哈爾濱工業大學

实验报告

实 验（七）

题　　目 TinyShell

微壳

专　　业 计算机科学与技术

学　　号 1171000410

班　　级 1703005

学　　生 强文杰

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2018.12.2

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

Gcc ,Codeblocks

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.进程的概念：

狭义上：进程是一个执行中程序的示例。

广义上：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。

2.进程的创建方法：

每次用户通过 shell 输入一个可执行目标文件的名字，运行程序时，shell 就会创建一个新的进程。

具体到实现：父进程同通过调用 fork 函数创建一个新的运行的子进程。

进程的创建： //内核里面只有 fork 和 exec 两种可以创建进程。其他方式都是使用的这两种方式，如 system()封装了 exec。

1.system 函数:不常用

```
#include <stdlib.h>
int main(){
system("ls -l") //创建了一个 ls -l 的进程。system("clear")表示清屏。
}
```

2.fork()

复制出一份。各自是各自的。每个进程都有一个文件表项。

其实就是为了

比如说双胞胎，出生之后完全一样。一个身上有什么东西，另一个人相同的地方就有什么东西。

父进程先返回子进程的 pid()，第二次子进程返回 0.

例子 1：

```
pid_t pid;
```

pid=fork();//fork 不需要传递参数。
if(pid==0){ //这里写子进程接下来做的事
printf("%d%d",getpid(),getppid());
while(1);
}else{ //这里写父进程接下来做的事
printf("%d",pid);
}
3.exec 系列(工作中用的太少了)
一个进程创建另一个进程，会直接用新进程覆盖原有进程的代码段。
exec 系列有 6 个函数：execl()、execlp()、execle()、execv()、execvp()、execvpe()

例：
add.c
int main(int argc,char *argv[]){
int i=atoi(argv[0]);
int j=atoi(argv[1]);
return i+j;
}
execl.c
#include <unistd.h>
#include <stdio.h>
int main(){
excel("./add","add","1","2",NULL) ； //./ 可执行文件 argv[0] argv[1] argv[2].....NULL 。参数一定要从 argv[0]开始写，最后写 NULL
printf("hello\n"); printf("hello\n");及以下的都不会再执行了，因为从 excel 走进了另一个世界
....
}
4.popen

3.进程的回收：

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 init 进程去回收它们。一个进程可以调用 waitpid 函数来等待它的子进程终止或者停止，其中 wait 函数是 waitpid 函数的简单版本，wait(&status) 等价于调用 waitpid(-1,&status,0) 。

孤儿进程：

如果父进程先于子进程退出，则子进程成为孤儿进程，此时将自动被 PID 为 1 的进程（即 init）接管。孤儿进程退出后，它的清理工作有祖先进程 init 自动处理。init 清理没那么及时，毕竟不是亲爹。

子进程退出时，应当由父进程回收资源。应当避免父进程退出了，子进程还在的情况。即，应当避免孤儿进程。所以父进程要等子进程运行完了再退出。

例：孤儿进程

```
pid_t pid = fork();
if( pid == 0) {
while(1) ; //父进程退出了，之后子进程还在运行。通过 ps -elf 可以看到进程的 ppid 变成了 1.
}else{
exit(0);
}
```

僵尸进程：你死了没人给你收尸！

如果子进程先退出，系统不会自动清理掉子进程的环境，而必须由父进程调用 wait 或 waitpid 函数来完成清理工作，如果父进程不做清理工作，则已经退出的子进程将成为僵尸进程(defunct),在系统中如果存在的僵尸（zombie）进程过多，将会影响系统的性能，所以必须对僵尸进程进行处理。

例：僵尸进程

```
pid_t pid=fork();
if(pid == 0){
exit(0);
}else{
while(1); //子进程都已经退出了，但是父进程没有回收。ps-elf 可以看到僵尸进程。
}
```

避免僵尸进程(wait 函数):

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

pid_t waitpid(pid_t pid, int *status, int options); //opeions 若为 WNOHANG，则父进程运行到这，看看有僵尸就回收，没僵尸就直接走了，往下运行，不等了。

例 1：

```
pid_t pid;
pid=fork();
if(pid == 0){
sleep(5);
exit(0);
}else{
wait(NULL); //子进程 sleep(5)期间,可以看到父进程的状态时阻塞(sleep),等待子进程结束。等子进程结束之后回收资源。
exit(0); wait(NULL)表示等待所有进程。
}
```

例 2：

```
pid_t pid;
pid=fork();
if(pid == 0){
sleep(5);
exit(0);
}else{
waitpid(pid,NULL,0); //等待指定 pid 的子进程。
exit(0);
}
}
```

