**操作系统课程设计实验报告**

——实验一：shell实验

负责人姓名：李奕成

学号：14061157

日期：2016.3.25

**小组成员**

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 姓名 | 学号 | 实验分工 |
| 1 | 李奕成 | 14061157 | 基本要求、ctrl+c、yacc&lex、测试 |
| 2 | 姬索肇 | 14061163 | 管道、通配符、yacc&lex、测试 |
| 3 | 何林鑫 | 14061169 | 通配符、测试 |
| 4 | 邓越 | 14061151 | 通配符、测试 |

目录

[1.实验目的 4](#_Toc446001831)

[2.需求说明 4](#_Toc446001832)

[2.1基本要求 4](#_Toc446001833)

[2.2 提高要求 4](#_Toc446001834)

[2.3 完成情况 4](#_Toc446001835)

[3.设计说明 5](#_Toc446001836)

[3.1 程序流程图 5](#_Toc446001837)

[3.2基本要求实现说明 5](#_Toc446001838)

[3.3 提高要求实现说明 5](#_Toc446001839)

[4.收获和感想 5](#_Toc446001840)

# 1.实验目的

1.完成基本要求，程序能够正常运行、支持fg/bg等内部指令、支持带参数的外部命令、支持IO重定向、支持作业状态的控制和变化。

2.完成提高要求，根据提高要求更改yacc语法分析部分、添加通配符“\*”“？”、支持管道、支持ctrl+c杀死前台进程。

3.利用本次实验，初步了解linux的一些运行机制。主要是进程的通信机制、信号量控制方法。Yacc、lex工具的使用方法。

# 2.需求说明

## 2.1基本要求

程序能够正常运行。

能够执行fg、bg、cd、history、exit。

能够执行外部程序命令，命令可以带参数。

使用I/O重定向。

支持前后台作业，提供作业控制功能，包括打印作业的清单，改变当前运行的前台/后台状态，以及控制作业的挂起、中止和继续执行。

## 2.2 提高要求

尝试对yacc语法分析的文法进行进一步的修改与完善。

尝试在linux下将lex和yacc结合起来使用进行词法分析、语法分析。

对其他常用的内部命令进行实现，并可以尝试对通配符的支持与实现。

实现对管道的支持。

考虑并实现组合键ctrl+c命令。

## 2.3 完成情况

【简述实验完成过程】。完成了以下功能：

1.完成了基本要求。

2.完成了对扩充的指令修改对应的yacc、lex代码部分。

