**华东师范大学数据学院上机实践报告**

|  |  |  |
| --- | --- | --- |
| **课程名称**：操作系统 | **年级**：2016级 | **上机实践成绩**： |
| **指导教师**：翁楚良 | **姓名**：杜云滔 |  |
| **上机实践名称**：Shell及系统调用 | **学号**：10153903105 | **上机实践日期**：2018.3.16 |
| **上机实践编号**： | **组号**： | **上机实践时间**：2018.3.16 |

1. **目的**

1）带参数的程序运行功能。

program arg1 arg2 … argN

2）重定向功能，将文件作为程序的输入/输出。

a)“>”表示覆盖写。

program arg1 arg2 … argN > output-file

b)“>>”表示追加写。

program arg1 arg2 … argN >> output-file

c)“<”表示文件输入。

program arg1 arg2 … argN < input-file

3）管道符号“|”，在程序间传递数据。

programA arg1 … argN | programB arg1 … argN | programC …

4) 后台符号& ,表示此命令将以后台运行的方式执行。

program arg1 arg2 … argN &

5) 工作路径移动命令cd

6) shell退出命令exit

7) history显示开始任务后执行的命令；history n显示最近执行的n条指令

1. **内容与设计思想**

# 基本思路

很明显本次实验主要是以考察Shell基本功能以及管道的实现为主。之前已经写了一篇管道的理解，可以参考[这篇文章](http://zealscott.com/2018/03/06/%E7%AE%A1%E9%81%93%E7%9A%84%E7%90%86%E8%A7%A3%E4%B8%8E%E5%AE%9E%E7%8E%B0/)。

## 熟悉命令

首先我们先在UNIX自带的Shell下实现重定向和管道功能，示例命令可以参考如下：

# ps &

# cat numbers.txt | sort > temp.txt

# sort < numbers.txt | grep 1 > a.txt

# ps -ef | grep -sh

# cd ..

我们不难发现：

1. “>”，“>>”重定向命令只能在命令中**出现一次**，一旦出现后，之后还有什么命令也是无效的。
2. “<”命令也只能出现一次，但是后面可以接管道命令。
3. “|”管道命令可以出现多次，且管道之后还可以使用重定向符号。
4. 实际上所有命令进入程序之后都是一串字符串，因此对字符串的解析是最重要的。
5. 对于**ps**，**ls**，**cd**等命令，可以使用**exceve**命令进行操作，并不需要我们自己实现。
6. 如果注意，可以发现系统Shell在实现后台进程时，可能会出现如下情况： fig:

* 我们让**ls**的结果在后台运行，但为什么会在结果前多一个“#”呢？ 原因是因为后台运行的子进程和前台运行的父进程同时进行，谁先谁后不能确定，图中就是父进程先运行，打印了“#”，子进程再打印**ls**的结果，因此出现了这种情况。

## 难点

1. 管道的实现以及**fork()**的使用。
2. 子父进程进行信号交互，以及回收僵尸进程。
3. 多文件的协调和编译。

# 大体框架

## 主函数入口

由于我们在Windows下写这个Shell无法编译，每次必须在UNIX下编译，因此必须在写之前就想好模块布局，不然很难debug和进行单元测试。 一个Shell其实就是一个**while(1)**的死循环，每次输出提示符到屏幕，然后执行输入的字符串命令。因此不难写出大体框架：

int main() {  
 /\*Command line\*/  
 while (1) {  
  
 printf("cmd >");  
 /\*set buf to empty\*/  
 memset(buf, 0, sizeof(buf));  
 /\*Read from keyboard\*/  
 fgets(buf, MAXLINE, stdin);  
  
 /\*The function feof() tests the end-of-file indicator  
 for the stream pointed to by stream,  
 returning non-zero if it is set. \*/  
 if (feof(stdin))  
 exit(0);  
 /\*update the command history\*/  
 UpdateHistory();  
 /\*command exceve\*/  
 command();  
 }  
 return 0;  
}

