

**实验报告**

**实 验（七）**

题 目 TinyShell

微壳

专 业 计算机类

学　　 号 1190202405

班　　 级 1903005

学 生 林逸灏

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2021.06.02

**计算机科学与技术学院**

**目 录**

[第1章 实验基本信息 - 4 -](#_Toc26657523)

[1.1 实验目的 - 4 -](#_Toc26657524)

[1.2 实验环境与工具 - 4 -](#_Toc26657525)

[1.2.1 硬件环境 - 4 -](#_Toc26657526)

[1.2.2 软件环境 - 4 -](#_Toc26657527)

[1.2.3 开发工具 - 4 -](#_Toc26657528)

[1.3 实验预习 - 4 -](#_Toc26657529)

[第2章 实验预习 - 6 -](#_Toc26657530)

[2.1 进程的概念、创建和回收方法（5分） - 6 -](#_Toc26657531)

[2.2信号的机制、种类（5分） - 6 -](#_Toc26657532)

[2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5分） - 6 -](#_Toc26657533)

[2.4 什么是shell，功能和处理流程（5分） - 6 -](#_Toc26657534)

[第3章 TinyShell的设计与实现 - 7 -](#_Toc26657535)

[3.1.1 void eval(char \*cmdline)函数（10分） - 7 -](#_Toc26657536)

[3. 1.2 int builtin\_cmd(char \*\*argv)函数（5分） - 7 -](#_Toc26657537)

[3. 1.3 void do\_bgfg(char \*\*argv) 函数（5分） - 7 -](#_Toc26657538)

[3. 1.4 void waitfg(pid\_t pid) 函数（5分） - 7 -](#_Toc26657539)

[3. 1.5 void sigchld\_handler(int sig) 函数（10分） - 8 -](#_Toc26657540)

[第4章 TinyShell测试 - 10 -](#_Toc26657541)

[4.1 测试方法 - 10 -](#_Toc26657542)

[4.2 测试结果评价 - 10 -](#_Toc26657543)

[4.3 自测试结果 - 10 -](#_Toc26657544)

[4.3.1测试用例trace01.txt - 10 -](#_Toc26657545)

[4.3.2测试用例trace02.txt - 11 -](#_Toc26657546)

[4.3.3测试用例trace03.txt - 11 -](#_Toc26657547)

[4.3.4测试用例trace04.txt - 11 -](#_Toc26657548)

[4.3.5测试用例trace05.txt - 11 -](#_Toc26657549)

[4.3.6测试用例trace06.txt - 12 -](#_Toc26657550)

[4.3.7测试用例trace07.txt - 12 -](#_Toc26657551)

[4.3.8测试用例trace08.txt - 12 -](#_Toc26657552)

[4.3.9测试用例trace09.txt - 13 -](#_Toc26657553)

[4.3.10测试用例trace10.txt - 13 -](#_Toc26657554)

[4.3.11测试用例trace11.txt - 13 -](#_Toc26657555)

[4.3.12测试用例trace12.txt - 14 -](#_Toc26657556)

[4.3.13测试用例trace13.txt - 14 -](#_Toc26657557)

[4.3.14测试用例trace14.txt - 14 -](#_Toc26657558)

[4.3.15测试用例trace15.txt - 15 -](#_Toc26657559)

[4.4 自测试评分 - 15 -](#_Toc26657560)

[第5章 总结 - 16 -](#_Toc26657561)

[5.1 请总结本次实验的收获 - 16 -](#_Toc26657562)

[5.2 请给出对本次实验内容的建议 - 16 -](#_Toc26657563)

[参考文献 - 18 -](#_Toc26657564)

# 第1章 实验基本信息

## 1.1 实验目的

* + 理解现代计算机系统进程与并发的基本知识
  + 掌握linux 异常控制流和信号机制的基本原理和相关系统函数
  + 掌握shell的基本原理和实现方法
  + 深入理解Linux信号响应可能导致的并发冲突及解决方法
  + 培养Linux下的软件系统开发与测试能力

## 1.2 实验环境与工具

### 1.2.1 硬件环境

* + X64 CPU；2GHz；2G RAM；256GHD Disk 以上

### 1.2.2 软件环境

* + Windows7 64位以上；VirtualBox/Vmware 11以上；Ubuntu 16.04 LTS 64位/优麒麟 64位

### 1.2.3 开发工具

Visual Studio 2010 64位以上；Shell；gcc等

## 1.3 实验预习

填写

# 第2章 实验预习

**总分20分**

## 2.1 进程的概念、创建和回收方法（5分）

进程的概念：进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文( context)中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据,它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

进程的创建方法：在C语言中，我们可以使用fork函数来创建一个新的进程。父进程使用fork函数，创建一个新的子进程。

进程的回收方法：（1）当一个进程由于某种原因终止时,内核并不是立即把它从系统中清除。相反,进程被保持在一种已终止的状态中,直到被它的父进程回收( reaped)。当父进程回收已终止的子进程时,内核将子进程的退出状态传递给父进程,然后抛弃已终止的进程,从此时开始, 该进程就不存在了。一个终止了但还未被回收的进程称为僵死进程( zombie)。

