Operating System Project 2

潘梓丞 517030910349

The goal of the project is to build a UNIX shell and Linux Kernel Module for Task Information
The porject was done by VM VirtualBox 5.2.18
The code are written by C and the library needed will be shown in code

1.simple-shell

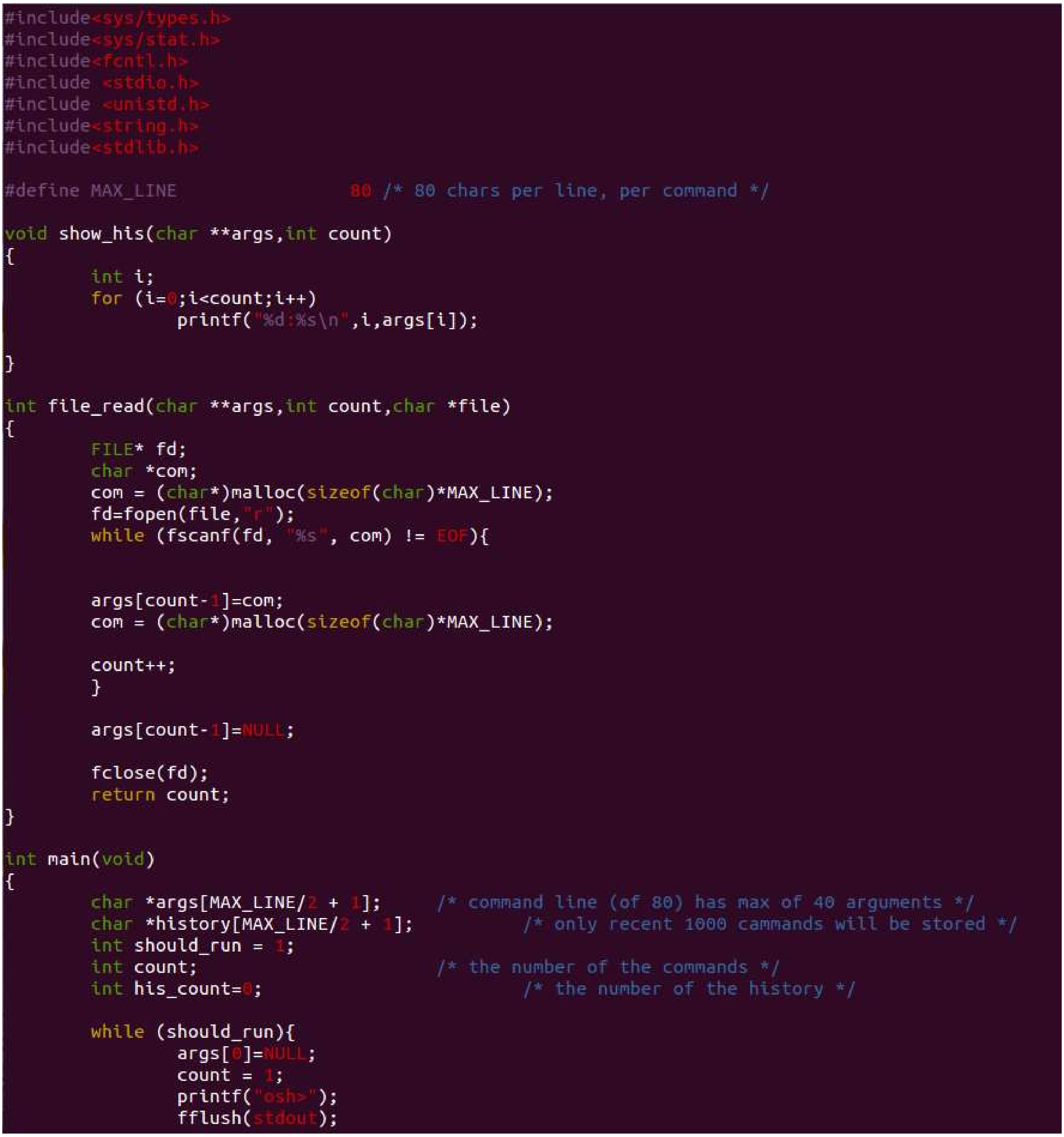
idea:

using a $char * command$ to get commands from terminal string by string. Once we get a string, we check whether we need a special operation, if need ,set a flag. After getting all the commands, the program will use $fork()$ to create a new process and complete its work according to different flags. The meanings of different flags are labeled in the code

