**操作系统课程设计实验报告**

——实验一：shell实验

负责人姓名：李子涵

学号：14061145

日期：2016.3.26

**小组成员**

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 姓名 | 学号 | 实验分工 |
| 1 | 李子涵 | 14061145 | 实验一 |
| 2 | 许楷舟 | 14061141 | 实验二 |
| 3 | 陈斌 | 14061144 | 实验三 |
| 4 | 林之力 | 14061133 | 实验四 |

备注：后三次负责人可能会有变动。

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.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

6.掌握进程控制和进程间通信的方法

# 2.需求说明

## 2.1基本要求

1.程序能够正常运行

2.能够执行fg、bg、cd、history、exit等内部命令

3.能够执行外部程序命令，命令可以带参数

4.使用I/O重定向

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

## 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，并可以尝试考虑对通配符的支持与实现

4.实现对管道的支持

5.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.程序能够正常运行

2.执行fg、bg、cd、history、exit等内部命令

3.执行外部程序命令，命令可以带参数

4.使用I/O重定向

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

6.对YACC语法分析的文法进行进一步的修改与完善

7.在Linux下将Lex和YACC结合起来使用进行词法和语法分析

8.支持其他常用的内部命令

9.对管道的支持

10.组合键ctrl+c命令

# 3.设计说明

## 3.1 程序流程图

从lex读入输入>>>>bison.y匹配命令>>>>调用ececute.c中的execute()函数

## 3.2基本要求实现说明

**1.解决中文的乱码问题**

先使用系统自带的shell验证exit,history,jobs,fg,bg,Ctrl+Z,Ctrl+C的功能和现象

再使用自己编写的shell，即user-sh验证上述命令的功能，首先exit和history看似已经完成，没有明显的问题。这两个比较简单好理解。接下来使用fg,bg,jobs,再进一步了解一下这三个命令的功能，man.linuxde.net，这个网上有对linux命令的一些比书上更详细一点的讲解。看完之后在user-sh里进行验证，借助Demo程序，在使用fg，bg时发现编码错误。下面尝试解决编码问题。先安装enca软件包来查看编码，输入apt-get install enca，安装成功后检查execute.c的编码 方式

$ enca filename

输出

Simplified Chinese National Standard; GB2312

CRLF line terminators

说明是GB2321编码

下面需要将文件编码转换成UTF-8

$ iconv -f GB2312 -t UTF-8 execute.c > newexecute.c

再enca命令查看newexecute.c的编码方式，发现已经成功转化为utf-8编码了

Universal transformation format 8 bits; UTF-8

CRLF line terminators

下面我想编译newexecute.c来代替execute.c发现编译出错了，应该是之前所有文件都是关联execute.c的，突然使用一个文件名确实不妥。

于是输入下面命令

$ iconv -f GB2312 -t UTF-8 execute.c > execute.c

再检查execute.c的编码，发现输出是未识别的编码，这个方法还是行不通，再试别的方法

先把execute.c文件名改成execute1.c

再转码$ iconv -f GB2312 -t UTF-8 execute1.c > execute.c，再检查编码发现成功了

这时已经成功解决编码问题，再使用fg，bg已经可以正常输出中文了。

再检查其他文件的编码，发现bison.y的编码是GB2312，按照刚才的方法再把bison.y转码成功即可

**2.解决输入Ctrl+Z后无法识别其他命令的问题**

$ ./test

$ Ctrl+Z

$ jobs

index pid state command

1 3985 stopped ./test&

$ bg %3985

$ jobs

找不到命令 15�jobs

解决办法：

修改bison.y的main函数，在while里添加while((c=getchar())==-1);ungetc(c,stdin);

**3.解决三个坑的问题**

坑1：关于kill()函数

调试结论：说明在父进程执行完kill(pid, SIGUSR1)命令后子进程被杀死；SIGUSR1、SIGUSR2等信号量默认会导致进程退出；保证子进程在父进程Kill函数执行前运行。

