OS_lab6 Shell挑战性任务实验报告

22373407 王飞阳

具体实现过程

实现不带 `.b` 后缀指令

主要修改文件：`user/lib/spawn.c`

主要思路：在使用 `spawn` 函数打开文件时，当第一次使用文件路径打开失败时，定义了一个 `prog2` 变量，将文件路径拷贝过来并在后边加上 `.b` ，再次尝试打开。

代码如下：

```
int fd;
char prog2[MAXPATHLEN];
if ((fd = open(prog, O_RDONLY)) < 0) {
    strcpy(prog2, prog);
    int len = strlen(prog);
    prog2[len] = '.';
    prog2[len+1] = 'b';
    prog2[len+2] = '\0';
    if ((fd = open(prog2, O_RDONLY)) < 0) return fd;
}
```

实现指令条件执行

主要修改文件：`user/sh.c` `user/lib/libos.c`

主要思路：

在 `libos.c` 中，实现了一个新的函数 `my_exit` 函数，在此函数中每次传入一个退出状态值，并加入了 `ipc` 通信，使得子进程结束之前向父进程发送状态值再结束，父进程接受状态值判断子进程是否成功执行，并修改了 `libmain` 函数。代码如下：

```
void my_exit(int status) {
    // After fs is ready (lab5), all our open files should be closed before
dying.
#if !defined(LAB) || LAB >= 5
    close_all();
#endif
    ipc_send(env->env_parent_id, status, 0 ,0);
    syscall_env_destroy(0);
    user_panic("unreachable code");
}

void libmain(int argc, char **argv) {
    // set env to point at our env structure in envs[].
    env = &envs[ENVX(syscall_getenvid())];

    // call user main routine
```

```
    int r = main(argc, argv);

    // exit gracefully
    my_exit(r);
}
```

而在 `sh.c` 中，对于 `parsecmd` 中，增加了对于 `&&` 和 `||` 的判断，当属于这两种情况时，通过 `fork_and_run` 函数 `fork` 一个子进程运行已经解析的命令，并接受返回值，通过返回值来判断是否需要调用 `skip_next_command` 函数跳过下一段命令，代码如下：

```
case '&':
    if (gettoken(0, &t, 0) == '&' && argc > 0) {
        if(argc != 0) last_status = fork_and_run(argv, argc);
        if (last_status != 0) {
            skip_next_command();
        }
        argc = 0;
        break;
    }
    else {
        debugf("syntax error: & not followed by &\n");
        my_exit(1);;
    }
    break;

case '|':
    if (gettoken(0, &t, 0) == '|' && argc > 0) {
        if(argc != 0) last_status = fork_and_run(argv, argc);
        if (last_status == 0) {
            skip_next_command();
        }
        argc = 0;
        break;
    }
    else{
        ...
        //管道
    }
```

而所引用到的 `fork_and_run` 函数， `skip_next_command` 函数如下：

```
int fork_and_run(char **argv, int argc) {
    int child = fork();
    if (child == 0) {
        argv[argc] = 0;
        int r = spawn(argv[0], argv);
        int t;
        int status = ipc_recv(&t, 0, 0);
        if(status == 1) my_exit(1);
        close_all();
        my_exit(0);
    }
    else {
```

```
        int t;
        int r = ipc_recv(&t, 0, 0);
        // debugf("recv: %d\n", r);
        return r;
    }
}

void skip_next_command() {
    char *t;
    int c;
    while ((c = gettoken(0, &t, 0)) != 0) {
        if(c == '&'){
            c = gettoken(0, &t, 0);
            if(c == 0 || c == '&') break;
        }
        else if(c == '|'){
            c = gettoken(0, &t, 0);
            if(c == 0 || c == '|') break;
        }
        // 只消耗标记直到下一个 && 或 ||
    }
}
```