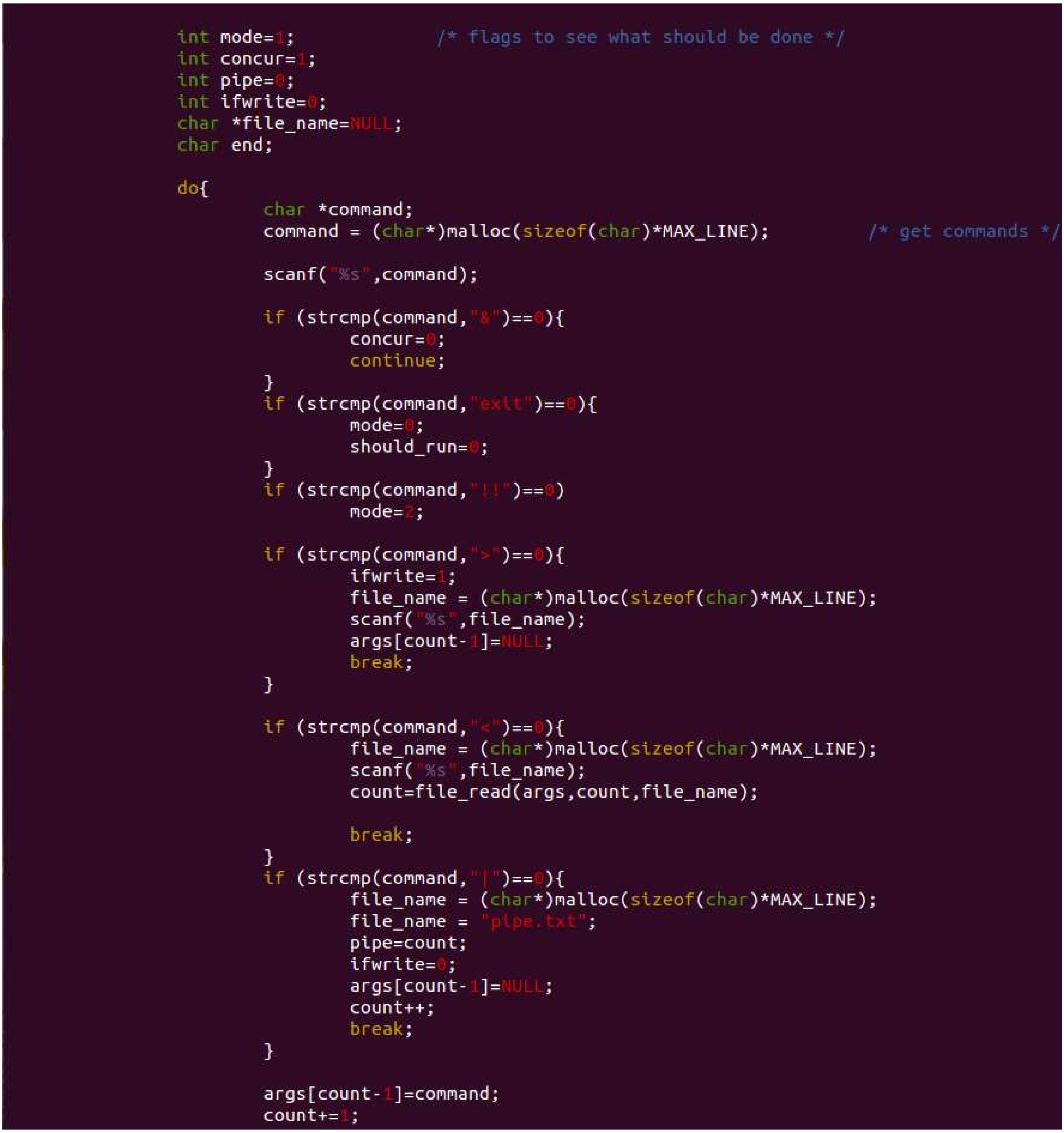
problems

some problems occurred while coding.
<1> in the child process, after executing the commands, forget to set $should\_run$ to 0. Because the function $fork()$ will create a totally same parent process, so once after executing ,the program will loop in child process and never quit， which yields a bug that $history$ will always return $NULL$ and the program will use space larger and larger by time.
<2> in the history part, we have to store the commands we did last time, so we can't delete the strings immediately. So it is hard to decide which time to free the space, so each time we write $char * history$, we need to free it first.

code



```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include<string.h>
#include<stdlib.h>

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

void show_his(char **args,int count)
{
        int i;
        for (i=0;i<count;i++)
                printf("%d:%s\n",i,args[i]);

}

int file_read(char **args,int count,char *file)
{
        FILE* fd;
        char *com;
        com = (char*)malloc(sizeof(char)*MAX_LINE);
        fd=fopen(file,"r");
        while (fscanf(fd, "%s", com) != EOF){


        args[count-1]=com;
        com = (char*)malloc(sizeof(char)*MAX_LINE);

        count++;
        }

        args[count-1]=NULL;

        fclose(fd);
        return count;
}

int main(void)
{
        char *args[MAX_LINE/2 + 1];     /* command line (of 80) has max of 40 arguments */
        char *history[MAX_LINE/2 + 1];          /* only recent 1000 cammands will be stored */
        int should_run = 1;
        int count;                      /* the number of the commands */
        int his_count=0;                        /* the number of the history */

        while (should_run){
                args[0]=NULL;
                count = 1;
                printf("osh>");
                fflush(stdout);
```



