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 _____

班　　级 _____

学　　生 _____

指 导 教 师 _____

实 验 地 点 _____

实 验 日 期 2021.6.3

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

Ubuntu，gcc

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）

进程的概念：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。

进程的创建：在 unix 中系统调用的是：fork，fork 会建立一个和父进程基本一模一样的子进程，在 windows 中该系统调用是：cresteprocess，既负责处理进程的创建，也负责把正确的程序装入新进程。具体可以分为如下步骤：

1．系统的初始化

2．一个进程在运行过程中开启了子进程

3．用户的交互式请求，而创建一个新进程（如双击应用图标）

4．一个批处理作业的初始化（只在大型机的批处理系统中应用）

关于进程的创建，UNIX 和 WINDOWS

进程的回收：由于代码编写的不完善，随着时间的推移，应用程序的性能会越来越低，有时会陷于某一循环中，导致不必要的 CPU 负载。这些应用程序还可能导致内存泄漏，这时应用程序不再将不需要的内存释放回操作系统。这些应用程序可能会导致服务器停止运行，因此需要重新启动服务器。进程回收就是为解决这些问题而创建的。子进程有父进程回收，孤儿进程有 init 回收，没有及时回收则出现僵尸进程。

2.2 信号的机制、种类（5 分）

可以从两个不同的分类角度对信号进行分类：

可靠性方面：可靠信号与不可靠信号；

与时间的关系上：实时信号与非实时信号。

①可靠信号与不可靠信号

Linux 信号机制基本上是从 Unix 系统中继承过来的。早期 Unix 系统中的信号

机制比较简单和原始，信号值小于 SIGRTMIN 的信号都是不可靠信号。这就是"不可靠信号"的来源。它的主要问题是信号可能丢失。

随着时间的发展，实践证明了有必要对信号的原始机制加以改进和扩充。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失。

信号值位于 SIGRTMIN 和 SIGRTMAX 之间的信号都是可靠信号,可靠信号克服了信号可能丢失的问题。Linux 在 支持新版本的信号安装函数 sigation()以及信号发送函数 sigqueue()的同时，仍然支持早期的 signal()信号安装函数，支持信号发送 函数 kill()。

信号的可靠与不可靠只与信号值有关，与信号的发送及安装函数无关。目前 linux 中的 signal()是通过 sigation()函数实现的，因此，即使通过 signal()安装的信号，在信号处理函数的结尾也不必再调用一次信号安装函数。同时，由 signal()安装的实时信号支持排队，同样不会丢失。

②实时信号与非实时信号

早期 Unix 系统只定义了 32 种信号，前 32 种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的 CTRL^C 时，会产生 SIGINT 信号，对该信号的默认反应就是进程终止。后 32 个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收。

非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号。

2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分）

进程可以通过 kill 函数向包括它本身在内的其他进程发送一个信号，如果程序没有发送这个信号的权限，对 kill 函数的调用就将失败，而失败的常见原因是目标进程由另一个用户所拥有。

阻塞方法：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和待处理信号。显示阻塞进制：应用程序可以调用 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：

1） 调用 signal 函数，调用 signal(SIG,handler)，SIG 代表信号类型，handler 代表接收到 SIG 信号之后对应的处理程序。

2） 因为 signal 的语义各有不同，所以我们需要一个可移植的信号处理函

数设置方法，Posix 标准定义了 sigaction 函数，它允许用户在设置信号处理时明确指定他们想要的信号处理语义

2.4 什么是 shell，功能和处理流程（5 分）

什么是 shell：Shell 是一个用 C 语言编写的程序，他是用户使用 Linux 的桥梁。Shell 既是一种命令语言，又是一种程序设计语言，Shell 是指一种应用程序。

功能：Shell 应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

处理流程：

1）从终端读入输入的命令。

2）将输入字符串切分获得所有的参数

3）如果是内置命令则立即执行

4 否则调用相应的程序执行

5）shell 应该接受键盘输入信号，并对这些信号进行相应处理。

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：判断用户刚输入的命令行是否存在内置命令，如果是的话立即执行。
参　　数：char *cmdline
处理流程：1.在初始化一些变量之后，首先调用 parseline 函数解析输出后，判断这是一个内置命令还是一个程序。
2.如果是内置命令，进入 built_cmd 函数，执行内置命令。否则 fork 一个子进程并添加在 jobs 列表里。

