Lab6挑战性任务实验报告

一、各项任务实现思路

必做部分

1.1 实现一行多命令

在Lab6的挑战性任务中，我们接触到的第一个任务是实现“；”的功能，这一功能不算太难，整体上参考实验代码中管道的实现即可，代码如下所示。

```
case ';':       //lab6-challenge
        child = fork();
        *rightpipe = child;
        //重点在于让父亲执行分号右边的指令。
        if (child > 0) {
                wait(child);
                return parsecmd(argv, rightpipe);
        } else if (child == 0) {
                return argc;
        } else {
                user_panic("fork error!\n");
        }
        break;
```

这一功能需要注意的地方是为了**依次**实现分号两侧的命令，需要让父进程执行分号右边的指令，并且等待子进程结束才执行，我们可以通过wait函数来控制指令执行的顺序。

1.2 实现后台任务

第二个任务是实现“&”，实现思路是让父进程执行&右侧的指令，子进程执行&左侧的指令，但此时父进程无需等待子进程。shell的读取机制是读到下一条之前一直忙等，其余进程都被阻塞，所以可以让父进程不被阻塞住，继续执行之后读入的命令，子进程继续执行该命令。代码如下所示。

```
//parse_cmd()
case '&':
        child = fork();
        if (child == 0) {
                return argc;
        } else if (child > 0) {
                return parsecmd(argv, 0);
        } else {
                user_panic("fork error!\n");
        }
```

1.3 实现引号支持

该任务的实现思路是_gettoken()函数中对引号进行特判，当读到右引号时结束特判，过程有点像一道大一的程设题。为了向真实的linux的shell靠拢，我加入了转义符的实现，主要难点在于对于循环结束条件的判断比较复杂。代码如下所示，这里的代码保障了不会因为识别到被转义过的双引号而结束（如：\"）。

```
//_gettoken()
//实现引号支持
if (*s == '\"') {
    *p1 = ++s;
        while ((*s != '\"' || *(s-1) == '\\')
                && (*s != '\"' || *(s-1) != '\\' || *(s-2) != '\\')) {
                s++;
        }
        *s++ = 0;       //必须要封好字符串
        *p2 = s;
        return 'w';
}
```

同时需要修改parse_cmd的部分：

```
//parse_cmd()
case 'w':
        if (argc >= MAXARGS) {
                debugf("too many arguments\n");
                exit();
        }
        remove_slash(t);
        argv[argc++] = t;

//remove_slash()
void remove_slash(char *t) {
        char tmp[128] = {0};
        int length = 0;
        for (int i = 0; t[i]; i++) {
            //将需要转义的字符列举在这里
            int flag = (t[i+1] == '\'') || (t[i+1] == '\"');
            if (t[i] == '\\' && t[i+1] == '\\') {
                tmp[length++] = '\\';
                i++;
            } else if (t[i] == '\\' && flag) {
                continue;
            } else {
                tmp[length++] = t[i];
            }
        }
        tmp[length] = '\0';
        strcpy(t, tmp);
}
```

我新写了一个remove_slash的函数用于去除转义符。这里我只考虑了单双引号与\的转义，但也留出了日后添加转义符的空间，具有良好的可扩展性。

1.4 实现键入命令时任意位置的修改

从这一要求开始Lab6的挑战性任务的难度陡然而升，因为涉及到了与外设的交互。首先，经过查阅资料与试验，上下左右四个键的ascii码与我们平时接触到的字符不太一样，这四个键分为了三个值表示，分别是27+91+65/66/67/68（依次对应上下左右）。

