哈爾濱工業大學

实验报告

实 验（七）

题　　目 TinyShell

微壳

专　　业 计算机类

学　　号 1190200910

班　　级 1903012

学　　生 严幸

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2021-06-02

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

- VScode

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

Kill 命令

- kill －l：列出信号
- kill －SIGKILL 17130： 杀死 pid 为 17130 的进程
- kill -9 17130 ：杀死 pid 为 17130 的进程，或者：
- kill -9 -17130：杀死进程组 17130 中的每个进程
- killall -9 pname： 杀死名字为 pname 的进程

进程状态

- D 不可中断睡眠 (通常是在 IO 操作) 收到信号不唤醒和不可运行, 进程必须等待直到有中断发生
- R 正在运行或可运行（在运行队列排队中）
- S 可中断睡眠 (休眠中, 受阻, 在等待某个条件的形成或接受到信号)
- T 已停止的 进程收到 SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU 信号后停止运行
- W 正在换页(2.6.内核之前有效)
- X 死进程 (未开启)
- Z 僵尸进程 a defunct (”zombie”) process
- < 高优先级(not nice to other users)
- N 低优先级(nice to other users)
- L 页面锁定在内存（实时和定制的 IO）
- s 一个信息头
- l 多线程（使用 CLONE_THREAD，像 NPTL 的 pthreads 的那样）
- + 在前台进程组

ps t /ps aux /ps

- t <终端机编号 n> 列终端 n 的程序的状况。
- a 显示现行终端机下的所有程序，包括其他用户的程序。
- u 以用户为主的格式来显示程序状况。
- x 显示所有程序，不以终端来区分。

作业 ： jobs、 fg %n 、 bg%n

- jobs 显示当前暂停的进程
- bg %n 使第 n 个任务在后台运行(%前有空格)
- fg %n 使第 n 个任务在前台运行
- bg, fg 不带%n 表示对最后一个进程操作

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

第 2 章 实验预习

总分 20 分

2.1 进程的概念、创建和回收方法（5 分）

1) 进程的概念[7]

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

2) 进程的创建

Linux 系统允许任何一个用户进程创建一个子进程，创建成功后，子进程存在于系统之中，并且独立于父进程。该子进程可以接受系统调度，可以得到分配的系统资源。系统也可以检测到子进程的存在，并且赋予它与父进程同样的权利。

Linux 系统下使用 fork()函数创建一个子进程，其函数原型如下：

```
#include <unistd.h>
pid_t fork(void);
```

创建新进程的进程，即调用 fork()函数的进程就是父进程，而新创建的进程就是子进程。

3) 进程的回收[8]

进程运行结束后由其父进程回收。父进程可以调用 wait 函数或 waitpid 函数对子进程进行回收。其函数原型如下：

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

wait 函数主要完成三个任务：

- 阻塞父进程的运行，直到子进程终止再继续，停等同步。
- 获取子进程的 PID 和终止状态，令父进程得知谁因何而死。
- 为子进程收尸，防止大量僵尸进程耗费系统资源。

- 对于孤儿进程：调用 wait 会使父进程进入等待状态，直到所有子进程都终止。
- 对于僵尸进程：调用 wait 会立即收回子进程，获取子进程的终止状态。

Waitpid 函数的参数如下设置：

pid < -1　　// 等待并回收特定进程组（由 pid 标识）的任意子进程

pid == -1　// 等待并回收任意子进程，相当于 wait 函数　（*）

pid == 0　　// 等待并回收与调用进程同进程组的任意子进程

pid > 0　　// 等待并回收特定子进程（由 pid 标识）　（*）

status 用于输出子进程的终止状态

第三个参数可以取以下值：

0　　// 阻塞模式，若所等子进程仍在运行，则阻塞，直至其终止。　（*）

WNOHANG　// 非阻塞模式，若所等子进程仍在运行，则返回 0　（*）

WCONTINUED　// 若实现支持作业控制，那么由 pid 指定的任一子进程在停止后已经继续，但其装填尚未报告，则返回其状态。

WUNTRACED　// 若某实现支持作业控制，而由 pid 指定的任一子进程已处于停止状态，并且其状态自停止以来还未报告过，则返回其状态。WIFSTOPPED 宏确定返回值是否对应于一个停止的子进程。

2.2 信号的机制、种类（5 分）

1) 信号机制

信号是 4 种异常控制流(ECF)中的一种，信号相当于是软中断，由内核向进程发送，许多重要的程序都需要处理信号。信号为 Linux 提供了一种处理异步事件的方法。比如，终端用户输入了 ctrl+c 来中断程序，会通过信号机制停止一个程序。

2) 信号的种类

Linux 中有多种信号，可以在 bash 中用 kill -l 命令列出。

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

每个信号都有一个名字和编号，这些名字都以“SIG”开头，例如“SIGIO ”、“SIGCHLD”等等。信号定义在 signal.h 头文件中，信号名都定义为正整数。每种信号都有自己的特定作用。

2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分）

1) 信号的发送

Linux 中可以采用三种方式给进程发信号：

a. 在命令行中用 kill 命令给进程发信号

kill –[signum] [pid]

