哈爾濱工業大學

实验报告

实 验（七）

题　　目 TinyShell

微壳

专　　业 计算机系

学　　号 1190201421

班　　级 1936603

学　　生 张瑞

指 导 教 师 刘宏伟

实 验 地 点 G709

实 验 日 期 2021 年 6 月 3 日

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

2.1 进程的概念、创建和回收方法

概念：**进程的经典定义就是一个执行中的程序的实例。**系统的每一个程序都是运行在某一个进程上下文中。上下文是由程序正确运行所需要的状态构成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及上下文描述符的集合。

创建：每次用户通过向 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，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

父进程通过调用 fork 函数来创建一个新的运行的子进程。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用户级虚拟地址相同的（但是是独立的）一个副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用 fork 时，子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程最大的不同是他们有不同的 PID。

回收：当父进程回收已经终止的子进程时，内核将子进程的退出状态传递给父进程，然后抛弃已经终止的进程，从此时开始，该进程就不存在了。**一个进程可以通过调用 waitpid（或者 wait）函数来等待它的子进程终止或者停止**。通过使用不同的参数来改变函数等待集合、行为和返回效果。

2.2 信号的机制、种类

机制：Linux 信号允许进程和内核中断其他进程。一个信号就是一条小消息，它通知进程系统中发生了一个某种类型的事件。传送一个信号到目的进程由两个不同步骤组成：发送信号和接收信号。发送信号指内核通过更新目的进程上下文的某种状态，发送一个信号给目的进程。发送信号的原因可能是内核检测到一个系统事件，或一个进程调用了 kill 函数。接收信号指内核强迫目的进程以某种方式对信号的发送做出反应，则接收了这种信号。进程可以忽略、终止或捕获信号。

种类：Linux 系统上支持 30 种不同类型的信号，如下：

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

2.3 信号的发送方法、阻塞方法、处理程序的设置方法

发送方法：

Unix 系统提供了大量向进程发送信号的机制：

用 bin/kill 程序发送信号（发送 SIGKILL 给进程或者进程组）

从键盘发送信号（ctrl+c：发送 SIGINT 给前台进程组的每个进程，ctrl+z：发送 SIGSTP 给前台进程组的每个进程）

用 kill 函数发送信号（发送 SIGKILL 给指定进程组的每个进程）

用 alarm 函数发送信号（发送 SIGALRM 给调用进程）

还有很多，比如子进程终止后会发送 SIGCHLD 给父进程等..

阻塞方法：

Linux 提供阻塞信号隐式和显式的机制：

隐式阻塞机制：内核默认阻塞任何当前处理程序正在处理信号类型的待处理的信号。

显式阻塞机制：系统可以通过使用 sigpromask 函数和他的辅助函数，明确的

阻塞和解除阻塞选定的信号。

处理程序的设置方法：

可以使用 signal 函数修改和信号 signum 相关联的默认行为，参数 handler 的不同取值代表不同行为:

SIG_IGN: 忽略类型为 signum 的信号

SIG_DFL: 类型为 signum 的信号行为恢复为默认行为

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，就会调用这个程序。通过把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。当处理程序执行 return 时，控制传递回控制流中被信号接收所中断的指令处，信号处理完成。

2.4 什么是 shell，功能和处理流程

Shell 是一个命令解释程序，作为 Unix 的一个重要组成部分,是它的外壳，也是用户与 Unix 系统的交互作用界面。

Shell 作为外壳，一个解释程序，一个人机接口，负责接受、翻译用户或程序发出的各种命令，然后把这个翻译出来指令传给内核，代表用户运行，内核接受成功或者失败执行相应操作返回运行结果，然后再经由 shell 求值组织翻译显示给用户交互显示。系统启动后，内核为每个终端用户建立一个进程去执行 shell 解释程序。

Shell 的基本的执行步骤如下：

1.读取用户由键盘输入的命令行。

2.分析命令，以命令名作为文件名，并将其它参数改造为系统调用 execve( ) 内部处理所要求的形式。

3.终端进程调用 fork( )建立一个子进程。

4.终端进程本身调用 wait( )来等待子进程完成（如果是后台命令，则不等待）。当子进程运行时调用 execve( )，子进程根据文件名到目录中查找有关文件，调入内存，执行这个程序。

5.如果命令末尾有&,则终端进程不用执行系统调用 wait( )，立即发提示符，让用户输入下一条命令；否则终端进程会一直等待，当子进程完成工作后，向父进程报告，此时中断进程醒来，作必要的判别工作后，终端发出命令提示符，重复上述处理过程。

第 3 章 TinyShell 的设计与实现

3.1 设计

3.1.1 void eval(char *cmdline)函数

函数功能：对用户输入的命令行进行解析并计算。
首先解析该 cmdline 字符串，配置 argv，得到前后台信息，然后判断这是否为一个内置命令（quit, jobs, bg or fg)。如果用户输入内置命令那么立即执行；否则，fork 一个新的子进程并在其上下文中运行。如果该作业是前台作业那么需要等到它运行结束才返回；若是后台作业则打印进程信息。
参　　数：char *cmdline
处理流程：
1.解析命令行，配置好 argv，并且得到前后台信息。
2.若指令为空，则直接返回。
3.判断是否为内置命令，如果是，则立即执行。
4.不是内置命令，则设置阻塞信号集合并 fork 一个子进程：
（1）在子进程中：解除信号阻塞，获得新的进程组 ID 并执行指令。
（2）在父进程中：先把子进程按前后台的分类加入 jobs（进程组）中，解除信号阻塞。然后判断如果是前台作业，则等待前台作业结束，如果是后台作业，则打印出相关进程信息。
要点分析：
1. 在键盘上输入 ctrl-c(ctrl-z)时，前台进程组的每一个进程都会从内核接收到 SIGINT(SIGTSTP)信号，导致前台作业的终止（停止）。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.需要设置阻塞信号集合，让信号可以被发送但不会被接收，这样就能够避免父进程与信号处理程序出现竞争，不会出现子进程先对 job 各种操作，然后再执行父进程 addjob 的问题。

3.1.2 int builtin_cmd(char **argv)函数

函数功能：判断命令行是否为内置命令，如果是则立即执行相应的操作（返回值为 1），否则返回 0。
参　　数：char **argv
处理流程：读取 argv[0]并进行比对，若是内置命令（jobs,fg,bg,quit），则执行相应的命令，返回值为 1；否则不执行操作，返回 0。注意要忽略单独的“&”。
要点分析：
1.用 strcmp 判断命令行与内置命令是否匹配，若 strcmp 的返回值为 0，则匹配。
2.（1）内置命令：

quit→exit(0)直接退出，不返回
jobs→用 listjobs(jobs)显示 jobs 信息，返回值为 1
fg/bg→用 do_bgfg(argv)在前台或后台运行，返回值为 1
（2）非内置命令：返回值为 0

3.1.3 void do_bgfg(char **argv) 函数

函数功能：处理 bg 和 fg 这两个内置命令。
参　　数：char **argv
处理流程：
1.没有 pid 和 jobid 时，忽略该命令，直接返回。
2.解析对应的 pid 和 jobid，找到对应的作业，若找不到则返回。
3.bg 命令是后台运行，改变作业的状态标签为 bg，输出相关信息，然后返回。
4.fg 命令是前台运行，改变作业的状态标签为 fg，等待前台程序退出，然后返回。
要点分析：
1.获取进程号 pid 或者作业号 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，再根据进程号或作业号来获取作业，然后在前台或后台执行相关操作。
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

