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 1183200123

班　　级 1803003

学　　生 祁 天

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2019.12.4

计算机科学与技术学院

目 录

第 1 章 实验基本信息 - 4 -

1.1 实验目的 - 4 -
1.2 实验环境与工具 - 4 -
1.2.1 硬件环境 - 4 -
1.2.2 软件环境 - 4 -
1.2.3 开发工具 - 4 -
1.3 实验预习 - 4 -

第 2 章 实验预习 - 5 -

2.1 进程的概念、创建和回收方法（5 分） - 5 -
2.2 信号的机制、种类（5 分） - 5 -
2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分） - 5 -
2.4 什么是 SHELL，功能和处理流程（5 分） - 6 -

第 3 章 TINYSHELL 的设计与实现 - 9 -

3.1.1 VOID EVAL(CHAR *CMDLINE)函数（10 分） 错误！未定义书签。
3. 1.2 INT BUILTIN_CMD(CHAR **ARGV)函数（5 分） 错误！未定义书签。
3. 1.3 VOID DO_BGFG(CHAR **ARGV) 函数（5 分） 错误！未定义书签。
3. 1.4 VOID WAITFG(PID_T PID) 函数（5 分） 错误！未定义书签。
3. 1.5 VOID SIGCHLD_HANDLER(INT SIG) 函数（10 分） 错误！未定义书签。

第 4 章 TINYSHELL 测试 - 29 -

4.1 测试方法 - 29 -
4.2 测试结果评价 - 29 -
4.3 自测试结果 - 29 -
4.3.1 测试用例 trace01.txt 的输出截图（1 分） - 29 -
4.3.2 测试用例 trace02.txt 的输出截图（1 分） - 30 -
4.3.3 测试用例 trace03.txt 的输出截图（1 分） - 30 -
4.3.4 测试用例 trace04.txt 的输出截图（1 分） - 30 -
4.3.5 测试用例 trace05.txt 的输出截图（1 分） - 31 -
4.3.6 测试用例 trace06.txt 的输出截图（1 分） - 31 -
4.3.7 测试用例 trace07.txt 的输出截图（1 分） - 31 -
4.3.8 测试用例 trace08.txt 的输出截图（1 分） - 32 -
4.3.9 测试用例 trace09.txt 的输出截图（1 分） - 32 -
4.3.10 测试用例 trace10.txt 的输出截图（1 分） - 32 -
4.3.11 测试用例 trace11.txt 的输出截图（1 分） - 33 -
4.3.12 测试用例 trace12.txt 的输出截图（1 分） - 33 -
4.3.13 测试用例 trace13.txt 的输出截图（1 分） - 33 -

4.3.14 测试用例 trace14.txt 的输出截图（1 分）..... - 33 -
4.3.15 测试用例 trace15.txt 的输出截图（1 分）..... - 34 -
4.4 自测试评分..... - 34 -
第 4 章 总结..... - 36 -
4.1 请总结本次实验的收获..... - 36 -
4.2 请给出对本次实验内容的建议..... - 36 -
参考文献..... - 37 -

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；Code::Blocks；gedit，gcc，notepad++；

1.3 实验预习

填写上实验课前，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）

进程：

一个执行中程序的实例

创建进程：

父进程通过调用 fork()函数创建一个新的运行的子进程

回收进程：

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)。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。

2.2 信号的机制、种类（5 分）

信号的机制：

一个信号就是一条小消息，它通知进程系统中发生了一个某种类型的事件。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。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。

种类：

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（5 分）

信号的发送方法：

1. 用 /bin/kill 程序发送信号， /bin/kill 程序可以向另外的进程或进程组发送任意的信号，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。例如，从键盘上输入 ctrl-c (ctrl-z) 会导致内核发送一个 SIGINT (SIGTSTP)信 号到前台进程组中的每个作业 ；

3. 用 kill 函数发送信号。int kill(pid_t pid,int sig);