向 pid 进程发送 signum 信号

b. 利用键盘给进程发信号

Ctrl+C 向前台进程组的每一个进程发送 SIGINT 信号

Ctrl+Z 向前台进程组的每一个进程发送 SIGSTOP 信号

c. 利用 C 语言库函数 kill()显式地要求内核给一个进程发信号

```
#include <signal.h>
#include <sys/types.h>
int kill(pid_t pid, int sig);
```

sig 是要发送的信号代码

pid>0 时，将信号发送给进程号为 pid 的进程；

pid=0 时，将信号发送给与目前进程相同进程组的所有进程；

pid=-1 时，将信号发给调用进程的所有子进程

pid<-1 时，向进程组 ID 为 pid 绝对值的进程组中的所有进程发送信号

2) 信号的阻塞

信号的阻塞分隐式阻塞和显式阻塞。

隐式阻塞：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：处理一个 SIGINT 信号时，如果此时又收到了一个 SIGINT 信号，则这个新的 SIGINT 信号被阻塞，只有内核下一次调度该进程时，这个新的 SIGINT 信号才能被接收。

显式阻塞：调用 C 语言库函数 sigprocmask()来显式地对某一(或某些)信号进行阻塞。这个函数的原型如下：

```
#include<signal.h>
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oldset);
```

此外还有一些辅助函数如 sigemptyset()，sigaddset()等等，对 sigprocmask 起辅助作用。

3) 信号处理程序的设置

使用 C 语言库函数 signal()设置某一信号的处理程序，即修改程序接收某一信号时的默认行为。

其函数原型如下：

```
void (*signal(int sig, void (*func)(int)))(int)
```

sig 是要处理信号的代码

func 是函数指针，代表着用户处理函数的地址。

func=SIG_DFL，修改为默认的信号处理程序。

func=SIG_IGN，忽视该信号。

2.4 什么是 shell，功能和处理流程（5 分）

1) Shell

在计算机科学中，Shell 俗称壳（用来区别于核），是指“为使用者提供操作界面”的软件（command interpreter，命令解析器）。

2) 功能

Shell 是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。不仅如此，Shell 有自己的编程语言用于对命令的编辑，它允许用户编写由 shell 命令组成的程序。Shell 编程语言具有普通编程语言的很多特点，比如它也有循环结构和分支控制结构等，用这种编程语言编写的 Shell 程序与其他应用程序具有同样的效果

3) 处理流程

Shell 读取一行命令，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。然后 shell 在搜索路径里寻找这些应用程序。如果键入的命令不是一个内部命令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：执行用户输入的一行命令。
参　　数：char *cmdline：字符指针，指向了用户输入的一行命令的字符串。
处理流程：
函数程序先对 cmdline 中的每一个参数进行解析，按空格分开，并把解析的结果放到 argv 数组当中，同时返回它是否是一个后台任务。然后对 argv 中包含的命令参数进行解析，如果它是 tsh 内置命令*(quit, jobs, bg or fg)*则直接执行对应的命令，否则的话要 fork 一个子进程，并在子进程中 execve 整个 argv 中的命令。同时父进程要对新增的作业进行管理，如果是前台进程还需要等待前台进程结束。

要点分析：
函数实现上，对 cmdline 的解析通过调用 parseline 函数实现，此函数同时会返回它是否是后台任务的标记。另外，此函数需要忽略空指令，即只包含空格或&的无意义命令。然后，在 fork 子进程之前，为了避免竞争(并发错误)，需要在父进程中事先阻塞 SIGCHLD、SIGINT、SIGTSTP 信号。Fork 之后，对于子进程，在解除信号阻塞后，还需要调用 setpgid 修改进程组号，防止在 bash 中将 tsh 和子进程一起杀死，然后直接 execve 即可。对于父进程，需要将 fork 出的子进程通过 addjob 函数添加到作业管理集合当中，这个函数同样存在潜在的“竞争”并发错误，在这个函数执行时仍要保持信号的阻塞，在执行玩 addjob 函数后，解除父进程中的信号阻塞，然后判断这是否是一个前台进程，如果是前台的，则调用 waitfg 等待，否则打印后台进程信息。

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

函数功能：判断 argc 中的命令是否是内置命令，如果是，则执行它。
参　　数：char **argv，一个字符指针的数组，包含了 cmdline 解析后的所有命令参数列表。
处理流程：先判断是否是四个内置命令之一，如果是，则执行它并且返回 1；如果不是则返回 0。
要点分析：
四个内置命令包括 quit, jobs, bg or fg，可以直接拿 argv[0]和它们一一匹配即可。只要其中一个匹配上，则执行对应的命令即可。quit 命令只需要调用 exit()函数，退

出 tsh 即可；jobs 命令通过调用 listjobs 函数来输出全部的作业列表；bg 和 fg 命令通过调用 do_bgfg()函数来执行 bg 和 fg 命令；四个命令除了 quit，都要返回值 1。如果四个都没有匹配上则返回 0。另外，需要在函数的一开始判断一下 argv[0]是不是&，如果是，则直接返回 1。

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

函数功能：执行 bg 或 fg 命令。
参　　数：char **argv，一个字符指针的数组，包含了 cmdline 解析后的所有命令参数列表。
处理流程：由于 bg 和 fg 命令都需要输入额外的参数来表征对应进程的 PID，因此首先要检查 argv[1]是否为空，如果为空则说明命令有误。然后检查 bg/fg 后面跟的参数是否合法，看他是否是数字或%后接数字，并通过 getjobpid 函数或 getjobjid 函数拿到需要操作的 job 在 jobs 数组中的指针。然后就根据 bg 或 fg 命令分别向对应的进程通过 kill 函数发送 SIGCONT 信号，并修改 job 状态。如果是 fg 命令，就还需要在前台通过 waitfg()函数等待进程。

