Myshell实验报告

一、实验目的

本实验旨在通过使用C语言编写一个类似于bash的命令行程序，实现基本的命令行功能，包括文件操作、进程管理等。

二、实验环境

操作系统：wsl

开发工具：Visual Studio Code

1. 实验内容

1. 实现基本的命令行界面

在程序启动时，显示命令行提示符，等待用户输入命令。用户输入命令后，程序解析命令，并根据命令执行相应的操作。

1. 实现一些内建函数

bg、cd 、clr、dir、echo 、exec 、exit 、fg 、help、jobs 、pwd 、set 、test 、time 、umask等基本功能均已实现。

1. 实现进程管理功能

命令行程序支持进程管理功能，包括进程的启动、终止、暂停。

1. 实现管道和重定向
2. 实现思路
3. 命令的读取

Main函数为程序的主题，进入程序后先对进程和信号做初始化。

调用JobInit()函数创建共享内存用来给JobList（其实这一段还不是很懂为什么不能用普通的申请的内存，必须要共享内存的原因是什么？），并将joblist中的第一个job设置为myshell。

调用signalInit函数来设置两种信号：TSTP即中断信号，CHLD即停止信号，并分别设置两个handler函数来处理这两种信号

然后设置一些环境变量。用getcwd得到当前工作路径并将其设置为环境变量。并将pwd变量设置为当前路径。

接下来先判断有没有脚本并读取脚本内容并用paser解析，用execute执行，而后进入用户交互界面。

1. 命令的解析

主要在paser函数中解析。用strtok函数将命令用空格分割成小的部分，然后利用struct command结构，将一行中的各个小部分分到不同的命令中（只有在有管道的情况下一行才会有多条命令），并顺便判断每条命令有无输入输出重定向。最后返回一个命令列表。

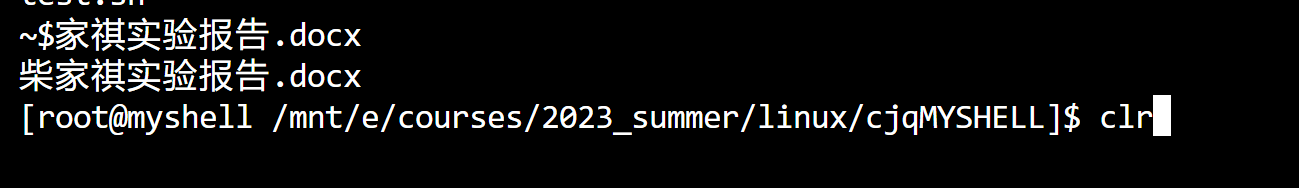
1. 命令的执行

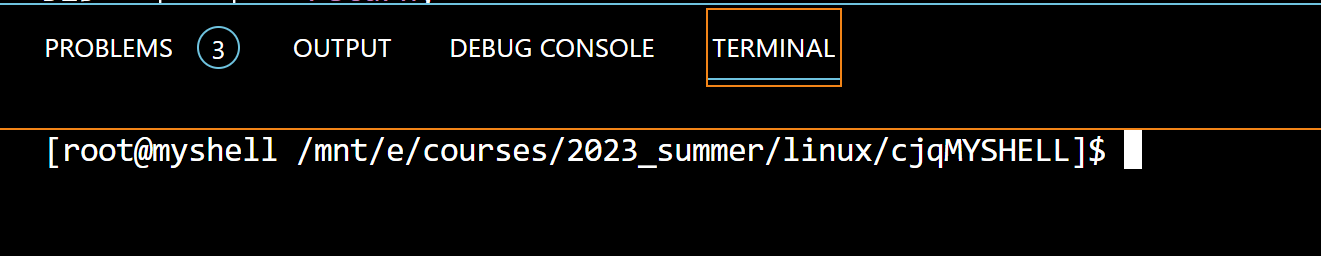
命令的执行主要用到execute函数，它分为三个部分：有管道的情况，无管道外部命令，无管道内部命令。首先有管道内部命令直接先判断重定向并执行即可。有管道的情况和无管道外部命令均要建立子进程。创建后父进程根据子进程的状态决定是继续进行还是等待，子进程做命令的执行。外部命令调用exec函数。管道的话要循环新建子进程做一个命令，并用到一个管道的共享文件做输入输出。

