 void print_prompt(); // 打印命令行提示符
5 void get_command(char *cline, int size); // 从标准输入获取命令行
6 int splitCline_getArgcv(char cline[], char *_argv[]); // 分割命令行，存储命令参数并且返回参数个数
7 void runExternalCommands(char *_argv[]); // 执行外部命令
8 bool isInternalCommands(char *_argv[]); // 判断是否为内部命令
9 void runInternalCommands(char *_argv[], int argc); // 执行内部命令
10 void handle_basic_commands(char *_argv[], int argc); // 处理基本命令（外部和内部命令）
11 void handle_mayRedir_commands(char commands[]); // 处理可能带重定向的命令（即基本命令 + 重定向）
12 void handle_single_pipe(char commands[]); // 处理带单个管道命令
13 void handle_multiple_pipe(char commands[]); // 处理带多个管道命令
14 int count_pipe(const char *commands); // 统计获取命令行中管道符的个数
15 bool isRedir(char *commands); // 判断传入的命令是否含有重定向符号
16 void handle_redirection(char *commands); // 处理带有重定向符号的命令

```

### 1.3 打印命令行提示符

命令行提示符主要包括 1: 用户名; 2: “@” 符号; 3: 主机名; 4: “:” 符号; 5: 当前目录; 6: 用户权限符号 (通常均是 \$, 用 “#” 来表示管理员) 对于用户名, 可以使用环境变量 USER 获取, 而主机名和当前工作目录可以使用标准库函数 gethostname 和 getcwd 获取。其他符号可以用字符表示。

为了避免调试时和原本的 shell 提示符混淆, 实现的 shell 用红色打印出来。这需要使⽤ ANSI 转义序列 \033[31m, 表示后续的输出内容为红色。在提示符打印完成后再通过序列 \033[0m 恢复默认颜色。

具体实现代码如下:

```
// 获取当前用户名
const char* getusername() {
    return getenv("USER");
}

// 获取主机名
void get_host_name() {
    gethostname(hostname, sizeof(hostname));
}

// 获取当前工作目录
void getpwd() {
    getcwd(pwd, sizeof(pwd));
}

// 打印命令行提示符
void print_prompt() {
    get_host_name();
    getpwd();
    printf("\033[31m%s@%s %s$ \033[0m", getusername(), hostname, pwd);
}
```

### 1.4 获取命令行并且分割

对于用户输入的命令, Shell 需要接收并且分割出命令名和参数。

使用 fgets 函数获取标准输入, 将输入最后的换行符设置为 '\0', 这样就得到了命令字符串。

```
1 // 从标准输入获取命令行
2 void get_command(char *cline, int size) {
```

```

3   if (fgets(cline, size, stdin) != NULL) { //使用 fgets 函数从标准输入
    获取命令行输入
4       cline[strcspn(cline, "\n")] = '\0'; // 使用 strcspn 函数找到换行
    符并且移除
5   } else {
6       if (feof(stdin)) {
7           printf("EOF reached\n");
8       } else {
9           perror("fgets error");
10      }
11      exit(EXIT_CODE);
12  }
13  }

```

对于处理后的命令字符串，调用 `strtok` 函数进行分割，得到命令名和参数，其中 `argv[0]` 为命令。需要注意的是，`argv` 的最后一个参数要设置为 `NULL`，以便于 `execvp` 函数使用需要。

```

1   // 分割命令行，存储命令参数并且返回参数个数
2   int splitCline_getArgcv(char cline[], char *_argv[]) {
3       int i = 0;
4       char *token = strtok(cline, "t"); // 命令名称
5       while (token != NULL) {
6           _argv[i++] = token;           // 保存命令或者其参数
7           token = strtok(NULL, "t");   // 获取下一个 token
8       }
9       _argv[i] = NULL; // 确保以 NULL 结尾，满足 execvp 要求
10      return i;        // 返回参数个数
11  }

```

## 1.5 基本命令的执行

基本命令主要包括外部命令和内部命令，不包括管道和重定向符号。基本命令的执行是 Shell 的核心功能。

### 1.5.1 外部命令执行

外部命令不是由 Shell 直接执行的命令，而是作为独立的程序运行的。这些命令通常位于系统 `PATH` 环境变量指定的目录中，比如 `/bin`、`/usr/bin` 等目录下。为了主进

程的稳定和安全，执行外部命令通常 fork 一个子进程，然后在子进程中调用 `execvp` 函数，它会在系统 `PATH` 环境变量指定的目录查找外部命令的可执行文件，并执行相应的程序。

```

1 // 利用子进程去执行外部命令
2 void runExternalCommands(char *_argv[]) {
3     pid_t pid = fork();
4
5     if (pid < 0) { // fork 失败
6         perror("Fork failed");
7         return;
8     }
9     if (pid == 0) { // 子进程执行命令
10        execvp(_argv[0], _argv); //execvp 会从系统的 PATH 环境变量指定的
        目录中查找 _argv[0] 命令并执行
11        fprintf(stderr, "execvp error: Command not found or failed to execute:
        %s\n", _argv[0]);
12        exit(EXIT_CODE); // 退出子进程
13    } else { // 父进程等待子进程结束
14        int status;
15        if (waitpid(pid, &status, 0) == -1) { // 等待子进程结束
16            perror("Waitpid failed");
17        } else if (WIFEXITED(status)) { // 子进程正常退出
18            lastcode = WEXITSTATUS(status);
19        } else if (WIFSIGNALED(status)) { // 子进程因信号终止
20            fprintf(stderr, "Process terminated by signal %d\n", WTERMSIG(
            status));
21            lastcode = EXIT_CODE; // 设为错误代码
22        }
23    }
24 }

```

### 1.5.2 内部命令执行

内部命令是指 Shell 自身提供的命令，如 `help`、`exit` 等。内部命令的执行不需要 fork 子进程，而是直接调用相应的函数。本实验实现的内部命令有：

- `exit`: 退出 Shell 程序
- `cd`: 改变当前目录

- echo: 输出字符串
- help: 输出内部指令执行的帮助信息

内部指令执行的实现思路是，根据命令的第一个参数判断是哪种内部指令，然后执行相应的操作。

