Shell 挑战性任务

实现不带 `.b` 后缀指令

你需要实现不带 `.b` 后缀的指令，但仍需兼容带有 `.b` 后缀的指令，如 `ls` 与 `ls.b` 都应能够正确列出当前目录下的文件。

- 一条命令的结构是`[命令名][空格][参数*]`。观察到 `spawn` 函数会解析每一个 `argument` ，于是在 `spawn` 函数一开头便检测 `.b` 。
  考虑到有的编译不一定以 `.b` 结尾，那么只需要将原 `argument` 先解析一遍，成功则继续执行；如果 fd 返回错误，则对该 `argument` + `.b` 再解析一遍即可，成功则继续执行；如果都不成功说明未编译。
- 实现如下：

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

实现指令条件执行

你需要实现 Linux shell 中的 `&&` 与 `||`。 对于 `command1 && command2`，`command2` 被执行当且仅当 `command1` 返回 0；对于 `command1 || command2`，`command2` 被执行当且仅当 `command1` 返回非 0 值。

> 注: 评测中保证不出现括号。并且需要注意的是，在 bash 中 `&&` 与 `||` 的优先级相同，按照从左到右的顺序求值。
>
> 例如 `cmd1 || cmd2 && cmd3`，若 `cmd1` 返回 0，则 `cmd1` 执行后 `cmd2` 不会被执行，`cmd3` 会被执行；若 `cmd1` 返回非 0 且 `cmd2` 返回非 0，则 `cmd3` 将不会被执行。
>
> 提示：你可能需要修改 MOS 中对用户进程 `exit` 的实现，使其能够返回值。

- 首先，这个要求需要我们命令结束的时候返回值来判断下一个指令是否实现。我的处理如下：由于想让 `exit` 函数返回值，则得变更较多的指令（原exit为void），于是我打算用通信来解决。执行指令仍然类同 `;` 的处理，子进程执行**左边**，父进程执行**右边**，唯一的区别是父进程需要接收到**子进程的返回值**，从而判断需不需要继续执行。
- 在同一函数里其实可以轻松获得父子关系的envid。但是再例如 user_panic 这种函数里面，可能无法找到父子的envid。往往来说只能通过现在调用这个函数的进程，从而去使用**系统调用**获得父进程id。故而得实现一个系统调用。

代码如下：

- 首先处理系统调用，从而能得到当前进程的父进程id：

```
// include/syscall.h
enum {
    ...添加
    SYS_get_parent_envid,
    MAX_SYSNO,
}

// user/include/lib.h
...添加
int syscall_get_parent_envid(u_int envid);
...

// kern/syscall_all.c
...实现
int sys_get_parent_envid(u_int envid){
    struct Env *env;
    try(envid2env(envid, &env, 0));
    return env->env_parent_id;
}

void *syscall_table[MAX_SYSNO] = {
    ...添加
    [SYS_get_parent_envid] = sys_get_parent_envid,
}

// user/lib/syscall_lib.c
...实现系统调用
int syscall_get_parent_envid(u_int envid)
{
    return msyscall(SYS_get_parent_envid, envid);
}
...
```

- 然后实现条件指令：

  实现返回值：

```
// user/lib/debugf.c
void _user_panic(const char *file, int line, const char *fmt, ...) {
    debugf("panic at %s:%d: ", file, line);
    va_list ap;
    va_start(ap, fmt);
```

```
    vdebugf(fmt, ap);
    va_end(ap);
    debugf("\n");
    // 在exit前面向父进程发送返回值
    int parent_id = syscall_get_parent_envid(0);
    ipc_send(parent_id, 1, 0, 0);
    exit();
}

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

    // 在exit前面向父进程发送main函数执行的返回值，需要对烧录文件报错返回1，正常退出返回
0，这里不再赘述更改方式，过于简单。
    int recv = main(argc, argv);
    int recv = main(argc, argv);
    int parent_id = syscall_get_parent_envid(0);
    if (syscall_get_env_status(parent_id) == ENV_NOT_RUNNABLE ||
syscall_get_env_status(parent_id) == ENV_RUNNABLE)
        ipc_send(parent_id, recv, 0, 0);
    // exit gracefully
    exit();
}
```

