

# Project 2 Report

Litao Zhou, 518030910407

## Project Description

In this project, we implement a shell application which supports the following functions.

- Execute user commands
- Remember and rerun last command
- Redirect Inputs and Outputs
- A simple pipelining

We also create another kernel module which can let user input pid number and return the detail of the process.

The source code of this project can be found [here](#).

## Contents

<b>Project Description .....</b>	<b>1</b>
<b>Unix Shell – Basic.....</b>	<b>3</b>
<i>Problem Description.....</i>	<i>3</i>
<i>Analysis &amp; Implementation .....</i>	<i>3</i>
Parse the Input.....	3
Background Running .....	4
Command Execution.....	4
<i>Result Demonstration.....</i>	<i>5</i>
<b>Unix Shell – History Feature .....</b>	<b>6</b>
<i>Problem Description.....</i>	<i>6</i>
<i>Analysis &amp; Implementation .....</i>	<i>6</i>
History Storage .....	6
History Execution .....	6
<i>Result Demonstration.....</i>	<i>7</i>
<b>Unix Shell – Redirecting I/O .....</b>	<b>9</b>
<i>Problem Description.....</i>	<i>9</i>
<i>Analysis &amp; Implementation .....</i>	<i>9</i>

Parse I/O Directory .....	9
I/O Redirection .....	9
<i>Result Demonstration</i> .....	10
<b>Unix Shell – Pipe Communication .....</b>	<b>11</b>
<i>Problem Description</i> .....	11
<i>Analysis &amp; Implementation</i> .....	11
Input Parsing .....	11
Pipe Implementation through child process.....	12
<i>Result Demonstration</i> .....	13
<b>Linux Kernel Module for Task Information.....</b>	<b>14</b>
<i>Problem Description</i> .....	14
<i>Analysis &amp; Implementation</i> .....	14
Writing to the /proc File System .....	14
Reading from the /proc File System .....	14
<i>Result Demonstration</i> .....	15

# Unix Shell – Basic

## Problem Description

In this part, we implement the basic functions of unix shell. We first parse the user input into a linked list of arguments which can be accepted by the `execvp()` command. Then we create a child process to run the command input by the user.

## Analysis & Implementation

### Parse the Input

The input of the user will be read character by character. The tokens will be split by spaces, and ended with a linebreak. Note that there may be multiple spaces between tokens, so we use a flag called `should_new_arg` to indicate whether a/some space(s) have been encountered.

The arguments will be read into a linked list called `args`, with length `arg_cnt`. These two variables will be used later in the command execution part. The `arg_cnt` will be changed on the fly, to make sure that every character is input into the argument list in the right place.

For simplicity in implementation, for every argument, we assign an array of characters of length `MAX_LINE`. All arguments in the linked list will end with an `'\0'`.

```
char c;
int should_new_arg = 1;
int arg_cnt = 0;
int arg_len = 0;
/** collect the input into args
 *
 * arg_cnt: length of input tokens
 *
 */
while (1) {
    c = getchar();
    if (c == '\n'){
        if (arg_cnt > 0)
            args[arg_cnt-1][arg_len++] = '\0';
        args[arg_cnt] = NULL;
        break;
    }
    if (c == ' '){
        if (should_new_arg == 0){
            if (arg_cnt > 0)
                args[arg_cnt-1][arg_len++] = '\0';
            should_new_arg = 1;
        }
    } else {
        if (should_new_arg){
            char* new_arg;
            new_arg = (char*) malloc(MAX_LINE*sizeof(char));
            args[arg_cnt++] = new_arg;
        }
    }
}
```

```

        arg_len = 0;
        args[arg_cnt-1][arg_len++] = c;
        should_new_arg = 0;
    } else {
        args[arg_cnt-1][arg_len++] = c;
    }
}
} /* end of input processing */

if (arg_cnt == 0){
    continue;
}

if (arg_cnt == 1 && strcmp(args[0], "exit") == 0){
    should_run = 0;
    break;
}

```

Note that after the input processing, we will check whether the input command is empty or the input argument is exit. If the input command is empty, the loop will be restarted, waiting for an input in the new line. If an "exit" is detected, a break will happen, which will end the shell program.

