重庆邮电大学

学生实验报告册

学年学期：2024 -2025 学年 □春☑秋学期

课程名称：计算机网络

学生学院：国际学院

专业班级：34082201

学生学号：2022214961

学生姓名：周明宇

联系电话：13329148059

重庆邮电大学教务处制

课程名称	计算机网络	课程编号	A2130350
实验地点	综合实验楼 C410/C411		
实验名称	Operating Systems Project2		

一、 实验目的

1. 理解 Linux 进程管理机制，掌握 fork()、exec()系列函数的使用。
2. 掌握进程间通信（IPC）机制，特别是管道（pipe）的使用。
3. 实现一个简单的 Shell 程序，能够解析并执行包含管道（|）和输出重定向（>）的命令。
4. 熟悉 Linux 文件 I/O 操作，实现输出重定向功能。

二、 实验内容

①实验任务简述

用 C 语言编写一个简易 Shell 程序 myShell.c，在 Linux 环境下运行后，显示提示符：

```
MyShPrompt >
```

用户可以输入形如：

```
command1 | command2 | command3 | command4 > file.txt
```

的命令行，程序需完成如下功能：

- 最多支持 **4 个命令**用|管道连接。
- 最终命令的输出可通过>重定向至文件。
- 每个命令可以包含参数（如 wc -l）。
- 自动解析命令行、创建子进程、设置管道与重定向，执行命令。

示例命令

```
MyShPrompt> cat country.txt city.txt | egrep 'g' | sort | more > countryCitygSorted.txt
MyShPrompt> cat country.txt city.txt | egrep 'g' | sort | wc -l > countryCitygCount.txt
```

要求使用以下命令：

- 使用 fork()创建子进程
- 用 pipe()实现命令间通信
- 用 exec()执行命令
- 用 dup2()重定向输入输出

②输入文件

- country.txt

```
zimbabwe
russia
australia
brazil
china
denmark
germany
france
angola
italy
japan
korea
poland
mexico
nicaragua
```

- city.txt

```
miami
shanghai
albany
chongqing
tokyo
beijing
detroit
new york
hong kong
macau
```

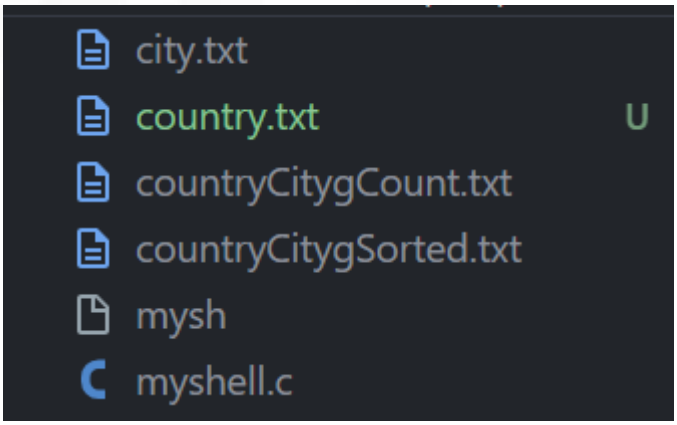
③输出文件：

- countryCitygSorted.txt
- countryCitygCount.txt

三、 实验步骤及方案

①流程图

②项目总体框架



③代码细节

1 显示提示符

```
int getCurWorkDir() { // Get current working directory
    char* result = getcwd(curPath, BUF_SZ);
    if (result == NULL)
        return ERROR_SYSTEM;
    else return RESULT_NORMAL;
}
/* Get current working directory, username, hostname */
    int result = getCurWorkDir();
    if (ERROR_SYSTEM == result) {
        fprintf(stderr, "\e[31;1mError: System error while getting current work
directory.\n\e[0m");
        exit(ERROR_SYSTEM);
    }
```

获取当前工作目录。如果失败，退出程序

```
void getUsername() { // Get current logged in username
    struct passwd* pwd = getpwuid(getuid());
    strcpy(username, pwd->pw_name);
}

void getHostname() { // Get hostname
    gethostname(hostname, BUF_SZ);
}
```