函数功能：阻塞进程，直到它不再是当前的前台进程。
参　　数：pid_t pid
处理流程：传入函数的 pid 等于此时前台进程的 pid，则让其一直在循环中挂起（时间设为 0）;否则返回。
要点分析：
1.调用 fgpid 函数向 jobs 查询当前 state 是 FG 的 job 的 PID。
2.使用 sleep（0）使进程实时检查循环终止条件。

3.1.5 void sigchld_handler(int sig) 函数

函数功能：处理 SIGCHLD 信号。回收所有僵死进程但不等待其他正在运行的子进程终止。
参　　数：int sig
处理流程：
1.首先设置 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.按 PPT 提示，在 while 循环条件判断处使用 pid = waitpid(-1, &status, WNOHANG | WUNTRACED) ，WNOHANG | WUNTRACED 使得 waitpid 能立即返回，如果等待集合中没有子进程被中止或停止，返回 0；否则返回子进程的 pid。
3.循环体中，因为不等待其他正在运行的子进程终止，需要暂时阻塞信号，模板如

下（省略对返回值的检查）：
sigset_t mask, prev_mask;
sigfillset(&mask);

…

sigprocmask(SIG_BLOCK,&mask,&prev_mask);
sigprocmask(SIG_SETMASK,&prev_mask,NULL);
4.同时用 job = getjobpid(jobs, pid)得到对应的 job
5.判断引起 waitpid 函数返回的原因：
（1）WIFSTOPPED(status)为真：子进程停止，输出相关信息。
（2）WIFEXITED(status)为真：子进程正常终止，执行 deletejob(jobs, pid)进行回收。
（3）上述两个条件都为假（WIFSIGNALED(status)为真）：子进程是因为未捕获的信号而终止，输出相关信息，然后执行 deletejob(jobs, pid)进行回收。
5. 执行 fflush(stdout)和 sigprocmask(SIG_SETMASK, &prev_mask, NULL)来清空缓冲区并且解除阻塞。
6.恢复 errno = olderrno。
要点分析：
1.waitpid 使用 WNOHANG|WUNTRACED 的组合，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，否则返回进程的 pid。这样做的目的是尽可能的回收子进程,使得如果当前进程都没有终止时，直接返回，而不是挂起该回收进程。
2.通过检验 status 的值来执行不同操作：
（1）WIFEXITED(status): 函数是通过调用 exit 或者 return 正常终止的，返回 true。
（2）WIFSIGNALED(status)：如果进程因为一个未捕获信号而终止的，返回 true。
（3）WIFSTOPPED(status)：如果进程当前是停止的，返回 true。
3.保存和恢复 errno ，防止并发问题。
4.阻塞信号和解除阻塞的模板如下（省略对返回值的检查）：
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）