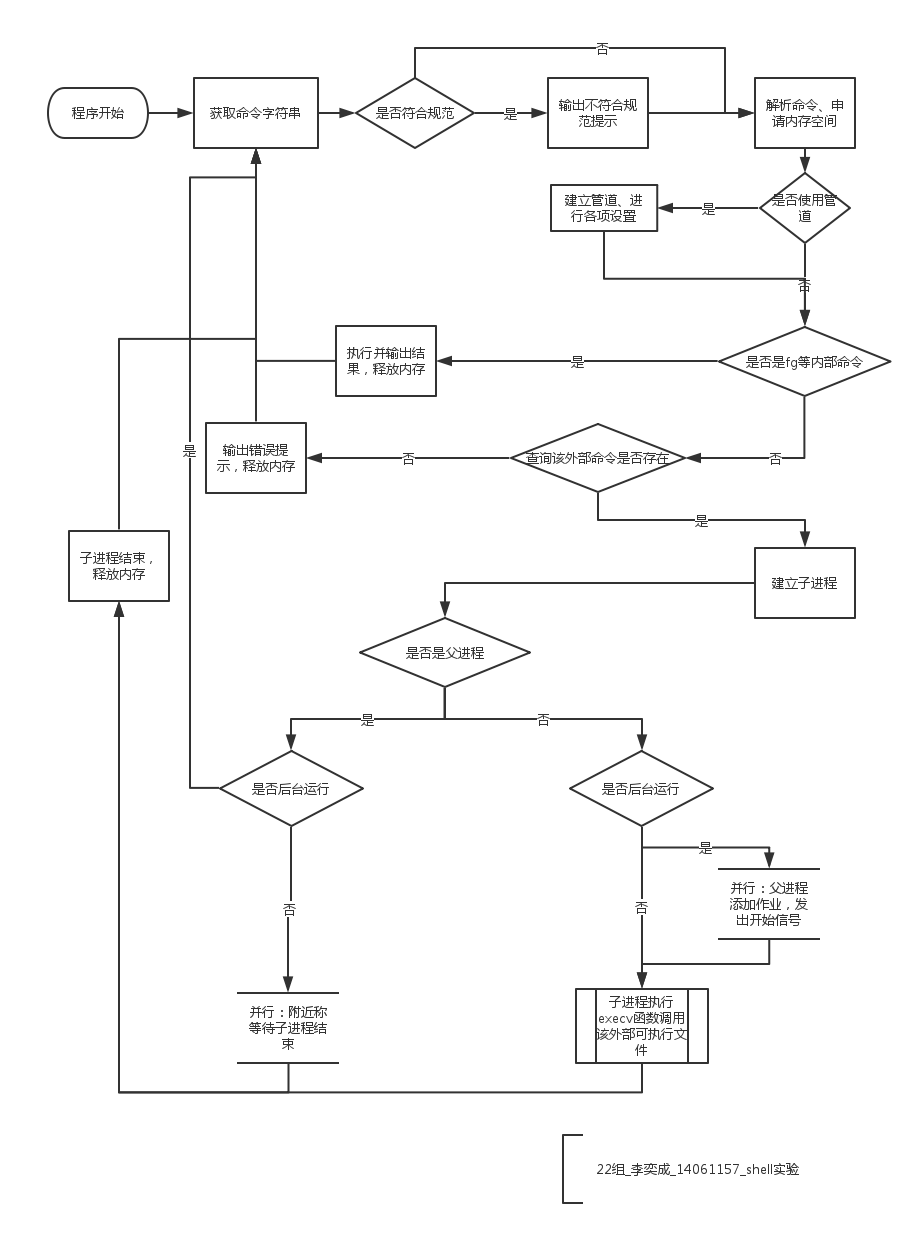
3.支持带有管道的指令。

4.支持查询文件相关类命令的通配符“\*”“？”的支持。其中“\*”可以出现在路径中，但不会代替“/”。

5.支持Ctrl+c杀死当前的前台进程。

# 3.设计说明

## 3.1 程序流程图



## 3.2基本要求实现说明

配以详细的代码和实现步骤截图。

【注：由于最终调整，代码可能有细微差别，但是思路是一样的】

修复bug如下：

1.键入形如”./Demo&”的后台命令，后台命令不会执行。为了解决这个问题，修改了父进程对于后台命令的处理代码，具体代码和运行截图如下。

子进程内：

if(cmd->isBack){ //若是后台运行命令，等待父进程增加作业

signal(SIGUSR1, setGoon); //收到信号，setGoon函数将goon置1，以跳出下面的循环

while(goon == 0) ; //等待父进程SIGUSR1信号，表示作业已加到链表中

goon = 0; //置0，为下一命令做准备

kill(getppid(), SIGUSR1);//子进程设置已完成，即将开始运行

}

signal(SIGTTIN,SIG\_DFL);

signal(SIGTTOU,SIG\_DFL);

justArgs(cmd->args[0]);

setpgid(0,0);//设定一个子进程组,防止被前台挂起命令挂起

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

printf("execv failed!\n");

\_exit(0);//放弃使用return

}

父进程内：

if(cmd ->isBack){ //后台命令

fgPid = 0; //pid置0，为下一命令做准备

addJob(pid); //增加新的作业

sleep(1);//稍待子进程运行

kill(pid, SIGUSR1); //子进程发信号，表示作业已加入

signal(SIGUSR1, setGoon); //收到信号，setGoon函数将goon置1，以跳出下面的循环

while(goon == 0); //等待子进程SIGUSR1信号，表示已开始

goon = 0; //置0，为下一命令做准备

}else{ //非后台命令

fgPid = pid;

signal(SIGTTOU,SIG\_IGN);

ingnore=1;//前台禁止删除作业

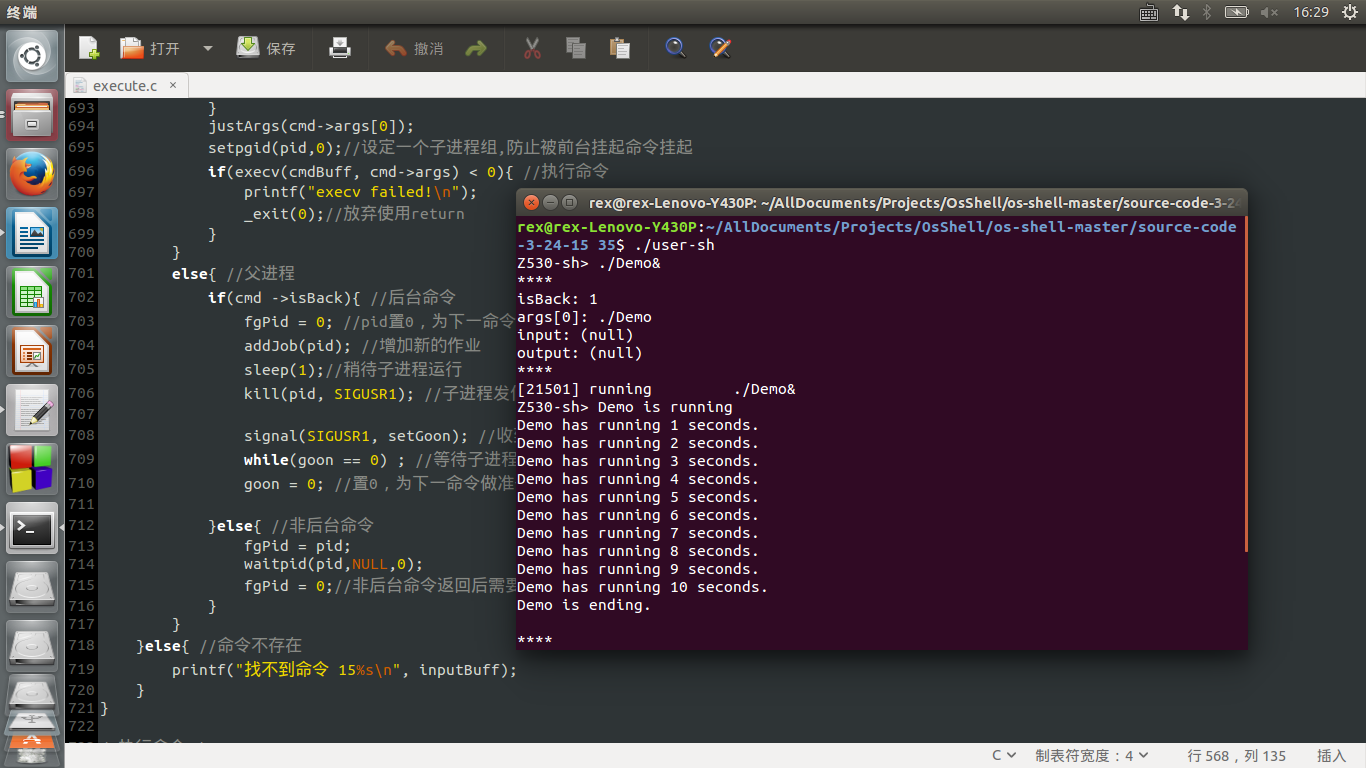
tcsetpgrp(0,pid);

