**操作系统课程设计实验报告**

——实验一：shell实验

负责人姓名：何玥

学号：13211114

日期：2016.3.24

**小组成员**

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 姓名 | 学号 | 实验分工 |
| 1 | 何玥 | 13211114 | 修改源代码和Ctrl-C |
| 2 | 周啸辰 | 14061130 | Lex和YACC |
| 3 | 张诚 | 14061136 | 管道、通配符及内部命令 |
| 4 | 柴舜 | 14061142 | 总结和报告 |

目录

[1.实验目的 4](#_Toc446761347)

[2.需求说明 4](#_Toc446761348)

[2.1基本要求 4](#_Toc446761349)

[2.2 提高要求 4](#_Toc446761350)

[2.3 完成情况 5](#_Toc446761351)

[3.设计说明 6](#_Toc446761352)

[3.1 程序流程图 6](#_Toc446761353)

[3.2 基本要求实现说明 7](#_Toc446761354)

[3.3 提高要求实现说明 9](#_Toc446761355)

[4.遇到的问题及解决方案 24](#_Toc446761356)

[4.1 Lex和YACC实验中的问题 24](#_Toc446761357)

[4.2 修改源代码 25](#_Toc446761358)

[4.3 管道和通配符的实现 26](#_Toc446761359)

[5.收获和感想 27](#_Toc446761360)

# 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

完成了以下功能：

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。

6、尝试对YACC语法分析的文法进行进一步的修改与完善。

7、尝试在Linux下将Lex和YACC结合起来使用进行词法和语法分析。

8、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。

9、实现对管道的支持。

10、考虑并实现组合键***ctrl+c***命令。

基本要求与提高要求都完成。

# 3.设计说明

## 3.1 程序流程图

返回中端管制权

分配终端控制权

Ctrl+Z

Ctrl+C

bg

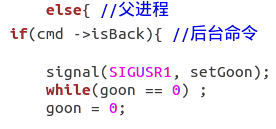
fg

## 3.2 基本要求实现说明

根据对源代码的不断阅读和测试，我们发现源代码在满足实验的基本要求上出现存在以下几个问题，现将问题的提出和我们对问题的解决方案进行陈述：

1、执行外部命令时，父进程执行完kill(pid, SIGUSR1)命令后子进程被杀死，这是由于SIGUSR1 、 SIGUSR2等信号量默认会导致进程退出，所以我们要保证子进程在父进程Kill函数执行前运行。

这里我们可以利用发送和接收信号的机制做一个锁，保证父进程必须等待来自子进程的已经运行的信号后，才能执行Kill函数。

对于父进程：

对于子进程：



这样，只有子进程开始运行，才能够向父进程发送kill信号，父进程接收信号后，得以继续执行，向子进程再发送信号。

2、执行fg %<int>命令继续运行前台作业时，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。而对于一个前台作业，父进程（这里是shell程序）要等待子进程运行结束，再拥有终端的控制权。

这里我们首先引用waitpid函数的第三个参数option, options提供了一些额外的选项来控制waitpid。这里我们可以有三个选择0，WNOHANG和WUNTRACED，可以用"|"运算符把它们连接起来使用 。这里我们选用第三个WUNTRACED，子进程暂停或者结束时会返回。之所以选择这个参数，是因为要处理ctrl-z的问题。

然后，我们可以用一个小技巧来解决问题：



始终等待子进程直至接收，因为除非子进程从暂停或结束状态返回时，waitpid始终会返回-1，父进程就始终在等待了。

3、整个程序最大的问题和bug，就是只是实现了伪前后台切换，左右的子进程都在父进程的进程组在中，实现切换的方式就只是通过父进程（这里是shell程序）的等待。这样的做法虽然看起来实现了切换，但任何一个给子进程的信号，都会传给进程组中的每一个进程，会导致其它的无关进程也收到信号，引起许多未知的后果，比如某些信号会使未安装信号的进程直接死掉。