（1） /*

（2） * tsh - A tiny shell program with job control

（3） *

（4） * 张瑞 1190201421

```
（5）    */
（6）    #include <stdio.h>
（7）    #include <stdlib.h>
（8）    #include <unistd.h>
（9）    #include <string.h>
（10）   #include <ctype.h>
（11）   #include <signal.h>
（12）   #include <sys/types.h>
（13）   #include <sys/wait.h>
（14）   #include <errno.h>
（15）
（16）   /* Misc manifest constants */
（17）   #define MAXLINE     1024    /* max line size */
（18）   #define MAXARGS      128    /* max args on a command line */
（19）   #define MAXJOBS       16    /* max jobs at any point in time */
（20）   #define MAXJID    1<<16    /* max job ID */
（21）
（22）   /* Job states */
（23）   #define UNDEF 0 /* undefined */
（24）   #define FG 1    /* running in foreground */
（25）   #define BG 2    /* running in background */
（26）   #define ST 3    /* stopped */
（27）
（28）   /*
（29）    * Jobs states: FG (foreground), BG (background), ST (stopped)
（30）    * Job state transitions and enabling actions:
（31）    *     FG -> ST  : ctrl-z
（32）    *     ST -> FG  : fg command
（33）    *     ST -> BG  : bg command
（34）    *     BG -> FG  : fg command
（35）    * At most 1 job can be in the FG state.
（36）    */
```

```
（37）
（38） /* Global variables */
（39） extern char **environ;          /* defined in libc */
（40） char prompt[] = "tsh> ";       /* command line prompt (DO NOT CHANGE) */
（41） int verbose = 0;                /* if true, print additional output */
（42） int nextjid = 1;                /* next job ID to allocate */
（43） char sbuf[MAXLINE];             /* for composing sprintf messages */
（44）
（45） struct job_t {                  /* The job struct */
（46）     pid_t pid;                  /* job PID */
（47）     int jid;                    /* job ID [1, 2, ...] */
（48）     int state;                  /* UNDEF, BG, FG, or ST */
（49）     char cmdline[MAXLINE];      /* command line */
（50） };
（51） struct job_t jobs[MAXJOBS]; /* The job list */
（52） /* End global variables */
（53）
（54）
（55） /* Function prototypes */
（56）
（57） /* Here are the functions that you will implement */
（58） void eval(char *cmdline);
（59） int builtin_cmd(char **argv);
（60） void do_bgfg(char **argv);
（61） void waitfg(pid_t pid);
（62）
（63） void sigchld_handler(int sig);
（64） void sigtstp_handler(int sig);
（65） void sigint_handler(int sig);
（66）
（67） /* Here are helper routines that we've provided for you */
```

```
（68） int parseline(const char *cmdline, char **argv);
（69） void sigquit_handler(int sig);
（70）
（71） void clearjob(struct job_t *job);
（72） void initjobs(struct job_t *jobs);
（73） int maxjid(struct job_t *jobs);
（74） int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
（75） int deletejob(struct job_t *jobs, pid_t pid);
（76） pid_t fgpid(struct job_t *jobs);
（77） struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
（78） struct job_t *getjobjid(struct job_t *jobs, int jid);
（79） int pid2jid(pid_t pid);
（80） void listjobs(struct job_t *jobs);
（81）
（82） void usage(void);
（83） void unix_error(char *msg);
（84） void app_error(char *msg);
（85） typedef void handler_t(int);
（86） handler_t *Signal(int signum, handler_t *handler);
（87）
（88） /*
（89）  * main - The shell's main routine
（90）  */
（91） int main(int argc, char **argv)
（92） {
（93）     char c;
（94）     char cmdline[MAXLINE];
（95）     int emit_prompt = 1; /* emit prompt (default) */
（96）
（97）     /* Redirect stderr to stdout (so that driver will get all output
（98）      * on the pipe connected to stdout) */
（99）     dup2(1, 2);
```

```
（100）
（101）     /* Parse the command line */
（102）     while ((c = getopt(argc, argv, "hvp")) != EOF) {
（103）         switch (c) {
（104）         case 'h':             /* print help message */
（105）             usage();
（106）       break;
（107）         case 'v':             /* emit additional diagnostic info */
（108）             verbose = 1;
（109）       break;
（110）         case 'p':             /* don't print a prompt */
（111）             emit_prompt = 0;  /* handy for automatic testing */
（112）       break;
（113）   default:
（114）             usage();
（115）   }
（116）     }
（117）
（118）     /* Install the signal handlers */
（119）
（120）     /* These are the ones you will need to implement */
（121）     Signal(SIGINT,  sigint_handler);   /* ctrl-c */
（122）     Signal(SIGTSTP, sigtstp_handler);  /* ctrl-z */
（123）     Signal(SIGCHLD, sigchld_handler);  /* Terminated or stopped child */
（124）
（125）     /* This one provides a clean way to kill the shell */
（126）     Signal(SIGQUIT, sigquit_handler);
（127）
（128）     /* Initialize the job list */
（129）     initjobs(jobs);
（130）
```

```
（131）        /* Execute the shell's read/eval loop */
（132）        while (1) {
（133）
（134）    /* Read command line */
（135）    if (emit_prompt) {
（136）            printf("%s", prompt);
（137）            fflush(stdout);
（138）    }
（139）    if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
（140）            app_error("fgets error");
（141）    if (feof(stdin)) { /* End of file (ctrl-d) */
（142）            fflush(stdout);
（143）            exit(0);
（144）    }
（145）
（146）    /* Evaluate the command line */
（147）    eval(cmdline);
（148）    fflush(stdout);
（149）    fflush(stdout);
（150）        }
（151）
（152）        exit(0); /* control never reaches here */
（153）  }
（154）
（155）  /*
（156）   * eval - Evaluate the command line that the user has just typed in
（157）   *
（158）   * If the user has requested a built-in command (quit, jobs, bg or fg)
（159）   * then execute it immediately. Otherwise, fork a child process and
（160）   * run the job in the context of the child. If the job is running in
（161）   * the foreground, wait for it to terminate and then return.    Note:
（162）   * each child process must have a unique process group ID so that our
```

```
（163）  * background children don't receive SIGINT (SIGTSTP) from the kernel
（164）  * when we type ctrl-c (ctrl-z) at the keyboard.
（165）  */
（166）  void eval(char *cmdline)
（167）  {
（168）      /* $begin handout */
（169）      char *argv[MAXARGS]; /* argv for execve() */
（170）      int bg;                  /* should the job run in bg or fg? */
（171）      pid_t pid;               /* process id */
（172）      sigset_t mask;           /* signal mask */
（173）
（174）      /* Parse command line */
（175）      bg = parseline(cmdline, argv);
（176）      if (argv[0] == NULL)
（177）  return;     /* ignore empty lines */
（178）
（179）      if (!builtin_cmd(argv)) {
（180）
（181）          /*
（182）   * This is a little tricky. Block SIGCHLD, SIGINT, and SIGTSTP
（183）   * signals until we can add the job to the job list. This
（184）   * eliminates some nasty races between adding a job to the job
（185）   * list and the arrival of SIGCHLD, SIGINT, and SIGTSTP signals.
（186）   */
（187）
（188）  if (sigemptyset(&mask) < 0)
（189）          unix_error("sigemptyset error");
（190）  if (sigaddset(&mask, SIGCHLD))
（191）          unix_error("sigaddset error");
（192）  if (sigaddset(&mask, SIGINT))
（193）          unix_error("sigaddset error");
（194）  if (sigaddset(&mask, SIGTSTP))
```

```
（195）         unix_error("sigaddset error");
（196）    if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
（197）         unix_error("sigprocmask error");
（198）
（199）    /* Create a child process */
（200）    if ((pid = fork()) < 0)
（201）         unix_error("fork error");
（202）
（203）    /*
（204）     * Child    process
（205）     */
（206）
（207）    if (pid == 0) {
（208）         /* Child unblocks signals */
（209）         sigprocmask(SIG_UNBLOCK, &mask, NULL);
（210）
（211）         /* Each new job must get a new process group ID
（212）              so that the kernel doesn't send ctrl-c and ctrl-z
（213）              signals to all of the shell's jobs */
（214）         if (setpgid(0, 0) < 0)
（215）          unix_error("setpgid error");
（216）
（217）         /* Now load and run the program in the new job */
（218）         if (execve(argv[0], argv, environ) < 0) {
（219）          printf("%s: Command not found\n", argv[0]);
（220）          exit(0);
（221）         }
（222）    }
（223）
（224）    /*
（225）     * Parent process
```

```
（226）     */
（227）
（228）    /* Parent adds the job, and then unblocks signals so that
（229）        the signals handlers can run again */
（230）    addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
（231）    sigprocmask(SIG_UNBLOCK, &mask, NULL);
（232）
（233）    if (!bg)
（234）          waitfg(pid);
（235）    else
（236）          printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
（237）        }
（238）        /* $end handout */
（239）        return;
（240）   }
（241）
（242）  /*
（243）    * parseline - Parse the command line and build the argv array.
（244）    *
（245）    * Characters enclosed in single quotes are treated as a single
（246）    * argument.    Return true if the user has requested a BG job, false if
（247）    * the user has requested a FG job.
（248）    */
（249）  int parseline(const char *cmdline, char **argv)
（250）  {
（251）        static char array[MAXLINE]; /* holds local copy of command line */
（252）        char *buf = array;                  /* ptr that traverses command line */
（253）        char *delim;                        /* points to first space delimiter */
（254）        int argc;                           /* number of args */
（255）        int bg;                             /* background job? */
（256）
（257）        strcpy(buf, cmdline);
```

```
（258）        buf[strlen(buf)-1] = ' ';    /* replace trailing '\n' with space */
（259）        while (*buf && (*buf == ' ')) /* ignore leading spaces */
（260）    buf++;
（261）
（262）        /* Build the argv list */
（263）        argc = 0;
（264）        if (*buf == '\'') {
（265）    buf++;
（266）    delim = strchr(buf, '\'');
（267）        }
（268）        else {
（269）    delim = strchr(buf, ' ');
（270）        }
（271）
（272）        while (delim) {
（273）    argv[argc++] = buf;
（274）    *delim = '\0';
（275）    buf = delim + 1;
（276）    while (*buf && (*buf == ' ')) /* ignore spaces */
（277）              buf++;
（278）
（279）    if (*buf == '\'') {
（280）          buf++;
（281）          delim = strchr(buf, '\'');
（282）    }
（283）    else {
（284）          delim = strchr(buf, ' ');
（285）    }
（286）        }
（287）        argv[argc] = NULL;
（288）
（289）        if (argc == 0)    /* ignore blank line */
```

```
（290）    return 1;
（291）
（292）        /* should the job run in the background? */
（293）        if ((bg = (*argv[argc-1] == '&')) != 0) {
（294）    argv[--argc] = NULL;
（295）        }
（296）        return bg;
（297）    }
（298）
（299）    /*
（300）     * builtin_cmd - If the user has typed a built-in command then execute
（301）     *       it immediately.
（302）     */
（303）    int builtin_cmd(char **argv)
（304）    {
（305）        if(!strcmp(argv[0],"quit"))        /*如果是 quit，直接退出*/
（306）        {
（307）            exit(0);
（308）        }
（309）        if(!strcmp(argv[0],"&"))       /*忽略单独的&*/
（310）        {
（311）            return 1;
（312）        }
（313）        if(!strcmp(argv[0],"jobs"))        /*如果是 jobs，调用 listjobs 显示各个 job 的状态*/
（314）        {
（315）            listjobs(jobs);
（316）            return 1;
（317）        }
（318）        if(!strcmp(argv[0],"bg") || !strcmp(argv[0],"fg"))     /*如果 fg 或者 bg，则调用 do_bgfg 进行处理即
        可*/
（319）        {
（320）            do_bgfg(argv);
```

```
（321）             return 1;
（322）         }
（323）         return 0;          /* not a builtin command */
（324）  }
（325）
（326）  /*
（327）   * do_bgfg - Execute the builtin bg and fg commands
（328）   */
（329）  void do_bgfg(char **argv)
（330）  {
（331）         /* $begin handout */
（332）         struct job_t *jobp=NULL;
（333）
（334）         /* Ignore command if no argument */
（335）         if (argv[1] == NULL) {
