

## 0.1 Project 2 实验报告

Students: Musicminion

### 0.1 Project 2 实验报告

#### 0.1.1 任务组一：UNIX Shell

##### 0.1.1.1 一、概述

###### 0.1.1.1.1 a) shell 界面介绍

###### 0.1.1.1.2 b) 代码样例

###### 0.1.1.1.3 b) 项目内容介绍

##### 0.1.1.2 二、在子进程中执行命令

##### 0.1.1.3 三、增加历史记录功能

##### 0.1.1.4 四、重定向输入和输出功能

##### 0.1.1.5 五、通过管道通讯

##### 0.1.1.6 六、完整的代码

##### 0.1.1.7 七、演示的效果图

###### 0.1.1.7.1 1、输入命令的演示

###### 0.1.1.7.2 2、重定向的演示

###### 0.1.1.7.3 3、历史功能的演示

###### 0.1.1.7.4 4、管道功能的演示

#### 0.1.2 任务组二：用于任务信息的 Linux 内核模块

##### 0.1.2.1 一、写入到 `/proc` 文件系统

##### 0.1.2.2 二、从 `/proc` 文件系统读取

##### 0.1.2.3 三、原始代码

#### 0.1.3 任务小结

### 0.1.1 任务组一：UNIX Shell

该项目包括设计一个 C 程序作为 shell 接口，该接口接受用户命令，然后在单独的进程中执行每个命令。您的实现将支持输入和输出重定向，以及作为 IPC 形式的一对命令之间的管道。完成这个项目将涉及使用 UNIX `fork()`、`exec()`、`wait()`、`dup2()` 和 `pipe()` 系统调用，并且可以在任何 Linux、UNIX 或 macOS 系统上完成。

#### 0.1.1.1 一、概述

##### 0.1.1.1.1 a) shell 界面介绍

shell 界面给用户一个提示，然后输入下一个命令。下面的示例说明了提示符 `osh>` 和用户的下一个命令：`cat prog.c`。（此命令使用 UNIX `cat` 命令在终端上显示文件 `prog.c`。）

```
1 | osh>cat prog.c
```

实现 shell 接口的一种技术是让父进程首先读取用户在命令行上输入的内容（在本例中为 `cat prog.c`），然后创建一个单独的子进程来执行命令。除非另有说明，否则父进程在继续之前等待子进程退出。这在功能上类似于图 3.9 中所示的新流程创建。然而，UNIX shell 通常也允许子进程在后台运行，或者同时运行。为此，我们在命令末尾添加一个 `&` 符号。因此，如果我们将上述命令重写为如下，这样父进程和子进程将同时运行。

```
1 | osh>cat prog.c &
```

##### 0.1.1.1.2 b) 代码样例

- 使用 `fork()` 系统调用创建单独的子进程，并使用 `exec()` 系列中的系统调用之一执行用户命令。
- 下面提供了一个提供命令行 shell 的一般操作的 C 程序。`main()` 函数显示提示 `osh>` 并概述在读取用户输入后要采取的步骤。只要应该 `run` 等于 1，`main()` 函数就会不断循环；当用户在提示符下输入 `exit` 时，您的程序将设置应该运行行为 0 并终止。

```
1 | #include <stdio.h>
2 | #include <unistd.h>
3 | #define MAX LINE 80                                /* The maximum length command */
4 | int main(void)
5 | {
6 |     char *args[MAX LINE/2 + 1];                    /* command line arguments */
7 |     int should run = 1;                             /* flag to determine when to exit
program */
8 |     while (should run) {
9 |         printf("osh>");
10 |         fflush(stdout);
11 |         /**
12 |          * After reading user input, the steps are:
13 |          * (1) fork a child process using fork()
14 |          * (2) the child process will invoke execvp()
15 |          * (3) parent will invoke wait() unless command included &
16 |          */
```

```

17     }
18     return 0;
19 }

```

#### 0.1.1.1.3 b) 项目内容介绍

本次的项目分为一下几个部分：

1. 创建子进程并在子进程中执行命令。
2. 提供一个历史功能
3. 增加对输入输出重定向的功能
4. 允许父子进程通过管道通讯

#### 0.1.1.2 二、在子进程中执行命令

第一个任务是修改图 3.36 中的 main() 函数，以便派生一个子进程并执行用户指定的命令。这将需要将用户输入的内容解析到单独的标记中，并将标记存储在字符串数组中（图 3.36 中的 args）。例如，如果用户在 osh> 提示符下输入命令 `ps -ael`，则存储在 args 数组中的值为：

```

1     args[0] = "ps"
2     args[1] = "-ael"
3     args[2] = NULLc

```

这个 args 数组将被传递给 execvp() 函数，该函数具有以下原型：

```

1 execvp(char *command, char *params[])

```

- command 表示要执行的命令，
- params 存储该命令的参数。
- 对于这个项目，execvp() 函数应该被调用为 execvp(args[0], args)。
- 一定要检查用户是否包含&来判断父进程是否要等待子进程退出。

例如，下面的代码完成口令的识别与切割功能，分割单位是空格。

```

1 void token_initAndGet(char** args, char** history_buffer, char* Instruction,
   char* last_parameter) {
2     for(int i=0; i<MAX_LINE/2+1; ++i)
3         args[i]=NULL;
4
5     printf("osh>");
6     fflush(stdout);
7     fgets(Instruction, MAX_LINE, stdin);
8     init_args(Instruction, args, last_parameter);
9     init_history(history_buffer, args, last_parameter);
10    free(Instruction);
11    free(last_parameter);
12 }

