《Linux 程序设计》

课程设计报告书

队长	成员
	Banban
60%	40%
指导教师	

2023 年 5 月 29 日

目录

一、 实验设计目的

《Linux 课程设计》是在完成理论课程学习之后安排的综合实践训练，本实验旨在通过设计和实现一个类似于 Linux shell 的程序，帮助我们深入理解操作系统和 C 语言编程，并提升其命令解析、程序加载和输出重定向等技能。

同时，通过这个实验掌握 Linux 下 shell 编程的基本原理和技术，包括命令解析、程序加载和重定向等。培养独立思考和解决问题的能力，在设计和开发过程中逐步发展自己的创造性和创新精神。增强代码编写和调试能力，通过实践不断提高自己的编程技能和开发效率。

二、 实验设计的功能及模块划分

本实验是一个基于 Linux 的 Shell 程序，它可以执行命令及其参数，支持管道操作、输出重定向和输入重定向，能够让用户通过命令行进行各种操作。此外，该程序还能处理信号，使得用户可以在操作过程中及时获得系统的反馈信息，从而更好地掌握程序运行状态，提高操作效率。

本实验的模块划分有以下几个部分：

1. main 函数：负责读取用户输入并调用 do_parse 函数对用户输入的命令进行解析执行；
2. do_parse 函数：对用户输入的命令进行解析，并根据命令类型调用相应的处理函数（如执行命令、管道操作、输出重定向、输入重定向）；
3. do_exe 函数：执行命令及其参数；
4. do_pipe 函数：管道操作；
5. do_redirect_input 函数：输入重定向；
6. do_redirect_output 函数：输出重定向；
7. sig_handler 函数：处理信号；
8. err_exit 函数和 err_command 函数：提供错误提示信息。

三、 实验设计的人员组成及任务划分

- **郝娜娜：队长**

  负责项目需求分析和概要设计，建立程序基础框架；

  编写项目文档，包括设计文档和使用手册；

  实现程序的基本功能；

- **徐龙：组员**

  负责程序体验优化工作；

  完善、添加、实现程序的附加功能；

  测试和调试程序，确保程序质量和稳定性。

四、 程序设计与实现

4.1 系统结构与模块划分

基于 Linux 系统的简单 Shell 程序，它能够接收用户在终端输入的命令，并对其进行解析并执行。程序采用 C 语言编写，使用了系统调用库函数和进程相关的函数。通过 sigaction 函数注册了三个信号处理函数分别用来处理 SIGINT、SIGTSTP 和 SIGQUIT 信号，当用户在终端输入 ctrl+c 或 ctrl+z 或 ctrl+\ 时，会触发相应的信号处理函数。

该程序还可以解析用户输入的命令并进行相应的操作，支持常见的命令、管道和重定向等功能。程序主要分为定义常量和函数、执行命令及其参数 do_exe 函数、管道操作 do_pipe 函数、输入和输出重定向操作函数、解析用户输入命令 do_parse 函数以及主函数 main 等几个部分。整体代码结构清晰简洁，实现了基本的 shell 功能。

4.2 main 函数的实现

这段代码实现了一个简单的命令行解释器(shell)，工作流程如下：定义信号处理函数 sig_handler，用于处理接收到的 SIGINT、SIGTSTP 和 SIGQUIT 信号。注册 SIGINT、SIGTSTP 和 SIGQUIT 信号处理函数，以便在程序运行过程中能够捕获并处理这些信号。进入无限循环，在每次循环中读取用户输入的命令。对用户输入的命令进行解析，并执行相应的操作。其中，