处理的方法则是依靠两个函数setpgid和tcsetpgrp。前者用来设置进程到一个进程组，后者是用来设置某一个进程为前台的进程。

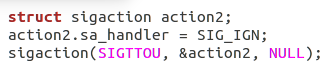
我们可以利用setpgid函数，把一个新的作业，设置为自己的独立进程组，并且通过tcsetpgrp函数设置前台作业控制终端输入。这样，一个信号只会发送给特定的独立进程组，进程组间互不干扰。同时保证了只有一个前台作业，有多个后台做的前后台切换的行为和准则。这样看起来好像已经解决了全部问题？不，我们还需要一个关键的信号就是SIGTTOU信号。

tcsetpgrp函数确实可以用来设置前台进程组，但tcsetpgrp必须是当前拥有终端控制权通过使用该函数，把前台控制权给本次会晤中的某一个进程组。也就是说，当父进程把自己的终端控制权给到子进程后，子进程必须在存在时调用该函数把终端控制权还给父进程。这里就会存在一个很大的问题，子进程被新的覆盖后，我们根本不能保证新的程序会把终端控制还给父进程。这样，子进程死后，终端控制端就失踪了！

幸好我们还有SIGTTOU信号，该信号的作用就是修改终端模式时，帮助父进程夺回终端控制权。当子进程死后，终端控制权会被释放，安装了SIGTTOU信号的父进程就能获得此信息，并调用tcsetpgrp函数夺回终端的控制权。

这样，在涉及前后台调度的地方，我们只要设计好终端控制权的分配和回收，就可很好的实现前后台的调度了。当然，设计时我们还会遇到一些问题，包括整个shell设计的结构也会有所调整。这些我们在问题与解决部分会再来讨论。

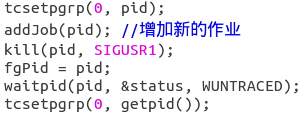
父进程安装SIGTTOU信号：



子进程设置为独立的进程组：



父进程将终端控制权（输入）分配给子进程和在子进程运行结束后收回终端控制权：



## 3.3 提高要求实现说明

**1、Lex和YACC实验**

**LEX**

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（或者常规表达式）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。这一动作可能还包括返回一个标记。 当 Lex 接收到文件或文本形式的输入时，它试图将文本与常规表达式进行匹配。 它一次读入一个输入字符，直到找到一个匹配的模式。 如果能够找到一个匹配的模式，Lex 就执行相关的动作（可能包括返回一个标记）。 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。

**Lex 编程可以分为三步：**

1. 以 Lex 可以理解的格式指定模式相关的动作。
2. 在这一文件上运行 Lex，生成扫描器的 C 代码。
3. 编译和链接 C 代码，生成可执行的扫描器。

Lex的文件形式分为三段，第一段头文件以及声明，用{% ……%}格式，之后定义匹配的内容参数。第二段是匹配格式，用%%……%%格式，中间的格式使用类似正则表达式的格式表述匹配的字符。第三段是一些c代码用来实现一些功能，例如yywrap()函数等。

**YACC**

YACC和LEX的描述方法很像，在第一段位置需要分析文法，lexer 通过 Yacc 的变量 yylval 返回这些值，然后通过文法分析进行程序运行。值得注意的是，在yacc里面通常有一个yyerror()的文件用来报错。通过int yylex()函数可以将lex和yacc结合使用。一个程序通常在每次返回一个标记时都要调用 yylex() 函数。只有在文件结束或者出现错误标记时才会终止。一个由 Yacc 生成的解析器调用 yylex() 函数来获得标记。 yylex() 可以由 Lex 来生成或完全由自己来编写。 对于由 Lex 生成的 lexer 来说，要和 Yacc 结合使用，每当 Lex 中匹配一个模式时都必须返回一个标记。

**首先是修改过的bision.y文件源代码**

%{  
    #include "global.h"  
    #include <string.h>  
    #define YYSTYPE string  
    typedef char\* string;  
    void yyerror ();       
    int yylex();  
    int offset, len, commandDone;  
    int err;  
%}  
  
%token STRING END  
  
%%  
line            :   /\* empty \*/ END {   if(!err) return;}  
                    |command  END   {   if(!err){  
                                            execute();  
                                            commandDone = 1;  
                                            }  
  
                                            return; }  
;  
  
command         :   fgCommand  
                    |fgCommand '&'  
;  
  
fgCommand       :   simpleCmd  
;  
  
simpleCmd       :   progInvocation inputRedirect outputRedirect  
;  
  
progInvocation  :   STRING args  
;  
  
inputRedirect   :   /\* empty \*/    
                    |'<' STRING  
;  
  
outputRedirect  :   /\* empty \*/    
                    |'>' STRING  
;  
  
args            :   /\* empty \*/    
                    |args STRING  
;  
  
