shell-challenge

task1：实现不带 .b 后缀指令

`user/lib/spawn.c`中在尝试打开程序路径失败后，检查是否是.b结尾，如果不是，更改为以.b结尾再次尝试打开。

```
// user/lib/spawn.c
    int fd;
    if ((fd = open(prog, O_RDONLY)) < 0) {
        // delete:return fd;
        int len = strlen(prog);
        if(len < 2 || prog[len-1] != 'b' || prog[len-2] != '.'){
            char tmp_prog[MAXPATHLEN];
            strcpy(tmp_prog, prog);
            tmp_prog[len] = '.';
            tmp_prog[len + 1] = 'b';
            tmp_prog[len + 2] = '\0';
            if((fd = open(tmp_prog, O_RDONLY)) < 0){
                return fd;
            }
        } else {
            return fd;
        }
    }
```

task2：实现指令条件执行

指令条件执行与一行多指令相似，只是需要进一步判断要不要执行下一条。

1. 修改 `parsecmd` 函数，使其能够解析 `&&` 和 `||` 运算符。

```
// user/sh.c
int _gettoken(char *s, char **p1, char **p2) {
    // ...
    if (*s == '&' && *(s + 1) == '&') {
        *p1 = s;
        *s = 0;
        s++;
        *s = 0;
        s ++;
        *p2 = s;
        return 'a'; // 'a' stands for &&
    }

    if (*s == '|' && *(s + 1) == '|') {
        *p1 = s;
        *s = 0;
        s += 2;
```

```
        *p2 = s;
        return 'o'; // 'o' stands for ||
    }
    // ...
}
```

2. 修改 `runcmd` 函数，通过ipc通信来判断是否要执行下一条指令。

   以 `&&` 为例：

   fork出的子进程将need_to_send设为1后执行解析完毕的命令

   父进程通过接受的返回值①判断要不要继续执行，基本逻辑是：

   - 如果返回0，继续执行后面的指令
   - 否则，在这条指令之后如果有'||'，可以执行'||'之后的内容，否则退出

```
int parsecmd(char **argv, int *rightpipe) {
    int argc = 0;
    while (1) {
        need_to_send = 0;   // 发回执行消息
        back_commend = 0;   // 是否是后台指令
        char *t;
        int fd, r;
        int left;               // 条件指令
        int c = gettoken(0, &t);
        // ...
        case 'a':  // &&
            left = fork();
            if (left > 0) {
                r = ipc_recv(0, 0, 0);              //①
                if (r == 0) {
                    return parsecmd(argv, rightpipe);
                } else {
                    while (1) {
                        int op = gettoken(0, &t);
                        if (op == 0) {
                            return 0;
                        } else if (op == 'o') {
                            return parsecmd(argv, rightpipe);
                        }
                    }
                }
            } else {
                need_to_send = 1;
                return argc;
            }
            break;
        // ...
    }
    return argc;
}
```

进一步的，子进程通过spawn出的child执行命令，在sh.c的 `runcmd` 做出如下修改：

如果need_to_send为1，子进程需要向父进程发送执行完毕的返回值

```
    int child = spawn(argv[0], argv);
    if (back_commend == 1) {
        syscall_add_job(child, back_cmd);
    }

    close_all();
    if (child >= 0) {
        int r = ipc_recv(0, 0, 0);              // ②
        if (need_to_send) {
            ipc_send(env->env_parent_id, r, 0, 0);
        }
        wait(child);
    } else {
        debugf("spawn %s: %d\n", argv[0], child);
    }
```

