**操作系统课程设计实验报告**

——实验一：shell实验

负责人姓名：常涛

学号：14061161

日期：2016.3.25

**小组成员**

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 姓名 | 学号 | 实验分工 |
| 1 | 于东方 | 14061149 | 待定 |
| 2 | 牛钰浩 | 14061155 | 待定 |
| 3 | 常涛 | 14061161 | 实验一 |
| 4 | 姜铭 | 14061167 | 待定 |

目录

[目录 3](#_Toc14166)

[1.实验目的 5](#_Toc6867)

[2.需求说明 5](#_Toc18827)

[2.1基本要求 5](#_Toc9733)

[2.2 提高要求 7](#_Toc14882)

[2.3 完成情况 7](#_Toc5854)

[3.设计说明 8](#_Toc13408)

[3.1 程序流程图 8](#_Toc998)

[3.2基本要求实现说明 9](#_Toc5026)

[3.2.1命令提示符 9](#_Toc17969)

[3.2.2内部命令（exit、jobs、history、fg、bg） 10](#_Toc13103)

[3.2.3前台和后台作业切换 13](#_Toc8291)

[3.2.4实现对I/O重定向的支持 14](#_Toc13738)

[3.2.5使用YACC工具实现user-sh的语法分析功能 15](#_Toc13027)

[3.3 提高要求实现说明 18](#_Toc18690)

[3.3.1对YACC分析的文法进一步修改与完善 18](#_Toc20084)

[3.3.2在Linux下将Lex和YACC结合起来进行词法和语法分析 18](#_Toc21278)

[3.3.3支持通配符\*?[]{} 19](#_Toc210)

[3.3.4对管道的支持 25](#_Toc5261)

[3.3.5实现组合键Ctrl+C 32](#_Toc21257)

[3.3.6实现进程组、会晤、作业控制 35](#_Toc31229)

[3.3.7实现内部命令pwd、type 37](#_Toc12184)

[4.问题及解决方法 42](#_Toc23126)

[4.1 源代码中存在的结构性问题 42](#_Toc22177)

[4.2 lex的加入和EOF的问题 45](#_Toc3742)

[4.3对堆内存的不规范使用 45](#_Toc14522)

[4.4 进程组设置过程中遇到的问题 46](#_Toc31257)

[4.5有关malloc()产生的问题 47](#_Toc5400)

[5.收获和感想 47](#_Toc3130)

1.实验目的

1.学习LInux相关软件工具的使用，如gcc、gdb和make

2.熟悉使用Linux中YACC工具进行语法分析的基本方法

3.运用man帮助手册查询相关命令

4.理解并发程序的同步问题

5.学习POSIX/UNIX系统调用的使用方法

6.掌握进程口止和进程间通信的方法

2.需求说明

2.1基本要求

本实验要求在Linux环境中使用C语言编写一个简单的shell命令解释器程序，称之为user-sh。其设计类似与目前流行的shell解释程序，如bash。user-sh程序应具有如下重要特征：

1.能够执行fg、bg、cd、history、exit等内部命令。

2.能够执行外部程序命令，命令可带参数。

3.使用I/O重定向和管道

4.支持前后台作业，提供作业控制功能，包括打印作业清单，改变当前运 行作业的前台/后台状态，以及控制作业的挂起、终止和继续运行。

5.使用make工具建立工程。

6.使用gdb等调试工具来调试程序

7.提供清晰、详细的设计文档和解决方案

8.锻炼团队协作开发能力

shell程序的具体要求如下：

(1)本实验的user-sh程序设计不包括对配置文件和命令行参数的支持。user-sh应提供一个命令提示符例如：“user-sh>”，表示等待用户的输入，执行命令输出必要的信息，然后再打印下一个命令提示符。当用户没有输入时，user-sh需要一直处于随时等待输入状态，同时在屏幕上显示一些基本提示信息。

(2)实现以下内部命令。

1）exit

结束所有的子进程并退出user-sh。

2）jobs

打印当前正在后台执行的作业和被挂起的作业信息。输出信息应采用便于用户理解的格式。jobs自身是一条内部命令，所以不需要显示在输出上。

3）history

列出用户最近输入过的N条命令，不论这个命令是否正确执行过。

4）fg%<int>

把<int>所标识的作业放到前台运行。若这个作业原来已经挂起则让其继续运行。user-sh应当在打印新的命令提示符之前等待前台运行的子进程结束。

5）bg<int>

把<int>所标识的已挂起的进程放在后台运行。

(3)进行前后台作业的切换。

执行前台作业即在提示符后输入一个可执行文件的路径（绝对路径）即可，执行后台作业则需在可执行文件路径后加一个”&”符号。

user-sh通过组合键实现前/后台作业切换：

1）Ctrl+Z

产生SIGTSTP信号，这个信号不是挂起user-sh，而是让user-sh挂起在前台运行的作业，如果没有任何前台作业，则该特殊键无效。

2）Ctrl+C

产生SIGINT信号，这个信号不终止user-sh，而是通过user-sh发出信号杀死前台作业中的进程。如果没有任何前台作业，则该特殊键无效。

(4)实现对I/O重定向的支持

一个命令后面可能还跟有元字符”<”或”>”，他们是重定向符号，而重定向符号后面还跟着一个文件名。

1）<

程序的输入被重定向到一个制定的文件中。

2）>

程序的输出被重定向到一个制定的文件中。如果输出文件不存在，需要创建一个输出输出文件。如果输入文件不存在，则认为命令出现错误。

(5)使用YACC工具实现user-sh的语法分析功能。

2.2 提高要求

（1）尝试对YACC语法分析的文法进行进一步的修改与完善。

（2）尝试在Linux下将Lex和YACC结合起来使用进行词法和语法分析。

（3）对其他常用的内部命令进行实现，并可以尝试考虑通配符支持与实现

（4）实现对管道的支持。

当若干个命令被元字符”|”分割开时，他们可以放在同一条命令行当中，这个元字符代表管道符号。在这种情况下，user-sh为每一个子命令都创建一个进程，并把它们的I/O用管道连接起来。这种命令行可以通过负责进程间通信的管道来实现。由管道连接的多个进程所组成的作业只有当其所有子进程都执行完毕时才算结束。

（5）参照组合键Ctrl+Z命令实现方法，考虑并实现组合键Ctrl+C。

（6）其他自行提出的改进。

2.3 完成情况

能够完成本次实验的所有基本要求，并在此基础上增加了以下东西：

1.完成对YACC分析的文法的进一步修改与完善

2实现了Linux下Lex和YACC结合起来进行词法和语法分析

3..能够完成内部命令type、kill，并支持通配符\*?[]{}，但不支持涉及目录 的含通配符命令

4.实现了管道的支持，可以将两条命令用“|”隔开，放在一行执行

5.实现了组合键Ctrl+C命令

6.实现进程组、会晤、作业控制

3.设计说明

3.1 程序流程图

yyparse

main

execute,

parse

execSimpleCmd

内部命令

外部命令

execOuterCmd

创建管道

fork1()