waitpid(pid,&status,0);

tcsetpgrp(0,getpid());

fgPid = 0;//非后台命令返回后需要将前台置为零

}



2.前台命令运行过程中（如Demo运行的10s期间），使用组合键不会使进程终止。为了解决这个问题，最开始的设计修改了Ctrl\_C处理函数，但由于实际运行中的前后台 进程组问题，实际起作用的是rmjob函数。具体代码和运行截图如下。

/\*移除一个作业\*/

void rmJob(int sig, siginfo\_t \*sip, void\* noused){

pid\_t pid;

Job \*now = NULL, \*last = NULL;

if(MY\_DEBUG) printf("in rmjobs ingnore -->>%d\n",ingnore);

pid = sip->si\_pid;

now = head;

while(now != NULL && now->pid < pid){

last = now;

now = now->next;

}

if(now == NULL){ //作业不存在，必为初次前台进程暂停或死亡

sig\_chld();

now=addJob(pid);//强制添加

int res=kill(now->pid,0);

//printf("check res-->%d pid-->%d\n",res,pid);

if(res==0)//被暂停的

{

strcpy(now->state, STOPPED);

now->cmd[strlen(now->cmd)] = '&';

now->cmd[strlen(now->cmd) + 1] = '\0';

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

ingnore=0;

return;

}

else//死亡了，删去jobs

{

if(MY\_DEBUG)printf("fg dead!\n");

//printf("nowpid:%d headpid:%d\n",now->pid,head->pid);

if(now == head){

head = now->next;

}else{

last->next = now->next;

}

//free(now);

ingnore=0;

return;

}

}

//作业存在，必为挂起过的前台进程或后台进程 暂停或死亡

sig\_chld();

int res=kill(now->pid,0);

if(res==0)//被暂停的

{

strcpy(now->state, STOPPED);

now->cmd[strlen(now->cmd)] = '&';

now->cmd[strlen(now->cmd) + 1] = '\0';

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

ingnore=0;

return;

}

if(MY\_DEBUG) printf("after fg process\n");

//开始移除该作业

if(now == head){

head = now->next;

