Project 2-1: UNIX Shell Programming

白骐硕 518030910102

任务概述：

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 mac OS system.

具体实现

Overview

本任务所有的代码都是在实例代码的基础上不断增加和改进而完成的。

实例代码如下：

```
#include <stdio.h>
#include <unistd.h>

#define MAX LINE 80 /* The maximum length command */
int main(void)
{
    char *args[MAX LINE/2 + 1]; /* command line arguments */
    int should run = 1; /* flag to determine when to exit program */
    while (should run) {
        printf("osh>");
        fflush(stdout);

        /**
        * After reading user input, the steps are:
        * (1) fork a child process using fork()
        * (2) the child process will invoke execvp()
        * (3) parent will invoke wait() unless command included &
        */
        return 0;
    }
}
```

综合考虑后续所给出的几项具体要求，我们实现的思路如下：

1. 读入用户输入的指令，如果为exit，则中止；如果输入不合法（如空），则重新输入；如果成功输入，则往下执行
2. 解析指令，获得参数数组，如果指令为"!!"，则将指令替换为前一指令（前一指令不存在则报错），若为常规指令则往下执行
3. 创建子进程执行指令：

    - 先判断是否有pipe，|，如果有单独处理。如果没有，则往下执行
    - 判断是否需要redirecting I/O，如果有处理相关文件操作，并执行指令，如果没有，则往下执行
    - 则该指令为普通指令，直接调用 `execvp(args[0], args)` 执行。
4. 父进程根据指令最后是否有 & （判断过程在执行指令之前）来决定是否执行 `wait(NULL)`。

具体的实现细节见下文。

Executing Command in a Child Process

首先，我们需要从stdin中，读入用户输入的指令。通过 fgets()函数实现，将读入的命令存入 command 字符串中。

```
int get_input(char *command) {
    char input_buffer[MAX_LINE + 1];
    if(fgets(input_buffer, MAX_LINE + 1, stdin) == NULL) {
        fprintf(stderr, "Failed to read input!\n");
        return 0;
    }
    strcpy(command, input_buffer);  // update the command
    return 1;
}
```

之后，我们需要对读入的 command 进行解析，拆分并生成参数的字符串数组。

```
size_t parse_input(char *args[], char *original_command) {
    size_t num = 0;
    char command[MAX_LINE + 1];
    strcpy(command, original_command);  // make a copy since `strtok` will
modify it
    char *token = strtok(command, DELIMITERS);
    while(token != NULL) {
        args[num] = malloc(strlen(token) + 1);
        strcpy(args[num], token);
        ++num;
        token = strtok(NULL, DELIMITERS);
    }
    return num; // return the number of tokens
}
```

对指令进行解析后，我们便可以进入指令的执行阶段。在这一步中，我们编写了一个最基础的 run_command() 函数，后续所有的改进均在此基础上展开。

其中 `has_ampersand()` 函数是用于判断指令最后是否含有 &，如果有则将其移除并且并发执行。

```
int run_command(char **args, size_t args_num){
    int run_concurrently = has_ampersand(agrs, &args_num);

    pid_t pid = fork();
    if(pid<0){
        fprintf(stderr, "Fail to fork!\n");
        return 0;
    } else if(pid == 0){//child process
        execvp(args[0], args);
```

```
    }else{// parent process
        if(!run_concurrently){
            wait(NULL);
        }
    }
}
```

Creating a History Feature

为了支持这一特殊指令，我们直接在读入用户输入时对 !! 输入进行特殊处理，修改后的 get_input() 函数如下：

```
int get_input(char *command) {
    char input_buffer[MAX_LINE + 1];
    if(fgets(input_buffer, MAX_LINE + 1, stdin) == NULL) {
        fprintf(stderr, "Failed to read input!\n");
        return 0;
    }
    if(strncmp(input_buffer, "!!", 2) == 0) {
        if(strlen(command) == 0) {  // no history yet
            fprintf(stderr, "No commands in history.\n");
            return 0;
        }
        printf("%s", command);    // keep the command unchanged and print it
        return 1;
    }
    strcpy(command, input_buffer);  // update the command
    return 1;
}
```

在其中加入了对于输入等于 !! 的单独处理：判断 command 是否为空，如果为空，则报错 "No history available yet!"。如果不为空，则 command中存的字符串即为上个命令的指令，无需对其进行修改，后续将会对其再次执行。