SIGSTOP1

重定向（插入管道）

wait1()

fork2()

重定向（插入管道）

SIGSTOP2

wait2()

放入前台进程组

前后台处理

addJob

SIGCONT

continue2

continue1

3.2基本要求实现说明

3.2.1命令提示符

Bison.y:

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

init(); //初始化环境

reset();//初始化相应变量

commandDone = 0;

while(1){

char c;

SIGCHLD\_Handler();

printf("\033[1;32mB529-sh >\033[0m");

if((c=getchar())==EOF){

continue;

}else{

ungetc(c,stdin);

}

yyparse(); //调用语法分析函数，该函数由yylex()提供当前输入的单词符号

if(commandDone == 1){ //命令已经执行完成后，添加历史记录信息

commandDone = 0;

addHistory(inputBuff);

}

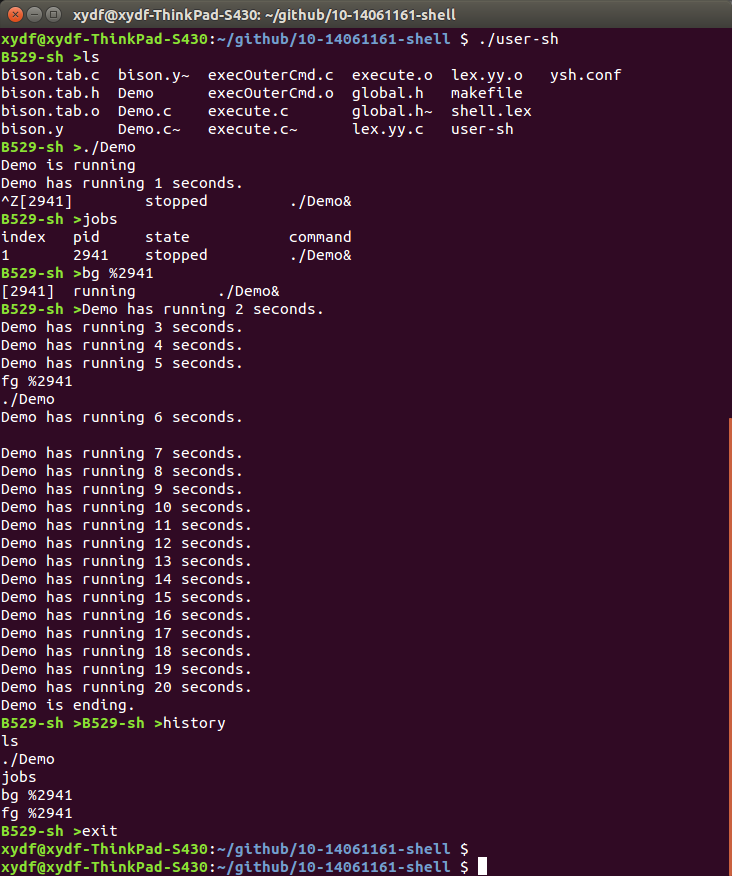
reset();//重置相关变量，为下一次处理做准备

}

return (EXIT\_SUCCESS);

}

3.2.2内部命令（exit、jobs、history、fg、bg）

execute.c:

void execSimpleCmd(SimpleCmd \*cmd){

int i, pid;

char \*temp;

Job \*now = NULL;

if(cmd->isPipe)

{

if(

strcmp(cmd->args[0], "exit") == 0 ||

strcmp(cmd->args[0], "history") == 0 ||

strcmp(cmd->args[0], "jobs") == 0 ||

strcmp(cmd->args[0], "cd") == 0 ||

strcmp(cmd->args[0], "fg") == 0 ||

strcmp(cmd->args[0], "bg") == 0 ||

strcmp(cmd->pipeEnd->args[0], "exit") == 0 ||

strcmp(cmd->pipeEnd->args[0], "history") == 0 ||

strcmp(cmd->pipeEnd->args[0], "jobs") == 0 ||

strcmp(cmd->pipeEnd->args[0], "cd") == 0 ||

strcmp(cmd->pipeEnd->args[0], "fg") == 0 ||

strcmp(cmd->pipeEnd->args[0], "bg") == 0 )

{

fprintf(stderr,"内部命令不支持管道");

return;

}

}//$

if(strcmp(cmd->args[0], "exit") == 0) { //exit命令

if(head != NULL){

printf("尚有进程正在执行\n");//$$

return;

}

exit(0);

} else if (strcmp(cmd->args[0], "history") == 0) { //history命令

if(history.end == -1){

printf("尚未执行任何命令\n");

return;

}

i = history.start;

do {

printf("%s\n", history.cmds[i]);

i = (i + 1)%HISTORY\_LEN;

} while(i != (history.end + 1)%HISTORY\_LEN);

} else if (strcmp(cmd->args[0], "jobs") == 0) { //jobs命令

if(head == NULL){

printf("尚无任何作业\n");

} else {

printf("index\tpid\tstate\t\tcommand\n");

for(i = 1, now = head; now != NULL; now = now->next, i++){

printf("%d\t%d\t%s\t\t%s\n", i, now->pid, now->state, now->cmd);

}

}

} else if (strcmp(cmd->args[0], "cd") == 0) { //cd命令

temp = cmd->args[1];

if(temp != NULL){

if(chdir(temp) < 0){

printf("cd; %s 错误的文件名或文件夹名！\n", temp);

}

}

} else if (strcmp(cmd->args[0], "fg") == 0) { //fg命令

temp = cmd->args[1];

if(temp != NULL && temp[0] == '%'){

pid = str2Pid(temp, 1, strlen(temp));

if(pid != -1){

fg\_exec(pid);

}

}else{

printf("fg; 参数不合法，正确格式为：fg %%<int>\n");

}

} else if (strcmp(cmd->args[0], "bg") == 0) { //bg命令

temp = cmd->args[1];

if(temp != NULL && temp[0] == '%'){

pid = str2Pid(temp, 1, strlen(temp));

if(pid != -1){

bg\_exec(pid);

}

}

else{

printf("bg; 参数不合法，正确格式为：bg %%<int>\n");

}

} else{ //外部命令

execOuterCmd(cmd);

