CS356 Project 2

Name : HaotianXue

Student ID : 518021910506

Email : xavihart@sjtu.edu.cn

Introduction

This is the report for Project2 of course CS356, also for chapter 3 in COS book. This project is divided into two sections.

In the first part, we designed a simple shell which can simulate some functions of linux shell, which is supported by utilization of process operations and pipe operations in linux.

In the second part we designed a Linux kernel module that uses the /proc file system for displaying a task's information based on its process identifier value, which is supported by what we have done in Project1.

Simple Shell Implementation

Key Functions of Simple Shell

To simulate the Linux Shell, we should realize the following functions:

- Users can input basic command they want to execute, like `ls`, `ps` ,`cat ...`
- Users can input `!!` to reuse the last command, which will through out a warning is no history exists
- Users can use `&` to decide whether we should wait for the child command to be done

- Users can use `>` and `<` to redirect the file input and output
- Users can use `|` to do pipe communication between two processes, e.g. `ls | less`, `ls -l | sort`
- Users can use `exit` to close the simple shell

Implementation Details

Basic Command Execution

The program is run in a while loop, and we use `should_run` to decide if we can run, then we try to get the command from user input:

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

Not considering special cases, we use `execvp(args[0], args)` to execute the command simulating the shell, after that we continue the loop.

Also, we fork a child process to run the execution.

History Recalling

If we get the command `!!`, then we need to replace the current command with the history command, if there is no history command, we through out warnings. If the command is not `!!` we need to save the current command for use:

```
 if(strcmp(arg[0], "!!") == 0){
           hist = 1;
       if(!has_hist){
               printf("There is no history command!\n");
           n = 0;
               hist = 0;
           continue;
       }
       else{
         //  printf("last_n:%d\n",last_n);
           for(int i = 0;i < last_n;++i){
               args[i] = last_args[i];
               printf("%s ", args[i]);
           }
               printf("\n");
               n = last_n;
       }
      }else{
         // printf("copy to last:%d\n", n);
          for(int i = 0;i < n;++i){
                 strcpy(last_args[i], args[i]); //without the last "NULL"
          }
              last_n = n;
      }
```

Termination Detection

We also need to check if the command is `exit` to determine whether or not we should close down the shell:

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

Wait-or-not Detection

In this part we check the `&` symbol to decide if we should wait for the child process to be done:

```
if(strcmp(args[n- 1], "&") ==  0){
       n -- ;
       wait_ = 0;
       args[n] = NULL;
}
```

Redirection

We need to detect if the second to last command is `>` or `<` to make file redirection, if needed, we use `dup2` function to do that, this is done in the `pid == 0` part:

```
if(n >= 3 && strcmp(args[n - 2], ">") == 0){
              int f;
              f = open(args[n - 1],  O_CREAT|O_RDWR|O_TRUNC, S_IRUSR|S_IWUSR);
              dup2(f, STDOUT_FILENO);
              args[n - 1] = NULL;
              args[n - 2] = NULL;
              n -= 2;
}
if(n >= 3 && strcmp(args[n - 2], "<") == 0){
              int f2 = open(args[n - 1], O_RDONLY);
              dup2(f2, STDIN_FILENO);
              args[n - 2] = NULL;
              args[n - 1] = NULL;
              n -= 2;
}
```

Pipe communication

If there is `|` is the command, we use the output of the first command as the input for the second half. For example if we have `ls -l | less`, then the output for `ls -l` is input into `less`. We use the `pipe(fd)` function to implement this:

pipe detection

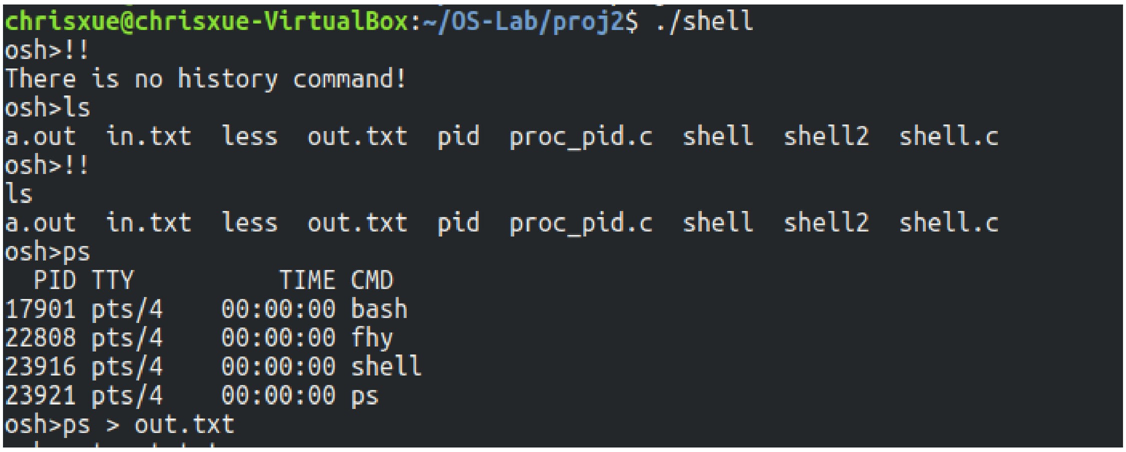
```
for(int i = 0;i < n;++i){
         if(strcmp(args[i], "|") == 0){
             use_pipe = 1;
         args[i] = NULL;
         for(int k = i + 1;k < n;++k){
             pipes[pipe_len ++] = args[k];
         }
                 n -= pipe_len;
                // printf("pipe output : %s, pipe len:%d\n", pipes[0], pipe_len);
             break;
         }
     }
```

pipe implementation

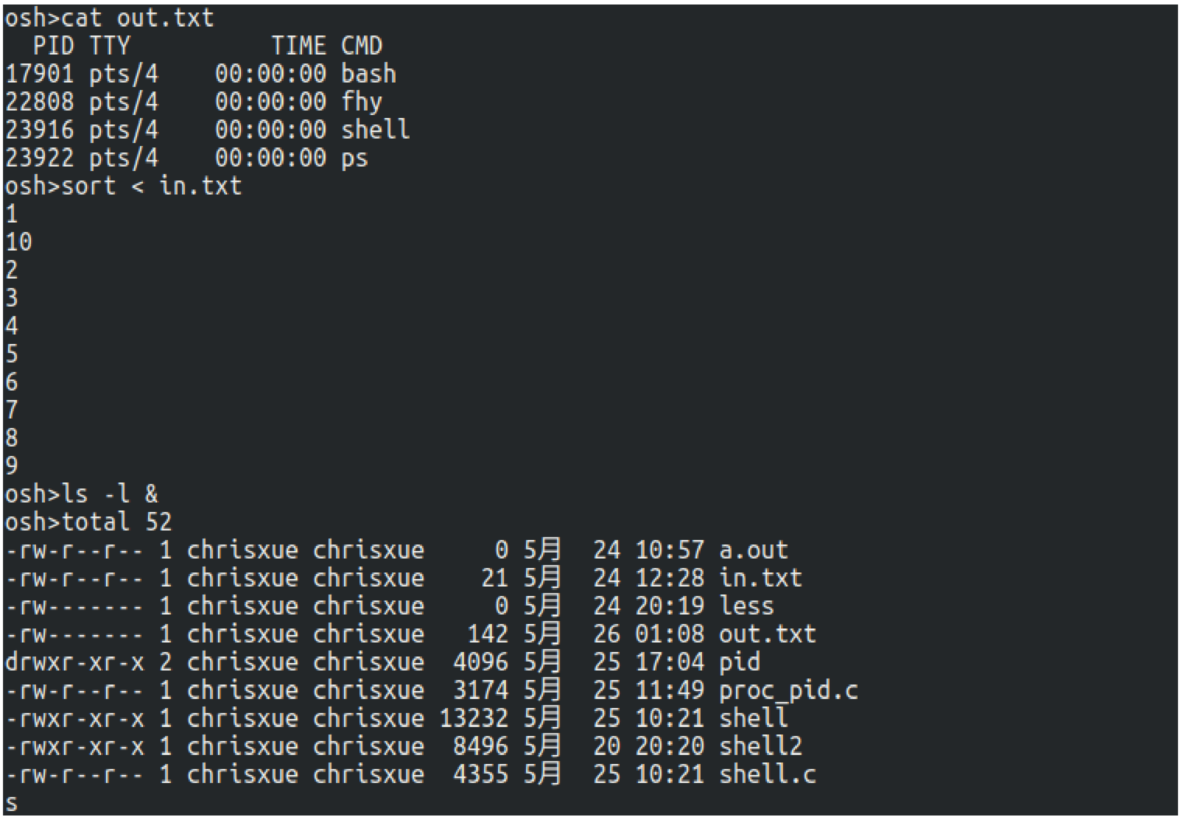
```
if(use_pipe){
                // printf("use pipe\n");
       if(pipe(pipe_fd) == -1){
             fprintf(stderr, "Pipe created failed!------\n");
             return 1;
       }

       pid_t pipe_pid;
       pipe_pid = fork();
       if(pipe_pid < 0){
             fprintf(stderr, "Fork failed(pipe)\n");
             return 1;
        }
       else if(pipe_pid == 0){
             close(pipe_fd[READ_END]);
             dup2(pipe_fd[WRITE_END], STDOUT_FILENO);
             execvp(args[0], args);
             close(pipe_fd[WRITE_END]);
             exit(0);
        }else{
            close(pipe_fd[WRITE_END]);
            dup2(pipe_fd[READ_END], STDIN_FILENO);
            pipes[pipe_len] = NULL;
            execvp(pipes[0], pipes);
            close(pipe_fd[READ_END]);
            wait(NULL);
        }

}
```