  
%%  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
                  词法分析函数  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
                  错误信息执行函数  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
void yyerror()  
{  
    printf("你输入的命令不正确，请重新输入！\n");  
    err = 1;  
}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
                  main主函数  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
int main(int argc, char\*\* argv) {  
    int i;  
    char c;  
  
    init(); //初始化环境  
    commandDone = 0;  
      
    printf("user-sh> "); //打印提示符信息  
  
    while(1){  
        if(!err){  
  
        c = getchar();  
        if(c != -1)  
            inputBuff[0] = c;  
  
        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);  
  
        }  
                      
        for(i = 0;i < 100; i++){  
            inputBuff[i] = 0;  
        }  
    }  
    else{  
        err = 0;  
    }  
              
        printf("user-sh> "); //打印提示符信息  
          
    }  
    return (EXIT\_SUCCESS);  
}

**之后是lex的源代码**

%{  
#include "bison.tab.h"  
#include "global.h"  
#include <stdio.h>  
#include <string.h>  
%}  
  
char [^" "&\<\>\n]  
op [\<\>&]  
blank [" "\t]  
end [\n]  
  
string ({blank}\*{char}\*{op}\*)+  
  
%%  
{string}  {strcat(inputBuff,yytext);return STRING;}  
{end} {return END;}  
%%  
int yywrap(){  
return 1;  
}

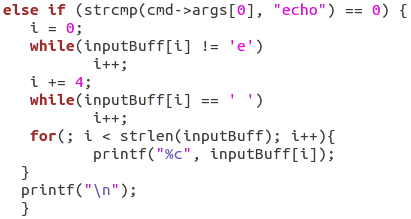
**最后是makefile文件源代码**

user-sh : bison.tab.o execute.o  
cc -o user-sh bison.tab.o execute.o lex.yy.o  
  
bison.tab.o : bison.tab.c global.h lex.yy.c bison.tab.h  
cc -c bison.tab.c lex.yy.c  
  
execute.o : execute.c global.h  
cc -c execute.c  
  
bison.tab.c:  
bison -d bison.y  
  
lex.yy.c :   
lex bison.lex  
  
  
  
clean :  
rm user-sh bison.tab.o execute.o bison.tab.c lex.yy.c  lex.yy.o bison.tab.c

1. **修改源代码bug及添加内部命令**

1、添加内部命令:

1. echo 命令: 对输入的字符串打印出来：这个命令的是相比较简单，从输入流里面取出echo后面，去掉空格的字符串即可：



1. kill命令：有两种形式——kill a 或者 kill -a b。

第一种形式是向进程号为a的进程默认发送信号（15）；

第二种形式是向进程号为b的进程发送信号（a）.

这里的难点主要是分析输入流的格式是否正确，从输入流取出信号值和进程号，并判断是否存在进程号。

2、Ctrl-C

Ctrl-C的实现其实与Ctrl-Z的实现差不多，主要的区别只是Ctrl-C不需打印进程的状态信息，需要向对应的进程发送SIGKILL信号来杀死进程。

基本的思路：判断当前是否有前台进程，有的话通过作业链确认找到进程，向进程发送SIGKILL信号杀死进程，最后父进程重新获得终端的控制权。



然而，在调整了前后台进程的调度之后，Ctrl-C和Ctrl-Z这样的命令其实并不在作用于父进程，而是直接作用于掌握终端控制权的前台进程。所以，Ctrl-Z和Ctrl-C函数其实并没有用到。这里主要是讲一下实现的设计思路。

**3、管道和通配符的实现**

**1、管道**

基本思路是在execSimpleCmd函数开头检测命令的参数列表，如果发现单独为“|”的字符串，则进入管道处理函数，否则正常执行。在该函数中创建两个新的SimpleCmd，记为A和B（程序中是cmd1和cmd2），把“|”之前的内容传递给A，之后的内容传递给B。这一就将原指令转化为两个指令，再用管道将A的输出端和B的输入端连接，将A和B分别放在两个子进程中运行。