解决办法：在“kill(pid, SIGUSR1);”前增加sleep(1);//父进程kill前面睡一秒，等子进程开始运行

坑2：关于wait()函数

现象：运行前台作业，执行ctrl+z命令挂起：正常

执行fg %<int>命令继续运行前台作业：错误（命令提示符在作业运行完之前就打印出来，即父进程没有等待子进程执行完）

调试结论：wait函数要等待子进程收到开始启动进程的信号后运行才能得到正确的结果。

解决办法：在fg的信号代码里的“kill(now->pid, SIGCONT);”后增加sleep(1);

坑3：ctrl+z 命令问题

现象：运行后台作业过程中，键入ctrl+z会导致后台作业停止运行

原因：子进程会继承父进程的信号处理方式，直到子进程调用exec函数；子进程调用exec函数后，exec将父进程中设置为捕捉的信号变为默认处理方式，其余不变

解决方案：使用setpgid函数将子进程分配到独立的进程组，则不会捕获到SIGTSTP信号

关于该问题相关的知识：getpid返回当前进程标识，getppid返回父进程标识。

进程组id = 父进程id，即父进程为组长进程

组长进程标识: 其进程组ID==其进程ID

组长进程可以创建一个进程组，创建该进程组中的进程，然后终止

只要进程组中有一个进程存在，进程组就存在，与组长进程是否终止无关

进程组生存期: 进程组创建到最后一个进程离开(终止或转移到另一个进程组)

一个进程可以为自己或子进程设置进程组ID

setpgid()加入一个现有的进程组或创建一个新进程组

解决过程：

Team7@computer:/home/lizihan/1$ ./Demo

命令执行

\*\*\*\*

isBack: 0

args[0]: ./Demo

input: (null)

output: (null)

\*\*\*\*

执行外部命令

进入执行外部命令

命令存在

即将创建子进程

我是父进程

非后台命令

父进程pid是3107

等待子进程执行

子进程创建成功

子进程即将执行命令

Demo is running

Demo has running 1 seconds.

^Z[3107] stopped ./Demo&

子进程执行命令结束

Team7@computer:/home/lizihan/1$ ps

命令执行

\*\*\*\*

isBack: 0

args[0]: ps

input: (null)

output: (null)

\*\*\*\*

执行外部命令

进入执行外部命令

命令存在

即将创建子进程

我是父进程

非后台命令

子进程创建成功

子进程即将执行命令

父进程pid是3109

等待子进程执行

PID TTY TIME CMD

2769 pts/0 00:00:00 bash

3102 pts/0 00:00:00 user-sh

3107 pts/0 00:00:00 Demo <defunct>

3109 pts/0 00:00:00 ps

子进程执行命令结束

解决办法：在execOuterCmd函数进入子进程时加入setpgid(pid,pid);， 改变子进程的组id为子进程本身。

**4.解决其他bug**

在解决完上述三个坑再进行测试时发现了新的问题，

$ ./test &

如果把一个程序直接放入后台运行，则无法再输入任何命令

解决办法：由于之间调试在代码中加入了很多的printf，将其全部注释掉，这时不会再出现上述问题

至此，基本要求均已实现。

## 3.3 提高要求实现说明

**1.使用YACC和LEX进行文法分析**

**一、程序简单介绍**

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中的词汇模式的程序。 这些词汇模式在一种特殊的句子结构中定义，一种匹配的常规表达式可能会包含相关的动作，包括返回一个标记。 当 Lex匹配到一个模式时，Lex 就返回一个标记给YACC程序。 另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，Lex 将显示一个错误消息。

程序依此来进行shell命令的识别和执行。

**二、程序实现步骤**

1、**Lex程序的编写**

首先按照命令的格式，在.Lex文件中用表达式定义可能出现的命令模式，使得.Lex程序能够识别正确指令并返回标记。

char [A-Za-z\.]

int [0-9]

inputre [<]

outputre [>]

nothing [' ']

none {nothing}\*

commend((\.\/)?{char}+{none}\*(%(int)+)?&?)|((\.\/)?{char}+{none}\*[><]?{none}\*{char}+)|({char}+{none}\*(\/{char}\*)\*)

args %{int}+

enter [^.]

