中山大學 网络空间安全学院
SUN YAT-SEN UNIVERSITY SCHOOL OF CYBER SCIENCE AND TECHNOLOGY

自定义 Shell 解释器

（实验）课程名称：操作系统原理

目录

一　实验目的

(1) 通过设计一个自定义的 Shell 解释器，加深对 Linux 系统的底层系统调用和解释器工作原理的理解。
(2) 综合运用操作系统原理中所学的知识和 Linux 系统实验中锻炼的动手能力完成实验。

二　实验内容

2.1　实验要求

2.1.1　基础要求

(1) 命令帮助信息:类似于 help 命令或 man 命令。该功能提供一个命令，用于显示设计的自定义 shell 解释器支持的所有命令及其用法。用户可以随时调用此命令，以获取对其他命令的简要描述和正确使用方式的指导。

(2) 显示指定目录文件:类似于 ls 命令。该功能提供一个命令，用于列出指定目录下的所有文件和文件夹。用户可以通过这个命令查看当前工作目录中的内容

(3) 切换工作目录:类似于 cd 命令。该功能提供一个命令，允许用户改变当前工作目录，切换到指定的路径。

(4) 复制文件、文件夹:类似于 cp 命令。该功能提供一个命令，允许用户将源文件或目录复制到指定的目标文件或目录中。

(5) 移动文件、文件夹:类似于 mv 命令。该功能提供一个命令，用户可以调用它来对文件或目录重新命名，或者将文件从一个目录移到另一个目录中。

(6) 删除文件、文件夹: 类似于 rm 命令。该功能提供一个命今，用于删除文件或目录，也可以将某个目录及其下属的所有文件及其子目录递归删除。

(7) 显示工作目录:类似于 pwd 命令。该功能提供一个命令，用于显示当前工作目录的路径。用户可以通过调用此命令立刻获取目前所在的工作目录的绝对路径名称。

(8) 显示输入命令历史:类似于 history 命令。该功能提供一个命令，用户可以调用它来显示之前输入的命令历史记录，方便回顾和重用命令。

(9) 显示进程信息:类似于 ps 命令。该功能提供一个命令，用户可以调用它来获取有关系统中运行进程的详细信息，例如进程 ID、状态和资源使用情况里

(10) 显示目录结构:类似于 tree 命令。该功能提供一个命令，以树状结构显示目录的内容。执行命令会列出指定目录下的所有文件,包括子目录里的文件。

(11) 输入输出重定向: 允许用户使用符号将命令的输入或输出重定向到文件,实现更灵活的输入输出控制。

(12) 管道:类似于[|]操作符,该功能允许用户将一个命令的输出作为另一个命令的输入，实现命令之间的数据流传递。

(13) 后台运行程序:类似于[&]操作符,该功能使得某个程序在后台运行允许用户继续输入其他命令而不阻塞 shell。

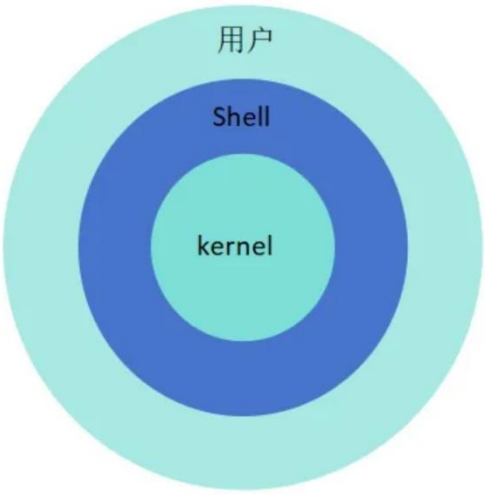
2.1.2　进阶要求

在完成上述基础功能后,学有余力的同学可以进一步了解 shell 解释器的底层设计,尝试在自定义的 shell 解释器中实现更复杂的功能,如外部指令调用等。

三　实验原理

3.1 Shell 原理：

Shell 是一个用户与操作系统内核之间的接口，它接收用户输入的命令并解释执行，从而使用户能够与计算机系统进行交互。Shell 构成了用户与操作系统之间的桥梁，它接受用户的指令、解释这些指令，并将其转化为可以通过系统调用等方式实现的底层代码，最终由操作系统内核执行。如下图所示：



在 linux 中有很多类型的 Shell，不同的 Shell 具备不同的功能，Shell 还决定了脚本中函数的语法。大多数 linux 系统默认的 shell 命令解释器是 bash（/bin/bash），流行的 shell 有 ash、bash、ksh、csh、zsh 等，不同的 shell 都有自己的特点以及用途。在 Linux 操作系统中的 Shell 命令包括 touch、cd、pwd、mkdir 等等。

3.2 Linux 基础：

3.2.1 系统调用：

Linux 内核中设置了一组用于实现系统功能的子程序，称为系统调用。在 Linux 中系统调用是用户空间访问内核的唯一手段，除异常和陷入外，他们是内核唯一的合法入口。在设计一个自定义的 Shell 解释器时，我们使用了如下表所示的系统调用：

1、进程管理和通信

fork()	创建子进程
wait()	等待子进程的结束
exit()	终止进程

2、文件和文件描述符操作

open()	打开文件或创建文件
close()	关闭文件描述符
dup2()	复制文件描述符

3、文件操作相关的系统调用

printf()	格式化输出字符串
pipe()	创建管道，用于进程间通信

4、文件打开方式和权限控制

O_WRONLY	文件打开方式，只写
O_TRUNC	文件打开方式，截断文件
O_CREAT	文件打开方式，如果文件不存在则创建
0666	文件权限，允许所有用户读写

5、执行新程序和程序替换

execl()	执行一个新的程序

6、文件和目录信息获取

stat()	获取文件/目录的信息，例如文件类型和权限
opendir()	打开目录以获取目录流
readdir()	读取目录流中的目录项
closedir()	关闭目录流
basename()	返回路径中的文件名部分

7、文件和目录操作

remove()	删除文件
unlink()	删除文件的链接
rmdir()	删除目录
mkdir()	创建目录
fopen()	打开文件
fclose()	关闭文件
fwrite()	向文件写入数据
fread()	从文件读取数据

8、文件复制

copy_file_to_file()	从一个文件复制到另一个文件

9、递归复制目录：

copy_directory()	递归地复制一个目录及其内容

10、文件和路径操作

access()	检查文件或目录的访问权限
basename()	获取文件名
snprintf()	格式化字符串输出

3.2.2 进程管理：

Shell 主要负责接收用户输入的命令、创建新进程执行等命令，并在必要时进行进程控制。

1、fork() 函数：

‘fork()’函数用于创建一个新的进程，新进程是调用进程的副本，被称为子进程。

调用‘fork()’时，操作系统会复制调用进程的内存空间、寄存器状态和文件描述符等信息，创建一个几乎完全相同的子进程。在父进程中，‘fork()’返回子进程的进程 ID（PID），而在子进程中，返回值为 0。通过检查‘fork()’的返回值，可以在父子进程中执行不同的代码，实现并发执行。

