哈爾濱工業大學

实验报告

实 验（七）

题　　目 TinyShell

微壳

专　　业 计算机类

学　　号 1180300829

班　　级 1803008

学　　生 余　涛

指 导 教 师 吴　锐

实 验 地 点 G709

实 验 日 期 2019.12.7

计算机科学与技术学院

目 录

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；TestStudio；Gprof;Valgrind 等

1.3 实验预习

- **上实验课前，必须认真预习实验指导书（PPT 或 PDF）**
- **了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。**
- **了解进程、作业、信号的基本概念和原理**

- 了解 **shell** 的基本原理
- 熟知进程创建、回收的方法和相关系统函数
- 熟知信号机制和信号处理相关的系统函数

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：指程序的依次运行过程。更确切说，进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，对于函数 int fork(void):

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，对于函数 int wait(int *child_status)：

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：

signal 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：

类似于异常和中断

从内核发送到（有时是在另一个进程的请求下）一个进程

信号类型是用小整数 ID 来标识的(1-30)

信号中唯一的信息是它的 ID 和它的到达。

包括发送信号：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；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	**信号名称**	**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	dump	键盘产生的退出
4	SIGILL	dump	非法指令
5	SIGTRAP	dump	debug 中断
6	SIGABRT / SIGIOT	dump	异常中止
7	SIGBUS / SIGEMT	dump	总线异常/EMT 指令
8	SIGFPE	dump	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	dump	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	继续	如进程停止状态则开始运行
19	SIGSTOP	停止	停止进程运行
20	SIGSTP	停止	键盘产生的停止
21	SIGTTIN	停止	后台进程请求输入
22	SIGTTOU	停止	后台进程请求输出
23	SIGURG	忽略	socket 发生紧急情况
24	SIGXCPU	dump	CPU 时间限制被打破

25	SIGXFSZ	dump	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/SIGPOLL	终止	I/O 可用
30	SIGPWR	终止	电源异常
31	SIGSYS / SYSUNUSED	dump	系统调用异常

2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分）

一. 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 程序发送信号

/bin/kill 程序可以向另外的进程或进程组发送任意的信号

Examples：/bin/kill －9 24818 发送信号 9（SIGKILL）给进程 24818

/bin/kill －9 －24817 发送信号 SIGKILL 给进程组 24817 中的每个进程

（负的 PID 会导致信号被发送到进 程组 PID 中的每个进程）

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 SIGINT 默认情况是终止前台作业 SIGTSTP 默认情况是停止（挂起）前台作业。

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

(1)kill()和 raise()

kill()函数和熟知的 kill 系统命令一样，可以发送信号给信号和进程组（实际上 kill 系统命令只是 kill 函数的一个用户接口），需要注意的是他不仅可以终止进程(发送 SIGKILL 信号)，也可以向进程发送其他信号。

与 kill 函数不同的是 raise()函数允许进程向自身发送信号。

(2)函数格式：

kill 函数的语法格式：

int kill(pid_t pid,int sig)，函数传入值为 sig 信号和 pid，返回值成功为 0，出错为-1

raise()函数语法要点：

int raise(int sig),函数传入值为 sig 信号，返回值成功为 0，出错为-1

(3)alarm()和 pause()

alarm()-----也称为闹钟函数，可以在进程中设置一个定时器，等到时间到达时，就会想进程发送 SIGALARM 信号，注意的是一个进程只能有一个闹钟时间，如果调用 alarm()之前已经设置了闹钟时间，那么任何以前的闹钟时间都会被新值所代

pause()----此函数用于将进程挂起直到捕捉到信号为止，这个函数很常用，通常用于判断信号是否已到

alarm()函数语法：

unsigned int alram(unsigned int seconds)，函数传入值为 seconds 指定秒数，返回值如果成功且在调用函数前设置了闹钟时间，则返回闹钟时间的剩余时间，否则返回 0，出错返回-1

pause()函数语法如下：

int pause(void)，返回-1，并且把 error 值设为 ETNTR

二. 阻塞信号

阻塞和解除阻塞信号

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

选定的信号辅助函数：

sigempty(set) – 初始化 set 为空集合

sigfill(set) – 把每个信号都添加到 set 中

sigadd(set) – 把指定的信号 signum 添加到 set

sigdel(set) – 从 set 中删除指定的信号 signum

三. 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)