具体的实现思路是：为了实现在不同位置的增加删除，需要设置一个pointer，专门记录当前光标所在的位置，每次在pointer处读入字符。若是读入字符的位置有字符，即对应增加字符的情况，就把从pointer到结尾的内容都先暂存起来，再用strcat把这几部分拼接起来。同时需要注意光标位置的挪动，我的方法是输出空格消除控制台的输出，然后输出退格键将光标位置挪回来。删除的情况与之类似。

```
//读入左右键时的情况
//readline()
if (buf[i+2] == 68) { //left
    if (pointer > 0) { //注意左右移动光标需要有个极限
        pointer--;
    } else {
        printf(" ");
    }
    continue;
} else if (buf[i+2] == 67) { //right
    if (cmd[pointer] != '\0') {
        pointer++;
    } else {
        printf("\b");
    }
    continue;
}


/***********正常读入字符***********/
if (cmd[pointer] != 0) {    //增加字符
    char tmp[1024] = {0};
    int length = 0;
    for (int j = pointer; j < n; j++) {
        if (cmd[j] == 0) {
            break;
        }
        tmp[length++] = cmd[j];
        printf(" ");    //清空控制台
    }
    tmp[length] = '\0';
    for (int j = 0; j < length; j++) {
        printf("\b");   //将光标挪回来
    }
    cmd[pointer++] = buf[i];
    cmd[pointer] = '\0';
    strcat(cmd, tmp);
    for (int j = 0; j < length; j++) {
        printf("%c", tmp[j]);
    }
    for (int j = 0; j < length; j++) {
        printf("\b");
    }
} else {
    cmd[pointer++] = buf[i];
}
```

```
/***********读入删除键***********/
if (buf[i] == 127) { //实现删除功能，与插入类似
    char tmp[1024] = {0};
    if (pointer == 0) { //不能无止境的删除
        continue;
    }
    printf("\b ");
    int length = 0;
    for (int j = pointer; j < n; j++) {
        if (cmd[j] == 0) {
            break;
        }
        tmp[length++] = cmd[j];
        printf(" ");
    }
    tmp[length] = '\0';
    for (int j = 0; j <= length; j++) {
        printf("\b");
    }
    cmd[--pointer] = '\0';
    strcat(cmd, tmp);
    for (int j = 0; j < length; j++) {
        printf("%c", tmp[j]);
    }
    for (int j = 0; j < length; j++) {
        printf("\b");
    }
    continue;
}
```

1.5 实现程序名称中 .b 的省略

这一任务相对简单，只需要当打不开文件时在文件名后面加上“.b“重新打开一遍即可。

```
// user/spawn.c
if ((fd = open(prog, O_RDONLY)) < 0) {
    char prog1[1024] = {0};
    int i;
    for (i = 0; prog[i] != 0; i++) {
    prog1[i] = prog[i];
    }
    prog1[i] = '.';
    prog1[i+1] = 'b';
    prog1[i+2] = '\0';
    if ((fd = open(prog1, O_RDONLY)) < 0) {
        return fd;
    }
}
```

1.6 实现更丰富的命令

1.6.1 tree

实现tree命令可以参考实验代码里的ls.c文件，循环读目录里的文件，如果这个文件也是个目录，就递归读取，并记录递归的层数以形式化输出。

```
// user/tree.c
void tree(char *path, int level) {
    int fd, n;
    struct File f;
    if ((fd = open(path, O_RDONLY)) < 0) {
        user_panic("open %s: %d", path, fd);
    }
    while ((n = readn(fd, &f, sizeof f)) == sizeof f) {
        if (f.f_name[0]) {
            for (int i = 0; i < level; i++) {
                debugf("   ");
            }
            debugf("|--");
            if (f.f_type == FTYPE_REG) {
                debugf("%s\n", f.f_name);
            } else if (f.f_type == FTYPE_DIR) {
                debugf("%s\n", f.f_name);
                char newPath[1024];
                strcpy(newPath, path);
                strcat(newPath, "/");
                strcat(newPath, f.f_name);
                tree(newPath, level+1);
            }
        }
    }
}
```

1.6.2 mkdir

mkdir的实现思路：先将路径里的文件拆分，存进二维数组；再分成两种情况，该指令可以添加-p的选项，如果添加-p就可以实现如果遇到不存在的目录就创建，反之则会报错。

```
int flag[256];

int dir_is_exist(char *path) {
        int fd;
        fd = open(path, O_RDONLY);
        close(fd);
        return fd >= 0 ? 1 : 0;
}

void mkdir(char *path) {
    int r, i, count = 0;;
    char dirs[50][1024];
    i = 0;
    int pathLen = strlen(path);
    while (i < pathLen) {
        int length = 0;
        while (path[i] != '/' && path[i]) {
            dirs[count][length++] = path[i++];
        }
        dirs[count++][length] = '\0';
        i++;
    }
    char now[1024] = {0};
    struct File *f;
    i = 0;
    while (i < count) {
        strcat(now, "/");
        strcat(now, dirs[i]);
        if (flag['p']) {
            if (!dir_is_exist(now)) {
                create(now, FTYPE_DIR);
            }
        } else {
            if (i == count - 1) { //最后一个dir，要创建
                create(now, FTYPE_DIR);
            } else {
                if (!dir_is_exist(now)) {
                    debugf("mkdir fail\n");
                }
            }
        }
        i++;
    }
}

void usage(void) {
        printf("usage: mkdir [-p] [file...]\n");
        exit();
}

int main(int argc, char **argv) {
        ARGBEGIN {
        default:
                usage();
        case 'p':
                flag[(u_char)ARGC()]++;
                break;
        }
        ARGEND
        for (int i = 0; i < argc; i++) {
                char *path = parse_path(argv[i]);
                mkdir(path);
```

```
        }
        return 0;
}
```