```

1 // 内部命令处理
2 void runInternalCommands(char *_argv[], int argc){
3     if (argc == 0) {
4         fprintf(stderr, "No Internal commands specified\n");
5         return;
6     }
7
8     else if (strcmp(_argv[0], "exit") == 0) { // exit 命令
9         exit(0);
10    }
11
12    else if (strcmp(_argv[0], "cd") == 0) { // cd 命令
13        if (argc == 1) { // 只有一个 cd 命令，未指定目录，切换到用户主目录
14            chdir(getenv("HOME"));
15        } else {
16            if (chdir(_argv[1]) != 0) { // 切换到指定目录 argv[1]
17                perror("chdir error");
18            }
19        }
20        getpwd(); // 更新当前目录
21        return;
22    }
23    else if (strcmp(_argv[0], "echo") == 0){ // echo 命令
24        if (argc == 1) { // 只有一个 echo 命令，打印换行符
25            printf("\n");
26        } else{ // 打印跟在后面的 string，即使中间带了空格，会被存在 argv
中。
27            for (int i = 1; _argv[i] != NULL && i < argc; i++) {
28                printf("%s ", _argv[i]); // 中间加空格
29            }
30            printf("\n");
31        }
32        return;
33    }
34    else if (strcmp(_argv[0], "help") == 0){ // help 命令，打印内部命令列表

```



```

35     printf("Internal commands:\n");
36     printf("  exit\n");
37     printf("  cd [directory]\n");
38     printf("  help\n");
39     printf("  echo [string]\n");
40     return;
41 }
42 }

```

## 1.6 管道功能的实现

管道功能是指在命令之间建立管道，使得命令的输出能够被其他命令作为输入，是 Shell 的重要特性之一。管道分为单个管道和多重管道，两者实现的逻辑有所差别。此外，管道的输入、输出操作是互斥的，不能同时进行。

### 1.6.1 单个管道的实现

要处理管道，首先要找到的就是管道“|”符号的位置。对于传入的命令行字符串，可以通过 strchr 函数找到“|”符号的位置。找到位置后，需要修改“|”为‘\0’，也就是通过字符串结束符将原本的命令分成左右两个子命令。使用 pipe(pipefd) 函数创建一个管道，pipefd 数组用来保存管道的两个文件描述符，其中 pipefd[0] 是读取端，pipefd[1] 是写入端。

对于左边的命令，调用 fork 函数创建一个子进程执行管道功能。由于左边命令的结果是传给右边命令作为输入的，因此在子进程中需要关闭 pipefd[0] 读端，接着使用 dup2 函数将 pipefd[1] 写端文件描述符复制给标准输出，这样左边命令的输出就会被写入管道，然后关闭写端，最后调用函数处理左边命令。

对于右边的命令，同样调用 fork 函数创建一个子进程执行管道功能。由于右边命令的输入是左边命令的输出，因此在子进程中需要关闭 pipefd[1] 写端，接着使用 dup2 函数将 pipefd[0] 读端文件描述符复制给标准输入，这样右边命令的输入就是从管道读取，然后关闭读端，最后调用函数处理右边命令。

父进程关闭管道的读写端并且等待子进程结束。

```

1  void handle_single_pipe(char commands[]){
2      int pipefd[2]; // 管道文件描述符
3      pid_t pid;
4      char *pipe_pos = strchr(commands, '|');
5      if(pipe_pos){
6          *pipe_pos = '\0'; // 将管道符号替换为字符串结束符，变成左右两个字

```

符串

```
7      char * left_command = commands;
8      char * right_command = pipe_pos + 1;
9
10     if(pipe(pipefd) < 0){
11         perror("pipr error");
12         exit(EXIT_CODE);
13     }
14     // 创建第一个子进程, 执行左边命令
15     if ((pid = fork()) == 0) {
16         // 子进程执行左边命令
17         close(pipefd[0]); // 关闭读端
18         dup2(pipefd[1], STDOUT_FILENO); // 将输出定向到管道的写入端
19         close(pipefd[1]);
20
21         handle_mayRedir_commands(left_command); // 处理左边命令
22         exit(EXIT_SUCCESS); // 退出子进程
23     } else if (pid < 0) {
24         perror("fork");
25         exit(EXIT_CODE);
26     }
27     // 创建第二个子进程, 执行右边命令
28     if ((pid = fork()) == 0) {
29         // 子进程执行右边命令
30         close(pipefd[1]); // 关闭写端
31         dup2(pipefd[0], STDIN_FILENO); // 将输入定向到管道的读取端
32         close(pipefd[0]);
33
34         handle_mayRedir_commands(right_command); // 处理右边命令
35         exit(EXIT_SUCCESS); // 退出子进程
36     } else if (pid < 0) {
37         perror("fork");
38         exit(EXIT_CODE);
39     }
40     close(pipefd[0]);
41     close(pipefd[1]);
42     // 等待所有子进程结束
43     while (wait(NULL) > 0);
44 }
45 }
```

### 1.6.2 多重管道的实现

多重管道大体的实现思路和单个管道相同，但略有区别：

在找到第一个“|”符号的位置后，将其替换为‘\0’，分割左右命令；对于左边命令，创建一个子进程执行，和单个管道的处理逻辑相同；对于右边命令，同样创建一个子进程执行，但是由于可能仍然包含“|”符号，因此需要递归地处理右边命令，也就是递归式调用本函数处理右边命令，直到右边命令没有“|”符号为止。

