

# Proj2 UNIX Shell Programming & Linux Kernel Module for Task Information

---

by 郑航 520021911347

## Proj2 UNIX Shell Programming & Linux Kernel Module for Task Information

- 1 Introduction
  - 1.1 Objectives
  - 1.2 Environment
- 2 Project 1—UNIX Shell
  - 2.1 Abstract
  - 2.2 功能要求
  - 2.3 Creating the child process and executing the command in the child
    - 2.3.1 读入指令并分词存储于args中
    - 2.3.2 判断是否是exit指令
    - 2.3.3 创建子进程，将参数传入并调用execvp函数
    - 2.3.4 执行结果
  - 2.4 Providing a history feature
    - 2.4.1 存储历史指令
    - 2.4.2 !! 指令的实现
    - 2.4.3 执行结果
  - 2.5 Adding support of input and output redirection
    - 2.5.1 重定向的判断：
    - 2.5.2 重定向的执行
    - 2.5.3 实现的改进
    - 2.5.4 执行结果
  - 2.6 pipe通信
    - 2.6.1 pipe的准备
    - 2.6.2 pipe的执行
    - 2.6.3 执行结果
  - 2.7 simple\_shell.c完整代码
  - 2.8 difficulty&summary
- 3 Project 2 — Linux Kernel Module for Task Information
  - 3.1 Abstract
  - 3.2 proc\_write
  - 3.3 proc\_read
  - 3.4 执行结果
  - 3.5 pid.c 完整代码
  - 3.6 difficulty

## 1 Introduction

---

## 1.1 Objectives

- 初步了解UNIX shell的运作，并实现一个简单的shell
- 进一步学习内核编程，实现一个内核模块，可根据pid查询并返回对应进程的信息

## 1.2 Environment

- win10下的VMware Workstation Pro中运行的Ubuntu18.04

# 2 Project 1—UNIX Shell

---

## 2.1 Abstract

- 编写一个简单的Unix shell，完成了在子进程中执行指令、存储/执行历史指令、输入输出重定向、通过pipe实现父子进程交流的功能
- 尝试使用了一系列系统调用，如fork(), execvp(), wait(), dup2(), and pipe()等

## 2.2 功能要求

we need to implement several parts:

- Creating the child process and executing the command in the child
- Providing a history feature
- Adding support of input and output redirection
- Allowing the parent and child processes to communicate via a pipe

## 2.3 Creating the child process and executing the command in the child

### 2.3.1 读入指令并分词存储于args中

- 首先要将args中的指针全部清空，便于后续判断指令是否已读空，排除之前指令的干扰
- 然后循环将字符串按空格分隔读入一个暂时存储指令的input\_tmp字符串数组中，再逐个参数赋值给args，期间每次存完一个参数，就读入下一个字符判断其是否为'\n'，是的话说明指令已读完，退出循环。

```
1 //清空args
2 for (int i = 0; i < MAX_LINE / 2 + 1; i++)
3 {
4     args[i] = NULL;
5 }
6
7 //读入指令
8 char ch;
9 while (scanf("%s", input_tmp[arg_count]))
10 {
11     args[arg_count] = input_tmp[arg_count];
12     arg_count++;
13     scanf("%c", &ch);
```

```

14     if (ch == '\n')
15         break;
16 }

```

### 2.3.2 判断是否是exit指令

若读入的指令是exit，则无需进行后续操作，将should\_run设置为0，跳出循环，终止shell程序

```

1  if(strcmp(args[0], "exit")==0) {
2      should_run=0;
3      continue;
4  }

```

### 2.3.3 创建子进程，将参数传入并调用execvp函数

- 进行fork前，先判断是否有&参数，用wait\_flag变量进行记录，有的话需要将"&"参数删去
- 利用课内给的创建新进程的代码框架，调用fork函数创建新进程，并判断父子进程
- 在子进程内，将args中的参数传入并调用execvp函数
- 在父进程内，根据wait\_flag状态决定是否调用wait函数

```

1  //判断wait
2  if (strcmp(args[count], "&") == 0)
3  {
4      args[count] = NULL;
5      count--;
6      wait_flag=0;
7  }
8  //创建新进程
9  pid_t pid;
10 pid = fork();
11 if (pid < 0) {
12     fprintf(stderr, "Fork Failed");
13     return 1;
14 }
15 else if (pid == 0) //child
16 {
17     execvp(args[0], args);
18 }
19 else //parent
20 {
21     if (wait_flag)
22     {
23         wait(NULL);
24     }
25 }

