

# 哈尔滨工业大学

# 实验报告

## 实验（五）

题 目 TinyShell

微壳

专 业 计算机科学与技术

学 号 2022113573

班 级 2203101

学 生 张宇杰

指导教师 史先俊

实验地点 格物 712

实验日期 2023/12/14

计算学部

# 目 录

第 1 章 实验基本信息 .....	- 4 -
1.1 实验目的 .....	- 4 -
1.2 实验环境与工具 .....	- 4 -
1.2.1 硬件环境 .....	- 4 -
1.2.2 软件环境 .....	- 4 -
1.2.3 开发工具 .....	- 4 -
1.3 实验预习 .....	- 4 -
第 2 章 实验预习 .....	- 5 -
2.1 进程的概念、创建和回收方法（5 分） .....	- 5 -
2.2 信号的机制、种类（5 分） .....	- 5 -
2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分） .....	- 7 -
2.4 什么是 SHELL，简述其功能和处理流程（5 分） .....	- 9 -
第 3 章 TINY SHELL 的设计与实现 .....	- 10 -
3.1 设计 .....	- 10 -
3.1.1 VOID EVAL(CHAR *CMDLINE) 函数（10 分） .....	- 10 -
3.1.2 INT BUILTIN_CMD(CHAR **ARGV) 函数（5 分） .....	- 11 -
3.1.3 VOID DO_BGFG(CHAR **ARGV) 函数（5 分） .....	- 11 -
3.1.4 VOID WAITFG(PID_T PID) 函数（5 分） .....	- 12 -
3.1.5 VOID SIGCHLD_HANDLER(INT SIG) 函数（10 分） .....	- 12 -
3.2 程序实现（TSH.C 的全部内容）（10 分） .....	- 13 -
第 4 章 TINY SHELL 测试 .....	- 27 -
4.1 测试方法 .....	- 27 -
4.2 测试结果评价 .....	- 27 -
4.3 自测试结果 .....	- 27 -
4.3.1 测试用例 trace01.txt 的输出截图（1 分） .....	- 27 -
4.3.2 测试用例 trace02.txt 的输出截图（1 分） .....	- 27 -
4.3.3 测试用例 trace03.txt 的输出截图（1 分） .....	- 28 -
4.3.4 测试用例 trace04.txt 的输出截图（1 分） .....	- 28 -
4.3.5 测试用例 trace05.txt 的输出截图（1 分） .....	- 28 -
4.3.6 测试用例 trace06.txt 的输出截图（1 分） .....	- 29 -
4.3.7 测试用例 trace07.txt 的输出截图（1 分） .....	- 29 -
4.3.8 测试用例 trace08.txt 的输出截图（1 分） .....	- 29 -
4.3.9 测试用例 trace09.txt 的输出截图（1 分） .....	- 30 -
4.3.10 测试用例 trace10.txt 的输出截图（1 分） .....	- 30 -
4.3.11 测试用例 trace11.txt 的输出截图（1 分） .....	- 30 -

4.3.12 测试用例 <i>trace12.txt</i> 的输出截图 (1 分)	- 31 -
4.3.13 测试用例 <i>trace13.txt</i> 的输出截图 (1 分)	- 31 -
4.3.14 测试用例 <i>trace14.txt</i> 的输出截图 (1 分)	- 33 -
4.3.15 测试用例 <i>trace15.txt</i> 的输出截图 (1 分)	- 33 -
4.4 自测试评分	- 34 -
<b>第 4 章 总结</b>	<b>- 35 -</b>
4.1 请总结本次实验的收获	- 35 -
4.2 请给出对本次实验内容的建议	- 35 -
<b>参考文献</b>	<b>- 36 -</b>

## 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

Windows7/10 64 位以上; VirtualBox/Vmware 11 以上; Ubuntu 16.04 LTS 64 位/优麒麟 64 位 以上

#### 1.2.3 开发工具

Visual Studio 2010 64 位以上; CodeBlocks 64 位; vi/vim/gedit+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）

#### 1. 进程的概念

进程是一个执行中程序的实例。

#### 2. 进程的创建

使用 `fork` 函数来为一个已经存在的进程创建一个新进程，并称其为原进程的子进程，原进程称为父进程。

使用 `execve` 函数在一个进程中直接创建另一个进程，新进程会直接覆盖原有进程的代码段等内容。通常需要在父进程中先调用 `fork` 函数创建一个子进程，然后在子进程中使用 `execve` 函数执行其他程序。

使用 `system` 函数执行一行外部命令，变相的创建了一个新进程。实际上是对 `fork`，`execve` 与 `waitpid` 的封装。

使用 `popen` 函数可以执行一个外部命令并建立与该命令的标准输入或输出之间的管道连接。这是一个标准 C 库函数。

#### 3. 进程的回收方法

应该由父进程进行回收。在父进程收到子进程的退出信号（`SIGCHLD`）后，通过调用 `wait` 或 `waitpid`，通知内核回收僵死子进程，从系统中删除掉它的所有痕迹。

如果父进程不进行回收，当父进程终止时，由 `init` 进程进行回收。

### 2.2 信号的机制、种类（5 分）

#### 1. 信号的机制

信号就是一条小消息，它通知进程系统中发生了一个某种类型的事件。信号是内核处理程序在运行时发生错误的解决方式，还是终端管理进程的方式，并且还是一种进程间通信机制。

### 1.1. 信号的发送

内核通过更新目的进程上下文中的某个状态，发送一个信号给目的进程。

发送信号可以是如下原因之一：

内核检测到一个系统事件，如 SIGFPE，SIGCHLD，SIGINT 等；

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。

### 1.2. 信号的接收与处理

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，就接收了信号。

可能的反应方式有：

忽略；

终止进程（可选的核心转储）；

执行相应的信号处理程序。

### 1.3. 待处理信号与阻塞信号

一个发出而没有被接收的信号叫做待处理信号。

任何时刻，一种类型（1-30）至多只有一个待处理信号

如果一个进程有一个类型为 k 的待处理信号，那么任何接下来发送到这个进程的类型为 k 的信号都会被丢弃。

一个进程可以选择阻塞接收某种信号。

阻塞的信号仍可以被发送，但不会被接收，直到进程取消对该信号的阻塞。

## 2. 信号的种类

通过 kill -l 查看所有定义的信号

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

常见信号的含义：

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存 <sup>①</sup>	跟踪陷阱
6	SIGABRT	终止并转储内存 <sup>①</sup>	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存 <sup>①</sup>	浮点异常
9	SIGKILL	终止 <sup>②</sup>	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存 <sup>①</sup>	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT <sup>②</sup>	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

## 2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分）

### 1. 信号的发送方法

/bin/kill 程序可以向另外的进程或进程组发送任意的信号。

如，/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)