```

1  void handle_multiple_pipe(char commands[]){ //多重管道情况
2      int pipefd[2]; // 管道文件描述符
3      pid_t pid;
4      char *pipe_pos = strchr(commands, '|');
5
6      if(pipe_pos){ // 管道符号存在
7          *pipe_pos = '\0'; // 将管道符号替换为字符串结束符，变成左右两个字
            符串
8          char * left_command = commands;
9          char * right_command = pipe_pos + 1;
10         if(pipe(pipefd) < 0){
11             perror("pipr error");
12             exit(EXIT_CODE);
13         }
14         // 创建第一个子进程，执行左边命令
15         if ((pid = fork()) == 0) {
16             // 子进程执行左边命令
17             close(pipefd[0]); // 关闭读端
18             dup2(pipefd[1], STDOUT_FILENO); // 将输出定向到管道的写入端
19             close(pipefd[1]);
20             handle_mayRedir_commands(left_command); // 处理左边命令
21             exit(EXIT_SUCCESS); // 退出子进程!!! 非常重要，补充到报告中!
22         } else if (pid < 0) {
23             perror("fork");
24             exit(EXIT_CODE);
25         }
26         // 创建第二个子进程，执行右边命令
27         if((pid =fork())==0){
28             // 子进程执行右边命令
29             close(pipefd[1]); // 关闭写端

```

```

30         dup2(pipefd[0], STDIN_FILENO); // 将输入定向到管道的读取端
31         close(pipefd[0]);
32         // 递归处理右边命令
33         if(strchr(right_command, '|') != NULL){ // 中止条件
34             handle_multiple_pipe(right_command);
35         }
36         else{ // 右边命令没有管道的情况
37             handle_mayRedir_commands(right_command); // 处理右边命
38         }
39         exit(EXIT_SUCCESS); // 退出子进程
40     }
41     else if (pid < 0) {
42         perror("fork");
43         exit(EXIT_CODE);
44     }
45     close(pipefd[0]);
46     close(pipefd[1]);
47     while(wait(NULL)>0); // 等待所有子进程结束
48 }
49 }

```

## 1.7 重定向功能实现

实现重定向，需要考虑重定向命令有哪些形式。

首先要明白，输入重定向”<”和输出重定向”>”之间除了管道符号外，插入其他命令符号都是不合法的。而通过前面的管道实现逻辑可以发现，带管道的命令会将大命令分成好多个子命令，而这些子命令是不带管道符号的，也就是处理重定向的时候不用考虑管道的影响。因此实际上需要考虑的重定向命令只有以下四种形式：

- command ”>” outfile
- command ”<” infile
- command ”<” infile ”>” outfile
- command ”>” outfile ”<” infile

而实现重定向的逻辑主要是：

对于给定命令行，首先遍历判断重定向符号”<”和”>”的数量，如果任一符号数量超过 1，则不合法，报错；在遍历过程中，如果检测到输入重定向符号”<”，以只读

方式（O\_RDONLY）打开该符号后面的文件，并将其文件描述符 fd 复制给标准输入，这样命令执行的输入就来自于该文件；如果检测到输出重定向符号“>”，以写入方式（O\_WRONLY）、覆盖方式（O\_TRUNC）、创建方式（O\_CREAT）和读写权限（0666）打开该符号后面的文件，并将其文件描述符 fd 复制给标准输出，这样命令执行的输出就写入该文件；

而在上述遍历中，一旦检测到重定向符号，就修改其为'\0'，这样就分割出了左侧需要运行的命令。在遍历命令行完成后，创建子进程执行左侧命令。

由于之前对标准输入和标准输出修改是在主进程完成，因此最后还需要重新还原主进程的标准输入和输出，否则标准输入输出还是定向在之前的重定向文件上。

```

1  void handle_redirection(char *commands) {
2      int inNum = 0, outNum = 0; // < 和 > 的个数
3      pid_t pid;
4      //保存原始标准输入和标准输出
5      int saved_stdin = dup(STDIN_FILENO);
6      int saved_stdout = dup(STDOUT_FILENO);
7
8      char* commands_temp = malloc(sizeof(commands)); //避免设置重定向
      为'\0' 时字符串遍历出错，采用 temp 保存原始命令行进行遍历
9      strcpy(commands_temp, commands);
10
11     for(int i = 0; i < strlen(commands_temp); i++){
12         if(commands_temp[i] == '>'){ //处理输出重定向
13             outNum++;
14             if(outNum > 1){
15                 perror("Too many output redirections");
16                 exit(EXIT_CODE);
17             }
18             char *outredir_pos = &commands[i];
19             char *out_file = strtok(outredir_pos + 1, " \t"); //输出文件，
      并且使用 strtok 去除前面的空格，否则路径不正确
20
21             int fd = open(out_file, O_WRONLY | O_CREAT | O_TRUNC, 0666)
22             ;
23             if(fd < 0){
24                 perror("Output redirection error");
25                 exit(EXIT_CODE);
26             }
27             dup2(fd, STDOUT_FILENO); //复制 fd 到 STDOUT_FILENO（标准输
      出），即将标准输出写入文件 fd。
28             close(fd);

```