实现词法解析：

```
// user/sh.c
    int _gettoken(char *s, char **p1, char **p2) {
        //...
        if (*s == '&' && *(s + 1) == '&')
        {
            *p1 = s;
            *s++ = 0;
            *s++ = 0;
            *p2 = s;
            return 'A'; // AND
        }
        if (*s == '|' && *(s + 1) == '|')
        {
            *p1 = s;
            *s++ = 0;
            *s++ = 0;
            *p2 = s;
            return 'O'; // OR
        }
        //...
    }

    int condition_flag = 0; //标识出是不是条件指令
    int parsecmd(char **argv, int *rightpipe) {
        //...
        while (1) {
            condition_flag = 0; //每次循环得刷新
```

```
            //...
            case 'A':;
            fktmp = fork();
            if (fktmp == 0) {
                condition_flag = 1; //子进程标注：是条件指令
                return argc;
            } else {
                int son;
                int recv = ipc_recv(&son, 0, 0);    //父进程接收子进程的返回值
                if(recv){//!0 && cmd2(不执行)
                    int next_cond = gettoken(0, &t);
                    if (next_cond == 0) {
                        return 0;
                    } else if (next_cond == 'o') {//!0 && cmd2 || cmd3(执行)
                        return parsecmd(argv, rightpipe);
                    }
                }else{//0 && cmd2
                    return parsecmd(argv, rightpipe);
                }
            }
            break;
        case 'o':;
            fktmp = fork();
            if (fktmp == 0) {
                condition_flag = 1; //子进程标注：是条件指令
                return argc;
            } else {
                int son;
                int recv = ipc_recv(&son, 0, 0);    //父进程接收子进程的返回值
                if(recv == 0){//0 || cmd2(不执行)
                    while (1)
                    {
                        int next_cond = gettoken(0, &t);
                        if (next_cond == 0)
                        {
                            return 0;
                        }
                        else if (next_cond == 'A')
                        { // 0 || cmd2 && cmd3(执行)
                            return parsecmd(argv, rightpipe);
                        }
                    }
                } else {//!0 || cmd2
                    return parsecmd(argv, rightpipe);
                }
            }
            break;
        //...
        }
    }
    void runcmd(char *s){
        argv[argc] = 0;
        int child = spawn(argv[0], argv);
        if (child >= 0) {
            int son;
```

```
            int recv = ipc_recv(&son, 0, 0);     //接受子进程执行命令的返回值
            //printf("recv,ok: %d\n",recv);
            if (condition_flag) {
                int parent_id = syscall_get_parent_envid(0);
                ipc_send(parent_id, recv, 0, 0);     //发送给自己的父进程，也就是右
边
            }
            close_all();     //关闭文件得放在他们之后，否则无法发送ipc信号。
        } else {
            debugf("spawn %s: %d\n", argv[0], child);
        }
        //...
    }
```

实现更多指令

你需要实现 `touch`，`mkdir`，`rm` 指令，只需要考虑如下情形：

- `touch`：

> - `touch <file>`：创建空文件 `file`，若文件存在则放弃创建，正常退出无输出。 若创建文件的父目录不存在则输出 `touch: cannot touch '<file>': No such file or directory`。 例如 `touch nonexistent/dir/a.txt` 时应输出 `touch: cannot touch 'nonexistent/dir/a.txt': No such file or directory`。

- `mkdir`：

> - `mkdir <dir>`：若目录已存在则输出 `mkdir: cannot create directory '<dir>': File exists`，若创建目录的父目录不存在则输出 `mkdir: cannot create directory '<dir>': No such file or directory`，否则正常创建目录。
> - `mkdir -p <dir>`：当使用 `-p` 选项时忽略错误，若目录已存在则直接退出，若创建目录的父目录不存在则递归创建目录。

- `rm`：

> - `rm <file>`：若文件存在则删除 `<file>`，否则输出 `rm: cannot remove '<file>': No such file or directory`。
> - `rm <dir>`：命令行输出: `rm: cannot remove '<dir>': Is a directory`。
> - `rm -r <dir>|<file>`：若文件或文件夹存在则删除，否则输出 `rm: cannot remove '<dir>|<file>': No such file or directory`。
> - `rm -rf <dir>|<file>`：如果对应文件或文件夹存在则删除，否则直接退出。

- 按照题目要求逐步实现即可。`mkdir、rm` 的实现较为复杂，将详细说明。

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

void main(int argc, char **argv)
```

```
{
    if (argc != 2)
    {
        printf("usage: touch [filename]\n");
        return;
    }
    int r = open(argv[1], O_RDONLY); // 打开文件
    if (r >= 0)
    {
        close(r);
        return;
    }
    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;
        }
    }
}
```

```
// mkdir.c
#include <lib.h>

void create_directory(char *path, int recursive);

void main(int argc, char **argv)
{
    if (argc != 2 && !(argc == 3 && strcmp(argv[1], "-p") == 0))    //判定参数
个数
    {
        printf("usage: mkdir [-p] [dirname]\n");
        return;
    }

    int recursive = 0;
    char *dir;

    if (argc == 3)  //只有3个参数的情况下才会recursive
    {
        recursive = 1;
        dir = argv[2];
    }
    else    //2个参数的情况下不需要recursive
    {
        dir = argv[1];
    }

    create_directory(dir, recursive);   //根据解析不同情况分类讨论
}

void create_directory(char *path, int recursive)
{
```

```
    int r = open(path, O_RDONLY);   //打开目录

    if(recursive){  //-p递归情况
        if(r >= 0){
            close(r);
            return;
        }
        else
        {
            char *p;
            for (p = path; *p; p++) //逐步解析输入的路径
            {
                if (*p == '/')  //只有碰到分割，才执行操作
                {
                    *p = 0; //切割第一个路径
                    if (create(path, FTYPE_DIR) < 0)    //创建父根
                    {