内建命令的执行用到BuildIn函数，里面分类调用了多种实现的小函数来执行某一项内部命令。

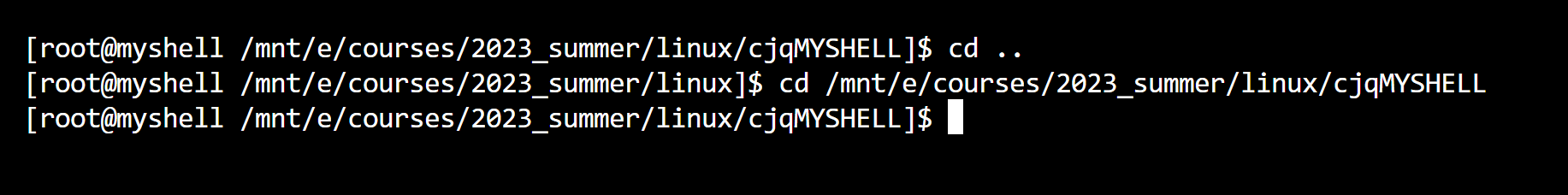
1. 实验结果

clr : 清屏

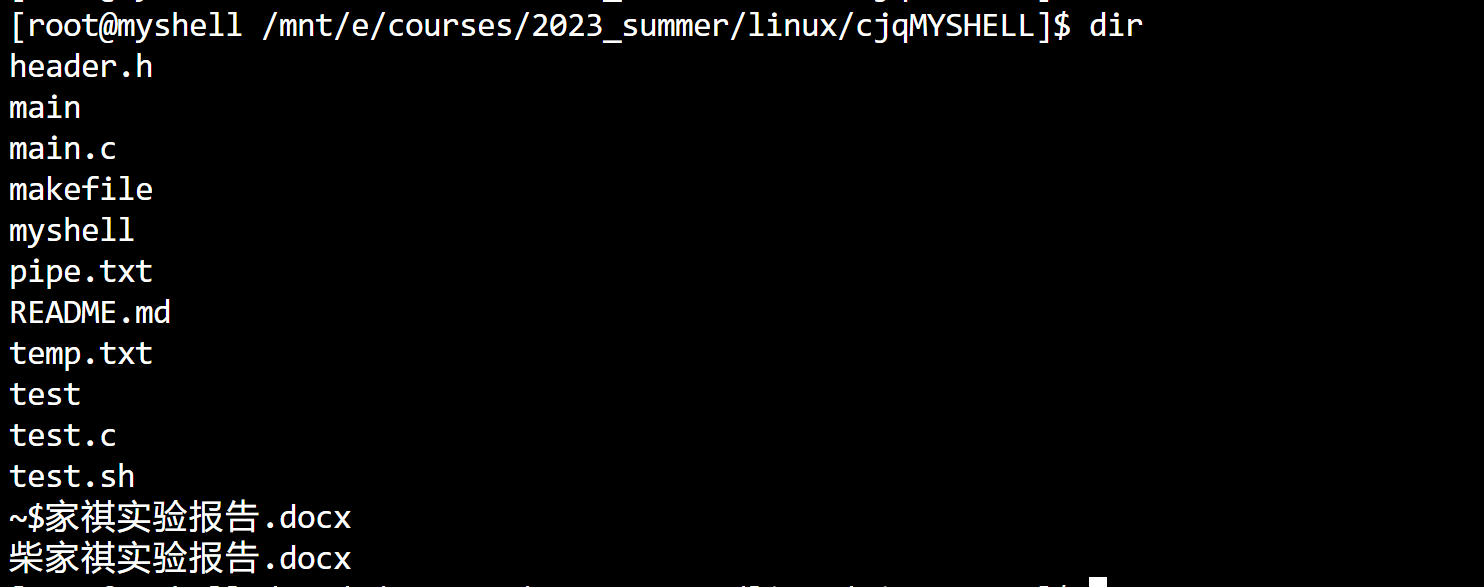




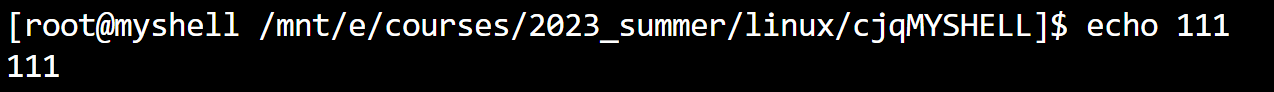
cd : 改变工作目录

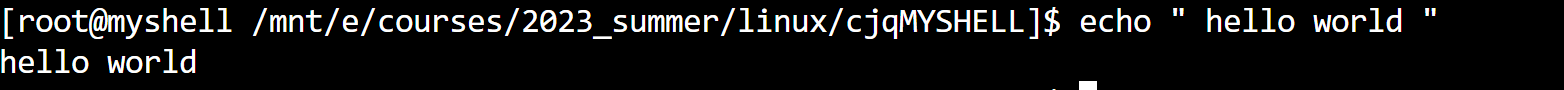


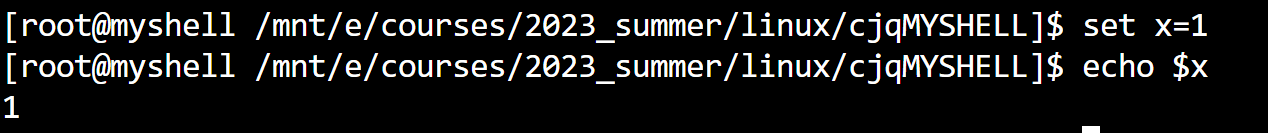
dir : 列出目录内容 e.g. dir



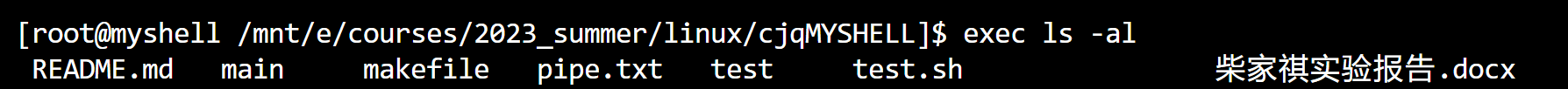
echo : 在屏幕上显示文字,若要显示的字符串有空格，要用双引号括出，且双引号和前后要有空格 e.g. echo 111 echo " hello world " echo $x