获取当前的用户名和主机名，以便在 shell 提示符中显示

```
// Shell loop
while (TRUE) {
    // show prompt
    printf("\e[32;1m%s@%s:%s\e[0m$ ", username, hostname, curPath);
}
```

拼接并打印提示符

2 提取参数

将用户输入的一整条命令按照空格分割成一个个参数，并返回参数个数

```
int splitCommands(char command[BUF_SZ]) { // Split command by space, return count of strings
    int num = 0;
    int i, j;
    int len = strlen(command);

    for (i=0, j=0; i<len; ++i) {
        if (command[i] != ' ') {
            commands[num][j++] = command[i];
        } else {
            if (j != 0) {
                commands[num][j] = '\0';
                ++num;
                j = 0;
            }
        }
    }
    if (j != 0) {
        commands[num][j] = '\0';
        ++num;
    }

    return num;
}
```

- 构造管道：创建一个管道，用于在父进程和子进程之间传输数据。
- 子进程中调用 command -v：子进程通过 system()执行 command -v <command>来检查命令是否存在，并将结果重定向到管道。
- 父进程读取管道输出：父进程通过管道读取子进程的输出，并根据输出判断命令是否存在。
- 根据输出判断命令是否存在：如果管道没有输出，表示命令不存在，返回 FALSE；否则返回 TRUE。

- 恢复标准输入输出：恢复父进程的标准输入和输出流，以保持正常的输入输出操作。

3 判断执行是否可行

判断用户输入的命令是否是有效的、可执行的命令

```
int isCommandExist(const char* command) { // Check if command exists
    if (command == NULL || strlen(command) == 0) return FALSE;

    int result = TRUE;

    int fds[2];
    if (pipe(fds) == -1) {
        result = FALSE;
    } else {
        /* Temporarily store input/output redirection flags */
        int inFd = dup(STDIN_FILENO);
        int outFd = dup(STDOUT_FILENO);

        pid_t pid = vfork();
        if (pid == -1) {
            result = FALSE;
        } else if (pid == 0) {
            /* Redirect output to file descriptor */
            close(fds[0]);
            dup2(fds[1], STDOUT_FILENO);
            close(fds[1]);

            char tmp[BUF_SZ];
            sprintf(tmp, "command -v %s", command);
            system(tmp);
            exit(1);
        } else {
            waitpid(pid, NULL, 0);
            /* Input redirection */
            close(fds[1]);
            dup2(fds[0], STDIN_FILENO);
            close(fds[0]);

            if (getchar() == EOF) { // No data means command doesn't exist
                result = FALSE;
            }

            /* Restore input/output redirection */
            dup2(inFd, STDIN_FILENO);
            dup2(outFd, STDOUT_FILENO);
        }
```

```
    }

    return result;
}
```

- 构造管道 pipe
- 子进程中调用 command -v
- 将结果输出重定向到管道
- 父进程读取管道输出
- 根据是否有输出判断命令是否存在

4 执行命令

创建一个子进程来执行用户输入的命令

```
int callCommand(int commandNum) { // Function for user to execute commands
    pid_t pid = fork();
    if (pid == -1) {
        return ERROR_FORK;
    } else if (pid == 0) {
        /* Get standard input/output file descriptors */
        int inFds = dup(STDIN_FILENO);
        int outFds = dup(STDOUT_FILENO);

        int result = callCommandWithPipe(0, commandNum);

        /* Restore standard input/output redirection */
        dup2(inFds, STDIN_FILENO);
        dup2(outFds, STDOUT_FILENO);
        exit(result);
    } else {
        int status;
        waitpid(pid, &status, 0);
        return WEXITSTATUS(status);
    }
}
```

- 创建子进程
- 在子进程中执行命令：备份标准输入和输出文件描述符：调用 dup()备份标准输入和标准输出的文件描述符。
- 执行命令：在子进程中调用 callCommandWithPipe() 来处理命令执行。假设 callCommandWithPipe() 函数负责处理命令的具体执行，并且可能涉及管道、重定向等操作。
- 恢复标准输入输出：使用 dup2()恢复标准输入和输出的文件描述符，以确保子进程