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 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 也是一支程序，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）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：用户输入自己的命令行，该函数对命令行进行评估，如果用户请求了 quit、jobs、bg、fg 这些内核内置的命令，就执行这些命令。若不是这些内置命令，则 fork 的一个子进程会在子进程的上下文中运行该作业。当作业在前台运行时，则等待终止返回，若是在后台运行则打印进程信息。

参　　数：形参：cmdline。实参：argv，mask，SIG_UNBLOCK，argv[0]，environ

处理流程：

1.定义 argv[MAXARGS]，bg，pid，mask 这些变量。

2.调用 parseline()函数解析用户输入的 cmdline 命令行，得到命令行参数。

3.eval 调用 builtin_cmd()函数，判断命令行是否是内置命令 quit、jobs、bg、fg，如果不是，就继续执行。

4. if (sigemptyset(&mask) < 0)　unix_error("sigemptyset error");来设置阻塞集合，将 mask 初始化为空集合，然后调用 if (sigaddset(&mask, SIGCHLD)) unix_error("sigaddset error");if (sigaddset(&mask, SIGINT)) unix_error("sigaddset error");if (sigaddset(&mask, SIGTSTP))　unix_error("sigaddset error"); 来将 SIGCHLD , SIGINT ,SIGTSTP 信号加入阻塞集合。

5.在子进程中调用 sigprocmask(SIG_UNBLOCK, &mask, NULL);解除对 SIG_CHLD 的阻塞，再调用 setpgid(0, 0)创建一个虚拟的进程组。然后调用 execve(argv[0], argv, environ)执行文件。

6.调用 addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);将 job 添加到 job list，调用 sigprocmask(SIG_UNBLOCK, &mask, NULL);解除 SIG_CHID 阻塞信号，然后运行 if (!bg)　waitfg(pid);else　printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);来判

断进程是什么进程，若是前台进程，则调用 waitfg 函数；若是后台进程，则打印当前的进程信息。

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.我们最开始需要设置阻塞信号，让信号可以被发送但不会被接收，这样就能够避免执行 addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);是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：判断用户输入的是哪种内置命令，执行内置命令 quit、jobs、bg、fg

参　　数：形参：argv。实参：argv[0],”quit”,”&”,”jobs”,”bg”,”fg”,argv,SIG_BLOCK, mask,pre_mask,NULL

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.判断是否为退出命令：

```
if (!strcmp(argv[0], "quit"))   //如果为退出命令
    {
        exit(0);
    }
```

3.忽略单独的&：

```
if (!strcmp(argv[0], "&"))      //忽略单独的&
    {
        return 1;
    }
```

4.如果为 jobs 命令：

```
if (!strcmp(argv[0], "jobs"))      //如果为 jobs，用 listjobs 输出作业列表的所有作业
```

信息

```
{
    listjobs(jobs);
    return 1;
}
```

5.如果为 bg 或 fg 命令：

if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg")) //如果为 bg 或 fg，调用 do_bgfg 执行 bg 和 fg 两天内置指令

```
{
    do_bgfg(argv);
    return 1;
}
```

6.如果不是内置命令返回 0。

要点分析：由于 jobs 是全局变量，可能会被修改，于是可以尝试阻塞全部信号的方式防止其被修改，但我发现 listjobs 函数只是输出作业列表的所有信息，并未改变 jobs 的值，所以不阻塞全部信号也不会改变 jobs 的值，所以我又去掉了阻塞全部信号的过程

3. 1.3 void do_bgfg(char **argv) 函数（5 分）

函数功能：执行用户输入的内置 bg 和 fg 命令

参　　数：形参：argv。实参：argv[1] , argv[1][0] , jobs , pid , argv[1] , argv[0]

处理流程：

1.通过 if (argv[1] == NULL) {printf("%s command requires PID or %%jobid argument\n", argv[0]);return;}判断 bg 和 fg 后面是否有参数，如果没有参数，那么久忽略命令。

2.通过 if (isdigit(argv[1][0]))和 else if (argv[1][0] == '%')判断 bg 或 fg 后面是数字还是%加数字的形式。若是数字，则说明取得进程号，得到进程号后，执行 jobp = getjobpid(jobs, pid)得到作业号 job；若是%，则需要知道%后面的作业 job 号，则需要 int jid = atoi(&argv[1][1]);和 jobp = getjobjid(jobs, jid)得到作业号 job。

