哈爾濱工業大學

实验报告

实 验（七）

题　　目 TinyShell

微壳

专　　业 计算机

学　　号 1190201423

班　　级 1903004

学　　生 顾海耀

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2021/6/2

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

理解现代计算机系统进程与并发的基本知识

掌握 linux 异常控制流和信号机制的基本原理和相关系统函数

掌握 shell 的基本原理和实现方法

深入理解 Linux 信号响应可能导致的并发冲突及解决方法

培养 Linux 下的软件系统开发与测试能力

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上;VirtualBox/Vmware 11 以上;Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.2.3 开发工具

Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind 等

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

了解进程、作业、信号的基本概念和原理

了解 shell 的基本原理

熟知进程创建、回收的方法和相关系统函数

熟知信号机制和信号处理相关的系统函数

第 2 章 实验预习

总分 20 分

2.1 进程的概念、创建和回收方法（5 分）

1.概念：指程序的依次运行过程。更确切说，进程是具有独立功能的一个程序关于某个数据集合的依次运行活动，进而进程具有动态含义。同一个程序处理不同的数据就是不同的进程。

2.创建方法：父进程通过调用 fork 函数创建一个新的运行的子进程，对于函数 int fork(void):子进程中，fork 返回 0；父进程中，返回子进程的 PID；新创建的子进程几乎但不完全与父进程相同，子进程得到与父进程虚拟地址空间相同的但是独立的一份副本，子进程获得与父进程任何打开文件描述符相同的副本，最大区别是子进程有不同于父进程的 PID。

3.回收进程：当进程终止时，它仍然消耗系统资源，被称为“僵尸 zombie”进程。父进程通过 wait 函数回收子进程，对于函数 int wait(int *child_status)：挂起当前进程的执行直到它的一个子进程终止，返回已终止子进程的 PID。

2.2 信号的机制、种类（5 分）

1.信号的机制：signal 就是一条小消息，它通知进程系统中发生了一个某种类型的事件：

类似于异常和中断;

从内核发送到（有时是在另一个进程的请求下）一个进程;

信号类型是用小整数 ID 来标识的(1-30);

信号中唯一的信息是它的 ID 和它的到达。

包括发送信号：内核通过更新目的进程上下文中的某个状态，发送（递送）一个信号给目的进程；接受信号：当目的进程被内核强迫以某种方式对信号的发送做出反应时，它就接收了信号。

发送信号的原因： 内核检测到一个系统事件如除零错误(SIGFPE)或者子进程终止(SIGCHLD)；一个进程调用了 kill 系统调用，显式地请求内核发送一个信号到目的进程

反应的方式：

忽略这个信号(do nothing)；终止进程(with optional core dump)；通过执行一个称为信号处理程序（signal handler）的用户层函数捕获这个信号

2.信号种类

编号	信号名称	缺省动作	说明
1	SIGHUP	终止	终止控制终端或进程
2	SIGINT	终止	键盘产生的中断(Ctrl-C)
3	SIGQUIT	dump	键盘产生的退出
4	SIGILL	dump	非法指令
5	SIGTRAP	dump	debug 中断
6	SIGABRT / SIGIOT	dump	异常中止
7	SIGBUS / SIGEMT	dump	总线异常/EMT 指令
8	SIGFPE	dump	浮点运算溢出
9	SIGKILL	终止	强制进程终止
10	SIGUSR1	终止	用户信号,进程可自定义用途
11	SIGSEGV	dump	非法内存地址引用
12	SIGUSR2	终止	用户信号，进程可自定义用途
13	SIGPIPE	终止	向某个没有读取的管道中写入数据
14	SIGALRM	终止	时钟中断(闹钟)
15	SIGTERM	终止	进程终止
16	SIGSTKFLT	终止	协处理器栈错误
17	SIGCHLD	忽略	子进程退出或中断
18	SIGCONT	继续	如进程停止状态则开始运行
19	SIGSTOP	停止	停止进程运行
20	SIGSTP	停止	键盘产生的停止
21	SIGTTIN	停止	后台进程请求输入
22	SIGTTOU	停止	后台进程请求输出
23	SIGURG	忽略	socket 发生紧急情况
24	SIGXCPU	dump	CPU 时间限制被打破
25	SIGXFSZ	dump	文件大小限制被打破
26	SIGVTALRM	终止	虚拟定时时钟
27	SIGPROF	终止	profile timer clock
28	SIGWINCH	忽略	窗口尺寸调整
29	SIGIO/SIGPOLL	终止	I/O 可用
30	SIGPWR	终止	电源异常
31	SIGSYS / SYSUNUSED	dump	系统调用异常

2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分）

一. 发送信号

内核通过更新目的进程上下文中的某个状态，发送（递送）一个信号给目的进程.

