

**实验报告**

**实 验（七）**

题 目 TinyShell

微壳

专 业 计算机类

学　　 号 1190200523

班　　 级 1903002

学 生 石翔宇

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2021.6.4

**计算机科学与技术学院**

**目 录**

[第1章 实验基本信息 - 4 -](#_Toc73657916)

[1.1 实验目的 - 4 -](#_Toc73657917)

[1.2 实验环境与工具 - 4 -](#_Toc73657918)

[1.2.1 硬件环境 - 4 -](#_Toc73657919)

[1.2.2 软件环境 - 4 -](#_Toc73657920)

[1.2.3 开发工具 - 4 -](#_Toc73657921)

[1.3 实验预习 - 4 -](#_Toc73657922)

[第2章 实验预习 - 6 -](#_Toc73657923)

[2.1 进程的概念、创建和回收方法（5分） - 6 -](#_Toc73657924)

[2.2信号的机制、种类（5分） - 6 -](#_Toc73657925)

[2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5分） - 7 -](#_Toc73657926)

[2.4 什么是shell，功能和处理流程（5分） - 8 -](#_Toc73657927)

[第3章 TinyShell的设计与实现 - 10 -](#_Toc73657928)

[3.1.1 void eval(char \*cmdline)函数（10分） - 10 -](#_Toc73657929)

[3.1.2 int builtin\_cmd(char \*\*argv)函数（5分） - 10 -](#_Toc73657930)

[3.1.3 void do\_bgfg(char \*\*argv) 函数（5分） - 11 -](#_Toc73657931)

[3.1.4 void waitfg(pid\_t pid) 函数（5分） - 11 -](#_Toc73657932)

[3.1.5 void sigchld\_handler(int sig) 函数（10分） - 11 -](#_Toc73657933)

[第4章 TinyShell测试 - 26 -](#_Toc73657934)

[4.1 测试方法 - 26 -](#_Toc73657935)

[4.2 测试结果评价 - 26 -](#_Toc73657936)

[4.3 自测试结果 - 26 -](#_Toc73657937)

[4.3.1测试用例trace01.txt - 26 -](#_Toc73657938)

[4.3.2测试用例trace02.txt - 27 -](#_Toc73657939)

[4.3.3测试用例trace03.txt - 27 -](#_Toc73657940)

[4.3.4测试用例trace04.txt - 27 -](#_Toc73657941)

[4.3.5测试用例trace05.txt - 27 -](#_Toc73657942)

[4.3.6测试用例trace06.txt - 28 -](#_Toc73657943)

[4.3.7测试用例trace07.txt - 28 -](#_Toc73657944)

[4.3.8测试用例trace08.txt - 28 -](#_Toc73657945)

[4.3.9测试用例trace09.txt - 28 -](#_Toc73657946)

[4.3.10测试用例trace10.txt - 29 -](#_Toc73657947)

[4.3.11测试用例trace11.txt - 29 -](#_Toc73657948)

[4.3.12测试用例trace12.txt - 29 -](#_Toc73657949)

[4.3.13测试用例trace13.txt - 30 -](#_Toc73657950)

[4.3.14测试用例trace14.txt - 30 -](#_Toc73657951)

[4.3.15测试用例trace15.txt - 31 -](#_Toc73657952)

[第5章 评测得分 - 32 -](#_Toc73657953)

[第6章 总结 - 33 -](#_Toc73657954)

[5.1 请总结本次实验的收获 - 33 -](#_Toc73657955)

[5.2 请给出对本次实验内容的建议 - 33 -](#_Toc73657956)

[参考文献 - 34 -](#_Toc73657957)

# 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

* Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
* 16GB RAM
* 1TB HDD + 512G SSD

### 1.2.2 软件环境

* Windows 10 21H1
* Ubuntu 20.04 LTS

### 1.2.3 开发工具

* VSCode，CodeBlocks，gcc+gdb

## 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）

概念：

进程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。程序是指令、数据及其组织形式的描述，进程是程序的实体。

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。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。

回收方法：

父进程通过 wait 函数回收子进程，wait函数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）

机制：

信号是用来通知进程发生了异步事件，是在软件层次上是对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是进程间通信机制中唯一的异步通信机制，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。进程之间可以互相通过系统调用 kill 发送软中断信号，内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。

种类：



## 2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5分）

发送方法：

1. 用/bin/kill程序可以向另外的进程或进程组发送任意的信号。
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。
3. 还可以调用系统函数来发送信号，主要有kill()，raise()，alarm()，pause()。

阻塞方法：

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）

显式阻塞机制：可以使用sigprocmask函数和它的辅助函数明确地阻塞和解除阻塞选定的信号。

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)

## 2.4 什么是shell，功能和处理流程（5分）

概念：

Shell 是一种命令行解释器，其读取用户输入的字符串命令，解释并且执行命令。 它是一种特殊的应用程序，介于系统调用/库与应用程序之间，其提供了运行其他程序的的接口。它可以是交互式的，即读取用户输入的字符串;也可以是非交互式的，即读取脚本文件并解释执行，直至文件结束。 无论是在类 UNIX，Linux 系统，还是 Windows，有很多不同种类的 Shell: 如类 UNIX，Linux 系统上的 Bash，Zsh 等; Windows 系统上的 cmd，PowerShell 等。