3.判断 bg 和 fg。通过 strcmp(argv[0], "bg")判断 argv[0]是否为后台进程 bg，若是，则执行 kill(-(jobp->pid), SIGCONT)发送 SIGCONT 给进程组 PID 的每一个进程，并且通过 jobp->state = BG 设置任务状态为 BG，打印任务的 jid，pid 和命令行。通过 strcmp(argv[0], "fg")判断 argv[0]是否为前台进程 fg，若是，则执行 kill(-(jobp->pid), SIGCONT)发送 SIGCONT 给进程组 PID 的每一个进程，并且通过 jobp->state = FG 设置任务状态为 FG，然后调用 waitfg(jobp->pid)等待前台进程 fg 结束

4.不是 bg 和 fg 则打印"do_bgfg: Internal error\n"并且终止。

要点分析：

1.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。

2.SIGCONT 信号的作用是继续执行一个停止的进程。

3. 1.4 void waitfg(pid_t pid) 函数（5 分）

函数功能：不断阻止前台进程直到 pid 不是前台进程

参　　数：形参：前台进程 pid。实参：jobs

处理流程：

通过 while (pid == fgpid(jobs))不断判断传入的 pid 是否是一个前台进程的 pid，是的话则一直循环，不是就跳出循环。循环内部如果不是前台进程的 pid，则使用 sleep(1)使进程休眠，直到收到一个让该进程终止或处理的程序信号。

要点分析：

1.根据教材的提示，循环的内部如果只是使用 pause(),程序就会等待很长时间才再次检查循环终止条件，导致运行时间特别长，导致一直不能出现结果，且如果在 while 测试后和 pause 之前收到了 SIGCHLD 信号，pause 将永远睡眠。使用 nanosleep 这样高精度的休眠函数也不可能接受，所以最合适的方法是使用 siguspend。但由于 ppt 建议使用 while(xxx),sleep(1)的方式，所以使用了这种方式。

- **waitfg函数和SIGCHLD信号处理程序的分工配合需要斟酌**
- **建 议：**
  - 在waitfg函数中，在sleep函数附近使用busy loop，例如：

    while (xxxxx) sleep(1);

2.siguspend 函数等价于下面代码：

sigpromask(SIG_SETMASK,&mask,&prev);

pause();

sigprocmask(SIG_SETMASK,&prev,NULL);

3. 1.5 void sigchld_handler(int sig) 函数（10 分）

函数功能：当子作业终止变为僵尸时，或者由于收到了 SIGSTOP 或者 SIGTSTP 信号而停止了，内核就像 shell 发送 SIGCHID，将所有可用的僵尸子节点收集，但并不等待其他正在运行的子节点终止

参　　数：形参：sig。实参：status , WNOHANG|WUNTRACED , SIG_BLOCK , mask , prev_mask , SIG_SETMASK

处理流程：

1.首先我们需要设置 int olderrno = errno，然后执行 sigfillset(&mask)将所有信号都添加到 mask 阻塞集合里面。

2.根据 ppt 的提示，为了最大可能的回收子进程，我们可以使用 pid = waitpid(-1, &status, WNOHANG | WUNTRACED) 在 while 循环中，WNOHANG | WUNTRACED 表示立即返回，如果等待集合中没有进程被中止或停止返回 0，否则孩子返回进程的 pid。

```
while ((child_pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0)
{
.....
}
```

3.进入循环中，首先需要阻塞信号和解除阻塞信号，大概过程是：

sigset_t mask,prev_mask;

sigfillset(&mask);

…

sigprocmask(SIG_BLOCK,&mask,&prev_mask);
sigprocmask(SIG_SETMASK,&prev_mask,NULL);
为了找到 job，需要执行 jobnew = getjobpid(jobs, pid)得到 pid 的 jib
4.一共有两种子进程的退出状态情况，第一种为 WIFSTOPPED(status)，这种情况为真的话就说明是由子进程停止引起 waitpid 函数返回，这个时候使用 jobnew->state = ST 将 job 的状态改为 ST，并且执行 printf("Job [%d] (%d) terminated by signal %d\n", jobnew->jid, jobnew->pid, WSTOPSIG(status))进行输出。第二种为 WIFSIGNALED(status)为真，说明子进程是因为某一个信号没有被捕获导致终止的，执行 printf("Job [%d] (%d) terminated by signal %d\n", jobnew->jid, jobnew->pid, WTERMSIG(status))输出信息即可。然后执行 deletejob(jobs, pid)进行回收。
5. fflush(stdout); sigprocmask(SIG_SETMASK, &prev_mask, NULL);来清空缓冲区并且解除阻塞
6.恢复 errno = olderrno 并且返回
要点分析：
1.由于 deletejob()函数会对全局变量 jobs 的值进行更改，所以需要阻塞信号来防止这种更改。
2. 阻塞信号和解除阻塞信号的大概过程是：
sigset_t mask,prev_mask;
sigfillset(&mask);