要点分析：
先判断 argv[1] 是否等于 NULL，检查命令是否带有额外参数，如果没有则命令非法。然后通过 isdigit(argv[1][0])来判断 bg/fg 的参数是否以数字开头，如果是以数字开头，则用 atoi(argv[1])获取目标进程的 PID，并用 getjobpid(jobs, pid))拿到对应 job 的指针。如果 argv[1][0] == '%'，则证明它是通过 jobID 来访问进程的，用 atoi(&argv[1][1])拿到 jobID 的数值，然后用 jobp = getjobjid(jobs, jid))拿到 Job 的指针。然后分别用 strcmp(argv[0], "bg")和 strcmp(argv[0], "fg")来看执行的是 bg 命令还是 fg 命令。两个命令都需要用(kill(-(jobp->pid), SIGCONT)来向目标进程发送 SIGCONT 信号，fg 命令还要用 waitfg(jobp->pid);在前台显式地等待进程运行。

3.1.4 void waitfg(pid_t pid) 函数（5 分）

函数功能：显式地阻塞程序，等待前台进程运行。
参　　数：前台进程的 PID
处理流程：
先找到前台进程对应的 job 在 jobs 数组中的下标，不断地检测这个进程对应的 job 的状态是不是 FG(前台运行)，如果不是在前台运行则退出程序。
要点分析：
函数开头检查 pid 是否合法，即验证 pid 是否为正数，如果 pid<=0 则直接返回。然后遍历一下作业数组 jobs，找到 job 的 pid 为函数参数 pid 的那一个 job 的下标，然后用 while (jobs[i].state == FG)循环不断地验证这一个 job 是否是在前台运行，每次验证成功就在循环体内 sleep(1)。即采用“busy loop”。一旦循环验证失败，则退出循环，函数返回。

3.1.5 void sigchld_handler(int sig) 函数（10 分）

函数功能：处理 SIGCHLD 信号
参　　数：int sig 信号代码
处理流程：
通过一个 while ((child_pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)循环，不断的收集父进程的子进程终止信息，每收到一个子进程的 child_pid，就对这个进程对应的 job 进行相应操作。如果是子进程被挂起(STOP)导致的 SIGCHLD 信号，就把这个 job 的状态修改为 ST；如果是子进程结束，分为正常 EXIT 结束和被其他信号终止的异常结束，异常结束时还要打印额外的信息。最后要用 deletejob 释放对应的 job。

要点分析：
对 job 进行操作时，如果同时出现了其他信号，就会中断对 job 的操作，因此对 job 进行操作前必须先把所有信号阻塞，用 sigfillset(&mask)方法设置 mask 的所有位，然后用 sigprocmask(SIG_BLOCK, &mask, &prev_mask)阻塞掉所有信号。在对 job 操作后通过 sigprocmask(SIG_SETMASK, &prev_mask, NULL)来恢复之前的阻塞。当然，这些系统调用的返回值都应该为 0，如果返回-1 则代表出错需要处理，需要输出错误信息。然后为了防止在系统调用中修改全局错误码 error，在函数的一开始需要将 error 保存在 olderrno 当中，函数结束前恢复 error 码。函数流程上总体就是用一个 while 循环，回收掉所有僵死子进程。
child_pid = waitpid(-1, &status, WNOHANG | WUNTRACED)会把僵死子进程的 pid 放到 child_pid 中，把状态码放到 status 中。
每发现一个僵死子进程，通过 for 循环找到其对应的 job 下标，通过这个下标修改对应的 job 条目。通过 WIFSTOPPED(status)来判断是不是子进程被挂起(STOP)而导致了 SIGCHLD 信号，如果子进程是被挂起，则只需要修改对应 job 的 state 为 ST 即可，然后输出提示信息。如果子进程不是被挂起，则子进程是被终止了才导致了 SIGCHLD 信号。对于终止的情况，需要用 deletejob(jobs, child_pid)来释放子进程对应的 job。其次还需要通过 WIFSIGNALED(status)来判断子进程是自己 exit()退出了还是被一个信号给 kill 掉异常结束了。如果是被信号 kill 异常结束，需要打印提示信息。其中涉及到的系统调用，返回值都需要进行处理，以避免潜在的异常。另外，全局错误码 error 还需要做好保存和恢复。

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

```c
/*
 * tsh - A tiny shell program with job control
 *
 * 严幸 1190200910
 */
#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);
```

```c
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;
}
```

```c
/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *    it immediately.
 */
int builtin_cmd(char **argv)
{
    if (strcmp(argv[0], "&") == 0) //忽略单独的&
        return 1;
    if (strcmp(argv[0], "quit") == 0) //退出
    {
        exit(0);
    }
    else if (strcmp(argv[0], "fg") == 0 || strcmp(argv[0], "bg") == 0) //
fg 命令和bg 命令
    {
        do_bgfg(argv); //这里调用已实现的函数即可，do_bgfg 函数内部会对bg 和fg
指令进行区分
        return 1;
    }
    else if (strcmp(argv[0], "jobs") == 0)
    {
        listjobs(jobs);
        return 1;
    }
    return 0; /* not a builtin command */
}

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

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

