

# 操作系统 project-2 实验报告

\* 姓名: 管仁阳 学号:519021911058 邮箱: guanrenyang@sjtu.edu.cn

## 1 实验名称

1. UNIX Shell
2. Linux Kernel Module for Task Information

## 2 实验目的

### 2.1 UNIX Shell:

设计一个 C 语言程序来实现 unix shell 交互界面。它可以接收用户指令并使用独立的进程实现指令。需要能够满足以下需求

1. 使用独立的进程执行独立的指令。
2. 使用!! 来执行历史指令
3. 使用 > 与 < 符号来进行输出、输入的重定向
4. 使用 pipe 来进行进程通信

### 2.2 Linux Kernel Module for Task Information:

1. 创建/proc/pid 目录
2. 使用 echo 命令向/proc/pid 目录中写入进程信息
3. 使用 cat 命令来链取进程信息

## 3 预备知识

### 3.1 UNIX Shell:

#### 3.1.1 execvp(char \*command, char \*params[]) 函数

第一个参数为指令名称, 第二个参数是一个字符串数组, 包括从命令本身开始的所有参数。该函数进入内核态执行对应指令。

#### 3.1.2 fork() 函数

```
1 pid_t pid;  
2 pid=fork();
```

该函数拷贝当前进程的状态作为子进程 (pid=0), 此进程 (父进程) 的 pid=1。

### 3.1.3 int dup2(int oldfd,int newfd) 函数

传入两个文件描述符 (int 型), 讲 oldfd 的值赋值给 newfd。

```
1    dup2( fd ,SEDOUT_FILENO); //用于输出重定向。
2    dup2( fd ,SEDIN_FILENO); //用于输入重定向。
```

### 3.1.4 pipe() 函数

pipe(int fd[2]) 函数传入一个文件描述符数组, 此即进程间 pipe 的读取端和写入端。配合 write() 与 read() 函数完成 ordinary pipe.

```
1    /* now fork a child process */
2    pid = fork();
3
4    if (pid > 0) { /* parent process */
5        /* close the unused end of the pipe */
6        close( fd [READ_END] );
7
8        /* write to the pipe */
9        write( fd [WRITE_END], write_msg, strlen(write_msg)+1);
10
11        /* close the write end of the pipe */
12        close( fd [WRITE_END] );
13    }
14    else { /* child process */
15        /* close the unused end of the pipe */
16        close( fd [WRITE_END] );
17
18        /* read from the pipe */
19        read( fd [READ_END], read_msg, BUFFER_SIZE);
20
21        /* close the write end of the pipe */
22        close( fd [READ_END] );
23    }
```

## 3.2 Linux Kernel Module for Task Information:

### 3.2.1 file\_operations 结构体:

在此项目中需要用到其中的三个参数:

1. *.owner* 一个指向拥有这个结构的模块的指针。
2. *.read* 指定执行文件系统读取的函数。
3. *.write* 指定执行文件系统写入的函数。

### 3.2.2 task\_struct 结构体

task\_struct 结构体中保存每一个进程的 PCB 信息。在此项目中我们需要用到其中的命令 *comm* 与状态 *state* 变量。

### 3.2.3 proc\_init() 函数

/proc 文件系统初始化函数，详细内容见 project-1 的报告。

### 3.2.4 proc\_create() 函数

/proc/PROC\_NAME 目录创建函数，详细内容见 project-1 的报告。

### 3.2.5 remove\_proc\_entry() 函数

/proc/PROC\_NAME 目录删除函数，详细内容见 project-1 的报告。

### 3.2.6 proc\_read() 函数

函数原型:

```
1 static ssize_t proc_read(struct file *file ,
2     char __user *usr_buf, size_t count, loff_t *pos)
```

当/proc/PROC\_NAME 目录在使用 cat 指令读取时调用此函数，当返回值为 0 时正常返回。

### 3.2.7 proc\_write() 函数

函数原型:

```
1 static ssize_t proc_write(struct file *file , const
2     char __user *usr_buf, size_t count, loff_t *pos)
```

