# Unix_shell 实验报告

20231164 张岳霖

myshell是一个可以在Linux操作系统下运行的C语言程序，能够为用户提供命令行交互窗口。

## 一、功能简介

1. 支持运行外部指令
2. 支持内部指令cd, history, exit
3. 支持I/O重定向
4. 支持管道连接的两个命令
5. 支持跳转运行历史指令

## 二、输入限制

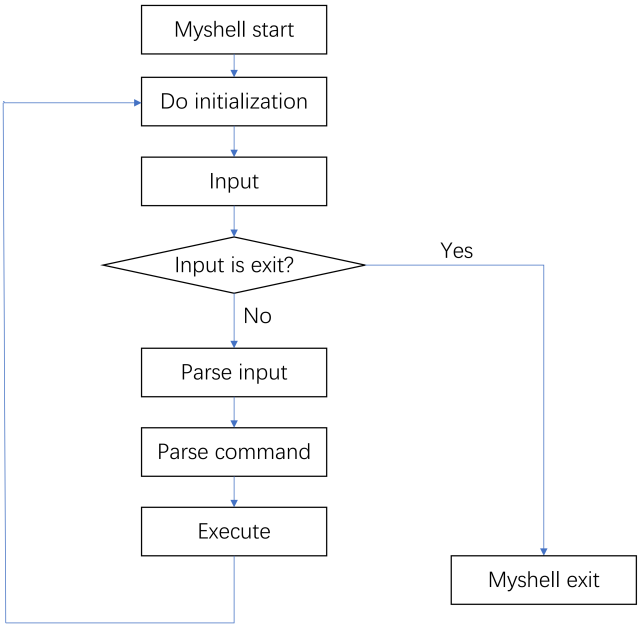
1. 指令最大长度限制：256
2. 历史指令最大查询范围：50
3. 历史指令快速跳转范围：9
4. 管道连接的指令最大数量：2

## 三、实现方法

1. 函数功能介绍

| 函数名称 | 函数功能 | 系统调用 |
| --- | --- | --- |
| int main() | 主函数，读取用户的输入，当输入exit时退出程序 | 无 |
| void init() | 初始化函数，在主函数读取用户输入前进行初始化，主要是将标志位和上一条指令的运行结果清零 | 无 |
| void parse_input() | 解析字符串，分离指令，确定指令类型(cd类/history类/重定向类/管道类) | 无 |
| void parse_command() | 将指令按照空格分割保存在cmd.argv中 | 无 |
| void execute() | 运行指令 | chdir(); getcwd(); dup2(); execvp(); waitpid(); pipe(); close(); exit(); fork(); |

2. 程序执行流程




在程序中，我定义了一个如下的command结构体，对输入的信息进行管理；并定义了下面的标志位，用于区分输入的指令类型。

```
struct command {
    char *argv[MAX_SIZE];           // 指令参数
    char *target_file;              // 重定向输入或输出到的文件
    char *cd_path;                  // cd指令跳转到的路径
    int history_num;                // history指令查询的条数
};

int redirect_flag;                  // 表示当前指令是重定向指令
int pipeline_flag;                  // 表示当前指令是管道指令
int history_flag;                   // 表示当前指令是查询历史指令
int cd_flag;                        // 表示当前指令是切换目录指令

#define REDIRECT_OUT_A 1            // 用于进一步区分重定向指令：输出重定向，附加 >>
#define REDIRECT_OUT_N 2            // 用于进一步区分重定向指令：输出重定向，新建 >
#define REDIRECT_IN    3            // 用于进一步区分重定向指令：输入重定向 <
```

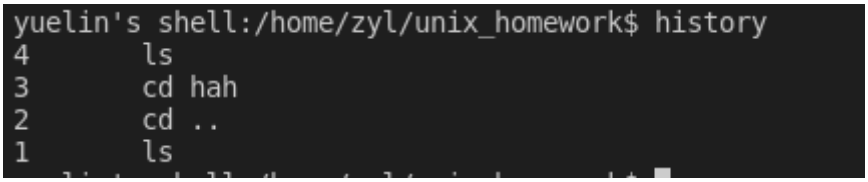
在执行完流程：初始化->读取输入->解析输入->解析指令后，以上指令及标志位信息被填写完毕，而后execute()函数将读取上述信息，进行指令的执行。

