Project2:UNIX Shell Programming & Linux Kernel Module for Task Information

Name: 韩冰

Number: 516030910523

Environment:

Ubuntu 14.04 (Vmware Workstation)

kernel-version: 4.4.0-31-generic

1. Unix Shell Programming

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. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX fork(), exec(), wait(), dup2(), and pipe() system calls and can be completed on any Linux, UNIX, or macOS system.

1. Executing Command in a Child Process

Trying to program a Unix, we must design a function to analysis command because what we input is just the string. In this function, we split the string inputed into servel strings included in an array according the space or '\t'. And judge whether the command is "exit", if true, then exit all the function, else do the next step.

```
int analysis_command(){
    // 无关代码已省略
    char *s = command;
    int i=0,j=0,state=0;
    strcat(command," ");
    while(*s){
        switch(state){
            case 0:
                if(!isspace(*s))
                    state=1;
                else
                    s++;
                break;
            case 1:
                if(isspace(*s)){
                    argv[i][j]='\0';
                    i++;
                    j=0;
                    state=0;
                }
                else{
                    argv[i][j]=*s;
                    j++;
                }
                s++;
                break;
            }
        }
        num_argv = i;
        if(num_argv==0)
            return 0;
        if(strcmp(argv[0],"exit")==0)
            exit(0);
        }
        return 1;
    }
```

Now, we get the command which has been split. Then we need to run the command in the child process using the function execvp() after forking, just like in the below:

```
int run_command(){
    //无关代码已省略
    int i,j;
    int f;
    char *argv_tmp[50];
    // copy a argv
    for(i=0;i<num_argv;++i){
        argv_tmp[i]=argv[i];
    }
    argv_tmp[i] = NULL;
    int pid = fork();
    if(pid<0){
        // error
        perror("fork error!");
        exit(0);
    }
    else if(pid>0){
        // father process
            waitpid(pid,NULL,0);
    }
    else{
        // child process
                execvp(argv_tmp[0],argv_tmp);
        }
    }
    return 1;
}
```

2. Create a History Feature

In the normal Unix shell, you press the 'up' key, the edit box will be filled with the last command. In this Unix Shell, if we input the "!!", it will run the last command.

I create a global variable, everytime we input a command, after we judge whether this is special command such as "exit" or "!!". Then I will store the command into the global variable. If the current command is "!!", I will fetch the command from the global variable, and pass the command to next function to run it.

```
// store the history command
char history_command[100]={0};
int analysis_command(){
    //..........无关代码已省略
    if(strcmp(argv[0],"!!")==0){
        // if no info stored in history
        if(!history_command[0]){
            printf("No commands in history.\n");
            return 0;
        }
        // copy the history_command to current command, restart the function
        strcpy(command,history_command);
        return analysis_command();
    }
    int t=0;
    // store the command into history_command before running
    while(command[t]){
        history_command[t] = command[t];
        ++t;
    }
    strcpy(history_command,command);
    if(strcmp(argv[0],"exit")==0)
        exit(0);
    return 1;
}
```

3. Redirecting Input and Output

My shell should support the ">" and "<" redirection operators.

To realize it, in the run_command function, I will search the '<' or '>' in the argv. If we found one, then I will reagard the argv before the symbol is the command, the argv after the symbol is the filename. By using the open function we can read or write the data in the file. The function we must using is dup2(), using this function, we can connect the filestream and STD_stream.

```
int run_command(){
    enum specify type=NORMAL;
    int i,j;
    int f;
    char *argv_tmp[50];
    char *filename;
    for(i=0;i<num_argv;++i){ // search the < or > symbols
        if(strcmp(argv_tmp[i],"<")==0){
            f++;
            filename = argv_tmp[i+1];
            argv_tmp[i]=NULL;
            type = IN_REDIRECT;
        }
        else if(strcmp(argv_tmp[i],">")==0){
            f++;
            filename = argv_tmp[i+1];
            argv_tmp[i]=NULL;
            type = OUT_REDIRECT;
        }
    }
    int pid = fork();
    int in,out;
    int pd[2];
    if(pid<0){ //省略 }
    else if(pid>0){//省略}
    else{
        // child process
        switch(type){
            case IN_REDIRECT: // <
                in = open(filename,O_RDONLY);
                dup2(in,STDIN_FILENO);
                execvp(argv_tmp[0],argv_tmp);
                close(in);
                break;
            case OUT_REDIRECT: // >
                out = open(filename,O_WRONLY|O_CREAT, 0666);
                dup2(out,STDOUT_FILENO);
                execvp(argv_tmp[0],argv_tmp);
                close(out);
                break;
        }
    }
    return 1;
}
```