                        int err = open(path, O_RDONLY); //若创建失败，打开这个文
件看状况
                        if (err < 0)
                        {
                            printf("mkdir: cannot create directory '%s': No
such file or directory\n", path);
                            return;
                        }
                        close(err);
                    }
                    *p = '/';   //还原分割符
                }
            }
            // 在以上的分割操作下，已经将path的父目录全部递归构建完成。
            if (create(path, FTYPE_DIR) < 0)    // 正常创建path目录
            {
                printf("mkdir: cannot create directory '%s': No such file or
directory\n", path);
            }
        }
    }
    else//正常情况
    {
        if (r >= 0) //存在了则报错退出
        {
            printf("mkdir: cannot create directory '%s': File exists\n",
path);
            close(r);
            return;
        }
        else
        {
            if (create(path, FTYPE_DIR) < 0)    //直接进行create，会判定有没有父
目录
            {
                printf("mkdir: cannot create directory '%s': No such file or
directory\n", path);
            }
```

```
        }
    }
}
```

```
// rm.c
#include <lib.h>

void remove_path(char *path, int recursive, int force);

void main(int argc, char **argv)
{
    if (argc < 2)   //判定参数个数
    {
        printf("usage: rm [-r|-rf] [file|dir]\n");
        return;
    }

    int recursive = 0;
    int force = 0;
    char *path;

    if (argc == 3 && strcmp(argv[1], "-r") == 0)    //3个参数情况下会分-r和-rf
    {
        recursive = 1;
        force = 0;
        path = argv[2];
    }
    else if (argc == 3 && strcmp(argv[1], "-rf") == 0)
    {
        recursive = 1;
        force = 1;
        path = argv[2];
    }
    else if (argc == 2) //2个参数情况下只有r和f均为0的情况
    {
        recursive = 0;
        force = 0;
        path = argv[1];
    }
    else    //判定参数个数
    {
        printf("usage: rm [-r|-rf] [file|dir]\n");
        return;
    }

    remove_path(path, recursive, force);    //根据解析不同情况分类讨论
}

void remove_path(char *path, int recursive, int force)
{
    int fd;
    struct Stat path_stat;
    if(recursive == 0 && force == 0){
        if ((fd = open(path, O_RDONLY)) < 0){   // 如果失败则表示文件或目录不存在
```

```
            printf("rm: cannot remove '%s': No such file or directory\n",
path);
            return;
        }
        close(fd);
        stat(path, &path_stat);
        if (path_stat.st_isdir) // 如果路径是目录，打印错误信息
        {
            printf("rm: cannot remove '%s': Is a directory\n", path);
            return;
        }
        remove(path);           // 删除文件
    }
    else if (recursive == 1 && force == 0){ // 如果递归删除且不强制删除
        if ((fd = open(path, O_RDONLY)) < 0)
        {
            printf("rm: cannot remove '%s': No such file or directory\n",
path);
            return;
        }
        close(fd);
        remove(path);           // 删除文件
    }
    else if (recursive == 1 && force == 1){ // 如果递归删除且强制删除
        if ((fd = open(path, O_RDONLY)) < 0)    //无报错
        {
            return;
        }
        close(fd);
        remove(path);           // 删除文件
    }
}
```

实现反引号

你需要使用反引号实现指令替换。只需要考虑 `echo` 进行的输出，你需要将反引号内指令执行的所有标准输出替换为 `echo` 的参数。例如：

```
echo `ls | cat | cat | cat`
```

- 我的想法是在 runcmd 函数里单独处理 ` 的情况。可以想到的是，任何一个 ` 都会有前者对应，那么依据 strchr来找到一头一尾，将其包含的字符串用子进程再runcmd一下，再用管道输出到父进程，替换到原来用 `` 包含的字符串，就可以完成反引号了。
- 需要注意的是，可能出现 `"string"` 和 "`string`" 的这种情况。前者是先执行反引号，后者先执行引号。

代码如下：