要点分析：
在 addjob 之前需要进行信号阻塞，防止把不存在的子进程添加到列表里

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

函数功能：通过比较确定需要执行哪个内置命令
参　　数：char **argv
处理流程：通过几个 if 语句来匹配根据输入的命令行的指令，需要执行哪一个内置命令，并调用该内置命令的函数，执行结束后返回。例如：如果命令行的第一个命令是 quit，则直接退出函数，其余情况类似。
要点分析：需要注意在 listjobs 之前需要对信号进行阻塞，防止 jobs 被信号修改，保证函数的准确性。

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

函数功能：实现内置命令 bg 和 fg.
参　　数：char **argv
处理流程：
第一步：先判断 fg 和 bg 命令后是否有参数，若没有忽略此条命令。
第二步：如果 fg 或 bg 后面只是数字，说明取的是进程号，获取该进程号后，使用 getjobpid(jobs, pid)得到 job；如果 fg 或 bg 后面是%加上数字的形式，说明%后面是

任务号，此时获取 jid 后，可以使用 getjobjid(jobs, jid)得到 job。因此第二步之后得到了一个 job。
第三步：根据 fg 和 bg 两种命令的不同根据第二步获得的 job 获得这个函数需要返回的 pid 的值。
要点分析：
1. 注意需要使用 atoi 来把字符串转化为一个整数以供调用。
2. 函数需要先判断是否需要执行命令，如果需要执行的话还需要根据两个不同的命令使用两种不同的方式获得 job 并获得最终的 pid 的值。
3. 如果遇到了 SIGCONT 我们需要做的就是继续停止的进程。

3.1.4 void waitfg(pid_t pid) 函数（5 分）

函数功能：等待前台程序结束
参　　数：pid_t pid
处理流程：判断前台程序是否结束，如果还没结束的话继续等待直到收到进程终止的信号跳出循环，返回。
要点分析：在 while 内部，如果使用的只是 pause()函数，那么程序必须等待相当长的一段时间才会再次检查循环的终止条件，如果使用像 nanosleep 这样的高精度休眠函数也是不可接受的，因为没有很好的办法来确定休眠的间隔。

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

函数功能：捕获 SIGCHILD 信号.
参　　数：int sig
处理流程：首先将所有的 errno 存储，以便在获得需要的信号之后恢复，接着将所有的信号加入 mask，并进入一个 while 循环，准备进行循环，并在循环中回收子进程，如果等待集合中没有进程被中止或停止返回 0，否则孩子返回进程的 pid。接着，在循环中阻塞信号，并且使用 getjobpid()函数，通过 pid 找到 job。根据不同的条件判断需要采用哪一种输出形式与返回形式。在确定之后解除阻塞信号，恢复 errno，并返回。
要点分析：
1. while 循环来避免信号阻塞的问题，循环中使用 waitpid()函数，以尽可能多的回收僵尸进程。
2. deletejobs 时需要先阻塞信号。

3.2 程序实现（tsh.c 的全部内容）（10 分）

重点检查代码风格：

（1）用较好的代码注释说明——5 分

（2）检查每个系统调用的返回值——5 分

```
/*
 * tsh - A tiny shell program with job control
 *
 * <Put your name and login ID here>
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

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

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

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

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

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

```
/* Function prototypes */

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

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

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

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

```
int pid2jid(pid_t pid);
void listjobs(struct job_t* jobs);

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

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

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

    /* Parse the command line */
    while ((c = getopt(argc, argv, "hvp")) != EOF) {
        switch (c) {
        case 'h':                   /* print help message */
```

```
            usage();
            break;
        case 'v':             /* emit additional diagnostic info */
            verbose = 1;
            break;
        case 'p':             /* don't print a prompt */
            emit_prompt = 0;  /* handy for automatic testing */
            break;
        default:
            usage();
        }
}

/* Install the signal handlers */

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

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

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

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

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

        /* Evaluate the command line */
        eval(cmdline);
        fflush(stdout); //清空缓冲区并输出
        fflush(stdout);
    }

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

/*
 * eval - Evaluate the command line that the user has just typed in
```

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

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

    if (!builtin_cmd(argv)) {

        /*
     * This is a little tricky. Block SIGCHLD, SIGINT, and SIGTSTP
```

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

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

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

    /*
     * Child    process
     */

    if (pid == 0) {
        /* Child unblocks signals  解除阻塞*/
```

```
        sigprocmask(SIG_UNBLOCK, &mask, NULL);

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

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

/*
 * Parent process
 */

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

if (!bg)
    waitfg(pid);
```

