Proj2 UNIX Shell Programming & Linux Kernel Module for Task Information

by 郑航 520021911347

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。

```
//清空args
for (int i = 0; i < MAX_LINE / 2 + 1; i++)
{
    args[i] = NULL;
}

//读入指令
char ch;
while (scanf("%s", input_tmp[arg_count]))
{
    args[arg_count] = input_tmp[arg_count];
    arg_count++;
    scanf("%c", &ch);
```

```
    if (ch == '\n')
        break;
}
```

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

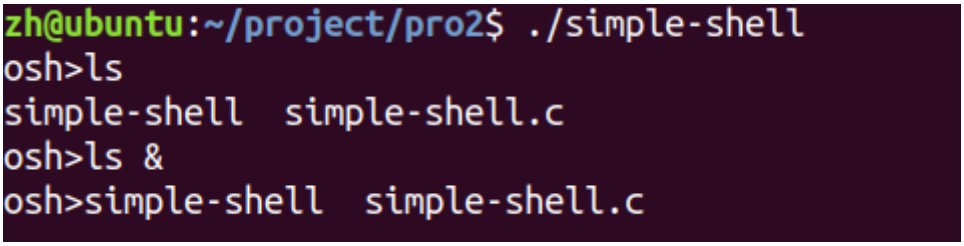
```
if(strcmp(args[0], "exit")==0) {
    should_run=0;
    continue;
}
```

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

```
//判断wait
if (strcmp(args[count], "&") == 0)
{
    args[count] = NULL;
    count--;
    wait_flag=0;
}
//创建新进程
pid_t pid;
pid = fork();
if (pid < 0) {
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) //child
{
    execvp(args[0], args);
}
else //parent
{
    if (wait_flag)
    {
        wait(NULL);
    }
}
```

2.3.4 执行结果



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：

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

history_cmd和have_history的赋值：

```
for (int i = 0; i < arg_count; i++)
{
    strcpy(tmp_str[i], args[i]);
    history_cmd[i] = tmp_str[i];
}
history_arg_count = arg_count;
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

```
//!!的实现
if (strcmp(args[0], "!!") == 0)
{
    if (!have_history)                                  //没有历史命令
    {
        printf("No commands in history!\n");
        continue;
    }
    else
    {
        printf("osh>");
        fflush(stdout);
        for (int i = 0; i < history_arg_count; i++)
        {
            strcpy(tmp_str[i], history_cmd[i]);
            args[i] = tmp_str[i];
            printf("%s", args[i]);
            printf("%s", " ");
        }
```

```
        printf("%c", '\n');
        arg_count = history_arg_count;
    }
}
```

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

```
//判断重定向
if (arg_count > 1 && strcmp(args[arg_count-2], ">")==0)
{
    redir_flag=1;   //redir_flag=1表示输出重定向
    filename=args[arg_count-1];
    args[arg_count-1]=NULL;
    args[arg_count-2]=NULL;
    arg_count-=2;
}
else if(arg_count > 1 && strcmp(args[arg_count-2], "<")==0){
    redir_flag=-1;   //redir_flag=1表示输出重定向
    filename=args[arg_count-1];
    args[arg_count-1]=NULL;
    args[arg_count-2]=NULL;
    arg_count-=2;
}
```

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

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

```
    if(fd==-1){
        fprintf(stderr, "Create file Failed\n");
        continue;
    }
    dup2(fd, STDOUT_FILENO);
}
else if(redir_flag==-1){    //输入重定向
    fd = open(filename,O_CREAT|O_RDONLY,S_IRUSR);
    dup2(fd, STDIN_FILENO);
}
```

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

```
//判断重定向
if (arg_count > 1 && strcmp(args[arg_count-2], ">")==0)
{
    redir_out_flag=1;   //redir_out_flag=1表示输出重定向
    out_file=args[arg_count-1];
    args[arg_count-1]=NULL;
    args[arg_count-2]=NULL;
    arg_count-=2;
}
if(arg_count > 1 && strcmp(args[arg_count-2], "<")==0){
    redir_in_flag=1;    //redir_in_flag=1表示输出重定向
    filename=args[arg_count-1];
    args[arg_count-1]=NULL;
    args[arg_count-2]=NULL;
    arg_count-=2;
}
```

```
//重定向执行
int fd;
if(redir_out_flag==1){    //输出重定向
    fd=open(out_file,O_CREAT|O_RDWR,S_IRWXU);
    if(fd==-1){
        fprintf(stderr, "Create file Failed\n");
        continue;
    }
    dup2(fd, STDOUT_FILENO);
}
if(redir_in_flag==1){    //输入重定向
    fd = open(in_file,O_CREAT|O_RDONLY,S_IRUSR);
    dup2(fd, STDIN_FILENO);
}
```

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