如果 pid 大于零，那么 kill 函数发送信号号码 sig 给进程 pid；如果 pid 等于零，那么 kill 发送信号 sig 给调用进程所在进程组中的每个进程，包括调用进程自己；如果 pid 小于零，kill 发送信号 sig 给进程组|pid|（pid 的绝对值）中的每

个进程。

4. 用 alarm 函数发送信号。Unsigned int alarm(unsigned int secs);

alarm 函数安排内核在 secs 秒后发送一个 SIGALRM 信号给调用进程。如果 secs 为零，那么不会调度安排新的闹钟（alarm）。在任何情况下，对 alarm 的调用都将取消任何待处理的闹钟，并且返回任何待处理的闹钟在被发送前还剩下的秒数；如果没有任何待处理的闹钟就返回零。

信号的阻塞方法：Linux 提供阻塞信号的隐式和显式的机制

1. 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。

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

处理程序的设置方法:通过把处理程序的地址传递给 signal 函数从而改变默认行为，这叫做设置信号处理程序。

进程可以使用 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)

signal 函数可以通过下列三种方法之一来改变和信号 signum 相关联的行为：

1. 如果 handler 是 SIG_IGN，那么忽略类型为 signum 的信号；

2. 如果 handler 是 SIG_DFL，那么类型为 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，这个函数被称为信号处理程序，只要接收到一个类型为 signum 的信号，就会调用这个程序。

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：解释输入命令行
参　　数：cmdline
处理流程：

1.判断是否为内部函数，如果是内部函数，直接执行 builtin_cmd 函数就行。如果不是内部函数，需要 Fork 出一个新的进程去执行相应的函数。

2. 函数框架提供了 parseline 命令行解析工具，即可以通过该解析函数来得到命令行参数。如果不是内部函数，首先要先将 SIGCHLD 信号阻塞住，以防出现竞争条件。

3. 子进程解决信号阻塞，并执行相关函数。

4. 父进程要判断子进程是前台进程还是后台进程，如果是前台进程，则调用 waitpid 来等待前台进程，如果是后台进程，则打印出相关进程信息。同时，把新添加的进程利用 addjob 添加到工作组中。

要点分析：

1. 如果用户请求了一个内置的命令（quit，jobs，bg 或者 fg），那么立即执行它。 否则，派生一个子进程并在子进程中运行这个 job。 如果作业在前台运行，等待它终止，然后返回。

2. 每个子进程必须有自己独一无二的进程组 id，通过在 fork()之后的子进程中 Setpgid(0,0)实现，这样当向前台程序发送 ctrl+c 或 ctrl+z 命令时，才不会影响到后台程序。如果没有这一步，则所有的子进程与当前的 tsh shell 进程为同一个进程组，发送信号时，前后台的子进程均会收到。

3. 在 fork()新进程前后要阻塞 SIGCHLD 信号，防止出现竞争，这是经典的同步错误，如果不阻塞会出现子进程先结束从 jobs 中删除，然后再执行到主进程 addjob 的竞争问题。

3.1.2 int builtin_cmd(char **argv)函数（5分）

函数功能：检查第一个命令行参数是否是一个内置的 shell 命令
参　　数：argv
处理流程：如果第一个命令行参数是一个内置的 shell 命令，它就立即解释这个命令，并返回 1，否则返回 0。
要点分析：

1. 当命令行参数为 quit 时，杀死所有进程，并直接终止 shell
2. 当命令行参数为 jobs 时，调用 listjobs 函数，显示 job 列表，返回 1
3. 当命令行参数为 bg 或 fg 时，调用 do_bgfg 函数，执行内置的 bg 和 fg 命令，返回 1
4. 不是内置命令时返回 0