主程序的确很简单，就是每次用**buf**读取输入的字符串，然后更新输入列表（为了 **history**功能的实现），然后再解析命令（**command**）即可。

## 字符串命令存储方式

Shell主要就是对得到的命令进行操作，因此命令如何存储是至关重要的。最简单的想法就是用一个**char\*[]**字符串数组存储，但是我们后面对命令解析需要 **命令的下标**等其他信息，因此这里选择用**struct**进行存储更为方便。 定义结构体如下：

struct CommandInfomation {  
 char\* argv[512]; /\*store the command after Parsing\*/  
 int argc; /\*the number of argv,split with space\*/  
 int index; /\*store the index of special character\*/  
 int background; /\*whether it is a background command\*/  
 enum specify type[50];  
 int override; /\* in case after < command has muti pipes \*/  
 char\* file;  
};

初始化函数为：

void initStruct(struct CommandInfomation\* a) {  
 a->argc = 0;  
 a->index = 0;  
 a->background = 0;  
 a->override = 0;  
 a->file = NULL;  
 memset(a->type, 0, sizeof(a->type));  
}

## 特殊字符命令

对于重定向">"，管道"|"等特殊命令，我们需要使用额外的标识来注明，方便后面的操作。这里使用**eunm**实现。

/\*the enum stand for different command\*/  
enum specify {NORMAL, OUT\_REDIRECT, IN\_REDIRECT, OUT\_ADD, PIPE};

## 主要函数详解

### pipe(fd[2])

此函数用于实现无名管道，将fd[2]数组中的两个文件描述符分别标记为管道读（fd[0]）和管道写（fd[1]）。

### dup(fd)

为复制文件操作符的系统函数，可以定向目前未被使用的最小文件操作符到fd所指的文件。相类似的函数还有**dup2[fd1,fd2]**,意思是 **未被使用**的文件描述符**fd2** 作为**fd1**的副本，进过此函数后，**fd1**和**fd2**都可访问同一个文件。

### execlp(const char \*file, const char \*arg, ...)

属于exec()函数族，会从PATH环境变量所指的目录中查找符合参数file的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的argv[0]、argv[1]……，最后一个参数必须用空指针(NULL)作结束。 命令中的**ls**，**ps**等内置系统命令都可以由此函数进行解析。要注意，此函数一经调用就不会再返回。

### chdir(const char \* path)

改变当前的工作路径以参数path所指的目录，使用比较简单，支持常用的改变路径的方式，例如退回上一级：**cd ..** ，也支持绝对路径。

## 执行命令

由主函数可知，我们得到了命令需要进行解析，由于我们知道**exceve**函数一旦调用就不会返回，因此要使用**fork()**函数对其子进程进行处理。 这里需要注意的是，由于子进程一定要比父进程先结束，因此我们需要将执行的命令放到**子进程中**，父进程进行等待或者执行后面的命令，否则会出现父进程结束子进程还在运行的错误。

### 父子进程进行通讯

需要注意的是，Shell支持**后台程序运行**，因此，父进程不一定要等待子进程运行结束才做后面的事情，但这就涉及到子进程结束后，父进程需要回收僵尸进程。那么，如何做到这一点呢？

#### Linux上进行信号屏蔽

在Linux系统上，我们可以使用**signal(int signum, sighandler\_t handler)**函数来设置某一类的信号处理或者屏蔽。我们知道，子进程要**exit()**之前，会发送**SIGCHLD**信号给父进程，提醒父进程来回收子进程的退出状态和其他信息。 在这里，我们可以使用一个特殊的技巧：

signal(SIGCHLD, SIG\_IGN)