3. 各功能实现方法

    1. 查询历史指令

        输入格式：`history`

        执行示例：




    2. 跳转执行历史指令

        输入格式：`h\d`，如h1(h后的数字在1-9范围内)

执行示例：

```
yuelin's shell:/home/zyl/unix_homework$ history
5       cat hello | ./main.o
4       cat hello
3       cat hello | ./main.o
2       cat hello
1       cat < hello
yuelin's shell:/home/zyl/unix_homework$ h3
execute cat hello | ./main.o
hello world
a is 20021021
yuelin's shell:/home/zyl/unix_homework$ h1
execute cat < hello
20021021
```

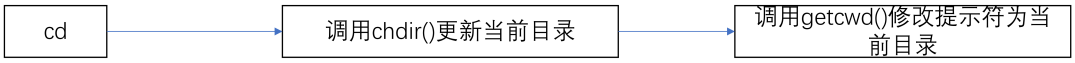
3. cd指令

输入格式：`cd`

执行示例：

```
yuelin's shell:/home/zyl/unix_homework$ cd hah
yuelin's shell:/home/zyl/unix_homework/hah$
```

实现流程图：




4. 重定向指令

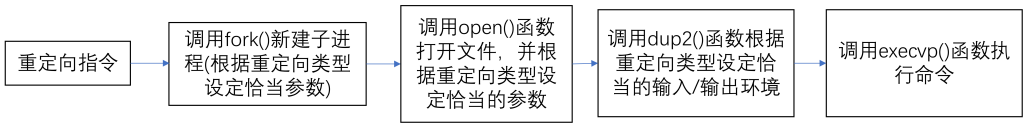
输入格式：`command > file  or  command >> file  or command < file`，重定向符号前后有无空格均可。

执行示例：

```
yuelin's shell:/home/zyl/unix_homework/hah$ ls -al > ls.txt
yuelin's shell:/home/zyl/unix_homework/hah$ cat ls.txt
总用量 32
drwxrwxr-x 2 zyl zyl  4096 5月    5 14:26 .
drwxrwxr-x 4 zyl zyl  4096 5月    5 14:25 ..
-rw-rw-r-- 1 zyl zyl     0 5月    5 14:26 ls.txt
-rw-rw-r-- 1 zyl zyl    71 4月   30 20:07 main.c
-rwxrwxr-x 1 zyl zyl 16696 4月   30 20:07 main.o
yuelin's shell:/home/zyl/unix_homework/hah$ echo append >> ls.tx
yuelin's shell:/home/zyl/unix_homework/hah$ cat ls.txt
总用量 32
drwxrwxr-x 2 zyl zyl  4096 5月    5 14:26 .
drwxrwxr-x 4 zyl zyl  4096 5月    5 14:25 ..
-rw-rw-r-- 1 zyl zyl     0 5月    5 14:26 ls.txt
-rw-rw-r-- 1 zyl zyl    71 4月   30 20:07 main.c
-rwxrwxr-x 1 zyl zyl 16696 4月   30 20:07 main.o
append
yuelin's shell:/home/zyl/unix_homework/hah$ echo new > ls.txt
yuelin's shell:/home/zyl/unix_homework/hah$ cat ls.txt
new
```

```
yuelin's shell:/home/zyl/unix_homework$ cat hello
20021021
yuelin's shell:/home/zyl/unix_homework$ cat < hello
20021021
```

实现流程图：



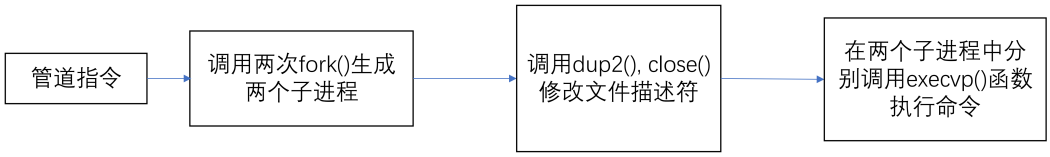

5. 管道指令

输入格式：`command1 | command2`

执行示例：

```
yuelin's shell:/home/zyl/unix_homework$ cat hello | ./main.o
hello world
a is 20021021
```