3.1.3 void do_bgfg(char **argv) 函数（5分）

函数功能：实现命令 bg 和 fg
参　　数：argv
处理流程：
1. 先做命令错误判断，fg 和 bg 后面是否有参数，参数是否符合%+数字或者数字，所表示的进程是否为正在运行的进程；
2. 做完这些判断之后，根据如果是%号，说明取的是工作组号，如果直接是数字说明取的是进程号，根据工作组号和进程号获取对应的 job 结构体；
3. 接下来如果是 bg，说明要恢复成后台进程，即改变 job 的 state；如果是 fg，说明要恢复成前台进程，即改变 job 的 state, 然后调用 waitfg；等前台进程运行结束。
要点分析：
做命令判断后，看 fg 和 bg 后面的参数，直接是数字则是工程组号，如果是%说明取的是进程号，然后如果是 bg 则恢复成后台进程，如果是 fg 恢复成前台进程。

3.1.4 void waitfg(pid_t pid) 函数（5分）

函数功能：等待进程号为 pid 的进程，直到其不是前台进程
参　　数：pid
处理流程：主进程 sleep 直到进程 pid 不再是前台进程
要点分析：前台进程与后台进程的唯一区别是 shell 会等待前台进程，因此前台进程只有一个。waitfg 实现了这一等待的功能。只需不断检查前台进程的状态即可，为了节省处理器资源的浪费，最佳选择是使用 sleep 函数，只要前台进程仍然是需要等待的进程，主进程就一直 sleep(1)。

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

函数功能：捕获 SIGCHILD 信号，用于回收僵尸子进程、改变被终止进程的状态标识、改变被挂起进程的状态标识。
参　　数：sig
处理流程：父进程用(pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0 作为 while 循环的测试条件。在每个子进程终止、停止时，对 waitpid 的调用会返回，返回值是该子进程的非零的 PID。通过检查 status 的状态，就能判断出，子进程是什么原因而导致 wait 返回：

若子进程是通过 exit 或 return 正常退出：则调用 deletejob 函数删除 jobs 数组的相应内容。
若子进程是收到信号而终止：打印提示信息，显示被终止的作业号、进程 pid、导致终止的信号，再调用 deletejob 函数删除 jobs 数组的相应内容。
若子进程是收到信号而挂起：打印提示信息，显示被挂起的作业号、进程 pid、导致挂起的信号。

要点分析：
这里调用 waitpid 函数的 option 参数为 WNOHANG | WUNTRACED，意为：立即返回。度过等待集合中的子进程都没有被停止或挂起，则返回值为 0；如果有一个停止或终止，则返回值为该子进程的 PID。这样就使得我们不仅能够处理被终止的进程，而且能够处理被挂起的进程，进而输出提示信息、改变 jobs 数组中的标识。

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
 *
 * < QiTian 1183200123>
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

/* Misc manifest constants */
```

```
#define MAXLINE 1024      /* max line size */
#define MAXARGS 128  /* max args on a command line */
#define MAXJOBS 16       /* max jobs at any point in time */
#define MAXJID 1 << 16 /* max job ID */

/* Job states */
#define UNDEF 0 /* undefined */
#define FG 1       /* running in foreground */
#define BG 2       /* running in background */
#define ST 3/* stopped */

/*
 * Jobs states: FG (foreground), BG (background), ST (stopped)
 * Job state transitions and enabling actions:
 *         FG -> ST    : ctrl-z
 *         ST -> FG    : fg command
 *         ST -> BG    : bg command
 *         BG -> FG    : fg command
 * At most 1 job can be in the FG state.
 */

/* Global variables */
extern char **environ;      /* defined in libc */
char prompt[] = "tsh> "; /* command line prompt (DO NOT CHANGE) */
int verbose = 0;            /* if true, print additional output */
int nextjid = 1;            /* next job ID to allocate */
char sbuf[MAXLINE];             /* for composing sprintf messages */

struct job_t
{                                   /* The job struct */
    pid_t pid;                      /* job PID */
    int jid;                    /* job ID [1, 2, ...] */
    int state;                  /* UNDEF, BG, FG, or ST */
    char cmdline[MAXLINE]; /* command line */
};
struct job_t jobs[MAXJOBS]; /* The job list */
/* End global variables */