（2）如果一个父进程终止了, 内核会安排init进程成为它的孤儿进程的养父.Init进程的PID为1, 是在系统启动时由内核创建的,它不会终止,是所有进程的祖先。如果父进程没有回收它的僵死子进程就终止了, 那么内核会安排inlt进程去回收它们.不过,长时间运行的程序.比如shell或者服务器,总是应该回收它们的僵死子进程。即使僵死子进程没有运行,它们m然消耗系统的内存资源。

## 2.2信号的机制、种类（5分）

信号的机制：一个信号就是一条小消息,它通知进程系统中发生了一个某种类型的事件。它允许进程和内核中断其他进程。信号类型是用小整数ID来标识，每一个信号对应唯一的ID。

发送信号：内核通过更新目的进程上下文中的某个状态,发送(递送)一个信号给目的进程。发送信号可以有如下两种原因:1)内核检测到一个系统事件,比如除零错误或者子进程终止。2)一个进程调用了ki11函数，显式地要求内核发送一个信号给目的进程。一个进程可以发送信号给它自己。

接收信号：当目的进程被内核强迫以某种方式对信号的发送做出反应时,它就接收了信号。进程可以忽略这个信号,终止或者通过执行一个称为信号处理程序(signal handler)的用户层函数捕获这个信号。图8-27给出了信号处理程序捕获信号的基本思想。

信号的种类：



## 2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5分）

信号的发送方法：

1.用/bin/kill程序发送信号，/bin/kill程序可以向另外的进程发送任意的信号

2.从键盘发送信号，在键盘上输入Ctrl+C会导致内核发送一个SIGINT信号到前台进程中的每个进程，默认情况下是终止前台作业

3.用kill函数发送信号给其他进程（包括他们自己）

4.使用alarm函数发送信号，进程可以通过调用 alarm 函数在指定 secs 秒后发送一个 SIGALRM 信号给调用进程。

信号的阻塞方法：

- 隐式阻塞机制。内核默认阻塞任何当前处理程序正在处理信号类型的待处理的信号。例如,图8-31中,假设程序捕获了信号s,当前正在运行处理程序S。如果发送给该进程另一个信号s,那么直到处理程序S返回,s会变成待处理而没有被接收。

- 显式阻塞机制。应用程序可以使用 sigprocmask函数和它的辅助函数,明确地阻塞和解除阻塞选定的信号。

处理程序的设置方法：

可以使用 signal函数修改和信号signum相关联的默认行为:

handler\_t \*signal(int signum, handler\_t \*handler)

handler的不同取值:

1. SIG\_IGN: 忽略类型为signum的信号

2. SIG\_DFL: 类型为 signum的信号行为恢复为默认行为

3. 否则， handler就是用户定义的函数的地址，这个函数称为信号处理程序

  只要进程接收到类型为 signum 的信号就会调用信号处理程序

将处理程序的地址传递到signal函数从而改变默认行为，这叫作设置信号处理程序。调用信号处理程序称为捕获信号

执行信号处理程序称为处理信号

当处理程序执行return时，控制会传递到控制流中被信号接收所中断的指令处

## 2.4 什么是shell，功能和处理流程（5分）

1.shell的定义：

shell是一个交互型应用级程序，代表用户运行其他程序。是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。

2.功能：

其实shell 也是一个应用程序，它由输入设备读取命令，再将其转为计算机可以了解的机械码，然后执行它。各种操作系统都有它自己的 shell，以 DOS 为例，它的 shell 就是 command.com文件。如同 DOS 下有 NDOS，4DOS，DRDOS 等不同的命令解译程序可以取代标准的command.com ，UNIX 下除了 Bourne shell（/bin/sh） 外还有 C shell（/bin/csh）、Korn shell（/bin/ksh）、Bourne again shell（/bin/bash）、Tenex C shell（tcsh）、zsh等其它的 shell。UNIX/linux将 shell 独立于核心程序之外， 使得它就如同一般的应用程序， 可以在不影响操作系统本身的情况下进行修改、更新版本或是添加新的功能。

Shell是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。不仅如此，Shell有自己的编程语言用于对命令的编辑，它允许用户编写由shell命令组成的程序。Shell编程语言具有普通编程语言的很多特点，比如它也有循环结构和分支控制结构等，用这种编程语言编写的Shell程序与其他应用程序具有同样的效果

3.处理流程：

shell首先检查命令是否是内部命令，若不是再检查是否是一个应用程序（这里的应用程序可以是Linux本身的实用程序，如ls和rm，也可以是购买的商业程序，如xv，或者是自由软件，如emacs）。然后shell在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果键入的命令不是一个内部命令并且在路径里没有找到这个可执行文件，将会显示一条错误信息。如果能够成功找到命令，该内部命令或应用程序将被分解为系统调用并传给Linux内核。