例 3：

```
pid_t pid;
pid=fork();
if(pid == 0){
while(1);
exit(2);  //如果进程正常结束，返回 2。如果不正常的话(比如，ctrl+c)，返回 0.
}else{
int status; //status 存放子进程的退出状态，status 含有很多信息，其中就包括返回值。
```

```
wait(&status);
printf("I am wake\n");
if(WIFEXITED(status)){//这个宏用来获取状态信息。如果进程正常结束，则结果非零。
printf("the exit value=%d\n",WEXITSTATUS(status));  //这个宏返回子进程的退出码。这个例子中是 2.
}else{
printf("the child abort\n"); //如果 WIFEXITED(status)==0，说明不是正常退出。
}
exit(0);
}
```

2.2 信号的机制、种类（5 分）

一. 信号机制

信号机制指的是内核如何向一个进程发送信号、进程如何接收一个信号、进程怎样控制自己对信号的反应、内核在什么时机处理和怎样处理进程收到的信号。还要介绍一下 setjmp 和 longjmp 在信号中起到的作用。

1、内核对信号的基本处理方法

内核给一个进程发送软中断信号的方法，是在进程所在的进程表项的信号域设置对应于该信号的位。这里要补充的是，如果信号发送给一个正在睡眠的进程，那么要看 该进程进入睡眠的优先级，如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位，而不唤醒进程。这一点比较重要，因为进程检查是否收到信号的时机是：一个进程在即将从内核态返回到用户态时；或者，在一个进程要进入或离开一个适当的低调度优先级睡眠状态时。

内核处理一个进程收到的信号的时机是在一个进程从内核态返回用户态时。所以，当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。

内核处理一个进程收到的软中断信号是在该进程的上下文中，因此，进程必须处于运行状态。前面介绍概念的时候讲过，处理信号有三种类型：进程接收到信号后退出；进程忽略该信号；进程收到信号后执行用户设定用系统调用 signal 的函数。当进程接收到一个它忽略的信号时，进程丢弃该信号，就象没有收到该信号似的继续运行。如果进程收到一个要捕捉的信号，那么进程从内核态返回用户

态时执行用户定义的函数。而且执行用户定义的函数的方法很巧妙，内核是在用户栈上创建一个新的层，该层中将返回地址的值设置成用户定义的处理函数的地址，这样进程从内核返回弹出栈顶时就返回到用户定义的函数处，从函数返回再弹出栈顶时，才返回原先进入内核的地方。这样做的原因是用户定义的处理函数不能且不允许在内核态下执行（如果用户定义的函数在内核态下运行的话，用户就可以获得任何权限）。

在信号的处理方法中有几点特别要引起注意。第一，在一些系统中，当一个进程处理完中断信号返回用户态之前，内核清除用户区中设定的对该信号的处理例程的地址，即下一次进程对该信号的处理方法又改为默认值，除非在下一次信号到来之前再次使用 signal 系统调用。这可能会使得进程 在调用 signal 之前又得到该信号而导致退出。在 BSD 中，内核不再清除该地址。但不清除该地址可能使得进程因为过多过快的得到某个信号而导致堆栈溢 出。为了避免出现上述情况。在 BSD 系统中，内核模拟了对硬件中断的处理方法，即在处理某个中断时，阻止接收新的该类中断。

第二个要引起注意的是，如果要捕捉的信号发生于进程正在一个系统调用中时，并且该进程睡眠在可中断的优先级上，这时该信号引起进程作一次 longjmp，跳出睡眠状态，返回用户态并执行信号处理例程。当从信号处理例程返回时，进程就象从系统调用返回一样，但返回了一个错误代码，指出该次系统调用曾经被中断。这要注意的是，BSD 系统中内核可以自动地重新开始系统调用。

第三个要注意的地方：若进程睡眠在可中断的优先级上，则当它收到一个要忽略的信号时，该进程被唤醒，但不做 longjmp，一般是继续睡眠。但用户感觉不到进程曾经被唤醒，而是象没有发生过该信号一样。

第四个要注意的地方：内核对子进程终止（SIGCHLD）信号的处理方法与其他信号有所区别。当进程检查出收到了一个子进程终止的信号时，缺省情况下，该进程就像没有收到该信号似的，如果父进程执行了系统调用 wait，进程将从系统调用 wait 中醒来并返回 wait 调用，执行一系列 wait 调用的后续操作（找出僵死的子进程，释放子进程的进程表项），然后从 wait 中返回。SIGCHLD 信号的作用是唤醒一个睡眠在可被中断优先级上的进程。如果该进程捕捉了这个信号，就像普通信号处理一样转到处理例程。如果进程忽略该信号，那么系统调用 wait 的动作就有所不同，因为 SIGCHLD 的作用仅仅是唤醒一个睡眠在可被中断优先级上的进程，那么执行 wait 调用的父进程被唤醒继续执行 wait 调用的后续操作，然后等待其他的子进程。

