本次挑战性任务我的完成流程是按照测试点，这样可以在保证一部分代码的正确性的情况下编写其它代码。\

测试点1：相对路径+新增指令\

对于相对路径:
我的实现思路是:在env结构体中加入当前工作目录cwd

```
    // Lab 6 shell working directory
    char env_cwd[256]; // current working directory
```

同时使用系统调用syscall_getcwd和syscall_writecwd对当前工作目录进行读写
对于相对路径的识别，通过当前动作目录来将相对路径转换为绝对路径

```
void normalize_path_withcwd(const char* path, char* cwd, char* full_path){
    //checkcwd();
    char tmp[256];
    char *p, *q;
    // debugf("path is %s\n", path);
    // Handle absolute path
    if (path[0] == '/') {
        strcpy(tmp, path);
    } else {
        // Handle relative path
        // syscall_getcwd(tmp);
        strcpy(tmp, (const char*)cwd);
        int len_tmp = strlen(tmp);
        if (tmp[len_tmp-1] != '/') {
            tmp[len_tmp] = '/';
            tmp[len_tmp+1] = '\0';
            len_tmp = len_tmp + 1;
        }
        for (int i = 0; i < strlen(path); i++) {
            tmp[len_tmp] = path[i];
            len_tmp++;
        }
        tmp[len_tmp] = '\0';
    }
    // tmp是cwd (+/) + path
    // debugf("tmp_full_path is %s\n", tmp);
    // Remove redundant slashes and handle . and ..
    p = tmp;
    q = full_path;
    while (*p) {
        if (*p == '/') {
            //只有第一个/会在这里被识别
            *q++ = '/';
            while (*p == '/') p++;
        } else if (*p == '.' && (*(p+1) == '/' || *(p+1) == '\0')) {
            //处理 . /dir1/./dir2
```

```
            if (*(p+1) == '/') {
                p += 2;
            } else if (*(p+1) == '\0') {
                p ++;
            } else {
                // user_panic("error char after .");
            }
        } else if (*p == '.' && *(p+1) == '.') {
            //处理 ..
            if (*(p+2) == '/') {
                p += 3;
            } else if (*(p+2) == '\0'){
                p += 2;
            } else {
                // user_panic("error char after ..");
            }
            //q回退
            q --;
            //q回退到了根目录/..
            if (q == full_path) {
                *q++ = '/';
            } else {
                // /dir1/..,此时*q=/
                q--;
                while(*q != '/'){
                    q--;
                }
                // / 此时q在/后面写东西
                q++;
            }
        } else {
            // 处理目录名
            while (*p && *p != '/') *q++ = *p++;
        }
    }
    if((q-1) > full_path && *(q-1) == '/') {q--;}
    *q = '\0';
}
```

在user/lib/file.c中的open函数中进行识别，来保障相对路径在错误输出时的格式
对于新增指令，可以分为两类:
内建指令:cd, pwd, exit
外部指令:mkdir, rm, touch
对于内建指令，我在sh.c的main函数中fork前就进行了运行，并在sh.c函数里实现，对于外部指令，仿照课程组已有的ls指令修改即可。
内建指令实现:
cd:

```
// cd命令实现，返回int，严格按照表格要求
int do_cd(int argc, char **argv) {
```

```
    char target[256];
    struct Stat st;

    if (argc > 2) {
        debugf("Too many args for cd command\n");
        return 1;
    }

    if (argc == 1) {
        // cd 无参数，切换到根目录
        syscall_writecwd("/");
        // debugf("cd / success! cwd is /");
        // checkcwd();
        return 0;
    }

    // 规范化路径
    strcpy(target, (const char*) argv[1]);
    char cwd[256];
    syscall_getcwd(cwd);
    char full_path[256];
    normalize_path_withcwd((const char*) target, cwd, full_path);
    // debugf("full_path is %s\n", full_path);
    if (stat(full_path, &st) < 0) {
        debugf("cd: The directory '%s' does not exist\n", target);
        return 1;
    }
    if (!st.st_isdir) {
        debugf("cd: '%s' is not a directory\n", target);
        return 1;
    }
    syscall_writecwd((const char*)full_path);
    // debugf("cd success! cwd is %s\n", full_path);
    // checkcwd();
    return 0;
}
```