1.6.3 touch

touch的实现与mkdir类似，只是将创建的文件的类型改为普通文件。这里只放上核心功能代码。

```
void touch(char *path) {
    int r;
    char realPath[1024] = {0};
    if (path[0] != '/') {
        realPath[0] = '/';
    }
    strcat(realPath, path);
    r = create(realPath, FTYPE_REG);
    if (r < 0) {
        debugf("touch fail\n");
    }
}
```

1.6.4 O_CREAT与O_MKDIR

O_CREAT与O_MKDIR是实验代码里已经内置好的两个宏定义。并且修改/user/serv.c里的serve_open函数，使其在打开文件时，若找不到该文件就创建一个。

```
// Open the file.
    if ((r = file_open(rq->req_path, &f)) < 0) {
        //支持O_CREAT和O_MKDIR
        if (r == -E_NOT_FOUND && (rq->req_omode & O_CREAT)) {
            r = file_create(rq->req_path, &f);
            f->f_type = FTYPE_REG;
            if (r < 0) {
                ipc_send(envid, r, 0, 0);
                return;
            }
        } else if (r == -E_NOT_FOUND && (rq->req_omode & O_MKDIR)) {
            r = file_create(rq->req_path, &f);
            f->f_type = FTYPE_DIR;
            if (r < 0) {
                ipc_send(envid, r, 0, 0);
                return;
            }
        } else {
            ipc_send(envid, r, 0, 0);
            return;
        }
    }
```

1.7 实现历史命令功能

历史功能实现思路：

- 首先创建一个.history文件，创建一个his_pointer变量，指向当前.history里的那条指令。

```
void history_init() {
    int r;
    if ((r = create("/.history", FTYPE_REG)) < 0) {
        debugf("create .history fail\n");
    } else {
        debugf(".history is created\n");
    }
}
```

- 当读入一个换行符时，打开.history，为了实现每次都将命令添加到文件的末尾，并且调用seek函数，寻找到文件的末尾，将当前的指令写入.history。同时也在fd.c中实现一个get_size函数，为了寻找到文件的末尾。

```
/***********读入回车键***********/
if (buf[i] == '\r' || buf[i] == '\n') {
    buf[i] = 0;
    //存进历史记录里
    int fd;
    if ((fd = open("/.history", O_WRONLY)) < 0) {
        user_panic("open .history fail!");
    }
    int size;
    if ((size = get_size(fd)) < 0) {
        user_panic("get size fail!");
    }
    if ((r = seek(fd, size))) {
        user_panic("seek fail!");
    }
    if ((r = write(fd, cmd, strlen(cmd))) < 0) {
        user_panic("write .history fail!");
    }
    if ((r = write(fd, "\n", 1)) < 0) {
        user_panic("write .history fail!");
    }
    close(fd);
    his_num++;
    his_pointer = his_num; //每次读入回车都让指针指向最后一个
    return;
}
```

- 当读入上下键时，首先还是要控制光标的移动，其次是将.history文件中所有的指令存进一个二维数组，然后读出his_pointer处的指令，最后将当前控制台的输出清空，将这条历史指令输出到控制台。下展示读入上键时的代码，读入下键与之类似。

```
//......
else if (buf[i+2] == 65) { //up
    printf("%c%c%c", 27, 91, 66);
    if (his_pointer == 0) {
        continue;
    }
    his_pointer--;
    char tmp[128 * 128] = {0};
    int fd, count = 0, length = 0;
    if ((fd =  open("/.history", O_RDWR)) < 0) {
        user_panic("open fail!");
    }
    //保存当前的指令
    cmd[pointer] = '\0';
    //读历史指令
    if ((r = read(fd, tmp, sizeof tmp)) < 0) {
        user_panic("read .history fail!");
    }
    close(fd);
    //存进二维数组
    char records[100][1024] = {0};
    for (int j = 0; tmp[j]; j++) {
        if (tmp[j] != '\n') {
            records[count][length++] = tmp[j];
        } else {
            records[count][length] = '\0';
            count++;
            length = 0;
        }
    }
    strcpy(cmd, records[his_pointer]);
    //清除当前控制台的输出
    for (int j = 0; j < pointer; j++) {
        printf("\b \b");
    }
    pointer = strlen(cmd);
    printf("%s", cmd);
    continue;
}
```

