OS Project-2 Report

UNIX Shell && Linux Kernel Module for Task Information

Zhou Yiyuan

516021910270
1025152261@qq.com

1 UNIX Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process.

1.1 Requirements

(a) Creating the child process and executing the command in the child

(b) Providing a history feature

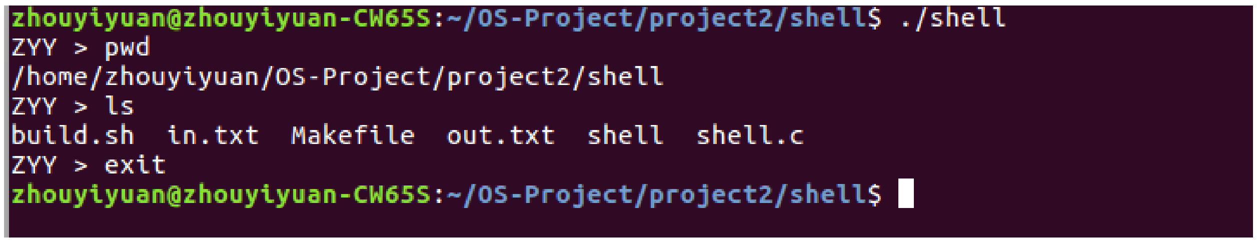
(c) Adding support of input and output redirection

(d) Allowing the parent and child processes to communicate via a pipe

1.2 Results Display

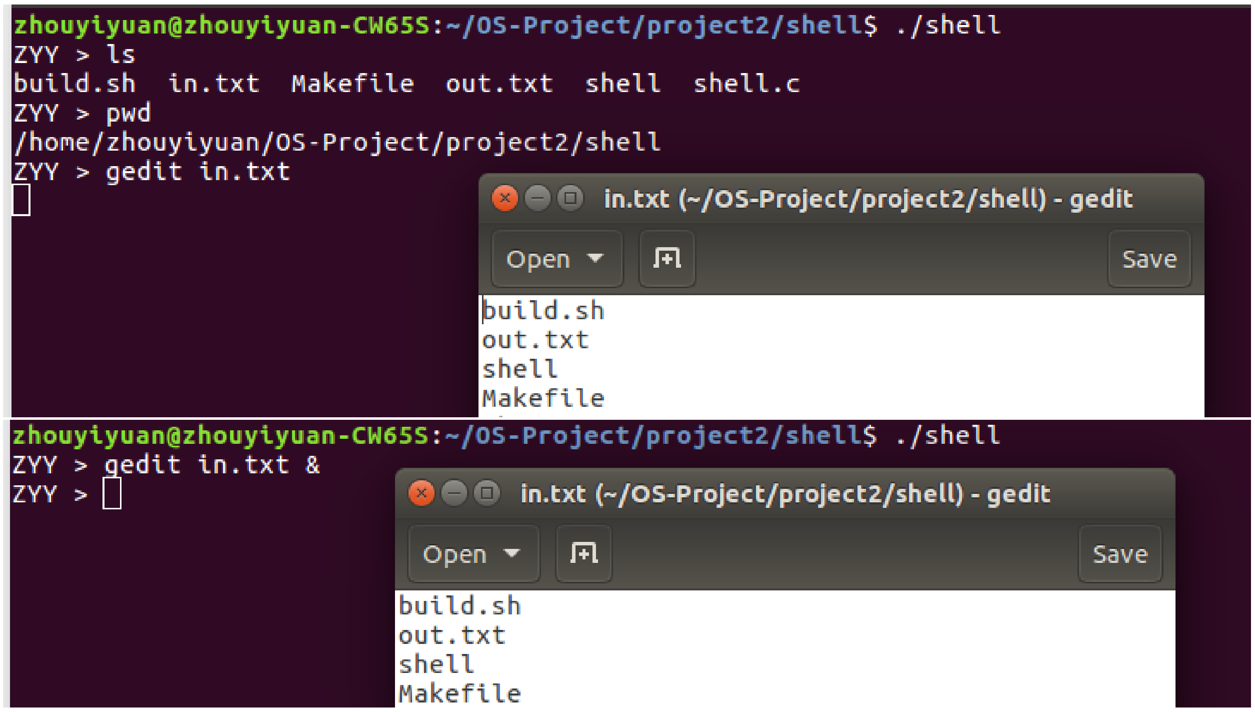
1.2.1 Basic

The results are shown as Figure 1 and Figure 2.