**管道的第二种实现**：上述的管道实现两个程序之间数据的互传，对于n个程序之间的管道连接(a | b | c | …)，我们依然可以按照上述的方法解决。不过程序中使用过多的Pipe函数，我觉得还是增加了程序维护的难度，有时也不够稳定。所以我们用另外一种方式来解决多管道连接的问题。

我们用两个临时数据文件来分别保存多管道中某一程序的输入和输出，而下一个程序以上一个程序的输出文件作为自己的输入文件，输入文件作为自己的输出文件。这样，交替进行，通过两个临时数据文件很好地解决了多管道的问题。这里，我们只需保证程序按照次序执行，并分配好输入输出文件，就可以很好地维护管道连接。

对于整个管道命令，我们像正常的外部命令一样，由父进程开辟一个子进程。而这里，我们不是立即找到程序刷掉子进程，而是以此进程为这个任务的进程组长，编写循环体，再开辟子进程，处理具体的程序。所有命令处理完后要返回父进程。

对于管道中一个程序的实现，我们可以将这一命令包括参数从输入的命令中取出，在命令的末尾添加重定向输入输出到临时文件名，构造新的命令。最后调用handleSimpleCmdStr0（handleSimpleCmdStr函数的小的变形）和execSimpleCmd函数，处理程序）。

**2、通配符**

基本思路是在检测完管道后，检测是否存在“\*“、”？“通配符。如果存在，则进入通配符处理函数，否则正常执行。在该函数中，创建一个新的SimpleCmd，记为dupcmd，给它的args申请更大的空间，用于存放可能更多的参数，然后将cmd的内容复制到其中。接下来利用相关函数（来源于网络）获得当前目录下的所有文件及文件夹名称，将其和含通配符的参数一一比对（\*和？采用不同的比对规则），发现符合条件的名称则将其添加至dupcmd参数数组的末尾，最后删除带有通配符的参数，将修改好参数的dupcmd传入execSimpleCmd函数中执行。总之就是先修改参数，再正常执行。

