

重庆邮电大学

学生实验报告册

第 X 次 实 验 报 告

学年学期： 2024 -2025 学年 ☐春☒秋学期

课程名称： 操作系统

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 Project1		

一、实验目的

- 理解 Linux 进程管理机制，掌握 fork()、exec()系列函数的使用。
- 掌握进程间通信（IPC）机制，特别是管道（pipe）的使用。
- 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。
- 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

```
1  zimbabwe
2  russia
3  australia
4  brazil
5  china
6  denmark
7  germany
8  france
9  angola
10 italy
11 japan
12 korea
13 poland
14 mexico
15 nicaragua
```

图 1 country.txt

```
1  miami
2  shanghai
3  albany
4  chongqing
5  tokyo
6  beijing
7  detroit
8  new york
9  hong kong
10 macau
```

图 2 city.txt

③输出文件:

- countryCitygSorted.txt
- countryCitygCount.txt

三、 实验步骤及方案

①流程图

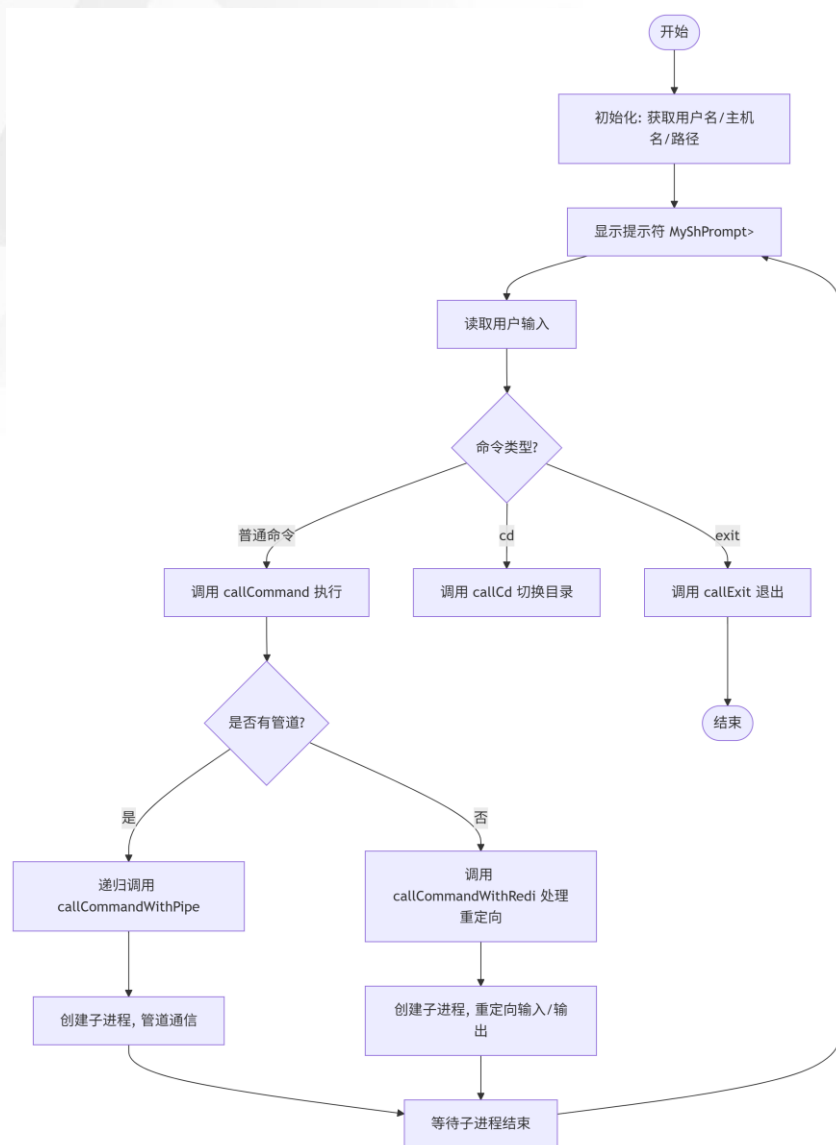


图 3 系统流程图

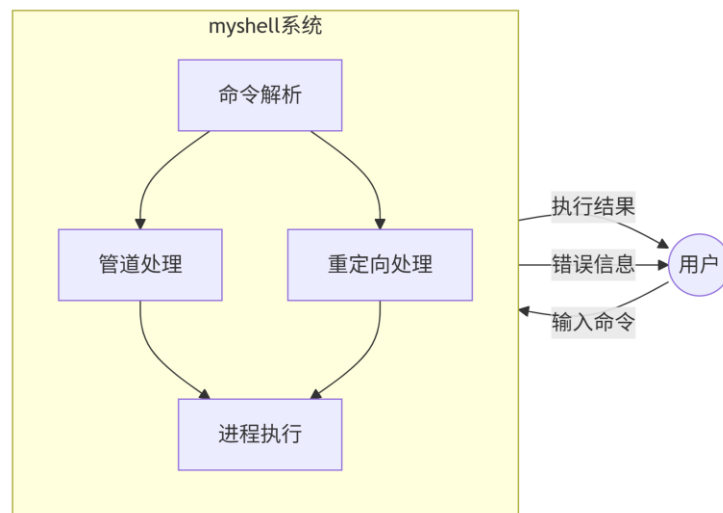


图 4 数据流图

②项目总体框架

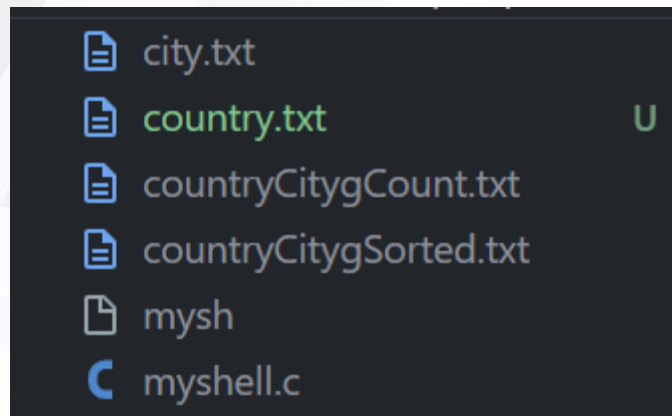


图 5 系统架构图

③代码细节

1 显示提示符

```
// 获取当前工作目录
int getCurWorkDir() {
    char* result = getcwd(curPath, BUF_SZ);
    if (result == NULL)
        return ERROR_SYSTEM;
    else return RESULT_NORMAL;
}
```

获取当前工作目录。如果失败，退出程序

```
// 获取当前用户名
void getUsername() {
    struct passwd* pwd = getpwuid(getuid());
    strcpy(username, pwd->pw_name);
}

// 获取主机名
void getHostname() {
    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

```
while (TRUE) {
    // 打印提示符（绿色）
    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

```
// 将命令按空格分割，返回分割后的命令数量
```

```

int splitCommands(char command[BUF_SZ]) {
    int num = 0; // 命令计数
    int i, j;
    int len = strlen(command); // 获取命令字符串长度