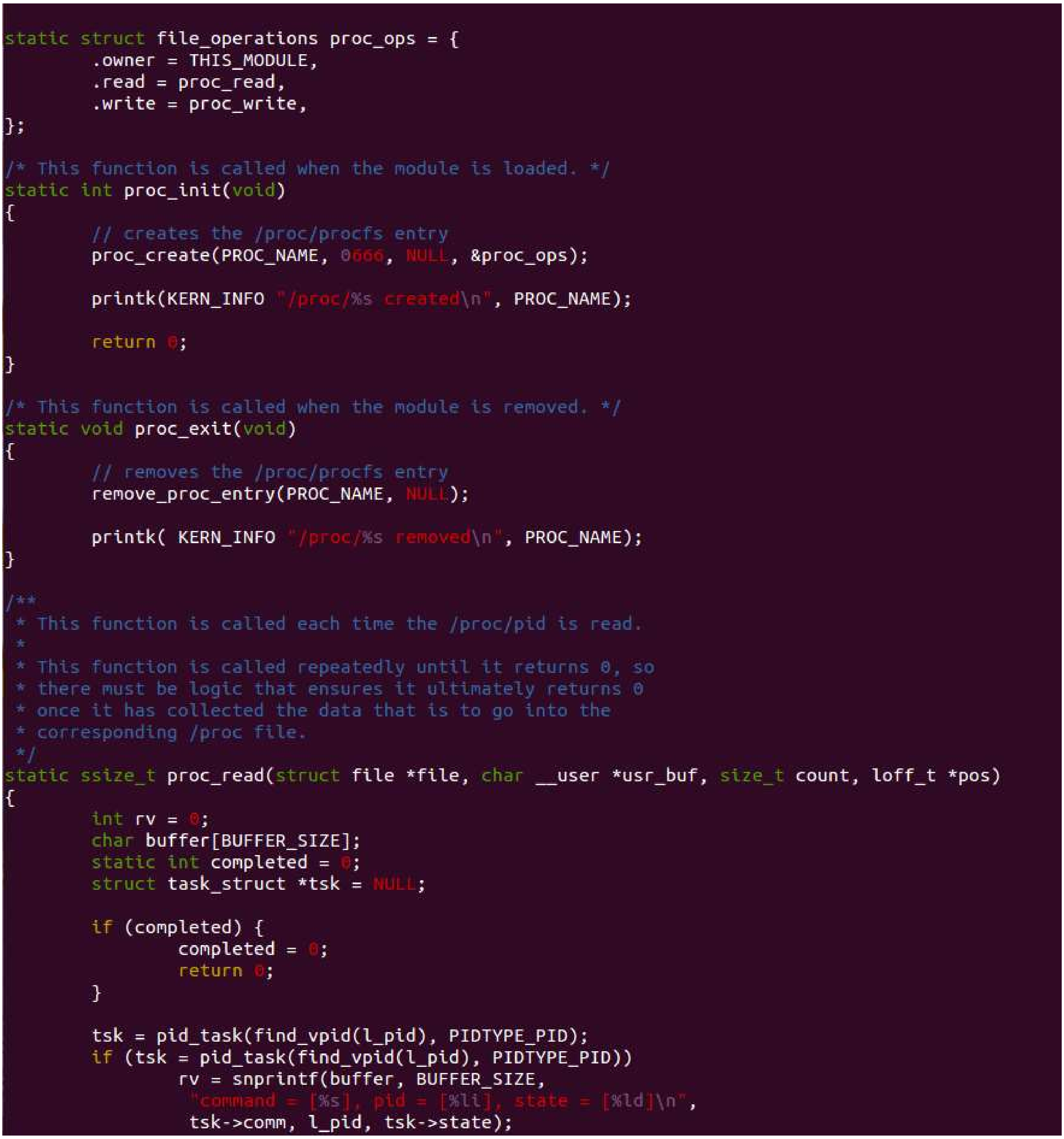
```
                int mode=1;             /* flags to see what should be done */
                int concur=1;
                int pipe=0;
                int ifwrite=0;
                char *file_name=NULL;
                char end;

                do{
                        char *command;
                        command = (char*)malloc(sizeof(char)*MAX_LINE);         /* get commands */

                        scanf("%s",command);

                        if (strcmp(command,"&")==0){
                                concur=0;
                                continue;
                        }
                        if (strcmp(command,"exit")==0){
                                mode=0;
                                should_run=0;
                        }
                        if (strcmp(command,"!!")==0)
                                mode=2;

                        if (strcmp(command,">")==0){
                                ifwrite=1;
                                file_name = (char*)malloc(sizeof(char)*MAX_LINE);
                                scanf("%s",file_name);
                                args[count-1]=NULL;
                                break;
                        }

                        if (strcmp(command,"<")==0){
                                file_name = (char*)malloc(sizeof(char)*MAX_LINE);
                                scanf("%s",file_name);
                                count=file_read(args,count,file_name);

                                break;
                        }
                        if (strcmp(command,"|")==0){
                                file_name = (char*)malloc(sizeof(char)*MAX_LINE);
                                file_name = "pipe.txt";
                                pipe=count;
                                ifwrite=0;
                                args[count-1]=NULL;
                                count++;
                                break;
                        }

                        args[count-1]=command;
                        count+=1;
```