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

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

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

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

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

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

```
    argv[argc] = NULL;

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

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

/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *       it immediately.
    builtin_cmd   - 如果用户键入了内置命令，则立即执行。
 */
int builtin_cmd(char** argv)
{
    sigset_t mask, prev_mask;
    sigfillset(&mask);
    if (!strcmp(argv[0], "quit"))     //如果命令行的首个命令是 quit 直接退出进程
        exit(0);
    else if (!strcmp(argv[0], "&"))        //命令行的第一个字符是&的时候忽略这一条命令
```

```
        return 1;
    else if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg"))//处理 bg 和 fg
    {
        do_bgfg(argv);
        return 1;
    }
    else if (!strcmp(argv[0], "jobs"))    //输出所有 job 信息
    {
        sigprocmask(SIG_BLOCK, &mask, &prev_mask); //由于 jobs 是全局变量，需要阻塞信号
        listjobs(jobs);
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
        return 1;
    }

    return 0;         /* not a builtin command */
}

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

```
/* Ignore command if no argument
如果没有参数，则忽略命令*/
if (argv[1] == NULL) {
    printf("%s command requires PID or %%jobid argument\n", argv[0]);
    return;
}

/* Parse the required PID or %JID arg */
if (isdigit(argv[1][0])) //判断串 1 的第 0 位是否为数字
{
    pid_t pid = atoi(argv[1]);    //atoi 把字符串转化为整型数
    if (!(jobp = getjobpid(jobs, pid))) {
        printf("(%d): No such process\n", pid);
        return;
    }
}
else if (argv[1][0] == '%') {
    int jid = atoi(&argv[1][1]);
    if (!(jobp = getjobjid(jobs, jid))) {
        printf("%s: No such job\n", argv[1]);
        return;
    }
}
else {
    printf("%s: argument must be a PID or %%jobid\n", argv[0]);
```

```
        return;
    }

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

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

```
/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid) //传入的是一个前台进程的 pid
{
    sigset_t mask;
    sigemptyset(&mask);   //初始化 mask 为空集合
    while (pid == fgpid(jobs))
    {
        sigsuspend(&mask);   //暂时挂起进程，比 pause 方法更准确一些
    }
}

/*****************
 * 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)
{
```

```
    struct job_t* job1;
    int olderrno = errno, status;
    sigset_t mask, prev_mask;
    pid_t pid;
    sigfillset(&mask);
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        //通过这个循环能实现尽可能多地回收子进程
        sigprocmask(SIG_BLOCK, &mask, &prev_mask);  //由于 jobs 是全局变量，因此 delete 的时候需要阻塞所有的信号
        job1 = getjobpid(jobs, pid);  //通过 pid 找到 job
        if (WIFSTOPPED(status)) //子进程停止引起的 waitpid 函数返回
        {
            job1->state = ST;
            printf("Job [%d] (%d) terminated by signal %d\n", job1->jid, job1->pid, WSTOPSIG(status));
        }
        else
        {
            if (WIFSIGNALED(status)) //子进程终止引起返回
                printf("Job [%d] (%d) terminated by signal %d\n", job1->jid, job1->pid, WTERMSIG(status));
            deletejob(jobs, pid);  //直接回收进程
        }
        fflush(stdout);
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

```
    }
    errno = olderrno;
}

/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenver the
 *      user types ctrl-c at the keyboard.  Catch it and send it along
 *      to the foreground job.
 */
void sigint_handler(int sig)
{
    pid_t pid;
    sigset_t mask, prev_mask;
    int olderrno = errno;
    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask);  //阻塞信号
    pid = fgpid(jobs);  //获取 job 的 pid
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    if (pid != 0)  //只处理前台 job
        kill(pid, SIGINT);
    errno = olderrno;
    return;
}

/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
```

```
 *     the user types ctrl-z at the keyboard. Catch it and suspend the
 *     foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    pid_t pid;
    sigset_t mask, prev_mask;
    int olderrno = errno;
    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask);  //阻塞信号
    pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    if (pid != 0)
        kill(-pid, SIGTSTP);
    errno = olderrno;
    return;
}

/*********************
 * End signal handlers
 *********************/

 /***********************************************
  * Helper routines that manipulate the job list
  **********************************************/
```

```
   /* clearjob - Clear the entries in a job struct */
void clearjob(struct job_t* job) {
        job->pid = 0;
        job->jid = 0;
        job->state = UNDEF;
        job->cmdline[0] = '\0';
}

/* initjobs - Initialize the job list */
void initjobs(struct job_t* jobs) {
        int i;

        for (i = 0; i < MAXJOBS; i++)
                clearjob(&jobs[i]);
}

/* maxjid - Returns largest allocated job ID */
int maxjid(struct job_t* jobs)
{
        int i, max = 0;

        for (i = 0; i < MAXJOBS; i++)
                if (jobs[i].jid > max)
                        max = jobs[i].jid;
        return max;
}
```