    for (i = 0, j = 0; i < len; ++i) {
        if (command[i] != ' ') { // 如果不是空格，加入当前命令
            commands[num][j++] = command[i];
        } else {
            if (j != 0) { // 遇到空格且当前命令非空，结束当前命令
                commands[num][j] = '\0'; // 添加字符串结束符
                ++num;
                j = 0;
            }
        }
    }

    if (j != 0) { // 最后一个命令处理
        commands[num][j] = '\0';
        ++num;
    }

    return num; // 返回命令数
}

```

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

- 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

```

// 检查命令是否存在
int isCommandExist(const char* command) {
    if (command == NULL || strlen(command) == 0) return FALSE;

    int result = TRUE;
    int fds[2];

    if (pipe(fds) == -1) {

```

```

    result = FALSE;
} else {
    // 保存标准输入输出
    int inFd = dup(STDIN_FILENO);
    int outFd = dup(STDOUT_FILENO);

    pid_t pid = vfork();
    if (pid == -1) {
        result = FALSE;
    } else if (pid == 0) {
        // 子进程重定向输出到管道
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);

        // 使用 system 调用 command -v 检查命令
        char tmp[BUF_SZ];
        sprintf(tmp, "command -v %s", command);
        system(tmp);
        exit(1);
    } else {
        // 父进程等待子进程
        waitpid(pid, NULL, 0);
        // 读取子进程输出
        close(fds[1]);
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);

        // 如果无输出，说明命令不存在
        if (getchar() == EOF) {
            result = FALSE;
        }

        // 恢复标准输入输出
        dup2(inFd, STDIN_FILENO);
        dup2(outFd, STDOUT_FILENO);
    }
}

return result;
}