```

```

1 char **init_args(char* Instruction, char** args, char * last_parameter)
2 {
3     //DeInstruction
4     int Index_Instruction=0;
5     int Index_args=0;
6     char tmp=Instruction[Index_Instruction];
7     int precious_is_space=0;
8
9     while (tmp!='\n' && tmp!=EOF)
10    {
11        if (Index_Instruction==0)
12        {
13            free(args[0]);
14            args[0]=(char *) malloc(sizeof(char)*MAX_LINE);
15            memset(args[0], 0, sizeof(args[0]));
16
17        }
18        if (tmp==' ')
19        {
20            if (precious_is_space==0)
21            {
22                Index_args++;
23                args[Index_args]=(char *) malloc(sizeof(char)*MAX_LINE);
24                precious_is_space=1;
25            }
26        }
27        else
28        {
29            strncat(args[Index_args], &tmp, 1);
30
31            precious_is_space=0;
32        }
33
34        Index_Instruction++;
35        tmp=Instruction[Index_Instruction];
36    }
37
38    if (args[Index_args] != NULL)
39        strcpy(last_parameter, args[Index_args]);
40    if (strcmp(last_parameter, "&&")==0)
41        args[Index_args]=NULL;
42 }

```

### 0.1.1.3 三、增加历史记录功能

下一个任务是修改 shell 接口程序，使其提供历史功能，允许用户通过输入 `!!` 来执行最近的命令。例如，如果用户输入命令 `ls -l`，然后她可以通过输入 `!!` 再次执行该命令。在提示下。以这种方式执行的任何命令都应该在用户屏幕上回显，并且该命令也应该作为下一个命令放置在历史缓冲区中。您的程序还应该管理基本的错误处理。如果历史记录中没有最近的命令，输入 `!!` 应打印消息“历史记录中没有命令”。

- 我编写了下面的函数，针对性的处理 `!!` 的命令
- 首先检测 `buffer` 里面有没有历史命令，如果没有，那就说明用户之前没有历史命令，需要提示
- 有的话就会进行一次历史命令的拷贝，这样方便后期的执行工作。

```
1
2 int historyDealer(char** history_buffer, char** args){
3     ///!! check
4
5     if(args[0]!=NULL&&strcmp(args[0],"!!")==0)
6     {
7         free(args[0]);
8         args[0]=NULL;
9         if(history_buffer[0]==NULL)
10        {
11            printf("No commands in history.\n");
12            return 1;
13        }
14        else
15        {
16            for(int i=0;i<MAX_LINE/2+1;++i)
17            {
18                if(history_buffer[i]==NULL)
19                    break;
20                else
21                {
22                    args[i]=(char*)malloc(MAX_LINE*sizeof(char));
23                    strcpy(args[i],history_buffer[i]);
24                }
25
26                printf("%s ",history_buffer[i]);
27            }
28            printf("\n");
29        }
30    }
31
32    return 0;
33 }
34
```

#### 0.1.1.4 四、重定向输入和输出功能

- 现在应该修改您的 shell 以支持“>”和“<”重定向运算符，其中“>”将命令的输出重定向到文件，“<”将输入重定向到文件中的命令。例如，如果用户输入

```
1 | osh>ls > out.txt
```

- ls 命令的输出将被重定向到文件 out.txt。同样，输入也可以重定向。例如，如果用户输入

```
1 | osh>sort < in.txt
```

- in.txt 文件将用作排序命令的输入。
- 管理输入和输出的重定向将涉及使用 dup2() 函数，该函数将现有文件描述符复制到另一个文件描述符。例如，如果 fd 是文件 out.txt 的文件描述符，则调用 dup2 (fd, 标准输出文件号)
- 将 fd 复制到标准输出（终端）。这意味着对标准输出的任何写入实际上都将发送到 out.txt 文件。您可以假设命令将包含一个输入或一个输出重定向，并且不会同时包含两者。换句话说，您不必关心诸如 `sort < in.txt > out.txt` 之类的命令序列。

具体实现：

- 对于 `>` `<` 两个特殊的口令单独检测，例如下面的两个函数

```
1 | redirect_output(args);
2 | redirect_input(args,last_parameter);
```

```
1 | void redirect_output(char** args)
2 | {
3 |     for(int i=0;i<MAX_LINE/2+1;++i)
4 |     {
5 |         if(args[i]!=NULL && strcmp(args[i], ">")==0) // > exists
6 |         {
7 |             char file_name[MAX_LINE];
8 |             strcpy(file_name,args[i+1]);
9 |             int fd=open(file_name,O_RDWR | O_NOCTTY | O_NDELAY);
10 |
11 |             if(fd == -1){
12 |                 fd = creat(file_name,S_IRWXU);
13 |             }
14 |             dup2(fd,STDOUT_FILENO);
15 |
16 |             free(args[i]);
17 |             free(args[i+1]);
18 |             args[i]=NULL;
19 |             args[i+1]=NULL;
20 |
21 |             close(fd);
22 |         }
```

```

23     }
24 }
25
26 int redirect_input(char** args, char* last_parameter)
27 {
28     for(int i=0; i<MAX_LINE/2+1; ++i)
29     {
30         if(args[i]!=NULL && strcmp(args[i], "<")==0) // < exists
31         {
32             char file_name[MAX_LINE];
33             strcpy(file_name, args[i+1]);
34
35             int fd=open(file_name, O_RDWR | O_NOCTTY | O_NDELAY);
36
37             free(args[i]);
38             free(args[i+1]);
39             args[i]=NULL;
40             args[i+1]=NULL;
41             dup2(fd, STDIN_FILENO);
42             char Instruction[MAX_LINE*sizeof(char)];
43             fgets(Instruction, MAX_LINE, stdin);
44             init_args(Instruction, args+i, last_parameter);
45
46             close(fd);
47         }
48     }
49     return 0;
50 }

```

#### 0.1.1.5 五、通过管道通讯

对 shell 的最后修改是允许一个命令的输出使用管道作为另一个命令的输入。例如下面的命令序列

```

1 | osh>ls -l | less

```

将命令 `ls -l` 的输出用作 `less` 命令的输入。`ls` 和 `less` 命令都将作为单独的进程运行，并将使用 3.7.4 节中描述的 UNIX `pipe()` 函数进行通信。创建这些独立进程的最简单方法可能是让父进程创建子进程（它将执行 `ls -l`）。这个子进程还将创建另一个子进程（执行较少），并将在它自己和它创建的子进程之间建立一个管道。实现管道功能还需要使用上一节中描述的 `dup2()` 函数。最后，尽管可以使用多个管道将多个命令链接在一起，但您可以假设命令将仅包含一个管道字符并且不会与任何重定向运算符组合。