### Background Running

We add an additional flag in the program to detect whether there exists an "&" at the end of the command. If it is detected, the "&" argument will be removed from the args linked list and the flag will be set true for later execution.

```

int should_wait = 1;
if (args[arg_cnt-1][0] == '&'){
    should_wait = 0;
    free(args[arg_cnt-1]);
    args[--arg_cnt] = NULL;
}

```

### Command Execution

The command will be executed as a child process of the shell program. We use fork() to create a child process. In the child process, we call execvp() function to run the user command. The function takes in two arguments, one is the command name, and the other is a linked list of command arguments. The arguments have already been ready during the parsing stage, so we can simply pass them to the function here.

```

pid_t pid = fork();

if (pid == -1) {
    perror("fork failed");
    exit(EXIT_FAILURE);
}
else if (pid == 0) {

```

```

    execvp(args[0],args);
    _exit(EXIT_SUCCESS);
}
else {
    int status;
    if (should_wait){
        (void)waitpid(pid, &status, 0);
    }
    for (int i=0;args[i]!=NULL;++i){
        // printf("%s\n",args[i]);
        free(args[i]);
    }
}
}

```

As for the parent process, we should discriminate whether we should let the child process run in the background. For normal case, there should be a wait() instruction in the main process, so that the shell will continue taking inputs only if the user command has finished. While for the case where "&" is specified, there should be no wait() instruction, we just leave the child process to end itself.

## Result Demonstration

The following screenshot demonstrates the basic function of our simple shell. Note that when we add the "&" after ls command, the "osh>" is printed out prior to the ls results, indicating that the shell process is not waiting for the child process to end. When we input exit, the shell quits normally.

```

(base) ltzhou@TonydeMacBook-Pro shell % gcc simple-shell.c -o osh
(base) ltzhou@TonydeMacBook-Pro shell % ./osh
osh>ls -l
total 2968
-rw-r--r--  1 ltzhou  staff   150  11  20  20:00 Makefile
-rw-r--r--@ 1 ltzhou  staff 1427711 11 13 13:52 Project 2 Shell.docx
drwxr-xr-x 26 ltzhou  staff   832  11  20  19:46 ch3
-rw-r--r--  1 ltzhou  staff  3009  11  26  22:01 history-shell.c
-rwxr-xr-x  1 ltzhou  staff 50096  11  26  22:15 osh
-rw-r--r--  1 ltzhou  staff   655  11  26  22:03 osh.out
-rw-r--r--  1 ltzhou  staff  3157  11  20  20:23 pid.c
-rw-r--r--  1 ltzhou  staff  5203  11  26  22:01 pipe-shell.c
drwxr-xr-x  3 ltzhou  staff    96  11  20  18:17 pipe-shell.dSYM
-rw-r--r--  1 ltzhou  staff  3878  11  26  22:02 redirect-shell.c
-rw-r--r--  1 ltzhou  staff  2062  11  26  22:14 simple-shell.c
drwxr-xr-x  3 ltzhou  staff    96  11  20  10:42 try.dSYM
-rw-r--r--@ 1 ltzhou  staff   162  11  26  22:05 ~$object 2 Shell.docx
osh>ls &
osh>Makefile      history-shell.c      pid.c                redirect-shell.c     ~$object 2 Shell.docx
osh>Project 2 Shell.docx  osh                  pipe-shell.c         simple-shell.c
osh>ch3            osh.out              pipe-shell.dSYM      try.dSYM
osh>
osh>exit
(base) ltzhou@TonydeMacBook-Pro shell %

```

# Unix Shell – History Feature

## Problem Description

In this part, we will modify the shell interface program so that it can provide a history feature to allow a user to execute the most recent command by entering `!!`. The command executed in this fashion will be echoed on the user's screen, and the command should also be placed in the history buffer as the next command. If there is no recent command in the history, an error message will pop up.