```

28
29         *outredir_pos = '\0'; //设置为字符串结束符号会影响到循环，所以
        循环用的是 temp
30     }
31     else if(commands_temp[i]=='<'){ //处理输入重定向
32         inNum++;
33         if(inNum > 1){
34             perror("Too many input redirections");
35             exit(EXIT_CODE);
36         }
37         char *inredir_pos = &commands[i];
38         char *in_file = strtok(inredir_pos + 1, " \t"); //输入文件
39
40         int fd = open(in_file, O_RDONLY); //只读模式打开文件
41         if(fd<0){
42             perror("Input redirection error");
43             exit(EXIT_CODE);
44         }
45         dup2(fd,STDIN_FILENO);
46         close(fd);
47
48         *inredir_pos = '\0';
49     }
50 }
51 //处理最左边的命令
52 char *left_command = commands;
53 if(pid=fork() == 0){
54     char *argv_left[ARGC_SIZE]; // 左边命令
55     int argc_left = splitCline_getArgcv(left_command, argv_left);
56     // 分割左边命令，存储 argv 并返回 argc。
57     handle_basic_commands(argv_left, argc_left); // 处理左边命令
58     exit(EXIT_SUCCESS); // 退出子进程
59 }
60 else if(pid < 0){
61     perror("fork");
62     exit(EXIT_CODE);
63 }
64 else{
65     wait(NULL); // 等待子进程结束
66 }

```

```

66     dup2(saved_stdout, STDOUT_FILENO); //重新还原主进程的标准输入和输出
67     close(saved_stdout);
68     dup2(saved_stdin, STDIN_FILENO);
69     close(saved_stdin);
70 }

```

## 1.8 main 函数

在主循环中，处理的主要逻辑如下：

首先打印命令行提示符，接着获取用户输入的命令行，由于分割会对原命令有影响，因此预先存储一个命令行的 temp 变量。然后分割命令行得到命令和参数，再根据命令行中管道的数量进入不同的处理函数中。

```

1  int main() {
2      while (1) {
3          print_prompt(); // 打印提示符
4          get_command(commandline, sizeof(commandline)); // 获取用户输入
5
6          char *commandline_copy = malloc(sizeof(commandline));
7          strcpy(commandline_copy, commandline);
8          int argc = splitCline_getArgcv(commandline, argv); // 分割命令，
           存储 agrv 并返回 argc。
9          if (argc == 0) continue; // 空命令直接跳过该轮循环
10
11         int pipe_count = count_pipe(commandline_copy); // 管道的个数
12         if(pipe_count == 1){
13             handle_single_pipe(commandline_copy); // 单管道情况
14         }
15         else if(pipe_count > 1){
16             handle_multiple_pipe(commandline_copy); // 多管道情况
17         }
18         else{
19             handle_mayRedir_commands(commandline_copy); // 处理内可能带
           有重定向的命令
20         }
21     }
22     return 0;
23 }

```

## 2 Shell 的测试

### 2.1 Shell 的基本形式

编译 c 文件后在终端运行，基本形式如图 2 所示。可见命令行提示符是正确的。

```
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ls
OSshell OSshell.c test6.txt test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ./OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$
```

图 2: Shell

### 2.2 基本命令测试

#### 2.2.1 外部命令测试

外部命令主要测试了 ls、pwd、cat，结果如图 3 所示，结果都正确。

```
mercurystraw@mercurystraw-VMware-Virtual-Platform: ~/Desktop/OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ls
OSshell OSshell.c test6.txt test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ./OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ ls
OSshell OSshell.c test6.txt test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ pwd
/home/mercurystraw/Desktop/OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ cat test.txt
输入重定向数量：0输出重定向数量：1redir_left_command: echo hellowrold
left command argc: 2
left command: echo hellowrold
hellowrold
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ ^C
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ cat test.txt
输入重定向数量：0输出重定向数量：1redir_left_command: echo hellowrold
left command argc: 2
left command: echo hellowrold
hellowrold
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$
```

图 3: 外部命令执行

#### 2.2.2 内部命令测试

内部命令共实验设置的四种 cd, echo, help, exit。测试结果如图 4 所示，都是符合预期的。

### 2.3 管道功能测试

单个和多个管道的测试结果如图 5 所示，结果和原终端的命令结果一样，说明功能正确。