如果输入的是"exit"或"q"，则退出程序，否则继续下一轮循环。在每次命令执行完毕后，使用 waitpid 函数回收已经终止的子进程资源，避免出现僵尸进程。

代码实现

```c
int main(void) {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = sig_handler;    // 设置信号处理函数为sig_handler
    sigaction(SIGINT, &sa, NULL);   // 注册SIGINT 信号处理函数
    sigaction(SIGTSTP, &sa, NULL);  // 注册SIGTSTP 信号处理函数
    sigaction(SIGQUIT, &sa, NULL);
    char buf[1024] = {};  // 定义存储用户输入的缓冲区
    int a;
    while (1) {  // 循环读取用户输入
        printf("\033[33mmy shell# \033[0m");
        fflush(stdout);                  // 刷新标准输出缓冲区
        memset(buf, 0x00, sizeof(buf));  // 清空buf 缓冲区

        if (read(STDIN_FILENO, buf, sizeof(buf)) == -1) {
            // perror("read");
            fflush(stdout);  // 刷新标准输出缓冲区
            continue;
        }
        if (*buf == 0) {  //  handle: ctrl + d
            err_exit();
            continue;
        }
        char *p = buf + strlen(buf) - 1;  // 去掉字符串末尾的换行符
        if (*p == '\n') *p = '\0';
        // 如果输入的是"exit"或"q"，则退出程序
        if (strcmp(buf, "exit") == 0 || strcmp(buf, "q") == 0) break;
        do_parse(buf);  // 对用户输入的命令进行解析并执行

        // 循环回收所有已经终止的子进程资源，避免出现僵尸进程
        while (waitpid(-1, NULL, WNOHANG) > 0);
    }
    return 0;
}
```

4.3 命令解析功能的实现

命令解析的功能具体是根据 do_parse 函数实现的，通过终端输入命令，解析命令中是否包含管道符号、重定向符号等特殊符号来执行相应的操作。

do_parse 函数通过遍历传入的字符串参数 buf，将其中的命令及其参数存储在一个字符串数组 argv 中。如果发现管道符号'|'，则将管道符号后面的命令及其参数解析出来，并调用 do_pipe 函数执行管道操作；如果发现输出重定向符号'>'，则将符号后面的文件名解析出来，并调用 do_redirect_output 函数将命令的输出重定向到指定文件；如果发现输入重定向符号'<'，则将符号后面的文件名解析出来，并调用 do_redirect_input 函数将命令的输入从指定文件读取。最后，如果没有发现任何特殊符号，则调用 do_exe 函数执行命令。

代码实现

```
1. void do_parse(char *buf) {
2.     char *argv[MAX_ARGV] = {};
3.     int argc = 0;
4.     char *args[MAX_ARGV] = {};  // 用于存储管道或重定向符号后面的命令及其参数
5.     int args_count = 0;
6.
7.     // 解析命令行参数
8.     while (*buf != '\0') {
9.         if (!isspace(*buf)) {
10.            argv[argc++] = buf;
11.            while (!isspace(*buf) && *buf != '\0' && *buf != '|'
12.                    && *buf != '<' && *buf != '>') {
13.                buf++;
14.            }
15.            if (*buf == '\0') break;
16.            if (*buf == '|') {  // 解析管道符号后面的命令及其参数
17.                *buf++ = '\0';
18.                while (isspace(*buf)) buf++;  // 跳过管道符号后面的空格
19.                while (*buf != '\0' && *buf != '|' && *buf != '<' && *buf != '>') {
20.                    args[args_count++] = buf;
21.                    while (!isspace(*buf) && *buf != '\0' && *buf != '|'
22.                            && *buf != '<' && *buf != '>') {
23.                        buf++;
24.                    }
25.                    if (*buf == '\0') break;
26.                    if (*buf == '|' || *buf == '<' || *buf == '>') {
27.                        *buf++ = '\0';
```

```
                        break;
                    }
                    *buf++ = '\0';
                }
                do_pipe(argv, argc, args, args_count);  // 执行管道操作
                return;
            } else if (*buf == '>') {
                // 解析输出重定向符号后面的文件名
                *buf++ = '\0';
                while (isspace(*buf)) buf++;  // 跳过输出重定向符号后面的空格
                if (*buf != '\0') {
                    do_redirect_output(argv, argc, buf);
                    return;
                } else {
                    printf("Error: missing destination file after '>'\n");
                    return;
                }
            } else if (*buf == '<') {
                // 解析输入重定向符号后面的文件名
                *buf++ = '\0';
                while (isspace(*buf)) buf++;  // 跳过输入重定向符号后面的空格
                if (*buf != '\0') {
                    do_redirect_input(argv, argc, buf);
                    return;
                } else {
                    printf("Error: missing source file after '<'\n");
                    return;
                }
            } else {
                *buf++ = '\0';
            }
        } else {
            buf++;
        }
    }

    if (argc == 0) return; // 没有输入任何命令
    do_exe(argv[0], argv);
}
```