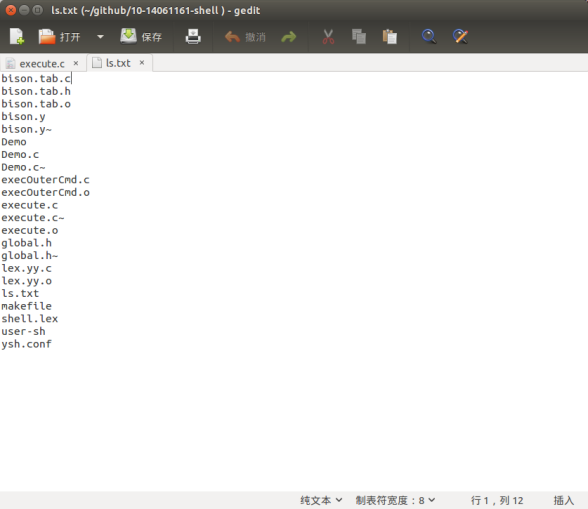
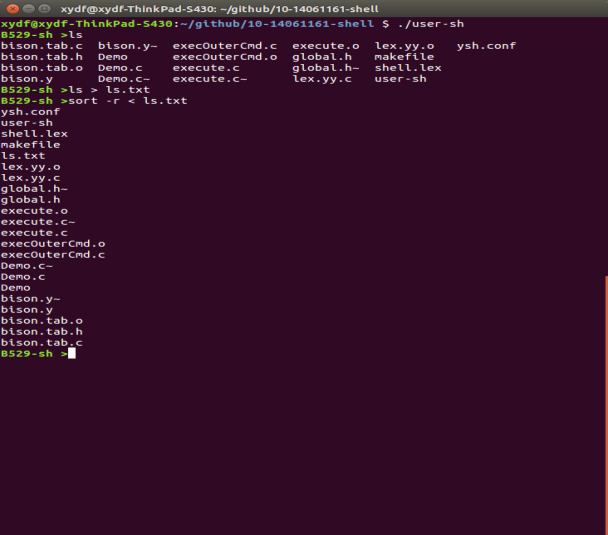
}

}

3.2.3前台和后台作业切换

见3.2.2的fg、bg部分。

3.2.4实现对I/O重定向的支持



execOuterCmd.c:

if(cmd->input != NULL){ //存在输入重定向

if((pipeIn = open(cmd->input, O\_RDONLY, S\_IRUSR|S\_IWUSR)) == -1){

printf("不能打开文件 %s！\n", cmd->input);

exit(EXIT\_FAILURE);

}

if(dup2(pipeIn, 0) == -1){

printf("重定向标准输入错误！\n");

exit(EXIT\_FAILURE);

}

}

if(cmd->output != NULL){ //存在输出重定向

if((pipeOut = open(cmd->output, O\_WRONLY|O\_CREAT|O\_TRUNC, S\_IRUSR|S\_IWUSR)) == -1){

printf("不能打开文件 %s！\n", cmd->output);

exit(EXIT\_FAILURE);

}

if(dup2(pipeOut, 1) == -1){

printf("重定向标准输出错误！\n");

exit(EXIT\_FAILURE);//neal 17:28 不应该是return;

}

}

3.2.5使用YACC工具实现user-sh的语法分析功能

bison.y:

%{

#include "global.h"

int yylex ();

void yyerror ();

SimpleCmd\* parse(char[][CHARS\_LEN],char[],char[],int,int,int);

void argsExtend(char\*);

void ioExtend(char \*,int);

char \*extend(char \*);

int commandDone;

void reset();

char inputBuff[100];//保存命令

char buff[BUFF\_LEN][CHARS\_LEN];

char inputFile[FNAME\_LEN];

char outputFile[FNAME\_LEN];

int args\_num=0;

char buff\_2[BUFF\_LEN][CHARS\_LEN];

char inputFile\_2[FNAME\_LEN];

char outputFile\_2[FNAME\_LEN];

int args\_num\_2=0;

int isBackFlag=0;

int isPipeFlag=0;

%}

%union{

char \*str;

}

%token <str> STRING

%%

line : /\* empty \*/

|command {execute(); commandDone = 1;}//

;

command : fgCommand

|bgCommand

;

bgCommand : fgCommand '&' {isBackFlag=1;}

;

fgCommand : simpleCmd

|pipeCmd

;

pipeCmd : simpleCmd '|' simpleCmd {isPipeFlag=1;}

;

simpleCmd : args

|args inputRedirect

|args outputRedirect

|args inputRedirect outputRedirect

;

inputRedirect : '<' STRING {ioExtend($2,1);}//1 : input

;

outputRedirect : '>' STRING {ioExtend($2,0);}//0 : output

;

args : STRING {argsExtend($1);}

|args STRING {argsExtend($2);}

;

%%

3.3 提高要求实现说明

3.3.1对YACC分析的文法进一步修改与完善

见3.2.5。

3.3.2在Linux下将Lex和YACC结合起来进行词法和语法分析

shell.lex:

%{

#include "global.h"

#include "bison.tab.h"

extern int isPipeFlag;

%}

%%

[\n] {strcat(inputBuff,"\0");return 0;}

[\t ]+ {strcat(inputBuff,yytext); }

[\|] {isPipeFlag=1;strcat(inputBuff,yytext);return \*yytext;}

[><&] {strcat(inputBuff,yytext);return \*yytext;}

[^\|\n\t <>&]+ {strcat(inputBuff,yytext);yylval.str=yytext;return STRING;}

%%

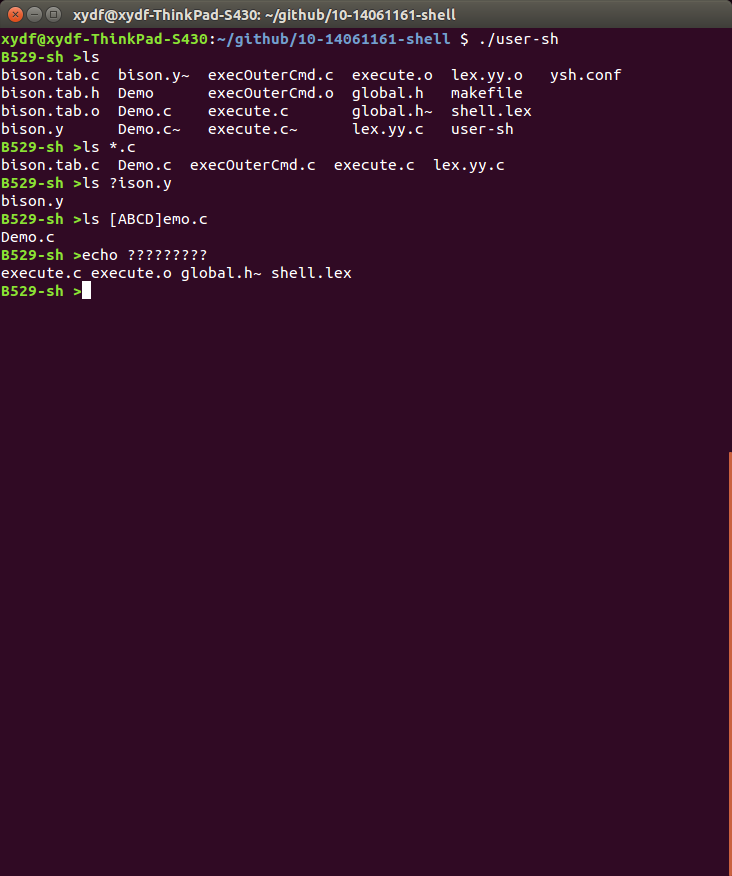
int yywrap()