**4、最终的实验结果**

1对基本要求的实现

首先键入make clean清除之前的编译工程

输入make编译工程

打开shell命令输入./usr –sh打开shell命令解释程序

输入./Demo执行一个前台程序，程序的意义是每秒输出一行，共20秒

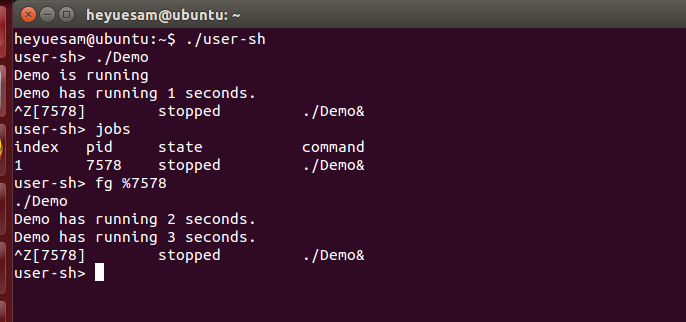
输入ctrl+Z挂起进程

输入jobs查看当前工作情况

输入fg %+挂起进程的进程号

挂起的程序继续运行

再次键入ctrl+Z挂起进程

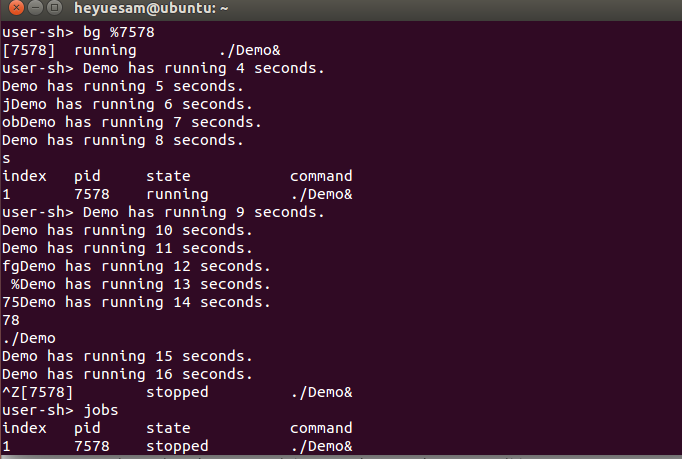


输入bg %+挂起的进程号，挂起的进程在后台执行

此时输入ls打印当前目录的情况

在后台执行进程未完成之前，输入fg %+进程号,让进程放在前台执行

输入ctrl+z，暂停进程，输入jobs查看任务情况



输入history查看输入过的历史命令

输入cd a 进入当前目录文件夹a

输入ls查看是否进入该目录

输入exit结束命令解释程序



2展示个独立进程之间终端控制权的分配和返回

打开命令解释程序

键入./Demo2 程序的意义是无限的输入输出字符串

此时前台终端控制权分配给独立的进程组Demo2

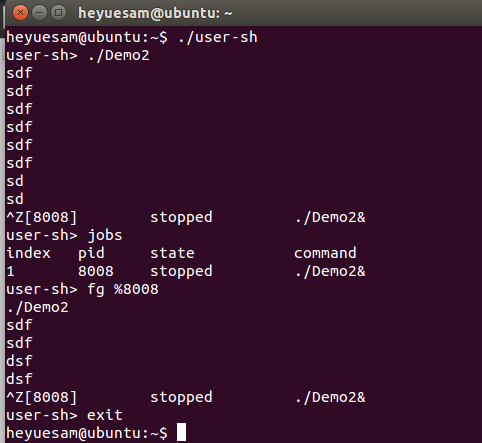
输入ctrl+Z 终端进程，终端控制权返回给父进程，即shell程序

输入jobs查看目录工作状态，检验父进程确实有终端控制权

再输入fg %+进程号

终端控制权再次分配给Demo2进程组，可以输入输出

输入ctrl+Z键入exit结束命令解释程序



3提高要求的展示

打开命令解释程序。

1. 内部命令展示

（1）type命令

输入type type，输出type是一个内部命令

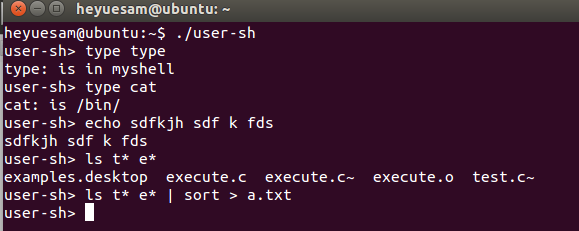
输入type cat显示外部命令cat的路径

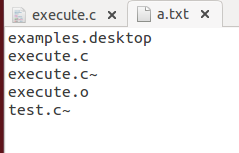
(2)echo命令

输入echo +任意字符串，打印该字符串

1. 通配符的实现

输入ls t\* e\*,打印当前目录下满足开头是t或e的相关文件





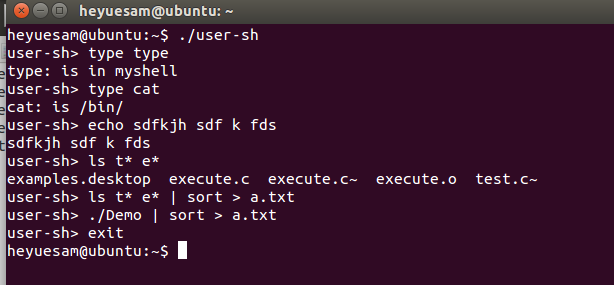
1. 管道的实现

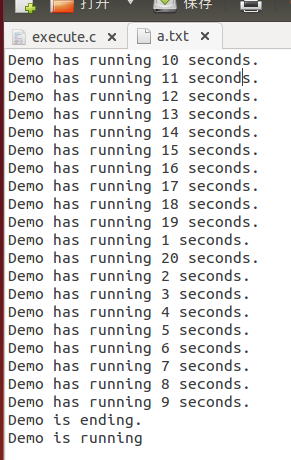
输入ls t\* e\* | sort > a.txt

打开a.txt文件，发现通过管道记录了当前当前目录下符合要求的文件名排序

输入./Demo | sort > a.txt

打开a.txt记录了./Demo程序输出的排序





至此我们完全实现了所有要求。

# 4.遇到的问题及解决方案

## 4.1 Lex和YACC实验中的问题

1、语法有些晦涩难懂，助教给的例子只能作为一个参考，想要完全学会还需要大量的学习，而网上的资料多且杂，想看懂并不是很容易，所以虽然写出来的东西不多，但是耗费了大量的时间