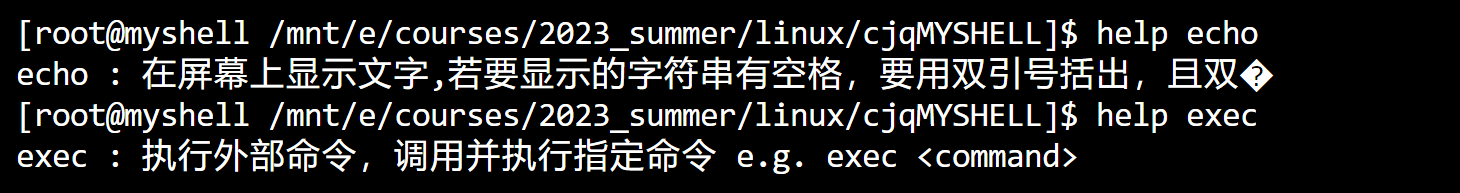




exec : 执行外部命令，调用并执行指定命令 e.g. exec <command>

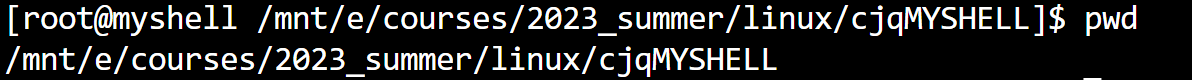


help : 显示用户手册中对于某个指令的帮助 e.g. help <command>

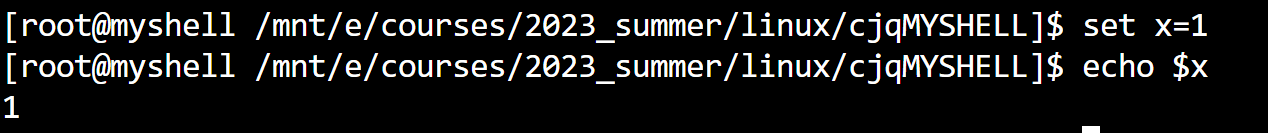


jobs : 显示后台进程

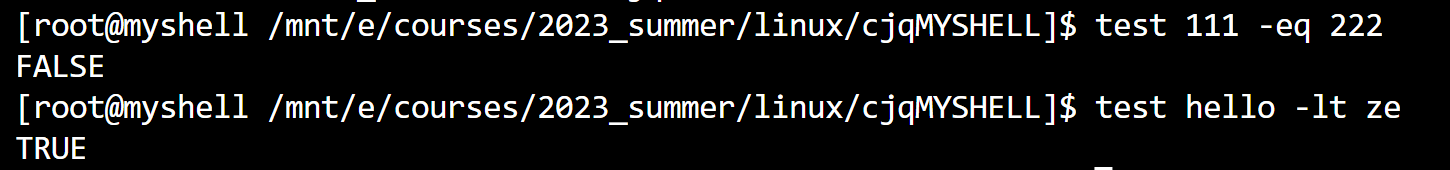
pwd : 输出当前工作目录



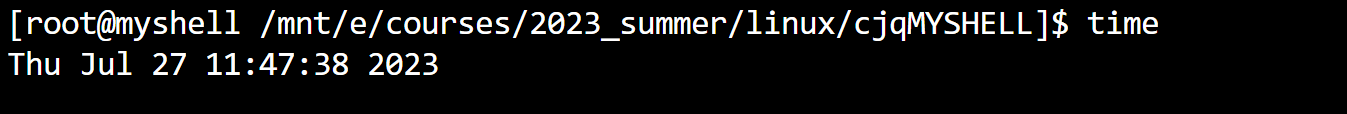
set : 设置环境变量 e.g. set x=1



test : test <string1> <parameter> <string2> 比较两个字符串并返回表达式bool值。参数：-eq -ne -gt -lt -ge -le