```
        return;
    }

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

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

    /* fg command */
    else if (!strcmp(argv[0], "fg"))
```

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

/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
    if (pid <= 0)
        return;
    int i = 0;
    for (; i < MAXJOBS; i++)
    {
        if (jobs[i].pid == pid)
            break;
    }
    while (jobs[i].state == FG)
        sleep(1);
    if (verbose)
    {
        printf("waitfg: Process (%d) no longer the fg process\n", pid);
    }
    return;
}

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

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

/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 *     a child job terminates (becomes a zombie), or stops because it
 *     received a SIGSTOP or SIGTSTP signal. The handler reaps all
 *     available zombie children, but doesn't wait for any other
 *     currently running children to terminate.
 */
void sigchld_handler(int sig)
{
    if (verbose)
        printf("sigchld_handler: entering");
    int olderrno = errno;
    sigset_t mask, prev_mask;
    pid_t child_pid;
    int status;
    //先设置mask的所有位，马上阻塞掉所有信号
    if (sigfillset(&mask) < 0)
    {
        unix_error("Can't fill in mask!!\n"); //处理一下系统调用的返回值，防止出错
    }

    while ((child_pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        //根据pid找到作业的下标
        int i = -1;
        for (i = 0; i < MAXJOBS; i++)
        {
            if (jobs[i].pid == child_pid)
                break;
        }
        if (i == -1)
            unix_error("Can't find the job!");
        if (WIFSTOPPED(status)) //如果是子进程被挂起导致的SIGCHLD信号
        {
            jobs[i].state = ST;
```

```
            printf("Job [%d] (%d) stopped by signal %d\n", jobs[i].jid, j
obs[i].pid, WSTOPSIG(status));
        }
        else
        {
            if (WIFSIGNALED(status)) //如果子进程异常结束，即由其他进程给他发
信号导致子进程结束，引起的 SIGCHLD 信号
            {
                printf("Job [%d] (%d) terminated by signal %d\n", jobs[i]
.jid, jobs[i].pid, WTERMSIG(status));
            }
            fflush(stdout);
            //deletejob 需要阻塞所有信号，防止竞争
            if (sigprocmask(SIG_BLOCK, &mask, &prev_mask) < 0) //阻塞所有
信号，先前的阻塞情况放在 prev_mask 当中，恢复时使用
            {
                unix_error("Can't block signal!!\n"); //处理一下系统调用的
返回值，防止出错
            }
            if (deletejob(jobs, child_pid) == 0) //终止的进程直接回收，如果
返回 0 则代表出错
            {
                unix_error("Delete job error!\n"); //处理一下系统调用的返回
值，防止出错
            }

            if (sigprocmask(SIG_SETMASK, &prev_mask, NULL) < 0) //deletej
ob 之后需要恢复信号阻塞
            {
                unix_error("Can't restore block!!\n"); //处理一下系统调用的
返回值，防止出错
            }
        }
    }

    errno = olderrno;
    return;
}
```

```c
/*
 * 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; //先保存一遍错误代码，防止在下面的系统调用中修改全局的错误代码error

    //先设置mask的所有位，马上阻塞掉所有信号
    if (sigfillset(&mask) < 0)
    {
        unix_error("Can't fill in mask!!\n"); //处理一下系统调用的返回值，防止出错
    }

    //阻塞掉所有信号，先前的阻塞情况保存到prev_mask，用来恢复阻塞时使用
    if (sigprocmask(SIG_BLOCK, &mask, &prev_mask))
    {
        unix_error("Can't block signal!!\n"); //处理一下系统调用的返回值，防止无法阻塞
    }

    //获取前台作业下标
    for (int i = 0; i < MAXJOBS; i++)
    {
        if (jobs[i].state == FG)
        {
            if (jobs[i].pid > 0)                    //检查一下pid是否合法
                if (kill(-jobs[i].pid, SIGINT) < 0) //向一整个进程组发送信号，杀死整个进程组
                {
                    unix_error("Can't send signal!!\n");
                }
            break;
        }
    }
```

```
    errno = olderrno;                                   //恢复原先的全局错误代码
    if (sigprocmask(SIG_SETMASK, &prev_mask, NULL) < 0) //恢复原先的阻塞情况
    {
        unix_error("Can't restore the signal block!!\n"); //处理系统调用返回值防止出错
    }
    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; //先保存一遍错误代码，防止在下面的系统调用中修改全局的错误代码error

    //先设置mask的所有位，马上阻塞掉所有信号
    if (sigfillset(&mask) < 0)
    {
        unix_error("Can't fill in mask!!\n"); //处理一下系统调用的返回值，防止出错
    }

    //阻塞掉所有信号，先前的阻塞情况保存到prev_mask，用来恢复阻塞时使用
    if (sigprocmask(SIG_BLOCK, &mask, &prev_mask))
    {
        unix_error("Can't block signal!!\n"); //处理一下系统调用的返回值，防止无法阻塞
    }

    //获取前台作业下标
    for (int i = 0; i < MAXJOBS; i++)
    {
```

```
        if (jobs[i].state == FG)
        {
            if (jobs[i].pid > 0)                    //检查一下pid是否合法
                if (kill(-jobs[i].pid, SIGTSTP) < 0) //向一整个进程组发送
SIGTSTP信号
                {
                    unix_error("Can't send signal!!\n"); //处理系统调用返回
值防止出错
                }
            break;
        }
    }
    errno = olderrno;                               //恢复原先的全局错
误代码
    if (sigprocmask(SIG_SETMASK, &prev_mask, NULL) < 0) //恢复原先的阻塞情
况
    {
        unix_error("Can't restore the signal block!!\n"); //处理系统调用返
回值防止出错
    }
    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';
}
```

```c
/* 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
 ******************************/
```

```c
/**********************
 * Other helper routines
 **********************/

/*
 * usage - print a help message
 */
void usage(void)
{
    printf("Usage: shell [-hvp]\n");
    printf("   -h   print this message\n");
    printf("   -v   print additional diagnostic information\n");
    printf("   -p   do not emit a command prompt\n");
    exit(1);
}

/*
 * unix_error - unix-style error routine
 */
void unix_error(char *msg)
{
    fprintf(stdout, "%s: %s\n", msg, strerror(errno));
    exit(1);
}

/*
 * app_error - application-style error routine
 */
void app_error(char *msg)
{
    fprintf(stdout, "%s\n", msg);
    exit(1);
}

/*
 * Signal - wrapper for the sigaction function
 */
handler_t *Signal(int signum, handler_t *handler)
{
```

```
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}

/*
 * sigquit_handler - The driver program can gracefully terminate the
 *    child shell by sending it a SIGQUIT signal.
 */
void sigquit_handler(int sig)
{
    printf("Terminating after receipt of SIGQUIT signal\n");
    exit(1);
}
```