在执行后恢复到正常的 I/O 状态。

- 退出子进程：调用 exit(result)，将 callCommandWithPipe()函数的返回值作为退出码传递给子进程。
- 在父进程中等待子进程完成：父进程通过 waitpid()等待子进程结束，并获取子进程的退出状态。
- 返回子进程的退出状态：使用 WEXITSTATUS(status)从 waitpid()返回的状态中提取子进程的退出码，并返回

5 处理管道符

执行一系列命令，如果命令包含管道，函数会创建子进程并执行命令，通过管道连接多个命令的输入输出。它递归地处理每个命令，直到所有命令都执行完成。

```
int callCommandWithPipe(int left, int right) { // Execute commands in range [left, right), may contain pipes
    if (left >= right) return RESULT_NORMAL;

    /* Check if there are pipe commands */
    int pipeIdx = -1;
    for (int i=left; i<right; ++i) {
        if (strcmp(commands[i], COMMAND_PIPE) == 0) {
            pipeIdx = i;
            break;
        }
    }
    if (pipeIdx == -1) { // No pipe command
        return callCommandWithRedi(left, right);
    } else if (pipeIdx+1 == right) { // Pipe command '|' has no following command, missing parameter
        return ERROR_PIPE_MISS_PARAMETER;
    }

    /* Execute command */
    int fds[2];
    if (pipe(fds) == -1) {
        return ERROR_PIPE;
    }
    int result = RESULT_NORMAL;
    pid_t pid = vfork();
    if (pid == -1) {
        result = ERROR_FORK;
    } else if (pid == 0) { // Child process executes single command
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO); // Redirect stdout to fds[1]
        close(fds[1]);
```

```
        result = callCommandWithRedi(left, pipeIdx);
        exit(result);
    } else { // Parent process recursively executes following commands
        int status;
        waitpid(pid, &status, 0);
        int exitCode = WEXITSTATUS(status);

        if (exitCode != RESULT_NORMAL) { // Child process didn't exit normally, print
error
            char info[4096] = {0};
            char line[BUF_SZ];
            close(fds[1]);
            dup2(fds[0], STDIN_FILENO); // Redirect stdin to fds[0]
            close(fds[0]);
            while(fgets(line, BUF_SZ, stdin) != NULL) { // Read child process error
message
                strcat(info, line);
            }
            printf("%s", info); // Print error message

            result = exitCode;
        } else if (pipeIdx+1 < right){
            close(fds[1]);
            dup2(fds[0], STDIN_FILENO); // Redirect stdin to fds[0]
            close(fds[0]);
            result = callCommandWithPipe(pipeIdx+1, right); // Recursively execute
following commands
        }
    }

    return result;
}
```

- 检查命令是否包含管道：该函数首先检查在给定的命令范围（left 到 right）内是否存在管道命令 (|)。
- 使用一个 for 循环遍历所有命令，查找是否存在管道符。如果没有找到管道命令，调用 callCommandWithRedi()执行命令（假设该函数用于处理没有管道的命令）。
- 处理管道命令：如果管道命令存在且在命令范围内有效（即 pipeIdx != -1 且 pipeIdx + 1 != right），则：
- 创建管道：使用 pipe(fds)创建一个管道。fds[0] 用于读取数据，fds[1] 用于写入数据。
- 创建子进程：使用 fork()创建一个子进程，在子进程中执行命令。在子进程中关闭管道的读取端 fds[0]，将标准输出重定向到管道的写入端 fds[1]。调用 callCommandWithRedi()执行当前命令，这个命令将其输出通过管道传递到父进程。子进程执行完命令后退出，返回结果给父进程。

- 在父进程中处理子进程输出：父进程等待子进程完成并检查退出状态：如果子进程执行异常，读取并打印子进程的错误信息（从管道中获取）。如果子进程正常退出，检查是否还有后续的命令（即管道后的命令）。
- 递归处理剩余命令：如果管道后的命令存在（即 pipeIdx+1< right），父进程会将管道的读取端 fds[0]作为标准输入，继续递归地调用 callCommandWithPipe()来执行后续命令。
- 返回执行结果：函数返回当前命令的执行结果。如果在执行过程中出现错误，将返回相应的错误代码。

