

**实验报告**

**实 验（七）**

题 目 TinyShell

微壳

专 业 计算机专业

学　　 号 1190200501

班　　 级 1903002

学 生 林燕燕

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2021.06.04

**计算机科学与技术学院**

**目 录**

[第1章 实验基本信息 - 4 -](#_Toc26657523)

**!异常的公式结尾**

[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） - 8 -](#_Toc26657533)

[2.4 什么是shell，功能和处理流程（5分） - 9 -](#_Toc26657534)

[第3章 TinyShell的设计与实现 - 10 -](#_Toc26657535)

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

[3. 1.2 int builtin\_cmd(char \*\*argv)函数（5分） - 10 -](#_Toc26657537)

[3. 1.3 void do\_bgfg(char \*\*argv) 函数（5分） - 11 -](#_Toc26657538)

[3. 1.4 void waitfg(pid\_t pid) 函数（5分） - 11 -](#_Toc26657539)

[3. 1.5 void sigchld\_handler(int sig) 函数（10分） - 12 -](#_Toc26657540)

[第4章 TinyShell测试 - 28 -](#_Toc26657541)

[4.1 测试方法 - 28 -](#_Toc26657542)

[4.2 测试结果评价 - 28 -](#_Toc26657543)

[4.3 自测试结果 - 28 -](#_Toc26657544)

[4.3.1测试用例trace01.txt - 28 -](#_Toc26657545)

[4.3.2测试用例trace02.txt - 29 -](#_Toc26657546)

[4.3.3测试用例trace03.txt - 29 -](#_Toc26657547)

[4.3.4测试用例trace04.txt - 29 -](#_Toc26657548)

[4.3.5测试用例trace05.txt - 30 -](#_Toc26657549)

[4.3.6测试用例trace06.txt - 30 -](#_Toc26657550)

[4.3.7测试用例trace07.txt - 30 -](#_Toc26657551)

[4.3.8测试用例trace08.txt - 31 -](#_Toc26657552)

[4.3.9测试用例trace09.txt - 31 -](#_Toc26657553)

[4.3.10测试用例trace10.txt - 32 -](#_Toc26657554)

[4.3.11测试用例trace11.txt - 32 -](#_Toc26657555)

[4.3.12测试用例trace12.txt - 33 -](#_Toc26657556)

[4.3.13测试用例trace13.txt - 33 -](#_Toc26657557)

[4.3.14测试用例trace14.txt - 35 -](#_Toc26657558)

[4.3.15测试用例trace15.txt - 35 -](#_Toc26657559)

[4.4 自测试评分 **错误!未定义书签。**](#_Toc26657560)

[第5章 总结 - 37 -](#_Toc26657561)

[5.1 请总结本次实验的收获 - 37 -](#_Toc26657562)

[5.2 请给出对本次实验内容的建议 - 37 -](#_Toc26657563)

[参考文献 - 38 -](#_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；1.6GHz；8G RAM；256G SSD Disk；1T HDD Disk

### 1.2.2 软件环境

Windows10 64位；Vmware 14pro；Ubuntu 20.04.2 LTS 64位

### 1.2.3 开发工具

Visual Studio Code 64位；vim/gpedit+gcc

## 1.3 实验预习

* Kill命令

kill –l：列出信号

kill –SIGKILL 17130： 杀死pid为17130的进程

kill -9 17130 ：杀死pid为17130的进程，或者：

kill -9 -17130：杀死进程组17130中的每个进程

killall -9 pname： 杀死名字为pname的进程

* 进程状态

D 不可中断睡眠 (通常是在IO操作) 收到信号不唤醒和不可运行, 进程必须等待直到有中断发生

R 正在运行或可运行（在运行队列排队中）

S 可中断睡眠 (休眠中, 受阻, 在等待某个条件的形成或接受到信号)

T 已停止的 进程收到SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU信号后停止运行

W 正在换页(2.6.内核之前有效)

X 死进程 (未开启)

Z 僵尸进程a defunct (”zombie”) process

< 高优先级(not nice to other users)

N 低优先级(nice to other users)