- 最后实现history.c，当输入history指令时，将.history文件中都读出来，然后依次输出即可。

选做部分

选做部分的两个功能都有实现，下面按照实现顺序依次介绍。

选做部分 2：支持相对路径

实现思路：在进程控制块中加入一个当前路径的属性，运用系统调用改变当前所处路径和得到当前路径，增加系统调用的流程在上机的时候已经非常熟练了。在fork一个新进程时，需要让子进程复制父进程的当前路径。

```
// include
struct Env {
    //......
    char env_cur_path[1024];
}

// kern/syscall_all.c
void sys_change_dir(char *path) {
        strcpy(curenv->env_cur_path, path);
}

char *sys_get_cur_path() {
        return curenv->env_cur_path;
}
```

之后是实现cd与pwd的功能，这里要注意cd、pwd命令是内部命令，不能在外部文件中实现，要在sh.c内部实现。

```
// user/sh.c
void chdir(char *path) {
    char *new_path = parse_path(path);
    int fd;
    if ((fd = open(new_path, O_RDONLY)) < 0) {
        printf("no such file or directory!\n");
        return;
    }
    syscall_change_dir(new_path);
}

void getcwd() {
        char cur_path[1024] = {0};
        strcpy(cur_path, syscall_get_cur_path());
        printf("%s\n", cur_path);
}

//main
/*********** 内部命令cd ***********/
if (cmd[0] == 'c' && cmd[1] == 'd') {
    char path[MAXPATHLEN] = {0};
    int i;
    for (i = 0; cmd[i]; i++) {
        path[i] = cmd[i + 3];
    }
    path[i] = '\0';
    chdir(path);
    continue;
}
/*********** 内部命令pwd ***********/
if (cmd[0] == 'p' && cmd[1] == 'w' && cmd[2] == 'd') {
    getcwd();
    continue;
}
```

最后是要修改之前所有的指令，使之支持相对路径。为了增强代码的复用性，我在user/lib中新建了一个path.c文件，实现了将相对路径转化为绝对路径的功能。具体将在下文实验难点一节中重点介绍。

选做部分1：实现 shell 环境变量

实现思路：

- 同样也是需要在内核态更改，首先为了识别出父进程与子进程，需要给进程控制块编号，为1则为父进程，否则则为子进程，用以区分全局变量与环境变量。

```
// kern/syscall_all.c
void sys_set_shell_id() {
        curenv->env_shell_id++;
}

u_int sys_get_shell_id() {
        return curenv->env_shell_id;
}
```

- 在内核态设置两个全局二维数组，分别为env_key[128][128]和env_value[128][128]，key和value一定是一一对应的，还有两个一维数组is_global和is_read_only用以标识环境变量是否为全局变量和是否为只读变量。并在内核态实现对于环境变量的get与set方法。当前shell的id为1时可以读局部变量，否则不行。此外，还需在include/error.h中定义两种错误类型。

```
char env_key[128][128];
char env_value[128][128];
int is_global[128];
int is_read_only[128];
int env_var_cnt;
int sys_get_env_var(char *key, char *value) {
    int global_perm = curenv->env_shell_id == 1;
    for (int i = 0; i < env_var_cnt; i++) {
    if (strcmp(env_key[i], key) == 0) {
        if (!is_global[i]) {
            if (global_perm) {
                    strcpy(value, env_value[i]);
                return 0;
            } else {
                return -E_BAD_ENV_VAR;
            }
            } else {
                strcpy(value, env_value[i]);
                return 0;
            }
        }
    }
    return -E_ENV_VAR_NOT_FOUND;
}

int sys_set_env_var(char *key, char *value, int global, int read_only) {
    for (int i = 0; i < env_var_cnt; i++) {
        if (strcmp(key, env_key[i]) == 0) {
            if (is_read_only[i]) {
                return -E_BAD_ENV_VAR;
            }
            strcpy(env_value[i], value);
            is_global[i] = global;
            is_read_only[i] = read_only;
            return 0;
        }
    }
    strcpy(env_key[env_var_cnt], key);
    strcpy(env_value[env_var_cnt], value);
    is_global[env_var_cnt] = global;
    is_read_only[env_var_cnt++] = read_only;
    return 0;
}
```