2.Linux Kernel Module for Task Information

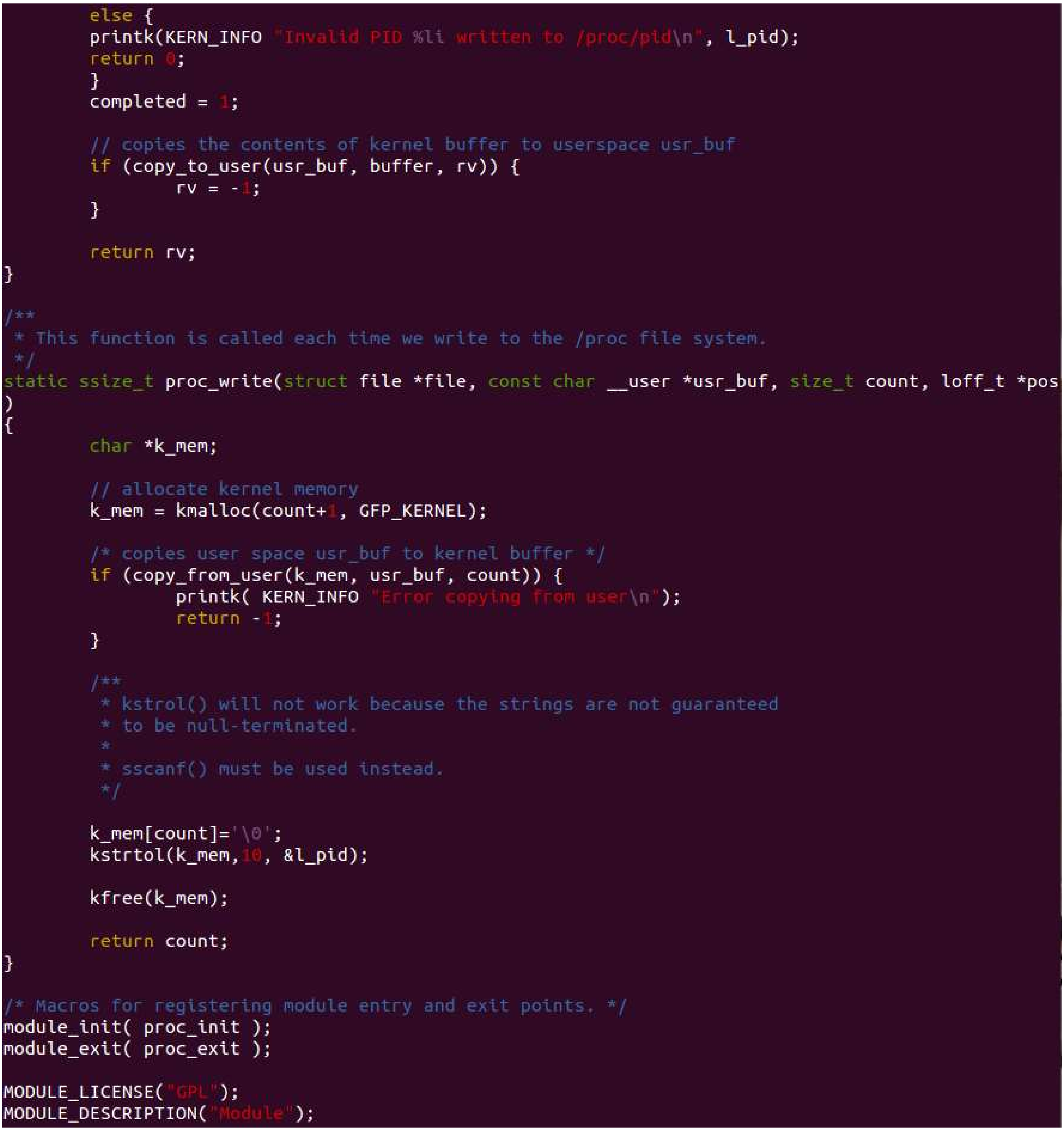
idea

Using kernel functions kstrtol to get information from terminal and when cat is called, print the thread of given id. Using the given template, it is a simple work. Don't foeget $Makefile$

code



```
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 = pid_task(find_vpid(l_pid), PIDTYPE_PID))
                rv = snprintf(buffer, BUFFER_SIZE,
                 "command = [%s], pid = [%li], state = [%ld]\n",
                 tsk->comm, l_pid, tsk->state);
```



```
        else {
        printk(KERN_INFO "Invalid PID %li written to /proc/pid\n", l_pid);
        return 0;
        }
        completed = 1;

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

        k_mem[count]='\0';
        kstrtol(k_mem,10, &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("Module");
```