4.4 命令间通信的实现

do_pipe 函数实现了两个命令之间的管道。函数通过使用 pipe 系统调用创建一个管道，然后 fork 出两个子进程。第一个子进程执行 argv1 中的指令，它关闭了管道的读取端（fd[0]），将管道的写入端（fd[1]）复制到标准输出（1），然后关闭原来的写入端。接着，它使用 execvp 系统调用来执行该命令。第二个子进程执行 argv2 中的指令，它关闭了管道的写入端（fd[1]），将管道的读取端（fd[0]）复制到标准输入（0），然后关闭原来的读取端。接着，它使用 execvp 系统调用来执行该命令。

父进程等待两个子进程结束，然后恢复原来保存在 stdout_copy 中的标准输出文件描述符。如果执行过程中出现错误，使用 perror 函数报告错误信息。

代码实现

```
1. void do_pipe(char **argv1, int argc1, char **argv2, int argc2) {
2.     int fd[2];
3.     pid_t pid1, pid2;
4.     int stdout_copy = dup(STDOUT_FILENO);  // 保存标准输出的文件描述符
5.
6.     if (pipe(fd) < 0) {
7.         perror("pipe error");
8.         return;
9.     }
10.
11.    if ((pid1 = fork()) < 0) {
12.        perror("fork error");
13.        return;
14.    } else if (pid1 == 0) {  // 子进程1，执行管道左边的命令
15.        close(fd[0]);    // 关闭读端
16.        dup2(fd[1], 1);  // 标准输出 定向到 写端
17.        close(fd[1]);    // 关闭写端
18.
19.        if (execvp(argv1[0], argv1) < 0) {  // 执行命令
20.            perror("exec error");
21.            exit(1);
22.        }
23.    } else {
24.        if ((pid2 = fork()) < 0) {
25.            perror("fork error");
26.            return;
27.        } else if (pid2 == 0) {  // 子进程2，执行管道右边的命令
```

```
            close(fd[1]);    // 关闭写端
            dup2(fd[0], 0);  // 标准输入 定向到 读端
            close(fd[0]);    // 关闭读端

            if (execvp(argv2[0], argv2) < 0) {  // 执行
                perror("exec error");
                exit(1);
            }
        } else {
            close(fd[0]);
            close(fd[1]);

            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
        }
    }
    dup2(stdout_copy, STDOUT_FILENO);  // 恢复标准输出的文件描述符
    close(stdout_copy);
}
```

4.5 输入输出重定向的实现

我们在编写输入输出重定向时，使用了 do_redirect_input 和 do_redirect_output 两个函数可以实现指定功能，即将命令的输入从文件中读取，或者将命令的输出写入文件中，方便用户进行批量处理或者记录命令的输出。

do_redirect_input 函数实现了将标准输入从文件读取的功能。函数使用 open 系统调用打开指定的文件，然后 fork 出一个子进程，在子进程中将标准输入重定向到该文件，然后执行由 argv 参数指定的命令及其参数。如果执行过程中出现错误，则报告错误信息。

do_redirect_output 函数实现了将标准输出写入文件的功能。函数使用 open 系统调用打开指定的文件，如果文件不存在则创建文件，如果文件已经存在则清空文件内容。然后 fork 出一个子进程，在子进程中将标准输出重定向到该文件，然后执行由 argv 参数指定的命令及其参数。如果执行过程中出现错误，则报告错误信息。

在两个函数中，父进程等待子进程结束。如果子进程创建失败，则报告错误信息。

代码实现

```
1. void do_redirect_input(char *argv[], int argc, char *filename) {
```

```
    int fd;
    fd = open(filename, O_RDONLY);
    if (fd < 0) {
        printf("Failed to open file %s for reading\n", filename);
        return;
    }
    pid_t pid = fork();
    if (pid == 0) {
        dup2(fd, STDIN_FILENO);  // 将标准输入 重定向到 文件
        close(fd);
        if (execvp(argv[0], argv) < 0) {  // 执行
            perror("exec error");
            exit(1);
        }
        printf("Unknown command: %s\n", argv[0]);
        exit(1);
    } else if (pid < 0) {
        printf("Failed to create child process\n");
        return;
    } else {
        wait(NULL);
    }
}
```

```
void do_redirect_output(char *argv[], int argc, char *filename) {
    int fd;
    // 打开文件，如果文件不存在，则创建该文件；如果文件存在，则清空文件内容
    fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (fd < 0) {
        printf("Failed to open file %s for writing\n", filename);
        return;
    }

    pid_t pid = fork();
    if (pid == 0) {
        dup2(fd, STDOUT_FILENO);  // 将标准输出 重定向到 文件
        close(fd);

        if (execvp(argv[0], argv) < 0) {  // 执行
            perror("exec error");
            exit(1);
        }
        printf("Unknown command: %s\n", argv[0]);
```

```
45.         exit(1);
46.     } else if (pid < 0) {
47.         printf("Failed to create child process\n");
48.         return;
49.     } else {
50.         wait(NULL);
51.     }
52. }
```