从键盘输入 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。

使用 **kill** 函数，于头文件<signal.h>中，原型为 `int kill(pid_t pid, int sig);`

**pid>0** 将此信号发送给进程 ID 为 **pid** 的进程；

**pid=0** 将此信号发送给当前进程所在进程组的所有进程；

**pid<0** 将此信号发送给进程组 ID 为 **-pid** 的所有进程；

**pid=-1** 将此信号发送给所有有权接受该信号的进程。

返回 0 时成功执行。

使用 **raise** 函数，于头文件<signal.h>中，原型为 `int raise(int sig);`

给自己发送 **sig** 信号。

返回 0 时成功执行。

使用 **alarm** 函数，于头文件<unistd.h>中，原型为 `unsigned int alarm(unsigned int seconds);`

设置定时器。在指定 **seconds** 后，内核会给当前进程发送 **SIGALRM** 信号。

返回 0 或剩余的秒数。

## 2. 信号的阻塞方法

隐式阻塞：

内核默认阻塞与当前正在处理信号类型相同的待处理信号。

显示阻塞与解除阻塞：

使用 **sigprocmask** 函数设定对信号屏蔽集内的信号的处理方式(阻塞或不阻塞)。

`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`

**how**: 用于指定信号修改的方式，可能选择有三种：

**SIG\_BLOCK**: 将 **set** 所指向的信号集中包含的信号加到当前的信号掩码中。

**SIG\_UNBLOCK**: 将 **set** 所指向的信号集中包含的信号从当前的信号掩码中删除。

**SIG\_SETMASK**: 将 **set** 的值设定为新的进程信号掩码。

**set**: 为指向新设的信号集的指针，若仅想读取现在的屏蔽值，将其置为 **NULL**。



**oldset:** 也是指向存放原来的信号集的指针。可用来检测信号掩码中存在什么信号。

成功执行时，返回 0。

### 3. 信号处理程序的设置

使用 `signal` 函数进行设置，于头文件 `<signal.h>` 中

原型为：`sighandler_t signal(int signum, sighandler_t handler);`

**signum:** 指明了所要处理的信号类型，可以取除了 `SIGKILL` 和 `SIGSTOP` 外的任何一种信号。

**handler:** 描述了与信号关联的动作，它可以取以下三种值：

`SIG_IGN`: 忽略该信号；

`SIG_DFL`: 恢复对信号的系统默认处理；

`sighandler_t` 类型的函数指针[即：`void (*)(int)`]：当接收到一个类型为 `sig` 的信号时，就执行 `handler` 所指定的函数。

## 2.4 什么是 shell，简述其功能和处理流程（5 分）

`shell` 是一个交互型应用级程序，可代表用户运行其他程序。

`shell` 最重要的功能是命令解释，从这种意义上说，`shell` 是一个命令解释器。  
Linux 系统上的所有可执行文件都可以作为 `shell` 命令来执行。

当用户提交了一个命令后，`shell` 首先判断它是否为内置命令，如果是就通过 `shell` 内部的解释器将其解释为系统功能调用并转交给内核执行；若是外部命令或应用程序就试图在硬盘中查找该命令并将其调入内存，再将其解释为系统功能调用并转交给内核执行。在查找该命令时分为两种情况：（1）用户给出了命令的路径，`shell` 就沿着用户给出的路径进行查找，若找到则调入内存，若没找到则输出提示信息；（2）用户没有给出命令的路径，`shell` 就在环境变量 `PATH` 所制定的路径中依次进行查找，若找到则调入内存，若没找到则输出提示信息。

`shell` 还具有如下的一些功能：

通配符，命令补全、别名机制、命令历史，重定向，管道，命令替换，Shell 编程语言。

## 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:

解析用户输入的命令。如果是内置命令(fg, bg, jobs, quit)则立即执行, 否则, fork 一个子进程并尝试在子进程中执行用户输入的作业。如果该作业是前台作业, 阻塞输入直到该作业不在是前台作业。

参 数:

char \*cmdline: 用户输入的命令

处理流程:

- 1、进入函数后, 使用 `parseline` 函数对命令行进行解析, 得到 `argv` 数组, 并得到是否需要在后台运行作业的标记 `bg`;
- 2、使用 `builtin_cmd` 函数判断命令是否是内置命令, 如果是, 该命令立即在 `builtin_cmd` 函数内部进行执行, 并直接返回自 `eval` 函数。如果不是, 进入非内置命令处理分枝 (即 3 以后);
- 3、设置阻塞信号, 将 `SIGCHLD`, `SIGINT` 与 `SIGTSTP` 加入阻塞集合中, 防止父进程在创建子进程时被上述信号打断, 防止 `addjob` 函数错误地把 (不存在的) 子进程添加到作业列表中;
- 4、fork 完子进程后, 在子进程中, 解除子进程对上述信号的阻塞 (因为子进程继承了父进程的 `PCB`), 使用 `setpgid` 为子进程分配新的进程组号, 并在子进程中使用 `execve` 函数执行用户的自定义命令;
- 5、fork 完子进程后, 在父进程中, 通过 `addjob` 函数添加子进程作业至作业表中, 解除父进程对上述信号的阻塞。如果该作业是前台作业, 调用 `waitfg` 函数, 等待该作业不在是前台作业, 否则, 输出该后台作业的作业信息。

要点分析:

- 1、在 `fork` 后、`execve` 前, 子进程调用函数 `setpgid(0, 0)` 将自己放到一个新的进程组中, 确保前台进程组中只有一个进程 `tsh`。这是因为, `ctrl-c` (`ctrl-z`) 会给所有前台进程组中的进程发送 `SIGINT` (`SIGTSTP`) 信号, 包括 `tsh` 和 `tsh` 创建的进程 (如果没有调用 `setpgid` 的话), 这样不符合预期;
- 2、在 `eval` 中, 父进程必须在用 `fork` 创建子进程前, 使用 `sigprocmask` 阻塞

SIGCHLD 信号。父进程创建完成子进程并用 `addjob` 记录后，用 `sigprocmask` 解除阻塞。子进程从父进程处继承了信号阻塞向量，子进程必须确保在执行新程序之前解除对 SIGCHLD 的阻塞。这样可以防止父进程调用 `addjob` 之前子进程就被信号处理程序回收。

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

函数功能：

判断输入的命令行参数 `argv` 是否是内置命令(`fg`, `bg`, `jobs`, `quit`)，如果是，立即调用相关函数来执行，返回 1；如果不是，返回 0。

参 数：

`char **argv`: 命令行参数。

处理流程：

根据 `argv[0]` 的内容对命令进行判断；  
如果是 `bg` 或 `fg`，调用 `do_bgfg` 函数，返回 1；  
如果是 `jobs`，调用 `listjobs` 函数，返回 1；  
如果是 `quit`，调用 `exit` 函数直接退出；  
否则，返回 0；

要点分析：

注意 `quit` 命令的表现是直接退出，而不是给自己发送 SIGQUIT 信号

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

函数功能：

执行内置命令 `bg` 与 `fg`。

参 数：

`char **argv`: 命令行参数。

处理流程：

- 1、判断 `fg` 或 `bg` 后是否有带参数。若否，则忽略该命令，弹出提示信息并返回；
- 2、根据第二个参数 `argv[1]` 的内容进行分支处理；
- 3、如果是纯数字，说明是指 PID，用 `jobp` 指向该 PID 对应的作业，尔后执行 6。如果不存在对应作业，弹出提示信息并返回；
- 4、如果第一个字符是 '%'（百分号），说明是指 JID，用 `jobp` 指向该 JID 对应的作业，尔后执行 6。如果不存在对应作业，弹出提示信息并返回；
- 5、否则，说明第二个参数错误，弹出提示信息并返回；
- 6、如果是 `bg` 命令，使用 `kill` 发送 SIGCONT 信号给对应进程组，将该作用

- 的状态设为 BG（后台作业），输出该作业的信息；
- 7、如果是 fg 命令，使用 kill 发送 SIGCONT 信号给对应进程组，将该作业的状态设为 FG（前台作业），调用 waitfg 等待前台作业执行；
  - 8、如果都不是，弹出错误信息。