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > pwd
/home/zhouyiyuan/OS-Project/project2/shell
ZYY > ls
build.sh  in.txt  Makefile  out.txt  shell  shell.c
ZYY > exit
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ 
```

Figure 1: When either command ”ls” or ”pwd” is entered, the output is same as that of UNIX.When command ”exit” is entered, my shell exits and it return back to initial terminal.



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > ls
build.sh  in.txt  Makefile  out.txt  shell  shell.c
ZYY > pwd
/home/zhouyiyuan/OS-Project/project2/shell
ZYY > gedit in.txt
```

in.txt (~/OS-Project/project2/shell) - gedit
Open
Save

```
build.sh
out.txt
shell
Makefile
```

```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > gedit in.txt &
ZYY >
```

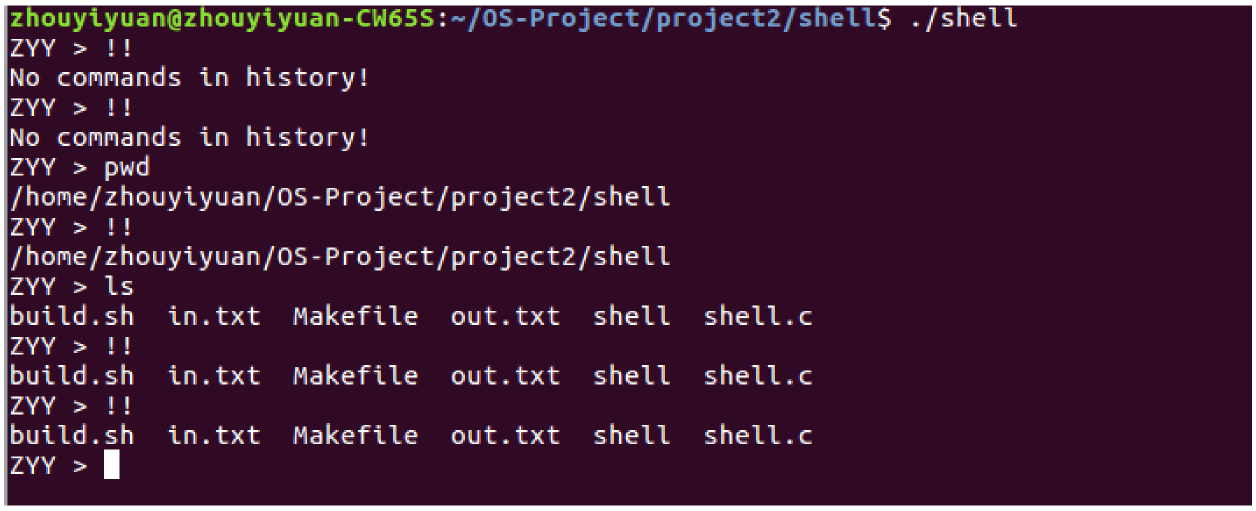
in.txt (~/OS-Project/project2/shell) - gedit
Open
Save

```
build.sh
out.txt
shell
Makefile
```

Figure 2: When a command is ended with '&', the parent will not wait for the child to exit, as shown in the lower figure. Otherwise, the parent will wait for the child, as shown in the upper figure.

1.2.2 History Feature

The results are shown as Figure 3.