- 实现declare.c，在该文件中先解析参数，将其按照“=”两侧划分，再进行系统调用存储该环境变量。

```
// user/declare.c
void declare(char *key, char *value) {
    int read_only = flag['r'];
    int global = flag['x'];
    int r;
    if ((r = syscall_set_env_var(key, value, global, read_only)) < 0) {
        user_panic("declare environment variable fail!");
    }
}
```

- 实现declare命令，同样进行系统调用，打印所有环境变量。

```
void sys_print_env_var() {
    for (int i = 0; i < env_var_cnt; i++) {
        printk("%d: name: %s value: %s\n", i, env_key[i], env_value[i]);
    }
}
```

- 实现echo $name功能，对$进行特判。

```
// user/echo.c
if (argv[i][0] == '$') {
    char key[128], value[128];
    for (j = 1; argv[i][j]; j++) {
        key[j-1] = argv[i][j];
    }
    key[j-1] = '\0';
    if ((r = syscall_get_env_var(key, value)) < 0) {
        user_panic("get environment variable fail!");
    }
    printf("%s", value);
    continue;
}
```

二、额外实现的功能

2.1 strcat

strcat是c语言标准库中string.h的一个重要函数，也是Lab6挑战性任务中我用的最多的函数。但是我们的string.h居然没有这个函数，所以我做的第一项任务就是把这个空缺给弥补上。

```
char *strcat(char *dest, const char *src) {
    char *tmp = dest;
    while (*dest)
        dest++;
        while ((*dest++ = *src++) != '\0');
        return tmp;
}
```

2.2 rm指令

rm指令也是shell命令中非常重要的指令，而且我们的实验代码中已经实现了remove的用户接口，只需要调用一下并且实现一些关键的选项即可。我一共实现了三种选项：分别为-r、-i和-f，分别为递归删除、删除前询问以及强制删除。代码如下所示。

```
int flag[256];

int is_dir(int fd) {
    int r;
    if ((r = get_type(fd)) < 0) {
        user_panic("cannot find the file!");
    }
    return r == FTYPE_DIR;
}

void query(char *path) {
    int r;
    if (flag['i']) {
        printf("delete the file?: %s [y/n] ", path);
        char c;
        if ((r = read(0, &c, 1)) != 1) {
            if (r < 0) {
                debugf("read error: %d\n", r);
            }
            exit();
        }
        printf("\n");
        if (c != 'y' && c != 'Y') {
            return;
        }
    }
}

void rm(char *path) {
    int r, fd, n;
    struct File f;
    if ((fd = open(path, O_RDONLY)) < 0) {
        user_panic("no such file or directory!");
    }
    if ((r = get_type(fd)) == FTYPE_DIR) {
        if (flag['r']) {
            while ((n = readn(fd, &f, sizeof f)) == sizeof f) {
                char new_path[1024] = {0};
                strcpy(new_path, path);
                if (f.f_name[0]) {
                    strcat(new_path, "/");
                    strcat(new_path, f.f_name);
                    if (f.f_type == FTYPE_REG) {
                        query(new_path);
                        remove(new_path);
                    } else {
                        rm(new_path);
                    }
                }
            }
            query(path);
            remove(path);
        } else {
            user_panic("cannot remove: it is a directory");
        }
    } else {
        query(path);
        remove(path);
    }
}

void usage() {
```

```
    printf("usage: rm [-选项] [file...]\n");
    exit();
}

int main(int argc, char **argv) {
        ARGBEGIN {
        case 'r':
    case 'i':
    case 'f':
        flag[(u_char)ARGC()]++;
        break;
    default:
        usage();
        break;
    }
    ARGEND
    for (int i = 0; i < argc; i++) {
        char *path = parse_path(argv[i]);
        rm(path);
    }
    return 0;
}
```