第 4 章 TinyShell 测试

总分 15 分

4.1 测试方法

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

4.2 测试结果评价

tsh 与 tshref 的输出在以下两个方面可以不同：

（1）pid

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

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

4.3 自测试结果

填写以下各个测试用例的测试结果，每个测试用例 1 分。

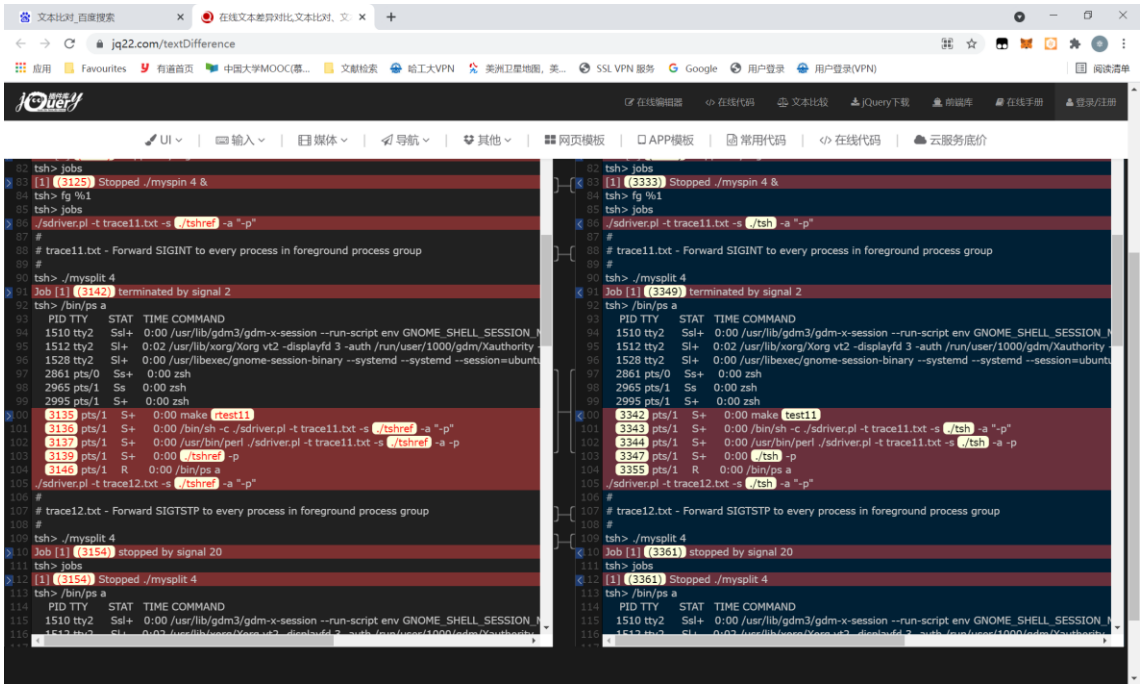
对于这里的测试，我编写了如下的自动化测试脚本。

```
#!/bin/bash

gcc tsh.c -o tsh

rm tshref_ret
rm tsh_ret

make rtest01 >> tshref_ret
make rtest02 >> tshref_ret
make rtest03 >> tshref_ret
make rtest04 >> tshref_ret
make rtest05 >> tshref_ret
make rtest06 >> tshref_ret
make rtest07 >> tshref_ret
make rtest08 >> tshref_ret
make rtest09 >> tshref_ret
make rtest10 >> tshref_ret
make rtest11 >> tshref_ret
make rtest12 >> tshref_ret
make rtest13 >> tshref_ret
make rtest14 >> tshref_ret
make rtest15 >> tshref_ret
```

```
make test01 >> tsh_ret
make test02 >> tsh_ret
make test03 >> tsh_ret
make test04 >> tsh_ret
make test05 >> tsh_ret
make test06 >> tsh_ret
make test07 >> tsh_ret
make test08 >> tsh_ret
make test09 >> tsh_ret
make test10 >> tsh_ret
make test11 >> tsh_ret
make test12 >> tsh_ret
make test13 >> tsh_ret
make test14 >> tsh_ret
make test15 >> tsh_ret

"~/Desktop/ICS/lab7/shlab
```

对于每个测试用例，采用 make testX 和 make rtestX 对每个用例按相同的顺序

测试，并将所有用例的运行结果分别输出到 tsh_ret 和 tshref_ret 文件当中，然后在运行结束后将两个文件当中的内容通过文本比对工具[9]进行比对。比对结果如下：



发现，我编写的 tsh 和参考的 reftsh 在运行 15 个测试用例后，输出的全部内容只有 PID、test 文件名、shell 名不同，其他全部一样，符合预期，证明我编写的 tsh 没有问题，15 个测试用例全部通过。