功能：

Shell为用户提供了启动程序、管理文件系统中的文件和运行在Linux系统上的进程。

处理流程：

1. Shell首先从命令行中找出特殊字符（元字符），在将元字符翻译成间隔符号。
2. 程序块tokens被处理，检查看他们是否是shell中所引用到的关键字。
3. 当程序块tokens被确定以后，shell根据aliases文件中的列表来检查命令的第一个单词。如果这个单词出现在aliases表中，执行替换操作并且处理过程回到第一步重新分割程序块tokens。
4. Shell对~符号进行替换。
5. Shell对所有前面带有$符号的变量进行替换。
6. Shell将命令行中的内嵌命令表达式替换成命令；他们一般都采用$(command)标记法。
7. Shell计算采用$(expression)标记的算术表达式。
8. Shell将命令字符串重新划分为新的块tokens。
9. Shell执行通配符\* ? [ ]的替换。
10. Shell把所有处理的结果中用到的注释删除，并且按照下面的顺序实行命令的检查：
    1. 内建的命令
    2. shell函数（由用户自己定义的）
    3. 可执行的脚本文件（需要寻找文件和PATH路径）
11. 初始化所有的输入输出重定向。
12. 执行命令。

# 第3章 TinyShell的设计与实现

**总分45分**

**3.1 设计**

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

函数功能：

解析传入的命令行命令。

参 数：

char \*cmdline

处理流程：

1. 调用parseline函数进行命令参数的解析。
2. 判断命令是否为内置命令
   1. 若是内置命令则在builtin\_cmd处理。
   2. 若不是内置命令，则：
      1. 阻塞SIGCHLD, SIGINT和SIGTSTP信号。
      2. fork一个新的进程。
      3. 在父进程中将子进程添加到jobs里面，解除阻塞。若该命令是在后台运行的，则输出提示信息；否则调用waitfg函数等待命令完成。
      4. 在子进程中解除阻塞，获取一个新的组ID。调用execve函数执行命令。

要点分析：

1. 在流程第二步开始前判断命令是否存在，若不存在则返回。
2. 若不是内置命令，则在addjob前要阻塞SIGCHLD, SIGINT和SIGTSTP信号，防止还没有将新进程添加到jobs列表里面就收到上述信号，出现错误。

## 3.1.2 int builtin\_cmd(char \*\*argv)函数（5分）

函数功能：

处理内置命令。

参 数：

char \*\*argv

处理流程：

1. 定义char指针指向4个内置命令。
2. 对4个内置命令依次判断，若相等则执行相应的命令。
   1. 若是quit则调用exit直接结束。
   2. 若是fg或者bg则调用do\_bgfg。
   3. 若是jobs则调用listjobs。

要点分析：

1. 注意若输入的命令能够与4个内置命令匹配，则除了quit命令，其他命令执行完之后要return 1以告诉eval函数输入的命令是内置命令。

## 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：

1. 判断是否存在第一个参数，若不存在则输出提示信息并返回。
2. 判断第一个参数的第一位是数字还是‘%’：
   1. 若是数字，则说明输入的可能是PID，查找PID对应的JobID，若查找失败则输出提示信息并返回。
   2. 若是‘%’，则说明输入的可能是JobID，查找Job列表中是否有当前的JobID，若查找失败则输出提示信息并返回。
   3. 若上述都不是，则输出提示信息并返回。
3. 判断当前命令是bg还是fg：
   1. 若是bg，则调用kill函数向该进程发送SIGCONT信号，将job的状态设置为BG并输出提示信息。
   2. 若是fg，则调用kill函数向该进程发送SIGCONT信号，将job的状态设置为FG并调用waitfg等待该进程运行完毕。

要点分析：

1. 要考虑到输入参数的各种情况，若处理不完善则可能会发生不可预知的错误。

## 3.1.4 void waitfg(pid\_t pid) 函数（5分）

函数功能：

等待一个前台命令结束运行。

参 数：

pid\_t pid

处理流程：

1. 使用while循环，每次循环调用sleep函数休眠1秒，循环条件为前台运行的进程的pid等于pid。

要点分析：

1. 我们可以调用fgpid函数获取前台运行的进程的pid。

## 3.1.5 void sigchld\_handler(int sig) 函数（10分）

函数功能：

SIGCHILD的信号处理程序。

参 数：

int sig

处理流程：

1. 使用while循环，循环条件是有被终止或暂停的子进程，判断导致返回的进程的状态：
   1. 若进程正常结束了，则先阻塞所有信号，再在job列表中删除被终止的进程，最后将信号阻塞列表还原。
   2. 若进程被暂停了，则调用getjobpid函数找到进程在job列表中的jobID，将该job的状态设置为ST，输出提示信息。
   3. 若进程被异常终止了，则调用getjobpid函数找到进程在job列表中的jobID，输出提示信息。再阻塞所有信号，在job列表中删除被终止的进程，将信号阻塞列表还原。