如果一个进程调用 signal 系统调用，并设置了 SIGCLD 的处理方法，并且该进程有子进程处于僵死状态，则内核将向该进程发一个 SIGCLD 信号。

2、setjmp 和 longjmp 的作用

前面在介绍信号处理机制时，多次提到了 setjmp 和 longjmp，但没有仔细说明它们的作用和实现方法。这里就此作一个简单的介绍。

在介绍信号的时候，我们看到多个地方要求进程在检查收到信号后，从原来的系统调用中直接返回，而不是等到该调用完成。这种进程突然改变其上下文的情况，就是使用 setjmp 和 longjmp 的结果。setjmp 将保存的上下文存入用户区，并继续在旧的上下文中执行。这就是说，进程执行一个系统调用，当因为资源或其他原因要去睡眠时，内核为进程作了一次 setjmp，如果在睡眠中被信号唤醒，进程不能再进入睡眠时，内核为进程调用 longjmp，该操作是内核 为进程将原先 setjmp 调用保存在进程用户区的上下文恢复成现在的上下文，这样就使得进程可以恢复等待资源前的状态，而且内核为 setjmp 返回 1，使得进程知道该次系统调用失败。这就是它们的作用。

二. 信号的种类

linux 信号有如下的种类：

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

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig)	
函数传入值	pid:	正数：要发送信号的进程号
		0：信号被发送到所有和当前进程在同一个进程组的进程
		-1：信号发给所有的进程表中的进程（除了进程号最大的进程
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig：信号	
函数返回值	成功：0	
	出错：-1	

raise()函数语法要点：

所需头文件	#include <signal.h> #include <sys/types.h>
函数原型	int raise(int sig)
函数传入值	sig：信号
函数返回值	成功：0
	出错：-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替

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：

所需头文件	#include <unistd.h>
函数原型	unsigned int alarm(unsigned int seconds)
函数传入值	seconds：指定秒数，系统经过 seconds 秒之后向该进程发送 SIGALRM 信号
函数返回值	成功：如果调用此 alarm()前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回 0
	出错：-1

pause()函数语法如下：

所需头文件	#include <unistd.h>
函数原型	int pause(void)
函数返回值	-1，并且把 error 值设为 EINTR

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：

sigemptyset – 初始化 set 为空集合

sigfillset – 把每个信号都添加到 set 中

sigaddset – 把指定的信号 signum 添加到 set

sigdelset – 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）

一. shell 定义

Shell 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

二. shell 功能

实际上 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

三. shell 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：评估用户刚输入的命令行。如果用户请求了内置命令（quit，jobs，bg 或 fg），则立即执行。 否则，fork 一个子进程并在子进程的上下文中运行该作业。如果作业在前台运行，等待它终止然后返回。

参　　数：形参 用户输入的命令行 cmdline 。实参 argv , mask , SIG_UNBLOCK , argv[0] , environ

处理流程：
第一步：定义各个变量。使用 parseline()函数函数解析命令行，得到命令行参数。
第二步：使用 builtin_cmd()函数判断命令是否为内置命令，如果不是内置命令，则继续执行。
第三步：设置阻塞集合。先初始化 mask 为空集合，再将 SIGCHLD , SIGINT ,SIGTSTP 信号加入阻塞集合。
第四步：阻塞 SIGCHLD，防止子进程在父进程之前结束，防止 addjob()函数错误地把（不存在的）子进程添加到作业列表中。
第五步：子进程中，先解除对 SIG_CHLD 的阻塞，再使用 setpgid(0,0)创建一个虚拟的进程组，进程组 ID 是 15213，不和 tsh 进程在一个进程组。然后调用 execve 函数，执行相应的文件。
第六步：将 job 添加到 job list，解除 SIG_CHLD 阻塞信号。判断进程是否为前台进程，如果是前台进程，调用 waitfg()函数，等待前台进程，如果是后台进程，则打印出进程信息。

要点分析：
1.每个子进程必须具有唯一的进程组 ID，以便当我们在键盘上键入 ctrl-c（ctrl-z）时，我们的后台子进程不会从内核接收 SIGINT（SIGTSTP）
2.在执行 addjob 之前需要阻塞信号，防止 addjob()函数错误地把（不存在的）子进程添加到作业列表中。

3. 1.2 int builtin_cmd(char **argv)函数（5 分）

```
int builtin_cmd(char **argv)
{
    sigset_t mask,prev_mask;
    sigfillset(&mask);
    if(!strcmp(argv[0],"quit"))    //退出命令
        exit(0);
    else if(!strcmp(argv[0],"&"))      //忽略单独的&
        return 1;
    else if(!strcmp(argv[0],"jobs"))  //输出作业列表中所有作业信息
    {
        sigprocmask(SIG_BLOCK,&mask,&prev_mask); //访问全局变量，阻塞所有信号
        listjobs(jobs);
        sigprocmask(SIG_SETMASK,&prev_mask,NULL);
        return 1;
    }
    else if(!strcmp(argv[0],"bg")||!strcmp(argv[0],"fg"))  //实现bg和fg两条内置命令
    {
        do_bgfg(argv);
        return 1;
    }
    return 0;     /* not a builtin command */
}
```