当/proc/PROC\_NAME 目录在使用 echo 指令写入时调用此函数，当返回值为 0 时正常返回。

### 3.2.8 kmalloc() 与 kfree() 函数

内核态中的堆空间分配与释放函数，类似于 malloc() 与 free() 函数。

## 4 实验内容

### 4.1 UNIX Shell:

#### 4.1.1 Shell 基本功能:

UNIX Shell 包括读取指令、创建子线程、执行指令读取指令通过以下 init\_args() 函数完成通过字符串函数的操作与条件判断，将输入函数解析成可以传入 execvp() 函数的形式。

```
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     while(tmp!='\n'&&tmp!=EOF)
9     {
10        if(Index_Instruction==0)
11        {
12            free(args[0]);
```

```

13     args[0]=(char *) malloc(sizeof(char)*MAX_LINE);
14     memset(args[0], 0, sizeof(args[0]));
15 }
16 if(tmp==' ')
17 {
18     if(precious_is_space==0)
19     {
20         Index_args++;
21         args[Index_args]=(char *) malloc(sizeof(char)*MAX_LINE);
22         precious_is_space=1;
23     }
24 }
25 else
26 {
27     strncat(args[Index_args],&tmp,1);
28
29     precious_is_space=0;
30 }
31 Index_Instruction++;
32 tmp=Instruction[Index_Instruction];
33 }

```

#### 4.1.2 使用子进程执行指令:

使用 fork() 函数创建子进程, 使用 execvp() 函数执行指令, 实现代码如下所示。

```

1 else
2 {
3     pid_t pid;
4     pid=fork();
5
6     if (pid==0) //child process
7     {
8         redirect_output(args);
9         redirect_input(args,last_parameter);
10        execvp(args[0],args);
11        should_run=0;
12    }
13    else //parent process
14    {
15        if(strcmp(last_parameter,"&&")!=0)
16            wait(NULL);
17    }
18 }

```

#### 4.1.3 !! 指令执行历史指令

在初始化指令和参数后调用如下 init\_history 函数来保存当前指令在缓存 history\_buffer 里。当判断下一条指令是!! 时执行 history\_ 中的指令。

```

1 char **init_history(char** history_buffer, char** args, char* last_parameter)

```

```

2 {
3     if (args[0]==NULL || strcmp(args[0], "!!")==0)
4         return NULL;
5     int i=0;
6     for (i=0; i<MAX_LINE/2+1; i++)
7     {
8         if (args[i]==NULL)
9             break;
10        history_buffer[i]=(char*) malloc(MAX_LINE*sizeof(char));
11        strcpy(history_buffer[i], args[i]);
12    }
13    if (strcmp(last_parameter, "&&")==0)
14    {
15        history_buffer[i]=(char*) malloc(MAX_LINE*sizeof(char));
16        history_buffer[i]="&&";
17    }
18 }
19 }

```

#### 4.1.4 输入输出重定向

在读取到指令中有重定向符号以后用 dup2() 函数将文件重定向到标准输入/输出，之后对文件的操作就和对标准输入输出的操作相同。

```

1 void redirect_output(char** args) //输出重定向
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
10            int fd=open(file_name, O_RDWR | O_NOCTTY | O_NDELAY);
11            dup2(fd, STDOUT_FILENO);
12
13            free(args[i]);
14            free(args[i+1]);
15            args[i]=NULL;
16            args[i+1]=NULL;
17
18            close(fd);
19        }
20    }
21 }
22 int redirect_input(char** args, char* last_parameter) //输入重定向
23 {
24     for (int i=0; i<MAX_LINE/2+1; ++i)
25     {
26         if (args[i]!=NULL && strcmp(args[i], "<")==0) // < exists
27         {

```

```

28         char file_name[MAX_LINE];
29         strcpy(file_name, args[i+1]);
30
31         int fd=open(file_name,O_RDWR | O_NOCTTY | O_NDELAY);
32
33         free(args[i]);
34         free(args[i+1]);
35         args[i]=NULL;
36         args[i+1]=NULL;
37         dup2(fd,STDIN_FILENO);
38         char Instruction[MAX_LINE*sizeof(char)];
39         fgets(Instruction,MAX_LINE,stdin);
40         init_args(Instruction,args+i,last_parameter);
41
42         close(fd);
43     }
44 }
45 return 0;
46 }

```

#### 4.1.5 pipe 进行进程间通信

```

1  //pipe
2  int pipe_position=detect_pipe(args);
3  if(pipe_position!=0) // pipe exists
4  {
5      pid_t pid;
6      pid=fork();
7
8      if(pid==0) //child process{
9          int fd[2];
10         pid_t pid;
11         /*create a pipe*/
12         if(pipe(fd)==-1){
13             fprintf(stderr,"Pipe failed");
14             return 1;
15         }
16         pid=fork();
17         if(pid>0){
18             for(int i=pipe_position;i<MAX_LINE/2+1;++i){
19                 free(args[i]);
20                 args[i]=NULL;
21             }
22             close(fd[READ_END]);
23
24             dup2(fd[WRITE_END],STDOUT_FILENO);
25
26             execvp(args[0],args);
27         }
28         else if(pid==0) //grandson process{