```
mercurystraw@mercurystraw-VMware-Virtual-Platform: ~/Desktop/OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ./OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ cd ..
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop$ cd ..
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw$ echo helloworld
helloworld
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw$ echo hello world !!
hello world !!
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw$ help
Internal commands:
  exit
  cd [directory]
  help
  echo [string]
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw$ exit
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$
```

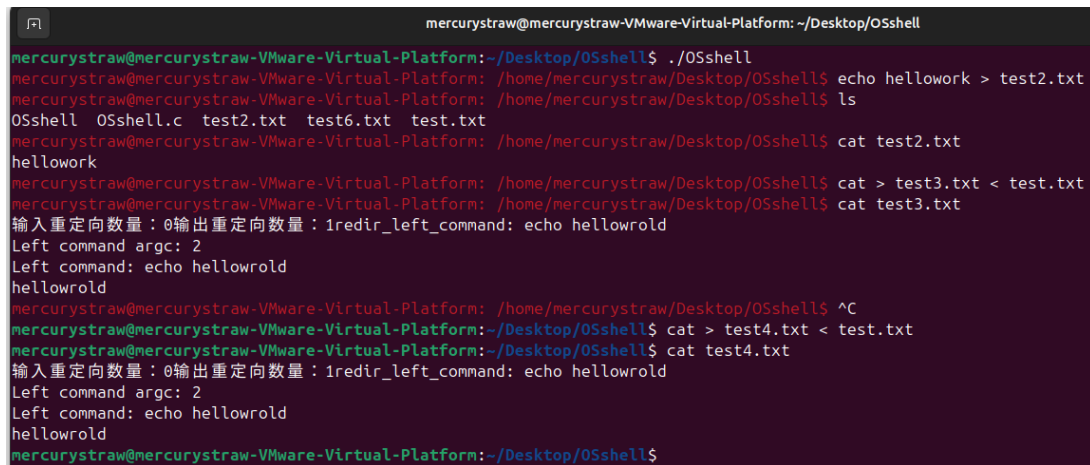
图 4: 内部命令执行

```
mercurystraw@mercurystraw-VMware-Virtual-Platform: ~/Desktop/OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ls
OSshell OSshell.c test6.txt test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ls -l
total 52
-rwxrwxr-x 1 mercurystraw mercurystraw 22296 Dec 20 21:13 OSshell
-rw-rw-r-- 1 mercurystraw mercurystraw 16470 Dec 20 21:13 OSshell.c
-rw-rw-r-- 1 mercurystraw mercurystraw 130 Dec 20 18:42 test6.txt
-rw-rw-r-- 1 mercurystraw mercurystraw 151 Dec 20 18:25 test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ls -l | wc
 5      38      278
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ls -l | grep txt | wc
 2       18      135
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ./OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ ls -l | wc
 5      38      278
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ ls -l | grep txt | wc
 2       18      135
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$
```

图 5: 管道功能测试

## 2.4 重定向功能测试

重定向功能测试结果如图 6 所示，无论是单个重定向符号还是两个重定向符号，结果和原终端的命令结果一样，说明功能正确。



```
mercurystraw@mercurystraw-VMware-Virtual-Platform: ~/Desktop/OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ ./OSshell
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ echo helloworld > test2.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ ls
OSshell OSshell.c test2.txt test6.txt test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ cat test2.txt
helloworld
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ cat > test3.txt < test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ cat test3.txt
输入重定向数量: 0输出重定向数量: 1redir_left_command: echo helloworld
Left command argc: 2
Left command: echo helloworld
helloworld
mercurystraw@mercurystraw-VMware-Virtual-Platform: /home/mercurystraw/Desktop/OSshell$ ^C
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ cat > test4.txt < test.txt
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$ cat test4.txt
输入重定向数量: 0输出重定向数量: 1redir_left_command: echo helloworld
Left command argc: 2
Left command: echo helloworld
helloworld
mercurystraw@mercurystraw-VMware-Virtual-Platform:~/Desktop/OSshell$
```

图 6: 重定向功能测试

## 3 实验疑难与体会

### 3.1 实验疑难

在实验中我遇到的主要困难有以下几点，都是通过不断调试和查阅资料解决的。

1. 字符串相关函数的使用不够熟练，如 `strchr`、`strtok`、`strcspn` 等。尤其是 `strtok` 函数，其在处理命令行时，会用 `'\0'` 替换选定的分隔符（如代码中的 `"\t"`）。使用 `strtok` 函数分割确实能得到命令名和参数，但是会破坏了原来的命令行。因此需要预先保存一个命令行的 `temp` 变量备用。

2. 文件描述符。如 `dup`、`dup2`、`close` 等。尤其是 `dup2` 函数，其作用是复制文件描述符，但是需要注意，复制的文件描述符需要在关闭之前，否则会造成文件描述符泄露。

3. 子进程处理。在子进程完成后需要设置退出状态 `exit(EXIT_SUCCESS)`，否则父进程会一直等待子进程，造成僵尸进程。此外，还要注意文件描述符重定向是在子进程还是主进程上修改的，若在主进程，则记得需要重新还原。

4. 没有预先做好代码架构，导致实验过程中代码写的较乱，冗余重复也较多。

本实验参考资料：

- <https://blog.csdn.net/dxyt2002/article/details/129800496>
- [https://www.cnblogs.com/hellovenus/p/os\\_shell\\_implementation.html](https://www.cnblogs.com/hellovenus/p/os_shell_implementation.html)
- [https://blog.csdn.net/weixin\\_43679657/article/details/109482700](https://blog.csdn.net/weixin_43679657/article/details/109482700)

- <https://blog.csdn.net/OCTODOG/article/details/70942194>
- <https://blog.csdn.net/feng964497595/article/details/80297318>

## 3.2 实验体会

实现一个自己的 shell，并且包含管道和重定向功能，是一项很有意思的任务。在实现过程中，我对 shell 的运行原理有了更深入的理解，同时通过自己不断调试和解决 shell 中的各种问题，我增强了自己解决问题的能力，并且对操作系统的底层机制有了更深刻的理解。shell 作为用户与操作系统之间的接口，不仅仅是一个简单的命令行工具，里面更是大有门路。

# 4 完整源代码

## 4.1 代码地址

- Github: <https://github.com/mercurystraw/Shell>

## 4.2 源代码