这里是让父进程屏蔽子进程的信号，为什么这样就可以做到回收僵尸进程的作用呢？原来是因为在Linux中，**当我们忽略SIGCHLD信号时，内核将把僵尸进程交由init进程去处理，能够省去大量僵尸进程占用系统资源**。因此，屏蔽了子信号后，子程序在要结束时发送信号没人应答，内核就会认为这是一个孤儿进程，因此被init进程去回收，可以很好的解决我们面临的问题。

#### BSD系统上的信号处理

而我们使用的是Minix3.3的系统，经过实测，内核**并不会在父进程屏蔽信号后主动回收孤儿进程**，因此不能使用这种方法。 那怎么办呢？因此只能自己写一个handler，规定父进程在收到子进程结束的信号后再wait，这样也可以实现此功能。但缺点就是**wait**函数需要阻塞父进程直到子进程结束为止，对于并发要求较高的并发服务器，可能就不是很适用。 我们使用这种方法完成后台程序的运行：

void SIG\_IGN\_handler(int sig)  
 {  
 waitpid(-1, NULL, 0);  
 return;  
 }

在**主程序**中install这个handler：

signal(SIGCHLD, SIG\_IGN\_handler);

这样就完成了后台进程的功能。

1. **使用环境**

**Minix3.3**

1. **实验过程**

### history功能实现

查找前n个命令是比较简单的功能，我们可以使用队列进行实现，在这里我就稍微偷懒一点，直接使用定长的字符串数组进行。

/\*update command history\*/  
void UpdateHistory()  
{  
 char \*temp;  
 if (strcmp(buf, "\n") == 0)  
 return;  
 if (HistoryIndex > MAXLINE)  
 HistoryIndex = 0;  
 temp = (char \*)malloc(sizeof(buf));  
 strcpy(temp, buf);  
 CommandHistory[HistoryIndex++] = temp;  
 return;  
}  
  
  
/\*print the command with n lines\*/  
void PrintCommand(int n)  
{  
 int i,j=0;  
 if (n == -1) {  
 for (i = 0 ; i < HistoryIndex; i++)  
 printf("the %d command: %s\n", i, CommandHistory[i]);  
 }  
 else {  
 if (n > HistoryIndex) {  
 printf("Warning: the argument is too large.\n");  
 return;  
 }  
 for (i = HistoryIndex - n; i < HistoryIndex; i++)  
 printf("the %d command: %s\n", ++j, CommandHistory[i]);  
 }  
}

# 命令解析

完成一个命令的解析，最重要的步骤就是字符串的解析。我们如何对拿到的字符串进行分解呢？我的思路如下：

1. 使用**fgets()**等函数将输入的命令存放在缓存区中。
2. 对其用空格对其进行分割（使用[**strtok()**](http://www.cplusplus.com/reference/cstring/strtok/)等字符串处理函数），解析出特殊命令符（重定向">"，管道"|"，后台程序"&"等）
3. 识别出特殊命令：例如**history**，**回车**，**exit**，**cd**等，这些命令不能使用**exceve**函数进行解析和运行，需要单独处理。
4. 如果字符串中有特殊命令符，则需要对命令两边分别进行操作。

## 分割字符串

/\*  
 Parse the command line with space and get the argv array  
\*/  
void parseline() {  
 initStruct(&CommandInfo);  
  
 buf[strlen(buf) - 1] = ' '; /\*Replace trailing '\n' with space\*/  
  
 /\*split buf with space\*/  
 char\* token = strtok(buf, " ");  
 while (token) {  
 CommandInfo.argv[CommandInfo.argc++] = token;  
 token = strtok(NULL, " ");  
 }  
 /\*set the last command with NULL\*/  
 CommandInfo.argv[CommandInfo.argc] = NULL;  
  
 /\*empty command line\*/  
 if (CommandInfo.argc == 0)  
 return;  
  
 /\*indicate whether its background Command\*/  
 CommandInfo.background = (\*(CommandInfo.argv[CommandInfo.argc - 1]) == '&');  
 if (CommandInfo.background)  
 CommandInfo.argv[--CommandInfo.argc] = NULL;  
 return;  
}