实现更多指令

主要添加文件：`user/touch.c` `user/mkdir.c` `user/rm.c`

`touch`

主要思路：借鉴已有的 `ls.c`，使用类似的参数解析方法，获得参数之后，调用 `touch` 函数创建文件。

提前准备：实现了 `user_create` 函数，用于使用系统调用创建一个文件或者文件夹。

`touch.c` 如下：

```
#include <lib.h>

void touch(char* path){
    int r;
    r = user_create(path, 0);
    if (r == -E_FILE_EXISTS) return;
    else if (r < 0) user_panic("touch: cannot touch '%s': No such file or
directory\n", path);
}

void usage(void){
    debugf("usage: touch [] [file...]\n");
    my_exit(1);
}

int main(int argc, char **argv){
    ARGBEGIN {
        default:
            usage();
    }ARGEND
```

```
    for(int i = 0; i < argc; i++) touch(argv[i]);
    return 0;
}
```

`user_create` 函数，及其相关依赖：

```
// user/lib/file.c
int user_create(const char *path, int type) {
    int r;
    if ((r = fsipc_create(path, type)) < 0) return r;
    return 0;
}

// user/lib/fsipc.c
int fsipc_create(const char *path, int isdir) {
    struct Fsreq_create *req = (struct Fsreq_create *)fsipcbuf;

    if (strlen(path) >= MAXPATHLEN) return -E_BAD_PATH;
    strcpy((char *)req->req_path, path);
    req->req_isdir = isdir;
    return fsipc(FSREQ_CREATE, req, 0, 0);
}

// user/include/fsreq.h
enum {
    FSREQ_OPEN,
    FSREQ_MAP,
    FSREQ_SET_SIZE,
    FSREQ_CLOSE,
    FSREQ_DIRTY,
    FSREQ_REMOVE,
    FSREQ_SYNC,
    FSREQ_CREATE,
    MAX_FSREQNO,
};
struct Fsreq_create {
    u_char req_path[MAXPATHLEN];
    u_int req_isdir;
};

// fs/serv.c
void serve_create(u_int envid, struct Fsreq_create *rq) {
    struct File *f;
    int r = file_create(rq->req_path, NULL, rq->req_isdir);
    ipc_send(envid, r, 0, 0);
}
void *serve_table[MAX_FSREQNO] = {
    [FSREQ_OPEN] = serve_open,   [FSREQ_MAP] = serve_map,     [FSREQ_SET_SIZE] =
serve_set_size,
    [FSREQ_CLOSE] = serve_close, [FSREQ_DIRTY] = serve_dirty, [FSREQ_REMOVE] =
serve_remove,
    [FSREQ_SYNC] = serve_sync, [FSREQ_CREATE] = serve_create,
};
```

`mkdir`

主要思路：和实现 `touch` 类似，同样是调用 `user_create` 函数创建文件夹，只是增加了再 `-p` 选项下的递归创建文件夹（`createRec` 函数）。

提前准备：实现了 `user_create` 函数，用于使用系统调用创建一个文件或者文件夹。

具体代码如下：

```
#include <lib.h>

int flag[256];
void createRec(char *p) {
    char *t = p;
    int r;
    if (*p == '/') t = p + 1;
    while (*t) {
        while (*t && *t != '/') t++;
        char save = *t;
        *t = '\0';
        r = user_create(p, 1);
        if (r != -E_FILE_EXISTS && r!=0) user_panic("damn");
        *t = save;
        if (*t == '/') t++;
    }
}

void mkdir(char* path) {
    int r;
    // debugf("%d\n",flag['p']);
    if (flag['p']) createRec(path);
    else {
        r = user_create(path, 1);
        if (r == -E_FILE_EXISTS) user_panic("mkdir: cannot create directory '%s':
File exists\n", path);
        else if (r < 0) user_panic("mkdir: cannot create directory '%s': No such
file or directory\n", path);
    }
}

void usage(void) {
    debugf("usage: mkdir [-p] [file...]\n");
    my_exit(1);
}

int main(int argc, char **argv) {
    ARGBEGIN{
    default:
        usage();
    case 'p':
        flag[(u_char)ARGC()]++;
        break;
    } ARGEND
    if (argc == 0) debugf("please input a directory name\n");
    else {
        for (int i = 0; i < argc; i++) mkdir(argv[i]);
```

```
    }
    return 0;
}
```

