CSAPP-SHELL-LAB

PB16030899-朱河勤

为了让用户可以控制系统，Linux 系统一般会运行一个 shell 程序。通常来说，shell 程序不会是系统启动后运行的第一个进程（也就是 init 进程), 下面通过c语言来实现一个简单的shell. 首先实现大致框架, 然后逐步增强,添加功能. 它支持一些内部命令, 如 pwd, ls, cd, cat, env, export, unset 以及外部命令 支持一些特色

- features:
  - \t support redundant blank(\t, spaces)
  - " ' support quote
  - \ multi-line input
  - | pipe
  - < > >> redirect
  - ; multi-cmd
  - & background
  - $ support varible: echo ".. $VAR"

1. 测试结果

先上结果 (。・∀・)ノ

```
2-mbinary ./init
# echo " $NAME  \"  ' \'  \$NAME"
 mbinary  \"  ' \'  \$NAME
# echo '$NAME " \' '
$NAME " \'
# echo "I am ${NAME}>_<" > tmp ; cat tmp
I am mbinary>_<
# echo 'b y e' >> tmp
# cat tmp
I am mbinary>_<
b y e
# wc <  tmp
 2  6 22
# pwd  ;  cd   ..     ;pwd
/mnt/c/Users/mbinary/gitrepo/2-mbinary
/mnt/c/Users/mbinary/gitrepo
# env  | \
    wc \
     | wc
      1       3      24
# export zhq=mbinary ; env | grep zhq
zhq=mbinary
# unset zhq ; env | grep zhq
# rm tmp ; exit
```

2. 大致框架

首先可以大致写出框架: 打印提示符, 解析命令, 执行内置命令, 执行外部命令. 循环

```c
//by osh助教
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>

int main() {
    /* 输入的命令行 */
    char cmd[256];
    /* 命令行拆解成的各部分，以空指针结尾 */
    char *args[128];
    while (1) {
        /* 提示符 */
        printf("# ");
        fflush(stdin);
        fgets(cmd, 256, stdin);
        /* 清理结尾的换行符 */
        int i;
        for (i = 0; cmd[i] != '\n'; i++)
            ;
        cmd[i] = '\0';
        /* 拆解命令行 */
        args[0] = cmd;
        for (i = 0; *args[i]; i++)
            for (args[i+1] = args[i] + 1; *args[i+1]; args[i+1]++)
```

```
                if (*args[i+1] == ' ') {
                    *args[i+1] = '\0';
                    args[i+1]++;
                    break;
                }
        args[i] = NULL;

        /* 没有输入命令 */
        if (!args[0])
            continue;

        /* 内建命令 */
        if (strcmp(args[0], "cd") == 0) {
            if (args[1])
                chdir(args[1]);
            continue;
        }
        if (strcmp(args[0], "pwd") == 0) {
            char wd[4096];
            puts(getcwd(wd, 4096));
            continue;
        }
        if (strcmp(args[0], "exit") == 0)
            return 0;

        /* 外部命令 */
        pid_t pid = fork();
        if (pid == 0) {
            /* 子进程 */
            execvp(args[0], args);
            /* execvp失败 */
            return 255;
        }
        /* 父进程 */
        wait(NULL);
    }
}
```

3. 全局变量说明

3.1. cmdStr

是用来接收输入的一个字符串数组

3.2. cmdNum, varNum

cmdNum记录 以 ; 分开的命令数目, varNum 记录 每条命令中的变量 $ 的个数

3.3. envVar

存储环境变量

3.4. cmd 结构

```
struct cmd{
    struct cmd * next;
    int begin,end;  // pos in cmdStr
    int argc;
    char lredir,rredir; //0:no redirect  1 <,>   ;  2  >>
    char toFile[MAX_PATH_LENGTH],fromFile[MAX_PATH_LENGTH];  // redirect file path
    char *args[MAX_ARG_NUM];
    char bgExec;   //failExec
};
```

next 是用来指向管道的下一次指令, 而全局变量 `cmdinfo` 数组定义如下

```
struct cmd cmdinfo[MAX_CMD_NUM];
```

是用来存放以 `;` 分开的多条指令.