## 特殊命令处理

针对空格、history、cd等特殊命令，可以先做预处理。

/\*if return 1, ignore the command\*/  
int IgnoreCommand() {   
 /\*if no command,continue\*/  
 if (CommandInfo.argc < 1)  
 return 1;  
  
 /\*exit command\*/  
 if (strcmp(CommandInfo.argv[0], "exit") == 0)  
 exit(0);  
  
 /\*history command\*/  
 if (strcmp(CommandInfo.argv[0], "history") == 0) {  
 if (CommandInfo.argc == 1)  
 /\*print all the history\*/  
 PrintCommand(-1);  
 else {  
 PrintCommand(atoi(CommandInfo.argv[1])); /\*convert string to int\*/  
 }  
 return 1;  
 }  
  
 /\*cd command to change directory\*/  
 if (strcmp(CommandInfo.argv[0], "cd") == 0) {  
 if (CommandInfo.argc > 1) {  
 if (chdir(CommandInfo.argv[1]) == -1) {  
 printf("error directory!\n");  
 }  
 }  
 return 1;  
 }  
  
 /\*wrong command\*/  
 if (strcmp(CommandInfo.argv[CommandInfo.argc - 1], "<") == 0 ||  
 strcmp(CommandInfo.argv[CommandInfo.argc - 1], ">") == 0 ||  
 strcmp(CommandInfo.argv[CommandInfo.argc - 1], "|") == 0) {  
 printf("Error:command error\n");  
 return 1;  
 }  
  
 return 0;  
}

## 解析命令操作符

对于“>”，“<”,“>>”操作符，不需要进行管道操作，因此直接先读取文件名。

int ReviseCommand() {  
 /\*  
 if the command is empty or exit or cd or history, should ignore the command;  
 \*/  
 if (IgnoreCommand())  
 return -1;  
  
 int i, override = 0;  
  
 /\*search the command with special charactors,and store the file and type\*/  
 for (i = 0; i < CommandInfo.argc; i++) {  
 if (strcmp(CommandInfo.argv[i], "<") == 0) {  
 CommandInfo.argv[i] = NULL;  
 CommandInfo.file = CommandInfo.argv[i + 1];  
 CommandInfo.type[CommandInfo.index++] = IN\_REDIRECT;  
 override = 1;  
   
 } else if (strcmp(CommandInfo.argv[i], ">") == 0) {  
 /\* if > is not the first command, should not set the file \*/  
 CommandInfo.argv[i] = NULL;  
 if (!override)  
 CommandInfo.file = CommandInfo.argv[i + 1];  
 CommandInfo.type[CommandInfo.index++] = OUT\_REDIRECT;  
 break;  
   
 } else if (strcmp(CommandInfo.argv[i], ">>") == 0) {  
 CommandInfo.argv[i] = NULL;  
 if (!override)  
 CommandInfo.file = CommandInfo.argv[i + 1];  
 CommandInfo.type[CommandInfo.index++] = OUT\_ADD;  
 break;  
   
 }  
 /\*multi - PIPE\*/  
 else if (strcmp(CommandInfo.argv[i], "|") == 0) {  
 CommandInfo.type[CommandInfo.index++] = PIPE;  
 CommandInfo.argv[i] = NULL;  
 }  
 }  
 return 1;  
}

# 命令主体框架

我们首先使用**parseline()**对得到的命令按照空格进行解析，之后再使用**ReviseCommand()**提取关键命令字符，识别回车键等，最后再对进程进行**fork()**，子进程（ChildCommand）执行命令，父进程根据是否有“&”选择等待子进程结束或者继续执行。

void command() {  
 pid\_t pid;  
 int indicator = 0;  
  
 parseline();  
  