函数功能：如果用户键入了内置命令，则立即执行。

参　　数：形参 传入的参数数组 argv 。实参 argv[0] ， "quit" ， "&" ， "jobs" ， "bg" ， "fg"， argv
SIG_BLOCK ， mask ， prev_mask ， NULL

处理流程：
函数需要根据传入的参数数组，判断用户键入的是否为内置命令，采取的办法就是比较 argv[0]和内置命令，如果是内置命令，则跳到相应的函数，并且返回 1，如果不是，则什么也不做，并且返回 0。

其中，如果命令为 quit，则直接退出；如果命令是内置的 jobs 命令，则调用 listjobs()函数，打印 job 列表；如果是 fg 或是 bg 两条内置命令，则调用 do_bgfg()函数来处理即可。

要点分析：因为 jobs 是全局变量，为了防止其被修改，需要阻塞全部信号，过程大致为（后面函数阻塞全部信号的做法与此基本一致）：

```
sigset_t mask,prev_mask;
sigfillset(&mask);

...

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

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

函数功能：执行内置 bg 和 fg 命令
参　　数:形参 传入的参数数组 argv 。实参 argv[1] , argv[1][0] , jobs , pid , argv[1] , argv[0]

处理流程：
第一步：先判断 fg 或 bg 后是否有参数，如果没有，则忽略命令。
第二步：如果 fg 或 bg 后面只是数字，说明取的是进程号，获取该进程号后，使用 getjobpid(jobs, pid)得到 job；如果 fg 或 bg 后面是%加上数字的形式，说明%后面是任务号(第几个任务)，此时获取 jid 后，可以使用 getjobjid(jobs, jid)得到 job 。
第三步：比较区分 argv[0]是“bg”还是“fg”。如果是后台进程，则发送 SIGCONT 信号给进程组 PID 的每个进程，并且设置任务的状态为 BG，打印任务的 jid，pid 和命令行；如果是前台进程，则发送 SIGCONT 信号给进程组 PID 的每个进程，并且设置任务的状态为 FG，调用 waitfg(jobp->pid)，等待前台进程结束。

要点分析：
1.函数主要是先判断 fg 后面是%+数字还是只有数字的形式，从而根据进程号 pid 或是工作组号 jid 来获取结构体 job；然后在根据前台和后台进程的不同，执行相应的操作。
2. isdigit()函数判断是否为数字，不是数字返回 0
3. atoi()函数把字符串转化为整型数
4. SIGCONT 信号对应事件为：继续进程如果该进程停止。

3. 1.4 void waitfg(pid_t pid) 函数（5 分）

```
void waitfg(pid_t pid) //传入的是一个前台进程的pid
{
    sigset_t mask;
    sigemptyset(&mask);  //初始化mask为空集合
    while(pid==fgpid(jobs))
    {
        sigsuspend(&mask);  //暂时挂起
    }
}
```

函数功能：阻止直到进程 pid 不再是前台进程
参　　数：形参 前台进程 pid 。实参 mask ， jobs 。

处理流程：
函数主体是 while 循环语句，判断传入的 pid 是否为一个前台进程的 pid，如果是，则一直循环，如果不是，则跳出循环。其中 while 循环内部使用 sigsuspend()函数，

暂时用 mask 替换当前的阻塞集合，然后挂起该进程，直到收到一个信号，选择运行一个处理程序或者终止该进程。

要点分析：
1.在 while 内部，如果使用的只是 pause()函数，那么程序必须等待相当长的一段时间才会再次检查循环的终止条件，如果使用向 nanosleep 这样的高精度休眠函数也是不可接受的，因为没有很好的办法来确定休眠的间隔。
2.在 while 循环语句之前，初始化 mask 结合为空，在 while 内部用 SIG_SETMASK 使 block=mask，这样 sigsuspend()才不会因为收不到 SIGCHLD 信号而永远睡眠。

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