rm

主要思路：同上面的函数类似，只是调用的已经实现好的 `remove` 函数，并新增了几个参数选项。

具体代码如下：

```
#include <lib.h>

int flag[256];

void rm(char* path){

    if(!flag['r']) rm_file(path);
    else if(!flag['f']){
        int r;
        r = remove(path);
        if (r < 0) user_panic("rm: cannot remove '%s': No such file or
directory\n", path);
    }
    else{
        int r;
        r = remove(path);
    }
}

void rm_file(char * path){
    int r;
    struct Stat st;
    if ((r = stat(path, &st)) < 0){
        user_panic("rm: cannot remove '%s': No such file or directory\n", path);
        return;
    }
    if (st.st_isdir){
        user_panic("rm: cannot remove '%s': Is a directory\n", path);
        return;
    }
    r = remove(path);
    if (r < 0) user_panic("rm: cannot remove '%s': No such file or directory\n",
path);
}

void usage(void){
    debugf("usage: rm [] [file...]\n");
    my_exit(1);
}

int main(int argc, char **argv){
    ARGBEGIN {
        default:
            usage();
        case 'r':
```

```
            flag[(u_char)ARGC()]++;
            break;
        case 'f':
            flag[(u_char)ARGC()]++;
            break;
    }ARGEND
    // debugf("%d   %d\n",flag['r'],flag['f']);
    for(int i = 0; i < argc; i++) rm(argv[i]);
    return 0;
}
```

实现反引号

主要修改文件：`user/sh.c` `user/echo.c`

主要思路：在`parsecmd`每次循环开始之前，判断是否存在反引号，如果存在，就将后面的输入全当作字符串经行处理，将其完整的传给`echo.c`中。

```
while (1) {
        char *t;
        int fd, r;
        int c = gettoken(0, &t, 0);
        if( t != 0){
            if(*t == '`') flag = 1;
            if(*t == '"') flag_double = 1;
        }
        ...
        case '<':
            if(flag || flag_double){
                char *temp = "<";
                argv[argc++] = temp;
                break;
            }
            ...
        ...
}
```

而在`echo.c`中，对传来的命令进行处理，如果存在反引号，就将反引号中的命令以字符串的形式调用`runcmd`再运行一遍。

```
if(buf[0] == '`'){
    char *end = strchr(buf+1, '`');
    if(end != NULL) {
        int cmd_len = end - buf - 1;
        for(int j = 0; j < cmd_len; j++){
            cmd[j] = buf[j+1];
        }
        cmd[cmd_len] = '\0';
        execute_command();
    }
    else printf("%s", buf);
}
else printf("%s", buf);
```

```
void execute_command(){
    runcmd(cmd);
}
```

实现注释功能

主要修改文件：`user/sh.c`

主要实现思路：这个较为容易，修改 `parsecmd` ，用一个标记 `node` 来记录已经出现 `#` ，如果有了此标记，那就直接退出 `while` 循环，并返回 `argc` ，相当于忽略后面的内容。

```
case '#':
    if(flag || flag_double){
        char *temp = "#";
        argv[argc++] = temp;
        break;
    }
    note = 1;
    break;

// while循环最后：
if(note) return argc;
```

实现历史指令

主要修改文件：`user/sh.c` `user/init.c`

主要实现思路：首先，在刚初始化之时，就创建 `.mosh_history` 文件，用于存储历史指令。

```
//存储历史指令
user_create("/.mosh_history", 0);
```

由于 `history` 为 `shell built-in command`，故需要直接在 `sh.c` 中实现 `history` 的功能。

具体实现如下，其中：

- `to_num`：用于将一个数字型字符串转为 `int` 型整数
- `save_history_cmd`：用于将输入的指令存在 `.mosh_history` 文件中，注意为追加型而不是覆盖型
- `get_history_cmd`：用于使用 `Up` 和 `Down` 键输出历史指令
- `history`：用于使用 `history` 指令输出历史记录
- `run_history`：对 `history` 的封装，调用 `history`