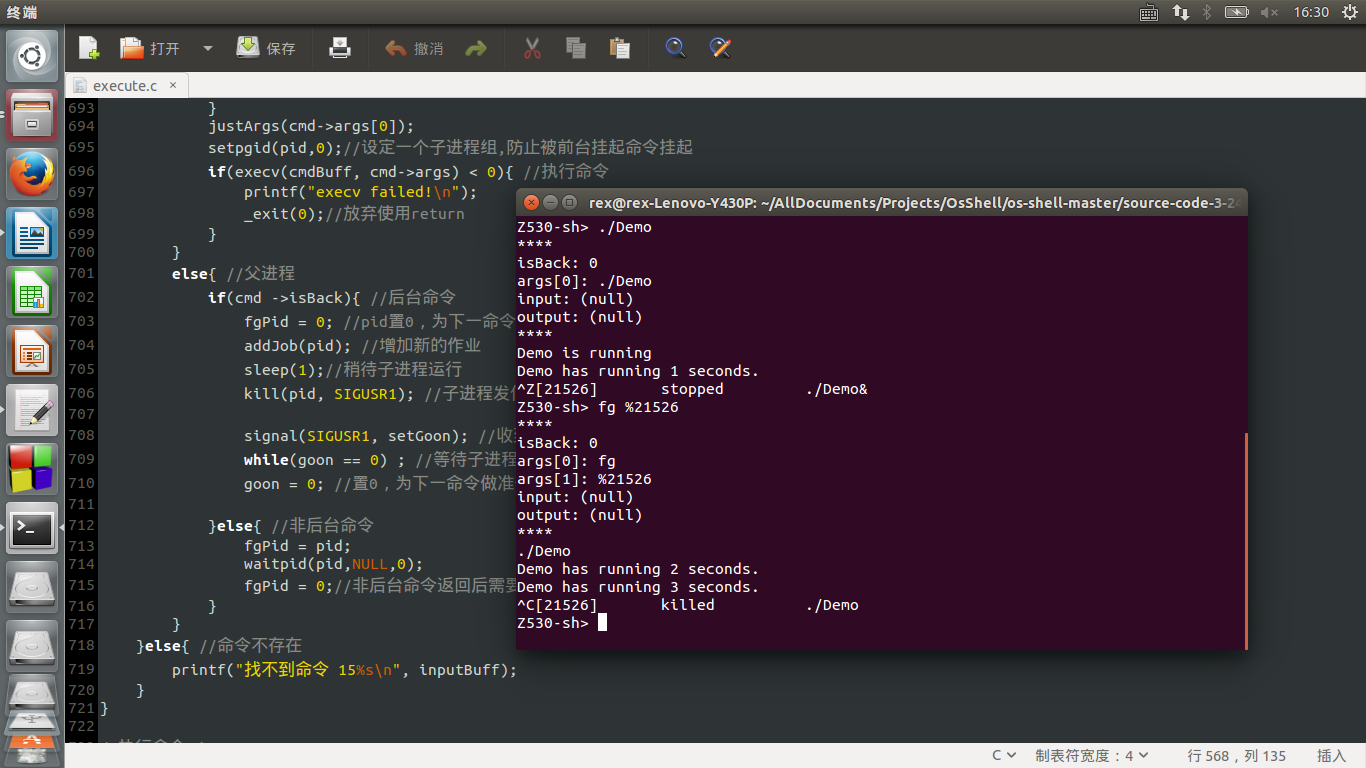
}else{

last->next = now->next;

}

free(now);

}



3.后台运行程序运行时，在终端键入”exit”命令不会使后台运行的程序退出。修复了这个问题，具体代码和运行截图如下。

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

kill(now->pid,SIGKILL);

}

4.前台进程挂起之后不能自动回到前台状态。

fgPid = 0;//非后台命令返回后需要将前台置为零

5.fg指令运行前台指令，父进程没有等待子进程。

waitpid(fgPid, NULL, 0);//父进程等待前台进程的运行，成功结束返回fgPid

tcsetpgrp(0,getpid());

6.后台进程结束后成为僵尸进程。

void sig\_chld() {

pid\_t pid;

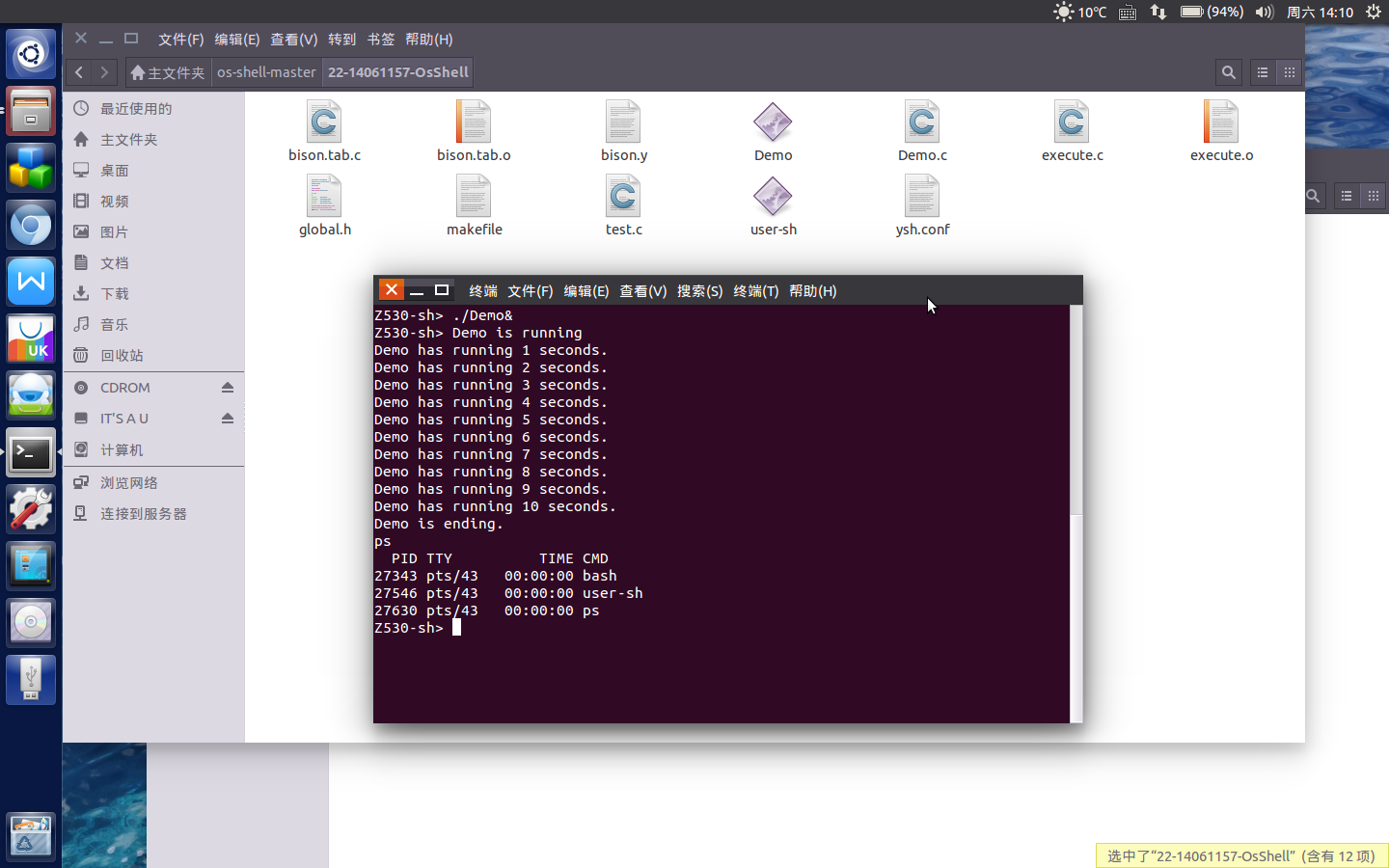
int stat;

pid = waitpid(-1,&stat,WNOHANG);

return;