要点分析：

- 1、对于错误的命令，需要打印相应的错误信息进行提示；
- 2、调用 getjobpid 与 getjobjid 时，需要判断对应作业是否存在（即返回了 NULL 与否）；
- 3、调用 kill 时，注意是给对应进程组发信号（即参数为 -pid）。

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

函数功能：

阻塞直到 pid 对应的作业不再是前台作业

参 数：

pid\_t pid: 前台进程的 pid

处理流程：

根据 pid 通过 getjobpid 使 jobp 指向 pid 对应作业。当 jobp 指向作业的状态为前台作业时，重复执行 sleep(1)。

要点分析：

建议在 waitfg 函数中，在 sleep 函数附近使用 busy loop，例如：

```
while (xxxxx) sleep(1);
```

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

函数功能：

处理由内核发来的 SIGCHLD 信号。根据由 waitpid 返回的子进程的状态进行相应处理。

参 数：

int sig: 信号，可断言其值为 SIGCHLD。

处理流程：

- 1、保存 errno，阻塞所有信号
- 2、执行循环 while ((child\_pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)。waitpid 会返回被终止或停止的子进程的 pid，WNOHANG | WUNTRACED 表示立即返回，不进行等待；
- 3、调用 getjobpid 使 jobp 指向 child\_pid 对应作业。根据返回的 status 值进行分支处理；

- 4、如果子进程正常退出，即 `WIFEXITED(status)` 为 1，调用 `deletejob` 删除 `child_pid` 对应作业；
- 5、如果子进程被停止（挂起），即 `WIFSTOPPED(status)` 为 1，打印作业被停止的信息，并将 `jobp` 指向作业的状态修改为 ST；
- 6、如果子进程被终止，即 `WIFSIGNALED(status)` 为 1，打印作业被终止的信息，调用 `deletejob` 删除 `child_pid` 对应作业；
- 7、否则，发生内部错误；
- 8、还原 `errno`，还原对信号的阻塞。

要点分析：

- 1、在处理 `SIGCHLD` 的处理程序中，使用 `while(waitpid)` 的结构，尽可能的对子进程进行处理，避免某些子进程被遗漏（如果使用 `if(waitpid)` 结构的话）；
- 2、仅在 `SIGCHLD` 处理程序中回收进程，可以使程序逻辑清晰、简单；
- 3、应理解宏 `WIFEXITED`，`WIFSTOPPED`，`WIFSIGNALED` 分别的含义；
- 4、根据信号处理程序 G2 原则（G2: 保存和恢复 `errno`），在函数入口需保存 `errno`，并在函数出口还原 `errno`；
- 5、由于对全局数据结构 `jobs` 进行了操作，根据信号处理程序 G3 原则（G3: 阻塞所有信号，保护对共享全局数据结构的访问），在函数入口需阻塞所有信号，在出口处进行还原。

### 3.2 程序实现（tsh.c 的全部内容）（10 分）

```

1  /*
2   * tsh - A tiny shell program with job control
3   *
4   * <张宇杰 2022113573>
5   */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <string.h>
10 #include <ctype.h>
11 #include <signal.h>
12 #include <sys/types.h>
13 #include <sys/wait.h>
14 #include <errno.h>
15
16 /* Misc manifest constants */
17 #define MAXLINE 1024 /* max line size */
18 #define MAXARGS 128 /* max args on a command line */
19 #define MAXJOBS 16 /* max jobs at any point in time */
20 #define MAXJID 1 << 16 /* max job ID */
21
22 /* Job states */
23 #define UNDEF 0 /* undefined */
24 #define FG 1 /* running in foreground */
25 #define BG 2 /* running in background */
26 #define ST 3 /* stopped */
27
28 /*

```

```

29  * Jobs states: FG (foreground), BG (background), ST (stopped)
30  * Job state transitions and enabling actions:
31  *   FG -> ST : ctrl-z
32  *   ST -> FG : fg command
33  *   ST -> BG : bg command
34  *   BG -> FG : fg command
35  * At most 1 job can be in the FG state.
36  */
37
38  /* Global variables */
39  extern char **environ; /* defined in libc */
40  char prompt[] = "tsh> "; /* command line prompt (DO NOT CHANGE) */
41  int verbose = 0; /* if true, print additional output */
42  int nextjid = 1; /* next job ID to allocate */
43  char sbuf[MAXLINE]; /* for composing sprintf messages */
44
45  struct job_t
46  {
47      pid_t pid; /* job PID */
48      int jid; /* job ID [1, 2, ...] */
49      int state; /* UNDEF, BG, FG, or ST */
50      char cmdline[MAXLINE]; /* command line */
51  };
52  struct job_t jobs[MAXJOBS]; /* The job list */
53  /* End global variables */
54
55  /* Function prototypes */
56
57  /* Here are the functions that you will implement */
58  void eval(char *cmdline);
59  int builtin_cmd(char **argv);
60  void do_bgfg(char **argv);
61  void waitfg(pid_t pid);
62
63  void sigchld_handler(int sig);
64  void sigtstp_handler(int sig);
65  void sigint_handler(int sig);
66
67  /* Here are helper routines that we've provided for you */
68  int parseline(const char *cmdline, char **argv);
69  void sigquit_handler(int sig);
70
71  void clearjob(struct job_t *job);
72  void initjobs(struct job_t *jobs);
73  int maxjid(struct job_t *jobs);
74  int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
75  int deletejob(struct job_t *jobs, pid_t pid);
76  pid_t fgpid(struct job_t *jobs);
77  struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
78  struct job_t *getjobjid(struct job_t *jobs, int jid);
79  int pid2jid(pid_t pid);
80  void listjobs(struct job_t *jobs);
81
82  void usage(void);
83  void unix_error(char *msg);
84  void app_error(char *msg);
85  typedef void handler_t(int);
86  handler_t *Signal(int signum, handler_t *handler);
87
88  /*
89  * main - The shell's main routine