6 重定向

执行给定范围内的命令，并处理可能的输入和输出重定向

```
int callCommandWithRedi(int left, int right) { // Execute commands in range [left, right), no pipes, may have redirection
    if (!isCommandExist(commands[left])) { // Command doesn't exist
        return ERROR_COMMAND;
    }

    /* Check for redirection */
    int inNum = 0, outNum = 0;
    char *inFile = NULL, *outFile = NULL;
    int endIdx = right; // End index before redirection

    for (int i=left; i<right; ++i) {
        if (strcmp(commands[i], COMMAND_IN) == 0) { // Input redirection
            ++inNum;
            if (i+1 < right)
                inFile = commands[i+1];
            else return ERROR_MISS_PARAMETER; // Missing filename after redirection symbol

            if (endIdx == right) endIdx = i;
        } else if (strcmp(commands[i], COMMAND_OUT) == 0) { // Output redirection
            ++outNum;
            if (i+1 < right)
                outFile = commands[i+1];
            else return ERROR_MISS_PARAMETER; // Missing filename after redirection symbol

            if (endIdx == right) endIdx = i;
        }
    }
    /* Handle redirection */
    if (inNum == 1) {
        FILE* fp = fopen(inFile, "r");
        if (fp == NULL) // Input redirection file doesn't exist
```

```
            return ERROR_FILE_NOT_EXIST;

        fclose(fp);
    }

    if (inNum > 1) { // More than one input redirection symbol
        return ERROR_MANY_IN;
    } else if (outNum > 1) { // More than one output redirection symbol
        return ERROR_MANY_OUT;
    }

    int result = RESULT_NORMAL;
    pid_t pid = vfork();
    if (pid == -1) {
        result = ERROR_FORK;
    } else if (pid == 0) {
        /* Input/output redirection */
        if (inNum == 1)
            freopen(inFile, "r", stdin);
        if (outNum == 1)
            freopen(outFile, "w", stdout);

        /* Execute command */
        char* comm[BUF_SZ];
        for (int i=left; i<endIdx; ++i)
            comm[i] = commands[i];
        comm[endIdx] = NULL;
        execvp(comm[left], comm+left);
        exit(errno); // Execution error, return errno
    } else {
        int status;
        waitpid(pid, &status, 0);
        int err = WEXITSTATUS(status); // Read child process return code

        if (err) { // Return code not 0 means child process error, print in red
            printf("\e[31;1mError: %s\n\e[0m", strerror(err));
        }
    }

    return result;
}
```

- 首先检查命令是否存在，如果不存在则返回 ERROR_COMMAND。
- 检查输入输出重定向通过遍历命令列表，查找重定向符号<和>。
- 如果找到<，则设置 inFile 为重定向的文件，并记录重定向符号的索引。如果文件名缺失，则返回 ERROR_MISS_PARAMETER。

- 如果找到>，则设置 outFile 为重定向的文件，并记录重定向符号的索引。如果文件名缺失，则返回 ERROR_MISS_PARAMETER。
- 如果有多个重定向符号（例如多个<或>），返回 ERROR_MANY_IN 或 ERROR_MANY_OUT。
- 处理重定向：如果有输入重定向，尝试打开文件并检查其是否存在。如果文件不存在，返回 ERROR_FILE_NOT_EXIST。
- 创建子进程在子进程中，重定向标准输入输出：如果存在输入重定向，使用 freopen 将标准输入重定向到指定的输入文件。如果存在输出重定向，使用 freopen 将标准输出重定向到指定的输出文件。
- 使用 execvp 执行命令。execvp 将命令和其参数传递给操作系统执行。如果命令执行失败，子进程退出并返回 errno。
- 父进程等待子进程，父进程使用 waitpid()等待子进程完成，并获取子进程的退出状态。如果退出状态不为零（即执行失败），父进程会打印错误信息。

7 终止进程

终止当前进程

```
int callExit() { // Send terminal signal to exit process
    pid_t pid = getpid();
    if (kill(pid, SIGTERM) == -1)
        return ERROR_EXIT;
    else return RESULT_NORMAL;
}
```

- 获取当前进程的进程 ID (pid)
- 使用 kill(pid, SIGTERM)向当前进程发送一个终止信号 SIGTERM。
- 判断是否发送信号成功：如果 kill()调用返回 -1，表示发送信号失败，函数返回 ERROR_EXIT。如果信号发送成功，返回 RESULT_NORMAL，表示操作成功。