## Analysis & Implementation

### History Storage

In order to remember the last command that user inputs, we should allocate a space similar to the already existing args list to store the recent command. Here we use a linked list called `last_args` with length stored in `last_arg_cnt`. The configuration is similar to args list. It will be initialized to be an empty list. Since every command later will have at least one argument, we can check whether the pointer is `NULL` to identify whether the previous command exists.

```
int main(void)
{
    char *last_args[MAX_LINE/2 + 1];
    char *args[MAX_LINE/2 + 1]; /* command line (of 80) has max of 40 args */
    int should_run = 1;
    int last_arg_cnt = 0;

    last_args[last_arg_cnt] = NULL;
    ... ..
}
```

### History Execution

Here is the history execution logic. For normal case where user inputs a complete command, in addition to updating the args list, we should also move the current args list into the `last_args` list. For the history case, the current args will be `!!`. We should copy the contents from `last_args` to args so that the history commands can be executed in the same fashion as before in the execution part.

The implementation is shown as follows. Since args and `last_args` are implemented in the form of linked lists, some memory allocation operations are required.

```
if (arg_cnt == 1 && args[0][0] == '!' && args[0][1] == '!'){
    if (last_arg_cnt == 0){
        printf("Error: last command does not exist\n");
        continue;
    }
    printf("History Mode Executing: ");
    /** history mode, copy history storage to current args */
    for (int i=0; i<last_arg_cnt; ++i){
        if (i>=1){
            args[i] = (char*) malloc(MAX_LINE*sizeof(char));
        }
        strcpy(args[i], last_args[i]);
    }
}
```

```

        printf("%s ",args[i]);
    }
    printf("\n");
    arg_cnt = last_arg_cnt;
    args[arg_cnt] = NULL;
} else {
    /* copy current args to history storage */
    for (int i=0;i<arg_cnt;++i){
        if (i>=last_arg_cnt){
            last_args[i] =(char*) malloc(MAX_LINE*sizeof(char));
        }
        strcpy(last_args[i],args[i]);
    }
    for (int i=arg_cnt;i<last_arg_cnt;++i){
        free(last_args[i]);
    }
    last_arg_cnt = arg_cnt;
    last_args[last_arg_cnt] = NULL;
}

```

## Result Demonstration

The history feature is fully implemented in the frontend part. We leave the execution part unchanged. The demonstration result are listed below. Note that at first, calling "!!" will cause the shell to report an error. After we execute a command, the history is remembered and can be invoked again. When running "!!", "ls -l" is echoed and executed again.

```

(base) ltzhou@TonydeMacBook-Pro shell % gcc history-shell.c -o osh
(base) ltzhou@TonydeMacBook-Pro shell % ./osh
osh>!!
Error: last command does not exist
osh>ls -l
total 2968
-rw-r--r--  1 ltzhou  staff    150 11 20 20:00 Makefile
-rw-r--r--@ 1 ltzhou  staff 1427711 11 13 13:52 Project 2 Shell.docx
drwxr-xr-x 26 ltzhou  staff   832 11 20 19:46 ch3
-rw-r--r--  1 ltzhou  staff   3009 11 26 22:01 history-shell.c
-rwxr-xr-x  1 ltzhou  staff  50152 11 27 10:02 osh
-rw-r--r--  1 ltzhou  staff   655 11 26 22:03 osh.out
-rw-r--r--  1 ltzhou  staff  3220 11 27 09:55 pid.c
-rw-r--r--  1 ltzhou  staff   5203 11 26 22:01 pipe-shell.c
drwxr-xr-x  3 ltzhou  staff    96 11 20 18:17 pipe-shell.dSYM
-rw-r--r--  1 ltzhou  staff  3878 11 26 22:02 redirect-shell.c
-rw-r--r--  1 ltzhou  staff   2062 11 26 22:14 simple-shell.c
drwxr-xr-x  3 ltzhou  staff    96 11 20 10:42 try.dSYM
-rw-r--r--@ 1 ltzhou  staff   162 11 26 22:05 ~$object 2 Shell.docx
osh>!!
History Mode Executing: ls -l
total 2968
-rw-r--r--  1 ltzhou  staff    150 11 20 20:00 Makefile
-rw-r--r--@ 1 ltzhou  staff 1427711 11 13 13:52 Project 2 Shell.docx
drwxr-xr-x 26 ltzhou  staff   832 11 20 19:46 ch3
-rw-r--r--  1 ltzhou  staff   3009 11 26 22:01 history-shell.c
-rwxr-xr-x  1 ltzhou  staff  50152 11 27 10:02 osh
-rw-r--r--  1 ltzhou  staff   655 11 26 22:03 osh.out
-rw-r--r--  1 ltzhou  staff  3220 11 27 09:55 pid.c
-rw-r--r--  1 ltzhou  staff   5203 11 26 22:01 pipe-shell.c
drwxr-xr-x  3 ltzhou  staff    96 11 20 18:17 pipe-shell.dSYM
-rw-r--r--  1 ltzhou  staff  3878 11 26 22:02 redirect-shell.c
-rw-r--r--  1 ltzhou  staff   2062 11 26 22:14 simple-shell.c
drwxr-xr-x  3 ltzhou  staff    96 11 20 10:42 try.dSYM
-rw-r--r--@ 1 ltzhou  staff   162 11 26 22:05 ~$object 2 Shell.docx
osh>!!
History Mode Executing: ls -l
total 2968
-rw-r--r--  1 ltzhou  staff    150 11 20 20:00 Makefile
-rw-r--r--@ 1 ltzhou  staff 1427711 11 13 13:52 Project 2 Shell.docx
drwxr-xr-x 26 ltzhou  staff   832 11 20 19:46 ch3
-rw-r--r--  1 ltzhou  staff   3009 11 26 22:01 history-shell.c
-rwxr-xr-x  1 ltzhou  staff  50152 11 27 10:02 osh
-rw-r--r--  1 ltzhou  staff   655 11 26 22:03 osh.out
-rw-r--r--  1 ltzhou  staff  3220 11 27 09:55 pid.c
-rw-r--r--  1 ltzhou  staff   5203 11 26 22:01 pipe-shell.c
drwxr-xr-x  3 ltzhou  staff    96 11 20 18:17 pipe-shell.dSYM
-rw-r--r--  1 ltzhou  staff  3878 11 26 22:02 redirect-shell.c
-rw-r--r--  1 ltzhou  staff   2062 11 26 22:14 simple-shell.c
drwxr-xr-x  3 ltzhou  staff    96 11 20 10:42 try.dSYM
-rw-r--r--@ 1 ltzhou  staff   162 11 26 22:05 ~$object 2 Shell.docx
osh>exit