```

```
90  */
91  int main(int argc, char **argv)
92  {
93      int c;
94      char cmdline[MAXLINE];
95      int emit_prompt = 1; /* emit prompt (default) */
96
97      /* Redirect stderr to stdout (so that driver will get all output
98       * on the pipe connected to stdout) */
99      dup2(1, 2);
100
101      /* Parse the command line */
102      while ((c = getopt(argc, argv, "hvp")) != EOF)
103      {
104          switch (c)
105          {
106              case 'h': /* print help message */
107                  usage();
108                  break;
109              case 'v': /* emit additional diagnostic info */
110                  verbose = 1;
111                  break;
112              case 'p': /* don't print a prompt */
113                  emit_prompt = 0; /* handy for automatic testing */
114                  break;
115              default:
116                  usage();
117          }
118      }
119
120      /* Install the signal handlers */
121
122      /* These are the ones you will need to implement */
123      Signal(SIGINT, sigint_handler); /* ctrl-c */
124      Signal(SIGTSTP, sigtstp_handler); /* ctrl-z */
125      Signal(SIGCHLD, sigchld_handler); /* Terminated or stopped child */
126
127      /* This one provides a clean way to kill the shell */
128      Signal(SIGQUIT, sigquit_handler);
129
130      /* Initialize the job list */
131      initjobs(jobs);
132
133      /* Execute the shell's read/eval loop */
134      while (1)
135      {
136
137          /* Read command line */
138          if (emit_prompt)
139          {
140              printf("%s", prompt);
141              fflush(stdout);
142          }
143          if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
144              app_error("fgets error");
145          if (feof(stdin))
146          { /* End of file (ctrl-d) */
147              fflush(stdout);
148              exit(0);
149          }
150      }
```

```
151     /* Evaluate the command line */
152     eval(cmdline);
153     fflush(stdout);
154     fflush(stdout);
155 }
156
157     exit(0); /* control never reaches here */
158 }
159
160 /*
161  * eval - Evaluate the command line that the user has just typed in
162  *
163  * If the user has requested a built-in command (quit, jobs, bg or fg)
164  * then execute it immediately. Otherwise, fork a child process and
165  * run the job in the context of the child. If the job is running in
166  * the foreground, wait for it to terminate and then return. Note:
167  * each child process must have a unique process group ID so that our
168  * background children don't receive SIGINT (SIGTSTP) from the kernel
169  * when we type ctrl-c (ctrl-z) at the keyboard.
170  */
171 void eval(char *cmdline)
172 {
173     /* $begin handout */
174     char *argv[MAXARGS]; /* argv for execve() */
175     int bg;               /* should the job run in bg or fg? */
176     pid_t pid;            /* process id */
177     sigset_t mask;        /* signal mask */
178
179     /* Parse command line */
180     bg = parseline(cmdline, argv);
181     if (argv[0] == NULL)
182         return; /* ignore empty lines */
183
184     if (!builtin_cmd(argv))
185     {
186
187         /*
188          * This is a little tricky. Block SIGCHLD, SIGINT, and SIGTSTP
189          * signals until we can add the job to the job list. This
190          * eliminates some nasty races between adding a job to the job
191          * list and the arrival of SIGCHLD, SIGINT, and SIGTSTP signals.
192          */
193
194         if (sigemptyset(&mask) < 0)
195             unix_error("sigemptyset error");
196         if (sigaddset(&mask, SIGCHLD))
197             unix_error("sigaddset error");
198         if (sigaddset(&mask, SIGINT))
199             unix_error("sigaddset error");
200         if (sigaddset(&mask, SIGTSTP))
201             unix_error("sigaddset error");
202         if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
203             unix_error("sigprocmask error");
204
205         /* Create a child process */
206         if ((pid = fork()) < 0)
207             unix_error("fork error");
208
209         /*
210          * Child process
211          */
212     }
```



```

212
213     if (pid == 0)
214     {
215         /* Child unblocks signals */
216         sigprocmask(SIG_UNBLOCK, &mask, NULL);
217
218         /* Each new job must get a new process group ID
219          so that the kernel doesn't send ctrl-c and ctrl-z
220          signals to all of the shell's jobs */
221         if (setpgid(0, 0) < 0)
222             unix_error("setpgid error");
223
224         /* Now load and run the program in the new job */
225         if (execve(argv[0], argv, environ) < 0)
226         {
227             printf("%s: Command not found\n", argv[0]);
228             exit(0);
229         }
230     }
231
232     /*
233     * Parent process
234     */
235
236     /* Parent adds the job, and then unblocks signals so that
237      the signals handlers can run again */
238     addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
239     sigprocmask(SIG_UNBLOCK, &mask, NULL);
240
241     if (!bg)
242         waitfg(pid);
243     else
244         printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
245 }
246 /* $end handout */
247 return;
248 }
249
250 /*
251 * parseline - Parse the command line and build the argv array.
252 *
253 * Characters enclosed in single quotes are treated as a single
254 * argument. Return true if the user has requested a BG job, false if
255 * the user has requested a FG job.
256 */
257 int parseline(const char *cmdline, char **argv)
258 {
259     static char array[MAXLINE]; /* holds local copy of command line */
260     char *buf = array;          /* ptr that traverses command line */
261     char *delim;                 /* points to first space delimiter */
262     int argc;                    /* number of args */
263     int bg;                      /* background job? */
264
265     strcpy(buf, cmdline);
266     buf[strlen(buf) - 1] = ' '; /* replace trailing '\n' with space */
267     while (*buf && (*buf == ' ')) /* ignore leading spaces */
268         buf++;
269
270     /* Build the argv list */
271     argc = 0;
272     if (*buf == '\\')

```

```
273     {
274         buf++;
275         delim = strchr(buf, '\\');
276     }
277     else
278     {
279         delim = strchr(buf, ' ');
280     }
281
282     while (delim)
283     {
284         argv[argc++] = buf;
285         *delim = '\\0';
286         buf = delim + 1;
287         while (*buf && (*buf == ' ')) /* ignore spaces */
288             buf++;
289
290         if (*buf == '\\')
291         {
292             buf++;
293             delim = strchr(buf, '\\');
294         }
295         else
296         {
297             delim = strchr(buf, ' ');
298         }
299     }
300     argv[argc] = NULL;
301
302     if (argc == 0) /* ignore blank line */
303         return 1;
304
305     /* should the job run in the background? */
306     if ((bg = (*argv[argc - 1] == '&')) != 0)
307     {
308         argv[--argc] = NULL;
309     }
310     return bg;
311 }
312
313 /*
314  * builtin_cmd - If the user has typed a built-in command then execute
315  * it immediately.
316  */
317 int builtin_cmd(char **argv)
318 {
319     if (argv == NULL || *argv == NULL) /* handle null pointer */
320         return 0;
321
322     int ret = 0;
323
324     if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg")) /* bg or fg builtin
command */
325     {
326         do_bgfg(argv);
327         ret = 1; /* indicate a builtin cmd */
328     }
329     else if (!strcmp(argv[0], "jobs")) /* jobs builtin command */
330     {
331         listjobs(jobs); /* show jobs */
332         ret = 1; /* indicate a builtin cmd */
333     }
```

```
333     }
334     else if (!strcmp(argv[0], "quit")) /* quit builtin command */
335     {
336         exit(0); /* quit immediately */
337     }
338
339     return ret;
340 }
341
342 /*
343  * do_bgfg - Execute the builtin bg and fg commands
344  */
345 void do_bgfg(char **argv)
346 {
347     /* $begin handout */
348     struct job_t *jobp = NULL;
349
350     /* Ignore command if no argument */
351     if (argv[1] == NULL)
352     {
353         printf("%s command requires PID or %%jobid argument\n", argv[0]);
354         return;
355     }
356
357     /* Parse the required PID or %%JID arg */
358     if (isdigit(argv[1][0]))
359     {
360         pid_t pid = atoi(argv[1]);
361         if (!(jobp = getjobpid(jobs, pid)))
362         {
363             printf("(%d): No such process\n", pid);
364             return;
365         }
366     }
367     else if (argv[1][0] == '%')
368     {
369         int jid = atoi(&argv[1][1]);
370         if (!(jobp = getjobjid(jobs, jid)))
371         {
372             printf("%s: No such job\n", argv[1]);
373             return;
374         }
375     }
376     else
377     {
378         printf("%s: argument must be a PID or %%jobid\n", argv[0]);
379         return;
380     }
381
382     /* bg command */
383     if (!strcmp(argv[0], "bg"))
384     {
385         if (kill(-(jobp->pid), SIGCONT) < 0)
386             unix_error("kill (bg) error");
387         jobp->state = BG;
388         printf("[%d] (%d) %s", jobp->jid, jobp->pid, jobp->cmdline);
389     }
390
391     /* fg command */
392     else if (!strcmp(argv[0], "fg"))
393     {
```

```

394         if (kill(-(jobp->pid), SIGCONT) < 0)
395             unix_error("kill (fg) error");
396         jobp->state = FG;
397         waitfg(jobp->pid);
398     }
399     else
400     {
401         printf("do_bgfg: Internal error\n");
402         exit(0);
403     }
404     /* $end handout */
405     return;
406 }
407
408 /*
409  * waitfg - Block until process pid is no longer the foreground process
410  */
411 void waitfg(pid_t pid)
412 {
413     struct job_t* jobp = getjobpid(jobs, pid); /* get corresponding job pointer by pid */
414     if (jobp) /* exist a job with PID pid */
415     {
416         while (jobp->state == FG) /* when job is foreground */
417         {
418             sleep(1); /* block */
419         }
420         /* block until the job is no longer foreground */
421     }
422     return;
423 }
424
425 /*****
426  * Signal handlers
427  *****/
428
429 /*
430  * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
431  *   a child job terminates (becomes a zombie), or stops because it
432  *   received a SIGSTOP or SIGTSTP signal. The handler reaps all
433  *   available zombie children, but doesn't wait for any other
434  *   currently running children to terminate.
435  */
436 void sigchld_handler(int sig)
437 {
438     int olderrno = errno; /* save original errno */
439     sigset_t mask, oldmask;
440     if (sigfillset(&mask))
441         unix_error("sigfillset error");
442
443     if (sigprocmask(SIG_SETMASK, &mask, &oldmask)) /* block all signals */
444         unix_error("sigprocmask error");
445
446     int status; /* child process exit status */
447     pid_t child_pid; /* child process pid */
448
449     while ((child_pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
450         /* child process which terminates or stops */
451     {
452         struct job_t* jobp = getjobpid(jobs, child_pid); /* get corresponding
job pointer by pid */