②处ipc_recv对应的ipc_send是：由于`user/lib/libos.c`的libmain函数调用了`main`函数，可以在libmain的结尾通过ipc_send传输`main`的返回值

```
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);
    ipc_send(env->env_parent_id, r, 0, 0);
    // exit gracefully
    exit();
}
```

task3：实现更多指令

准备工作：

通过IPC实现用户进程和文件系统服务进程的交互，实现文件系统服务进程创建文件的接口。

1. 用户：在user/include/fsreq.h中定义结构体

```
enum {
    FSREQ_OPEN,
    FSREQ_MAP,
    FSREQ_SET_SIZE,
    FSREQ_CLOSE,
    FSREQ_DIRTY,
    FSREQ_REMOVE,
    FSREQ_SYNC,
    FSREQ_CREATE,   // add
    MAX_FSREQNO,
};

struct Fsreq_create {
    char req_path[MAXPATHLEN];
```

```
    u_int f_type;
};
```

2. 用户：在user/include/lib.h中声明以下函数

```
//fsipc.c
int fsipc_create(const char*, u_int);
//file.c
int create(const char *path, u_int f_type);
```

在user/lib/fsipc.c中完成`fsipc_creat`函数，与文件系统进程进行ipc通信。
在user/lib/file.c中完成`creat`函数，对`fsipc_creat`进行封装，供用户进程调用。

3. 文件系统：在fs/serv.h中声明

```
int file_create(char *path, struct File **file);
```

在fs/serv.c中完成`serve_create`函数并在serve_table中添加`[FSREQ_CREATE] = serve_create`。

```
void serve_create(u_int envid, struct Fsreq_create *rq) {
    int r;
    struct File *f;
    if ((r = file_create(rq->req_path, &f)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
    f->f_type = rq->f_type; // file_create未设置文件类型，需重新设置
    ipc_send(envid, 0, 0, 0);
}
```

touch

```
//user/touch.c
#include <lib.h>

void usage(void) {
    printf("usage: touch [filename]\n");
    exit();
}

int main(int argc, char **argv) {
    // printf("argc = %d\n", argc);
    if(argc != 2) {
        usage();
        return -1;
    }

    int r = open(argv[1], O_RDONLY);     // 打开文件
    if(r >= 0) {                         // 文件存在
        close(r);
        return 0;
    } else {                             // 不存在则创建
```

```
        if(create(argv[1], FTYPE_REG) < 0) {
            printf("touch: cannot touch \'%s\': No such file or
directory\n", argv[1]);
            return -1;
        }
        return 0;
    }
}
```

mkdir

```
#include <lib.h>
int flag[256];
void usage(void) {
    printf("usage: mkdir <dir>\n\
    -p: no error send if existing \n");
    exit();
}

int main(int argc, char **argv) {
    // printf("brfore argc = %d\n", argc);
    ARGBEGIN {
    case 'p':
        flag[(u_char)ARGC()]++;
        break;
    }
    ARGEND

    // printf("after argc = %d\n", argc);
    if(argc == 0) {
        usage();
        return -1;
    }

    int i = 0;
    int r = open(argv[i], O_RDONLY);    // 打开文件
    if (r >= 0) {                       // 文件存在
        if (!flag['p']) {
            printf("mkdir: cannot create directory \'%s\': File exists\n",
argv[i]);
        }
        close(r);
        return -1;
    } else {                            // 文件不存在
        if(create(argv[i], FTYPE_DIR) < 0) {
            if (!flag['p']) {
                printf("mkdir: cannot create directory \'%s\': No such file
or directory\n", argv[i]);
                return -1;
            }

            // 递归创建文件：循环识别/并尝试打开，打开失败则创建文件
            char path[1024];
            strcpy(path, argv[i]);
```

```
            for (int i = 0; path[i] != '\0'; ++i) {
                if (path[i] == '/') {
                    path[i] = '\0';
                    r = open(path, O_RDONLY);
                    if (r >= 0) {
                        close(r);
                    } else {
                        r = create(path, FTYPE_DIR);
                        if (r < 0) {
                            printf("some err\n");
                        }
                    }
                    path[i] = '/';
                }
            }
            r = create(path, FTYPE_DIR);
            if (r < 0) {
                printf("some err\n");
            }
        }
    }
    return 0;
}
```