/* Function prototypes */

/* Here are the functions that you will implement */
void eval(char *cmdline);
int builtin_cmd(char **argv);
void do_bgfg(char **argv);
void waitfg(pid_t pid);

void sigchld_handler(int sig);
```

```
void sigtstp_handler(int sig);
void sigint_handler(int sig);

/* Here are helper routines that we've provided for you */
int parseline(const char *cmdline, char **argv);
void sigquit_handler(int sig);

void clearjob(struct job_t *job);
void initjobs(struct job_t *jobs);
int maxjid(struct job_t *jobs);
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
int deletejob(struct job_t *jobs, pid_t pid);
pid_t fgpid(struct job_t *jobs);
struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
struct job_t *getjobjid(struct job_t *jobs, int jid);
int pid2jid(pid_t pid);
void listjobs(struct job_t *jobs);

void usage(void);
void unix_error(char *msg);
void app_error(char *msg);
typedef void handler_t(int);
handler_t *Signal(int signum, handler_t *handler);

/*
 * main - The shell's main routine
 */
int main(int argc, char **argv)
{
    char c;
    char cmdline[MAXLINE];
    int emit_prompt = 1; /* emit prompt (default) */

    /* Redirect stderr to stdout (so that driver will get all output
     * on the pipe connected to stdout) */
    dup2(1, 2);

    /* Parse the command line */
    while ((c = getopt(argc, argv, "hvp")) != EOF)
    {
        switch (c)
        {
        case 'h': /* print help message */
            usage();
            break;
        case 'v': /* emit additional diagnostic info */
            verbose = 1;
```

```
            break;
        case 'p':               /* don't print a prompt */
            emit_prompt = 0; /* handy for automatic testing */
            break;
        default:
            usage();
        }
}

/* Install the signal handlers */

/* These are the ones you will need to implement */
Signal(SIGINT, sigint_handler);      /* ctrl-c */
Signal(SIGTSTP, sigtstp_handler); /* ctrl-z */
Signal(SIGCHLD, sigchld_handler); /* Terminated or stopped child */

/* This one provides a clean way to kill the shell */
Signal(SIGQUIT, sigquit_handler);

/* Initialize the job list */
initjobs(jobs);

/* Execute the shell's read/eval loop */
while (1)
{

    /* Read command line */
    if (emit_prompt)
    {
        printf("%s", prompt);
        fflush(stdout);
    }
    if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
        app_error("fgets error");
    if (feof(stdin))
    { /* End of file (ctrl-d) */
        fflush(stdout);
        exit(0);
    }

    /* Evaluate the command line */
    eval(cmdline);
    fflush(stdout);
    fflush(stdout);
}

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

```
}

/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return.    Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
*/
void eval(char *cmdline)
{
    /* $begin handout */
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */
    sigset_t mask;       /* signal mask */

    /* Parse command line */
    bg = parseline(cmdline, argv);
    if (argv[0] == NULL)
        return; /* ignore empty lines */

    if (!builtin_cmd(argv))
    {

        /*
     * This is a little tricky. Block SIGCHLD, SIGINT, and SIGTSTP
     * signals until we can add the job to the job list. This
     * eliminates some nasty races between adding a job to the job
     * list and the arrival of SIGCHLD, SIGINT, and SIGTSTP signals.
     */

        if (sigemptyset(&mask) < 0)
            unix_error("sigemptyset error");
        if (sigaddset(&mask, SIGCHLD))
            unix_error("sigaddset error");
        if (sigaddset(&mask, SIGINT))
            unix_error("sigaddset error");
        if (sigaddset(&mask, SIGTSTP))
            unix_error("sigaddset error");
        if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
            unix_error("sigprocmask error");
```

```
        /* Create a child process */
        if ((pid = fork()) < 0)
            unix_error("fork error");

        /*
     * Child    process
     */

        if (pid == 0)
        {
            /* Child unblocks signals */
            sigprocmask(SIG_UNBLOCK, &mask, NULL);

            /* Each new job must get a new process group ID
            so that the kernel doesn't send ctrl-c and ctrl-z
            signals to all of the shell's jobs */
            if (setpgid(0, 0) < 0)
                unix_error("setpgid error");

            /* Now load and run the program in the new job */
            if (execve(argv[0], argv, environ) < 0)
            {
                printf("%s: Command not found\n", argv[0]);
                exit(0);
            }
        }

        /*
     * Parent process
     */

        /* Parent adds the job, and then unblocks signals so that
       the signals handlers can run again */
        addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
        sigprocmask(SIG_UNBLOCK, &mask, NULL);

        if (!bg)
            waitfg(pid);
        else
            printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
    /* $end handout */
    return;
}

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