实现流程图：




# 五、源代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

#define MAX_SIZE 256
#define MAX_HISTORY_NUM 50
#define REDIRECT_FILE_N O_WRONLY|O_CREAT|O_TRUNC    // 重定向到全新文件
#define REDIRECT_FILE_A O_WRONLY|O_CREAT|O_APPEND   // 重定向附加到文件
#define REDIRECT_FILE_IN O_RDONLY                   // 输入重定向
#define REDIRECT_OUT_A 1  // >>
#define REDIRECT_OUT_N 2  // >
#define REDIRECT_IN    3  // <
#define MODE  S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH      // 重定向文件的权限 -rw-rw- r--

struct command {
    char *argv[MAX_SIZE];          // 指令参数
    char *target_file;             // 重定向输入或输出到的文件
    char *cd_path;                 // cd指令跳转到的路径
    int history_num;               // history指令查询的条数
};

char input[MAX_SIZE] = {0};
```

```c
struct command cmd1, cmd2;
int redirect_flag;
int pipeline_flag;
int history_flag;
int cd_flag;
int pipefd[2];
int fd;
pid_t pid;
char history_cmd[MAX_HISTORY_NUM][MAX_SIZE];
int history_cmd_index = 0;
char path[MAX_SIZE];
int sum_commands = 0;
char *save;  // for strtok_r MT_safe


void init();
void parse_input();
void parse_command();
void execute();

int main() {

    getcwd(path, MAX_SIZE);
    int i;
    for(i = 0; i < MAX_HISTORY_NUM; i++) {
        memset(history_cmd[i], 0, MAX_SIZE);                    // 初始化历史指令
数组
    }

    while (1)
    {
        init();                                                  // 初始化
        printf("yuelin's shell:%s$ ", path);                     // 输出提示符
        fgets(input, MAX_SIZE, stdin);                           // 读取输入
        if (input[strlen(input) - 1] == '\n') {
            input[strlen(input) - 1] = '\0';
        }
        if(strncmp(input, "history", strlen("history")) != 0 && !(input[0] ==
'h' && input[1] > '0' && input[1] <= '9' && input[2] == 0)) {
            strcpy(history_cmd[history_cmd_index++], input);          // 保存历史
指令，不包含history
            history_cmd_index = history_cmd_index % MAX_HISTORY_NUM;
            sum_commands ++;
        }
        if (strcmp(input, "exit") == 0) {
            printf("Shell closed.\n");
            break;
        }
        parse_input();
        execute();


    }
}

void init() {
    int index = 0;
    for (index = 0; index < MAX_SIZE; index ++) {
```

```
        input[index] = 0;
        cmd1.argv[index] = 0;
        cmd2.argv[index] = 0;
    }
    redirect_flag = 0;
    pipeline_flag = 0;
    history_flag = 0;
    cd_flag = 0;
    cmd1.history_num = 0;
    memset(input, 0, MAX_SIZE);
}

void parse_input() {
    if ((input[0] == 'h' && input[1] > '0' && input[1] <= '9' && input[2] == 0)
|| strncmp(input, "history", strlen("history")) == 0) {
        if(strcmp(input, "history") != 0) {
            if(input[1] > '0' && input[1] <= '9') {
                cmd1.history_num = input[1] - '0';
            } else {
                cmd1.history_num = atoi(strtok_r(input, "history", &save));
            }
            int index = (history_cmd_index - cmd1.history_num + MAX_HISTORY_NUM)
% MAX_HISTORY_NUM;
            if(history_cmd[index][0] == 0) {
                printf("Error: former %d command not exist\n",
cmd1.history_num);
            } else {
                printf("execute %s\n", history_cmd[index]);
                memcpy(input, history_cmd[index], MAX_SIZE);
            }
        } else {
            history_flag = 1;
            return;
        }
    }
    if (strncmp(input, "cd",  strlen("cd")) == 0) {
        cd_flag = 1;
        cmd1.cd_path = strtok_r(input, "cd ", &save);
        return;
    } else if (strstr(input, ">>") != NULL) {
        parse_command(strtok_r(input, ">>", &save), cmd1.argv);
        cmd1.target_file = strtok_r(strtok_r(NULL, ">>", &save), " ", &save);
        redirect_flag = REDIRECT_OUT_A;
        return;
    } else if (strstr(input, ">") != NULL) {
        parse_command(strtok_r(input, ">", &save), cmd1.argv);
        cmd1.target_file = strtok_r(strtok_r(NULL, ">", &save), " ", &save);;
        redirect_flag = REDIRECT_OUT_N;
        return;
    } else if (strstr(input, "<") != NULL) {
        parse_command(strtok_r(input, "<", &save), cmd1.argv);
        cmd1.target_file = strtok_r(strtok_r(NULL, "<", &save), " ", &save);
        redirect_flag = REDIRECT_IN;
        return;
    } else if (strstr(input, "|") != NULL) {
        parse_command(strtok_r(input, "|", &save), cmd1.argv);
        parse_command(strtok_r(NULL, "|", &save), cmd2.argv);
        pipeline_flag = 1;
```

