Project 2: UNIX Shell Programming & Linux Kernel Module for Task Information

1 UNIX Shell Programming

In this task, i will implement a UNIX Shell with following parts:

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

(2) Providing a history feature

(3) Adding support of input and output redirection

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

Design: My key designs is as follows:

- To parse the input command, i use **flatten()** function to remove extra space, then i use **tokenize()** function to transform string into array.
- After parsing, i check whether it is `!!` or `exit`, and give the result.
- Then i use **fork()** function to create the child process. The instruction will be execute in the child process.
- Then i use **concurrentFlag** function to check whether it asks for concurrent execution. If true, the parent process will not execute `wait(NULL)`. If false, the parent process will wait until the child process is done.
- In the child process, then i check whether the instruction needs a pipe. If true, **pipe(pipe_fd)** will be called to generate a pipe. It will be used to carry out data communication.
- If the instruction need to be redirected, **dup2()** function is used to redirect the input/output.
- As for the concrete instruction such as **ls**, i use **execvp()** to execute it.

The code of simple-shell.c is shown as follows.

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

#define MAX_LINE                80 /* 80 chars per line, per command */
#define bool                    int /* simulate boolean... */
#define true                    1
#define false                   0

```

```
13 void flatten(char *a, char *b) { // remove extra space, \n, \t
14         int length = strlen(a);
15         int current_location = 0;
16         bool blank_end = false;
17         for (int i = 0; i < length; i++) {
18                 if (a[i] == ' ' || a[i] == '\t' || a[i] == '\n') {
19                         if (blank_end) {
20                                 b[current_location++] = ' ';
21                         }
22                         blank_end = false;
23                 } else {
24                         b[current_location++] = a[i];
25                         blank_end = true;
26                 }
27         }
28         if (b[current_location - 1] == ' ') {
29                 b[current_location - 1] = 0;
30         }
31         return;
32 }
33
34 int tokenize(char *a, char **b) {  //tokenize flatten string
35         int num = 0; // token num
36         int current_location = 0; // location inside string
37         int length = strlen(a);
38         if (length == 0) return 0;
39         for(int i = 0; i < length; i++) {
40                 if (a[i] == ' ') {
41                         num++;
42                         current_location = 0;
43                 } else {
44                         b[num][current_location++] = a[i];
45                 }
46         }
47         return num + 1;
48 }
49
50 bool concurrentFlag(char *a) {
51         int length = strlen(a);
52         if (length == 0) return false;
53         return (a[length - 1] == '&');
54 }
55
56 int main(void) {
57         char *args[MAX_LINE/2 + 1];     /* command line (of 80) has max of 40 arguments
              */
58         char *raw_string;
59         char *flatten_string;
60         char *last_command;
61     int should_run = 1;
62         bool have_last_command = false;
```

```
63          raw_string = (char*) malloc(MAX_LINE * sizeof(char));
64          flatten_string = (char*) malloc(MAX_LINE * sizeof(char));
65          last_command = (char*) malloc(MAX_LINE * sizeof(char));
66      while (should_run){
67                  if (concurrentFlag(flatten_string) == false) wait(NULL);
68          printf("\033[42;37mosh>\033[2;7;0m"); // pretty print ~
69          fflush(stdout);
70                  memset(raw_string, 0, sizeof(raw_string));
71                  memset(flatten_string, 0, sizeof(raw_string));
72                  fflush(stdout);
73          fgets(raw_string, MAX_LINE, stdin);
74                  flatten(raw_string, flatten_string); // remove space \t \n from
                        raw_string
75
76                  if (strcmp(flatten_string, "exit") == 0) {
77                          should_run = 0;
78                          break;
79                  }
80                  if (strcmp(flatten_string, "!!") == 0) {
81                          if (have_last_command) {
82                                  strcpy(flatten_string, last_command);
83                                  fprintf(stdout, "%s\n", flatten_string);
84                          } else {
85                                  fprintf(stderr, "No commands in history.\n");
86                                  continue;
87                          }
88                  } else {
89                          have_last_command = true;
90                          strcpy(last_command, flatten_string);
91                  }
92
93                  pid_t pid;
94                  pid = fork();
95                  if (pid < 0) {
96                          fprintf(stderr, "Failed when creating a child process.\n");
97                          continue;
98                  }
99                  if (pid == 0) { // child
100                         for (int i = 0; i < MAX_LINE/2 + 1; i++) {
101                                 args[i] = (char*) malloc(MAX_LINE * sizeof(char));
102                                 memset(args[i], 0, sizeof(args[i]));
103                         }
104                         int command_num = tokenize(flatten_string, args);
105                         for (int i = command_num; i < MAX_LINE/2 + 1; i++) {
106                                 free(args[i]);
107                                 args[i] = NULL;
108                         }
109
110                         if (concurrentFlag(flatten_string)) {
111                                 free(args[command_num - 1]);
112                                 args[command_num - 1] = NULL;
```