```

### 2.3.4 执行结果

```

zh@ubuntu:~/project/pro2$ ./simple-shell
osh>ls
simple-shell  simple-shell.c
osh>ls &
osh>simple-shell  simple-shell.c

```

## 2.4 Providing a history feature

### 2.4.1 存储历史指令

- 使用一个和args规格相同的字符串数组history\_cmd，在每个指令执行周期的最后对该指令进行存储
- 设置一个int变量have\_history来标记是否有历史指令
- 该过程中需要用到一个临时的字符串指针数组tmp\_str，原因是history\_cmd和args都是字符串指针，所以两者所包含的内容不同的话其地址也必然不同（此例中分别是input\_tmp和tmp\_str）

history\_cmd的初始化：

```
1  if (!have_history)
2  {
3      for (int i = 0; i < MAX_LINE / 2 + 1; i++)
4      {
5          history_cmd[i] = NULL;
6      }
7  }
```

history\_cmd和have\_history的赋值：

```
1  for (int i = 0; i < arg_count; i++)
2  {
3      strcpy(tmp_str[i], args[i]);
4      history_cmd[i] = tmp_str[i];
5  }
6  history_arg_count = arg_count;
7  have_history = 1;
```

### 2.4.2 !! 指令的实现

- 在判断完“exit”后，即可以开始判断是否为“!!”
- 执行过程中，首先判断是否有历史指令，没有的话直接输出错误信息；否则，一边将history\_cmd复制到args中，一边将其打印出来

```
1  //!!的实现
2  if (strcmp(args[0], "!!") == 0)
3  {
4      if (!have_history) //没有历史命令
5      {
6          printf("No commands in history!\n");
7          continue;
8      }
9      else
10     {
11         printf("osh>");
12         fflush(stdout);
13         for (int i = 0; i < history_arg_count; i++)
14         {
15             strcpy(tmp_str[i], history_cmd[i]);
16             args[i] = tmp_str[i];
17             printf("%s", args[i]);
18             printf("%s", " ");
19         }
```

```

20     printf("%c", '\n');
21     arg_count = history_arg_count;
22 }
23 }

```

### 2.4.3 执行结果

```

zh@ubuntu:~/project/pro2$ ./simple-shell
osh>!!
No commands in history!
osh>ls
simple-shell  simple-shell.c
osh>!!
osh>ls
simple-shell  simple-shell.c
osh>!!
osh>ls
simple-shell  simple-shell.c

```

## 2.5 Adding support of input and output redirection

### 2.5.1 重定向的判断:

- 设置一个int变量redir\_flag来表示是否需要重定向
- 假如需要重定向，将文件名存储于filename中，并将重定向参数从args中删去

```

1  //判断重定向
2  if (arg_count > 1 && strcmp(args[arg_count-2], ">")==0)
3  {
4      redir_flag=1;    //redir_flag=1表示输出重定向
5      filename=args[arg_count-1];
6      args[arg_count-1]=NULL;
7      args[arg_count-2]=NULL;
8      arg_count-=2;
9  }
10 else if (arg_count > 1 && strcmp(args[arg_count-2], "<")==0){
11     redir_flag=-1;    //redir_flag=1表示输出重定向
12     filename=args[arg_count-1];
13     args[arg_count-1]=NULL;
14     args[arg_count-2]=NULL;
15     arg_count-=2;
16 }

```

### 2.5.2 重定向的执行

- 在子进程中进行输入输出的重定向，由于有了前期的redir\_flag和filename的准备，此时只需要简单调用dup2函数进行重定向即可

```

1  //重定向执行
2  int fd;
3  if (redir_flag==1){    //输出重定向
4      fd=open(filename,O_CREAT|O_RDWR,S_IRWXU);

```

```

5     if(fd==-1){
6         fprintf(stderr, "Create file Failed\n");
7         continue;
8     }
9     dup2(fd, STDOUT_FILENO);
10 }
11 else if(redir_flag==1){    //输入重定向
12     fd = open(filename,O_CREAT|O_RDONLY,S_IRUSR);
13     dup2(fd, STDIN_FILENO);
14 }