```
 *
 * Characters enclosed in single quotes are treated as a single
 * argument.    Return true if the user has requested a BG job, false if
 * the user has requested a FG job.
 */
int parseline(const char *cmdline, char **argv)
{
    static char array[MAXLINE]; /* holds local copy of command line */
    char *buf = array;              /* ptr that traverses command line */
    char *delim;                /* points to first space delimiter */
    int argc;                   /* number of args */
    int bg;                         /* background job? */

    strcpy(buf, cmdline);
    buf[strlen(buf) - 1] = ' ';     /* replace trailing '\n' with space */
    while (*buf && (*buf == ' ')) /* ignore leading spaces */
        buf++;

    /* Build the argv list */
    argc = 0;
    if (*buf == '\'')
    {
        buf++;
        delim = strchr(buf, '\'');
    }
    else
    {
        delim = strchr(buf, ' ');
    }

    while (delim)
    {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf && (*buf == ' ')) /* ignore spaces */
            buf++;

        if (*buf == '\'')
        {
            buf++;
            delim = strchr(buf, '\'');
        }
        else
        {
            delim = strchr(buf, ' ');
        }
```

```
    }
    argv[argc] = NULL;

    if (argc == 0) /* ignore blank line */
        return 1;

    /* should the job run in the background? */
    if ((bg = (*argv[argc - 1] == '&')) != 0)
    {
        argv[--argc] = NULL;
    }
    return bg;
}

/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *       it immediately.
 */
int builtin_cmd(char **argv)
{
    sigset_t mask, prev_mask;
    if (!strcmp(argv[0], "quit")) //若第一个参数为 quit
    {

        sigfillset(&mask);
        sigprocmask(SIG_BLOCK, &mask, &prev_mask); //暂时阻隔所有信
号，以保护对全局变量的访问

        for (int i = 0; i < MAXJOBS; i++)
        {
            if (jobs[i].state != UNDEF) //杀死所有后台运行、挂起的进程组
            {
                kill(-(jobs[i].pid), SIGKILL);
                deletejob(jobs, jobs[i].pid);
            }
        }
        sigprocmask(SIG_SETMASK, &prev_mask, NULL); //解除阻隔

        exit(0);
    }
    else if (!strcmp(argv[0], "fg") || !strcmp(argv[0], "bg")) //若第一个参数为
fg 或 bg
    {
        do_bgfg(argv);
        return 1; //返回：“是” 内部参数
    }
```

```
    else if (!strcmp(argv[0], "jobs"))
    {
        listjobs(jobs);
        return 1; //返回：“是” 内部参数
    }
    else
        return 0; //返回：“不是” 内部参数
}

/*
 * do_bgfg - Execute the builtin bg and fg commands
 */
void do_bgfg(char **argv)
{
    /* $begin handout */
    struct job_t *jobp = NULL;

    /* Ignore command if no argument */
    if (argv[1] == NULL)
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    /* Parse the required PID or %JID arg */
    if (isdigit(argv[1][0]))
    {
        pid_t pid = atoi(argv[1]);
        if (!(jobp = getjobpid(jobs, pid)))
        {
            printf("(%d): No such process\n", pid);
            return;
        }
    }
    else if (argv[1][0] == '%')
    {
        int jid = atoi(&argv[1][1]);
        if (!(jobp = getjobjid(jobs, jid)))
        {
            printf("%s: No such job\n", argv[1]);
            return;
        }
    }
    else
    {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
```