```
113                     command_num = command_num - 1;
114             }
115
116             int pipe_loc = -1;
117             for (int i = 0; i < command_num; i++) {
118                     if (strcmp(args[i], "|") == 0) {
119                             pipe_loc = i;
120                             break;
121                     }
122             }
123
124             if (pipe_loc != -1) { // if pipe is need
125                     if (pipe_loc == 0 || pipe_loc == command_num -1) {
126                             fprintf(stderr, "Pipe command can't be conducted,
                                    expect more commands.\n");
127                     } else {
128                             int pipe_fd[2];
129                             int pipe_flag = pipe(pipe_fd);
130                             if (pipe_flag == -1) {
131                                     fprintf(stderr, "Pipe can't be
                                        established.\n");
132                             } else { // pipe is created and no error, just do
                                    it ~
133                                     pid = fork();
134                                     if (pid < 0) {
135                                             fprintf(stderr, "Failed when
                                                creating a child process.\n")
                                                ;
136                                     }
137                                     if (pid == 0) { // child of child
138                                             for (int i = pipe_loc; i <
                                                command_num; i++) {
139                                                     free(args[i]);
140                                                     args[i] = NULL;
141                                             }
142
143                                             close(pipe_fd[0]); // close read
144                                             if (dup2(pipe_fd[1],
                                                STDOUT_FILENO) < 0) {
145                                                     fprintf(stderr, "Dup
                                                        failed.\n");
146                                             } else {
147                                                     execvp(args[0], args);
148                                             }
149                                             close(pipe_fd[1]); // close write
150
151                                             for (int i = 0; i < pipe_loc; i
                                                ++) { // child of child
                                                menmory free
152                                                     free(args[i]);
153                                             }
```

```
154                                             free(raw_string);
155                                             free(flatten_string);
156                                             free(last_command);
157                                             exit(0);
158                                     } else { // child
159                                             wait(NULL);
160
161                                             for (int i = 0; i <= pipe_loc; i
                                                    ++) {
162                                                     free(args[i]);
163                                             }
164                                             for (int i = pipe_loc + 1; i <
                                                    command_num; i++) { // move 2
                                                    nd command to 1st
165                                                     args[i - pipe_loc - 1] =
                                                            args[i];
166                                             }
167                                             for (int i = command_num -
                                                    pipe_loc - 1; i < command_num
                                                    ; i++) {
168                                                     args[i] = NULL;
169                                             }
170                                             command_num = command_num -
                                                    pipe_loc - 1;
171
172                                             close(pipe_fd[1]);
173                                             if (dup2(pipe_fd[0], STDIN_FILENO
                                                    ) < 0) {
174                                                     fprintf(stderr, "Pipe dup
                                                            failed.\n");
175                                             } else {
176                                                     execvp(args[0], args);
177                                             }
178                                             close(pipe_fd[0]);
179                                     }
180                             }
181                     }
182             } else { // no pipe, consider redirect input/output
183                     bool input_redirect_flag = false;
184                     bool output_redirect_flag = false;
185                     char *file_name;
186                     file_name = (char*) malloc(MAX_LINE * sizeof(char));
187                     if (command_num >= 3) {
188                             if ((strcmp(args[command_num - 2], "<") == 0) ||
                                    (strcmp(args[command_num - 2], ">") == 0)) {
189                                     if (strcmp(args[command_num - 2], "<") ==
                                            0) {
190                                             input_redirect_flag = true;
191                                     } else {
192                                             output_redirect_flag = true;
193                                     }
```