L 页面锁定在内存（实时和定制的IO）

s 一个信息头

l 多线程（使用 CLONE\_THREAD，像NPTL的pthreads的那样）

+ 在前台进程组

* ps t /ps aux /ps

t <终端机编号n> 　列终端n的程序的状况。

a 显示现行终端机下的所有程序，包括其他用户的程序。

u 以用户为主的格式来显示程序状况。

x 显示所有程序，不以终端来区分。

* 作业 ：jobs、 fg %n 、 bg%n

jobs 显示当前暂停的进程

bg %n 使第n个任务在后台运行(%前有空格)

fg %n 使第n个任务在前台运行

bg, fg 不带%n 表示对最后一个进程操作

ctrl+c: 终止前台作业(进程组的每个进程)

ctrl+z: 停止前台作业(进程组的每个进程)，随后可用bg恢复后台运行， fg 恢复前台运行。

# 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）

1. 概念：

是程序的依次运行过程，是具有独立功能的一个程序关于某个数据集合的依次运行活动，进而进程具有动态含义。同一个程序处理不同的数据就是不同的进程。

2. 创建方法：

父进程通过调用 fork 函数创建一个新的运行的子进程，子进程中，fork 返回 0；父进程中，返回子进程的 PID；新创建的子进程几乎但不完全与父进程相同，子进程得到与父进程虚拟地址空间相同的但是独立的一份副本，子进程获得与父进程任何打开文件描述符相同的副本，最大区别是子进程有不同于父进程的 PID。

3. 回收进程：

当进程终止时，它仍然消耗系统资源，被称为“僵尸 zombie”进程。父进程通过 wait 函数回收子进程，挂起当前进程的执行直到它的一个子进程终止，返回已终止子进程的 PID。

## 2.2信号的机制、种类（5分）

1. 信号的机制：

信号通知进程系统中发生了异步事件

发送信号：内核通过更新目的进程上下文中的某个状态，发送一个信号给目的进程；

接受信号：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。

发送信号的原因：内核检测到一个系统事件如除零错误(SIGFPE)或者子进程终止(SIGCHLD) ；一个进程调用了kill系统调用，显式地请求内核发送一个信号到目的进程反应的方式：

忽略这个信号(do nothing)；终止进程(with optional core dump)；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

2. 信号种类：

|  |  |  |  |
| --- | --- | --- | --- |
| **编号** | **名称** | **缺省动作** | **说明** |
| 1 | SIGHUP | 终止 | 终止控制终端或进程 |
| 2 | SIGINT | 终止 | 键盘产生的中断(Ctrl-C) |
| 3 | SIGQUIT | 终止 | 键盘产生的退出 |
| 4 | SIGILL | 终止 | 非法指令 |
| 5 | SIGTRAP | 终止并转储内存 | debug中断 |
| 6 | SIGABRT | 终止并转储内存 | 异常中止 |
| 7 | SIGBUS | 终止 | 总线异常/EMT指令 |
| 8 | SIGFPE | 终止并转储内存 | 浮点运算溢出 |
| 9 | SIGKILL | 终止 | 强制进程终止 |
| 10 | SIGUSR1 | 终止 | 用户信号,进程可自定义用途 |
| 11 | SIGSEGV | 终止并转储内存 | 非法内存地址引用 |
| 12 | SIGUSR2 | 终止 | 用户信号，进程可自定义用途 |
| 13 | SIGPIPE | 终止 | 向某个没有读取的管道中写入数据 |
| 14 | SIGALRM | 终止 | 时钟中断(闹钟) |
| 15 | SIGTERM | 终止 | 进程终止 |
| 16 | SIGSTKFLT | 终止 | 协处理器栈错误 |
| 17 | SIGCHLD | 忽略 | 子进程退出或中断 |
| 18 | SIGCONT | 忽略 | 如进程停止状态则开始运行 |
| 19 | SIGSTOP | 停止直到下一个SIGCONT | 停止进程运行 |
| 20 | SIGSTP | 停止直到下一个SIGCONT | 键盘产生的停止 |
| 21 | SIGTTIN | 停止直到下一个SIGCONT | 后台进程请求输入 |
| 22 | SIGTTOU | 停止直到下一个SIGCONT | 后台进程请求输出 |
| 23 | SIGURG | 忽略 | socket发生紧急情况 |
| 24 | SIGXCPU | 终止 | CPU时间限制被打破 |
| 25 | SIGXFSZ | 终止 | 文件大小限制被打破 |
| 26 | SIGVTALRM | 终止 | 虚拟定时时钟 |
| 27 | SIGPROF | 终止 | profile timer clock |
| 28 | SIGWINCH | 忽略 | 窗口尺寸调整 |
| 29 | SIGIO | 终止 | I/O可用 |
| 30 | SIGPWR | 终止 | 电源异常 |

## 2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5分）

1. 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.