- 对于父进程，只执行第一段的命令
- 对于子进程，执行 `|` 后面的命令
- 将父进程的输出结果作为子进程的标准输入。

```

1 | int pipeDealer(char** args, int pipe_position){

```

```

2     pid_t pid;
3     pid=fork();
4
5     if(pid==0)//child process
6     {
7         int fd[2];
8         pid_t pid;
9
10        /*create a pipe*/
11        if(pipe(fd)==-1)
12        {
13            fprintf(stderr,"Pipe failed");
14            return 1;
15        }
16
17        pid=fork();
18
19        if(pid>0)//parent process(actually child process of the initial one)
20        {
21            for(int i=pipe_position;i<MAX_LINE/2+1;++i)
22            {
23                free(args[i]);
24                args[i]=NULL;
25            }
26            close(fd[READ_END]);
27            dup2(fd[WRITE_END],STDOUT_FILENO);
28            execvp(args[0],args);
29        }
30        else if(pid==0)//grandson process
31        {
32            strcpy(args[0],args[pipe_position+1]);
33            for(int i=1;i<MAX_LINE/2+1;++i)
34            {
35                free(args[i]);
36                args[i]=NULL;
37            }
38            close(fd[WRITE_END]);
39            dup2(fd[READ_END],STDIN_FILENO);
40            execvp(args[0],args);
41        }
42    }
43    else // father process
44    {
45        wait(NULL);
46    }
47    return 0;
48 }

```



#### 0.1.1.6 六、完整的代码

- 主函数内容还是比较简洁的
- 首先要获取用户的输入口令
- 然后要针对集中特别的情况：
  - 例如 `exit` 退出口令
  - 例如 `!!` 历史口令
- 执行口令，要分为两种情况
  - 带有管道的情况
  - 不带有管道的情况
- 清理内存，释放空间

```
1 // Student: zhangziqian
2 // ID: 520111910121
3 // 2022 OS Project2 - shell
4
5 int main(void)
6 {
7     char *args[MAX_LINE/2 + 1]={NULL};
8     char *history_buffer[MAX_LINE/2 + 1]={NULL};
9     int should_run = 1;
10
11     while (should_run) {
12         char * Instruction=(char*) malloc(sizeof(char)*MAX_LINE);
13         char * last_parameter=(char *) malloc (sizeof(char)*MAX_LINE);
14
15         token_initAndGet(args,history_buffer,Instruction,last_parameter);
16
17         // specially deal with !!
18         if(historyDealer(history_buffer,args) == 1){
19             continue;
20         }
21
22         // deal with exit
23         if(args[0]!=NULL&&strcmp(args[0],"exit")==0)
24         {
25             should_run=0;
26             continue;
27         }
28
29         //token executor
30         int pipe_position=detect_pipe(args);
31         if(pipe_position!=0) // pipe exists
32             pipeDealer(args,pipe_position);
```

```

33         else
34             NONEpipeDealer(args, last_parameter, &should_run);
35     }
36     clearAll(args, history_buffer);
37     return 0;
38 }

```

```

1 // Student: zhangziqian
2 // ID: 520111910121
3 // 2022 OS Project2
4
5 #include <stdlib.h>
6 #include <string.h>
7 #include <unistd.h>
8 #include <sys/wait.h>
9 #include <stdio.h>
10 #include <fcntl.h>
11 #include <sys/types.h>
12 #define MAX_LINE 80
13 #define REDIRECT_INPUT_FAIL 1
14 #define READ_END 0
15 #define WRITE_END 1
16
17
18 char **init_history(char** history_buffer, char** args, char* last_parameter)
19 {
20     if(args[0]==NULL || strcmp(args[0], "!!")==0)
21         return NULL;
22     int i=0;
23     for(i=0; i<MAX_LINE/2+1; i++)
24     {
25         if(args[i]==NULL)
26             break;
27         history_buffer[i]=(char*) malloc(MAX_LINE*sizeof(char));
28         strcpy(history_buffer[i], args[i]);
29     }
30     if(strcmp(last_parameter, "&&")==0)
31     {
32         history_buffer[i]=(char*) malloc(MAX_LINE*sizeof(char));
33         history_buffer[i]="&&";
34     }
35
36 }
37
38 char **init_args(char* Instruction, char** args, char * last_parameter)

```

```

39 {
40     //DeInstruction
41     int Index_Instruction=0;
42     int Index_args=0;
43     char tmp=Instruction[Index_Instruction];
44     int precious_is_space=0;
45
46     while (tmp!='\n' && tmp!=EOF)
47     {
48         if (Index_Instruction==0)
49         {
50             free(args[0]);
51             args[0]=(char *) malloc(sizeof(char)*MAX_LINE);
52             memset(args[0], 0, sizeof(args[0]));
53
54         }
55         if (tmp==' ')
56         {
57             if (precious_is_space==0)
58             {
59                 Index_args++;
60                 args[Index_args]=(char *) malloc(sizeof(char)*MAX_LINE);
61                 precious_is_space=1;
62             }
63         }
64         else
65         {
66             strncat(args[Index_args], &tmp, 1);
67
68             precious_is_space=0;
69         }
70
71         Index_Instruction++;
72         tmp=Instruction[Index_Instruction];
73     }
74
75     if (args[Index_args] != NULL)
76         strcpy(last_parameter, args[Index_args]);
77     if (strcmp(last_parameter, "&&")==0)
78         args[Index_args]=NULL;
79 }
80
81
82 void redirect_output(char** args)
83 {
84     for (int i=0; i<MAX_LINE/2+1; ++i)
85     {