```
194                                         strcpy(file_name, args[command_num - 1]);
195
196                                         free(args[command_num - 1]);
197                                         free(args[command_num - 2]);
198                                         args[command_num - 1] = NULL;
199                                         args[command_num - 2] = NULL;
200                                         command_num = command_num - 2;
201                                 }
202                         }
203                         int file_fd;
204                         if (input_redirect_flag == true) {
205                                 file_fd = open(file_name, O_RDONLY, 0644);
206                                 if (file_fd < 0) {
207                                         fprintf(stderr, "File doesn't exist.\n");
208                                 } else {
209                                         if (dup2(file_fd, STDIN_FILENO) < 0) {
210                                                 fprintf(stderr, "Dup failed.\n");
211                                         } else {
212                                                 execvp(args[0], args);
213                                                 close(file_fd);
214                                         }
215                                 }
216                         } else if (output_redirect_flag == true) {
217                                 file_fd = open(file_name, O_WRONLY | O_CREAT,
                                        0644);
218                                 if (file_fd < 0) {
219                                         fprintf(stderr, "Can't create file.\n");
220                                 } else {
221                                         if (dup2(file_fd, STDOUT_FILENO) < 0) {
222                                                 fprintf(stderr, "Dup failed.\n");
223                                         } else {
224                                                 execvp(args[0], args);
225                                                 close(file_fd);
226                                         }
227                                 }
228                         } else {
229                                 execvp(args[0], args); // normal, no pipe, no
                                        redirect
230                         }
231                         free(file_name);
232                 }
233
234                 for (int i = 0; i < command_num; i++) { // child memory free
235                         free(args[i]);
236                 }
237                 free(raw_string);
238                 free(flatten_string);
239                 free(last_command);
240                 exit(0);
241         } else { // father
242                 if (concurrentFlag(flatten_string) == false) wait(NULL);
```

```
                }
    }

        free(raw_string);
        free(flatten_string);
        free(last_command);
        return 0;
}
```

Since it is a simple program, i use the following command to compile it. (without Makefile)

```
gcc simple-shell.c -o shell
```

The result is shown as follows.

```
misaka@ubuntu:~/Documents/project2$ gcc simple-shell.c -o shell
misaka@ubuntu:~/Documents/project2$ ./shell
osh>!!
No commands in history.
osh>ls -a
.  ..  shell  simple-shell  simple-shell.c  test_redirect
osh>!!
ls -a
.  ..  shell  simple-shell  simple-shell.c  test_redirect
osh>ls -l > test_output
osh>sort < test_output
-rw-r--r-- 1 misaka misaka     0 Apr  6 22:36 test_output
-rw-r--r-- 1 misaka misaka   182 Apr  6 20:30 test_redirect
-rwxrw-rw- 1 misaka misaka  6781 Apr  6 22:35 simple-shell.c
-rwxrwxr-x 1 misaka misaka 21848 Apr  6 22:34 simple-shell
-rwxrwxr-x 1 misaka misaka 21848 Apr  6 22:36 shell
total 60
osh>ls -a
.  ..  shell  simple-shell  simple-shell.c  test_output  test_redirect
osh>ls -a &
osh>.  ..  shell  simple-shell  simple-shell.c  test_output  test_redirect
ls
shell  simple-shell  simple-shell.c  test_output  test_redirect
osh>ls -l | sort
-rw-r--r-- 1 misaka misaka   182 Apr  6 20:30 test_redirect
-rw-r--r-- 1 misaka misaka   299 Apr  6 22:36 test_output
-rwxrw-rw- 1 misaka misaka  6781 Apr  6 22:35 simple-shell.c
-rwxrwxr-x 1 misaka misaka 21848 Apr  6 22:34 simple-shell
-rwxrwxr-x 1 misaka misaka 21848 Apr  6 22:36 shell
total 64
osh>exit
misaka@ubuntu:~/Documents/project2$
```

图 1: Shell

2 Linux Kernel Module for Task Information

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

Design: My key design is as follows:

- In **proc_write()**, i use **sscanf()** function to get the pid value from input.
- Memory allocation is different in the kernel. I use **kmalloc()** and **kfree()** to allocate memory.
- In **proc_read()**, i use **pid_task(find_vpid(...), ...)** function to read the PCB information with a pid number.
- When there is a error (not valid pid number), i will print the kernel information to inform the user.

The code of pid.c is shown as follows.

