# 挑战性任务实验报告

## 实现思路

### 必做部分

#### 实现一行多命令

由于在词法解析的过程中已经包含了“`;`”，所以不需要专门在`_gettoken`函数中为其做特殊的判断。一行多命令的整体运行逻辑是依次执行分号前后的两条命令，可以仿照管道"|"的实现过程，在遇到分号时利用`fork()`新建一个子进程，将已经解析过的命令行参数作为返回值，返回给子进程的`runcmd`进行执行，而父进程在`fork()`之后利用`fork`的返回值等待子进程完成第一条命令的处理，然后再次调用`parsecmd`解析分号之后的命令并返回`runcmd`完成即可。

代码表现形式具体如下：

```
case ';':
            *rightpipe = fork();                    //利用fork出的子进程完成第一条指令
            if(*rightpipe < 0){
                user_panic("fork error in sh.c");
            }
            if(*rightpipe == 0) {
                return argc;
            }else {
                wait(*rightpipe);                   //等待子进程完成其指令的执行
                return parsecmd(argv, rightpipe);   //继续解析并执行;之后的指令
            }
            break;
```

#### 实现后台任务

```
case '&':
            *rightpipe = fork();
            if(*rightpipe < 0){
                user_panic("fork error in sh.c");
            }
            if(*rightpipe == 0) {
                return argc;
            }else {
                return parsecmd(argv, rightpipe);
            }
            break;
```

在有了完成实现一行多命令的经验下，实现后台任务也是类似的操作。不同于实现一行多命令有着明确的前后关系，在实现后台任务时，最开始进行`runcmd`的进程不需要等待执行&左侧指令的子进程执行完毕，在创建完子进程并将命令行参数交给其之后就可以直接进行下一部分指令的处理。

### 实现引号支持

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

不同于以上两个符号，`SYMBOLS`宏定义中定义的特殊符号并没有包含"，故需要在`_gettoken`中为其增加特殊的判断。具体操作则表现为在识别到"之后，就不断寻找下一个"与之匹配。然后将中间部分整体作为一个命令行参数，记录其起始位置为第一个"的下一位，将第二个冒号置为"\0"用于表示参数的结尾，并返回给`parsecmd`即可。

### 实现键入命令时任意位置的修改

这是本次挑战性任务中遇到的第一个相当具有挑战的部分。在此部分，我重写了`readline`的实现逻辑，部分实现过程如下：

```
while (pos < n) {
    if ((r = read(0, &c, 1)) != 1) {
        if (r < 0) {
            debugf("read error: %d\n", r);
        }
        exit();
    }
    if(c == '\b' || c == 0x7f) {
        if(pos != 0) {
            if(pos == draftLen) {            //从末尾输出退格键的情况
                buf[--draftLen] = 0;
                pos--;
                printf("\b \b");            //先退格到目标光标处，然后输出一个空格再退格
覆盖掉被删除的位置
            } else {                         //在中间进行删除的情况
                strcpy(after, buf + pos);    //
                pos--;
                strcpy(buf + pos, after);
                buf[--draftLen] = 0;
                printf("\b");                //退一格代表光标完成删掉一个字符的动作
                printf("\x1b[K");            //删去原有后面的内容
                printf("%s", after);         //补充上被截断的后半部分
                for(int i = 0; i < strlen(after); i++) {
                        printf("\b");        //调整光标
                }
                memset(after,0,1024);
            }
    } else if(c == 27) {
        switch (checkDirKey()) {
            case 3:                          //输入左键进行缓冲区的指针左移
                if(pos > 0) {
                    pos--;
```

```
                } else {
                    printf("\x1b[1C");       //超出左移范围输出光标右移
                }
                break;
            case 4:
                if(pos < draftLen) {         //右移情况类似
                    pos++;
                } else {
                    printf("\x1b[1D");
                }
                break;
            default:
                break;
        }
    } else if(c == '\r' || c == '\n') {
        memset(after,0,1024);
        return ;
    } else {
        if(pos == draftLen) {                          //在末尾新加字符的只需要将字符放入
缓冲区即可
            buf[draftLen] = c;
            buf[++draftLen] = 0;
            pos++;
        } else {
            strcpy(after, draft + pos);              //保留原有光标后的内容到after中
            draft[pos] = c;
            printf("\x1b[K");                        //清除掉当前光标到末尾
            pos++;
            strcpy(draft + pos, after);
            printf("%s",after);                      //输出后半部分
            draft[++draftLen] = 0;
            for(int i = 0; i < strlen(after); i++) {
                printf("\b");                        //调整光标位置
            }
            memset(after,0,1024);
        }
    }
    if(draftLen > n) {
            break;
    }

}
```