rm

`rm` 函数：根据可选项执行相应操作

```
void rm(char *path) {
    struct Stat st;
    if (stat(path, &st) < 0) {      // 获取文件状态
        if (!flag['f']) {
            printf("rm: cannot remove \'%s\': No such file or directory\n",
path);
        }
        return;
    }

    if (st.st_isdir == 1) {         // 文件目录->判断可选项是否满足
        if (flag['r']) {
            if (remove(path) < 0) {
                printf("rm: cannot remove '%s': No such file or
directory\n", path);
                return;
            }
        } else {
            printf("rm: cannot remove '%s': Is a directory\n", path);
            return;
        }
    } else {                        // 文件->直接删除
        if (remove(path) < 0) {
            printf("rm: cannot remove '%s': No such file or directory\n",
path);
            return;
        }
```

```
    }
}
```

主函数：处理可选项及调用rm函数

```
int main(int argc, char **argv) {
    // printf("brfore argc = %d\n", argc);
    ARGBEGIN {
    case 'r':
    case 'f':
        flag[(u_char)ARGC()]++;
        break;
    }
    ARGEND

    // printf("after argc = %d\n", argc);
    if(argc == 0) {
        usage();
        return -1;
    }

    u_int i;
    for (i = 0; i < argc; i++) {
        rm(argv[i]);
    }
    return 0;
}
```

task4：实现反引号

在识别到`之后，提取出反引号内的指令并调用`runcmd()`函数（注：反引号的识别最好放在第一步，并需要在结束之后再次执行一次准备工作）。

```
// user/sh.c
int _gettoken(char *s, char **p1, char **p2) {
    *p1 = 0;
    *p2 = 0;
    // 一些准备工作：判断指令是否为空 跳过空白符 判断是否到达结尾
    if (s == 0) {
        return 0;
    }
    while (strchr(WHITESPACE, *s)) {
        *s++ = 0;
    }
    if (*s == 0) {
        return 0;
    }

    // 识别`
    if (*s == '`') {
        // 读取并执行反引号中的指令
        s++;
```

```
        char cmd[1024];
        int i = 0;
        while (*s && *s != '`') {
            cmd[i++] = *s;
            s++;
        }
        if (s == 0) {
            printf("syntax error: unmatched \"\n");
            exit();
        }
        *s = 0;
        cmd[i] = 0;
        runcmd(cmd);

        // 重复识别到`之前的操作 返回到识别token的准备状态
        if (s == 0) {
            return 0;
        }

        while (strchr(WHITESPACE, *s)) {
            *s++ = 0;
        }
        if (*s == 0) {
            return 0;
        }
    }
    // ... 继续识别token
}
```

task5：实现注释功能

修改`int _gettoken(char *s, char **p1, char **p2)`函数，在识别到'#'后相当于指令已经结束，将该位置设为`\0`后直接返回即可。

```
// user/sh.c
int _gettoken(char *s, char **p1, char **p2) {
    // ...
    if (*s == '#') {
        *s = 0;  // Null-terminate the command before the comment
        return 0;
    }
    // ... 继续识别token
}
```

task6：实现历史指令

history部分定义函数及相关声明

```
// sh.c
void save_history(char *cmd);                // 保存指令
int lookup_history(int op, char* buf);       // 上下键切换指令
int history_print(int argc, char** argv);    // history命令

#define HISTORY_FILE "/.mosh_history"        // 历史记录文件的存储路径
#define HISTFILESIZE 20                      // 存储大小

char history[HISTFILESIZE][1025];            // 存储历史记录的临时数组
int history_count = 0;                       // 记录存储个数
int history_file_fd = -1;                    //
static int current_history_index = -1;       // 当前命令下标
char input_cmd[1025];                        // 控制台当前输入
char input_flag = 0;                         // input_cmd中是否需要更新 0:需要更新
```