```
void sigchld_handler(int sig)
{
    struct job_t *job1;   //新建结构体
    int olderrno = errno,status;
    sigset_t mask, prev_mask;
    pid_t pid;
    sigfillset(&mask);
    while((pid=waitpid(-1,&status,WNOHANG|WUNTRACED))>0)
    {
        /*尽可能回收子进程，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*/
        sigprocmask(SIG_BLOCK,&mask,&prev_mask);  //需要deletejob，阻塞所有信号
        job1 = getjobpid(jobs,pid);  //通过pid找到job
        if(WIFSTOPPED(status)) //子进程停止引起的waitpid函数返回
        {
            job1->state = ST;
            printf("Job [%d] (%d) terminated by signal %d\n",job1->jid,job1->pid,WSTOPSIG(status));
        }
        else
        {
            if(WIFSIGNALED(status) //子进程终止引起的返回
                printf("Job [%d] (%d) terminated by signal %d\n",job1->jid,job1->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;
}
```

函数功能：只要子作业终止（变为僵尸），或者因为收到 SIGSTOP 或 SIGTSTP 信号而停止，内核就会向 shell 发送 SIGCHLD。处理程序收集所有可用的僵尸子节点，但不等待任何其他当前正在运行的子节点终止。

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

处理流程：
第一步：把每个信号都添加到 mask 阻塞集合中，设置 olderrno = errno 。
第二步：在 while 循环中使用 waitpid(-1,&status,WNOHANG|WUNTRACED))，其中，目的是尽可能回收子进程，其中 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。

第三步：在循环中阻塞信号，并且使用 getjobpid()函数，通过 pid 找到 job 。
第四步：通过 waitpid 在 status 中放上的返回子进程的状态信息，判断子进程的退出状态。如果引起返回的子进程当前是停止的，那么 WIFSTOPPED(status)就返回真，此时只需要将 pid 找到的 job 的状态改为 ST，并且按照示例程序输出的信息，将 job 的 jid，pid 以及导致子进程停止的信号的编号输出即可。如果子进程是因为一个未被捕获的信号终止的，那么 WIFSIGNALED(status)就返回真，此时同样按照示例程序输出的信息，将 job 的 jid，pid 以及导致子进程终止的信息的编号输出即可，因为此时进程是中止的的进程，所以还需要 deletejob()将发出 SIGCHLD 信号的将其直接回收。
第五步：清空缓冲区，解除阻塞，恢复 errno 。

要点分析：
1.while 循环来避免信号阻塞的问题，循环中使用 waitpid()函数，以尽可能多的回收僵尸进程。
2.调用 deletejob()函数时，因为 jobs 是全局变量，因此需要阻塞信号。
3. 通过 waitpid 在 status 中放上的返回子进程的状态信息,判断子进程的退出状态。WIFSIGNALED 判断子进程是否因为一个未被捕获的信号中止的，WIFSTOPPED 判断引起返回地子进程当前是否为停止的。

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); //清空缓冲区并输出
    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  解除阻塞*/
      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.
   builtin_cmd   - 如果用户键入了内置命令，则立即执行。
 */
int builtin_cmd(char **argv)
{
    sigset_t mask,prev_mask;
    sigfillset(&mask);
    if(!strcmp(argv[0],"quit"))     //退出命令
        exit(0);
    else if(!strcmp(argv[0],"&"))        //忽略单独的&
        return 1;
    else if(!strcmp(argv[0],"jobs"))   //输出作业列表中所有作业信息
    {
        sigprocmask(SIG_BLOCK,&mask,&prev_mask); //访问全局变量，阻塞所有信号
        listjobs(jobs);
        sigprocmask(SIG_SETMASK,&prev_mask,NULL);
        return 1;
    }
    else if(!strcmp(argv[0],"bg")||!strcmp(argv[0],"fg"))   //实现 bg 和 fg 两条内置命令
    {
        do_bgfg(argv);
        return 1;
    }
    return 0;        /* not a builtin command */
}

/*
 * do_bgfg - Execute the builtin bg and fg commands
    执行内置 bg 和 fg 命令
 */
void do_bgfg(char **argv)
```

```
{
    /* $begin handout */
    struct job_t *jobp=NULL;

    /* Ignore command if no argument
    如果没有参数，则忽略命令*/
    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])) //判断串 1 的第 0 位是否为数字
    {
  pid_t pid = atoi(argv[1]);    //atoi 把字符串转化为整型数
  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) //传入的是一个前台进程的 pid
{
    sigset_t mask;
    sigemptyset(&mask);   //初始化 mask 为空集合
    while(pid==fgpid(jobs))
    {
        sigsuspend(&mask);   //暂时挂起
    }
}

/******************
 * 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.
 sigchld_handler  - 只要子作业终止（变为僵尸），
```