```

```

29 strcpy ( args [ 0 ] , args [ pipe_position + 1 ] );
30 for ( int i=1; i<MAX_LINE/2+1; ++i ) {
31     free ( args [ i ] );
32     args [ i ] = NULL;
33 }
34 close ( fd [ WRITE_END ] );
35 dup2 ( fd [ READ_END ] , STDIN_FILENO );
36 execvp ( args [ 0 ] , args );
37 }
38 }
39 else
40     wait ( NULL );

```

## 4.2 Linux Kernel Module for Task Information:

```

1  #include <linux/init.h>
2  #include <linux/slab.h>
3  #include <linux/sched.h>
4  #include <linux/module.h>
5  #include <linux/kernel.h>
6  #include <linux/proc_fs.h>
7  #include <linux/vmalloc.h>
8  #include <asm/uaccess.h>
9
10 #define BUFFER_SIZE 128
11 #define PROC_NAME "pid"
12
13 static long l_pid;
14 static ssize_t proc_read ( struct file *file , char *buf ,
15     size_t count , loff_t *pos );
16 static ssize_t proc_write ( struct file *file , const char __user *usr_buf ,
17     size_t count , loff_t *pos );
18
19 static struct file_operations proc_ops = {
20     .owner = THIS_MODULE,
21     .read = proc_read ,
22     .write = proc_write ,
23 };
24 static int proc_init ( void ) {
25     proc_create ( PROC_NAME , 0666 , NULL , &proc_ops );
26     printk ( KERN_INFO "/proc/%s created\n" , PROC_NAME );
27     return 0;
28 }
29
30 static void proc_exit ( void ) {
31     remove_proc_entry ( PROC_NAME , NULL );
32     printk ( KERN_INFO "/proc/%s removed\n" , PROC_NAME );
33 }
34 static ssize_t proc_read ( struct file *file ,
35     char __user *usr_buf , size_t count , loff_t *pos )

```

```

36 {
37     int rv = 0;
38     char buffer[BUFFER_SIZE];
39     static int completed = 0;
40     struct task_struct *tsk = NULL;
41
42     if (completed) {
43         completed = 0;
44         return 0;
45     }
46     tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
47     if (tsk==NULL){
48         printk(KERN_INFO "invalid pid!\n");
49         return 0;
50     }
51     completed = 1;
52     rv=sprintf(buffer, "command = [%s] pid=[%ld] state=[%ld]\n",
53         tsk->comm, l_pid, tsk->state);
54     raw_copy_to_user(usr_buf, buffer, rv);
55     return rv;
56 }
57 static ssize_t proc_write(struct file *file,
58     const char __user *usr_buf, size_t count, loff_t *pos)
59 {
60     char *k_mem;
61     k_mem = kmalloc(count, GFP_KERNEL);
62
63     if (raw_copy_from_user(k_mem, usr_buf, count)) {
64         printk(KERN_INFO "Error copying from user\n");
65         return -1;
66     }
67     printk("%s", k_mem);
68     sscanf(k_mem, "%ld", &l_pid);
69     kfree(k_mem);
70     return count;
71 }
72
73 module_init( proc_init );
74 module_exit( proc_exit );
75
76 MODULE_LICENSE("GPL");
77 MODULE_DESCRIPTION("Module");
78 MODULE_AUTHOR("SGG");

```

## 5 实验结果

### 5.1 UNIX Shell:

结果见图 1。图 2 为测试 pipe 的代码，其他功能都在图 1 中得以体现。



```

guanrenyang@ubuntu:~/ch3$ ./simple-shell
osh>!!
No commands in history.
osh>ls
in.txt      Module.symvers  pid.ko      pid.mod.o    simple-shell
Makefile    out.txt         pid.mod     pid.o        simple-shell.c
modules.order pid.c          pid.mod.c   README.md
osh>!!
ls
in.txt      Module.symvers  pid.ko      pid.mod.o    simple-shell
Makefile    out.txt         pid.mod     pid.o        simple-shell.c
modules.order pid.c          pid.mod.c   README.md
osh>ls > out.txt
osh>ls < in.txt
total 80
-rw-r--r-- 1 guanrenyang guanrenyang 3 May 17 03:39 in.txt
-rw-rw-r-- 1 guanrenyang guanrenyang 171 May 4 09:16 Makefile
-rw-r--r-- 1 root root 29 May 5 01:41 modules.order
-rw-r--r-- 1 root root 0 May 3 10:04 Module.symvers
-rw-r--r-- 1 guanrenyang guanrenyang 138 May 17 03:47 out.txt
-rw-r--r-- 1 guanrenyang guanrenyang 3220 May 5 01:48 pid.c
-rw-r--r-- 1 root root 8440 May 5 01:41 pid.ko
-rw-r--r-- 1 root root 29 May 5 01:41 pid.mod
-rw-r--r-- 1 root root 1157 May 5 01:41 pid.mod.c
-rw-r--r-- 1 root root 3896 May 5 01:41 pid.mod.o
-rw-r--r-- 1 root root 5456 May 5 01:41 pid.o
-rw-r--r-- 1 guanrenyang guanrenyang 1088 May 9 11:16 README.md
-rwxr-xr-x 1 guanrenyang guanrenyang 13552 May 17 00:24 simple-shell
-rw-rw-r-- 1 guanrenyang guanrenyang 6133 May 16 10:39 simple-shell.c
osh>S