发送方法：

(1)用/bin/kill程序可以向另外的进程或进程组发送任意的信号

(2)从键盘发送信号输入 ctrl-c (ctrl-z) 会导致内核发送一个 SIGINT (SIGTSTP)信号到前台进程组中的每个作业

(3)发送信号的函数主要有kill(),raise(),alarm(),pause()

2. 阻塞信号

隐式阻塞机制 ：内核默认阻塞与当前正在处理信号类型相同的待处理信号 如： 一个SIGINT 信号处理程序不能被另一个 SIGINT信号中断 （此时另一个SIGINT信号被阻塞）

显示阻塞和解除阻塞机制 ：sigprocmask 函数及其辅助函数可以明确地阻塞/解除阻塞

3. 设置信号处理程序

 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)

## 2.4 什么是shell，功能和处理流程（5分）

1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](https://baike.baidu.com/item/%E7%94%A8%E6%88%B7%E7%95%8C%E9%9D%A2)，提供了用户与[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8)进行交互操作的一种[接口](https://baike.baidu.com/item/%E6%8E%A5%E5%8F%A3)。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 也是一支程序，它由[输入设备](https://baike.baidu.com/item/%E8%BE%93%E5%85%A5%E8%AE%BE%E5%A4%87)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）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：解析和解释命令行的主例程。

参 数：char \*cmdline

处理流程：

1. 定义argv[MAXARGS]，bg，pid，mask这些变量。
2. 调用parseline()函数解析命令参数。
3. eval调用builtin\_cmd()函数，判断命令行是否是内置命令
   1. 如果是内置命令，则在builtin\_cmd()函数处理；
   2. 如果不是内置命令，则：
      1. 将SIGCHLD , SIGINT ,SIGTSTP 信号加入阻塞集合。
      2. 创建一个新的进程
      3. 将子进程添加到jobs里，接触阻塞信号。若该命令是前台进程，调用waitfg函数；若是后台进程，打印当前进程信息
      4. 在子进程中解除阻塞信号，获取新的组ID，调用execve。

要点分析：

1. 所有的子进程必须有自己的唯一进程组ID，否则我们在键盘上输入ctrl-c(ctrl-z)时，后台的子进程就会从内核接收SIGINGT(SIGTSTP)，为了避免这种错误，所以每个子进程必须有唯一的进程组ID。

2. 不是内置命令时需要设置阻塞信号，让信号可以被发送但不会被接收，这样就能够避免把不存在的子进程添加到了作业列表中。

## 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：

1. 构建函数判断传入的参数数组argv中的实参argv[0]是否为内置命令，只需要比较实参与内置命令即可。
2. 如果为quit命令，调用exit
3. 如果为jobs命令，调用listjobs
4. 如果为bg或fg命令，调用do\_ bgfg
5. 如果不是内置命令返回0。

要点分析：

如果是内置命令，在执行完成后要return 1，返回输入的命令是内置命令。

## 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. 判断bg或fg后面是否有参数，如果没有参数，忽略命令，返回。
2. 判断bg或fg后面第一位是数字还是%
   1. 若是数字，则说明可能是PID，查找对应jobID，查找失败则返回；
   2. 若是%，则可能是jobID，查找joblist中是否有当前jobID，查找失败则返回；
   3. 若都不是，则返回；
3. 判断命令是bg还是fg
   1. 若是bg，则执行kill发送SIGCONT给进程，并且将job状态设为 BG；
   2. 若是fg，则执行kill发送SIGCONT给进程，并将job状态设为 FG，然后调用waitfg等待前台进程结束；

要点分析：

获取进程号PID或者工作组号JID的方式是通过判断fg和bg后面是数字还是%后面加数字的形式，然后根据进程号或工作组号来获取结构体job，分别在前台和后台执行相关操作。

## 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，每次循环调用sleep函数，while的终止条件是前台程序的PID不再是pid

要点分析：调用fgpid函数获取前台进程的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. 设置ermo并处理所有执行程序和中已经停止或终止的子进程；
2. 如果子进程通过调用exit或者一个返回正常终止，则阻塞信号，删除job，恢复信号。
3. 如果该子进程当前已停止，向屏幕打印信息。
4. 如果该子进程是因为一个未被捕获的信号终止的，则向屏幕打印信息，阻塞信号，删除job，恢复信号。
5. 恢复ermo