内核就会向 shell 发送 SIGCHLD，或者因为收到 SIGSTOP 或 SIGTSTP 信号而停止。

处理程序收集所有可用的僵尸子节点，但不等待任何其他当前正在运行的子节点终止。

*/

```
void sigchld_handler(int sig)
{
    struct job_t *job1;     //新建结构体
    int olderrno = errno,status;
    sigset_t mask, prev_mask;
    pid_t pid;
    sigfillset(&mask);
    while((pid=waitpid(-1,&status,WNOHANG|WUNTRACED))>0)
    {
        /*尽可能回收子进程，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*/
        sigprocmask(SIG_BLOCK,&mask,&prev_mask);   //需要 deletejob，阻塞所有信号
        job1 = getjobpid(jobs,pid);   //通过 pid 找到 job
        if(WIFSTOPPED(status)) //子进程停止引起的 waitpid 函数返回
        {
            job1->state = ST;
            printf("Job [%d] (%d) terminated by signal %d\n",job1->jid,job1->pid,WSTOPSIG(status));
        }
        else
        {
            if(WIFSIGNALED(status) //子进程终止引起的返回
                printf("Job [%d] (%d) terminated by signal %d\n",job1->jid,job1->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;
}

/*
 * 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.
  sigint handler  - 只要用户在键盘上键入 ctrl-c，
 内核就会向 shell 发送一个 SIGINT。 抓住它并将其发送到  前台作业。
 */
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.
 sigtstp_handler  - 只要用户在键盘上键入 ctrl-z，
 内核就会向 shell 发送 SIGTSTP。 通过向它发送 SIGTSTP 来捕获它并暂停前台作业。
 */
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
qwj@qwj-virtual-machine:~/shlab-handout-hit$ make test01 ./sdriver.pl -t trace01.txt -s ./tsh -a "-p" # # trace01.txt - Properly terminate on EOF. # qwj@qwj-virtual-machine:~/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：

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
qwj@qwj-virtual-machine:~/shlab-handout-hit$ make test02 ./sdriver.pl -t trace02.txt -s ./tsh -a "-p" # # trace02.txt - Process builtin quit command. # qwj@qwj-virtual-machine:~/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：

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
qwj@qwj-virtual-machine:~/shlab-handout-hit$ make test03 ./sdriver.pl -t trace03.txt -s ./tsh -a "-p" # # trace03.txt - Run a foreground job. # tsh> quit qwj@qwj-virtual-machine:~/shlab-handout-hit$ make rtest03 ./sdriver.pl -t trace03.txt -s ./tshref -a "-p" # # trace03.txt - Run a foreground job. # tsh> quit	
测试结论	相同/不同，原因分析如下：

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
qwj@qwj-virtual-machine:~/shlab-handout-hit$ make test04 ./sdriver.pl -t trace04.txt -s ./tsh -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 & [1] (10678) ./myspin 1 & qwj@qwj-virtual-machine:~/shlab-handout-hit$ make rtest04 ./sdriver.pl -t trace04.txt -s ./tshref -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 & [1] (10684) ./myspin 1 &	
测试结论	相同/不同，原因分析如下：

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
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make test05 ./sdriver.pl -t trace05.txt -s ./tsh -a "-p" # # trace05.txt - Process jobs builtin command. # tsh> ./myspin 2 & [1] (12408) ./myspin 2 & tsh> ./myspin 3 & [2] (12410) ./myspin 3 & tsh> jobs [1] (12408) Running ./myspin 2 & [2] (12410) Running ./myspin 3 &	

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make rtest05
./sdriver.pl -t trace05.txt -s ./tshref -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (12437) ./myspin 2 &
tsh> ./myspin 3 &
[2] (12439) ./myspin 3 &
tsh> jobs
[1] (12437) Running ./myspin 2 &
[2] (12439) Running ./myspin 3 &
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/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] (11382) terminated by signal 2
```

```
qwj@qwj-virtual-machine:~/hitics/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] (11373) terminated by signal 2
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make test07
./sdriver.pl -t trace07.txt -s ./tsh -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (12472) ./myspin 4 &
tsh> ./myspin 5
Job [2] (12474) terminated by signal 2
tsh> jobs
[1] (12472) Running ./myspin 4 &
```

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make rtest07
./sdriver.pl -t trace07.txt -s ./tshref -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (12461) ./myspin 4 &
tsh> ./myspin 5
Job [2] (12463) terminated by signal 2
tsh> jobs
[1] (12461) Running ./myspin 4 &
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/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] (12794) ./myspin 4 &
tsh> ./myspin 5
Job [2] (12796) terminated by signal 20
tsh> jobs
[1] (12794) Running ./myspin 4 &
[2] (12796) Stopped ./myspin 5
```

```
qwj@qwj-virtual-machine:~/hitics/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] (12805) ./myspin 4 &
tsh> ./myspin 5
Job [2] (12807) stopped by signal 20
tsh> jobs
[1] (12805) Running ./myspin 4 &
[2] (12807) Stopped ./myspin 5
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (12828) ./myspin 4 &
tsh> ./myspin 5
Job [2] (12830) terminated by signal 20
tsh> jobs
[1] (12828) Running ./myspin 4 &
[2] (12830) Stopped ./myspin 5
tsh> bg %2
[2] (12830) ./myspin 5
tsh> jobs
[1] (12828) Running ./myspin 4 &
[2] (12830) Running ./myspin 5
```

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make rtest09
./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (12814) ./myspin 4 &
tsh> ./myspin 5
Job [2] (12816) stopped by signal 20
tsh> jobs
[1] (12814) Running ./myspin 4 &
[2] (12816) Stopped ./myspin 5
tsh> bg %2
[2] (12816) ./myspin 5
tsh> jobs
[1] (12814) Running ./myspin 4 &
[2] (12816) Running ./myspin 5
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make test10
./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (12849) ./myspin 4 &
tsh> fg %1
Job [1] (12849) terminated by signal 20
tsh> jobs
[1] (12849) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make rtest10
./sdriver.pl -t trace10.txt -s ./tshref -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (12839) ./myspin 4 &
tsh> fg %1
Job [1] (12839) stopped by signal 20
tsh> jobs
[1] (12839) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/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] (12871) terminated by signal 2
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1035 tty7     Ssl+   3:49 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
   1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  10788 pts/4    Ss     0:00 bash
  12866 pts/4    S+     0:00 make test11
  12867 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tsh -a "
-p"
  12868 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh -
a -p
  12869 pts/4    S+     0:00 ./tsh -p
  12874 pts/4    R      0:00 /bin/ps a
```

```
qwj@qwj-virtual-machine:~/hitics/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] (12859) terminated by signal 2
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1035 tty7     Ssl+   3:49 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
   1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  10788 pts/4    Ss     0:00 bash
  12854 pts/4    S+     0:00 make rtest11
  12855 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tshref -
a "-p"
  12856 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tshre
f -a -p
  12857 pts/4    S+     0:00 ./tshref -p
  12863 pts/4    R      0:00 /bin/ps a
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/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] (12880) terminated by signal 20
tsh> jobs
[1] (12880) Stopped ./mysplit 4
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1035 tty7     Ssl+   3:49 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
   1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  10788 pts/4    Ss     0:00 bash
  12875 pts/4    S+     0:00 make test12
  12876 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tsh -a "
-p"
  12877 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tsh -
a -p
  12878 pts/4    S+     0:00 ./tsh -p
  12880 pts/4    T      0:00 ./mysplit 4
  12881 pts/4    T      0:00 ./mysplit 4
  12884 pts/4    R      0:00 /bin/ps a
```