Time : 显示时间



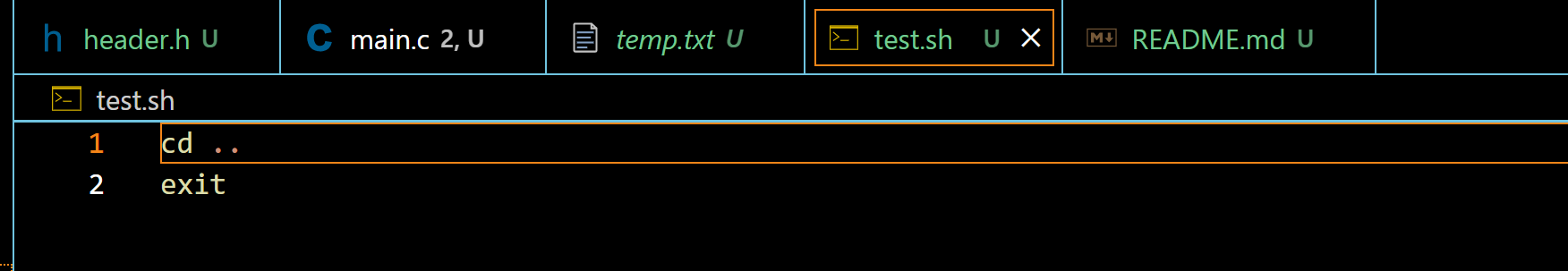
Umask : 无参数时显示当前mask，有参数时将mask设置为参数

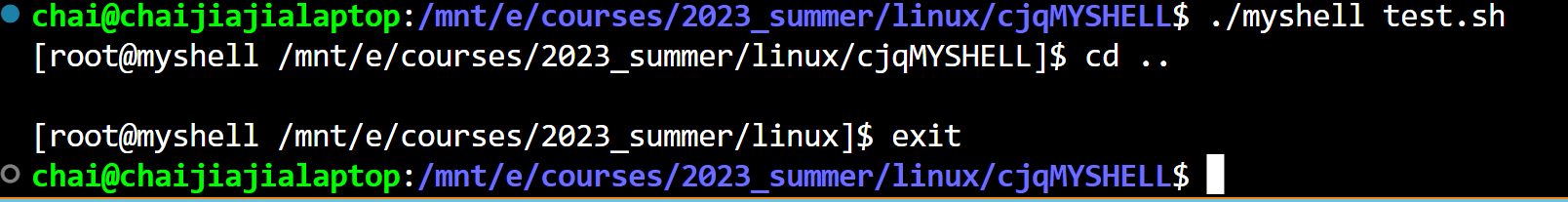


## 脚本与交互

运行时在终端输入./myshell 进入用户交互界面

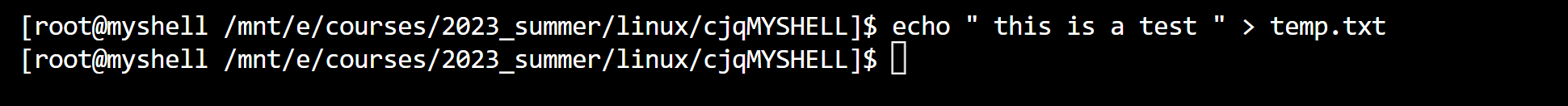
输入 ./myshell <document>自动读入脚本，脚本执行完毕后进入用户交互界面 e.g. ./myshell test.sh

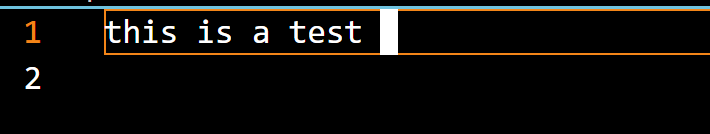




## 重定向

使用输出重定向时，如果重定向字符是>，则创建输出文件，如果存在则覆盖之；如果重定向字符为>>，也会创建输出文件，如果存在则添加到文件尾。e.g. echo <string> > <file>

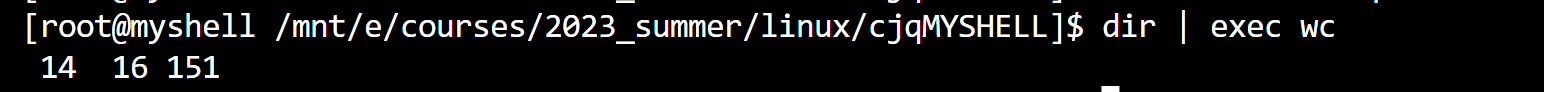






## 管道

在“|”的两边分别有两个指令，将前一段指令的输出作为下一段指令的输入，管道两边要加空格。e.g. dir | exec wc



1. 实验源代码

Head.h:

#ifndef HEADER\_H

# define HEADER\_H

#define \_XOPEN\_SOURCE//不加这行信号类会报错

//头文件

# include <stdio.h>

# include <stdlib.h>

# include <signal.h>

# include <string.h>

# include <unistd.h>

# include <dirent.h>

# include <stddef.h>

#include <fcntl.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <wait.h>

#include <time.h>

//宏定义

# define MAXLENGTH 100 //命令最大长度

# define MAXNAME 100 //进程名最大长度

# define MAXJOB 1024 //最大进程数

# define MAXPATH 100 //路径最长长度

# define MAXSECTION 30 //一行最多的部分数量

# define SHMSIZE (sizeof(JobList) \* MAXJOB) //共享内存大小

//结构体

enum Type{FG=0/\*前台\*/, BG=1/\*后台\*/};

enum JobStatus{RUN=0/\*运行\*/, SUSPEND=1/\*中止\*/, DONE=2/\*完成\*/};

typedef struct job{ //进程结构体

int pid; //进程号

char name[MAXLENGTH];//进程名

enum Type type;

enum JobStatus status;

}Job;

struct joblist{ //进程列表结构体

int num; //进程数

struct job List[MAXJOB]; //进程数组

};

struct joblist \*JobList; //进程列表指针

//命令

struct command{

char SplitCMD[MAXSECTION][MAXLENGTH];

int PipIn;//管道输入

int PipOut;//管道输出

int RedirectionIn;//重定向输入

int RedirectionOut;//重定向覆盖输出

int RedirectionAppend;//重定向追加输出

char RedirectionInFile[MAXPATH];//重定向输入文件

char RedirectionOutFile[MAXPATH];//重定向输出文件

char num;//命令数量

};

typedef struct commandlist{//命令列表

struct command cmd[MAXSECTION];

int num;

enum Type type;

}CommandList;

//信号

struct sigaction TSTP; //SIGTSTP信号结构，中断

struct sigaction TSTP\_old; //SIGTSTP信号原值

struct sigaction CHLD; //SIGCHLD信号结构，停止

struct sigaction CHLD\_old; //SIGCHLD信号原值

//函数声明

void JobInit(); //初始化进程列表

void SignalInit(); //初始化信号

void CHLD\_handler(int sign); //SIGCHLD信号处理函数

void TSTP\_handler(int sign); //SIGTSTP信号处理函数

CommandList\* paser(char\* cmd);

void execute(CommandList\* CMDlist);

int FindJob(int pid);

void DeleteJob(pid\_t pid);

int BuildIn(struct command cmd);