```
// sh.c
    char *start = strchr(s, '`');
    while (start)
```

```
    {
        int flag = 0;
        for (int ii = 0; ii < start - s; ii++)
        {
            if(s[ii] == '"'){   //检查前面有没有 " 的存在，若有则得退出，不执行 `。
                flag = 1;
                break;
            }
        }
        if(flag == 1){
            break;
        }
        char *end = strchr(start + 1, '`'); //找到下一个 `
        if (!end)
        {
            debugf("syntax error: unmatched `\n");
            return;
        }

        *end = '\0';
        char subcommand[4096] = {0};
        strcpy(subcommand, start + 1);      //找到 `` 包裹的字符串
        *end = '`';

        char output[4096] = {0};


        int p[2], r;

        if ((r = pipe(p)) != 0){
            exit();
        }

        if ((r = fork()) < 0){
            exit();
        }
        if (r == 0){
            close(p[0]);
            dup(p[1], 0);
            close(p[1]);
            runcmd(subcommand);             //执行 subcommand
            exit();
        }
        else
        {
            close(p[1]);
            int n = 0;
            int offset = 0;

            while ((n = read(p[0], output + offset, 4095 - offset)) > 0)
            {
                offset += n;
            }

            if (offset >= 0)
            {
```

```
                output[offset] = '\0'; // 添加字符串终止符
            }
            //管道输出到父进程
            else
            {
                printf("error in ` when piping.\n");
                exit();
            }
            close(p[0]);
        }

        char newcmd[4096] = {0};
        *start = 0;
        strcpy(newcmd, s);
        *start = '`';
        newcmd[start - s] = '\0';   //父进程将 ` 替换

        strcpy(newcmd + (start - s), output);
        strcpy(newcmd + strlen(newcmd) , end + 1);


        strcpy(s, newcmd);          //s为更新后 ` 执行完的命令。
        start = strchr(s, '`');
    }
```

实现注释功能

你需要使用 `#` 实现注释功能，例如 `ls | cat # this is a comment meow`，`ls | cat` 会被正确执行，而后面的注释则会被抛弃。

- 想法也是在 runcmd 中处理。十分简单，找到了 # 后直接将其置0，那么就不会解析、执行后面的部分了。

代码如下：

```
// sh.c
    char *comment = strchr(s, '#');
    if (comment)
    {
        *comment = '\0';
    }
```

实现历史指令

你需要实现 shell 中保存历史指令的功能，可以通过 Up 和 Down 选择所保存的指令并执行。你需要将历史指令保存到根目录的 `.mosh_history` 文件中（一条指令一行），为了评测的方便，我们设定 `$HISTFILESIZE=20`（bash 中默认为 500），即在 `.mosh_history` 中至多保存最近的 20 条指令。你还需要支持通过 `history` 命令输出 `.mosh_history` 文件中的内容。

> 注：在 bash 中，`history` 为 shell built-in command，我们规定需要将 `history` 实现为 built-in command。

你需要将当前执行的指令先存入 `.mosh_history` 中，例如：

```
echo `ls | cat`
echo meow # comment
history
history | cat
```

当历史指令为空时，依次执行上述四条指令后，后两条指令会分别输出

```
echo `ls | cat`
echo meow # comment
history
```

与

```
echo `ls | cat`
echo meow # comment
history
history | cat
```

使用 Up 能够切换到上一条指令（如果上一条指令存在），使用 Down 能够切换到下一条指令（如果下一条指令存在）。能够选择的指令范围为：用户当前输入的指令与 `.mosh_history` 文件中保存的所有指令。例如在执行了上述四条指令后，用户输入了 `echo`，此时 Up 应该将指令切换至 `history | cat`，再次进行三次 Up 后切换至 `echo 'ls | cat'`，此时再次 Up 应保留在该指令（因为已经不存在上一条指令）；再进行四次 Down 后将切换回 `echo`，此时再次 Down 应保留在该指令（因为不存在下一条指令）。

- 由于要将history设置为内置指令，于是我将history的实现均放在了 sh.c 中，并且考虑用一个全局数组来记录每一个历史的实现。

```
// sh.c
//在 runcmd 中更改，使得history指令能输出历史指令。

//处理history
    char oldchar = s[7];
    s[7] = 0;
    if (strcmp(s, "history") == 0){
        s[7] = oldchar;
        char oldcmd[4096] = {0};
        strcpy(oldcmd, s);
        s[7] = 0;

        strcpy(s, his_cmd); //char his_cmd[MAX_CMD_LENGTH] = "cat .mosh_history
";
        strcpy(s + strlen(s), oldcmd + 7);  //以上操作直接输出.mosh_history里的内容，
灵活的解决了管道问题。
    } else {
        s[7] = oldchar;
```