```
/* addjob - Add a job to the job list */
int addjob(struct job_t* jobs, pid_t pid, int state, char* cmdline)
{
    int i;

    if (pid < 1)
        return 0;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid == 0) {
            jobs[i].pid = pid;
            jobs[i].state = state;
            jobs[i].jid = nextjid++;
            if (nextjid > MAXJOBS)
                nextjid = 1;
            strcpy(jobs[i].cmdline, cmdline);
            if (verbose) {
                printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid,
jobs[i].cmdline);
            }
            return 1;
        }
    }
    printf("Tried to create too many jobs\n");
    return 0;
}
```

```
/* deletejob - Delete a job whose PID=pid from the job list */
int deletejob(struct job_t* jobs, pid_t pid)
{
    int i;

    if (pid < 1)
        return 0;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid == pid) {
            clearjob(&jobs[i]);
            nextjid = maxjid(jobs) + 1;
            return 1;
        }
    }
    return 0;
}

/* fgpid - Return PID of current foreground job, 0 if no such job */
pid_t fgpid(struct job_t* jobs) {
    int i;

    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].state == FG)
            return jobs[i].pid;
```

```
    return 0;
}

/* getjobpid    - Find a job (by PID) on the job list */
struct job_t* getjobpid(struct job_t* jobs, pid_t pid) {
    int i;

    if (pid < 1)
        return NULL;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].pid == pid)
            return &jobs[i];
    return NULL;
}

/* getjobjid    - Find a job (by JID) on the job list */
struct job_t* getjobjid(struct job_t* jobs, int jid)
{
    int i;

    if (jid < 1)
        return NULL;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].jid == jid)
            return &jobs[i];
    return NULL;
```

```
}

/* pid2jid - Map process ID to job ID */
int pid2jid(pid_t pid)
{
    int i;

    if (pid < 1)
        return 0;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].pid == pid) {
            return jobs[i].jid;
        }
    return 0;
}

/* listjobs - Print the job list */
void listjobs(struct job_t* jobs)
{
    int i;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid != 0) {
            printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
            switch (jobs[i].state) {
            case BG:
```

```
                printf("Running ");
                break;
            case FG:
                printf("Foreground ");
                break;
            case ST:
                printf("Stopped ");
                break;
            default:
                printf("listjobs: Internal error: job[%d].state=%d ",
                    i, jobs[i].state);
            }
            printf("%s", jobs[i].cmdline);
        }
    }
}
/******************************
 * end job list helper routines
 ******************************/


/************************
 * Other helper routines
 ************************/

/*
```

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

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

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

```
    exit(1);
}

/*
 * Signal - wrapper for the sigaction function
 */
handler_t* Signal(int signum, handler_t* handler)
{
    struct sigaction action, old_action;

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

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

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

```
        exit(1);
    }
```

第 4 章 TinyShell 测试

总分 15 分

4.1 测试方法

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

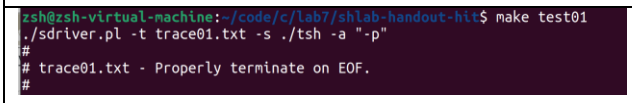
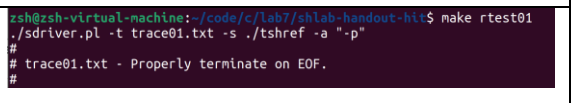
（2)测试文件 trace11.txt, trace12.txt 和 trace13.txt 中的/bin/ps 命令，每次运行的输出都会不同，但每个 mysplit 进程的运行状态应该相同。

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

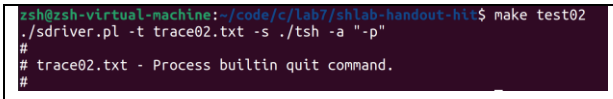
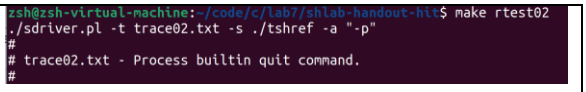
填写以下各个测试用例的测试结果，每个测试用例 1 分。

4.3.1 测试用例 trace01.txt

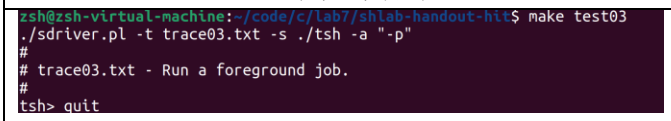
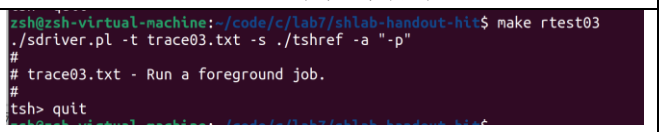
<table>
<tr><td>tsh 测试结果</td><td>tshref 测试结果</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>测试结论</td><td colspan="2">相同</td></tr>
</table>