```



# Unix Shell – Redirecting I/O

## Problem Description

The shell should be modified to support the '<' and '>' redirection operators, where '>' redirects the output of a command to a file and '<' redirects the input to a command from a file. We assume that commands will contain either one input or one output redirection and will not contain both.

## Analysis & Implementation

### Parse I/O Directory

As for the frontend, we should check whether the command includes a redirection component. The following codes are inserted after the '&' check. We use the `redirect_output` and `redirect_input` as two flags indicating whether there is an I/O redirection, and use a string to remember the redirection directory.

```
int redirect_output = 0;
char* output_dir = NULL;
if (arg_cnt > 2 && args[arg_cnt-2][0] == '>'){
    redirect_output = 1;
    output_dir = args[arg_cnt-1];
    free(args[arg_cnt-2]);
    arg_cnt -= 2;
    args[arg_cnt] = NULL;
}

int redirect_input = 0;
char* input_dir = NULL;
if (arg_cnt > 2 && args[arg_cnt-2][0] == '<'){
    redirect_input = 1;
    input_dir = args[arg_cnt-1];
    free(args[arg_cnt-2]);
    arg_cnt -= 2;
    args[arg_cnt] = NULL;
}

if (redirect_input && redirect_output){
    printf("Input and Output can't be simultaneously redirected\n");
    continue;
}
```

### I/O Redirection

When executing the command with I/O redirection, we first call the `open()` API to get the I/O file descriptor. Then we use the `dup2()` function to duplicate the file descriptor to the standard I/O. The remaining codes are left unchanged.

```
if (pid == 0) {
    if (redirect_input){
        int input_f = open(input_dir,O_RDONLY);
```

```

    dup2(input_f,STDIN_FILENO);
}
if (redirect_output){
    int output_f = open(output_dir,O_WRONLY);
    dup2(output_f,STDOUT_FILENO);
}
execvp(args[0],args);
_exit(EXIT_SUCCESS);
}