…

sigprocmask(SIG_BLOCK,&mask,&prev_mask);
sigprocmask(SIG_SETMASK,&prev_mask,NULL);

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)
{
    if (!strcmp(argv[0], "quit"))    //如果为退出命令
    {
        exit(0);
    }
    if (!strcmp(argv[0], "&"))       //忽略单独的&
    {
        return 1;
    }
```

```
    if (!strcmp(argv[0], "jobs"))       //如果为 jobs，用 listjobs 输出作业列表的所有作业信息
    {
        listjobs(jobs);
        return 1;
    }
    if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg"))   //如果为 bg 或 fg，调用 do_bgfg 执行 bg 和 fg 两天内置指令
    {
        do_bgfg(argv);
        return 0;
    }
    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(pid==fgpid(jobs))
        {
```

```
            sleep(1);    //暂时休眠挂起
        }

}

/******************
 * 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)
{

    struct job_t* jobnew;      //新建 job_t 的结构体指针 job_new
    sigset_t mask, prev_mask;
    int olderrno = errno, status;
    pid_t pid;
    sigfillset(&mask);      //将每个信号都添加到 mask 中
```

```
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)    //尽可能回收子进程
    {
        sigprocmask(SIG_BLOCK, &mask, &prev_mask);   //需要 deletejob，阻塞所有信号
        jobnew = getjobpid(jobs, pid);   //通过 pid 找到作业 job
        if (WIFSTOPPED(status)) //如果子进程停止引起 waitpid 函数返回
        {
            jobnew->state = ST;
            printf("Job [%d] (%d) terminated by signal %d\n", jobnew->jid, jobnew->pid, WSTOPSIG(status));
        }
        else
        {
            if (WIFSIGNALED(status))   //如果子进程终止引起的返回
            printf("Job [%d] (%d) terminated by signal %d\n", jobnew->jid, jobnew->pid, WTERMSIG(status));
            deletejob(jobs, pid);   //终止的进程直接回收
        }
        fflush(stdout);
        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)
{
    pid_t pid;
    sigset_t mask, prev_mask;
    int olderrno = errno;
    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask);  //需要获取前台进程 pid，阻塞所有信号
    pid = fgpid(jobs);  //获取前台作业 pid
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    if (pid != 0)  //只处理前台作业
        kill(-pid, 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)
{
    pid_t pid;
    sigset_t mask, prev_mask;
    int olderrno = errno;
    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask);  //需要获取前台进程 pid，阻塞所有信号
    pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    if (pid != 0)
        kill(-pid, SIGTSTP);
    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)。

4.2 测试结果评价

tsh 与 tshref 的输出在一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

4.3.1 测试用例 trace01.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
```

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
```