4. Communication via a Pipe

Similar with above, now we need to search the '|' symbol, if we found it, so we can reagrd that the argv before it is the first command and the argv after that is the second command. In the child process, we need to fork again, because we need two process to run two commands.

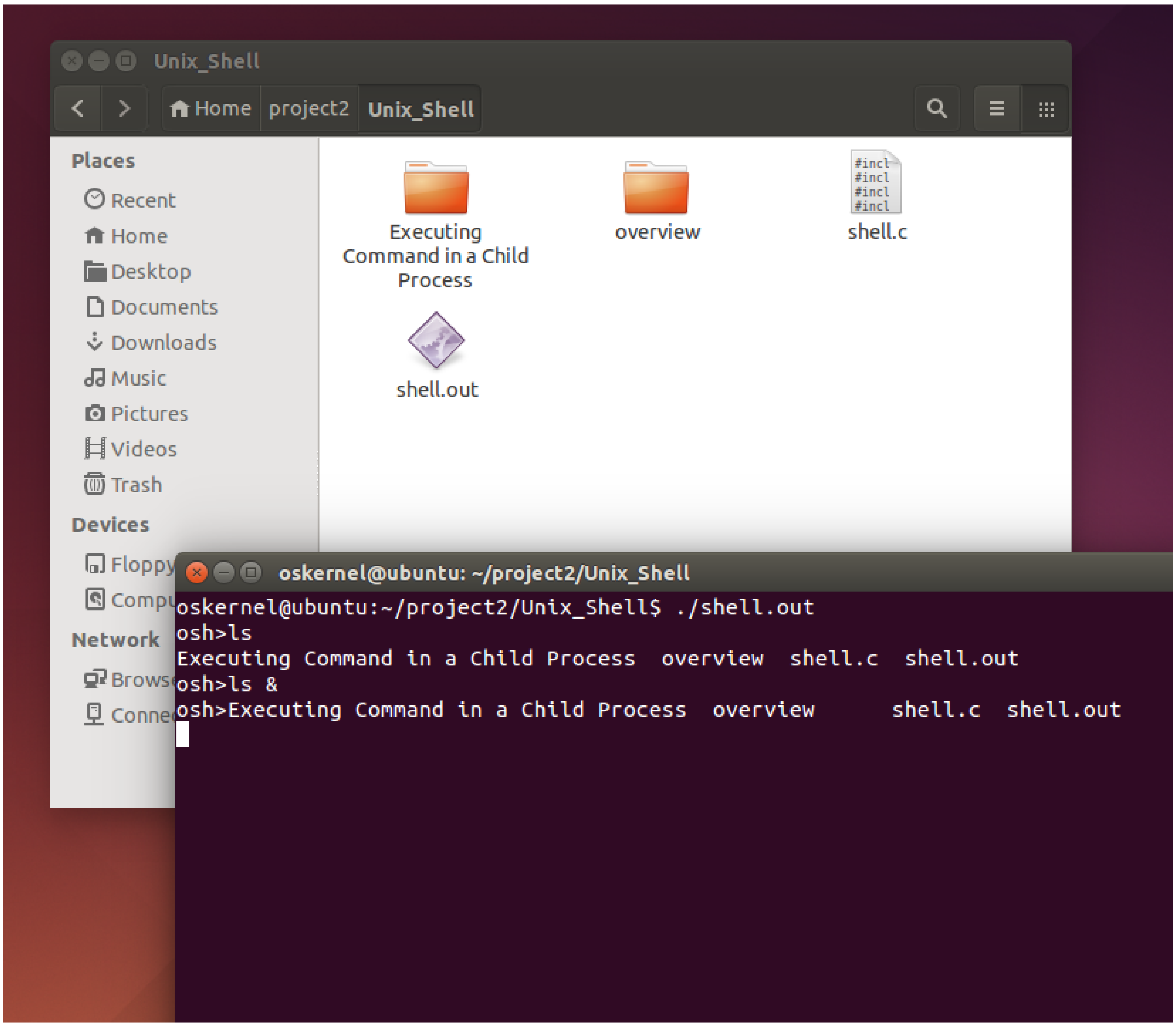
```
int run_command(){
    enum specify type=NORMAL;
    int i,j;
    int f;
    char *argv_tmp[50];
    char *argv_tmp2[50];
    for(i=0;i<num_argv;++i){
        if(strcmp(argv_tmp[i],"|")==0){
            f++;
            type = PIPE;
            argv_tmp[i]=NULL;
            for(j=i+1;j<num_argv;++j){
                argv_tmp2[j-i-1]=argv[j];
            }
```

```
                argv_tmp2[j-i-1] = NULL;
            }
        }
// 省略
        else{
            // child process
            switch(type){
                    // 省略
                case PIPE:
                    pipe(pd);
                    int pid2 = fork();
                    if(pid2<0){
                        perror("fork error!");
                        exit(0);
                    }
                    else if(pid2>0){
                        //father process
                        close(pd[1]); // close write pipe
                        dup2(pd[0],STDIN_FILENO);
                        execvp(argv_tmp2[0],argv_tmp2);
                    }
                    else{
                        // child process
                        close(pd[0]);// close read pipe
                        dup2(pd[1],STDOUT_FILENO);
                        execvp(argv_tmp[0],argv_tmp);
                    }
                    break;}}
        return 1;
}
```