pwd:

```
// pwd命令实现，返回int，严格按照表格要求
int do_pwd(int argc, char **argv) {
    if (argc != 1) {
        printf("pwd: expected 0 arguments; got %d\n", argc-1);
        return 2;
    }
    char cwd[256];
    syscall_getcwd(cwd);
    printf("%s\n", cwd);
    return 0;
}
```

mkdir:

```
#include <lib.h>

// 递归创建目录 (mkdir -p)
void mkdir_p(const char *path) {
    char buf[1024];
    int len = 0;
    int mkfd = 0;
    // 逐级创建目录
    for (int i = 0; path[i] != '\0'; i++) {
        buf[len++] = path[i];
        if (path[i] == '/' || path[i] == '\0') {
            buf[len] = '\0';
            if (len > 1) { // 跳过根目录
                int fd = open(buf, O_RDONLY);
                if (fd >= 0) {
                    close(fd);
                } else {
                    mkfd = open(buf, O_MKDIR);
                    if (mkfd < 0) {
                        debugf("mkdir: cannot create directory '%s': No such file
or directory\n", buf);
                        return;
                    } else {
                        close(mkfd);
                    }
                }
            }
        }
    }
    mkfd = open(buf, O_MKDIR);
    if (mkfd < 0) {
        debugf("mkdir: cannot create directory '%s': No such file or directory\n",
buf);
        return;
    } else {
        close(mkfd);
    }
}

int main(int argc, char **argv) {
    int recursive = 0;
    int has_dir = 0;
    // 检查是否有-p参数
    for (int i = 1; i < argc; ++i) {
        if (argv[i] && strcmp(argv[i], "-p") == 0) {
            recursive = 1;
            argv[i] = 0;
            break;
        }
    }
    // 处理每个参数,其实最多只有两个参数
```

```
    for (int i = 1; i < argc; ++i) {
        if (!argv[i]) continue;
        has_dir = 1;
        int fd = open(argv[i], O_RDONLY);
        if (recursive) {
            // -p选项，目录存在直接返回，不存在递归创建
            if (fd >= 0) {
                close(fd);
                continue;
            }
            char full_path[256];
            char cwd[256];
            syscall_getcwd(cwd);
            normalize_path_withcwd((const char*) argv[i], cwd, full_path);
            mkdir_p(full_path);
        } else {
            // 非-p，目录存在报错
            if (fd >= 0) {
                close(fd);
                debugf("mkdir: cannot create directory '%s': File exists\n",
argv[i]);
                return 1;
            }
            int mkfd = open(argv[i], O_MKDIR);
            if (mkfd == -10) {
                debugf("mkdir: cannot create directory '%s': No such file or
directory\n", argv[i]);
                return 1;
            } else if (mkfd < 0) {
                debugf("mkdir: cannot create directory '%s': error %d\n", argv[i],
mkfd);
                return 1;
            } else {
                // debugf("mkdir: create directory successful '%s'.\n", argv[i]);
                close(mkfd);
            }
        }
    }
    if (!has_dir) {
        debugf("mkdir: missing operand\n");
        return 1;
    }
    return 0;
}
```

touch

```
#include <lib.h>

void do_touch(const char *filename) {
    int r = open(filename, O_RDONLY);
    if (r >= 0) {
```

```
        close(r);
        return;
    }
    // debugf("touch %s/n", filename);
    int create_result = open(filename, O_CREAT);
    if (create_result == -10) {
        debugf("touch: cannot touch '%s': No such file or directory\n", filename);
    } else if (create_result < 0) {
        debugf("touch: unknown error for %s, code %d\n", filename, create_result);
    } else {
        close(create_result);
    }
}

int main(int argc, char **argv) {
    if (argc <= 1) {
        debugf("touch: missing operand\n");
        return 0;
    }
    for (int i = 1; i < argc; i++) {
        do_touch(argv[i]);
    }
    return 0;
}
```