2、exec() 函数族：

‘exec()’函数族用于在当前进程中执行新的程序，替换当前进程的内存映像。

调用‘exec()’函数时，操作系统会加载一个新的程序，替换当前进程的代码、数据和堆栈，从而执行新的程序。新程序继承了调用进程的文件描述符等资源。在 Shell 解释器中，常用于执行用户输入的命令。

3、wait() 和 waitpid() 函数：

‘wait()’和 ‘waitpid()’函数用于等待子进程的结束。

父进程调用‘wait()’或‘waitpid()’后会被阻塞，直到一个子进程退出。此时，操作系统将回收子进程的资源，并返回子进程的退出状态。返回退出子进程的 PID。在 Shell 中，父进程通常需要等待子进程执行完毕，以便获取执行结果。

4、exit() 函数：

‘exit()’函数用于终止当前进程的执行。

调用‘exit()’时，操作系统会释放当前进程的资源，并将退出状态传递给父进程。子进程在执行完任务后可以调用‘exit()’通知父进程任务的完成。

5、管道（pipe）：

管道用于在进程之间建立通信通道，实现进程间的数据传输。

通过调用‘pipe()’创建管道，然后使用‘fork()’创建子进程，父进程和子进程通过管道进行通信。在 Shell 解释器中，管道可用于实现命令的连接，将一个命令的输出作为另一个命令的输入。

6、信号处理：

信号用于通知进程发生的事件，如中断、终止等。

进程可以通过调用 ‘signal()’设置信号处理函数，定义在接收到信号时执行的操作。Shell 解释器需要处理用户输入的信号，如 Ctrl+C 发送的中断信号。

7、进程组和作业控制：

进程组和作业控制用于组织和管理相关联的进程。

进程组是一组相互关联的进程，作业控制用于管理一组相关联的进程。在 Shell 解释器中，可以使用作业控制来控制和管理用户输入的命令。

3.3 语言规范与注意事项：

我们使用 C 语言来实现设计一个自定义的 Shell 解释器，而在 C 语言中有一些需要注意的 C 语言用法与规范：

1、extern 用于声明，而不是定义。声明用于告诉编译器变量或函数在其他文件中定义，实际的定义应该在其他地方。只有 1 个定义的文件，在定义的文件中不要使用 extern。

```
extern int cmdnum;                              //记录历史命令的个数
extern char history[MAX_HISTORY][BUF_SIZE]; //存放历史命令
```

2、宏的定义：在宏定义中使用括号，尤其是在涉及到运算符的表达式中，可以确保宏在展开时不会产生意外的结果。在宏定义中避免使用可能带有副作用的表达式，以免出现意外的错误。避免在宏中改变参数的值。宏定义可以提高代码的灵活性，但过度使用可能导致代码可读性降低。

```
if (!WIFEXITED(status)) ERROR_EXIT;
pid_t pid2 = fork();
```

3、动态内存分配和释放：使用 malloc 分配动态内存后，需要在合适的时候使用 free 释放，以防止内存泄漏。

```
Command *cmds = malloc(MAX_COMMANDS * sizeof(Command));

commandsCount = 1;

if (splitCommands(buf, cmds)) executeCommands(cmds);
free(cmds);
```

4、环境变量处理：处理“$”时，需要判断其后是否为环境变量，并获取环境变量的值。需要注意环境变量名的提取和合理使用 getenv 函数。

```
printf("\033[1;34m%s%s%s\033[0m%s\033[1;36m%s\033[0m $ ", pwd->pw_name, "@", host, ":", path);
```

5、系统调用的错误处理：在代码中使用了一些系统调用，如 pipe、fork、wait、open 等，需要确保对这些系统调用的返回值进行错误处理。

```
if (pid < 0) {
perror("fork");
exit(1);
```

四　设计思路及实现

4.1 整体框架

设计一个命令行解释器（shell）时，我们首要考虑的是建立整体框架。我们最初构思了如图一所示的框架结构，其中包括后台运行（使用“&”），管道操作（使用“|”），以及输入输出重定向（“<”，”>”，“>>”）；也包括了基本命令，例如 ls，cd 等（在我们的 shell 中为 ourls，ourcd 等），并且利用了 execvp()函数来支持对外部命令的调用。

在这一设计中，我们的目标是实现一个灵活且可扩展的命令行解释器。通过这种模块化的方式，我们能够以函数调用的形式逐步实现每个功能，确保每个部分都相对独立，易于调试和维护。