Result:

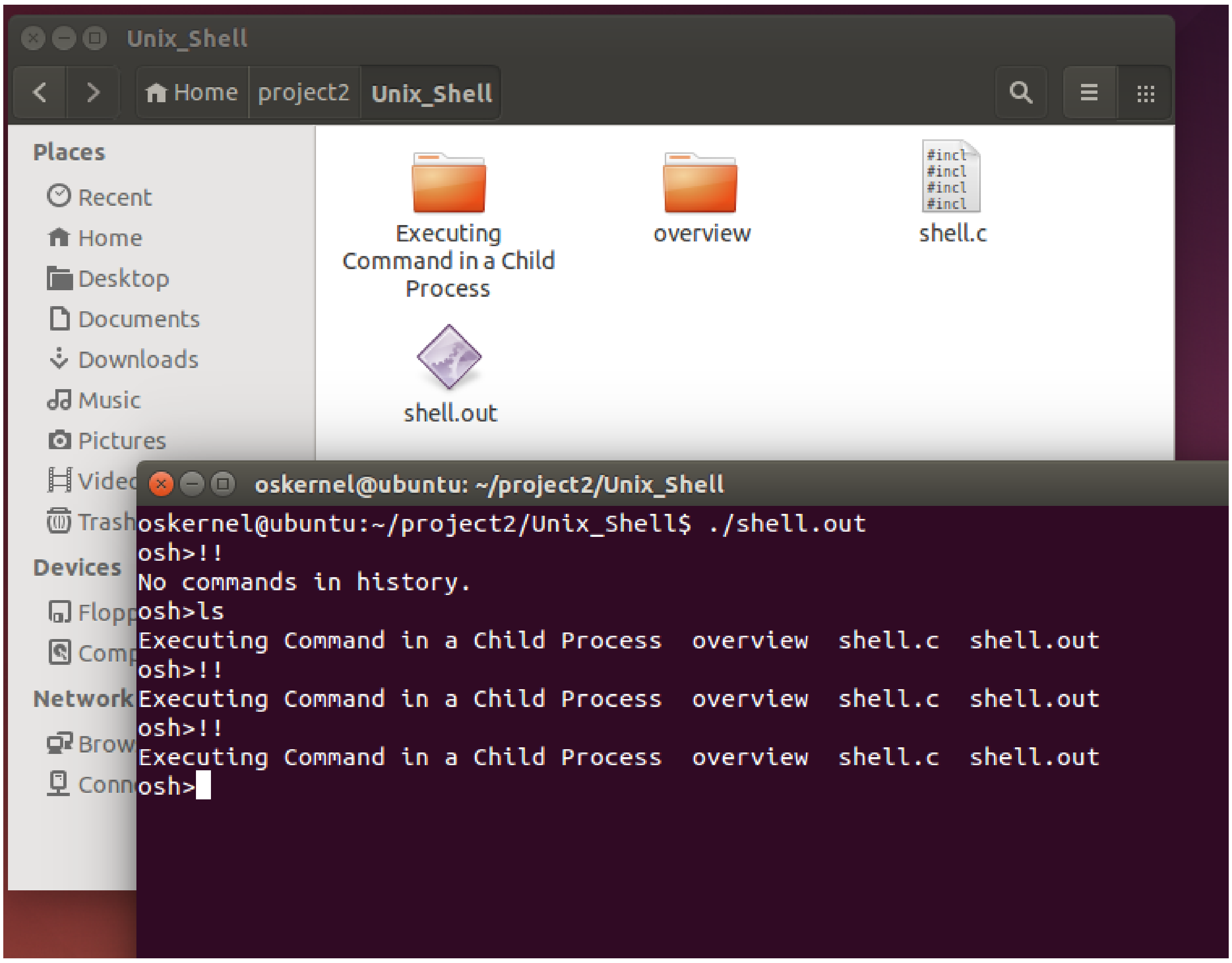
We will test the command and give the result as follow.