# 第3章 TinyShell的设计与实现

**总分45分**

**3.1 设计**

## 3.1.1 void eval(char \*cmdline)函数（10分）

函数功能：

解析用户输入的命令行命令并执行相关操作。若用户输入了内置指令，如：quit、bg、fg、jobs等，直接执行这些命令。若用户输入的不是内置指令，则fork一个新的子进程，并将该作业在子进程的上下文中运行。当该作业为前台作业时，需要等待它运行结束才返回。当该作业为后台作业时，则打印进程信息。

参 数：char \*cmdline

处理流程：

1. 先调用parseline函数解析输入的cmdline，将cmdline划分为参数数组argv，并且parseline函数的返回值为bg，判断该操作是否需要后台运行。
2. if条件语句中调用buildin\_cmd判断输入的指令是否为内置指令，若是内置指令，则直接执行该指令。
3. 若不是内置指令，则创建一个子进程并添加在job的列表中。
4. 最后判断进程，若是后台进程，则打印当前进程的信息；若是前台进程，则调用waitfg(pid)函数，等待它结束。

要点分析：

1. 该函数的实现中，阻塞的时机需要仔细考虑，在创建子进程中，需要将SIGCHLD、SIGINT、SIGSTP信号阻塞；在父进程中使用addjob函数也需要用sigprocmask函数来阻塞信号；这些过程结束后，父进程需要调用sigprocmask函数来接触信号的阻塞。访问全局变量时，前后函数调用必须满足前后顺序等情况下，需要阻塞信号。
2. 每一个子进程都应该有一个唯一的进程组ID（pid），用于规避用户使用ctrl-z与ctrl-c等操作时可能会引发的程序错误。

## 3.1.2 int builtin\_cmd(char \*\*argv)函数（5分）

函数功能：识别并解释内置命令: **quit, fg, bg, 和 jobs**

参 数：char \*\*argv

处理流程：依次判断用户输入的指令：

1. quit，调用exit(0)并退出tsh。
2. bg或者fg，调用do\_bgfg(argv)，并返回1，tsh继续运行。
3. jobs，调用listjobs，列出所有的job的信息，并返回1，tsh继续运行。
4. 若以上都不是，则该指令不是内置指令，返回0。

要点分析：

需要分不同的情况讨论输入的指令，针对不同情况的内置指令，我们需要理解这些内置指令的含义，并实现相关操作。

## 3.1.3 void do\_bgfg(char \*\*argv) 函数（5分）

函数功能：实现内置命令bg 和 fg.

参 数：char \*\*argv

处理流程：

1. 对传入的命令行指令进行解析，若为NULL，则输出提示信息，直接返回。对于bg与fg的调用，函数提供了两种可供调用的模式，分别为PID与%%jobid，并分别判断是否有对应的进程（process）或作业（job），若没有，直接返回。若有，则继续下面的bg与fg操作。
2. 对于bg操作，向job所在的进程组发送SIGCONT信号，更改job的state为BG。
3. 对于fg操作，向job所在的进程组发送SIGCONT信号，更改job的state为FG，然后调用waitfg(jobp->pid)等待前台进程结束。
4. 若不是bg或fg，则打印提示信息，并终止tsh。

要点分析：

1. 需要区分输入的指令是bg还是fg，并且判断接下来的指令是PID还是%%jobid，来获取相应的job，并执行bg或者fg的相关操作。
2. 程序对用户的错误输入进行了较为完善的判断。
3. 执行bg与fg等操作时，需要向目标job的进程组发送SIGCONT信号。。
4. 在fg指令中，需要调用waitfg来阻塞tsh进行，用来进行前台的job，直至运行结束。

## 3.1.4 void waitfg(pid\_t pid) 函数（5分）

函数功能：等待一个前台作业结束

参 数：pid\_t pid

处理流程：

进行while循环，每一次循环都sleep 1秒，循环终止条件为前台进程的pid不再是输入的pid。

要点分析：

1. 使用sleep函数来使循环每隔一秒进行一次，检测一次当前前台进程的pid。
2. 使用了 fgpid函数，查询当前的state是FG的job的PID。

## 3.1.5 void sigchld\_handler(int sig) 函数（10分）

函数功能：捕获SIGCHILD信号

参 数：int sig

处理流程：

1. 先设置olderrno = errno，保存原errno的值，然后执行sigfillset(&mask)将所有信号都添加到mask阻塞集合中。
2. 处理所有子进程集合中已经停止或终止的子进程。分为几种情况：
   1. WIFEXITED(status) 该子进程通过调用exit或者一个return正常终止，则阻塞信号，在job列表中删除该job，解除信号阻塞。
   2. WIFSTOPPED(status) 该子进程已经停止，则向屏幕打印信息。
   3. WIFSIGNALED(status) 该子进程是因为一个未被捕获的信号而终止的，则向屏幕打印信息，阻塞信号，从job列表中删除该job，解除信号阻塞。
3. errno = olderrno，恢复原errno

要点分析：

1. 使用while循环，while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)，其中WNOHANG|WUNTRACED的含义为立即返回，waitpid函数调用，若已经没有停止或终止的子进程，则返回值为0，结束循坏。
2. 使用WIFEXITED、WIFSTOPPED、WIFSIGNALED来判断子进程的状态。
3. 由于deletejob()可能修改jobs的值，需要sigprocmask来阻塞信号，防止更改。
4. 由于在信号处理中可能会修改errno的值，所以需要提前保存一个初始errno(olderrno)，最后再恢复errno。