```

### 2.5.3 实现的改进

- 一开始只用了一个flag和filename，只能支持输入或输出重定向，但后来思考后发现，只需要适当安排一下顺序，并且按如下方式进行实现，可以同时支持输入输出的重定向

```

1 //判断重定向
2 if (arg_count > 1 && strcmp(args[arg_count-2], ">")==0)
3 {
4     redir_out_flag=1;    //redir_out_flag=1表示输出重定向
5     out_file=args[arg_count-1];
6     args[arg_count-1]=NULL;
7     args[arg_count-2]=NULL;
8     arg_count-=2;
9 }
10 if(arg_count > 1 && strcmp(args[arg_count-2], "<")==0){
11     redir_in_flag=1;    //redir_in_flag=1表示输出重定向
12     filename=args[arg_count-1];
13     args[arg_count-1]=NULL;
14     args[arg_count-2]=NULL;
15     arg_count-=2;
16 }

```

```

1 //重定向执行
2 int fd;
3 if(redir_out_flag==1){    //输出重定向
4     fd=open(out_file,O_CREAT|O_RDWR,S_IRWXU);
5     if(fd==-1){
6         fprintf(stderr, "Create file Failed\n");
7         continue;
8     }
9     dup2(fd, STDOUT_FILENO);
10 }
11 if(redir_in_flag==1){    //输入重定向
12     fd = open(in_file,O_CREAT|O_RDONLY,S_IRUSR);
13     dup2(fd, STDIN_FILENO);
14 }

```

### 2.5.4 执行结果

```
osh>sort < in.txt
123
235
24
325
663
756
osh>ls > out.txt
osh>cat out.txt
in.txt
out.txt
simple-shell
simple-shell.c
osh>sort < in.txt > out.txt
osh>cat out.txt
123
235
24
325
663
756
```

## 2.6 pipe通信

### 2.6.1 pipe的准备

- 检查命令中是否有“|”参数，使用pipe\_flag进行记录
- 由于需要将命令拆分为两条子命令分别在两个进程中执行，因此还需要对args进行拆分，并将第二条指令用字符串指针数组pipe\_arg进行存储
- 存储过程中，可以复用之前args中的地址，不需要额外的地址空间，只需要将pipe\_arg中各项指向原先args各项指向的地址即可，并将args第二条命令的部分置NULL

```
1  //判断pipe
2  for (int i = 0; i < arg_count; i++)
3  {
4      if (strcmp(args[i], "|") == 0)
5      {
6          pipe_flag = 1;
7          pipe_count = arg_count - i - 1;
8          arg_count = i;
9          for (int j = 0; j < MAX_LINE / 2 + 1; j++) //清空pipe_arg
10         {
11             pipe_arg[j] = NULL;
12         }
13         for (int j = 0; j < pipe_count; j++) //将原命令进行切分
14         {
15             pipe_arg[j] = args[i + 1];
16             args[i + 1] = NULL;
17         }
18         args[i] = NULL;
```

```

19     break;
20 }
21 }

```

## 2.6.2 pipe的执行

- 在子进程中进行fork，在child中进行前一条命令的执行，并将输出通过管道传给parent，作为parent中命令的输入
- 注意管道的用法，管道的参数pipe\_fd，0代表读端，1代表写端，读端和写端分别需要调用close关闭另一个端口

```

1  //执行pipe
2  if (pipe_flag == 1)
3  {
4      int pipe_result = pipe(pipe_fd);           //pipe的创建
5      if (pipe_result == -1)
6      {
7          printf("Pipe Create Failed!\n");
8      }
9
10     pid_t pipe_pid = fork();                    //创建新进程
11     if (pipe_pid < 0)
12     {
13         printf("Pipe Fork Failed\n");
14         return 1;
15     }
16     else if (pipe_pid == 0)                    //child, 执行第一条命令并写入pipe
17     {
18         close(pipe_fd[0]);
19         dup2(pipe_fd[1], STDOUT_FILENO);
20         execvp(args[0], args);
21         exit(0);
22     }
23     else                                        //parent, 从pipe中读取输入, 并执行第
二条命令
24     {
25         close(pipe_fd[1]);
26         dup2(pipe_fd[0], STDIN_FILENO);
27         execvp(pipe_arg[0], pipe_arg);
28         wait(NULL);
29     }
30 }