```

判断用户输入的命令是否是有效的、可执行的命令

- 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

```
// 执行用户命令（带子进程处理）
int callCommand(int commandNum) {
    pid_t pid = fork(); // 创建子进程
    if (pid == -1) {
        return ERROR_FORK; // 创建失败
    } else if (pid == 0) { // 子进程部分
        // 复制标准输入输出文件描述符
        int inFds = dup(STDIN_FILENO);
        int outFds = dup(STDOUT_FILENO);

        int result = callCommandWithPipe(0, commandNum); // 调用管道执行命令

        // 恢复标准输入输出
        dup2(inFds, STDIN_FILENO);
        dup2(outFds, STDOUT_FILENO);
        exit(result); // 退出并返回结果
    } else { // 父进程部分
        int status;
        waitpid(pid, &status, 0); // 等待子进程完成
        return WEXITSTATUS(status); // 返回子进程退出码
    }
}
```

创建一个子进程来执行用户输入的命令

- 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。
- 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。

5 处理管道符

```
// 在范围[left, right)内执行命令，支持管道操作
```



```
int callCommandWithPipe(int left, int right) {
    if (left >= right) return RESULT_NORMAL; // 无命令直接返回

    int pipeIdx = -1; // 用于标记管道符“|”位置

    // 查找管道符
    for (int i = left; i < right; ++i) {
        if (strcmp(commands[i], COMMAND_PIPE) == 0) {
            pipeIdx = i;
            break;
        }
    }

    if (pipeIdx == -1) {
        // 没有管道符，直接执行命令（可能有重定向）
        return callCommandWithRedi(left, right);
    } else if (pipeIdx + 1 == right) {
        // 管道符后没有命令，参数缺失
        return ERROR_PIPE_MISS_PARAMETER;
    }

    // 执行带管道的命令
    int fds[2]; // 管道文件描述符：fds[0]读端，fds[1]写端
    if (pipe(fds) == -1) {
        return ERROR_PIPE; // 创建管道失败
    }

    int result = RESULT_NORMAL;
    pid_t pid = vfork(); // 创建子进程执行左侧命令
    if (pid == -1) {
        result = ERROR_FORK; // 创建失败
    } else if (pid == 0) {
        // 子进程执行左边命令，将输出重定向到管道写端
        close(fds[0]); // 关闭管道读端
        dup2(fds[1], STDOUT_FILENO); // 标准输出重定向到管道写端
        close(fds[1]);

        result = callCommandWithRedi(left, pipeIdx); // 执行左边命令
        exit(result); // 退出子进程
    } else {
        // 父进程等待子进程结束，然后继续执行右边命令
        int status;
        waitpid(pid, &status, 0);
        int exitCode = WEXITSTATUS(status); // 获取子进程退出码

        if (exitCode != RESULT_NORMAL) {
```

```

        // 如果子进程出错，从管道中读取错误信息
        char info[4096] = {0};
        char line[BUF_SZ];
        close(fds[1]); // 关闭写端
        dup2(fds[0], STDIN_FILENO); // 将标准输入重定向到管道读端
        close(fds[0]);
        while (fgets(line, BUF_SZ, stdin) != NULL) {
            strcat(info, line); // 拼接错误信息
        }
        printf("%s", info); // 打印错误信息

        result = exitCode;
    } else if (pipeIdx + 1 < right) {
        // 正常执行且还有右边命令，继续递归处理右边部分
        close(fds[1]); // 关闭写端
        dup2(fds[0], STDIN_FILENO); // 标准输入重定向为管道读端
        close(fds[0]);
        result = callCommandWithPipe(pipeIdx + 1, right);
    }
}

return result;
}