```

## Result Demonstration

We first redirect the output of "ls -l" to out.txt, then we redirect the input of sort command from the output we have created. It can be seen that the redirection works well from the screenshot.

```

C history-shell.c  out.txt  X
El338 > Project > shell > out.txt
1 total 2968
2 -rw-r--r-- 1 ltzhou staff 150 11 20 20:00 Makefile
3 -rw-r--r--@ 1 ltzhou staff 1427711 11 13 13:52 Project 2 Shell.docx
4 drwxr-xr-x 26 ltzhou staff 832 11 20 19:46 ch3
5 -rw-r--r-- 1 ltzhou staff 3009 11 26 22:01 history-shell.c
6 -rw-r--r-- 1 ltzhou staff 50240 11 27 10:04 osh
7 -rw-r--r-- 1 ltzhou staff 655 11 26 22:03 osh.out
8 -rw-r--r-- 1 ltzhou staff 0 11 27 10:04 out.txt
9 -rw-r--r-- 1 ltzhou staff 3220 11 27 09:55 pid.c
10 -rw-r--r-- 1 ltzhou staff 5203 11 26 22:01 pipe-shell.c
11 drwxr-xr-x 3 ltzhou staff 96 11 20 18:17 pipe-shell.dSYM
12 -rw-r--r-- 1 ltzhou staff 3878 11 26 22:02 redirect-shell.c
13 -rw-r--r-- 1 ltzhou staff 2062 11 26 22:14 simple-shell.c
14 drwxr-xr-x 3 ltzhou staff 96 11 20 10:42 try.dSYM
15 -rw-r--r--@ 1 ltzhou staff 162 11 26 22:05 ~$object 2 Shell.docx
16

输出 终端 调试控制台 问题
osh>exit
(base) ltzhou@TonydeMacBook-Pro shell % gcc redirect-shell.c -o osh
(base) ltzhou@TonydeMacBook-Pro shell % ./osh
osh>ls -l > out.txt
osh>sort < out.txt
-rw-r--r-- 1 ltzhou staff 0 11 27 10:04 out.txt
-rw-r--r-- 1 ltzhou staff 150 11 20 20:00 Makefile
-rw-r--r-- 1 ltzhou staff 655 11 26 22:03 osh.out
-rw-r--r-- 1 ltzhou staff 2062 11 26 22:14 simple-shell.c
-rw-r--r-- 1 ltzhou staff 3009 11 26 22:01 history-shell.c
-rw-r--r-- 1 ltzhou staff 3220 11 27 09:55 pid.c
-rw-r--r-- 1 ltzhou staff 3878 11 26 22:02 redirect-shell.c
-rw-r--r-- 1 ltzhou staff 5203 11 26 22:01 pipe-shell.c
-rw-r--r--@ 1 ltzhou staff 162 11 26 22:05 ~$object 2 Shell.docx
-rw-r--r--@ 1 ltzhou staff 1427711 11 13 13:52 Project 2 Shell.docx
-rwxr-xr-x 1 ltzhou staff 50240 11 27 10:04 osh
drwxr-xr-x 3 ltzhou staff 96 11 20 10:42 try.dSYM
drwxr-xr-x 3 ltzhou staff 96 11 20 18:17 pipe-shell.dSYM
drwxr-xr-x 26 ltzhou staff 832 11 20 19:46 ch3
total 2968
osh>exit

```

## Unix Shell – Pipe Communication

### Problem Description

The final modification to our shell is to allow the output of one command to serve as input to another using a pipe, that is to say, the output of the first command separated by '|' will serve as the input to the second command. Both commands will run as separate processes and will communicate using the UNIX pipe() function. We assume that commands will contain only one pipe character and will not be combined with any redirection operators.

### Analysis & Implementation

#### Input Parsing

Our frontend will first traverse through the args and check whether there exists a '|'. pipe\_loc serves both as a flag indicating whether there is a pipe and an index to show where to start the second arguments.