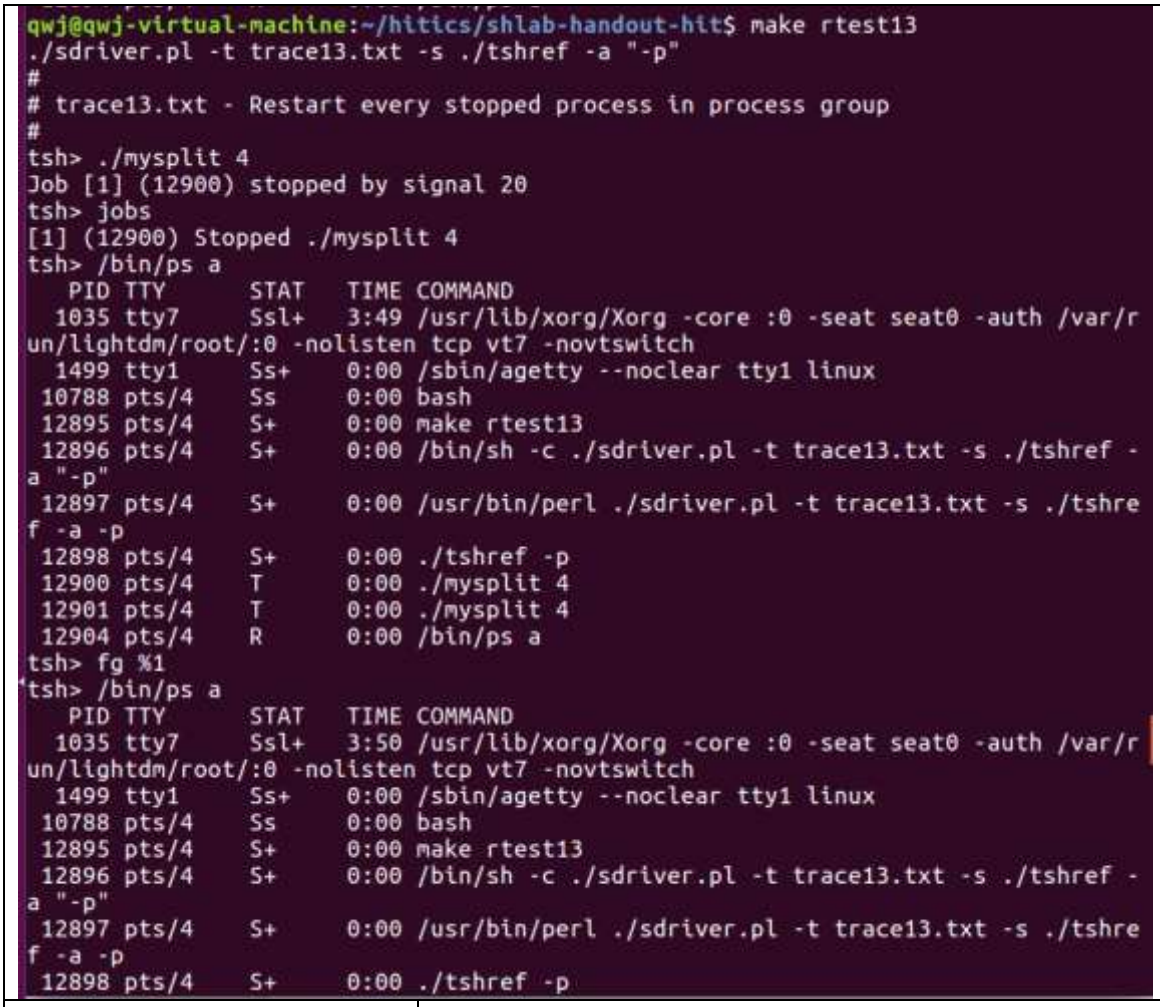
```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make rtest12
./sdriver.pl -t trace12.txt -s ./tshref -a "-p"
#
# trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (12890) stopped by signal 20
tsh> jobs
[1] (12890) Stopped ./mysplit 4
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
  1035 tty7     Ssl+   3:49 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
  1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
 10788 pts/4    Ss     0:00 bash
 12885 pts/4    S+     0:00 make rtest12
 12886 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tshref -
a "-p"
 12887 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tshre
f -a -p
 12888 pts/4    S+     0:00 ./tshref -p
 12890 pts/4    T      0:00 ./mysplit 4
 12891 pts/4    T      0:00 ./mysplit 4
 12894 pts/4    R      0:00 /bin/ps a
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/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] (12913) terminated by signal 20
tsh> jobs
[1] (12913) Stopped ./mysplit 4
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1035 tty7     Ssl+   3:50 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
   1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  10788 pts/4    Ss     0:00 bash
  12908 pts/4    S+     0:00 make test13
  12909 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "
-p"
  12910 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -
a -p
  12911 pts/4    S+     0:00 ./tsh -p
  12913 pts/4    T      0:00 ./mysplit 4
  12914 pts/4    T      0:00 ./mysplit 4
  12917 pts/4    R      0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1035 tty7     Ssl+   3:50 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
   1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  10788 pts/4    Ss     0:00 bash
  12908 pts/4    S+     0:00 make test13
  12909 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "
-p"
  12910 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -
a -p
  12911 pts/4    S+     0:00 ./tsh -p
  12920 pts/4    R      0:00 /bin/ps a
```



```
qwj@qwj-virtual-machine:~/hitics/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] (12900) stopped by signal 20
tsh> jobs
[1] (12900) Stopped ./mysplit 4
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1035 tty7     Ssl+   3:49 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
   1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  10788 pts/4    Ss     0:00 bash
  12895 pts/4    S+     0:00 make rtest13
  12896 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tshref -
a "-p"
  12897 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshre
f -a -p
  12898 pts/4    S+     0:00 ./tshref -p
  12900 pts/4    T      0:00 ./mysplit 4
  12901 pts/4    T      0:00 ./mysplit 4
  12904 pts/4    R      0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   1035 tty7     Ssl+   3:50 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/r
un/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
   1499 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  10788 pts/4    Ss     0:00 bash
  12895 pts/4    S+     0:00 make rtest13
  12896 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tshref -
a "-p"
  12897 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshre
f -a -p
  12898 pts/4    S+     0:00 ./tshref -p
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/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] (12948) ./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] (12948) terminated by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (12948) ./myspin 4 &
tsh> jobs
[1] (12948) Running ./myspin 4 &
```

```
qwj@qwj-virtual-machine:~/hitics/shlab-handout-hit$ make rtest14
./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (12929) ./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] (12929) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (12929) ./myspin 4 &
tsh> jobs
[1] (12929) Running ./myspin 4 &
```