```

```

86         if(args[i]!=NULL && strcmp(args[i], ">")==0) // > exists
87         {
88             char file_name[MAX_LINE];
89             strcpy(file_name, args[i+1]);
90             int fd=open(file_name, O_RDWR | O_NOCTTY | O_NDELAY);
91
92             if(fd == -1){
93                 fd = creat(file_name, S_IRWXU);
94             }
95             dup2(fd, STDOUT_FILENO);
96
97             free(args[i]);
98             free(args[i+1]);
99             args[i]=NULL;
100            args[i+1]=NULL;
101
102            close(fd);
103        }
104    }
105 }
106
107 int redirect_input(char** args, char* last_parameter)
108 {
109     for(int i=0; i<MAX_LINE/2+1; ++i)
110     {
111         if(args[i]!=NULL && strcmp(args[i], "<")==0) // < exists
112         {
113             char file_name[MAX_LINE];
114             strcpy(file_name, args[i+1]);
115
116             int fd=open(file_name, O_RDWR | O_NOCTTY | O_NDELAY);
117
118             free(args[i]);
119             free(args[i+1]);
120             args[i]=NULL;
121             args[i+1]=NULL;
122             dup2(fd, STDIN_FILENO);
123             char Instruction[MAX_LINE*sizeof(char)];
124             fgets(Instruction, MAX_LINE, stdin);
125             init_args(Instruction, args+i, last_parameter);
126
127             close(fd);
128         }
129     }
130     return 0;
131 }
132

```

```

133 int detect_pipe(char** args)
134 {
135     for(int i=0;i<MAX_LINE/2+1;++i)
136     {
137         if(args[i]!=NULL && strcmp(args[i],"")==0)
138             return i;
139     }
140     return 0;
141 }
142
143 int historyDealer(char** history_buffer, char** args){
144     ///!! check
145
146     if(args[0]!=NULL&&strcmp(args[0],"!!")==0)
147     {
148
149
150         free(args[0]);
151         args[0]=NULL;
152
153
154         if(history_buffer[0]==NULL)
155         {
156             printf("No commands in history.\n");
157             return 1;
158         }
159         else
160         {
161             for(int i=0;i<MAX_LINE/2+1;++i)
162             {
163                 if(history_buffer[i]==NULL)
164                     break;
165                 else
166                 {
167                     args[i]=(char*)malloc(MAX_LINE*sizeof(char));
168                     strcpy(args[i],history_buffer[i]);
169                 }
170
171                 printf("%s ",history_buffer[i]);
172             }
173             printf("\n");
174         }
175     }
176
177     return 0;
178 }
179

```

```

180 int pipeDealer(char** args,int pipe_position){
181     pid_t pid;
182     pid=fork();
183
184     if(pid==0)//child process
185     {
186         int fd[2];
187         pid_t pid;
188
189         /*create a pipe*/
190         if(pipe(fd)==-1)
191         {
192             fprintf(stderr,"Pipe failed");
193             return 1;
194         }
195
196         pid=fork();
197
198         if(pid>0)//parent process(actually child process of the initial
one)
199         {
200
201             for(int i=pipe_position;i<MAX_LINE/2+1;++i)
202             {
203                 free(args[i]);
204                 args[i]=NULL;
205             }
206             close(fd[READ_END]);
207
208             dup2(fd[WRITE_END],STDOUT_FILENO);
209
210             execvp(args[0],args);
211
212         }
213         else if(pid==0)//grandson process
214         {
215             strcpy(args[0],args[pipe_position+1]);
216             for(int i=1;i<MAX_LINE/2+1;++i)
217             {
218                 free(args[i]);
219                 args[i]=NULL;
220             }
221             close(fd[WRITE_END]);
222             dup2(fd[READ_END],STDIN_FILENO);
223             execvp(args[0],args);
224         }
225     }

```

```

226     else // father process
227     {
228         wait(NULL);
229     }
230     return 0;
231 }
232
233 int NONEpipeDealer(char** args, char* last_parameter, int* should_run){
234     pid_t pid;
235     pid=fork();
236
237     if (pid==0) //child process
238     {
239         /*
240         redirection output >
241         */
242         redirect_output(args);
243         redirect_input(args, last_parameter);
244         execvp(args[0], args);
245         *should_run=0;
246     }
247     else //parent process
248     {
249         if(strcmp(last_parameter, "&&")!=0)
250             wait(NULL);
251     }
252
253     return 0;
254 }
255
256 void token_initAndGet(char** args, char** history_buffer, char* Instruction,
char* last_parameter){
257     for(int i=0; i<MAX_LINE/2+1; ++i)
258         args[i]=NULL;
259
260     printf("osh>");
261     fflush(stdout);
262     fgets(Instruction, MAX_LINE, stdin);
263     init_args(Instruction, args, last_parameter);
264     init_history(history_buffer, args, last_parameter);
265     free(Instruction);
266     free(last_parameter);
267 }
268
269 void clearAll(char** args, char** history_buffer){
270     for(int i=0; i<MAX_LINE/2+1; i++)
271         free(args[i]);