A trick here is that when encountering the pipe symbol, we remove the string and set that entry in the args index to be NULL, so that when executing the first command, we can send the head of the args list into the execvp() function as usual, it will automatically stop at the pipe location. For the second command, we can just use pipe\_loc as an offset to tell execvp() where to begin reading commands.

```
int pipe_loc = 0;
if (! (redirect_input || redirect_output)){ /* detect pipe if no redirection */
    for (int i=0;i<arg_cnt;++i){
        if (args[i][0] == '|'){
            pipe_loc = i+1;
            free(args[i]);
            args[i] = NULL;
        }
    }
}
```

Note that we need to restore the args list and refill the '|' entry, so that it will not cause memory management confusion to later use.

```
else { /* parent process */
    int status;
    if (should_wait){
        (void)waitpid(pid, &status, 0);
    }
    if (pipe_loc){ /* re-fill the '|' in args */
        args[pipe_loc] = malloc(1);
    }
    for (int i=0;args[i]!=NULL;++i){
        // printf("%s\n",args[i]);
        free(args[i]);
    }
}
```

```
}
```

## Pipe Implementation through child process

As for the execution part, we implement the pipe function with two processes. The first part of the pipe will be executed as the child process of the second command. The communication between the two processes will be implemented using the UNIX pipe() function. After calling pipe(fd), the fd[] serves as two descriptors that can be passed to the UNIX function dup2(). By redirecting the standard I/O to the pipe correctly as indicated in the comment below, we can implement the function of pipe.

```
#define READ_END 0
#define WRITE_END 1

.....

if (pid == 0) {
    if (pipe_loc == 0){ /* normal mode */
        ... /* same as before */
    } else { /* pipe mode */
        int fd[2];
        if (pipe(fd) == -1) {
            fprintf(stderr, "Pipe failed");
            return 1;
        }
        pid_t pipe_pid = fork();
        if (pipe_pid == -1){
            perror("fork failed");
            exit(EXIT_FAILURE);
        } else if (pipe_pid == 0){ /* child: pipe1 process, write to pipe */
            /* close the unused end of the pipe */
            close(fd[READ_END]);
            /* read from the pipe */
            dup2(fd[WRITE_END], STDOUT_FILENO);
            execvp(args[0], args);
            /* close the write end of the pipe */
            close(fd[WRITE_END]);
        } else { /* parent: pipe2 process, read from pipe */
            /* close the unused end of the pipe */
            close(fd[WRITE_END]);
            /* write to the pipe */
            dup2(fd[READ_END], STDIN_FILENO);
            execvp(args[pipe_loc], args+pipe_loc);
            /* close the read end of the pipe */
            close(fd[READ_END]);
            int pipe_status;
            (void)waitpid(pipe_pid, &pipe_status, 0);
        }
        _exit(EXIT_SUCCESS);
    }
}
```

## Result Demonstration

In the first example, we create a pipe between ls and sort command, we can see that the sort command takes ls as an input and print out the sorted results.

```
(base) ltzhou@TonydeMacBook-Pro shell % gcc pipe-shell.c -o osh
(base) ltzhou@TonydeMacBook-Pro shell % ./osh
osh>ls | sort
Makefile
Project 2 Shell.docx
ch3
history-shell.c
osh
osh.out
out.txt
pid.c
pipe-shell.c
pipe-shell.dSYM
redirect-shell.c
simple-shell.c
try.dSYM
~$object 2 Shell.docx
osh>
```

In our second example, we are executing "ls | less", the less application takes input from ls, and user can read the result from ls within the application of less.

```
Makefile
Project 2 Shell.docx
ch3
history-shell.c
osh
osh.out
out.txt
pid.c
pipe-shell.c
pipe-shell.dSYM
redirect-shell.c
simple-shell.c
try.dSYM
~$object 2 Shell.docx
(END)
```

A slight fault with our implementation is that the pipe function will not work well with &, but since the semantics of a combination of pipe and background running is vague, we assume that users will not run such commands in our shell.

# Linux Kernel Module for Task Information

## Problem Description

In this project, we will write a Linux kernel module that uses the /proc file system for displaying a task's information based on its process identifier value pid.