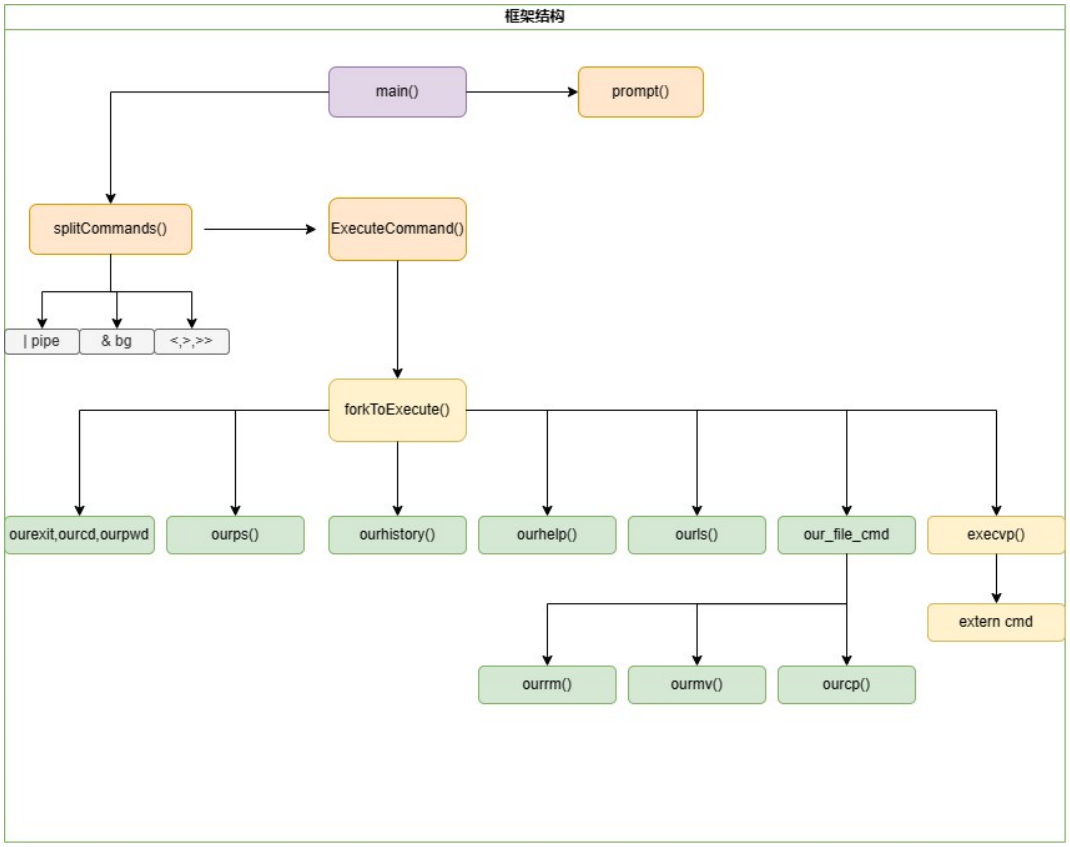


图 1 shell 的框架结构

4.2 Command 类型的结构体

为了更好地处理我们的命令行解释器，我们在头文件 parameter.h 中构建了一个名为 Command 的结构体，如图 2 所示，其中包含了几个关键字段：

1、read,write,overwrite：这三个字符串字段用于记录每条命令是否包含重定向操作，并且记录了重定向的类型（输入或输出），同时还有需要重定向的文件名信息。

2、args：这是一个数组，用于存储命令的参数字符串。每个元素对应命令的一个参数，例如，如果通过 splitCommand()函数检测到命令是"ourcp dest src"，那么 Command 结构体中的 args[0]就是"ourcp"，args[1]就是"dest"，args[2]就是"src"。

3、cmd：这是存储命令名称的字段，用于记录该条命令的名称，例如"ourcp"。

4、cnt：这个字段用于记录整个命令的参数数量，包括命令名本身在内。

5、is_background：这是一个特别添加的字段，用于实现“&”功能，它用于标记一条命令是否需要再后台执行，即是否在命令末尾包含“&”符号。

这个结构体的设计旨在对命令进行清晰的表示，使得解释器能够轻松地获取和操作命令的各个部分，包括参数、重定向信息以及后台执行的需求。通过这种结构化的方式，我们可以更方便地处理和解析用户输入的命令，并在解释器中有效地执行相应的操作。

```c
//指令结构体
typedef struct {
    char *read;          //输入重定向 的文件名
    char *write;         //追加输出重定向 的文件名
    char *overwrite;     //覆盖输出重定向 的文件名
    char **args;         //参数字符串数组—针对于这一条命令
    char *cmd;           //命令名
    int cnt;             //这条命令的参数个数
    //--为了判断每个命令是否需要后台执行--添加一个参数
    int is_background;  //is_background 初始为0，设置为1的时候说明需要后台执行
} Command;
```

图 2 Command 结构体

4.3 执行命令的逻辑

有了上述的整体框架以及基本的 Command 结构体，我们现在来考虑 main 函数中执行命令的逻辑。

首先我们定义各种函数，用于解析命令、处理命令等：

- prompt 函数打印提示符，显示当前工作路径和用户名。
- createCommand 函数用于初始化 Command 结构体。
- freeCommand 函数用于释放 Command 结构体的内存。
- fetchFileName 函数从输入字符串中提取文件名。
- splitCommands 函数解析用户输入的命令，将其拆分为多个子命令，并填充到 commands 数组中。
- forkToExecute 函数根据命令类型执行相应的操作，例如执行内置命令、执行外部命令、重定向输入输出等。
- executeCommands 函数根据命令的数量和类型，选择不同的执行方式，例如单个命令、多个命令使用管道等。

main 函数是程序的入口。它首先打印欢迎信息，然后进入一个无限循环，等待用户输入命令。在循环中，它调用 prompt 函数打印提示符，然后使用 fgets 函数从标准输入读取用户输入的命令。接着调用 saveHistory 函数将命令保存到历史记录中，并创建一个 Command 结构体类型的数组 cmds。然后调用 splitCommands 函数解析用户输入的命令，并根据解析结果调用 executeCommands 函数执行命令。最后释放内存。

```c
while (1) {
    prompt(); //每个while开始都会输出这个最左边那些提示信息
    if (fgets(buf, BUF_SIZE, stdin) != NULL) { //从标准输入-键盘读入-char字符串到buf这个char[]数组中
```

图 3 获取输入的命令

```c
        if (splitCommands(buf, cmds)) executeCommands(cmds);
        free(cmds);
    }
```

图 4 执行命令后退出

4.4 Shell 的 UI 设计

进入和退出页面的信息是 shell 的一个重要部分，能够提供用户友好的交互体验。采用显眼的显示可以增强用户对 shell 界面的关注和识别度。

进入页面的信息：

```
//(1)输入进入时候的提示页面
printf("\033[1;33m-------------------------------------------------------\n");
printf("\033[5;33m|         Welcome to WANG-AN's shell :)        |\n");
printf("\033[1;33m-------------------------------------------------------\n");
```

图 5 进入页面设计

图 6 进入页面效果

退出信息：使用简单的”Good Bye!”。

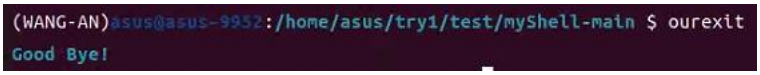


图 7 退出信息效果

prompt()函数：设计了一个独特的提示符，让用户知道他们可以开始输入命令，且区别于 bash。

```
printf("(WANG-AN)");
printf("\033[1;34m%s%s%s\033[0m%s\033[1;36m%s\033[0m $ ", pwd->pw_name, "@", host, ":", path); //彩色显示
```

图 8 提示符设计

4.5 “&后台执行命令”，“|管道命令”和“重定向”命令符号

有了上述的大致框架，我们开始进入对命令的实现部分。

4.5.1 后台执行命令“&”

在处理“&”执行命令的功能时，会引入额外的复杂性。这个功能需要我们考虑多种情况，包括“&”符号出现在输入的不同位置，比如整个输入的末尾或者每条命令的末尾。

在 splitCommand()函数中，我们需要检测输入 buf 是否包含“&”符号。一旦发现这个符号，我们会立即将该命令的 is_background 设置为 1，同时暂时性地增加命令数。如果后续没有再检测到命令，我们会抵消这个暂时性的命令数增加。如图 3、4 所示。

```
commands->is_background = 1; // 设置后台执行标识
buf++;
```

图 9 splitCommand()处理“&”（1）

```
if(tmp->is_background == 1 && waitCommand == 1) //因为waitCommand==1说明连命令名都没遇到
{
    commandsCount--;
```