4.3.1 测试用例 trace01.txt

tsh 测试结果	tshref 测试结果
./sdriver.pl -t trace01.txt -s ./tsh -a "-p" # # trace01.txt - Properly terminate on EOF. #	./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
./sdriver.pl -t trace02.txt -s ./tsh -a "-p" # # trace02.txt - Process builtin quit command. #	./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

tsh 测试结果	tshref 测试结果
./sdriver.pl -t trace03.txt -s ./tsh -a "-p" # # trace03.txt - Run a foreground job. # tsh> quit	./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

tsh 测试结果	tshref 测试结果
./sdriver.pl -t trace04.txt -s ./tsh -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 & [1] (3252) ./myspin 1 &	./sdriver.pl -t trace04.txt -s ./tshref -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 & [1] (3023) ./myspin 1 &
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
./sdriver.pl -t trace05.txt -s ./tsh -a "-p" # # trace05.txt - Process jobs builtin command. # tsh> ./myspin 2 & [1] (3262) ./myspin 2 & tsh> ./myspin 3 & [2] (3264) ./myspin 3 & tsh> jobs [1] (3262) Running ./myspin 2 & [2] (3264) Running ./myspin 3 &	./sdriver.pl -t trace05.txt -s ./tshref -a "-p" # # trace05.txt - Process jobs builtin command. # tsh> ./myspin 2 & [1] (3053) ./myspin 2 & tsh> ./myspin 3 & [2] (3055) ./myspin 3 & tsh> jobs [1] (3053) Running ./myspin 2 & [2] (3055) Running ./myspin 3 &
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

./sdriver.pl -t trace06.txt -s ./tsh -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh> ./myspin 4 Job [1] (3276) terminated by signal 2	./sdriver.pl -t trace06.txt -s ./tshref -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh> ./myspin 4 Job [1] (3067) terminated by signal 2
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
./sdriver.pl -t trace07.txt -s ./tsh -a "-p" # # trace07.txt - Forward SIGINT only to foreground job. # tsh> ./myspin 4 & [1] (3284) ./myspin 4 & tsh> ./myspin 5 Job [2] (3286) terminated by signal 2 tsh> jobs [1] (3284) Running ./myspin 4 &	./sdriver.pl -t trace07.txt -s ./tshref -a "-p" # # trace07.txt - Forward SIGINT only to foreground job. # tsh> ./myspin 4 & [1] (3077) ./myspin 4 & tsh> ./myspin 5 Job [2] (3079) terminated by signal 2 tsh> jobs [1] (3077) Running ./myspin 4 &
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
./sdriver.pl -t trace08.txt -s ./tsh -a "-p" # # trace08.txt - Forward SIGTSTP only to foreground job. # tsh> ./myspin 4 & [1] (3301) ./myspin 4 & tsh> ./myspin 5 Job [2] (3303) stopped by signal 20 tsh> jobs [1] (3301) Running ./myspin 4 & [2] (3303) Stopped ./myspin 5	./sdriver.pl -t trace08.txt -s ./tshref -a "-p" # # trace08.txt - Forward SIGTSTP only to foreground job. # tsh> ./myspin 4 & [1] (3093) ./myspin 4 & tsh> ./myspin 5 Job [2] (3095) stopped by signal 20 tsh> jobs [1] (3093) Running ./myspin 4 & [2] (3095) Stopped ./myspin 5
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
./sdriver.pl -t trace09.txt -s ./tsh -a "-p" # # trace09.txt - Process bg builtin command # tsh> ./myspin 4 & [1] (3316) ./myspin 4 & tsh> ./myspin 5 Job [2] (3318) stopped by signal 20 tsh> jobs [1] (3316) Running ./myspin 4 & [2] (3318) Stopped ./myspin 5 tsh> bg %2 [2] (3318) ./myspin 5 tsh> jobs [1] (3316) Running ./myspin 4 & [2] (3318) Running ./myspin 5	./sdriver.pl -t trace09.txt -s ./tshref -a "-p" # # trace09.txt - Process bg builtin command # tsh> ./myspin 4 & [1] (3108) ./myspin 4 & tsh> ./myspin 5 Job [2] (3110) stopped by signal 20 tsh> jobs [1] (3108) Running ./myspin 4 & [2] (3110) Stopped ./myspin 5 tsh> bg %2 [2] (3110) ./myspin 5 tsh> jobs [1] (3108) Running ./myspin 4 & [2] (3110) Running ./myspin 5
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
./sdriver.pl -t trace10.txt -s ./tsh -a "-p" # # trace10.txt - Process fg builtin command. # tsh> ./myspin 4 & [1] (3333) ./myspin 4 & tsh> fg %1 Job [1] (3333) stopped by signal 20 tsh> jobs [1] (3333) Stopped ./myspin 4 & tsh> fg %1 tsh> jobs	./sdriver.pl -t trace10.txt -s ./tshref -a "-p" # # trace10.txt - Process fg builtin command. # tsh> ./myspin 4 & [1] (3125) ./myspin 4 & tsh> fg %1 Job [1] (3125) stopped by signal 20 tsh> jobs [1] (3125) Stopped ./myspin 4 & tsh> fg %1 tsh> jobs
测试结论	相同

