操作系统第一次实验报告

1. 实验内容：Shell，具有部分功能，具体如下：
   1. 执行内部命令：cd、history、fg、bg、exit、jobs；
   2. 在当前目录和环境变量指定的目录下寻找外部命令并执行；
   3. 通过在命令末尾添加&让命令一开始就在后台运行；
   4. 支持Ctrl-Z挂起当前进程组；
   5. 添加了内部命令：echo、pwd、clear、reset、type（提高要求）；
   6. Ctrl-C结束当前进程组（提高要求）；
   7. 支持管道，包括多组管道进程的前后台切换，每组管道支持多个进程（提高要求）；
   8. 用lex替换yylex函数实现词法分析（提高要求）。
2. 实验分工（按学号顺序）：
   1. 黄秋宇（14061176）：Lex/Yacc改写与调试，内部命令；
   2. 李搏（14061187）（主要负责人）：总体设计，添加管道；
   3. 林子义（14061194）：部分Bug修正，实验报告主体部分；
   4. 胡亚龙（14061196）：修改CtrlC/Z处理方式，总体测试，会议记录。
3. 实验原理
   1. Linux创建新进程的方法

在Linux中，唯一的创建新进程的方法是使用fork函数（或vfork，类似），但fork函数的作用不是创建一个新的进程，而是将原进程克隆一遍，赋予新进程新的pid和地址空间，但从相同的位置开始执行相同的代码。程序可通过fork的返回值判断自己是新进程还是原进程。子进程随后可以通过exec系列函数，加载新的可执行文件到自己的内存空间中并开始执行。

Linux的fork-exec机制，使得父进程在创建子进程的时候，可以在新程序开始运行之前仍保留短暂的控制权，从而实现了较高的灵活性。

* 1. 进程、进程组、前后台进程组

本次实验需要处理的另外一个问题是前后台进程的管理问题。如果不对进程进行分组处理，使用CtrlC、CtrlZ产生的信号将传播到shell产生的所有子进程，这就会导致前后台子进程一起终止。

为解决这个问题，我们需要使用进程组的机制，Linux会将CtrlC/Z产生的信号发送到前台进程组的所有进程，因此，只需要为每一个创建的子进程建立自己的进程组，就可以实现对CtrlC/Z信号的隔离。

隔离以后，还需要管理进程组的前后台切换问题。在运行前台程序的时候，shell需要将前台控制权交予子进程，并等待子进程完成后收回控制权；如果是后台进程，则无需做特殊的处理。

* 1. 信号与进程间通信

Shell要实现对创建的子进程进行管理，需要与子进程进行通信。进程间通信使用的是Linux系统提供的信号机制。本次对信号的运用主要在两个方面：通过SIGUSR1实现父进程与子进程的同步工作；当子进程状态发生变化时，通知shell更新子进程运行信息。

对SIGUSR1的运用流程大致如下：当创建后台进程时，父进程等待子进程完成输入输出重定向；子进程随后通知父进程完成，自身进入等待状态，父进程完成addJob工作；父进程通知子进程完成，并等待子进程完成运行信息输出；子进程完成输出后，通知父进程，返回正常工作状态。其中两进程间的“通知”与处理工作就是通过发送SIGUSR1信号和setGoon函数完成的。

Shell需要处理的另一个信号是SIGCHLD。SIGCHLD是子进程运行状态发生变化（挂起、继续运行、结束、后台进程尝试向标准输出写入等）时向父进程发送的信号。SIGCHLD包含了子进程发送信号的详细原因，因此与普通的信号处理函数有所不同，处理程序接受的参数更多，注册处理函数也需要使用sigaction而不是signal。SIGCHLD需要忽略部分信号，并在子进程挂起、结束时更新任务列表。

事实上，shell并不需要特别处理SIGINT、SIGTSTP两个信号：一方面，当没有前台进程时，它们应该被忽略；另一方面，当有前台进程时，它们会首先到达前台进程而非shell，随后前台进程发送SIGCHLD给shell。可见，shell并没有需要直接处理CtrlC、CtrlZ的情况，只需要设置为收到这两个信号的时候忽略即可。

* 1. 词法分析和语法分析

Shell需要对输入的命令进行语法检查与归类，本次实验的做法是使用词法分析器lex与语法分析器yacc。

Lex的功能是将输入单字拼成词，根据预先设定的类似于正则表达式的模式，依次识别输入各字符，遇到匹配的情况执行相应操作，特别地在本次实验中，是返回一个特定值通知yacc找到了相应的单词。Yacc使用BNF定义命令的各个部分，并组成最终的合法命令格式，通过依次接收lex的返回信息，判断输入能否组成一种合法的命令。