4. 解析命令字符串

上面的大致框架简单实现中, 不够强壮, 比如命令字符串中不能连续多个空格等等. 所以在最后面的代码中, 重新实现解析命令字符串, 就是 `parseArgs`函数, 限于篇幅, 代码见文末.

这些函数解析命令字符串, 能支持多个空格, 支持多行输入, 支持了变量$, 支持引号',", 同时为重定向 <,>,<<,以及管道 |,做好准备

5. 多条命令的解析--;

`parseCmds` 函数解析多行输入,处理多个空格, \t 符号换为空格, 将多行命令通过命令结点形成链表. 在这个函数中, 也解析后台运行&符号, 如果有的话, 就设置命令头结点 的 head->bgEXec

6. 实现后台运行---&

这只需在创建子进程的实现, 是否让父进程 wait 这在 main 函数中可以看到

```
if(!pcmd->bgExec)wait(NULL);
```

7. 处理变量--$

在 `parseCmds` 函数 解析命令字符串时, 调用 `handleVar` 函数解析变量, 其工作是指示是否有变量, 如果有就解析记录下变量的名字

8. 内建命令

对于内建命令, 比如 ls, pwd, exit, env, unset 可以直接执行 在代码中, 内建命令的实现都在 `execInner` 函数中, 如果不是内建命令, 则返回1, 然后会调用执行外部命令的函数 `execOuter`

8.1. 实现 ls

```
int LS(char *path){
   DIR *dirp;
   struct dirent d,*dp = &d;
   dirp  = opendir(path);
   int ct=0;
   while((dp=readdir(dirp))!=NULL){
       printf("%s\n",dp->d_name);//,++ct%5==0?'\n':'');
   }
   closedir(dirp);
   return 0;
}
```

8.2. 实现 cd

pcmd->args[1] 是目的路径的指针

```
        struct stat st;
        if (pcmd->args[1]){
            stat(pcmd->args[1],&st);
            if (S_ISDIR(st.st_mode))
                chdir(pcmd->args[1]);
            else{
                printf("[Error]: cd '%s': No such directory\n",pcmd->args[1]);
                return -1;
            }
        }
```

8.3. 实现 pwd

```
printf("%s\n",getcwd(pcmd->args[1] , MAX_PATH_LENGTH));
```

8.4. 实现unset

unsetenv 调用, pcmd->args[i]是命令的各个参数的指针, 注意从1开始, 第0个参数是命令程序自己

```
for(int i=1;i<pcmd->argc;++i)unsetenv(pcmd->args[i]);
```

8.5. 实现 export

```
for(int i=1;i<pcmd->argc;++i){  //putenv( pcmd->args[i]);
            char *val,*p;
            for(p = pcmd->args[i];*p!='=';++p);
            *p='\0';
            val = p+1;
            setenv(pcmd->args[i],val,1);
        }
```

9. 实现重定向与管道-- <,>,>>,|

首先要知道一些关于linux文件I/O的知识, 可以看我这篇笔记

重定向的I/O 以及 管道的I/O, 我都放在 `setIO` 函数中处理,如下. 这个函数接受的参数包括一个命令指针 `pcmd` (以;分隔的, 包括管道中的命令), 以及 一个输入文件描述符`rfd`,一个输出文件描述符`wfd`.

9.1. 文件重定向

如果这条命令中(`pcmd->rredir`输出重定向) /(`pcmd->lredir` 输入重定向) 不为0, 就打开重定向的文件得到其文件描述符, 然后将标准 输出/输入文件描述符关闭, 再复制(用的dup2)到此文件描述符, 注意最后用完 此文件描述符 要用close关闭它.