可见程序首先定义了char、int等基本字符、数字元素，再将其组合为commend命令的模式，使得Lex能够识别正确格式的命令。

%%

{commend} { strcat(inputBuff,yytext);

printf("yes1");

return STRING; }

{args} { strcat(inputBuff,yytext);

printf("yes2");

return ARGS; }

{enter} { printf("yes3");

return ENTER; }

{inputre} { strcat(inputBuff,yytext);

printf("yes5");

return INPUTRE; }

{outputre} { strcat(inputBuff,yytext);

printf("yes6");

return OUTPUTRE; }

%%

在Lex程序匹配段，程序将识别到正确commend的标记STRING返回给bison.y程序

**2、bison.y程序的修改**

在原来的bison.y程序中，用c语言模拟了yylex()函数的实现过程。在新的程序中，由于使用了lex来匹配命令格式，所以需要将原来的yylex()函数注释掉。

原程序中只有STRING标记，新增了ARGS、ENTER、INPUTRE、OUTPUTRE标记，ARGS返回指令参数，INPUTRE、OUTPUTRE用来返回输入输出重定向标记，ENTER用来结束输入，后面将对ENTER的作用进行详细说明。

**3、Lex程序与bison.y的协同工作**

由于程序最后需要将所有文件编译为一个user-sh可执行文件，所以要将global.h文件包含在bison.y和Lex文件中，同时Lex文件中还需包含bison.tab.h文件。

Make file文件也需要重新编写。

user-sh : bison.tab.o execute.o bison.tab.h

cc -o user-sh bison.tab.o execute.o lex.yy.o

bison.tab.o : bison.tab.c global.h bison.tab.h lex.yy.c

cc -c bison.tab.c lex.yy.c

execute.o : execute.c global.h

cc -c execute.c

bison.tab.c:bison.y global.h

bison -d bison.y

lex.yy.c:bison.tab.h name.lex global.h

lex name.lex

编译流程为：

（1）：使用lex name.lex编译生成lex.yy.c文件

（2）：使用bison -d bison.y编译生成bison.tab.c文件

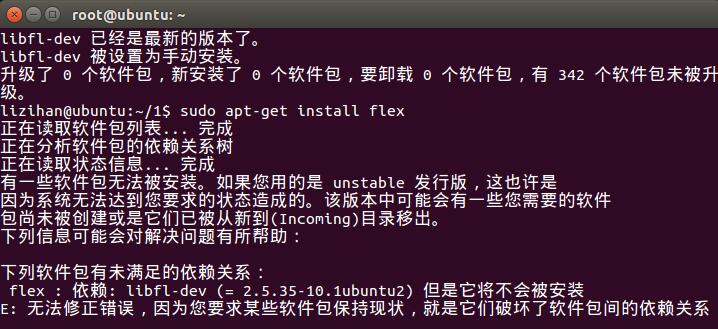
（3）：使用cc -c execute.c编译生成execute.o文件

（4）：使用cc -c bison.tab.c lex.yy.c将bison文件和Lex文件联合编译，生成bison.tab.o文件

（5）使用cc -o user-sh bison.tab.o execute.o lex.yy.o将前面生成的-o文件编译成user-sh可执行文件。

**三、实验遇到的问题及解决方法**

**1、flex包无法安装问题**



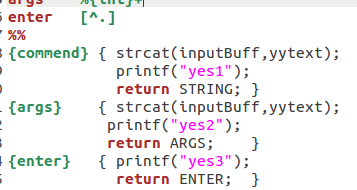
查找资料发现是本机UBUNTU更新源太过老旧，可以在网上找到新的ubuntu更新源地址，将其替换到etc/apt/sources.list文件中即可。

**2、无法判定输入结束问题**

在写完lex进行系统测试的时候我发现，我的匹配模式明明是正确的，程序却总是无法识别正确指令并执行，在将所有的符号捕获调试才发现，Lex将我输入的最后一个回车符\n当作了输入的一部分，使得输入结束之后Lex依然处于等待输入状态而不是结束输入。这时指令执行错误。