```
//OSshell.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdbool.h>

#define LINE_SIZE 1024 // 命令行最大长度
#define ARGV_SIZE 32 // 命令行参数个数最大个数
#define EXIT_CODE 55 // 设定的退出代码

int lastcode = 0; // 最后执行命令的返回值
char pwd[LINE_SIZE]; // 当前工作目录
char hostname[64]; // 主机名
char commandline[LINE_SIZE]; // 存放命令行
```

```

char *argv[ARGC_SIZE];           // 存放命令行参数,字符指针数组
char *internalCommands[] = {"exit", "cd", "help", "echo", NULL}; // 内部命令列表

// 函数声明, 避免后面函数相互调用受到定义的先后顺序 影响
const char* getusername();           // 获取当前用户名
void get_host_name();               // 获取主机名
void getpwd();                      // 获取当前工作目录
void print_prompt();                // 打印命令行提示符
void get_command(char *cline, int size); // 从标准输入获取命令行
int splitCline_getArgcv(char cline[], char *_argv[]); // 分割命令行, 存储命令参数并
void runExternalCommands(char *_argv[]); // 执行外部命令
bool isInternalCommands(char *_argv[]); // 判断是否为内部命令
void runInternalCommands(char *_argv[], int argc); // 执行内部命令
void handle_basic_commands(char *_argv[], int argc); // 处理基本命令 (外部和内部命令)
void handle_mayRedir_commands(char commands[]); // 处理可能带重定向的命令 (即
void handle_single_pipe(char commands[]); // 处理带单个管道命令
void handle_multiple_pipe(char commands[]); // 处理带多个管道命令
int count_pipe(const char *commands); // 统计获取命令行中管道符的个
bool isRedir(char *commands); // 判断传入的命令是否含有重定
void handle_redirection(char *commands); // 处理带有重定向符号的命令

// 获取当前用户名
const char* getusername() {
    return getenv("USER");
}

// 获取主机名
void get_host_name() {
    gethostname(hostname, sizeof(hostname));
}

// 获取当前工作目录
void getpwd() {
    getcwd(pwd, sizeof(pwd));
}

```

```
// 打印命令行提示符
void print_prompt() {
    get_host_name();
    getpwd();
    printf("\033[31m%s@%s %s$ \033[0m", getusername(), hostname, pwd);
}

// 从标准输入获取命令行
void get_command(char *cline, int size) {
    if (fgets(cline, size, stdin) != NULL) { //使用fgets函数从标准输入获取命令行输入
        cline[strcspn(cline, "\n")] = '\0'; // 使用strcspn函数找到换行符并且移除
    } else {
        if (feof(stdin)) {
            printf("EOF reached\n");
        } else {
            perror("fgets error");
        }
        exit(EXIT_CODE);
    }
}

// 分割命令行，存储命令参数并且返回参数个数
int splitCline_getArgcv(char cline[], char *_argv[]) {
    int i = 0;
    char *token = strtok(cline, " \t"); // 命令名称
    while (token != NULL) {
        _argv[i++] = token;           // 保存命令或者其参数
        token = strtok(NULL, " \t"); // 获取下一个 token
    }
    _argv[i] = NULL; // 确保以 NULL 结尾，满足 execvp 要求
    return i;        // 返回参数个数
}

// 利用子进程去执行外部命令
void runExternalCommands(char *_argv[]) {
    pid_t pid = fork();
```

```

    if (pid < 0) { // fork 失败
        perror("Fork failed");
        return;
    }
    if (pid == 0) { // 子进程执行命令
        //printf("内部指令运行, Child process executing command %s\n", _argv[0]);
        execvp(_argv[0], _argv); //execvp 会从系统的 PATH 环境变量指定的目录中查找 _a
        fprintf(stderr, "execvp error: Command not found or failed to execute: %s\n",
            exit(EXIT_CODE); // 退出子进程
    } else { // 父进程等待子进程结束
        int status;
        if (waitpid(pid, &status, 0) == -1) { // 等待子进程结束
            perror("Waitpid failed");
        } else if (WIFEXITED(status)) { // 子进程正常退出
            lastcode = WEXITSTATUS(status);
        } else if (WIFSIGNALED(status)) { // 子进程因信号终止
            fprintf(stderr, "Process terminated by signal %d\n", WTERMSIG(status));
            lastcode = EXIT_CODE; // 设为错误代码
        }
    }
}

bool isInternalCommands(char *_argv[]) {
    for(int i=0; internalCommands[i] != NULL; i++){
        if(strcmp(_argv[0], internalCommands[i]) == 0) return true;
    }
    return false;
}

// 内部命令处理
void runInternalCommands(char *_argv[], int argc){
    if (argc == 0) {
        fprintf(stderr, "No Internal commands specified\n");
        return;
    }

    else if (strcmp(_argv[0], "exit") == 0) { // exit命令
        exit(0);
    }
}

```

```

    }

    else if(strcmp(_argv[0], "cd") == 0) { // cd命令
        if (argc == 1) { // 只有一个cd命令，未指定目录，切换到用户主目录
            chdir(getenv("HOME"));
        } else {
            if(chdir(_argv[1]) != 0) { // 切换到指定目录 argv[1]
                perror("chdir error");
            }
        }
        getpwd(); // 更新当前目录
        return;
    }

    else if(strcmp(_argv[0], "echo") == 0){ // echo命令
        if (argc == 1) { // 只有一个echo命令，打印换行符
            printf("\n");
        } else{ // 打印跟在后面的string，即使中间带了空格，会被存在argv中。
            for (int i = 1; _argv[i] != NULL && i < argc; i++) {
                printf("%s ", _argv[i]); // 中间加空格
            }
            printf("\n");
        }
        return;
    }

    else if(strcmp(_argv[0], "help")== 0){ // help命令，打印内部命令列表
        printf("Internal commands:\n");
        printf("  exit\n");
        printf("  cd [directory]\n");
        printf("  help\n");
        printf("  echo [string]\n");
        return;
    }
}

void handle_basic_commands(char *_argv[], int argc){ //处理不带管道和重定向的基本命令
    if(isInternalCommands(_argv)){ // 内部命令执行

```