（336）    printf("%s command requires PID or %%jobid argument\n", argv[0]);
（337）    return;
（338）         }
（339）
（340）         /* Parse the required PID or %JID arg */
（341）         if (isdigit(argv[1][0])) {
（342）    pid_t pid = atoi(argv[1]);
（343）    if (!(jobp = getjobpid(jobs, pid))) {
（344）           printf("(%d): No such process\n", pid);
（345）           return;
（346）    }
（347）         }
（348）         else if (argv[1][0] == '%') {
（349）    int jid = atoi(&argv[1][1]);
（350）    if (!(jobp = getjobjid(jobs, jid))) {
（351）           printf("%s: No such job\n", argv[1]);
（352）           return;
```

```
（353）    }
（354）        }
（355）        else {
（356）    printf("%s: argument must be a PID or %%jobid\n", argv[0]);
（357）    return;
（358）        }
（359）
（360）        /* bg command */
（361）        if (!strcmp(argv[0], "bg")) {
（362）    if (kill(-(jobp->pid), SIGCONT) < 0)
（363）          unix_error("kill (bg) error");
（364）    jobp->state = BG;
（365）    printf("[%d] (%d) %s", jobp->jid, jobp->pid, jobp->cmdline);
（366）        }
（367）
（368）        /* fg command */
（369）        else if (!strcmp(argv[0], "fg")) {
（370）    if (kill(-(jobp->pid), SIGCONT) < 0)
（371）          unix_error("kill (fg) error");
（372）    jobp->state = FG;
（373）    waitfg(jobp->pid);
（374）        }
（375）        else {
（376）    printf("do_bgfg: Internal error\n");
（377）    exit(0);
（378）        }
（379）        /* $end handout */
（380）        return;
（381）  }
（382）
（383）  /*
（384）   * waitfg - Block until process pid is no longer the foreground process
```

```
（385）  */
（386）  void waitfg(pid_t pid)
（387）  {
（388）      while(pid == fgpid(jobs))
（389）      {
（390）          sleep(0);     /*一直循环挂起*/
（391）      }
（392）      return;
（393）  }
（394）
（395）  /******************
（396）   * Signal handlers
（397）   ******************/
（398）
（399）  /*
（400）   * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
（401）   *      a child job terminates (becomes a zombie), or stops because it
（402）   *      received a SIGSTOP or SIGTSTP signal. The handler reaps all
（403）   *      available zombie children, but doesn't wait for any other
（404）   *      currently running children to terminate.
（405）   */
（406）  void sigchld_handler(int sig)
（407）  {
（408）      int olderrno = errno,status;      /*  保存 errno*/
（409）      sigset_t mask, prev_mask;
（410）      pid_t pid;
（411）      struct job_t* job;
（412）      if(sigfillset(&mask) < 0)
（413）          unix_error("sigfillset error");
（414）
（415）      while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)      /*回收僵死进程,waitpid
        返回值为正，说明有被终止或停止的子进程*/
```

```
（416）     {
（417）         if (sigprocmask(SIG_BLOCK, &mask, &prev_mask) < 0)
（418）             unix_error("sigpromask error");
（419）         job = getjobpid(jobs, pid);
（420）         if(WIFSTOPPED(status))    /*子进程停止引起 waitpid 函数返回*/
（421）         {
（422）             job->state = ST;       /*将状态更改为 ST*/
（423）             printf("Job [%d] (%d) stopped by signal %d\n", job->jid, job->pid, WSTOPSIG(status));
（424）         }
（425）         else if(WIFEXITED(status))       /*子进程正常终止引起 waitpid 函数返回*/
（426）         {
（427）             deletejob(jobs, pid);     /*回收已终止的进程*/
（428）         }
（429）         else       /*子进程因为未捕获的信号而终止引起 waitpid 函数返回*/
（430）         {
（431）             printf("Job [%d] (%d) terminated by signal %d\n", job->jid, job->pid,
        WTERMSIG(status));
（432）             deletejob(jobs, pid);     /*回收已终止的进程*/
（433）         }
（434）         if (sigprocmask(SIG_SETMASK, &prev_mask, NULL) < 0)
（435）             unix_error("sigpromask error");
（436）     }
（437）     errno = olderrno;     /*恢复 errno*/
（438）     return;
（439） }
（440）
（441） /*
（442）  * sigint_handler - The kernel sends a SIGINT to the shell whenver the
（443）  *      user types ctrl-c at the keyboard.    Catch it and send it along
（444）  *      to the foreground job.
（445）  */
```

```
（446） void sigint_handler(int sig)
（447） {
（448）     int olderrno = errno;
（449）  sigset_t mask, prev_mask;
（450）  if(sigfillset(&mask) < 0)
（451）         unix_error("sigfillset error");
（452）  if (sigprocmask(SIG_BLOCK, &mask, &prev_mask) < 0)   /*需要访问全局变量 jobs，先暂时阻塞信号
        */
（453）             unix_error("sigpromask error");
（454）  pid_t pid = fgpid(jobs);
（455）  if (sigprocmask(SIG_SETMASK, &prev_mask, NULL) < 0)
（456）             unix_error("sigpromask error");
（457）  if (pid != 0)    /*有前台作业时才处理*/
（458）     {
（459）         if (kill(-pid, SIGINT) < 0)
（460）             unix_error("kill error");
（461）     }
（462）  errno = olderrno;
（463）     return;
（464） }
（465）
（466） /*
（467）  * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
（468）  *      the user types ctrl-z at the keyboard. Catch it and suspend the
（469）  *      foreground job by sending it a SIGTSTP.
（470）  */
（471） void sigtstp_handler(int sig)
（472） {
（473）     int olderrno = errno;
（474）  sigset_t mask, prev_mask;
（475）  if(sigfillset(&mask) < 0)
（476）         unix_error("sigfillset error");
```

```
（477）   if (sigprocmask(SIG_BLOCK, &mask, &prev_mask) < 0)   /*需要访问全局变量 jobs，先暂时阻塞信号
          */
（478）                   unix_error("sigpromask error");
（479）   pid_t pid = fgpid(jobs);
（480）   if (sigprocmask(SIG_SETMASK, &prev_mask, NULL) < 0)
（481）                   unix_error("sigpromask error");
（482）   if (pid != 0)     /*有前台作业时才处理*/
（483）        {
（484）              if (kill(-pid, SIGTSTP) < 0)
（485）                   unix_error("kill error");
（486）        }
（487）   errno = olderrno;
（488）        return;
（489）  }
（490）
（491）  /**********************
（492）   * End signal handlers
（493）   **********************/
（494）
（495）  /***********************************************
（496）   * Helper routines that manipulate the job list
（497）   ***********************************************/
（498）
（499）  /* clearjob - Clear the entries in a job struct */
（500）  void clearjob(struct job_t *job) {
（501）        job->pid = 0;
（502）        job->jid = 0;
（503）        job->state = UNDEF;
（504）        job->cmdline[0] = '\0';
（505）  }
（506）
（507）  /* initjobs - Initialize the job list */
```

```
（508） void initjobs(struct job_t *jobs) {
（509）     int i;
（510）
（511）     for (i = 0; i < MAXJOBS; i++)
（512）  clearjob(&jobs[i]);
（513） }
（514）
（515） /* maxjid - Returns largest allocated job ID */
（516） int maxjid(struct job_t *jobs)
（517） {
（518）     int i, max=0;
（519）
（520）     for (i = 0; i < MAXJOBS; i++)
（521）  if (jobs[i].jid > max)
（522）       max = jobs[i].jid;
（523）     return max;
（524） }
（525）
（526） /* addjob - Add a job to the job list */
（527） int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)
（528） {
（529）     int i;
（530）
（531）     if (pid < 1)
（532）  return 0;
（533）
（534）     for (i = 0; i < MAXJOBS; i++) {
（535）  if (jobs[i].pid == 0) {
（536）       jobs[i].pid = pid;
（537）       jobs[i].state = state;
（538）       jobs[i].jid = nextjid++;
```

```
（539）          if (nextjid > MAXJOBS)
（540）           nextjid = 1;
（541）          strcpy(jobs[i].cmdline, cmdline);
（542）                 if(verbose){
（543）                printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid, jobs[i].cmdline);
（544）                    }
（545）                    return 1;
（546）    }
（547）        }
（548）        printf("Tried to create too many jobs\n");
（549）        return 0;
（550）  }
（551）
（552）  /* deletejob - Delete a job whose PID=pid from the job list */
（553）  int deletejob(struct job_t *jobs, pid_t pid)
（554）  {
（555）        int i;
（556）
（557）        if (pid < 1)
（558）    return 0;
（559）
（560）        for (i = 0; i < MAXJOBS; i++) {
（561）    if (jobs[i].pid == pid) {
（562）          clearjob(&jobs[i]);
（563）          nextjid = maxjid(jobs)+1;
（564）          return 1;
（565）    }
（566）        }
（567）        return 0;
（568）  }
（569）
（570）  /* fgpid - Return PID of current foreground job, 0 if no such job */
```

```
（571） pid_t fgpid(struct job_t *jobs) {
（572）     int i;
（573）
（574）     for (i = 0; i < MAXJOBS; i++)
（575）  if (jobs[i].state == FG)
（576）       return jobs[i].pid;
（577）     return 0;
（578） }
（579）
（580） /* getjobpid    - Find a job (by PID) on the job list */
（581） struct job_t *getjobpid(struct job_t *jobs, pid_t pid) {
（582）     int i;
（583）
（584）     if (pid < 1)
（585）  return NULL;
（586）     for (i = 0; i < MAXJOBS; i++)
（587）  if (jobs[i].pid == pid)
（588）       return &jobs[i];
（589）     return NULL;
（590） }
（591）
（592） /* getjobjid    - Find a job (by JID) on the job list */
（593） struct job_t *getjobjid(struct job_t *jobs, int jid)
（594） {
（595）     int i;
（596）
（597）     if (jid < 1)
（598）  return NULL;
（599）     for (i = 0; i < MAXJOBS; i++)
（600）  if (jobs[i].jid == jid)
（601）       return &jobs[i];
（602）     return NULL;
```

```
（603） }
（604）
（605） /* pid2jid - Map process ID to job ID */
（606） int pid2jid(pid_t pid)
（607） {
（608）     int i;
（609）
（610）     if (pid < 1)
（611）  return 0;
（612）     for (i = 0; i < MAXJOBS; i++)
（613）  if (jobs[i].pid == pid) {
（614）             return jobs[i].jid;
（615）         }
（616）     return 0;
（617） }
（618）
（619） /* listjobs - Print the job list */
（620） void listjobs(struct job_t *jobs)
（621） {
（622）     int i;
（623）
（624）     for (i = 0; i < MAXJOBS; i++) {
（625）  if (jobs[i].pid != 0) {
（626）       printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
（627）       switch (jobs[i].state) {
（628）        case BG:
（629）             printf("Running ");
（630）             break;
（631）        case FG:
（632）             printf("Foreground ");
（633）             break;
（634）        case ST:
```

```
（635）                printf("Stopped ");
（636）                break;
（637）            default:
（638）                printf("listjobs: Internal error: job[%d].state=%d ",
（639）                    i, jobs[i].state);
（640）            }
（641）            printf("%s", jobs[i].cmdline);
（642）    }
（643）        }
（644）  }
（645）  /*********************************
（646）   * end job list helper routines
（647）   *********************************/
（648）
（649）
（650）  /************************
（651）   * Other helper routines
（652）   ************************/
（653）
（654）  /*
（655）   * usage - print a help message
（656）   */
（657）  void usage(void)
（658）  {
（659）      printf("Usage: shell [-hvp]\n");
（660）      printf("   -h   print this message\n");
（661）      printf("   -v   print additional diagnostic information\n");
（662）      printf("   -p   do not emit a command prompt\n");
（663）      exit(1);
（664）  }
（665）
（666）  /*
```

```
（667）  * unix_error - unix-style error routine
（668）  */
（669）  void unix_error(char *msg)
（670）  {
（671）      fprintf(stdout, "%s: %s\n", msg, strerror(errno));
（672）      exit(1);
（673）  }
（674）
（675）  /*
（676）  * app_error - application-style error routine
（677）  */
（678）  void app_error(char *msg)
（679）  {
（680）      fprintf(stdout, "%s\n", msg);
（681）      exit(1);
（682）  }
（683）
（684）  /*
（685）  * Signal - wrapper for the sigaction function
（686）  */
（687）  handler_t *Signal(int signum, handler_t *handler)
（688）  {
（689）      struct sigaction action, old_action;
（690）
（691）      action.sa_handler = handler;
（692）      sigemptyset(&action.sa_mask); /* block sigs of type being handled */
（693）      action.sa_flags = SA_RESTART; /* restart syscalls if possible */
（694）
（695）      if (sigaction(signum, &action, &old_action) < 0)
（696）   unix_error("Signal error");
（697）      return (old_action.sa_handler);
（698）  }
```

```
（699）
（700） /*
（701）  * sigquit_handler - The driver program can gracefully terminate the
（702）  *       child shell by sending it a SIGQUIT signal.
（703）  */
（704） void sigquit_handler(int sig)
（705） {
（706）       printf("Terminating after receipt of SIGQUIT signal\n");
（707）       exit(1);
（708） }
```