要点分析：

由于deletejob()函数会对全局变量jobs的值进行更改，所以需要在deletejob之前阻塞信号。

**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**

/\*

*\* tsh - A tiny shell program with job control*

*\**

*\* <Put your name and login ID here>*

\*/

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <unistd.h>

**#include** <string.h>

**#include** <ctype.h>

**#include** <signal.h>

**#include** <sys/types.h>

**#include** <sys/wait.h>

**#include** <errno.h>

/\**Misc manifest constants*\*/

**#define** **MAXLINE**    1024   /\**max line size*\*/

**#define** **MAXARGS**     128   /\**max args on a command line*\*/

**#define** **MAXJOBS**      16   /\**max jobs at any point in time*\*/

**#define** **MAXJID**    1<<16   /\**max job ID*\*/

/\**Job states*\*/

**#define** **UNDEF** 0 /\**undefined*\*/

**#define** **FG** 1    /\**running in foreground*\*/

**#define** **BG** 2    /\**running in background*\*/

**#define** **ST** 3    /\**stopped*\*/

/\*

*\* Jobs states: FG (foreground), BG (background), ST (stopped)*

*\* Job state transitions and enabling actions:*

*\*     FG -> ST  : ctrl-z*

*\*     ST -> FG  : fg command*

*\*     ST -> BG  : bg command*

*\*     BG -> FG  : fg command*

*\* At most 1 job can be in the FG state.*

\*/

/\**Global variables*\*/

**extern** char \*\***environ**;      /\**defined in libc*\*/

char prompt**[]** = "tsh> ";    /\**command line prompt (DO NOT CHANGE)*\*/

int verbose = 0;            /\**if true, print additional output*\*/

int nextjid = 1;            /\**next job ID to allocate*\*/

char sbuf[**MAXLINE**];         /\**for composing sprintf messages*\*/

**struct** job\_t {              /\**The job struct*\*/

    pid\_t pid;              /\**job PID*\*/

    int jid;                /\**job ID [1, 2, ...]*\*/

    int state;              /\**UNDEF, BG, FG, or ST*\*/

    char cmdline[**MAXLINE**];  /\**command line*\*/

};

**struct** job\_t jobs[**MAXJOBS**]; /\**The job list*\*/

/\**End global variables*\*/

/\**Function prototypes*\*/

/\**Here are the functions that you will implement*\*/

void **eval**(char \*cmdline);

int **builtin\_cmd**(char \*\*argv);

void **do\_bgfg**(char \*\*argv);

void **waitfg**(pid\_t pid);

void **sigchld\_handler**(int sig);

void **sigtstp\_handler**(int sig);

void **sigint\_handler**(int sig);

/\**Here are helper routines that we've provided for you*\*/

int **parseline**(**const** char \*cmdline, char \*\*argv);

void **sigquit\_handler**(int sig);

void **clearjob**(**struct** job\_t \*job);

void **initjobs**(**struct** job\_t \*jobs);

int **maxjid**(**struct** job\_t \*jobs);

int **addjob**(**struct** job\_t \*jobs, pid\_t pid, int state, char \*cmdline);

int **deletejob**(**struct** job\_t \*jobs, pid\_t pid);

pid\_t **fgpid**(**struct** job\_t \*jobs);

**struct** job\_t \***getjobpid**(**struct** job\_t \*jobs, pid\_t pid);

**struct** job\_t \***getjobjid**(**struct** job\_t \*jobs, int jid);

int **pid2jid**(pid\_t pid);

void **listjobs**(**struct** job\_t \*jobs);

void **usage**(void);

void **unix\_error**(char \*msg);

void **app\_error**(char \*msg);

**typedef** void handler\_t(int);

handler\_t \***Signal**(int signum, handler\_t \*handler);

/\*

*\* main - The shell's main routine*

\*/

int **main**(int argc, char \*\*argv)

{

    char c;

    char cmdline[**MAXLINE**];

    int emit\_prompt = 1; /\**emit prompt (default)*\*/

    /\**Redirect stderr to stdout (so that driver will get all output*

*\* on the pipe connected to stdout)*\*/

**dup2**(1, 2);

    /\**Parse the command line*\*/

**while** ((c = **getopt**(argc, argv, "hvp")) **!=** **EOF**) {

**switch** (c) {

**case** 'h':             /\**print help message*\*/

**usage**();

**break**;

**case** 'v':             /\**emit additional diagnostic info*\*/

            verbose = 1;