```
        return;
    }
    parse_command(input, cmd1.argv);
}

void parse_command(char *command, char *argv[]) {
    char* tmp;
    int index = 0;
    argv[index++] = strtok_r(command, " ", &tmp);
    int i = 0;
    while ((argv[index++] = strtok_r(NULL, " ", &tmp)) != NULL);
}

void execute() {
    if(history_flag == 1) {   // 查询历史指令
        if(sum_commands > MAX_HISTORY_NUM) {
            sum_commands = MAX_HISTORY_NUM;
        }
        int i, index;
        for(i = history_cmd_index, index = sum_commands;i < MAX_HISTORY_NUM; i
++,index --) {
            if(history_cmd[i][0] == 0) {
                break;
            }
            printf("%d\t%s\n", index, history_cmd[i]);
        }
        for(i = 0; i < history_cmd_index; i++, index--) {
            if(history_cmd[i][0] == 0) {
                break;
            }
            printf("%d\t%s\n", index, history_cmd[i]);
        }
        return;
    }
    if(cd_flag != 0) {  // 切换工作目录指令
        if (chdir(cmd1.cd_path) != 0) {
            perror("change directory failed!");
        }
        getcwd(path, MAX_SIZE);
        return;
    }
    if(redirect_flag != 0) {  // 重定向指令
        pid = fork();
        if(pid < 0) {   // error
            perror("subprocess fork failed!");
            return;
        } else if (pid == 0) {  // 子进程
            if (redirect_flag == REDIRECT_OUT_A) {
                fd = open(cmd1.target_file, REDIRECT_FILE_A, MODE);
            } else if (redirect_flag == REDIRECT_OUT_N) {
                fd = open(cmd1.target_file, REDIRECT_FILE_N, MODE);
            } else {
                fd = open(cmd1.target_file, REDIRECT_FILE_IN, MODE);
            }
            if (fd < 0) {
                char *msg;
                sprintf(msg, "file %s cannot be opened!", cmd1.target_file);
                perror(msg);
```

```
                return;
            }
            if (redirect_flag == REDIRECT_IN) {
                dup2(fd, STDIN_FILENO);
            } else {
                dup2(fd, STDOUT_FILENO);
            }
            close(fd);
            execvp(cmd1.argv[0], cmd1.argv);
            perror("exec failed!");
            exit(-1);
            return;
        } else {  // 父进程
            waitpid(0, NULL, 0);
        }
        return;
    }
    if(pipeline_flag != 0) {  // 管道指令
        if (pipe(pipefd) < 0) {  // pipefd[0] is read end, while pipefd[1] is
write end
            perror("pipeline pipe failed");
            return;
        }

        pid_t pid1 = fork();
        if (pid1 < 0) {
            perror("subprocess fork failed!");
            return;
        } else if (pid1 == 0) {
            dup2(pipefd[1], STDOUT_FILENO);  // replace stdout with write end
            close(pipefd[0]);
            close(pipefd[1]);
            execvp(cmd1.argv[0], cmd1.argv);
            perror("exec failed!");
            exit(-1);
            return;
        } else {
            pid_t pid2 = fork();
            if(pid2 < 0) {
                perror("subprocess fork failed!");
                return;
            } else if (pid2 == 0) {
                dup2(pipefd[0], STDIN_FILENO);  // replace stdin with read end
                close(pipefd[0]);
                close(pipefd[1]);
                execvp(cmd2.argv[0], cmd2.argv);
                perror("exec failed!");
                exit(-1);
                return;
            } else {
                close(pipefd[0]);
                close(pipefd[1]);
                waitpid(0, NULL, 0);
                waitpid(0, NULL, 0);
            }
        }
    }
    if((pid = fork()) < 0) {
```

```
        perror("subprocess fork failed!");
        return;
    } else if(pid == 0) {
        execvp(cmd1.argv[0], cmd1.argv);
        perror("exec failed!");
        exit(-1);
        return;
    } else {
        waitpid(0, NULL, 0);
    }
    return;
}
```