```
    }
```

- 并且上下键关乎于 readline 指令，故而在 readline 中执行上下键逻辑。

```
// sh.c
#define UP 'A'
#define DOWN 'B'
#define HISTFILE ".mosh_history"
#define MAX_HISTORY 20

int his_count = 0;                          // 历史命令总数
int now_index = 0;                          // 当前命令索引
char history[MAX_HISTORY][MAX_CMD_LENGTH]; // 历史命令数组
char tmp_save[MAX_CMD_LENGTH] = {0};

void readline(char *buf, u_int n) {
    int r;
    for (int i = 0; i < n; i++) {
        if (buf[i] == '\033')
        { // 处理方向键
            buf[i] = 0;
            if ((r = read(0, buf + i, 1)) != 1){
                if (r < 0){
                    debugf("read error: %d\n", r);
                }
                exit();
            }
            if (buf[i] != '['){
                continue;
            }
            buf[i] = 0;
            i++;
            if ((r = read(0, buf + i, 1)) != 1)
            {
                if (r < 0){
                    debugf("read error: %d\n", r);
                }
                exit();
            }
            if ((buf[i] != UP && buf[i] != DOWN)){
                continue;
            }
            //以上将上下键给分离开来，否则退出，执行正常的readline

            int flag = (buf[i] == UP) ? 0 : 1;
            buf[i] = 0;
            i++;
            if (strlen(buf) > 0 && now_index == his_count)
            {
                strcpy(tmp_save, buf);
            }
             //printf("now:%d his: %d", now_index,his_count);
            if (flag == 0 && now_index > 0)
            { // 上键
                now_index--;
```

```
                // 清除当前行
                printf("\033[B\033[4096D\033[K");
                // 显示历史命令
                strcpy(buf, history[now_index]);
                printf("$ %s", buf);
                i = strlen(buf) - 1;
            }
            else if (flag == 0 && now_index == 0)   //到顶了，仅仅显示最上方的一行即可
            {
                printf("\033[B\033[4096D\033[K");
                printf("$ %s", buf);
                i = strlen(buf) - 1;
            }
            else if (flag == 1 && now_index < his_count)
            { // 下键
                now_index++;
                if (now_index < his_count){ //显示历史记录
                    // 清除当前行
                    printf("\033[4096D\033[K");
                    // 显示历史命令
                    strcpy(buf, history[now_index]);
                    printf("$ %s", buf);
                    i = strlen(buf) - 1;
                }else{  //显示目前的保留行，它并不参与到history的记录之中，除非他被执行。
                    printf("\033[4096D\033[K");
                    strcpy(buf, tmp_save);
                    printf("$ %s", buf);
                    i = strlen(buf) - 1;
                }
            }
            else if (flag == 1 && now_index == his_count)   //到底了，仅显示用户输入
的一行
            {
                printf("\033[4096D\033[K");
                strcpy(buf, tmp_save);
                printf("$ %s", buf);
                i = strlen(buf) - 1;
            }
        }
        if (buf[i] == '\r' || buf[i] == '\n') {
            buf[i] = 0;

            if (strlen(buf) > 0)
            {
                // 保存当前命令到历史记录中
                if (his_count < MAX_HISTORY)
                {
                    strcpy(history[his_count], buf);
                    his_count++;
                }
                else
                {
                    // 如果历史记录已满，移除最旧的命令
                    for (int j = 1; j < MAX_HISTORY; j++)
                    {
                        strcpy(history[j - 1], history[j]);
```

```
                    }
                    strcpy(history[MAX_HISTORY - 1], buf);
                }
                now_index = his_count;

                // 将历史命令写入.mosh_history
                int fd;
                if ((fd = open("/.mosh_history", O_RDWR | O_CREAT)) < 0) {
                    user_panic("open ./mosh_history: %d", fd);
                }
                int n = 0;
                char inputFileHis[MAX_CMD_LENGTH * 10] = {0};
                for (int k = 0; k < his_count; k++){
                    strcpy(inputFileHis + strlen(inputFileHis), history[k]);
                    strcpy(inputFileHis + strlen(inputFileHis), "\n");
                }
                if ((n = write(fd, inputFileHis, strlen(inputFileHis))) < 0)
                {
                    user_panic("write /.mosh_history: %d", n);
                }
            }
            tmp_save[0] = 0;
            return;
        }
    }
    debugf("line too long\n");
    while ((r = read(0, buf, 1)) == 1 && buf[0] != '\r' && buf[0] != '\n') {
        ;
    }
    buf[0] = 0;
}
```

实现一行多指令

你需要实现使用 `;` 将多条指令隔开从而从左至右依顺序执行每条指令的功能。例如：

```
ls;ls | cat; echo nihao,mosh; echo `ls; echo meow`
```

- 想法上来说就是解析词法的时候拿出 `;` ，然后在 `parsecmd` 中处理它。只需要子进程处理分号左边的任务，父进程**再等待子进程**处理完后继续解决下面的指令；以此类推即可。

代码如下：