```

        runInternalCommands(_argv, argc);
    }
    else{
        runExternalCommands(_argv);        // 外部命令执行
    }
}

void handle_mayRedir_commands(char commands[]){ //处理可能有重定向符号命令的函数, han
    if(isRedir(commands)){ // 重定向符号存在
        handle_redirection(commands); //处理重定向
    }
    else{
        char *argv[ARGC_SIZE];
        int argc = splitCline_getArgcv(commands, argv); // 分割命令行, 存储参数
        handle_basic_commands(argv, argc); // 处理基本命令
    }
}

void handle_single_pipe(char commands[]){
    //strtok 函数在处理字符串时, 会用 \0 替换分隔符 (在您代码中的 " \t" 和 |),
    //这意味着当您查找 | 符号时, strtok 会在遇到第一个分隔符 (如下空格) 时将其替换为
    //so commandline became ls instead of ls | wc -l

    //printf("handle_single_pipe\n");
    //printf("commands: %s\n", commands);

    int pipefd[2]; // 管道文件描述符
    pid_t pid;
    char *pipe_pos = strchr(commands, '|');
    //printf("pipe_pos: %d\n", *pipe_pos);

    if(pipe_pos){
        //printf("pipe found\n");
        *pipe_pos = '\0'; // 将管道符号替换为字符串结束符, 变成左右两个字符串
        char * left_command = commands;
        char * right_command = pipe_pos + 1;
    }
}

```



```
//printf("left_command: %s\n", left_command);
//printf("right_command: %s\n", right_command);

if(pipe(pipefd) < 0){
    perror("pipe error");
    exit(EXIT_CODE);
}

// 创建第一个子进程，执行左边命令
if ((pid = fork()) == 0) {
    // 子进程执行左边命令
    //printf("child process execute left command\n");
    close(pipefd[0]); // 关闭读端
    dup2(pipefd[1], STDOUT_FILENO); // 将管道写端文件描述符复制给标准输出
    close(pipefd[1]);

    handle_mayRedir_commands(left_command); // 处理左边命令
    exit(EXIT_SUCCESS); // 退出子进程 !!! 非常重要，补充到报告中!
} else if (pid < 0) {
    perror("fork");
    exit(EXIT_CODE);
}

// 创建第二个子进程，执行右边命令
if ((pid = fork()) == 0) {
    // 子进程执行右边命令
    //printf("child process execute right command\n");
    close(pipefd[1]); // 关闭写端
    dup2(pipefd[0], STDIN_FILENO); // 将管道读端文件描述符复制给标准输入
    close(pipefd[0]);

    handle_mayRedir_commands(right_command); // 处理右边命令
    exit(EXIT_SUCCESS); // 退出子进程
} else if (pid < 0) {
    perror("fork");
    exit(EXIT_CODE);
}
```

```
        close(pipefd[0]);
        close(pipefd[1]);
        // 等待所有子进程结束
        while (wait(NULL) > 0);
    }
}

void handle_multiple_pipe(char commands[]){ //多重管道情况
    //printf("Handle_multiple_pipe\n");
    //printf("commands: %s\n", commands);

    int pipefd[2]; // 管道文件描述符
    pid_t pid;

    char *pipe_pos = strchr(commands, '|');
    //printf("pipe_pos: %d\n", *pipe_pos);

    if(pipe_pos){ // 管道符号存在
        //printf("pipe found\n");
        *pipe_pos = '\0'; // 将管道符号替换为字符串结束符，变成左右两个字符串
        char * left_command = commands;
        char * right_command = pipe_pos + 1;

        //printf("left_command: %s\n", left_command);
        //printf("right_command: %s\n", right_command);

        if(pipe(pipefd) < 0){
            perror("pipr error");
            exit(EXIT_CODE);
        }

        // 创建第一个子进程，执行左边命令
        if ((pid = fork()) == 0) {
            // 子进程执行左边命令
            //printf("child process execute left command\n");
            close(pipefd[0]); // 关闭读端
```

```
dup2(pipefd[1], STDOUT_FILENO); // 将管道写端文件描述符复制给标准输出
close(pipefd[1]);

handle_mayRedir_commands(left_command); // 处理左边命令
exit(EXIT_SUCCESS); // 退出子进程 !!! 非常重要, 补充到报告中!
} else if (pid < 0) {
    perror("fork");
    exit(EXIT_CODE);
}

if((pid =fork())==0){ // 创建第二个子进程, 执行右边命令
    // 子进程执行右边命令
    //printf("child process execute right command\n");
    close(pipefd[1]); // 关闭写端
    dup2(pipefd[0], STDIN_FILENO); // 将管道读端文件描述符复制给标准输入
    close(pipefd[0]);

    // 递归处理右边命令
    if(strchr(right_command, '|')!= NULL){ //中止条件
        handle_multiple_pipe(right_command);
    }
    else{ // 右边命令没有管道的情况

        handle_mayRedir_commands(right_command); // 处理右边命令
    }
    exit(EXIT_SUCCESS); // 退出子进程

