EI338 Computer Systems Engineering

Project 2

November 29, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Exercise 1 UNIX Shell

In this exercise, we are asked to design a simple shell interface program that accepts the input commands and then executes them by assigning a new process. It is organized into several 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

Basic Structure

To start with, we first illustrate how the program is implemented as a whole. After some initializations, we obviously need to read the user input, and parse it so that it can be executed. And then, depending on the type of the command, we will execute it in a desired way. To be more detailed, the structure of **main()** looks like this:

```
int main(void)
{
    /* some initializations */
    /* ———some lines——— */

    /* present the prompt repeated */
    while (should_run) {
        printf("osh>");
        fflush(stdout);

        /* read the user input */
        fgets(cmd, MAX_LINE, stdin);
        cmd[strlen(cmd) - 1] = '\0'; /* delete line-feed */

        /**
         * After reading user input, the steps are:
         * (1) fork a child process
         * (2) the child process will invoke execvp()
         * (3) if command included &, parent will invoke wait()
         */
```

```
        /* deal with the history command as a special case */
        if (!strcmp(cmd, "!!")) {
            /* ----some lines---- */
        }
        else strcpy(cmd_history, cmd);

        /* parse the command */
        wait_flag = parse_cmd(cmd, args, ifile, ofile, &split_pos);

        /* execute the command by creating a child process */
        exe(args, wait_flag, ifile, ofile, split_pos);

        /* clear values */
        for (int i = 0; i < MAX_LINE/2 + 1; ++i)
        {
            free(args[i]);
            args[i] = NULL;
        }
        ifile[0] = '\0';
        ofile[0] = '\0';
    }

    return 0;
}
```

Listing 1: main()

I. Executing Command in a Child Process

First, it is necessary for us to parse what the user has entered into separate tokens and store the tokens in an array of character strings. It is quite easy for us to parse the command by using **strtok(char *str, const char *delim)**. The function returns the first substring parsed by **delim**.

And also, we tackle redirection, concurrent, and pipe here separately:

- If redirection occurs, **ifile** and **ofile** will be set.
- If concurrent occurs, the function **parse_cmd** will reture the flag to indicate whether the parent process needs to wait.
- If pipe occurs, **split_pos** will be assigned a nonzero value.

```
int parse_cmd(char *cmd, char **args, char *ifile, char *ofile, int *split_pos) {
    char *token;
    int i = 0;
    char redirect;

    token = strtok(cmd, " "); /* take the first token */

    while (token) {
        if (!strcmp(token, "<") || !strcmp(token, ">")) { /* redirect */
            redirect = token[0];
        }
        else {
            switch (redirect) {
                case '<':
```

```
                    strcpy(ifile, token);
                    break;

                case '>':
                    strcpy(ofile, token);
                    break;

                default:
                    args[i] = (char*) malloc(sizeof(char) * (strlen(token)));
                    strcpy(args[i], token);

                    if (!strcmp(token, "|")) *split_pos = i;

                    ++i;
            }
        }

        token = strtok(NULL, " "); /* take the next token */
    }

    if (!strcmp(args[i - 1], "&"))
    {
        args[i - 1] = NULL; /* clear '$' */

        return 0; /* wait flag is set to 0 (concurrent) */
    }

    return 1; /* wait flag is set to 1 */
}
```

Listing 2: parse_cmd()

Once we obtain the parsed command **args** as well as other optional values, we can call the function **exe()** execute it. The basic structure of it is as follows:

```
int exe(char **args, int wait_flag, char *ifile, char *ofile, int split_pos) {
    /* some initializations */
    /* ———some lines——— */

    pid = fork(); /* create child process */

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed.\n");
        return 1;
    }
    else if (pid == 0) { /* child process */
        if (strlen(ifile)) { /* redirect to input file */
            fd = open(ifile, O_RDWR);

            if (fd < 0) {
                printf("Error occurs when opening %s.\n", ifile);
                exit(0);
            }

            dup2(fd, STDIN_FILENO);
        }
```

```
        else if (strlen(ofile)) /* redirect to output file */
        {
            fd = open(ofile, O_RDWR|O_CREAT, S_IRWXU);

            if (fd < 0) {
                printf("Error occurs when opening %s\n", ofile);
                exit(0);
            }

            dup2(fd, STDOUT_FILENO);
        }

        if (split_pos == 0) { /* no pipe */
            /* execute the command in the child process */
            exe_err = execvp(args[0], args);

            if (exe_err < 0) printf("The command is not executable.\n");
        }
        else {
            /* this part will be discussed in next exercise */
        }

    }
    else { /* parent process */
        if (wait_flag) waitpid(pid, NULL, 0); /* the parent process waits */
    }
}
```

Listing 3: exe()

Every time we need to execute a command, we use **fork()** to create a child process and execute the command in the child process by using **execvp()**. For the parent process, if the flag **wait_flag** is set, it calls **waitpid()** to wait until the child process exits. Notice that we clearify which process needs to be waited by the parent process to avoid it to wait in vain.