具体代码如下：

```
char history_flag[256];

char history_buf[8192];

int to_num(char *s) {
    int ans = 0;
    while(*s) {
```

```
        ans = ans * 10 + (*s) - '0';
        s++;
    }
    return ans;
}

void history(int arg) {
    int fd, n;
    int r;
    int i;
    int line;
    int fast = -1;
    int allLine = 0;
    if (history_flag['c']) {
        if ((fd = open("/.mosh_history", O_RDONLY)) < 0)
            user_panic("open %d: error in histroy\n",  fd);
        if ((r = seek(fd, 0)) < 0) {
            user_panic("error in seek history\n");
        }
        r = ftruncate(fd, 0);
        if (r) user_panic("error in ftruncate in history\n");
        return;
    }
    if (arg == -1) {
        if ((fd = open("/.mosh_history", O_RDONLY)) < 0)
            user_panic("open %d: error in histroy\n",  fd);
        if ((r = seek(fd, 0)) < 0) {
            user_panic("error in seek history\n");
        }
        n=read(fd, history_buf, (long)sizeof history_buf);
        line = 1;
        for (i = 0; i < n; i++) {
            debugf("%c", *(history_buf + i));
            if (*(history_buf + i) == '\n') {
                if (!history_buf[i + 1]) break;
            }
        }
    } else {
        if ((fd = open("/.mosh_history", O_RDWR | O_APPEND)) < 0)
            user_panic("open %d: error in histroy\n",  fd);
        if ((r = seek(fd, 0)) < 0) {
            user_panic("error in seek history\n");
        }
        n = read(fd, history_buf, (long)sizeof history_buf);
        for (i = 0; i < n; i++) {
            if (*(history_buf + i) == '\n') {
                allLine++;
            }
        }
        for (i = n - 1; i >= 0; i--) {
            if (*(history_buf + i) == '\n') {
                fast++;
                if (fast == arg) {
                    i++;
                    break;
                }
```

```
            }
            if (i == 0) {
                fast++;
                break;
            }
        }
        line = ( allLine - arg + 1 ) > 0 ? ( allLine - arg + 1 ) : 1;
        for (; i < n; i++) {
            debugf("%c", *(history_buf + i));
            if (*(history_buf + i) == '\n') {
                if (!history_buf[i + 1]) break;
            }
        }
    }
}


void save_history_cmd(char *buf) {
    int fd, n;
    int r;
    if ((fd = open("/.mosh_history", O_RDWR)) < 0)
        user_panic("open %d error in save_history_cmd\n", fd);
    struct Fd* fd2;
    struct Filefd* ffd2;
    fd_lookup(fd , &fd2);
    ffd2 = (struct Filefd*) fd2;
    seek(fd, ffd2->f_file.f_size);
    r = write(fd, buf, strlen(buf));
    if (r < 0) {
        user_panic("error in save_history_cmd of write Buf\n");
    }
    r = write(fd, "\n", 1);
    if (r < 0) {
        user_panic("error in save_history_cmd of write enter\n");
    }
    close(fd);
}

void get_history_cmd(char* command, int clear, int upOrDown, int* repeat){
    static int curLine = 0;
    int fd, n;
    int r;
    int i;
    char buf[8192];
    char cmp[256];
    int j;
    //upOrDown 1 is up, 0 is down
    if (clear) {
        curLine = 0;
        return;
    }
    if ((fd = open("/.mosh_history", O_RDONLY)) < 0)
        user_panic("open %s: error  in histroy\n", fd);
    if ((r = seek(fd, 0)) < 0) {
        user_panic("error in seek history\n");
    }
```

```
    n = read(fd, buf, (long)sizeof buf);
    if (upOrDown == 1) {
        curLine++;
    } else if (upOrDown == 0){
        curLine--;
    } else if (upOrDown == -1) {
        for (i = n - 2; i > 0; i--) {
            if (buf[i] == '\n') {
                i++;
                break;
            }
        }
        for (j = 0; i < n; i++, j++){
            cmp[j] = buf[i];
            if (buf[i] == '\n') {
                cmp[j] = 0;
                break;
            }
        }
        if (strcmp(cmp, command) == 0) {
            *repeat = 1;
        } else {
            *repeat = 0;
        }
        close(fd);
        return;
    }
    int fast = -1;
    for (i = n - 1; i >= 0; i--) {
        if (*(buf + i) == '\n') {
            fast++;
            if (fast >= curLine) {
                i++;
                break;
            }
        }
        if (i == 0) {
            fast++;
            break;
        }
    }
    char* tmp = command;
    for (; i < n; i++, tmp++){
        *tmp = buf[i];
        if (buf[i] == '\n') {
            *tmp = 0;
            break;
        }
    }
    for (i = n - 2; i > 0; i--) {
        if (buf[i] == '\n') {
            i++;
            break;
        }
    }
    for (j = 0; i < n; i++, j++){
```

```
            cmp[j] = buf[i];
            if (buf[i] == '\n') {
                cmp[j] = 0;
                break;
            }
        }
        if (strcmp(cmp, command) == 0) {
            *repeat = 1;
        } else {
            *repeat = 0;
        }
        if (fast <= curLine) {
            curLine = fast;
        }
        if (curLine <= 0) {
            curLine = 0;
            command[0] = 0;
        }
        close(fd);
}

int run_history(int argc, char **argv){
    int i;

    ARGBEGIN{
    default:
        usage();
    case 'c':
        history_flag[(u_char)ARGC()]++;
        break;
    }ARGEND
    if (argc == 1) {
        history(to_num(argv[0]));
    } else {
        history(-1);
    }
    return 0;
}
```