```

## 2.6.3 执行结果

```

osh>cat in.txt | sort
123
235
24
325
663
756

```



## 2.7 simple\_shell.c完整代码

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6  #include <fcntl.h>
7  #include <stdlib.h>
8  #include <signal.h>
9
10 #define MAX_LINE 80 /* 80 chars per line, per command */
11
12 int main(void)
13 {
14     char *args[MAX_LINE / 2 + 1]; /* command line (of 80) has max of 40
arguments */
15     int should_run = 1;
16     char input_tmp[MAX_LINE / 2 + 1][20];
17     int arg_count = 0;
18     char *history_cmd[MAX_LINE / 2 + 1];
19     int have_history = 0;
20     int history_arg_count = 0;
21     int wait_flag = 1;
22     char tmp_str[MAX_LINE / 2 + 1][20];
23     char *out_file;
24     char *in_file;
25     int redir_out_flag = 0;
26     int redir_in_flag = 0;
27     int pipe_flag = 0;
28     int pipe_count = 0;
29     char *pipe_arg[MAX_LINE / 2 + 1];
30     int pipe_fd[2];
31
32     while (should_run)
33     {
34         printf("osh>");
35         fflush(stdout);
36         arg_count = 0;
37         redir_out_flag = 0;
38         redir_in_flag = 0;
39         wait_flag = 1;
40         pipe_flag = 0;
41
42         //初始化清空args
43         for (int i = 0; i < MAX_LINE / 2 + 1; i++)
44         {
45             args[i] = NULL;
46         }
47         if (!have_history)
48         {
49             for (int i = 0; i < MAX_LINE / 2 + 1; i++)
50             {
51                 history_cmd[i] = NULL;
52             }
53         }
54         //读入指令
```

```

55     char ch;
56     while (scanf("%s", input_tmp[arg_count]))
57     {
58         args[arg_count] = input_tmp[arg_count];
59         arg_count++;
60         scanf("%c", &ch);
61         if (ch == '\n')
62             break;
63     }
64
65     //判断exit
66     if (strcmp(args[0], "exit") == 0)
67     {
68         should_run = 0;
69         continue;
70     }
71     //判断并执行 !!
72     if (strcmp(args[0], "!!") == 0)
73     {
74         if (!have_history)
75         {
76             printf("No commands in history!\n");
77             continue;
78         }
79         else
80         {
81             printf("osh>");
82             fflush(stdout);
83             for (int i = 0; i < history_arg_count; i++)
84             {
85                 strcpy(tmp_str[i], history_cmd[i]);
86                 args[i] = tmp_str[i];
87                 printf("%s", args[i]);
88                 printf("%s", " ");
89             }
90             printf("%c", '\n');
91             arg_count = history_arg_count;
92         }
93     }
94     else
95     {
96         for (int i = 0; i < arg_count; i++)
97         {
98             strcpy(tmp_str[i], args[i]);
99             history_cmd[i] = tmp_str[i];
100         }
101         history_arg_count = arg_count;
102         have_history = 1;
103     }
104     //判断wait
105     if (strcmp(args[arg_count - 1], "&") == 0)
106     {
107         args[arg_count - 1] = NULL;
108         arg_count--;
109         wait_flag = 0;
110     }
111
112     //判断重定向

```

```

113     if (arg_count > 1 && strcmp(args[arg_count - 2], ">") == 0)
114     {
115         redir_out_flag = 1; // redir_out_flag=1表示输出重定向
116         out_file = args[arg_count - 1];
117         args[arg_count - 1] = NULL;
118         args[arg_count - 2] = NULL;
119         arg_count -= 2;
120     }
121     if (arg_count > 1 && strcmp(args[arg_count - 2], "<") == 0)
122     {
123         redir_in_flag = 1; // redir_in_flag=1表示输出重定向
124         in_file = args[arg_count - 1];
125         args[arg_count - 1] = NULL;
126         args[arg_count - 2] = NULL;
127         arg_count -= 2;
128     }
129
130     //判断pipe
131     for (int i = 0; i < arg_count; i++)
132     {
133         if (strcmp(args[i], "|") == 0)
134         {
135             pipe_flag = 1;
136             pipe_count = arg_count - i - 1;
137             arg_count = i;
138             for (int j = 0; j < MAX_LINE / 2 + 1; j++)
139             {
140                 pipe_arg[j] = NULL;
141             }
142             for (int j = 0; j < pipe_count; j++)
143             {
144                 pipe_arg[j] = args[i + 1];
145                 args[i + 1] = NULL;
146             }
147             args[i] = NULL;
148             break;
149         }
150     }
151
152     //创建新进程
153     pid_t pid;
154     pid = fork();
155     if (pid < 0)
156     {
157         printf("Fork Failed\n");
158         return 1;
159     }
160     else if (pid == 0) // child
161     {
162         //重定向执行
163         int fd;
164         if (redir_out_flag == 1)
165         { //输出重定向
166             fd = open(out_file, O_CREAT | O_RDWR, S_IRWXU);
167             if (fd == -1)
168             {
169                 printf("Create file Failed\n");
170                 continue;

```