}

在rmjob内调用

sig\_chld();

7.前后台进程组切换，主要问题集中在了如何切换前后台进程组。使用了setpgrp()函数和tcsetpgrp()函数。注意父进程为了不被SIGTTIN信号和SIGTTOU信号阻塞，需要为子进程设置这两个信号的忽略。而子进程必须开启这两个信号的默认处理。

以fg命令处理函数为例：

tcsetpgrp(0,now->pid);

kill(now->pid, SIGCONT); //向对象作业发送SIGCONT信号，使其运行

sleep(1);//等待一秒，保证子进程收到开始启动信号（有没有更好的改法）

signal(SIGTTOU,SIG\_IGN);

waitpid(fgPid, NULL, 0);//父进程等待前台进程的运行，成功结束返回fgPid

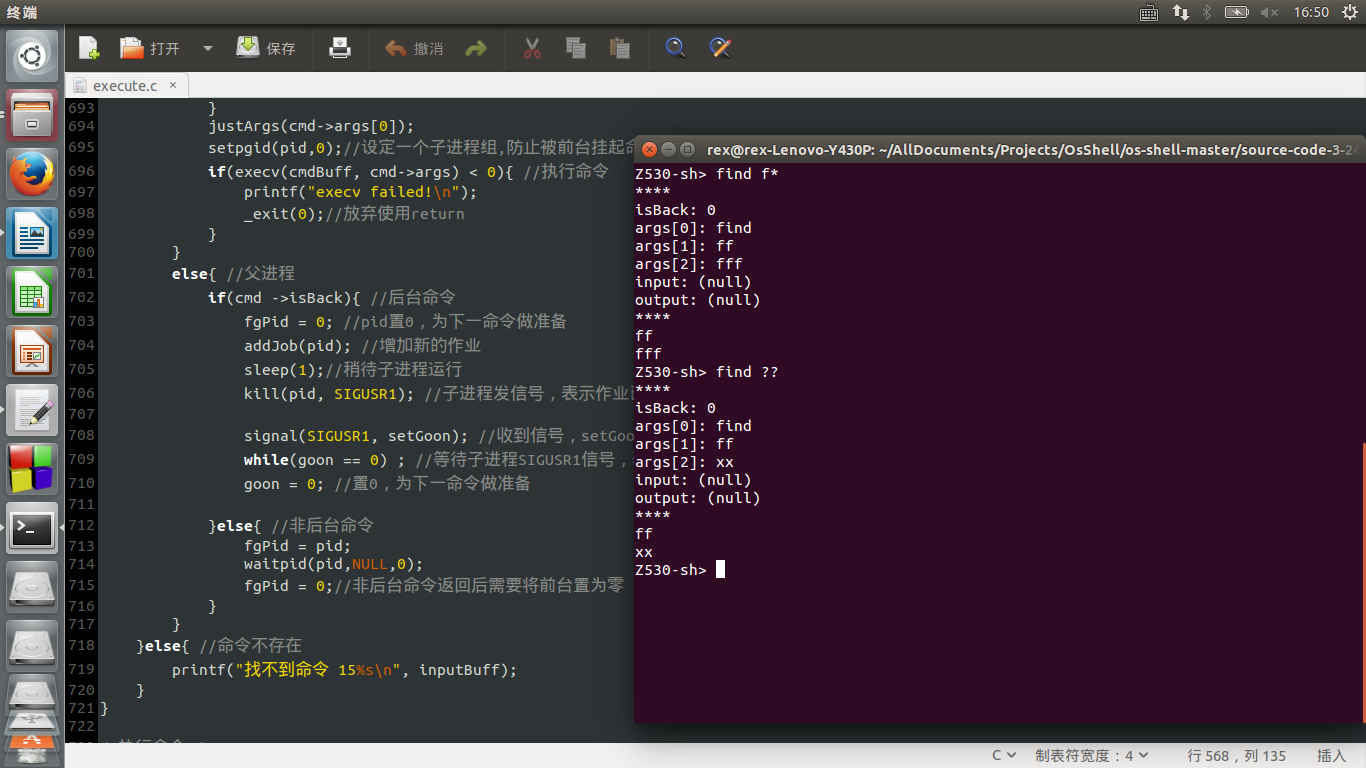
tcsetpgrp(0,getpid());

## 3.3 提高要求实现说明

配以详细的代码和实现步骤截图。

1.实现了\*和？通配符，\*能够代替任意长度的任意字符（但是不会代替目录分隔符/，bash的通配符也是这样设定的，？可以代替任意一个字符）与bash的运行结果一致。具体代码和运行截图如下。

代码量过大，不适合粘贴。在execute.c函数中为disearch()、match()、match\_imp()、strsort()、revstr()五个函数，另外修改了handleSimpleCmdStr()函数来进行分析。总体思路是分析指令之后，用系统库函数获取当前目录下的全部文件名，与通配符字符串进行匹配，并对结果进行排序然后在进行进一步处理。