```

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。

- 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。
- 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。
- 创建子进程：使用 fork() 创建一个子进程，在子进程中执行命令。调用 callCommandWithRedi() 执行当前命令，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。
- 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

```
int callCommandWithRedi(int left, int right) { // 执行命令区间 [left, right), 无管道, 但可能包含重定向
    if (!isCommandExist(commands[left])) { // 命令不存在
        return ERROR_COMMAND;
    }

    /* 检查是否包含重定向 */
    int inNum = 0, outNum = 0;
    char *inFile = NULL, *outFile = NULL;
    int endIdx = right; // 命令真正结束的位置 (不包含重定向部分)

    for (int i=left; i<right; ++i) {
        if (strcmp(commands[i], COMMAND_IN) == 0) { // 输入重定向符号 "<"
            ++inNum;
            if (i+1 < right)
                inFile = commands[i+1]; // 获取输入文件名
            else return ERROR_MISS_PARAMETER; // 缺少重定向后面的参数 (文件名)

            if (endIdx == right) endIdx = i; // 记录命令部分的终点
        } else if (strcmp(commands[i], COMMAND_OUT) == 0) { // 输出重定向符号 ">"
            ++outNum;
            if (i+1 < right)
                outFile = commands[i+1]; // 获取输出文件名
            else return ERROR_MISS_PARAMETER; // 缺少重定向后面的参数

            if (endIdx == right) endIdx = i;
        }
    }

    /* 输入文件存在性检查 */
    if (inNum == 1) {
        FILE* fp = fopen(inFile, "r");
        if (fp == NULL) // 输入文件不存在
            return ERROR_FILE_NOT_EXIST;
        fclose(fp);
    }

    /* 检查是否有多个重定向符号 */
    if (inNum > 1) {
        return ERROR_MANY_IN; // 输入重定向重复
    } else if (outNum > 1) {
        return ERROR_MANY_OUT; // 输出重定向重复
    }

    int result = RESULT_NORMAL;
```

```

pid_t pid = vfork(); // 创建子进程
if (pid == -1) {
    result = ERROR_FORK;
} else if (pid == 0) {
    /* 重定向处理 */
    if (inNum == 1)
        freopen(inFile, "r", stdin); // 重定向 stdin
    if (outNum == 1)
        freopen(outFile, "w", stdout); // 重定向 stdout

    /* 执行命令 */
    char* comm[BUF_SZ];
    for (int i=left; i<endIdx; ++i)
        comm[i] = commands[i];
    comm[endIdx] = NULL;

    execvp(comm[left], comm+left); // 使用 execvp 执行命令
    exit(errno); // 若 exec 失败，返回 errno 作为退出码
} else {
    int status;
    waitpid(pid, &status, 0); // 等待子进程结束
    int err = WEXITSTATUS(status); // 获取退出状态

    if (err) { // 非正常退出，输出红色错误信息
        printf("\e[31;1mError: %s\n\e[0m", strerror(err));
    }
}

return result;
}

```

执行给定范围内的命令，并处理可能的输入和输出重定向

- 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
- 使用 **execvp** 执行命令
- 父进程等待子进程，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

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

终止当前进程

- 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
```

图 6 编译

2 运行

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

图 7 运行

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
```

图 8 ls 指令运行结果

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
```

图 9 cat 指令运行结果

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
```

图 10 重定向运行结果

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
```

图 11 拼接后输出

将拼接后的结果正确输出

排序

```
root@LAPTOP-OCUR2SRA:/home/ICSI412/Project2$ cat city.txt | sort > result.txt
root@LAPTOP-OCUR2SRA:/home/ICSI412/Project2$ cat result.txt
albany
beijing
chongqing
detroit
hong kong
macau
miami
new york
shanghai
tokyo
```

图 12 文件内容排序后输出

②执行给定指令

指令 1

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

图 13 运行指令 1

结果如下

```
root@LAPTOP-OCUR2SRA:/home/ICSI412/Project2$ cat country.txt city.txt | egrep 'g' | sort | more > countryCitygSorted.txt
root@LAPTOP-OCUR2SRA:/home/ICSI412/Project2$ cat countryCitygSorted.txt
angola
beijing
chongqing
germany
hong kong
nicaragua
shanghai
```

图 14 指令 1 运行结果

指令 2

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

图 13 运行指令 2

结果如下