④编译运行

1 编译

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2# gcc myshell.c -o mysh
```

2 运行

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2# ./mysh
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$
```

四、 结果及分析

①基础指令

ls 指令

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ ls
city.txt  country.txt  mysh  myshell.c  temp.c
```

能够正确输出当前路径下的文件

cat 指令

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat city.txt
miami
shanghai
albany
chongqing
tokyo
beijing
detroit
new york
hong kong
```

可以显示文件内容

重定向

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat city.txt > result.txt
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat result.txt
miami
shanghai
albany
chongqing
tokyo
beijing
detroit
new york
hong kong
```

拼接多个文件

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat city.txt country.txt > result.txt
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat result.txt
miami
shanghai
albany
chongqing
tokyo
beijing
detroit
new york
hong kong
macauzimbabwe
russia
australia
brazil
china
denmark
germany
france
angola
italy
japan
korea
poland
mexico
nicaragua
```

将拼接后的结果正确输出

排序

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat city.txt | sort > result.txt
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat result.txt
albany
beijing
chongqing
detroit
hong kong
macau
miami
new york
shanghai
tokyo
```

②执行给定指令

指令 1

```
cat country.txt city.txt | egrep 'g' | sort | more > countryCitygSorted.txt
```

结果如下

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat country.txt city.txt | egrep 'g' | sort | more > countryCitygSorted.txt
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat countryCitygSorted.txt
angola
beijing
chongqing
germany
hong kong
nicaragua
shanghai
```

指令 2

```
cat country.txt city.txt | egrep 'g' | sort | wc -l > countryCitygCount.txt
```

结果如下

```
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat country.txt city.txt | egrep 'g' | sort | wc -l > countryCitygCount.txt
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$ cat countryCitygCount.txt
7
root@LAPTOP-0CUR2SRA:/home/ICSI412/Project2$
```

将执行结果与手动求解的结果比较，shell 给出了正确的结果。

五、 心得体会

通过完成这个简单的 shell 实现项目，我对操作系统底层机制有了更深刻的理解。最初看到项目要求时，觉得实现一个能处理管道和重定向的 shell 似乎很复杂，但通过逐步拆解问题，最终发现核心原理其实很清晰。在编码过程中，最让我印象深刻的是管道机制的实现，当第一次看到多个命令通过管道连接起来正确执行时，那种成就感难以言表。

调试阶段遇到了不少困难，特别是处理文件描述符的重定向时经常出现各种意外情况。记得有一次因为忘记关闭管道未使用的端口导致程序阻塞，花了整整一个下午才找到问题所在。这个过程虽然痛苦，但让我真正理解了文件描述符在进程间传递的机制。通过不断调试和查阅资料，最终看到自己实现的 shell 能够正确处理像"cat file.txt | grep 'text' > output.txt"这样的复杂命令时，所有的努力都变得值得。

这个项目让我认识到操作系统课程中理论知识与实践结合的重要性。课本上关于进程、管道的概念通过这个项目变得具体而生动。特别是在实现过程中，我发现很多看似简单的系统调用在实际应用中需要考虑各种边界条件和错误处理，这让我对系统编程的复杂性有了新的认识。当最终完成项目并看到自己实现的 shell 能够处理各种命令组合时，不仅加深了对操作系统原理的理解，更获得了解决复杂问题的信心和能力。

代码附录

Myshell.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/signal.h>
#include <sys/types.h>
#include <errno.h>
#include <pwd.h>
```

```
#define BUF_SZ 256
#define TRUE 1
#define FALSE 0

const char* COMMAND_EXIT = "exit";
const char* COMMAND_HELP = "help";
const char* COMMAND_CD = "cd";
const char* COMMAND_IN = "<";
const char* COMMAND_OUT = ">";
const char* COMMAND_PIPE = "|";

// Built-in status codes
enum {
    RESULT_NORMAL,
    ERROR_FORK,
    ERROR_COMMAND,
    ERROR_WRONG_PARAMETER,
    ERROR_MISS_PARAMETER,
    ERROR_TOO_MANY_PARAMETER,
    ERROR_CD,
    ERROR_SYSTEM,
    ERROR_EXIT,