```

```

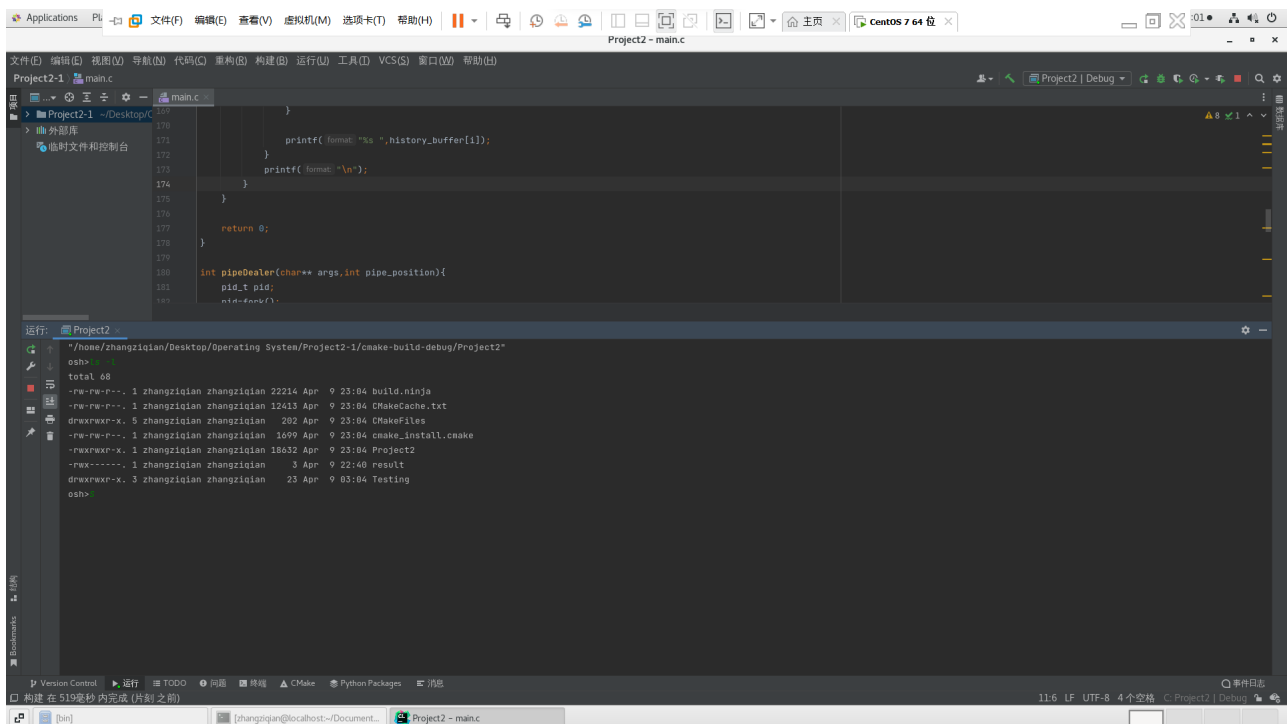
272     for(int i=0;i<MAX_LINE/2+1;i++)
273         free(history_buffer[i]);
274
275
276 }