```
root@LAPTOP-OCUR2SRA:/home/ICSI412/Project2$ cat country.txt city.txt | egrep 'g' | sort | wc -l > countryCitygCount.txt
root@LAPTOP-OCUR2SRA:/home/ICSI412/Project2$ cat countryCitygCount.txt
7
root@LAPTOP-OCUR2SRA:/home/ICSI412/Project2$
```

图 16 指令 2 运行结果

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。

附录

countryCitygSorted.txt

```
angola
beijing
chongqing
germany
hong kong
nicaragua
shanghai
```

countryCitygCount.txt 中保存统计结果，为 7

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

// 内置命令
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 = "|";
```



```
// 状态码定义
enum {
    RESULT_NORMAL,
    ERROR_FORK,
    ERROR_COMMAND,
    ERROR_WRONG_PARAMETER,
    ERROR_MISS_PARAMETER,
    ERROR_TOO_MANY_PARAMETER,
    ERROR_CD,
    ERROR_SYSTEM,
    ERROR_EXIT,

    // 重定向错误信息
    ERROR_MANY_IN,
    ERROR_MANY_OUT,
    ERROR_FILE_NOT_EXIST,

    // 管道错误信息
    ERROR_PIPE,
    ERROR_PIPE_MISS_PARAMETER
};

char username[BUF_SZ]; // 用户名
char hostname[BUF_SZ]; // 主机名
char curPath[BUF_SZ]; // 当前路径
char commands[BUF_SZ][BUF_SZ]; // 拆分后的命令数组

// 函数声明
int isCommandExist(const char* command); // 检查命令是否存在
void getUsername(); // 获取当前用户名
void getHostname(); // 获取主机名
int getCurWorkDir(); // 获取当前工作目录
int splitCommands(char command[BUF_SZ]); // 按空格拆分命令，返回命令数
int callExit(); // 处理 exit 命令
int callCommand(int commandNum); // 处理普通命令
int callCommandWithPipe(int left, int right); // 处理带管道的命令
int callCommandWithRedi(int left, int right); // 处理带重定向的命令
int callCd(int commandNum); // 处理 cd 命令

int main() {
    // 初始化：获取当前工作目录、用户名、主机名
    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();

// 启动 Shell
char argv[BUF_SZ];
while (TRUE) {
    // 打印提示符（绿色）
    printf("\e[32;1m%s@%s:%s\e[0m$ ", username, hostname, curPath);

    // 读取用户输入
    fgets(argv, BUF_SZ, stdin);
    int len = strlen(argv);
    if (len != BUF_SZ) {
        argv[len-1] = '\0'; // 去掉换行符
    }

    int commandNum = splitCommands(argv); // 拆分命令

    if (commandNum != 0) { // 用户输入了非空命令
        if (strcmp(commands[0], COMMAND_EXIT) == 0) { // 处理 exit 命令
            result = callExit();
            if (ERROR_EXIT == result) {
                exit(-1);
            }
        } else if (strcmp(commands[0], COMMAND_CD) == 0) { // 处理 cd 命令
            result = callCd(commandNum);
            switch (result) {
                case ERROR_MISS_PARAMETER:
                    fprintf(stderr, "\e[31;1mError: Miss parameter while using
command \"%s\".\n\e[0m",
                        commands[0], 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",
                        commands[0], COMMAND_CD);
                    break;
                case RESULT_NORMAL: // 执行 cd 成功后，更新当前目录
                    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 { // 其他普通命令
    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;
    }
}

}

}

}