```
        return;
    }

    /* bg command */
    if (!strcmp(argv[0], "bg"))
    {
        if (kill(-(jobp->pid), SIGCONT) < 0)
            unix_error("kill (bg) error");
        jobp->state = BG;
        printf("[%d] (%d) %s", jobp->jid, jobp->pid, jobp->cmdline);
    }

    /* fg command */
    else if (!strcmp(argv[0], "fg"))
    {
        if (kill(-(jobp->pid), SIGCONT) < 0)
            unix_error("kill (fg) error");
        jobp->state = FG;
        waitfg(jobp->pid);
    }
    else
    {
        printf("do_bgfg: Internal error\n");
        exit(0);
    }
    /* $end handout */
    return;
}

/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
    struct job_t *jobp = NULL;

    jobp = getjobpid(jobs, pid); //获取指向等待进程的指针
    while (jobp->state == FG)      //当该进程不再是前台进程时，跳出循环
    {
        sleep(1);
    }

    return;
}

/*****************
```

```
 * Signal handlers
 *****************/

/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 *         a child job terminates (becomes a zombie), or stops because it
 *         received a SIGSTOP or SIGTSTP signal. The handler reaps all
 *         available zombie children, but doesn't wait for any other
 *         currently running children to terminate.
 */

/*当有子进程挂起或终止时，
 *进入此信号处理函数。
 */
void sigchld_handler(int sig)
{
    int olderrno = errno; //保存 errno
    int status;
    int pid;

    /*回收当前每一个僵死进程 或 处理每个挂起的进程*/
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        if (WIFEXITED(status)) //若子进程是通过 exit 或 return 正常退出
            deletejob(jobs, pid);
        else if (WIFSIGNALED(status)) //若子进程是收到信号而终止
        {
            if (WTERMSIG(status) != SIGKILL)
            {
                printf("Job [%d] (%d) terminated by signal %d\n",
                        pid2jid(pid), pid, WTERMSIG(status)); //打印提示
            }
            deletejob(jobs, pid); //将此进程从进程列表中删除
        }
        else if (WIFSTOPPED(status)) //若子进程是收到信号而挂起
        {
            printf("Job [%d] (%d) stopped by signal %d\n",
                    pid2jid(pid), pid, WSTOPSIG(status)); //打印提示
            getjobpid(jobs, pid)->state = ST;                //将此进程标记为挂
起的进程
        }
    }

    errno = olderrno; //恢复 errno
    return;
```

```
}

/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenver the
 *      user types ctrl-c at the keyboard.  Catch it and send it along
 *      to the foreground job.
 */
void sigint_handler(int sig)
{
    sigset_t mask, prev_mask;
    int olderrno = errno; //保存 errno

    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask); //暂时阻隔所有信号，
以保护对全局变量的访问

    for (int i = 0; i < MAXJOBS; i++)
    {
        if (jobs[i].pid != 0)
        {
            if (jobs[i].state == FG) //找到前台进程，并发送信号将其杀死
            {
                if (kill(-(jobs[i].pid), SIGINT) < 0)
                    unix_error("kill error\n");
            }
        }
    }

    sigprocmask(SIG_SETMASK, &prev_mask, NULL); //解除阻隔
    errno = olderrno;                           //恢复 errno
    return;
}

/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 *      the user types ctrl-z at the keyboard. Catch it and suspend the
 *      foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    sigset_t mask, prev_mask;
    int olderrno = errno; //保存 errno

    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask); //暂时阻隔所有信号，
以保护对全局变量的访问
```

```
    for (int i = 0; i < MAXJOBS; i++)
    {
        if (jobs[i].pid != 0)
        {
            if (jobs[i].state == FG)
            {
                if (kill(-(jobs[i].pid), SIGTSTP) < 0) //找到前台进程，并发送信号将其停止
                    unix_error("kill error\n");
            }
        }
    }

    sigprocmask(SIG_SETMASK, &prev_mask, NULL); //解除阻隔
    errno = olderrno;                           //恢复 errno
    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）