9.2. 管道重定向

分别检查 文件描述符参数 是否 是标准输入,输出, 如果不是, 说明传递的是管道, 新的文件描述符, 就将相应的 标准输入/输出 关闭 ,再复制到 rfd/wfd, 最后close rfd/wfd

```
void setIO(struct cmd *pcmd,int rfd,int wfd){
        /* settle file and pipe redirect  */
    if(pcmd->rredir>0){  //  >,  >>
        int flag ;
        if(pcmd->rredir==1)flag=O_WRONLY|O_TRUNC|O_CREAT;  // >  note: trunc is
necessary!!!
        else flag=O_WRONLY|O_APPEND|O_CREAT; //    >>
        int wport = open(pcmd->toFile,flag);
        dup2(wport,STDOUT_FILENO);
        close(wport);
    }
    if(pcmd->lredir>0){  //<, <<
        int rport  = open(pcmd->fromFile,O_RDONLY);
        dup2(rport,STDIN_FILENO);
        close(rport);
    }

    /* pipe  */
    if(rfd!=STDIN_FILENO){
```

```
        dup2(rfd,STDIN_FILENO);
        close(rfd);
    }
    if(wfd!=STDOUT_FILENO){
        dup2(wfd,STDOUT_FILENO);
        close(wfd);
    }
}
```

10. 外部命令

实现的函数是 `execOuter`, 里面包括了重定向, 管道, 下面再介绍 对于外部命令, 应该 fork 一个 子进程, 让后让程序在子进程执行并返回, 可以使用 exec 家族的函数, 它会自动调用相应程序文, 件运行(忘了在哪个目录了😭), 我用的 `execvp` 函数

如果当前命令 的 `next` 为 `NULL`, 即没有下一条管道命令, 那么直接将标准文件描述符传给 `setIO` 处理好文件 IO, 然后调用execvp 执行外部命令即可

如果不为`NULL`, 说明有管道, 建立管道 , 用fork来新建子进程 执行管道命令, 这时传递到 `setIO` 函数的 对应 是管道文件描述符的 输入输出, 然后如果有多个管道, 可以递归地调用 `execOuter`函数, 如 cmd1 | cmd2 | cmd3... 我的实现是子进程执行 cmd1, 然后 将 cmd2 | cmd3 做为一个新命令传给 execOuter递归执行, 由于是用链表将各管道命令连起来的, 所以 直接传递 pmcd->next 即可, 非常方便

```
int execOuter(struct cmd * pcmd){
    if(!pcmd->next){
        setIO(pcmd,STDIN_FILENO,STDOUT_FILENO);
        execvp(pcmd->args[0],pcmd->args);
    }
    int fd[2];
    pipe(fd);
    pid_t pid = fork();
    if(pid<0){
        Error(FORK_ERROR);
    }else if (pid==0){
        close(fd[0]);
        setIO(pcmd,STDIN_FILENO,fd[1]);
        execvp(pcmd->args[0],pcmd->args);
        Error(EXEC_ERROR);
    }else{
        wait(NULL);
        pcmd = pcmd->next;  //notice
        close(fd[1]);
        setIO(pcmd,fd[0],STDOUT_FILENO);
        execOuter(pcmd);
    }
}
```