Redirecting Input and Output

首先需要判断指令中是否含有 < 和 > 这两个字符，如果含有，则记录下文件的名字，并且将该符号和文件名称从args字符串数组中移除。

```
int check_redirection(char **args, size_t *size, char **input_file, char
**output_file) {
    int flag = 0;
    size_t to_remove = -1;
    for(size_t i = 0; i != *size; ++i) {
        if(strcmp("<", args[i]) == 0) {     // input
            to_remove = i;
            if(i == (*size) - 1) {
                fprintf(stderr, "No input file provided!\n");
                break;
            }
            flag = 1;
            *input_file = args[i + 1];
            break;
```

```
        } else if(strcmp(">", args[i]) == 0) {   // output
            to_remove = i;
            if(i == (*size) - 1) {
                fprintf(stderr, "No output file provided!\n");
                break;
            }
            flag = 2;
            *output_file = args[i + 1];
            break;
        }
    }
    if(flag == 0) return flag;
    /* Remove I/O indicators and filenames from arguments */
    size_t pos = to_remove;  // the index of arg to remove
    while(pos != *size) {
        args[pos] = args[pos + 2];
        ++pos;
    }
    (*size) -= 2;

    return flag;
}
```

之后根据 I/O 的类型，进行相应的文件操作（为了简化代码，即使指令无需进行redirecting, 也执行该函数，只是不作任何操作，这样减少了一次对于redirecting的判断）。

```
int redirect_io(int io_flag, char *input_file, char *output_file, int
*input_desc, int *output_desc) {
    if(io_flag == 2) {  // redirecting output
        *output_desc = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if(*output_desc < 0) {
            fprintf(stderr, "Failed to open the output file: %s\n",
output_file);
            return 0;
        }
        dup2(*output_desc, STDOUT_FILENO);
    }
    if(io_flag == 1) { // redirecting input
        *input_desc = open(input_file, O_RDONLY);
        if(*input_desc < 0) {
            fprintf(stderr, "Failed to open the input file: %s\n", input_file);
            return 0;
        }
        dup2(*input_desc, STDIN_FILENO);
    }
    return 1;
}
```

执行指令的过程如下（下述代码为到 `run_command()` 函数的改进）：

```
char *input_file, *output_file;
int input_desc, output_desc;
int io_flag = check_redirection(args, &args_num, &input_file, &output_file);
if(redirect_io(io_flag, input_file, output_file, &input_desc, &output_desc) ==
0) {
    return 0;
}
execvp(args[0], args);
close_file(io_flag, input_desc, output_desc); //close the file
fflush(stdin);
```

Communication via a Pipe

首先判断指令中是否含有 |， 如果含有则将指令分为两部分，分别执行。执行过程为：原先的子进程 P1 再创建一个新的子进程 P2，则P2执行指令的前半部分，P1等待（`wait(nuLL)`）P2执行完毕之后，执行指令的后半部分。P1执行的结果通过 pipe 传给 P2。

具体实现代码如下：

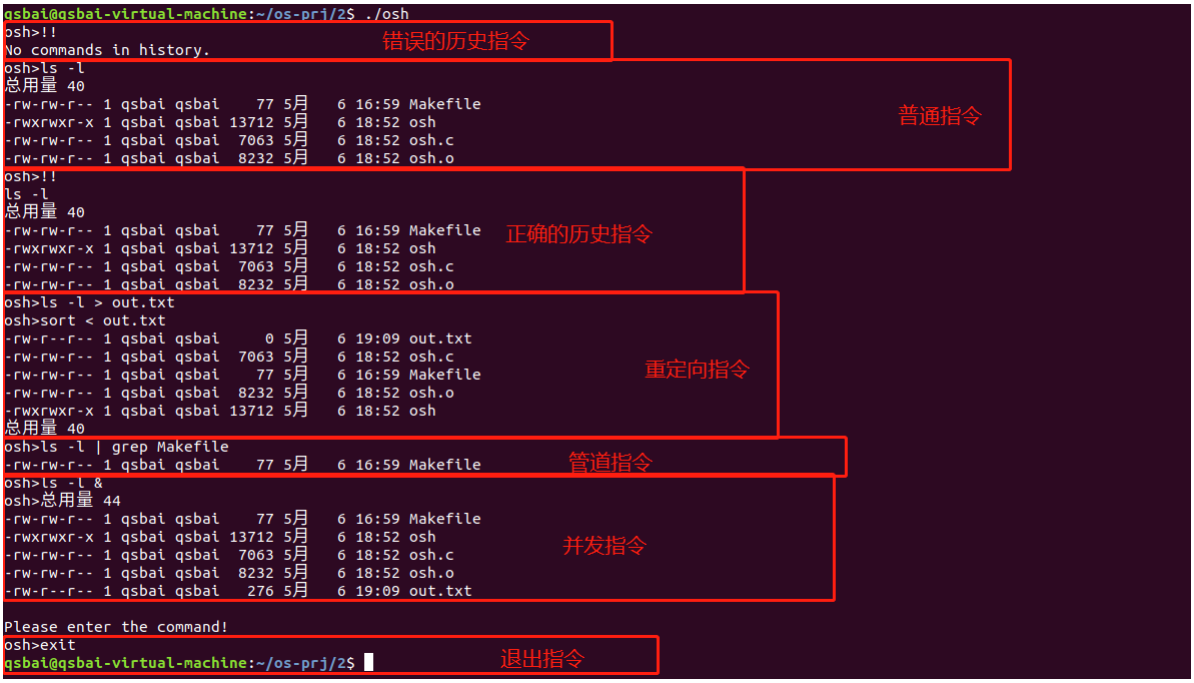
```
/* Create pipe */
int fd[2];
pipe(fd);
/* Fork into another two processes */
pid_t pid2 = fork();
if(pid2 > 0) {  // child process for the second command
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    wait(NULL);     // wait for the first command to finish
    execvp(args2[0], args2);
    close(fd[0]);
    fflush(stdin);
} else if(pid2 == 0) {  // grandchild process for the first command
    close(fd[0]);
    dup2(fd[1], STDOUT_FILENO);
    execvp(args[0], args);
    close(fd[1]);
    fflush(stdin);
}
```