```

### 0.1.1.7 七、演示的效果图

我使用的是CLion IDE。所以这里直接构建并且运行。

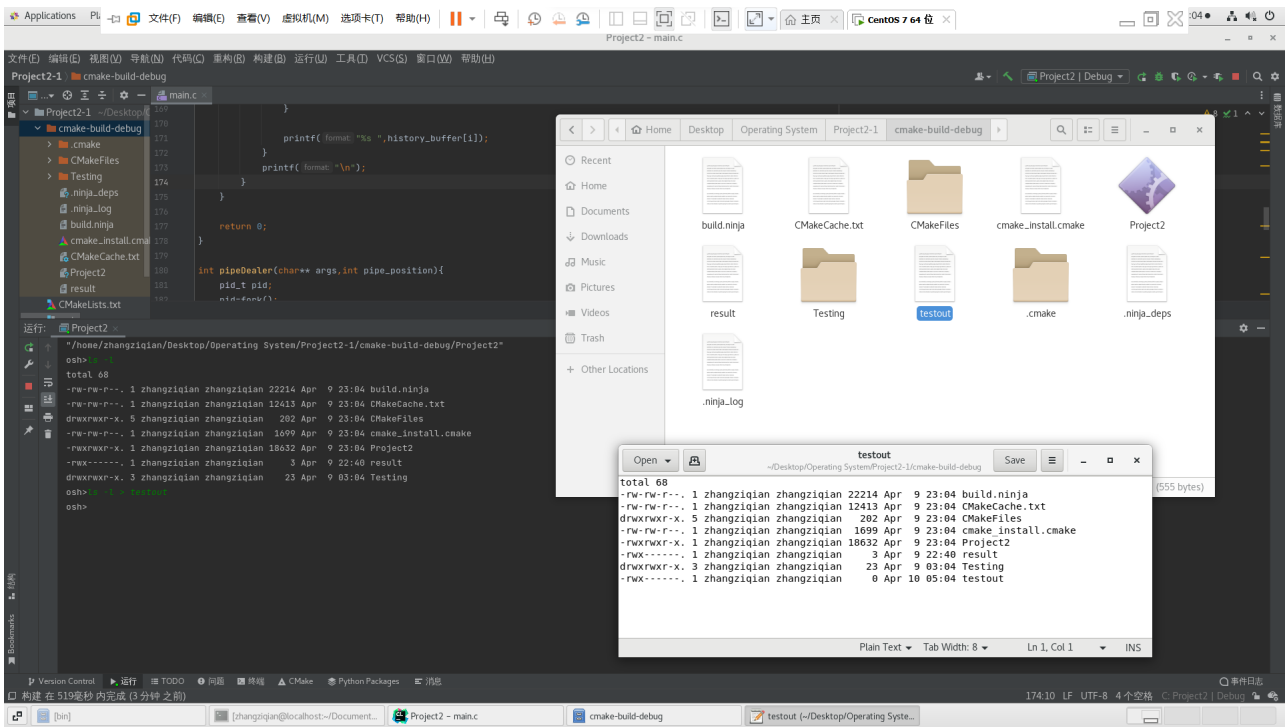
#### 0.1.1.7.1 1、输入命令的演示



- 可以看到可以识别命令并且正常输出

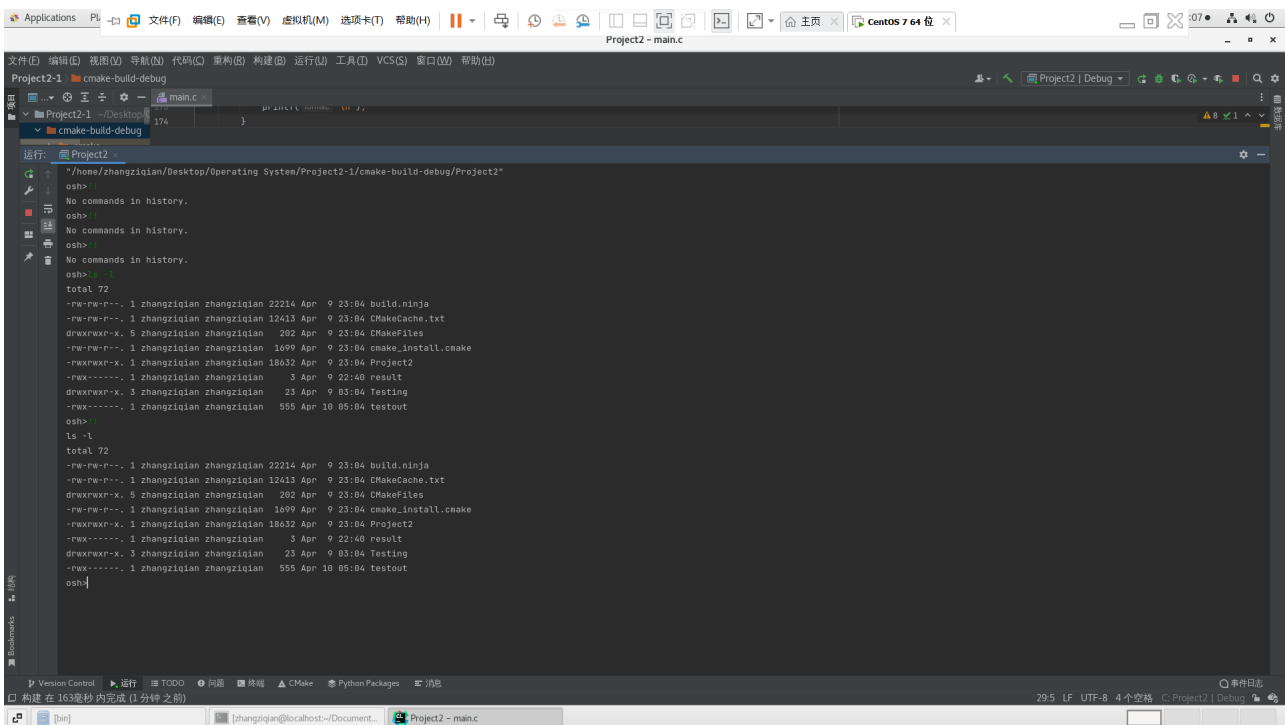
#### 0.1.1.7.2 2、重定向的演示





- 可以看到，我执行了 `ls -l > testout`，在文件的根目录中就会创建一个相对应名称的文件
- 打开文件可以看到原本输出的内容变成了文件中的内容
- 控制台上空白，没有任何内容输出。

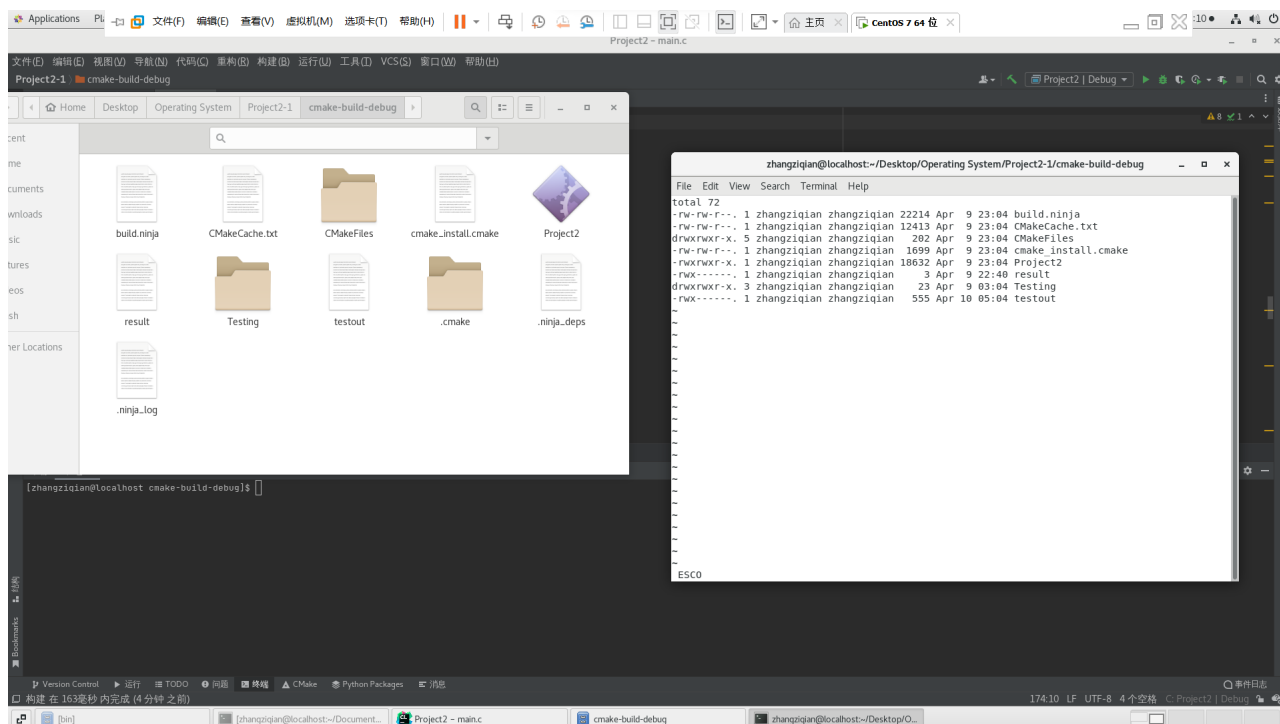
### 0.1.1.7.3 3、历史功能的演示



- 可以看到如果开始就执行 `!!`，就会提示没有历史命令
- 同样的，如果我们连续执行两次回退历史的命令，最终的效果也是提示没有历史命令的。
- 最后，我们执行了 `ls -l` 命令，然后执行 `!!`，可以看到，上一次的命令被正常输出。

