

# 计算机系统

## LAB4实验报告

班级：人工智能2103

学号：202107030125

姓名：姚丁钰

---

实验目的

实验准备

实验内容

Shell介绍

实验要求

实验评测

实验设计

0.explore

1.eval()函数

2.builtin\_cmd()函数

3.do\_bfg()函数

4.waitfg()函数

5.sigchld\_handler()函数

sigint\_handler()函数

sigtstp\_handler()函数

实验测试

1.trace01->正确终止EOF

2.trace 02->实现内置的quit

3.trace 03->运行一个前台job

4.trace 04->运行后台job

5.trace 05->处理jobs内置命令

6.trace 06->将SIGINT转发到前台作业

7.trace 07->仅将SIGINT转发给前台作业

8.trace 08->仅将SIGINT转发给前台作业

9.trace 09->进程bg内置命令

10.trace 10->进程fg内置命令

11.trace 11->将SIGINT转发给前台进程组中的每个进程

- 12.trace 12->将SIGTSTP转发到前台进程组中的每个进程
- 13.trace 13->重新启动进程组中的每个已停止的进程
- 14.trace 14->简单的错误处理
- 15.trace 15->把它们放在一起
- 16.trace 16->测试shell是否能够处理来自其他进程而不是终端的SIGTSTP和SIGINT信号

## 实验总结

---

# 实验目的

Shell Lab - Writing Your Own Unix Shell主要是有关于进程、信号以及异常处理的实验，对应于书本的第8章：异常控制流。主要通过实现一个带作业控制的 Unix Shell来熟悉进程以及信号的概念。

本实验已经给出了Tiny Shell的源程序的基本框架(`tsh.c`)，你需要做的是填充该框架中的`eval` `builtin_cmd` `do_bgfg` `waitfg` `sigchld_handler` `sigint_handler` `sigtstp_handler`等函数。使得编译后的Shell具有相应的功能。

本次实验需要完成一个简单的shell，只需要在已给出的程序中进行补充，实现七个函数。需要完成的函数如下：

- `eval`: 解析和解释命令行 [70 lines]
- `builtin_cmd`: 识别并解释内建命令： quit, fg, bg, and jobs [25 lines]
- `do_bgfg`: 补全bg和fg命令 [50 lines]
- `waitfg`: 等待一个前台作业完成 [20 lines]
- `sigchld handler`: 处理SIGCHLD信号 80 lines]
- `sigint handler`: 处理SIGINT信号 [15 lines]
- `sigtstp handler`: 处理SIGTSTP信号 [15 lines]

每次修改 `tsh.c` 文件时，输入 `make` 以重新编译它。要运行shell，在命令行输入 `tsh`:

```
1 | ./tsh
```

这个简单的shell提供了一些内置命令，如下：

- `jobs`: 列出运行和停止了的后台作业
- `bg` : 通过发送SIGCONT信号，使一个停止运行的后台作业运行

- `fg`: 通过发送 SIGCONT信号，使一个停止或运行中的后台作业转到前台运行
- `kill`: 终止作业
- `quit`: 终止shell

## 实验准备

---

1. 进程的概念、状态以及控制进程的几个函数 (`fork,waitpid,execve`) 。
2. 信号的概念，会编写正确安全的信号处理程序。
3. shell的概念，理解shell程序是如何利用进程管理和信号去执行一个命令行语句。

## 实验内容

---

### Shell介绍

Shell是一个代表用户运行程序的命令行解释器。一个Shell周期性的打印一个提示符，从标准输入流等待一条命令行输入，然后根据命令行的输入执行相应的功能。

一条命令行输入是由空格分隔ASCII文本词(words)。命令行输入的第一个词要不然是一個内建命令(built-in command)要不然是一個可执行文件的路径。剩余的词是命令行参数。如果第一个词是内建命令，Shell立刻在当前进程中执行该命令。否则，词会被假设为一个可执行程序的路径。在这种情况下，Shell会fork出一个子进程，在子进程的上下文中加载并运行这个程序。被创建的子进程被称作任务(job)。总的来说，一个任务可以包含通过Unix管道(Unix Pipe)连接的多个子进程。

如果命令行输入以`'&'`结尾，那么这个任务将会在后台执行，这意味着Shell在打印下一个提示符并等待下一条命令行输入之前不会等待当前任务终止。在默认情况下，任务运行在前台，这意味这Shell在下一条命令行输入之前会等待当前任务终止。在任何的情况下，只能有一个任务运行在前台，但是，可以有多个任务运行在后台。

例如，输入`tsh> jobs`使得Shell执行内建的jobs命令，输入`tsh> /bin/ls -l -d`则在前台运行ls程序，同时，这个程序的`argc == 3`, `argv[0] == "/bin/ls"`, `argv[1] == "-l"`, `argv[2] == "-d"`。相应地，输入`tsh> /bin/ls -l -d &`则会在后台运行ls程序。

Unix Shell支持作业控制的概念，这允许用户将任务在前台和后台间来回移动，并且改变一个任务中进程的状态（运行，停止，终止）。按下`ctrl-c`将会发送SIGINT信号到当前前台任务的每一个进程中。按下`ctrl-z`将会发送SIGTSTP信号到前台任务的每一个进程中，SIGTSTP信号的默认功能是将进程设置为停止状态，直到其被一个SIGCONT信号唤醒。Unix Shell提供了不同的内建命令来支持作业控制，如：

- `jobs` - 打印运行的和停止的后台任务
- `bg` - 将一个停止的后台任务转变为一个运行的后台任务
- `fg` - 将一个运行的或是停止的后台任务转变为一个运行的前台任务
- `kill` - 终止一个任务

## 实验要求

所实现的tsh shell应当具有如下的特性：

- 提示符应当为 "`tsh>`"
- 用户的命令行输入应当包括一个名字和0个或多个参数，均由一个或多个空格隔开。如果名字是一个内建命令，那么tsh应当立刻处理它并且等待下一个命令行输入。否则，tsh就会假设它是一个可执行文件的路径，并且在一个独立的子进程中加载并执行它
- tsh不需要支持管道(`|`)和I/O重定向(`>`和`<`)
- 按下`ctrl-c`(`ctrl-z`)将会导致一个SIGINT(SIGTSTP)信号被发送至当前的前台任务，如果现在没有前台任务，那么这些信号将没有效果
- 如果命令行输入以一个`'&'`结束，那么tsh将会在后台运行这个任务，否则它会在前台运行这个任务
- 每一个任务都可以被一个进程ID(PID)或是一个被tsh分配的正整数的任务ID(JID)唯一的标识。JIDS可以被前缀`'%'`标识，例如，`'%5'`就代表了JID 5，而`'5'`就代表了PID 5
- tsh应当提供如下的内建命令：
  - `quit`命令直接终止shell
  - `jobs`命令列出所有在后台运行的任务
  - `bg` 命令通过给发送SIGCONT信号将其重启，然后将其在后台运行。参数既可以是一个PID，也可以是一个JID
  - `fg` 命令通过给发送SIGCONT信号将其重启，然后将其在前台运行。参数既可以是一个PID，也可以使一个JID

- tsh应当回收它的所有僵死子进程。如果任何任务因为收到了一个未被捕获的信号而终止，那么tsh应当识别对应的事件并且输出相应的信息

## 实验评测

1. 参考答案 - 实验提供了一个作为参考的tshref可执行文件作为tsh的参考。你的tsh应当提供和tshref一致的输出（除了IDs以外）
2. Shell驱动 - sdriver.pl程序将shell作为一个子进程来执行，根据trace file来向它发送命令和信号，并且将shell的输出捕获并输出。实验总共提供了16个trace file。关于sdriver的具体用法请参考实验讲义。

实验提供了tshref.out作为16个trace file在tshref程序下的参考输出。为了方便比较，写了一个小的（可能会有很多错误）的shell脚本gather\_output.sh用来生成这些trace file在tsh下的输出，脚本如下：

```
1 #!/usr/bin bash
2
3 output_file=tsh.out
4
5 echo "" > tsh.out
6
7 echo -e "\n" >> tsh.out
8
9 for trace_file in ./*.txt
10 do
11     echo -e `./sdriver.pl -t $trace_file -s ./tsh -a "-`>> tsh.out
12     p`" `>> tsh.out
13 done
14 echo -e "\n" >> tsh.out
```

## 实验设计

### 0.explore

- tsh.c 具体内容

首先定义了一些宏

```
1 /* 定义了一些宏 */
2 #define MAXLINE    1024   /* max line size */
3 #define MAXARGS    128    /* max args on a command line */
4 #define MAXJOBS    16     /* max jobs at any point in time
   */
5 #define MAXJID     1<<16  /* max job ID */
```

定义了四种进程状态

```
1 /* 工作状态 */
2 #define UNDEF 0 /* undefined */
3 #define FG 1   /* 前台状态 */
4 #define BG 2   /* 后台状态 */
5 #define ST 3   /* 挂起状态 */
```

为了管理作业，作业的相关信息会保存在一个结构体内，包含jid, pid, 状态的等信息

```
1 struct job_t {           /* The job struct */
2     pid_t pid;            /* job PID */
3     int jid;               /* job ID [1, 2, ...] */
4     int state;              /* UNDEF, BG, FG, or ST */
5     char cmdline[MAXLINE]; /* command line */
6 };
7 struct job_t jobs[MAXJOBS]; /* The job list */
```

除作业相关的定义外，还有一些已定义好的全局变量，可以直接使用

```
1 /* Global variables */
2 extern char **environ;      /* 环境变量 */
3 char prompt[] = "tsh> ";    /* command line prompt (DO NOT
   CHANGE) */
4 int verbose = 0;             /* if true, print additional
   output */
5 int nextjid = 1;              /* 下一个分配的jid */
6 char sbuf[MAXLINE];         /* for composing sprintf
   messages */
```

接着是需要我们完成的七个函数定义

```
1 void eval(char *cmdline)//分析命令，并派生子进程执行 主要功能是  
解析cmdline并运行  
2 int builtin_cmd(char **argv)//解析和执行bulidin命令，包括  
quit, fg, bg, and jobs  
3 void do_bgfg(char **argv) //执行bg和fg命令  
4 void waitfg(pid_t pid)//实现阻塞等待前台程序运行结束  
5 void sigchld_handler(int sig)//SIGCHID信号处理函数  
6 void sigint_handler(int sig)//信号处理函数，响应 SIGINT  
(ctrl-c) 信号  
7 void sigtstp_handler(int sig)//信号处理函数，响应 SIGTSTP  
(ctrl-z) 信号
```

下面就是一些辅助的函数

```
1 int parseline(const char *cmdline, char **argv);  
//解析命令行  
2 void sigquit_handler(int sig);  
//处理quit信号  
3 void clearjob(struct job_t *job);  
//清除job_t结构  
4 void initjobs(struct job_t *jobs);  
//初始化作业列表  
5 int maxjid(struct job_t *jobs);  
//获取最大JID  
6 int addjob(struct job_t *jobs, pid_t pid, int state, char  
*cmdline); //增加一个作业  
7 int deletejob(struct job_t *jobs, pid_t pid);  
//根据pid删除作业  
8 pid_t fgpid(struct job_t *jobs);  
//返回当前前台作业的pid  
9 struct job_t *getjobpid(struct job_t *jobs, pid_t pid);  
//根据pid获取作业结构  
10 struct job_t *getjobjid(struct job_t *jobs, int jid);  
//根据jid获取作业结构  
11 int pid2jid(pid_t pid);  
//根据pid获取jid  
12 void listjobs(struct job_t *jobs);  
//输出作业列表  
13 void usage(void);  
//打印help信息  
14 void unix_error(char *msg);  
//包装好的错误报告函数
```