Tests and Results

- Test basic command, redirections, history and wait-or-not

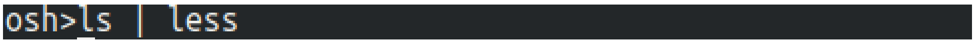


```
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2$ ./shell
osh>!!
There is no history command!
osh>ls
a.out  in.txt  less  out.txt  pid  proc_pid.c  shell  shell2  shell.c
osh>!!
ls
a.out  in.txt  less  out.txt  pid  proc_pid.c  shell  shell2  shell.c
osh>ps
  PID TTY          TIME CMD
17901 pts/4    00:00:00 bash
22808 pts/4    00:00:00 fhy
23916 pts/4    00:00:00 shell
23921 pts/4    00:00:00 ps
osh>ps > out.txt
```

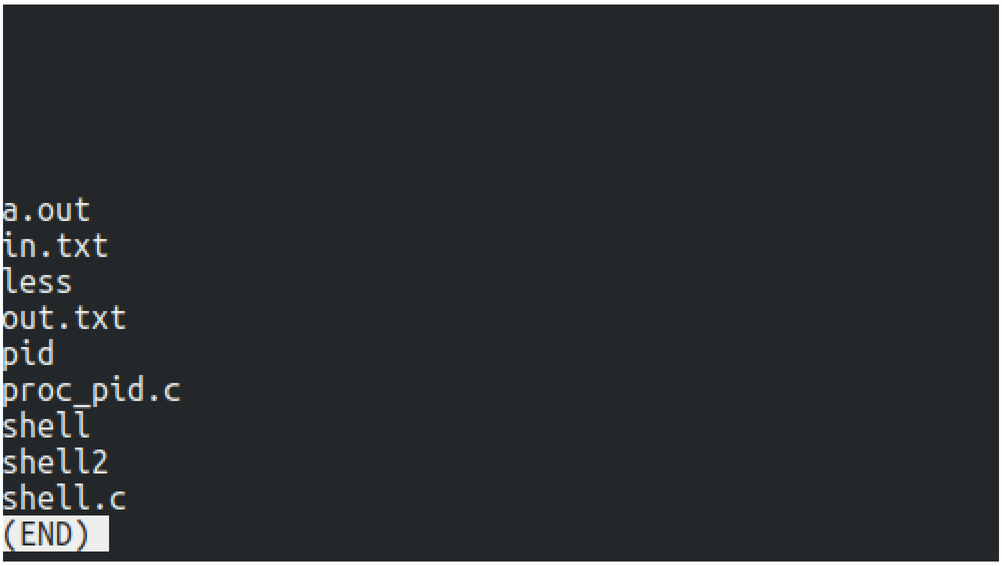


```
osh>cat out.txt
  PID TTY          TIME CMD
17901 pts/4    00:00:00 bash
22808 pts/4    00:00:00 fhy
23916 pts/4    00:00:00 shell
23922 pts/4    00:00:00 ps
osh>sort < in.txt
1
10
2
3
4
5
6
7
8
9
osh>ls -l &
osh>total 52
-rw-r--r-- 1 chrisxue chrisxue     0 5月  24 10:57 a.out
-rw-r--r-- 1 chrisxue chrisxue    21 5月  24 12:28 in.txt
-rw------- 1 chrisxue chrisxue     0 5月  24 20:19 less
-rw------- 1 chrisxue chrisxue   142 5月  26 01:08 out.txt
drwxr-xr-x 2 chrisxue chrisxue  4096 5月  25 17:04 pid
-rw-r--r-- 1 chrisxue chrisxue  3174 5月  25 11:49 proc_pid.c
-rwxr-xr-x 1 chrisxue chrisxue 13232 5月  25 10:21 shell
-rwxr-xr-x 1 chrisxue chrisxue  8496 5月  20 20:20 shell2
-rw-r--r-- 1 chrisxue chrisxue  4355 5月  25 10:21 shell.c
s
```

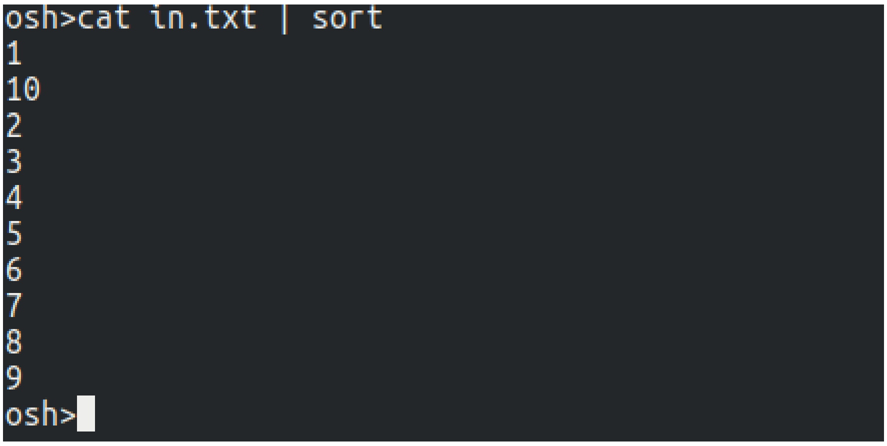
- test for pipe



```
osh>ls | less
```



```
a.out
in.txt
less
out.txt
pid
proc_pid.c
shell
shell2
shell.c
(END)
```



```
osh>cat in.txt | sort
1
10
2
3
4
5
6
7
8
9
osh>
```

Linux Kernel Module for Task Information

Key Features

We use /proc system to check information for certain process, and the key features are:

- Using linux /proc file system to implement
- Deal with user write to /proc file
- Return process information to users through /proc read

Implementation Details

The main structure is from the source code for COS book.