**3.2 程序实现（tsh.c的全部内容）（10分）**

**重点检查代码风格：**

1. **用较好的代码注释说明——5分**
2. **检查每个系统调用的返回值——5分**

|  |
| --- |
| 1. /\* 2. \* tsh - A tiny shell program with job control 3. \* 林逸灏1190202405 4. \* <Put your name and login ID here> 5. \*/ 6. #include <stdio.h> 7. #include <stdlib.h> 8. #include <unistd.h> 9. #include <string.h> 10. #include <ctype.h> 11. #include <signal.h> 12. #include <sys/types.h> 13. #include <sys/wait.h> 14. #include <errno.h> 15. /\* Misc manifest constants \*/ 16. #define MAXLINE 1024 /\* max line size \*/ 17. #define MAXARGS 128 /\* max args on a command line \*/ 18. #define MAXJOBS 16 /\* max jobs at any point in time \*/ 19. #define MAXJID 1<<16 /\* max job ID \*/ 20. /\* Job states \*/ 21. #define UNDEF 0 /\* undefined \*/ 22. #define FG 1 /\* running in foreground \*/ 23. #define BG 2 /\* running in background \*/ 24. #define ST 3 /\* stopped \*/ 25. /\* 26. \* Jobs states: FG (foreground), BG (background), ST (stopped) 27. \* Job state transitions and enabling actions: 28. \* FG -> ST : ctrl-z 29. \* ST -> FG : fg command 30. \* ST -> BG : bg command 31. \* BG -> FG : fg command 32. \* At most 1 job can be in the FG state. 33. \*/ 34. /\* Global variables \*/ 35. extern char \*\*environ; /\* defined in libc \*/ 36. char prompt[] = "tsh> "; /\* command line prompt (DO NOT CHANGE) \*/ 37. int verbose = 0; /\* if true, print additional output \*/ 38. int nextjid = 1; /\* next job ID to allocate \*/ 39. char sbuf[MAXLINE]; /\* for composing sprintf messages \*/ 40. struct job\_t { /\* The job struct \*/ 41. pid\_t pid; /\* job PID \*/ 42. int jid; /\* job ID [1, 2, ...] \*/ 43. int state; /\* UNDEF, BG, FG, or ST \*/ 44. char cmdline[MAXLINE]; /\* command line \*/ 45. }; 46. struct job\_t jobs[MAXJOBS]; /\* The job list \*/ 47. /\* End global variables \*/ 48. /\* Function prototypes \*/ 49. /\* Here are the functions that you will implement \*/ 50. void eval(char \*cmdline); 51. int builtin\_cmd(char \*\*argv); 52. void do\_bgfg(char \*\*argv); 53. void waitfg(pid\_t pid); 54. void sigchld\_handler(int sig); 55. void sigtstp\_handler(int sig); 56. void sigint\_handler(int sig); 57. /\* Here are helper routines that we've provided for you \*/ 58. int parseline(const char \*cmdline, char \*\*argv); 59. void sigquit\_handler(int sig); 60. void clearjob(struct job\_t \*job); 61. void initjobs(struct job\_t \*jobs); 62. int maxjid(struct job\_t \*jobs); 63. int addjob(struct job\_t \*jobs, pid\_t pid, int state, char \*cmdline); 64. int deletejob(struct job\_t \*jobs, pid\_t pid); 65. pid\_t fgpid(struct job\_t \*jobs); 66. struct job\_t \*getjobpid(struct job\_t \*jobs, pid\_t pid); 67. struct job\_t \*getjobjid(struct job\_t \*jobs, int jid); 68. int pid2jid(pid\_t pid); 69. void listjobs(struct job\_t \*jobs); 70. void usage(void); 71. void unix\_error(char \*msg); 72. void app\_error(char \*msg); 73. typedef void handler\_t(int); 74. handler\_t \*Signal(int signum, handler\_t \*handler); 75. /\* 76. \* main - The shell's main routine 77. \*/ 78. int main(int argc, char \*\*argv) 79. { 80. char c; 81. char cmdline[MAXLINE]; 82. int emit\_prompt = 1; /\* emit prompt (default) \*/ 83. /\* Redirect stderr to stdout (so that driver will get all output 84. \* on the pipe connected to stdout) \*/ 85. dup2(1, 2); 86. /\* Parse the command line \*/ 87. while ((c = getopt(argc, argv, "hvp")) != EOF) { 88. switch (c) { 89. case 'h': /\* print help message \*/ 90. usage(); 91. break; 92. case 'v': /\* emit additional diagnostic info \*/ 93. verbose = 1; 94. break; 95. case 'p': /\* don't print a prompt \*/ 96. emit\_prompt = 0; /\* handy for automatic testing \*/ 97. break; 98. default: 99. usage(); 100. } 101. } 102. /\* Install the signal handlers \*/ 103. /\* These are the ones you will need to implement \*/ 104. Signal(SIGINT, sigint\_handler); /\* ctrl-c \*/ 105. Signal(SIGTSTP, sigtstp\_handler); /\* ctrl-z \*/ 106. Signal(SIGCHLD, sigchld\_handler); /\* Terminated or stopped child \*/ 107. /\* This one provides a clean way to kill the shell \*/ 108. Signal(SIGQUIT, sigquit\_handler); 109. /\* Initialize the job list \*/ 110. initjobs(jobs); 111. /\* Execute the shell's read/eval loop \*/ 112. while (1) { 113. /\* Read command line \*/ 114. if (emit\_prompt) { 115. printf("%s", prompt); 116. fflush(stdout); 117. } 118. if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin)) 119. app\_error("fgets error"); 120. if (feof(stdin)) { /\* End of file (ctrl-d) \*/ 121. fflush(stdout); 122. exit(0); 123. } 124. /\* Evaluate the command line \*/ 125. eval(cmdline); 126. fflush(stdout); 127. fflush(stdout); 128. } 129. exit(0); /\* control never reaches here \*/ 130. } 132. /\* 133. \* eval - Evaluate the command line that the user has just typed in 134. \* 135. \* If the user has requested a built-in command (quit, jobs, bg or fg) 136. \* then execute it immediately. Otherwise, fork a child process and 137. \* run the job in the context of the child. If the job is running in 138. \* the foreground, wait for it to terminate and then return. Note: 139. \* each child process must have a unique process group ID so that our 140. \* background children don't receive SIGINT (SIGTSTP) from the kernel 141. \* when we type ctrl-c (ctrl-z) at the keyboard. 