rm

```
#include <lib.h>

// 检查目标是否为目录
static int is_dir(const char *path) {
    struct Stat st;
    stat(path, &st);
    // debugf("rm : st_isdir : %d\n", st.st_isdir);
    return st.st_isdir;
}

int main(int argc, char **argv) {
    int opt_r = 0, opt_f = 0;
    int has_target = 0;
    // 解析参数，支持 -r 和 -rf
    for (int i = 1; i < argc; ++i) {
        if (argv[i] && strcmp(argv[i], "-r") == 0) {
            opt_r = 1;
            argv[i] = 0;
        } else if (argv[i] && strcmp(argv[i], "-rf") == 0) {
            opt_r = 1;
            opt_f = 1;
            argv[i] = 0;
        }
    }
    // 处理每个目标
```

```
    for (int i = 1; i < argc; ++i) {
        if (!argv[i]) continue;
        has_target = 1;
        int fd = open(argv[i], O_RDONLY);
        if (fd < 0) {
            // 文件不存在，-f不报错，否则报错
            if (!opt_f) {
                debugf("rm: cannot remove '%s': No such file or directory\n",
argv[i]);
                return 1;
            }
            continue;
        }
        close(fd);
        // 目录处理
        if (is_dir(argv[i])) {
            if (!opt_r) {
                debugf("rm: cannot remove '%s': Is a directory\n", argv[i]);
                return 1;
            }
        }
        // 删除文件或目录
        char full_path[256];
        char cwd[256];
        syscall_getcwd(cwd);
        normalize_path_withcwd((const char*) argv[i], cwd, full_path);
        remove(full_path);
        // remove失败时不输出（实验环境下通常不会失败）
    }
    if (!has_target) {
        debugf("rm: missing operand\n");
        return 1;
    }
    return 0;
}
```

测试点2：环境变量+注释

对于注释的实现,只需要简单地把#换成/0就好

```
for (int i = 0; i < len; ++i) {
    if (buf[i] == '#') {
        buf[i] = '\0';
    }
}
```

对于环境变量，我在env结构体中加入了环境变量数组，同时加入了syscall_getvar, syscall_setvar, syscall_unsetvar, syscall_showvars系统调用实现对环境变量的读写删和打印。

```
struct value {
    char name[17];   // 变量名，最长16字节+1结尾
    char value[17];  // 变量值，最长16字节+1结尾
    int is_env;      // 1为环境变量，0为局部变量
    int is_readonly;    // 1为只读，0为可写
};

// struct env
// Lab 6 shell variables
struct value env_vars[10]; // 最多存储20个变量
```

然后在shell指令中实现declare, unset内建指令
declare:

```
int do_declare(int argc, char **argv) {
    if (argc == 1) {
        // 输出所有环境变量
        char vars[10][34];
        // 临时创建一个指针数组
        char *ptrs[10];
        for (int i = 0; i < 10; i++) {
            ptrs[i] = vars[i];  // 让每个指针指向对应的行
        }
        syscall_showvars(ptrs);
        for (int i = 0; i < 10; i++) {
            if (vars[i][0] != '\0') {
                debugf("%s\n", vars[i]);
            } else {
                continue;
            }
        }
        return 0;
    }
    int opt_x = 0;
    int opt_r = 0;
    for (int i = 1; i < argc; ++i) {
        if (argv[i] && strcmp(argv[i], "-x") == 0) {
            opt_x = 1;
            argv[i] = 0;
        } else if (argv[i] && strcmp(argv[i], "-xr") == 0) {
            opt_x = 1;
            opt_r = 1;
            argv[i] = 0;
        }
    }
    for (int i = 1; i < argc; i++) {
        if (argv[i] == 0) {
            continue;
        }
        //处理NAME=VALUE
        char name[17];
```

```
            int name_len = 0;
            char value[17];
            int value_len = 0;
            int len = strlen(argv[i]);
            for (int j = 0; j < len && argv[i][j] != '='; j++) {
                name[name_len++] = argv[i][j];
            }
            name[name_len] = '\0';
            for (int j = name_len + 1; j < len; j++) {
                value[value_len++] = argv[i][j];
            }
            value[value_len] = '\0';
            syscall_setvar(name, value, opt_x, opt_r);
        }
        return 0;
    }
```

unset:

```
int do_unset(int argc, char **argv) {
    if (argc == 1) {
        debugf("unset: expected 1 argument; got %d\n", argc-1);
        return 1;
    }
    for (int i = 1; i < argc; i++) {
        syscall_unsetvar(argv[i]);
    }
    return 0;
}
```