```
//判断pipe
for (int i = 0; i < arg_count; i++)
{
    if (strcmp(args[i], "|") == 0)
    {
        pipe_flag = 1;
        pipe_count = arg_count - i - 1;
        arg_count = i;
        for (int j = 0; j < MAX_LINE / 2 + 1; j++)  //清空pipe_arg
        {
            pipe_arg[j] = NULL;
        }
        for (int j = 0; j < pipe_count; j++)        //将原命令进行切分
        {
            pipe_arg[j] = args[i + 1];
            args[i + 1] = NULL;
        }
        args[i] = NULL;
```

```
            break;
        }
    }
```

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

```
//执行pipe
if (pipe_flag == 1)
{
    int pipe_result = pipe(pipe_fd);         //pipe的创建
    if (pipe_result == -1)
    {
        printf("Pipe Create Failed!\n");
    }

    pid_t pipe_pid = fork();                 //创建新进程
    if (pipe_pid < 0)
    {
        printf("Pipe Fork Failed\n");
        return 1;
    }
    else if (pipe_pid == 0)                  //child，执行第一条命令并写入pipe
    {
        close(pipe_fd[0]);
        dup2(pipe_fd[1], STDOUT_FILENO);
        execvp(args[0], args);
        exit(0);
    }
    else                                     //parent，从pipe中读取输入，并执行第二条命令
    {
        close(pipe_fd[1]);
        dup2(pipe_fd[0], STDIN_FILENO);
        execvp(pipe_arg[0], pipe_arg);
        wait(NULL);
    }
}
```

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

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

#define MAX_LINE 80 /* 80 chars per line, per command */

int main(void)
{
    char *args[MAX_LINE / 2 + 1]; /* command line (of 80) has max of 40 arguments */
    int should_run = 1;
    char input_tmp[MAX_LINE / 2 + 1][20];
    int arg_count = 0;
    char *history_cmd[MAX_LINE / 2 + 1];
    int have_history = 0;
    int history_arg_count = 0;
    int wait_flag = 1;
    char tmp_str[MAX_LINE / 2 + 1][20];
    char *out_file;
    char *in_file;
    int redir_out_flag = 0;
    int redir_in_flag = 0;
    int pipe_flag = 0;
    int pipe_count = 0;
    char *pipe_arg[MAX_LINE / 2 + 1];
    int pipe_fd[2];

    while (should_run)
    {
        printf("osh>");
        fflush(stdout);
        arg_count = 0;
        redir_out_flag = 0;
        redir_in_flag = 0;
        wait_flag = 1;
        pipe_flag = 0;

        //初始化清空args
        for (int i = 0; i < MAX_LINE / 2 + 1; i++)
        {
            args[i] = NULL;
        }
        if (!have_history)
        {
            for (int i = 0; i < MAX_LINE / 2 + 1; i++)
            {
                history_cmd[i] = NULL;
            }
        }
        //读入指令
```

```
        char ch;
        while (scanf("%s", input_tmp[arg_count]))
        {
            args[arg_count] = input_tmp[arg_count];
            arg_count++;
            scanf("%c", &ch);
            if (ch == '\n')
                break;
        }

        //判断exit
        if (strcmp(args[0], "exit") == 0)
        {
            should_run = 0;
            continue;
        }
        //判断并执行 !!
        if (strcmp(args[0], "!!") == 0)
        {
            if (!have_history)
            {
                printf("No commands in history!\n");
                continue;
            }
            else
            {
                printf("osh>");
                fflush(stdout);
                for (int i = 0; i < history_arg_count; i++)
                {
                    strcpy(tmp_str[i], history_cmd[i]);
                    args[i] = tmp_str[i];
                    printf("%s", args[i]);
                    printf("%s", " ");
                }
                printf("%c", '\n');
                arg_count = history_arg_count;
            }
        }
        else
        {
            for (int i = 0; i < arg_count; i++)
            {
                strcpy(tmp_str[i], args[i]);
                history_cmd[i] = tmp_str[i];
            }