要点分析：

1. 要根据进程的返回状态来分开处理，使用不同的处理方法。
2. 调用deletejob之前要阻塞所有信号的输入。

**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

 \*

 \* 1190200523 Xiangyu Shi

 \*/

#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 \*/

        if (sigprocmask(SIG\_UNBLOCK, &mask, NULL) < 0)

            unix\_error("sigprocmask error");

        /\* 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);

    if (sigprocmask(SIG\_UNBLOCK, &mask, NULL) < 0)

        unix\_error("sigprocmask error");

    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)

{

    char \*cmd\_quit = "quit";

    char \*cmd\_fg = "fg";

    char \*cmd\_bg = "bg";

    char \*cmd\_jobs = "jobs";

    /\* if the command is "quit", just exit. \*/

    if (!strcmp(argv[0], cmd\_quit)) {

        exit(0);

    }

    /\* if the command is "fg" or "bg", call do\_bgfg. \*/

    else if (!strcmp(argv[0], cmd\_fg) || !strcmp(argv[0], cmd\_bg)) {

        do\_bgfg(argv);

        return 1;

    }

    /\* if the command is "jobs", call listjobs to print job list. \*/

    else if (!strcmp(argv[0], cmd\_jobs)) {

        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)

{

    /\* sleep until process pid is no longer the foreground process \*/

    while (fgpid(jobs) == pid)

        sleep(1);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* 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)

{

    int status;

    sigset\_t mask\_all, prev\_all;

    pid\_t pid;

    if (sigfillset(&mask\_all) < 0)

        unix\_error("sigfillset error");

    while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0) {

        if (WIFEXITED(status)) {

            /\* killblock all signals. \*/

            if (sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all) < 0)

                unix\_error("sigprocmask error");

            /\* delete process from job list \*/

            deletejob(jobs, pid);

            /\* unblock signals. \*/

            if (sigprocmask(SIG\_SETMASK, &prev\_all, NULL) < 0)

                unix\_error("sigprocmask error");

        }

        else if (WIFSTOPPED(status)) {

            struct job\_t \*job = getjobpid(jobs, pid);

            /\* set status. \*/

            job->state = ST;

            printf("Job [%d] (%d) stopped by signal %d\n", job->jid, job->pid, WSTOPSIG(status));

        }

        else if (WIFSIGNALED(status))

        {

            struct job\_t \*job = getjobpid(jobs, pid);

            printf("Job [%d] (%d) terminated by signal %d\n", job->jid, job->pid, WTERMSIG(status));

            /\* killblock all signals. \*/

            if (sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all))

                unix\_error("sigprocmask error");

            /\* delete process from job list \*/

            deletejob(jobs, pid);

            /\* unblock signals. \*/

            if (sigprocmask(SIG\_SETMASK, &prev\_all, NULL))

                unix\_error("sigprocmask error");

        }

    }

}

/\*

 \* 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\_all, prev\_all;

    sigfillset(&mask\_all);

    pid\_t fg\_pid = fgpid(jobs);

    if (fg\_pid != 0) {

        /\* killblock all signals. \*/

        if (sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all))

            unix\_error("sigprocmask error");

        /\* send signal SIGINT to the foreground job. \*/

        if (kill(-fg\_pid, SIGINT) < 0)

            unix\_error("kill error");

        /\* unblock signals. \*/

        if (sigprocmask(SIG\_SETMASK, &prev\_all, NULL))

            unix\_error("sigprocmask error");

    }

}

/\*

 \* 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\_all, prev\_all;

    sigfillset(&mask\_all);

    pid\_t fg\_pid = fgpid(jobs);

    if (fg\_pid != 0) {

        /\* killblock all signals. \*/

        if (sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all))

            unix\_error("sigprocmask error");

        /\* send signal SIGTSTP to the foreground job. \*/

        if (kill(-fg\_pid, SIGTSTP) < 0)

            unix\_error("kill error");

        /\* unblock signals. \*/

        if (sigprocmask(SIG\_SETMASK, &prev\_all, NULL))

            unix\_error("sigprocmask error");

    }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* 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

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.2测试用例trace02.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.3测试用例trace03.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.4测试用例trace04.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.5测试用例trace05.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.6测试用例trace06.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.7测试用例trace07.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.8测试用例trace08.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.9测试用例trace09.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 测试结论 | 相同 | |

### 4.3.10测试用例trace10.txt

|  |  |  |
| --- | --- | --- |
| tsh测试结果 | | tshref测试结果 |
|  | |  |
| 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 |
|  | |  |
| 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 |
|  | |  |
| 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 |
|  | |  |
| 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 |
|  | |  |
| 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

* 对信号的发出、接收、阻塞等机制有了更深的理解。
* 更加深入地了解了shell的工作流程。

## 5.2 请给出对本次实验内容的建议

* 希望提供的代码能够少一些。

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.