//内建函数

void bg();//找到最近暂停的后台进程并将其运行

void clr();//清屏

void cd(char\* path);//改变工作目录

void dir(char\* path);//列出目录

void echo(struct command cmd);//输出字符串

void exec(struct command cmd);//执行程序

void fg();//找到最近后台并将其前台运行

void help(char\* str);//输出帮助信息

void jobs();//列出所有后台进程

void pwd();//输出当前工作目录

void set(char\* str);//设置环境变量

int test(struct command cmd);//测试函数

void Time();//输出当前时间

void Umask(char\* str);//设置文件权限掩码

//全局变量

char path[MAXPATH]; //工作目录

char CurrentPath[MAXLENGTH];//当前目录

char PipePath[MAXLENGTH]="pipe.txt";//管道文件路径

# endif

Main.c:

# include "header.h"

int main(int argc,char \*\*argv)

{

JobInit();//初始化进程列表

SignalInit();//初始化信号处理

getcwd(path, sizeof(path));

strcpy(CurrentPath,path);//得到当前目录

strcat(path,"/myshell");//得到工作路径

// 调用setenv设置myshell环境变量

setenv( "myshell"/\*变量名\*/, path/\*变量值\*/, 1/\*覆写\*/);

setenv("pwd", path, 1);//设置工作目录环境变量(后面能被getcwd读取

FILE \*fp;

for(int i=1;i<argc;i++){//处理脚本

if((fp=fopen(argv[i],"r"))==NULL){//用只读模式打开脚本

printf("error: failed to open file\n");

break;

};

char ScripCMD[MAXLENGTH];//存放脚本中的命令

while(!feof(fp))//处理脚本每一行命令

{

fgets(ScripCMD,MAXLENGTH,fp);//得到每一行

printf("[root@myshell %s]$ %s\n",getcwd(NULL, 0),ScripCMD);

if(strcmp(ScripCMD,"exit")==0) exit(0);//退出

execute(paser(ScripCMD));

}

}

char CMD[MAXLENGTH];

while(1){//处理用户指令

printf("[root@myshell %s]$ ",getcwd(NULL, 0));

gets(CMD);//得到用户输入

if(strcmp(CMD,"exit")==0) exit(0);

execute(paser(CMD));

}

return 0;

}

void JobInit(){

int Ipc\_ID;

if((Ipc\_ID=shmget( IPC\_PRIVATE/\*忽略键，建立一个新的共享内存，指定一个键值 ，然后返回这块共享内存IPC标识符ID。\*/ , SHMSIZE, 0666/\*全局读取和写入许可\*/ | IPC\_CREAT/\*如果不存在就创建\*/ ))==-1) {printf("failed to create shared memeory");exit(1);}

// 挂接共享内存

void \*shm\_addr=NULL;

shm\_addr = shmat( Ipc\_ID, NULL/\*共享内存会被attach到一个合适的虚拟地址空间\*/, 0/\*可读可写\*/);

//初始化共享内存

memset(shm\_addr, 0, SHMSIZE) ;

// 将共享内存地址赋给进程表

JobList = (Job\*)shm\_addr;

//初始化进程表

JobList->List[0].pid = getpid();//getpid()返回当前进程的进程号，即myshell的进程号

strcpy( JobList->List[0].name, "myshell");

JobList->List[0].type = BG;

JobList->List[0].status = RUN;

JobList->num = 1;

}

void SignalInit(){

//中断信号

memset(&TSTP,0,sizeof(TSTP));//清空TSTP

TSTP.sa\_handler = TSTP\_handler;//设置TSTP的信号处理函数

sigaction(SIGTSTP, &TSTP, &TSTP\_old);//注册TSTP信号处理函数

//退出信号

memset(&CHLD, 0, sizeof(CHLD));//清空CHLD

CHLD.sa\_handler = CHLD\_handler;//设置CHLD的信号处理函数

sigaction(SIGCHLD, &CHLD, &CHLD\_old);//注册CHLD信号处理函数

}

void TSTP\_handler(int sign){

for(int i=JobList->num-1;i>0;i--){//从后往前找，找到第一个RUN的前台进程，将其状态改为后台SUSPEND

if(JobList->List[i].status==RUN&&JobList->List[i].type==FG){

JobList->List[i].status = SUSPEND;

JobList->List[i].type = BG;

kill(JobList->List[i].pid,SIGSTOP);//用kill函数发送信号

break;

}

}

}

void CHLD\_handler(int sign){

if(JobList->num==1) return;//如果进程表中只有myshell

if(JobList->List[JobList->num-1].type=FG){//若是前台进程，直接删除

JobList->num-=1;

}

else if(JobList->List[JobList->num-1].status==SUSPEND||JobList->List[JobList->num-1].status==RUN){//如果最后一个进程是后台挂起或正在执行的进程

JobList->List[JobList->num-1].status = DONE;//将其状态改为DONE

}

}

CommandList\* paser(char\* cmd){

if(strcmp(cmd,"\0")==0) return;//为空

char SplitCMD[MAXSECTION][MAXLENGTH];

memset(SplitCMD,0,sizeof(SplitCMD));

char\* separator=" \n\t";//用空格换行和制表符分隔

strcpy(SplitCMD[0],strtok(cmd,separator));

int size;

for(size=1;;size++){//循环分解

char \*temp=strtok(NULL/\*strtok用this指针指向了分解符的下一位\*/,separator);

if(temp==NULL) break;//分解完毕

strcpy(SplitCMD[size],temp);