发送方法：

用 /bin/kill 程序发送信号

/bin/kill 程序可以向另外的进程或进程组发送任意的信号

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

/bin/kill – 9 – 24817 发送信号 SIGKILL 给进程组 24817 中的每个进程（负的 PID 会导致信号被发送到进 程组 PID 中的每个进程）

2.从键盘发送信号输入 ctrl-c (ctrl-z) 会导致内核发送一个 SIGINT(SIGTSTP)信号到前台进程组中的每个作业 SIGINT 默认情况是终止前台作业 SIGTSTP 默认情况是停止（挂起）前台作业。

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

(1)kill()和 raise()

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

(2)函数格式：

kill 函数的语法格式：

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

raise()函数语法要点：

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

(3)alarm()和 pause()

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

pause()----此函数用于将进程挂起直到捕捉到信号为止，这个函数很常用，

通常用于判断信号是否已到

alarm()函数语法：

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

pause()函数语法如下：

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

二. 阻塞信号

阻塞和解除阻塞信号

隐式阻塞机制 ：

内核默认阻塞与当前正在处理信号类型相同的待处理信号 如：一个 SIGINT 信号处理程序不能被另一个 SIGINT 信号中断（此时另一个 SIGINT 信号被阻塞）

显示阻塞和解除阻塞机制 ：

sigprocmask 函数及其辅助函数可以明确地阻塞/解除阻塞

选定的信号辅助函数：

sigempty(set) – 初始化 set 为空集合

sigfill(set) – 把每个信号都添加到 set 中

sigadd(set) – 把指定的信号 signum 添加到 set

sigdel(set) – 从 set 删除指定的信号 signum

三. 设置信号处理程序

可以使用 signal 函数修改和信号 signum 相关联的默认行为：handler_t *signal(int signum, handler_t *handler)

handler 的不同取值:

1. SIG_IGN: 忽略类型为 signum 的信号

2. SIG_DFL: 类型为 signum 的信号行为恢复为默认行为

3. 否则，handler 就是用户定义的函数的地址，这个函数称为信号处理程序

只要进程接收到类型为 signum 的信号就会调用信号处理程序将处理程序的地址传递到 signal 函数从而改变默认行为，这叫作设置信号处理程序。调用信号处理程序称为捕获信号执行信号处理程序称为处理信号当处理程序执行 return 时，控制会传递到控制流中被信号接收所中断的指令处

2.4 什么是 shell，功能和处理流程（5 分）

1 定义:

shell 是一个交互型应用级程序,代表用户运行其他程序。是系统的用户界面,提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。

2.功能:

其实 shell 也是一支程序,它由输入设备读取命令,再将其转为计算机可以了解的机械码,然后执行它。各种操作系统都有它自己的 shell,以 DOS 为例,它的 shell 就是 command.com 文件。如同 DOS 下有 NDOS, 4DOS, DRDOS 等不同的命令解译程序可以取代标准的 command.com ,UNIX 下除了 Bourne shell（/bin/sh）外还有 C shell（/bin/csh）、 Korn shell（/bin/ksh）、 Bourne again shell（/bin/bash）、Tenex C shell（tcsh）等其它的 shell。UNIX/linux 将 shell 独立于核心程序之外,使得它就如同一般的应用程序,可以在不影响操作系统本身的情况下进行修改、更新版本或是添加新的功能。Shell 是一个命令解释器,它解释由用户输入的命令并且把它们送到内核。不仅如此,Shell 有自己的编程语言用于对命令的编辑,它允许用户编写由 shell 命令组成的程序。Shell 编程语言具有普通编程语言的很多特点,比如它也有循环结构和分支控制结构等,用这种编程语言编写的 Shell 程序与其他应用程序具有同样的效果