```

```

453         if (!jobp)
454             app_error("getjobpid (sigchld_handler) error");
455
456         if (WIFEXITED(status)) /* child process exited normally */
457         {
458             if (!deletejob(jobs, child_pid)) /* delete the job */
459                 app_error("deletejob (sigchld_handler) error");
460         }
461         else if (WIFSTOPPED(status)) /* child process stopped */
462         {
463             int sig = WSTOPSIG(status); /* get the signal id which causes child process stopped */
464             printf("Job [%d] (%d) stopped by signal %d\n", jobp->jid, jobp->pid, sig);
465             jobp->state = ST; /* stop the job */
466         }
467         else if (WIFSIGNALED(status)) /* child process terminated */
468         {
469             int sig = WTERMSIG(status); /* get the signal id which causes child process terminated */
470             printf("Job [%d] (%d) terminated by signal %d\n", jobp->jid, jobp->pid, sig);
471             if (!deletejob(jobs, child_pid)) /* delete the job */
472                 app_error("deletejob (sigchld_handler) error");
473         }
474         else
475             app_error("unexpected status returned in waitpid (sigchld_handler)");
476     }
477
478     if (sigprocmask(SIG_SETMASK, &oldmask, NULL)) /* restore original signal mask */
479         unix_error("sigprocmask error");
480     errno = olderrno; /* restore original errno */
481     return;
482 }
483
484 /*
485  * sigint_handler - The kernel sends a SIGINT to the shell whenever the
486  * user types ctrl-c at the keyboard. Catch it and send it along
487  * to the foreground job.
488  */
489 void sigint_handler(int sig)
490 {
491     int olderrno = errno; /* save original errno */
492     sigset_t mask, oldmask;
493     if (sigfillset(&mask))
494         unix_error("sigfillset error");
495
496     if (sigprocmask(SIG_SETMASK, &mask, &oldmask)) /* block all signals */
497         unix_error("sigprocmask error");
498
499     pid_t fg = fgpj(jobs); /* get foreground job pid */
500     if (fg != 0)
501     {
502         if (kill(-fg, SIGINT) < 0) /* send SIGINT to process group */
503             unix_error("kill (ctrl-c) error");
504     }
505
506     if (sigprocmask(SIG_SETMASK, &oldmask, NULL)) /* restore original signal mask */

```

```

507     unix_error("sigprocmask error");
508     errno = olderrno; /* restore original errno */
509     return;
510 }
511
512 /*
513  * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
514  * the user types ctrl-z at the keyboard. Catch it and suspend the
515  * foreground job by sending it a SIGTSTP.
516  */
517 void sigtstp_handler(int sig)
518 {
519     int olderrno = errno; /* save original errno */
520     sigset_t mask, oldmask;
521     if (sigfillset(&mask))
522         unix_error("sigfillset error");
523
524     if (sigprocmask(SIG_SETMASK, &mask, &oldmask)) /* block all signals */
525         unix_error("sigprocmask error");
526
527     pid_t fg = fgpid(jobs); /* get foreground job pid */
528     if (fg != 0)
529     {
530         if (kill(-fg, SIGTSTP) < 0) /* send SIGTSTP to process group */
531             unix_error("kill (ctrl-z) error");
532     }
533
534     if (sigprocmask(SIG_SETMASK, &oldmask, NULL)) /* restore original signal
535 mask */
536         unix_error("sigprocmask error");
537     errno = olderrno; /* restore original errno */
538     return;
539 }
540
541 /* End signal handlers
542 *****/
543
544 /* Helper routines that manipulate the job list
545 *****/
546
547 /* clearjob - Clear the entries in a job struct */
548 void clearjob(struct job_t *job)
549 {
550     {
551         job->pid = 0;
552         job->jid = 0;
553         job->state = UNDEF;
554         job->cmdline[0] = '\0';
555     }
556
557 /* initjobs - Initialize the job list */
558 void initjobs(struct job_t *jobs)
559 {
560     int i;
561
562     for (i = 0; i < MAXJOBS; i++)
563         clearjob(&jobs[i]);
564 }
565
566 /* maxjid - Returns largest allocated job ID */