#### 0.1.1.7.4 4、管道功能的演示

- 执行 `le -l | less`，可以看到如下结果。说明管道功能正常，成功的将第一个运行的参数传递给第二个
- 编辑页面打开，在靠右侧的编辑框



#### 0.1.2 任务组二：用于任务信息的 Linux 内核模块

在这个项目中，您将编写一个 Linux 内核模块，该模块使用 `/proc` 文件系统根据其进程标识符值 `pid` 显示任务的信息。在开始这个项目之前，请确保您已经完成了第 2 章中的 Linux 内核模块编程项目，其中涉及在 `/proc` 文件系统中创建一个条目。该项目将涉及将进程标识符写入文件 `/proc/pid`。一旦 `pid` 被写入 `/proc` 文件，随后从 `/proc/pid` 读取将报告 (1) 任务正在运行的命令，(2) 任务 `pid` 的值，以及 (3) 当前状态 任务。

加载到系统后如何访问内核模块的示例如下：

```
1 echo "1395" > /proc/pid
2 cat /proc/pid
3 command = [bash] pid = [1395] state = [1]
```

`echo` 命令将字符“1395”写入 `/proc/pid` 文件。您的内核模块将读取此值并存储其等效整数，因为它表示进程标识符。 `cat` 命令从 `/proc/pid` 读取，您的内核模块将从与 `pid` 值为 1395 的任务关联的任务结构中检索三个字段。

```

1 ssize_t proc write(struct file *file, char _user *usr_buf, size_t count,
  loff_t *pos)
2 {
3     int rv = 0;
4     char *k_mem;
5     /* allocate kernel memory */
6     k_mem = kmalloc(count, GFP_KERNEL);
7     /* copies user space usr_buf to kernel memory */
8     copy_from_user(k_mem, usr_buf, count);
9     printk(KERN_INFO "%s\n", k_mem);
10    /* return kernel memory */
11    kfree(k_mem);
12    return count;
13 }

```

#### 0.1.2.1 一、写入到 `/proc` 文件系统

- 在第 2 章的内核模块项目中，我们学习了如何从 `/proc` 文件系统中读取。我们现在介绍如何写入 `/proc`。将结构文件操作中的字段“.write”设置为

```

1 .write = proc write

```

- 从而使得当对 `/proc/pid` 进行写操作时，会调用图 3.37 的 `proc write()` 函数。
- `kmalloc()` 函数是用于分配内存的用户级 `malloc()` 函数的内核等效函数，除了分配内核内存。GFP\_KERNEL 标志表示例行内核内存分配。user() 函数的 copy 将 `usr_buf` 的内容（包含已写入 `/proc/pid` 的内容）复制到最近分配的内核内存。内核模块必须使用具有签名的内核函数 `kstrtol()` 来获取此值的整数等价物。
- 这会将 `str` 的等价字符存储为 `res` 的基数。最后，请注意，我们通过调用 `kfree()` 将先前使用 `kmalloc()` 分配的内存返回给内核。仔细的内存管理（包括释放内存以防止内存泄漏）在开发内核级代码时至关重要。

#### 0.1.2.2 二、从 `/proc` 文件系统读取

- 一旦存储了进程标识符，从 `/proc/pid` 读取的任何内容都将返回命令的名称、进程标识符和状态。如 3.1 节所示，Linux 中的 PCB 由结构体 `task_struct` 表示，该结构体 `task_struct` 位于 `<linux/sched.h>` 包含文件中。给定进程标识符，函数 `pid_task()` 返回关联的任务结构。该函数的签名如下所示：

```

1 struct task_struct pid_task(struct pid *pid,
2     enum pid_type type)

```

#### 0.1.2.3 三、原始代码

```

1 /**
2  * Kernel module that communicates with /proc file system.
3  *

```

```

4  * This provides the base logic for Project 2 - displaying task information
5  */
6
7  #include <linux/init.h>
8  #include <linux/slab.h>
9  #include <linux/sched.h>
10 #include <linux/module.h>
11 #include <linux/kernel.h>
12 #include <linux/proc_fs.h>
13 #include <linux/vmalloc.h>
14 #include <asm/uaccess.h>
15
16 #define BUFFER_SIZE 128
17 #define PROC_NAME "pid"
18
19 /* the current pid */
20 static long l_pid;
21
22 /**
23  * Function prototypes
24  */
25 static ssize_t proc_read(struct file *file, char *buf, size_t count, loff_t
*pos);
26 static ssize_t proc_write(struct file *file, const char __user *usr_buf,
size_t count, loff_t *pos);
27
28 static struct file_operations proc_ops = {
29     .owner = THIS_MODULE,
30     .read = proc_read,
31     .write = proc_write,
32 };
33
34 /* This function is called when the module is loaded. */
35 static int proc_init(void)
36 {
37     // creates the /proc/procfs entry
38     proc_create(PROC_NAME, 0666, NULL, &proc_ops);
39
40     printk(KERN_INFO "/proc/%s created\n", PROC_NAME);
41
42     return 0;
43 }
44
45 /* This function is called when the module is removed. */
46 static void proc_exit(void)
47 {
48     // removes the /proc/procfs entry