    /* Redirection error messages */
    ERROR_MANY_IN,
    ERROR_MANY_OUT,
    ERROR_FILE_NOT_EXIST,

    /* Pipe error messages */
    ERROR_PIPE,
    ERROR_PIPE_MISS_PARAMETER
};

char username[BUF_SZ];
char hostname[BUF_SZ];
char curPath[BUF_SZ];
char commands[BUF_SZ][BUF_SZ];

int isCommandExist(const char* command);  // Check if command exists
void getUsername();  // Get current username
void getHostname();  // Get hostname
int getCurWorkDir();  // Get current working directory
int splitCommands(char command[BUF_SZ]);  // Split commands by space, return count
int callExit();  // Exit the shell
int callCommand(int commandNum);  // Execute user command
```

```c
int callCommandWithPipe(int left, int right);  // Execute commands with pipe
int callCommandWithRedi(int left, int right);  // Execute commands with redirection
int callCd(int commandNum);  // Execute cd command

int main() {
    /* Get current working directory, username, hostname */
    int result = getCurWorkDir();
    if (ERROR_SYSTEM == result) {
        fprintf(stderr, "\e[31;1mError: System error while getting current work
directory.\n\e[0m");
        exit(ERROR_SYSTEM);
    }
    getUsername();
    getHostname();

    /* Start myshell */
    char argv[BUF_SZ];
    while (TRUE) {
        printf("\e[32;1m%s@%s:%s\e[0m$ ", username, hostname,curPath); // Display in
green
        /* Get user input command */
        fgets(argv, BUF_SZ, stdin);
        int len = strlen(argv);
        if (len != BUF_SZ) {
            argv[len-1] = '\0';
        }

        int commandNum = splitCommands(argv);

        if (commandNum != 0) { // User entered command
            if (strcmp(commands[0], COMMAND_EXIT) == 0) { // exit command
                result = callExit();
                if (ERROR_EXIT == result) {
                    exit(-1);
                }
            } else if (strcmp(commands[0], COMMAND_CD) == 0) { // cd command
                result = callCd(commandNum);
                switch (result) {
                    case ERROR_MISS_PARAMETER:
                        fprintf(stderr, "\e[31;1mError: Miss parameter while using
command \"%s\".\n\e[0m"
                                , COMMAND_CD);
                        break;
                    case ERROR_WRONG_PARAMETER:
                        fprintf(stderr, "\e[31;1mError: No such path
\"%s\".\n\e[0m", commands[1]);
```

```
                    break;
                case ERROR_TOO_MANY_PARAMETER:
                    fprintf(stderr, "\e[31;1mError: Too many parameters while
using command \"%s\".\n\e[0m"
                        , COMMAND_CD);
                    break;
                case RESULT_NORMAL: // cd command executed normally, update
current working directory
                    result = getCurWorkDir();
                    if (ERROR_SYSTEM == result) {
                        fprintf(stderr
                            , "\e[31;1mError: System error while getting current
work directory.\n\e[0m");
                        exit(ERROR_SYSTEM);
                    } else {
                        break;
                    }
            }
        } else { // Other commands
            result = callCommand(commandNum);
            switch (result) {
                case ERROR_FORK:
                    fprintf(stderr, "\e[31;1mError: Fork error.\n\e[0m");
                    exit(ERROR_FORK);
                case ERROR_COMMAND:
                    fprintf(stderr, "\e[31;1mError: Command not exist in
myshell.\n\e[0m");
                    break;
                case ERROR_MANY_IN:
                    fprintf(stderr, "\e[31;1mError: Too many redirection symbol
\"%s\".\n\e[0m", COMMAND_IN);
                    break;
                case ERROR_MANY_OUT:
                    fprintf(stderr, "\e[31;1mError: Too many redirection symbol
\"%s\".\n\e[0m", COMMAND_OUT);
                    break;
                case ERROR_FILE_NOT_EXIST:
                    fprintf(stderr, "\e[31;1mError: Input redirection file not
exist.\n\e[0m");
                    break;
                case ERROR_MISS_PARAMETER:
                    fprintf(stderr, "\e[31;1mError: Miss redirect file
parameters.\n\e[0m");
                    break;
                case ERROR_PIPE:
                    fprintf(stderr, "\e[31;1mError: Open pipe error.\n\e[0m");
```