}

CommandList\* CMD =(CommandList\*)malloc(sizeof(CommandList));//创建命令结构体

memset(CMD,0,sizeof(CommandList));//初始化为0

int cnt=0;

for(int i=0;i<size;i++){//检查每个部分,命令预处理

if(strcmp(SplitCMD[i],"<")==0){//输入重定向

CMD->cmd[CMD->num].RedirectionIn=1;

strcpy(CMD->cmd[CMD->num].RedirectionInFile,SplitCMD[++i]);//下一部分一定是重定向的文件路径

break;

}

else if(strcmp(SplitCMD[i],">")==0){//输出重定向

CMD->cmd[CMD->num].RedirectionOut=1;

strcpy(CMD->cmd[CMD->num].RedirectionOutFile,SplitCMD[++i]);

break;

}

else if(strcmp(SplitCMD[i],">>")==0){//输出重定向,追加

CMD->cmd[CMD->num].RedirectionAppend=1;

strcpy(CMD->cmd[CMD->num].RedirectionOutFile,SplitCMD[++i]);

break;

}

else if(strcmp(SplitCMD[i],"&")==0){//后台运行

CMD->type=BG;

break;

}

else if(strcmp(SplitCMD[i],"|")==0){//管道

CMD->cmd[CMD->num].PipOut=1;//这条命令为管道输出

CMD->cmd[++CMD->num].PipIn=1;//下一条命令为管道输入

}

else{//非关键词

strcpy(CMD->cmd[CMD->num].SplitCMD[CMD->cmd[CMD->num].num++],SplitCMD[i]);//将命令存入命令结构体

}

}

return CMD;

}

void execute(CommandList\* CMDlist){

int stdin\_copy = dup(0);//备份标准输入

int stdout\_copy = dup(1);//备份标准输出

int FI=-1,FO=-1;//重定向文件描述符

//处理重定向

if(CMDlist->cmd[0].RedirectionIn){

close(0);//关闭标准输入

int FI=open(CMDlist->cmd[0].RedirectionInFile,O\_RDONLY/\*只读\*/);

}

if(CMDlist->cmd[0].RedirectionOut){

close(1);//关闭标准输入

int FO=open(CMDlist->cmd[0].RedirectionOutFile,O\_WRONLY/\*只写\*/ | O\_CREAT/\*自动创建\*/ | O\_TRUNC/\*打开时清空\*/, 0666/\*所有用户有读写权限\*/);

}

if(CMDlist->cmd[0].RedirectionAppend){

close(1);

int FO=open(CMDlist->cmd[0].RedirectionOutFile, O\_RDWR/\*读写\*/ | O\_CREAT/\*自动创建\*/ | O\_APPEND/\*追加\*/, 0666);

}

//无管道内建命令，直接执行

if(CMDlist->cmd[0].PipIn==0&&CMDlist->cmd[0].PipOut==0&&BuildIn(CMDlist->cmd[0])){

if(FI>=0)close(FI);//恢复标准输入

if(FO>=0)close(FO);//恢复标准输出

dup2(stdin\_copy, 0);

dup2(stdout\_copy, 1);

close(stdin\_copy);

close(stdout\_copy);

return;

}

pid\_t pid = fork();//创建子进程

if(pid==0){//在新进程中执行

usleep(1);//睡眠1微秒，防止子进程先于父进程执行

// FILE \*fp=fopen("temp.txt","w");//打开临时文件

// fprintf(fp,"pid is %d\nnum %d in job list",getpid(),FindJob(getpid()));

// fclose(fp);

while(1){

int temp=FindJob(getpid());

if(temp!=-1&&JobList->List[temp].status==RUN){//在进程表中已经有此项

break;

}

}

//无管道外部命令

if(CMDlist->cmd[0].PipIn==0&&CMDlist->cmd[0].PipOut==0){//开头没有管道，后面也不会有管道

//处理重定向

if(CMDlist->cmd[0].RedirectionIn){

close(0);//关闭标准输入

int FI=open(CMDlist->cmd[0].RedirectionInFile,O\_RDONLY/\*只读\*/);

}

if(CMDlist->cmd[0].RedirectionOut){

close(1);//关闭标准输入

int FO=open(CMDlist->cmd[0].RedirectionOutFile,O\_WRONLY/\*只写\*/ | O\_CREAT/\*自动创建\*/ | O\_TRUNC/\*打开时清空\*/, 0666/\*所有用户有读写权限\*/);

}

if(CMDlist->cmd[0].RedirectionAppend){

close(1);

int FO=open(CMDlist->cmd[0].RedirectionOutFile, O\_RDWR/\*读写\*/ | O\_CREAT/\*自动创建\*/ | O\_APPEND/\*追加\*/, 0666);

}

//查找命令并执行

// char\* argv[MAXSECTION];

// for(int j=0;j<CMDlist->cmd[0].num;j++){//生成参数列表

// strcpy(argv[j],CMDlist->cmd[0].SplitCMD[j]);

// }

// argv[CMDlist->cmd[0].num]=NULL;//参数列表最后一位为NULL

// execvp(CMDlist->cmd[0].SplitCMD[0]/\*文件名\*/,argv/\*参数列表\*/);//execvp()会从环境变量所指的目录中查找符合参数 file 的文件名, 找到后执行该文件

exec(CMDlist->cmd[0]);