4.3.2 测试用例 trace02.txt

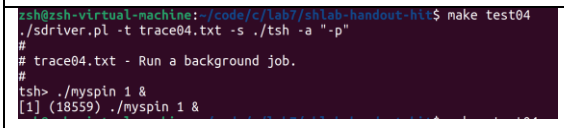
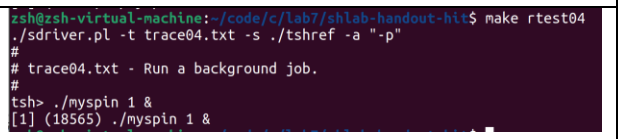
tsh 测试结果	tshref 测试结果

	
测试结论	相同

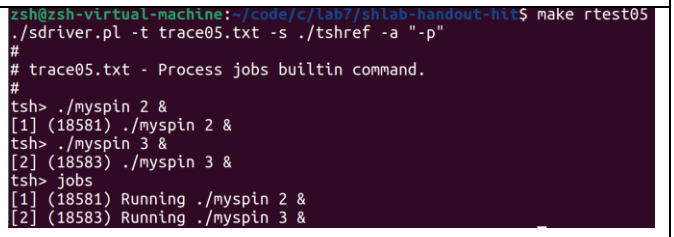
4.3.3 测试用例 trace03.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

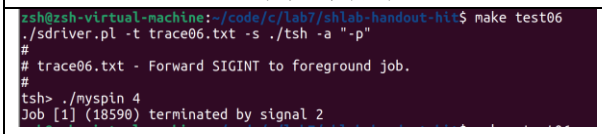
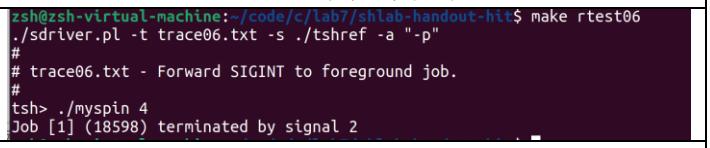
4.3.4 测试用例 trace04.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

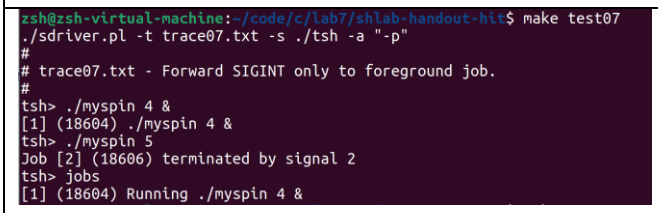
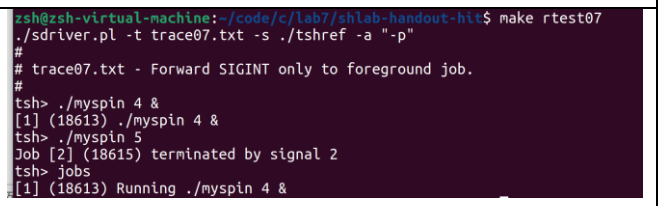
4.3.5 测试用例 trace05.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

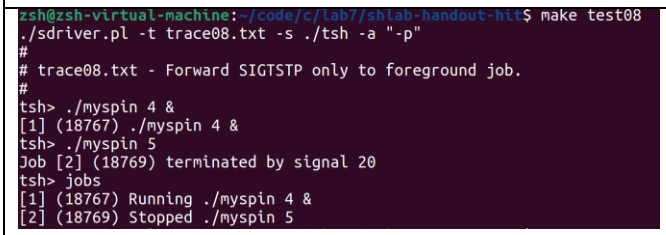
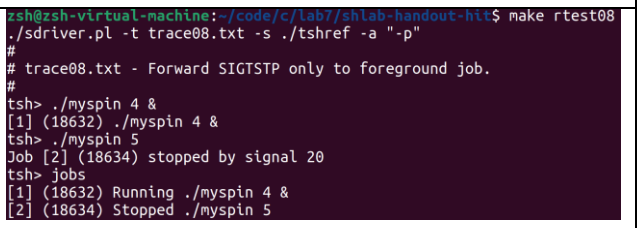
4.3.6 测试用例 trace06.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.7 测试用例 trace07.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.8 测试用例 trace08.txt

tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.9 测试用例 trace09.txt

tsh 测试结果	tshref 测试结果

<table>
<tr><td>

```
zsh@zsh-virtual-machine:~/code/c/lab7/shlab-handout-hit$ make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (18776) ./myspin 4 &
tsh> ./myspin 5
Job [2] (18778) terminated by signal 20
tsh> jobs
[1] (18776) Running ./myspin 4 &
[2] (18778) Stopped ./myspin 5
tsh> bg %2
[2] (18778) ./myspin 5
tsh> jobs
[1] (18776) Running ./myspin 4 &
[2] (18778) Running ./myspin 5
```