**break**;

**case** 'p':             /\**don't print a prompt*\*/

            emit\_prompt = 0;  /\**handy for automatic testing*\*/

**break**;

**default**:

**usage**();

    }

    }

    /\**Install the signal handlers*\*/

    /\**These are the ones you will need to implement*\*/

**Signal**(**SIGINT**,  **sigint\_handler**);   /\**ctrl-c*\*/

**Signal**(SIGTSTP, **sigtstp\_handler**);  /\**ctrl-z*\*/

**Signal**(SIGCHLD, **sigchld\_handler**);  /\**Terminated or stopped child*\*/

    /\**This one provides a clean way to kill the shell*\*/

**Signal**(SIGQUIT, **sigquit\_handler**);

    /\**Initialize the job list*\*/

**initjobs**(jobs);

    /\**Execute the shell's read/eval loop*\*/

**while** (1) {

    /\**Read command line*\*/

**if** (emit\_prompt) {

**printf**("%s", prompt);

**fflush**(**stdout**);

    }

**if** ((**fgets**(cmdline, **MAXLINE**, **stdin**) **==** **NULL**) **&&** **ferror**(**stdin**))

**app\_error**("fgets error");

**if** (**feof**(**stdin**)) { /\**End of file (ctrl-d)*\*/

**fflush**(**stdout**);

**exit**(0);

    }

    /\**Evaluate the command line*\*/

**eval**(cmdline);

**fflush**(**stdout**);

**fflush**(**stdout**);

    }

**exit**(0); /\**control never reaches here*\*/

}

/\*

*\* eval - Evaluate the command line that the user has just typed in*

*\**

*\* If the user has requested a built-in command (quit, jobs, bg or fg)*

*\* then execute it immediately. Otherwise, fork a child process and*

*\* run the job in the context of the child. If the job is running in*

*\* the foreground, wait for it to terminate and then return.  Note:*

*\* each child process must have a unique process group ID so that our*

*\* background children don't receive SIGINT (SIGTSTP) from the kernel*

*\* when we type ctrl-c (ctrl-z) at the keyboard.*

\*/

void **eval**(char \*cmdline)

{

    /\**$begin handout*\*/

    char \*argv[**MAXARGS**]; /\**argv for execve()*\*/

    int bg;              /\**should the job run in bg or fg?*\*/

    pid\_t pid;           /\**process id*\*/

    sigset\_t mask;       /\**signal mask*\*/

    /\**Parse command line*\*/

    bg = **parseline**(cmdline, argv);

**if** (argv[0] **==** **NULL**)

**return**;   /\**ignore empty lines*\*/

**if** (**!builtin\_cmd**(argv)) {

        /\*

*\* This is a little tricky. Block SIGCHLD, SIGINT, and SIGTSTP*

*\* signals until we can add the job to the job list. This*

*\* eliminates some nasty races between adding a job to the job*

*\* list and the arrival of SIGCHLD, SIGINT, and SIGTSTP signals.*

\*/

**if** (**sigemptyset**(&mask) **<** 0)

**unix\_error**("sigemptyset error");

**if** (**sigaddset**(&mask, SIGCHLD))

**unix\_error**("sigaddset error");

**if** (**sigaddset**(&mask, **SIGINT**))

**unix\_error**("sigaddset error");

**if** (**sigaddset**(&mask, SIGTSTP))

**unix\_error**("sigaddset error");

**if** (**sigprocmask**(SIG\_BLOCK, &mask, **NULL**) **<** 0)

**unix\_error**("sigprocmask error");

    /\**Create a child process*\*/

**if** ((pid = **fork**()) **<** 0)

**unix\_error**("fork error");

    /\*

*\* Child  process*

\*/

**if** (pid **==** 0) {

        /\**Child unblocks signals*\*/

**sigprocmask**(SIG\_UNBLOCK, &mask, **NULL**);

        /\**Each new job must get a new process group ID*

*so that the kernel doesn't send ctrl-c and ctrl-z*

*signals to all of the shell's jobs*\*/

**if** (**setpgid**(0, 0) **<** 0)

**unix\_error**("setpgid error");

        /\**Now load and run the program in the new job*\*/

**if** (**execve**(argv[0], argv, **environ**) **<** 0) {

**printf**("%s: Command not found\n", argv[0]);

**exit**(0);

        }

    }

    /\*

*\* Parent process*

\*/

    /\**Parent adds the job, and then unblocks signals so that*

*the signals handlers can run again*\*/

**addjob**(jobs, pid, (bg **==** 1 **?** **BG** **:** **FG**), cmdline);

**sigprocmask**(SIG\_UNBLOCK, &mask, **NULL**);

**if** (**!**bg)

**waitfg**(pid);

**else**

**printf**("[%d] (%d) %s", **pid2jid**(pid), pid, cmdline);

    }

    /\**$end handout*\*/

**return**;

}

/\*

*\* parseline - Parse the command line and build the argv array.*

*\**

*\* Characters enclosed in single quotes are treated as a single*

*\* argument.  Return true if the user has requested a BG job, false if*

*\* the user has requested a FG job.*

\*/

int **parseline**(**const** char \*cmdline, char \*\*argv)

{

**static** char array[**MAXLINE**]; /\**holds local copy of command line*\*/

    char \*buf = array;          /\**ptr that traverses command line*\*/

    char \*delim;                /\**points to first space delimiter*\*/

    int argc;                   /\**number of args*\*/

    int bg;                     /\**background job?*\*/

**strcpy**(buf, cmdline);

    buf[**strlen**(buf)-1] = ' ';  /\**replace trailing '\n' with space*\*/

**while** (\*buf **&&** (\*buf **==** ' ')) /\**ignore leading spaces*\*/

    buf++;

