Unix Shell Programming

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. 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.

Overview

1. Take input from the user and parse it into separate tokens.
2. Fork a child process to execute commands.
3. Provide a history feature to allow a user to execute the recent command by entering command again by `!!` and `!{num}`+at the prompt.
4. Support `<` and `>` to redirect the input and output respectively.
5. allow the output of one command to serve as input to another using a pipe.

Main Function

```
int main()
{
    char *command;
    int status;

    do {
        command = read_command();
        status = execute(command);
        free(command);

    }while(status);
    return 0;
}
```

Read Command

```
char* read_command()
{
    char *buffer = malloc(sizeof(char) * MAX_LINE);

    do {
        printf("osh>");
        fflush(stdout);
        read(STDIN_FILENO, buffer,MAX_LINE);
    }while (buffer[0] == '\n');

    return buffer;
}
```

The decision condition in while is arranged to swallow newline characters.

Split Command

```
char **command_split(char * cmd)
{
    char **args = malloc(MAX_ARG * sizeof(char*));
    if(!args){
        fprintf(stderr, "allocation error\n");
        exit(-2);
    }

    char *arg = strtok(cmd, ARG_DELIM);
    int cnt = 0;
    while(arg)
    {
        args[cnt++] = arg;
        arg = strtok(NULL, ARG_DELIM);
    }

    if(strcmp(args[cnt-1], "&") == 0)
    {
        bg = true;
        args[cnt - 1] = NULL;
    }
    args[cnt] = NULL;
    return args;
}
```

`command_split` is designed to split the command into arguments list.

Execute

```
int execute(char *cmd)
{
    char* cmd_tmp = malloc(MAX_LINE * sizeof(char));
    strcpy(cmd_tmp, cmd);

    char** args = command_split(cmd);
    //char **args = lsh_split_line(cmd);
    //printf(args[0]);
    //printf(args[1]);
    if(strncmp(cmd, "history", 7) == 0 || strncmp(cmd,
"!!", 2) == 0 || strncmp(cmd, "!", 1) == 0)
        return history(args);
    else if(strncmp(cmd, "exit", 4) == 0)
        return 0;

    strcpy(histories[command_count %
MAX_HISTORY],cmd_tmp);
    free(cmd_tmp);
    command_count++;

    int i = 0;


    while (true)
    {
        if (strcmp(args[i], ">") == 0)
            return redirect_output(args);
        else if (strcmp(args[i], "<") == 0)
            return redirect_input(args);
        else if (strcmp(args[i], "|") == 0)
            return pipe_(args);
        i++;
        if (args[i] == NULL) break;
    }

    return launch(args);

}


int launch(char **args)
{
    pid_t pid;

    pid = fork();
    if(pid == 0){ /* child process */
```

```
        if(execvp(args[0], args) == -1) perror("error");
        exit(-2);
    }
    else if(pid > 0){ /* parent process */
        if (bg == 0) /* handle parent,wait for child */
            while (pid != wait(NULL)) ;
    }else{ /* error forking */
        perror("error");
    }

    bg = false;
    return 1;
```

`execute` divides the command into several categories. `launch` is to fork a process to execute the command.

History Feature

```
int history(char **args)
{
    if(command_count == 0){
        fprintf(stderr, "No commands in history\n");
        exit(-1);
    }
    if(strcmp(args[0], "history") == 0){

        if(command_count <= 5){
            int cnt = 1;
            for(int i = command_count - 1; i >= 0; i--)
            {
                printf("%d %s\n", cnt++, histories[i]);
            }
        }
        else{
            int cnt = command_count - 1;
            for(int i = 0; i < 5; i++)
            {
                int idx = cnt % MAX_HISTORY;
                printf("%d %s\n", i+1, histories[idx]);
                cnt--;
            }
        }
        return 1;

    }
    else
```

```
    {
        char *cmd;
        if(strcmp(args[0],"!!") == 0) {
            return execute(histories[(command_count-1) %
MAX_HISTORY]);
        }
        else if(args[0][0] == '!'){
            if(args[0][1] == '\0'){
                fprintf(stderr, "Expected arguments for
\"!\"\n");
                exit(-1);
            }
            else{
                int pos = args[0][1] - '0';
                return execute(histories[(command_count-
pos) % MAX_HISTORY]);
            }
        }

    }

    return 1;

}
```

`histories` is a global array to store history records of commands. and function `history` is designed specially to deal with the case when the command is in the category of history.