{

return 1;

}

3.3.3支持通配符\*?[]{}



bison.y

void argsExtend(char \*args)

{

int i=0;

int j=0;

char \*extendArgs=extend(args);

if(extendArgs==NULL)

return ;

for(i=0;i<strlen(extendArgs);i++){

if(extendArgs[i]=='\n'){

if(0==isPipeFlag){

buff[args\_num][j]='\0';//获得一个args，放到buff中

args\_num++;

if(args\_num>=BUFF\_LEN){//如果args太多，不继续存放

fprintf(stderr, "参数过多\n");

break;

}

}else{

buff\_2[args\_num\_2][j]='\0';

args\_num\_2++;

if(args\_num\_2>=BUFF\_LEN){//如果args太多，不继续存放

fprintf(stderr, "参数过多\n");

break;

}

}

j=0;

}else{

if(0==isPipeFlag){

buff[args\_num][j]=extendArgs[i];

}else{

buff\_2[args\_num\_2][j]=extendArgs[i];

}

j++;

if(j>=CHARS\_LEN){

fprintf(stderr, "参数过长\n");

j--;

}

}

}

free(extendArgs);

extendArgs=NULL;

}

char \*extend(char \*args)

{

////////字符串处理

int length =strlen(args);

char tmp[200];

int i,j;

char \*patternArg;

int fd1[2];

int fd2[2];

int pipe\_pid[2];

char \*result;

i=0,j=0;

tmp[j++]='^';//开始符号

for(i=0;i<length;i++)

{

switch(args[i]){

case '.':tmp[j++]='\\';

tmp[j++]='.';

break;

case '\*':tmp[j++]='.';

tmp[j++]='\*';

break;

case '?':tmp[j++]='.';

break;

case '%':tmp[j++]='%';

tmp[j++]='%';

break;

default :tmp[j++]=args[i];

}

}

tmp[j++]='$';//结束符号

tmp[j]='\0';

patternArg=(char\*)malloc(sizeof(char)\*(j+1));

memcpy(patternArg,tmp,j+1);

////////管道处理

if(pipe(fd1)<0)

{

printf("pipe1 failed");

}

if(pipe(fd2)<0)

{

printf("pipe2 failed");

}

if((pipe\_pid[0]=fork())==-1)

{

printf("fork ls failed");

}

if(!pipe\_pid[0])

{

close(fd1[0]);

close(STD\_OUTPUT);

dup2(fd1[1],1); //把标识符1作为管道的另一个写指针

close(fd1[1]); //关闭管道1原有的写指针

execl("/bin/ls","-a",NULL);

}

if((pipe\_pid[1]=fork())==-1)

{

printf("fork grep failed");

}

if(!pipe\_pid[1])

{

close(fd1[1]);

close(STD\_INPUT);

dup2(fd1[0],0); //把标识符0作为管道1的另一个读指针

close(fd1[0]); //关闭管道1原有的读指针

/////////////

close(fd2[0]);

dup2(fd2[1],1); //把标识符1作为管道2的另一个写指针

close(fd2[1]); //关闭管道2原有的写指针

/////////////

execl("/bin/grep","-P",patternArg,NULL);

}

close(fd1[0]);

close(fd1[1]);

close(fd2[1]); //关闭管道2写端

result = (char \*)malloc(sizeof(char)\*3000);

for(i=0;i<3000;i++){//申请得到的内存里,内容是随机的

result[i]='\0';

}

waitpid(pipe\_pid[1],NULL,0);

waitpid(pipe\_pid[0],NULL,0);

if(read(fd2[0],result,3000)>0){ //如果result[]倒数第二项是'\n'(result[]最后一项'\0')，则有扩展匹配项

;

}else{ //无扩展匹配项,返回原来匹配项

for(i=0;i<strlen(args);i++){

result[i]=args[i];

}

result[i]='\n';

i++;

result[i]='\0';

}

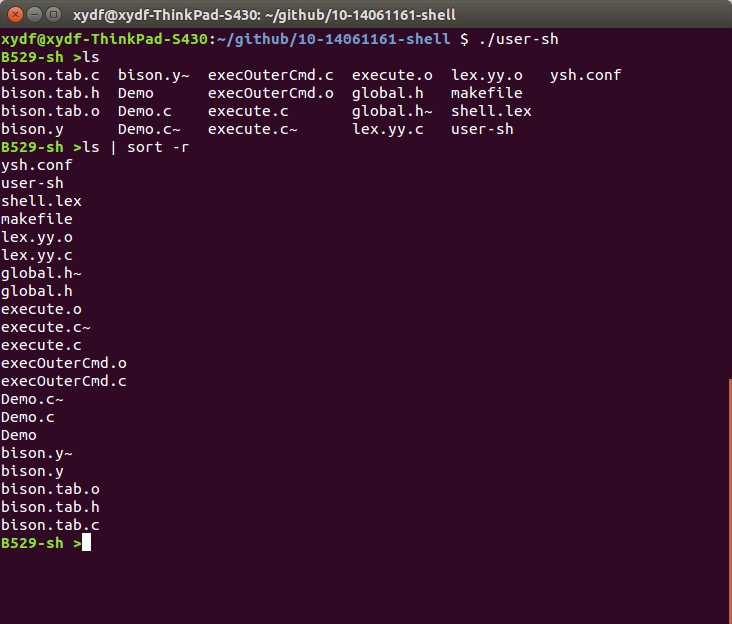
free(patternArg);

patternArg=NULL;

return result;

}:

3.3.4对管道的支持

execOuterCmd.c:

#include "global.h"

extern char cmdBuff[40];

/\*执行外部命令\*/

void execOuterCmd(SimpleCmd \*cmd){

pid\_t pid=0,pid2=0;//neal +pid2

int pipeIn, pipeOut;

int pipe\_fd[2];

//如果命令不存在

if(!(

exists(cmd->args[0]) &&

((!cmd->isPipe)||exists(cmd->pipeEnd->args[0]))

)){

printf("找不到命令 %15s\n", inputBuff);

return;

}

//如果是管道命令，申请管道

if(pipe(pipe\_fd)<0)

{

perror("pipe error.\n");

return;

}

if((pid = fork()) < 0){

perror("fork failed");

if(cmd->isPipe){

close(pipe\_fd[0]);

close(pipe\_fd[1]);

}

return;

}

if(pid == 0){ //子进程1

if(cmd->input != NULL){ //存在输入重定向

if((pipeIn = open(cmd->input, O\_RDONLY, S\_IRUSR|S\_IWUSR)) == -1){

printf("不能打开文件 %s！\n", cmd->input);

exit(EXIT\_FAILURE);