3.处理流程:

shell 首先检查命令是否是内部命令,若不是再检查是否是一个应用程序（这里的应用程序可以是 Linux 本身的实用程序,如 ls 和 rm,也可以是购买的商业程序,如 xv,或者是自由软件,如 emacs）。然后 shell 在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果键入的命令不是一个内部命令并且在路径里没有找到这个可执行文件,将会显示一条错误信息。如果能够成功找到命令,该内部命令或应用程序将被分解为系统调用并传给 Linux 内核。

第 3 章 TinyShell 的设计与实现

总分 45 分

3.1 设计

3.1.1 void eval(char *cmdline)函数（10 分）

函数功能：对用户输入的命令进行解析
参　　数：char *cmdline
处理流程：

1. 调用 parseline 函数对命令进行切分，返回 bg 说明是否为后台程序
2. 调用 build_cmd 函数判断该输入行是否为内置命令，若是，则直接执行
3. 若不是内置命令，创建子进程，利用 execve 函数进行命令执行

要点分析：

1. 所有子进程应该有唯一 PID，否则输入 Ctrl-C/Z 时，后台子进程会从内核获取信号：SIGINT，为了避免这种错误，则应该所有子进程应该有唯一 PID
2. 最开始应该设置阻塞信号，让信号可以发送但不会被接收，避免把不存在的进行加载在作业列表中
3. 需要用 sigprocmask 解除阻塞

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

函数功能：判断用户是否输入内置命令
参　　数：char **argv
处理流程：

1. 对于 quit，直接 exit(0)推出即可
2. 对于 jobs，调用 listjobs 函数
3. 对于 fg/bg，调用 do_bgfg 函数
4. 对于&，什么都不用做

要点分析：

1. 注意使用已经给出的函数

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

函数功能：对 builtin_cmd 的函数中输入 fg/bg 的情况进行处理
参　　数：char **argv

处理流程：

1. 先进行格式判断，通过获取 jobs 的 JID
2. bg 命令：向 job 所在进程组发送 SIGINT 信号，更改 job 的 state 为 BG
3. fg 命令：向 job 所在进程组发送 SIGINT 信号，更改 job 的 state 为 FG，等待当前 job 不再是前台程序

要点分析：

1.注意对错误的分析

3.1.4 void waitfg(pid_t pid) 函数（5 分）

函数功能：阻塞进程，直到进程 pid 不再是前台进程
参　　数：pid_t pid
处理流程：

1. 若当前进程 PID 与当前前台进程的 PID 相等，则进行挂起即可

要点分析：

1.注意循环条件，使用 PID 比较进行判断是否要继续挂起

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

函数功能：回收‘僵尸进程’
参　　数：int sig
处理流程：

1. 先获取所有信号
2. 如果子进程停止，则可以直接返回
3. 如果子进程未停止，终止子进程并进行回收

要点分析：

1. 注意先判断子进程是否已经终止

3.2 程序实现（tsh.c 的全部内容）（10 分）

重点检查代码风格：

（1）用较好的代码注释说明——5 分

（2）检查每个系统调用的返回值——5 分

```
/*
 * tsh - A tiny shell program with job control
 *
 * <Put your name and login ID here>
```

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

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

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

/*
 * Jobs states: FG (foreground), BG (background), ST (stopped)
 * Job state transitions and enabling actions:
```

```
 *         FG -> ST    : ctrl-z
 *         ST -> FG    : fg command
 *         ST -> BG    : bg command
 *         BG -> FG    : fg command
 * At most 1 job can be in the FG state.
 */

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

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


/* Function prototypes */
```

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

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

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

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

void usage(void);
```

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

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

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

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

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

  /* Install the signal handlers */

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

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

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

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

/* Read command line */
```

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

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

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

/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
```

```
 * the foreground, wait for it to terminate and then return.  Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
*/
void eval(char *cmdline)
{
    /* $begin handout */
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */
    sigset_t mask;       /* signal mask */

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

    if (!builtin_cmd(argv)) {

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

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

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

/*
 * Child    process
 */

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

        /* Each new job must get a new process group ID
           so that the kernel doesn't send ctrl-c and ctrl-z
```

```
            signals to all of the shell's jobs */
        if (setpgid(0, 0) < 0)
        unix_error("setpgid error");

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

/*
  * Parent process
  */

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

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