142. \*/ 143. void eval(char \*cmdline) 144. { 145. /\* $begin handout \*/ 146. char \*argv[MAXARGS]; /\* argv for execve() \*/ 147. int bg; /\* should the job run in bg or fg? \*/ 148. pid\_t pid; /\* process id \*/ 149. sigset\_t mask; /\* signal mask \*/ 150. /\* Parse command line \*/ 151. bg = parseline(cmdline, argv); 152. if (argv[0] == NULL) 153. return; /\* ignore empty lines \*/ 154. if (!builtin\_cmd(argv)) { 155. /\* 156. \* This is a little tricky. Block SIGCHLD, SIGINT, and SIGTSTP 157. \* signals until we can add the job to the job list. This 158. \* eliminates some nasty races between adding a job to the job 159. \* list and the arrival of SIGCHLD, SIGINT, and SIGTSTP signals. 160. \*/ 161. if (sigemptyset(&mask) < 0) 162. unix\_error("sigemptyset error"); 163. if (sigaddset(&mask, SIGCHLD)) 164. unix\_error("sigaddset error"); 165. if (sigaddset(&mask, SIGINT)) 166. unix\_error("sigaddset error"); 167. if (sigaddset(&mask, SIGTSTP)) 168. unix\_error("sigaddset error"); 169. if (sigprocmask(SIG\_BLOCK, &mask, NULL) < 0) 170. unix\_error("sigprocmask error"); 171. /\* Create a child process \*/ 172. if ((pid = fork()) < 0) 173. unix\_error("fork error"); 175. /\* 176. \* Child process 177. \*/ 178. if (pid == 0) { 179. /\* Child unblocks signals \*/ 180. sigprocmask(SIG\_UNBLOCK, &mask, NULL); 181. /\* Each new job must get a new process group ID 182. so that the kernel doesn't send ctrl-c and ctrl-z 183. signals to all of the shell's jobs \*/ 184. if (setpgid(0, 0) < 0) 185. unix\_error("setpgid error"); 186. /\* Now load and run the program in the new job \*/ 187. if (execve(argv[0], argv, environ) < 0) { 188. printf("%s: Command not found\n", argv[0]); 189. exit(0); 190. } 191. } 192. /\* 193. \* Parent process 194. \*/ 195. /\* Parent adds the job, and then unblocks signals so that 196. the signals handlers can run again \*/ 197. addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline); 198. sigprocmask(SIG\_UNBLOCK, &mask, NULL); 199. if (!bg) 200. waitfg(pid); 201. else 202. printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline); 203. } 204. /\* $end handout \*/ 205. return; 206. } 207. /\* 208. \* parseline - Parse the command line and build the argv array. 209. \* 210. \* Characters enclosed in single quotes are treated as a single 211. \* argument. Return true if the user has requested a BG job, false if 212. \* the user has requested a FG job. 213. \*/ 214. int parseline(const char \*cmdline, char \*\*argv) 215. { 216. static char array[MAXLINE]; /\* holds local copy of command line \*/ 217. char \*buf = array; /\* ptr that traverses command line \*/ 218. char \*delim; /\* points to first space delimiter \*/ 219. int argc; /\* number of args \*/ 220. int bg; /\* background job? \*/ 221. strcpy(buf, cmdline); 222. buf[strlen(buf)-1] = ' '; /\* replace trailing '\n' with space \*/ 223. while (\*buf && (\*buf == ' ')) /\* ignore leading spaces \*/ 224. buf++; 225. /\* Build the argv list \*/ 226. argc = 0; 227. if (\*buf == '\'') { 228. buf++; 229. delim = strchr(buf, '\''); 230. } 231. else { 232. delim = strchr(buf, ' '); 233. } 234. while (delim) { 235. argv[argc++] = buf; 236. \*delim = '\0'; 237. buf = delim + 1; 238. while (\*buf && (\*buf == ' ')) /\* ignore spaces \*/ 239. buf++; 240. if (\*buf == '\'') { 241. buf++; 242. delim = strchr(buf, '\''); 243. } 244. else { 245. delim = strchr(buf, ' '); 246. } 247. } 248. argv[argc] = NULL; 250. if (argc == 0) /\* ignore blank line \*/ 251. return 1; 252. /\* should the job run in the background? \*/ 253. if ((bg = (\*argv[argc-1] == '&')) != 0) { 254. argv[--argc] = NULL; 255. } 256. return bg; 257. } 258. /\* 259. \* builtin\_cmd - If the user has typed a built-in command then execute 260. \* it immediately. 