整体上来说，需要同时完成两项任务：一是数组形式模拟的缓冲区的内容的更改与调整；二是屏幕上光标位置的模拟与调整。其中第二个任务需要明白光标与控制台输出运行的机理，灵活地调整对应的位置。

### 实现程序名称中 `.b` 的省略

```
//spawn.c
int fd;
if ((fd = open(prog, O_RDONLY)) < 0) {
    char temp[128];
    strcpy(temp,prog);
    int length = strlen(prog);
    temp[length] = '.';
    temp[length+1] = 'b';
    temp[length+2] = 0;
    if ((fd = open(temp, O_RDONLY)) < 0) {
        return fd;
    }
}
```

这部分修改相较于上一部分可以说是简单了很多，只需要在`spawn`打开文件失败的情况下，将原有的路径末尾补上`.b`再重新尝试打开即可。

### 实现更丰富的命令

不同于以上实现过程是对系统内部的命令做一定程度的修改与完善，如下三个命令是新的直接可以由用户发出的指令。因此，需要单独为其编写对应的`.c`文件，并在`user/include.mk`中加入，利用`makefile`将其制为二进制可执行文件供用户进行调用。

- `tree`

  tree的过程实际上是对给定路径下的文件夹(directory)及其内部文件的遍历过程，并在遍历的过程中按一定格式输出其文件名。而在已经实现的`ls`功能中就有对文件夹的内部的遍历过程，因此可以对此进行模仿。

  ```
  void treeDir(char *path, int depth) {                    //参数代表路径与遍历深度，用于空白字符的输出
      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) {          //对文件夹内部的文件进行遍历
          if (f.f_name[0]) {                                       //遍历到的文件有效，则对文件单位操作
              tree1(path, f.f_type == FTYPE_DIR, f.f_name, depth + 1);
          }
      }
  }

  void tree1(char *path, int isDir, char *name, int depth) {
      if(depth >= 2) {
          for(int i = 0; i < depth - 1; i++) {
              printf("  ");                                      //按照层深输出一定量的空格
          }
      }
      printf("|-- ");
  ```

```
    printf("%s\n", name);
    if(isDir) {
        dirCnt += 1;
        char newPath[MAXPATHLEN];
        int oldPathLen = strlen(path);
        strcpy(newPath, path);
        if(path[oldPathLen-1] != '/') {
            newPath[oldPathLen] = '/';
            strcpy(newPath + oldPathLen + 1, name);
        } else {
            strcpy(newPath + oldPathLen, name);
        }
        treeDir(newPath,depth);                                    //如果是目
录文件则更新path进行递归遍历
    } else {
        fileCnt += 1;                                              //如果是普
通文件则更新统计个数
    }
}
```

- `mkdir & touch`

  在已经定义的用户态文件操作中，并没有`create`类型的函数。但所幸，`fs.c`中已经完成了`file_create`函数。因此，要实现这两个命令只需要为`file_create`设置对应的接口，利用文件系统的模式化服务即可完成。

## 实现历史命令功能

整体上,历史命令功能的实现可以由两部分构成。第一部分是`history`命令，第二部分是根据`↑`与`↓`进行命令之间的跳转。

首先，两部分都需要做的，就是保存键入的命令，以保证能够实现列举所有的历史命令或者调出某一个历史命令。这个操作整体上比较简单，但需要实现一种新的文件打开方式，即追加写。实际上就是在每次打开时将偏移量(offset)调整到文件末尾即可。

```
void savecmd(char *cmd) {                                          //保存指令的函数
    int r;
    r = open(".history", O_WRONLY | O_APPEND);
    if(r < 0) {
        r = create(".history", FTYPE_REG);
        if(r < 0) {
            user_panic("create .history failed!");
        }
        r = open(".history", O_WRONLY | O_APPEND);                 //以追加写方式打
开.history文件
    }
    write(r, cmd, strlen(cmd));
    write(r, "\n", 1);                                            //写入对应的指令
    if(cmdCnt == 0) {
        cmdOffset[cmdCnt++] = strlen(cmd) + 1; //with \n
    } else {
        cmdOffset[cmdCnt] = cmdOffset[cmdCnt-1] + strlen(cmd) + 1;  //记录每条指令
相对于起始的偏移量，便于输出
        cmdCnt++;
```

```
    }
    close(r);
}
```