2.3 >>

我们的mos目前可以支持重定向，但还不能支持追加重定向。>> 的实现整体上可参考> 的实现，同样也是需要用到seek来锁定文件末尾，随后把标准输出复制到目标文件里，并且也是支持当文件不存在时就新建一个。

```
// user/sh.c
case '>>':
    if (gettoken(0, &t) != 'w') {
        debugf("syntax error: >> not followed by word\n");
        exit();
    }
    if ((fd = open(t, O_WRONLY | O_CREAT)) < 0) {
        user_panic("open fail!");
    }
    int size;
    if ((size = get_size(fd)) < 0) {
        user_panic("get size fail!");
    }
    if ((r = seek(fd, size))) {
        user_panic("seek fail!");
    }
    if ((r = dup(fd, 1)) < 0) {
        user_panic("dup fail!");
    }
    if ((r = close(fd)) < 0) {
        user_panic("close fail!");
    }
    break;
```

2.4 实现自动补全

实现思路：当读入到tab键时，比较当前这段字符串是否是当前所有支持的命令的前缀，再比较该字符串是否是当前目录下文件的前缀。当找到唯一一个可补全的字符串时可以补全，其余情况不做处理。

2.5 实现cp指令

实现思路：与rm类似，分为-r、-i选项，分别为递归复制，覆盖前询问。复制的本质是将来源文件的内容读出，并写入目标文件。注意赋值时从来源文件读的字符数组与目标文件写的字符数组不能是同一个，否则只能达成“浅拷贝”的效果。

```
// user/cp.c
int copy(int srcfd, int dstfd) {
    int r;
    char text[128 * 128] = {0};
    char new_text[128 * 128] = {0};
    if ((r = read(srcfd, text, sizeof text)) < 0) {
        user_panic("cannot read the file");
    }
    strcpy(new_text, text);
    if ((r = write(dstfd, new_text, sizeof text)) < 0) {
        user_panic("cannot write the file");
    }
}
```

三、实验结果

3.1 实现一行多命令 + 实现引号支持 + 实现程序名称中 .b 的省略

```
21373035:/ $ echo -n hello ; echo " world!"
hello world!
```

3.2 实现后台任务

```
@21373035:/ $ ls & echo hello
hello

@21373035:/ $ echo a
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b testpipe.b motd init.b num.b lorem touch.b mkdir.b testfdsharing.b testshell.sh declare.b script rm.b ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .history
a
```

3.3 实现键入命令时任意位置的修改

该任务适宜实时展现效果，难以通过图片形式看出效果

3.4 实现更多命令（tree mkdir touch）

```
21373035:/ $ mkdir -p a/b/c

21373035:/ $ touch a/d.txt

21373035:/ $ tree
|--aaa.txt
|--testarg.b
|--cat.b
|--pingpong.b
|--testbss.b
|--newmotd
|--history.b
|--testpiperace.b
|--testpipe.b
|--motd
|--init.b
|--num.b
|--lorem
|--touch.b
|--mkdir.b
|--testfdsharing.b
|--testshell.sh
|--declare.b
|--script
|--rm.b
|--ls.b
|--echo.b
|--sh.b
|--tree.b
|--halt.b
|--testptelibrary.b
|--.history
|--a
   |--b
      |--c
   |--d.txt
```

3.5 实现历史命令功能

上下键的功能不便以图片形式展示，此处只展现history指令

```
21373035:/ $ history
0 ls & echo a
1 echo -n hello ; echo " world!"
2 mkdir -p a/b/c
3 touch a/d.txt
4 tree
5 history
```

3.6 支持相对路径

为了更贴近linux，我将指令左侧改为了学号+相对路径的形式，与跳板机的格式相同。

```
21373035:/ $ mkdir -p a/b/c

21373035:/ $ cd a

21373035:/a $ cd ../a/./b/c/..

21373035:/a/b $ pwd
/a/b
```

3.7 实现 shell 环境变量

依次先后测试了：不同类型环境变量的创建、删除非只读变量、删除只读变量、修改只读变量、修改非只读变量、启动子shell后能否读取局部变量。

```
21373035:/ $ declare jkm1=aaa

21373035:/ $ declare -x jkm2=bbb

21373035:/ $ declare -r jkm3=ccc

21373035:/ $ declare -xr jkm4=ddd

21373035:/ $ declare
name: jkm1 value: aaa
name: jkm2 value: bbb
name: jkm3 value: ccc
name: jkm4 value: ddd

21373035:/ $ unset jkm1

21373035:/ $ unset jkm3
panic at unset.c:8: cannot unset: variable is read only

21373035:/ $ declare
name: jkm2 value: bbb
name: jkm3 value: ccc
name: jkm4 value: ddd

21373035:/ $ declare jkm3=eee
panic at declare.c:29: declare environment variable fail!

21373035:/ $ declare jkm2=eee

21373035:/ $ declare
name: jkm2 value: eee
name: jkm3 value: ccc
name: jkm4 value: ddd

21373035:/ $ sh

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                                                            ::
::                     JkmOS Shell 2023                         ::
::                                                            ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

21373035:/ $ echo $jkm3
panic at echo.c:23: get environment variable fail!
```