图 10 splitCommand()处理“&”（2）

在 executeCommand()函数中，处理“&”命令需要一些改进。通常，我们可以在适当的位置调用 forkToCommand()函数来执行命令。然而，由于“&”功能，我们需要在父进程中判断是否需要 fork 出一个子进程并将其挂到后台。这可能需要对代码进行如下改进：

```
//forkToExecute(commands, -1, -1);
if (commands->is_background == 1) {
    pid_t pid = fork();

    if (pid < 0) {
    perror("fork");
    exit(1);
    } else if (pid == 0) {
    // Child process
    forkToExecute(commands, -1, -1);
    printf("\n%d Finished\n", getpid());
    exit(0);
    } else {
    // Parent process
    printf("PID: %d\n", pid);
    }
} else {
    forkToExecute(commands, -1, -1);
}
freeCommand(commands++);
```

图 11 executeCommand()处理“&”

4.5.2 管道命令“|”

实现管道命令也需要一些较为复杂的操作，特别是在处理管道的输入输出时，需要使用”pipe()”函数来实现不同命令间的通信。

在 splitCommand()函数中，检测到管道符号“|”时，需要将 waitCommand 标志设置为 1，这表示后续会有另一条命令需要等待执行（如图 6 所示）。同时，增加命令数，为下一条命令的参数分配空间。这样做相当于刷新了接收命令参数的容器，为下一个命令的参数做好准备，如图 7 所示：

```
case '|' :
    if (waitCommand) ERROR_EMPTY("\033[1;31mError:\033[0m Pipe should be used after a command!\n");

    waitCommand = 1; //继续等待“管道符|后面的那个命令的获取”，所有还是要把waitCommand设置为1
```

图 12 splitCommand()处理“|”（1）

```
*args = malloc(sizeof(char));
*args = 0;                          //NULL结尾
commandsCount++;                    //这个buf中包含的指令数+1,
commands++;                         //commands指针后移一位 -
createCommand(commands);
args = commands->args = malloc(sizeof(char *) * MAX_ARGS);
break;
```

图 13 splitCommand()处理"|"（2）

至于管道命令核心部分的实现，主要位于 executeCommand()函数体内。使用 pipe()函数建立管道，然后通过 fork()创建子进程，使得子进程可以通过管道传输数据。这个过程需要在循环中实现连续的管道操作，确保前一个命令的输出能够成为后一个命令的输入。

```
//--中间的命令需要从另一个管道读取，并写入到当前管道
for (int i = 1; i < (commandsCount - 1); ++i) {
    newPoint = 1 - newPoint;
    pipe(pipes[newPoint]);
    //forkToExecute(commands,(pipes[1-newPoint])[0],(pipes[newPoint])[1]);
```

图 14 executeCommand()处理"|"

```
    }
    //----
    close((pipes[1-newPoint])[0]);
    close((pipes[newPoint])[1]);
    freeCommand(commands++);
```

图 15 释放相关资源

4.5.3 重定向命令

对于重定向命令的实现，以输入重定向"<"为例，在"splitCommand()"函数中，当检测到输入 buf 中包含"<"符号时，需要将当前命令的"commands->read"结构体中用于记录"重定向输入文件名"的变量设置为读取到的参数名。这样可以明确告知命令需要从指定文件读取输入。

```
case '<' :
    if (waitCommand) ERROR_EMPTY("\033[1;31mError:\033[0m I/O redirection should be used after a command!\n")

    buf++;
    while ((*buf != '\n') && isspace(*buf)) ++buf; //跳过所有的空格
    if (fetchFileName(&buf, &(commands->read)) == false) ERROR_IOFILE; //传递此时跳过空格的buf，按道理应该是一个
    commands->cnt += 2; //commands->cnt记录这个command命令的参数的个数+2 （包括这个< 和 这个文件名）
    break;
```

图 16 splitCommand()处理重定向

在 forkToExecute()函数中，如果命令的 commands->read 变量不为空，表明需要进

行输入重定位。此时，我们可以利用 open()函数打开指定文件并获得文件句柄，然后将该文件句柄赋值给 in 参数，最后使用 dup2()函数将标准输入重定向到这个文件句柄，完成输入重定向的操作。最后，关闭文件句柄以清理资源。

```
// (1) 如果有输入重定向，就将0通过dup2函数给到command->read这个文件名的文件描述符句柄
if (command->read) {
    int in = open(command->read, O_RDONLY, 0666);
    if (in < 0) ERROR_OPEN;
    dup2(in, 0);
    close(in);
}
```

图 17 forkToExecute()处理重定向（1）

```
//-否则，如果fd_in>0，就将0给到fd_in文件描述符，也就是这个函数的参数fd_in作为输入重定向略
else if (fd_in > 0) {
    dup2(fd_in, 0);
}
```

图 18 forkToExecute()处理重定向（2）

Fd_in 变量作为 forkToExecute()函数的参数传递，主要用于外部指示是否需要进行文件重定位的信息。这在管道“|”命令的实现中扮演着重要的角色，帮助确保命令的输入可以正确地从文件中获取。

对于输出覆盖重定向“>”和输出追加重定向“>>”，实现思路类似。在 splitCommand()中检测到这些符号时，需要设置相应的命令结构体字段，标明输出重定向的文件名。在 forkToExecute()中根据命令结构体字段的设置，使用 open()打开对应文件并获取文件句柄，然后利用 dup2()将标准输出重定向到这个文件句柄，完成输出重定向的操作。最后，关闭文件句柄以释放资源。

4.6 其他命令的实现

在该部分实现我们的其他命令。

4.6.1 使用系统调用实现的命令

对于”ourexit”、”ourcd”、”ourpwd”这些命令，它们可以直接利用系统调用来实现，因此可以在 main 函数的命令处理部分直接实现相应的逻辑。通常情况下，这些命令是直接调用操作系统提供的相关功能，比如退出 shell、改变当前工作目录或者显示当前工作目录。

ourexit：这个命令可以直接调用操作系统提供的退出函数 exit(0)，表示正常退出。

```
strcmp(command->cmd, "ourexit") == 0) {printf("\033[1;36mGood Bye!\033[0m\n");freeCommand(command);exit(0);}
```

图 19 ourexit 实现

ourcd：通过调用"chdir()"系统调用可以改变当前工作目录。需要解析用户输入的路径参数，然后使用"chdir()"将当前工作目录切换到指定路径。

```
    path = command->args[1];
if (chdir(path) < 0) {
```

图 20 ourcd 实现

ourpwd：这个命令可以使用"getcwd()"系统调用获取当前工作目录，并将其输出到 shell 界面。

```
if (getcwd(cwd, sizeof(cwd)) != NULL) {
    printf("Current working directory: %s\n", cwd);
} else {
```

图 21 ourpwd 实现

4.6.2 文件操作等命令

对于 ps,history,help,ls,rm,cp,mv 等命令的实现过程，我们考虑单独使用函数或者函数文件进行分工实现，具体实现在.c 文件中有对关键步骤的详细阐述。