2.实现了管道。以截图中的命令为例，cat ff|head -5|tail -5，会将cat ff的输出作为head -5的输入，将head -5的输出作为tail -5的输入，输出结果是ff文件的前五行。具体代码和运行截图如下。

两个同学分别实现了中转文件的方法和建立进程间管道通信的方法。测试过程中由于前者比较稳定，只采用了前种办法。总体思路是分析命令字符串，建立中间文件，进行输入输出重定向，循环执行每一条指令。

if(strstr(inputBuff,myEOF)!=NULL){//去掉EOF

int i;

int len=strlen(inputBuff);

char newstr[1000]={0};

int newstrcount=0;

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

if(inputBuff[i]!=EOF)

newstr[newstrcount++]=inputBuff[i];

strcpy(inputBuff,newstr);

}

if(strstr(inputBuff,"|")!=NULL){//当输入中包含管道符|

strcpy(inputBuffCopy,inputBuff);

strcpy(meminputBuff,inputBuff);

part=strtok(inputBuffCopy,delim);

while(part!=NULL){

newpart=(char\*)malloc(sizeof(char)\*strlen(inputBuff));

strcpy(newpart,part);

part=strtok(NULL,delim);//读取下一个命令

strcpy(inputBuff,newpart);

SimpleCmd \*cmd = handleSimpleCmdStr(0, strlen(inputBuff));

if(cmdcount==0){

cmd->output=(char\*)malloc(sizeof(char)\*6);

strcpy(cmd->output,"temp1");

}

else if(part!=NULL){

cmd->input=(char\*)malloc(sizeof(char)\*6);

cmd->output=(char\*)malloc(sizeof(char)\*6);

if(cmdcount%2==1){

strcpy(cmd->input,"temp1");

strcpy(cmd->output,"temp2");

}

else{

strcpy(cmd->input,"temp2");

strcpy(cmd->output,"temp1");

}

}

else{

cmd->input=(char\*)malloc(sizeof(char)\*6);

if(cmdcount%2==1){

strcpy(cmd->input,"temp1");

}

else{

strcpy(cmd->input,"temp2");

}

}

cmdcount++;

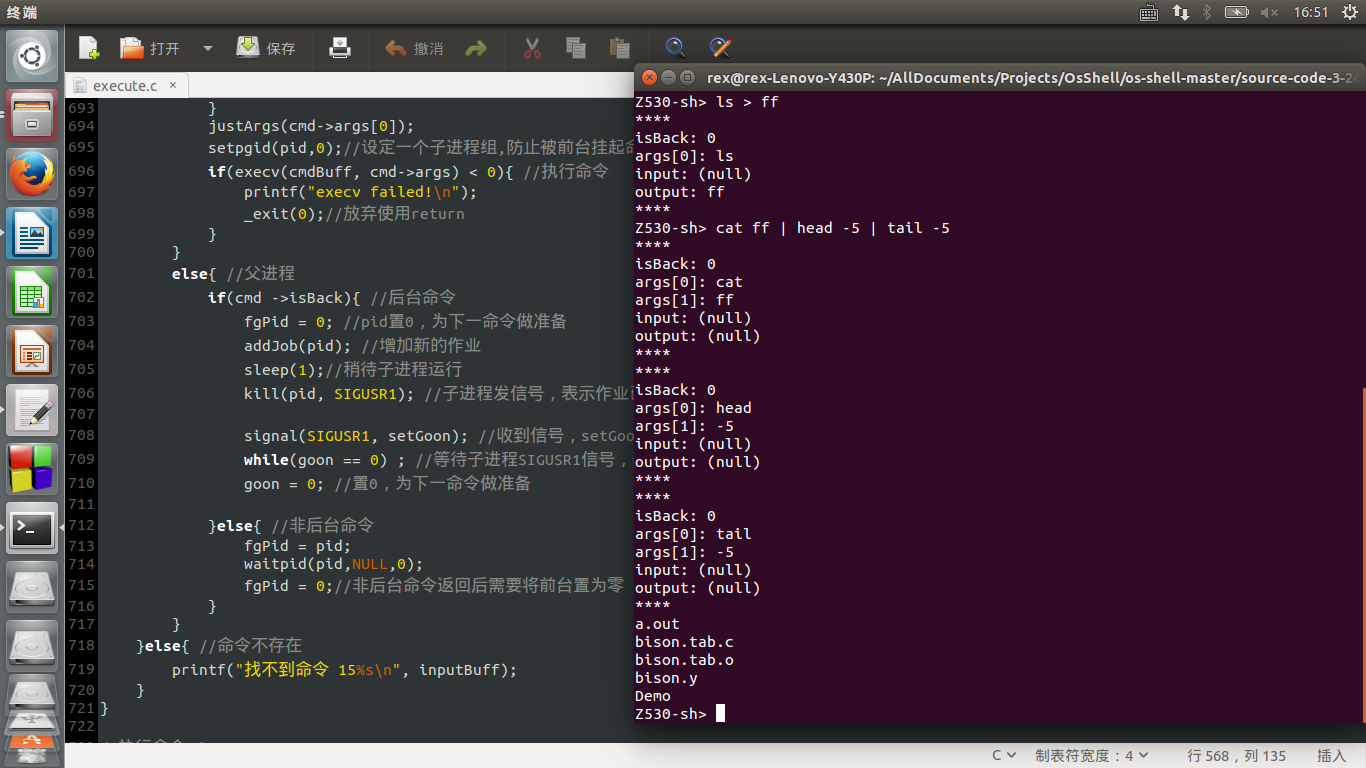
execSimpleCmd(cmd);//执行命令

strcpy(inputBuff,meminputBuff);//将原来的inputBuff还原，以便添加历史记录

free(newpart);