```
    return;
}

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

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

    /* Build the argv list */
    argc = 0;
    if (*buf == '\'') {
```

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

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

if (*buf == '\"') {
        buf++;
        delim = strchr(buf, '\"');
}
else {
        delim = strchr(buf, ' ');
}
  }
  argv[argc] = NULL;

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

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

/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *       it immediately.
 */
int builtin_cmd(char **argv)
{
    if(!strcmp(argv[0],"&"))    //后台执行
    {
  return 1;
    }

    if(!strcmp(argv[0],"quit"))    //退出
    {
  exit(0);
    }

    if(!strcmp(argv[0],"jobs"))    //打印作业列表
    {
```

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

     if(!strcmp(argv[0],"bg") || !strcmp(argv[0],"fg"))    //调用 do_bgfg
     {
    do_bgfg(argv);
    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)
{
    while(pid == fgpid(jobs))     //若当前进程 PID 与当前前台进程的 PID 相等，则进行挂起
    {
```

```
    sleep(1);//休眠挂起
     }
     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)
{
     //定义局部变量
     sigset_t sigg,pr_sigg;
     struct job_t* jobb;
     int err = errno;
     int stat;
     pid_t pid;
     sigfillset(&sigg);     //将所有信号添加到 sigg 中
     while((pid = waitpid(-1,&stat,WNOHANG | WUNTRACED)) > 0)  //回收子
```

进程

```
    {
    sigprocmask(SIG_BLOCK,&sigg,&pr_sigg);   //阻塞所有信号
    jobb = getjobpid(jobs,pid);     //通过 pid 找到操作 job
    if (WIFSTOPPED(stat)) //如果子进程停止则返回
    {
        jobb->state = ST;
        printf("Job [%d] (%d) terminated by signal %d\n", jobb->jid, jobb->pid,
WSTOPSIG(stat));
    }
    else
    {
        if (WIFSIGNALED(stat))   //子进程终止
        {
            printf("Job [%d] (%d) terminated by signal %d\n", jobb->jid,
jobb->pid, WTERMSIG(stat));
        }
        deletejob(jobs, pid);   //终止进程直接回收
    }
    fflush(stdout);
    sigprocmask(SIG_SETMASK, &pr_sigg, NULL);
    }
    errno = err;
    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)
{
    //定义局部变量
    int err = errno;
    pid_t pid;
    sigset_t sigg, pr_sigg;
    sigfillset(&sigg);
    sigprocmask(SIG_BLOCK, &sigg, &pr_sigg);  //阻塞所有信号
    pid = fgpid(jobs);  //获取前台作业 pid
    sigprocmask(SIG_SETMASK, &pr_sigg, NULL);
    if (pid != 0)  //处理前台作业
        kill(-pid, SIGINT);
    errno = err;
    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)
{
        //定义局部变量
    int err = errno;
    pid_t pid;
    sigset_t sigg, pr_sigg;
    sigfillset(&sigg);
    sigprocmask(SIG_BLOCK, &sigg, &pr_sigg);   //阻塞所有信号
    pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &pr_sigg, NULL);
    if (pid != 0)
        kill(-pid, SIGTSTP);
    errno = err;
    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)，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。

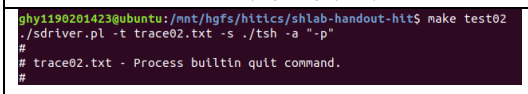
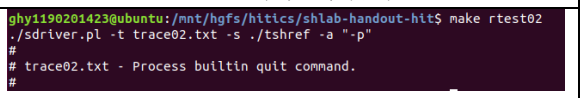
4.3 自测试结果