    /\**Build the argv list*\*/

    argc = 0;

**if** (\*buf **==** '\'') {

    buf++;

    delim = **strchr**(buf, '\'');

    }

**else** {

    delim = **strchr**(buf, ' ');

    }

**while** (delim) {

    argv[argc++] = buf;

    \*delim = '\0';

    buf = delim + 1;

**while** (\*buf **&&** (\*buf **==** ' ')) /\**ignore spaces*\*/

           buf++;

**if** (\*buf **==** '\'') {

        buf++;

        delim = **strchr**(buf, '\'');

    }

**else** {

        delim = **strchr**(buf, ' ');

    }

    }

    argv[argc] = **NULL**;

**if** (argc **==** 0)  /\**ignore blank line*\*/

**return** 1;

    /\**should the job run in the background?*\*/

**if** ((bg = (\*argv[argc-1] **==** '&')) **!=** 0) {

    argv[--argc] = **NULL**;

    }

**return** bg;

}

/\*

*\* builtin\_cmd - If the user has typed a built-in command then execute*

*\*    it immediately.*

\*/

int **builtin\_cmd**(char \*\*argv)

{

    /\**command is "quit" ,exit*\*/

**if** (**!strcmp**(argv[0], "quit")){

**exit**(0);

    }

    /\**command is "jobs" ,listjobs*\*/

**else** **if** (**!strcmp**(argv[0], "jobs")){

**listjobs**(jobs);

**return** 1;

    }

    /\**command is "fg" or "bg", do\_bgfg*\*/

**else** **if** (**!strcmp**(argv[0], "bg") **||** **!strcmp**(argv[0], "fg")){

**do\_bgfg**(argv);

**return** 1;

    }

**return** 0;     /\**not a builtin command*\*/

}

/\*

*\* do\_bgfg - Execute the builtin bg and fg commands*

\*/

void **do\_bgfg**(char \*\*argv)

{

    /\**$begin handout*\*/

**struct** job\_t \*jobp=**NULL**;

    /\**Ignore command if no argument*\*/

**if** (argv[1] **==** **NULL**) {

**printf**("%s command requires PID or %%jobid argument\n", argv[0]);

**return**;

    }

    /\**Parse the required PID or %JID arg*\*/

**if** (**isdigit**(argv[1][0])) {

    pid\_t pid = **atoi**(argv[1]);

**if** (**!**(jobp = **getjobpid**(jobs, pid))) {

**printf**("(%d): No such process\n", pid);

**return**;

    }

    }

**else** **if** (argv[1][0] **==** '%') {

    int jid = **atoi**(&argv[1][1]);

**if** (**!**(jobp = **getjobjid**(jobs, jid))) {

**printf**("%s: No such job\n", argv[1]);

**return**;

    }

    }

**else** {

**printf**("%s: argument must be a PID or %%jobid\n", argv[0]);

**return**;

    }

    /\**bg command*\*/

**if** (**!strcmp**(argv[0], "bg")) {

**if** (**kill**(-(jobp**->**pid), SIGCONT) **<** 0)

**unix\_error**("kill (bg) error");

    jobp**->**state = **BG**;

**printf**("[%d] (%d) %s", jobp**->**jid, jobp**->**pid, jobp**->**cmdline);

    }

    /\**fg command*\*/

**else** **if** (**!strcmp**(argv[0], "fg")) {

**if** (**kill**(-(jobp**->**pid), SIGCONT) **<** 0)

**unix\_error**("kill (fg) error");

    jobp**->**state = **FG**;

**waitfg**(jobp**->**pid);

    }

**else** {

**printf**("do\_bgfg: Internal error\n");

**exit**(0);

    }

    /\**$end handout*\*/

**return**;

}

/\*

*\* waitfg - Block until process pid is no longer the foreground process*

\*/

void **waitfg**(pid\_t pid)

{

**while** (**fgpid**(jobs) **==** pid){

**sleep**(1);

    }

**return**;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*\* Signal handlers*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

*\* sigchld\_handler - The kernel sends a SIGCHLD to the shell whenever*

*\*     a child job terminates (becomes a zombie), or stops because it*

*\*     received a SIGSTOP or SIGTSTP signal. The handler reaps all*

*\*     available zombie children, but doesn't wait for any other*

*\*     currently running children to terminate.*

\*/

void **sigchld\_handler**(int sig)

{

    sigset\_t mask, prev\_mask;

    int olderrno = **errno**;

    int status;

    pid\_t pid;