```
                    break;
                case ERROR_PIPE_MISS_PARAMETER:
                    fprintf(stderr, "\e[31;1mError: Miss pipe
parameters.\n\e[0m");
                    break;
            }
        }
    }
}

int isCommandExist(const char* command) { // Check if command exists
    if (command == NULL || strlen(command) == 0) return FALSE;

    int result = TRUE;

    int fds[2];
    if (pipe(fds) == -1) {
        result = FALSE;
    } else {
        /* Temporarily store input/output redirection flags */
        int inFd = dup(STDIN_FILENO);
        int outFd = dup(STDOUT_FILENO);

        pid_t pid = vfork();
        if (pid == -1) {
            result = FALSE;
        } else if (pid == 0) {
            /* Redirect output to file descriptor */
            close(fds[0]);
            dup2(fds[1], STDOUT_FILENO);
            close(fds[1]);

            char tmp[BUF_SZ];
            sprintf(tmp, "command -v %s", command);
            system(tmp);
            exit(1);
        } else {
            waitpid(pid, NULL, 0);
            /* Input redirection */
            close(fds[1]);
            dup2(fds[0], STDIN_FILENO);
            close(fds[0]);

            if (getchar() == EOF) { // No data means command doesn't exist
                result = FALSE;
```

```
            }

            /* Restore input/output redirection */
            dup2(inFd, STDIN_FILENO);
            dup2(outFd, STDOUT_FILENO);
        }
    }

    return result;
}

void getUsername() { // Get current logged in username
    struct passwd* pwd = getpwuid(getuid());
    strcpy(username, pwd->pw_name);
}

void getHostname() { // Get hostname
    gethostname(hostname, BUF_SZ);
}

int getCurWorkDir() { // Get current working directory
    char* result = getcwd(curPath, BUF_SZ);
    if (result == NULL)
        return ERROR_SYSTEM;
    else return RESULT_NORMAL;
}

int splitCommands(char command[BUF_SZ]) { // Split command by space, return count of
strings
    int num = 0;
    int i, j;
    int len = strlen(command);

    for (i=0, j=0; i<len; ++i) {
        if (command[i] != ' ') {
            commands[num][j++] = command[i];
        } else {
            if (j != 0) {
                commands[num][j] = '\0';
                ++num;
                j = 0;
            }
        }
    }
    if (j != 0) {
        commands[num][j] = '\0';
```

```
        ++num;
    }

    return num;
}

int callExit() { // Send terminal signal to exit process
    pid_t pid = getpid();
    if (kill(pid, SIGTERM) == -1)
        return ERROR_EXIT;
    else return RESULT_NORMAL;
}

int callCommand(int commandNum) { // Function for user to execute commands
    pid_t pid = fork();
    if (pid == -1) {
        return ERROR_FORK;
    } else if (pid == 0) {
        /* Get standard input/output file descriptors */
        int inFds = dup(STDIN_FILENO);
        int outFds = dup(STDOUT_FILENO);

        int result = callCommandWithPipe(0, commandNum);

        /* Restore standard input/output redirection */
        dup2(inFds, STDIN_FILENO);
        dup2(outFds, STDOUT_FILENO);
        exit(result);
    } else {
        int status;
        waitpid(pid, &status, 0);
        return WEXITSTATUS(status);
    }
}

int callCommandWithPipe(int left, int right) { // Execute commands in range [left,
right), may contain pipes
    if (left >= right) return RESULT_NORMAL;
    /* Check if there are pipe commands */
    int pipeIdx = -1;
    for (int i=left; i<right; ++i) {
        if (strcmp(commands[i], COMMAND_PIPE) == 0) {
            pipeIdx = i;
            break;
        }
    }
```

```
    if (pipeIdx == -1) { // No pipe command
        return callCommandWithRedi(left, right);
    } else if (pipeIdx+1 == right) { // Pipe command '|' has no following command, missing parameter
        return ERROR_PIPE_MISS_PARAMETER;
    }