五、 测试与调试

5.1 程序运行与测试

1. 将程序编写到 final-prog.c 文件中，用 gcc 进行编译。
2. 开始运行，将输出 my shell# 字符，于是我们可以在上面输入一些命令，以测试本次程序：

```
banbanstar@banbanstar-virtual-machine:~/Desktop/Linux-Course Design$ ./final-prog
my shell#
my shell#
my shell#
```

3. 信号测试：按键依次输入 Ctrl+D，Ctrl+C，Ctrl+Z，运行结果如下：

```
my shell#
 NOTE!!!  Please enter 'exit' or 'q' to exit the program.
my shell# ^C
 NOTE!!!  Please enter 'exit' or 'q' to exit the program.
my shell# ^Z
 NOTE!!!  Please enter 'exit' or 'q' to exit the program.
my shell#
```

4. 命令测试：输入 ls -l 或 pwd

```
my shell# ls -l
total 208
-rw-r--r-- 1 banbanstar banbanstar     4 6月   2 18:45 123.txt
-rwxrwxr-x 1 banbanstar banbanstar 17952 6月   2 18:45 final-prog
-rw-rw-r-- 1 banbanstar banbanstar  8931 6月   2 18:45 final-prog.c
-rwxrwxr-x 1 banbanstar banbanstar 17256 6月   2 11:17 lsl
-rw-rw-r-- 1 banbanstar banbanstar  1872 6月   2 11:14 lsl.c
-rw-rw-r-- 1 banbanstar banbanstar   468 6月   2 17:21 makefile
-rw-rw-r-- 1 banbanstar banbanstar  1241 6月   2 10:08 ptp.c
-rwxrwxr-x 1 banbanstar banbanstar 17216 6月   2 11:17 test1
-rw-rw-r-- 1 banbanstar banbanstar  1643 6月   1 10:05 test1.c
-rwxrwxr-x 1 banbanstar banbanstar 17216 6月   2 11:17 test2
-rw-rw-r-- 1 banbanstar banbanstar  2144 6月   1 10:05 test2.c
-rwxrwxr-x 1 banbanstar banbanstar 17464 6月   2 11:17 test3
-rw-rw-r-- 1 banbanstar banbanstar  5045 6月   1 10:32 test3.c
-rwxrwxr-x 1 banbanstar banbanstar 17704 6月   2 13:37 test4
-rw-rw-r-- 1 banbanstar banbanstar  6230 6月   2 13:37 test4.c
-rwxrwxr-x 1 banbanstar banbanstar 22136 6月   2 17:46 test5
-rw-rw-r-- 1 banbanstar banbanstar  9714 6月   2 17:46 test5.c
my shell#
```

```
my shell# pwd
/home/banbanstar/Desktop/Linux-Course Design
my shell#
```

5. 管道测试：输入以下命令

- ls -l|wc

```
my shell#
my shell# ls -l|wc
     20     173    1217
my shell#
```

- echo 111>123.txt

```
my shell# echo 111>123.txt
my shell#
my shell# cat 123.txt
111
```

- cat<123.txt

```
my shell#
my shell# cat<123.txt
111
my shell#
```

5.2 错误与程序退出处理

这段代码定义了两个函数用于错误处理。第一个函数 err_exit() 打印出红色的提示信息，告诉用户如果想要退出程序可以输入 "exit" 或者 "q"。第二个函数 err_command(char *buf) 打印红色的提示信息，告诉用户输入的命令不被程序所支持。

其中 \033[31m 和 \033[0m 是 ANSI 转义序列，用于改变文本颜色，将当前文本设置为红色，然后再恢复为默认颜色。fflush(stdout) 用于清空输出缓冲区，确保提示信息能立即