}

if(dup2(pipeIn, 0) == -1){

printf("重定向标准输入错误！\n");

exit(EXIT\_FAILURE);

}

}

if(cmd->output != NULL){ //存在输出重定向

if((pipeOut = open(cmd->output, O\_WRONLY|O\_CREAT|O\_TRUNC, S\_IRUSR|S\_IWUSR)) == -1){

printf("不能打开文件 %s！\n", cmd->output);

exit(EXIT\_FAILURE);

}

if(dup2(pipeOut, 1) == -1){

printf("重定向标准输出错误！\n");

exit(EXIT\_FAILURE);//neal 17:28 不应该是return;

}

}

if(cmd->isPipe){

if(cmd->output != NULL){

close(pipe\_fd[0]);

close(pipe\_fd[1]);

}else{

close(pipe\_fd[0]);

if(dup2(pipe\_fd[1], STDOUT\_FILENO) == -1){

printf("重定向到管道错误！\n");

exit(EXIT\_FAILURE);

}

close(pipe\_fd[1]);

}

}

kill(getpid(),SIGSTOP);//初始化完毕，暂停运行

signal(SIGCHLD, SIG\_DFL);//$$

signal(SIGTSTP, SIG\_DFL);//$$

signal(SIGTTOU, SIG\_DFL);//$$为了成功调用tcsetpgrp()

signal(SIGTTIN, SIG\_DFL);//$$

signal(SIGQUIT, SIG\_DFL);//$$

signal(SIGINT, SIG\_DFL);//$$

exists(cmd->args[0]);

justArgs(cmd->args[0]);

if(execv(cmdBuff, cmd->args) < 0){ //执行命令

printf("execv failed!\n");

exit(EXIT\_FAILURE);//neal 17:28 不应该是return;

}

}

//父进程等待子进程初始化

int status1;

waitpid(pid,&status1,WUNTRACED);

if(WIFEXITED(status1)||WIFSIGNALED(status1)){

if(cmd->isPipe){

close(pipe\_fd[0]);

close(pipe\_fd[1]);

}

return;

}

//如果是管道命令，父进程继续fork()

if(cmd->isPipe){

if((pid2 = fork()) < 0){

perror("fork failed");

close(pipe\_fd[0]);

close(pipe\_fd[1]);

kill(pid,SIGKILL);

return;

}

if(pid2==0){//子进程2

if(cmd->pipeEnd->input != NULL){ //存在输入重定向

if((pipeIn = open(cmd->pipeEnd->input, O\_RDONLY, S\_IRUSR|S\_IWUSR)) == -1){

printf("不能打开文件 %s！\n", cmd->pipeEnd->input);

exit(EXIT\_FAILURE);

}

if(dup2(pipeIn, 0) == -1){

printf("重定向标准输入错误！\n");

exit(EXIT\_FAILURE);

}

}

if(cmd->pipeEnd->output != NULL){ //存在输出重定向

if((pipeOut = open(cmd->pipeEnd->output, O\_WRONLY|O\_CREAT|O\_TRUNC, S\_IRUSR|S\_IWUSR)) == -1){

printf("不能打开文件 %s！\n", cmd->pipeEnd->output);

exit(EXIT\_FAILURE);

}

if(dup2(pipeOut, 1) == -1){

printf("重定向标准输出错误！\n");

exit(EXIT\_FAILURE);//neal 17:28 不应该是return;

}

}

if(cmd->isPipe){

if(cmd->pipeEnd->input != NULL){

close(pipe\_fd[0]);

close(pipe\_fd[1]);

}else{

close(pipe\_fd[1]);

if(dup2(pipe\_fd[0], STDIN\_FILENO) == -1){

printf("重定向到管道错误！\n");

exit(EXIT\_FAILURE);

}

close(pipe\_fd[0]);

}

}

kill(getpid(),SIGSTOP);//初始化完毕，暂停运行

signal(SIGCHLD, SIG\_DFL);//$$恢复默认信号处理方式

signal(SIGTSTP, SIG\_DFL);//$$

signal(SIGTTOU, SIG\_DFL);//$$为了成功调用tcsetpgrp()

signal(SIGTTIN, SIG\_DFL);//$$

signal(SIGQUIT, SIG\_DFL);//$$

signal(SIGINT, SIG\_DFL);//$$

exists(cmd->pipeEnd->args[0]);

justArgs(cmd->pipeEnd->args[0]);

if(execv(cmdBuff, cmd->pipeEnd->args) < 0){ //执行命令

printf("execv failed!\n");

exit(EXIT\_FAILURE);//neal 17:28 不应该是return;

}

}

int status2;

waitpid(pid2,&status2,WUNTRACED);

if(WIFEXITED(status2)||WIFSIGNALED(status2)){

close(pipe\_fd[0]);

close(pipe\_fd[1]);

kill(pid,SIGKILL);

return;

}

}

//父进程 设置子进程进程组

setpgid(pid,pid);

if(cmd->isPipe){

close(pipe\_fd[0]);

close(pipe\_fd[1]);

setpgid(pid2,pid);

}

if(!cmd->isBack){//前台任务，设为前台进程组

if(tcsetpgrp(TERM\_FILENO, pid)<0){

perror("tcsetpgrp():");

kill(pid,SIGKILL);

kill(pid2,SIGKILL);

return;

}

}else{//后台任务

printf("[%d]\t%s\t\t%s\n", pid, RUNNING, inputBuff);

}

addJob(pid,pid2,cmd->isBack);

kill(pid,SIGCONT);

if(cmd->isPipe){

kill(pid2,SIGCONT);

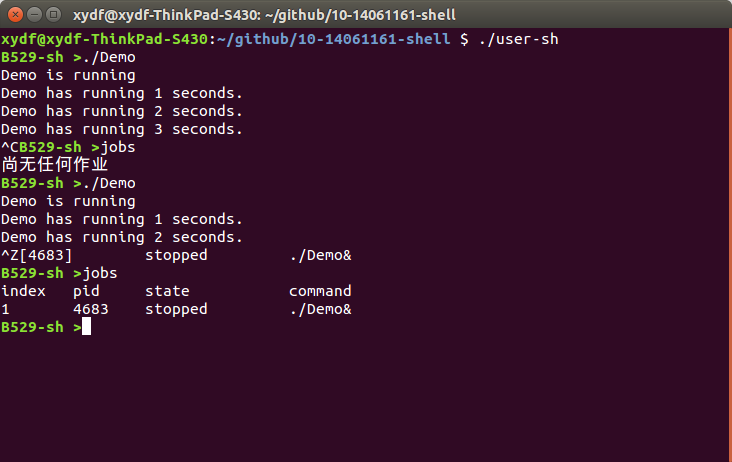
}

if(!cmd->isBack)

waiting(pid,pid2);

}

3.3.5实现组合键Ctrl+C

execute.c: //添加了一行代码

void init(){

int fd, len;

char c, buf[80];