测试结论	相同/不同，原因分析如下：

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

```
qwj@qwj-virtual-machine:~/hitics/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] (12967) terminated by signal 2
tsh> ./myspin 3 &
[1] (12971) ./myspin 3 &
tsh> ./myspin 4 &
[2] (12973) ./myspin 4 &
tsh> jobs
[1] (12971) Running ./myspin 3 &
[2] (12973) Running ./myspin 4 &
tsh> fg %1
Job [1] (12971) terminated by signal 20
tsh> jobs
[1] (12971) Stopped ./myspin 3 &
[2] (12973) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (12971) ./myspin 3 &
tsh> jobs
[1] (12971) Running ./myspin 3 &
[2] (12973) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

```
qwj@qwj-virtual-machine:~/hitics/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] (12990) terminated by signal 2
tsh> ./myspin 3 &
[1] (12992) ./myspin 3 &
tsh> ./myspin 4 &
[2] (12994) ./myspin 4 &
tsh> jobs
[1] (12992) Running ./myspin 3 &
[2] (12994) Running ./myspin 4 &
tsh> fg %1
Job [1] (12992) stopped by signal 20
tsh> jobs
[1] (12992) Stopped ./myspin 3 &
[2] (12994) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (12992) ./myspin 3 &
tsh> jobs
[1] (12992) Running ./myspin 3 &
[2] (12994) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

测试结论	相同/不同，原因分析如下：

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. 通过书写系统级用户界面 tsh，更好地理解了 shell 的工作流程以及处理方法。
2. 深入理解了信号的机制以及各种发送，接受和阻塞方法。

4.2 请给出对本次实验内容的建议

个人认为 eval 函数和 do_bgfg 函数不必全部给出，可以在其中留一些空缺，让学生独立补充，因为这个两个函数比较重要，通过对他们的书写，可以加深对 shell 的理解。

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.