</td><td>

```
zsh@zsh-virtual-machine:~/code/c/lab7/shlab-handout-hit$ make rtest09
./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (18787) ./myspin 4 &
tsh> ./myspin 5
Job [2] (18789) stopped by signal 20
tsh> jobs
[1] (18787) Running ./myspin 4 &
[2] (18789) Stopped ./myspin 5
tsh> bg %2
[2] (18789) ./myspin 5
tsh> jobs
[1] (18787) Running ./myspin 4 &
[2] (18789) Running ./myspin 5
```

</td></tr>
<tr><td>测试结论</td><td>相同</td></tr>
</table>

4.3.10 测试用例 trace10.txt

<table>
<tr><td>tsh 测试结果</td><td>tshref 测试结果</td></tr>
<tr><td>

```
zsh@zsh-virtual-machine:~/code/c/lab7/shlab-handout-hit$ make test10
./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (18798) ./myspin 4 &
tsh> fg %1
Job [1] (18798) terminated by signal 20
tsh> jobs
[1] (18798) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

</td><td>

```
zsh@zsh-virtual-machine:~/code/c/lab7/shlab-handout-hit$ make rtest1
./sdriver.pl -t trace10.txt -s ./tshref -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (18808) ./myspin 4 &
tsh> fg %1
Job [1] (18808) stopped by signal 20
tsh> jobs
[1] (18808) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

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

4.3.11 测试用例 trace11.txt

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

<table>
<tr><td>tsh 测试结果</td><td>tshref 测试结果</td></tr>
<tr><td>

```
./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] (18830) terminated by signal 2
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   7712 tty2     Sslk   0:00 /usr/libexec/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
   7714 tty2     Slt    1:34 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -nolisten tcp -background none -noreset -keeptty -novtswitch -ver
bose 3
   7744 tty2     Sl+    0:00 /usr/libexec/gnome-session-binary --systemd --session=ubuntu
  17915 pts/0    Ss     0:00 bash
  17945 pts/0    S      0:00 ./tsh -p
  18085 pts/0    S      0:00 ./tsh -p
  18128 pts/0    S      0:00 ./tsh -p
  18214 pts/0    S      0:00 ./tsh -p
  18219 pts/0    S      0:00 ./tsh -p
  18361 pts/0    S      0:00 ./tsh -p
  18376 pts/0    S      0:00 ./tsh -p
  18409 pts/0    S      0:00 ./tsh -p
  18424 pts/0    S      0:00 ./tsh -p
  18438 pts/0    S      0:00 ./tsh -p
  18621 pts/0    S      0:00 ./tsh -p
  18625 pts/0    T      0:00 ./mysplit 5
  18639 pts/0    S      0:00 ./tsh -p
  18641 pts/0    T      0:00 ./mysplit 5
  18825 pts/0    S+     0:00 make test11
  18826 pts/0    S+     0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
  18827 pts/0    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh -a -p
  18828 pts/0    S+     0:00 ./tsh -p
  18833 pts/0    R      0:00 /bin/ps a
```