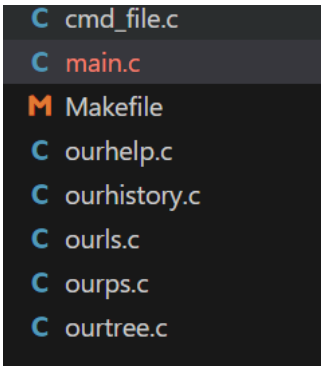


图 22 用于实现其他命令的函数

4.7 外部命令处理

这一部分主要是在 forkToExecute()函数体内部，如果我们在前面的所有 if-else 中都没有匹配到命令名的字符串，那么，我们就要考虑使用外部命令调用方式了，这里的 execvp()在实验原理部分有详细阐述，这里也有相应的 error 异常处理。

```
if((status = execvp(command->cmd, command->args)) < 0) ERROR_EXECUTE(command->cmd);
```

图 23 外部命令实现

4.8 异常情况处理

经过上述步骤，我们已经大致完成了一个基本的 Shell。除此之外，处理异常情况是非常重要的，因为用户输入可能包含各种意外情况或错误。

因此，我们在 parameter.h 文件中定义了各种异常情况处理的宏：

```
//这里的宏都是直接替换成了指定的"语句序列"依次执行，比如第一个ERROR_EMPTY (errorStr) 就是接收一个参数errorStr，然后执行
#define ERROR_EMPTY(errorStr) do {fprintf(stderr,"%sPlease check and try again :D\n",errorStr); return false;}
#define ERROR_IOFILE do {fprintf(stderr,"\033[1;31mError:\033[0m I/O file format error! \nPlease check and try
#define ERROR_STR do {fprintf(stderr,"\033[1;31mError:\033[0m Your str miss \' or \" ! \nPlease check and try a
#define ERROR_ENV do {fprintf(stderr,"\033[1;31mError:\033[0m Your variable miss ! \nPlease check and try again
#define ERROR_FORK do {fprintf(stderr,"\033[1;31mError:\033[0m Failed to fork process!"); return; } while(0)
#define ERROR_OPEN do {fprintf(stderr,"\033[1;31mError:\033[0m Failed to open file! \nPlease check and try agai
#define ERROR_EXECUTE(errorCmd) do {fprintf(stderr, "\033[1;31mError:\033[0m Failed to execute cmd %s!\n", erro
#define ERROR_EXIT do {fprintf(stderr, "\033[1;31mError:\033[0m Failed to exit with status %d!\n", WEXITSTATUS(
#define ERROR_FMT(errorCmd) do {fprintf(stderr, "\033[1;31mError:\033[0m Please read help doc and check the usa
```

图 24 异常情况处理

此外，在设计实现命令的相关代码时，我们也有尽可能地考虑输入的不确定性问题，以保证程序的稳定性，比如切换工作目录时，输入的 path 路径可能不存在。

```
if (chdir(path) < 0) {
    char error[MAX_CHAR_SIZE];
    sprintf(error, "No such dir or file: %s !\n", path);
```

图 25 函数中的异常处理

经过上述设计思路及步骤，我们大致完成了我们的 shell，我们的模块化设计使得该 shell 灵活且可扩展，每个部分都相对独立，易于调试和维护。完成后的项目文件结构如下图所示：

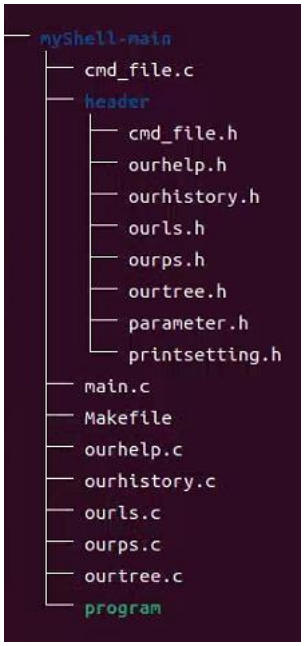


图 26 完整项目文件结构

五 实验运行截图

下面从用户的使用顺序角度来进行实验成功的截图展示。

- 首先运行项目中编译成功的可执行文件，此时启动 shell 终端，显示提示信息：

```
user@VM:~/桌面/shell$ ./program
-----------------------------------------------
|           Welcome to WANG-AN's shell :)      |
-----------------------------------------------
(WANG-AN)user@VM:/home/user/桌面/shell $
```

图 5-1 shell 的欢迎界面

- 输入命令 ourhelp 显示帮助内容：

```
(WANG-AN)user@VM:/home/user/桌面/shell $ ourhelp
欢迎查看网安自研shell的用户手册！

*************************************************************************************
1. ourhelp
命令示例：   ourhelp
命令作用：   输出用户帮助手册
使用示例：   ourhelp
参数个数：   无参数

2. ourls
命令示例：   ourls
命令示例：   ourls [目录]
命令示例：   ourls [选项] [目录]
命令作用：   显示指定工作目录下之内容，[OPTION]可为-l（输出详细信息），[FILE]为目
标目录，无参数则代表当前目录，..为上级目录
使用示例：   ourls
使用示例：   ourls -l
使用示例：   ourls /dir1
使用示例：   ourls -l /dir1
使用示例：   ourls ..
参数个数：   无参数或1个参数或2个参数

3. ourcd
命令示例：   ourcd
命令示例：   ourcd [目录]
命令作用：   跳转到目标目录，无参数则默认为家目录，有参数则改变当前目录为参数内容
，..号代表回到之前的目录
使用示例：   ourcd
使用示例：   ourcd /dir1
使用示例：   ourcd ..
参数个数：   无参数或1个参数
```

图 5-2 ourhelp 命令的帮助信息（1）

```
4. ourcp
命令示例:   ourcp [源文件]... [目标文件]
命令作用:   将源文件的内容复制到目标文件，可以复制文件到文件，文件夹到文件夹，多个文件到文件夹
使用示例:   ourcp /file1 /file2
使用示例:   ourcp /dir1 /dir2
使用示例:   ourcp /dir1 /file1 /file2 /file3 /file4
参数个数:   2个或以上参数

5. ourmv
命令示例:   ourmv [源文件]...  [目标文件]
命令作用:   将源文件的内容移动到目标文件，可以移动文件到文件，文件夹到文件夹，多个文件到文件夹
使用示例:   ourmv /file1 /file2
使用示例:   ourmv /dir1 /dir2
使用示例:   ourmv /dir1 /file1 /file2 /file3 /file4
参数个数:   2个或以上参数

6. ourrm
命令示例:   ourrm [文件]...
命令示例:   ourrm [选项] [目录]
命令作用:   删除文件或目录，当选项为-r时是删除目录
使用示例:   ourrm /file1
使用示例:   ourrm /file1 /file2
使用示例:   ourrm -r /dir1
参数个数:   1个或以上参数
```