**sigfillset**(&mask);

**while**((pid = **waitpid**(-1, &status, WNOHANG | WUNTRACED)) **>** 0){

**if** (**WIFEXITED**(status)){

**sigprocmask**(SIG\_BLOCK, &mask, &prev\_mask);   //*block all signals*

**deletejob**(jobs, pid);           //*delete from joblist*

**sigprocmask**(SIG\_SETMASK, &prev\_mask, **NULL**);   //*unbock*

        }

**else** **if** (**WIFSTOPPED**(status)){

**struct** job\_t \*job = **getjobpid**(jobs, pid);

            job**->**state = **ST**;

**printf**("Job [%d] (%d) stopped by signal %d\n", job**->**jid, job**->**pid, **WSTOPSIG**(status));

        }

**else** **if** (**WIFSIGNALED**(status)){

**struct** job\_t \*job = **getjobpid**(jobs, pid);

**printf**("Job [%d] (%d) terminated by signal %d\n", job**->**jid, job**->**pid, **WTERMSIG**(status));

**sigprocmask**(SIG\_BLOCK, &mask, &prev\_mask);

**deletejob**(jobs, pid);

**sigprocmask**(SIG\_SETMASK, &prev\_mask, **NULL**);

        }

    }

**errno** = olderrno;

**return**;

}

/\*

*\* sigint\_handler - The kernel sends a SIGINT to the shell whenver the*

*\*    user types ctrl-c at the keyboard.  Catch it and send it along*

*\*    to the foreground job.*

\*/

void **sigint\_handler**(int sig)

{

    int olderrno = **errno**;

    pid\_t pid = **fgpid**(jobs);

    sigset\_t mask, prev\_mask;

**sigfillset**(&mask);

**sigprocmask**(SIG\_BLOCK, &mask, &prev\_mask);

**sigprocmask**(SIG\_SETMASK, &prev\_mask, **NULL**);

**if**(pid **!=** 0){

**kill**(-pid,**SIGINT**);  //*sent SIGINT*

    }

**errno** = olderrno;

**return**;

}

/\*

*\* sigtstp\_handler - The kernel sends a SIGTSTP to the shell whenever*

*\*     the user types ctrl-z at the keyboard. Catch it and suspend the*

*\*     foreground job by sending it a SIGTSTP.*

\*/

void **sigtstp\_handler**(int sig)

{

    int olderrno = **errno**;

    pid\_t pid = **fgpid**(jobs);

    sigset\_t mask, prev\_mask;

**sigfillset**(&mask);

**sigprocmask**(SIG\_BLOCK, &mask, &prev\_mask);

**sigprocmask**(SIG\_SETMASK, &prev\_mask, **NULL**);

**if** (pid **!=** 0){

**kill**(-pid, SIGTSTP); //*sent SIGINT*

    }

**errno** = olderrno;

**return**;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*\* End signal handlers*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*\* Helper routines that manipulate the job list*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\**clearjob - Clear the entries in a job struct*\*/

void **clearjob**(**struct** job\_t \*job) {

    job**->**pid = 0;

    job**->**jid = 0;

    job**->**state = **UNDEF**;

    job**->**cmdline[0] = '\0';

}

/\**initjobs - Initialize the job list*\*/

void **initjobs**(**struct** job\_t \*jobs) {

    int i;

**for** (i = 0; i **<** **MAXJOBS**; i++)

**clearjob**(&jobs[i]);

}

/\**maxjid - Returns largest allocated job ID*\*/

int **maxjid**(**struct** job\_t \*jobs)

{

    int i, max=0;

**for** (i = 0; i **<** **MAXJOBS**; i++)

**if** (jobs[i]**.**jid **>** max)

        max = jobs[i]**.**jid;

**return** max;

}

/\**addjob - Add a job to the job list*\*/

int **addjob**(**struct** job\_t \*jobs, pid\_t pid, int state, char \*cmdline)

{

    int i;

**if** (pid **<** 1)

**return** 0;

**for** (i = 0; i **<** **MAXJOBS**; i++) {

**if** (jobs[i]**.**pid **==** 0) {

        jobs[i]**.**pid = pid;

        jobs[i]**.**state = state;

        jobs[i]**.**jid = nextjid++;

**if** (nextjid **>** **MAXJOBS**)

        nextjid = 1;

**strcpy**(jobs[i]**.**cmdline, cmdline);

**if**(verbose){

**printf**("Added job [%d] %d %s\n", jobs[i]**.**jid, jobs[i]**.**pid, jobs[i]**.**cmdline);

            }

**return** 1;

    }

    }

**printf**("Tried to create too many jobs\n");

**return** 0;

}

/\**deletejob - Delete a job whose PID=pid from the job list*\*/

int **deletejob**(**struct** job\_t \*jobs, pid\_t pid)

{

    int i;