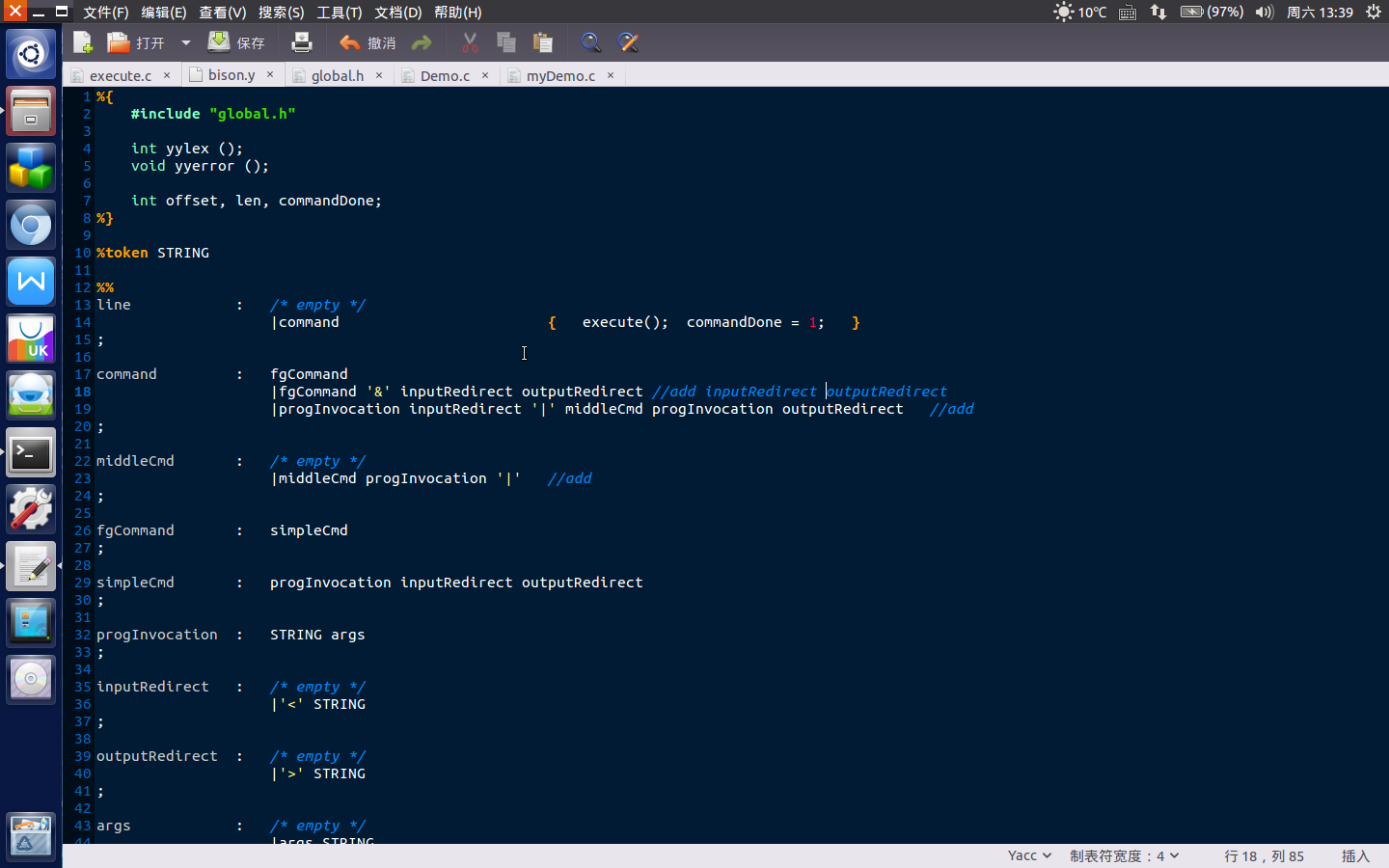
}

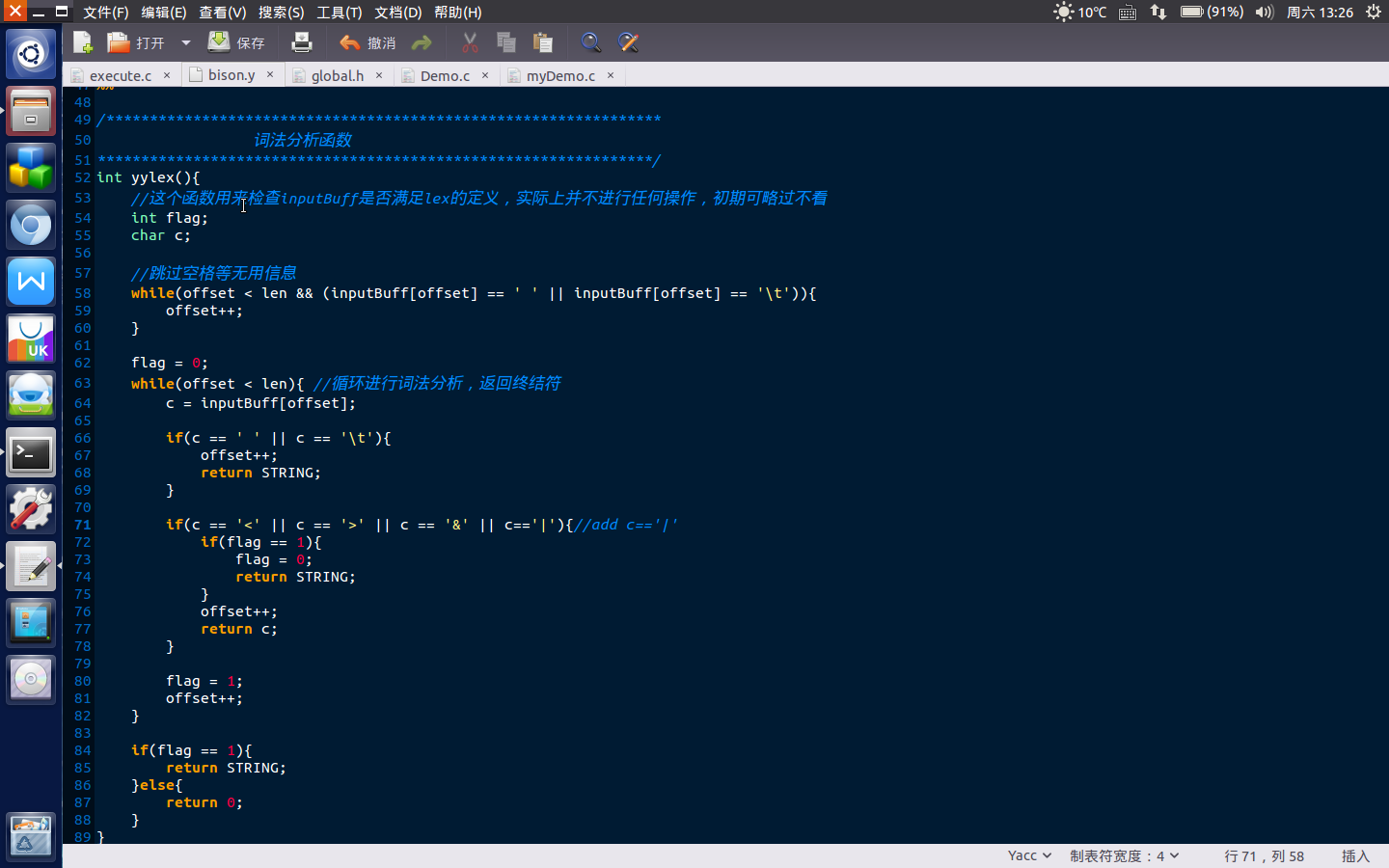
}



3.完善lex和yacc的文法，由于加入了管道符’|’，所以lex和yacc的文法需要进行完善。首先是在yylex函数中加入返回’|’的功能，使得’|’能够称为文法中单独出现的符号，然后加入了一种middleCmd标识，识别管道命令的中间几条命令，由于中间几条命令的输入和输出分别是上一条命令的输出和下一条命令的输入，所以不应该被重定向，middleCmd能够识别这一点。然后在command中加入了一条progInvocation inputRedirect ‘|’ middleCmd progInvocation outputRedirect，即第一条命令的输入可以重定向，最后一条命令的输出可以重定向，从而识别出管道命令。

注意到”./Demo&>ff”这样的后台命令（后台命令可以输出，但不可以输入）不会被识别，所以修改command 中的 fg ‘&’ 为fg ‘&’ inputRedirect outputRedirect。具体代码和运行截图如下。