本实验中，在lex中直接实现了几乎所有的命令识别功能，直接将命令识别为PIPE、COMMAND两种模式并返回给yacc，yacc根据模式采取对应的动作（管道和普通命令使用不同的处理函数）。这里相对于原有程序的功能分布有较大差别。

事实上，shell的功能非常强大，以至于几乎就是一种编程语言，因此，需要使用Lex+Yacc的组合以进行更加复杂的语法分析。本次实验并没有实现更加复杂的功能，但这一工具为shell功能的继续扩展保留了空间。

* 1. 输入输出重定向、管道

本实验的shell支持对外部命令进行输入输出重定向。对重定向的处理是在子进程fork出去之后，此时，如果是普通重定向，子进程完成打开文件与重定向的功能，使用的库函数是open和dup2，open打开一个磁盘文件并获得文件描述符，dup2将打开的文件与标准输入输出关联。对于管道，由父进程创建管道，随后将管道两端交由fork得到的子进程进行关联。文件重定向的优先级高于管道。将管道关联后，可能需要关闭部分无用的管道副本。

程序的其他部分实现原理较为简单，此处不再赘述。

1. 实验遇到的奇葩bug和解决办法
   1. shell有时会收到一个EOF，这会导致lex读入失败并崩溃。这种情况通常发生在启动的一个后台程序运行完后，具体的原因不明。  
      解决方法：lex生成代码以后，手动修改lex的读入函数。现有的代码中读入函数的定义是int my\_getinput(char \*s, int max\_len);
   2. 等待管道时，使用sigsuspend而不是waitpid，因为当一个管道结束时，多个子进程同时向父进程发送信号，期间很可能有信号丢失，因此由运行转为暂停时，针对特定进程的waitpid很可能会导致shell锁死。  
      解决方法是等待管道时使用sigsuspend，并且每次接收到一个SIGCHLD信号就处理多个进程的状态，直到前台进程全部处理完毕，此时wait\_pipe的循环条件不满足，退出循环，shell继续运行。
2. 实验收获

经过本次实验，我们熟悉了Linux的进程管理方法，熟悉了Linux为进程创建和进程间通信提供的部分系统函数。进程管理是现代操作系统为用户提供的最重要的功能之一，也能在一定程度上体现出操作系统的设计思想。在本次实验中，我们尝试了通过查阅Linux在内核对几个系统调用的实现来理解Linux的进程管理机制。例如Linux的fork，会复制原进程的页表，将原进程的文件描述符计数加一，但保留原进程的内存内容，使得两个看起来完全相同的进程开始在不同的上下文中执行，为了降低fork后拷贝所有内存内容的开销，又使用了写时拷贝，为了加载新程序，还需另外使用系统函数exec。Linux内核的进程管理采用树形模式：除PID为1的init进程由内核启动，另外的进程都由某一进程启动，即有一父进程，所有进程形成以init为根的“进程树”；每一进程退出后，其进程信息不会立刻被系统清除，而是继续保留以供父进程获取子进程的退出信息，但此时进程不会获得CPU调度，进程进入了“僵尸”状态，在shell实验的某些情况下，需要手动使用wait函数清除僵尸进程；Linux对于子进程运行期间丢失父进程的情况也有对应的处理机制……但是这套看似繁琐、不那么直观的进程管理机制却被认为是最高效的处理方法之一，以至于Linux只需要将线程视作共享内存空间的进程而无需额外的定义；在进程创建性能测试中，Linux通常也以压倒性的优势领先于Windows，这从一个侧面说明了Linux系统的博大精深。由于时间所限，我们无法对相关机制做进一步的分析，以后希望能有机会将Linux内核中进程机制的实现方法与其他操作系统进行对比研究，从而对其设计思想有更加深刻的理解。

通过SIGUSR1信号的处理程序，我们初步了解了使用信号进行进程间同步的方法；我们也初步了解了Lex和Yacc的使用流程，为以后的编译课程做了一点微小的准备。

除Shell外，另外三次试验都是内核中的内容，shell实验的意义，在于使我们对Linux的基本工作机制有所了解，初步对Linux的内核“从外看”是什么样有了一定的感觉，从而为了解后三次内核实验需要提供的功能打下了基础。