11. 其他

一些初始化, 错误处理等代码, 我就不再介绍, 可以直接看代码, 代码中有注释, 很容易看懂

12. 完整代码

访问 github

```
/*************************************************************************
        > File Name: init.c
    > Author: mbinary
    > Mail: zhuheqin1@gmail.com
    > Blog: https://mbinary.github.io
    > Created Time: 2018-04-15  11:18
    > Function:
        implemented some shell cmds and features;
        including:
            cmds: pwd,ls, cd ,cat, env, export , unset,
            features:$ \  |  <>>>   ;   & " ' quote handle \t redundent blank
 ************************************************************************/

#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <malloc.h>

#define MAX_CMD_LENGTH 255
#define MAX_PATH_LENGTH 255
#define MAX_BUF_SIZE  4096
#define MAX_ARG_NUM 50
#define MAX_VAR_NUM 50
#define MAX_CMD_NUM 10
#define MAX_VAR_LENGTH 500

#define FORK_ERROR 2
#define EXEC_ERROR 3

struct cmd{
    struct cmd * next;
    int begin,end;  // pos in cmdStr
    int argc;
    char lredir,rredir; ////0:no redirect  1 <,>   ;  2  >>
    char toFile[MAX_PATH_LENGTH],fromFile[MAX_PATH_LENGTH];  // redirect file path
    char *args[MAX_ARG_NUM];
```

```
    char bgExec;   //failExec
};

struct cmd cmdinfo[MAX_CMD_NUM];
char cmdStr[MAX_CMD_LENGTH];
int cmdNum,varNum;
char envVar[MAX_VAR_NUM][MAX_PATH_LENGTH];


void Error(int );
void debug(struct cmd*);
void init(struct cmd*);
void setIO(struct cmd*,int ,int );
int  getInput();
int  parseCmds(int);
int  handleVar(struct cmd *,int);
int  getItem(char *,char *,int);
int  parseArgs();
int  execInner(struct cmd*);
int  execOuter(struct cmd*);


int main(){
    while (1){
        cmdNum = varNum = 0;
        printf("# ");
        fflush(stdin);
        int n = getInput();
        if(n<=0)continue;
        parseCmds(n);
        if(parseArgs()<0)continue;
        for(int i=0;i<cmdNum;++i){
            struct cmd *pcmd=cmdinfo+i, * tmp;
            //debug(pcmd);
            //pcmd = reverse(pcmd);
            int status = execInner(pcmd);
            if(status==1){
                /*notice!!!  Use child proc to  execute outer cmd,
                bacause exec funcs won't return when successfully execed.  */
                pid_t pid = fork();
                if(pid==0)execOuter(pcmd);
                else if(pid<0)Error(FORK_ERROR);
                if(!pcmd->bgExec)wait(NULL);  //background exec
                /*  free malloced piep-cmd-node,
                    and the first one is static , no need to free;   */
                pcmd=pcmd->next;
                while(pcmd){
                    tmp = pcmd->next;
                    free(pcmd);
                    pcmd=tmp;
                }
            }
        }
```

```
    }
    return 0;
}


/* funcs implementation */
void init(struct cmd *pcmd){
    pcmd->bgExec=0;
    pcmd->argc=0;
    pcmd->lredir=pcmd->rredir=0;
    pcmd->next = NULL;
    pcmd->begin=pcmd->end=-1;
    /* // notice!!! Avoid using resudent args  */
    for(int i=0;i<MAX_ARG_NUM;++i)pcmd->args[i]=NULL;
}

void Error(int n){
    switch(n){
        case FORK_ERROR:printf("fork error\n");break;
        case EXEC_ERROR:printf("exec error\n");break;
                default:printf("Error, exit ...\n");
    }
    exit(1);
}



int getInput(){
        /* multi line input */
    int pCmdStr=0,cur;
    char newline = 1;
    while(newline){
        cur = MAX_CMD_LENGTH-pCmdStr;
        if(cur<=0){
            printf("[Error]: You cmdStr is too long to exec.\n");
            return -1;// return -1 if cmdStr size is bigger than LENGTH
        }
        fgets(cmdStr+pCmdStr,cur,stdin);
        newline = 0;
        while(1){
            if(cmdStr[pCmdStr]=='\\'&&cmdStr[pCmdStr+1]=='\n'){
                newline=1;
                cmdStr[pCmdStr++]='\0';
                break;
            }
            else if(cmdStr[pCmdStr]=='\n'){
                break;
            }
            ++pCmdStr;
        }
    }
    return pCmdStr;
}
```

```
int  parseCmds(int n){
    /* clean the cmdStr and get pos of each cmd in the cmdStr (OoO) */
