Lab Report of Project 2
UNIX Shell Programming & Linux Kernel Module for Task Information

Name: Li Dongyue Student Number: 516030910502
Computer System Engineering (Undergraduate), Fall 2018

November 4, 2018

1 UNIX Shell

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. The implementation are supposed to support **input and output redirection**, as well as **pipes** as a form of IPC between a pair of commands. Completing this project involves using the UNIX `fork(), exec(), wait(), dup2(),` and `pipe()` system calls.

1.1 The Overall Structure of My Shell

A C program that provides the general operations of a command-line shell are basically doing the following steps repeatedly:

- Take input from the user and separate a input line into commands and parameters.
- Fork a child process to execute the input commands and the main process waits for the child process to complete execution.
- The main process continues running if the return status of execution of child process is equal to one, otherwise stop reading input and exit.

The basic structure of simple shell is shown below.

```c
/* Read input from stdin */
char *lsh_read_line(void)
{
    cur_bufsize = LSH_RL_BUFSIZE;
    int position = 0;
    char *buffer = malloc(sizeof(char) * cur_bufsize);
    int c;

    if(!buffer){
        fprintf(stderr, "lsh: allocation error\n");
        exit(EXIT_FAILURE);
    }

    while(1){
        /* Read a character */
        c = getchar();

        if(c == EOF || c == '\n'){
            buffer[position] = '\0';
            return buffer;
        }else{
            buffer[position] = c;
        }
        position++;

        /* If buffer exceeded, reallocate buffer */
        if(position >= cur_bufsize){
            cur_bufsize += LSH_RL_BUFSIZE;
            buffer = realloc(buffer, cur_bufsize);
            if(!buffer){
                fprintf(stderr, "lsh: allocation error\n");
                exit(EXIT_FAILURE);
            }
```

```
        }
    }
}

/* Loop for getting input and executing it */
void lsh_loop(void)
{
    char *line;
    int status;

    do {
        printf("osh>");
        line = lsh_read_line();
        status = lsh_execute(line);

        free(line);
    } while(status);
}

int main(void)
{
    lsh_loop();

    return EXIT_SUCCESS;
}
```

shell.c

1.2 History Feature

To generate a history feature which allows a user to execute the most recent command by entering `!!`, I use a global variable to store the history commands, and when the input commands are detected as `!!`, the shell are going to use the history commands to execute:

```
    /* Histroy Run */
    else if(strcmp(args[0], "history") == 0 ||
            strcmp(args[0], "!!") == 0 || args[0][0] == '!'){
        return lsh_history(args);
    }
```

shell.c

The history feature part of the code is shown below:

```
/* History of commands */
int lsh_history(char **args)
{
    if (cur_pos == -1 || history[cur_pos] == NULL){
        fprintf(stderr, "No commands in history\n");
        exit(EXIT_FAILURE);
    }

    if (strcmp(args[0], "history") == 0){ /* Print out history commands */
        int last_pos = 0, position = cur_pos, count = 0;

        if(cur_pos != LSH_HIST_SIZE && history[cur_pos + 1] != NULL){
            last_pos = cur_pos + 1;
        }

        count = (cur_pos - last_pos + LSH_HIST_SIZE) % LSH_HIST_SIZE + 1;

        while(count > 0){
            char *command = history[position];
            printf("%d %s\n", count, command);
            position = position - 1;
            position = (position + LSH_HIST_SIZE) % LSH_HIST_SIZE;
            count --;
        }
    }
    else{
        char **cmd_args;
        char *command;
        if(strcmp(args[0], "!!") == 0){ /* Run History Command */
            command = malloc(sizeof(history[cur_pos]));
            strcat(command, history[cur_pos]);
```

```
            return lsh_execute(command);
        }
        else if(args[0][0] == '!'){
            if(args[0][1] == '\0'){
                fprintf(stderr, "Expected arguments for \"!\"\n");
                exit(EXIT_FAILURE);
            }
            /* position of the command to execute */
            int offset = args[0][1] - '0';

            int next_pos = (cur_pos + 1) % LSH_HIST_SIZE;
            if(next_pos != 0 && history[cur_pos + 1] != NULL){
                offset = (cur_pos + offset) % LSH_HIST_SIZE;
            }else{
                offset--;
            }

            if(history[offset] == NULL){
                fprintf(stderr, "No such command in history\n");
                exit(EXIT_FAILURE);
            }

            command = malloc(sizeof(history[cur_pos]));
            strcat(command, history[offset]);
            return lsh_execute(command);
        }else{
            perror("lsh");
        }
    }
}
```

shell.c

1.3 Input and Output Redirection

Basically we use `dup2()` to redirect the input and output. When the shell detects the commands are using input or output redirection:

```
    /* Redirect output */
    int j=0;
    while (true)
    {
        if (strcmp(args[j], ">") == 0)
            return lsh_redirect_output(args);
        j++;
        if (args[j] == NULL) break;
    }

    /* Redirect input */
    j=0;
    while (true)
    {
        if (strcmp(args[j], "<") == 0)
        {
            return lsh_redirect_input(args);
        }
        j++;
        if (args[j] == NULL) break;
    }
```

shell.c

The shell does things followed:

- Separate out the commands and file name by the redirect operands > or <
- Open the file and redirect the input or output to that file
- Execute the commands normally

The example code of **Output Redirection** is shown below:

```
    if(pid == 0){ /* child process */
        char ** run_arg = malloc(LSH_TOK_BUFSIZE * sizeof(char*));
        int j=0;
```

```
        while (true)
        {
            if (strcmp(args[j], ">")==0) break;
            run_arg[j] = args[j];
            j++;
        }
        char * output = args[j+1];

        int out = open(output, O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR);
        dup2(out, 1);
        close(out);
        if(execvp(run_arg[0], run_arg) == -1) perror("lsh");
        exit(EXIT_FAILURE);
    }
```

shell.c

1.4 Communication via a Pipe

The communication via a pipe is similar to the input and output redirection. When a pipe communication command is detected:

```
    /* Pipe */
    j=0;
    while (true)
    {
        if (strcmp(args[j], "|") == 0)
        {
            return lsh_pipe(args);
        }
        j++;
        if (args[j] == NULL) break;
    }
```

shell.c

The shell does:

- Initialize a pipe in the child process
- Separate out two commands
- Put each commands to each side of the pipe
- Execute them in different process

The code is shown below:

```
int lsh_pipe(char **args)
{
    pid_t pid, wpid;
    int status;

    pid = fork();
    if(pid == 0){ /* child process */
        char **pipeHeadArgs = malloc(LSH_TOK_BUFSIZE * sizeof(char*));
        char **pipeTailArgs = malloc(LSH_TOK_BUFSIZE * sizeof(char*));
        int j=0; int tmpNum;
        while (true)
        {
            if (strcmp(args[j],"|")==0) {tmpNum = ++j;break;}
            pipeHeadArgs[j] = args[j];
            j++;
        }
        pipeHeadArgs[j] = NULL;
        while (true)
        {
            if (args[j] == NULL) break;
            pipeTailArgs[j-tmpNum] = args[j];
            j++;
        }
        pipeTailArgs[j-tmpNum+1] = NULL;

        int pipefd[2];
        int childPid;
```

```
        pipe(pipefd);
        childPid = fork();

        if (childPid == 0)
        {   // replace standard input with input part of pipe

            dup2(pipefd[0], 0);

            // close unused hald of pipe

            close(pipefd[1]);

            execvp(pipeTailArgs[0], pipeTailArgs);
        }
        else
            {
            // parent gets here and handles "cat scores"

            // replace standard output with output part of pipe

            dup2(pipefd[1], 1);

            // close unused unput half of pipe

            close(pipefd[0]);

            execvp(pipeHeadArgs[0], pipeHeadArgs);
        }
    }
    else if(pid > 0){ /* parent process */
        if(!conc){
            do{
                wpid =  waitpid(pid, &status, WUNTRACED);
            }while(!WIFEXITED(status) && !WIFSIGNALED(status));
        }
    }else{ /* error forking */
        perror("lsh");
    }

    conc = false;
    return 1;
}
```

shell.c

1.5 Other Implementation

We also provide other implementations such as `cd` command, `help` command, and `exit`. Please check this out in the code attached.

2 Task Information

In this project, it is required to write a Linux kernel module that uses the `/proc/file` system for displaying a task's information based on its process identifier value `pid`. Before beginning this project, be sure you have completed the Linux kernel module programming project in Chapter 2, which involves creating an entry in the `/proc/file` system. This project will involve writing a process identifier to the `/proc/pid`. Once a `pid` has been written to the `/proc/file`, subsequent reads from `/proc/pid` will report:

- the command the task is running
- the value of the task's `pid`
- the current state of the task

2.1 Writing to the /proc File System

First, I set the field `.write` in struct file operations to cause the `proc_write()` function called when taking an input from user. Second, I use a global `long int` variable to store the input `pid`. The code is shown below:

```c
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 (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;
}
```

pid.c

2.2 Reading from the `/proc` File System

Once the process identifier has been stored, any reads from `/proc/pid` will return the name of the command, its process identifier, and its state. The PCB in Linux is represented by the structure task struct, which is found in the `<linux/sched.h>` include file. Given a process identifier, the function `pid_task()` returns the associated task struct. Then we can then display the values of the command, pid, and state. The code is shown below:

```c
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);
        if (tsk==NULL) rv = sprintf(buffer, "command = [no such task] pid = [%ld] state=[no such task]",
    l_pid);
  else rv = sprintf(buffer, "command = [%s] pid = [%ld] state=[%ld]", tsk->comm, l_pid, tsk->state);
        completed = 1;

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

        return rv;
}
```

pid.c

Whole code is attached to this file.