 /\*re-edit command and get the file\*/  
 indicator = ReviseCommand();  
  
 if (indicator == -1)  
 return;  
  
 pid = fork();  
 if (!pid) {  
 /\*the background process should not be  
 disturbed by CTRL+C and CTRL+\\*/  
 /\*sigaction(SIGINT, SIG\_IGN, NULL);  
 sigaction(SIGQUIT, SIG\_IGN, NULL);\*/  
 ChildCommand();  
 exit(0);  
 } else {  
 if (!CommandInfo.background)  
 waitpid(pid, NULL, 0);  
 else {  
 /\*if background process, the father should ignore the signal  
 let init to reap it \*/  
 printf("there is a background process\n");  
 }  
 }  
 return;  
}

# 子进程命令框架

对于fork出来的子进程，如果只有重定向这种简单的命令，我们通过解析到的字符串和文件名就可以直接进行操作，如果涉及到多个管道的操作，那就要小心了。

/\*  
 the > command must be the last command,  
 so | command is ahead  
\*/  
void ChildCommand() {  
 int fd;  
 switch (CommandInfo.type[0]) {  
 case NORMAL:  
 Execvp(CommandInfo.argv[0], CommandInfo.argv);  
 break;  
  
 case IN\_REDIRECT: /\* < command\*/  
 fd = open(CommandInfo.file, O\_RDONLY);  
 if (fd == -1) {  
 printf("Error: wrong input!\n");  
 break;  
 }  
 dup2(fd, STDIN\_FILENO);  
  
 if (CommandInfo.type[1] == PIPE) {  
 EditInfo();  
 pipe\_command();  
 }  
 Execvp(CommandInfo.argv[0], CommandInfo.argv);  
 break;  
  
 case OUT\_REDIRECT: /\* > command\*/  
 fd = open(CommandInfo.file, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);  
 dup2(fd, STDOUT\_FILENO);  
 Execvp(CommandInfo.argv[0], CommandInfo.argv);  
 break;  
  
 case OUT\_ADD: /\* >> command\*/  
 fd = open(CommandInfo.file, O\_WRONLY | O\_APPEND | O\_CREAT, 0666);  
 dup2(fd, STDOUT\_FILENO);  
 Execvp(CommandInfo.argv[0], CommandInfo.argv);  
 break;  
  
 case PIPE: /\* | command\*/  
 pipe\_command();  
 break;  
 }  
}

# 管道

首先，我们在使用**pipe**创建管道后，需要**fork**一个进程，子进程用于向管道写，父进程用于向管道读。很有趣的一个问题是，当我们使用**fork**命令时，子父进程的执行顺序是不能确定的，那么是让父进程向管道读还是子进程向管道读呢？

我的理解是，由于父进程不能先于子进程结束，而如果管道中没有东西，从管道读的操作会被堵塞，可以利用这个性质让子进程先于父进程结束。具体做法就是：让父进程向管道读，子进程向管道写。由于“向管道写”操作总是先于“向管道读”操作，因此可以做到父进程结束前回收子进程的工作。 那么，我们怎么做到多管道进行操作呢？其实也不难，我们可以先把所需要的所有管道建立好，然后当子进程要进行**execv操作**之前，把它的输出fd指向下一个管道的输入，这样重复进行就能实现多个管道进行通讯了。

要注意的是，当进行到最后一个命令，这时候我们需要判断：若最后一个命令就是“|”，则我们需要将管道中的数据输出到屏幕即可；若最后一个命令时">",则需要将管道中的命令写入对应的文件中。

## 实现

总体来说对于管道的操作有三种情况：

1. 最开始有“<”重定向符号，接着有多个管道。
2. 一开始就是多个管道进行传输，最后输出到屏幕。
3. 通过多个管道传输后，最后重定向到指定文件。

其实这三种情况都可以写成一个函数，因为只需要对第一个命令和最后一个命令进行特殊处理即可。由于我太蠢，最开始没想到第一种情况，因此后面只能用修改字符串的方法曲线救国了，代码很丑，将就看吧。 pipe\_command：