具体实现细节

- `void save_history(char *cmd)`

> char * cmd：输入的命令

在读取控制台输入结束后调用save_history即可。

在存储未满时写入文件需要以`O_APPEND`的形式打开，需要先实现一下`O_APPEND`。

```
void save_history(char *cmd) {
    int flag_full = 0;                       // 记录存储是否已满
    if (history_count == HISTFILESIZE) {
        flag_full = 1;
        for (int i = 1; i < HISTFILESIZE; i++) {
            strcpy(history[i - 1], history[i]);
        }
        history_count--;
    }
    strcpy(history[history_count], cmd);
    history_count++;
    current_history_index = history_count - 1;
    input_flag = 0;                          // 完成存储历史记录的临时数组的更新 完成相关管理数据的更新

    if (!flag_full) {
        history_file_fd = open(HISTORY_FILE, O_WRONLY | O_CREAT | O_APPEND);
        if (history_file_fd < 0) {
            debugf("/.mosh_history open in err\n");
            return;
        }
        write(history_file_fd, cmd, strlen(cmd));
        write(history_file_fd, "\n", 1);
        close(history_file_fd);
    } else {
        ftruncate(history_file_fd, 0);
        history_file_fd = open(HISTORY_FILE, O_WRONLY | O_CREAT);
```

```
        if (history_file_fd < 0) {
            debugf("/.mosh_history open in err\n");
            return;
        }
        for (int i = 0; i < history_count; i++) {
            write(history_file_fd, history[i], strlen(history[i]));
            write(history_file_fd, "\n", 1);
        }
        close(history_file_fd);
    }                                           // 写入历史记录文件 注意历史文件中命令是以'\n'结尾 方便打印
}
```

- `int lookup_history(int op, char* buf)`

> op：1代表输入上键 0代表输入下键
>
> buf：当前控制台输入内容的缓冲区
>
> 返回值：完成上下键切换后buf中的指令长度

此函数主要用于辅助实现上下键切换，在sh.c的`readline`函数中添加如下进行调用：

```
void readline(char *buf, u_int n) {
    //...
    if (buf[i] == 27) {
            char tmp;
            buf[i] = 0;             // 将此时控制台输入的命令末尾写入'\0'
            read(0, &tmp, 1);
            if (tmp == '[') {
                read(0, &tmp, 1);
                if (tmp == 'A') { //up
                    i = lookup_history(1, buf);
                }
                if (tmp == 'B') { //down
                    i = lookup_history(0, buf);
                }
            }
            i--;
        }
    // ...
}
```

```
int lookup_history(int op, char* buf) {
    // up:1 down:0
    int flag = 0;                       // flag为1代表第一次进行上下键切换
    if (input_flag == 0) {              // input_flag为0代表需要更新记录当前输入的命令
        input_flag = 1;
        flag = 1;
        memset(input_cmd, 0 ,1024);
        strcpy(input_cmd, buf);
    }

```

```
    int len = strlen(buf);                  // 通过输出左键-空格-左键的方式在控制台删除一个字符(但是好像不能实时显示)
    for (size_t i = 0; i < len; i++) {
        printf("\033[D");
        printf(" ");
        printf("\033[D");
    }

    if (op == 1) {                          // 上键
        printf("%c[B", 27); // 不许乱动
        if (flag == 0) {
            current_history_index = current_history_index - 1 < 0 ? 0 : current_history_index - 1;
        }
        if (current_history_index != -1) {
            len = strlen(history[current_history_index]);
            strcpy(buf, history[current_history_index]);
            buf[len] = 0;
        } else {
            strcpy(buf, input_cmd);
        }
    } else {                                // 下键
        if (current_history_index + 1 > history_count - 1) {
            strcpy(buf, input_cmd);
        }
        else {
            current_history_index = current_history_index + 1;
            len = strlen(history[current_history_index]);
            strcpy(buf, history[current_history_index]);
            buf[len] = 0;
        }
    }
    for (int i = 0; i < len; i++) {     // 由于用户在控制台新输入的指令后续还需要更改 输出到'\0'之前即可
        printf("%c", buf[i]);
    }
    return len;                             // 需要返回buf中的指令长度更新readline中的i
}
```