3.8 实现rm

- 删除普通文件

```
21373035:/ $ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b testpipe.b motd init.b num.b lorem touch.b mkdir.b testfdsharing.b testshell.sh declare.b
script rm.b ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .history jkm.txt

21373035:/ $ rm jkm.txt

21373035:/ $ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b testpipe.b motd init.b num.b lorem touch.b mkdir.b testfdsharing.b testshell.sh declare.b
script rm.b ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .history
```

- 递归删除目录

```
21373035:/ $ mkdir -p a/b/c

21373035:/ $ touch a/d.c

21373035:/ $ tree
|--aaa.txt
|--testarg.b
|--cat.b
|--pingpong.b
|--testbss.b
|--newmotd
|--history.b
|--testpiperace.b
|--testpipe.b
|--motd
|--init.b
|--num.b
|--lorem
|--touch.b
|--mkdir.b
|--testfdsharing.b
|--testshell.sh
|--declare.b
|--script
|--rm.b
|--ls.b
|--echo.b
|--sh.b
|--tree.b
|--unset.b
|--halt.b
|--testptelibrary.b
|--.history
|--a
   |--b
      |--c
   |--d.c
```

```
21373035:/ $ rm -r a

21373035:/ $ tree
|--aaa.txt
|--testarg.b
|--cat.b
|--pingpong.b
|--testbss.b
|--newmotd
|--history.b
|--testpiperace.b
|--testpipe.b
|--motd
|--init.b
|--num.b
|--lorem
|--touch.b
|--mkdir.b
|--testfdsharing.b
|--testshell.sh
|--declare.b
|--script
|--rm.b
|--ls.b
|--echo.b
|--sh.b
|--tree.b
|--unset.b
|--halt.b
|--testptelibrary.b
|--.history
```

- 删除前询问

```
21373035:/ $ mkdir -p a/b/c

21373035:/ $ touch a/d.c

21373035:/ $ rm -ri a
delete the file?: /a/b/c [y/n] y
delete the file?: /a/b [y/n] y
delete the file?: /a/d.c [y/n] y
delete the file?: /a [y/n] y

21373035:/ $ tree
|--aaa.txt
|--testarg.b
|--cat.b
|--pingpong.b
|--testbss.b
|--newmotd
|--history.b
|--testpiperace.b
|--testpipe.b
|--motd
|--init.b
|--num.b
|--lorem
|--touch.b
|--mkdir.b
|--testfdsharing.b
|--testshell.sh
|--declare.b
|--script
|--rm.b
|--ls.b
|--echo.b
|--sh.b
|--tree.b
|--unset.b
|--halt.b
|--testptelibrary.b
|--.history
```

3.9 实现>>

```
21373035:/ $ echo -n hello >> jkm.txt

21373035:/ $ cat jkm.txt
hello
21373035:/ $ echo " world!" >> jkm.txt

21373035:/ $ cat jkm.txt
hello world!
```

四、实验难点及解决方案

4.1 控制台输入部分与程序实际接收到的指令的协同

在user/sh.c的readline函数中，我们的代码用的是一个名为buf的字符数组读取的输入。而如果没有引入移动光标的功能，我们可以直接向runcmd中传入这一数组进行解析。但现在引入了光标的移动，buf会读入方向键的ascii码，并且所有字符也都是顺序读入的，显然是无法直接解析的。所以我选择了将这个buf仅作为读入字符使用，实际的指令部分新建一个名为cmd的字符数组，所有更改都对cmd进行更改。

4.2 如何掌握光标的位置

事实上，我们始终都需要保证控制台输入、程序接收到的指令与控制台输出三者保持完全一致，而后两者的一致性可以靠连续输出三个字符来完成，如输出一个下键可以通过 `printf("%c%c%c", 27, 91, 66)` 这条语句来实现。以此类推，我们可以通过ascii码来实现控制台中光标的随意移动。