    char beginCmd=0;
    struct cmd * head; // use head cmd to mark background.
    for( int i=0;i<=n;++i){
        switch(cmdStr[i]){
            case '&':{
                if(cmdStr[i+1]=='\n'||cmdStr[i+1]==';'){
                    cmdStr[i]=' ';
                    head->bgExec=1;
                }
            }
                        case '\t':cmdStr[i]=' ';break;
            case ';':{//including  ';'  a new cmdStr
                beginCmd = 0;
                cmdStr[i]='\0';
                cmdinfo[cmdNum++].end=i;
                break;
            }
            case '\n':{
                cmdStr[i]='\0';
                cmdinfo[cmdNum++].end =i;
                return 0;
            }
            case ' ':break;
            default:if(!beginCmd){
                beginCmd=1;
                head = cmdinfo+cmdNum;
                cmdinfo[cmdNum].begin =  i;
            }
        }
    }
}

int getItem(char *dst,char*src, int p){
    /* get redirect file path from the cmdStr */
    int ct=0;
    while(src[++p]==' ');
    if(src[p]=='\n')return -1; //no file
    char c;
    while(c=dst[ct]=src[p]){
        if(c==' '||c=='|'||c=='<'||c=='>'||c=='\n')break;
        ++ct,++p;
    }
    dst[ct]='\0';
    return p-1;
}

int handleVar(struct cmd *pcmd,int n){
    char * arg = pcmd->args[n];
    int p_arg=0,p_var=0;
    while(arg[p_arg]){
        if((arg[p_arg]=='$')&&(arg[p_arg-1]!='\\')){
            if(arg[p_arg+1]=='{')p_arg+=2;
```

```
            else p_arg+=1;
            char *tmp=&envVar[varNum][p_var];
            int ct=0;
            while(tmp[ct]=arg[p_arg]){
                if(tmp[ct]=='}'){
                    ++p_arg;
                    break;
                }
                if(tmp[ct]==' '||tmp[ct]=='\n'||tmp[ct]=='\0')break;
                ++ct,++p_arg;
            }
            tmp[ct]='\0';
            tmp = getenv(tmp);
            for(int i=0;envVar[varNum][p_var++]=tmp[i++];);
            p_var-=1; //necessary
        }
        else envVar[varNum][p_var++]=arg[p_arg++];
    }
    envVar[varNum][p_var]='\0';
    pcmd->args[n] = envVar[varNum++];
    return 0;
}

int parseArgs(){
    /* get args of each cmd and  create cmd-node seperated by pipe */
    char beginItem=0,beginQuote=0,beginDoubleQuote=0,hasVar=0,c;
        int begin,end;
        struct cmd* pcmd;
    for(int p=0;p<cmdNum;++p){
                if(beginQuote||beginItem||beginDoubleQuote){
                        return -1;  // wrong cmdStr
                }
        pcmd=&cmdinfo[p];
        begin = pcmd->begin,end = pcmd->end;
        init(pcmd);// initalize
        for(int i=begin;i<end;++i){
            c = cmdStr[i];
                        if((c=='\"')&&(cmdStr[i-1]!='\\'&&(!beginQuote))){
                                if(beginDoubleQuote){
                                        cmdStr[i]=beginDoubleQuote=beginItem=0;
                    if(hasVar){
                        hasVar=0;
                        handleVar(pcmd,pcmd->argc-1);  //note that is argc-1, not
argc
                    }
                }else{
                                        beginDoubleQuote=1;
                                        pcmd->args[pcmd->argc++]=cmdStr+i+1;
                                }
                continue;
                        }else if(beginDoubleQuote){
                if((c=='$') &&(cmdStr[i-1]!='\\')&&(!hasVar))hasVar=1;
                continue;
            }
```

```
            if((c=='\'')&&(cmdStr[i-1]!='\\')){
                if(beginQuote){
                                        cmdStr[i]=beginQuote=beginItem=0;
                }else{
                    beginQuote=1;
                    pcmd->args[pcmd->argc++]=cmdStr+i+1;
                }
                continue;
            }else if(beginQuote) continue;


            if(c=='<'||c=='>'||c=='|'){
                if(beginItem)beginItem=0;
                cmdStr[i]='\0';
            }
            if(c=='<'){
                if(cmdStr[i+1]=='<'){
                    pcmd->lredir+=2;  //<<