/\*take care of pipe\*/  
void pipe\_command() {  
 /\*pointerindex is the index of each Command\*/  
 int i = 1, j = 0, pointerindex[20], commandnumber = CommandInfo.index;  
 int pipenumber = 0, pid, pipefd[20][2];  
 char\*\* P\_temp;  
 pointerindex[0] = 0; /\*the first command location\*/  
  
 /\*get all command's index\*/  
 while (i <= commandnumber) {  
 if (CommandInfo.argv[j] == NULL) {  
 pointerindex[i] = j + 1;  
 i++;  
 }  
 j++;  
 }  
  
 /\*if the last command is > or >>\*/  
  
 for (i = 0; i <= CommandInfo.index; i++) {  
 if (CommandInfo.type[i] == PIPE)  
 pipenumber++;  
 }  
  
 /\*init the pipe\*/  
 for (i = 0; i < 20; i++)  
 pipe(pipefd[i]);  
  
 i = 1;  
  
 if (fork() != 0) {  
 /\*this is the father, need to read\*/  
 dup2(pipefd[0][0], 0);  
 close(pipefd[0][1]);  
  
 P\_temp = CommandInfo.argv + pointerindex[1];  
 /\*fork a process\*/  
 pid = fork();  
  
 if (pid != 0) {  
 /\*this is the father,wait child return\*/  
 waitpid(pid, NULL, 0);  
 close(pipefd[1][1]); /\*no need to write\*/  
 next\_pipe(pipefd, pipenumber - 1, pointerindex);  
 } else {  
 /\*read the result from pipe and write to the next pipe\*/  
 dup2(pipefd[1][1], 1);  
 close(pipefd[1][0]); /\*no need to read\*/  
 Execvp(P\_temp[0], P\_temp);  
 }  
 } else {  
 /\*this is the child, need to write\*/  
 dup2(pipefd[0][1], 1);  
 close(pipefd[0][0]);  
 Execvp(CommandInfo.argv[0], CommandInfo.argv);  
 }  
}

对于多管道进行循环操作：

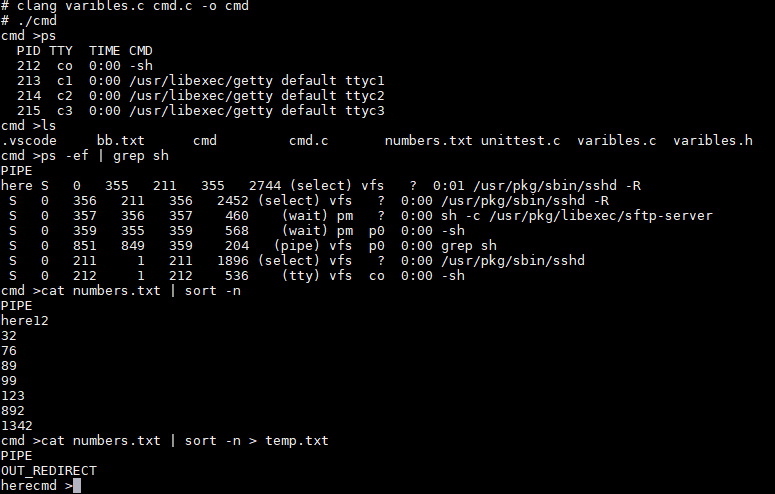
/\*special run for mult-pipes\*/  
void next\_pipe(int pipefd[20][2], int pipenumber, int pointerindex[]) {  
 char\*\* P\_temp;  
 char buffer[MAXLINE];  
 int commandnumber = CommandInfo.index; /\*get the all command number\*/  
 int i = 1, pid, fd, run\_index = 2; /\*refer to the index of command to run\*/  
 int flag = !(pipenumber + 1 == commandnumber);  
 int lastCommand = 0;  
  