test normal command and background



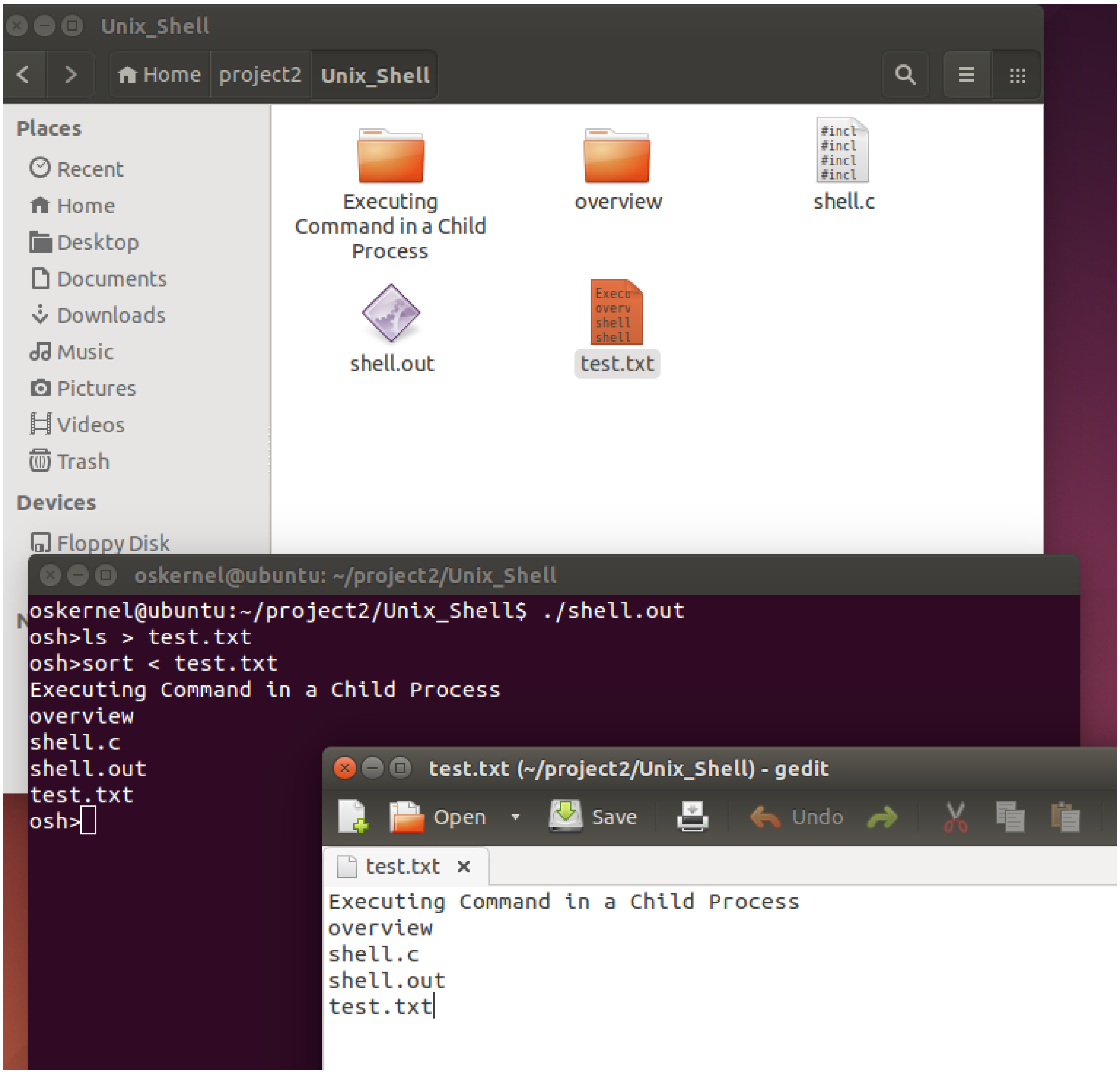
```
oskernel@ubuntu:~/project2/Unix_Shell$ ./shell.out
osh>ls
Executing Command in a Child Process  overview  shell.c  shell.out
osh>ls &
osh>Executing Command in a Child Process  overview      shell.c  shell.out
```