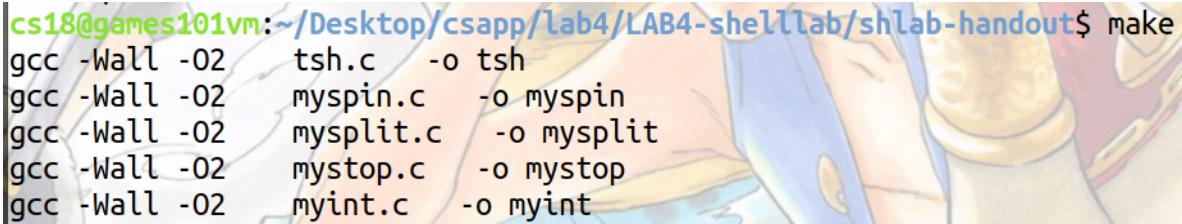
```

15 void app_error(char *msg);
//应用错误报告函数
16 typedef void handler_t(int);
//信号处理程序类型
17 handler_t *Signal(int signum, handler_t *handler);
//设定信号处理程序

```

接着就是main函数，作用是在文件中逐行获取命令，并且判断是不是文件结束(EOF)，将命令 cmdline送入 eval 函数进行解析。我们需要做的就是逐步完善这个过程

- 使用 make 命令编译 tsh.c 文件

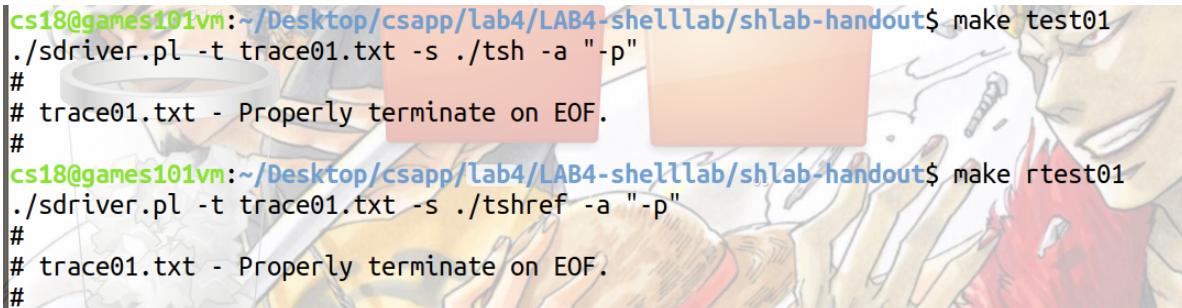


```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make
gcc -Wall -O2 tsh.c -o tsh
gcc -Wall -O2 myspin.c -o myspin
gcc -Wall -O2 mysplit.c -o mysplit
gcc -Wall -O2 mystop.c -o mystop
gcc -Wall -O2 myint.c -o myint

```

然后就执行 make rtest01 make test01 进行比对，如果我们的执行结果与参考结果一致，则实现正确

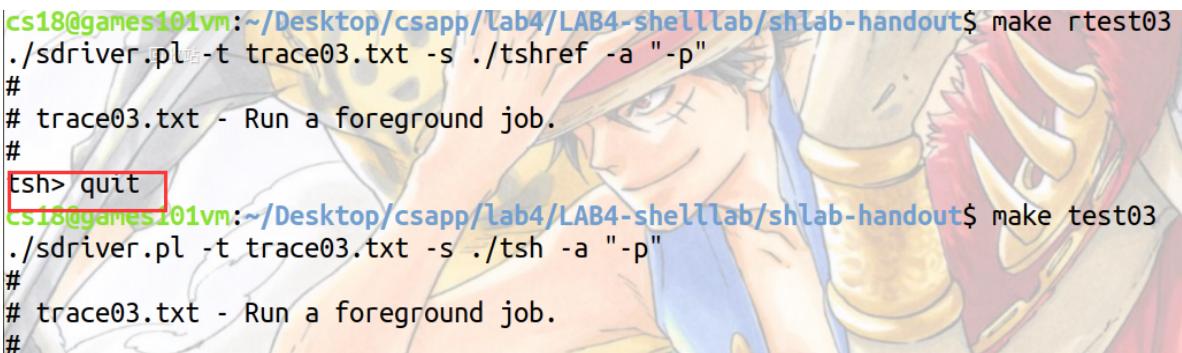


```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#

```

否则不正确



```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest03
./sdriver.pl -t trace03.txt -s ./tshref -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test03
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
# trace03.txt - Run a foreground job.
#

```

输出不一致，说明功能未成功实现

- 例如 ./myspin 20 &后台停顿20秒 然后看看 jobs 示例出所有目前的工作  
看看示例程序怎么做的 就输入

- ```
1 ./tshref (示例程序)
2 ./myspin 20 & (后台程序启动 20s '&' 指的是后台)
3 jobs (列出目前所有状态的程序)
```

输入启动一个后台程序后,输出了子进程的pid 还有jid (在tsh.c中定义) 还有我们的输入的命令,对于jobs 列出了我们所有的示例程序

```
cs18@games101vm:~/Desktop/csapp/Lab4/LAB4-shellLab/shlab-handout$ ./tshref
tsh> ./myspin 20 &
[1] (3343) ./myspin 20 &
tsh> jobs
[1] (3343) Running ./myspin 20 &
```

可以就按照这样的思路 和调试 去试我们程序的反馈 还有应该出现的行文 我们就可以按照下面的示例程序输出来做我们的程序

## 1.eval()函数

### 函数功能

eval函数用于解析和解释命令行。eval首先解析命令行，如果用户请求一个内置命令quit、jobs、bg或fg（即内置命令）那么就立即执行。否则，fork子进程并在子进程中运行作业。如果作业正在运行前台，等待它终止，然后返回。

- ```
1 /*
2  eval-评估用户输入的命令行
3  如果用户请求内置命令 (quit、jobs、bg或fg) ,那么就立即执行它。
4  否则, fork一个子进程并在子进程中运行作业。
5  如果作业正在运行前台, 等待它终止, 然后返回。
6  注:每个子进程必须具有唯一的进程组ID, 以便当在键盘上键入ctrl-c
    (ctrl-z) 时, 后台子进程不从内核接收SIGINT (SIGTSTP)
7 */
```

### 函数原型

```
void eval(char *cmdline), 传入的参数为cmdline, 即命令行字符串
```

### 函数思路

1. 读取命令行 (调用函数 `parseline()` 提取参数)
2. 判断是否是内置命令 (调用函数 `builtin_cmd()` )
  - 是内置命令：函数 `builtin_cmd()` 会直接执行，本函数return就好

- 不是内置命令：

- 创建子进程
- 寻找路径中的可执行文件并执行（调用函数 `execve()` 通过参数 `argv[0]` 来寻找路径中的可执行文件，并执行）
- 将作业添加至后台作业管理的函数使用（后台or前台？）

注意：需要写信号阻塞

```
1 void eval(char * cmdline)
2 {
3     char* argv[MAXARGS]; // execve() 函数的参数，存储命令行
4     // 参数
5     int state = UNDEF; // 工作状态， FG 表示前台作业， BG 表示后
6     // 台作业
7     sigset_t set; // 信号集
8     pid_t pid; // 进程id
9
10    // 处理输入的数据
11    if(parseline(cmdline, argv) == 1) // 解析命令行，返回给
12        argv数组
13        state = BG; // 后台作业
14    else
15        state = FG; // 前台作业
16
17    // 如果不是内置命令
18    if(!builtin_cmd(argv))
19    {
20        if(sigemptyset(&set) < 0)
21            unix_error("sigemptyset error"); // 清空信号
22            集，如果失败则报错
23            if(sigaddset(&set, SIGINT) < 0 || sigaddset(&set,
24            SIGTSTP) < 0 || sigaddset(&set, SIGCHLD) < 0)
25            unix_error("sigaddset error"); // 添加
26            SIGINT、SIGTSTP、SIGCHLD 信号到信号集，如果失败则报错
27
28            // 在派生子进程之前阻塞 SIGCHLD 信号，防止竞争条件
29            if(sigprocmask(SIG_BLOCK, &set, NULL) < 0)
30                unix_error("sigprocmask error"); // 阻塞信号集
31                中的信号，如果失败则报错
```

```
28     if((pid = fork()) < 0) // fork创建子进程失败
29         unix_error("fork error"); // 如果fork失败，则
30         报错
31     else if(pid == 0) // fork创建子进程
32     {
33         // 子进程的控制流开始
34
35         if(sigprocmask(SIG_UNBLOCK, &set, NULL) < 0)
36             // 解除阻塞
37             unix_error("sigprocmask error"); // 解除
38             信号集中的阻塞信号，如果失败则报错
39
40         if(setpgid(0, 0) < 0) // 设置子进程id
41             unix_error("setpgid error"); // 设置子进程
42             组ID，如果失败则报错
43
44         if(execve(argv[0], argv, environ) < 0){ // 执
45             行命令，如果失败则报错
46             printf("%s: command not found\n",
47             argv[0]); // 打印未找到命令的错误信息
48             exit(0); // 子进程退出
49         }
50
51         // 将当前进程添加进job中，无论是前台进程还是后台进程
52         addjob(jobs, pid, state, cmdline); // 将子进程添加
53         到作业列表中
54
55         // 恢复受阻塞的信号SIGINT、SIGTSTP、SIGCHLD
56         if(sigprocmask(SIG_BLOCK, &set, NULL) < 0)
57             unix_error("sigprocmask error"); // 解除信号集
58             中的阻塞信号，如果失败则报错
59
60         // 判断子进程类型并做处理
61         if(state == FG)
62             waitfg(pid); // 前台作业等待
63         else
64             printf("[%d] (%d) %s", pid2jid(pid), pid,
65             cmdline); // 将进程id映射到job id，并打印后台作业信息
66     }
67
68     return;
```

```
61 }
```

注意：

1. 每个子进程必须有自己独一无二的进程组id，通过在fork()之后的子进程中Setpgid(0,0)实现，这样当向前台程序发送ctrl+c或ctrl+z命令时，才不会影响到后台程序。如果没有这一步，则所有的子进程与当前的tsh shell进程为同一个进程组，发送信号时，前后台的子进程均会收到。
2. 在fork()新进程前要阻塞SIGCHLD信号，防止出现竞争，这是经典的同步错误，如果不阻塞会出现子进程先结束从jobs中删除，然后再执行到主进程addjob的竞争问题。