261. \*/ 262. int builtin\_cmd(char \*\*argv) 263. { 264. if(!strcmp(argv[0],"quit")){ //内置命令quit，退出tsh 265. exit(0); 266. } 267. if(!strcmp(argv[0],"&")){//忽略单独的&，返回1 268. return 1; 269. } 270. if(!strcmp(argv[0],"bg") || (!strcmp(argv[0],"fg"))) { //内置命令fg或bg，调用do\_fgbg，返回1 271. do\_bgfg(argv); 272. return 1; 273. } 274. if(!strcmp(argv[0],"jobs")) { //内置命令jobs，列出所有任务，调用 listjobs 即可，返回1 275. listjobs(jobs); 276. return 1; 277. } 278. return 0; /\* not a builtin command \*/ 279. } 280. /\* 281. \* do\_bgfg - Execute the builtin bg and fg commands 282. \*/ 283. void do\_bgfg(char \*\*argv) 284. { 285. /\* $begin handout \*/ 286. struct job\_t \*jobp=NULL; 288. /\* Ignore command if no argument \*/ 289. if (argv[1] == NULL) { 290. printf("%s command requires PID or %%jobid argument\n", argv[0]); 291. return; 292. } 294. /\* Parse the required PID or %JID arg \*/ 295. if (isdigit(argv[1][0])) { 296. pid\_t pid = atoi(argv[1]); 297. if (!(jobp = getjobpid(jobs, pid))) { 298. printf("(%d): No such process\n", pid); 299. return; 300. } 301. } 302. else if (argv[1][0] == '%') { 303. int jid = atoi(&argv[1][1]); 304. if (!(jobp = getjobjid(jobs, jid))) { 305. printf("%s: No such job\n", argv[1]); 306. return; 307. } 308. } 309. else { 310. printf("%s: argument must be a PID or %%jobid\n", argv[0]); 311. return; 312. } 313. /\* bg command \*/ 314. if (!strcmp(argv[0], "bg")) { 315. if (kill(-(jobp->pid), SIGCONT) < 0) 316. unix\_error("kill (bg) error"); 317. jobp->state = BG; 318. printf("[%d] (%d) %s", jobp->jid, jobp->pid, jobp->cmdline); 319. } 320. /\* fg command \*/ 321. else if (!strcmp(argv[0], "fg")) { 322. if (kill(-(jobp->pid), SIGCONT) < 0) 323. unix\_error("kill (fg) error"); 324. jobp->state = FG; 325. waitfg(jobp->pid); 326. } 327. else { 328. printf("do\_bgfg: Internal error\n"); 329. exit(0); 330. } 331. /\* $end handout \*/ 332. return; 333. } 334. /\* 335. \* waitfg - Block until process pid is no longer the foreground process 336. \*/ 337. void waitfg(pid\_t pid) 338. { 339. while (fgpid(jobs) == pid) { //判断前台进程的pid是否为输入的pid 340. sleep(1); //睡眠一秒 341. } 342. return; 343. } 344. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 345. \* Signal handlers 346. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 347. /\* 348. \* sigchld\_handler - The kernel sends a SIGCHLD to the shell whenever 349. \* a child job terminates (becomes a zombie), or stops because it 350. \* received a SIGSTOP or SIGTSTP signal. The handler reaps all 351. \* available zombie children, but doesn't wait for any other 352. \* currently running children to terminate. 353. \*/ 354. void sigchld\_handler(int sig) 355. { 356. int olderrno = errno; //记录下errno 防止在信号处理过程中 调用函数 出错 产生errno变化 357. int status; 358. sigset\_t mask\_all, prev\_all; 359. pid\_t pid; 361. sigfillset(&mask\_all); 362. while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) { //while循环 回收所有僵死进程 364. //对导致退出的情况进行屏幕回显 365. // WHOANG|WUNTRACED 代表立即返回，如果等待集合中的子进程都没有停止或终止则返回为0，如果有一个，则返回他的PID 367. if (WIFEXITED(status)) { //子进程通过调用exit或者一个返回正常终止 368. sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all); //阻塞所有的信号 369. deletejob(jobs, pid); //在任务列表中删除任务 370. sigprocmask(SIG\_SETMASK, &prev\_all, NULL); //解除阻塞 371. }   375. if (WIFSTOPPED(status)) { //引起返回的子进程当前是停止的 376. printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status)); 377. } 379. if (WIFSIGNALED(status)) { //子进程是因为一个未被捕获的信号终止的 380. printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status)); 381. sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all); 382. deletejob(jobs, pid); 383. sigprocmask(SIG\_SETMASK, &prev\_all, NULL); 384. } 386. } 387. errno = olderrno; 388. return; 389. } 390. /\* 391. \* sigint\_handler - The kernel sends a SIGINT to the shell whenver the 392. \* user types ctrl-c at the keyboard. Catch it and send it along 393. \* to the foreground job. 394. \*/ 395. void sigint\_handler(int sig) 396. { 397. int olderrno = errno; //保存errno 398. pid\_t pid = fgpid(jobs); 399. if (pid == 0) { 400. return; 401. } 403. kill(-pid, SIGINT); //向处于前台正在运行的进程的进程组发送SIGINT 404. errno = olderrno; 405. return; 406. } 407. /\* 408. \* sigtstp\_handler - The kernel sends a SIGTSTP to the shell whenever 409. \* the user types ctrl-z at the keyboard. Catch it and suspend the 410. \* foreground job by sending it a SIGTSTP. 411. \*/ 412. void sigtstp\_handler(int sig) 413. { 414. pid\_t pid = fgpid(jobs); 415. if(pid == 0) { 416. return ; 417. } 418. getjobpid(jobs, pid)->state = ST; //设置state为ST 419. kill(-pid,SIGTSTP); 420. printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, sig); 422. return; 423. } 424. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 425. \* End signal handlers 426. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 427. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 428. \* Helper routines that manipulate the job list 429. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 430. /\* clearjob - Clear the entries in a job struct \*/ 431. void clearjob(struct job\_t \*job) { 432. job->pid = 0; 433. job->jid = 0; 434. job->state = UNDEF; 435. job->cmdline[0] = '\0'; 436. } 437. /\* initjobs - Initialize the job list \*/ 438. void initjobs(struct job\_t \*jobs) { 439. int i; 440. for (i = 0; i < MAXJOBS; i++) 441. clearjob(&jobs[i]); 442. } 443. /\* maxjid - Returns largest allocated job ID \*/ 444. int maxjid(struct job\_t \*jobs) 445. { 446. int i, max=0; 447. for (i = 0; i < MAXJOBS; i++) 448. if (jobs[i].jid > max) 449. max = jobs[i].jid; 450. return max; 451. } 452. /\* addjob - Add a job to the job list \*/ 453. int addjob(struct job\_t \*jobs, pid\_t pid, int state, char \*cmdline) 454. { 455. int i; 457. if (pid < 1) 458. return 0; 459. for (i = 0; i < MAXJOBS; i++) { 460. if (jobs[i].pid == 0) { 461. jobs[i].pid = pid; 462. jobs[i].state = state; 463. jobs[i].jid = nextjid++; 464. if (nextjid > MAXJOBS) 465. nextjid = 1; 466. strcpy(jobs[i].cmdline, cmdline); 467. if(verbose){ 468. printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid, jobs[i].cmdline); 469. } 470. return 1; 471. } 472. } 473. printf("Tried to create too many jobs\n"); 474. return 0; 475. } 476. /\* deletejob - Delete a job whose PID=pid from the job list \*/ 477. int deletejob(struct job\_t \*jobs, pid\_t pid) 478. { 479. int i; 480. if (pid < 1) 481. return 0; 482. for (i = 0; i < MAXJOBS; i++) { 483. if (jobs[i].pid == pid) { 484. clearjob(&jobs[i]); 485. nextjid = maxjid(jobs)+1; 486. return 1; 487. } 488. } 489. return 0; 490. } 491. /\* fgpid - Return PID of current foreground job, 0 if no such job \*/ 492. pid\_t fgpid(struct job\_t \*jobs) { 493. int i; 494. for (i = 0; i < MAXJOBS; i++) 495. if (jobs[i].state == FG) 496. return jobs[i].pid; 497. return 0; 498. } 499. /\* getjobpid - Find a job (by PID) on the job list \*/ 500. struct job\_t \*getjobpid(struct job\_t \*jobs, pid\_t pid) { 501. int i; 502. if (pid < 1) 503. return NULL; 504. for (i = 0; i < MAXJOBS; i++) 505. if (jobs[i].pid == pid) 506. return &jobs[i]; 507. return NULL; 508. } 509. /\* getjobjid - Find a job (by JID) on the job list \*/ 510. struct job\_t \*getjobjid(struct job\_t \*jobs, int jid) 511. { 512. int i; 513. if (jid < 1) 514. return NULL; 515. for (i = 0; i < MAXJOBS; i++) 516. if (jobs[i].jid == jid) 517. return &jobs[i]; 518. return NULL; 519. } 520. /\* pid2jid - Map process ID to job ID \*/ 521. int pid2jid(pid\_t pid) 522. { 523. int i; 524. if (pid < 1) 525. return 0; 526. for (i = 0; i < MAXJOBS; i++) 527. if (jobs[i].pid == pid) { 528. return jobs[i].jid; 529. } 530. return 0; 531. } 532. /\* listjobs - Print the job list \*/ 533. void listjobs(struct job\_t \*jobs) 534. { 535. int i; 537. for (i = 0; i < MAXJOBS; i++) { 538. if (jobs[i].pid != 0) { 539. printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid); 540. switch (jobs[i].state) { 541. case BG: 542. printf("Running "); 543. break; 544. case FG: 545. printf("Foreground "); 546. break; 547. case ST: 548. printf("Stopped "); 549. break; 550. default: 551. printf("listjobs: Internal error: job[%d].state=%d ", 552. i, jobs[i].state); 553. } 554. printf("%s", jobs[i].cmdline); 555. } 556. } 557. } 558. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 559. \* end job list helper routines 560. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 561. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 562. \* Other helper routines 563. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 564. /\* 565. \* usage - print a help message 566. \*/ 567. void usage(void) 568. { 569. printf("Usage: shell [-hvp]\n"); 570. printf(" -h print this message\n"); 571. printf(" -v print additional diagnostic information\n"); 572. printf(" -p do not emit a command prompt\n"); 573. exit(1); 574. } 575. /\* 576. \* unix\_error - unix-style error routine 577. \*/ 578. void unix\_error(char \*msg) 579. { 580. fprintf(stdout, "%s: %s\n", msg, strerror(errno)); 581. exit(1); 582. } 583. /\* 584. \* app\_error - application-style error routine 585. \*/ 586. void app\_error(char \*msg) 587. { 588. fprintf(stdout, "%s\n", msg); 589. exit(1); 590. } 591. /\* 592. \* Signal - wrapper for the sigaction function 593. \*/ 594. handler\_t \*Signal(int signum, handler\_t \*handler) 595. { 596. struct sigaction action, old\_action; 597. action.sa\_handler = handler; 598. sigemptyset(&action.sa\_mask); /\* block sigs of type being handled \*/ 599. action.sa\_flags = SA\_RESTART; /\* restart syscalls if possible \*/ 600. if (sigaction(signum, &action, &old\_action) < 0) 601. unix\_error("Signal error"); 602. return (old\_action.sa\_handler); 603. } 604. /\* 605. \* sigquit\_handler - The driver program can gracefully terminate the 606. \* child shell by sending it a SIGQUIT signal. 607. \*/ 608. void sigquit\_handler(int sig) 609. { 610. printf("Terminating after receipt of SIGQUIT signal\n"); 611. exit(1); 612. } |