test history feature



```
oskernel@ubuntu:~/project2/Unix_Shell$ ./shell.out
osh>!!
No commands in history.
osh>ls
Executing Command in a Child Process  overview  shell.c  shell.out
osh>!!
Executing Command in a Child Process  overview  shell.c  shell.out
osh>!!
Executing Command in a Child Process  overview  shell.c  shell.out
osh>
```

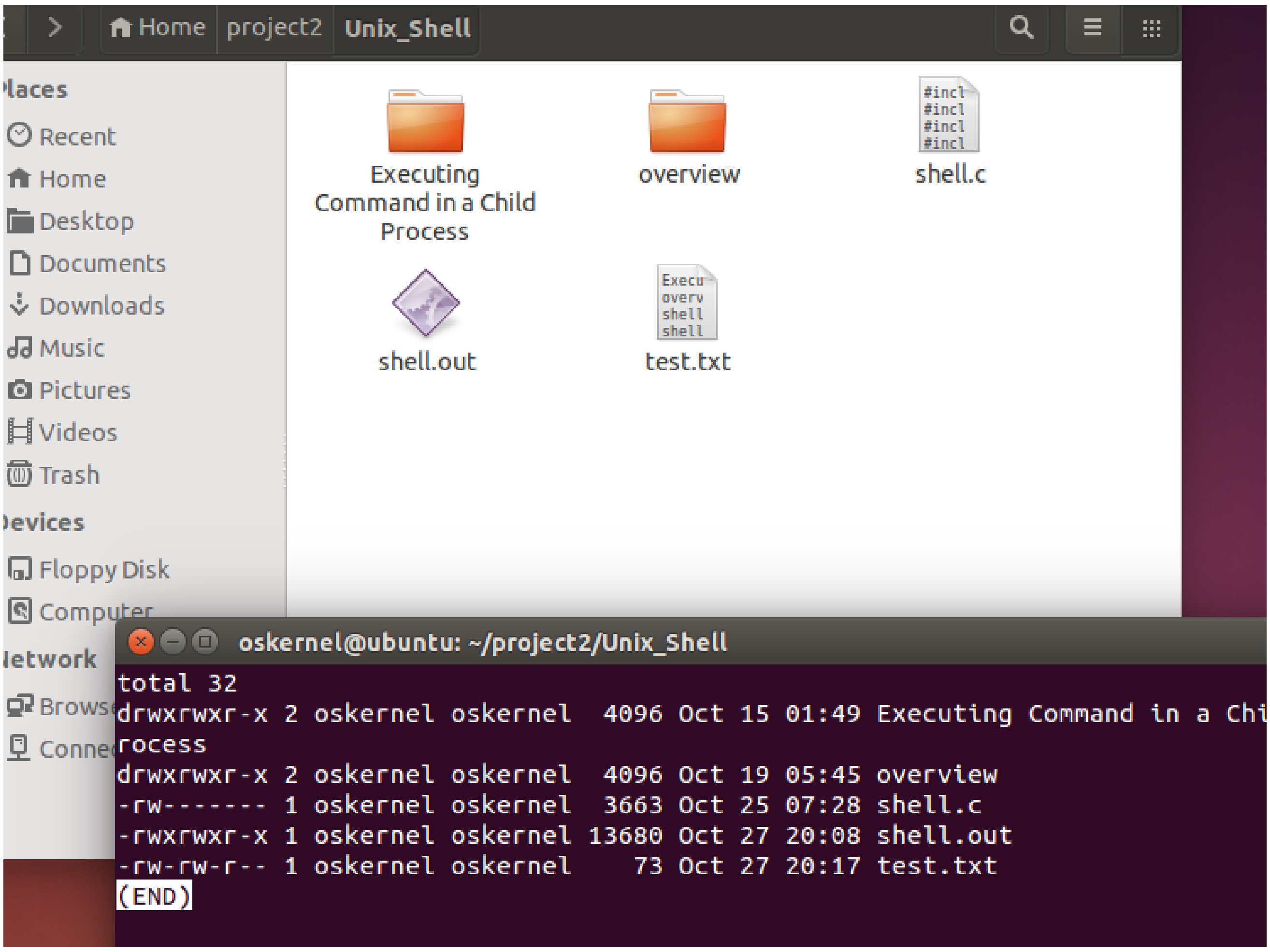
test redirection



```
oskernel@ubuntu:~/project2/Unix_Shell$ ./shell.out
osh>ls > test.txt
osh>sort < test.txt
Executing Command in a Child Process
overview
shell.c
shell.out
test.txt
osh>
```

```
Executing Command in a Child Process
overview
shell.c
shell.out
test.txt
```