## 2.builtin\_cmd()函数

函数功能

识别并执行内置命令：quit, fg, bg, 和 jobs。

函数原型

```
int builtin_cmd(char **argv), 参数为argv参数列表
```

函数思路

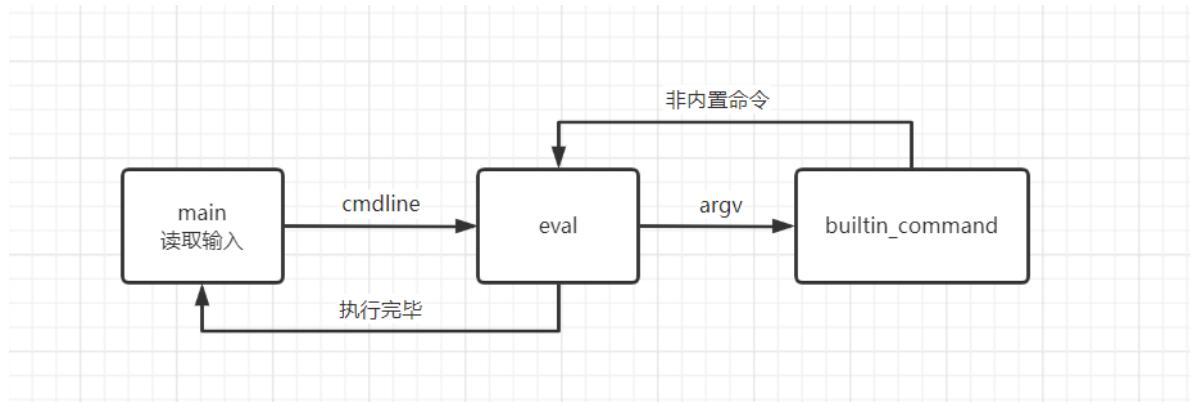
- 当命令行参数为quit时，直接终止shell
- 当命令行参数为jobs时，调用listjobs函数，显示job列表
- 当命令行参数为bg或fg时，调用do\_bgfg函数，执行内置的bg和fg命令
- 不是内置命令时返回0

```
1 int builtin_cmd(char **argv)
2 {
3     if (!strcmp(argv[0], "quit")) // 如果命令是quit，退出
4         exit(0);
5     else if (!strcmp(argv[0], "bg") || !strcmp(argv[0],
6         "fg")) // 如果是bg或者fg命令，执行do_fgfg函数
7         do_bgfg(argv);
8     else if (!strcmp(argv[0], "jobs")) // 如果命令是jobs，
9         listjobs(jobs);
10    else
11        return 0;      // 不是内置命令
12    return 1; // 是内置命令
13 }
```

main函数中负责读入cmdline发送给eval函数进行处理，如果发现读入EOF则退出程序。

eval函数的主要流程为使用parseline函数将cmdline解析为argv数组，然后发送到builtin\_command函数进行处理，如果内置命令则在此函数内直接处理并返回1，反之则不处理返回0交还控制权到eval函数。

接下来eval函数运用fork-execve惯用法执行cmdline，父进程根据cmdline为前台或后台程序做不同处理，前台程序则等待其子进程执行完毕，后台程序则直接输出子进程PID和命令，而后返回控制权给main函数继续读入新的cmdline。



## 3.do\_bgfg()函数

### 函数功能

实现内置命令bg 和 fg

- bg :将停止的后台作业更改为正在运行的后台作业。通过发送SIGCONT信号重新启动，然后在后台运行它。参数可以是PID，也可以是JID。ST -> BG
- fg :将已停止或正在运行的后台作业更改为前台正在运行的作业。通过发送SIGCONT信号重新启，然后在前台运行它。参数可以是PID，也可以是JID。ST -> FG, BG -> FG

### 函数原型

```
void do_bgfg(char **argv), 参数为argv 参数列表
```

### 函数思路

1. 解析参数，判断是否会出现命令错误。
2. 找到需要操作的进程，然后用kill函数对它发出信号，因为前面创建进程的时候都是用setgid，每个进程在单独一个进程组，所以这里可以用kill，不用担心对其他进程发送信号。

- 对于 bg 命令，我们只是向目标进程发送SIGCONT 信号，让它继续执行；
- 对于 fg 命令，先判断目标进程是不是已经暂停了（如果是，就先启动它）；之后调用 waitfg 等待进程结束。

```

1 void do_bgfg(char **argv)
2 {
3     int num;
4     struct job_t *job;
5
6     // 没有参数的fg/bg应该被丢弃
7     if (!argv[1]) {
8         printf("%s command requires PID or %%jobid
argument\n", argv[0]);
9         return;
10    }
11
12    // 检测fg/bg参数，其中%开头的数字是JobID，纯数字的是PID
13    if (argv[1][0] == '%') { // 解析jid
14        if ((num = strtol(&argv[1][1], NULL, 10)) <= 0) {
15            printf("%s: argument must be a PID or
%%jobid\n", argv[0]); // 解析失败，打印错误消息
16            return;
17        }
18        if ((job = getjobjid(jobs, num)) == NULL) {
19            printf("%%%d: No such job\n", num); // 没找到
对应的job
20            return;
21        }
22    } else {
23        if ((num = strtol(argv[1], NULL, 10)) <= 0) {
24            printf("%s: argument must be a PID or
%%jobid\n", argv[0]); // 解析失败，打印错误消息
25            return;
26        }
27        if ((job = getjobpid(jobs, num)) == NULL) {
28            printf("(%d): No such process\n", num); // 没
找到对应的进程
29            return;
30        }
31    }
32 }
```

```

33     if (!strcmp(argv[0], "bg")) {
34         // bg会启动子进程，并将其放置于后台执行
35         job->state = BG; // 设置状态
36         if (kill(-job->pid, SIGCONT) < 0) // 采用负数发送
37             unix_error("kill error");
38         printf("[%d] (%d) %s", job->jid, job->pid, job-
39             >cmdline);
40     } else if (!strcmp(argv[0], "fg")) {
41         job->state = FG; // 设置状态
42         if (kill(-job->pid, SIGCONT) < 0) // 采用负数发送
43             unix_error("kill error");
44         // 当一个进程被设置为前台执行时，当前tsh应该等待该子进程
45         // 结束
46         waitfg(job->pid);
47     } else {
48         puts("do_bgfg: Internal error");
49         exit(0);
50     }
51 }
```

这段代码的作用是根据传入的参数将作业 (job) 设置为后台执行 (bg) 或前台执行 (fg)。它接受一个参数数组 `argv`，其中 `argv[0]` 是指定的命令 ("bg" 或 "fg")，`argv[1]` 是作业的标识符（可以是作业ID以 "%" 开头的字符串，或者是进程ID）。

代码的执行逻辑如下：

1. 首先检查是否提供了正确的参数，如果没有提供则打印错误消息并返回。
2. 解析参数，判断是根据作业ID还是进程ID来获取作业对象。
  - 如果参数以 "%" 开头，则解析为作业ID，通过 `getjobjid` 函数获取作业对象。
  - 否则，解析为进程ID，通过 `getjobpid` 函数获取作业对象。
  - 如果解析失败或找不到对应的作业，则打印错误消息并返回。
3. 根据命令 ("bg" 或 "fg") 执行相应的操作：
  - 如果是 "bg"，将作业状态设置为后台执行 (`BG`)，然后发送 `SIGCONT` 信号给作业的进程组，使其继续执行。

- 如果是 "fg"，将作业状态设置为前台执行（FG），然后发送 SIGCONT 信号给作业的进程组，使其继续执行。同时，等待前台作业执行完毕。
4. 如果命令既不是 "bg" 也不是 "fg"，则打印错误消息并退出。

## 4.waitfg()函数

### 函数功能

等待一个前台作业结束，或者说是阻塞一个前台的进程直到这个进程变为后台进程

### 函数原型

```
void waitfg(pid_t pid), 参数为进程ID
```

### 函数思路

因为回收子进程交给了 sigchld\_handler 来做，所以 waitfg 只要用 sleep 写一个忙等待来等到前台进程结束。

- 调用自带的 fgpid 函数获得前台进程号，判断当前进程号是否是前台进程（因为不需要负责子进程回收）
- 如果前台进程号变化，就说明结束了，就跳出循环，否则忙等待。

```

1 void waitfg(pid_t pid)
2 {
3     struct job_t *job = getjobpid(jobs, pid);
4     if(!job) return; // 如果没有找到对应的作业结构体，则返回
5
6     // 如果当前子进程的状态没有发生改变，则tsh继续休眠
7     while(job->state == FG)
8         // 使用sleep的这段代码会比较慢，最好使用sigsuspend
9         sleep(1); // 休眠1秒
10
11    return;
12 }
```

函数首先根据传入的进程ID (pid) 通过 getjobpid 函数获取对应的作业结构体。如果没有找到对应的作业结构体，则直接返回。

然后，函数使用一个循环来等待前台作业的状态发生变化。循环条件是 `job->state == FG`，即只要作业状态仍然是前台运行状态 (FG)，就会继续循环。

在循环中，使用 `sleep(1)` 函数让 tsh (shell) 休眠1秒。这段代码的作用是暂时阻塞 tsh，直到前台作业的状态发生变化，避免 tsh 忙等待。然而，使用 `sleep` 函数的效率相对较低，更好的实现方式是使用 `sigsuspend` 函数。

## 5.sigchld\_handler()函数

### 函数功能

处理 SIGCHLD 信号

### 函数原型

```
void sigchld_handler(int sig), 参数为信号类型
```

### 函数思路

一个子进程结束的主要3种原因：

- 正常运行结束
- 收到SIGINT终止
- 收到SIGTSTP停止

三种情况有不同的处理：

- 正常结束的进程需要 delete。不然 jobs 的时候，前台程序太多了
- 终止的进程，是不会再启动了，也需要 delete，并且输出信息
- 停止的，可能会再启动，所以只修改其 state

```
1 void sigchld_handler(int sig)
2 {
3     int status, jid;
4     pid_t pid;
5     struct job_t *job;
6
7     if(verbose)
8         puts("sigchld_handler: entering"); // 如果 verbose
    为真，则打印进入函数的消息
9
10    /*
11     * 以非阻塞方式等待所有子进程
12     * waitpid 参数3：
13     *     1.      0      : 执行waitpid时， 只有在子进程 **终止**
    时才会返回。
14     *     2. WNOHANG   : 若子进程仍然在运行，则返回0。

```

```
15             注意只有设置了这个标志，waitpid才有可能返回0
16         3. WUNTRACED：如果子进程由于传递信号而停止，则马上返
17             回。
18             只有设置了这个标志，waitpid返回时，其
19             WIFSTOPPED(status)才有可能返回true
20             */
21             while((pid = waitpid(-1, &status, WNOHANG |
22             WUNTRACED)) > 0){
23
24                 // 如果当前这个子进程的job已经删除了，则表示有错误发生
25                 if((job = getjobpid(jobs, pid)) == NULL){
26                     printf("Lost track of (%d)\n", pid); // 打印
27                     丢失子进程的错误消息
28                     return;
29
30
31                 jid = job->jid;
32
33                 // 接下来判断三种状态
34                 // 如果这个子进程收到了一个暂停信号（还没退出）
35                 if(WIFSTOPPED(status)){
36                     printf("Job [%d] (%d) stopped by signal
37                     %d\n", jid, job->pid, WSTOPSIG(status)); // 打印作业被暂停
38                     的消息和信号
39                     job->state = ST; // 状态设为挂起
40
41
42                 // 如果子进程通过调用 exit 或者一个返回 (return) 正常
43                 终止
44                 else if(WIFEXITED(status)){
45                     if(deletejob(jobs, pid)){ // 删除作业
46                         if(verbose){
47                             printf("sigchld_handler: Job [%d]
48                             (%d) deleted\n", jid, pid); // 如果verbose为真，则打印删除
49                             作业的消息
50
51                             printf("sigchld_handler: Job [%d]
52                             (%d) terminates OK (status %d)\n", jid, pid,
53                             WEXITSTATUS(status)); // 打印作业终止的消息和状态
54
55                         }
56
57                     }
58
59                 // 如果子进程是因为一个未被捕获的信号终止的，例如
60                 SIGKILL
```

```

45     else {
46         if(deletejob(jobs, pid)){ // 清除作业
47             if(verbose)
48                 printf("sigchld_handler: Job [%d]
(%d) deleted\n", jid, pid); // 如果verbose为真，则打印删除
作业的消息
49             }
50             printf("Job [%d] (%d) terminated by signal
%d\n", jid, pid, WTERMSIG(status)); // 返回导致作业终止的信
号的数量
51         }
52     }
53
54     if(verbose)
55         puts("sigchld_handler: exiting"); // 如果verbose
为真，则打印退出函数的消息
56
57     return;
58 }
59