</td><td>

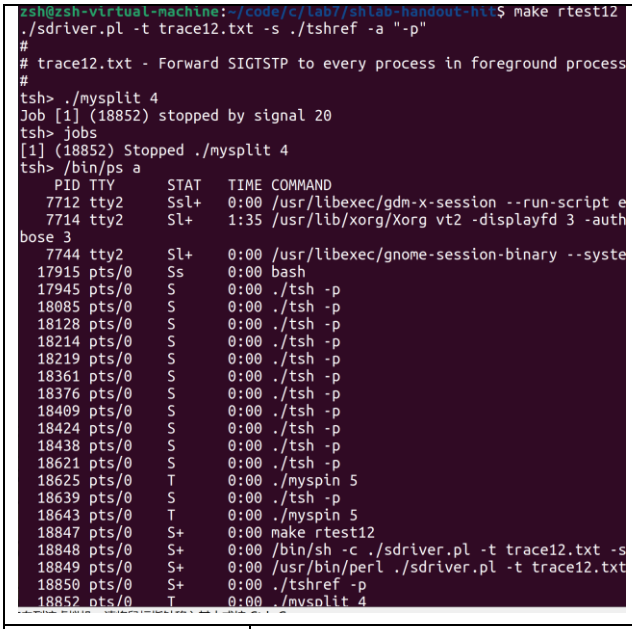
```
./sdriver.pl -t trace11.txt -s ./tshref -a "-p"
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (18821) terminated by signal 2
tsh> /bin/ps a
    PID TTY      STAT   TIME COMMAND
   7712 tty2     Sslk   0:00 /usr/libexec/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
   7714 tty2     Slt    1:34 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -nolisten tcp -background none -noreset -keeptty -novtswitch -ver
bose 3
   7744 tty2     Sl+    0:00 /usr/libexec/gnome-session-binary --systemd --session=ubuntu
  17915 pts/0    Ss     0:00 bash
  17945 pts/0    S      0:00 ./tsh -p
  18085 pts/0    S      0:00 ./tsh -p
  18128 pts/0    S      0:00 ./tsh -p
  18214 pts/0    S      0:00 ./tsh -p
  18219 pts/0    S      0:00 ./tsh -p
  18361 pts/0    S      0:00 ./tsh -p
  18376 pts/0    S      0:00 ./tsh -p
  18409 pts/0    S      0:00 ./tsh -p
  18424 pts/0    S      0:00 ./tsh -p
  18438 pts/0    S      0:00 ./tsh -p
  18621 pts/0    S      0:00 ./tsh -p
  18625 pts/0    T      0:00 ./mysplit 5
  18639 pts/0    S      0:00 ./tsh -p
  18641 pts/0    T      0:00 ./mysplit 5
  18816 pts/0    S+     0:00 make rtest11
  18817 pts/0    S+     0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tshref -a "-p"
  18818 pts/0    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tshref -a -p
  18819 pts/0    S+     0:00 ./tshref -p
  18824 pts/0    R      0:00 /bin/ps a
```

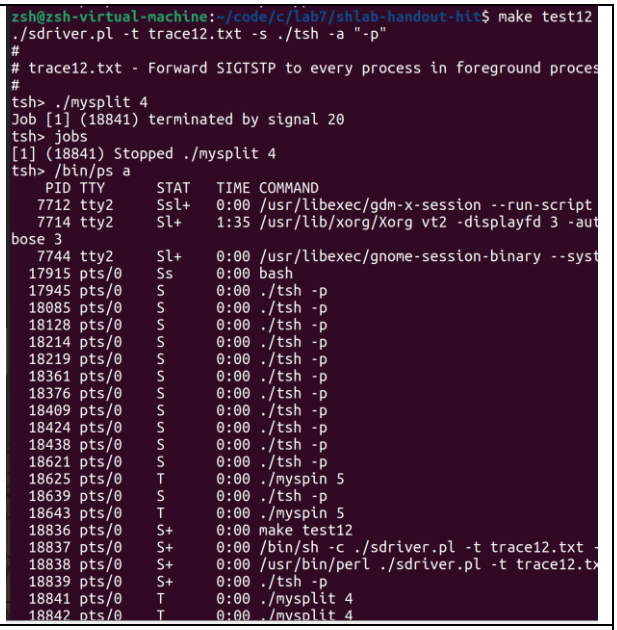
</td></tr>
<tr><td>测试结论</td><td>相同</td></tr>
</table>

4.3.12 测试用例 trace12.txt

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

<table>
<tr><td>tsh 测试结果</td><td>tshref 测试结果</td></tr>
</table>

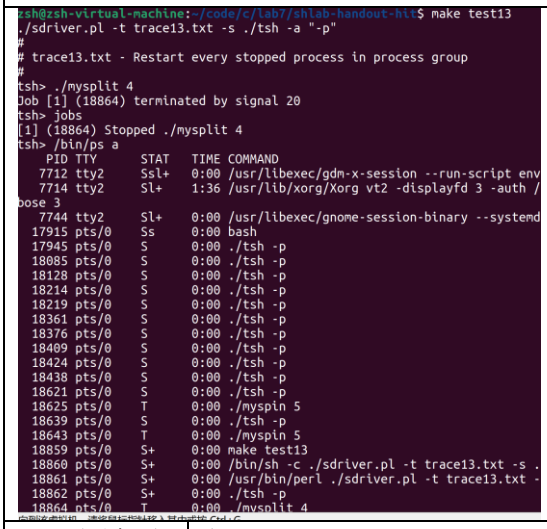
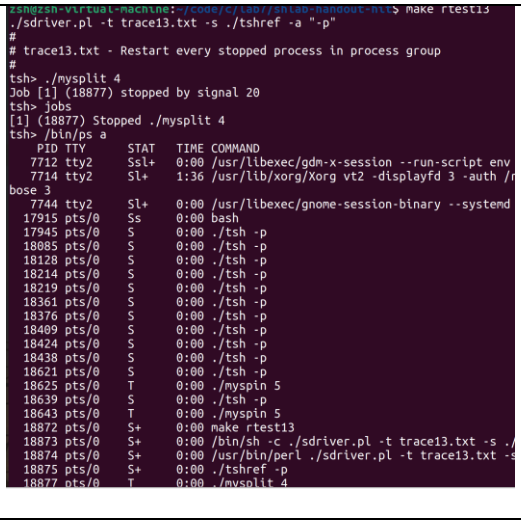