}

//有管道内部命令

else{

for(int i=0;i<=CMDlist->num;i++){

pid\_t pid\_son=fork();

if(pid\_son==0){//子进程

//输入

if(i){//只要不是第一个文件，都要去共享文件读输入

close(0);

int FI=open(PipePath,O\_RDONLY); //只读打开管道文件

}

else if(CMDlist->cmd[0].RedirectionIn){//第一个文件有重定向输入

close(0);//关闭标准输入

int FI=open(CMDlist->cmd[0].RedirectionInFile,O\_RDONLY/\*只读\*/);

}

//输出

if(i!=CMDlist->num){//只要不是最后一个文件，都要去共享文件写输出

close(1);

int FO=open(PipePath,O\_WRONLY/\*只写\*/ | O\_CREAT/\*自动创建\*/ | O\_TRUNC/\*打开时清空\*/, 0666/\*所有用户有读写权限\*/);

}

else if(CMDlist->cmd[CMDlist->num].RedirectionOut){//最后一个文件有重定向输出

close(1);

int FO=open(CMDlist->cmd[CMDlist->num].RedirectionOutFile,O\_WRONLY/\*只写\*/ | O\_CREAT/\*自动创建\*/ | O\_TRUNC/\*打开时清空\*/, 0666/\*所有用户有读写权限\*/);

}

else if(CMDlist->cmd[CMDlist->num].RedirectionAppend){//最后一个文件有重定向输出,追加

close(1);

int FO=open(CMDlist->cmd[CMDlist->num].RedirectionOutFile, O\_RDWR/\*读写\*/ | O\_CREAT/\*自动创建\*/ | O\_APPEND/\*追加\*/, 0666);

}

//查找命令并执行

BuildIn(CMDlist->cmd[i]);

}

else{//父进程

waitpid(pid\_son,NULL,0);//等待子进程完成

}

}

}

if(FI>=0)close(FI);//恢复标准输入

if(FO>=0)close(FO);//恢复标准输出

dup2(stdin\_copy, 0);

dup2(stdout\_copy, 1);

close(stdin\_copy);

close(stdout\_copy);

exit(0);//退出子进程

}

else{//在父进程执行

// FILE \*fp=fopen("temp.txt","w");//打开临时文件

// fprintf(fp,"111");

// fclose(fp);

JobList->List[JobList->num].pid=pid;//将子进程加入进程表

strcpy(JobList->List[JobList->num].name,CMDlist->cmd[0].SplitCMD[0]);

JobList->List[JobList->num].type=CMDlist->type;

JobList->num+=1;

getcwd(path,sizeof(path));//得到当前目录

setenv("PARENT", path, 1);//设置父进程环境变量

if(CMDlist->type==BG){//是后台进程

JobList->List[FindJob(pid)].status=RUN;//状态为运行,可以开始运行

}

else{//是前台进程

JobList->List[FindJob(pid)].status=RUN;//状态为运行,可以开始运行

waitpid(pid,NULL,WUNTRACED);//等待子进程结束

DeleteJob(pid);

}

}

return;

}

int FindJob(int pid){

for(int i=JobList->num-1;i>=0;i--){

if(JobList->List[i].pid==pid) return i;

}

return -1;

}

void DeleteJob(pid\_t pid){//删除进程表中的进程(前台进程)

int i;

for(i=JobList->num-1;i>=0;i--){//从后往前找

if(JobList->List[i].pid==pid) break;

}

for(;i<JobList->num-1;i++){//将后面的进程向前移动

JobList->List[i]=JobList->List[i+1];

}

JobList->num-=1;

}

int BuildIn(struct command cmd){//内建命令

if(strcmp(cmd.SplitCMD[0],"bg")==0) {bg(); return 1;}

else if(strcmp(cmd.SplitCMD[0],"clr")==0){clr();return 1;}

else if(strcmp(cmd.SplitCMD[0],"cd")==0){cd(cmd.SplitCMD[1]);return 1;}

else if(strcmp(cmd.SplitCMD[0],"dir")==0){dir(cmd.SplitCMD[1]);return 1;}

else if(strcmp(cmd.SplitCMD[0],"echo")==0){echo(cmd);return 1;}

else if(strcmp(cmd.SplitCMD[0],"exit")==0)exit(0);

else if(strcmp(cmd.SplitCMD[0],"exec")==0){exec(cmd);return 1;}

else if(strcmp(cmd.SplitCMD[0],"fg")==0){fg();return 1;}

else if(strcmp(cmd.SplitCMD[0],"help")==0){help(cmd.SplitCMD[1]);return 1;}

else if(strcmp(cmd.SplitCMD[0],"jobs")==0){jobs();return 1;}

else if(strcmp(cmd.SplitCMD[0],"pwd")==0){pwd();return 1;}

else if(strcmp(cmd.SplitCMD[0],"set")==0){set(cmd.SplitCMD[1]);return 1;}

else if(strcmp(cmd.SplitCMD[0],"test")==0){if(test(cmd)) printf("TRUE\n");else printf("FALSE\n");return 1;}

else if(strcmp(cmd.SplitCMD[0],"time")==0){Time();return 1;}

else if(strcmp(cmd.SplitCMD[0],"umask")==0){Umask(cmd.SplitCMD[1]);return 1;}

else return 0;

}

void bg(){

for(int i=JobList->num-1;i>0;i--){//从后往前找，找到第一个SUSPEND的后台进程，将其状态改为前台RUN

if(JobList->List[i].status==SUSPEND&&JobList->List[i].type==BG){

JobList->List[i].status = RUN;

kill(JobList->List[i].pid,SIGCONT);//用kill函数发送信号，重新激活进程

printf("[%d] %d\t\t%s\n",i,JobList->List[i].pid,JobList->List[i].name);

break;