```

该函数用于处理子进程终止的SIGCHLD信号。函数通过调用 `waitpid` 函数以非阻塞方式等待所有子进程的状态变化。根据子进程的状态，函数执行相应的操作。如果 `verbose` 为真，则会在函数的不同位置打印一些调试信息。

函数首先通过 `waitpid` 函数获取终止的子进程的进程ID和状态。然后根据子进程的状态进行判断：

- 如果子进程收到了一个暂停信号（但还没有退出），则打印作业被暂停的消息和信号，并将作业状态设为挂起（ST）。
- 如果子进程通过调用 `exit` 或者一个返回（`return`）正常终止，则删除该作业，并在 `verbose` 为真时打印删除作业的消息和作业终止的消息以及状态。
- 如果子进程是因为一个未被捕获的信号终止的（例如SIGKILL），则删除该作业，并打印作业被终止的消息和终止的信号。

如果在等待子进程状态变化的过程中，没有子进程终止，则 `waitpid` 函数返回 0。如果发生错误导致无法获取子进程的作业结构体，则会打印错误消息。

最后，如果 `verbose` 为真，则打印退出函数的消息。

# sigint\_handler()函数

## 函数功能

捕获SIGINT信号

## 函数原型

```
void sigchld_handler(int sig), 参数为信号类型
```

## 函数思路

1. 调用函数fgpid返回前台进程pid
2. 如果当前进程pid不为0，那么调用kill函数发送SIGINT信号给前台进程组
3. 在2中调用kill函数如果返回值为-1表示进程不存在。输出error

```
1 void sigint_handler(int sig)
2 {
3     if(verbose)
4         puts("sigint_handler: entering"); // 如果verbose
5         // 为真，则打印进入函数的消息
6
6     pid_t pid = fgpid(jobs); // 获取当前前台进程组的进程ID
7
8     if(pid){
9         // 发送SIGINT给前台进程组里的所有进程
10        // 需要注意的是，前台进程组内的进程除了当前前台进程以外，
11        // 还包括前台进程的子进程。
12        // 最多只能存在一个前台进程，但前台进程组内可以存在多个进
13        // 程
14        if(kill(-pid, SIGINT) < 0) // 向前台进程组发送
15        // SIGINT信号
16        unix_error("kill (sigint) error"); // 发送信
17        // 号失败时打印错误消息
18
19        if(verbose){
20            printf("sigint_handler: Job (%d) killed\n",
21 pid); // 如果verbose为真，则打印被终止的进程ID
22        }
23    }
24
25    if(verbose)
```

```
21         puts("sigint_handler: exiting"); // 如果verbose为  
真，则打印退出函数的消息  
22  
23     return;  
24 }
```

该函数的作用是在收到SIGINT信号时，向前台进程组发送SIGINT信号，以终止正在运行的进程。函数首先获取当前前台进程组的进程ID，然后使用kill函数向该进程组发送SIGINT信号。如果发送失败，则会打印错误消息。如果verbose为真，则会在函数的不同位置打印一些调试信息。

## sigtstp\_handler()函数

### 函数功能

同sigint\_handler差不多，捕获SIGTSTP信号

- SIGTSTP的作用：SIGTSTP信号默认行为是停止直到下一个 SIGCONT，是来自终端的停止信号，在键盘上输入 CTR+Z会导致一个 SIGTSTP信号被发送到外壳。外壳捕获该信号，然后发送SIGTSTP信号到这个前台进程组中的每个进程。在默认情况下，结果是停止或挂起前台作业。

### 函数原型

```
void sigtstp_handler(int sig), 参数为信号类型
```

### 函数思路

- 找到前台的进程
- 改变进程的状态，并发送信号给其进程组

```
1 void sigtstp_handler(int sig)  
2 {  
3     if(verbose)  
4         puts("sigtstp_handler: entering"); // 如果verbose  
为真，则打印进入函数的消息  
5  
6     pid_t pid = fgpid(jobs); // 获取当前前台进程组的进程ID  
7     struct job_t *job = getjobpid(jobs, pid); // 根据进程  
ID获取相应的作业结构体  
8  
9     if(pid){
```

```

10         if(kill(-pid, SIGTSTP) < 0) // 向前台进程组发送
11             SIGTSTP信号
12
13         unix_error("kill (tstp) error"); // 发送信号失
14     败时打印错误消息
15
16     }
17
18     if(verbose)
19         printf("sigstp_handler: Job [%d] (%d)
20 stopped\n", job->jid, pid); // 如果verbose为真，则打印被停
21 止的作业ID和进程ID
22     }
23

```

该函数的作用是在收到SIGTSTP信号时，向前台进程组发送SIGTSTP信号，以停止正在运行的进程。函数首先获取当前前台进程组的进程ID，并通过该ID获取相应的作业结构体。然后使用kill函数向该进程组发送SIGTSTP信号。如果发送失败，则会打印错误消息。如果verbose为真，则会在函数的不同位置打印一些调试信息。

注意：使用kill函数，如果pid小于零才会发送信号sig给进程组中的每个进程，因此这里使用-pid。

## 实验测试

测试文件的符号和命令的定义

### 符号

- 空格：分隔指令作用
- &：如果命令以&结尾，表示该作业在后台运行
- #：直接打印#后一行的文本内容
- %：后接一个整数，表示job的ID号

### 命令

- jobs：列出正在运行和停止的后台作业

- `bg <job>`: 将停止的后台作业更改为正在运行的后台作业
- `fg <job>`: 将已停止或正在运行的后台作业更改为前台正在运行的作业
- `kill`: 终止一个作业

### | 用户程序

- myint程序：函数睡眠，使程序睡眠n秒，运行结束后不会自动退出，并会检测系统错误；
- myspin程序：函数睡眠，使程序睡眠n秒，在睡眠结束后就自动退出，不检测系统错误；
- mysplit程序：函数睡眠，使程序睡眠n秒，创建一个子进程进行睡眠，然后父进程等待子进程正常睡眠n秒后，继续运行；
- mystop程序：让进程暂定n秒，并发送信号

## 1.trace01->正确终止EOF

```

1 #trace01.txt - Properly terminate on EOF.
2 #第一关调用了linux命令close关闭文件并wait等待，所以不需要我们做任何事
3 CLOSE
4 WAIT

```

#可以理解为//，不需要我们解析。

```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#

```

| 第一关调用了linux命令close关闭文件并wait等待，在EOF上正常终止，所以不需要我们做任何事

要求在读取EOF信号时退出Shell，在初始代码中该功能已经实现

```
1     if ((fgets(cmdline, MAXLINE, stdin) == NULL) &&
2         ferror(stdin))
3         app_error("fgets error");
4     if (feof(stdin)) { /* End of file (ctrl-d) */
5         fflush(stdout);
6         exit(0);
7 }
```

## 2.trace 02->实现内置的quit

```
1 # trace02.txt - Process builtin quit command.
2 #需要在函数中实现quit
3 #需我们针对输入的命令quit退出shell进程，我们需要解析cmdline，判断
4 #是不是“quit”字符串，是就退出
5 quit
6 WAIT
```

可以看到无法正常终止，因为tsh的quit内置命令还未编写，所以不能正常退出。因此需要我们实现终止命令(quit)

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shLab-handout$ make test02
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

针对输入的命令quit退出shell进程，我们需要解析cmdline（输入的命令），判断是不是“quit”字符串，是就退出

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shLab-handout$ make test02
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shLab-handout$ make rtest02
./sdriver.pl -t trace02.txt -s ./tshref -a "-p"
#
# trace02.txt - Process builtin quit command.
```

tsh实验现象和tshref一致，结果正确

测试内置的quit命令，课本示例中也已经进行实现，说明builtin\_cmd()函数的部分实现是正确的

```
1 // quit command
2 if (!strcmp(argv[0], "quit"))
3     exit(0);
```

### 3.trace 03->运行一个前台job

```
1 # trace03.txt - Run a foreground job.
2 #运行一个在路径为/bin/echo的执行文件echo
3 #需要实现eval函数中非内执行命令的部分
4 #即eval发现/bin/echo不是内置命令，查询路径找到了echo程序并且在前
5 #台执行，输出了其后面的一句话。
6 /bin/echo tsh> quit
7 quit
```

这里解释一下/bin/echo：

eval函数先通过builtin\_cmd查询cmdline是不是内置命令如quit，如果是则当前进程执行命令

如果不是则创建一个子进程，在子进程中调用execve()函数通过 argv[0]来寻找路径，并在子进程中运行路径中的可执行文件，如果找不到则说明命令为无效命令，输出命令无效，并用exit(0)结束该子进程

/bin/echo就是打开bin目录下的echo文件，echo可以理解为将其后面的内容当作字符串输出

所以第三关的任务是：

- 首先是/bin/echo tsh> quit 意思是打开 bin 目录下的 echo 可执行文件，在 foreground 开启一个子进程运行它（因为末尾没有&符号，如果有，就是在 background 运行）
- 运行 echo 这个进程的过程中，通过 tsh>quit 命令，调用 tsh 并执行内置命令 quit，退出 echo 这个子进程
- 最后在 tsh 中执行内置命令 quit，退出 tsh 进程，回到我们的终端。

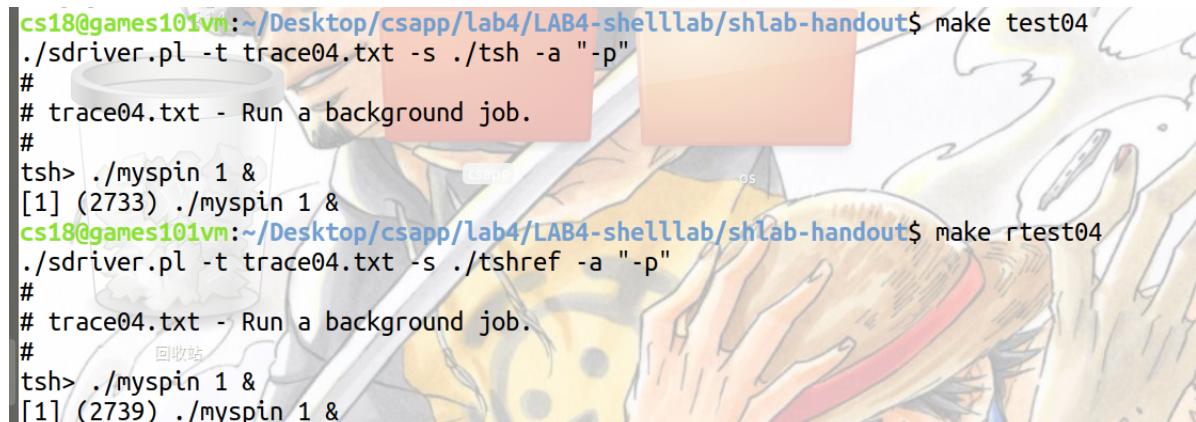
```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlabb-handout$ make k test03
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlabb-handout$ make r test03
./sdriver.pl -t trace03.txt -s ./tshref -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
```

tsh实验现象和tshref一致，结果正确

## 4.trace 04->运行后台job