## Analysis & Implementation

First we add the proc\_write function into the proc\_ops struct to activate the writing function.

```
static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
    .write = proc_write
};
```

### Writing to the /proc File System

Then in the proc\_write function body, we use sscanf to read the input of user, and store the input pid into the a static variable l\_pid. The l\_pid will be used in the proc\_read function.

```
char *k_mem;

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

/* copies user space usr_buf to kernel buffer */
if (raw_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 c
```

### Reading from the /proc File System

In the proc\_read function, we call the kernel function pid\_task() to find the task information struct. Then according to the kernel manual, we visit the command and state of the task, and print it out to users.

```
tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);

completed = 1;
```

```

if (tsk == NULL){
    return 0;
}

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

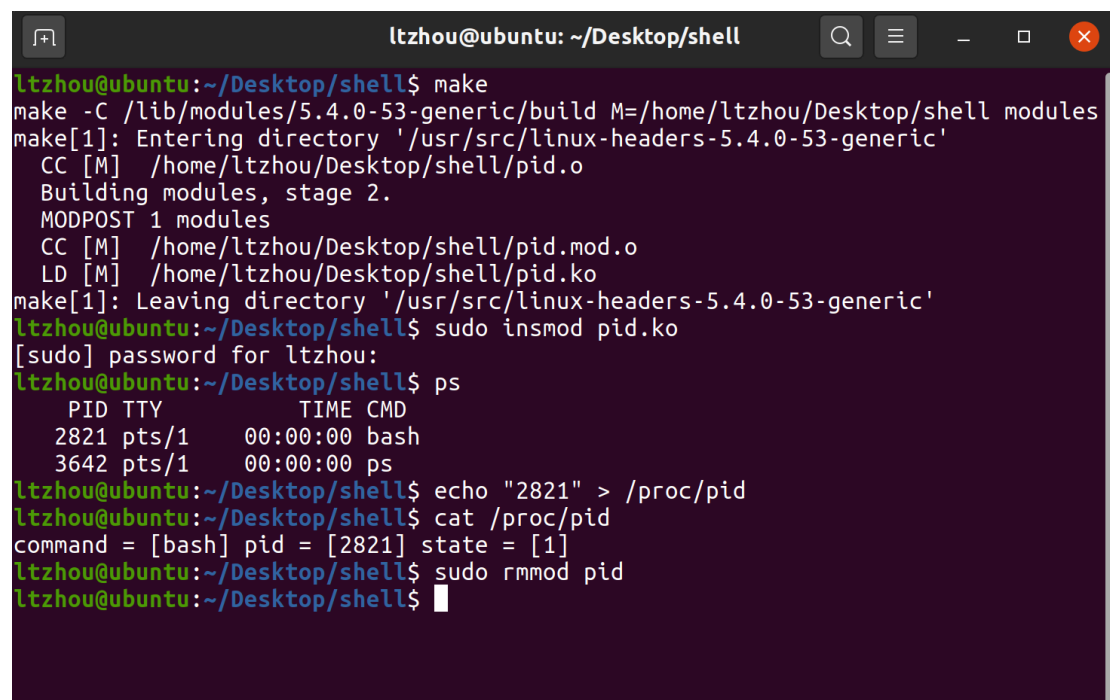
// copies the contents of kernel buffer to userspace usr_buf
if (raw_copy_to_user(usr_buf, buffer, rv)) {
    rv = -1;
}

return rv;

```

## Result Demonstration

In the following example, we write the current bash process pid into /proc/pid and read the /proc/pid. What we get from /proc/pid is the detail information of the task.



```

ltzhou@ubuntu: ~/Desktop/shell
ltzhou@ubuntu:~/Desktop/shell$ make
make -C /lib/modules/5.4.0-53-generic/build M=/home/ltzhou/Desktop/shell modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-53-generic'
CC [M] /home/ltzhou/Desktop/shell/pid.o
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/ltzhou/Desktop/shell/pid.mod.o
LD [M] /home/ltzhou/Desktop/shell/pid.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-53-generic'
ltzhou@ubuntu:~/Desktop/shell$ sudo insmod pid.ko
[sudo] password for ltzhou:
ltzhou@ubuntu:~/Desktop/shell$ ps
  PID TTY          TIME CMD
  2821 pts/1        00:00:00 bash
  3642 pts/1        00:00:00 ps
ltzhou@ubuntu:~/Desktop/shell$ echo "2821" > /proc/pid
ltzhou@ubuntu:~/Desktop/shell$ cat /proc/pid
command = [bash] pid = [2821] state = [1]
ltzhou@ubuntu:~/Desktop/shell$ sudo rmmod pid
ltzhou@ubuntu:~/Desktop/shell$

```