第 4 章 TinyShell 测试

4.1 测试方法

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

4.2 测试结果评价

tsh 与 tshref 的输出在以下两个方面可以不同：

（1）pid

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

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

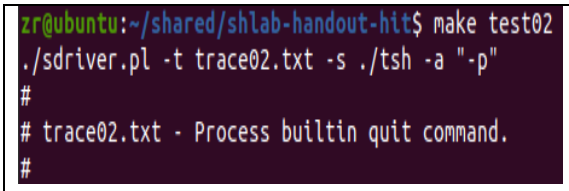
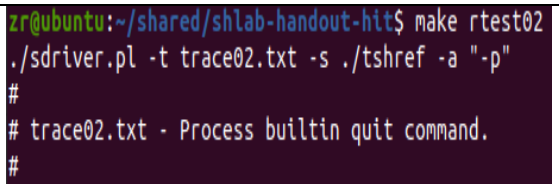
4.3 自测试结果

4.3.1 测试用例 trace01.txt

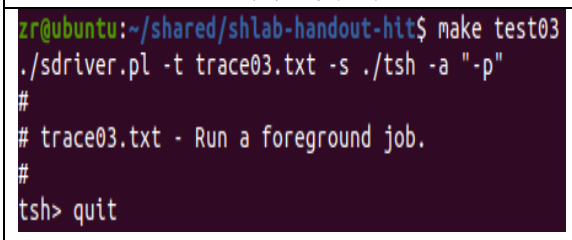
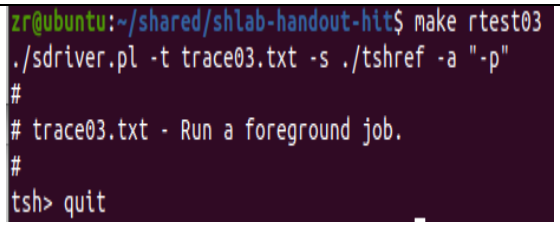
tsh 测试结果	tshref 测试结果
zr@ubuntu:~/shared/shlab-handout-hit$ make test01 ./sdriver.pl -t trace01.txt -s ./tsh -a "-p" # # trace01.txt - Properly terminate on EOF. #	zr@ubuntu:~/shared/shlab-handout-hit$ make rtest01 ./sdriver.pl -t trace01.txt -s ./tshref -a "-p" # # trace01.txt - Properly terminate on EOF. #
测试结论	相同