```
1 # trace04.txt - Run a background job.  
2 #先在前台执行echo文件，等待程序执行完毕回收子进程  
3 #这里需要注意为了避免addjob和deletejob竞争，我们需要添加阻塞  
4 # &代表是一个后台程序  
5 # 然后在后台执行myspin程序并且输出任务ID (jid)，程序ID (pid) 命  
令行 ( cmdline )  
6 /bin/echo -e tsh> ./myspin 1 \046  
7 ./myspin 1 &
```

可以看到，这个测试文件要求运行一个后台工作，这将涉及到eval函数、waitfg函数、sigchld\_handler函数。



```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shellab/shlab-handout$ make test04  
.sdriver.pl -t trace04.txt -s ./tsh -a "-p"  
#  
# trace04.txt - Run a background job.  
#  
tsh> ./myspin 1 &  
[1] (2733) ./myspin 1 &  

```

tsh实验现象和tshref一致，结果正确

trace03为测试前台运行quit，trace04为测试后台运行myspin程序。

主要需要解析命令行末尾的&，并针对前后台运行进行不同的处理。其中parseline函数已经帮助解析了命令行末尾&，所以只需要对前后台程序进行不同处理即可。

如前所述，前台则需等待执行完毕，后台则只需要将其添加到jobs即可。

首先在eval函数中实现添加作业的代码以及前后台程序处理。特别注意这里对SIGCHLD信号在适当的地方进行了阻塞和解除阻塞。另外进行阻塞所使用的函数是包裹了错误处理的系统调用。

```
1     Sigemptyset(&mask);  
2     Sigaddset(&mask, SIGCHLD);  
3  
4     if (!builtin_cmd(argv)) {
```

```

5         sigprocmask(SIG_BLOCK, &mask, &prev); // block
6         SIGCHLD
7
8     if ((pid = fork()) == 0) { /* Child runs user
9      job */
10        sigprocmask(SIG_UNBLOCK, &prev, NULL); // unblock SIGCHLD
11        if (execve(argv[0], argv, environ) < 0) {
12            printf("%s: Command not found.\n",
13                   argv[0]);
14            exit(0);
15        }
16    }
17    addjob(jobs, pid, bg ? BG : FG, cmdline);
18
19    sigprocmask(SIG_SETMASK, &prev, NULL); // unblock SIGCHLD

```

对于后台程序按照给出的对照程序 (tshref) 输出其相应的任务号，PID以及命令行。

对于前台程序处理则依赖于sigchld\_handler信号处理程序，接收到其终止信号时将其移出jobs数组。于是可以通过判断fgpid函数返回当前前台程序PID是否等于子进程的PID来判断是否运行完毕。

```

1 // code in evalvoid sigchld_handler(int sig)
2 {
3     int old_errno = errno;
4
5     pid_t pid;
6     int status;
7
8     while ((pid = waitpid(-1, &status, WNOHANG |
9 WUNTRACED)) > 0) {
10        if (WIFEXITED(status)) {
11            deletejob(jobs, pid);
12        }
13    }
14    if (errno != ECHILD)
15        unix_error("waitpid_error");

```

```

16     errno = old_errno;
17     return;
18 }
19             /* Parent waits for foreground job to
20              terminate */
21             if (!bg) // foreground
22                 waitfg(pid);
23             else // background
24                 printf("[%d] (%d) %s", pid2jid(pid), pid,
25 cmdline);
26 // waitfg function
27 void waitfg(pid_t pid)
28 {
29     while (pid == fgpid(jobs))
30         sleep(0);
31     return;
32 }
```

具体到SIGCHLD的处理，需要在其中使用waitpid回收所有的终止的子进程。其中WNOHANG | WUNTRACED代表立即返回，如果有子进程停止或终止则返回其PID，用while循环包起来确保一次尽可能将所有已经终止或停止的子进程回收

```

1 void sigchld_handler(int sig)
2 {
3     int old_errno = errno;
4
5     pid_t pid;
6     int status;
7
8     while ((pid = waitpid(-1, &status, WNOHANG |
9 WUNTRACED)) > 0) {
10         if (WIFEXITED(status)) {
11             deletejob(jobs, pid);
12         }
13     }
14     errno = old_errno;
15     return;
16 }
```

## ace 05->处理jobs内置命令

```
1 # trace05.txt - Process jobs builtin command. 需要实现jobs命令
2 # 分别运行了前台echo、后台myspin、前台echo、后台myspin
3 # 然后需要实现一个内置命令job，功能是显示目前任务列表中的所有任务的所有属性，这里可以利用listjob函数
4 /bin/echo -e tsh> ./myspin 2 \046
5 ./myspin 2 &
6
7 /bin/echo -e tsh> ./myspin 3 \046
8 ./myspin 3 &
9
10 /bin/echo tsh> jobs
11 jobs
```

分别运行了前台echo、后台myspin、前台echo、后台myspin，然后需要实现一个内置命令job，功能是显示目前任务列表中的所有任务的所有属性

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test05
./sdriver.pl -t trace05.txt -s ./tsh -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (3295) ./myspin 2 &
tsh> ./myspin 3 &
[2] (3297) ./myspin 3 &
tsh> jobs
[1] (3295) Running ./myspin 2 &
[2] (3297) Running ./myspin 3 &
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest05
./sdriver.pl -t trace05.txt -s ./tshref -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (3305) ./myspin 2 &
tsh> ./myspin 3 &
[2] (3307) ./myspin 3 &
tsh> jobs
[1] (3305) Running ./myspin 2 &
[2] (3307) Running ./myspin 3 &
```

tsh实验现象和tshref一致，结果正确

trace05为实现jobs功能，在完成了前面的基本的作业控制后非常简单，只需要在builtin\_cmd中调用起始代码已经提供了的listjobs函数即可

```
1 // jobs command
2 if (!strcmp(argv[0], "jobs")) {
3     listjobs(jobs);
4     return 1;
5 }
```

## 6.trace 06->将SIGINT转发到前台作业

```
1 # trace06.txt - Forward SIGINT to foreground job.
2 # 如果接收到了中断信号SIGINT(即CTRL_C)那么结束前台进程
3 /bin/echo -e tsh> ./myspin 4
4 ./myspin 4
5
6 SLEEP 2
7 INT
```

接收到了中断信号SIGINT(即CTRL\_C)那么结束前台进程

```
cs18@games101vm:~/Desktop/csapp/Lab4/shelllab/shlab-handout$ make test06
./sdriver.pl -t trace06.txt -s ./tsh -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (3372) terminated by signal 2
cs18@games101vm:~/Desktop/csapp/Lab4/shelllab/shlab-handout$ make rtest06
./sdriver.pl -t trace06.txt -s ./tshref -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (3431) terminated by signal 2
```

tsh实验现象和tshref一致，结果正确

## 7.trace 07->仅将SIGINT转发给前台作业

```
1 # trace07.txt - Forward SIGINT only to foreground job.
2 #第七关是检测你之前是否是正确完成了所有的函数，有没有投机取巧
3 /bin/echo -e tsh> ./myspin 4 \046
4 ./myspin 4 &
5
6 /bin/echo -e tsh> ./myspin 5
7 ./myspin 5
8
9 SLEEP 2
10 INT
11
12 /bin/echo tsh> jobs
13 jobs
```

只将SIGINT转发给前台作业。这里的命令行其实根据前面的就很好理解了，就是给出两个作业，一个在前台工作，另一个在后台工作，接下来传递SIGINT指令，然后调用内置指令jobs来查看此时的工作信息，来对比出是不是只将SIGINT转发给前台作业。



```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test07
./sdriver.pl -t trace07.txt -s ./tsh -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (3484) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3486) terminated by signal 2
tsh> jobs
[1] (3484) Running ./myspin 4 &
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest07
./sdriver.pl -t trace07.txt -s ./tshref -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (3493) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3495) terminated by signal 2
tsh> jobs
[1] (3493) Running ./myspin 4 &
```

tsh实验现象和tshref一致，结果正确

## 8.trace 08->仅将SIGTSTP转发给前台作业

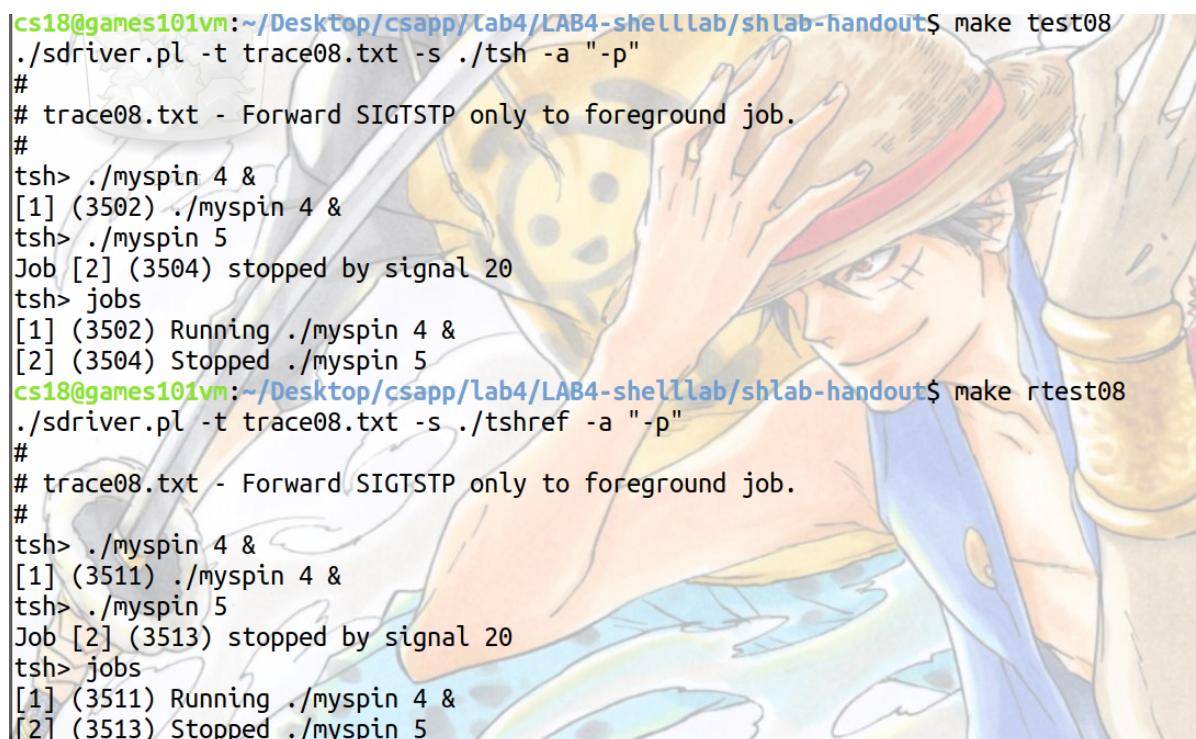
```
1 # trace08.txt - Forward SIGTSTP only to foreground job.
2 #这里有空命令行，需要在eval里加一句判断是否是空（加在eval里）
```

```

3 #当接收到了TSTP中断信号（即CTRL_Z），将前台进程挂起，然后输出被挂
4 /bin/echo -e tsh> ./myspin 4 \046
5 ./myspin 4 &
6
7 /bin/echo -e tsh> ./myspin 5
8 ./myspin 5
9
10 SLEEP 2
11 TSTP
12
13 /bin/echo tsh> jobs
14 jobs