而在原来的 `runcmd`，需要对 `history` 进行特判，而不是调用 `spawn` 函数：

```
if(strcmp(history,argv[0]) == 0){
    // debugf("success\n");
    run_history(argc, argv);
    my_exit(0);
}
else{
    ...
}
```

实现一行多指令

主要修改文件：`user/sh.c`

主要思路：在 `parsecmd` 函数中，如果遇到 `;`，就 `fork` 出一个子进程去运行已经解析出来的命令。而父进程则继续解析之后的部分，代码如下：

```
case ';':
    pid = fork();
    if (pid == 0) {
        return argc;
    }
    else {
        int whom;
        int rev = ipc_recv(&whom, 0, 0);
        *rightpipe = 0;
        return parsecmd(argv, rightpipe);
    }
    break;
```

实现追加重定向

主要修改文件：`user/sh.c`

主要思路：在 `parsecmd` 函数中，如果遇到 `>>`，首先通过打开文件的 `fd` 调用 `fd_lookup` 函数获得对应的 `struct Fd*`，再将其强制类型转化为 `struct Filefd*`，这样就能通过 `ffd2->f_file.f_size`，获得文件的大小。最后使用 `seek` 函数，将 `fd->fd_offset` 修改为文件大小，这样每次就可以追加写，而不会覆盖原来的内容了。

具体代码如下：

```
case '>':
    if(flag || flag_double){
        char *temp = ">";
        argv[argc++] = temp;
        break;
    }
    char ch = gettoken(0, &t, 0);
    if (ch == '>'){
        if (gettoken(0, &t, 0) != 'w') {
            debugf("syntax error: >> not followed by word\n");
            my_exit(1);
        }
        if ((r = open(t, O_WRONLY | O_CREAT)) < 0) {
            debugf(">> open error\n");
            my_exit(1);
        }
        fd = r;
        struct Fd* fd2;
        struct Filefd* ffd2;
        fd_lookup(fd , &fd2);
        ffd2 = (struct Filefd*) fd2;
        seek(fd, ffd2->f_file.f_size);
```

```
        dup(fd, 1);
        close(fd);
        break;
    }
    ...
```

实现引号支持

主要修改文件：`user/sh.c` `user/echo.c`

主要思路：这部分较简单，和反引号类似，设置双引号的标记，如果读到双引号，就将后面内容当字符串传给 `echo.c` ，由 `echo.c` 处理：

```
while (1) {
        char *t;
        int fd, r;
        int c = gettoken(0, &t, 0);
        if( t != 0){
            if(*t == '`') flag = 1;
            if(*t == '"') flag_double = 1;
        }
        ...
        case '<':
            if(flag || flag_double){
                char *temp = "<";
                argv[argc++] = temp;
                break;
            }
            ...
        ...
}
```

而在 `echo.c` 中，对传入的命令进行解析，直接输出双引号里的内容就行：

```
if(buf[0] == '"'){
    buf[k-2] = 0;
    printf("%s\n",buf+1);
    return 0;
}
...
```

实现前后台任务管理

由于软院的期末周考试较多，复习压力大，前后台任务管理部分就没有实现（QAQ），不过会接下来在假期尝试实现……