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

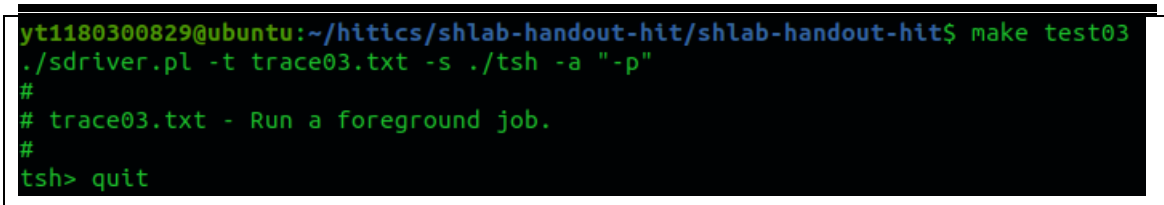
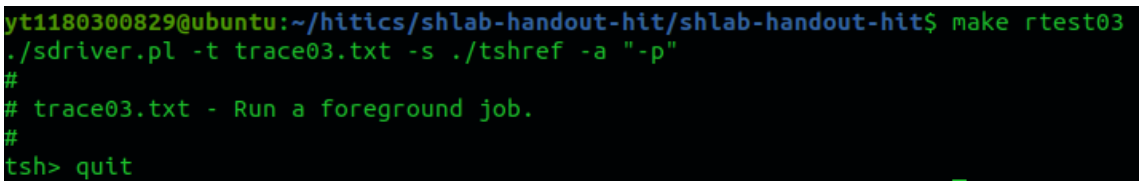
```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test02
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest02
./sdriver.pl -t trace02.txt -s ./tshref -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

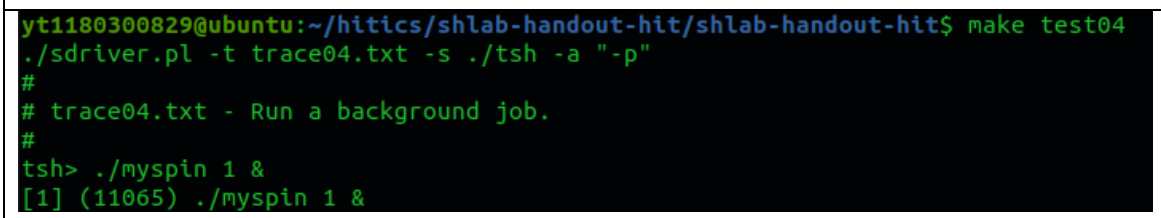
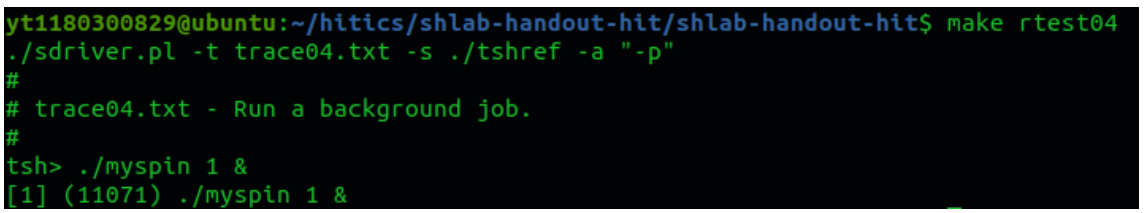
测试结论	相同/不同，原因分析如下：相同

4.3.3 测试用例 trace03.txt 的输出截图（1 分）

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 的输出截图（1 分）

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 的输出截图（1 分）

tsh 测试结果	tshref 测试

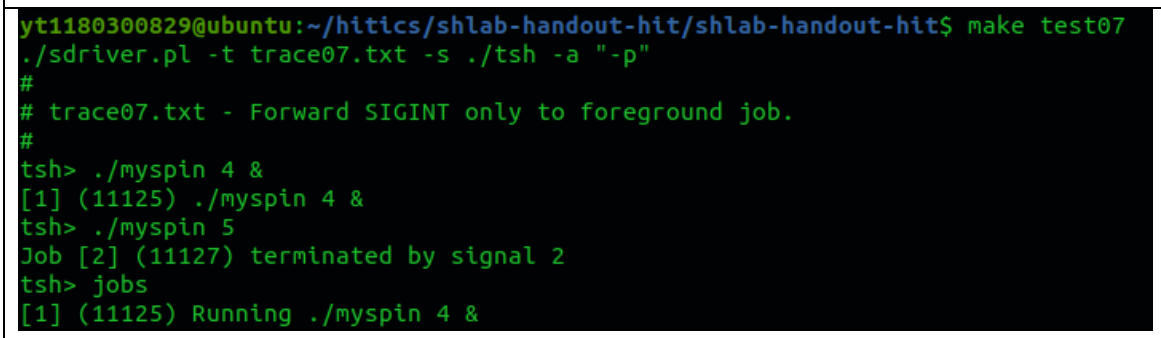
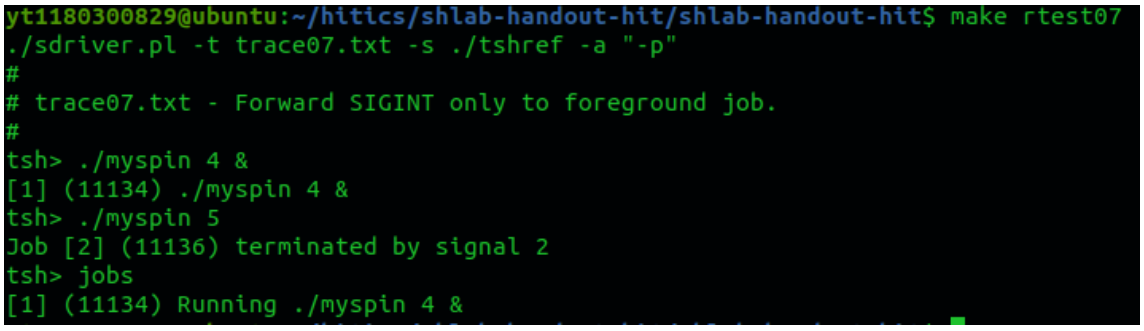
	结果
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test05 ./sdriver.pl -t trace05.txt -s ./tsh -a "-p" # # trace05.txt - Process jobs builtin command. # tsh> ./myspin 2 & [1] (11077) ./myspin 2 & tsh> ./myspin 3 & [2] (11079) ./myspin 3 & tsh> jobs [1] (11077) Running ./myspin 2 & [2] (11079) Running ./myspin 3 & yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest05 ./sdriver.pl -t trace05.txt -s ./tshref -a "-p" # # trace05.txt - Process jobs builtin command. # tsh> ./myspin 2 & [1] (11086) ./myspin 2 & tsh> ./myspin 3 & [2] (11088) ./myspin 3 & tsh> jobs [1] (11086) Running ./myspin 2 & [2] (11088) Running ./myspin 3 &	
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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test06 ./sdriver.pl -t trace06.txt -s ./tsh -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh> ./myspin 4 Job [1] (11096) terminated by signal 2 yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest06 ./sdriver.pl -t trace06.txt -s ./tshref -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh> ./myspin 4 Job [1] (11102) terminated by signal 2	

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 的输出截图（1 分）

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test08
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (11143) ./myspin 4 &
tsh> ./myspin 5
Job [2] (11145) terminated by signal 20
tsh> jobs
[1] (11143) Running ./myspin 4 &
[2] (11145) Stopped ./myspin 5
```