4.3.2 测试用例 trace02.txt

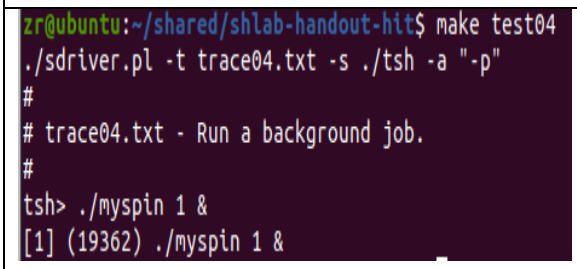
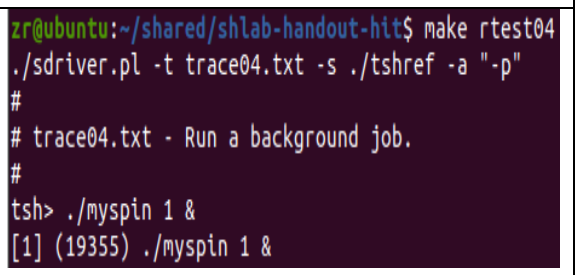
tsh 测试结果	tshref 测试结果

	
测试结论	相同

4.3.3 测试用例 trace03.txt

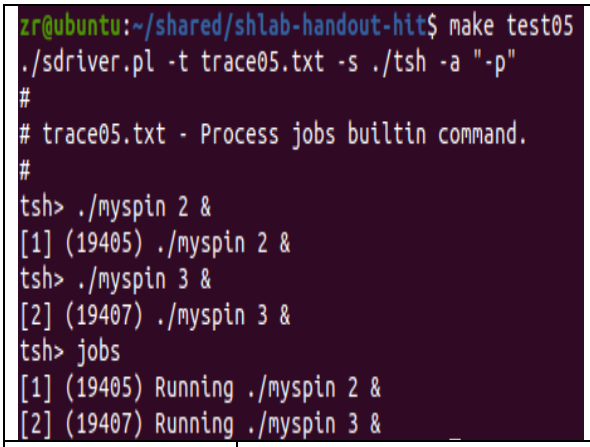
tsh 测试结果	tshref 测试结果
	
测试结论	相同

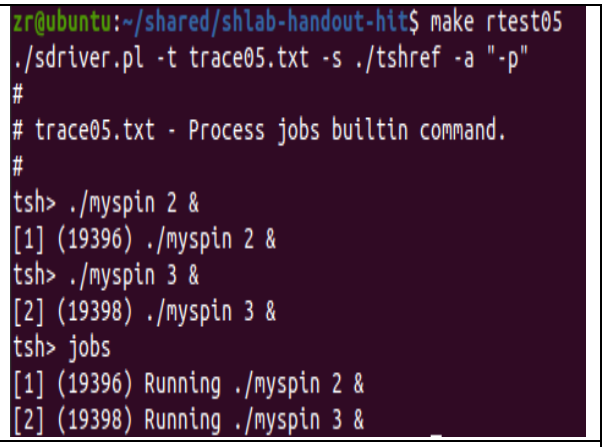
4.3.4 测试用例 trace04.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.5 测试用例 trace05.txt

tsh 测试结果	tshref 测试结果

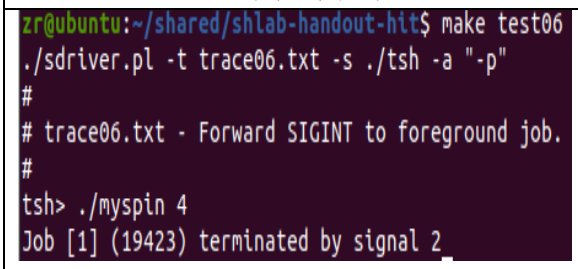


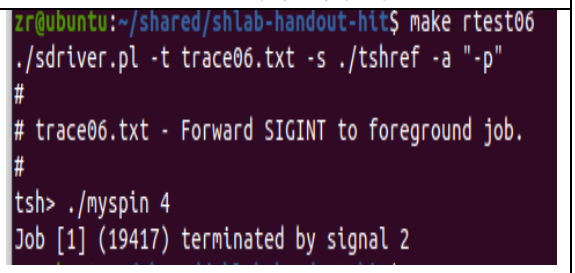


测试结论	相同

4.3.6 测试用例 trace06.txt

tsh 测试结果	tshref 测试结果

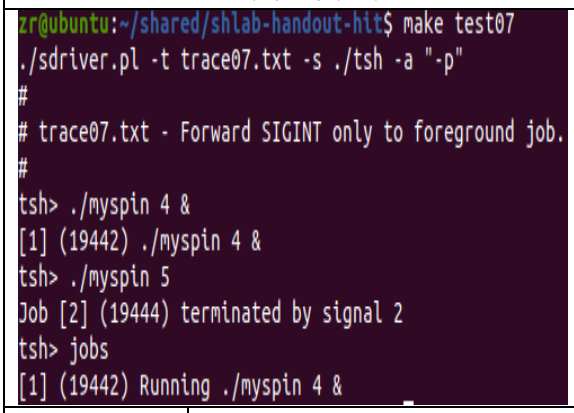


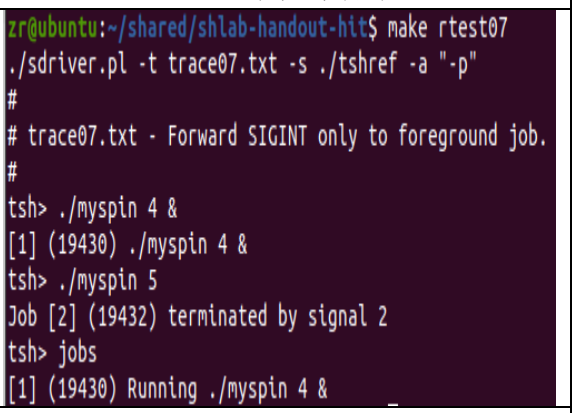


测试结论	相同

4.3.7 测试用例 trace07.txt

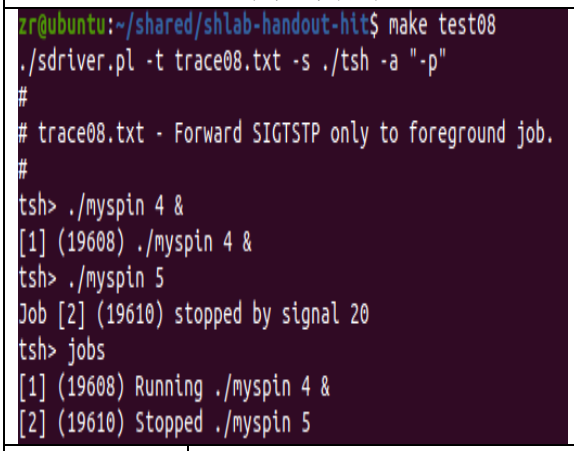
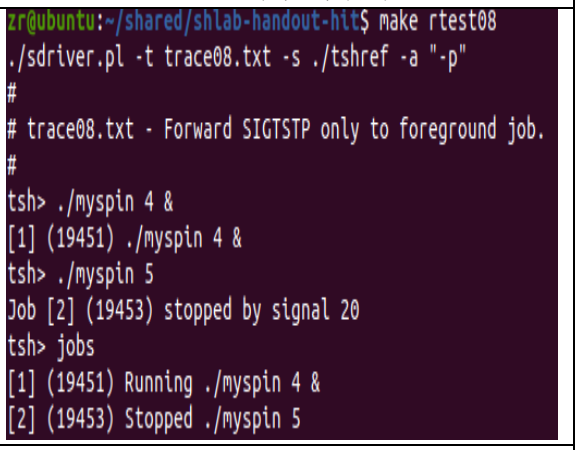
tsh 测试结果	tshref 测试结果