# 第4章 TinyShell测试

**总分15分**

## 4.1 测试方法

针对tsh和参考shell程序tshref，完成测试项目4.1-4.15的对比测试，并将测试结果截图或者通过重定向保存到文本文件(例如：./sdriver.pl -t trace01.txt -s ./tsh -a "-p" > tshresult01.txt)，并填写完成4.3节的相应表格。

## 4.2 测试结果评价

tsh与tshref的输出在以下两个方面可以不同：

（1）pid

（2）测试文件trace11.txt, trace12.txt和trace13.txt中的/bin/ps命令，每次运行的输出都会不同，但每个mysplit进程的运行状态应该相同。

除了上述两方面允许的差异，tsh与tshref的输出相同则判为正确，如不同则给出原因分析。

## 4.3 自测试结果

填写以下各个测试用例的测试结果，每个测试用例1分。

### 4.3.1测试用例trace01.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.2测试用例trace02.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.3测试用例trace03.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.4测试用例trace04.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.5测试用例trace05.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.6测试用例trace06.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.7测试用例trace07.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.8测试用例trace08.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.9测试用例trace09.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.10测试用例trace10.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.11测试用例trace11.txt

测试中ps指令的输出内容较多，仅记录和本实验密切相关的tsh、mysplit等进程的部分信息即可。

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.12测试用例trace12.txt

测试中ps指令的输出内容较多，仅记录和本实验密切相关的tsh、mysplit等进程的部分信息即可。

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.13测试用例trace13.txt

测试中ps指令的输出内容较多，仅记录和本实验密切相关的tsh、mysplit等进程的部分信息即可。

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.14测试用例trace14.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

### 4.3.15测试用例trace15.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同/不同，原因分析如下：相同 | |

# 第5章 评测得分

**总分20分**

实验程序统一测试的评分（教师评价）：

（1）正确性得分： （满分10）

（2）性能加权得分： （满分10）

# 第6章 总结

## 5.1 请总结本次实验的收获

1.更加深入理解了信号的处理过程，了解了信号的阻塞与恢复。

2.深入理解了shell的原理，并能实现一个微型的shell。

## 5.2 请给出对本次实验内容的建议

无

注：本章为酌情加分项。

# 参考文献

**为完成本次实验你翻阅的书籍与网站等**

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/ collection/anatmorp.