<table>
<tr><th>tsh 测试结果</th><th>tshref 测试结果</th></tr>
<tr><td>qt1183200123@ubuntu:~/hitics/shlab-handout-hit$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
trace01.txt - Properly terminate on EOF.
#</td><td>qt1183200123@ubuntu:~/hitics/shlab-handout-hit$ make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
trace01.txt - Properly terminate on EOF.
#</td></tr>
<tr><td>测试结论</td><td colspan="2">相同</td></tr>
</table>

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
qt1183200123@ubuntu:~/hitics/shlab-handout-hit$ make test02 ./sdriver.pl -t trace02.txt -s ./tsh -a "-p" # # trace02.txt - Process builtin quit command. #	qt1183200123@ubuntu:~/hitics/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

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
qt1183200123@ubuntu:~/hitics/shlab-handout-hit$ make test03 ./sdriver.pl -t trace03.txt -s ./tsh -a "-p" # # trace03.txt - Run a foreground job. # tsh> quit	qt1183200123@ubuntu:~/hitics/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

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
qt1183200123@ubuntu:~/hitics/shlab-handout-hit$ make test04 ./sdriver.pl -t trace04.txt -s ./tsh -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 &	qt1183200123@ubuntu:~/hitics/shlab-handout-hit$ make rtest04 ./sdriver.pl -t trace04.txt -s ./tshref -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 &
测试结论	相同

4.3.5 测试用例 trace05.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4779) ./myspin 2 & tsh> ./myspin 3 & [2] (4781) ./myspin 3 & tsh> jobs [1] (4779) Running ./myspin 2 & [2] (4781) Running ./myspin 3 &	qt1183200123@ubuntu:~/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] (4788) ./myspin 2 & tsh> ./myspin 3 & [2] (4790) ./myspin 3 & tsh> jobs [1] (4788) Running ./myspin 2 & [2] (4790) Running ./myspin 3 &
测试结论	相同

4.3.6 测试用例 trace06.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4798) terminated by signal 2	qt1183200123@ubuntu:~/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] (4804) terminated by signal 2
测试结论	相同

4.3.7 测试用例 trace07.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4811) ./myspin 4 & tsh> ./myspin 5 Job [2] (4813) terminated by signal 2 tsh> jobs [1] (4811) Running ./myspin 4 &	qt1183200123@ubuntu:~/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] (4830) ./myspin 4 & tsh> ./myspin 5 Job [2] (4832) terminated by signal 2 tsh> jobs [1] (4830) Running ./myspin 4 &
测试结论	相同

4.3.8 测试用例 trace08.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4843) ./myspin 4 & tsh> ./myspin 5 Job [2] (4845) stopped by signal 20 tsh> jobs [1] (4843) Running ./myspin 4 & [2] (4845) Stopped ./myspin 5	qt1183200123@ubuntu:~/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] (4852) ./myspin 4 & tsh> ./myspin 5 Job [2] (4854) stopped by signal 20 tsh> jobs [1] (4852) Running ./myspin 4 & [2] (4854) Stopped ./myspin 5
测试结论	相同

4.3.9 测试用例 trace09.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4861) ./myspin 4 & tsh> ./myspin 5 Job [2] (4863) stopped by signal 20 tsh> jobs [1] (4861) Running ./myspin 4 & [2] (4863) Stopped ./myspin 5 tsh> bg %2 [2] (4863) ./myspin 5 tsh> jobs [1] (4861) Running ./myspin 4 & [2] (4863) Running ./myspin 5	qt1183200123@ubuntu:~/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] (4872) ./myspin 4 & tsh> ./myspin 5 Job [2] (4874) stopped by signal 20 tsh> jobs [1] (4872) Running ./myspin 4 & [2] (4874) Stopped ./myspin 5 tsh> bg %2 [2] (4874) ./myspin 5 tsh> jobs [1] (4872) Running ./myspin 4 & [2] (4874) Running ./myspin 5
测试结论	相同