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest08
./sdriver.pl -t trace08.txt -s ./tshref -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (11152) ./myspin 4 &
tsh> ./myspin 5
Job [2] (11154) stopped by signal 20
tsh> jobs
[1] (11152) Running ./myspin 4 &
[2] (11154) Stopped ./myspin 5
```

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (11161) ./myspin 4 &
tsh> ./myspin 5
Job [2] (11163) terminated by signal 20
tsh> jobs
[1] (11161) Running ./myspin 4 &
[2] (11163) Stopped ./myspin 5
tsh> bg %2
[2] (11163) ./myspin 5
tsh> jobs
[1] (11161) Running ./myspin 4 &
[2] (11163) Running ./myspin 5
```

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (11161) ./myspin 4 &
tsh> ./myspin 5
Job [2] (11163) terminated by signal 20
tsh> jobs
[1] (11161) Running ./myspin 4 &
[2] (11163) Stopped ./myspin 5
tsh> bg %2
[2] (11163) ./myspin 5
tsh> jobs
[1] (11161) Running ./myspin 4 &
[2] (11163) Running ./myspin 5
```

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test10
./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (11183) ./myspin 4 &
tsh> fg %1
Job [1] (11183) terminated by signal 20
tsh> jobs
[1] (11183) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest10
./sdriver.pl -t trace10.txt -s ./tshref -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (11212) ./myspin 4 &
tsh> fg %1
Job [1] (11212) stopped by signal 20
tsh> jobs
[1] (11212) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test11
./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (11239) terminated by signal 2
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
  1987 tty2     Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_
SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1989 tty2     Sl+    0:50 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user
/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
  2000 tty2     Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --sessi
on=ubuntu
  2199 tty2     Sl+    4:20 /usr/bin/gnome-shell
  2232 tty2     Sl     0:00 ibus-daemon --xim --panel disable
  2236 tty2     Sl     0:00 /usr/lib/ibus/ibus-dconf
  2237 tty2     Sl     0:01 /usr/lib/ibus/ibus-extension-gtk3
```