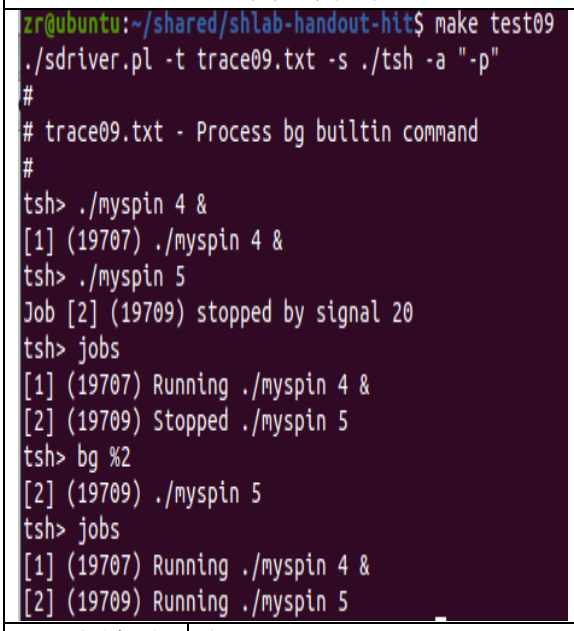


测试结论	相同

4.3.8 测试用例 trace08.txt

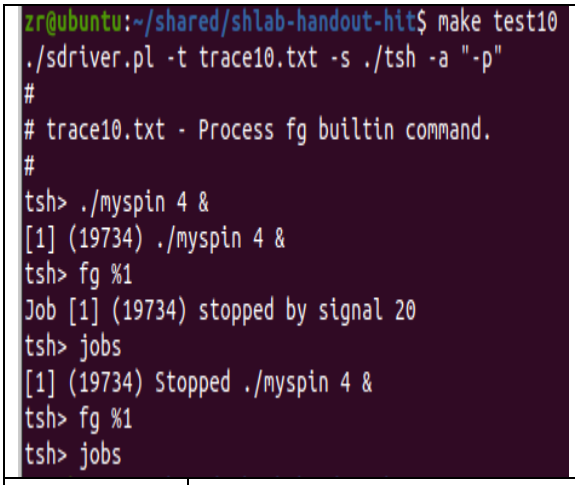
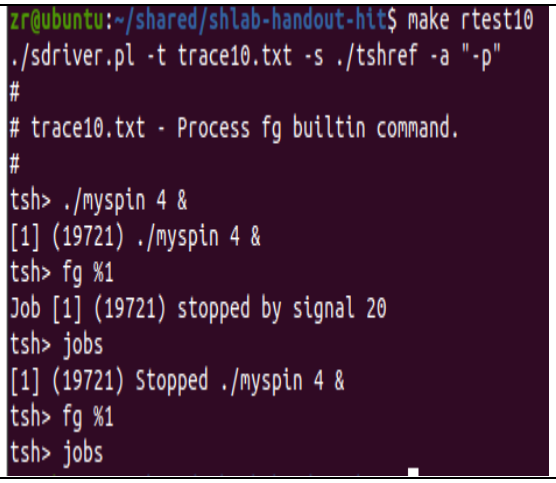
tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.9 测试用例 trace09.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

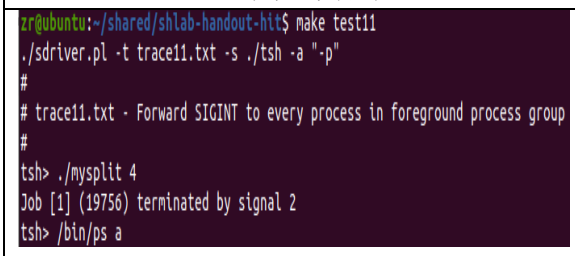
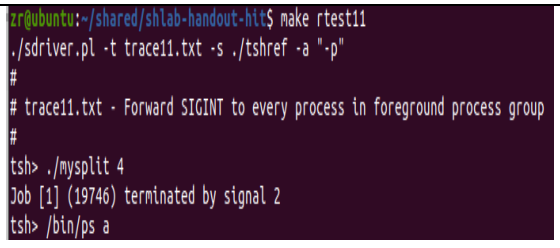
4.3.10 测试用例 trace10.txt

tsh 测试结果	tshref 测试结果

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>测试结论</td><td colspan="2">相同</td></tr>
</table>

4.3.11 测试用例 trace11.txt

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

<table>
<tr><td>tsh 测试结果</td><td>tshref 测试结果</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>测试结论</td><td colspan="2">相同</td></tr>
</table>

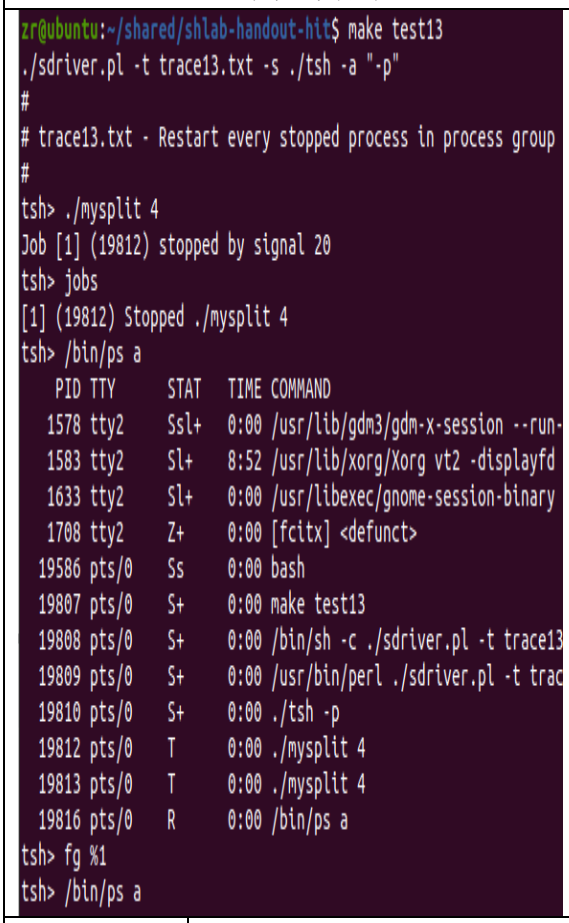
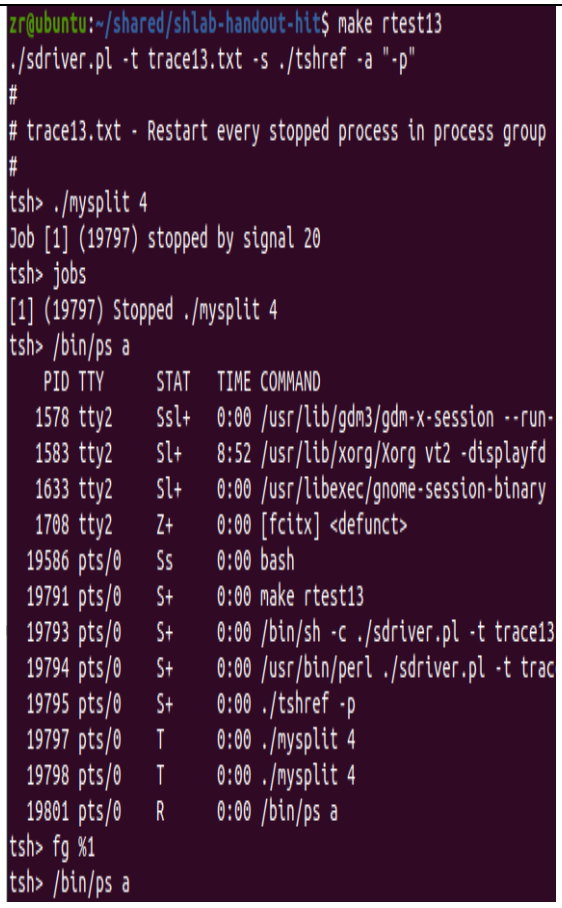
4.3.12 测试用例 trace12.txt