填写以下各个测试用例的测试结果，每个测试用例 1 分。

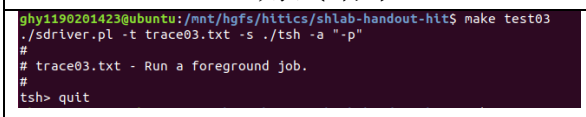
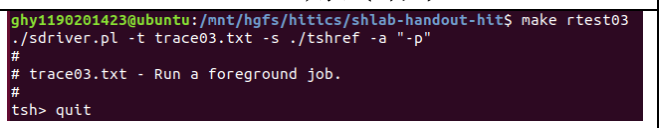
4.3.1 测试用例 trace01.txt

<table>
<tr><td>tsh 测试结果</td><td>tshref 测试结果</td></tr>
<tr><td>ghy1190201423@ubuntu:/mnt/hgfs/hitics/shlab-handout-hit$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
trace01.txt - Properly terminate on EOF.
#</td><td>ghy1190201423@ubuntu:/mnt/hgfs/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/不同，原因分析如下：相同</td></tr>
</table>

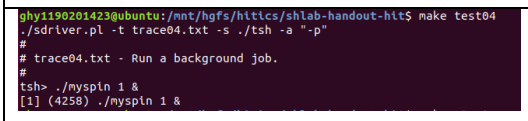
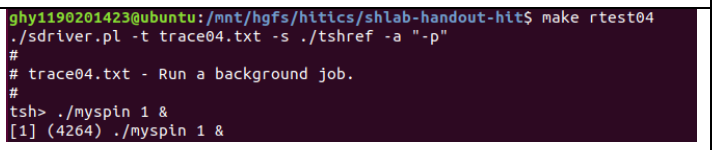
4.3.2 测试用例 trace02.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

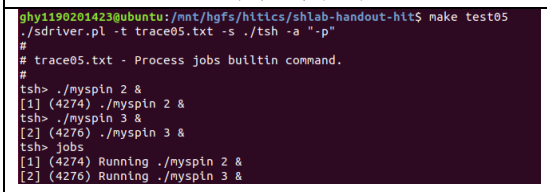
4.3.3 测试用例 trace03.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

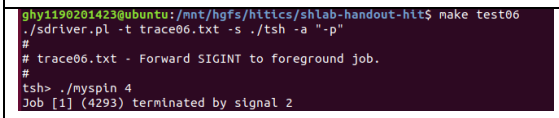
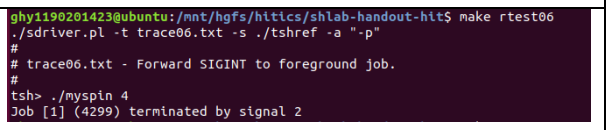
4.3.4 测试用例 trace04.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

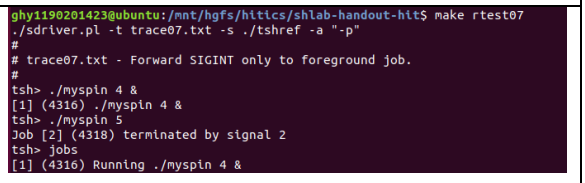
4.3.5 测试用例 trace05.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

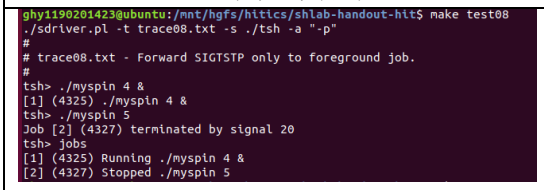
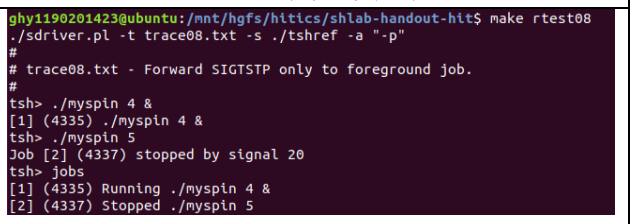
4.3.6 测试用例 trace06.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

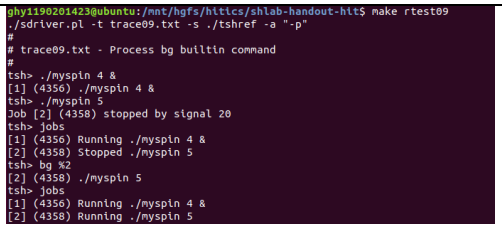
4.3.7 测试用例 trace07.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：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/不同，原因分析如下：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/不同，原因分析如下：相同