**if** (pid **<** 1)

**return** 0;

**for** (i = 0; i **<** **MAXJOBS**; i++) {

**if** (jobs[i]**.**pid **==** pid) {

**clearjob**(&jobs[i]);

        nextjid = **maxjid**(jobs)+1;

**return** 1;

    }

    }

**return** 0;

}

/\**fgpid - Return PID of current foreground job, 0 if no such job*\*/

pid\_t **fgpid**(**struct** job\_t \*jobs) {

    int i;

**for** (i = 0; i **<** **MAXJOBS**; i++)

**if** (jobs[i]**.**state **==** **FG**)

**return** jobs[i]**.**pid;

**return** 0;

}

/\**getjobpid  - Find a job (by PID) on the job list*\*/

**struct** job\_t \***getjobpid**(**struct** job\_t \*jobs, pid\_t pid) {

    int i;

**if** (pid **<** 1)

**return** **NULL**;

**for** (i = 0; i **<** **MAXJOBS**; i++)

**if** (jobs[i]**.**pid **==** pid)

**return** &jobs[i];

**return** **NULL**;

}

/\**getjobjid  - Find a job (by JID) on the job list*\*/

**struct** job\_t \***getjobjid**(**struct** job\_t \*jobs, int jid)

{

    int i;

**if** (jid **<** 1)

**return** **NULL**;

**for** (i = 0; i **<** **MAXJOBS**; i++)

**if** (jobs[i]**.**jid **==** jid)

**return** &jobs[i];

**return** **NULL**;

}

/\**pid2jid - Map process ID to job ID*\*/

int **pid2jid**(pid\_t pid)

{

    int i;

**if** (pid **<** 1)

**return** 0;

**for** (i = 0; i **<** **MAXJOBS**; i++)

**if** (jobs[i]**.**pid **==** pid) {

**return** jobs[i]**.**jid;

        }

**return** 0;

}

/\**listjobs - Print the job list*\*/

void **listjobs**(**struct** job\_t \*jobs)

{

    int i;

**for** (i = 0; i **<** **MAXJOBS**; i++) {

**if** (jobs[i]**.**pid **!=** 0) {

**printf**("[%d] (%d) ", jobs[i]**.**jid, jobs[i]**.**pid);

**switch** (jobs[i]**.**state) {

**case** **BG**:

**printf**("Running ");

**break**;

**case** **FG**:

**printf**("Foreground ");

**break**;

**case** **ST**:

**printf**("Stopped ");

**break**;

**default**:

**printf**("listjobs: Internal error: job[%d].state=%d ",

               i, jobs[i].state);

        }

**printf**("%s", jobs[i]**.**cmdline);

    }

    }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*\* end job list helper routines*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*\* Other helper routines*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

*\* usage - print a help message*

\*/

void **usage**(void)

{

**printf**("Usage: shell [-hvp]\n");

**printf**("   -h   print this message\n");

**printf**("   -v   print additional diagnostic information\n");

**printf**("   -p   do not emit a command prompt\n");

**exit**(1);

}

/\*

*\* unix\_error - unix-style error routine*

\*/

void **unix\_error**(char \*msg)

{

**fprintf**(**stdout**, "%s: %s\n", msg, **strerror**(**errno**));

**exit**(1);

}

/\*

*\* app\_error - application-style error routine*

\*/

void **app\_error**(char \*msg)

{

**fprintf**(**stdout**, "%s\n", msg);

**exit**(1);

}

/\*

*\* Signal - wrapper for the sigaction function*

\*/

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

{

**struct** sigaction action, old\_action;

    action**.**sa\_handler = handler;

**sigemptyset**(&action**.**sa\_mask); /\**block sigs of type being handled*\*/

    action**.**sa\_flags = SA\_RESTART; /\**restart syscalls if possible*\*/

**if** (**sigaction**(signum, &action, &old\_action) **<** 0)

**unix\_error**("Signal error");

**return** (old\_action**.**sa\_handler);

}

/\*

*\* sigquit\_handler - The driver program can gracefully terminate the*

*\*    child shell by sending it a SIGQUIT signal.*

\*/

void **sigquit\_handler**(int sig)

{

**printf**("Terminating after receipt of SIGQUIT signal\n");

**exit**(1);

}

# 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 | |

### 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 | |

### 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 | |

### 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 | |

### 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 | |

### 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 | |

### 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 | |

### 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 | | |

### 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 | |

### 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 | |

### 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 |

### 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 |

### 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 |

### 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 | |

### 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 | |

# 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. 理解了shell的工作原理。

2. 对信号的处理过程有了更深的理解。

3. 明白了信号处理相关的系统函数的作用

## 5.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

# 参考文献

[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.