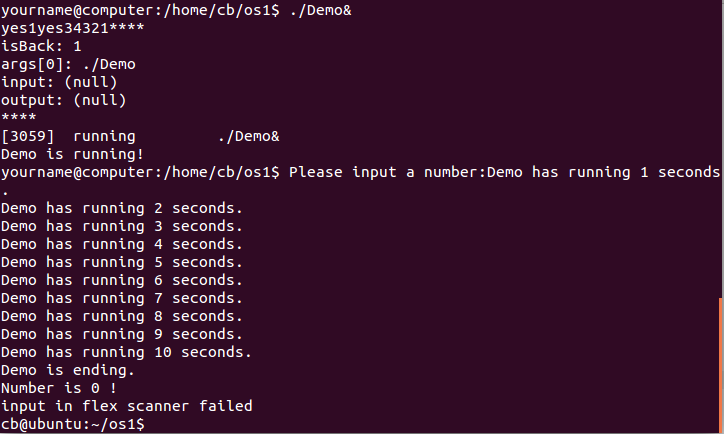
查阅资料可知，lex只有在匹配错误的时候才会返回。此时可以在Lex文件中设置[^.]模式，此表达式的意思是匹配\n，建立ENTER标识，但是在bison.y文件中不捕获ENTER,这时\n的输入使得匹配失败，即flex程序运行结束，指令正常执行。





**3、运行后台命令时flex出错**

执行后台命令如./Demo时程序运行之后会显示如图的错误，



并且程序会回到本地shell。百思不得其解。

最后发现是因为后台运行程序时会返回EOF,而lex程序会将EOF当作错误标识，所以在bison.y中，在yyparse()函数前面加上

while((c=getchar())==-1);

ungetc(c,stdin);

即如果读到EOF，则将其退回输入流。依此达到过滤EOF的目的。

**2.管道的实现**

1. **管道的相关知识**

管道是一种比较复杂的进程间通讯，它可以将一个程序的输出作为另一个程序的输入。

管道实际上是利用文件系统中的循环缓冲区。若以生产者消费者模式来使用它，写进程相当于生产者，而读进程相当于消费者。一个进程向管道写东西，如果缓冲区写满了则阻塞写进程。另一个进程从管道中读东西，如果管道为空，则读进程阻塞。当生产者进程或关闭了管道或者该进程死掉的时候，读管道就会失败。如果消费者的进程关闭管道或该进程死掉的时候，写进程就会失败。

1. **管道的实现**

1.实现原理

在父进程中使用pipe系统调用创建一个管道，系统调用的参数是包含两个文件描述符数组pfd[0]和pfd[1]。

1. 使用pfd[0]作为管道输入描述符，使用pfd[1]作为管道输出描述符。
2. fork产生子进程。
3. 现在父进程和子进程都共享了管道文件描述符。每个进程都关闭管道的一端。
4. 每个进程使用dup2把打开的管道描述符副本给标准输入或者标准输出，然后关闭管道描述符。
5. 两个进程可以使用管道通过标准输入和标准输出进行通信。

2.具体实现情况

（1）首先在命令解析中加入对“|”的识别

case '|': //管道标志

if(j != 0){

temp[j] = '\0';

j = 0;

if(!fileFinished){

k++;

temp = buff[k];

}

}

temp = progBuff;

fileFinished = 1;

i++;

break;

（2）为了记录|两边的命令，我在global.h中typedef struct SimpleCmd里增加char \*\*progB; // 管道程序2。下面的代码仍然是命令解析中对cmd->progB赋值。

//如果有管道，则为变量赋值

if(strlen(progBuff) != 0){

cmd->progB = (char\*\*)malloc(sizeof(char\*)\*2);

cmd->progB[1] = NULL;

j = strlen(progBuff);

cmd->progB[0] = (char\*)malloc(sizeof(char) \* (j + 1));

strcpy(cmd->progB[0], progBuff);

（3）在命令执行execOuterCmd函数中具体实现管道的功能

int pid[2];

int pipe\_fd[2];

int len;

int i,j;

char \*prog2\_argv[2];

for(len = 0;cmd->args[len] != NULL;len++)

;

char \*\*prog1\_argv = (char\*\*)malloc(sizeof(char\*)\*(len+1));