```
// sh.c
    case ';':
            fktmp = fork();
            if (fktmp)
            {
                wait(fktmp);
                return parsecmd(argv, rightpipe);
            }
            else
            {
                return argc;
            }
            break;
```

实现追加重定向

你需要实现 shell 中 `>>` 追加重定向的功能，例如：

```
ls >> file1; ls >> file1
```

最后文件 `file1` 中将会有两次 `ls` 指令的输出。

- 首先，再解决词法的过程中，得单独分析出 `>>` 的组合。只需要在 `SYMBOLS` 的基础上找到 `>`，再判断它后面是否又跟着一个 `>` 即可。
  代码如下：

  ```
  // sh.c
  if (strchr(SYMBOLS, *s)) {
          if (*s == '>' && *(s + 1) == '>'){
              *p1 = s;
              *s++ = 0;
              *s++ = 0;
              *p2 = s;
              return 'a'; // 'a' for append redirection
          }
      //...
  }
  ```

- 然后，在 `parsecmd` 中解决 append 类型。这里可以参照已经写过的 `>`，故而实现起来不难，但是得实现一个 O_APPEND 来解决增加问题：
  代码如下：

  ```
  // sh.c
      case 'a': // Handle append redirection
              if (gettoken(0, &t) != 'w')
              {
                  debugf("syntax error: >> not followed by word\n");
                  exit();
              }
  ```

```
            if ((r = open(t, O_WRONLY | O_CREAT | O_APPEND)) < 0)   //唯一区别是把原来的 O_TRUNC 改为 O_APPEND
            {
                debugf("open error: cannot open '%s'\n", t);
                exit();
            }

            dup(r, 1);
            close(r);
            break;
```

- 实现 O_APPEND：首先去给 O_APPEND 在 lib.h 中分配一个值（未出现即可，我使用的是 0x2000）；然后，在open函数中实现打开情况为 `APPEND` 的时候，文件描述符中的位移距离的设置，设置到文件末端。
  代码如下：

```
// lib/file.c
//在代码末端，返回的前面加上这样一段话
    if (mode & O_APPEND)
    {
        fd->fd_offset = ffd->f_file.f_size; //将fd的offset设置为文件的大小，从而指针移动到文件末端。
    }
```

实现引号支持

你需要实现引号支持，比如 `echo "ls >"`，shell 在解析时需要将双引号内的内容看作是单个字符串。

- 想法上是在词法解析的时候单独找出来 `"` ，然后将其包裹的一整个字符串当作一个 argument 即可。那么就需要在找到第一个 `"` 的时候先设置一个 `0`；从他开始往后遍历找到下一个 `"` ，然后再设置为 `0`。将其跳转到 `'w'` 只写状态即可。

代码如下：

```
// sh.c
    if (*s == '"'){
        *s = 0;
        s++;
        *p1 = s;
        while (*s && (*s != '"')){
            s++;
        }
        *s++ = 0;
        *p2 = s;
        return 'w';
    }
```

实现前后台任务管理

- 你需要支持 mosh 运行后台进程，当命令的末尾添加上 `&` 符号时，该命令应该在后台执行。
- 实现 `jobs` 指令列出当前 shell 中所有后台任务的状态。你需要为任务创建 ID（每次启动 mosh 时，任务从 1 开始编号，每个新增任务编号应加 1），并且通过 `jobs` 指令输出包括：任务 ID（`job_id`）、任务的运行状态（`status`：可能的取值为 `Running`，`Done`）、任务的进程 ID（`env_id`）与运行任务时输入的指令（`cmd`）。请以 `printf("[%d] %-10s 0x%08x %s", job_id, status, env_id, cmd)` 的格式进行输出。
- 实现 `fg` 将后台任务带回前台继续运行，用户通过 `fg <job_id>` 的方式将对应任务带回前台。
- 实现 `kill` 指令，用户通过 `kill <job_id>` 来实现结束后台任务。

在 `fg` 或 `kill` 指令中，若 `job_id` 对应的后台任务不存在则输出 `printf("fg: job (%d) do not exist\n", job_id)`，若 `job_id` 对应的 ID 为 `envid` 的进程状态不为 `Running` 则输出 `printf("fg: (0x%08x) not running\n", envid)`。

例如：

```
sleep 10&
sleep 60 &
jobs
# wait for about 10 seconds...
jobs
```

依次执行上述指令，则第一个 `jobs` 应输出（其中进程 ID 的值可能与你本地运行的输出结果不同）：

```
[1] Running    0x00003805 sleep 10&
[2] Running    0x00005006 sleep 60 &
```

第二个 `jobs` 应输出：