```

将SIGTSTP转发给前台作业。根据这个信号的作用，也就是该进程会停止直到下一个SIGCONT也就是挂起，让别的程序继续运行。这里也就是运行了后台程序，然后使用jobs来打印出进程的信息。



```

cs18@games101vm:~/Desktop/csapp/Lab4/LAB4-shelllab/shlab-handout$ make test08
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (3502) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3504) stopped by signal 20
tsh> jobs
[1] (3502) Running ./myspin 4 &
[2] (3504) Stopped ./myspin 5
cs18@games101vm:~/Desktop/csapp/Lab4/LAB4-shelllab/shlab-handout$ make rtest08
./sdriver.pl -t trace08.txt -s ./tshref -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (3511) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3513) stopped by signal 20
tsh> jobs
[1] (3511) Running ./myspin 4 &
[2] (3513) Stopped ./myspin 5

```

tsh实验现象和tshref一致，结果正确

这三个trace是测试SIGINT和SIGSTOP能否被正确处理，值得注意的是，前台程序收到这两个信号都应该将其发送给其所在组的所有程序，而不是本身。

具体发送于是sigint和sigstop的任务非常简单，即收到信号后转手给所在的整个组发一下信号，给整个组发信号只需要给kill的pid为负数即可。

```

1 void sigint_handler(int sig)

```

```

2 {
3     int olderrno = errno;
4
5     // get the foreground job pid
6     pid_t fg_pid;
7     fg_pid = fgpid(jobs);
8
9     // send the signal to the group in the foreground
10    kill(-fg_pid, sig);
11
12    errno = olderrno;
13    return;
14}
15 void sigstp_handler(int sig)
16 {
17     int olderrno = errno;
18
19     // get the foreground job pid
20     pid_t fg_pid;
21     fg_pid = fgpid(jobs);
22
23     // send the signal to the group in the foreground
24     kill(-fg_pid, sig);
25
26     errno = olderrno;
27     return;
28 }

```

具体处理这两个的信号在sigchld\_handler里，sigchld\_handler里收到了进程终止或停止的消息后给出对应的输出然后改变其状态，对于终止的进程就在jobs里将其删除，对于停止的进程则设置其state为ST。值得注意的是在信号处理程序里不可以使用异步信号不安全的printf，这里使用的是csapp.h里给出的Sio包。

```

1 while ((pid = waitpid(-1, &status, WNOHANG |
2 WUNTRACED)) > 0) {
3     if (WIFEXITED(status)) {
4         deletejob(jobs, pid);
5     }
6     if (WIFSIGNALED(status)) { // terminated by ctrl-
7
8         sio_puts("Job [");

```

```

7         Sio_putl(pid2jid(pid));
8         Sio_puts("] ());
9         Sio_putl(pid);
10        Sio_puts(") terminated by signal ");
11        Sio_putl(WTERMSIG(status));
12        Sio_puts("\n");
13        deletejob(jobs, pid);
14    }
15    if (WIFSTOPPED(status)) { // stopped by ctrl-z
16        Sio_puts("Job [");
17        Sio_putl(pid2jid(pid));
18        Sio_puts("] ());
19        Sio_putl(pid);
20        Sio_puts(") stopped by signal ");
21        Sio_putl(WSTOPSIG(status));
22        Sio_puts("\n");
23        getjobpid(jobs, pid)->state = ST;
24    }
25 }
```

此外还有非常重要的一点就是，我们的shell程序本身是所有子进程的父进程，那么就会分配在同一个组里，终止子进程所在组会导致shell程序本身也被终止，这里的解决办法是给子进程设置一个单独的组，只需要添加在fork和exec之间。

```

1         if ((pid = fork()) == 0) { /* child runs user job
*/
2             setpgid(0, 0);
3             Sigprocmask(SIG_UNBLOCK, &prev, NULL); // unblock SIGCHLD
4             if (execve(argv[0], argv, environ) < 0) {
5                 printf("%s: Command not found.\n",
6                     argv[0]);
7                 exit(0);
8             }
}
```

## 9.trace 09->进程bg内置命令

```

1 # trace09.txt - Process bg builtin command
2 #九关创建了内置命令bg
```

```
3 # bg命令是将一个挂起的程序转到后台继续执行
4 /bin/echo -e tsh> ./myspin 4 \046
5 ./myspin 4 &
6
7 /bin/echo -e tsh> ./myspin 5
8 ./myspin 5
9
10 SLEEP 2
11 TSTP
12
13 /bin/echo tsh> jobs
14 jobs
15
16 /bin/echo tsh> bg %2
17 bg %2
18
19 /bin/echo tsh> jobs
20 jobs
```

在第八关的测试文件之上一个更加完整的测试，这里也就是在停止后，输出进程信息之后，使用bg命令来唤醒进程2，也就是刚才被挂起的程序，接下来继续使用Jobs命令来输出结果。

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (3520) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3522) stopped by signal 20
tsh> jobs
[1] (3520) Running ./myspin 4 &
[2] (3522) Stopped ./myspin 5
tsh> bg %2 回收站
[2] (3522) ./myspin 5
tsh> jobs
[1] (3520) Running ./myspin 4 &
[2] (3522) Running ./myspin 5
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest09
./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (3531) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3533) stopped by signal 20
tsh> jobs
[1] (3531) Running ./myspin 4 &
[2] (3533) Stopped ./myspin 5
tsh> bg %2
[2] (3533) ./myspin 5
tsh> jobs
[1] (3531) Running ./myspin 4 &
[2] (3533) Running ./myspin 5
```

tsh实验现象和tshref一致，结果正确

## 10.trace 10->进程fg内置命令

```
1 # trace10.txt - Process fg builtin command.
2 #十关创建了内置命令 fg
3 # fg命令是将一个挂起的程序转到前台继续执行，或者将一个后台执行的程
4 #序转到前台执行
5 /bin/echo -e tsh> ./myspin 4 \046
6 ./myspin 4 &
7
8 SLEEP 1
9 /bin/echo tsh> fg %1
10 fg %1
11
12 SLEEP 1
13 TSTP
14
15 /bin/echo tsh> jobs
16 jobs
```

```
16  
17 /bin/echo tsh> fg %1  
18 fg %1  
19  
20 /bin/echo tsh> jobs  
21 jobs
```

将后台的进程更改为前台正在运行的程序。测试文中进程1根据&可以知道，进程1是一个后台进程。先使用fg命令将其转化为前台的一个程序，接下来停止进程1，然后打印出进程信息，这时候进程1应该是前台程序同时被挂起了，接下来使用fg命令使其继续运行，使用jobs来打印出进程信息



```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test10  
.sdriver.pl -t trace10.txt -s ./tsh -a "-p"  
#  
# trace10.txt - Process fg builtin command.  
#  
tsh> ./myspin 4 &  
[1] (3542) ./myspin 4 &  

```

tsh实验现象和tshref一致，结果正确

trace09是关于内置命令bg和fg的，其使用方法为

```
1 | $ fg/bg <job>
```

其中为响应任务的PID或JID，如果为JID则需%作为前缀。fg和bg都是发送SIGCONT信号来将相应任务重启。

首先在builtin\_cmd函数中判断是否为bg或fg，如果是则执行相应的操作。

```
1 // bg or fg command
2 if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg")) {
3     do_bgfg(argv);
4     return 1;
5 }
```

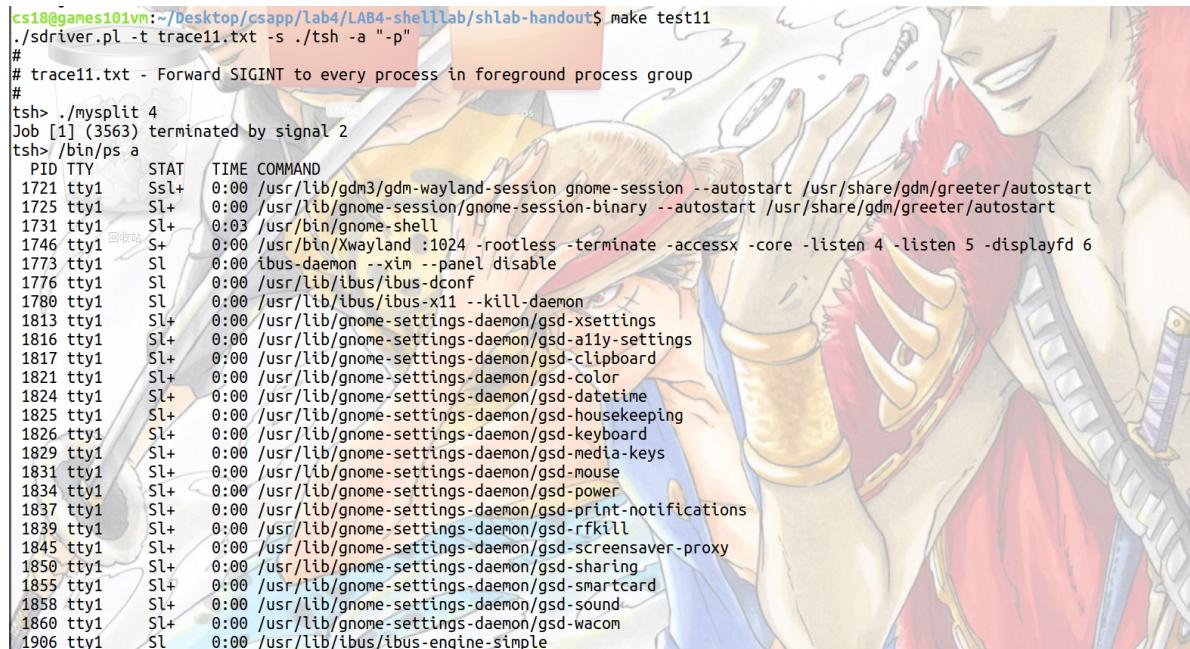
具体的do\_bgfg函数首先根据有无%判断是PID还是JID，然后取得该job指针，然后给其所在进程组发送SIGCONT，最后根据其是fg还是bg来做出与eval中类似的行为。

```
1 void do_bgfg(char** argv)
2 {
3     struct job_t* job;
4     char* id = argv[1];
5     if (id[0] == '%') { // jid
6         job = getjobjid(jobs, atoi(id + 1));
7     }
8     else { // pid
9         job = getjobpid(jobs, atoi(id));
10    }
11
12    kill(-(job->pid), SIGCONT);
13
14    if (!strcmp(argv[0], "fg")) { // fg command
15        job->state = FG;
16        // wait for the job to terminate
17        waitfg(job->pid);
18    }
19    else { // bg command
20        job->state = BG;
21        printf("[%d] (%d) %s", pid2jid(job->pid), job-
22            >pid, job->cmdline);
23    }
24
25 }
```

# 11.trace 11->将SIGINT转发给前台进程组中的每个进程

```
1 # trace11.txt - Forward SIGINT to every process in
2   foreground process group
3
4 #不需要修改
5 /bin/echo -e tsh> ./mysplit 4
6 ./mysplit 4
7
8
9 /bin/echo tsh> /bin/ps a
10 /bin/ps a
```

将SIGINT发给前台进程组中的每个进程。ps -a 显示所有进程，这里是有两个进程的，mysplit创建了一个子进程，接下来发送指令SIGINT，所以进程组中的所有进程都应该停止，接下来调用pi来查看该进程组中的每个进程是否都停止了。