2、修改makefile文件，makefile文件是从下往上读，一开始不知道这个所以饶了很多弯路，导致各种各样的错误。

3、修改makefile文件时，编译某些文件时如果需要别的头文件，则需要把头文件和该文件放在一起编译。

4、从YACC读进去的指令不知道什么时候结束，即“\n”并不能被当作指令结束的标识，将命令输入之后敲回车，结果把“\n”被当成字符一起读入。这是个大坑，解决办法是在.y里面再token一个量专门来检测“\n”，用来判断返回结束。

5、由于把输入的函数全部注释掉了，因此inputBuff无法被传递数据，所以直接在Lex里面使用strcat函数把读进来的字符加到数组里面去，一旦出现问题就在报错的函数里面清楚数组。

6、在程序运行的过程中有的时候会莫名其妙的返回一个EOF之类，然而返回这个的时候会作为输入端从lex输入，但是EOF这个问题lex是无法判断解决的，这就造成了只要是后台运行结束的程序一定会崩溃。后来想到的方法是直接把inputBuff放到lex里面（全局变量），在yacc中单独判断第一个字符是什么，如果是EOF就什么都不做，重置inputBuff，如果是正常输入，那么就把这个字传入inputBuff的第一个位置，接下来读入的字符串都在lex里面进行，并且用strcat函数将输入的内容拼接到inputBuff中，完成对EOF的检测。

## 4.2 修改源代码

本次实验中，我们遇到的最大的问题和挑战就是处理前后台进程调度的问题了。大体的设计和代码实现已经在上文说过。然而，当我们把所有的命令（外部命令）创建独立的进程组，并通过tcsetpgrp函数不断的有父进程（shell）设置（分配和回收）终端控制权调度进程时，源代码的结构显然存在问题，我们需要调整设计的结构来满实验中对进程调度的要求。

这里有三个主要的变化：

1、 所有子进程独立为进程组后，父进程（shell）和各子进程仅通过”父子“关系连接，当子进程被新的程序刷掉之后，父进程对子进程的掌握只通过waitpid函数，得到子进程暂停和结束运行的信息。

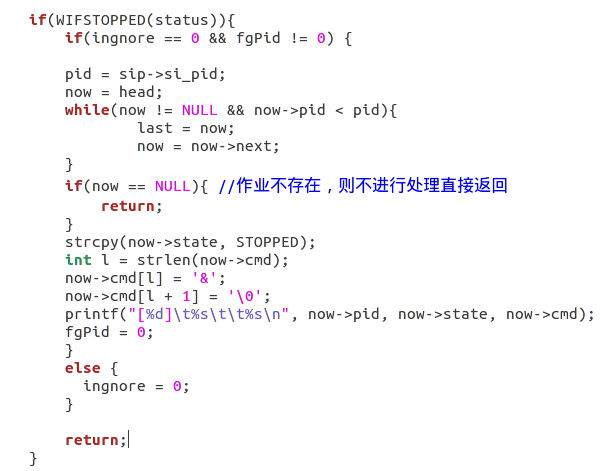
2、 Ctrl-Z和Ctrl-C函数直接作用与前台运行的子进程，直接中断和终止子进程，而不再通过父进程（shell）做中介。

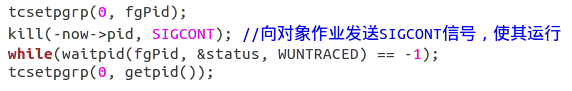
3、 对于进程控制的几种操作fg,bg,Ctrl-Z和Ctrl-C父进程都要分配和及时回收终端的控制权。

而解决这些问题，我们需要依赖waitpid函数，因为这是父进程与前台执行的子进程的唯一联系。waitpid函数的第二个参数起到了关键的作用。第二个参数status 是一个宏变量，表示子进程的状态。这里，我们使用WIFSTOPPED(status) ，如果子进程处于暂停执行情况则此宏值为真。通过这个参数，父进程可以在子进程跳出运行后，判断子进程的状态，从而执行相应的操作。我们的设计中，将这个参数的判断写在rmJob的函数中。子进程一旦跳出运行，一定会返回给父进程SIGCHID信号，进入rmJob函数，我们则判断子进程是正常结束还是被Ctrl-Z中断。