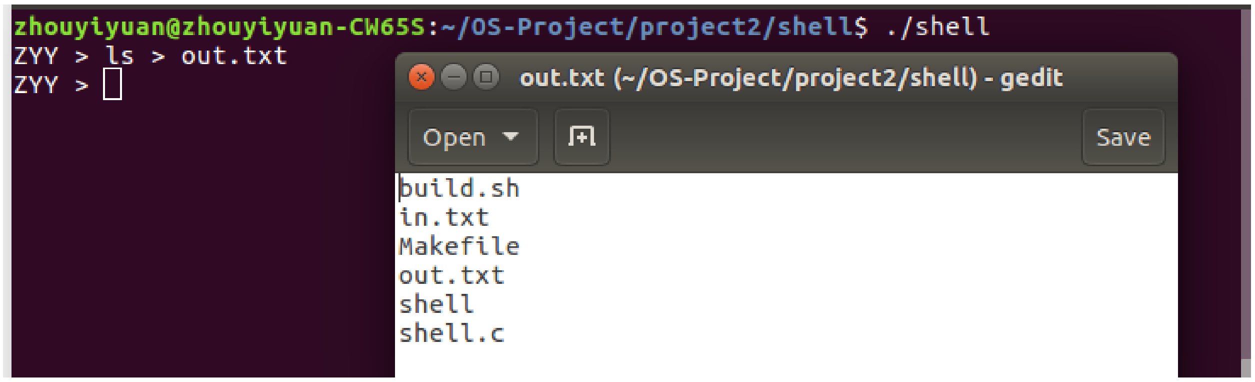


```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > !!
No commands in history!
ZYY > !!
No commands in history!
ZYY > pwd
/home/zhouyiyuan/OS-Project/project2/shell
ZYY > !!
/home/zhouyiyuan/OS-Project/project2/shell
ZYY > ls
build.sh  in.txt  Makefile  out.txt  shell  shell.c
ZYY > !!
build.sh  in.txt  Makefile  out.txt  shell  shell.c
ZYY > !!
build.sh  in.txt  Makefile  out.txt  shell  shell.c
ZYY >
```

Figure 3: When "!!" is entered, the last command is executed again. However, no last command will result in a message "No command in history".

1.2.3 Redirecting

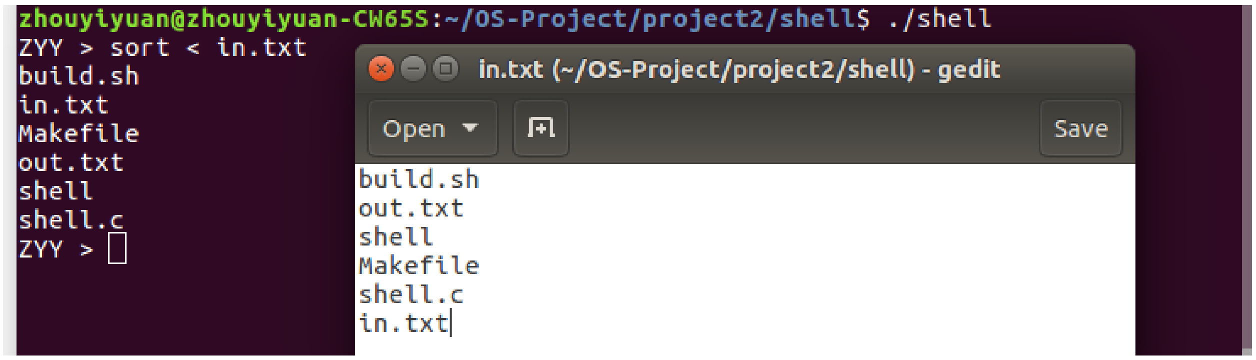
The results are shown as Figure 4, Figure 5, and Figure 6.