```

171     }
172     dup2(fd, STDOUT_FILENO);
173 }
174 if (redir_in_flag == 1)
175 { //输入重定向
176     fd = open(in_file, O_CREAT | O_RDONLY, S_IRUSR);
177     dup2(fd, STDIN_FILENO);
178 }
179
180 //执行pipe
181 if (pipe_flag == 1)
182 {
183     int pipe_result = pipe(pipe_fd);
184     if (pipe_result == -1)
185     {
186         printf("Pipe Create Failed!\n");
187     }
188
189     pid_t pipe_pid = fork();
190     if (pipe_pid < 0)
191     {
192         printf("Pipe Fork Failed\n");
193         return 1;
194     }
195     else if (pipe_pid == 0)
196     {
197         close(pipe_fd[0]);
198         dup2(pipe_fd[1], STDOUT_FILENO);
199         execvp(args[0], args);
200         exit(0);
201     }
202     else
203     {
204         close(pipe_fd[1]);
205         dup2(pipe_fd[0], STDIN_FILENO);
206         execvp(pipe_arg[0], pipe_arg);
207         wait(NULL);
208     }
209 }
210 else
211 {
212     execvp(args[0], args);
213 }
214 exit(0);
215 }
216 else // parent
217 {
218     if (wait_flag)
219     {
220         wait(NULL);
221     }
222 }
223 /**
224  * After reading user input, the steps are:
225  * (1) fork a child process
226  * (2) the child process will invoke execvp()
227  * (3) if command includes &, parent and child will run
228 concurrently

```

```

228         */
229     }
230
231     return 0;
232 }
233

```

## 2.8 difficulty&summary

- 一开始不太习惯在Linux下进行编程和调试，花费一些时间学习了本机vscode远程连接虚拟机以及vscode的环境配置
- 课本对一系列系统调用并没有详细的介绍，需要学习一下其具体的参数和用法
- pipe的创建要在fork后的子进程中进行，一开始是在父进程中提前创建了pipe，运行后发现无法传递数据，父进程一直处于wait的状态；后来查询了一下资料后将该过程放在子进程中进行，问题得到了解决

## 3 Project 2 — Linux Kernel Module for Task Information

### 3.1 Abstract

- 实现一个内核模块，可根据pid查询并返回进程的command、pid和state
- pid.c文件主体已经给出，需要完善proc\_read和proc\_write两个函数
- 通过以下方式将待查询pid输入

```
1 echo "2" > /proc/pid
```

- 通过以下方式输出信息

```
1 cat /proc/pid
```

- 由于是内核代码，因此需要编写和project 1 类似的Makefile文件帮助编译

### 3.2 proc\_write

- 根据课本所给的实现思路以及代码框架，理解每个部分的功能后，只需要完善将从用户缓冲区中copy的数据拷贝到l\_pid中进行存储这个功能即可
- 该实现需要把一个字符指针所指的内容转换并存储于long int类型的l\_pid中，课本上介绍的实现函数是kstrol()，但代码中补充说kstrol()函数使用有风险，可以使用sscanf()函数替代

函数实现如下：

```

1 static ssize_t proc_write(struct file *file, const char __user *usr_buf,
2 size_t count, loff_t *pos)
3 {
4     char *k_mem;
5
6     // allocate kernel memory
7     k_mem = kmalloc(count, GFP_KERNEL);
8
9

```

```

7
8     /* copies user space usr_buf to kernel buffer */
9     if (copy_from_user(k_mem, usr_buf, count))
10    {
11        printk(KERN_INFO "Error copying from user\n");
12        return -1;
13    }
14
15    /**
16     * kstrol() will not work because the strings are not guaranteed
17     * to be null-terminated.
18     *
19     * sscanf() must be used instead.
20     */
21    sscanf(k_mem, "%ld", &l_pid);
22
23    kfree(k_mem);
24
25    return count;
26 }

```

### 3.3 proc\_read

- 根据课本的实现思路和代码框架，还需要增加如下的部分：①判断tsk的返回值；②将tsk的信息存于内核buffer中；③若信息读取、转存成功，将completed置为1
- tsk的返回值如是NULL的话说明查询失败，此时函数需要返回0
- tsk的信息存于buffer，可类比proc\_write中的sscanf，使用对应的sprintf函数，并将返回值存于proc\_read函数返回值rv中，表示读取是否成功
- completed置1应该在整个函数的最后进行

函数实现如下：