显示在屏幕上。

代码实现

```
void err_exit() {
    printf("\033[31m\n NOTE!!!  Please enter 'exit' or 'q' to exit the program.\n\033[0m");
}

void err_command(char *buf) {
    printf("\033[31m NOTE!!!  Command \033[0m'%s'\033[31m not found!\n\033[0m", buf);
    fflush(stdout);
}
```

5.3 信号处理控制

我们调试阶段，采用信号处理函数控制程序的退出，屏蔽 Ctrl+C、Ctrl+Z、Ctrl+D 的信号，并设置专用字符 exit、q 使程序退出，方便程序的控制。于是我们编写注册信号处理函数。当进程收到 SIGINT 或 SIGTSTP 信号时，会调用 sig_handler 函数。对于其他信号，我们会打印出"Unknown signal received."。

在主函数中，我们首先创建一个 sigaction 结构体，并将其初始化为 0。然后，我们将 sa_handler 字段设置为我们定义的 sig_handler 函数。接着，我们分别使用 sigaction 函数将 SIGINT、SIGTSTP 和 SIGQUIT 信号与该结构体关联起来，从而注册了这三个信号的处理函数。

需要注意的是，在 sig_handler 函数中调用了自定义的 err_exit()函数，它可能是一个退出程序的函数。另外，fflush(stdout)语句可以确保在输出数据之前，标准输出缓冲区中的所有数据都被刷新了，避免输出的数据不完整。

代码实现

```
// 定义信号处理函数，忽略SIGINT 和SIGTSTP 信号
void sig_handler(int signo) {
    fflush(stdout);
    if (signo == SIGINT || signo == SIGTSTP) {
        err_exit();
    } else if (signo == SIGQUIT) {
        printf("\n");
    } else {
        printf("Unknown signal received.\n");
    }
```

```
    fflush(stdout);
}

int main(void) {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = sig_handler;    // 设置信号处理函数为sig_handler
    sigaction(SIGINT, &sa, NULL);   // 注册SIGINT 信号处理函数
    sigaction(SIGTSTP, &sa, NULL);  // 注册SIGTSTP 信号处理函数
    sigaction(SIGQUIT, &sa, NULL);
    // ...
}
```

5.4 自定义命令 lsl

我们自定义了一个命令 lsl，这是一个使用 C 语言编写的程序，主要实现了类似于 Linux 系统中 ls 命令的功能。用户输入一个选项，然后程序会列出当前目录下所有文件和子目录的名称、权限信息以及文件大小（如果选择了对应的选项）。这是一个简单但实用的程序，可以帮助用户快速查看当前目录下的文件和子目录的基本信息。注意，这个程序只能列出当前目录下的文件列表，如果你想要列出其他目录下的文件，你需要修改程序中 opendir(".")这一行中的"."为其他目录的路径。

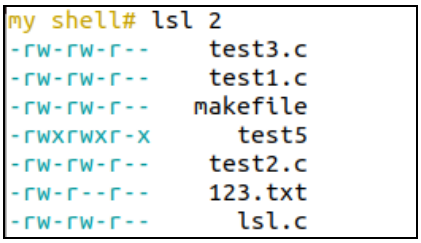
本程序使用 lsl <option>，其中<option>是一个数字，表示你想要使用哪种选项。在这个例子中，有两种选项可供选择：

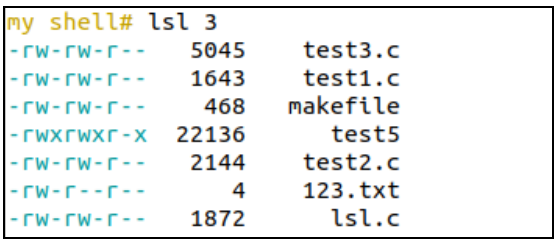
- lsl 2: 列出文件名和权限信息
- lsl 3: 列出文件名、权限和大小信息