4.3.11 测试用例 trace11.txt

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

tsh 测试结果	tshref 测试结果

./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] (3349) terminated by signal 2 tsh> /bin/ps a PID TTY STAT TIME COMMAND 1510 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu 1512 tty2 Sl+ 0:02 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 1528 tty2 Sl+ 0:00 /usr/libexec/gnome-session-binary --systemd --systemd --session=ubuntu 2861 pts/0 Ss+ 0:00 zsh 2965 pts/1 Ss 0:00 zsh 2995 pts/1 S+ 0:00 zsh 3342 pts/1 S+ 0:00 make test11 3343 pts/1 S+ 0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tsh -a "-p" 3344 pts/1 S+ 0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh -a -p 3347 pts/1 S+ 0:00 ./tsh -p 3355 pts/1 R 0:00 /bin/ps a	./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] (3142) terminated by signal 2 tsh> /bin/ps a PID TTY STAT TIME COMMAND 1510 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu 1512 tty2 Sl+ 0:02 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 1528 tty2 Sl+ 0:00 /usr/libexec/gnome-session-binary --systemd --systemd --session=ubuntu 2861 pts/0 Ss+ 0:00 zsh 2965 pts/1 Ss 0:00 zsh 2995 pts/1 S+ 0:00 zsh 3135 pts/1 S+ 0:00 make rtest11 3136 pts/1 S+ 0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tshref -a "-p" 3137 pts/1 S+ 0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tshref -a -p 3139 pts/1 S+ 0:00 ./tshref -p 3146 pts/1 R 0:00 /bin/ps a
测试结论	相同

4.3.12 测试用例 trace12.txt

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

tsh 测试结果	tshref 测试结果
./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] (3361) stopped by signal 20 tsh> jobs	./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] (3154) stopped by signal 20 tsh> jobs

[1] (3361) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND 1510 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu 1512 tty2 Sl+ 0:02 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 1528 tty2 Sl+ 0:00 /usr/libexec/gnome-session-binary --systemd --systemd --session=ubuntu 2861 pts/0 Ss+ 0:00 zsh 2965 pts/1 Ss 0:00 zsh 2995 pts/1 S+ 0:00 zsh 3356 pts/1 S+ 0:00 make test12 3357 pts/1 S+ 0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tsh -a "-p" 3358 pts/1 S+ 0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tsh -a -p 3359 pts/1 S+ 0:00 ./tsh -p 3361 pts/1 T 0:00 ./mysplit 4 3362 pts/1 T 0:00 ./mysplit 4 3370 pts/1 R 0:00 /bin/ps a	[1] (3154) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND 1510 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu 1512 tty2 Sl+ 0:02 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3 1528 tty2 Sl+ 0:00 /usr/libexec/gnome-session-binary --systemd --systemd --session=ubuntu 2861 pts/0 Ss+ 0:00 zsh 2965 pts/1 Ss 0:00 zsh 2995 pts/1 S+ 0:00 zsh 3149 pts/1 S+ 0:00 make rtest12 3150 pts/1 S+ 0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tshref -a "-p" 3151 pts/1 S+ 0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tshref -a -p 3152 pts/1 S+ 0:00 ./tshref -p 3154 pts/1 T 0:00 ./mysplit 4 3155 pts/1 T 0:00 ./mysplit 4 3160 pts/1 R 0:00 /bin/ps a
测试结论	相同

4.3.13 测试用例 trace13.txt

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

tsh 测试结果	tshref 测试结果
./sdriver.pl -t trace13.txt -s ./tsh -a "-p" # # trace13.txt - Restart every stopped process in process group # tsh> ./mysplit 4 Job [1] (3376) stopped by signal 20 tsh> jobs [1] (3376) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND	./sdriver.pl -t trace13.txt -s ./tshref -a "-p" # # trace13.txt - Restart every stopped process in process group # tsh> ./mysplit 4 Job [1] (3169) stopped by signal 20 tsh> jobs [1] (3169) Stopped ./mysplit 4 tsh> /bin/ps a PID TTY STAT TIME COMMAND

```
    1510 tty2        Ssl+     0:00
/usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu
/usr/bin/gnome-session --systemd
--session=ubuntu
    1512 tty2        Sl+      0:02
/usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1000/gdm/Xauthority -background
none -noreset -keeptty -verbose 3
    1528 tty2        Sl+      0:00
/usr/libexec/gnome-session-binary --systemd
--systemd --session=ubuntu
    2861 pts/0       Ss+      0:00 zsh
    2965 pts/1       Ss       0:00 zsh
    2995 pts/1       S+       0:00 zsh
    3371 pts/1       S+       0:00 make test13
    3372 pts/1       S+       0:00 /bin/sh
-c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
    3373 pts/1       S+       0:00
/usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a
-p
    3374 pts/1       S+       0:00 ./tsh -p
    3376 pts/1       T        0:00 ./mysplit 4
    3377 pts/1       T        0:00 ./mysplit 4
    3382 pts/1       R        0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
     PID TTY          STAT     TIME
COMMAND
    1510 tty2        Ssl+     0:00
/usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu
/usr/bin/gnome-session --systemd
--session=ubuntu
    1512 tty2        Sl+      0:02
/usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1000/gdm/Xauthority -background
none -noreset -keeptty -verbose 3
    1528 tty2        Sl+      0:00
/usr/libexec/gnome-session-binary --systemd
--systemd --session=ubuntu
    2861 pts/0       Ss+      0:00 zsh
    2965 pts/1       Ss       0:00 zsh
    2995 pts/1       S+       0:00 zsh
    3371 pts/1       S+       0:00 make test13
    3372 pts/1       S+       0:00 /bin/sh
-c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
    3373 pts/1       S+       0:00
/usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a
-p
    3374 pts/1       S+       0:00 ./tsh -p
```

```
    1510 tty2        Ssl+     0:00
/usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu
/usr/bin/gnome-session --systemd
--session=ubuntu
    1512 tty2        Sl+      0:02
/usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1000/gdm/Xauthority -background
none -noreset -keeptty -verbose 3
    1528 tty2        Sl+      0:00
/usr/libexec/gnome-session-binary --systemd
--systemd --session=ubuntu
    2861 pts/0       Ss+      0:00 zsh
    2965 pts/1       Ss       0:00 zsh
    2995 pts/1       S+       0:00 zsh
    3161 pts/1       S+       0:00 make rtest13
    3163 pts/1       S+       0:00 /bin/sh
-c ./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
    3164 pts/1       S+       0:00
/usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshref
-a -p
    3165 pts/1       S+       0:00 ./tshref -p
    3169 pts/1       T        0:00 ./mysplit 4
    3170 pts/1       T        0:00 ./mysplit 4
    3176 pts/1       R        0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
     PID TTY          STAT     TIME
COMMAND
    1510 tty2        Ssl+     0:00
/usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu
/usr/bin/gnome-session --systemd
--session=ubuntu
    1512 tty2        Sl+      0:02
/usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1000/gdm/Xauthority -background
none -noreset -keeptty -verbose 3
    1528 tty2        Sl+      0:00
/usr/libexec/gnome-session-binary --systemd
--systemd --session=ubuntu
    2861 pts/0       Ss+      0:00 zsh
    2965 pts/1       Ss       0:00 zsh
    2995 pts/1       S+       0:00 zsh
    3161 pts/1       S+       0:00 make rtest13
    3163 pts/1       S+       0:00 /bin/sh
-c ./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
    3164 pts/1       S+       0:00
/usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshref
-a -p
    3165 pts/1       S+       0:00 ./tshref -p
```