 /\*if only one pipe and no other command, print it\*/  
 if (commandnumber == 1) {  
 read(pipefd[1][0], buffer, MAXLINE);  
 printf("%s", buffer);  
 return;  
 }  
  
 /\*loop to connect the pipe\*/  
  
 while (pipenumber--) {  
 pid = fork();  
 if (pid == 0) {  
 /\*this is the child, to run the command\*/  
 dup2(pipefd[i][0], 0); /\*read data\*/  
 close(pipefd[i][1]); /\*no need to write\*/  
 dup2(pipefd[i + 1][1], 1); /\*write to the next pipe\*/  
 /\*get the command\*/  
  
 P\_temp = CommandInfo.argv + pointerindex[run\_index];  
 Execvp(P\_temp[0], P\_temp);  
 } else {  
 /\*wait child completes the command\*/  
 waitpid(pid, NULL, 0);  
 i++; /\*next pipe\*/  
 run\_index++; /\*next command\*/  
 }  
 }  
  
 /\*if the last command is > or >>\*/  
 if (flag) {  
 if (CommandInfo.override)  
 lastCommand = CommandInfo.type[commandnumber];  
 else  
 lastCommand = CommandInfo.type[commandnumber - 1];  
 /\*if the command is >\*/  
 if (lastCommand == OUT\_REDIRECT) {  
 fd = open(CommandInfo.argv[pointerindex[commandnumber]],  
 O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);  
 } else { /\*the command is >>\*/  
 fd = open(CommandInfo.argv[pointerindex[commandnumber]],  
 O\_RDWR | O\_APPEND, 0666);  
 }  
 read(pipefd[i][0], buffer, MAXLINE); /\*read from pipe\*/  
 write(fd, buffer, strlen(buffer)); /\*write into file\*/  
 } else {  
 read(pipefd[i][0], buffer, MAXLINE); /\*read from pipe\*/  
 fprintf(stdout,"%s", buffer); /\*print into screen\*/  
 }  
 return;  
}

对第一种情况进行曲线救国：

void EditInfo() {  
 int i, location;  
 CommandInfo.index--;  
 /\* find the first command -- NULL \*/  
 for (i = 1; i < CommandInfo.argc; i++) {  
 if (CommandInfo.argv[i] == NULL) {  
 location = i;  
 break;  
 }  
 }  
 /\* override the used command\*/  
 for (i = location; i < CommandInfo.argc; i++) {  
 CommandInfo.argv[i] = CommandInfo.argv[i + 2];  
 }  
 CommandInfo.override = 1;  
}

1. **总结**

# 结果展示



# 总结

至此，终于完成了一个简单Shell的实现。虽然实现中碰到了不少的问题，最后的代码也不够优美，但这次实验让自己认识到了不少的问题。

1. C语言使用还不够熟练，特别是对于UNIX的各种系统调用。
2. 没有想清楚了再动笔，代码组织结构有问题，模块化不够。
3. 多文件的协调和编译还不够熟练，编译顺序？makefile文件？还不太会用。
4. 很多小bug没有意识到，比如说直接输入回车，子父进程的管理和回收等等。
5. 单元测试还不太会。

# ToDo

虽然把project要求完成了，但前前后后改得面目全非，导致自己都不愿意再看代码了。先挖个坑，等有时间了再重新理理思路，定个小目标，用两百行左右代码写好。

# 参考资料

1. [管道的理解与实现](http://zealscott.com/2018/03/06/%E7%AE%A1%E9%81%93%E7%9A%84%E7%90%86%E8%A7%A3%E4%B8%8E%E5%AE%9E%E7%8E%B0/)
2. [linux信号函数signal](http://blog.csdn.net/u013246898/article/details/52985739)
3. [Linux shell的实现](http://blog.csdn.net/tankai19880619/article/details/49678565)
4. Operating System:Design and Implementation,Third Edition
5. Computer Systems: A Programmer's Perspective, 3/E