图 5-3 ourhelp 命令的帮助信息（2）

```
7. ourpwd
命令示例:   ourpwd
命令作用:   显示当前所在工作目录的全路径
使用示例:   ourpwd
参数个数:   无参数

8. ourhistory
命令示例:   ourhistory
命令示例:   ourhistory [数字]
命令作用:   查看历史命令，不加参数则是输出全部历史命令，增加参数[数字]则输出最后执行的num条命令
使用示例:   ourhistory
使用示例:   ourhistory 42
参数个数:   无参数或1个参数

9. ourps
命令示例:   ourps
命令示例:   ourps [选项] [PID]
命令作用:   显示当前进程的状态，不加选项则是输出全部进程，增加选项-p则显示查询的单个进程
使用示例:   ourps
使用示例:   ourps -p 123
参数个数:   无参数或2个参数

10. ourtree
命令示例:   ourtree
命令示例:   ourtree [目录]
命令作用:   展示目录的文件结构，无参数则默认为当前目录，有参数则将参数内容作为目标目录
使用示例:   ourtree
使用示例:   ourtree /dir1
参数个数:   无参数或1个参数
```

图 5-4 ourhelp 命令的帮助信息（3）

```
11. 重定向
命令作用:   命令通常从标准输入读取输入和标准输出写入输出，使用重定向可以改变输入和输出的地方

命令示例:   [命令] < [文件]
命令作用:   将输入重定向到 [文件]
使用示例:   ourtree < /file1

命令示例:   [命令] > [文件]
命令作用:   将输出重定向到 [文件]
使用示例:   ourhelp > /file1

命令示例:   [命令] >> [文件]
命令作用:   将输出以追加的方式重定向到 [文件]
使用示例:   ourls >> /file1

12. 管道 |
命令示例:   [命令] | [命令]命令作用:   将一个命令的输出作为另一个命令的输入，实现命令之间的数据流传递
使用示例:   ls | wc -l
13. 后台运行程序 &
命令示例:   [命令] &
命令作用:   使得某个程序在后台运行，允许用户继续输入其他命令而不阻塞 Shell
使用示例:   sleep 3 &
14. ourexit
命令示例:   ourexit
命令作用:   退出Shell
使用示例:   ourexit
```

图 5-5 ourhelp 命令的帮助信息（4）

- ourls 命令：
  不带参数显示当前目录下的文件；
  带参数可以显示详细信息；
  带目标参数可以查看目标目录下的文件。

```
(WANG-AN)user@VM:/home/user/桌面/shell $ ourcd ..
(WANG-AN)user@VM:/home/user/桌面 $ ourls
shell
test
(WANG-AN)user@VM:/home/user/桌面 $ ourls -l
drwxrwxr-x  3 user user  4096 Dec 17 14:39 shell
drwxrwxr-x  3 user user  4096 Dec 17 14:43 test
(WANG-AN)user@VM:/home/user/桌面 $ ourls test
2
3
t
1
(WANG-AN)user@VM:/home/user/桌面 $ ourls -l test
-rw-rw-r--  1 user user     0 Dec 17 14:43 2
-rw-rw-r--  1 user user     0 Dec 17 14:43 3
drwxrwxr-x  3 user user  4096 Dec 17 14:44 t
-rw-rw-r--  1 user user     0 Dec 17 14:43 1
```

图 5-6 ourls 命令

- ourcd 命令
  可以使用".."跳转到上一目录；
  使用参数跳转到对应目录；
  使用"/"可以跳转到根目录；
  不加参数跳转至用户目录。

```
(WANG-AN)user@VM:/home/user/桌面/shell $ ourcd ..
(WANG-AN)user@VM:/home/user/桌面 $ ourcd test
(WANG-AN)user@VM:/home/user/桌面/test $ ourcd /
(WANG-AN)user@VM:/ $ ourcd
(WANG-AN)user@VM:/home/user $
```

图 5-7 ourcd 命令

- ourcp 命令
  复制文件。

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourcp t/test.txt .
(WANG-AN)user@VM:/home/user/桌面/test $ ourls
test.txt
2
3
t
1
```

图 5-8 ourcp 命令

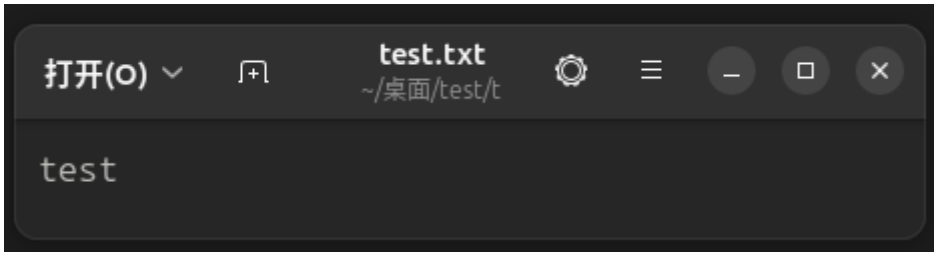


图 5-9 验证复制的文件内容是否正确

由上图可见内容是正确的。

```
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  2  3  many  t  test.txt
(WANG-AN)user@VM:/home/user/桌面/test $ ourcp 1 2 3 many
(WANG-AN)user@VM:/home/user/桌面/test $ ls many
1  2  3
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  2  3  many  t  test.txt
```

图 5-10 复制多个文件至文件夹

- ourmv 命令
  移动单个文件：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ls
m  t
(WANG-AN)user@VM:/home/user/桌面/test $ ls m
1  2  3
(WANG-AN)user@VM:/home/user/桌面/test $ ourmv m/1 .
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  m  t
(WANG-AN)user@VM:/home/user/桌面/test $ ls m
2  3
```

图 5-11 移动单个文件

移动文件夹至文件夹：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ls m
2  3
(WANG-AN)user@VM:/home/user/桌面/test $ ourmv m ..
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  t
(WANG-AN)user@VM:/home/user/桌面/test $ ls ..
m  shell  t  test
(WANG-AN)user@VM:/home/user/桌面/test $ ls ../m
2  3
```

图 5-12 移动文件夹至文件夹

移动多个文件至文件夹：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  2  3  m  t
(WANG-AN)user@VM:/home/user/桌面/test $ ourmv 1 2 3 m
(WANG-AN)user@VM:/home/user/桌面/test $ ls
m  t
(WANG-AN)user@VM:/home/user/桌面/test $ ls m
1  2  3
```

图 5-13 移动多个文件至文件夹

- ourrm 命令

  删除单个文件：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  2  3  many  t  test.txt
(WANG-AN)user@VM:/home/user/桌面/test $ ourrm test.txt
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  2  3  many  t
```

图 5-14 删除单个文件

删除目录：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourrm -r many
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  2  3  t
```