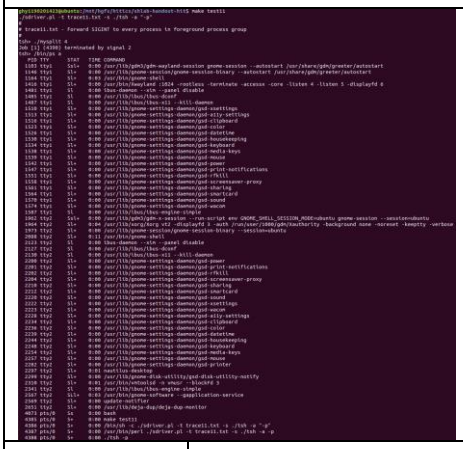
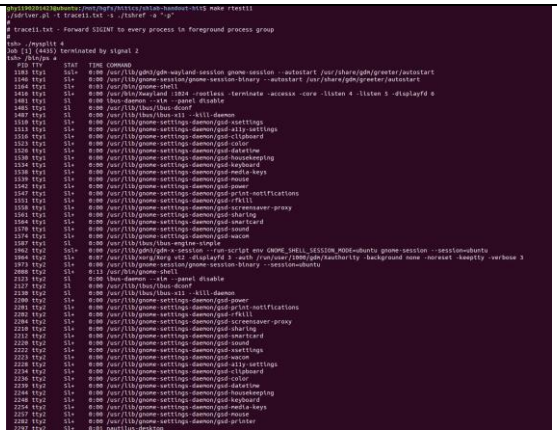
4.3.10 测试用例 trace10.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

4.3.11 测试用例 trace11.txt

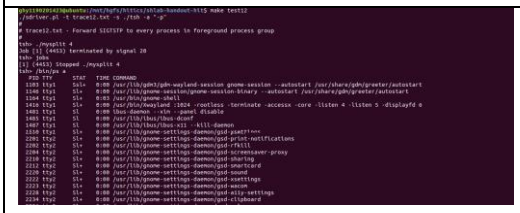
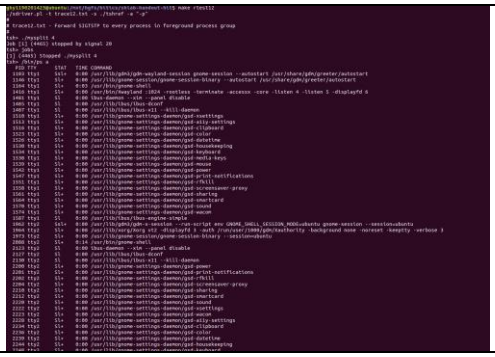
测试中 ps 指令的输出内容较多，仅记录和本实验密切相关的 tsh、

mysplit 等进程的部分信息即可。

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

4.3.12 测试用例 trace12.txt

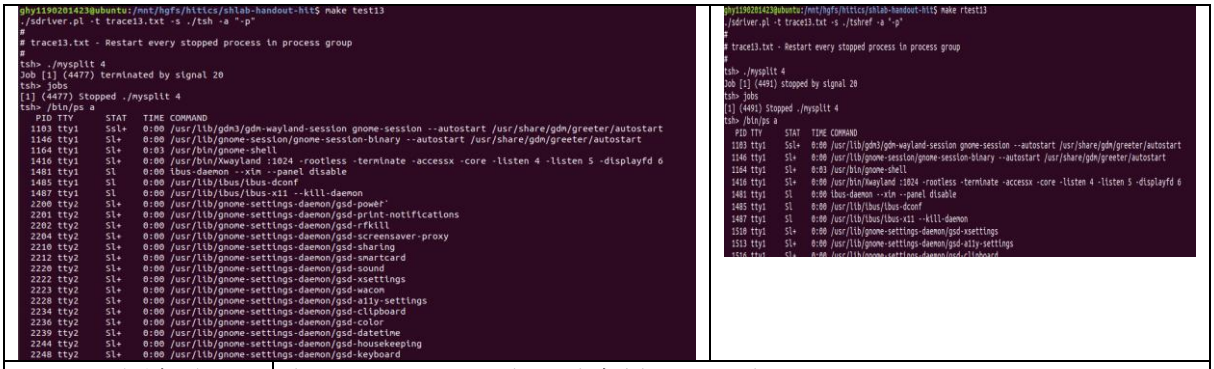
测试中 ps 指令的输出内容较多，仅记录和本实验密切相关的 tsh、mysplit 等进程的部分信息即可。