至此，所有的功能需求已经全部实现，完整的 `run_command()` 函数（即指令执行过程）如下：

```
int run_command(char **args, size_t args_num) {
    /* Detect '&' to determine whether to run concurrently */
    int run_concurrently = has_ampersand(args, &args_num);
    /* Detect pipe */
    char **args2;
    size_t args_num2 = 0;
    detect_pipe(args, &args_num, &args2, &args_num2);
    /* Create a child process and execute the command */
    pid_t pid = fork();
    if(pid < 0) {   // fork failed
        fprintf(stderr, "Failed to fork!\n");
        return 0;
    } else if (pid == 0) { // child process
```

```
        if(args_num2 != 0) {    // pipe
            /* Create pipe */
            int fd[2];
            pipe(fd);
            /* Fork into another two processes */
            pid_t pid2 = fork();
            if(pid2 > 0) {  // child process for the second command
                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                wait(NULL);     // wait for the first command to finish
                execvp(args2[0], args2);
                close(fd[0]);
                fflush(stdin);
            } else if(pid2 == 0) {  // grandchild process for the first command
                close(fd[0]);
                dup2(fd[1], STDOUT_FILENO);
                execvp(args[0], args);
                close(fd[1]);
                fflush(stdin);
            }
        } else {    // no pipe
            // Redirect I/O
            char *input_file, *output_file;
            int input_desc, output_desc;
            int io_flag = check_redirection(args, &args_num, &input_file,
&output_file);    // 2 for output, 1 for input
            if(redirect_io(io_flag, input_file, output_file, &input_desc,
&output_desc) == 0) {
                return 0;
            }
            execvp(args[0], args);
            close_file(io_flag, input_desc, output_desc);
            fflush(stdin);
        }
    } else { // parent process
        if(!run_concurrently) { // parent and child run concurrently
            wait(NULL);
        }
    }
    return 1;
}
```

实验结果


qsbai@qsbai-virtual-machine:~/os-prj/2$ ./osh
osh>!!
No commands in history.
错误的历史指令
osh>ls -l
总用量 40
-rw-rw-r-- 1 qsbai qsbai   77 5月   6 16:59 Makefile
-rwxrwxr-x 1 qsbai qsbai 13712 5月   6 18:52 osh
-rw-rw-r-- 1 qsbai qsbai 7063 5月   6 18:52 osh.c
-rw-rw-r-- 1 qsbai qsbai 8232 5月   6 18:52 osh.o
普通指令
osh>!!
ls -l
总用量 40
-rw-rw-r-- 1 qsbai qsbai   77 5月   6 16:59 Makefile
-rwxrwxr-x 1 qsbai qsbai 13712 5月   6 18:52 osh
-rw-rw-r-- 1 qsbai qsbai 7063 5月   6 18:52 osh.c
-rw-rw-r-- 1 qsbai qsbai 8232 5月   6 18:52 osh.o
正确的历史指令
osh>ls -l > out.txt
osh>sort < out.txt
-rw-r--r-- 1 qsbai qsbai   0 5月   6 19:09 out.txt
-rw-rw-r-- 1 qsbai qsbai 7063 5月   6 18:52 osh.c
-rw-rw-r-- 1 qsbai qsbai   77 5月   6 16:59 Makefile
-rw-rw-r-- 1 qsbai qsbai 8232 5月   6 18:52 osh.o
-rwxrwxr-x 1 qsbai qsbai 13712 5月   6 18:52 osh
总用量 40
重定向指令
osh>ls -l | grep Makefile
-rw-rw-r-- 1 qsbai qsbai   77 5月   6 16:59 Makefile
管道指令
osh>ls -l &
osh>总用量 44
-rw-rw-r-- 1 qsbai qsbai   77 5月   6 16:59 Makefile
-rwxrwxr-x 1 qsbai qsbai 13712 5月   6 18:52 osh
-rw-rw-r-- 1 qsbai qsbai 7063 5月   6 18:52 osh.c
-rw-rw-r-- 1 qsbai qsbai 8232 5月   6 18:52 osh.o
-rw-r--r-- 1 qsbai qsbai   276 5月   6 19:09 out.txt
并发指令
Please enter the command!
osh>exit
qsbai@qsbai-virtual-machine:~/os-prj/2$
退出指令