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > ls > out.txt
ZYY >
```

```
build.sh
in.txt
Makefile
out.txt
shell
shell.c
```

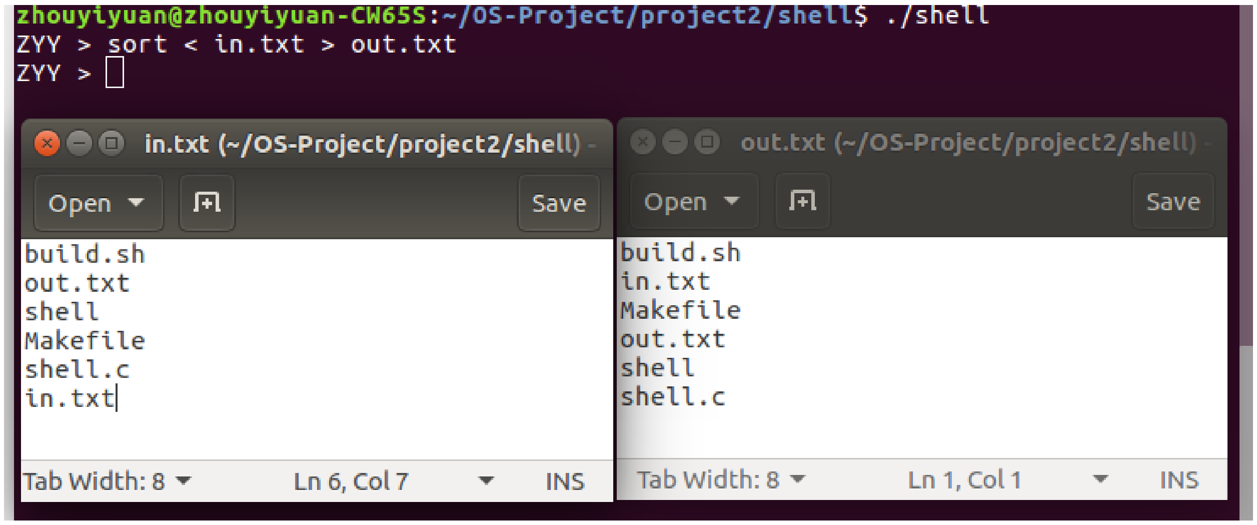
Figure 4: The result of command "ls" is redirected into "out.txt".



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > sort < in.txt
build.sh
in.txt
Makefile
out.txt
shell
shell.c
ZYY >
```

```
build.sh
out.txt
shell
Makefile
shell.c
in.txt
```

Figure 5: The lines in "in.txt" are sorted and display in shell.



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > sort < in.txt > out.txt
ZYY >
```

in.txt (~/OS-Project/project2/shell)
Open
Save
Tab Width: 8
Ln 6, Col 7
INS

```
build.sh
out.txt
shell
Makefile
shell.c
in.txt
```

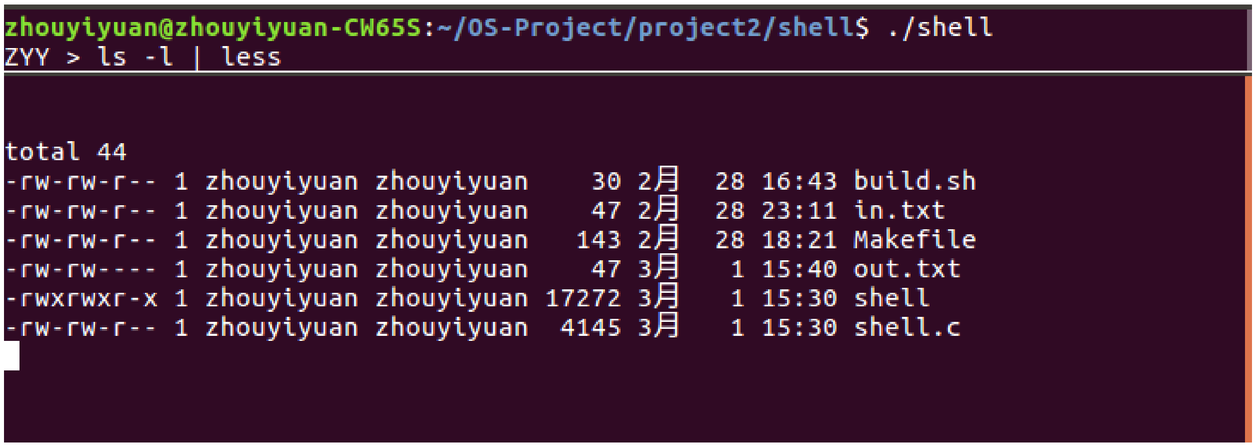
out.txt (~/OS-Project/project2/shell)
Open
Save
Tab Width: 8
Ln 1, Col 1
INS