            history_arg_count = arg_count;
            have_history = 1;
        }
        //判断wait
        if (strcmp(args[arg_count - 1], "&") == 0)
        {
            args[arg_count - 1] = NULL;
            arg_count--;
            wait_flag = 0;
        }

        //判断重定向
```

```
        if (arg_count > 1 && strcmp(args[arg_count - 2], ">") == 0)
        {
            redir_out_flag = 1; // redir_out_flag=1表示输出重定向
            out_file = args[arg_count - 1];
            args[arg_count - 1] = NULL;
            args[arg_count - 2] = NULL;
            arg_count -= 2;
        }
        if (arg_count > 1 && strcmp(args[arg_count - 2], "<") == 0)
        {
            redir_in_flag = 1; // redir_in_flag=1表示输出重定向
            in_file = args[arg_count - 1];
            args[arg_count - 1] = NULL;
            args[arg_count - 2] = NULL;
            arg_count -= 2;
        }

        //判断pipe
        for (int i = 0; i < arg_count; i++)
        {
            if (strcmp(args[i], "|") == 0)
            {
                pipe_flag = 1;
                pipe_count = arg_count - i - 1;
                arg_count = i;
                for (int j = 0; j < MAX_LINE / 2 + 1; j++)
                {
                    pipe_arg[j] = NULL;
                }
                for (int j = 0; j < pipe_count; j++)
                {
                    pipe_arg[j] = args[i + 1];
                    args[i + 1] = NULL;
                }
                args[i] = NULL;
                break;
            }
        }

        //创建新进程
        pid_t pid;
        pid = fork();
        if (pid < 0)
        {
            printf("Fork Failed\n");
            return 1;
        }
        else if (pid == 0) // child
        {
            //重定向执行
            int fd;
            if (redir_out_flag == 1)
            { //输出重定向
                fd = open(out_file, O_CREAT | O_RDWR, S_IRWXU);
                if (fd == -1)
                {
                    printf("Create file Failed\n");
                    continue;
```

```
                }
                dup2(fd, STDOUT_FILENO);
            }
            if (redir_in_flag == 1)
            { //输入重定向
                fd = open(in_file, O_CREAT | O_RDONLY, S_IRUSR);
                dup2(fd, STDIN_FILENO);
            }

            //执行pipe
            if (pipe_flag == 1)
            {
                int pipe_result = pipe(pipe_fd);
                if (pipe_result == -1)
                {
                    printf("Pipe Create Failed!\n");
                }

                pid_t pipe_pid = fork();
                if (pipe_pid < 0)
                {
                    printf("Pipe Fork Failed\n");
                    return 1;
                }
                else if (pipe_pid == 0)
                {
                    close(pipe_fd[0]);
                    dup2(pipe_fd[1], STDOUT_FILENO);
                    execvp(args[0], args);
                    exit(0);
                }
                else
                {
                    close(pipe_fd[1]);
                    dup2(pipe_fd[0], STDIN_FILENO);
                    execvp(pipe_arg[0], pipe_arg);
                    wait(NULL);
                }
            }
            else
            {
                execvp(args[0], args);
            }
            exit(0);
        }
        else // parent
        {
            if (wait_flag)
            {
                wait(NULL);
            }
        }
        /**
         * After reading user input, the steps are:
         * (1) fork a child process
         * (2) the child process will invoke execvp()
         * (3) if command includes &, parent and child will run
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sccanf()函数替代

函数实现如下：

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