使用方法

```
gcc -o lsl lsl.c
sudo cp lsl /usr/bin/
lsl 2
lsl 3
```

运行结果





代码实现

```c
void list_files(int option);

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: ls <option>\n");
        printf("\t-2\tList file names and permissions\n");
        printf("\t-3\tList file names, permissions, and sizes\n");
        exit(1);
    }

    int option = atoi(argv[1]);

    switch (option) {
        case 2:
        case 3: list_files(option); break;
        default: printf("Invalid option\n"); exit(1);
    }
    return 0;
}

void list_files(int option) {
    DIR *dp = opendir(".");
    if (dp == NULL) {
        perror("opendir error");
        exit(1);
    }

    struct dirent *dirp;
    struct stat filestat;
    char filename[1024];

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0 || strcmp(dirp->d_name, "..") == 0)
            continue;
        sprintf(filename, "./%s", dirp->d_name);
        if (stat(filename, &filestat) == -1) continue;
```

```
37.
38.         printf((S_ISDIR(filestat.st_mode)) ? "\033[36md" : "\033[36m-");
39.         printf((filestat.st_mode & S_IRUSR) ? "r" : "-");
40.         printf((filestat.st_mode & S_IWUSR) ? "w" : "-");
41.         printf((filestat.st_mode & S_IXUSR) ? "x" : "-");
42.         printf((filestat.st_mode & S_IRGRP) ? "r" : "-");
43.         printf((filestat.st_mode & S_IWGRP) ? "w" : "-");
44.         printf((filestat.st_mode & S_IXGRP) ? "x" : "-");
45.         printf((filestat.st_mode & S_IROTH) ? "r" : "-");
46.         printf((filestat.st_mode & S_IWOTH) ? "w" : "-");
47.         printf((filestat.st_mode & S_IXOTH) ? "x\033[0m" : "-\033[0m");
48.
49.         if (option == 2) printf(" %10s\n", dirp->d_name);  // 2
50.         else printf(" %6ld %10s\n", filestat.st_size, dirp->d_name);  // 3
51.     }
52.     closedir(dp);
53. }
```

六、 实验总结与分析

6.1 实验分析与展望

➢ 优点

1. 我们使用信号处理函数控制程序的退出，屏蔽 ctrl+c、ctrl+z、ctrl+d 的信号，并设置专用字符：exit、q 使程序退出，方便程序的控制。
2. 我们自定义了一个命令：lsl，需要通过命令行传递参数 2，3。当为 2 时，输出当前文件夹下每个文件的权限信息和文件名字；当为 3 时，输出当前文件夹下的每个文件的权限信息和大小与文件名字。
3. 使用管道 pipe，使当输入 | 时，使得两个子进程内的运行命令之间的通信得以实现。
4. 本程序能识别 |、>、< 的三种重定向功能。

➢ 缺点

1. 缺乏完善的错误处理机制，当发生错误时只是简单地输出错误信息，并没有进行相应的恢复或提示。

2. 不支持后台运行、环境变量设置、命令别名等高级功能，且不足够灵活方便。

- 展望

1. 可以将信号处理函数设置为可重入函数 sigaction，并使用 sigaction() 函数注册信号处理函数，避免使用 signal() 函数造成的不稳定性问题。
2. 可以使用 strtok() 函数代替手动解析命令行参数，使代码更加简洁。
3. 可以在子进程中执行 execvp() 之前，将标准输入、输出、错误重定向到/dev/null，以避免子进程继承父进程的标准输入、输出、错误描述符，从而导致意外的输出或输入。
4. 可以添加对 execvp() 函数的返回值进行判断，如果返回-1 则说明指定的程序不存在，需要打印出错信息并退出子进程。
5. 可以当执行管道操作或重定向操作时，需要及时回收所有已经终止的子进程资源，避免出现僵尸进程。重新设计提示符，使其更加友好和易用。

6.2 实验总结

本次的 Linux 课程设计是一个 Shell 模拟编程程序，我们通过使用 C 语言编写，实现了基本的命令行解析和执行功能。在这个过程中，我们小组学习到了很多关于进程、管道和 I/O 重定向等 Linux 基础知识，并且锻炼了自己的编程能力。感谢课程设计的许老师在本次实验中对我们小组的帮助与支持，为我们小组解答了我们许多的疑惑。

首先，在本次实验中，我了解到了信号的概念以及如何在程序中进行信号处理，通过注册信号处理函数，可以对程序接收到的各种信号进行处理，比如忽略某些信号或在接收到指定信号时执行相应操作。在本次实验中，我们忽略了 SIGINT 和 SIGTSTP 信号，并且在接收到其他信号时输出相应提示信息。