图 5-15 删除目录

- ourpwd 命令：
  显示当前工作目录：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourpwd
Current working directory: /home/user/桌面/test
```

图 5-16 显示当前工作目录

- ourhistory 命令
  不带参数显示所有的命令：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourhistory
1  ourcd ../test
2  ls
3  ourcd t
4  ls
5  cd ..
6  ourcd ..
7  ourcp t/test.txt .
8  ourls
9  ls
10  ourcp 1 2 3 many
11  ls many
12  ls
13  ourrm many
14  ls many
15  ls
16  ourrm test.txt
17  ls
18  ourrm -r many
19  ls
20  mkdir
21  mkdir m
22  ls
23  ourmv 1 2 3 m
24  ls
25  ls m
26  ourmv m/1 .
27  ls
28  ls m
29  ourmv m ..
30  ls
```

图 5-17 不带参数的 ourhistory 命令

带 num 参数显示最近 num 条命令：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourhistory 10
29  ourmv m ..
30  ls
31  ls ..
32  ls ../m
33
34
35  ourpwd
36
37  ourhistory
38  ourhistory 10
```

图 5-18 带 num 参数的 ourhistory 命令

- ourps 命令

不带参数查看所有的进程状态：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourps
PID      Command         State      User Mode Time       Kernel Mode Time
-------------------------------------------------------------------------
1        (systemd)       S          144                  195
2        (kthreadd)      S          0                    3
3        (rcu_gp)        I          0                    0
4        (rcu_par_gp)    I          0                    0
5        (slub_flushwq)  I          0                    0
6        (netns)         I          0                    0
11       (mm_percpu_wq)  I          0                    0
12       (rcu_tasks_kthread) I          0                    0
13       (rcu_tasks_rude_kthread) I          0                    0
14       (rcu_tasks_trace_kthread) I          0                    0
15       (ksoftirqd/0)   S          1                    32
16       (rcu_preempt)   I          97                   141
17       (migration/0)   S          0                    5
18       (idle_inject/0) S          0                    0
19       (cpuhp/0)       S          0                    0
20       (cpuhp/1)       S          0                    0
21       (idle_inject/1) S          0                    0
22       (migration/1)   S          63                   0
23       (ksoftirqd/1)   S          5                    80
26       (kdevtmpfs)     S          0                    0
```

图 5-19 不带参数的 ourps

带选项可以查看单个进程：

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourps -p 14645
PID      Command         State      User Mode Time       Kernel Mode Time
-------------------------------------------------------------------------
14645    (program)       R          1                    0
```

图 5-20 带选项的 ourps

- ourtree 命令
  查看文件树：

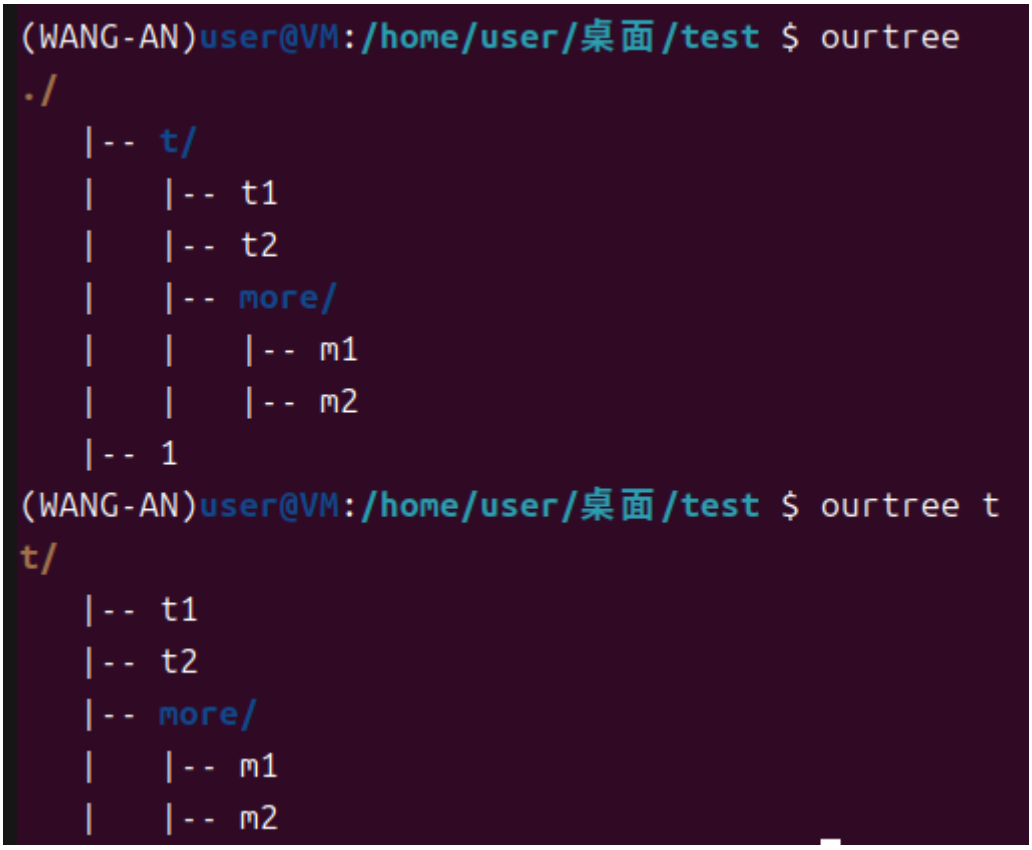


图 5-21 ourtree 命令查看文件树

- 重定向

图 5-22 重定向符号使用

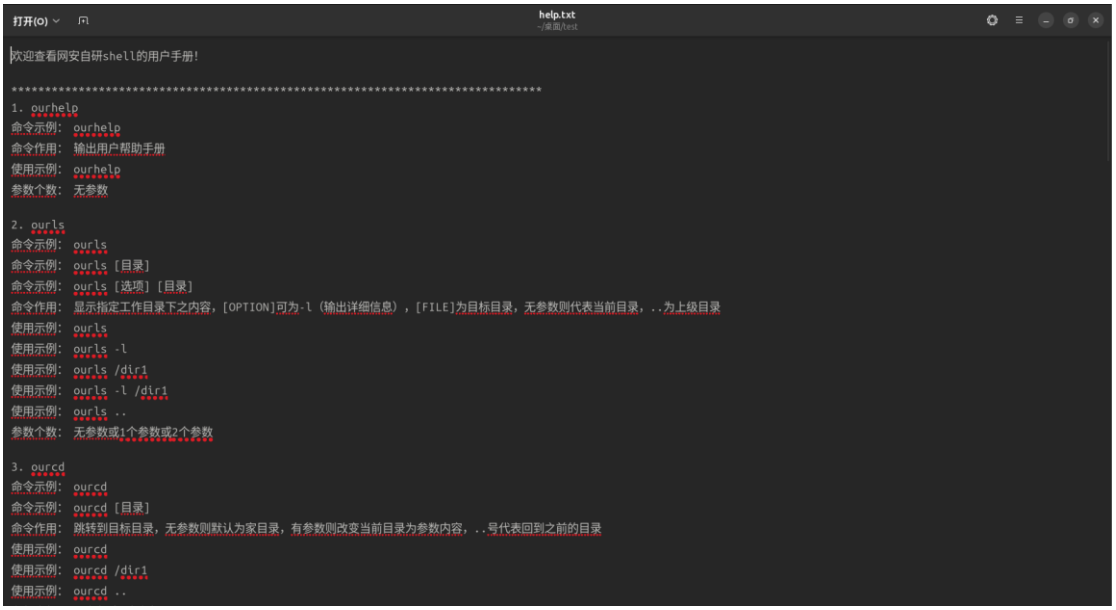


图 5-23 重定向符号使用的验证

- 管道