```
        /* copies user space usr_buf to kernel buffer */
        if (copy_from_user(k_mem, usr_buf, count))
        {
                printk(KERN_INFO "Error copying from user\n");
                return -1;
        }

        /**
         * kstrol() will not work because the strings are not guaranteed
         * to be null-terminated.
         *
         * sscanf() must be used instead.
         */
        sscanf(k_mem, "%ld", &l_pid);

        kfree(k_mem);

        return count;
}
```

3.3 proc_read

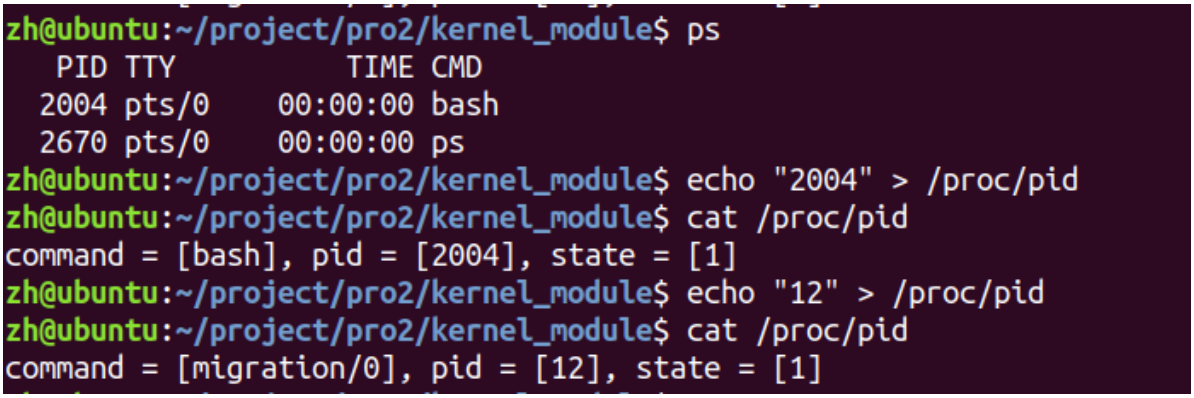
- 根据课本的实现思路和代码框架，还需要增加如下的部分：①判断tsk的返回值；②将tsk的信息存于内核buffer中；③若信息读取、转存成功，将completed置为1
- tsk的返回值如是NULL的话说明查询失败，此时函数需要返回0
- tsk的信息存于buffer，可类比proc_write中的sccanf，使用对应的sprintf函数，并将返回值存于proc_read函数返回值rv中，表示读取是否成功
- completed置1应该在整个函数的最后进行

函数实现如下：

```
static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t
count, loff_t *pos)
{
        int rv = 0;
        char buffer[BUFFER_SIZE];
        static int completed = 0;
        struct task_struct *tsk = NULL;

        if (completed)
        {
                completed = 0;
                return 0;
        }

        tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
        if (tsk == NULL)
                return 0;

        rv = sprintf(buffer, "command = [%s], pid = [%ld], state = [%ld]\n",
                     tsk->comm, tsk->pid, tsk->state);

        // copies the contents of kernel buffer to userspace usr_buf
        if (copy_to_user(usr_buf, buffer, rv))
        {
```

```
                rv = -1;
        }

        completed = 1;

        return rv;
}
```

3.4 执行结果



3.5 pid.c 完整代码

```c
/**
 * Kernel module that communicates with /proc file system.
 *
 * This provides the base logic for Project 2 - displaying task information
 */

#include <linux/init.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/vmalloc.h>
#include <asm/uaccess.h>
#include <linux/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "pid"

/* the current pid */
static long l_pid;

/**
 * Function prototypes
 */
static ssize_t proc_read(struct file *file, char *buf, size_t count, loff_t *pos);
static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
```

```
    .write = proc_write};

/* This function is called when the module is loaded. */
static int proc_init(void)
{
        // creates the /proc/procfs entry
        proc_create(PROC_NAME, 0666, NULL, &proc_ops);

        printk(KERN_INFO "/proc/%s created\n", PROC_NAME);

        return 0;
}

/* This function is called when the module is removed. */
static void proc_exit(void)
{
        // removes the /proc/procfs entry
        remove_proc_entry(PROC_NAME, NULL);

        printk(KERN_INFO "/proc/%s removed\n", PROC_NAME);
}

/**
 * This function is called each time the /proc/pid is read.
 *
 * This function is called repeatedly until it returns 0, so
 * there must be logic that ensures it ultimately returns 0
 * once it has collected the data that is to go into the
 * corresponding /proc file.
 */
static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t
count, loff_t *pos)
{
        int rv = 0;
        char buffer[BUFFER_SIZE];
        static int completed = 0;
        struct task_struct *tsk = NULL;

        if (completed)
        {
                completed = 0;
                return 0;
        }

        tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
        if (tsk == NULL)
                return 0;

        rv = sprintf(buffer, "command = [%s], pid = [%ld], state =
[%ld]\n",
                        tsk->comm, tsk->pid, tsk->state);

        // copies the contents of kernel buffer to userspace usr_buf
        if (copy_to_user(usr_buf, buffer, rv))
        {
                rv = -1;
        }

```

```
        completed = 1;

        return rv;
}

/**
 * This function is called each time we write to the /proc file system.
 */
static ssize_t proc_write(struct file *file, const char __user *usr_buf,
size_t count, loff_t *pos)
{
        char *k_mem;

        // allocate kernel memory
        k_mem = kmalloc(count, GFP_KERNEL);

        /* copies user space usr_buf to kernel buffer */
        if (copy_from_user(k_mem, usr_buf, count))
        {
                printk(KERN_INFO "Error copying from user\n");
                return -1;
        }

        /**
         * kstrol() will not work because the strings are not guaranteed
         * to be null-terminated.
         *
         * sscanf() must be used instead.
         */
        sscanf(k_mem, "%ld", &l_pid);

        kfree(k_mem);

        return count;
}

/* Macros for registering module entry and exit points. */
module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Proj 2");
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