```

```

49     remove_proc_entry(PROC_NAME, NULL);
50
51     printk( KERN_INFO "/proc/%s removed\n", PROC_NAME);
52 }
53
54 /**
55  * This function is called each time the /proc/pid is read.
56  *
57  * This function is called repeatedly until it returns 0, so
58  * there must be logic that ensures it ultimately returns 0
59  * once it has collected the data that is to go into the
60  * corresponding /proc file.
61  */
62 static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t
count, loff_t *pos)
63 {
64     int rv = 0;
65     char buffer[BUFFER_SIZE];
66     static int completed = 0;
67     struct task_struct *tsk = NULL;
68
69     if (completed) {
70         completed = 0;
71         return 0;
72     }
73
74     tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
75     if (tsk==NULL) {
76         printk(KERN_INFO "invalid pid!\n");
77         return 0;
78     }
79     completed = 1;
80     rv=sprintf(buffer,"command = [%s] pid=[%ld] state=[%ld]\n",tsk-
>comm,l_pid,tsk->state);
81     // copies the contents of kernel buffer to userspace usr_buf
82     copy_to_user(usr_buf, buffer, rv);
83
84     return rv;
85 }
86
87 /**
88  * This function is called each time we write to the /proc file system.
89  */
90 static ssize_t proc_write(struct file *file, const char __user *usr_buf,
size_t count, loff_t *pos)
91 {
92     char *k_mem;

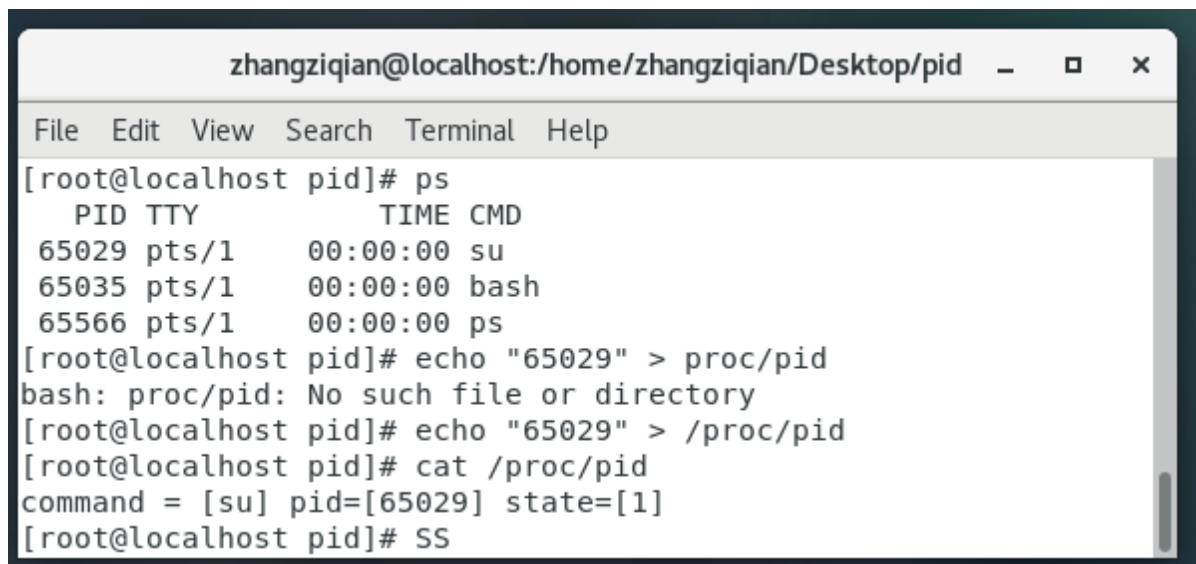
```

```

93
94     // allocate kernel memory
95     k_mem = kmalloc(count, GFP_KERNEL);
96
97     /* copies user space usr_buf to kernel buffer */
98     if (copy_from_user(k_mem, usr_buf, count)) {
99         printk( KERN_INFO "Error copying from user\n");
100         return -1;
101     }
102
103     /**
104      * kstrol() will not work because the strings are not guaranteed
105      * to be null-terminated.
106      *
107      * sscanf() must be used instead.
108      */
109     printk("%s", k_mem);
110     sscanf(k_mem, "%ld", &l_pid);
111
112     kfree(k_mem);
113
114     return count;
115 }
116
117 /* Macros for registering module entry and exit points. */
118 module_init( proc_init );
119 module_exit( proc_exit );
120
121 MODULE_LICENSE("GPL");
122 MODULE_DESCRIPTION("Module");
123 MODULE_AUTHOR("SGG");
124

```

- 演示效果图

A terminal window titled 'zhangziqian@localhost:/home/zhangziqian/Desktop/pid' with standard window controls. The terminal shows a sequence of commands and their outputs. First, 'ps' is run, displaying a table of running processes. Then, an attempt is made to create a file 'proc/pid' using 'echo', which fails with a 'No such file or directory' error. Next, the user navigates to '/proc/pid' and runs 'cat', which displays the details of the process with PID 65029. Finally, the 'SS' command is entered.

```
zhangziqian@localhost:/home/zhangziqian/Desktop/pid
File Edit View Search Terminal Help
[root@localhost pid]# ps
  PID TTY          TIME CMD
 65029 pts/1        00:00:00 su
 65035 pts/1        00:00:00 bash
 65566 pts/1        00:00:00 ps
[root@localhost pid]# echo "65029" > proc/pid
bash: proc/pid: No such file or directory
[root@localhost pid]# echo "65029" > /proc/pid
[root@localhost pid]# cat /proc/pid
command = [su] pid=[65029] state=[1]
[root@localhost pid]# SS
```

### 0.1.3 任务小结

- 第一个项目 UNIX Shell 让我对 Shell 有更深入的了解，难度也设置的恰到好处，四个功能层次递进，让我们一步步的实现这功能。更重要的是，我们可以学习到练习使用管道和重定向操作。
- 第二个项目 Linux Kernel Module for Task Information 深入到 Linux /proc 系统，让我复习了进程控制堵塞的知识。
- 这两个项目都很有趣。通过完成这两个项目，我对C语言的一些要点有了更深刻的认识，尽管在C语言的运用和编写上与C++的熟练程度还有很大的差距。此外通过这个项目复习了我们在课堂上学习的知识，对一些基本的概念有了更深入的了解。