4.3.10 测试用例 trace10.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4885) ./myspin 4 & tsh> fg %1 Job [1] (4885) stopped by signal 20 tsh> jobs [1] (4885) Stopped ./myspin 4 & tsh> fg %1 tsh> jobs	qt1183200123@ubuntu:~/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] (4895) ./myspin 4 & tsh> fg %1 Job [1] (4895) stopped by signal 20 tsh> jobs [1] (4895) Stopped ./myspin 4 & tsh> fg %1 tsh> jobs
测试结论	相同

4.3.11 测试用例 trace11.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4906) terminated by signal 2 tsh> /bin/ps a PID TTY STAT TIME COMMAND 2656 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu 2658 tty2 Sl+ 0:32 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 2667 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu （截图有省略）	qt1183200123@ubuntu:~/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] (4917) terminated by signal 2 tsh> /bin/ps a PID TTY STAT TIME COMMAND 2656 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu 2658 tty2 Rl+ 0:33 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 2667 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu （截图有省略）
测试结论	相同

4.3.12 测试用例 trace12.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4933) stopped by signal 20 tsh> jobs [1] (4933) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND 2656 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu 2658 tty2 Sl+ 0:37 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 2667 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu （截图有省略）	qt1183200123@ubuntu:~/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] (4943) stopped by signal 20 tsh> jobs [1] (4943) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND 2656 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu 2658 tty2 Sl+ 0:38 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 2667 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu （截图有省略）
测试结论	相同

4.3.13 测试用例 trace13.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
qt1183200123@ubuntu:~/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] (4954) stopped by signal 20 tsh> jobs [1] (4954) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND 2656 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu 2658 tty2 Sl+ 0:38 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 2667 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu （截图有省略）	qt1183200123@ubuntu:~/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] (4968) stopped by signal 20 tsh> jobs [1] (4968) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND 2656 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu 2658 tty2 Rl+ 0:40 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 2667 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu （截图有省略）
测试结论	相同

4.3.14 测试用例 trace14.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
qt1183200123@ubuntu:~/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] (4984) ./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] (4984) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (4984) ./myspin 4 &
tsh> jobs
[1] (4984) Running ./myspin 4 &
```

```
qt1183200123@ubuntu:~/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] (5004) ./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] (5004) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (5004) ./myspin 4 &
tsh> jobs
[1] (5004) Running ./myspin 4 &
```

测试结论	相同

4.3.15 测试用例 trace15.txt 的输出截图（1 分）

tsh 测试结果

```
qt1183200123@ubuntu:~/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] (5094) terminated by signal 2
tsh> ./myspin 3 &
[1] (5096) ./myspin 3 &
tsh> ./myspin 4 &
[2] (5098) ./myspin 4 &
tsh> jobs
[1] (5096) Running ./myspin 3 &
[2] (5098) Running ./myspin 4 &
tsh> fg %1
Job [1] (5096) stopped by signal 20
tsh> jobs
[1] (5096) Stopped ./myspin 3 &
[2] (5098) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (5096) ./myspin 3 &
tsh> jobs
[1] (5096) Running ./myspin 3 &
[2] (5098) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

tshref 测试结果

```
qt1183200123@ubuntu:~/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] (5114) terminated by signal 2
tsh> ./myspin 3 &
[1] (5116) ./myspin 3 &
tsh> ./myspin 4 &
[2] (5118) ./myspin 4 &
tsh> jobs
[1] (5116) Running ./myspin 3 &
[2] (5118) Running ./myspin 4 &
tsh> fg %1
Job [1] (5116) stopped by signal 20
tsh> jobs
[1] (5116) Stopped ./myspin 3 &
[2] (5118) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (5116) ./myspin 3 &
tsh> jobs
[1] (5116) Running ./myspin 3 &
[2] (5118) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

测试结论	相同

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

4.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

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.