Input & Output Redirection

```
int redirect_output(char ** args)
{
    pid_t pid;

    pid = fork();
    if(pid == 0){ /* child process */
        char ** run_arg = malloc(MAX_LINE *
sizeof(char*));
        int i=0;
        while (true)
        {
            if (strcmp(args[i], ">")==0) break;
            run_arg[i] = args[i];
            i++;
        }
        char * output = args[i+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");
        free(run_arg);
        exit(EXIT_FAILURE);

    }
    else if(pid > 0){ /* parent process */
        if (bg == 0) /* handle parent,wait for child */
            while (pid != wait(NULL)) ;
    }else{ /* error forking */
        perror("error");
    }

    bg = 0;
    return 1;
}
```

`redirect_output` or `redirect_input` first locates `<` or `>` and use `dup2` to redirect the input or output to the given file. Here we take `redirect_output` as an example.

Communication via a Pipe

```
int pipe_(char **args)
{

    pid_t pid;

    pid = fork();
    if(pid == 0){ /* child process */
        char ** pipe_source = malloc(MAX_LINE *
sizeof(char*));
        char ** pipe_des = malloc(MAX_LINE *
sizeof(char*));
        int i=0;

        while (true)
        {
            if (strcmp(args[i], "|")==0) break;
            pipe_source[i] = args[i];
            i++;
        }
```

```
        pipe_source[i] = NULL;
        int tmp = ++i;
        while (true)
        {
            if (args[i] == NULL) break;
            pipe_des[i-tmp] = args[i];
            i++;
        }
        pipe_des[i-tmp+1] = NULL;



        int pipefd[2];
        pid_t child;
        pipe(pipefd);
        child = fork();
        if(child == 0)
        {
            dup2(pipefd[0], 0);
            close(pipefd[1]);
            execvp(pipe_des[0], pipe_des);
        }
        else
        {
            dup2(pipefd[1], 1);

            close(pipefd[0]);

            execvp(pipe_source[0], pipe_source);
        }
    }
    else if(pid > 0){ /* parent process */
        if (bg == 0) /* handle parent,wait for child */
            while (pid != wait(NULL)) ;
    }else{ /* error forking */
        perror("error");
    }

    bg = 0;
    return 1;
}
```

`pipe_` first initialize a pipe in the child process and separate the command into two parts. Then put each commands to each side of the pipe and execute them in different processes.

Result

```
gavin@ubuntu:~/Documents/Project2/1$ ./test
osh>ls
demo.c  out.txt   shell.c         test   test.txt
ls.txt  sample.c  simple-shell.c  test1
osh>ls -l
total 64
-rwxrw-rw- 1 gavin gavin  6662 Oct 31 02:45 demo.c
-rw-r----- 1 gavin gavin    68 Oct 31 21:07 ls.txt
-rw-r--r-- 1 gavin gavin    74 Oct 31 21:58 out.txt
-rw-rw-r-- 1 gavin gavin 11816 Oct 31 02:45 sample.c
-rw-rw-r-- 1 gavin gavin     0 Oct 31 02:45 shell.c
-rw-r--r-- 1 gavin gavin  6950 Nov  2 23:45 simple-shell.c
-rwxr-xr-x 1 gavin gavin 21960 Oct 31 02:45 test
-rw-r--r-- 1 gavin gavin     0 Oct 31 21:57 test1
-rw-r--r-- 1 gavin gavin    74 Nov  3 00:11 test.txt
osh>ls > test.txt
osh>sort < test.txt
demo.c
ls.txt
out.txt
sample.c
shell.c
simple-shell.c
test
test1
test.txt
osh>ls | sort
demo.c
ls.txt
out.txt
sample.c
shell.c
simple-shell.c
test
test1
test.txt
osh>!!
demo.c
ls.txt
out.txt
sample.c
shell.c
simple-shell.c
test
test1
test.txt
osh>history
1 ls | sort

2 ls
3 sort < test.txt

4 ls > test.txt

5 ls -l
```

Task Information

Overview

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` system. This project will involve writing a process identifier to the `/proc/pid`. Once a `pid` has been written to the `/proc`, subsequent reads from `/proc/pid` will report:

Writing to `/proc` File System

```
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;
}
```

Here we use a global variable `l_pid` to store the process identifier. Every time we make a entry into `/proc/pid` and write something into it, it will be stored in `l_pid`.

Reading from `/proc` File System

```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;
}
```

Every time we read from `/pro/pid` we will get the name of the process, its pid, and its state which can be obtained by `pid_task`.

Result

```
gavin@ubuntu:~/Documents/Project2/2$ sudo insmod pid.ko
gavin@ubuntu:~/Documents/Project2/2$ echo "12" > /proc/pid
gavin@ubuntu:~/Documents/Project2/2$ cat /proc/pid
command = [idle_inject/0] pid = [12] state=[1]
```