操作系统实验报告：shell

# 1需求说明

## 1.1基本要求和提高要求

### 1.1.1基本要求

（1）能够执行fg、bg、cd、history、exit、jobs等内部命令。

（2）能够执行外部程序命令，命令可以带参数。

（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，包括打印作业的清单，改变当前运行作业的前台/后台作业，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或者ddd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

### 1.1.2 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，使用进行词法和语法分析。

（3）实现对管道的支持。

（4）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。

（5）参照组合键Ctrl+Z命令的实现方法，考虑并实现组合键Ctrl+C命令。

## 1.2完成情况

本次实验的基本要求和提高要求本小组基本实现。

# 2设计说明

## ctrl+C的实现：

基本模仿Ctrl+Z的实现。主要区别有：

1. 与Ctrl+Z相对应的信号为SIGTSTP，与Ctrl+C相对应的信号为SIGINT.
2. Ctrl+Z会使ignore=1,以屏蔽rmJob,但Ctrl+C恰恰相反，调用rmJob移除前台作业。

但实现Ctrl+C后，发现Ctrl+C不仅会杀死前台作业，也会杀死后台作业，经过调试，发现后台作业运行时，若前台发送Ctrl+C信号，并不是Ctrl\_C()函数与rmJob()函数杀死了后台作业，经过查阅《UNIX高级环境编程》发现，这是因为子进程默认情况下会初始化信号处理方式，即按照默认处理方式（杀死作业）来进行处理，于是我组尝试着在执行外部指令创建子进程时屏蔽Ctrl+C信号，结果发现是可行的。屏蔽信号的指令：(Ctrl+Z的处理方式同理)

signal(SIGINT, SIG\_IGN);

signal(SIGTSTP, SIG\_IGN);

## 管道的实现：

在管道实现中，有3个十分重要的系统调用，fork();pipe(int fildes[2]);dup2(int fildes,int fildes2).

fork()是创建新的进程；pipe(int fildes[2])是创建一个管道，把管道的读和写文件描述符放到数组fileds中；dup2(int fildes,int fildes2)是把fileds文件描述符复制给fildes2.如果fildes2已经打开，则先将其关闭，然后进行复制，使fildes和fildes2指向同一文件。

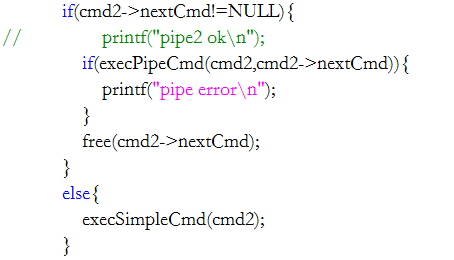
管道实现的主要思路如下：

先创建一个管道，然后在当前进程创建两个子进程，将第一个子进程的标准输出定向到管道的输入端，将第二个子进程的标准输入定向到管道的输出端，这样，执行两个子进程时就可以进行通信了。

当然，这只是实现了两个进程之间的通信，但实际中还可能出现多个进程之间进行通信。为了实现这一功能，首先，要更改SimpleCmd的结构。在原有结构的基础上加上Simple \*nextCmd,用链表来存储不定长的管道指令。这样，只需要在解析命令时，递归调用自身来不断解析并有序存储管道指令。

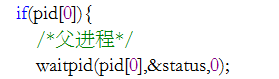


命令解析完毕后，就要开始执行了。执行指令也是采取递归式的执行。

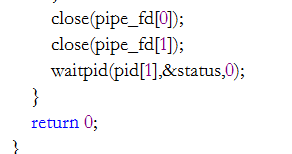


在实现管道指令后，同样发现了bug，即父进程并没有等待子进程就运行，导致命令提示符在任务未执行完毕时就显示了出来，严重影响与用户的交互性，所以，父进程必须等待两个子进程：

等待第一个子进程：

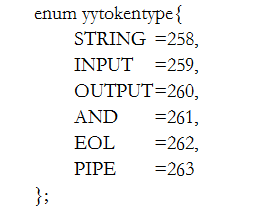


等待第二个子进程：



## flex与bison结合：

本次用到的token有：



bison只是稍作修改，每个命令都是以EOL（回车符）结束，执行完指令后return。添加了对管道指令的描述：

fgCommand : simpleCmd

|fgCommand PIPE fgCommand

在实现此要求过程中，主要遇到两个困难：第一，如何在使用flex生成的yylex()时获取inputBuff；第二，为何后台运行的程序一结束flex必崩。

对于第一个问题，最初的想法是通过getchar()获取命令，并将其存入文本文件中，再将flex的输入源定向到该文本文件，但感觉略有麻烦，且效率较低，于是，采取了利用全局变量、在判断token的同时将字符存入inputBuff中。

对于第二个问题，经过不断的测试，发现是因为每个后台运行的程序结束后总会返回一个int值为-1的字符给标准输入，但flex无法识别，故崩溃。为了解决这一问题，我采取了用getchar()预先判断输入的第一个字符，若大于零，即flex能够识别，则回退给标准输入(int ungetc(int c, FILE \*stream);)，让flex继续处理;；若小于零，则不返回给标准输入。

## 实现通配符

最开始想用C语言写一个匹配\*与？的程序，和一个列出当前文件夹下所有文件的程序，当写完后，发现若路径中也出现通配符则工作量巨大，于是，采取了shell脚本，将通配符的处理交给shell。

每当指令中出现?或\*时，将该指令存入.shell文本中，然后执行bash指令，运行此脚本。该脚本中除了有含有统配符的指令外，还有删除自身的指令，以免给用户产生垃圾。

## 解决bg的bug

经过测试发现，源代码中的bg指令并不能真正将作业挂到后台并运行，原因：fork()需要一定时间，但子进程还未创建时，父进程已经发送了用于解锁子进程的信号，最终导致死锁。找到原因后，解决方案也自然就产生了，即：加上一种同步机制，在只有进入子进程后父进程才能发送解锁子进程的信号。最初想采用全局变量来实现，发现行不通，这也让我明白了：全局变量的范围仅限于同一个进程，不同进程之间不能共享全局变量，这也是信号的重要性吧。最终，我模仿SIGUSR1定义了一个SIGUSR2，达到了先子后父这样一种同步机制，bg指令的bug已修复。

## 解决fg的bug

fg的bug也是因为时间差问题导致了先打出命令提示符后运行作业，具体原因：向对象作业发送信号使其运行需要一定的时间，然而，当子进程还未收到此信号时父进程就执行了wait函数，但这时并不需要等待，于是此处的wait做了无用功。最好的方法也是为这两个进程设立先子后父的同步机制，但此处我偷了个懒，只是在wait前面sleep了一下。

## 其他bug

还有其他小bug（比如，每一个外部指令导致运行的作业运行结束后都未将fgPid置为零，导致ctrl+z会挂起一个早就结束的作业）等等，在此就不一一赘述了。

# 3收获与感想

通过本次实验，本小组熟练了linux的基本操作，对shell的实现有了初步掌握，并且深刻理解了理论课上所讲的进程通信的例子，深刻体会到了并发程序所带来的一系列问题、并掌握最基本的解决方法，还学习了功能强大的bison与flex。除上述之外，还加强了组员之间的合作能力、解决问题能力，提高了学习氛围，有助于以后的学习与进步。