```

图 1: UNIX Shell

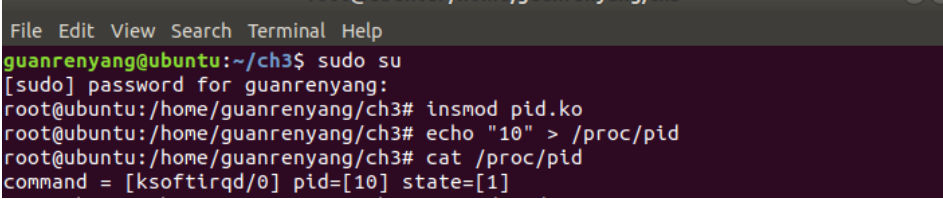
```

total 80
-rw-r--r-- 1 guanrenyang guanrenyang 3 May 17 03:39 in.txt
-rw-rw-r-- 1 guanrenyang guanrenyang 171 May 4 09:16 Makefile
-rw-r--r-- 1 root root 29 May 5 01:41 modules.order
-rw-r--r-- 1 root root 0 May 3 10:04 Module.symvers
-rw-r--r-- 1 guanrenyang guanrenyang 138 May 17 03:47 out.txt
-rw-r--r-- 1 guanrenyang guanrenyang 3220 May 5 01:48 pid.c
-rw-r--r-- 1 root root 8440 May 5 01:41 pid.ko
-rw-r--r-- 1 root root 29 May 5 01:41 pid.mod
-rw-r--r-- 1 root root 1157 May 5 01:41 pid.mod.c
-rw-r--r-- 1 root root 3896 May 5 01:41 pid.mod.o
-rw-r--r-- 1 root root 5456 May 5 01:41 pid.o
-rw-r--r-- 1 guanrenyang guanrenyang 1088 May 9 11:16 README.md
-rwxr-xr-x 1 guanrenyang guanrenyang 13552 May 17 00:24 simple-shell
-rw-rw-r-- 1 guanrenyang guanrenyang 6133 May 16 10:39 simple-shell.c
(END)

```

图 2: Result of `ls -l | less`

## 5.2 Linux Kernel Module for Task Information:

A terminal window with a dark background and light text. The menu bar at the top includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows a user 'guanrenyang' at 'ubuntu' in the directory '~/ch3' running 'sudo su'. After entering the password, the prompt changes to 'root@ubuntu:/home/guanrenyang/ch3#'. The user then runs 'insmod pid.ko', followed by 'echo "10" > /proc/pid', and finally 'cat /proc/pid'. The output of the last command is 'command = [ksoftirqd/0] pid=[10] state=[1]'.

```
File Edit View Search Terminal Help
guanrenyang@ubuntu:~/ch3$ sudo su
[sudo] password for guanrenyang:
root@ubuntu:/home/guanrenyang/ch3# insmod pid.ko
root@ubuntu:/home/guanrenyang/ch3# echo "10" > /proc/pid
root@ubuntu:/home/guanrenyang/ch3# cat /proc/pid
command = [ksoftirqd/0] pid=[10] state=[1]
```

图 3: Result of Linux Kernel Module for Task Information

## 6 总结与思考

### 6.1 UNIX Shell

通过此次实验我学会了如何模拟一个 UNIX Shell 进行命令行交互。我还学到了如何创建子进程执行特定的任务，如何进行文件描述符重定向，如何使用 pipe 进行进程间通信。

### 6.2 Linux Kernel Module for Task Information:

通过此次实验我学会了如何向 /proc 文件系统中写入文件。并且了解了一个进程的 PCB 在内存中的存储与访问方式。