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

j = strlen(cmd->args[i]);

prog1\_argv[i] = (char\*)malloc(sizeof(char)\*(j+1));

strcpy(prog1\_argv[i],cmd->args[i]);

}

prog2\_argv[0] = cmd->progB[0];

prog2\_argv[1] = cmd->progB[1];

//创建管道

if(pipe(pipe\_fd) < 0){

perror("pipe failed");

return;

}

//为第一个命令创建进程

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

perror("fork failed");

return;

}

if(!pid[0]){

close(pipe\_fd[0]);

dup2(pipe\_fd[1],1);

close(pipe\_fd[1]);

execvp(prog1\_argv[0],prog1\_argv);

}

if(pid[0]){

if((pid[1]=fork()) < 0){

perror("pipe failed");

return;

}

if(!pid[1]){

close(pipe\_fd[1]);

dup2(pipe\_fd[0],0);

close(pipe\_fd[0]);

execvp(prog2\_argv[0],prog2\_argv);

}

close(pipe\_fd[0]);

close(pipe\_fd[1]);

}

1. **遇到的问题及解决办法**

实现管道的时候遇到的主要有两个问题

1. 命令的读入

开始想先在global里的SimpleCmd的结构里加入一个管道的标记量，然后把管道|两边的命令都读入cmd->args里。在经过一番尝试后发现命令要么是无法正常读入cmd->args里，要么是和其他类型的命令区分不开。最后决定推翻之前的设想，模仿重定向，在SimpleCmd的结构里加入字符数组存储第二个命令，就类似于输出重定向把输出的文件存在\*output里。这样就成功解决了管道两边命令的读入。

1. 命令的参数问题

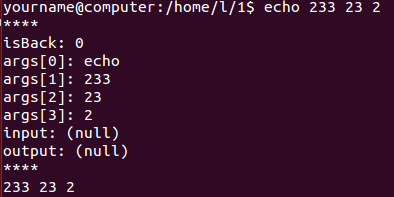
刚开始实现的管道没有考虑带参数的命令，导致如果命令带参数，则管道无法正常工作，因为管道的数组就没有给参数分配空间。

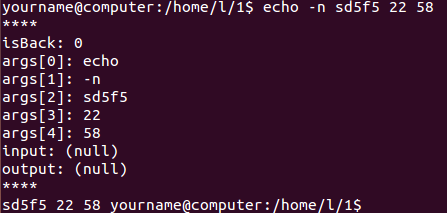
在管道存储命令时不使用固定大小的数组改为动态数组就解决了这个问题。

**3.对常用的内部命令进行实现**

**1.echo命令**

功能是将输入的字符串送往标准输出。输出的字符串间以空白字符隔开, 并在最后加上换行号。命令的一般格式为：echo [ -n ] 字符串，其中选项n表示输出文字后不换行。





通过判断echo命令是否给出-n选项，标记j的值，来决定最后是否输出换行符。

if (strcmp(cmd->args[0], "echo") == 0) {

i = 1;

int j=0;

temp = cmd->args[1];

if(strcmp(temp,"-n")==0){

j = 1;

}

i = j + 1;

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

printf("%s ",cmd->args[i]);

i++;

}

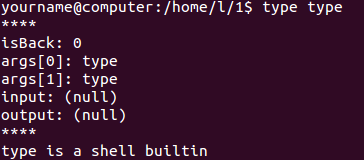
if(j == 0)

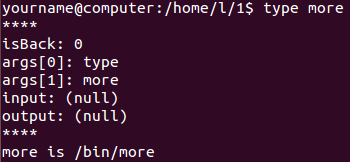
printf("\n");

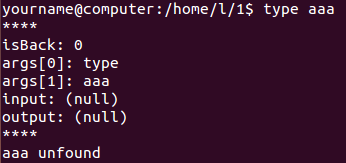
}

**2.type命令**

type命令被用于判断另外一个命令是否是内部命令。命令的一般格式为：type 参数。如果是内部命令，显示该命令为内部命令。如果是外部命令，给出绝对路径。否则显示命令未找到。