```

1  static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t
count, loff_t *pos)
2  {
3      int rv = 0;
4      char buffer[BUFFER_SIZE];
5      static int completed = 0;
6      struct task_struct *tsk = NULL;
7
8      if (completed)
9      {
10         completed = 0;
11         return 0;
12     }
13
14     tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
15     if (tsk == NULL)
16         return 0;
17
18     rv = sprintf(buffer, "command = [%s], pid = [%ld], state = [%ld]\n",
19                 tsk->comm, tsk->pid, tsk->state);
20
21     // copies the contents of kernel buffer to userspace usr_buf
22     if (copy_to_user(usr_buf, buffer, rv))
23     {

```

```

24         rv = -1;
25     }
26
27     completed = 1;
28
29     return rv;
30 }

```

### 3.4 执行结果

```

zh@ubuntu:~/project/pro2/kernel_module$ ps
  PID TTY          TIME CMD
 2004 pts/0    00:00:00 bash
 2670 pts/0    00:00:00 ps
zh@ubuntu:~/project/pro2/kernel_module$ echo "2004" > /proc/pid
zh@ubuntu:~/project/pro2/kernel_module$ cat /proc/pid
command = [bash], pid = [2004], state = [1]
zh@ubuntu:~/project/pro2/kernel_module$ echo "12" > /proc/pid
zh@ubuntu:~/project/pro2/kernel_module$ cat /proc/pid
command = [migration/0], pid = [12], state = [1]

```

### 3.5 pid.c 完整代码

```

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 #include <linux/uaccess.h>
16
17 #define BUFFER_SIZE 128
18 #define PROC_NAME "pid"
19
20 /* the current pid */
21 static long l_pid;
22
23 /**
24  * Function prototypes
25  */
26 static ssize_t proc_read(struct file *file, char *buf, size_t count, loff_t
*pos);
27 static ssize_t proc_write(struct file *file, const char __user *usr_buf,
size_t count, loff_t *pos);
28
29 static struct file_operations proc_ops = {
30     .owner = THIS_MODULE,
31     .read = proc_read,

```

```

32     .write = proc_write};
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         {
71             completed = 0;
72             return 0;
73         }
74
75         tsk = pid_task(find_vpid(1_pid), PIDTYPE_PID);
76         if (tsk == NULL)
77             return 0;
78
79         rv = sprintf(buffer, "command = [%s], pid = [%ld], state =
[%ld]\n",
80                     tsk->comm, tsk->pid, tsk->state);
81
82         // copies the contents of kernel buffer to userspace usr_buf
83         if (copy_to_user(usr_buf, buffer, rv))
84         {
85             rv = -1;
86         }
87

```



```

88         completed = 1;
89
90         return rv;
91     }
92
93     /**
94      * This function is called each time we write to the /proc file system.
95      */
96     static ssize_t proc_write(struct file *file, const char __user *usr_buf,
97                             size_t count, loff_t *pos)
98     {
99
100         char *k_mem;
101
102         // allocate kernel memory
103         k_mem = kmalloc(count, GFP_KERNEL);
104
105         /* copies user space usr_buf to kernel buffer */
106         if (copy_from_user(k_mem, usr_buf, count))
107         {
108             printk(KERN_INFO "Error copying from user\n");
109             return -1;
110         }
111
112         /**
113          * kstrol() will not work because the strings are not guaranteed
114          * to be null-terminated.
115          *
116          * sscanf() must be used instead.
117          */
118         sscanf(k_mem, "%ld", &l_pid);
119
120         kfree(k_mem);
121
122         return count;
123     }
124
125     /** Macros for registering module entry and exit points. */
126     module_init(proc_init);
127     module_exit(proc_exit);
128
129     MODULE_LICENSE("GPL");
130     MODULE_DESCRIPTION("Proj 2");
131     MODULE_AUTHOR("zheng hang");

```

### 3.6 difficulty

由于是内核态代码，无法直接运行调试，前期出现bug的时候会导致要么terminal直接关闭，要么运行失败且无法rmmod（会显示rmmod ERROR: module pid is in use），只能将虚拟机重启（重启过程会非常缓慢，系统会对一些进程进行处理，并且重启后原来无法删除的内核模块都已经被删除），因此花费了较多的时间调试代码