//打开查找路径文件ysh.conf

if((fd = open("ysh.conf", O\_RDONLY, 660)) == -1){

perror("init environment failed\n");

exit(1);

}

//初始化history链表

history.end = -1;

history.start = 0;

len = 0;

//将路径文件内容依次读入到buf[]中

while(read(fd, &c, 1) != 0){

buf[len++] = c;

}

buf[len] = '\0';

//将环境路径存入envPath[]

getEnvPath(len, buf);

//检测是否为前台

while(getpgrp()!=tcgetpgrp(TERM\_FILENO))

kill(0,SIGTTIN);

//将shell放入自己的进程组

if(setpgid(0,0)<0){

perror("setpgid():");

exit(EXIT\_FAILURE);

}

//注册信号

/\*struct sigaction action;

action.sa\_sigaction = rmJob;

sigfillset(&action.sa\_mask);

action.sa\_flags = SA\_SIGINFO|SA\_RESTART;

sigaction(SIGCHLD, &action, NULL);

signal(SIGTSTP, ctrl\_Z);\*/

signal(SIGCHLD, SIG\_IGN);//neal

signal(SIGTSTP, SIG\_IGN);//neal

//neal 16:54 注册其他信号$

signal(SIGTTOU, SIG\_IGN);//为了成功调用tcsetpgrp()

signal(SIGTTIN, SIG\_IGN);

signal(SIGQUIT, SIG\_IGN);

signal(SIGINT, SIG\_IGN); //Ctrl + C

//neal 16:56 将shell提q到前台$

if(tcsetpgrp(TERM\_FILENO, getpgid(0))<0){

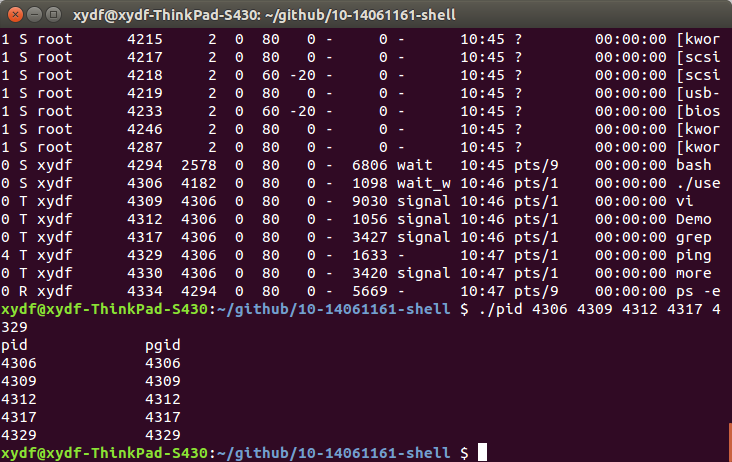
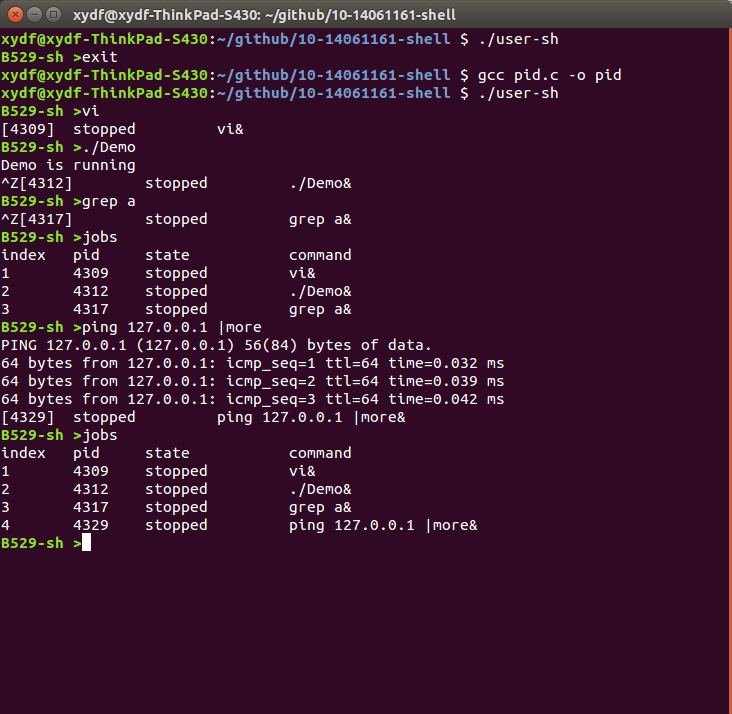
perror("tcsetpgrp():");

exit(EXIT\_FAILURE);

}

}

3.3.6实现进程组、会晤、作业控制



测试程序 pid.c:

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

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

{

printf("pid\t\tpgid\n");

for(int i=1; i<argc; ++i){

pid\_t pid=atoi(argv[i]);

pid\_t pgid=getpgid(pid);

printf("%d\t\t%d\n",(int)pid,(int)pgid);

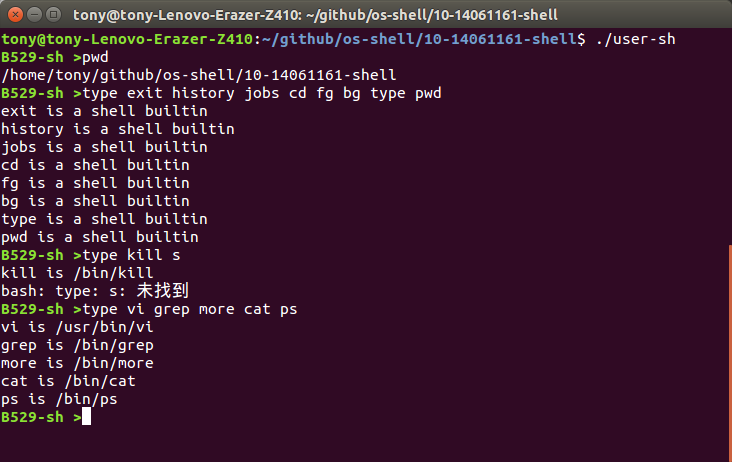
}

return 0;

}

同3.3.4代码部分。

3.3.7实现内部命令pwd、type



execute.c: //新增内部命令pwd、type

void execSimpleCmd(SimpleCmd \*cmd){

int i, pid;

char \*temp;

Job \*now = NULL;

//neal 22:42 确认管道不是内部命令

if(cmd->isPipe)