首先进行字符串比较判定命令是否为内部命令，是则输出，若不是，则查找指定目录，确定命令是否存在，若存在则输出其绝对路径。若不存在，输出命令未找到。

if (strcmp(cmd->args[0], "type") == 0) {

i = 0;

int flag = 0;

temp = cmd->args[1];

if(strcmp(temp,"exit")==0 || strcmp(temp,"history")==0 ||

strcmp(temp,"jobs")==0 || strcmp(temp,"cd")==0 ||

strcmp(temp,"fg")==0 || strcmp(temp,"bg")==0 ||

strcmp(temp,"echo")==0 || strcmp(temp,"type")==0){

flag = 1;

printf("%s is a shell builtin\n",temp);

}

else{

while(envPath[i] != NULL){

strcpy(cmdBuff, envPath[i]);

strcat(cmdBuff, temp);

if(access(cmdBuff, F\_OK) == 0){

printf("%s is %s\n",temp,cmdBuff);

flag = 1;

break;

}

i++;

}

}

if(flag == 0)

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

}

**4.Ctrl+C命令的实现**

/\*ctrl\_C命令\*/

void ctrl\_C(){

Job \*now = NULL;

Job \*prev = NULL;

if(fgPid == 0){

return;

}

ingnore = 1;

now = head;

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

prev= now;

now = now->next;

}

if(now == NULL){

now = addJob(fgPid);

kill(fgPid, SIGKILL);

strcpy(now->state, "shut down");

return;

}

strcpy(now->state, "shut down");

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);

kill(fgPid, SIGKILL);

if(now == head){

head = now->next;

}else{

prev->next = now->next;

}

free(now);

fgPid = 0;

return;

}

# 收获和感想

这次实验的收获还是蛮大的。

刚开始的时候真是一脸懵逼，虽然有两周的时间，但是第一周因为有面向对象课的智能电梯的作业几乎占用了第一周所有的时间。本实验的第一次实验课，助教就用了十分钟就讲完了，完全什么也不知道。一点思路都没有，根本不知道要干啥。说第一周的任务就是阅读源码，但是当时没有一个明确的目标要干嘛，读源码也还是无用，直到开始着手来做时，再读源码才有思路。本实验第二次实验课仍然是一脸懵逼，还是不知道要干嘛，或者说是告诉我要实现什么目标了但是我根本不知道从哪里下手，根本不知道接下来要干嘛。本实验前一半的时间都完全处于迷茫的状态。就像有同学说，给了我们木头和锤子，让我们造个飞机一样。

第二周没有了OO编程，就可以全身心的投入OS实验了。找了一个已经入手的同学询问了一下，大概需要怎么做，从哪里下手，大概有了一点点思路，便准备做。但是一看代码还是不懂，不知道它运行的原理是啥。于是我便在关键的一些函数里加了很多的printf，然后再通过使用系统自带的shell，才大概明白了程序是如何运行的，各个命令都是怎么样执行的。

随着使用时间的增加，对程序有了进一步的理解。开始先解决程序的3个坑，保证程序能够正常运行。这用了两天的时间。最后解决了发现其实并不是很难，所以这两天的意义不全在于解决bug，更重要的是加深对shell的理解。

基本的要求实现后，便开始准备实现提高要求，一人一个任务。开始还是不太懂，先通过课本和网络学习相关的知识，然后再一步步实现代码。提高要求的实现过程给了我们很多经验教训，尤其是对于组员间的交流合作。刚开始组员间大部分交流都是通过qq或微信进行的。结果出现了两个大问题。第一个，实现CtrlC功能的那个同学一直用的是助教写的那个代码，还以为我们也用的那个，结果我们把他的代码加上后根本不管用，但他却说在他电脑里管用，开始一直没找到问题所在，后来才发现是由于使用不同的源代码，他的里面有两句话我们的里面没有。第二个问题是lex写好后没有及时和总代码合起来测试，分开测试都是正确的，最后一天才合起来，发现了很多问题，幸亏及时解决了。所有给以后实验的经验就是成员之间要多见面交流，不能只局限于网络的交流。

这次实验从完全不知道如何下手到成功完成整个实验对我们能力的培养和锻炼还是非常有帮助的。通过第一次实验，对后面的实验就有一定的思路了。