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest11
./sdriver.pl -t trace11.txt -s ./tshref -a "-p"
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (11265) terminated by signal 2
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1987 tty2     Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_
SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
   1989 tty2     Sl+    0:53 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user
/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
   2000 tty2     Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --sessi
on=ubuntu
   2199 tty2     Sl+    4:33 /usr/bin/gnome-shell
   2232 tty2     Sl     0:00 ibus-daemon --xim --panel disable
   2236 tty2     Sl     0:00 /usr/lib/ibus/ibus-dconf
   2237 tty2     Sl     0:01 /usr/lib/ibus/ibus-extension-gtk3
   2241 tty2     Sl     0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
   2347 tty2     Sl+    0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
```

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test12
./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
#
# trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (11276) terminated by signal 20
tsh> jobs
[1] (11276) Stopped ./mysplit 4
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1987 tty2     Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_
SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
   1989 tty2     Sl+    0:54 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user
/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
   2000 tty2     Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --sessi
on=ubuntu
   2199 tty2     Sl+    4:36 /usr/bin/gnome-shell
   2232 tty2     Sl     0:00 ibus-daemon --xim --panel disable
   2236 tty2     Sl     0:00 /usr/lib/ibus/ibus-dconf
   2237 tty2     Sl     0:01 /usr/lib/ibus/ibus-extension-gtk3
```

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest12
./sdriver.pl -t trace12.txt -s ./tshref -a "-p"
#
# trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (11286) stopped by signal 20
tsh> jobs
[1] (11286) Stopped ./mysplit 4
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
  1987 tty2     Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_
SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1989 tty2     Sl+    0:54 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user
/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
  2000 tty2     Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --sessi
on=ubuntu
  2199 tty2     Sl+    4:37 /usr/bin/gnome-shell
  2232 tty2     Sl     0:00 ibus-daemon --xim --panel disable
  2236 tty2     Sl     0:00 /usr/lib/ibus/ibus-dconf
  2237 tty2     Sl     0:01 /usr/lib/ibus/ibus-extension-gtk3
  2241 tty2     Sl     0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
  2347 tty2     Sl+    0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
```

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test13
./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (11296) terminated by signal 20
tsh> jobs
[1] (11296) Stopped ./mysplit 4
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
  1987 tty2     Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_
SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1989 tty2     Sl+    0:55 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user
/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
  2000 tty2     Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --sessi
on=ubuntu
  2199 tty2     Sl+    4:40 /usr/bin/gnome-shell
  2232 tty2     Sl     0:00 ibus-daemon --xim --panel disable
  2236 tty2     Sl     0:00 /usr/lib/ibus/ibus-dconf
  2237 tty2     Sl     0:01 /usr/lib/ibus/ibus-extension-gtk3
  2241 tty2     Sl     0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
  2347 tty2     Sl+    0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
```