```

```
// 检查命令是否存在
int isCommandExist(const char* command) {
    if (command == NULL || strlen(command) == 0) return FALSE;

    int result = TRUE;
    int fds[2];

    if (pipe(fds) == -1) {
        result = FALSE;
    } else {
        // 保存标准输入输出
        int inFd = dup(STDIN_FILENO);
        int outFd = dup(STDOUT_FILENO);

        pid_t pid = vfork();
        if (pid == -1) {
            result = FALSE;
        } else if (pid == 0) {
            // 子进程重定向输出到管道
            close(fds[0]);
            dup2(fds[1], STDOUT_FILENO);
            close(fds[1]);

            // 使用 system 调用 command -v 检查命令
            char tmp[BUF_SZ];
            sprintf(tmp, "command -v %s", command);
            system(tmp);
            exit(1);
        } else {
            // 父进程等待子进程
            waitpid(pid, NULL, 0);
            // 读取子进程输出
            close(fds[1]);
            dup2(fds[0], STDIN_FILENO);
            close(fds[0]);

            // 若无输出，说明命令不存在
            if (getchar() == EOF) {
                result = FALSE;
            }

            // 恢复标准输入输出
            dup2(inFd, STDIN_FILENO);
            dup2(outFd, STDOUT_FILENO);
        }
    }
}
```

```

    }

    return result;
}

// 获取当前用户名
void getUsername() {
    struct passwd* pwd = getpwuid(getuid());
    strcpy(username, pwd->pw_name);
}

// 获取主机名
void getHostname() {
    gethostname(hostname, BUF_SZ);
}

// 获取当前工作目录
int getCurWorkDir() {
    char* result = getcwd(curPath, BUF_SZ);
    if (result == NULL)
        return ERROR_SYSTEM;
    else return RESULT_NORMAL;
}

// 将命令按空格分割，返回分割后的命令数量
int splitCommands(char command[BUF_SZ]) {
    int num = 0; // 命令计数
    int i, j;
    int len = strlen(command); // 获取命令字符串长度

    for (i = 0, j = 0; i < len; ++i) {
        if (command[i] != ' ') { // 如果不是空格，加入当前命令
            commands[num][j++] = command[i];
        } else {
            if (j != 0) { // 遇到空格且当前命令非空，结束当前命令
                commands[num][j] = '\0'; // 添加字符串结束符
                ++num;
                j = 0;
            }
        }
    }

    if (j != 0) { // 最后一个命令处理
        commands[num][j] = '\0';
        ++num;
    }
}

```

```

    return num; // 返回命令数
}

// 向当前进程发送终止信号（退出）
int callExit() {
    pid_t pid = getpid(); // 获取当前进程 ID
    if (kill(pid, SIGTERM) == -1) // 向自己发送 SIGTERM 信号
        return ERROR_EXIT;
    else return RESULT_NORMAL;
}

// 执行用户命令（带子进程处理）
int callCommand(int commandNum) {
    pid_t pid = fork(); // 创建子进程
    if (pid == -1) {
        return ERROR_FORK; // 创建失败
    } else if (pid == 0) { // 子进程部分
        // 复制标准输入输出文件描述符
        int inFds = dup(STDIN_FILENO);
        int outFds = dup(STDOUT_FILENO);

        int result = callCommandWithPipe(0, commandNum); // 调用管道执行命令

        // 恢复标准输入输出
        dup2(inFds, STDIN_FILENO);
        dup2(outFds, STDOUT_FILENO);
        exit(result); // 退出并返回结果
    } else { // 父进程部分
        int status;
        waitpid(pid, &status, 0); // 等待子进程完成
        return WEXITSTATUS(status); // 返回子进程退出码
    }
}

// 在范围[left, right)内执行命令，支持管道操作
int callCommandWithPipe(int left, int right) {
    if (left >= right) return RESULT_NORMAL; // 无命令直接返回

    int pipeIdx = -1; // 用于标记管道符“|”位置

    // 查找管道符
    for (int i = left; i < right; ++i) {
        if (strcmp(commands[i], COMMAND_PIPE) == 0) {
            pipeIdx = i;
            break;
        }
    }
}