- `int history_print(int argc, char** argv);`

> argc、argv即命令的参数个数和参数数组
>
> 由于不能自行更改user/include.mk，history指令也应该在sh.c中完成

在sh.c的`runcmd`函数中添加如下进行调用：

```
void runcmd(char *s) {
    // ...
    if(strcmp(argv[0] , "history") == 0) {
        history_print(argc, argv);
        if (rightpipe) {
            wait(rightpipe);
        }
        exit();
    }
    // ...
}
```

直接逐个字符读取打印输出即可（先前存储指令时我们让指令以'\n'结尾 无需考虑换行）

```
int history_print(int argc, char** argv) {
    char buf;
    if (argc != 1)
    {
        printf("usage: history\n");
        exit();
    }

    int r = open(HISTORY_FILE, O_RDONLY);

    if (r < 0)
    {
        printf("open /.mosh_history in err\n");
        exit();
    }

    int fd = r;
    while((r = read(fd, &buf, 1)) == 1)
    {
        printf("%c", buf);
    }
    close(fd);
    return 0;
}
```

task7：实现一行多指令

`#define SYMBOLS "<|>&;()"`中已经定义`;`

修改sh.c中`int parsecmd(char **argv, int *rightpipe)`函数，添加`case ';'`，使fork出的子进程执行已经解析完成的命令，父进程等待子进程执行。

```
    case ';':
            left = fork();
            if (left > 0) {
                wait(left);
                return parsecmd(argv, rightpipe);
            } else {
                return argc;
            }
            break;
```

task8：实现追加重定向

1. 在`int _gettoken(char *s, char **p1, char **p2)`中增加识别`>>`

```
if (*s == '>' && *(s + 1) == '>') {
        *p1 = s;
        *s = 0;
        s += 2;
        *p2 = s;
        return 'A'; // 'A' stands for APPEND
}
```

2. 在`int parsecmd(char **argv, int *rightpipe)`函数，添加`case 'A'`进行相应处理

```
case 'A':
            if (gettoken(0, &t) != 'w') {
                debugf("syntax error: > not followed by word\n");
                exit();
            }

            if((fd = open(t, O_RDONLY)) < 0) {
                fd = open(t, O_CREAT);
                if (fd < 0) {
                    debugf("error in open %s\n", t);
                    exit();
                }
            }
            close(fd);                  // 如果不存在文件创建文件

            if ((fd = open(t, O_WRONLY | O_APPEND)) < 0) {
                debugf("syntax error: >> followed the word: %s cannot
open\n", t);
                exit();
            }
            dup(fd, 1);
            close(fd);                  // 将文件以O_WRONLY | O_APPEND形式
打开进行写入

            break;
```

task9：实现引号支持

在`int _gettoken(char *s, char **p1, char **p2)`中增加识别`\"`，将引号中的内容以`'w'`的类型返回即可。

```
    if (*s == '"') {
         s++;
         *p1 = s;
         while (*s && *s != '"') {
             s++;
         }
         if (s == 0) {
             printf("syntax error: unmatched \"\n");
             exit();
         }
         *s = 0;
         s++;
         *p2 = s;
         return 'w';
     }
```

task10：实现前后台任务管理

> 考虑：如果是后台指令，在每次在执行结束销毁进程之前需要把它的status由JOB_RUNNING变为JOB_DONE，而销毁进程属于系统调用的指令，所以可以将对job的相关管理放在内核态中，由用户态通过系统调用进行。