{

if(

strcmp(cmd->args[0], "exit") == 0 ||

strcmp(cmd->args[0], "history") == 0 ||

strcmp(cmd->args[0], "jobs") == 0 ||

strcmp(cmd->args[0], "cd") == 0 ||

strcmp(cmd->args[0], "fg") == 0 ||

strcmp(cmd->args[0], "bg") == 0 ||

strcmp(cmd->args[0], "type") == 0||//$$

strcmp(cmd->args[0], "pwd") == 0||//$$

strcmp(cmd->pipeEnd->args[0], "exit") == 0 ||

strcmp(cmd->pipeEnd->args[0], "history") == 0 ||

strcmp(cmd->pipeEnd->args[0], "jobs") == 0 ||

strcmp(cmd->pipeEnd->args[0], "cd") == 0 ||

strcmp(cmd->pipeEnd->args[0], "fg") == 0 ||

strcmp(cmd->pipeEnd->args[0], "bg") == 0 ||

strcmp(cmd->pipeEnd->args[0], "type") == 0||//$$

strcmp(cmd->pipeEnd->args[0], "pwd") == 0 )//$$

{

fprintf(stderr,"内部命令不支持管道");

return;

}

}//$

if(strcmp(cmd->args[0], "exit") == 0) { //exit命令

if(head != NULL){

printf("尚有进程正在执行\n");//$$

return;

}

exit(0);

} else if (strcmp(cmd->args[0], "history") == 0) { //history命令

if(history.end == -1){

printf("尚未执行任何命令\n");

return;

}

i = history.start;

do {

printf("%s\n", history.cmds[i]);

i = (i + 1)%HISTORY\_LEN;

} while(i != (history.end + 1)%HISTORY\_LEN);

} else if (strcmp(cmd->args[0], "jobs") == 0) { //jobs命令

if(head == NULL){

printf("尚无任何作业\n");

} else {

printf("index\tpid\tstate\t\tcommand\n");

for(i = 1, now = head; now != NULL; now = now->next, i++){

printf("%d\t%d\t%s\t\t%s\n", i, now->pid, now->state, now->cmd);

}

}

} else if (strcmp(cmd->args[0], "cd") == 0) { //cd命令

temp = cmd->args[1];

if(temp != NULL){

if(chdir(temp) < 0){

printf("cd; %s 错误的文件名或文件夹名！\n", temp);

}

}

} else if (strcmp(cmd->args[0], "fg") == 0) { //fg命令

temp = cmd->args[1];

if(temp != NULL && temp[0] == '%'){

pid = str2Pid(temp, 1, strlen(temp));

if(pid != -1){

fg\_exec(pid);

}

}else{

printf("fg; 参数不合法，正确格式为：fg %%<int>\n");

}

} else if (strcmp(cmd->args[0], "bg") == 0) { //bg命令

temp = cmd->args[1];

if(temp != NULL && temp[0] == '%'){

pid = str2Pid(temp, 1, strlen(temp));

if(pid != -1){

bg\_exec(pid);

}

}

else{

printf("bg; 参数不合法，正确格式为：bg %%<int>\n");

}

}else if(strcmp(cmd->args[0], "type") == 0)//type命令$$^

{

i=1;

while(cmd->args[i]!=NULL)

{

if(strcmp(cmd->args[i], "type")==0||strcmp(cmd->args[i], "exit")==0||

strcmp(cmd->args[i], "history")==0||strcmp(cmd->args[i], "jobs")==0||

strcmp(cmd->args[i], "cd")==0||strcmp(cmd->args[i], "fg")==0||

strcmp(cmd->args[i], "bg")==0||strcmp(cmd->args[i], "pwd")==0)

printf("%s is a shell builtin\n",cmd->args[i]);

else

if(exists(cmd->args[i]))

{

justArgs(cmd->args[i]);

printf("%s is %s\n",cmd->args[i],cmdBuff);

}

else

printf("bash: type: %s: 未找到\n",cmd->args[i]);

i++;

}

}

else if(strcmp(cmd->args[0], "pwd") == 0)//pwd命令

{

char \*dir=get\_current\_dir\_name();

printf("%s\n",dir);

free(dir);

} else{ //外部命令$$$

execOuterCmd(cmd);

}

}

4.问题及解决方法

4.1 源代码中存在的结构性问题

首先，从github上fork来的源代码已经能够实现基本的要求，但其存在诸多编程风格、结构上不合理的地方。比如：

1. 源程序中中大量使用了全局变量，比如fgPid、cmdBuff。虽然这种变量需要在多个函数中使用，但其使用的范围仅仅局限于几个特定的函数。为了方便而在其他函数中耗费大量精力来维护变量并不值得。同时，由于这种全局变量会在不经意的函数中进行修改，所以这种用法并不值得称赞。

2. Jobs列表的维护有些反常理。程序中用来存放作业的链表Jobs并不会在任务开始时添加，而会在任务暂停(Stop)时添加到列表中。这种做法虽然能够大致实现既定的功能，但逻辑性差、严重影响了程序的可维护性。

3. 进程组的问题。真正的Shell要实现前台与后台的区别，必须有设置进程组的步骤。否则，“后台”进程也会接收到终端发出的^Z、^C等命令。

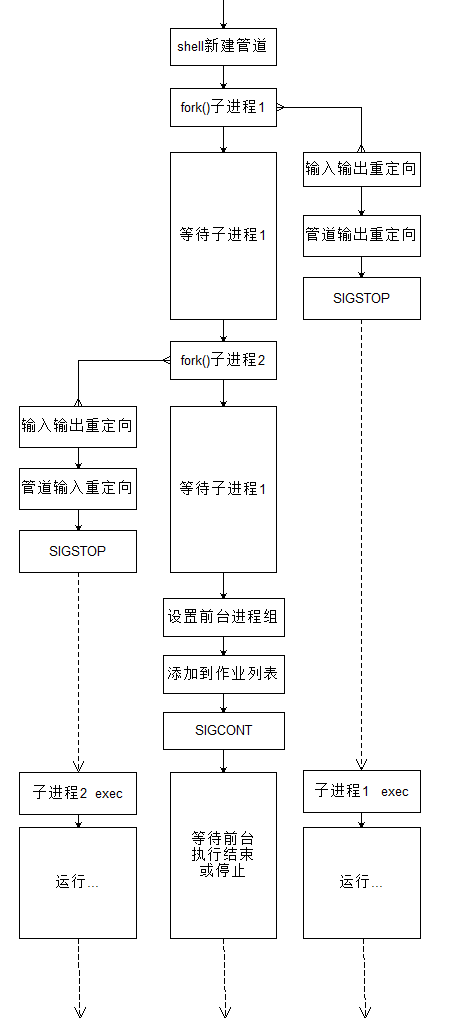
4. 进程间通信的混乱。在程序中，有大量的信号处理部分，比如在新建后台程序时需要多次通信和自定义信号SIGUSR1，甚至移除任务的rmJob也被作为信号处理函数。过多的信号导致了过多的分支，使得程序难以理解；分支也导致程序部分并行，使得程序行为不可控。