```
[1] Done       0x00003805 sleep 10&
[2] Running    0x00005006 sleep 60 &
```

- 首先，全局定义 `jobs` 的基本属性以及方法。按照题目的要求书写即可，难度不大，解释包含在注释之中，实现如下：

```c
#define MAXJOBS 128

typedef struct
{
    int job_id;     // 作业ID
    int env_id;     // 进程ID
    char cmd[4096]; // 命令字符串
    int status; // 0: Running, 1: Done
} job_t;

job_t jobs[MAXJOBS]; // 存储所有作业的数组
int job_count = 0;   // 当前作业计数
int next_job_id = 1; // 下一个作业的ID
```

```
// 添加新作业
void add_job(int env_id, char *cmd)
{
    int flag = 0; // 标志是否为后台运行
    char *s = strchr(cmd, '&'); // 查找命令中的'&'字符
    if (s)
    {
        if (*s == '&' && *(s + 1) != '&')
        {
            flag = 1; // 是后台运行
        }
    }

    if (job_count < MAXJOBS && flag) // 如果作业数未达到上限且为后台运行
    {
        jobs[job_count].job_id = next_job_id++; // 分配作业ID
        jobs[job_count].env_id = env_id; // 设置环境ID
        strcpy(jobs[job_count].cmd, cmd); // 复制命令字符串
        jobs[job_count].status = 0; // 设置作业状态为运行中
        job_count++; // 增加作业计数
    }
}

// 列出所有作业
void list_jobs()
{
    for (int i = 0; i < job_count; i++)
    {
        if (jobs[i].status == 0) // 如果作业正在运行
        {
            // 检查作业状态是否已改变
            if (syscall_get_env_status(jobs[i].env_id) != ENV_NOT_RUNNABLE &&
syscall_get_env_status(jobs[i].env_id) != ENV_RUNNABLE)
            {
                jobs[i].status = 1; // 设置作业状态为已完成
            }
        }
        // 打印作业信息
        printf("[%d] %-10s 0x%08x %s\n", jobs[i].job_id, jobs[i].status == 0
? "Running" : "Done", jobs[i].env_id, jobs[i].cmd);
    }
}

// 杀死指定作业
void kill_job(char job_id_str[])
{
    int job_id = string_to_int(job_id_str); // 将字符串转换为整数作业ID
    for (int i = 0; i < job_count; i++)
    {
        if (jobs[i].job_id == job_id)
        {
            if (jobs[i].status != 0)
            {
                printf("fg: (0x%08x) not running\n", jobs[i].env_id);
                return;
            }
```

```
            syscall_kill_env(jobs[i].env_id); // 杀死作业
            jobs[i].status = 1; // 设置作业状态为已完成
            return;
        }
    }
    printf("fg: job (%d) do not exist\n", job_id); // 如果作业不存在，打印错误信息
}

// 将作业置于前台
void fg_job(char job_id_str[])
{
    int job_id = string_to_int(job_id_str); // 将字符串转换为整数作业ID
    for (int i = 0; i < job_count; i++)
    {
        if (jobs[i].job_id == job_id)
        {
            if (jobs[i].status != 0)
            {
                printf("fg: (0x%08x) not running\n", jobs[i].env_id);
                return;
            }
            jobs[i].status = 1; // 将作业状态设置为已完成
            return;
        }
    }
    printf("fg: job (%d) do not exist\n", job_id); // 如果作业不存在，打印错误信息
}
```

- 其中有两个系统调用：`syscall_kill_env(jobs[i].env_id);` 和 `(syscall_get_env_status(jobs[i].env_id)`。由于它们的实现实在是雷同上文的 `syscall_get_env_status`，故而不再赘述。基本思路是：
  - 在 $user/include/lib.h$ 中添加：

    `void syscall_func(u_int envid, u_int value, const void *srcva, u_int perm);`
  - 在 `user/lib/syscall_lib.c` 中添加：

    ```
    void syscall_func(u_int envid, u_int value, const void *srcva, u_int
    perm) {
        msyscall(SYS_func, envid, value, srcva, perm);
    }
    ```
  - 在 $include/syscall.h$ 中的 `enum` 的 `MAX_SYSNO` 前面加上 `SYS_func`
  - 在 `kern/syscall_all.c` 的 `void *syscall_table[MAX_SYSNO]` 的**最后**加上 `[SYS_func] = sys_func`
  - 在 `kern/syscall_all.c` 的 `void *syscall_table[MAX_SYSNO]` 的**前面**具体编写实现函数
- 由于要实现为内置命令，而且我**不希望** jobs 相关的指令被**父子进程干扰**（否则无法在主进程中对 jobs 表更新），故而做以下处理：
  - 在runcmd的时候，我发现runcmd是在**主进程开了一个子进程**中运行的runcmd。我决定遇到相关指令则直接跳过，不在子进程中处理，这样就能**避免操作不会记录到主进程**，从而引发相关问题。如下：

    ```
    void runcmd(char *s)
    ```