4.3 修改fd.c，使得用户可以更方便的得知文件的一些基本属性。

在实现指令的过程中，我们经常需要得知文件的一些基本属性，如文件的名字、大小、种类等等，所以我们需要像面向对象一样，为文件写一些get方法，我是在fd.c实现的，可以通过fdnum就获得文件的大小、种类。

```c
// user/lib/fd.c
int get_size(int fdnum) {
    struct Fd *fd;
    struct Filefd *ffd;
    int r;
    if ((r = fd_lookup(fdnum, &fd)) < 0) {
        return r;
    }
    ffd = (struct Filefd *)fd;
    int size = ffd->f_file.f_size;
    return size;
}

int get_type(int fdnum) {
    struct Fd *fd;
    struct Filefd *ffd;
    int r;
    if ((r = fd_lookup(fdnum, &fd)) < 0) {
        return r;
    }
    ffd = (struct FileFd *)fd;
    int type = ffd->f_file.f_type;
    return type;
}
```

4.4 让所有指令都支持相对路径（./..)

为了向真实的linux靠拢，我还实现了..（上级目录）与.（当前目录），为了在所有文件中都能实现相对路径，我在user/lib里新建了一个path.c文件用于解析路径。代码如下所示。

```
char *parse_path(char *path) {
    char cur_path[MAXPATHLEN] = {0};
    char tmp_path[MAXPATHLEN] = {0};
    char new_path[MAXPATHLEN] = {0};
    strcpy(cur_path, syscall_get_cur_path());
    if (path[0] != '/') {       //传进来的是相对路径
        if (cur_path[strlen(cur_path) - 1] != '/') {s
            strcpy(tmp_path, cur_path);
            strcat(tmp_path, "/");
            strcat(tmp_path, path);
        } else {
            strcpy(tmp_path, cur_path);
            strcat(tmp_path, path);
        }
    } else {
        strcpy(tmp_path, path);
    }
    char files[128][128] = {0};
    char new_files[128][128] = {0};
    char tmp_file[128] = {0};
    int p = 0;  //指针
    int length = 0, file_cnt = 0;
    int tmp_path_len = strlen(tmp_path);
    if (tmp_path[tmp_path_len - 1] != '/') {
        tmp_path[tmp_path_len] = '/';
        tmp_path[tmp_path_len + 1] = '\0';
    }
    for (int i = 1; tmp_path[i]; i++) {
        if (tmp_path[i] != '/') {
            tmp_file[length++] = tmp_path[i];
        } else {
            tmp_file[length] = '\0';
            strcpy(files[file_cnt++], tmp_file);
            memset(tmp_file, 0, sizeof(tmp_file));
            length = 0;
        }
    }
    for (int i = 0; i < file_cnt; i++) {
        if (strcmp(files[i], "..") == 0) {
            p--;
        } else if (strcmp(files[i], ".") == 0) {
            continue;
        } else {
            memset(new_files[p], 0, sizeof(new_files[p]));
            strcpy(new_files[p++], files[i]);
        }
    }
    new_path[0] = '/';
    for (int i = 0; i < p; i++) {
        strcat(new_path, new_files[i]);
        if (i != p - 1) {
            strcat(new_path, "/");
        }
    }
    char *pointer = new_path;
    return pointer;
}
```

五、实验体会

挑战性任务的实验难度无疑是比任何一个Lab的难度都要高的，它考验的是你对于这些代码综合运用的能力，比之前的完形填空高到不知哪里去了。我在Lab6挑战性任务也是深有体会，此前偷懒没看的代码在挑战性任务里都会一一偿还的。诚然挑战性任务的确很难，但一点一点去完成实现的过程也是非常快乐并且富有满足感。尤其是当你选了Lab6这样效果非常直观的挑战性任务，看到属于自己的mos不断完善，并逐渐强大，就觉得所有的努力都是值得的。

回顾一学期操作系统的学习，我想对我提升最大的除了对于操作系统的理解，就是对于c语言的掌握程度了。学习过操作系统以后我对c语言的了解程度可以说是比大一上完程设、数据结构又更上一层楼。在完成Lab6的过程中，我和伙伴曾经仅仅对于指针的一个小问题就探讨了两个小时，并且还写了很多的测试小程序用来探究原理。我想这些方面对我的提升是最关键的，或许若干年后我会忘记操作系统的知识、会忘记c语言的机制，但是遇到问题去写一个测试程序验证的这种精神与方法是可以伴随我很长时间的。这就是这门课带给我的最大的提升。