测试结论	相同

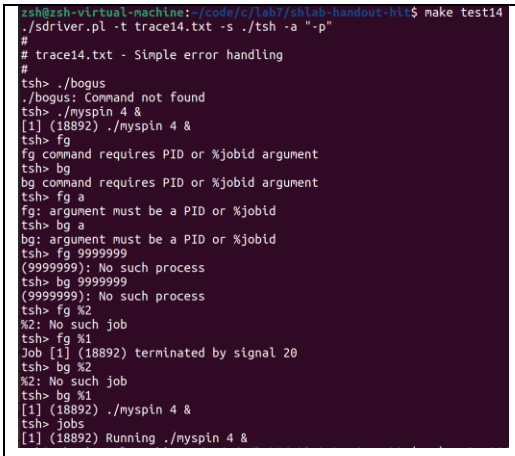
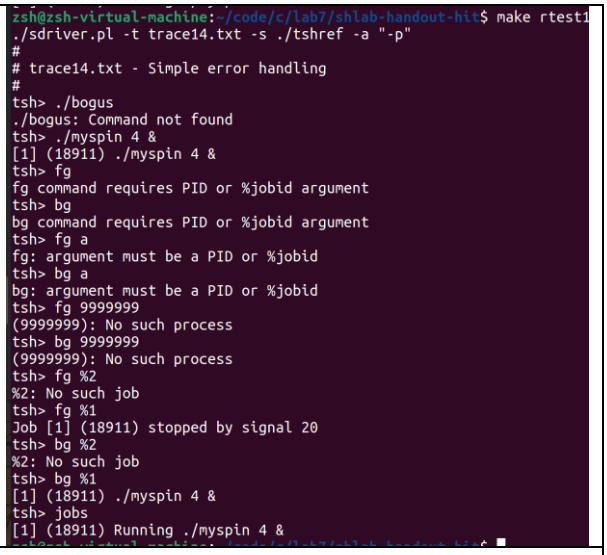
4.3.13 测试用例 trace13.txt

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。

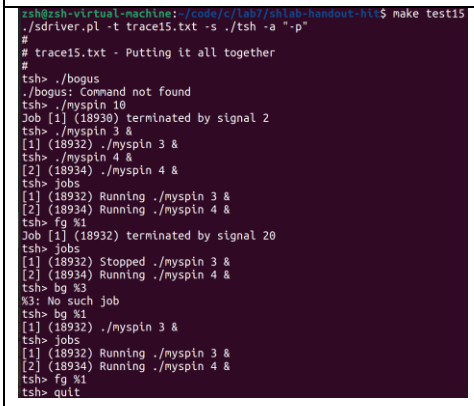
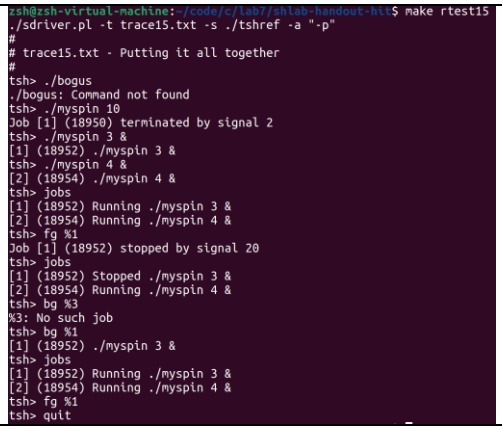
tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.14 测试用例 trace14.txt

tsh 测试结果	tshref 测试结果

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>测试结论</td><td>相同</td></tr>
</table>

4.3.15 测试用例 trace15.txt

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

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

对于 shell 的一些基本指令和 shell 的基本工作原理有了更加深入的了解，学会了调用一些 shell 指令的调用。

5.2 请给出对本次实验内容的建议

建议可以多一些教学。

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