```
{
    // 处理jobs
    oldchar = s[4];
    s[4] = 0;
    if (strcmp(s, "jobs") == 0)
    {
        s[4] = oldchar;
        return;
    }
    else
    {
        s[4] = oldchar;
    }

    // 处理kill
    oldchar = s[4];
    s[4] = 0;
    if (strcmp(s, "kill") == 0)
    {
        s[4] = oldchar;
        return;
    }
    else
    {
        s[4] = oldchar;
    }

    // 处理fg
    oldchar = s[2];
    s[2] = 0;
    if (strcmp(s, "fg") == 0)
    {
        s[2] = oldchar;
        return;
    }
    else
    {
        s[2] = oldchar;
    }
}

int main(int argc, char **argv) {
    for (;;) {
        if (interactive) {
            printf("\n$ ");
        }
        readline(buf, sizeof buf);

        if (buf[0] == '#') {
            continue;
        }
        if (echocmds) {
            printf("# %s\n", buf);
        }

        if ((r = pipe(pipe_fd)) != 0)
```

```
        {
            user_panic("pipe wrong: %d\n",r);
        }
        strcpy(globel_cmd, buf);
        if ((r = fork()) < 0) {
            user_panic("fork: %d", r);
        }
        if (r == 0) {
            //...
        }
        else    //在主进程中处理各种jobs指令
        {
            //...
            add_job(father_fktmp, globel_cmd);

            char *s = buf;
            // 处理jobs
            char oldchar = s[4];
            s[4] = 0;
            if (strcmp(s, "jobs") == 0)
            {
                s[4] = oldchar;
                list_jobs();
            }
            else
            {
                s[4] = oldchar;
            }

            // 处理kill
            oldchar = s[4];
            s[4] = 0;
            if (strcmp(s, "kill") == 0)
            {
                s[4] = oldchar;
                s = s + 4;
                char job_id_str[MAX_CMD_LENGTH] = {0};
                strcpy(job_id_str, s);

                kill_job(job_id_str);
            }
            else
            {
                s[4] = oldchar;
            }

            // 处理fg
            oldchar = s[2];
            s[2] = 0;
            if (strcmp(s, "fg") == 0)
            {
                s[2] = oldchar;
                s = s + 2;
                char job_id_str[MAX_CMD_LENGTH] = {0};
                strcpy(job_id_str, s);
```

```
                fg_job(job_id_str);
            }
            else
            {
                s[2] = oldchar;
            }
        }
    }
    return 0;
}
```

- 最后一切结束，**只剩下每次运行后台指令，需要做到记录**。这一点有些复杂，但是也不算复杂，只需要在最后的runcmd函数所在的子进程**开通一个管道**，将输出**运输到主进程**，然后主进程通过**add_list记录**job即可。

```
        strcpy(globel_cmd, buf);     //首先记录下输入的是什么指令。至于存不存进jobs，取决于他末尾是不是 &
        if ((r = fork()) < 0) {
            user_panic("fork: %d", r);
        }
        if (r == 0) {
            close(pipe_fd[0]); // 关闭读端
            runcmd(buf);

            if ((r=write(pipe_fd[1], &father_fktmp, sizeof(father_fktmp))) == -1)
            {
                user_panic("pipe write wrong: %d\n", r);
            }   //这一步是为了把后台指令的进程号运输到主进程，从而存下去。

            close(pipe_fd[1]); // 关闭写端
            exit();
        } else
        {
            close(pipe_fd[1]); // 关闭写端
            wait(r);

            if ((r = read(pipe_fd[0], &father_fktmp, sizeof(father_fktmp))) == -1)
            {
                user_panic("pipe read wrong: %d\n", r);
            }   //这一步是从管道中读取子进程的后台指令的进程号，从而记录到jobs表
            close(pipe_fd[0]); // 关闭读端
            add_job(father_fktmp, globel_cmd);

            //...
        }
```

- 除此之外，需要处理一下&判定为后台的一个词法解析。这里类同 `;` 的行为，只不过就是说，父进程**不需要等待子进程的结束**才能运行。比如 `sleep 10&`，此时父进程是 & **右边**，可是没有指令；子进程是 & **左边**，需要执行休眠10秒，于是巧妙实现了前后台的效果：父进程在**不执行任何指令的情况下也就可以接受新的指令的输入**了。

```
    case '&':
            fktmp = fork();
            if (fktmp)
            {
                father_fktmp = fktmp;    //传递进程号，同时不需要wait子进程，实现后台运行。
                return parsecmd(argv, rightpipe);
            }
            else
            {
                return argc;
            }
            break;
```

上述envid传递关系如下图：