```
build.sh
in.txt
Makefile
out.txt
shell
shell.c
```

Figure 6: The lines in ”in.txt” are sorted and then are redirected into ”out.txt”.

1.2.4 Communication via s Pipe

The results are shown as Figure 7.



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/shell$ ./shell
ZYY > ls -l | less


total 44
-rw-rw-r-- 1 zhouyiyuan zhouyiyuan    30 2月  28 16:43 build.sh
-rw-rw-r-- 1 zhouyiyuan zhouyiyuan    47 2月  28 23:11 in.txt
-rw-rw-r-- 1 zhouyiyuan zhouyiyuan   143 2月  28 18:21 Makefile
-rw-rw---- 1 zhouyiyuan zhouyiyuan    47 3月   1 15:40 out.txt
-rwxrwxr-x 1 zhouyiyuan zhouyiyuan 17272 3月   1 15:30 shell
-rw-rw-r-- 1 zhouyiyuan zhouyiyuan  4145 3月   1 15:30 shell.c
```

Figure 7: The output of the command ”ls -l” is served as the input to the command ”less”.

1.3 Code

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

```
#define MAX_LINE 80

int process_input(char input[], char *args[]);
void process_args(char *args[], int *in, int *out, int *pipe);

int main(void)
{
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;
    char input[MAX_LINE];
    char last_command[MAX_LINE];
    last_command[0] = '\0';

    // printf(pipe_file);
    // fflush(stdout);

    while(should_run){
        printf("ZYY > ");
        fflush(stdout);

        gets(input);

        int waiting = process_input(input, args);
        int in = -1;
        int out = -1;
        int pipe_index = -1;

        int pipefd[2];
        pipe(pipefd);

        //history feature
        if(strcmp(args[0], "!!") == 0 && !args[1]){
            if(last_command[0] == '\0'){
                printf("No commands in history!\n");
                input[0] = '\0';
            }else{
                strcpy(input, last_command);
                waiting = process_input(input, args);
            }
        }

        //command "exit"
        if(strcmp(args[0], "exit") == 0){
            should_run = 0;
            continue;
        }
        strcpy(last_command, input);

        //detect '>'and'<'and'|'
        process_args(args, &in, &out, &pipe_index);

        pid_t pid = fork();
        if(pid < 0){
            printf("fail to fork\n");
            exit(1);
        }else if(pid == 0) {
            //son

            //communication via a pipe
            if(pipe_index > 0){
                pid_t pid_son = fork();
                if(pid_son < 0){
                    printf("fail to fork\n");
                    exit(1);
                }else if(pid_son == 0)
                {
                    //son'son
                    close(pipefd[0]);
                    dup2(pipefd[1], STDOUT_FILENO);
                    execvp(args[0], args);
                    exit(1);
                }else{
                    //son
                    wait(NULL);

                    close(pipefd[1]);
                    dup2(pipefd[0], STDIN_FILENO);
                    execvp(args[pipe_index + 1], args + pipe_index + 1);
                    close(pipefd[1]);
                    exit(1);
                }
            }else{
                //redirecting input
                if(in > 0){
                    dup2(open(args[in + 1], O_RDONLY), STDIN_FILENO);
                }
                //redirection output
                if(out > 0){
                    dup2(open(args[out + 1], O_WRONLY | O_TRUNC | O_CREAT,
                     ↪  S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR), STDOUT_FILENO);
                }
                execvp(args[0], args);
```

```c
                exit(1);
            }


        }else{
            //father
            if(waiting) wait(NULL);
            // printf("son exit\n");
        }
        sleep(1);
        fflush(stdin);
    }
    return 0;
}

//if command ends with &, it returns 0 else 1
int process_input(char input[], char *args[])
{
    int i = 0;
    int j = 0;
    while(1)
    {
        char c = *(input + i);
        if(*(input + i) == ' ' || *(input + i) == '\t') *(input + i) = '\0';
        else if(*(input + i) == '\0')
        {
            args[j] = NULL;
            if(j > 0 && args[j - 1][0] == '&' && args[j - 1][1] == '\0')
            {
                args[j - 1] = NULL;
                return 0;
            }
            else return 1;
        }
        else
        {
            if(i == 0 || *(input + i - 1) == '\0')
           {
               args[j++] = input + i;
           }
        }
        i++;
    }
}

//detect > and < and |
void process_args(char *args[], int *in, int *out, int *pipe)
```

```
{
    for(int i = 0; i < MAX_LINE/2 + 1; i++){
        if(!args[i]) break;
        if(strcmp(args[i], "<") == 0){
            *in = i;
            args[i] = NULL;
        }else if(strcmp(args[i], ">") == 0){
            *out = i;
            args[i] = NULL;
        }else if(strcmp(args[i], "|") == 0){
            *pipe = i;
            args[i] = NULL;
        }
    }
}
```