# 4.收获和感想

通过这次实验，我们完整的阅读了source-code的代码，对于Shell程序的设计有了更加深刻的理解，同时通过自己修改程序中的Bug，提升了自己的调试和编程能力。在完成提高要求的过程中，深入地yacc&lex的文法和linux的管道，通过自己的努力实现了通配符，可以说是获益匪浅。

基础部分，几个同学一起合作，互相交流自己对于源码的写法的理解以及各种新的想法。在调试几个基础bug的过程中，我们越来越明白linux的进程生命过程的各个阶段的特点，以及execv这个比较重要的系统调用函数的特性。另外通过对各种信号量的实验，了解到了signal()的用法，以及sigaction的优点。而在调试进程组这个事情中，我们更加深入的理解到了SIGCHLD和SIGTTIN以及SIGTTOU这三个信号量的特性。丰富了对SIG\_IGN/SIG\_DFL这两种处理方法的理解。这个过程真的很有启发性，特别有趣。

而在提高部分，大家也是积极的讨论了各种可能的解决方法。进行了各种模型试验，最后比较快速、安全的完成了提高要求。在学习管道的使用方法的过程中，大家更是深入的理解了进程间通信的各种需要注意的情况，以及管道描述符的使用方法，标准输入输出的绑定办法。当程序成功运行的时候一种愉悦感油然而生。

今后我们小组会更加努力，争取写出更好的程序。