```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 <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "pid"

/* the current pid */
static long l_pid;

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 proc_ops proc_op = {
        .proc_read = proc_read,
        .proc_write = proc_write
};

static int proc_init(void)
{
        proc_create(PROC_NAME, 0666, NULL, &proc_op);
        printk(KERN_INFO "/proc/%s created\n", PROC_NAME);
        return 0;
}

static void proc_exit(void)
{
        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];
```

```
49          static int completed = 0;
50          struct task_struct *tsk = NULL;
51
52          if (completed) {
53                  completed = 0;
54                  return 0;
55          }
56
57          tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
58          if (tsk == NULL) {
59                  printk(KERN_INFO "Not a valid pid %ld\n", l_pid);
60                  return 0;
61          }
62          completed = 1;
63          rv = sprintf(buffer, "command = [%s] pid = [%ld] state = [%ld]\n", tsk -> comm,
                l_pid, tsk -> state);
64          if (copy_to_user(usr_buf, buffer, rv)) {
65                  rv = -1;
66          }
67
68          return rv;
69  }
70
71  /**
72   * This function is called each time we write to the /proc file system.
73   */
74  static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count,
        loff_t *pos)
75  {
76          char *k_mem;
77
78          k_mem = kmalloc(count, GFP_KERNEL);
79
80          if (copy_from_user(k_mem, usr_buf, count)) {
81                  printk( KERN_INFO "Error copying from user\n");
82                  return -1;
83          }
84
85          sscanf(k_mem, "%ld", &l_pid);
86
87          kfree(k_mem);
88
89          return count;
90  }
91
92  /* Macros for registering module entry and exit points. */
93  module_init( proc_init );
94  module_exit( proc_exit );
95
96  MODULE_LICENSE("GPL");
97  MODULE_DESCRIPTION("Pid");
```

```
98 MODULE_AUTHOR("MisakaCenter");
```

The Makefile for this task is written as follows.

```
1 obj-m := pid.o
2 all:
3         make -C /usr/src/linux-5.11.3/ M=$(shell pwd) modules
4 clean:
5         make -C /usr/src/linux-5.11.3/ M=$(shell pwd) clean
```

The result is shown as follows.

```
misaka@ubuntu:~/Documents/project2/pid$ make
make -C /usr/src/linux-5.11.3/ M=/home/misaka/Documents/project2/pid modules
make[1]: Entering directory '/usr/src/linux-5.11.3'
  CC [M]  /home/misaka/Documents/project2/pid/pid.o
  MODPOST /home/misaka/Documents/project2/pid/Module.symvers
  CC [M]  /home/misaka/Documents/project2/pid/pid.mod.o
  LD [M]  /home/misaka/Documents/project2/pid/pid.ko
make[1]: Leaving directory '/usr/src/linux-5.11.3'
misaka@ubuntu:~/Documents/project2/pid$ sudo dmesg -C
misaka@ubuntu:~/Documents/project2/pid$ sudo insmod pid.ko
misaka@ubuntu:~/Documents/project2/pid$ sudo dmesg
[13443.032003] /proc/pid created
misaka@ubuntu:~/Documents/project2/pid$ echo "1" > /proc/pid
misaka@ubuntu:~/Documents/project2/pid$ cat /proc/pid
command = [systemd] pid = [1] state = [1]
misaka@ubuntu:~/Documents/project2/pid$ echo "233" > /proc/pid
misaka@ubuntu:~/Documents/project2/pid$ cat /proc/pid
misaka@ubuntu:~/Documents/project2/pid$ sudo dmesg
[13443.032003] /proc/pid created
[13485.801628] Not a valid pid 233
misaka@ubuntu:~/Documents/project2/pid$ echo "2" > /proc/pid
misaka@ubuntu:~/Documents/project2/pid$ cat /proc/pid
command = [kthreadd] pid = [2] state = [1]
misaka@ubuntu:~/Documents/project2/pid$ sudo rmmod pid
misaka@ubuntu:~/Documents/project2/pid$ sudo dmesg
[13443.032003] /proc/pid created
[13485.801628] Not a valid pid 233
[13540.301943] /proc/pid removed
```

图 2: Task information