2 Linux Kernel Module for Task Information

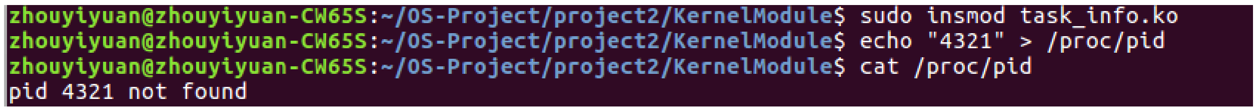
2.1 Requirement

Write a Linux kernel module that uses /proc file system for displaying a task's information based on its process identifier value pid. Once a pid has been written to the /proc file, subsequent reads from /proc/pid will report (1) the command the task is running, (2) the value of the task's pid , and (3) the current state of the task. An example of how your kernel module will be accessed once loaded into the system is as follows:

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

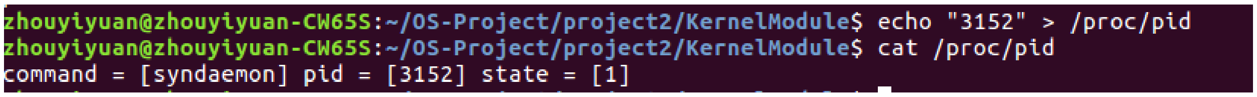
2.2 Result Display

The results are shown as Figure 8 and Figure 9.



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/KernelModule$ sudo insmod task_info.ko
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/KernelModule$ echo "4321" > /proc/pid
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/KernelModule$ cat /proc/pid
pid 4321 not found
```

Figure 8: No process's pid is 4321 and a warning is raised.



```
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/KernelModule$ echo "3152" > /proc/pid
zhouyiyuan@zhouyiyuan-CW65S:~/OS-Project/project2/KernelModule$ cat /proc/pid
command = [syndaemon] pid = [3152] state = [1]
```

Figure 9: The command is "syndaemon", the pid value is 3152, and the current state is 1.

2.3 Code

```c
#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 <linux/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "pid"

static long pid_num;

ssize_t proc_read(struct file *file, char __user *usr_buf, size_t count, loff_t
↪   *pos);
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
};

ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count,
↪   loff_t *pos)
{
    char *k_mem;

    k_mem = kmalloc(count, GFP_KERNEL);

    if(copy_from_user(k_mem, usr_buf, count)){
        printk(KERN_INFO "Error copying from user\n");
        return -1;
    }

    k_mem[count-1] = '\0';
    kstrtol(k_mem, 10, &pid_num);

    kfree(k_mem);

    return count;
}
```

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

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

    tsk = pid_task(find_vpid(pid_num), PIDTYPE_PID);
    if(tsk){
        volatile long state = tsk -> state;
        pid_t pid = tsk -> pid;
        char *command = tsk -> comm;

        rv = sprintf(buffer, "command = [%s] pid = [%u] state = [%u]\n",
         ↪  command, pid, state);
    }else{
        rv = sprintf(buffer, "pid %u not found\n", pid_num);
    }

    completed = 1;

    copy_to_user(usr_buf, buffer, rv);

    return rv;
}


int task_info_init(void)
{
    printk(KERN_INFO "/proc/%s created", PROC_NAME);
    proc_create(PROC_NAME, 0666, NULL, &proc_ops);

    return 0;
}

void task_info_exit(void)
{
    remove_proc_entry(PROC_NAME, NULL);
    printk(KERN_INFO "/proc/%s removed", PROC_NAME);
}
```

```c
module_init(task_info_init);
module_exit(task_info_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("PID Task Information Module");
MODULE_AUTHOR("ZYY");
```