test pipe(ls -l | less)



```
total 32
drwxrwxr-x 2 oskernel oskernel  4096 Oct 15 01:49 Executing Command in a Chi
rocess
drwxrwxr-x 2 oskernel oskernel  4096 Oct 19 05:45 overview
-rw------- 1 oskernel oskernel  3663 Oct 25 07:28 shell.c
-rwxrwxr-x 1 oskernel oskernel 13680 Oct 27 20:08 shell.out
-rw-rw-r-- 1 oskernel oskernel    73 Oct 27 20:17 test.txt
(END)
```

2. Linux Kernel Module for Task Information

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

1.Writing to the /proc File System

Firstly, we need add the proc_write function to the proc_ops struct. Then we will design our write function.

In the write function ,we need to allocate kernel memory at first, then we use copy_from_user function to copy the string we input in the user space to the kernel space. At this time, I found that the string will not include the '\0', so I set the tail is 0. After that, I use sscanf to transfer the string to long int. Remember to free the kernel memory before returning.

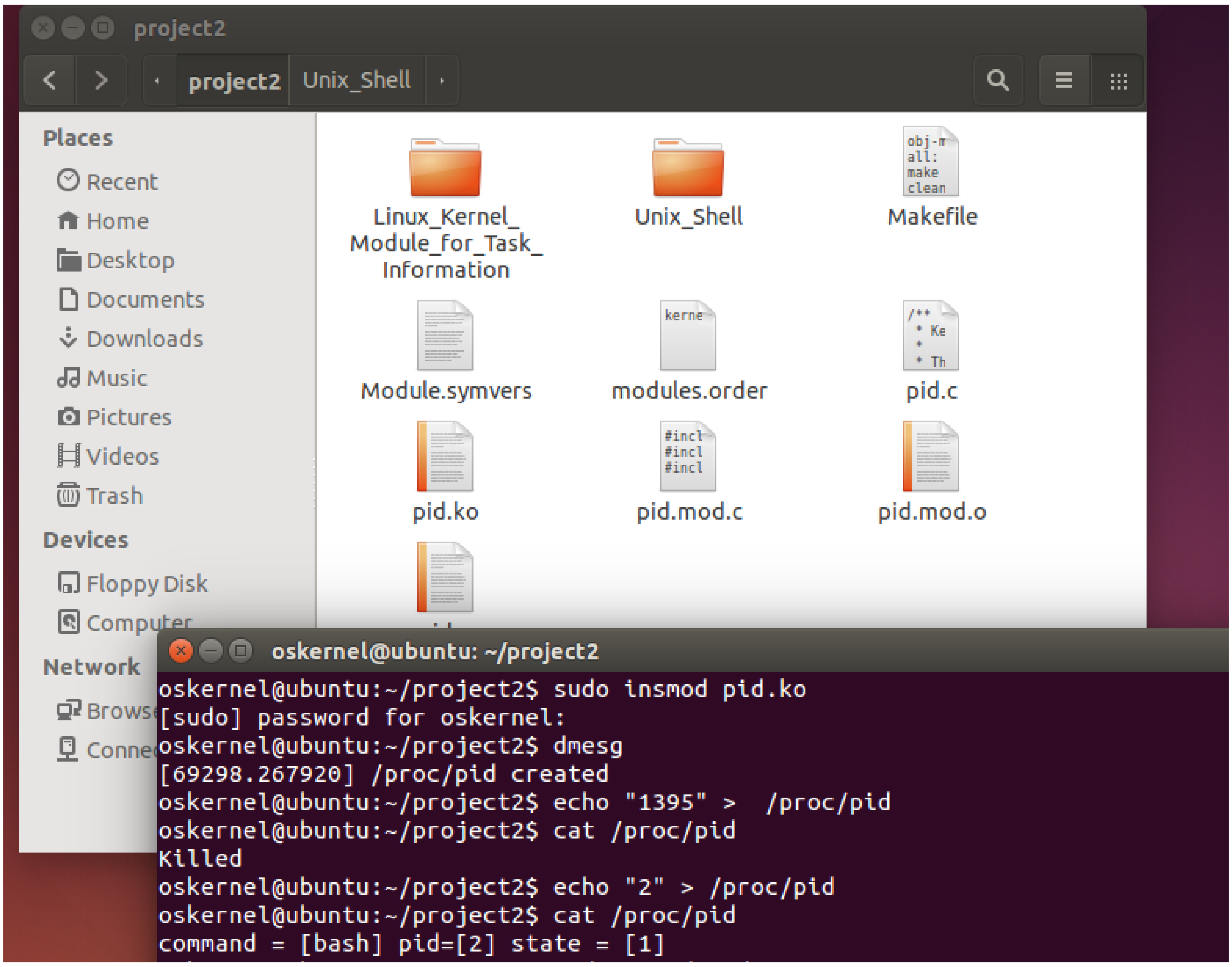
```
static long l_pid;
static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count,
loff_t *pos)
{
        int rv = 0;
        char *k_mem;
        // allocate kernel memory