在完成了对指令内容的记录之后，最基本的`history`功能只需要将已经存入`.history`中的文件直接输出即可。

然后，再看到`↑`与`↓`。

上与下的调整，实际对应的是`.history`文件中对应第几行的调整。因此，在遇到上下键时，只需要调整目标行号，然后再根据行号调用函数获取`.history`中的历史命令。同时，为了保证能够回到当前已经输入部分的指令，需要将当前已经输入的内容保存下来。

```
void getHistoryCommand(int target, char *result) {                    //获取第i
行的历史指令
    int r, fd, len;
    char temp[4096];
    r = open("/.history", O_RDONLY);
    if(r < 0) {
        printf("open history failed or have no history command\n");
        exit();
    }
    fd = r;
    if(target == 0) {
        r = read(fd, result, cmdOffset[target]);
        if(r < 0) {
            printf("read history failed!\n");
            exit();
        }
        result[cmdOffset[target] - 1] = 0;
    } else {
        if((r = read(fd, temp, cmdOffset[target - 1])) != cmdOffset[target - 1])
{      //根据偏移量读取
            printf("read history failed!\n");
            exit();
        }
        len = cmdOffset[target] - cmdOffset[target - 1];
        if((r = read(fd, result, len)) != len) {
            printf("read history failed!\n");
            exit();
        }
        result[len - 1] = 0;
    }
    close(fd);
}
```

## 选做部分——实现相对路径

实际上并不存在什么相对路径，只不过是拼接成绝对路径罢了。因此，我们需要时刻维护`shell`运行的进程当前所在的绝对路径，在此基础上，采用拼接的方式，来达到相对路径的效果。而为了满足这样的效果，需要实现两个系统调用。一个用来获取当前的绝对路径，一个用来对`shell`进程的绝对路径进行修改。在初始化时，需要将进程的当前所在目录设置为根目录`/`。

```
// env.h
```

```
// Lab 6 challenge
    char env_cur_path[128];

// kern/env.c/env_create
    e->env_cur_path[0] = '/';
    e->env_cur_path[1] = '\0';

// syscall_all.c
void sys_get_cur_path(char *buf) {
    //printk("%s1\n",curenv->env_cur_path);
    strcpy(buf, curenv->env_cur_path);
}

int sys_set_cur_path(u_int envid, char *buf) {
    if(strlen(buf) >= 1024) {               //MAXPATHLEN
        return -E_BAD_PATH;
    }
    struct Env *e;
    envid2env(envid, &e, 0);
    strcpy(e->env_cur_path, buf);
    return 0;
}
```

在实现了以上两个系统调用之后，设计两个库函数`chdir`与`getcwd`来充当这两个系统调用的接口。同时，实现一个路径拼接函数，用于实现相对路径功能。在实现之后，务必在`USERLIB`中添加`path.o`用于编译。

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

int chdir(u_int envid, char *buf) {
    return syscall_set_cur_path(envid, buf);
}

void getcwd(char *buf) {
    return syscall_get_cur_path(buf);
}

void link_paths(char *prePath, char *posPath) {
    int preLen = strlen(prePath);
    if(preLen != 1) {
        prePath[preLen++] = '/';
    }
    if(posPath[0] == '.' && posPath[1] == '/') {
        posPath += 2;                //跳过./
    }
    int poslen = strlen(posPath);
    for(int i = 0; i < poslen; i++) {
        prePath[preLen + i] = posPath[i];
    }
    prePath[preLen + poslen] = 0;
}
```

在完成了以上函数之后，就可以真正实现相对路径，`pwd`，`cd`了。`pwd`直接在调用`getcwd`之后输出即可。相对路径的实现需要在之前所有可能用到路径的地方进行修改，如`open`函数，`create`函数（用于`mkdir`与`touch`命令）。在利用相对路径创建或打开文件时，这些函数都需要将当前路径与当前绝对路径拼接起来，完成和绝对路径相同的操作。而其他如`tree`,`ls`,`spawn`,`history`中，也需要做一些相关的调整。

```
if(strcmp(argv[0], "cd") == 0) {
        int r;
        if(argc != 2) {
            printf("wrong cd order\n");
            exit();
        }
        if(argv[1][0] == '/') {                                    //绝对路径直接切换cd
            if((r = chdir(shellEnvid, argv[1])) < 0) {
                printf("cd failed");
                exit();
            }
        } else {
            char temp[128];                                        //相对路径需要进行一些拼接
            getcwd(temp);
            link_paths(temp, argv[1]);
            if((r = chdir(shellEnvid, temp)) < 0) {
                printf("cd failed");
                exit();
            }
        }
        return ;
    }
```

# 测试程序

## 一行多命令

```
$ touch test1; ls
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b
testpipe.b motd init.b num.b lorem mkdir.b touch.b testfdsharing.b testshell.sh
script ls.b pwd.b echo.b sh.b tree.b halt.b testptelibrary.b .history test1
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...
```

可以看到，在ls的内容中出现了test1，证明一行多命令及其先后顺序合理。

## 后台任务

```
$ touch t1 & touch t2
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00004006] destroying 00004006
[00004006] free env 00004006
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

$ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b
testpipe.b motd init.b num.b lorem mkdir.b touch.b testfdsharing.b testshell.sh
script ls.b pwd.b echo.b sh.b tree.b halt.b testptelibrary.b .history t2 t1
[00005004] destroying 00005004
[00005004] free env 00005004
i am killed ...
[00004803] destroying 00004803
[00004803] free env 00004803
i am killed ...
```

符合要求。

## 引号支持

```
$ echo "ls"
ls
[00005004] destroying 00005004
[00005004] free env 00005004
i am killed ...
[00004803] destroying 00004803
[00004803] free env 00004803
i am killed ...
```

可以观察到，只输出了`ls`而没有按ls的真正功能完成，符合要求。

## 键入命令时的修改

这个在文档中比较难以展示，但能够正常运行（只要输入backspace的频率不要过高）。

## .b的省略

```
$ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b
testpipe.b motd init.b num.b lorem mkdir.b touch.b testfdsharing.b testshell.sh
script ls.b pwd.b echo.b sh.b tree.b halt.b testptelibrary.b .history test1
[00006004] destroying 00006004
[00006004] free env 00006004
i am killed ...
[00005803] destroying 00005803
[00005803] free env 00005803
i am killed ...
```

可以观察到输入ls后指令正常运行，符合要求。

## 更丰富的命令

```
$ mkdir testdir;touch testdir/test
[00007805] destroying 00007805
[00007805] free env 00007805
i am killed ...
[00007004] destroying 00007004
[00007004] free env 00007004
i am killed ...
[00008004] destroying 00008004
[00008004] free env 00008004
i am killed ...
[00006803] destroying 00006803
[00006803] free env 00006803
i am killed ...

$ tree
./
|-- aaa.txt
|-- testarg.b
|-- cat.b
|-- pingpong.b
|-- testbss.b
|-- newmotd
|-- history.b
|-- testpiperace.b
|-- testpipe.b
|-- motd
|-- init.b
|-- num.b
|-- lorem
|-- mkdir.b
|-- touch.b
|-- testfdsharing.b
|-- testshell.sh
|-- script
|-- ls.b
|-- pwd.b
|-- echo.b
```

```
|-- sh.b
|-- tree.b
|-- halt.b
|-- testptelibrary.b
|-- .history
|-- test1
|-- testdir
  |-- test

1 directories, 28 files
[00009004] destroying 00009004
[00009004] free env 00009004
i am killed ...
[00008803] destroying 00008803
[00008803] free env 00008803
i am killed ...
```

可以观察到mkdir,touch,tree都正常完成了其相应的功能。

## 历史功能命令

由于上下键难以在文档中展示，在此处只展示history的结果

```
$ history
1     touch test1; ls
2     echo "ls"
3     ls
4     mkdir testdir;touch testdir/test
5     tree
6     history
```

可以观察到，之前测试的全部内容都能够在通过history列出（此时没有测试后台任务），符合要求。

## 相对路径功能

```
$ mkdir dic1
[0000b004] destroying 0000b004
[0000b004] free env 0000b004
i am killed ...
[0000a803] destroying 0000a803
[0000a803] free env 0000a803
i am killed ...

$ cd dic1
[0000b803] destroying 0000b803
[0000b803] free env 0000b803
i am killed ...

$ touch ts1
[0000c804] destroying 0000c804
[0000c804] free env 0000c804
i am killed ...
[0000c003] destroying 0000c003
[0000c003] free env 0000c003
```

```
i am killed ...

$ cd /
[0000d003] destroying 0000d003
[0000d003] free env 0000d003
i am killed ...

$ tree
./
|-- aaa.txt
|-- testarg.b
|-- cat.b
|-- pingpong.b
|-- testbss.b
|-- newmotd
|-- history.b
|-- testpiperace.b
|-- testpipe.b
|-- motd
|-- init.b
|-- num.b
|-- lorem
|-- mkdir.b
|-- touch.b
|-- testfdsharing.b
|-- testshell.sh
|-- script
|-- ls.b
|-- pwd.b
|-- echo.b
|-- sh.b
|-- tree.b
|-- halt.b
|-- testptelibrary.b
|-- .history
|-- test1
|-- testdir
  |-- test
|-- dic1
  |-- ts1

2 directories, 29 files
```

可以观察到在cd到dic1之后仍然正确完成了touch命令，符合要求。

# 问题与解决方案

整体上来说，在完成的过程中阻力是相当大的。由于在完成Lab6时没有了课上测试的压力，因此并没有完全弄清shell的运行机制，同时文件系统部分的一知半解也为完成挑战性任务的过程中制造了很大的隐患。在整个过程中，不断通过自学体悟与请教他人，才最终完成了任务基本功能的实现。