3388 pts/1 R 0:00 /bin/ps a	3180 pts/1 R 0:00 /bin/ps a
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
./sdriver.pl -t trace14.txt -s ./tsh -a "-p" # # trace14.txt - Simple error handling # tsh> ./bogus ./bogus: Command not found tsh> ./myspin 4 & [1] (3398) ./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] (3398) stopped by signal 20 tsh> bg %2 %2: No such job tsh> bg %1 [1] (3398) ./myspin 4 & tsh> jobs [1] (3398) Running ./myspin 4 &	./sdriver.pl -t trace14.txt -s ./tshref -a "-p" # # trace14.txt - Simple error handling # tsh> ./bogus ./bogus: Command not found tsh> ./myspin 4 & [1] (3190) ./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] (3190) stopped by signal 20 tsh> bg %2 %2: No such job tsh> bg %1 [1] (3190) ./myspin 4 & tsh> jobs [1] (3190) Running ./myspin 4 &
测试结论	相同

4.3.15 测试用例 trace15.txt

tsh 测试结果	tshref 测试结果

./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] (3423) terminated by signal 2 tsh> ./myspin 3 & [1] (3428) ./myspin 3 & tsh> ./myspin 4 & [2] (3430) ./myspin 4 & tsh> jobs [1] (3428) Running ./myspin 3 & [2] (3430) Running ./myspin 4 & tsh> fg %1 Job [1] (3428) stopped by signal 20 tsh> jobs [1] (3428) Stopped ./myspin 3 & [2] (3430) Running ./myspin 4 & tsh> bg %3 %3: No such job tsh> bg %1 [1] (3428) ./myspin 3 & tsh> jobs [1] (3428) Running ./myspin 3 & [2] (3430) Running ./myspin 4 & tsh> fg %1 tsh> quit	./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] (3215) terminated by signal 2 tsh> ./myspin 3 & [1] (3218) ./myspin 3 & tsh> ./myspin 4 & [2] (3220) ./myspin 4 & tsh> jobs [1] (3218) Running ./myspin 3 & [2] (3220) Running ./myspin 4 & tsh> fg %1 Job [1] (3218) stopped by signal 20 tsh> jobs [1] (3218) Stopped ./myspin 3 & [2] (3220) Running ./myspin 4 & tsh> bg %3 %3: No such job tsh> bg %1 [1] (3218) ./myspin 3 & tsh> jobs [1] (3218) Running ./myspin 3 & [2] (3220) Running ./myspin 4 & tsh> fg %1 tsh> quit
测试结论	相同

第 5 章 评测得分

总分 20 分

实验程序统一测试的评分（教师评价）：

（1）正确性得分：_____（满分 10）

（2）性能加权得分：_____（满分 10）

第 6 章 总结

5.1 请总结本次实验的收获

- 熟悉了信号的处理机制，学会了信号的阻塞、发送、接收的方法。
- 学会了用阻塞信号的方法来处理并发可能带来的“竞争”错误
- 学会了在 SIGCHLD 信号处理程序中回收垃圾子进程

5.2 请给出对本次实验内容的建议

- Tsh 的测试比较麻烦，实验缺少一个比较方便的**自动化**测试工具，我即使是编写了脚本自动化运行所有测试用例，仍然需要通过“文本比对”来对结果进行测试。对文本比对的内容进行过滤超过了我的能力范围，我没有花费更多的时间来研究这个实验自动化测试工具的编写。
- 实验报告的模板不知道为什么多了几个空白页，希望老师在给下一届学弟学妹发送报告模板时去掉空白页。

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.

[7] https://www.cnblogs.com/xiaomanon/p/4201006.html

[8] https://blog.csdn.net/daaikuaichuan/article/details/82782594

[9] https://www.jq22.com/textDifference