```

```
567 int maxjid(struct job_t *jobs)
568 {
569     int i, max = 0;
570
571     for (i = 0; i < MAXJOBS; i++)
572         if (jobs[i].jid > max)
573             max = jobs[i].jid;
574     return max;
575 }
576
577 /* addjob - Add a job to the job list */
578 int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)
579 {
580     int i;
581
582     if (pid < 1)
583         return 0;
584
585     for (i = 0; i < MAXJOBS; i++)
586     {
587         if (jobs[i].pid == 0)
588         {
589             jobs[i].pid = pid;
590             jobs[i].state = state;
591             jobs[i].jid = nextjid++;
592             if (nextjid > MAXJOBS)
593                 nextjid = 1;
594             strcpy(jobs[i].cmdline, cmdline);
595             if (verbose)
596             {
597                 printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid, job
s[i].cmdline);
598             }
599             return 1;
600         }
601     }
602     printf("Tried to create too many jobs\n");
603     return 0;
604 }
605
606 /* deletejob - Delete a job whose PID=pid from the job list */
607 int deletejob(struct job_t *jobs, pid_t pid)
608 {
609     int i;
610
611     if (pid < 1)
612         return 0;
613
614     for (i = 0; i < MAXJOBS; i++)
615     {
616         if (jobs[i].pid == pid)
617         {
618             clearjob(&jobs[i]);
619             nextjid = maxjid(jobs) + 1;
620             return 1;
621         }
622     }
623     return 0;
624 }
625
626 /* fgpid - Return PID of current foreground job, 0 if no such job */
```

```
627 pid_t fgpid(struct job_t *jobs)
628 {
629     int i;
630
631     for (i = 0; i < MAXJOBS; i++)
632         if (jobs[i].state == FG)
633             return jobs[i].pid;
634     return 0;
635 }
636
637 /* getjobpid - Find a job (by PID) on the job list */
638 struct job_t *getjobpid(struct job_t *jobs, pid_t pid)
639 {
640     int i;
641
642     if (pid < 1)
643         return NULL;
644     for (i = 0; i < MAXJOBS; i++)
645         if (jobs[i].pid == pid)
646             return &jobs[i];
647     return NULL;
648 }
649
650 /* getjobjid - Find a job (by JID) on the job list */
651 struct job_t *getjobjid(struct job_t *jobs, int jid)
652 {
653     int i;
654
655     if (jid < 1)
656         return NULL;
657     for (i = 0; i < MAXJOBS; i++)
658         if (jobs[i].jid == jid)
659             return &jobs[i];
660     return NULL;
661 }
662
663 /* pid2jid - Map process ID to job ID */
664 int pid2jid(pid_t pid)
665 {
666     int i;
667
668     if (pid < 1)
669         return 0;
670     for (i = 0; i < MAXJOBS; i++)
671         if (jobs[i].pid == pid)
672         {
673             return jobs[i].jid;
674         }
675     return 0;
676 }
677
678 /* listjobs - Print the job list */
679 void listjobs(struct job_t *jobs)
680 {
681     int i;
682
683     for (i = 0; i < MAXJOBS; i++)
684     {
685         if (jobs[i].pid != 0)
686         {
687             printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
```



```
688         switch (jobs[i].state)
689         {
690             case BG:
691                 printf("Running ");
692                 break;
693             case FG:
694                 printf("Foreground ");
695                 break;
696             case ST:
697                 printf("Stopped ");
698                 break;
699             default:
700                 printf("listjobs: Internal error: job[%d].state=%d ",
701                     i, jobs[i].state);
702         }
703         printf("%s", jobs[i].cmdline);
704     }
705 }
706 }
707 /*****
708  * end job list helper routines
709  *****/
710
711 /*****
712  * Other helper routines
713  *****/
714
715 /*
716  * usage - print a help message
717  */
718 void usage(void)
719 {
720     printf("Usage: shell [-hvp]\n");
721     printf("  -h  print this message\n");
722     printf("  -v  print additional diagnostic information\n");
723     printf("  -p  do not emit a command prompt\n");
724     exit(1);
725 }
726
727 /*
728  * unix_error - unix-style error routine
729  */
730 void unix_error(char *msg)
731 {
732     fprintf(stdout, "%s: %s\n", msg, strerror(errno));
733     exit(1);
734 }
735
736 /*
737  * app_error - application-style error routine
738  */
739 void app_error(char *msg)
740 {
741     fprintf(stdout, "%s\n", msg);
742     exit(1);
743 }
744
745 /*
746  * Signal - wrapper for the sigaction function
747  */
748 handler_t *Signal(int signum, handler_t *handler)
```

```
749     {
750         struct sigaction action, old_action;
751
752         action.sa_handler = handler;
753         sigemptyset(&action.sa_mask); /* block sigs of type being handled */
754         action.sa_flags = SA_RESTART; /* restart syscalls if possible */
755
756         if (sigaction(signum, &action, &old_action) < 0)
757             unix_error("Signal error");
758         return (old_action.sa_handler);
759     }
760
761     /*
762     * sigquit_handler - The driver program can gracefully terminate the
763     *   child shell by sending it a SIGQUIT signal.
764     */
765     void sigquit_handler(int sig)
766     {
767         printf("Terminating after receipt of SIGQUIT signal\n");
768         exit(1);
769     }
```

## 第 4 章 TinyShell 测试

总分 15 分

### 4.1 测试方法

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

### 4.2 测试结果评价

tsh 与 tshref 的输出在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

#### 4.3.1 测试用例 trace01.txt 的输出截图 (1 分)

tsh 测试结果		tshref 测试结果	
<pre>./sdriver.pl -t trace01.txt -s ./tsh -a "-p" # # trace01.txt - Properly terminate on EOF. #</pre>		<pre>./sdriver.pl -t trace01.txt -s ./tshref -a "-p" # # trace01.txt - Properly terminate on EOF. #</pre>	
测试结论	相同		

#### 4.3.2 测试用例 trace02.txt 的输出截图 (1 分)

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

<pre>./sdriver.pl -t trace02.txt -s ./tsh -a "-p" # # trace02.txt - Process builtin quit command. #</pre>	<pre>./sdriver.pl -t trace02.txt -s ./tshref -a "-p" # # trace02.txt - Process builtin quit command. #</pre>
测试结论	相同

#### 4.3.3 测试用例 trace03.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace03.txt -s ./tsh -a "-p" # # trace03.txt - Run a foreground job. # tsh&gt; quit</pre>	<pre>./sdriver.pl -t trace03.txt -s ./tshref -a "-p" # # trace03.txt - Run a foreground job. # tsh&gt; quit</pre>
测试结论	相同

#### 4.3.4 测试用例 trace04.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace04.txt -s ./tsh -a "-p" # # trace04.txt - Run a background job. # tsh&gt; ./myspin 1 &amp; [1] (11189) ./myspin 1 &amp;</pre>	<pre>./sdriver.pl -t trace04.txt -s ./tshref -a "-p" # # trace04.txt - Run a background job. # tsh&gt; ./myspin 1 &amp; [1] (11183) ./myspin 1 &amp;</pre>
测试结论	相同

#### 4.3.5 测试用例 trace05.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace05.txt -s ./tsh -a "-p" # # trace05.txt - Process jobs builtin command. # tsh&gt; ./myspin 2 &amp; [1] (11205) ./myspin 2 &amp; tsh&gt; ./myspin 3 &amp; [2] (11207) ./myspin 3 &amp; tsh&gt; jobs [1] (11205) Running ./myspin 2 &amp; [2] (11207) Running ./myspin 3 &amp;</pre>	<pre>./sdriver.pl -t trace05.txt -s ./tshref -a "-p" # # trace05.txt - Process jobs builtin command. # tsh&gt; ./myspin 2 &amp; [1] (11196) ./myspin 2 &amp; tsh&gt; ./myspin 3 &amp; [2] (11198) ./myspin 3 &amp; tsh&gt; jobs [1] (11196) Running ./myspin 2 &amp; [2] (11198) Running ./myspin 3 &amp;</pre>

测试结论	相同
------	----

## 4.3.6 测试用例 trace06.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace06.txt -s ./tsh -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh&gt; ./myspin 4 Job [1] (11509) terminated by signal 2</pre>	<pre>./sdriver.pl -t trace06.txt -s ./tshref -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh&gt; ./myspin 4 Job [1] (11218) terminated by signal 2</pre>
测试结论	相同

## 4.3.7 测试用例 trace07.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace07.txt -s ./tsh -a "-p" # # trace07.txt - Forward SIGINT only to foreground job. # tsh&gt; ./myspin 4 &amp; [1] (11664) ./myspin 4 &amp; tsh&gt; ./myspin 5 Job [2] (11666) terminated by signal 2 tsh&gt; jobs [1] (11664) Running ./myspin 4 &amp;</pre>	<pre>./sdriver.pl -t trace07.txt -s ./tshref -a "-p" # # trace07.txt - Forward SIGINT only to foreground job. # tsh&gt; ./myspin 4 &amp; [1] (11518) ./myspin 4 &amp; tsh&gt; ./myspin 5 Job [2] (11520) terminated by signal 2 tsh&gt; jobs [1] (11518) Running ./myspin 4 &amp;</pre>
测试结论	相同