}
else if (pid < 0) {
    perror("fork");
    exit(EXIT_CODE);
}

close(pipefd[0]);
close(pipefd[1]);
```

```

        while(wait(NULL)>0); // 等待所有子进程结束
    }
}

```

```

int count_pipe(const char *commands){
    int count = 0;
    while(*commands != '\0'){
        if(*commands == '|' )count++;
        commands++;
    }
    return count;
}

```

```

bool isRedir(char *commands){
    for(int i =0;i < strlen(commands) ; i++){
        if(commands[i] == '<' || commands[i] == '>')
            return true;
    }
    return false;
}

```

// 处理单命令（无管道的）重定向

//由于带重定向的指令就只有ppt上的四种形式即：<,>,<>,><; <和>之间不存在其他command，除  
//而管道的处理逻辑是会将多个命令分开为不带管道的单命令执行，所以重定向只需要考虑单条指

```

void handle_redirection(char *commands) {
    //printf("处理重定向: Handle_redirection\n");
    int inNum = 0, outNum = 0; // < 和 > 的个数
    pid_t pid;
    //保存原始标准输入和标准输出
    int saved_stdin = dup(STDIN_FILENO);
    int saved_stdout = dup(STDOUT_FILENO);

    char* commands_temp = malloc(sizeof(commands)); //避免设置重定向为'\0'时字符串遍历
    strcpy(commands_temp, commands);

    for(int i =0;i<strlen(commands_temp);i++){

```

```

if(commands_temp[i] == '>'){ //处理输出重定向
    outNum++;
    if(outNum > 1){
        perror("Too many output redirections");
        exit(EXIT_CODE);
    }
    char *outredir_pos = &commands[i];
    //printf("outredir_pos: %d\n", *outredir_pos);
    char *out_file = strtok(outredir_pos + 1, " \t"); //输出文件, 并且使用strtok
    //printf("out_file: %s\n", out_file);

    int fd = open(out_file, O_WRONLY | O_CREAT | O_TRUNC, 0666); //写入方式 (O_WRONLY)
    if(fd<0){
        perror("Output redirection error");
        exit(EXIT_CODE);
    }
    dup2(fd, STDOUT_FILENO); //复制fd到STDOUT_FILENO (标准输出), 即将标准输出重定向到fd
    close(fd);

    *outredir_pos = '\0'; //设置为字符串结束符号会影响到循环, 所以循环用的是temp[i]
}
else if(commands_temp[i]=='<'){ //处理输入重定向
    inNum++;
    if(inNum > 1){
        perror("Too many input redirections");
        exit(EXIT_CODE);
    }
    char *inredir_pos = &commands[i];
    //printf("inredir_pos: %d\n", *inredir_pos);
    char *in_file = strtok(inredir_pos + 1, " \t"); //输入文件
    //printf("in_file: %s\n", in_file);

    int fd = open(in_file, O_RDONLY); //只读模式打开文件
    if(fd<0){
        perror("Input redirection error");
        exit(EXIT_CODE);
    }
}

```

```
        dup2(fd,STDIN_FILENO);
        close(fd);

        *inredir_pos = '\0';
    }
}

//printf("输入重定向数量: %d", inNum);
//printf("输出重定向数量: %d", outNum);
//处理最左边的命令
char *left_command = commands;
//printf("redir_left_command: %s\n", left_command);

if(pid=fork() == 0){
    char *argv_left[ARGC_SIZE]; // 左边命令
    int argc_left = splitCline_getArgcv(left_command, argv_left); // 分割左边命令

    /*printf("Left command argc: %d\n", argc_left);
    printf("Left command: ");
    for (int i = 0; i < argc_left; i++) {
        printf("%s ", argv_left[i]); // 打印左边命令
    }
    printf("\n");*/

    handle_basic_commands(argv_left, argc_left); // 处理左边命令
    exit(EXIT_SUCCESS); // 退出子进程
}

else if(pid < 0){
    perror("fork");
    exit(EXIT_CODE);
}

else{
    wait(NULL); // 等待子进程结束
}

dup2(saved_stdout,STDOUT_FILENO); //重新还原主进程的标准输入和输出, 否则终端命令的
close(saved_stdout);
```

```
    dup2(saved_stdin, STDIN_FILENO);
    close(saved_stdin);
}

int main() {
    while (1) {
        print_prompt(); // 打印提示符
        get_command(commandline, sizeof(commandline)); // 获取用户输入

        char *commandline_copy = malloc(sizeof(commandline));
        strcpy(commandline_copy, commandline);

        int argc = splitCline_getArgcv(commandline, argv); // 分割命令，存储argv并返回
        if (argc == 0) continue; // 空命令直接跳过该轮循环

        //printf("commandline: %s\n", commandline);

        int pipe_count = count_pipe(commandline_copy); // 管道的个数

        //printf("管道数量为: %d\n", pipe_count);
        if(pipe_count == 1){
            //printf("管道数量为1，进入单管道情况\n");
            handle_single_pipe(commandline_copy); // 单管道情况
        }
        else if(pipe_count > 1){
            //printf("管道数量大于1，进入多管道情况\n");
            handle_multiple_pipe(commandline_copy); // 多管道情况
        }
        else{
            //printf("没有管道；main function enter handle normal commands\n");
            handle_mayRedir_commands(commandline_copy); // 处理内可能带有重定向的命令
        }
    }
    return 0;
}
```