准备工作

在`int parsecmd(char **argv, int *rightpipe)`函数，添加`case '&'`实现指令在后台处理

设立back_commend标志当前进程执行的是后台指令，与实现一行多指令相比父进程无须等待子进程

```
    case '&':
            left  = fork();
            if (left > 0) {
                return parsecmd(argv, rightpipe);
            } else {
                back_commend = 1;
                return argc;
            }
            break;
```

job管理

由于把对job的相关管理放在内核，需要实现一系列的系统调用

1. kern/syscall_all.c中相关结构体和函数

```c
#define MAXJOBS 128
#define JOB_RUNNING 1
#define JOB_DONE 0

typedef struct {
    int job_id;
    int env_id;
    int status;
    char cmd[1024];
} job_t;

job_t jobs[MAXJOBS];
int next_job_id = 1;
```

按照实现系统调用的一般方式

kern/syscall_all.c实现具体函数并在syscall_table中添加-> include/syscall.h添加枚举类型 -> user/lib/syscall_lib.c中包装msyscall->user/include/lib.h声明

需要为用户提供如下系统调用接口：

```c
// user/include/lib.h
// job
void syscall_add_job(u_int envid, char *cmd);
void syscall_list_jobs();
void syscall_kill_job(int job_id);
int syscall_find_job_envid(int job_id);
```

正常执行结束的后台指令的状态设置直接在sys_env_destroy函数中进行：

```c
int sys_env_destroy(u_int envid) {
    struct Env *e;
    try(envid2env(envid, &e, 1));
    for (int i = 0; i < MAXJOBS; i++) {
        if (jobs[i].env_id == e->env_id) {
            jobs[i].status = 0;
            // printk("%d set to done\n", envid);
            break;
        }
    }
    printk("[%08x] destroying %08x\n", curenv->env_id, e->env_id);
    env_destroy(e);
    return 0;
}
```

相关调用

- `jobs`

  在sh.c的`runcmd`函数中添加如下进行调用：

```
void runcmd(char *s) {
    // ...
    if(strcmp(argv[0] , "jobs") == 0) {
        syscall_list_jobs();
        if (rightpipe) {
            wait(rightpipe);
        }
        exit();
    }
    // ...
}
```

- `kill`

  在sh.c的`runcmd`函数中添加如下进行调用：

```
void runcmd(char *s) {
    // ...
    if(strcmp(argv[0] , "kill") == 0) {
        int job_id = 0;
        for (int i = 0; argv[1][i]; i++) {
            job_id = 10 * job_id + argv[1][i] - '0';
        }
        syscall_kill_job(job_id);
        if (rightpipe) {
            wait(rightpipe);
        }
        exit();
    }
    // ...
}
```

- `fg`

  在sh.c的`runcmd`函数中添加如下进行调用：

```
void runcmd(char *s) {
    // ...
    if(strcmp(argv[0] , "fg") == 0) {
        int job_id = 0;
        for (int i = 0; argv[1][i]; i++) {
            job_id = 10 * job_id + argv[1][i] - '0';
        }
        int env_id = syscall_find_job_envid(job_id);
        if (env_id >= 0) {
            wait(env_id);
        }
        if (rightpipe) {
            wait(rightpipe);
        }
        exit();
    }
    // ...
}
```

- 添加job

```
void runcmd(char *s) {
    char *cmd_ptr = s;      // 记录输入的首地址
    char cmd[1025];
    strcpy(cmd, s);         // 拷贝保存控制台输入
    // ...
    char back_cmd[1024];
    if (back_commend == 1) { // 如果是后台指令 下一步拆分得到具体指令内容
        int begin_index = argv[0] - cmd_ptr;    // 指令开始的下标
        int end_index = 1024;
        for (int i = argv[argc - 1] - cmd_ptr + strlen(argv[argc - 1]);
i < 1024; i++) {
            if (cmd[i] == '&') {
                end_index = i + 1;              // 指令结束的下标
                break;
            }
        }
        strcpy(back_cmd, cmd + begin_index);
        back_cmd[end_index - begin_index] = 0;
    }

    int child = spawn(argv[0], argv);
    if (back_commend == 1) {
        syscall_add_job(child, back_cmd);       // 通过系统调用添加job
    }
    // ...
}
```