tsh 测试结果	tshref 测试结果
	
测试结论	相同/不同，原因分析如下：相同

4.3.13 测试用例 trace13.txt

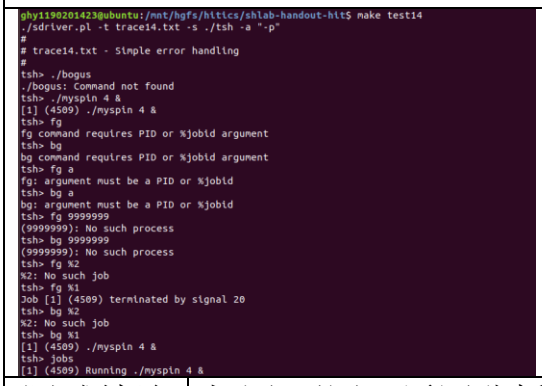
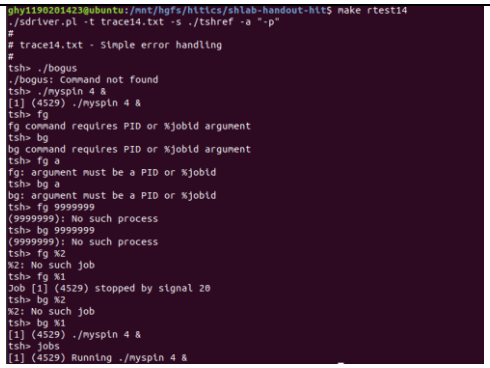
测试中 ps 指令的输出内容较多，仅记录和本实验密切相关的 tsh、mysplit 等进程的部分信息即可。

tsh 测试结果	tshref 测试结果



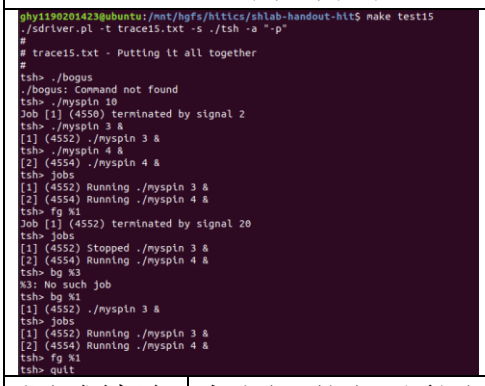
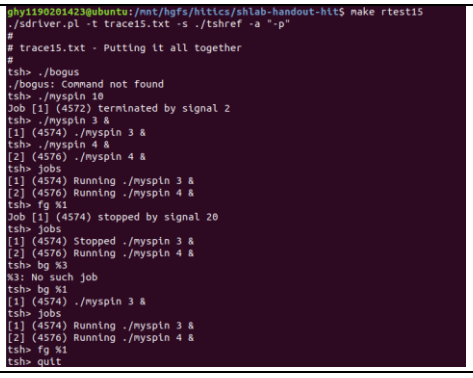
测试结论	相同/不同，原因分析如下：相同

4.3.14 测试用例 trace14.txt

tsh 测试结果	tshref 测试结果
	

测试结论	相同/不同，原因分析如下：相同

4.3.15 测试用例 trace15.txt

tsh 测试结果	tshref 测试结果
	

测试结论	相同/不同，原因分析如下：相同

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：10 （满分 10）

（2）性能加权得分：10 （满分 10）

第 6 章 总结

5.1 请总结本次实验的收获

1.理解了 shell 的工作原理。
2.了解了信号的处理机制，对一些信号的函数有更深的了解
3.明白了信号处理相关的系统函数的作用

5.2 请给出对本次实验内容的建议

无

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.