```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test11
./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] (3563) terminated by signal 2
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
1721 tty1    Ssl+  0:00 /usr/lib/gdm3/gdm-wayland-session gnome-session --autostart /usr/share/gdm/greeter/autostart
1725 tty1    Sl+   0:00 /usr/lib/gnome-session/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
1731 tty1    Sl+   0:03 /usr/bin/gnome-shell
1746 tty1    S+    0:00 /usr/Xwayland :1024 -rootless -terminate -accessx -core -listen 4 -listen 5 -displayfd 6
1773 tty1    Sl    0:00 ibus-daemon --xim --panel disable
1776 tty1    Sl    0:00 /usr/lib/ibus/ibus-dconf
1780 tty1    Sl    0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1813 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1816 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1817 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1821 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-color
1824 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1825 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping
1826 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-keyboard
1829 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-media-keys
1831 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-mouse
1834 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-power
1837 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1839 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1845 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1850 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1855 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1858 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-sound
1860 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-wacom
1906 tty1    Sl    0:00 /usr/lib/ibus/ibus-engine-simple
```

```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest11
./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] (3576) terminated by signal 2
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
1721 tty1    Ssl+  0:00 /usr/lib/gdm3/gdm-wayland-session gnome-session --autostart /usr/share/gdm/greeter/autostart
1725 tty1    SL+   0:00 /usr/lib/gnome-session/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
1731 tty1    SL+   0:03 /usr/bin/gnome-shell
1746 tty1    S+    0:00 /usr/bin/Xwayland :1024 -rootless -terminate -accessx -core -listen 4 -listen 5 -displayfd 6
1773 tty1    SL    0:00 ibus-daemon --xim --panel disable
1776 tty1    SL    0:00 /usr/lib/ibus/ibus-dconf
1780 tty1    SL    0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1813 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1816 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1817 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1821 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-color
1824 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1825 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping
1826 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-keyboard
1829 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-media-keys
1831 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-mouse
1834 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-power
1837 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1839 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1845 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1850 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1855 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1858 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-sound
1860 tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-wacom
1906 tty1    SL    0:00 /usr/lib/ibus/ibus-engine-simple

```

tsh实验现象和tshref一致，结果正确

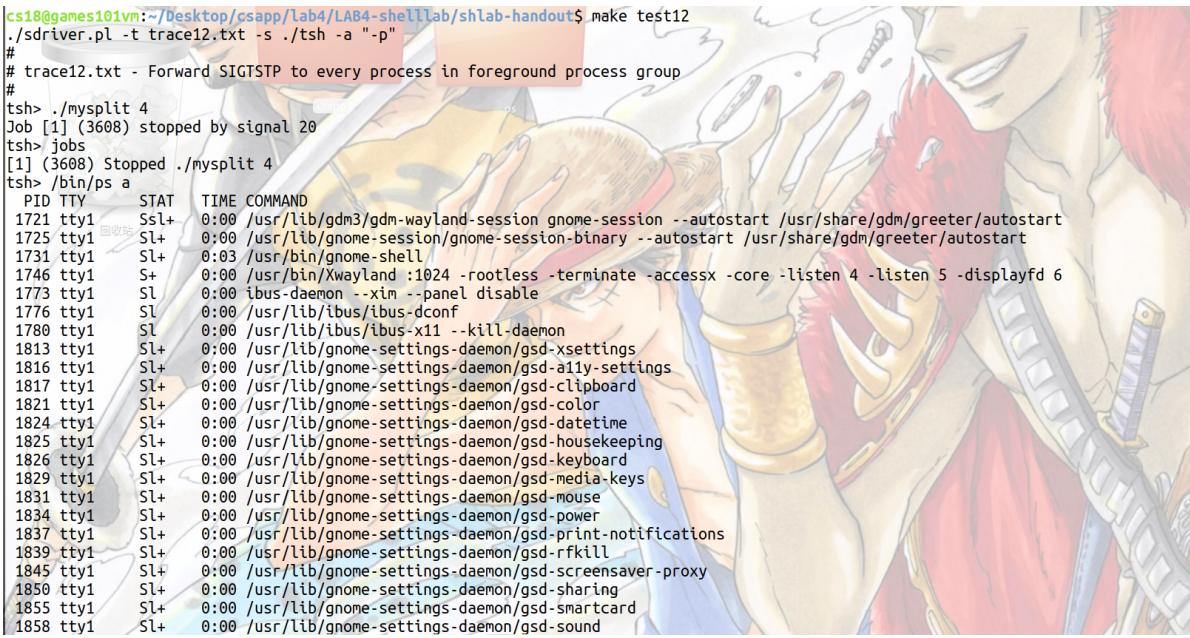
## 12.trace 12->将SIGTSTP转发到前台进程组中的每个进程

```

1 # trace12.txt - Forward SIGTSTP to every process in
2   foreground processgroup
3 #不需要修改
4 /bin/echo -e tsh> ./mysplit 4
5 ./mysplit 4
6
7 SLEEP 2
8
9 /bin/echo tsh> jobs
10 jobs
11
12 /bin/echo tsh> /bin/ps a
13 /bin/ps a

```

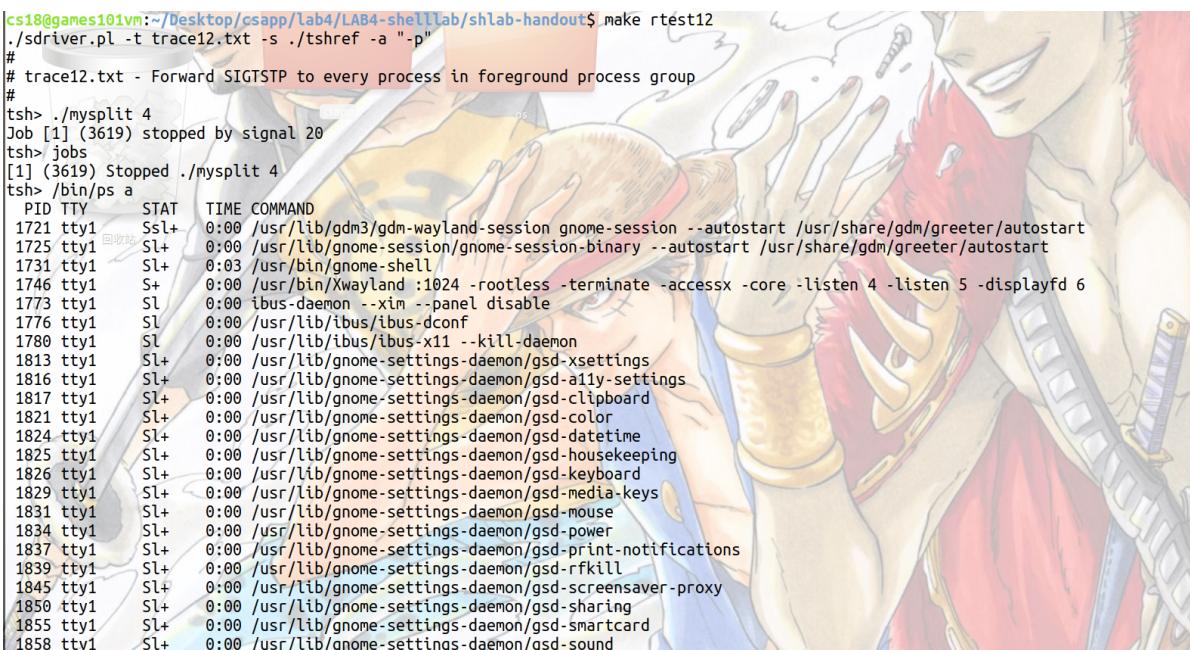
测试将SIGTSTP转发给前台进程组中的每个进程。与上一关相同，只需要相应的进程被挂起即可



```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test12
./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
#
# trace12.txt - Forward SIGSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (3608) stopped by signal 20
tsh> jobs
[1] (3608) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
1721 tty1    Ssl+  0:00 /usr/lib/gdm3/gdm-wayland-session gnome-session --autostart /usr/share/gdm/greeter/autostart
1725 tty1    Sl+   0:00 /usr/lib/gnome-session/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
1731 tty1    Sl+   0:03 /usr/bin/gnome-shell
1746 tty1    S+    0:00 /usr/bin/Xwayland :1024 -rootless -terminate -accessx -core -listen 4 -listen 5 -displayfd 6
1773 tty1    Sl    0:00 ibus-daemon --xim --panel disable
1776 tty1    Sl    0:00 /usr/lib/ibus/ibus-dconf
1780 tty1    Sl    0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1813 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1816 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1817 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1821 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-color
1824 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1825 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping
1826 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-keyboard
1829 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-media-keys
1831 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-mouse
1834 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-power
1837 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1839 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1845 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1850 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1855 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1858 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-sound

```



```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest12
./sdriver.pl -t trace12.txt -s ./tshref -a "-p"
#
# trace12.txt - Forward SIGSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (3619) stopped by signal 20
tsh> jobs
[1] (3619) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
1721 tty1    Ssl+  0:00 /usr/lib/gdm3/gdm-wayland-session gnome-session --autostart /usr/share/gdm/greeter/autostart
1725 tty1    Sl+   0:00 /usr/lib/gnome-session/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
1731 tty1    Sl+   0:03 /usr/bin/gnome-shell
1746 tty1    S+    0:00 /usr/bin/Xwayland :1024 -rootless -terminate -accessx -core -listen 4 -listen 5 -displayfd 6
1773 tty1    Sl    0:00 ibus-daemon --xim --panel disable
1776 tty1    Sl    0:00 /usr/lib/ibus/ibus-dconf
1780 tty1    Sl    0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1813 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1816 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1817 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1821 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-color
1824 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1825 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping
1826 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-keyboard
1829 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-media-keys
1831 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-mouse
1834 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-power
1837 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1839 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1845 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1850 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1855 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1858 tty1    Sl+   0:00 /usr/lib/gnome-settings-daemon/gsd-sound

```

tsh实验现象和tshref一致，结果正确

## 13.trace 13->重新启动进程组中的每个已停止的进程

```

1 # trace13.txt - Restart every stopped process in process
2 group
3 #不需要修改
4 /bin/echo -e tsh> ./mysplit 4
5 .mysplit 4
6
7 SLEEP 2
8 TSTP

```

```

9 /bin/echo tsh> jobs
10 jobs
11
12 /bin/echo tsh> /bin/ps a
13 /bin/ps a
14
15 /bin/echo tsh> fg %1
16 fg %1
17
18 /bin/echo tsh> /bin/ps a
19 /bin/ps a

```

测试重新启动进程组中的每个停止的进程。这里也就是使用fg来唤醒整个工作，中间使用ps -a来查看停止整个工作和唤醒整个工作的区别。



```

cs18@games10vm:~/Desktop/csapp/LAB4/LAB4-shelllab/shlab-handout$ make test13
./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (3629) stopped by signal 20
tsh> jobs
[1] (3629) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
1721 tty1    Ssl+  0:00 /usr/lib/gdm3/gdm-wayland-session gnome-session --autostart /usr/share/gdm/greeter/autostart
1725 tty1    Sl+   0:00 /usr/lib/gnome-session/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
1731 tty1    Sl+   0:03 /usr/bin/gnome-shell
1746 tty1    S+   0:00 /usr/bin/Xwayland :1024 -rootless -terminate -accessx -core -listen 4 -listen 5 -displayfd 6
1773 tty1    Sl   0:00 ibus-daemon --xim --panel disable
1776 tty1    Sl   0:00 /usr/lib/ibus/ibus-dconf
1780 tty1    Sl   0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1813 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1816 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1817 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1821 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-color
1824 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1825 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping
1826 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-keyboard
1829 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-media-keys
1831 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-mouse
1834 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-power
1837 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1839 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1845 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1850 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1855 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1858 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-sound
1860 tty1    Sl+  0:00 /usr/lib/gnome-settings-daemon/gsd-wacom
1906 tty1    Sl   0:00 /usr/lib/ibus/ibus-engine-simple

```