在fg函数中，我们把一个进程调到前台运行时，也要给进程终端的控制权，并在进程返回后，回收控制权。而我们意外的一个现象是，即使一个进程被唤起，进程依然会返回SIGCHID信号，进入rmJob函数，所以我们要设置一个全局变量来表示子进程是由于怎样的操作进入rmJob函数的。

我们解决问题的具体代码如下：

rm\_job中的判断过程：

fg函数中的终端控制权分配与回收：

使用waitpid函数获取进程的状态：



## 4.3 管道和通配符的实现

**1、管道**

对A和B的args赋值时，没有事先申请空间，导致运行时出错，用malloc申请空间后即可解决。

多次输出，即同样的内容重复输出了两遍。原因在于父进程的waitpid函数，最初该函数写法完全参考实验课本上的示例，后改为waitpid(pidpid[0], &status, WUNTRACED);即解决。

当”|”和其他参数之间没有空格时，程序未能将其区分出来。解决方法为参考原有代码中对重定向符号“>””<“的处理，在handleSimpleCmdStr0函数的switch(inputbuff[i])下添加一种case。

**2、通配符**

测试最初反复遇到出错情况，主要在于修改参数的操作不够谨慎。当删除某个参数时，它后方所有参数的下标数值会减一，由于忽视这类情况，导致初步调试过程中出现各种数组越界问题。

如果没有找到符合条件的名称，则程序会出错。原因在于没有经过任何修改的dupcmd在传入execSimpleCmd之后会再次进入通配符处理函数，如此反复循环，大量占据内存。解决方法为，在没有找到符合条件的文件名称时，直接把cmd传入execOuterCmd函数中执行。

# 5.收获和感想

首先刚刚拿到实验要求的我们是震惊的，这感觉用之前答疑群里一位哥们的话讲，真的就是把要求丢给我们之后，生生用榔头造飞机(甚至连图纸和材料都没有)。

对于没有材料这一点，也就是知识上的问题我们可以选择自学，查资料，多方询问的各种各样的办法来解决，但是没有图纸的问题……连做实验的大体方向都没有，要做实验实在是太困难了。只是甩下要求，然后啥也不说，连点提示都不给就要做整个实验，实在是难上加难，好歹去年的计组还有实验指导书，怎样实现要求也有一定的提示和基本要求，相对还是轻松不少，这次的OS实验简直天坑。由于没有相对明确的方向，最终导致时间大量的浪费，很多时候漫无目的（没错，就是像楞头苍蝇一样到处乱撞，自己甚至都不知道要搜索的关键字能不能解决目前遇到的问题）搜索感觉就是在赌，运气好，节约不少时间，运气不好，行了，基本上搜一圈都找不到有用的。

在不断地摸索中寻找方向，之后开始进入最艰难的部分:面相玄学的代码和操作系统。在找到有用的东西，并加以学习和了解之后，还是对源代码进行修改，虽然最后的改动不是很大（但是听说有的组已经改的面目全非了……）这个过程中需要学习和理解的东西很多，很麻烦。总的来讲就是恶心……但是更恶心的是那不科学的代码和操作系统:也许在之前还跑得好好的程序过一会结果就崩了；也许在你自己的电脑上跑得好好的程序在别人的电脑上崩了；也许你在功能都已经实现了之后，稍作改动，发现崩了；也许刚才你完全调不通的程序被你乱改一气竟然就没事了……所以，可以说这次实验真的是面相玄学的实验。

最终在一周没有OO大作业的条件下终于完成了本次实验，要说到收获，自然是有很多的，首先还是自学是真的能学到东西的，但是如果能有更加明确的方向的话，自学会更有效率也会节约很多时间。其次就是要多于别人交流沟通，通过听取别人的意见和指导，能够少走很多的弯路。然后就是团队协作的重要性，讲真的，这么大的实验，分工合作尚且有很多困难更别说一个人来完成整个大实验了，没有团队的力量是绝对没有办法两周之内搞完这个实验的。最后就是无论当前的状况有多困难，都不应该放弃，要坚信胜利的曙光迟早会出席那在你眼前!

最后的最后，希望助教可以在以后的实验中多提供一些可以学习的途径方便完成实验，不然学习的过程将非常困难耗费大量时间，别的课程会受到影响。