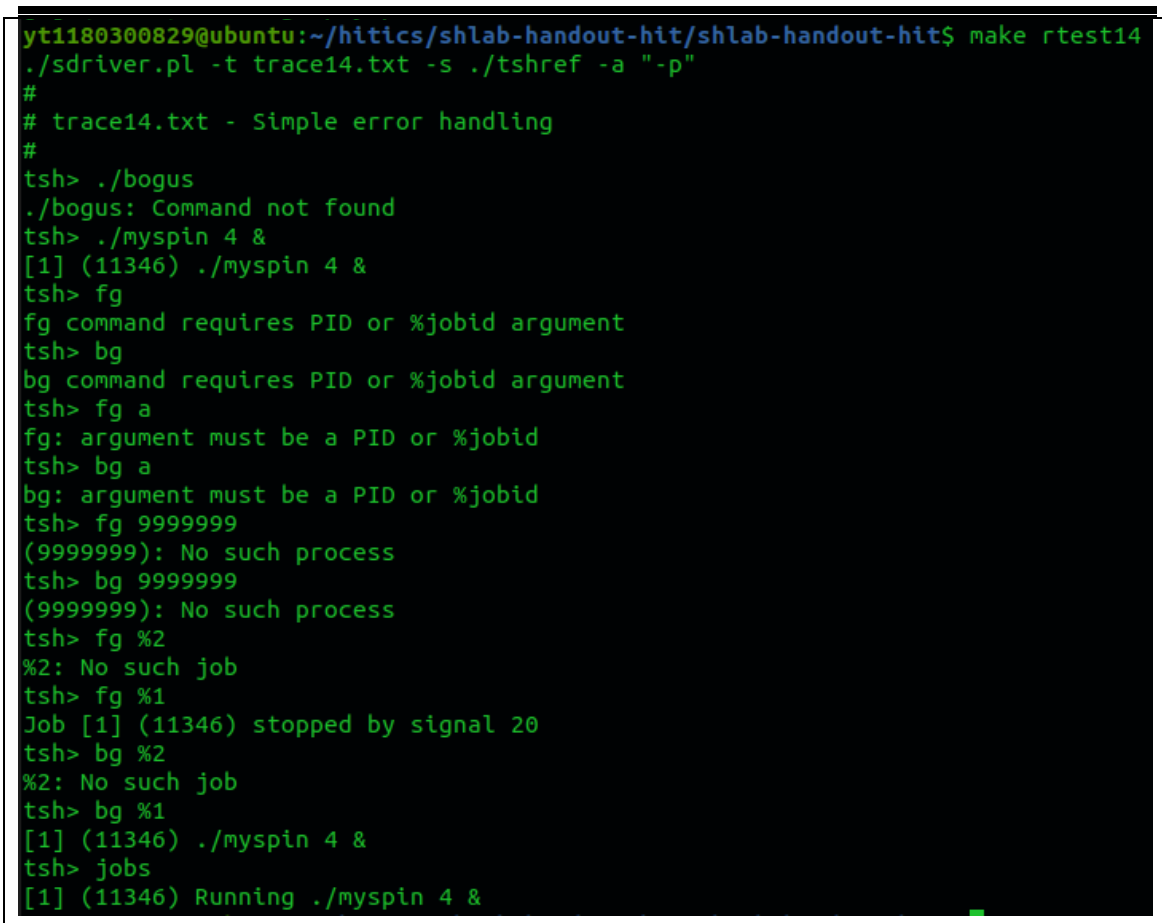
```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest13
./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (11309) stopped by signal 20
tsh> jobs
[1] (11309) Stopped ./mysplit 4
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
  1987 tty2     Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_
SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1989 tty2     Sl+    0:55 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user
/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
  2000 tty2     Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --sessi
on=ubuntu
  2199 tty2     Sl+    4:41 /usr/bin/gnome-shell
  2232 tty2     Sl     0:00 ibus-daemon --xim --panel disable
  2236 tty2     Sl     0:00 /usr/lib/ibus/ibus-dconf
  2237 tty2     Sl     0:01 /usr/lib/ibus/ibus-extension-gtk3
  2241 tty2     Sl     0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
```

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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test14
./sdriver.pl -t trace14.txt -s ./tsh -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (11327) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
Job [1] (11327) terminated by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (11327) ./myspin 4 &
tsh> jobs
[1] (11327) Running ./myspin 4 &
```



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 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (11365) terminated by signal 2
tsh> ./myspin 3 &
[1] (11367) ./myspin 3 &
tsh> ./myspin 4 &
[2] (11369) ./myspin 4 &
tsh> jobs
[1] (11367) Running ./myspin 3 &
[2] (11369) Running ./myspin 4 &
tsh> fg %1
Job [1] (11367) terminated by signal 20
tsh> jobs
[1] (11367) Stopped ./myspin 3 &
[2] (11369) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (11367) ./myspin 3 &
tsh> jobs
[1] (11367) Running ./myspin 3 &
[2] (11369) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

<table>
<tr><td colspan="2">

```
yt1180300829@ubuntu:~/hitics/shlab-handout-hit/shlab-handout-hit$ make rtest15
./sdriver.pl -t trace15.txt -s ./tshref -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (11385) terminated by signal 2
tsh> ./myspin 3 &
[1] (11387) ./myspin 3 &
tsh> ./myspin 4 &
[2] (11389) ./myspin 4 &
tsh> jobs
[1] (11387) Running ./myspin 3 &
[2] (11389) Running ./myspin 4 &
tsh> fg %1
Job [1] (11387) stopped by signal 20
tsh> jobs
[1] (11387) Stopped ./myspin 3 &
[2] (11389) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (11387) ./myspin 3 &
tsh> jobs
[1] (11387) Running ./myspin 3 &
[2] (11389) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

</td></tr>
<tr><td>测试结论</td><td>相同/不同，原因分析如下：相同</td></tr>
</table>

4.4 自测试评分

根据节 4.3 的自测试结果，程序的测试评分为：15 。

第 4 章 总结

4.1 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

4.2 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.