```
(WANG-AN)user@VM:/home/user/桌面/test $ ourls | wc -l
3
(WANG-AN)user@VM:/home/user/桌面/test $ ourls
10.txt
t
1
```

图 5-24 管道符号的使用示例

- 后台运行程序&
  输入 sleep 3 &直接转后台运行，shell 没有阻塞

```
(WANG-AN)user@VM:/home/user/桌面/test $ sleep 3 &
PID: 15215
(WANG-AN)user@VM:/home/user/桌面/test $
```

图 5-25 “&”的使用示例

- ourexit 命令
  退出 shell，并发出提示：

```
(WANG-AN)user@VM:/home/user $ ourexit
Good Bye!
user@VM:~/桌面/shell$
```

图 5-26 退出界面信息

- 外部命令的实现

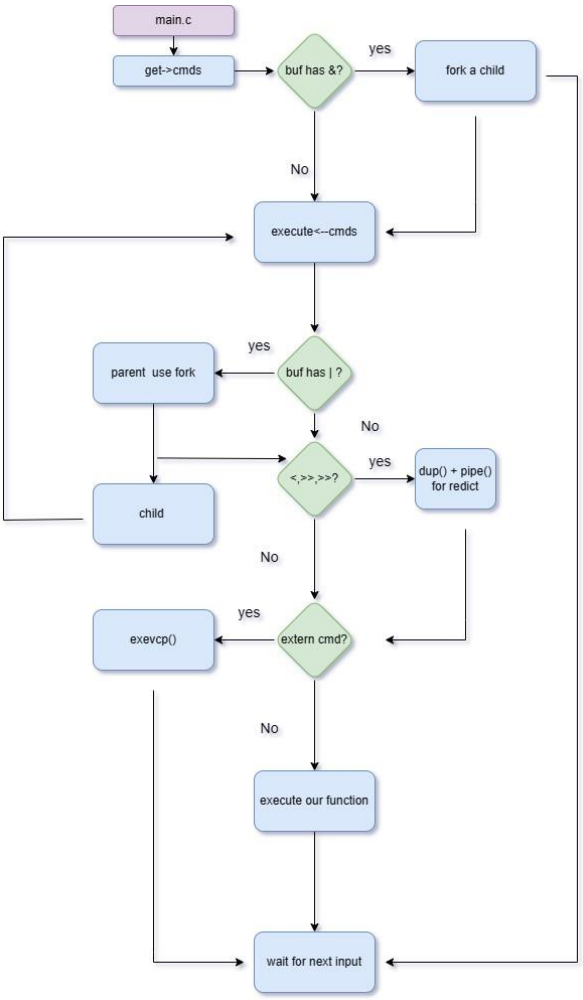
```
(WANG-AN)user@VM:/home/user/桌面/test $ ls
1  10.txt  help.txt  t
(WANG-AN)user@VM:/home/user/桌面/test $ ps
    PID TTY          TIME CMD
   2958 pts/0    00:00:00 bash
   3150 pts/0    00:00:00 program
   3152 pts/0    00:00:00 ps
```

图 5-27 外部命令使用示例（1）

```
(WANG-AN)user@VM:/home/user/桌面/test $ ls | wc -l
4
```

图 5-28 外部命令使用示例（2）

六 实验流程图



- 输入接收和解析：程序从主入口开始，在接收到用户输入的指令后，首先进行对输入内容的解析和分析，确定是否存在后台执行符号“&”、管道符号“|”或者重定向符号等特殊操作符。
- 后台执行处理：在检测到命令需要后台执行时，程序会创建一个子进程来执行该命令，以确保父进程不被阻塞。这使得用户可以继续输入其他指令而不必等待当前指令的执行完成。
- 管道命令处理：当程序检测到管道符号“|”时，它会创建两个子进程。首先，父进程会执行第一个命令并将输出传输给第二个命令的输入，第二个子进程则会接收父进程传来的输入，并执行第二个命令。
- 重定向处理：如果遇到重定向符号，程序会利用管道函数和 dup 函数来修改标准输入输出的位置，以达到重定向输入输出的目的。这使得命令可以读取自定义的输入或者将输出写入到指定文件中。
- 内建命令和外部命令执行：在没有匹配到特殊操作符的情况下，程序会检查命令是否是内建命令，如 ourcd、ourexit 等，如果是则直接执行。若不是内建命令，则通

过 execvp()函数调用外部命令来执行。

- 状态迭代：执行完当前命令后，程序会回到等待状态，继续等待用户的下一条输入指令，并按照上述流程进行处理。

七　实验总结

我们通过实验成功实现了我们自定义的 Shell 解释器。大致实验流程如下：

1、设计整体框架，用模块化的方式进行实现

2、分模块进行代码实现

3、将代码整合，进行自定义 shell 的测试与维护

通过上述基本步骤，我们实现了一个灵活且可扩展的命令行解释器，完美实现了基本功能要求及进阶功能要求。

通过自定义 Shell 解释器，我们对 Linux 系统底层的系统调用有了更深入的理解，知道如何利用这些调用来构建一个完整的解释器。除此之外，我们也自己实现了使用系统调用之外的其他命令，加深了对解释器工作原理的理解。

通过本次实验，我们体会到了在设计大型软件项目中框架图的意义，框架图能指导大项目的实验步骤，并且能实现模块化设计。这种方式能够让团队分解问题、逐步实现功能，同时保持灵活性和可扩展性，这对一个健壮的大项目是至关重要的。

通过实现一个解释器，我们有机会将课堂上学到的操作系统原理应用到实际中。例如，理解进程管理、内存管理、文件系统等知识，并将其直接应用到 Shell 解释器的设计中。

总的来说，这个实验不仅仅是关于构建一个自定义的 Shell 解释器，还涉及到了许多实践和理论结合的内容。这种综合性的项目对于深入理解操作系统和 Linux 系统以及提升实际编程能力都是非常有益的。

分工如下：