```

cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest13
./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (3642) stopped by signal 20
tsh> jobs
[1] (3642) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
1721  tty1    Ssl+  0:00 /usr/lib/gdm3/gdm-wayland-session gnome-session --autostart /usr/share/gdm/greeter/autostart
1725  tty1    Sl+   0:00 /usr/lib/gnome-session/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
1731  tty1    Sl+   0:03 /usr/bin/gnome-shell
1746  tty1    S+    0:00 /usr/bin/Xwayland :1024 -rootless -terminate -accessx -core -listen 4 -listen 5 -displayfd 6
1773  tty1    S     0:00 ibus-daemon --xim --panel disable
1776  tty1    SL    0:00 /usr/lib/ibus/ibus-dconf
1780  tty1    SL    0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1813  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1816  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1817  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1821  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-color
1824  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1825  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping
1826  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-keyboard
1829  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-media-keys
1831  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-mouse
1834  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-power
1837  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1839  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1845  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1850  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1855  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1858  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-sound
1860  tty1    SL+   0:00 /usr/lib/gnome-settings-daemon/gsd-wacom
1906  tty1    SL    0:00 /usr/lib/ibus/ibus-engine-simple

```

tsh实验现象和tshref一致，结果正确

这三个traces主要测试前面是否正确实现了SIGSTOP和SIGINT的处理程序，以及fg/bg的实现，如果没有将进程组中的所有程序一并处理这里可能会出现错误，前面的实现中已经处理了这些情况，这里不再赘述。

## 14.trace 14->简单的错误处理

```

1 # trace14.txt - Simple error handling
2 #本关主要是测试所有的命令，判断是否正确
3 /bin/echo tsh> ./bogus
4 ./bogus
5
6 /bin/echo -e tsh> ./myspin 4 \046
7 ./myspin 4 &
8
9 /bin/echo tsh> fg
10 fg
11
12 /bin/echo tsh> bg
13 bg
14
15 /bin/echo tsh> fg a
16 fg a
17
18 /bin/echo tsh> bg a
19 bg a

```

```
20
21 /bin/echo tsh> fg 9999999
22 fg 9999999
23
24 /bin/echo tsh> bg 9999999
25 bg 9999999
26
27 /bin/echo tsh> fg %2
28 fg %2
29
30 /bin/echo tsh> fg %1
31 fg %1
32
33 SLEEP 2
34 TSTP
35
36 /bin/echo tsh> bg %2
37 bg %2
38
39 /bin/echo tsh> bg %1
40 bg %1
41
42 /bin/echo tsh> jobs
43 jobs
```

测试简单的错误处理。这里的测试文件，也就是测试fg和bg后面的参数，我们知道fg和bg后面需要一个JID或者是PID，其中JID是加上%的整型数。其余参数都应该报错，或是没有参数也应该报错。接下来测试的功能，都在上面的关卡测试过了

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test14  
./sdriver.pl -t trace14.txt -s ./tsh -a "-p"  
#  
# trace14.txt - Simple error handling  
#  
tsh> ./bogus  
.bogus: command not found  
tsh> ./myspin 4 &  
[1] (3657) ./myspin 4 &  
tsh> fg  
fg command requires PID or %jobid argument  
tsh> bg  
bg command requires PID or %jobid argument  
tsh> fg a  
fg: argument must be a PID or %jobid  
tsh> bg a  
bg: argument must be a PID or %jobid  
tsh> fg 9999999  
(9999999): No such process  
tsh> bg 9999999  
(9999999): No such process  
tsh> fg %2  
%2: No such job  
tsh> fg %1  
Job [1] (3657) stopped by signal 20  
tsh> bg %2  
%2: No such job  
tsh> bg %1  
[1] (3657) ./myspin 4 &  
tsh> jobs  
[1] (3657) Running ./myspin 4 &
```

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest14  
./sdriver.pl -t trace14.txt -s ./tshref -a "-p"  
#  
# trace14.txt - Simple error handling  
#  
tsh> ./bogus  
.bogus: Command not found  
tsh> ./myspin 4 &  
[1] (3677) ./myspin 4 &  
tsh> fg  
fg command requires PID or %jobid argument  
tsh> bg  
bg command requires PID or %jobid argument  
tsh> fg a  
fg: argument must be a PID or %jobid  
tsh> bg a  
bg: argument must be a PID or %jobid  
tsh> fg 9999999  
(9999999): No such process  
tsh> bg 9999999  
(9999999): No such process  
tsh> fg %2  
%2: No such job  
tsh> fg %1  
Job [1] (3677) stopped by signal 20  
tsh> bg %2  
%2: No such job  
tsh> bg %1  
[1] (3677) ./myspin 4 &  
tsh> jobs  
[1] (3677) Running ./myspin 4 &
```

tsh实验现象和tshref一致，结果正确

这个测试需要对fg和bg的输入参数进行一些错误处理，例如没有参数或参数非数值或所选任务或进程不存在等。在do\_bgfg函数中进行相应处理即可。

```
1 void do_bgfg(char** argv)
2 {
3     struct job_t* job;
4     char* id = argv[1];
5
6     // no argument for bg/fg
7     if (id == NULL)
8     {
9         printf("%s command requires PID or %%jobid
argument\n", argv[0]);
10        return;
11    }
12
13    if (id[0] == '%') { // jid
14        if (!checkNum(id + 1)) {
15            printf("%s: argument must be a PID or
%%jobid\n", argv[0]);
16            return;
17        }
18        int jid = atoi(id + 1);
19        job = getjobjid(jobs, jid);
20        if (job == NULL) {
21            printf("%%d: No such job\n", jid);
22            return;
23        }
24    }
25    else {           // pid
26        if (!checkNum(id)) {
27            printf("%s: argument must be a PID or
%%jobid\n", argv[0]);
28            return;
29        }
30        int pid = atoi(id);
31        job = getjobpid(jobs, pid);
32        if (job == NULL) {
33            printf("(d): No such process\n", pid);
34            return;
35        }
36    }
}
```

```

37     kill(-(job->pid), SIGCONT);
38
39
40     if (!strcmp(argv[0], "fg")) { // fg command
41         job->state = FG;
42         // wait for the job to terminate
43         waitfg(job->pid);
44     }
45     else { // bg command
46         job->state = BG;
47         printf("[%d] (%d) %s", pid2jid(job->pid), job-
48             >pid, job->cmdline);
49     }
50
51     return;
52 }
```

## 15.trace 15->把它们放在一起

```

1 # trace15.txt - Putting it all together
2 #测试所有命令
3 /bin/echo tsh> ./bogus
4 ./bogus
5
6 /bin/echo tsh> ./myspin 10
7 ./myspin 10
8
9 SLEEP 2
10 INT
11
12 /bin/echo -e tsh> ./myspin 3 \046
13 ./myspin 3 &
14
15 /bin/echo -e tsh> ./myspin 4 \046
16 ./myspin 4 &
17
18 /bin/echo tsh> jobs
19 jobs
20
21 /bin/echo tsh> fg %1
22 fg %1
23
```

```
24 SLEEP 2
25 TSTP
26
27 /bin/echo tsh> jobs
28 jobs
29
30 /bin/echo tsh> bg %3
31 bg %3
32
33 /bin/echo tsh> bg %1
34 bg %1
35
36 /bin/echo tsh> jobs
37 jobs
38
39 /bin/echo tsh> fg %1
40 fg %1
41
42 /bin/echo tsh> quit
43 quit
```

测试了上述所有命令，如jobs,fg,bg,quit。

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: command not found
tsh> ./myspin 10
Job [1] (3701) terminated by signal 2
tsh> ./myspin 3 &
[1] (3703) ./myspin 3 &
tsh> ./myspin 4 &
[2] (3705) ./myspin 4 &
tsh> jobs
[1] (3703) Running ./myspin 3 &
[2] (3705) Running ./myspin 4 &
tsh> fg %1
Job [1] (3703) stopped by signal 20
tsh> jobs
[1] (3703) Stopped ./myspin 3 &
[2] (3705) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (3703) ./myspin 3 &
tsh> jobs
[1] (3703) Running ./myspin 3 &
[2] (3705) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest15
./sdriver.pl -t trace15.txt -s ./tshref -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (3721) terminated by signal 2
tsh> ./myspin 3 &
[1] (3723) ./myspin 3 &
tsh> ./myspin 4 &
[2] (3725) ./myspin 4 &
tsh> jobs
[1] (3723) Running ./myspin 3 &
[2] (3725) Running ./myspin 4 &
tsh> fg %1
Job [1] (3723) stopped by signal 20
tsh> jobs
[1] (3723) Stopped ./myspin 3 &
[2] (3725) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (3723) ./myspin 3 &
tsh> jobs
[1] (3723) Running ./myspin 3 &
[2] (3725) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

tsh实验现象和tshref一致，结果正确

## 16.trace 16->测试shell是否能够处理来自其他进程而不是终端的SIGTSTP和SIGINT信号

```
1 # trace16.txt - Tests whether the shell can handle
2   SIGTSTP and SIGINT
3
4
5 /bin/echo tsh> ./mystop 2
6 ./mystop 2
7
8 SLEEP 3
9
10 /bin/echo tsh> jobs
11 jobs
12
13 /bin/echo tsh> ./myint 2
14 ./myint 2
```

这个测试文件的具体含义就是，用户程序向job 2传送了中止信号，所以最后会输出进程2被中止的信息。同时，mystop需要自己停止才能给别的进程发送信号，所以中间也会出现进程1被中止的信息

```
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make test16
./sdriver.pl -t trace16.txt -s ./tsh -a "-p"
#
# trace16.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#   signals that come from other processes instead of the terminal.
#
tsh> ./mystop 2
Job [1] (3740) stopped by signal 20
tsh> jobs
[1] (3740) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (3743) terminated by signal 2
cs18@games101vm:~/Desktop/csapp/lab4/LAB4-shelllab/shlab-handout$ make rtest16
./sdriver.pl -t trace16.txt -s ./tshref -a "-p"
#
# trace16.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#   signals that come from other processes instead of the terminal.
#
tsh> ./mystop 2
Job [1] (3750) stopped by signal 20
tsh> jobs
[1] (3750) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (3753) terminated by signal 2
```

tsh实验现象和tshref一致，结果正确

对前面的程序进行的一些综合性测试，已经通过。

## 实验总结

总的来说不难的一个实验，关键在于要先理解整个框架中的代码，然后根据trace file渐进地完成程序。需要注意的地方实验讲义中的提示以及书本中都已经给出，这里不再赘述。需要强调的是SIGCHLD的信号处理程序需要处理未捕获的SIGTSTP和SIGINT信号。此外SIGINT/SIGTSTP和SIGCHLD的信号处理程序之间可能会有潜在的导致错误的冲突。（信号处理程序是可以被其他信号中断的，可以见trace16.txt）

此外，handler中的printf是异步不安全的。以及书本中虽然使用了sigsuspend来实行同步，但是为了简化程序，根据实验讲义的提示，使用了忙循环处理前台等待，并且将回收僵死进程任务交给了sigchld\_handler。这些都是本次实验中不足和可以修改的地方。

1. 对现代计算机系统进程的概念有了更深刻的了解
2. 掌握了linux 异常控制流和信号机制的基本原理和相关系统函数
3. 掌握了shell 的基本原理和简单功能的实现方法

#### 4. 深入了解 Linux 信号响应可能导致的并发冲突及解决方法