## 4.3.8 测试用例 trace08.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace08.txt -s ./tsh -a "-p" # # trace08.txt - Forward SIGTSTP only to foreground job. # tsh&gt; ./myspin 4 &amp; [1] (11754) ./myspin 4 &amp; tsh&gt; ./myspin 5 Job [2] (11756) stopped by signal 20 tsh&gt; jobs [1] (11754) Running ./myspin 4 &amp; [2] (11756) Stopped ./myspin 5</pre>	<pre>./sdriver.pl -t trace08.txt -s ./tshref -a "-p" # # trace08.txt - Forward SIGTSTP only to foreground job. # tsh&gt; ./myspin 4 &amp; [1] (11674) ./myspin 4 &amp; tsh&gt; ./myspin 5 Job [2] (11676) stopped by signal 20 tsh&gt; jobs [1] (11674) Running ./myspin 4 &amp; [2] (11676) Stopped ./myspin 5</pre>
测试结论	相同

#### 4.3.9 测试用例 trace09.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace09.txt -s ./tsh -a "-p" # # trace09.txt - Process bg builtin command # tsh&gt; ./myspin 4 &amp; [1] (11776) ./myspin 4 &amp; tsh&gt; ./myspin 5 Job [2] (11778) stopped by signal 20 tsh&gt; jobs [1] (11776) Running ./myspin 4 &amp; [2] (11778) Stopped ./myspin 5 tsh&gt; bg %2 [2] (11778) ./myspin 5 tsh&gt; jobs [1] (11776) Running ./myspin 4 &amp; [2] (11778) Running ./myspin 5</pre>	<pre>./sdriver.pl -t trace09.txt -s ./tshref -a "-p" # # trace09.txt - Process bg builtin command # tsh&gt; ./myspin 4 &amp; [1] (11765) ./myspin 4 &amp; tsh&gt; ./myspin 5 Job [2] (11767) stopped by signal 20 tsh&gt; jobs [1] (11765) Running ./myspin 4 &amp; [2] (11767) Stopped ./myspin 5 tsh&gt; bg %2 [2] (11767) ./myspin 5 tsh&gt; jobs [1] (11765) Running ./myspin 4 &amp; [2] (11767) Running ./myspin 5</pre>
测试结论	相同

#### 4.3.10 测试用例 trace10.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace10.txt -s ./tsh -a "-p" # # trace10.txt - Process fg builtin command. # tsh&gt; ./myspin 4 &amp; [1] (11800) ./myspin 4 &amp; tsh&gt; fg %1 Job [1] (11800) stopped by signal 20 tsh&gt; jobs [1] (11800) Stopped ./myspin 4 &amp; tsh&gt; fg %1 tsh&gt; jobs</pre>	<pre>./sdriver.pl -t trace10.txt -s ./tshref -a "-p" # # trace10.txt - Process fg builtin command. # tsh&gt; ./myspin 4 &amp; [1] (11789) ./myspin 4 &amp; tsh&gt; fg %1 Job [1] (11789) stopped by signal 20 tsh&gt; jobs [1] (11789) Stopped ./myspin 4 &amp; tsh&gt; fg %1 tsh&gt; jobs</pre>
测试结论	相同

#### 4.3.11 测试用例 trace11.txt 的输出截图（1 分）

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

<pre>./sdriver.pl -t trace11.txt -s ./tsh -a "-p" # # trace11.txt - Forward SIGINT to every process in foreground process group # tsh&gt; ./mysplit 4 Job [1] (11905) terminated by signal 2 tsh&gt; /bin/ps a   PID TTY          STAT       TIME COMMAND   1102 tty2      Ssl+    0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SE SSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu   1111 tty2      Sl+      0:00 /usr/libexec/gnome-session-binary --session=ubuntu   10636 pts/0     Ss+      0:00 bash   10877 pts/1     Ss        0:00 bash   11900 pts/1     S+        0:00 make test11   11901 pts/1     S+        0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tsh -a "-p"   11902 pts/1     S+        0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh -a -p   11903 pts/1     S+        0:00 ./tsh -p   11908 pts/1     R         0:00 /bin/ps a</pre>	<pre>./sdriver.pl -t trace11.txt -s ./tshref -a "-p" # # trace11.txt - Forward SIGINT to every process in foreground process group # tsh&gt; ./mysplit 4 Job [1] (11820) terminated by signal 2 tsh&gt; /bin/ps a   PID TTY          STAT       TIME COMMAND   1102 tty2      Ssl+    0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SE SSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu   1111 tty2      Sl+      0:00 /usr/libexec/gnome-session-binary --session=ubuntu   10636 pts/0     Ss+      0:00 bash   10877 pts/1     Ss        0:00 bash   11815 pts/1     S+        0:00 make rtest11   11816 pts/1     S+        0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tshref -a "-p"   11817 pts/1     S+        0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tshr ef -a -p   11818 pts/1     S+        0:00 ./tshref -p   11823 pts/1     R         0:00 /bin/ps a</pre>
测试结论	相同

#### 4.3.12 测试用例 trace12.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace12.txt -s ./tsh -a "-p" # # trace12.txt - Forward SIGTSTP to every process in foreground process group # tsh&gt; ./mysplit 4 Job [1] (12190) stopped by signal 20 tsh&gt; jobs [1] (12190) Stopped ./mysplit 4 tsh&gt; /bin/ps a   PID TTY          STAT       TIME COMMAND   1102 tty2      Ssl+    0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SE SSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu   1111 tty2      Sl+      0:00 /usr/libexec/gnome-session-binary --session=ubuntu   10636 pts/0     Ss+      0:00 bash   10877 pts/1     Ss        0:00 bash   12185 pts/1     S+        0:00 make test12   12186 pts/1     S+        0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tsh -a "-p"   12187 pts/1     S+        0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tsh -a -p   12188 pts/1     S+        0:00 ./tsh -p   12190 pts/1     T         0:00 ./mysplit 4   12191 pts/1     T         0:00 ./mysplit 4   12194 pts/1     R         0:00 /bin/ps a</pre>	<pre>./sdriver.pl -t trace12.txt -s ./tshref -a "-p" # # trace12.txt - Forward SIGTSTP to every process in foreground process group # tsh&gt; ./mysplit 4 Job [1] (12018) stopped by signal 20 tsh&gt; jobs [1] (12018) Stopped ./mysplit 4 tsh&gt; /bin/ps a   PID TTY          STAT       TIME COMMAND   1102 tty2      Ssl+    0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SE SSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu   1111 tty2      Sl+      0:00 /usr/libexec/gnome-session-binary --session=ubuntu   10636 pts/0     Ss+      0:00 bash   10877 pts/1     Ss        0:00 bash   12013 pts/1     S+        0:00 make rtest12   12014 pts/1     S+        0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tshref -a "-p"   12015 pts/1     S+        0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tshr ef -a -p   12016 pts/1     S+        0:00 ./tshref -p   12018 pts/1     T         0:00 ./mysplit 4   12019 pts/1     T         0:00 ./mysplit 4   12022 pts/1     R         0:00 /bin/ps a</pre>
测试结论	相同