综上，要想对shell继续编写，需要进行大刀阔斧的改变。

对程序结构性的修改表现在以下几点上：

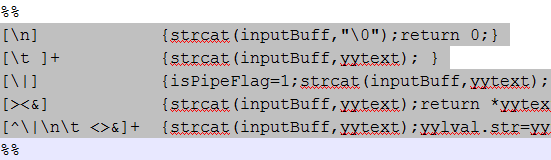
|  |  |  |
| --- | --- | --- |
|  | 修改前 | 修改后 |
| 命令执行机制 | 在当前进程组中运行  使用了SIGCLD信号、goon全局变量、setGoon函数等机制 | 将一行命令创建的一个或多个进程放入到一个进程组中，必要时将其设为前台进程组  执行addJob函数，将作业添加到Jobs列表中 |
| 作业停止运行机制 | 利用shell的信号处理函数ctrl\_Z捕获信号、向前台进程发出SIGSTOP信号  添加作业信息 | ^Z和^C的响应由前台进程组完成，shell负责等待前台进程组是否全部停止运行，并取回shell的终端控制权  这种方式保证了类似vi这种程序可以自行处理^Z和^C信号 |
| 作业继续运行机制 | 发送SIGCONT信号给进程组 | 发送SIGCONT信号前先加入进程组的控制 |
| 作业结束机制 | 根据SIGCHLD信号判断子进程是否退出  使得有可能在操作Jobs列表时被中断、进入rmJob函数，发生不可预料的后果  需要多次设置ignore全局变量 | 在每次打印命令提示符前用kill函数查询子进程是否退出 |

表中命令执行机制的改变尤其值得一提。原代码中在执行一个后台进程的过程中，为了保证进程间的同步，就使用了SIGCLD信号、goon全局变量、setGoon函数等机制，然而仍然难以理解、控制和扩展（比如管道）。我们想到了一种简单自然的方法来实现这个过程，同时满足了通道的要求——当子进程完成重定向和管道相关的工作后，直接发送SIGCHLD信号使自己停止，等待shell设置其进程组等后续过程。整个外部命令执行过程如图：

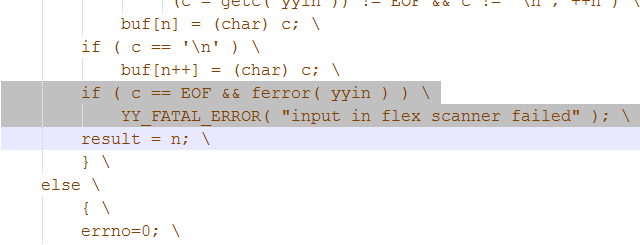


4.2 lex的加入和EOF的问题

原代码中进行词法分析的函数yylex()，是自己编写的，不仅难以理解，而且不利于扩展。在新的程序中，使用lex描述词法，并用lex工具自动生成词法分析函数。这样会使描述规则的部分简洁易懂、利于扩展。



从lex生成的代码中，可以看出，lex默认不支持对EOF的处理。



对此，新程序在读取输入的过程中即对每个字符进行检测，并使用ungetc()函数返回到标准输入中供lex使用，如果是EOF则将结束本次输入。

4.3对堆内存的不规范使用

在原代码中，有大量对堆内存使用不规范的案例，造成调用malloc()的过程使shell崩溃的奇怪bug。新的程序中规范了堆内存的使用。

1. 在打印命令提示符的时候，使用了get\_current\_dir\_name()函数。该函数申请一块内存供存放字符串，但原来的代码中没有free的过程。在新的代码中，调用该函数后

2. 表示命令的结构体cmd在原来的程序中尝试进行释放，但没有释放完全。新的代码中增加了releaseCmd()函数，专门负责将cmd完全释放。

3. 正如前文所述，原来的代码中对Job的不符合逻辑的用法，以及信号处理函数内对Job列表的强行修改，都会造成内存隐患。新的代码使用新的结构避免了这些用法。

在新的程序中，没有出现“连续两次重定向到文件shell会崩溃”的bug。

4.4 进程组设置过程中遇到的问题

设置进程的进程组的函数setpgid()、设置前台进程组的函数tcsetpgrp()中有许多限制。经过查阅大量资料，发现有下述规则：

1. setpgid() 可以用来设置子进程的进程组，但是要求子进程没有调用exec系列函数。

2. tcsetpgrp() 可以由后台进程组内的进程调用，但是默认情况下会接收到SIGTTOU信号而停止。只有该后台进程通过signal(SIGTTOU, SIG\_IGN);来忽略掉这个信号时，它的进程组才会被设置为前台进程组。

在继续调试过程中，我们发现shell的子进程也具有对SIGINT、SIGTSTP等信号的免疫作用。经过测试得知通过signal信号注册的被忽略的信号，会继承给子进程，除非显示使用SIG\_DFT来恢复默认行为，否则即使调用exec系列函数，信号的处理过程也会保留下来。

4.5有关malloc()产生的问题

当我们利用malloc()申请空间的时候需要注意：1.将指针所指区域free后，再次malloc()可能申请到相同地址的空间；2.malloc()所得空间并没有初始化。这两个问题导致我们的user-sh在执行第二个命令的时候，传入的参数错误（误将上一个命令残留的参数读入）。

对此我们给出以下两个解决方案：

1.利用malloc()申请空间后，手动遍历并清空（利用for循环将空间内数据清空）

2.利用calloc()代替malloc()，其主要区别是：calloc()在动态分配完内存后，自动初始化该内存空间为零，而[malloc](http://baike.baidu.com/view/736228.htm" \t "http://baike.baidu.com/_blank)()不初始化，里边数据是随机的垃圾数据。

我们在user-sh中将多处malloc()的空间清空，并将不用的指针(例如\*cmd)及时释放，解决的问题有：

1. user-sh在执行第二个命令的时候，传入的参数错误，导致执行结果错误
2. 在一行命令中输入大量数据，导致程序错误执行。

5.收获和感想

经过这次实验，我们熟悉了Linux操作系统下的基本编程环境和gcc、gbd、YACC等开发工具的使用，了解了shell程序的实现机制、shell与系统内核的相互关系，学习了Linux系统调用的使用方法和相关函数的功能及用途。通过开发该shell程序，我们对前后台命令、进程管理、进程组、管道和进程间通信等概念及其编程实现有较为深刻的理解。在实验中我们对课上所学的相关理论知识的有了更深层次的理解，在实践中更好地理解了难点和疑点。同时，在编写程序的过程中，提高了我们分析问题、发现并解决问题的能力以及团队成员间的沟通协作能力。

当然，我们在实际实验过程中也遇到了很多困难。如：程序中的bug难以定位、通配符的具体实现等。我们通过上网查询相关资料、与同学们讨论、请教老师和助教等方式逐一解决了部分问题。对于其他问题，我们通过编程实践，不断尝试各种情况，反复修改和调试程序，分析程序的运行结果，这些问题最终得到了有效地解决。

纸上得来终觉浅，绝知此事要躬行。总的来说，这次实验虽然遇到了不少困难，但我们在克服困难的过程中学到了很多相关知识，加深了所学知识的理解，很有收获。