}

}

}

void clr(){

printf("\033[2J");//清屏

printf("\033[0;0H");//光标移动到左上角

}

void cd(char\* path){

if(chdir(path)==-1) printf("error: failed to change directory\n");//改变工作目录

else{

setenv("PWD",getcwd(NULL,0), 1);//设置工作目录环境变量(后面能被getcwd读取)

}

}

void dir(char\* path){

DIR \*dir;//目录指针

if(strcmp(path,"\0")==0) dir=opendir(".");//打开当前目录

else dir=opendir(path);//打开指定目录

struct dirent \*ptr;//目录结构体指针

if(dir==NULL) {printf("error: failed to open directory\n");return;}

while((ptr=readdir(dir))!=NULL){

if(ptr->d\_name[0]!='.')//不显示隐藏文件

printf("%s\n",ptr->d\_name);

}

closedir(dir);

}

void echo(struct command cmd){

if (cmd.num == 1) {//没有参数

printf("\n");

return;

}

else if(cmd.SplitCMD[1][0]=='$'){//环境变量

char \*temp=getenv(cmd.SplitCMD[1]+1);//得到环境变量的值

if(temp==NULL) printf("\n");

else printf("%s\n",temp);

return;

}

else if(cmd.SplitCMD[1][0]=='\"'){//字符串

for(int i=2;i<cmd.num&&cmd.SplitCMD[i][0]!='\"'/\*最后一个是引号\*/;i++){

printf("%s ",cmd.SplitCMD[i]);

}

}

else{//只有一个字符串

printf("%s\n",cmd.SplitCMD[1]);

}

printf("\n");

}

void exec(struct command cmd){

char \*argv[MAXSECTION];

for(int j=2;j<cmd.num;j++){//生成参数列表

argv[j-2]=(char\*)malloc(sizeof(char)\*MAXLENGTH);

strcpy(argv[j-2],cmd.SplitCMD[j]);

}

argv[cmd.num-2]=NULL;//参数列表最后一位为NULL

execvp(cmd.SplitCMD[1]/\*文件名\*/,argv/\*参数列表\*/);//execvp()会从环境变量所指的目录中查找符合参数 file 的文件名, 找到后执行该文件

}

void fg(){

for(int i=JobList->num-1;i>0;i--){//从后往前找，找到第一个SUSPEND的后台进程，将其状态改为前台RUN

if(JobList->List[i].status==SUSPEND&&JobList->List[i].type==BG){

JobList->List[i].status = RUN;

JobList->List[i].type = FG;

kill(JobList->List[i].pid,SIGCONT);//用kill函数发送信号，重新激活进程

printf("[%d] %d\t\t%s\n",i,JobList->List[i].pid,JobList->List[i].name);

waitpid(JobList->List[i].pid,NULL,WUNTRACED);//等待子进程结束

DeleteJob(JobList->List[i].pid);

break;

}

}

}

void help(char\* str){

FILE\* fp=fopen("README.md","r");//打开用户手册

char temp[MAXLENGTH];

while(!feof(fp)){

fgets(temp,MAXLENGTH,fp);//读取一行

if(strncmp(temp,str,strlen(str)/\*前几个字符匹配\*/)==0)

{printf("%s\n",temp);break;}//如果包含关键词，输出

}

}

void jobs(){

for(int i=JobList->num-1;i>0;i--){

if(JobList->List[i].type==BG){//显示后台进程

printf("[%d] %d\t\t%s\n",i,JobList->List[i].pid,JobList->List[i].name);

}

}

}

void pwd(){

printf("%s\n",getenv("PWD"));//输出工作目录

}

void set(char\* str){

char\* temp=strtok(str,"=");//分解字符串

char\* temp2=strtok(NULL,"=");

setenv(temp,temp2,1);//设置环境变量

}

int test(struct command cmd){

if(strcmp(cmd.SplitCMD[2],"-eq")==0){

return strcmp(cmd.SplitCMD[1],cmd.SplitCMD[3])==0;

}

else if(strcmp(cmd.SplitCMD[2],"-ne")==0){

return strcmp(cmd.SplitCMD[1],cmd.SplitCMD[3])!=0;

}

else if(strcmp(cmd.SplitCMD[2],"-gt")==0){

return strcmp(cmd.SplitCMD[1],cmd.SplitCMD[3])>0;

}

else if(strcmp(cmd.SplitCMD[2],"-lt")==0){

return strcmp(cmd.SplitCMD[1],cmd.SplitCMD[3])<0;

}

else if(strcmp(cmd.SplitCMD[2],"-ge")==0){

return strcmp(cmd.SplitCMD[1],cmd.SplitCMD[3])>=0;

}

else if(strcmp(cmd.SplitCMD[2],"-le")==0){

return strcmp(cmd.SplitCMD[1],cmd.SplitCMD[3])<=0;

}

}

void Time(){

time\_t t=time(0);

char\* date=(char\*)ctime(&t);//将时间转换为字符串

printf("%s\n",date);//输出当前时间

}

void Umask(char\* str){

mode\_t mode;

if(strcmp(str,"")==0) //如果没有参数，就将umask设置为最大权限

{

mode=umask(0);//umask(0)意思就是0取反再创建文件时权限相与，即0777&这个文件，得到这个文件的权限

umask(mode);

printf("%04o\n",mode);

}

else{

mode=atoi(str);//将字符串转换为数字

umask(mode);

}

}