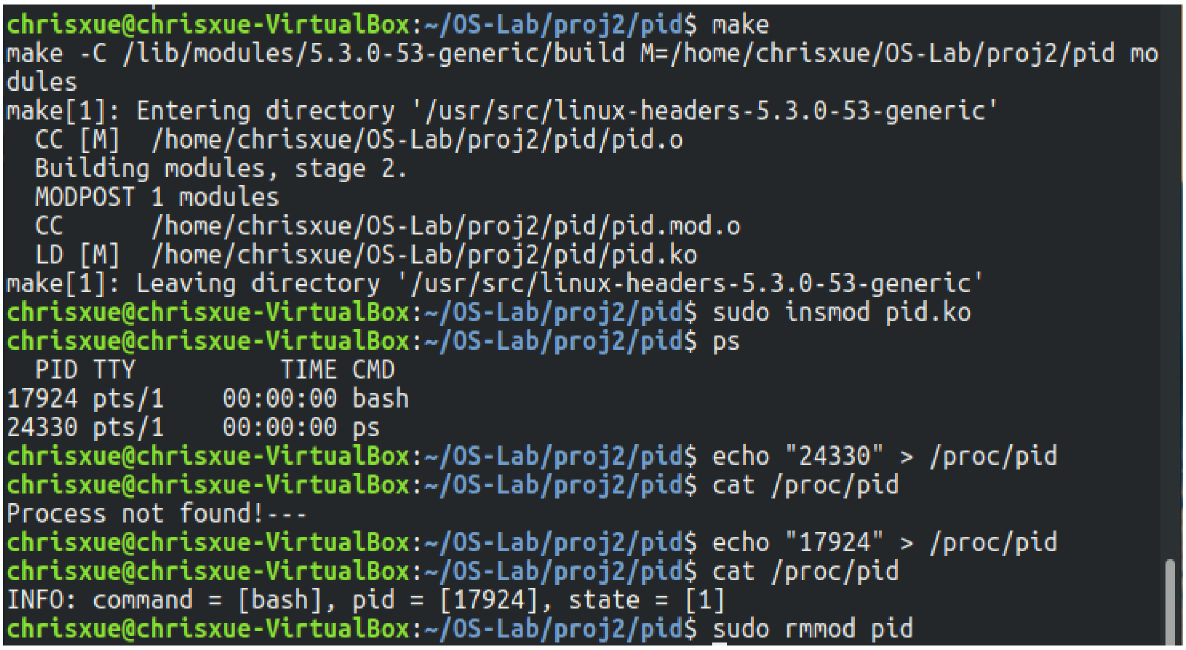
We need to change the `proc_read` and `proc_write` , in read part we deal with user read, we get he target process id and use `pid_task` structure to get the information includes `command`, `pid` and `state`. Then we check if the process exist in current state.

```
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 (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 count;
}
```

```
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);

        completed = 1;

    if (tsk == NULL)  // pid not exist in current state
    {
        rv = sprintf(buffer, "Process not found!---\n");

    }
    else
    {
        int state;
                pid_t pid;
                char command[MAX_LINE];
                state = tsk -> state;
        pid = tsk -> pid;

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

    }
```

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

        return rv;
}
```

Tests and Results



```
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ make
make -C /lib/modules/5.3.0-53-generic/build M=/home/chrisxue/OS-Lab/proj2/pid mo
dules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-53-generic'
  CC [M]  /home/chrisxue/OS-Lab/proj2/pid/pid.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/chrisxue/OS-Lab/proj2/pid/pid.mod.o
  LD [M]  /home/chrisxue/OS-Lab/proj2/pid/pid.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-53-generic'
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ sudo insmod pid.ko
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ ps
  PID TTY          TIME CMD
17924 pts/1    00:00:00 bash
24330 pts/1    00:00:00 ps
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ echo "24330" > /proc/pid
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ cat /proc/pid
Process not found!---
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ echo "17924" > /proc/pid
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ cat /proc/pid
INFO: command = [bash], pid = [17924], state = [1]
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj2/pid$ sudo rmmod pid
```

(Since `ps` command is done when we execute the next command, so it does not exit in current state)

Conclusion

Through this project, I implement a simple shell using knowledge of process and pipe communication, which we learn more about the programming for simple multi-process program under linux environment. Also I learn to write linux kernel modules for task information and have a better understanding of the /proc file system.

Thanks for the instructions and useful help offered by Prof. Wu and all the TAs!