#### 4.3.13 测试用例 trace13.txt 的输出截图（1 分）

tsh 测试结果
----------

```
./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (12260) stopped by signal 20
tsh> jobs
[1] (12260) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY          STAT       TIME COMMAND
  1102 tty2      Ssl+      0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1111 tty2      Sl+       0:00 /usr/libexec/gnome-session-binary --session=ubuntu
 10636 pts/0      Ss+       0:00 bash
 10877 pts/1      Ss        0:00 bash
 12255 pts/1      S+        0:00 make test13
 12256 pts/1      S+        0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
 12257 pts/1      S+        0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
 12258 pts/1      S+        0:00 ./tsh -p
 12260 pts/1      T         0:00 ./mysplit 4
 12261 pts/1      T         0:00 ./mysplit 4
 12265 pts/1      R         0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
  PID TTY          STAT       TIME COMMAND
  1102 tty2      Ssl+      0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1111 tty2      Sl+       0:00 /usr/libexec/gnome-session-binary --session=ubuntu
 10636 pts/0      Ss+       0:00 bash
 10877 pts/1      Ss        0:00 bash
 12255 pts/1      S+        0:00 make test13
 12256 pts/1      S+        0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
 12257 pts/1      S+        0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
 12258 pts/1      S+        0:00 ./tsh -p
 12268 pts/1      R         0:00 /bin/ps a
```

## tshref 测试结果

```
./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (12332) stopped by signal 20
tsh> jobs
[1] (12332) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY          STAT       TIME COMMAND
  1102 tty2      Ssl+      0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1111 tty2      Sl+       0:00 /usr/libexec/gnome-session-binary --session=ubuntu
 10636 pts/0      Ss+       0:00 bash
 10877 pts/1      Ss        0:00 bash
 12327 pts/1      S+        0:00 make rtest13
 12328 pts/1      S+        0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
 12329 pts/1      S+        0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshref -a -p
 12330 pts/1      S+        0:00 ./tshref -p
 12332 pts/1      T         0:00 ./mysplit 4
 12333 pts/1      T         0:00 ./mysplit 4
 12336 pts/1      R         0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
  PID TTY          STAT       TIME COMMAND
  1102 tty2      Ssl+      0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
  1111 tty2      Sl+       0:00 /usr/libexec/gnome-session-binary --session=ubuntu
 10636 pts/0      Ss+       0:00 bash
 10877 pts/1      Ss        0:00 bash
 12327 pts/1      S+        0:00 make rtest13
 12328 pts/1      S+        0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
 12329 pts/1      S+        0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshref -a -p
 12330 pts/1      S+        0:00 ./tshref -p
 12339 pts/1      R         0:00 /bin/ps a
```

测试结论

相同



## 4.3.14 测试用例 trace14.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
<pre>./sdriver.pl -t trace14.txt -s ./tsh -a "-p" # # trace14.txt - Simple error handling # tsh&gt; ./bogus ./bogus: Command not found tsh&gt; ./myspin 4 &amp; [1] (12420) ./myspin 4 &amp; tsh&gt; fg fg command requires PID or %jobid argument tsh&gt; bg bg command requires PID or %jobid argument tsh&gt; fg a fg: argument must be a PID or %jobid tsh&gt; bg a bg: argument must be a PID or %jobid tsh&gt; fg 9999999 (9999999): No such process tsh&gt; bg 9999999 (9999999): No such process tsh&gt; fg %2 %2: No such job tsh&gt; fg %1 Job [1] (12420) stopped by signal 20 tsh&gt; bg %2 %2: No such job tsh&gt; bg %1 [1] (12420) ./myspin 4 &amp; tsh&gt; jobs [1] (12420) Running ./myspin 4 &amp;</pre>	<pre>./sdriver.pl -t trace14.txt -s ./tshref -a "-p" # # trace14.txt - Simple error handling # tsh&gt; ./bogus ./bogus: Command not found tsh&gt; ./myspin 4 &amp; [1] (12374) ./myspin 4 &amp; tsh&gt; fg fg command requires PID or %jobid argument tsh&gt; bg bg command requires PID or %jobid argument tsh&gt; fg a fg: argument must be a PID or %jobid tsh&gt; bg a bg: argument must be a PID or %jobid tsh&gt; fg 9999999 (9999999): No such process tsh&gt; bg 9999999 (9999999): No such process tsh&gt; fg %2 %2: No such job tsh&gt; fg %1 Job [1] (12374) stopped by signal 20 tsh&gt; bg %2 %2: No such job tsh&gt; bg %1 [1] (12374) ./myspin 4 &amp; tsh&gt; jobs [1] (12374) Running ./myspin 4 &amp;</pre>
测试结论	相同

## 4.3.15 测试用例 trace15.txt 的输出截图（1 分）

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

<pre>./sdriver.pl -t trace15.txt -s ./tsh -a "-p" # # trace15.txt - Putting it all together # tsh&gt; ./bogus ./bogus: Command not found tsh&gt; ./myspin 10 Job [1] (12486) terminated by signal 2 tsh&gt; ./myspin 3 &amp; [1] (12488) ./myspin 3 &amp; tsh&gt; ./myspin 4 &amp; [2] (12490) ./myspin 4 &amp; tsh&gt; jobs [1] (12488) Running ./myspin 3 &amp; [2] (12490) Running ./myspin 4 &amp; tsh&gt; fg %1 Job [1] (12488) stopped by signal 20 tsh&gt; jobs [1] (12488) Stopped ./myspin 3 &amp; [2] (12490) Running ./myspin 4 &amp; tsh&gt; bg %3 %3: No such job tsh&gt; bg %1 [1] (12488) ./myspin 3 &amp; tsh&gt; jobs [1] (12488) Running ./myspin 3 &amp; [2] (12490) Running ./myspin 4 &amp; tsh&gt; fg %1 tsh&gt; quit</pre>	<pre>./sdriver.pl -t trace15.txt -s ./tshref -a "-p" # # trace15.txt - Putting it all together # tsh&gt; ./bogus ./bogus: Command not found tsh&gt; ./myspin 10 Job [1] (12466) terminated by signal 2 tsh&gt; ./myspin 3 &amp; [1] (12468) ./myspin 3 &amp; tsh&gt; ./myspin 4 &amp; [2] (12470) ./myspin 4 &amp; tsh&gt; jobs [1] (12468) Running ./myspin 3 &amp; [2] (12470) Running ./myspin 4 &amp; tsh&gt; fg %1 Job [1] (12468) stopped by signal 20 tsh&gt; jobs [1] (12468) Stopped ./myspin 3 &amp; [2] (12470) Running ./myspin 4 &amp; tsh&gt; bg %3 %3: No such job tsh&gt; bg %1 [1] (12468) ./myspin 3 &amp; tsh&gt; jobs [1] (12468) Running ./myspin 3 &amp; [2] (12470) Running ./myspin 4 &amp; tsh&gt; fg %1 tsh&gt; quit</pre>
测试结论	相同

#### 4.4 自测试评分

根据节 4.3 的自测试结果，程序的测试评分为：15。

## 第 4 章 总结

### 4.1 请总结本次实验的收获

通过本次实验，加深了我们对于信号、进程相关概念的了解，并了解了 shell 的基本工作原理。

### 4.2 请给出对本次实验内容的建议

对 main 里面的代码进行修改，否则在泰山服务器上无法正常运行：

将 `char c;` 改为 `int c;`

或将 `while ((c = getopt(argc, argv, "hvp")) != EOF)` 改为 `while ((c = getopt(argc, argv, "hvp")) != (char)EOF)`。

注：本章为酌情加分项。

## 参考文献

- [1] Linux 系统编程之进程（system 函数、popen 函数）  
[https://blog.csdn.net/m0\\_62140641/article/details/133889033](https://blog.csdn.net/m0_62140641/article/details/133889033)
- [2] shell 的功能  
<https://blog.csdn.net/hzp020/article/details/83765267>
- [3] [读书笔记]CSAPP：19[VB]ECF：信号和非本地跳转  
<https://zhuanlan.zhihu.com/p/117269612>