其次，本次实验涉及到了进程的创建和管理。在程序中使用 fork() 函数可以创建新的子进程，同时使用 waitpid() 函数可以回收已经终止的子进程资源，避免出现僵尸进程。此外，本次实验还使用了管道和 I/O 重定向技术，实现了命令行的管道和输入输出重定向功能。

我们也深刻地认识到了程序的错误处理机制的重要性。当我们编写程序时，应该习惯于使用 perror() 函数以及自定义的错误提示函数来处理程序中可能出现的错误，并且通过跟踪程

序的调试信息，分析程序调试程序。

本次实验不仅让我们学习到了很多 Linux 基础知识，也更好地了解了编程中各种工具和技术的应用。同时，通过实验过程中的调试和错误处理，不断提升了自己的编程能力和错误排查能力，这些都是我们在日后的学习和工作中必不可少的。最后，再次感谢帮助我们小组完成本次实验的所有老师和同学！衷心感谢！

附录：程序代码

```
#include <ctype.h>
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

#define MAX_ARGV 64  // 定义最大参数数量为 64

// 退出错误处理
void err_exit() {
    printf("\033[31m\n NOTE!!!  Please enter 'exit' or 'q' to exit the
program.\n\033[0m");
}

// 命令错误处理
void err_command(char *buf) {
    printf("\033[31m NOTE!!!  Command \033[0m'%s'\033[31m not found!\n\033[0m", buf);
    fflush(stdout);
}

// 定义信号处理函数
void sig_handler(int signo) {
    fflush(stdout);
    if (signo == SIGINT || signo == SIGTSTP) {
        err_exit();
    } else if (signo == SIGQUIT) {
        printf("\n");
    } else {
        printf("Unknown signal received.\n");
    }
    fflush(stdout);
}

// 加载程序
void do_exe(char *buf, char **argv) {
    pid_t pid = fork();
    if (pid == 0) {
        if (execvp(buf, argv) < 0) {  // 执行
```

```
            perror("exec error");
            exit(1);
        }
        err_command(buf);
        exit(1);
    } else {
        waitpid(pid, NULL, 0);
    }
}

void do_pipe(char **argv1, int argc1, char **argv2, int argc2) {
    int fd[2];
    pid_t pid1, pid2;
    int stdout_copy = dup(STDOUT_FILENO);  // 保存标准输出的文件描述符

    if (pipe(fd) < 0) {
        perror("pipe error");
        return;
    }

    if ((pid1 = fork()) < 0) {
        perror("fork error");
        return;
    } else if (pid1 == 0) {  // 子进程 1，执行管道左边的命令
        close(fd[0]);    // 关闭读端
        dup2(fd[1], 1);  // 标准输出 定向到 写端
        close(fd[1]);    // 关闭写端

        if (execvp(argv1[0], argv1) < 0) {  // 执行命令
            perror("exec error");
            exit(1);
        }
    } else {
        if ((pid2 = fork()) < 0) {
            perror("fork error");
            return;
        } else if (pid2 == 0) {  // 子进程 2，执行管道右边的命令
            close(fd[1]);    // 关闭写端
            dup2(fd[0], 0);  // 标准输入 定向到 读端
            close(fd[0]);    // 关闭读端

            if (execvp(argv2[0], argv2) < 0) {  // 执行
                perror("exec error");
                exit(1);
```

```
            }
        } else {
            close(fd[0]);
            close(fd[1]);

            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
        }
    }
    dup2(stdout_copy, STDOUT_FILENO);  // 恢复标准输出的文件描述符
    close(stdout_copy);
}

void do_redirect_input(char *argv[], int argc, char *filename) {
    int fd;
    fd = open(filename, O_RDONLY);
    if (fd < 0) {
        printf("Failed to open file %s for reading\n", filename);
        return;
    }

    pid_t pid = fork();
    if (pid == 0) {
        dup2(fd, STDIN_FILENO);  // 将标准输入 重定向到 文件
        close(fd);
        if (execvp(argv[0], argv) < 0) {  // 执行
            perror("exec error");
            exit(1);
        }
        printf("Unknown command: %s\n", argv[0]);
        exit(1);
    } else if (pid < 0) {
        printf("Failed to create child process\n");
        return;
    } else {
        wait(NULL);
    }
}

void do_redirect_output(char *argv[], int argc, char *filename) {
    int fd;
    // 打开文件，如果文件不存在，则创建该文件；如果文件存在，则清空文件内容
    fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