II. Creating a History Feature

Here we apply a very simple way to realize the history function. We maintain a value called **cmd_history** and when we receive the user input, we check whether it is "!!" and **cmd_history** is empty. Everything else is the same as before.

```
/* deal with the history command as a special case */
if (!strcmp(cmd, "!!")) {
    if (strlen(cmd_history) == 0)
    {
        printf("-----------------------\n");
        printf("No commands in history.\n");
        printf("-----------------------\n");
    }
    else {
        printf("--------------------------------------------------\n");
        printf("Command '%s' in the history will be executed.\n", cmd_history);
        printf("--------------------------------------------------\n");

        strcpy(cmd, cmd_history);
    }
```

```
}
else strcpy(cmd_history, cmd);
```

Listing 4: history

III. Redirecting Input and Output

If the command contains "<" or ">", redirection occurs. By checking the length of strings **ifile** and **ofile**, we can tell whether redirection happens.

And when it happens, we can use **dup2()** to bound the specific file and STDIN/STDOUT. The code is present in **??**.

IV. Communication via a Pipe

This is the most difficult part of the project, and various kinds of bug occur during the implementation.

First, as usual, we check the denotion of pipe "|" and store the value of the position to split in **split_pos**. And in **exe()**, we check whether **split_pos** is 0 to determine the neccessity to extablish a pipe.

Since in the exercise, the former command will serve as the input of the latter command, we consider a reversed order, having the latter command executed in the parent process, and wait for the former command's result from the parent process.

To create a pipe, we use **pip()**. And then, a child process is also announced by using **fork()**.

```
char *former_args[MAX_LINE/2 + 1];
char *latter_args[MAX_LINE/2 + 1];
pid_t pipe_pid;

/* get the commands */
if (get_former_args(args, split_pos, former_args) != 0
    || get_latter_args(args, split_pos, latter_args) != 0) {
    printf("Error occurs when separating the command.\n");
    return 1;
}

/* create the pipe */
if (pipe(pipe_fd) == -1) {
    fprintf(stderr, "Pipe failed");
    return 1;
}

/* fork a child process */
pipe_pid = fork();

if (pipe_pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
```

Listing 5: pipe()-I

In the child process, we redirect the child's standard output to the write handle of the pipe. And similarly, the parent's standart input is redirected to the read handle of the pipe. In this way, two processes are able to communicate.

```
else if(pipe_pid == 0) { /* child process */
    /* close the unused end of the pipe */
    close(pipe_fd[READ_END]);
```

```
    dup2(pipe_fd[WRITE_END], STDOUT_FILENO);

    /* execute the command in the child process */
    exe_err = execvp(former_args[0], former_args);

    if (exe_err < 0) printf("The former command is not executable.\n");

    /* close the write end of the pipe */
    close(pipe_fd[WRITE_END]);
    }
}

else { /* parent process */
    waitpid(pid, NULL, 0);

    /* close the unused end of the pipe */
    close(pipe_fd[WRITE_END]);

    dup2(pipe_fd[READ_END], STDIN_FILENO);

    print_args(latter_args);

    exe_err = execvp(latter_args[0], latter_args);

    if (exe_err < 0) printf("The latter command is not executable.\n");

    close(pipe_fd[READ_END]);
}
```

Listing 6: pipe()-II

Both the parent process and the child process initially close their unused ends of the pipe. It is an important step to ensure that a process reading from the pipe can detect end-of-file when the writer has closed its end of the pipe.

Result

Several tests[**??**, **??**, **??**] demonstrate the correctness.

Figure 1: Test 1

Figure 2: Test2

Figure 3: Test 3

Exercise 2 Linux Kernel Module for Task

Similar to Project 1, we again deal with the */proc* file system.

Since the exercise itself is not difficult, we here will not present the basic structure as a whole, but go into these two parts directly.

I. Writing to the */proc* File System

The part is mainly about **proc_write()**. This function is called each time we write to the */proc/pid*. It first allocates some kernel memeroy to receive the input, and then convert the string of "pid" into integer. This value will be saved in **current_pid**.

```
static ssize_t proc_write(struct file *file,
const char __user *usr_buf, size_t count, loff_t *pos)
{
    char *k_mem;
    long pid;

    /* allocates kernel memory */
    k_mem = kmalloc(count + 1, GFP_KERNEL);

    /* copies user space usr_buf to kernel buffer */
    if (copy_from_user(k_mem, usr_buf, count)) {
        printk(KERN_INFO "Error occurs when copying from user\n");
        return -1;
    }

    k_mem[count] = '\0';

    /* obtains the long integer */
    kstrtol(k_mem, 10, &pid);
    current_pid = (int) pid;

    /* return kernel memory */
    kfree(k_mem);

    return count;
}
```

Listing 7: proc_read()

II. Reading from the */proc* File System

To return the name of the command, its process identifier, and its state, we invoke the function **find_vpid()** to obtain the struct pid, and then the function **pid_task()** returns the associated task_struct given the process identifier. The rest remains the same as that in Project 1.

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

    if (*pos > 0 || count < BUFFER_SIZE) {
        return 0;
    }

    tsk = pid_task(find_vpid(current_pid), PIDTYPE_PID);

    if (tsk) {
        len += sprintf(buffer, "command = [%s], pid = [%d], state = [%ld]\n",
```

```
                tsk->comm, current_pid, tsk->state);
    }

    else {
        printk(KERN_INFO "Invalid PID %d written to /proc/pid\n", current_pid);
        return 0;
    }

    /* copies the contents of kernel buffer to userspace usr_buf */
    if (copy_to_user(usr_buf, buffer, len)) {
        printk(KERN_INFO "Error occurs when copying from user\n");
        return -1;
    }

    /* updates the position and returns the number of bytes we received */
    *pos = len;
    return len;
}
```

Listing 8: proc_write()

Result

A simple tast is performed as follows:

Figure 4: Test 4