                    cmdStr[i+1]=' ';
                }else{
                    pcmd->lredir+=1;  //<
                }
                int tmp = getItem(pcmd->fromFile,cmdStr,i);
                if(tmp>0)i = tmp;
            }else if(c=='>'){
                if(cmdStr[i+1]=='>'){
                    pcmd->rredir+=2;  //>>
                    cmdStr[i+1]=' ';
                }else{
                    pcmd->rredir+=1;  //>
                }
                int tmp = getItem(pcmd->toFile,cmdStr,i);
                if(tmp>0)i = tmp;
            }else if (c=='|'){
                /*when encountering pipe | , create new cmd node chained after the
fommer one   */
                pcmd->end = i;
                pcmd->next = (struct cmd*)malloc(sizeof(struct cmd));
                pcmd = pcmd->next;
                init(pcmd);
            }else if(c==' '||c=='\0'){
                if(beginItem){
                    beginItem=0;
                    cmdStr[i]='\0';
                }
            }else{
                if(pcmd->begin==-1)pcmd->begin=i;
                if(!beginItem){
                    beginItem=1;
                    if((c=='$') &&(cmdStr[i-1]!='\\')&&(!hasVar))hasVar=1;
                    pcmd->args[pcmd->argc++]=cmdStr+i;
                }
            }
```

```
            if(hasVar){
                hasVar=0;
                handleVar(pcmd,pcmd->argc-1);  //note that is argc-1, not argc
            }
        }
        pcmd->end=end;
        //printf("%dfrom:%s   %dto:%s\n",pcmd->lredir,pcmd->fromFile,pcmd-
>rredir,pcmd->toFile);
    }
}

int execInner(struct cmd* pcmd){
    /*if inner cmd, {exec, return 0} else return 1  */
    if (!pcmd->args[0])
        return 0;
    if (strcmp(pcmd->args[0], "cd") == 0) {
        struct stat st;
        if (pcmd->args[1]){
            stat(pcmd->args[1],&st);
            if (S_ISDIR(st.st_mode))
                chdir(pcmd->args[1]);
            else{
                printf("[Error]: cd '%s': No such directory\n",pcmd->args[1]);
                return -1;
            }
        }
        return 0;
    }
    if (strcmp(pcmd->args[0], "pwd") == 0) {
        printf("%s\n",getcwd(pcmd->args[1] , MAX_PATH_LENGTH));
        return 0;
    }
    if (strcmp(pcmd->args[0], "unset") == 0) {
        for(int i=1;i<pcmd->argc;++i)unsetenv(pcmd->args[i]);
        return 0;
    }
    if (strcmp(pcmd->args[0], "export") == 0) {
        for(int i=1;i<pcmd->argc;++i){  //putenv(pcmd->args[i]);
            char *val,*p;
            for(p = pcmd->args[i];*p!='=';++p);
            *p='\0';
            val = p+1;
            setenv(pcmd->args[i],val,1);
        }
        return 0;
    }
    if (strcmp(pcmd->args[0], "exit") == 0)
        exit(0);
    return 1;
}

void setIO(struct cmd *pcmd,int rfd,int wfd){
        /* settle file redirect  */
```

```
    if(pcmd->rredir>0){  //  >,  >>
        int flag ;
        if(pcmd->rredir==1)flag=O_WRONLY|O_TRUNC|O_CREAT;  // >  note: trunc is
necessary!!!
        else flag=O_WRONLY|O_APPEND|O_CREAT; //>>
        int wport = open(pcmd->toFile,flag);
        dup2(wport,STDOUT_FILENO);
        close(wport);
    }
    if(pcmd->lredir>0){  //<, <<
        int rport  = open(pcmd->fromFile,O_RDONLY);
        dup2(rport,STDIN_FILENO);
        close(rport);
    }

    /* pipe  */
    if(rfd!=STDIN_FILENO){
        dup2(rfd,STDIN_FILENO);
        close(rfd);
    }
    if(wfd!=STDOUT_FILENO){
        dup2(wfd,STDOUT_FILENO);
        close(wfd);
    }
}

int execOuter(struct cmd * pcmd){
    if(!pcmd->next){
        setIO(pcmd,STDIN_FILENO,STDOUT_FILENO);
        execvp(pcmd->args[0],pcmd->args);
    }
    int fd[2];
    pipe(fd);
    pid_t pid = fork();
    if(pid<0){
        Error(FORK_ERROR);
    }else if (pid==0){
        close(fd[0]);
        setIO(pcmd,STDIN_FILENO,fd[1]);
        execvp(pcmd->args[0],pcmd->args);
        Error(EXEC_ERROR);
    }else{
        wait(NULL);
        pcmd = pcmd->next;  //notice
        close(fd[1]);
        setIO(pcmd,fd[0],STDOUT_FILENO);
        execOuter(pcmd);
    }
}
```