```
    if (fd < 0) {
        printf("Failed to open file %s for writing\n", filename);
        return;
    }

    pid_t pid = fork();
    if (pid == 0) {
        dup2(fd, STDOUT_FILENO);  // 将标准输出 重定向到 文件
        close(fd);

        if (execvp(argv[0], argv) < 0) {  // 执行
            perror("exec error");
            exit(1);
        }
        printf("Unknown command: %s\n", argv[0]);
        exit(1);
    } else if (pid < 0) {
        printf("Failed to create child process\n");
        return;
    } else {
        wait(NULL);
    }
}

void do_parse(char *buf) {
    char *argv[MAX_ARGV] = {};
    int argc = 0;
    char *args[MAX_ARGV] = {};  // 用于存储管道或重定向符号后面的命令及其参数
    int args_count = 0;

    // 解析命令行参数
    while (*buf != '\0') {
        if (!isspace(*buf)) {
            argv[argc++] = buf;
            while (!isspace(*buf) && *buf != '\0' && *buf != '|'
                   && *buf != '<' && *buf != '>') {
                buf++;
            }
            if (*buf == '\0') break;

            if (*buf == '|') {  // 解析管道符号后面的命令
                *buf++ = '\0';
                while (isspace(*buf)) buf++;  // 跳过 | 后面的空格
                while (*buf != '\0' && *buf != '|' && *buf != '<' && *buf != '>') {
```

```
                args[args_count++] = buf;
                while (!isspace(*buf) && *buf != '\0' && *buf != '|'
                       && *buf != '<' && *buf != '>') {
                    buf++;
                }
                if (*buf == '\0') break;
                if (*buf == '|' || *buf == '<' || *buf == '>') {
                    *buf++ = '\0';
                    break;
                }
                *buf++ = '\0';
            }
            do_pipe(argv, argc, args, args_count);  // 执行管道操作
            return;
        } else if (*buf == '>') {
            // 解析输出重定向符号后面的文件名
            *buf++ = '\0';
            while (isspace(*buf)) buf++;  // 跳过 > 后面的空格
            if (*buf != '\0') {
                do_redirect_output(argv, argc, buf);
                return;
            } else {
                printf("Error: missing destination file after '>'\n");
                return;
            }
        } else if (*buf == '<') {
            // 解析输入重定向符号后面的文件名
            *buf++ = '\0';
            while (isspace(*buf)) buf++;  // 跳过 < 后面的空格
            if (*buf != '\0') {
                do_redirect_input(argv, argc, buf);
                return;
            } else {
                printf("Error: missing source file after '<'\n");
                return;
            }
        } else {
            *buf++ = '\0';
        }
    } else {
        buf++;
    }
}
```

```
    // 没有输入任何命令
    if (argc == 0) return;
    do_exe(argv[0], argv);
}

int main(void) {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = sig_handler;    // 设置信号处理函数为 sig_handler
    sigaction(SIGINT, &sa, NULL);   // 注册 SIGINT 信号处理函数
    sigaction(SIGTSTP, &sa, NULL);  // 注册 SIGTSTP 信号处理函数
    sigaction(SIGQUIT, &sa, NULL);

    char buf[1024] = {};  // 定义存储用户输入的缓冲区
    int a;
    while (1) {  // 循环读取用户输入
        printf("\033[33mmy shell# \033[0m");
        fflush(stdout);                 // 刷新标准输出缓冲区
        memset(buf, 0x00, sizeof(buf));  // 清空 buf 缓冲区

        if (read(STDIN_FILENO, buf, sizeof(buf)) == -1) {
            // perror("read");
            fflush(stdout);  // 刷新标准输出缓冲区
            continue;
        }

        if (*buf == 0) {  // handle: ctrl + d
            err_exit();
            continue;
        }

        char *p = buf + strlen(buf) - 1;  // 去掉字符串末尾的换行符
        if (*p == '\n') *p = '\0';

        // 如果输入的是"exit"或"q"，则退出程序
        if (strcmp(buf, "exit") == 0 || strcmp(buf, "q") == 0) break;

        do_parse(buf);  // 对用户输入的命令进行解析并执行

        // 循环回收所有已经终止的子进程资源，避免出现僵尸进程
        while (waitpid(-1, NULL, WNOHANG) > 0);
    }
    return 0;
}
```