指令输入优化+历史指令

对于历史指令，我的实现思路是在sh.c进程开始的时候初始化一个历史(从/.mosh_history文件中读取)，然后把每个指令都存进去。这里使用init_history, store_line两个函数来实现
init_history:

```
void init_history() {

    int fd;
    if ((fd = open("/.mosh_history", O_RDONLY | O_CREAT)) < 0) {
        debugf("There is some wrong with mosh_history: %d\n", fd);
        return;
    }
    hist_count = 0;
    // debugf("init\n");
    // 把文件中的内容读到hist_lines中
    while(hist_count < 20) {
        int offset = 0;
        int r = 0;
```

```
        // debugf("prepare to read one line\n");
        while((r = readn(fd, hist_lines[hist_count] + offset, 1)) == 1) {
            if (hist_lines[hist_count][offset] == '\n') {
                offset++;
                hist_lines[hist_count][offset] = '\0';
                hist_count++;
                break;
            }
        }
        // debugf("read one line: %s\n", hist_lines[hist_count-1]);
        if (r < 1) {
            break;
        }
    }
    current_count = hist_count - 1;
    close(fd);
}
```

store_line:

```
// 首先更新hist_lines,如果hist_count超过20，则删除最早的指令；之后将数组写到文件中
void store_line(char *buf) {
    int fd;

    if ((fd = open("/.mosh_history", O_WRONLY | O_TRUNC | O_CREAT)) < 0) {
        debugf("Failed to open .mosh_history\n");
        return;
    }

    if (hist_count >= 20)  {
        for (int i = 0; i < 19; i++) {
            strcpy(hist_lines[i], hist_lines[i + 1]);
        }
        strcpy(hist_lines[hist_count - 1], buf);
    } else {
        strcpy(hist_lines[hist_count++], buf);
    }
    current_count = hist_count - 1;
    int len = strlen(hist_lines[current_count]);
    hist_lines[current_count][len++] = '\n';
    hist_lines[current_count][len] = '\0';

    for (int i = 0; i < hist_count; ++i) {
        len = strlen(hist_lines[i]);
        write(fd, hist_lines[i], len);
    }

    close(fd);
}
```

对于快捷键的实现，我是在readline中实现的,使用cursor记录当前光标位置，使用len记录字符串总长度
上下左右键的实现:

```
        // 处理ESC序列（上下左右键）
        if (ch == 0x1b) { // ESC
            char seq[2];
            if (read(0, &seq[0], 1) == 1 && read(0, &seq[1], 1) == 1) {
                if (seq[0] == '[') {
                    if (seq[1] == 'A') { // 上键
                        if (hist_count > 0) {
                            if (current_count == -1) {
                                strcpy(original, buf);
                                current_count = hist_count - 1;
                            } else if (current_count > 0) {
                                current_count--;
                            }
                            strcpy(buf, hist_lines[current_count]);
                            len = strlen(buf);
                            cursor = len;
                            refresh_line(buf, len);
                        }
                        continue;
                    } else if (seq[1] == 'B') {
                        // 下键
                        if (current_count != -1) {
                            if (current_count < hist_count - 1) {
                                current_count++;
                                strcpy(buf, hist_lines[current_count]);
                            } else {
                                current_count = -1;
                                strcpy(buf, original);
                            }
                            len = strlen(buf);
                            cursor = len;
                            refresh_line(buf, len);
                        }
                        continue;
                    } else if (seq[1] == 'C') { // 右键
                        if (cursor < len) {
                            printf("\033[C");
                            cursor++;
                        }
                        continue;
                    } else if (seq[1] == 'D') { // 左键
                        if (cursor > 0) {
                            printf("\033[D");
                            cursor--;
                        }
                        continue;
                    }
                }
            }
```

```
            continue;
        }
```

同时对于屏幕输出问题，可以用refresh函数来刷新输出，本质上就是通过一个新的printf来覆盖原来的:

```
// 刷新行并将光标移动到正确位置
void refresh_line(const char *buf, int len) {
    printf("\r$ ");
    for (int i = 0; i < len; i++) {
        printf("%c", buf[i]);
    }
    printf("\033[K"); // 清除行尾
}
```