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

<table>
<tr><td>tsh 测试结果</td><td>tshref 测试结果</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>测试结论</td><td colspan="2">相同</td></tr>
</table>

4.3.13 测试用例 trace13.txt

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

tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.14 测试用例 trace14.txt

tsh 测试结果	tshref 测试结果

<table>
<tr>
<td>

```
zr@ubuntu:~/shared/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] (19848) ./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] (19848) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (19848) ./myspin 4 &
tsh> jobs
[1] (19848) Running ./myspin 4 &
```

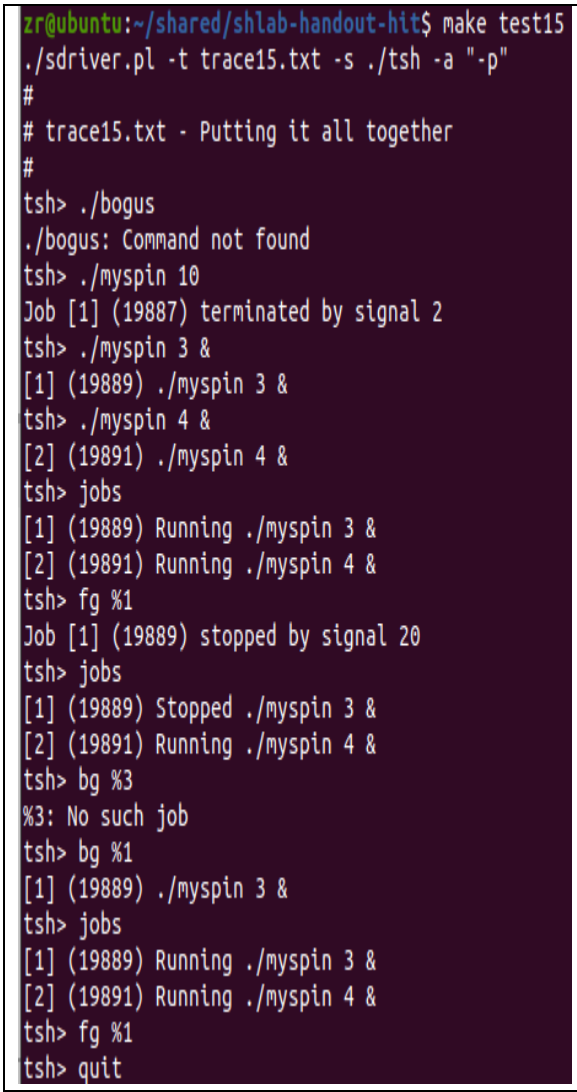
</td>
<td>

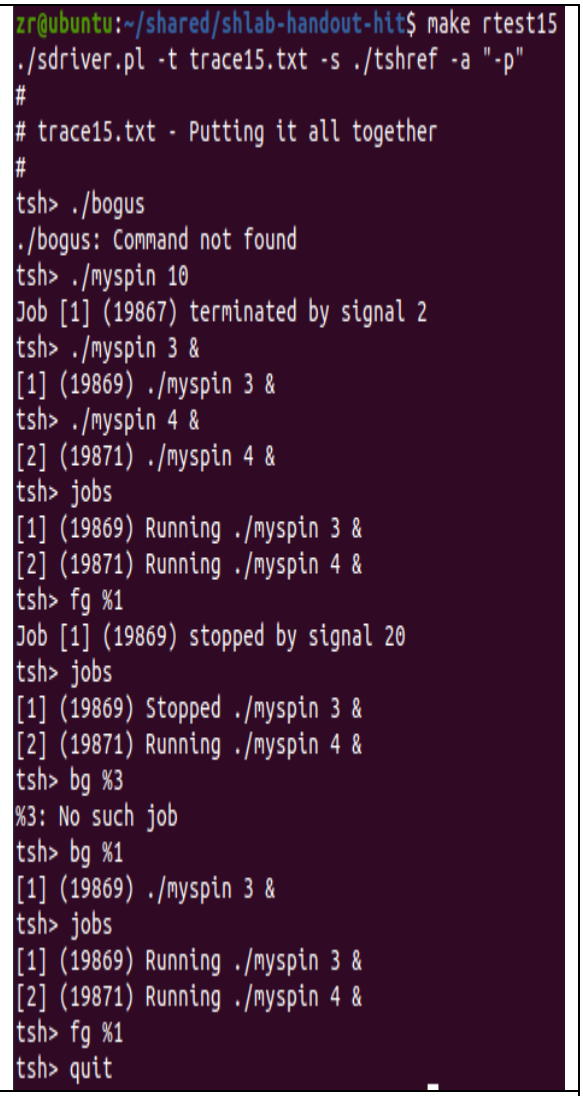
```
zr@ubuntu:~/shared/shlab-handout-hit$ make rtest14
./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (19828) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires 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] (19828) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (19828) ./myspin 4 &
tsh> jobs
[1] (19828) Running ./myspin 4 &
```

</td>
</tr>
<tr><td>测试结论</td><td colspan="2">相同</td></tr>
</table>

4.3.15 测试用例 trace15.txt

tsh 测试结果	tshref 测试结果


zr@ubuntu:~/shared/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] (19887) terminated by signal 2
tsh> ./myspin 3 &
[1] (19889) ./myspin 3 &
tsh> ./myspin 4 &
[2] (19891) ./myspin 4 &
tsh> jobs
[1] (19889) Running ./myspin 3 &
[2] (19891) Running ./myspin 4 &
tsh> fg %1
Job [1] (19889) stopped by signal 20
tsh> jobs
[1] (19889) Stopped ./myspin 3 &
[2] (19891) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (19889) ./myspin 3 &
tsh> jobs
[1] (19889) Running ./myspin 3 &
[2] (19891) Running ./myspin 4 &
tsh> fg %1
tsh> quit


zr@ubuntu:~/shared/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] (19867) terminated by signal 2
tsh> ./myspin 3 &
[1] (19869) ./myspin 3 &
tsh> ./myspin 4 &
[2] (19871) ./myspin 4 &
tsh> jobs
[1] (19869) Running ./myspin 3 &
[2] (19871) Running ./myspin 4 &
tsh> fg %1
Job [1] (19869) stopped by signal 20
tsh> jobs
[1] (19869) Stopped ./myspin 3 &
[2] (19871) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (19869) ./myspin 3 &
tsh> jobs
[1] (19869) Running ./myspin 3 &
[2] (19871) Running ./myspin 4 &
tsh> fg %1
tsh> quit

测试结论	相同

第 5 章 评测得分

实验程序统一测试的评分（教师评价）：

（1）正确性得分：_____

（2）性能加权得分：_____

第 6 章 总结

5.1 请总结本次实验的收获

在本次实验中，我复习了计算机系统进程与并发的相关知识，更加理解了 shell 的工作原理和运行机制，对 Linux 异常控制流和信号机制有了更加熟练的了解与掌握，在实践中理解了 Linux 信号响应可能导致的并发冲突，并尝试写出了解决方案，而且对于 C 语言的代码能力有了一定提高。

5.2 请给出对本次实验内容的建议

建议在 PPT 里对每个待补充函数里需要的操作做更为详细的解释，有些地方确实不太知道需要在函数里干什么，也不太清楚输出的格式要求，需要和 tshref 的输出进行一个对照才知道一些对应的输出格式。

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.