```

```

}

if (pipeIdx == -1) {
    // 没有管道符，直接执行命令（可能有重定向）
    return callCommandWithRedi(left, right);
} else if (pipeIdx + 1 == right) {
    // 管道符后没有命令，参数缺失
    return ERROR_PIPE_MISS_PARAMETER;
}

// 执行带管道的命令
int fds[2]; // 管道文件描述符：fds[0]读端，fds[1]写端
if (pipe(fds) == -1) {
    return ERROR_PIPE; // 创建管道失败
}

int result = RESULT_NORMAL;
pid_t pid = vfork(); // 创建子进程执行左侧命令
if (pid == -1) {
    result = ERROR_FORK; // 创建失败
} else if (pid == 0) {
    // 子进程执行左边命令，将输出重定向到管道写端
    close(fds[0]); // 关闭管道读端
    dup2(fds[1], STDOUT_FILENO); // 标准输出重定向到管道写端
    close(fds[1]);

    result = callCommandWithRedi(left, pipeIdx); // 执行左边命令
    exit(result); // 退出子进程
} else {
    // 父进程等待子进程结束，然后继续执行右边命令
    int status;
    waitpid(pid, &status, 0);
    int exitCode = WEXITSTATUS(status); // 获取子进程退出码

    if (exitCode != RESULT_NORMAL) {
        // 如果子进程出错，从管道中读取错误信息
        char info[4096] = {0};
        char line[BUF_SZ];
        close(fds[1]); // 关闭写端
        dup2(fds[0], STDIN_FILENO); // 将标准输入重定向到管道读端
        close(fds[0]);
        while (fgets(line, BUF_SZ, stdin) != NULL) {
            strcat(info, line); // 拼接错误信息
        }
        printf("%s", info); // 打印错误信息
    }
}

```

```

        result = exitCode;
    } else if (pipeIdx + 1 < right) {
        // 正常执行且还有右边命令，继续递归处理右边部分
        close(fds[1]); // 关闭写端
        dup2(fds[0], STDIN_FILENO); // 标准输入重定向为管道读端
        close(fds[0]);
        result = callCommandWithPipe(pipeIdx + 1, right);
    }
}

return result;
}

int callCommandWithRedi(int left, int right) { // 执行命令区间 [left, right)，无管道，但可能包含重定向
    if (!isCommandExist(commands[left])) { // 命令不存在
        return ERROR_COMMAND;
    }

    /* 检查是否包含重定向 */
    int inNum = 0, outNum = 0;
    char *inFile = NULL, *outFile = NULL;
    int endIdx = right; // 命令真正结束的位置（不包含重定向部分）

    for (int i=left; i<right; ++i) {
        if (strcmp(commands[i], COMMAND_IN) == 0) { // 输入重定向符号 "<"
            ++inNum;
            if (i+1 < right)
                inFile = commands[i+1]; // 获取输入文件名
            else return ERROR_MISS_PARAMETER; // 缺少重定向后面的参数（文件名）

            if (endIdx == right) endIdx = i; // 记录命令部分的终点
        } else if (strcmp(commands[i], COMMAND_OUT) == 0) { // 输出重定向符号 ">"
            ++outNum;
            if (i+1 < right)
                outFile = commands[i+1]; // 获取输出文件名
            else return ERROR_MISS_PARAMETER; // 缺少重定向后面的参数

            if (endIdx == right) endIdx = i;
        }
    }

    /* 输入文件存在性检查 */
    if (inNum == 1) {
        FILE* fp = fopen(inFile, "r");
        if (fp == NULL) // 输入文件不存在
            return ERROR_FILE_NOT_EXIST;
    }
}

```



```
    fclose(fp);
}

/* 检查是否有多个重定向符号 */
if (inNum > 1) {
    return ERROR_MANY_IN; // 输入重定向重复
} else if (outNum > 1) {
    return ERROR_MANY_OUT; // 输出重定向重复
}

int result = RESULT_NORMAL;
pid_t pid = vfork(); // 创建子进程
if (pid == -1) {
    result = ERROR_FORK;
} else if (pid == 0) {
    /* 重定向处理 */
    if (inNum == 1)
        freopen(inFile, "r", stdin); // 重定向 stdin
    if (outNum == 1)
        freopen(outFile, "w", stdout); // 重定向 stdout

    /* 执行命令 */
    char* comm[BUF_SZ];
    for (int i=left; i<endIdx; ++i)
        comm[i] = commands[i];
    comm[endIdx] = NULL;

    execvp(comm[left], comm+left); // 使用 execvp 执行命令
    exit(errno); // 若 exec 失败，返回 errno 作为退出码
} else {
    int status;
    waitpid(pid, &status, 0); // 等待子进程结束
    int err = WEXITSTATUS(status); // 获取退出状态

    if (err) { // 非正常退出，输出红色错误信息
        printf("\e[31;1mError: %s\n\e[0m", strerror(err));
    }
}

return result;
}
```