追加重定向+条件执行+反引号

追加重定向比较简单,只需要仿照O_MKDIR添加O_APPEND，再将fd_offset的值调整魏f_size就可以实现重定向了。

```
    if (rq->req_omode & O_APPEND) {
        ff->f_fd.fd_offset = ff->f_file.f_size;
    }
```

反引号实现时将反引号当成一个字符串读入，就像是读入一个文件名一样(不要最后一个),然后再parsecmd中处理，遇到反引号的指令就开一个进程执行，并且使用管道来传递输出:

```
        case 'w':
            if (argc >= MAXARGS) {
                debugf("too many arguments\n");
                exit();
            }
            // 处理反引号
            if (t[0] == '`') {
                t[0] = ' ';
                int p[2];
                if(pipe(p) < 0) {
                    debugf("failed to create pipe\n");
                    exit();
                }
                int env_id = fork();
                if (env_id < 0) {
                    debugf("failed to fork in sh.c\n");
                    exit();
                } else if (env_id == 0) { // 子进程执行反引号部分
                    close(p[0]);
                    dup(p[1], 1);
                    close(p[1]);
```

```
                    char tmp[1024] = {0};
                    strcpy(tmp, t);
                    runcmd(tmp);
                    exit();
                } else { // 父进程处理argv
                    close(p[1]);
                    memset(cmd, 0, sizeof(cmd));
                    int offset = 0;
                    int read_num = 0;
                    while((read_num = read(p[0], cmd + offset, sizeof(cmd))) > 0)
{
                        offset += read_num;
                    }
                    if (read_num < 0) {
                        debugf("error in `\n");
                        exit();
                    }
                    close(p[0]);
                    int len = strlen(cmd);
                    int j = 0;
                    for (j = len - 1; j > 0; j--) {
                        if (strchr(WHITESPACE, cmd[j]) && !strchr(WHITESPACE,
cmd[j-1])) {
                            cmd[j] = '\0';
                            len = j;
                            break;
                        }
                    }
                    // debugf("the new argv is %s\n", cmd);
                    argv[argc++] = cmd;
                    break;
                }
            }
            argv[argc++] = t;
            break;
```

条件执行:这个会比较麻烦，我的思路是识别到&&或者||，那么当前进程就fork一个子进程，这个进程会继续spwan新的孙进程，孙进程消亡后会用ipc传递运行返回值，子进程再将结果通过ipc传递给当前进程，当前进程根据返回值来决定是否继续执行。
一些重要函数:

```
//sh.c
//runcmd
    if (child >= 0) {
        u_int tmp;
        int res = ipc_recv(&tmp, 0, 0);
        if (condition_flag) {
            ipc_send(syscall_parent_id(0), res, 0, 0);
        }
        wait(child);
    }
```

```
//prasecmd
        case 'O':
            env_id = fork();
            if (env_id == 0) { // 子进程
                condition_flag = 1;
                return argc;
            } else {
                u_int tmp;
                int res = ipc_recv(&tmp, 0, 0);
                if (res != 0) {
                    return parsecmd(argv, rightpipe);
                } else {
                    return 0;
                }
            }
            break;
//user/lib/lisos.c
    int res = main(argc, argv);
    int parent_id = syscall_parent_id((u_int) 0);
    ipc_send(parent_id, res, 0, 0);
```

最后有一个小细节，就是exit(内建指令)的执行和外部指令执行不同，不会经过swpan，所以exit执行时需要进行返回值传递:

```
// sh.c main
            if (argc_local > 0 && strcmp(argv_local[0], "exit") == 0) {
                int parent_id = syscall_parent_id((u_int) 0);
                ipc_send(parent_id, 0, 0, 0);
                exit();
            }
```

最后shell原有的对指令的处理太过简单，因此课程组建议使用语法树进行分析。当时觉得没有必要而且风险大就没有做，但是后来因此多了很多麻烦。所以对于shell来说，在最开始写的时候不要害怕麻烦，不管是代码上的还是理解上的，最开始认真做了后面真的会轻松很多很多。