    /* Execute command */
    int fds[2];
    if (pipe(fds) == -1) {
        return ERROR_PIPE;
    }
    int result = RESULT_NORMAL;
    pid_t pid = vfork();
    if (pid == -1) {
        result = ERROR_FORK;
    } else if (pid == 0) { // Child process executes single command
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO); // Redirect stdout to fds[1]
        close(fds[1]);

        result = callCommandWithRedi(left, pipeIdx);
        exit(result);
    } else { // Parent process recursively executes following commands
        int status;
        waitpid(pid, &status, 0);
        int exitCode = WEXITSTATUS(status);

        if (exitCode != RESULT_NORMAL) { // Child process didn't exit normally, print error
            char info[4096] = {0};
            char line[BUF_SZ];
            close(fds[1]);
            dup2(fds[0], STDIN_FILENO); // Redirect stdin to fds[0]
            close(fds[0]);
            while(fgets(line, BUF_SZ, stdin) != NULL) { // Read child process error message
                strcat(info, line);
            }
            printf("%s", info); // Print error message

            result = exitCode;
        } else if (pipeIdx+1 < right){
            close(fds[1]);
            dup2(fds[0], STDIN_FILENO); // Redirect stdin to fds[0]
            close(fds[0]);
```

```
            result = callCommandWithPipe(pipeIdx+1, right); // Recursively execute
following commands
        }
    }

    return result;
}

int callCommandWithRedi(int left, int right) { // Execute commands in range [left,
right), no pipes, may have redirection
    if (!isCommandExist(commands[left])) { // Command doesn't exist
        return ERROR_COMMAND;
    }

    /* Check for redirection */
    int inNum = 0, outNum = 0;
    char *inFile = NULL, *outFile = NULL;
    int endIdx = right; // End index before redirection

    for (int i=left; i<right; ++i) {
        if (strcmp(commands[i], COMMAND_IN) == 0) { // Input redirection
            ++inNum;
            if (i+1 < right)
                inFile = commands[i+1];
            else return ERROR_MISS_PARAMETER; // Missing filename after redirection
symbol

            if (endIdx == right) endIdx = i;
        } else if (strcmp(commands[i], COMMAND_OUT) == 0) { // Output redirection
            ++outNum;
            if (i+1 < right)
                outFile = commands[i+1];
            else return ERROR_MISS_PARAMETER; // Missing filename after redirection
symbol

            if (endIdx == right) endIdx = i;
        }
    }
    /* Handle redirection */
    if (inNum == 1) {
        FILE* fp = fopen(inFile, "r");
        if (fp == NULL) // Input redirection file doesn't exist
            return ERROR_FILE_NOT_EXIST;

        fclose(fp);
    }
```

```
    if (inNum > 1) { // More than one input redirection symbol
        return ERROR_MANY_IN;
    } else if (outNum > 1) { // More than one output redirection symbol
        return ERROR_MANY_OUT;
    }

    int result = RESULT_NORMAL;
    pid_t pid = vfork();
    if (pid == -1) {
        result = ERROR_FORK;
    } else if (pid == 0) {
        /* Input/output redirection */
        if (inNum == 1)
            freopen(inFile, "r", stdin);
        if (outNum == 1)
            freopen(outFile, "w", stdout);

        /* Execute command */
        char* comm[BUF_SZ];
        for (int i=left; i<endIdx; ++i)
            comm[i] = commands[i];
        comm[endIdx] = NULL;
        execvp(comm[left], comm+left);
        exit(errno); // Execution error, return errno
    } else {
        int status;
        waitpid(pid, &status, 0);
        int err = WEXITSTATUS(status); // Read child process return code

        if (err) { // Return code not 0 means child process error, print in red
            printf("\e[31;1mError: %s\n\e[0m", strerror(err));
        }
    }

    return result;
}

int callCd(int commandNum) { // Execute cd command
    int result = RESULT_NORMAL;

    if (commandNum < 2) {
        result = ERROR_MISS_PARAMETER;
    } else if (commandNum > 2) {
        result = ERROR_TOO_MANY_PARAMETER;
    } else {
```

```
        int ret = chdir(commands[1]);
        if (ret) result = ERROR_WRONG_PARAMETER;
    }

    return result;
}
```