        k_mem = kmalloc(count, GFP_KERNEL);
        /* copies user space usr_buf to kernel buffer */
        copy_from_user(k_mem, usr_buf, count);
        k_mem[count-1] = 0;

        sscanf(k_mem, "%ld", &l_pid);

        kfree(k_mem);
        return count;
}
```

2.Reading from the /proc File System

Just like we finish in the last project. We use the struct given in the text book. Then we print the information as the format. Done!

```
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];
        struct task_struct *tsk = NULL;
        tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);
        completed = 1;
        rv = sprintf(buffer,"command = [bash] pid=[%ld] state = [%ld]\n", l_pid, (tsk-
>state));
        copy_to_user(usr_buf, buffer, rv);
        // copies the contents of kernel buffer to userspace usr_buf
        return rv;
}
```

Result:



```
oskernel@ubuntu:~/project2$ sudo insmod pid.ko
[sudo] password for oskernel:
oskernel@ubuntu:~/project2$ dmesg
[69298.267920] /proc/pid created
oskernel@ubuntu:~/project2$ echo "1395" >  /proc/pid
oskernel@ubuntu:~/project2$ cat /proc/pid
Killed
oskernel@ubuntu:~/project2$ echo "2" > /proc/pid
oskernel@ubuntu:~/project2$ cat /proc/pid
command = [bash] pid=[2] state = [1]
```